TEXAS INSTRUMENTS

TMS34082

Designer's Handbook

1991

Datapath VLSI Products

TMS34082
Designer's Handbook

2564007-9721 revision A
May 1991

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to or to discontinue any semiconductor product or service identified in this publication without notice. TI advises its customers to obtain the latest version of the relevant information to verify, before placing orders, that the information being relied upon is current.

TI warrants performance of its semiconductor products to current specifications in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed.

TI assumes no liability for TI applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Texas Instruments products are not intended for use in life-support appliances, devices, or systems. Use of a TI product in such applications without the written consent of the appropriate TI officer is prohibited.

Preface

Read This First

How to Use This Manual

The purpose of this user's guide is to provide the TI customer with information on the TMS34082 graphics floating-point processor. This manual can also be used as a reference guide for developing hardware or software applications. The following list summarizes the contents of the chapters and appendices in this user's guide.

Chapter 1 Overview of the TMS34082
Introduces the TMS34082, its key features, typical applications, and support tools available.

Chapter 2 Pinout and Pin Descriptions
Illustrates the TMS34082s package, identifies the interfaces that signals are associated with, and provides an explanation of each signal.

Chapter 3 Data Formats
Discusses the integer and floating-point operand formats accepted by the TMS34082.

Chapter 4 Architecture
Describes the architectural elements of the TMS34082. Includes the bus interfaces, sequence control, registers, internal floating-point unit core, and test logic.

Chapter 5 Coprocessor Mode
Describes using the TMS34082 as a coprocessor to the TMS34020, including the hardware interface, recommended configurations, and example programs with timing diagrams.

Chapter 6 Host-Independent Mode
Provides information on using the TMS34082 as a stand-alone processor or a coprocessor to another host.

Chapter 7 Internal Instructions
Shows how to use internal instructions in both coprocessor and host-independent mode. Explains the format and provides an alphabetical reference of the internal instruction set.

Chapter 8 External Instructions
Shows how to use external instructions in both coprocessor and host-independent mode. Explains the format and provides an alphabetical reference of the external instruction set.

Appendix A System Design Considerations
Provides recommendations on logic design, bypass capacitors, PWB design, and thermal considerations.

Appendix B TMS34082 Data Sheet
Contains the commercial data sheet for the TMS34082A.

Appendix C SMJ34082 Data Sheet
Contains the advance information military data sheet for the SMJ34082A.

Appendix D Maximizing Your MFLOPS with the TMS34082 and Motorola MC68030
Contains an application note on interfacing the TMS34082 (in host-independent mode) to the Motorola MC68030.

Appendix E A High-Performance Floating-Point Image Computing Workstation for Medical Applications
Contains an application note on an imaging system using a TMS34020 with four TMS34082 coprocessors.

Appendix F Parallel Signal and Matrix Processing with the TMS34082
Contains an application note outlining and analyzing a TMS34082-based parallel architecture design.

Related Documentation

The following documents are available from Texas Instruments. To obtain a copy of any of these TI documents, please call the Customer Response Center (CRC) at (800) 232-3200 unless otherwise noted. When ordering, please identify the book by its title and corresponding literature number.

TMS34082A Data Sheet (literature number SCGS001) is included in Appendix B of this book. It contains electrical specifications, timing information, and mechanical data for the TMS34082A.

SMJ34082A Data Sheet (literature number SGUS012A) is included in Appendix C of this book. It contains electrical specifications, timing information, and mechanical data for the SMJ34082A.

TMS34020 User's Guide (literature number SPVU019) discusses hardware aspects of the TMS34020, such as pin functions, architecture, stack operations, and interfaces. Contains the TMS34020 instruction set and interface to the TMS34082.

TMS34020 Data Sheet (literature number SPVS004) contains electrical specifications, timing information, and mechanical data for the TMS34020.

TMS34082 Software Tool Kit User's Guide describes the C compiler, assembler, linker, librarian, and simulator that are available for developing TMS34082 external instruction code. Call your TI sales representative for the demonstration version of the tool kit.

TMS340 Family Code-Generation Tools User's Guide (literature number SPVU004) describes the C compiler, assembler, linker, archiver, and auxiliary tools that are available for developing TMS34010, TMS34020, or TMS34020/TMS34082 code.

TMS34082 Assembly Support for Code-Generation Tools User's Guide (literature number SPVU029) summarizes the instruction code used with the TMS34082.

TIGA Interface User's Guide (literature number SPVU015) describes the Texas Instruments Graphics Architecture (TIGA), a software interface that standardizes communication between application software and TMS340-based hardware for IBM-compatible PCs.

TMS34082 3-D Graphics Library User's Guide describes an extensive array of C-callable functions including polygon clipping, shading, and vector and matrix operations. This library is TIGA-compatible and can also be used in non-TIGA applications. Call your TI sales representative or the DVP System Engineering Hotline for information on purchasing this product.

You may also find the following documentation useful. Many of the complex graphics instructions in the TMS34082 are based on algorithms found in this book:

Foley, James, and Andries van Dam. *Fundamentals of Interactive Computer Graphics.* Reading, Massachusetts: Addison-Wesley, 1982.

Style and Symbol Conventions

This document uses the following conventions.

Program listings, program examples, filenames, and symbol names are shown in a `special typeface` similar to a typewriter's. Examples use a **`bold version`** of the special typeface for emphasis.

Here is a sample program listing:

```
0011  0005  0001          .field    1, 2
0012  0005  0003          .field    3, 4
0013  0005  0006          .field    6, 3
0014  0006               .even
```

In syntax descriptions, the instruction is in a **bold** typeface font and parameters are in an *italic* typeface. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are in *italics* describe the type of information that should be entered. Here is an example of an instruction syntax:

NEGF *CRs, CRd*

This instruction has two parameters, indicated by *CRs* and *CRd.* When you use NEGF, the parameters must be actual TMS34082 registers, such as RA9 and RB1.

Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you don't enter the brackets themselves. Here's an example of an instruction that has an optional parameter:

MOVD **Rs+, CRd [, count]*

The MOVD instruction has three parameters. The first two parameters, *Rs* and *CRd*, are required. The third parameter, *count*, is optional. As this syntax shows, if you use the optional third parameter, you must precede it with a comma.

In the internal instruction set listings, Rs and Rd refer to TMS34020 source and destination registers, respectively. CRs and CRd refer to coprocessor or TMS34082 registers.

Trademarks

EPIC, SCOPE, and TIGA are trademarks of Texas Instruments Incorporated.

IBM, PC-DOS, and PC/AT are trademarks of International Business Machines, Inc.

MS-DOS is a trademark of Microsoft Corporation.

NeXT is a trademark of NeXT, INC.

PAL is a registered trademark of Monolithic Memories, Inc.

X Windows Systems is a trademark of the Massachusetts Institute of technology.

If You Need Assistance . . .

If you want to . . .	Do this . . .
Receive more information about TI floating-point products	Call the CRC Hot Line: **(800) 232-3200** Or write to: Texas Instruments Incorporated Datapath VLSI Products Marketing Communications P.O. Box 655303, M/S 8316 Dallas, Texas 75265
Order TI documentation	Call the CRC Hot Line: **(800) 232-3200**
Ask questions about product operation or report suspected problems	Call DVP Systems Engineering Hot Line: **(214) 997–3970**
Inquiries related to this document:	Write to: Texas Instruments Incorporated Datapath VLSI Products Marketing Communications P.O. Box 655303, M/S 8316 Dallas, Texas 75265

Contents

Figures

Tables

Examples

Chapter 1

Overview of the TMS34082

The Texas Instruments TMS34082 Graphics Floating-Point Processor is designed for your advanced numeric applications. This high-performance device offers an outstanding price/performance ratio, flexibility, and ease of use with TI's development tools. The TMS34082 acts as either a tightly coupled coprocessor for the TMS34020 Graphics System Processor (GSP), as an independent processor, or as a coprocessor to another host.

By integrating a 64-bit IEEE Floating-Point Unit (FPU) with a modified Harvard architecture microprocessor and multi-port register files onto a single device, the TMS34082 can sustain exceptionally high internal throughput rates. All internal data paths are 64 bits wide. The RISC-like basic instruction set executes at a rate of one instruction per clock cycle. In addition, many popular numeric and graphics routines are contained directly on-chip.

The TMS34082 offers an attractive cost/performance ratio and supports the integration of graphics- and computation-intensive solutions in a single, low-cost device. The cost per MFLOP performance achieved by the TMS34082 makes it an ideal floating-point solution.

Texas Instruments supports the TMS34082 with a complete set of PC-based hardware and software development tools, including an easy-to-use simulator, a TMS34020/TMS34082 software development board, a TMS34082 demonstration board, a 3-D graphics library, an optimizing C compiler, a macro-assembler, and software libraries.

1.1 TMS34082 Key Features

High-performance floating-point RISC processor optimized for graphics

Two operating modes:

- Floating-point coprocessor for the TMS34020 Graphics System Processor
- Independent floating-point processor

Direct connection to TMS34020 coprocessor interface

- Direct extension to the TMS34020 instruction set
- Multiple TMS34082 capability

Fast instruction cycle time:

- TMS34082-40. . .50-ns coprocessor mode, 50-ns host-independent mode
- TMS34082-32. . .62.5-ns coprocessor mode, 60-ns host-independent mode

Sustained data transfer rates of 160M bytes/second (TMS34082-40)

Sequencer executes internal or user-programmed instructions

Twenty-two 64-bit data registers

Comprehensive floating-point and integer instruction set

Internal programs for vector, matrix, and 3-D graphics operations

Full IEEE Std 754-1985 compatibility:

- Addition, subtraction, multiplication, and comparison
- Division and square root

Selectable data formats:

- 32-bit integer
- 32-bit single-precision floating-point
- 64-bit double-precision floating-point

External memory addressing capability:

- Program storage (up to 64K words)
- Data storage (up to 64K words)

0.8-μm EPIC™ CMOS technology

- High-performance
- Low power (<1.5 W)

1.2 Performance Benchmarks

Tables 1–1 and 1–2 show benchmark timings. Table 1–3 describes the benchmarks selected to show TMS34082 performance.

Table 1–1. TMS34082 Integer Benchmark Timings†

Benchmark	Units of Measure	Integer	
		TMS34082A-32	TMS34082A-40
MIPS Equivalents	MIPS	32	40
Dhrystones	Dhrystones/second	10,240	12,800

Table 1–2. TMS34082 Floating-Point Benchmark Timings†

Benchmark	Units of Measure	Single-Precision		Double-Precision	
		TMS34082A-32	TMS34082A-40	TMS34082A-32	TMS34082A-40
Peak MFLOPS	MFLOPS	32	40	16	20
Linpack	MFLOPS	11.0	13.7	6.3	7.9
Whetstones	MWhetstones/second	7.9	9.9	4.6	5.7

† Based on actual measured system performance.

Table 1–3. Description of the Benchmarks Used‡

Benchmark	Operations Tested	Where Applicable
Linpack	Floating-point and integer array manipulation, including Gaussian elimination, vector dot products, and matrix multiplication	Dense systems of linear equations with array manipulation
Whetstones	Mathematical operations: integer, floating-point, and trigonometric operations	Engineering and scientific computing applications
Dhrystones	Enumeration, record and pointer manipulation, and integer operations	Systems programming applications

‡ Reference: Hinnant, David F., "What Makes a Good Benchmark?", *MIPS*, September, 1989, pp. 102-103.

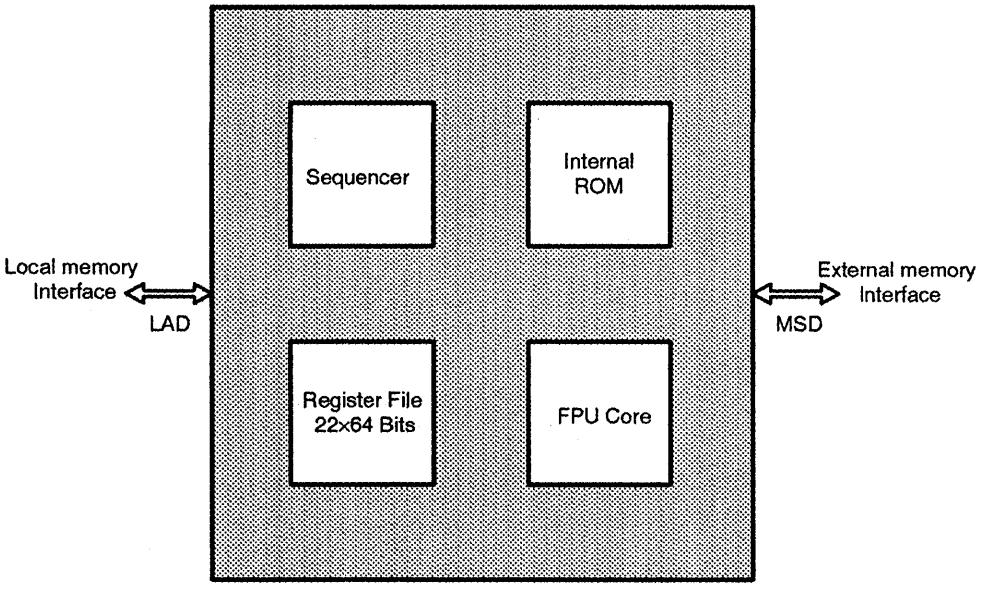
1.3 TMS34082 General Description

The TMS34082 is a high-speed floating-point processor implemented in the Texas Instruments advanced 0.8 μm CMOS technology. On a single chip, the TMS34082 combines a 16-bit sequencer and a three-operand 64-bit FPU (source A, source B, destination) with twenty-two 64-bit data registers. The data registers are organized into two banks of 10 registers each, with two registers for internal feedback. In addition, an instruction register to control FPU execution, a status register to retain the most recent FPU status results, eight control registers, and a two-register stack are provided. The key architectural elements are shown in Figure 1–1.

The ALU and the multiplier are closely coupled and work in parallel to perform sums of products and products of sums. During multiply/accumulate operations, both the ALU and the multiplier are active, and the registers in the FPU core can be used to feed back products and accumulate sums without tying up locations in register banks A and B.

Data or code may be transferred between the LAD and MSD ports at the rate of one 32-bit word per clock cycle with a one clock latency. That comes out to 1.28 billion bits/second. This provides sufficient bandwidth to quickly transfer vector or scalar arrays into or out of external memories. Up to 512 words may be transferred with a single memory move instruction.

Figure 1–1. TMS34082 High-Level Block Diagram



The TMS34082 complies fully with IEEE Std 754-1985, the industry standard for binary floating-point formats. Floating-point operands can be either single- or double-precision. In addition to floating-point operations, the TMS34082 performs 32-bit integer arithmetic, logical comparisons, and shifts. Integer operations may be performed on 32-bit 2s complement or unsigned operands. Floating-point to integer and integer to floating-point conversions are also available.

The comprehensive RISC-like instruction set eliminates the need for complex CISC-type instructions or wide microcoded instruction words. By programming the TMS34082 at the simplest level, operations are customized for each application and most instructions execute in one clock cycle. Divide and square root instructions are ideal for numeric processing and graphics rendering, such as ray tracing routines. Using dedicated hardware and patented algorithms, the TMS34082 calculates a 64-bit double-precision divide or square root result in only 13 or 16 clock cycles, respectively.

In a single clock cycle, two single-precision or integer operands may:

1) Be read from the register file

2) Be run through the ALU and/or multiplier

3) Have result placed back into the register file

This is accomplished with both the internal pipeline and output registers disabled. Double-precision multiplies take two clock cycles to complete. Such low latencies simplify writing assembly language code, eliminating the problem of data coherency in a long pipeline. Refilling or flushing the instruction pipeline is fast, also.

An internal ROM includes many commonly used matrix, graphics, and vector routines as described below. With the exception of MIN-MAX and compare operations, these routines are constructed directly from the TMS34082's basic instruction set. The internal routines include:

Matrix operations consisting of 1×3, 3×3, 1×4, and 4×4 matrix multiplies

Graphics routines such as backface testing, clipping, 2-D and 3-D compares, linear interpolation, 1-D and 2-D MIN-MAX, viewport scaling and conversion, cubic splines, and polygon elimination

Vector operations including add, subtract, magnitude, scaling, dot product, cross product, normalization, and reflection

Additional routines for 3×3 convolution, multiply/accumulate, and polynomial expansion

When used with the TMS34020, the TMS34082 operates in coprocessor mode. The TMS34020 can control multiple TMS34082 coprocessors without any additional glue logic or buffering. The clock and control signals are generated directly by the TMS34020. You can use external memory to store subroutines as well as data for those subroutines. See Chapter 5 for additional information.

When used alone or with processors other than the TMS34020, the TMS34082 functions in host-independent mode. The TMS34082 is fully programmable and can interface to other processors (such as a RISC, 80x86, or Motorola MC680x0 processor) or floating-point subsystems through its two 32-bit bidirectional buses. Chapter 6 covers this mode in greater detail.

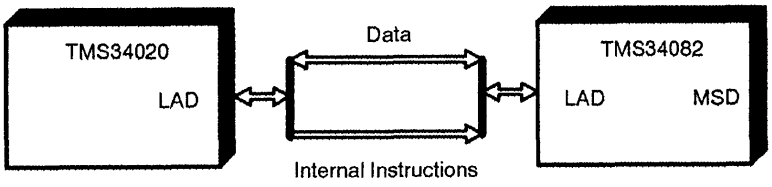
Other features include:

- Support of common microprocessor addressing modes (register, direct, indirect, postincrement, immediate)
- A fully synchronous, on-chip, direct memory interface to SRAMs/EPROMs with no glue logic and to DRAMs/VRAMs with minimal glue logic
- Fully user-programmable hardware and software realtime interrupts.

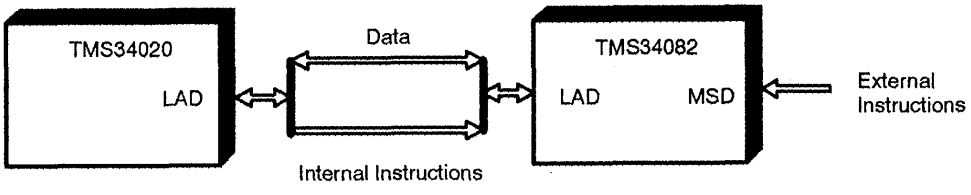
The TMS34082 may implement a von Neumann architecture, a modified Harvard architecture, or a mixture of both. In a von Neumann architecture, data and instruction memories both reside on the same bus. However, a Harvard architecture has separate data and instruction sources so that both may be fetched in parallel. External data may originate from either the LAD or MSD ports. External instructions may only come from the MSD port, but the LAD port can be used to input jump entries into the MSD port memory.

Figure 1–2 shows possible TMS34082 bus architectures for coprocessor mode. In addition, Figure 1–3 shows several example architectures for host-independent mode.

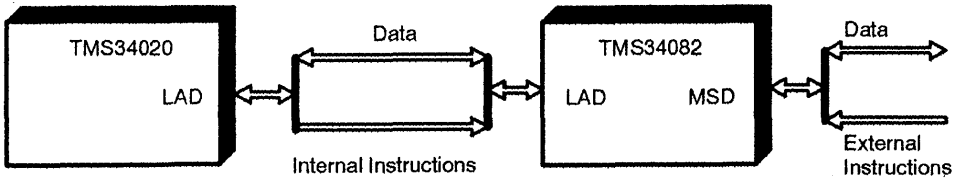
Figure 1–2. Coprocessor Mode Bus Architectures



Data and Instructions from LAD Port

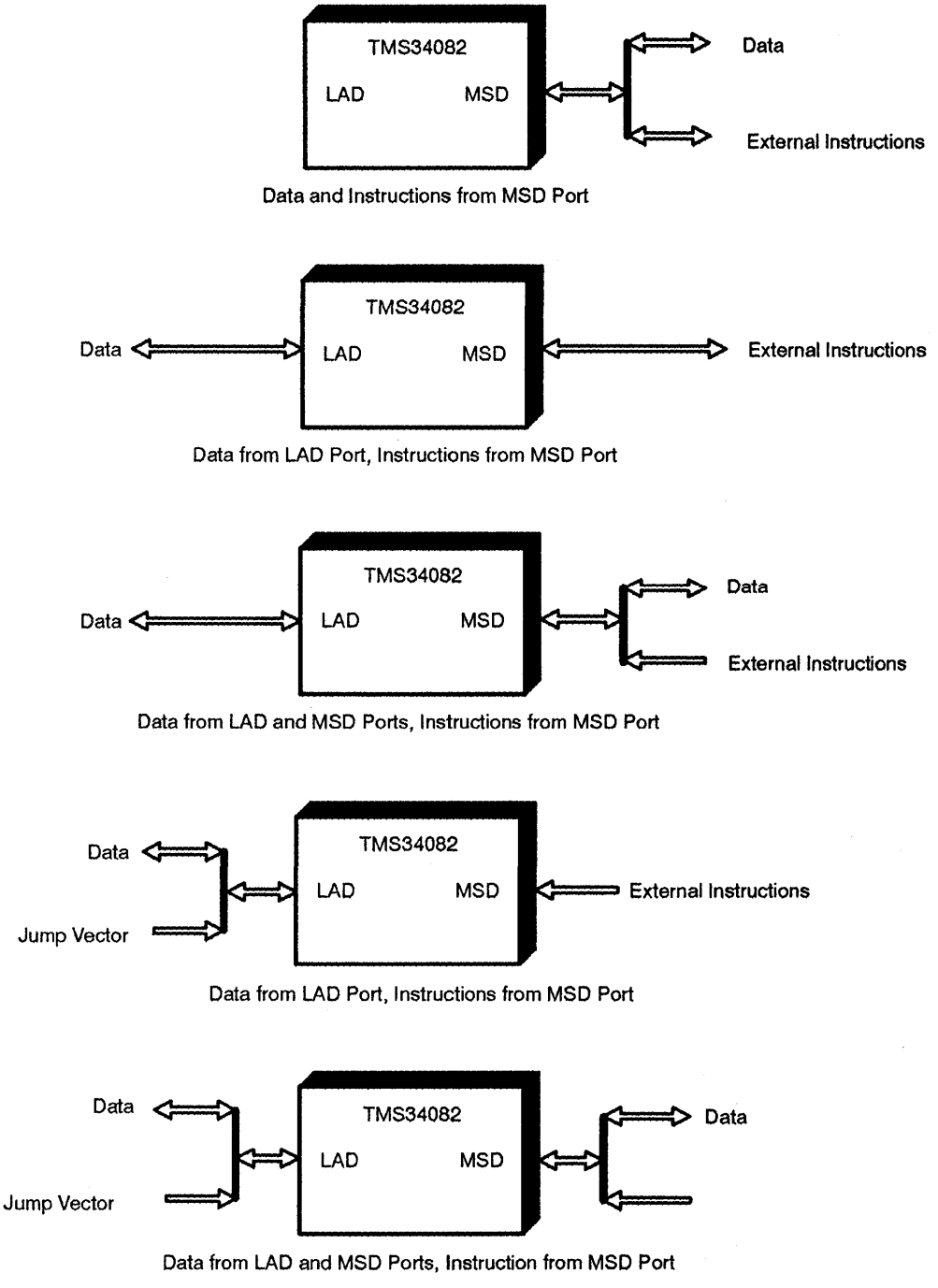


Data from LAD Port, Instructions from LAD and MSD Port



Data from LAD and MSD Ports, Instructions from LAD and MSD Port

Figure 1–3. Host-Independent Mode Bus Architectures



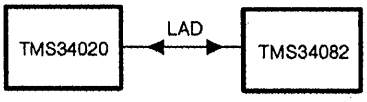
1.4 Typical Applications

The 64-bit power and exceptional flexibility of the TMS34082 meet system computing requirements across the performance spectrum. These range from workstations to personal computers to embedded controllers. Table 1–4 lists typical end uses for this device. Figure 1–4 shows several examples of systems using the TMS34082.

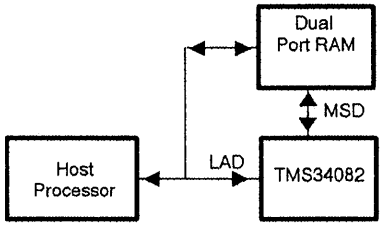
Table 1–4. Applications for the TMS34082

Numeric Processor	Graphics Processor
CAD/CAE workstations	3-D graphics processing
UNIX/DOS accelerator for RISC/CISC machines	Graphics workstations/super workstations
Scientific computing	Image processing
Personal computers	Laser printers
Vector processing	Graphics rendering engines
Multiprocessing architectures	Imaging compression/decompression, JPEG
Digital signal processing	Flight simulators
High-speed protocol engines	Electronic publishing
Array processing	Computer animation

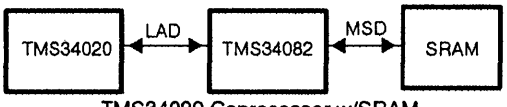
Figure 1–4. Sample TMS34082 Architectures



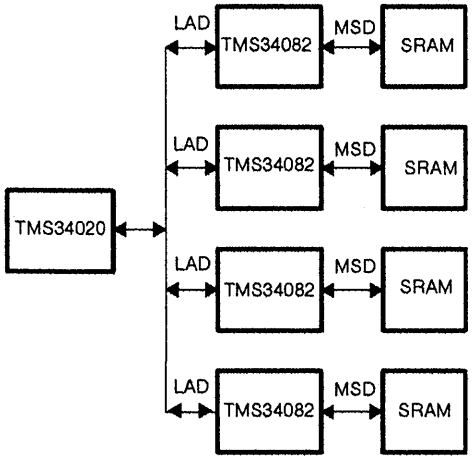
TMS34020 Coprocessor



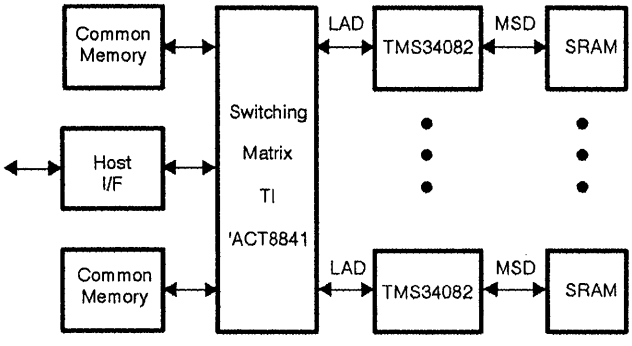
Coprocessor for 80x86, Motorola MC680x0, or RISC Processor



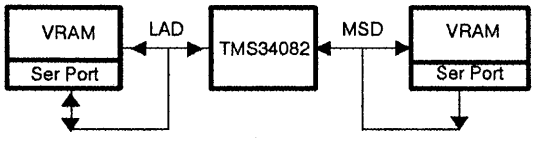
TMS34020 Coprocessor w/SRAM



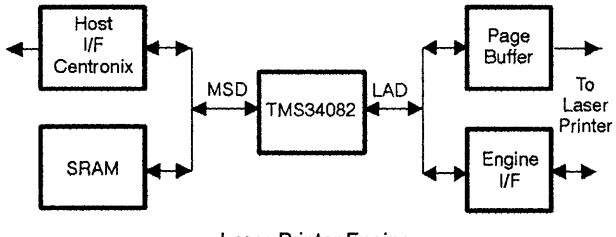
TMS34020 with Multiple TMS34082s



Multicomputer



Numerical/Graphics Engine



Laser Printer Engine

1.5 Development Tools

1.5.1 TMS34082 Software Tool Kit

The TMS34082 Software Tool Kit can be used to develop code for host-independent mode applications or for external subroutines in coprocessor mode. The tool kit includes:

An ANSI standard, optimizing C compiler

A macro-assembler

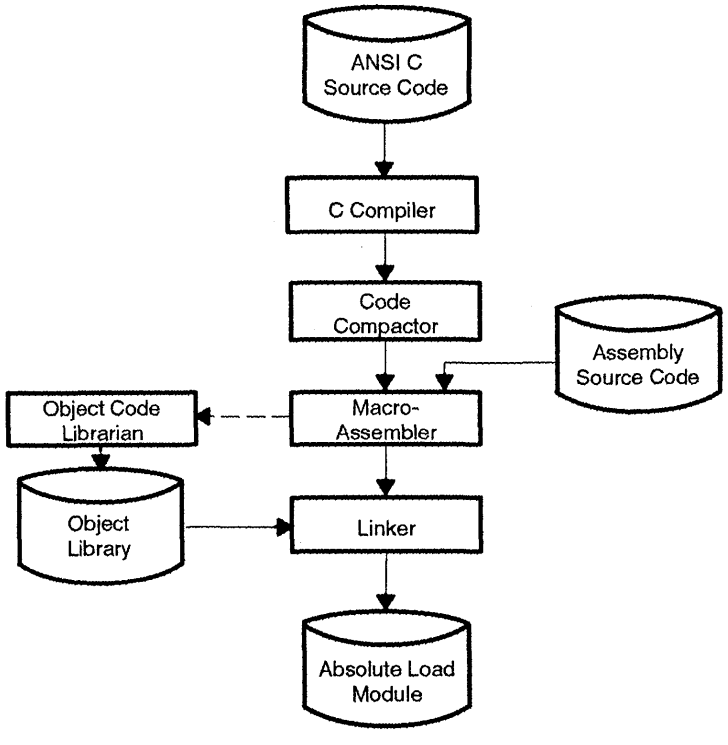
A linker

An object code librarian

A functional simulator

The C compiler supports common subexpression elimination. A peephole optimizer is also provided to further enhance the execution speed and the code size of the source program. Inline assembly code can be incorporated into the C program for time-critical and hardware-dependent code sections. The object librarian allows the storage of frequently used functions in libraries for easy access (see Figure 1–5).

Figure 1–5. Overview of TMS34082 Code-Generation Tools



Included with the TMS34082 tool kit are highly optimized transcendental assembly language routines for sine, cosine, tangent, arc sine, arc cosine, and arc tangent. These are accurate to the least significant bit.

The TMS34082 tool kit will execute on an IBM PC/AT or compatible machine with MS-DOS (or PC-DOS) 2.0 or higher, 640K of memory, one floppy drive, and one hard drive. An 80287/80387 math coprocessor is required for the simulator. A demonstration version of the Software Tool Kit is also available.

The interactive simulator displays the entire machine state of the TMS34082 (such as registers, address counter, stack, status register) and works with the C compiler/assembler/linker object files. The simulator is menu driven. During program execution, breakpoints may be set and the trace memory displayed. The cycle counting feature is useful when evaluating performance of the processor or during code optimization.

The TMS340 Family compiler and assembler, which support both the TMS34020 and TMS34082, are described in subsection 1.6.3 of this document.

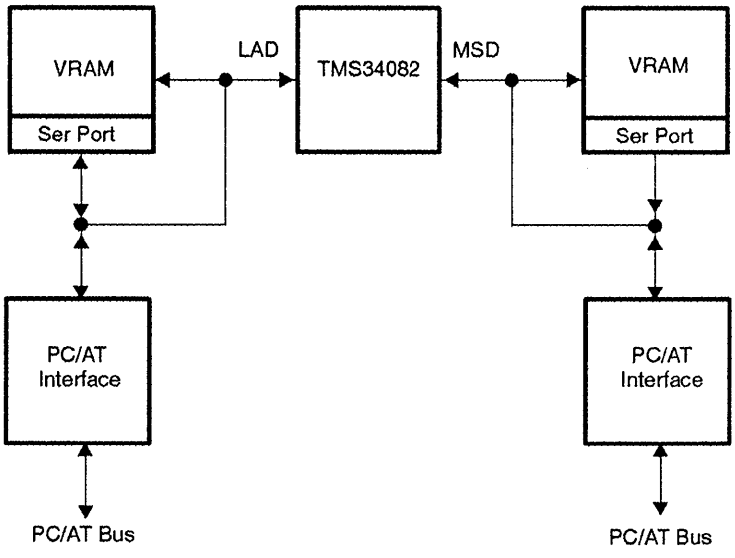
1.5.2 TMS34082 3-D Graphics Library

The TMS34082 3-D Graphics Library contains an extensive array of C-callable functions including polygon clipping, shading, and vector and matrix operations. The library is TIGA-compatible and can also run as a non-TIGA product, giving the user portability and flexibility. The task of porting graphics standard to the TMS34020/TMS34082 is greatly simplified with the variety of functions in the library. The library also includes a 3-D graphics pipeline that can shorten the development time for application programs.

1.5.3 TMS34082 Demonstration Board

The TMS34082 Demonstration Board is a 40-MFLOP parallel processor with up to 3M bytes of on-board memory. This powerful board allows you to evaluate performance and write code for the TMS34082 using the software tool kit, develop algorithm implementations, and integrate the software modules with the hardware. In addition, programs are executed directly on the TMS34082, resulting in much faster execution times than a software simulator. The board plugs into a PC/AT™ 32-bit card slot. Figure 1–6 is a block diagram of the demonstration board.

Figure 1–6. TMS34082 Demonstration Board Block Diagram



Built on a PC/AT card occupying a single slot, the TMS34082 Demonstration Board features:

TMS34082-40 Floating-Point Processor (operating in host-independent mode)

20 MHz processor clock speed, 7.9 MFLOPs double-precision Linpack

Fully programmable: von Neumann or modified Harvard architectures or both

2M-bytes VRAM memory on LAD port accessible though PC/AT bus interface

256K-bytes VRAM memory on MSD port accessible through PC/AT bus interface, expandable up to 1M bytes of VRAM memory

1.6 TMS34020 Graphics System Processor

The TMS34020 Graphics System Processor (GSP) is an advanced 32-bit microprocessor optimized for graphic display systems. The TMS34020 is a member of the TMS340 family of computer graphics products from Texas Instruments.

The TMS34020 provides high-performance cost-effective solutions for applications that require efficient data manipulations in a graphics environment. The TMS34020 can be configured to serve in a host-based, standalone, or multiprocessing system. It has both host and multiprocessor interfaces to facilitate implementation of multiple TMS34020 systems.

The TMS34020 is supported by a full set of hardware and software development tools, including an optimizing C compiler, assembler, software libraries, a PC-based development board on a PC-based emulator. The TMS340 Family Code Generation Tools may be used to develop code for the TMS34082 in coprocessor mode. In addition, the TMS34020 is fully compatible with and supported by the Texas Instruments Graphics Architecture (TIGA).

1.6.1 TMS34020 Key Features

Fully programmable 32-bit general-purpose processor with 512M-byte linear address range (bit addressable)

Second generation graphics system processor:

- Object code compatible with the TMS34010
- Enhanced instruction set
- Optimized graphics instructions
- Direct coprocessor interface to TMS34082 Floating-Point Processor

On-chip peripheral features include:

- Programmable CRT control
- Direct DRAM/VRAM interface
- Direct communication with an external (host) processor
- Communication with multiple TMS34020s
- Functional expansion with the coprocessor interface
- Automatic CRT display refresh

Instruction set supports special graphics functions such as pixel processing, XY addressing, and window clip/hit detection

Programmable 1-,2-,4-,8-,16-, or 32-bit pixel size

16 Boolean and 6 arithmetic pixel processing options (raster-ops)

30 general-purpose 32-bit registers

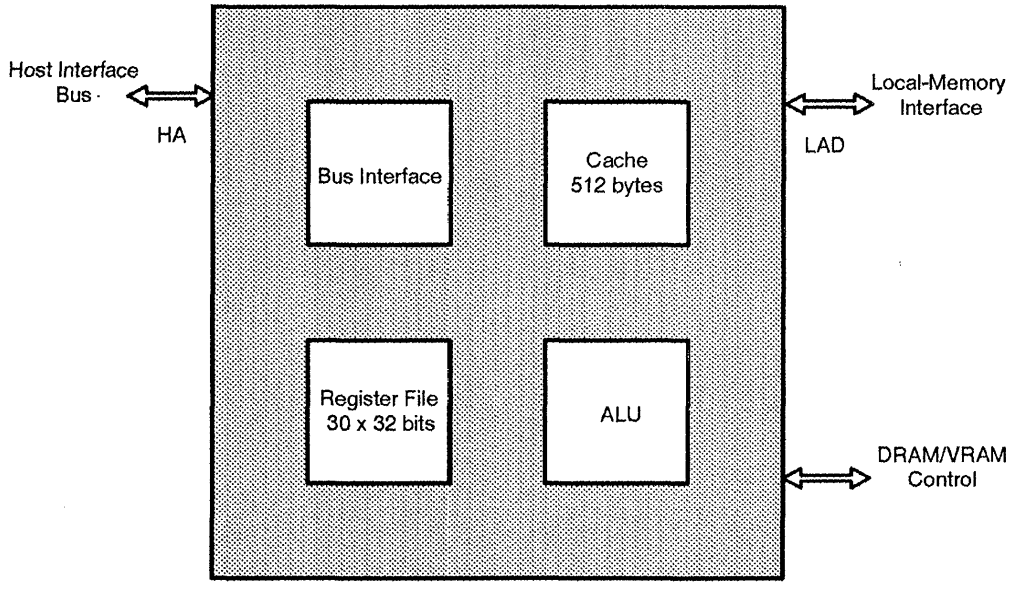
512-byte LRU on-chip instruction cache

General Description

The TMS340 family from Texas Instruments combines the best features of general-purpose microprocessors and graphics controllers to create a range of cost-effective, flexible, powerful graphics systems. The key features of the TMS340 family are speed, a high degree of programmability, and efficient manipulation of hardware-supported data types such as pixels and 2-dimensional pixel arrays.

With a built-in instruction cache, the ability to simultaneously access memory and registers, and an instruction set that enhances raster graphics operations, the TMS34020 provides programmable control of the CRT interface as well as the memory interface (both standard DRAM and multiport RAM). The 4G-bit (512M-byte) physical address space is completely bit addressable on bit boundaries using variable width data fields (1 to 32 bits). Figure 1–7 is a TMS34020 high-level block diagram.

Figure 1–7. TMS34020 High-Level Block Diagram



The TMS34020 unique memory interface speeds performance of tasks such as bit alignment and masking while supporting advanced DRAM access modes. The 32-bit architectures supplies the large blocks of contiguously-addressable memory that are necessary in graphics applications.

Systems designed with the TMS34020 can utilize VRAM technology to facilitate applications such as high-bandwidth frame buffers. This circumvents the bottleneck often encountered when using conventional DRAMs in graphics systems.

The TMS34020 instruction set includes a full complement of general-purpose instructions, as well as graphics functions, that can be used to construct efficient high-level instructions. The instructions support arithmetic and Boolean operations, data moves, conditional jumps, and subroutine calls and returns.

The TMS34020 architecture supports a variety of pixel sizes, frame buffer sizes, and screen sizes. On-chip functions have been carefully selected so that no functions tie the TMS34020 to a particular display resolution. This enhances the portability of graphics software and allows the TMS34020 to adapt to graphics standards such as MIT's X-Windows™, CGI/CGM, GKS, NAPLPS, PHIGS, and evolving industry standards.

Texas Instruments offers a wide variety of system solutions. The simplest TMS340 graphics system consists of the TMS34020 alone. Floating-point computations are performed in software using IEEE floating-point libraries. Adding a TMS34082 appears merely as an extension to the TMS34020 instruction set. The same calculations run much faster in dedicated hardware rather than software.

Adding external memory to the TMS34082 allows user-programmed subroutines, such as shading or contour fitting, to execute while the TMS34020 is performing other functions. Since the data for the subroutines is also in external memory, the TMS34082 is effectively decoupled from the TMS34020. The TMS34020 can poll the TMS34082 to see if the subroutine has finished. The highest performance TMS340 graphics solutions contain one or more TMS34020 along with multiple TMS34082s in a parallel processing environment. The TMS34020 acts as the display manager and also orchestrates tasks for the floating-point coprocessors. Jobs and/or data may be loaded into external memory of one TMS34082 while other TMS34082s are still executing.

1.6.2 TMS34020 Software Tools

Texas Instruments offers extensive development support for the TMS340 graphics family. Software tools for the TMS34020 also comprehend the TMS34082. The TMS340 Family software tools include:

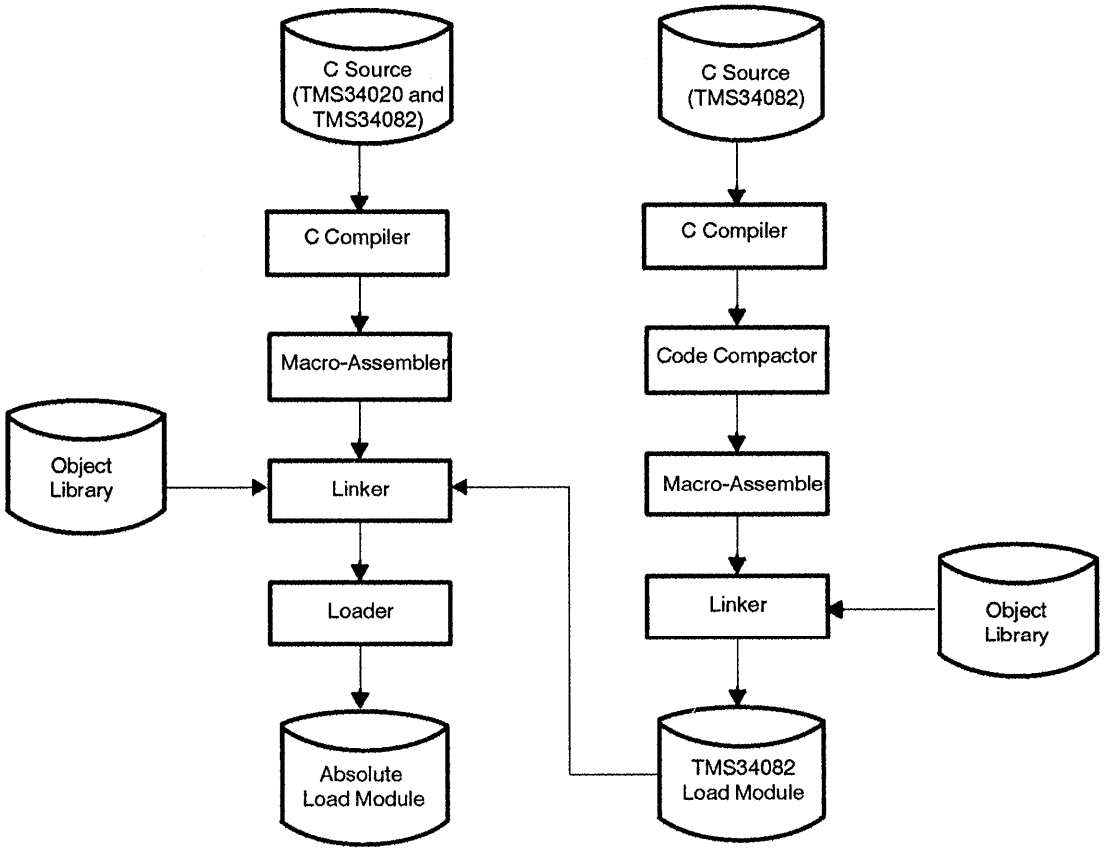
An optimizing C compiler
An assembler
An archiver for building object libraries
A linker
A loader for TMS34020 and TMS34082 absolute load modules
A C source debugger

The compiler accepts programs written in C language. It outputs assembly language source code that is then processed by the assembler to convert the mnemonics to object code. The compiler and assembler generate efficient TMS34082 code in the form of internal instructions. The C compiler allows time-critical routines written in assembly language to be called from within the C program. The converse is also available; assembly routines may call C functions.

If external TMS34082 memory is present, the TMS34082 Software Tool Kit must be used to generate the subroutine code in the form of external instructions. When the TMS34082 load module has been generated, the TMS34020 loader can download both load modules as shown in Figure 1–8.

The TMS340 Family C Source Debugger supports both the TMS34020 and the TMS34082 in coprocessor mode. Other debugging tools for the TMS34082 in coprocessor and host-independent modes are available from third-party vendors.

Figure 1–8. TMS34020 and TMS34082 Software Tools



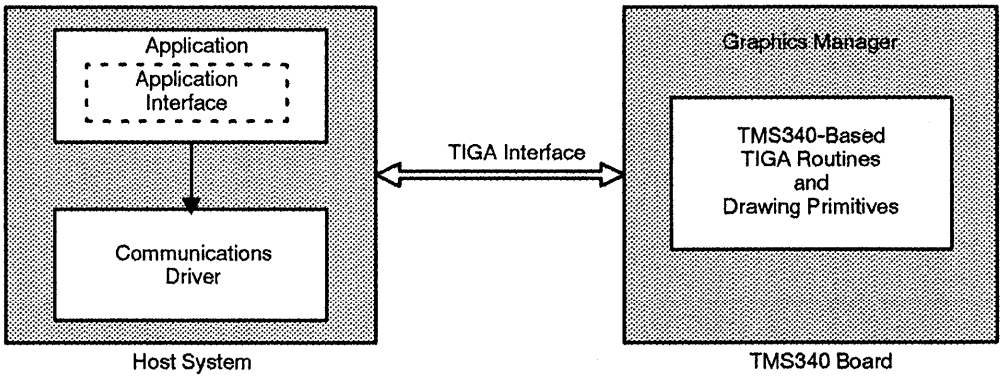
TMS340 Family Software Code Generation Tools (used for generating TMS34020 code and TMS34082 internal instructions)

TMS34082 Software Tool Kit (generates TMS34082 code for external memory)

1.6.3 TIGA™ Graphics Interface

The Texas Instruments Graphics Architecture (TIGA) is a software interface standard for the TMS340 family of graphics system processors. TIGA enhances the performance of MS-DOS-based PCs that contain a TMS34020 or TMS34020 (and an optional TMS34082) and an 8088/86 or 80286/80386 host microprocessor by optimizing communications between the graphics processor and the host processor. The TIGA interface allows the host and graphics processors to share execution of the application, as shown in Figure 1–9.

Figure 1–9. Graphics Processing Shared Between TMS340 and Host Processors

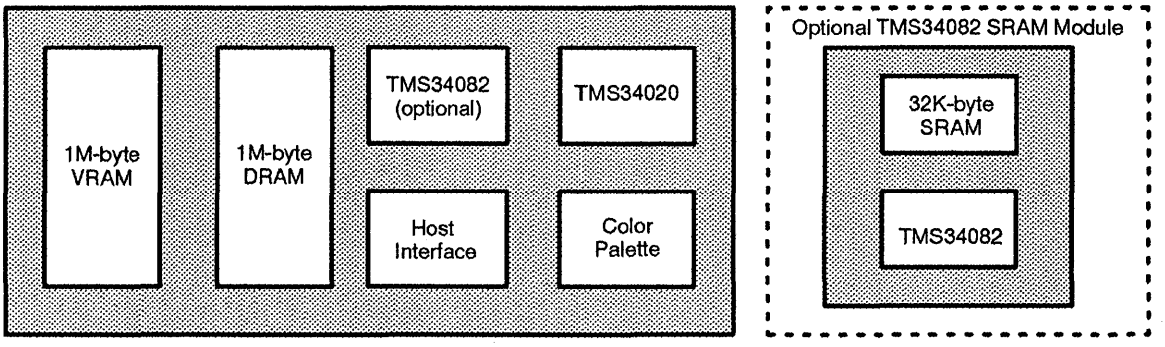


1.6.4 TMS34020 Software Development Board

The TMS34020 Software Development Board (SDB20) is a high-performance PC/AT bus graphics card. It allows you to write applications software for the TMS34020 and its companion floating-point processor, the TMS34082. The board also demonstrates the simplicity of hardware design using the TMS34020 and TMS34082 for high-performance bit-mapped graphics displays.

An optional upgrade kit, the TMS34082 SRAM Upgrade Kit, contains a business card sized board with the TMS34082 and 32K bytes of SRAM, plus software and documentation. The board plugs into the TMS34082 socket presently existing on the SDB20.

Figure 1–10. TMS34020 SDB Block Diagram



Key features of the TMS34020 SDB include:

- 1M-byte VRAM organized as 256K × 32 bits
- 1M-byte DRAM organized as 256K × 32 bits
- TMS34082 Floating-Point Coprocessor (optional)
- VGA support for 640 × 480 pixel resolution
- Software selectable resolutions:
  - 1024 × 768 by 4 or 8 bits per pixel
  - 640 × 480 by 4 or 8 bits per pixel
  - 640 × 480 VGA mode
- Software configurable base address over a full 16M-byte range
- TMS34020 emulation support

1.7 TMS34082 Ordering Information

For the latest ordering and pricing information, please call your local TI field sales representative or authorized TI distributor. Table 1–5 summarizes the products available for the TMS34082.

Table 1–5. TMS34082 Product Information

Type	Description	Part Number
Silicon Devices	TMS34082A device, 32 MHz, 145-pin ceramic PGA package	TMS34082AGC-32
	TMS34082A device, 40 MHz, 145-pin ceramic PGA package	TMS34082AGC-40
Documentation	TMS34082A Data Sheet	SCGS001
	TMS34082 Designer's Handbook	SCGU004
Software	TMS34082 Demonstration Software Tool Kit	Contact TI
	TMS34082 Software Tool Kit	TMDS3440808201
	TMS34082 3-D Graphics Library	Contact TI
	TIGA Software Developer's Kit (includes the TMS340 Family Code Generation Tools and C Debugger for the PC)	TMS340SDK-PC
Hardware	TMS34020 Software Development Board (SDB20)	TMS3460120000
	TMS34082 SRAM Upgrade Kit	TMDS3481800-02

1.8 Technical Assistance

The Texas Instruments Datapath VLSI Products Systems Engineering group is a resource available to help you in the selection of TI's high-performance FPUs, such as the TMS34082 Graphics Floating-Point Processor. Located in Dallas, the group works directly with designers to provide ready answers to device-related questions and also prepares a variety of applications information. The phone number for the DVP Systems Engineering hotline is (214) 997-3970.

Chapter 2

Pinout and Pin Descriptions

This chapter illustrates the TMS34082 pinouts and provides detailed descriptions of the TMS34082 signals. For mechanical dimensions of the TMS34082A packages, please refer to the data sheet in Appendix B. For mechanical dimensions of the SMJ34082A packages, please refer to the data sheet in Appendix C.

2.1 Pinout

The TMS34082A and the SMJ34082A are offered in a ceramic, 145-pin grid array (PGA) package (GC). Figure 2-1 shows the145-pin PGA pinout.

Figure 2–1. TMS34082 Pinout, 145-Pin PGA Package

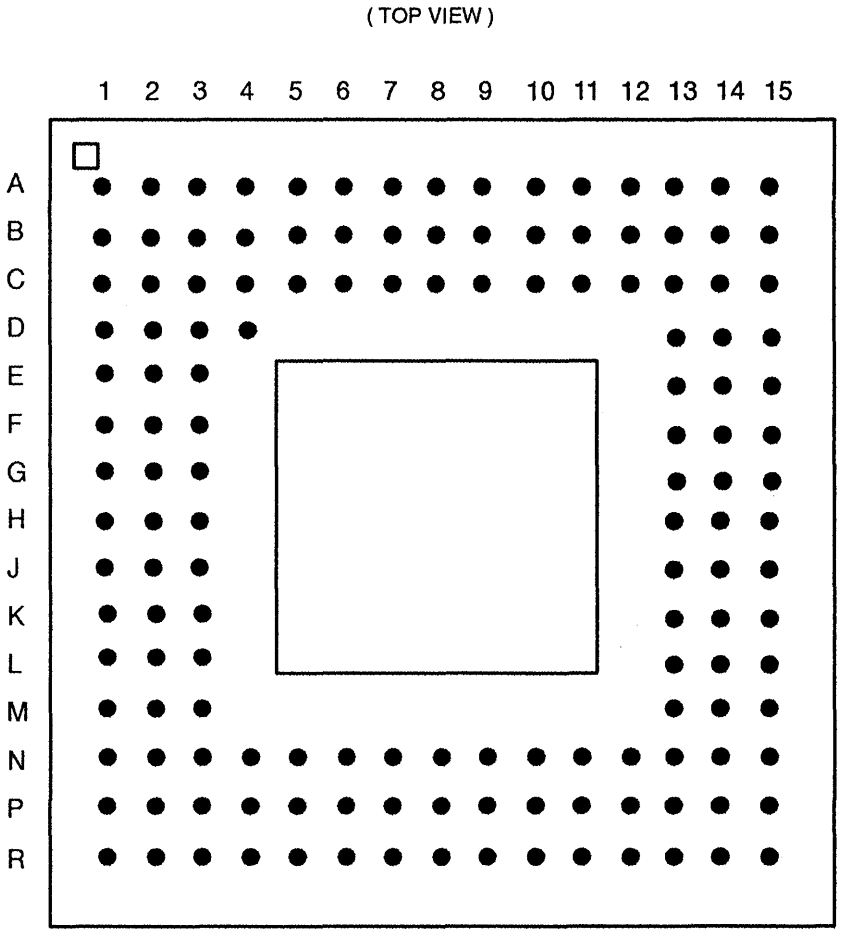


Table 2–1. Pin Assignments (PGA Package)

Pin		Pin		Pin		Pin		Pin	
GC#	**Name**	**GC#**	**Name**	**GC#**	**Name**	**GC#**	**Name**	**GC#**	**Name**
A1	NC	B15	LAD27	F1	MSD10	K15	RDY	P2	NC
A2	LAD1	C1	MSD4	F2	MSD9	L1	MSD18	P3	MSD29
A3	LAD3	C2	MSD3	F3	V_{CC}	L2	MSD21	P4	MSD31
A4	LAD5	C3	MSD0	F13	CORDY	L3	MSD23	P5	MSA1
A5	LAD8	C4	V_{SS}	F14	$\overline{ALTCH}$	L13	V_{SS}	P6	MSA3
A6	LAD9	C5	V_{CC}	F15	$\overline{CAS}$	L14	CID0	P7	MSA6
A7	LAD11	C6	LAD6	G1	MSD13	L15	CID2	P8	MSA8
A8	LAD12	C7	V_{SS}	G2	MSD12	M1	MSD20	P9	MSA10
A9	LAD13	C8	V_{CC}	G3	MSD11	M2	MSD24	P10	MSA13
A10	LAD15	C9	V_{SS}	G13	$\overline{WE}$	M3	V_{SS}	P11	MWR
A11	LAD17	C10	V_{CC}	G14	EC1	M13	V_{CC}	P12	$\overline{MOE}$
A12	LAD19	C11	LAD21	G15	EC0	M14	LCLK1	P13	INTG
A13	LAD22	C12	V_{SS}	H1	MSD14	M15	LCLK2	P14	BUSFLT
A14	LAD24	C13	LAD25	H2	TDO	N1	MSD22	P15	$\overline{RAS}$
A15	NC	C14	LAD26	H3	V_{SS}	N2	MSD26	R1	NC
B1	MSD1	C15	LAD29	H13	V_{SS}	N3	V_{CC}	R2	MSD27
B2	NC	D1	MSD6	H14	$\overline{LOE}$	N4	MSD28	R3	MSD30
B3	LAD0	D2	MSD5	H15	TDI	N5	V_{SS}	R4	MSA0
B4	LAD2	D3	MSD2	J1	MSD15	N6	V_{CC}	R5	MSA2
B5	LAD4	D4	NC	J2	MSD16	N7	MSA5	R6	MSA4
B6	LAD7	D13	V_{CC}	J3	V_{CC}	N8	V_{SS}	R7	MSA7
B7	LAD10	D14	LAD28	J13	CC	N9	V_{CC}	R8	TCK
B8	TMS	D15	LAD31	J14	MASTER	N10	MSA14	R9	MSA9
B9	LAD14	E1	MSD8	J15	CLK	N11	V_{SS}	R10	MSA11
B10	LAD16	E2	MSD7	K1	MSD17	N12	$\overline{MAE}$	R11	MSA12
B11	LAD18	E3	V_{SS}	K2	MSD19	N13	LRDY	R12	MSA15
B12	LAD20	E13	V_{SS}	K3	V_{SS}	N14	SF	R13	DS/$\overline{CS}$
B13	LAD23	E14	LAD30	K13	CID1	N15	$\overline{RESET}$	R14	$\overline{MCE}$
B14	NC	E15	$\overline{COINT}$	K14	$\overline{INTR}$	P1	MSD25	R15	NC

Table 2-2. Alphabetical Listing — Pin Assignments (PGA Package)

Pin		Pin		Pin		Pin		Pin	
Name	**GC#**	**Name**	**GC#**	**Name**	**GC#**	**Name**	**GC#**	**Name**	**GC#**
$\overline{\text{ALTCH}}$	F14	LAD14	B9	MSA3	P6	MSD16	J2	TCK	R8
BUSFLT	P14	LAD15	A10	MSA4	R6	MSD17	K1	TDI	H15
$\overline{\text{CAS}}$	F15	LAD16	B10	MSA5	N7	MSD18	L1	TDO	H2
CC	J13	LAD17	A11	MSA6	P7	MSD19	K2	TMS	B8
CID0	L14	LAD18	B11	MSA7	R7	MSD20	M1	V_{CC}	C5
CID1	K13	LAD19	A12	MSA8	P8	MSD21	L2	V_{CC}	C8
CID2	L15	LAD20	B12	MSA9	R9	MSD22	N1	V_{CC}	C10
CLK	J15	LAD21	C11	MSA10	P9	MSD23	L3	V_{CC}	D13
$\overline{\text{COINT}}$	E15	LAD22	A13	MSA11	R10	MSD24	M2	VCC	F3
CORDY	F13	LAD23	B13	MSA12	R11	MSD25	P1	V_{CC}	J3
DS/$\overline{\text{CS}}$	R13	LAD24	A14	MSA13	P10	MSD26	N2	V_{CC}	M13
EC0	G15	LAD25	C13	MSA14	N10	MSD27	R2	V_{CC}	N3
EC1	G14	LAD26	C14	MSA15	R12	MSD28	N4	V_{CC}	N6
INTG	P13	LAD27	B15	MSD0	C3	MSD29	P3	V_{CC}	N9
$\overline{\text{INTR}}$	K14	LAD28	D14	MSD1	B1	MSD30	R3	V_{SS}	C4
LAD0	B3	LAD29	C15	MSD2	D3	MSD31	P4	V_{SS}	C7
LAD1	A2	LAD30	E14	MSD3	C2	MWR	P11	V_{SS}	C9
LAD2	B4	LAD31	D15	MSD4	C1	NC	A1	V_{SS}	C12
LAD3	A3	LCLK1	M14	MSD5	D2	NC	A15	V_{SS}	E3
LAD4	B5	LCLK2	M15	MSD6	D1	NC	B2	V_{SS}	E13
LAD5	A4	$\overline{\text{LOE}}$	H14	MSD7	E2	NC	B14	V_{SS}	H3
LAD6	C6	LRDY	N13	MSD8	E1	NC	D4	V_{SS}	H13
LAD7	B6	$\overline{\text{MAE}}$	N12	MSD9	F2	NC	P2	V_{SS}	K3
LAD8	A5	MASTER	J14	MSD10	F1	NC	R1	V_{SS}	L13
LAD9	A6	$\overline{\text{MCE}}$	R14	MSD11	G3	NC	R15	V_{SS}	M3
LAD10	B7	$\overline{\text{MOE}}$	P12	MSD12	G2	$\overline{\text{RAS}}$	P15	V_{SS}	N5
LAD11	A7	MSA0	R4	MSD13	G1	RDY	K15	V_{SS}	N8
LAD12	A8	MSA1	P5	MSD14	H1	$\overline{\text{RESET}}$	N15	V_{SS}	N11
LAD13	A9	MSA2	R5	MSD15	J1	SF	N14	$\overline{\text{WE}}$	G13

2.2 Pin Functional Descriptions

The following tables contain the TMS34082 signal descriptions grouped by their functions.

Table 2–3. LAD Bus Signals

Pin Name	Pin No.	I/O/Z	Description
$\overline{\text{ALTCH}}$	F14	I	**Address Latch**, active low. In coprocessor mode, falling edge of $\overline{\text{ALTCH}}$ latches instruction and status present on the LAD bidirectional bus (LAD31-0).
		O	In host-independent mode, $\overline{\text{ALTCH}}$ is an address output write strobe for memory accesses on LAD31-0.
BUSFLT	P14	I	**Bus Fault**. In coprocessor mode when high, indicates a data fault on the LAD bus (LAD31-0) during current bus cycle which causes TMS34082 not to capture the current data on LAD bus. Tied low if not used. Not used in host-independent mode.
$\overline{\text{CAS}}$	F15	I	**Column Address Strobe**, active low. In the coprocessor mode, causes TMS34082 to latch LAD bus data on $\overline{\text{CAS}}$ low-to-high transition if LRDY was high and BUSFLT was low at the previous LCLK2 rising edge.
		O/Z	In host-independent mode, this signal is the read strobe output.
LAD0	B3	I/O/Z	**Local Address and Data Bus.** In coprocessor mode, used by TMS34020 to input instructions and data operands to TMS34082, and used by TMS34082 to output results. In host-independent mode, used by the TMS34082 for address output and data I/O.
LAD1	A2		
LAD2	B4		
LAD3	A3		
LAD4	B5		
LAD5	A4		
LAD6	C6		
LAD7	B6		
LAD8	A5		
LAD9	A6		
LAD10	B7		
LAD11	A7		
LAD12	A8		
LAD13	A9		
LAD14	B9		
LAD15	A10		
LAD16	B10		
LAD17	A11		
LAD18	B11		
LAD19	A12		
LAD20	B12		
LAD21	C11		
LAD22	A13		
LAD23	B13		
LAD24	A14		

Table 2–3. LAD Bus Signals (Continued)

Pin Name	Pin No.	I/O/Z	Description
LAD25 LAD26 LAD27 LAD28 LAD29 LAD30 LAD31	C13 C14 B15 D14 C15 E14 D15	I/O/Z	**Local Address and Data Bus.** In coprocessor mode, used by TMS34020 to input instructions and data operands to TMS34082, and used by TMS34082 to output results. In host-independent mode, used by the TMS34082 for address output and data I/O.
$\overline{\text{LOE}}$	H14	I	**Local Bus Output Enable,** active low. Enables the local bus (LAD31-0) to be driven at the proper times when low. In addition, during the host-independent mode when LADCFG is low, does not affect $\overline{\text{ALTCH}}$, $\overline{\text{CAS}}$, $\overline{\text{WE}}$, CORDY, or $\overline{\text{COINT}}$. When LADCFG is high, $\overline{\text{ALTCH}}$, $\overline{\text{COINT}}$, and CORDY are not disabled by $\overline{\text{LOE}}$ high; $\overline{\text{CAS}}$ and $\overline{\text{WE}}$ are disabled.
LRDY	N13	I	**Local Bus Data Ready.** In coprocessor mode, LDRY high indicates that data is available on LAD bus. LRDY low indicates that the TMS34082 should not load data from LAD31-0. In host-independent mode, when LRDY goes low, the device is stalled until LRDY is set high again. Tied high if not used.
$\overline{\text{RAS}}$	P15	I	**Row Address Strobe,** active low. In coprocessor mode this signal is high during all coprocessor instruction cycles. Not used in host-independent mode.
SF	N14	I	**Special Function.** When high, indicates the LAD bus input is an instruction or data from TMS34020 registers. When low, indicates the LAD input is a data operand from memory. Not used in host-independent mode.
$\overline{\text{WE}}$	G13	I	**Write Enable,** active low. In coprocessor mode, the LAD bus write strobe from the TMS34020 to enable a write to or from the TMS34082 LAD bus.
		O/Z	In host-independent mode, $\overline{\text{WE}}$ is the TMS34082 data write strobe.

Table 2–4. MSD Bus Signals

Pin Name	Pin No.	I/O/Z	Description
DS/$\overline{CS}$	R13	O	**Data Space/Code Space Select**. When MEMCF is low and DS/$\overline{CS}$ is low, selects program memory on MSD port. When MEMCFG is low and DS/$\overline{CS}$ is high, selects data memory on MSD port. When MEMCFG is high, DS/$\overline{CS}$ is memory chip select, active low.
$\overline{MAE}$	N12	I	**External Memory Address and Data Output Enable**, active low. When this signal is low, the TMS34082 can output an address on MSA15-0 and data on MSD31-0. $\overline{MAE}$ high does not disable DS/$\overline{CS}$, $\overline{MCE}$, $\overline{MWR}$, or $\overline{MOE}$.
$\overline{MCE}$	R14	O	**Memory Chip Enable**. When MEMCFG is low, active (low) indicates access to external memory on MSD31-0. When MEMCFG is high, $\overline{MCE}$ low is external code memory chip select.
$\overline{MOE}$	P12	O	**Memory Output Enable**, active low. When low, enables output from external memory onto the MSD port.
MSA0	R4	O/Z	**Memory Address Bus.** Addresses up to 64K words of external program memory or up to 64K words of external data memory on the MSD port, depending on setting of DS/$\overline{CS}$ select.
MSA1	P5		
MSA2	R5		
MSA3	P6		
MSA4	R6		
MSA5	N7		
MSA6	P7		
MSA7	R7		
MSA8	P8		
MSA9	R9		
MSA10	P9		
MSA11	R10		
MSA12	R11		
MSA13	P10		
MSA14	N10		
MSA15	R12		

Table 2–4. MSD Bus Signals (Continued)

Pin Name	Pin No.	I/O/Z	Description
MSD0	C3	I/O/Z	**External Memory Data Bus.** Used to read from or write to external data or program memory.
MSD1	B1		
MSD2	D3		
MSD3	C2		
MSD4	C1		
MSD5	D2		
MSD6	D1		
MSD7	E2		
MSD8	E1		
MSD9	F2		
MSD10	F1		
MSD11	G3		
MSD12	G2		
MSD13	G1		
MSD14	H1		
MSD15	J1		
MSD16	J2		
MSD17	K1		
MSD18	L1		
MSD19	K2		
MSD20	M1		
MSD21	L2		
MSD22	N1		
MSD23	L3		
MSD24	M2		
MSD25	P1		
MSD26	N2		
MSD27	R2		
MSD28	N4		
MSD29	P3		
MSD30	R3		
MSD31	P4		
$\overline{\text{MWR}}$	P11	O	**Memory Write Enable.** When low, data on MSD31-0 can be written to external program or data memory.

Table 2–5. Clock and Control Signals

Pin Name	Pin No.	I/O/Z	Description
CC	J13	I	**Condition Code Input**. May be used as an external conditional input for branch conditions.
CID0 CID1 CID2	L14 K13 L15	I	**Coprocessor ID**. Used to set a coprocessor ID so that TMS34020 Graphics System Processor controlling multiple TMS34082s can designate which coprocessor is being selected by the current instruction. Tied low in host-independent mode.
CLK	J15	I	**System Clock** in host-independent mode. Tied low in coprocessor mode.
$\overline{\text{COINT}}$	E15	O	**Coprocessor Interrupt Request**, active low. In coprocessor mode, signals an exception not masked out in the configuration register. Remains low until the status register is read. In host-independent mode, user programmable I/O when LADCFG is low. Designates bus cycle boundaries on LAD31-0 when LADCFG is high.
CORDY	F13	O	**Coprocessor Ready**. In coprocessor mode, if the TMS34020 sends an instruction before the TMS34082 has completed a previous instruction, this signal goes low to indicate that the TMS34020 should wait. User-programmable in host-independent mode.
INTG	P13	O	**Interrupt Grant**. This signal is set high to acknowledge an interrupt request input in host-independent mode.
$\overline{\text{INTR}}$	K14	I	**Interrupt Request**, active low. Causes call to subroutine address in interrupt vector register in host-independent mode. Tied high in coprocessor mode.
LCLK1 LCLK2	M14 M15	I	**Local Clock 1 and 2,** generated by the TMS34020, 90 degrees out of phase, to provide timing inputs to TMS34082 in coprocessor mode. Tied low in host-independent mode.
MSTR	J14	I	**Coprocessor/Host-Independent Mode Select**. When low, puts the TMS34082 in coprocessor mode. When high, puts the TMS34082 in host-independent mode.
RDY	K15	I	**Ready**. When RDY is low, causes a nondestructive stall of sequencer and floating-point operations. All internal registers and status in the FPU core are preserved. Also, no output lines will change state.
$\overline{\text{RESET}}$	N15	I	**Reset**, active low. Resets sequencer output and clears pipeline registers, internal states, status, and exception disable registers in FPU core. Other registers are unaffected.

Table 2–6. Emulation Control Signals

Pin Name	Pin No.	I/O/Z	Description
EC0 EC1	G14 G15	I	**Emulator Mode Control and Test**. Tied high for normal operation.
TCK	R8	I	**Test Clock** for JTAG 4-wire boundary scan. Tied low for normal operation.
TDI	H15	I	**Test Data Input** for JTAG 4-wire boundary scan. May be left floating.
TDO	H2	O	**Test Data Output** for JTAG 4-wire boundary scan.
TMS	B8	I	**Test Mode Select** for JTAG 4-wire boundary scan. May be left floating.

Table 2–7. Power and N/C Signals

Pin Name	Pin No.	Description
NC	A1	No internal connection. These pins should be left floating.
NC	A15	
NC	B2	
NC	B14	
NC	D4	
NC	P2	
NC	R1	
NC	R15	
V_{CC}	C5	5-V power supply. All pins must be connected and used.
V_{CC}	C8	
V_{CC}	C10	
V_{CC}	D13	
V_{CC}	F3	
V_{CC}	J3	
V_{CC}	M13	
V_{CC}	N3	
V_{CC}	N6	
V_{CC}	N9	
V_{SS}	C4	Ground pins. All pins must be connected and used.
V_{SS}	C7	
V_{SS}	C9	
V_{SS}	C12	
V_{SS}	E3	
V_{SS}	E13	
V_{SS}	H3	
V_{SS}	H13	
V_{SS}	K3	
V_{SS}	L13	
V_{SS}	M3	
V_{SS}	N5	
V_{SS}	N8	
V_{SS}	N11	

Chapter 3

Data Formats

The TMS34082 accepts operands as either:

- IEEE floating-point numbers (IEEE Standard 754-1985)
- Unsigned 32-bit integers
- 32-bit 2s-complement signed integers

Floating-point operands may be either single-precision (32 bits) or double-precision (64 bits). All internal integer instructions use signed integer data formats.

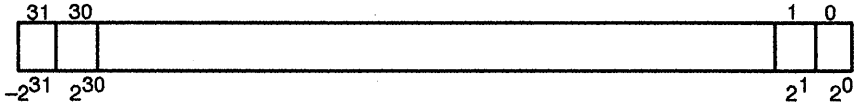
3.1 Integer Formats

The TMS34082 recognizes two types of integers: signed and unsigned. Only one type may be used in a single instruction. Internal instructions use only signed integers.

3.1.1 Signed Integers

A signed integer is a 32-bit value in 2s-complement format, as shown below. The most significant bit is the sign bit; a 1 signifies a negative number. Signed integers can represent values from –2,147,438,648 to +2,147,438,647.

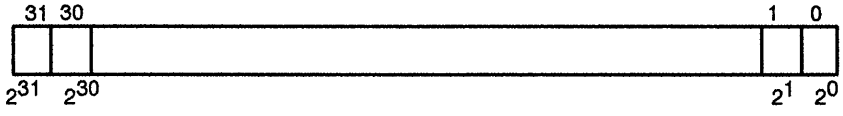
Figure 3–1. IEEE Signed Integer Format



3.1.2 Unsigned Integer

An unsigned integer is also a 32-bit value, but can only represent positive numbers. The range for unsigned integers is 0 to 4,294,967,295.

Figure 3–2. IEEE Unsigned Integer Format



3.2 Floating-Point Formats

IEEE formats for floating-point operands, both single- and double-precision, consist of three fields; the sign (s), the exponent (e), and the fraction (f), in that order. The most significant bit is the sign bit. The value of the mantissa contains a hidden bit, an implicit leading 1, as shown below:

1.fraction

The representation of a normalized floating-point number is:

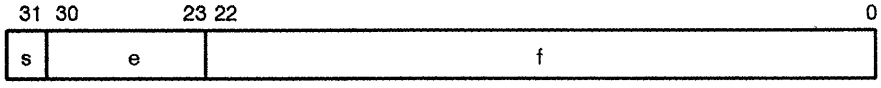
$(-1)^s \times 1.f \times 2^{(e\text{-}bias)}$

The bias is a number added to the true exponent to ensure that the exponent (e) is always positive. The bias is 127 for single-precision or 1023 for double-precision. Further details of IEEE formats and exceptions are covered in the IEEE Standard for Binary Floating-Point Arithmetic, IEEE Standard 754-1985.

3.2.1 Single-Precision Floating-Point

Single-precision floating-point numbers are 32 bits long; the exponent field is 8 bits, and the fraction field is 23 bits. The exponent is biased by 127. Single precision can represent values from $\pm 2^{-126}$ to $\pm 2^{127} \times (2-2^{-23})$. That is approximately $\pm 1.2 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$. The format for a single-precision number is shown in Figure 3–3.

Figure 3–3. IEEE Single-Precision Format

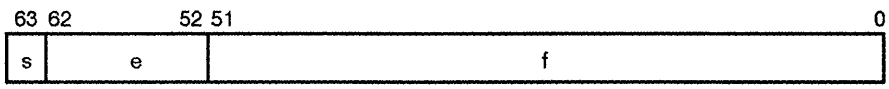


s: sign of fraction
e: 8-bit exponent, biased by 127 (true exponent + 127)
f: 23-bit fraction

3.2.2 Double-Precision Floating-Point

A double-precision floating-point number is a 64-bit value. The exponent field is 11 bits, biased by 1023, and the fraction field is 52 bits. The range for double-precision is $\pm 2^{-1022}$ to $\pm 2^{1023} \times (2-2^{-52})$, or approximately $\pm 2.2 \times 10^{-308}$ to $\pm 1.8 \times 10^{308}$.

Figure 3–4. IEEE Double-Precision Format



s: sign of fraction
e: 11-bit exponent, biased by 1023 (true exponent + 1023)
f: 52-bit fraction

3.2.3 Denormal and Wrapped Numbers

The TMS34082 also handles two other data formats that permit operations on very small floating-point numbers. Denormalized and wrapped floating-point numbers represent the same values, but in different formats. If very small values can be approximated by 0 in your application, you can set the Fast bit in the configuration register to force all denormal and wrapped inputs and outputs to 0.

The ALU accepts denormalized numbers, that is, floating-point numbers so small that they cannot be normalized. A denormalized number results from decrementing the biased exponent field to 0 before normalization is complete. A denormal has the form of a floating-point number with a 0 exponent, a nonzero fraction, and a 0 in the leftmost (hidden) bit of a mantissa.

A single-precision denormalized number is equal to the following:

$$(-1)^s \times (2)^{-126} \times 0.f$$

For double-precision, a denormal is equal to the following:

$$(-1)^s \times (2)^{-1022} \times 0.f$$

If denormalized numbers are input to the multiplier, they will cause status exceptions. Denormals can be passed to the ALU to be *wrapped*. The wrapped operand is then input to the multiplier.

A wrapped number is a number created by normalizing a denormalized number's fraction field and subtracting from the exponent the number of shift positions (minus one) required to do so. The exponent is encoded as a 2s-complement negative number. When the mantissa of the denormal is normalized by shifting it left, the exponent field decrements from all 0s (wraps past 0) to a negative 2s-complement number (except in the case of 0.1xxx . . . , where the exponent is not decremented).

3.2.4 Special Floating-Point Numbers

There are three other special floating-point value representations (see Figure 3–5):

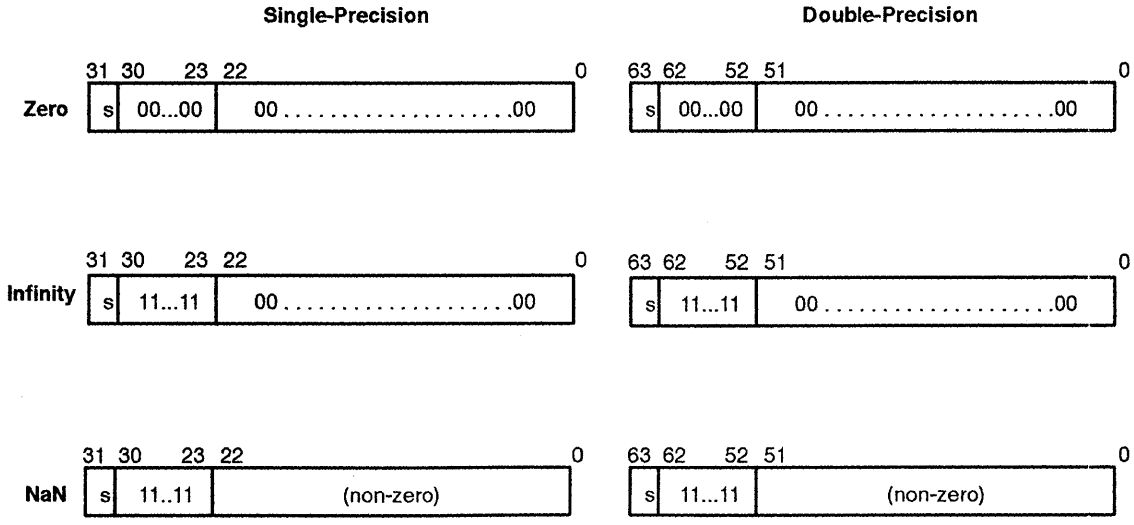
Zero (positive or negative) is represented by the appropriate sign bit, a 0 exponent field, and a 0 fraction field.

Infinity (positive or negative) is represented by the appropriate sign bit, 1s in the exponent field, and a 0 fraction field.

A Not a Number (NaN) designates data that has no mathematical value. A NaN has 1s in the exponent field with a nonzero fraction.

A NaN is produced whenever an invalid operation (such as division by 0) is executed. The TMS34082 treats all NaNs as signaling NaNs, setting the invalid (I) flag in the status register. The TMS34082 outputs all NaNs (regardless of input form) with a 0 sign bit and all 1s in the exponent and fraction fields.

Figure 3–5. Special Floating-Point Formats



3.2.5 Range of Floating-Point Numbers

Table 3–1 shows the range of possible single- and double-precision floating-point numbers.

Table 3–1. Floating-Point Number Representations

Type	Sign	Exponent	Hidden Bit	Fraction
NaNs	0	11 .. 11		11 .. 11
		: :	1	: :
	0	11 .. 11		10 .. 00
	0	11 .. 11		01 .. 11
		: :	1	: :
	0	11 .. 11		00 .. 01
Positive Infinity	0	11 .. 11	1	00 .. 00
Positive Normals	0	11 .. 10		11 .. 11
		: :	1	: :
	0	00 .. 01		00 .. 00
Positive Denormals	0	00 .. 00		11 .. 11
		: :	0	: :
	0	00 .. 00		00 .. 01
Zero (Positive)	0	00 .. 00	1	00 .. 00
Zero (Negative)	1	00 .. 00	1	00 .. 00
Negative Denormals	1	00 .. 00		00 .. 01
		: :	0	: :
	1	00 .. 00		11 .. 11
Negative Normals	1	00 .. 01		00 .. 00
		: :	1	: :
	1	11 ..10		11 .. 11
Negative Infinity	1	11 .. 11	1	00 .. 00
NaNs	1	11 .. 11		00 .. 01
		: :	1	: :
	1	11 .. 11		01 .. 11
	1	11 .. 11		10 .. 00
		: :	1	: :
	1	11 ..11		11 .. 11
	Single:	< 8 bits >		<– 23 bits – >
	Double:	< 11 bits >		<– 52 bits – >

Chapter 4

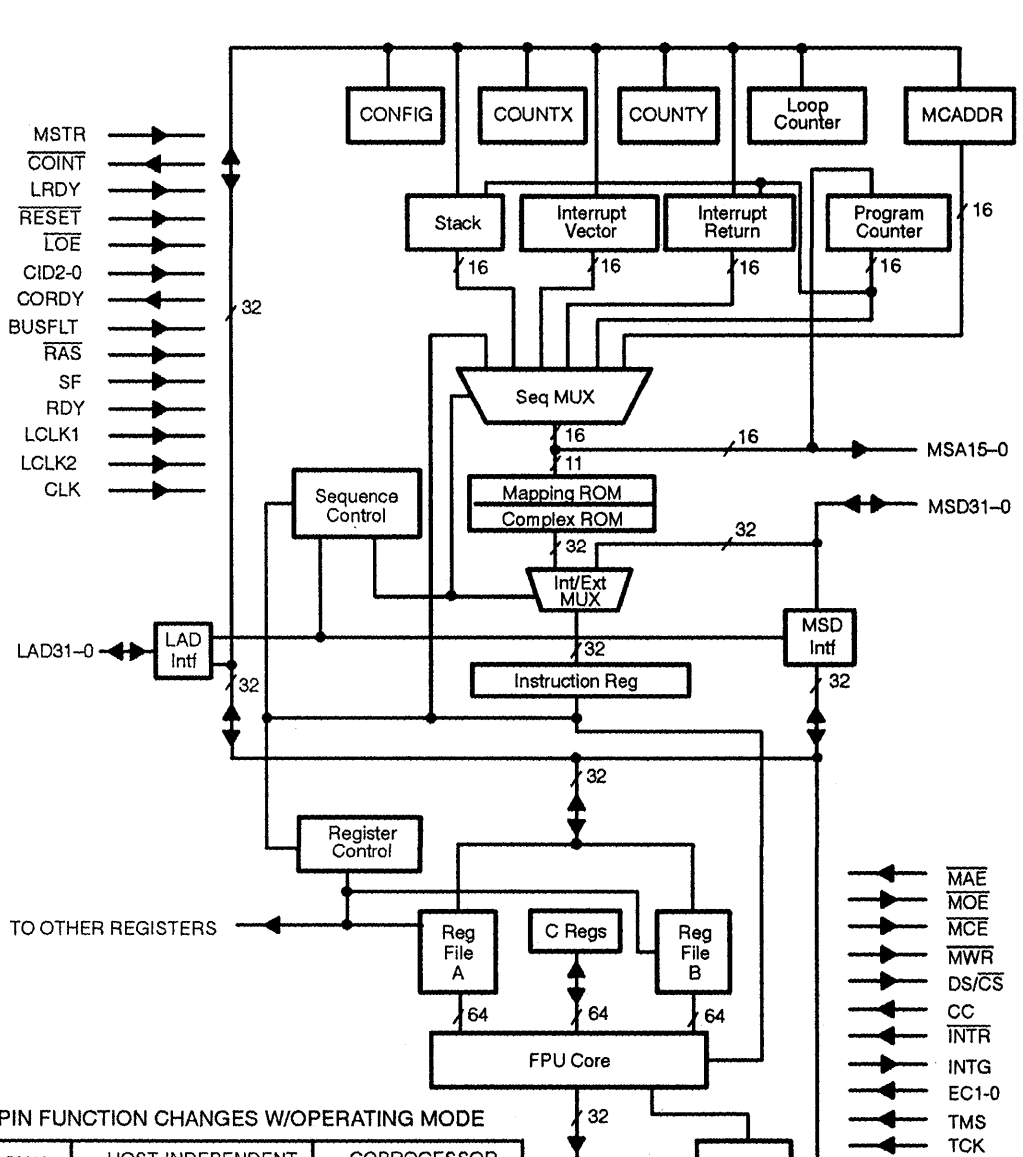
Architecture

Because the sequencer, control and data registers, and FPU core are closely coupled, the TMS34082 can execute a wide variety of complex floating-point or integer calculations rapidly with a minimum of external data transfers. The internal architecture of the FPU core supports concurrent operation of the multiplier and the ALU, providing several options for storing or feeding back intermediate results. Also, several special registers are available to support calculations for graphics algorithms. Each of the main architectural elements of the TMS34082 is discussed in this chapter.

4.1 Functional Block Diagram

The main architectural features of the TMS34082 are illustrated in Figure 4–1.

Figure 4–1. Functional Block Diagram



PIN FUNCTION CHANGES W/OPERATING MODE

SIGNAL NAME	HOST-INDEPENDENT MODE	COPROCESSOR MODE
ALTCH	Output	Input
WE	Output	Input
CAS	Output	Input

4.2 Operating Modes

The TMS34082 has two operating modes: coprocessor mode and host-independent mode.

In coprocessor mode, the TMS34082 acts as a floating-point coprocessor to the TMS34020 Graphics System Processor. The TMS34082 is a direct extension of the TMS34020 and its instruction set. Operation in coprocessor mode is signaled by tying the MSTR input low. Chapter 5 details this operating mode.

In host-independent mode, the TMS34082 is a floating-point RISC processor. It may be used as a coprocessor to another host processor, as a parallel processor, or as a stand-alone processor. To operate in host-independent mode, the MSTR input must be high. This mode is covered in Chapter 6.

4.3 Bus Interfaces

The TMS34082 has two buses: the LAD (LAD31-0) and the MSD (MSD31-0). Each is a 32-bit bidirectional bus which can be used to transfer instructions and/or data.

One 32-bit operand can be input to the TMS34082 data registers each cycle. A 64-bit double-precision floating-point operand is input in two cycles. Transfers to and from the data registers can normally be programmed as block moves (loading one or more sets of operands with a single move instruction to minimize I/O overhead). Block transfers up to 512 words in length can be programmed in either direction between the LAD and MSD buses.

4.3.1 LAD Bus

When the TMS34082 is used as a coprocessor to the TMS34020, the LAD bus is the main interface between the two devices. Both data and instructions from the TMS34020 are input on the LAD bus. The data can be stored in internal registers or transferred to memory on the MSD port. In addition, data (from registers or the MSD bus) can be sent to the TMS34020.

With a single TMS34020 instruction, the TMS34020 can transfer both an instruction and data to the TMS34082. Data may be from TMS34020 registers or the local memory controlled by the TMS34020.

In host-independent mode, the LAD bus is used as a data bus. Instructions may not be input on the LAD bus. However, data (an address) may be read from the LAD port to an internal register, and a jump to that address performed.

To permit direct input to or output from the LAD bus, other options are available for control of the bus in host-independent mode. When two 32-bit operands are selected for input to the FPU core, one operand may come directly from the LAD bus. A result from the FPU core may simultaneously be written to a data register and the LAD bus.

The main control signals for the LAD bus are:

$\overline{\text{ALTCH}}$

$\overline{\text{CAS}}$

$\overline{\text{WE}}$

$\overline{\text{LOE}}$

SF (coprocessor mode only).

The function of these signals depends upon the operating mode and are discussed further in Chapters 5 — Coprocessor Mode — and Chapter 6 — Host-Independent Mode.

4.3.2 MSD Bus

The MSD bus (MSD31-0) and its associated address bus (MSA15-0) are the external memory interface for the TMS34082. Control signals allow you to have separate code and data storage on the MSD port. Up to 64K 32-bit words of code space and 64K words of data space are directly supported. The bus and control signals are optimized for use with static RAM (SRAM) memory. However, with some external logic, this bus may also be connected to DRAMs, VRAMs, or other system buses.

The MSD bus is the main instruction source in host-independent mode. Data may also be accessed on this port. The TMS34082 can operate with the LAD bus as its single data bus and the MSD bus as the instruction source, or with data storage on both ports and the program memory on the MSD port.

In coprocessor mode, use of the MSD bus is optional. External user-generated subroutines may be accessed via the MSD bus. In addition, data for these routines may be stored in memory on the MSD port. The code and data for these subroutines may be downloaded from the TMS34020 memory using an LAD-to-MSD move.

MSD bus control is the same in both coprocessor and host-independent modes. Control signals are summarized in Table 4–1. Different combinations of $\overline{\text{MCE}}$, $\overline{\text{MWR}}$, and $\overline{\text{MOE}}$ distinguish between memory reads and writes. Table 4–2 lists the memory operation performed for each combination of signals.

Table 4–1. MSD Bus Control Signals

Name	Function
MSA15-0	Memory Address Output
DS/$\overline{\text{CS}}$	Data Space/Code Space Select. This signal goes low to select program memory or high to select data memory.
$\overline{\text{MCE}}$	Memory Chip Enable. This signal goes low when reading from or writing to memory.
$\overline{\text{MOE}}$	Memory Output Enable. This signal goes low when reading from memory.
$\overline{\text{MWR}}$	Memory Write Enable. This signal goes low when writing to memory.
$\overline{\text{MAE}}$	MSD Bus Enable. When this input is low, the TMS34082 can output data and address on MSD and MSA.

Table 4–2. Memory Operations on MSD

$\overline{\text{MCE}}$	$\overline{\text{MWR}}$	$\overline{\text{MOE}}$	Memory Operation
0	0	0	Invalid
0	0	1	Write to memory
0	1	0	Read from memory
0	1	1	Invalid
1	x	x	No memory access

The DS/$\overline{CS}$ output acts as the most significant address bit selecting between code and data memory. If a single block of memory is used for both code and data space, this output may be ignored. Without DS/$\overline{CS}$, only 64K words of memory can be accessed.

An alternate control scheme is chosen by setting the MEMCFG bit in the configuration register high. Then, DS/$\overline{CS}$ is the data space chip enable and $\overline{MCE}$ is the code space chip enable. Refer to subsection 4.5.3.3 — MSD Bus Configuration — for more information.

If the memory on the MSD port is shared with another processor, $\overline{MAE}$ may be used to prevent bus conflicts. When memory on the MSD port is shared, the host processor can monitor the state of the memory chip enable ($\overline{MCE}$) to determine when the TMS34082 is accessing memory.

Otherwise, $\overline{MAE}$ may be tied low. The TMS34082 will only drive the MSD bus when writing to memory (signaled by $\overline{MWR}$ low).

4.4 Sequence Control

The sequencer selects the next program execution address either from internal code or from external program memory. Next address sources include:

Program counter

Instruction register

Stack

Interrupt vector register

Interrupt return register

Indirect address register

The two-deep stack is used to store return addresses for jump-to-subroutine instructions. When the TMS34082 receives an interrupt, the sequencer jumps to the interrupt service routine at the address given by the interrupt vector register. The interrupt return register stores the address where execution resumes after the interrupt routine is completed. The indirect address register is used for indirect branches and jumps to subroutines.

The sequencer allows many options for program execution control. Branches on status, conditional and unconditional jumps to subroutines, counted loops, and interrupt service routines may be programmed.

4.5 Registers

The TMS34082 contains:

Twenty 64-bit general-purpose registers

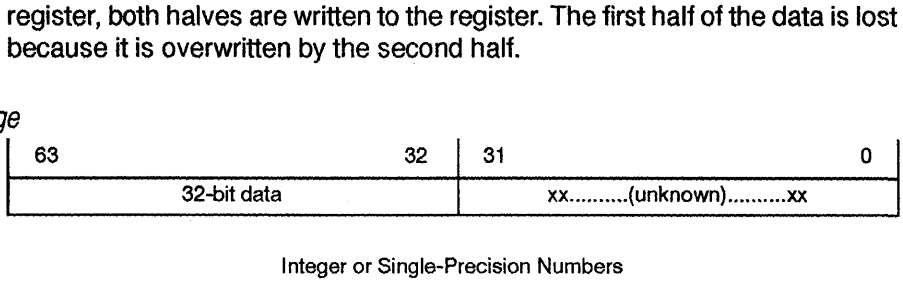
Two embedded 64-bit feedback registers

Ten control registers

Control registers are 17 to 32 bits long as shown in the register model in Figure 4–3. The 32-bit control registers COUNTX, COUNTY, and MIN-MAX/LOOPCT are used for internal graphics instructions. When you are not using these instructions, the registers are available for temporary storage.

32-bit single-precision floating-point or integer data is stored in the upper half (bits 63-32) of a register as shown in Figure 4–2. Double-precision data uses the complete 64-bit register. If a double-precision number is loaded into a 32-bit register, both halves are written to the register. The first half of the data is lost because it is overwritten by the second half.

Figure 4–2. Register Usage



Register files RA and RB can be written to or read from the external buses as can the control registers. Internal registers C and CT are embedded in the FPU core and can only be accessed by the FPU internal buses. The C and CT registers cannot be used as sources or destinations for move instructions. Several other registers are not available as sources for FPU operations as listed in Table 4–3.

Block moves begin at the register address given in the instruction and sequence through the registers in the order shown in the register model, Figure 4–3. C and CT are omitted from the sequence because they cannot be accessed by the external buses. After the last register address (MIN-MAX/LOOPCT), the sequence starts again at address 0 (RA0).

Figure 4–3. TMS34082 Register Model

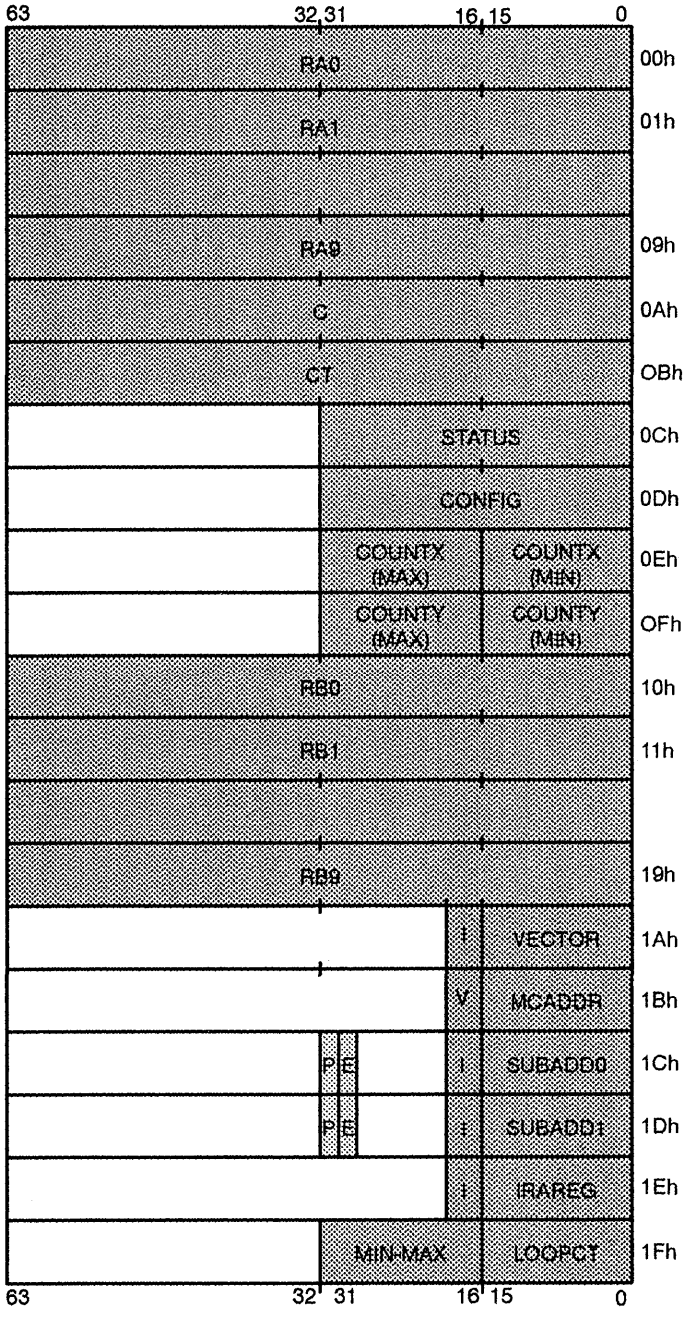


Table 4–3. Internal Registers

Address	Register	Restrictions on Use
00000	RA0	
00001	RA1	
00010	RA2	
00011	RA3	
00100	RA4	
00101	RA5	
00110	RA6	
00111	RA7	
01000	RA8	
01001	RA9	
01010	C	Not a source or destination for external moves. C and CT cannot both be used as operands in the same instruction.
01011	CT	Not a source or destination for external moves. C and CT cannot both be used as operands in the same instruction.
01100†	STATUS	Not a source for FPU instructions
01101‡	CONFIG	Not a source for FPU instructions
01110‡	COUNTX	Not a source for FPU instructions
01111‡	COUNTY	Not a source for FPU instructions
10000	RB0	
10001	RB1	
10010	RB2	
10011	RB3	
10100	RB4	
10101	RB5	
10110	RB6	
10111	RB7	
11000	RB8	
11001	RB9	
11010	VECTOR	Not a source for FPU instructions
11011	MCADDR	Not a source for FPU instructions
11100†	SUBADD0	Not a source for FPU instructions
11101‡	SUBADD1	Not a source for FPU instructions
11110‡	IRAREG	Not a source for FPU instructions
11111‡	MIN-MAX/LOOPCT	Not a source for FPU instructions

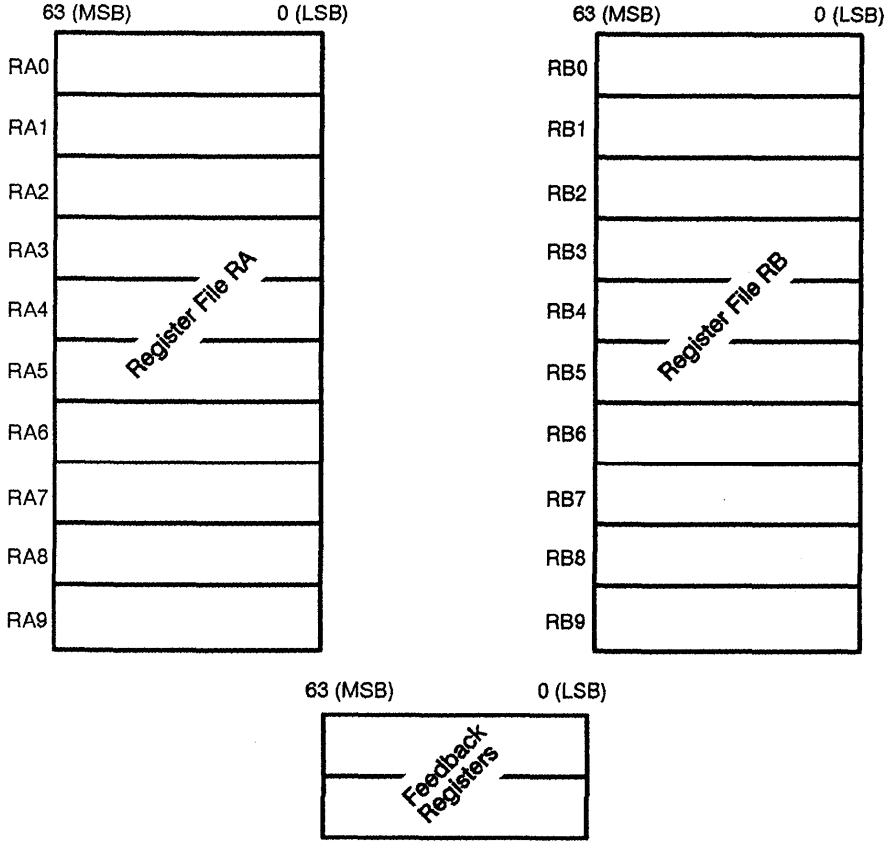
† Using this address as a source register in external code inputs data directly from the LAD bus to the FPU. Only valid in host-independent mode.

‡ Using this address as a source register in external code inputs the value one of the appropriate format (integer, single-, or double-precision) to the FPU.

4.5.1 Register Files RA and RB

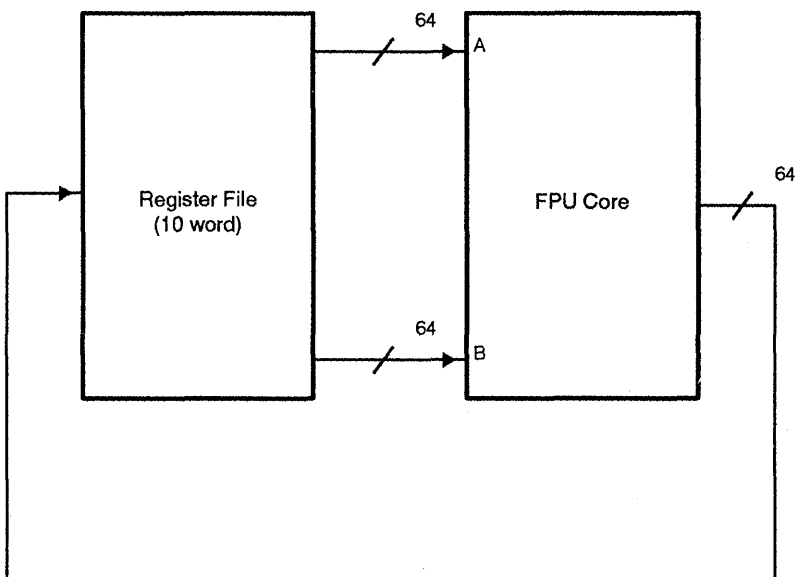
The TMS34082 contains two register files, each with ten 64-bit registers. Most instructions operate on one value from each of the RA and RB register files and return the result to any register. Figure 4–4 illustrates the general-purpose registers of the TMS34082.

Figure 4–4. General-Purpose Registers



When the ONEFILE control bit is set high in the configuration register, data written to a register in RA file is simultaneously written to the corresponding location in RB file. For example, the same data is written to both RA1 and RB1 at once. In this mode the two register files act as a ten-word, two-read/one-write register file, as shown in Figure 4–5.

Figure 4–5. Register Files with ONEFILE High



4.5.2 Feedback Registers C and CT

The two 64-bit feedback registers, C and CT, are embedded in the FPU core. Data is stored in the C and CT registers in an unpacked format. That is, integer and single-precision numbers are not stored in the upper 32-bits of the registers, but aligned in fields throughout the 64 bits. *For this reason, you should always make sure the data type in the instruction matches the actual data in the register.*

C or CT can be used as one or both operands in an instruction, but may not be used together in the same instruction. For example, C + CT is not valid, but C + C is. The feedback registers may not be accessed for external moves.

The CT feedback register is used in integer divide and square root operations as a temporary holding register. *Any data stored in CT will be lost during an integer divide or square root.*

4.5.3 Configuration Register (CONFIG)

The configuration register (CONFIG) is a special 32-bit register which you load to set up the following TMS34082 functions:

Exception handling

Bus configurations

Pipeline configurations

Denormalized number handling

Data transfer operations

Rounding modes

The configuration register is initialized to FFE00020h. Writing to this register during a block move will not change the operation of LADCFG, MEMCFG, and LOAD until the move is complete. There is a one-cycle delay from when a new value is moved to the configuration register until that value takes effect. If the instruction following a move to the configuration register requires the new setting of the register to be valid, insert one nop (No Operation) instruction after the move.

The format of the configuration register is given in Table 4–4.

Table 4–4. Configuration Register Definition

Bit No.	Name	Description
31	MIVAL	Multiplier invalid operation (I) exception mask. Initialized to one (enabled).
30	MOVER	Multiplier overflow (V) exception mask. Initialized to one (enabled).
29	MUNDER	Multiplier underflow (U) exception mask. Initialized to one (enabled).
28	MINEX	Multiplier inexact (X) exception mask. Initialized to one (enabled).
27	MDIV0	Divide by zero (DIV0) exception mask. Initialized to one (enabled).
26	MDENORM	Multiplier wrapped number output (DENORM) exception mask. Initialized to one (enabled).
25	AIVAL	ALU invalid operation (I) exception mask. Initialized to one (enabled).
24	AOVER	ALU overflow (V) exception mask. Initialized to one (enabled).
23	AUNDER	ALU underflow (U) exception mask. Initialized to one (enabled).
22	AINEX	ALU inexact (X) exception mask. Initialized to one (enabled).
21	ADENORM	ALU denormal output (DENORM) exception mask. Initialized to one (enabled).
20-11	N/A	Reserved for later use. Initialized to all zeros.
10	VERSION	Version number, read only. Set to one.
9	LADCFG	LAD bus configuration for host-independent mode. When high, $\overline{\text{COINT}}$ defines LAD bus cycle boundaries. The setting of this bit has no effect in coprocessor mode. Initialized to zero.
8	MEMCFG	MSD bus configuration. When high, $\overline{\text{MCE}}$ and DS/$\overline{\text{CS}}$ are code and data space chip enable, respectively. Initialized to zero.
7	N/A	Reserved for later use. Initialized to zero. Note: You must *always* write a zero to this bit.
6	ONEFILE	When high, causes simultaneous write to both register files. Initialized to zero.
5	PIPES2	When high, makes the FPU core output registers transparent. When low, the output registers are enabled. Initialized to one.
4	PIPES1	When high, makes the FPU core internal pipeline registers transparent. When low, the FPU internal pipeline registers are enabled. Initialized to zero.
3	FAST	When high, Fast mode is selected (all denormalized inputs and outputs are zeroed). When low, IEEE mode is selected. Initialized to zero.
2	LOAD	Load order. 0 = MSH, then LSH; 1= LSH, then MSH. Initialized to zero.
1	RND1	Rounding mode select 1. Initialized to zero.
0	RND0	Rounding mode select 0. Initialized to zero.

4.5.3.1 Exception Mask

The mask bits (bits 31-21) serve as exception detect enables. Setting bits high enables the detection of the specific exceptions. Exceptions that are unimportant to your specific application may be masked to prevent unwanted interrupts. When an enabled exception occurs, the ED bit in the status register is set high and can be used to generate interrupts.

When the exception mask has been loaded, the mask is applied to the contents of the status register to disable unnecessary exceptions. Status results are ORed together and, if true, the exception detect (ED) status bit is set high. Individual status flags remain active and can be read independently of mask operations.

Since inexact results are normal for floating-point operations, you should usually mask out this exception for both the ALU (AINEX) and multiplier (MINEX).

4.5.3.2 LAD Bus Configuration (Host-Independent Mode)

The LADCFG bit (bit 9) defines the LAD bus configuration for host-independent mode. Two different configurations are possible.

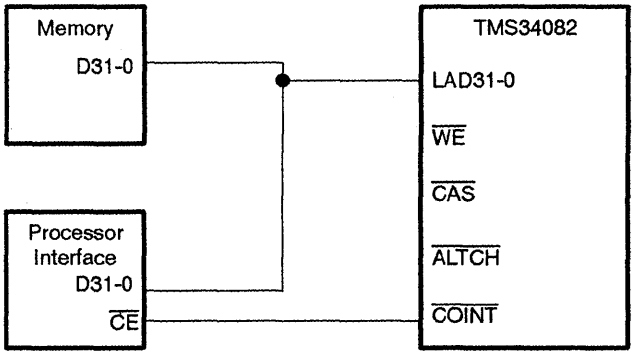
When LADCFG is low, $\overline{\text{COINT}}$ is a user-programmable signal not associated with the LAD bus. $\overline{\text{CAS}}$ and $\overline{\text{WE}}$ are not affected by $\overline{\text{LOE}}$ (LAD bus enable).

When LADCFG is high, $\overline{\text{COINT}}$ defines LAD bus cycle boundaries and is controlled by bit 1 (C bit) of LAD move instructions. Also, $\overline{\text{CAS}}$ and $\overline{\text{WE}}$ are disabled (placed in a high impedance state) when $\overline{\text{LOE}}$ is high.

With LADCFG high, a move instruction with the C bit high sets $\overline{\text{COINT}}$ low before the first word is moved. $\overline{\text{COINT}}$ remains low until the move is complete. You could use $\overline{\text{COINT}}$ to select between two devices on the LAD bus. $\overline{\text{COINT}}$ becomes the chip enable for one of the devices as shown in Figure 4–6.

The setting of $\overline{\text{COINT}}$ has no effect in coprocessor mode.

Figure 4–6. Host-Independent Mode LAD Bus Configuration for LADCFG high

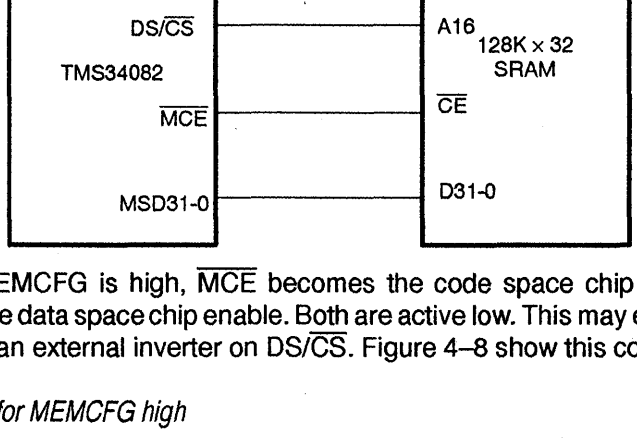


4.5.3.3 MSD Bus Configuration

The MEMCFG bit defines the function of control signals for the MSD bus. Two different configurations are possible.

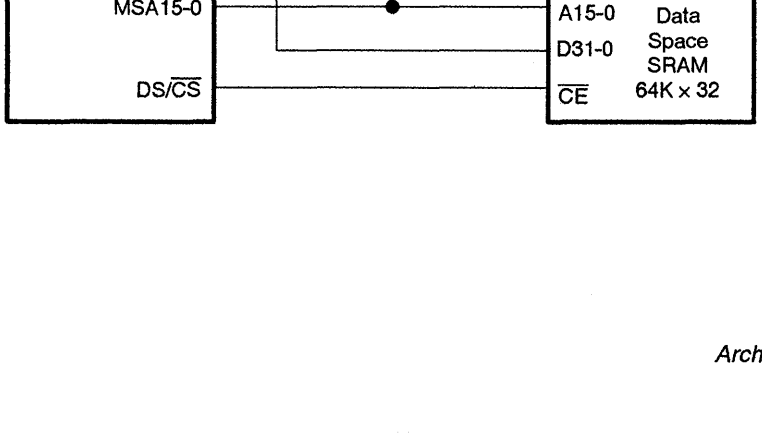
When MEMCFG is low, $\overline{\text{MCE}}$ is the memory chip enable signal. It goes low when memory is being accessed. DS/$\overline{\text{CS}}$ functions as the most significant address bit, selecting data memory when high or code memory when low. This configuration is illustrated in Figure 4–7.

Figure 4–7. MSD Bus Configuration for MEMCFG low



When MEMCFG is high, $\overline{\text{MCE}}$ becomes the code space chip enable and DS/$\overline{\text{CS}}$ the data space chip enable. Both are active low. This may eliminate the need for an external inverter on DS/$\overline{\text{CS}}$. Figure 4–8 show this configuration.

Figure 4–8. MSD Bus Configuration for MEMCFG high



4.5.3.4 Pipeline Settings

The PIPES2 and PIPES1 bits (bits 5-4) of the configuration register define the piepline register settings for the internal FPU core. PIPES2 is the enable for the FPU core output registers; PIPES1 is for the FPU core internal pipeline registers. Both are active low. When high, data flows through the registers. Table 4–5 details the pipeline operation for each setting of PIPES.

Table 4–5. Pipeline Settings

PIPES2	PIPES1	Operation
0	0	Both pipeline registers enabled
0	1	Only FPU internal pipeline registers enabled
1	0	Only FPU output registers enabled
1	1	Both pipeline registers disabled (flowthrough)

For more information on pipeline registers, refer to subsection 4.6.2 — Pipeline Registers.

4.5.3.5 Fast and IEEE Modes

The FAST bit (bit 3) selects the mode for handling denormalized inputs and outputs. For many applications, very small numbers may be treated as zero, allowing the programmer to use Fast mode. In the Fast mode (FAST=1):

All denormalized or wrapped inputs and outputs are forced to zero and do not cause any status exceptions.

The DENIN (denormal input) status exception is disabled.

Using Fast mode simplifies error handling because you do not have to wrap and unwrap denormalized numbers. Forcing very small (denormalized) numbers to zero causes a loss of accuracy, however. If you multiply a very large number by a denormal, the result may be significantly larger than zero. If it is important in your application to distinguish between very small numbers and zero, use IEEE mode.

Setting FAST = 0 selects IEEE mode. In this mode, the ALU can operate on denormalized inputs and return denormals. Denormals are not valid input to the multiplier; they must be wrapped first. If you input a denormal to the multiplier, the DENIN flag will be asserted and the result will be invalid (I flag set). Exponent underflow is possible during multiplication of small operands even when the operands are not wrapped numbers. If the multiplier result underflows, a wrapped number will be output. In IEEE mode, the wrapped number is not forced to zero.

When the multiplier produces a wrapped number as its result, it may be passed to the ALU and unwrapped. A zero is output if the wrapped result is too small to represent as a denormal (smaller then the minimum denormal). Table 4–6 describes how you should unwrap multiplier results and the status flags that are set when wrapped numbers are output from the multiplier.

Table 4–6. Handling Wrapped Multiplier Outputs

Type of Result	Status Bits			Notes
	DENORM	X	RND	
Wrapped, exact	1	0	0	Unwrap with *Wrapped, exact* instruction
Wrapped, inexact	1	1	0	Unwrap with *Wrapped, inexact* instruction
Wrapped, increased in magnitude	1	1	1	Unwrap with *Wrapped, rounded* instruction

4.5.3.6 Load Order

Since 64-bit double-precision data must be transferred 32 bits at a time, the TMS34082 must know which half of the word is loaded first. The LOAD bit (bit 2) defines the expected order. If LOAD = 0, the most significant half (MSH) is transferred first, followed by the least significant half (LSH). When LOAD = 1, the LSH is transferred first. The LOAD bit also determines the order data is read out of a register. Table 4–7 shows the load order for all data formats.

Table 4–7. Data Ordering for Loads/Stores

Data Format	Size	Words Accessed	
		CONFIG LOAD bit=0 31_____0	CONFIG LOAD bit =1 31_____0
Integer	32 bits	Word 0	Word 0
Single-precision	32 bits	Word 0	Word 0
Double-precision	64 bits	Word 1 (MSH) Word 0 (LSH)	Word 0 (LSH) Word 1 (MSH)

4.5.3.7 Rounding Modes

The TMS34082 supports the four IEEE standard rounding modes:

Round to nearest

Round towards zero (truncate)

Round towards positive infinity (round up)

Round towards minus infinity (round down)

The rounding function is selected by bits RND1 and RND0 as shown in Table 4–8. The default setting is round to nearest.

Table 4–8. Rounding Modes

RND1	RND0	Rounding Modes
0	0	Round towards nearest
0	1	Round towards zero (truncate)
1	0	Round towards infinity (round up)
1	1	Round towards negative infinity (round down)

You should select the rounding mode which will minimize procedural errors. Rounding to nearest introduces an error no more than half of the least significant bit. Since rounding to nearest may involve rounding either up or down in successive steps, rounding errors tend to cancel each other.

In contrast, directed rounding modes may introduce errors approaching one bit for each rounding operation. Rounding errors may accumulate rapidly, particularly with single-precision operations.

4.5.4 Status Register

The floating-point status register (STATUS) is a 32-bit register used for reporting the exceptions that occur during TMS34082 operations and status codes set by the results of implicit and explicit compare operations. The status register is cleared upon reset, except for the INTENED flag which is set to one in coprocessor mode.

The status register can be used by test-and-branch instructions to control program flow. Because of the large number of FPU status outputs, branches on status can be used to save program execution time. The status register contents are also important when dealing with status exceptions including such conditions as overflow, underflow, invalid operations, or illegal data formats (such as infinity, Not a Number (NaN), or denormalized operands).

Table 4–9. Status Register Definition

Bit No.	Name	Description
31	N	Sign bit. When high, the result is negative. (A < B for compare operations)
30	GT	A > B (valid only for compare operations)
29	Z	zero flag. (A = B for compare operations)
28	V	IEEE Overflow flag. The result is greater than the largest allowable value for the specified format.
27	I	IEEE Invalid Operation flag. A NaN has been input to the FPU or an invalid operation has been requested. If I goes high because a NaN was input, the STX flags indicate which port had the NaN.
26	U	IEEE Underflow flag. The result is inexact and less than the minimum allowable value for the specified format. In Fast mode, this condition causes a zero result.
25	X	IEEE inexact flag. The result of an operation is inexact.
24	DIV0	Divide by zero. An invalid operation involving a zero divisor has been detected by the multiplier.
23	RND	The mantissa of a number has been increased in magnitude by rounding. If the number generated was wrapped, then the *unwrap, rounded* instruction must be used to properly unwrap the wrapped number (see Table 4–6).
22	DENIN	The input to the multiplier is a denormal number. When DENIN goes high, the STX flags indicate which port had the denormal input.
21	DENORM	The multiplier output is a wrapped number or the ALU output is a denormal number. In the Fast mode, this condition causes the result to go to zero. It also indicates an invalid integer operation, for example, PASS (–A) with unsigned integer operand.
20	STX1	A NaN or a denormal has been input on the A port.
19	STX0	A NaN or a denormal has been input on the B port.
18	ED	Exception detect status signal representing logical OR of all enabled exceptions in the exception disable register.
17	UNORD	The two inputs of a comparison operation are unordered, that is, one or both of the inputs is a NaN.
16	INTFLG	Software interrupt flag. Set by external code to signal a software interrupt.
15	INTENHW	Hardware interrupt ($\overline{\text{INTR}}$) enable
14	NXOROV	N (negative) XOR V (overflow)
13	VANDZB	V(overflow) AND NOTZ (not zero)
12	INTENED	ED interrupt enable (initialized to zero in host-independent mode, one in coprocessor mode).
11	INTENSW	Software interrupt enable for INTFLG (bit 16)
10	ZGT	Zn > Zmax (valid for 2-D MIN-MAX instructions)
9	ZLT	Zn < Zmin (valid for 2-D MIN-MAX instructions)
8	YGT	Yn > Ymax (valid for 1-D or 2-D MIN-MAX instructions)
7	YLT	Yn < Ymin (valid for 1-D or 2-D MIN-MAX instructions)
6	XGT	Xn > Xmax (valid for 1-D or 2-D MIN-MAX instructions)
5	XLT	Xn < Xmin (valid for 1-D or 2-D MIN-MAX instructions)
4	HINT	Hardware interrupt flag
3-0	n/a	Reserved, set to zero

Output exceptions may be due to either an illegal data format or to a procedural error, such as:

Results too large or too small to be represented in the selected precision are signaled by V (overflow) and U (underflow).

An ALU output which was increased in magnitude by rounding causes X (inexact) to be set.

Wrapped outputs from the multiplier may be inexact and increased in magnitude by rounding, which sets the X (inexact) and RND (rounded) status flags high.

DENORM is set when the multiplier output is wrapped or the ALU output is denormalized.

DENORM is also set high when an illegal integer operation is performed.

DIV0 is set whenever the divisor is zero. The result of the operation is infinity.

Invalid operations cause the I flag to be set. The I bit will also go high if a NaN is input to the FPU.

The ED flag is a logical OR of the above exceptions. If any of the exception flags is high, ED will also be high. Exceptions can be masked out of ED by setting the appropriate bits in the configuration register. If the ED interrupt (INTENED) is enabled, an interrupt is generated when ED goes high.

Status flags are provided for both floating-point and integer results. Integer status is provided using Z for zero detect, N for sign, and V for overflow/carryout. Bits 14 and 13 are logical combinations of these three flags.

If the floating-point input to the multiplier is a denorm, DENIN will be set. If the input to the FPU is a NaN, I (invalid operation) will be set. STX1-0 indicate which operand is the source of the exception when either a denormal is input to the multiplier (DENIN=1) or a NaN is input (I=1).

NaN inputs are all treated as IEEE signaling NaNs causing the I flag to be set. When the FPU outputs a NaN, it is always in the form of a signaling NaN with the I and appropriate STX flags set high. The exponent and fraction fields of the NaN are set to all 1s, regardless of the input fraction.

Invalid operations that set the I flag include:

Operations with NaN inputs

Zero divided by zero

Positive infinity minus positive infinity or negative infinity minus negative infinity

Positive infinity plus negative infinity

Square root of a negative number

Zero multiplied by infinity

The result of these operations is a NaN.

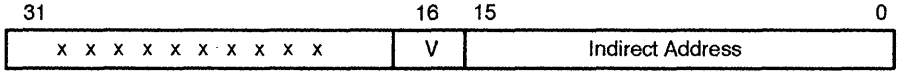
Bits 15, 12, and 11 in the status register are used to enable interrupts. Interrupts are enabled by setting INENHW (hardware interrupt), INTENSW (software interrupt), or INTENED (ED interrupt) high. A software interrupt is generated by writing to the status register with bit 16 (INTFLG) set to one.

4.5.5 Indirect Address Register

The indirect address register (MCADDR) can be set to point to a memory location for indirect move or jump operations on the MSD port. MCADDR is cleared upon reset. Although MCADDR cannot be used directly as an operand for FPU instructions, you can do an arithmetic operation on the value in MCADDR by first moving the contents to a register file location. Then perform the operation, choosing MCADDR as the destination.

The function of bit 16 varies, depending on whether the instruction is a move or jump. During a move instruction, bit 16 selects data space when set high or code space when low. During a jump instruction, bit 16 selects an internal instruction when set high or an external instruction when low (see Figure 4–9).

Figure 4–9. Indirect Address Register Format



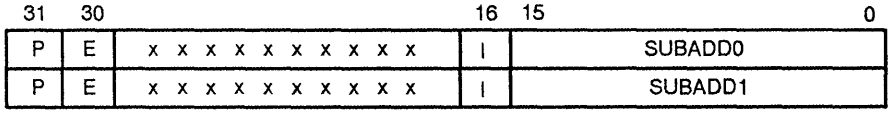
4.5.6 Stack

The stack contains two subroutine return address registers (SUBADD0 and SUBADD1) which serves as a two-deep last-in, first-out (LIFO) stack. A subroutine jump causes the program counter to be pushed onto the stack, and a return from subroutine pops the last address pushed onto the stack. More than two pushes will overwrite the contents of SUBADD1.

Bit 31 (Pointer) is set high in the stack location that was written last and reset to zero in the other stack location. Setting bit 30 (Enable) high enables a write into bit 31 (set or reset the pointer) in either stack location. If bit 31 is zero in both SUBADD0 and SUBADD1 (as when the stack has been saved externally and later restored), SUBADD0 can be designated as top of stack by setting bit 31. The stack pointers are cleared upon reset.

Bit 16 (I) is set high when the address in a stack location points to an internal routine or set low when the address is an external instruction.

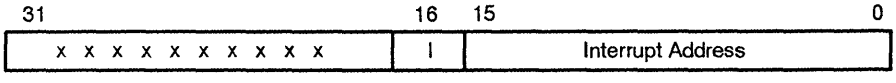
Figure 4–10. Stack Register Format



4.5.7 Interrupt Vector Register

The interrupt vector register (VECTOR) serves as a pointer to an external program to be executed upon receipt of an interrupt. Bit 16 (I) is always set low to point to a routine in external code space. The interrupt vector is cleared on reset. This register is only 17 bits wide (as shown in Figure 4–11) and should not be used for temporary storage.

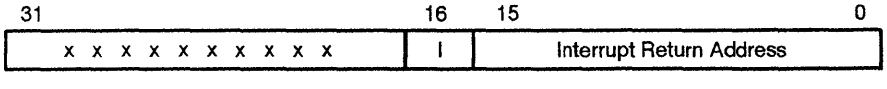
Figure 4–11. Interrupt Vector Register Format



4.5.8 Interrupt Return Register

The interrupt return register (IRAREG) retains a copy of the program counter at the time of an external interrupt. This address is used as the next execution address upon returning from the interrupt. Bit 16 (I) is set high when the address points to an internal instruction or set low when the address is in an external instruction. This register is not affected by the reset signal and, as illustrated in Figure 4–12, is only 17 bits wide and should not be used for temporary storage.

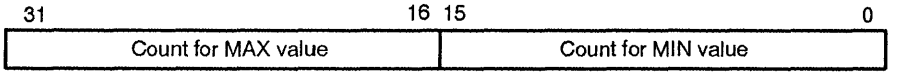
Figure 4–12. Interrupt Return Register Format



4.5.9 COUNTX and COUNTY Registers

The counter registers (COUNTX, COUNTY) are used to store the current counts of the minimum and maximum values when executing MIN-MAX instructions. They may also serve as temporary storage for the user. COUNTX and COUNTY are cleared on reset.

Figure 4–13. COUNT Registers Format

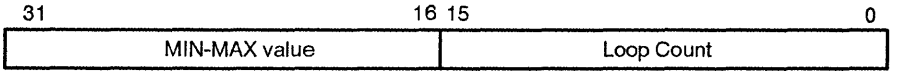


The COUNTX register is updated on both the 1-D and 2-D MIN-MAX instruction such that the count of the current minimum value is in the lower 16 bits of the register and the count of the current maximum value is in the upper 16 bits. The COUNTY register is used only in the 2-D MIN-MAX instruction to keep track of the counts of the minimum and maximum for the second value of a pair.

4.5.10 MIN-MAX/LOOPCT Register

The MIN-MAX/LOOPCT register stores the current values of two separate counters. The LSH contains the current loop counter and the MSH is used to hold the current minimum or maximum value of a MIN-MAX operation. This register may also serve as temporary storage for the user. The MIN-MAX/LOOPCT register is cleared upon reset.

Figure 4–14. MIN-MAX/LOOPCT Register Format



4.6 FPU Core

The FPU core consists of a multiplier and ALU, each with an intermediate pipeline register and an output register. The multiplier and ALU may operate independently or in parallel.

The major components include:

Operand multiplexers

Pipeline registers

ALU

Multiplier

Output control

Figure 4–15 shows a functional block diagram of the FPU core.

4.6.1 Operand Selection

Four multiplexers select the multiplier and ALU operands. Possible operand sources are:

RA and RB register files

Internal feedback registers C and CT

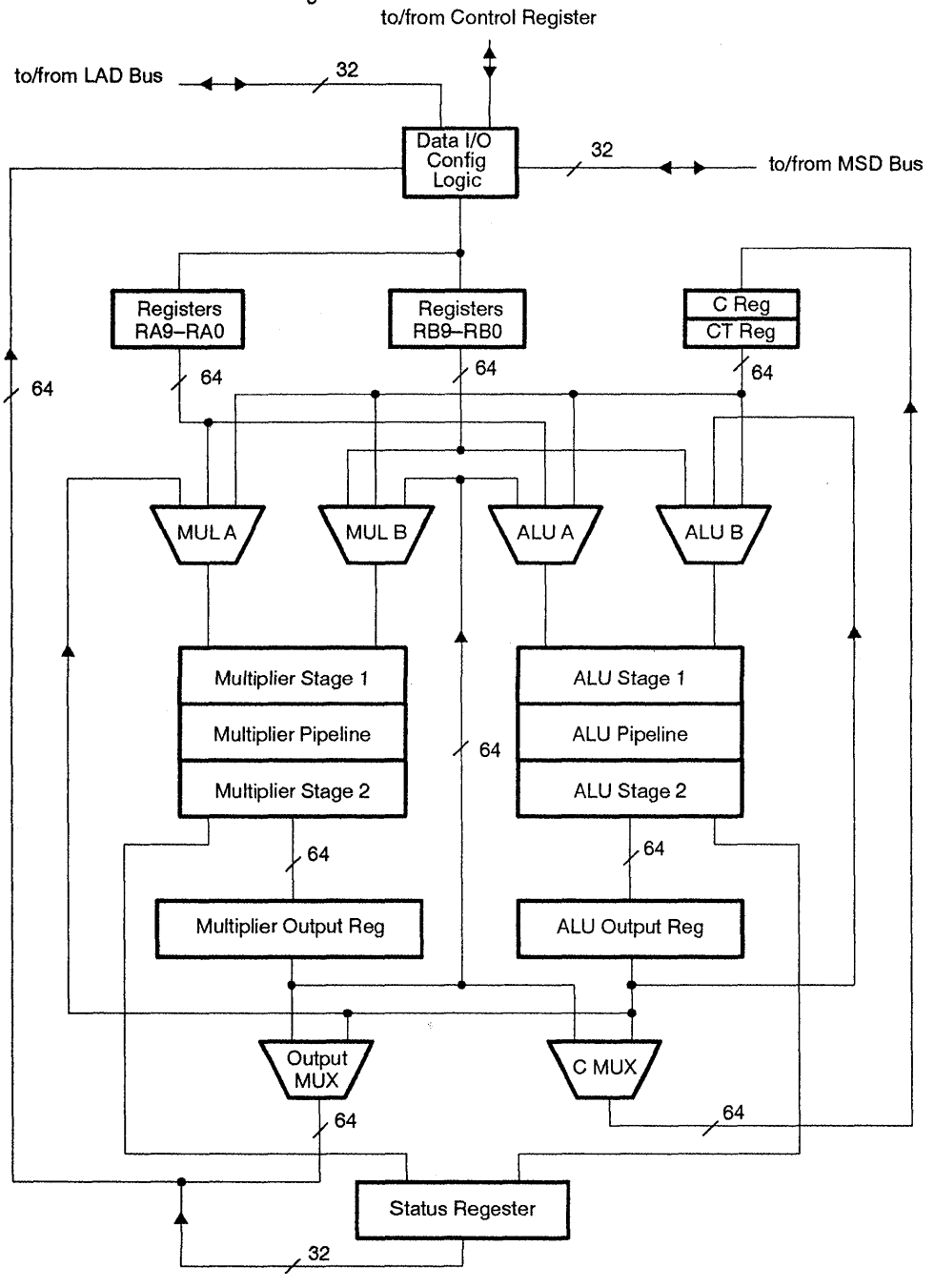
FPU core output registers

The FPU core output registers provide the previous multiplier or ALU result. Note that if the output registers are used as operands, they must be enabled. (See subsection 4.6.2 — Pipeline Registers — for additional details.)

For external instructions, immediate data from the LAD bus or the value 1 may also be chosen as operands. These are selected by setting the appropriate address bits (see section 4.5 — Registers) and selecting the RA or RB register file as operands.

The selection of operands also depends on the ALU or multiplier operation chosen. Single-operand instructions are generally performed only on registers in the RA file. Exceptions to this are the PASS instruction and certain complex internal instructions. Also in chained mode (the ALU and multiplier acting in parallel) the RB operand may optionally be forced to 0 in the ALU or 1 in the multiplier.

Figure 4–15. FPU Core Functional Block Diagram



4.6.2 Pipeline Registers

Two levels of internal data registers are available to segment the internal data paths of the TMS34082 FPU core. The registers are enabled by setting the PIPES2-1 bits in the configuration register.The most basic choice is whether to use the device in unpipelined mode (with no internal registers enabled) or whether to enable one or more pipeline registers. When no internal registers are enabled, the clock period is longest (the TMS34082 timing specifications are contained in Appendix A) .

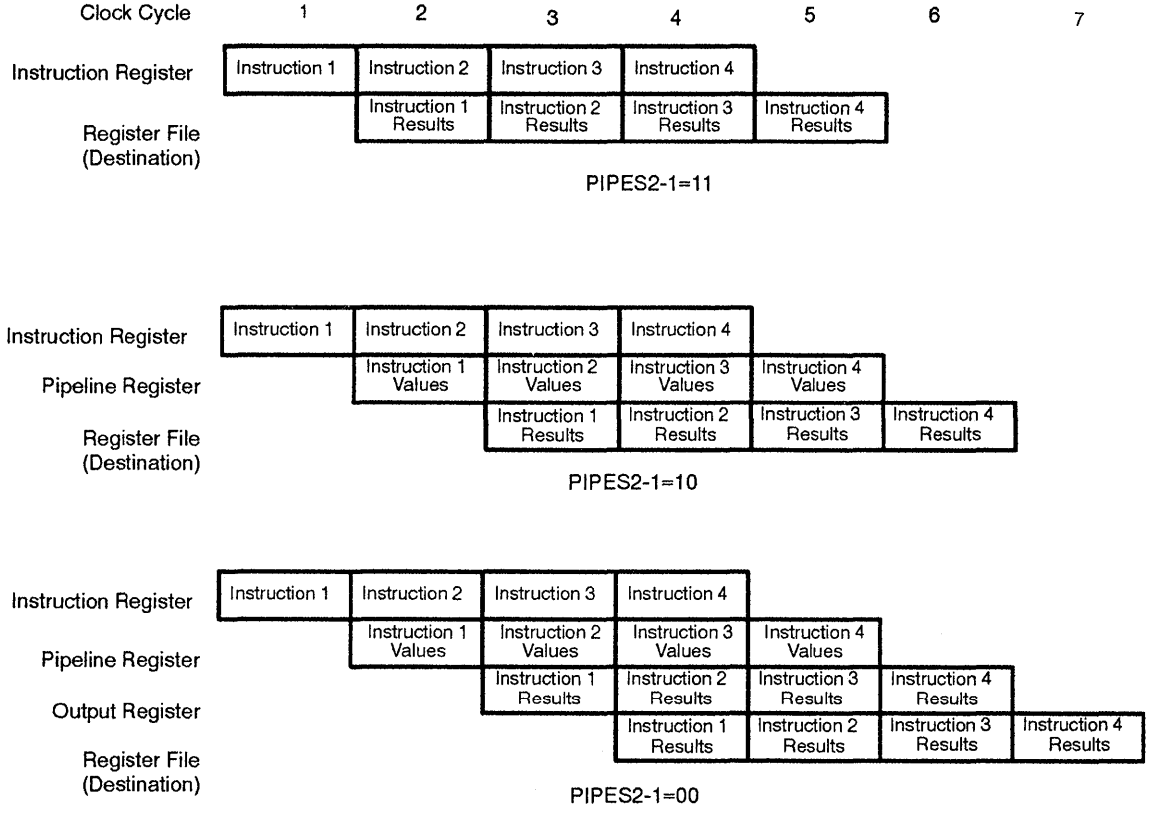
Enabling one or both sets of pipeline registers segments the data paths. When the intermediate pipeline is enabled, the register-to-register delay inside the device is minimized, allowing operation with the minimum cycle time. While one FPU instruction is executing, the next instruction may be input so that overlapping operations occur. This is commonly known as pipelined execution.

The TMS34082 may also operate with both sets of pipeline registers disabled. With this setting, two 32-bit operands are read from the register file, an operation is performed by the ALU or multiplier, and the result is stored in the register file, all in one clock cycle. A double-precision ALU operation takes one clock cycle, but double-precision multiplies require two clock cycles to complete.

When the ALU and multiplier operate in parallel (chained mode), two data operands come from the register files while multiplier and ALU feedback provide the other two operands. Therefore, *in chained mode the FPU core output registers must be enabled.* After the chained operation is completed and the results have been stored, the FPU core output registers may be disabled again. Wait until all operations have completed to change pipeline settings to avoid loosing any results.

The selection of pipeline registers determines the latency from input to output, the number of cycles required for an instruction to be processed and the results to appear. For each register level enabled in the data path, one clock cycle is added to the latency from input to when the result is valid in the register file. Figure 4–16 shows the latency of different pipeline settings. A result may be used as input on the same cycle that it is clocked into the register file.

Figure 4–16. Effects of Pipelining



Both sets of pipeline registers are controlled by the PIPES2 and PIPES1 bits in the configuration register. When the device is powered up or reset, the intermediate pipeline registers are enabled (PIPES1=0) and the output registers are transparent (PIPES2=1). For internal instructions, control logic sets the pipeline registers as needed and restores them to their previous configuration after the instruction is completed.

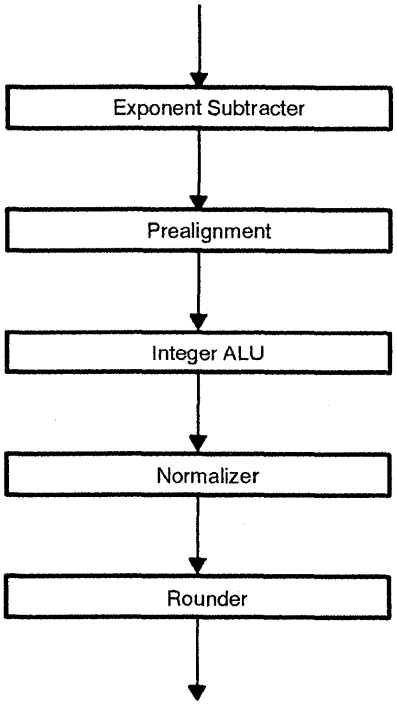
Pipeline settings should be changed only when all instructions executing in the FPU core are completed and results are stored in the register file. Otherwise, results will be lost. The nop (No Operation) instruction may be inserted to allow time for the last instruction to finish before changing the pipeline configuration.

When using chained mode, the nop instruction may be used to adjust output register timing. Each nop instruction keeps the results in the output registers for one additional clock cycle. nop may be used in this manner only when the output registers are enabled.

4.6.3 ALU

The pipelined ALU contains a circuit for floating-point addition and/or subtraction of aligned operands, a pipeline register, an exponent adjuster, and a normalizer/rounder as shown in Figure 4–17. Exception logic is provided to detect denormalized inputs; these can be flushed to zero if the FAST input is set high. If the FAST input is low, the ALU accepts a denormal as input. The denormal exception flag (DENORM) goes high when the ALU output is a denormal.

Figure 4–17. Functional Diagram for ALU



Integer processing in the ALU includes both arithmetic and logical operations in either 2s complement numbers or unsigned integers. The ALU performs addition, subtraction, comparison, logical shifts, logical AND, logical OR, and logical XOR. Format conversions and wrapping/unwrapping of denormals are also done by the ALU.

4.6.4 Multiplier

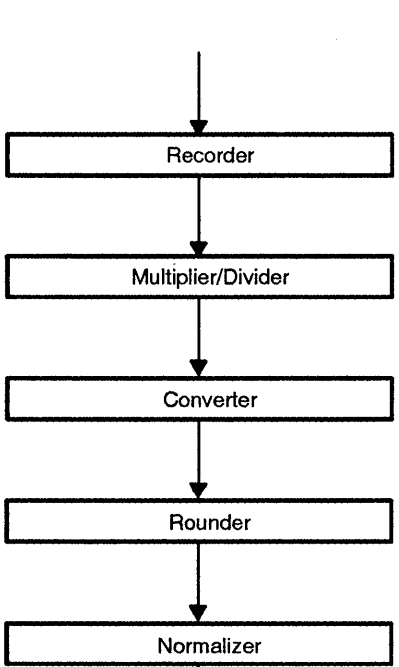
The pipelined multiplier (see Figure 4–6) performs a basic multiply function, division, and square root. The operands can be single- or double-precision floating-point numbers and can be converted to absolute values before multiplication takes place. Integer operands may also be used.

If the operands to the multiplier are double-precision or mixed precision (i.e., one single-precision and one double-precision), then one extra clock cycle is required to get the product through the multiplier pipeline. This means that for PIPES1=1, one clock cycle is required for the multiplier pipeline; for PIPES1=0, two clock cycles are required for the multiplier pipeline.

An exception circuit is provided to detect denormalized inputs; these are indicated by a high on the DENIN signal. Denormalized inputs must be wrapped by the ALU before multiplication, division, or square root. If results are wrapped (signaled by a high on the DENORM status pin), they must be unwrapped by the ALU first.

The multiplier and ALU can be operated simultaneously. Division and square root are each performed as an independent multiplier operation, even though both multiplier and ALU are active during these operations.

Figure 4–18. Functional Diagram for Multiplier



4.6.5 Output Control

An output MUX selects which result (ALU or multiplier) is written to the register. The instruction specifies where the result is stored. Results may be directed to the twenty registers in files RA and RB, the feedback registers (C and CT), or the other temporary storage registers.

Although it is possible to direct the result to the CONFIG, STATUS, MCADDR, VECTOR, IRAREG, SUBADD0, and SUBADD1 registers, it is not recommended. These registers have dedicated functions as discussed in section 4.5.

The COUNTX, COUNTY, and MIN-MAX/LOOPCT may be used as temporary storage registers. Because they are only 32-bits wide, double-precision results cannot be stored in these registers.

4.7 RESET and RDY

The RESET input is an active low signal that asynchronously clears the internal states and resets the configuration and status registers to the default values. Internal pipeline registers are cleared, but the register files, C, and CT are not affected.

During reset, control inputs are in an inactive state as shown in Table 4–10. The LAD and MSD buses are placed in a high-impedance state, and the MSA bus outputs an address of 0.

Table 4–10. Signal States During Reset

Signal Name	Logic Level
LAD31-0	high impedance
ALTCH†	high
CAS†	high
WE†	high
MSD31-0	high impedance
MSA15-0	low
DS/CS	high
MAE	high
MCE	high
MOE	high
MWR	high
COINT	high
CORDY	high
INTG	low

† Host-independent mode only.

Operation resumes on the rising edge of the clock after RESET is set high again. In host-independent mode, MCE becomes active and causes a read from code address 0. In coprocessor mode, the TMS34082 goes to an idle state, waiting for an instruction from the TMS34020.

The TMS34082 can be nondestructively stalled by setting the RDY input low. The next rising clock edge is inhibited. Normal operation resumes on the cycle after the RDY input is set high again.

While halted, the registers and internal states are unaltered. Output pins remain at their previous levels. The asynchronous inputs (LOE, MOE, and RESET) are still active. If an interrupt is received while the device is stalled, it will be queued and serviced after operation resumes.

4.8 Emulation Control

Two emulation mode control pins, EC1-0, support system testing. These may be used, for example, to place all outputs in a high-impedance state, isolating the TMS34082 from the rest of the system.

Test modes are given in Table 4–11 . For normal operation, EC1 and EC0 must both be high.

Table 4–11. Test Modes

EC1-0		Operation
0	0	All output and I/O pins are forced low
0	1	All output and I/O pins are forced high
1	0	All output pins are placed in high-impedance state
1	1	Normal operation

4.9 JTAG Test Port

The TMS34082 includes a 4-wire Test Access Port (TAP) interface that allows serial scan access to test circuitry within the device. This TAP is compatible with the IEEE 1149.1 (JTAG) specification. It was designed using the TI Scope™ (System Controllability, Observability, and Partitioning Environments) guidelines. For normal operation, the input pins should be connected as shown below.

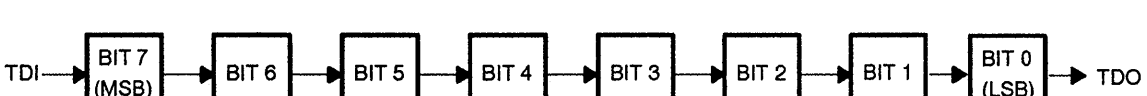
Table 4–12. Test Pins for Normal Operation

Signal Name	Logic Level
TCLK	Tied low or high
TDI	Tie high or leave floating
TMS	Tie high or leave floating

4.9.1 Test Instructions

The TAP includes an 8-bit instruction register used to tell the device what instruction is to be executed. The instruction register is loaded serially via the TDI input. The order of scan is shown in Figure 4–19.

Figure 4–19. Instruction Register Order of Scan



Four test instructions are supported; Table 4–13 lists their binary opcodes. Any instruction code not supported is interpreted as the Bypass instruction.

Bypass

A one-bit bypass register is selected in the scan path. Data input from TDI is shifted into the bypass register, then out through TDO.

Extest

This is the 1149.1 Extest instruction with the boundary scan register in the scan path. Data appearing at the device inputs and outputs is captured. Data previously loaded into the boundary scan register is applied to the device inputs and through the device outputs.

Intest

This is the 1149.1 Intest instruction with the boundary scan register in the scan path. Data appearing at the device inputs and outputs is captured. Data previously loaded into the boundary scan register is applied to the device inputs and through the device outputs.

Sample

This instruction conforms to the 1149.1 Sample/Preload instruction. Data appearing at the device inputs and outputs is sampled without affecting normal operation. The boundary scan register is selected in the scan path.

Table 4–13. Instruction Register Opcodes

Binary Opcode	Opcode	Description
00000000	BYPASS	Bypass scan
00000011	INTEST	Boundary scan in test mode
10000010	SAMPLE	Sample boundary scan in normal mode
11111111	EXTEST	Boundary scan in test mode

4.9.2 Boundary Scan Register

The boundary scan register contains 181 bits, one for each functional input and output on the TMS34082. Each I/O pin has both an input and an output register bit associated with it. In addition, some three-state outputs have an additional bit in the scan register. These represent internal three-state enable registers, not actual pins on the package. Table 4–14 lists these scan bits and the outputs they affect.

Table 4–14. Boundary Scan Register Enable Bits

Scan Name	Affected Outputs
CO-EN	$\overline{\text{COINT}}$, CORDY
ALTCH-EN	$\overline{\text{ALTCH}}$ (output)
CAS-EN	$\overline{\text{CAS}}$ (output), $\overline{\text{WE}}$ (output)
LAD-EN	LAD31-0 (outputs only)
MSD-EN	MSD31-0 (outputs only)
MSA-EN	MSA15-0
MWR-EN	$\overline{\text{MWR}}$, $\overline{\text{MOE}}$, DS/$\overline{\text{CS}}$, $\overline{\text{MCE}}$, INTG

The boundary scan register is used to store test data that is to be applied internally and/or externally to the TMS34082 and to capture and store data that is applied to the functional inputs and outputs. The boundary scan register order of scan is shown in Figure 4–20.

Figure 4–20. Boundary Scan Register Order of Scan

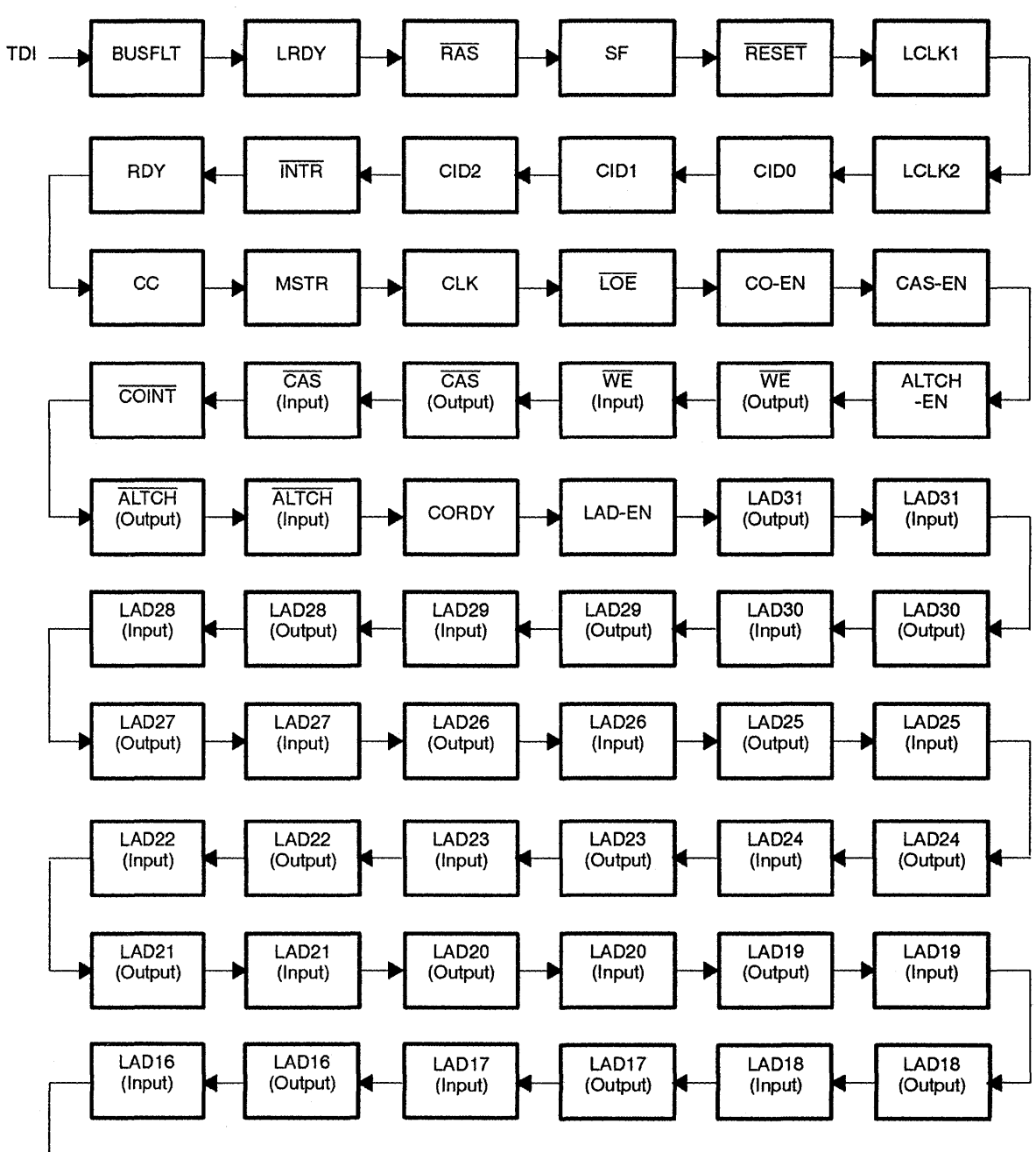


Figure 4–20. Boundary Scan Register Order of Scan (Continued)

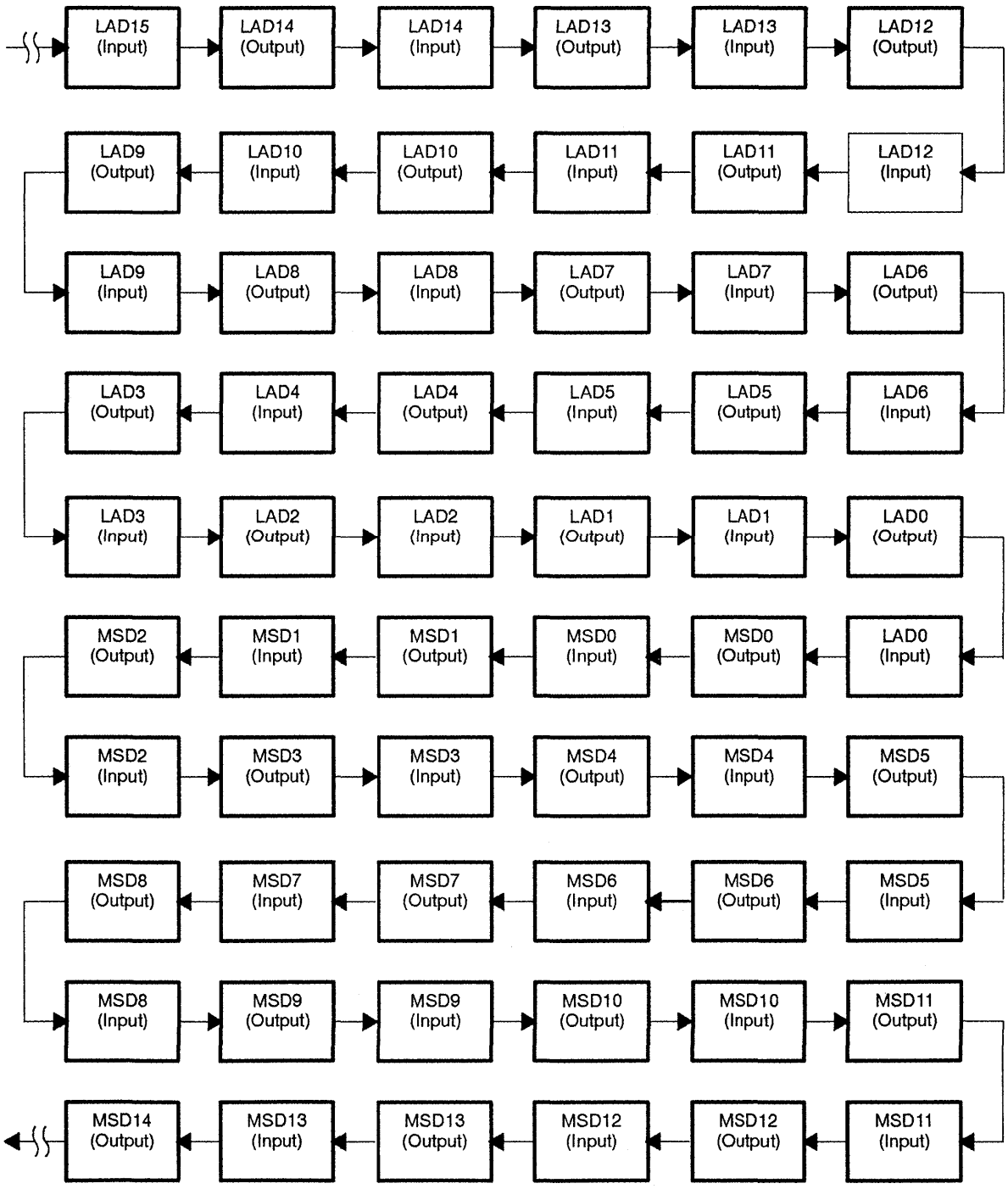
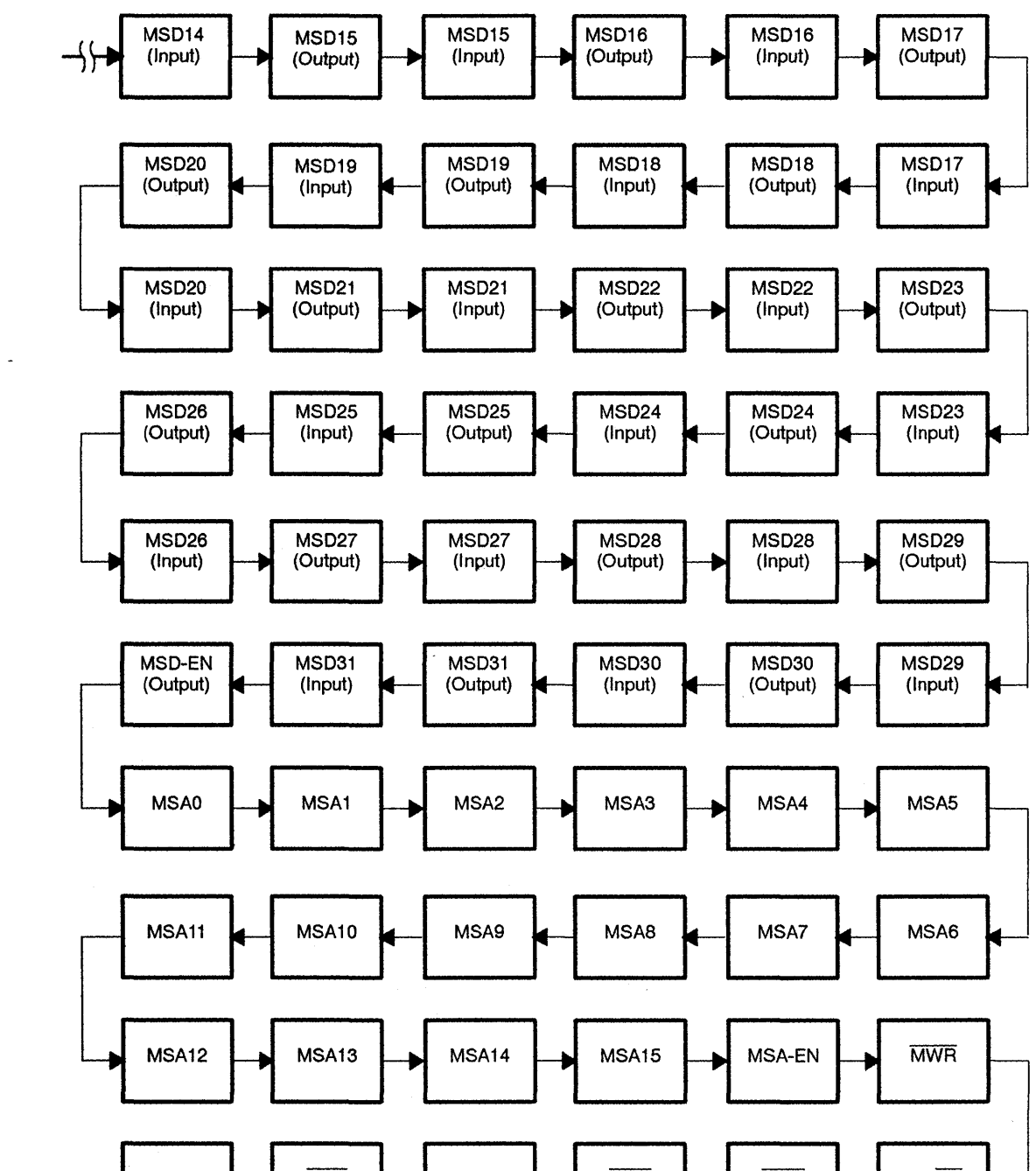


Figure 4–20. Boundary Scan Register Order of Scan (Continued)



Chapter 5

Coprocessor Mode

The TMS34082 provides closely coupled floating-point support for the TMS34020. The devices were designed with a direct-wire interface that requires no additional external glue logic. Combinations of TMS34020 and TMS34082 devices provide the performance to cover a broad range of graphic applications. This family of solutions makes upgrading your design easy.

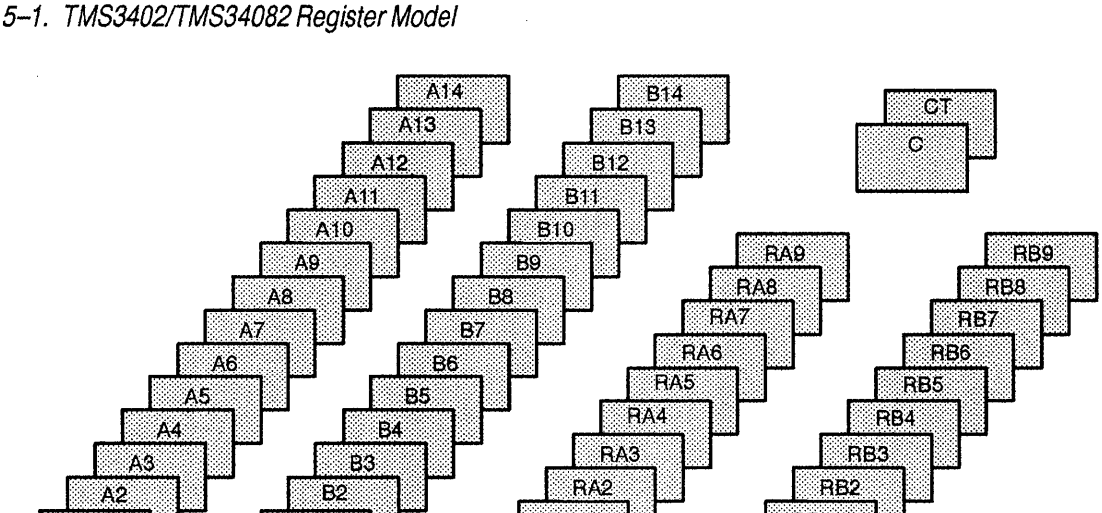
The TMS34082 is more than a simple coprocessor. It contains complex instructions specifically tailored for graphics operations. The ability of the TMS34020 and TMS34082 to operate in parallel, support for multiple TMS34082 devices, and the option of adding external user-generated subroutines also increase system performance.

5.1 TMS34020/TMS34082 Interface Overview

Operation in coprocessor mode assumes the MSTR input signal is set low. In this mode, the TMS34082 acts as a tightly coupled coprocessor to the TMS34020. In terms of the instruction set and register resources, the TMS34082 appears as an extension to the TMS34020 register and instruction set.

Figure 5–1 shows the register allocation for the TMS34020/TMS34082 combination.

Figure 5–1. TMS3402/TMS34082 Register Model



TMS34020 Registers **TMS34082 Registers**

The TMS34082 executes two different instruction sets:

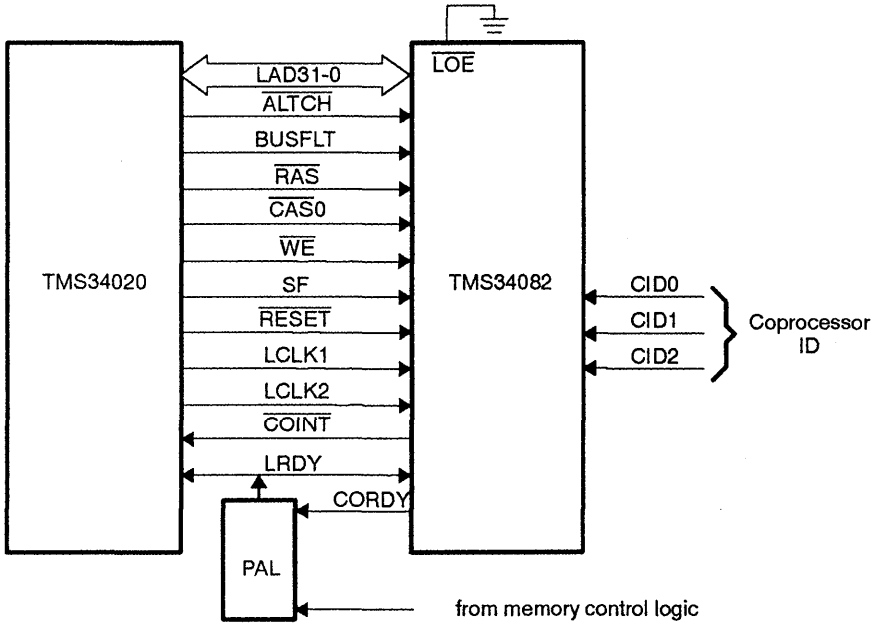
Internal instructions from the TMS34020 are input on the LAD port. They include complex graphics, matrix, and vector routines. These are described in Chapter 7.

External instructions are input on the MSD port. This is a RISC-like instruction set. They are used to write user-defined subroutines. External instruction are covered in Chapter 8.

The interface between the TMS34020 and the TMS34082 consists of direct connections between pins. No glue logic is required other than gating the ready signals into the TMS34020. Figure 5–2 shows the interconnection.

The LAD interface includes the following signals: LAD31-0, $\overline{\text{LOE}}$, $\overline{\text{ALTCH}}$, LRDY, BUSFLT, $\overline{\text{RAS}}$, $\overline{\text{CAS}}$, $\overline{\text{WE}}$, SF, $\overline{\text{COINT}}$, CORDY. These signals communicate between TMS34020 and TMS34082 in coprocessor mode. $\overline{\text{COINT}}$ and CORDY are the only signals that go from the TMS34082 to the TMS34020; all other signals are inputs to the TMS34082. $\overline{\text{COINT}}$ should be connected to one of the TMS34020 local interrupt requests, $\overline{\text{LINT1}}$ or $\overline{\text{LINT2}}$.

Figure 5–2. TMS34020/TMS34082 Interconnection



CORDY from the TMS34082 is logically ORed with other ready signals from the system to form the TMS34020 LRDY input ready signal. Note that LRDY connects to both the TMS34020 and the TMS34082 inputs.

When operating in the coprocessor mode, connect the remaining TMS34082 pins as shown in Table 5–1.

Table 5–1. Recommended TMS34082 Pin Connections

Signal Name	Description	Logic Level
MSTR	Coprocessor/Host-independent mode select	tie low
CLK	Host-independent mode clock	tie low
CID2-0	Coprocessor ID (assembler default is 000_2)	tie low
EC1-0	Emulator mode control	tie high
TCK	Test clock input	tie low
$\overline{\text{LOE}}$	LAD output enable	tie low
$\overline{\text{INTR}}$	Interrupt request input	tie high

5.2 Clocks

Local clock input signals LCLK1 and LCLK2 are generated by the TMS34020. Internally, the TMS34082 generates a rising clock edge from each LCLK1 edge (rising or falling). In coprocessor mode, the TMS34082 actually operates at twice the LCLK1 input clock frequency.

LCLK1 controls most of the TMS34082 internal logic while LCLK2 is used for several simple functions such as synchronizing interrupt requests.

CLK is the system clock input in host-independent mode. It should be tied low for coprocessor mode.

5.3 TMS34082 Initialization

The TMS34082 uses the same $\overline{\text{RESET}}$ input signal that the TMS34020 uses. Upon reset, the TMS34082 clears all pipeline registers and internal states. The configuration register and status register return to their default values. When $\overline{\text{RESET}}$ returns high in coprocessor mode, the TMS34082 is in an idle state waiting for the next instruction from the TMS34020. The $\overline{\text{RESET}}$ signal is an asynchronous signal and does not require specific setup or hold times to a clock. However, the minimum pulse duration requirement must be met.

5.4 Configuration Register Settings for Coprocessor Mode

The configuration register (CONFIG) defines several selectable features of the TMS34082. The following subsections recommend settings for this register in coprocessor mode. Part of your system initialization program should set the configuration register to the appropriate value.

5.4.1 Exception Masks

Since inexact operations are common in floating-point operations, you should usually disable this exception for both the multiplier and ALU by setting the MINEX and AINEX bits to 0.

5.4.2 Fast vs IEEE Mode

For most graphics applications where integer and single-precision floating-point number formats are used, operating the TMS34082 in Fast mode is sufficient. This also holds true for most double-precision floating-point applications. Because the internal instruction set does not include instructions to wrap and unwrap denormalized numbers, you should use Fast mode if you do not have memory on the MSD port for external instructions.

However, when working with very large or very small double-precision values, IEEE mode can be used to operate on denormalized numbers. Possible uses of IEEE mode include image processing and digital signal processing applications where accuracy is critical. External instructions must be used to wrap and unwrap denormalized numbers. See Chapter 8 for details on these instruction.

5.4.3 Pipeline Mode Settings

For coprocessor mode, the TMS34082 pipeline mode settings (PIPES2-1 in the CONFIG register) affect the performance of very few internal instructions. Most simple instructions, such as adds or multiplies, finish executing before the TMS34020 can issue the next instruction. Using the default setting allows you to run the TMS34082 at the maximum clock rate. This setting (PIPES2 = 1, PIPES1 = 0) is recommended unless you are using chained mode external instructions. While using chained mode instructions, PIPES2 should be set low to enable the FPU core output registers.

The complex instructions contained in internal ROM change the pipeline setting as needed and restore the previous pipeline setting after the instruction is completed.

5.5 TMS34020/TMS34082 LAD Bus Operation

The TMS34020 local memory interface is made up of a multiplexed address/data bus and associated control signals. During a memory cycle, the address and status are output on the LAD bus, and then the LAD bus is used for the data transfer. The local memory and DRAM/VRAM interfaces are used for transferring data or instructions between the TMS34020, memory, or the TMS34082 in addition to generating refreshing cycles for DRAM/VRAM.

In coprocessor mode, the TMS34082 LAD bus connects directly to the TMS34020 LAD bus. Coprocessor commands from the TMS34020 are input on this bus. In addition, data transfers between the TMS34020 or its local memory and the TMS34082 occur through the LAD bus. Transfers between the LAD and MSD buses can also be programmed.

A single coprocessor instruction may be used to pass a command to the TMS34082 and transfer data to/from the TMS34020 or memory. There are five general types of coprocessor instructions.

Command-only instructions transfer no data to the TMS34082.

TMS34020 to TMS34082 transfer instructions pass a command and data to the coprocessor. Three types of transfers are available:

move one 32-bit parameter

move two 32-bit parameters

move one 64-bit parameter

TMS34082 to TMS34020 transfer instructions pass a command to the coprocessor and the TMS34082 outputs data to the LAD bus. Two types of instructions are available:

move one 32-bit parameter

move one 64-bit parameter

Memory to TMS34082 transfer instructions pass a command from the TMS34020 and data from memory to the coprocessor. Up to 32 32-bit words may be transferred. Three types of memory moves are available:

move the number of words specified in the coprocessor instruction using postincrement

move the number of words specified in the coprocessor instruction using predecrement

move the number of words specified in a register using postincrement.

TMS34082 to memory transfer instructions pass a command to the coprocessor and the TMS34082 outputs data to the LAD bus. Up to 32 32-bit words may be transferred. Two types of memory moves are available:

move the number of words specified in the coprocessor instruction using postincrement

move the number of words specified in the coprocessor instruction using predecrement

5.5.1 LAD Bus Protocol

Both data and instructions are transferred over the bidirectional LAD bus in coprocessor mode. A unique combination of signal inputs distinguishes an instruction from data. SF, $\overline{\text{ALTCH}}$, $\overline{\text{CAS}}$, $\overline{\text{RAS}}$, and $\overline{\text{WE}}$ are used to distinguish coprocessor functions from other operations on the LAD bus.

The TMS34020 first fetches a coprocessor instruction from either internal cache or from local memory on the LAD bus. A coprocessor command is then issued to the TMS34082 from the TMS34020 by way of the following protocol:

A valid coprocessor ID (CID2-0) on LAD31-29

LAD3-0 = 0000_2

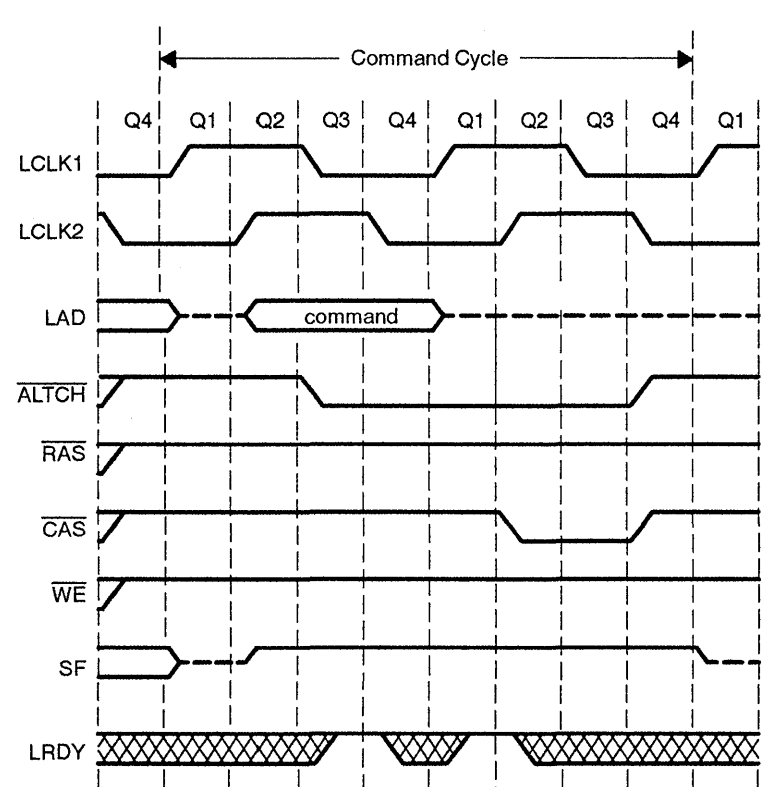
$\overline{\text{RAS}}$ high

SF high during the falling edge of $\overline{\text{ALTCH}}$

Note: When using one TMS34082 in a system, the assembler/compiler default for CID2-0 = 000_2.

The command is then decoded and executed by the appropriate TMS34082. If a command-only instruction is issued, the TMS34082 begins execution at the rising edge of LCLK1 after $\overline{\text{ALTCH}}$ falls. A timing diagram for command-only instructions is shown in Figure 5–3.

Figure 5–3. Transferring a Command from the TMS34020 to the TMS34082



If operands are required from DRAM/VRAM, the TMS34020 sets up the appropriate DRAM/VRAM address and timing. The data is then transferred directly between the TMS34082 and DRAM/VRAM.

All transfers to/from the TMS34082 are 32 bits wide. Therefore, the TMS34082 uses neither the TMS34020 $\overline{\text{SIZE16}}$ signal nor all four individual byte enables ($\overline{\text{CAS}}$3-0). *Also, the* **.even 32** *TMS34020 assembler directive should be placed before all blocks of DRAM/VRAM memory that are used to store data or external code to be sent to the TMS34082.* If the 32-bit words are not aligned on long word boundaries, the data is not sent to the TMS34082 correctly.

Instructions that pass data and commands to the TMS34082 begin execution on the rising edge of LCLK1 after $\overline{\text{CAS}}$ rises after the last data transfer. Timing diagrams for instructions that transfer data and commands are given in Figures 5–4 through 5–7.

Figure 5–4. Transferring TMS34020 Registers to the TMS34082

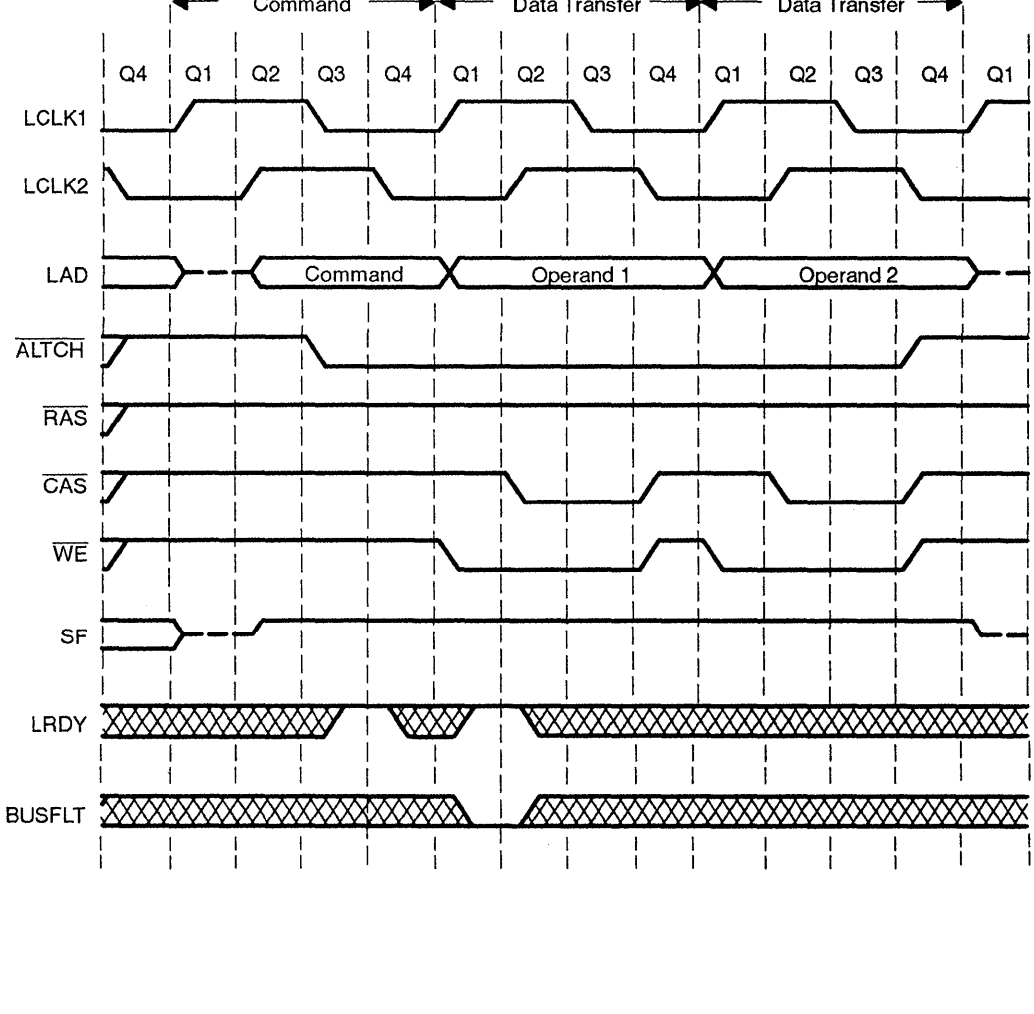


Figure 5–5. Transferring from the TMS34082 to a TMS34020 Register

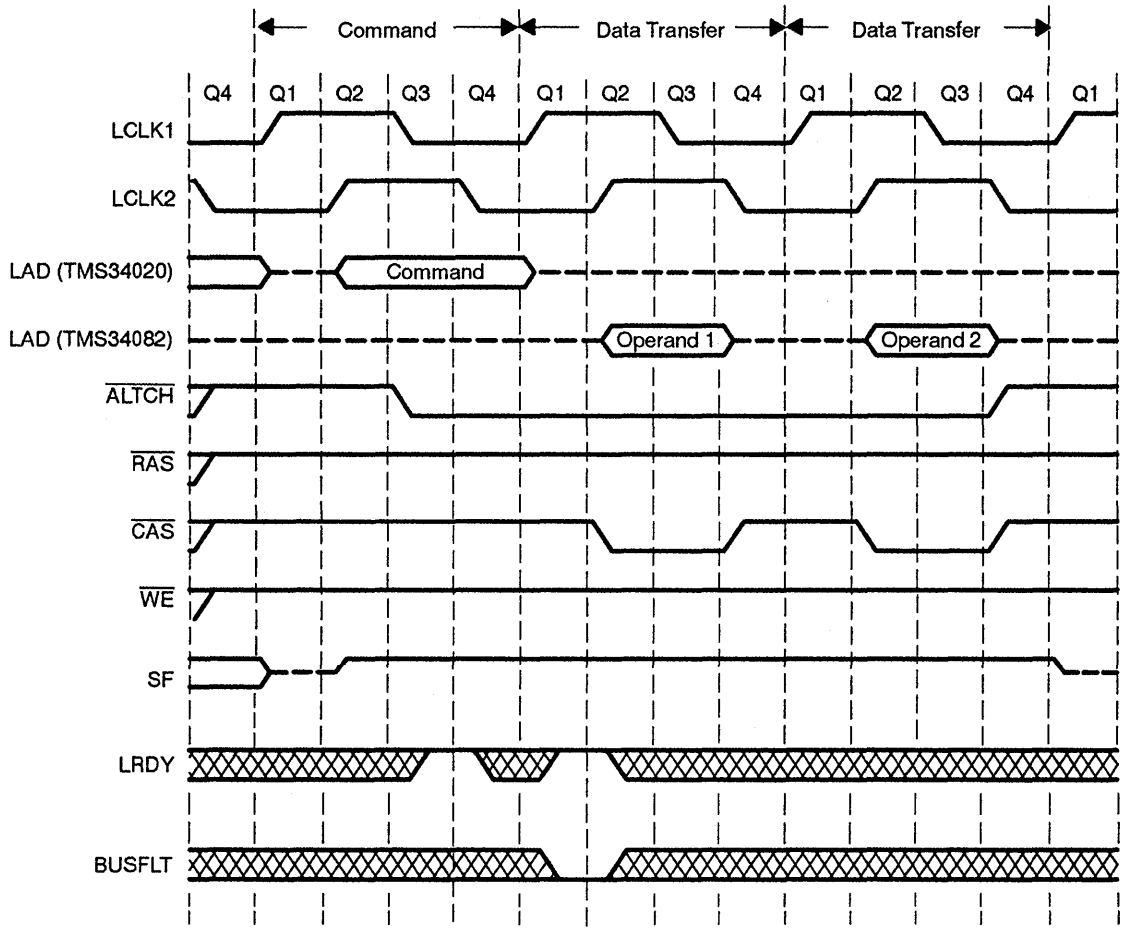
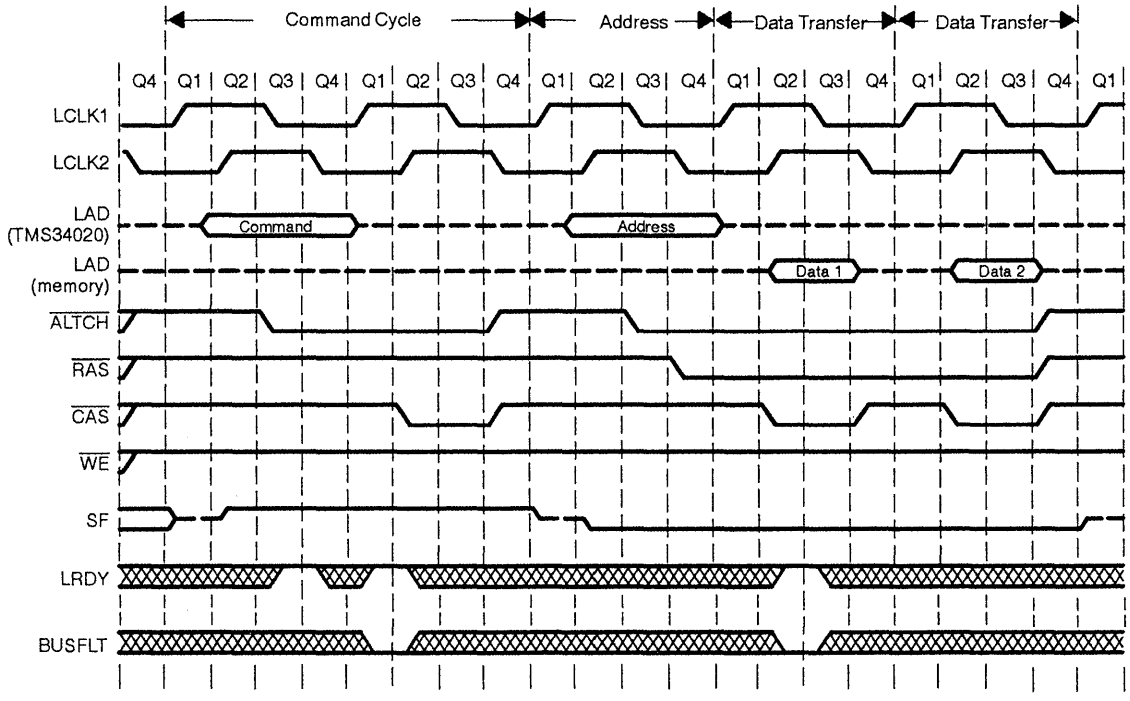
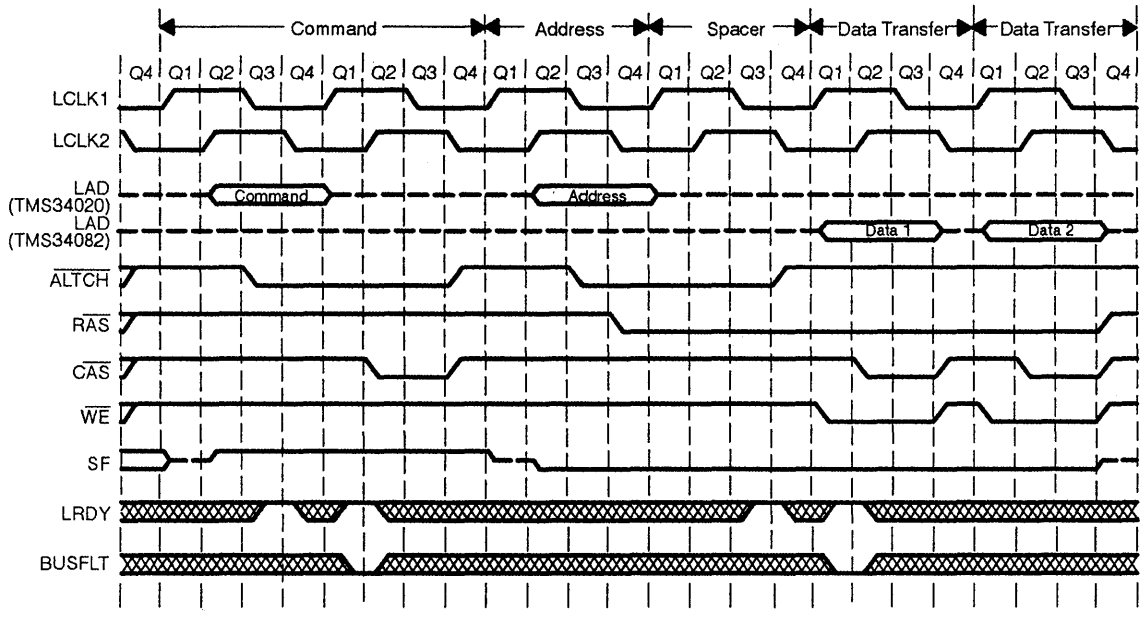


Figure 5–6. Transferring Memory to the TMS34082



When the TMS34082 is transferring data to memory, the TMS34020 outputs the memory address on the LAD bus. An extra clock cycle, called a spacer, is then inserted before the TMS34082 outputs data. The spacer is added to allow time for the TMS34020 to stop driving the LAD bus and the TMS34082 to set up valid data on LAD.

Figure 5–7. Transferring from the TMS34082 to Memory



5.5.2 Enabling the LAD Bus Drivers

The LAD bus drivers are enabled only when $\overline{LOE}$ is low, the correct TMS34082 coprocessor ID has been selected, and during the proper time slot within the execution cycle. Just bringing $\overline{LOE}$ low does not cause the LAD bus drivers to turn on. For most applications using a single TMS3020, TMS34082, and DRAM/VRAM, $\overline{LOE}$ may be tied low.

In a system with multiple TMS34082 coprocessors, only one coprocessor can drive the LAD bus at a time. The TMS34082 contains internal logic that only allows it to drive the LAD bus when its coprocessor ID is contained in the move instruction. A TMS34082 write instruction with the broadcast ID is ignored.

5.5.3 Bus Faults

The TMS34082 BUSFLT input signal also ties directly to the TMS34020 BUSFLT pin. The TMS34082 supports bus retries and bus fault conditions in conjunction with the TMS34020. The bus cycle conditions are defined in Table 5–2.

Table 5–2. Bus Cycle Completion Conditions

Completion Condition	BUSFLT	LRDY
Wait	0	0
Successful transfer	0	1
Retry	1	0
Bus fault	1	1

In the event of a systems fault involving the TMS34082, the abort command allows the TMS34020 to regain control. The abort terminates all coprocessor activity, restoring the TMS34082 to a known state so that it is available for further commands from the TMS34020. Chapter 7 covers the abort command in greater detail.

5.6 Polling the Coprocessor

When the TMS34020 issues an instruction to the TMS34082, CORDY (coprocessor ready) is high. It remains high even while the TMS34082 is busy executing the instruction. However, if another instruction is sent by the TMS34020 before the previous instruction has completed, CORDY will go low immediately, indicating that the TMS34020 must wait. When the TMS34082 is ready to accept the new instruction, CORDY returns high to signal the TMS34020 that the coprocessor is ready to accept a command. Because CORDY is usually ORed with other terms to form LRDY, CORDY going low also sends LRDY low, halting the TMS34020.

The instruction will still be valid on the LAD bus when CORDY (and LRDY) toggle, and the TMS34082 will latch the instruction. However, for longer TMS34082 operations, such as lengthy subroutines stored in SRAM, the TMS34020 may have to wait for a long period of time before the TMS34082 is ready. This ties up the TMS34020 and keeps it from executing other code. Instead, the TMS34020 can check the coprocessor's operating condition before issuing an instruction by way of the *check status* command. The TMS34020 assembler pseudo-op for this command is CHECK.

In response to the check status command, the TMS34082 outputs a status code to signal if it is busy or not. The TMS34082 returns a value of all 1s if busy or all 0s if idle, as shown in Table 5–3. This instruction is described further in Chapter 7.

Table 5–3. Bit Definitions for TMS34020 Status Check Command

Description	LAD Output
Coprocessor not busy	0000 0000h
Coprocessor busy	FFFF FFFFh

The TMS34020 does not have to enter an extended wait state to obtain access to the selected coprocessor, but may continue with another task not requiring the TMS34082. This allows the two devices to execute instructions in parallel. See Example 5–1 for an example of code using the check status command.

Example 5–1. Using the Status Check Command

```
   CHECK A1            ; put output status in TMS34020 register A1
   CMPI 0,A1           ; compare with all zeros
   JRNE busy           ; if busy, then execute more TMS34020 code

not_busy:              ; start next TMS34082 routine
    .
    .
    .
busy:                  ; execute more TMS34020 code while coprocessor is busy
```

5.7 Interrupt Handling

The TMS34082 has two interrupt input sources in coprocessor mode:

An exception detect (ED) interrupt used to signal the TMS34082 that a status exception occurred

A software interrupt generated by an external instruction input on the MSD bus

Each exception has its own interrupt enable flag in the status register. If external SRAM memory is not used, the software interrupt should be disabled. On reset, the exception detect (ED) interrupt is enabled and the software interrupt is disabled.

Because hardware interrupts are *not* allowed in coprocessor mode, the hardware interrupt should be disabled. This is the default setting of the hardware interrupt enable flag in the status register. Also, $\overline{\text{INTR}}$ should be tied high.

5.7.1 Exception Detect Interrupts

If the exception detect interrupt is enabled, $\overline{\text{COINT}}$ goes low when the ED flag in the status register is 1. The ED flag goes high when a status exception occurs (see subsection 4.5.3.1) $\overline{\text{COINT}}$ signals the exception to the TMS34020. This exception does *not* cause the TMS34082 to branch to the interrupt vector register address. The TMS34082 aborts the current instruction and goes to an idle state.

The $\overline{\text{COINT}}$ signal may be connected to either the TMS34020 LINT1 or LINT2 input. You can also combine $\overline{\text{COINT}}$ with other interrupt requests in the system to form LINT1 or LINT2. If its interrupts are enabled, the TMS34020 will branch to an interrupt vector to service the TMS34082 request.

$\overline{\text{COINT}}$ and ED are reset by reading the STATUS register. You should do this as part of your interrupt service routine.

In the interrupt service routine, saving the state of the TMS34082 may be desired. This is best accomplished by executing a block move of the TMS34082 registers to DRAM/VRAM memory. The TMS34020 assembly language instructions listed in Example 5–2 can be used for the desired precision. These routines do not save or restore the C and CT register. Restoring the TMS34082 machine state consists of moving the register values from memory back to the TMS34082. Restoring the status register sets the ED flag high. However, writing a 1 to ED will *not* cause an interrupt.

Example 5–2. Saving and Restoring the TMS34082 Machine State

```
MOVE RA0, *A1+, 30 ; integer    move,    use    TMS34020    register    A1
                   ; as the memory pointer

MOVF RA0, *A1+, 30 ; single-precision      move,      use      TMS34020
                   ; register A1 as memory pointer

MOVD RA0, *A1+, 15 ; double-precision      move,      use      TMS34020
                   ; register A1 as memory pointer,
MOVD RB1, *A1+, 15 ; remainder of double-precision move

                   ; restoring TMS34082 machine state

MOVE *A1+, RA0, 30 ; integer    move,    use    TMS34020    register    A1
                   ; as the memory pointer

MOVF *A1+, RA0, 30 ; single-precision      move,      use      TMS34020
                   ; register A1 as memory pointer

MOVD *A1+, RA0, 15 ; double-precision      move,      use      TMS34020
                   ; register A1 as memory pointer,

MOVD *A1+, RB1, 15 ; remainder of double-precision move
```

5.7.2 Software Interrupts

If software interrupts are enabled, an interrupt may be generated by an external instruction fetched from the MSD port. The interrupt sets the interrupt grant output (INTG) low, saves the current program counter in the interrupt return register (IRAREG) and branches to the address in the interrupt vector register. Interrupts are also disabled.

Your service routine should restore software interrupts at the end. The final instruction should be a return from interrupt that will branch to the value in the interrupt return register.

5.7.3 Interrupting the TMS34020

For some applications using long external subroutines, it is desirable to interrupt the TMS34082 to signal that the subroutine is finished. This relieves the TMS34020 from having to check the TMS34082 to see if it is ready for the next instruction.

This may be accomplished by intentionally executing an instruction (in external code) that sets the ED flag high. This causes $\overline{\text{COINT}}$ to go low, signaling an interrupt to the TMS34020. Any instruction that generates an exception flag, such as invalid operation, will work.

Possible instructions include:

Divide using 0 as the dividend

Use NaN as the operand for any instruction

Unwrap the floating-point value one (unwrap ONE.f)

In order to distinguish an intentional ED interrupt from one generated by a real exception, a register or memory location should first be loaded with a status code. Then the illegal operation is performed. The TMS34020 interrupt service routine should read the register or memory location to determine if the interrupt was intentional. The routine should also reset the register or memory location.

Before causing the ED interrupt, the external routine should make sure the internal stack (registers SUBADDR0 and SUBADDR1) is empty. This can be accomplished by clearing the stack pointers (bit 31) in both registers. You may wish to save the contents of these registers in external memory *before* clearing the stack pointers.

5.8 TMS34020/TMS34082 Code Example

Using combinations of the MMPY0F, MMPY1F, MMPY2F,and MMPY3F single-precision floating-point multiply instructions allows for several matrix multiply operations: 1 × 3 by 3 × 3, 1 × 4 by 4 × 4, 3 × 3 by 3 × 3, and 4 × 4 by 4 × 4.The following example shows the use of MMPY0F, MMPY1F and MMPY2F in performing a single-precision floating-point 3 × 3 by 3 × 3 matrix multiply, giving a 3 × 3 matrix result.

Example 5–3. Multiplying Two 3 × 3 Matrices

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} & B_{02} \\ B_{10} & B_{11} & B_{12} \\ B_{20} & B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} & C_{02} \\ C_{10} & C_{11} & C_{12} \\ C_{20} & C_{21} & C_{22} \end{bmatrix}$$

Algorithm:

$C_{00} = A_{00} \times B_{00} + A_{01} \times B_{10} + A_{02} \times B_{20}$
$C_{01} = A_{00} \times B_{01} + A_{01} \times B_{11} + A_{02} \times B_{21}$
$C_{02} = A_{00} \times B_{02} + A_{01} \times B_{12} + A_{02} \times B_{22}$

$C_{10} = A_{10} \times B_{00} + A_{11} \times B_{10} + A_{12} \times B_{20}$
$C_{11} = A_{10} \times B_{01} + A_{11} \times B_{11} + A_{12} \times B_{21}$
$C_{12} = A_{10} \times B_{02} + A_{11} \times B_{12} + A_{12} \times B_{22}$

$C_{20} = A_{20} \times B_{00} + A_{21} \times B_{10} + A_{22} \times B_{20}$
$C_{21} = A_{20} \times B_{01} + A_{21} \times B_{11} + A_{22} \times B_{21}$
$C_{22} = A_{20} \times B_{02} + A_{21} \times B_{12} + A_{22} \times B_{22}$

Matrix values:

MATRIX A =	10	0	11
	–3	–1	–5
	13	1	6
MATRIX B =	3	1	5
	2	1	3
	4	–1	1
MATRIX C =	76	–1	61
	–32	1	–23
	65	8	74

Example 5–4. Instructions for a 3 × 3 by 3 × 3 Matrix Multiply

```
; Code for multiplying one 3 x 3 by another 3 x 3 matrix

   .IEEEFL                  ; Force IEEE floating-point representations

BEGIN;

; Move matrix B to the TMS34082
   MOVI       MATRIXB,A0
   MOVF       *A0+,RA0, 10
   MOVF       *A0+,RB0,6

; Point A0 to first row of matrix A
   MOVI       MATRIXA,A0

; Point A1 to first row of matrix C
   MOVI       MATRIXC, A1
   MOVI       3, A2
                            ; three rows

ROWLOOP;
; Loop through all three rows
   MOVF       *A0+,RB9,1    ; Movefirst A value on row to the TMS34082
   MMPY0F                   ; Multiply down the B column
   MOVF       *A0+, RB9, 1  ; Move second A value on row to the TMS34082
   MMPY1F                   ; Multiply and accumulate down the second B column
   MOVF       *A0+, RB9, 1  ; Move third A value on row to the TMS34082
   MMPY2F                   ; Multiply and accumulate down the third B column

; Move the current C row into TMS34020 memory
   MOVF       RB6, *A1+, 3  ; Get the three row values
   DEC        A2            ; Done four rows yet?
   JRNZ       ROWLOOP       ; If no, then compute the next row

HERE;         JRUC    HERE  ; Done, endless loop

;Matrix storage
   .SECT "DATA"

MATRIXA
   .FLOAT     10,   0,  11
   .FLOAT     -3,  -1,  -5
   .FLOAT     13,   1,  16

MATRIXB
   .FLOAT      3,   1,   5,  0       ; The zeros on the end of these rows are
   .FLOAT      2,   1,   3,  0       ; not necessary, but allow a memory-to-
   .FLOAT      4,  -1,   1,  0       ; register transfer for the matrix.
   .FLOAT      0,   0,   0,  0       ; This row of zeros is necessary

MATRIXC
   .FLOAT      0,   0,   0
   .FLOAT      0,   0,   0
   .FLOAT      0,   0,   0

.SECT         "TEXT"
```

5.9 TMS34020/TMS34082 Timing Examples

The following timing diagrams illustrate the timing relationships between the TMS34020 and TMS34082.

Figure 5–8 shows the multiplication of two double-precision numbers in TMS34020 registers and assumes that the TMS34020 instructions are contained in cache. The assembler source code is shown below.

Example 5–5. Assembler Source for Double-Precision Multiply

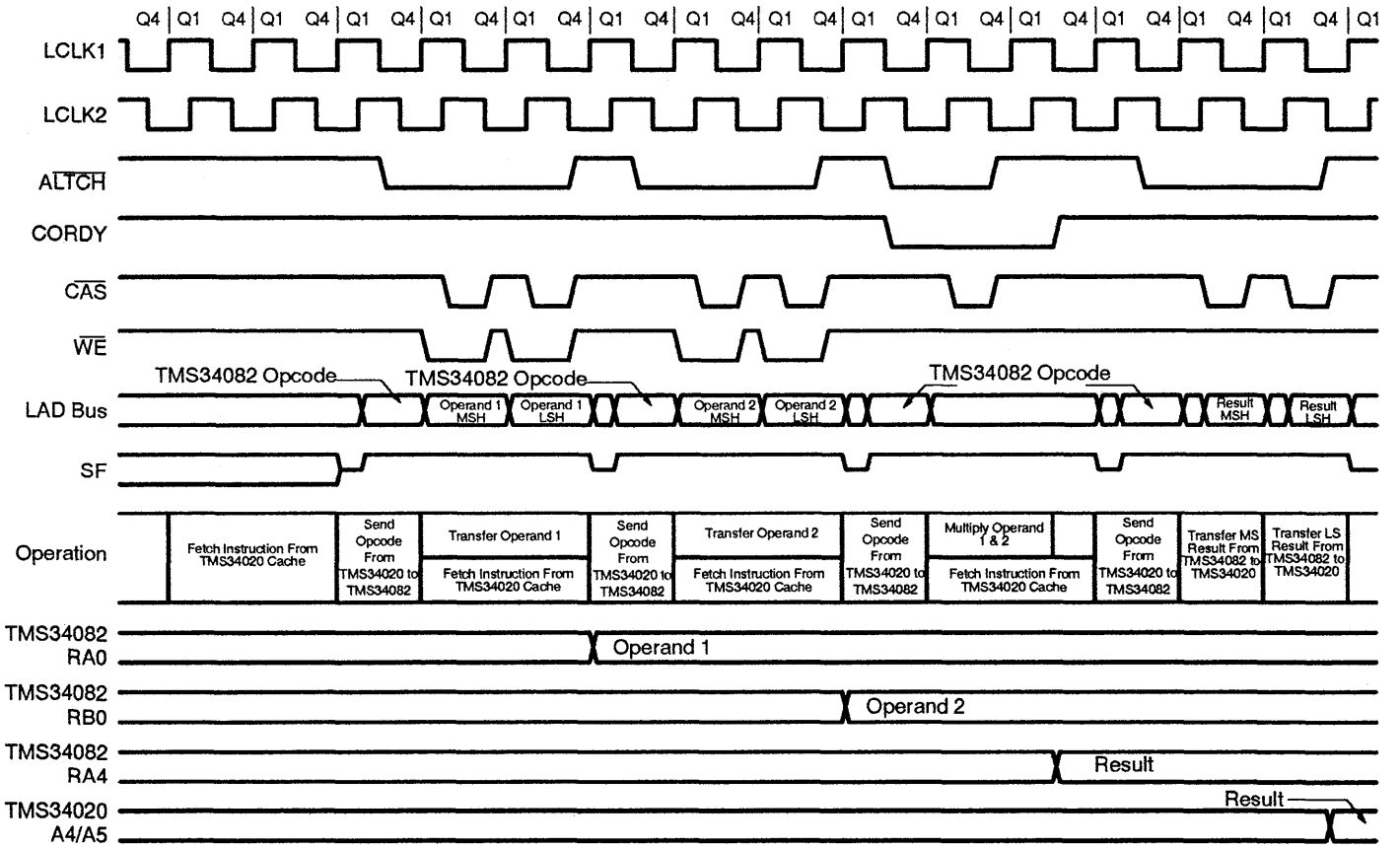
```
MOVD   A0, A1, RA0
MOVD   A2, A3, RB0
MPYD   RA0, RB0, RA4
MOVD   RA4, A4, A5
```

Figure 5–9 shows an add operation for two single-precision numbers from DRAM assuming that the TMS34020 instructions are contained in cache. The assembler source code is shown below.

Example 5–6. Assembler Source for Single-Precision Add

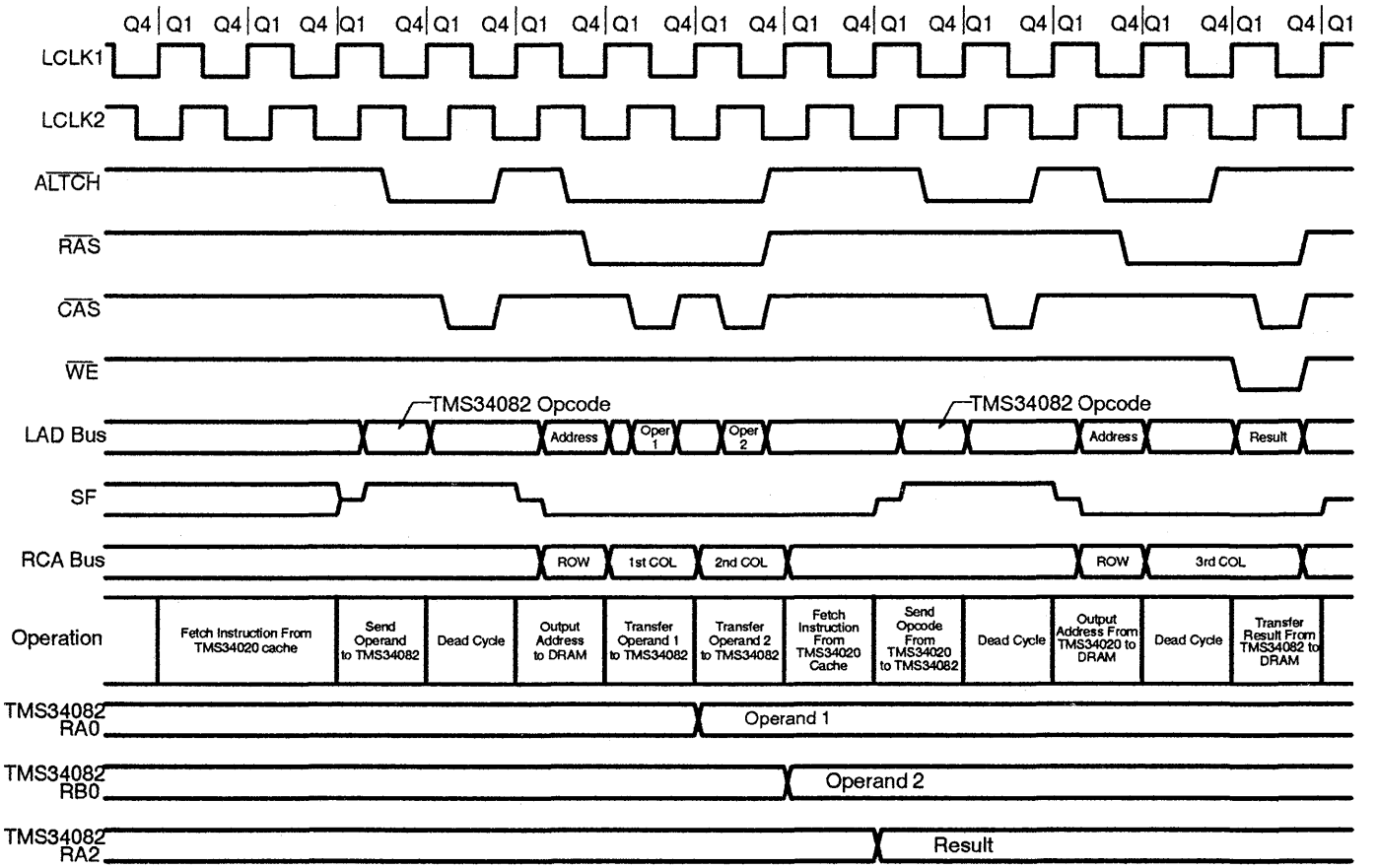
```
ADDF          *A0+, RA0, RB0, RA2
MOVF          RA2, *A1+
```

Figure 5–10 shows the same add operation (adding two single-precision numbers from DRAM). However, this time the TMS34020 instructions are not in cache.



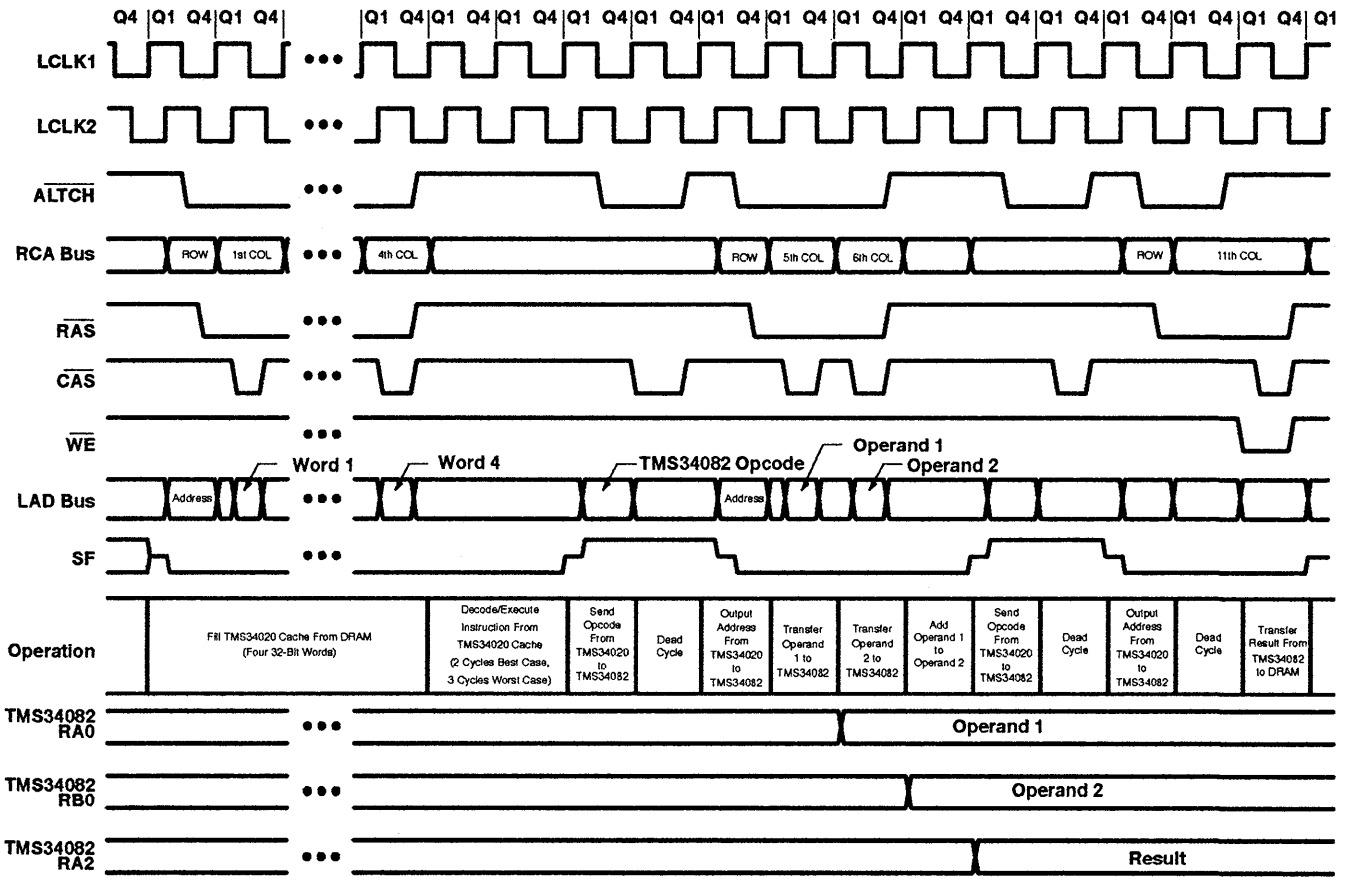
Note: Assume instructions are in TMS34020 cache, TMS34082 pipeline registers turned on (PIPES1=0) and output registers turned off (PIPES2=1), TMS34082 load order is MSH, then LSH.

Figure 5–8. Multiply 2 Double-Precision Numbers in TMS34020 Registers and Store Result Back to TMS34020 Register (Mode 0)



Note: Assume instructions are in TMS34020 cache, TMS34082 pipeline registers turned on (PIPES1=0) and output registers turned off (PIPES2–1), DRAM page mode accesses.

Figure 5–9. Add 2 Single-Precision Numbers from DRAM and Store Result Back to DRAM (Mode 2)



Note: Assume instructions are not in TMS34020 cache, TMS34082 pipeline registers turned on (PIPES1=0) and output registers turned off (PIPES2=1) DRAM page mode accesses.

Figure 5–10. Add 2 Single-Precision Numbers from DRAM and Store Result Back to DRAM (Mode 2), Instructions Not in TMS34020 Cache

5.10 MSD Bus Operation in Coprocessor Mode

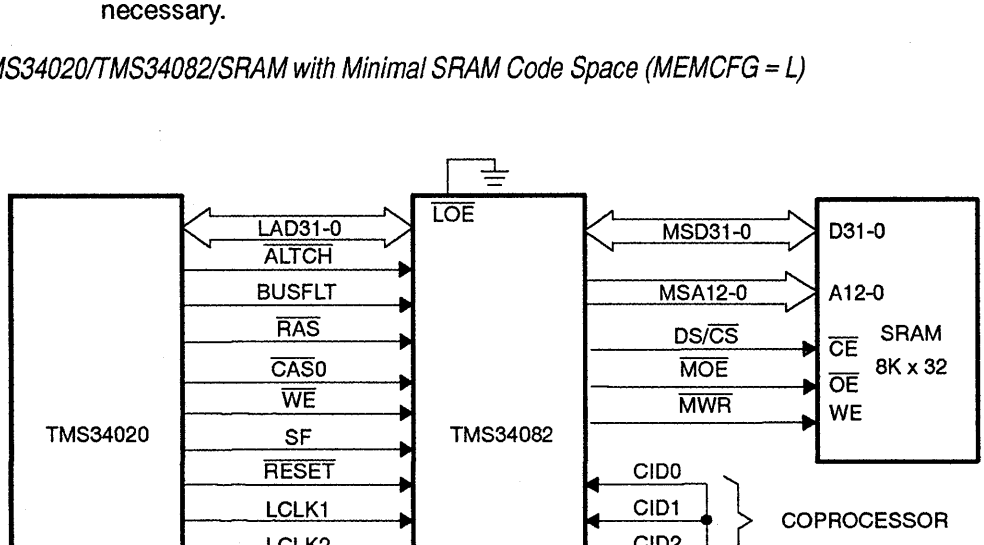
Use of the MSD bus in coprocessor mode is optional. External memory on MSD31-0 can be used to store data, user-programmed subroutines, or both. External instructions for user-defined subroutines are covered in Chapter 8. Control signals for MSD and MSA buses, discussed in subsection 4.3.2, operate the same in host-independent and coprocessor modes. Different combinations of control signals distinguish between data memory and code memory.

Data or program code can be downloaded to external memory from the LAD bus. The data (or code) can be stored in the TMS34020's DRAM/VRAM memory and loaded by a LAD-to-MSD bus transfer.

5.10.1 Connecting External Memory

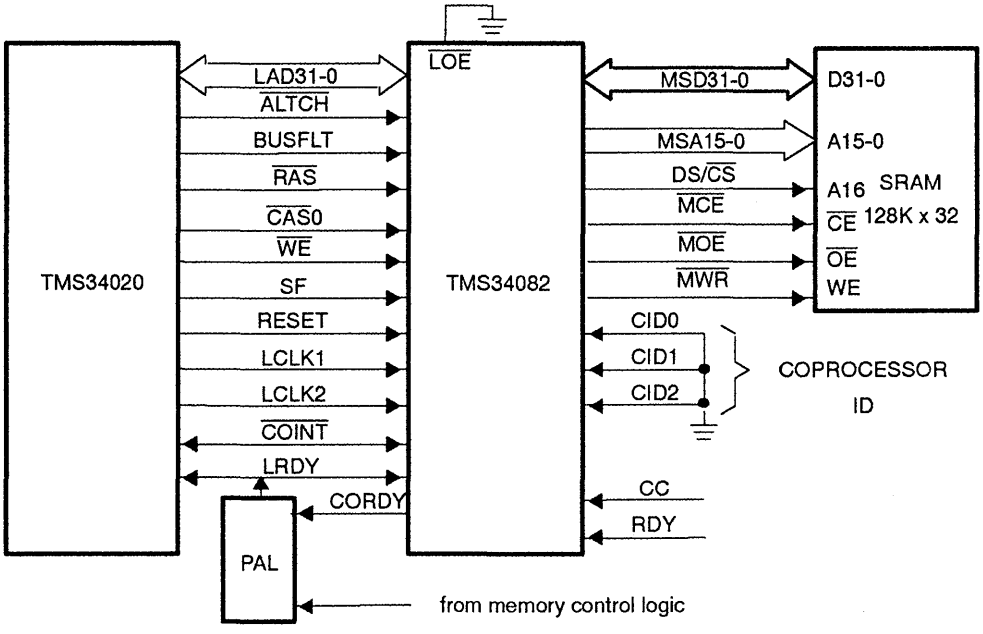
External coprocessor code space is added to the TMS34082 MSD port by adding external SRAM as shown in Figure 5–11. No external glue logic is necessary.

Figure 5–11. TMS34020/TMS34082/SRAM with Minimal SRAM Code Space (MEMCFG = L)



The maximum amount of external memory directly addressable by the TMS34082 is 64K words of program code and 64K words of data as shown in Figure 5–13. This comes out to 512K bytes total. When additional memory is necessary, segmentation or paging techniques can be utilized.

Figure 5–12. TMS34020/TMS34082/SRAM with Maximum SRAM Code/Data Space (MEMCFG = L)



CC is a condition code input and may be used as an external input for branch conditions in external code. It is not used in internal instructions.

5.10.2 TMS34082 External SRAM Timing Analysis

When connecting external SRAM to the TMS34082 for code space and/or data space on the MSD port, the following calculations can be used in determining the total SRAM access time. These times must also include any chip select decode delays. The general formula for computing SRAM access times is:

$$(1/2) \times t_{c(LC1)} - t_{su(MSD)} - t_{p(LC1-MSAV)} = \text{SRAM access speed}$$

A description of these parameters is provided in Table 5–4.

Table 5–4. Parameters Used for Calculating SRAM Speed

Parameter	Description
$t_{c(LC1)}$	Local clock LCLK1 period: $1/f_{clock}$
$t_{su(MSD)}$	Setup time: MSD data before LCLK1 high
$t_{p(LC1-MSAV)}$	Propagation delay: LCLK1 to MSA valid

The time delay incurred by inserting decode logic between the TMS34082 and external SRAM memory would be subtracted from the left side of the equation. For example, if an SN74AS32 (with a propagation delay of 6 ns maximum) is used in generating the SRAM chip enable ($\overline{CE}$), then the SRAM access time requirements would subsequently be decreased by 6 ns.

5.10.3 Using External Code

Adding external memory to the MSD port allows you to write customized subroutines for your applications. External code is executed by performing a jump to subroutine command issued by the TMS34020.

The memory space is divided into a jump table and general-purpose memory for code and data, as shown in Figure 5–13. There are 32 entries into the subroutine jump table. The jump entry points start at address 0 and increment by 2. This allows two instructions (in the jump table) per subroutine. Using this memory organization, the jump table is relatively small, leaving the remaining memory to be partitioned as best suits your application.

Figure 5–13. Memory Map for External Memory

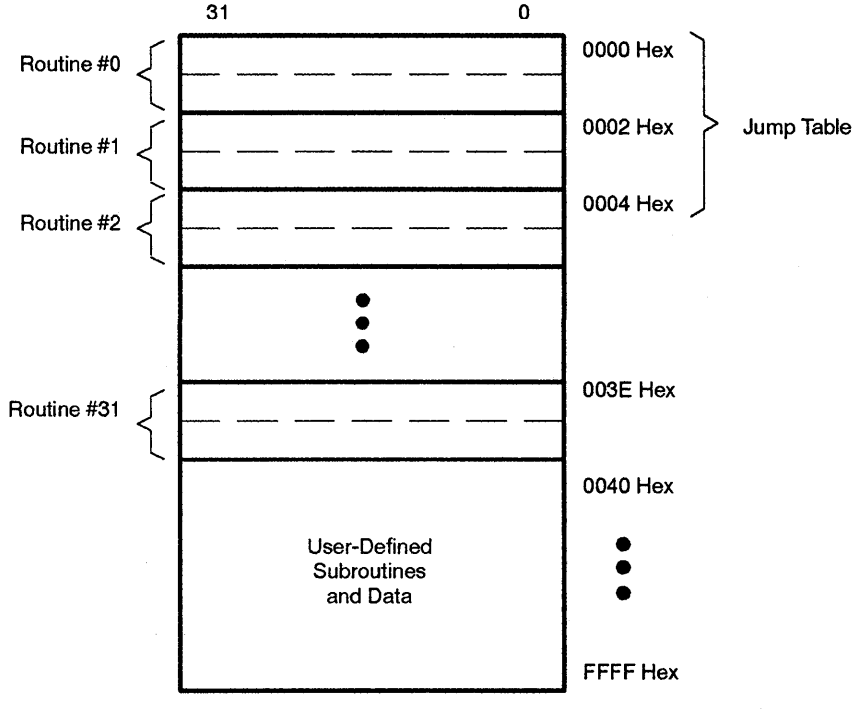
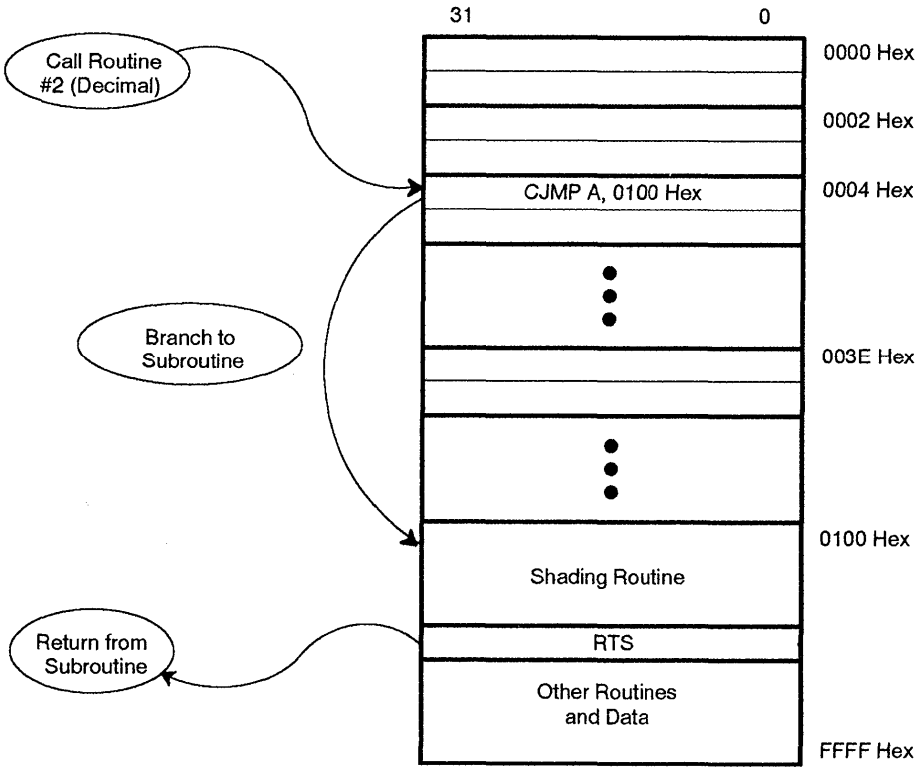


Figure 5–14 illustrates how an external subroutine would execute. The final instruction in the subroutine should be a return from subroutine (RTS). This puts the TMS34082 in an idle mode, waiting for the next instruction from the TMS34020.

Note: Before executing the final return from subroutine, the stack (SUBADDR1-0) must be empty. You may wish to save the contents of these registers in external memory. Then clear the stack pointers (bit 31) in both registers.

Figure 5–14. Example Subroutine Using the Jump Table



5.11 TMS34020/TMS34082/SRAM Code Example

This example describes a 3×3 by 3×3 matrix multiply routine using a subroutine stored in TMS34082 external SRAM. Data values for both matrices are stored in DRAM/VRAM. Therefore, they must be fetched from memory and transferred to RA8-0 and RB8-0 (using the memory address pointers contained in TMS34020 registers B1 and B2, respectively).

Description of operation:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} & B_{02} \\ B_{10} & B_{11} & B_{12} \\ B_{20} & B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} & C_{02} \\ C_{10} & C_{11} & C_{12} \\ C_{20} & C_{21} & C_{22} \end{bmatrix}$$

Algorithm:

$$C_{00} = A_{00} \times B_{00} + A_{01} \times B_{10} + A_{02} \times B_{20}$$
$$C_{01} = A_{00} \times B_{01} + A_{01} \times B_{11} + A_{02} \times B_{21}$$
$$C_{02} = A_{00} \times B_{02} + A_{01} \times B_{12} + A_{02} \times B_{22}$$

$$C_{10} = A_{10} \times B_{00} + A_{11} \times B_{10} + A_{12} \times B_{20}$$
$$C_{11} = A_{10} \times B_{01} + A_{11} \times B_{11} + A_{12} \times B_{21}$$
$$C_{12} = A_{10} \times B_{02} + A_{11} \times B_{12} + A_{12} \times B_{22}$$

$$C_{20} = A_{20} \times B_{00} + A_{21} \times B_{10} + A_{22} \times B_{20}$$
$$C_{21} = A_{20} \times B_{01} + A_{21} \times B_{11} + A_{22} \times B_{21}$$
$$C_{22} = A_{20} \times B_{02} + A_{21} \times B_{12} + A_{22} \times B_{22}$$

The register file contents before the routine are:

$RA0 = A_{00}$	$RB0 = B_{00}$
$RA1 = A_{01}$	$RB1 = B_{01}$
$RA2 = A_{02}$	$RB2 = B_{02}$
$RA3 = A_{10}$	$RB3 = B_{10}$
$RA4 = A_{11}$	$RB4 = B_{11}$
$RA5 = A_{12}$	$RB5 = B_{12}$
$RA6 = A_{20}$	$RB6 = B_{20}$
$RA7 = A_{21}$	$RB7 = B_{21}$
$RA8 = A_{22}$	$RB8 = B_{22}$

The register file contents after the routine are:

RA0 = C_{00}	RB0 = B_{00}
RA1 = C_{01}	RB1 = B_{01}
RA2 = C_{02}	RB2 = B_{02}
RA3 = C_{10}	RB3 = B_{10}
RA4 = C_{11}	RB4 = B_{11}
RA5 = C_{12}	RB5 = B_{12}
RA6 = C_{20}	RB6 = B_{20}
RA7 = C_{21}	RB7 = B_{21}
RA8 = C_{22}	RB8 = B_{22}
CT = unknown	

Examples 5–7 and 5–8 are the assembly language source listings for both the TMS34020 and the TMS34082. The TMS34082 listing is for the TMS34082 external matrix multiply instructions contained in SRAM. Assume that the matrix multiply routine begins at address 3Eh in SRAM and that the SRAM area for constants is from address FEh through FFh. The timing diagram for this example is shown in Figure 5–15.

Example 5–7. TMS34020 Assembler Listing for 3 × 3 by 3 × 3 Matrix Multiply

```
MOVEF *B1+, RA0, 9   ; move first matrix to coprocessor register file A,
                     ; starting at memory address contained in 34030
                     ; register B1
MOVEF *B2+, RB0, 9   ; move second matrix to coprocessor register file B,
                     ; starting at register file B, memory address
                     ; contained in 34020 register B2
CEXEC 0, 0000FFF     ; coprocessor jump to external routine #31 decimal, at
                     ; SRAM address 3Eh
```

Example 5–8. TMS34082 Assembler Listing for 3 × 3 by 3 × 3 Matrix Multiply

```
segment     code,memtype=0
    cjmp A, MAT
      ; jump to matrix multiply routine
MAT: ld CONFIG.i, all_pipes, 1
      ; load CONFIG register to turn on output registers (PIPES2=0)
    mult RA0.f, RB0.f, CT
      ; A00 * B00
    mult RA0.f, RB1.f, C
      ; A00 * B01
    mult.pass RA1.f, MULFB, RB3.f, CT, MULT
      ; A01 * B10, (A00 * B00) + 0
    mult.pass RA1.f, MULFB, RB4.f, CT, MULT
      ; A01 * B11, (A00 * B01) + 0
    mult.add RA2.f, MULFB, RB6.f, ALUFB, CT, ALU
      ; A02 * B20, (A01 * B10) + (A00 * B00)
    mult.add RA2.f, MULFB, RB7.f, ALUFB, CT, ALU
      ; A02 * B21, (A01 * B11) + (A00 * B01)
    mult.add RA0.f, MULFB, RB2.f, ALUFB, RA0, ALU
      ; A00 * B02, (A02 * B20) + ((A01 * B10) + (A00 * B00)) = C00
    mult.add RA3.f, MULFB, RB0.f, ALUFB, RA1, ALU
      ; A10 * B00, (A02 * B21) + ((A01 * B11) + (A00 * B01)) = C01
    mult.pass RA1.f, MULFB, RB5.f, CT, MULT
      ; A01 * B12, (A00 * B02) + 0
    mult.pass RA4.f, MULFB, RB3.f, CT, MULT
      ; A11 * B10, (A10 * B00) + 0
    mult.add RA2.f, MULFB, RB8.f, ALUFB, CT, ALU
      ; A02 * B22, (A01 * B12) + (A00 * B02)
    mult.add RA5.f, MULFB, RB6.f, ALUFB, CT, ALU
      ; A12 * B20, (A11 * B10) + (A10 * B00)
    mult.add RA3.f, MULFB, RB1.f, ALUFB, RA2, ALU
      ; A10 * B01, (A12 * B22) + ((A01 * B12) + (A00 * B02)) = C02
    mult.add RA3.f, MULFB, RB2.f, ALUFB, RA3, ALU
      ; A10 * B02, (A12 * B20) + ((A11 * B10) + (A10 * B00)) = C10
    mult.pass RA4.f, MULFB, RB4.f, CT, MULT
      ; A11 * B11, (A10 * B01) + 0
    mult.pass RA4.f, MULFB, RB5.f, CT, MULT
      ; A11 * B12, (A10 * B02) + 0
    mult.add RA5.f, MULFB, RB7.f, ALUFB, CT, ALU
      ; A12 * B21, (A11 * B11) + (A10 * B01)
    mult.add RA5.f, MULFB, RB8.f, ALUFB, CT, ALU
      ; A12 * B22, (A11 * B12) + (A10 * B02)
    mult.add RA6.f, MULFB, RB0.f, ALUFB, RA4, ALU
      ; A20 * B00, (A12 * B21) + ((A11 * B11) + (A10 * B01)) = C11
    mult.add RA6.f, MULFB, RB1.f, ALUFB, RA5, ALU
      ; A20 * B01, (A12 * B22) + ((A11 * B12) + (A10 * B02)) = C12
    mult.pass RA7.f, MULFB, RB3.f, CT, MULT
      ; A21 * B10, (A20 * B00) + 0
    mult.pass RA7.f, MULFB, RB4.f, CT, MULT
      ; A21 * B11, (A20 * B01) + 0
    mult.add RA8.f, MULFB, RB6.f, ALUFB, CT, ALU
      ; A22 * B20, (A21 * B10) + (A20 * B00)
```

Example 5–8. TMS34082 Assembler Listing for 3x3 by 3x3 Matrix Multiply (Continued)

```
      mult.add RA8.f, MULFB, RB7.f, ALUFB, CT, ALU
         ; A22 * B21, (A21 * B11) + (A20 * B01)
      mult.add RA6.f, MULFB, RB2.f, ALUFB, RA6, ALU
         ; A20 * B02, (A22 * B20) + ((A21 * B10) + (A20 * B00)) = C20
      mult.add RA7.f, MULFB, RB5.f, ALUFB, RA7, ALU
         ; A21 * B12, (A22 * B21) + ((A21 * B11) + (A20 * B01)) = C21
      mult.pass RA8.f, MULFB, RB8.f, CT, MULT
         ; A22 * B22, (A20 * B02) + 0
      pass MULFB.f, RA8
         ; (A21 * B12) + 0
      add MULFB.f, ALUFB.f, CT
         ; (A22 * B22) + (A20 * B02)
      nop
         ; no operation
      add RA8.f, ALUFB, RA8.f
         ; (A21 * B12) + ((A22 * B22) + (A20 * B20)) = C22
      nop
         ; no operation
      nop
         ; no operation
      ld CONFIG.i, pipeline_only, 1
         ; load configuration register to turn off output registers (PIPES2=1)
      rts
         ; return from subroutine, go to internal TMS34082 wait state
.segment data,memtype=1
      all_pipes:        .data     0xFFC08
         ; CONFIG register setting for all pipeline registers enabled
      pipeline_only:    .data     0xFFC28
         ; CONFIG register setting to turn off output registers only
```

Figure 5-15. 3 x 3 Matrix Multiply Using External SRAM for Data Space and Code Space (Mode 3)

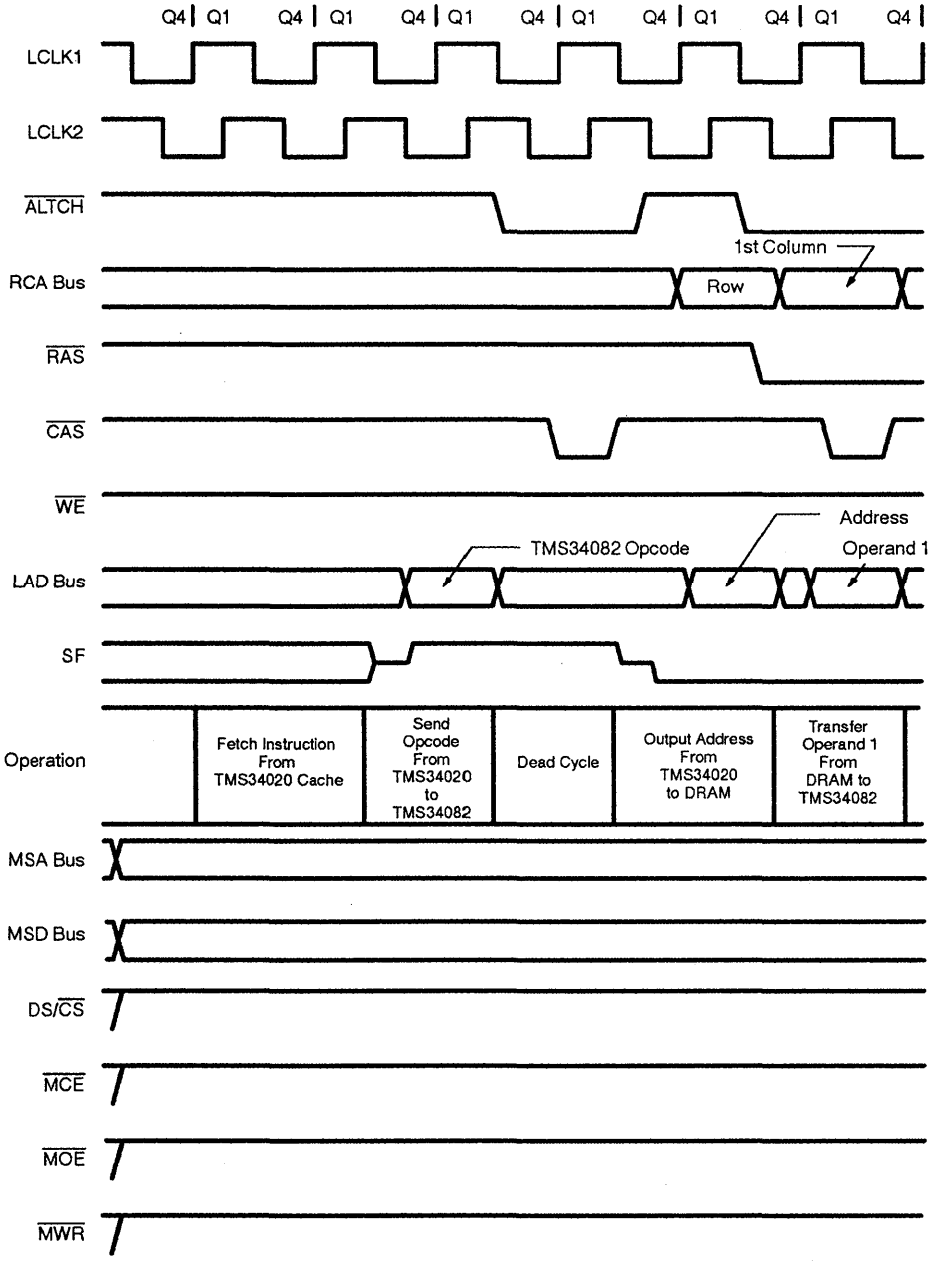


Figure 5-15. 3 × 3 Matrix Multiply Using External SRAM for Data Space and Code Space (Mode 3) (Continued)

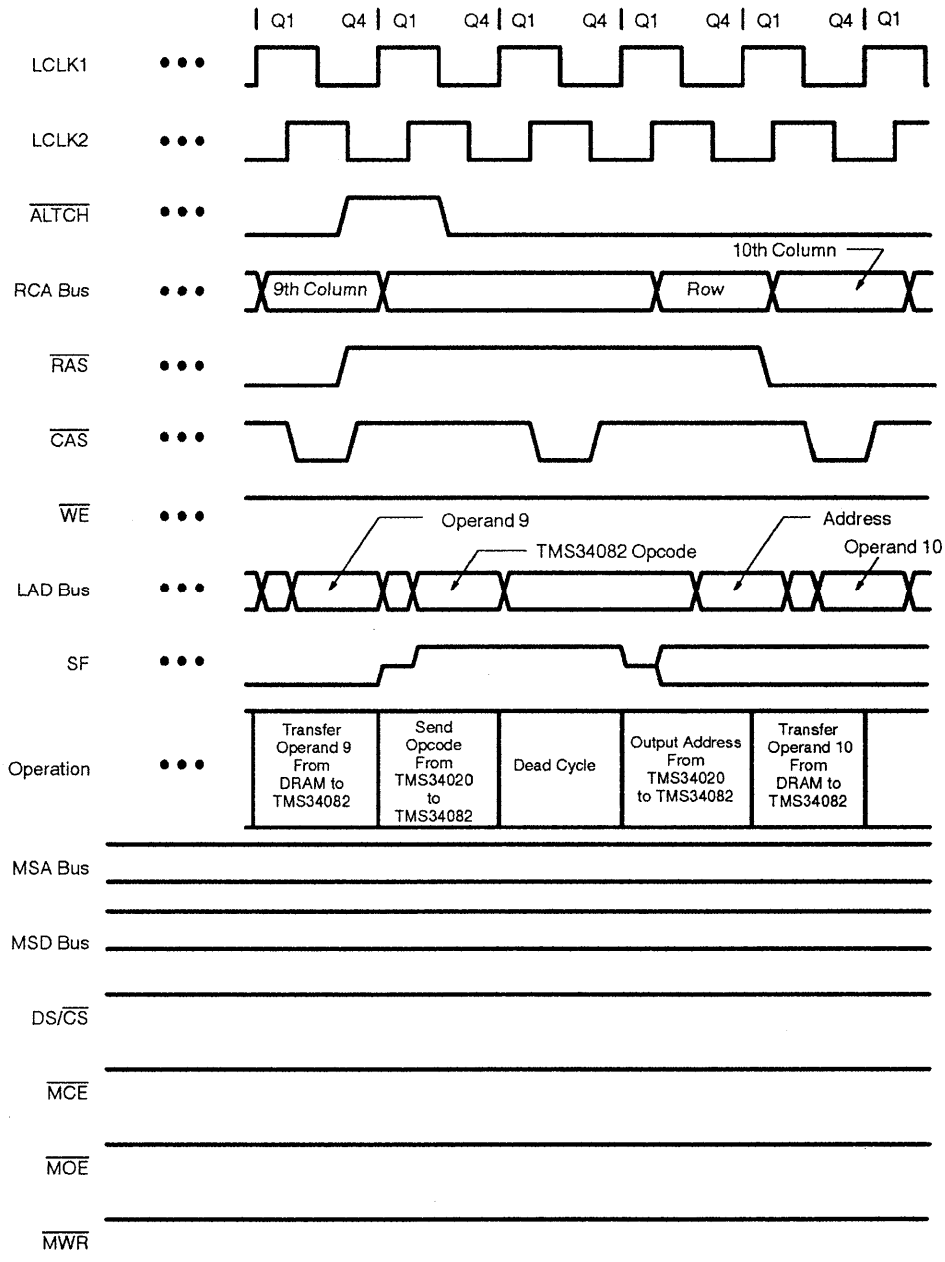


Figure 5-15. 3 × 3 Matrix Multiply Using External SRAM for Data Space and Code Space (Mode 3) (Continued)

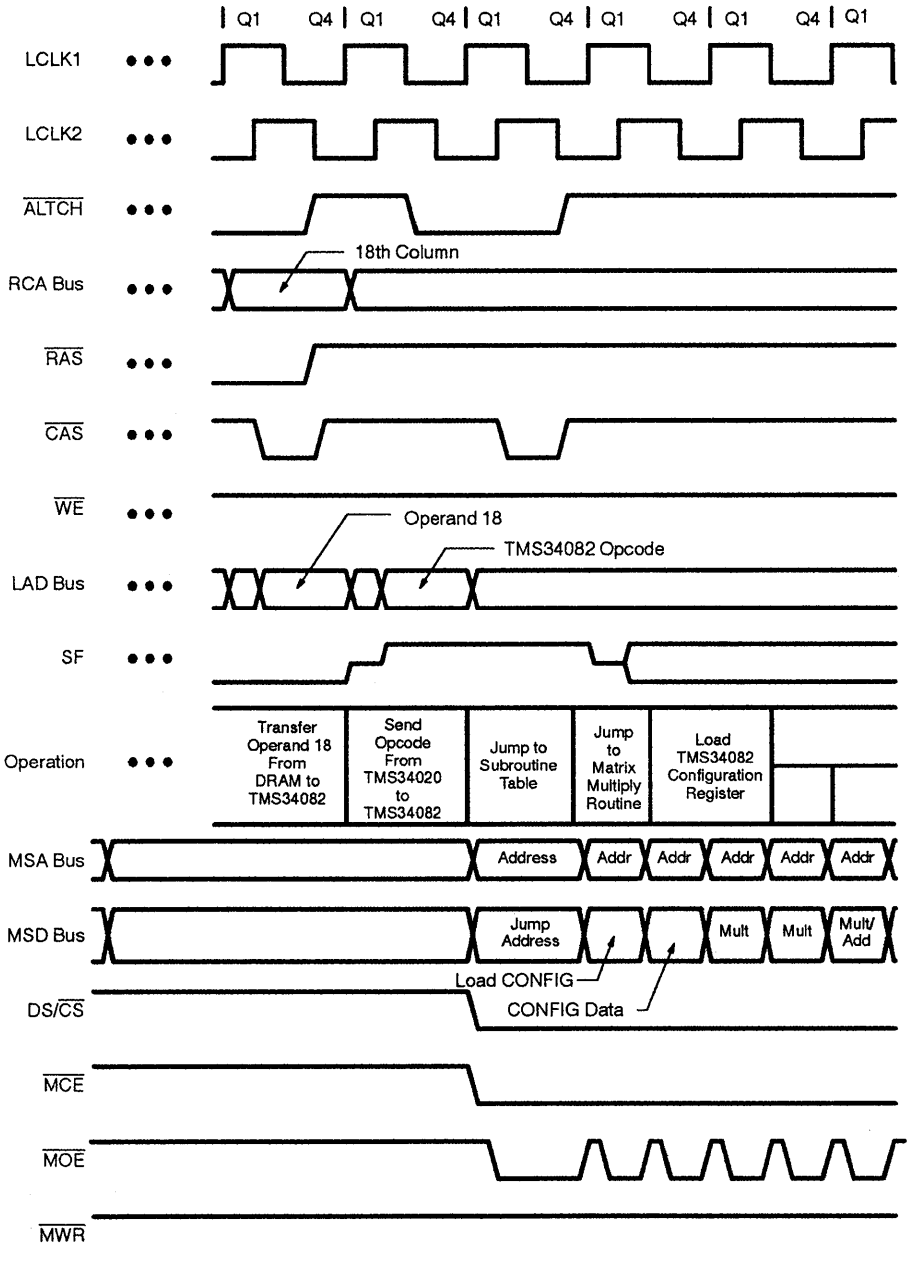
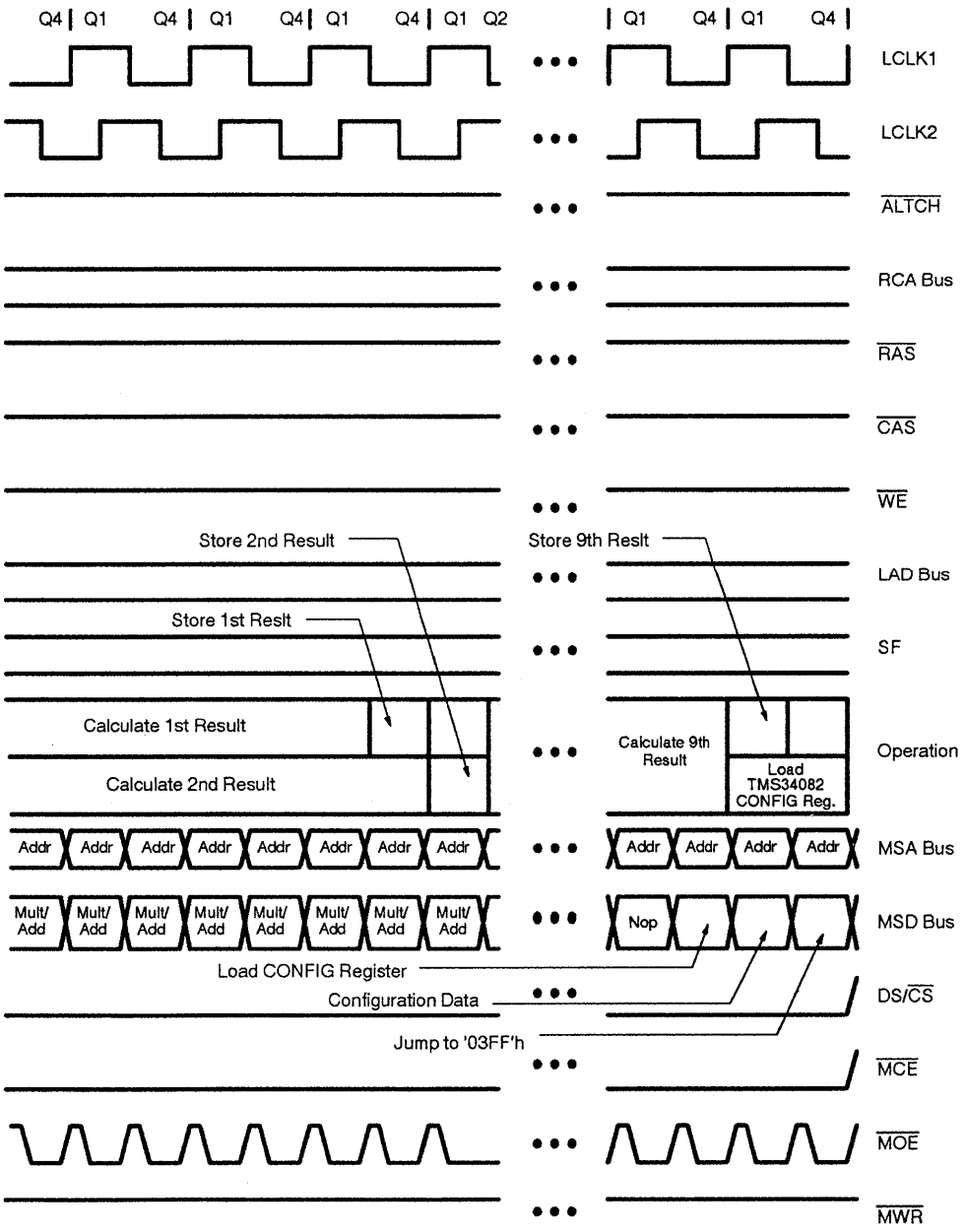


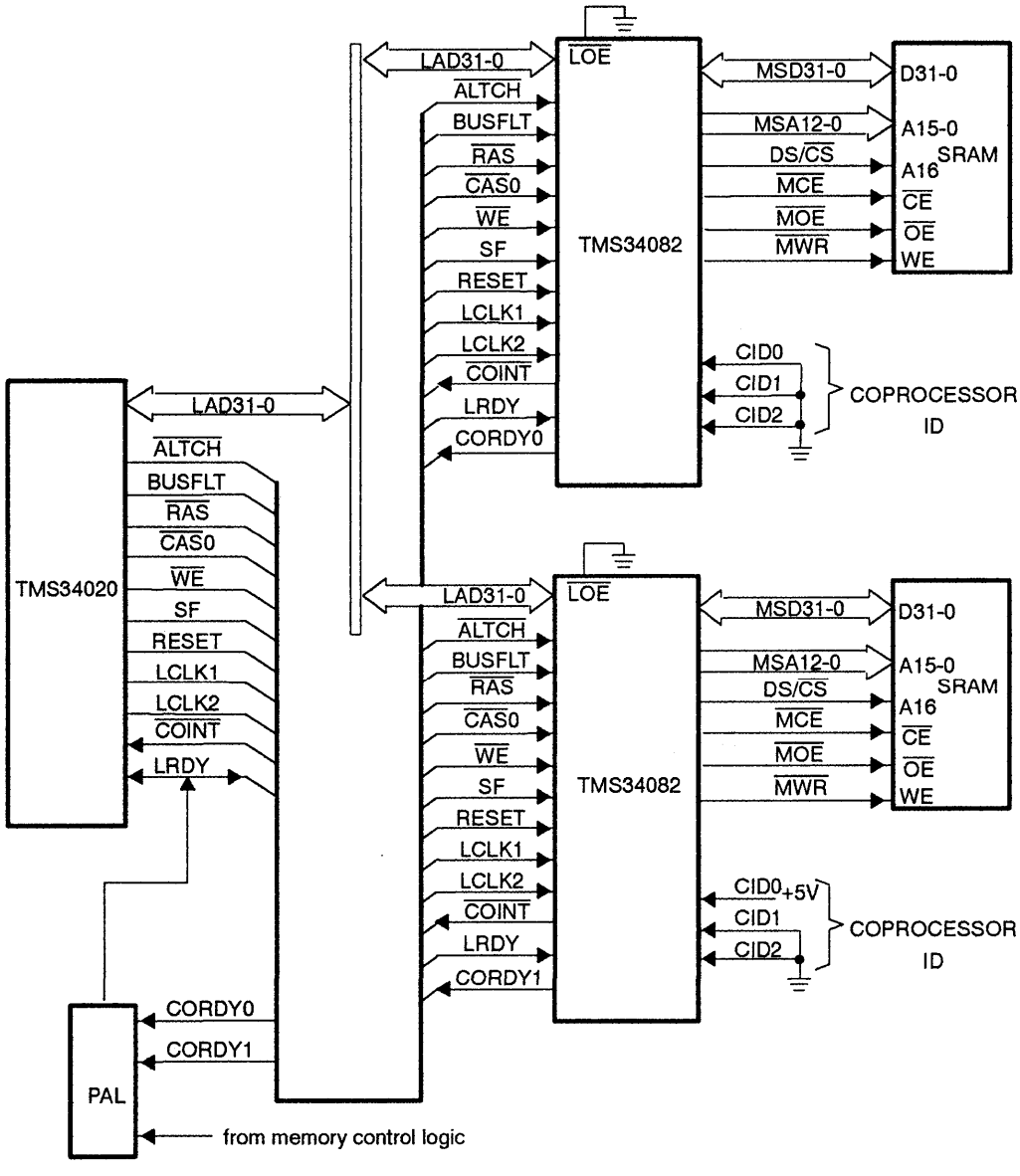
Figure 5-15 3× 3 Matrix Multiply Using External SRAM for Data Space and Code Space (Mode 3) (Continued)



5.12 Multiple TMS34082s

More than one coprocessor may be connected to the TMS34020 by setting the appropriate coprocessor ID field (CID2-0). Up to seven TMS34082s may be used with each TMS34020. See Figure 5–16. Assuming that each TMS34082 CORDY pin has a separate pull-up resistor, the TMS34020 can determine which coprocessors are present in the system by writing to and reading from TMS34082 register locations.

Figure 5–15. TMS34020 with Multiple TMS34082/SRAM Blocks (MEMCFG = L)



When CID2-0 = 100_2, the TMS34020 broadcasts the instruction to all coprocessors. Broadcast reads by the TMS34082s are not permitted and are ignored.

Using the TMS34020 assembler directive called **.coproc**, the coprocessor ID number (between 0 and 7) may be set for generic coprocessor instructions. This directive maintains the coprocessor ID until another directive is received. An example follows where the default coprocessor ID is set to 1 and then to 0.

Example 5–9. Assembler Code for Multiple TMS34082s

```
.coproc 1     ; set the default coprocessor ID to 001 for the following
              ; instructions
  MPYF        RA2,RB0,RA8
  ADDF        RA8,RB2,RA5
  SQRTF       RA5,RA5
.coproc 0     ; set the default coprocessor ID to 000 for the following
              ; instructions
  SUB         RA0,C,RA0
  SUB         RA1,C,RA1
```

Thus, while coprocessor 1 is still calculating its floating-point square root, coprocessor 0 is performing integer subtracts.For additional details on the assembler directives, refer to the *TMS340 Family Code Generation Tools User's Guide*.

Chapter 6

Host-Independent Mode

Operation in the host-independent mode assumes that the MSTR input signal is set high. The TMS34082 has several hardware control signals, as well as programmable features, which support system functions such as initialization, data transfer, or interrupts in host-independent mode. Details of initialization, LAD bus (LAD31-0) and MSD bus (MSD31-0) interface control, and interrupt handling are provided in this chapter.

6.1 Initialization

The following sections detail pin connections and initialization in host-independent mode.

6.1.1 Pin Connections

When operating in host-independent mode, you should connect TMS34082 pins as shown in Table 6–1.

Table 6–1. Pin Connections

Signal Name	Description	Logic Level
SF	Special function input; not used in host-independent mode	tie low
RAS	Row Address Strobe; not used in host-independent mode	tie low
CID2-0	Coprocessor ID; not used in host-independent mode	tie low
LCLK1-2	Local clocks for coprocessor mode	tie low
MSTR	Host-independent/coprocessor mode select	tie high
EC1-0	Emulator mode control	tie high
TCK	Test Clock	tie low

6.1.2 Bootstrap Loader

To simplify initialization of external program memory, the TMS34082 provides a bootstrap loader. Once invoked, the loader causes the TMS34082 to read 65 words from the LAD bus and write 64 words to the external program memory on the MSD bus. The first word read is used to initialize the configuration register. The remaining words are instructions written to the code space of external memory, starting at address 0.

To invoke the loader:

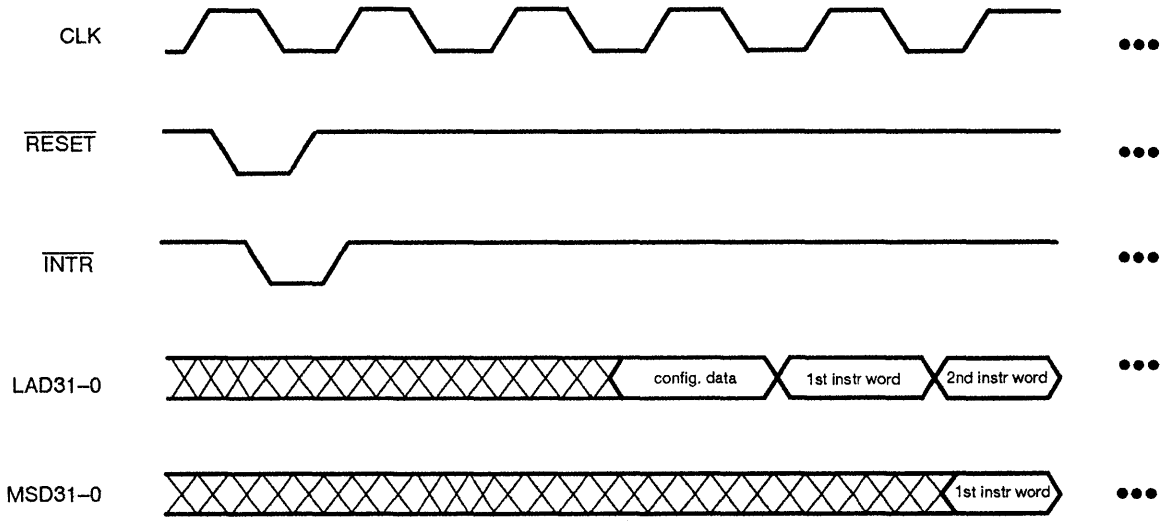
1) Set $\overline{\text{RESET}}$ low

2) Set $\overline{\text{INTR}}$ low

3) After the minimum pulse duration, set $\overline{\text{RESET}}$ and $\overline{\text{INTR}}$ high again

As shown in Figure 6–1, $\overline{\text{RESET}}$ must remain low while $\overline{\text{INTR}}$ is pulled low. During the initialization, the TMS34082 is reset. Internal states and status are cleared, but data registers are not affected; the control registers return to their default values.

Figure 6–1. Bootstrap Loader



Loader operation begins on the second clock cycle after $\overline{\text{RESET}}$ and $\overline{\text{INTR}}$ return high. The first word is read into the configuration register on the rising edge of the third clock. Each successive rising edge loads an instruction word. The instruction word is output on the MSD bus one clock cycle after it is input on the LAD bus.

Once the loader is activated, an external interrupt (signaled by $\overline{\text{INTR}}$ low) is not granted until the load sequence is finished. However, $\overline{\text{RESET}}$ going low terminates the loader. When the load sequence is finished, program execution begins at external address 0.

6.2 LAD Bus

In host-independent mode, the LAD bus is used to transfer data or instructions to and from the TMS34082 or the MSD bus. Instruction words may be transferred from the LAD bus to the MSD bus, but instructions cannot be input to the TMS34082 from the LAD bus. Details of LAD bus control and data input are given in the following sections.

6.2.1 Control Signals

Data transfers on the LAD bus are controlled primarily by the following signals:

$\overline{\text{ALTCH}}$, the address write strobe

$\overline{\text{CAS}}$, the memory read strobe

$\overline{\text{WE}}$, the memory write enable

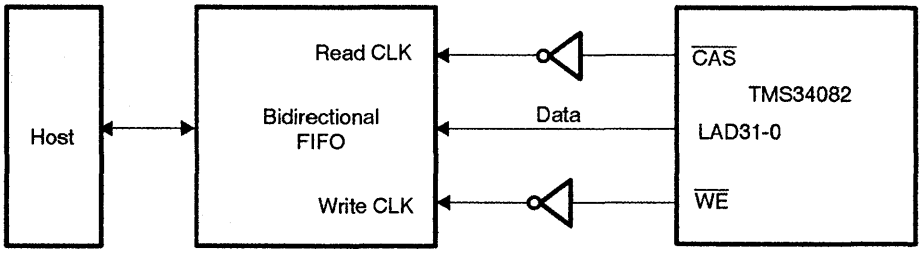
The TMS34082 outputs an address during a cycle when $\overline{\text{ALTCH}}$ is low. The address may be latched externally on the rising edge of $\overline{\text{ALTCH}}$. Because all 32 bits of the LAD bus can be used for an address, the LAD bus accesses up to 4G 32-bit words of memory.

When $\overline{\text{WE}}$ is low, data is output by the TMS34082 on the LAD bus. If multiple 32-bit words are output, $\overline{\text{WE}}$ toggles high at each rising clock edge, then returns low.

When $\overline{\text{CAS}}$ is low, the LAD bus is an input, reading data into the TMS34082. When multiple words are input, $\overline{\text{CAS}}$ toggles at each rising clock edge.

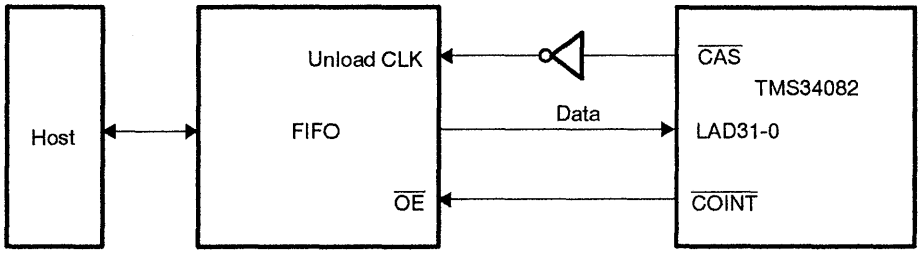
If a bidirectional FIFO is used instead of memory, $\overline{\text{CAS}}$ can be directly connected to the read clock and $\overline{\text{WE}}$ to the write clock. The CC input can be used to signal the TMS34082 when data is ready for input from the FIFO stack. (See Figures 6–2 and 6–3 for possible configurations.)

Figure 6–2. Using FIFOs on the LAD Bus



If LADCFG is set high in the configuration register, $\overline{\text{COINT}}$ defines bus cycle boundaries. If an indirect move to or from the LAD bus is coded with the C bit (bit 1) set high, $\overline{\text{COINT}}$ goes low at the beginning of the move and remains low until the move is complete. $\overline{\text{COINT}}$ can be used to select a device on the LAD bus, as shown in Figure 6–2. In this case, $\overline{\text{COINT}}$ is the output enable for a FIFO.

Figure 6–3. Using $\overline{\text{COINT}}$ as a Device Select (LADCFG=H)



The TMS34082 only drives the LAD bus during instructions that output an address or data. The LAD bus drivers are disabled at any other time.

$\overline{\text{LOE}}$, the LAD bus output enable, enables and disables the LAD bus. The LAD bus is placed in a high-impedance state when $\overline{\text{LOE}}$ is high. However, bringing $\overline{\text{LOE}}$ low does not cause the LAD bus drivers to turn on. The instruction being executed must also enable the drivers.

If no other processors share the LAD bus, $\overline{\text{LOE}}$ may be tied low. Other wise, $\overline{\text{LOE}}$ may be used to prevent bus conflicts between the TMS34082 and other system masters.

LADCFG controls the signals affected by $\overline{\text{LOE}}$. If LADCFG is high, setting $\overline{\text{LOE}}$ high also disables $\overline{\text{CAS}}$ and $\overline{\text{WE}}$. When LADCFG is low, $\overline{\text{COINT}}$ is a user-programmable output. $\overline{\text{LOE}}$ does not affect $\overline{\text{CAS}}$ or $\overline{\text{WE}}$.

6.2.2 Immediate Data Transfers

Data input on the LAD bus can be written to data registers, control registers, or passed through for output on the MSD bus. Alternatively, the LAD bus input can be selected directly as an FPU source operand without writing to a register.

The clock period may be extended for immediate data input that does not meet the minimum data setup time. The clock is stretched by the data delay plus 5 ns. Refer to TMS34082 data sheet timing diagrams for additional information.

An FPU result can be written to a data register and passed out to the LAD bus. When this is done, the minimum clock period is extended by 15 ns (TMS34082-40) to allow for the propagation delay from the FPU core to the outputs.

Depending on the specific system implementation, transferring data to and from the LAD bus without intervening register operations can significantly improve throughput. Data moves to and from internal registers can be minimized at the cost of adjusting the clock period to assure integrity of FPU results onto the LAD bus.

6.3 MSD Bus

The MSD bus can be used to access either external data memory or external code memory, depending on the combination of control signals required. In the host-independent mode, the MSD bus is the source for all instructions. Data can also be transferred to or from the TMS34082 over the MSD bus, and data transfers between the LAD and MSD buses are possible.

6.3.1 MSD Bus Control Signals

Up to 64K 32-bit data operands and 64K instructions may be directly addressed on the MSD bus. The address of memory is output on MSA15-0.

External memory operations are controlled by:

DS/$\overline{CS}$, data space/code space select

$\overline{MCE}$, memory chip enable

$\overline{MOE}$, memory output enable

$\overline{MWR}$, memory write enable

$\overline{MAE}$, MSD bus output enable

When memory configuration (MEMCFG) is low, DS/$\overline{CS}$ functions as the most significant address bit. DS/$\overline{CS}$ high selects data memory; DS/$\overline{CS}$ low selects code memory. $\overline{MCE}$ is the memory chip enable for both code and data memory.

When MEMCFG is high, DS/$\overline{CS}$ is the chip select for data memory and $\overline{MCE}$ is the chip select for code memory. This may eliminate the need for an external inverter.

The TMS34082 outputs data on the MSD bus when $\overline{MWR}$ and $\overline{MAE}$ are low. Otherwise, the device does not drive the MSD bus. If memory on the MSD bus is not shared, $\overline{MAE}$ can be tied low.

If the memory on the MSD port is shared with a host processor, the $\overline{MAE}$ and RDY signals can be used to prevent conflicts between the TMS34082 and the host processor. The host processor can monitor the state of $\overline{MCE}$ (for MEMCFG low) to determine when the TMS34082 is not accessing memory. If $\overline{MCE}$ is not active, the host processor takes control of the MSD bus by asserting $\overline{MAE}$ and RDY low. Setting RDY low halts the TMS34082.

6.3.2 Memory Models

The TMS34082 Software Tool Kit supports three memory models: small, medium, and large.

The small memory model places the code and data in the same memory space. DS/$\overline{CS}$ is unused. The maximum memory allowed is 64K 32-bit words, a combination of instructions and data.

The medium memory model uses separate data and code spaces. Up to 64K of data words and 64K of instructions are accessed.

The large memory model partitions the code space into banks, each containing 64K words. External segment registers determine which bank is being accessed. Constants are stored in the same bank as the code that uses them. Variable data is stored in memory on the LAD bus. For more information on segment register requirements, see the *TMS34082 Software Tool Kit User's Guide.*

6.4 Reset

The TMS34082 is reset when the $\overline{RESET}$ input is brought low. $\overline{RESET}$ is an asynchronous signal that requires no setup or hold times with respect to the clock. However, the minimum pulse duration requirement must be met. Data registers are not affected by reset.

Upon reset, all internal states and pipeline registers are cleared. Control registers return to their default values, except for the interrupt register which is unaffected. Data registers are also not affected by reset. The state of control signals during reset is listed in Chapter 4, Table 4–10.

The TMS34082 ignores the first rising clock edge after $\overline{RESET}$ is returned high. Program execution begins on the second cycle at address 0. $\overline{RESET}$ is also used in conjunction with the $\overline{INTR}$ signal to call a bootstrap loader. This operation is detailed in subsection 6.1.2.

6.5 Wait States

Setting RDY low causes the TMS34082 to stall. This input can be used to create wait states for slow memory accesses. Stalling the device does not affect any internal states or registers and output lines do not change.

In host-independent mode, LRDY can be used to stall the device. The function and timing are the same as RDY.

RDY (or LRDY) must be set low a minimum setup time before the rising clock edge you wish to inhibit. Operation resumes on the next rising clock edge after RDY (or LRDY) is set high. Again, there is a minimum setup time requirement before that clock edge.

6.6 User Programmable Outputs

In the host-independent mode, CORDY is a user-programmable output. If the LADCFG bit in the configuration register is low, $\overline{\text{COINT}}$ is also a user-programmable output. When LADCFG is high, $\overline{\text{COINT}}$ is used in LAD bus moves and is not programmable.

CORDY (or $\overline{\text{COINT}}$) is set high or low using the set mask instruction. CORDY (or $\overline{\text{COINT}}$) remains at that setting until it is changed by another set mask instruction. $\overline{\text{COINT}}$ and CORDY are set/reset independent of each other.

6.7 Conditional Code Input

The CC pin is an external condition code input. A conditional jump to subroutine or conditional branch can be performed based on the state of this pin.

The CC input allows you to control program flow based on some external status from other devices in your system. By polling this input, you can determine, for example, if a host processor has an instruction queued for the TMS34082.

6.8 Interrupts

The TMS34082 supports three types of interrupts in host-independent mode: hardware, software, and exception detects. Each of these has its own interrupt enable.

6.8.1 Hardware Interrupts

Upon power up or reset, hardware interrupts are disabled. Before enabling interrupts, the address of the interrupt handling routine should be stored in the interrupt address register. Hardware interrupts are enabled by setting INTENHW (bit 15 of the status register) high using the set mask instruction. A hardware interrupt is then signaled by setting $\overline{\text{INTR}}$ low.

When a hardware interrupt is received, the current program counter is pushed into the interrupt return register. The hardware interrupt flag, HINT (bit 4 of the status register), and interrupt grant, INTG, are set high. The interrupt mask is saved and all interrupts are disabled. The address in the interrupt vector is output to MSA15-0, causing a branch to the interrupt service routine.

After the interrupt service routine, the interrupts should be enabled again before a return from interrupt instruction is executed. Restoring the hardware interrupt clears the HINT flag and INTG.

Only one hardware interrupt may be queued. If a hardware interrupt is received while the first interrupt is being processed, the interrupt is recorded and serviced after the first interrupt sequence is finished. If a third or subsequent hardware interrupt is signaled, it will be ignored.

If a hardware interrupt is received during a multicycle instruction (such as divides, square roots, or moves), the interrupt is queued and serviced after the instruction is completed.

6.8.2 Software Interrupts

Upon power up or reset, software interrupts are disabled. Before enabling interrupts, the address of the interrupt handling routine should be stored in the interrupt address register. Software interrupts are enabled by setting INTENSW (bit 11 of the status register) high using the set mask instruction. An interrupt is then signaled by using the set mask instruction to send a software interrupt.

When a software interrupt is received, the current program counter is pushed into the interrupt return register. The software interrupt flag, INTFLG (bit 16 of the status register), and INTG is set high. The address in the interrupt vector is output to MSA15-0, causing a branch to the interrupt service routine.

The interrupts should be re-enabled before a return from interrupt instruction is executed. Restoring the software interrupt clears the HINT flag.

Because hardware interrupts may be queued, a hardware interrupt received while a software interrupt is being processed is recorded and serviced after the software interrupt is complete. This assumes the hardware interrupt was enabled before the software interrupt was received. If another hardware interrupt is signaled, it will be ignored.

6.8.3 Exception Detect Interrupts

A third type of interrupt is the exception detect interrupt. In the event of an FPU status exception in host-independent mode, the internal ED signal (bit 18 of the status register) is set high, causing an exception detect interrupt. If interrupts based on specific exceptions are not desired, the exceptions can be masked from the error detect (ED) logic by using the appropriate bits in the configuration register.

Upon power up or reset, exception detect interrupts are disabled. Before enabling interrupts, the address of the exception handling routine should be stored in the interrupt address register. Exception interrupts are enabled by setting INTENED (bit 12 of the status register) high using the set mask instruction.

When an error is detected and ED interrupts are enabled, the current program counter is pushed into the interrupt return register. ED is set high. The address in the interrupt vector is output to MSA15-0, causing a branch to the interrupt service routine.

The interrupts should be restored before a return from interrupt instruction is executed. Restoring interrupts clears the ED flag.

Because hardware interrupts may be queued, a hardware interrupt received while an exception interrupt is being processed is recorded and serviced after the first interrupt is finished. This assumes the hardware interrupt was enabled before the exception interrupt was received. If another hardware interrupt is signaled, it will be ignored.

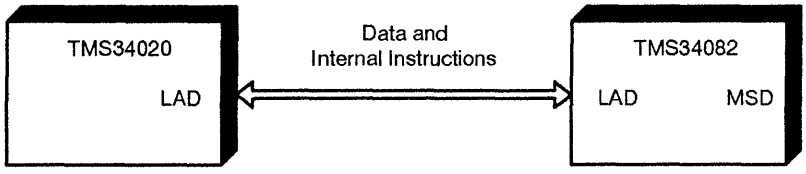
Chapter 7

Internal Instructions

The TMS34082 internal instruction set includes arithmetic and logical operations, as well as complex instructions stored in an internal program ROM. Several addressing modes are available for internal instructions in addition to data types for integer, single- and double-precision floating-point formats.

In the coprocessor mode, the TMS34082 executes internal instructions through the LAD bus as shown in Figure 7–1.

Figure 7–1. Source for Internal Instructions in Coprocessor Mode



In the host-independent mode, an internal instruction can be executed by jumping to the proper internal ROM address. Chapter 8 of this manual shows the correct syntax for the JSR (jump to subroutine) and CJSR (conditional jump to subroutine) instructions.

7.1 Internal Instructions Overview

The TMS34082 FPU performs a wide range of internal arithmetic and logical operations, as well as complex operations (flagged †), summarized below. Complex instructions are multicycle routines stored in the internal program ROM. These form a powerful set of primitives for graphics operations.

One Operand Operations

Absolute Value
Square Root
Reciprocal†
1s Complement
2s Complement

Conversions

Integer to Single-Precision
Integer to Double-Precision
Single- to Double-Precision
Single-Precision to Integer
Double-Precision to Integer
Double- to Single-Precision

Two Operand Operations

Add
Subtract
Compare
Multiply
Divide

Matrix Operations

4×4, 4×4 Multiply†
1×4, 4×4 Multiply†
3×3, 3×3 Multiply†
1×3, 3×3 Multiply†

Graphics Operations

Backface Testing†
Polygon Clipping†
2-D Linear Interpolation†
2-D Window Compare†
2-Plane Clipping (X, Y, X)†
2-D Cubic Spline†
Polygon Elimination†
Viewport Scaling and Conversion†
3-D Linear Interpolation†
3-D Volume Compare†
2-Plane Color Clipping (R, B, G, I)†
3-D Cubic Spline†

Image Processing

3×3 Convolution†

Chained Operations

Polynomial Expansion†
1-D Min/Max†
Multiply/Accumulate†
2-D Min/Max†

Vector Operations

Add†
Subtract†
Magnitude†
Scaling†
Dot Product†
Cross Product†
Normalization†
Reflection†

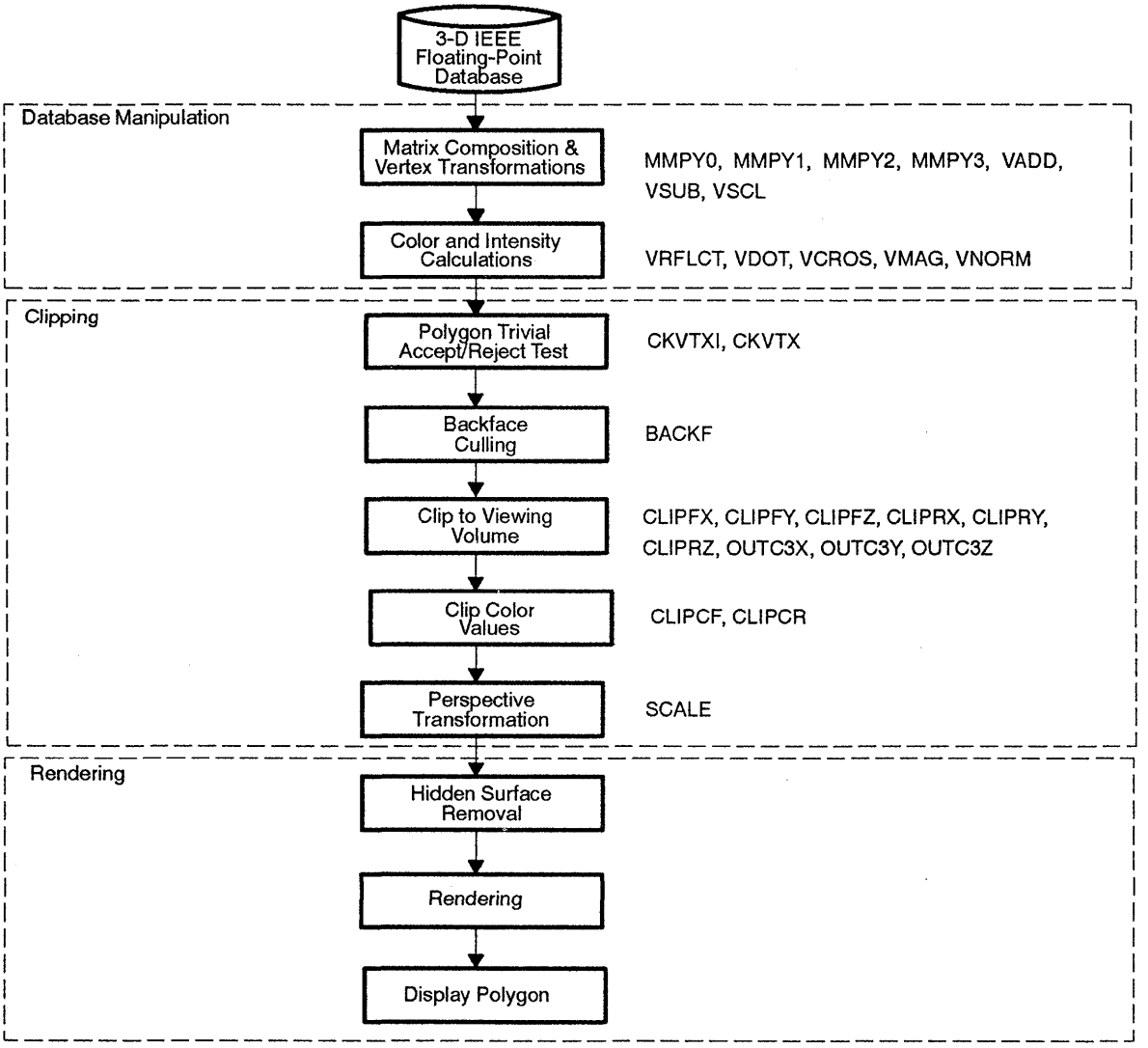
† Indicates complex instructions

The internal routines can be used in either coprocessor or host-independent mode. In coprocessor mode, the internal routines are invoked by TMS34020 instructions to its coprocessor(s). When the TMS34082 is used as a stand-alone processor, the internal microprograms can be called as subroutines by the externally stored code.

7.2 Complex Graphics Instructions

The internal complex instructions may be combined to form a 3-D graphics pipeline. A typical 3-D graphics pipeline includes three major operations on the input object database. The object database is first manipulated to generate normal vectors, and then transformed. The color and intensity values are also calculated. The second step involves the clipping of the objects to the viewing volume. Finally, the objects are displayed according to the rendering style selected. Figure 7-2 shows a typical 3-D graphics pipeline using the complex instructions.

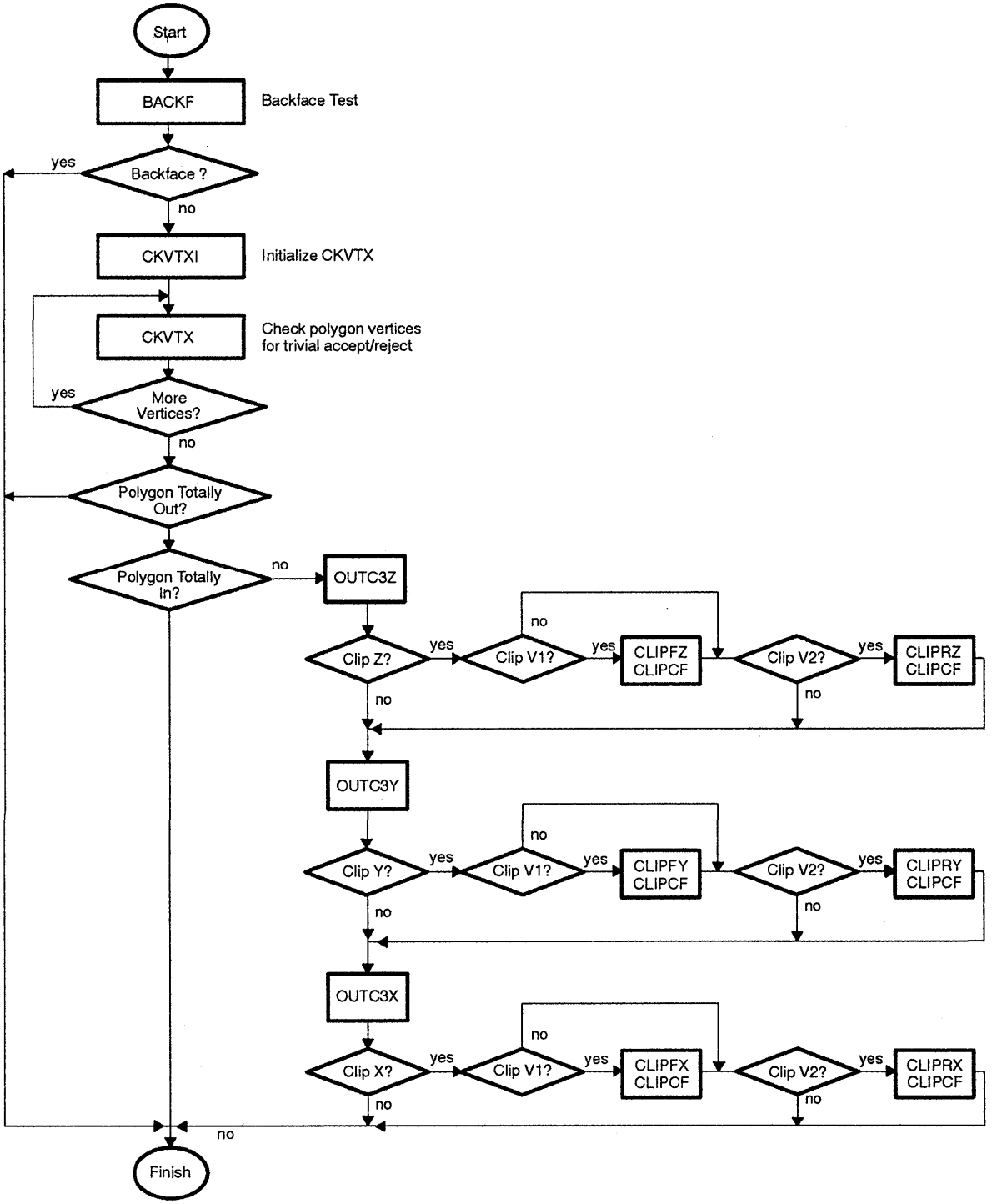
Figure 7–2. 3-D Graphics Pipeline Using TMS34082 Complex Instructions



The complex instructions used in the polygon clipping mechanism can be organized into three functional groups. The first set consists of a single test (BACKF) to determine whether the polygon is forward or backward facing. The second set of instructions (CKVTXI, CKVTX) performs a test to trivially accept or reject a polygon as being visible by checking the vertex coordinates against the viewing volume. The third set of instructions (OUTC3X, OUTC3Y, OUTC3Z, CLIPFX, CLIPFY, CLIPFZ, CLIPRX, CLIPRY, CLIPRZ, CLIPCF, CLIPCR) determines whether the polygon edge crosses the viewing volume boundary and generates the new vertices and color values for the clipped polygon.

The complex instructions are implemented to make efficient use of the TMS34082 Registers and internal status is maintained throughout the clipping mechanism, thus allowing successive polygon edges to be clipped without repeated loading of vertex information. Figure 7-3 details the clipping portion of the pipeline.

Figure 7–3. 3-D Polygon Clipping Flow Chart



7.3 Internal Routine Addresses and Cycle Counts

External programs can call internal routines by executing a jump to subroutine with bit 16 (internal code select) set high and the address of the internal routine as the jump address. Internal routine addresses are given in Table 7–1.

The following table lists internal routines, their addresses, and the number of machine states required to complete the routine. The number in parenthesis after the machine states is the number of cycles before the next operation may begin. For example, it takes five clock cycles to complete an integer CPW (compare point to window) instruction where the status and results are valid; it would take 4 cycles after the CPW began executing before another operation to begin. In coprocessor mode, a machine state is half an LCLK1 period. Therefore, the number of LCLK1 cycles required is the number of machine states divided by 2. In host-independent mode, a machine state is one CLK period.

These cycle counts are for mode 0 instructions only (no data transfers) *after* the instruction reaches the TMS34082. Only mode 0 instructions may be used in host-independent mode. In coprocessor mode, the time required to execute mode 1 and mode 2 instructions is the same as the related mode 1 instruction *after* both instruction and data have reached the TMS34082. The TMS34020 takes one LCLK1 cycle to output a mode 0 instruction and two (one-operand) or three (two-operand) LCKL1 cycles for a mode 1 instruction. A mode 2 instruction requires three TMS34020 LCLK1 cycles, plus one cycle for each memory transfer.

Table 7–1. Internal ROM Routines (for Mode 0 FPU Operations)

Hex Address	Assembler Opcode	Description	Precision	Machine States
000	ADD	Sum of ra and rb	integer	2(1)
001	SUB	Subtract rb from ra	integer	2(1)
002	CMP	Set status bits on result of ra minus rb	integer	2(1)
003	SUB	Subtract ra from rb	integer	2(1)
004		reserved		
005		reserved		
006	MOVE†	Load n FPU registers from TMS34020 GSP or its memory	integer	(see Note)
007	MOVE†	Save n FPU registers from TMS34020 GSP or its memory	integer	(see Note)
008	MPYS	Multiply ra and rb	integer	2(1)
009	DIVS	Divide ra by rb	integer	16(15)
00A	INV	Divide 1 by rb	integer	16(15)
00B		reserved		
00C		reserved		
00D	MOVE	Move ra to rd, multiple, for n registers	integer	(see Note)
00E	MOVE	Move rb to rd, multiple, for n registers	integer	(see Note)
00F		reserved		
010	CPW	Compare point to window	integer	5(4)
011	CPV	Compare point to volume	integer	7(6)
012	BACKF	Test polygon for facing direction (backface test)	integer	16(15)
013	INMNMX	Setup FPU registers for MNMX1 or MNMX2 instruction		2(1)
014	LINTX	Linear interpolation, X plane	integer	26(25)
015	CLIPFX	Clip a line to an X plane pair boundary (start w/point 1)	integer	34(33)
016	CLIPRX	Clip a line to an X plane pair boundary (start w/point 2)	integer	34(33)
017	CLIPC	Clip color values to a plane pair boundary (start w/point 1)	integer	27(26)
018	SCALE	Scale and convert coordinates for viewport	integer	56(55)
019	MTRAN	Transpose a matrix	integer	13(12)
01A	CKVTX	Compare polygon vertex to a clipping volume	integer	6(5)
01B	CONV	3x3 convolution	integer	32(31)
01C	CLIPCR	Clip color values to a plane pair boundary (start w/point 2)	integer	27(26)
01D	OUTC3X	Compare a line to a clipping value, X plane	integer	5(4)
01E	CSPLN	Calculate cubic spline	integer	22(21)
01F		reserved		
020	MOVE	Copy ra to rd	integer	2(1)
021	NOT	Place 1's complement of ra in rd	integer	2(1)
022	ABS	Place absolute value of ra in rd	integer	2(1)
023	NEG	Place negated value of ra in rd	integer	2(1)
024		reserved		
025		reserved		

† Cannot be used in host-independent mode.

NOTE: Number of machine states varies, depending on the number of words moved.

Table 7–1. Internal ROM Routines (for Mode 0 FPU Operations) (Continued)

Hex Address	Assembler Opcode	Description	Precision	Machine States
026		reserved		
027	VSCL†	Multiply vector by a scaling factor	integer	4(3)
028	SQAR	Place (ra * ra) in rd	integer	4(3)
029	SQRT	Extract square root of ra	integer	20(19)
02A	SQRTA	Extract square root of absolute value of ra	integer	20(19)
02B	ABORT	Stop execution of any FPU instruction	integer	2(1)
02C	CKVTX1	Initialize check vertex instruction		2(1)
02D	CHECK	Check for previous instruction completion		2(1)
02E	MOVTSRAM†	Move data from system memory to external memory		
02F	MOVFSRAM†	Move data to system memory from external memory		
030	POLY†	Polynomial expansion	integer	4(3)
031	MAC†	Multiply and accumulate	integer	4(3)
032	MNMX1†	Determine 1-D minimum and maximum of a series	integer	3(2)
033	MNMX2†	Determine 2-D minimum and maximum of a series of pairs	integer	5(4)
034	MMPY0	Multiply matrix elements 3-0 by vector element 0	integer	6(5)
035	MMPY1	Multiply matrix elements 7-4 by vector element 1	integer	10(9)
036	MMPY2	Multiply matrix elements 11-8 by vector element 2	integer	12(11)
037	MMPY3	Multiply matrix elements 15-12 by vector element 3	integer	12(11)
038	MADD	Add matrix elements 15-12 to vector integer	integer	9(8)
039	VADD	Add two vectors	integer	4(3)
03A	VSUB	Subtract a vector from a vector	integer	4(3)
03B	VDOT	Compute scalar dot product of two vectors	integer	7(6)
03C	VCROS	Compute cross product of two vectors	integer	9(8)
03D	VMAG	Determine the magnitude of a vector	integer	30(29)
03E	VNORM	Normalize a vector to unit magnitude	integer	50(49)
03F	VRFLCT	Given normal and incident vectors, find the reflection	integer	16(15)
080	ADDF	Sum of ra and rb	single	2(1)
081	SUBF	Subtract rb from ra	single	2(1)
082	CMPF	Set status bits on result of ra minus rb	single	2(1)
083	SUBF	Subtract ra from rb	single	2(1)
084	ADDA	Absolute value of sum of ra and rb	single	2(1)
085	SUBA	Absolute value of (ra minus rb)	single	2(1)
086	MOVF	Load n FPU registers from TMS34020 GSP or its memory	single	
087	MOVF	Save n FPU registers from TMS34020 GSP or its memory	single	
088	MPYF	Multiply ra and rb	single	2(1)
089	DIVF	Divide ra by rb	single	7(6)
08A	INVF	Divide 1 by rb	single	7(6)
08B	ASUBA	Absolute value of ra minus absolute value of rb	single	2(1)
08C		reserved		

† Cannot be used in host-independent mode.

NOTE: Number of machine states varies, depending on the number of words moved.

Table 7–1. Internal ROM Routines (for Mode 0 FPU Operations) (Continued)

Hex Address	Assembler Opcode	Description	Precision	Machine States
08D	MOVEF†	Move ra to rd, multiple, for n registers	single	(see Note)
08E	MOVEF†	Move ra to rd, multiple, for n registers	single	(see Note)
08F		reserved		
090	CPWF	Compare point to window	single	5(4)
091	CPVF	Compare point to volume	single	7(6)
092	BACKFF	Test polygon for facing direction (backface test)	single	16(15)
093	INMNMXF	Setup FPU registers for MNMX1 and MNMX2	single	2(1)
094	LINTXF	Linear interpolation, X plane	single	17(16)
095	CLIPFXF	Clip a line to an X plane pair boundary (start w/point 1)	single	25(24)
096	CLIPRXF	Clip a line to an X plane pair boundary (start w/point 2)	single	25(24)
097	CLIPCF	Clip color values to a plane pair boundary (start w/point 1)	single	18(17)
098	SCALEF	Scale and convert coordinates for viewport	single	21(20)
099	MTRANF	Transpose a matrix	single	13(12)
09A	CKVTXF	Compare polygon vertex to a clipping volume	single	6(5)
09B	CONVF	3x3 convolution	single	17(16)
09C	CLIPCRF	Clip color values to a plane pair boundary (start w/point2)	single	18(17)
09D	OUTC3XF	Compare a line to a clipping value, X plane	single	5(4)
09E	CSPLNF	Calculate cubic spline	single	22(21)
09F		reserved		
0A0	MOVE	copy ra to rd	single	2(1)
0A1	NOT	Place 1's complement of ra in rd	single	2(1)
0A2	ABS	Place absolute value of ra in rd	single	2(1)
0A3	NEG	Place negated value of ra in rd	single	2(1)
0A4	CVFD	Convert single-precision to double-precision	single	2(1)
0A5	CVFI	Convert single-precision to integer	single	2(1)
0A6	CVIF	Convert integer to single-precision	single	2(1)
0A7	VSCLF†	Multiply vector by a scaling factor	single	4(3)
0A8	SQARF	Place (ra * ra) in rd	single	4(3)
0A9	SQRTF	Extract square root of ra	single	10(9)
0AA	SQRTAF	Extract square root of absolute value of ra	single	10(9)
0AB	ABORT	Stop execution of any FPU instruction		2(1)
0AC	CKVTX1	Initialize check vertex instruction		2(1)
0AD	CHECK	Check for previous instruction completion		2(1)
0AE	MOVTSRAM†	Move data from system memory to external memory		
0AF	MOVFSRAM†	Move data to system memory from external memory		
0B0	POLYF†	Polynomial expansion	single	4(3)
0B1	MACF†	Multiply and accumulate	single	4(3)
0B2	MNMX1F†	Determine 1-D minimum and maximum of a series	single	3(2)
0B3	MNMX2F†	Determine 2-D minimum and maximum of a series of pairs	single	5(4)

† Cannot be used in host-independent mode.

NOTE: Number of machine states varies, depending on the number of words moved.

Table 7–1. Internal ROM Routines (for Mode 0 FPU Operations) (Continued)

Hex Address	Assembler Opcode	Description	Precision	Machine States
0B4	MMPY0F	Multiply matrix elements 3-0 by vector element 0	single	6(5)
0B5	MMPY1F	Multiply matrix elements 7-4 by vector element 1	single	10(9)
0B6	MMPY2F	Multiply matrix elements 11-8 by vector element 2	single	12(11)
0B7	MMPY3F	Multiply matrix elements 15-12 by vector element 3	single	12(11)
0B8	MADDF	Add matrix elements 15-12 to vector	single	9(8)
0B9	VADDF	Add two vectors	single	4(3)
0BA	VSUBF	Subtract a vector from a vector	single	4(3)
0BB	VDOTF	Compute scalar dot product of two vectors	single	7(6)
0BC	VCROSF	Compute cross product of two vectors	single	9(8)
0BD	VMAGF	Determine the magnitude of a vector	single	20(19)
0BE	VNORMF	Normalize a vector to unit magnitude	single	31(30)
0BF	VRFLCTF	Given normal and incident vectors, find the reflection	single	16(15)
0C0	ADDD	Sum of ra and rb	double	2(1)
0C1	SUBD	Subtract rb from ra	double	2(1)
0C2	CMPD	Set status bits on result of ra minus rb	double	2(1)
0C3	SUBD	Subtract ra from rb	double	2(1)
0C4	ADDA	Absolute value of sum of ra and rb	double	2(1)
0C5	SUBA	Absolute value of (ra minus rb)	double	2(1)
0C6	MOVD†	Load n FPU registers from TMS34020 GSP or its memory	double	(see Note)
0C7	MOVD†	Save n FPU registers from TMS34020 GSP or its memory	double	(see Note)
0C8	MPYD	Multiply ra and rb	double	3(2)
0C9	DIVD	Divide ra by rb	double	13(12)
0CA	INVD	Divide 1 by rb	double	13(12)
0CB	ASUBA	Absolute value of ra minus absolute value of rb	double	2(1)
0CC		reserved		
0CD	MOVD†	Move ra to rd, multiple, for n registers	double	(see Note)
0CE	MOVD†	Move rb to rd, multiple, for n registers	double	(see Note)
0CF		reserved		
0D0	CPWD	Compare point to window	double	5(4)
0D1	CPVD	Compare point to volume	double	7(6)
0D2	BACKFD	Test polygon for facing direction (backface test)	double	25(24)
0D3	INMNMXD	Setup FPU registers for MNMX1 and MNMX2	double	2(1)
0D4	LINTXD	Linear interpolation, X plane	double	26(25)
0D5	CLIPFXD	Clip a line to an X plane pair boundary (start w/point 1)	double	35(34)
0D6	CLIPRXD	Clip a line to an X plane pair boundary (start w/point 2)	double	35(34)
0D7	CLIPCD	Clip color values to a plane pair boundary (start w/point 1)	double	28(27)
0D8	SCALED	Scale and convert coordinates for viewport	double	33(32)
0D9	MTRAND	Transpose a matrix	double	13(12)
0DA	CKVTXD	Compare polygon vertex to a clipping volume	double	6(5)

† Cannot be used in host-independent mode.

NOTE: Number of machine states varies, depending on the number of words moved.

Table 7–1. Internal ROM Routines (for Mode 0 FPU Operations) (Continued)

Hex Address	Assembler Opcode	Description	Precision	Machine States
0DB	CONVD	3x3 convolution	double	29(30)
0DC	CLIPCRD	Clip color values to a plane pair boundary (start w/point 1)	double	31(30)
0DD	OUTC3XD	Compare a line to a clipping value, X plane	double	5(4)
0DE	CSPLND	Calculate cubic spline	double	31(30)
0DF		reserved		
0E0	MOVE	Copy ra to rd	double	2(1)
0E1	NOT	Place 1's complement of ra in rd	double	2(1)
0E2	ABS	Place absolute value of ra in rd	double	2(1)
0E3	NEG	Place negated value of ra in rd	double	2(1)
0E4	CVDF	Convert double-precision to single-precision	double	2(1)
0E5	CVDI	Convert double-precision to integer	double	2(1)
0E6	CVID	Convert integer to double-precision	double	2(1)
0E7	VSCLD†	Multiply vector by a scaling factor	double	7(6)
0E8	SQARD	Place (ra * ra) in rd	double	5(4)
0E9	SQRTD	Extract square root of ra	double	16(15)
0EA	SQRTAD	Extract square root of absolute value of ra	double	16(15)
0EB	ABORT	Stop execution of any FPU instruction		2(1)
0EC	CKVTX1	Initialize check vertex instruction		2(1)
0ED	CHECK	Check for previous instruction completion		2(1)
0EE		reserved		
0EF		reserved		
0F0	POLYD†	Polynomial expansion	double	5(4)
0F1	MACD†	Multiply and accumulate	double	5(4)
0F2	MNMX1D†	Determine 1-D minimum and maximum of a series	double	3(2)
0F3	MNMX2D†	Determine 2-D minimum and maximum of a series of pairs	double	5(4)
0F4	MMPY0D	Multiply matrix elements 3-0 by vector element 0	double	11(10)
0F5	MMPY1D	Multiply matrix elements 7-4 by vector element 1	double	14(13)
0F6	MMPY2D	Multiply matrix elements 11-8 by vector element 2	double	16(15)
0F7	MMPY3D	Multiply matrix elements 15-12 by vector element 3	double	16(15)
0F8	MADDD	Add matrix elements 15-12 to vector	double	9(8)
0F9	VADDD	Add two vectors	double	4(3)
0FA	VSUBD	Subtract a vector from a vector	double	4(3)
0FB	VDOTD	Compute scalar dot product of two vectors	double	10(9)
0FC	VCROSD	Compute cross product of two vectors	double	15(14)
0FD	VMAGD	Determine the magnitude of a vector	double	29(28)
0FE	VNORMD	Normalize a vector to unit magnitude	double	49(48)
0FF	VRFLCTD	Given normal and incident vectors, find the reflection	double	23(22)
114	LINTY	Linear interpolation, Y plane	integer	26(25)

† Cannot be used in host-independent mode.

NOTE: Number of machine states varies, depending on the number of words moved.

Table 7–1. Internal ROM Routines (for Mode 0 FPU Operations) (Continued)

Hex Address	Assembler Opcode	Description	Precision	Machine States
115	CLIPFY	Clip a line to an Y plane pair boundary (start w/point 1)	integer	34(33)
116	CLIPRY	Clip a line to an Y plane pair boundary (start w/point 2)	integer	34(33)
11D	OUTC3Y	Compare a line to a clipping value, Y plane	integer	5(4)
194	LINTYF	Linear interpolation, Y plane	single	17(16)
195	CLIPFYF	Clip a line to an Y plane pair boundary (start w/point 1)	single	25(24)
196	CLIPRYF	Clip a line to an Y plane pair boundary (start w/point 2)	single	25(24)
19D	OUTC3YF	Compare a line to a clipping value, Y plane	single	5(4)
1D4	LINTYD	Linear interpolation, Y plane	double	17(16)
1D5	CLIPFYD	Clip a line to an Y plane pair boundary (start w/point 1)	double	25(24)
1D6	CLIPRYD	Clip a line to an Y plane pair boundary (start w/point 2)	double	25(24)
1DD	OUTC3YD	Compare a line to a clipping value, Y plane	double	5(4)
214	LINTZ	Linear interpolation, Z plane	integer	26(25)
215	CLIPFZ	Clip a line to an Z plane pair boundary (start w/point 1)	integer	34(33)
216	CLIPRZ	Clip a line to an Z plane pair boundary (start w/point 2)	integer	34(33)
21D	OUTC3Z	Compare a line to a clipping value, Z plane	integer	5(4)
294	LINTZF	Linear interpolation, Z plane	single	17(16)
295	CLIPFZF	Clip a line to an Z plane pair boundary (start w/point 1)	single	25(24)
296	CLIPRZF	Clip a line to an Z plane pair boundary (start w/point 2)	single	25(24)
29D	OUTC3ZF	Compare a line to a clipping value, Z plane	single	5(4)
2D4	LINTZD	Linear interpolation, Z plane	double	17(16)
2D5	CLIPFZD	Clip a line to an Z plane pair boundary (start w/point 1)	double	25(24)
2D6	CLIPRZD	Clip a line to an Z plane pair boundary (start w/point 2)	double	25(24)
2DD	OUTC3ZD	Compare a line to a clipping value, Z plane	double	5(4)

† Cannot be used in host-independent mode.

NOTE: Number of machine states varies, depending on the number of words moved.

7.4 Coprocessor Mode Internal Instruction Format

The format of the TMS34082 instruction in coprocessor mode is shown below. The instruction is issued by the TMS34020 via the LAD bus.

31 28	24	20	15	13	8	7	6	5	0
ID	ra	rb	rd	md	fpuop	type	size	0	I	00000

7.4.1 Coprocessor ID Field

The 3-bit ID field identifies which coprocessor the instruction is intended for. This coprocessor ID corresponds to the settings of the CID2-0 pins. To broadcast an instruction to all coprocessors, the ID field is set to 4. The TMS34020 documentation recommends the coprocessor ID assignments shown below. However, both the TMS34020 and TMS34082 support using up to seven TMS34082s per TMS34020.

The assembler defaults to an ID of 000_2. To define another ID as the current ID, use the coprocessor assembler directive.

Table 7–2. Coprocessor IDs

ID	Coprocessor	ID	Coprocessor
000	FPU 0	100	FPU broadcast
001	FPU 1	101	Reserved (or FPU 4)
010	FPU 2	110	Reserved (or FPU 5)
011	FPU 3	111	User defined (or FPU 6)

7.4.2 Register Field

The ra, rb, and rd fields are for the two sources (A and B) and destination within the FPU. For most two-operand instructions, one operand must come from each register file. Register addresses were listed in Table 4–3. For the ra and rb fields, only the four least significant bits of the register address are used. Some multi-operand instructions redefine the ra, rb, and rd field.

Valid values for registers operands are:

ra: RA0-RA9 (also, C, and CT following rules below)

rb: RB0-RB9 (also, C, and CT following rules below)

rd: RA0-RA9 RB0-RB9, C, and CT

NOTE: Although the TMS34020 assembler only allows the above registers as destinations, the TMS34082 will accept any register address as a destination.

The following is a list of rules for using the C and CT registers as operands:

1) *Do not* use C or CT as source operands in any mode 1 or 2 ("Load and.") instructions.

2) *Do not* use C or CT in any MOVE, MOVD, or MOVF instructions. If it is necessary to move a value to or from the C or CT register, use the PASS, PASSF, or PASSD instruction (depending on the type of number in C or CT). C and CT *are* legal operands for the PASSx instructions. However, the type of number in C or CT *must* match the type (integer single-, or double-precision) of the PASSx instruction.

3) *Do not* use C or CT as source operands for integer divide (DIVS), integer inverse (INV), convert integer to single-precision (CVIF) or convert integer to double-precision (CVID) instructions.

4) For instructions requiring two source operands, C *or* CT can be used as both operands, but cannot be used together in the same instruction.

7.4.3 Addressing Mode Field

Four addressing modes are defined for the TMS34082. The md field indicates the addressing mode. Each addressing mode corresponds to one or two general-purpose TMS34020 coprocessor commands. Specific TMS34082 instructions are created by specifying the fields of the internal instruction as shown above.

Table 7–3. Addressing Modes

Mode	md Field	Operation	General TMS34020 Coprocessor Command
0	00	FPU internal operations with no jumps or external moves	CEXEC
1	01	Transfer instruction and data to/from TMS34020 registers	CMOVGC, CMOVCG
2	10	Transfer instruction and data to/from memory (controlled by TMS34020) on LAD bus	CMOVMC, CMOVCM
3	11	Jump to external instructions in TMS34082 external memory	CEXEC

7.4.4 FPU Operation Field

The fpuop field tells the TMS34082 which operation (such as addition or subtraction) or complex instructions (such as clipping) to perform. Sometimes the rb field is also used to specify the operation. A list of instructions and their associated fpuop field is given in the TMS34082A Data Sheet (Appendix B).

7.5 Type, Size, and I Fields

The type and size bits identify the type of operand, as shown in Table 7–4. The I bit is used to indicate to the coprocessor that this is a 'reissue' of a coprocessor instruction due to a bus interruption. The least significant four bits are the bus status bits, which will all be zero to indicate a coprocessor cycle.

Table 7–4. Operand Types

Type	Size	Operand Type
0	0	32-bit Integer
0	1	Reserved
1	0	Single-precision floating-point (32-bit)
1	1	Double-precision floating-point (64-bit)

7.6 Internal Instruction Opcodes

Details of each internal routine follow. The routines are listed alphabetically by their TMS34020 assembler opcodes.

Sets of related instructions (same operation, different operand types) are listed together. Sets begin on a new page and may contain the following information.

Syntax: Shows you how to enter an instruction. Each valid operand type is listed, along with its syntax. Bold text should be entered as shown. Italic text represents a symbol that tells what type of information should be entered. These symbols are further described in the *operand* section.

Execution: Illustrates the effects of execution on TMS34020 and TMS34082 registers and memory. The shaded portion represents steps that are executed for double-precision instructions only.

TMS34020 Instruction Words: Shows the object code generated for an instruction. This is the instruction *to* the TMS34020. In this instruction, *transfers* is the number of 32-bit words moved across the LAD bus. *Transfers* will generally be the number of operands for an integer or single-precision instruction. For a double-precision instruction, *transfers* is twice the number of operands.

TMS34082 Instruction Word: Shows the command generated by the TMS34020 that is sent (via the LAD bus) to the TMS34082. In this word, ***t*** and ***s*** are used to specify the type and size bits, respectively.

Operands: Explains the symbols used in the syntax section. Implied operands are values that must be in the appropriate register(s) before the instruction is executed. The following symbols are used as operands:

Rs, Rs_1, Rs_2	TMS34020 source register(s)
Rd, Rd_1, Rd_2	TMs34020 destination register(s)
CRs, CRs_1, CRs_2	TMS34082 source register. Must be from the RA or RB register files, C, or CT. See the restrictions on the use of C and CT given in subsection 7.3.2.
CRd	Unless otherwise noted, C or CT may be substituted for RA or RB registers in any instruction which does not require data transfers to/from the TMS34020 or memory.

Description: Discusses the purpose of the instruction and any other general information related to it.

Algorithm: Illustrates the operations performed in a multicycle, complex instruction. The shaded portion represents steps that are executed for double-precision instructions only.

Temporary Storage: Lists registers that are used in complex instructions. Any value stored in these registers prior to instruction execution will be lost.

Outputs: Lists the registers that contain the result(s) of the complex instruction.

Instruction Type: Shows the type of TMS34020 coprocessor instruction. The TMS34020 has several general-purpose coprocessor instructions that are used to create the specific TMS34082 instructions.

Examples: Illustrates the correct syntax for a specific instruction and describes the effects of the instruction on memory and registers using various sets of data.

Not all topics are included for each instruction set. Each set contains at least the Syntax, Execution or Algorithm, both Instruction Words, and the Description sections.

Syntax **ABORT**

Execution Halts TMS34082

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	0	0
ID			0	0	0	0	1	0	1	1	0	0	0	0	0

Instruction to '34082

31	29														0
ID	0	0	0	0	1	0	1	1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0

Description This instruction will cancel all activity within the TMS34082, returning the FPU to an inactive state. Any time this instruction is present on a coprocessor cycle with a valid coprocessor ID, the addressed TMS34082 will ABORT all internal processing activity immediately. Block moves will be aborted before completion of the last move.

Instruction Type CEXEC, short

Syntax

Type	Syntax
Integer	**ABS** *CRs, CRd*
Double-Precision	**ABSD** *CRs, CRd*
Single-Precision	**ABSF** *CRs, CRd*

Execution

$|CRs| \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	type	size
ID			CRs				0	0	1	0	CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs	0010	CRd	0001 111t s000 0000

Operands

CRs TMS34082 RA source register containing the operand

CRd TMS34082 destination register

Description

ABSx takes the absolute value of the contents of CRs and stores the result in CRd.

The source register, CRs, must be in the RA register file.

Instruction Type

CEXEC, short

Example

```
ABS RA6, RB7
```

This example takes the absolute value of the integer contents of RA6 and stores the integer result in RB7.

Syntax

Type	Syntax
Integer	**ABS** *Rs_1, CRs, CRd*
Double-Precision	**ABSD** *Rs_1, Rs_2, CRs, CRd*
Single-Precision	**ABSF** *Rs_1, CRs, CRd*

Execution

$Rs_1 \rightarrow CRs$

$Rs_2 \rightarrow CRs$

$|CRs| \rightarrow CRd$

'34020 Instruction Words

Integer or Single-Precision:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs_1			
0	1	0	1	1	1	1	type	0	0	0	0	0	0	0	0
ID			CRs				0	0	1	0	CRd				

Double-Precision:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	1	1	1	1	1	1	0	0	R	Rs_2			
ID			CRs				0	0	1	0	CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs	0010	CRd	0101 111t s000 0000

Operands

Rs_1 TMS34020 source register for the integer or single-precision operand to the TMS34082 (or half of the 64-bit value for double-precision operands)

Rs_2 TMS34020 source register for the remaining half of the 64-bit double-precision floating-point value to TMS34082.

CRs TMS34082 RA register to contain the 32-bit integer operand

CRd TMS34082 destination register

Description

ABSx loads the contents of Rs_1 (and Rs_2 for double-precision values) into CRs, takes the absolute value of the contents of CRs, and stores the result in CRd.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVGC, one or two registers

Example

```
ABSF A5, RA6, RB7
```

This example loads the single-precision contents of TMS34020 register A5 into TMS34082 register RA6, takes the absolute value of the contents of RA6, and stores the single-precision result in RB7.

Syntax

Type	Syntax
Integer	**ABS** **Rs+, CRs, CRd*
Double-Precision	**ABSD** **Rs+, CRs, CRd*
Single-Precision	**ABSF** **Rs+, CRs, CRd*

Execution

*Rs → CRs
Rs + 32 → Rs
*Rs → CRs
Rs + 32 → Rs
|CRs| → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	0	transfers	
1	0	0	1	1	1	1	type	size	0	0	R	Rs			
ID			CRs				0	0	1	0	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0010		CRd		1001 111t s000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 RA register to contain the operand

CRd TMS34082 destination register

Description

ABSx loads the contents of memory pointed to by Rs into CRs, takes the absolute value of the contents of CRs, and stores the result in CRd. After each load from memory, Rs is incremented by 32.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
ABSD *A5+, RA6, RB7
```

This example loads the double-precision floating-point contents of memory at the address given by TMS34020 register A5 into TMS34082 register RA6, takes the absolute value of the contents of RA6, and stores the result in RB7.

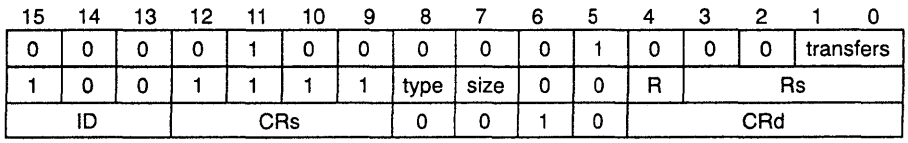
Syntax

Type	Syntax
Integer	**ABS** –*****Rs, CRs, CRd*
Double-Precision	**ABSD** –*****Rs, CRs, CRd*
Single-Precision	**ABSF** –*****Rs, CRs, CRd*

Execution

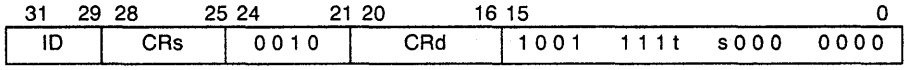
Rs – 32 → Rs
*Rs → CRs
Rs – 32 → Rs
*Rs → CRs
|CRs| → CRd

'34020 Instruction Words



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	transfers	
1	0	0	1	1	1	1	type	size	0	0	R	Rs			
ID			CRs				0	0	1	0	CRd				

Instruction to '34082



31 29	28 25	24 21	20 16	15 0
ID	CRs	0010	CRd	1001 111t s000 0000

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 RA register to contain the operand

CRd TMS34082 destination register

Description

ABSx loads the contents of memory pointed to by Rs into CRs, takes the absolute value of the contents of CRs, and stores the result in CRd. Before each load from memory, Rs is decremented by 32.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVMC, predecrement, constant count

Example

```
ABS –*A5, RA6, RB7
```

This example loads the integer contents of memory at the address given by TMS34020 register A5 minus 32 into TMS34082 register RA6, takes the absolute value of the contents of RA6, and stores the integer result in RB7.

Syntax

Type	Syntax
Integer	**ADD** *CRs$_1$, CRs$_2$, CRd*
Double-Precision	**ADDD** *CRs$_1$, CRs$_2$, CRd*
Single-Precision	**ADDF** *CRs$_1$, CRs$_2$, CRd*

Execution

$CRs_1 + CRs_2 \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	0	0	type	size
ID			CRs$_1$				CRs$_2$				CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs$_1$	CRs$_2$	CRd	0000 000t s000 0000

Operands

CRs$_1$ TMS34082 register containing the first operand

CRs$_2$ TMS34082 register containing the second operand

CRd TMS34082 destination register

Description

ADDx adds the contents of CRs$_1$ and CRs$_2$ and stores the result in CRd.

The two source registers, CRs$_1$ and CRs$_2$, must be in opposite register files.

Instruction Type

CEXEC, short

Example

```
ADDD RA5, RB6, RB7
```

This example adds the double-precision floating-point contents of RA5 and RB6 and stores the result in RB7.

Syntax

Type	Syntax
Integer	**ADD** *Rs_1, Rs_2, CRs_1, CRs_2, CRd*
Single-Precision	**ADDF** *Rs_1, Rs_2, CRs_1, CRs_2, CRd*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
$CRs_1 + CRs_2 \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	0	0	0	type	0	0	0	R	Rs_2			
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs_1		CRs_2		CRd		0100 000t 0000 0000	

Operands

Rs_1 TMS34020 source register for the first value to TMS34082

Rs_2 TMS34020 source register for the second value to TMS34082

CRs_1 TMS34082 register to contain the first operand

CRs_2 TMS34082 register to contain the second operand

CRd TMS34082 destination register

Description

ADDx loads the contents of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, adds the contents of CRs_1 and CRs_2, and stores the result in CRd.

The two TMS34082 source registers, CRs_1 and CRs_2, must be in opposite register files.

The double-precision floating-point form of this instruction is not supported.

Instruction Type

CMOVGC, two registers

Example

```
ADDF A5, A6, RA5, RB6, RB7
```

This example loads TMS34020 registers A5 and A6 into TMS34082 registers RA5 and RB6 respectively, adds the single-precision floating-point values from RA5 and RB6, and stores the result in RA7.

Syntax

Type	Syntax
Integer	**ADD** **Rs+, CRs$_1$, CRs$_2$, CRd*
Double-Precision	**ADDD** **Rs+, CRs$_1$, CRs$_2$, CRd*
Single-Precision	**ADDF** **Rs+, CRs$_1$, CRs$_2$, CRd*

Execution

*Rs → CRs$_1$
Rs + 32 → Rs
*Rs → CRs$_1$
Rs + 32 → Rs
*Rs → CRs$_2$
Rs + 32 → Rs
*Rs → CRs$_2$
Rs + 32 → Rs
CRs$_1$ + CRs$_2$ → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	transfers		
1	0	0	0	0	0	0	type	size	0	0	R	Rs			
ID			CRs$_1$				CRs$_2$				CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs$_1$	CRs$_2$	CRd	1000 000t s000 0000

Operands

Rs TMS34020 register containing the memory address

CRs$_1$ TMS34082 register to contain the first operand

CRs$_2$ TMS34082 register to contain the second operand

CRd TMS34082 destination register

Description

ADDx loads the contents of memory pointed to by Rs into CRs$_1$ and CRs$_2$, adds the contents of CRs$_1$ and CRs$_2$, and stores the result in CRd. After each load from memory, Rs is incremented by 32.

The two TMS34082 source registers, CRs$_1$ and CRs$_2$, must be in opposite register files.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
ADD *A5+, RA5, RB6, RB7
```

This example loads memory starting at the address given by TMS34020 register A5 into TMS34082 registers RA5 and RB6, adds the integer values from RA5 and RB6, and stores the result in RB7.

Syntax

Type	Syntax
Integer	**ADD** $-*Rs, CRs_1, CRs_2, CRd$
Double-Precision	**ADDD** $-*Rs, CRs_1, CRs_2, CRd$
Single-Precision	**ADDF** $-*Rs, CRs_1, CRs_2, CRd$

Execution

Rs – 32 → Rs
*Rs → CRs_1
Rs – 32 → Rs
*Rs → CRs_1
Rs – 32 → Rs
*Rs → CRs_2
Rs – 32 → Rs
*Rs → CRs_2
CRs_1 + CRs_2 → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	transfers		
1	0	0	0	0	0	0	type	size	0	0	R	Rs			
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs_1		CRs_2		CRd		1000 000t s000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs_1 TMS34082 register to contain the first operand

CRs_2 TMS34082 register to contain the second operand

CRd TMS34082 destination register

Description

ADDx loads the contents of memory pointed to by Rs into CRs_1 and CRs_2, add the contents of CRs_1 and CRs_2, and stores the result in CRd. Before each load from memory, Rs is decremented by 32.

The two TMS34082 source registers, CRs_1 and CRs_2, must be in opposite register files.

Instruction Type

CMOVMC, predecrement, constant count

Example

```
ADD –*A5, RA5, RB6, RB7
```

This example loads memory starting at the address given by TMS34020 register A5 minus 32 into TMS34082 registers RA5 and RB6, adds the integer contents of RA5 and RB6, and stores the result in RB7.

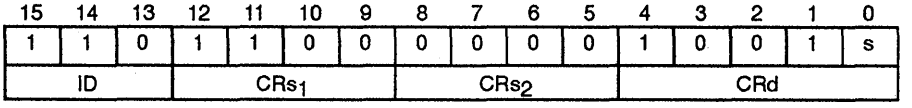
Syntax

Type	Syntax
Double-Precision	**ADDAD** *CRs_1, CRs_2, CRd*
Single-Precision	**ADDAF** *CRs_1, CRs_2, CRd*

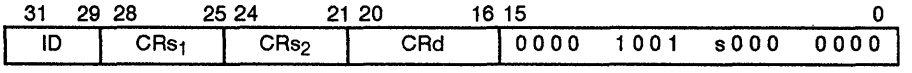
Execution

$|CRs_1 + CRs_2| \rightarrow CRd$

'34020 Instruction Words



Instruction to '34082



Operands

CRs_1 TMS34082 register containing the first operand

CRs_2 TMS34082 register containing the second operand

CRd TMS34082 destination register

Description

ADDAx takes the absolute value of the sum of CRs_1 and CRs_2, and places the result in CRd.

CRs_1 and CRs_2, the two TMS34082 source registers, must be in opposite register files.

The integer form of this instruction is not supported.

Instruction Type

CEXEC, short

Example

```
ADDAF RA3, RB9, RA1
```

This example adds the single-precision floating-point contents of RA3 and RB9, takes the absolute value, and stores the result in RA1.

Syntax

ADDAF *Rs_1, Rs_2, CRs_1, CRs_2, CRd*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
$|CRs_1 + CRs_2| \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	1	0	0	1	0	0	0	R	Rs_2			
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs_1	CRs_2	CRd	0100 1001 0000 0000

Operands

Rs_1 TMS34020 source register for first 32-bit single-precision floating-point value to TMS34082

Rs_2 TMS34020 source register for second 32-bit single-precision floating-point value to TMS34082

CRs_1 TMS34082 register to contain the first single-precision operand

CRs_2 TMS34082 register to contain the second single-precision operand

CRd TMS34082 destination register

Description

ADDAF loads the contents or Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, takes the absolute value of the sum of CRs_1 and CRs_2, and stores the result in CRd.

CRs_1 and CRs_2, the two TMS34082 source registers, must be in opposite register files.

The integer and double-precision floating-point forms of this instruction are not supported.

Instruction Type

CMOVGC, two registers

Example

```
ADDAF A5, A9, RA7, RB9, RB0
```

This example loads the contents of TMS34020 registers A5 and A9 into TMS34082 registers RA7 and RB9 respectively, adds the contents of RA7 and RB9, takes the absolute value, and stores the result in RB0.

Syntax

Type	Syntax
Double-Precision	**ADDAD** **Rs+, CRs$_1$, CRs$_2$, CRd*
Single-Precision	**ADDAF** **Rs+, CRs$_1$, CRs$_2$, CRd*

Execution

*Rs → CRs$_1$
Rs + 32 → Rs
*Rs → CRs$_1$
Rs + 32 → Rs
*Rs → CRs$_2$
Rs + 32 → Rs
*Rs → CRs$_2$
Rs + 32 → Rs
|CRs$_1$ + CRs$_2$| → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	transfers		
1	0	0	0	1	0	0	1	size	0	0	R	Rs			
ID			CRs$_1$				CRs$_2$				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs$_1$		CRs$_2$		CRd		1000 1001 s000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs$_1$ TMS34082 register to contain the first operand

CRs$_2$ TMS34082 register to contain the second operand

CRd TMS34082 destination register

Description

ADDAx loads the contents of memory pointed to by Rs into CRs$_1$ and CRs$_2$, adds the contents of CRs$_1$ and CRs$_2$, takes the absolute value, and stores the result in CRd. After each load from memory, Rs is incremented by 32.

CRs$_1$ and CRs$_2$, the two TMS34082 source registers, must be in opposite register files.

The integer form of this operation is not supported.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
ADDAD *A5+, RA7, RB9, RB0
```

This example loads memory starting at the address given by TMS34020 register A5 into TMS34082 registers RA7 and RB9, adds the double-precision contents of RA7 and RB9, takes the absolute value, and stores the result in RB0.

Syntax

Type	Syntax
Double-Precision	**ADDAD** $-*Rs, CRs_1, CRs_2, CRd$
Single-Precision	**ADDAF** $-*Rs, CRs_1, CRs_2, CRd$

Execution

$Rs - 32 \rightarrow Rs$
$*Rs \rightarrow CRs_1$
$Rs - 32 \rightarrow Rs$
$*Rs \rightarrow CRs_1$
$Rs - 32 \rightarrow Rs$
$*Rs \rightarrow CRs_2$
$Rs - 32 \rightarrow Rs$
$*Rs \rightarrow CRs_2$
$|CRs_1 + CRs_2| \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	transfers		
1	0	0	0	1	0	0	1	size	0	0	R	Rs			
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs_1	CRs_2	CRd	1000 1001 s000 0000

Operands

Rs TMS34020 register containing the memory address

CRs_1 TMS34082 register to contain the first operand

CRs_2 TMS34082 register to contain the second operand

CRd TMS34082 destination register

Description

ADDAx loads the contents of memory pointed to by Rs into CRs_1 and CRs_2, adds the contents of CRs_1 and CRs_2, takes the absolute value, and stores the result in CRd. Before each load from memory, Rs is decremented by 32.

CRs_1 and CRs_2, the two TMS34082 source registers, must be in opposite register files.

The integer form of this instruction is not supported.

Instruction Type

CMOVMC, predecrement, constant count

Example

```
ADDAD –*A5, RA7, RB9, RB0
```

This example loads memory starting at the address given by TMS34020 register A5 minus 32 into TMS34082 registers RA7 and RB9, adds the double-precision floating-point contents of RA7 and RB9, takes the absolute value, and stores the result in RB0.

Syntax

Type	Syntax
Double-Precision	**ASUBAD** *CRs$_1$, CRs$_2$, CRd*
Single-Precision	**ASUBAF** *CRs$_1$, CRs$_2$, CRd*

Execution

$|CRs_1| - |CRs_2| \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	0	1	1	1	size
ID			CRs$_1$				CRs$_2$				CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs$_1$	CRs$_2$	CRd	0001 0111 s000 0000

Operands

CRs$_1$ TMS34082 register containing the first operand. Must be from RA register file.

CRs$_2$ TMS34082 register containing the second operand. Must be from RB register file.

CRd TMS34082 destination register.

Description

ASUBADx subtracts the absolute value of CRs$_2$ from the absolute value of CRs$_1$, placing the result in CRd.

The integer form of this instruction is not supported.

Instruction Type

CEXEC, short

Example

```
ASUBAF RA7, RB2, C
```

This example subtracts the absolute value of the single-precision contents of RB2 from the absolute value of the single-precision contents of RA7 and stores the result in the C register.

Syntax

ASUBAF *Rs_1, Rs_2, CRs_1, CRs_2, CRd*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
$|CRs_1| - |CRs_2| \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	1	0	1	1	1	0	0	0	R	Rs_2			
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs_1		CRs_2		CRd		0101 0111 0000 0000	

Operands

Rs_1 TMS34020 source register for first 32-bit single-precision floating-point operand

Rs_2 TMS34020 source register for second 32-bit single-precision floating-point operand

CRs_1 TMS34082 register to contain the first single-precision operand. Must be from RA register file

CRs_2 TMS34082 register to contain the second single-precision operand. Must be from RB register file

CRd TMS34082 destination register

Description

ASUBAF loads the contents of Rs_1 and Rs_2 into CRs_1 and CRs_2, respectively, and subtracts the absolute value in CRs_2 from the absolute value in CRs_1, placing the result in CRd.

The integer and double-precision forms of this instruction are not supported.

Instruction Type

CMOVGC, two registers

Example

```
ASUBAF A3, A2, RA5, RB3, RB1
```

This example loads the contents of TMS34020 registers A3 and A2 into RA5 and RB3 respectively, subtracts the absolute value of the contents of RB3 from the absolute value of RA5, and stores the result in RB1.

Syntax

Type	Syntax
Double-Precision	**ASUBAD** **Rs+, CRs$_1$, CRs$_2$, CRd*
Single-Precision	**ASUBAF** **Rs+, CRs$_1$, CRs$_2$, CRd*

Execution

*Rs → CRs$_1$
Rs + 32 → Rs
*Rs → CRs$_1$
Rs + 32 → Rs
*Rs → CRs$_2$
Rs + 32 → Rs
*Rs → CRs$_2$
Rs + 32 → Rs
|CRs$_1$| – |CRs$_2$| → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	transfers		
1	0	0	1	0	1	1	1	size	0	0	R	Rs			
ID			CRs$_1$				CRs$_2$				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs$_1$		CRs$_2$		CRd		1001 0111 s000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs$_1$ TMS34082 register to contain the first operand. Must be from RA register file.

CRs$_2$ TMS34082 register to contain the second operand. Must be from RB register file.

CRd TMS34082 destination register

Description

ASUBAx loads the contents of memory pointed to by Rs into CRs$_1$ and CRs$_2$ and subtracts the absolute value in CRs$_2$ from the absolute value in CRs$_1$, placing the result in CRd. After each load from memory, Rs is incremented by 32.

The integer form of this instruction is not supported.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
ASUBAD *A3+, RA7, RB3, RB1
```

This example loads memory starting at the address given by TMS34020 register A3 into TMS34082 registers RA7 and RB3, subtracts the absolute value of the contents of RB3 from the absolute value of RA7, and stores the result in RB1.

Syntax

Type	Syntax
Double-Precision	**ASUBAD** $-*Rs, CRs_1, CRs_2, CRd$
Single-Precision	**ASUBAF** $-*Rs, CRs_1, CRs_2, CRd$

Execution

$Rs - 32 \rightarrow Rs$
$*Rs \rightarrow CRs_1$
$Rs - 32 \rightarrow Rs$
$*Rs \rightarrow CRs_1$
$Rs - 32 \rightarrow Rs$
$*Rs \rightarrow CRs_2$
$Rs - 32 \rightarrow Rs$
$*Rs \rightarrow CRs_2$
$|CRs_1| - |CRs_2| \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	transfers		
1	0	0	1	0	1	1	1	size	0	0	R	Rs			
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs_1		CRs_2		CRd		1001 0111 s000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs_1 TMS34082 register to contain the first operand. Must be from RA register file.

CRs_2 TMS34082 register to contain the second operand. Must be from RB register file.

CRd TMS34082 destination register

Description

ASUBAx loads the contents of memory pointed to by Rs into CRs_1 and CRs_2 and subtracts the absolute value in CRs_2 from the absolute value in CRs_1, placing the result in CRd. Before each load from memory, Rs is decremented by 32.

The integer form of this instruction is not supported.

Instruction Type

CMOVMC, predecrement, constant count

Example

```
ASUBAF –*A3, RA7, RB3, RB1
```

This example loads memory starting at the address given by TMS34020 register A3 minus 32 into TMS34082 registers RA7 and RB3, subtracts the absolute values of the contents of RB3 and RB7, and stores the result in RB1.

Syntax

Type	Syntax
Integer	**BACKF**
Double-Precision	**BACKFD**
Single-Precision	**BACKFF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	0	0	1	0	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction to '34082

31	29							0
ID	0	0000	0000	0000	0010	010t	s000	0000

Description

A convex polygon is tested to determine whether it is facing the current view area or if it is facing away from the current view area. This allows the elimination of polygons that do not need to be drawn in the current image. The first three vertices of the polygon are entered and tested as to rotation direction. If the direction is clockwise (forward facing), the polygon is visible; if the direction is counterclockwise (backward facing), then the polygon is invisible. This instruction also detects the case where the plane defined by the three points passes through the viewing point (position) of the eye. In this case, the polygon may be drawn as a line or ignored. The algorithm assumes that all of the vertices of the polygon lie on the plane defined by the first three vertices.

Implied Operands

RA0 = X0,	RA1 = Y0,	RA2 = Z0,	RA3 = W0
RA4 = X1,	RA5 = Y1,	RA6 = Z1,	RA7 = W1
RB0 = X2,	RB1 = Y2,	RB2 = Z2,	RB3 = W2

where Xn,Yn,Zn,Wn are the coordinates of vertex Vn, already stored in the coprocessor registers.

Algorithm

```
C = RB1 × RA3                ; Y2 × W0
C = C – (RA1 × RB3)          ; (Y2 × W0) – (Y0 × W2)
RB8 = C × RA4                ; ((Y2 × W0) – (Y0 × W2)) × X1
C = RA0 × RB3                ; X0 × W2
C = C – (RB0 × RA3)          ; (X0 × W2) – (X2 – W0)
RB9 = C × RA5                ; ((X0 × W2) – (X2 × W0)) × Y1
C = RB0 × RA1                ; X2 × Y0
C = C – (RA0 × RB1)          ; (X2 × Y0) – (Y2 × Y0)
RA8 = C × RA7                ; ((X2 × Y0) – (Y2 × X0)) × W1
RA8 = RA8 + RB9              ; ((X2 × Y0) – (Y2 × X0)) × W1
                             ;  + ((X0 × W2) – (X2 × W0)) × Y1
RA8 = RA8 + RB8              ; ((Y2 × W0) – (Y0 × W2)) × X1
                             ;  + ((X0 × W2) – (X2 × W0)) × Y1
                             ;  + ((X2 × Y0) – (Y2 × X0)) × W1
if RA8 < 0 then N = 1        ; set N as appropriate
  else N = 0
if RA8 = 0 then Z = 1        ; set Z as appropriate
  else Z = 0
```

Temporary Storage C, CT, RA8, RB8, RB9

Outputs The N and V status bits are set to indicate the following:

N	Z	Description
0	0	Polygon is forward facing
0	1	Polygon is parallel to view (reject or draw as line)
1	0	Polygon is backward facing
1	1	Polygon is backward facing

Instruction Type CEXEC, short

Syntax

CHECK *Rd*

Execution

If coprocessor is busy
FFFF FFFFh → Rd
If coprocessor is idle
0000 0000h → Rd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	1	R	Rd			
0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0
ID			0	0	0	0	1	1	0	1	0	0	0	0	0

Instruction to '34082

31	29														0
ID	0	0001		1010		0000		0001		1110		0000		0000	

Operands

Rd TMS34020 destination register for status information

Description

CHECK checks the status of the coprocessor. If the TMS34082 coprocessor is busy, CHECK sets all the bits in Rd to 1. If the TMS34082 coprocessor is idle, CHECK sets all the bits in Rd to 0.

This instruction allows polling of the TMS34082 prior to sending subsequent instructions to avoid halting the TMS34020 if the FPU is not ready to accept new commands. This polling may be required for user-defined instruction sequences that utilize the external program and data memory of the TMS34082.

Instruction Type

CMOVGC, one register

Example

```
CHECK A4
```

If the TMS34082 coprocessor is busy, this example sets all the bits in register A4 to 1. If the TMs34082 coprocessor is idle, this example resets all the bits in register A4 to 0.

Syntax

Type	Syntax
Integer	**CKVTX**
Double-Precision	**CKVTXD**
Single-Precision	**CKVTXF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	0	1	0	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction to '34082

31	29				0
ID	0	0000 0000 0000 0011	010t	s000	0000

Description

The CKVTXx instruction is used to compare polygon vertices to the viewing volume in a perspective display. It may be used with a list of vertices describing a polygon to determine if the entire polygon is totally within, totally outside, or partially within the clipping volume. The TMS34082 must be initialized with the CKVTXIx instruction before the first iteration. The vertices must be specified using homogeneous coordinates.

Implied Operands

```
RA0 = Xn                    ; vertex Vn [Xn, Yn, Zn, Wn] to check,
RA1 = Yn                    ; these are homogeneous coordinates
RA2 = Zn
RA3 = Wn
```

Algorithm

```
RB9 = RA3                   ; copy RA3 to RB9
If (RB9 – |RA0|) < 0
  set XLT                   ; X OR outcode, status bit 5
 else
  reset XGT                 ; X AND outcode, status bit 6

If (RB9 – |RA1|) < 0
  set YLT                   ; Y OR outcode, status bit 7
else
  reset YGT                 ; Y AND outcode, status bit 8

If (RB9 – |RA2|) < 0
  set ZLT                   ; Z OR outcode, status bit 9
else
  reset ZGT                 ; Z AND outcode, status bit 10

If ((XGT OR YGT OR ZGT) = 1)
  set V bit                 ; if AND outcode = 1, then outside
else
  reset V bit               ; all AND outcodes = 0, partially visible

If ((XLT OR YLT OR ZLT) = 0)
  set Z bit                 ; if OR outcode = 0, then inside
else
  reset Z bit               ; all OR outcodes = 1, not entirely inside
```

You may now reload vertex V(n+1) and repeat the instruction for all vertices in a polygon.

Temporary Storage C, RB9

Outputs The status is set (ZGT, ZLT, YGT, YLT, XGT, and XLT) according to position.
V = 1 Vertex out
Z = 1 Vertex in

If repeated for all vertices in a polygon then:

V	Z	Description
0	0	The polygon crosses the boundary of the clipping volume
0	1	The polygon is totally inside the clipping volume
1	0	The polygon is totally outside the clipping volume
1	1	Not valid

The boundaries of the clipping volume that are crossed by the polygon may be determined by the ZLT (Z-plane), YLT (Y-plane), and XLT (X-plane) bits.

Instruction Type CEXEC, short

Example

```
CKVTXI
MOVF *A5+, RA0, 4
CKVTXF
```

This example first initializes the TMS34082 by executing the check vertex initialize instruction. Then the four homogeneous coordinates of the vertex are loaded, starting at the address given in TMS34020 register A5. Finally the status register is set according to the results of the check.

Syntax

CKVTXI

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	0	0
ID			0	0	0	0	1	1	0	0	0	0	0	0	0

Instruction to '34082

31	29							0
ID	0	0001	1000	0000	0001	1100	0000	0000

Description

The CKVTXI instruction is used to initialize several bits in the status register before the first Check Vertex (CKVTX) instruction.

Algorithm

```
reset XLT          ;set starting X OR outcode to 0
reset YLT          ;set starting Y OR outcode to 0
reset ZLT          ;set starting Z OR outcode to 0
set XGT            ;set starting X AND outcode to 1
set YGT            ;set starting Y AND outcode to 1
set ZGT            ;set starting Z AND outcode to 1
```

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**CLIPCF**
Double-Precision	**CLIPCFD**
Single-Precision	**CLIPCFF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	0	1	1	1	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction to '34082

31	29						0
ID	0	0000	0000	0000	0010	111t	s000	0000

Description

The CLIPCFx instruction clips a color value of the first vertex of a Gouraud shaded line after the first vertex has been clipped to the viewing volume using the CLIPFx instruction. The clipped color value represents the color value (red, green, blue) for the endpoint of the line when the line is perspective-projected to the viewing surface. The interpolation factor (t) from the CLIPFx instruction is modified to take into account the color distortion caused by perspective transformation.

Implied Operands

RA3 = W1' (intensity)
RA4 = R1 (red)
RA5 = B1 (blue)
RA6 = G1(green)
RB3 = W2
RB4 = R2 (red)
RB5 = B2 (blue)
RB6 = G2 (green)
C = t (interpolation factor) from CLIPF*x* instruction

Algorithm

```
C    = C × RB3          ; t × W2
RB9  = RA3
RA8  = RB4 – RA4        ; R2 – R1
C    = C / RB9          ; t' = t × W2 / W1'
RA9  = RB5 – RA5        ; B2 – B1
CT   = RA8 × C          ; (R2 – R1) × t'
RA4  = CT + RA4         ; R1' = R1 + (R2 – R1) × t'
CT   = RA9 × C          ; (B2 – B1) × t'
RA5  = CT + RA5         ; B1' = B1 + (B2 – B1) × t'
RA8  = RB6 – RA6        ; G2 – G1
CT   = RA8 × C          ; (G2 – G1) × t'
RA6  = RA6 + CT         ; G1' = G1 + (G2 – G1) × t'
```

Temporary Storage

CT, RA8, RA9

Outputs

RA4 = R1' (red)
RA5 = B1' (blue)
RA6 = G1' (green)
CT = t'

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**CLIPCR**
Double-Precision	**CLIPCRD**
Single-Precision	**CLIPCRF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	0	0	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction to '34082

31	29							0
ID	0	0000	0000	0000	0011	100t	s000	0000

Description

The CLIPCRx instruction clips a color value of the second vertex of a Gouraud shaded line after the second vertex has been clipped to the viewing volume using the CLIPRx instruction. The clipped color value represents the color value (red, green, blue) for the endpoint of the line when the line is perspective-projected to the viewing surface. The interpolation factor (t) distortion caused by perspective transformation.

Implied Operands

RA3 = W2' (intensity)
RA4 = R1 (red)
RA5 = B1 (blue)
RA6 = G1(green)
RB4 = R2 (red)
RB5 = B2 (blue)
RB6 = G2 (green)
RB7 = W1 (intensity)
C = t (interpolation factor) from CLIPR*x* instruction

Algorithm

```
C    = C × RB7      ; t × W1
RB9  = RA3          ;
RA8  = RA4 – RB4    ; R1– R2
C    = C / RB9      ; t' = t × W1 / W2'
RA9  = RA5 – RB5    ; B1 – B2
CT   = RA8 × C      ; (R1 – R2) × t'
RA4  = CT + RB4     ; R2' = R2 + (R1 – R2) × t'
CT   = RA9 × C      ; (B1 – B2) × t'
RA5  = CT + RB5     ; B2' = B2 + (B1 – B2) × t'
RA8  = RA6 – RB6    ; G1 – G2
CT   = RA8 × C      ; (G1 – G2) × t'
RA6  = RA6 + CT     ; G2' = G2 + (G1 – G2) × t'
```

Temporary Storage

CT, RA8, RA9, RB9

Outputs

RA4 = R2' (red)
RA5 = B2' (blue)
RA6 = G2' (green)
CT = t'

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**CLIPFX**
Double-Precision	**CLIPFXD**
Single-Precision	**CLIPFXF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	0	1	0	1	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction to '34082

31	29						0
ID	0	0000	0000	0000	0010	101t	s000	0000

Description

The CLIPFXx Instruction clips a line to the viewing volume when its first endpoint is outside the clipping (viewable) volume. Use CLIPFXx only if the X coordinate of the first endpoint of a line is outside of the viewing volume. It also provides an interpolation factor that is used by the CLIPCx instruction when performing Gouraud shading. The endpoints are described by the homogeneous coordinates P1 = [X1, Y1, Z1, W1] and P2 = [X2, Y2, Z2, W2].

Implied Operands

RA0 = X1	RB0 = X2
RA1 = Y1	RB1 = Y2
RA2 = Z1	RB2 = Z2
RA3 = W1	RB3 = W2

Algorithm

```
C  = RA0
CT = RB0
If RA0 < 0 then set (N=1)
If N = 1 then
  RB8 = RB3 + CT          ; b = W2 + X2
  RA8 = RA3 + C           ; a = W1 + X1
else
  RB8 = RB3 – CT          ; b = W2 – X2
  RA8 = RA3 – C           ; a = W1 – X1
RB9 = RB0 – RA0           ; X2 – X1
RB8 = RA8 – RB8           ; a – b
RA9 = RB1 – RA1           ; Y2 – Y1
C   = RA8 / RB8           ; t = a / (a – b)
RA8 = RB2 – RA2           ; Z2 – Z1
CT  = RB9 × C             ; (X2 – X1) × t
RA0 = CT + RA0            ; X1' = X1 + (X2 – X1) × t
CT  = RA9 × C             ; (Y2 – Y1) × t
RA1 = CT + RA1            ; Y1' = Y1 + (Y2 – Y1) × t
RA9 = RB3 – RA3           ; W2 – W1
CT  = RA8 × C             ; (Z2 – Z1) × t
RA2 = CT + RA2            ; Z1' = Z1 + (Z2 – Z1) × t
CT  = RA9 × C             ; (W2 – W1) × t
RA3 = CT + RA3            ; W1' = W1 + (W2 – W1) × t
```

Temporary Storage CT, RA8, RA9, RB8, RB9

Outputs RA0 = X1'
RA1 = Y1'
RA2 = Z1'
RA3 = W1'
C = t

Instruction Type CEXEC, short

Syntax

Type	Syntax
Integer	**CLIPFY**
Double-Precision	**CLIPFYD**
Single-Precision	**CLIPFYF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	0	1	0	1	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	1

Instruction to '34082

31	29						0
ID	0	0000	0000	0001	0010	101t	s000	0000

Description

The CLIPFYx Instruction clips a line to the viewing volume when its first endpoint is outside the clipping (viewable) volume. Use CLIPFYx only if the Y coordinate of the first endpoint of a line is outside of the viewing volume. It also provides an interpolation factor that is used by the CLIPCx instruction when performing Gouraud shading. The endpoints are described by the homogeneous coordinates P1 = [X1, Y1, Z1, W1] and P2 = [X2, Y2, Z2, W2].

Implied Operands

RA0 = X1	RB0 = X2
RA1 = Y1	RB1 = Y2
RA2 = Z1	RB2 = Z2
RA3 = W1	RB3 = W2

Algorithm

```
C  = RA1
CT = RB1
If RA1 < 0 then set (N=1)
If N = 1 then
  RB8 = RB3 + CT          ; b = W2 + Y2
  RA8 = RA3 + C           ; a = W1 + Y1
else
  RB8 = RB3 – CT          ; b = W2 – Y2
  RA8 = RA3 – C           ; a = W1 – Y1
RB9 = RB0 – RA0           ; X2 – X1
RB8 = RA8 – RB8           ; a – b
RA9 = RB1 – RA1           ; Y2 – Y1
C   = RA8 / RB8           ; t = a / (a – b)
RA8 = RB2 – RA2           ; Z2 – Z1
CT  = RB9 × C             ; (X2 – X1) × t
RA0 = CT + RA0            ; X1' = X1 + (X2 – X1) × t
CT  = RA9 × C             ; (Y2 – Y1) × t
RA1 = CT + RA1            ; Y1' = Y1 + (Y2 – Y1) × t
RA9 = RB3 – RA3           ; W2 – W1
CT  = RA8 × C             ; (Z2 – Z1) × t
RA2 = CT + RA2            ; Z1' = Z1 + (Z2 – Z1) × t
CT  = RA9 × C             ; (W2 – W1) × t
RA3 = CT + RA3            ; W1' = W1 + (W2 – W1) × t
```

Temporary Storage CT, RA8,RA9, RB8, RB9

Outputs RA0 = X1'
RA1 = Y1'
RA2 = Z1'
RA3 = W1'
C = t

Instruction Type CEXEC, short

Syntax

Type	Syntax
Integer	**CLIPFZ**
Double-Precision	**CLIPFZD**
Single-Precision	**CLIPFZF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	0	1	0	1	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	1	0

Instruction to '34082

31	29							0
ID	0	0000	0000	0010	0010	101t	s000	0000

Description

The CLIPFZx Instruction clips a line to the viewing volume when its first endpoint is outside the clipping (viewable) volume. Use CLIPFZx only if the Z coordinate of the first endpoint of a line is outside of the viewing volume. It also provides an interpolation factor that is used by the CLIPCx instruction when performing Gouraud shading. The endpoints are described by the homogeneous coordinates P1 = [X1, Y1, Z1, W1] and P2 = [X2, Y2, Z2, W2].

Implied Operands

RA0 = X1	RB0 = X2
RA1 = Y1	RB1 = Y2
RA2 = Z1	RB2 = Z2
RA3 = W1	RB3 = W2

Algorithm

```
C  = RA2
CT = RB2
If RA2 < 0 then set (N=1)
If N = 1 then
  RB8 = RB3 + CT          ; b = W2 + Z2
  RA8 = RA3 + C           ; a = W1 + Z1
else
  RB8 = RB3 – CT          ; b = W2 – Z2
  RA8 = RA3 – C           ; a = W1 – Z1
RB9 = RB0 – RA0           ; X2 – X1
RB8 = RA8 – RB8           ; a – b
RA9 = RB1 – RA1           ; Y2 – Y1
C   = RA8 / RB8           ; t = a / (a – b)
RA8 = RB2 – RA2           ; Z2 – Z1
CT  = RB9 × C             ; (X2 – X1) × t
RA0 = CT + RA0            ; X1' = X1 + (X2 – X1) × t
CT  = RA9 × C             ; (Y2 – Y1) × t
RA1 = CT + RA1            ; Y1' = Y1 + (Y2 – Y1) × t
RA9 = RB3 – RA3           ; W2 – W1
CT  = RA8 × C             ; (Z2 – Z1) × t
RA2 = CT + RA2            ; Z1' = Z1 + (Z2 – Z1) × t
CT  = RA9 × C             ; (W2 – W1) × t
RA3 = CT + RA3            ; W1' = W1 + (W2 – W1) × t
```

Temporary Storage CT, RA8, RA9, RB8, RB9

Outputs RA0 = X1'
RA1 = Y1'
RA2 = Z1'
RA3 = W1'
C = t

Instruction Type CEXEC, short

Syntax

Type	Syntax
Integer	**CLIPRX**
Double-Precision	**CLIPRXD**
Single-Precision	**CLIPRXF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	0	1	1	0	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction to '34082

31	29														0
ID	0	0000		0000		0010		0010		110t		s000		0000	

Description

The CLIPRXx Instruction clips a line to the viewing volume when its second endpoint is outside the clipping (viewable) volume. Use CLIPRXx only if the X coordinate of the second endpoint of a line is outside of the viewing volume. It also provides an interpolation factor that is used by the CLIPCRx instruction when performing Gouraud shading. The endpoints are described by the homogeneous coordinates P1 = [X1, Y1, Z1, W1] and P2 = [X2, Y2, Z2, W2].

Implied Operands

RA0 = X1	RB0 = X2
RA1 = Y1	RB1 = Y2
RA2 = Z1	RB2 = Z2
RA3 = W1	RB3 = W2

Algorithm

```
CT = RB0
C  = RA0
If RB0 < 0 then set (N=1)
If N = 1 then
  RB8 = RA3 + C          ; b = W1 – X1
  RA8 = RB3 + CT         ; a = W2 – X2
else
  RB8 = RA3 – C          ; b = W1 + X1
  RA8 = RB3 – CT         ; a = W2 + X2
RB9 = RA0 – RB0          ; X1 – X2
RB8 = RA8 – RB8          ; a – b
RA9 = RA1 – RB1          ; Y1 – Y2
C   = RA8 / RB8          ; t = a / (a – b)
RA8 = RA2 – RB2          ; Z1 – Z2
CT  = RB9 × C            ; (X1 – X2) × t
RA0 = CT + RB0           ; X2' = X2 + (X1 – X2) × t
CT  = RA9 × C            ; (Y1 – Y2) × t
RA1 = CT + RB1           ; Y2' = Y2 + (Y1 – Y2) × t
RA9 = RA3 – RB3          ; W1 – W2
CT  = RA8 × C            ; (Z1 – Z2) × t
RA2 = CT + RB2           ; Z2' = Z2 + (Z1 – Z2) × t
CT  = RA9 × C            ; (W1 – W2) × t
RA3 = CT + RB3           ; W2' = W2 + (W1 – W2) × t
```

Temporary Storage CT, RA8, RA9, RB8, RB9

Outputs This writes [X2',Y2',Z2',W2'] over [X1,Y1,Z1,W1].
RA0 = X2'
RA1 = Y2'
RA3 = Z2'
RA4 = W2'
C = t

Instruction Type CEXEC, short

Syntax

Type	Syntax
Integer	**CLIPRY**
Double-Precision	**CLIPRYD**
Single-Precision	**CLIPRYF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	0	1	1	0	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	1

Instruction to '34082

31	29						0
ID	0	0000	0000	0001	0010	110t	s000	0000

Description

The CLIPRYx Instruction clips a line to the viewing volume when its second endpoint is outside the clipping (viewable) volume. Use CLIPRYx only if the Y coordinate of the second endpoint of a line is outside of the viewing volume. It also provides an interpolation factor that is used by the CLIPCRx instruction when performing Gouraud shading. The endpoints are described by homogeneous coordinates P1 = [X1, Y1, Z1, W1] and P2 = [X2, Y2, Z2, W2].

Implied Operands

RA0 = X1 RB0 = X2
RA1 = Y1 RB1 = Y2
RA2 = Z1 RB2 = Z2
RA3 = W1 RB3 = W2

Algorithm

```
CT = RB1
C  = RA1
If RB1 < 0 then set (N = 1)
If N = 1 then
  RB8 = RA3 + C          ; b = W1 – Y1
  RA8 = RB3 + CT         ; a = W2 – Y2
else
  RB8 = RA3 – C          ; b = W1 + Y1
  RA8 = RB3 – CT         ; a = W2 + Y2
RB9 = RA0 – RB0          ; X1 – X2
RB8 = RA8 – RB8          ; a – b
RA9 = RA1 – RB1          ; Y1 – Y2
C   = RA8 / RB8          ; t = a / (a – b)
RA8 = RA2 – RB2          ; Z1 – Z2
CT  = RB9 × C            ; (X1 – X2) × t
RA0 = CT + RB0           ; X2' = X2 + (X1 – X2) × t
CT  = RA9 × C            ; (Y1 – Y2) × t
RA1 = CT + RB1           ; Y2' = Y2 + (Y1 – Y2) × t
RA9 = RA3 – RB3          ; W1 – W2
CT  = RA8 × C            ; (Z1 – Z2) × t
RA2 = CT + RB2           ; Z2' = Z2 + (Z1 – Z2) × t
CT  = RA9 × C            ; (W1 – W2) × t
RA3 = CT + RB3           ; W2' = W2 + (W1 – W2) × t
```

Temporary Storage CT, RA8, RA9, RB8, RB9

Outputs This writes [X2',Y2',Z2',W2'] over [X1,Y1,Z1,W1].
RA0 = X2'
RA1 = Y2'
RA3 = Z2'
RA4 = W2'
C = t

Instruction Type CEXEC, short

Syntax

Type	Syntax
Integer	**CLIPRZ**
Double-Precision	**CLIPRZD**
Single-Precision	**CLIPRZF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	0	1	1	0	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	1	0

Instruction to '34082

31	29							0
ID	0	0000	0000	0010	0010	110t	s000	0000

Description

The CLIPRZx Instruction clips a line to the viewing volume when its second endpoint is outside the clipping (viewable) volume. Use CLIPRZx only if the Z coordinate of the second endpoint of a line is outside of the viewing volume. It also provides an interpolation factor that is used by the CLIPCRx instruction when performing Gouraud shading. The endpoints are described by the homogeneous coordinates P1 = [X1, Y1, Z1, W1] and P2 = [X2, Y2, Z2, W2].

Implied Operands

RA0 = X1 RB0 = X2
RA1 = Y1 RB1 = Y2
RA2 = Z1 RB2 = Z2
RA3 = W1 RB3 = W2

Algorithm

```
CT = RB2
C  = RA2
If RA2 < 0 then set (N = 1)
If N = 1 then
  RB8 = RA3 + C          ; b = W1 – Z1
  RA8 = RB3 + CT         ; a = W2 – Z2
else
  RB8 = RA3 – C          ; b = W1 + Z1
  RA8 = RB3 – CT         ; a = W2 + Z2
RB9 = RA0 – RB0          ; X1 – X2
RB8 = RA8 – RB8          ; a – b
RA9 = RA1 – RB1          ; Y1 – Y2
C   = RA8 / RB8          ; t = a / (a – b)
RA8 = RA2 – RB2          ; Z1 – Z2
CT  = RB9 × C            ; (X1 – X2) × t
RA0 = CT + RB0           ; X2' = X2 + (X1 – X2) × t
CT  = RA9 × C            ; (Y1 – Y2) × t
RA1 = CT + RB1           ; Y2' = Y2 + (Y1 – Y2) × t
RA9 = RA3 – RB3          ; W1 – W2
CT  = RA8 × C            ; (Z1 – Z2) × t
RA2 = CT + RB2           ; Z2' = Z2 + (Z1 – Z2) × t
CT  = RA9 × C            ; (W1 – W2) × t
RA3 = CT + RB3           ; W2' = W2 + (W1 – W2) × t
```

Temporary Storage CT, RA8, RA9, RB8, RB9

Outputs This writes [X2',Y2',Z2',W2'] over [X1,Y1,Z1,W1].
RA0 = X2'
RA1 = Y2'
RA3 = Z2'
RA4 = W2'
C = t

Instruction Type CEXEC, short

Syntax

Type	Syntax
Integer	**CLR** *CRd*
Double-Precision	**CLRD** *CRd*
Single-Precision	**CLRF** *CRd*

Execution

0 → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	0	1	type	size
ID			1	1	0	1	1	1	0	1	CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	1101	1101	CRd	0000 001t s000 0000

Operands

CRd TMS34082 destination register.

Description

CLRx loads a zero of the appropriate type in the register, CRd. The Z (zero) bit in the status register will be set also.

Instruction Type

CEXEC, short

Example

```
CLRF C
```

This example loads a single-precision floating-point zero into TMS34082 register C.

Syntax

Type	Syntax
Integer	**CMP** *CRs$_1$, CRs$_2$*
Double-Precision	**CMPD** *CRs$_1$, CRs$_2$*
Single-Precision	**CMPF** *CRs$_1$, CRs$_2$*

Execution

Flags (CRs$_1$ – CRs$_2$) → TMS34082 Status Register

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	1	0	type	size
ID			CRs$_1$				CRs$_2$				0	0	0	0	0

Instruction to '34082

31	29	28	25	24	21	20	0
ID		CRs$_1$		CRs$_2$		00000 0000 0101 s000 0000	

Operands

CRs$_1$ TMS34082 register containing the first operand. Must be from RA register file.

CRs$_2$ TMS34082 register containing the second operand. Must be from RB register file.

Description

CMPx subtracts the contents of CRs$_2$ from CRs$_1$ and sets the appropriate status bits in the TMS34082 status register.

Instruction Type

CEXEC, short

Example

```
CMP RA5, RB6
```

This example subtracts the integer contents of RB6 from RA5 and sets the status bits in the TMS34082 status register.

Syntax

Type	Syntax
Integer	**CMP** *Rs_1, Rs_2, CRs_1, CRs_2*
Single-Precision	**CMPF** *Rs_1, Rs_2, CRs_1, CRs_2*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
Flags ($CRs_1 - CRs_2$) $\rightarrow$ TMS34082 Status Register

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	0	1	0	t	0	0	0	R	Rs_2			
ID			CRs_1				CRs_2				0	0	0	0	0

Instruction to '34082

31	29	28	25	24	21	20				0
ID		CRs_1		CRs_2		00000	0100	010t	0000	0000

Operands

Rs_1 TMS34020 source register for the first value to TMS34082

Rs_2 TMS34020 source register for the second value to TMS34082

CRs_1 TMS34082 register to contain the first operand. Must be from RA register file.

CRs_2 TMS34082 register to contain the second operand. Must be from RB register file.

Description

CMPx loads the contents of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, subtracts CRs_2 from CRs_1, and sets the appropriate status bits in the TMS34082 status register.

The double-precision form of this instruction is not supported.

Instruction Type

CMOVGC, two registers

Example

```
CMPF A5, A6, RA5, RB6
```

This example loads TMS34020 registers A5 and A6 into TMS34082 registers RA5 and RB6 respectively, subtracts the single-precision floating-point contents of RB6 from the contents of RA5, and sets the status bits in the TMS34082 status register.

Syntax

Type	Syntax
Integer	**CMP** **Rs+, CRs$_1$, CRs$_2$*
Double-Precision	**CMPD** **Rs+, CRs$_1$, CRs$_2$*
Single-Precision	**CMPF** **Rs+, CRs$_1$, CRs$_2$*

Execution

*Rs → CRs$_1$
Rs + 32 → Rs
*Rs → CRs$_1$
Rs + 32 → Rs
*Rs → CRs$_2$
Rs + 32 → Rs
*Rs → CRs$_2$
Rs + 32 → Rs
Flags (CRs$_1$ – CRs$_2$) → TMS34082 Status Register

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	transfers		
1	0	0	0	0	1	0	t	s	0	0	R	Rs			
ID			CRs$_1$				CRs$_2$				0	0	0	0	0

Instruction to '34082

31	29	28	25	24	21	20	0
ID		CRs$_1$		CRs$_2$		00000 1000 010t s000 0000	

Operands

Rs — TMS34020 register containing the memory address

CRs$_1$ — TMS34082 register to contain the first operand. Must be from RA register file.

CRs$_2$ — TMS34082 register to contain the second operand. Must be from RB register file.

Description

CMPx loads the contents of memory pointed to by Rs into CRs$_1$ and CRs$_2$, subtracts CRs$_2$ from CRs$_1$, and sets the appropriate status bits in the TMS34082 status register. After each load from memory, Rs is incremented by 32.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
CMP *A5+, RA5,RB6
```

This example loads the contents of memory starting at the address given by TMS34020 register A5 into TMS34082 registers RA5 and RB6, subtracts the integer contents of RB6 from RA5, and sets the status bits in the TMS34082 status register.

Syntax

Type	Syntax
Integer	**CMP** $-*Rs, CRs_1, CRs_2$
Double-Precision	**CMPD** $-*Rs, CRs_1, CRs_2$
Single-Precision	**CMPF** $-*Rs, CRs_1, CRs_2$

Execution

Rs – 32 → Rs
*Rs → CRs_1
Rs – 32 → Rs
*Rs → CRs_1
Rs – 32 → Rs
*Rs → CRs_2
Rs – 32 → Rs
*Rs → CRs_2
Flags (CRs_1 – CRs_2) → TMS34082 Status Register

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	transfers		
1	0	0	0	0	1	0	t	s	0	0	R	Rs			
ID			CRs_1				CRs_2				0	0	0	0	0

Instruction to '34082

31	29	28	25	24	21	20	0
ID		CRs_1		CRs_2		00000 1000 010t s000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs_1 TMS34082 register to contain the first operand. Must be from RA register file.

CRs_2 TMS34082 register to contain the second operand. Must be from RB register file.

Description

CMPx loads the contents of memory pointed to by Rs into CRs_1 and CRs_2, subtracts CRs_2 from CRs_1, and sets the appropriate status bits in the TMS34082 status register. Before each load from memory, Rs is decremented by 32.

Instruction Type

CMOVMC, predecrement, constant count

Example

```
CMP -*A5, RA5, RB6
```

This example loads the integer contents of memory starting at the address given by TMS34020 register A5 minus 32 into TMS34082 registers RA5 and RB6, subtracts the integer contents of RB6 from RA5, and sets the status bits in the TMS34082 status register.

Syntax

Type	Syntax
Integer	**CONV**
Double-Precision	**CONVD**
Single-Precision	**CONVF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	0	1	1	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction to '34082

31	29						0
ID	0	0000	0000	0000	0011	011t	s000	0000

Description

The CONVx instruction performs the multiplies and accumulates for a 3×3 convolution assuming the constants (C9-C1) and the integer values (P9-P1) are in TMS34082 registers. The convolution divide constant (K) is maintained in register RA9for the integer instruction (CONV). For floating-point instructions (CONVD and CONVF), the inverse of the divide constant is maintained in RA9 to reduce the division to a single multiply. Note that K is typically greater than zero.

Implied Operands

RA0 = P1 RA3 = P4 RA6 = P7
RA1 = P2 RA4 = P5 RA7 = P8
RA2 = P3 RA5 = P6 RA8 = P9

RA9 = K or (for the integer instruction, CONV)
RA9 = 1/K (for floating-point instructions, CONVD and CONVF)

RB0 = C1 RB3 = C4 RB6 = C7
RB1 = C2 RB4 = C5 RB7 = C8
RB2 = C3 RB5 = C6 RB8 = C9

Algorithm

```
C  = RA0 × RB0                    ; determine influence due to points P9-P1
CT = C + (RA1 × RB1)
C  = CT + (RA2 × RB2)
CT = C + (RA3 × RB3)
C  = CT + (RA4 × RB4)
CT = C + (RA5 × RB5)
C  = CT + (RA6 × RB6)
CT = C + (RA7 × RB7)
C  = CT + (RA8 × RB8)
If type = integer, then
   RB9 = C / RA9                  ; divide by the convolution divide constant
else
   RB9 = C × RA9                  ; multiply the inverse of the divide constant
```

Temporary Storage

C, CT

Outputs

$C = [(C11) + (C2 \times P2) + ... + (C9 \times P9)]$
$RB9 = [(C1 \times P1) + (C2 \times P2) + ... + (C9 \times P9)] / K$

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**CPV**
Double-Precision	**CPVD**
Single-Precision	**CPVF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	0	0	0	1	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction to '34082

31	29						0
ID	0	0000	0000	0000	0010	001t	s000	0000

Description

A point [Xn,Yn,Zn] is compared to the volume defined by Xmin,Ymin, Zmin and Xmax,Ymax, Zmax. Six comparison bits within the status register are set according to the comparison. The TMS34020 may read the status and perform a 64-way branch based on the six comparison bits.

Implied Operands

RA0 = Xmin	RA3 = Xmax	RB0 = Xn
RA1 = Ymin	RA4 = Ymax	RB1 = Yn
RA2 = Zmin	RA5 = Zmax	RB2 = Zn

Algorithm

```
If RB0 – RA0 < 0, set XLT      ; test for XLT (Xn – Xmin)
  else reset XLT
If RA3 – RB0 < 0, set XGT      ; test for XGT (Xmax – Xn)
  else reset XGT
If RB1 – RA1 < 0, set YLT      ; test for YLT (Yn – Ymin)
  else reset YLT
If RA4 – RB1 < 0, set YGT      ; test for YGT (Ymax – Yn)
  else reset YGT
If RB2 – RA2 < 0, set ZLT      ; test for ZLT (Zn – Zmin)
  else reset ZLT
If RA5 – RB2 < 0, set ZGT      ; test for ZGT (Zmax – Zn)
  else reset ZGT
```

Temporary Storage

CT

Outputs

Status register set

Status Bits

XLT (bit 5) is set high if (Xn < Xmin)
XGT (bit 6) is set high if (Xn > Xmax)
YLT (bit 7) is set high if (Yn < Ymin)
YGT (bit 8) is set high if (Yn > Ymax)
ZLT (bit 9) is set high if (Zn < Zmin)
ZGT (bit10) is set high if (Zn > Zmax)

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**CPW**
Double-Precision	**CPWD**
Single-Precision	**CPWF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	0	0	0	0	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction to '34082

31	29				0
ID	0	0000	0000	0000	0010	000t	s000	0000

Description

A point [Xn,Yn] is compared to the window defined by Xmin, Ymin and Xmax, Ymax. Four comparison bits within the status register are set according to the comparison. The TMS34020 may read the status and perform a 16-way branch based on the four comparison bits.

Implied Operands

RA0 = Xmin RA2 = Xmax RB0 = Xn
RA1 = Ymin RA3 = Ymax RB1 = Yn

Algorithm

```
If RB0 – RA0 < 0, set XLT      ; test for XLT (Xn – Xmin)
  else reset XLT
If RA3 – RB0 < 0, set XGT      ; test for XGT (Xmax – Xn)
  else reset XGT
If RB1 – RA1 < 0, set YLT      ; test for YLT (Yn – Ymin)
  else reset YLT
If RA4 – RB1 < 0, set YGT      ; test for YGT (Ymax – Yn)
  else reset YGT
```

Temporary Storage

CT

Outputs

Status register set

Status Bits

XLT (bit 5) is set high if (Xn < Xmin)
XGT (bit 6) is set high if (Xn > Xmax)
YLT (bit 7) is set high if (Yn < Ymin)
YGT (bit 8) is set high if (Yn > Ymax)

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**CSPLN**
Double-Precision	**CSPLND**
Single-Precision	**CSPLNF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	0	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction to '34082

31	29				0
ID	0	0000 0000 0000 0011	110t	s000 0000	

Description

Given a cubic spline defined by:

$X = (A3 \times T^3) + (A2 \times T^2) + (A1 \times T) + A0$
$Y = (B3 \times T^3) + (B2 \times T^2) + (B1 \times T) + B0$
$Z = (C3 \times T^3) + (C2 \times T^2) + (C1 \times T) + C0$

This routine will calculate X,Y,Z for a series of values of T. The previous T value is incremented from 0 to 1 by an amount dT. Note this instruction may also be used to calculate X and Y for a 2-D cubic spline by ignoring the values of the Z coefficients and results.

Implied Operands

RB0 = A0, RB1 = A1, RB2 = A2, RB3 = A3
RB4 = B0, RB5 = B1, RB6 = B2, RB7 = B3
RB8 = C0, RB9 = C1, RA0 = C2, RA1 = C3
C = Previous T value (or 0 if first T value)
RA4 = dT

Algorithm

```
C = C + RA4          ; T = T + dT
RA7 = RB3            ; X = A3
RA7 = (RA7 × C) + RB2  ; X = (X × T) + A2
RA7 = (RA7 × C) + RB1  ; X = (X × T) + A1
RA7 = (RA7 × C) + RB0  ; X = (X × T) + A0
RA8 = RB7            ; Y = B3
RA8 = (RA8 × C) + RB6  ; Y = (Y × T) + B2
RA8 = (RA8 × C) + RB5  ; Y = (Y × T) + B1
RA8 = (RA8 × C) + RB4  ; Y = (Y × T) + B0
CT  = RA1 × C        ; Z = C3 × T
RA9 = CT + RA0       ; Z = Z + C2
RA9 = (RA9 × C) + RB9  ; Z = (Z × T) + C1
RA9 = (RA9 × C) + RB8  ; Z = (Z × T) + C0
```

Temporary Storage

C, CT

Outputs

RA7 = X
RA8 = Y
RA9 = Z

Instruction Type

CEXEC, short

Syntax

CVDF *CRs, CRd*

Execution

(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	1
ID			CRs				0	1	0	0	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0100		CRd		0001 1111 1000 0000	

Operands

CRs TMS34082 source register containing a 64-bit double-precision floating-point operand

CRd TMS34082 destination register

Description

CVDF converts a 64-bit IEEE double-precision floating-point number to a 32-bit IEEE single-precision floating-point number. The double-precision number resides in CRs, and the converted single-precision number is stored in CRd.

The source register, CRs, must be in the RA register file.

Instruction Type

CEXEC, short

Example

```
CVDF RA5, RA7
```

This example converts the contents of RA5 to a single-precision floating-point number and stores the result in RA7.

Syntax

CVDF *Rs_1, Rs_2, CRs, CRd*

Execution

Rs_1, $Rs_2 \rightarrow$ CRs
(CRs) $\rightarrow$ CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	1	1	1	1	1	1	0	0	R	Rs_2			
ID			CRs_1				0	1	0	0	CRd				

Instruction to '34082

31	29	28	25	24	20	21	16	15	0
ID		CRs		0100		CRd		0101 1111 1000 0000	

Operands

Rs_1 TMS34020 source register for half the 64-bit double-precision floating-point value to TMS34082.

Rs_2 TMS34020 source register for remaining half of the 64-bit double-precision floating-point operand.

CRs TMS34082 source register to contain the double-precision floating-point operand

CRd TMS34082 destination register

Description

CVDF loads the double-precision contents of Rs_1 and Rs_2 into CRs and converts the 64-bit IEEE double-precision floating-point number to a 32-bit IEEE single-precision floating-point number. The converted single-precision number is stored in CRd.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVGC, two registers

Example

```
CVDF RA5, RA7
```

This example converts the contents of RA5 to a single-precision floating-point number and stores the result in RA7.

Syntax

CVDF **Rs+, CRs, CRd*

Execution

*Rs → CRs
Rs + 32 → Rs
*Rs → CRs
Rs + 32 → Rs
(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	0	1	0
1	0	0	1	1	1	1	1	1	0	0	R	Rs			
ID			CRs$_1$				0	1	0	0	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0100		CRd		1001 1111 1000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 register to contain the 64-bit double-precision floating-point operand

CRd TMS34082 destination register

Description

CVDF loads the double-precision contents of memory pointed to by Rs into CRs and converts the 64-bit IEEE double-precision floating-point value to a 32-bit IEEE single-precision floating-point value. The double-precision number is stored in CRs, and the converted single-precision number is stored in CRd. After each load from memory, Rs is incremented by 32.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
CVDF *B5+, RA5, RA7
```

This example loads the contents of memory starting at the address given by TMS34020 register B5 into TMS34082 register RA5, converts the contents of RA5 to a single-precision number, and stores the result in RA7.

Syntax

CVDF – **Rs, CRs, CRd*

Execution

Rs – 32 → Rs
*Rs → CRs
Rs – 32 → Rs
*Rs → CRs
(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0
1	0	0	1	1	1	1	1	1	0	0	R	Rs			
ID			CRs				0	1	0	0	CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs	0100	CRd	1001 1111 1000 0000

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 register to contain the 64-bit double-precision floating-point operand

CRd TMS34082 destination register

Description

CVDF loads the double-precision contents of memory pointed to by Rs into CRs and converts the 64-bit IEEE double-precision floating-point value to a 32-bit IEEE single-precision floating-point value. The double-precision number resides in CRs, and the converted single-precision number is stored in CRd. Before each load from memory, Rs is decremented by 32.

The TMS34082 source register, CRs, must be in the RA register file.

Temporary Storage

CMOVMC, predecrement, constant cont

Example

```
CVDF -*B5, RA5, RA7
```

This example loads the contents of memory starting at the address given by TMS34020 register B5 minus 32 into TMS34082 register RA5, converts the contents of RA5 to a single-precision number, and stores the result in RA7.

Syntax

CVDI *CRs, CRd*

Execution

(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	1
ID			CRs				0	1	0	1	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0101		CRd		0001 1111 1000 0000	

Operands

CRs TMS34082 source register containing a 64-bit double-precision floating-point operand

CRd TMS34082 destination register

Description

CVDI converts a 64-bit IEEE double-precision floating-point number to a 32-bit integer number. The double-precision number resides in CRs, and the converted integer number is stored in CRd.

The source register, CRs, must be in the RA register file.

Instruction Type

CEXEC, short

Example

```
CVDI RA5, RB7
```

This example converts the contents of RA5 to an integer and stores the result in RB7.

Syntax

CVDI *Rs_1, Rs_2 CRs, CRd*

Execution

(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	1	R	Rs_1			
0	1	0	1	1	1	1	1	1	0	0	R	Rs_2			
ID			CRs				0	1	0	1	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0101		CRd		0101 1111 1000 0000	

Operands

Rs_1 TMS34020 source register for half the 64-bit double-precision floating-point value to TMS34082

Rs_2 TMS34020 source register for remaining half of the 64-bit double-precision floating-point operand

CRs TMS34082 source register to contain the double-precision floating-point operand

CRd TMS34082 destination register

Description

CVDI loads a 64-bit IEEE double-precision floating-point number from Rs_1 and Rs_2 into CRs and converts it to a 32-bit integer number. The double-precision number resides in CRs, and the converted integer number is stored in CRd.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVGC, two registers

Example

```
CVDI A4, A5, RA5, RB7
```

This example loads TMS34020 registers A4 and A5 into TMS34082 register RA5, converts the contents of RA5 to an integer, and stores the result in RB7.

Syntax

CVDI **Rs+, CRs, CRd*

Execution

*Rs → CRs
Rs + 32 → Rs
*Rs → CRs
Rs + 32 → Rs
(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	0	1	0
1	0	0	1	1	1	1	1	1	0	0	R	Rs			
ID			CRs				0	1	0	1	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0101		CRd		1001 1111 1000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 register to contain the 64-bit double-precision floating-point operand

CRd TMS34082 destination register

Description

CVDI loads the double-precision contents of memory pointed to by Rs into CRs and converts the 64-bit IEEE double-precision floating-point value to an integer value. The double-precision number resides in CRs, and the converted integer number is stored in CRd. After each load from memory, Rs is incremented by 32.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
CVDI *B5+, RA5, RA7
```

This example loads the contents of memory starting at the address given by TMS34020 register B5 into TMS34082 register RA5, converts the contents of RA5 to an integer number, and stores the result in RA7.

Syntax

CVDI *–*Rs, CRs, CRd*

Execution

Rs – 32 → Rs
*Rs → CRs
Rs – 32 → Rs
*Rs → CRs
(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0
1	0	0	1	1	1	1	1	1	0	0	R	Rs			
ID			CRs				0	1	0	1	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0101		CRd		1001 1111 1000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 register to contain the 64-bit double-precision floating-point operand

CRd TMS34082 destination register

Description

CVDI loads the double-precision contents of memory pointed to by the predecremented value of Rs into CRs and converts the 64-bit IEEE double-precision floating-point value to an integer value. The double-precision number resides in CRs, and the converted integer number is stored in CRd. Before each load from memory, Rs is decremented by 32.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVMC, predecrement, constant count

Example

```
CVDI -*B5, RA5, RA7
```

This example loads the contents of memory starting at the address given by TMS34020 register B5 minus 32 into TMS34082 register RA5, converts the contents of RA5 to an integer number, and stores the result in RA7.

Syntax

CVFD *CRs, CRd*

Execution

(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0
ID			CRs				0	1	0	0	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0100		CRd		0001 1111 0000 0000	

Operands

CRs TMS34082 source register containing a 32-bit single-precision floating-point operand

CRd TMS34082 destination register

Description

CVFD converts a 32-bit IEEE single-precision floating-point value to a 64-bit IEEE double-precision floating-point value. The single-precision number resides in CRs, and the converted double-precision number is stored in CRd.

The source register, CRs, must be in the RA register file.

Instruction Type

CEXEC, short

Example

```
CVFD RA5, RB7
```

This example converts the contents of RA5 to a double-precision number and stores the result in RB7.

Syntax

CVFD *Rs, CRs, CRd*

Execution

Rs → CRs
(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0
ID			CRs				0	1	0	0	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0100		CRd		0101 1111 0000 0000	

Operands

Rs TMS34020 source register containing the 32-bit single-precision floating-point value to TMS34082

CRs TMS34082 register to contain the 32-bit single-precision floating-point operand

CRd TMS34082 destination register

Description

CVFD loads the single-precision contents of Rs into CRs and converts the 32-bit IEEE single-precision floating-point value to a 64-bit IEEE double-precision floating-point value. The single-precision number resides in CRs, and the converted double-precision number is stored in CRd.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVGC, one register

Example

```
CVFD B5, RA5, RA7
```

This example loads TMS34020 register B5 into TMS34082 register RA5, converts the contents of RA5 to a double-precision number, and stores the result in RA7.

Syntax

CVFD **Rs+, CRs, CRd*

Execution

*Rs → CRs
Rs + 32 → Rs
(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	1
1	0	0	1	1	1	1	1	0	0	0	R	Rs			
ID			CRs				0	1	0	0	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0100		CRd		1001 1111 0000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 register to contain the 32-bit single-precision floating-point operand

CRd TMS34082 destination register

Description

CVFD loads the single-precision contents of memory pointed to by Rs into CRs and converts the 32-bit IEEE single-precision floating-point value to a 64-bit IEEE double-precision floating-point value. The single-precision number resides in CRs, and the converted double-precision number is stored in CRd. After each load from memory, Rs is incremented by 32.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
CVFD *B5+, RA5, RA7
```

This example loads the contents of memory starting at the address given by TMS34020 register B5 into TMS34082 register RA5, converts the contents of RA5 to a double-precision number, and stores the result in RA7.

Syntax

CVFD *–*Rs+, CRs, CRd*

Execution

Rs – 32 → Rs
*Rs → CRs
(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1
1	0	0	1	1	1	1	1	0	0	0	R	Rs			
ID			CRs				0	1	0	0	CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs	0100	CRd	1001 1111 0000 0000

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 register to contain the 32-bit single-precision floating-point operand

CRd TMS34082 destination register

Description

CVFD loads the single-precision contents of memory pointed to by Rs into CRs and converts the 32-bit IEEE single-precision floating-point value to a 64-bit IEEE double-precision floating-point value. The single-precision number resides in CRs, and the converted double-precision number is stored in CRd. Before each load from memory, Rs is decremented by 32.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVMC, predecrement, constant count

Example

```
CVFD -*B5, RA5, RA7
```

This example loads the contents of memory starting at the address given by TMS34020 register B5 minus 32 into TMS34082 register RA5, converts the contents of RA5 to a double-precision number, and stores the result in RA7.

Syntax

CVFI *CRs, CRd*

Execution

(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0
ID			CRs				0	1	0	1	CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs	0101	CRd	0001 1111 0000 0000

Operands

CRs TMS34082 source register containing a 32-bit single-precision floating-point operand

CRd TMS34082 destination register

Description

CVFI converts a 32-bit IEEE single-precision floating-point value to a 32-bit integer value. The single-precision number resides in CRs, and the converted integer number is stored in CRd.

The source register, CRs, must be in the RA register file.

Instruction Type

CEXEC, short

Example

```
CVFI RA5, RA7
```

This example converts the contents of RA5 to an integer and stores the result in RA7.

Syntax

CVFI *Rs, CRs, CRd*

Execution

Rs → CRs
(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0
ID			CRs				0	1	0	1	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0101		CRd		0101 1111 0000 0000	

Operands

Rs TMS34020 source register for the 32-bit single-precision floating-point value to TMS34082

CRs TMS34082 register to contain the 32-bit single-precision floating-point operand

CRd TMS34082 destination register

Description

CVFI loads the single-precision contents of Rs into CRs and converts the 32-bit IEEE single-precision floating-point value to a 32-bit integer value. The single-precision number resides in CRs, and the converted integer number is stored in CRd.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVGC, one register

Example

```
CVFI B5, RA5, RB7
```

This example loads TMS34020 register B5 into TMS34082 register RA5, converts the contents of RA5 to an integer, and stores the result in RB7.

Syntax

CVFI **Rs+, CRs, CRd*

Execution

*Rs → CRs
Rs + 32 → Rs
(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	1
1	0	0	1	1	1	1	1	0	0	0	R	Rs			
ID			CRs				0	1	0	1	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0101		CRd		1001 1111 0000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 register to contain the 32-bit single-precision floating-point operand

CRd TMS34082 destination register

Description

CVFI loads the single-precision contents of memory pointed to by Rs into CRs and converts the 32-bit IEEE single-precision floating-point value to a 32-bit integer value. The single-precision number resides in CRs, and the converted integer number is stored in CRd. After each load from memory, Rs is incremented by 32.
The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
CVFI *B5+, RA5, RA7
```

This example loads the contents of memory starting at the address given by TMS34020 register B5 into TMS34082 register RA5, converts the contents of RA5 to an integer number, and stores the result in RA7.

Syntax

CVFI – **Rs, CRs, CRd*

Execution

Rs – 32 → Rs
*Rs → CRs
(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1
1	0	0	1	1	1	1	1	0	0	0	R	Rs			
ID			CRs				0	1	0	1	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0101		CRd		1001 1111 0000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 register to contain the 32-bit single-precision floating-point operand

CRd TMS34082 destination register

Description

CVFI loads the single-precision contents of memory pointed to by Rs into CRs and converts the 32-bit IEEE single-precision floating-point value to a 32-bit integer value. The single-precision number resides in CRs, and the converted integer number resides in CRd. Before each load from memory, Rs is decremented by 32.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVMC, predecrement, constant count

Example

```
CVFI -*B5, RA5, RA7
```

This example loads the contents of memory starting at the address given by TMS34020 register B5 minus 32 into TMS34082 register RA5, converts the contents of RA5 to an integer number, and stores the result in RA7.

Syntax

CVID *CRs, CRd*

Execution

(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	1
ID			CRs				0	1	1	0	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0 1 1 0		CRd		0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0	

Operands

CRs TMS34082 source register containing the 32-bit integer operand

CRd TMS34082 destination register

Description

CVID converts a 32-bit integer value to a 64-bit IEEE double-precision floating-point value. The integer resides in CRs, and the converted double-precision number is stored in CRd.

The source register, CRs, must be in the RA register file. C and CT may not be used as operands for this instruction.

Instruction Type

CEXEC, short

Example

```
CVID RA5, RB7
```

This example converts the contents of RA5 to a double-precision number and stores the result in RB7.

Syntax

CVID *Rs, CRs, CRd*

Execution

Rs → CRs
(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs			
0	1	0	1	1	1	1	1	1	0	0	R	Rs			
ID			CRs				0	1	1	0	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0 1 1 0		CRd		0 1 0 1 1 1 1 1 1 0 0 0	0 0 0 0

Operands

Rs TMS34020 source register containing the 32-bit integer value to TMS34082

CRs TMS34082 source register to contain the 32-bit integer operand

CRd TMS34082 destination register

Description

CVID loads the integer contents of Rs into CRs and converts a 32-bit integer value to a 64-bit IEEE double-precision floating-point value. The integer resides in CRs, and the converted double-precision number is stored in CRd. For this instruction, the integer in Rs must be sent as both words of a 64-bit transfer.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVGC, two registers

Example

```
CVID B5, RA5, RA7
```

This example loads TMS34020 register B5 into TMS34082 register RA5, converts the contents of RA5 to a double-precision number, and stores the result in RA7.

Syntax

CVIF *CRs, CRd*

Execution

(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0
ID			CRs				0	1	1	0	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0 1 1 0		CRd		0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0	

Operands

CRs TMS34082 source register containing the 32-bit integer operand

CRd TMS34082 destination register

Description

CVIF converts a 32-bit integer value to a 32-bit IEEE single-precision floating-point value. The integer resides in CRs, and the converted single-precision number is stored in CRd.

The source register, CRs, must be in the RA register file. C and CT may not be used as operands for this instruction.

Instruction Type

CEXEC, short

Example

```
CVIF RA5, RA7
```

This example converts the contents of RA5 to a single-precision number and stores the result in RA7.

Syntax

CVIF *Rs, CRs, CRd*

Execution

Rs → CRs
(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0
ID			CRs				0	1	1	0	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0 1 1 0		CRd		0 1 0 1 1 1 1 1 0 0 0 0 0 0 0 0	

Operands

Rs TMS34020 source register for the 32-bit integer value to TMS34082

CRs TMS34082 source register to contain the 32-bit integer operand

CRd TMS34082 destination register

Description

CVIF loads the integer contents of Rs into CRs and converts a 32-bit integer value to a 32-bit IEEE single-precision floating-point value. The integer resides in CRs, and the converted single-precision number resides in CRd.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVGC, one register

Example

```
CVIF A3, RA5, RA7
```

This example loads TMS34020 registers of A3 into TMS34082 register RA5, converts the contents of RA5 to a single-precision number, and stores the result in RA7.

Syntax

CVIF **Rs+, CRs, CRd*

Execution

*Rs → CRs
Rs + 32 → Rs
(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	1
1	0	0	1	1	1	1	1	0	0	0	R	Rs			
ID			CRs				0	1	1	0	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0110		CRd		1001 1111 0000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 register to contain the 32-bit integer operand

CRd TMS34082 destination register

Description

CVIF loads the integer contents of memory pointed to by Rs into CRs and converts the 32-bit integer value to a 32-bit IEEE single-precision floating-point value. The integer number resides in CRs, and the converted single-precision number is stored in CRd. After each load from memory, Rs is incremented by 32.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
CVIF *B5+, RA5, RA7
```

This example loads the contents of memory starting at the address given by TMS34020 register B5 into TMS34082 register RA5, converts the contents of RA5 to a single-precision number, and stores the result in RA7.

Syntax

CVIF – *****Rs, CRs, CRd*

Execution

Rs – 32 → Rs
*Rs → CRs
(CRs) → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1
1	0	0	1	1	1	1	1	0	0	0	R	Rs			
ID			CRs				0	1	1	0	CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs	0110	CRd	1001 1111 0000 0000

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 register to contain the 32-bit integer operand

CRd TMS34082 destination register

Description

CVIF loads the integer contents of memory pointed to by Rs into CRs and converts the 32-bit integer value to a 32-bit IEEE single-precision floating-point value. The integer number resides in CRs, and the converted single-precision number is stored in CRd. Before each load from memory, Rs is decremented by 32.

The TMS34082 source register, CRs, must be in the RA register file.

Instruction Type

CMOVMC, predecrement, constant count

Example

```
CVIF -*B5, RA5, RA7
```

This example loads the contents of memory starting at the address given by TMS34020 register B5 minus 32 into TMS34082 register RA5, converts the contents of RA5 to a single-precision number, and stores the result in RA7.

Syntax

Type	Syntax
Integer	**DEC** *CRs [, CRd]*
Double-Precision	**DECD** *CRs [, CRd]*
Single-Precision	**DECF** *CRs [, CRd]*

Execution

CRs – 1 → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	1	1	type	size
ID			CRs				1	1	0	1	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		1101		CRd		0000 001t s000 0000	

Operands

CRs TMS34082 source register (also destination register if CRd is not specified). Must be from RA register file.

CRd TMS34082 destination register.

Description

DECx subtracts one (of the appropriate type) from the value in CRs and stores the result in CRd. If CRd is not specified, the result is stored in CRs.

Instruction Type

CEXEC, short

Example

```
DEC CT
```

This example subtracts an integer one from the value in TMS34082 register CT and stores the result in CT.

Syntax

Type	Syntax
Integer	**DEC** *CRs [, CRd]*
Double-Precision	**DECD** *CRs [, CRd]*
Single-Precision	**DECF** *CRs [, CRd]*

Execution

CRs – 1 → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	1	1	type	size
ID			CRs				1	1	0	1	CRd				

Instruction to '34082

31 – 29	28 – 25	24 – 21	20 – 16	15 – 0
ID	CRs	1101	CRd	0000 011t s000 0000

Operands

CRs TMS34082 source register (also destination register if CRd is not specified). Must be from RB register file.

CRd TMS34082 destination register.

Description

DECx subtracts one (of the appropriate type) from the value in CRs and stores the result in CRd. If CRd is not specified, the result is stored in CRs.

Instruction Type

CEXEC, short

Example

```
DECF RB2, C
```

This example subtracts a single-precision one from the value in TMS34082 register RB2 and stores the result in the C register.

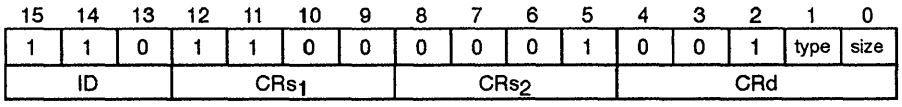
Syntax

Type	Syntax
Integer	**DIVS** *CRs$_1$, CRs$_2$, CRd*
Double-Precision	**DIVD** *CRs$_1$, CRs$_2$, CRd*
Single-Precision	**DIVF** *CRs$_1$, CRs$_2$, CRd*

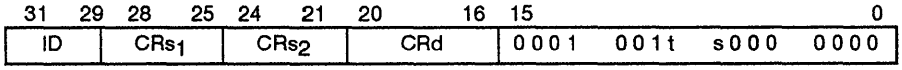
Execution

$$\left(\frac{CRs_1}{CRs_2}\right) \rightarrow CRd$$

'34020 Instruction Words



Instruction to '34082



Operands

CRs_1 TMS34082 register containing the first operand. Must be in RA register file.

CRs_2 TMS34082 register containing the second operand. Must be in RB register file.

CRd TMS34082 destination register

Description

DIVx divides the contents of CRs_1 by CRs_2 and stores the result in CRd. For integer divides, the CT register is used for temporary storage. Any value stored in this register prior to DIVS will be corrupted.

C and CT may not be used as operands for the integer form of this instruction, DIVS.

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**DIVS** *Rs_1, Rs_2, CRs_1, CRs_2, CRd*
Single-Precision	**DIVF** *Rs_1, Rs_2, CRs_1, CRs_2, CRd*

Execution

$Rs_1 \rightarrow CRs_1$

$Rs_2 \rightarrow CRs_2$

$$\left(\frac{CRs_1}{CRs_2}\right)^{100} \rightarrow CRd$$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	1	0	0	1	type	0	0	0	R	Rs_2			
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs_1		CRs_2		CRd		0101 001t 0000 0000	

Operands

Rs_1 TMS34020 source register for the first value to TMS34082

Rs_2 TMS34020 source register for the second value to TMS34082

CRs_1 TMS34082 register to contain the first operand. Must be in RA register file.

CRs_2 TMS34082 register to contain the second operand. Must be in RB register file.

CRd TMS34082 destination register

Description

DIVx loads the contents of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, divides the contents of CRs_1 by CRs_2, and stores the result in CRd. For integer divides, the CT register is used for temporary storage. Any value stored in this register prior to DIVS will be corrupted.

The double-precision form of this instruction is not supported.

Instruction Type

CMOVGC, two registers

Example

```
DIVF A5, A6, RA5, RB6, RA7
```

This example loads TMS34020 registers A5 and A6 into TMS34082 registers RA5 and RB6 respectively, divides the contents of RA5 by RB6, and stores the result in RA7.

Syntax

Type	Syntax
Integer	**DIVS** **Rs+, CRs$_1$, CRs$_2$, CRd*
Double-Precision	**DIVD** **Rs+, CRs$_1$, CRs$_2$, CRd*
Single-Precision	**DIVF** **Rs+, CRs$_1$, CRs$_2$, CRd*

Execution

$*Rs \rightarrow CRs_1$
$Rs + 32 \rightarrow Rs$
$*Rs \rightarrow CRs_1$
$Rs + 32 \rightarrow Rs$
$*Rs \rightarrow CRs_2$
$Rs + 32 \rightarrow Rs$
$*Rs \rightarrow CRs_2$
$Rs + 32 \rightarrow Rs$

$$\left(\frac{CRs_1}{CRs_2}\right) \rightarrow CRd$$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	transfers		
1	0	0	1	0	0	1	t	s	0	0	R	Rs			
ID			CRs$_1$				CRs$_2$				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs$_1$		CRs$_2$		CRd		1001 001t s000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs_1 TMS34082 register to contain the first operand. Must be in RA register file.

CRs_2 TMS34082 register to contain the second operand. Must be in RB register file.

CRd TMS34082 destination register

Description

DIVx loads the contents of memory pointed to by Rs into CRs_1 and CRs_2, divides the contents of CRs_1 by CRs_2, and stores the result in CRd. After each load from memory, Rs is incremented by 32. For integer divides, the CT register is used for temporary storage. Any value stored in this register prior to DIVS will be corrupted.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
DIVS *A5+, RA5, RB6, RA7
```

This example loads the contents of memory starting at the address given in TMS34020 register A5 into TMS34082 registers RA5 and RB6, divides the contents of RA5 by RB6, and stores the result in RA7.

Syntax

Type	Syntax
Integer	**DIVS** – **Rs, CRs$_1$, CRs$_2$, CRd*
Double-Precision	**DIVD** – **Rs, CRs$_1$, CRs$_2$, CRd*
Single-Precision	**DIVF** – **Rs, CRs$_1$, CRs$_2$, CRd*

Execution

Rs – 32 → Rs
*Rs → CRs$_1$
~~Rs – 32 → Rs~~
~~*Rs → CRs$_1$~~
Rs – 32 → Rs
*Rs → CRs$_2$
~~Rs – 32 → Rs~~
~~*Rs → CRs$_2$~~

$$\left(\frac{CRs_1}{CRs_2}\right) \rightarrow CRd$$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	transfers		
1	0	0	1	0	0	1	type	size	0	0	R	Rs			
ID			CRs$_1$				CRs$_2$				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs$_1$		CRs$_2$		CRd		1001 001t	s000 0000

Operands

Rs TMS34020 register containing the memory address

CRs$_1$ TMS34082 register to contain the first operand. Must be in RA register file.

CRs$_2$ TMS34082 register to contain the second operand. Must be in RB register file.

CRd TMS34082 destination register

Description

DIVx loads the contents of memory pointed to by Rs into CRs$_1$ and CRs$_2$, divides the contents of CRs$_1$ by CRs$_2$, and stores the result in CRd. Before each load from memory, Rs is decremented by 32. For integer divides, the CT register is used for temporary storage. Any value stored in this register prior to DIVS will be corrupted.

Instruction Type

CMOVMC, predecrement

Example

```
DIVF -*A5, RA5, RB6, RA7
```

This example loads the single-precision floating-point contents of memory starting at the address given in TMS34020 register A5 minus 32 into TMS34082 registers RA5 and RB6, divides the single-precision floating-point contents of RA5 by RB6, and stores the result in RA7.

Syntax

GETCST

Execution

TMS34082 Status Register → ST register of TMS34020

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	0
0	1	0	0	1	1	1	0	0	0	0	0	0	0	0	1
ID			0	0	0	0	0	0	0	0	0	1	1	0	0

Instruction to '34082

31	29														0
ID	0	0000		0000		0000		0100		1110		0000		0000	

Description

GETCST loads 4 MSBs of the TMS34082 status register (STATUS) into the TMS34020 status register (ST).

Instruction Type

CMOVCS

Example

```
GETCST
```

This example sends the TMS34082 status register to the TMS34020. The TMS34020 takes the value and masks off the 4 MSBs; it then stuffs the values in the TMS34020 status register corresponding to the N, C, Z, V bits.

Syntax

Type	Syntax
Integer	**INC** *CRs [, CRd]*
Double-Precision	**INCD** *CRs [, CRd]*
Single-Precision	**INCF** *CRs [, CRd]*

Execution

1 + CRs → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	0	0	type	size
ID			CRs				1	1	0	1	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		1101		CRd		0000 000t s000 0000	

Operands

CRs TMS34082 source register. (Also destination register if CRd is not specified.)

CRd TMS34082 destination register.

Description

INCx adds one (of the appropriate type) to the value in RA register CRs and stores the result in CRd. If CRd is not specified, the result is stored in CRs.

Instruction Type

CEXEC, short

Example

```
INC RA0
```

This example adds an integer one to the value in TMS34082 register RA0 and stores the result in RA0.

Syntax

Type	Syntax
Integer	**INC** *CRs [, CRd]*
Double-Precision	**INCD** *CRs [, CRd]*
Single-Precision	**INCF** *CRs [, CRd]*

Execution

1 + CRs → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	0	0	type	size
ID			CRs				1	1	0	1	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		1101		CRd		0000 000t s000 0000	

Operands

CRs TMS34082 source register. (Also destination register if CRd is not specified.)

CRd TMS34082 destination register.

Description

INCx adds one (of the appropriate type) to the value in RB register CRs and stores the result in CRd. If CRd is not specified, the result is stored in CRs.

Instruction Type

CEXEC, short

Example

```
INCD RB1, RA7
```

This example adds a double-precision one to the value in TMS34082 register RB1 and stores the result in RA7.

Syntax

INMNMX

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	0	0	1	1	0	0
ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction to '34082

31	29						0
ID	0	0000	0000	0000	0010	0110	0000	0000

Description

The IMNMX instruction configures the registers in preparation for either the MNMX1 or MNMX2 instruction. The following initializations occur (internal flags are set; register values are not altered):

Algorithm

```
RB0 = MAX      ; set to positive infinity (used to store minimum X values)
RB1 = MIN      ; set to negative infinity (used to store maximum X values)
RB2 = MAX      ; set to positive infinity (used to store minimum Y values)
RB3 = MIN      ; set to negative infinity (used to store maximum Y values)
COUNTX = 0     ; bits 15-0 for X minimums, bits 31-16 for X maximums
COUNTY = 0     ; bits 15-0 for Y minimums, bits 31-16 for Y maximums
Count = 0      ; set count to zero (bits 31-16 of MIN-MAX/LOOPCT register
```

Instruction Type

CEXEC, short

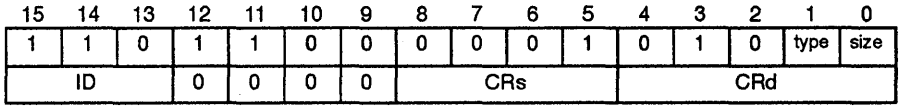
Syntax

Type	Syntax
Integer	**INV** *CRs, CRd*
Double-Precision	**INVD** *CRs, CRd*
Single-Precision	**INVF** *CRs, CRd*

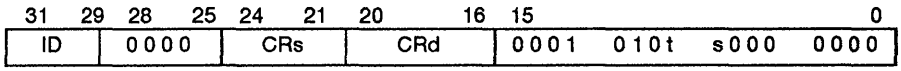
Execution

$$\frac{1}{CRs} \rightarrow CRd$$

'34020 Instruction Words



Instruction to '34082



Operands

CRs TMS34082 source register containing the operand. Must be from the RB register file.

CRd TMS34082 destination register

Description

This instruction divides 1 by CRs, and places the result in CRd. For integer instructions, CT is used as a temporary register. Any value stored in CT prior to INV will be corrupted.

C and CT may not be used as operands for the integer form if this instruction, INV.

Instruction Type

CEXEC, short

Example

```
INV RB9, RA7
```

This example divides 1 by the contents of RB9 and stores the result in RA7.

Syntax

Type	Syntax
Integer	**INV** *Rs_1, CRs, CRd*
Double-Precision	**INVD** *Rs_1, Rs_2, CRs, CRd*
Single-Precision	**IINVF** *Rs_1, CRs, CRd*

Execution

$Rs_1 \rightarrow CRs$

$Rs_2 \rightarrow CRs$

$\frac{1}{CRs} \rightarrow CRd$

'34020 Instruction Words

Integer or Single-Precision:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs_1			
0	1	0	1	0	1	0	type	0	0	0	0	0	0	0	0
ID			0	0	0	0	CRs				CRd				

Double-Precision:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	1	0	1	0	1	1	0	0	R	Rs_2			
ID			0	0	0	0	CRs				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15							0
ID		0000		CRs		CRd		0101			010t		s000		0000

Operands

Rs_1 TMS34020 source register containing the operand (or half of the 64-bit double-precision floating-point operand.)

Rs_2 TMS34020 source register containing the remaining half of the double-precision operand.

CRs TMS34082 register to contain the operand. Must be in the RB register file.

CRd TMS34082 destination register

Description

This instruction loads the contents of the Rs_1 (and Rs_2 for double-precision) into CRs, divides 1 by CRs, and places the result in CRd. For integer inverses, CT is used as a temporary storage register. Any value stored in CT prior to INV will be corrupted.

Instruction Type

CMOVGC, one or two registers

Example

```
INV A2, RB8, RB2
```

This example loads the contents of TMS34020 register A2 into RB8, divides 1 by RB8, and stores the integer result in RB2.

Syntax

Type	Syntax
Integer	**INV** **Rs+, CRs, CRd*
Double-Precision	**INVD** **Rs+, CRs, CRd*
Single-Precision	**INVF** **Rs+, CRs, CRd*

Execution

*Rs → CRs
Rs + 32 → Rs
*Rs → CRs
Rs + 32 → Rs
$\frac{1}{CRs}$ → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	0	transfers	
1	0	0	1	0	1	0	type	size	0	0	R	Rs			
ID			0	0	0	0	CRs				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		0000		CRs		CRd		1001 010t s000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 register to contain the operand. Must be in the RB register file.

CRd TMS34082 destination register

Description

This instruction loads the contents of memory pointed to by Rs into CRs, divides 1 by CRs, and places the result in CRd. After each load from memory, Rs is incremented by 32. For integer inverses, CT is used as a temporary storage register. Any value stored in CT prior to INV will be corrupted.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
INVD *A2+, RB8, RB1
```

This example loads the double-precision contents of memory starting at the address given by TMS34020 register A2 into TMS34082 register RB8, divides 1 by RB8, and stores the result in RB1.

Syntax

Type	Syntax
Integer	**INV** *– *Rs, CRs, CRd*
Double-Precision	**INVD** *– *Rs, CRs, CRd*
Single-Precision	**INVF** *– *Rs, CRs, CRd*

Execution

Rs – 32 → Rs

*Rs → CRs

Rs – 32 → Rs

*Rs → CRs

$$\frac{1}{CRs} \rightarrow CRd$$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	transfers	
1	0	0	1	0	1	0	type	size	0	0	R	Rs			
ID			0	0	0	0	CRs				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		0000		CRs		CRd		1001 010t s000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 register to contain the operand. Must be from the RB register file.

CRd TMS34082 destination register

Description

This instruction loads the contents of memory pointed to by Rs into CRs, divides 1 by CRs, and places the result in CRd. Before each load from memory, Rs is decremented by 32. For integer inverses, CT is used as a temporary storage register. Any value stored in CT prior to INV will be corrupted.

Instruction Type

CMOVMC, predecrement, constant count

Example

```
INVF -*A2, RB8, RB1
```

This example loads the single-precision contents of memory at the address given by TMS34020 register A2 minus 32 into TMS34082 register RB8, divides 1 by RB8, and stores the result in RB1.

Syntax

JUMPC *n*

Execution

Execute external TMS34082 instructions found at address 2 × *n*

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
1	1	*n*					0	0	0	0	0	0	0	0	0
ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction to '34082

31 29	28 25	24 21	20 16	15 14	13 9	8 4	3 0
ID	0000	0000	0000	11	n	00000	0000

Operands

n Specifies the jump table entry to which the TMS34082 instruction execution is sent May be a number from 0 to 15.

Description

JUMPC begins execution of TMS34082 external instructions stored in TMS34082 external local memory. The starting address is specified as TMS34082 external memory address 2 × *n*. Usually, a jump table is stored in these locations to permit calling several complex subroutines.

Instruction Type

CEXEC, long

Example

JUMPC 4

This example executes TMS34082 instructions stored in the TMS34082's local memory on the MSD bus. Instruction execution begins at address 8.

Syntax

Type	Syntax
Integer	**LINTX**
Double-Precision	**LINTXD**
Single-Precision	**LINTXF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	0	1	0	0	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction to '34082

31	29							0
ID	0	0000	0000	0000	0010	100t	s000	0000

Description

Perform linear interpolation given two points and a plane (the plane is assumed perpendicular to one of the coordinate axes).

NOTE: If the Z1 and Z2 values are ignored, this will perform the equivalent of a 2-D linear interpolation.

Implied Operands

RA0 = X1 RB0 = X2
RA1 = Y1 RB1 = Y2
RA2 = Z1 RB2 = Z2
RB9 = X3

Algorithm

```
RA3 = RB9 – RA0          ; X3 – X1
RB6 = RB0 – RA0          ; X2 – X1
RB7 = RB1 – RA1          ; Y2 – Y1
RB8 = RB2 – RA2          ; Z2 – Z1
C = RA3/RB6              ; t = (X3 – X1)/(X2 – X1)
RB6 = C × RB6            ; t × (X2 – X1)
RB7 = C × RB7            ; t × (Y2 – Y1)
RB8 = C × RB8            ; t × (Z2 – Z1)
RA0 = RB6 + RA0          ; X3 = X1 + (t × (X2 – X1))
RA1 = RB7 + RA1          ; Y3 = Y1 + (t × (Y2 – Y1))
RA2 = RB8 + RA2          ; Z3 = Z1 + (t × (Z2 – Z1))
```

Temporary Storage

C, RA3, RB8-RB6

Outputs

```
RA0 = X3                 ; interpolated values
RA1 = Y3
RA2 = Z3
```

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**LINTY**
Double-Precision	**LINTYD**
Single-Precision	**LINTYF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	0	1	0	0	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	1

Instruction to '34082

31	29						0
ID	0	0000	0000	0001	0010	100t	s000	0000

Description

Perform linear interpolation given two points and a plane (the plane is assumed perpendicular to one of the coordinate axes).

NOTE: If the Z1 and Z2 values are ignored, this will perform the equivalent of a 2-D linear interpolation.

Implied Operands

RA0 = X1 RB0 = X2
RA1 = Y1 RB1 = Y2
RA2 = Z1 RB2 = Z2
RB9 = Y3

Algorithm

```
RA3 = RB9 – RA1          ; Y3 – Y1
RB6 = RB0 – RA0          ; X2 – X1
RB7 = RB1 – RA1          ; Y2 – Y1
RB8 = RB2 – RA2          ; Z2 – Z1
C = RA3/RB7              ; t = (Y3 – Y1) / (Y2 – Y1)
RB6 = C × RB6            ; t × (X2 – X1)
RB7 = C × RB7            ; t × (Y2 – Y1)
RB8 = C × RB8            ; t × (Z2 – Z1)
RA0 = RB6 + RA0          ; X3 = X1 + (t × (X2 – X1))
RA1 = RB7 + RA1          ; Y3 = Y1 + (t × (Y2 – Y1))
RA2 = RB8 + RA2          ; Z3 = Z1 + (t × (Z2 – Z1))
```

Temporary Storage

C, RA3, RB8-RB6

Outputs

```
RA0 = X3                 ; interpolated values
RA1 = Y3
RA2 = Z3
```

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**LINTZ**
Double-Precision	**LINTZD**
Single-Precision	**LINTZF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	0	1	0	0	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	1	0

Instruction to '34082

31	29						0
ID	0	0000	0000	0010	0010	100t	s000	0000

Description

Perform linear interpolation given two points and a plane (the plane is assumed perpendicular to one of the coordinate axes).

Implied Operands

RA0 = X1 RB0 = X2
RA1 = Y1 RB1 = Y2
RA2 = Z1 RB2 = Z2
RB9 = Z3

Algorithm

```
RA3 = RB9 – RA2          ; Z3 – Z1
RB6 = RB0 – RA0          ; X2 – X1
RB7 = RB1 – RA1          ; Y2 – Y1
RB8 = RB2 – RA2          ; Z2 – Z1
C = RA3/RB8              ; t = (Z3 – Z1) / (Z2 – Z1)
RB6 = C × RB6            ; t × (X2 – X1)
RB7 = C × RB7            ; t × (Y2 – Y1)
RB8 = C × RB8            ; t × (Z2 – Z1)
RA0 = RB6 + RA0          ; X3 = X1 + (t × (X2 – X1))
RA1 = RB8 + RA1          ; Y3 = Y1 + (t × Y2 – Y1))
RA2 = RB8 + RA2          ; Z3 = Z1 + (t × (Z2 – Z1))
```

Temporary Storage

C, RA3, RB8-RB6

Outputs

```
RA0 = X3                 ; interpolated values
RA1 = Y3
RA2 = Z3
```

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**MAC** CRs_1, CRs_2
Double-Precision	**MACD** CRs_1, CRs_2
Single-Precision	**MACF** CRs_1, CRs_2

Execution

$C + (CRs_1 \times CRs_2) \rightarrow C$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	1	type	size
ID			CRs_1				0	0	0	1	1	CRs_2			

Instruction to '34082

31	29	28	25	24	20	19	16	15	0
ID		CRs_1		00011		CRs_2		0011 111t s000 0000	

Operands

CRs_1 TMS34082 register containing an A_n operand. Must be in the RA register file.

CRs_2 TMS34082 register containing a B_n operand. Must be in the RB register file.

Implied Operands

C Register Previously accumulated sum

Description

MACx is used to perform multiply and accumulate operations of the form:

$$((A_0 \times B_0) + (A_1 \times_1) + (A_2 \times B_2) + \ldots (A_n \times B_n)).$$

The MACx instruction performs one multiply and adds the result to the previously accumulated sum.

Outputs

The new accumulated sum is stored in the C Register. The next multiply/accumulate may now be performed.

Instruction Type

CEXEC, short

Example

```
CLRD C
MACD RA0, RB0
MACD RA1, RB1
MACD RA2, RB2
```

This example performs a sum of three products. First, the C register is set to zero. Then, the double-precision contents of RA0 and RB0 are multiplied. The next instruction multiplies RA1 by RB1 and adds this product to the previous result, storing the sum in the C register. The next instruction multiplies RA2 by RB2 and adds the product to the value in C. The sum of products is stored in C.

Syntax

Type	Syntax
Integer	**MAC** Rs_1, Rs_2, CRs_1, CRs_2
Single-Precision	**MACF** *Rs_1, Rs_2, CRs_1, CRs_2*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
$C + (CRs_1 \times CRs_2) \rightarrow C$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3 2 1 0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1
0	1	1	1	1	1	1	type	0	0	0	R	Rs_2
ID			CRs_1				0	0	0	1	1	CRs_2

Instruction to '34082

31 29	28 25	24 20	19 16	15 0
ID	CRs_1	00011	CRs_2	0111 111t 0000 0000

Operands

Rs_1 TMS34020 source register for the first (An) value to TMS34082

Rs_2 TMS34020 source register for the second (Bn) value to TMS34082

CRs_1 TMS34082 register to contain the A_n operand. Must be in the RA register file.

CRs_2 TMS34082 register to contain the B_n operand. Must be in the RB register file.

Implied Operands

C Register Previously accumulated sum

Description

MACx is used to perform multiply and accumulate operations of the form:

$$((A_0 \times B_0) + (A_1 \times B_1) + (A_2 \times B_2) + \ldots (A_n \times B_n)).$$

This instruction loads two operands from Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, performs one multiply, and adds the result to the previously accumulated sum.

The double-precision form of this instruction is not supported.

Outputs

The new accumulated sum is stored in the C Register. The next multiply/accumulate may now be performed.

Instruction Type

CMOVGC, two registers

Example

```
MAC A1, A2, RA1, RB1
```

This instruction loads the integer contents of A1 and A2 into RA1 and RB1, respectively, and multiples the contents of RA1 by RB1. The product is added to the value stored in the C register and the result is stored back in C.

Syntax

Type	Syntax
Integer	**MAC** **Rs+, CRs$_1$, CRs$_2$ [, count]*
Single-Precision	**MACF** **Rs+, CRs$_1$, CRs$_2$ [, count]*

Execution

Repeat *count* times:

$*Rs \rightarrow CRs_1$

$Rs + 32 \rightarrow Rs$

$*Rs \rightarrow CRs_2$

$C + (CRs_1 \times CRs_2) \rightarrow C$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	transfers				
1	0	1	1	1	1	1	type	0	0	0	R	Rs			
ID			CRs$_1$				0	0	0	1	1	CRs$_2$			

Instruction to '34082

31	29	28	25	24	20	19	16	15	0
ID		CRs$_1$		00011		CRs$_2$		1011 111t 0000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs_1 TMS34082 register to contain the A_n operand. Must be in the RA register file.

CRs_2 TMS34082 register to contain the B_n operand. Must be in the RB register file.

count Number of times the instruction is executed; must be between 1-16 (default is 1). The number of transfers is $2 \times$ *count*

implied Operands

C Register Previously accumulated sum

Description

MACx is used to perform multiply and accumulate operations of the form:

$$((A_0 \times B_0) + (A_1 \times {}_1) + (A_2 \times B_2) + \ldots (A_n \times B_n)).$$

This instruction loads two operands from memory starting at the address given by TMS34020 register Rs into TMS34082 registers CRs_1 and CRs_2, performs one multiply, and adds the result to the previously accumulated sum. This sequence is repeated *count* times. After each load from memory, Rs is incremented by 32.

The double-precision form of this instruction is not supported.

Outputs

The new accumulated sum is stored in the C register. The next multiply/accumulate may now be performed.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
CLRF C
MACF *A1+, RA9, RB9, 6
```

This example performs a sum of six products. First, the TMS34082 C register is set to zero. Then, the single-precision contents of memory starting at TMS34020 register A1 is loaded into TMS34082 registers RA9 and RB9. The contents of RA9 and RB9 are multiplied, the result is added to the C register, and the sum is stored in C. This process is repeated 5 more times. The end result is stored in C.

Syntax

Type	Syntax
Integer	**MAC** *–*Rs, CRs_1, CRs_2 [, count]*
Single-Precision	**MACF** *–*Rs, CRs_1, CRs_2 [, count]*

Execution

Repeat *count* times:

Rs – 32 → Rs
*Rs → CRs_1
Rs – 32 → Rs
*Rs → CRs_2
C + (CRs_1 × CRs_2) → C

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	transfers				
1	0	1	1	1	1	1	type	0	0	0	R	Rs			
ID			CRs_1				0	0	0	1	1	CRs_2			

Instruction to '34082

31	29	28	25	24	20	19	16	15	0
ID		CRs_1		00011		CRs_2		1011 111t 0000 0000	

Operands

Rs — TMS34020 register containing the memory address

CRs_1 — TMS34082 register to contain the A_n operand. Must be in the RA register file.

CRs_2 — TMS34082 register to contain the B_n operand. Must be in the RB register file.

count — Number of times the instruction is executed; must be between 1-16 (default is 1). The number of transfers is 2 × *count*

Implied Operands

C Register — Previously accumulated sum

Description

MACx is used to perform multiply and accumulate operations of the form:

$$((A_0 \times B_0) + (A_1 \times B_1) + (A_2 \times B_2) + \ldots (A_n \times B_n)).$$

This instruction loads two operands from memory starting at the address given by TMS34020 register Rs (minus 32) into TMS34082 registers CRs_1 and CRs_2 respectively, performs one multiply, and adds the result to the previously accumulated sum. This sequence is repeated *count* times. Before each load from memory, Rs is decremented by 32.

The double-precision form of this instruction is not supported.

Outputs

The new accumulated sum is stored in the C register. The next multiply/accumulate may now be performed.

Instruction Type

CMOVMC, predecrement, constant count

Syntax

Type	Syntax
Integer	**MADD**
Double-Precision	**MADDD**
Single-Precision	**MADDF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	1	type	size
ID			0	0	0	0	1	0	0	0	0	0	0	0	0

Instruction to '34082

31	29							0
ID	0	0001	0000	0000	0011	111t	s000	0000

Description

This instruction is used with the matrix multiply instructions (MMPY0, MMPY1, and MMPY2) to expedite the multiplication of a 3 × 4 matrix by a vector where the fourth element of the vector is an implied 1.

Implied Operands

A 4 × 4 matrix in FPU registers

RA0 = B00	RA1 = B01	RA2 = B02	RA3 = B03
RA4 = B10	RA5 = B11	RA6 = B12	RA7 = B13
RA8 = B20	RA9 = B21	RB0 = B22	RB1 = B23
RB2 = B30	RB3 = B31	RB4 = B32	RB5 = B33

The accumulated sums from MMPY0, MMPY1, and MMPY2

RB6 = (A00 × B00) + (A01 × B10) + (A02 × B20)
RB7 = (A00 × B01) + (A01 × B11) + (A02 × B21)
RB8 = (A00 × B02) + (A01 × B12) + (A02 × B22)
RB9 = (A00 × B03) + (A01 × B13) + (A02 × B23)

Algorithm

RB6 = RB6 + RB2
RB7 = RB7 + RB3
RB8 = RB8 + RB4
RB9 = RB9 + RB5

Temporary Storage

CT

Outputs

The resulting vector is stored in FPU registers.

RB6 = (A00 × B00) + (A01 × B10) + (A02 × B20) + B30
RB7 = (A00 × B01) + (A01 × B11) + (A02 × B21) + B31
RB8 = (A00 × B02) + (A01 × B12) + (A02 × B22) + B32
RB9 = (A00 × B03) + (A01 × B13) + (A02 × B23) + B33

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**MMPY0**
Double-Precision	**MMPY0D**
Single-Precision	**MMPY0F**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	1	type	size
ID			0	0	0	0	0	1	0	0	0	0	0	0	0

Instruction to '34082

31	29							0
ID	0	0000	1000	0000	0011	111t	s000	0000

Description

This instruction multiplies the matrix B by a vector element, A0. This instruction may be combined with MMPY1, MMPY2, and MMPY3 to multiply matrices of several sizes.1 × 4 by 4 × 4, 4 × 4 by 4 × 4, 1 × 3 by 3 × 3, and 3 × 3 by 3 × 3 matrix multiplies may be implemented.

Implied Operands

A 4 × 4 matrix in the FPU registers

RA0 = B00	RA1 = B01	RA2 = B02	RA3 = B03
RA4 = B10	RA5 = B11	RA6 = B12	RA7 = B13
RA8 = B20	RA9 = B21	RB0 = B22	RB1 = B23
RB2 = B30	RB3 = B31	RB4 = B32	RB5 = B33

The first element (Ax0) of a row vector: RB9 = Ax0

Algorithm

```
RB6 = RB9 × RA0     ; Ax0 × B00
RB7 = RB9 × RA1     ; Ax0 × B01
RB8 = RB9 × RA2     ; Ax0 × B02
RB9 = RB9 × RA3     ; Ax0 × B03
CT= RB9             ; CT is used to store (Ax0 × B03) value
                    ; since RB9 will be corrupted.
```

Temporary Storage

C

Outputs

RB6 = Ax0 × B00
RB7 = Ax0 × B01
RB8 = Ax0 × B02
RB9 = CT = Ax0 × B03

Instruction Type

CEXEC, short

Example

See Example 5–4 for code for a 3 × 3 by 3 × 3 matrix multiply.

Syntax

Type	Syntax
Integer	**MMPY1**
Double-Precision	**MMPY1D**
Single-Precision	**MMPY1F**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	1	type	size
ID			0	0	0	0	0	1	0	1	0	0	0	0	0

Instruction to '34082

31	29							0
ID	0	0000	1010	0000	0011	111t	s000	0000

Description

This instruction multiplies the matrix B by an vector element, A1. This instruction may be combined with MMPY0, MMPY2, and MMPY3 to multiply matrices of several sizes. 1×4 by 4×4, 4×4 by 4×4, 1×3 by 3×3, and 3×3 by 3×3 matrix multiplies may be implemented.

Implied Operands

A 4×4 matrix in the FPU registers:

RA0 = B00	RA1 = B01	RA2 = B02	RA3 = B03
RA4 = B10	RA5 = B11	RA6 = B12	RA7 = B13
RA8 = B20	RA9 = B21	RB0 = B22	RB1 = B23
RB2 = B30	RB3 = B31	RB4 = B32	RB5 = B33

The initial products from MMPY0 for the resulting matrix row:

RB6 = Ax0 × B00
RB7 = Ax0 × B01
RB8 = Ax0 × B02
CT = Ax0 × B03

The second element (Ax1) of a row vector: RB9 = Ax1

Algorithm

```
RB6 = RB6 + (RB9 × RA4)    ; (Ax0 × B00) + (Ax1 × B10)
RB7 = RB7 + (RB9 × RA5)    ; (Ax0 × B01) + (Ax1 × B11)
RB8 = RB8 + (RB9 × RA6)    ; (Ax0 × B02) + (Ax1 × B12)
RB9 = CT + (RB9 × RA7)     ; (Ax0 × B03) + (Ax1 × B13)
CT = RB9                   ; CT is used to store the fourth value since
                           ; RB9 will be corrupted.
```

Temporary Storage

C

Inputs

RB6 = (Ax0 × B00) + (Ax1 × B10)
RB7 = (Ax0 × B01) + (Ax1 × B11)
RB8 = (Ax0 × B02) + (Ax1 × 12)
RB9 = CT = (Ax0 × B03) + (Ax1 × B13)

Instruction Type

CEXEC, short

Example

See Example 5–4 for code for a 3×3 by 3×3 matrix multiply.

Syntax

Type	Syntax
Integer	**MMPY2**
Double-Precision	**MMPY2D**
Single-Precision	**MMPY2F**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	1	type	size
ID			0	0	0	0	0	1	1	0	0	0	0	0	0

Instruction to '34082

31	29								0
ID	0	0000	1100	0000	0011	111t	s000	0000	

Description

This instruction multiplies the matrix B by a vector element, A2. This instruction may be combined with MMPY0, MMPY1, and MMPY3 to multiply matrices of several sizes. 1 × 4 by 4 × 4, 4 × 4 by 4 × 4, 1 × 3 by 3 × 3, and 3 × 3 by 3 × 3 matrix multiplies may be implemented.

Implied Operands

A 4 × 4 matrix in the FPU registers:

RA0 = B00	RA1 = B01	RA2 = B02	RA3 = B03
RA4 = B10	RA5 = B11	RA6 = B12	RA7 = B13
RA8 = B20	RA9 = B21	RB0 = B22	RB1 = B23
RB2 = B30	RB3 = B31	RB4 = B32	RB5 = B33

The accumulated sums from MMPY0 and MMPY1 for the resulting matrix:

RB6 = (Ax0 × B00) + (Ax1 × B10)
RB7 = (Ax0 × B01) + (Ax1 × B11)
RB8 = (Ax0 × B02) + (Ax1 × B12)
CT = (Ax0 × B03) + (Ax1 × B13)

The third element (Ax2) of a row vector: RB9 = Ax2

Algorithm

```
RB6 = RB6 + (C × RA8)   ; (Ax0 × B00 + Ax1 × B10) + (Ax2 × B20)
RB7 = RB7 + (C × RA9)   ; (Ax0 × B01 + Ax1 × B11) + (Ax2 × B21)
RB8 = RB8 + (C × RB0)   ; (Ax0 × B02 + Ax1 × B12) + (Ax2 × B22)
RB9 = CT + (C × RB1)    ; (Ax0 × B03 + Ax1 × B13) + (Ax2 × B23)
CT = RB9                ; CT is used to store the fourth value since
                        ; RB9 will be corrupted.
```

Temporary Storage

CT

Outputs

RB6 = (Ax0 × B00) + (Ax1 × B10) + (Ax2 × B20)
RB7 = (Ax0 × B01) + (Ax1 × B11) + (Ax2 × B21)
RB8 = (Ax0 × B02) + (Ax1 × B12) + (Ax2 × B22)
RB9 = CT = (Ax0 × B03) + (Ax1 × B13) + (Ax2 × B23)

Note that the result of this operation is the completed row for a 1 × 3 by 3 × 3 or 3 × 3 by 3 × 3 matrix multiply.

Instruction Type

CEXEC, short

Example

See Example 5–4 for code for a 3 × 3 by 3 × 3 matrix multiply.

Syntax

Type	Syntax
Integer	**MMPY3**
Double-Precision	**MMPY3D**
Single-Precision	**MMPY3F**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	1	type	size
ID			0	0	0	0	0	1	1	1	0	0	0	0	0

Instruction to '34082

31	29							0
ID	0	0000	1110	0000	0011	111t	s000	0000

Description

This instruction multiplies the matrix B by a vector element, A3. This instruction may be combined with MMPY0, MMPY1, and MMPY2 to multiply matrices of several sizes. 1 × 4 by 4 × 4, 4 × 4 by 4 × 4, 1 × 3 by 3 × 3, and 3 × 3 by 3 × 3 matrix multiplies may be implemented.

Implied Operands

A matrix in FPU registers:

RA0 = B00	RA1 = B01	RA2 = B02	RA3 = B03
RA4 = B10	RA5 = B11	RA6 = B12	RA7 = B13
RA8 = B20	RA9 = B21	RB0 = B22	RB1 = B23
RB2 = B30	RB3 = B31	RB4 = B32	RB5 = B33

The accumulated sums from MMPY0, MMPY1 and MMPY2 for the resulting matrix:

RB6 = (Ax0 × B00) + (Ax1 × B10) + (Ax2 × B20)
RB7 = (Ax0 × B01) + (Ax1 × B11) + (Ax2 × B21)
RB8 = (Ax0 × B02) + (Ax1 × B12) + (Ax2 × B22)
CT = (Ax0 × B03) + (Ax1 × B13) + (Ax2 × B23)

The fourth element (Ax3) of a row vector: RB9 = Ax3

Algorithm

```
C = RB9
RB9 = CT
RB6 = RB6 + (C × RB2)      ; (Ax0 × B00 + Ax1 × B10 + Ax2 × B20)
                           ; + (Ax3 × B30)
RB7 = RB7 + (C × RB3)      ; (Ax0 × B01 + Ax1 × B11 + Ax2 × B21)
                           ; + (Ax3 × B31)
RB8 = RB8 + (C × RB4)      ; (Ax0 × B02 + Ax1 × B12 + Ax2 × B22)
                           ; + (Ax3 × B32)
RB9 = RB9 + (C × RB5)      ; (Ax0 × B03 + Ax1 × B13 + Ax2 × B23)
                           ; + (Ax3 × B33)
```

Temporary Storage

C

Outputs

The output of this operation is the result matrix row.
RB6 = Result x0
RB7 = Result x1
RB8 = Result x2
RB9 = Result x3

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**MNMX1** *CRs*
Double-Precision	**MNMX1D** *CRs*
Single-Precision	**MNMX1F** *CRs*

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	1	type	size
ID			CRs				0	0	1	0	0	0	0	0	0

Instruction to '34082

31 29	28 25	24 0
ID	CRs	00100 0000 0011 111t s000 0000

Description

The 1-D Min/Max function compares the current data to a current minimum value and a current maximum value. If the current data is less than the minimum then the minimum is set to the current data; if the current data is greater than the current maximum then the maximum value is set to the current data. For each current data tested a counter is incremented and when the minimum or maximum values are updated the current counter value is put in a minimum count or maximum count register so that the count of the data responsible for the minimum or maximum is in the respective count register. The INMNMX instruction should be used to initialize the min/max registers before the first MNMX1 instruction.

Operands

CRs TMS34082 register containing the value to test for minimum/maximum. Must be in the RA register file.

Implied Operands

RB0 = Current integer minimum
RB1 = Current integer maximum
COUNTX contains the counts for the current maximum and minimum values
Bits 15-0 are the count value for the current minimum
Bits 31-16 are the count value for the current maximum

Algorithm

```
If CRs < RB0
  RB0 = CRs                    ; RB0 tracks current X minimum
  COUNTX bits 15-0 = Count
If CRs > RB1
  RB1 = CRs                    ; RB1 tracks current X maximum
  COUNTX bits 31-16 = Count
Count = Count + 1
```

Temporary Storage

None

Outputs

RB0 = minimum of (CRs and RB0)
RB1 = maximum of (CRs and RB1)
COUNTX 15-0 is updated to the current count if CRs is a minimum.
COUNTX 31-16 is updated to the current count if CRs is a maximum.

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**MNMX2** CRs_1, CRs_2
Double-Precision	**MNMX2D** CRs_1, CRs_2
Single-Precision	**MNMX2F** CRs_1, CRs_2

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	1	type	size
ID			CRs_1				0	0	1	1	0	CRs_2			

Instruction to '34082

31 29	28 25	24 20	19 16	15 0
ID	CRs_1	00110	CRs_2	1011 111t s000 0000

Description

The 2-D Min/Max function compares two current data values (X and Y) to a current minimum value and a current maximum value. If the current data is less than the minimum then the minimum is set to the current data; if the current data is greater than the current maximum then the maximum value is set to the current data. For each current data tested a counter is incremented and when the minimum or maximum values are updated, the current counter value is put in a minimum count or maximum count register so that the count of the data responsible for the minimum or maximum is in the respective count register. The INMNMX instruction should be used to initialize the min/max registers before the first MNMX2 instruction.

Operands

CRs_1 TMS34082 register containing the value to test for X minimum/maximum. Must be in RA register file.

CRs_2 TMS34082 register containing the value to test for Y minimum/maximum. Must be in RA register file.

Implied Operands

RB0 = current X minimum
RB1 = current X maximum
RB2 = current Y minimum
RB3 = current Y maximum
COUNTX contains the counts for the current maximum and minimum values
Bits 15-0 are the count value for the current X minimum
Bits 31-16 are the count value for the current X maximum
COUNTY contains the counts for the current Y maximum and minimum values
Bits 15-0 are the count value for the current Y minimum
Bits 31-16 are the count value for the current Y maximum

Algorithm

```
If CRs1 < RB0
  RB0 = CRs1                    ; RB0 tracks current X minimum
  COUNTX bits 15-0 = Count
If CRs1 > RB1
  RB1 = CRs1                    ; RB1 tracks current X maximum
  COUNTX bits 31-16 = Count
If CRs2 < RB2
  RB2 = CRs2                    ; RB2 tracks current Y minimum
  COUNTY bits 15-0 = Count
If CRs2 > RB3
  RB3 = CRs2                    ; RB3 tracks current Y maximum
  COUNTY bits 31-16 = Count
Count = Count + 1
```

Temporary Storage

None

Outputs

RB0 = minimum of (CRs_1 and RB0)
RB1 = maximum or (CRs_1 and RB1)
RB2 = minimum of (CRs_2 and RB2)
RB3 = maximum or (CRs_2 and RB3)
COUNTX 15-0 is updated to the current count if CRs_1 is a X minimum.
COUNTX 31-16 is updated to the current count if CRs_1 is a X maximum.
COUNTY 15-0 is updated to the current count if CRs_2 is a Y minimum.
COUNTY 31-16 is updated to the current count if CRs_2 is a Y maximum.

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**MOVE** *Rs, CRd*
Single-Precision	**MOVF** *Rs, CRd*

Execution

Rs → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	0	1	1	0	type	0	0	0	0	0	0	0	0
ID			0	0	0	0	0	0	0	0	CRd				

Instruction to '34082

31	29	28	21	20	16	15	0
ID	0000	0000	CRd	0100	110t	0000	0000

Operands

Rs TMS34020 source register for the 32-bit value to TMS34082

CRd TMS34082 destination register to hold the 32-bit value

Description

MOVx moves the contents of Rs into CRd.

Instruction Type

CMOVGC, one register

Example

```
MOVF A5, RA7
```

This example moves the single-precision floating-point contents of TMS34020 register A5 into TMS34082 register RA7.

Syntax

Type	Syntax
Integer	**MOVE** *Rs_1, Rs_2, CRd*
Double-Precision	**MOVD** *Rs_1, Rs_2, CRd*
Single-Precision	**MOVF** *Rs_1, Rs_2, CRd*

Execution

Integer or Single-Precision:
$Rs_1 \rightarrow$ CRd
advance to next TMS34082 register
$Rs_2 \rightarrow$ CRd

Double-Precision:
$Rs_1 \rightarrow$ CRd (MSH or LSH)
$Rs_2 \rightarrow$ CRd (LSH or MSH)

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	1	1	0	type	size	0	0	R	Rs_2			
ID			0	0	0	0	0	0	0	0	CRd				

Instruction to '34082

31	29	28	20	19	16	15	0
ID		0000 0000		CRd		0100 110t s000 0000	

Operands

Rs_1 TMS34020 source register for the first value (or half of a double-precision value) to TMS34082

Rs_2 TMS34020 source register for the second value (or the remaining half of the double-precision value) to TMS34082

CRd TMS34082 destination register that holds the first value. For integer and single-precision moves, the second value will be placed in the next register in the TMS34082 register sequence list.

Description

MOVx moves the contents of Rs_1 and Rs_2 into CRd (and CRd+1 for integer and single-precision instructions).

For double-precision moves, the TMS34082 configuration register LOAD bit determines whether the LSBs or the MSBs will be moved first:

If the LOAD bit = 1, then the LSBs are moved first (32 LSBs of the fraction)

If the LOAD bit = 0, then the MSBs are moved first (sign, exponent, and 20 MSBs of the fraction)

The LOAD bit default is 0.

Instruction Type

CMOVGC, two registers

Example

```
MOVE A5, A6, RA7
```

This instruction moves the integer contents of TMS34020 registers A5 and A6 into TMS34082 registers, RA7 and RA8, respectively.

Syntax

Type	Syntax
Integer	**MOVE** *CRs, Rd*
Single-Precision	**MOVF** *CRs, Rd*

Execution

CRs → Rd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	1	R	Rd			
0	1	0	0	1	1	1	type	0	0	0	0	0	0	0	0
ID			0	0	0	0	0	0	0	0	CRs				

Instruction to '34082

31	29	28	21	20	16	15	0
ID	0000	0000	CRs	0100	111t	0000	0000

Operands

CRs TMS34082 source register holding the 32-bit value

Rd TMS34020 destination register

Description

MOVx moves 32-bit value from TMS34082 register CRs to TMS34020 register Rd.

Instruction Type

CMOVCG, one register

Example

```
MOVE RA7, A5
```

This example moves the integer contents of TMS34082 register RA7 to TMS34020 register A5.

Syntax

MOVD *CRs, Rd_1, Rd_2*

Execution

CRs (MSH or LSH) → Rd_1
CRs (LSH or MSH) → Rd_2

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	1	R	Rd_1			
0	1	0	0	1	1	1	1	1	0	0	R	Rd_2			
ID			0	0	0	0	0	0	0	0	CRd				

Instruction to '34082

31	29	28	20	19	16	15	0
ID	0000	0000	CRd	0100	1111	1000	0000

Operands

CRs TMS34082 source register holding the value to TMS34020

Rd_1 TMS34020 destination register for half the double-precision value

Rd_2 TMS34020 destination register for the remaining half of the double-precision value

Description

MOVD moves one 64-bit value from TMS34082 register CRs to TMS34020 registers Rd_1 and Rd_2.

The TMS34082 configuration register LOAD bit determines whether the LSBs or the MSBs will be moved first:

If the LOAD bit = 1, then the LSBs are moved first (32 LSBs of the fraction)

If the LOAD bit = 0, then the MSBs are moved first (sign, exponent, and 20 MSBs of the fraction)

The LOAD bit default is 0.

Instruction Type

CMOVCG, two registers

Example

```
MOVD RA7, A5, A6
```

This example moves the double-precision floating-point contents of TMS34082 register RA7 to TMS34020 registers A5 and A6. The order (MSBs or LSBs in A5) depends on the value of the LOAD bit in the configuration register.

Syntax

Type	Syntax
Integer	**MOVE** **Rs+, CRd, Rd*
Double-Precision	**MOVD** **Rs+, CRd, Rd*
Single-Precision	**MOVF** **Rs+, CRd, Rd*

Execution

Integer or Single-Precision:

If **Rd = 0**
Repeat 32 times
*Rs → CRd
Rs + 32 → Rs
advance to next TMS34082 register

If **Rd = 1 → 31**
Repeat *Rd* times
*Rs → CRd
Rs + 32 → Rs
advance to next TMS34082 register

Double-Precision:

If **Rd = 0**
Repeat 16 times
*Rs → CRd
Rs + 32 → CRd
*Rs → CRd
Rs + 32 → CRd
advance to next TMS34082 register

If **Rd = 1 → 31**
Repeat *Rd/2* times
*Rs → CRd
Rs + 32 → CRd
*Rs → CRd
Rs + 32 → CRd
advance to next TMS34082 *register*

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	1	R	Rd			
1	0	0	0	1	1	0	type	size	0	0	R	Rs			
ID			0	0	0	0	0	0	0	0	CRd				

Instruction to '34082

31	29	28	20	19	16	15	0
ID	0000	0000	CRd	1000	110t	s000	0000

Operands

Rs TMS34020 source register containing the address of the first 32-bit value (or half of the 64-bit value) to move to the TMS34082

CRd TMS34082 destination register to hold the first value

Rd TMS34020 register containing the number of 32-bit transfers to make. This value must be in the range 0 to 31

If Rd = 0, then 32 32-bit transfers are made
If Rd = 1 → 31, then Rd 32-bit transfers are made

Note that because 64-bit floating-point values require two 32-bit moves, an odd number in Rd will give unpredictable results.

Description

MOVx moves values from memory beginning at the address in Rs into TMS34082 registers beginning at CRd. Rs is incremented after each transfer. CRs is advanced to the next register in the sequence list after each 32-bit transfer for integer and single-precision moves, after every two 32-bit transfers for double-precision moves. The number of 32-bit transfers made is determined by the value of *Rd*.

For double-precision moves, the TMS34082 configuration register LOAD bit determines whether the LSBs or the MSBs will be moved first:

If the LOAD bit = 1,	then the LSBs are moved first (32 LSBs of the fraction)
If the LOAD bit = 0,	then the MSBs are moved first (sign, exponent, and 20 MSBs of the fraction)

The LOAD bit default is 0.

Instruction Type

CMOVMC, postincrement, register count

Example

```
MOVE *A5+, RA7, B7
```

This instruction moves integer values from TMS34020 memory location pointed to by A5 to TMS34082 registers beginning at RA7. After each 32-bit transfer, register A5 is incremented, and the TMS34082 destination is advanced to the next register in the TMS34082 register sequence list. B7 holds the number of 32-bit transfers to be made.

Syntax

Type	Syntax
Integer	**MOVE** **Rs+, CRd, [,count]*
Double-Precision	**MOVD** **Rs+, CRd, [,count]*
Single-Precision	**MOVF** **Rs+, CRd, [,count]*

Execution

Repeat count times
- *Rs → CRd
- Rs + 32 → Rs
- *Rs → CRd
- Rs + 32 → Rs
- advance to the next TMS34082 register

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	transfers				
1	0	0	0	1	1	0	type	size	0	0	R	Rs			
ID			0	0	0	0	0	0	0	0	CRd				

Instruction to '34082

31	29	28	20	19	16	15	0
ID	0000	0000	CRd	1000	110t	s000	0000

Operands

Rs — TMS34020 source register containing the address of the first 32-bit value (or half the first 64-bit value) to move to the TMS34082

CRd — TMS34082 destination register to hold the first operand

count — The number of 32-bit or 64-bit transfers to make. This value must be in the range 1 to 32 for integer and single-precision moves or 1 to 16 for double-precision moves. The default value is 1. Count determines the value of transfers:

Integer or Single-Precision:

If *count* = 32,	then transfers = 0
If *count* = 1 → 31,	then transfers = *count*

Double-Precision:

If *count* = 16,	then transfers = 0
If *count* = 1 → 15,	then transfers = $2 \times count$

Description

MOVx moves values from memory beginning at the address in Rs into TMS34082 registers beginning at CRd. Rs is incremented after each transfer. CRs is advanced to the next register in the sequence list after each 32-bit transfer for integer and single-precision moves, after every two 32-bit transfers for double-precision moves. The number of 32-bit transfers made is determined by the value of *count*.

For double-precision moves, the TMS34082 configuration register LOAD bit determines whether the LSBs or the MSBs will be moved first:

If the LOAD bit = 1,	then the LSBs are moved first (32 LSBs of the fraction)
If the LOAD bit = 0,	then the MSBs are moved first (sign, exponent, and 20 MSBs of the fraction)

The LOAD bit default is 0.

Instruction Type CMOVMC, postincrement, constant count

Example

```
MOVD *A5+, RB7, 4
```

This example moves four 64-bit double-precision floating-point values from TMS34020 memory location pointed to by A5 to TMS34082 registers beginning at RB7. After each 32-bit transfer, register A5 is incremented; after every two 32-bit transfers, the TMS34082 destination is advanced to the next register in the TMS34082 register sequence list. *Count* specifies that four 64-bit transfers (eight 32-bit transfers) are made.

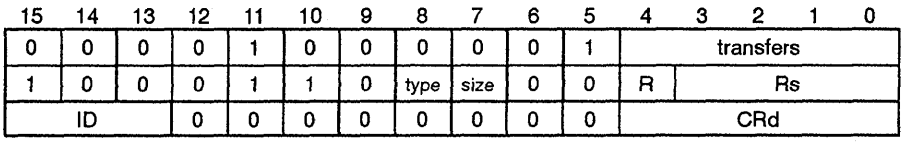
Syntax

Type	Syntax
Integer	**MOVE** *–*Rs, CRd, [, count]*
Double-Precision	**MOVD** *–*Rs, CRd [, count]*
Single-Precision	**MOVF** *–*Rs, CRd [, count]*

Execution

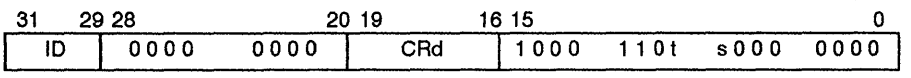
Repeat count times
  Rs – 32 → Rs
  *Rs → CRd
  Rs – 32 → Rs
  *Rs → CRd
  advance to the next TMS34082 register

'34020 Instruction Words



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	transfers				
1	0	0	0	1	1	0	type	size	0	0	R	Rs			
ID			0	0	0	0	0	0	0	0	CRd				

Instruction to '34082



31	29	28			20	19		16	15				0
ID		0000		0000		CRd			1000	110t	s000	0000	

Operands

Rs TMS34020 source register containing the address of the bit immediately after the first 32- or 64-bit value to move to the TMS34082

CRd TMS34082 destination register to hold the first value

count The number of 32- or 64-bit transfers to make. This value must be in the range 1 to 32 for integer and single-precision moves or 1 to 16 for double-precision moves; the default value is 1. Count determines the value of transfers:

Integer or Single-Precision:
  If *count* = 32, then transfers = 0
  If *count* = 1 → 31, then transfers = *count*

Double-Precision:
  If *count* = 16, then transfers = 0
  If *count* = 1 → 15, then transfers = $2 \times count$

Description

MOVx moves values from memory beginning at the address in (Rs – 32) into TMS34082 registers beginning at CRd. Before each transfer, the contents of Rs are decremented; after each transfer (or every two transfers for double-precision moves), the TMS34082 destination is advanced to the next register in the TMS34082 register sequence list. The number of transfers made is determined by the value of *count*.

For double-precision moves, the TMS34082 configuration register LOAD bit determines whether the LSBs or the MSBs will be moved first:

If the LOAD bit = 1, then the LSBs are moved first (32 LSBs of the fraction)

If the LOAD bit = 0, then the MSBs are moved first (sign, exponent, and 20 MSBs of the fraction)

The LOAD bit default is 0.

Instruction Type CMOVMC, predecrement, constant count

Example

```
MOVF -*A5, RB7, 4
```

This example moves four 32-bit single-precision floating-point values from TMS34020 memory location pointed to by (A5–32) to TMS34082 registers beginning at RB7. Before each 32-bit transfer, register A5 is decremented; after each transfer, TMS34082 destination is advanced to the next register in the TMS34082 register sequence list. Count specifies that four 32-bit transfers are made.

Syntax

Type	Syntax
Integer	**MOVE** *CRd, *Rd+ [, count]*
Double-Precision	**MOVD** *CRd, *Rd+ [, count]*
Single-Precision	**MOVF** *CRd, *Rd+ [, count]*

Execution

Repeat count times
  CRs → *Rd
  Rd + 32 → Rd
  CRs → *Rd
  Rd + 32 → Rd
  advance to the next TMS34082 register

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	1	R	Rd			
1	0	0	0	1	1	1	type	size	0	0	transfers				
ID			0	0	0	0	0	0	0	0	CRd				

Instruction to '34082

31	29	28	20	19	16	15	0
ID	0000	0000	CRd	1000	111t	s000	0000

Operands

CRs TMS34082 source register for the first 32-bit value (or half of the first 64-bit value) to TMS34020 memory

Rd TMS34020 register containing the address for the first value transferred

count The number of 32- or 64-bit transfers to make. This value must be in the range 1 to 32 for integer and single-precision moves or 1 to 16 for double-precision moves. The default value is 1. Count determines the value of transfers:

Integer or Single-Precision:

If *count* = 32,	then transfers = 0
If *count* = 1 → 31,	then transfers = *count*

Double-Precision:

If *count* = 16,	then transfers = 0
If *count* = 1 → 15,	then transfers = $2 \times$ *count*

Description

MOVx moves the values from TMS34082 registers beginning at CRd to memory beginning at the address in Rd. After each 32-bit transfer, Rd is incremented. The TMS34082 register is advanced to the next register in the TMS34082 register sequence after every 32-bit transfer for integer and single-precision moves or after every second 32-bit transfer for double-precision moves. The number of transfers made is determined by the value of *count*.

For double-precision moves, the TMS34082 configuration register LOAD bit determines whether the LSBs or the MSBs will be moved first:

If the LOAD bit = 1,	then the LSBs are moved first (32 LSBs of the fraction)
If the LOAD bit = 0,	then the MSBs are moved first (sign, exponent, and 20 MSBs of the fraction)

The LOAD bit default is 0.

Instruction Type CMOVCM, postincrement, constant count

Example

```
MOVE RB7, *A5+, 4
```

This example moves four 32-bit integer values from TMS34082 registers beginning at RB7 to TMS34020 memory pointed to by A5. After each 32-bit transfer, register A5 is incremented, and the TMS34082 destination is advanced to the next register in the TMS34082 register sequence list. *Count* specifies that four 32-bit transfers are made.

Syntax

Type	Syntax
Integer	**MOVE** *CRs, –*Rd [, count]*
Double-Precision	**MOVD** *CRs, –*Rd [, count]*
Single-Precision	**MOVF** *CRs, –*Rd [, count]*

Execution

Repeat count times
  Rd – 32 → Rd
  CRs → *Rd
  Rd – 32 → Rd
  CRs → *Rd
  advance to the next TMS34082 register

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	1	R	Rd			
1	0	0	0	1	1	1	type	size	0	0	transfers				
ID			0	0	0	0	0	0	0	0	CRd				

Instruction to '34082

31	29	28	20	19	16	15	0
ID	0000	0000	CRd	1000	111t	s000	0000

Operands

CRd TMS34082 source register for the first value to TMS34020 memory

Rd TMS34020 register containing the address of the bit immediately following the 32 bits (or 64 bits for double-precision moves) used to store the first value transferred.

count The number of 32- or 64-bit transfers to make. This value must be in the range 1 to 32 for integer and single-precision moves or 1 to 16 for double-precision moves. The default value is 1. Count determines the value of transfers:

Integer or Single-Precision:

If *count* = 32,	then transfers = 0
If *count* = 1 → 31,	then transfers = *count*

Double-Precision:

If *count* = 16,	then transfers = 0
If *count* = 1 → 15,	then transfers = 2 × *count*

Description

MOVx moves the values from TMS34082 registers beginning at CRd to memory beginning at the address (Rd – 32). Before each 32-bit transfer, Rd is decremented; after each 32-bit transfer (or every two transfers for double-precision moves), the TMS34082 register is advanced to the next register in the TMS34082 register sequence. The number of 32-bit transfers made is determined by the value of *count*.

For double-precision moves, the TMS34082 configuration register LOAD bit determines whether the LSBs or the MSBs will be moved first:

If the LOAD bit = 1,	then the LSBs are moved first (32 LSBs of the fraction)
If the LOAD bit = 0,	then the MSBs are moved first (sign, exponent, and 20 MSBs of the fraction)

The LOAD bit default is 0.

Instruction Type CMOVCM, predecrement, constant count

Example

```
MOVD RB7, -*A5, 2
```

This example moves two 64-bit double-precision values from TMS34082 registers beginning at RB7 to TMS34020 memory pointed to by (A5 – 32). Before each 32-bit transfer, register A5 is decremented; after every two 32-bit transfers, the TMS34082 destination is advanced to the next register in the TMS34082 register sequence list. *Count* specifies that two 64-bit transfers are made (four 32-bit transfers).

Syntax

Type	Syntax
Integer	**MOVE** *CRs, CRd [, count]*
Double-Precision	**MOVD** *CRs, CRd [, count]*
Single-Precision	**MOVF** *CRs, CRd [, count]*

Execution

Repeat count times:
CRs → CRd
advance to the next TMS34082 CRs and CRd registers

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	0	1	type	size
ID			CRs				count				CRd				

Instruction to '34082

31 29	28 25	24 20	19 16	15 0
ID	CRs	count	CRd	0001 101t s000 0000

Operands

CRs Source register RA that holds the first value to move

CRd Destination register to hold the first value moved

count The number of registers to move. This value must be in the range of 1 to 15; the default is 1.

Description

MOVx moves *count* values from registers starting with CRs to registers starting with CRd. Both source and destination registers are advanced to the next register in the TMS34082 register sequence after each move.

The first source register, CRs, must be in the RA register file.

Instruction Type

CEXEC, short

Example

```
MOVF RA7, RB4, 3
```

This example moves three 32-bit single-precision floating-point values from TMS34082 register RA7, RA8, and RA9 to TMS34082 registers RB4, RB5, and RB6, respectively.

Syntax

Type	Syntax
Integer	**MOVE** *CRs, CRd [, count]*
Double-Precision	**MOVD** *CRs, CRd [, count]*
Single-Precision	**MOVF** *CRs, CRd [, count]*

Execution

Repeat count times:

CRs → CRd

advance to the next TMS34082 CRs and CRd registers

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	0	type	size
ID			CRs				count				CRd				

Instruction to '34082

31	29 28	25 24	20 19	16 15	0
ID	0000	count	CRd	0001 110t	s000 0000

Operands

CRs TMS34082 source register RB that holds the first value to move

CRd Destination register to hold the first value moved

count The number of registers to move. This value must be in the range of 1 to 15; the default is 1.

Description

MOVx moves *count* values from registers starting with CRs to registers starting with CRd. Both source and destination registers are advanced to the next registers in the TMS34082 register sequence after each move.

The first source register, CRs, must be in the RB register file.

Instruction Type

CEXEC, short

Example

```
MOVD RB3, RA7, 3
```

This example moves the 64-bit double-precision values from TMS34082 registers RB3, RB4, and RB5 to TMS34082 registers RA7, RA8, RA9, respectively.

Syntax

MOVFSRAM **Rd+ [, count]*

Execution

Repeat *count* times
*MCADDR → *Rd
Rd + 32 → Rd
MCADDR + 32 → MCADDR

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	1	R	Rd			
1	0	0	1	1	1	1	0	0	0	0	transfers				
ID			0	0	0	0	1	1	1	0	0	0	0	0	0

Instruction to '34082

31 29	28														0
ID	0	0	0	1	1	1	1	0	0	0	0	1	0	0	1	1	1	0	0	0	0	0	0	0	0

Operands

Rd — TMS34020 register (indirect postincrement) containing the address of the first 32-bit integer value transferred

count — The number of 32-bit transfers to make. This value must be in the range 1 to 32; the default value is 1. Count determines the value of transfers:

If *count* = 32,	then transfers = 0
If *count* = 1 → 31,	then transfers = *count*

Implied Operands

MCADDR
TMS34082 indirect address register containing the first address in memory on the MSD port for the first 32-bit value to move

Description

MOVFSRAM moves the 32-bit values from memory on the MSD point beginning with the address in MCADDR to memory beginning at the address in Rd. After each 32-bit transfer, Rd and MCADDR are incremented. The number of 32-bit transfers made is determined by the value of count.

NOTE: Since MCADDR refers to 32-bit word addresses and Rs refers to bit addresses, MCADDR is incremented by 1 (one 32-bit word) and Rs is incremented by 32 (one 32-bit word).

Instruction Type

CMOVCM, postincrement, constant count

Syntax

MOVFSRAM *–*Rd [, count]*

Execution

Repeat count times
Rd – 32 → Rd
*MCADDR → *Rd
MCADDR + 32 → MCADDR

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	1	R	Rd			
1	0	0	1	1	1	1	0	0	0	0	transfers				
ID			0	0	0	0	1	1	1	1	0	0	0	0	0

Instruction to '34082

31 29	28														0
ID	0	0011		1100		0000		1001		1110		0000		0000	

Operands

Rd TMS34020 register (indirect predecrement) containing the address of the bit immediately following the 32-bits used to store the first 32-bit integer value transferred

count The number of 32-bit transfers to make. This value must be in the range 1 to 32; the default value is 1. Count determines the value of transfers:

If *count* = 32,	then transfers = 0
If *count* = 1 → 31,	then transfers = *count*

Implied Operands

MCADDR
TMS34082 indirect address register containing the first address in memory on the MSD port for the first 32-bit value to move

Description

MOVFSRAM moves the 32-bit values from memory on the MSD port beginning at the address in MADDR to memory beginning at the address (Rd – 32). Before each 32-bit transfer, Rd is decremented; after each 32-bit transfer, MCADDR is incremented. The number of 32-bit transfers made is determined by the value of count.

NOTE: Since MCADDR refers to 32-bit word addresses and Rs refers to bit addresses, MCADDR is incremented by 1 (one 32-bit word) and Rs is decremented by 32 (one 32-bit word).

Instruction Type

CMOVCM, predecrement, constant count

Syntax

MOVTSRAM **Rs+, Rd*

Execution

If **Rd = 0**	If **Rd = 1 → 31**
Repeat 32 times	Repeat *Rd* times
*Rs → *MCADDR	*Rs → *MCADDR
Rs + 32 → Rs	Rs + 32 → Rs
MCADDR + 32 → MCADDR	MCADDR + 32 → MCADDR

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	1	R	Rd			
1	0	0	1	1	1	1	0	0	0	0	R	Rs			
ID			0	0	0	0	1	1	1	1	0	0	0	0	0

Instruction to '34082

31	29	28													0
ID		0 0001		1100		0000		1001		1110		0000		0000	

Operands

Rs TMS34020 source register (indirect postincrement) containing the address of the first 32-bit value to move

Rd TMS34020 register containing the number of 32-bit transfers to make. This value must be in the range 0 to 31

If Rd = 0, then 32 32-bit transfers are made
If Rd = 1 → 31, then Rd 32-bit transfers are made.

Implied Operands

MCADDR
TMS34082 indirect address register containing the first address in memory on the MSD port where the 32-bit values are to be stored

Description

MOVTSRAM moves 32-bit values from memory beginning at the address in Rs into memory on the MSD port beginning at the address in MCADDR. After each transfer, Rs and MCADDR are incremented. The number of 32-bit transfers made is determined by the contents of Rd.

NOTE: Since MCADDR refers to 32-bit word addresses and Rs refers to bit addresses, MCADDR is incremented by 1 (one 32-bit word) and Rs is incremented by 32 (one 32-bit word).

Instruction Type

CMOVMC, postincrement, register count

Syntax

MOVTSRAM **Rs+ [, count]*

Execution

Repeat *count* times
  *Rs → *MCADDR
  Rs + 32 → Rs
  MCADDR + 32 → MCADDR

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	transfers				
1	0	0	1	1	1	1	0	0	0	0	R	Rs			
ID			0	0	0	0	1	1	1	0	0	0	0	0	0

Instruction to '34082

31–29	28–0
ID	0 0001 1100 0000 1001 1110 0000 0000

Operands

Rs — TMS34020 source register (indirect postincrement) containing the address of the first 32-bit value to move

count — The number of 32-bit transfers to make. This value must be in the range 1 to 32; the default value is 1. Count determines the value of transfers:

If *count* = 32, then transfers = 0
If *count* = 1 → 31, then transfers = *count*

Implied Operands

MCADDR
TMS34082 indirect address register containing the first address in memory on the MSD port where the 32-bit values are to be stored

Description

MOVTSRAM moves the 32-bit values from memory beginning at the address in Rs into memory on the MSD port beginning at the address in MCADDR. After each transfer, Rs and MCADDR are incremented. The number of 32-bit transfers made is determined by the value of *count*.

NOTE: Since MCADDR refers to 32-bit word addresses and Rs refers to bit addresses, MCADDR is incremented by 1 (one 32-bit word) and Rs is incremented by 32 (one 32-bit word).

Instruction Type

CMOVMC, postincrement, constant count

Syntax

MOVTSRAM *–*Rs [, count]*

Execution

Repeat *count* times
Rs – 32 → Rs
*Rs → *MCADDR
MCADDR + 32 → MCADDR

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	transfers				
1	0	0	1	1	1	0	0	0	0	0	R	Rs			
ID			0	0	0	0	1	1	1	0	0	0	0	0	0

Instruction to '34082

31 29	28																0
ID	0	0	0	0	1	1	1	0	0	0	0	1	0	0	1	1	1	0	0	0	0	0	0	0	0

Operands

Rs — TMS34020 source register (indirect predecrement) containing the address of the bit immediately after first 32-bit integer to move to the coprocessor

count — The number of 32-bit transfers to make. This value must be in the range 1 to 32; the default value is 1. Count determines the value of transfers:

If *count* = 32,	then transfers = 0
If *count* = 1 → 31,	then transfers = *count*

Implied Operands

MCADDR
TMS34082 indirect address register containing the first address in memory on the MSD port where the 32-bit values are to be stored

Description

MOVTSRAM moves the 32-bit values from memory beginning at the address in (Rs – 32) into memory on the MSD port beginning at the address in MCADDR. Before each transfer, the contents of Rs are decremented; after each transfer, the contents of the MCADDR register are incremented. The number of 32-bit transfers made is determined by the value of count.

NOTE: Since MCADDR refers to 32-bit word addresses and Rs refers to bit addresses, MCADDR is incremented by 1 (one 32-bit word) and Rs is decremented by 32 (one 32-bit word).

Instruction Type

CMOVMC, predecrement, constant count

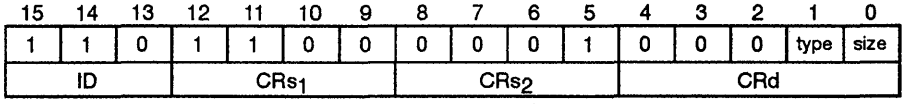
Syntax

Type	Syntax
Integer	**MPYS** *CRs$_1$, CRs$_2$, CRd*
Double-Precision	**MPYD** *CRs$_1$, CRs$_2$, CRd*
Single-Precision	**MPYF** *CRs$_1$, CRs$_2$, CRd*

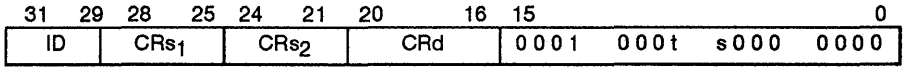
Execution

$CRs_1 \times CRs_2 \rightarrow CRd$

'34020 Instruction Words



Instruction to '34082



Operands

CRs_1 Coprocessor register containing the first operand

CRs_2 Coprocessor register containing the second operand

CRd Coprocessor destination register

Description

MPYx multiplies the contents of CRs_1 by the contents of CRs_2 and stores the result in CRd. The two operands, CRs_1 and CRs_2, must be in opposite register files.

Instruction Type

CEXEC, short

Example

```
MPYD RA5, RB6, RA7
```

This example multiplies the double-precision floating-point contents of RA5 by RB6 and stores the double-precision-point result in RA7.

Syntax

Type	Syntax
Integer	**MPYS** *Rs_1, Rs_2, CRs_1, CRs_2, CRd*
Single-Precision	**MPYF** *Rs_1, Rs_2, CRs_1, CRs_2, CRd*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
$CRs_1 \times CRs_2 \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	1	0	0	0	type	0	0	0	R	Rs_2			
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs_1		CRs_2		CRd		0101 000t 0000 0000	

Operands

Rs_1 TMS34020 source register for the first value to coprocessor

Rs_2 TMS34020 source register for the second value to coprocessor

CRs_1 Coprocessor register to contain the first operand

CRs_2 Coprocessor register to contain the second operand

CRd Coprocessor destination register

Description

MPYx loads the contents of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, multiplies $CRs_1 \times CRs_2$, and stores the result in CRd. The two operands, CRs_1 and CRs_2, must be in opposite register files.

The double-precision form of this instruction is not supported.

Instruction Type

CMOVGC, two registers

Example

```
MPYS A5, A6, RA5, RB6, RA7
```

This example loads TMS34020 registers A5 and A6 into TMS34082 registers RA5 and RB6 respectively, multiplies the contents of RA5 by RB6, and stores the integer result in RA7.

Syntax

Type	Syntax
Integer	**MPYS** **Rs+, CRs$_1$, CRs$_2$, CRd*
Double-Precision	**MPYD** **Rs+, CRs$_1$, CRs$_2$, CRd*
Single-Precision	**MPYF** **Rs+, CRs$_1$, CRs$_2$, CRd*

Execution

*Rs → CRs$_1$
Rs + 32 → Rs
*Rs → CRs$_1$
Rs + 32 → Rs
*Rs → CRs$_2$
Rs + 32 → Rs
*Rs → CRs$_2$
Rs + 32 → Rs
CRs$_1$ × CRs$_2$ → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	count		
1	0	0	1	0	0	0	type	size	0	0	R	Rs			
ID			CRs$_1$				CRs$_2$				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs$_1$		CRs$_2$		CRd		1001 000t s000 0000	

Operands

Rs TMS34020 source register containing the memory address

CRs$_1$ Coprocessor register to contain the first operand

CRs$_2$ Coprocessor register to contain the second operand

CRd Coprocessor destination register

Description

MPYx loads the contents of memory pointed to by Rs into CRs$_1$ and CRs$_2$, multiplies CRs$_1$ by CRs$_2$ and stores the result in CRd. After each load from memory, Rs is incremented by 32. The two operands, CRs$_1$ and CRs$_2$, must be in opposite register files.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
MPYS *A5+, RA5, RB6, RA7
```

This example loads memory starting at the address given by TMS34020 register A5 into coprocessor registers RA5 and RB6, multiplies the contents of RA5 by RB6 and stores the result in RA7.

Syntax

Type	Syntax
Integer	**MPYS** *–*Rs+, CRs$_1$, CRs$_2$, CRd*
Double-Precision	**MPYD** *– *Rs, CRs$_1$, CRs$_2$, CRd*
Single-Precision	**MPYF** *–*Rs+, CRs$_1$, CRs$_2$, CRd*

Execution

Rs – 32 → Rs
*Rs → CRs$_1$
Rs – 32 → Rs
*Rs → CRs$_1$
Rs – 32 → Rs
*Rs → CRs$_2$
Rs – 32 → Rs
*Rs → CRs$_2$
CRs$_1$ × CRs$_2$ → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	count		
1	0	0	1	0	0	0	type	size	0	0	R	Rs			
ID			CRs$_1$				CRs$_2$				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs$_1$		CRs$_2$		CRd		1001 000t s000 0000	

Operands

Rs TMS34020 source register containing the memory address

CRs$_1$ Coprocessor register to contain the first operand

CRs$_2$ Coprocessor register to contain the second operand

CRd Coprocessor destination register

Description

MPYx loads the contents of memory pointed to by Rs into CRs$_1$ and CRs$_2$, multiplies CRs$_1$ by CRs$_2$ and stores the result in CRd. Before each load from memory, Rs is decremented by 32. The two operands, CRs$_1$ and CRs$_2$, must be in opposite register files.

Instruction Type

CMOVMC, predecrement, constant count

Example

```
MPYD -*A5, RA5, RB6, RA7
```

This example loads memory starting at the address given by TMS34020 register A5 minus 32 into coprocessor registers RA5 and RB6, multiplies the contents of RA5 by RB6 and stores the result in RA7.

Syntax

Type	Syntax
Integer	**MTRAN**
Double-Precision	**MTRAND**
Single-Precision	**MTRANF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	0	0	1	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction to '34082

31	29	28													0
ID		0 0000		0000		0000		0011		001t		s000		0000	

Description

This instruction transposes a matrix. (Interchanges the row and column elements of the matrix.)

Implied Operands

RA0 = B00, RA1 = B01, RA2 = B02, RA3 = B03,
RA4 = B10, RA5 = B11, RA6 = B12, RA7 = B13,
RA8 = B20, RA9 = B21, RB0 = B22, RB1 = B23,
RB2 = B30, RB3 = B31, RB4 = B32, RB5 = B33

Temporary Storage

None

Outputs

RA0 = B00, RA1 = B10, RA2 = B20, RA3 = B30,
RA4 = B01, RA5 = B11, RA6 = B21, RA7 = B31,
RA8 = B02, RA9 = B12, RB0 = B22, RB1 = B32,
RB2 = B03, RB3 = B13, RB4 = B23, RB5 = B33

Instruction Type

CEXEC, short

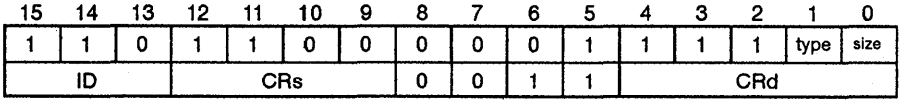
Syntax

Type	Syntax
Integer	**NEG** *CRs, CRd*
Double-Precision	**NEGD** *CRs, CRd*
Single-Precision	**NEGF** *CRs, CRd*

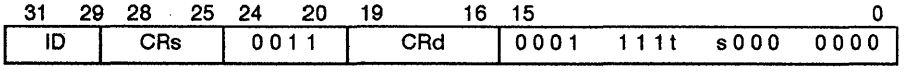
Execution

$-CRs \rightarrow CRd$

'34020 Instruction Words



Instruction to '34082



Operands

CRs TMS34082 register containing the operand

CRd TMS34082 destination register

Description

NEGx negates the contents of register CRs and stores the result in CRd. The integer instruction (NEG) takes the 2s complement of the contents of CRs and stores the result in CRd.

The source register, CRs, must be in the RA register file.

Instruction Type

CEXEC, short

Example

```
NEGD RA5, RB7
```

This example negates the double-precision floating-point value in RA5 and stores the result in RB7.

Syntax

Type	Syntax
Integer	**NEG** *Rs_1, CRs, CRd*
Double-Precision	**NEGD** *Rs_1, Rs_2, CRs, CRd*
Single-Precision	**NEGF** *Rs_1, CRs, CRd*

Execution

$Rs_1 \rightarrow CRs$

$Rs_1, Rs_2 \rightarrow CRs$

$-CRs \rightarrow CRd$

'34020 Instruction Words

Integer or Single-Precision:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs_1			
0	1	0	1	1	1	1	type	0	0	0	0	0	0	0	0
ID			CRs					0	0	1	1	CRd			

Double-Precision:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	1	1	1	1	1	1	0	0	R	Rs_2			
ID			CRs				0	0	1	1	CRd				

Instruction to '34082

31 29	28 25	24 20	19 16	15 0
ID	CRs	0011	CRd	0101 111t s000 0000

Operands

Rs_1 TMS34020 source register for the value (or half the value for double-precision) to TMS34082

Rs_2 TMS34020 source register for the remainder of the 64-bit double-precision floating-point value to TMS34082

CRs TMS34082 register containing the operand

CRd TMS34082 destination register

Description

NEGx loads the contents of Rs_1 (and Rs_2 for double-precision) into register CRs, negates CRs, and stores the result in CRd. The integer instruction (NEG) takes the 2s complement of the value.

The source register, CRs, must be in the RA TMS34082 register file.

Instruction Type

CMOVGC, one or two registers

Example

```
NEGD A5, A6, RA5, RB7
```

This example loads the double-precision floating-point contents of TMS34020 registers A5 and A6 into RA5, negates the contents of RA5 and stores the result in RB7.

Syntax

Type	Syntax
Integer	**NEG** **Rs+, CRs, CRd*
Double-Precision	**NEGD** **Rs+, CRs, CRd*
Single-Precision	**NEGF** **Rs+, CRs, CRd*

Execution

*Rs → CRs
Rs + 32 → Rs
*Rs → CRs
Rs + 32 → Rs
–CRs → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	0	transfers	
1	0	0	1	1	1	1	type	size	0	0	R	Rs			
ID			CRs				0	0	1	1	CRd				

Instruction to '34082

31	29	28	25	24	20	19	16	15	0
ID		CRs		0011		CRd		1001 111t s000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 register to contain the operand

CRd TMS34082 destination register

Description

NEGx loads the contents of memory pointed to by Rs into CRs, negates the contents of CRs, and stores the result in CRd. The integer instruction (NEG) takes the 2s complement of the value. After each load from memory, Rs is incremented by 32.

The source register, CRs, must be in the RA TMS34082 register file.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
NEGF *A5+, RA5, RB7
```

This example loads memory at the address given by TMS34020 register A5 into TMS34082 register RA5, negates the contents of RA5, and stores the result in RB7.

Syntax

Type	Syntax
Integer	**NEG** *–*Rs, CRs, CRd*
Double-Precision	**NEGD** *–*Rs, CRs, CRd*
Single-Precision	**NEGF** *–*Rs, CRs, CRd*

Execution

Rs – 32 → Rs
*Rs → CRs
Rs – 32 → Rs
*Rs → CRs
–CRs → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0
1	0	0	1	1	1	1	type	size	0	0	R	Rs			
ID			CRs				0	0	1	1	CRd				

Instruction to '34082

31	29	28	25	24	20	19	16	15	0
ID		CRs		0011		CRd		1001 111t s000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 register to contain the operand

CRd TMS34082 destination register

Description

NEGx loads the contents of memory pointed to by Rs into CRs, negates the contents of CRs, and stores the result in CRd. The integer instruction (NEG) takes the 2s complement of the value. Before each load from memory, Rs is decremented by 32.

The source register, CRs, must be in the RA TMS34082 register file.

Instruction Type

CMOVMC, predecrement, constant count

Example

```
NEGD -*A5, RA5, RB7
```

This example loads memory starting at the address given by TMS34020 register A5 minus 32 into TMS34082 register RA5 negates the contents of RA5, and stores the result in RB7.

Syntax

NOT *CRs, CRd*

Execution

NOT CRs → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	0	0
ID			CRs				0	0	0	1	CRd				

Instruction to '34082

31	29	28	25	24	20	19	16	15					0
ID		CRs		0 0 0 1		CRd		0 0 0 1	1 1 1 0	0 0 0 0	0 0 0 0		

Operands

CRs TMS34082 source register containing the 32-bit integer operand

CRd TMS34082 destination register

Description

NOT takes the 1s complement of the contents (integer) of CRs and stores the result in CRd.

The source register, CRs, must be in the RA TMS34082 register file.

Instruction Type

CEXEC, short

Example

```
NOT RA5, RA7
```

This example takes the 1s complement of the contents of RA5 and stores the result in RA7.

Syntax

NOT *Rs, CRs, CRd*

Execution

Rs → CRs
NOT CRs → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0
ID			CRs				0	0	0	1	CRd				

Instruction to '34082

31	29	28	25	24	20	19	16	15	0
ID		CRs		0001		CRd		0101 1110 0000 0000	

Operands

Rs TMS34020 source register for the 32-bit integer value to TMS34082

CRs TMS34082 register to contain the 32-bit integer operand

CRd TMS34082 destination register

Description

NOT loads the contents (integer) of Rs into the CRs, takes the 1s complement of the contents of register CRs, and stores the result in CRd.

The source register, CRs, must be in the RA TMS34082 register file.

Instruction Type

CMOVGC, one register

Example

```
NOT A5, RA5, RA7
```

This example loads TMS34020 register A5 into TMS34082 register RA5, takes the 1s complement of the contents of RA5, and stores the result in RA7.

Syntax

NOT **Rs+, CRs, CRd*

Execution

*Rs → CRs
Rs + 32 → Rs
NOT CRs → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	1
1	0	0	1	1	1	1	0	0	0	0	R	Rs			
ID			CRs				0	0	0	1	CRd				

Instruction to '34082

31	29	28	25	24	20	19	16	15	0
ID		CRs		0001		CRd		0101 1110 0000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 register to contain the 32-bit integer operand

CRd TMS34082 destination register

Description

NOT loads the integer contents of memory pointed to by Rs into the CRs, takes the 1s complement of the contents of register CRs, and stores the result in CRd. After each load from memory, Rs is incremented by 32.

The source register, CRs, must be in the RA TMS34082 register file.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
NOT *A5+, RA5, RA7
```

This example loads memory at the address given by TMS34020 register A5 into TMS34082 register RA5, takes the 1s complement of the contents of RA5, and stores the result in RA7.

Syntax

NOT *–*Rs, CRs, CRd*

Execution

Rs – 32 → Rs
*Rs → CRs
NOT CRs → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1
1	0	0	1	1	1	1	0	0	0	0	R	Rs			
ID			CRs				0	0	0	1	CRd				

Instruction to '34082

31	29	28	25	24	20	19	16	15	0
ID		CRs		0001		CRd		0101 1110 0000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs TMS34082 register to contain the 32-bit integer operand

CRd TMS34082 destination register

Description

NOT loads the contents (integer) of memory pointed to by Rs into the CRs, takes the 1s complement of the contents of register CRs, and stores the result in CRd. Before each load from memory, Rs is decremented by 32.

The source register, CRs, must be in the RA TMS34082 register file.

Instruction Type

CMOVMC, predecrement, constant count

Example

```
NOT -*A5, RA5, RA7
```

This example loads memory at the address given by TMS34020 register A5 minus 32 into TMS34082 register RA5 takes the 1s complement of the contents of RA5, and stores the result in RA7.

Syntax

Type	Syntax
Integer	**ONE** *CRd*
Double-Precision	**ONED** *CRd*
Single-Precision	**ONEF** *CRd*

Execution

1 → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	0	0	0	type	size
ID			1	1	0	1	1	1	0	1	CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	1101	1101	CRd	0001 000t s000 0000

Operands

CRd TMS34082 destination register.

Description

ONEx loads the value one (of the appropriate type) in the CRd register.

Instruction Type

CEXEC, short

Example

```
ONED RA3
```

This example loads RA3 with a double-precision one.

Syntax

Type	Syntax
Integer	**OUTC3X**
Double-Precision	**OUTC3XD**
Single-Precision	**OUTC3XF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	0	1	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction to '34082

31	29	28													0
ID		0 0000		0000		0000		0011		101t		s000		0000	

Description

The OUTC3Xx algorithm compares the given endpoints of a line to the clipping volume in the X-axis. The instruction sets three status register bits based on the location of the two endpoints with respect to the clipping volume. OUTC3Xx is used before the clipping instructions to determine which ends of the line need to be clipped.

Implied Operands

RA0 = X1	RB0 = X2
RA1 = Y1	RB1 = Y2
RA2 = Z1	RB2 = Z2
RA3 = W1	RB3 = W2

Algorithm

```
CT = RB3                    ; CT = W2
C  = RA3                    ; C  = W1
CT = CT – |RB0|             ; set V = 1 if (W2 – |X2|) < 0
C  = C – |RA0|              ; set N = 1 if (W1 – |X1|) < 0
If N = 1 and V = 1 and (sign X1 = sign X2), then set Z = 1
```

Temporary Storage

C, CT

Outputs

Status bits set:

Z	N	V	Description
1	1	1	both points outside on same side of volume in X-axis
0	1	1	both points outside on opposite sides of the volume in X-axis
0	1	0	only point P1 [X1,Y1,Z1,W1] outside of volume in X-axis
0	0	1	only point P2 [X2,Y2,Z2,W2] outside of volume in X-axis
0	0	0	both points P1 and P2 inside the volume in X-axis

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**OUTC3Y**
Double-Precision	**OUTC3YD**
Single-Precision	**OUTC3YF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	0	1	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	1

Instruction to '34082

31 29	28	0
ID	0 0000 0000 0001 0011 101t s000 0000	

Description

The OUTC3Yx algorithm compares the given endpoints of a line to the clipping volume in the Y-axis. The instruction sets three status register bits based on the location of the two endpoints with respect to the clipping volume. OUTC3Yx is used before the clipping instructions to determine which ends of the line need to be clipped.

Implied Operands

RA0 = X1	RB0 = X2
RA1 = Y1	RB1 = Y2
RA2 = Z1	RB2 = Z2
RA3 = W1	RB3 = W2

Algorithm

```
CT = RB3                    ; CT = W2
C  = RA3                    ; C  = W1
CT = CT - |RB1|             ; set V = 1 if (W2 - |Y2|) < 0
C  = C - |RA1|              ; set N = 1 if (W1 - |Y1|) < 0
If N = 1 and V = 1 and (sign Y1 = sign Y2), then set Z = 1
```

Temporary Storage

C, CT

Outputs

Status bits set:

Z	N	V	Description
1	1	1	both points outside on same side of volume in Y-axis
0	1	1	both points outside on opposite sides of the volume in Y-axis
0	1	0	only point P1 [X1,Y1,Z1,W1] outside of volume in Y-axis
0	0	1	only point P2 [X2,Y2,Z2,W2] outside of volume in Y-axis
0	0	0	both points P1 and P2 inside the volume in Y-axis

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**OUTC3Z**
Double-Precision	**OUTC3ZD**
Single-Precision	**OUTC3ZF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	0	1	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	1	0

Instruction to '34082

31	29	28						0
ID		0 0000	0000	0010	0011	101t	s000	0000

Description

The OUTC3Zx algorithm compares the given endpoints of a line to the clipping volume in the Z-axis. The instruction sets three status register bits based on the location of the two endpoints with respect to the clipping volume. OUTC3Zx is used before the clipping instructions to determine which ends of the line need to be clipped.

Implied Operands

RA0 = X1	RB0 = X2
RA1 = Y1	RB1 = Y2
RA2 = Z1	RB2 = Z2
RA3 = W1	RB3 = W2

Algorithm

```
CT = RB3                    ; CT = W2
C  = RA3                    ; C  = W1
CT = CT – |RB2|             ; set V = 1 if (W2 – |Z2|) < 0
C  = C – |RA2|              ; set N = 1 if (W1 – |Z1|) < 0
If N = 1 and V = 1 and (sign Z1 = sign Z2), then set Z = 1
```

Temporary Storage

C, CT

Outputs

Status bits set:

Z	N	V	Description
1	1	1	both points outside on same side of volume in Z-axis
0	1	1	both points outside on opposite sides of the volume in Z-axis
0	1	0	only point P1 [X1,Y1,Z1,W1] outside of volume in Z-axis
0	0	1	only point P2 [X2,Y2,Z2,W2] outside of volume in Z-axis
0	0	0	both points P1 and P2 inside the volume in Z-axis

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**PASS** *CRs, CRd*
Double-Precision	**PASSD** *CRs, CRd*
Single-Precision	**PASSF** *CRs, CRd*

Execution

CRs → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	type	size
ID			CRs				0	0	0	0	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		0000		CRd		0001 111t s000 0000	

Operands

CRs TMS34082 source register containing the operand. Must be from RA register file

CRd TMS34082 destination register

Description

PASSx moves a value from CRs to CRd. PASSx may be used to move values into and out of the C and CT feedback registers.

Instruction Type

CEXEC, short

Example

```
PASSD CT, RB0
```

This example moves the 64-bit double-precision value from feedback register CT to TMS34082 register RB0.

Syntax

Type	Syntax
Integer	**POLY** *CRs1, CRs2*
Double-Precision	**POLYD** *CRs1, CRs2*
Single-Precision	**POLYF** *CRs1, CRs2*

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	1	type	size
ID			CRs1				0	0	0	0	1	CRs2			

Instruction to '34082

31 29	28 25	24 20	19 16	15 0
ID	CRs1	0 0001	CRs2	0011 111t s000 0000

Description

POLYx performs a multiply and accumulate of the form:

$$A_n \times X^n + A_{n-1} \times X^{n-1} + A_{n-2} \times X^{n-2} + ...) + A_0$$

which can also be represented as:

$$(...((A_n \times X + A_{n-1}) \times X + A_{n-2}) \times X + ...) + A_0$$

where the value X is assumed present in the TMS34082 C register and the coefficients A_n through A_1 are to be multiplied by X and accumulated. This instruction multiplies CRs_1 by C, adds the result to CRs_2, and stores the sum in CRs_1.

Implied Operands

CRs_1 TMS34082 register containing A_n or accumulated value. Must be in the RA register file.

CRs_2 TMS34082 register containing A_{n-1} or next coefficient in series. Must be in the RB register file.

Algorithm

$CT = C \times CRs_1$; $A_n \times X$

$CRs_1 = CT + CRs_2$; $(A_n \times X) + A_{n-1}$

Temporary Storage

CT

Outputs

The new accumulated value in CRs_1

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**SCALE**
Double-Precision	**SCALED**
Single-Precision	**SCALEF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	0	0	0	type	size
ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Instruction to '34082

31	29	28													0
ID		0 0000		0000		0000		0011		000t		s000		0000	

Description

This instruction is used to scale and translate screen coordinates. Sn is the viewport scaling constant, Cn is the center of viewport constant, and V1 (X1, Y1, Z1, W1) is the vertex to scale and convert.

Implied Operands

```
RA0 = X1                    ; Vertex to scale and convert,
RA1 = Y1                    ; these are homogeneous coordinates
RA2 = Z1
RA3 = W1
RA7 = Sx                    RB7 = Cx
RA8 = Sy                    RB8 = Cy
RA9 = Sz                    RB9 = Cz
```

Algorithm

```
CT = RA3                    ; W1
C  = RA0 / CT
RA0 = (C × RA7) + RB7       ; X1 = ((X1 / W1) × Sx) + Cx
C  = RA1 / CT
RA1 = (C × RA8) + RB8       ; Y1 = ((Y1 / W1) × Sy) + Cy
RA2 = RA2 / CT
RA3 = CT
RA2 = RA2 × RA9
RA2 = RA2 + RB9             ; Z1 = ((Z1 / W1 ) × Sz) + Cz
```

Temporary Storage

C, CT

Outputs

```
RA0 = X1'
RA1 = Y1'
RA2 = Z1'
RA3 = W1
```

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**SQR** *CRs, CRd*
Double-Precision	**SQRD** *CRs, CRd*
Single-Precision	**SQRF** *CRs, CRd*

Execution

CRs × CRs → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	type	size
ID			CRs				1	0	0	0	CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs	1000	CRd	0001 111t s000 0000

Operands

CRs TMS34082 source register containing the operand

CRd TMS34082 destination register

Description

SQRx squares the contents of CRs and stores the result in CRd.

The source register, CRs, must be in the RA TMS34082 register file.

Instruction Type

CEXEC, short

Example

```
SQR RA5, RA7
```

This example squares the contents of RA5 and stores the result in register RA7.

Syntax

Type	Syntax
Integer	**SQR** *Rs_1, CRs, CRd*
Double-Precision	**SQRD** *Rs_1, Rs_2, CRs, CRd*
Single-Precision	**SQRF** *Rs_1, CRs, CRd*

Execution

$Rs_1 \rightarrow CRs$

$Rs_2 \rightarrow CRs$

$CRs \times CRs \rightarrow CRd$

'34020 Instruction Words

Integer or Single-Precision:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs_1			
0	1	0	1	1	1	1	type	0	0	0	0	0	0	0	0
ID			CRs				1	0	0	0	CRd				

Double-Precision:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	1	1	1	1	1	1	0	0	R	Rs_2			
ID			CRs				1	0	0	0	0	CRd			

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs	1000	CRd	0101 111t s000 0000

Operands

Rs_1 TMS34020 source register for the value (or half the value for double-precision operands) to TMS34082

Rs_2 TMS34020 source register for the remaining half of the 64-bit operand to the TMS34082

CRs TMS34082 register to contain the operand

CRd TMS34082 destination register

Description

SQRx loads the contents of Rs into CRs, squares the contents of CRs, and stores the result in CRd.

The source register, CRs, must be in the RA TMS34082 register file.

Instruction Type

CMOVGC, one register

Example

```
SQR A5, RA5, RB7
```

This example loads TMS34020 register A5 into TMS34082 register RA5, squares the contents of RA5, and stores the result in RB7.

Syntax

Type	Syntax
Integer	**SQR** **Rs+, CRs, CRd*
Double-Precision	**SQRD** **Rs+, CRs, CRd*
Single-Precision	**SQRF** **Rs+, CRs, CRd*

Execution

*Rs → CRs
Rs + 32 → Rs
~~*Rs → CRs~~
~~Rs + 32 → Rs~~
CRs × CRs → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	0	transfers	
1	0	0	1	1	1	1	type	size	0	0	R	Rs			
ID			CRs				1	0	0	0	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		1000		CRd		1001 111t s000 0000	

Operands

Rs TMS34020 source register containing the memory address

CRs TMS34082 register to contain the operand

CRd TMS34082 destination register

Description

SQRx loads the contents of memory pointed to by Rs into CRs, squares the contents of CRs, and stores the result in CRd. After each load from memory, Rs is incremented by 32.

The source register, CRs, must be in the RA TMS34082 register file.

Instruction Type

CMOVMC, postincrement, constant count

Example

```
SQR *A5+, RA5, RB7
```

This example loads memory starting at the address given by TMS34020 register A5 into TMS34082 register RA5, squares the contents of RA5, and stores the result in RB7.

Syntax

Type	Syntax
Integer	**SQR** *–*Rs, CRs, CRd*
Double-Precision	**SQRD** *–*Rs, CRs, CRd*
Single-Precision	**SQRF** *–*Rs, CRs, CRd*

Execution

Rs – 32 → Rs
*Rs → CRs
Rs – 32 → Rs
*Rs → CRs
CRs × CRs → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	transfers	
1	0	0	1	1	1	1	type	size	0	0	R	Rs			
ID			CRs				1	0	0	0	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		1000		CRd		1001 111t s000 0000	

Operands

Rs TMS34020 source register containing the memory address

CRs TMS34082 register to contain the operand

CRd TMS34082 destination register

Description

SQRx loads the contents of memory pointed to by Rs minus 32 into CRs, squares the contents of CRs, and stores the result in CRd. Before each load from memory, Rs is decremented by 32.

The source register, CRs, must be in the RA TMS34082 register file.

Instruction Type

CMOVMC, predecrement, constant count

Example

SQR –*A5, RA5, RB7

This example loads memory starting at the address given by TMS34020 register A5 minus 32 into TMS34082 register RA5, squares the contents of RA5, and stores the result in RB7.

Syntax

Type	Syntax
Integer	**SQRT** *CRs, CRd*
Double-Precision	**SQRTD** *CRs, CRd*
Single-Precision	**SQRTF** *CRs, CRd*

Execution

$\sqrt{CRs} \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	type	size
ID			CRs				1	0	0	1	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		1001		CRd		0001 111t s000 0000	

Operands

CRs TMS34082 source register containing the operand

CRd TMS34082 destination register

Description

SQRTx takes the square root of the contents of CRs and stores the result in CRd.

The source register, CRs, must be in the RA TMS34082 register file.

Transparency

CEXEC, short

Example

SQRTD RA5, RA7

This example takes the square root of the contents of RA5 and stores the result in RA7.

Syntax

Type	Syntax
Integer	**SQRT** *Rs_1, CRs, CRd*
Double-Precision	**SQRTD** *Rs_1, Rs_2, CRs, CRd*
Single-Precision	**SQRTF** *Rs_1, CRs, CRd*

Execution

$Rs_1 \rightarrow CRs$

$Rs_2 \rightarrow CRs$

$\sqrt{CRs} \rightarrow CRd$

'34082 Instruction Words

Integer or Single-Precision:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs_1			
0	1	0	1	1	1	1	type	0	0	0	0	0	0	0	0
ID			CRs				1	0	0	1	CRd				

Double-Precision:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	1	1	1	1	1	1	0	0	R	Rs_2			
ID			CRs				1	0	0	1	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		1001		CRd		0101 111t s000 0000	

Operands

Rs_1 TMS34020 source register for the value (or half the double-precision value) to the TMS34082

Rs_2 TMS34020 source register for the value for the remaining half of the double-precision value to the TMS34082

CRs TMS34082 register to contain the operand

CRd TMS34082 destination register

Description

SQRTx loads the contents of Rs into CRs, takes the square root of the contents of CRs, and stores the result in CRd.

The source register, CRs, must be in the RA TMS34082 register file.

Transparency

CMOVGC, one register

Example

SQRTF A5, RA5, RA7

This example loads TMS34020 register A5 into TMS34082 register RA5, takes the square root of the single-precision floating-point value in RA5, and stores the result in RA7.

Syntax

Type	Syntax
Integer	**SQRT** **Rs+, CRs, CRd*
Double-Precision	**SQRTD** **Rs+, CRs, CRd*
Single-Precision	**SQRTF** **Rs+, CRs, CRd*

Execution

*Rs → CRs
Rs + 32 → Rs
*Rs → CRs
Rs + 32 → Rs
$\sqrt{CRs}$ → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	0	transfers	
1	0	0	1	1	1	1	type	size	0	0	R	Rs			
ID			CRs				1	0	0	1	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		1001		CRd		1001 111t s000 0000	

Operands

Rs TMS34020 source register containing the memory address

CRs TMS34082 register to contain the operand

CRd TMS34082 destination register

Description

SQRTx loads the contents of memory pointed to by Rs into CRs, takes the square root of the contents of CRs, and stores the result in CRd. After each load from memory, Rs is incremented by 32.

Transparency

CMOVMC, postincrement, constant count

Example

SQRTD *A5+, RA5, RA7

This example loads memory starting at the address given by TMS34020 register A5 into TMS34082 register RA5, takes the square root of the double-precision floating-point value in RA5, and stores the result in RA7.

Syntax

Type	Syntax
Integer	**SQRT** *–*Rs, CRs, CRd*
Double-Precision	**SQRTD** *–*Rs, CRs, CRd*
Single-Precision	**SQRTF** *–*Rs, CRs, CRd*

Execution

Rs – 32 → Rs
*Rs → CRs
Rs – 32 → Rs
*Rs → CRs
$\sqrt{CRs}$ → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	transfers	
1	0	0	1	1	1	1	type	size	0	0	R	Rs			
ID			CRs				1	0	0	1	CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs		1001		CRd		1001 111t s000 0000	

Operands

Rs TMS34020 source register containing the memory address

CRs TMS34082 register to contain the operand

CRd TMS34082 destination register

Description

SQRTx loads the contents of memory pointed to by Rs minus 32 into CRs, takes the square root of the contents of CRs, and stores the result in CRd. Before each load from memory, Rs is decremented by 32.

The source register, CRs, must be in the RA TMS34082 register file.

Transparency

CMOVMC, predecrement, constant count

Example

SQRTF –*A5, RA5, RA7

This example loads memory starting at the address given by TMS34020 register A5 minus 32 into TMS34082 register RA5, takes the square root of the single-precision floating-point value in RA5, and stores the result in RA7.

Syntax

Type	Syntax
Integer	**SQRTA** *CRs, CRd*
Double-Precision	**SQRTAD** *CRs, CRd*
Single-Precision	**SQRTAF** *CRs, CRd*

Execution

$\sqrt{CRs} \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	type	size
ID			CRs				1	0	1	0	CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs	1010	CRd	0001 111t s000 0000

Operands

CRs TMS34082 register containing the operand

CRd TMS34082 destination register

Description

SQRTAx takes the square root of the absolute value of the contents of CRs and stores the result in CRd.

The source register, CRs, must be in the RA TMS34082 register file.

Transparency

CEXEC, short

Example

SQRTA RA5, RB7

This example takes the square root of the absolute value of RA5 and stores the result in RB7.

Syntax

Type	Syntax
Integer	**SQRTA** *Rs_1, CRs, CRd*
Double-Precision	**SQRTAD** *Rs_1, Rs_2, CRs, CRd*
Single-Precision	**SQRTAF** *Rs_1, CRs, CRd*

Execution

$Rs_1 \rightarrow CRs$

$Rs_2 \rightarrow CRs$

$\sqrt{CRs} \rightarrow CRd$

'34082 Instruction Words

Integer or Single-Precision:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs_1			
0	1	0	1	1	1	1	type	0	0	0	0	0	0	0	0
ID			CRs				1	0	1	0	CRd				

Double-Precision:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	0	R	Rs_1			
0	1	0	1	1	1	1	1	1	0	0	R	Rs_2			
ID			CRs				1	0	1	0	CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs	1010	CRd	0101 1111 s000 0000

Operands

Rs_1 TMS34020 source register for the value (or half of the 64-bit double-precision value) to TMS34082

Rs_2 TMS34020 source register for remaining half of the double-precision value to TMS34082

CRs TMS34082 register to contain the operand

CRd TMS34082 destination register

Description

SQRTAx loads the contents of Rs into CRs, takes the square root of the absolute value of the contents of CRs, and stores the result in CRd.

The source register, CRs, must be in the RA TMS34082 register file.

Transparency

CMOVGC, one register

Example

SQRTAD A3, A5, RA5, RA7

This example loads TMS34020 register A5 and A3 into TMS34082 register RA5, takes the square root of the absolute value of the contents of RA5, and stores the result in RA7.

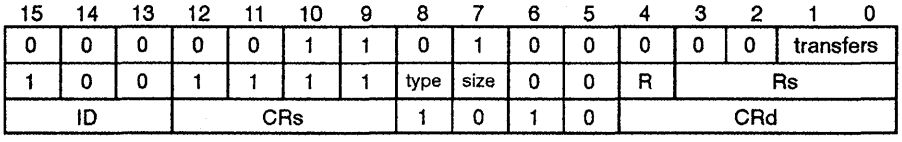
Syntax

Type	Syntax
Integer	**SQRTA** **Rs+, CRs, CRd*
Double-Precision	**SQRTAD** **Rs+, CRs, CRd*
Single-Precision	**SQRTAF** **Rs+, CRs, CRd*

Execution

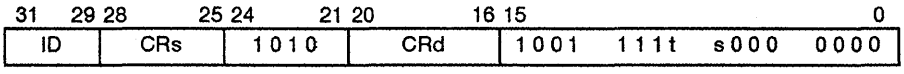
*Rs → CRs
Rs + 32 → Rs
~~*Rs → CRs~~
~~Rs + 32 → Rs~~
$\sqrt{CRs}$ → CRd

'34020 Instruction Words



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	0	transfers	
1	0	0	1	1	1	1	type	size	0	0	R	Rs			
ID			CRs				1	0	1	0	CRd				

Instruction to '34082



31 29	28 25	24 21	20 16	15 0
ID	CRs	1010	CRd	1001 111t s000 0000

Operands

Rs TMS34020 source register containing the memory address

CRs TMS34082 register to contain the operand

CRd TMS34082 destination register

Description

SQRTAx loads the contents of memory pointed to by Rs into CRs, takes the square root of the absolute value of the contents of CRs, and stores the result in CRd. After each load from memory, Rs is incremented by 32.

The source register, CRs, must be in the RA TMS34082 register file.

Transparency

CMOVMC, postincrement, constant count

Example

SQRTA *A5+, RA5, RA7

This example loads memory starting at the address given by TMS34020 register A5 into TMS34082 register RA5, takes the square root of the absolute value of the contents of RA5, and stores the result in RA7.

Syntax

Type	Syntax
Integer	**SQRTA** *–*Rs, CRs, CRd*
Double-Precision	**SQRTAD** *–*Rs, CRs, CRd*
Single-Precision	**SQRTAF** *–*Rs, CRs, CRd*

Execution

Rs – 32 → Rs
*Rs → CRs
~~*Rs → CRs~~
~~Rs – 32 → Rs~~
*Rs → CRs
$\sqrt{CRs}$ → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	transfers	
1	0	0	1	1	1	1	type	size	0	0	R	Rs			
ID			CRs				1	0	1	0	CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs	1010	CRd	1001 111t s000 0000

Operands

Rs TMS34020 source register containing the memory address

CRs TMS34082 register to contain the operand

CRd TMS34082 destination register

Description

SQRTAx loads the contents of memory pointed to by Rs minus 32 into CRs, takes the square root of the absolute value of the contents of CRs, and stores the result in CRd. Before each load from memory, Rs is decremented by 32.

The source register, CRs, must be in the RA TMS34082 register file.

Transparency

CMOVMC, predecrement, constant count

Example

SQRTA –*A5, RA5, RA7

This example loads memory starting at the address given by TMS34020 register A5 minus 32 into TMS34082 register RA5, takes the square root of the absolute value of the contents of RA5, and stores the result in RA7.

Syntax

Type	Syntax
Integer	**SUB** *CRs_1, CRs_2, CRd*
Double-Precision	**SUBD** *CRs_1, CRs_2, CRd*
Single-Precision	**SUBF** *CRs_1, CRs_2, CRd*

Execution

$CRs_1 - CRs_2 \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	0	1	type	size
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs_1	CRs_2	CRd	0000 001t s000 0000

Operands

CRs_1 TMS34082 RA register containing the minuend operand

CRs_2 TMS34082 RB register containing the subtrahend operand

CRd TMS34082 destination register

Description

SUBx subtracts the contents of CRs_2 from CRs_1 and stores the result in CRd.

The syntax for this instruction and the next instruction for subtract (RB register — RA register) is similar. The order of the operands determines which instruction is used. If an RA register is listed first, this instruction is used. If an RB register is first, the other instruction is used.

Transparency

CEXEC, short

Example

SUBD RA5, RB3, RA7

This example subtracts the contents of RB3 from RA5 and stores the result in RA7.

Syntax

Type	Syntax
Integer	**SUB** *CRs_2, CRs_1, CRd*
Double-Precision	**SUBD** *CRs_2, CRs_1, CRd*
Single-Precision	**SUBF** *CRs_2, CRs_1, CRd*

Execution

$CRs_2 - CRs_1 \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	1	1	type	size
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs_1	CRs_2	CRd	0000 011t s000 0000

Operands

CRs_1 TMS34082 RA register containing the subtrahend operand

CRs_2 TMS34082 RB register containing the minuend operand

CRd TMS34082 destination register

Description

SUBx subtracts the contents of CRs_1 from CRs_2 and stores the result in CRd. Notice in the syntax that the CRs_2 operand is listed first.

The syntax for this instruction and the previous instruction, subtract (RA register — RB register), is similar. The order of the operands determines which instruction is used. If an RA register is listed first, the previous instruction is used. If an RB register is first, this instruction is used.

Transparency

CEXEC, short

Example

SUB RB5, RA3, RA7

This example subtracts the contents of RA3 from RB5 and stores the result in RA7.

Syntax

Type	Syntax
Integer	**SUB** *Rs_1, Rs_2, CRs_1, CRs_2, CRd*
Single-Precision	**SUBF** *Rs_1, Rs_2, CRs_1, CRs_2, CRd*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
$CRs_1 - CRs_2 \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	0	0	1	type	0	0	0	R	Rs_2			
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs_1	CRs_2	CRd	0100 001t 0000 0000

Operands

Rs_1 TMS34020 source register for the first (minuend) value to TMS34082

Rs_2 TMS34020 source register for the second (subtrahend) value to TMS34082

CRs_1 TMS34082 RA register to contain the minuend operand

CRs_2 TMS34082 RB register to contain the subtrahend operand

CRd TMS34082 destination register

Description

SUBx loads the contents of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, subtracts the contents of CRs_2 from CRs_1, and stores the result in CRd.

The syntax for this instruction and the next instruction for subtract (RB register — RA register) is similar. The order of the operands determines which instruction is used. If an RA register is listed first, this instruction is used. If an RB register is first, the other instruction is used.

The double-precision form of this instruction is not supported.

Transparency

CMOVGC, two registers

Example

SUBF A0, A3, RA5, RB3, RA7

This example loads TMS34020 registers A0 and A3 into TMS34082 registers RA5 and RB3, subtracts the contents of RB3 from RA5, and stores the result in RA7.

Syntax

Type	Syntax
Integer	**SUB** *Rs_2, Rs_1, CRs_2, CRs_1, CRd*
Single-Precision	**SUBF** *Rs_2, Rs_1, CRs_2, CRs_1, CRd*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
$CRs_2 - CRs_1 \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3 2 1 0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1
0	1	0	0	0	1	1	type	0	0	0	R	Rs_2
ID			CRs_1				CRs_2				CRd	

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs_1	CRs_2	CRd	0100 011t 0000 0000

Operands

Rs_1 TMS34020 source register for the first (subtrahend) value to TMS34082

Rs_2 TMS34020 source register for the second (minuend) value to TMS34082

CRs_1 TMS34082 RA register to contain the subtrahend operand

CRs_2 TMS34082 RB register to contain the minuend operand

CRd TMS34082 destination register

Description

SUBx loads the contents of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, subtracts the contents of CRs_1 from CRs_2, and stores the result in CRd. Note that in the syntax, Rs_2 and CRs_2 are listed before Rs_1 and CRs_1.

The syntax for this instruction and the previous instruction, subtract (RA register — RB register), is similar. The order of the operands determines which instruction is used. If an RA register is listed first, the previous instruction is used. If an RB register is first, this instruction is used.

The double-precision form of this instruction is not supported.

Transparency

CMOVGC, two registers

Example

SUB A3, A0, RB5, RA3, RA7

This example loads TMS34020 registers B6 and A0 into TMS34082 registers RB5 and RA3, subtracts the contents of RA3 from RB5, and stores the result in RA7.

Syntax

Type	Syntax
Integer	**SUB** **Rs+, CRs$_1$, CRs$_2$, CRd*
Double-Precision	**SUBD** **Rs+, CRs$_1$, CRs$_2$, CRd*
Single-Precision	**SUBF** **Rs+, CRs$_1$, CRs$_2$, CRd*

Execution

*Rs → CRs$_1$
Rs + 32 → Rs
Rs → CRs$_1$
Rs + 32 → Rs
*Rs → CRs$_2$
Rs + 32 → Rs
*Rs → CRs$_2$
Rs + 32 → Rs
CRs$_1$ – CRs$_2$ → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	transfers		
1	0	0	0	0	0	1	type	size	0	0	R	Rs			
ID			CRs$_1$				CRs$_2$				CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs$_1$	CRs$_2$	CRd	1000 001t s000 0000

Operands

Rs TMS34020 register containing the memory address

CRs$_1$ TMS34082 RA register to contain the minuend operand

CRs$_2$ TMS34082 RB register to contain the subtrahend operand

CRd TMS34082 destination register

Description

SUBx loads the contents of memory pointed to by Rs into CRs$_1$ and CRs$_2$, subtracts the contents of CRs$_2$ from CRs$_1$, and stores the result in CRd. After each load from memory, Rs is incremented by 32.

The syntax for this instruction and the next instruction for subtract (RB register — RA register) is similar. The order of the operands determines which instruction is used. If an RA register is listed first, this instruction is used. If an RB register is first, the other instruction is used.

Transparency

CMOVMC, postincrement, constant count

Example

SUBF *A0+, RA5, RB3, RA7

This example loads memory starting at the address given by TMS34020 register A0 into TMS34082 registers RA5 and RB3, subtracts the contents of RB3 from RA5, and stores the result in RA7.

Syntax

Type	Syntax
Integer	**SUB** **Rs+, CRs_2, CRs_1, CRd*
Double-Precision	**SUBD** **Rs+, CRs_2, CRs_1, CRd*
Single-Precision	**SUBF** **Rs+, CRs_2, CRs_1, CRd*

Execution

$*Rs \rightarrow CRs_1$
$Rs + 32 \rightarrow Rs$
~~$*Rs \rightarrow CRs_1$~~
~~$Rs + 32 \rightarrow Rs$~~
$*Rs \rightarrow CRs_2$
$Rs + 32 \rightarrow Rs$
~~$*Rs \rightarrow CRs_2$~~
~~$Rs + 32 \rightarrow Rs$~~
$CRs_2 - CRs_1 \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	transfers		
1	0	0	0	0	1	1	type	size	0	0	R	Rs			
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31 29	28 25	21 20	16 15	0
ID	CRs_1	CRs_2	CRd	1000 011t s000 0000

Operands

Rs TMS34020 register containing the memory address

CRs_1 TMS34082 RA register to contain the subtrahend operand

CRs_2 TMS34082 RB register to contain the minuend operand

CRd TMS34082 destination register

Description

SUBx loads the contents of memory pointed to by Rs into CRs_1 and CRs_2, subtracts the contents of CRs_1 from CRs_2, and stores the result in CRd. After each load from memory, Rs is incremented by 32. Note in the syntax that CRs_2 is listed before CRs_1.

The syntax for this instruction and the previous instruction, subtract (RA register — RB register), is similar. The order of the operands determines which instruction is used. If an RA register is listed first, the previous instruction is used. If an RB register is first, this instruction is used.

Transparency

CMOVMC, postincrement, constant count

Example

SUBF *B6+, RB5, RA3, RA7

This example loads memory starting at the address given by TMS34020 register B6 into TMS34082 registers RB5 and RA3, subtracts the contents of RA3 from RB5, and stores the result in RA7.

Syntax

Type	Syntax
Integer	**SUB** *–*Rs, CRs$_1$, CRs$_2$, CRd*
Double-Precision	**SUBD** *–*Rs, CRs$_1$, CRs$_2$, CRd*
Single-Precision	**SUBF** *–*Rs, CRs$_1$, CRs$_2$, CRd*

Execution

Rs – 32 → Rs
*Rs → CRs$_1$
Rs – 32 → Rs
*Rs → CRs$_1$
Rs – 32 → Rs
*Rs → CRs$_2$
Rs – 32 → Rs
*Rs → CRs$_2$
CRs$_1$ – CRs$_2$ → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	transfers		
1	0	0	0	0	0	1	type	size	0	0	R	Rs			
ID			CRs$_1$				CRs$_2$				CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs$_1$	CRs$_2$	CRd	1000 001t s000 0000

Operands

Rs TMS34020 register containing the memory address

CRs$_1$ TMS34082 RA register to contain the minuend operand

CRs$_2$ TMS34082 RB register to contain the subtrahend operand

CRd TMS34082 destination register

Description

SUBx loads the contents of memory pointed to by Rs into CRs$_1$ and CRs$_2$, subtracts the contents of CRs$_2$ from CRs$_1$, and stores the result in CRd. Before each load from memory, Rs is decremented by 32.

The syntax for this instruction and the next instruction for subtract (RB register — RA register) is similar. The order of the operands determines which instruction is used. If an RA register is listed first, this instruction is used. If an RB register is first, the other instruction is used.

Transparency

CMOVMC, predecrement, constant count

Example

SUBF –*A0, RA5, RB3, RA7

This example loads memory starting at the address given by TMS34020 register A0 minus 32 into TMS34082 registers RA5 and RB3, subtracts the contents of RB3 from RA5, and stores the result in RA7.

Syntax

Type	Syntax
Integer	**SUB** *–*Rs, CRs_2, CRs_1, CRd*
Double-Precision	**SUBD** *–*Rs, CRs_2, CRs_1, CRd*
Single-Precision	**SUBF** *–*Rs, CRs_2, CRs_1, CRd*

Execution

Rs – 32 → Rs
*Rs → CRs_1
Rs – 32 → Rs
*Rs → CRs_1
Rs – 32 → Rs
*Rs → CRs_1
*Rs → CRs_2
CRs_2 – CRs_1 → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	transfers		
1	0	0	0	0	1	1	type	size	0	0	R	Rs			
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs_1	CRs_2	CRd	1000 011t s000 0000

Operands

Rs TMS34020 register containing the memory address

CRs_1 TMS34082 RA register to contain the subtrahend operand

CRs_2 TMS34082 RB register to contain the minuend operand

CRd TMS34082 destination register

Description

SUBx loads the contents of memory pointed to by Rs minus 32 into CRs_1 and CRs_2, subtracts the contents of CRs_1 from CRs_2, and stores the result in CRd. Before each load from memory, Rs is decremented by 32. Note in the syntax that CRs_2 is listed before CRs_1.

The syntax for this instruction and the previous instruction, subtract (RA register — RB register), is similar. The order of the operands determines which instruction is used. If an RA register is listed first, the previous instruction is used. If an RB register is first, this instruction is used.

Transparency

CMOVMC, postincrement, constant count

Example

SUBF *B6+, RB5, RA3, RA7

This example loads memory starting at the address given by TMS34020 register B6 minus 32 into TMS34082 registers RB5 and RA3, subtracts the contents of RA3 from RB5, and stores the result in RA7.

Syntax

Type	Syntax
Double-Precision	**SUBAD** *CRs_1, CRs_2, CRd*
Single-Precision	**SUBAF** *CRs_1, CRs_2, CRd*

Execution

$|CRs_1 - CRs_2| \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	1	0	1	1	size
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs_1	CRs_2	CRd	0000 1011 s000 0000

Operands

CRs_1 Coprocessor register containing the first operand. Must be from RA register file.

CRs_2 Coprocessor register containing the second operand. Must be from RB register file.

CRd Coprocessor destination register

Description

This instruction subtracts CRs_2 from CRs_1, placing the absolute value of the result in CRd.

The integer form of this instruction is not supported.

Instruction Type

CEXEC, short

Example

SUBAD RA8, RB3, RB1

This example subtracts the double-precision floating-point contents of RB3 from the contents of RA8, takes the absolute value of the difference, and stores the result in RB1.

Syntax

SUBAF *Rs_1, Rs_2, CRs_1, CRs_2, CRd*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
$|CRs_1 - CRs_2| \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	1	0	1	1	0	0	0	R	Rs_2			
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs_1	CRs_2	CRd	0100	1011	0000	0000	

Operands

Rs_1 TMS34020 source register for first 32-bit single-precision floating-point value to coprocessor

Rs_2 TMS34020 source register for second 32-bit single-precision floating-point value to coprocessor

CRs_1 Coprocessor RA register to contain the first single-precision operand

CRs_2 Coprocessor RB register to contain the second single-precision operand

CRd Coprocessor destination register

Description

This instruction loads the contents of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively and subtracts CRs_2 from CRs_1, placing the absolute value of the result in CRd.

The integer and double-precision forms of this instruction are not supported.

Instruction Type

CMOVGC, two registers

Example

SUBAF A9, A3, RA9, RB3, RB1

This instruction loads the contents of TMS34020 registers A9 and A3 into coprocessor registers RA9 and RB3 respectively, subtracts RB3 from RA9, takes the absolute value of the difference, and stores the result in RB1.

Syntax

Type	Syntax
Double-Precision	**SUBAD** **Rs+, CRs_1, CRs_2, CRd*
Single-Precision	**SUBAF** **Rs+, CRs_1, CRs_2, CRd*

Execution

*Rs → CRs_1
Rs + 32 → Rs
*Rs → CRs_1
Rs + 32 → Rs
*Rs → CRs_2
Rs + 32 → Rs
*Rs → CRs_2
Rs + 32 → Rs
$|CRs_1 - CRs_2|$ → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	transfers		
1	0	0	0	1	0	1	1	size	0	0	R	Rs			
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31	29	28	25	24	21	20	16	15	0
ID		CRs_1		CRs_2		CRd		1000 1011 s000 0000	

Operands

Rs TMS34020 register containing the memory address

CRs_1 Coprocessor RA register to contain the first operand

CRs_2 Coprocessor RB register to contain the second operand

CRd Coprocessor destination register

Description

This instruction loads the contents of memory pointed to by Rs into CRs_1 and CRs_2 and subtracts CRs_2 from CRs_1, placing the absolute value of the result in CRd. After each load from memory, Rs is incremented by 32.

The integer form of this instruction is not supported.

Instruction Type

CMOVMC, postincrement, constant count

Example

SUBAD *A9+, RA9, RB3, RB1

This instruction loads the contents memory starting at the address given by TMS34020 register A9 into coprocessor registers RA9 and RB3 respectively, subtracts RB3 from RA9, takes the absolute value of the difference, and stores the result in RB1.

Syntax

Type	Syntax
Double-Precision	**SUBAD** $-*Rs, CRs_1, CRs_2, CRd$
Single-Precision	**SUBAF** $-*Rs, CRs_1, CRs_2, CRd$

Execution

$Rs - 32 \rightarrow Rs$
$*Rs \rightarrow CRs_1$
~~$Rs - 32 \rightarrow Rs$~~
~~$*Rs \rightarrow CRs_1$~~
$Rs - 32 \rightarrow Rs$
$*Rs \rightarrow CRs_2$
~~$Rs - 32 \rightarrow Rs$~~
~~$*Rs \rightarrow CRs_2$~~
$|CRs_1 - CRs_2| \rightarrow CRd$

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	transfers		
1	0	0	0	1	0	1	1	size	0	0	R	Rs			
ID			CRs_1				CRs_2				CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	CRs_1	CRs_2	CRd	1000 1011 s000 0000

Operands

Rs TMS34020 register containing the memory address

CRs_1 Coprocessor RA register to contain the first operand

CRs_2 Coprocessor RB register to contain the second operand

CRd Coprocesor destination register

Description

This instruction loads the contents of memory pointed to by Rs into CRs_1 and CRs_2 and subtracts CRs_2 from CRs_1, placing the absolute value of the result in CRd. Before each load from memory, Rs is decremented by 32.

The integer form of this instruction is not supported.

Instruction Type

CMOVMC, predecrement, constant count

Example

SUBAD –*A9, RA9, RB3, RB1

This instruction loads the contents memory starting at the address given by TMS34020 register A9 minus 32 into coprocessor registers RA9 and RB3 respectively, subtracts RB3 from RA9, takes the absolute value of the difference, and stores the result in RB1.

Syntax

Type	Syntax
Integer	**TWO** *CRd*
Double-Precision	**TWOD** *CRd*
Single-Precision	**TWOF** *CRd*

Execution

2 → CRd

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	0	0	type	size
ID			1	1	0	1	1	1	0	1	CRd				

Instruction to '34082

31 29	28 25	24 21	20 16	15 0
ID	1101	1101	CRd	0000 000t s000 0000

Operands

CRd TMS34082 destination register.

Description

TWOx loads the value two (of the appropriate type) into register CRd.

Instruction Type

CEXEC, short

Example

```
TWO RB6
```

This example loads an integer two into TMS34082 register RB6.

Syntax

Type	Syntax
Integer	**VADD**
Double-Precision	**VADDD**
Single-Precision	**VADDF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	1	type	size
ID			0	0	0	0	1	0	0	1	0	0	0	0	0

Instruction to '34082

31 29	28						0
ID	0	0001	0010	0000	0011	111t	s000 0000

Description

Adds the X, Y and Z components of a vector in RB2–RB0 to the X, Y, and Z components of a vector in RA2–RA0.

Implied Operands

RA0 = X1	RB0 = X2
RA1 = Y1	RB1 = Y2
RA2 = Z1	RB2 = Z2

Algorithm

```
RA0 = RA0 + RB0     ; X1 + X2
RA1 = RA1 + RB1     ; Y1 + Y2
RA2 = RA2 + RB2     ; Z1 + Z2
```

Temporary Storage

None

Outputs

The sum of the vectors is stored in RA2–RA0.

Instruction Type

CEXEC, short

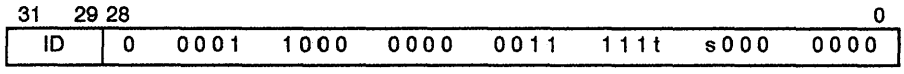
Syntax

Type	Syntax
Integer	**VCROS**
Double-Precision	**VCROSD**
Single-Precision	**VCROSF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	1	type	size
ID			0	0	0	0	1	1	0	0	0	0	0	0	0

Instruction to '34082



Description

Given two vectors V1 in (RA2–RA0) and V2 (RB2–RB0), find their vector cross product (V1 × V2).

Implied Operands

RA0 = X1	RB0 = X2
RA1 = Y1	RB1 = Y2
RA2 = Z1	RB2 = Z2

Algorithm

C = RA1 × RB2	; Y1 × Z2
RA0 = C – (RB1 × RA2)	; (Y1 × Z2) – (Y2 × Z1)
C = RA2 × RB0	; Z1 × X2
RA1 = C – (RB2 × RA0)	; (Z1 × X2) – (Z2 × X1)
C = RA0 × RB1	; X1 × Y2
RA2 = C – (RB0 × RA1)	; (X1 × Y2) – (X2 × Y1)

Temporary Storage C

Temporary Storage C, RB9

Temporary Storage C

Outputs

The vector cross product V3 is stored in registers RA2–RA0.
RA0 = X3
RA1 = Y3
RA2 = Z3

Instruction Type CEXEC, short

Syntax

Type	Syntax
Integer	**VDOT**
Double-Precision	**VDOTD**
Single-Precision	**VDOTF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	1	type	size
ID			0	0	0	0	1	0	1	1	0	0	0	0	0

Instruction to '34082

31 29	28						0
ID	0	0001	0110	0000	0011	111t	s000	0000

Description

Given two vectors V1 in RA2–RA0 and V2 in RB2–RB0, calculate the dot product.

Implied Operands

Vector V1 in RA2–RA0 and vector V2 in RB2–RB0

RA0 = X1	RB0 = X2
RA1 = Y1	RB1 = Y2
RA2 = Z1	RB2 = Z2

Algorithm

```
C    = RA0 × RB0          ; X1 × X2
C    = C + (RA1 × RB1)    ; (X1 × X2) + (Y1 × Y2)
RA4  = C + (RA2 × RB2)    ; (X1 × X2) + (Y1 × Y2) + (Z1 × Z2)
```

Temporary Storage

C

Outputs

The scalar dot product of the two vectors is stored in RA4.

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**VMAG**
Double-Precision	**VMAGD**
Single-Precision	**VMAGF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	1	type	size
ID			0	0	0	0	1	1	0	1	0	0	0	0	0

Instruction to '34082

31 29	28														0
ID	0 0001			1010			0000			0011			111t	s000	0000

Description

Given a vector in RA2–RA0, compute the length of the vector.

Implied Operands

RA0 = X1
RA1 = Y1
RA2 = Z1

Algorithm

```
C    = RA0
RA3  = C × C          ; (X × X)
CT   = RA1
CT   = CT × CT        ; (Y × Y)
RA3  = CT + RA3       ; (X × X) + (Y × Y)
C    = RA2
CT   = C × C          ; (Z × Z)
RA3  = CT + RA3       ; (X × X) + (Y × Y) + (Z × Z)
RA3  = SQRT(RA3)      ; SQRT (X^2 + Y^2 + Z^2)
```

Temporary Storage

C, CT

Outputs

The scalar magnitude of the vector is stored in RA3.

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Double-Precision	**VNORMD**
Single-Precision	**VNORMF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	1	1	size
ID			0	0	0	0	1	1	1	0	0	0	0	0	0

Instruction to '34082

31 29	28														0
ID	0	0001		1100		0000		0011		1111		s000		0000	

Description

Given a vector in RA2–RA0, find the unit length vector that is in the same direction as the given vector.

The integer form of this instruction is not supported.

Implied Operands

RA0 = X0
RA1 = Y0
RA2 = Z0

Algorithm

```
C    = RA0
C    = C × C                ; X0 × X0
CT   = RA1
RA9  = CT × CT              ; Y0 × Y0
RA9  = C + RA9              ; (X0 × X0) + (Y0 × Y0)
C    = RA2
C    = C × C                ; Z0 × Z0
RA9  = C + RA9              ; (X0 × X0) + (Y0 × Y0) + (Z0 × Z0)
C    = SQRT(RA9)            : SQRT (X0^2 + Y0^2 + Z0^2)
RA3  = C                    ; save the magnitude in RA3
C    = 1 / C                ; 1 / magnitude
RA0  = C × RA0
RA1  = C × RA1
RA2  = C × RA2
```

Temporary Storage

C, CT, RA9

Outputs

The unit length vector is stored in registers RA2–RA0.

$RA0 = X0 / (SQRT(X0^2 + Y0^2 + Z0^2))$
$RA1 = Y0 / (SQRT(X0^2 + Y0^2 + Z0^2))$
$RA2 = Z0 / (SQRT(X0^2 + Y0^2 + Z0^2))$
$RA3 = SQRT(X0^2 + Y0^2 + Z0^2)$
$C = 1 / (SQRT (X0^2 + Y0^2 + Z0^2))$

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**VRFLCT**
Double-Precision	**VRFLCTD**
Single-Precision	**VRFLCTF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	1	type	size
ID			0	0	0	0	1	1	1	1	0	0	0	0	0

Instruction to '34082

31 29	28						0
ID	0	0001	1110	0000	0011	111t	s000 0000

Description

The VRFLCT instruction calculates the vector reflection of a vector incident on a surface defined by a normal vector. The normal vector should be normalized before issuing the VRFLCT instruction.

Implied Operands

Vector in RA2—RA0 is the *normal* vector ($X_n\mathbf{i} + Y_n\mathbf{j} + Z_n\mathbf{k}$)). Vector in RB2–RB0 is the *incident* vector ($X_i\mathbf{i} + Y_i\mathbf{j} + Z_i\mathbf{k}$))

RA0 = Xn RB0 = Xi
RA1 = Yn RB1 = Yi
RA2 = Zn RB2 = Zi

Algorithm

```
C  = RA0 × RB0
C  = C + (RA1 × RB1)
C  = C + (RA2 × RB2)      ; scalar dot product in C  (cos(Theta))
C  = C + C                ; C = 2 × cos(Theta)
CT = C × RA0
RB0= CT – RB0             ; Xr = Xn × (2 ×cos(Theta)) – Xi
CT = C × RA1
RB1= CT – RB1             ; Yr = Yn × (2 ×cos(Theta)) – Yi
CT = C × RA2
RB2= CT – RB2             ; Zr = Zn × (2 ×cos(Theta)) – Zi
```

Temporary Storage

C, CT, RA9

Outputs

The reflected vector components x, y and z are stored in RB2–RB0.

RB0 = Xr = Xn × (2 × ((Xn × Xr) + (Yn × Yr) + (Zn × Zr))) – Xi
RB1 = Yr = Yn × (2 × ((Xn × Xr) + (Yn × Yr) + (Zn × Zr))) – Yi
RB2 = Zr = Zn × (2 × ((Xn × Xr) + (Yn × Yr) + (Zn × Zr))) – Zi

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**VSCL** *CRs*
Double-Precision	**VSCLD** *CRs*
Single-Precision	**VSCLF** *CRs*

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	type	size
ID			0	0	0	0	0	1	1	1	1	CRs			

Instruction to '34082

31	29	28	25	24											0
ID		CRs		0	1110		0000		0001		111t		s000		0000

Description

The X, Y, and Z components of a vector in registers RA2–RA0 are multiplied by a scalar in CRs.

Operands

CRs RB register containing the scaling factor. Must be in the RB register file.

Implied Operands

RA0 = X1
RA1 = Y1
RA2 = Z1

Algorithm

RA0 = RA0 × CRs
RA1 = RA1 × CRs
RA2 = RA2 × CRs

Temporary Storage

None

Outputs

The scaled vector is stored in RA2–RA0.
RA0 = X1'
RA1 = Y1'
RA2 = Z1'

Instruction Type

CEXEC, short

Syntax

Type	Syntax
Integer	**VSCL** *Rs_1, CRs*
Double-Precision	**VSCLD** *Rs_1, Rs_2, CRs*
Single-Precision	**VSCLF** *Rs_1, CRs*

'34082 Instruction Words

Integer or Single-Precision:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs_1			
0	1	0	1	1	1	1	type	0	0	0	0	0	0	0	0
ID			0	0	0	0	0	1	1	1	1	CRs			

Double-Precision:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	1	1	1	1	1	1	0	0	R	Rs_2			
ID			0	0	0	0	0	1	1	1	1	CRs			

Instruction to '34082

31 29	28 25	24 0
ID	CRs	0 1110 0000 0101 111t s000 0000

Description

The X, Y, and Z components of a vector in registers RA2-RA0 are multiplied by a scalar in CRs (loaded from Rs).

Operands

Rs_1 TMS34020 source register for the operand (or half of the 64-bit double-precision floating-point operand) to TMS34082

Rs_2 TMS34020 source register for rest of the double-precision operand to TMS34082

CRs Coprocessor RB register to contain the scaling factor. Must be in the RB register file.

Implied Operands

RA0 = X1
RA1 = Y1
RA2 = Z1

Algorithm

$Rs_1 \rightarrow$ CRs
$Rs_2 \rightarrow$ CRs
RA0 = RA0 × CRs
RA1 = RA1 × CRs
RA2 = RA2 × CRs

Temporary Storage

None

Outputs

The scaled vector is stored in RA2–RA0.
RA0 = X1'
RA1 = Y1'
RA2 = Z1'

Instruction Type

CMOVGC, one or two registers

Syntax

Type	Syntax
Integer	**VSCL** **Rs+, CRs*
Double-Precision	**VSCLD** **Rs+, CRs*
Single-Precision	**VSCLF** **Rs+, CRs*

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	0	0	0	transfers	
1	0	0	1	1	1	1	type	size	0	0	R	Rs			
ID			0	0	0	0	0	1	1	1	1	CRs			

Instruction to '34082

31 29	28 25	24													0
ID	CRs	0	1	1	1	0	0	0	0	1	0	0	1	1	1	t	s	0	0	0	0	0	0	0	0

Description

The X, Y, and Z components of a vector in registers RA2–RA0 are multiplied by a scalar in CRs (loaded from memory pointed to by Rs). After each load from memory, Rs is incremented by 32.

Operands

Rs TMS34020 register containing the memory address

CRs Coprocessor RB register to contain the scaling factor. Must be in the RB register file.

Implied Operands

RA0 = X1
RA1 = Y1
RA2 = Z1

Algorithm

*Rs → CRs
Rs + 32 → Rs
*Rs → CRs
Rs + 32 → Rs
RA0 = RA0 × CRs
RA1 = RA1 × CRs
RA2 = RA2 × CRs

Temporary Storage

None

Outputs

The scaled vector is stored in RA2–RA0.
RA0 = X1'
RA1 = Y1'
RA2 = Z1'

Instruction Type

CMOVMC, postincrement, constant count

Syntax

Type	Syntax
Integer	**VSCL** *–*Rs, CRs*
Double-Precision	**VSCLD** *–*Rs, CRs*
Single-Precision	**VSCLF** *–*Rs, CRs*

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	transfers	
1	0	0	1	1	1	1	type	size	0	0	R	Rs			
ID			0	0	0	0	0	1	1	1	1	CRs			

Instruction to '34082

31 29	28 25	24													0
ID	CRs	0	1	1	1	0	0	0	0	1	0	0	1	1	1	t	s	0	0	0	0	0	0	0	0

Description

The X, Y, and Z components of a vector in registers RA2–RA0 are multiplied by a scalar in CRs (loaded from memory pointed to by Rs). Before each load from memory, Rs is decremented by 32.

Operands

Rs TMS34020 register containing the memory address

CRs Coprocessor RB register to contain the scaling factor. Must be in the RB register file.

Implied Operands

RA0 = X1
RA1 = Y1
RA2 = Z1

Algorithm

Rs – 32 → Rs
*Rs → CRs
~~Rs – 32 → Rs~~
~~*Rs → CRs~~
RA0 = RA0 × CRs
RA1 = RA1 × CRs
RA2 = RA2 × CRs

Temporary Storage

None

Outputs

The scaled vector is stored in RA2–RA0.
RA0 = X1'
RA1 = Y1'
RA2 = Z1'

Instruction Type

CMOVMC, predecrement, constant count

Syntax

Type	Syntax
Integer	**VSUB**
Double-Precision	**VSUBD**
Single-Precision	**VSUBF**

'34020 Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	1	1	1	1	1	type	size
ID			0	0	0	0	1	0	1	0	0	0	0	0	0

Instruction to '34082

31 29	28						0
ID	0 0001	0100	0000	0011	111t	s000	0000

Description

Subtract a vector in RB2–RB0 from a vector in RA2–RA0.

Implied Operands

RA0 = X1 RB0 = X2
RA1 = Y1 RB1 = Y2
RA2 = Z1 RB2 = Z2

Algorithm

```
RA0 = RA0 – RB0   ; X1 – X2
RA1 = RA1 – RB1   ; Y1 – Y2
RA2 = RA2 – RB2   ; Z1 – Z2
```

Temporary Storage

None

Outputs

The resulting vector is stored in RA2–RA0.
RA0 = X1'
RA1 = Y1'
RA2 = Z1'

Instruction Type

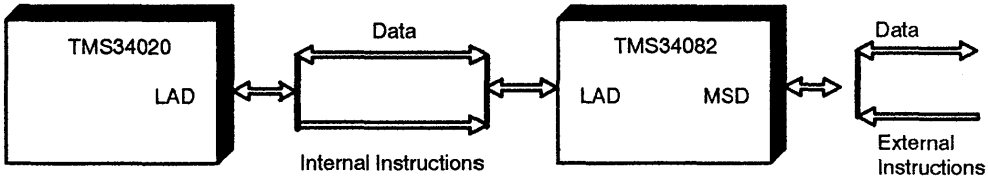
CEXEC, short

Chapter 8

External Instructions

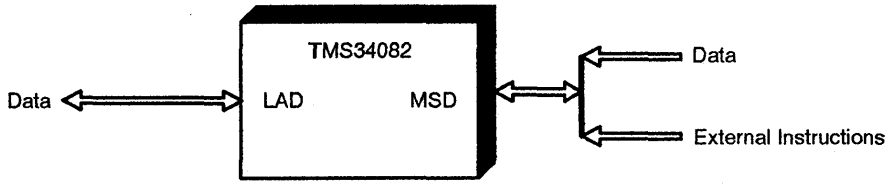
The external instruction set is executed through the MSD port of the TMS34082. The multiplier and ALU may be operated in parallel using these RISC-like instructions. Integer, single-precision, and double-precision floating-point formats are supported. In coprocessor mode, user-defined subroutines constructed out of external instructions may be executed through the MSD port. See Figure 8–1.

Figure 8–1. Source of Instructions for Coprocessor Mode



In host-independent mode, the TMS34082 is controlled by external instructions input on the MSD bus.

Figure 8–2. Instructions in Host-Independent Mode



8.1 Overview

External instructions are 32 bits long and their formats (number, length, and function of fields) depend upon the operations being selected. Separate formats are provided for data transfers to and from the TMS34082, FPU processing, test and branch operations, and subroutine calls.

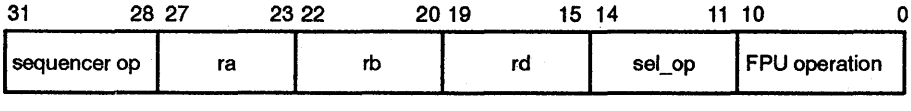
In the host-independent mode, the TMS34082 is controlled by external instructions input on the MSD bus. In the coprocessor mode, the TMS34082 executes user-defined routines (external instructions stored in memory on the MSD bus) by executing a jump to external code. Up to 32 routines may be defined by the user using external instructions in coprocessor mode.

To cause a jump to the external routine, the TMS34020 sends the TMS34082 an instruction with the md field (bits 15–14) set high. The fpuop is the routine number (0–31). The TMS34082 multiplies the routine number by two to get the jump address. This creates a compact jump table where every other address is the starting address of a routine. The remaining memory can then be allocated according to user need. Using every other address as a starting address allows a single-instruction subroutine to be implemented without another jump. For more complex routines, the first instruction in the routine will be a jump to another memory location. In either case, the last instruction should be a return from subroutine or jump to internal instruction address 10FFF (hex). This puts the TMS34082 in an idle state, waiting for the next instruction from the TMS34020. Before the last return from subroutine or jump to internal address 10FFF, the stack (SUBADDR1-0) *must* be cleared. This can be accomplished by setting the stack pointer (bit 31) in both registers to 0. You may wish to save the contents of these registers in external memory before clearing the stack pointers.

8.2 FPU Processing Instruction Format

The largest group of external instructions control FPU operations. These instructions can select operands from input registers, internal feedback, or from the LAD bus (32-bit operations only). Independent ALU or multiplier operations and chained-mode operations (ALU and multiplier acting in parallel) can be coded.

The format for an FPU processing instruction is shown below:



8.2.1 FPU Processing Sequencer Opcodes

Valid sequencer opcodes for this instruction format:

0000	continue
0001	continue with LAD enable for output ($\overline{\text{ALTCH}}$ strobe)
0010	continue with LAD enable for output $\overline{\text{WE}}$ strobe)†

† Permits simultaneous write to a register and to the LAD bus. Writing to the LAD bus during FPU operation requires a 15-ns extension (TMS34082-40) of the clock period when the write is performed.

8.2.2 Operand Selection

Instructions that control FPU operations can select operands from internal registers, internal feedback, or the LAD bus (32-bit operations only). When register addresses are used as sources (ra or rb field), only the lower four bits are used. Most instructions use three operands:

ra is the operand A source address (RA9-0, C, CT)

rb is the operand B source address (RB9-0, C, CT)

rd is the result destination address

When ra (or rb) is set to 1100_2, the A (or B) operand comes from the LAD bus without first being written into a register.

When the CONFIG, COUNTX, or COUNTY register (address 13, 14, or 15) is selected as the ra operand, a one is input to the FPU.

When the SUBADD1, IRAREG, or MIN-MAX/LOOPCT register (address 29, 30, or 31) is selected as the rb operand, a one is input to the FPU.

The sel_op field chooses the operands. When low, sel_op bits 14-11 select the following feedback operands:

bit 14 for ALU feedback to multiplier A input

bit 13 for multiplier feedback to multiplier B input

bit 12 for multiplier feedback to ALU A input

bit 11 for ALU feedback to ALU B input

The sel_op bits allow many different combinations of operands from the register file and feedback registers. Figure 8–1 shows the operands selected for each combination of sel_op bits.

Note: If feedback operands are used, the FPU core output registers must be enabled (PIPES2=0).

Figure 8–3. Operand Selection

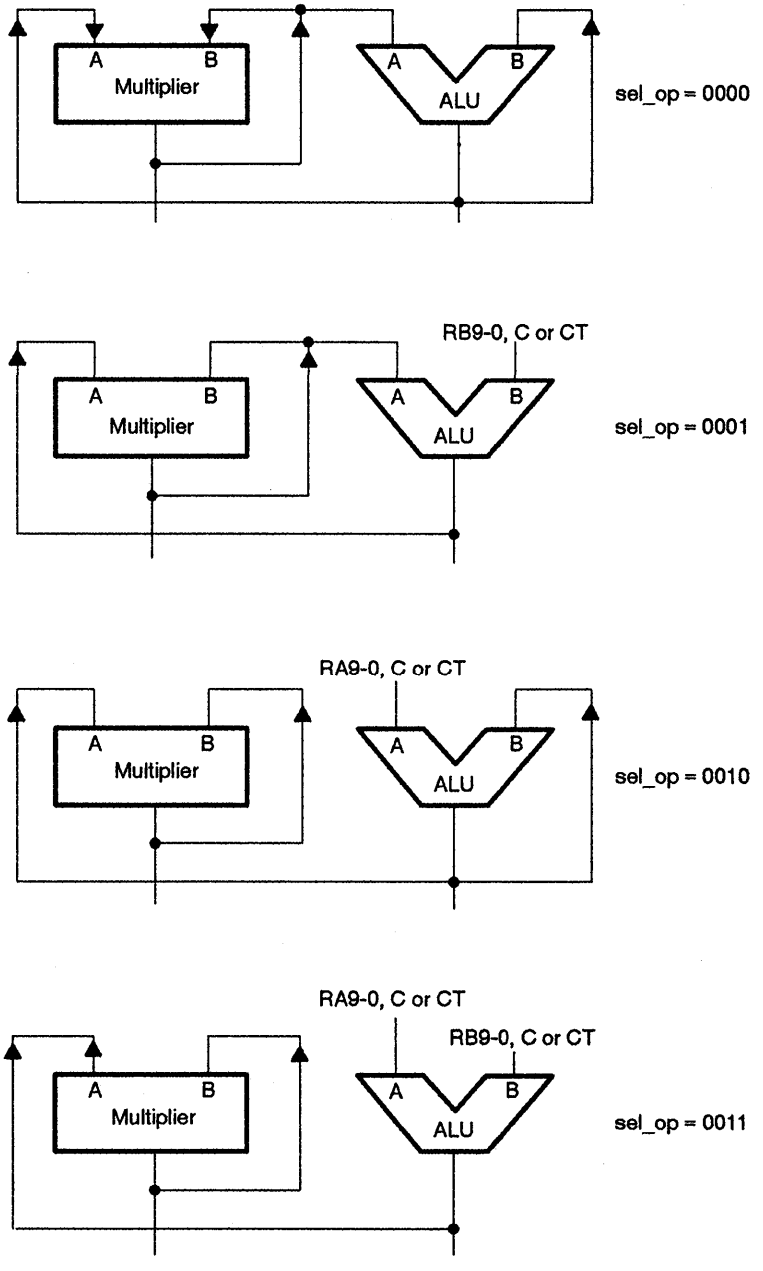


Figure 8–3. Operand Selection (Continued)

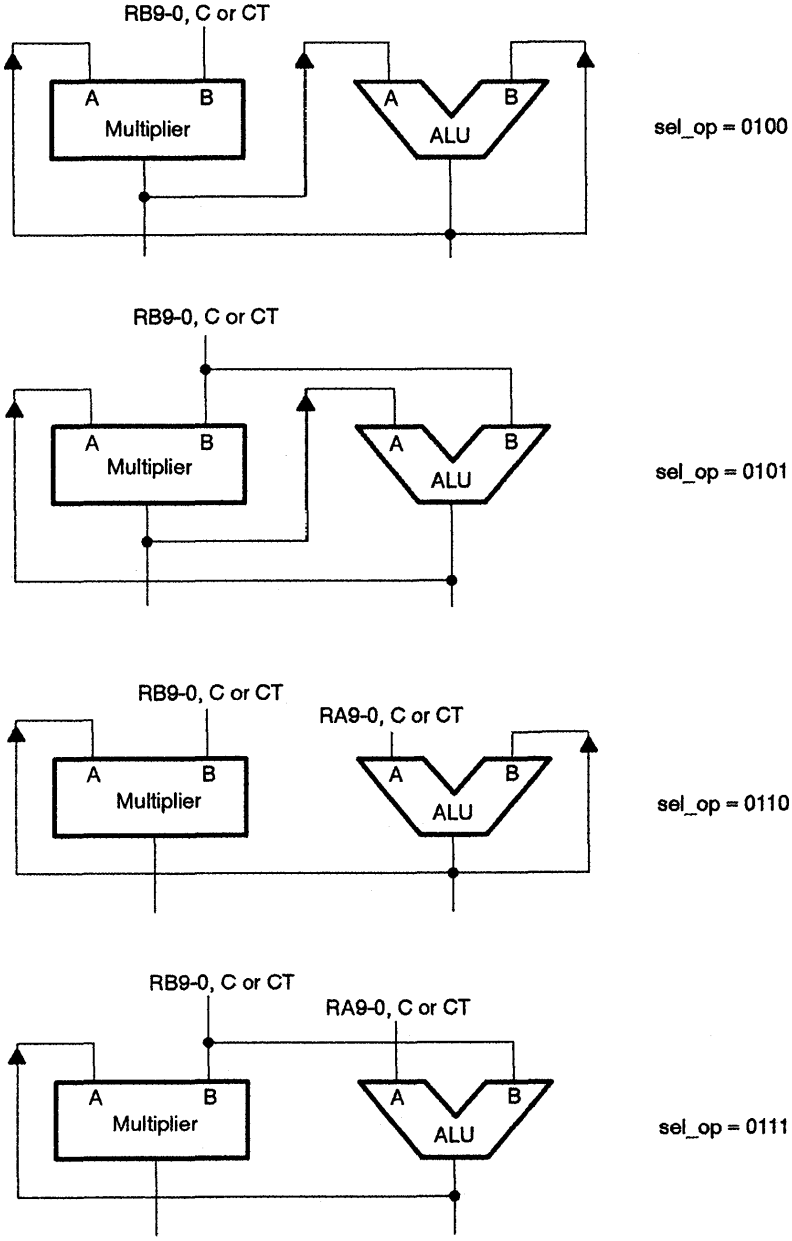


Figure 8–3. Operand Selection (Continued)

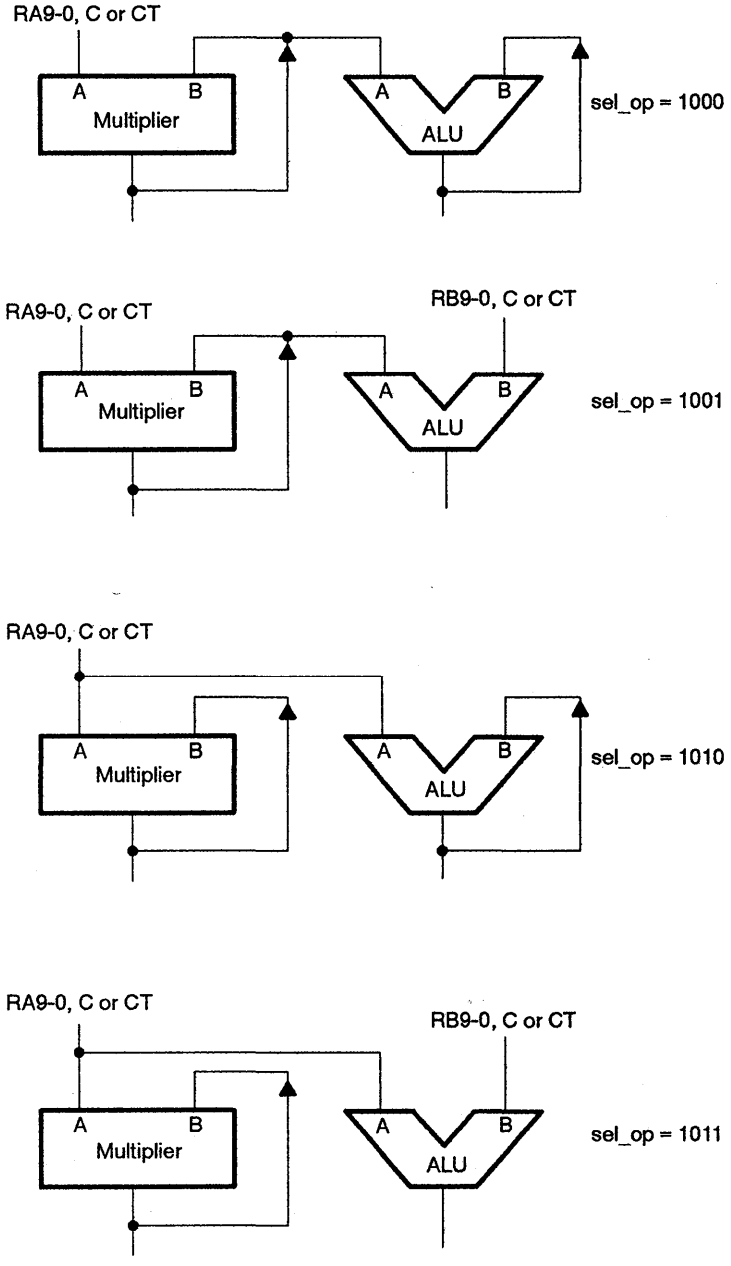
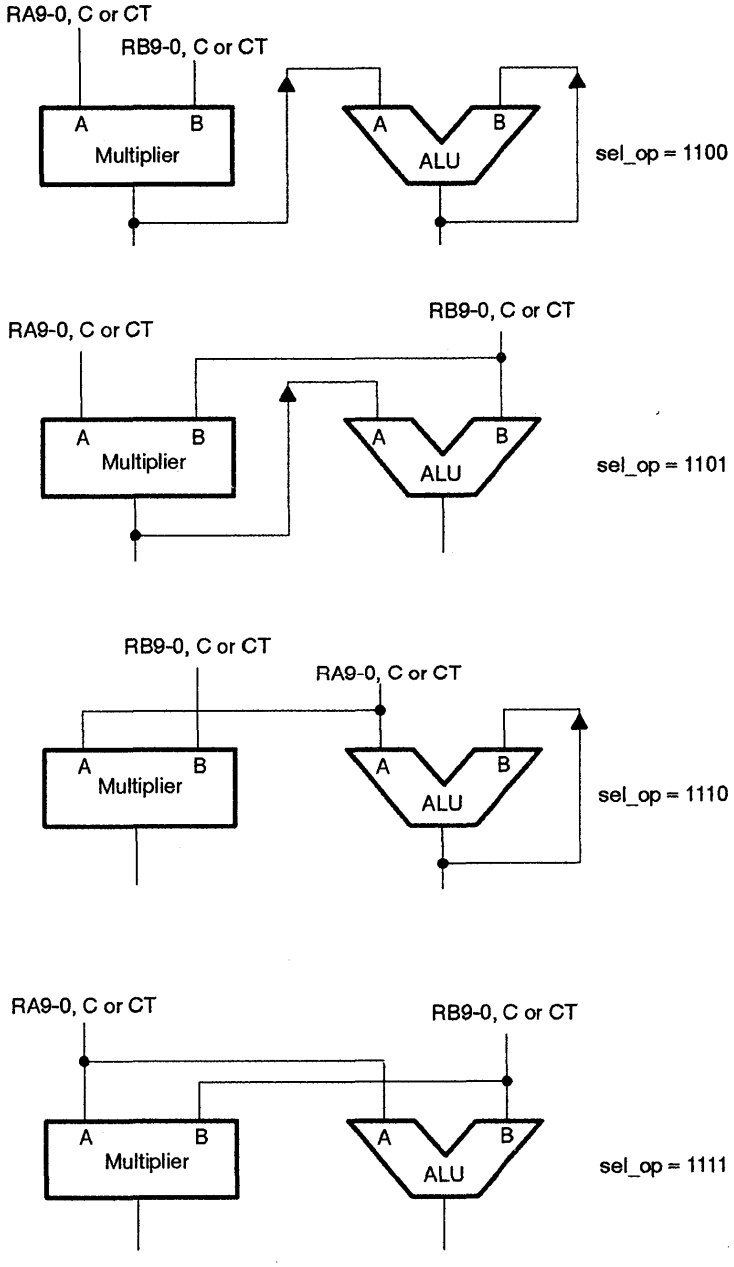


Figure 8–3. Operand Selection (Continued)



8.2.3 FPU Processing Instruction Codes

Instruction bits 10-0 select the multiplier or ALU operation. When the FPU core is busy with multicycle operations (division, square root, or double-precision floating-point multiplication), the FPU stops the sequencer until the FPU is ready for the next operation.

8.3 External Instruction Cycle Counts

Table 8–1 lists external instructions, pipeline settings, and the number of cycles required to complete each routine. The number in parenthesis after each cycle count is the number of cycles before the next operation may begin. For block move operations, *n* specifies the number of words transferred.

Table 8–1. Cycle Counts for External Instructions

Assembler Opcode	Description of Routine	Cycle Counts			
		PIPES2-1 11	PIPES2-1 10	PIPES2-1 01	PIPES2-1 00
ADD	Add A + B	1(1)	2(1)	2(1)	3(1)
AND	Logical AND A, B	1(1)	2(1)	2(1)	3(1)
ANDNA	Logical AND not A, B	1(1)	2(1)	2(1)	3(1)
ANDNB	Logical AND A, not B	1(1)	2(1)	2(1)	3(1)
CJMP	Conditional jump	1(1)	1(1)	1(1)	1(1)
CSJR	Conditional jump to subroutine	1(1)	1(1)	1(1)	1(1)
CMP	Compare A, B	1(1)	2(1)	2(1)	3(1)
COMPL	Pass 1's complement of A	1(1)	2(1)	2(1)	3(1)
DIV	Divide A / B				
	single-precision	8(8)	8(7)	9(7)	9(7)
	double-precision	13(13)	13(12)	15(12)	15(12)
	integer	16(16)	16(15)	17(15)	17(15)
DTOF	Convert from DP to SP	1(1)	2(1)	2(1)	3(1)
DTOI	Convert from DP to integer	1(1)	2(1)	2(1)	3(1)
DTOU	Convert from DP to unsigned integer	1(1)	2(1)	2(1)	3(1)
FTOD	Convert from SP to DP	1(1)	2(1)	2(1)	3(1)
FTOI	Convert from SP to integer	1(1)	2(1)	2(1)	3(1)
FTOU	Convert from SP to unsigned integer	1(1)	2(1)	2(1)	3(1)
ITOD	Convert from integer to DP	1(1)	2(1)	2(1)	3(1)
ITOF	Convert from integer to SP	1(1)	2(1)	2(1)	3(1)
LD	Load n words into register				
	single-precision	n + 1	n + 1	n + 1	n + 1
	double-precision	2n + 1	2n + 1	2n + 1	2n + 1
	integer	n + 1	n + 1	n + 1	n + 1
LDLCT	Load loop counter with value	1(1)	1(1)	1(1)	1(1)
MASK	Set programmable mask	1(1)	1(1)	1(1)	1(1)
MOVA	Move A	1(1)	2(1)	2(1)	3(1)
MOVLM	Move n words from LAD bus to MSD bus				
	single-precision	n + 1	n + 1	n + 1	n + 1
	double-precision	2n + 1	2n + 1	2n + 1	2n + 1
	integer	n + 1	n + 1	n + 1	n + 1

Table 8–1. Cycle Counts for External Instructions (Continued)

Assembler Opcode	Description of Routine	Cycle Counts			
		PIPES2-1 11	PIPES2-1 10	PIPES2-1 01	PIPES2-1 00
MOVML	Move n words from MSD bus to LAD bus				
	single-precision	n + 1	n + 1	n + 1	n + 1
	double-precision	2n + 1	2n + 1	2n + 1	2n + 1
	integer	n + 1	n + 1	n + 1	n + 1
MOVRR	Multiple move, register to register				
	single-precision	n + 1	n + 1	n + 1	n + 1
	double-precision	2n + 1	2n + 1	2n + 1	2n + 1
	integer	n + 1	n + 1	n + 1	n + 1
MULT	Multiply A * B				
	single-precision	1(1)	2(1)	2(1)	3(1)
	double-precision	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
MULT.ADD	Multiply A_1 * B_1, Add A_2 + B_2				
	single-precision	1(1)	2(1)	2(1)	3(1)
	double-precision	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
MULT.NEG	Multiply A_1 * B_1, Subtract 0 – A_2				
	single-precision	1(1)	2(1)	2(1)	3(1)
	double-precision	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
MULT.PASS	Multiply A_1 * B_1, Add A_2 + 0				
	single-precision	1(1)	2(1)	2(1)	3(1)
	double-precision	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
MULT.SUB	Multiply A_1 * B_1, Subtract A_2 – B_2				
	single-precision	1(1)	2(1)	2(1)	3(1)
	double-precision	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
MULT.2SUBA	Multiply A_1 * B_1, Subtract 2–A_2				
	single-precision	1(1)	2(1)	2(1)	3(1)
	double-precision	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
MULT.SUBRL	Multiply A_1 * B_1, Subtract B_2 – A_2				
	single-precision	1(1)	2(1)	2(1)	3(1)
	double-precision	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)

Table 8–1. Cycle Counts for External Instructions (Continued)

Assembler Opcode	Description of Routine	Cycle Counts			
		PIPES2-1 11	PIPES2-1 10	PIPES2-1 01	PIPES2-1 00
NEG	Pass –A	1(1)	2(1)	2(1)	3(1)
NOR	Logical NOR A, B	1(1)	2(1)	2(1)	3(1)
OR	Logical OR A, B	1(1)	2(1)	2(1)	3(1)
PASS	Pass A	1(1)	2(1)	2(1)	3(1)
PASS	Pass B	1(1)	2(1)	2(1)	3(1)
PASS.ADD	Multiply $A_1 * 1$, Add $A_2 + B_2$				
	single-precision	1(1)	2(1)	2(1)	3(1)
	double-precision	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
PASS.NEG	Multiply $A_1 * 1$, Subtract $0-A_2$				
	single-precision	1(1)	2(1)	2(1)	3(1)
	double-precision	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
PASS.PASS	Multiply $A_1 * 1$, Add $A_2 + 0$				
	single-precision	1(1)	2(1)	2(1)	3(1)
	double-precision	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
PASS.SUB	Multiply $A_1 * 1$, Subtract $A_2 - B_2$				
	single-precision	1(1)	2(1)	2(1)	3(1)
	double-precision	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
PASS.2SUBA	Multiply $A_1 * 1$, Subtract $2-A_2$				
	single-precision	1(1)	2(1)	2(1)	3(1)
	double-precision	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
PASS.SUBRL	Multiply $A_1 * 1$, Subtract $B_2 - A_2$				
	single-precision	1(1)	2(1)	2(1)	3(1)
	double-precision	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
RTI	Return from interrupt	1(1)	1(1)	1(1)	1(1)
RTS	Return from subroutine	1(1)	1(1)	1(1)	1(1)
SLL	Logical shift left A by B bits	1(1)	2(1)	2(1)	3(1)
SQRT	Square root of A				
	single-precision	11(11)	11(10)	12(10)	12(10)
	double-precision	16(16)	16(15)	17(15)	17(15)
	integer	20(20)	20(19)	21(19)	21(19)
SRA	Arithmetic shift right A by B bits	1(1)	2(1)	2(1)	3(1)
SRL	Logical shift right A by B bits	1(1)	2(1)	2(1)	3(1)

Table 8–1. Cycle Counts for External Instructions (Continued)

Assembler Opcode	Description of Routine	Cycle Counts			
		PIPES2-1 11	PIPES2-1 10	PIPES2-1 01	PIPES2-1 00
ST	Store n words from register				
	single-precision	n + 1	n + 1	n + 1	n + 1
	double-precision	2n + 1	2n + 1	2n + 1	2n + 1
	integer	n + 1	n + 1	n + 1	n + 1
SUB	Subtract A–B	1(1)	2(1)	2(1)	3(1)
SUBRL	Subtract B–A	1(1)	2(1)	2(1)	3(1)
UTOD	Convert from unsigned integer to DP	1(1)	2(1)	2(1)	3(1)
UTOF	Convert from unsigned integer to SP	1(1)	2(1)	2(1)	3(1)
UWRAPI	Unwrap inexact operand	1(1)	2(1)	2(1)	3(1)
UWRAPR	Unwrap rounded operand	1(1)	2(1)	2(1)	3(1)
UWRAPX	Unwrap exact operand	1(1)	2(1)	2(1)	3(1)
WRAP	Wrap denormalized operand	1(1)	2(1)	2(1)	3(1)
XOR	Logical exclusive OR A, B	1(1)	2(1)	2(1)	3(1)

8.4 General Restrictions for External Instructions

Restrictions that apply to all external instructions are as follows:

Registers C and CT cannot both be used as operands in the same instruction.

Absolute value modifiers are permitted with floating-point operations only.

Integer and floating-point operand types cannot be used in the same operation (except conversions).

Signed and unsigned integer operand types cannot be used in the same operation.

Operands with the LAD bus as the source cannot be specified with a double-precision operand type.

Multiplier and ALU feedback (MULFB and ALUFB) cannot be specified as operands unless the FPU core output registers are turned on (PIPES2 = 1).

Results from chained-mode operations are always of the same type. If one result is double-precision, the other is forced to be also. For example, a multiply/pass operation with double-precision multiplier inputs and a single-precision input for the pass operation will result in two double-precision outputs. Be careful that subsequent instructions have the correct data types when these results are used as input.

8.5 External Assembly Instructions

A detailed explanation of each external instruction is provided on the following pages of this chapter. The instructions are in alphabetical order by their TMS34082 assembler opcode. Table 8–2 is a list of the selectable bit definitions used in this chapter.

Table 8–2. Bit Definitions for External Instructions

Bit Number	Mnemonic	Description
29	e	0 = normal operation, 1 = send output to LAD bus with $\overline{WE}$ strobe
28	h	0 = normal operation, 1 = send output to LAD bus with $\overline{ALTCH}$ strobe
27-24	ra	operand A source address
23-20	rb	operand B source address
19-15	rd	result destination address
14-11	sel_op	operand selection (see subsection 8.2.2)
9–7	type or t	operand format: 000 = single-precision on ra and single-precision on rb 001 = single-precision on ra and double-precision on rb 010 = double-precision on ra and single-precision on rb 011 = double-precision on ra and double-precision on rb 100 = integer (2's complement) on both ra and rb 101 = unsigned integer on both ra and rb
8	pa	precision of ra: 0 = single-precision, 1 = double-precision
7	pb	precision of rb: 0 = single-precision, 1 = double-precision
6	s	output source: 0 = ALU result, 1 = multiplier result
5	a	negate ALU result: 0 = normal ALU result, 1 = negated ALU result
4	va	absolute value of ra: 0 = ra, 1 = | ra|
3	vb	absolute value of rb: 0 = rb, 1 = | rb|
2	vy	absolute value of rd: 0 = rd, 1 = | rd|
2	m	negate multiplier result: 0 = normal multiplier result, 1 = negated multiplier result
2	ny	negate output result: 0 = normal output result, 1 = negated output result
1	wa	wrapped number on ra: 0 = normal format, 1 = wrapped number
0	wb	wrapped number on rb: 0 = normal format, 1 = wrapped number

Syntax **add** *ra.[modifier]type, rb.[modifier]type, rd[.modifiers]*

Execution ra + rb → rd

Instruction Words

31	30	29	28	27 24	23 20	19 15
0	0	e	h	ra	rb	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	t	pa	pb	0	0	va	vb	vy	0	0

Description This instruction places the sum of the values in ra and rb in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra and rb v (absolute value, not valid for integer types)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
v (absolute value, not valid for integer types)
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Example

```
add RA7.vd, RB9.vd, C
```

Syntax **and** *ra.type, rb.type, rd[.modifier]*

Execution ra AND rb → rd

Instruction Words

31	30	29	28	27 24	23 20	19 15
0	0	e	h	ra	rb	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	1	0	t	0	0	0	1	0	0	0

Description This instruction takes the logical AND of ra with rb and places the result in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
i (signed integer)
u (unsigned integer)

Modifiers for ra and rb none

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Restrictions The types for ra and rb must be the same.

Example

```
and LAD.i, ONE.i, CT
```

Syntax **andna** *ra.type, rb.type, rd[.modifier]*

Execution (NOT ra) AND rb → rd

Instruction Words

31	30	29	28	27 24	23 20	19 15
0	0	e	h	ra	rb	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	1	0	t	0	0	0	1	0	1	0

Description This instruction takes the logical AND operation of (NOT ra) with rb and places the result in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
i (signed integer)
u (unsigned integer)

Modifiers for ra and rb none

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{WE}$ strobe)
h (send output to LAD bus, $\overline{ALTCH}$ strobe)

Restrictions The types for ra and rb must be the same.

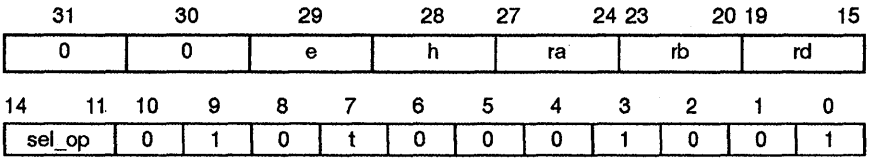
Example

```
andna RA0.u, RB8.u, C.h
```

Syntax **andnb** *ra.type, rb.type, rd[.modifier]*

Execution ra AND (NOT rb) → rd

Instruction Words



Description This instruction takes the logical AND operation of ra with (NOT rb) and places the result in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
i (signed integer)
u (unsigned integer)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Restrictions The types for ra and rb must be the same.

Example
```
andnb C.i, ONE.i, RB1.h
```

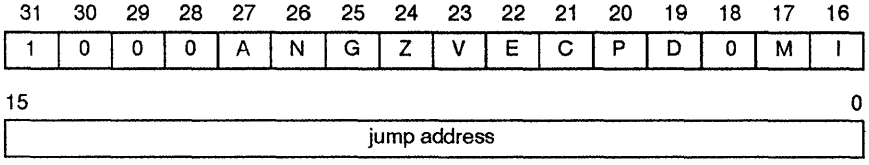
Syntax

cjmp *cond_masks, address*

Execution

If condition is true,
jump address → Program Count
If condition is false,
1+ Program Count → Program Count

Instruction Words



Description

Jump conditional to the specified branch address. During a jump instruction, no FPU operations are performed.

Conditions for cond_masks

Listed below are the jump instruction condition mask bits (enabled when high):

A	Always	C	CC pin
N	Negative	P	Change polarity (for N,G,Z,V,E,C, and D)
G	Greater than	D	Decrement LOOPCT, Jump not zero
Z	Zero	M	Jump indirect using MCADDR
V	Overflow	I	Jump to internal ROM routine
E	ED bit		

An unconditional jump may be done by setting the A mask bit high. If C is enabled, all other jump condition enables except P, M, and I are turned off. Multiple jump conditions are separated by vertical bar, |, and are logically ORed together. The condition mask P changes polarity for each individual bit before the logical OR operation.

Range for address

0x0-0xFFFF

Alternate Opcodes

The following are alternative opcodes recognized by the TMS34082 Software Tool Kit that perform the same instruction as **cjmp**.

Opcode	Description
beq *address*	Branch on equal
bge *address*	Branch on greater than or equal
bgt *address*	Branch on greater than
ble *address*	Branch on less than or equal
blt *address*	Branch on less than
bne *address*	Branch on not equal
boh *address*	Branch on overflow high
bol *address*	Branch on overflow low
br *address*	Branch unconditional
brloop *address*	Branch on loop count
bucch *address*	Branch on CC pin high
buccl *address*	Branch on CC pin low
jmpind	Jump indirect unconditional
jmpindcch	Jump indirect on CC pin high
jmpindccl	Jump indirect on CC pin low
jmpindeq	Jump indirect on equal
jmpindge	Jump indirect on greater than or equal
jmpindgt	Jump indirect on greater than
jmpindle	Jump indirect on less than or equal
jmpindlt	Jump indirect on less than
jmpindne	Jump indirect not equal
jmpindoh	Jump indirect on overflow high
jmpindol	Jump indirect on overflow low

Example

```
cjmp D | P, 0x030
```

This example decrements the value in the LOOPCT register, then checks to see if it is zero. If it is, the jump is taken (since P is set to change polarity). The address output on MSA15-MSA0 is 30 hex.

Syntax

cjsr *cond_masks, address*

Execution

If condition is true,
Program Counter → SABADDRx
jump address → Program Counter
If condition is false,
Program Counter +1 → Program Counter

Instruction Words

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
1	0	0	1	A	N	G	Z	V	E	C	P	D	0	M	I

15	0
jump address	

Description

Jump conditional to the specified subroutine address. During a jump instruction, no FPU operations are performed.

Conditions for cond_masks

Listed below are the jump instruction condition mask bits (enabled when high):

A	Always	C	CC pin
N	Negative	P	Change polarity (for N,G,Z,V,E,C, and D)
G	Greater than	D	Decrement LOOPCT, Jump not zero
Z	Zero	M	Jump indirect using MCADDR
V	Overflow	I	Jump to internal ROM routine
E	ED bit		

An unconditional jump may be done by setting the A mask bit high. If C is enabled and the CC bit is high, all other jump condition enables except P, M, and I are turned off. Multiple jump conditions are separated by vertical bar '|' and are logically ORed together. The condition mask P changes polarity for each individual bit before the logical OR operation.

Range for address

0x0-0xFFFF

Alternate Opcodes

The following are alternative opcodes recognized by the TMS34082 Software Tool Kit that perform the same instruction as **cjsr**.

Opcode	Description
call *address*	Call unconditional
callcch *address*	Call on CC pin high
callccl *address*	Call on CC pin low
calleq *address*	Call on equal
callge *address*	Call on greater than or equal
callgt *address*	Call on greater than
callle *address*	Call on less than or equal
calllt *address*	Call on less than
callne *address*	Call on not equal
calloh *address*	Call on overflow high
callol *address*	Call on overflow
callind	Call indirect unconditional
callindcch	Call indirect on CC pin high
callindccl	Call indirect on CC pin low
callindeq	Call indirect on equal
callindge	Call indirect on greater than or equal
callindgt	Call indirect on greater than
callindle	Call indirect on less than or equal
callindlt	Call indirect on less than
callindne	Call indirect on not equal
callindoh	Call indirect on overflow high
callindol	Call indirect on overflow low
intcall *address*	Internal call unconditional

Example

```
cjsr | C J M
```

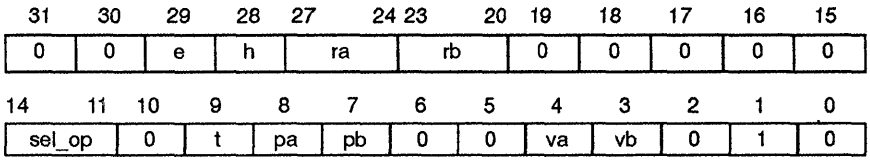
This instruction checks the CC input and jumps to the address in the MCADDR register if CC is high.

Note: 'cjsr A, address' is equivalent to 'call address'

Syntax **cmp** *ra.[modifier]type, rb.[modifier]type*

Execution status flags (ra – rb) → status register

Instruction Words



31	30	29	28	27 – 24	23 – 20	19	18	17	16	15
0	0	e	h	ra	rb	0	0	0	0	0

14 – 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	t	pa	pb	0	0	va	vb	0	1	0

Description This instruction subtracts the value in rb from the value in ra, and sets the status register accordingly.

Sources for ra RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb RB9-RB0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra and rb v (absolute value, not valid for integer types)

Example

```
cmp RA9.vf, CT.vf
```

Syntax **compl** *ra.type, rd[.modifier]*

Execution (NOT ra) → rd

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra	0	0	0	0	rd

14 11	10	9	8 7	6	5	4	3	2	1	0
sel_op	0	1	type	0	1	0	0	0	1	0

Description This instruction takes the 1s complement of ra and places it in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra
i (signed integer)
u (unsigned integer)

Modifiers for ra none

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

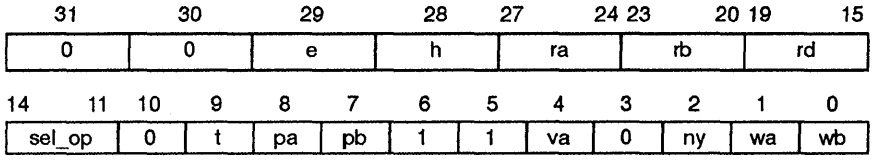
Example

```
compl RA7.i, C.h
```

Syntax **div** *ra.[modifier]type, rb.[modifier]type, rd[.modifiers]*

Execution ra / rb → rd

Instruction Words



Description This instruction takes the result of dividing ra by rb and places it in rd.

Sources for ra
RA9-RA0
C or CT Register
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
ONE (the value one)

Types for ra and rb
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra
v (absolute value, not valid for integer types)
w (wrapped, not valid for integer types)

Modifiers for rb
w (wrapped, not valid for integer types)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
n (negated, not valid for integer types)
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Restrictions Absolute value modifiers, negated result, and wrapped numbers are only permitted with floating-point operations.

Example

```
div ONE.d, CT.f, RA0.e
```

Syntax **dtof** *ra[.modifier], rd[.modifier]*

Execution ra (double-precision) → rd (single-precision)

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra	0	0	0	0	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	0	1	1	0	1	va	0	1	1	0

Description This instruction takes the double-precision floating-point formatted number in ra and converts it to a single-precision floating-point formatted number in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
ONE (the value one)

Types for ra type is implicit in the opcode

Modifiers for ra v (absolute value)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

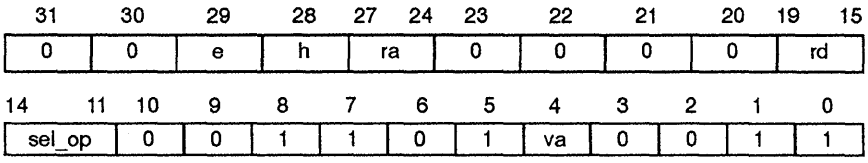
Example

```
dtof RA5.v, C.e
```

Syntax **dtoi** *ra[.modifier], rd[.modifier]*

Execution ra (double-precision) → rd (integer)

Instruction Words



31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra	0	0	0	0	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	0	1	1	0	1	va	0	0	1	1

Description This instruction converts the value in ra from double-precision floating-point format to its integer form and places the result in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
ONE (the value one)

Types for ra type is implicit in the opcode

Modifiers for ra v (absolute value)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Example

```
dtoi RA4. v, RA2
```

Syntax **dtou** *ra[.modifier], rd[.modifier]*

Execution ra (double-precision → rd (unsigned integer)

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra	0	0	0	0	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	0	1	1	0	1	va	0	1	1	1

Description This instruction takes a double-precision floating-point formatted value in ra and converts it to an unsigned integer.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
ONE (the value one)

Types for ra type is implicit in the opcode

Modifiers for ra v (absolute value)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Example

```
dtou RA7.v, C
```

Syntax **ftod** *ra[.modifier], rd[.modifier]*

Execution ra (single-precision) → rd (double-precision)

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra	0	0	0	0	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	0	0	0	0	1	va	0	1	1	0

Description This instruction converts the value in ra from single-precision floating-point to double-precision floating-point and places it in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
ONE (the value one)

Types for ra type is implicit in the opcode

Modifiers for ra v (absolute value)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Example

```
ftod LAD, CT.h
```

Syntax **ftoi** *ra[.modifier], rd[.modifier]*

Execution ra (single-precision) → rd (integer)

Instruction Words

31	30	29	28	27 24	23	22	21	20 19 15
0	0	e	h	ra	0	0	0	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	0	0	0	0	1	va	0	0	1	1

Description This instruction converts from a single-precision floating-point format in ra into the integer format and places the result in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
ONE (the value one)

Types for ra type is implicit in the opcode

Modifiers for ra v (absolute value)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS,CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Example

```
ftoi MULFB, C.h
```

Syntax **ftou** *ra[.modifier], rd[.modifier]*

Execution ra (single-precision) → rd (unsigned integer)

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra	0	0	0	0	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	0	0	0	0	1	va	0	1	1	1

Description This instruction take a value in single-precision floating-point format and converts it to an unsigned integer, placing it in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
ONE (the value one)

Types for ra type is implicit in the opcode

Modifiers for ra v (absolute value)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Example

```
ftou CT, RB5.h
```

Syntax **itod** *ra, rd[.modifier]*

Execution ra (integer) → rd (double-precision)

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra	0	0	0	0	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	0	1	1	0	1	0	0	0	1	0

Description This instruction takes the integer value in ra and places it in rd in double-precision floating-point format.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
ONE (the value one)

Types for ra type is implicit in the opcode

Modifiers for ra none

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{WE}$ strobe)
h (send output to LAD bus, $\overline{ALTCH}$ strobe)

Example

```
itod RA2, RA6.e
```

Syntax **itof** *ra, rd[.modifier]*

Execution ra (integer) → rd (single-precision)

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra	0	0	0	0	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	0	0	0	0	1	0	0	0	1	0

Description This instruction converts the value in ra from integer form to single-precision floating-integer and places the result in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
ONE (the value one)

Types for ra type is implicit in the opcode

Modifiers for ra none

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

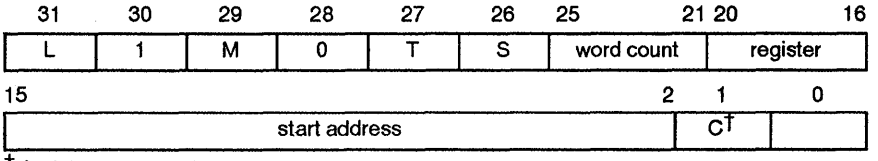
Example

```
itof ONE, RB0.h
```

Syntax

ld *reg.type, address, count*

Instruction Words



† for LAD moves only.

Description

During a move instruction no FPU operation is performed. Register control logic for move instructions counts sequentially from the beginning register address, with the exception that the C and CT registers are omitted from the count. The entire register file acts like a ring buffer during the move instruction. The C and CT registers are not accessible to moves. It is illegal to use the C or CT register address as the starting address for a move instruction.

T (Type) and S (Size) give the format of the numbers
T 0 = integer, 1 = floating point
S 0 = 32 bits 1 = 64 bits
Note: Setting TS = 01 is reserved

Word count is the number of operands to be moved (n). A count of 0 will move 256 items. The beginning register address is stored in the register field, and the beginning memory address is the start address field (bits 15-0).

An indirect move is designated by selecting MCADDR as the address. The M bit will be set low and the 16 low-order bits are then disregarded. The starting address in memory comes from the MCADDR register.

To move data from the LAD bus, LAD is selected as the address. The L bit will be set high, and the low-order 16 bits are set to 0. An address of 'COINT' will load data from the LAD bus and set $\overline{\text{COINT}}$ low for the cycles the load is executing. (C will be set high in the instruction word.) This option is valid for host-independent mode only.

Destinations for reg

RA9-RA0
RB9-RB0
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Types for reg

f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Sources for address

0x0-0xFFFF
MCADDR, LAD, COINT

Range for count

0-31

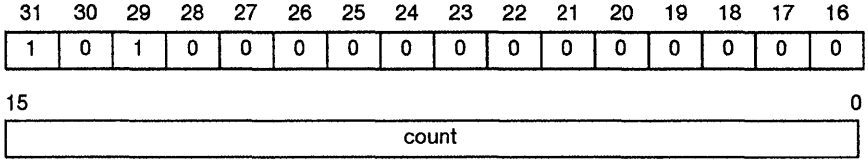
Example

```
ld CONFIG.i, 0x100, 1
ld RA0.i, MCADDR, 3
ld RB1.i, LAD, 3
```

Syntax **ldlct** *count*

Execution *count* → LOOPCT

Instruction Words



Description This instruction loads the LOOPCT register with the value specified by count. If the register is loaded with 0, the loop would execute 64K times.

Range for count 0x0-0xFFFF

Example

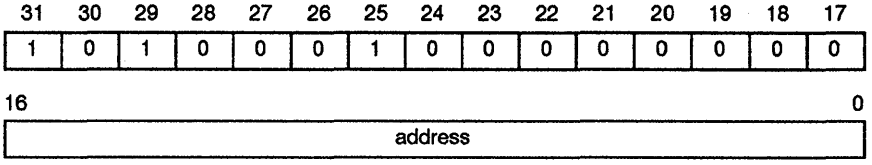
```
ldlct 0x0A
```

This example loads the LOOPCT register with 10 (A hex).

Syntax **ldmcaddr** *address*

Execution *address* → MCADDR

Instruction Words



Description This instruction loads the indirect address (MCADDR) register with the value specified by *count*. This is a 17-bit value; the most significant bit selects between code and data space.

Range for address 0x0-0x1FFFF

Example

```
ldmcaddr 0x0A
```

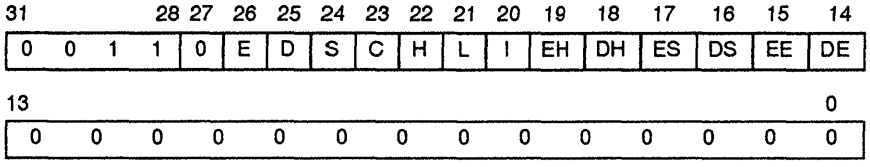
This example loads the MCADDR register with 10 (A hex).

Syntax

mask *prog_mask*

Execution

Instruction Words



Description

This instruction enables/disables interrupts, sets/clears programmable bits, and forces software interrupts. Multiple bits may be set by placing a vertical bar 'I' between each symbol.

Functions for prog_mask

When high, the bits below perform the functions described:

E	Restore interrupt mask (INTENED, INTENSW, INTENHW)
D	Save interrupt mask and disable interrupts
S	Set $\overline{\text{COINT}}$ high (set interrupt output to host)
C	Set $\overline{\text{COINT}}$ low (clear interrupt output to host)
H	Set CORDY high (host-independent mode only)
L	Set CORDY low (host-independent mode only)
I	Force software interrupt
EH	Enable hardware interrupt ($\overline{\text{INTR}}$ input)
DH	Disable hardware interrupt ($\overline{\text{INTR}}$ input)
ES	Enable software interrupt
DES	Disable software interrupt
EE	Enable ED interrupt
DE	Disable ED interrupt

Restrictions

E and D, S and C, H and L, EH and DH, ES and DS, EE and DE may not be used in pairs.

E and D may not be used with I, EH, DH, ES, DS, EE, and DE.

I may not be used with EH, DH, ES, DS, EE and DE.

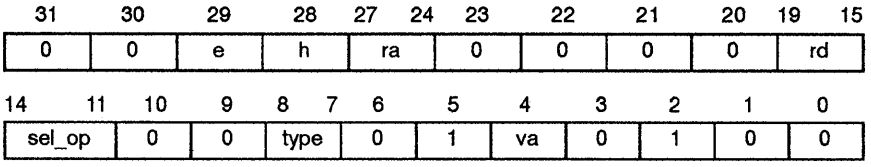
Example

```
mask EH | ES
```

Syntax **mova** *ra.[modifier]type, rd[.modifier]*

Execution ra → rd (no status flag set)

Instruction Words



Description This instruction copies the value in ra and places it in rd without setting status flags. NANs are not detected or changed to the standard format.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra
f (single-precision floating-point)
d (double-precision floating-point)

Modifiers for ra
v (absolute value)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

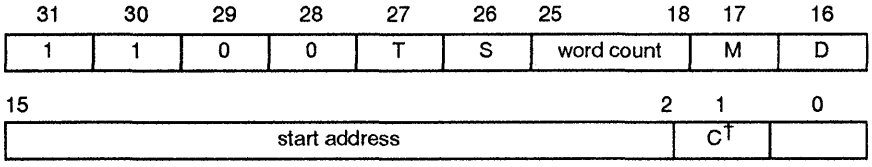
Example

```
mova CT.vf, RA7.e
```

Syntax

movlm *address.type, count[, memory_type]*

Instruction Words



† for indirect moves only.

Description

Each instruction can transfer up to 256 items that are 1 or 2 words long. During a move instruction, no FPU operation is performed.

T (Type) and S (Size) determine the number format.
T 0 = integer, 1 = floating point
S 0 = 32 bits, 1 = 64 bits
Note: Setting TS = 01 is reserved

Word count is the number of operands to be moved (n). A word count of 0 will move 256 items. 512 32-bit values may be moved by setting count=0 and specifying double-precision format. The beginning memory address (for MSD) is the start address field.

An indirect MOVE is selected by using 'MCADDR' for the address. Bit 17 (M) will be set high. The starting address is found in the indirect address register (MCADDR). When M is high, the 16 low-order bits are disregarded, with the exception of bit 1. Choosing 'COINT' as the address will set the C bit high. The $\overline{\text{COINT}}$ output will be low during the cycles the move is executing. The MCADDR register stores the starting address. This option is only valid for host-independent mode.

If bit 16 (D) is high, data space is used as the destination. If the bit is low, code space is used.

Valid memory types are CODE (D=0) and DATA (D=1). The default value, if none is specified, is CODE. If 'MCADDR' or 'COINT' is the address, the memory type must NOT be specified (bit 16 of the MCADDR register selects the memory type).

Destination addresses

0x0-0xFFFF
MCADDR, COINT

Types for reg

f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Range for count

0-255

Types for memory_type

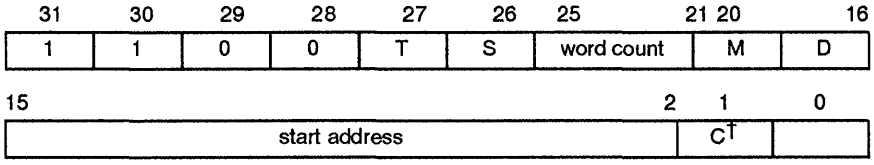
CODE
DATA

Example

```
movlm _vec1.i, 2, DATA
movlm MCADDR.f, 2
movlm COINT.i, 2
```

Syntax **movml** *address.type, count[, count[, memory_type]*

Instruction Word



† for indirect moves only.

Description

Each instruction can transfer up to 256 items that are 1 or 2 32-bit words long. During a move instruction, no FPU operation is performed.

Valid sequencer opcode for this instruction format 1101 move n words from MSD to LAD.

T (Type) and S (Size) determine the number format
T 0 = integer, 1 = floating point
S 0 = 32 bits, 1 = 64 bits
Note: Setting TS = 01 is reserved

Word count is the number of operands to be moved (n). A word count of 0 will move 256 items. 512 32-bit values may be moved by setting count=0 and specifying double-precision format. The beginning memory address (for MSD) is the start address field.

An indirect MOVE is selected by using 'MCADDR' for the address. Bit 17 (M) will be set high. The starting address is found in the indirect address register (MCADDR). When M is high, the 16 low-order bits are disregarded, with the exception of bit 1. Choosing 'COINT' as the address will set the C bit high. The $\overline{\text{COINT}}$ output will be low during the cycles the move is executing. The MCADDR register stores the starting address. This option is only valid for host-independent mode.

If bit 16 (D) is high, data space is used as the source. If the bit is low, code space is used.

Valid memory types are CODE (D=0) and DATA (D=1). The default value, if none is specified, is CODE. If 'MCADDR' or 'COINT' is the address, the memory type must NOT be specified (bit 16 of the MCADDR register selects the memory type).

Sources for address

0x0-0xFFFF
MCADDR, COINT

Types for reg

f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Range for count

0-255

Types for memory_type

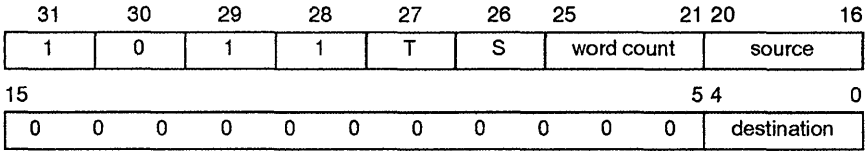
CODE
DATA

Example

```
movml _vec2.f, 3, DATA
movml MCADDR.f, 3
movml COINT.i, 3
```

Syntax **movrr** *srscreg.type, dstreg, count*

Instruction Word



Description

During a move, no FPU operations are performed. Register control logic for move instructions counts sequentially from the beginning register address, with the exception that the C and CT registers are omitted from the count. The entire register file acts like a ring buffer during the move instruction. The C and CT registers are not accessible to moves. It is illegal to use the C or CT register address as the starting address for a move instruction.

T (Type) and S (Size) give the format of the numbers
T 0 = integer, 1 = floating point
S 0 = 32 bits, 1 = 64 bits
Note: Setting TS = 01 is reserved

Word count is the number of operands (n) to be moved. A count of 0 will move 256 registers. The source and destination fields are the beginning register addresses. The source is the starting source register and destination is the starting destination address.

Sources for srcreg

RA9-RA0
RB9-RB0
STATUS,CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Types for srcreg

f (single-precision floating-point
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Destinations for dstreg

RA9-RA0
RB9-RB0
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Range for count

0-31

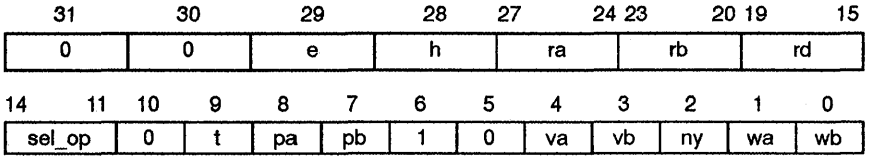
Example

```
movrr RA3.f, RB0, 3
```

Syntax **mult** *ra.[modifier]type, rb.[modifier]type, rd[.modifier]*

Execution ra × rb → rd

Instruction Words



Description This instruction takes the product of ra and rb and places it in rd.

Sources for ra
RA9-RA0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
n (negated, not valid for integer types)
e (send output to LAD bus, $\overline{WE}$ strobe)
h (send output to LAD bus, $\overline{ALTCH}$ strobe)

Restrictions Feedback registers (ALUFB or MULFB) may not be used as operands for double-precision multiplies.

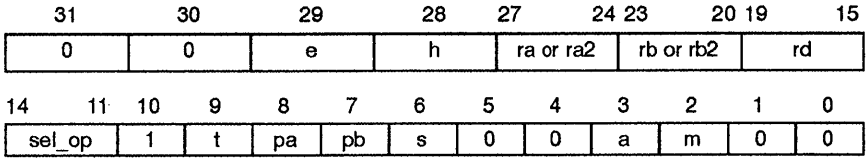
Example

```
mult RA0.f,C.f, CT
```

Syntax **mult.add** *ra.type, ra2, rb.type, rb2, rd[.modifier], output_source*

Execution ra × rb → rd or MULFB; ra2 + rb2 → rd or ALUFB

Instruction Words



31	30	29	28	27 24	23 20	19 15
0	0	e	h	ra or ra2	rb or rb2	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	1	t	pa	pb	s	0	0	a	m	0	0

Description This chained-mode instruction places the product of the values of ra and rb in either rd or MULFB, and concurrently places the sum of the next values from ra and rb into rd or ALUFB.

Sources for ra
RA9-RA0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra and rb none

Sources for ra2
RA9-RA0
C or CT register
MULFB (Multiplier feedback)
LAD (immediate data from LAD bus)
ONE (the value one)

Sources for rb2
RB9-RB0
C or CT register
ALUFB (ALU feedback)
LAD (immediate)
ONE (the value one)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADDO, SUBADM, IRAREG, LOOPCT

Modifiers for rd

a (negate ALU result, valid only for chained mode noninteger types)
m (negate multiplier result, valid only for chained mode noninteger types)
e (send output to LAD bus, $\overline{WE}$ strobe)
h (send output to LAD bus, $\overline{ALTCH}$ strobe)

Output_sources

ALU
MULT

Restrictions

If ra2 is specified, then at least one feedback source must be used (either ra or ra2). If rb2 is specified, then at least one feedback source must be used (either rb or rb2).

Feedback registers (ALUFB or MULFB) may not be used as operands for double-precision multiplies.

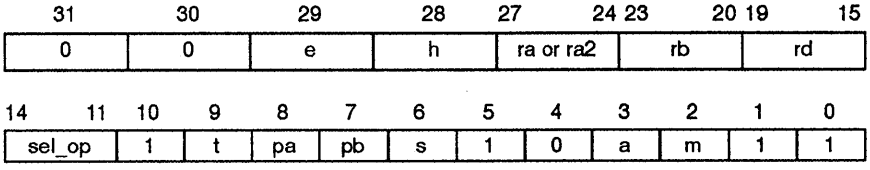
Example

```
mult.add RA2.f, lad2, RB7.u. ALUFB2, CT.a, ALU
```

Syntax **mult.neg** *ra.type, ra2. rb.type, rd[.modifier],output_source*

Execution ra × rb → rd or MULFB; 0 – ra2 → rd or ALUFB

Instruction Words



Description The chained-mode instruction places the product of values of ra and rb in either rd or the multiplier feedback, and concurrently subtracts the value of ra2 from 0 and places the result into either rd or the ALU feedback.

Sources for ra
RA9-RA0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra and rb none

Sources for ra2
RA9-RA0
C or CT register
MULFB (Multiplier feedback)
LAD (immediate data from LAD bus)
ONE (the value one)

Sources for rb2
RB9-RB0
C or CT register
ALUFB (ALU feedback)
LAD (immediate)
ONE (the value one)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT

Modifiers for rd

a (negate ALU result, valid only for chained mode noninteger types)
m (negate multiplier result, valid only for chained mode noninteger types)
e (send output to LAD bus, $\overline{WE}$ strobe)
h (send output to LAD bus, $\overline{ALTCH}$ strobe)

Output_sources

ALU
MULT

Restrictions

If ra2 is specified then at least one feedback source must be used (either ra or ra2). If rb2 is specified then at least one feedback source must be used (either rb or rb2).

Feedback registers (ALUFB or MULFB) may not be used as operands for double-precision multiplies.

Example

```
mult.neg RA1.f, LAD2, RB6d, RB0, MULT
```

Syntax **mult.pass** *ra.type, ra2, rb.type, rd[.modifier], output_source*

Execution ra × rb → rd or MULFB; ra2 + 0 → rd or ALUFB

Instruction Words

31	30	29	28	27 — 24	23 — 20	19 — 15
0	0	0	0	ra or ra2	rb	rd

14 — 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	1	t	pa	pb	s	1	0	a	m	0	0

Description This chained-mode instruction places the product of a value of ra and rb in rd or the multiplier feedback, and concurrently places the sum of the value of ra2 and 0 into either rd of the ALU feedback.

Sources for ra
RA9-RA0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Modifiers for ra and rb none

Sources for ra2
RA9-RA0
C or CT register
MULFB (Multiplier feedback)
LAD (immediate data from LAD bus)
ONE (the value one)

Sources for rb2
RB9-RB0
C or CT register
ALUFB (ALU feedback)
LAD (immediate)
ONE (the value one)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT

Modifiers for rd
a (negate ALU result, valid only for chained mode noninteger types)
m (negate multiplier result, valid only for chained mode noninteger types)
e (send output to LAD bus, $\overline{WE}$ strobe)
h (send output to LAD bus, $\overline{ALTCH}$ strobe)

Output_sources
ALU
MULT

Restrictions

If ra2 is specified then at least one feedback source must be used (either ra or ra2). If rb2 is specified then at least one feedback source must be used (either rb or rb2).

Feedback registers (ALUFB or MULFB) may not be used as operands for double-precision multiplies.

Example

```
mult.pass RA4.f, C2, RB6.d, CT.a, ALU
```

Syntax

mult.sub *ra.type, ra2, rb.type, rb2, rd[.modifier], output_source*

Execution

ra × rb → rd or MULFB; ra2 – rb2 → rd or ALUFB

Instruction Words

31	30	29	28	27 – 24	23 – 20	19 – 15
0	0	e	h	ra or ra2	rb or rb2	rd

14 – 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	1	t	pa	pb	s	0	0	a	m	0	1

Description

This chained-mode instruction places the product of the values of ra and rb in either rd or MULFB, and concurrently places the difference of the values from ra2 and rb2 into rd or ALUFB.

Sources for ra

RA9-RA0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb

RB9-RB0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb

f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra and rb

none

Sources for ra2

RA9-RA0
C or CT register
MULFB (Multiplier feedback)
LAD (immediate data from LAD bus)
ONE (the value one)

Sources for rb2

RB9-RB0
C or CT register
ALUFB (ALU feedback)
LAD (immediate)
ONE (the value one)

Destinations for rd

RA9-RA0
RB9-RB0
C or CT

Modifiers for rd

a (negate ALU result, valid only for chained mode noninteger types)
m (negate multiplier result, valid only for chained mode non-nteger types)
e (send output to LAD bus, $\overline{WE}$ strobe)
h (send output to LAD bus, $\overline{ALTCH}$ strobe)

Output_sources

ALU
MULT

Restrictions

If ra2 is specified then at least one feedback source must be used (either ra or ra2). If rb2 is specified then at least one feedback source must be used (either rb or rb2).

Feedback registers (ALUFB or MULFB) may not be used as operands for double-precision multiplies.

Example

```
mult.sub RA8.d, MULFB2, RB4.d, ALUFB2, RA9.m,MULT
```

Syntax **mult.2suba** *ra.type, ra2. rb.type, rd[.modifier], output_source*

Execution ra × rb → rd or MULFB; 2 – ra2 → rd or ALUFB

Instruction Words

31	30	29	28	27 – 24	23 – 20	19 – 15
0	0	e	h	ra or ra2	rb	rd

14 – 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	t	pa	pb	s	0	0	a	m	1	0

Description This chained-mode instruction places the product of ra and rb in rd or in MUL feedback, and concurrently subtracts the value of ra2 from 2 and places the result in rd or in the ALU feedback.

Sources for ra
RA9-RA0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra and rb none

Sources for ra2
RA9-RA0
C or CT register
MULFB (Multiplier feedback)
LAD (immediate data from LAD bus)
ONE (the value one)

Sources for rb2
RB9-RB0
C or CT register
ALUFB (ALU feedback)
LAD (immediate)
ONE (the value one)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT

Modifiers for rd

a (negate ALU result, valid only for chained mode noninteger types)
m (negate multiplier result, valid only for chained mode noninteger types)
e (send output to LAD bus, $\overline{WE}$ strobe)
h (send output to LAD bus, $\overline{ALTCH}$ strobe)

Output_sources

ALU
MULT

Restrictions

If ra2 is specified then at least one feedback source must be used (either ra or ra2). If rb2 is specified then at least one feedback source must be used (either rb or rb2).

Feedback registers (ALUFB or MULFB) may not be used as operands for double-precision multiplies.

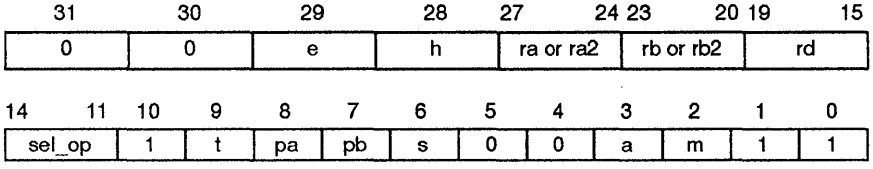
Example

```
mult.2suba RA3.i, LAD2, RB1.u, RA0.e, ALU
```

Syntax **mult.subrl** *ra.type, ra2. rb.type, rb2, rd[.modifier], output_source*

Execution ra × rb → rd or MULFB; rb2 – ra2 → rd or ALUFB

Instruction Words



Description This instruction places the product of a value in ra and rb in either rd or multiplier feedback and concurrently subtracts the value of ra2 from rb2 and places the result in either rd or the ALU feedback.

Sources for ra
RA9-RA0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra and rb none

Sources for ra2
RA9-RA0
C or CT register
MULFB (Multiplier feedback)
LAD (immediate data from LAD bus)
ONE (the value one)

Sources for rb2
RB9-RB0
C or CT register
ALUFB (ALU feedback)
LAD (immediate)
ONE (the value one)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT

Modifiers for rd

a (negate ALU result, valid only for chained mode noninteger types)
m (negate multiplier result, valid only for chained mode noninteger types)
e (send output to LAD bus, $\overline{WE}$ strobe)
h (send output to LAD bus, $\overline{ALTCH}$ strobe)

Output_sources

ALU
MULT

Restrictions

If ra2 is specified then at least one feedback source must be used (either ra or ra2). If rb2 is specified then at least one feedback source must be used (either rb or rb2).

Feedback registers (ALUFB or MULFB) may not be used as operands for double-precision multiplies.

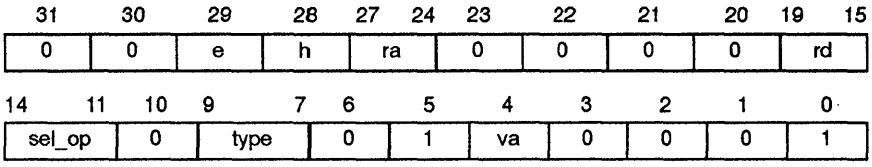
Example

```
mult.subrl MULFB.d, LAD2, R6.d, ONE2, C.a, MULT
```

Syntax **neg** *ra.[modifier]type, rd[.modifier]*

Execution –ra → rd

Instruction Words



Description This instruction negates the value in ra and places it in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra v (absolute value, not valid for integer types)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Example

```
neg RA4.vd, CT.e
```

Syntax

nop

Instruction Words

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0

Description

This instruction performs no operation. If FPU core output registers are enabled (PIPES2=0), the output registers hold their previous value.

This instruction may be used if the TMS34082 is idle, or to wait for a previous instruction to finish.

Syntax **nor** *ra.type, rb.type, rd[.modifier]*

Execution ra NOR rb → rd

Instruction Words

31	30	29	28	27 24	23 20	19 15
0	0	e	h	ra	rb	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	1	0	t	0	0	0	1	0	1	1

Description This instruction takes the logical NOR of ra with rb and places the result in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
i (signed integer)
u (unsigned integer)

Modifiers for ra and rb none

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Restrictions The types for ra and rb must be the same.

Example

```
nor CT.u, LAD.u, RB9.e
```

Syntax **or** *ra.type, rb.type, rd[.modifier]*

Execution ra OR rb → rd

Instruction Words

31	30	29	28	27 24	23 20	19 15
0	0	e	h	ra	rb	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	1	0	t	0	0	0	1	1	0	0

Description This instruction takes the logical OR of ra with rb and places the result in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
i (signed integer)
u (unsigned integer)

Modifiers for ra and rb none

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Restrictions The types for ra and rb must be the same.

Example

```
or MULFB.i, LAD.i, CT.e
```

Syntax **pass** *ra.[modifier]type, rd[.modifier]*

Execution ra → rd

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra	0	0	0	0	rd

14 11	10	9 7	6	5	4	3	2	1	0
sel_op	0	type	0	1	va	0	0	0	0

Description This instruction copies the value in ra to rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra v (absolute value, not valid for integer types)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

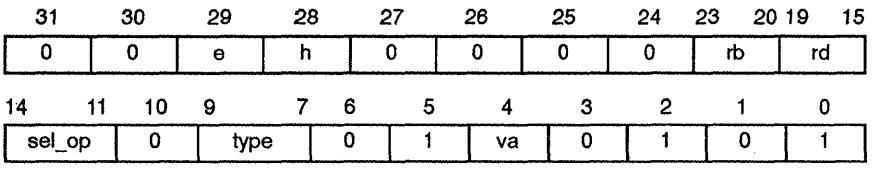
Example

```
pass RA5.vf, CT
```

Syntax **pass** *rb.[modifier]type, rd[.modifier]*

Execution rb → rd

Instruction Words



Description This instruction copies the value in rb to rd.

Sources for rb
RB9-RB0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for rb
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for rb v (absolute value, not valid for integer types)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

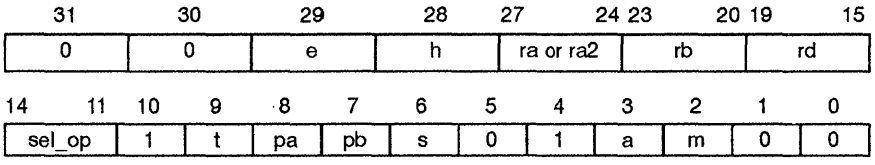
Execution

```
pass RB2.i, CT
```

Syntax **pass.add** *ra.type, ra2. rb.type, rd[.modifier], output_source*

Execution ra × 1 → rd or MULFB; ra2 + rb → rd or ALUFB

Instruction Words



Description This chained-mode instruction places the product of the values of ra and 1 in either rd or MULFB, and concurrently places the sum of the values from ra2 and rb into rd or ALUFB.

Sources for ra
RA9-RA0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra and rb none

Sources for ra2
RA9-RA0
C or CT register
MULFB (Multiplier feedback)
LAD (immediate data from LAD bus)
ONE (the value one)

Sources for rb2
RB9-RB0
C or CT register
ALUFB (ALU feedback)
LAD (immediate)
ONE (the value one)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd

a (negate ALU result, valid only for chained mode noninteger types)
m (negate multiplier result, valid only for chained mode noninteger types)
e (send output to LAD bus, $\overline{WE}$ strobe)
h (send output to LAD bus, $\overline{ALTCH}$ strobe)

Output_sources

ALU
MULT

Restrictions

If ra2 is specified then at least one feedback source must be used (either ra or ra2).

If rb2 is specified then at least one feedback source must be used (either rb or rb2).

Example

```
pass.add RA.d, MULFB2, RB9.f, CT,ALU
```

Syntax **pass.neg** *ra.type, ra2. rd[.modifier], output_source*

Execution ra × 1 → rd or MULFB; 0 – ra2 → rd or ALUFB

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra or ra2	0	0	0	0	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	1	t	pa	pb	s	1	1	a	m	1	1

Description This chained-mode instruction places the product of values of ra and 1 in either rd or the multiplier feedback, and concurrently subtracts the value of ra2 from 0 and places the result into either rd or the ALU feedback.

Sources for ra
RA9-RA0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra none

Sources for ra2
RA9-RA0
C or CT register
MULFB (Multiplier feedback)
LAD (immediate data from LAD bus)
ONE (the value one)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT

Modifiers for rd
a (negate ALU result, valid only for chained mode noninteger types)
m (negate multiplier result, valid only for chained mode noninteger types)
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Output_sources
ALU
MULT

Restrictions If ra2 is specified then at least one feedback source must be used (either ra or ra2).

Example

```
pass.neg CT,d, LAD2, RB1,a, MULT
```

Syntax **pass.pass** *ra.type, ra2. rd[.modifier], output_source*

Execution ra × 1 → rd or MULFB; ra2 + 0 → rd or ALUFB

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra or ra2	0	0	0	0	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	1	t	pa	pb	s	1	1	a	m	0	0

Description This chained-mode instruction places the product of a value of ra and 1 in rd or the multiplier feedback, and concurrently places the sum of the value of ra2 and 0 into either rd of the ALU feedback.

Sources for ra
RA9-RA0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra none

Sources for ra2
RA9-RA0
C or CT register
MULFB (Multiplier feedback)
LAD (immediate data from LAD bus)
ONE (the value one)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT

Modifiers for rd
a (negate ALU result, valid only for chained mode noninteger types)
m (negate multiplier result, valid only for chained mode noninteger types)
e (send output to LAD bus, $\overline{WE}$ strobe)
h (send output to LAD bus, $\overline{ALTCH}$ strobe)

Output_sources
ALU
MULT

Restrictions If rb2 is specified then at least one feedback source must be used (either rb or rb2).

Example

```
pass.pass RA7.f, C2, RB0, ALU
```

Syntax **pass.sub** *ra.type, ra2. rb.type, rd[.modifier], output_source*

Execution ra × 1 → rd or MULFB; ra2 – rb → rd or ALUFB

Instruction Words

31	30	29	28	27 24	23 20	19 15
0	0	0	0	ra or ra2	rb	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	1	t	pa	pb	s	0	1	a	m	0	1

Description This chained-mode instruction places the product of the values of ra and 1 in either rd or MULFB, and concurrently places the difference of the values from ra2 and rb into rd or ALUFB.

Sources for ra
RA9-RA0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Types for ra and rb
none

Sources for ra2
RA9-RA0
C or CT register
MULFB (Multiplier feedback)
LAD (immediate data from LAD bus)
ONE (the value one)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT

Modifiers for rd
a (negate ALU result, valid only for chained mode noninteger types)
m (negate multiplier result, valid only for chained mode noninteger types)
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Output_sources ALU
MULT

Restrictions If ra2 is specified then at least one feedback source must be used (either ra or ra2).

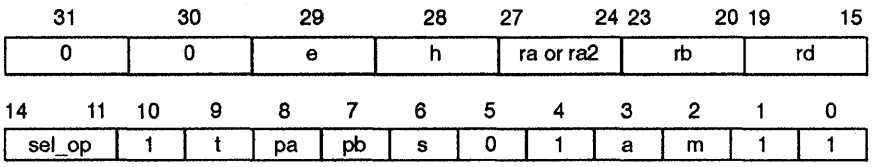
Example

```
pass.sub RA1.i, LAD2, RB7.u, CT, ALU
```

Syntax **pass.subrl** *ra2. rb.type, rd[.modifier], output_source*

Execution ra × 1 → rd or MULFB; rb – ra2 → rd or ALUFB

Instruction Words



Description This instruction places the product of a value in ra and 1 in either rd or multiplier feedback and concurrently the value of ra2 and rb and places the result in either rd or the ALU feedback.

Sources for ra
RA9-RA0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra and rb none

Sources for ra2
RA9-RA0
C or CT register
MULFB (Multiplier feedback)
LAD (immediate data from LAD bus)
ONE (the value one)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT

Modifiers for rd
a (negate ALU result, valid only for chained mode noninteger types)
m (negate multiplier result, valid only for chained mode noninteger types)
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Output_sources
ALU
MULT

Restrictions

If ra2 is specified then at least one feedback source must be used (either ra or ra2).

If rb2 is specified then at least one feedback source must be used (either rb or rb2).

Example

```
pass.subrl C.d, MULFB2, RB9.d, RA0.m, ALU
```

Syntax **pass.2suba** *ra.type, ra2, rd[.modifier], output_source*

Execution ra × 1 → rd or MULFB; 2 – ra2 → rd or ALUFB

Instruction Words

31	30	29	28	27 – 24	23 – 20	19 – 15
0	0	e	h	ra or ra2	rb	rd

14 – 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	1	t	pa	pb	s	0	1	a	m	1	0

Description This chained-mode instruction places the product of ra and 1 in rd or in MUL feedback, and concurrently subtracts the value of ra2 from 2 and places the result in rd or in the ALU feedback.

Sources for ra
RA9-RA0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra none

Sources for ra2
RA9-RA0
C or CT register
MULFB (Multiplier feedback)
LAD (immediate data from LAD bus)
ONE (the value one)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT

Modifiers for rd
a (negate ALU result, valid only for chained mode noninteger types)
m (negate multiplier result, valid only for chained mode noninteger types)
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Output_sources
ALU
MULT

Restrictions If ra2 is specified then at least one feedback source must be used (either ra or ra2).

Example

```
pass.2suba RA2.f, ONE2, RB9.f, CTa, MULT
```

Syntax **rti**

Execution IRAREG → Program Counter

Instruction Words

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Description This instruction causes a jump to the address stored in the interrupt return register (IRAREG). It does not affect the INTG signal, which remains active until interrupts are re-enabled.

Alternate Opcodes **reti** is an equivalent opcode for this instruction.

Syntax **rts**

Execution SUBADDR0 or SUBADDR1 → Program Counter

Instruction Words

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

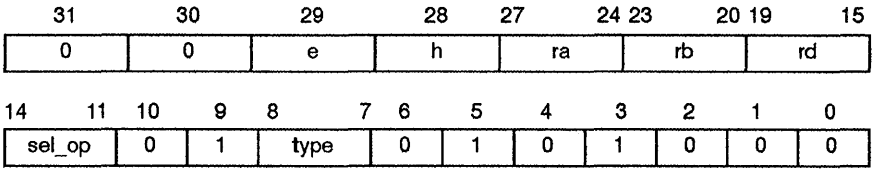
Description This instruction causes a jump to the address stored in the top of the stack. Note: ret is also valid

Alternate Opcodes **ret** is an equivalent opcode for this instruction.

Syntax

sll *ra.type, rb.type, rd[.modifier]*

Instruction Words



Description

This instruction shifts the value in ra to the left by the number of bit positions indicated in rb. Zeros are shifted into the least significant bit location.

Sources for ra

RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
ONE (the value one)

Sources for rb (see restrictions)

RB9-RB0

Types for ra and rb

i (signed integer)
u (unsigned integer)

Modifiers for ra and rb

none

Destinations for rd

RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd

e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Restrictions

The shift count is input as a five-bit positive number right-aligned in the exponent field of a single-precision floating point number. All other bits in the 32-bit word should be set to zero. For example 0x00 8000 00 = shift count of 1.

Example

This example shows how to shift using a variable shift value stored in RA0 and the operand to be shifted in RA1.

```
ld   RB0.u, shift_const, 1  ; load RB0 with shift count
                            ; of 23
sll RA0.u, RB0.u, RB1       ; prepare run-time shift
                            ; value (in RA0).
sll RA1.u, RB1.u, C         ; actual shift of RA1 with
                            ; the shift value in RA0
shift_const: data 0x0b 8000 00
                            ; equivalent shift count of 23
```

Syntax **sqrt** *ra.[modifier]type, rd[.modifier]*

Execution $\sqrt{ra} \rightarrow rd$

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra	0	0	0	0	rd

14 11	10	9 7	6	5	4	3	2	1	0
sel_op	0	type	1	1	va	1	ny	wa	wb

Description This instruction takes the square root of the value in ra and places it in rd.

Sources for ra
RA9-RA0
C or CT Register
ALUFB (ALU feedback)
ONE (the value one)

Types for ra
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra
v (absolute value, not valid for integer types)
w (wrapped, not valid for integer types)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
n (negated, not valid for integer types)
e (send output to LAD bus, $\overline{WE}$ strobe)
h (send output to LAD bus, $\overline{ALTCH}$ strobe)

Restrictions Absolute value modifiers, negated result and wrapped numbers are only permitted with floating-point operations.

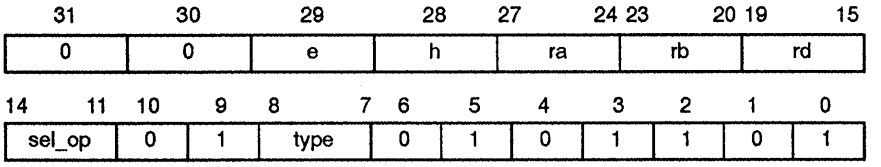
Example

```
sqrt RA7.u, C.n
```

Syntax

sra *ra.type, rb.type, rd[.modifier]*

Instruction Words



Description

This instruction shifts the value in ra to the right by the number of bit positions indicated in rb. The sign bit is not affected.

Sources for ra

RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
ONE (the value one)

Sources for rb (see restrictions)

RB9-RB0

Types for ra and rb

i (signed integer)
u (unsigned integer)

Modifiers for ra and rb

none

Destinations for rd

RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd

e (send output to LAD bus, $\overline{WE}$ strobe)
h (send output to LAD bus, $\overline{ALTCH}$ strobe)

Restrictions

The shift count is input as a five-bit positive number right-aligned in the exponent field of a single-precision floating point number. All other bits in the 32-bit word should be set to zero.

The types for ra and rb must be the same.

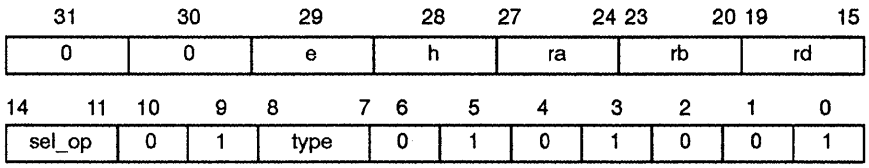
Example

```
sra MULFB.i, LAD.i, C.e
```

Syntax

srl *ra.type, rb,type, rd[.modifier]*

Instruction Words



Description

This instruction shifts the value in ra to the right by the number of bit positions indicated in rb. Zeros are shifted into the most significant bit location.

Sources for ra

RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
ONE (the value one)

Sources for rb (see restrictions)

RB9-RB0

Types for ra and rb

i (signed integer)
u (unsigned integer)

Modifiers for ra and rb

none

Destinations for rd

RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd

e (send output to LAD bus, $\overline{WE}$ strobe)
h (send output to LAD bus, $\overline{ALTCH}$ strobe)

Restrictions

The shift count is input as a five-bit positive number right-aligned in the exponent field of a single-precision floating point number. All other bits in the 32-bit word should be set to zero.

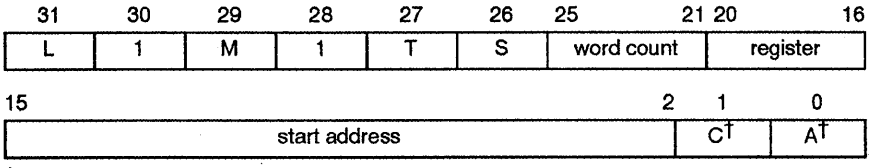
The types for ra and rb must be the same.

Example

```
srl CT.i, LAD.i, RA3
```

Syntax **st** *reg.type, address, count*

Instruction Word



† For LAD moves only.

Description

During a move instruction, no FPU operation is performed. Register control logic for move instructions counts sequentially from the beginning register address, with the exception that the C and CT registers are omitted from the count. The entire register file acts like a ring buffer during the move instruction. The C and CT registers are not accessible to moves. It is illegal to use the C or CT register address as the starting address for a move instruction.

T (Type) and S (Size) give the format of the numbers
T 0 = integer 1 = floating point
S 0 = 32 bits 1 = 64 bits
Note: Setting TS = 01 is reserved

Word count is the number of operands to be moved (n). A count of 0 will move 256 items 1 or 2 32-bit words long. The beginning register address is stored in the register field, and the beginning memory address is the start address field (bits 15-0).

An indirect move is designated by selecting MCADDR as the address. The M bit will be set low, and the 16 low-order bits are then disregarded. The starting address in memory comes from the MCADDR register.

To move data to the LAD bus, 'LAD' is selected as the address. The L bit will be set high, and the low-order 16 bits are set to 0. An address of LAD_A will write data to the LAD bus with an $\overline{\text{ALTCH}}$ strobe (instead of the normal $\overline{\text{WE}}$). The A bit will be set high in the instruction word.

An address of 'COINT' will write data to the LAD bus and set $\overline{\text{COINT}}$ low for the cycles the store is executing. (C will be set high in the instruction word.) An address of 'COINT_A' will store to the LAD bus with $\overline{\text{COINT}}$ enabled and an $\overline{\text{ALTCH}}$ strobe (instead of the normal $\overline{\text{WE}}$). C and A will be set high in the instruction word. The COINT and COINT_A options are only valid for host-independent mode.

Sources for reg

RA9-RA0
RB9-RB0
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Types for reg f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Destination addresses 0x0-0xFFFF
MCADDR, LAD, COINT, COINT_A, LAD_A

Range for count 0-31

Example

```
st RA0.f, MCADDR, 3
st RB1.i, LAD, 5
```

Syntax **sub** *ra.[modifier]type, rb.[modifier]type, rd[.modifier]*

Execution ra – rb → rd

Instruction Words

31	30	29	28	27 24	23 20	19 15
0	0	e	h	ra	rb	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	t	pa	pb	0	0	va	vb	vy	0	1

Description This instruction places the difference in the values in ra and rb in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra and rb v (abslute value, not valid for integer types)

Destinations for rd
RA-9RA0
RB9-RB0
C or CT
STATUS, CONFIG,COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
v (absolute value, not valid for integer types)
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

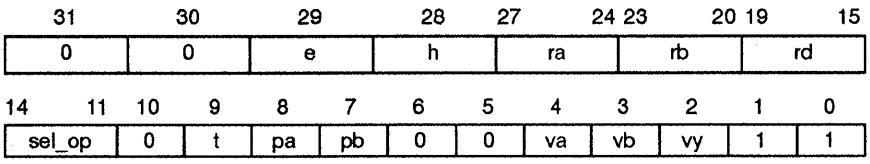
Example

```
sub LAD.vd, ONE.vf, RA0.h
```

Syntax **subrl** *ra.[modifier]type, rb.[modifier]type, rd[.modifier]*

Execution rb – ra → rd

Instruction Words



Description This instruction takes the difference in the value in rb from the value in ra and places it in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Sources for rb
RB9-RB0
C or CT Register
ALUFB (ALU feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
f (single-precision floating-point)
d (double-precision floating-point)
i (signed integer)
u (unsigned integer)

Modifiers for ra and rb v (absolute value, not valid for integer types)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Example

```
subrl RA.f, RB2.vf, RA0
```

Syntax **utod** *ra, rd[.modifier]*

Execution ra (unsigned integer) → rd (double-precision)

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra	0	0	0	0	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	0	1	1	0	1	0	1	0	1	0

Description This instruction converts an unsigned integer value in ra to a double-precision floating-point format and places the result in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
ONE (the value one)

Types for ra type is implicit in the opcode

Modifiers for ra none

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Example

```
utod MULFB, CT.e
```

Syntax **utof** *ra, rd[.modifier]*

Execution ra (unsigned integer) → rd (single-precision)

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra	0	0	0	0	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	0	0	0	0	1	0	1	0	1	0

Description This instruction converts an unsigned integer in ra to single-precision floating-point format and places it in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra type is implicit in the opcode

Modifiers for ra none

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Example

```
utof LAD, RA9.e
```

Syntax **uwrapi** *ra.[modifier]type, rd[.modifier]*

Execution wrapped in ra → denormal in rd

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra	0	0	0	0	rd

14 11	10	9	8 7	6	5	4	3	2	1	0
sel_op	0	0	type	0	1	va	1	1	0	1

Description This instruction unwraps the inexact operand in ra and places it in rd as a denormalized number.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra
f (single-precision floating-point)
d (double-precision floating-point)

Modifiers for ra
v (absolute value)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Example

```
unwrapi RA9.vf, C.h
```

Syntax **uwrapr** *ra.[modifier]type, rd[.modifier]*

Execution wrapped in ra → denormal in rd

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra	0	0	0	0	rd

14 11	10	9	8 7	6	5	4	3	2	1	0
sel_op	0	0	type	0	1	va	1	1	1	0

Description This instruction converts a wrapped rounded number in ra to a denormalized number in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra
f (single-precision floating-point)
d (double-precision floating-point)

Modifiers for ra
v (absolute value)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, CONTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG,LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{WE}$ strobe)
h (send output to LAD bus, $\overline{ALTCH}$ strobe)

Example

```
uwrapr RA3.d, CT.h
```

Syntax **uwrapx** *ra.[modifier]type, rd[.modifier]*

Execution wrapped in ra → denormal in rd

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra	0	0	0	0	rd

14 11	10	9	8 7	6	5	4	3	2	1	0
sel_op	0	0	type	0	1	va	1	1	0	0

Description This instruction takes the exact, wrapped operand in ra and converts it to a denormalized number in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra
f (single-precision floating-point)
d (double-precision floating-point)

Modifiers for ra
v (absolute value)

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Example

```
uwrapx C.vf, RA8.e
```

Syntax

wrap *ra.[modifier]type, rd[.modifier]*

Execution

denormal in ra → wrapped in rd

Instruction Words

31	30	29	28	27 24	23	22	21	20	19 15
0	0	e	h	ra	0	0	0	0	rd

14 11	10	9	8 7	6	5	4	3	2	1	0
sel_op	0	0	type	0	1	va	1	0	0	0

Description

This instruction takes a denormalized number in ra and converts it to a wrapped number in rd.

Sources for ra

RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra

f (single-precision floating-point)
d (double-precision floating-point)

Modifiers for ra

v (absolute value)

Destinations for rd

RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd

e (send output to LAD bus, $\overline{\text{WE}}$ strobe)
h (send output to LAD bus, $\overline{\text{ALTCH}}$ strobe)

Example

```
wrap RA0.d, RB1.h
```

Syntax **xor** *ra.type, rb.tpye, rd[.modifier]*

Execution ra XOR rb → rd

Instruction Words

31	30	29	28	27 24	23 20	19 15
0	0	e	h	ra	rb	rd

14 11	10	9	8	7	6	5	4	3	2	1	0
sel_op	0	1	0	t	0	0	0	1	1	0	1

Description This instruction takes the logical exclusive OR of ra with rb and places the result in rd.

Sources for ra
RA9-RA0
C or CT Register
MULFB (Multiplier feedback)
LAD (Immediate data from LAD bus)
ONE (the value one)

Types for ra and rb
i (signed integer)
u (unsigned integer)

Modifiers for ra and rb none

Destinations for rd
RA9-RA0
RB9-RB0
C or CT
STATUS, CONFIG, COUNTX, COUNTY
VECTOR, MCADDR, SUBADD0, SUBADD1, IRAREG, LOOPCT

Modifiers for rd
e (send output to LAD bus, $\overline{WE}$ strobe)
h (send output to LAD bus, $\overline{ALTCH}$ strobe)

Restrictions The types for ra and rb must be the same.

Example

```
xor RA7.u, RB2.u, CT.h
```

Appendix A

System Design Considerations

Using high-performance CMOS logic devices, such as the TMS34082, requires careful attention to high-speed logic design and PWB design practices. A few simple design techniques can reduce check-out time during the development phase and, more importantly, improve system reliability as your product enters production. The following sections are general recommendations to reduce your chances of intermittent problems.

A.1 Logic Design

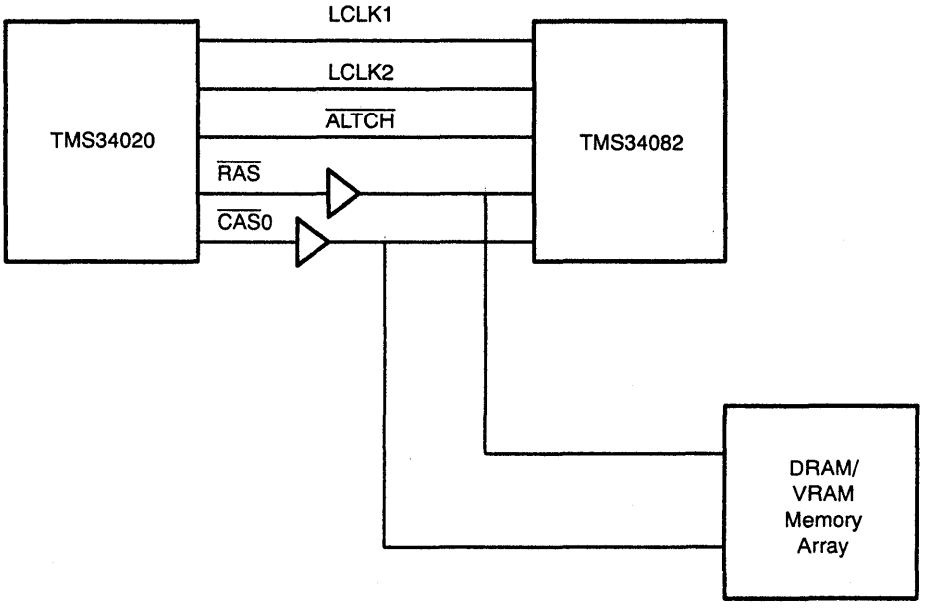
Check to make sure that the drive capability of each TMS34082 output driver is not exceeded, particularly with the clock drivers. This can affect the output signal quality as well as driver supply demands.

When operating in coprocessor mode, do not use buffers on the following signals between the TMS34020 and TMS34082 (unless a critical path timing analysis between the two devices has been completed):

- LCLK1 and LCLK2 (local clocks)
- $\overline{\text{ALTCH}}$ (address latch)
- $\overline{\text{CAS}}$ (column address strobe)
- SF (special function)

Figure A-1 shows how $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ buffers can be added for DRAM/VRAM memory. These buffers effectively isolate the DRAM/VRAM devices from the TMS34020.

Figure A–1. Example of Using $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ Buffers in Coprocessor Mode

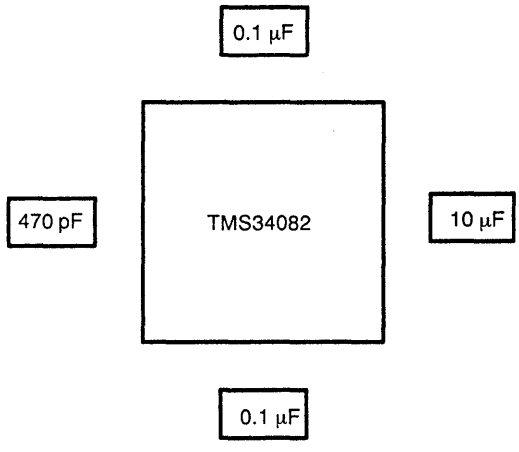


A.2 Bypass Capacitors

The TMS34082 is a high-speed CMOS device containing two 32-bit data buses and one 16-bit address bus. As a result, a constant voltage source must be maintained for the device during signal transitions. The TMS34082 contains 10 V_{CC} pins and 14 GND pins for internal power requirements.

External bypass capacitors must also be used for decoupling the switching transitions. Use two or more 0.1-μF low-leakage high-quality capacitors around the perimeter of the TMS34082 package or under the device. Place the capacitors as close to the TMS34082 as possible. These are used to filter out unwanted switching noise caused by the CMOS output drivers, one of the major sources of noise. Also, use one 470-pF low-leakage high-quality capacitor to reduce the very high frequency noise (such as clock frequencies) and at least one 10-μF solid tantalum filter capacitor to take care of low frequency noise (such as power supply surges). The 10-μF filter capacitor smooths out voltage spikes during switching transitions. The capacitance values are approximate and should have a working voltage of at least 10 V. By using three capacitor sizes, three different frequency bands of noise are filtered as opposed to just one narrow band for one bypass capacitor size.

Figure A–2. Recommended Bypass Capacitor Placement



A.3 PWB Design

The TMS34082 should be designed into a PC board environment with an embedded V_{CC} or GND plane. For any production high-speed logic board, power planes are an absolute necessity. Each V_{CC} and GND pin on the TMS34082 must be connected to the appropriate supply pin. Use the shortest amount of PWB etch possible. This effectively forms a common reference point throughout the PC board as well as the device substrate.

As with most complex CMOS devices, extra care must be used when distributing CMOS logic over more than one GND plane. An example of this is when a TMS34020 is on one board and multiple TMS34082s (running in coprocessor mode) are located on a daughtercard. The common ground connection between the two power planes behaves like an inductor according to transmission line theory. The greater the current, the greater the inductance. Here, the solution is to use many GND connections and to make them as short as possible. In addition, even more bypass capacitors should be used.

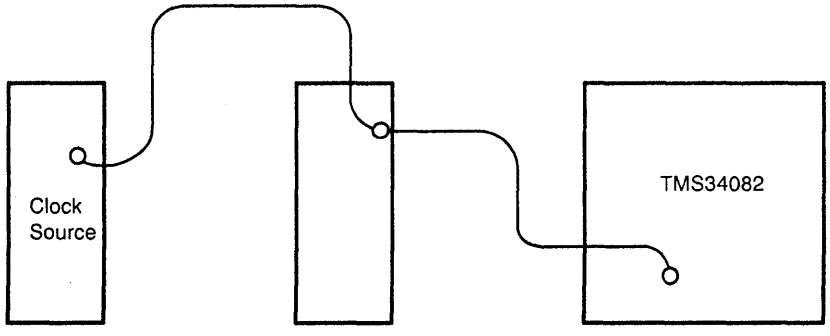
When using a PGA socket, use gold-plated contacts where the TMS34082 pins mate into the socket to lower the inductance and resistance. A gold plating thickness of 10 microinches is sufficient.

A.4 Clock Routing

Clocks are the heart of a high-performance system, so a little extra care will pay off many times over. Many of these ideas not only apply to the TMS34082, but to most high-speed CMOS logic devices.

PC board layout must take into account transmission-line theory. It is generally accepted that any clock line over 7 inches long should be considered as a transmission line. Use a daisy-chained clock distribution system and avoid using a 'T' (where three lines of etch come into a common vertex) or stubs. Avoid the use of 90° angles within the clock trace; use arcs or smooth lines instead, as shown in Figure A–3. This reduces the number of signal reflections within the clock trace.

Figure A–3. Recommended Clock Routing Techniques



When routing your PC board, route the clock signals first (they may even be hand routed). To help reduce cross talk and radiated RF interference, keep the length of clock interconnections as short as possible and place the majority of clock routing next to one of the V_{CC} or GND power planes. Cross talk is where one signal gets coupled onto another signal; one trace behaves like a transmitter antenna and the other trace acts as a receiver.To further reduce cross talk, make certain that the clock trace does not run parallel to data or control lines for more than three inches if they are spaced within 100 mils of each other. Traces adjacent to the clock lines that are connected to GND also may be used.

Since many clock interconnections behave like transmission lines, impedance mismatches can generate reflections. From a time-domain point of view, these can result in ringing, undershoot, and overshoot. If the clock drivers generate excessive amounts of ringing and undershoot at their destinations, it will be necessary to put either an impedance matching termination network at the farthest signal point from the driver or a series resistor (22 Ω to 39 Ω) between the clock driver output and the receiving input. Using a series resistor also slows down the signal response times slightly. The amount of undershoot or ringing may be difficult to predict before hand, but there are many good articles on transmission line theory for PC board design.

A.5 Thermal Considerations

Because the TMS34082 is implemented in CMOS, its power consumption requirements are low and generate little heat. You must make certain that the operating temperature of the surrounding environments is within TMS34082 operating specifications.

Appendix B

TMS34082A Data Sheet

The pinout, electrical specifications, timing diagrams, and mechanical specifications are contained within the TMS34082 Data Sheet and appear in this appendix.

SCGS001 – D3150, SEPTEMBER 1988 – REVISED MAY 1991

- **High-Performance Floating-Point RISC Processor Optimized for Graphics**
- **Two Operating Modes**
  - **Floating-Point Coprocessor for TMS34020 Graphics System Processor**
  - **Independent Floating-Point Processor**
- **Direct Connection to TMS34020 Coprocessor Interface**
  - **Direct Extension to the TMS34020 Instruction Set**
  - **Multiple TMS34082A Capability**
- **Fast Pipeline Instruction Cycle Time**
  - **TMS34082A-40 . . . 50-ns Coprocessor Mode . . . 50-ns Host-Independent Mode**
  - **TMS34082A-32 . . . 62.5-ns Coprocessor Mode . . . 60-ns Host-Independent Mode**
- **Sustained Data Transfer Rates of 160 MBytes/s (TMS34082-40)**
- **Sequencer Executes Internal or User-Programmed Instructions**
- **22 64-Bit Data Registers**
- **Comprehensive Floating-Point and Integer Instruction Set**
- **Internal Programs for Vector, Matrix, and 3-D Graphics Operations**
- **Full IEEE Standard 754-1985 Compatibility**
  - **Addition, Subtraction, Multiplication, and Comparison**
  - **Division and Square Root**
- **Selectable Data Formats**
  - **32-Bit Integer**
  - **32-Bit Single-Precision Floating-Point**
  - **64-Bit Double-Precision Floating-Point**
- **External Memory Addressing Capability**
  - **Program Storage (up to 64K Words)**
  - **Data Storage (up to 64K Words)**
- **0.8-μm EPIC™ CMOS Technology**
  - **High-Performance**
  - **Low Power (< 1.5 W)**

description

The TMS34082A is a high-speed graphics floating-point processor implemented in Texas Instruments advanced 0.8-μm CMOS technology. The TMS34082A combines a 16-bit sequencer and a 3-operand (source A, source B, and destination) 64-bit Floating-Point Unit (FPU) with 22 64-bit data registers on a single chip. The data registers are organized into two files of ten registers each, with two registers for internal feedback. In addition, it provides an instruction register to control FPU execution, a status register to retain the most recent FPU status outputs, eight control registers, and a two-deep stack (see functional block diagram).

The TMS34082A is fully compatible with IEEE Standard 754-1985 for binary floating-point addition, subtraction, multiplication, division, square root, and comparison. Floating-point operands can be either in single- or double-precision IEEE format.

In addition to floating-point operations, the TMS34082A performs 32-bit integer arithmetic, logical comparisons, and shifts. Integer operations may be performed on 32-bit 2s complement or unsigned operands. Integer results are 32-bits long (even for 32 x 32 integer multiplication). Absolute value conversions, floating-point to integer conversions, and integer to floating-point conversions are available.

The ALU and the multiplier are closely coupled and can be operated in parallel to perform sums of products or products of sums. During multiply/accumulate operations, both the ALU and the multiplier are active and the registers in the FPU core can be used to feedback products and accumulate sums without tying up locations in register files A and B.

When used with the TMS34020, the TMS34082A operates in the coprocessor mode. The TMS34020 can control multiple TMS34082A coprocessors. When used as a stand-alone or with processors other than the TMS34020, the TMS34082A operates in the host-independent mode. The TMS34082A is fully programmable by the user and can interface to other processors or floating-point subsystems through its two 32-bit bidirectional buses. In

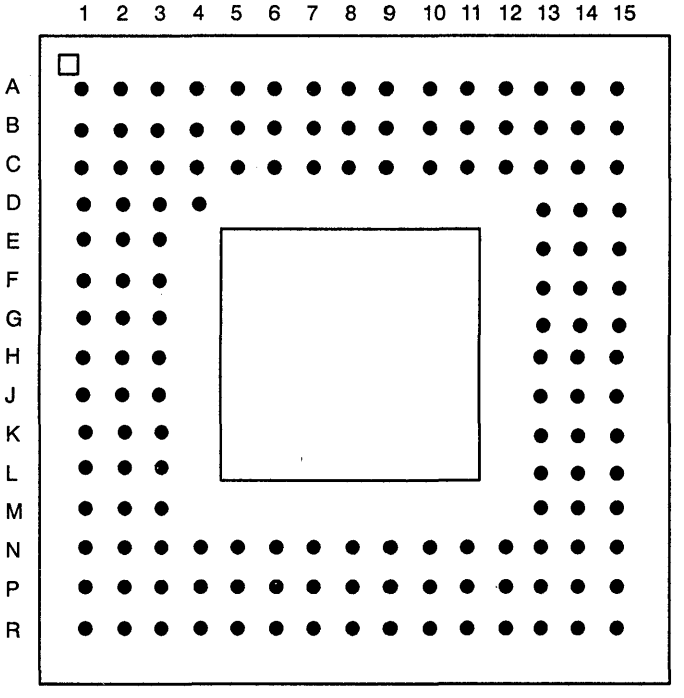
EPIC is a trademark of Texas Instruments Incorporated.

the coprocessor mode, the TMS340 family tools may be used to develop code for the TMS34082A. The TMS34082A software tool kit is used to develop code for host-independent mode applications or for external routines in the coprocessor mode.

pin descriptions

Pin descriptions and grid assignments for the TMS34082A are given on the following pages. The pin at location D4 has been added for indexing purposes.

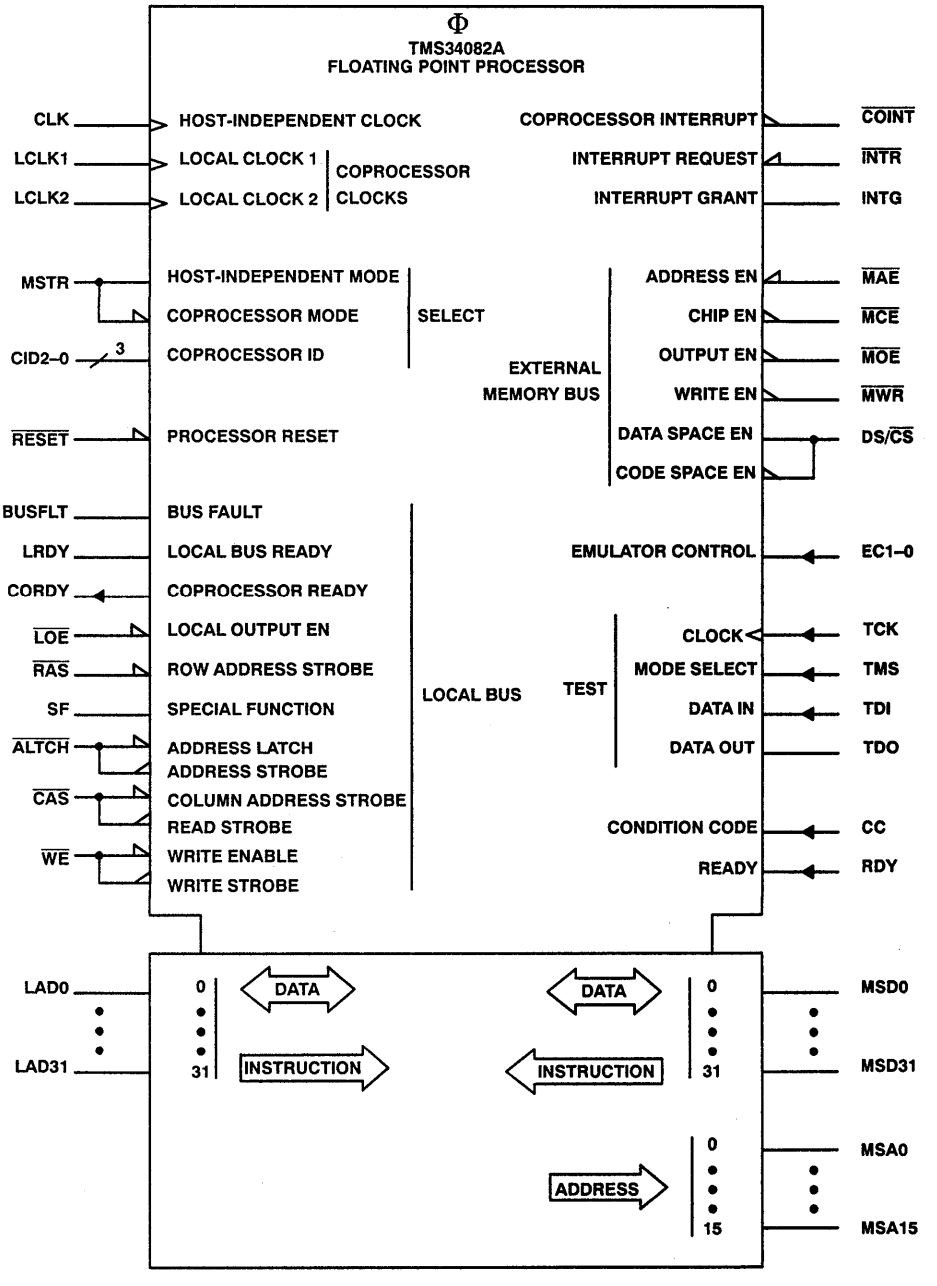
145-PIN GC PACKAGE
(TOP VIEW)



PIN GRID ASSIGNMENTS

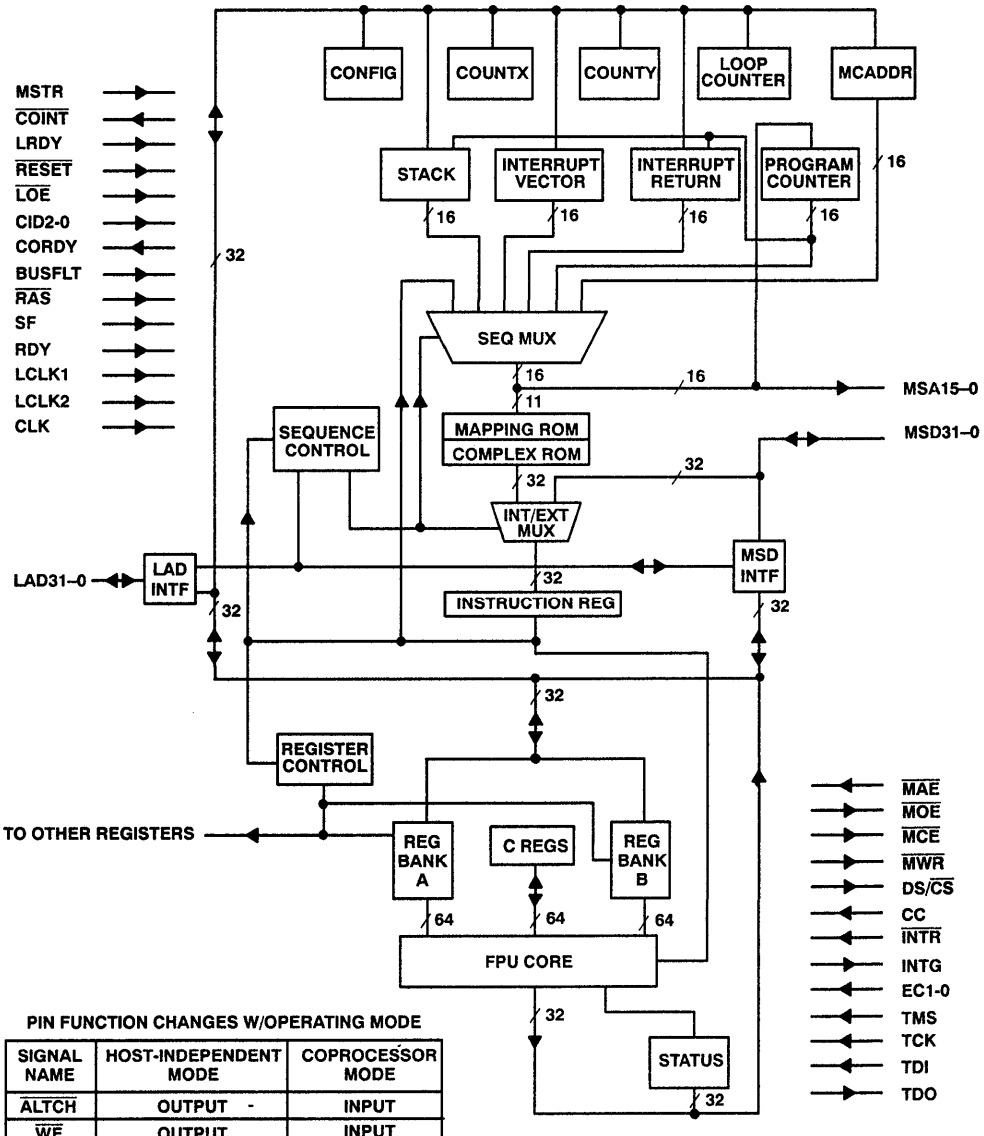
PIN NO.	PIN NAME	PIN NO.	PIN NAME	PIN NO.	PIN NAME	PIN NO.	PIN NAME	PIN NO.	PIN NAME
A1	NC	B15	LAD27	F1	MSD10	K15	RDY	P2	NC
A2	LAD1	C1	MSD4	F2	MSD9	L1	MSD18	P3	MSD29
A3	LAD3	C2	MSD3	F3	V_{CC}	L2	MSD21	P4	MSD31
A4	LAD5	C3	MSD0	F13	CORDY	L3	MSD23	P5	MSA1
A5	LAD8	C4	V_{SS}	F14	$\overline{ALTCH}$	L13	V_{SS}	P6	MSA3
A6	LAD9	C5	V_{CC}	F15	$\overline{CAS}$	L14	CID0	P7	MSA6
A7	LAD11	C6	LAD6	G1	MSD13	L15	CID2	P8	MSA8
A8	LAD12	C7	V_{SS}	G2	MSD12	M1	MSD20	P9	MSA10
A9	LAD13	C8	V_{CC}	G3	MSD11	M2	MSD24	P10	MSA13
A10	LAD15	C9	V_{SS}	G13	$\overline{WE}$	M3	V_{SS}	P11	$\overline{MWR}$
A11	LAD17	C10	V_{CC}	G14	EC1	M13	V_{CC}	P12	$\overline{MOE}$
A12	LAD19	C11	LAD21	G15	EC0	M14	LCLK1	P13	INTG
A13	LAD22	C12	V_{SS}	H1	MSD14	M15	LCLK2	P14	BUSFLT
A14	LAD24	C13	LAD25	H2	TDO	N1	MSD22	P15	$\overline{RAS}$
A15	NC	C14	LAD26	H3	V_{SS}	N2	MSD26	R1	NC
B1	MSD1	C15	LAD29	H13	V_{SS}	N3	V_{CC}	R2	MSD27
B2	NC	D1	MSD6	H14	$\overline{LOE}$	N4	MSD28	R3	MSD30
B3	LAD0	D2	MSD5	H15	TDI	N5	V_{SS}	R4	MSA0
B4	LAD2	D3	MSD2	J1	MSD15	N6	V_{CC}	R5	MSA2
B5	LAD4	D4	NC	J2	MSD16	N7	MSA5	R6	MSA4
B6	LAD7	D13	V_{CC}	J3	V_{CC}	N8	V_{SS}	R7	MSA7
B7	LAD10	D14	LAD28	J13	CC	N9	V_{CC}	R8	TCK
B8	TMS	D15	LAD31	J14	MSTR	N10	MSA14	R9	MSA9
B9	LAD14	E1	MSD8	J15	CLK	N11	V_{SS}	R10	MSA11
B10	LAD16	E2	MSD7	K1	MSD17	N12	$\overline{MAE}$	R11	MSA12
B11	LAD18	E3	V_{SS}	K2	MSD19	N13	LRDY	R12	MSA15
B12	LAD20	E13	V_{SS}	K3	V_{SS}	N14	SF	R13	DS/$\overline{CS}$
B13	LAD23	E14	LAD30	K13	CID1	N15	$\overline{RESET}$	R14	$\overline{MCE}$
B14	NC	E15	$\overline{COINT}$	K14	$\overline{INTR}$	P1	MSD25	R15	NC

logic symbol†



† This symbol is in accordance with ANSI/IEEE Std 91-1984.

functional block diagram



PIN FUNCTION CHANGES W/OPERATING MODE

SIGNAL NAME	HOST-INDEPENDENT MODE	COPROCESSOR MODE
ALTCH	OUTPUT	INPUT
WE	OUTPUT	INPUT
CAS	OUTPUT	INPUT

TERMINAL FUNCTIONS

PIN NAME	PIN NO.	I/O†	DESCRIPTION
$\overline{\text{ALTCH}}$	F14	I [O]	Address Latch, active low. In the coprocessor mode, falling edge of $\overline{\text{ALTCH}}$ latches instruction and status present on the LAD bidirectional bus (LAD31-0). In the host-independent mode, $\overline{\text{ALTCH}}$ is address output strobe for memory accesses on LAD31-0.
BUSFLT	P14	I	Bus Fault. In the coprocessor mode, BUSFLT high indicates a data fault on the LAD bus (LAD31-0) during current bus cycle, which in turn causes TMS34082A not to capture current data on LAD bus. Tied low if not used or in the host-independent mode.
$\overline{\text{CAS}}$	F15	I [O]	Column Address Strobe, active low. In the coprocessor mode, causes TMS34082A to latch LAD bus data when $\overline{\text{CAS}}$ has a low-to-high transition if LRDY was high and BUSFLT was low at the previous LCLK2 rising edge. In the host-independent mode, this signal is the read strobe output.
CC	J13	I	Condition Code Input. In both modes, may be used as an external conditional input for branch conditions.
CID0 CID1 CID2	L14 K13 L15	I	Coprocessor ID. In the coprocessor mode, used to set a coprocessor ID so that a TMS34020 Graphics System Processor controlling multiple TMS34082A coprocessors can designate which coprocessor is being selected by the current instruction. Tied low in the host-independent mode.
CLK	J15	I	System Clock. In the coprocessor mode, tied low. In the host-independent mode, input is the system clock.
$\overline{\text{COINT}}$	E15	O	Coprocessor Interrupt Request, active low. In the coprocessor mode, signals an exception not masked out in the configuration register. Remains low until the status register is read. In the host-independent mode, user programmable I/O when LADCFG is low. When LADCFG is high, designates bus cycle boundaries on LAD31-0.
CORDY	F13	O	Coprocessor Ready. In the coprocessor mode, if the TMS34020 sends an instruction before the TMS34082A has completed a previous instruction, this signal goes low to indicate that the TMS34020 should wait. In the host-independent mode, user programmable.
DS/$\overline{\text{CS}}$	R13	O	Data Space/Code Space. In both modes, when MEMCFG is low and DS/$\overline{\text{CS}}$ is low, selects program memory on MSD port. When MEMCFG is low and DS/$\overline{\text{CS}}$ is high, selects data memory on MSD port. When MEMCFG is high, DS/$\overline{\text{CS}}$ is memory chip select, active low.
EC0 EC1	G15 G14	I	Emulator Mode Control and Test. In both modes, tied high for normal operation.
INTG	P13	O	Interrupt Grant Output. In the coprocessor mode, INTG is low. In the host-independent mode, this signal is set high to acknowledge an interrupt request input.
$\overline{\text{INTR}}$	K14	I	Interrupt Request Input, active low. In the coprocessor mode, $\overline{\text{INTR}}$ is tied high. In the host-independent mode, causes call to subroutine address in interrupt vector register.

† The [ ]'s denote the type of buffer utilized in the host-independent mode. If no [ ]'s appear, the buffer type is identical for both modes of operation.

TERMINAL FUNCTIONS (Continued)

PIN NAME	PIN NO.	I/O	DESCRIPTION
LAD0	B3	I/O	Local Address and Data Bus. In the coprocessor mode, used by TMS34020 to input instructions and data operands to TMS34082A, and used by TMS34082A to output results. In the host-independent mode, used by the TMS34082A for address output and data I/O.
LAD1	A2		
LAD2	B4		
LAD3	A3		
LAD4	B5		
LAD5	A4		
LAD6	C6		
LAD7	B6		
LAD8	A5		
LAD9	A6		
LAD10	B7		
LAD11	A7		
LAD12	A8		
LAD13	A9		
LAD14	B9		
LAD15	A10		
LAD16	B10		
LAD17	A11		
LAD18	B11		
LAD19	A12		
LAD20	B12		
LAD21	C11		
LAD22	A13		
LAD23	B13		
LAD24	A14		
LAD25	C13		
LAD26	C14		
LAD27	B15		
LAD28	D14		
LAD29	C15		
LAD30	E14		
LAD31	D15		
LCLK1	M14	I	Local Clocks 1 and 2. In the coprocessor mode, two local clocks generated by the TMS34020, 90 degrees out of phase, to provide timing inputs to TMS34082A. In the host-independent mode, tied low.
LCLK2	M15		
$\overline{\text{LOE}}$	H14	I	Local Bus Output Enable, active low. In both modes, enables the local bus (LAD31-0) to be driven at the proper times when low. In addition during the host-independent mode when LADCFG is low, does not affect $\overline{\text{ALTCH}}$, $\overline{\text{CAS}}$, $\overline{\text{WE}}$, CORDY, or $\overline{\text{COINT}}$. When LADCFG is high, $\overline{\text{ALTCH}}$, $\overline{\text{COINT}}$, and CORDY are not disabled by $\overline{\text{LOE}}$ high; $\overline{\text{CAS}}$ and $\overline{\text{WE}}$ are disabled.
LRDY	N13	I	Local Bus Data Ready. In the coprocessor mode, when LRDY is high, indicates that data is available on LAD bus. When LRDY is low, indicates that the TMS34082A should not load data from LAD31-0 and may also be used in conjunction with BUSFLT. In the host-independent mode, when LRDY is low, the device is stalled until LRDY is set high again and tied high if not used.
$\overline{\text{MAE}}$	N12	I	Memory Address and Data Output Enable, active low. In both modes, with $\overline{\text{MAE}}$ low, the TMS34082A can output an address on MSA15-0 and data on MSD31-0. $\overline{\text{MAE}}$ high does not disable $\overline{\text{DS}}$/$\overline{\text{CS}}$, $\overline{\text{MCE}}$, $\overline{\text{MWR}}$, or $\overline{\text{MOE}}$.
$\overline{\text{MCE}}$	R14	O	Memory Chip Enable. In both modes, when MEMCFG low, active (low) indicates access to external memory on MSD31-0. When MEMCFG is high, $\overline{\text{MCE}}$ low is external code memory chip select.
$\overline{\text{MOE}}$	P12	O	Memory Output Enable, active low. In both modes when low, enables output from external memory on to MSD port.

TERMINAL FUNCTIONS (Continued)

PIN NAME	PIN NO.	I/O	DESCRIPTION
MSA0	R4	O	Memory Address output. In both modes, addresses up to 64K words of external program memory and/or up to 64K words of data memory on the MSD port, depending on setting of DS/$\overline{CS}$ select.
MSA1	P5		
MSA2	R5		
MSA3	P6		
MSA4	R6		
MSA5	N7		
MSA6	P7		
MSA7	R7		
MSA8	P8		
MSA9	R9		
MSA10	P9		
MSA11	R10		
MSA12	R11		
MSA13	P10		
MSA14	N10		
MSA15	R12		
MSD0	C3	I/O	External Memory Data. In both modes, I/Os to external memory. Used to read from or write to external data or program memory on the MSD port.
MSD1	B1		
MSD2	D3		
MSD3	C2		
MSD4	C1		
MSD5	D2		
MSD6	D1		
MSD7	E2		
MSD8	E1		
MSD9	F2		
MSD10	F1		
MSD11	G3		
MSD12	G2		
MSD13	G1		
MSD14	H1		
MSD15	J1		
MSD16	J2		
MSD17	K1		
MSD18	L1		
MSD19	K2		
MSD20	M1		
MSD21	L2		
MSD22	N1		
MSD23	L3		
MSD24	M2		
MSD25	P1		
MSD26	N2		
MSD27	R2		
MSD28	N4		
MSD29	P3		
MSD30	R3		
MSD31	P4		
MSTR	J14	I	Host-Independent/Coprocessor Mode Select. In the coprocessor mode, MSTR must be tied low to operate properly. In the host-independent mode, MSTR must be tied high to operate properly.
$\overline{MWR}$	P11	O	Memory Write Enable. In both modes, when low, data on MSD31-0 can be written to external program or data memory.

TERMINAL FUNCTIONS (Continued)

PIN NAME	PIN NO.	I/O†	DESCRIPTION
NC	A1 A15 B2 B14 D4 P2 R1 R15		No Internal Connection. These pins should be left floating.
$\overline{RAS}$	P15	I	Row Address Strobe, active low. In the coprocessor mode, $\overline{RAS}$ is high during all of coprocessor instruction cycle. In the host-independent mode, it is not used.
RDY	K15	I	Ready. In both modes, when RDY is low, it causes a nondestructive stall of sequencer and floating-point operations. All internal registers and status in the FPU core are preserved. Also, no output lines will change state.
$\overline{RESET}$	N15	I	Reset, active low. In both modes, resets sequencer output and clears pipeline registers, internal states, status, and exception disable registers in FPU core. Other registers are unaffected.
SF	N14	I	Special Function Input. In the coprocessor mode when SF is high, indicates the LAD bus input is an instruction or data from TMS34020 registers. When SF is low, indicates the LAD input is a data operand from memory. In the host-independent mode, not used.
TCK	R8	I	Test Clock for JTAG four-wire boundary scan. In both modes, TCK is low for normal operation.
TDI	H15	I	Test Data Input for JTAG four-wire boundary scan. In both modes, TDI may be left floating.
TDO	H2	O	Test Data Output for JTAG four-wire boundary scan
TMS	B8	I	Test Mode Select for JTAG four-wire boundary scan. In both modes, TMS may be left floating.
V_{CC}	C5 C8 C10 D13 F3 J3 M13 N3 N6 N9	I	5-V Power Supply. All pins must be connected and used.
V_{SS}	C4 C7 C9 C12 E3 E13 H3 H13 K3 L13 M3 N5 N8 N11	I	Ground Pins. All pins must be connected and used.
WE	G13	I [O]	Write Enable, active low. In the coprocessor mode, the write strobe from the TMS34020 to enable a write to or from the TMS34082A LAD bus. In the host-independent mode, the TMS34082A write strobe output.

† The [ ]'s denote the type of buffer utilized in the host-independent mode. If no [ ]'s appear, the buffer type is identical for both modes of operation.

data flow

The TMS34082A has two bidirectional 32-bit buses, LAD31-0 and MSD31-0. Each bus can be used to pass instructions and data operands to the FPU core and to output results. A separate 16-bit bus, MSA15-0, provides memory addressing capability on the MSD bus.

When the TMS34082A is used as a coprocessor for the TMS34020 Graphics System Processor (GSP), data for the TMS34082A can be transferred through the 32-bit bidirectional data bus (LAD31-0) and may be passed to any internal registers or to external memory on the memory expansion interface (MSD31-0). When the TMS34082A is used as a standalone FPU, it can use both the LAD bus (LAD31-0) and the MSD bus (MSD31-0) to interface with external data memory or system buses.

In the host-independent mode, the TMS34082A can be operated with the LAD bus as its single data bus and the MSD bus as the instruction source, or with data storage on either port and the program memory on the MSD bus.

The data space/code space (DS/$\overline{\text{CS}}$) output can be used to control access either to data memory or program memory on the MSD port. Up to 64K words of code space and 64K words of data space are directly supported. In the coprocessor mode, both instructions and data are transferred on the LAD bus with the option of accessing external user-generated programs on the MSD port.

One 32-bit operand can be input to the data registers each clock cycle. A 64-bit double-precision floating-point operand is input in two cycles. Transfers to or from the data registers can normally be programmed as block moves, loading one or more sets of operands with a single move instruction to minimize I/O overhead. Several modes for moving operands and instructions are available. Block transfers up to 512 words between the LAD and MSD buses can be programmed in either direction.

To permit direct input to or output from the LAD bus in the host-independent mode, other options for controlling the LAD bus have been implemented. When two 32-bit operands are being selected for input to the FPU core, one operand may be selected from LAD. On output from the FPU, a result may simultaneously be written to a register and to the LAD bus.

During initialization in the host-independent mode, a bootstrap loader can bring 65 32-bit words from the LAD bus and write them out to external program memory on the MSD bus, after which the device begins executing from the first memory location (zero). The first word is loaded into the configuration register. This option facilitates the initial loading of program memory on the MSD port upon power-up.

architecture

Because the sequencer, control and data registers, and FPU core are closely coupled, the TMS34082A can execute a variety of complex floating-point or integer calculations rapidly, with a minimum of external data transfers. The internal architecture of the FPU core supports concurrent operation of the multiplier and the ALU, providing several options for storing or feeding back intermediate results. Also, several special registers are available to support specific calculations for graphics algorithms. Each of the main architectural elements of the TMS34082A is discussed below.

The control functions of the TMS34082A are provided by sequence control logic, register control logic, and bus interface control logic, together with user-programmed configuration settings stored in the configuration register. The on-board sequencer selects the next program execution address, either from internal code or from external program memory. Next-address sources include the program counter, stack, interrupt vector register, interrupt return register, or address register (for indirect jumps).

COUNTX, COUNTY, and MIN-MAX/LOOPCT registers are used for temporary storage by internal graphics routines. They may also serve as temporary storage for the user.

A separate FPU status register is provided, which can be used by test-and-branch instructions to control program execution. Because of the large number of status outputs, branches on status can be easily programmed. The status register contents are also important when dealing with status exceptions including such conditions as overflow, underflow, invalid operations (divide by zero), or illegal data formats such as infinity, Not a Number (NaN), or denormalized operands.

Register control logic permits all data and control registers to be accessed in accordance with applicable architectural restrictions. Register files A and B can be written to or read from the external buses, as can the control registers. Internal registers C and CT are embedded in the FPU core and can only be accessed by the FPU internal buses. The C and CT registers cannot be used as sources or destinations for MOVE instructions, and several registers (listed in Table 1) are not available as sources for FPU operations.

TABLE 1. INTERNAL REGISTERS

REGISTER ADDRESS	REGISTER NAME	RESTRICTIONS ON USE
00000	RA0	
00001	RA1	
00010	RA2	
00011	RA3	
00100	RA4	
00101	RA5	
00110	RA6	
00111	RA7	
01000	RA8	
01001	RA9	
01010	C†	Not a source or destination for moves
01011	CT†	Not a source or destination for moves
01100	STATUS	Not a source for FPU instructions
01101	CONFIG	Not a source for FPU instructions
01110	COUNTX	Not a source for FPU instructions
01111	COUNTY	Not a source for FPU instructions
10000	RB0	
10001	RB1	
10010	RB2	
10011	RB3	
10100	RB4	
10101	RB5	
10110	RB6	
10111	RB7	
11000	RB8	
11001	RB9	
11010	VECTOR	Not a source for FPU instructions
11011	MCADDR	Not a source for FPU instructions
11100	SUBADD0	Not a source for FPU instructions
11101	SUBADD1	Not a source for FPU instructions
11110	IRAREG	Not a source for FPU instructions
11111	MIN-MAX/LOOPCT	Not a source for FPU instructions

† C and CT registers cannot both be used for FPU operand sources in the same instruction.

register files A and B, feedback registers C and CT

TMS34082A contains two register files, each with ten 64-bit registers and two 64-bit feedback registers. Most instructions will operate on one value from each of the RA and RB register files and return the result to either the RA or RB files or one of the feedback registers.

When the ONEFILE control bit is high in the configuration register, data written to a register in file RA is simultaneously written to the corresponding location in file RB. In this mode, the two register files act as a ten-word, two-read/one-write register file.

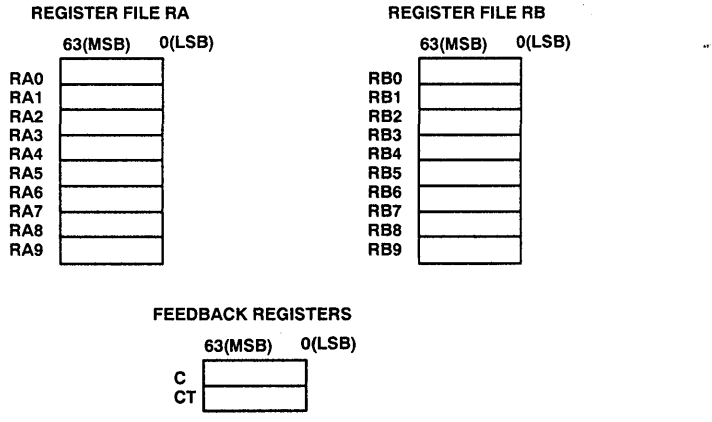


FIGURE 1. DATA REGISTERS

Two 64-bit feedback registers, C and CT, are embedded in the FPU core. FPU instructions may use the feedback registers as one of the operands, but the registers cannot be accessed for external moves. The C and CT registers can be used as either the A or B operand, but both cannot be used as operands during the same instruction. However, C (or CT) may be used for more than one operand in the same instruction. For example, C + CT is not a valid instruction, but C + C is.

The CT feedback register is used in integer divide operations as a temporary holding register. Any data stored in CT will be lost during an integer divide.

internal control/status register definitions

configuration register definition

The configuration register (CONFIG) is a special 32-bit register that the user loads to configure the TMS34082A for exception handling, IEEE mode (vs. fast mode), rounding modes, and data-fetch operations. The configuration register is initialized to 'FFE00420' hex.

TABLE 2. CONFIGURATION REGISTER DEFINITION

BIT NO.	NAME	DESCRIPTION
31	MIVAL	Multiplier invalid operation (I) exception mask. Initialized to 1 (enabled).
30	MOVER	Multiplier overflow (V) exception mask. Initialized to 1 (enabled).
29	MUNDER	Multiplier underflow (U) exception mask. Initialized to 1 (enabled).
28	MINEX	Multiplier inexact (X) exception mask. Initialized to 1 (enabled).
27	MDIV0	Divide by zero (DIV0) exception mask. Initialized to 1 (enabled).
26	MDENORM	Multiplier denormal (DENORM) exception mask. Initialized to 1 (enabled).
25	AIVAL	ALU invalid operation (I) exception mask. Initialized to 1 (enabled).
24	AOVER	ALU overflow (V) exception mask. Initialized to 1 (enabled).
23	AUNDER	ALU underflow (U) exception mask. Initialized to 1 (enabled).
22	AINEX	ALU inexact (X) exception mask. Initialized to 1 (enabled).
21	ADENORM	ALU denormal (DENORM) exception mask. Initialized to 1 (enabled).
11-20	N/A	Reserved, set to all 0s.
10	REVISION	Revision number, read only. Set to 1.
9	LADCFG	When low, $\overline{CAS}$, $\overline{WE}$, CORDY, $\overline{COINT}$, and $\overline{ALTCH}$ are active signals not affected by $\overline{LOE}$. When high, $\overline{LOE}$ high places $\overline{CAS}$ and $\overline{WE}$ in high impedance, as well as the LAD bus. $\overline{COINT}$, which defines the LAD cycle boundaries, is controlled by bit 1 of the LAD move instruction instead of the set mask instruction. $\overline{COINT}$ will remain high unless a LAD move instruction (with bit 1 high) is in progress. The setting of this bit has no effect in the coprocessor mode. Initialized to 0.
8	MEMCFG	When high, $\overline{MCE}$ becomes code space chip enable and DS/$\overline{CS}$ becomes data space chip enable (eliminates need for external inverter). When low, $\overline{MCE}$ is chip select for external code and data space. DS/$\overline{CS}$ functions as an address bit which selects code space (when low) or data space (when high). Initialized to 0.
7	N/A	Reserved for later use. Initialized to 0. Must be loaded with 0.
6	ONEFILE	When high, causes simultaneous write to both register files (for example, to both RA0 and RB0 at once). The register files act as a single two-read, one-write register file. Initialized to 0.
5	PIPES2	When high, makes FPU output registers transparent. When low, registers are enabled. Initialized to 1.
4	PIPES1	When high, makes FPU internal pipeline registers transparent. When low, registers are enabled. Initialized to 0.
3	FAST	When high,fast mode is selected (all denormalized inputs and outputs are 0). When low, IEEE mode is selected. Initialized to 0.
2	LOAD	Load order. 0 = MSH, then LSH; 1 = LSH, then MSH. Initialized to 0.
1	RND1	Rounding mode select 1. Initialized to 0.
0	RND0	Rounding mode select 0. Initialized to 0.

LSH denotes least-significant half of a 64-bit word, MSH denotes most-significant half of a 64-bit word.

The mask bits serve as exception detect enables for the exception masks listed above. Setting the bit high (logic '1') enables the detection of the specific exception. When an enabled exception occurs, the ED bit in the status register will be set high and can be used to generate interrupts. The fast bit allows the TMS34082A to control the handling of denormalized numbers. When the fast bit is set high, all denormalized numbers input to the device are flushed to zero, and all denormalized results are also flushed to zero (this is also called 'sudden underflow'). When the fast bit is low, IEEE mode is selected. Denormalized numbers may be generated by (or input to) the ALU. Denormalized numbers must first be wrapped before being used as operands for multiply or divide instructions.

The LOAD bit defines the expected order of double-precision operands. At reset, this bit will default to 0 indicating that the most significant 32 bits are transferred first. If the bit is set to a 1, then the expected order of 64-bit data transfers starts with the least significant 32 bits.

The RND0 and RND1 bits select the IEEE rounding mode, as shown in Table 3.

TABLE 3. ROUNDING MODE

RND1 - RND0	ROUNDING MODES
0 0	Round towards nearest
0 1	Round toward zero (truncated)
1 0	Round towards infinity (round up)
1 1	Round towards negative infinity (round down)

status register definition

The floating-point status register (STATUS) is a 32-bit register used for reporting the exceptions that occur during TMS34082A operations and status codes set by the results of implicit and explicit compare operations. The status register is cleared upon reset, except for the INTENED flag, which is set to 1 in the coprocessor mode.

TABLE 4. STATUS REGISTER DEFINITION

BIT NO.	NAME	DESCRPTION
31	N	Sign bit (A < B flag for compare)
30	GT	A > B (valid on compare)
29	Z	Zero flag (A = B for compare)
28	V	IEEE overflow flag. The result is greater than the largest allowable value for the specified format.
27	I	IEEE invalid operation flag. A NaN has been input to the multiplier or the ALU, or an invalid operation [(0 * 1) or(∞−∞) or (− ∞+ ∞)] has been requested. This signal also goes high if an operation involves the square root of a negative number. When IVAL hoes high, the STX pins indicate which port had the NaN.
26	U	IEEE underflow flag. The result is inexact and less than the minimum allowable value for the specified format. In fast mode, this condition causes the result to go to zero.
25	X	IEEE inexact flag. The result of an operation is inexact.
24	DIV0	Divide by zero. An invalid operation involving a zero divisor has been detected by the multiplier.
23	RND	The mantissa of a number has been increased in magnitude by rounding. If the number generated was wrapped, then the 'unwrap rounded' instruction must be used to properly unwrap the wrapped number.
22	DENIN	Input to the multiplier is a denormalized number. When DENIN goes high, the STX pins indicate which port has the denormal input.
21	DENORM	The multiplier output is wrapped number or the ALU output is a denormalized number. In fast mode, this condition causes the result to go to zero. It also indicates an invalid integer operation with a negative unsigned integer result.
20	STX1	A NaN or a denormalized number has been input on the A port.
19	STX0	A NaN or a denormalized number has been input on the B port.
18	ED	Exception detect status signal representing logical OR of all enabled exceptions in the configuration register.
17	UNORD	The two inputs of a comparison operation are unordered, i.e.; one or both of the inputs is an NaN.
16	INTFLG	Software interrupt flag. Set by external code to signal a software interrupt.
15	INTENHW	Hardware interrupt ($\overline{\text{INTR}}$) enable, active high (initialized to zero)
14	NXOROV	N (negative) XOR V (overflow)
13	VANDZB	V (overflow) AND $\overline{Z}$ (NOT zero)
12	INTENED	ED interrrupt enable, active high (initialized to zero in the host-independent mode, one in the coprocessor mode)
11	INTENSW	Software interrupt (INTFLG) enable, active high (initialized to zero)
10	ZGT	Zn > Zmax (valid for 2-D MIN-MAX instruction)
9	ZLT	Zn < Zmin (valid for 2-D MIN-MAX instruction)
8	YGT	Yn > Ymax (valid for 1-D or 2-D MIN-MAX instruction)
7	YLT	Yn < Ymin (valid for 1-D or 2-D MIN-MAX instruction)
6	XGT	Xn > Xmax (valid for 1-D or 2-D MIN-MAX instruction)
5	XLT	Xn < Xmin (valid for 1-D or 2-D MIN-MAX instruction)
4	HINT	Hardware interrupt flag
3-0	N/A	Reserved

indirect address register (MCADDR) definition

The indirect address register (MCADDR) can be set to point to a memory location for indirect move or jump operations through the MSD port. MCADDR is cleared upon reset.

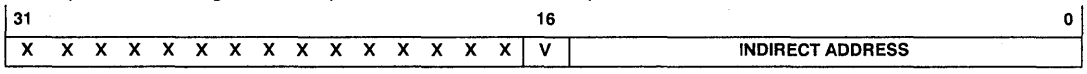


FIGURE 2. INDIRECT ADDRESS DEFINITION

The function of bit 16 varies, depending on whether the instruction is a MOVE or JUMP. During a MOVE instruction, bit 16 selects data space when set high, or code space when low. During a JUMP instruction, bit 16 selects an internal instruction when set high, or an external instruction when low.

stack registers (SUBADD1-SUBADD0) definition

The stack contains two subroutine return address registers, SUBADD0 and SUBADD1, which serves as a two-deep LIFO (last-in, first-out) stack. A subroutine jump causes the program counter to be pushed onto the stack, and a return from subroutine pops the last address pushed on the stack. More than two pushes will overwrite the contents of SUBADD1.

Bit 31 (Pointer) is set high in the stack location that was written last and reset to zero in the other stack location. Setting bit 30 (Enable) high enables a write into bit 31 (set or reset the pointer) in either stack location. If bit 31 is zero in both SUBADD0 and SUBADD1 (as when the stack has been saved externally and later restored), SUBADD0 can be designated as top of stack by setting bit 31. The stack pointers (bit 31) are cleared upon reset.

Bit 16 (I) is set high when the address in a stack location points to an internal routine, or set low when the address is for an external instruction.

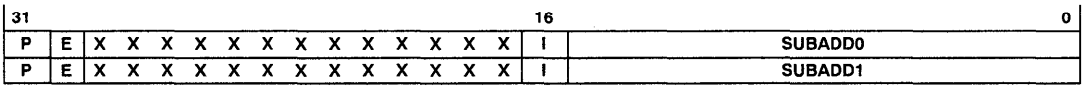


FIGURE 3. STACK DEFINITION

interrupt vector register (VECTOR) definition

The interrupt vector register (VECTOR) serves as a pointer to an external program to be executed upon receipt of an interrupt. Bit 16 (I) is always set low to point to a routine in external code space. The interrupt vector is cleared on reset.

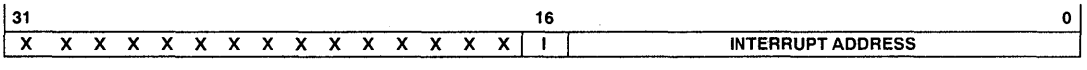


FIGURE 4. INTERRUPT VECTOR DEFINITION

interrupt return register (IRAREG) definition

The interrupt return register (IRAREG) retains a copy of the program counter at the time of an external interrupt. This address is used as the next execution address upon returning from the interrupt. Bit 16 (I) is set high when the address in the stack location points to an internal instruction, or set low when the address is for an external instruction. This register is not affected by the reset signal.

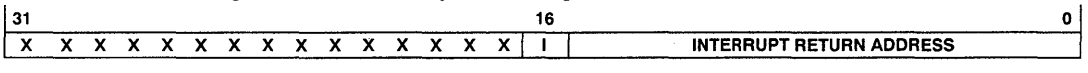


FIGURE 5. INTERRUPT RETURN DEFINITION

COUNTX and COUNTY registers definition

The counter registers (COUNTX, COUNTY) are used to store the current counts of the minimum and maximum values when executing MIN-MAX instructions. COUNTX and COUNTY are cleared on reset.

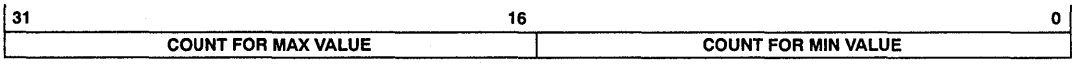


FIGURE 6. COUNTY AND COUNTX REGISTER DEFINITION

The COUNTX register is updated on both the 1-D and 2-D MIN-MAX instruction such that the count of the current minimum value is in the lower 16 bits of the register and the count of the current maximum value is in the upper 16 bits. The COUNTY register is used only in the 2-D MIN-MAX instruction to keep track of the counts of the minimum and maximum for the second value of a pair. The COUNTX and COUNTY registers may also be used for temporary storage when not using the MIN-MAX instructions.

MIN-MAX/LOOPCT register

The MIN-MAX/LOOPCT register stores the current values of two separate counters. The LSH contains the current loop counter, and the MSH is used to hold the current minimum or maximum value of a MIN-MAX operation. The MIN-MAX/LOOPCT register is cleared upon reset. The MIN-MAX/LOOPCT register may also be used for temporary storage when not using the MIN-MAX instructions.

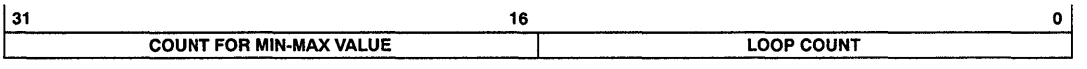


FIGURE 7. MIN-MAX/LOOPCT REGISTER DEFINITION

FPU core

The FPU core itself consists of a multiplier and an ALU, each with an intermediate pipeline register and an output register (see Figure 8, FPU core functional block diagram). Four multiplexers select the multiplier and ALU operands from the data registers, feedback registers, or previous multiplier or ALU result. Results are directed either to the internal feedback registers (C or CT), the 20 data registers in register files RA and RB, or the ten other miscellaneous registers.

Both the internal pipeline registers and the output registers can be enabled or made transparent (disabled) by setting the PIPES2-PIPES1 bits in the configuration register. When the device is powered up, the default settings of the internal registers are PIPES2 high (output registers transparent) and PIPES1 low (internal pipeline registers enabled).

When the FPU core is used for chained operations, the multiplier and ALU operate in parallel. Two data inputs are provided from the RA and RB input registers, while multiplier and ALU feedback are used as the other two operands. While in the chained mode, the output registers of the FPU must be enabled to latch feedback operands. The appropriate registers must be enabled by setting the PIPES2-PIPES1 controls in the configuration register at the beginning of chained operations, and the PIPES2-PIPES1 control should then be reinitialized upon termination.

Fully pipelined operation (both pipeline and output registers enabled) affects timing when writing results back to the RA and RB register files. To adjust writeback timing, it is possible to issue the NOP (no operation) instruction to the FPU core when the results are to be retained in the output registers for one or more additional cycles. The NOP instruction is only effective when the output registers are enabled, as each NOP causes the output register contents to be retained for one additional cycle.

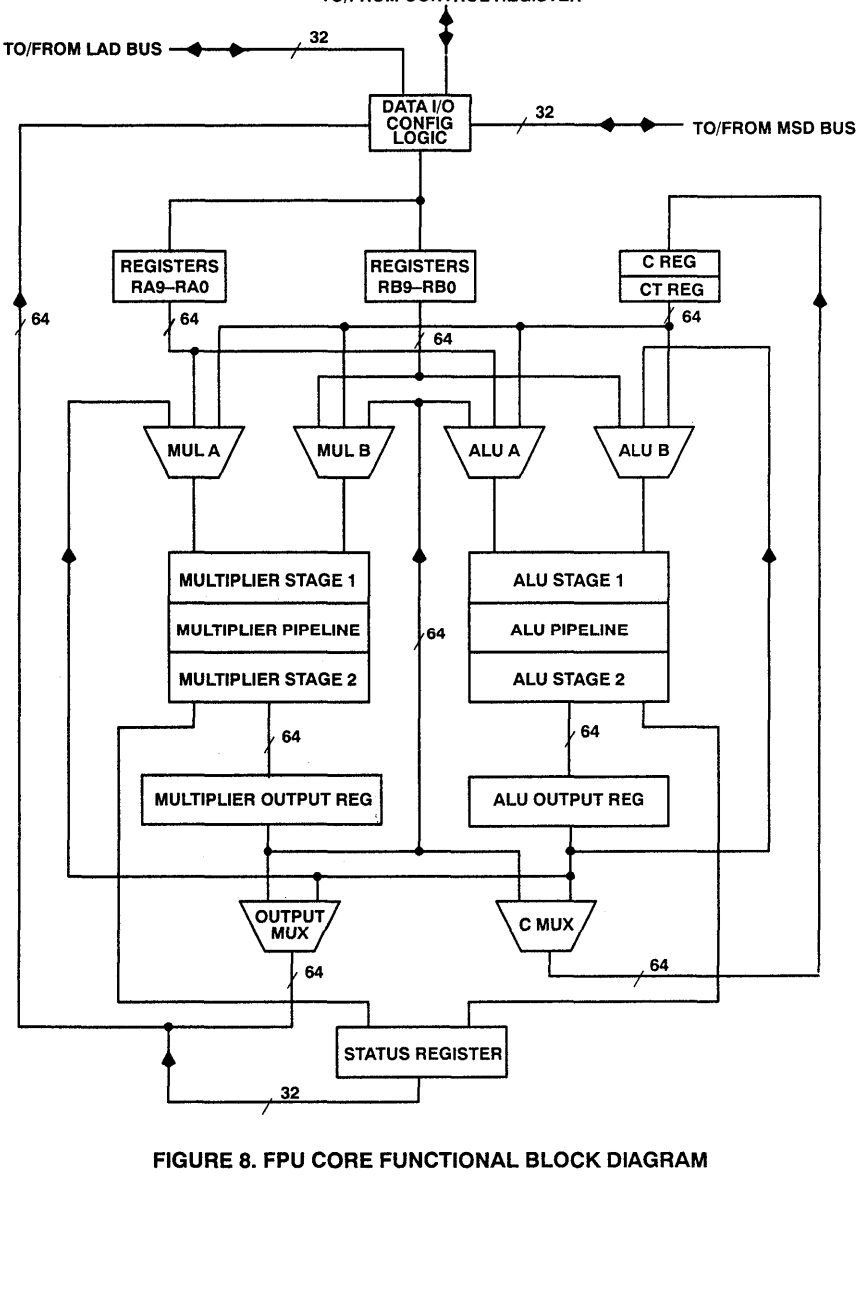


FIGURE 8. FPU CORE FUNCTIONAL BLOCK DIAGRAM

TMS34082A operating modes

The TMS34082A can operate as a stand-alone floating-point processor or a graphics coprocessor to the TMS34020 Graphics System Processor. Control of FPU operation is provided either from external program memory or from the TMS34020. External instructions are addressed by address lines MSA15-0 and are input on MSD31-0. TMS34020 instructions are input on LAD31-0.

Both the MSD and LAD buses can be used for data transfers as well. Combinations of control signals distinguish instruction fetches from data transfers. A single instruction may be used to transfer data and to perform an operation within the FPU.

The TMS34082A supports external code and data storage with the memory expansion interface, MSD31-0. Up to 64K 32-bit data operands and 64K instructions may be added externally to the TMS34082A. The signal DS/$\overline{CS}$ controls whether data space or code space is being accessed, and read/write control is provided with the chip enable ($\overline{MCE}$), output enable ($\overline{MOE}$), address enable ($\overline{MAE}$), write enable ($\overline{MWR}$), and address lines (MSA15-0).

The TMS34082A also provides instructions that allow the TMS34020 to read/write directly from/to external memory. The external code support permits full utilization of the TMS34082A features and instruction set.

coprocessor-mode operation

Operation in the coprocessor mode assumes MSTR is low. In this mode, the TMS34082A acts as a closely coupled coprocessor to the TMS34020. The interface between the two devices consists of direct connections between pins. More than one coprocessor may be connected to the TMS34020 by setting the appropriate coprocessor ID (CID2-CID0). Up to four coprocessors executing in parallel may be used with a single TMS34020.

In the coprocessor mode, clock signals are provided by LCLK1 and LCLK2 from the TMS34020. Internally, the FPU generates a rising clock edge from each LCLK1 edge (rising or falling). Thus, the TMS34082A actually operates at twice the LCLK1 input clock frequency.

initialization (coprocessor mode)

On reset, the TMS34082A clears all pipeline registers and internal states. The configuration register and status register return to their initialization values. When $\overline{RESET}$ returns high in the coprocessor mode, the TMS34082A is in an idle state waiting for the next instruction from the TMS34020.

LAD bus control (coprocessor mode)

Both data and instructions are transferred over the bidirectional LAD bus in the coprocessor mode. A unique combination of signal inputs distinguishes an instruction from data. SF, $\overline{ALTCH}$, $\overline{CAS}$, $\overline{RAS}$, and $\overline{WE}$ are used to designate coprocessor functions from other operations on the LAD bus.

Data may be transferred to or from TMS34020 registers or memory via LAD31-0. Transfers between the LAD and MSD buses can also be programmed. A single coprocessor instruction may be used to transfer data to the TMS34082A and then perform an FPU operation.

MSD bus control (coprocessor mode)

Use of the MSD bus in the coprocessor mode is optional. External memory on MSD31-0 can be used to store data, user-programmed subroutines, or both. Different combinations of control signals distinguish between data memory and code memory. Control signals for MSD and MSA buses operate the same in the host-independent and coprocessor modes.

interrupt handling (coprocessor mode)

A software interrupt to the TMS34082A is generated by the set mask external instruction. When the interrupt is granted, the current program counter is stored in the interrupt return register, and a branch to the interrupt vector address is executed. Software interrupts may be disabled.

If the exception detect interrupt (ED) is enabled, a TMS34082A exception causes $\overline{\text{COINT}}$ to go low, signalling the exception to the TMS34020. This exception does *not* cause a branch to the interrupt vector. If its interrupts are enabled, the TMS34020 will branch to an interrupt vector to service the TMS34082A request. Interrupts are cleared by reading the TMS34082A status register.

host-independent mode operation

Operation in the host-independent mode assumes MSTR high. The TMS34082A has several hardware control signals, as well as programmable features, which support system functions such as initialization, data transfer, or interrupts in the host-independent mode. CLK provides the input clock to the TMS34082A. Details of initialization, LAD and MSD bus interface control, and interrupt handling are provided in the following sections.

initialization (host-independent mode)

To simplify initialization of external program memory, the TMS34082A provides a bootstrap loader to perform an initial program load of 64 instructions. Once invoked, the loader causes the TMS34082A to read 65 words from the LAD bus and write 64 words out to the external program memory on the MSD bus, beginning with location 0. The first word read is used to initialize the configuration register.

This loader is invoked by first setting $\overline{\text{RESET}}$ low, and then $\overline{\text{INTR}}$ low. A separate timing diagram for using the bootstrap loader is provided (see Figure 34). $\overline{\text{INTR}}$ should be taken low after $\overline{\text{RESET}}$ is already low, as shown in the diagram. When the bootstrap loader is started, the FPU core is reset (internal states and status are cleared, but not data registers) and the stack pointer, program counter, and interrupt vector register are all set to zero.

$\overline{\text{RESET}}$ must be set high again before the loader operation can start (see Figure 34). Once the loader is active, an external interrupt (signalled by $\overline{\text{INTR}}$ low) will not be granted until the load sequence is finished. However, $\overline{\text{RESET}}$ going low terminates the load sequence, regardless of whether the sequence is complete. When the load sequence is finished, the device begins program execution at external address 0.

LAD bus control (host-independent mode)

Data transfer from the LAD bus (LAD31-0) is controlled primarily by output signals, $\overline{\text{ALTCH}}$, $\overline{\text{WE}}$, and $\overline{\text{CAS}}$. $\overline{\text{ALTCH}}$ is the address write strobe that signals an address is being output on the LAD bus. The $\overline{\text{CAS}}$ signal is the read strobe, and $\overline{\text{WE}}$ is the write enable output to memory.

If a bidirectional FIFO is used instead of memory, $\overline{\text{CAS}}$ can be directly connected to the read clock and $\overline{\text{WE}}$ to the write clock. The CC input can be used to signal the TMS34082A when data is ready for input from the FIFO stack.

Data input on the LAD bus can be written to data registers, control registers, or passed through for output on the MSD bus. Alternatively, the LAD bus input can be selected directly as an FPU source operand without writing to a register.

An FPU result can be written to a data register and at the same time be passed out on the LAD bus. When this is done, the clock period may need to be extended up to 15 ns (TMS34082A-40) to allow for the propagation delay from the FPU core to the outputs.

Depending on the specific system implementation, transferring data to and from the LAD bus without intervening register operations may significantly improve throughput. In the host-independent mode, data moves to and from internal registers can be minimized at the cost of adjusting the clock period to assure integrity of FPU inputs to and output from the LAD bus.

MSD bus control (host-independent mode)

The MSD bus can be used to access either external data memory or external code memory, depending on the combination of control signals required. If the memory on the MSD port is shared with a host processor, the $\overline{\text{MAE}}$ and RDY signals can be used to prevent conflicts between the host and the TMS34082A. When memory on the MSD port is shared, the host processor can monitor the state of the TMS34082A memory chip enable ($\overline{\text{MCE}}$) to determine when the TMS34082A is not accessing the memory.

Otherwise, the $\overline{\text{MAE}}$ signal may be tied low (if unused), and the TMS34082A can use $\overline{\text{MOE}}$, $\overline{\text{MCE}}$, $\overline{\text{MWR}}$, and DS/$\overline{\text{CS}}$ to control external memory operations into either data space or code space, as selected by DS/$\overline{\text{CS}}$.

interrupt handling (host-independent mode)

Interrupts to the TMS34082A can be signalled by setting the interrupt request input ($\overline{\text{INTR}}$) low. $\overline{\text{INTR}}$ is associated with the vector in the interrupt vector register. Software interrupts are signalled by setting the software interrupt flag in the status register.

In the event of an FPU status exception in the host-independent mode, an interrupt is generated that causes a branch to an exception handler routine. The address of the exception handler is stored in the interrupt vector register by the user prior to execution of the FPU program. Interrupts may be disabled by setting the appropriate bits in the status register.

absolute maximum ratings over operating free-air temperature range (unless otherwise noted)†

Supply voltage, V_{CC} (see Note 1) ..... 6 V
Input voltage range, V_I ..... – 0.3 V to 6 V
Off-state output voltage range ..... – 2 V to 6 V
Operating free-air temperature range ..... – 0°C to 70°C
Storage temperature range ..... – 10°C to 150°C

† Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. These are stress ratings only and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

NOTE 1: All voltage levels are with respect to ground (V_{SS}).

recommended operating conditions

				MIN	NOM	MAX	UNIT
V_{CC}	Supply voltage			4.75	5	5.25	V
V_{SS}	Supply voltage (see Note 2)			0	0	0	V
V_{IH}	High-level input voltage			2		V_{CC}+0.3	V
V_{IL}	Low-level input voltage			– 0.3		0.8	V
I_{OH}	High-level output current					– 8	mA
I_{OL}	Low-level output current					8	mA
f_{clock}	Clock frequency	Coprocessor mode	TMS34082A-32			8	MHz
			TMS34082A-40			10	
		Host-independent mode	TMS34082A-32			16.7	
			TMS34082A-40			20	
T_A	Operating free-air temperature			0		70	°C

NOTE 2: In order to minimize noise on V_{SS}, care should be taken to provide a minimum-inductance path between the V_{SS} pins and system ground.

electrical characteristics over recommended operating free-air temperature range (unless otherwise noted)

PARAMETER			TEST CONDITIONS		MIN	TYP‡	MAX	UNIT
V_{OH}	High-level output voltage		V_{CC} = 4.75 V,	I_{OH} = – 8 mA	2.6			V
V_{OL}	Low-level output voltage		V_{CC} = 4.75 V,	I_{OL} = 8 mA			0.6	V
I_O	High-impedance bidirectional pins output current		V_{CC} = 4.75 V,	V_O = 2.8 V			10	μA
			V_{CC} = 4.75 V,	V_O = 0.6 V			– 10	
I_I	Input current		V_I = V_{SS} to V_{CC}				±5	μA
I_{CC}§	Supply current	Dynamic	V_{CC} = 5.25 V				300	mA
		Quiescent	V_I = V_{IL}max or V_{IH}min,	I_{OH} = I_{OL} = 0			50	mA
			V_I = 0.2 V or V_{CC} – 0.2 V,	I_{OH} = I_{OL} = 0			50	
C_i	Input capacitance					10		pF

‡ All typical values are at V_{CC} = 5 V and T_A = 25°C.

§ I_{CC} is measured at maximum clock frequency. Inputs are presented with random logic highs and lows to assure the toggling of internal nodes.

coprocessor mode (MSTR low)

switching characteristics over recommended ranges of supply voltage and operating free-air temperature (unless otherwise noted)†

propagation delay times

PARAMETER		FIGURE	TMS34082A-32		TMS34082A-40		UNIT
			MIN	MAX	MIN	MAX	
$t_{p(ATCL-CORV)}$	Propagation delay time, $\overline{ALTCH}$ low to CORDY valid	11		40		35	ns
$t_{p(ATCH-LADV)}$	Propagation delay time, $\overline{ALTCH}$ high to LAD data valid	16		35		30	
$t_{p(CASL-LADV)}$	Propagation delay time, $\overline{CAS}$ low to LAD data valid	14		30		25	
$t_{p(CASH-LADZ)}$	Propagation delay time, $\overline{CAS}$ high to LAD disabled	14		30		25	
$t_{p(LC1-DCSL)ML}$	Propagation delay time, LCLK1 ↑ or ↓ to DS/$\overline{CS}$ low with MEMCFG low	17, 21, 23		21		18	
$t_{p(LC1-DCSH)ML}$	Propagation delay time, LCLK1 ↑ or ↓ to DS/$\overline{CS}$ high with MEMCFG low	17, 19, 21, 23, 24, 26		21		18	
$t_{p(LC1-DCSL)MH}$	Propagation delay time, LCLK1 ↑ or ↓ to DS/$\overline{CS}$ low with MEMCFG high	18, 20, 22, 25, 27	3	26	3	18	
$t_{p(LC1-DCSH)ML}$	Propagation delay time, LCLK1 ↑ or ↓ to DS/$\overline{CS}$ high with MEMCFG high	18, 20, 22, 25, 27	3	13	3	11	
$t_{p(LC1-DCSH)ML}$	Propagation delay time, LCK1 ↑ or ↓ to $\overline{MCE}$ low	17-19, 21-27	3	21	3	18	
$t_{p(LC1-DCSH)ML}$	Propagation delay time, LCLK1 ↑ or ↓ to $\overline{MCE}$ high with MEMCFG low	17, 19, 21, 23	3	23	3	18	
$t_{p(LC1-MCEH)MH}$	Propagation delay time, LCLK1 ↑ or ↓ to $\overline{MCE}$ high with MEMCFG high	18, 22, 25, 27	3	13		11	
$t_{p(LC1-MOEL)}$	Propagation delay time, LCLK1 ↑ or ↓ to $\overline{MOE}$ low	17, 18, 21-23, 26, 27	10	30		25	
$t_{p(LC1-MOEH)}$	Propagation delay time, LCLK1 ↑ or ↓ to $\overline{MOE}$ high	17, 18, 21-23, 26, 27	3	13		11	
$t_{p(LC1-MSDV)}$	Propagation delay time, LCLK1 ↑ or ↓ to MSA address valid	17-27		20		18	
$t_{p(LC1-MSDV)}$	Propagation delay time, LCLK1 ↑ or ↓ to MSD data valid	19, 20-22, 24, 25		38		36	
$t_{p(LC1-MWRL)}$	Propagation delay time, LCLK1 ↑ or ↓ to $\overline{MWR}$ low	19-22, 24, 25	10	30	10	25	
$t_{p(LC1-MWRH)}$	Propagation delay time, LCLK1 ↑ or ↓ to $\overline{MWR}$ high	20-22, 24, 25	3	13	3	11	
$t_{p(LC1H-COIL)}$	Propagation delay time, LCLK1 ↑ to $\overline{COINT}$ low	12		23		15	
$t_{p(LC1H-COIH)}$	Propagation delay time, LCLK1 ↑ to $\overline{COINT}$ high	12		23		15	
$t_{p(LC1H-LADV)}$	Propagation delay time, LCLK1 ↑ to LAD data valid	16		28		23	
$t_{p(MSDV-LADV)}$	Propagation delay time, MSD data valid to LAD data valid	26, 27		30		25	
$t_{p(RASH-LADXZ)}$	Propagation delay time, $\overline{RAS}$ high to LAD disabled	16		30		25	

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

coprocessor mode (MSTR low)

switching characteristics over recommended ranges of supply voltage and operating free-air temperature (unless otherwise noted) (continued)†

enable and disable times

PARAMETER		FIGURE	TMS34082A-32		TMS34082A-40		UNIT
			MIN	MAX	MIN	MAX	
$t_{en(LOEL-LADZX)}$	Enable time, $\overline{LOE}$ low to LAD enabled	16	3	15	3	14	ns
$t_{en(MAEL-MSAZX)}$	Enable time, $\overline{MAE}$ low to MSA enabled	21, 22	3	15	3	12	
$t_{en(MAEL-MSDZX)}$	Enable time, $\overline{MAE}$ low to MSD enabled	22	3	15	3	12	
$t_{dis(LOEH-LADXZ)}$	Disable time, $\overline{LOE}$ high to LAD disabled	16	3	15	3	12	ns
$t_{dis(MAEH-MSAXZ)}$	Disable time, $\overline{MAE}$ high to MSA disabled	21, 22	3	15	3	12	
$t_{dis(MAEH-MSDXZ)}$	Disable time, $\overline{MAE}$ high to MSD disabled	21	3	15	3	12	

valid times

PARAMETER		FIGURE	TMS34082A-32		TMS34082A-40		UNIT
			MIN	MAX	MIN	MAX	
$t_{v(MWRH-MSA)}$	Valid time, MSA address after $\overline{MWR}$ high	20-22, 24, 25	1		1		ns
$t_{v(MWRH-MSD)}$	Valid time, MSD output data after $\overline{MWR}$ high	20-22, 24, 25	1		1		
$t_{v(LC1-MSA)}$	Valid time, MSA address valid after LCK ↑ or ↓	17-22, 24-27	3		3		
$t_{v(LC1L-COR)}$	Valid time, CORDY valid after LCLK1 low	11	0		0		

timing requirements over recommended ranges of supply voltage and operating free-air temperature (unless otherwise noted)†

clock period and pulse duration

PARAMETER		FIGURE	PIPELINE CONTROLS	TMS34082A-32		TMS34082A-40		UNIT
			PIPES2-PIPES1	MIN	MAX	MIN	MAX	
$t_{c(LC1)}$	Clock period, LCLK1 ($1/f_{clock}$)	10, 17-22, 24-27	X0	125		100		ns
			11	152		136		
$t_{c(LC2)}$	Clock period, LCLK2 ($1/f_{clock}$)	10	X0	125		100		
			11	152		136		
$t_{w(LC1H)}$	Pulse duration, LCLK1 high	10	X0	52.5		42.5		ns
			11	66		61		
$t_{w(LC1L)}$	Pulse duration, LCLK1 low	10	X0	52.5		42.5		
			11	66		61		
$t_{w(LC2H)}$	Pulse duration, LCLK2 high	10	X0	52.5		42.5		
			11	66		61		
$t_{w(LC2L)}$	Pulse duration, LCLK2 low	10	X0	52.5		42.5		
			11	66		61		
$t_{w(DCSH)MH}$	Pulse duration, DS/$\overline{CS}$ high with MEMCFG high	20, 25, 27	XX	5		7		
$t_{w(RSTL)}$	Pulse duration, $\overline{RESET}$ low	12	XX	30		30		
$t_{w(MCEH)}$	Pulse duration, $\overline{MCE}$ high	18, 25, 27	XX	5		7		
$t_{w(MOEH)}$	Pulse duration, $\overline{MOE}$ high	17, 18, 23, 26, 27	XX	8		8		
$t_{w(MWRH)}$	Pulse duration, $\overline{MWR}$ high	20, 24, 25	XX	8		8		

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

coprocessor mode (MSTR low)

timing requirements over recommended ranges of supply voltage and operating free-air temperature (unless otherwise noted) (continued)†

transition times

PARAMETER		FIGURE	TMS34082A-32		TMS34082A-40		UNIT
			MIN	MAX	MIN	MAX	
$t_{t(LC1)}$	Transition time, LCLK1	10		15		13.5	ns
$t_{t(LC2)}$	Transition time, LCLK2	10		15		13.5	

setup and hold times

PARAMETER		FIGURE	TMS34082A-32		TMS34082A-40		UNIT
			MIN	MAX	MIN	MAX	
$t_{su(BUS-LC2H)}$	Setup time, BUSFLT valid before LCLK2 ↑	11	20		13		ns
$t_{su(CC-LC1)}$	Setup time, CC valid before LCLK1 ↑ or ↓	12	7		5		
$t_{su(LAD-ATCL)}$	Setup time, LAD address valid before $\overline{ALTCH}$ low	13-16, 23	15		12		
$t_{su(LAD-CASH)}$	Setup time, LAD address valid before $\overline{cas}$ high	13, 15, 24, 25	13		10		
$t_{su(LRD-LC2H)}$	Setup time, LRDY valid before LCLK2 ↑	11	20		13		
$t_{su(MSD-LC1)}$	Setup time, MSD data valid before LCLK1 ↑ or ↓	17, 18, 23	11		7		
$t_{su(RASH-ATCL)}$	Setup time, $\overline{RAS}$ high before $\overline{ALTCH}$ low	13-15, 23	35		30		
$t_{su(RDYL-LC1)}$	Setup time, RDY low before LCLK1 ↑ or ↓	12	20		10		
$t_{su(RSTH-LC1)}$	Setup time, $\overline{RESET}$ high before LCLK1 ↑ or ↓	12	40		40		
$t_{su(SF-ATCL)}$	Setup time, SF valid before $\overline{ALTCH}$ low	13-16, 23	15		10		
$t_{su(WEL-CASL)}$	Setup time, $\overline{WE}$ low for data write before $\overline{CAS}$ low	13, 16	15		12		
$t_{h(ATCH-SF)}$	Hold time, SF valid after $\overline{ALTCH}$ high	13-15, 23	15		12		ns
$t_{h(ATCL-LAD)}$	Hold time, LAD address valid after $\overline{ALTCH}$ low	13-16, 23	21		13		
$t_{h(CASH-LAD)}$	Hold time, LAD data valid after $\overline{CAS}$ high	13, 15, 24, 25	0		0		
$t_{h(CASH-SF)}$	Hold time, SF valid after $\overline{CAS}$ high	13-15, 23	15		12		
$t_{h(LC1-CC)}$	Hold time, CC valid after LCLK1 ↑ or ↓	12	3		3		
$t_{h(LC1-MSD)}$	Hold time, MSD input data valid after LCLK1 ↑ or ↓	17, 18, 23	5		5		
$t_{h(LC1-RDY)}$	Hold time, RDY valid after LCLK1 ↑ or ↓	12	3		3		
$t_{h(LC1H-LC2L)}$	Hold time, LCLK2 low after LCLK1 high	10	16		12		
$t_{h(LC2H-BUS)}$	Hold time, BUSFLT valid after LCLK2 high	11	0		0		
$t_{h(LC2H-LC1H)}$	Hold time, LCLK1 high after LCLK2 high	10	16		12		
$t_{h(LC2H-LRD)}$	Hold time, LRDY valid after LCLK2 high	11	0		0		
$t_{h(WEH-SF)}$	Hold time, SF valid after $\overline{WE}$ high	13	15		12		

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

coprocessor mode (MSTR low)

timing requirements over recommended ranges of supply voltage and operating free-air temperature (unless otherwise noted) (continued)†

delay times

PARAMETER		FIGURE	TMS34082A-32		TMS34082A-40		UNIT
			MIN	MAX	MIN	MAX	
$t_{d(DCSL-MCEL)MH}$	Delay time, DS/$\overline{CS}$ high to $\overline{MCE}$ low with MEMCFG high	18, 22	4		4		ns
$t_{d(DCSH-MWRL)}$	Delay time, DS/$\overline{CS}$ high to $\overline{MWR}$ low	19, 24	6		6		
$t_{d(MCEH-DCSL)MH}$	Delay time, $\overline{MCE}$ high to DS/$\overline{CS}$ low with MEMCFG high	20	4		4		
$t_{d(MCEH-MWRL)}$	Delay time, $\overline{MCE}$ high to $\overline{MWR}$ low	25	7		7		
$t_{d(MOEH-MWRL)}$	Delay time, $\overline{MOE}$ high to $\overline{MWR}$ low	19	7		7		
$t_{d(MSAV-MWRL)}$	Delay time, MSA valid to $\overline{MWR}$ low	20-22, 24, 25	5		5		
$t_{d(MSDZ-MOEL)}$	Delay time, MSD disabled to $\overline{MOE}$ low	21, 22	3		3		
$t_{d(MWRH-MCEL)MH}$	Delay time, $\overline{MWR}$ high to $\overline{MCE}$ low with MEMCFG high	25	4		4		
$t_{d(MWRH-MOEL)}$	Delay time, $\overline{MWR}$ high to $\overline{MOE}$ low	19, 21, 22	7		7		
$t_{d(MWRH-MSDVZ)}$	Delay time, $\overline{MWR}$ high to MSD disabled	21	1	9	1	9	
$t_{d(MWRL-MSDZX)}$	Delay time, $\overline{MWR}$ low to MSD enabled	21, 22	0	7	0	7	

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

host-independent mode (MSTR high)

switching characteristics over recommended ranges of supply voltage and operating free-air temperature (unless otherwise noted)†

propagation delay times

PARAMETER		FIGURE	TMS34082A-32		TMS34082A-40		UNIT
			MIN	MAX	MIN	MAX	
$t_{p(CLKH-ATCH)}$	Propagation delay time, CLK ↑ to $\overline{ALTCH}$ high	29, 30		10		8	ns
$t_{p(CLKH-ATCL)}$	Propagation delay time, CLK ↑ to $\overline{ALTCH}$ low	29, 30		23		20	
$t_{p(CLKH-CASH)}$	Propagation delay time, CLK ↑ to $\overline{CAS}$ high	29, 31, 32, 34-36		10		8	
$t_{p(CLKH-CASL)}$	Propagation delay time, CLK ↑ to $\overline{CAS}$ low	29, 31, 32, 34-36		23		20	
$t_{p(CLKH-COIH)}$	Propagation delay time, CLK ↑ to $\overline{COINT}$ high	29-31, 33, 35, 36, 46		20		15	
$t_{p(CLKH-COIL)}$	Propagation delay time, CLK ↑ to $\overline{COINT}$ low	29-31, 33, 35, 36, 46		20		15	
$t_{p(CLKH-CORH)}$	Propagation delay time, CLK ↑ to CORDY high	46		20		15	
$t_{p(CLKH-CORL)}$	Propagation delay time, CLK ↑ to CORDY low	46		20		15	
$t_{p(CLKH-DCSH)MH}$	Propagation delay time, CLK ↑ to DS/$\overline{CS}$ high with MEMCFG high	36, 38, 40, 42-44	1	10	1	10	
$t_{p(CLKH-DCSH)ML}$	Propagation delay time, CLK ↑ to DS/$\overline{CS}$ high with MEMCFG low	35, 37, 39, 41, 45, 46		20		17	
$t_{p(CLKH-DCSL)MH}$	Propagation delay time, CLK ↑ to DS/$\overline{CS}$ low with MEMCFG high	36, 38, 40, 42-44	3	20	3	17	
$t_{p(CLKH-DCSL)ML}$	Propagation delay time, CLK ↑ to DS/$\overline{CS}$ low with MEMCFG low	37, 41, 45-47		20		17	
$t_{p(CLKH-ITGH)}$	Propagation delay time, CLK ↑ to INTG high‡	47		20		15	
$t_{p(CLKH-ITGL)}$	Propagation delay time, CLK ↑ to INTG low	47		25		15	
$t_{p(CLKH-LADV)}$	Propagation delay time, CLK ↑ to LAD valid	29, 30, 33-35, 43, 44		30		25	
$t_{p(CLKH-MCEH)MH}$	Propagation delay time, CLK ↑ to $\overline{MCE}$ high with MEMCFG high	36, 38, 42-46	1	10	1	10	
$t_{p(CLKH-MCEH)ML}$	Propagation delay time, CLK ↑ to $\overline{MCE}$ high with MEMCFG low	37, 39, 41, 45-47	2	20	2	17	
$t_{p(CLKH-MCEL)}$	Propagation delay time, CLK ↑ to $\overline{MCE}$ low	35-39, 41-47	3	20	3	17	
$t_{p(CLKH-MOEH)}$	Propagation delay time, CLK ↑ to $\overline{MOE}$ high	37, 38, 41-47	1	10	1	10	
$t_{p(CLKH-MOEL)}$	Propagation delay time, CLK ↑ to $\overline{MOE}$ low	37, 38, 41-47	10	28	10	25	
$t_{p(CLKH-MSAV)}$	Propagation delay time, CLK ↑ to MSA address valid	35-47		20		17	
$t_{p(CLKH-MSDV)}$	Propagation delay time, CLK ↑ to MSD data valid	35, 36, 39-42		35		33	
$t_{p(CLKH-MWRH)}$	Propagation delay time, CLK ↑ to $\overline{MWR}$ high	35, 36, 40-42	1	10	1	10	
$t_{p(CLKH-MWRL)}$	Propagation delay time, CLK ↑ to $\overline{MWR}$ low	35, 36, 39-42	10	28	10	25	
$t_{p(CLKH-WEH)}$	Propagation delay time, CLK ↑ to $\overline{WE}$ high	30, 33, 43, 44		10		8	
$t_{p(CLKH-WEL)}$	Propagation delay time, CLK ↑ to $\overline{WE}$ low	30, 33, 43, 44		23		20	

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

‡ Interrupts are not granted during multicycle instructions.

host-independent mode (MSTR high)

switching characteristics over recommended ranges of supply voltage and operating free-air temperature (unless otherwise noted) (continued)†

enable and disable times

PARAMETER		FIGURE	TMS34082A-32		TMS34082A-40		UNIT
			MIN	MAX	MIN	MAX	
t_{en}(CLKH-LADZX)	Enable time, CLK high to LAD enabled	29, 30	5		5		ns
t_{en}(LOEL-LADZX)	Enable time, $\overline{LOE}$ low to LAD enabled	33	5	18	5	14	
t_{en}(MAEL-MSAZX)	Enable time, $\overline{MAE}$ low to MSA enabled	41, 42	3	15	3	12	
t_{en}(MAEL-MSDZX)	Enable time, $\overline{MAE}$ low to MSD enabled	42	3	15	3	12	
t_{dis}(CLKH-LADXZ)	Disable time, CLK high to LAD disabled‡	29, 30		25		23	ns
t_{dis}(LOEH-LADXZ)	Disable time, $\overline{LOE}$ high to LAD disabled	33	5	15	5	12	
t_{dis}(MAEH-MSAXZ)	Disable time, $\overline{MAE}$ high to MSA disabled	41, 42	3	15	3	12	
t_{dis}(MAEH-MSDXZ)	Disable time, $\overline{MAE}$ high to MSD disabled	42	3	15	3	12	

valid times

PARAMETER		FIGURE	TMS34082A-32		TMS34082A-40		UNIT
			MIN	MAX	MIN	MAX	
t_v(ATCH-LAD)	Valid time, LAD output data after $\overline{ALTCH}$ high	29, 30	2		2		ns
t_v(CLKH-MSA)	Valid time, MSA address valid after CLK high	35-47	3		3		
t_v(MWRH-MSD)	Valid time, MSD data valid after $\overline{MWR}$ high	35, 36, 40-42	1		1		
t_v(MWRH-MSA)	Valid time, MSA address valid after $\overline{MWR}$ high	35, 36, 40-41	1		1		
t_v(WEH-LAD)	Valid time, LAD data valid after $\overline{WE}$	30, 33, 43, 44	2		2		

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

‡ Valid only for last write in sesries. The LAD bus is not placed in high-impedance state between consecutive outputs.

host-independent mode (MSTR high)

timing requirements over recommended ranges of supply voltage and operating free-air temperatre (unless otherwise noted)†

clock period and pulse duration

PARAMETER		FIGURE	PIPELINE CONTROLS	TMS34082A-32		TMS34082A-40		UNIT
			PIPES2-PIPES1	MIN	MAX	MIN	MAX	
$t_{c(CLK)}$	Clock period time, CLK ($1/f_{clock}$)	28-31, 33-48	X0 11	60 66		50 61		ns
$t_{w(ATCH)}$	Pulse duration, $\overline{ALTCH}$ high	30	XX	7		7		ns
$t_{w(CASH)}$	Pulse duration, $\overline{CAS}$ high	29, 31, 32, 35, 36	XX	7		7		
$t_{w(CLKH)}$	Pulse duration, CLK high	28	XX	15		15		
$t_{w(CLKL)}$	Pulse duration, CLK low	28	XX	15		15		
$t_{w(DCSH)}$	Pulse duration, DS/$\overline{CS}$ high	36, 40, 44	XX	5		5		
$t_{w(ITRL)}$	Pulse duration, $\overline{INTR}$ low	34, 47	XX	20		15		
$t_{w(MCEH)}$	Pulse duration, $\overline{MCE}$ high	36, 38, 44-46	XX	5		5		
$t_{w(MOEH)}$	Pulse duration, $\overline{MOE}$ high	37, 38, 43-46	XX	8		8		
$t_{w(MWRH)}$	Pulse duration, $\overline{MWR}$ high	35, 36, 40	XX	8		8		
$t_{w(RSTL)}$	Pulse duration, $\overline{RESET}$ low	34	XX	30		20		
$t_{w(WEH)}$	Pulse duration, $\overline{WE}$ high	30, 33, 43, 44	XX	7		7		

transition time

PARAMETER		FIGURE	TMS34082A-32		TMS34082A-40		UNIT
			MIN	MAX	MIN	MAX	
$t_{t(CLK)}$	Transition time, CLK	28		15		15	ns

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

host-independent mode (MSTR high)

timing requirements over recommended ranges of supply voltage and operating free-air temperatre (unless otherwise noted) (continued)†

setup and hold times

PARAMETER		FIGURE	TMS34082A-32		TMS34082A-40		UNIT
			MIN	MAX	MIN	MAX	
$t_{su(CC-CLKH)}$	Setup time, CC before CLK high	45	7		5		ns
$t_{su(LADV-CLKL)}$	Setup time, LAD data valid before CLK low for immediate data input‡	32	10		10		
$t_{su(ITRL-CLKH)}$	Setup time, $\overline{INTR}$ before CLK high	47	20		10		
$t_{su(LAD-CLKH)}$	Setup time, LAD input data valid before CLK high	29, 31, 34-36	9		9		
$t_{su(LRD-CLKH)}$	Setup time, LRDY before CLK high	48	20		15		
$t_{su(MSD-CLKH)}$	Setup time, MSD data valid before CLK high	37, 38, 43-47	10		8		
$t_{su(RDYV-CLKH)}$	Setup time, RDY valid before CLK high	48	20		10		
$t_{su(RSTH-CLKH)}$	Setup time, $\overline{RESET}$ high before CLK high	34	40		40		
$t_{su(RSTL-ITRL)}$	Setup time, $\overline{RESET}$ low before $\overline{INTR}$ low for bootstrap loader	34	10		10		ns
$t_{h(CLKH-CC)}$	Hold time, CC after CLK high	45	0		0		
$t_{h(CLKH-ITR)}$	Hold time, $\overline{INTR}$ after CLK high	47	0		0		
$t_{h(CLKH-LAD)}$	Hold time, LAD input data valid after CLK high	29, 31, 35, 36	3		3		
$t_{h(CLKH-LRD)}$	Hold time, LRDY after CLK high	48	0		0		
$t_{h(CLKH-MSD)}$	Hold time, MSD data valid after CLK high	37, 38, 43-47	2		2		
$t_{h(CLKH-RDY)}$	Hold time, RDY after CLK high	48	0		0		
$t_{h(CLKL-LAD)}$	Hold time, LAD data after CLK low for immediate data input‡	32	5		5		
$t_{h(ITRL-RSTH)}$	Hold time, $\overline{RESET}$ low after $\overline{INTR}$ low for bootstrap loader	34	10		10		

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

‡ This mode permits data input that does not meet the minimum setup before CLK high. The clock period for this mode must be extended according to the equation:

$$Adjusted\ clock\ period = Normal\ clock\ period + Data\ delay + 5\ ns$$

The data delay is the delay from CLK high to valid data. This mode may not be used to input data for divides or square roots.

host-independent mode (MSTR high)

timing requirements over recommended ranges of supply voltage and operating free-air temperature (unless otherwise noted) (continued)†

delay times

PARAMETER		FIGURE	TMS34082A-32		TMS34082A-40		UNIT
			MIN	MAX	MIN	MAX	
$t_{d(ATCH-CASL)}$	Delay time, $\overline{ALTCH}$ high to $\overline{CAS}$ low	29	6		6		ns
$t_{d(ATCH-WEL)}$	Delay time, $\overline{ALTCH}$ high to $\overline{WE}$ low	30	5		5		
$t_{d(CASH-ATCL)}$	Delay time, $\overline{CAS}$ high to $\overline{ALTCH}$ low	29	5		5		
$t_{d(CASH-WEL)}$	Delay time, $\overline{CAS}$ high to $\overline{WE}$ low	33	5		5		
$t_{d(COIL-ATCL)}$	Delay time, $\overline{COINT}$ low to $\overline{ALTCH}$ low	29, 30	0		0		
$t_{d(COIL-CASL)}$	Delay time, $\overline{COINT}$ low to $\overline{CAS}$ low	31, 35, 36	2		2		
$t_{d(COIL-WEL)}$	Delay time, $\overline{COINT}$ low to $\overline{WE}$ low	33	0		0		
$t_{d(DCSH-MCEL)MH}$	Delay time, DS/$\overline{CS}$ high to $\overline{MCE}$ low with MEMCFG high	38, 42	4		4		
$t_{d(DCSH-MWRL)}$	Delay time, DS/$\overline{CS}$ high to $\overline{MWR}$ low	35, 39	6		6		
$t_{d(MCEH-DCSL)MH}$	Delay time, $\overline{MCE}$ high to DC/$\overline{CS}$ low with MEMCFG high	40	4		4		
$t_{d(MCEH-MWRL)}$	Delay time, $\overline{MCE}$ high to $\overline{MWR}$ low	36	7		7		
$t_{d(MOEH-MWRL)}$	Delay time, $\overline{MOE}$ high to $\overline{MWR}$ low	39	7		7		
$t_{d(MSAV-MWRL)}$	Delay time, MSA valid to $\overline{MWR}$ low	35, 36, 40-42	5		5		
$t_{d(MSDZ-MOEL)}$	Delay time, MSD disabled to $\overline{MOE}$ low	41, 42	3		3		
$t_{d(MWRH-MCEL)MH}$	Delay time, $\overline{MWR}$ high to $\overline{MCE}$ low with MEMCFG high	36	4		4		
$t_{d(MWRH-MOEL)}$	Delay time, $\overline{MWR}$ high to $\overline{MOE}$ low	41, 42	7		7		
$t_{d(MWRH-MSDXZ)}$	Delay time, $\overline{MWR}$ high to MSD disabled	42	1	9	1	9	
$t_{d(MWRL-MSDZX)}$	Delay time, $\overline{MWR}$ low to MSD enabled	41, 42	0	7	0	7	
$t_{d(WEH-ATCL)}$	Delay time, $\overline{WE}$ high to $\overline{ALTCH}$ low	29	5		5		
$t_{d(WEH-CASL)}$	Delay time, $\overline{WE}$ high to $\overline{CAS}$ low	31	5		5		

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

EXPLANATION OF LETTER SYMBOLS

This data sheet uses a type of letter symbol based on JEDEC Std-100 and IEC Publication 748-2, 1985, to describe time intervals. The format is:

$$t_{A(BC\text{-}DE)F}$$

Where:

Subscript A indicates the type of dynamic parameter being represented. One of the following is used:

Switching Characteristics:
- p = Propagation delay time
- en = Enable time
- dis = Disable time

Timing Requirements:
- c = Clock period
- w = Pulse duration
- t = Transition time
- d = Delay time
- su = Setup time
- h = Hold time
- v = Valid time

Subscript B indicates the name of the signal or terminal for which a change of state or level (or establishment of a state or level) constitutes a signal event assumed to occur first, that is, at the beginning of the time interval.

Subscript C indicates the direction of the transistion and/or the final state or level of the signal represented by B. One or two of the following are used:

- H = High or transition to high
- L = Low or transition to low
- V = A valid steady-state level
- X = Unknown, changing, or "don't care" level
- Z = High-impedance (off) state

Subscript D indicates the name of the signal or terminal for which a change of state or level (or establishment of a state or level) constitutes a signal event assumed to occur last, that is, at the end of the time interval.

Subscript E indicates the direction of the transition and/or the final state or level of the signal represented by D. One or two of the symbols described in *Subscript C* are used.

Subscript F indicates additional information such as mode of operation, test conditions, etc.

The hyphen between the C and D subscripts is omitted when no confusion is likely to occur. For these letter symbols on this data sheet, the signal names are further abbreviated as follows:

SIGNAL NAME	B & D SUBSCRIPT	SIGNAL NAME	B & D SUBSCRIPT	SIGNAL NAME	B & D SUBSCRIPT	SIGNAL NAME	B & D SUBSCRIPT	SIGNAL NAME	B & D SUBSCRIPT
$\overline{\text{ALTCH}}$	ATC	CORDY	COR	LCLK2	LC2	MSA(0:15)	MSA	TCK	TCK
BUSFLT	BFT	DC/$\overline{\text{CS}}$	DCS	$\overline{\text{LOE}}$	LOE	MSD(0:31)	MSD	TDI	TDI
$\overline{\text{CAS}}$	CAS	EC(0:1)	EC	LRDY	LRD	$\overline{\text{MWR}}$	MWR	TDO	TDO
CC	CC	INTG	INT	$\overline{\text{MAE}}$	MAE	$\overline{\text{RAS}}$	RAS	TMS	TMS
CID(0:2)	CID	$\overline{\text{INTR}}$	ITR	MSTR	MST	RDY	RDY	V_{CC}/V_{SS}	—
CLK	CLK	LAD(0:31)	LAD	$\overline{\text{MCE}}$	MCE	$\overline{\text{RESET}}$	RST	$\overline{\text{WE}}$	WE
$\overline{\text{COINT}}$	COI	LCLK1	LC1	$\overline{\text{MOE}}$	MOE	SF	SF	MEMCFG	M

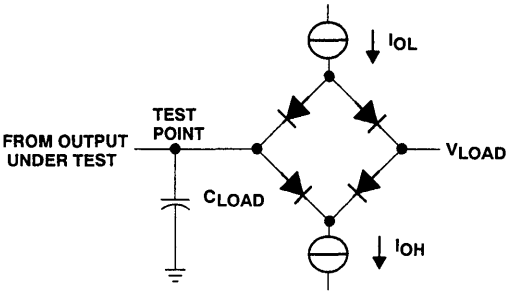
PARAMETER MEASUREMENT INFORMATION

LOAD CIRCUIT PARAMETERS

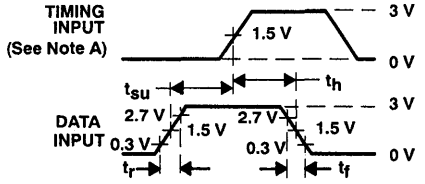
TIMING PARAMETERS		C_{LOAD}† (pF)	I_{OL} (mA)	I_{OH} (mA)	V_{LOAD} (V)
t_{en}	t_{PZH}	65	8	−8	0
	t_{PZL}				3
t_{dis}	t_{PHZ}	65	8	−8	1.5
	t_{PLZ}				
t_p		65	8	−8	‡

† C_{LOAD} includes the typical load circuit and distributed capacitance.

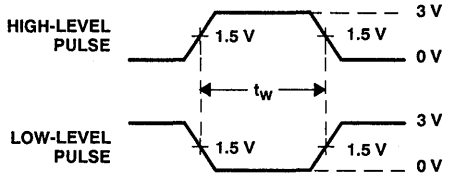
‡ $\frac{V_{LOAD} - V_{OL}}{I_{OL}}$ = 50 Ω, where V_{OL} = 0.6 V, I_{OL} = 8 mA.



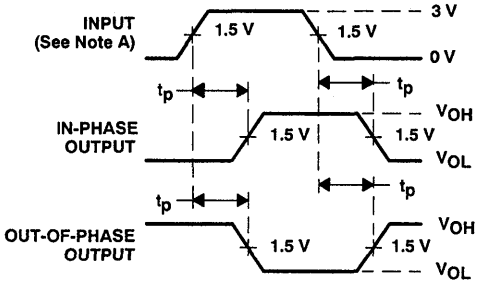
LOAD CIRCUIT



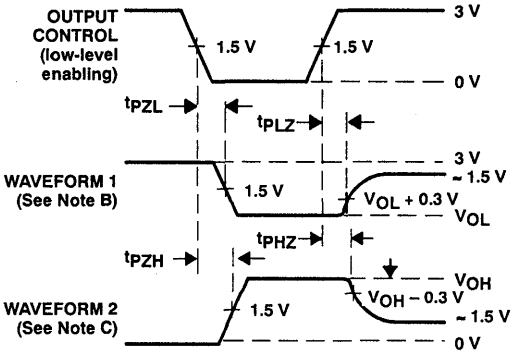
VOLTAGE WAVEFORMS
SETUP AND HOLD TIMES
INPUT RISE AND FALL TIMES



VOLTAGE WAVEFORMS
PULSE DURATION



VOLTAGE WAVEFORMS
PROPAGATION DELAY TIMES



VOLTAGE WAVEFORMS
ENABLE AND DISABLE TIMES, 3-STATE OUTPUTS

NOTES: A. Phase relationships between waveforms were chosen arbitrarily. All input pulses are supplied by pulse generators having the following characteristics: PRR = 1 MHz, Z_O = 50 Ω, $t_r \leq 6$ ns, $t_f \leq 6$ ns.

B. Waveform 1 is for an output with internal conditions such that the output is low except when disabled by the output control.

C. Waveform 2 is for an output with internal conditions such that the output is high except when disabled by the output control. For t_{PLZ} and t_{PHZ}, V_{OL} and V_{OH} are measured values.

FIGURE 9

PARAMETER MEASUREMENT INFORMATION

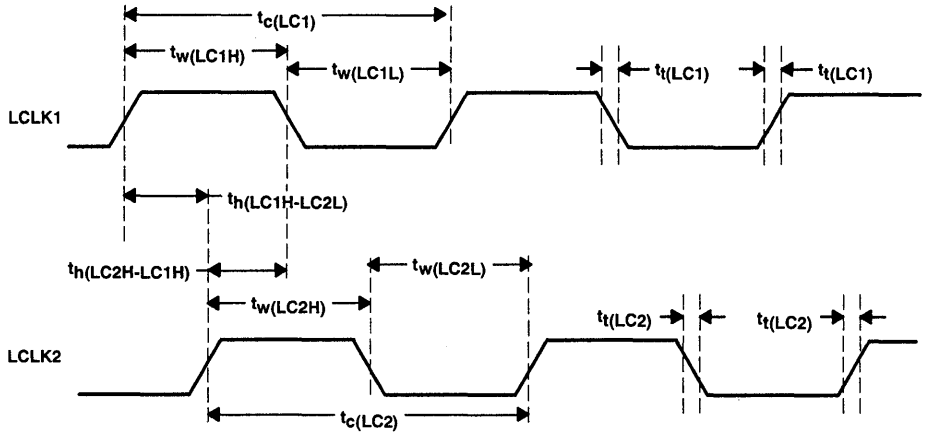
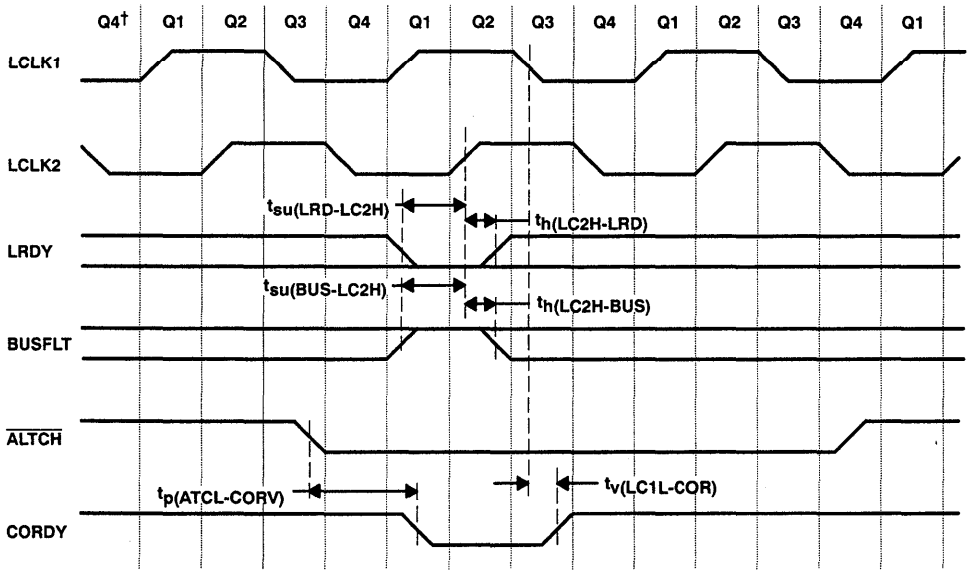


FIGURE 10. COPROCESSOR MODE, INPUT CLOCKS



† Q1, Q2, Q3, and Q4 represent the first, second, third, and fourth quarter clocks, respectively, of the LCLK1 clock period.

FIGURE 11. COPROCESSOR MODE, BUS CONTROL SIGNALS

PARAMETER MEASUREMENT INFORMATION

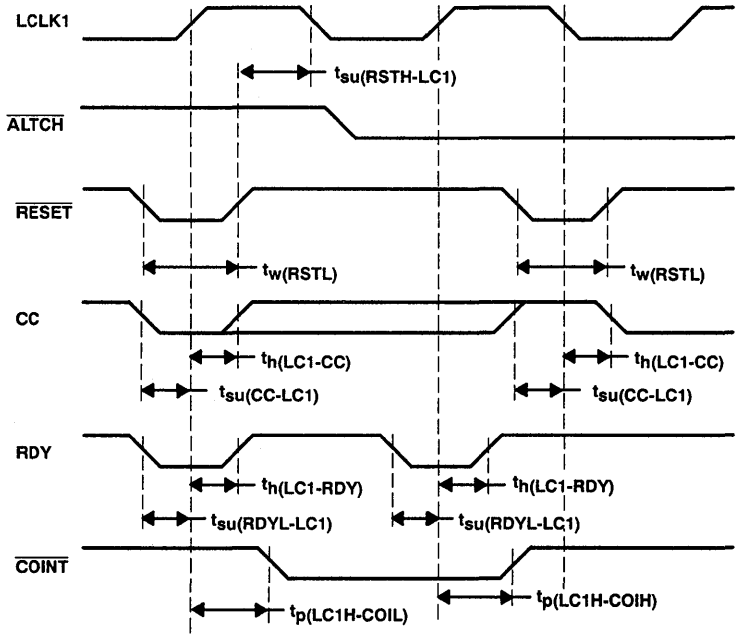
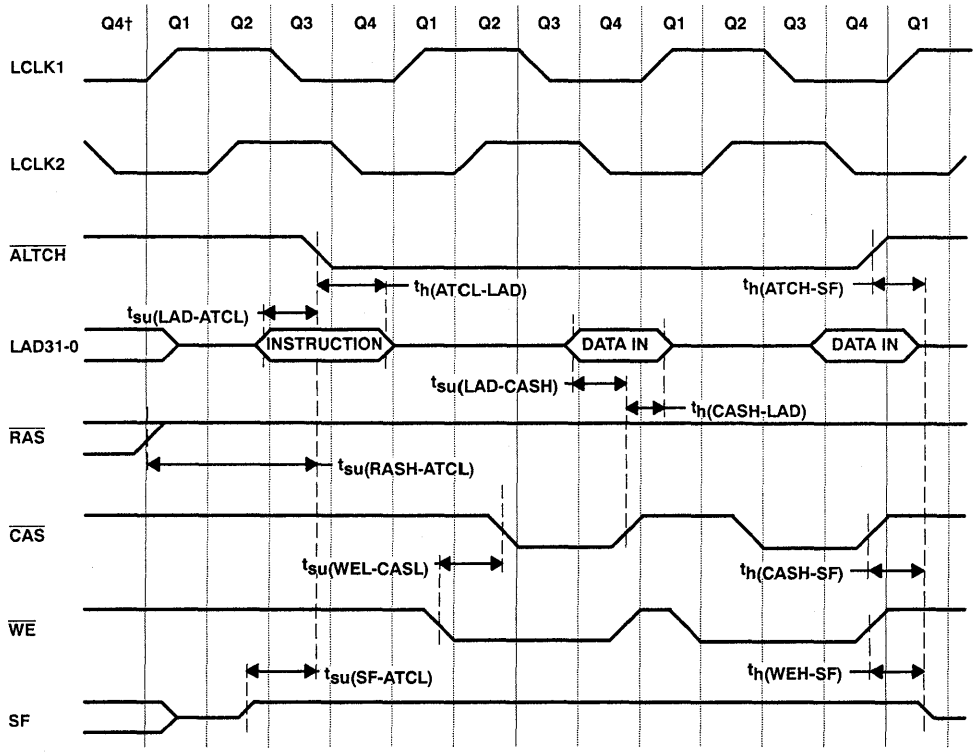


FIGURE 12. COPROCESSOR MODE, CONTROL SIGNALS

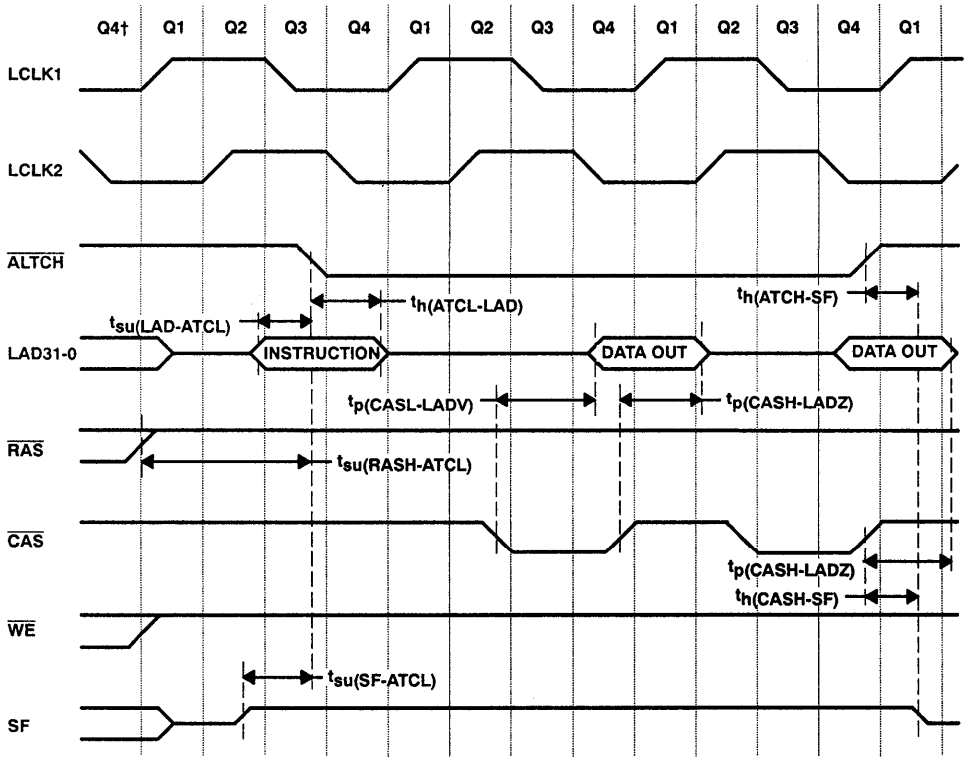
PARAMETER MEASUREMENT INFORMATION



† Q1, Q2, Q3, and Q4 represent the first, second, third, and fourth quarter clocks, respectively, of the LCLK1 clock period.

FIGURE 13. COPROCESSOR MODE, TMS34020 GSP TO TMS34082A

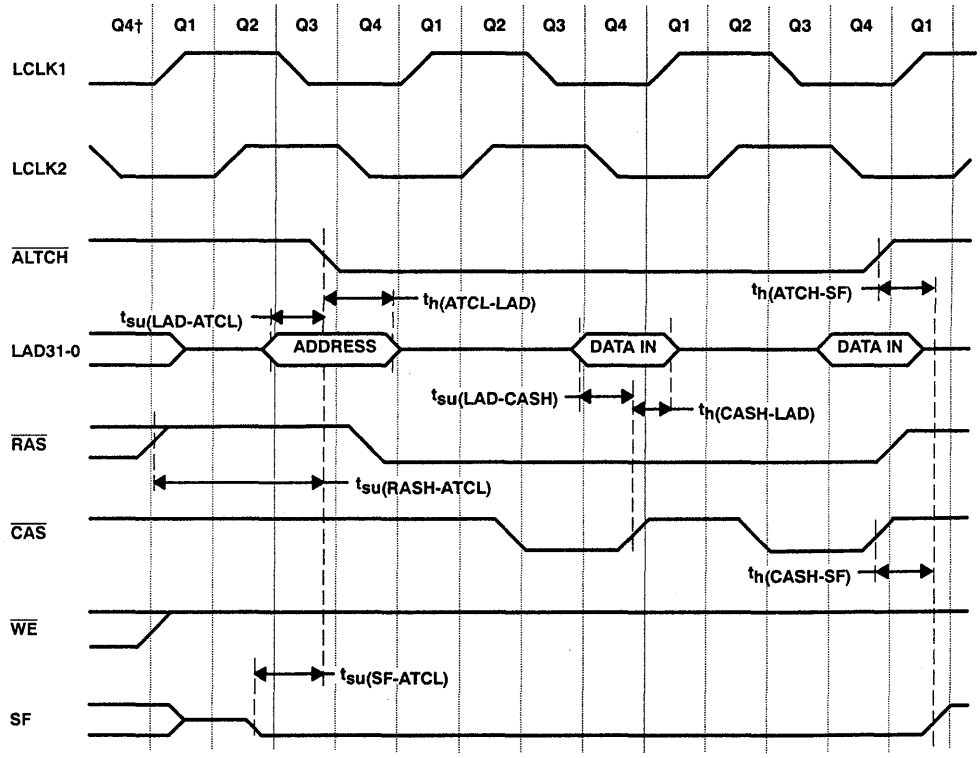
PARAMETER MEASUREMENT INFORMATION



† Q1, Q2, Q3, and Q4 represent the first, second, third, and fourth quarter clocks, respectively, of the LCLK1 clock period.

FIGURE 14. COPROCESSOR MODE, TMS34082A TO TMS34020 GSP INCLUDING COPROCESSOR INTERNAL CYCLE

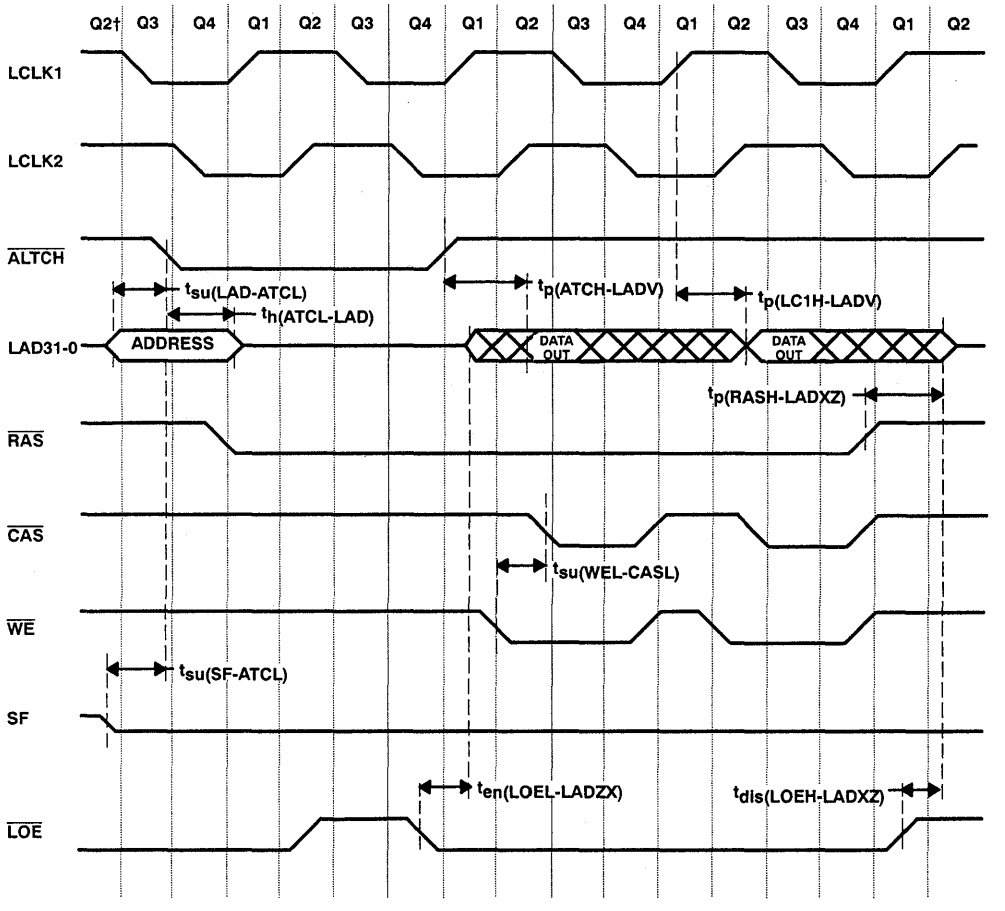
PARAMETER MEASUREMENT INFORMATION



† Q1, Q2, Q3, and Q4 represent the first, second, third, and fourth quarter clocks, respectively, of the LCLK1 clock period.

FIGURE 15. COPROCESSOR MODE, DRAM/VRAM MEMORY TO TMS34082A

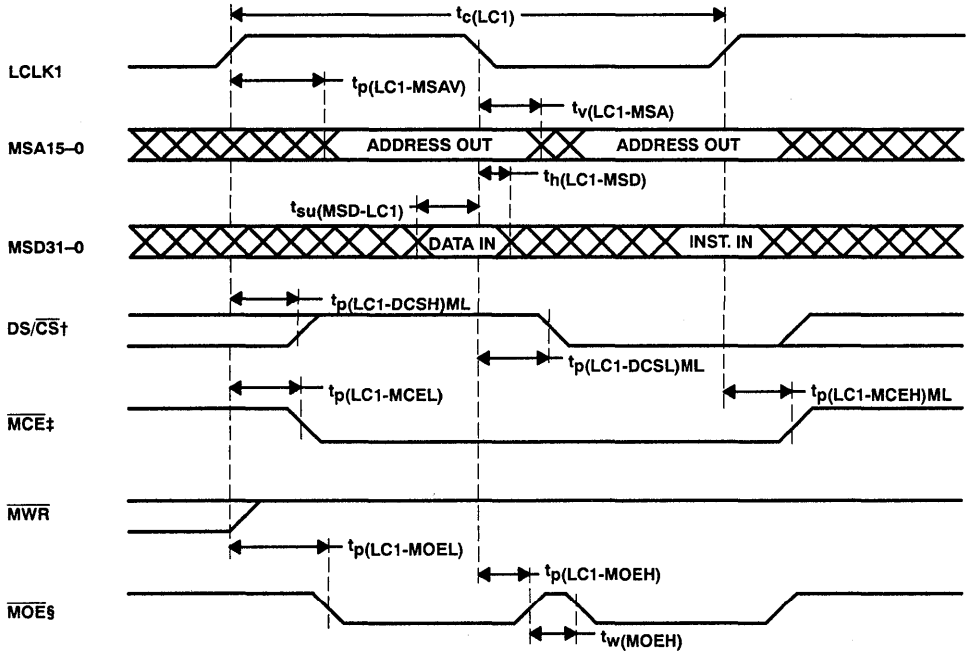
PARAMETER MEASUREMENT INFORMATION



† Q1, Q2, Q3, and Q4 represent the first, second, third, and fourth quarter clocks, respectively, of the LCLK1 clock period.

FIGURE 16. COPROCESSOR MODE, TMS34082A TO DRAM/VRAM MEMORY

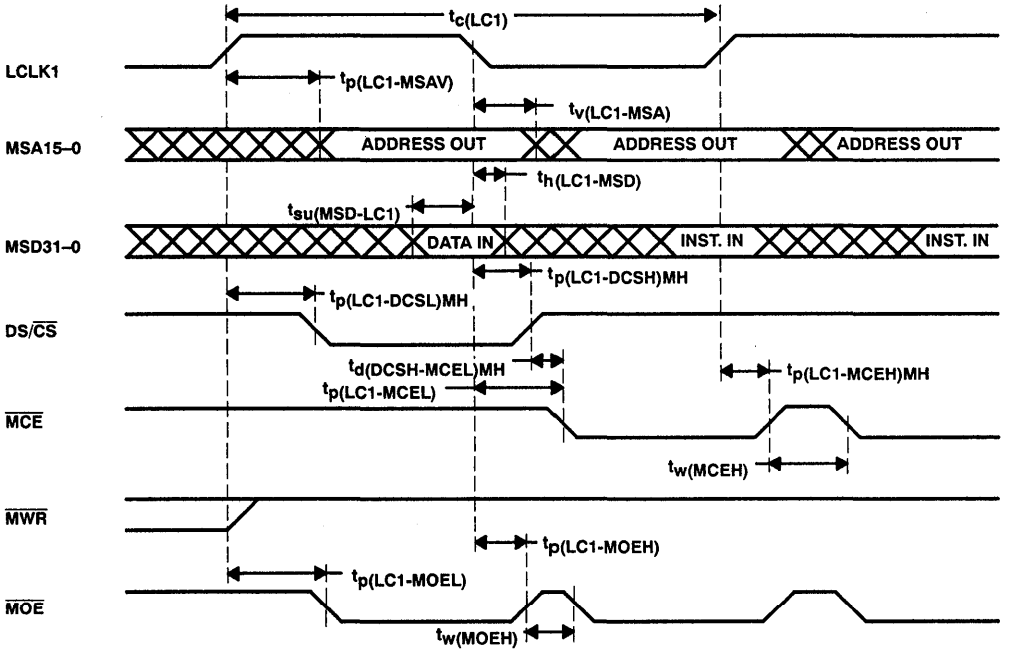
PARAMETER MEASUREMENT INFORMATION



† The setting of DS/$\overline{CS}$ determines whether the value on the MSD bus is an instruction or data.
‡ $\overline{MCE}$ dos not toggle at each clock edge.
§ $\overline{MOE}$ goes high at each clock edge.
NOTE: This example shows a data read followed by an instruction read.

FIGURE 17. COPROCESSOR MODE MSD BUS TIMING, MEMORY TO TMS34082A WITH MEMCFG LOW

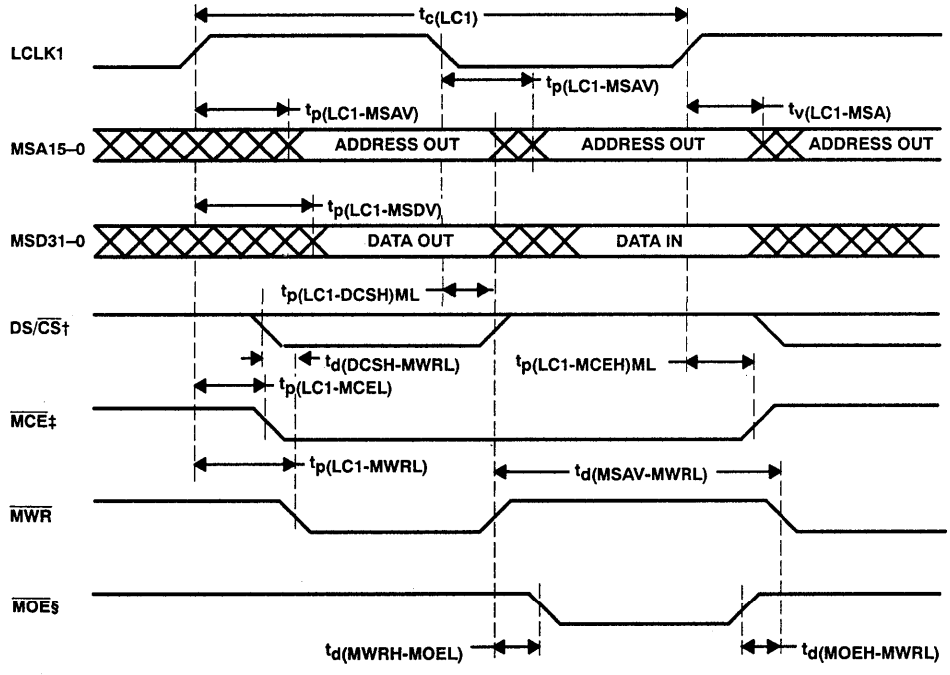
PARAMETER MEASUREMENT INFORMATION



NOTE: This example shows a data read followed by an instruction read followed by an instruction read. This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is invoked by setting the MEMCFG bit high in the configuration register. When MEMCFG is high, DS/$\overline{CS}$ and $\overline{MCE}$ rise after every clock edge. In this mode, DS/$\overline{CS}$ and $\overline{MCE}$ may not both be active (low) at the same time.

FIGURE 18. COPROCESSOR MODE MSD BUS TIMING, MEMORY TO TMS34082A WITH MEMCFG HIGH

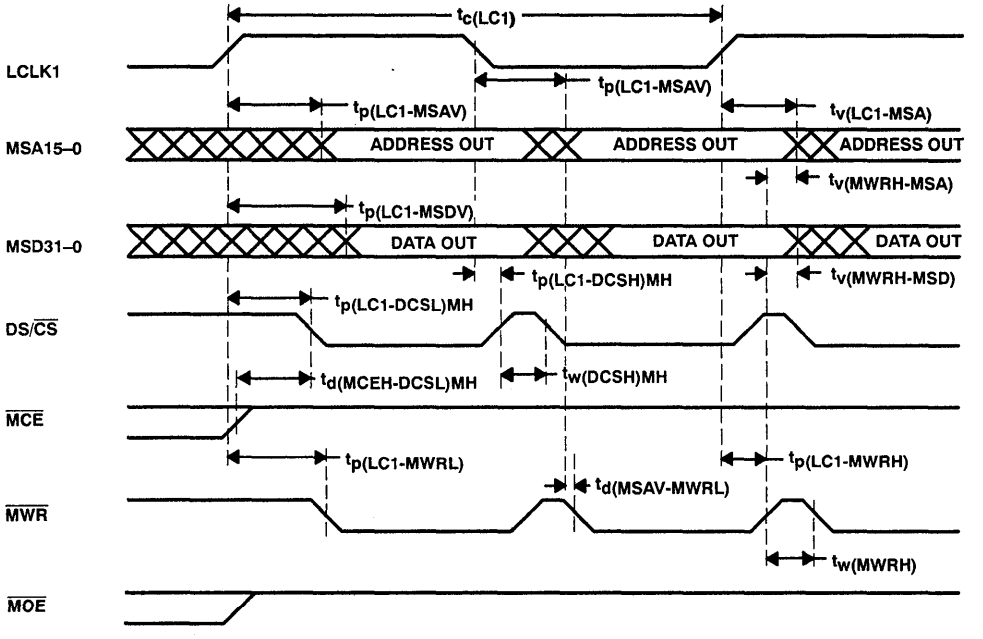
PARAMETER MEASUREMENT INFORMATION



† The setting of DS/$\overline{CS}$ determines whether the value on the MSD bus is an instruction or data.
‡ $\overline{MCE}$ does not toggle at each clock edge.
§ $\overline{MWR}$ goes high at each clock edge.
NOTE: This example shows a data write followed by a code read.

FIGURE 19. COPROCESSOR MODE MSD BUS TIMING, TMS34082A TO MEMORY WITH MEMCFG LOW

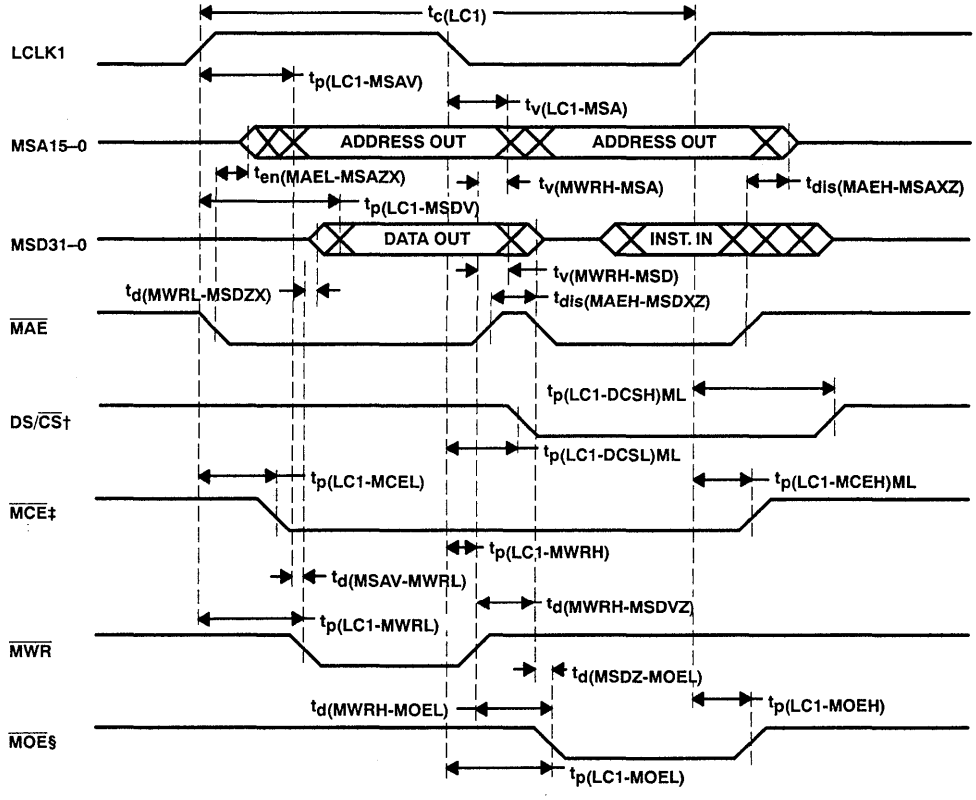
PARAMETER MEASUREMENT INFORMATION



NOTE: This example shows multiple data writes. Timing for multiple code writes would be similar. This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is invoked by setting the MEMCFG bit high in the configuration register. When MEMCFG is high, DS/$\overline{CS}$ and $\overline{MCE}$ rise after every clock edge. In this mode, DS/$\overline{CS}$ and $\overline{MCE}$ may not both be active (low) at the same time.

FIGURE 20. COPROCESSOR MODE MSD BUS TIMING, TMS34082A TO MEMORY WITH MEMCFG HIGH

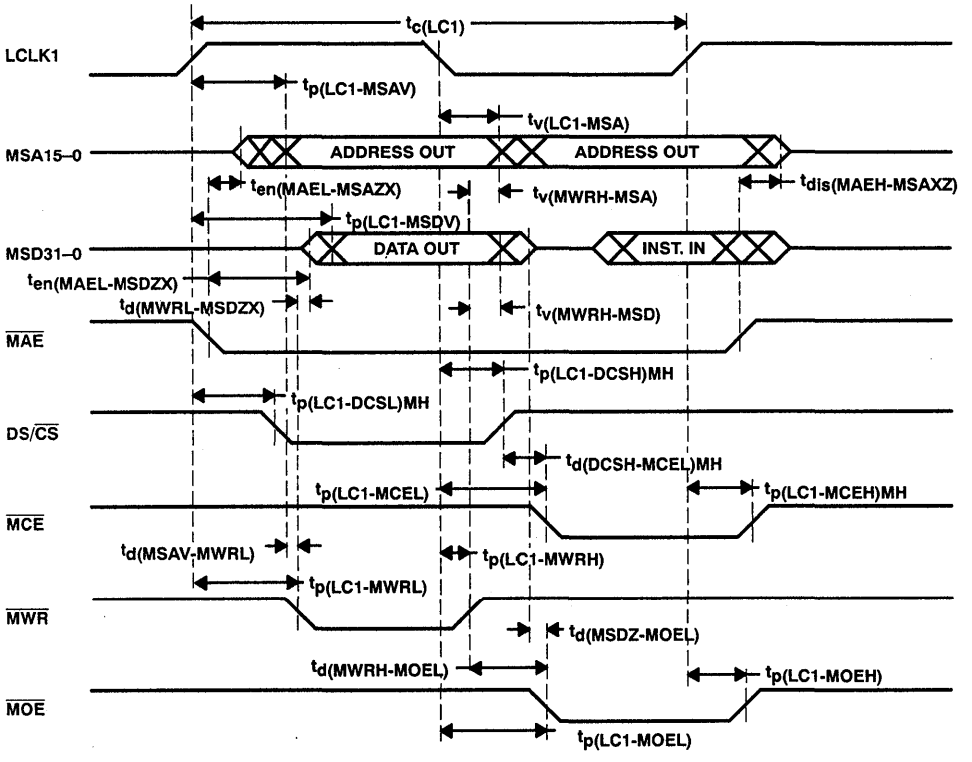
PARAMETER MEASUREMENT INFORMATION



† The setting of DS/$\overline{CS}$ determines whether the value on the MSD bus is an instruction or data.
‡ $\overline{MCE}$ does not toggle at each clock edge.
§ $\overline{MOE}$ goes high at each clock edge.
NOTE: This example shows a data write followed by an instruction read.

FIGURE 1. COPROCESSOR MODE, MSD ENABLE/DISABLE TIMING WITH MEMCFG LOW

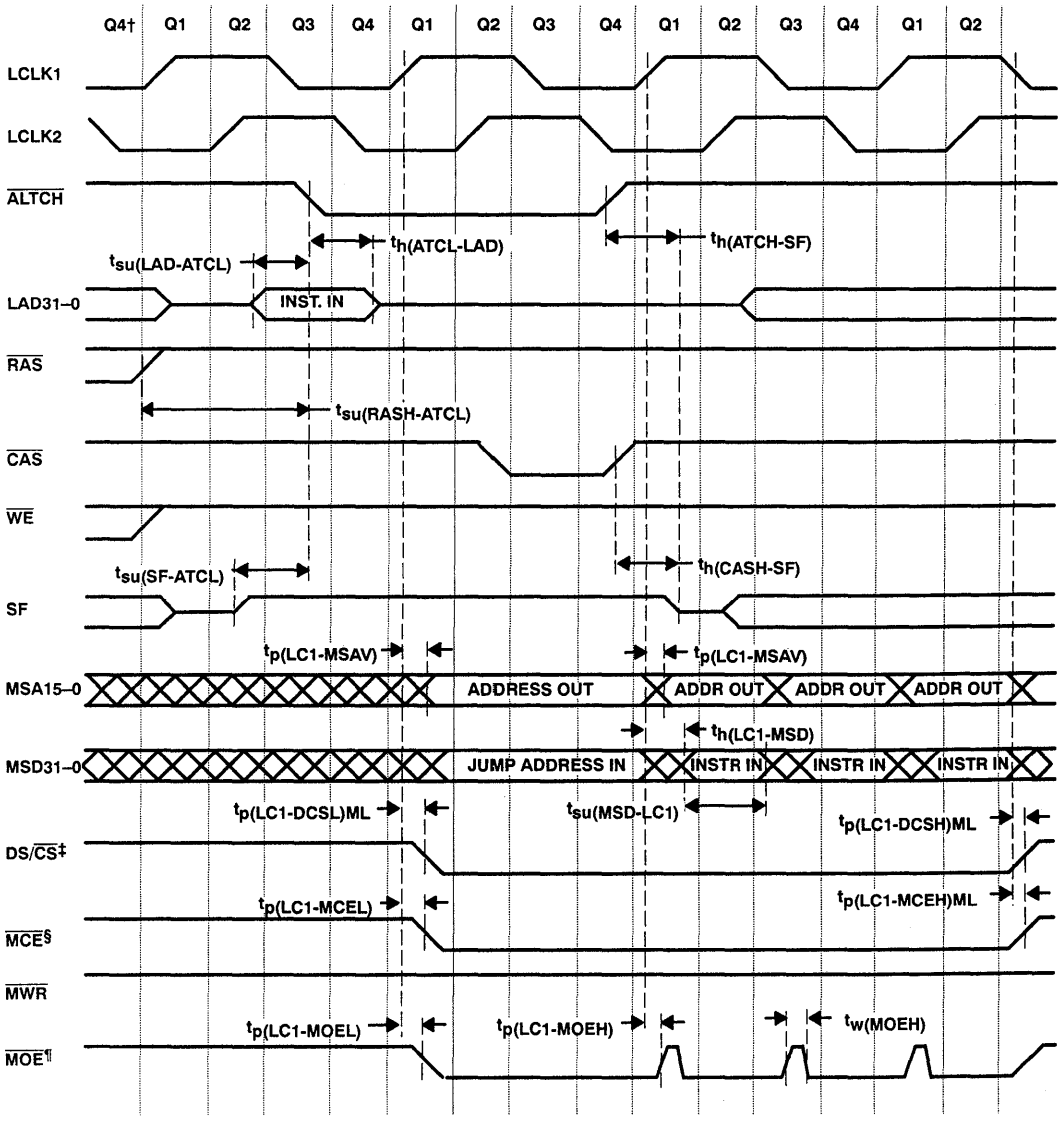
PARAMETER MEASUREMENT INFORMATION



NOTE: This example shows a data write followed by an instruction read. Timing for multiple code writes would be similar. This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is invoked by setting the MEMCFG bit high in the configuration register. When MEMCFG is high, DS/$\overline{CS}$ and $\overline{MCE}$ rise after every clock edge. In this mode, DS/$\overline{CS}$ and $\overline{MCE}$ may not both be active (low) at the same time.

FIGURE 2. COPROCESSOR MODE, MSD BUS ENABLE/DISABLE TIMING WITH MEMCFG HIGH

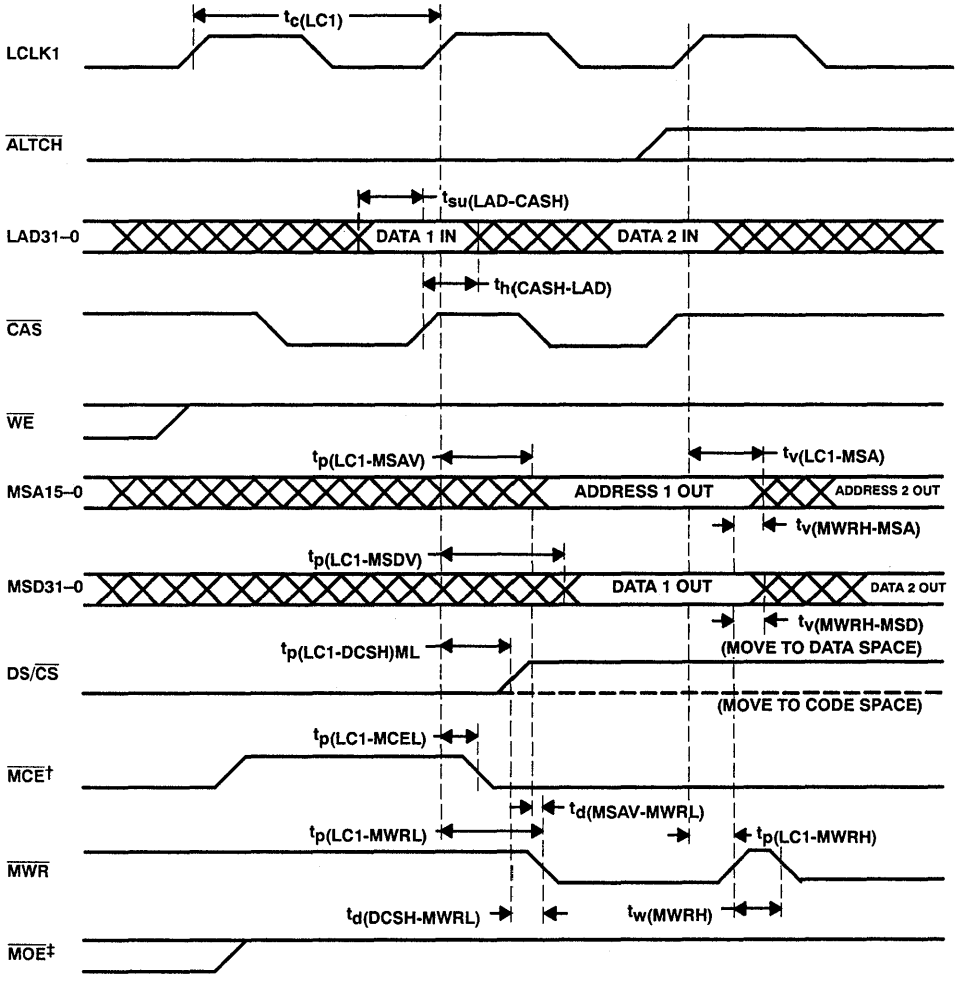
PARAMETER MEASUREMENT INFORMATION



† Q1, Q2, Q3, and Q4 represent the first, second, third, and fourth quarter clocks, respectively, of the LCLK1 clock period.
‡ The setting of DS/$\overline{CS}$ determines whether the value on the MSD bus in an instruction or data.
§ $\overline{MCE}$ does not toggle at each rising clock edge.
¶ $\overline{MOE}$ goes hiigh at each rising clock edge.

FIGURE 3. COPROCESSOR MODE, JUMP TO EXTERNAL MEMORY SUBROUTINE WITH MEMCFG LOW

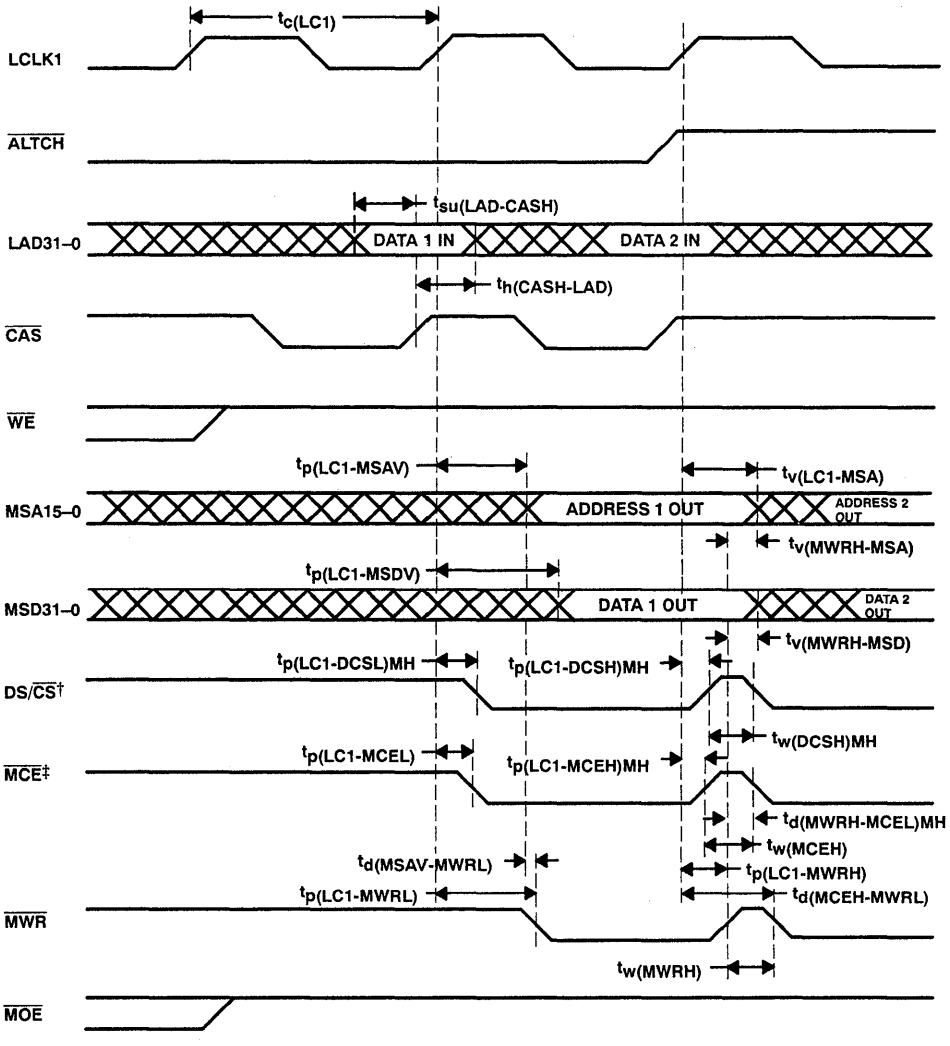
PARAMETER MEASUREMENT INFORMATION



† $\overline{\text{MCE}}$ does not toggle at each clock edge.
‡ $\overline{\text{MOE}}$ goes high at each clock edge.

FIGURE 4. COPROCESSOR MODE, LAD TO MSD BUS TRANSFER TIMING WITH MEMCFG LOW

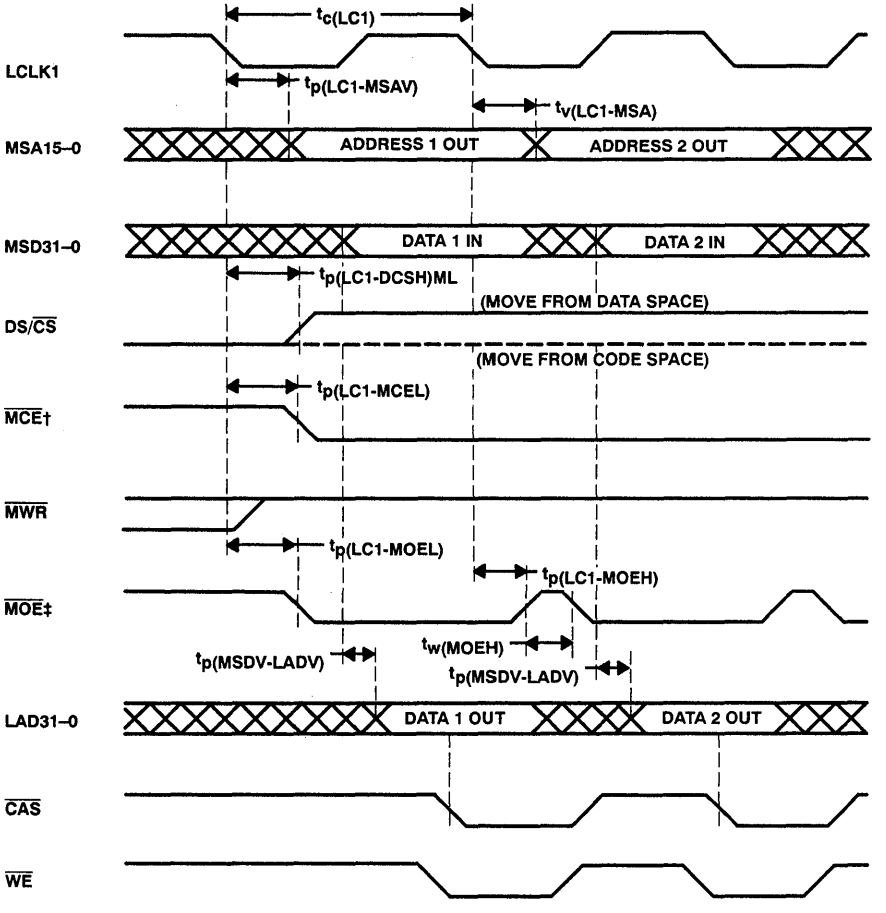
PARAMETER MEASUREMENT INFORMATION



† DS/$\overline{CS}$ valid for moves to data space; $\overline{MCE}$ valid for moves to code space. Only one of these would be valid for each move instruction.
‡ This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is invoked by setting the MEMCFG bit high in the configuration register.

FIGURE 5. COPROCESSOR MODE, LAD TO MSD BUS TRANSFER TIMING WITH MEMCFG HIGH

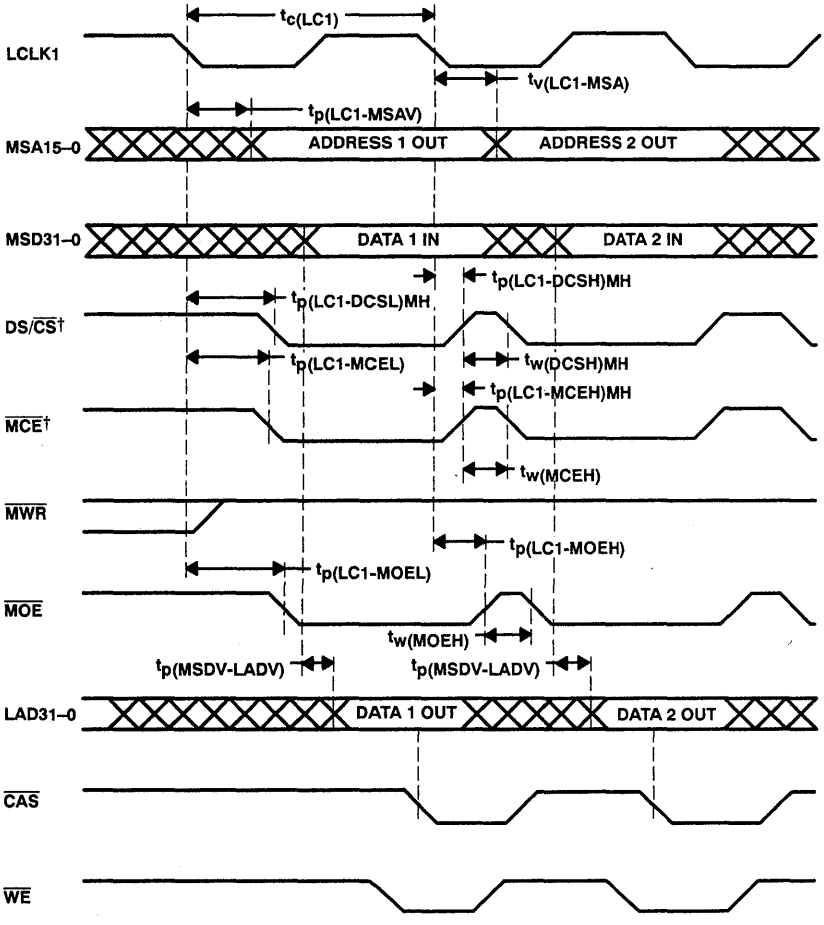
PARAMETER MEASUREMENT INFORMATION



† $\overline{MCE}$ does not toggle at each clock edge.
‡ $\overline{MOE}$ goes high at each clock edge.

FIGURE 6. COPROCESSOR MODE, MSD TO LAD BUS TRANSFER TIMING WITH MEMCFG LOW

PARAMETER MEASUREMENT INFORMATION



† DS/$\overline{CS}$ valid for moves to data space; $\overline{MCE}$ valid for moves to code space. Only one would be valid for each move instruction.

NOTE: This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is involved by setting the MEMCFG bit high in the configuration register.

FIGURE 7. COPROCESSOR MODE, MSD TO LAD BUS TRANSFER TIMING WITH MEMCFG HIGH

PARAMETER MEASUREMENT INFORMATION

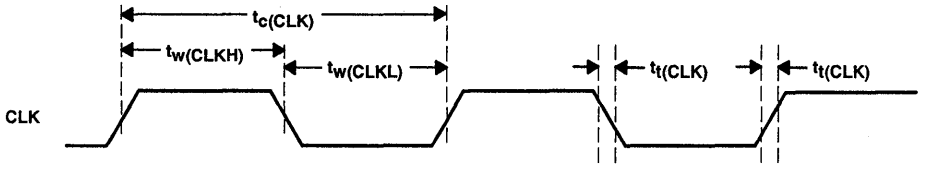
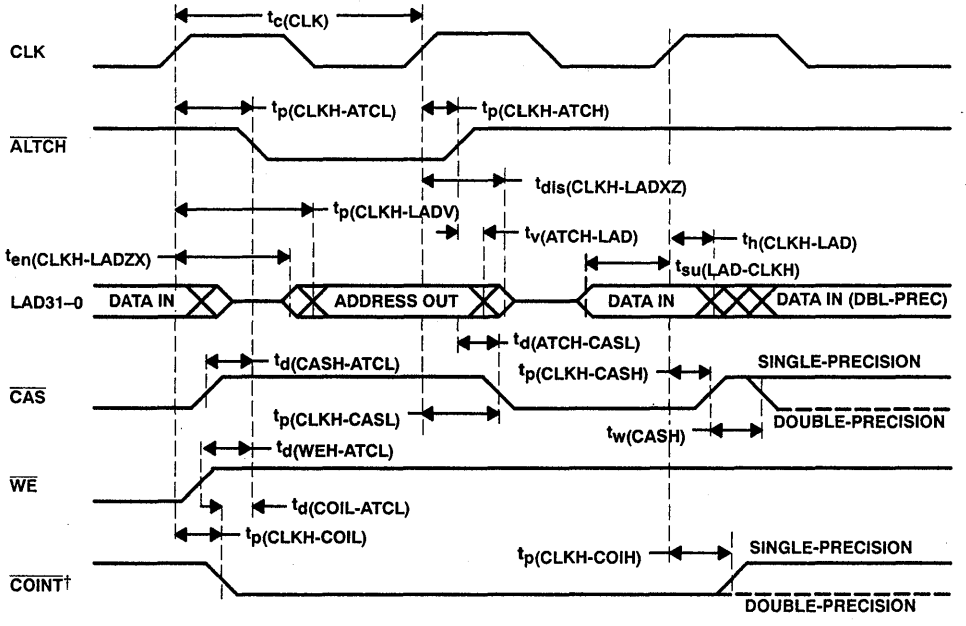


FIGURE 8. HOST-INDEPENDENT MODE, LAD BUS TIMING FOR MEMORY TO TMS34082A

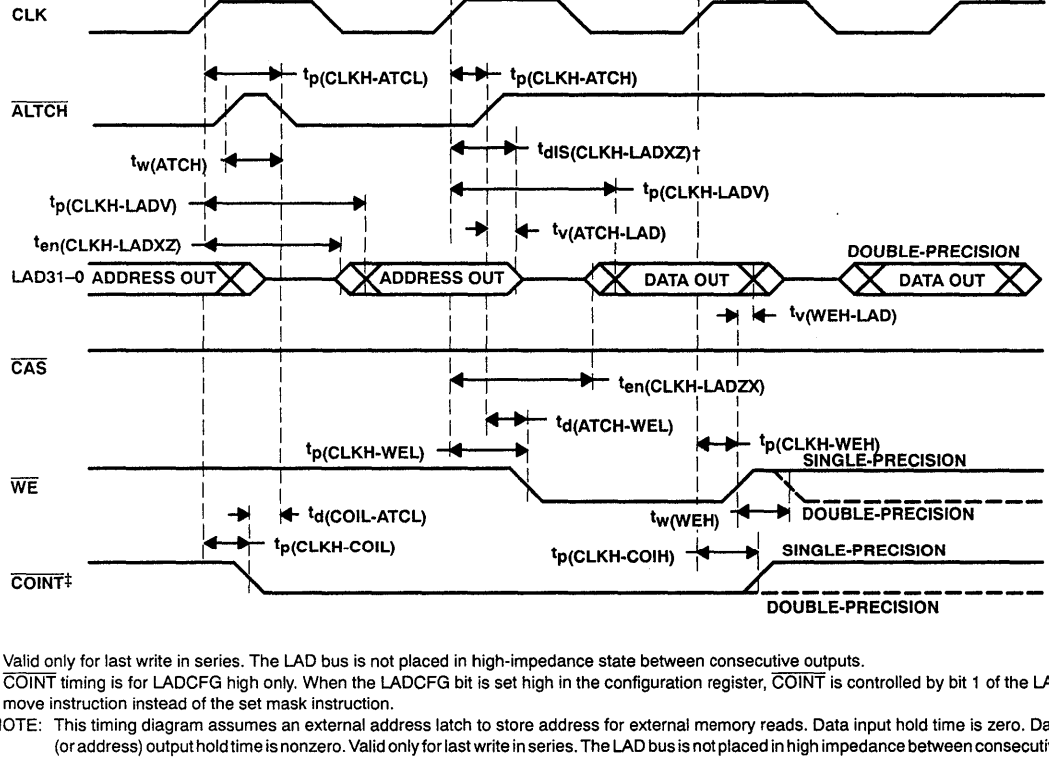


† $\overline{\text{COINT}}$ timing is for LADCFG high only. When the LADCFG bit is set high in the configuratin register, $\overline{\text{COINT}}$ is controlled by bit 1 of the LAD move instruction instead of the set mask instruction.

NOTE: This timing diagram assumes an external address latch to store address for external memory reads. Data input hold time on the latch is zero; data (or address) output hold time is nonzero.

FIGURE 9. HOST-INDEPENDENT MODE, LAD BUS TIMING FOR MEMORY TO TMS34082A

PARAMETER MEASUREMENT INFORMATION



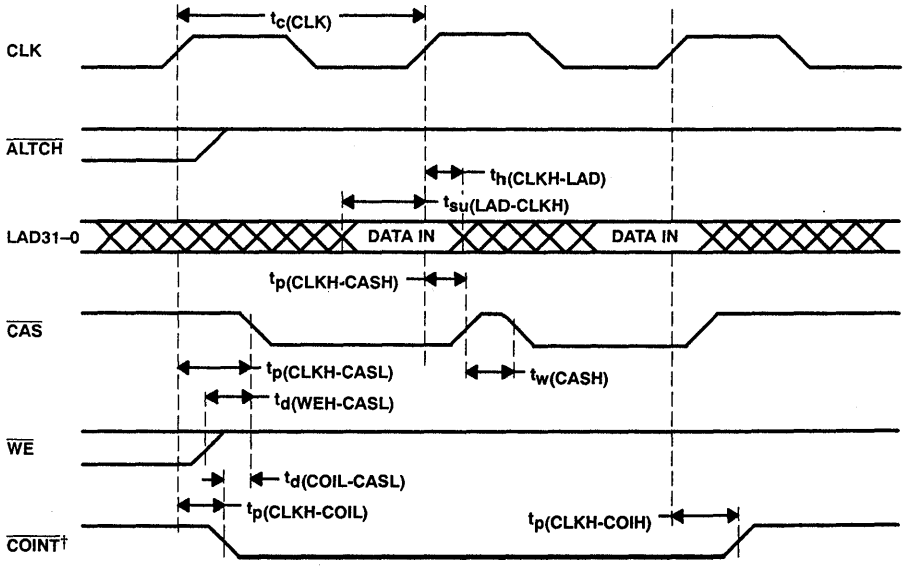
† Valid only for last write in series. The LAD bus is not placed in high-impedance state between consecutive outputs.

‡ $\overline{\text{COINT}}$ timing is for LADCFG high only. When the LADCFG bit is set high in the configuration register, $\overline{\text{COINT}}$ is controlled by bit 1 of the LAD move instruction instead of the set mask instruction.

NOTE: This timing diagram assumes an external address latch to store address for external memory reads. Data input hold time is zero. Data (or address) output hold time is nonzero. Valid only for last write in series. The LAD bus is not placed in high impedance between consecutive outputs.

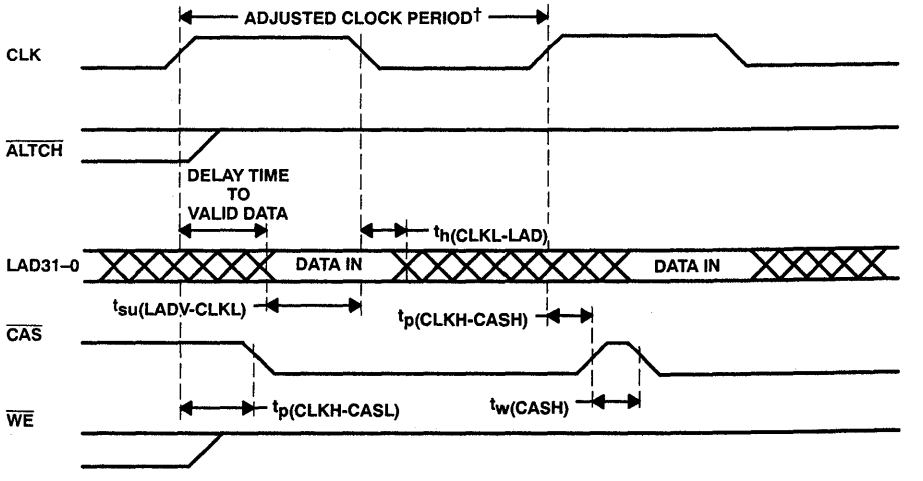
FIGURE 10. HOST-INDEPENDENT MODE, LAD BUS TIMING FOR TMS34082A TO MEMORY

PARAMETER MEASUREMENT INFORMATION



† $\overline{\text{COINT}}$ timing is for LADCFG high only. When the LADCFG bit is set high in the configuration register, $\overline{\text{COINT}}$ is controlled by bit 1 of the LAD move instruction instead of the set mask instruction.

FIGURE 11. HOST-INDEPENDENT MODE, LAD BUS TIMING INPUT TO TMS34082A

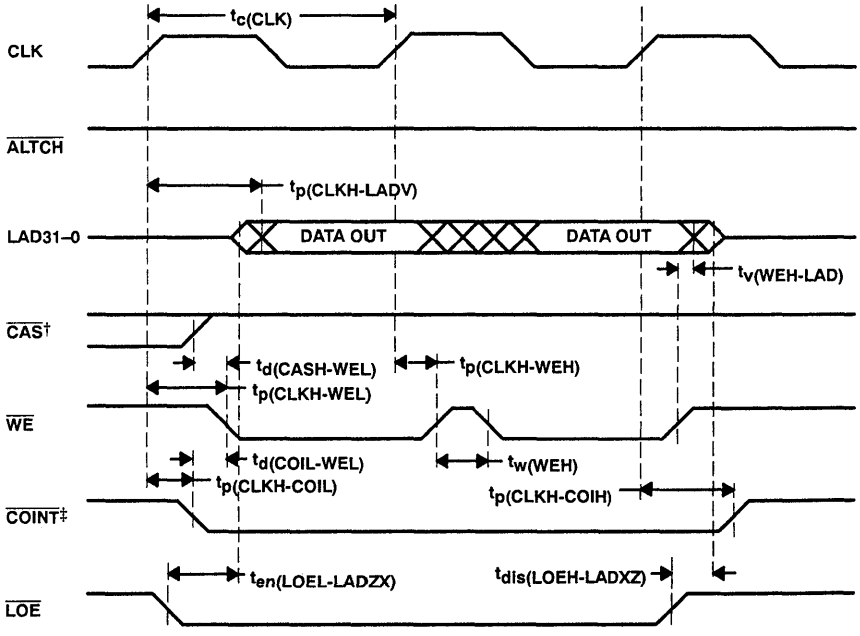


† This mode permits data input which does not meet the minimum setup before CLK high. For immediate data input, CLK must be high for more than 20 ns. This input mode cannot be used to input data for divides and square roots.

Adjusted clock period = Normal clock period + Data delay + 5 ns

FIGURE 12. HOST-INDEPENDENT MODE, LAD BUS TIMING INPUT OF IMMEDIATE DATA TO TMS34082A

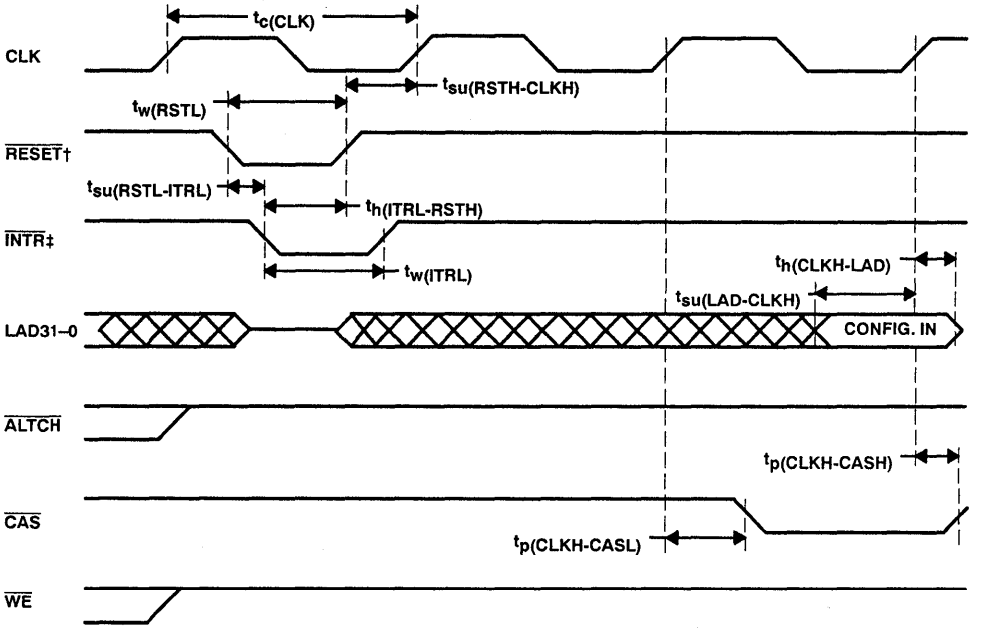
PARAMETER MEASUREMENT INFORMATION



† When the LADCFG bit is high, $\overline{LOE}$ high places $\overline{CAS}$ and $\overline{WE}$ (as well as the LAD bus) in high impedance.
‡ Valid only for LADCFG high. When the LADCFG bit is high in the configuration register, $\overline{COINT}$ is controlled by bit 1 of the LAD move instruction instead of the set mask instruction.
NOTE: If the instruction writes the result of an FPU operation to a register and outputs the result to the LAD bus, in the same cycle, the minimum clock period must be extended.

FIGURE 13. HOST-INDEPENDENT MODE, LAD BUS TIMING OUTPUT FROM TMS34082A

PARAMETER MEASUREMENT INFORMATION



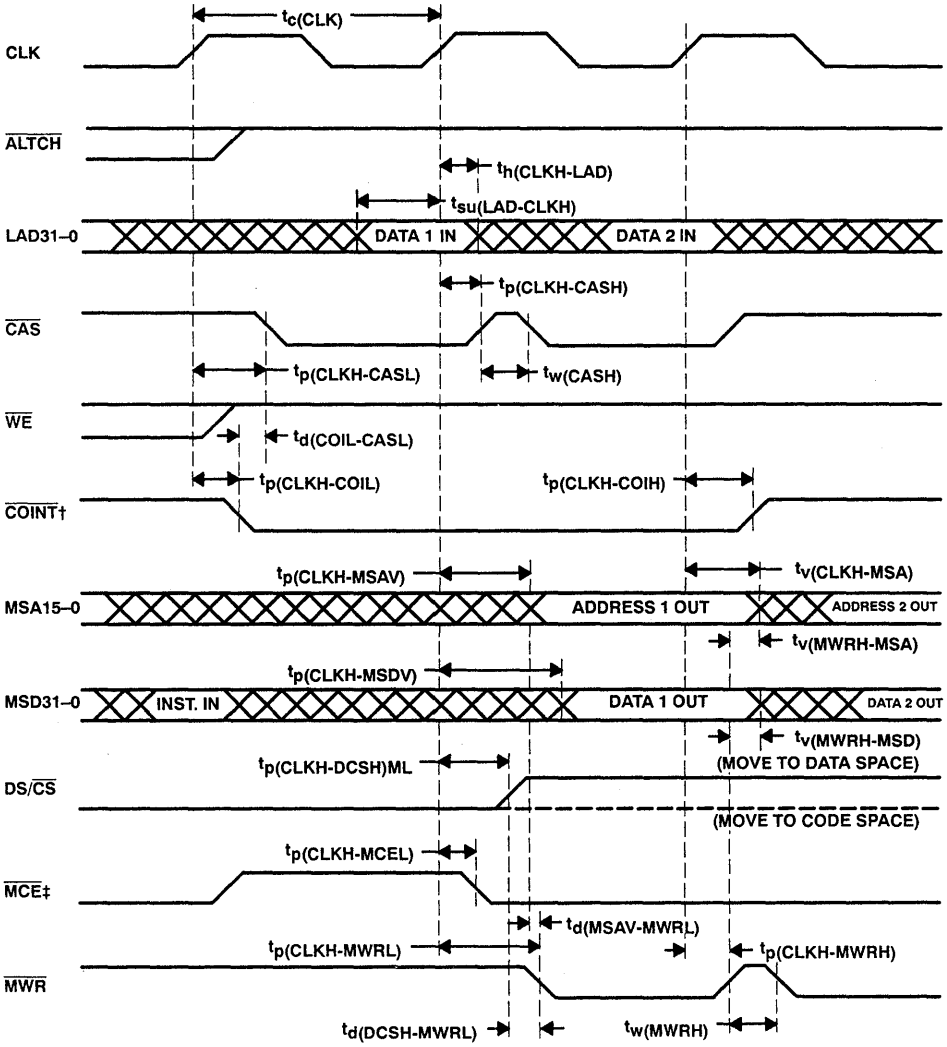
† RESET is level sensitive. When RESET is set low, both LAD and MSD buses are placed in high-impedance state. When RESET is released, the sequencer forces a jump to address 0. If INTR goes low while RESET is low, the loader moves 64 words through to the external memory on MSD. Timing for the LAD to MSD move is shown in a later diagram, with the exception that the first word on LAD loads the configuration register and does not pass to the MSD bus.

‡ INTR may be low one or more cycles after RESET goes low. RESET is held low, and then INTR is taken low. The bootstrap loader starts when RESET is set high, which may involve a delay of one or more cycles after INTR goes low.

NOTE: When the bootstrap loader is invoked, the first data word input on the LAD bus should be the configuration register settings, which will be written into the configuration register. This allows the user to select the MEMCFG setting, for reading or writing memory on the MSD port, as well as the LADCFG setting for the LAD bus interface.

FIGURE 14. HOST-INDEPENDENT MODE LAD BUS TIMING, BOOTSTRAP LOADER OPERATION

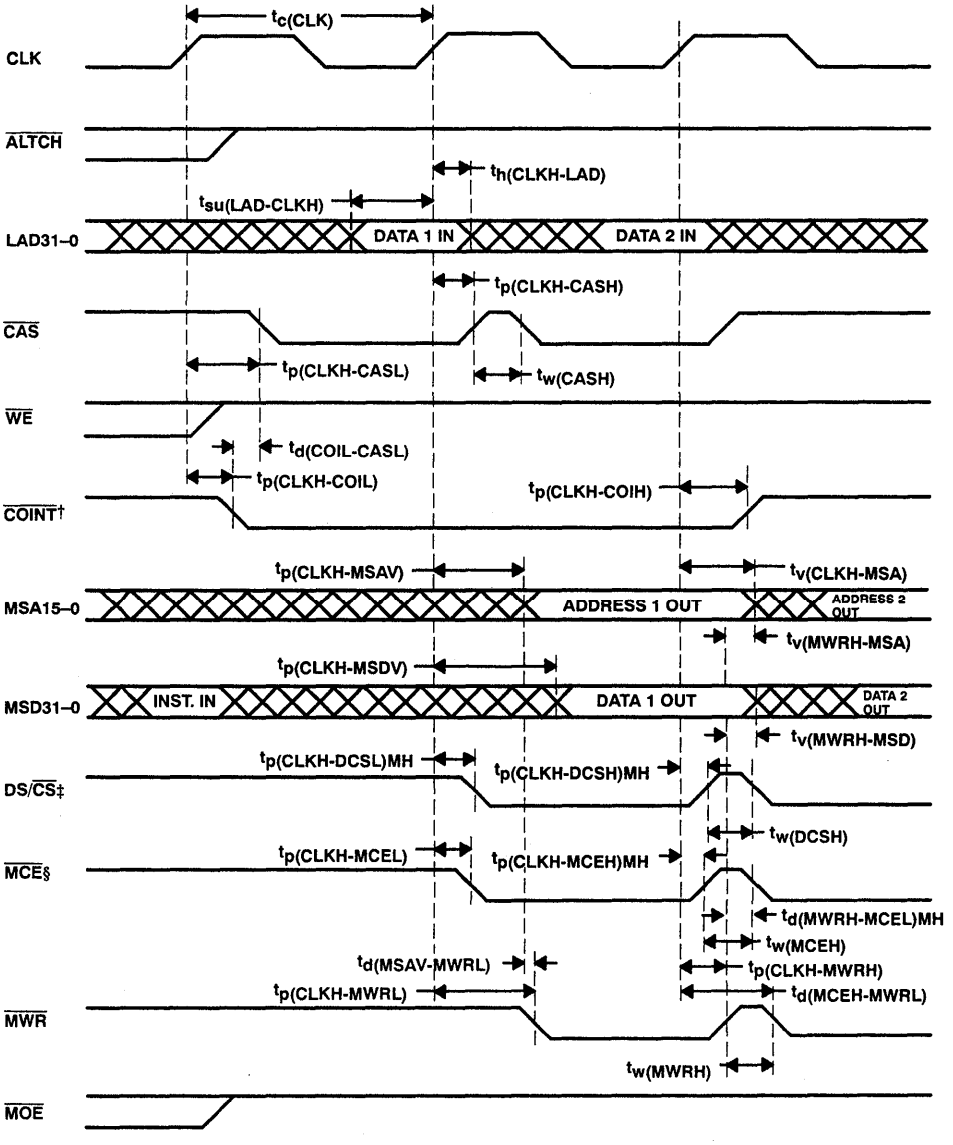
PARAMETER MEASUREMENT INFORMATION



† $\overline{\text{COINT}}$ timing is for LADCFG high only. When the LADCFG bit is set high in the configuration register, $\overline{\text{COINT}}$ is controlled by bit 1 of the LAD move instruction instead of the set mask instruction.
‡ $\overline{\text{MCE}}$ does not toggle at each rising clock edge.
§ $\overline{\text{MOE}}$ goes high at each rising clock edge.

FIGURE 15. HOST-INDEPENDENT MODE, LAD TO MSD BUS TRANSFER TIMING WITH MEMCFG LOW

PARAMETER MEASUREMENT INFORMATION



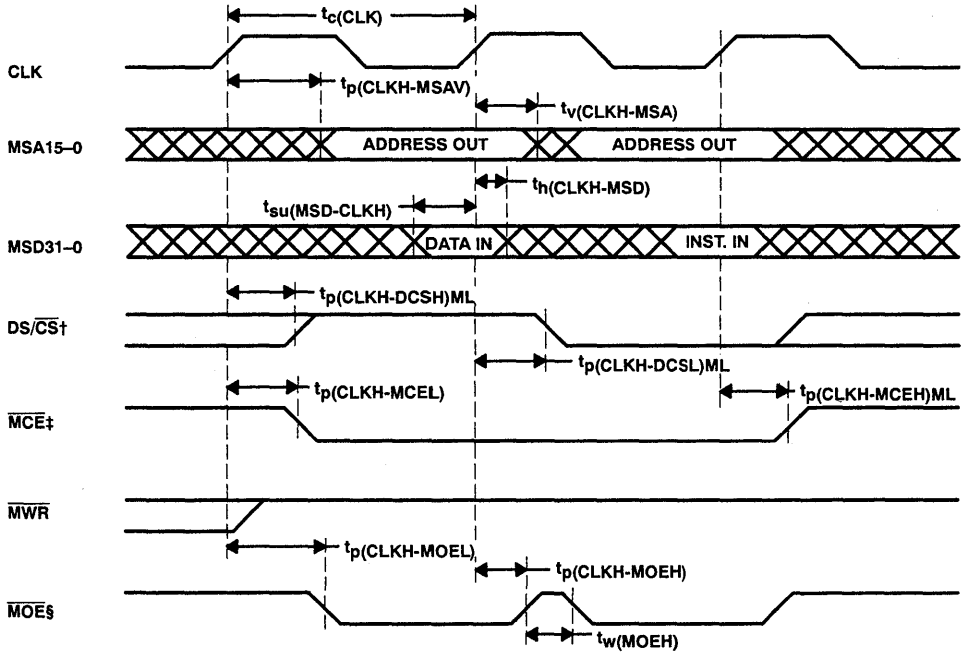
† $\overline{\text{COINT}}$ timing is for LADCFG high only. When the LADCFG bit is set high in the configuration register, $\overline{\text{COINT}}$ is controlled by bit 1 of the LAD move instruction instead of the set mask instruction.

‡ DS/$\overline{\text{CS}}$ valid for moves to data space; $\overline{\text{MCE}}$ valid for moves to code space. Only one of these would be valid for each move instruction.

§ This option for using DS/$\overline{\text{CS}}$ as data space chip enable and $\overline{\text{MCE}}$ as code space chip enable is invoked by setting the MEMCFG bit high in the configuration register.

FIGURE 16. HOST-INDEPENDENT MODE, LAD TO MSD BUS TRANSFER TIMING WITH MEMCFG HIGH

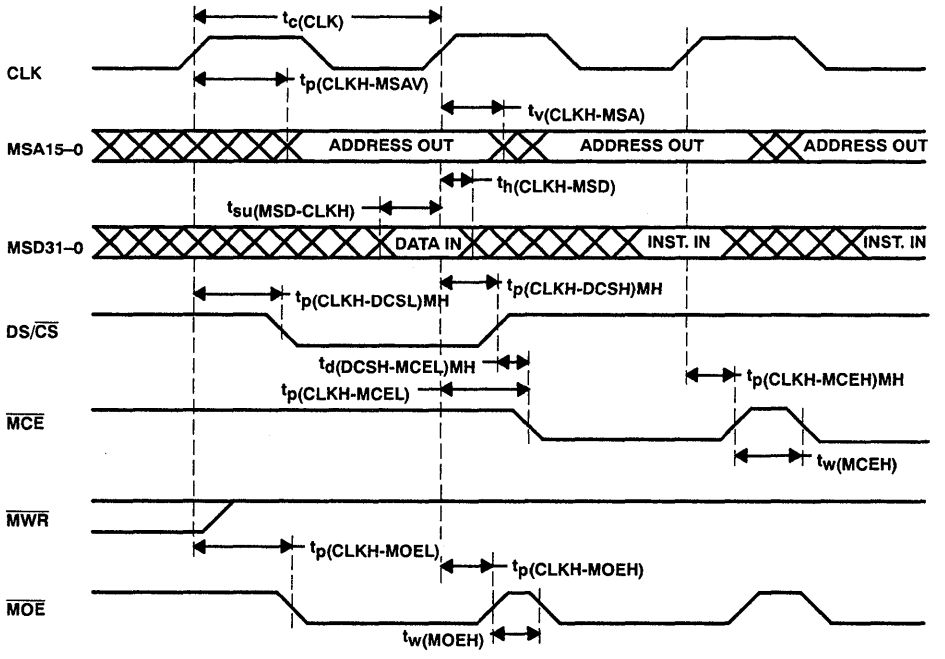
PARAMETER MEASUREMENT INFORMATION



† The setting of DS/$\overline{CS}$ determines whether the value on the MSD bus is an instruction or data.
‡ $\overline{MCE}$ does not toggle at each rising clock edge.
§ $\overline{MOE}$ goes high at each rising clock edge.
NOTE; This example shows a data read followed by an instruction read.

FIGURE 17. HOST-INDEPENDENT MODE MSD BUS TIMING, MEMORY TO TMS34082A WITH MEMCFG LOW

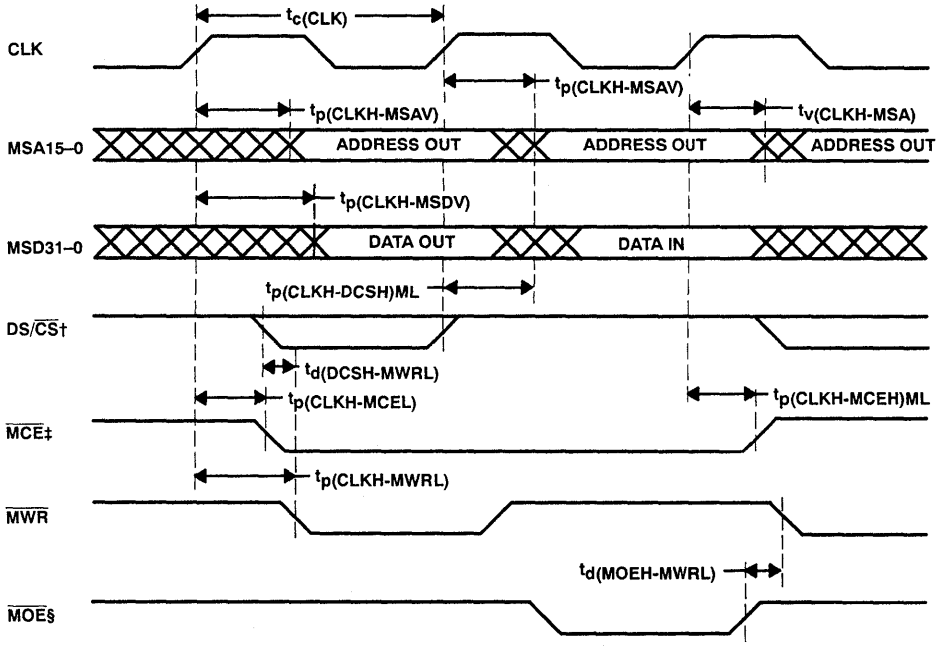
PARAMETER MEASUREMENT INFORMATION



NOTE: This example shows a data read followed by an instruction read followed by an instruction read. This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is invoked by setting the MEMCFG bit high in the configuration register. When MEMCFG is high, DS/$\overline{CS}$ and $\overline{MCE}$ rise after every rising clock edge. In this mode, DS/$\overline{CS}$ and $\overline{MCE}$ may not both be active (low) at the same time.

FIGURE 18. HOST-INDEPENDENT MODE MSD BUS TIMING, MEMORY TO TMS34082A WITH MEMCFG HIGH

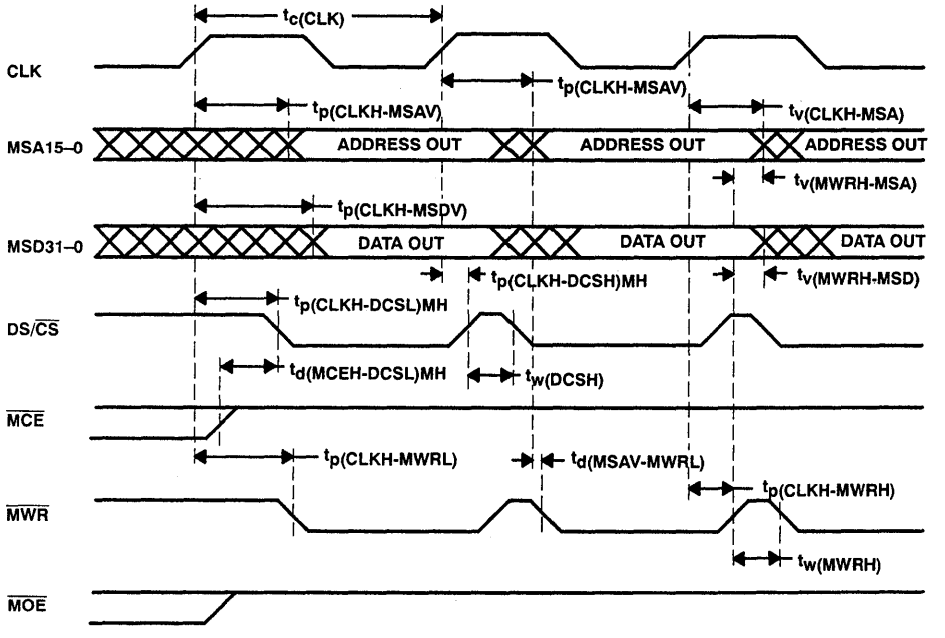
PARAMETER MEASUREMENT INFORMATION



† The setting of DS/$\overline{CS}$ determines whether the value on the MSD bus is an instruction or data.
‡ $\overline{MCE}$ does not toggle at each rising clock edge.
§ $\overline{MWR}$ goes high at each rising clock edge.
NOTE: This example shows a data write followed by a code read.

FIGURE 19. HOST-INDEPENDENT MODE MSD BUS TIMING, TMS34082A TO MEMORY WITH MEMCFG LOW

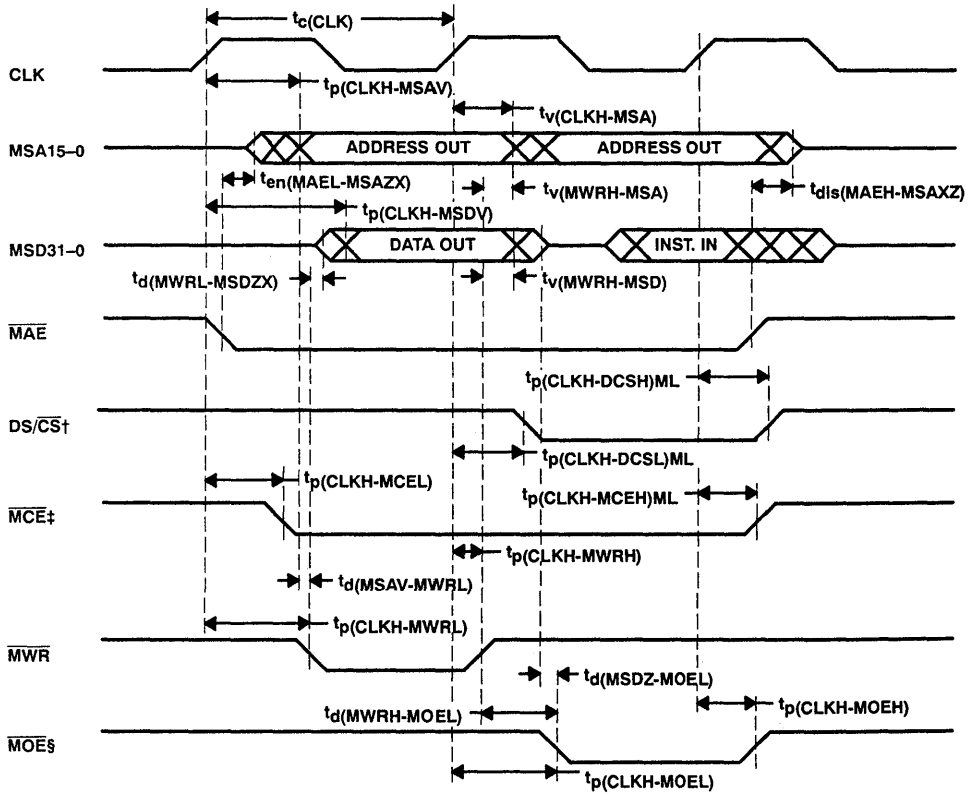
PARAMETER MEASUREMENT INFORMATION



NOTE: This example shows multiple data writes. Timing for multiple code writes would be similar. This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is invoked by setting the MEMCFG bit high in the configuration register. When MEMCFG is high, DS/$\overline{CS}$ and $\overline{MCE}$ rise after every rising clock edge. In this mode, DS/$\overline{CS}$ and $\overline{MCE}$ may not both be active (low) at the same time.

FIGURE 20. HOST-INDEPENDENT MODE MSD BUS TIMING, TMS34082A TO MEMORY WITH MEMCFG HIGH

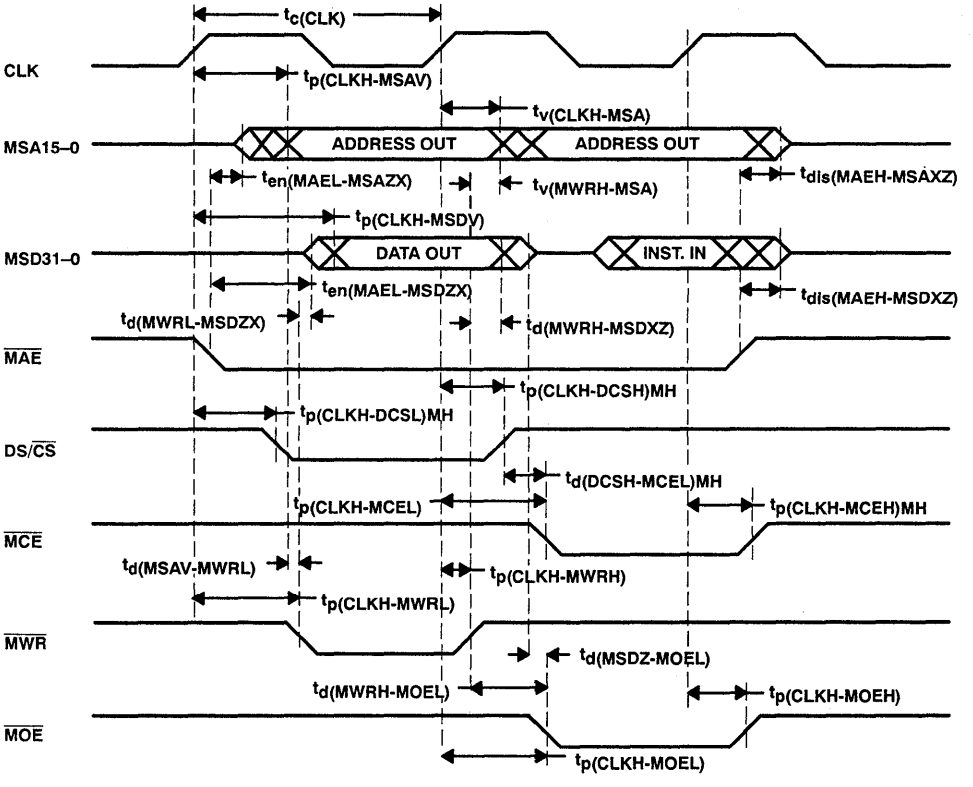
PARAMETER MEASUREMENT INFORMATION



† The setting of DS/CS determines whether the value on the MSD bus is an instruction or data.
‡ MCE does not toggle at each rising clock edge.
§ MOE goes high at each rising clock edge.
NOTE: This example shows a data write followed by an instruction read.

FIGURE 21. HOST-INDEPENDENT MODE, MSD ENABLE/DISABLE TIMING WITH MEMCFG LOW

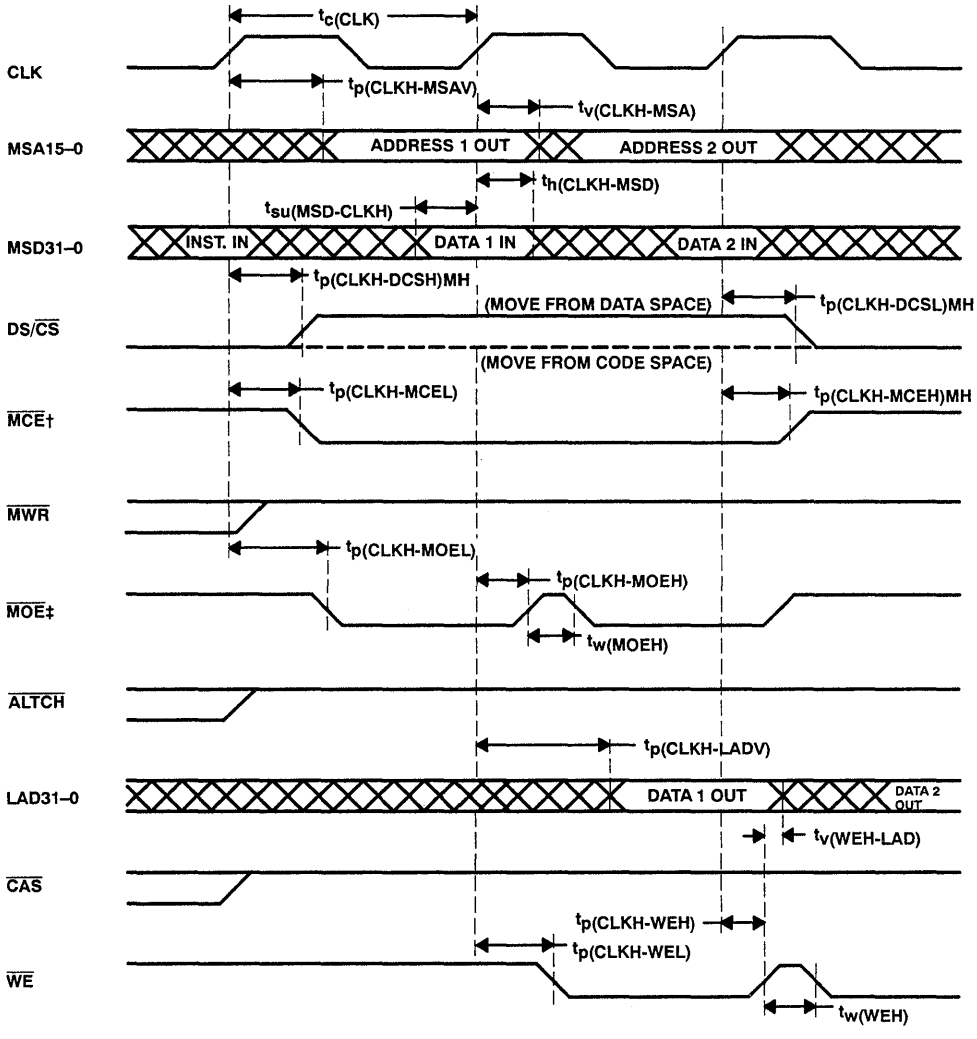
PARAMETER MEASUREMENT INFORMATION



NOTE: This example shows a data write followed by an instruction read. Timing for multiple code writes would be similar. This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is invoked by setting the MEMCFG bit high in the configuration register. When MEMCFG is high, DS/$\overline{CS}$ and $\overline{MCE}$ rise after every rising clock edge. In this mode, DS/$\overline{CS}$ and $\overline{MCE}$ may not both be low at the same time.

FIGURE 22. HOST-INDEPENDENT MODE, MSD BUS ENABLE/DISABLE TIMING WITH MEMCFG HIGH

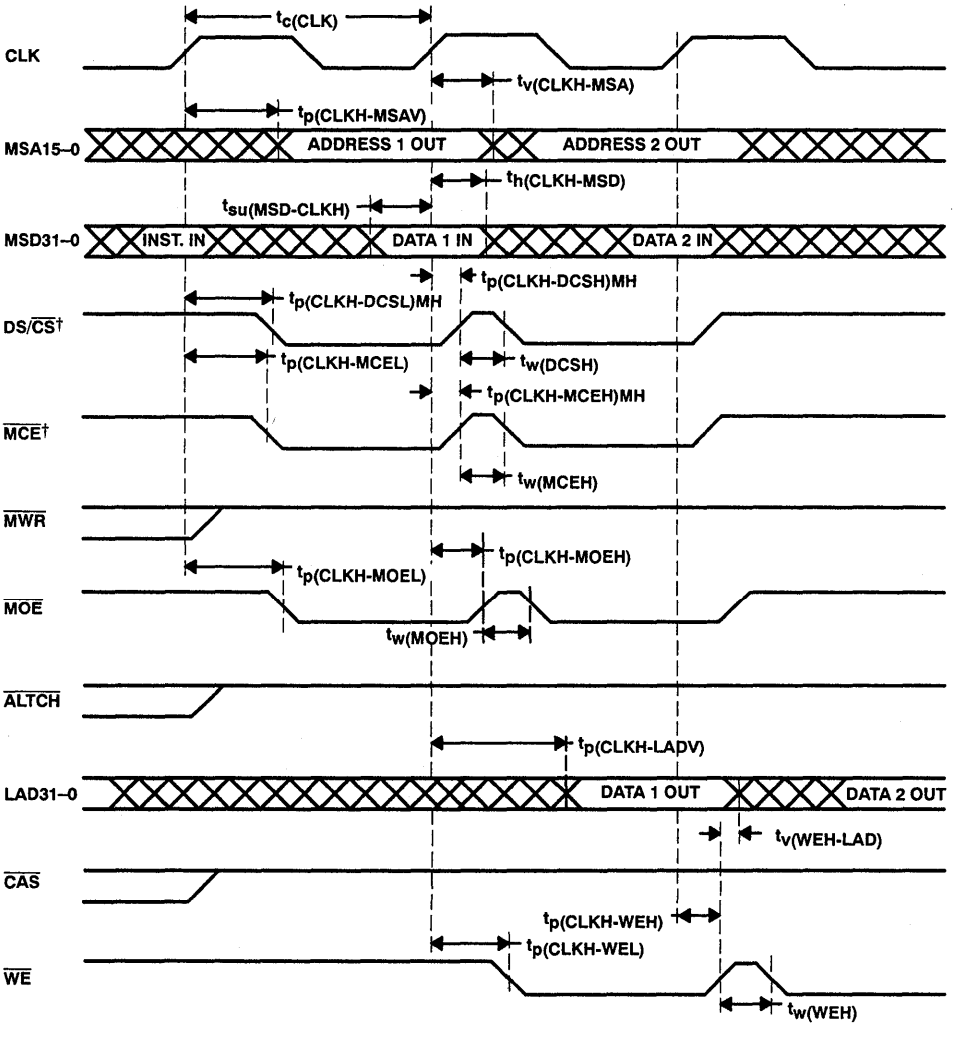
PARAMETER MEASUREMENT INFORMATION



† $\overline{MCE}$ does not toggle at each rising clock edge.
‡ $\overline{MOE}$ goes high at each rising clock edge.

FIGURE 23. HOST-INDEPENDENT MODE, MSD TO LAD BUS TRANSFER TIMING WITH MEMCFG HIGH

PARAMETER MEASUREMENT INFORMATION

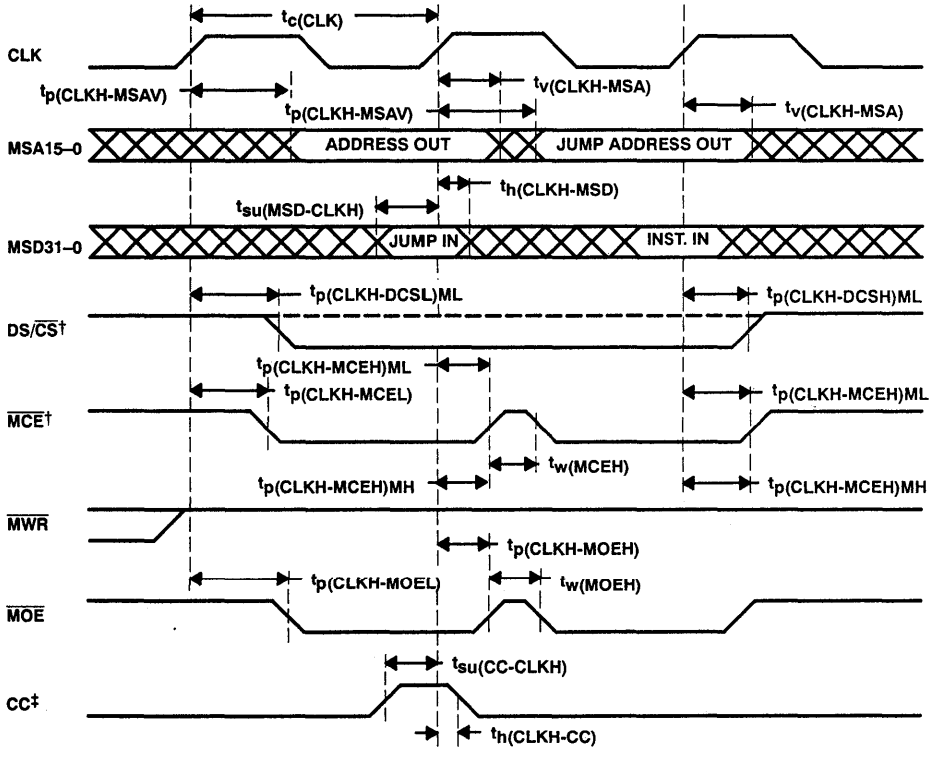


† DS/$\overline{CS}$ valid for moves to data space; $\overline{MCE}$ valid for moves to code space. Only one would be valid for each move instruction.

NOTE: This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is involved by setting the MEMCFG bit high in the configuration register.

FIGURE 24. HOST-INDEPENDENT MODE, MSD TO LAD BUS TRANSFER TIMING WITH MEMCF HIGH

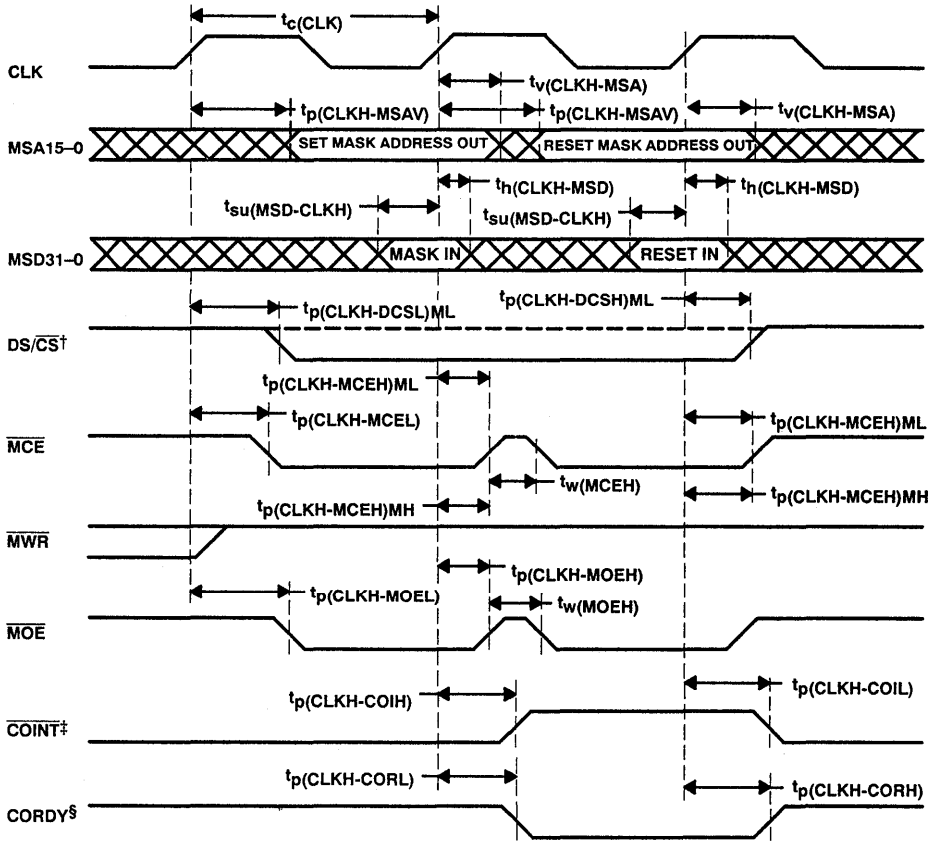
PARAMETER MEASUREMENT INFORMATION



† Dotted line shows DS/$\overline{CS}$ for MEMCFG high.
‡ The CC input is registered on each rising edge of the clock, so the CC bit can be latched one cycle and tested during the next cycle.

FIGURE 25. HOST-INDEPENDENT MODE, MSD BUS TIMING TEST CONDITION (CC) AND BRANCH

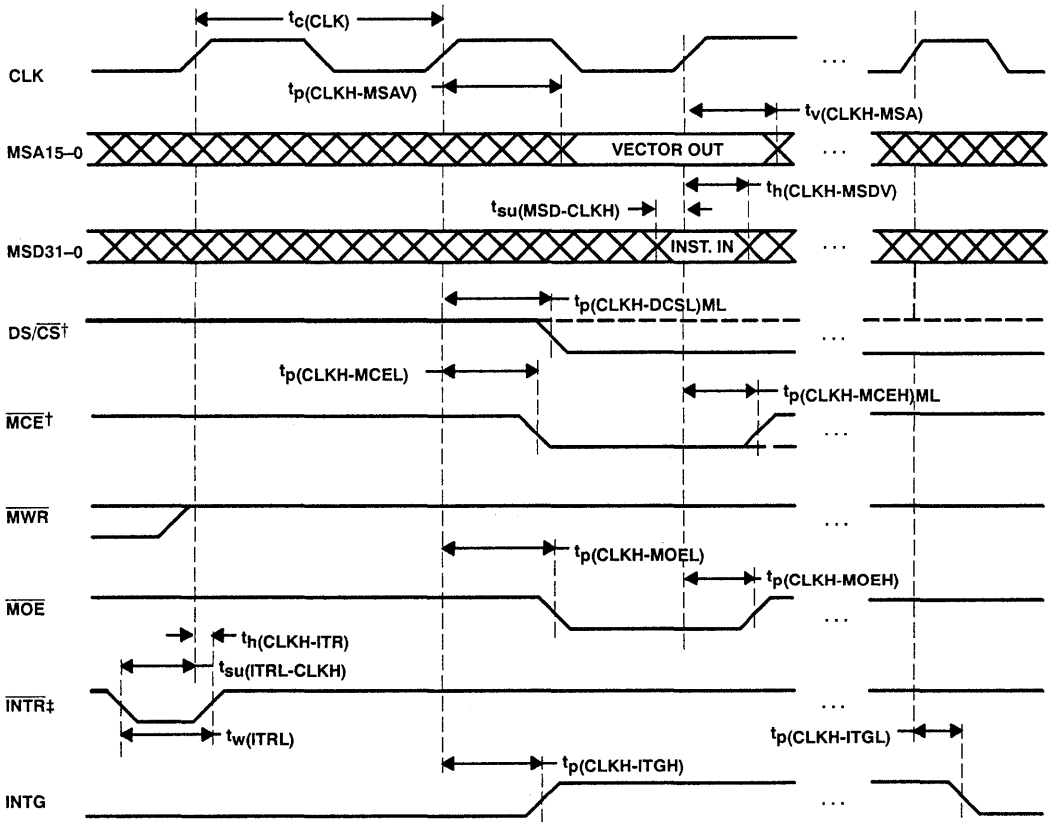
PARAMETER MEASUREMENT INFORMATION



† Dotted line shows DS/$\overline{CS}$ for MEMCFG high.
‡ Valid for MEMCFG low only. When MEMCFG low, $\overline{COINT}$ is set high by the set mask instruction, and it remains high until reset with another set mask instruction.
§ The CORDY output is set low by the set mask instruction, and it remains low until reset with another set mask instruction.

FIGURE 26. HOST-INDEPENDENT MODE MSD BUS TIMING, SET/RESET $\overline{COINT}$ AND CORDY

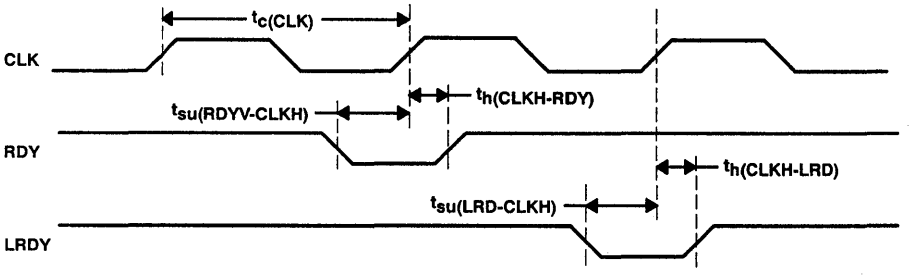
PARAMETER MEASUREMENT INFORMATION



† Dotted lines show DS/$\overline{CS}$ and $\overline{MCE}$ for MEMCFG high.
‡ $\overline{INTR}$ is negative-edged triggered.
NOTE: Interrupts are not granted during multi-cycle instructions. This example shows two interrupt requests. The first is granted immediately; the second, after the first is finished. INTG remains high after an interrupt is granted until interrupts are reenabled or a return from interrupt instruction is executed.

FIGURE 27. HOST-INDEPENDENT MODE, MSD BUS TIMING EXTERNAL INTERRUPT TO TMS34082A

PARAMETER MEASUREMENT INFORMATION



NOTE: When either RDY or LRDY is set low and the setup time before CLK high is observed, the device is stalled for one or more clock cycles, until RDY or LRDY is set high again. During a wait state, internal states and status are preserved and output signals do not change. LRDY can be used in this manner only in the host-independent mode.

FIGURE 28. HOST-INDEPENDENT MODE, MSD BUS TIMING WAIT STATE TIMING

PROGRAMMING INFORMATION

programming the TMS34082A

The TMS34082A is supported by a software development tool kit, including a C compiler and an assembler. Program development using the tools is described in the TMS34082A tool kit documentation. Information on internal instructions and listing of the external instructions are provided in the following sections.

In both the coprocessor and host-independent modes, the TMS34082A instruction word is 32 bits long. The number, length, and arrangement of fields in the 32-bit word depends on the operating mode and operation selected. Internal microcode to the TMS34082A is not restricted to the same 32-bit instruction formats so certain internal programs may execute faster than the same operations written with external code can achieve.

In the coprocessor mode, the TMS34082A can execute instructions both from the TMS34020 and from the program memory on the MSD bus (MSD31-0). In the host-independent mode the TMS34082A is controlled from code input on the MSD bus. Internal instructions may be executed in the host-independent mode by performing a jump to the internal address.

PROGRAMMING INFORMATION

internal instructions

The TMS34082A FPU performs a wide range of internal arithmetic and logical operations, as well as complex operations (flagged '†'), summarized below. Complex instructions are multi-cycle routines stored in the internal program ROM.

One-Operand Operations:

Absolute Value	1s Complement
Square Root	2s Complement
Reciprocal†	

Conversions:

Integer to Single	Single to Integer
Integer to Double	Double to Integer
Single to Double	Double to Single

Two-Operand Operations:

Add	Multiply
Subtract	Divide
Compare	

Matrix Operations:

4x4, 4x4 Multiply†	3x3, 3x3 Multiply†
1x4, 4x4 Multiply†	1x3, 3x3 Multiply†

Graphics Operations:

Backface Testing†	Polygon Elimination†
Polygon Clipping†	Viewport Scaling and Conversion†
2-D Linear Interpolation†	3-D Linear Interpolation†
2-D Window Compare†	3-D Volume Compare†
2-Plane Clipping (X,Y,Z)†	2-Plane Color Clipping (R,B,G,I)†
2-D Cubic Spline†	3-D Cubic Spline†

Image Processing:

3x3 Convolution†	

Chained Operations :

Polynomial Expansion†	Multiply/Accumulate†
1-D Min/Max†	2-D Min/Max†

Vector Operations:

Add†	Dot Product†
Subtract†	Cross Product†
Magnitude†	Normalization†
Scaling†	Reflection†

The internal ROM routines may be used in either the coprocessor or host-independent mode. In the coprocessor mode, the internal routines are invoked by TMS34020 instructions to its coprocessor(s).

In the host-independent mode, the internal programs can be called as subroutines by the externally stored code. External programs can call internal routines by executing a jump to subroutine with bit 16 (internal code select) set high and the address of the internal routine as the jump address.

The format of the TMS34082A instruction in the coprocessor mode is shown in Figure 49. The instruction is issued by the TMS34020 via the LAD bus.

† Indicates a complex instruction.

PROGRAMMING INFORMATION

31	28	24	20	15	13	8	7	6	5					0
ID	ra	rb	rd	md	fpuop	type	size	0	1	0	0	0	0	0

FIGURE 29. TMS34082A INSTRUCTION

The 3-bit ID field identifies the coprocessor for which the instruction is intended. This coprocessor ID corresponds to the settings of the CID2-CID0 pins. To broadcast an instruction to all coprocessors, the ID is set to 4h.

TABLE 5. COPROCESSOR ID

ID	COPROCESSOR
000	FPU0
001	FPU1
010	FPU2
011	FPU3
100	FPU broadcast
101	Reserved
110	Reserved
111	User defined

Four coprocessor addressing modes are defined for the TMS34082A. The md field indicates the addressing mode.

TABLE 6. ADDRESSING MODES

MODE	MD FIELD	OPERATION
0	00	FPU internal operations with no jump or external moves
1	01	Transfer data to/from TMS34020 registers
2	10	Transfer data to/from memory (controlled by TMS34020)
3	11	External instructions

The type and size bits identify the type of operand; as shown below in Table 7. The I bit is used to indicate to the TMS34082A that this is a reissue of a coprocessor instruction due to a bus interruption. The least significant four bits are the bus status bits, which will all be zero to indicate a coprocessor cycle.

TABLE 7. OPERAND TYPES

TYPE	SIZE	OPERAND TYPE
0	0	32-bit integer
0	1	Reserved
1	0	Single-precision floating-point (32-bit)
1	1	Double-precision floating-point (64-bit)

The ra, rb, and rd fields are for the two sources and destination within the FPU. Register addresses are listed in Table 1. For the ra and rb fields, only the four least significant bits of the register address are used. The ra field may only use the RA register file, C, and CT. The RB field may only use the RB register file, C and CT.

The Floating-Point Unit Operation (fpuop) field is the FPU opcode (5 bits) described in Tables 8, 9, and 10.

In the coprocessor mode, the TMS34082A executes user-defined routines (stored in external memory on the MSD bus) by executing a jump to external code. For this instruction, the md field (bits 15-13) is set high and the fpuop field gives the routine number (0-31). The TMS34082A multiplies the routine number by two to get the jump address. For example, routine number 14 would have a jump address of 28 decimal or 1C hex.

PROGRAMMING INFORMATION

The routines are coded using the external instruction format discussed in the next section. The last instruction should be a jump to internal instruction address 0FFFh with the I-bit(internal) set or a return from subroutine instruction. This puts the FPU in an idle state, waiting for the next instruction from the TMS34020.

TABLE 8. COPROCESSOR MODE INSTRUCTIONS

FPUOP	TMS34020 ASSEMBLER OPCODE	DESCRIPTION
00000	ADDx	Sum of ra and rb, place in rd
00001	SUBx	Subtract rb from ra, place result in rd
00010	CMPx	Set status bits on result of ra minus rb
00011	SUBx	Subtract ra from rb, place result in rd
00100	ADDAx	Absolute value of sum of ra and rb, place result in rd
00101	SUBAx	Absolute value of (ra minus rb), place result in rd
00110	MOVE or MOVx	Load multiple FPU registers from TMS34020 GSP or its memory
00111	MOVE or MOVx	Save multiple FPU registers to TMS34020 GSP or its memory
01000	MPYx	Multiply ra and rb, place result in rd
01001	DIVx	Divide ra by rb, place result in rd
01010	INVx	Divide 1 by rb, place result in rd
01011	ASUBAx	Absolute value of ra minus absolute value of rb, place in rd
01100	reserved	
01101	MOVEx	Move ra to rd, multiple, for n registers
01110	MOVEx	Move rb to rd, multiple, for n registers
01111	(see Table 10)	Single operand instructions, rb field redefined
10000	CPWx	Compare point to window (set XLT, XGT, YLT, TGT)
10001	CPVx	Compare point to volume (set XLT, XGT, YLT, YGT, ZLT, ZGT)
10010	BACKFx	Test polygon for facing direction (backface test)
10011	INMNMXx	Setup FPU registers for MNMX1 or MNMX2 instruction
10100	LINTx	Given [X1, Y1, Z1], [X2, Y2, Z2], and a plane, find [X3, Y3, Z3]
10101	CLIPFx	Clip a line to a plane pair boundary (start with point 1)
10110	CLIPRx	Clip a line to a plane pair boundary (start with point 2)
10111	CLIPCFx	Clip color values to a plane pair boundary (start with point 1)
11000	SCALEx	Scale and convert coordinates for viewpoint
11001	MTRANx	Transpose a matrix
11010	CKVTXx	Compare a polygon vertex to a clipping volume
11011	CONVx	3x3 convolution
11100	CLIPCRx	Clip color values to a plane pair boundary (start with point 2)
11101	OUTC3x	Compare a line to a clipping value
11110	CSPLNx	Calculate cubic spline for given coefficients
11111	(see Table 11)	Vector and matrix instructions, rb field redefined

F denotes single-precision, D denotes double-precision floating-point, x denotes operand type, and a blank designates signed integer

PROGRAMMING INFORMATION

TABLE 9. COPROCESSOR MODE INSTRUCTIONS, FPUOP = 01111_2

RB	TMS34020 ASSEMBLER OPCODE	DESCRIPTION
0000	PASS	Copy ra to rd
0001	NOT	Place 1s complement of ra in rd
0010	ABS	Place absolute value of ra in rd
0011	NEG	Place negated value of ra in rd
0100	CVDF	Convert double in ra to single in rd (T and S define ra)
0100	CVFD	Convert single in ra to double in rd (T and S define ra)
0101	CVDI	Convert double in ra to integer in rd (T and S define ra)
0101	CVFI	Convert single in ra to integer in rd (T and S define ra)
0110	CVID	Convert integer in ra to double in rd (T and S define ra)
0110	CVIF	Convert integer in ra to single in rd (T and S define ra)
0111	VSCLx	Multiply each component of a velocity by a scaling factor
1000	SQARx	Place (ra * ra) in rd
1001	SQRTx	Extract square root or ra, place in rd
1010	SQRTAx	Extract square root of absolute value of ra, place in rd
1011	ABORT	Stop execution of any FPU instruction
1100	CKVTXI	Initialize check vertex instruction
1101	CHECK	Check for previous instruction completion
1110	MOVMEM	Move data from system memory to external memory @ MCADDR
1111	MOVMEM	Move data to system memory from external memory @ MCADDR

TABLE 10. COPROCESSOR MODE INSTRUCTIONS, FPUOP = 11111_2

RB	TMS34020 ASSEMBLER OPCODE	DESCRIPTION
0000	POLYx	Polynomial expansion
0001	MACx	Multiply and accumulate
0010	MNMX1x	Determine 1-D minimum and maximum of a series
0011	MNMX2x	Determine 2-D minimum and maximum of a series of pairs
0100	MMPY0x	Multiply matrix elements 0, 1, 2, 3 by vector element 0
0101	MMPY1x	Multiply matrix elements 4, 5, 6, 7 by vector element 1
0110	MMPY2x	Multiply matrix elements 8, 9, 10, 11 by vector element 2
0111	MMPY3x	Multiply matrix elements 12, 13, 14, 15 by vector element 3
1000	MADDx	Add matrix elements 12, 13, 14, 15 to vector
1001	VADDx	Add two vectors
1010	VSUBx	Subtract a vector from a vector
1011	VDOTx	Compute scalar dot product of two vectors
1100	VCROSx	Compute cross product of two vectors
1101	VMAGx	Determine the magnitude of a vector
1110	VNORMx	Normalize a vector to unit magnitude
1111	VRFLCTx	Given normal and incident vectors, find the reflection

F denotes single-precision, D denotes double-precision floating-point, x denotes operand type, and a blank designates signed integer

PROGRAMMING INFORMATION

external instructions

External instructions are 32 bits long, and their formats (number, length, and function of fields) depend on the operations being selected. Separate formats are provided for data transfers, FPU processing, test and branch operations, and subroutine calls.

Instructions that control FPU operations can select operands from input registers, internal feedback, or from the LAD bus (32-bit operations only). The format for an FPU processing instruction is shown in Figure 50.

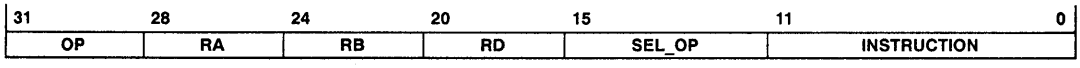


FIGURE 30. FPU PROCESSING EXTERNAL INSTRUCTION FORMAT

The op field selects the sequencer operation. Three continue instructions are available to permit control of the $\overline{WE}$ and $\overline{ALTCH}$ strobe outputs, which enable LAD output in the host-independent mode. The ra, rb, and rd fields are for the two sources and destination in the TMS34082A register file. The sel_op field selects the source of the operands: register file or feedback registers. The instruction field designates the operation to be performed.

External instructions and cycle counts are listed in Table 11. Absolute values of operands or results, negated results, and wrapped number inputs are selectable options. Chained operations, using the multiplier and ALU in parallel, and other instructions to control program flow and move data are included.

External instruction timing depends on the pipeline registers setting, controlled by the PIPES2-1 bits in the configuration register. Most FPU processing instructions (with the exception of divide, square root, and double-precision multiply) execute in one cycle per pipeline stage.

PROGRAMMING INFORMATION

TABLE 11. EXTERNAL INSTRUCTIONS AND TIMING

TMS34082A ASSEMBLER OPCODE	DESCRIPTION OF ROUTINE	PIPES2–1 11	PIPES2–1 10	PIPES2–1 01	PIPES2–1 00
ADD	Add A + B	1(1)	2(1)	2(1)	3(1)
AND	Logical AND A, B	1(1)	2(1)	2(1)	3(1)
ANDNA	Logical AND NOT A, B	1(1)	2(1)	2(1)	3(1)
ANDNB	Logical AND A, NOT B	1(1)	2(1)	2(1)	3(1)
CJMP	Conditional jump	1(1)	1(1)	1(1)	1(1)
CSJR	Conditional jump to subroutine	1(1)	1(1)	1(1)	1(1)
CMP	Compare A, B	1(1)	2(1)	2(1)	3(1)
COMPL	Pass 1s complement of A	1(1)	2(1)	2(1)	3(1)
DIV	Divide A / B				
	SP	8(8)	8(7)	9(7)	9(7)
	DP	13(13)	13(12)	15(12)	15(12)
	integer	16(16)	16(15)	17(15)	17(15)
DTOF	Convert from DP to SP	1(1)	2(1)	2(1)	3(1)
DTOI	Convert from DP to integer	1(1)	2(1)	2(1)	3(1)
DTOU	Convert from DP to unsigned integer	1(1)	2(1)	2(1)	3(1)
FTOD	Convert from SP to DP	1(1)	2(1)	2(1)	3(1)
FTOI	Convert from SP to integer	1(1)	2(1)	2(1)	3(1)
FTOU	Convert from SP to unsigned integer	1(1)	2(1)	2(1)	3(1)
ITOD	Convert from integer to DP	1(1)	2(1)	2(1)	3(1)
ITOF	Convert from integer to SP	1(1)	2(1)	2(1)	3(1)
LD	Load n words into register				
	SP	n + 1	n + 1	n + 1	n + 1
	DP	2n + 1	2n + 1	2n + 1	2n + 1
	integer	n + 1	n + 1	n + 1	n + 1
LDLCT	Load loop counter with value	1(1)	1(1)	1(1)	1(1)
LDMCADDR	Load MCADDR with value	1(1)	1(1)	1(1)	1(1)
MASK	Set programmable mask	1(1)	1(1)	1(1)	1(1)
MOVA	Move A (no status flags active)	1(1)	2(1)	2(1)	3(1)
MOVLM	Move n words from LAD bus to MSD bus				
	SP	n + 1	n + 1	n + 1	n + 1
	DP	2n + 1	2n + 1	2n + 1	2n + 1
	integer	n + 1	n + 1	n + 1	n + 1
MOVML	Move n words from MSD bus to LAD bus				
	SP	n + 1	n + 1	n + 1	n + 1
	DP	2n + 1	2n + 1	2n + 1	2n + 1
	integer	n + 1	n + 1	n + 1	n + 1
MOVRR	Multiple move, register to register				
	SP	n + 1	n + 1	n + 1	n + 1
	DP	2n + 1	2n + 1	2n + 1	2n + 1
	integer	n + 1	n + 1	n + 1	n + 1
MULT.ADD	Multiply $A_1 * B_1$, Add $A_2 + B_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)

DP denotes double-precision, and SP denotes single-precision.

PROGRAMMING INFORMATION

TABLE 11. EXTERNAL INSTRUCTIONS AND TIMING (Continued)

TMS34082A ASSEMBLER OPCODE	DESCRIPTION OF ROUTINE	PIPES2–1 11	PIPES2–1 10	PIPES2–1 01	PIPES2–1 00
MULT.NEG	Multiply $A_1 * B_1$, Subtract $0 - A_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
MULT	Multiply A * B				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
MULT.PASS	Multiply $A_1 * B_1$, Add $A_2 + 0$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
MULT.SUB	Multiply $A_1 * B_1$, Subtract $A_2 - B_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
MULT.2SUBA	Multiply $A_1 * B_1$, Subtract $2 - A_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
MULT.SUBRL	Multiply $A_1 * B_1$, Subtract $B_2 - A_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
NEG	Pass –A (2s Complement)	1(1)	2(1)	2(1)	3(1)
NOR	Logical NOR A, B	1(1)	2(1)	2(1)	3(1)
OR	Logical OR A, B	1(1)	2(1)	2(1)	3(1)
PASS	Pass A	1(1)	2(1)	2(1)	3(1)
PASS	Pass B	1(1)	2(1)	2(1)	3(1)
PASS.ADD	Multiply $A_1 * 1$, Add $A_2 + B_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
PASS.NEG	Multiply $A_1 * 1$, Subtract $0 - A_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
PASS.PASS	Multiply $A_1 * 1$, Add $A_2 + 0$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
PASS.SUB	Multiply $A_1 * 1$, Subtract $A_2 - B_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
PASS.2SUBA	Multiply $A_1 * 1$, Subtract $2 - A_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)

DP denotes double-precision, and SP denotes single-precision.

PROGRAMMING INFORMATION

TABLE 11. EXTERNAL INSTRUCTIONS AND TIMING (Continued)

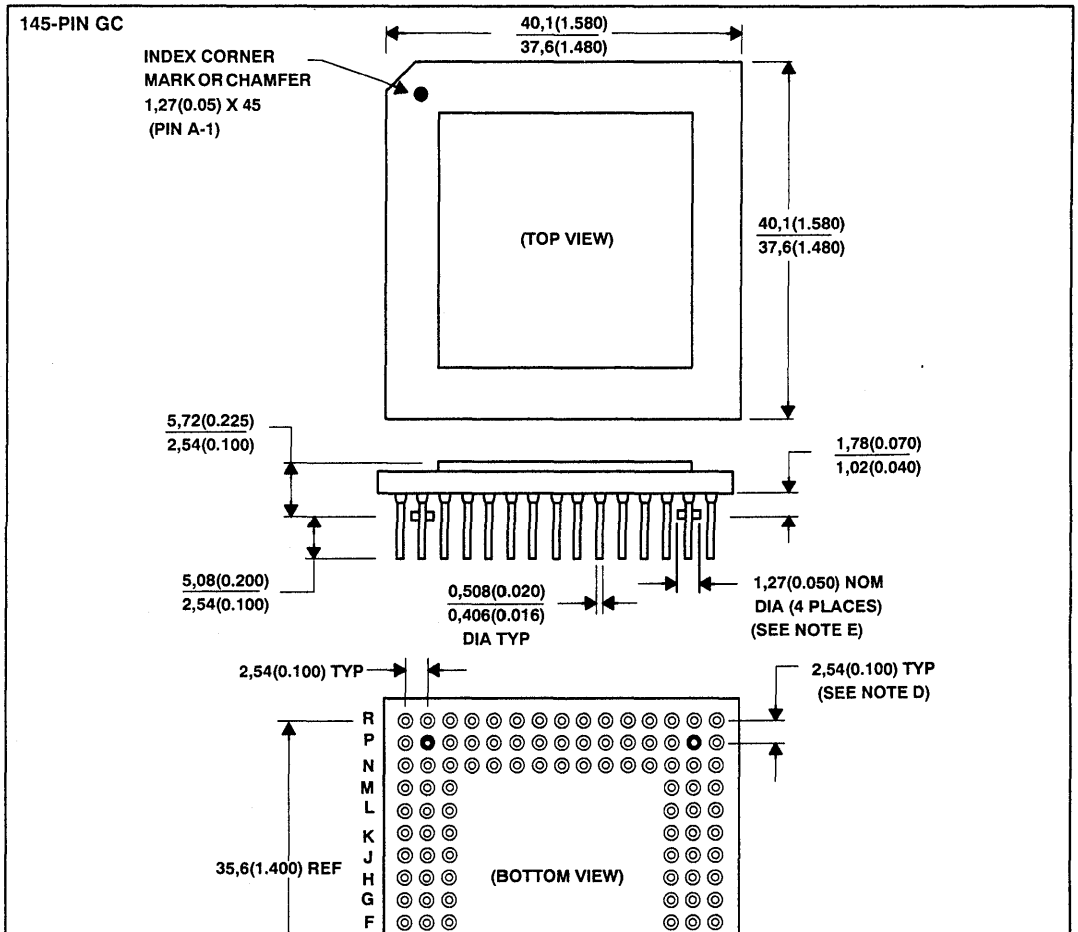
TMS34082A ASSEMBLER OPCODE	DESCRIPTION OF ROUTINE	CYCLE COUNTS			
		PIPES2–1 11	PIPES2–1 10	PIPES2–1 01	PIPES2–1 00
RTS	Return from subroutine	1(1)	1(1)	1(1)	1(1)
SLL	Logical shift left A by B bits	1(1)	2(1)	2(1)	3(1)
SQRT	Square root of A				
	SP	11(11)	11(10)	12(10)	12(10)
	DP	16(16)	16(15)	17(15)	17(15)
	integer	20(20)	20(19)	21(19	21(19)
PASS.SUBRL	Multiply $A_1 * 1$, Subtract $B_2 - A_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
SRA	Arithmetic shift right A by B bits	1(1)	2(1)	2(1)	3(1)
SRL	Logical shift right A by B bits	1(1)	2(1)	2(1)	3(1)
ST	Store n words from register				
	SP	n + 1	n + 1	n + 1	n + 1
	DP	2n + 1	2n + 1	2n + 1	2n + 1
	integer	n + 1	n + 1	n + 1	n + 1
SUB	Subtract A – B	1(1)	2(1)	2(1)	3(1)
SUBRL	Subtract B – A	1(1)	2(1)	2(1)	3(1)
UTOD	Convert from unsigned integer to DP	1(1)	2(1)	2(1)	3(1)
UTOF	Convert from unsigned integer to SP	1(1)	2(1)	2(1)	3(1)
UWRAPI	Unwrap inexact operand	1(1)	2(1)	2(1)	3(1)
UWRAPR	Unwrap rounded operand	1(1)	2(1)	2(1)	3(1)
UWRAPX	Unwrap exact operand	1(1)	2(1)	2(1)	3(1)
WRAP	Wrap denormalized operand	1(1)	2(1)	2(1)	3(1)
XOR	Logical exclusive OR A, B	1(1)	2(1)	2(1)	3(1)

DP denotes double-precision, and SP denotes single-precision.

MECHANICAL DATA

GC pin-grid-array ceramic package

This is a hermetically sealed package.



NOTES: A. Pins are located within 0,13 (0.005) radius of true position relative to each other at maximum meterial condition and within 0,457 (0.018) radius of the center of the ceramic.
B. Dimensions do not include solder finish.

Appendix C

SMJ34082A Data Sheet

The pinout, electrical specifications timing diagrams, and mechanical specifications are contained within the SMJ34082A Data Sheet and appear in this appendix.

The SMJ34082A is fully characterized over military temperature range.

SGUS012A – D3592, SEPTEMBER 1990 – REVISED MAY 1991

- Military Temperature Range (–55°C to 125°C)
- Class B, High-Reliability Processing
- High-Performance Floating-Point RISC Processor Optimized for Graphics
- Two Operating Modes
  - Floating-Point Coprocessor for SMJ34020 Graphics System Processor
  - Independent Floating-Point Processor
- Direct Connection to SMJ34020 Coprocessor Interface
  - Direct Extension to the SMJ34020 Instruction Set
  - Multiple SMJ34082A Capability
- Fast Pipelined Instruction Cycle Time
  - SMJ34082A-30 . . . 66-ns Coprocessor Mode . . . 65-ns Host-Independent Mode
  - SMJ34082A-28 . . . 70-ns Coprocessor Mode . . . 70-ns Host-Independent Mode
- Sustained Data Transfer Rates of 120 Mbytes/s (SMJ34082A-30)
- Sequencer Executes Internal or User-Programmed Instructions
- 22 64-Bit Data Registers
- Comprehensive Floating-Point and Integer Instruction Set
- Internal Programs for Vector, Matrix, and 3-D Graphics Operations
- Full IEEE Standard 754-1985 Compatibility
  - Addition, Subtration, Multiplication, and Comparison
  - Division and Square Root
- Selectable Data Formats
  - 32-Bit Integer
  - 32-Bit Single-Precision Floating-Point
  - 64-Bit Double-Precision Floating-Point
- External Memory Addressing Capability
  - Program Storage (up to 64K Words)
  - Data Storage (up to 64K Words)
- 0.8-μm EPIC™ CMOS Technology
  - High-Performance
  - Low Power (< 2 W)

description

The SMJ34082A is a high-speed graphics floating-point processor implemented in Texas Instruments advanced 0.8-μm CMOS technology. The SMJ34082A combines a 16-bit sequencer and a 3-operand (source A, source B, and destination) 64-bit Floating-Point Unit (FPU) with 22 64-bit data registers on a single chip. The data registers are organized into two files of ten registers each, with two registers for internal feedback. In addition, it provides an instruction register to control FPU execution, a status register to retain the most recent FPU status outputs, eight control registers, and a two-deep stack (see functional block diagram).

The SMJ34082A is fully compatible with IEEE Standard 754-1985 for binary floating-point addition, subtraction, multiplication, division, square root, and comparison. Floating-point operands can be either in single- or double-precision IEEE format.

In addition to floating-point operations, the SMJ34082A performs 32-bit integer arithmetic, logical comparisons, and shifts. Integer operations may be performed on 32-bit 2s complement or unsigned operands. Integer results are 32-bits long (even for 32 x 32 integer multiplication). Absolute value conversions, floating-point to integer conversions, and integer to floating-point conversions are available.

The ALU and the multiplier are closely coupled and can be operated in parallel to perform sums of products or products of sums. During multiply/accumulate operations, both the ALU and the multiplier are active and the registers in the FPU core can be used to feedback products and accumulate sums without tying up locations in register files A and B.

When used with the SMJ34020, the SMJ34082A operates in the coprocessor mode. The SMJ34020 can control multiple SMJ34082A coprocessors. When used as a stand-alone or with processors other than the SMJ34020, the SMJ34082A operates in the host-independent mode. The SMJ34082A is fully programmable by the user

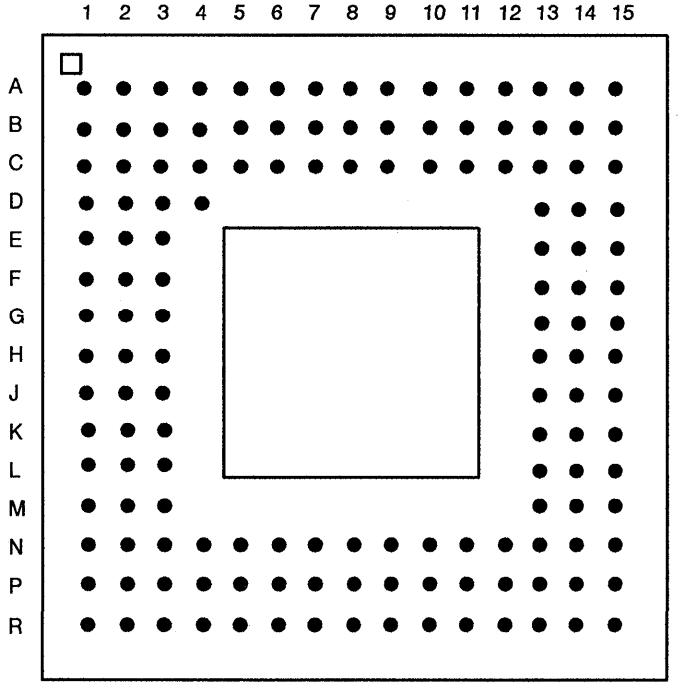
EPIC is a trademark of Texas Instruments Incorporated.

and can interface to other processors or floating-point subsystems through its two 32-bit bidirectional buses. In the coprocessor mode, the TMS340 family tools may be used to develop code for the SMJ34082A. The TMS34082A Software Tool Kit is used to develop code for host-independent mode applications or for external routines in the coprocessor mode.

pin descriptions

Pin descriptions and grid assignments for the SMJ34082A are given on the following pages. The pin at location D4 has been added for indexing purposes.

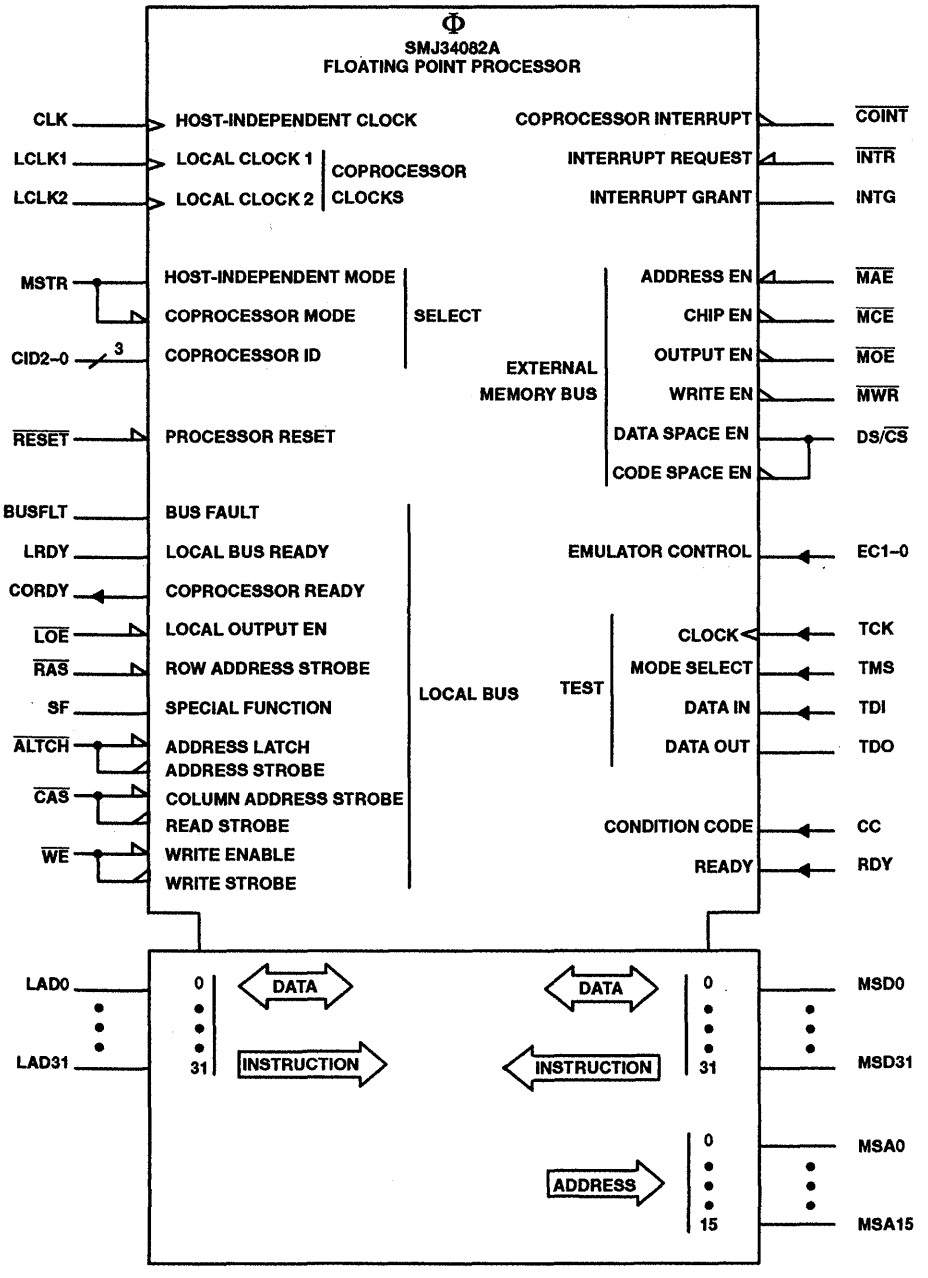
145-PIN GB PACKAGE
(TOP VIEW)



Pin Grid Assignments

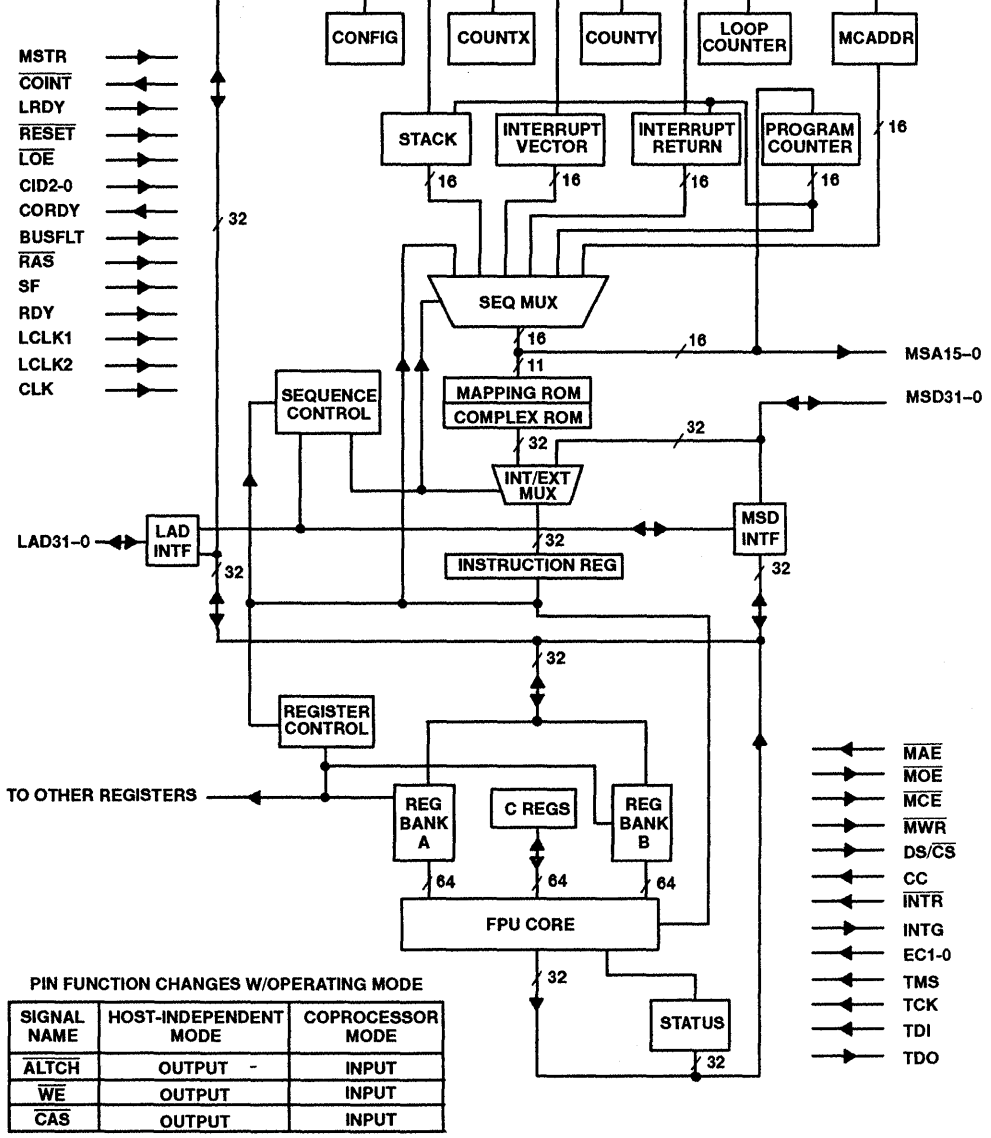
PIN NO.	PIN NAME	PIN NO.	PIN NAME	PIN NO.	PIN NAME	PIN NO.	PIN NAME	PIN NO.	PIN NAME
A1	NC	B15	LAD27	F1	MSD10	K15	RDY	P2	NC
A2	LAD1	C1	MSD4	F2	MSD9	L1	MSD18	P3	MSD29
A3	LAD3	C2	MSD3	F3	V_{CC}	L2	MSD21	P4	MSD31
A4	LAD5	C3	MSD0	F13	CORDY	L3	MSD23	P5	MSA1
A5	LAD8	C4	V_{SS}	F14	$\overline{ALTCH}$	L13	V_{SS}	P6	MSA3
A6	LAD9	C5	V_{CC}	F15	$\overline{CAS}$	L14	CID0	P7	MSA6
A7	LAD11	C6	LAD6	G1	MSD13	L15	CID2	P8	MSA8
A8	LAD12	C7	V_{SS}	G2	MSD12	M1	MSD20	P9	MSA10
A9	LAD13	C8	V_{CC}	G3	MSD11	M2	MSD24	P10	MSA13
A10	LAD15	C9	V_{SS}	G13	$\overline{WE}$	M3	V_{SS}	P11	$\overline{MWR}$
A11	LAD17	C10	V_{CC}	G14	EC1	M13	V_{CC}	P12	$\overline{MOE}$
A12	LAD19	C11	LAD21	G15	EC0	M14	LCLK1	P13	INTG
A13	LAD22	C12	V_{SS}	H1	MSD14	M15	LCLK2	P14	BUSFLT
A14	LAD24	C13	LAD25	H2	TDO	N1	MSD22	P15	$\overline{RAS}$
A15	NC	C14	LAD26	H3	V_{SS}	N2	MSD26	R1	NC
B1	MSD1	C15	LAD29	H13	V_{SS}	N3	V_{CC}	R2	MSD27
B2	NC	D1	MSD6	H14	$\overline{LOE}$	N4	MSD28	R3	MSD30
B3	LAD0	D2	MSD5	H15	TDI	N5	V_{SS}	R4	MSA0
B4	LAD2	D3	MSD2	J1	MSD15	N6	V_{CC}	R5	MSA2
B5	LAD4	D4	NC	J2	MSD16	N7	MSA5	R6	MSA4
B6	LAD7	D13	V_{CC}	J3	V_{CC}	N8	V_{SS}	R7	MSA7
B7	LAD10	D14	LAD28	J13	CC	N9	V_{CC}	R8	TCK
B8	TMS	D15	LAD31	J14	MSTR	N10	MSA14	R9	MSA9
B9	LAD14	E1	MSD8	J15	CLK	N11	V_{SS}	R10	MSA11
B10	LAD16	E2	MSD7	K1	MSD17	N12	$\overline{MAE}$	R11	MSA12
B11	LAD18	E3	V_{SS}	K2	MSD19	N13	LRDY	R12	MSA15
B12	LAD20	E13	V_{SS}	K3	V_{SS}	N14	SF	R13	DS/$\overline{CS}$
B13	LAD23	E14	LAD30	K13	CID1	N15	$\overline{RESET}$	R14	$\overline{MCE}$
B14	NC	E15	$\overline{COINT}$	K14	$\overline{INTR}$	P1	MSD25	R15	NC

logic symbol†



† This symbol is in accordance with ANSI/IEEE Std 91-1984.

functional block diagram



PIN FUNCTION CHANGES W/OPERATING MODE

SIGNAL NAME	HOST-INDEPENDENT MODE	COPROCESSOR MODE
ALTCH	OUTPUT	INPUT
WE	OUTPUT	INPUT
CAS	OUTPUT	INPUT

Terminal Functions

PIN NAME	PIN NO.	I/O†	DESCRIPTION
$\overline{\text{ALTCH}}$	F14	I [O]	Address Latch, active low. In the coprocessor mode, falling edge of $\overline{\text{ALTCH}}$ latches instruction and status present on the LAD bidirectional bus (LAD31-0). In the host-independent mode, $\overline{\text{ALTCH}}$ is address output strobe for memory accesses on LAD31-0.
BUSFLT	P14	I	Bus Fault. In the coprocessor mode, BUSFLT high indicates a data fault on the LAD bus (LAD31-0) during current bus cycle, which in turn causes SMJ34082A not to capture current data on LAD bus. Tied low if not used or in the host-independent mode.
$\overline{\text{CAS}}$	F15	I [O]	Column Address Strobe, active low. In the coprocessor mode, causes SMJ34082A to latch LAD bus data when $\overline{\text{CAS}}$ has a low-to-high transition if LRDY was high and BUSFLT was low at the previous LCLK2 rising edge. In the host-independent mode, this signal is the read strobe output.
CC	J13	I	Condition Code Input. In both modes, may be used as an external conditional input for branch conditions.
CID0 CID1 CID2	L14 K13 L15	I	Coprocessor ID. In the coprocessor mode, used to set a coprocessor ID so that a SMJ34020 Graphics System Processor controlling multiple SMJ34082A coprocessors can designate which coprocessor is being selected by the current instruction. Tied low in the host-independent mode.
CLK	J15	I	System Clock. In the coprocessor mode, tied low. In the host-independent mode, input is the system clock.
$\overline{\text{COINT}}$	E15	O	Coprocessor Interrupt Request, active low. In the coprocessor mode, signals an exception not masked out in the configuration register. Remains low until the status register is read. In the host-independent mode, user programmable I/O when LADCFG is low. When LADCFG is high, designates bus cycle boundaries on LAD31-0.
CORDY	F13	O	Coprocessor Ready. In the coprocessor mode, if the SMJ34020 sends an instruction before the SMJ34082A has completed a previous instruction, this signal goes low to indicate that the SMJ34020 should wait. In the host-independent mode, user programmable.
DS/$\overline{\text{CS}}$	R13	O	Data Space/Code Space. In both modes, when MEMCFG is low and DS/$\overline{\text{CS}}$ is low, selects program memory on MSD port. When MEMCFG is low and DS/$\overline{\text{CS}}$ is high, selects data memory on MSD port. When MEMCFG is high, DS/$\overline{\text{CS}}$ is memory chip select, active low.
EC0 EC1	G15 G14	I	Emulator Mode Control and Test. In both modes, tied high for normal operation.
INTG	P13	O	Interrupt Grant Output. In the coprocessor mode, INTG is low. In the host-independent mode, this signal is set high to acknowledge an interrupt request input.
$\overline{\text{INTR}}$	K14	I	Interrupt Request Input, active low. In the coprocessor mode, $\overline{\text{INTR}}$ is tied high. In the host-independent mode, causes call to subroutine address in interrupt vector register.

† The [ ]'s denote the type of buffer ut
ilized in the host-independent mode. If no [ ]'s appear, the buffer type is identical for both modes of operation.

Terminal Functions (Continued)

PIN NAME	PIN NO.	I/O†	DESCRIPTION
LAD0	B3	I/O	Local Address and Data Bus. In the coprocessor mode, used by SMJ34020 to input instructions and data operands to SMJ34082, and used by SMJ34082A to output results. In the host-independent mode, used by the SMJ34082A for address output and data I/O.
LAD1	A2		
LAD2	B4		
LAD3	A3		
LAD4	B5		
LAD5	A4		
LAD6	C6		
LAD7	B6		
LAD8	A5		
LAD9	A6		
LAD10	B7		
LAD11	A7		
LAD12	A8		
LAD13	A9		
LAD14	B9		
LAD15	A10		
LAD16	B10		
LAD17	A11		
LAD18	B11		
LAD19	A12		
LAD20	B12		
LAD21	C11		
LAD22	A13		
LAD23	B13		
LAD24	A14		
LAD25	C13		
LAD26	C14		
LAD27	B15		
LAD28	D14		
LAD29	C15		
LAD30	E14		
LAD31	D15		
LCLK1	M14	I	Local Clocks 1 and 2. In the coprocessor mode, two local clocks generated by the SMJ34020, 90 degrees out of phase, to provide timing inputs to SMJ34082A. In the host-independent mode, tied low.
LCLK2	M15		
$\overline{LOE}$	H14	I	Local Bus Output Enable, active low. In both modes, enables the local bus (LAD31-0) to be driven at the proper times when low. In addition during the host-independent mode when LADCFG is low, does not affect $\overline{ALTCH}$, $\overline{CAS}$, $\overline{WE}$, CORDY, or $\overline{COINT}$. When LADCFG is high, $\overline{ALTCH}$, $\overline{COINT}$, and CORDY are not disabled by $\overline{LOE}$ high; $\overline{CAS}$ and $\overline{WE}$ are disabled.
LRDY	N13	I	Local Bus Data Ready. In the coprocessor mode, when LRDY is high, indicates that data is available on LAD bus. When LRDY is low, indicates that the SMJ34082A should not load data from LAD31-0 and may also be used in conjunction with BUSFLT. In the host-independent mode, when LRDY is low, the device is stalled until LRDY is set high again and tied high if not used.
$\overline{MAE}$	N12	I	Memory Address and Data Output Enable, active low. In both modes, with $\overline{MAE}$ low, the SMJ34082A can output an address on MSA15-0 and data on MSD31-0. $\overline{MAE}$ high does not disable DS/$\overline{CS}$, $\overline{MCE}$, $\overline{MWR}$, or $\overline{MOE}$.
$\overline{MCE}$	R14	O	Memory Chip Enable. In both modes, when MEMCFG low, active (low) indicates access to external memory on MSD31-0. When MEMCFG is high, $\overline{MCE}$ low is external code memory chip select.
$\overline{MOE}$	P12	O	Memory Output Enable, active low. In both modes when low, enables output from external memory on to MSD port.

Terminal Functions (Continued)

PIN NAME	PIN NO.	I/O†	DESCRIPTION
MSA0	R4	O	Memory Address output. In both modes, addresses up to 64K words of external program memory and/or up to 64K words of data memory on the MSD port, depending on setting of DS/$\overline{CS}$ select.
MSA1	P5		
MSA2	R5		
MSA3	P6		
MSA4	R6		
MSA5	N7		
MSA6	P7		
MSA7	R7		
MSA8	P8		
MSA9	R9		
MSA10	P9		
MSA11	R10		
MSA12	R11		
MSA13	P10		
MSA14	N10		
MSA15	R12		
MSD0	C3	I/O	External Memory Data. In both modes, I/Os to external memory. Used to read from or write to external data or program memory on the MSD port.
MSD1	B1		
MSD2	D3		
MSD3	C2		
MSD4	C1		
MSD5	D2		
MSD6	D1		
MSD7	E2		
MSD8	E1		
MSD9	F2		
MSD10	F1		
MSD11	G3		
MSD12	G2		
MSD13	G1		
MSD14	H1		
MSD15	J1		
MSD16	J2		
MSD17	K1		
MSD18	L1		
MSD19	K2		
MSD20	M1		
MSD21	L2		
MSD22	N1		
MSD23	L3		
MSD24	M2		
MSD25	P1		
MSD26	N2		
MSD27	R2		
MSD28	N4		
MSD29	P3		
MSD30	R3		
MSD31	P4		
MSTR	J14	I	Host-Independent/Coprocessor Mode Select. In the coprocessor mode, MSTR must be tied low to operate properly. In the host-independent mode, MSTR must be tied high to operate properly.
$\overline{MWR}$	P11	O	Memory Write Enable. In both modes, when low, data on MSD31-0 can be written to external program or data memory.

Terminal Functions (Continued)

PIN NAME	PIN NO.	I/O†	DESCRIPTION
NC	A1 A15 B2 B14 D4 P2 R1 R15		No Internal Connection. These pins should be left floating.
$\overline{RAS}$	P15	I	Row Address Strobe, active low. In the coprocessor mode, $\overline{RAS}$ is high during all of coprocessor instruction cycle. In the host-independent mode, it is not used.
RDY	K15	I	Ready. In both modes, when RDY is low, it causes a nondestructive stall of sequencer and floating-point operations. All internal registers and status in the FPU core are preserved. Also, no output lines will change state.
$\overline{RESET}$	N15	I	Reset, active low. In both modes, resets sequencer output and clears pipeline registers, internal states, status, and exception disable registers in FPU core. Other registers are unaffected.
SF	N14	I	Special Function Input. In the coprocessor mode when SF is high, indicates the LAD bus input is an instruction or data from SMJ34020 registers. When SF is low, indicates the LAD input is a data operand from memory. In the host-independent mode, not used.
TCK	R8	I	Test Clock for JTAG four-wire boundary scan. In both modes, TCK is low for normal operation.
TDI	H15	I	Test Data Input for JTAG four-wire boundary scan. In both modes, TDI may be left floating.
TDO	H2	O	Test Data Output for JTAG four-wire boundary scan
TMS	B8	I	Test Mode Select for JTAG four-wire boundary scan. In both modes, SMJ may be left floating.
V_{CC}	C5 C8 C10 D13 F3 J3 M13 N3 N6 N9	I	5-V Power Supply. All pins must be connected and used.
V_{SS}	C4 C7 C9 C12 E3 E13 H3 H13 K3 L13 M3 N5 N8 N11	I	Ground Pins. All pions must be connected and used.
$\overline{WE}$	G13	I [O]	Write Enable, active low. In the coprocessor mode, the write strobe from the SMJ34020 to enable a write to or from the SMJ34082A LAD bus. In the host-independent mode, the SMJ34082A write strobe output.

† The [ ]'s denote the type of buffer utilized in the host-independent mode. If no [ ]'s appear, the buffer type is identical for both modes of operation.

data flow

The SMJ34082A has two bidirectional 32-bit buses, LAD31-0 and MSD31-0. Each bus can be used to pass instructions and data operands to the FPU core and to output results. A separate 16-bit bus, MSA15-0, provides memory addressing capability on the MSD bus.

When the SMJ34082A is used as a coprocessor for the SMJ34020 Graphics System Processor (GSP), data for the SMJ34082A can be transferred through the 32-bit bidirectional data bus (LAD31-0) and may be passed to any internal registers or to external memory on the memory expansion interface (MSD31-0). When the SMJ34082A is used as a standalone FPU, it can use both the LAD bus (LAD31-0) and the MSD bus (MSD31-0) to interface with external data memory or system buses.

In the host-independent mode, the SMJ34082A can be operated with the LAD bus as its single data bus and the MSD bus as the instruction source, or with data storage on either port and the program memory on the MSD bus.

The data space/code space (DS/$\overline{CS}$) output can be used to control access either to data memory or program memory on the MSD port. Up to 64K words of code space and 64K words of data space are directly supported. In the coprocessor mode, both instructions and data are transferred on the LAD bus with the option of accessing external user-generated programs on the MSD port.

One 32-bit operand can be input to the data registers each clock cycle. A 64-bit double-precision floating-point operand is input in two cycles. Transfers to or from the data registers can normally be programmed as block moves, loading one or more sets of operands with a single move instruction to minimize I/O overhead. Several modes for moving operands and instructions are available. Block transfers up to 512 words between the LAD and MSD buses can be programmed in either direction.

To permit direct input to or output from the LAD bus in the host-independent mode, other options for controlling the LAD bus have been implemented. When two 32-bit operands are being selected for input to the FPU core, one operand may be selected from LAD. On output from the FPU, a result may simultaneously be written to a register and to the LAD bus.

During initialization in the host-independent mode, a bootstrap loader can bring 65 32-bit words from the LAD bus and write them out to external program memory on the MSD bus, after which the device begins executing from the first memory location (zero). The first word is loaded into the configuration register. This option facilitates the initial loading of program memory on the MSD port upon power-up.

architecture

Because the sequencer, control and data registers, and FPU core are closely coupled, the SMJ34082A can execute a variety of complex floating-point or integer calculations rapidly, with a minimum of external data transfers. The internal architecture of the FPU core supports concurrent operation of the multiplier and the ALU, providing several options for storing or feeding back intermediate results. Also, several special registers are available to support specific calculations for graphics algorithms. Each of the main architectural elements of the SMJ34082A is discussed below.

The control functions of the SMJ34082A are provided by sequence control logic, register control logic, and bus interface control logic, together with user-programmed configuration settings stored in the configuration register. The on-board sequencer selects the next program execution address, either from internal code or from external program memory. Next-address sources include the program counter, stack, interrupt vector register, interrupt return register, or address register (for indirect jumps).

COUNTX, COUNTY, and MIN-MAX/LOOPCT registers are used for temporary storage by internal graphics routines. They may also serve as temporary storage for the user.

A separate FPU status register is provided, which can be used by test-and-branch instructions to control program execution. Because of the large number of status outputs, branches on status can be easily programmed. The status register contents are also important when dealing with status exceptions including such conditions as overflow, underflow, invalid operations (divide by zero), or illegal data formats such as infinity, Not a Number (NaN), or denormalized operands.

Register control logic permits all data and control registers to be accessed in accordance with applicable architectural restrictions. Register files A and B can be written to or read from the external buses, as can the control registers. Internal registers C and CT are embedded in the FPU core and can only be accessed by the FPU internal buses. The C and CT registers cannot be used as sources or destinations for MOVE instructions, and several registers (listed in Table 1) are not available as sources for FPU operations.

Table 1. Internal Registers

REGISTER ADDRESS	REGISTER NAME	RESTRICTIONS ON USE
00000	RA0	
00001	RA1	
00010	RA2	
00011	RA3	
00100	RA4	
00101	RA5	
00110	RA6	
00111	RA7	
01000	RA8	
01001	RA9	
01010	C†	Not a source or destination for moves
01011	CT†	Not a source or destination for moves
01100	STATUS	Not a source for FPU instructions
01101	CONFIG	Not a source for FPU instructions
01110	COUNTX	Not a source for FPU instructions
01111	COUNTY	Not a source for FPU instructions
10000	RB0	
10001	RB1	
10010	RB2	
10011	RB3	
10100	RB4	
10101	RB5	
10110	RB6	
10111	RB7	
11000	RB8	
11001	RB9	
11010	VECTOR	Not a source for FPU instructions
11011	MCADDR	Not a source for FPU instructions
11100	SUBADD0	Not a source for FPU instructions
11101	SUBADD1	Not a source for FPU instructions
11110	IRAREG	Not a source for FPU instructions
11111	MIN-MAX/LOOPCT	Not a source for FPU instructions

† C and CT registers cannot both be used for FPU operand sources in the same instruction.

register files A and B, feedback registers C and CT

SMJ34082A contains two register files, each with ten 64-bit registers and two 64-bit feedback registers. Most instructions will operate on one value from each of the RA and RB register files and return the result to either the RA or RB files or one of the feedback registers.

When the ONEFILE control bit is high in the configuration register, data written to a register in file RA is simultaneously written to the corresponding location in file RB. In this mode, the two register files act as a ten-word, two-read/one-write register file.

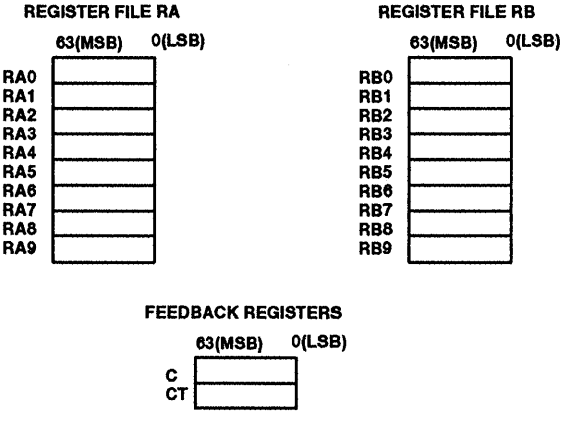


Figure 1. Data Registers

Two 64-bit feedback registers, C and CT, are embedded in the FPU core. FPU instructions may use the feedback registers as one of the operands, but the registers cannot be accessed for external moves. The C and CT registers can be used as either the A or B operand, but both cannot be used as operands during the same instruction. However, C (or CT) may be used for more than one operand in the same instruction. For example, C + CT is not a valid instruction, but C + C is.

The CT feedback register is used in integer divide operations as a temporary holding register. Any data stored in CT will be lost during an integer divide.

Internal control/status register definitions

configuration register definition

The configuration register (CONFIG) is a special 32-bit register that the user loads to configure the SMJ34082A for exception handling, IEEE mode (vs. fast mode), rounding modes, and data-fetch operations. The configuration register is initialized to 'FFE00420' hex.

Table 2. Configuration Register Definition

BIT NO.	NAME	DESCRIPTION
31	MIVAL	Multiplier invalid operation (I) exception mask. Initialized to 1 (enabled).
30	MOVER	Multiplier overflow (V) exception mask. Initialized to 1 (enabled).
29	MUNDER	Multiplier underflow (U) exception mask. Initialized to 1 (enabled).
28	MINEX	Multiplier inexact (X) exception mask. Initialized to 1 (enabled).
27	MDIV0	Divide by zero (DIV0) exception mask. Initialized to 1 (enabled).
26	MDENORM	Multiplier denormal (DENORM) exception mask. Initialized to 1 (enabled).
25	AIVAL	ALU invalid operation (I) exception mask. Initialized to 1 (enabled).
24	AOVER	ALU overflow (V) exception mask. Initialized to 1 (enabled).
23	AUNDER	ALU underflow (U) exception mask. Initialized to 1 (enabled).
22	AINEX	ALU inexact (X) exception mask. Initialized to 1 (enabled).
21	ADENORM	ALU denormal (DENORM) exception mask. Initialized to 1 (enabled).
11-20	N/A	Reserved, set to all 0s.
10	REVISION	Revision number, read only. Set to 1.
9	LADCFG	When low, $\overline{CAS}$, $\overline{WE}$, CORDY, $\overline{COINT}$, and $\overline{ALTCH}$ are active signals not affected by $\overline{LOE}$. When high, $\overline{LOE}$ high places $\overline{CAS}$ and $\overline{WE}$ in high impedance, as well as the LAD bus. $\overline{COINT}$, which defines the LAD cycle boundaries, is controlled by bit 1 of the LAD move instruction instead of the set mask instruction. $\overline{COINT}$ will remain high unless a LAD move instruction (with bit 1 high) is in progress. The setting of this bit has no effect in the coprocessor mode. Initialized to 0.
8	MEMCFG	When high, $\overline{MCE}$ becomes code space chip enable and DS/$\overline{CS}$ becomes data space chip enable (eliminates need for external inverter). When low, $\overline{MCE}$ is chip select for external code and data space. DS/$\overline{CS}$ functions as an address bit which selects code space (when low) or data space (when high). Initialized to 0.
7	N/A	Reserved for later use. Initialized to 0. Must be loaded with 0.
6	ONEFILE	When high, causes simultaneous write to both register files (for example, to both RA0 and RB0 at once). The register files act as a single two-read, one-write register file. Initialized to 0.
5	PIPES2	When high, makes FPU output registers transparent. When low, registers are enabled. Initialized to 1.
4	PIPES1	When high, makes FPU internal pipeline registers transparent. When low, registers are enabled. Initialized to 0.
3	FAST	When high,fast mode is selected (all denormalized inputs and outputs are 0). When low, IEEE mode is selected. Initialized to 0.
2	LOAD	Load order. 0 = MSH, then LSH; 1 = LSH, then MSH. Initialized to 0.
1	RND1	Rounding mode select 1. Initialized to 0.
0	RND0	Rounding mode select 0. Initialized to 0.

LSH denotes least-significant half of a 64-bit word, MSH denotes most-significant half of a 64-bit word.

The mask bits serve as exception detect enables for the exception masks listed above. Setting the bit high (logic '1') enables the detection of the specific exception. When an enabled exception occurs, the ED bit in the status register will be set high and can be used to generate interrupts. The fast bit allows the SMJ34082A to control the handling of denormalized numbers. When the fast bit is set high, all denormalized numbers input to the device are flushed to zero, and all denormalized results are also flushed to zero (this is also called 'sudden underflow'). When the fast bit is low, IEEE mode is selected. Denormalized numbers may be generated by (or input to) the ALU. Denormalized numbers must first be wrapped before being used as operands for multiply or divide instructions.

The LOAD bit defines the expected order of double-precision operands. At reset, this bit will default to 0 indicating that the most significant 32 bits are transferred first. If the bit is set to a 1, then the expected order of 64-bit data transfers starts with the least significant 32 bits.

The RND0 and RND1 bits select the IEEE rounding mode, as shown in Table 3.

Table 3. Rounding Mode

RND1 - RND0	ROUNDING MODES
0 0	Round towards nearest
0 1	Round toward zero (truncated)
1 0	Round towards infinity (round up)
1 1	Round towards negative infinity (round down)

status register definition

The floating-point status register (STATUS) is a 32-bit register used for reporting the exceptions that occur during SMJ34082A operations and status codes set by the results of implicit and explicit compare operations. The status register is cleared upon reset, except for the INTENED flag, which is set to 1 in the coprocessor mode.

Table 4. Status Register Definition

BIT NO.	NAME	DESCRPTION
31	N	Sign bit (A < B flag for compare)
30	GT	A > B (valid on compare)
29	Z	Zero flag (A = B for compare)
28	V	IEEE overflow flag. The result is greater than the largest allowable value for the specified format.
27	I	IEEE invalid operation flag. A NaN has been input to the multiplier or the ALU, or an invalid operation [(0 * 1) or (∞−∞) or (−∞ +∞)] has been requested. This signal also goes high if an operation involves the square root of a negative number. When IVAL hoes high, the STX pins indicate which port had the NaN.
26	U	IEEE underflow flag. The result is inexact and less than the minimum allowable value for the specified format. In fast mode, this condition causes the result to go to zero.
25	X	IEEE inexact flag. The result of an operation is inexact.
24	DIV0	Divide by zero. An invalid operation involving a zero divisor has been detected by the multiplier.
23	RND	The mantissa of a number has been increased in magnitude by rounding. If the number generated was wrapped, then the 'unwrap rounded' instruction must be used to properly unwrap the wrapped number.
22	DENIN	Input to the multiplier is a denormalized number. When DENIN goes high, the STX pins indicate which port has the denormal input.
21	DENORM	The multiplier output is wrapped number or the ALU output is a denormalized number. In fast mode, this condition causes the result to go to zero. It also indicates an invalid integer operation with a negative unsigned integer result.
20	STX1	A NaN or a denormalized number has been input on the A port.
19	STX0	A NaN or a denormalized number has been input on the B port.
18	ED	Exception detect status signal representing logical OR of all enabled exceptions in the configuration register.
17	UNORD	The two inputs of a comparison operation are unordered, i.e.; one or both of the inputs is an NaN.
16	INTFLG	Software interrupt flag. Set by external code to signal a software interrupt.
15	INTENHW	Hardware interrupt ($\overline{\text{INTR}}$) enable, active high (initialized to zero)
14	NXOROV	N (negative) XOR V (overflow)
13	VANDZB	V (overflow) AND $\overline{Z}$ (NOT zero)
12	INTENED	ED interrrupt enable, active high (initialized to zero in the host-independent mode, one in the coprocessor mode)
11	INTENSW	Software interrupt (INTFLG) enable, active high (initialized to zero)
10	ZGT	Zn > Zmax (valid for 2-D MIN-MAX instruction)
9	ZLT	Zn < Zmin (valid for 2-D MIN-MAX instruction)
8	YGT	Yn > Ymax (valid for 1-D or 2-D MIN-MAX instruction)
7	YLT	Yn < Ymin (valid for 1-D or 2-D MIN-MAX instruction)
6	XGT	Xn > Xmax (valid for 1-D or 2-D MIN-MAX instruction)
5	XLT	Xn < Xmin (valid for 1-D or 2-D MIN-MAX instruction)
4	HINT	Hardware interrupt flag
3-0	N/A	Reserved

indirect address register (MCADDR) definition

The indirect address register (MCADDR) can be set to point to a memory location for indirect move or jump operations through the MSD port. MCADDR is cleared upon reset.

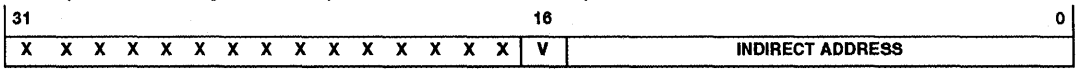


Figure 2. Indirect Address Definition

The function of bit 16 varies, depending on whether the instruction is a MOVE or JUMP. During a MOVE instruction, bit 16 selects data space when set high, or code space when low. During a JUMP instruction, bit 16 selects an internal instruction when set high, or an external instruction when low.

stack registers (SUBADD1-SUBADD0) definition

The stack contains two subroutine return address registers, SUBADD0 and SUBADD1, which serves as a two-deep LIFO (last-in, first-out) stack. A subroutine jump causes the program counter to be pushed onto the stack, and a return from subroutine pops the last address pushed on the stack. More than two pushes will overwrite the contents of SUBADD1.

Bit 31 (Pointer) is set high in the stack location that was written last and reset to zero in the other stack location. Setting bit 30 (Enable) high enables a write into bit 31 (set or reset the pointer) in either stack location. If bit 31 is zero in both SUBADD0 and SUBADD1 (as when the stack has been saved externally and later restored), SUBADD0 can be designated as top of stack by setting bit 31. The stack pointers (bit 31) are cleared upon reset.

Bit 16 (I) is set high when the address in a stack location points to an internal routine, or set low when the address is for an external instruction.

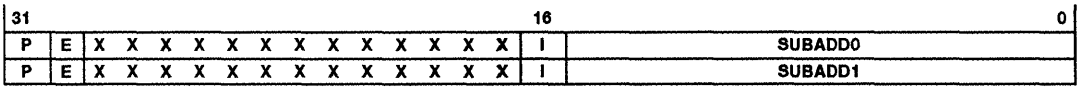


Figure 3. Stack Definition

interrupt vector register (VECTOR) definition

The interrupt vector register (VECTOR) serves as a pointer to an external program to be executed upon receipt of an interrupt. Bit 16 (I) is always set low to point to a routine in external code space. The interrupt vector is cleared on reset.

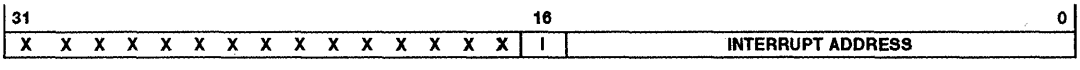


Figure 4. Interrupt Vector Definition

interrupt return register (IRAREG) definition

The interrupt return register (IRAREG) retains a copy of the program counter at the time of an external interrupt. This address is used as the next execution address upon returning from the interrupt. Bit 16 (I) is set high when the address in the stack location points to an internal instruction, or set low when the address is for an external instruction. This register is not affected by the reset signal.

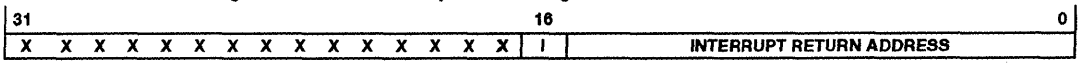


Figure 5. Interrupt Return Definition

COUNTX and COUNTY registers definition

The counter registers (COUNTX, COUNTY) are used to store the current counts of the minimum and maximum values when executing MIN-MAX instructions. COUNTX and COUNTY are cleared on reset.

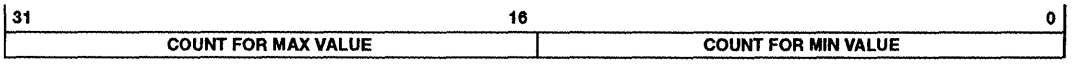


Figure 6. COUNTY and COUNTX Register Definition

The COUNTX register is updated on both the 1-D and 2-D MIN-MAX instruction such that the count of the current minimum value is in the lower 16 bits of the register and the count of the current maximum value is in the upper 16 bits. The COUNTY register is used only in the 2-D MIN-MAX instruction to keep track of the counts of the minimum and maximum for the second value of a pair. The COUNTX and COUNTY registers may also be used for temporary storage when not using the MIN-MAX instructions.

MIN-MAX/LOOPCT register

The MIN-MAX/LOOPCT register stores the current values of two separate counters. The LSH contains the current loop counter, and the MSH is used to hold the current minimum or maximum value of a MIN-MAX operation. The MIN-MAX/LOOPCT register is cleared upon reset. The MIN-MAX/LOOPCT register may also be used for temporary storage when not using the MIN-MAX instructions.

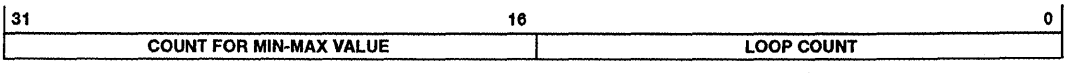


Figure 7. MIN-MAX/LOOPCT Register Definition

FPU core

The FPU core itself consists of a multiplier and an ALU, each with an intermediate pipeline register and an output register (see Figure 8, FPU core functional block diagram). Four multiplexers select the multiplier and ALU operands from the data registers, feedback registers, or previous multiplier or ALU result. Results are directed either to the internal feedback registers (C or CT), the 20 data registers in register files RA and RB, or the ten other miscellaneous registers.

Both the internal pipeline registers and the output registers can be enabled or made transparent (disabled) by setting the PIPES2-PIPES1 bits in the configuration register. When the device is powered up, the default settings of the internal registers are PIPES2 high (output registers transparent) and PIPES1 low (internal pipeline registers enabled).

When the FPU core is used for chained operations, the multiplier and ALU operate in parallel. Two data inputs are provided from the RA and RB input registers, while multiplier and ALU feedback are used as the other two operands. While in the chained mode, the output registers of the FPU must be enabled to latch feedback operands. The appropriate registers must be enabled by setting the PIPES2-PIPES1 controls in the configuration register at the beginning of chained operations, and the PIPES2-PIPES1 control should then be reinitialized upon termination.

Fully pipelined operation (both pipeline and output registers enabled) affects timing when writing results back to the RA and RB register files. To adjust writeback timing, it is possible to issue the NOP (no operation) instruction to the FPU core when the results are to be retained in the output registers for one or more additional cycles. The NOP instruction is only effective when the output registers are enabled, as each NOP causes the output register contents to be retained for one additional cycle.

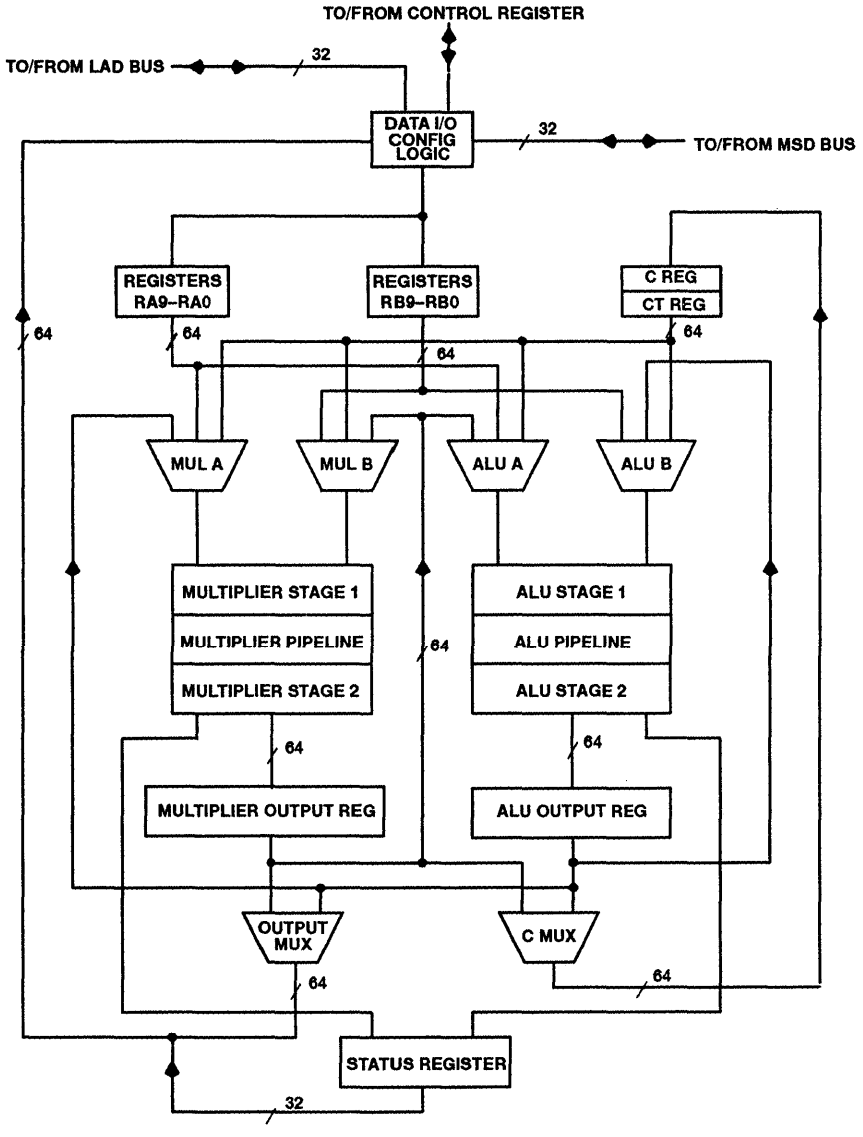


Figure 8. FPU Core Functional Block Diagram

SMJ34082A operating modes

The SMJ34082A can operate as a stand-alone floating-point processor or a graphics coprocessor to the SMJ34020 Graphics System Processor. Control of FPU operation is provided either from external program memory or from the SMJ34020. External instructions are addressed by address lines MSA15-0 and are input on MSD31-0. SMJ34020 instructions are input on LAD31-0.

Both the MSD and LAD buses can be used for data transfers as well. Combinations of control signals distinguish instruction fetches from data transfers. A single instruction may be used to transfer data and to perform an operation within the FPU.

The SMJ34082A supports external code and data storage with the memory expansion interface, MSD31-0. Up to 64K 32-bit data operands and 64K instructions may be added externally to the SMJ34082A. The signal DS/$\overline{CS}$ controls whether data space or code space is being accessed, and read/write control is provided with the chip enable ($\overline{MCE}$), output enable ($\overline{MOE}$), address enable ($\overline{MAE}$), write enable ($\overline{MWR}$), and address lines (MSA15-0).

The SMJ34082A also provides instructions that allow the SMJ34020 to read/write directly from/to external memory. The external code support permits full utilization of the SMJ34082A features and instruction set.

coprocessor-mode operation

Operation in the coprocessor mode assumes MSTR is low. In this mode, the SMJ34082A acts as a closely coupled coprocessor to the SMJ34020. The interface between the two devices consists of direct connections between pins. More than one coprocessor may be connected to the SMJ34020 by setting the appropriate coprocessor ID (CID2-CID0). Up to four coprocessors executing in parallel may be used with a single SMJ34020.

In the coprocessor mode, clock signals are provided by LCLK1 and LCLK2 from the SMJ34020. Internally, the FPU generates a rising clock edge from each LCLK1 edge (rising or falling). Thus, the SMJ34082A actually operates at twice the LCLK1 input clock frequency.

initialization (coprocessor mode)

On reset, the SMJ34082A clears all pipeline registers and internal states. The configuration register and status register return to their initialization values. When $\overline{RESET}$ returns high in the coprocessor mode, the SMJ34082A is in an idle state waiting for the next instruction from the SMJ34020.

LAD bus control (coprocessor mode)

Both data and instructions are transferred over the bidirectional LAD bus in the coprocessor mode. A unique combination of signal inputs distinguishes an instruction from data. SF, $\overline{ALTCH}$, $\overline{CAS}$, $\overline{RAS}$, and $\overline{WE}$ are used to designate coprocessor functions from other operations on the LAD bus.

Data may be transferred to or from SMJ34020 registers or memory via LAD31-0. Transfers between the LAD and MSD buses can also be programmed. A single coprocessor instruction may be used to transfer data to the SMJ34082A and then perform an FPU operation.

MSD bus control (coprocessor mode)

Use of the MSD bus in the coprocessor mode is optional. External memory on MSD31-0 can be used to store data, user-programmed subroutines, or both. Different combinations of control signals distinguish between data memory and code memory. Control signals for MSD and MSA buses operate the same in the host-independent and coprocessor modes.

interrupt handling (coprocessor mode)

A software interrupt to the SMJ34082A is generated by the set mask external instruction. When the interrupt is granted, the current program counter is stored in the interrupt return register, and a branch to the interrupt vector address is executed. Software interrupts may be disabled.

If the exception detect interrupt (ED) is enabled, a SMJ34082A exception causes $\overline{\text{COINT}}$ to go low, signalling the exception to the SMJ34020. This exception does *not* cause a branch to the interrupt vector. If its interrupts are enabled, the SMJ34020 will branch to an interrupt vector to service the SMJ34082A request. Interrupts are cleared by reading the SMJ34082A status register.

host-independent mode operation

Operation in the host-independent mode assumes MSTR high. The SMJ34082A has several hardware control signals, as well as programmable features, which support system functions such as initialization, data transfer, or interrupts in the host-independent mode. CLK provides the input clock to the SMJ34082A. Details of initialization, LAD and MSD bus interface control, and interrupt handling are provided in the following sections.

initialization (host-independent mode)

To simplify initialization of external program memory, the SMJ34082A provides a bootstrap loader to perform an initial program load of 64 instructions. Once invoked, the loader causes the SMJ34082A to read 65 words from the LAD bus and write 64 words out to the external program memory on the MSD bus, beginning with location 0. The first word read is used to initialize the configuration register.

This loader is invoked by first setting $\overline{\text{RESET}}$ low, and then $\overline{\text{INTR}}$ low. A separate timing diagram for using the bootstrap loader is provided (see Figure 34). $\overline{\text{INTR}}$ should be taken low after $\overline{\text{RESET}}$ is already low, as shown in the diagram. When the bootstrap loader is started, the FPU core is reset (internal states and status are cleared, but not data registers) and the stack pointer, program counter, and interrupt vector register are all set to zero.

$\overline{\text{RESET}}$ must be set high again before the loader operation can start (see Figure 34). Once the loader is active, an external interrupt (signalled by $\overline{\text{INTR}}$ low) will not be granted until the load sequence is finished. However, $\overline{\text{RESET}}$ going low terminates the load sequence, regardless of whether the sequence is complete. When the load sequence is finished, the device begins program execution at external address 0.

LAD bus control (host-independent mode)

Data transfer from the LAD bus (LAD31-0) is controlled primarily by output signals, $\overline{\text{ALTCH}}$, $\overline{\text{WE}}$, and $\overline{\text{CAS}}$. $\overline{\text{ALTCH}}$ is the address write strobe that signals an address is being output on the LAD bus. The $\overline{\text{CAS}}$ signal is the read strobe, and $\overline{\text{WE}}$ is the write enable output to memory.

If a bidirectional FIFO is used instead of memory, $\overline{\text{CAS}}$ can be directly connected to the read clock and $\overline{\text{WE}}$ to the write clock. The CC input can be used to signal the SMJ34082A when data is ready for input from the FIFO stack.

Data input on the LAD bus can be written to data registers, control registers, or passed through for output on the MSD bus. Alternatively, the LAD bus input can be selected directly as an FPU source operand without writing to a register.

An FPU result can be written to a data register and at the same time be passed out on the LAD bus. When this is done, the clock period may need to be extended up to 15 ns (SMJ34082-30) to allow for the propagation delay from the FPU core to the outputs.

Depending on the specific system implementation, transferring data to and from the LAD bus without intervening register operations may significantly improve throughput. In the host-independent mode, data moves to and from internal registers can be minimized at the cost of adjusting the clock period to assure integrity of FPU inputs to and output from the LAD bus.

MSD bus control (host-independent mode)

The MSD bus can be used to access either external data memory or external code memory, depending on the combination of control signals required. If the memory on the MSD port is shared with a host processor, the $\overline{\text{MAE}}$ and RDY signals can be used to prevent conflicts between the host and the SMJ34082A. When memory on the MSD port is shared, the host processor can monitor the state of the SMJ34082A memory chip enable ($\overline{\text{MCE}}$) to determine when the SMJ34082A is not accessing the memory.

Otherwise, the $\overline{\text{MAE}}$ signal may be tied low (if unused), and the SMJ34082A can use $\overline{\text{MOE}}$, $\overline{\text{MCE}}$, $\overline{\text{MWR}}$, and DS/$\overline{\text{CS}}$ to control external memory operations into either data space or code space, as selected by DS/$\overline{\text{CS}}$.

interrupt handling (host-independent mode)

Interrupts to the SMJ34082A can be signalled by setting the interrupt request input ($\overline{\text{INTR}}$) low. $\overline{\text{INTR}}$ is associated with the vector in the interrupt vector register. Software interrupts are signalled by setting the software interrupt flag in the status register.

In the event of an FPU status exception in the host-independent mode, an interrupt is generated that causes a branch to an exception handler routine. The address of the exception handler is stored in the interrupt vector register by the user prior to execution of the FPU program. Interrupts may be disabled by setting the appropriate bits in the status register.

absolute maximum ratings over operating free-air temperature range (unless otherwise noted)†

Supply voltage, V_{CC} (see Note 1) ..... 6 V
Input voltage range, V_I ..... −0.3 V to 6 V
Off-state output voltage range ..... −2 V to 6 V
Operating free-air (minimum) and case (maximum) temperature range ..... −55°C to 125°C
Storage temperature range ..... −65°C to 150°C

† Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. These are stress ratings only and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

NOTE 1: All voltage levels are with respect to ground (V_{SS}).

recommended operating conditions

PARAMETER				MIN	NOM	MAX	UNIT
V_{CC}	Supply voltage			4.5	5	5.5	V
V_{SS}	Supply voltage (see Note 2)				0		V
V_{IH}	High-level input voltage			2.4		V_{CC}+0.3	V
V_{IL}	Low-level input voltage			−0.3		0.6	V
I_{OH}	High-level output current					−8	mA
I_{OL}	Low-level output current					8	mA
f_{clock}	Clock frequency	Coprocessor mode	SMJ34082A-28			7.1	MHz
			SMJ34082A-30			7.6	
		Host-independent Mode	SMJ34082A-28			14.3	
			SMJ34082A-30			15.4	
T_A	Operating free-air temperature			−55			°C
T_C	Operating case temperature					125	°C

NOTE 2: In order to minimize noise on V_{SS}, care should be taken to provide a minimum-inductance path between the V_{SS} pins and system ground.

electrical characteristics over recommended operating free-air (minimum) and case (maximum) temperature range (unless otherwise noted)

PARAMETER			TEST CONDITIONS		MIN	TYP‡	MAX	UNIT
V_{OH}	High-level output voltage		V_{CC} = 4.5 V,	I_{OH} = −8 mA	2.6			V
V_{OL}	Low-level output voltage		V_{CC} = 4.5 V,	I_{OL} = 8 mA			0.6	V
I_O	High-impedance bidirectional pins output current		V_{CC} = 4.5 V,	V_O = 2.8 V			10	µA
			V_{CC} = 4.5 V,	V_O = 0.6 V			−10	
I_I	Input current		V_I = V_{SS} to V_{CC}				±10	µA
I_{CC}§	Supply current	Dynamic	V_{CC} = 5.5 V				325	mA
		Quiescent	V_I = V_{IL}max or V_{IH}min,	I_{OH} = I_{OL} = 0			50	mA
			V_I = 0.2 V or V_{CC} − 0.2 V,	I_{OH} = I_{OL} = 0			50	
C_i	Input capacitance					10		pF

‡ All typical values are at V_{CC} = 5 V and T_A = 25°C.

§ I_{CC} is measured at maximum clock frequency. Inputs are presented with random logic highs and lows to assure the toggling of internal nodes.

coprocessor mode (MSTR low)

switching characteristics over recommended ranges of supply voltage and operating free-air (minimum) and case (maximum) temperature range (unless otherwise noted)†

propagation delay times

PARAMETER		FIGURE	SMJ34082A-28		SMJ34082A-30		UNIT
			MIN	MAX	MIN	MAX	
$t_{p(ATCL-CORV)}$	Propagation delay time, $\overline{ALTCH}$ low to CORDY valid	11		40		40	ns
$t_{p(ATCH-LADV)}$	Propagation delay time, $\overline{ALTCH}$ high to LAD data valid	16		35		35	
$t_{p(CASL-LADV)}$	Propagation delay time, $\overline{CAS}$ low to LAD data valid	14		30		25	
$t_{p(CASH-LADZ)}$	Propagation delay time, $\overline{CAS}$ high to LAD disabled	14		30		25	
$t_{p(LC1-DCSL)ML}$	Propagation delay time, LCLK1 ↑ or ↓ to DS/$\overline{CS}$ low with MEMCFG low	17, 21, 23		25		25	
$t_{p(LC1-DCSH)ML}$	Propagation delay time, LCLK1 ↑ or ↓ to DS/$\overline{CS}$ high with MEMCFG low	17, 19, 21, 23, 24, 26		25		25	
$t_{p(LC1-DCSL)MH}$	Propagation delay time, LCLK1 ↑ or ↓ to DS/$\overline{CS}$ low with MEMCFG high	18, 20, 22, 25, 27		30	2	22	
$t_{p(LC1-DCSH)MH}$	Propagation delay time, LCLK1 ↑ or ↓ to DS/$\overline{CS}$ high with MEMCFG high	18, 20, 22, 25, 27		21	2	21	
$t_{p(LC1-MCEL)}$	Propagation delay time, LCK1 ↑ or ↓ to $\overline{MCE}$ low	17-19, 21-27		21	2	21	
$t_{p(LC1-MCEH)ML}$	Propagation delay time, LCLK1 ↑ or ↓ to $\overline{MCE}$ high with MEMCFG low	17, 19, 21, 23		23	2	23	
$t_{p(LC1-MCEH)MH}$	Propagation delay time, LCLK1 ↑ or ↓ to $\overline{MCE}$ high with MEMCFG high	18, 22, 25, 27		15	2	15	
$t_{p(LC1-MOEL)}$	Propagation delay time, LCLK1 ↑ or ↓ to $\overline{MOE}$ low	17, 18, 21-23, 26, 27	10	35	10	35	
$t_{p(LC1-MOEH)}$	Propagation delay time, LCLK1 ↑ or ↓ to $\overline{MOE}$ high	17, 18, 21-23, 26, 27	3	13	3	13	
$t_{p(LC1-MSAV)}$	Propagation delay time, LCLK1 ↑ or ↓ to MSA address valid	17-27		25		25	
$t_{p(LC1-MSDV)}$	Propagation delay time, LCLK1 ↑ or ↓ to MSD data valid	19, 20-22, 24, 25		40		40	
$t_{p(LC1-MWRL)}$	Propagation delay time, LCLK1 ↑ or ↓ to $\overline{MWR}$ low	19-22, 24, 25	10	35	10	35	
$t_{p(LC1-MWRH)}$	Propagation delay time, LCLK1 ↑ or ↓ to $\overline{MWR}$ high	20-22, 24, 25	3	13	3	13	
$t_{p(LC1H-COIL)}$	Propagation delay time, LCLK1 ↑ to $\overline{COINT}$ low	12		23		20	
$t_{p(LC1H-COIH)}$	Propagation delay time, LCLK1 ↑ to $\overline{COINT}$ high	12		23		20	
$t_{p(LC1H-LADV)}$	Propagation delay time, LCLK1 ↑ to LAD data valid	16		28		23	
$t_{p(MSDV-LADV)}$	Propagation delay time, MSD data valid to LAD data valid	26, 27		30		25	
$t_{p(RASH-LADXZ)}$	Propagation delay time, $\overline{RAS}$ high to LAD disabled	16		30		25	

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

coprocessor mode (MSTR low)

switching characteristics over recommended ranges of supply voltage and operating free-air (minimum) and case (maximum) temperature range (unless otherwise noted) (continued)†

enable and disable times

PARAMETER		FIGURE	SMJ34082A-28		SMJ34082A-30		UNIT
			MIN	MAX	MIN	MAX	
$t_{en(LOEL-LADZX)}$	Enable time, $\overline{LOE}$ low to LAD enabled	16	2	17	2	17	ns
$t_{en(MAEL-MSAZX)}$	Enable time, $\overline{MAE}$ low to MSA enabled	21, 22	2	17	2	17	
$t_{en(MAEL-MSDZX)}$	Enable time, $\overline{MAE}$ low to MSD enabled	22	2	17	2	17	
$t_{dis(LOEH-LADXZ)}$	Disable time, $\overline{LOE}$ high to LAD disabled	16	2	17	2	17	ns
$t_{dis(MAEH-MSAXZ)}$	Disable time, $\overline{MAE}$ high to MSA disabled	21, 22	2	17	2	17	
$t_{dis(MAEH-MSDXZ)}$	Disable time, $\overline{MAE}$ high to MSD disabled	21	2	17	2	17	

valid times

PARAMETER		FIGURE	SMJ34082A-28		SMJ34082A-30		UNIT
			MIN	MAX	MIN	MAX	
$t_{v(MWRH-MSA)}$	Valid time, MSA address after $\overline{MWR}$ high	20-22, 24, 25	0		0		ns
$t_{v(MWRH-MSD)}$	Valid time, MSD data output after $\overline{MWR}$ high	20-22, 24, 25	0		0		
$t_{v(LC1-MSA)}$	Valid time, MSA address valid after LCK ↑ or ↓	17-22, 24-27	3		3		
$t_{v(LC1L-COR)}$	Valid time, CORDY valid after LCLK1 low	11	0		0		

timing requirements over recommended ranges of supply voltage and operating free-air (minimum) and case (maximum) temperature range (unless otherwise noted)†

clock period and pulse duration

PARAMETER		FIGURE	SMJ34082A-28		SMJ34082A-30		UNIT
			MIN	MAX	MIN	MAX	
$t_{c(LC1)}$	Clock period, LCLK1 ($1/f_{clock}$) with PIPES1 low	10, 17-22, 24-27	170		162		ns
$t_{c(LC2)}$	Clock period, LCLK2 ($1/f_{clock}$) with PIPES1 low	10	170		162		
$t_{w(LC1H)}$	Pulse duration, LCLK1 high	10	76		72		ns
$t_{w(LC1L)}$	Pulse duration, LCLK1 low	10	76		72		
$t_{w(LC2H)}$	Pulse duration, LCLK2 high	10	76		72		
$t_{w(LC2L)}$	Pulse duration, LCLK2 low	10	76		72		
$t_{w(DCSH)MH}$	Pulse duration, DS/$\overline{CS}$ high with MEMCFG high	20, 25, 27	5		5		
$t_{w(RSTL)}$	Pulse duration, $\overline{RESET}$ low	12	35		30		
$t_{w(MCEH)}$	Pulse duration, $\overline{MCE}$ high	18, 25, 27	5		5		
$t_{w(MOEH)}$	Pulse duration, $\overline{MOE}$ high	17, 18, 23, 26, 27	5		5		
$t_{w(MWRH)}$	Pulse duration, $\overline{MWR}$ high	20, 24, 25	5		5		

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

coprocessor mode (MSTR low)

timing requirements over recommended ranges of supply voltage and operating free-air (minimum) and case (maximum) temperature range (unless otherwise noted) (continued)†

transition times

PARAMETER		FIGURE	SMJ34082A-28		SMJ34082A-30		UNIT
			MIN	MAX	MIN	MAX	
$t_{t(LC1)}$	Transition time, LCLK1	10		15		15	ns
$t_{t(LC2)}$	Transition time, LCLK2	10		15		15	

setup and hold times

PARAMETER		FIGURE	SMJ34082A-28		SMJ34082A-30		UNIT
			MIN	MAX	MIN	MAX	
$t_{su(BUS-LC2H)}$	Setup time, BUSFLT valid before LCLK2 ↑	11	20		13		ns
$t_{su(CC-LC1)}$	Setup time, CC valid before LCLK1 ↑ or ↓	12	7		7		
$t_{su(LAD-ATCL)}$	Setup time, LAD address valid before $\overline{ALTCH}$ low	13-16, 23	17		17		
$t_{su(LAD-CASH)}$	Setup time, LAD address valid before $\overline{CAS}$ high	13, 15, 24, 25	15		15		
$t_{su(LRD-LC2H)}$	Setup time, LRDY valid before LCLK2 ↑	11	20		20		
$t_{su(MSD-LC1)}$	Setup time, MSD data valid before LCLK1 ↑ or ↓	17, 18, 23	12		12		
$t_{su(RASH-ATCL)}$	Setup time, $\overline{RAS}$ high before $\overline{ALTCH}$ low	13-15, 23	35		30		
$t_{su(RDYL-LC1)}$	Setup time, RDY low before LCLK1 ↑ or ↓	12	20		15		
$t_{su(RSTH-LC1)}$	Setup time, $\overline{RESET}$ high before LCLK1 ↑ or ↓	12	50		50		
$t_{su(SF-ATCL)}$	Setup time, SF valid before $\overline{ALTCH}$ low	13-16, 23	15		15		
$t_{su(WEL-CASL)}$	Setup time, $\overline{WE}$ low for data write before $\overline{CAS}$ low	13, 16	15		15		
$t_{h(ATCH-SF)}$	Hold time, SF valid after $\overline{ALTCH}$ high	13-15, 23	15		12		ns
$t_{h(ATCL-LAD)}$	Hold time, LAD address valid after $\overline{ALTCH}$ low	13-16, 23	21		17		
$t_{h(CASH-LAD)}$	Hold time, LAD data valid after $\overline{CAS}$ high	13, 15, 24, 25	0		0		
$t_{h(CASH-SF)}$	Hold time, SF valid after $\overline{CAS}$ high	13-15, 23	15		15		
$t_{h(LC1-CC)}$	Hold time, CC valid after LCLK1 ↑ or ↓	12	5		5		
$t_{h(LC1-MSD)}$	Hold time, MSD input data valid after LCLK1 ↑ or ↓	17, 18, 23	4		4		
$t_{h(LC1-RDY)}$	Hold time, RDY valid after LCLK1 ↑ or ↓	12	5		5		
$t_{h(LC1H-LC2L)}$	Hold time, LCLK2 low after LCLK1 high	10	20		20		
$t_{h(LC2H-BUS)}$	Hold time, BUSFLT valid after LCLK2 high	11	5		5		
$t_{h(LC2H-LC1H)}$	Hold time, LCLK1 high after LCLK2 high	10	20		20		
$t_{h(LC2H-LRD)}$	Hold time, LRDY valid after LCLK2 high	11	5		5		
$t_{h(WEH-SF)}$	Hold time, SF valid after $\overline{WE}$ high	13	20		20		

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

coprocessor mode (MSTR low)

timing requirements over recommended ranges of supply voltage and operating free-air (minimum) and case (maximum) temperature range (unless otherwise noted) (continued)†

delay times

PARAMETER		FIGURE	SMJ34082A-28		SMJ34082A-30		UNIT
			MIN	MAX	MIN	MAX	
$t_{d(DCSH-MCEL)MH}$	Delay time, DS/$\overline{CS}$ high to $\overline{MCE}$ low with MEMCFG high	18, 22	4		4		ns
$t_{d(DCSH-MWRL)}$	Delay time, DS/$\overline{CS}$ high to $\overline{MWR}$ low	19, 24	5		5		
$t_{d(MCEH-DCSL)MH}$	Delay time, $\overline{MCE}$ high to DS/$\overline{CS}$ low with MEMCFG high	20	4		4		
$t_{d(MCEH-MWRL)}$	Delay time, $\overline{MCE}$ high to $\overline{MWR}$ low	25	5		5		
$t_{d(MOEH-MWRL)}$	Delay time, $\overline{MOE}$ high to $\overline{MWR}$ low	19	5		5		
$t_{d(MSAV-MWRL)}$	Delay time, MSA valid to $\overline{MWR}$ low	20-22, 24, 25	4		4		
$t_{d(MSDZ-MOEL)}$	Delay time, MSD disabled to $\overline{MOE}$ low	21, 22	2		2		
$t_{d(MWRH-MCEL)MH}$	Delay time, $\overline{MWR}$ high to $\overline{MCE}$ low with MEMCFG high	25	5		5		
$t_{d(MWRH-MOEL)}$	Delay time, $\overline{MWR}$ high to $\overline{MOE}$ low	19, 21, 22	5		5		
$t_{d(MWRH-MSDVZ)}$	Delay time, $\overline{MWR}$ high to MSD disabled	21	1	12	1	9	
$t_{d(MWRL-MSDZX)}$	Delay time, $\overline{MWR}$ low to MSD enabled	21, 22	1	13	1	13	

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

host-independent mode (MSTR high)

switching characteristics over recommended ranges of supply voltage and operating free-air (minimum) and case (maximum) temperature range (unless otherwise noted)†

propagation delay times

PARAMETER		FIGURE	SMJ34082A-28		SMJ34082A-30		UNIT
			MIN	MAX	MIN	MAX	
$t_{p(CLKH-ATCH)}$	Propagation delay time, CLK ↑ to $\overline{ALTCH}$ high	29, 30		10		10	ns
$t_{p(CLKH-ATCL)}$	Propagation delay time, CLK ↑ to $\overline{ALTCH}$ low	29, 30		28		28	
$t_{p(CLKH-CASH)}$	Propagation delay time, CLK ↑ to $\overline{CAS}$ high	29, 31, 32, 34-36		10		10	
$t_{p(CLKH-CASL)}$	Propagation delay time, CLK ↑ to $\overline{CAS}$ low	29, 31, 32, 34-36		28		28	
$t_{p(CLKH-COIH)}$	Propagation delay time, CLK ↑ to $\overline{COINT}$ high	29-31, 33, 35, 36, 46		20		20	
$t_{p(CLKH-COIL)}$	Propagation delay time, CLK ↑ to $\overline{COINT}$ low	29-31, 33, 35, 36, 46		20		20	
$t_{p(CLKH-CORH)}$	Propagation delay time, CLK ↑ to CORDY high	46		20		17	
$t_{p(CLKH-CORL)}$	Propagation delay time, CLK ↑ to CORDY low	46		20		17	
$t_{p(CLKH-DCSH)MH}$	Propagation delay time, CLK ↑ to DS/$\overline{CS}$ high with MEMCFG high	36, 38, 40, 42-44	1	10	1	10	
$t_{p(CLKH-DCSH)ML}$	Propagation delay time, CLK ↑ to DS/$\overline{CS}$ high with MEMCFG low	35, 37, 39, 41, 45, 46		23		20	
$t_{p(CLKH-DCSL)MH}$	Propagation delay time, CLK ↑ to DS/$\overline{CS}$ low with MEMCFG high	36, 38, 40, 42-44	1	23	1	20	
$t_{p(CLKH-DCSL)ML}$	Propagation delay time, CLK ↑ to DS/$\overline{CS}$ low with MEMCFG low	37, 41, 45-47		23		20	
$t_{p(CLKH-ITGH)}$	Propagation delay time, CLK ↑ to INTG high‡	47		20		15	
$t_{p(CLKH-ITGL)}$	Propagation delay time, CLK ↑ to INTG low	47		25		20	
$t_{p(CLKH-LADV)}$	Propagation delay time, CLK ↑ to LAD valid	29, 30, 33-35, 43, 44		35		35	
$t_{p(CLKH-MCEH)MH}$	Propagation delay time, CLK ↑ to $\overline{MCE}$ high with MEMCFG high	36, 38, 42-46	1	10	1	10	
$t_{p(CLKH-MCEH)ML}$	Propagation delay time, CLK ↑ to $\overline{MCE}$ high with MEMCFG low	37, 39, 41, 45-47	1	20	1	20	
$t_{p(CLKH-MCEL)}$	Propagation delay time, CLK ↑ to $\overline{MCE}$ low	35-39, 41-47	1	23	1	20	
$t_{p(CLKH-MOEH)}$	Propagation delay time, CLK ↑ to $\overline{MOE}$ high	37, 38, 41-47	1	11	1	11	
$t_{p(CLKH-MOEL)}$	Propagation delay time, CLK ↑ to $\overline{MOE}$ low	37, 38, 41-47	10	35	10	35	
$t_{p(CLKH-MSAV)}$	Propagation delay time, CLK ↑ to MSA address valid	35-47		20		20	
$t_{p(CLKH-MSDV)}$	Propagation delay time, CLK ↑ to MSD data valid	35, 36, 39-42		40		40	
$t_{p(CLKH-MWRH)}$	Propagation delay time, CLK ↑ to $\overline{MWR}$ high	35, 36, 40-42	1	10	1	10	
$t_{p(CLKH-MWRL)}$	Propagation delay time, CLK ↑ to $\overline{MWR}$ low	35, 36, 39-42	10	35	10	35	

† *See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for* PIPES2 high and PIPES1 low. No other pipeline settings are specified.

‡ Interrupts are not granted during multicycle instructions.

host-independent mode (MSTR high)

switching characteristics over recommended ranges of supply voltage and operating free-air (minimum) and case (maximum) temperature range (unless otherwise noted) (continued)†

propagation delay times (continued)

PARAMETER		FIGURE	SMJ34082A-28		SMJ34082A-30		UNIT
			MIN	MAX	MIN	MAX	
$t_{p(CLKH-WEH)}$	Propagation delay time, CLK ↑ to $\overline{WE}$ high	30, 33, 43, 44		10		10	ns
$t_{p(CLKH-WEL)}$	Propagation delay time, CLK ↑ to $\overline{WE}$ low	30, 33, 43, 44		30		30	

enable and disable times

PARAMETER		FIGURE	SMJ34082A-28		SMJ34082A-30		UNIT
			MIN	MAX	MIN	MAX	
$t_{en(CLKH-LADZX)}$	Enable time, CLK high to LAD enabled	29, 30	5		5		ns
$t_{en(LOEL-LADZX)}$	Enable time, $\overline{LOE}$ low to LAD enabled	33	2	17	2	17	
$t_{en(MAEL-MSAZX)}$	Enable time, $\overline{MAE}$ low to MSA enabled	41, 42	2	17	2	17	
$t_{en(MAEL-MSDZX)}$	Enable time, $\overline{MAE}$ low to MSD enabled	42	2	17	2	17	
$t_{dis(CLKH-LADZX)}$	Disable time, CLK high to LAD disabled‡	29, 30		25		25	
$t_{dis(LOEH-LADXZ)}$	Disable time, $\overline{LOE}$ high to LAD disabled	33	2	17	2	17	ns
$t_{dis(MAEH-MSAXZ)}$	Disable time, $\overline{MAE}$ high to MSA disabled	41, 42	2	17	2	17	
$t_{dis(MAEH-MSDXZ)}$	Disable time, $\overline{MAE}$ high to MSD disabled	42	2	17	2	17	

valid times

PARAMETER		FIGURE	SMJ34082A-28		SMJ34082A-30		UNIT
			MIN	MAX	MIN	MAX	
$t_{v(ATCH-LAD)}$	Valid time, LAD output data after $\overline{ALTCH}$ high	29, 30	2		2		ns
$t_{v(CLKH-MSA)}$	Valid time, MSA address valid after CLK high	35-47	3		3		
$t_{v(MWRH-MSD)}$	Valid time, MSD data valid after $\overline{MWR}$ high	35, 36, 40-42	1		1		
$t_{v(MWRH-MSA)}$	Valid time, MSA address valid after $\overline{MWR}$ high	35, 36, 40-41	1		1		
$t_{v(WEH-LAD)}$	Valid time, LAD data valid after $\overline{WE}$	30, 33, 43, 44	2		2		

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

‡ Valid only for last write in series. The LAD bus is not placed in high-impedance state between consecutive outputs.

host-independent mode (MSTR high)

timing requirements over recommended ranges of supply voltage and operating free-air (minimum) and case (maximum) temperature range (unless otherwise noted)†

clock period and pulse duration

PARAMETER		FIGURE	SMJ34082A-28		SMJ34082A-30		UNIT
			MIN	MAX	MIN	MAX	
$t_{c(CLK)}$	Clock period time, CLK ($1/f_{clock}$) with PIPES1 low	28-31, 33-48	78		73		ns
$t_{w(ATCH)}$	Pulse duration, $\overline{ALTCH}$ high	30	5		5		ns
$t_{w(CASH)}$	Pulse duration, $\overline{CAS}$ high	29, 31, 32, 35, 36	5		5		
$t_{w(CLKH)}$	Pulse duration, CLK high	28	17		17		
$t_{w(CLKL)}$	Pulse duration, CLK low	28	22		22		
$t_{w(DCSH)}$	Pulse duration, DS/$\overline{CS}$ high	36, 40, 44	5		5		
$t_{w(ITRL)}$	Pulse duration, $\overline{INTR}$ low	34, 47	30		30		
$t_{w(MCEH)}$	Pulse duration, $\overline{MCE}$ high	36, 38, 44-46	5		5		
$t_{w(MOEH)}$	Pulse duration, $\overline{MOE}$ high	37, 38, 43-46	6		6		
$t_{w(MWRH)}$	Pulse duration, $\overline{MWR}$ high	35, 36, 40	6		6		
$t_{w(RSTL)}$	Pulse duration, $\overline{RESET}$ low	34	40		40		
$t_{w(WEH)}$	Pulse duration, $\overline{WE}$ high	30, 33, 43, 44	5		5		

transition time

PARAMETER		FIGURE	SMJ34082A-28		SMJ34082A-30		UNIT
			MIN	MAX	MIN	MAX	
$t_{t(CLK)}$	Transition time, CLK	28		15		15	ns

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

host-independent mode (MSTR high)

timing requirements over recommended ranges of supply voltage and operating free-air (minimum) and case (maximum) temperature range (unless otherwise noted) (continued)†

setup and hold times

PARAMETER		FIGURE	SMJ34082A-28		SMJ34082A-30		UNIT
			MIN	MAX	MIN	MAX	
$t_{su(CC-CLKH)}$	Setup time, CC before CLK high	45	7		7		ns
$t_{su(LADV-CLKL)}$	Setup time, LAD data valid before CLK low for immediate data input‡	32	15		15		
$t_{su(ITRL-CLKH)}$	Setup time, $\overline{INTR}$ before CLK high	47	20		15		
$t_{su(LAD-CLKH)}$	Setup time, LAD input data valid before CLK high	29, 31, 34-36	15		13		
$t_{su(LRD-CLKH)}$	Setup time, LRDY before CLK high	48	20		15		
$t_{su(MSD-CLKH)}$	Setup time, MSD data valid before CLK high	37, 38, 43-47	13		13		
$t_{su(RDYV-CLKH)}$	Setup time, RDY valid before CLK high	48	20		12		
$t_{su(RSTH-CLKH)}$	Setup time, $\overline{RESET}$ high before CLK high	34	45		45		
$t_{su(RSTL-ITRL)}$	Setup time, $\overline{RESET}$ low before $\overline{INTR}$ low for bootstrap loader	34	20		20		ns
$t_{h(CLKH-CC)}$	Hold time, CC after CLK high	45	3		3		
$t_{h(CLKH-ITR)}$	Hold time, $\overline{INTR}$ after CLK high	47	3		3		
$t_{h(CLKH-LAD)}$	Hold time, LAD input data valid after CLK high	29, 31, 35, 36	5		5		
$t_{h(CLKH-LRD)}$	Hold time, LRDY after CLK high	48	0		0		
$t_{h(CLKH-MSD)}$	Hold time, MSD input data valid after CLK high	37, 38, 43-47	4		4		
$t_{h(CLKH-RDY)}$	Hold time, RDY after CLK high	48	0		0		
$t_{h(CLKL-LAD)}$	Hold time, LAD data after CLK low for immediate data input‡	32	5		5		
$t_{h(ITRL-RSTH)}$	Hold time, $\overline{RESET}$ low after $\overline{INTR}$ low for bootstrap loader	34	15		15		

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

‡ This mode permits data input that does not meet the minimum setup before CLK high. The clock period for this mode must be extended according to the equation:

$$Adjusted\ clock\ period = Normal\ clock\ period + Data\ delay + 5\ ns$$

The data delay is the delay from CLK high to valid data. This mode may not be used to input data for divides or square roots.

host-independent mode (MSTR high)

timing requirements over recommended ranges of supply voltage and operating free-air (minimum) and case (maximum) temperature range (unless otherwise noted) (continued)†

delay times

PARAMETER		FIGURE	SMJ34082A-26		SMJ34082A-30		UNIT
			MIN	MAX	MIN	MAX	
$t_{d(ATCH-CASL)}$	Delay time, $\overline{ALTCH}$ high to $\overline{CAS}$ low	29	5		5		ns
$t_{d(ATCH-WEL)}$	Delay time, $\overline{ALTCH}$ high to $\overline{WE}$ low	30	3		3		
$t_{d(CASH-ATCL)}$	Delay time, $\overline{CAS}$ high to $\overline{ALTCH}$ low	29	3		3		
$t_{d(CASH-WEL)}$	Delay time, $\overline{CAS}$ high to $\overline{WE}$ low	33	3		3		
$t_{d(COIL-ATCL)}$	Delay time, $\overline{COINT}$ low to $\overline{ALTCH}$ low	29, 30	0		0		
$t_{d(COIL-CASL)}$	Delay time, $\overline{COINT}$ low to $\overline{CAS}$ low	31, 35, 36	0		0		
$t_{d(COIL-WEL)}$	Delay time, $\overline{COINT}$ low to $\overline{WE}$ low	33	0		0		
$t_{d(DCSH-MCEL)MH}$	Delay time, DS/$\overline{CS}$ high to $\overline{MCE}$ low with MEMCFG high	38, 42	5		5		
$t_{d(DCSH-MWRL)}$	Delay time, DS/$\overline{CS}$ high to $\overline{MWR}$ low	35, 39	4		4		
$t_{d(MCEH-DCSL)MH}$	Delay time, $\overline{MCE}$ high to DC/$\overline{CS}$ low with MEMCFG high	40	5		5		
$t_{d(MCEH-MWRL)}$	Delay time, $\overline{MCE}$ high to $\overline{MWR}$ low	36	5		5		
$t_{d(MOEH-MWRL)}$	Delay time, $\overline{MOE}$ high to $\overline{MWR}$ low	39	5		5		
$t_{d(MSAV-MWRL)}$	Delay time, MSA valid to $\overline{MWR}$ low	35, 36, 40-42	4		4		
$t_{d(MSDZ-MOEL)}$	Delay time, MSD disabled to $\overline{MOE}$ low	41, 42	2		2		
$t_{d(MWRH-MCEL)MH}$	Delay time, $\overline{MWR}$ high to $\overline{MCE}$ low with MEMCFG high	36	5		5		
$t_{d(MWRH-MOEL)}$	Delay time, $\overline{MWR}$ high to $\overline{MOE}$ low	41, 42	5		5		
$t_{d(MWRH-MSDXZ)}$	Delay time, $\overline{MWR}$ high to MSD disabled	42	1	12	1	9	
$t_{d(MWRL-MSDZX)}$	Delay time, $\overline{MWR}$ low to MSD enabled	41, 42	1	13	1	13	
$t_{d(WEH-ATCL)}$	Delay time, $\overline{WE}$ high to $\overline{ALTCH}$ low	29	3		3		
$t_{d(WEH-CASL)}$	Delay time, $\overline{WE}$ high to $\overline{CAS}$ low	31	3		3		

† See Parameter Measurement Information for load circuit, voltage waveforms, and timing diagrams. The device parameters are measured for PIPES2 high and PIPES1 low. No other pipeline settings are specified.

EXPLANATION OF LETTER SYMBOLS

This data sheet uses a type of letter symbol based on JEDEC Std-100 and IEC Publication 748-2, 1985, to describe time intervals. The format is:

$$t_{A(BC\text{-}DE)F}$$

Where:

Subscript A indicates the type of dynamic parameter being represented. One of the following is used:

Switching Characteristics:
- p = Propagation delay time
- en = Enable time
- dis = Disable time

Timing Requirements:
- c = Clock period
- w = Pulse duration
- t = Transition time
- d = Delay time
- su = Setup time
- h = Hold time
- v = Valid time

Subscript B indicates the name of the signal or terminal for which a change of state or level (or establishment of a state or level) constitutes a signal event assumed to occur first, that is, at the beginning of the time interval.

Subscript C indicates the direction of the transistion and/or the final state or level of the signal represented by B. One or two of the following are used:

- H = High or transition to high
- L = Low or transition to low
- V = A valid steady-state level
- X = Unknown, changing, or "don't care" level
- Z = High-impedance (off) state

Subscript D indicates the name of the signal or terminal for which a change of state or level (or establishment of a state or level) constitutes a signal event assumed to occur last, that is, at the end of the time interval.

Subscript E indicates the direction of the transition and/or the final state or level of the signal represented by D. One or two of the symbols described in *Subscript C* are used.

Subscript F indicates additional information such as mode of operation, test conditions, etc.

The hyphen between the C and D subscripts is omitted when no confusion is likely to occur. For these letter symbols on this data sheet, the signal names are further abbreviated as follows:

SIGNAL NAME	B & D SUBSCRIPT	SIGNAL NAME	B & D SUBSCRIPT	SIGNAL NAME	B & D SUBSCRIPT	SIGNAL NAME	B & D SUBSCRIPT	SIGNAL NAME	B & D SUBSCRIPT
$\overline{\text{ALTCH}}$	ATC	CORDY	COR	LCLK2	LC2	MSA(0:15)	MSA	TCK	TCK
BUSFLT	BFT	DC/$\overline{\text{CS}}$	DCS	$\overline{\text{LOE}}$	LOE	MSD(0:31)	MSD	TDI	TDI
$\overline{\text{CAS}}$	CAS	EC(0:1)	EC	LRDY	LRD	$\overline{\text{MWR}}$	MWR	TDO	TDO
CC	CC	INTG	INT	$\overline{\text{MAE}}$	MAE	$\overline{\text{RAS}}$	RAS	TMS	TMS
CID(0:2)	CID	$\overline{\text{INTR}}$	ITR	MSTR	MST	RDY	RDY	V_{CC}/V_{SS}	—
CLK	CLK	LAD(0:31)	LAD	$\overline{\text{MCE}}$	MCE	$\overline{\text{RESET}}$	RST	$\overline{\text{WE}}$	WE
$\overline{\text{COINT}}$	COI	LCLK1	LC1	$\overline{\text{MOE}}$	MOE	SF	SF	MEMCFG	M

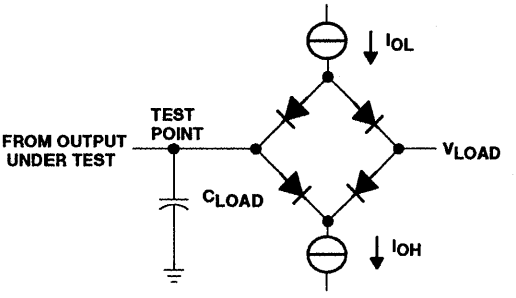
PARAMETER MEASUREMENT INFORMATION

LOAD CIRCUIT PARAMETERS

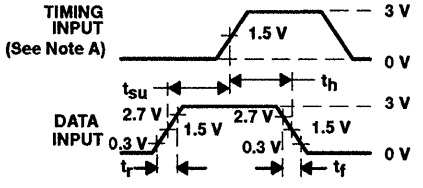
TIMING PARAMETERS		C_{LOAD}† (pF)	I_{OL} (mA)	I_{OH} (mA)	V_{LOAD} (V)
t_{en}	t_{PZH}	65	8	−8	0
	t_{PZL}				3
t_{dis}	t_{PHZ}	65	8	−8	1.5
	t_{PLZ}				
t_p		65	8	−8	‡

† C_{LOAD} includes the typical load circuit and distributed capacitance.

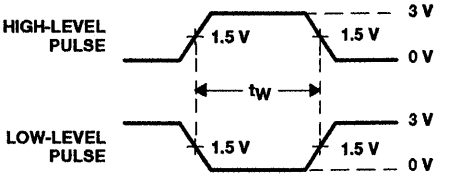
‡ $\frac{V_{LOAD} - V_{OL}}{I_{OL}}$ = 50 Ω, where V_{OL} = 0.6 V, I_{OL} = 8 mA.



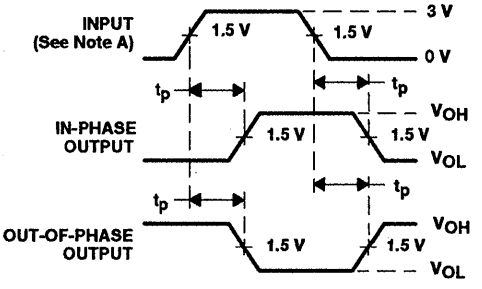
LOAD CIRCUIT



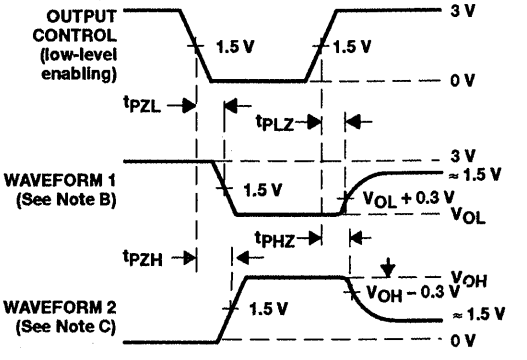
VOLTAGE WAVEFORMS
SETUP AND HOLD TIMES
INPUT RISE AND FALL TIMES



VOLTAGE WAVEFORMS
PULSE DURATION



VOLTAGE WAVEFORMS
PROPAGATION DELAY TIMES



VOLTAGE WAVEFORMS
ENABLE AND DISABLE TIMES, 3-STATE OUTPUTS

NOTES: A. Phase relationships between waveforms were chosen arbitrarily. All input pulses are supplied by pulse generators having the following characteristics: PRR = 1 MHz, Z_o = 50 Ω, $t_r \le 6$ ns, $t_f \le 6$ ns.

B. Waveform 1 is for an output with internal conditions such that the output is low except when disabled by the output control.

C. Waveform 2 is for an output with internal conditions such that the output is high except when disabled by the output control. For t_{PLZ} and t_{PHZ}, V_{OL} and V_{OH} are measured values.

Figure 9

PARAMETER MEASUREMENT INFORMATION

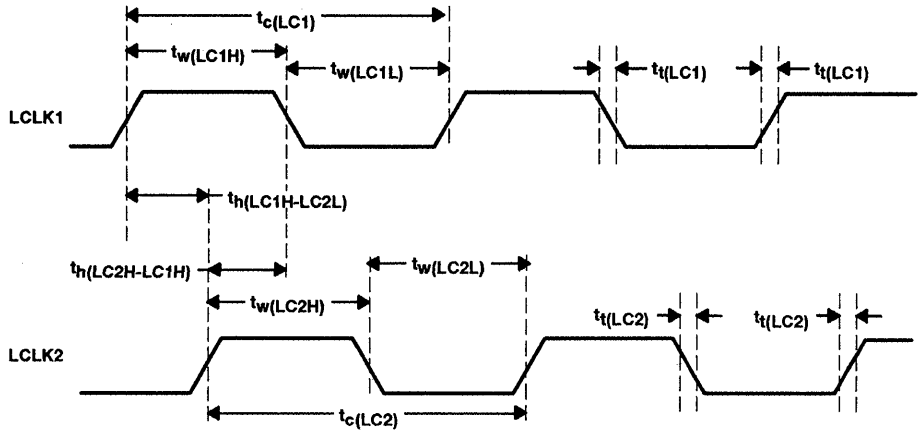
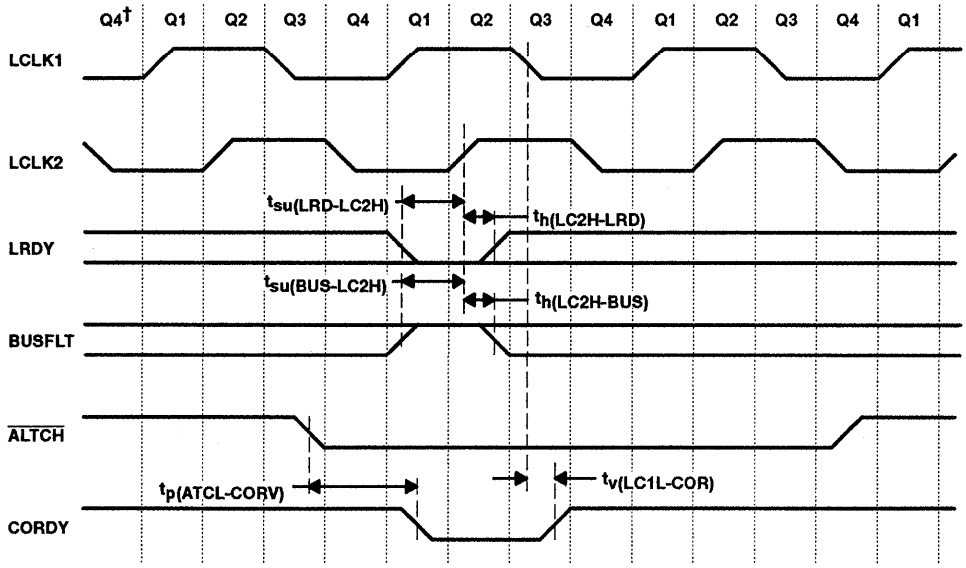


Figure 10. Coprocessor Mode, Input Clocks



† Q1, Q2, Q3, and Q4 represent the first, second, third, and fourth quarter clocks, respectively, of the LCLK1 clock period.

Figure 11. Coprocessor Mode, Bus Control Signals

PARAMETER MEASUREMENT INFORMATION

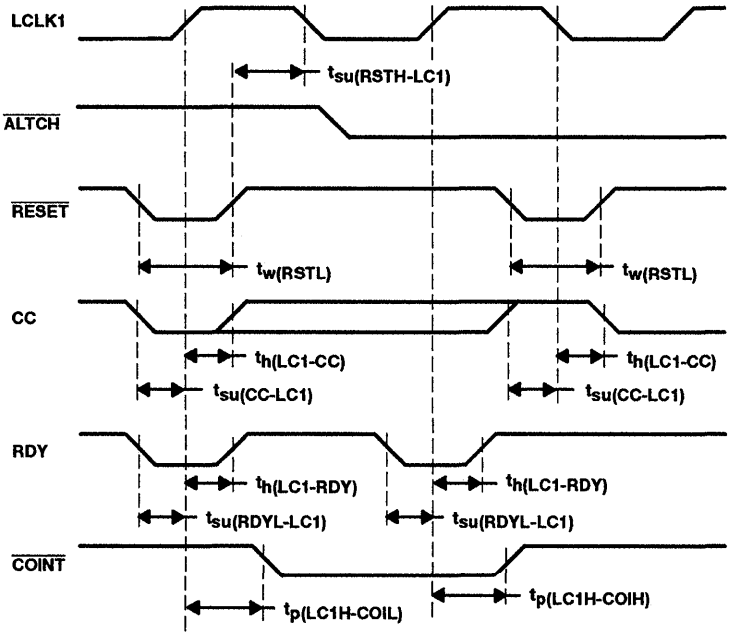
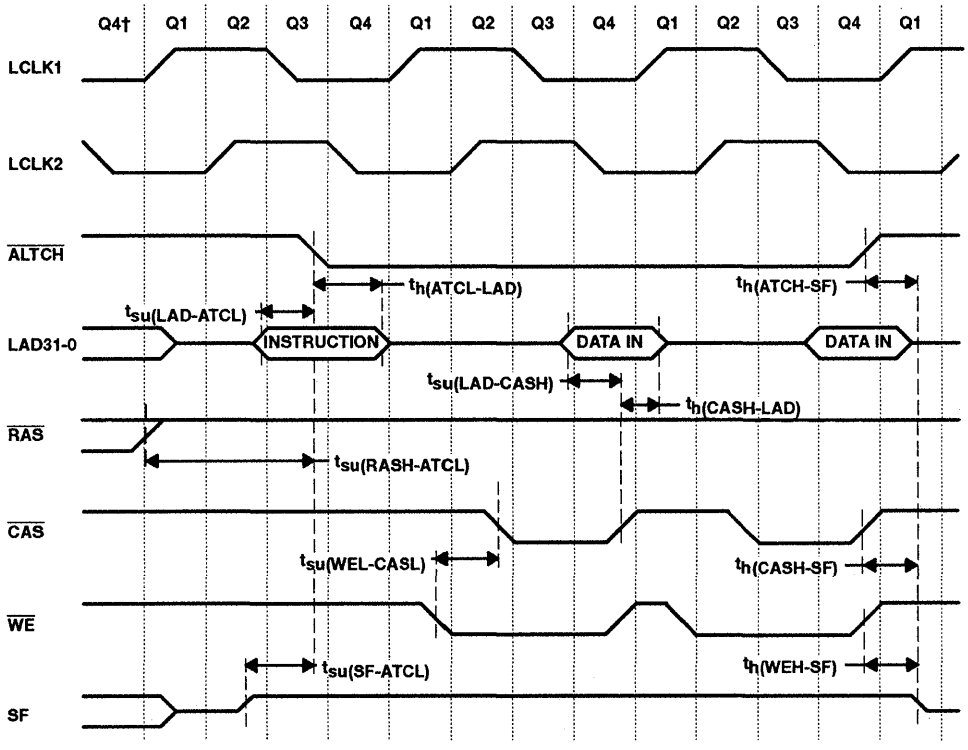


Figure 12. Coprocessor Mode, Control Signals

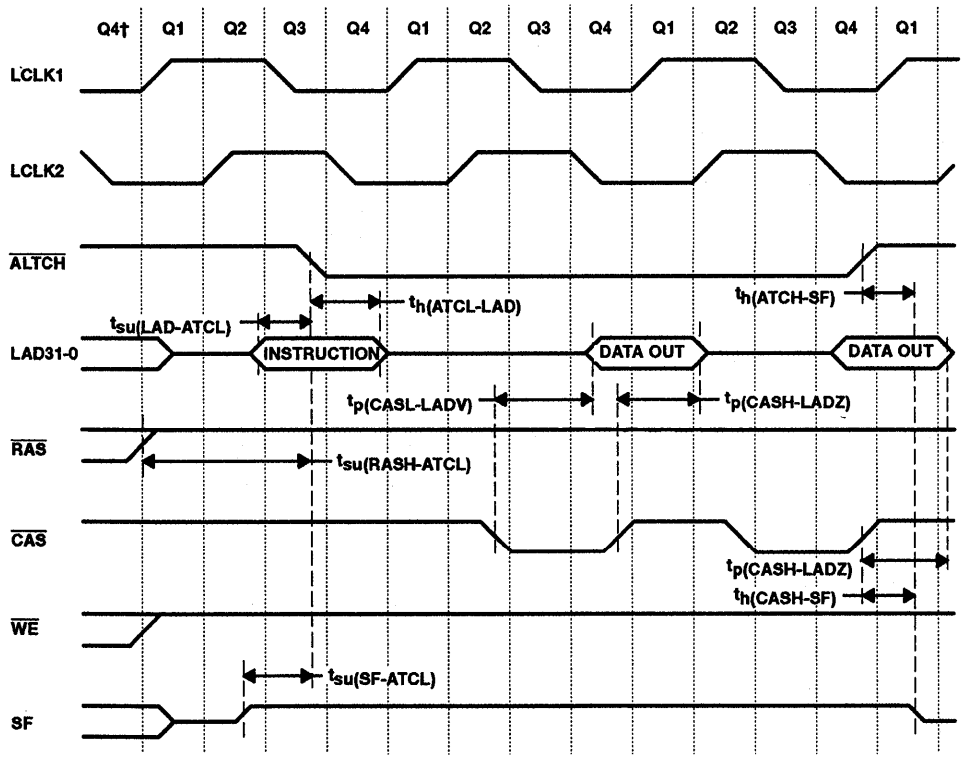
PARAMETER MEASUREMENT INFORMATION



† Q1, Q2, Q3, and Q4 represent the first, second, third, and fourth quarter clocks, respectively, of the LCLK1 clock period.

Figure 13. Coprocessor Mode, SMJ34020 GSP to SMJ34082

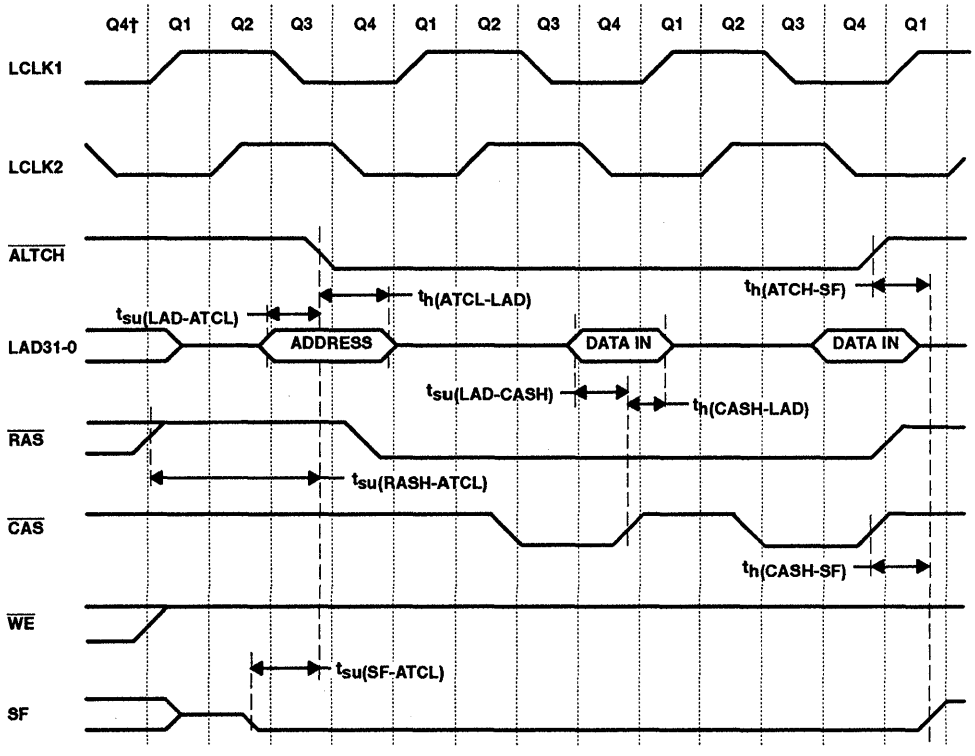
PARAMETER MEASUREMENT INFORMATION



† Q1, Q2, Q3, and Q4 represent the first, second, third, and fourth quarter clocks, respectively, of the LCLK1 clock period.

Figure 14. Coprocessor Mode, SMJ34082A to SMJ34020 GSP Including Coprocessor Internal Cycle

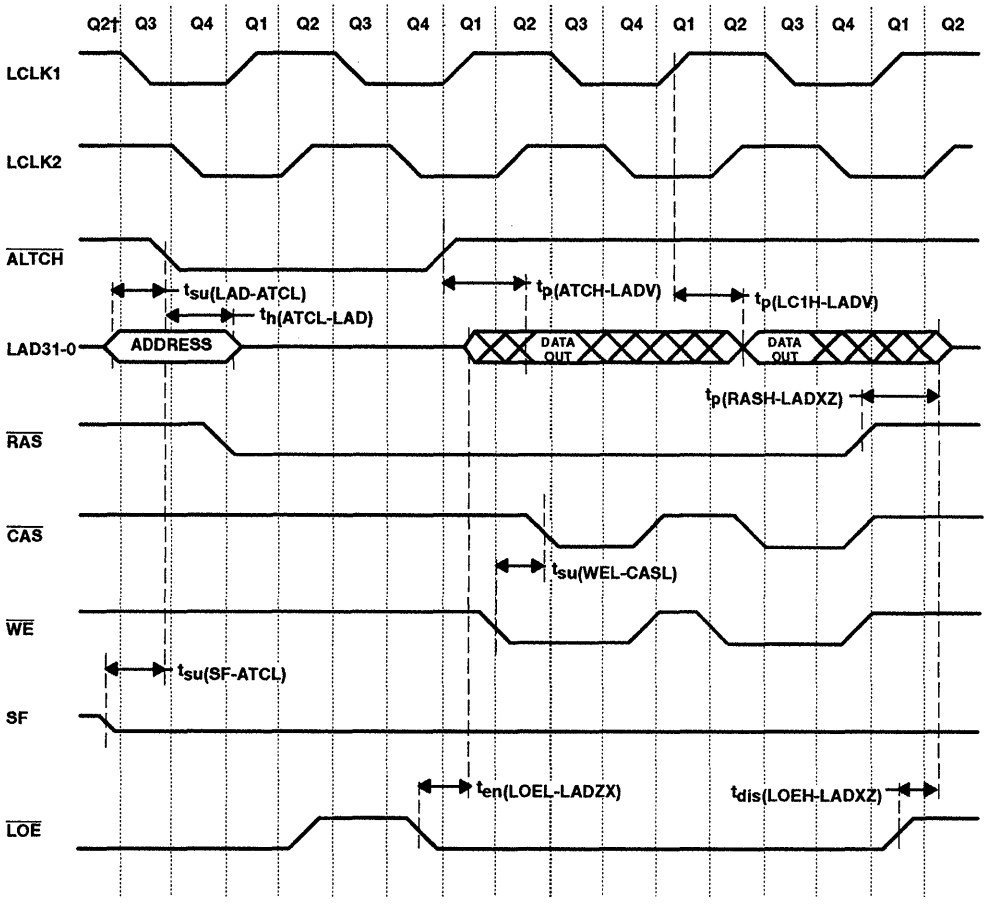
PARAMETER MEASUREMENT INFORMATION



† Q1, Q2, Q3, and Q4 represent the first, second, third, and fourth quarter clocks, respectively, of the LCLK1 clock period.

Figure 15. Coprocessor Mode, DRAM/VRAM Memory to SMJ34082

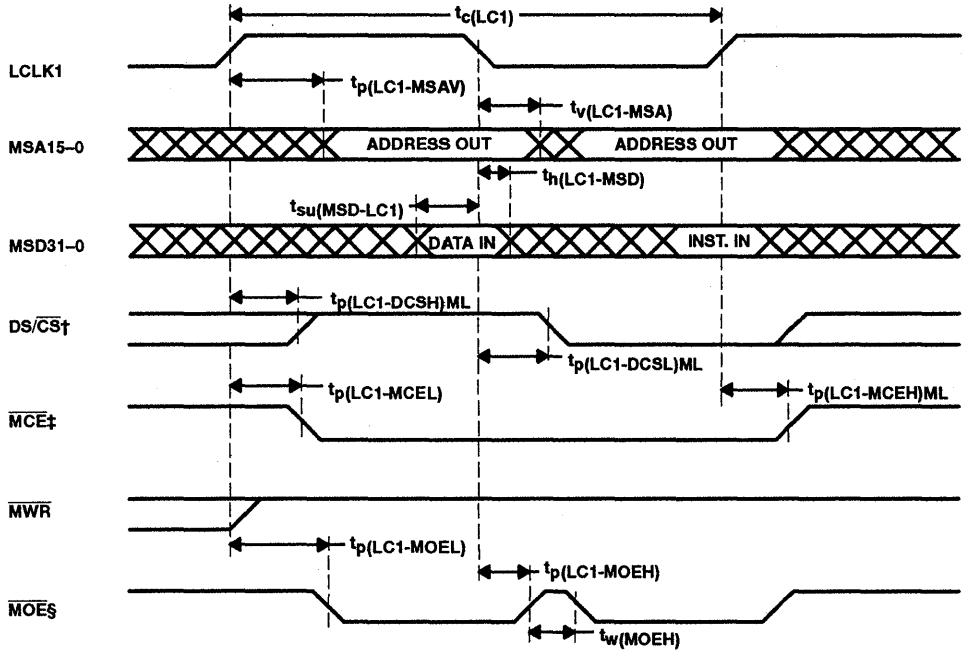
PARAMETER MEASUREMENT INFORMATION



† Q1, Q2, Q3, and Q4 represent the first, second, third, and fourth quarter clocks, respectively, of the LCLK1 clock period.

Figure 16. Coprocessor Mode, SMJ34082A to DRAM/VRAM Memory

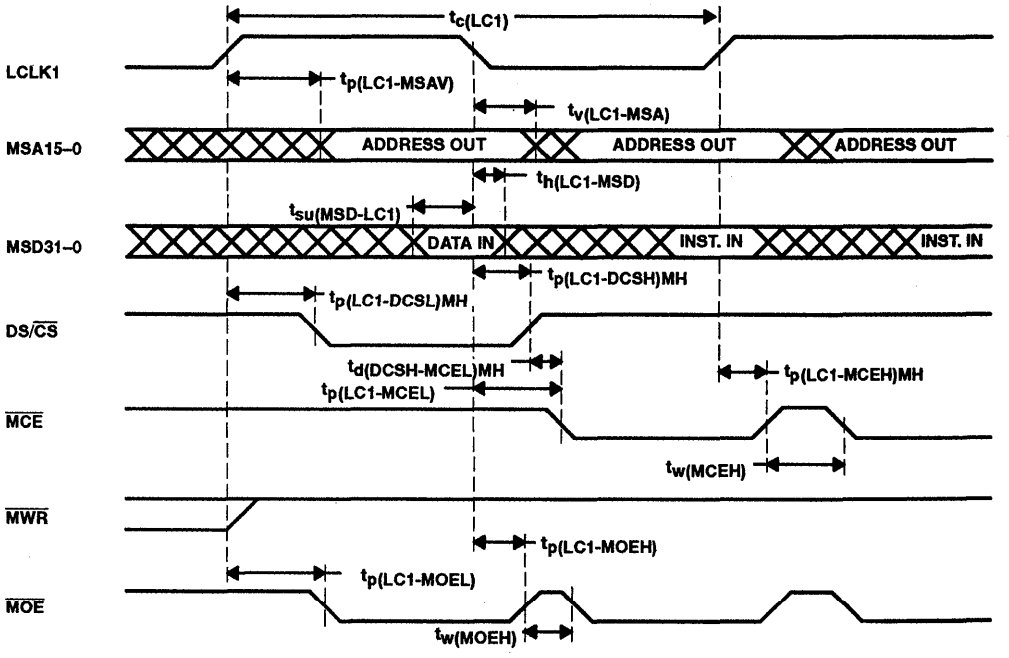
PARAMETER MEASUREMENT INFORMATION



† The setting of DS/$\overline{CS}$ determines whether the value on the MSD bus is an instruction or data.
‡ $\overline{MCE}$ dos not toggle at each clock edge.
§ $\overline{MOE}$ goes high at each clock edge.
NOTE: This example shows a data read followed by an instruction read.

Figure 17. Coprocessor Mode MSD Bus Timing, Memory to SMJ34082A with MEMCFG Low

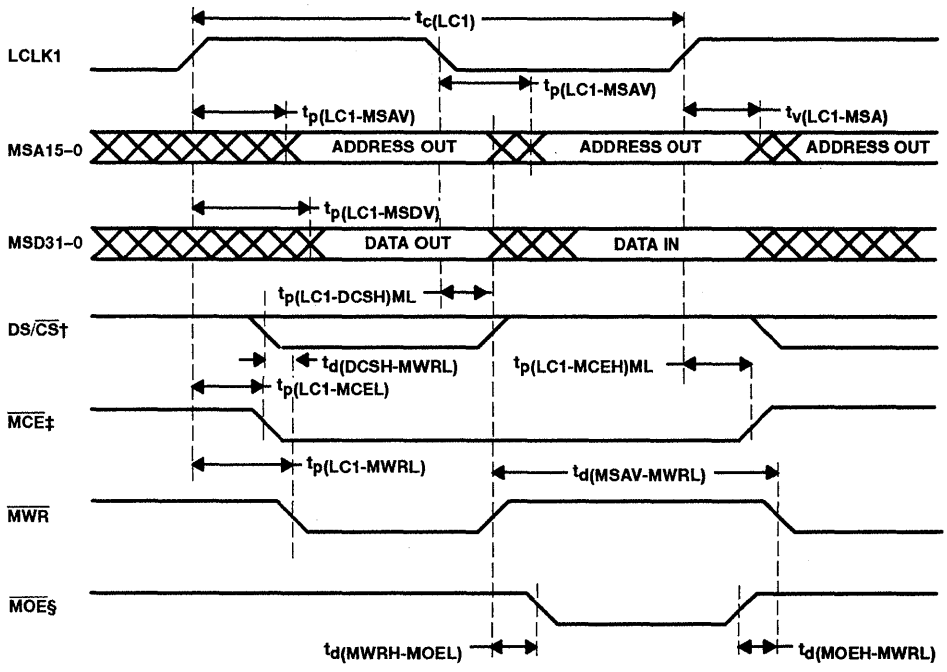
PARAMETER MEASUREMENT INFORMATION



NOTE: This example shows a data read followed by an instruction read followed by an instruction read. This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is invoked by setting the MEMCFG bit high in the configuration register. When MEMCFG is high, DS/$\overline{CS}$ and $\overline{MCE}$ rise after every clock edge. In this mode, DS/$\overline{CS}$ and $\overline{MCE}$ may not both be active (low) at the same time.

Figure 18. Coprocessor Mode MSD Bus Timing, Memory to SMJ34082A with MEMCFG High

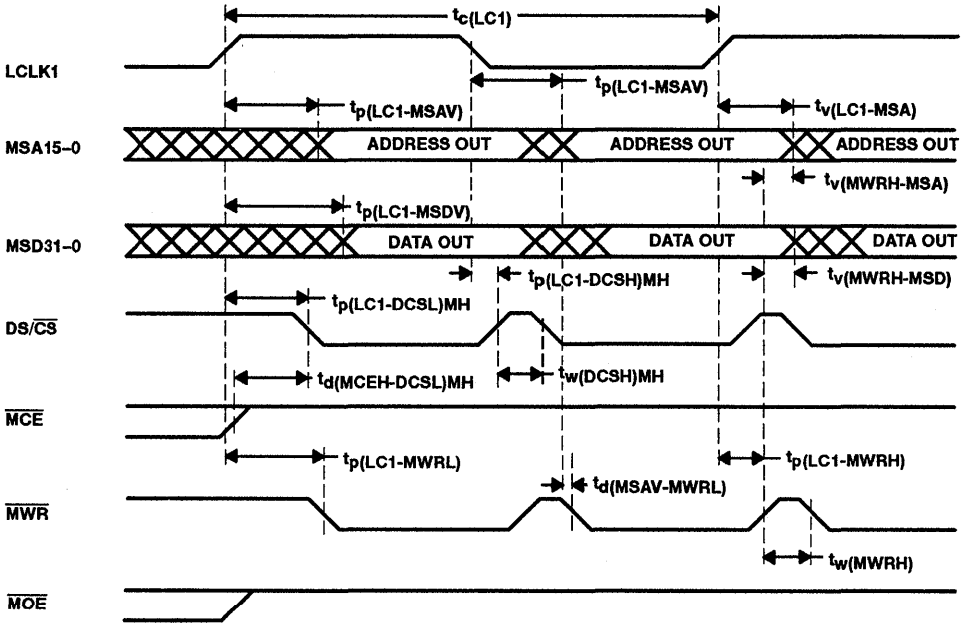
PARAMETER MEASUREMENT INFORMATION



† The setting of DS/$\overline{CS}$ determines whether the value on the MSD bus is an instruction or data.
‡ $\overline{MCE}$ does not toggle at each clock edge.
§ $\overline{MWR}$ goes high at each clock edge.
NOTE: This example shows a data write followed by a code read.

Figure 19. Coprocessor Mode MSD Bus Timing, SMJ34082A to Memory with MEMCFG Low

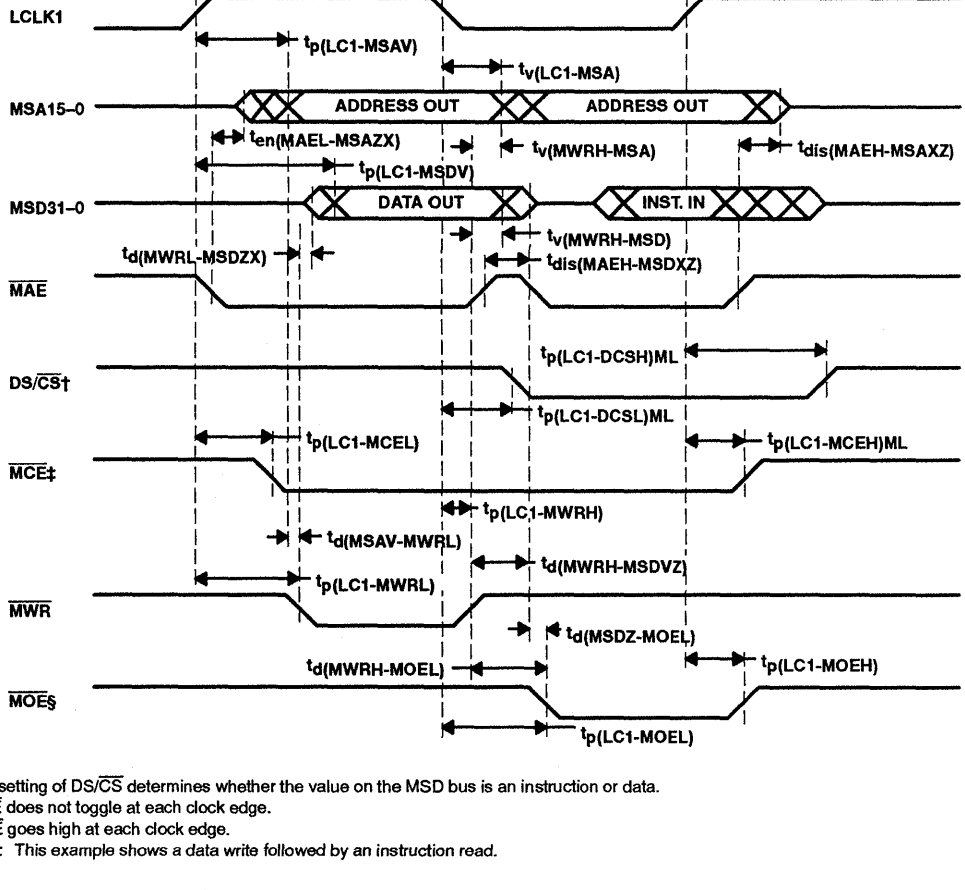
PARAMETER MEASUREMENT INFORMATION



NOTE: This example shows multiple data writes. Timing for multiple code writes would be similar. This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is invoked by setting the MEMCFG bit high in the configuration register. When MEMCFG is high, DS/$\overline{CS}$ and $\overline{MCE}$ rise after every clock edge. In this mode, DS/$\overline{CS}$ and $\overline{MCE}$ may not both be active (low) at the same time.

Figure 20. Coprocessor Mode MSD Bus Timing, SMJ34082A to Memory with MEMCFG High

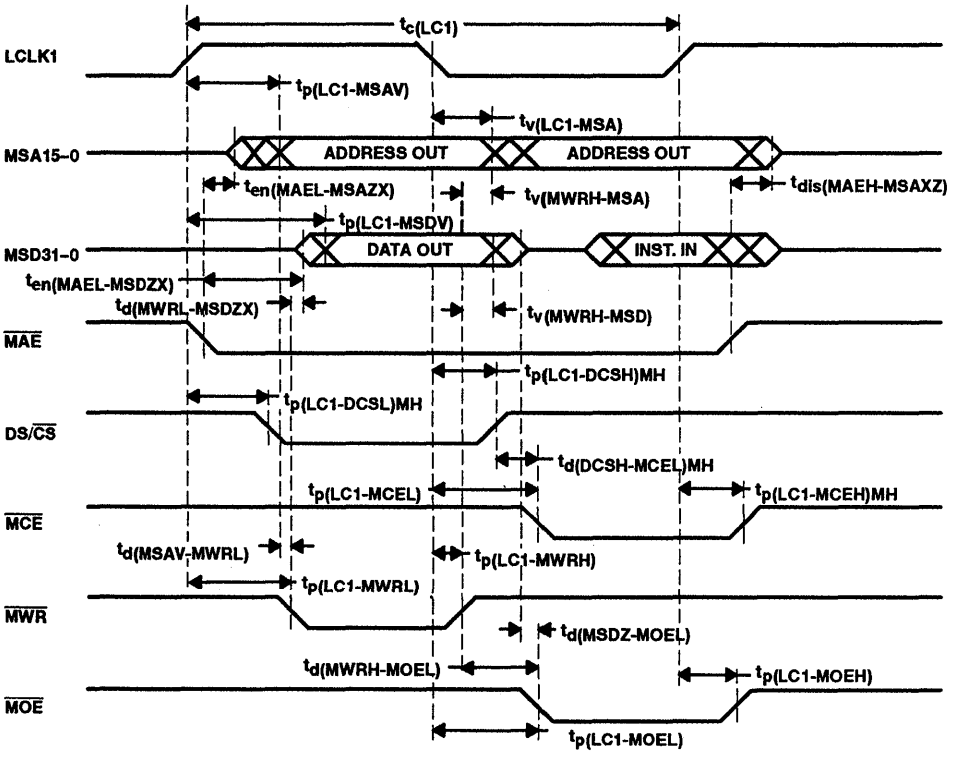
PARAMETER MEASUREMENT INFORMATION



† The setting of DS/$\overline{CS}$ determines whether the value on the MSD bus is an instruction or data.
‡ $\overline{MCE}$ does not toggle at each clock edge.
§ $\overline{MOE}$ goes high at each clock edge.
NOTE: This example shows a data write followed by an instruction read.

Figure 21. Coprocessor Mode, MSD Enable/Disable Timing with MEMCFG Low

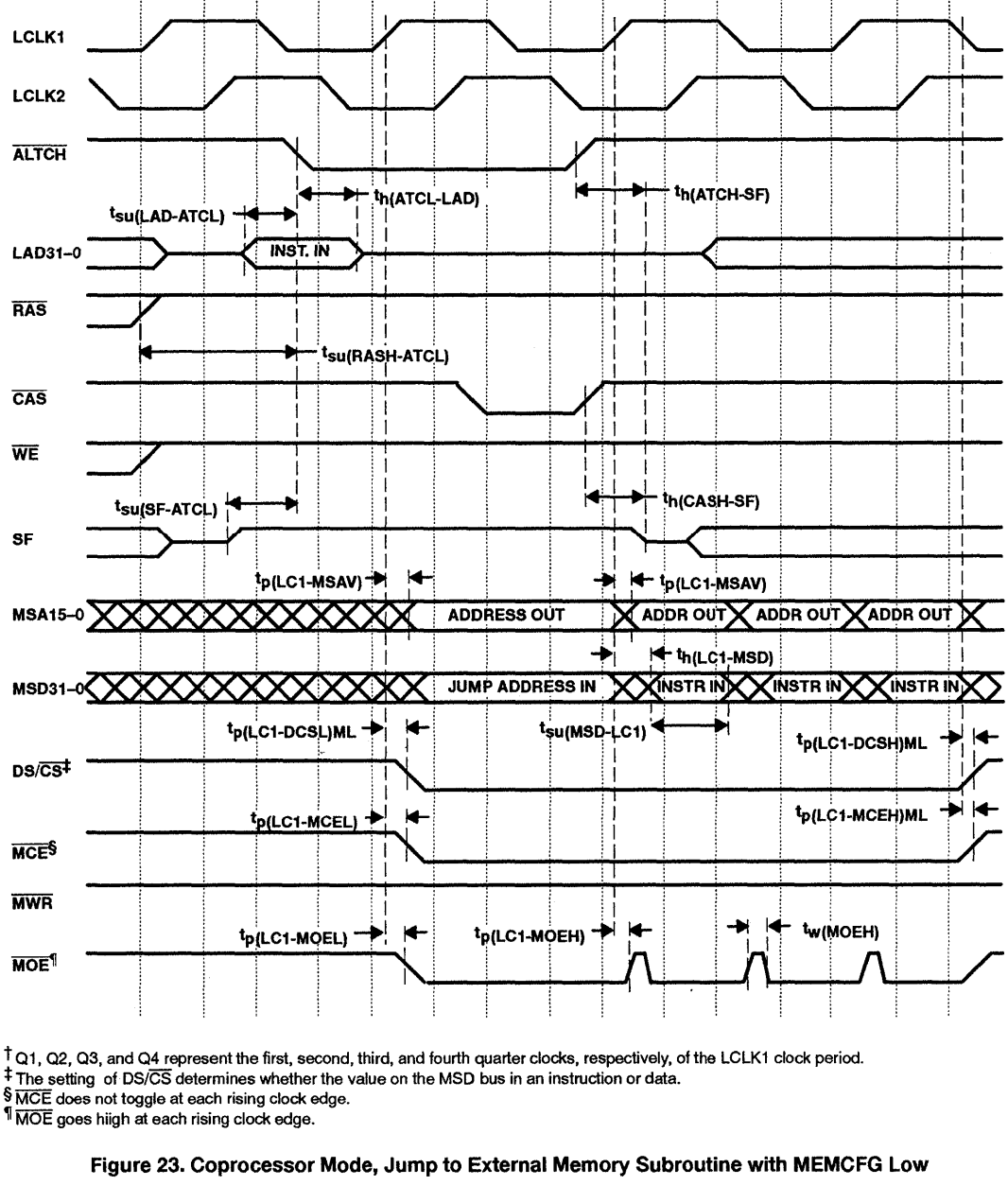
PARAMETER MEASUREMENT INFORMATION



NOTE: This example shows a data write followed by an instruction read. Timing for multiple code writes would be similar. This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is invoked by setting the MEMCFG bit high in the configuration register. When MEMCFG is high, DS/$\overline{CS}$ and $\overline{MCE}$ rise after every clock edge. In this mode, DS/$\overline{CS}$ and $\overline{MCE}$ may not both be active (low) at the same time.

Figure 22. Coprocessor Mode, MSD Bus Enable/Disable Timing with MEMCFG High

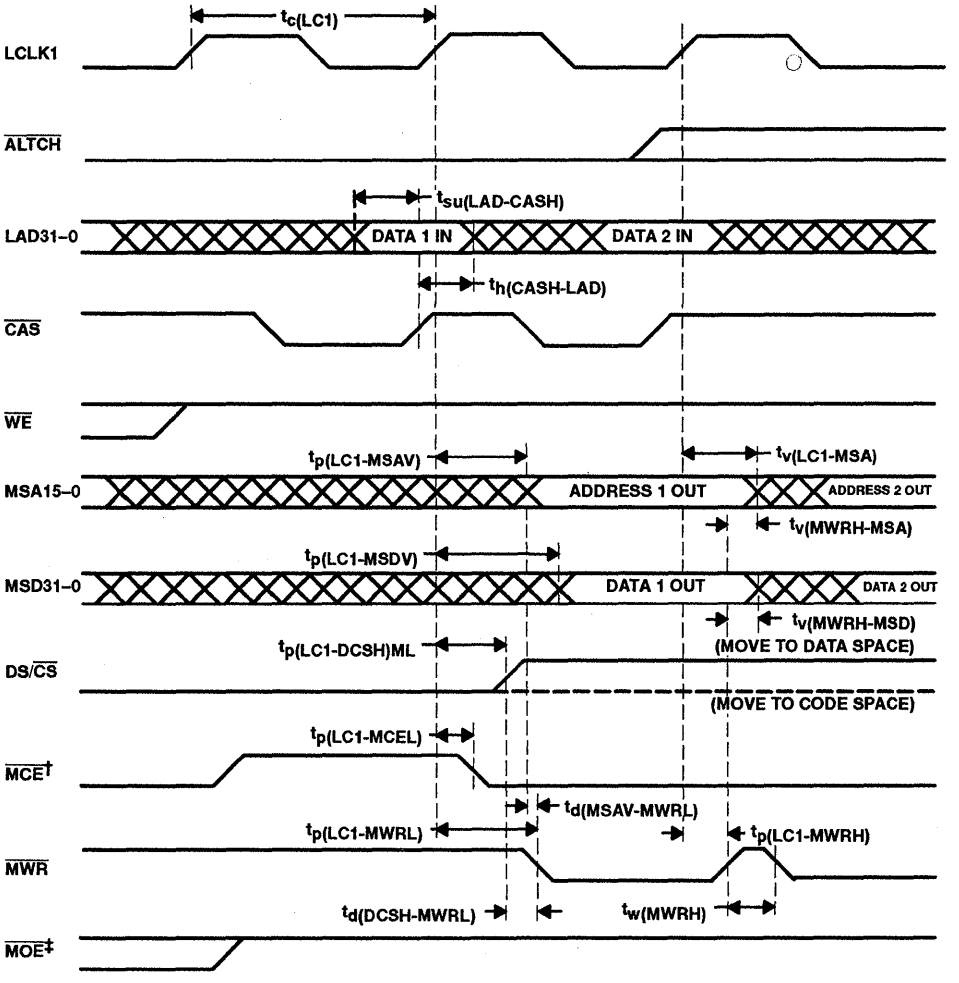
PARAMETER MEASUREMENT INFORMATION



† Q1, Q2, Q3, and Q4 represent the first, second, third, and fourth quarter clocks, respectively, of the LCLK1 clock period.
‡ The setting of DS/$\overline{CS}$ determines whether the value on the MSD bus in an instruction or data.
§ $\overline{MCE}$ does not toggle at each rising clock edge.
¶ $\overline{MOE}$ goes hiigh at each rising clock edge.

Figure 23. Coprocessor Mode, Jump to External Memory Subroutine with MEMCFG Low

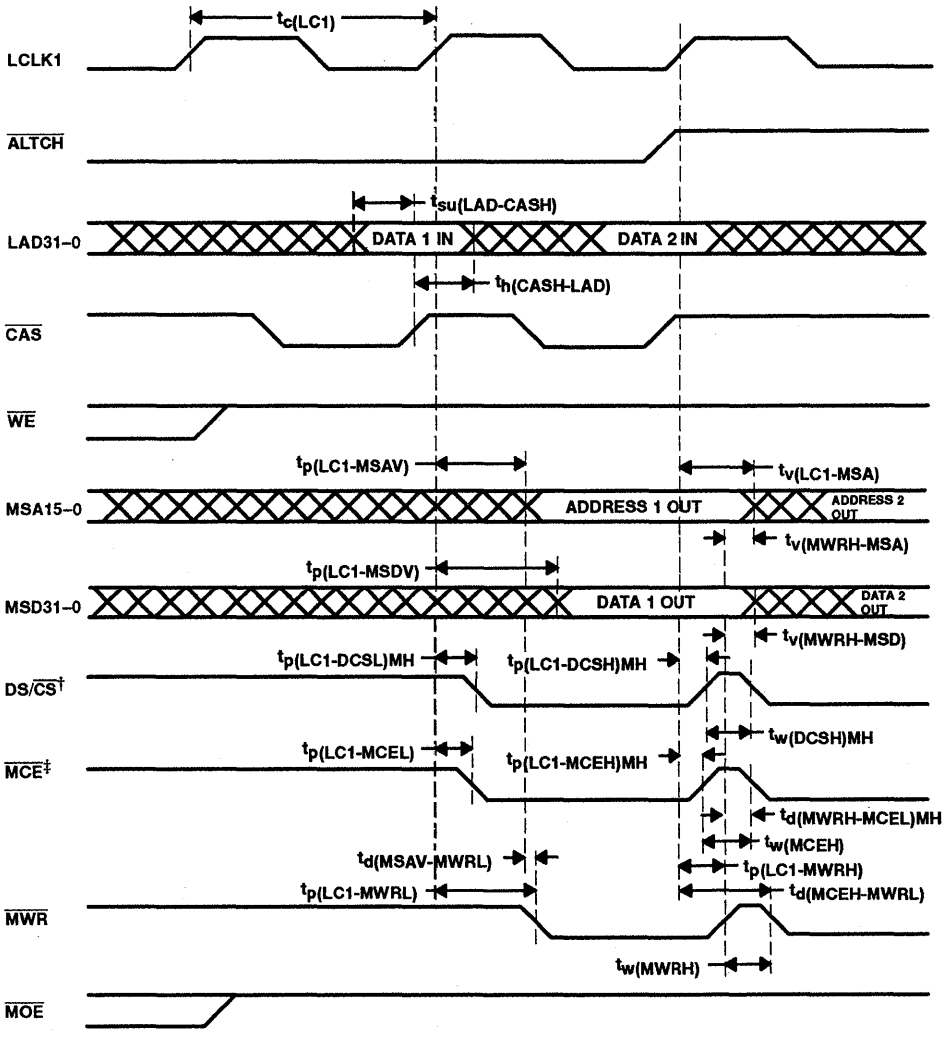
PARAMETER MEASUREMENT INFORMATION



† $\overline{MCE}$ does not toggle at each clock edge.
‡ $\overline{MOE}$ goes high at each clock edge.

Figure 24. Coprocessor Mode, LAD to MSD Bus Transfer Timing with MEMCFG Low

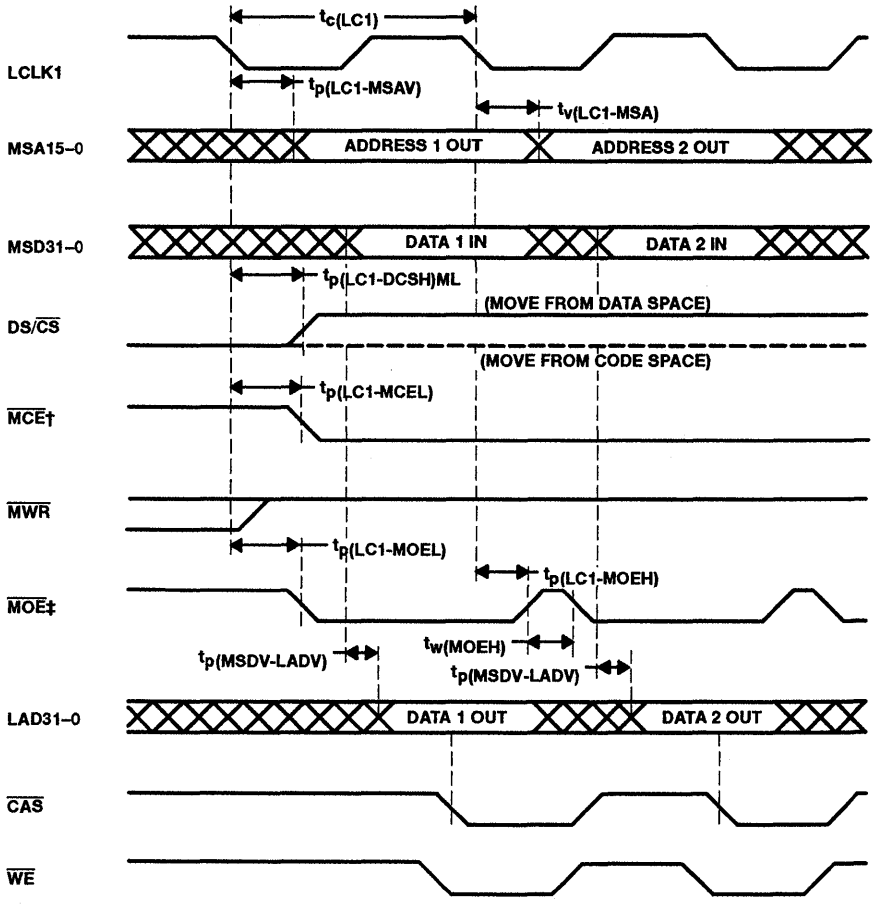
PARAMETER MEASUREMENT INFORMATION



† DS/$\overline{CS}$ valid for moves to data space; $\overline{MCE}$ valid for moves to code space. Only one of these would be valid for each move instruction.
‡ This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is invoked by setting the MEMCFG bit high in the configuration register.

Figure 25. Coprocessor Mode, LAD to MSD Bus Transfer Timing with MEMCFG High

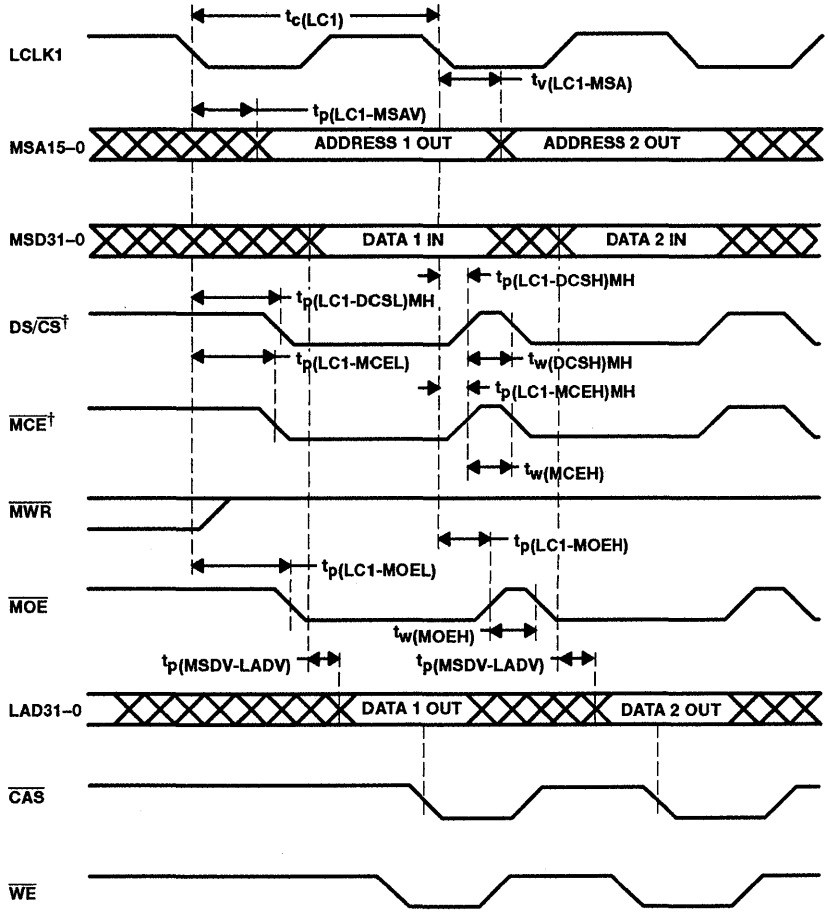
PARAMETER MEASUREMENT INFORMATION



† $\overline{MCE}$ dos not toggle at each clock edge.
‡ $\overline{MOE}$ goes high at each clock edge.

Figure 26. Coprocessor Mode, MSD to LAD Bus Transfer Timing with MEMCFG Low

PARAMETER MEASUREMENT INFORMATION



† DS/$\overline{CS}$ valid for moves to data space; $\overline{MCE}$ valid for moves to code space. Only one would be valid for each move instruction.

NOTE: This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is involved by setting the MEMCFG bit high in the configuration register.

Figure 27. Coprocessor Mode, MSD to LAD Bus Transfer Timing with MEMCFG High

PARAMETER MEASUREMENT INFORMATION

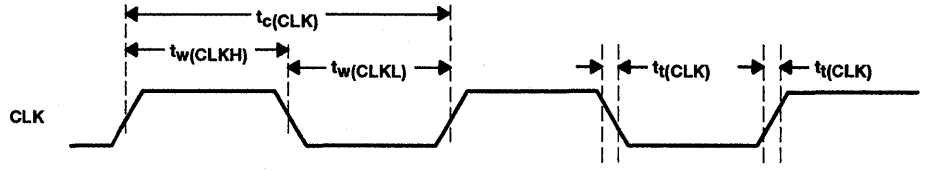
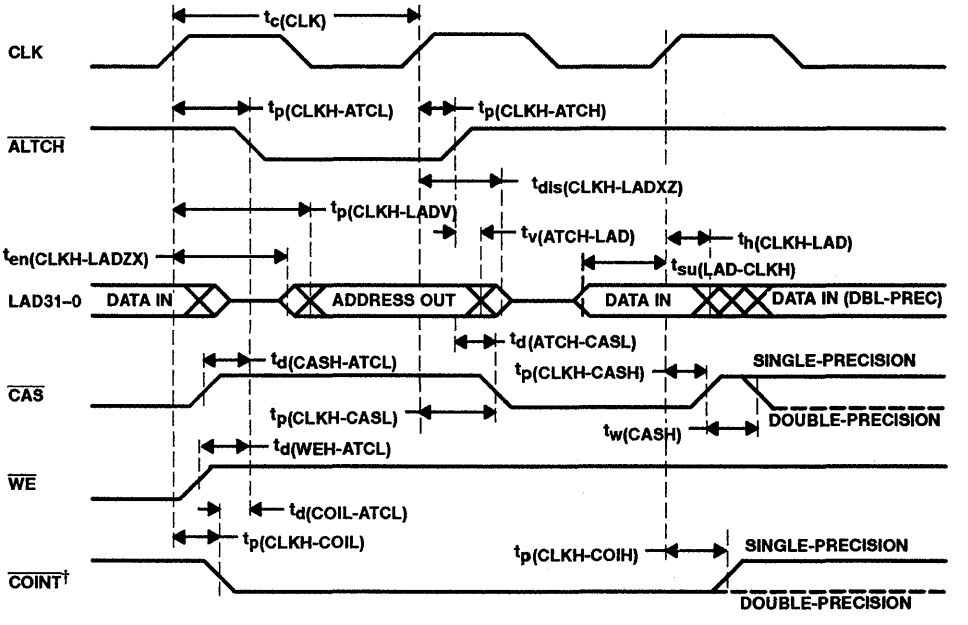


Figure 28. Host-Independent Mode, Input Clock

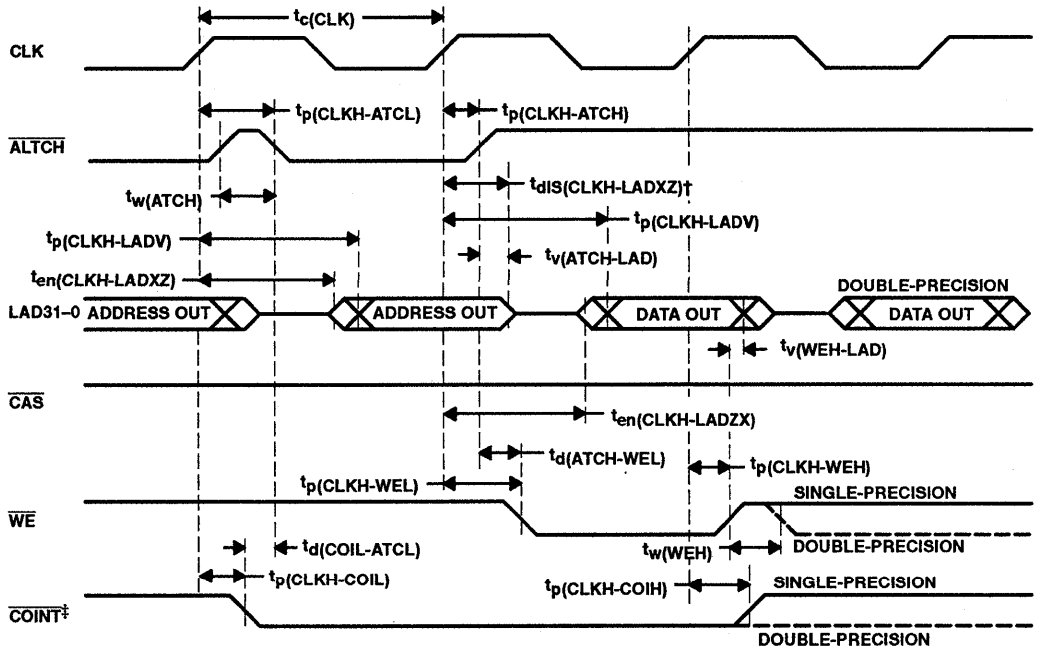


† $\overline{\text{COINT}}$ timing is for LADCFG high only. When the LADCFG bit is set high in the configuratin register, $\overline{\text{COINT}}$ is controlled by bit 1 of the LAD move instruction instead of the set mask instruction.

NOTE: This timing diagram assumes an external address latch to store address for external memory reads. Data input hold time on the latch is zero; data (or address) output hold time is nonzero.

Figure 29. Host-Independent Mode, LAD Bus Timing for Memory to SMJ34082A

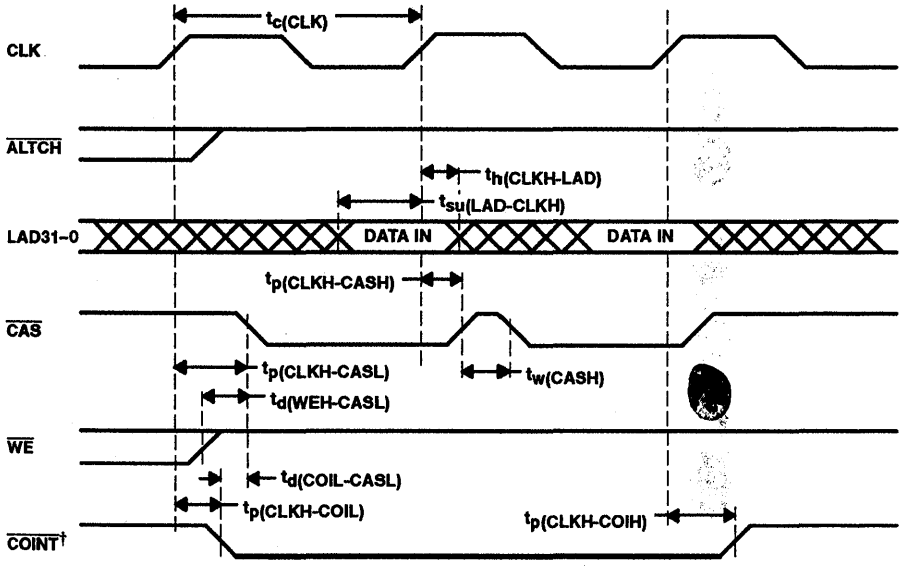
PARAMETER MEASUREMENT INFORMATION



† Valid only for last write in series. The LAD bus is not placed in high-impedance state between consecutive outputs.
‡ $\overline{\text{COINT}}$ timing is for LADCFG high only. When the LADCFG bit is set high in the configuration register, $\overline{\text{COINT}}$ is controlled by bit 1 of the LAD move instruction instead of the set mask instruction.
NOTE: This timing diagram assumes an external address latch to store address for external memory reads. Data input hold time is zero. Data (or address) output hold time is nonzero. Valid only for last write in series. The LAD bus is not placed in high impedance between consecutive outputs.

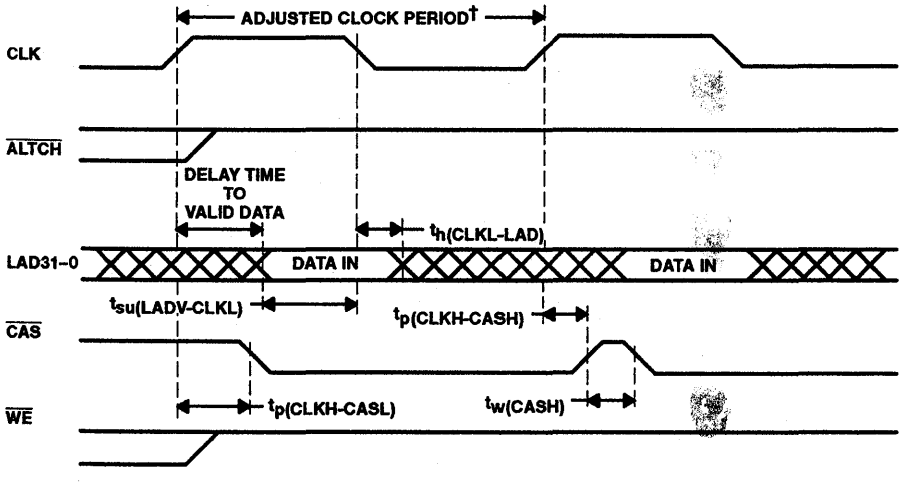
Figure 30. Host-Independent Mode, LAD Bus Timing for SMJ34082A to Memory

PARAMETER MEASUREMENT INFORMATION



† $\overline{\text{COINT}}$ timing is for LADCFG high only. When the LADCFG bit is set high in the configuration register, $\overline{\text{COINT}}$ is controlled by bit 1 of the LAD move instruction instead of the set mask instruction.

Figure 31. Host-Independent Mode, LAD Bus Timing Input to SMJ34082A

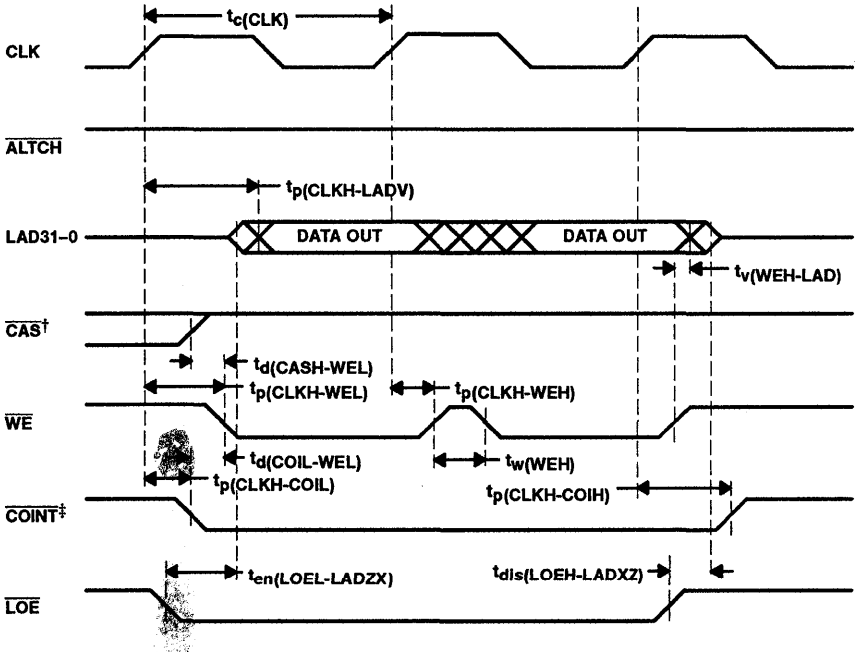


† This mode permits data input which does not meet the minimum setup before CLK high. For immediate data input, CLK must be high for more than 20 ns. This input mode cannot be used to input data for divides and square roots.

Adjusted clock period = Normal clock period + Data delay + 5 ns

Figure 32. Host-Independent Mode, LAD Bus Timing Input of Immediate Data to SMJ34082A

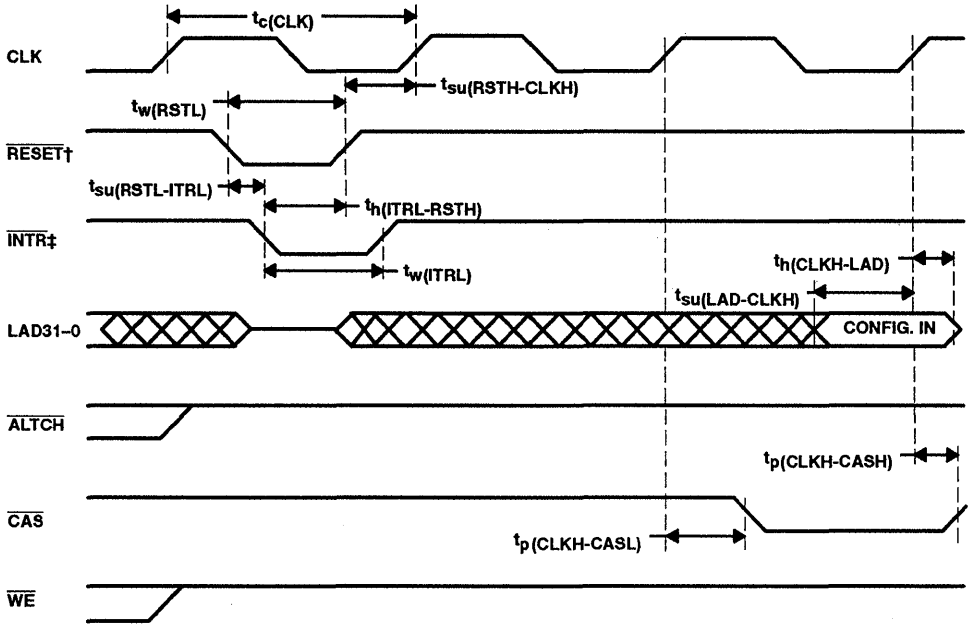
PARAMETER MEASUREMENT INFORMATION



† When the LADCFG bit is high, $\overline{LOE}$ high places $\overline{CAS}$ and $\overline{WE}$ (as well as the LAD bus) in high impedance.
‡ Valid only for LADCFG high. When the LADCFG bit is high in the configuration register, $\overline{COINT}$ is controlled by bit 1 of the LAD move instruction instead of the set mask instruction.
NOTE: If the instruction writes the result of an FPU operation to a register and outputs the result to the LAD bus, in the same cycle, the minimum clock period must be extended.

Figure 33. Host-Independent Mode, LAD Bus Timing Output from SMJ34082A

PARAMETER MEASUREMENT INFORMATION



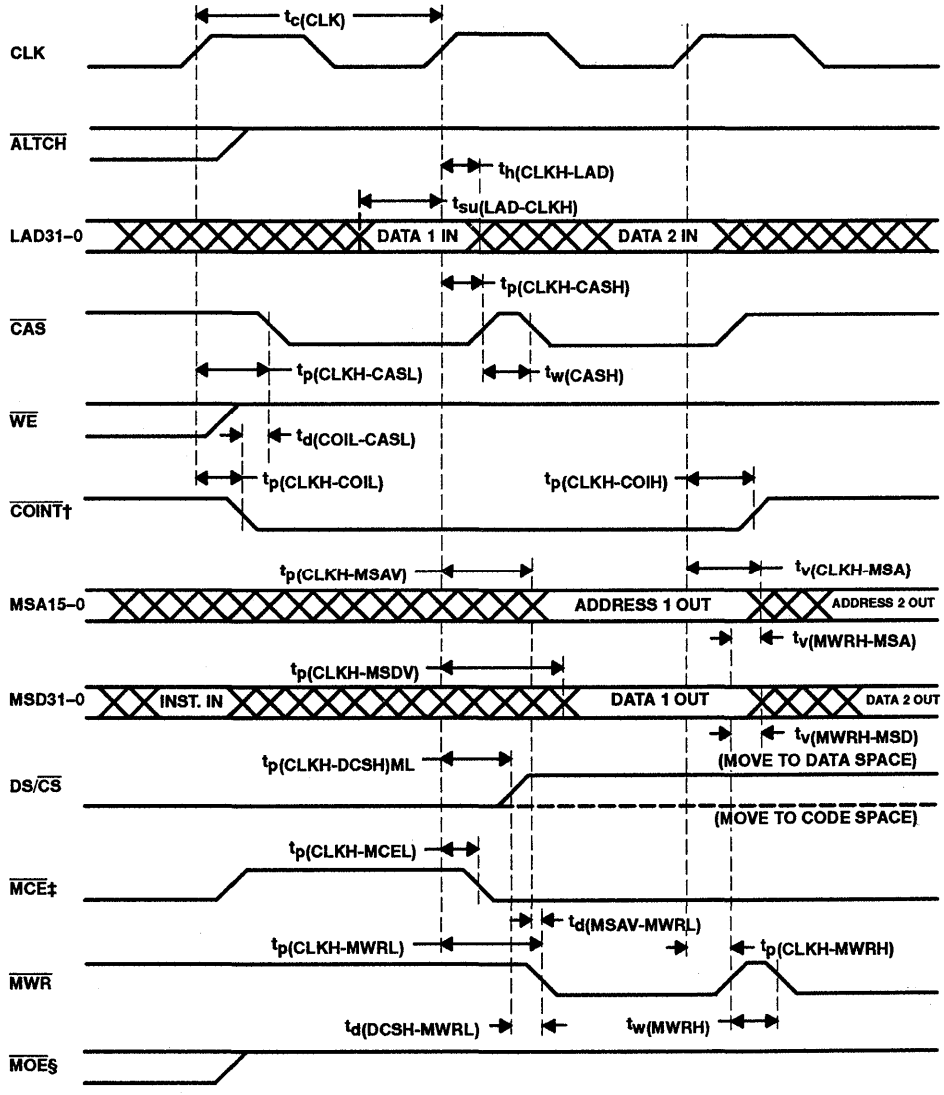
† RESET is level sensitive. When RESET is set low, both LAD and MSD buses are placed in high-impedance state. When RESET is released, the sequencer forces a jump to address 0. If INTR goes low while RESET is low, the loader moves 64 words through to the external memory on MSD. Timing for the LAD to MSD move is shown in a later diagram, with the exception that the first word on LAD loads the configuration register and does not pass to the MSD bus.

‡ INTR may be low one or more cycles after RESET goes low. RESET is held low, and then INTR is taken low. The bootstrap loader starts when RESET is set high, which may involve a delay of one or more cycles after INTR goes low.

NOTE: When the bootstrap loader is invoked, the first data word input on the LAD bus should be the configuration register settings, which will be written into the configuration register. This allows the user to select the MEMCFG setting, for reading or writing memory on the MSD port, as well as the LADCFG setting for the LAD bus interface.

Figure 34. Host-Independent Mode LAD Bus Timing, Bootstrap Loader Operation

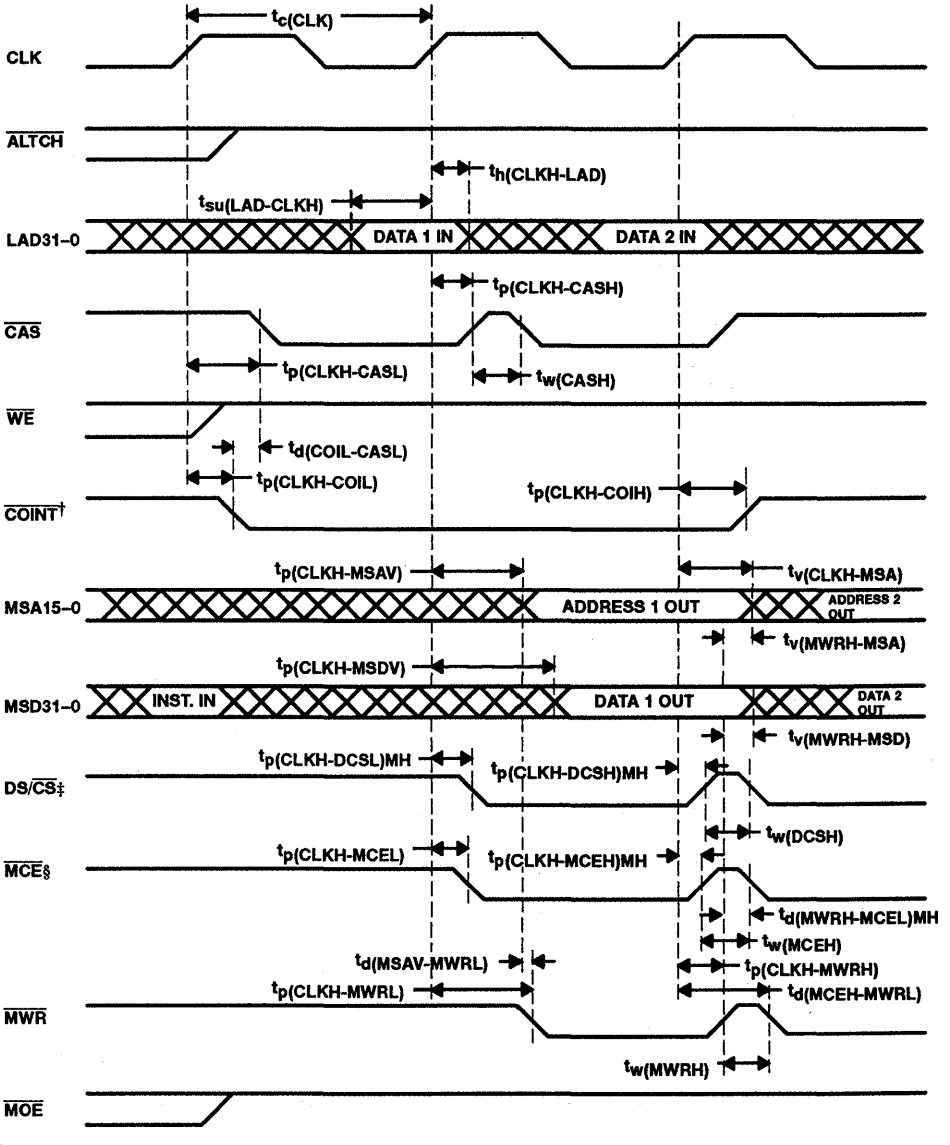
PARAMETER MEASUREMENT INFORMATION



† $\overline{\text{COINT}}$ timing is for LADCFG high only. When the LADCFG bit is set high in the configuration register, $\overline{\text{COINT}}$ is controlled by bit 1 of the LAD move instruction instead of the set mask instruction.
‡ $\overline{\text{MCE}}$ *does not toggle at each rising clock edge.*
§ $\overline{\text{MOE}}$ goes high at each rising clock edge.

Figure 35. Host-Independent Mode, LAD to MSD Bus Timing with MEMCFG Low

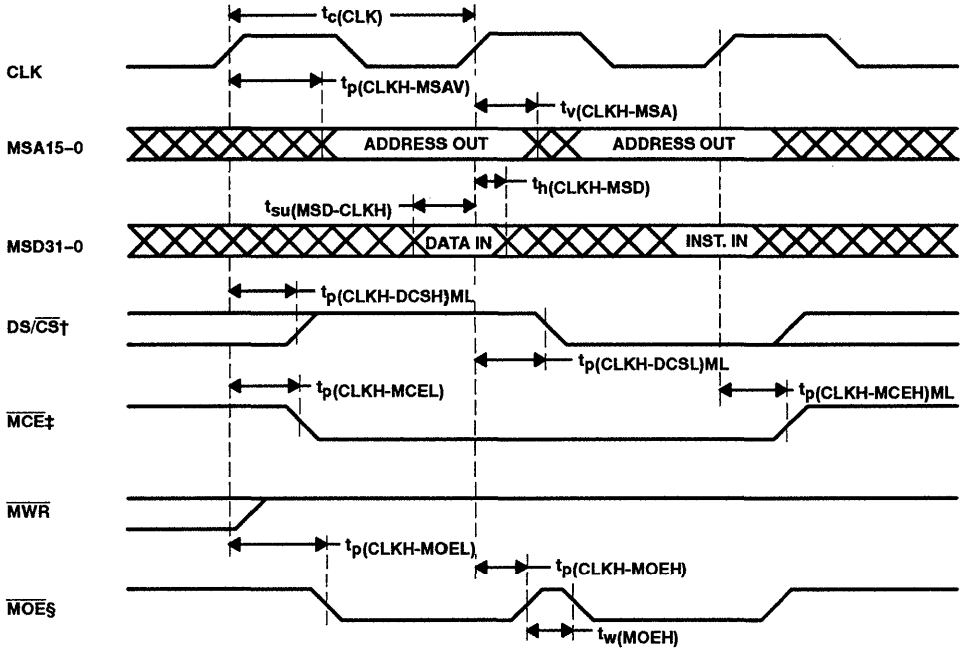
PARAMETER MEASUREMENT INFORMATION



† COINT timing is for LADCFG high only. When the LADCFG bit is set high in the configuration register, COINT is controlled by bit 1 of the LAD move instruction instead of the set mask instruction.
‡ DS/CS valid for moves to data space; MCE valid for moves to code space. Only one of these would be valid for each move instruction.
§ This option for using DS/CS as data space chip enable and MCE as code space chip enable is invoked by setting the MEMCFG bit high in the configuration register.

Figure 36. Host-Independent Mode, LAD to MSD Bus Transfer Timing with MEMCFG High

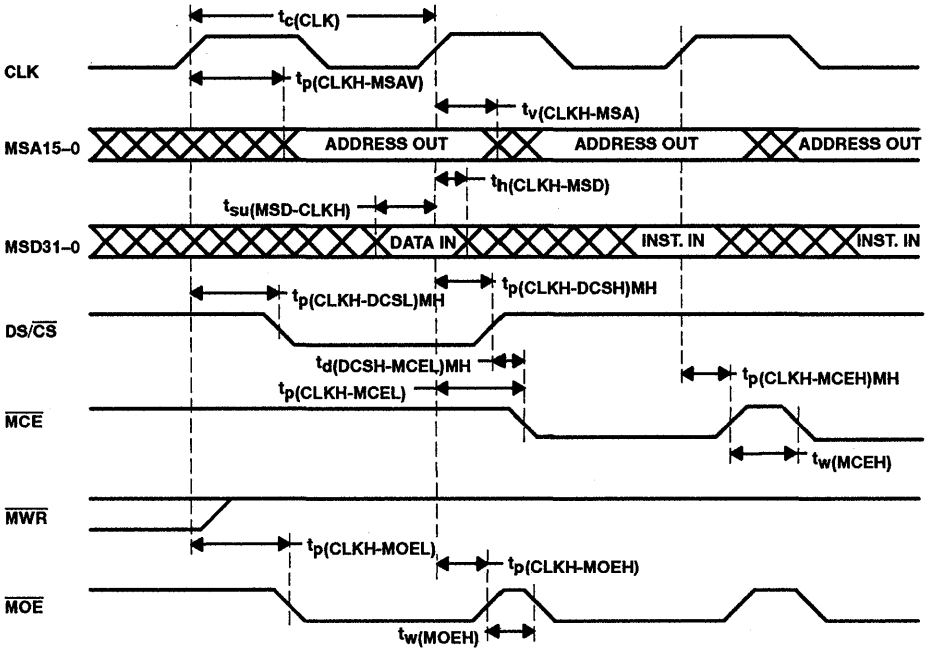
PARAMETER MEASUREMENT INFORMATION



† The setting of DS/$\overline{CS}$ determines whether the value on the MSD bus is an instruction or data.
‡ $\overline{MCE}$ dos not toggle at each rising clock edge.
§ $\overline{MOE}$ goes high at each rising clock edge.
NOTE; This example shows a data read followed by an instruction read.

Figure 37. Host-Independent Mode MSD Bus Timing, Memory to SMJ34082A with MEMCFG Low

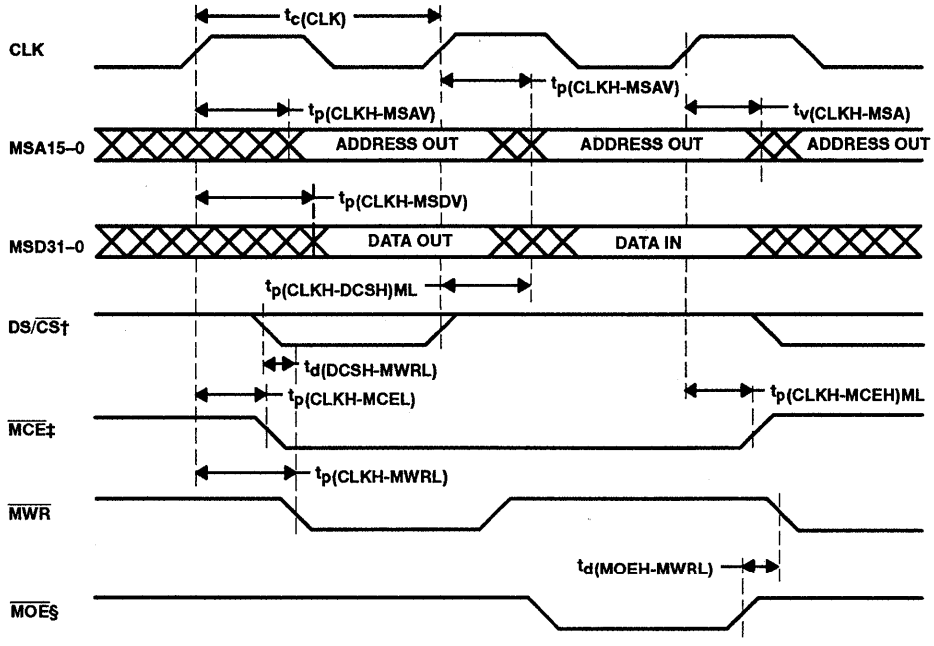
PARAMETER MEASUREMENT INFORMATION



NOTE: This example shows a data read followed by an instruction read followed by an instruction read. This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is invoked by setting the MEMCFG bit high in the configuration register. When MEMCFG is high, DS/$\overline{CS}$ and $\overline{MCE}$ rise after every rising clock edge. In this mode, DS/$\overline{CS}$ and $\overline{MCE}$ may not both be active (low) at the same time.

Figure 38. Host-Independent Mode MSD Bus Timing, Memory to SMJ34082A with MEMCFG High

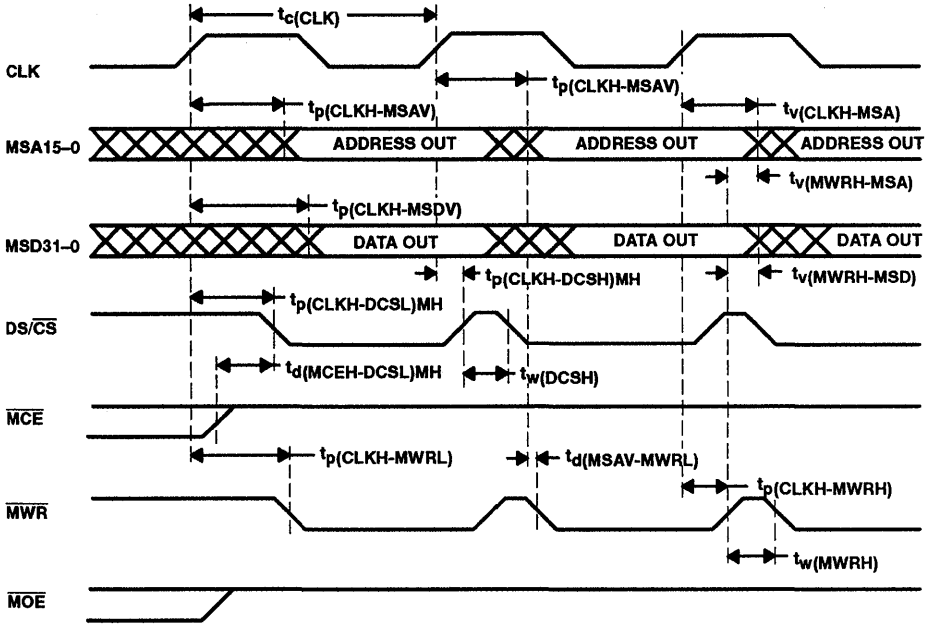
PARAMETER MEASUREMENT INFORMATION



† The setting of DS/$\overline{CS}$ determines whether the value on the MSD bus is an instruction or data.
‡ $\overline{MCE}$ dos not toggle at each rising clock edge.
§ $\overline{MWR}$ goes high at each rising clock edge.
NOTE: This example shows a data write followed by a code read.

Figure 39. Host-Independent Mode MSD Bus Timing, SMJ34082A to Memory with MEMCFG Low

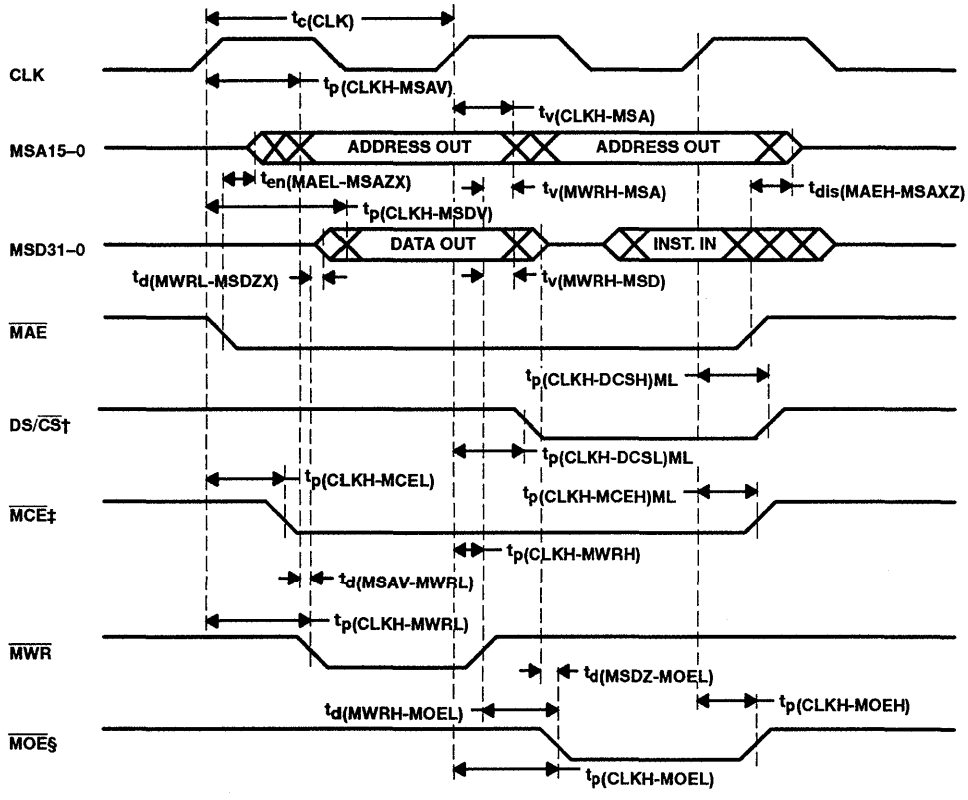
PARAMETER MEASUREMENT INFORMATION



NOTE: This example shows multiple data writes. Timing for multiple code writes would be similar. This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is invoked by setting the MEMCFG bit high in the configuration register. When MEMCFG is high, DS/$\overline{CS}$ and $\overline{MCE}$ rise after every rising clock edge. In this mode, DS/$\overline{CS}$ and $\overline{MCE}$ may not both be active (low) at the same time.

Figure 40. Host-Independent Mode MSD Bus Timing, SMJ34082A to Memory with MEMCFG High

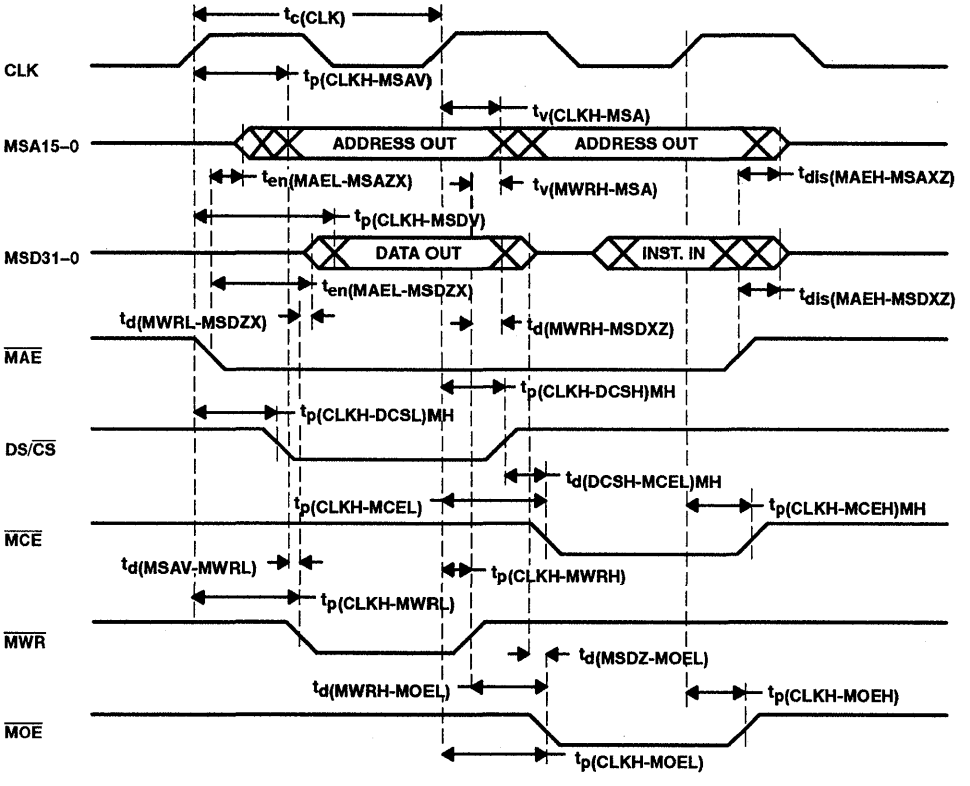
PARAMETER MEASUREMENT INFORMATION



† The setting of DS/$\overline{CS}$ determines whether the value on the MSD bus is an instruction or data.
‡ $\overline{MCE}$ dos not toggle at each rising clock edge.
§ $\overline{MOE}$ goes high at each rising clock edge.
NOTE: This example shows a data write followed by an instruction read.

Figure 41. Host-Independent Mode, MSD Enable/Disable Timing with MEMCFG Low

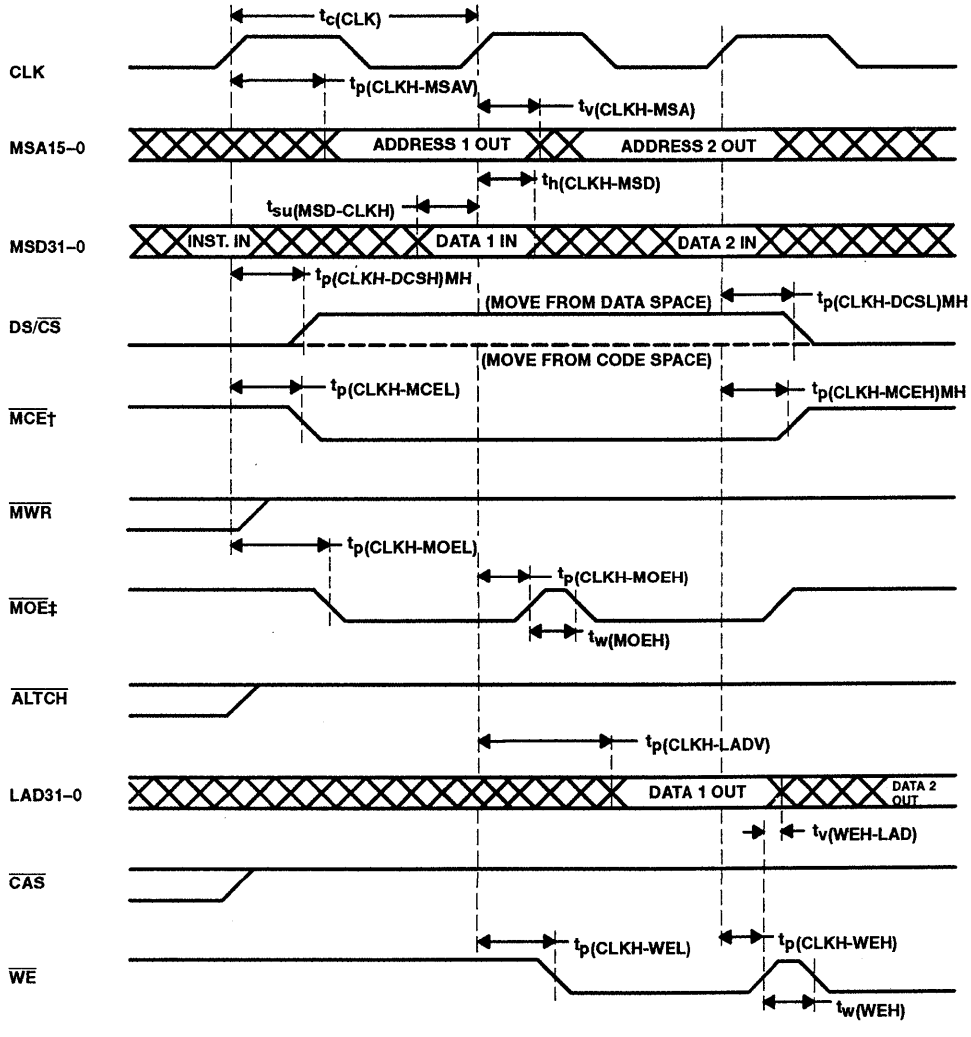
PARAMETER MEASUREMENT INFORMATION



NOTE: This example shows a data write followed by an instruction read. Timing for multiple code writes would be similar. This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is invoked by setting the MEMCFG bit high in the configuration register. When MEMCFG is high, DS/$\overline{CS}$ and $\overline{MCE}$ rise after every rising clock edge. In this mode, DS/$\overline{CS}$ and $\overline{MCE}$ may not both be low at the same time.

Figure 42. Host-Independent Mode, MSD Bus Enable/Disable Timing with MEMCFG High

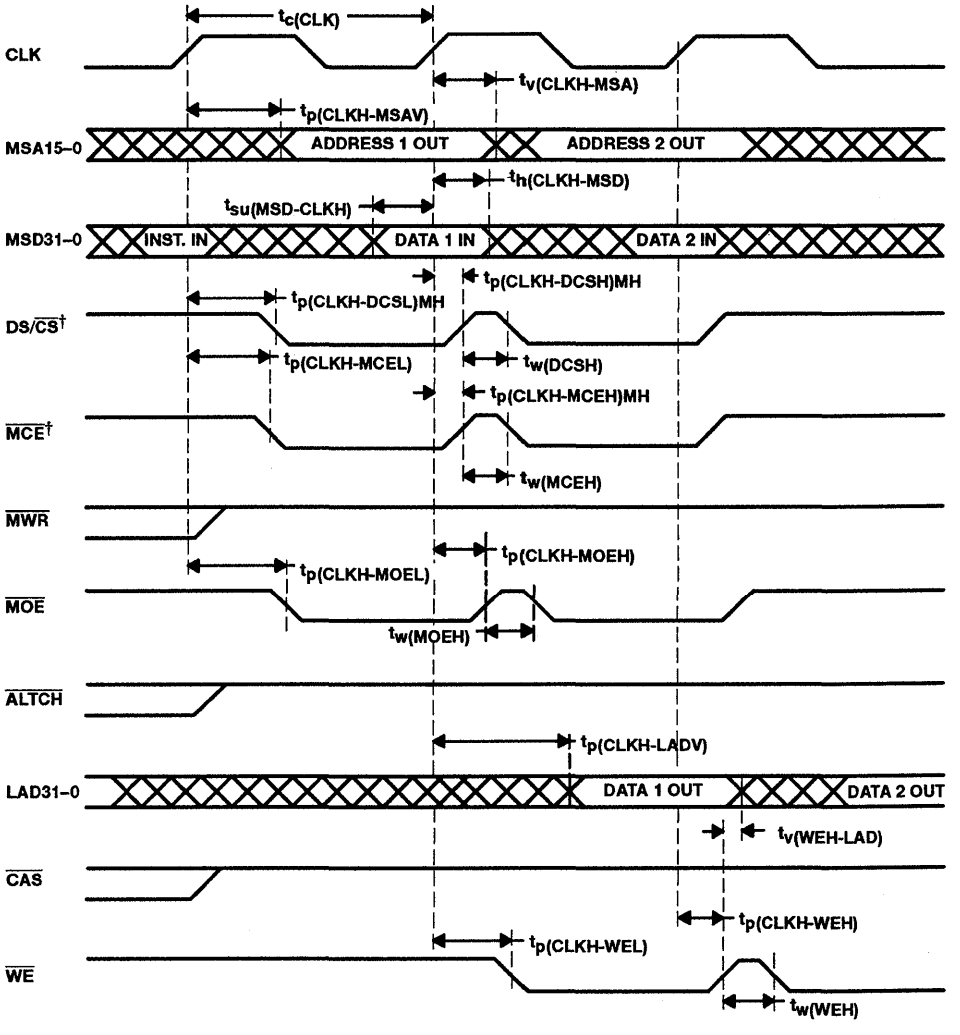
PARAMETER MEASUREMENT INFORMATION



† $\overline{MCE}$ dos not toggle at each rising clock edge.
‡ $\overline{MOE}$ goes high at each rising clock edge.

Figure 43. Host-Independent Mode, MSD to LAD Bus Transfer Timing with MEMCFG High

PARAMETER MEASUREMENT INFORMATION

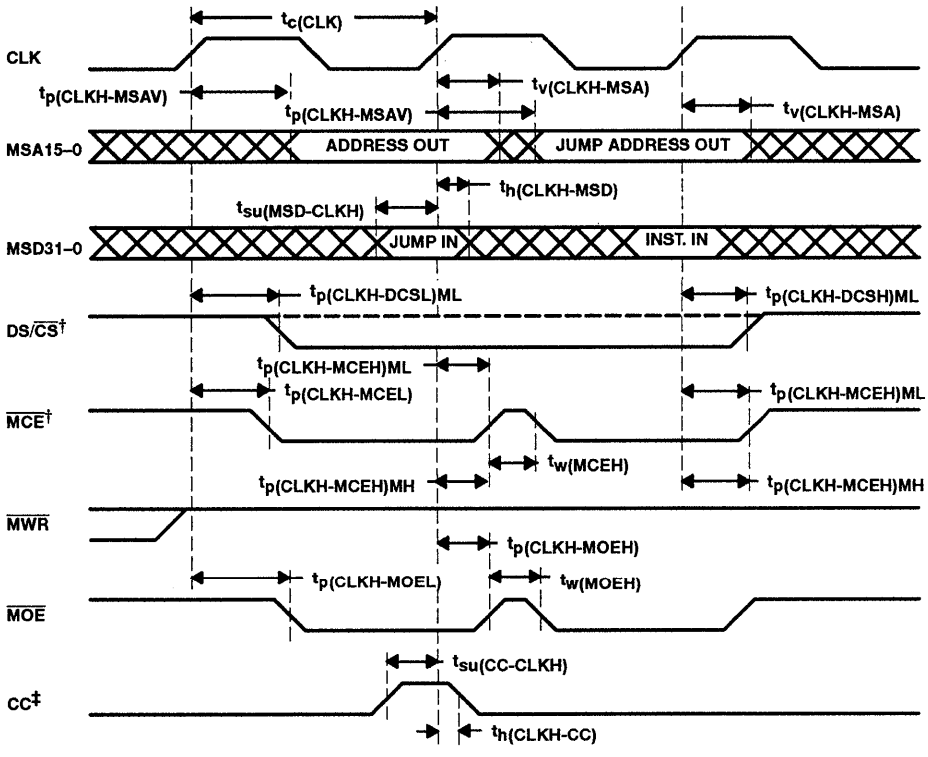


† DS/$\overline{CS}$ valid for moves to data space; $\overline{MCE}$ valid for moves to code space. Only one would be valid for each move instruction.

NOTE: This option for using DS/$\overline{CS}$ as data space chip enable and $\overline{MCE}$ as code space chip enable is involved by setting the MEMCFG bit high in the configuration register.

Figure 44. Host-Independent Mode, MSD to LAD Bus Transfer Timing with MEMCF High

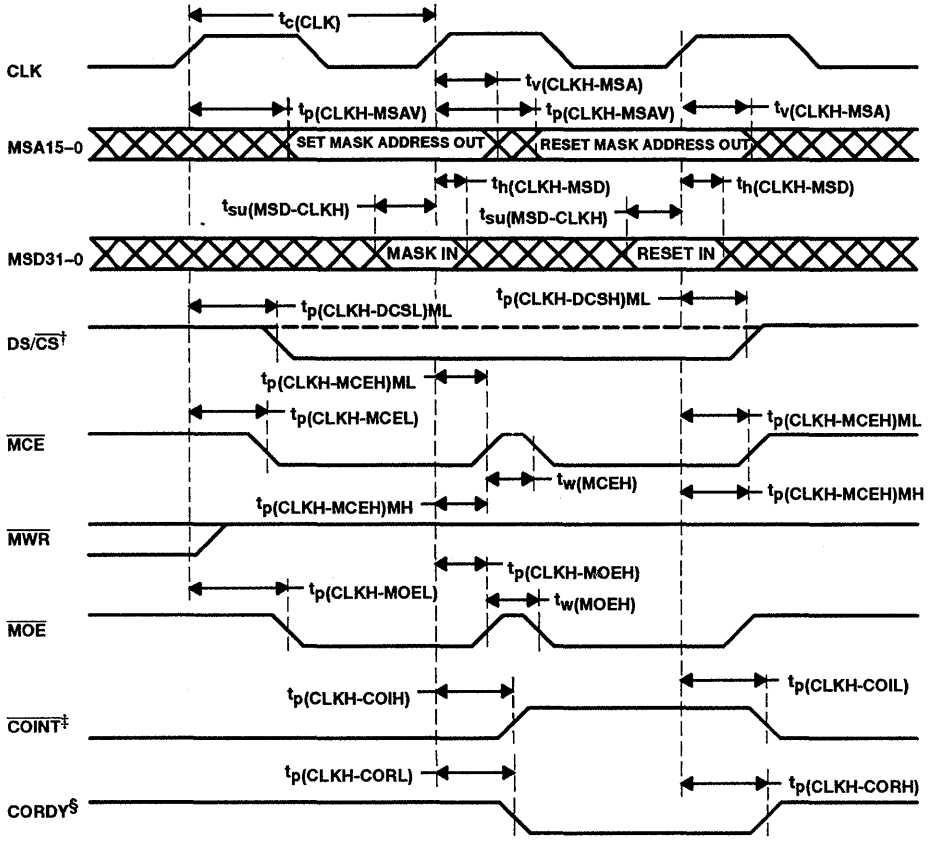
PARAMETER MEASUREMENT INFORMATION



† Dotted line shows DS/$\overline{CS}$ for MEMCFG high.
‡ The CC input is registered on each rising edge of the clock, so the CC bit can be latched one cycle and tested during the next cycle.

Figure 45. Host-Independent Mode, MSD Bus Timing Test Condition (CC) and Branch

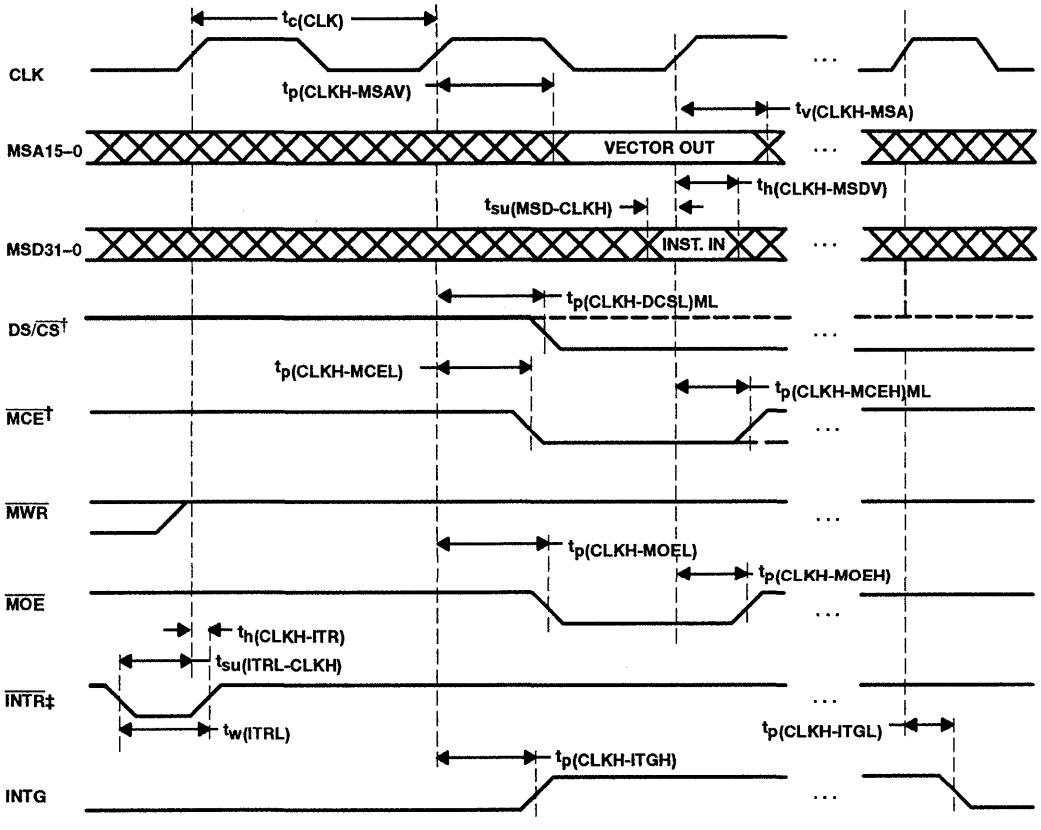
PARAMETER MEASUREMENT INFORMATION



† Dotted line shows DS/$\overline{CS}$ for MEMCFG high.
‡ Valid for MEMCFG low only. When MEMCFG low, $\overline{COINT}$ is set high by the set mask instruction, and it remains high until reset with another set mask instruction.
§ The CORDY output is set low by the set mask instruction, and it remains low until reset with another set mask instruction.

Figure 46. Host-Independent Mode MSD Bus Timing, SET/RESET $\overline{COINT}$ and CORDY

PARAMETER MEASUREMENT INFORMATION

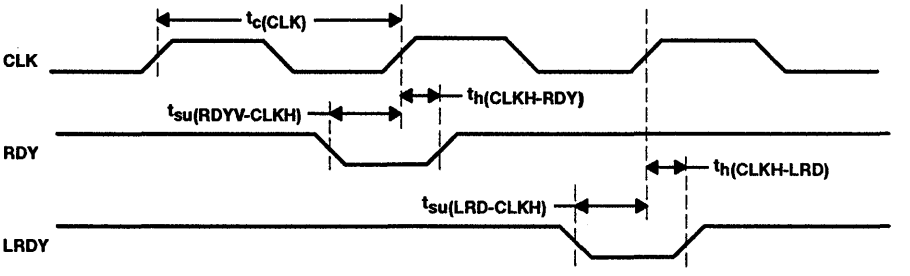


† Dotted lines show DS/$\overline{CS}$ and $\overline{MCE}$ for MEMCFG high.
‡ $\overline{INTR}$ is negative-edged triggered.

NOTE: Interrupts are not granted during multi-cycle instructions. This example shows two interrupt requests. The first is granted immediately; the second, after the first is finished. INTG remains high after an interrupt is granted until interrupts are reenabled or a return from interrupt instruction is executed.

Figure 47. Host-Independent Mode, MSD Bus Timing External Interrupt to SMJ34082A

PARAMETER MEASUREMENT INFORMATION



NOTE: When either RDY or LRDY is set low and the setup time before CLK high is observed, the device is stalled for one or more clock cycles, until RDY or LRDY is set high again. During a wait state, internal states and status are preserved and output signals do not change. LRDY can be used in this manner only in the host-independent mode.

Figure 48. Host-Independent Mode, MSD Bus Timing Wait State Timing

PROGRAMMING INFORMATION

programming the SMJ34082A

The SMJ34082A is supported by a software development tool kit, including a C compiler and an assembler. Program development using the tools is described in the TMS34082A tool kit documentation. Information on internal instructions and listing of the external instructions are provided in the following sections.

In both the coprocessor and host-independent modes, the SMJ34082A instruction word is 32 bits long. The number, length, and arrangement of fields in the 32-bit word depends on the operating mode and operation selected. Internal microcode to the SMJ34082A is not restricted to the same 32-bit instruction formats so certain internal programs may execute faster than the same operations written with external code can achieve.

In the coprocessor mode, the SMJ34082A can execute instructions both from the SMJ34020 and from the program memory on the MSD bus (MSD31-0). In the host-independent mode the SMJ34082A is controlled from code input on the MSD bus. Internal instructions may be executed in the host-independent mode by performing a jump to the internal address.

PROGRAMMING INFORMATION

internal instructions

The SMJ34082A FPU performs a wide range of internal arithmetic and logical operations, as well as complex operations (flagged '†'), summarized below. Complex instructions are multi-cycle routines stored in the internal program ROM.

One-Operand Operations:

Absolute Value	1s Complement
Square Root	2s Complement
Reciprocal†	

Conversions:

Integer to Single	Single to Integer
Integer to Double	Double to Integer
Single to Double	Double to Single

Two-Operand Operations:

Add	Multiply
Subtract	Divide
Compare	

Matrix Operations:

4x4, 4x4 Multiply†	3x3, 3x3 Multiply†
1x4, 4x4 Multiply†	1x3, 3x3 Multiply†

Graphics Operations:

Backface Testing†	Polygon Elimination†
Polygon Clipping†	Viewport Scaling and Conversion†
2-D Linear Interpolation†	3-D Linear Interpolation†
2-D Window Compare†	3-D Volume Compare†
2-Plane Clipping (X,Y,Z)†	2-Plane Color Clipping (R,B,G,I)†
2-D Cubic Spline†	3-D Cubic Spline†

Image Processing:

3x3 Convolution†	

Chained Operations :

Polynomial Expansion†	Multiply/Accumulate†
1-D Min/Max†	2-D Min/Max†

Vector Operations:

Add†	Dot Product†
Subtract†	Cross Product†
Magnitude†	Normalization†
Scaling†	Reflection†

The internal ROM routines may be used in either the coprocessor or host-independent mode. In the coprocessor mode, the internal routines are invoked by SMJ34020 instructions to its coprocessor(s).

In the host-independent mode, the internal programs can be called as subroutines by the externally stored code. External programs can call internal routines by executing a jump to subroutine with bit 16 (internal code select) set high and the address of the internal routine as the jump address.

The format of the SMJ34082A instruction in the coprocessor mode is shown in Figure 49. The instruction is issued by the SMJ34020 via the LAD bus.

† Indicates a complex instruction.

PROGRAMMING INFORMATION

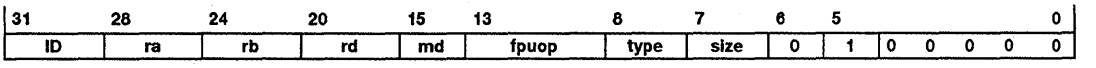


Figure 49. SMJ34082A Instruction

The 3-bit ID field identifies the coprocessor for which the instruction is intended. This coprocessor ID corresponds to the settings of the CID2-CID0 pins. To broadcast an instruction to all coprocessors, the ID is set to 4h.

Table 5. Coprocessor ID

ID	COPROCESSOR
000	FPU0
001	FPU1
010	FPU2
011	FPU3
100	FPU broadcast
101	Reserved
110	Reserved
111	User defined

Four coprocessor addressing modes are defined for the SMJ34082A. The md field indicates the addressing mode.

Table 6. Addressing Modes

MODE	MD FIELD	OPERATION
0	00	FPU internal operations with no jump or external moves
1	01	Transfer data to/from SMJ34020 registers
2	10	Transfer data to/from memory (controlled by SMJ34020)
3	11	External instructions

The type and size bits identify the type of operand; as shown below in Table 7. The I bit is used to indicate to the SMJ34082A that this is a reissue of a coprocessor instruction due to a bus interruption. The least significant four bits are the bus status bits, which will all be zero to indicate a coprocessor cycle.

Table 7. OPERAND Types

TYPE	SIZE	OPERAND TYPE
0	0	32-bit integer
0	1	Reserved
1	0	Single-precision floating-point (32-bit)
1	1	Double-precision floating-point (64-bit)

The ra, rb, and rd fields are for the two sources and destination within the FPU. Register addresses are listed in Table 1. For the ra and rb fields, only the four least significant bits of the register address are used. The ra field may only use the RA register file, C, and CT. The RB field may only use the RB register file, C and CT.

The Floating-Point Unit Operation (fpuop) field is the FPU opcode (5 bits) described in Tables 8, 9, and 10.

In the coprocessor mode, the SMJ34082A executes user-defined routines (stored in external memory on the MSD bus) by executing a jump to external code. For this instruction, the md field (bits 15-13) is set high and the fpuop field gives the routine number (0-31). The SMJ34082A multiplies the routine number by two to get the jump address. For example, routine number 14 would have a jump address of 28 decimal or 1C hex.

PROGRAMMING INFORMATION

The routines are coded using the external instruction format discussed in the next section. The last instruction should be a jump to internal instruction address 0FFFh with the I-bit(internal) set or a return from subroutine instruction. This puts the FPU in an idle state, waiting for the next instruction from the SMJ34020.

Table 8. Coprocessor Mode Instructions

FPUOP	TMS34020 ASSEMBLER OPCODE	DESCRIPTION
00000	ADDx	Sum of ra and rb, place in rd
00001	SUBx	Subtract rb from ra, place result in rd
00010	CMPx	Set status bits on result of ra minus rb
00011	SUBx	Subtract ra from rb, place result in rd
00100	ADDAx	Absolute value of sum of ra and rb, place result in rd
00101	SUBAx	Absolute value of (ra minus rb), place result in rd
00110	MOVE or MOVx	Load multiple FPU registers from SMJ34020 GSP or its memory
00111	MOVE or MOVx	Save multiple FPU registers to SMJ34020 GSP or its memory
01000	MPYx	Multiply ra and rb, place result in rd
01001	DIVx	Divide ra by rb, place result in rd
01010	INVx	Divide 1 by rb, place result in rd
01011	ASUBAx	Absolute value of ra minus absolute value of rb, place in rd
01100	reserved	
01101	MOVEx	Move ra to rd, multiple, for n registers
01110	MOVEx	Move rb to rd, multiple, for n registers
01111	(see Table 10)	Single operand instructions, rb field redefined
10000	CPWx	Compare point to window (set XLT, XGT, YLT, TGT)
10001	CPVx	Compare point to volume (set XLT, XGT, YLT, YGT, ZLT, ZGT)
10010	BACKFx	Test polygon for facing direction (backface test)
10011	INMNMXx	Setup FPU registers for MNMX1 or MNMX2 instruction
10100	LINTx	Given [X1, Y1, Z1], [X2, Y2, Z2], and a plane, find [X3, Y3, Z3]
10101	CLIPFx	Clip a line to a plane pair boundary (start with point 1)
10110	CLIPRx	Clip a line to a plane pair boundary (start with point 2)
10111	CLIPCFx	Clip color values to a plane pair boundary (start with point 1)
11000	SCALEx	Scale and convert coordinates for viewpoint
11001	MTRANx	Transpose a matrix
11010	CKVTXx	Compare a polygon vertex to a clipping volume
11011	CONVx	3x3 convolution
11100	CLIPCRx	Clip color values to a plane pair boundary (start with point 2)
11101	OUTC3x	Compare a line to a clipping value
11110	CSPLNx	Calculate cubic spline for given coefficients
11111	(see Table 11)	Vector and matrix instructions, rb field redefined

F denotes single-precision, D denotes double-precision floating-point, x denotes operand type, and a blank designates signed integer

PROGRAMMING INFORMATION

Table 9. Coprocessor Mode Instructions, FPUOP = 01111_2

RB	TMS34020 ASSEMBLER OPCODE	DESCRIPTION
0000	PASS	Copy ra to rd
0001	NOT	Place 1s complement of ra in rd
0010	ABS	Place absolute value of ra in rd
0011	NEG	Place negated value of ra in rd
0100	CVDF	Convert double in ra to single in rd (T and S define ra)
0100	CVFD	Convert single in ra to double in rd (T and S define ra)
0101	CVDI	Convert double in ra to integer in rd (T and S define ra)
0101	CVFI	Convert single in ra to integer in rd (T and S define ra)
0110	CVID	Convert integer in ra to double in rd (T and S define ra)
0110	CVIF	Convert integer in ra to single in rd (T and S define ra)
0111	VSCLx	Multiply each component of a velocity by a scaling factor
1000	SQARx	Place (ra * ra) in rd
1001	SQRTx	Extract square root or ra, place in rd
1010	SQRTAx	Extract square root of absolute value of ra, place in rd
1011	ABORT	Stop execution of any FPU instruction
1100	CKVTXI	Initialize check vertex instruction
1101	CHECK	Check for previous instruction completion
1110	MOVMEM	Move data from system memory to external memory @ MCADDR
1111	MOVMEM	Move data to system memory from external memory @ MCADDR

Table 10. Coprocessor Mode Instructions, FPUOP = 11111_2

RB	TMS34020 ASSEMBLER OPCODE	DESCRIPTION
0000	POLYx	Polynomial expansion
0001	MACx	Multiply and accumulate
0010	MNMX1x	Determine 1-D minimum and maximum of a series
0011	MNMX2x	Determine 2-D minimum and maximum of a series of pairs
0100	MMPY0x	Multiply matrix elements 0, 1, 2, 3 by vector element 0
0101	MMPY1x	Multiply matrix elements 4, 5, 6, 7 by vector element 1
0110	MMPY2x	Multiply matrix elements 8, 9, 10, 11 by vector element 2
0111	MMPY3x	Multiply matrix elements 12, 13, 14, 15 by vector element 3
1000	MADDx	Add matrix elements 12, 13, 14, 15 to vector
1001	VADDx	Add two vectors
1010	VSUBx	Subtract a vector from a vector
1011	VDOTx	Compute scalar dot product of two vectors
1100	VCROSx	Compute cross product of two vectors
1101	VMAGx	Determine the magnitude of a vector
1110	VNORMx	Normalize a vector to unit magnitude
1111	VRFLCTx	Given normal and incident vectors, find the reflection

F denotes single-precision, D denotes double-precision floating-point, x denotes operand type, and a blank designates signed integer

PROGRAMMING INFORMATION

external instructions

External instructions are 32 bits long, and their formats (number, length, and function of fields) depend on the operations being selected. Separate formats are provided for data transfers, FPU processing, test and branch operations, and subroutine calls.

Instructions that control FPU operations can select operands from input registers, internal feedback, or from the LAD bus (32-bit operations only). The format for an FPU processing instruction is shown in Figure 50.

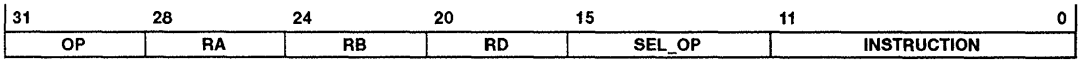


Figure 50. FPU Processing External Instruction Format

The op field selects the sequencer operation. Three continue instructions are available to permit control of the $\overline{WE}$ and $\overline{ALTCH}$ strobe outputs, which enable LAD output in the host-independent mode. The ra, rb, and rd fields are for the two sources and destination in the SMJ34082A register file. The sel_op field selects the source of the operands: register file or feedback registers. The instruction field designates the operation to be performed.

External instructions and cycle counts are listed in Table 11. Absolute values of operands or results, negated results, and wrapped number inputs are selectable options. Chained operations, using the multiplier and ALU in parallel, and other instructions to control program flow and move data are included.

External instruction timing depends on the pipeline registers setting, controlled by the PIPES2-1 bits in the configuration register. Most FPU processing instructions (with the exception of divide, square root, and double-precision multiply) execute in one cycle per pipeline stage.

PROGRAMMING INFORMATION

Table 11. External Instructions and Timing

SMJ34082A ASSEMBLER OPCODE	DESCRIPTION OF ROUTINE	PIPES2–1 11	PIPES2–1 10	PIPES2–1 01	PIPES2–1 00
ADD	Add A + B	1(1)	2(1)	2(1)	3(1)
AND	Logical AND A, B	1(1)	2(1)	2(1)	3(1)
ANDNA	Logical AND NOT A, B	1(1)	2(1)	2(1)	3(1)
ANDNB	Logical AND A, NOT B	1(1)	2(1)	2(1)	3(1)
CJMP	Conditional jump	1(1)	1(1)	1(1)	1(1)
CSJR	Conditional jump to subroutine	1(1)	1(1)	1(1)	1(1)
CMP	Compare A, B	1(1)	2(1)	2(1)	3(1)
COMPL	Pass 1s complement of A	1(1)	2(1)	2(1)	3(1)
DIV	Divide A / B				
	SP	8(8)	8(7)	9(7)	9(7)
	DP	13(13)	13(12)	15(12)	15(12)
	integer	16(16)	16(15)	17(15)	17(15)
DTOF	Convert from DP to SP	1(1)	2(1)	2(1)	3(1)
DTOI	Convert from DP to integer	1(1)	2(1)	2(1)	3(1)
DTOU	Convert from DP to unsigned integer	1(1)	2(1)	2(1)	3(1)
FTOD	Convert from SP to DP	1(1)	2(1)	2(1)	3(1)
FTOI	Convert from SP to integer	1(1)	2(1)	2(1)	3(1)
FTOU	Convert from SP to unsigned integer	1(1)	2(1)	2(1)	3(1)
ITOD	Convert from integer to DP	1(1)	2(1)	2(1)	3(1)
ITOF	Convert from integer to SP	1(1)	2(1)	2(1)	3(1)
LD	Load n words into register				
	SP	n + 1	n + 1	n + 1	n + 1
	DP	2n + 1	2n + 1	2n + 1	2n + 1
	integer	n + 1	n + 1	n + 1	n + 1
LDLCT	Load loop counter with value	1(1)	1(1)	1(1)	1(1)
LDMCADDR	Load MCADDR with value	1(1)	1(1)	1(1)	1(1)
MASK	Set programmable mask	1(1)	1(1)	1(1)	1(1)
MOVA	Move A (no status flags active)	1(1)	2(1)	2(1)	3(1)
MOVLM	Move n words from LAD bus to MSD bus				
	SP	n + 1	n + 1	n + 1	n + 1
	DP	2n + 1	2n + 1	2n + 1	2n + 1
	integer	n + 1	n + 1	n + 1	n + 1
MOVML	Move n words from MSD bus to LAD bus				
	SP	n + 1	n + 1	n + 1	n + 1
	DP	2n + 1	2n + 1	2n + 1	2n + 1
	integer	n + 1	n + 1	n + 1	n + 1
MOVRR	Multiple move, register to register				
	SP	n + 1	n + 1	n + 1	n + 1
	DP	2n + 1	2n + 1	2n + 1	2n + 1
	integer	n + 1	n + 1	n + 1	n + 1
MULT.ADD	Multiply $A_1 * B_1$, Add $A_2 + B_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)

DP denotes double-precision, and SP denotes single-precision.

PROGRAMMING INFORMATION

Tabale 11. External Instructions and Timing (Continued)

SMJ34082A ASSEMBLER OPCODE	DESCRIPTION OF ROUTINE	PIPES2–1 11	PIPES2–1 10	PIPES2–1 01	PIPES2–1 00
MULT.NEG	Multiply $A_1 * B_1$, Subtract $0 - A_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
MULT	Multiply A * B				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
MULT.PASS	Multiply $A_1 * B_1$, Add $A_2 + 0$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
MULT.SUB	Multiply $A_1 * B_1$, Subtract $A_2 - B_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
MULT.2SUBA	Multiply $A_1 * B_1$, Subtract $2 - A_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
MULT.SUBRL	Multiply $A_1 * B_1$, Subtract $B_2 - A_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
NEG	Pass –A (2s Complement)	1(1)	2(1)	2(1)	3(1)
NOR	Logical NOR A, B	1(1)	2(1)	2(1)	3(1)
OR	Logical OR A, B	1(1)	2(1)	2(1)	3(1)
PASS	Pass A	1(1)	2(1)	2(1)	3(1)
PASS	Pass B	1(1)	2(1)	2(1)	3(1)
PASS.ADD	Multiply $A_1 * 1$, Add $A_2 + B_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
PASS.NEG	Multiply $A_1 * 1$, Subtract $0 - A_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
PASS.PASS	Multiply $A_1 * 1$, Add $A_2 + 0$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
PASS.SUB	Multiply $A_1 * 1$, Subtract $A_2 - B_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
PASS.2SUBA	Multiply $A_1 * 1$, Subtract $2 - A_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)

DP denotes double-precision, and SP denotes single-precision.

PROGRAMMING INFORMATION

Table 11. External Instructions and Timing (Continued)

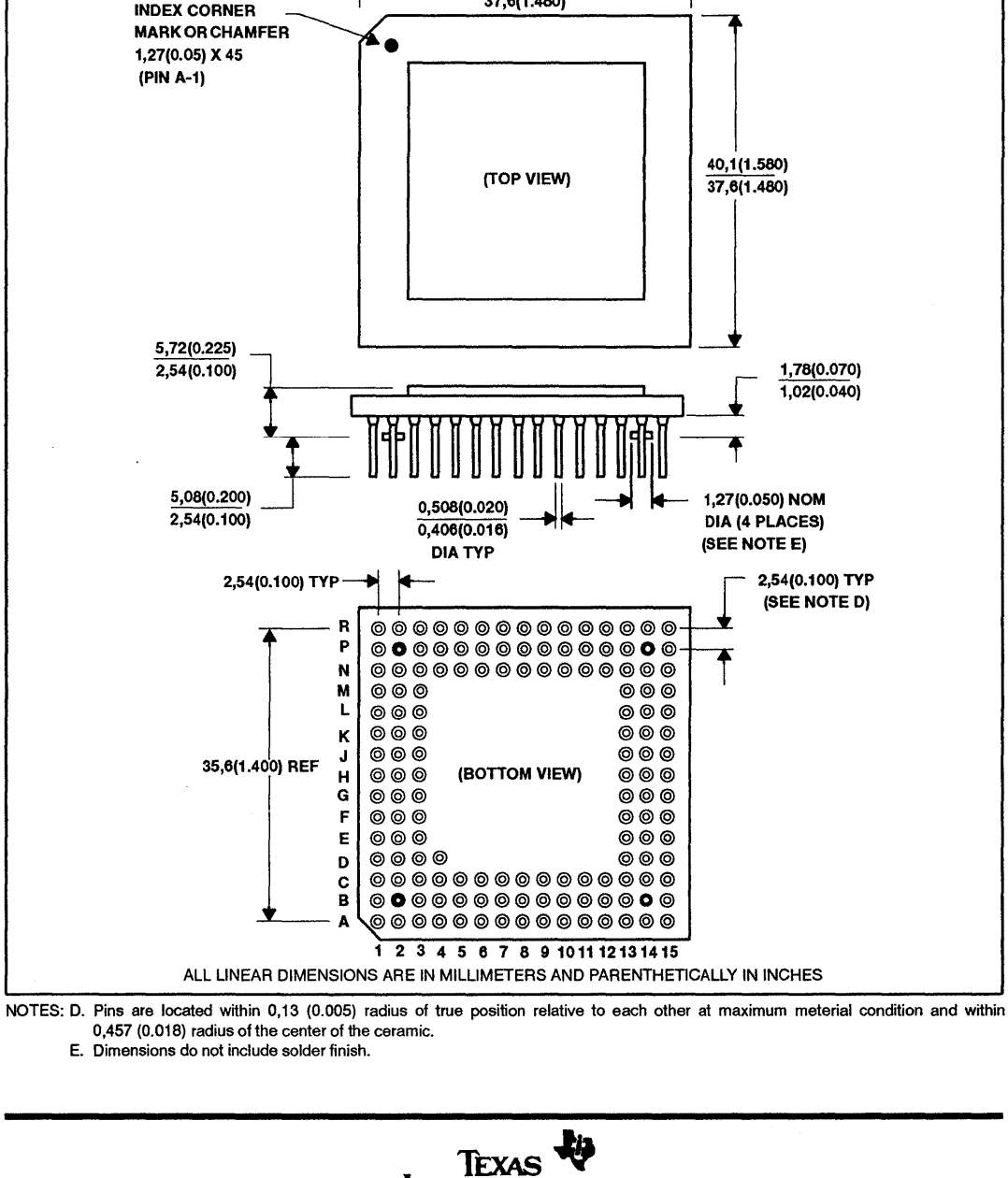
SMJ34082A ASSEMBLER OPCODE	DESCRIPTION OF ROUTINE	CYCLE COUNTS			
		PIPES2–1 11	PIPES2–1 10	PIPES2–1 01	PIPES2–1 00
RTS	Return from subroutine	1(1)	1(1)	1(1)	1(1)
SLL	Logical shift left A by B bits	1(1)	2(1)	2(1)	3(1)
SQRT	Square root of A				
	SP	11(11)	11(10)	12(10)	12(10)
	DP	16(16)	16(15)	17(15)	17(15)
	integer	20(20)	20(19)	21(19	21(19)
PASS.SUBRL	Multiply $A_1 * 1$, Subtract $B_2 - A_2$				
	SP	1(1)	2(1)	2(1)	3(1)
	DP	2(2)	3(2)	3(2)	4(2)
	integer	1(1)	2(1)	2(1)	3(1)
SRA	Arithmetic shift right A by B bits	1(1)	2(1)	2(1)	3(1)
SRL	Logical shift right A by B bits	1(1)	2(1)	2(1)	3(1)
ST	Store n words from register				
	SP	n + 1	n + 1	n + 1	n + 1
	DP	2n + 1	2n + 1	2n + 1	2n + 1
	integer	n + 1	n + 1	n + 1	n + 1
SUB	Subtract A – B	1(1)	2(1)	2(1)	3(1)
SUBRL	Subtract B – A	1(1)	2(1)	2(1)	3(1)
UTOD	Convert from unsigned integer to DP	1(1)	2(1)	2(1)	3(1)
UTOF	Convert from unsigned integer to SP	1(1)	2(1)	2(1)	3(1)
UWRAPI	Unwrap inexact operand	1(1)	2(1)	2(1)	3(1)
UWRAPR	Unwrap rounded operand	1(1)	2(1)	2(1)	3(1)
UWRAPX	Unwrap exact operand	1(1)	2(1)	2(1)	3(1)
WRAP	Wrap denormalized operand	1(1)	2(1)	2(1)	3(1)
XOR	Logical exclusive OR A, B	1(1)	2(1)	2(1)	3(1)

DP denotes double-precision, and SP denotes single-precision.

MECHANICAL DATA

GB pin-grid-array ceramic package

This is a hermetically sealed package.



NOTES: D. Pins are located within 0,13 (0.005) radius of true position relative to each other at maximum meterial condition and within 0,457 (0.018) radius of the center of the ceramic.

E. Dimensions do not include solder finish.

Appendix D

Maximizing Your MFLOPS with the TMS34082 and Motorola MC68030

This application report demonstrates one way that the TMS34082 floating-point processor can be coupled to a Motorola MC68030 microprocessor for high-performance and cost-effective, IEEE 74–1985 compatible, floating-point solutions.

Overview

Objectives

The TMS34082 Floating-Point Processor from Texas Instruments is a cost-effective, high-performance floating-point device. The objective of this application report is to demonstrate one way that the TMS34082 floating-point processor can be tightly coupled to a Motorola MC68030 microprocessor for high-performance and cost-effective, IEEE 754-1985 compatible, floating-point solutions. This application report is for Motorola MC680x0 users who interface to the VME/VSB bus, develop stand-alone systems, or who require fast floating-point processor solutions.

This document will show the simplicity and efficiency with which the TMS34082 interfaces with the Motorola MC68030 as a parallel floating-point processor. This report will also show the advanced floating-point capabilities of the TMS34082 compared to the Motorola MC6888X family.

Direct comparisons have been made between Motorola's coprocessor family and the TMS34082 Floating-Point Processor. Table 1 in the performance analysis section details a comparison of the TMS34082 and the Motorola MC68881. The results clearly show the increase in performance realized by choosing the TMS34082 as the host floating-point processor. By operating the TMS34082 in parallel with the Motorola MC68030, multiple operations can be processed simultaneously for enhanced performance.

When running the TMS34082 floating-point processor in parallel with the Motorola MC68030 as a host, the host processor must ensure the floating-point processor is always busy. In addition, the host processor must also have access to the floating-point processor's outputs and complete control for immediate stalls or interrupts. Details of the system architecture can be found in the System Architecture section.

TMS34082 Overview

The TMS34082 has features that are unique to floating-point processors. Some of these features are described below.

- Dual buses for accessing both data space and code space: This design allows you to download data over the LAD bus and transfer both instructions and data over the MSD bus, using the TMS34082's ability to simultaneously load instructions and operands over its two buses.
- Dynamic bus-switching: The CC pin can be triggered to affect an immediate jump to a preloaded address. Similar to an interrupt, this feature lets you jump straight to a routine in SRAM.
- Pipelining: The Harvard architecture within the TMS34082 allows pipelined data flow through the internal TMS34082 FPU, maximizing sustained throughput.
- Dynamic pipeline settings: Dynamic pipelining allows flexibility with data flow and feedbacks. Pipeline settings in the configuration register will direct feedback to registers, maximize throughput, or process vectors.
- FAST vs IEEE mode: The TMS34082 can function in fully IEEE 754-1985 compatible mode as well as in a mode that allows flushing all denormalizezd numbers to zero (FAST mode).
- Exception handling: The internal structure of the TMS34082 allows detection of status exceptions via software interrupts that generate address vectors to exception handling subroutines.

Internally, the TMS34082 has 22 onboard registers, which are well suited for matrix multiply graphic routines. It also has selectable data formats such as 32-bit integer, 32-bit floating-point, and 64-bit floating-point processors. In addition, there are internal programs for vector, matrix, and graphics operations.

The TMS34082s dual-bus structure gives you greater design flexibility. You can dynamically switch between the LAD and MSD buses while downloading instructions and data. Results are output on either the LAD or MSD bus.

System Architecture

System Overview

Memory mapping is chosen to interface the TMS34082 to the Motorola MC68030 in this design because it is direct and yields high-performance solutions. Furthermore, memory mapping allows the designer the flexibility to develop the floating-point processing interface around system memory.

Parallel processing provides the greatest throughput when coupling the TMS34082 to the Motorola MC68030 processor. In this design, the parallel processing tasks use buffers for data, instructions, and output (see Figure 1). The TMS34082 receives instructions and secondary data via the MSD bus from a dual-port SRAM (DP-SRAM). The dual-port SRAM has been preloaded by the Motorola MC68030. Primary data is obtained over the LAD bus through a FIFO buffer, which has also been preloaded by the Motorola MC68030.

Employing a FIFO buffer to download data to the LAD bus makes effective use of the Motorola MC68030's blocking loading capability, thus freeing the host processor for other functions. The LAD bus FIFO buffer can block load the TMS34082's internal registers with minimal overhead.

After receiving the data, the TMS34082 completes its calculations and writes its results into the dual-port SRAM buffer. To communicate when the calculations have been completed, the TMS34082 can interrupt the Motorola MC68030 and tell it to poll to the dual-port SRAM for output. Alternately, an optimizing compiler can set up boundary limits indicating when the DP-SRAM is full.

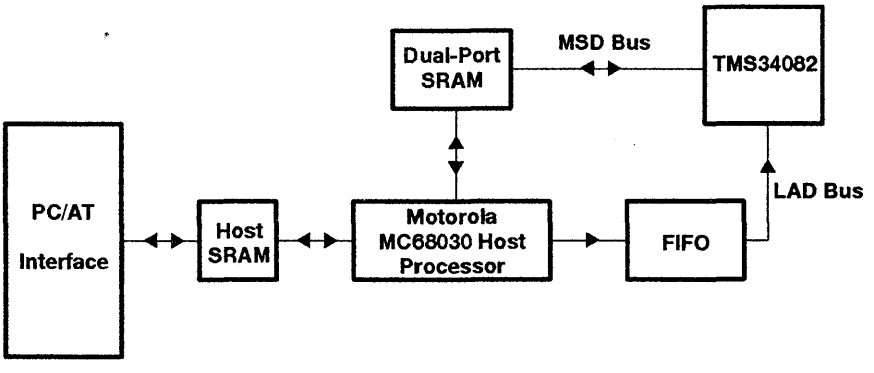


Figure 1. Motorola MC68030 Interface to the TMS34082 – Block Diagram

The system is initialized through a bootstrap loader program. The TMS34082 reads its start-up data through the LAD bus and transfers it via the MSD bus to the DP-SRAM. The first word of data is used to load the configuration register. After 65 clock cycles, the onboard program counter resets itself to 0 and reads from that address in the DP-SRAM.

The Motorola MC68030 receives its code and data from a dual-port, 8K × 32 SRAM. The SRAM information is uploaded from an PC/AT supervisory host through address and data buffers. The bus arbitration handshaking between the PC/AT bus and the Motorola MC68030 is accomplished by I/O mapping on the PC/AT.

Maximum throughput could be realized with an optimizing compiler by grouping functions and operands so that calculations can be pipelined and the registers can be loaded as a block.

As a download host, the IBM PC/AT is accessible to most users, allowing the duplication of this design with relative ease.

Objectives and Trade-Offs

The design objectives during the initial phases of the project were, in order of rank: performance, cost, size, and power. To maximize performance while keeping costs to a minimum, the following guidelines were used:

- Gain in performance should be commensurate with the gain in cost. In other words, a 10% increase in performance must be justified by no more than 10% gain in cost.
- A primary objective was to demonstrate the TMS34082's full capabilities by operating at maximum speed without wait states. This is accomplished by using parts that sufficiently meet the TMS34082 throughput requirements for maximum performance.

There are two schools of thought in processing floating-point operands. The first is to load all data from the Motorola MC68030 host through the FIFOs onto the LAD bus. Instructions and other data are loaded into the DP-SRAM, which the TMS34082 could access over the MSD bus. Results are then placed back into the DP-SRAM and read by the Motorola MC68030. This method is slightly faster, but requires a more sophisticated compiler.

The other approach is to toggle the CC signal to the TMS34082. CC is activated by setting the appropriate mask bit in the configuration register. Toggling CC signals the TMS34082 program loads an address vector over the LAD bus that points to an MSD address in external memory. The TMS34082 then executes the routine at this address. This example is useful when the DP-SRAM acts as a monitor and contains routines that are accessed frequently. An optimizing compiler would load relevant operands to the DP-SRAM or to TMS34082 internal registers and then point to a routine contained in DP-SRAM. The trade-off is that one clock cycle is lost in the jump process, but the compiler would have less overhead.

Software Description

Overview of Code Development

The objective of the software programs is to demonstrate the full capabilities of the TMS34082. Operands to the TMS34082 are represented in single-precision, double-precision, and integer formats.

Other features presented in these programs are:

- matrix operations,
- conversions between formats,
- arithmetic operations,
- vector processing,
- feedback operations,
- internal ROM routines,
- and block moves, making efficient use of the internal register set.

All of the resident software has been written in the processor's respective assembler language. Software driving the PC/AT is primarily written in C or assembly language.

Code development necessarily begins with the TMS34082. This then becomes the data code for the Motorola MC68030. Routines are written in Motorola MC68030 assembly language to handle data uploads to the FIFO, both uploads and downloads of data/instruction code to the DP-SRAM, and Motorola MC68030 host code resident in the 8K × 32 host SRAMs.

The initial code supplied to host SRAMs is transferred from the PC/AT. The resident Motorola MC68030 assembly language routines are translated from that format to one that the PC/AT recognizes.

The test software developed for this system writes and reads data from the host SRAM to test for correctness, address range functionality, and setup time validity. In addition, it allows thorough testing of the PC/AT bus and validation of host memory setup and hold times. The MS-DOS debugger is initially used for testing, while C code is implemented for more thorough test capabilities. In addition, the C code allows for ready upload and download of system software routines.

Big Endian, Little Endian

Programmers of this system must take into consideration the differences between Big Endian and Little Endian. The Motorola MC68030 device memory can be addressed on a byte-by-byte basis. The data for each byte in a 32-bit word (long word) is in order from most significant to least significant bit. But, the bytes are arranged in order of least significant to most significant (Little Endian). Intel microprocessors reverse their bytes as compared to Motorola processors. Intel arranges bytes from most significant to least significant (Big Endian). Figure 2 illustrates further details on byte arrangement. The hardware description, Appendix B, details more information on mixed implementation of Big Endian/Little Endian.

The PC/AT's backplane uses a different technique to address memory. A byte starts on an addressable byte boundary. A word consisting of two bytes starts on an arbitrary boundary, and the high byte corresponds to a high address (see Figure 2), while the low byte corresponds to a low address.

Code written for this design must take these data formats into consideration.

Motorola MC68030

Data				Address
Long Word $ 0000 0000				$ 0000 0000
Word $ 0000 0000		Word $ 0000 0002		
Byte $ 0000 0000	Byte $ 0000 0001	Byte $ 0000 0002	Byte $ 0000 0003	
Long Word $ 0000 0004				$ 0000 0004
Word $ 0000 0004		Word $ 0000 0006		
Byte $ 0000 0004	Byte $ 0000 0005	Byte $ 0000 0006	Byte $ 0000 0007	

Intel 80286

Data		Address
Word $ 00000		$ 00000
Byte $ 00001	Byte $ 00000	
Word $ 00002		$ 00002
Byte $ 00003	Byte $ 00002	

TMS34082

Data	Address
Long Word $ 0000	$ 0000
Long Word $ 0001	$ 0001

Figure 2. Data Organization in Memory

TMS34082 Code Development

Code for the TMS34082 includes a bootstrap loader, hardware test, and processor routines. During startup, routines confirm proper operation of all supporting hardware such as the SRAMs and FIFOs, to ensure the functioning of interfaces to the Motorola MC68030 host and to evaluate the accuracy of internal TMS34082 firmware.

A simple walking 1s and 0s is used to check the DP-SRAMs. The FIFOs can be checked with the bootstrap routine to verify that the proper data is being clocked through the device. In addition, the bootstrap also confirms LAD to MSD bus transfers. The bootstrap is enacted by the Motorola MC68030 by asserting the $\overline{\text{HALT}}$ and the $\overline{\text{INTR}}$ pins. (Consult the TMS34082 data sheet for bootstrap timing characteristics.)

The main software routine will make use of all the relevant internal instructions that demonstrate the TMS34082's processing capabilities. Two subprograms demonstrating the device's superior floating-point capabilities in processing matrix-multiply and transcendental functions are also included. Further, the TMS34082 can be reset either by the host processor, by the PC/AT, or manually.

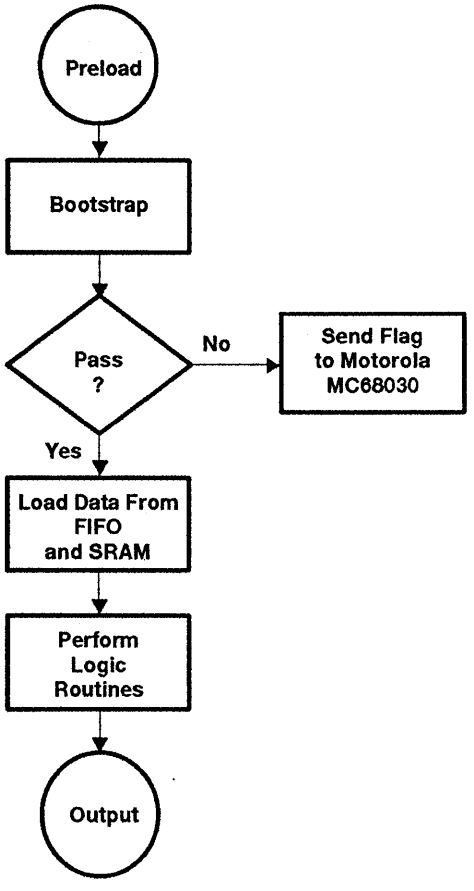


Figure 3. Block Diagram – TMS34082 Code

Motorola MC68030 Code Development

The Motorola MC68030 software is divided into six sections:

1. test,
2. read data/code from host SRAM,
3. output code to FIFOs and DP-SRAMs,
4. retrieve code from DP-SRAM,
5. and write data back to host SRAM.

Transferring data is relatively simple and can be seen in detail under the Software Listing section. The fundamental purpose of the test section is to check the DP-SRAM access and functionality from the Motorola MC68030 side. Checkout of the FIFOs has already been completed by the TMS34082 software. The host SRAM needs to be checked by the PC/AT before loading and by the Motorola MC68030 upon startup to verify correct dual access after arbitration (see Figure 4).

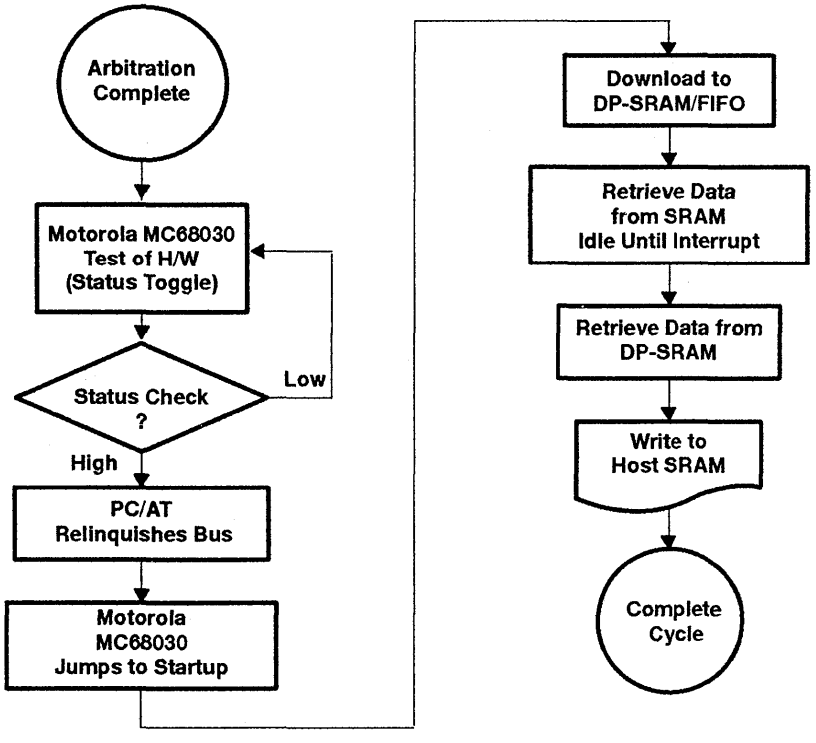


Figure 4. Block Diagram – Motorola MC68030 Code

Intel 80286 Code Development

The PC/AT code has five primary functions (see Figure 5):

1. to upload and download code to Motorola MC68030's host SRAM,
2. to test hardware,
3. to provide for a convenient development platform,
4. to perform as a supervisory controller,
5. and to establish communication with the host system.

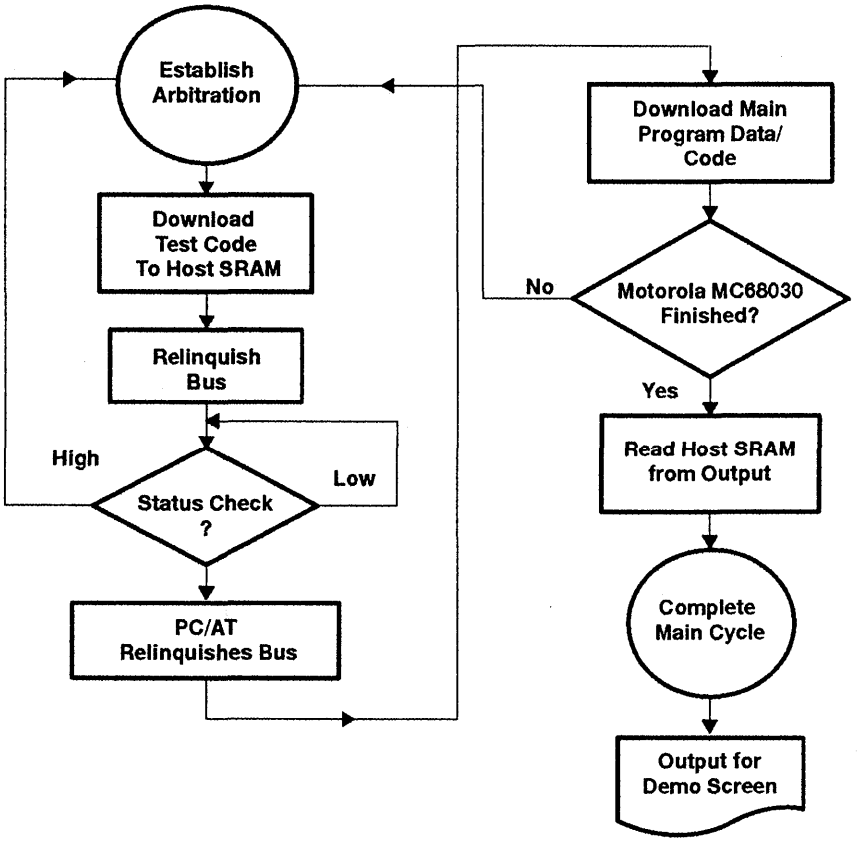


Figure 5. Block Diagram – PC/AT Code

Hardware Description

Overview

The board is an add-on card that fits into a 16-bit PC/AT slot. The speed at which the PC/AT bus accesses the board is not critical, since it acts as a supervisory host only. Its purpose is to transfer data to the 8K × 8 SRAM, control bus arbitration, and to read status signals on the card. See the enclosed schematics section for details.

The hardware is best described by breaking the board down into two subsystems: the PC/AT interface and the Motorola MC68030 subsystem.

PC/AT Interface

Two types of data are down loaded from the PC/AT to the Motorola MC68030: memory and I/O information. In the PC/AT, main memory is established for addresses 000000H to 07FFFFH (512K), I/O expansion ROM for locations 0C0000H–0DFFFFH, and prototype card I/O addresses for 300H–31FH. The software for this design makes use of all three of these memory ranges. All system development software is written in the 512K bytes of user memory space.

To down load data to the board's 8K words host SRAM an address in the middle of the PC/AT's I/O expansion ROM memory is chosen, 0D0000H. Thus, 8K words (32 bits wide) is placed in addresses 0D0000H–0D8000H (See Figure 6).

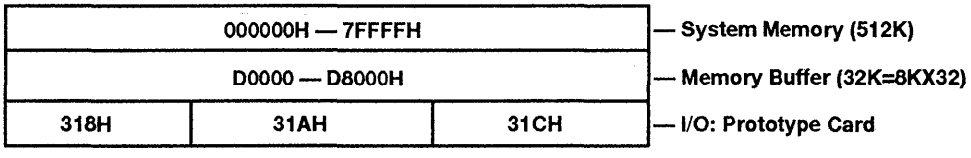


Figure 6. PC/AC Interface: I/O and Memory Addressing

Since the PC/AT system bus is 16 bits wide, it needs to match the 32-bit logic of the Motorola MC68030/TMS34082 system. Interface decode logic handles this by way of an odd/even address toggle, i.e. an even address indicates the lower 16 bits and an odd address indicates the upper 16 bits. Data must be loaded sequentially for this to function properly. An alternative would be to use the dynamic bus sizing capabilities of the Motorola MC68030, but this would require additional handshake logic and minimize real estate for future expansion plans.

The PC/AT Technical Reference Manual recommends that prototype I/O addresses lie between 310H and 31FH. Input/output data is configured to be read and written from address 318H. I/O is mainly used to control bus arbitration, to read status information, and to avoid address bus contention.

If you are only designing with the Motorola MC68030 and the TMS34082, then no design changes need to be made to compensate for byte addressing. However, if you prefer to implement the design as it is described in this report, byte addressing is of concern. While both Motorola's MC68030 and Intel's 80286 have their MSB at the leftmost position, the order in which the bytes are addressed is reversed, also known as a Big Endian/Little Endian format.

Two simple solutions present themselves. The first, a software solution, is to write code to reverse the byte order. The second, a hardware solution, is to simply reverse the data bits to match the byte order. For a prototype system such as this, a software solution is the preferred choice.

The PC/AT interface subsystem uses four PALs to handle address decoding for memory and I/O mapping, status acquisitions, and bus arbitration (See Figure 1).

Host Processor Interface

This design operates in purely synchronous mode, reducing the overhead logic required to notify the Motorola MC68030 of data size acknowledgements and reducing instruction overhead.

To assist the system designer in applying the TMS34082 to their Motorola MC68030-based system, the following guidelines have been used:

- Interrupts to the Motorola MC68030 are disabled by pulling the signals $\overline{IPL0}$, $\overline{IPL1}$, $\overline{IPL2}$ high. This implies that $\overline{AVEC}$ is tied high, which also simplifies synchronous operations.
- Occasionally, the TMS34082 and the Motorola MC68030 may attempt to access the same address location in the DP-SRAM, causing a collision. This contention is handled by having the DP-SRAM's BUSY flag pull the $\overline{BERR}$ and the $\overline{HALT}$ signals low simultaneously to delay the current cycle.
- Because this application always uses 32-bit data formats, $\overline{DSACK0}$ and $\overline{DSACK1}$ are pulled high to prevent assertion during synchronous operation with $\overline{STERM}$.
- $\overline{STERM}$ is decoded as a synchronous bus cycle terminator. This also reduces bus cycle delays due to misaligned transfers as they are always 32 bits wide.
- Since this project employs relatively fast SRAMs, external cache is not needed and Motorola MC68030 internal cache is not used. Therefore, $\overline{CIN}$, $\overline{CDIS}$, and $\overline{CBACK}$ are tied high. This also assists in stabilizing the setup and hold times during periods when $\overline{AS}$ is asserted during sychronous operations.
- The memory management features of the Motorola MC68030 are not used. Consequently, $\overline{MMUDIS}$ is tied high.
- Arbitration between the PC/AT bus and the Motorola MC68030's bus is handled by onboard PAL logic.

Memory Addressing has been encoded as follows: host SRAM accesses at 00008000H, DP-SRAM accesses at 00010000H, and FIFO accesses at location 00020000H. Thus, address bits 15, 16, and 17 can be used for each individual memory access.

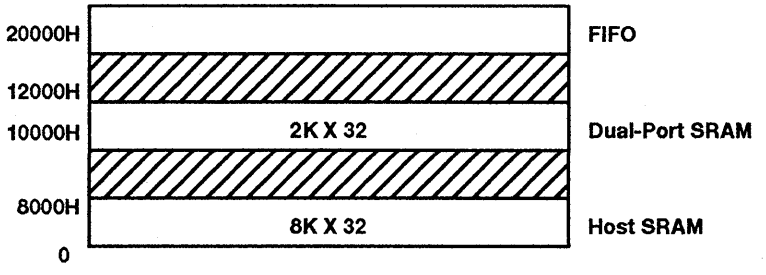


Figure 7. Motorola MC68030 Interface: Memory Addressing

TMS34082 as a Parallel Processor

Interfacing to the TMS34082 is simple and direct. This project emphasizes a design approach that requires minimal support hardware. By coupling a FIFO buffer directly to the LAD bus, external address latching and decoding is not required. From the MSD bus, the TMS34082 is directly coupled to the DP-SRAM, further reducing the decode hardware.

The data/code space is directly linked to address locations 0000H–07FFH and could be expanded to 64K words as required. For further information regarding pin definitions and electrical characteristics, please refer to the TMS34082 data sheet.

Performance Analysis

To accurately compare the performance of two coprocessors produced by two different manufacturers, it is essential to incorporate commonalties. A preliminary analysis has been completed that compares execution times of functions that are similar to the Motorola MC68881 and the TMS34082 (see Table 1).

The TMS34082 typically speeds execution by 30-40 times. This does not take into consideration effective addressing, overlap, and pipelining, which widens the gap between the TMS34082's execution times and those of the Motorola coprocessor family. Detailed calculations are available upon request.

Table 1. Performance Comparison Chart

Generic Instruction	Format/ Precision	TMS34082		Motorola MC68881	
		TI Instruction Syntax	Execution Time	Motorola Instruction Syntax	Execution Time
Add	Integer	ADD	2	FADD	80
	Single	ADDF	2		72
	Double	ADDD	2		78
Divide	Integer	DIV	16	FDIV	132
	Single	DIVF	7		124
	Double	DIVD	13		130
1s Complement	Integer	NOT	2	FCMP	62
	Single		2		54
	Double		2		60
Absolute Value	Integer	ABS	2	FABS	62
	Single		2		54
	Double		2		60
Negate	Integer	NEG	2	FNEG	62
	Single		2		54
	Double		2		60
Multiply	Integer	MPY	2	FMUL	100
	Single	MPYF	2		92
	Double	MPYD	3		98
Square Root	Integer	SQRT	20	FSQRT	134
	Single	SQRTF	10		126
	Double	SQRTD	16		132
Subtract	Integer	SUB	2	FSUB	80
	Single	SUBF	2		72
	Double	SUBD	2		78

System Information — Parts List

Table 2. Parts List

Reference Designation	Name	Pins	WIDTH (MILS)
U1, U2	PAL20L8	24	300
U3	PAL16RA8	20	300
U4	PAL16R4	20	300
U5, U6, U7, U8	74BCT245	20	300
U9	74AS74	14	300
U10	Motorola MC68030	13 x 13	PGA
U11, U12	74F08	14	300
U13	PAL22V10	24	300
U14	74F08	14	300
U15, U16, U17, U18	7C185	28	300
U19	IDT7132	48	600
U20, U21, U22	IDT7142	48	600
U23, U24, U25, U26	74ALS2232	24	300
U27	74AS74	14	300
U28	TMS34082	15 x 15	PGA
U29	74BCT244	20	300
U30	74F74	24	300
U31, U32, U33	74BCT244	20	300
U34	74F74	14	300
U35	74F08	14	300
U36	74F374	20	300
U37	74F08	14	300
U38	74F00	14	300
RP1, RP2, RP3	10K Pull-up Resistor	9	SIP
HDR1	Platform	16	300
C1-41	0.01 µF Capacitor		
C42	10 µF Capacitor		
X1	25 MHZ Oscillator	14	300
X2	40 MHZ Oscillator	14	300
SW1	SPST Switch		
SW2	SPDT Switch		
SW3	8 POS. DIP SW	16	300

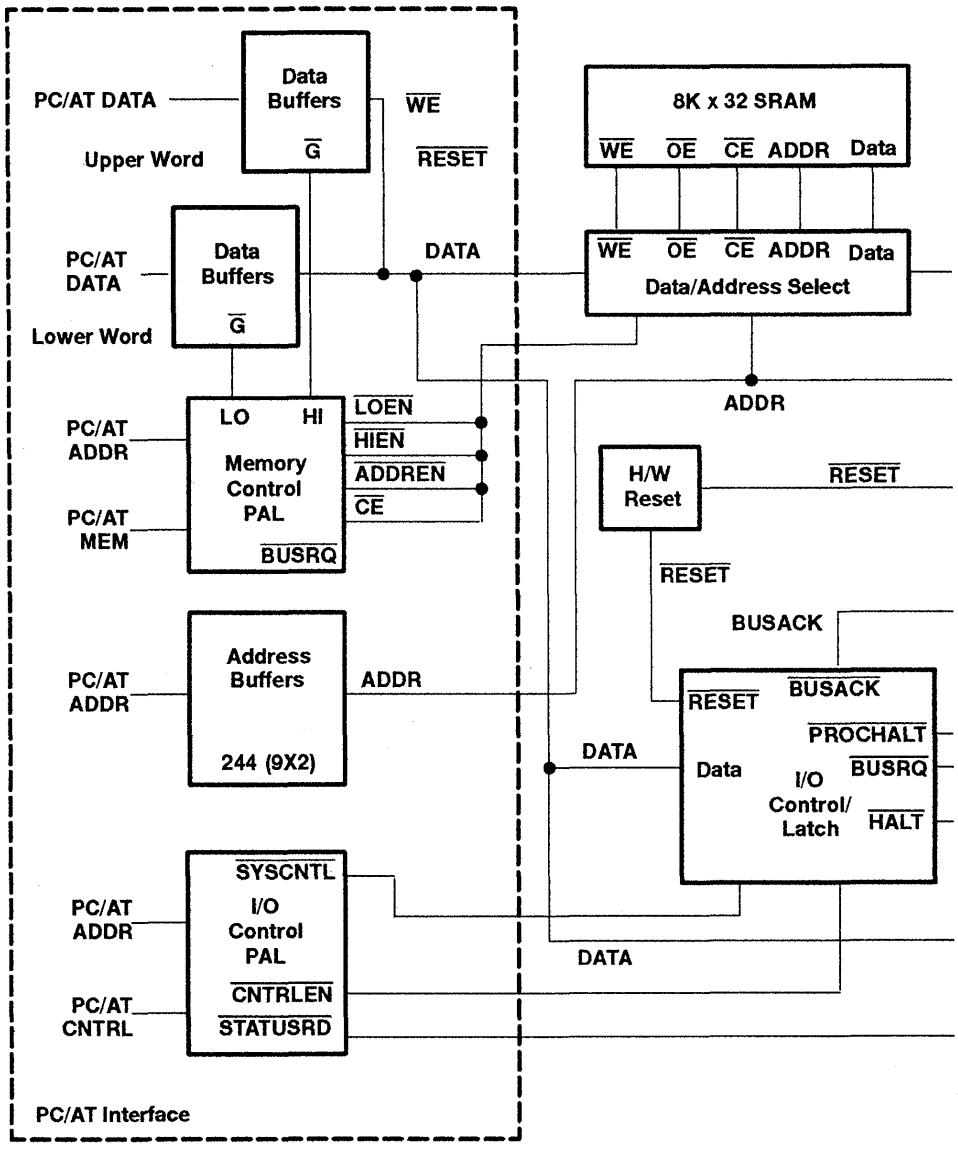


Figure 8(a). Block Diagram

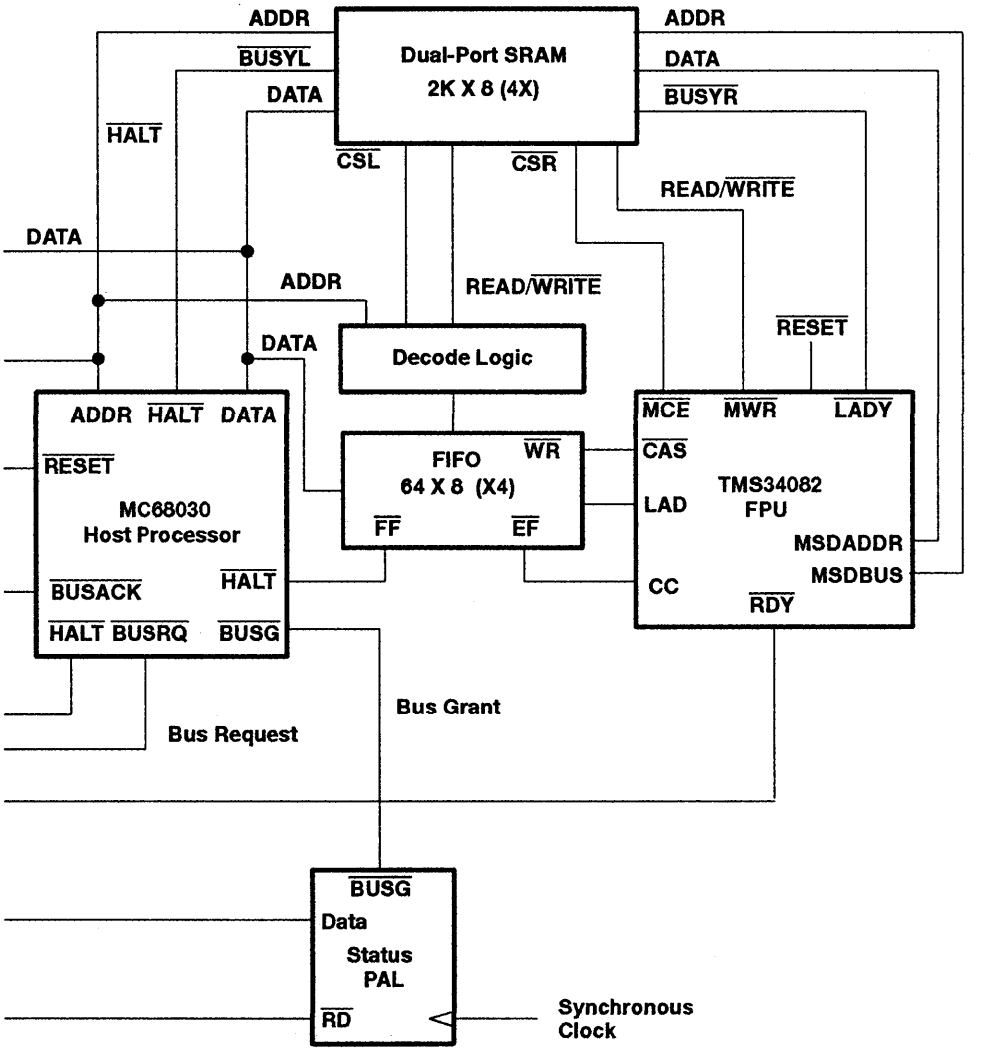


Figure 8(b). Block Diagram

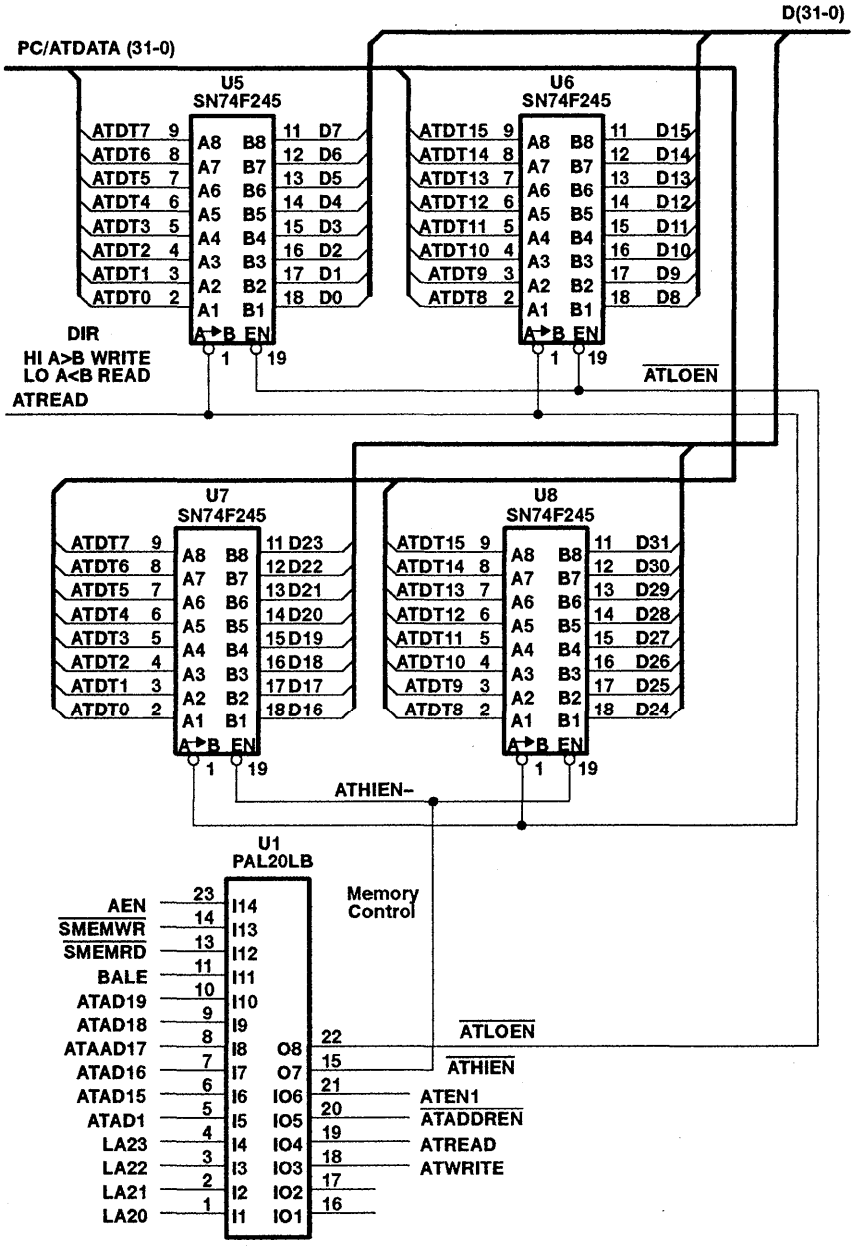


Figure 9. PC/AT I/F and Control, Details of U1, U5, U6, U7, and U8

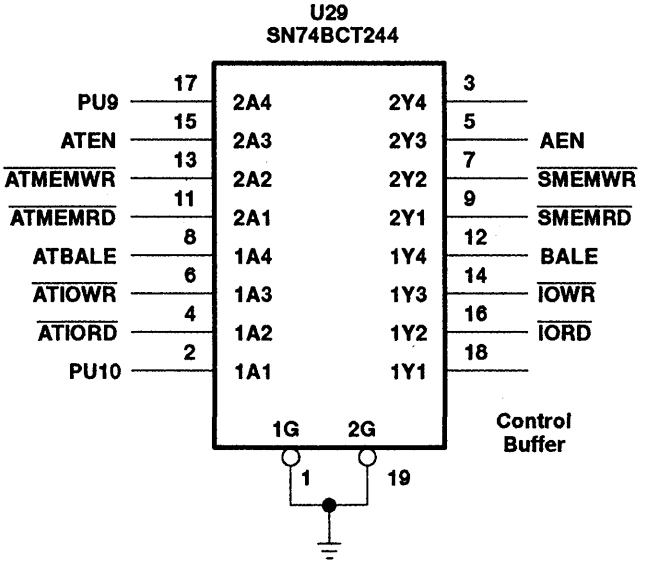


Figure 10. PC/AT I/F and Control, Details of U29

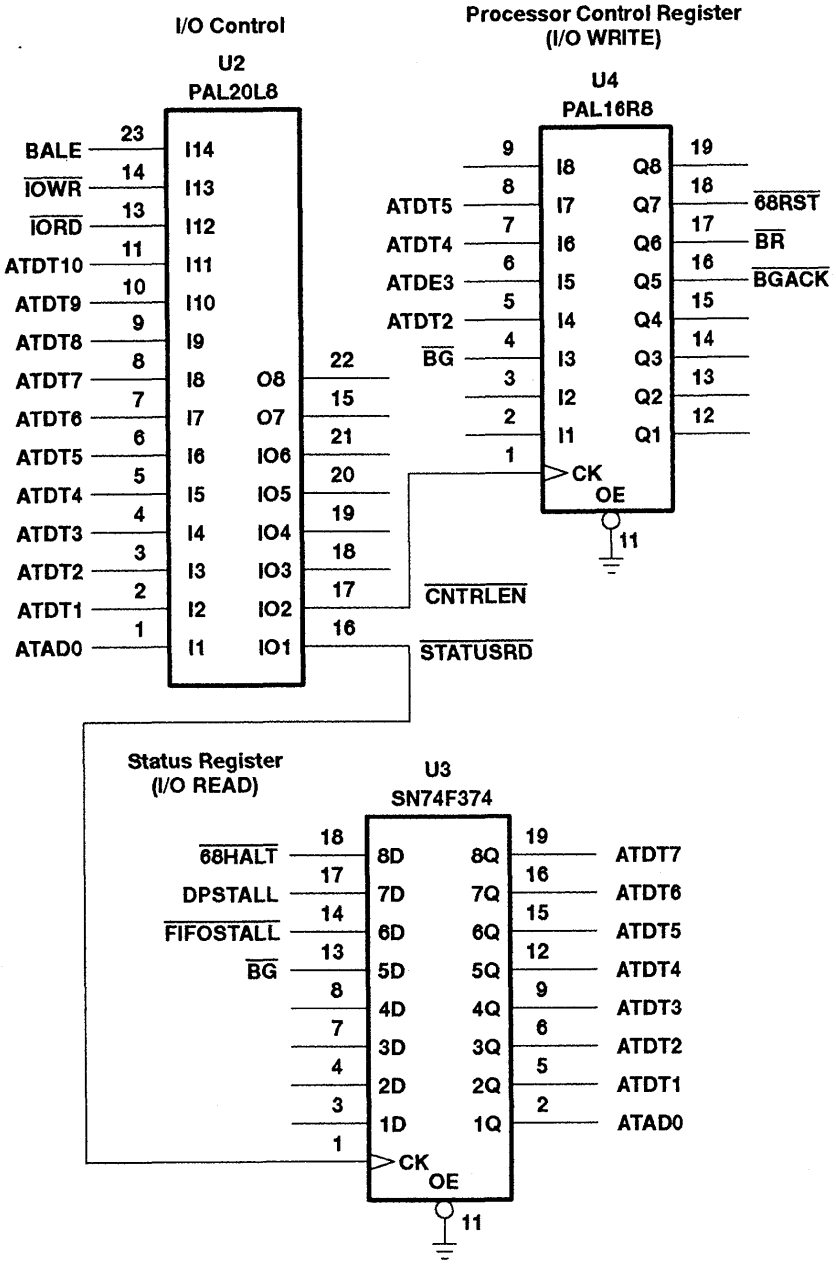


Figure 11. PC/AT I/F and Control, Details of U2, U3, and U4

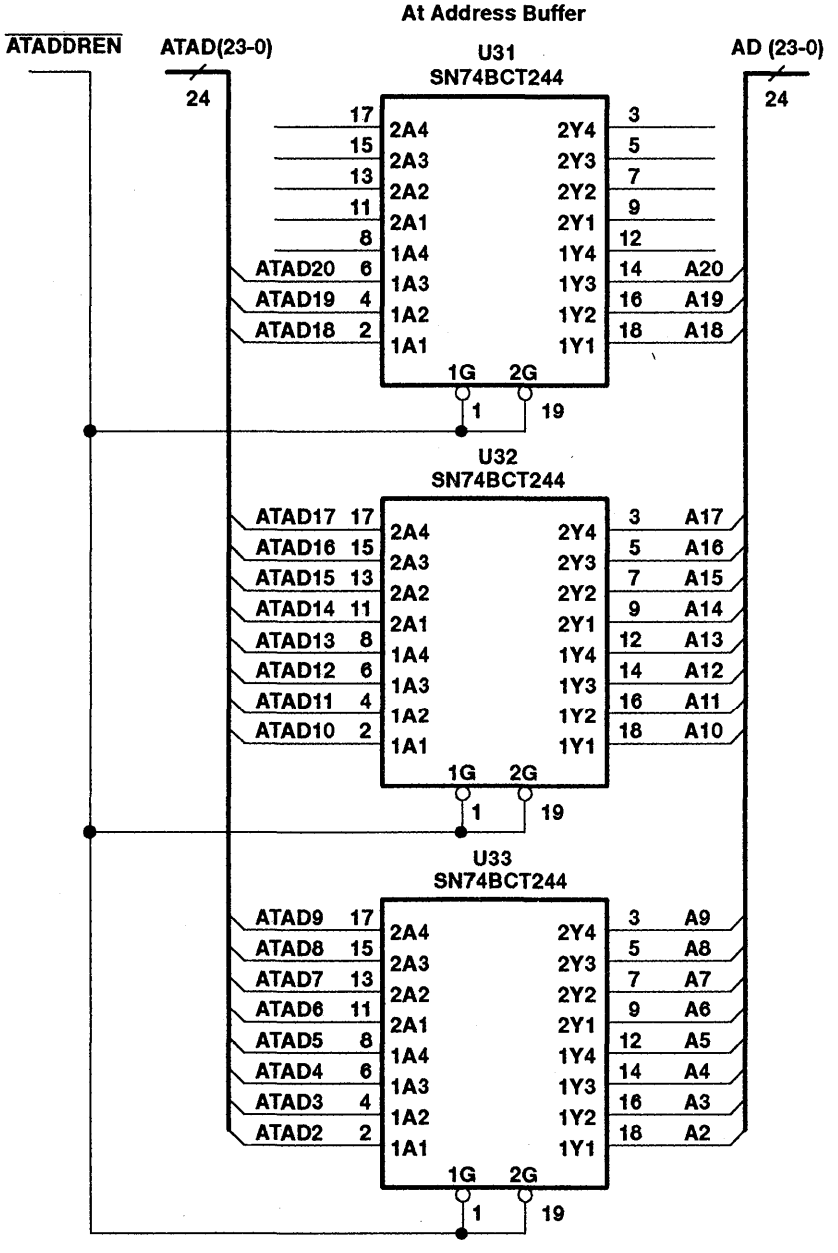


Figure 12. Motorola MC68030 and Address Buffers, Details of U31, U32, and U33

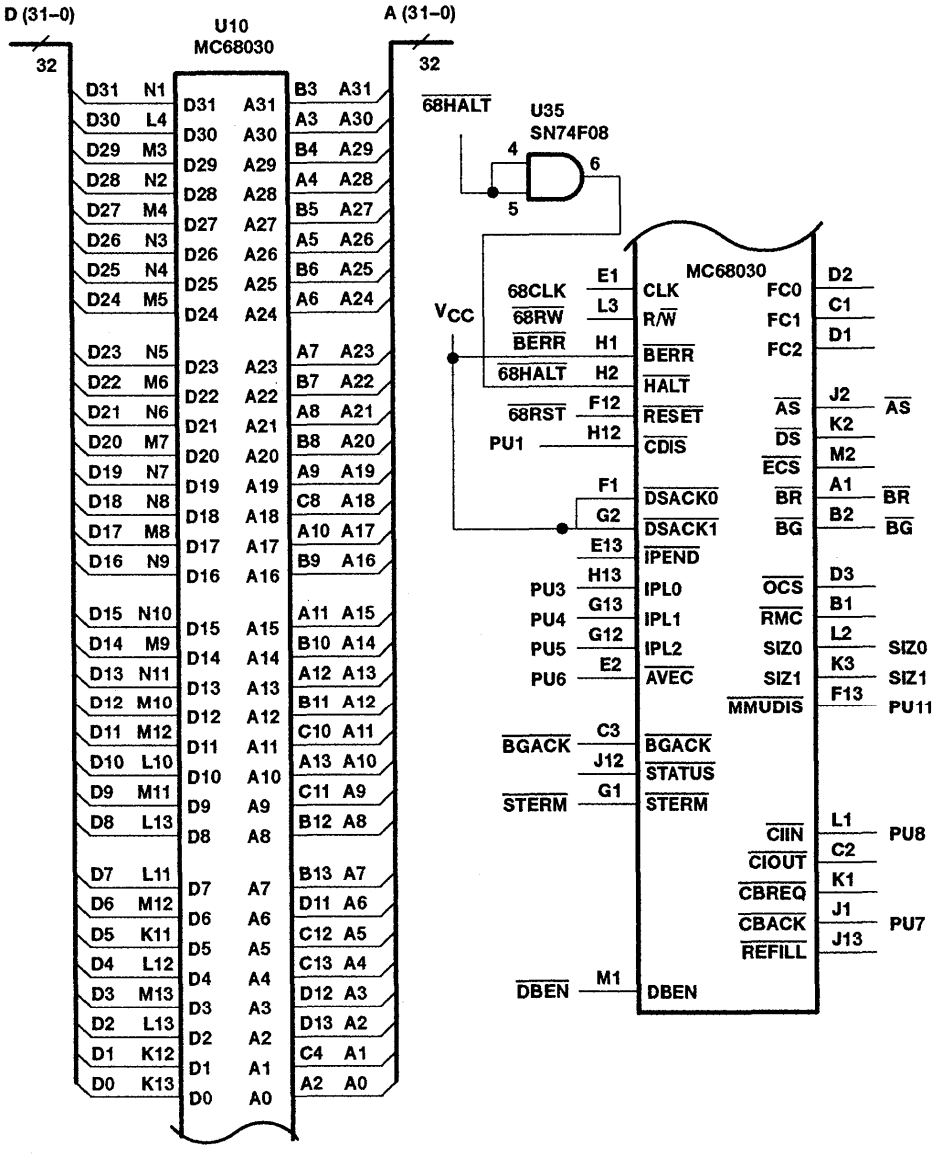


Figure 13. Motorola MC68030 and Address Buffers, Details of U10

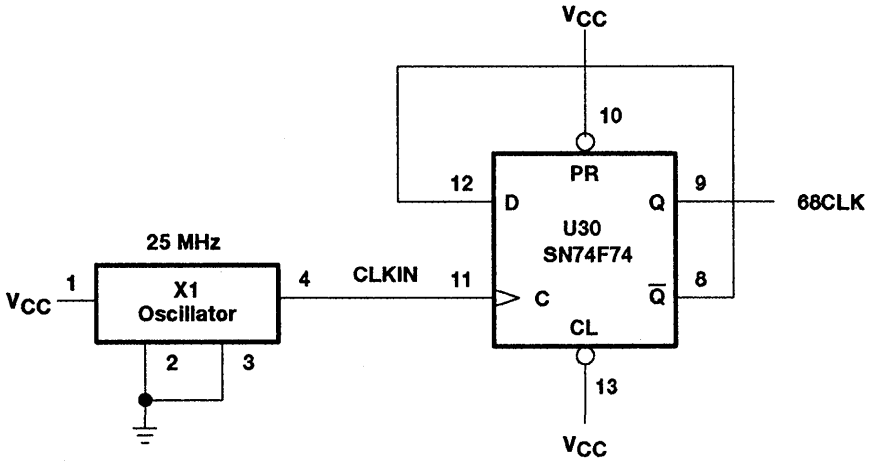


Figure 14. Motorola MC68030 and Address Buffers, Details of Oscillator and U30

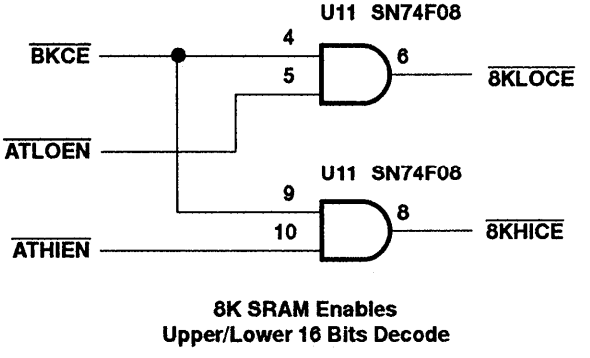


Figure 15. Motorola MC68030 Decode/Control, Details of U11

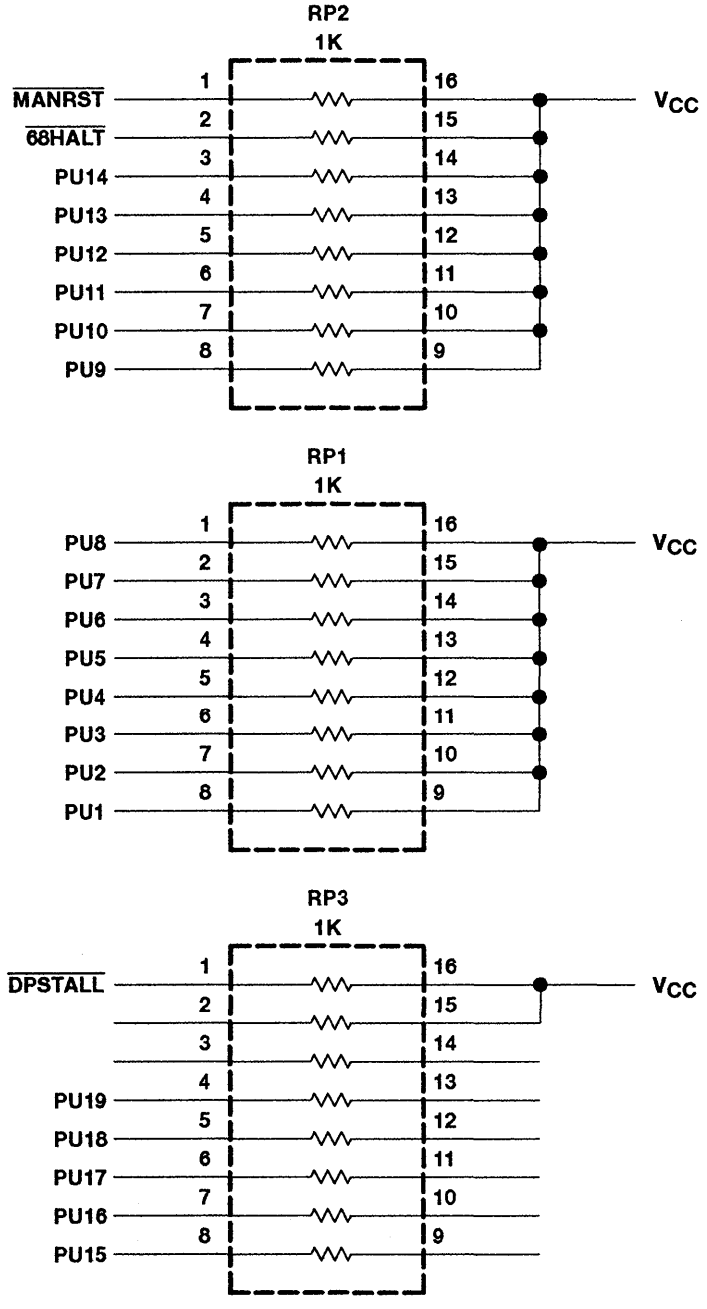


Figure 16. Motorola MC68030 Decode/Control, Details of RP1, RP2, and RP3

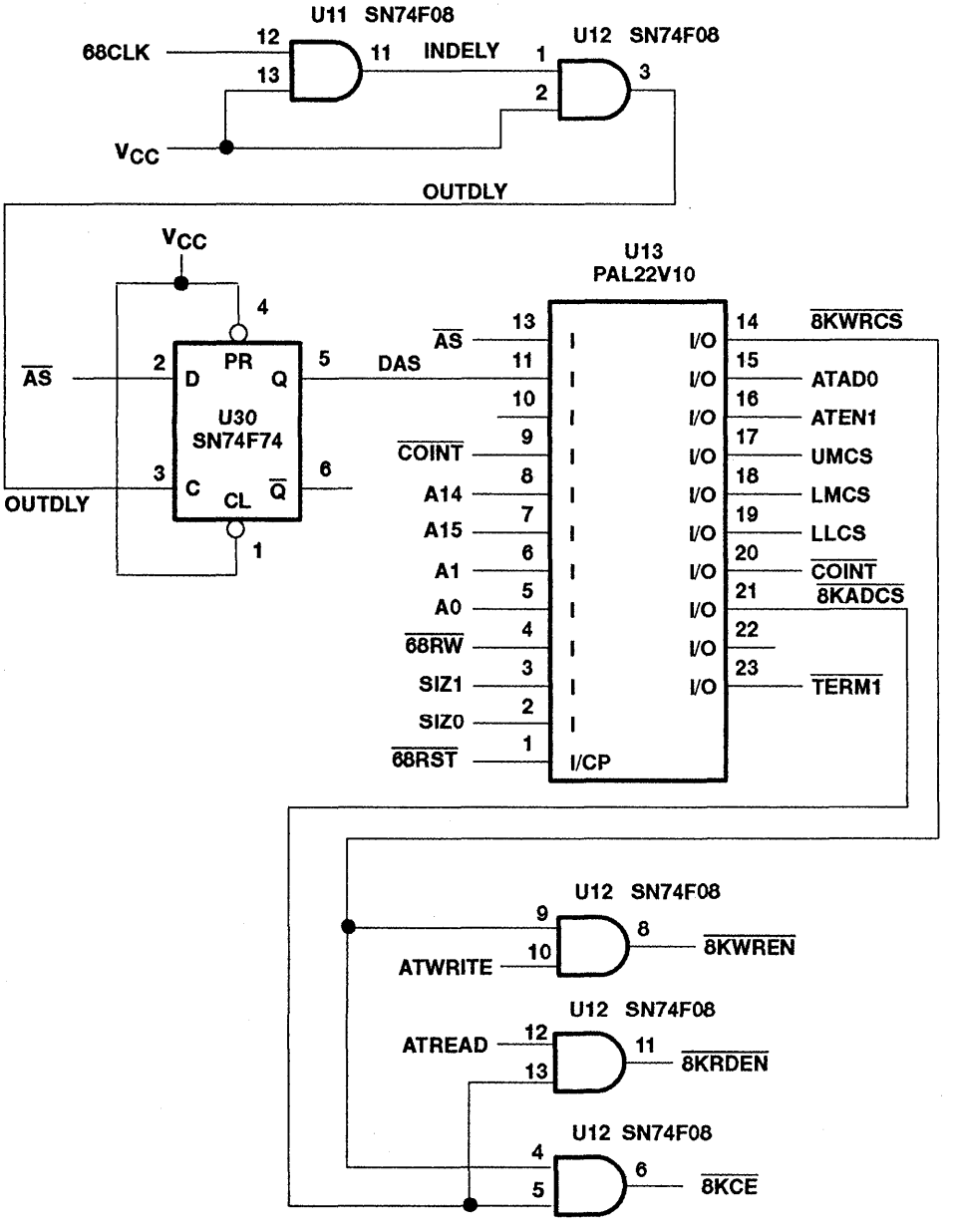


Figure 17. Motorola MC68030 Decode/Control, Details of U11, U12, U13, and U30

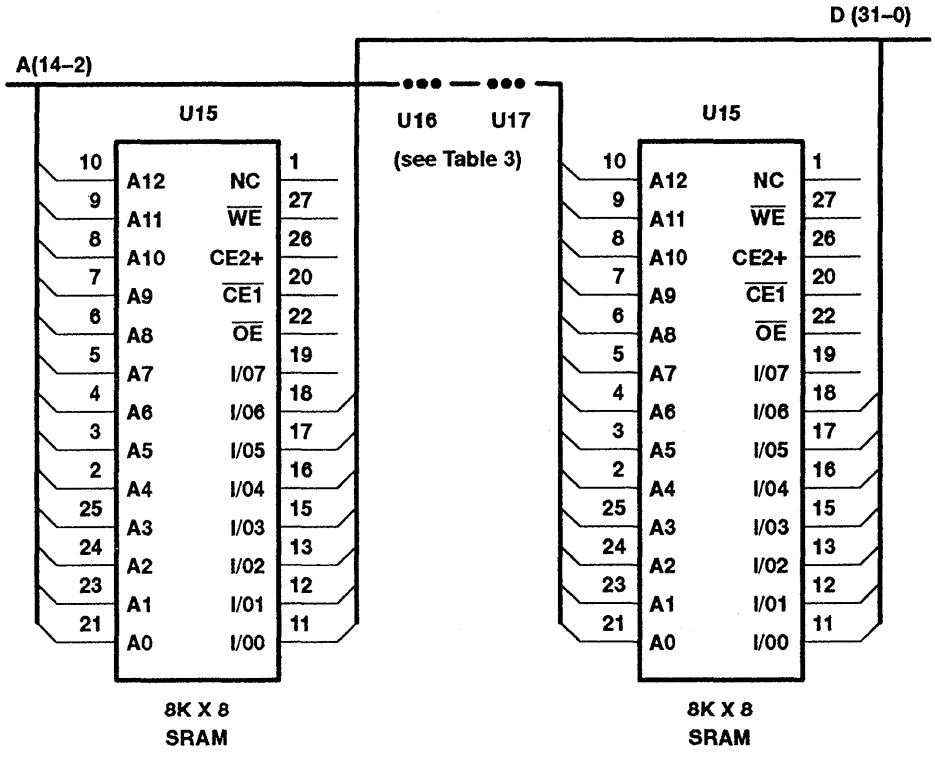


Figure 18. 8K x 8 SRAM DP-SRAM, Details of U15, U16, U17, and U18

Table 3. 8K x 8 SRAM DP-SRAM, Detail Pin Assignments for U15, U16, U17, and U18

Device		8K x 8 SRAM			
Block Number		U15	U16	U17	U18
Pin Name	Pin Number	External Connection Signal Name			
A0	21	A2	A2	A2	A2
A1	23	A3	A3	A3	A3
A2	24	A4	A4	A4	A4
A3	25	A5	A5	A5	A5
A4	2	A6	A6	A6	A6
A5	3	A7	A7	A7	A7
A6	4	A8	A8	A8	A8
A7	5	A9	A9	A9	A9
A8	6	A10	A10	A10	A10
A9	7	A11	A11	A11	A11
A10	8	A12	A12	A12	A12
A11	9	A13	A13	A13	A13
A12	10	A14	A14	A14	A14
I/O0	11	D0	D8	D16	D24
I/O1	12	D1	D9	D17	D25
I/O2	13	D2	D10	D18	D26
I/O3	15	D3	D11	D19	D27
I/O4	16	D4	D12	D20	D28
I/O5	17	D5	D13	D21	D29
I/O6	18	D6	D14	D22	D30
I/O7	19	D7	D15	D23	D31
$\overline{OE}$	22	$\overline{8KRDEN}$	$\overline{8KRDEN}$	$\overline{8KRDEN}$	$\overline{8KRDEN}$
$\overline{CE1}$	20	LLCS	LMCS	$\overline{UMCS}$	UUCS
CE2+	26	V_{CC}	V_{CC}	V_{CC}	V_{CC}
$\overline{WE}$	27	$\overline{8KWREN}$	$\overline{8KWREN}$	$\overline{8KWREN}$	$\overline{8KWREN}$
NC	1	No Connection			

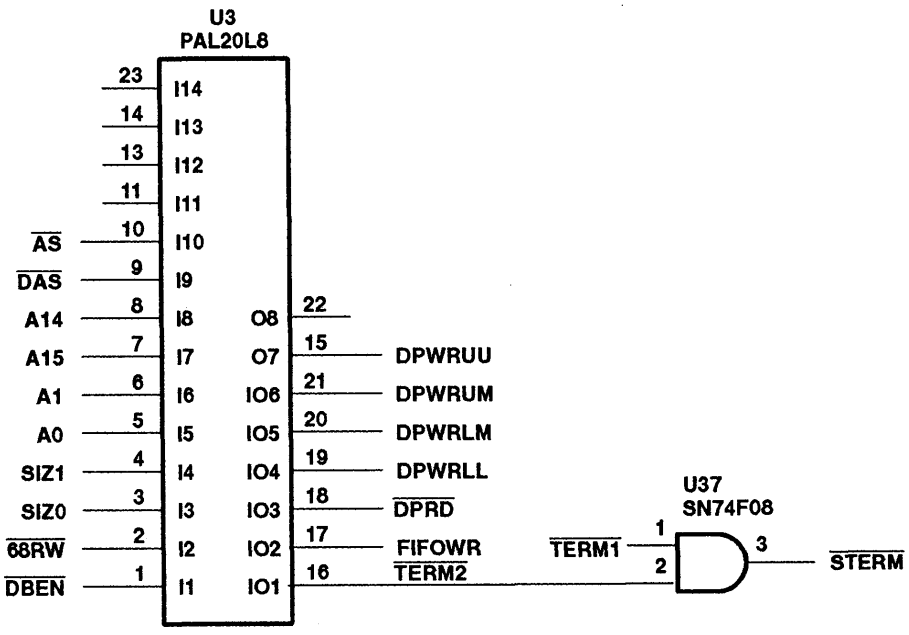


Figure 19. Motorola MC68030 Decode/Control, Details of U3 and U37

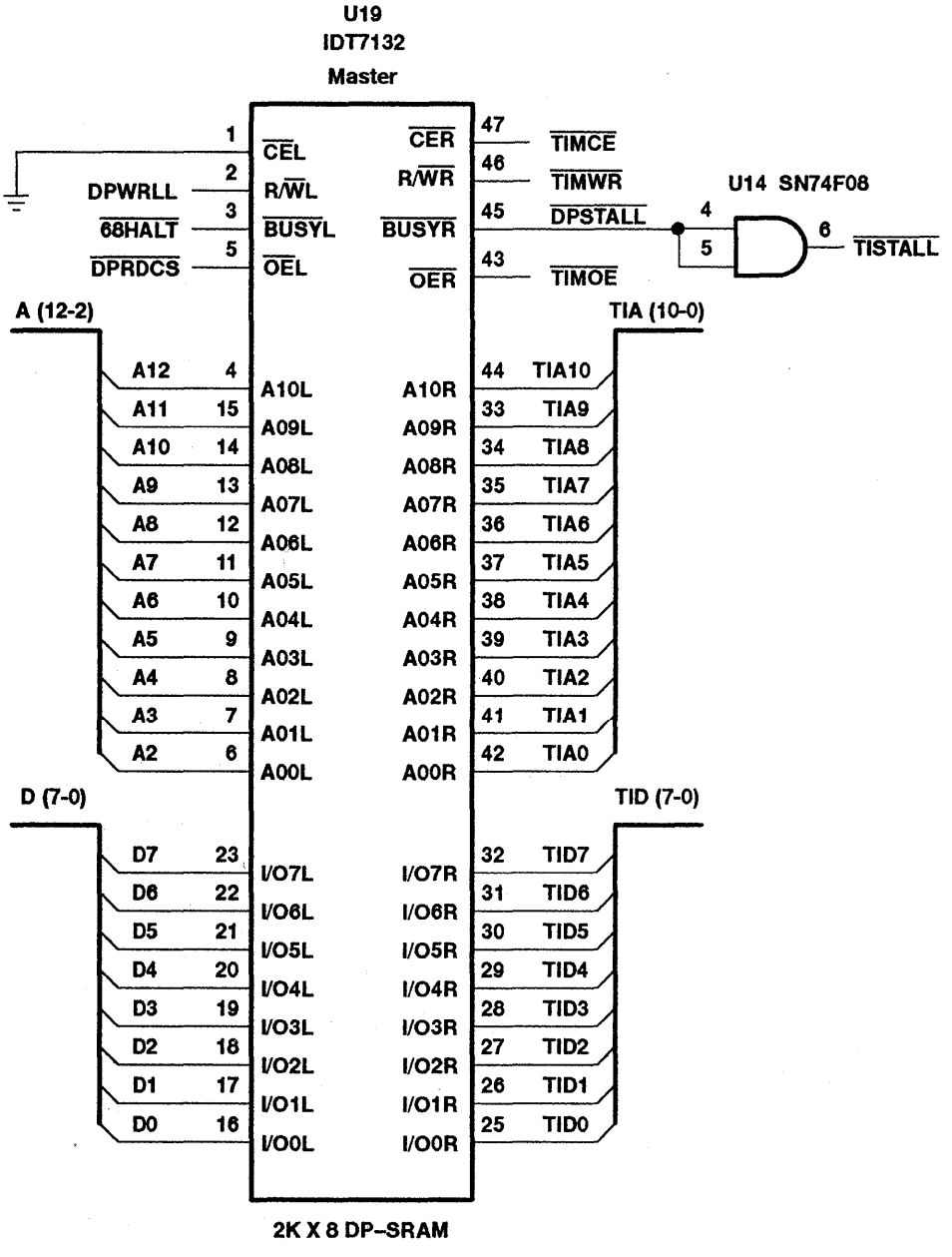


Figure 20. 8K x 8 SRAM DP-SRAM, Details of U14 and U19

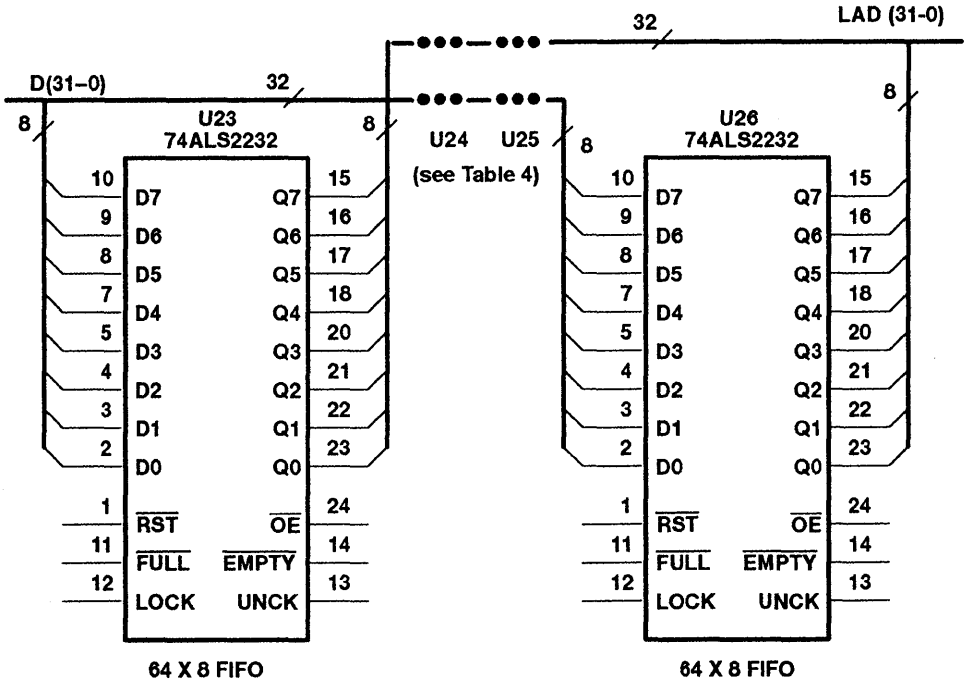


Figure 21. FIFO Logic, Details of U23, U24, U25, and U26

Table 4. FIFO Logic, Detail Pin Assignments for U23, U24, U25, and U26

Device		SN74ALS2232			
Block Number		U23	U24	U25	U26
Pin		External Connection Signal Name			
Name	Number				
D0	2	D0	D8	D16	D24
D1	3	D1	D9	D17	D25
D2	4	D2	D10	D18	D26
D3	5	D3	D11	D19	D27
D4	7	D4	D12	D20	D28
D5	8	D5	D13	D21	D29
D6	9	D6	D14	D22	D30
D7	10	D7	D15	D23	D31
$\overline{\text{RST}}$	1	$\overline{\text{RESET}}$	$\overline{\text{RESET}}$	$\overline{\text{RESET}}$	$\overline{\text{RESET}}$
$\overline{\text{FULL}}$	11	$\overline{\text{68HALT}}$	$\overline{\text{68HALT}}$	$\overline{\text{68HALT}}$	$\overline{\text{68HALT}}$
LOCK	12	$\overline{\text{FIFOWR}}$	$\overline{\text{FIFOWR}}$	$\overline{\text{FIFOWR}}$	$\overline{\text{FIFOWR}}$
Q0	23	LAD0	LAD8	LAD16	LAD24
Q1	22	LAD1	LAD9	LAD17	LAD25
Q2	21	LAD2	LAD10	LAD18	LAD26
Q3	20	LAD3	LAD11	LAD19	LAD27
Q4	18	LAD4	LAD12	LAD20	LAD28
Q5	17	LAD5	LAD13	LAD21	LAD29
Q6	16	LAD6	LAD14	LAD22	LAD30
Q7	15	LAD7	LAD15	LAD23	LAD31
$\overline{\text{OE}}$	24	$\overline{\text{COINT}}$	$\overline{\text{COINT}}$	$\overline{\text{COINT}}$	$\overline{\text{COINT}}$
$\overline{\text{EMPTY}}$	14	$\overline{\text{FIFOSTAL}}$	$\overline{\text{FIFOSTAL}}$	$\overline{\text{FIFOSTAL}}$	$\overline{\text{FIFOSTAL}}$
UNCK	13	$\overline{\text{FIFORD}}$	$\overline{\text{FIFORD}}$	$\overline{\text{FIFORD}}$	$\overline{\text{FIFORD}}$

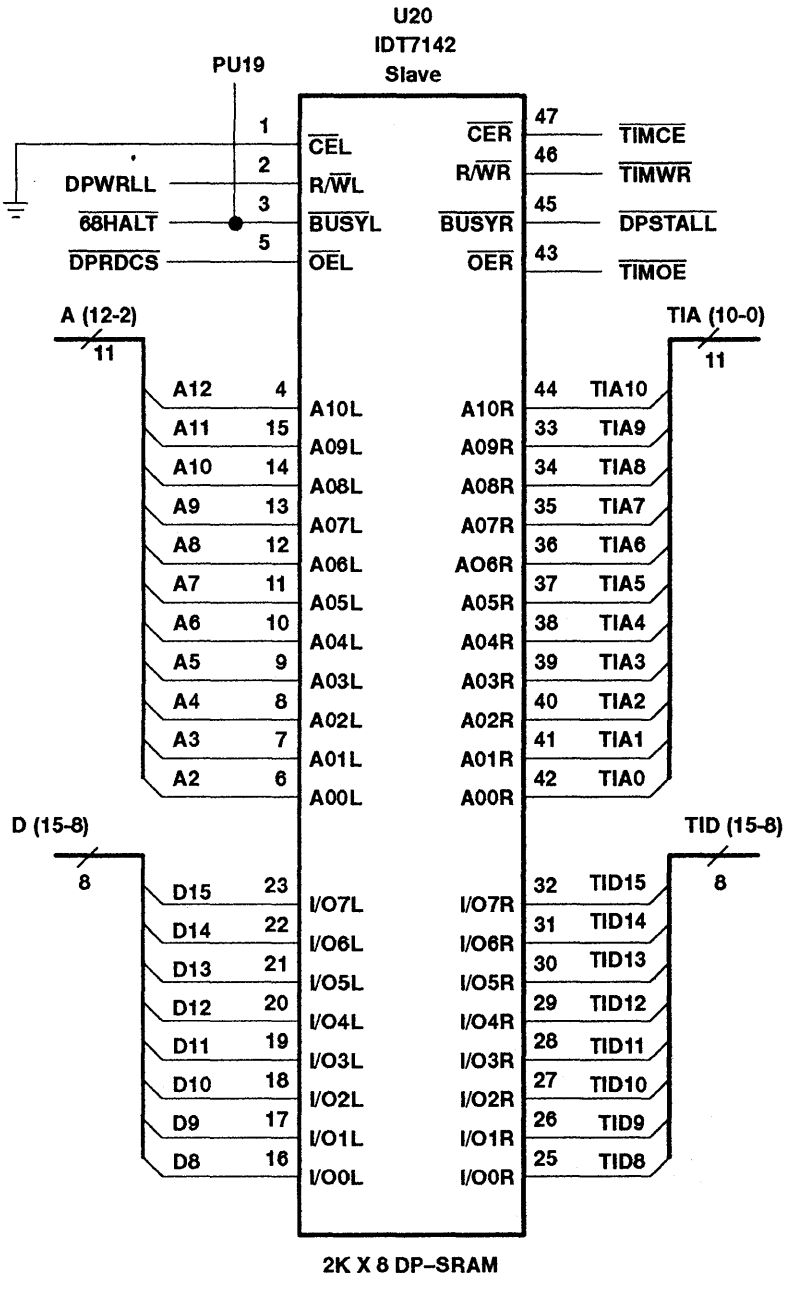


Figure 22. 8K x 8 SRAM DP-SRAM, Details of U20

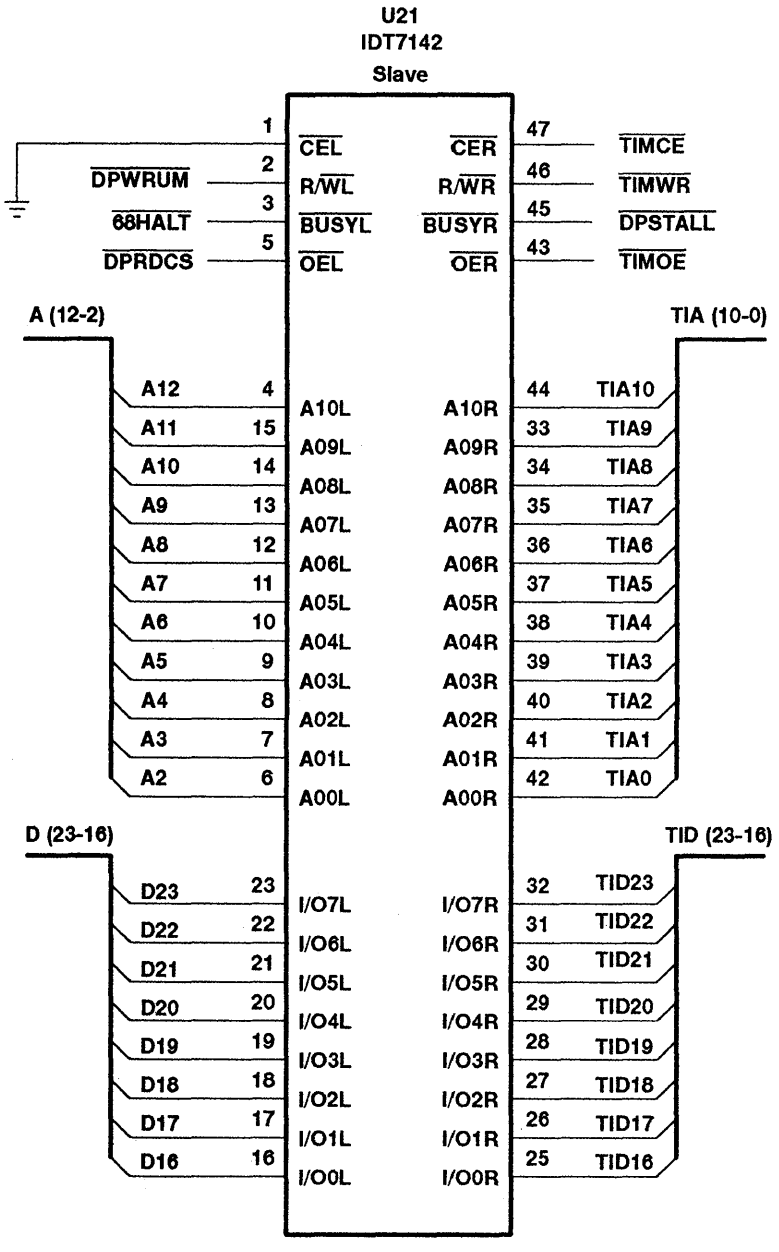


Figure 23. FIFO Logic, Details of U21

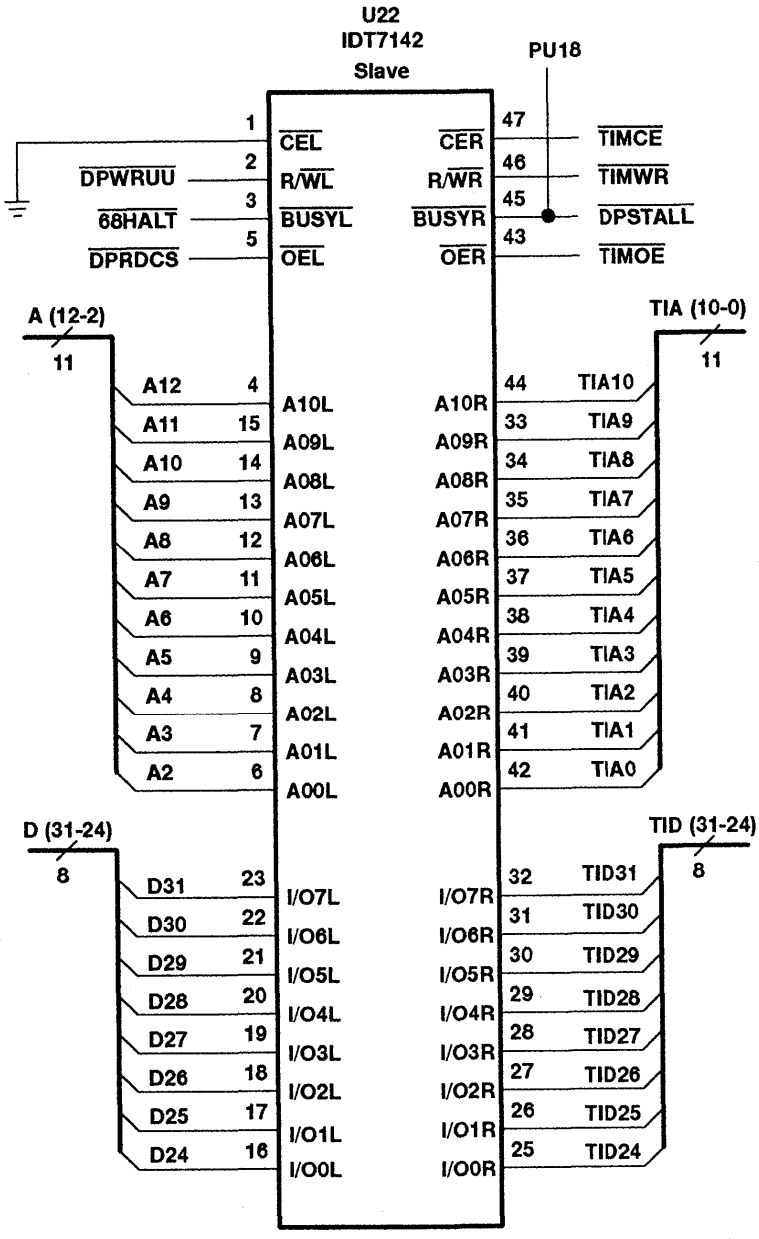


Figure 24. FIFO Logic DP-SRAM, Details of U22

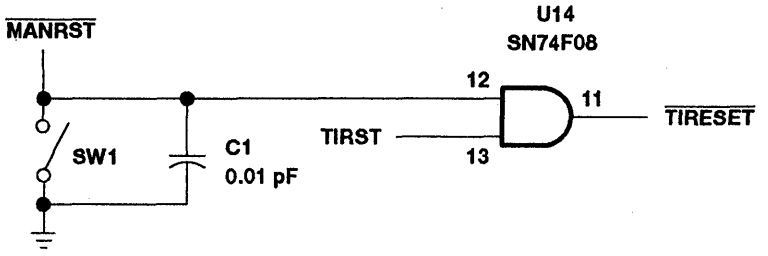


Figure 25. FIFO Logic, Details of U14

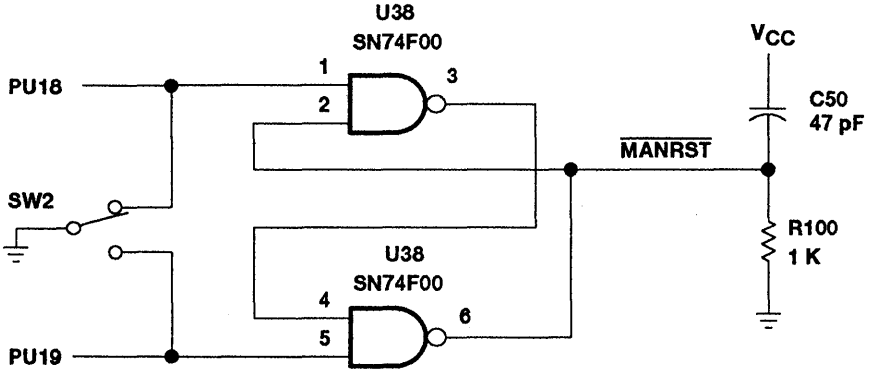


Figure 26. TMS34082, Details of U38

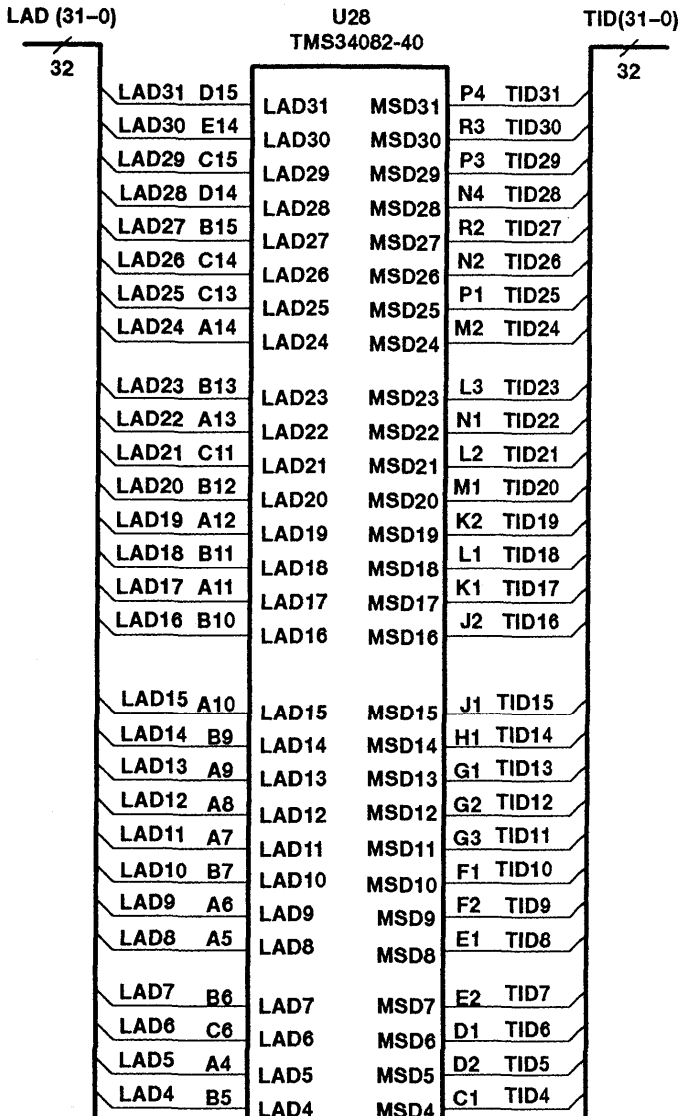


Figure 27. TMS34082, Details of U28

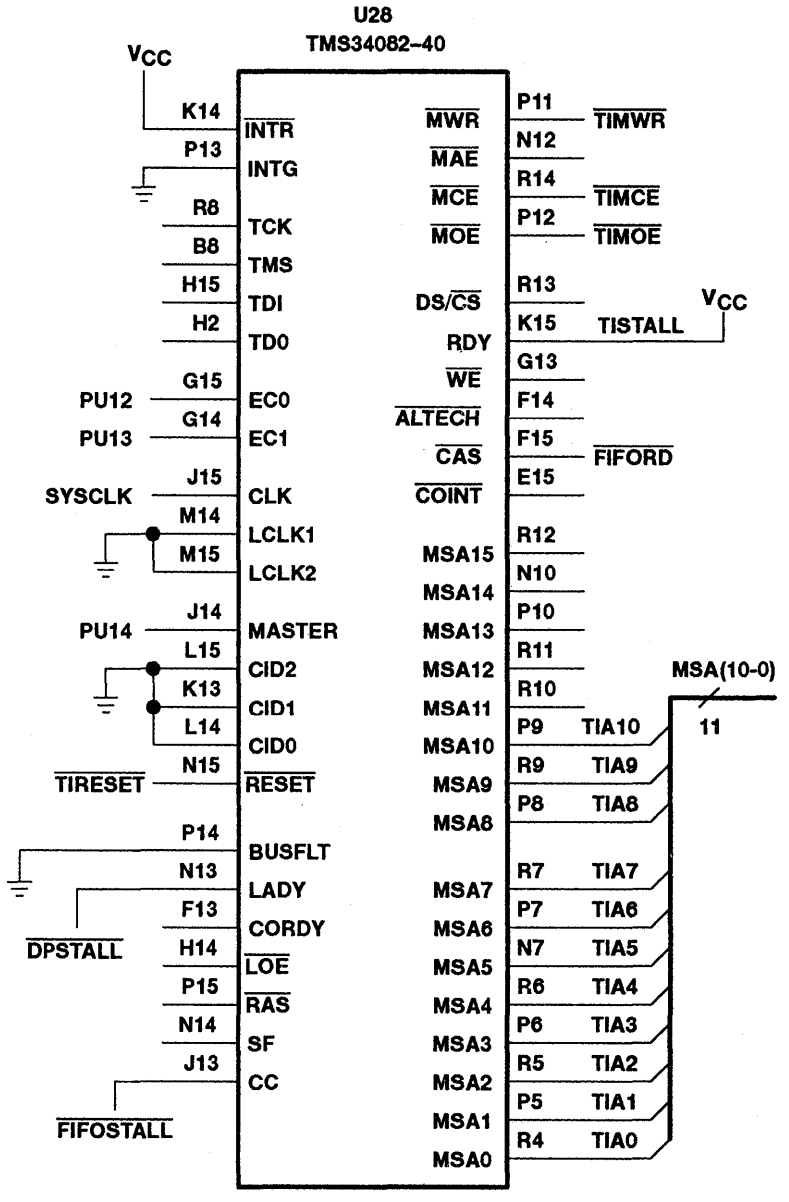


Figure 28. TMS34082, Details of U28

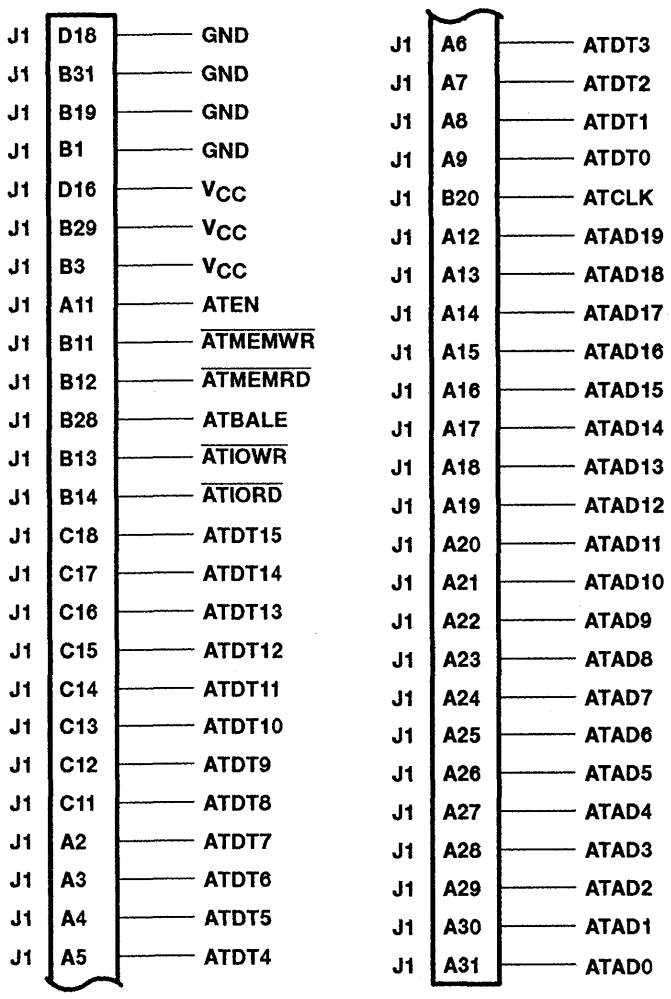


Figure 29. AT-Bus Connector

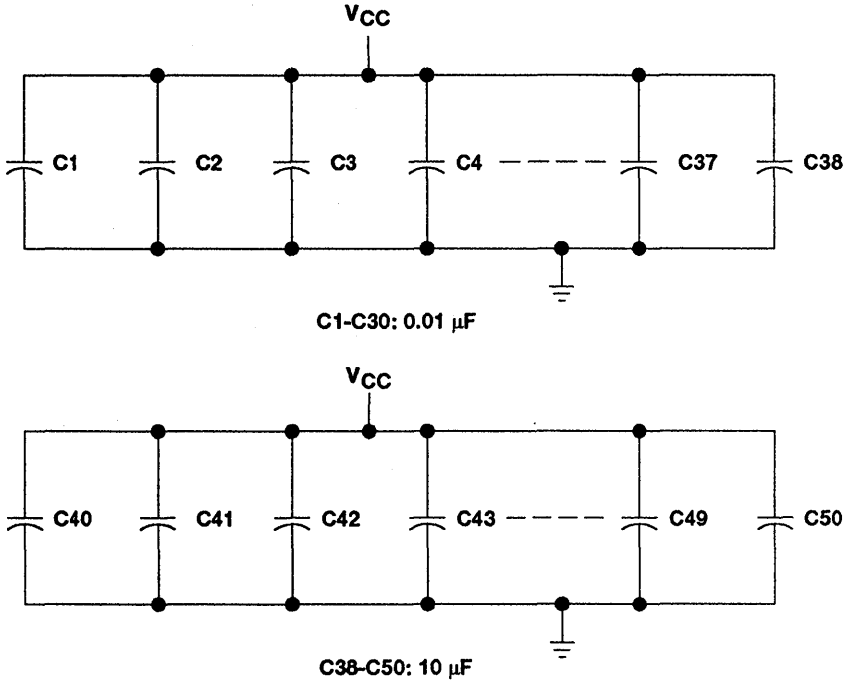


Figure 30. Capacitors

PAL® Code Listing

NOTE: All code is written using PAL® ASM software.

Memory Decode for TMS34082 Accelerator Board

```
PATTERN     MEMORY DECODE FUNCTIONS
REVISION    1A
AUTHOR      MIKE ROBERTS
COMPANY     TEXAS INSTRUMENTS
DATE        OCTOBER 11, 1989

; This PAL will decode memory functions from the PC/AT to the Motorola MC68030 host SRAM.

CHIP        20L8        PAL20L8
; DEVICE U1
;   1       2       3       4       5       6       7       8       9
  ATAD15  ATAD01  ATAD2   ATAD20 ATAD16  ATAD17  ATAD18  ATAD19  /IORD

;   10      11      12
  /IOWR   BALE    GND

; PIN
;         13          14          15          16          17          18
        /SMEMRD     /SMEMWR     /ATHIEN     ATAD23      ATAD22      ATEN1

;         19          20          21          22          23          24
        ATCNTL      /ATADDREN   ATEN2       /ATLOEN     AEN         VCC

EQUATIONS

; ALWAYS ENABLED

                ATLOEN.TRST     = VCC
                ATHIEN.TRST     = VCC
                ATADDREN.TRST   = VCC
                ATCNTL.TRST     = VCC
                ATEN1.TRST      = VCC
                ATEN2.TRST      = VCC

; ABOVE EQUATIONS NOT REQUIRED SINCE PAL ASM DEFAULTS TO THESE.
; THEY HAVE BEEN ADDED FOR CLARITY.

ATEN1       = /BALE * ATAD19 * ATAD18 * ATAD16 * /ATAD17 * /AEN
; USED AS A GATE TO ASSERT ACCESS TO HOST SRAM

ATEN2       = /(ATAD23 + ATAD22 + ATAD21 + ATAD20)
; INTERMEDIATE TERM DESELECTING ADDRESS LINES 23–20

ATLOEN      = ATEN1 * /ATAD01 * /SMEMRD * ATEN2
            + ATEN1 * /ATAD01 * /SMEMWR * ATEN2
; DECODES LOW BYTE IN EITHER READ OR WRITE MODE

ATHIEN      = ATEN1 * ATAD01 * /SMEMRD * ATEN2
            + ATEN1 * ATAD01 * /SMEMWR * ATEN2
; DECODES HIGH BYTE IN EITHER READ OR WRITE MODE
```

PAL is a registered trademark of Monolithic Memories Inc.

```
ATADDREN  = ATEN1 * /ATAD01 * /SMEMRD * ATEN2
          + ATEN1 * /ATAD01 * /SMEMWR * ATEN2
          + ATEN1 * ATAD01  * /SMEMRD * ATEN2
; ENABLES THE ADDRESS BUFFERS FOR HIGH OR LOW BYTE
```

I/O Decode for TMS34082 Accelerator Board

```
PATTERN     DECODE CONTROL FUNCTIONS
REVISION 1A
AUTHOR      MIKE ROBERTS
COMPANY     TI
DATE        10/10/89

; This PAL will decode I/O functions to set I/O signals.

; DEVICE U2

CHIP        20L8        PAL20L8
; PIN
;    1       2       3       4       5       6       7       8       9
  ATAD0   ATAD1   ATAD2   ATAD3   ATAD4   ATAD5   ATAD6   ATAD7   ATAD8
;   10      11      12      13      14
  ATAD9  ATAD10   GND    /IORD   /IOWR

; PIN
;    15      16      17      18      19      20      21      22      23
    NC1 /STATUSRD/CNTRLENNC2     NC3     NC4     NC5  SYSCNTL  BALE
;    24
    Vcc

EQUATIONS
; ALWAYS ENABLED

                /STATUSRD.TRST      = Vcc
                /CNTRLEN.TRST       = Vcc
                SYSCNTL.TRST        = Vcc
                NC1.TRST            = Vcc
                NC2.TRST            = Vcc
                NC3.TRST            = Vcc
                NC4.TRST            = Vcc

CNTRLEN   = /BALE * ATAD9 * ATAD8 * ATAD4 * ATAD3 * /ATAD2 * /ATAD1* /IOWR
; USED TO ENABLE I/O WRITE REGISTER FOR PC/AT Motorola MC68030 ARBITRATION AND
CONTROL OF TMS34082 HALT FUNCTIONS

STATUSRD  = /IORD * /BALE * ATAD9 * ATAD8 * ATAD4 * ATAD3 * /ATAD2 * ATAD1
; READ STATUS FROM ASYNCHRONOUS PAL
; USED TO READ STATUS FROM STATUS REGISTER

INVERT      = ATADO
; USED TO INVERT PC/AT ADDRESS 0
```

Status Control for TMS34082 Accelerator Board

```
PATTERN    STATUS CONTROL FUNCTIONS
REVISION   1A
AUTHOR     MIKE ROBERTS
COMPANY    TI
DATE       10/16/89

; This PAL will decode memory from the Motorola MC68030 to external DP-SRAM and the FIFO buffer.

CHIP       16RA8      PAL16RA8

; DEVICE U3
; PIN
;    1       2       3       4       5       6       7       8       9
   PRLD     NC2   /RESET     NC3     NC4     NC5     /BG     CLK     NC6
;   10
   DBEN

; PIN
;    11      12      13      14      15      16      17      18      19
/STATUSRD NC7      NC8    ATDT2   ATDT1   ATDT0     NC9    NC10    NC11
;    20      21      22      23      24
   DPRD   /TERM  /FIFOWR     NC5     VCC

EQUATIONS

TERM       = 68RW * A14 * /A15 * /AS
           + /68RW * A14 * /A15 * /DAS
           + /68RW * /A14 * /A15 * /DAS
; SYNCHRONOUS TERMINATION SIGNAL FOR FIFO AND DP-SRAM

/FIFOWR = /(/68RW * /DBEN * AS * /A14 * A15)
; FIFO WRITE ENABLE. SINCE FIFO IS EDGE-TRIGGERED, THESE SIGNALS ARE
RECOMMENDED.

DPCE       = 68R2 * A14 * /A15 * /AS
           + /68RW * A14 * /A15 * /DAS

; DUAL-PORT CHIP ENABLE

DPRD       = 68RW * A14 * A15 * /AS\; 8K SRAM READ SELECT

DPWRUU     = A14 * /A15 * /A1 * /A0 * /68RW
; BYTE ENABLE SELECTS FOR UPPER-UPPER BYTE

DPWRUM     = A14 * /A15 * /A1 * A0 * /68RW
           + A14 * /A15 * /A1 * /68RW * /SIZ0
           + A14 * /A15 * /A1 * /68RW * SIZ1

; BYTE ENABLE FOR UPPER-MIDDLE BYTE

DPWRLM     = A14 * /A15 * A1 * /A0 * /68RW
           + A14 * /A15 * /A1 * /68RW * /SIZ1 * SIZ0
```

```
                + A14 * /A15 * /A1 * /68RW * SIZ1 * SIZ0
                + A14 * /A15 * /A1 * /A0 * /68RW * /SIZ0

; BYTE ENABLE FOR UPPER-LOWER BYTE

DPWRLL      = A14 * /A15 * A1 * A0 * /68RW
                + A14 * /A15 * A0 * /68RW * SIZ1 * SIZ0
                + A14 * /A15 * /68RW * /SSIZ1 * /SIZ0
                + A14 * /A15 * A1 * /68RW * SIZ1

; BYTE ENABLE FOR LOWER-LOWER BYTE
```

Byte Enable Decode for TMS34082 Accelerator Board

```
PATTERN    DECODE CONTROL FUNCTION
REVISION   1A
AUTHOR     MIKE ROBERTS
COMPANY    TI
DATE       10/12/1989

; This PAL will decode byte enables of the Motorola MC68030 and PC/AT bytes to the 8K-SRAM.

CHIP       16R4        PAL16R4

; DEVICE U4
; PIN
;    1        2        3        4        5        6        7        8        9
CNTRLEN   NC1      NC2      BG     ATDT2    ATDT3    ATDT4    ATDT5    ATEN1
;   10
  GND

;    11       12       13       14       15       16       17       18       19
/OUTEN    NC2      NC3    /BGACK    /BR      NC4      NC5      NC6      NC7
;   20
  VCC

EQUATIONS
; ALL FUNCTIONS ARE ALWAYS ENABLED

BR         := ATDT3 * BG
; BUS REQUEST TO Motorola MC68030, ONLY ACTIVE WHEN Motorola MC68030 BUS GRANT
HIGH

BGACK      := ATDT2 * /BG
; BUS GRANT ACKNOWLEDGE SIGNAL FROM PC/AT, ACTIVE WHEN BUS GRANTED

68RST      := NATDT4
; THIS SIGNAL RESETS THE Motorola MC68030

TIRST      := ATDT5
; THIS SIGNAL RESETS THE TMS34082
```

Pattern Decode for TMS34082 Accelerator Board

```
PATTERN    DECODE CONTROL FUNCTION
REVISION   1A
AUTHOR     MIKE ROBERTS
COMPANY    TI
DATE       10/12/1989

; This PAL will decode memory from the Motorola MC68030 to external devices.

CHIP       22V10       PAL22V10

;PIN
;   1      2      3      4       5      6      7      8      9
   NC1    A19    A18   /68RW    A0     A1     A15    A16    A17
;   10     11     12
   A30    DAS    GND

;   13     14     15     16      17     18     19     20     21
   /C1   /8KWRCS /BOOT /68TIRS  NC4  /DPRDCS /STERM  NC2  /8KRDCS
;   22     23     24
  /8KCE  /FIFOWR  VCC

; MYTHICAL PIN THPC/AT SETS THE REGISTERS ON BOOT-UP CALLED VAPOR
VAPOR

EQUATIONS

          /8KWRCS.TRST        = VCC
          /BOOT.TRST          = VCC
          /68TIRST.TRST       = VCC
          /DPRDCS.TRST        = VCC
          /STERM.TRST         = VCC
          /8KRDCS.TRST        = VCC
          /8KCE.TRST          = VCC
          /FIFOWR.TRST        = VCC
          VAPOR.SETF                  = VCC


8KWRCS    = /68RW * /A14 * /A15 * /DAS * /AS * RST
; 8K SRAM WRITE RE-SELECT

8KRDCS    = 68RW * /A14 * /A15 * /AS * RST
; 8K SRAM READ PRE-SELECT

STERM     = /A14 * /A15 * 68RW ;8KRDCS
          + /A14 * /A15 * /68RW * /DAS ;8KWRCS
; SYNCHRONOUS TERMINATION ENDING SYNCHRONOUS CYCLES
/UUCS = /9/A14 * /A15 * /A1 * /68RW * RST + 68RW * /A14 * /A15 * /AS * RST + ATEN1 * ATADO)

; BYTE ENABLE SELECTS FOR UPPER-UPPER BYTE

/UMCS     = /(/A14 * /A15 * /A1 * A0 * /68RW * RST + /A14 * /A15 * /A1 * /SIZO * RST + /A14 * /A15
* /A1 * /68RW * SIZ1 * RST + /A14 * /A15 (68RW * RST + 68RW * /A14 * /A15 * /AS * RST * ATEN1
* ATADO)
```

```
; BYTE ENABLE FOR UPPER-MIDDLE BYTE

/LMCS       = /(A14 * /A15 * A1 * A0 /68RW * RST
              + /A14 * /A15 * /A1 /68RW * /SIZ1 * /SIZ0 * RST
              + /A14 * /A15 * /A1 * /68RW * AIZ1 * SIZ0 * RST
              + /A14 * /A15 * /A1 * A0 * /68RW * /SIZ0 * RST
              + 68RW * /A14 * /A15 * /AS * RST ;8KREAD
              + ATEN1 * ATADO)

; BYTE ENABLE FOR UPPER-MIDDLE BYTE

/LLCS       = /(A14 * /A15 * A1 * A0 * /68RW * RST
              + /A14 * /A15 * A0 * /68RW * SIZ1 * SIZ0 * RST
              + /A14 * /A15 * /68RW * /SIZ1 * SIZ0 * RST
              + /A14 * /A15 * A1 * /68RW SIZ1 * RST
              + 68RW * /A14 * /A15 * /AS * RST ;8KREAD
              + ATEN1 * /ATAD0

; BYTE ENABLE FOR LOWER-LOWER BYTE
```

Software Listings

Software listings available upon request. Contact the DVP Systems Engineering group at (214) 997-3970.

References

TMS34082 Designer's Handbook, Texas Instruments, 1990.

TMS34082 Data Sheet, Texas Instruments, 1990.

Motorola MC68030 User's Manual, Motorola, Inc., 1989.

680x0: Programming by Example, Stan Kelly-Bootle, Howard Sams and Company, 1988.

Motorola MC68881: Floating-Point Coprocessor User's Manual, Motorola, Inc., 1988.

The Motorola Motorola MC68020 and Motorola MC68030 Microprocessors: Assembly Language, Interfacing, and Design, Thomas L. Harman, Prentice Hall, 1989.

80286 and 80287 Programmer's Reference Manual, Intel, Inc. 1987.

8086/8088/80286 Assembly Language, Leo J. Scanlon, Brady Publishing Co., 1988.

Assembly Language Primer for the IBM PC & XT, Robert Lafore, Plume/Waite, 1984.

Technical Reference: Personal Computer AT, IBM, 1985.

80286 Hardware Reference Manual, Intel, Inc., 1987.

Appendix E

A High Performance Floating-Point Image Computing Workstation for Medical Applications

This appendix describes the hardware and software architecture of a medium-cost floating-point image processing and display subsystem for the NeXT™ computer, and its applications as a medical imaging workstation.

Karl S. Mills, Gilman K. Wong, and Yongmin Kim
Image Computing Systems Laboratory (ICSL)
Department of Electrical Engineering
University of Washington
Seattle, WA 98195

Reprinted with permission from
Medical Imaging IV: Image Capture and Display,
Society of Photo-Optical Instrumentation Engineers (SPIE)
Conference Proceedings
Yongman Kim, Editor
Volume 1232, Summer 1990

Abstract

The medical imaging field relies increasingly on imaging and graphics techniques in diverse applications with needs similar to (or more stringent than) those of the military, industrial and scientific communities. However, most image processing and graphics systems available for use in medical imaging today are either expensive, specialized, or in most cases both. High performance imaging and graphics workstations which can provide real-time results for a number of applications, while maintaining affordability and flexibility, can facilitate the application of digital image computing techniques in many different areas.

This paper describes the hardware and software architecture of a medium-cost floating-point image processing and display subsystem for the NeXT ™ computer, and its applications as a medical imaging workstation. Medical imaging applications of the workstation include use in a Picture Archiving and Communications System (PACS), in multimodal image processing and 3-D graphics workstation for a broad range of imaging modalities, and as an electronic alternator utilizing its multiple monitor display capability and large and fast frame buffer.

The subsystem provides a 2048 x 2048 x 32-bit frame buffer (16 Mbytes of image storage) and supports both 8-bit gray scale and 32-bit true color images. When used to display 8-bit gray scale images, up to four different 256-color palettes may be used for each of four 2K x IK x 8-bit image frames. Three of these image frames can be used simultaneously to provide pixel selectable region of interest display. A 1280 x 1024 pixel screen with 1:1 aspect ratio can be windowed into the frame buffer for display of any portion of the processed image or images. In addition, the system provides hardware support for integer zoom and an 82-color cursor. This subsystem is implemented on an add-in board occupying a single slot in the NeXT ™ computer. Up to three boards may be added to the NeXT ™ for multiple display capability (e.g., three 1280 x 1024 monitors, each with a 16-Mbyte frame buffer).

Each add-in board provides an expansion connector to which an optional image computing coprocessor board may be added. Each coprocessor board supports up to four processors for a peak performance of 160 MFLOP5. The coprocessors can execute programs from external high-speed microcode memory as well as built-in internal microcode routines. The internal microcode routines provide support for 2-D and 3-D graphics operations, matrix and vector arithmetic, and image processing in integer, IEEE single-precision floating point, or IEEE double-precision floating point.

In addition to providing a library of C functions which links the NeXT ™ computer to the add-in board and supports its various operational modes, algorithms and medical imaging application programs are being developed and implemented for image display and enhancement. As an extension to the built-in algorithms of the coprocessors, 2-D Fast Fourier Transform (FET), 2-D Inverse FFT, convolution, warping and other algorithms (e.g., Discrete Cosine Transform) which exploit the parallel architecture of the coprocessor board are being implemented.

Introduction

The medical field relies increasingly on image computing in many applications areas. Current needs in the medical field include the employment of image processing and graphics in medical image enhancement, simple measurement, or scientific visualization of change, movement, and flow, as well as successive 2-D slices in 3-D medical images. X-ray Computed Tomography (CT), Magnetic Resonance Imaging (MRI) and Positron Emission Tomography (PET) all use computationally intensive reconstruction methods to produce detailed cross sections of the structure. Other medical imaging modalities include digital radiography (digital X-rays), ultrasound and nuclear medicine scanners. These imaging modalities are used to understand internal anatomical and functional pathologies and to utilize that information in various clinical cases, for example during brain or orthopedic surgery. Image processing techniques are necessary for picture enhancement, and computing various statistics in applications like detecting suspicious cancer cells from pap smears. Picture Archiving and Communications System (PACS) with filmless archiving for all the images is a powerful concept with vast untapped potential. High-performance graphics and imaging workstations are essential for successful PACS.

This paper describes the most recent of a series of affordable, high-performance image computing workstations, the University of Washington Graphics System Processor #3 (UWGSP3) and its application to medical imaging. The UWGSP3 image processing board set supports the following features:

- Single 2k x 2k x 32-bit (16 Mbytes) roamable video/frame memory implemented entirely with 1 Mbit VRAMs
- 32 bits per pixel configured as 24-bit true-color system with 8 overlay bits, or up to four 8-bit pseudo-color or gray-scale frames (or 3 frames with overlay)
- 160 MFLOPS peak performance for high-speed integer and floating-point image processing and graphics functions
- 1280 x 1024 60-Hz noninterlaced color display with 1:1 screen aspect ratio
- Hardware zoom, roam, and cursor support
- Up to 3 different color palettes, each driven by a different plane, can be displayed at once for region of interest (ROI) operations
- Expansion port for digitizer, additional frame memory or other devices
- Improved system performance (4 to 8 times that of previous UWGSP systems)
- Support for window-oriented user interface
- NeXT™ host system
- Medium-cost

The UWGSP3 offers the powerful, yet flexible environment necessary for meeting the stringent needs of many imaging applications. Applications other than those in medicine include scientific applications: astronomy, remote sensing, geology, seismology, oceanography, and earth resources planning; industrial applications: machine vision and robotics, tolerance verification, parts identification, optical character recognition, and thermography; military applications: field-deployable military workstations for map analysis and processing, target identification and tracking, and surveillance; forensics: fingerprint analysis and identification, signature verification and dental records analysis; and graphics applications: computer image display and synthesis (for example, solid modeling, ray tracing, object rendering and shading), image overlay, graphic arts, and ad preparation. Although some of these applications may never be implemented, these are the types of applications which could be developed on UWGSP3.

Background

Several image computing subsystems have been developed in the Image Computing Systems Laboratory (ICSL) of the University of Washington. The University of Washington Graphics System Processor #1 (UWGSP1), developed in 1987, was the first of these systems. It has been used as a low-end PACS medical imaging workstation in the University of Washington PACS prototype system [Gee et al., 1989]. This first generation image computing subsystem was implemented on 2 IBM IC/AT protyping cards, heavily utilizing the processing power of the TMS34010 Graphics System Processor (GSP) and TMS32020 Digital Signal Processor (DSP). In UWGSP1, the screen and graphics functions are controlled by the GSP, and the DSP is used as a numeric coprocessor accessed via First-In First-Out (FIFO) buffers from the GSP. The spatial resolution of the display is 512 x 512 pixels with a contrast resolution of 8 bits per pixel. Hardware zoom, pan and scroll, one video frame buffer, and three workspace buffers are incorporated in the system. Software developed for the UWGSP1 includes point operations, arithmetic and logical operations, Region of interest (ROI), convolution, geometric transformation, Fast Fourier Transform (FFT) and Inverse (IFFT). Used in conjunction with a PC/AT host, UWGSP1 provides a flexible three processor low-cost medium performance workstation for fixed-point image processing applications.

While the UWGSP1 has proven to be a viable performer in various image analysis and processing applications, experience with the system exposed problem areas that required attention. UWGSP1 suffered from the following problems which somewhat limited its usefulness as an image computing workstation:

- The DSP's 16-bit fixed-point arithmetic can cause serious problems in accuracy of some image processing and graphics operations due to overflow, truncation, and other problems,
- Communication between the GSP and DSP through FIFO buffers is inefficient and difficult to manage,
- Some DSP operations are slow (e.g., 2-D FFT on 512 x 512 images takes about 16 seconds),
- For many applications, 512 x 512 display resolution is not enough, and
- Because the screen aspect ratio is not 1:1, warping of images is required for them to appear in proper proportion.

Because of these limitations, a second generation image processing subsystem was proposed (UWGSP2) and implemented at the ICSL in 1988 [Chinn et al., 1988]. UWGSP2 utilizes the Texas Instruments' 74ACT8837 Floating Point Processor (FPP) as a replacement for the TMS32020 DSP, to provide high-performance floating-point implementation of computationally-intensive image processing and graphics algorithms. By incorporating the FPP in the second generation design, most of the problems associated with the DSP's 16-bit fixed-point arithmetic operations were alleviated, while still obtaining a performance increase of about 2 times that of UWGSP1. However, the GSP to FPP FIFO interface continued to be a data flow bottleneck in the system, the display resolution was still insufficient for many imaging applications, and the screen aspect ratio was still other than 1:1.

A third generation system (UWGSP3) has been designed and implemented at the ICSL in 1989, and overcomes the limitations of the earlier systems by adding increased display resolution (from 512 x 512 to 1280 x 1024), increased frame buffer storage (from 1 Mbyte to 16 Mbytes), support for 32-bit true-color as well as 8-bit gray scale images or up to 24-bit gray scale images windowed and leveled into 8 bits, an intuitive graphical user interface, and multiple floating-point coprocessors for 160 MFLOPS of peak processing performance. This system and its application to medical imaging are described below.

The UWGSP3 is implemented on a single multilayer printed circuit board, with an expansion connector for an optional coprocessor board. It is designed around two special purpose VLSI processors, the TMS34020 second generation Graphics System Processor and the TMS34082 Floating-Point Processor. Figure 1 shows a block diagram of the system with major components which include a NeXT ™ Host System and Interface Logic, the TMS34020 Graphics System Processor, four TMS34082 Floating-Point Processors, Local Program Memory (1 Mbyte), and Video Display and Frame Buffer Memory (16 Mbytes). Each of these major design blocks is described below.

System Architecture

NeXT ™ Host System and Interface Logic

The host system for UWGSP3, the NeXT ™ computer, was selected over other potential host systems (e.g., MAC II, PC/AT compatibles, SUN, etc.) mainly for its flexibility and ease of use and programming. The NeXT 's™ operating system, Mach (compatible with BSD 4.3 UNIX), provides a popular, portable, and flexible environment for software development and maintenance. Although UNIX provides an extremely versatile development environment, it is somewhat cryptic and cumbersome for the general user. However, the NeXT ™ provides a user friendly "Macintosh-like" interface for the nonprogrammer, while still providing the excellent development environment afforded by UNIX. Furthermore, the NeXT ™ architecture includes a high-speed 32-bit bus (NextBus, an enhanced NuBus) providing burst transfer rates of up to 100 Mbytes per second, and the significant board real estate necessary to support complex hardware designs. Another benefit afforded by the NeXT ™ is an interactive interface development environment (Interface Builder) which can generate user interface code directly. Because the user interface usually represents approximately 20% of the code, but requires as much as 80% of the effort, this capability can provide a significant savings in the time to develop various medical imaging applications by simplifying the generation and modification of application software interfaces [Jobs, 1989]. NeXT 's™ object-oriented approach to software development makes it possible to develop image processing code modules which could be integrated into applications and user interfaces by the end user.

The backplane of the NeXT ™ computer supplies three expansion slots. Thus, up to three UWGSP3 subsystems can be inserted into the NeXT ™ for applications that require multiple displays. Interfacing of the NeXT ™ host to the UWGSP3 system is provided using a dedicated host interface port on the TMS34020. Executable programs, operands, images, and commands are passed to the UWGSP3 and its local memory via this host interface, with the NeXT ™ acting as the master and the UWGSP3 acting as a slave device. The host initializes the subsystem by transferring a GSP executable command decoder into GSP program memory via the host interface port. With the command decoder installed, image processing and graphics functions may be issued from the host. Once a command has been issued to the GSP, the host is free to pursue other functions as may be required, while the GSP decodes the command and executes the appropriate program on the UWGSP3 local bus.

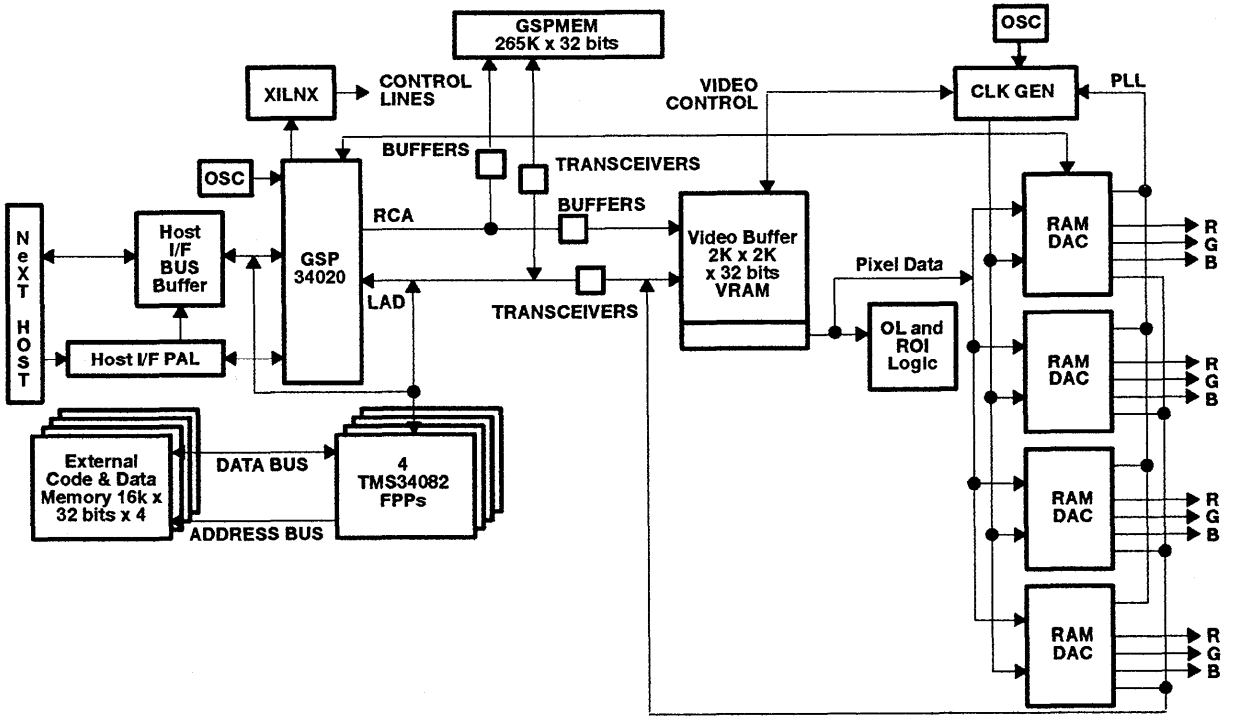


Figure 1. UWGSP Block Diagram

Processors

The two high-performance special purpose VLSI processors used in UWGSP3 represent state-of-the-art performance and integration. Texas Instruments TMS34020 is the second generation of an advanced high-performance CMOS 32-bit microprocessor optimized for graphics display systems [Texas Instruments, 1989]. Addressing is bit oriented and all data structures such as pixel size and frame size as well as display characteristics are defined in internal GSP control registers, allowing the GSP to be configured to support a wide variety of display devices and formats. The TMS34020 contains a built-in instruction cache, hardware support for raster graphics instructions, video display timing generation hardware, as well as a memory controller and video memory controller. Extensions to the basic architecture of the GSP are provided through its coprocessor interface. Special instructions and cycles are available for enhancing data flow to coprocessors while maintaining a closely coupled processor-coprocessor environment.

The TMS34082 is a high-speed (40 MFLOPS peak) floating-point processor combining on a single chip a 16-bit sequencer, address generation and a three operand floating-point unit with twenty-two 64-bit data registers [Texas Instruments, 1989]. Single and double precision IEEE floating-point operations are supported for addition, subtraction, multiplication, division, square root, and comparison. In addition to floating-point operations, 32-bit integer arithmetic, logic operations and shifts may be performed by the 34082. To allow integer pixels to be manipulated in floating point, conversions are provided from integer to single or double precision formats and vice versa. To make the FPP more useful in imaging and graphics applications, internl micrcoode routines are provided for vector and matrix operations and the following graphics and image processing functions:

- 3 x 3 variable kernel convolution
- Backface elimination
- Polygon, 2-Plane, and 2-Plane color clipping
- 2-D and 3-D cubic spline
- 2-D window compare and 3-D volume compare
- Viewport scaling and conversion
- 2-D and 3-D linear interpolation
- Polygon elimination

External microcode support is also available to allow custom algorithm implementation on the 34082 processors. Additional image processing and graphics algorithms utilizing one to four 34082 processors are currently being implemented on UWGSP3.

Using the Texas Instruments TMS34020 GSP and the Texas Instruments TMS34082 Floating-Point Processor as a closely coupled processor pair alleviates much of the data transfer bottleneck experienced in the first and second generation UWGSP subsystems. Images stored in frame buffer memory can be transferred directly to the FPP rather than being read by the GSP and rewritten to FFO buffers as in the earlier UWGSP systems. But, with the display area and pixel depth each more than four times that of UWGSP1 or 2, additional processing capability is required to overcome the added computational demands imposed. For this reason multiple (up to 4) FPPs can be attached to the local GSP bus to provide this processing horsepower. As indicated In Figure 1, the FPPs connect directly to the Local Address and Data (LAD) bus of the GSP. Each FPP is also attached to its own bank of high speed 16K x 32 bit static memory for external microcode and data storage via the MicroStore Data (MSD) and Address (MSA) buses. The static memory and the MSD and MSA buses operate independently from the GSP's LAD bus, thus reducing GSP local bus activity. Transfers between the GSP memory and the FPP static memory pass through the FPP via the LAD and MSD buses when data or programs are needed by the coprocessors. Registers may also be transferred between the GSP and FPPs at any time.

Using the computing horsepower of the TMS34082's, UWGSP3 can outperform the UWGSP2 system by 4 to 8 times for computationally intensive operations requiring floating-point accuracy. By incorporating the TMS34020 as the graphics engine, graphics and other imaging operations also see a performance increase of 4 to 8 times that of the current UWGSP subsystems.

Memory

Memory on the GSP local bus is linear and can be partitioned in a user-defined manner. The video buffer on UWGSP3 is configured normally as a single 2048 x 2048 x 32-bit buffer, but may be reconfigured as four 2048 x 2048 x 8-bit planes, four 4096 x 4096 x 4-bit planes, four 8K x 8K x 2-bit planes, or four 16K x 16K x 1-bit planes. The large video display buffer provides the ability to load large images into the buffer and roam through them, or to load several different images (e.g., an entire CT or MR study) into the buffer at once. For graphics or computer image generation applications, having a video buffer of more than two times the screen size allows double buffering of the display for smooth image and graphics transitions. The video frame buffer is implemented entirely in 1 Mbit multiport Video RAM (VRAM). The use of VRAM substantially increases the availability of the local bus because screen refresh data moves over a separate path to the combined lookup tables and digital to analog converters (RAMDACs).

The GSP program memory consists of 256K x 32-bits of Dynamic RAM (DRAM). This memory is used to store the local programs and data needed to control the display, manipulate images and graphics, and control the four coprocessors. Because the GSP contains the necessary hardware to control both DRAM and VRAM directly, the memory interface requires only the addition of buffers, transceivers and minimal control logic.

Video Display

UWGSP3 also provides a solution to the resolution and aspect ratio problems experienced in earlier UWGSP systems. The aspect ratio for the subsystem is adjusted for 1:1 in all display modes, providing a proportionally correct image required for most graphics and image processing applications. Furthermore, the 1280 x 1024 display resolution provides sufficient display resolution for most applications, while a roamable video/frame buffer of 2K x 2K x 32-bits (16 Mbytes) provides an acceptable solution to all others. The GSP generates the video timing signals; however, it cannot drive the display itself.

Four Brooktree RAMDACs are used to drive the monitor. Each RAMDAC has a 256 x 24 bit lookup table (LUT) which drives 8 bits each of red, green and blue signals. The red, green and blue outputs of each RAMDAC are summed together and the composite signals are used to drive the monitor. For true color applications, one RAMDAC will drive only red, one will drive only green and one will drive only blue. The fourth RAMDAC provides 8 bits of overlay information. For gray scale or pseudo-color applications, a single RAMDAC drives red, green and blue outputs concurrently while the other RAMDACs are disabled. While in this mode, it is possible to do region-of-interest (ROI) (i.e., different portions of the screen are assigned different color mappings and/or image data) by switching on and off different RAMDACs In specific regions of the display on a pixel-by-pixel basis. Thus, by enabling different combinations of the RAMDACs, the frame buffer can be configured either as a 24-bit true color buffer with 8-bit overlay or as four separate 8, 4, 2, or 1-bit buffers. Bit-per-pixel selection of 8, 4, 2, or 1 is supported directly in hardware in the RAMDACs and augmented by appropriate clocking of the VRAMs. Overlay is also available in the ROI or 8, 4, 2, or 1-bit modes; however, one of the 8-bit planes must be used for the overlay leaving only three available for image display. Images with contrast resolution ranging from 9 to 24 bits per pixel may also be windowed and leveled into 8 bit gray scale or pseudocolor images using the FPPs. The RAMDACs also include support for a hardware cursor and integer zoom. The cursor shape, color and intensity are stored in a 64 x 64 x 2-bit array within each RAMDAC. The hardware zoom feature requires the support of an external state machine implemented in a Xilinx Logic Cell Array.

Software Architecture

The overall software architecture for the UWGSP3 and NeXT ™ host system is shown in Figure 2. At the lowest level, drivers local to the UWGSP3 board provide screen management functions as well as graphics and image processing primitives. Commands and data are transferred from the host system to the UWDGSP3 over the NextBus using NeXT ™ hardware specific drivers via the host interface. The NeXT ™ drivers use memory mapped I/O to make the entire usable GSP address space available to the host. Implemented on top of this functionality will be device independent image processing and graphics functions which will provide a consistent and portable software interface for applications.

The communication protocol between the host and UWGSP3 is administered by a command decoder running locally on the UWGSP3. The command decoder provides entry points to screen management functions as well as entry points to FPP external microcode routines and FPP management functions.

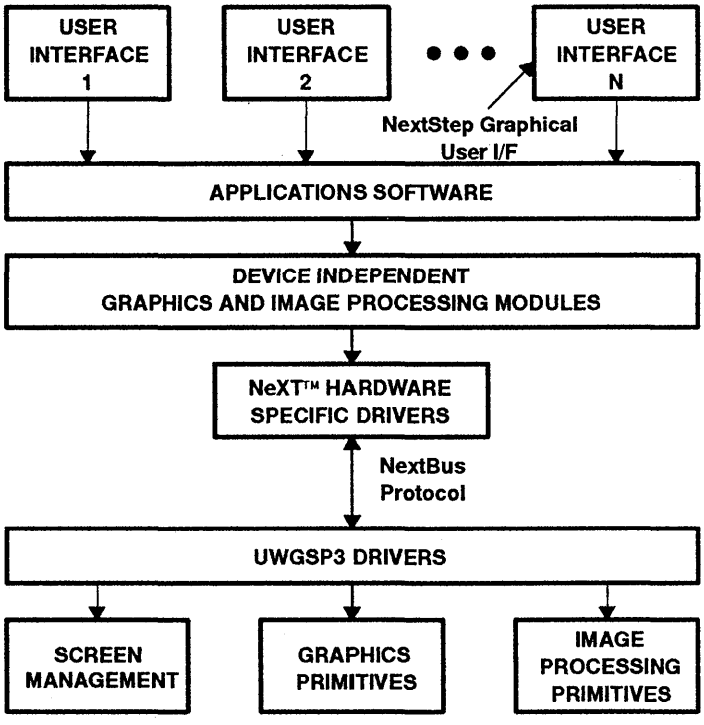


Figure 2. UWGSP3 Software Architecture

Microcoded routines are implemented as parallelized algorithms. Each FPP is assigned a different portion of the image to process while the GSP is responsible for handling the data transfers to and from the FPPs. This is done in such a way as to maintain the scalability of the coprocessor board; that is, the routines will be able to utilize any number of FPPs up to the maximum of four. As long as the parallel algorithms maintain a ratio of 3:1 or greater for the amount of time spent processing relative to the amount of time spent transferring data, the power of all four FFPs can be fully utilized. Code generation for both the TMS34020 and TMS34082 is being done in C with assembly language and microcode mixed in as necessary for optimization.

Using the Interface Builder development tools available on the NeXT ™, different graphical user interfaces can be quickly prototyped and implemented for applications. Figures 3 and 4 illustrate a prototype interface of an interactive filtering package for UWGSP3 being developed. Figure 3 shows an example of the window used to specify a filter's frequency response. The shape of the frequency response curve may he changed by either typing in the desired parameters or by using the mouse to interactively drag one of the control points (identified as a solid black dot). In this example, a lowpass fllter is shown; but in addition, there are filter windows for highpass, bandpass, bandstop, and azimuthal filters. Once a desired filter is designed, the user can apply the filter to the image by performing a 2-D FFT operation on the image, multiplying the filter and image in the frequency domain, peforming a 2-D IFFT operation, and displaying the filtered image in its window. The UWGSP3 can complete the entire process interactively (e.g., taking only a few seconds for a 256 x 256 image). Thus, trying fllters with different characteristics can be easily supported on UWGSP3 without undue delay to the user.

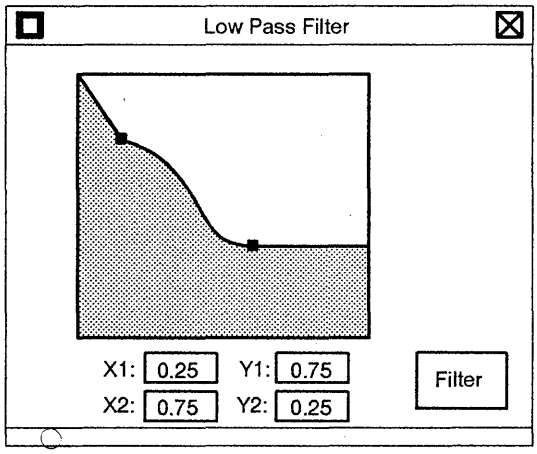


Figure 3. Lowpass Filter Specification Window

Besides allowing interactive filter specification for frequency domain filtering, the package supports image loading and frame buffer roaming and zooming (Figure 4). The image load window (on the left) allows any size image (up to 2048 x 2048) to be loaded anywhere within the 2048 x 2048 frame buffer. Control buttons are provided for standard sized images from 64 x 64 to 2048 x 2048. Another set of control buttons determines which channel the image will he loaded into: red, green, blue, or overlay. The frame buffer window is a scaled representation of the entire frame buffer area. The rectangular black outline defines the boundaries of the current display region. The mouse may be used to move the display to a different portion of the frame buffer by clicking and dragging on the display outline. Zoom buttons on the right allow the display to be zoomed by any integer from 1 to 8. The size of the display outline shrinks to reflect the reduced display region as higher zoom levels are employed. The position of previously loaded images are indicated by the shaded rectangles in the frame buffer window. The interface development effort has been in progress for several months and is almost complete at the time of this writing. The programmer attributes the short development time to the use of the Interface Builder tools.

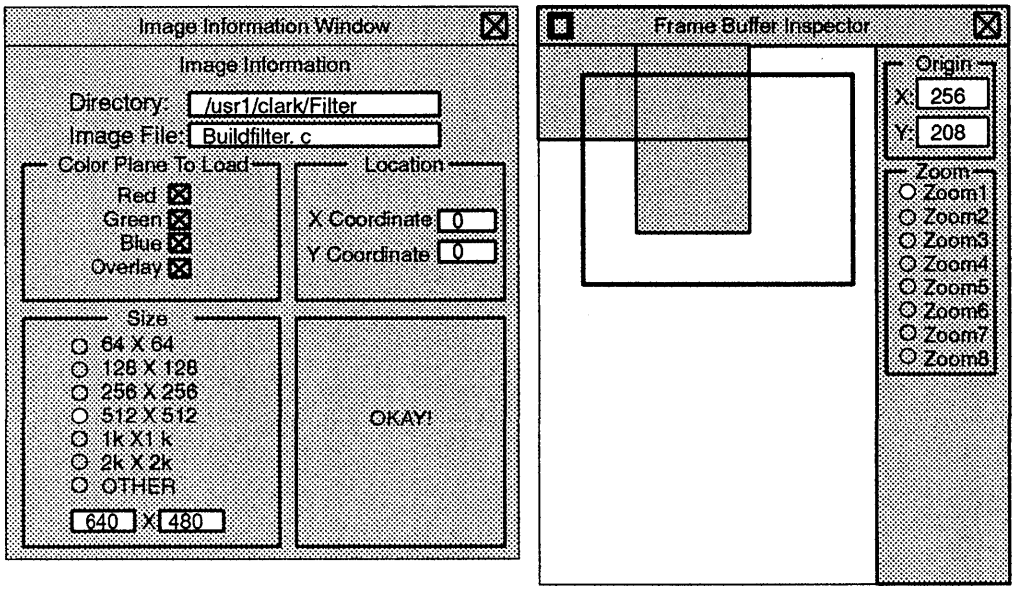


Figure 4. Image Load (left) and Virtual Frame Buffer (right) Windows

Application Areas

The UWGSP3 board set and software libraries transform the NeXT ™ computer into an affordable image processing and graphics workstation with processing performance currently available only with much higher cost workstations. Furthermore, the UWGSP3 board set is designed to be flexible enough to provide processing in a wide range of imaging and graphics applications while most other systems are optimized for specific tasks. Described below are a few of the application areas of UWGSP3 in medical imaging, which show mainly the image processing and display capabilities of the system; however, 2-D and 3-D graphics tools are also being developed.

PACS Workstation

Requirements for a PACS workstation include the following:

- High-resolution image display
- Large image frame buffer and magnetic storage
- Text display
- Network to tie together workstations
- Archival storage
- Rapid image retrieval and display

Together, the UWGSP3 and the NeXT ™ computer provide a solution to each of these requirements. The 1280 x 1024 display satisfies the condition for high resolution display in most applications (with the possible exception of digital radiography where resolutions of up to 2048 x 2048 are needed). In the arena of on-line storage, we have 16 Mbytes of frame buffer memory available for image storage. In terms of 8-bit gray-scale images, this is enough storage for 16 1k x 1k images, 64 512 x 512 images, 256 256 x 256 images, or 1024 128 x 128 images. In addition, NeXT ™ offers a 660 Mbyte ESDI hard drive which may be used for temporary storage of a local image database to hold a reasonable number of images downloaded from the central database. Furthermore, the UWGSF3's overlay channel meets the PACS workstation's textual (as well as graphical) annotation needs. As for network capabilities, the NeXT ™ features a built-in thin wire Ethernet interface and its operating system supports the TCP1IP protocol. This makes the UWGSP3 system suitable for use with other PACS systems, such as the UW prototype PACS, which already use Ethernet and TCP/IP Kim et al., 1989]. Finally, the NeXT 's™ leading role in using optical disk technology provides UWGSP3 with a high-capacity transportable storage media capable of storing 256 Mbytes per disk at a cost of about $0.20 per Mbyte, or the UWGSP3 system can have access to a central archival storage unit, e.g., a 5-114" optical disk jukebox at a lower cost per Mbyte.

One area in which UWGSP3 could use improvement is in the rapid transfer of data from storage to the display. Currently, it takes ahout 3 seconds to load a 512 x 512 x 8-bit image and 38 seconds to load a 2048 x 2048 x 32-bit true-color image. However, we are currently evaluating the feasibility of incorporating a parallel transfer disk to improve the NeXT 's™ disk performance. Furthermore, we are studying a hardware modification to allow UWGSP3 to operate closer to the NextBus' 100 Mbytes second peak transfer rate.

Another benefit of UWGSP3 is derived directly from the NeXT ™ host system. O'Malley [1989] advocates an iterative approach to PACS workstation development that involves a cycle of prototype evaluation and revision. The NeXT 's™ Interface Builder tool and its use of the Objective-C object-oriented programming language, provide the rapid prototyping ability needed for this type of development paradigm.

Electronic Alternator

The use of a digital system for emulating a conventional film alternator has been analyzed by several researchers [McNeill et al., 1988][Beard et al., 1988][Choi et al., 1990]. Analysis of these systems and their ability to emulate the current film altemator configuration digitally, has revealed several problems which must be overcome before workstations of this type can be used in radiology departments. Some of the problems include:

- Slow image loading rates for large images
- Inadequate number of image displays
- Resolution requirements (2k x 2k) are cost prohibitive
- Current systems do not address radiologists' needs beyond display and processing

Image loading rates for images as large as 2k x 2k vary from system to system, but may require as much as 1 minute to load. Recent advances in disk technology such as the Parallel Transfer Disk (PTD) can reduce this to a few seconds, but the ability to maintain as many images as possible resident in memory still remains important, to provide as interactive a system as possible. The UWGSP3 can hold up to four 2k x 2k x 8 bit images in memory at one time, any of which can be displayed instantaneously. In addition, up to three UWGSP3 may be installed in a single NeXT ™ computer system allowing a total of up to twelve 2k x 2k x 8-bit images or six 2k x 2k x 16-bit images to be resident in memory at any one time. Coupled with a PTD this large number of image storage frames would allow for acceptable speeds in displaying radiological studies.

Viewing a large number of images at one time is important to the radiologist in that it better emulates the current mechanical alternator configumtion and allows images to be compared side-by-side. As previously indicated, currently up to 3 UWGSP3 boards can be installed in a single NeXT ™ computer. Thus 3 separate displays, in addition to the NeXT ™ display are available for image viewing and manipulation. This limitation on the number of displays is not limited by the design of the UWGSP3 itself, but in the number of backplane slots available on the NeXT ™ computer system. In later iterations of the board, multiple displays may be available on a single board, further increasing the number of available displays.

Providing displays with resolution as high as 2k x 2k is at present cost prohibitive when a large number of displays are required in UWGSP3, this issue was addressed by implementing a 1280 x 1024 display which is adequate for display of most imaging modalities. Display of multiple images as large as 2k x 2k is also possible on UWGSP3 by using hardware pan and scroll of the 1280 x 1024 display in the 2k x 2k image frame. For multiple displays, this roaming can be done on all images concurrently, providing the same positioning of the display in all images. Or, if desired, the images can be panned and scrolled individually to view different areas of each image.

Many electronic altemator systems have addressed the display and processing needs of the radiologist. However, the integration of verbal annotation and/or digital film annotation is not always addressed. Verbal annotation may still be done using existing dictation hardware (e.g., a dictaphone system); however, the integration of this into the electronic altemator system would allow the verbal annotation to be directly associated with the digital image in a complex database. The NeXT ™ computer host provides the built-in capability for voice digitization, which could be linked with the image in a database. Potential also exists for voice recognition of commands and for speech to text conversion, using the Digital Signal Processor (DSP) available on the NeXT ™ computer host or some other specialized hardware.

Image Processing and Graphics

The optional 160 MFLOPS peak performance coprocessor board makes the UWGSF3 an excellent platform for image computing. The GSP supports many frame buffer manipulation functions (e.g., P1XBLT, FILL, and image arithmetic) and the FPPs include many built-in microcode routines for both image processing and 2-D and 3-D graphics operations (e.g., 3 x 3 convolution, matrix and vector operations, polygon clipping, and backface elimination). Thus, the UWGSF3 serves as a platform suitable for both image processing and graphics applications.

NxN Variable Size 2-D Convolution

NxN 2-D convolution can be used to implement a variety of image processing filtering operations such as lowpass, highpass, edge enhancement and edge detection. The algorithm is parallelized by segmenting the image into regions and assigning each locality to a different FPP. The predicted performance for a 512 x 512 x 32-bit image (using all four FPPs) is as follows:

- 5 x 5 kernel 0.7 seconds
- 11 x 11 kernel 3.0 seconds
- 15 x 15 kernel 5.5 seconds

FFT/IFFT

In some image analysis applications, FFT filtering techniques are often more convenient and intuitive than 2-D convolution and therefore more desirable. Thus, UWGSP3 must provide efficient FFT and IFFT algorithms. The FFT and IFFT will be implemented using the row and column method. Each of the four FPPs will be given an entire row or column to process, thereby parallelizing the operation. The predicted performance for a 512 x 512 x 32-bit image using all four FPPs is 4 seconds for either an FFT or an IFFT.

Geometric Transformations (warping)

Geometric transformations are utilized in correction of image distortion arising from deficiencies in the acquisition apparatus, image registration for multimodal image analysis, and interpolated zooms for applications in image magnification and minification. Each FPP is assigned a different region of the resultant image. For each destination pixel, the FPPs calculate the coordinates of the source pixels. The GSP passes the source pixels to the FPPs which then perform a bilinear interpolation using the pixel values. Predicted performance of a lower-order warp using bilinear interpolation is 2.5 seconds for a 512 x 512 x 32-bit image.

Window and Level

In digital radiography and CT and MR images, pixel sizes of up to 12 bits are generated. Most systems do window and leveling of these images by manipulating a 12-bit to 8-bit video output lookup table [Austin, et al, 1988]. Because UWGSP3 utilizes 8-bit RAMDACs, this method cannot be used. Instead the coprocessor board is used to perform a transformation of the 12-bit image into a window and leveled 8-bit image. The FPPs are used to calculate a transformation lookup table. Regions of the image are then sent to the FPPs which use the lookup tables to produce the 8-bit image. One of the advantages of this method is that the window and level operation can be limited to user defined regions and need not affect the entire display. Furthermore, this method may be used with pixel sizes greater than 12 bits (up to 24 bits per pixel). A full screen (1280 x 1024) transformation of a 24-bit image to an 8-bit image is expected to take less than 0.3 seconds.

Graphics

As mentioned in previous sections of this paper, the TMS34082 FPP includes many built-in microcode routines for 2-D and 3-D graphics which can be used to form the core functionality of a graphics library. In addition, the relatively large external microcode and data storage (16K x 32-bit for each FPP) allows higher level operations such as ray tracing and volume rendering to be added to the standard set of functions. Furthermore, the UWGSP3 architecture which couples the GSP with multiple FPPs, allows the computational workload to be distributed among the processors. Thus, each FPP can be given a different portion of the object database or a different set of tasks in the rendering process while the GSP is utilized to maintain the integrity of the frame buffer and transfer blocks of data to and from the FPPs.

Since the design is centered around the TM534020, the availability of the Texas Instruments Graphics Architecture TIGA-340 interface allows a UWGSP3 ported to an IBM AT-compatible (or MCA or EISA-based) architecture to be immediately compatible with many graphics applications written for this standard. In addition, UWGSP3 can be made to emulate other widely accepted video adapters such as EGA and VGA to support a vast number of different application programs.

For the current NeXT ™-based version of UWGSF3, graphics standards including PHIGS, PHIGS+, GKS, and Renderman are being evaluated for implementation. The use of one of these standardized graphical programming interfaces, with the low level operations written to exploit the multiple FPP architecture, will further enhance UWGSP3's utility for 2-D and 3-D graphics applications.

Conclusion

With its increased display resolution, enlarged frame buffer storage, multiple floating point processors and intuitive graphical user interface, UWGSP3 represents an innovation in image computing workstation design and a significant step towards providing affordable real-time display and processing for a variety of applications. It provides an integrated platform for more acceptable and productive end-user environments for both image processing and graphics in the future. In this paper, we have described the basic architecture of UWGSP3, and several solutions to medical imaging applications including use as a PACS workstation, image processing and graphics computational engine, and a multiple display electronic alternator. With the hardware implementation and low-level software now completed, the task of creating the application software to achieve the UWGSP3's potential in these areas and others will extend into the months ahead.

Acknowledgements

We would like to acknowledge the efforts of Clark Haass of the ICSL for his work on the interactive filtering package user interface. We would also like to acknowledge Texas Instruments for their help and the donation of the prototype chips and development tools used in UWGSP3; in particular, we wish to thank Mike Asal of the GSP Group in Houston and Ron Drafz of the VLSI Group in Dallas.

References

J. D. Austin and T. Van Hook, "Medical image processing on an enhanced workstation," SPIE Medical Imaging II, 914:1317-1324, 1988.

D. Beard, J. L. Creasy, J. Symon and R. Cromartie, "Experiment comparing fllm and the FilmPlane radiology workstation," SPIE Medical Imaging II, 914:933-937, 1988.

P. Chinn, R. M. Pier, L. A. DeSoto, H. G. Zieber, D. A. Verheiden and Y. Kim, "PC-based floating point image processing system," Electronic Imaging Spring '88, pp. 233-238, 1988.

H. S. Choi, H. W. Park, D. R. Haynor and Y. Kim, "Development of a prototype electronic alternator for DIN/PACS environment and its evaluation," SPIE Medical Imaging IV, 1234:in press, 1990.

S. Jobs, Keynote Address at the University of Washington Computer Fair, March, 1989.

Y. Kim, D. R. Haynor, 0. Saarinen, A. H. Rowberg, and J. W. Loop, "Preliminary PACS experience at the University of Washington," Electronic Imaging Spring '89, pp. 142-147, 1989.

J. C. Gee, L. A. DeSoto, D. R. Haynor, Y. Kim and J. W. Loop, "User interface design for a radiological imaging workstation," SPIE Medical Imaging III, 1093:122-132, 1989.

K. McNeill, G.W. Seeley, K. Maloney, L. Fajardo, and M. Kozik, "Comparison of a digital workstation and a film alternator," SPIE Medical Imaging II, 914:929-932, 1988.

K. G. O'Malley, "An iterative approach to development of a PACS display workstation," SPIE Medical Imaging III, 1093:293-300, 1989.

Texas Instruments TMS34020 User's Guide, 2nd Review Copy, August, 1989.

Texas Instruments TMS34082 Floating-Point Processor Product Preview, September, 1989.

Appendix F

Parallel Signal and Matrix Processing with the TMS34082

This application note will discuss and analyze a TMS34082 based parallel architecture.

E. M. Dowling and Z. Fu
Erik Jonsson School of Engineering and Computer Science
The University of Texas at Dallas
Richardson, Texas 75083–0688

R. S. Drafz
Texas Instruments, Incorporated, M/S 8316
Dallas, Texas 75265–5303

Introduction

VLSI floating-point processor technology is evolving to meet the increasing need to execute sophisticated algorithms at higher and higher rates. Architectural advances in floating-point pipelines and processor organization have led to the TMS34082's high speed arithmetic core and its RISC control structure. However, some applications require much higher speeds than provided by a single TMS34082. A parallel processing solution may be the answer.

The goal of parallel processing is to speed computation by designing the appropriate number of processors into the system. These processors each work on pieces of the algorithm separately, and pass intermediate results among themselves. The simplest and most common form of parallel processing is to assign each processor a different tasks. For example, in a typical computer system, there may be a simple processor in the keyboard, a CPU, and coprocessors for memory management and floating-point operations. Of interest here are the parallel processing architectures that use many identical processors to solve a single numerical problem.

Some common architectures that achieve this are shared memory machines. (Sequent and Multi-Max), distributed memory machines (Ncube, IPSC) and systolic arrays (WARP) [1]. They all use duplicate processors, but have different storage, communication, and programming schemes. Some parallel architectures require all processors to execute the same instructions, but to work on multiple data streams (SIMD = Single Instruction Multiple Data.) Others allow each processor to act independently by providing multiple instruction streams (MIMD = Multiple Instruction Multiple Data.) Many architectures exist to solve numerical problems such as those that arise in scientific computation and signal and image processing applications. Many experimental machines have been targeted to these structured computation intensive applications [1] [3] [4] [5].

In this application note we will design and analyze a TMS34082 based parallel architecture. The architecture will be a MIMD hybrid shared /distributed memory machine that supports message passing as well as systolic data streaming. This structure provides maximum flexibility at a relative low cost. In addition, the system can be scaled so that any number of processors can be added as the application requires. The system reaches a peak of 40 MFLOPs per processor, and sustains a rate of 10 MFLOPs per processor on structured numerical algorithms. For example, if an algorithm must be executed in real time at 150 MFLOPs, a system with about fifteen or sixteen processors is needed.

Parallel architectures are measured in terms of their speed increase over a sequential architecture using the same type basic cell. (A cell can be thought of as a processing unit that would be the CPU/memory/I/O system in a sequential machine). In the MIMD system, the I/O portion of a cell is generally connected to other cells as well as other conventional I/O devices such as disks, A/D converters, etc. Parallel architectures performance is limited by dependencies found in many algorithms, or, more fundamentally, found in many mathematical problems. These dependencies might cause the parallel architecture to perform less efficiently than a straight sequential architecture due to communication overhead. If the problem itself is not inherently serial, then a parallel algorithm must be designed to solve the problem. Often this parallel algorithm can be derived from the sequential algorithm by rescheduling the computations so that the algorithm dependencies are satisfied, but many independent calculations will be computed at each step.

At this stage of design, the algorithm becomes linked to the underlying architecture. The most challenging part of the design is not to decide which computations can computed in parallel, but to minimize communication delays and waiting time. This requires the algorithm designer to match the dependence structure of the algorithm to the systems communication structure and processor granularity.

The parallel algorithm must be analyzed to see how it performed. The optimum is to use n processors for an n times speed increase (linear speed-up.) However, Amdahl predicted that most algorithms will have a logarithmic speed-up because their communication burden typically grows exponentially. Fortunately, linear speed-ups can often be attained on modern architectures solving well structured practical numerical problems, due to their regular communications requirements.

As we see, there is a tight interplay between the algorithm and the architecture. In a sense, the algorithm is mapped onto the architecture. It is our intent to design an architecture that can efficiently support many different algorithms. A hybrid approach as discussed in the next section was taken to achieve this level of flexibility. The architecture provides all point-to-point and broadcast paths through a single bus. A bidirectional ring of FIFO buffers connects adjacent processors so that high throughput can be achieved. The architecture design was driven by the matrix multiplication, FFT, QRD, and SVD algorithms. Simulation was used to arrive at an architecture that could support all of these representative algorithms.

Once the basic architecture is established, the cell must be carefully designed. As stated above, the performance of the parallel architecture is based on the speed increase over a single cell. If the cell is poorly designed and slow, the increase will be negligible. The cell designed here is based on the TMS34082 acting in host-independent mode. The MSD bus is used for instructions, while the LAD bus is used for data. The TMS34082 requires some sort of addressing assistance on the LAD side, an address latch at the very least. With this assistance, the TMS34082 has a Harvard architecture so can be made to maintain a steady instruction stream while manipulating data on the LAD side. This capability is of paramount importance in a TMS34082 system, and does not come without careful attention. The TMS34082 does not have any organic LAD addressing support. All LAD addresses must be computed in the floating-point core. Furthermore, when the C-compiler is used, local variables are stored on the MSD side and are accessed through stack operations. Even the stack pointer manipulations are carried out in the floating-point pipe, causing 'bubbles; and reducing performance. To bring the performance of the cell up to the full TMS34082 capabilities, a more sophisticated LAD bus controller will be specified. This controller will have its own register set and an integer unit to perform pointer manipulations under the control of an extended instruction field. The same LAD bus controller will be capable of routing data to more than one destination in a single bus cycle and will be able to move data while the TMS34082 is performing other functions such as floating-point loops.

The HARP Architecture

The design of the Hybrid Array Ring Processor (HARP) architecture was guided by a set of goals. First, the architecture was designed to perform a wide range scientific and DSP oriented algorithms. Second, it was designed to be scalable and expandable so that applications specific systems could be easily configured to the user's needs. It was also decided that the architecture should support both single- and double-precision IEEE standard floating-point arithmetic. The principle concern with the interconnection structure was that it had to support both high throughput and fast point-to-point paths. The interconnection topology was to be as simple as possible, yet had to support the target algorithms cleanly and efficiently. From a software perspective, the architecture had to be programmable in an extended version of C, much like hypercubes. Also the architecture needed to support an operating system such as UNIX. Finally, a version had to be implemented that could be accurately simulated in software so that performance measurements could be made.

Matrix multiplication, FFT, QRD and SVD algorithms shaped the architecture. A simulator was built using the Rice Parallel Processing Testbed (RPPT) package along with the TMS34082 C-compiler and chip simulator. The RPPT simulator can run programs written in a superset of C, called Concurrent C (CC). An architecture model was written so that waiting time, date transfer delays, and the overall effects of the system topology could be measured. Profile information from the TMS34082 simulator is fed into the RPPT simulation so that the overall simulator measures the actual cycle counts of the parallel program executing on the architecture. Architectural modifications were made whenever limitations and bottlenecks were revealed by the simulation process.

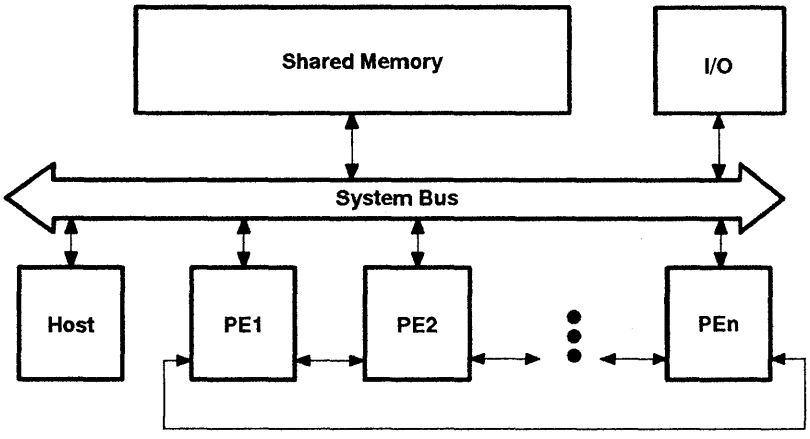


Figure 1. System Architecture

The overall HARP architecture is depicted in Figure 1. The host can be any subsystem that can run the desired operating system. For purposes of simulation, the host was considered to be a 33 MHz MC68030 with associated support hardware. The system bus can be any high-performance bus, but for simulation purposes was taken to be the native MC68030 processor bus running in asynchronous mode. The architecture, however, can be readily implemented for an open bus standard such as VME, Future Bus, etc. for example, a VME based version would be built around an available single-board UNIX engine, memory boards, and I/O cards. PEs would be added in groups of four per card. Each card would have a ring port in and a ring port out connector on the front. Thus flexible systems can be configured to meet specific processing, memory and I/O requirements. Application programs are written in CC to run on P processors, so that the same program will run on systems with different numbers of processors.

The hybrid aspects of the architecture are highlighted when looking at the system's programming models. From a global perspective, the host sees a conventional system that is augmented with smart memory segments as outlined in the memory map of Figure 2. The host must load these segments with code and data and read out the results. From the PE/bus perspective, the system takes the shape of a shared memory machine. Using the shared memory mode, processes are forked to the various nodes and communication primarily takes place over the bus where synchronization is maintained using semaphores and join constructs. The system can also be viewed as a message passing machine. Here processes on the nodes communicate using the send- and receive-message commands. The messages can be routed through the ring or over the bus. Finally, the system can be programmed as a linear or ring systolic array (with broadcast.) The systolic mode is the fastest mode if local PE to PE communication is required by the application algorithm. In this mode a steady data stream flows through the ring network in lockstep with processing. The cell architecture is optimized so that systolic communication and computation overlap.

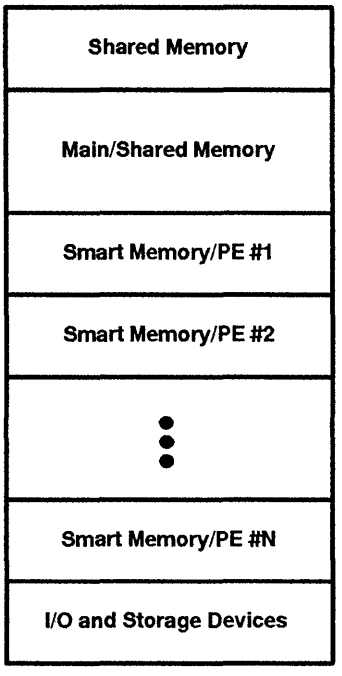


Figure 2. System Memory Map

The PE is depicted in Figure 3. It consists of a TMS34082 floating-point RISC processor, a 30ns 512K word memory bank, a bus interface, a local bus controller, and a 35ns 32-bit BIFIFO that connects the PE[k] to the PE[k+1]. A full system bus interface implementation includes a local bus arbitration protocol that allows any PE to directly access any other PEs' local memory. The PE is built around the TMS34082's Harvard architecture. The program is stored on the Micro Store Data/Address (MDS & MSA buses) side while the data is stored on the Local Address and Data (LAD) side. The TMS34082 architecture consists of a high-performance FPU, a register file of twenty 32-bit (or ten 64-bit) registers, and a microsequencer. The TMS34082 does not have an address arithmetic unit nor does it have an address bus on the LAD side. In order for the system to reach high performance levels, an external LAD bus controller was designed to compute LAD addresses and provide low level timing and control signals to the devices attached to the LAD bus (FIFOs, SRAM, bus interface).

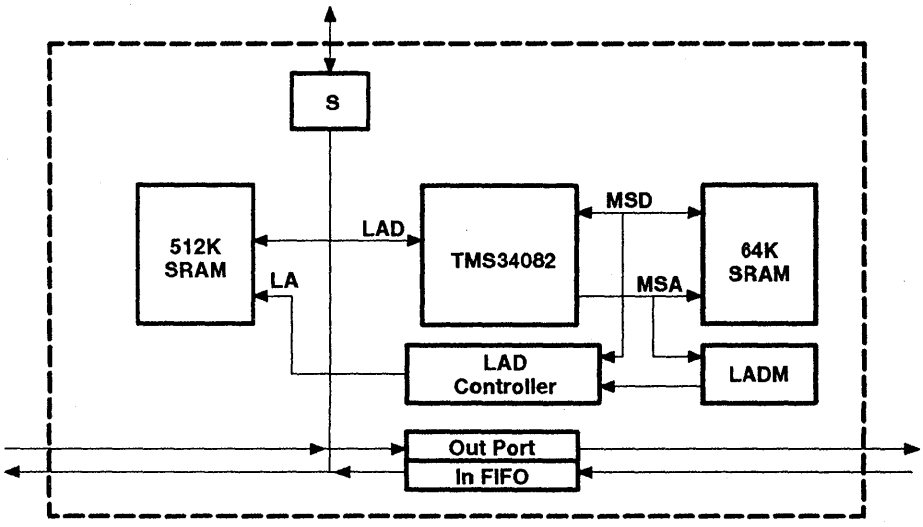


Figure 3. PE Architecture

The LAD controller provided over an order of magnitude performance increase. With the controller, the TMS34082 is capable of performing an entire load or store operation and pointer update in a single clock cycle. The LAD controller can interleave accesses from the memory and the FIFO, as is often needed. The LAD bus controller also can be configured to accelerate systolic ring communication and local memory operations. Suppose the TMS34082 of PE[k] is to read a word from the FIFO connected to PE[k–1]. In many applications it will be also necessary to pass the datum onto the FIFO that is connected to PE[k+1] and possibly store the received datum in local memory. If both are required, the LAD controller will generate the signals for the FIFO[k–1] read, the FIFO[k+1] write, and the memory write all in the same TMS34082 read cycle. The LAD controller's FFT address generator speeds FFTs by over a factor of 100.

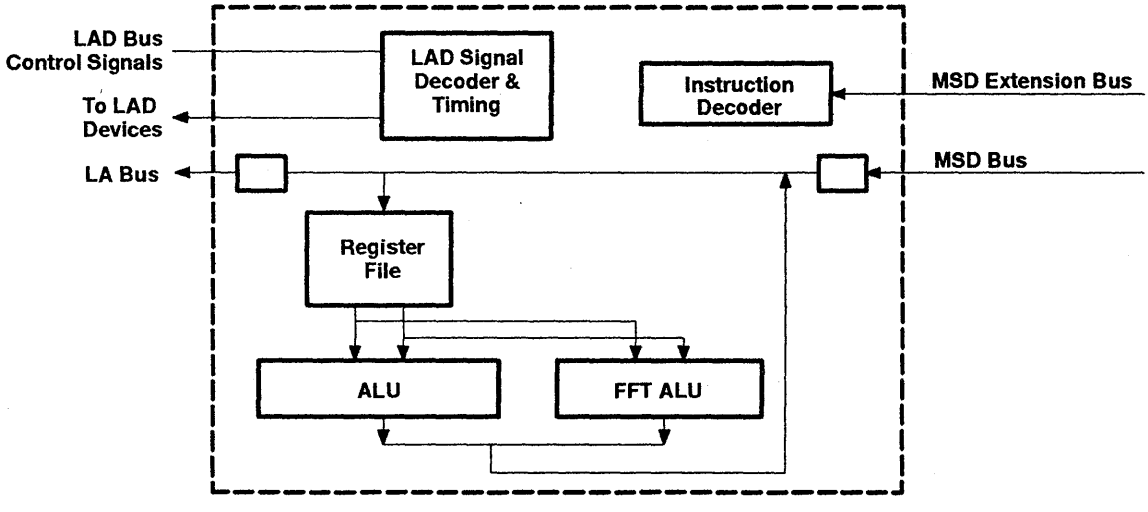


Figure 4. LAD Bus Controller Architecture

Different LAD controller designs have been considered. Some are algorithm specific, like the FFT, and some are more general. The most general LAD controller, as depicted in Figure 3, can be configured and performs LAD addressing under program control. The architecture of the LAD controller is shown in Figure 4. It consists of a register file, and addressing ALU (complete with FFT instructions), a LAD bus timing and control signal decode section, and an instruction decoder. The LAD controller accepts an instruction stream from an extended microcode field on the MSD bus. The registers may be loaded from the MSD bus as immediate operands. In a sense, the TMS34082's instruction set is expanded to include LAD addressing modes. The LAD addressing related fields in the instructions are stored in a separate four-bit memory (LADM) whose address lines are connected to the MSA bus.

TMS34082 Host-Independent Mode Optimizations

The TMS34082 operates in either the coprocessor or the host-independent mode. It is most commonly used as a floating-point accelerator for the TMS340 graphics processor and is less commonly used in the host-independent mode, where it acts as an autonomous floating-point RISC processor. While the HARP architecture does assume a host processor to run the operating system, it employs the TMS34082s as loosely coupled processors operating in the host-independent mode. In this section some of the more striking aspects of designing a system around the TMS34082 in host-independent mode are discussed. The TMS34082 is capable of achieving very high computational throughputs.

The TMS34082 is truly a compact architecture, fitting somewhere between a conventional RISC and a dedicated floating-point unit. It lacks many common microprocessor features such as an on-board parallel address generator, while it provides an assortment of high-performance floating-point operations and subroutines such as division and square root. The key to success with the TMS34082 in host-independent mode is to keep it fed with data. If the TMS34082 is asked to manage the data stream into the chip, performance will be downgraded. Also, special care must be taken with loop management.

The first thing to take into consideration when developing a system for the host-independent TMS34082 is its Harvard architecture. The chip is designed to accept an instruction stream from the MSD (microstore data) bus and a data stream from the LAD bus. In coprocessor mode, the TMS340 has no problem keeping the TMS34082 fed with data because the LAD bus is directly connected to the TMS340 bus. However, in host-independent mode, special care must be taken to keep the data flowing on the LAD bus under the control of something other than the TMS34082. The reason for this is two fold. First of all, the TMS34082 LAD bus has both address and data multiplexed onto the same bus. Without external support, this means that an address must be first sent out to an external address latch prior to any LAD access, reducing the LAD bus bandwidth by 50%. A more sever problem arises if the TMS34082 is used to generate data addresses as would be done in a conventional architecture. Because the TMS34082 does not have a separate integer addressing unit, all pointer updates must be computed in the FPU core. This means that the floating-point pipeline must be broken every time an external access is required. It turns out that non-judicious use of the TMS34082 for pointer updating can easily downgrade performance by an order of magnitude or more. Thus it is recommended that an external bus controller be designed into the system that performs the pointer manipulations needed to support the algorithms that will run on the target system.

Once the hardware has been configured, it is important to optimize the software. The first rule of thumb is to make judicious use of registered variables. Often compilers will use frame pointers and related pointer arithmetic to access local variables. As mentioned previously, if the TMS34082 is to perform pointer arithmetic, it will have to do it in the floating-point pipeline at great expense. The compiler will often place local variables on the stack which is located on the MSD side. This means load and store operations take two cycles each instead of the one cycle on the LAD side. More importantly, each local variable access will involve several external accesses to compute the stack pointer relative address of the local variable. Due to these considerations, it takes the TMS34082 11 cycles to compute k = k + 1 if k is a local variable defined on the MSD stack. On the other hand, if k were declared as a registered variable, the same operation would require only one cycle. Thus great performance dividends will be paid to those who put as many of the most often used local variables of each routine into registers.

The old axiom that 90% of a program's time is spent on 10% of the code is very true when it comes to numerical routines. In fact, most numerical routines spend 90% or more of their time in tight inner loops. For example, the inner loop of the routine to multiply two 256 x 256 matrices on a single TMS34082 would be entered and exited 65,536 times. At each iteration, one multiplication and one addition is performed, which can be computed by the TMS34082 in a single cycle using the *mult.add* command. Now consider the overhead needed to run the loop. If the loop counter variable were located on the MSD stack it would take 11 cycles to increment the loop counter, it would take another 12 cycles to check to see if the loop terminated. In addition, using the FPU core to perform loop counter iterations and stack pointer manipulations forces the compiler to use separate multiply and add operations and store intermediate results. Thus while the TMS34082 provides the ability to compute a floating-point multiply-accumulate a single cycle, an unoptimized loop might spend 30 cycles each iteration in loop overhead. If care is not taken, loop overhead alone can reduce the performance to 3.33% efficiency. This figure does not even account for data accesses. The loop overhead can be reduced to about four cycles per iteration if registered variables are used. Even better, it can be reduced to one cycle through the use of the LOOPCT register and the *cjmp.d* instruction. The *cjmp.d* instruction is a decrement and branch instruction that decrements LOOPCT, compares it against zero, and takes the appropriate branch all within a single cycle.

Now take data accesses into account. An inner-product loop is set up by initializing LOOPCT with the inner-product length, clearing the accumulator, and loading the base addresses of the two input data arrays. First consider the case where the pointers are loaded into an external LAD controller. Two data loads are performed in two cycles while the LAD controller autoincrements the pointers for the next loop iteration. Next a *mult.add* instruction is used to perform the multiply accumulate in a single cycle. Finally, the *cjmp.d* is used to decrement the loop counter and branch to the beginning of the loop. This implementation required four cycles per loop iteration and a LAD controller that could interleave two increment pointer registers. Now consider an implementation that does not use an external LAD controller, but does use LOOPCT and registered variables for the loop pointers. The loop starts out by performing two pointer additions with the output sent to the LAD bus and two loads; four cycles. Next the multiply and add are computed in two instructions because the FPU pipeline had been interrupted. The *cjmp.d* is the last instruction in the loop. This loop had a total count of nine cycles.

The two above loops can sustain 10 MFLOPs and 4.44 MFLOPs respectively. Slightly enhanced performance can be achieved many loop iterations are in-line coded into a single loop iteration. If the vector length is 100, then the inner product could be computed in ten loop iterations if ten multiply accumulates are performed in each loop iteration. With the external LAD controller, twenty loads are followed by ten *mult.adds* and one *cjmp.d.* This reduces the number of *cjmps* by nine, but adds additional loop end condition checking overhead if the number of loop iterations is not a multiple of ten. Anywhere between one and ten multiply accumulates can be performed in the inner loop depending on the divisors of the inner product length. Using this optimization technique, the sustained inner product performance can be raised from 10 MFLOPs to 15 MFLOPs.

Experience shows that one must recognize what the TMS34082 is and what it is not. It is a high-performance floating-point unit that can execute floating-point code efficiently. It is not a general purpose processor with a full set of addressing modes and parallel on-board executions units. Great speed-ups in compiler generated code can be easily achieved through judicious use of registered variables. Hand optimized assembler level optimizations can be attained by using the LOOPCT register and the *cjmp.d* instruction. Further speed-ups come through the use of and external LAD side address generator. Once a system architecture is defined, systems level optimizations can be made to overlay various bus operations into the same cycle.

Algorithms

In this section several algorithms will be briefly discussed. First consider the problem of multiplying the matrices $A \in R^{m \times r}$ and $B \in R^{r \times n}$ to form the product $C \in R^{m \times n}$ on a system with P PEs. At the start of the algorithm, A and B are stored in the shared memory. At the end of the algorithm, the product matrix C is returned to the shared memory. The first step of the algorithm is for the host to move the columns of B into the PE s using the system bus. Column b_j is moved to PE[j mod P]. This column will be used to compute $c_j = Ab_j$ so that c_j will be accumulated on PE[j mod P]. The matrix A is moved into the array at PE[0] and PE[0]'s right output buffer concurrently. The inner product of row a_i and each resident b_j is formed and stored as c_{ij}'s. PE[k] reads a word or row a_i from PE[k–1], one word at a time, directly into the TSM34082's FPU pipeline and stores the row in memory for future use and transfers it to the FIFO connected to PE[k+1] all in the same cycle. The rows of A stream through the system until the trailing row cycles through. Due to the ordering of computations, PE[0] will finish first, then PE[1] etc. Once PE[0] finishes, it sends its results over the bus to the system memory. Then PE[1] will follow suit, then PE[2] ... etc. on down the line.

Next consider the radix-2 decimation in time (DIT) FFT algorithm. Assume that the number of PEs, P is a power of two. Also assume that the LAD bus controller is capable of performing FFT address generation in addition to the autoincrement mode used in the previous algorithm. For purposes of illustration, suppose that a N = 1024 point FFT is to PE performed on P = 8 processors. The algorithm is outlined as follows. First decimate the time series into eight 128-point subsequences and send the ith subsequence to PE[i] over the system bus. Next each node computes a 128-point radix-2 DIT FFT on the local subsequence. These sequences are built back up using standard binary tree recombination with twiddling. The tree is viewed as having the root node in processor zero $log_2(N)$ iterations into the future. At the first iteration, each PE is considered to be a leaf of the tree. At this iteration each PE[2k + 1] sends its results to PE[k] for k = 0...(N/2) – 1. The even PEs then perform the twiddle and recombination operations so that the even cells now have 256-point sequences. At the next iteration, PE[4K+2] sends its results to PE[4k] for k = 0...(N/4) – 1. Now the mod 4 PEs twiddle and recombine. Next PE[4] sends its result to PE[0] and PE[0] assembles the final 1024 point result. The communication of the algorithm is not local, but the algorithm permits the data to be routed through idle cells so that a negligible penalty is paid for the nonlocal communication. The nonlocal communication only costs one cycle of delay per route-through node; the data rate of the data stream is not effected. Simulation studies have shown that this extra cost has a negligible effect on performance. The simulation studies did show that performance was reduced due to the nonsequential access requirement within a given local vector computation. The nonsequential addressing forces one to send entire messages instead of single elements at a time transparently. Also, as the recombination process progresses, more and more processors become idle.

Another version of the FFT was studied that was able realize the full potential of the system. Most applications that require an FFT actually require many FFTs. For example, a real-time processing system might require that 1024 point frames of an incoming signal be computed continuously. In image processing, a 512 x 512 pixel FFT can be computed by first performing 512 512-point column FFTs followed by 512 512-point row FFTs. Similarly, spectral based PDE solvers used in scientific application s require large numbers of 1-D FFTs to compute a single 3-D FFT. The course granularity of the system allows each separate PE to compute an FFT separately. This is a pure smart memory algorithm. The host loads the PEs with data and pulls out results. If enough processors are in the system, the overall computation rate is limited only by the amount of time it takes to load and unload a single smart memory segment with a complex data vector.

The next algorithm considered was the QRD. The QRD provides an alternative way to solve linear systems. The standard algorithm used to solve linear systems is Gaussian elimination with partial pivoting. The pivoting portion of that algorithm degrades performance in the HARP architecture, but a Householder QRD maps quite well. $A \in R^{m \times n}$ has a factorization A = QR where $Q \in R^{m \times m}$ is orthognal and $R \in R^{m \times n}$ is upper triangular [11]. Consider the case where m = n and rand(A) = n. Write Ax = b as QRx = b so that multiplying on both sides by Q^T gives the triangular system Rx = Q^Tb which can be solved by back substitution. Note that multiplying both sides by Q^T is equivalent triangularizing A by a sequence of orthognal transformations and applying these same transformations to b. In the case where m>n, this procedure may also be applied to solve linear least squares problems.

Suppose $A \in R^{m \times n}$ is to be decomposed on a P processor system. Assign column a_j to PE[j mod P]. If P does not divide n, then some PEs will have extra columns. First set the iteration variable, k, and proceed as follows. At iteration k, the PE containing a_k computes the vector $v_k \in R^{m-k+1}$ such that the bottom (k – m)-element subvector of a_k, denoted $a_k(k : m)$ satisfies $H_1 a_k((k : m) = \alpha\varepsilon_1$ where ε_1 is the k-order unit standard basis vector and $\alpha = \| a_k (k : m) \|$. The kth transformation must be applied only to columns k through n. So v_k is sent down the ring to the right from the PE where a_k resides. When the head of the v_k data stream arrives, the remaining PEs perform the transformation, $a_j (k : m) = a_j (k : m) - (vT_k \, a_j (k : m)) \, v_k \; \forall \, j > k$ to the local columns. The algorithm is essentially a waveform algorithm where the computation wave propagates to the right and a trail of results (R) are left behind. If the matrix Q is desired, the v-vectors may be saved so that Q is available in factored form. If the algorithm is used to solve a linear or linear least squares system, the vector b is augmented as the last column of A and loaded accordingly.

The final algorithm to be discussed is the SVD. Hestenes's method [6] [9] for computing the SVD has received much attention lately in the literature due to its parallel nature. The Hestense method is a one-sided Jaccobi algorithm that operates by applying orthognalizing plane rotations to all pairs of columns of the matrix $A \in R^{m \times n}$. A sequence of all pairs of such rotation is called a sweep. The algorithm can be shown to converge after a sufficient number of sweeps have been applied. Once the algorithm has converged, the matrix, A, will have been transformed via orthognal transformations to another matrix, $B \in R^{m \times n}$, whose columns are orthognal to each other. If the product of the sequence of orthognal transformations is collected in $V \in R^{n \times n}$, then we have $AV = B$. It is trivial to next factor B as $B = U\Sigma$ gives $A = U \Sigma V^T$, which can be seen to be the SVD of A. We do note that this algorithm generated $U \in R^{m \times n}$ and $\Sigma \in R^{n \times n}$ instead of $U \in R^{m \times m}$ and $\Sigma \in R^{m \times n}$ but that no information was lost.

Several systolic array algorithms have been devised to perform the Hestenes SVD algorithm on a linear bidirectional systolic array [2] [8] [10]. The methods are based on a theorem that states that the order in which all the pairs columns are orthognalized does not affect the overall convergence of the Jaccobi algorithm [8]. Algorithms are designed by selecting an ordering where groups of P pairs can be computed at each time step on P processors. The key to a successful ordering is that the next group of P pairs in the ordering can be generated by local shifts of columns between processors. Figure 5 shows how columns are switched in order to generate such an ordering on a P-element bidirectional array. In the figure each PE is assumed to have two vector registers, VRa, and VRb which each hold a column of the matrix. If the columns are loaded into the vector registers, and permuted as depicted in the figure, one sweep will be computed every N-2 update cycles. At the end of the sweep, the updated columns will return to their original position in the array, ready for the next sweep.

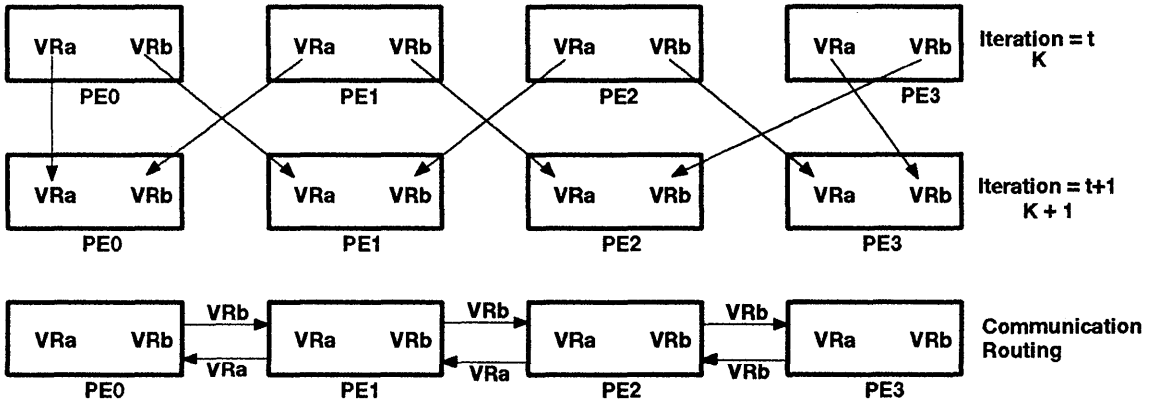


Figure 5. Parallel Jaccobi Updating on a Systolic Architecture

It is clear that this algorithm can be directly implemented on the HARP architecture. The bidirectional ring is sufficient for communication, and the vector registers can be implemented as buffer areas in the local SRAM. Each node must first compute the Jaccobi rotation matrix, apply it to the local columns stored on that node, and send the results to adjacent nodes as indicated in the figure. All nodes perform the same function except the end nodes. The shifting of data between the two data buffers on PE[N-1] can be accomplished by a single pointer swap. If the number of columns in the matrix exceeds 2P, then the algorithm is slightly more complicated. Now PEs will have to do the job of more than one PE. The complication comes in the form of pointer housekeeping and additional conditional statements. Only the computations that represent the boarder PEs of the sub-array need to communicate with the neighbor PEs, the internal nodes of the subarrays just exchange pointers.

Simulation Results and Performance Analysis

A simulator was constructed using the RPPT simulation package. The RPPT package consists of a CC compiler, an architecture modeling / analysis package, and a facility to bind CC programs to the architecture model. Once a program and architecture have been bound, RPPT runs a simulation of the program running on the architecture. While doing so, RPPT keeps track of time using a parallel time construct. It is able to account for delays caused by bus contentions, processes waiting for input, data transfers across a bus or communication channel, and processors executing code. The above mentioned delays can be caused by either the parallel program or the underlying architecture or both. In order to account for the time spent by processors executing code, RPPT converts the CC program into assembly code and assigns to each basic block a cycle count (a basic block has one entry, one exit, and each instruction in the block is performed exactly once.) It then inserts an instruction at the front of each basic block that increments a cycle counter variable by the number of cycles spent in that basic block. This act is called profiling, and is done to the native MC68020 assembly code generated for the execution of the simulation of a SUN3 platform. In order to make RPPT count TMS34082 cycles, the node program is recompiled on the TMS34082 C compiler, translated to assembly code, and profiled with using the TMS34028 simulator in single step mode. The basic blocks of the MC68020 assembly code are then cross profiled by replacing the MC68020 cycle counts with the TMS34082 cycle counts. The key is that both programs execute the same C code so are essentially the same. The architectural modifications of the LAD bus controller are brought into the simulation here by updating the cycle count numbers to reflect the elimination of the cycles that are actually performed in parallel by the LAD controller.

The matrix multiplication was analyzed first. It showed us that the maximum sustainable throughput of a node was essentially limited to 10 MFLOPs. This is so because in the inner-loop of a long inner product required two loads, a *mult.add*, and a conditional jump. Thus two FLOPs are performed every four cycles, so that at 20 MHz, the TMS34082 can continuously compute data streams at a rate of 10 MFLOPs. This limit can be raised to up to 15.5 MFLOPs if the exact number of elements in the inner product is known ahead of time. For example, if the inner product length were 100, the loop iterations consisting of 20 loads, ten *mult.adds* and one conditional jump would each perform 20 FLOPs every 31 cycles. The program used in the simulation was written for the general case so that the nodes were essentially limited to 10 MFLOPs sustained throughput rate.

The simulator was used to measure the efficiency of the HARP and to study the effects of matrix size and the number of processors in the system. The simulation accounts for the time to move the inputs to the nodes from the shared memory, compute the results, and move the results from the individual nodes back into the shared memory. The simulator counts all cycles to include addressing, loop management, testing of conditions, etc. It gives an indication as to the amount of time spent performing FLOPs and the amount of time spent on communication and overhead. Figure 6 shows a plot of the average MFLOPs achieved by a ten element array running matrix multiplications. Note that as the size of the matrices increase, the overall performance of the system approaches the theoretical limit of 100 MFLOPs. The reason that performance is not as close to the limit for smaller matrices, is that the I/O and program overhead becomes more significant. Figure 7 show a plot of the performance measured in average MFLOPs for systems running a 128 x 128 matrix multiplication using P processors. Note that for up to 32 processors (the largest array the simulator could handle) there is a linear speed-up as more processors are added. This violation of Amdahl's law is predictable because the communication overhead of the algorithm/architecture combination clearly does not increase exponentially as more processors are added.

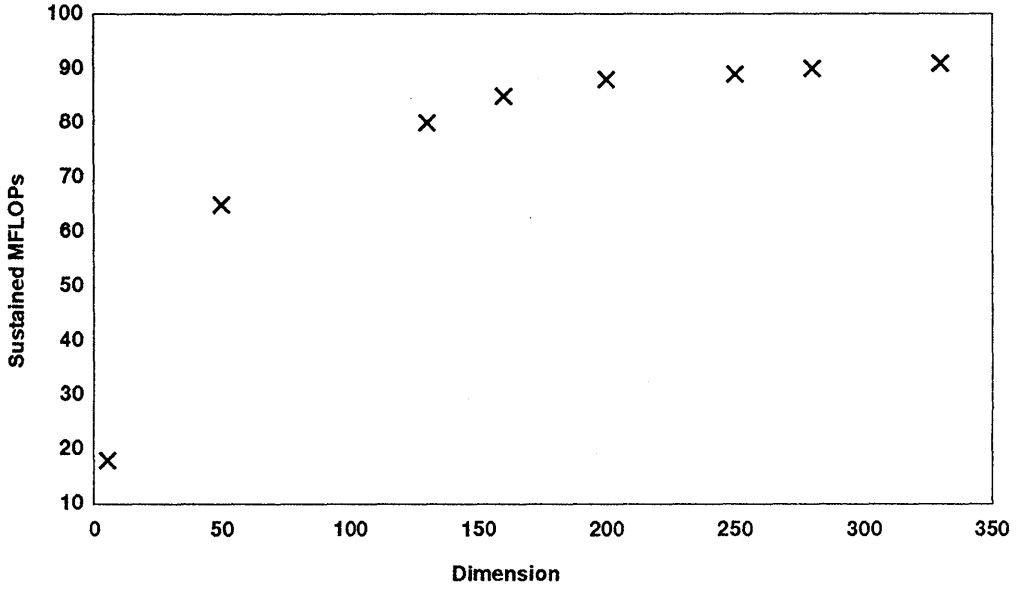


Figure 6. Matrix Multiplication Performance on 10-Processor Systems

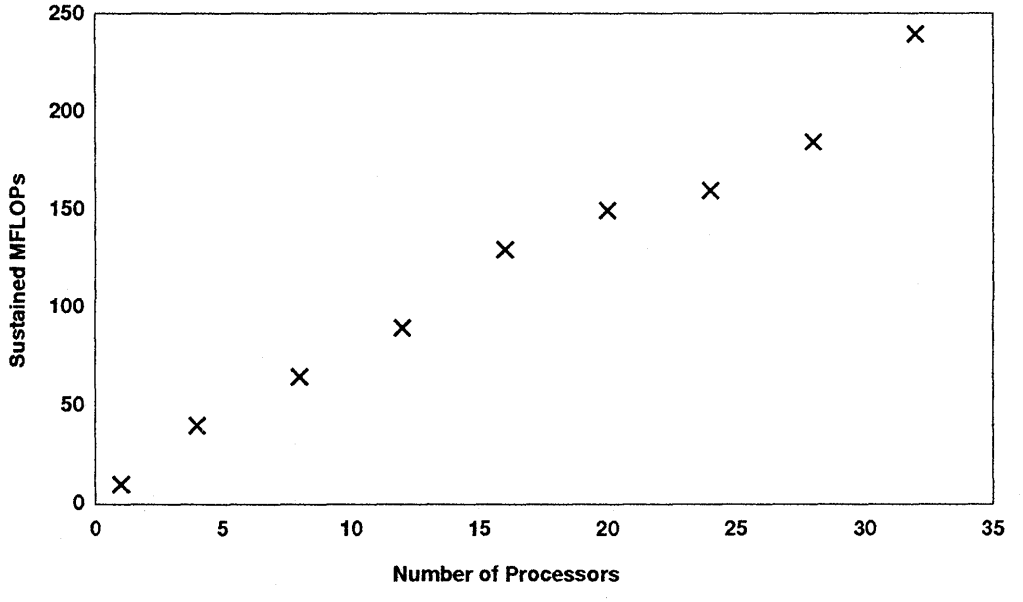


Figure 7. 128 x 128 Matrix Multiplication on P-Processor Systems

The next algorithm that was implemented was the FFT. The LAD controller provided hardware support for the radix-2 addressing scheme. In the first set of experiments, single FFTs of various lengths were computed on different sized arrays. The results are summarized in Table 1. We note that the sustained MFLOPS on a single processor is within 75% of the maximum sustainable throughput for a single node. The pay off for adding more processors, however, is less pronounced than in the matrix multiply algorithm. This is due to the fact that communication overhead can not be completely overlapped with computations. Thus, as more processors are added, the execution time of the algorithm decreases, but the efficiency of the system also decreases.

The pipeline FFT algorithm was also analyzed. Here the number of processors was determined to form an FFT pipeline that maximized overall performance. Using this number of processors, performance was limited only by the system bus bandwidth. Table 2 shows the results of the second set of experiments for transform lengths from 256 to 4096 that were performed on the optimum sized arrays. For each transform length/ array size pair, the table lists several parameters. First the 1-D pipeline FFT effective computation time is listed followed by the maximum sampling rate that could be accommodated for the various transform lengths. The next column shows how much time it takes to compute an N x N 2-D FFT using the row/column algorithm. The sustained MFLOPs achieved for each 2-D FFT is listed last. The maximum attainable sustained computation rate can be taken to be 10*P MFLOPs, were P is the number of processors in the array. The efficiency is the measured sustained MFLOPS divided by the total attainable MFLOPs. The simulation shows the system efficiency ranges from 67.8% for the 256 x 256 transform to 80.8% for the 4096 x 4096 transform.

Table 1. Distributed FFT Performance Results.

	N=512	N=1024	N=2048	N=4096
P=1	t=3.69 SM=7.47	t=7.94 SM=7.74	t=17.0 SM=7.96	t=36.2 SM=8.14
P=4	t=1.64 SM=16.86	t=3.39 SM=18.12	t=7.03 SM=19.23	t=14.59 SM=20.22
P=8	t=1.40 SM=19.79	t=2.82 SM=21.77	t=5.74 SM=23.55	t=11.72 SM=25.17
P=16	t=1.31 SM=21.15	t=2.59 SM=23.74	t=5.18 SM=26.09	t=10.43 SM=28.26

P = # of processors. t = time in milliseconds and SM = sustained MFLOPS.

The column-systolic Householder QRD was also executed on the simulator. Figure 8 shows the sustained MFLOP rating of the QRD as a function of matrix dimension on a ten processor system. Note that the algorithm approaches the 100 MFLOPs maximum sustainable capacity of the system nearly as fast as the matrix multiplication algorithm, but levels off to 90 MFLOPS due to additional serial threads in the QRD algorithm. Figure 9 indicates that for large matrix size, that the algorithm has linear speed-up as more processors are added. This is due to the fact that communications and computations are nearly full pipelined. It is also due to the modulo P wrapping of the columns to the array and the use of the external ring connection. This mapping strategy achieves nearly perfect load balancing and allows the inherent dependencies of QRD to be effectively eliminated by allowing the system to execute more than one iteration of the algorithm at a time.

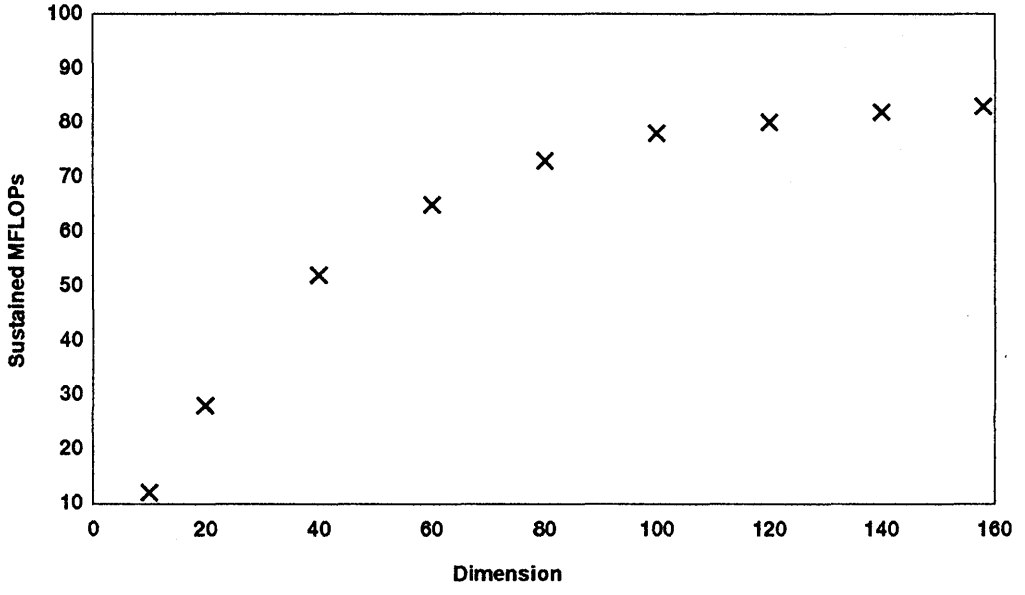


Figure 8. QRD Performance on 10-Processor Systems

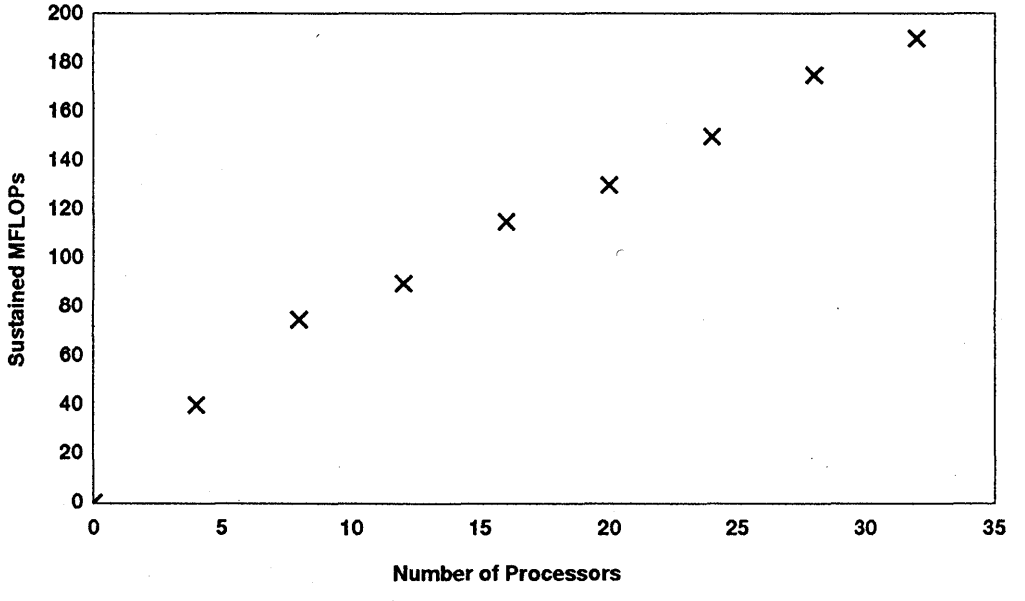


Figure 9. 128 x 128 QRD on P-Processor Systems

The last algorithm that was implemented to date was the modified Hestenes-Luk SVD. The algorithm was found to be efficient for large matrices especially if the number of processors was large. As Figure 10 indicates, as more and more columns are mapped to each processor, the efficiency of the algorithm diminishes slightly. This is due to the fact that more and more time must be spent on pointer operations and loop overhead since the array is actually emulating a large array. Memory constraints limited the range of the number of processors that could be used to implement the SVD, but Figure 11 shows the results of the system performing 48 x 48 SVDs on differing numbers of processors. We note that the number of processors must divide the dimensions of the matrix or some processors will need to be idle. The figure indicates a linear speed-up as more processors are added for large sized problems. This behavior is expected due to the fact that the communication overhead does not grow as more processors are added. Actually, as more processors are added, the communication delay remains constant while the pointer overhead and housekeeping diminishes.

Table 2. Pipelined FFT Performance Results for Real-Time Signal and Image Processing

N	Processors Required	Time Per Pipelined FFT	Maximum Data Rate	NxN FFT Execution Time	NxN FFT Sustained MFLOPs
256	18	94	5.45 MHz	51.5ms	122
512	19	186	5.51 MHz	197ms	143
1024	21	372	5.51 MHz	776ms	162
2048	22	746	5.50 MHz	3.09sec	179
4096	24	1468	5.43 MHz	12.43sec	194

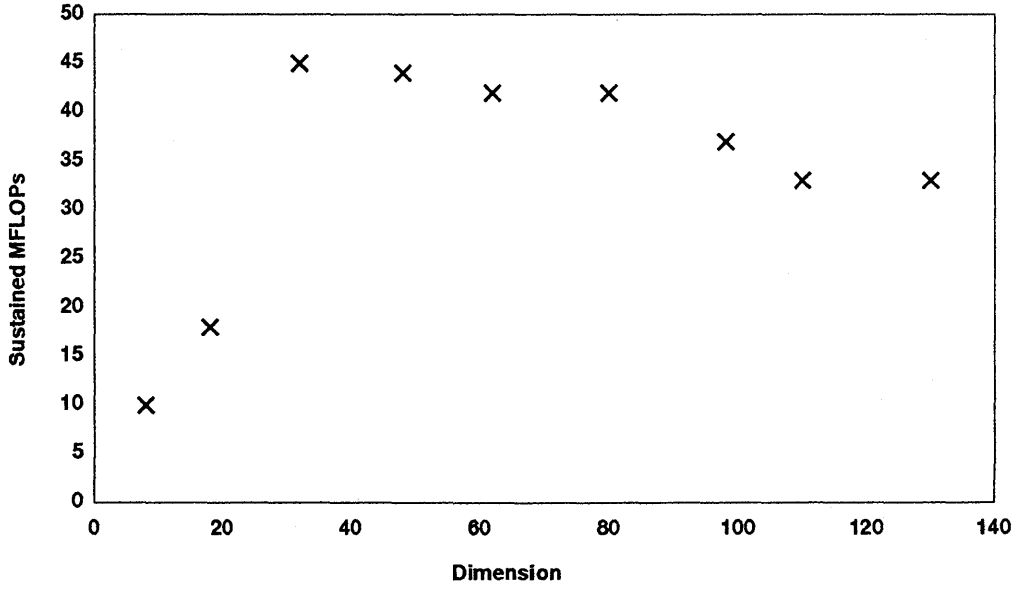


Figure 10. SVD Performance on 8-Processor Systems

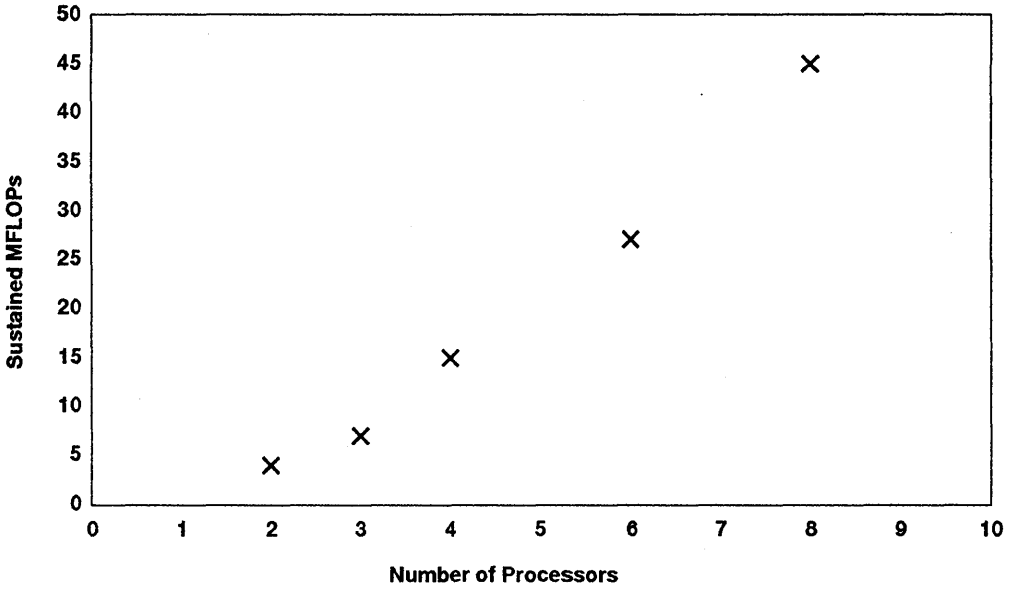


Figure 11. 48 x 48 SVD on P-Processor Systems

Conclusion

In this paper a hybrid architecture for matrix, DSP, image, and scientific computations has been presented to harness the power of N TMS34082 floating-point processors. The architecture can be programmed using many different programming models and parallel processing paradigms so that efficient programs can be written for a broad range of algorithms. The machine may be programmed as a shared memory machine, a message passing distributed memory machine, or a systolic array. The architecture may dynamically switch between any of these modes under software control.

The architecture is optimized to operate with multiple TMS34082s. To this end, a local bus controller is introduced to assist the TMS34082s in pointer manipulations and to provide a fast addressing capability on the LAD bus. The bus controller also provides the ability to perform multiple bus operations, such as a fetch and a send, in the same cycle. By allowing the bus controller to have its own instruction stream, a program controlled DMA mechanism makes it possible for the cell to send messages or pass systolic data streams while the processor was executes numerical loops. While a simple address latching scheme seems reasonable, use of the smart LAD bus controller leads to speed-ups of two to three orders of magnitude.

The system was implemented using the TMS34082 Toolkit along with the RPPT simulation package. Matrix Multiplication, FFT, QRD and SVD algorithms were coded in Concurrent C and executed on the architecture model to provide detailed cycles counts which were converted into MFLOPs ratings for each algorithm. The simulations showed what must be done to make the system execute code efficiently. The main findings were that the TMS34082 must be freed from pointer manipulations whenever possible, that registered variables should be utilized to reduce costly stack operations, and that the LOOPCT register together with the *cjmp.d* instruction should be used to control loops. Hand optimizations to the assembly code generated by the C compiler were needed off-load LAD pointer manipulations to the bus controller hardware. The simulation showed that high performance can be achieved if the system is carefully designed and code is optimized. Algorithms can often sustain computation rates approaching MFLOPs per processor, where the MFLOPs rating account for program overhead and data I/O time. For example, the simulation showed the matrix multiplication algorithm could run at just under 100 MFLOPs on a ten TMS34082 system.

Bibliography

1. M. Annaratone, E. Arnould, R. Cohn, T. Gross, and H. T. Kung. Harp architecture: From prototype to production. In *Proceedings of the 1987 National Computer Conference,* Chicago, Illinois, June 1987.
2. R. P. Brent and F. T. Luk. The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays. *SIAM J. Sci. Stat. Compt.,* 6(1):69–84, January 1985.
3. E. M. Dowling, M. Griffin, and F. J. Taylor. A multi-purpose VLSI floating-point array processor. *22nd Annual Asilomar Conference on Signals, Systems, and Computers,* October, 1988.
4. B. L. Drake, F. T. Luk, J. M. Speiser, and J. J. Symanski. Slapp: A systolic linear algebra parallel processor. *Computer,* 20(7):35-43, July, 1987.
5. D. W. Foulser and R. Schreiber. The saxpy matrix-1: A general purpose systolic computer. *Computer,* 2–(7):35-43, July, 1987.
6. M. R., Hestenes. Inversion of matrices by biorthogonalization and related results. *SIAM J. App. Math,* 6(1):51-90, March, 1958.
7. H. T. Kung. Why systolic architectures? *IEEE Computer Magazine,* pages 37-46, January, 1982.
8. F. T. Luk. On parallel Jaccobi orderings. *SIAM J. Sci. Stat. Compt.,* 10(1), January, 1989.
9. J. C. Nash. A one sided transformation method of the singular value decomposition and algebraic eigenproblem. *The Computer Journal,* 18:74-76, 1975.
10. R. A. Whiteside, N. S. Ostlund, and P. G. Hibbard. a parallel diagonalization algorithm for a loop multiple processor system. *IEEE Trans. on Computers,* C-33(5):409-413, May 1984.
11. G. H. Golub and C. F. Van Loan. Matrix Computations, 2nd Ed., John Hopkins University Press, Baltimore, MD, 1989.

TI Worldwide Sales Offices

ALABAMA: Huntsville: 4960 Corporate Drive, Suite N-150, Huntsville, AL 35805-6202, (205) 837-7530.

ARIZONA: Phoenix: 8825 N. 23rd Avenue, Suite 100, Phoenix, AZ 85021, (602) 995-1007.

CALIFORNIA: Irvine: 1920 Main Street, Suite 900, Irvine, CA 92714, (714) 660-1200; **Roseville:** 1 Sierra Gate Plaza, Suite 255B, Roseville, CA 95678, (916) 786-9208; **San Diego:** 5625 Ruffin Road, Suite 100, San Diego, CA 92123, (619) 278-9601; **Santa Clara:** 5353 Betsy Ross Drive, Santa Clara, CA 95054, (408) 980-9000; **Woodland Hills:** 21550 Oxnard Street, Suite 700, Woodland Hills, CA 91367, (818) 704-8100.

COLORADO: Aurora: 1400 S. Potomac Street, Suite 101, Aurora, CO 80012, (303) 368-8000.

CONNECTICUT: Wallingford: 9 Barnes Industrial Park So., Wallingford, CT 06492, (203) 269-0074.

FLORIDA: Altamonte Springs: 370 S. North Lake Boulevard, Suite 1008, Altamonte Springs, FL 32701, (407) 260-2116; **Fort Lauderdale:** 2950 N.W. 62nd Street, Suite 100, Fort Lauderdale, FL 33309, (305) 973-8502; **Tampa:** 4803 George Road, Suite 390, Tampa, FL 33634-6234, (813) 882-0017.

GEORGIA: Norcross: 5515 Spalding Drive, Norcross, GA 30092, (404) 662-7900.

ILLINOIS: Arlington Heights: 515 W. Algonquin, Arlington Heights, IL 60005, (708) 640-3000.

INDIANA: Carmel: 550 Congressional Drive, Suite 100, Carmel, IN 46032, (317) 573-6400; **Fort Wayne:** 118 E. Ludwig Road, Suite 102, Fort Wayne, IN 46825, (219) 482-3311.

IOWA: Cedar Rapids: 373 Collins Road N.E., Suite 201, Cedar Rapids, IA 52402, (319) 395-9551.

KANSAS: Overland Park: 7300 College Boulevard, Lighton Plaza, Suite 150, Overland Park, KS 66210, (913) 451-4511.

MARYLAND: Columbia: 8815 Centre Park Drive, Suite 100, Columbia, MD 21045, (301) 964-2003.

MASSACHUSETTS: Waltham: 950 Winter Street, Suite 2800, Waltham, MA 02154, (617) 895-9100.

MICHIGAN: Farmington Hills: 33737 W. 12 Mile Road, Farmington Hills, MI 48331, (313) 553-1500; **Grand Rapids:** 3075 Orchard Vista Drive S.E., Grand Rapids, MI 49506, (616) 957-4202.

MINNESOTA: Eden Prairie: 11000 W. 78th Street, Suite 100, Eden Prairie, MN 55344, (612) 828-9300.

MISSOURI: St. Louis: 12412 Powerscourt Drive, Suite 125, St. Louis, MO 63131, (314) 821-8400.

NEW JERSEY: Iselin: Parkway Towers, 485E. Route 1 South, Iselin, NJ 08830, (201) 750-1050.

NEW MEXICO: Albuquerque: 1224 Parsons Court, N.E., Albuquerque, NM 87112, (505) 291-0495.

NEW YORK: East Syracuse: 6365 Collamer Drive, East Syracuse, NY 13057, (315) 463-9291; **Fishkill:** 300 Westage Business Center, Suite 140, Fishkill, NY 12524, (914) 897-2900; **Melville:** 1895 Walt Whitman Road, P.O. Box 2936, Melville, NY 11747, (516) 454-6600; **Pittsford:** 2851 Clover Street, Pittsford, NY 14534, (716) 385-6770.

NORTH CAROLINA: Charlotte: 8 Woodlawn Green, Suite 100, Charlotte, NC 28217, (704) 527-0930; **Raleigh:** 2809 Highwoods Boulevard, Suite 100, Raleigh, NC 27625, (919) 876-2725.

OHIO: Beachwood: 23775 Commerce Park Road, Beachwood, OH 44122, (216) 464-6100; **Beavercreek:** 4200 Colonel Glenn Highway, Suite 600, Beavercreek, OH 45431, (513) 427-6200.

OREGON: Beaverton: 6700 S.W. 105th Street, Suite 110, Beaverton, OR 97005, (503) 643-6758.

PENNSYLVANIA: Blue Bell: 670 Sentry Parkway, Blue Bell, PA 19422, (215) 825-9500.

PUERTO RICO: Hato Rey: 615 Merchantile Plaza Building, Suite 505, Hato Rey, PR 00918, (809) 753-8700.

TEXAS: Austin: 12501 Research Boulevard, Austin, TX 78759, (512) 250-7655; **Dallas:** 7839 Churchill Way, Dallas, TX 75251, (214) 917-1264; **Houston:** 9301 Southwest Freeway, Suite 360, Houston, TX 77074, (713) 778-6592.

UTAH: Salt Lake City: 1800 S. West Temple Street, Suite 201, Salt Lake City, UT 84115, (801) 466-8973.

WASHINGTON: Redmond: 5010 148th Avenue N.E., Building B, Suite 107, Redmond, WA 98052, (206) 881-3080.

WISCONSIN: Waukesha: 20825 Swenson Drive, Suite 900, Waukesha WI 53186, (414) 798-1001.

CANADA: Nepean: 301 Moodie Drive, Mallorn Center, Suite 102, Nepean, Ontario, Canada K2H 9C4, (613) 726-1970; **Richmond Hill:** 280 Centre Street East, Richmond Hill, Ontario, Canada L4C 1B1, (416) 884-9181; **St. Laurent:** 9460 Trans Canada Highway, St. Laurent, Quebec, Canada H4S 1R7, (514) 335-8392.

ARGENTINA: Texas Instruments Argentina Viamonte 1119, 1053 Capital Federal, Buenos Aires, Argentina, 1/748-3699.

AUSTRALIA (& NEW ZEALAND): Texas Instruments Australia Ltd., 6-10 Talavera Road, North Ryde (Sydney), New South Wales, Australia 2113, 2-878-9000; 5th Floor, 418 Street, Kilda Road, Melbourne, Victoria, Australia 3004, 3 267-4677; 171 Philip Highway, Elizabeth, South Australia 5112, 8 255-2066.

AUSTRIA: Texas Instruments GmbH., Hietzinger Kai 101-105, A-1130 Wien, (0222) 9100-0.

BELGIUM: S.A. Texas Instruments Belgium N.V., 11, Avenue Jules Bordetlaan 11, 1140 Brussels, Belgium, (02) 242 30 80.

BRAZIL: Texas Instruments Electronicos do Brasil Ltda., Rua Paes Leme, 524-7 Andar Pinheiros, 05424 Sao Paulo, Brazil, 0815-6166.

DENMARK: Texas Instruments A/S, Borupvang 2D, DK-2750 Ballerup, (44) 68 7400.

FINLAND: Texas Instruments OY, P.O. Box 86, 02321 Espoo, Finland, (0) 802 6517.

FRANCE: Texas Instruments France, 8-10 Avenue Morane Saulnier-B.P. 67, 78141 Velizy Villacoublay cedex, France, (1) 30 70 10 03.

GERMANY: Texas Instruments Deutschland GmbH., Haggertystrasse 1, 8050 Freising, (08161) 80-0 od. Nbst; Kurfürstendamm 195-196, 1000 Berlin 15, (030) 8 82 73 65; Düsseldorfer Strasse 40, 6236 Eschborn 1, (06196) 80 70; Kirchhorster Strasse 2, 3000 Hannover 51, (0511) 64 68-0; Maybachstrasse II, 7302 Ostfildern 2 (Nellingen), (0711) 34 03-0; Gildehofcenter, Hollestrasse 3, 4300 Essen 1, (0201) 24 25-0.

HOLLAND: Texas Instruments Holland B.V., Hogehilweg 19, Postbus 12995, 1100 AZ Amsterdam-Zuidoost, Holland, (020) 5602911.

HONG KONG: Texas Instruments Hong Kong Ltd., 8th Floor, World Shipping Center, 7 Canton Road, Kowloon, Hong Kong, 7351223.

HUNGARY: Texas Instruments International, Budaorsi u.42, H-1112 Budapest, Hungary, (1) 1 66 66 17.

IRELAND: Texas Instruments Ireland Ltd., 7/8 Harcourt Street, Dublin 2, Ireland, (01) 755233.

ITALY: Texas Instruments Italia S.p.A., Centro Direzionale Colleoni, Palazzo Perseo-Via Paracelso, 12, 20041, Agrate Brianza (Mi), (039) 63221; Via Castello della Magliana, 38, 00148 Roma, (06) 5222651; Via Amendola, 17, 40100 Bologna, (051) 554004.

JAPAN: Texas Instruments Japan Ltd., Aoyama Fuji Building 3-6-12 Kita-aoyama Minato-ku, Tokyo, Japan 107, 03-3498-2111; MS Shibaura Building 9F, 4-13-23 Shibaura, Minato-ku, Tokyo, Japan 108, 03-3769-8700; Nissho-iwai Building 5F, 2-5-8 Imabashi, Chuou-ku, Osaka, Japan 541, 06-204-1881; Dai-ni Toyota Building Nishi-kan 7F, 4-10-27 Meieki, Nakamura-ku, Nagoya, Japan 450, 052-583-8691; Kanazawa Oyama-cho Daiichi Seimei Building 6F, 3-10 Oyama-cho, Kanazawa, Ishikawa, Japan 920, 0762-23-5471; Matsumoto Showa Building 6F, 1-2-11 Fukashi, Matsumoto, Nagano, Japan 390, 0263-33-1060; Daiichi Olympic Tachikawa Building 6F, 1-25-12, Akebono-cho, Tachikawa, Tokyo, Japan 190, 0425-27-6760; Yokohama Nishiguchi KN Building 6F, 2-8-4 Kita-Saiwai, Nishi-Ku, Yokohama, Kanagawa, Japan 220, 045-322-6741; Nihon Seimei Kyoto Yasaka Building 5F, 843-2, Higashi Shiokohjicho, Higashi-iru, Nishinotoh-in, Shiokohji-dori, Shimogyo-ku, Kyoto, Japan 600, 075-341-7713; Sumitomo Seimei Kumagaya Building 8F, 2-44 Yayoi, Kumagaya, Saitama, Japan 360, 0485-22-2440; 2597-1, Aza Harudai, Oaza Yasaka, Kitsuki, Oita, Japan 873, 09786-3-3211.

KOREA: Texas Instruments Korea Ltd., 28th Floor, Trade Tower, 159-1, Samsung-Dong, Kangnam-ku Seoul, Korea, 2 551 2800.

MEXICO: Texas Instruments de Mexico S.A., Alfonso Reyes 115, Col. Hipodromo Condesa, Mexico, D.F., Mexico 06120, 5/525-3860.

MIDDLE EAST: Texas Instruments, No. 13, 1st Floor Mannai Building, Diplomatic Area, P.O. Box 26335, Manama Bahrain, Arabian Gulf, 973 274681.

NORWAY: Texas Instruments Norge A/S, PB 106, Refstad (Sinsenveien 53), 0513 Oslo 5, Norway, (02) 155090.

PEOPLE'S REPUBLIC OF CHINA: Texas Instruments China Inc., Beijing Representative Office, 7-05 CITIC Building, 19 Jianguomenwai Dajie, Beijing, China, 500-2255, Ext. 3750.

PHILIPPINES: Texas Instruments Asia Ltd., Philippines Branch, 14th Floor, Ba-Lepanto Building, Paseo de Roxas, Makati, Metro Manila, Philippines, 2 817 6031.

PORTUGAL: Texas Instruments Equipamento Electronico (Portugal) LDA., 2650 Moreira Da Maia, 4470 Maia, Portugal (2) 948 1003.

SINGAPORE (& INDIA, INDONESIA, MALAYSIA, THAILAND): Texas Instruments Singapore (PTE) Ltd., Asia Pacific Division, 101 Thomson Road, #23-01, United Square, Singapore 1130, 350 8100.

SPAIN: Texas Instruments España S.A., c/Gobelas 43, Ctra de La Coruna km. 14, La Florida, 28023 Madrid, Spain, (1) 372 8051; c/Diputacion, 279-3-5, 08007 Barcelona, Spain, (3) 317 91 80.

SWEDEN: Texas Instruments International Trade Corporation (Sverigefilialen), Box 30, S-164 93 Kista, Sweden, (08) 752 58 00.

SWITZERLAND: Texas Instruments Switzerland AG, Riedstrafse 6, CH-8953 Dietikon, Switzerland, (01) 74 42 811.

TAIWAN: Texas Instruments Supply Company, Taiwan Branch, Room 903, 9th Floor, Bank Tower, 205 Tung Hua N. Road, Taipei, Taiwan, Republic of China, 2 713 9311.

UNITED KINGDOM: Texas Instruments Ltd., Manton Lane, Bedford, England, MK41 7PA, (0234) 270 111.

TEXAS INSTRUMENTS

A0291

Printed in the U.S.A.

TI North American Sales Offices

ALABAMA: Huntsville: (205) 837-7530
ARIZONA: Phoenix: (602) 995-1007
CALIFORNIA: Irvine: (714) 660-1200
Roseville: (916) 786-9208
San Diego: (619) 278-9601
Santa Clara: (408) 980-9000
Woodland Hills: (818) 704-8100
COLORADO: Aurora: (303) 368-8000
CONNECTICUT: Wallingford: (203) 269-0074
FLORIDA: Altamonte Springs: (407) 260-2116
Fort Lauderdale: (305) 973-8502
Tampa: (813) 882-0017
GEORGIA: Norcross: (404) 662-7900
ILLINOIS: Arlington Heights: (708) 640-3000
INDIANA: Carmel: (317) 573-6400
Fort Wayne: (219) 482-3311
IOWA: Cedar Rapids: (319) 395-9551
KANSAS: Overland Park: (913) 451-4511
MARYLAND: Columbia: (301) 964-2003
MASSACHUSETTS: Waltham: (617) 895-9100
MICHIGAN: Farmington Hills: (313) 553-1500
Grand Rapids: (616) 957-4202
MINNESOTA: Eden Prairie: (612) 828-9300
MISSOURI: St. Louis: (314) 821-8400
NEW JERSEY: Iselin: (201) 750-1050
NEW MEXICO: Albuquerque: (505) 291-0495
NEW YORK: East Syracuse: (315) 463-9291
Fishkill: (914) 897-2900
Melville: (516) 454-6600
Pittsford: (716) 385-6770
NORTH CAROLINA: Charlotte: (704) 527-0930
Raleigh: (919) 876-2725
OHIO: Beachwood: (216) 464-6100
Beavercreek: (513) 427-6200
OREGON: Beaverton: (503) 643-6758
PENNSYLVANIA: Blue Bell: (215) 825-9500
PUERTO RICO: Hato Rey: (809) 753-8700
TEXAS: Austin: (512) 250-7655
Dallas: (214) 917-1264
Houston: (713) 778-6592
UTAH: Salt Lake City: (801) 466-8973
WASHINGTON: Redmond: (206) 881-3080
WISCONSIN: Waukesha: (414) 798-1001
CANADA: Nepean: (613) 726-1970
Richmond Hill: (416) 884-9181
St. Laurent: (514) 335-8392

TI Regional Technology Centers

CALIFORNIA: Irvine: (714) 660-8140
Santa Clara: (408) 748-2220
GEORGIA: Norcross: (404) 662-7950
ILLINOIS: Arlington Heights: (708) 640-2909
INDIANA: Indianapolis: (317) 573-6400
MASSACHUSETTS: Waltham: (617) 895-9196
MEXICO: Mexico City: 491-70834
MINNESOTA: Minneapolis: (612) 828-9300
TEXAS: Dallas: (214) 917-3881
CANADA: Nepean: (613) 726-1970

Customer Response Center

TOLL FREE: (800) 336-5236
OUTSIDE USA: (214) 995-6611
(8:00 a.m. - 5:00 p.m. CST)

TI Authorized North American Distributors

Alliance Electronics, Inc. (military product only)
Almac Electronics
Arrow/Kierulff Electronics Group
Arrow (Canada)
Future Electronics (Canada)
GRS Electronics Co., Inc.
Hall-Mark Electronics
Lex Electronics
Marshall Industries
Newark Electronics
Wyle Laboratories
Zeus Components
Rochester Electronics, Inc. (obsolete product only (508) 462-9332)

TI Distributors

ALABAMA: Arrow/Kierulff (205) 837-6955; Hall-Mark (205) 837-8700; Marshall (205) 881-9235; Lex (205) 895-0480.

ARIZONA: Arrow/Kierulff (602) 437-0750; Hall-Mark (602) 437-1200; Marshall (602) 496-0290; Lex (602) 431-0030; Wyle (602) 437-2088.

CALIFORNIA: Los Angeles/Orange County: Arrow/Kierulff (818) 701-7500, (714) 838-5422; Hall-Mark (818) 773-4500, (714) 727-6000; Marshall (818) 407-4100, (714) 458-5301; Lex (818) 880-9686, (714) 863-0200; Wyle (818) 880-9000, (714) 863-9953; Zeus (714) 921-9000, (818) 889-3838;

Sacramento: Hall-Mark (916) 624-9781; Marshall (916) 635-9700; Lex (916) 364-0230; Wyle (916) 638-5282;

San Diego: Arrow/Kierulff (619) 565-4800; Hall-Mark (619) 268-1201; Marshall (619) 578-9600; Lex (619) 495-0015; Wyle (619) 565-9171; Zeus (619) 277-9681;

San Francisco Bay Area: Arrow/Kierulff (408) 441-9700; Hall-Mark (408) 432-4000; Marshall (408) 942-4600; Lex (408) 432-7171; Wyle (408) 727-2500; Zeus (408) 629-4789.

COLORADO: Arrow/Kierulff (303) 373-5616; Hall-Mark (303) 790-1662; Marshall (303) 451-8383; Lex (303) 799-0258; Wyle (303) 457-9953.

CONNECTICUT: Arrow/Kierulff (203) 265-7741; Hall-Mark (203) 271-2844; Marshall (203) 265-3822; Lex (203) 264-4700.

FLORIDA: Fort Lauderdale: Arrow/Kierulff (305) 429-8200; Hall-Mark (305) 971-9280; Marshall (305) 977-4880; Lex (305) 977-7511;

Orlando: Arrow/Kierulff (407) 333-9300; Hall-Mark (407) 830-5855; Marshall (407) 767-8585; Lex (407) 331-7555; Zeus (407) 365-3000;

Tampa: Hall-Mark (813) 541-7440; Marshall (813) 573-1399; Lex (813) 541-5100.

GEORGIA: Arrow/Kierulff (404) 497-1300; Hall-Mark (404) 623-4400; Marshall (404) 923-5750; Lex (404) 449-9170.

ILLINOIS: Arrow/Kierulff (708) 250-0500; Hall-Mark (708) 860-3800; Marshall (708) 490-0155; Newark (312)784-5100; Lex (708) 330-2888.

INDIANA: Arrow/Kierulff (317) 299-2071; Hall-Mark (317) 872-8875; Marshall (317) 297-0483; Lex (317) 843-1050.

IOWA: Arrow/Kierulff (319) 395-7230; Lex (319) 373-1417.

KANSAS: Arrow/Kierulff (913) 541-9542; Hall-Mark (913) 888-4747; Marshall (913) 492-3121; Lex (913) 492-2922.

MARYLAND: Arrow/Kierulff (301) 995-6002; Hall-Mark (301) 988-9800; Marshall (301) 622-1118; Lex (301) 596-7800; Zeus (301) 997-1118.

MASSACHUSETTS: Arrow/Kierulff (508) 658-0900; Hall-Mark (508) 667-0902; Marshall (508) 658-0810; Lex (508) 694-9100; Wyle (617) 272-7300; Zeus (617) 863-8800.

MICHIGAN: Detroit: Arrow/Kierulff (313) 462-2290; Hall-Mark (313) 462-1205; Marshall (313) 525-5850; Newark (313) 967-0600; Lex (313) 525-8100;

Grand Rapids: Arrow/Kierulff (616) 243-0912.

MINNESOTA: Arrow/Kierulff (612) 830-1800; Hall-Mark (612) 941-2600; Marshall (612) 559-2211; Lex (612) 941-5280.

MISSOURI: Arrow/Kierulff (314) 567-6888; Hall-Mark (314) 291-5350; Marshall (314) 291-4650; Lex (314) 739-0526.

NEW HAMPSHIRE: Lex (800) 833-3557.

NEW JERSEY: Arrow/Kierulff (201) 538-0900, (609) 596-8000; GRS (609) 964-8560; Hall-Mark (201) 515-3000, (609) 235-1900; Marshall (201) 882-0320, (609) 234-9100; Lex (201) 227-7880, (609) 273-7900.

NEW MEXICO: Alliance (505) 292-3360.

NEW YORK: Long Island: Arrow/Kierulff (516) 231-1000; Hall-Mark (516) 737-0600; Marshall (516) 273-2424; Lex (516) 231-2500; Zeus (914) 937-7400;

Rochester: Arrow/Kierulff (716) 427-0300; Hall-Mark (716) 425-3300; Marshall (716) 235-7620; Lex (716) 383-8020;

Syracuse: Marshall (607) 798-1611.

NORTH CAROLINA: Arrow/Kierulff (919) 876-3132; (919) 725-8711; Hall-Mark (919) 872-0712; Marshall (919) 878-9882; Lex (919) 876-0000.

OHIO: Cleveland: Arrow/Kierulff (216) 248-3990; Hall-Mark (216) 349-4632; Marshall (216) 248-1788; Lex (216) 464-2970;

Columbus: Hall-Mark (614) 888-3313;

Dayton: Arrow/Kierulff (513) 435-5563; Marshall (513) 898-4480; Lex (513) 439-1800; Zeus (513) 293-6162.

OKLAHOMA: Arrow/Kierulff (918) 252-7537; Hall-Mark (918) 254-6110; Lex (918) 622-8000.

OREGON: Almac (503) 629-8090; Arrow/Kierulff (503) 627-7667; Marshall (503) 644-5050; Wyle (503) 643-7900.

PENNSYLVANIA: Arrow/Kierulff (215) 928-1800; GRS (215) 922-7037; Marshall (412) 788-0441; Lex (412) 963-6804.

TEXAS: Austin: Arrow/Kierulff (512) 835-4180; Hall-Mark (512) 258-8848; Lex (512) 339-0088; Wyle (512) 345-8853;

Dallas: Arrow/Kierulff (214) 380-6464; Hall-Mark (214) 553-4300; Marshall (214) 233-5200; Lex (214) 247-6300; Wyle (214) 235-9953; Zeus (214) 783-7010;

Houston: Arrow/Kierulff (713) 530-4700; Hall-Mark (713) 781-6100; Marshall (713) 895-9200; Lex (713) 784-3600; Wyle (713) 879-9953;

UTAH: Arrow/Kierulff (801) 973-6913; Marshall (801) 485-1551; Wyle (801) 974-9953.

WASHINGTON: Almac (206) 643-9992, (509) 924-9500; Arrow/Kierulff (206) 643-4800; Marshall (206) 486-5747; Wyle (206) 881-1150.

WISCONSIN: Arrow/Kierulff (414) 792-0150; Hall-Mark (414) 797-7844; Marshall (414) 797-8400; Lex (414) 784-9451.

CANADA: Calgary: Future (403) 235-5325;

Edmonton: Future (403) 438-2858;

Montreal: Arrow Canada (514) 735-5511; Future (514) 694-7710; Marshall (514) 694-8142;

Ottawa: Arrow Canada (613) 226-6903; Future (613) 820-8313; **Quebec City:** Arrow Canada (418) 871-7500;

Toronto: Arrow Canada (416) 670-7769; Future (416) 612-9200; Marshall (416)458-8046;

Vancouver: Arrow Canada (604) 421-2333; Future (604) 294-1166.

TEXAS INSTRUMENTS

D0291

Printed in the U.S.A.

TEXAS INSTRUMENTS

Printed in U.S.A., May 1991
2564007-9761 revision A

SCGU004