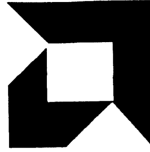


Advanced Micro Devices

Am29300 DEMONSTRATION SYSTEM

Application Note

By Mark McClain

©1987 Advanced Micro Devices

Advanced Micro Devices reserves the right to make changes in its products without notice in order to improve design or performance characteristics.

This appication note neither states nor implies any warranty of any kind, including but not limited to implied warranties of merchantability or fitness for a particular application. AMD assumes no responsibility for the use of any circuitry other than the circuitry embodied in an AMD product.
The information in this publication is believed to be accurate in all respects at the time of publication, but is subject to change without notice. AMD assumes no responsibility for any errors or omissions, and disclaims responsibility for any consequences resulting from the use of the information included herein. Additionally, AMD assumes no responsibility for the functioning of undescribed features or parameters.

901 Thompson Place, P.O. Box 3453, Sunnyvale, California 94088
(408) 732-2400 TWX: 910-339-9280 TELEX: 34-6306

Abstract

This application note describes the design of a high performance microprogrammed 32-bit processor using the Am29300 family of 32-bit building blocks. Basic design philosophy for a microprogrammed processor is discussed as the design choices made for this system are explained. Support circuitry used with the Am29300 family components is also covered in detail. This circuitry includes: Writable Control Store, Serial Shadow Register diagnostics, and Programable Array Logic.

Table of Contents

SECTION 1	**Overview**	1-1
	SYSTEM LAYOUT	1-2
	DATA FLOW	1-2
	Memory and I/O Sections	1-2
	Data Section	1-2
	Control Section	1-3
SECTION 2	**Nomenclature**	2-1
SECTION 3	**Data Section Description**	3-1
	REGISTER FILE	3-1
	ARITHMETIC LOGIC UNIT	3-1
	Am29332	3-1
	Macro Status Register	3-1
	FLOATING POINT PROCESSOR	3-4
	Am29325	3-4
	FPP External Status Register	3-6
	Seed Look-Up Table	3-6
	PARALLEL MULTIPLIER	3-8
SECTION 4	**Memory and External System Interface**	4-1
	EXTERNAL BUS INTERFACE CONTROL	4-2
	Host Access Definition	4-2
	Host Interface Block Diagram	4-3
	Event Signals	4-4
	Memory Enable	4-5
	AmPAL22V10 Support Logic	4-5
	SSR Diagnostics	4-5
	Controller Description	4-7
	MEMORY	4-10
	Memory Components	4-10
	Addressing Scheme	4-10
	CPU - Memory Buffers	4-12
	External System Buffers	4-13
SECTION 5	**Control Section Description**	5-1
	MACRO OPCODE SUPPORT	5-1
	Macro Opcode Register	5-1
	Macro Opcode Format Restrictions	5-2
	Macro Opcode Decoding Method	5-3
	Macro Opcode Map RAM	5-3
	WCS Port	5-4
	Macro Operand Address Counters	5-4
	REGISTER FILE ADDRESS MULTIPLEXER	5-6
	Read Ports A and B	5-6
	Write Port A	5-7
	Write Port B	5-8

POSITION AND WIDTH MULTIPLEXERS 5-8
SEQUENCER 5-9
D BUS TRANSCEIVER 5-12
INTERRUPT CONTROL 5-12
Interrupt and Trap Philosophy 5-12
Interrupt Operation 5-13
Trap Operation 5-15
MICROCODE CONTROL STORE AND CONTROL PIPELINE REGISTER 5-16
Control Store Function 5-16
Pipeline Register Function 5-16
Control Store Implementation 5-16
CLOCK CONTROL 5-18
Clock Qualification Circuit 5-18
Clock Generator 5-20
MICROCODE WORD 5-22
Control Philosophy 5-22
Microcode Word Field Descriptions 5-22
Alternate Arrangements 5-28
CONTROL DECODE 5-30
What Is It Good For? 5-30
Control Logic Description 5-30

SECTION 6 System Timing and Critical Path Analysis 6-1
DEFINITIONS 6-1
CONTROL AND DATA PATHS 6-1
WORST CASE PATHS 6-2
Case Definitions 6-2
FINAL RESULTS 6-4

SECTION 7 Physical Issues 7-1
ELECTRICAL LAYOUT ISSUES FOR POWER SUPPLY 7-1
DECOUPLING CAPACITORS 7-1
SOCKETS 7-1

SECTION 8 Conclusion 8-1

APPENDIX A Related Reference Material A-1
B Signal to Figure Cross Reference B-1
C FPP Status PAL Definition C-1
D Host Interface Glue PAL Definition D-1
E Host Interface Am29PL141 Definition E-1
F Memory Address Counter PAL Definition F-1
G Macro Operand Counter PAL Definition G-1
H Write Port A Multiplexer PAL Definition H-1
I Write Port B Multiplexer PAL Definition I-1
J Trap Logic PAL Definition J-1
K Clock Qualification PAL Definition K-1
L Clock Generator PAL Definition L-1
M Control Decode PALs Definitions M-1
N Components List N-1
O Goals O-1
Disclaimer O-1

SECTION 1
Overview

This application note describes the design of a high performance microprogrammed 32-bit processor using the Am29300 family of 32-bit building blocks.

Basic design philosophy for a microprogrammed processor is discussed as the design choices made for this system are explained. Issues of microprogram sequence control, interrupt handling, microprogram memory options, microword layout, macroprogramming, high speed multiply, and clock control are covered.

Support circuitry used with the Am29300 family components is also covered in detail. This circuitry includes: Writable Control Store, Serial Shadow Register diagnostics, and Programable Array Logic.

The use of the following Advanced Micro Devices components is illustrated in extensively documented examples:

Am29331	- 16-bit Address Sequencer,
Am29332	- 32-bit Arithmetic Logic Unit,
Am29334	- 64 x 18-bit Four Port Register File,
Am29C323	- 32-bit Parallel (Integer) Multiplier Accumulator,
Am29325	- 32-bit Floating Point Unit,
Am29114	- Interrupt Controller,
Am29800	- Family of Interface and Diagnostics Logic Devices,
Am29PL141	- Fuse Programmable State Machine,
AmPAL18P8	- Programmable Output 20-pin Combinatorial PAL,
AmPAL22V10	- Output Macrocell 24-pin PAL,
Am9151	- Registered RAM with SSR™,
Am99C165	-16K x 4-bit CMOS high speed RAM.

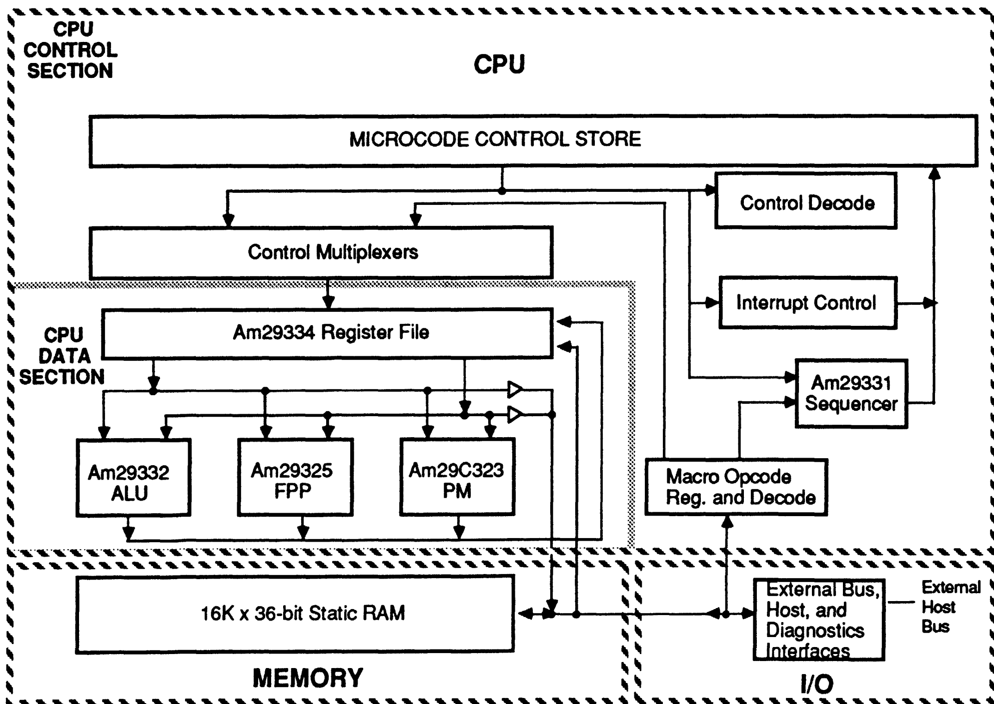


Figure 1-1. System Components

09856A 1-1

SSR is a trademark of Advanced Micro Devices, Inc.

SYSTEM LAYOUT

As with all processors, this system contains three main portions: Central Processing Unit (CPU), memory, and input/output (I/O) (see Figure 1-1).

The CPU consists of a control section and a data section:

The data section manipulates data via operations such as addition, subtraction, shifting, merging, multiplication, and division. These functions are implemented with the Am29332 Arithmetic Logic Unit (ALU), Am29325 Floating Point Processor (FPP), and Am29C323 Parallel Multiplier (PM). The data section also stores operands and intermediate results in Am29334 register files.

The control section directs the operations performed by the data section and determines the order in which the operations are performed. This section contains the Am29331 Microprogram Sequencer, macro opcode register & decode, interrupt control logic, microcode control store, control decoding logic, and control multiplexers for the register file and ALU.

The memory contains a 16K word by 36-bit static RAM. Included as part of the memory block are two address registers/counters, which may be used to speed up sequential reads and writes made by the CPU.

The I/O portion is a simple connection to a host system's address and data bus. It is assumed that the Am29300 demonstration system operates as a peripheral processor to a larger host system, as might be the case with an array or digital signal co-processor. Information to be processed by the demonstration system is loaded into the memory portion via Direct Memory Access (DMA). When processing of the data is complete, the host system unloads the memory portion via DMA.

A diagnostics port is also provided as part of the I/O section. This port allows control over the demonstration system clock for single stepping, and it allows for serial diagnostics to display and control the state of the system.

Throughout the remainder of this application note, it is assumed that the reader has some previous experience with microprogrammed processor design and is familiar with the Am29300 family data sheets. For those readers not familiar with microprogrammed design, some reference material is listed in Appendix A.

DATA FLOW

The system data paths are illustrated in the block diagram of Figure 1-2.

Memory and I/O Sections

Information processed by the Am29300 system is exchanged between the host system and the memory via the external bus interface. The information may be both data and macroinstructions.

From the external bus, the host system is able to address the memory via the bus driver connected to the memory address bus. Data is moved over the memory data bus. The host system's only access to the Am29300 system is via these buses to the memory. Therefore, all data to the system flows through the memory via DMA accesses by the host system.

Diagnostic control and information flows through the external bus interface via the host interface controller. It controls the clocking and single stepping of the system while loading and reading serial diagnostics via Serial Shadow Registers (SSR) that are placed in key locations throughout the system.

(SSR is a trademark of Advanced Micro Devices, Inc.)

Data Section

Data must be moved from the memory to the register file to be available to the ALU and multipliers for processing.

The register file has four access ports, two ports for writing data into the file and two ports for reading data out to the ALU and multipliers. This arrangement allows two operands to be read from the file in the same cycle as two operands are being written. The two read operands are used either as A and B operands for the ALU, FPP, or PM, or as address and data inputs to the memory.

To move data from the memory to the register file, an address to the memory is selected from the register file on the A read port. This address selects a word from the memory that is transferred on the memory data bus to the B write port of the register file.

Once data is loaded into the register file, it can then be selected for use on either the A or B read ports for input to the ALU, FPP, or PM.

Data processing results from the ALU, FPP, or PM are then placed on the Y bus for return to the register file A write port.

Finally, processed data is moved back to the memory via the B read port of the register file, while the location to be written in the memory is addressed by the value on the A read port of the register file.

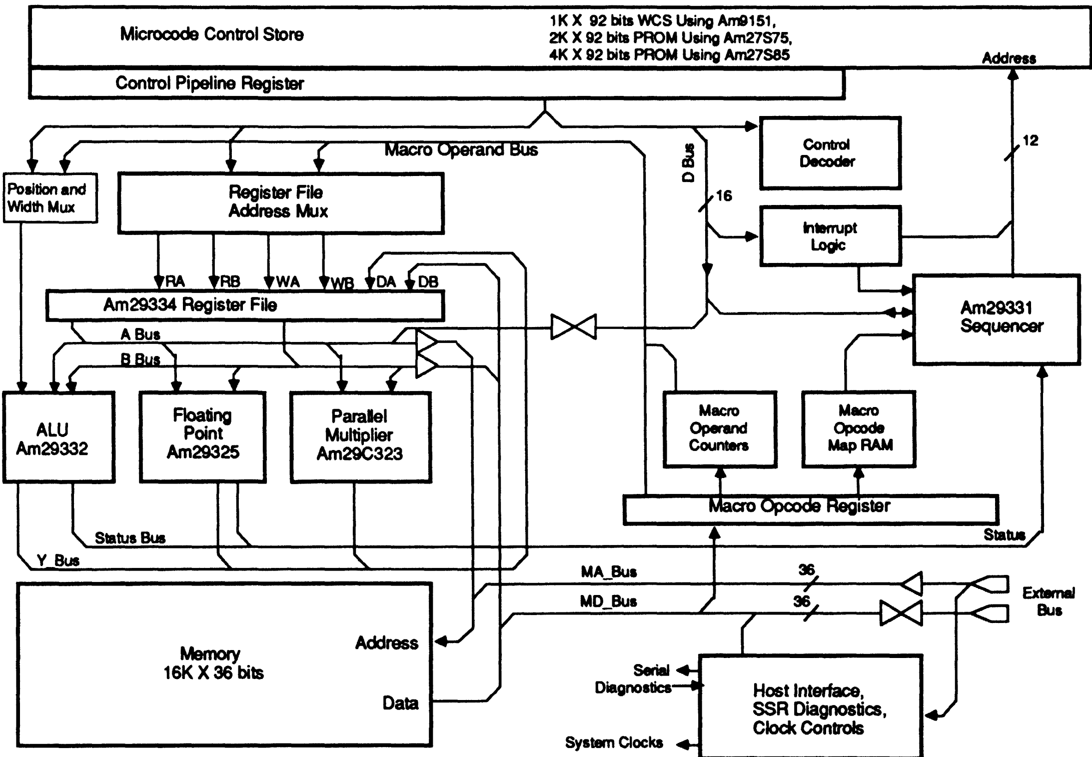


Figure 1-2. Am29300 Demonstration System

(NOTE: The advantage of using both write ports on the register file is that it is possible to perform calculations and write the results via the A write port at the same time that new data is being moved into the register file from the memory via the B write port. This will be illustrated in more detail later in this document.)

Control Section

D Bus

The D bus is a highway for information flow between the microcode control store, interrupt control sequencer, and data section of the CPU.

Branch addresses or constants from the microcode can pass to the sequencer via the D bus. The interrupt controller's interrupt vector base address register may also be loaded via the D bus.

Constants from the microcode can pass to the data section for use in calculations via the D bus to A bus transceiver. Microcode constants can also be used as addresses to the memory, via a D bus to A bus to memory address bus connection.

Variable data can be passed from the register file to the sequencer. The sequencer can also return data to the register file, via the A bus to ALU Y bus to A write port path. The D bus path to the sequencer is valuable for storing and retrieving the state information in the sequencer when interrupts, traps, or context switches occur.

Control Decode

This section of logic expands encoded microcode fields into individual control lines used throughout the system.

Interrupt Logic

This circuit monitors interrupt and trap conditions such as parity errors and breakpoints. When an interrupt condition is detected, an interrupt request to the sequencer is made and an interrupt address vector generated.

Sequencer

The sequencer is an address multiplexer with an on-chip address incrementer and stack. It selects the address for each microinstruction word read from the control store. The address selected depends on the instruction to the sequencer and on the state of test conditions. The sequencer can select addresses from the branch field of the control pipeline register, the macro opcode map, the internal stack, the increment of the last microinstruction address, or one of four status condition driven multi-way branch inputs.

Macro Opcode Support

Macro vs. Micro Programs: A microprogram is the definition for the state of the primary system control signals during each system clock cycle. Each word of microcode usually has a large number of bits so that many parallel operations may be controlled simultaneously. Each microcode word must deal with the intricate details of system operation. The writing of microcode is a slow tedious process that must take into account every facet of system operation in order to provide the most efficient use of system resources.

The advantage of microcode is that, very often, different system operations can be overlapped (done in parallel) since there is parallel control over all the system resources.

A "macroprogram" is a series of microcode subroutine calls. Each macroinstruction has an opcode field that is simply a value that can be translated into the starting address of a microcode subroutine within the system microprogram. The macroinstruction may include parameters that are passed to the microprogram. These parameters might be register addresses, loop counter values, immediate data, or memory addresses.

The advantage of a macroprogram is that the instructions are very simple and require relatively few bits to define as compared to a microcode word. The macroinstructions are simpler because all the details of system operation are specified by the underlying microcode instructions. The simpler instructions allow macroprograms to be written much more quickly than microprograms. Therefore, once a set of microcode subroutines are developed to perform the most often needed system operations, a wide variety of macroprogram applications can be quickly written. Macroinstructions remove the system programmer's concern over every detail of system operation.

The disadvantage of a macroprogram is that each instruction must be fetched from memory and decoded (translated to a microcode subroutine address) before each microcode subroutine is executed. When each subroutine execution is long compared to the overhead of fetching and decoding the macroinstruction, the macroprogram will run nearly as fast as an equivalent microprogram with the advantage being a much easier programming task. When the microcode subroutines are short compared to the macroinstruction overhead, the system speed can drop significantly.

So, if macroprogramming concepts are used carefully, a macroprogrammed approach to system design can yield a significant improvement in the ease of system use without a large decline in system performance.

For that reason, the Am29300 demonstration system includes the features described below, which allow a macroprogrammed approach. These features are intended to show how basic macroprogramming can be implemented.

Macro Opcode Register: When macro-instructions are executed, the instructions are addressed in the memory via the A read port of the register file in the same way as described earlier for data. The selected instruction is read from the memory via the memory data bus and written into the macro opcode register. The instruction can also be written into the register file via the B write port in the same cycle (which may be useful for instructions that contain immediate operands that would be used by the data section).

Macro Opcode Map RAM: The macro opcode map RAM is made of three Am9150 high speed SRAMs. The opcode portion of the macro opcode register addresses a microcode entry point table in the map RAM. This entry point is then used by the Am29331 sequencer as a branch address to the microcode routine that performs the function required by the macroinstruction.

Macro Operands: The operand portion of the macro opcode register is loaded into the macro operand counters. The macroinstruction operands allow the direct specification of register file addresses, ALU shift values, or ALU field masks to be used by the microcode routines.

Register File Address, Position, and Width Multiplexers: Register file addresses are passed to the register file via the register file address multiplexer. Position and width information for shift values and field masks are passed to the ALU via the position and width multiplexers. These multiplexers allow either the microcode or the macroinstructions to control the register file and ALU.

SECTION 2
Nomenclature

Throughout the remaining figures in this application note, some naming and drawing conventions are used as noted below.

All signal names are written as single word identifiers with underlines used to provide visual space between sections of a multi-word identifier.

Signals that are active low have names that end with an asterisk. In some of this document's programmable logic definition files, this convention is not allowed. In those situations, the active low signal names will begin with an exclamation point or end with an underline character.

Clock and qualified clock signals have names that begin with CLK_.

Groups of signals that form buses are shown as single lines with an associated number that indicates how many lines are involved. Bus lines are drawn with 45 degree turns and intersections instead of the usual right angle turns and intersections used with individual signal lines, in order to highlight buses visually. Major data highways such as the A_BUS, B_BUS, and Y_BUS have signal names that end in _BUS. The lines of a bus are numbered from least significant to most significant with the least significant identified as line zero (0). Where a subset of the lines in a bus is shown, the bus signal name will be followed by parentheses containing numbers that show the range of lines in use. The numbers of a continuous range are separated by a colon (:), non-contiguously numbered lines are separated by a comma (,). Where lines of a bus are split out to show the specific connection of bus lines in a circuit, a small number that indicates the line number within the bus will be shown near each line that is split off.

Four major buses in the system share a common structure. The A_BUS, B_BUS, Y_BUS, and MD_BUS all have the same layout. Each bus carries a 36-bit data word, which is arranged as four 8-bit bytes, each byte having its own parity bit. Byte zero (least significant) is located in bits 0:7; bit 32 is the parity bit for byte zero. Byte one is in bits 8:15 with its parity in bit 33. Byte two is in bits 16:23 with parity in bit 34. Byte three is in bits 24:31 with parity in bit 35.

Signals that come directly from the microcode memory pipeline register have signal names that begin with "P_".

Ground symbols (zero volt points) are drawn as downward pointing triangles, or the signal name GND is used.

Points tied to +5 volts are labeled with the signal name V_{CC}.

Components are shown with pin numbers immediately outside the rectangle that defines the component. Component-specific signal names related to component pins may be shown immediately inside the component rectangle. Where there are several components shown on a page with very similar connections, only one of the components will have pin numbers and signal names shown. The remaining components on the page are wired in the same manner.

Each component is assigned and labeled with a "U number" that uniquely identifies the component. This helps identify specific components for discussion and separates identical type devices in the system component list.

Because this demonstration system is complex by nature, it must be illustrated with many figures, each focusing on a different portion of the overall system. In order to show the signal interconnections between all parts of the system, each signal that leaves or enters a figure is given a name. Often the names are abbreviations in order to save space in the figures. Each name shows a relationship to the signal's use. Wherever the same signal name appears in different figures, a connection between the figures is defined. To help in identifying all the figures to which a signal travels, there is a signal-to-figure cross reference listing in Appendix B.

SECTION 3
Data Section Description

REGISTER FILE

Two Am29334 register files are used in tandem to provide a 64-register by 36-bit wide file. This allows the storage of 32-bit data plus parity (1 parity bit/byte). Each Am29334 contains 64 registers that are 18 bits wide; see Figure 3-1.

An Am29334 register file can both read and write data in the same cycle, but it does not perform the read and write simultaneously. The read must be performed during part of the system cycle and the write during another part of the cycle. Since read data is needed by the ALU and multipliers as early in the cycle as possible and, since data values to be written are only available later in the cycle, the reading of data is done in the first half of the cycle and the writing done in the second half of the cycle. A convenient way to separate the two parts of the cycle is to use the system clock signal to control the internal address mux and write enable.

As connected in Figure 3-1, the read port latch enables (LEA and LEB) and write port common enables (WEAC* and WEBC*) are tied to the data section clock line (CLK_D). This causes read data to be accessed while CLK_D is high and read data to be latched when CLK_D is low. Data is written when CLK_D is low if the port write enables are active (WEAL* and WEAH*, or WEBL* and WEBH*). The high and low byte write enables for each port are tied together since only full 36-bit word writes will be done in this system.

The various read and write addresses are provided from the register file address multiplexers, which will be covered later.

The output enable (P_OEA*) and write enables (P_WEA* and P_WEB*) come directly from the microcode pipeline register.

ARITHMETIC LOGIC UNIT

Am29332

The Am29332 provides a 64-bit funnel (barrel) shifter, 32-bit mask generator, and 32-bit ALU. The ALU can perform binary and BCD add or subtract, multi-cycle multiply or divide, and logical operations. This single, highly-integrated chip provides the complete function of the ALU block in this system. The only added component is an external register used to maintain status bits for the macroprogram separate from status information used by the micro program. The ALU is shown in Figure 3-2.

Most of the control lines come directly from the microcode control pipeline register.

The ALU output enable (ALU_OE*) is decoded from the control pipeline register.

The POSITION and WIDTH signals come from the position and width multiplexers. These multiplexers select the position and width values from either the microcode pipeline or the macroinstruction in the macro opcode register.

The slave mode input is tied to ground since there will be no use of the slave mode comparisons in this system.

The HOLD input is used as an enable control over the clocking of the internal micro status register and Q register during times the ALU is not in use. Because the ALU, FPP, and PM share the same data source and destination buses (A_BUS, B_BUS, and Y_BUS), they generally cannot be used simultaneously due to bus contention. In recognition of this, the control fields for the ALU, FPP, and PM have been overlapped in the microcode to minimize the required width of each microcode word. This means that at certain times the control lines to the ALU will be meaningless to the ALU because the values on the lines are determined by the needs of the FPP or PM. Therefore, unless the hold input is used to prevent clocking of the status and Q register duing these times, the ALU status could be lost whenever the FPP or PM are in use.

Note, however, that the hold input is not used as the general means to prevent clocking of the ALU registers when the whole system is halted (e.g., during single step mode). The data clock (CLK_D) that is distributed throughout the data section of the CPU is a qualified clock and will be used to control the state change of all registers in the data section, including those in the ALU at times when the whole system is halted.

Macro Status Register

There are two levels of status information that the programmer of a microprogrammed system must track if that system executes macroinstructions. These are referred to as the micro and macro status. The micro status of the system is updated at the end of each microcode step and is part of the system state. The macro status is part of the macroprogram state as reflected at the end of each macro step. Since many microinstructions may be executed to perform the function defined by a given macroinstruction, the macro status reflects the machine state

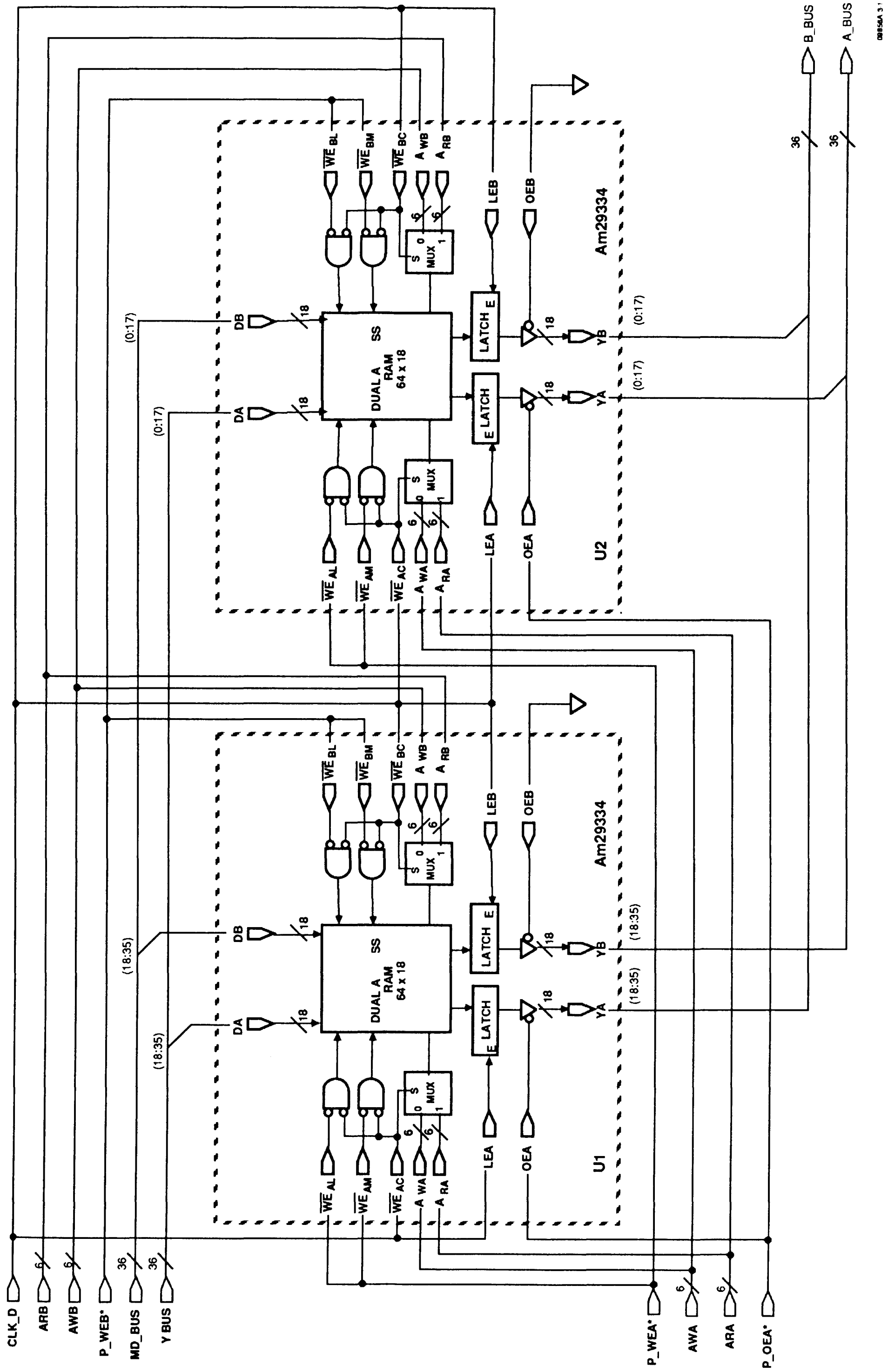


Figure 3-1 Register File

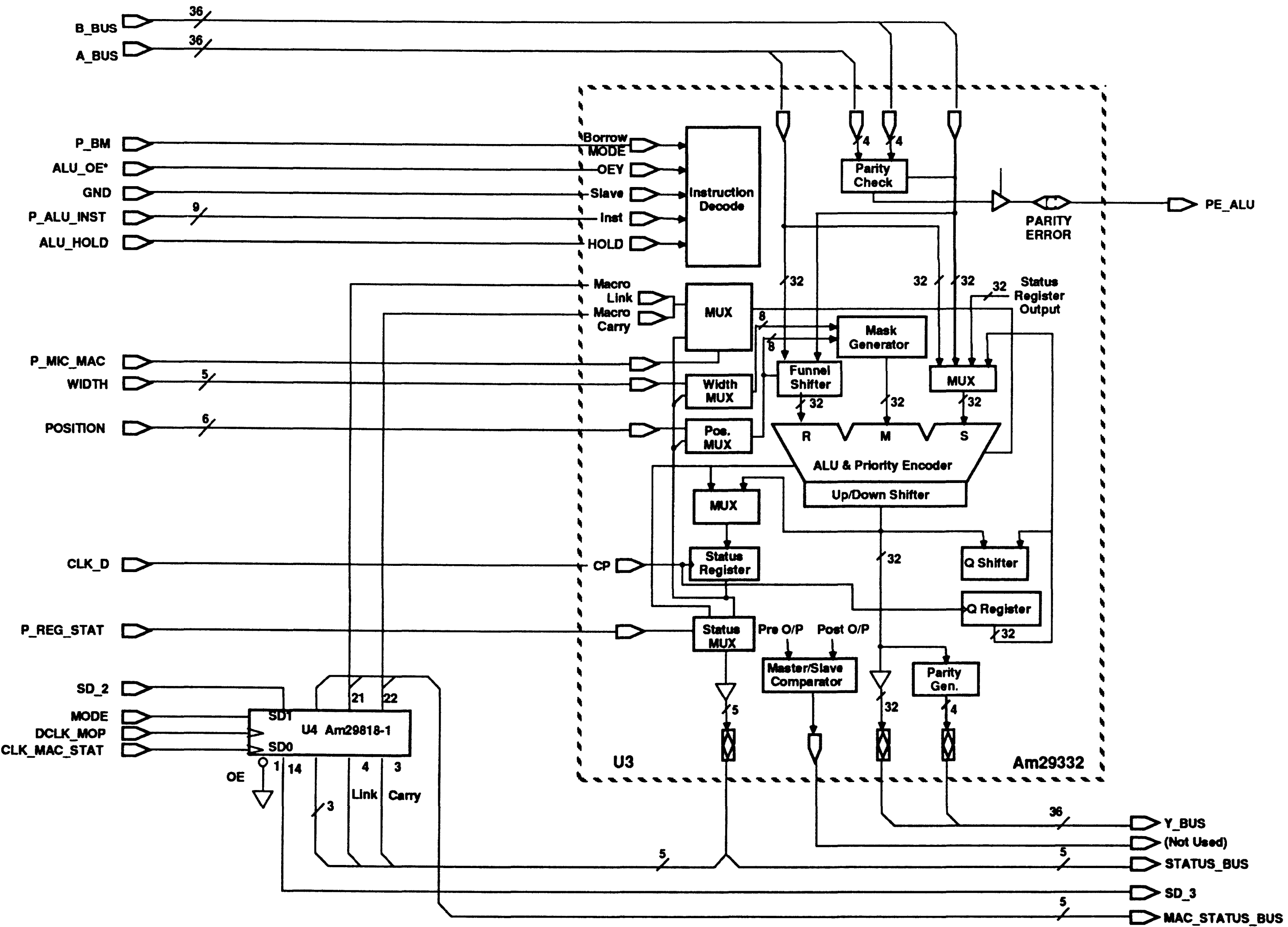


Figure 3-2 ALU Block

from the macroprogram viewpoint. The macro status may be carried across many microinstruction cycles without change. This requires a separate register to contain the macro status independent of the micro status. The Am29332 does not have an internal macro status register so one must be provided externally. The loading of the macro status register and the use of the macro status information by the microprogram must be controlled by microcode. The Am29332 does provide an onboard multiplexer to select between the micro and macro status inputs. Only the carry and link values are used directly by the Am29332 since these are the only status values normally used to modify data values. The macro stat us for the zero, sign, and overflow flags can be used by the sequencer as test conditions for branch instructions.

The register used for holding macro status is an Am29818-1. The register is loaded (clocked) by a qualified clock called CLK_MAC_STAT. This clock is qualified by the load macro status bit in the control pipeline register. The Am29818-1 is also used to provide a diagnostic ability to read and load the macro status register through the use of an internal serial shadow register (SSR).

FLOATING POINT PROCESSOR

Am29325

The Am29325 Floating Point Processor (FPP) performs 32-bit floating point multiplication, addition, or subtraction in a single cycle. Floating point division can be done in seven cycles using the Newton-Raphson method. The FPP is shown in Figure 3-3.

All the control lines for the FPP are driven directly by the microcode pipeline register with the exception of the FPP output enable and the register flow-through enables. Those signals are decoded from the data path select field of the microcode pipeline register. The output enable decode is done by the AmPAL22V10 in Figure 3-3. The register flow through enable decode is done by the control decode logic which is described later.

It should be noted that the Am29325 is not a full fledged member of the Am29300 family. It is different from the other Am29300 members with regard to three key characteristics: it is slower, does no data bus parity checking or generation, and has no slave mode capability.

The Am29325 flow through calculation time is 100 to 125 ns rather than the 42 or 70 ns for the ALU or PM (the current PM is at 120 ns, but the fastest version will be at 70 ns). This requires that whenever the FPP is used, the system clock cycle must be extended to allow for the slower propagation time. This extended clock timing is covered later in more detail.

The lack of parity checking is not much of a problem for the rest of the system since it only affects the data integrity of information going through the FPP. The lack of parity generation isn't a problem as long as only the FPP is working on the data. The problem starts when floating point data is moved back to memory or is converted to integer values for use by the ALU.

If data from the FPP is read by the ALU or PM, parity errors will be detected and a system interrupt may result. That problem can be avoided if the system has kept track of which data resulted from FPP calculations and if the parity errors are ignored when that data is read. But if FPP data results are moved directly to the memory and then on to the host system, the parity errors will eventually be found.

So some means of adding parity generation to the FPP should be provided. One way is to add four 8-bit parity generator chips to the FPP output bus. This consumes power and boardspace while providing a benefit only when FPP data is moved directly through the register file to the memory. A better way is to use the parity generators already available in the Am29332 by requiring that FPP data be passed through the ALU before being moved to the memory. Even though the data may not be modified by the ALU, correct parity will be generated on the ALU output.

With the use of a little trick, there is a way to provide parity checking on the FPP data inputs. To do this, one of the data path select codes is used to control the output enables of both the ALU and FPP. This code (P_DSP = 11) causes the FPP outputs to be disabled and the ALU outputs enabled, even though the data path selected is the FPP. By turning on the ALU outputs, the ALU parity error output will also be enabled and any parity error on the A_BUS or B_BUS will be reported. At the same time, the control microcode for the FPP is still valid and may be used to load registers with the data present on the A_BUS and B_BUS. Of course the register file should not be loaded from the Y_BUS in the cycle where this scheme is used because the ALU is driving nonsense information onto the Y_BUS. Enabling the ALU outputs is only a trick used to make the ALU parity checker results available for this scheme. Note that the ALU hold input remains active even though the ALU output enable is active. This prevents any state change in the ALU when the FPP is the data path actually in use.

Finally, the issue of no slave error checking is unimportant, since the slave mode is not used in this system.

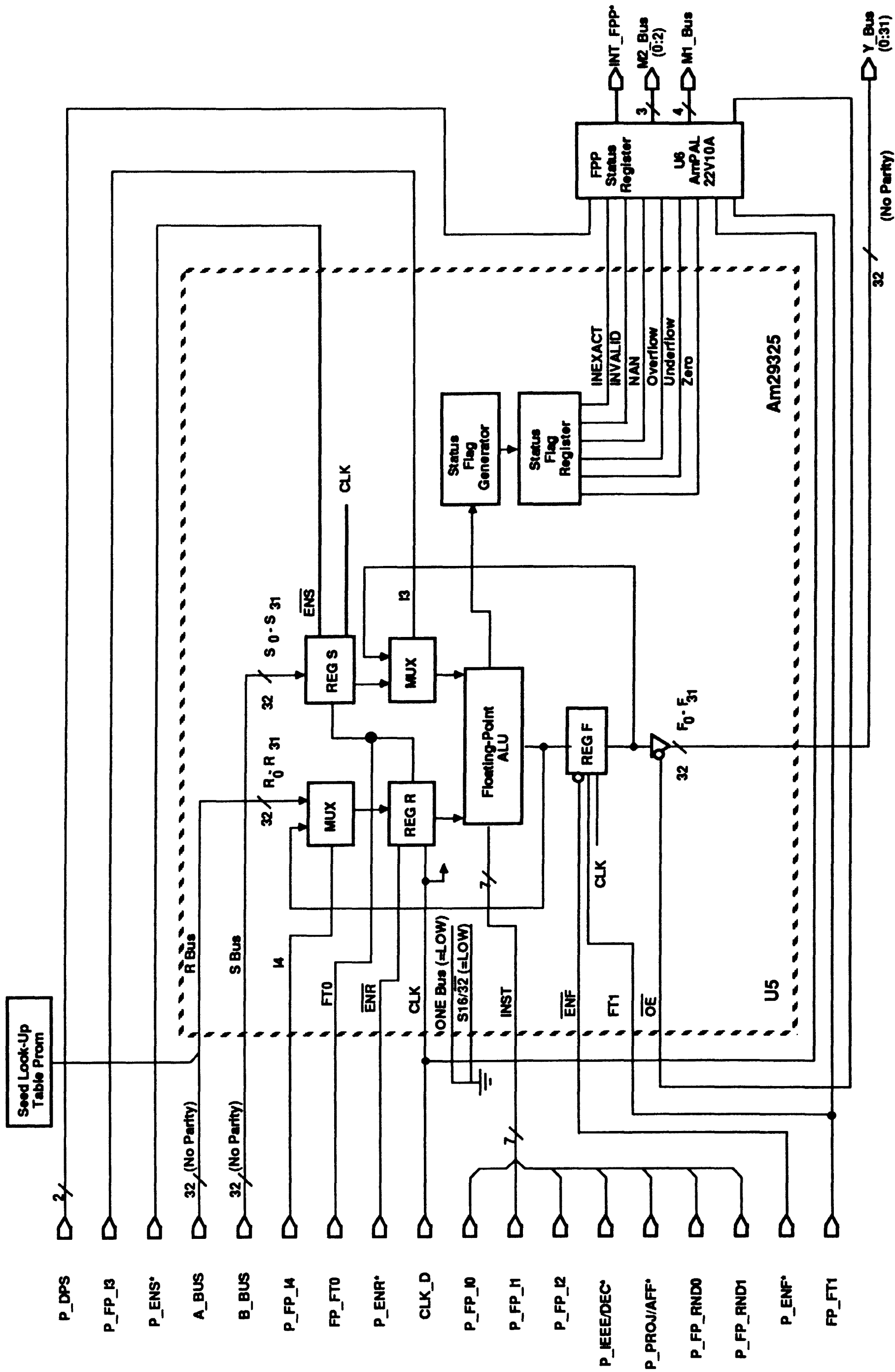


Figure 3-3 Floating Point Block

FPP External Status Register

Status Pipeline Issue

The FPP status flags appear at the status outputs along with data at the Y outputs. If the FPP "F" register is made transparent, the status flag register is also transparent. If the F register is clocked, so is the status register. In this demonstration system this presents a problem.

Normally, status conditions from the data section are registered before being used by the control section. This maintains the pipelined, parallel operation of the control and data sections. The control section bases its testing on registered status from the last data section cycle rather than being forced to wait for status results of the current execution cycle before determining the next microinstruction to execute.

To provide the same system for the FPP requires an external status register for cycles in which the F register is transparent to allow results to pass directly to the register file. In that situation the status flags are not registered by the FPP and thus, without an external register, there is no place to pipeline the status for the control section.

Multiple Status Flag Test Issue

Several of the FPP status flags signal events of equal importance such that it would be a convenience to be able to test multiple flags in a single cycle rather than basing branches on only one flag at a time.

A simple way to test multiple conditions at one time is to execute a multi-way branch based on the bits being tested. In the case of the FPP there are six flags, too many for a single multi-way branch which can be based on only four bits. A solution is to OR some of the flags together as one of the multi-way branch bits and use the remaining bits directly as part of the multi-way branch address. In that way, one multi-way branch can test all six flags.

When testing the status, if no flags are active, no abnormal condition exists, and the zero value destination of the multi-way branch continues. If one or more of the direct flags is active, the multi-way branch goes straight to a routine to handle the problem. If one of the ORed flags is active, the multi-way branch destination instruction can either ignore the flags or take a second multi-way branch that is based on direct inputs of the flags that were ORed in the first multi-way branch (an advantage of having more than one source for multi-way branch conditions). The second multi-way branch determines which of the ORed flags was active in the first multi-way branch.

FPP Status Register Implementation

An AmPAL22V10 Programmable Array Logic device is used to register the FPP status flags and perform the OR of some of the flags.

This external status register loads new status only as the result of cycles in which the FPP is the selected data path during an instruction execution. When the FPP "F" register is in transparent mode, the external status register is loaded with the flags at the end of an FPP cycle. This results in a one level deep pipeline on status in the same way that ALU status is pipelined one level internal to the ALU. When the F register is in clocked mode, the external status register will load in the cycle following an FPP cycle. This will capture the data that is loaded into the FPP on chip status register at the end of the FPP cycle. This causes the status to be double pipelined for cycles in which the F register is clocked.

The multi-way branch outputs for the first level branch are the following flags: Overflow, Underflow, Invalid, and the OR of the Inexact, OR, NAN, and Zero flags. The multi-way branch outputs for the second level branch are: Inexact, NAN, Zero, and Ground.

These groups of four bits are substituted for the least significant four bits of a branch address to act as a multi-way branch.

In addition to the multi-way branch test for flags, an added output of the status PAL ORs together the Overflow, Underflow, and Invalid flags for use as an interrupt signal to the system interrupt controller, thus giving one additional way to monitor the FPP error flags. Using the interrupt approach eliminates the need to follow floating point operations with multi-way branches in order to test for error conditions. Execution of instructions can proceed, assuming no major problems exist in an FPP cycle. If one of the above mentioned error flags is active, the resulting interrupt will deal with the error.

One last element of the status PAL is that it acts as part of the system control decode by decoding the data path select bits of the control pipeline to enable the FPP output when the FPP is the selected data path.

The logic definition file for the status PAL is listed in Appendix C.

Seed Look-Up Table

The Newton-Raphson division algorithm does a division of A by B by finding the inverse of B (i.e., 1/B) and performing a multiply against A. This scheme works with the Am29325 since finding the inverse of B requires only

a series of multiplies and subtracts which the Am29325 can do in single cycles. But, these multiplies and subtracts are performed only to refine the accuracy of a precalculated seed value (a rough approximation of the inverse of B). So a table of seed values must be available to support division with the Am29325.

This seed table is stored in PROM memory external to the FPP. The B variable is used to address the seed table, and the resulting seed value is fed into the FPP to be refined.

Placing the seed table in the path to one of the FPP inputs normally requires a 32-bit multiplexer to select between the PROM and the direct input bus for loading normal operands in multiply, add, and subtract operations. Building this multiplexer would require at least six hex-2-to-1 multiplexer chips. The PROM and multiplexer would also increase the propagation time needed to load the FPP, thereby requiring the cycle timing to be extended even more than is already required by the FPP.

The implementation of the seed table in this system has been modified to save chips and cycle length. Instead of placing the seed table between the A_BUS and the FPP, it is placed to the side as an appendage of the A_BUS (see Figure 3-3). The inputs and outputs of the table are tied together and to the A_BUS. The internal structure of the table is shown in Figure 3-4. It contains three PROMs, each of which is followed by a three-state output register (the Am27S25 has an internal register). In this arrangement the PROMs can be accessed by the value present on the A_BUS in one cycle and the resulting seed loaded into the registers. In the following cycle the registers can drive the A_BUS with the seed value. This scheme requires three fewer chips and no extension to the FPP cycle time. It is true that two cycles are now required to load the seed value but the cycle used to access the seed table can be combined with the operation of checking for a zero divisor. This operation is generally done during the setup for a divide.

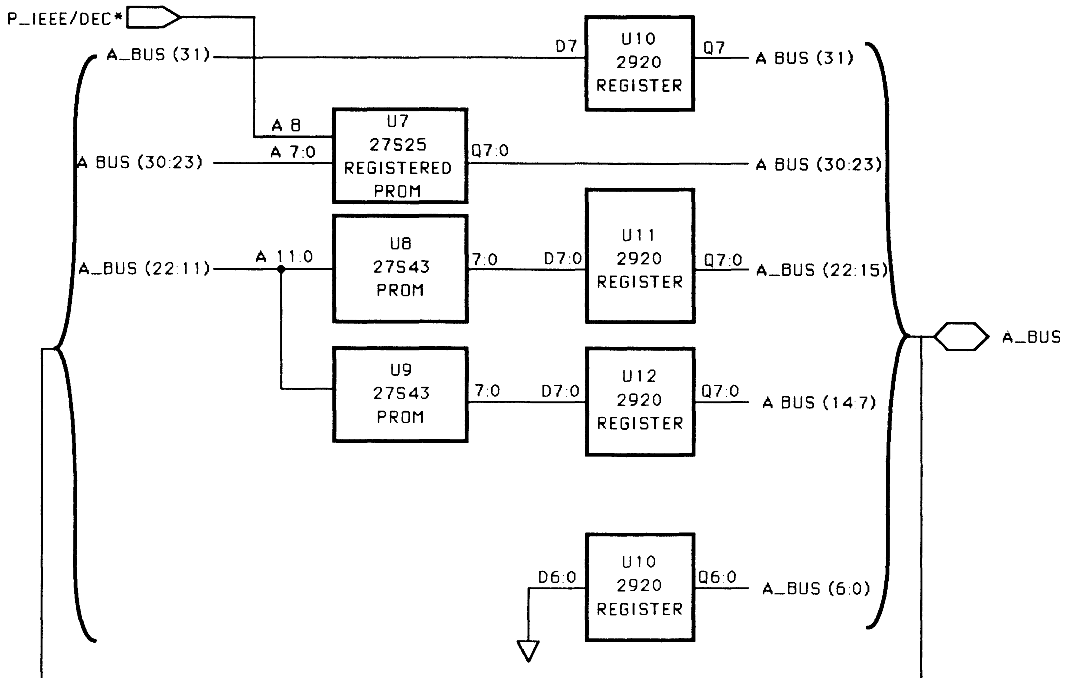


Figure 3-4. Floating Point Block Seed Look-Up Table -- Data Flow Diagram

The detailed connections of the seed table are shown in Figure 3-5. The Am27S25 contains the seed values for the exponent and the two Am27S43s contain the seed for the fraction. The seed table output enable (SEED_OE*) signal is a decoded output of the microcode control pipeline register. The output register of the seed look-up table is clocked by the data section clock.

PARALLEL MULTIPLIER

The entire Parallel Multiplier (PM) block's function is provided by the single chip Am29C323 Parallel Multiplier. This chip performs 32-bit, 64-bit, 96-bit, and 128-bit integer multiplies. It also can perform multiply accumulate using an internal 67-bit accumulator. The PM is shown in Figure 3-6.

Most of the control signals come directly from the control pipeline register. The Parallel Multiplier output enable (PM_OE*) is decoded from the data path select field of the microcode pipeline register. The enable and flow through controls for the instruction register (ENI* and FTI) are tied respectively to GND and VCC to allow instructions to flow directly from the microcode pipeline register to the multiplier, since the microcode pipeline register already provides the one level of pipeline required in the system. The flow through enable on the product register is enabled only when the PM data path is selected via the control decode logic.

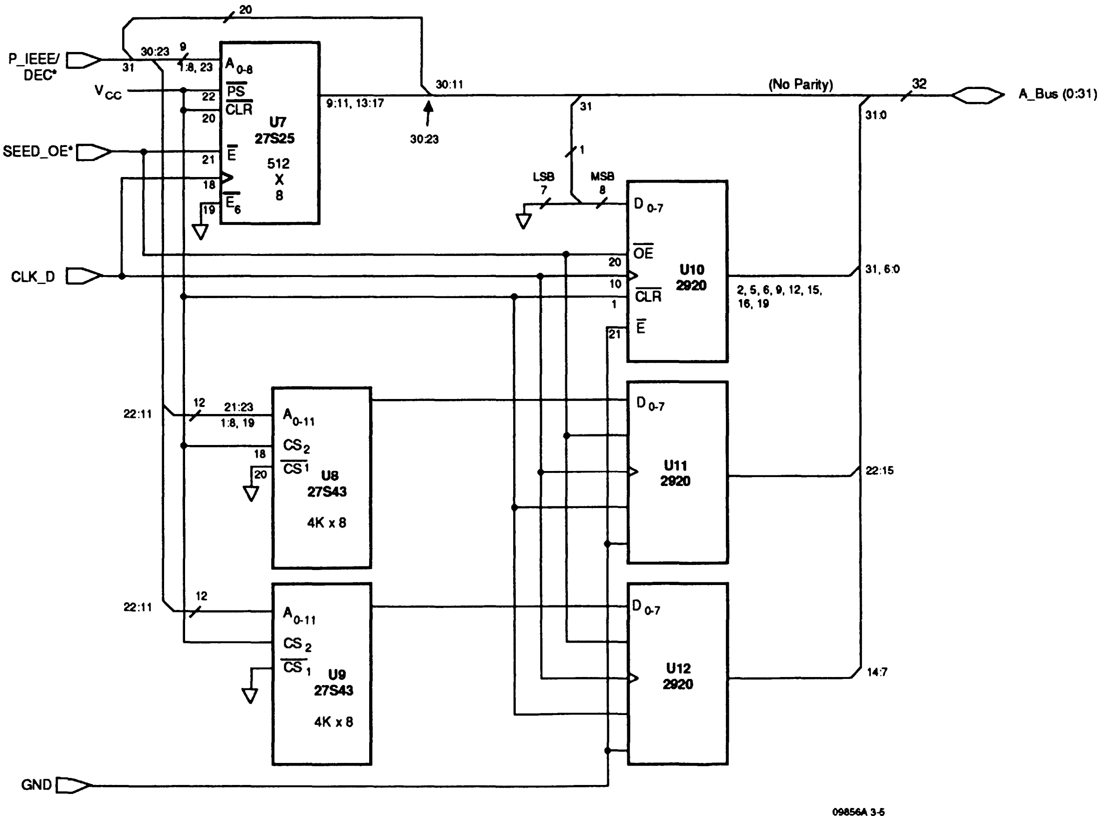


Figure 3-5. Floating Point Block Seed Look-Up Table -- Implementation

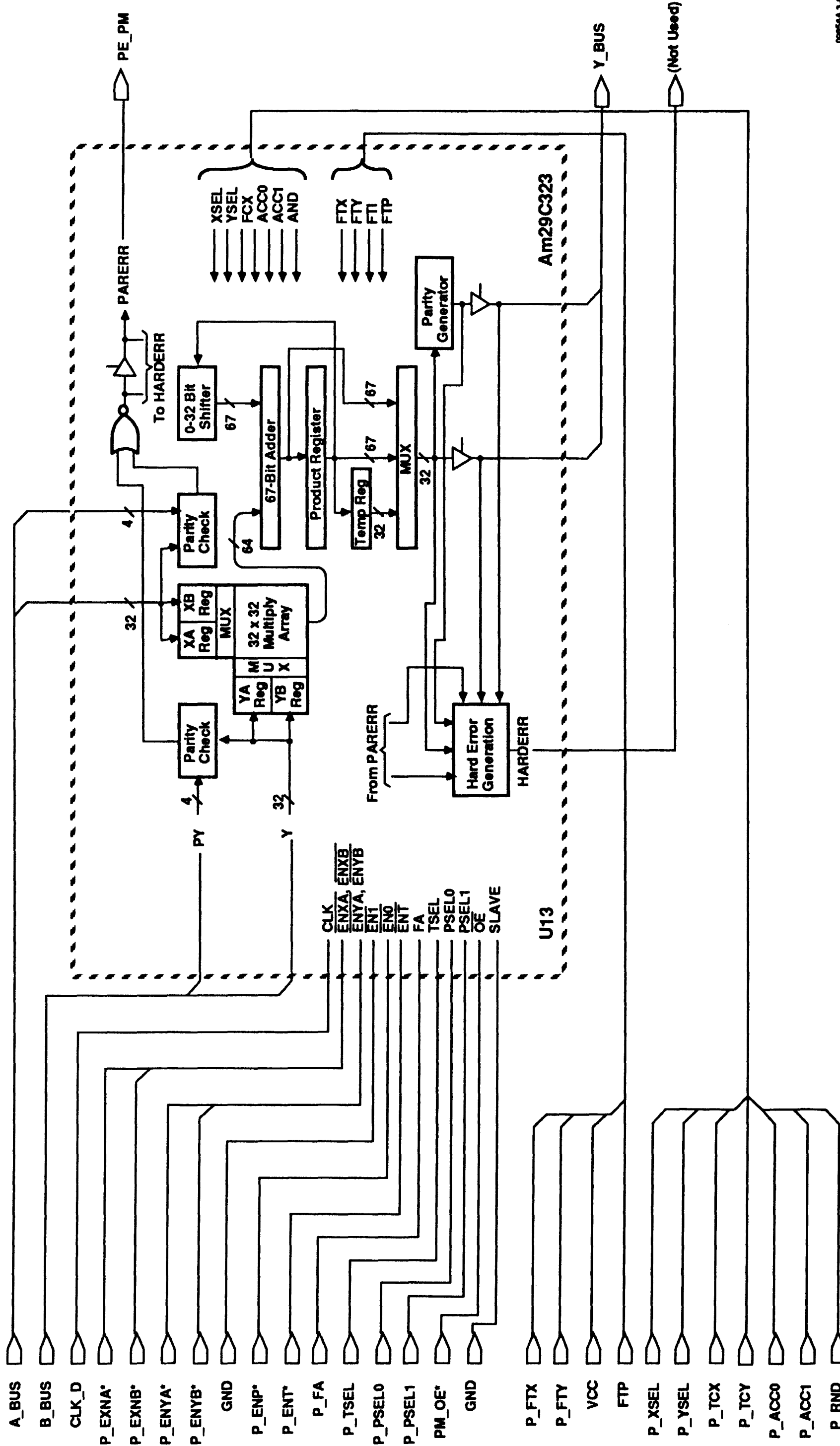


Figure 3-6. Integer Multiplier Block

SECTION 4

Memory and External System Interface

The memory block and external system interface are discussed together in this chapter because of the tight interconnection between these areas. It is helpful to view the two blocks together in order to understand the shared use these blocks make of the memory address bus (MA_BUS) and the memory data bus (MD_BUS). Figure 4-1 shows a block diagram of the data and address paths used in these sections.

One thing to note is that both the memory and the external interface are not elaborate in design. Essentially the external I/O section of this system is just a second port on the system memory. This system does little more than provide a simple arbitration scheme on access to the memory that allows an externally supplied DMA device to load and retrieve data from the memory. Event or interrupt signaling between the CPU and host system is limited to a single pair of interrupt signals, one from host to CPU, one from CPU to host. Memory itself is only a simple bank of static RAM with two address counters on the input that help speed up array calculation.

The reason for this simple approach is that the design to the CPU using the Am29300 family of building blocks is the focus of this application note. Every reader who may find the information in this application note useful will have different memory and I/O requirements to handle and will very likely design individual approachs to memory and I/O. Therefore, only this simple approach is covered here so that more time can be spent discussing the CPU design.

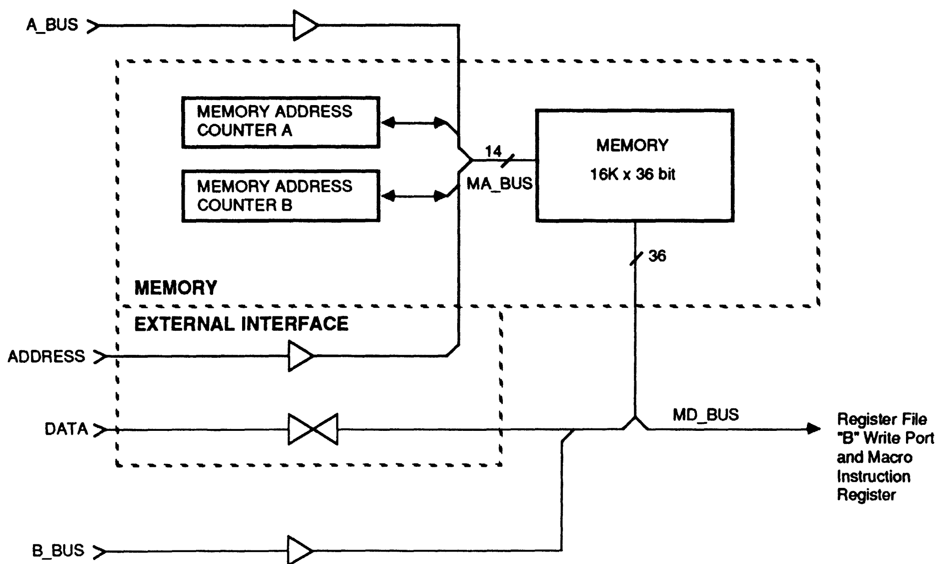


Figure 4-1. Memory and External Interface Address and Data Paths

EXTERNAL BUS INTERFACE CONTROL

Host Access Definition

A block diagram of the host interface controller and its connection to the MA_BUS and MD_BUS buffers is shown in Figure 4-2.

The Am29300 demonstration system is treated as a co-processor to some host system. It ultimately gets all of its instructions, data, and control from the external host system. To provide communication with the host using a minimum of design effort and special hardware, only two portals into the Am29300 system are allowed.

One portal is the Am29300 memory, which is treated as a dual port memory with all words directly mapped into the host bus address space. With this, the host has complete access to macroinstructions and data going into and out of the system.

The second port is a serial diagnostics shift chain that runs through key control registers of the system. This serial pathway gives access to loading and reading the microcode writable control store, to the control pipeline register, to loading and reading the macro opcode map RAM, to the macro opcode register, to the macro status register, and to the interrupt base address register. Through this serial port, the microinstructions are loaded by the host before program execution begins. Also, the system clocks can be controlled by the host to allow diagnostics and code debugging via single stepping and breakpoints.

These portals are controlled by a state machine that is separate from the Am29300 system. The state machine is referred to as the host interface controller. It constantly monitors the external host address bus. When the host presents an address that matches a preset address on the Am29300 system board, the host interface controller is selected to perform one of several interface functions.

Any function requested by the host takes priority over anything that the Am29300 CPU is doing. The host always gains control of the memory address and data buses as soon as the CPU clocks can be stopped and the CPU to memory bus buffers disabled.

The function performed is dependent on the address used, thus the commands from the host to the interface controller are memory mapped. A 24-bit address from the host is assumed for this design. The 6 most significant bits (23:18) of the address are matched to the Am29300 system board address to select the host interface controller. The next two most significant bits (17:16) are used to select a command mode. The 3 least significant bits (2:0)

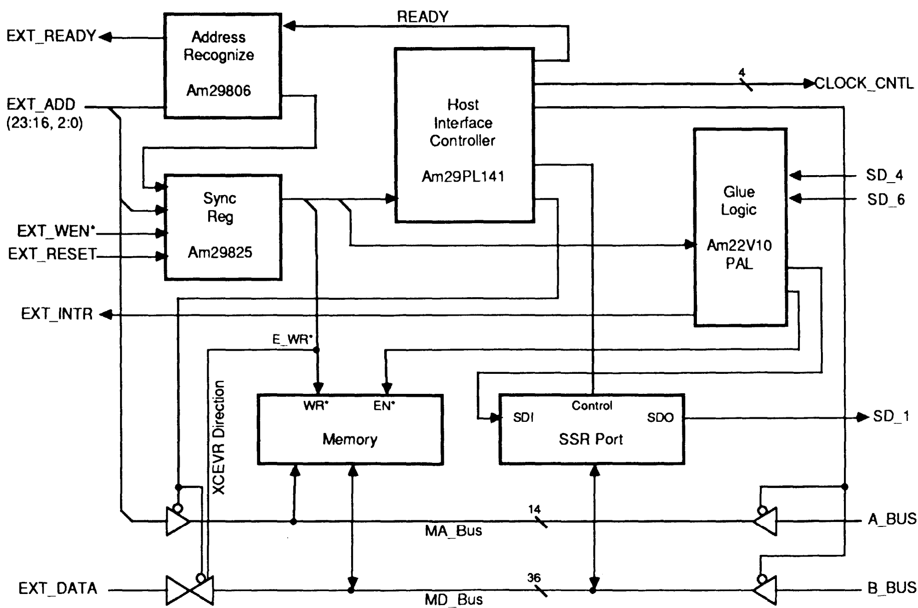


Figure 4-2. Host Interface Block Diagram

are used to select a specific command function within two of the command modes.

Host Interface Block Diagram

The 6 most significant bits of the host address are checked by the address recognition block: if the address matches the board address, then the match signal is fed into the input of a synchronizing register. Also fed into this register are: the external bus write enable line (EXT_WEN*); the external address bits 17, 16, 2:0 [EXT_ADD(17,16,2:0)]; and the host system reset line.

The synchronizing register is clocked by a free-running version of the Am29300 system clock. The register used has special meta-stable hardened circuitry that prevents the outputs from oscillating, regardless of the timing relationship of input data to clock. This register allows the entire Am29300 system to run asynchronously with regard to the host system clock. All the interaction between the host system and the Am29300 system is synchronized to the Am29300 system clock by the register. Each command to the host interface controller is thus presented at the output of this register in synchronization with the host interface controller clock.

The heart of the host interface is an Am29PL141 Fuse Programmable Controller. It is a microprogrammed sequencer with on-chip microcode memory and pipeline register. This sequencer implements the state machine functions needed to control the interaction between the host and the Am29300 system. Used with the Am29PL141 is an Am22V10 PAL. This PAL collects together some glue logic functions: an interrupt signal latch, a multiplexer, and some encoding logic, all of which are described later.

The Am29PL141 provides control signals to the clock gating and distribution section of the Am29300 system. It also controls the enabling of all the buffers and transceivers that connect with the MA_BUS and MD_BUS. The controller acts as a "traffic cop" that allows only one driver on those buses at a time to prevent contention. The controller also manages the loading, reading, and shifting of the Serial Shadow Register diagnostic chain.

The Serial Shadow Register (SSR) diagnostics port is a 32-bit-wide parallel read and write register that also functions as a shift register. Data to be read or written to the SSR diagnostic chain is loaded or read via this port. The port is connected to the host via the MD_BUS. The

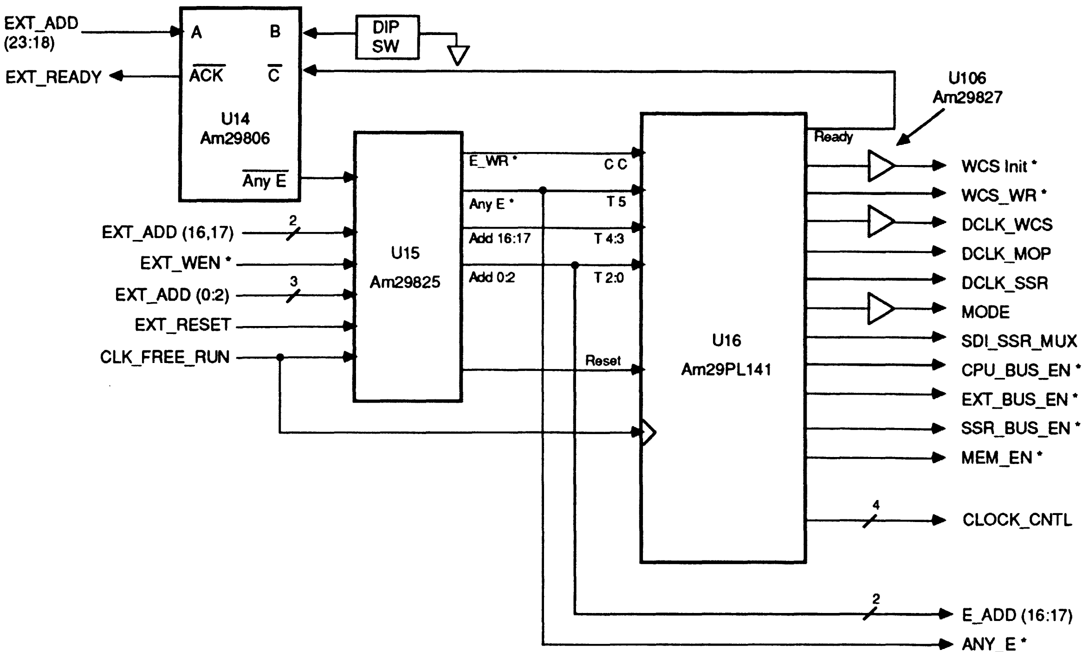


Figure 4-3. Host Interface Controller

port is built from four Am29818-1 SSR diagnostic pipeline registers. These registers, like all the registers in the diagnostics chain in this system, contain one normal parallel input and output pipeline register that is backed-up or "shadowed" by a second parallel input and output register that also acts as a serial shift register. The pipeline register can be loaded from the shadow register and the shadow register can be loaded from the outputs of the pipeline register. This gives the ability to move data into or out of the pipeline register via the shadow register. Data in the shadow register can be serially shifted to other similar registers in the system. By connecting all the diagnostic serial shadow registers together in a serial chain, data can be moved serially through a large number of key registers in the system using very few wires.

The SSR diagnostics port is just an extra section of the diagnostics chain that runs throughout the Am29300 system. This extra section is connected to the MD_BUS to serve as a parallel input and output port that gives access to the serial shadow register chain.

A slightly more detailed view of the Host Interface Controller is shown in Figures 4-3 and 4-4.

Event Signals

The host and the Am29300 system need to be able to signal each other when important events occur, such as the transfer of ownership over sections of the dual port memory. To allow this, a simple interrupt setting and clearing scheme is provided.

The host interrupts the Am29300 system with a command to the host interface controller. The controller in turn sets an interrupt flag in the Am29300 system interrupt controller. The interrupt is cleared when the Am29300 services its interrupt controller.

The Am29300 interrupts the host by using a microcode bit to set a latch that drives an interrupt line on the external bus. The interrupt is cleared whenever the host does an operation on the SSR port. The interrupt latch is implemented in the AmPAL22V10, as shown in Figure 4-4.

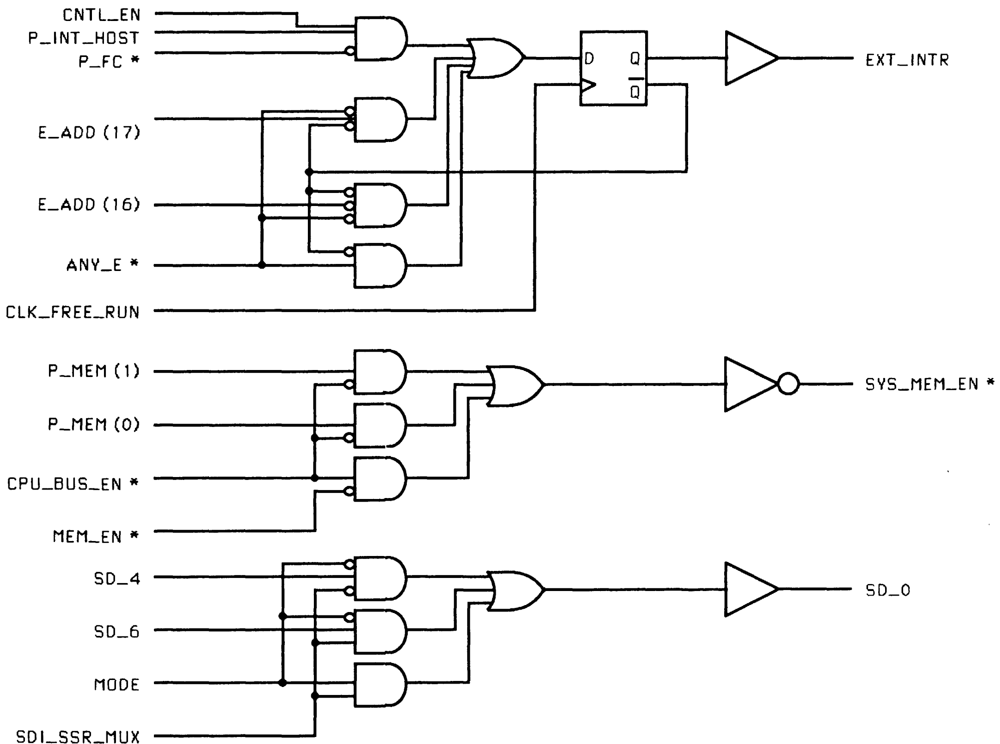


Figure 4-4. U17 Am22V10A Host Interface Glue Logic

Memory Enable

The Am29300 system memory can be enabled by either the Am29300 microcode or by the host interface controller. A simple multiplexer is needed to direct the correct control signal to the memory enable input. This logic is also implemented in the AmPAL22V10 shown in Figure 4-4.

AmPAL22V10 Support Logic

Figure 4-4 shows the logic for the AmPAL22V10 that integrates the interrupt signal latch, SDI multiplexer, and memory enable logic. The logic equation definition file for this PAL is listed in Appendix D.

SSR Diagnostics

SSR Shift Path

Figure 4-5 shows a block diagram of how the serial shadow registers in the system are linked together and how they relate to the macro opcode map RAM, sequencer, and microcode control store. Most of these registers are also depicted in other Figures throughout this application note in their rôles as parallel input and output pipeline registers. Figure 4-5 emphasizes the serial in and out and control connections of the shadow registers also contained in these registers.

The SSR diagnostics port is shown as the starting and ending point for the entire shift chain (or loop as seen here). Data to be loaded into the SSR loop is parallel loaded into this register from the MD_BUS via the bidirectional outputs of the registers in this port (note: the shadow register in the Am29818-1 gets its input from the output pins of the Am29818-1 pipeline register).

Data loaded into this shadow register is then shifted into one of two branches of the SSR loop. One branch flows through the Writable Control Store (WCS) port and the microcode control store pipeline shadow registers. The WCS port is used to address the microcode control store or to receive (load) data from (to) the macro opcode map RAM. The microcode control store shadow register is used to write data into the microcode writable control store or to read the contents of the control pipeline

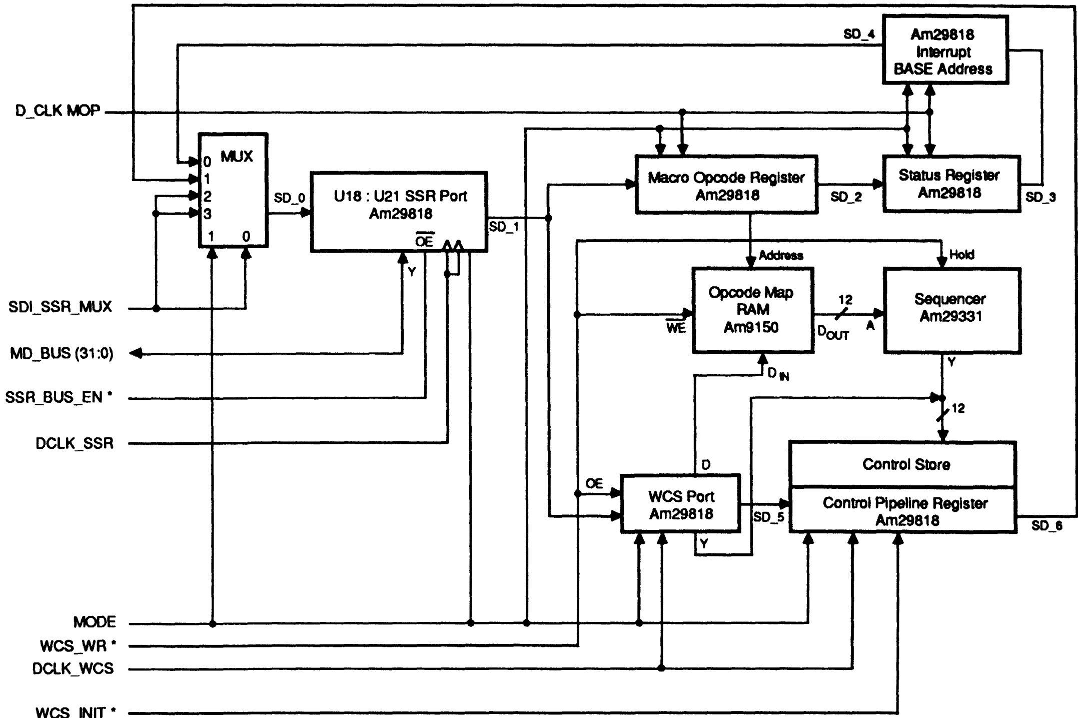


Figure 4-5. Serial Diagnostics Shift Path

register. The second branch flows through the macro opcode, macro status, and the interrupt base address registers. The macro opcode register is used in part to address the macro opcode map RAM .

These branches are separate because it helps to shorten the shift chain length by using branches and because the shift chain clock to the writable control store and WCS port must be separate from the shift clocks to the rest of the diagnostics chain. The shift clocks must be separate because of the way the writable control store is loaded.

The data outputs of the control store are connected to the inputs of the pipeline register as required for normal use in the system. To write the memory, the inputs must be driven with the data to be written, turning the input pins into outputs. In the Writable Control Store (WCS) pipeline register this is fine, since the memory outputs are disabled during the write.

If other diagnostic registers in the system were tied to the same shift clock and mode control lines as the WCS pipeline, there could be a problem every time the WCS is written. The other diagnostic registers not involved in the WCS write would see the same control signals as the WCS registers and would drive their input pins. Depending on what the other registers were connected to, this situation could cause serious contention problems through the system.

For this reason, the SSR used to load WCS is treated separately from other SSR registers in the system. It is worth noting that the only control signal that need be separate is the shift clock. The mode and serial path may be shared with all SSR in the system. Putting the SSR into WCS loading mode, requires the shift clock to load an internal mode flip flop. If the shift clock is active only to the SSR used for WCS when the MODE and Serial Data In (SDI) signals are set high, only the WCS SSR will go into the input pin driving mode.

The end of each branch in the SSR loop returns to a multiplexer at the serial data input (SDI) of the SSR diagnostics port. This multiplexer allows the selection of the shifted branch into the port when the SSR loop is being read rather than written. It also allows the SDI value to be forced when the MODE signal is high. When the MODE signal is high, all the SSRs in the system pass their SDI directly to their Serial Data Output (SDO). This causes the SDI value forced at the input of the SSR port to be passed directly to all SSRs in the system (note: significant propagation time from SDI to SDO for each SSR is involved). In this way the forced value of SDI becomes an additional control signal to all the SSRs in the system. The function of this multiplexer is integrated into the AmPAL22V10 as shown in Figure 4-4.

SSR Reading and Writing

To read the contents of the pipeline registers in the Am29300 system, the host must first send a command to load the SSR throughout the system from the pipeline registers. Then the host must shift the contents of the SSR into the SSR port register (up to 32 bits at a time). The host then performs a read of the SSR port. The host then repeats the shifting-and-reading process until the entire SSR chain has been read.

To write the system pipeline registers, the host reverses the above procedure. Data is first written into the SSR port. Then the SSR chain is shifted to move data into position. The SSR port loading and SSR chain shifting go on until the section of the SSR chain desired is filled. Finally a pipeline load command is issued by the host to load the contents of the SSR into the pipeline registers.

To write the macro opcode map RAM and the microcode writable control store (note: these are treated as a single WCS and must be written together), an address for the map RAM is first loaded into the macro opcode pipeline register via the method described above. Then the address for the microcode WCS is loaded into the WCS port pipeline register. Next, the data to be written into the map RAM and into the microcode WCS is shifted into the WCS port SSR and WCS SSR. A load WCS command is then given which performs the actual write of data into the memories. During the write operation the output of the WCS port is enabled and the Am29331 sequencer output is disabled (via its HOLD pin).

The only trick involved in the SSR Reading and Writing is knowing how much to shift the SSR during each read or write. The problem is that the SSR chain length in this system (and in nearly every real system) is not an even multiple of the SSR port size. During the first (or last) shift operation of either the read or the write of pipeline

registers, it will be necessary to shift fewer than the full 32 bits of the SSR port. The number of bits to be shifted depends on the chain length. One thing to note is that the chain length will be in a multiple of 4 bits because diagnostic pipeline registers are currently available only in 4-bit and 8-bit devices. So, when a shift operation is commanded by the host, the number of nibbles (4-bit shifts) to be shifted must be indicated.

A final note: during the shifting of the WCS SSR, the Am29300 system clocks must be halted. This is due to the fact that pipeline clock and shift clock to the Am9151 may not occur within 65 ns of each other. Since these clocks would occur within the above window in this system, the pipeline clock must not be active.

Controller Description

Function/Command Descriptions

The following is a list of the address values for functions that the host interface will perform when addressed by the host:

Memory Access: Reading and writing of the Am29300 system memory is done by selecting the address for the Am29300 system with address bits 16 and 17 equal to zero. The address for the specific word in memory is contained in address bits 0:15. The host interface controller, upon recognizing the host access, will stop the clocks to the Am29300 system and disable the CPU to MA_BUS and MD_BUS buffers. At the same time the external bus to MA_BUS and MD_BUS transceivers are enabled. This suspends the operation of the Am29300 system and gives memory access to the external host. The write enable line on the external bus determines whether a read or write occurs.

Note that by suspending the Am29300 system operation, the memory access is transparent to (or hidden from) the CPU. There is no action required on the part of the Am29300 microcode or interrupt control.

Serial Diagnostics Port Access: This access is very similar to that of a memory access. The difference is that the SSR port register is being read or written instead of memory.

ADDRESS BITS					FUNCTION
17	16	2	1	0	
0	0	x	x	x	Am29300 Memory Access
0	1	x	x	x	Serial Diagnostics Port Access
1	0	0	0	0	Illegal code
1	0	0	0	1	Halt CPU
1	0	0	1	0	Run CPU
1	0	0	1	1	Single Step CPU
1	0	1	0	0	Single Step CPU Control Section
1	0	1	0	1	Single Step CPU Data Section
1	0	1	1	0	Interrupt CPU
1	0	1	1	1	Reset CPU
1	1	0	0	0	Illegal code
1	1	0	0	1	Load Pipeline Register
1	1	0	1	0	Load Macro Opcode Register
1	1	0	1	1	Load Writable Control Store
1	1	1	0	0	Load Initialization Register
1	1	1	0	1	Load Serial Shadow Register
1	1	1	1	0	Shift WCS SSR Chain
1	1	1	1	1	Shift Macro Opcode SSR chain

Halt CPU: This command throws the Am29300 system clocks in to a continuous stop condition until the mode is cleared by the RUN CPU command or temporarily overriden by one of the single step commands.

Run CPU: This command starts the Am29300 system clocks running.

Single Step CPU: When the CPU is halted, this command will cause all the system clocks to cycle once to advance the state of the CPU one step. Note that gated clocks will be active during this cycle only if their enables are active (i.e., gated clocks operate as they would during a normal clock cycle; they are not forced to operate).

This mode is useful during diagnostic operations to single step the machine between serial load and unload of the SSR diagnostics.

Single Step CPU Control Section: This will step only the clocks in the control section of the CPU. The control pipeline, macro opcode, macro operand, status, sequencer, and interrupt registers may be affected.

This is useful for forcing the control section into a new state under the control of diagnostics, such as a forced branch to a new location in the microcode. This is done by first loading the control pipeline with an instruction to branch via the SSR diagnostics chain. The control section would then be single stepped to execute the branch. Note that during these operations, the data section is not affected and no data is modified.

Single Step CPU Data Section: This operation single steps the clocks only in the data section of the CPU. This may be useful for repetitive diagnostic operations involving only the data section.

Interrupt CPU: This command causes the host interface controller to set an interrupt input to the Am29300 system interrupt controller. The interrupt controller in turn prioritizes the interrupt and causes an interrupt to the CPU when that type of interrupt is enabled.

Reset CPU: This will make the reset line to the Am29300 system active and step all the ungated system clocks. The clocking is required by some parts of the system to affect reset state changes.

Load Pipeline Register: This command will step only the clock to the control pipeline and WCS port for one cycle while forcing the pipeline registers to load data from the SSR chain. This is used to control the state of the pipeline through serial diagnostics.

Load Macro Opcode Register: This steps only the clock to the macro opcode, macro operand, status, and interrupt base address pipeline registers while forcing the registers to load from the SSR chain.

Load Writable Control Store: This command initiates a series of clock cycles that cause data in the SSR chain to be loaded into the writable microcode control store and the macro opcode map RAM from the SSR chain. The address loaded is also specified in the SSR chain.

Load Initialization Register: Like the previous command, this operation loads the writable microcode store. The difference is that only the WCS (Am9151) initialize registers are loaded from the SSR chain.

Load Serial Shadow Register: This causes the contents of all diagnostic pipeline registers to be copied into the related SSR chain elements. This is used to read the Am29300 system state into the SSR chain so that it can be shifted out to the host.

Shift WCS SSR Chain: This command shifts the contents of the SSR port register into the SSR diagnostics chain used for the writable control store. It also brings the bits at the end of the WCS SSR chain into the SSR port register. This is the serial read and write operation of the WCS SSR chain (or loop).

Shift Macro Opcode SSR Chain: This is the same as the previous command but it affects the SSR chain associated with the macro opcode, status, and interrupt base address registers.

Illegal Code: Due to the way the host interface controller algorithm was implemented, this command (address combination) is illegal. If it is used, it will lock up the host interface controller in an infinite loop.

Access Timing

The speed of interaction between the host and the Am29300 system is regulated by both the host and the host interface controller.

Once the Am29300 system is addressed by the host, the host interface controller holds the external bus by driving EXT_READY inactive. This continues until the host interface controller completes the command requested. The EXT_READY signal is then made active and held active until the host stops addressing the Am29300 system. At that time, the host interface controller recognizes that the host has completed the transaction and the EXT_READY line is again made inactive.

In this fashion, either the host interface controller or the host can extend the length of the external bus transaction as required. The signal timing between the host and the host interface is treated as asynchronous. The timing of the host interface itself is synchronous with the Am29300 internal clock cycle.

An interaction diagram is shown below for a bus transaction between the host and the Am29300 system. The single-line dividers indicate one clock cycle of the Am29300 system. The double-line dividers indicate one or more clocks as needed for synchronization or algorithm execution.

The length of an external bus transaction can vary from about 6 Am29300 system clock cycles for a memory access, to about 80 clock cycles for an SSR shift operation. Regardless of the transaction type, the Am29300 system looks to the host like a slave bus peripheral. Sometimes, as in the case of the SSR shift operation, it is a rather slow peripheral.

External Bus Activity	Am29300 System Activity
Address to Am29300 is active on the bus.	CPU is active. CPU owns MA and MD bus.
Address is clocked into the host interface controller synchronizing register.	CPU is still active. CPU still owns internal bus. Host interface controller performs branch to command routine.
External bus transceivers are enabled if needed.	CPU clocks are stopped. CPU bus buffers are disabled. Host interface executes first instruction of command routine. READY may or may not be made active depending on routine.
If READY is inactive, wait for host interface to complete algorithm and make READY active. CPU operation is still suspended.	If READY is active, then wait for host to release external bus by stopping selection of the Am29300 system.
External bus address no longer selects Am29300 system.	CPU still suspended. Host interface waiting to see host release bus.
Lack of external bus address is clocked into host interface sync register.	CPU still suspended. Host interface branches back to idle loop.
External bus transceiver is disabled.	CPU clocks are active. CPU has MA and MD bus access. Host interface waits in idle loop for next command.

Program Definition

A detailed definition of the host interface controller's algorithm is contained in Appendix E.

MEMORY

Memory Components

The memory device used to construct the 16K word x 36-bit memory is the Am99C165. This is a 16K x 4-bit CMOS static RAM memory. The 35 ns access time version is assumed in any timing estimates for the Am29300 demonstration system. Nine memories are used as shown in Figure 4-6.

The Am99C165 is used so that an additional output enable is available to help prevent bus contention with other buffers on the MD_BUS. The memory outputs are disabled whenever the memory write enable line is active. The write enable line is also used to control the direction of the external bus data transceiver and the enable on the CPU data buffer. The delay of the inverter on the output enable input to the memory has been matched by a buffer in each of the other bus drivers just noted. This is so that when a write operation is signalled, each bus driver receives its bus enable or disable signal at the same time as the memory. This overlaps the turn off time of the memory outputs with the turn on time of the other bus drivers to minimize bus contention with the memory.

The enable line to the memory is used to power down the memory when it is not being selected by the Am29300 CPU.

The write enable line to the memory is gated with the Am29300 system free-running clock. This keeps the write line high (inactive) until late in the cycle when all the control signals that feed into the memory enable have settled. This is important for cycles in which there is a change of ownership on the memory address and data buses. The gating with clock ensures that unintended pulses on the write enable line that may occur early in the system cycle will not cause spurious writes in the memory.

Addressing Scheme

Description: With reference to Figure 4-1, the memory address bus (MA_BUS) is not only the address input to the memory, it is also a part of a 4 to 1 multiplexer. There are four address drivers tied to the MA_BUS. They are: the A_BUS to MA_BUS buffer, the External Bus address to MA_BUS buffer, and the two memory address counters. Each of these sources has three-state output drivers and, by careful control of which source is allowed to drive the MA_BUS at any one time, the sources form the 4 to 1 multiplexer.

In this way the memory can be addressed directly by the A_BUS or the External Bus. The memory can also be addressed indirectly by the A_BUS via the memory address counters.

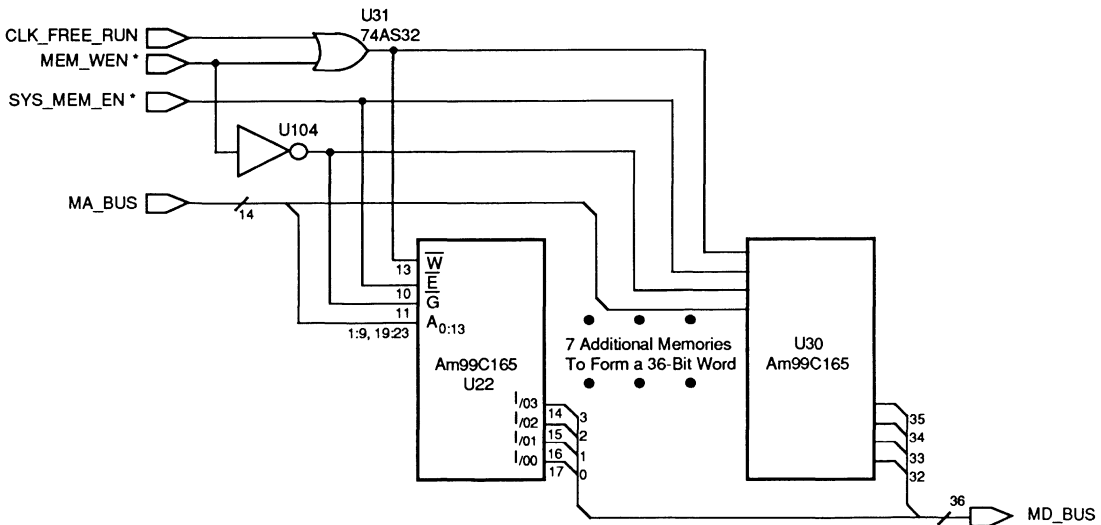


Figure 4-6. Memory

The memory address counters are loadable up/down counters that can serve as address pipeline registers, sequencers, or stack pointers independent of the CPU's data section. They allow sequential reads or writes to memory by the CPU without requiring the CPU to calculate an address on every read or write cycle.

In fact, after loading a memory address counter with an initial address, the CPU can perform sequential read cycles while at the same time continuing to use the data section for other calculations. This is possible because of the dual write port design of the CPU register file. The memory data is loaded into the register file via the B write port while calculation results on the Y_BUS are stored through the A write port.

Two counters are provided to allow for consecutive A and B operand data fetches from two separate arrays of data without the need to constantly reload the counter values. Each counter is built from two AmPAL22V10 Programmable Array Logic (PAL) devices that act as two cascaded 7-bit loadable up/down counters. The counters are connected as shown in Figure 4-7. The logic definition file for the PALs is given in Appendix F.

The two counters are only loaded from the A_BUS and not the External Bus, even though the connection of the counters to the MA_BUS would permit the latter. This is due to the difficulty in coordinating the use of the counters between the CPU and the External Bus. The counters are simply viewed as a resource of the CPU only.

Why This Approach?: Why address the memory from the A_BUS? Doing so means that data in the memory is selected by an address previously stored in the register file. So one cycle must be used to calculate an address in the data section of the CPU, store the result in the register file, and take a second cycle to actually address the memory. Why not just take the address as it is calculated and feed it directly from the Y_BUS to the memory?

First, the access time is better from the A_BUS than from the Y_BUS. The A_BUS address is valid 45 ns into a cycle which still leaves time to access a fast static RAM in the same time that data would normally flow from the A_BUS through the ALU and back to the register file. An address on the Y_BUS would not be valid until 87 ns into a cycle, which would require either that the memory access extend the cycle length significantly or that the address be pipelined into a memory address register and be used to address the memory in a second cycle.

Second, since the register file can present two data words in one cycle it is possible to address the memory and provide write data in the same cycle; the address and data go from the register file to the memory. If the Y_BUS is used as the path to the memory in a write operation, a second cycle must be used to provide the write data.

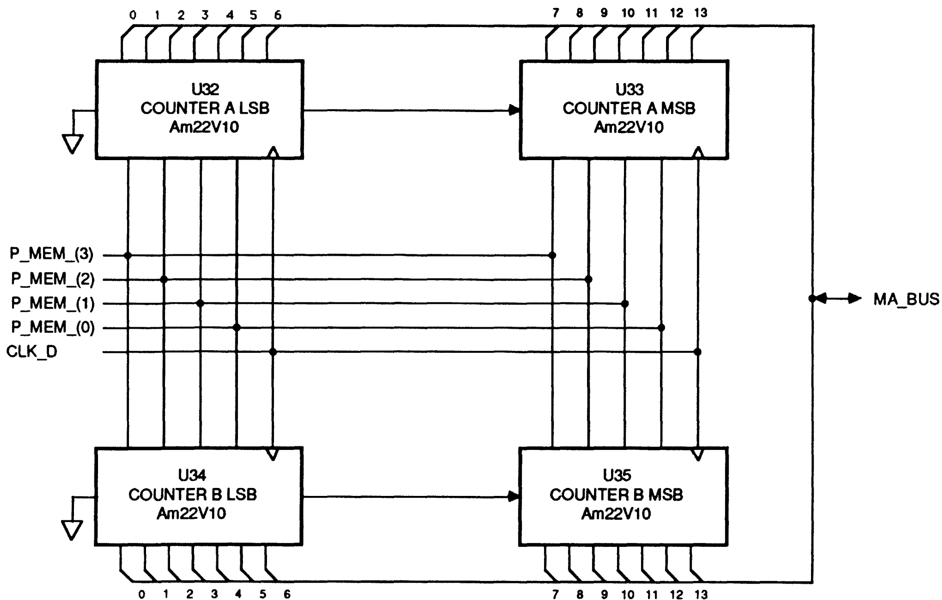


Figure 4-7. Memory Address Counters A and B

Third, the above comments are trick answers. If the two approaches of A_BUS or Y_BUS as the memory address path are carefully examined it can be seen that it is really a situation of "six of one, or half a dozen of the other". Ultimately, in either case, a cycle is use to calculate the address and a second cycle is used to read or write the memory; there is only one data path in the system and only one calculation can occur in a cycle. Between the two approaches there are various ways to overlap other calculations with memory accesses to make the best use of the system's time but either approach takes the same time.

The real difference is that the A_BUS method is simpler from the microprogrammer's point of view. With the A_BUS method a memory read is done in one cycle and the resulting data is in the register file in the next cycle.

With the Y_BUS approach there is a one cycle delay between a read access and the return of data, which requires that the microprogrammer "fill in the hole" in the microcode with other useful work to get the same system efficiency. So, as a designer's preference, the A_BUS for memory address approach is used.

CPU - Memory Buffers

The address buffers from the A_BUS to the MA_BUS and the data buffers from the B_BUS to the MD_BUS are shown in Figure 4-8. The address and data buffers are built from Am29827 10-bit-wide high speed buffers.

The address bus is 14-bits wide to address 16K words of 36-bit-wide memory. But these bits are taken from bit positions 2:15 of the A_BUS. This leaves the two least significant bits of the A_BUS unused and therefore treats the address as being in terms of bytes with the addressing restricted to four-byte (word) boundaries. This was done so that interface with an external host bus would be simpler. Many of the host systems with which this demonstration system could be mated use byte addressing.

With the above address scheme, all the address line numbering is consistent between the host and CPU. In addition, if there were a future need to allow byte addressing of the CPU memory, it would be possible with only a minor change to the address buffer wiring. Also, it

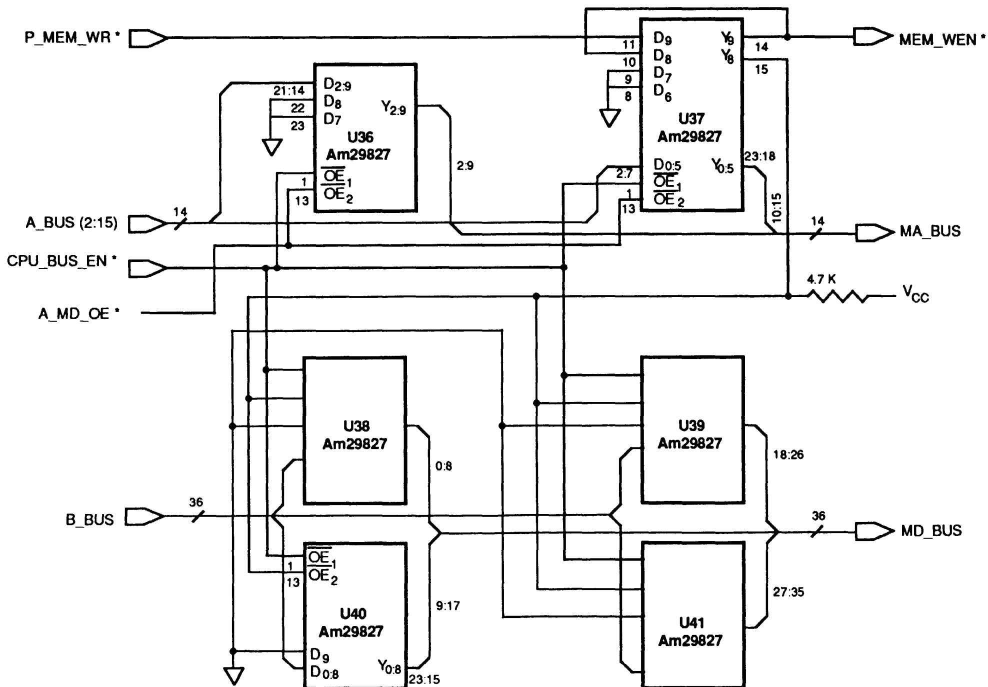


Figure 4-8. CPU to Memory Bus Buffers

may be noted that the parity bits on the A_BUS have been ignored in the MA_BUS since there is no parity checking implemented on the memory address.

The data buffers are arranged as one buffer per byte of the B_BUS (with parity on each byte). Note that, since the B_BUS provides only write data, and read data from the memory is received by the register file, only a unidirectional buffer is needed.

Whenever the external bus interface does not have the memory buses in use, the CPU to memory buffers receive the CPU_BUS_EN* signal to enable the buffers. If the operation is a write, the CPU_WEN* signal is provided by the CPU.

Note that the CPU_WEN* is routed through the address buffer twice and then to the data buffer to enable it on a write operation. This is done to help equalize the timing between this buffer and the output enable on the memory. Note also that the address buffers have a second enable input that is controlled by the control pipeline bits that manage whether the memory address comes from the A_BUS or from one of the memory address counters.

External System Buffers

The address buffers from the External Bus to the MA_BUS and the data buffers from the External Bus to the MD_BUS are shown in Figure 4-9. The address bus is built from Am29827 10-bit-wide high speed buffers. These buffers are connected in exactly the same way as described above for the CPU to memory address buffers.

The data buffers are, however, different from the earlier circuit description. These buffers are Am29863 non-inverting 9-bit high speed transceivers. The transceivers allow data to be both read and written by the external bus.

When the external host system addresses the Am29300 CPU memory, the external bus interface controller halts the system clocks in the CPU and disconnects the CPU from the MA_BUS and MD_BUS by making CPU_BUS_EN* inactive. Then the external bus is connected to the memory by making EXT_BUS_EN* active to enable the external bus buffers. The external bus supplies a write enable if the operation will be a write. Note again that the write enable timing is equalized with that of the write enable to the memory.

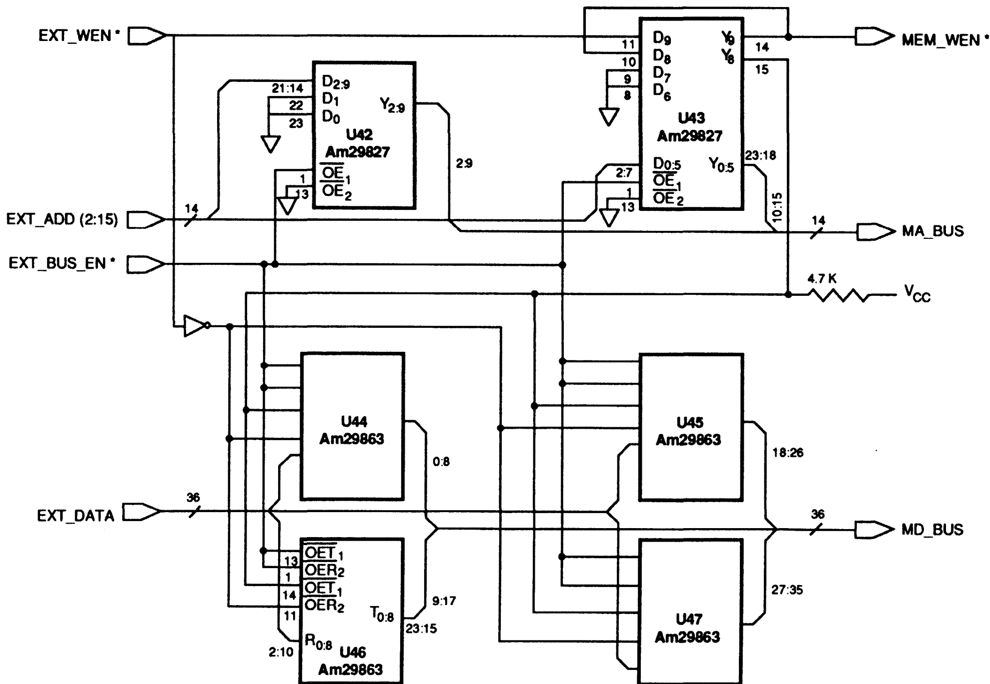


Figure 4-9. External Bus Buffers

SECTION 5
Control Section Description

MACRO OPCODE SUPPORT

Macro Opcode Register

In order for the control section of the CPU to make use of a macroinstruction, the instruction must be selected from memory and loaded into a register that is accessible to the control section.

This register is called the macro opcode register. It is a 32-bit register made from four Am29818-1 pipeline diagnostic registers. This register is shown in Figure 5-1.

The most significant 14 bits (bits 31:18) of the register output are used as the macro opcode. Bits 31:22 are connected to the address inputs of the macro opcode map RAM. Bits 21:18 are connected to one of the Am29331 sequencer's multi-way branch inputs. These lower four bits may thus be used as an opcode modifier via a multi-way branch.

Bits 17:0 are the instruction operand register addresses. These bits are divided into three 6-bit fields, one for each register file port. Bits 17:12 are used as the register file 'A' read port address. Bits 11:6 are used as the 'B' read port address. Bits 5:0 are used as the register file 'A' write port address. These addresses are respectively referred to as the 'A', 'B', and 'C' operand register addresses.

These three addresses allow macroinstructions to specify directly three address operations with two read operands and a separate write operand. Note however,

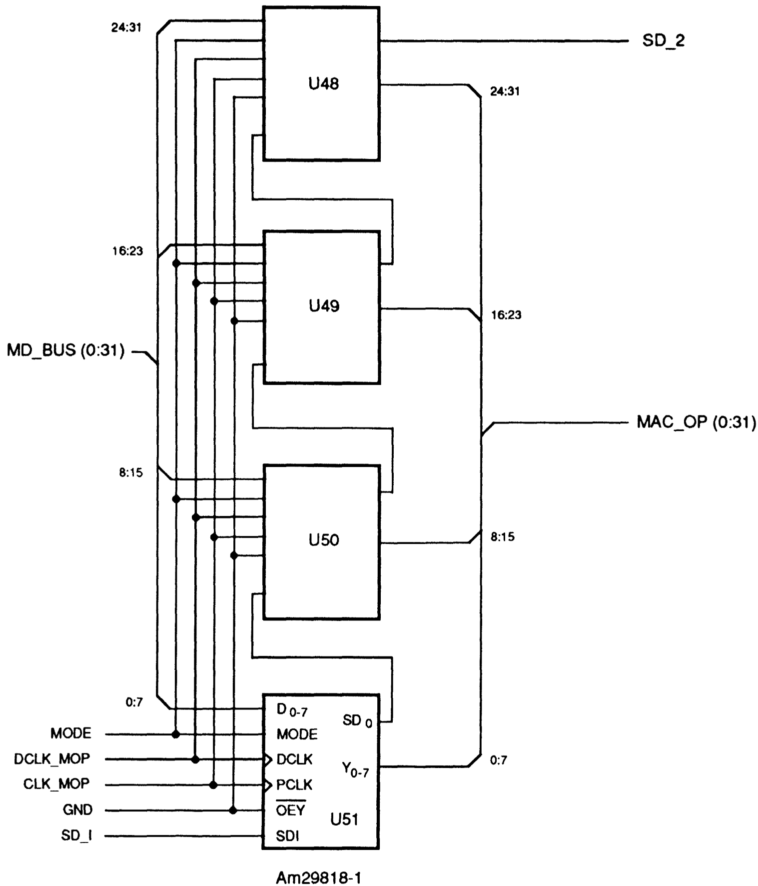


Figure 5-1. Macro Opcode Register

that these bits are connected to the macro operand address counters, which in turn are used to address the register file. This is more fully described in a later section.

In addition, bits 23:18 are connected to the position multiplexer. This allows macro instructions to specify directly the ALU position input as the lower bits of the opcode. Taking the position information from these bits still leaves all of the operand register addresses free for use in three address operations.

Also, bits 4:0 are connected to the width multiplexer. This allows macro instructions to specify directly the width input of the ALU for use in masked operations. Although this overrides this field of the opcode for use as the 'C' operand address, the 'C' operand address may internally be specified as the same as either the 'A' or 'B' operand register addresses. Thus two address macroinstructions involving width, or width and position specifiers are possible.

Macro Opcode Format Restrictions

Because of the large number of possible macroinstruction formats, this application note will not attempt to provide a detailed macroinstruction set definition. It is only important that the format restrictions imposed by the hardware design be stated.

As defined by connections of the macro opcode register, the macro opcode must always be located within bits 31:22. The size and position of the opcode within this field are determined by how the macro opcode map RAM is set up to interpret and map the opcode. The optional opcode modifier (multi-way branch input) must be in bits 21:18 if it is used.

The optional position field must be in bits 24:18 if used and the optional width field must come from bits 4:0 when used.

All three of the operand register addresses are optional and if used must come from the fields specified in the last section. The operand positions are fixed for the 'A' and 'B' operands since they may only come from the 'A' or 'B' operand bits of the macro opcode register. The 'C' operand address may come from any of the three operand fields.

The reason that the 'A' and 'B' operands do not share the positional flexibility of the 'C' operand is that the 'A' and 'B' operands specify registers to be read from the register file. These read addresses are in the critical timing path for the system, and any excess delay in selecting the address adds directly to the system cycle time. A multiplexer like that used for the 'C' operand address would add undesired cycle lengths. The 'C' operand address may afford its multiplexer delay since the 'C' operand address is not used by the register file until late in the machine cycle.

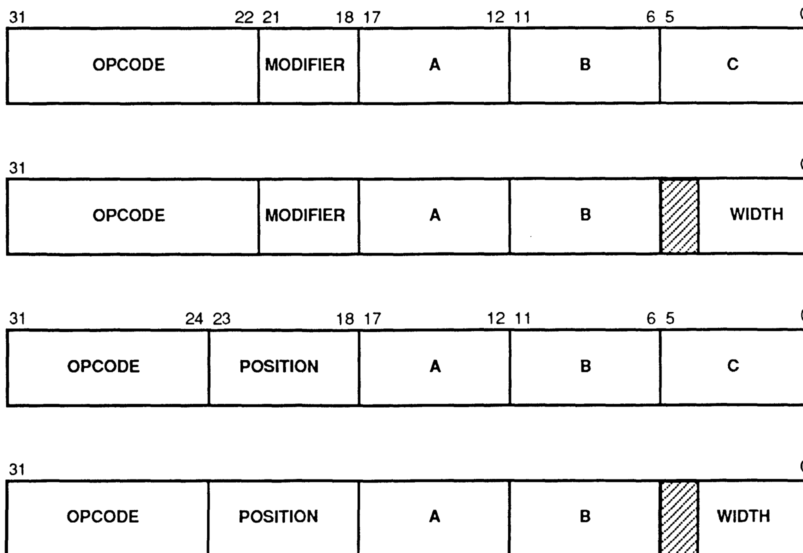


Figure 5-2. Example Macro Opcode Formats

09856A 5-2

Each operand address is optional, because the operand address may always be specified in the microcode.

Any optional field, even an unused portion of the opcode field, may be used as a data operand. Where a field is not used as part of the instruction control, it may be treated as data by loading the macroinstruction into the register file. Once the instruction is in the data section of the system, any data field may be extracted and used in calculations.

Some example macroinstruction formats are shown in Figure 5-2. The instructions are shown in a 32-bit word layout (byte parity is ignored for the moment).

Macro Opcode Decoding Method

The opcode portion of the macroinstruction is the index into the control store for the location of the first instruction of a microcode subroutine. Translating the bit pattern of the opcode into the microcode store address may be done several ways.

The opcode could be used directly to point to a table of first instructions at the base of the microcode store. In such a scheme all microcode routines longer than one word would require the first word of the routine to branch to the remaining part of the routine elsewhere in the microcode store. This would break up many routines into different parts of microcode store. It may also be inefficient, depending on what other functions the branch field of the microcode word could have performed if the first word of the routine did not have to be a branch.

The opcode could be used directly with zeros inserted at the least significant end to form an address that would point to microcode entry points separated by 2, 4, 8, 16, etc. words, depending on the number of zeros appended. This would allow more routines to be located in contiguous words. Only routines longer than the entry point spacing would have to be split by branching to other parts of microcode store. The disadvantage is that where routines are shorter than the entry point spacing, there would be unused holes in the microcode store. When microprograms are expanded and the microcode store gets full (as memories always seem to do), the microprograms will be split more and more times to fit into the unused holes in the microcode store. This will make the micro program more difficult to design and debug as the microcode store fills up.

A PAL may be programmed to decode the opcode into entry point addresses spaced to fit the microprograms. This allows the microcode words of the routines to be kept together in consecutive locations, making design and debugging of programs easier. But each time routines are moved or expanded in size, a new program for the opcode mapping PAL must be defined.

A RAM or PROM memory may be used as a look-up table for entry points in the microcode store. This allows the greatest flexibility. Microcode routines may be located anywhere in control store, independent of the opcode value. The entry points may be spaced to fit each routine. As routines are changed or moved, it is very easy to reload the look-up table with new entry points.

The opcode mapping method chosen for this system is the RAM approach.

Macro Opcode Map RAM

The map RAM is shown in Figure 5-3. It is formed from three Am9150 1K x 4 bit separate I/O high speed RAMs.

Together, the three RAMs provide a 12-bit output which is used as the microinstruction decode address. The address is limited to 12 bits since the maximum size of control store provided for in this system is 4K words.

This decode address is connected to the 'A' address input of the Am29331 sequencer. When this address is selected by the sequencer, a branch is made to the first microinstruction of the selected routine.

The address input to all the Am9150s comes from the most significant bits of the Macro Opcode Register (bits 31:22). This address selects the entry point into microcode control store from the map RAM when a macroinstruction is decoded. The macro opcode register is also used during diagnostics and WCS loading to address the map RAM.

The Am9150 RAMs are always selected and output enabled since no other device shares the 'A' input of the sequencer. Also the Am9151 has no power down mode, so there would be no advantage to deselecting the memory. Note: if lower power in the system is required, an alternate memory to use in implementing the map RAM would be the Am2148. That memory does save significant power when deselected and would increase map RAM access time only slightly.

When the Am9150 RAMs are loaded with data, they are written with data as though they were an extension of the microcode control store. The writable control store write enable line is connected to the Am9150's write enable input.

WCS Port

Also shown in Figure 5-3 is the Writable Control Store (WCS) port. This port is formed from two Am29818-1 pipeline diagnostics registers. The port was shown in block form in Figure 4-5. The port is used as part of the system serial diagnostics and writable control store loading scheme.

The bidirectional "inputs" of the Am29818-1 are connected to the macro opcode map RAM data inputs. When placed in a special mode, the port "inputs" are driven as data outputs. This data is then used as input to the map RAM during a WCS write operation. The data comes from the Am29818-1's internal shadow register.

The outputs of the WCS port are connected to the microcode control store address lines. The WCS port may thus be used as an alternate address source for the microcode control store. During a diagnostic read or write of the control store, the WCS port provides the needed address.

Note that the data for the outputs of the WCS port comes from the Am29818-1's internal pipeline register. The pipeline register contents are independent of the shadow register contents. This allows an address for the microcode control store to be in the pipeline register at the same time data for the map RAM is in the shadow register. These separate registers allow the WCS and map RAM to be written in the same cycle as though they were one writable control store.

Macro Operand Address Counters

These are three identical loadable up/down binary counters made from AmPAL22V10 PALs. They are shown in Figure 5-4. The logic definition file for the PALs is shown in Appendix G.

One counter is used for each operand register address. The counters are loaded from the data outputs of the macro opcode register. The outputs of the counters are tied to the address inputs of the read and write ports of the Am29334 register file.

The counter load, count direction, output enable, and count enable functions are internally decoded from inputs that come from the control pipeline register. These counters are intended for use in array processing algorithms, one example being a digital signal processing algorithm for a filter.

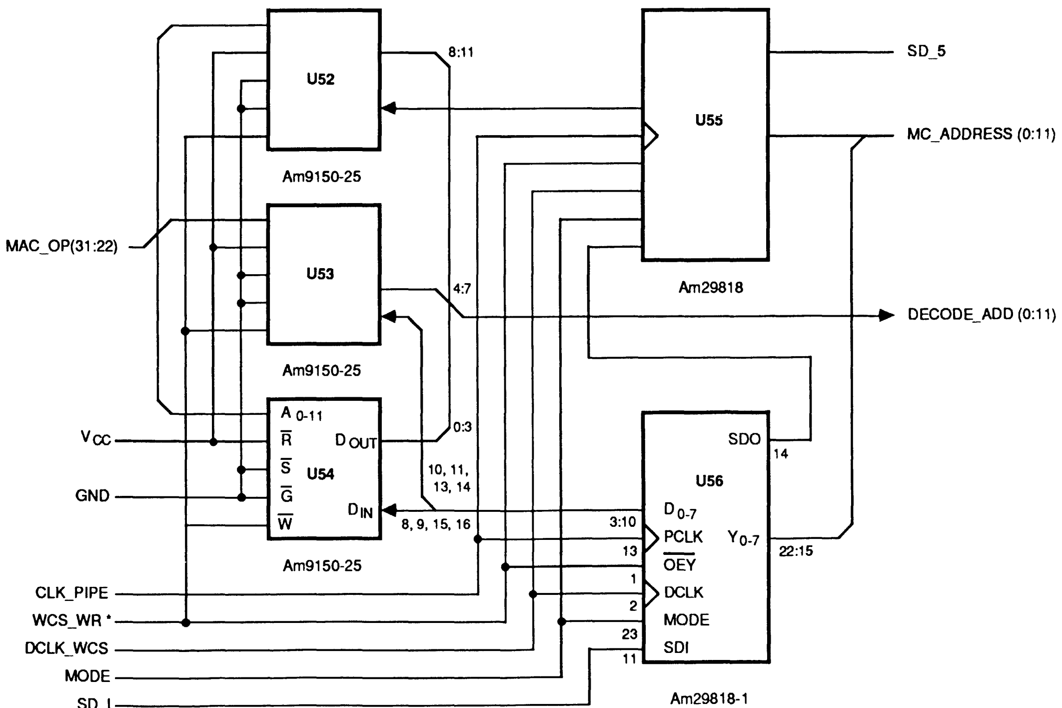


Figure 5-3. Macro Opcode Map RAM

The counters make it simple to perform the same calculation on arrays of data stored in the register file. One microinstruction or a short microinstruction routine can loop on an array calculation and at the end of each calculation cycle simply increment the operand address counters. In that way, new operands are fetched for each calculation on the array without the need for the microcode instructions to directly specify operand addresses.

Control pipeline bits determine whether the microcode operand address or the macro operand counter address is used. The selection is independent for each operand address. Thus, an example would be the operand 'A' address' coming from the microcode while the 'B' operand and 'C' operand addresses come from the counters.

An additional feature is that the 'C' operand counter address may be directed to the Am29334 register file 'B' write port address input. This allows the 'C' operand address to come from microcode while the 'C' operand counter address is used in writing data from system memory into the register file via the second write port. This means that CPU calculations may continue uninterrupted while new data is being loaded into the register file. Also, as long as data is coming from sequential locations in memory and going to sequential locations in the register file, the memory address counter and 'C' operand counter may be incremented together, thus loading several memory words in sequence. This loading may be accomplished without repeated address calculation by the CPU.

Operand Counter Use Example

To help illustrate the use of the operand address counters a typical Finite Impulse Response (FIR) digital signal processing filter algorithm is described here.

An FIR digital filter takes in a stream of amplitude samples from an analog waveform. Each sample is processed through a series of calculations to produce an output value. The resulting stream of output amplitude values produces a waveform that is the result of a filter operation on the input waveform.

The calculations involved are a series of multiplies between different coefficient values and several past input samples. The result of each multiply is accumulated to produce one output value. The number of coefficients

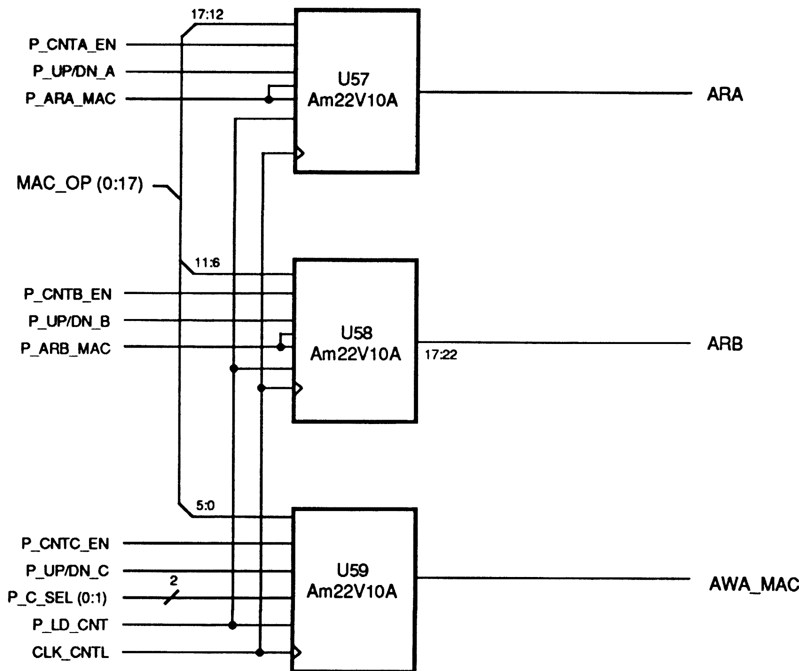


Figure 5-4. Macro Operand Address Counters

and retained past samples determines how selective the filter operation is. The values of the coefficients determine the type of filter operation; e.g., bandpass vs. lowpass.

The algorithm for calculating one output value would be the following:

```
Sum := 0;
for n = 0 to number_of_coefficients do
  Sum := Sum + (Sample(x - n) * Coefficient(n));
```

Each time a new input sample is acquired, the new sample becomes Sample(x), and all past samples shift down in the sample array such that Sample(x - 1) := Sample(x) for all x. Note that the number of retained past samples is equal to the number of coefficients.

This algorithm may be implemented with two arrays of data and a temporary register. One array contains coefficients and the other contains past input samples.

The coefficient and sample operands may be multiplied in a single system cycle by either the Parallel Multiplier or the Floating Point Processor. The Parallel Multiplier may also perform an accumulate in the same cycle. The Floating Point Processor requires a second cycle to do the accumulate function. So for each multiply and accumulate operation on a sample-coefficient pair, either one or two cycles are needed.

Obviously the operand counters may be used to address the data arrays. As each coefficient-sample pair is multiply-accumulated, the counters are incremented to point to the next pair of operands. This allows the inner multiply-accumulate loop to be only one or two microinstructions long.

One feature of the operand counters adds to the efficiency of this algorithm. When an operand counter reaches either the maximum or minimum count value, the counter will reload the original count value from the macro opcode register on the next increment. This creates a counter that may treat the register file as a circular buffer. The length of the buffer is determined by the distance from the original count value to either the base or upper limit of the register file address.

Note also that if one counter is always incremented while the other is decremented, two circular buffers may share the register file. One has a lower bound of zero and the other an upper bound of 63. With this scheme two equal size buffers could be up to 32 words each.

The circular buffer approach to the arrays works well with the FIR filter algorithm. At the end of each output value calculation, the counter addresses will point back to the first coefficient-sample pair, ready for the next input sample iteration.

Note that if on the last multiply-accumulate cycle of an iteratation the sample operand counter is not incremented, and the 'C' operand counter is used to load a new sample from memory into the oldest sample array location, the effect will be to shift all the samples down by one in the array while overlapping the new sample load with the last cycle of a sample iteration.

One additional cycle at the end of each iteration may move the output value from the register file to the memory. No memory address calculation cycle is needed since the memory address counter may be used to address the memory.

With this scheme only one cycle of overhead between iterations is needed. Therefore, assuming clocked multiply operation of the PM to achieve single cycle multiply-accumulate execution, a 31 coefficient FIR could complete one output value iteration in 32 cycles. Assuming a 100 ns cycle time (100 ns clocked multiply in the PM), that would allow over 312,000 samples per second or an input bandwidth of over 156 kHz. A 9 coefficient filter would have a 500 kHz bandwidth.

This is an example of how a microprogrammed system may have its architecture tuned to a particular application for the best possible performance. Much of the performance comes from the microprogrammed system's ability to control and perform several parallel functions at one time.

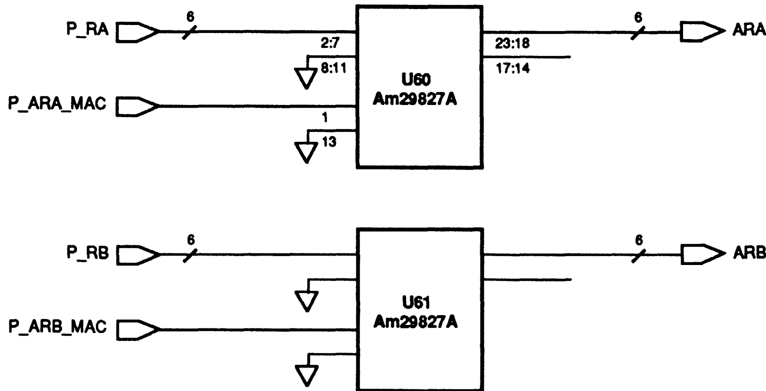
REGISTER FILE ADDRESS MULTIPLEXER

The Register File Address Multiplexer, shown in the block diagram of Figure 1-2, is made up of four separate multiplexers. One multiplexer is used for each register file address port; two read ports and two write ports.

Read Ports A and B

These multiplexers are shown in Figures 5-4 and 5-5. Each multiplexer is really a three-state bus that may be driven either from the control pipeline register via an Am29827 three-state buffer or from an operand counter output. A bit for each address from the control pipeline selects which source may drive each address bus.

The Am29827 three-state buffers are needed in addition to the three-state outputs of the control pipeline because each operand address is 6 bits. This number does not fit



Figrue 5-5. Register File Address MUX, Read Ports

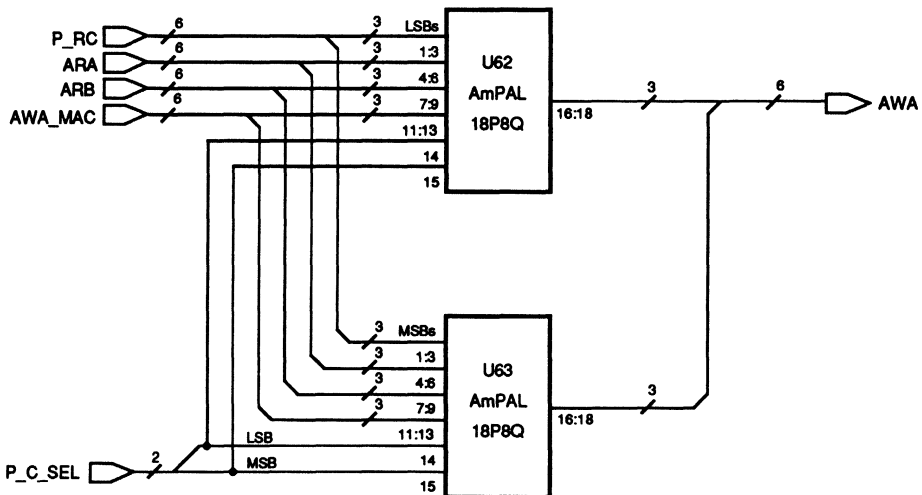


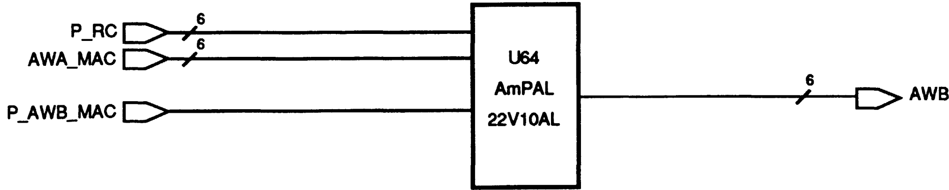
Figure 5-6. Register File Address MUX, Write Port A

well into the 4-bit boundaries of each slice of the microcode control store. So to avoid wasting control store bits, the external three-state buffer is used to gate the control pipeline address onto the register file address bus rather than trying to use the control store's own three-state outputs.

Write Port A

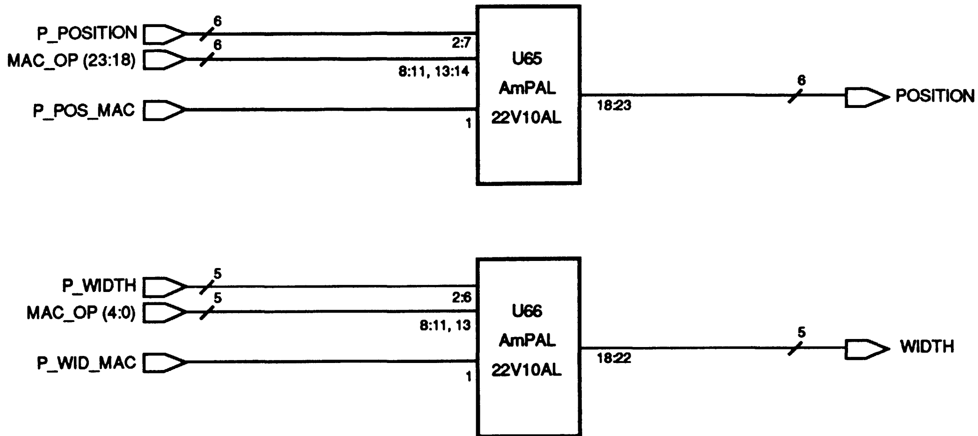
This multiplexer is implemented by a pair of AmPAL18P8 PALs. It is shown in Figure 5-6. The logic definition file for the PAL is contained in Appendix H.

It is this four input hex multiplexer that allows the 'C' register file operand (i.e., register file 'A' write port) address to come from four possible sources. The address may be provided from the 'C' operand in the control store, 'C' operand counter, 'A' operand final address, or 'B' operand final address. The 'A' and 'B' operand addresses are referred to as final because the multiplexer input is taken from the register address buses after the choice between control pipeline or operand counter has been made for the 'A' and 'B' operand addresses. The select bits for the multiplexer come from the control pipeline.



09856A 5-7

Figrue 5-7. Register File Address MUX, Write Port B



09856A 5-8

Figure 5-8. Position and Width MUX

Write Port B

This multiplexer is made from an AmPAL22V10. It operates as a two input hex multiplexer. It is shown in Figure 5-7. The logic definition file for the PAL is given in Appendix I.

It selects either the control pipeline 'C' operand address or the 'C' operand counter address as the source for the register file 'B' write port address. The select bit comes from the control pipeline register.

POSITION AND WIDTH MULTIPLEXERS

The position and width multiplexers are implemented with AmPAL22V10A PALs. They are shown in Figure 5-8. The logic definition file for the PALs is given in Appendix I.

Each is a two input hex multiplexer, identical to the multiplexer used for the B Write Port Mux. They select from the Position and Width values that may be provided either from the control pipeline or the Macro Opcode Register. The select control comes from the control pipeline.

'A' speed PALs are used here since these multiplexers are in the critical path to the ALU. They must use 7 ns less delay than the combined delay of the 'A' Read Port Mux and Register File access time. The required 7 ns advantage is consumed by the ALU's longer propagation delay from Position input to Y output vs. Data input to Y output.

SEQUENCER

The sequencer is a 16-bit-wide address generator that controls the execution sequence of microinstructions stored in the microcode control store. It may handle interrupts or traps at any microinstruction boundary. An interrupt or trap is treated like an unexpected procedure call.

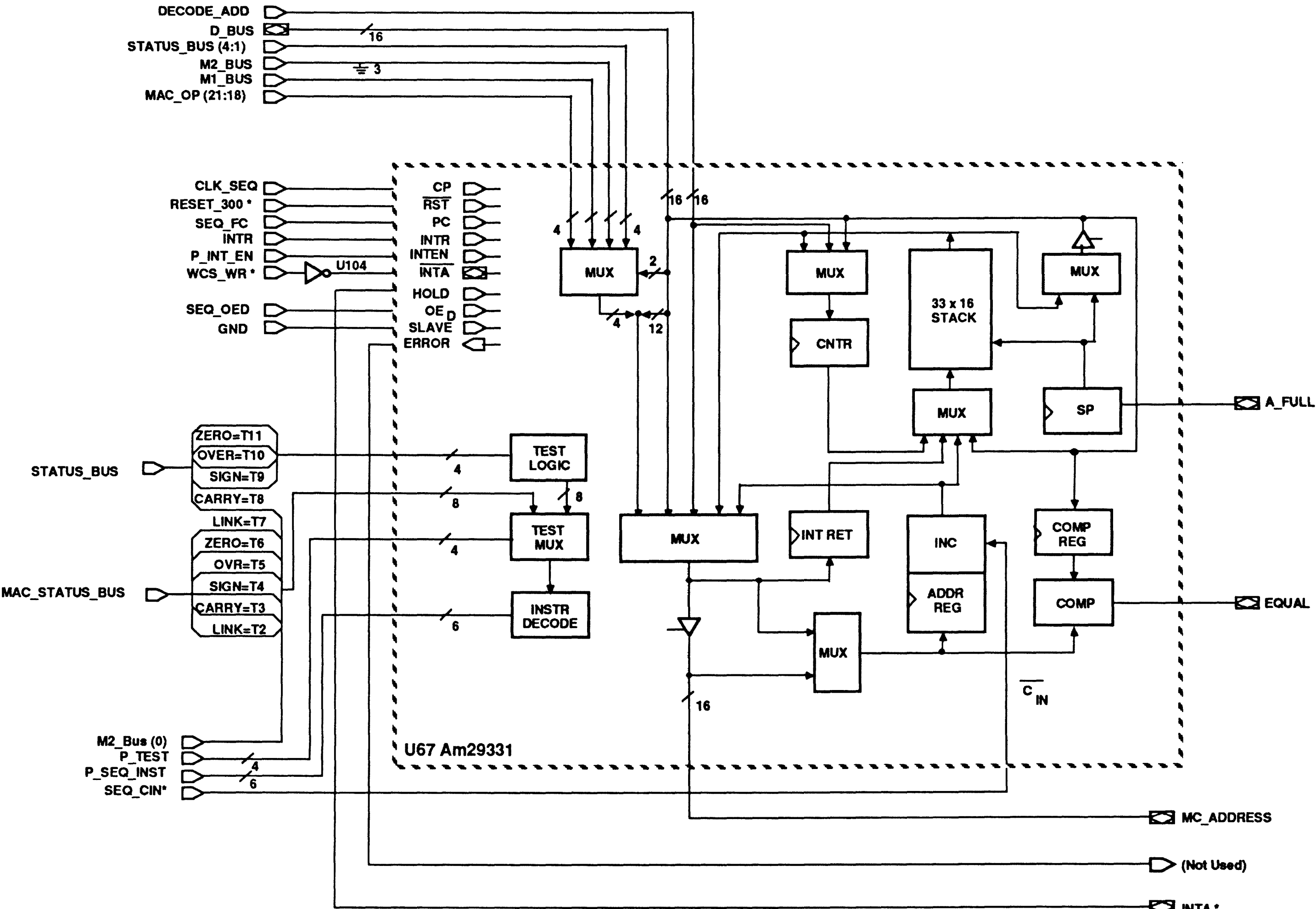


Figure 5-9. Sequencer Block

Two independent branch inputs as well as four multi-way branch address sources are provided. One of the branch address inputs is bidirectional and may be used to read or write information in the sequencer's internal 33-level deep stack.

A 16-bit counter, test condition multiplexer, and breakpoint address comparitor are also provided. The breakpoint comparitor is used as a hardware aid to microcode debugging. The connections to the sequencer are shown in Figure 5-9.

The sequencer's 'A' branch address input is connected to the Macro Opcode map RAM output and is the path through which the macroinstruction specifies its entry point into microcode.

The 'D' branch address input is tied to the D_BUS. Through this path, branch addresses or constants come from the control pipeline register and data may be exchanged with the data section of the CPU.

The 'M0' multi-way branch address input is connected to the macro opcode register bits 21:18. These bits may be used as a modifier to the macro opcode via a multi-way branch based on these bits.

The 'M1' multi-way branch address inputs come from the Floating Point Processor (FPP) external status register. These bits are the overflow, underflow, invalid, and 'extra' status flags from the FPP. The 'extra' status flag is the OR of the zero, NAN, and inexact status flags from the FPP. A single multi-way branch on these inputs may be used to detect and handle quickly any of the catastrophic status conditions from the FPP. If the 'extra' flag is active, it indicates that a second multi-way branch may be used to determine which of the 'extra' status flags is active.

The FPP zero, NAN, and inexact status flags are connected to the 'M2' multi-way branch input of the sequencer.

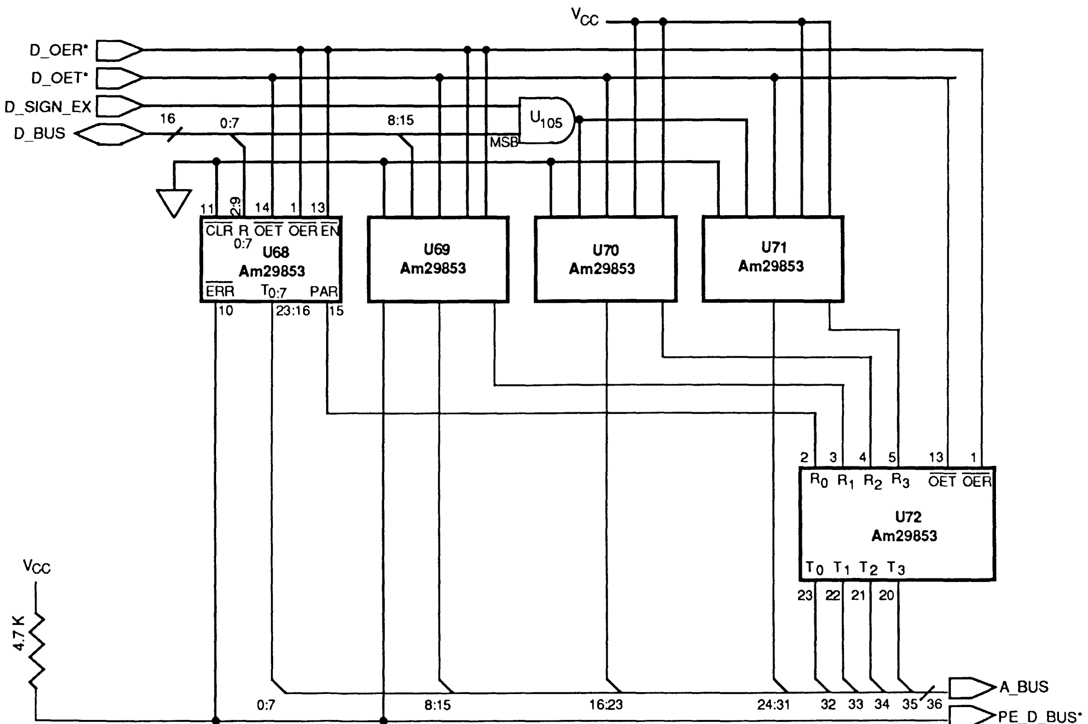


Figure 5-10. D Bus Transceiver

The 'M3' multi-way branch input is tied to the ALU microprogram status outputs so that an alternate means of checking ALU status is available. A multi-way branch based on these bits is able to check multiple condition flags in a single cycle.

The Force Continue and Carry-In inputs of the sequencer are active in a trap operation to prevent state change in the sequencer and capture the address of the trapped instruction in the interrupt return address register. Carry-in (CIN*) is driven high by a trap event signal from the trap logic in Figure 5-11. The trap event signal is also ORed with a signal from the control pipeline (P_FC) so that either signal will cause Force Continue to go high. The interrupt request input comes from the Trap circuit shown in Figure 5-11.

The sequencer's HOLD input is driven by the inverted value of the WCS_WR* signal from the host interface controller shown in Figure 4-3. When this signal is active, the sequencer's output will be three-stated so the WCS Port may drive the microcode control store address lines without contending with the sequencer's output drivers.

The Slave input is grounded since no use of the mode is made in this demonstration system.

The test condition inputs of the sequencer come from three sources. Conditions 11 though 7 are the ALU status bits for zero, overflow, sign, carry, and link. Conditions 6 through 2 come from the Macro Status Register; these bits are the macro version of the same ALU status bits. Condition 1 comes from the FPP external status register bit for zero. Condition 0 is unused.

Control for the sequencer's interrupt enable, test condition select, and instruction input comes from the control pipeline register.

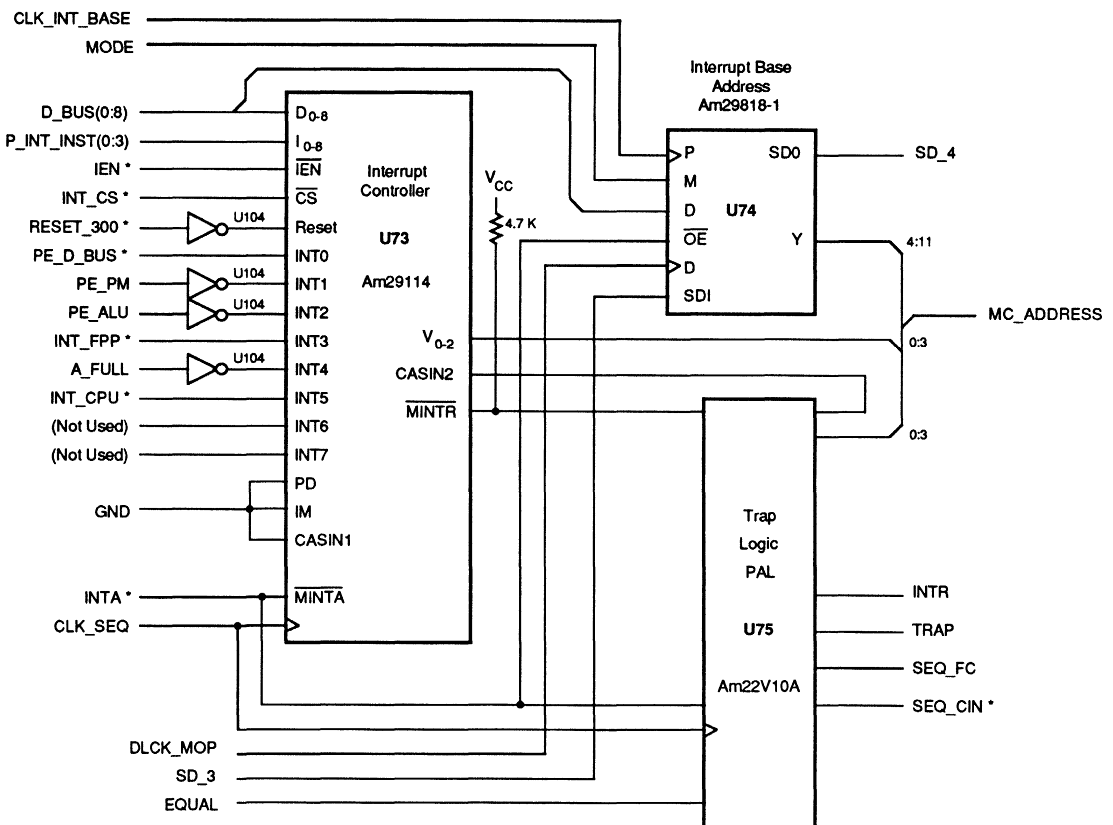


Figure 5-11. Interrupt and Trap Logic

The sequencer's D_BUS output enable comes from the control decode logic.

The sequencer A_FULL signal is used as an interrupt signal to the system interrupt controller.

The Equal (breakpoint) signal is used as a trap event signal to the Trap Logic.

Interrupt acknowledge goes to the interrupt controller and trap logic to enable the interrupt and trap vectors onto the microcode control store address bus when an interrupt is executed.

The 'Y' outputs of the sequencer drive the microcode control store address lines to select each microinstruction.

D BUS TRANSCEIVER

The transceiver between the A_BUS and the D_BUS is shown in Figure 5-10.

The D_BUS has no parity bits included where as the A_BUS does contain parity. It is therefore necessary to provide parity generation for the data moved from the D_BUS to the A_BUS.

The D_BUS is only 16 bits wide vs. the 32-bit-wide A_BUS. Thus it is also necessary to provide bus drivers and parity generators for the upper two bytes of the A_BUS, even though no variable data is passed to the A_BUS from the D_BUS through those bits.

The transceiver and parity generator/checker function are combined in a single device type: the Am29853. Four of these are used in addition to an Am29862 inverting transceiver. The inverting transceiver is used on the parity bits because the Am29853 uses odd parity while the Am29300 system uses even parity.

As an added convenience for when numeric constants are passed from the D_BUS to the A_BUS, an AND gate is provided to drive the inputs of the upper two bytes of transceiver. If the AND gate is enabled by the control pipeline, the most significant bit of the D_BUS will be copied to all the upper bits on the A_BUS, thus performing a sign extend for two's complement numbers. If the AND gate is disabled, the upper bits of the A_BUS are forced to zero.

INTERRUPT CONTROL

Interrupt and Trap Philosophy

What is a Trap?

Traps are events that require the immediate attention of the CPU. The urgency of the event is so great that the CPU must not even complete the execution of the instruction in progress in the cycle that the trap request happens. The CPU must not change any machine state in that cycle; it must store the address of the instruction that was to have been executed and must branch to a routine that services the trap event.

The implication here is that the trap will prevent some disastrous change in machine state from which no recovery would be possible. Also implied is that the trap servicing routine may repair what ever the problem is and then return to complete the execution of the instruction where the trap occurred.

One additional implication is that the trap event may be signaled early enough in the instruction cycle to prevent the clocking (change of machine state) that normally occurs at the end of each instruction.

An example of a trap event could be a miss on cache memory access. To complete an instruction when the data being accessed from a cache is invalid would be a disaster with little chance for recovery. If a trap routine to update the cache may be executed instead of completing the instruction, the program may be saved. After the cache has the correct data, the trap routine may return to the aborted instruction to continue execution of the program as if no problem had existed.

Another example of a trap would be a program breakpoint. When debugging a program it is very useful to be able to stop execution of a program just before executing a particular instruction. If this is done, the state of the machine before executing the breakpoint instruction may be examined. To do this the address of the breakpoint instruction is recognized as the instruction is fetched from microcode control store. In the next cycle before the instruction may complete, a trap occurs which branches to a debugging routine. When the programmer is ready to continue the program, a return from trap completes the execution of the breakpoint instruction. The breakpoint trap operation is easy to do, and hardware to implement

it is already provided in the Am29331 sequencer. The breakpoint trap operation will be shown in the Trap Logic described later.

What is an Interrupt?

Interrupts are events that require the attention of the CPU soon.

"Soon" is defined as faster than might happen if the event were polled by a CPU program but later than a few microinstruction execution cycles.

Interrupt events and the resolution of an interrupt are not directly tied to the CPU state. No disasters occur if a few cycles pass by before the interrupt may be handled.

Examples of events handled via interrupt could be: external mechanical events such as switches being opened or closed, an impending stack-full situation, a message signal from another processor, or a peripheral delay timer indicating time-out.

In this demonstration system one other class of interrupt source is included. It is the parity error. A parity error implies corrupted data in a program that cannot be corrected. Since the influence of corrupted data on the program is difficult to determine or correct for, the affected program should be aborted. A parity error is, therefore, important to detect so that the program in which it occurs may be terminated and perhaps rerun with corrected data.

Parity errors are treated as interrupts rather than traps for two reasons. The indication that an error has occurred comes fairly late in an instruction cycle and is therefore difficult to use as a trigger for a trap. When a parity error occurs, the program is generally corrupted and will be terminated; whether the termination happens in the cycle following the error as would be the case with a trap, or within a few cycles, as with an interrupt, is unimportant.

Interrupt Operations

There is no need to design an interrupt circuit from scratch when one already exists. The Am29114 interrupt controller is used in this system. It provides interrupt latching, priority, masking, and vector generation for eight interrupt inputs.

Interrupt Controller

Six interrupt sources are used in this Am29300 system; the two remaining interrupt source inputs are available for software generated interrupts.

The interrupt and trap circuit block diagram is shown in Figure 5-11.

The three highest priority interrupts are parity error signals from the D_BUS, the Am29C323 Parallel Multiplier, and the Am29332 ALU.

The next priority interrupt is a signal from the FPP external status PAL, which indicates that one of the following status flags is active: Overflow, Underflow, or Invalid.

The next priority interrupt is the A_FULL signal from the Am29331 sequencer. This interrupt indicates that the sequencer stack will be full if three additional stack pushes occur.

The next interrupt is the external bus interrupt signal from the host interface controller. This is a "tap on the shoulder" from the host that requests the Am29300 CPU take some previously agreed on action, such as reading a message from the host out of memory.

The two least significant interrupts are unused by hardware and are available for use as software interrupts. These interrupts would be set by the CPU writing into the Am29114 interrupt register.

The interrupt mode is set for capturing asynchronous low going pulses as interrupt signals. This is done because most of the interrupt signals are only guaranteed to be active for a single clock cycle. Therefore, the interrupts must be latched and held by the interrupt controller until acknowledged by the CPU.

The D_BUS is connected to the interrupt controller data pins so that the internal interrupt, mask, and in-service registers may be read and written.

The interrupt controller is selected and given instructions via outputs of the control pipeline register.

Interrupt Sequence

During a given clock, one of the interrupt inputs goes active. At the end of that cycle (active edge of clock), the interrupt signal is clocked into the interrupt register of the Am29114.

During the second clock cycle, the interrupt is ANDed with the interrupt mask register and, if the interrupt is allowed, its priority is compared to any currently in-service interrupt. If the new interrupt is of higher priority than any in-service interrupt, the MINTR* (interrupt request) will go active at the next active clock edge.

During the third clock cycle, the Am29114 interrupt request is externally ORed with the interrupt request from the trap logic. The combined interrupt request is then loaded into a delay flip flop. The delay flip flop is needed to synchronize the final interrupt request with the system clock. The reason for this is that the interrupt request from the Am29114 is stable too late (41 ns) in the third cycle to be useful in selecting an interrupt address. The set-up time for the microcode control store address could not be met if the Am29114 interrupt request were used directly with the Am29331 sequencer.

The external OR and delay functions are implemented in an AmPAL22V10A, whose logic is shown in Figure 5-12.

During the fourth clock cycle, the INTR* (interrupt request) input of the sequencer is driven by the delay flip flop. The sequencer then returns INTA* (interrupt acknowledge) if micro-interrupts are allowed. The INTA* signal enables the interrupt vector onto the microcode control store address lines.

The LSB three bits of the interrupt vector are provided by the Am29114 interrupt priority encoder. Bit 3 of the interrupt vector is provided by the trap logic. The bit is low for an interrupt and high for a trap vector. The upper bits (4:11) of the vector are provided by an external Am29818-1 register. This register provides a variable base address for a nine entry point table look-up (multi-way branch), which is based on the four bits of interrupt vector from the Am29114. The Am29818–1 register is loaded via the D_BUS or through the diagnostics SSR chain. The need for a nine entry point table is explained in the section on trap operation.

During the fifth clock cycle of the interrupt sequence, the first instruction of the interrupt routine will execute. During this cycle the interrupt return address will be pushed onto the sequencer stack.

In summary, from the time an interrupt signal becomes active until the interrupt service routine begins execution, four instructions in the main program will complete execution.

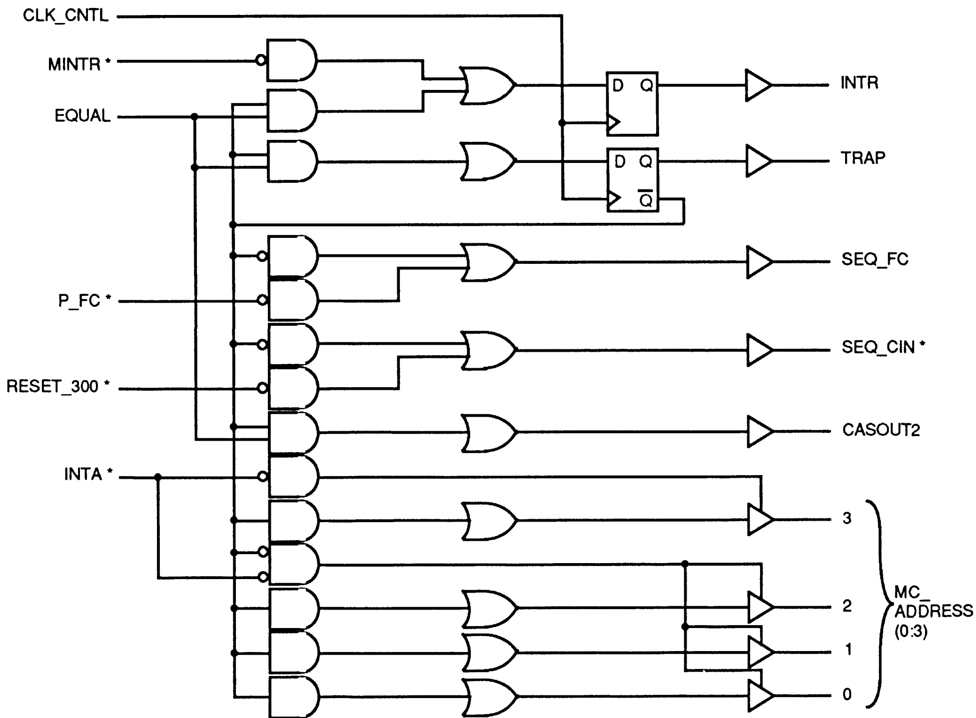


Figure 5-12. U75 AmPAL 22V10A Trap Logic PAL

Trap Operation

Trap Issues

A trap requires extremely fast response to the trap event signal.

The ideal situation is for the trap event signal to cause the abortion of the instruction in execution at the time the event signal appears.

This is extremely difficult in a high clock frequency system. To succeed, the trap event signal must be stable at least in time to prevent clocking of the data section of the CPU, which would otherwise change the system state (i.e., complete execution of the instruction). This implies that the trap event signal is stable one clock control circuit set-up time before the high to low edge of the system clock. The high-to-low edge of clock is significant, because once the clock signal falls, the writing of any write enabled port on the Am29334 register file will begin. In addition, the trap event signal must be stable in time to cause the Am29331 sequencer force continue (FC), interrupt request (INTR), and carry in (CIN*) signals to go high soon enough to disable the sequencer microprogram address in time to meet the set-up time requirements of the microcode control store.

In a 100 ns cycle time system, such as the one being discussed here, the trap event signal must be valid no later than 25 ns into the cycle. For a trap event signal that is to be derived from the effects of the instruction in execution in that cycle, this requirement is very difficult to meet.

Fortunately there are trap events that may be signalled on the one or two cycles previous to the cycle in which the trap must occur. Some examples would be: a cache miss that may be detected from the cache address created in a cycle prior to that in which the cache data is used in a calculation; or a breakpoint in which the breakpoint target instruction address is detected by the sequencer in the cycle prior to the instruction being loaded into the control pipeline for execution.

If a an instruction is a known potential trap, it is possible to execute the instruction so that no critical information is destroyed by completing its execution. This may be done by writing results back to a temporary register while allowing no other significant system state changes, such as updating the ALU Q register, or doing a return from procedure call. The instruction may then be allowed to execute and generate any trap event signals that might result from the execution, without concern for irrevocably destroying data because of some error condition.

In the above examples, the trap event signal may be loaded into a delay flip flop to synchronize the trap request with the beginning of the following cycle. This causes the trap operation to occur early in the cycle following the event and to complete successfully.

The only trap condition implemented in this design is the breakpoint.

Trap Logic

By definition, the response time between trap event signal and trap operation must be much faster than the four or more cycles that an interrupt takes to begin execution. This requires that the trap logic be different from the Am29114 interrupt controller. The trap logic design is implemented in an AmPAL22V10A. The logic is shown in Figure 5-12. The definition file for the PAL is shown in Appendix J.

The trap logic is in effect a simpler and faster interrupt controller. This "trap controller" is cascaded with the Am29114 interrupt controller so that the same address vector approach used with the interrupt controller may be extended to trap operations.

A trap is treated as a special form of interrupt with a higher priority. When a trap occurs, the trap logic generates a cascade out (CASOUT2) signal to the Am29114 to prevent any interrupt operation from beginning in the same cycle.

The trap logic also generates an INTR signal to the Am29331 sequencer. The INTR signal in turn causes the sequencer to three-state its microcode address outputs and return an INTA signal to the trap logic. The INTA signal enables a four bit vector from the trap logic and the interrupt base address from the Am29818-1 registers as shown in Figure 5-11.

The above steps essentially generate an interrupt and provide the interrupt vector. What makes a trap different is that the Trap Logic is also used to drive the Am29331 sequencer Force Continue and Carry-In inputs. This causes the sequencer to ignore the instruction being trapped and to perform a continue instruction instead, which changes no state in the sequencer. The CIN* signal's being high causes the trapped instruction address to not be incremented. Therefore, the trapped instruction's address will be loaded into the sequencer interrupt return address register. In addition, the TRAP signal is used to prevent any state change in the system other than in the sequencer, effectively aborting the trapped instruction.

Following are some other features to note in the trap logic.

Am29300 system RESET is used to generate the sequencer Carry-In signal (SEQ_CIN*). This is done to force SEQ_CIN* high during reset so that the first microcode instruction executed after reset will be at address zero rather than one.

In order for a trap operation to take effect, the instruction that is to be trapped must have its microcode interrupt enable bit active. This bit is used as the interrupt enable to the sequencer. If it is not active, then the microcode control store address from the sequencer will not be three-stated, and the interrupt vector will not be substituted. In addition, the TRAP signal will still occur, causing the trap target instruction not to execute correctly. Note that the interrupt enable bit could be externally forced active by the trap operation via an OR gate. But the added delay could cause the interrupt acknowledge to be too late to allow the interrupt vector address to meet required set-up times. (Of course, it is possible to design the system so that every trap causes all the system clocks to be stopped for one cycle. That would allow enough time for all kinds of tricks to be played. This design, however, will not explore that approach.)

MICROCODE CONTROL STORE AND CONTROL PIPELINE REGISTER

Control Store Function

The microcode control store is the high speed memory that contains the control bits comprising the instructions that the system may execute.

This system uses what is called "horizontal" microcode. Each microinstruction contains many control bits that manage a variety of different functions in parallel. "In parallel" is the key phrase. All the control information needed to manage the entire Am29300 system during the execution of one microinstruction is contained in one word of microcode control store.

The memory must be fast because its access time must be significantly shorter than the cycle time of the system. In general the access time must be less than half the cycle length. This is because of the time required by the sequencer to generate each new address to the control store, which takes up the remaining time in the cycle.

Pipeline Register Function

At the output of the microcode control store there is a register to hold the control information stable during the execution of an instruction. With the control information held in the pipeline register, the control section of the CPU is free to begin reading the next microinstruction from the control store. In this way, the control section is operating in parallel with the data section. The control section fetches the next instruction while the data section executes the current instruction. This parallel operation, where one section of the system works on one step of a problem while another section works on the next step, is called pipelining, hence the name for the pipeline register.

Through parallel operation, pipelining nearly doubles the speed of the system over what might be the case if the control section and data section were directly tied together in a serial fashion.

Control Store Implementation

Because this method of pipelining the output of a microcode store is so popular, there are special memories available that combine a high speed memory with a pipeline register at its output. These combined memory and pipeline devices may significantly reduce the system parts count.

These memories are available as either RAM or PROM devices. RAM versions are used to make writable control stores.

These memories also include Serial Shadow Registers (SSR) along with the pipeline register. This allows diagnostic routines to read and control the pipeline register outputs. Where RAM versions are used, the SSR is used as a built in means to load the writable control store.

This system is designed to use one of the following for control store: Am9151-50, 1K x 4 RAM; Am27S65, 1K x 4 PROM; Am27S75, 2K x 4 PROM; or Am27S85, 4K x 4 PROM. These devices all share a similar pinout so that simple jumper connections allow any of them to be placed in the same sockets.

The connections to the control store are shown in Figures 5-13 and 5-14.

A total of 23 memories are used to form the needed 92-bit-wide microcode words.

Because this system is designed to use no more than a 4K word deep control store, only the lower 12 bits of microcode address from the sequencer are connected.

The memories in the control store which provide the microcode branch field are connected differently from the remaining memories. This is because the branch field

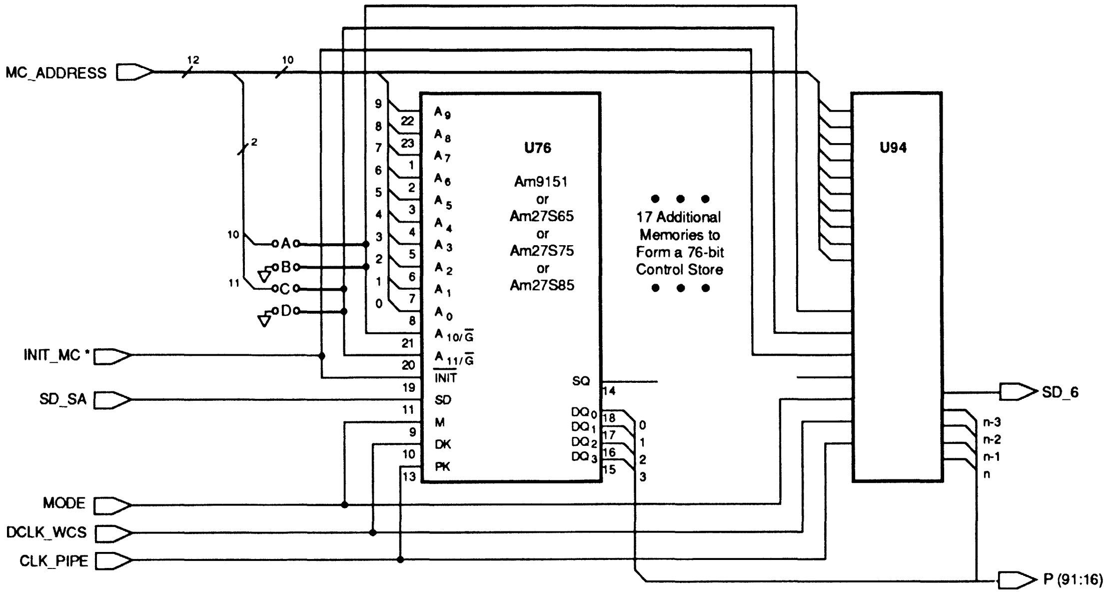


Figure 5-13. Microcode Control Store

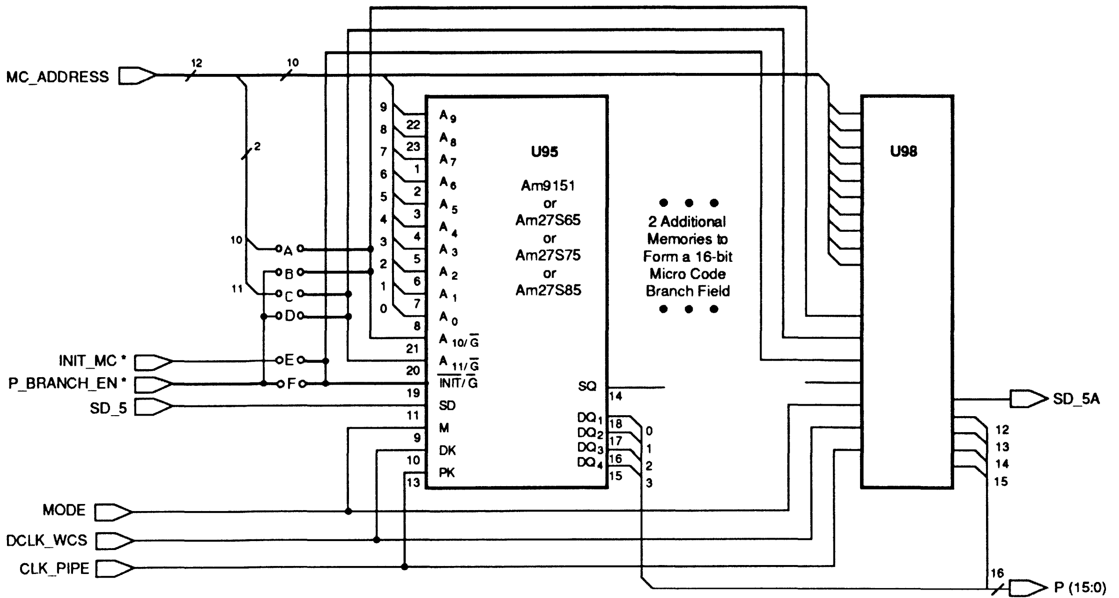


Figure 5-14. Microcode Control Store

outputs are connected to the D_BUS and must be three-stated when other devices drive the D_BUS. All the other outputs of the control store are always output enabled.

Figure 5-13 shows how the bulk of the control store is connected.

When the Am9151-50 or the Am27S65 is used, the jumper at location "B" is connected. This continuously enables the memory.

When the Am27S75 is used, the jumpers at locations A and D are connected. Also, the Am27S75 G/Gs* (pin 20) is internally programmed as an asynchronous enable. Those jumper connections will always enable the memory and connect address bit 10 to it.

When the Am27S85 is used, the jumpers at locations A and C are connected. The Am27S85 G/Gs/I/Is* (pin 19) is programmed as a synchronous initialize function. Those connections will always enable the memory and provide address bits 10 and 11 to it.

Figure 5-14 shows the connection for the memories that support the branch field.

When the Am9151-50 or the Am27S65 is used, the jumpers at location B and E are connected. This enables the memory when the control pipeline selects the control store to drive the D_BUS.

When the Am27S75 is used, the jumpers at locations A, D and E are connected. Also, the Am27S75 G/Gs* (pin 20) is internally programmed as an asynchronous enable. Those jumper connections will enable the memory when the control pipeline selects the control store to drive the D_BUS.

When the Am27S85 is used, the jumpers at locations A, C, and F are connected. The Am27S85 G/Gs/I/Is* (pin 19) is programmed as an asynchronous enable function. Those connections will enable the memory when the control pipeline selects the control store to drive the D_BUS. Also, these connections imply that when the Am27S85 is used, the branch field of the initialize word will not be valid.

CLOCK CONTROL

In almost every complex digital system there is a need to control and qualify selectively the system clock.

A register often needs a qualified clock that will clock (i.e., load) the register only when specified by some control signal. Sometimes a register will internally qualify its own clock by providing a load enable input. But most often, registers have only data input and outputs, an output enable, and an unqualified clock input. It is up to the system designer to provide a means to restrict the clock to the register so that it receives clock only on those cycles when its load enable control signal is active.

Restricting a clock in this fashion is referred to as qualifying a clock. The controlling signal that enables the qualified clock is called the qualifier.

Most synchronous digital systems have a system clock with a single active edge. This means that the system state will only change on either the low-to-high or high-to-low edge of the clock. The opposite transition of the clock will have no state changing effect in the system. The opposite transition of the clock is referred to as the inactive edge of the clock. It should be noted, however, that, even though there is a single active edge for the clocking of registered states in the system, the level of the clock may have an effect on some multiplexers or latches in the system. The level of the clock may control the path selected by a multiplexer, whether a latch is flow-through or held, or the write enable of a memory.

To qualify a clock, there must be a way to prevent the active edge from occurring. This implies that the clock is held either high or low when it is prevented from cycling. The choice of whether the clock will be stopped (held) at its high level or low level may depend on what, if any, effect the level of the clock has on system multiplexers, latches, or memories. For example, if the low level of the clock enables a memory write line, it may be preferred to stop the clock at the high level rather than the low level to prevent any change in state of the memory.

Clock Qualification Circuit

In the Am29300 system described here, the system clock will be stopped at the high level. This is because the low level of the clock may start the writing of data into the Am29334 register file. The active edge of the clock will be the low-to-high transition.

This method of qualifying clocks is referred to as 'OR' qualification. Usually with this method the free-running (unqualified) version of the system clock is 'ORed' with a low active enable signal. Thus, if the enable is active (low) the resulting qualified clock is allowed to track the free running clock. If the enable is inactive (high) the qualified clock will be forced high, stopping the clock, until the enable again goes active. Because the free running clock is always high during the first portion of each clock cycle, the clock enable signal need not be stable until just before the inactive edge of the free running clock.

In this Am29300 demonstration system the following are the desired controls over the system clocks:

1. The ability to stop all clocks to the Am29300 CPU, both control and data sections. This will suspend operation of (halt) the system.

2. The ability further to qualify register loading (register clocks) with control pipeline signals. The controlled registers would be the Macro Status, Macro Opcode, and Interrupt Base Address register.

3. The ability to single step all the system clocks when the system clocks are in the halt mode. Note this implies only conditional single stepping on those register clocks that are further qualified by load enable controls.

4. The ability to single step the data section or the control section independently.

5. The ability to force the control pipeline or the Macro Status, Macro Opcode, and Interrupt Base Address registers to load. This capability is used to implement diagnostic control over these registers.

To implement this kind of control over the system clocks, a separately qualified version of the system free running clock must be created for each differently handled register. The general clock for the control section is different from that for the data section. Also, each qualified register clock is different.

The block diagram for the clock qualification circuit is shown in Figure 5-15. The logic equation definition file for the PAL in this circuit is shown in Appendix K.

The qualifiers for the system clocks come from either the control pipeline, trap logic or the host interface controller. The AmPAL22V10A Programmable Array Logic (PAL) device is used to combine the various qualifiers into the appropriate clock enables for each differently handled set of registers. The output of the PAL is then logically ORed with the system free running clock to form the various qualified clocks in the system.

In this system, the free running clock generator produces an active low clock with the enables active high. By using negative logic OR gates (NAND gates) the clock and enable signals are logically ORed together to produce active high qualified clocks. The negative logic OR gates are external to the clock qualifier PALs.

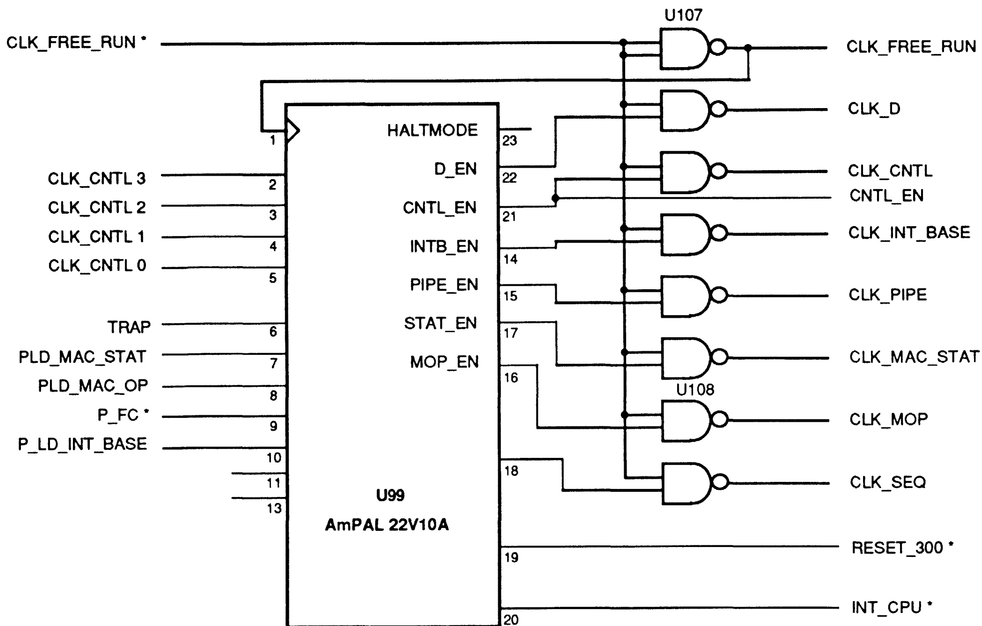


Figure 5-15. Clock Qualification Block Diagram

The NAND gates also serve as high output current buffers that allow the qualified clocks to drive many registers in the system. These NAND buffers also cause the clocks to have very high speed edges. This requires that clock lines be handled more carefully than other signal lines to help prevent noise, reflections, and ringing on the clock lines. Preventing these problems helps to ensure clean clock signals free from the glitches that may cause missed clocking or double clocking of registers. It is suggested that clock lines be routed serially, kept less than 12 inches in length, and terminated to the printed circuit board's characteristic impedance at the last point of use on each clock line.

Note that all the system clock lines, even the free-running clock line, pass through a NAND gate. This is done to equalize the delay of all clocks so that clock skew in the system is minimized.

Clock Generator

The unqualified (free running) source for all the clocks in the system comes from a clock generator implemented in an AmPAL16R6B. A diagram of the logic implemented in this PAL is shown in Figure 5–16. The logic equation definition file for this PAL is shown in Appendix L.

The only reason that a clock generator PAL is used in addition to a simple clock oscillator module is to provide the ability to vary dynamically the length of each system clock cycle. This ability allows the system to run at the maximum clock rate most of the time when the fastest data paths are in use and to run at a slower rate only when slower system data paths are in use. By slowing the system cycle time dynamically only when a slow data path is used, the average system speed is much higher than would be the case if the system clock rate were fixed at the rate required by the slowest data path.

A simple way to do this would be to divide the normal system clock by two and on each cycle select whether the normal length or the double length clock cycle would be used.

In this system, finer control over the length of each cycle is desired. Where the cycle need only be a little longer than usual, only a slightly longer cycle is used rather than doubling the cycle length.

This is done by dividing down a high speed clock, which runs three times faster than the normal system clock. It is then possible to extend a clock cycle in increments of the high speed clock. A cycle then may be 1, 1 1/3, 1 2/3, or 2 times the normal cycle length.

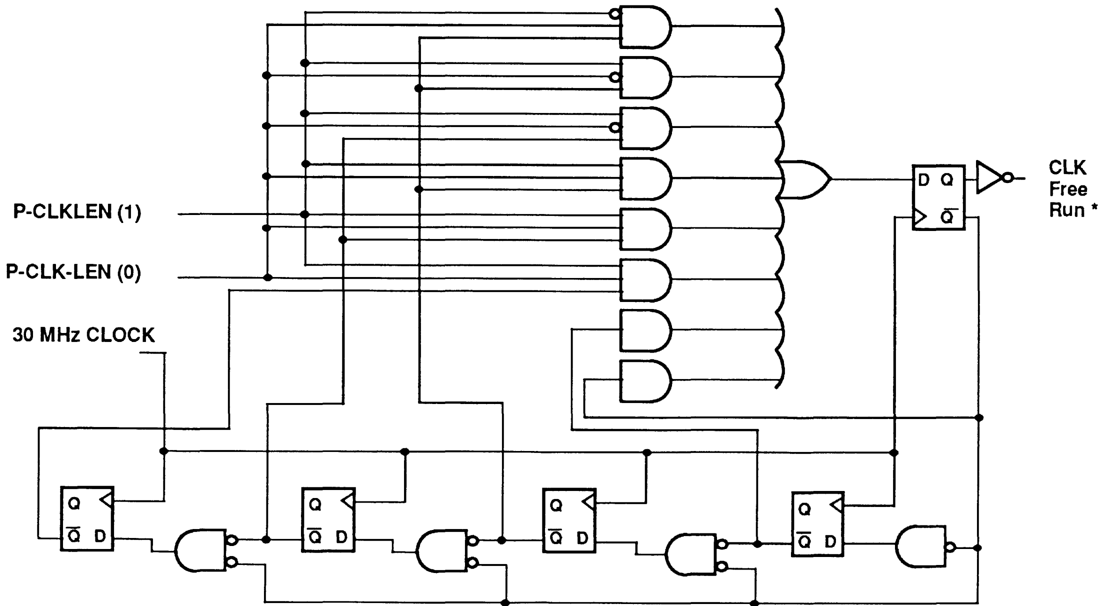


Figure 5-16. U100 AmPAL16R6B Clock Generator

The Am29300 demonstration system's normal clock is 10 MHz, or 100 ns, long. The high speed clock is then 30 MHz and is provided by a commercially available clock oscillator module.

The control over the cycle length comes from the control pipeline register and may thus be specified differently on each instruction. Two bits are provided to select one of the four cycle lengths. Each instruction may thus control its own cycle length based on the time required by the data paths that are used.

The waveform of the clock may be described in terms of the number of high speed clock periods during which it is active and then inactive.

Note that the output of the AmPAL16R6 is inverting. The logic internal to the PAL creates an "active high" clock with a low-to-high active edge. This waveform is inverted by the final output of the PAL and is later inverted once more in the clock qualifying circuit. The final system clocks are thus active high. When describing any system clock, it will be done in terms of an active high clock. The clock generator waveform is shown in Figure 5-17, where the outputs are shown active high, even though the actual PAL output is inverted.

Each clock cycle has two or more active periods followed by one inactive period.

The clock generator PAL output is from a D flip flop. When the flip flop output is inactive (low), one term feeds back the inverted output. This will force the flip flop high on the next high speed clock. The output of this flip flop feeds a shift chain of four other flip flops, which act as a simple timer for the extended cycle lengths.

During the first active period of the clock output, the output of the first flip flop in the timing chain is still inactive. This first flip flop's output is inverted and fed back into the clock output flip flop to force the clock output to remain high for a second active period.

During the second active period, the clock cycle length bits from the control pipeline become stable and determine whether additional active periods will be inserted into the output clock.

Note that since the first two periods of active clock are forced by the logic, the control bits need not be stable for two high speed clock periods minus the PAL set-up time (66.6 ns - 15 ns = 51.6 ns). This time margin is further reduced by the skew between the high speed clock and the qualified clock to the control pipeline which is equal to the clock-to-output time of the clock generator PAL plus the propagation delay of the qualifying NAND gate (51.6 ns - (10 ns + 5.5 ns) = 36.1 ns). Therefore, as long as the control pipeline register clock-to-output time does not exceed 36 ns, the clock generator will work as described here.

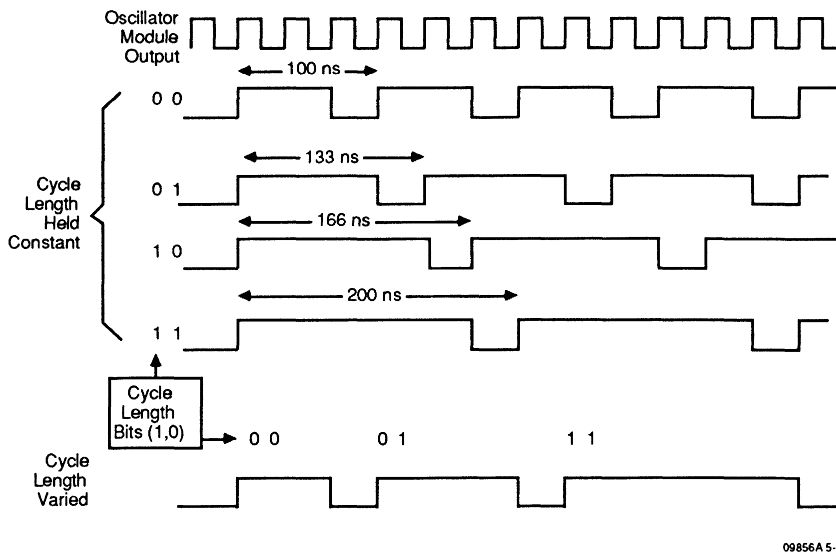


Figure 5-17. Clock Generator Outputs (Inverted)

If the clock cycle length bits are zero, no additional feedback terms are enabled and the clock output flip flop will go low in the next high speed clock period.

If the clock cycle length bits equal 1, the output of the second timing chain flip flop is fed back to the output flip flop to allow one additional active clock period.

Similarly, when the clock cycle length bits are equal to 2 or 3, an additional 2 or 3 active periods are inserted in the output clock waveform.

When the clock output flip flop again goes inactive, its output will force all of the timing chain flip flops to be cleared, thus beginning a new Am29300 clock cycle.

MICROCODE WORD

This section describes the structure and function of each field of bits in this system's microcode word. Included are some comments on how functions were determined and how they might vary in similar systems.

Control Philosophy

In a microprogrammed system, each word of the microcode functions as the determinate of all system action during one clock cycle of system operation. Each bit directly affects some aspect of the machine. Each field of bits may act independent of other fields to manage parallel data paths and simultaneous operations. This ability to manage parallel activities in each machine cycle gives a microprogrammed system high speed and flexibility. But the power of complete parallel control over nearly all the functions in a system comes at a cost.

The cost is wide control memory words. Fifty- to 150-bit-wide control words are common in microprogrammed systems. Three hundred-bit-wide control words have been used in large mainframe computers for years.

With each machine instruction's eating up 100 or more bits of memory, it doesn't take long to consume significant board space, power, and cost for high speed microcode memory.

The resulting dilemma between the need for parallel control and the cost, size, and power that accompanies it, is the basis of many a system designer's headache.

The usual approach used to strike a balance between the opposing issues is to determine carefully which functions must absolutely be able to occur in parallel, then to limit the microcode word size to that absolute minimum. Control over other less frequently used functions or over alternate operations is then overlapped with the primary control fields.

Overlapping of control fields means that during certain operations, the meaning of the bits in the overlapped control field changes. The hardware controlled by the primary meaning of an overlapped field must be disabled during the time that the alternative meaning is in effect. This of course means that the functions controlled by the overlapped fields cannot occur in the same machine cycle.

This results in winning a little and losing a little. More control and thus more functions may be managed with less control memory, but some operations then take multiple cycles to complete, due to the use of functions that may not be managed in one instruction. Also, the need to enable and disable control field meanings and the associated hardware, will add control bits and decoding logic. The decode logic adds delay into the machine cycles and will cause the system to run a little slower.

Additional savings in control word size may be made by encoding fields rather than having each bit directly drive a control signal. This again adds decoding logic and its associated delay.

The job of deciding what control must be parallel and what must be overlapped is more art than science. No matter how the microcode word is defined, there will always be other interesting ways to rearrange and overlap the control fields. Each way will cost something either in word width or control decoding, thus providing endless trade-offs.

All these possible variations make it extremely important to have a thorough understanding of the algorithms to be handled by a particular machine. The better the understanding, the better the chance to optimize the system architecture and control to solve the problem at hand.

Microcode Word Field Descriptions

Throughout the figures that detail the design of this system, signals that travel from page to page have been given meaningful names that imply the function of the signal. This helps in understanding what is going on in each figure. Many of these signals are the direct outputs of the control store pipeline register. As it turns out, many of the bits in the microcode carry multiple meanings because the function of several fields are overlapped to save microcode word size.

The result is that more that one signal name may often be associated with a particular bit of the control pipeline. Physically, of course, all signal lines that ultimately connect to a particular pipeline bit are one piece of wire. The logical separation of lines, by using different names, only helps to understand the function of a given signal, when the hardware that uses the signal is enabled. The following three Figures show the physical and logical relationships between the microcode control store bits and the signal names (meanings) that are attached.

Each Figure is split into pairs of columns preceded by one column that indicates the individual bit numbers for each signal. Each column pair contains a Field Name column that describes the function of the bit and a Signal Name column that gives the signal name used throughout the Figures in this document for that meaning. The left most column pair shows the primary meaning of the control bits. Other column pairs to the right give alternate (overlapped) meanings for the control bits along with the signal name used with each meaning.

Unless a control bit is overlapped with an alternate meaning in one of the columns to the right, the function of the control bit is constant.

Register File Controls

Figure 5-18 shows the microcode word bits that affect the Am29334 register file.

It was decided that a three address machine would be the most appropriate way to obtain the best performance from the Am29300 family components. Because of the common three bus architecture these parts share, a three address register file fits nicely. Two addresses are used to read an A and B operand from the file while the third address specifies an independent write location. This allows writing back results without requiring the destruction of one of the read operands in a single cycle.

An address multiplexer on the C operand register address does allow for two and one address operations by allowing either the A or B operand address to be used for the write operand address in addition to its use as a read operand.

Also, to support macroinstruction execution, address multiplexers are used on the read addresses so that macroprogram supplied register addresses may be directed to the register file. When macroprogram supplied addresses are in use, the meaning of the register address fields changes to control signals for the macro operand address counters. With this alternate meaning, the macro addresses may be incremented or decremented at the end of each cycle.

Bits 91 and 84 select whether the microcode or the macro opcode addresses are directed to the register file. If either bit is high, the alternate definition for the related address field takes effect, and the macro opcode address is used.

Bits 76 and 77 are used to select one of four addresses to be supplied to the A write port of the register file. The selections are as follows:

Bit 77	Bit 76	
0	0	C operand microcode address used.
0	1	A operand address, as specified by bit 91.
1	0	B operand address, as specified by bit 84.
1	1	C macro operand counter address used.

When any selection other than for the C operand microcode address is made, the field assumes the alternate meaning for control of the macro operand counter.

In addition to the three addresses used by the data section of the CPU, a fourth address is provided for the B write port of the register file so that data may be moved into the file via the second port while other calculations go on undisturbed.

The address for this fourth port comes from a multiplexer that may select either the C operand microcode address or the C macro opcode address counter as the source. Bit 69 is the select input for this fourth address multiplexer.

Bit 68 enables the register file A read port onto the A_BUS. If this bit is inactive and if the FPP seed register output is also inactive, the D_BUS to A_BUS transceiver is enabled so that constants, masks, and variables may be passed from the D_BUS to A_BUS.

Bits 67 and 66 are used as the write enable controls for the two write ports of the register file.

Data Path Controls

The data path controls are shown in Figure 5-19.

To provide a straightforward example of the usage of the PM and FPP, these devices have had their input and output buses paralleled with those of the ALU. In this arrangement it is not generally feasible to make use of more than one module in a given cycle. This is because the data buses may carry useful information to only one device at a time (this assumes that passing the same data to more than one device is of limited use). Also, only one device may drive the Y_BUS at a time.

Figure 5-18. Am29300 Demonstration System Microinstruction Word Layout -- Register File Controls

Control Pipeline Bit #	Field Name Primary Meaning	Signal Name Primary Meaning		Field Name Alternate 1 Meaning	Signal Name Alternate 1 Meaning	Field Name Alternate 2 Meaning	Signal Name Alternate 2 Meaning
P91	Reg A Macro/Micro*	P_ARA_MAC					
	If P91 = 0 then primary			If P91 = 1 then alternate 1			
P90	Register A Address (5)	P_RA	(5)				
P89	Register A Address (4)	P_RA	(4)				
P88	Register A Address (3)	P_RA	(3)				
P87	Register A Address (2)	P_RA	(2)				
P86	Register A Address (1)	P_RA	(1)	RA Count Direction	P_UP/DN_A		
P85	Register A Address (0)	P_RA	(0)	RA Count Enable	P_CNTA_EN		
P84	Reg B Macro/Micro*	P_ARB_MAC					
	If P84 = 0 then primary			If P84 = 1 then alternate 1			
P83	Register B Address (5)	P_RB	(5)				
P82	Register B Address (4)	P_RB	(4)				
P81	Register B Address (3)	P_RB	(3)				
P80	Register B Address (2)	P_RB	(2)				
P79	Register B Address (1)	P_RB	(1)	RB Count Direction	P_UP/DN_B		
P78	Register B Address (0)	P_RB	(0)	RB Count Enable	P_CNTB_EN		
P77	Reg C Add Source (1)	P_C_SEL	(1)				
P76	Reg C Add Source (0)	P_C_SEL	(0)				
	If P77:76 = 00 then primary			If P77:76 = 01, 10, 11 then alternate 1			
P75	Register C Address (5)	P_RC	(5)				
P74	Register C Address (4)	P_RC	(4)				
P73	Register C Address (3)	P_RC	(3)				
P72	Register C Address (2)	P_RC	(2)				
P71	Register C Address (1)	P_RC	(1)	RC Count Direction	P_UP/DN_C		
P70	Register C Address (0)	P_RC	(0)	RC Count Enable	P_CNTC_EN		
P69	B Write Port Select	P_AWB_MAC					
P68	A Bus Output Enable*	P_OEA*					
P67	A Port Write Enable*	P_WEA*					
P66	B Port Write Enable*	P_WEB*					

Figure 5-19. Am29300 Demonstration System Microinsturction Word Layout -- Data Path Controls

Control Pipeline Bit #	Field Name Primary Meaning	Signal Name Primary Meaning	Field Name Alternate 1 Meaning	Signal Name Alternate 1 Meaning	Field Name Alternate 2 Meaning	Signal Name Alternate 2 Meaning
P65	Data Path Select (1)	P_DPS (1)				
P64	Data Path Select (0)	P_DPS (0)				
ALU when P65:64 = 00			FPP when P65:64 = 10,11		PM when P65:64 = 01	
P63	ALU Instruction (8)	P_ALU_INST (8)	FPU Instruction (4)	P_FP_I (4)	TCX	P_TCX
P62	ALU Instruction (7)	P_ALU_INST (7)	FPU Instruction (3)	P_FP_I (3)	TCY	P_TCY
P61	ALU Instruction (6)	P_ALU_INST (6)	FPU Instruction (2)	P_FP_I (2)	ACC (1)	P_ACC (1)
P60	ALU Instruction (5)	P_ALU_INST (5)	FPU Instruction (1)	P_FP_I (1)	ACC (0)	P_ACC (0)
P59	ALU Instruction (4)	P_ALU_INST (4)	FPU Instruction (0)	P_FP_I (0)	RND	P_RND
P58	ALU Instruction (3)	P_ALU_INST (3)	ENR*	P_ENR*	XSEL	P_XSEL
P57	ALU Instruction (2)	P_ALU_INST (2)	ENS*	P_ENS*	YSEL	P_YSEL
P56	ALU Instruction (1)	P_ALU_INST (1)	ENF*	P_ENF*	TSEL	P_TSEL
P55	ALU Instruction (0)	P_ALU_INST (0)	Feed Through (1)	P_FP_FT (1)	ENXA*	P_ENXA*
P54	Position Mac/Mic*	P_POS_MAC	Feed Through (0)	P_FP_FT (0)	ENXB*	P_ENXB*
P53	Position (5)	P_POSITION (5)	IEEE/DEC*	P_IEEE/DEC*	ENYA*	P_ENYA*
P52	Position (4)	P_POSITION (4)	Seed Output Enable*	P_SEED_OE	ENYB*	P_ENYB*
P51	Position (3)	P_POSITION (3)	Projective/Affine	P_PROJ/AFF*	ENP*	P_ENP*
P50	Position (2)	P_POSITION (2)	Rounding Mode (1)	P_FP_RND (1)	ENT*	P_ENT*
P49	Position (1)	P_POSITION (1)	Rounding Mode (0)	P_FP_RND (0)	FA	P_FA
P48	Position (0)	P_POSITION (0)		FTX	P_FTX	
P47	Width Mac/Mic*	P_WID_MAC		FTY	P_FTY	
P46	Width (4)	P_Width (4)		FTP	P_FTP	
P45	Width (3)	P_Width (3)		PSEL (1)	P_PSEL (1)	
P44	Width (2)	P_Width (2)		PSEL (0)	P_PSEL (0)	
P43	Width (1)	P_Width (1)				
P42	Width (0)	P_Width (0)				
P41	Macro/Micro* Status	P_MIC/MAC				
P40	Register Status	P_REG_STAT				
P39	Load Macro Status	P_LD_MAC_STAT				
P38	Borrow Mode	P_BM				
P37	Memory Add Select (3)	P_MEM (3)				
P36	Memory Add Select (2)	P_MEM (2)				
P35	Memory Add Select (1)	P_MEM (1)				
P34	Memory Add Select (0)	P_MEM (0)				
P33	Memory Write En*	P_MEM_WR*				

Figure 5-20. Am29300 Demonstration System Microinstruction Word Layout -- Control Section Controls

Control Pipeline Bit #	Field Name Primary Meaning	Signal Name Primary Meaning	Field Name Alternate 1 Meaning	Signal Name Alternate 1 Meaning	Field Name Alternate 2 Meaning	Signal Name Alternate 2 Meaning
P32	Cycle Length (1)	P_CLK_LEN (1)				
P31	Cycle Length (0)	P_CLK_LEN (0)				
P30	Interrupt Enable	P_INT_EN				
P29	Force Continue	P_FC*				
	If P29 = 1 then primary		If P29 = 0 then alternate 1			
P28	Seq Instruction (5)	P_SEQ_INST (5)	Interrupt Host	P_INT_HOST		
P27	Seq Instruction (4)	P_SEQ_INST (4)	Sign Extend A_BUS	P_SIGN_EX		
P26	Seq Instruction (3)	P_SEQ_INST (3)	Initialize	P_INIT		
P25	Seq Instruction (2)	P_SEQ_INST (2)	Load Interrupt Base Add	P_LD_INT_BASE		
P24	Seq Instruction (1)	P_SEQ_INST (1)				
P23	Seq Instruction (0)	P_SEQ_INST (0)				
	If P29 = 1 AND P28:27 != 11 then primary		If P29 = 0 OR P28:27 = 11 then alternate 1			
P22	Test Select (3)	P_TEST (3)	Am29114 Instruction (3)	P_INT_INST (3)		
P21	Test Select (2)	P_TEST (2)	Am29114 Instruction (2)	P_INT_INST (2)		
P20	Test Select (1)	P_TEST (1)	Am29114 Instruction (1)	P_INT_INST (1)		
P19	Test Select (0)	P_TEST (0)	Am29114 Instruction (0)	P_INT_INST (0)		
P18	Load Operand Counter	P_LD_CNT				
P17	Load Macro Op Reg	P_LD_MAC_OP				
P16	Branch Field Enable*	P_BRANCH_EN*				
P15	Branch Address (15)	D_BUS (15)				
P14	Branch Address (14)	D_BUS (14)				
P13	Branch Address (13)	D_BUS (13)				
P12	Branch Address (12)	D_BUS (12)				
P11	Branch Address (11)	D_BUS (11)				
P10	Branch Address (10)	D_BUS (10)				
P 9	Branch Address (9)	D_BUS (9)				
P 8	Branch Address (8)	D_BUS (8)				
P 7	Branch Address (7)	D_BUS (7)				
P 6	Branch Address (6)	D_BUS (6)				
P 5	Branch Address (5)	D_BUS (5)				
P 4	Branch Address (4)	D_BUS (4)				
P 3	Branch Address (3)	D_BUS (3)				
P 2	Branch Address (2)	D_BUS (2)				
P 1	Branch Address (1)	D_BUS (1)				
P 0	Branch Address (0)	D_BUS (0)				

If separate control bits were provided for the FPP or PM, they could perform multi-cycle operations such as Newton-Raphson division in the FPP or greater than 32 by 32 bit multiplies in the PM, while remaining detached from the input and output buses during most of the multi-cycle operation. If this were done, the ALU could operate in parallel during such operations. The cost of doing this would be an additional 15 to 35 bits added to the microcode word width. These bits would get full use only during those situations that parallel calculations are possible.

For this design it was decided to use a smaller microcode word by overlapping control bits for each of the three functional units.

Data Path Selection: Only one functional unit (data path) in the data section is chosen in any one cycle. Bits 65 and 64 select one of four options:

Bit 65	Bit 64	
0	0	ALU enabled
0	1	PM enabled
1	0	FPP enabled
1	1	Special function

In the special function option, the FPP is enabled for calculation and the control bits are assumed to be set correctly for use by the FPP, but the output enable of the FPP is inactive with the ALU output enable active. The ALU is not enabled for calculation in the sense that its hold input is made active to prevent state change in the status or Q registers.

This odd-looking combination is used to provide input operand parity checking for the FPP. The FPP does not have its own parity checking circuits, so with this arrangement the ALU parity checkers will be enabled by the active output enable on the ALU. The FPP is still allowed to function and may complete its operation and store the result in its internal registers, while in the same cycle the input operand parity is checked by the ALU. The ALU state is left undisturbed by this operation.

How useful is this scheme? It may save a cycle once in a while, but mainly it illustrates the odd sort of opportunities one may find to use up an otherwise wasted control code.

ALU Path: When the data path select bits enable the ALU meaning for bits 63:38, bits 54 and 47 are used to select either the microcode or macroinstruction position and width fields. The macro supplied information is selected when these select bits are high. When the macro source is selected, the microcode position and width fields are unused.

Bit 41 selects macro or micro status inputs for the ALU. Bit 40 selects whether the status output of the ALU is flow-through or registered.

Bit 39 is used as a clock qualifier for the loading of the ALU external macro status register.

Bit 38 directly controls the Borrow mode of the ALU.

FPP Path: When the data path selects enable the FPP, the control bits shown directly manage the operation of the FPP as described by the Am29325 data sheet. Bit 52 is used to enable the output of the FPP external "division seed" registered PROM.

PM Path: When the data path selects enable the PM, the listed control bits are used as defined in the Am29C323 data sheet.

Data Path Enabling: What does it mean to enable or disable one of the functional units? The control bits that are shared between each functional unit are either high or low every cycle, and they are connected to the ALU and multipliers all the time. There is no intervening logic that turns all the control bits "off" when a particular path is not selected. Each device sees a jumble of nonsense on its control lines whenever the control field meaning is intended for another device. Nonsense or not, each device will do whatever the control bits specify.

Enabling a data path means making the output enable of the selected device active so that it drives the Y_BUS and is able to write calculation results back into the register file. In the case of the ALU, enabling also means that the ALU hold input will be made inactive so that state change of the ALU status and Q registers is allowed. Enabling one path implies disabling the other paths.

For the PM and FPP, disabling means their output enables are inactive. It also means that the PM product register feed through pin is disabled by the control decode logic. For the FPP it means that both of its register feed through lines are disabled by control decode logic. These register feed through controls are disabled because, if they are allowed to be active, it is possible for the PM and FPP multipliers to feedback on themselves and begin to oscillate. This action would not damage the devices, but it could add to power consumption and system power plane noise. A simple prevention is just to disable the feed-throughs when the data paths are not selected. Note that the ALU has no internal feedback paths and does not need any similar treatment.

Memory Control: Bits 37:33 are available at all times to control the Am29300 system memory.

Bit 33 is the memory write enable control.

Bits 35:34 select the source of the address for the memory.

Bit		
35	**34**	
0	0	No memory address or operation is selected
0	1	A_BUS data is used to address memory
1	0	The A memory address counter is selected for address
1	1	The B memory address counter is selected for address

Bits 37:36 select the following:

Bit		
37	36	
0	0	Load counter A
0	1	Load counter B
1	0	Selected counter is incremented
1	1	Selected counter is decremented

The increment and decrement commands have effect only when a counter is selected as the MA_BUS source. The load commands have effect only when the A_BUS is the selected source.

Control Section Controls

Figure 5-20 shows the bit definitions for the control section.

Pipeline bits 32:31 control the length of each machine cycle.

Bit		
32	31	
0	0	Normal cycle length
0	1	1.33 x Normal cycle length
1	0	1.66 x Normal cycle length
1	1	2 x Normal cycle length

Bit 30 enables sequencer interrupts on a cycle by cycle basis.

Bit 29 is the Force Continue signal for the sequencer. When this bit is active, the sequencer will execute a continue instruction regardless of the state of the sequencer instruction or test select lines. This effectively enables the alternate meaning for the sequencer instruction and test select fields.

Bits 28:19 are normally the sequencer instruction and test select inputs. When Force Continue is active, the sequencer instruction field meaning changes.

When Force Continue is active, bits 28:25 are used to control four individual functions. Bit 28 will send an interrupt signal to the host system. Bit 27 will enable the sign extension of data going from the D_BUS to the A_BUS. Bit 26 will force the control pipeline register to load data from the control store initialize register at the next active system clock. Bit 25 will enable the loading of the interrupt base address register.

Bits 22:19 are used to control the sequencer test selection. When an unconditional sequencer instruction is in effect or when the Force Continue bit is active, bits 22:19 are used to control the Interrupt controller instruction.

Bit 18 is used to load the macro operand counters from the macro opcode register.

Bit 17 is used to load the macro opcode register.

Bit 16 enables the three-state outputs on the branch field bits of the control pipeline register. If these outputs are disabled, then the sequencer, A_BUS to D_BUS transceiver, or Interrupt Controller may drive the D_BUS. How a device is chosen to drive the D_BUS is explained in the control decode logic description. It is only important to note that if bit 16 is active, the branch field outputs will be active and will have priority over any other driver on the D_BUS.

Bits 15:0 are the branch address field to the sequencer. This field is also used to contain constants or masks. These may be used by the data section, sequencer, interrupt base register, or interrupt controller. It is a full 16 bits long in order to allow for constants or masks that fill half of the 32-bit data path. This allows 32-bit microcode supplied masks to be formed with two microinstructions.

Alternate Arrangements

The microcode word size just defined for this system totals 92 bits wide. Having so many bits allows the flexibility to change the control over most of the machine's functions on any or every cycle. But, this degree of control flexibility is not required for every application. The size of the control store may be reduced based on how the system is used most often. Following are a few comments on ways to rearrange and reduce the control store size.

Current Control Bit Usage

First let's look at how the control bits are used in this design.

Seven of the bits are used to control the selection of alternate field meanings (i.e., overlap control in bits 91, 84, 77:76, 65:64, and 29).

Eleven bits are used to control functions that are desired to operate in all cycles, independent of other system operations. These are the register file write and read enables (bits 69:66), memory controls (bits 37:33), and the cycle length controls (bits 32:31).

Eight bits generally do not change state frequently. Their existence in this design is a convenience that reduces the need for control decode logic and adds system flexibility. These bits are 41:38, 30, 18:16.

Three bit fields are used only with some instruction types. These are the position, width, and branch fields. Whenever a particular instruction does not use a field, those bits in the field are currently wasted in that instruction cycle.

Alternative Usage

The bits that change infrequently could be replaced by decode logic that provides these same control signals via set-reset flip flops. The flip flops would be controlled by overlapping set and reset commands with some other control store field. This would add to the decode logic complexity and would limit when the flip flops could be changed by restricting the control over them to certain instruction types. Since they change only infrequently, the requirement to use certain instruction types when setting or resetting them should not be a problem.

Those bit fields that are limited to certain instruction types could be overlapped. An example might be to overlap the position and width fields with the branch address field. This would restrict branches to instructions that do not require the position or width information.

When alternative field meanings are enabled, often the alternative definition does not make use of all the bits in the field. This presents the opportunity to overlap other control bits that may be valid in the same cycle as the alternate meaning of the field.

For example, some of the infrequently-used control bits could be overlapped with the unused bits of the register C address when the primary meaning of the C address field is not active. When a two address instruction is executed, the address for the C register comes from the A or B address, thus leaving the microcode field for the C register address available for other functions.

In another example, the bits in the position and width fields that are not used by the PM or FPP could be overlapped with other control functions when the alternate meanings for the field are in effect. An alternate branch address field might be placed in those bits to allow branch instructions in combination with FPP or PM operations without the need for the currently defined branch field.

Careful analysis of how each data path is used may also allow reductions through the elimination of controls that are not needed. As an example: if the PM were used only in flow through mode, all the controls for register enables, flow through modes, and input multiplexers could be removed from the microcode word and those inputs to the PM tied to fixed voltage levels. If only two's complement mode is used then an additional two bits may be eliminated. This would leave only four necessary control bits, the accumulator controls, rounding mode, and format adjust. This reduction might allow PM operations to be overlapped with some multiply-accumulate operations in the FPP.

By combining these reduction techniques, the following changes could be made:

All of the eight infrequently used control bits could be moved to overlap with the C register address, with half in effect when the A address is substituted for the C address and half in effect when the B address is substituted.

The PM controls, except for flow though and two's complement mode, may be moved to overlap with the position, width, and memory control fields. Also, the fourth data path select field may be changed to disable the memory controls and select the ALU — minus the position and width fields — to be active along with the PM. In this mode the PM flow through and two's complement mode controls would be fixed with no flow through and two's complement mode active. The ALU position and width inputs would be set to 0 and 31 respectively by control decode logic (unless these fields were selected to come from the macro opcode).

The branch address field may be moved to overlap with the position, width, and memory control fields. When ever the sequencer instruction selects a branch operation, the position, width, and memory fields are disabled and the branch address meaning substituted.

If all of these changes are made, the currently defined branch address field and infrequently used control bits may be eliminated, which would save 24 bits of microcode word width. This would reduce the word size to 68 bits.

This savings would come at the cost of allowing branch instructions only when the ALU instruction does not need position or width information from the microcode (this information may still come from the macro opcode register) and when the system memory is not being used. Further, a PM operation could not occur with a memory access in the same cycle. Also, with these changes it would be possible to control the ALU and PM concurrently when the ALU does not need position or width information and when the PM operates on internally registered data.

There are many such combinations of microcode control field definition. Each one provides a different trade-off between word size and what operations may be concurrent. Each one requires a different degree of complexity in the control decode logic.

CONTROL DECODE

What Is It Good For?

The ideal microprogrammed system has a separate microcode control store bit for each control input that exists in the system. This kind of complete control over every aspect of the system directly from the control pipeline totally eliminates the need for decoding the meaning of any system control bits. It also requires a very large microcode word to manage most useful systems. So in the real world, most microprogrammed systems encode or overlap at least some control functions in the microcode word.

Encoded control or not, each control input in the system requires valid voltage levels during each machine cycle if the system is to operate as expected.

The control decode logic acts as the bridge between encoded or overlapped (i.e., sometimes unavailable) microcode control fields and the related control signals in the system. The control decode logic continuously provides valid logic levels for those control signals that cannot be directly driven by the control pipeline register.

If the control field for a particular function is encoded, the control logic translates the function codes into individual control signals. Where control fields are overlapped, the control logic may be used to disable logic affected by a control field when that field has a meaning different than that intended for the logic being disabled (i.e., when overlapped control is active).

In some cases, control logic is used to prevent harmful conflicts between the meaning of different control bits, for example when two separate control fields affect the three-state enables on different buffers which may drive the same signal line. Certain combinations of control bits might enable both buffers in the same cycle causing contention between the buffers. Allowed to continue for long periods, this kind of contention may destroy the buffers. Control logic may be used in this situation to disable one or both buffers when the combination of controls affecting them would otherwise cause damage. In fact it is strongly recommended that this kind of problem always be avoided by designing the control decode logic to prevent such disasters. The alternative is to watch hardware melt because of a software mistake.

Control Logic Description

Some of the control logic function in this demonstration system has been distributed into the devices being controlled. This is done when a PAL is used to implement a function. A PAL generally has excess inputs and internal logic that may be put to use in decoding the meaning of encoded control fields(e.g. the memory address counters). The memory address counters are implemented from AmPAL22V10 devices and are shown in Figure 4-7. The control for loading, incrementing, decrementing, and output enabling the counters is provided directly from the encoded memory control field. The PALs internally decode the meaning of the control bits.

When a device requires a decoded control signal, the signal must come from control decode logic that takes control pipeline bits as input and produces the needed control signal. In this system, the required control logic has been implemented in three AmPAL18P8B PALs. These PALs are fast to minimize the delay induced between the control pipeline register and the device controlled. The PALs also provide the convenience of having programmable output levels, either high or low active for each output, independent of other outputs.

The block diagram for these PALs is shown in Figures 5-21 and 5-22. The logic definition files for these PALs are in Appendix M.

The ALU output enable, ALU hold, and PM output enable are all direct results of the pipeline data path select bits.

The pipeline controls for seed register output enable, PM flow through, and FPP flow through are gated by the appropriate data path selection so that each control signal is active only when the related data path is selected.

The D_BUS to A_BUS direction of the D_BUS transceiver is enabled by the register file A output's being disabled in conjunction with the seed register output's being disabled.

The A_BUS to MD_BUS buffer is enabled by certain codes of the memory control field.

The control store initialize register select is enabled by the combination of the pipeline Force Continue and the pipeline control bit for the initialize select. It is also enabled by the WCS_INIT* signal from the host interface controller. Note that the initialize control is synchronous as used in this system so that the initialize word is loaded only at the next active clock.

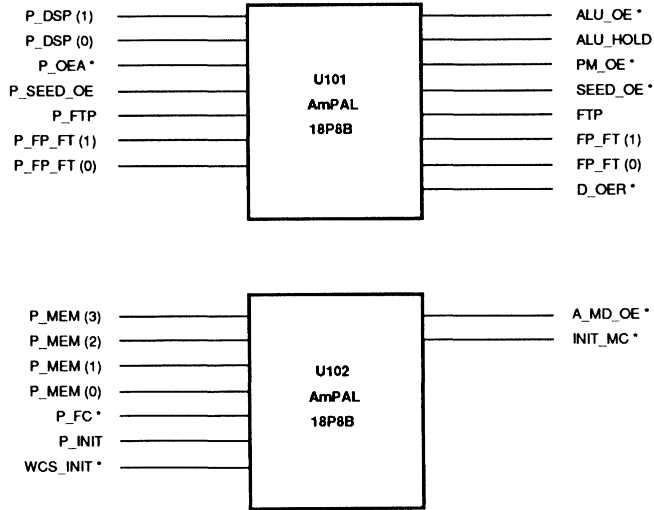


Figure 5-21. Control Decode Logic Part1

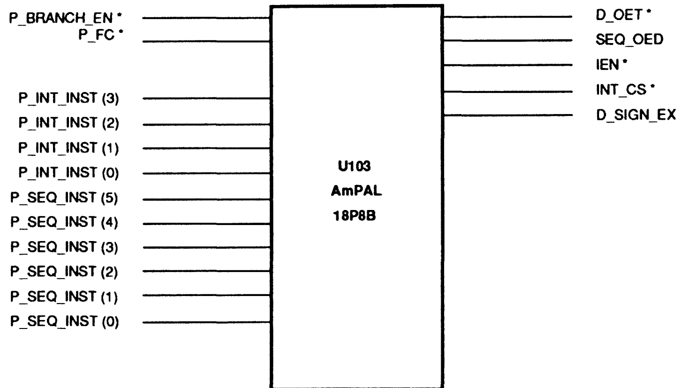


Figure 5-22. Control Decode Logic Part 2

The D_BUS sign extend, Sequencer output enable, Interrupt controller instruction and chip select enables, and A_BUS to D_BUS enable are all direct results of the pipeline sequencer instruction, interrupt controller instruction, branch enable, and Force Continue bits.

The Sequencer output enable, A_BUS to D_BUS enable, and interrupt controller chip select are used to control which device is allowed to drive the D_BUS in any given cycle. These output enables are arranged in a priority with only one output allowed to be active in any cycle; including the branch field of the control pipeline.

The highest priority output is the branch field. If it is enabled all other outputs are disabled.

If the branch field is disabled, then the Sequencer D output is enabled if either a Continue or a Pop D instruction is being executed.

If neither the branch field nor the sequencer are enabled, then the interrupt controller may drive the D bus if the interrupt controller instruction is one of three read operations.

If none of the above conditions exist to enable the other D_BUS devices, then the A_BUS to D_BUS transceive path is enabled.

Note that the interrupt controller chip select is treated as both an instruction enable and as an output disable. The chip select is active whenever there is a valid interrupt instruction that would not cause a conflict with another driver of the D_BUS. This means that when there is a valid instruction, the chip select will be inactive only if a read instruction is selected and either the branch field or sequencer are already enabled on the D_BUS. If any other interrupt instruction is in effect, the interrupt controller does not drive its outputs.

The above scheme for managing the access rights to the D_BUS may seem a bit complex but it allows great flexibility in movement of information over the D_BUS. Information may be moved between the interrupt controller and sequencer, interrupt controller and A_BUS, or sequencer and A_BUS. Information may be loaded into the interrupt base address register from the pipeline, sequencer, or A_BUS. Also, the pipeline may provide data to the sequencer, interrupt controller, interrupt base address register, or A_BUS.

SECTION 6
System Timing and Critical Path Analysis

DEFINITIONS

The upper limit on system speed is set by the slowest signal propagation path in the system.

The length of a signal propagation path is measured from the output of one register to the input of another register, where all registers are loaded by the same clock.

The slowest signal path will be different for different control states. An example would be the selection of the ALU data path vs. the FPP data path.

A signal path may be slower in the first cycle that control selects the path than it will be in a subsequent cycle that maintains the same path selection. This can be due to three-state enable or disable times being longer than normal propagation delays of the circuits involved.

CONTROL AND DATA PATHS

In determining the maximum system speed, every signal path must be analyzed. This requires tracing every control signal and every data signal and totaling the delay induced by each component along the path from source register to destination register. Where parallel paths exist, the time delay for the slowest path is used.

Most often, the critical (slowest) paths originate with the pipeline control register. In the data section the paths will end with data being loaded into the register file, an FPP or PM internal register, the system memory, or a D_BUS destination. In the control section the paths will end with loading of new control bits into the control pipeline register.

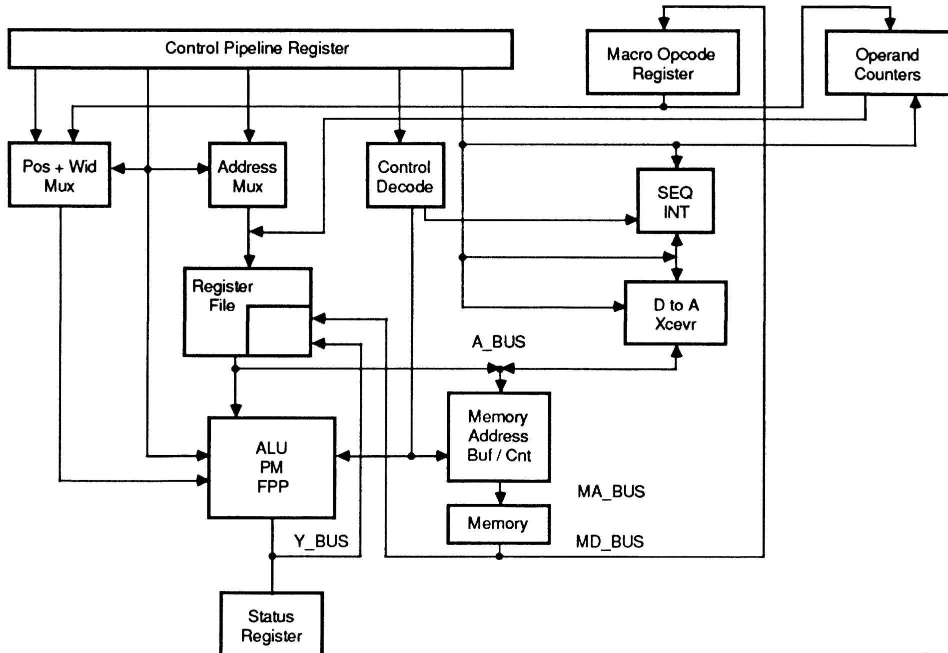


Figure 6-1. Data Section Timing Paths

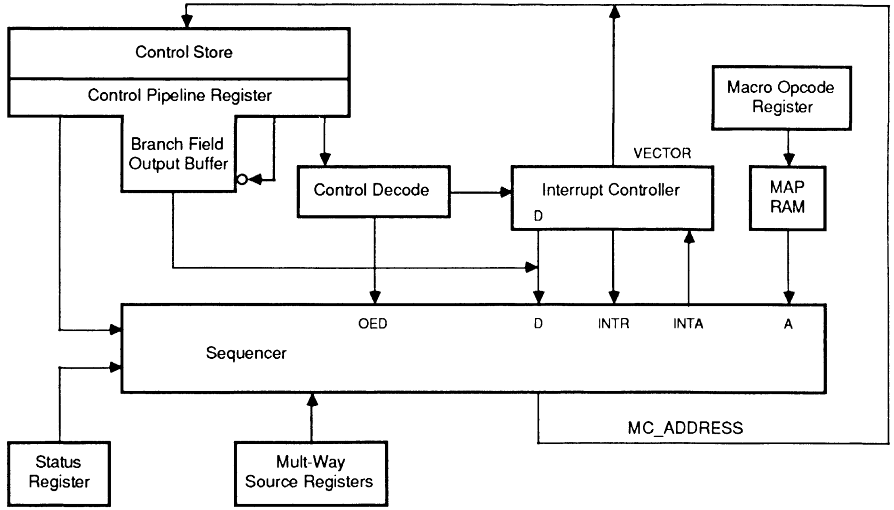


Figure 6-2. Control Section Timing Paths

Since the control section and data section operate in parallel, the slowest path in either section will determine the cycle length required for a specific operation.

Figures 6-1 and 6-2 provide a block diagram view of significant signal pathways for both control and data lines in both the control and data sections.

Referring to these figures as critical timing paths are discussed may help in following the timing analysis.

In this and nearly any complex system, there are hundreds of pathways that must be traced in order to ensure finding all the worst case delays. To go through all of them here would require too much time and space. Many of the timing paths for this design have already been analyzed, and what appear to be the worst case paths will be shown here.

WORST CASE PATHS

Each case is shown in Table 6-1. The table is separated into several pages due to its length. It can be viewed as a long spreadsheet calculation in which the appropriate timing parameters that apply to each case have been selected and placed in the correct column. Only the worst case delay for each segment of a critical path is shown. Parallel but faster paths have been eliminated from each case so that the total of the times listed for a case represents the minimum time in which a path can be traveled.

Case Definitions

1. Basic flow-through calculation, data path.

   Data is moved from the register file through the ALU and back to the register file. The timing path begins at the control pipeline where the register file address for the A and B read operands appear after the clock to output delay of the control pipeline register. These addresses flow through the Am29827 buffer that forms one side of the register file address multiplexer. The address accesses the register file and one access time later the data operands are presented to the ALU. By this time the control signals for the ALU instruction have been stable long enough that the flow through time of the data in the ALU will be the slower path. Once data is on the Y bus the last delay is the set-up time for the register file before clock can occur. Again, the control path to the register file (A port write address) is faster than the data path so the data path is the limiting factor.

   The total delay for this path is 96 ns. If the PM path is substituted for the ALU the delay would be 174 ns. If the FPP were substituted, the delay is 179 ns. So flow through calculations with either of the multipliers will require extended cycle length.

2. Basic flow-through calculation, position control path.

   This case is the same as Case 1 except that a careful look at the control path for the position input to the ALU is taken. This path turns out to be 97 ns worst case. This is an example where the control path is a little slower than the data path.

3. Flow-through calculation with address supplied by the Macro operand counter; counter output enabled same cycle.

   Again this path is similar to Case 1. The difference is that the read addresses are assumed to come from the Macro operand counters. It is further assumed that the counters are selected during the cycle analyzed. This means that the output enable time of the counter must be added to the clock to output time for the pipeline bit that selects the macro opcode counter.

   This increases the delay path to 115 ns, indicating that during the first cycle, in which a macro opcode counter is used as the address source, the cycle length will need to be extended.

4. Flow-through calculation with address supplied by the Macro operand counter; counter output enabled prior cycle.

   This case is a comparison with Case 3, where the Macro operand counter was output enabled in the previous cycle. The counter delay is thus limited to the clock to output delay of the counter. This reduces the cycle time requirement to 90 ns. So, sequential register file address cycles, using an operand counter can be completed within the normal cycle time.

5. First cycle of FPP Newton-Raphson division, seed value load.

   In this case the critical path starts at the control pipeline clock to output delay, and then goes through the control decode logic that enables the output of the Seed register. In this case it is assumed that the seed value is multiplied and stored in an FPP internal register to complete the first cycle of a Newton-Raphson division. This requires a total of 169 ns. Note that if the seed value had simply been moved into the input register of the FPP, the total delay would have only been 73 ns.

6. Memory read with address from the register file, selected by microcode.

   This is a simple memory read with the time starting at the pipeline clock to output delay, followed by the address mux, register file access, A_BUS to MA_BUS buffer, memory, and register file data set-up time. The total time comes in at 99 ns, just under the desired 100 ns basic cycle time.

7. Memory read with address from a memory address counter.

   Here the access time of the register file is essentially traded for the output enable time of a memory address counter. The total delay only improves to 94 ns, but there is a big advantage in the fact that for a sequential access the CPU did not need to calculate a memory address. This will save at least one cycle. Also, it is possible to overlap a memory read from an address counter with a calculation cycle in the CPU.

8. Memory write with data from register file, selected by operand counter.

   In a memory write case, time is saved by needing only to meet the data set-up time of the memory rather than the memory access time plus the register file set-up time, as would be the case in a read operation. In this case the time gained is traded for the time required to output enable an operand counter. Even so, the total time is still 94 ns.

9. Move register file data to interrupt controller or sequencer, data selected by operand counter.

   Here again, the long delay path of using a macro opcode counter as the register file address source is used. Even with the output enable delay of the counter in addition to the pipeline clock to output time, the total delay comes in at 89 ns.

10. Move sequencer or interrupt controller data to register file.

    In the reverse of the above case, the time to get data from D_BUS is similar to the time in Case 9 to access data from the register file. The big delay here is the need to move the data from the A_BUS, through the ALU and back to the register file. Not having a direct path to the Y_BUS has cost a good bit of time. The total comes in at 127 ns. Fortunately this type of data move is not likely to be a commonly executed cycle.

11. Sequencer branch, conditional or unconditional.

    In this case much of the delay is in the pipeline clock to output time for the branch field enable bit, cascaded with the output enable time of the branch field in the control pipeline register. This is followed by the branch address flow through time of the sequencer and the access time of the control store. Even with all the delay, this path is significantly faster than most of the data section paths. The total time is 84 ns.

12. Sequencer interrupt or trap cycle.

    In this case the pipeline output doesn't turn out to be in the main delay path. The interrupt starts at the clock to output delay of the trap logic where the interrupt request is generated. The sequencer then responds with interrupt acknowledge, which in turn output enables bit 3 of the interrupt vector from the trap logic. The interrupt vector then accesses the control store. The total for this cycle is 81 ns.

13. Sequencer branch to macro opcode specified instruction.

    Here the initial delay is the clock to output delay of the macro opcode register, followed by the access time of the map RAM. Next is the branch flow through time for the sequencer and the access of the control store. This cycle comes in at 85 ns.

FINAL RESULTS

Several cases were shown here to help give an idea of how fast the system is for different instructions. These cases were some of the worst identified during the critical analysis of this design. All but three of the cases shown fit within the desired 100 ns basic clock cycle. Two of the cases would only require a cycle 1 1/3 times normal. Case 5 officially needs a double length cycle.

As noted in the discussion of Case 1, both the PM and FPP require much longer cycles for flow through calculations. If the PM and FPP are used in clocked multiply mode for sequential pipelined multiplies, as would occur in array calculations, the cycle time can be significantly reduced. In clocked multiply mode the PM or the FPP requires only 100 ns cycle times.

With a dynamically variable clock cycle length, this system can run most instructions at the basic 100 ns cycle rate, but will still handle the occasional extended execution time instructions.

Am29300 Demonstration System **Signal Path Timing Analysis**

Table 6-1A

Data Path Element Parameter Description	Symbol	Value	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8	Case 9	Case 10	Case 11	Case 12	Case 13
	Worst Case Time Delay in Nanoseconds, Over Commercial Operating Range														
Control Store/Register - Am9151-50															
Clock to Output	Tpkhdqv1	15	15	15	15		15	15	15	15	15	15	15		
OE to Output Valid	Tgldqv	20											20		
Synchronous!															
I to Clock Set-up	Tivpkh	25													
Address to Clock Set-up	Tavpkh	30											30	30	30
Control Decode Logic - AmPAL18P8B															
Input to Output	Tpd	15					15					15			
Macro Opcode Register - Am29818-1															
Clock to Output	Tpd	11													11
Input to Clock Set-up	Ts	6													
Macro Operand Counters - AmPAL22V10A															
Clock to Output	Tco	15				15									
Input to Clock Set-up	Ts	20													
OE to Output Valid	Tea, Ter	25			25					25	25				
Reg File A or B Read Add Mux - Am29827A															
Input to Output	Tph	6	6					6							
OE to Output Valid	Tzh	10													
Reg File C Write Add Mux - AmPAL18P8Q															
Input to Output	Tpd	35													

Table 6-1B

Data Path Element Parameter Description	Worst Case Time Delay in Nanoseconds, Over Commercial Operating Range														
	Symbol	Value	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8	Case 9	Case 10	Case 11	Case 12	Case 13
Reg File B Write Add Mux - AmPAL22P10AL															
Input to Output	Tpd	25													
ALU Position & Width Mux - AmPAL22P10AL															
Input to Output	Tpd	25		25											
Register File - Am29334															
Address to Read Data Output	Access	24	24		24	24		24		24	24				
OE to Output Valid	Turn-on	20													
OE to Output Three-state	Turn-off	16													
Data Set-up	Tds	9	9	9	9	9		9	9			9			
ALU -- Am29332															
Data A or B to Y Parity		42	42		42	42						42			
Instruction to Y Parity		53													
Width to Y Parity		40													
Position to Y Parity		48		48											
Parallel Multiplier - Am29C323															
Unclocked Multiply X or Y to P Parity	Tmuc	150													
Clocked Multiply, Cycle Time	Tmc	125													
Clocked Multiply, Data to Clock Set-up	Tsxy	20													
Clocked Multiply, Clock to Output	Tpdpp	40													

Table 6-1C

Data Path Element Parameter Description	Symbol	Value	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8	Case 9	Case 10	Case 11	Case 12	Case 13
		Worst Case Time Delay in Nanoseconds, Over Commercial Operating Range													
Floating Point Processor - Am29325															
Unclocked Operations		125													
Clocked Operation		100													
Clocked Multiply, Data to Clock Set-up	Tsd1	13													
Clocked Multiply, Data to Clock Set-up	Tsd2	104					104								
FPP Seed Register - Am2920 & Am27S25															
OE to Output Valid	Tzh	35					35								
FPP External Status Register -AmPAL22V10A															
Clock to Output	Tco	15													
Input to Clock Set-up	Ts	20													
Macro Status Register - Am29818-1															
Clock to Output	Tpd	11													
Input to Clock Set-up	Ts	6													
Memory Address or Data Buffer -Am29827															
Input to Output	Tph	10						10		10					
OE to Output Valid	Tzh	17													
Memory Address Counters - AmPAL22V10															
Clock to Output	Tco	25													
Input to Clock Set-up	Ts	30													
OE to Output Valid	Tea, Ter	35							35						

Table 6-1D

Data Path Element Parameter Description	Symbol	Value	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8	Case 9	Case 10	Case 11	Case 12	Case 13
		Worst Case Time Delay in Nanoseconds, Over Commercial Operating Range													
Memory - Am99C165-35															
Chip Enable Access Time	Telqv	35													
Address Access Time	Tavqv	35						35	35						
Chip Enable to Output Disable	Thz	20													
Write Pulse Width	Twlwh	30													
Data to Write End Set-up	Tdvwh	20								20					
Address to Write End Set-up	Tavwh	30													
Write to Output Disable	Twlqz	10													
D_BUS - A_BUS Transceiver - Am29853															
Input to Parity Output	Tpd	15									15	15			
OE to Output Valid	Tzh	15													
D_BUS - A_BUS Parity Buffer - Am29862															
Input to Output	Tpd	6										6			
OE to Output Valid	Tzh	12													
Map RAM - Am9150-25															
Address to Data	Taa	25													25
Interrupt Controller - Am29114															
Clock to Interrupt Request		41													
Instruction Enable to Data Output		30													
Data in to Clock Set-up		10									10				
MINTA* to Vector OE		19													
Trap Logic -AmPAL22V10A															
Clock to Output	Tco	15												15	
Input to Clock Set-up	Ts	20													
OE to Output Valid	Tea, Ter	25												25	

Table 6-1E

Data Path Element Parameter Description	Worst Case Time Delay in Nanoseconds, Over Commercial Operating Range														
	Symbol	Value	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8	Case 9	Case 10	Case 11	Case 12	Case 13
Sequencer - Am29331															
Branch Input to Y Output		19											19		19
Instruction to Y Output		25													
Instruction to D Output		31													
Force Continue to Y Output		21													
Interrupt Request to Interrupt Ack		11												11	
OE D to D Valid		25										25			
Minimum Cycle Time per Case			96	97	115	90	169	99	94	94	89	127	84	81	85

Am29300 Demonstration System **Signal Path Timing Analysis**

Table 6-1F

Case Definitions

1. Basic flow through calculation, data path.
   Pipeline, Tco; Address Mux, Tpd; Register File, Tpd; ALU, Tpd; Register File, Set-up.

2. Basic flow through calculation, position control path.
   Pipeline, Tco; Position Mux, Tpd; ALU, Tpd; Register File, Set-up.

3. Flow through calculation with address supplied by operand counter; counter output enabled same cycle.
   Pipeline, Tco; Operand Counter, Tea; Register File, Tpd; ALU, Tpd; Register File, Set-up.

4. Flow through calculation with address supplied by operand counter; counter output enabled prior cycle.
   Pipeline, Tco; Operand Counter, Tco; Register File, Tpd; ALU, Tpd; Register File, Set-up.

5. First cycle of FPP Newton-Raphson division, seed value load.
   Pipeline, Tco; Control Decode, Tpd; Seed Register, Tzh; FPP Internal Register Set-up, Tsd2.

6. Memory read with address from the register file, selected by microcode.
   Pipeline, Tco; Address Mux, Tpd; Register File, Taa; Memory Address Buffer, Tpd; Memory, Taa; Register File, Set-up.

7. Memory read with address from a memory address counter.
   Pipeline, Tco; Control Decode, Tpd; Memory Address Counter, Tzh; Memory, Taa; Register File, Set-up.

8. Memory Write with data from register file, selected by operand counter.
   Pipeline, Tco; Operand Counter, Tea; Register File, Taa; Memory Address Buffer, Tpd; Memory, Write Set-up.

9. Move register file data to interrupt controller or sequencer, data selected by operand counter.
   Pipeline, Tco; Operand Counter, Tea; Register File, Taa; A to D Bus Xcever, Tpd; Interrupt Controller, Data Set-up.

10. Move sequencer or interrupt controller data to register file.
    Pipeline, Tco; Control Decode, Tpd; Sequencer, OED to D; D to A Bus Xcever, Tpd; Parity Buffer, Tpd; ALU, Tpd; Register File, Set-up.

11. Sequencer branch, conditional or unconditional.
    Pipeline, Tco; Pipeline Branch Field, Tzh; Sequencer, D to Y; Control Store, Address Set-up.

12. Sequencer interrupt or trap cycle.
    Trap Logic, Clock to INTR; Sequencer, INTR to INTA; Trap Logic, Tea; Control Store, Address Set-up.

13. Sequencer branch to macro opcode specified instruction.
    Macro Opcode Register, Tco; Map RAM, Taa; Sequencer A to Y, Control Store, Address Set-up.

SECTION 7

Physical Issues

ELECTRICAL LAYOUT ISSUES FOR POWER SUPPLY

The TTL compatible, bipolar, Am29300 family components all use internal ECL circuitry with TTL compatible I/O buffers.

Each part has a large number of output buffers due to the 32-bit output bus, plus various status outputs.

These two facts can make the real world interesting.

When a large number of the output buffers switch simultaneously, the local Printed Circuit Board (PCB) power and ground, and the chip internal power supply lines can experience significant noise transients.

This power supply noise can couple into the internal logic's ECL VCC pins. Since the internal ECL circuitry is referenced to the ECL VCC, the power supply noise can cause short duration shifts in the threshold levels of the internal logic.

Due to the way ECL circuitry operates, it has much smaller noise margins than equivalent TTL circuits. The threshold shifts result in lower than normal noise margins in already sensitive high speed circuits. These reduced noise margins can result in noise-induced logic errors.

It is, therefore, very important to provide very good power distribution and decoupling in a system using the Am29300 family. It is strongly suggested that a multilayer PCB be used to provide power and ground planes. It is also important to minimize coupling between the TTL and ECL VCC pins of any Am29300 bipolar device. This can be done in part through good power supply decoupling.

An additional way to decouple the ECL and TTL VCC pins is to introduce inductive isolation. The simplest way to do that is to place a cut in the VCC plane that separates the ECL supply pins from the TTL pins. This produces a longer electrical path between the pins, which adds inductance between the pins. This inductive isolation will significantly reduce noise coupling.

Some suggested PCB layouts for use with the Am29300 family are shown in Figures 7-1a and 7-1b. The images are negatives where black indicates an absence of metal in the VCC plane.

Although significant noise can also occur on the TTL and ECL ground lines, the ECL circuits are much less sensitive to this noise. Attempting to isolate the TTL and ECL ground pins from each other can create more problems than it solves. Any isolation will reduce the noise in the ECL circuitry and thereby make the chip internal ECL ground "different" from the TTL ground. This can reduce the noise margin in the ECL to TTL conversion logic, introducing potential for noise induced errors. It is recommended that no isolation between grounds be used.

DECOUPLING CAPACITORS

An added help in providing local VCC to ground decoupling is available in the form of under-chip capacitors.

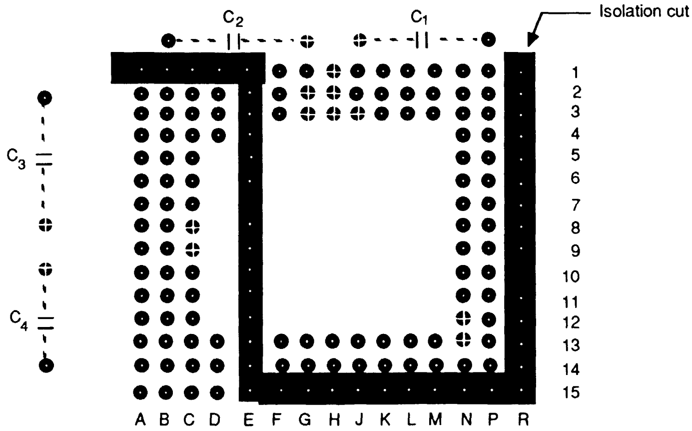
Special capacitors for PGA device packages have been developed by Rogers Corporation, Q-PAC Div., 2400 South Roosevelt St., Tempe, AZ. 85282.

SOCKETS

Whenever high pin count, expensive VLSI components are used in a system, many hardware designers prefer to have the devices in sockets. This allows easy removal for repairs or upgrades and provides an additional test point in the system.

Sockets for the Am29300 family are available from Augat Corporation, Interconnection Component Div. 33 Perry Ave. Attleboro, MA. 02703.

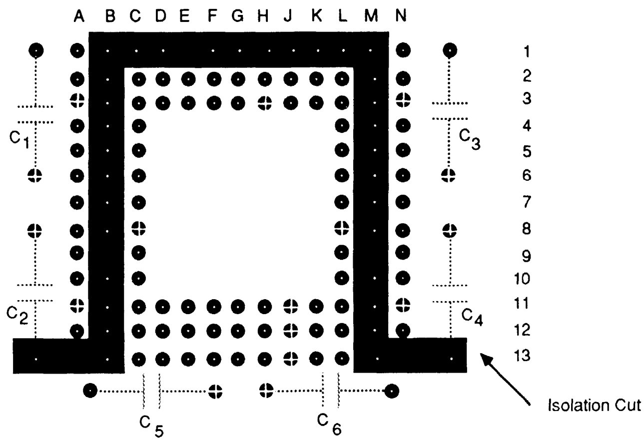
Am29325



● = Through Hole
⊕ = V_{CC} Plane Connection
$C_1 = C_3 = 0.1\ \mu F$
$C_2 = C_4 = 10\ \mu F$

05621D
29325

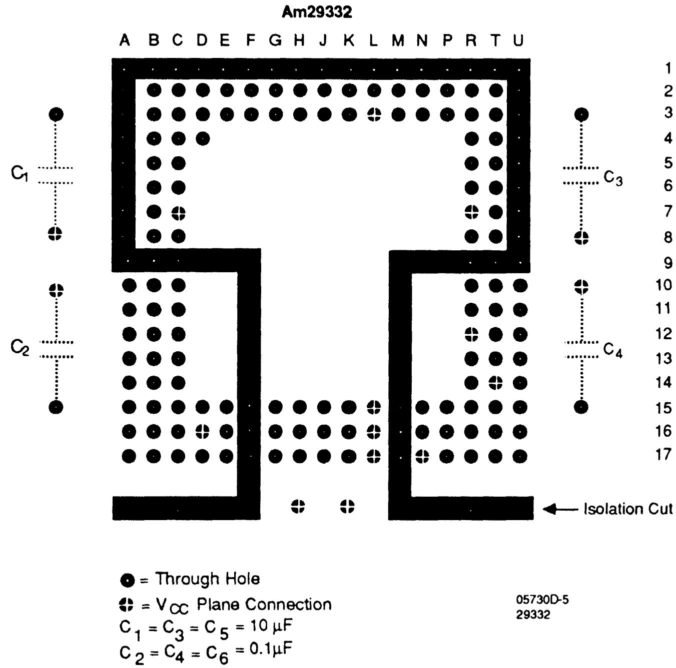
Am29331



● = Through Hole
⊕ = V_{CC} Plane Connection
$C_1 = C_3 = C_5 = 10\ \mu F$
$C_2 = C_4 = C_6 = 0.1 \mu F$

0572D-1
29331
09856A 7-1

Figure 7-1a. Layout Recommendations for the V_{cc} Plane



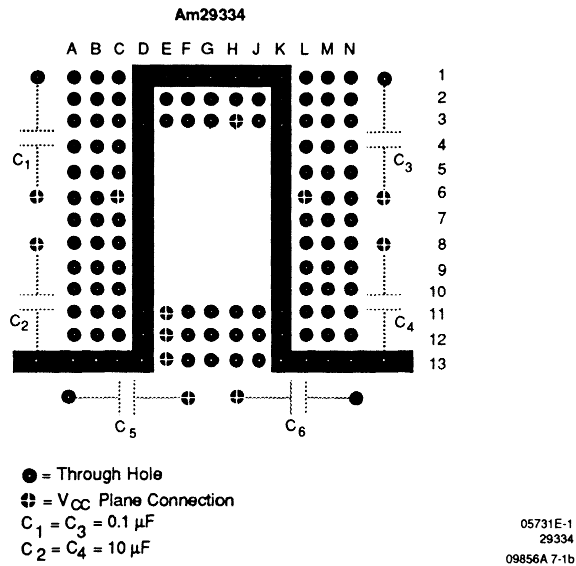


Figure 7-1b. Layout Recommendations for the V_{cc} Plane

SECTION 8

Conclusion

There are many ways to skin a cat and surprisingly, many more ways to build a computer. This application note has tried to guide the reader through just one simple implementation. The author hopes some of the reasons behind the design choices in a microprogrammed computer design were made clear during the course of the description.

Aside from some general notions about how a microprogrammed system works, the reader should walk away having noted the following thoughts:

This design is a full 32-bit processor capable of executing a full 32-bit add, barrel shift, logical, integer multiply, or even floating point multiply every 100 ns to 133 ns. That is a 7 to 10 Million Instructions Per Second (MIPS) rate, which is (loosely) comparable to 7 times the performance of a VAX 11/780.

For all that computing horsepower, the real core of this machine is made from only 6 chips: the Am29300 family of computer building blocks. That's an incredible degree of integration as compared with previous approaches to high performance microprogrammed computer design.

Most of the logic surrounding the Am29300 family components is not required. The additional logic is used to add system flexibility and to show off different aspects of microprogrammed design. Very little glue is needed to hold this family together.

There is very little in the way of standard SSI logic used. Virtually all the MSI and SSI level logic functions were incorporated into Programmable Array Logic. This shows the versatility and integration that PALs can provide.

Due to use of Serial Shadow Registers throughout the system, there is a reasonable hope that enough of the system state can be read and controlled so that debugging in the factory or field will be simple. This access to the internal structure of the machine is gained with very little “excess” logic.

APPENDIX A

Related Reference Material

Bit-Slice Microprocessor Design, John Mick and Jim Brick, McGraw-Hill, 1980.

Bit-Slice Design: Controllers and ALUs, Donnamaie E.White, Garland Publishing, 1981.

Am29300 Family Handbook, Advanced Micro Devices, Sunnyvale, CA., 1985.

Am29300 Family Datasheets, Advanced Micro Devices, Sunnyvale, CA.

32-Bit Building Blocks for High Performance Processor/Controller, Paul Chu, Advanced Micro Device, Sunnyvale, CA.,

A Very High Speed Floating Point Processor, B. J. New, Advanced Micro Devices, Sunnyvale, CA.

Signal-to-Figure Cross Reference

This is a "signal name to document figure number cross reference listing." Where a signal name is followed by a colon (:) the signal is identified as an I/O signal to that figure. Where a signal name is followed by a semi-colon (;) the signal is identified as an output from that figure. A plain signal name indicates an input to that figure.

ALU_HOLD	3-2
ALU_HOLD;	5-21
ALU_OE*	3-2
ALU_OE*;	5-21
ANY_E*	4-4
ANY_E*;	4-3
ARA	3-1
ARA	5-6
ARA;	5-4
ARA;	5-5
ARB	3-1
ARB	5-6
ARB;	5-4
ARB;	5-5
AWA	3-1
AWA;	5-6
AWA_MAC	5-6
AWA_MAC	5-7
AWA_MAC;	5-4
AWB	3-1
AWB;	5-7
A_BUS	3-2
A_BUS	3-3
A_BUS	3-6
A_BUS	4-2
A_BUS	4-8
A_BUS;	3-1
A_BUS;	3-5
A_BUS;	5-10
A_FULL	5-11
A_FULL;	5-9
A_MD_OE*	4-8
A_MD_OE*;	5-21
B_BUS	3-2
B_BUS	3-3
B_BUS	3-6
B_BUS	4-2
B_BUS	4-8
B_BUS;	3-1
CASOUT2	5-11
CASOUT2;	5-12
CLK_CNTL	5-13
CLK_CNTL	5-14
CLK_CNTL	5-4
CLK_CNTL;	5-15
CLK_D	3-1
CLK_D	3-2
CLK_D	3-3
CLK_D	3-5
CLK_D	3-6
CLK_D	4-7
CLK_D;	5-15
CLK_FREE_RUN	4-3
CLK_FREE_RUN	4-4
CLK_FREE_RUN	4-6
CLK_FREE_RUN*	5-15
CLK_FREE_RUN*;	5-16
CLK_FREE_RUN;	5-15
CLK_INT_BASE	5-11
CLK_INT_BASE;	5-15
CLK_MAC_STAT	3-2
CLK_MAC_STAT;	5-15
CLK_MOP	5-1
CLK_MOP;	5-15
CLK_PIPE	5-3
CLK_PIPE	5-13
CLK_PIPE	5-14
CLK_PIPE;	5-15
CLK_SEQ	5-11
CLK_SEQ	5-12
CLK_SEQ	5-9
CLK_SEQ;	5-15
CLOCK_CNTL	5-15
CLOCK_CNTL;	4-2
CLOCK_CNTL;	4-3
CNTL_EN	4-4
CNTL_EN;	5-15
CPU_BUS_EN*	4-4
CPU_BUS_EN*	4-8
CPU_BUS_EN*;	4-3
DCLK_MOP	3-2
DCLK_MOP	4-5
DCLK_MOP	5-1
DCLK_MOP	5-11
DCLK_MOP;	4-3
DCLK_SSR	4-5
DCLK_SSR;	4-3
DCLK_WCS	4-5
DCLK_WCS	5-13
DCLK_WCS	5-14
DCLK_WCS	5-3
DCLK_WCS;	4-3
DECODE_ADD	5-9
DECODE_ADD;	5-3
D_BUS	5-10
D_BUS	5-11
D_BUS	5-9
D_BUS;	5-20
D_OER*	5-10
D_OER*;	5-21
D_OET*	5-10
D_OET*;	5-22
D_SIGN_EX	5-10
D_SIGN_EX;	5-22
EQUAL	5-11
EQUAL	5-12
EQUAL;	5-9
EXT_ADD	4-2
EXT_ADD	4-3
EXT_ADD	4-9
EXT_BUS_EN*	4-9
EXT_BUS_EN*;	4-3
EXT_DATA	4-2
EXT_DATA	4-9
EXT_INTR;	4-2
EXT_INTR;	4-4
EXT_READY	4-2
EXT_READY	4-3
EXT_RESET	4-2
EXT_RESET	4-3
EXT_WEN*	4-2
EXT_WEN*	4-3
EXT_WEN*	4-9
E_ADD	4-4
E_ADD;	4-3
FP_FT	3-3
FP_FT;	5-21
FTP	3-6
FTP;	5-21
IEN*	5-11
IEN*;	5-22
INIT_MC*	5-13
INIT_MC*	5-14
INIT_MC*;	5-21
INTA*	5-11

Signal	Figure
INTA*	5-12
INTA*;	5-9
INTR	5-9
INTR;	5-11
INTR;	5-12
INT_CPU*	5-11
INT_CPU*;	5-15
INT_CS*	5-11
INT_CS*;	5-22
INT_FPP*	5-11
INT_FPP*;	3-3
M1_BUS	5-9
M1_BUS;	3-3
M2_BUS	5-9
M2_BUS;	3-3
MAC_OP	5-3
MAC_OP	5-4
MAC_OP	5-8
MAC_OP	5-8
MAC_OP	5-9
MAC_OP;	5-1
MAC_STATUS_BUS	5-9
MAC_STATUS_BUS;	3-2
MA_BUS	4-6
MA_BUS:	4-7
MA_BUS;	4-8
MA_BUS;	4-9
MC_ADDRESS	5-13
MC_ADDRESS	5-14
MC_ADDRESS;	5-11
MC_ADDRESS;	5-12
MC_ADDRESS;	5-3
MC_ADDRESS;	5-9
MD_BUS	3-1
MD_BUS	5-1
MD_BUS:	4-5
MD_BUS:	4-9
MD_BUS;	4-6
MD_BUS;	4-8
MEM_EN*	4-4
MEM_EN*;	4-3
MEM_WEN*	4-6
MEM_WEN*;	4-8
MEM_WEN*;	4-9
MINTR*	5-12
MINTR*;	5-11
MODE	3-2
MODE	4-4
MODE	4-5
MODE	5-1
MODE	5-11
MODE	5-13
MODE	5-14
MODE	5-3
MODE;	4-3
P(15:0);	5-14
P(91:0)	518, 5-19, 5-20
P(91:16);	5-13
PE_ALU	5-11
PE_ALU;	3-2
PE_D_BUS*	5-11
PE_D_BUS*;	5-10
PE_PM	5-11
PE_PM;	3-6
PM_OE*	3-6
PM_OE*;	5-21
POSITION	3-2
POSITION;	5-8
P_ACC	3-6
P_ACC;	5-19
P_ALU_INST	3-2
P_ALU_INST;	5-19
P_ARA_MAC	5-4
P_ARA_MAC	5-5
P_ARA_MAC;	5-18
P_ARB_MAC	5-4
P_ARB_MAC	5-5
P_ARB_MAC;	5-18
P_AWB_MAC	5-7
P_AWB_MAC;	5-18
P_BM	3-2
P_BM;	5-19
P_BRANCH_EN*	5-14
P_BRANCH_EN*	5-22
P_BRANCH_EN*;	5-20
P_CLK_LEN	5-16
P_CLK_LEN;	5-20
P_CNTA_EN	5-4
P_CNTA_EN;	5-18
P_CNTB_EN	5-4
P_CNTB_EN;	5-18
P_CNTC_EN	5-4
P_CNTC_EN;	5-18
P_C_SEL	5-4
P_C_SEL	5-6
P_C_SEL;	5-18
P_DPS	3-3
P_DSP	5-21
P_DSP;	5-19
P_ENF*	3-3
P_ENF*;	5-19
P_ENP*	3-6
P_ENP*;	5-19
P_ENR*	3-3
P_ENR*;	5-19
P_ENS*	3-3
P_ENS*;	5-19
P_ENT*	3-6
P_ENT*;	5-19
P_ENXA*	3-6
P_ENXA*;	5-19
P_ENXB*	3-6
P_ENXB*;	5-19
P_ENYA*	3-6
P_ENYA*;	5-19
P_ENYB*	3-6
P_ENYB*;	5-19
P_FA	3-6
P_FA;	5-19
P_FC*	4-4
P_FC*	5-12
P_FC*	5-15
P_FC*	5-21
P_FC*	5-22
P_FC*;	5-20
P_FP_FT	5-21
P_FP_FT;	5-19
P_FP_I	3-3
P_FP_I;	5-19
P_FP_RND	3-3
P_FP_RND;	5-19
P_FTP	5-21
P_FTP;	5-19
P_FTX	3-6
P_FTX;	5-19
P_FTY	3-6
P_FTY;	5-19
P_IEEE/DEC*	3-3
P_IEEE/DEC*	3-5
P_IEEE/DEC*;	5-19
P_INIT	5-21
P_INIT;	5-20
P_INT_EN	5-9
P_INT_EN;	5-20
P_INT_HOST	4-4
P_INT_HOST;	5-20
P_INT_INST	5-11
P_INT_INST	5-22
P_INT_INST;	5-20
P_LD_CNT	5-4

Signal	Figure
P_LD_CNT;	5-20
P_LD_INT_BASE	5-15
P_LD_INT_BASE;	5-20
P_LD_MAC_OP	5-15
P_LD_MAC_OP;	5-20
P_LD_MAC_STAT	5-15
P_LD_MAC_STAT;	5-19
P_MEM	4-4
P_MEM	4-7
P_MEM	5-21
P_MEM;	5-19
P_MEM_WR*	4-8
P_MEM_WR*;	5-19
P_MIC/MAC	3-2
P_MIC/MAC;	5-19
P_OEA*	3-1
P_OEA*	5-21
P_OEA*;	5-18
P_POSITION	5-8
P_POSITION;	5-19
P_POS_MAC	5-8
P_POS_MAC;	5-19
P_PROJ/AFF*	3-3
P_PROJ/AFF*;	5-19
P_PSEL	3-6
P_PSEL;	5-19
P_RA	5-5
P_RA;	5-18
P_RB	5-5
P_RB;	5-18
P_RC	5-6
P_RC	5-7
P_RC;	5-18
P_REG_STAT	3-2
P_REG_STAT;	5-19
P_RND	3-6
P_RND;	5-19
P_SEED_OE*	5-21
P_SEED_OE;	5-19
P_SEQ_INST	5-22
P_SEQ_INST	5-9
P_SEQ_INST;	5-20
P_TCX	3-6
P_TCX;	5-19
P_TCY	3-6
P_TCY;	5-19
P_TEST	5-9
P_TEST;	5-20
P_TSEL	3-6
P_TSEL;	5-19
P_UP/DN_A	5-4
P_UP/DN_A;	5-18
P_UP/DN_B	5-4
P_UP/DN_B;	5-18
P_UP/DN_C	5-4
P_UP/DN_C;	5-18
P_WEA*	3-1
P_WEA*;	5-18
P_WEB*	3-1
P_WEB*;	5-18
P_WIDTH	5-8
P_WIDTH;	5-19
P_WID_MAC	5-8
P_WID_MAC;	5-19
P_XSEL	3-6
P_XSEL;	5-19
P_YSEL	3-6
P_YSEL;	5-19
RESET_300*	5-11
RESET_300*	5-12
RESET_300*	5-9
RESET_300*;	5-15
SDI_SSR_MUX	4-4
SDI_SSR_MUX	4-5
SDI_SSR_MUX;	4-3
SD_0	4-5
SD_0;	4-4
SD_1	5-1
SD_1	5-3
SD_1;	4-2
SD_2	3-2
SD_2;	5-1
SD_3	5-11
SD_3;	3-2
SD_4	4-2
SD_4	4-4
SD_4;	5-11
SD_5	5-14
SD_5;	5-3
SD_5A	5-13
SD_5A;	5-14
SD_6	4-2
SD_6	4-4
SD_6;	5-13
SEED_OE*	3-5
SEED_OE;	5-21
SEQ_CIN*	5-9
SEQ_CIN*;	5-11
SEQ_CIN*;	5-12
SEQ_FC	5-9
SEQ_FC;	5-11
SEQ_FC;	5-12
SEQ_OED	5-9
SEQ_OED;	5-22
SSR_BUS_EN*	4-5
SSR_BUS_EN*;	4-3
STATUS_BUS	5-9
STATUS_BUS	5-9
STATUS_BUS;	3-2
SYS_MEM_EN*	4-6
SYS_MEM_EN*;	4-4
TRAP	5-15
TRAP;	5-11
TRAP;	5-12
WCS_INIT*	4-5
WCS_INIT*	5-21
WCS_INIT*;	4-3
WCS_WR*	4-5
WCS_WR*	5-3
WCS_WR*	5-9
WCS_WR*;	4-3
WIDTH	3-2
WIDTH;	5-8
Y_BUS	3-1
Y_BUS;	3-2
Y_BUS;	3-3
Y_BUS;	3-6

APPENDIX C
FPP Status PAL Definition

```
"           Advanced Micro Devices Application Note:
"
"           Am29300 Demonstration System
"
"           By Mark Mc Clain, Field Applications Enginee, San Dieg, CA.
"           (619)560-7030, Date = 1/87
Module
            FPP_STATUS_REG;
Title
'FPP status register PAL for an Am29300 Demonstration System.';
fpppl device 'P22V10';
"declarations
            X,Z,C,P = .X.,.Z.,.C.,.P.;
            "Signal names that end in an underline indicate an active low signal.
            CLK_D, P_DSP_1, P_DSP_0, FP_FT_1, INEXACT, INVALID, NAN Pin
              1,      2,       3,       4,       5,       6,      7;
            OVERFLOW, UNDERFLOW, ZERO Pin
               8,         9,       10;
            M1_0, M1_1, M1_2, M1_3, M2_0, M2_1, M2_2, INT_FPP_, delay_load Pin
             14,   15,   16,   17,   18,   19,   20,     21,          22;
"  Some outputs are declared as active high. This requires that ABEL
"  version 2.0 or later be used to compile this definition. Earlier
"  versions of ABEL have a bug that assumes all programmable pins
"  in the 22V10 are active low regardless of how they are declared.
"  Earlier versions of ABEL will generate an incorrect JEDEC file.
            M1_0, M1_1, M1_2, M1_3, M2_0, M2_1, M2_2
                 Istype 'pos,.reg';
            INT_FPP_
                 Istype 'neg, com';
" define some constants
            status      = [M2_2, M2_1, M2_0, M1_3, M1_2, M1_1, M1_0];
            dsp    = [P_DSP_1, P_DSP_0];
            fpp    = ^b10;
            special     = ^b11;
```

```
" define a macro

            OR_EM macro (a,b,c) {(?a # ?b # ?c)};

EQUATIONS

"           The delay_load signal will cause the status register to load the
"           cycle after a cycle in which the FPP data path was selected; if the
"           FPP output register flow through is disabled. If the flow through is
"           inactive, the flag register in the FPP holds the status in the first
"           cycle.

            delay_load :=     (dsp == fpp)     & !FP_FT_1 #
            (dsp == special) & !FP_FT_1;

"           The status register loads with the value of the input flags when the
"           FPP data path is enabled and the FPP output register flow through
"           signal is active. It also will load if the delay_load signal is
"           active. If neither condition is true, the status register will load
"           with its own current value. This retains the state of the status
"           register when there is no enable to load a new flag value.

            status :=

            !((dsp == fpp) # (dsp == special)) & status & !delay_load #

             ((dsp == fpp) # (dsp == special)) & status & !delay_load

                & !FP_FT_1    #

            delay_load
                & [ INEXACT, NAN, ZERO, OR_EM(INEXACT, NAN, ZERO),
                    INVALID, OVERFLOW, UNDERFLOW]  #

            ((dsp == fpp) # (dsp == special)) & FP_FT_1
                & [ INEXACT, NAN, ZERO, OR_EM(INEXACT, NAN, ZERO),
                    INVALID, OVERFLOW, UNDERFLOW]  ;

"           The FPP interrupt to the CPU is based on the OR of flags_iou.

            !INT_FPP_ =

                INVALID # OVERFLOW # UNDERFLOW;

"           Test_Vectors need to be defined.

End;
```

APPENDIX D

Host Interface Glue PAL Definition

```
"                       Advanced Micro Devices Application Note:
"
"                       Am29300 Demonstration System
"
"                       By Mark Mc Clain, Field Applications Engineer, San Diego,
                        CA.
"
                        (619)560-7030, Date = 1/87

Module
Host_Interface_Glue;

Title
'Host Interface Glue PAL for an Am29300 Demonstration System.';

hig device 'P22V10';

"declarations

            X,Z,C,P = .X.,.Z.,.C.,.P.;

            "Signal names that end in an underline indicate an active low signal.

            CLK_FREE_RUN, P_INT_HOST, P_FC_, E_ADD_17, E_ADD_16, ANY_E_ Pin
                1,            2,        3,       4,        5,        6;

            P_MEM_1, P_MEM_0, CPU_BUS_EN_, MEM_EN_, SD_4, SD_6, MODE Pin
               7,       8,         9,          10,     11,   13,    14;

            SDI_SSR_MUX, CNTL_EN Pin 15, 19;

            EXT_INTR, SYS_MEM_EN_, SD_0 Pin
               16,        17,      18;

"           Some outputs are declared as active high. This requires that ABEL
"           version 2.0 or later be used to compile this definition. Earlier
"           versions of ABEL have a bug that assumes all programmable pins
"           in the 22V10 are active low regardless of how they are declared.
"           Earlier versions of ABEL will generate an incorrect JEDEC file.

            EXT_INTR
               Istype 'pos, reg';

            SD_0
               Istype 'pos, com';

            SYS_MEM_EN_
               Istype 'neg, com';
```

```
EQUATIONS
"         The external bus (host) interrupt signal comes from a registered
"         latch which is set active when the pipeline Force Continue and
"         Interrupt host bits are active along with the control section clock
"         enable. The control section clock enable is needed since the
"         register is clocked by the system free running clock. The free
"         running clock is used so the latch can always be cleared by the
"         host whether the CPU clocks are active or not. But we don't want to
"         set an interrupt unless the control section of the CPU really means
"         it (i.e. completes the execution of the current clock cycle with
"         the interrupt bits active. The interrupt is cleared by the host
"         when any access (read or write) is done to the host interface SSR
"         Port.

            EXT_INTR :=

                CNTL_EN & P_INT_HOST & !P_FC_ #
                EXT_INTR & !(!ANY_E_ & !E_ADD_17 & E_ADD_16);

"         When the CPU bus buffers are enabled the system memory is enabled
"         by either of the LSB P_MEM pipeline bits being active. When the CPU
"         bus buffers are not enabled the memory is enabled only by the host
"         interface controller MEM_EN_.

            SYS_MEM_EN_ =

                !CPU_BUS_EN_ & (P_MEM_1 # P_MEM_0) #
                CPU_BUS_EN_ & !MEM_EN_ ;

"         The Serial Data path bit zero is the output of a Multiplexer. When
"         MODE is low, the SDI_SSR_MUX input is used as a mux selector for
"         the SD_4 or SD_6 input. If MODE is high then the value of
"         SDI_SSR_MUX is passed through the mux to be the SDI input of the
"         SSR Port.

            SD_0 =

                MODE & SDI_SSR_MUX #
                !MODE & ((SDI_SSR_MUX & SD_6) # (!SDI_SSR_MUX & SD_4));

"         Test_Vectors need to be defined.
End;
```

APPENDIX E

Host Interface Am29PL141 Program

```
"   Advanced Micro Devices Application Note:
    29300 Demonstration System
    External (Host) Interface Controller Program Definition

This file is formated for the Am29PL141 assembler program, which is available from
Advanced Micro Devices.

By Mark Mc Clain, Field Application Engineer, San Diego 12/86
"
Device (PL141)                      "The device used is an Am29PL141
                                    Fuse Programmed Controller"

SSR = 0;                            "No SSR diagnostics used."

Default = 1;        "Unused fuses will be left intact (just in case, I may be able to
                                    patch a bug without burning a new PL141 if the
                                    right fuses are still intact)"

Define

" Note:all test inputs are externally registered"

not_write =cc       "External bus !WR signal tied to CC input."

not_select =t5      "External bus selection signal (board select and address valid
                    indication) tied to test input 5."

address_17 =t4      "External bus address bit 17 tied to test input 4."

address_16 =t3      "External bus address bit 16 tied to test input 3."

address_2 =t2       "External bus address bit 2 tied to test input 2."

address_1 =t1       "External bus address bit 1 tied to test input 1."

address_0 =t0       "External bus address bit 0 tied to test input 0."

"  Address bits 23..18 are externally matched to a board address to
"  generate the not_select signal to the external bus interface
"  controller. Address bits 16 and 17 are used to select one of four
"  command modes in the interface controller. Address bits 2..0 are used
"  as command modifiers or as the upper 3 bits of a 5 bit count value
"  used in diagnostic shifting.

command_mask = 111000#b "Pass not_select (address bits valid if low) and pass ad-
                        dress_16 and address_17 to select the command mode."

cmd_modifier_mask = 011111#b "Pass address bits 2..0, address_17 and address_16 to
                             select a specific operation."

count_mask = 000111#b "Pass address bits 2..0  when a counter value is being
                      loaded."
```

```
;  "end of define section" "  "
"No Default output pattern is defined"

"No Default test condition is defined"

"
Assembler code format is as follows:

Label          Output Pattern      , Opcode      Comments (in quotes)

                           CCCC          <== If you put your glasses on
                           LLLL          and read vertically, you will
                      S!!! KKKK          see the headers for each bit
                      DCES ----          of the output pattern. Listed
                !     IPXS CCCC          most to least significant:
                W DDD -UTR OOOO          READY             !CPU_BUS_EN
                C!CCC S---!NNNN          !WCS_INIT         !EXT_BUS_EN
                SWLLL SBBBMTTTT          !WCS_WR           !SSR_BUS_EN
               R-CKKK RUUUERRRR          DCLK_WCS          !MEM_EN
               EIS---M-SSSMOOOO          DCLK_MOP          CLK_CONTROL_3
               AN-WMSOM----LLLL          DCLK_SSR          CLK_CONTROL_2
               DIWCOSDUEEEE----          MODE              CLK_CONTROL_1
               YTRSPREXNNNN3210          SDI_SSR_MUX       CLK_CONTROL_0
```

Let's define what the outputs are used for:

READY is the signal that is passed back to the host system bus to cause wait states that hold the host bus steady while the Am29PL141 (external bus interface controller) performs the function requested by the host. While READY is inactive (low) the external bus is held. When READY goes active the external bus may proceed. The default state of READY is inactive. It is set active only by the last instruction in a command execution routine.

!WCS_INIT is used in loading or reading the initialize word of the microcode memory. Its default state is inactive (high).

!WCS_WR is used to write the Macro Opcode Map RAM. This is done at the same time that the writable control store is written. Its default state is inactive (high).

DCLK_WCS is the shift clock to all shadow registers in the Writable Control Store SSR chain. It is used to shift data into the writable control store SSR. Its default state is inactive (low).

DCLK_MOP is the shift clock to all shadow registers in the Macro Opcode, Status, and Interrupt SSR chain. It is used to shift data into those SSR. Its default state is inactive (low).

DCLK_SSR is the shift clock to all shadow registers in the SSR diagnostics port. It is used to shift data into or out of the SSR port. Its default state is inactive (low).

MODE controls the mode of operation for all the SSR in the diagnostics chains. Its default state is inactive (low).

SDI_SSR_MUX is a signal that, depending on the state of MODE, controls whether the WCS or MOP SSR chain is shifted in to the SSR port register via the SSR multiplexer; or acts directly as the Serial Data Input to all the SSR diagnostics chain. Its default state is inactive (low).

!CPU_BUS_EN is the enable control over the buffers between the Am29300 CPU and the memory bus. Its default state is active (low).

!EXT_BUS_EN is the enable control over the buffers between the External Bus and the memory bus. Its default state is inactive (high).

!SSR_BUS_EN is the enable control over the output buffers of the SSR port registers. Its default state is inactive (high).

!MEM_EN is the enable control of the memory. Its default state is inactive (high).

CLK_CONTROL_ 3..0 is a four bit encoded command to a control PAL in the Am29300 system clock distribution circuit. The field defines the following commands to the clock control circuits:

```
0000    No Operation, default state.
0001    Single cycle HALT of all clocks.
0010    Enter HALT mode continuously.
0011    Enter RUN mode on all clocks.
0100    Single step all clocks.
0101    Load control pipeline only.
0110    Single step control section and pipeline only.
0111    Load Macro Opcode, Status, and Interrupt registers.
1000    Single step data section only.
1001    Single step all clocks with Am29300 reset active.
1010    Set semiphore interrupt to the Am29300 CPU.
1011    Reserved.
1100    Reserved.
1101    Reserved.
1110    Reserved.
1111    Reserved.
```

Program Structure:

The Am29PL141 is a MICROPROGRAMMED controller. The whole point of using the micro-program approach is to make hardware development easier, more structured, and flex-ible than would be possible with a pure hard-wired approach; do as many functions in parallel as possible; perform one instruction per clock cycle for maximum speed; and make very frugal use of the available program memory space.

The result tends to make either the memory map or the program flow of a microprogram difficult to follow. If the microprogram instructions are listed in the order they appear in memory so the positional relationships are clear, then the program flow is hard to see. If the instructions are listed to show program flow, then the posi-tional relationships necessary to the understanding of some instructions like multi-way branches are hard to see.

In this program definition a compromise has been made between the two views. Where it is possible to show program flow by listing instructions together, that has been done. Often these instructions are widely separated in physical location due to the requirements of positional instructions like multi-way branches. To provide the correct positioning of instructions while showing program flow, a liberal use of ORIGIN statements has been made.

A memory map showing the physical location of all instructions is shown at the end of the program listing.

Program Overview:

The general idea of this program is to spin in a wait loop until the host system addresses the Am29300 system. When addressed, this controller does a multi-way branch to one of four command modes based on the address used. One of the modes (memory access) has a valid address in bits 2..0. For that reason the multi-way branch is not based on any of the bits 2..0. To use those bits would consume too many entry points in the program memory. So, if the command specified by the host address is for a memory access or an access to the SSR port, then command execution begins immediately. This makes memory and SSR port accesses occur with the maximum possible speed. If the command address is for one of the other 14 commands, a second multi-way branch is executed based on address bits 2..0, which in that case contain command modifiers that select a specific command. Because these multi-way branches are based on the LSB address bits, the command entry points are all adjacent. This requires that any command requiring more than one instruction to execute does a branch to another location in program memory where the remainder of the instructions may be located.

A note about conditional instructions: most of the Am29PL141 instructions are conditional, but often an uncondition operation is desired. The way to get an unconditional operation is to use the eq = 0 test condition. The eq flag is initially cleared by the reset instruction and is always left reset in this program so that all instructions referencing eq are, in effect, unconditional instructions.

The remaining details of each command are well commented on in the program code itself.

THE PROGRAM:
As with all things that are logical and obvious, if you want to follow the program flow, you need to begin at the end.
The program flow starts at reset; location 63.

```
"

Begin

.org 63           "Hardware reset will force the program counter to location 63 so
                  we had better define the outputs we want to see during reset and
                  define where to start execution after reset.
                  "

reset: 0110000011110010#b , if (eq = 0) then goto pl(idle);

                  "Output explanation:

                  If there is a hardware reset we are definitely not READY. A hard-
                  ware reset will also put the Am29300 CPU clocks in HALT mode so
                  random strange things won't be happening. All other outputs are
                  default state.

                  Opcode explanation:

                  Since the eq flag is cleared by a hardware reset, the opcode is an
                  unconditional branch to the idle loop.
                  "
```

```
.org 100000#b

idle: 0110000001110000#b , if (eq = 0) then goto tm(command_mask);

.org 101000#b

idle2: 0110000001110000#b , if (eq = 0) then goto tm(command_mask);

.org 110000#b

idle3: 0110000001110000#b , if (eq = 0) then goto tm(command_mask);

.org 111000#b

idle4: 0110000001110000#b , if (eq = 0) then goto tm(command_mask);

                    "Output explanation:

                    In the idle states you are not READY. All other outputs
                    are in their default states (i.e. if the host knocks on the door,
                    you're not ready yet and while you're hanging around being idle
                    don't do anything rash with the other outputs).

                    Opcode explanation:

                    The main loop (i.e. idle loop) for the external bus interface
                    controller is a single instruction that loops on itself until a
                    valid command from the external bus is received.  When a valid
                    command does appear, an immediate multi-way branch occurs, which
                    transfers control to the code that handles a  command. This pro-
                    vides a very fast command execution by beginning the command
                    routine in the clock  cycle following the command appearance.

                    The multi-way branch address is based on three bits from the
                    external bus: the not_select, address_16, and address_17 bits.
                    This means that the multi-way branch will transfer control to one
                    of eight locations in the microcode. Four of these locations are
                    the starting instructions for the four command modes. These loca-
                    tions are reached when the not_select bit is active (low). Four
                    other locations are reached when the not_select bit is inactive
                    (high), meaning that address_16 and address_17 are invalid (don't
                    care). Since these two LSB bits of the branch address are don't
                    care, each location potentially addressed contains an identical
                    instruction that performs another multi-way branch, i.e. forming
                    the loop-on-itself instruction. These locations are referred  to
                    as idle states.

                    The idle states occur at locations 32, 40, 48, and 56 since the
                    MSB of the multi-way branch address is the not_select bit. If the
                    external address bits are not valid (not_select = 1), then the
                    Am29300 board is not being addressed and address_16  and ad-
                    dress_17 have no meaning. This implies that the branch address
                    will fall in the locations just noted. Therefore, each location
                    has the same opcode so that from the program execution point of
                    view the controller sits in a single cycle loop waiting for the
                    not_select input to go low.
```

```
                  When the not_select input does go low the multi-way branch trans-
                  fers control to one of the locations 0, 8, 16, or 24.
                  "

.org 000000#b

memory: 0110000010100001#b , continue;

                  "Output explanation:
                  This is the memory access command that requests a read or write of
                  the dual port main memory on the Am29300 board. During the execu-
                  tion of this micro instruction the CPU bus buffer is disabled and
                  the clocks to the CPU are halted to place the CPU in suspended
                  animation while the external bus (host) is given access to the
                  memory. The external bus buffers are enabled and the memory is
                  enabled. The memory begins its access cycle. The memory data on a
                  read cycle will not be valid until near the end of the microin-
                  struction cycle so the READY line will not go active in this
                  cycle. All other outputs are in the default state.

                  Opcode explanation:

                  Continue to next instruction.
                  "

memory2: 1110000010100001#b , if (not_select = 1) then goto pl(idle)    else wait;

                  "Output explanation:

                  The memory has had time to be accessed and drive data on to the
                  external bus (or receive data from the bus) so the bus transaction
                  can now proceed. The READY line is made active to allow progress.
                  All other outputs remain as they were in the last cycle of the
                  memory access routine.

                  Opcode explanation:

                  The bus interface controller now needs to hold things steady until
                  the external bus (host) indicates it is ready to proceed and
                  release the bus. This is indicated by the not_select signal going
                  inactive (i.e. the host stops addressing the Am29300 board). Until
                  the not_select input goes inactive this instruction loops on
                  itself. When it does go inactive the instruction branches back to
                  the idle routine to wait for the next command.
                  "

.org 001000#b

ssr_port: 0110001010110001#b , if (not_write) then goto pl(ssr_read);

                  "Output explanation:

                  Since this is only the first instruction of the ssr_port access
                  routine, the READY signal will be inactive. The MODE output is
                  made active now to put the ssr_port register into diagnostics mode
                  in anticipation of the next instruction, which will drive both
```

```
                  DCLK and PCLK active to the ssr_port. The SDI_SSR_MUX signal is
                  kept inactive so that when DCLK and PCLK go active the shadow
                  register will load from the ssr_port register outputs. The CPU bus
                  buffers are disabled and the CPU clocks halted while the external
                  bus buffers are enabled to allow for data set-up time if this will
                  be a write to the ssr_port register. All other outputs are in the
                  default state.

                  Opcode explanation:

                  If this ssr_port command is a read operation (not_write=1) then
                  branch to the instruction that enables the register data onto the
                  bus. Otherwise fall through to the instruction that writes data
                  into the ssr_port register.
                  "

ssr_write: 1110011010110001#b , if (not_select = 1) then goto pl(idle)  else wait;

                  "Output explanation:

                  The data on the bus has been written into the ssr_port shadow
                  register at the rising edge of DCLK_SSR and the READY signal is
                  made active to allow the bus transaction to end. MODE remains high
                  in this cycle to prevent any skew problems between it and the
                  rising edge of DCLK_SSR. SDI_SSR_MUX remains low in this cycle for
                  the same reason. The external bus buffers remain enabled and the
                  CPU clocks halted. All other outputs are in the default state.

                  Opcode explanation:

                  Return to idle when the host releases the bus, else wait.
                  "

ssr_read : 1110011010010001#b , if (not_select = 1) then goto pl(idle)  else wait;

                  "Output explanation:

                  The data from the shadow register has been written into the pipe-
                  line register of ssr_port register at the rising edge of DCLK_SSR
                  (which also serves as PCLK to the ssr_port) and the READY signal
                  is made active to allow the bus transaction to end. !SSR_BUS_EN is
                  made active to enable the ssr_port pipeline register contents onto
                  the bus to be read by the host system. MODE remains high in this
                  cycle to prevent any skew problems between it and the rising edge
                  of DCLK_SSR. SDI_SSR_MUX remains low in this cycle for the same
                  reason. The external bus buffers remain enabled and the CPU clocks
                  halted. All other outputs are in the default state.

                  Opcode explanation:

                  Return to idle when the host releases the bus, else wait.
                  "
```

```
.org 010000#b

comds_0_7: 0110000001110000#b , if (eq = 0) then goto tm(cmd_modifier_mask);

                  "Output explanation:

                  Hold all the outputs to the same values used in the idle loop,
                  since execution of a command has not started yet.

                  Opcode explanation:

                  Based on address_16 and addresses 2..0, do a multi-way branch to a
                  command execution routine.
                  "

haltmode : 1110000001110010#b , if (not_select = 1) then goto pl(idle)  else wait;

                  "Output explanation:

                  This command takes only a single cycle to perform, so it is READY
                  to proceed in this cycle. The clock control command is to enter
                  the halt mode. All other outputs are at the default values.

                  Opcode explanation:

                  Return to idle when the host releases the bus, else wait. While
                  waiting it helps to keep the clock command active.
                  "

runmode  : 1110000001110011#b , if (not_select = 1) then goto pl(idle)  else wait;

                  "Output explanation:

                  This command takes only a single cycle to perform so it is READY
                  to proceed in this cycle. The clock control command is to enter
                  the run mode. All other outputs are at the default values.

                  Opcode explanation:

                  Return to idle when the host releases the bus, else wait. While
                  waiting it helps to keep the clock command active.
                  "

singlestp: 1110000001110100#b , if (eq = 0) then goto pl(host_ack);

                  "Output explanation:

                  This is a single cycle command so READY is made active in this
                  cycle. The clock control command will single step all clocks. All
                  other outputs are at the default values.

                  Opcode explanation:

                  Since the clock step command should occur for only one cycle, the
                  program branches to a loop that waits for the host to release the
                  bus. That loop has a no operation clock code.
                  "
```

```
ss_contrl: 1110000001110110#b , if (eq = 0) then goto pl(host_ack);

                  "Output explanation:

                  This is a single cycle command so READY is made active in this
                  cycle. The clock control command will single step the control
                  section clocks. All other outputs are at the default values.

                  Opcode explanation:

                  Since the clock step command should occur for only one cycle, the
                  program branches to a loop that waits for the host to release the
                  bus. That loop has a no operation clock code.
                  "

ss_data   : 1110000001110011#b , if (eq = 0) then goto pl(host_ack);

                  "Output explanation:

                  This is a single cycle command, so READY is made active in this
                  cycle. The clock control command will single step only the data
                  section clocks. All other outputs are at the default values.

                  Opcode explanation:

                  Since the clock step command should occur for only one cycle, the
                  program branches to a loop that waits for the host to release the
                  bus. That loop has a no operation clock code.
                  "

interrupt: 1110000001111010#b , if (not_select = 1) then goto pl(idle)  else wait;

                  "Output explanation:

                  Use the clock control code that causes a semiphore interrupt to
                  the Am29300 CPU. The READY signal is active to end the transac-
                  tion.

                  Opcode explanation:

                  Return to idle when the host releases the bus, else wait. While
                  waiting it helps to keep the clock command active.
                  "

reset_cpu: 1110000001111001#b , if (not_select = 1) then goto pl(idle)
                                 else wait;

                  "Output explanation:

                  This instruction causes a reset to the Am29300 while forcing all
                  the CPU control section clocks to step. READY is made active to
                  end the bus transaction. All other outputs are default.

                  Opcode explanation:

                  Return to idle when the host releases the bus, else wait. While
                  waiting it helps to keep the clock command active.
                  "
```

```
.org 000111#b

host_ack : 1110000001110000#b , if (not_select = 1) then goto pl(idle)  else wait;

                "Output explanation:

                Hold READY active, otherwise do nothing (default outputs).

                Opcode explanation:

                Wait for the host to release the bus then go to idle. "

.org 011000#b

comds_8_F: 0110000001110000#b , if (eq = 0) then goto tm(cmd_modifier_mask);

                "Output explanation:

                Hold all the outputs to the same values used in the idle loop
                since execution of a command has not started yet.

                Opcode explanation:

                Based on address_16 and addresses 2..0 do a multi-way branch to a
                command execution routine.
                "

load_pipe: 1110001001110101#b , if (eq = 0) then goto pl(host_ack);

                "Output explanation:

                This is a single cycle command so READY is made active  in this
                cycle. MODE is active so that the pipeline register will load from
                the SSR when the pipeline register is next clocked. The clock
                control command will single step  only the pipeline clocks. All
                other outputs are at the default values.

                Opcode explanation:

                Since the clock step command should occur for only one cycle the
                program branches to a loop that waits for the host to release the
                bus. That loop has a no operation clock code.
                "

load_mop : 1110001001110111#b , if (eq = 0) then goto pl(host_ack);

                "Output explanation:

                This is a single cycle command so READY is made active in this
                cycle. MODE is active so that the  macro opcode, status, and
                interrupt address pipeline registers will be loaded from the SSR
                at the next active edge of the pipeline clock. The clock control
                command will single step the above mentioned registers. All other
                outputs are at the  default values.
```

```
                    Opcode explanation:

                    Since the clock step command should occur for only one cycle the
                    program branches to a loop that waits for the host to release the
                    bus. That loop has a no operation clock code.
                    "

load_wcs : 0110001101110001#b , if (eq = 0) then goto pl(load_wcs2);

                    "Output explanation:

                    This is only the first instruction of this routine so READY is
                    inactive. MODE and SDI_SSR_MUX are made active in anticipation of
                    the next rising edge of DCLK_WCS. The CPU clocks are halted so
                    that the pipeline registers in the system will not load from the
                    shadow register instead of the normal inputs as would be directed
                    by the state of MODE and SDI_SSR_MUX. All other outputs are in
                    default state.

                    Opcode explanation:

                    Unconditional branch to second word of this routine that is lo-
                    cated elsewhere in the instruction memory.
                    "

.org 001011#b

load_wcs2: 0110001101110001#b , if (eq = 0) then load pl(7);

"Output explanation:

                    Still not READY. MODE and SDI_SSR_MUX remain active to propagate
                    through the SDI-SDO chain of the SSR. The CPU clocks continue to
                    be held to prevent changing the state of the pipeline register.
                    All other outputs are at the default values.

                    Opcode explanation:

                    Load the CREG with a value of 7. Continue.
                    "

load_wcs3: 0100001101110001#b , while (creg <> 0) loop to pl(load_wcs3);

                    "Output explanation:

                    Still not READY. MODE is active in anticipation of DCLK_WCS.
                    !WCS_WR is active to enable the output of the WCS_PORT so that the
                    address to the WCS will be stable when DCLK_WCS next goes active
                    to begin the WCS write cycle. SDI_SSR_MUX remains active to con-
                    tinue propagating through the SDI-SDO chain in the SSR. The CPU
                    clocks continue to be held to prevent changing the state of the
                    pipeline register. All other outputs are at the default values.
```

```
                Opcode explanation:

                Kill time by looping at this location for 8 more clocks. This
                allows time for the SDI_SSR_MUX signal to propagate _
                through the serial data in to serial data out (SDI-SDO) path
                through all the SSR registers in the pipeline register. Each
                register in the 9151 WCS memories has a SDI to SDO propagation
                delay of 35ns worst case. There are about 25 of these connected in
                series in addition to the ssr_port register and wcs_port register
                which add an additional 15ns each. The total delay needed to
                ensure a valid SDI_SSR_MUX signal at the last register in the
                chain is about 1000ns (10 clocks). Two clocks have occured prior
                to the current instruction and this instruction will loop for an
                additional 8 cycles.
                "

load_wcs4: 0101001101110001#b , continue;

                "Output explanation:

                Still not READY. !WCS_WR is active in order to write data into the
                macro opcode map RAM  at the same time data is written into the
                Writable Control Store. DCLK_WCS goes active to clock the SSR
                internal register that enables shadow register data to be driven
                out on the input pins (i.e. driven to the data I/O pins of the
                WCS). That same rising edge of DCLK_WCS causes the writing of data
                into the RAM of the AM9151s used in the WCS. MODE and SDI_SSR_MUX
                remain active to satisfy hold times relative to DCLK_WCS. The CPU
                clocks continue to be held to prevent changing the state of  the
                pipeline register. All other outputs are at the default values.

                Opcode explanation:

                Continue to the next instruction.
                "

load_wcs5: 0110000001110001#b , continue;

                "Output explanation:

                Still not READY. !WCS_WR goes inactive to complete the write of
                the macro opcode map RAM. DCLK_WCS goes inactive to prepare for
                going active again in the next cycle. MODE and SDI_SSR_MUX go
                inactive to meet the set-up time to the rising edge of DCLK_WCS in
                the next cycle. MODE being inactive will cause the reseting of the
                SSR internal flip flop (at the next rising edge of DCLK_WCS) which
                enables the shadow register data onto the input pins of the SSR.
                The CPU clocks remain halted. All other outputs are in the default
                state.

                Opcode explanation:

                Continue to next instruction. Note: since MODE goes to all SSR in
                parallel there is no long delay required to wait for SDI_SSR_MUX
                to propagate through the SSR chain. MODE being inactive is alone
                enough to reset the SSR internal control flip flop.
                "
```

```
load_wcs6: 1111000001110001#b , if (not_select = 1) then goto pl(idle)  else wait;

                  "Output explanation:

                  This is the last cycle of the routine and the READY signal is made
                  active. DCLK_WCS goes active to clear the WCS write activity. The
                  CPU clocks continue to be held. All other outputs have returned to
                  the default states.

                  Opcode explanation:

                  Return to idle when the host releases the bus, else wait.
                  "

.org 011100#b

load_init: 0110001101110001#b , if (eq = 0) then goto pl(load_int2);

                  "Output explanation:

                  This is only the first instruction of this routine so READY is
                  inactive. MODE and SDI_SSR_MUX are made active in anticipation of
                  the next rising edge of DCLK_WCS occurring in the next cycle. The
                  CPU clocks are halted so that the pipeline registers in the system
                  will not load from the shadow register instead of the normal
                  inputs as would be directed by the state of MODE and SDI_SSR_MUX.
                  All other outputs are in default state.

                  Opcode explanation:

                  Unconditional branch to second word of this routine that is lo-
                  cated elsewhere in the instruction memory.
                  "

.org 000010#b

load_int2: 0110001101110001#b , if (eq = 0) then load pl(7);

                  "Output explanation:

                  Still not READY.  MODE and SDI_SSR_MUX remain active to propagate
                  through the SDI-SDO chain of the SSR. The CPU clocks continue to
                  be held to prevent changing the state of  the pipeline register.
                  All other outputs are at the default values.

                  Opcode explanation:

                  Load the CREG with a value of 7. Continue.
                  "

load_int3: 0010001101110001#b , while (creg <> 0) loop to pl(load_int3);

                  "Output explanation:

                  Still not READY.  MODE is active in anticipation of DCLK_WCS.
                  SDI_SSR_MUX remains active to continue propagating through the
```

```
                 SDI-SDO chain in the SSR. The CPU clocks continue to be held to
                 prevent changing the state of  the pipeline register. !WCS_INIT is
                 active to meet set-up time to DCLK_WCS. All other outputs are at
                 the default values.

                 Opcode explanation:

                 Kill time by looping at this location for 8 more clocks. This
                 allows time for the SDI_SSR_MUX signal to propagate through the
                 serial data in to serial data out (SDI-SDO) path through all the
                 SSR registers in the pipeline register. Each register in the 9151
                 WCS memories has a SDI to SDO propagation delay of 35ns worst
                 case. There are about 25 of these connected in series in addition
                 to the ssr_port register and wcs_port register which add an addi-
                 tional 15ns each. The total delay needed to ensure a valid
                 SDI_SSR_MUX signal at the last register in the chain is about
                 1000ns (10 clocks). Two clocks have occured prior to the current
                 instruction and this instruction will loop for an additional 8
                 cycles.
                 "

load_int4: 0011001101110001#b , continue;

                 "Output explanation:

                 Still not READY. !WCS_INIT is active in order to write data into
                 the initialization register of the WCS. DCLK_WCS goes active to
                 clock the SSR internal register that enables shadow register data
                 to be driven out on the input pins (i.e. driven to the data I/O
                 pins of the WCS). That same rising edge of DCLK_WCS causes the
                 writing of data into the initialization register of the AM9151s
                 used in the WCS. MODE and SDI_SSR_MUX remain active to satisfy
                 hold times relative to DCLK_WCS. The CPU clocks continue to be
                 held to prevent changing the state of  the pipeline register. All
                 other outputs are at the default values.

                 Opcode explanation:

                 Continue to the next instruction.
                 "

load_int5: 0110000001110001#b , continue;

                 "Output explanation:

                 Still not READY.  DCLK_WCS goes inactive to prepare for going
                 active again in the next cycle. MODE, !WCS_INIT, and SDI_SSR_MUX
                 go inactive to meet the set-up time to the rising edge of DCLK_WCS
                 in the next cycle. These signals being inactive will cause the
                 reseting of the SSR internal flip flop (at the next rising edge of
                 DCLK_WCS) which enables the shadow register data onto the input
                 pins of the SSR. The CPU clocks remain halted. All other outputs
                 are in the default state.
```

```
                  Opcode explanation:

                  Continue to next instruction.
                  "

load_int6: 1111000001110001#b , if (not_select = 1) then goto pl(idle)  else wait;

                  "Output explanation:

                  This is the last cycle of the routine and the READY signal is made
                  active. DCLK_WCS goes active to clear the WCS write activity. The
                  CPU clocks continue to be held. All other outputs have returned to
                  the default states.

                  Opcode explanation:

                  Return to idle when the host releases the bus, else wait.
                  "

.org 011101#b

load_ssr : 0110001001110001#b , if (eq = 0) then goto pl(load_ssr2);

                  "Output explanation:

                  This is only the first instruction of this routine so READY is
                  inactive. MODE is made active and SDI_SSR_MUX is made inactive in
                  anticipation of the next rising edge of DCLK_WCS. The CPU clocks
                  are halted so that the pipeline registers in the system will not
                  load from the shadow register instead of the normal inputs as
                  would be directed by the state of MODE and SDI_SSR_MUX. All other
                  outputs are in default state.

                  Opcode explanation:

                  Unconditional branch to second word of this routine that is lo-
                  cated elsewhere in the instruction memory.
                  "

.org 111001#b

load_ssr2: 0110001001110001#b , if (eq = 0) then load pl(7);

                  "Output explanation:

                  Still not READY.  MODE and SDI_SSR_MUX remain as before to propa-
                  gate through the SDI-SDO chain of the SSR. The CPU clocks continue
                  to be held to prevent changing the state of  the pipeline regis-
                  ter. All other outputs are at the default values.

                  Opcode explanation:

                  Load the CREG with a value of 7. Continue.
                  "
```

```
load_ssr3: 0110001001110001#b , while (creg <> 0) loop to pl(load_ssr3);

                  "Output explanation:

                  Still not READY.  MODE is active in anticipation of DCLK_WCS.
                  SDI_SSR_MUX remains inactive to continue propagating through the
                  SDI-SDO chain in the SSR. The CPU clocks continue to be held to
                  prevent changing the state of  the pipeline register. All other
                  outputs are at the default values.

                  Opcode explanation:

                  Kill time by looping at this location for 8 more clocks. This
                  allows time for the SDI_SSR_MUX signal to propagate through the
                  serial data in to serial data out (SDI-SDO) path through all the
                  SSR registers in the pipeline register. Each register in the 9151
                  WCS memories has a SDI to SDO propagation delay of 35ns worst
                  case. There are about 25 of these connected in series in addition
                  to the ssr_port register and wcs_port register which add an addi-
                  tional 15ns each. The total delay needed to ensure a valid
                  SDI_SSR_MUX signal at the last register in the chain is about
                  1000ns (10 clocks). Two clocks have occured prior to the current
                  instruction and this instruction will loop for an additional 8
                  cycles.
                  "

load_ssr4: 0111101001110001#b , if (not_select = 1) then goto pl(idle)  else wait;

                  "Output explanation:

                  Still not READY.  DCLK_WCS and DCLK_MOP go active to load pipeline
                  data into the shadow register throughout the diagnostic registers.
                  MODE and SDI_SSR_MUX remain as before to satisfy hold times rela-
                  tive to DCLK. The CPU clocks continue to be held to prevent chang-
                  ing the state of  the pipeline register. All other outputs are at
                  the default values.

                  Opcode explanation:

                  Return to idle when the host releases the bus, else wait. While
                  waiting it doesn't hurt to keep the clock command active.
                  "

.org 011110#b

shift_wcs: 0110000001110001#b , if (eq = 0) then goto pl(sh_wcs2);

                  "Output explanation:

                  Not READY. System clocks halted.

                  Opcode explanation:

                  Branch to remaining words of the routine.
                  "
```

```
.org 101001#b

sh_wcs2  : 0110000001110001#b , if (eq = 0) then load tm(count_mask);

                    "Output explanation:

                    Maintain last output.

                    Opcode explanation:

                    Load the CREG with the count for the number of nibbles to be
                    shifted. This count value is contained in address bits 2..0.
                    "

sh_wcs3  : 0110000001110001#b , if (eq = 0) then dec;

                    "Output explanation:

                    Maintain last output.

                    Opcode explanation:

                    Decrement the CREG so that the loop count for nibbles will be n-1.
                    This is required since the CREG = 0 comparison in a loop is only
                    made at the end of a loop, thus count values must be the number of
                    loops desired -1. It is assumed that the host loads the count for
                    the actual number of nibbles to be shifted.
                    "

sh_wcs4  : 0110000101110001#b , if (eq = 0) then load pl(3), nested;

                    "Output explanation:

                    SDI_SSR_MUX is active to meet the set-up time to DCLKs that will
                    occur in the next cycle. With MODE being inactive, SDI_SSR_MUX
                    will control the multiplexer at the input of the ssr_port to
                    select the WCS SSR chain as input. System clocks are still held.

                    Opcode explanation:

                    Load the inner loop count for shifting a nibble into CREG and push
                    the nibble count into SREG.
                    "

sh_wcs5  : 0111010101110001#b , continue;

                    "Output explanation:

                    DCLK_WCS and DCLK_SSR go active together to shift one bit of a
                    nibble from the ssr_port to the WCS SSR and also to shift one bit
                    from the end of the WCS SSR chain into the ssr_port. SDI_SSR_MUX
                    remains active to select the WCS SSR chain as the input to the
                    ssr_port. System clocks are held.
```

```
                  Opcode explanation:

                  Continue to next instruction.
                  "

sh_wcs6    : 0110000101110001#b , while (CREG <> 0) loop to pl(sh_wcs5)   else nest;

                  "Output explanation:

                  DCLK_WCS and DCLK_SSR go inactive in preperation for going active
                  again at the next shift of the SSR. SDI_SSR_MUX remains active.
                  System clocks are halted.

                  Opcode explanation:

                  Loop on the SSR shift operation for 4 cycles then drop out.
                  "

sh_wcs7    : 0110000101110001#b , while (CREG <> 0) loop to pl(sh_wcs4);

                  "Output explanation:

                  Maintain last output.

                  Opcode explanation:

                  For the number of nibbles in the shift count, loop through the
                  nibble shift routine. Then fall through.
                  "

sh_wcs8    : 1110000001110000#b , if (not_select = 1) then goto pl(idle)  else wait;

                  "Output explanation:

                  Last instruction so READY is active.

                  Opcode explanation:

                  Wait for the host to release the bus then go to idle.
                  "

.org 011111#b

shift_mop: 0110000001110001#b , if (eq = 0) then goto pl(sh_mop2);

                  "Output explanation:

                  Not READY. System clocks halted.

                  Opcode explanation:

                  Branch to remaining words of the routine.
                  "
```

```
.org 110001#b

sh_mop2  : 0110000001110001#b , if (eq = 0) then load tm(count_mask);

              "Output explanation:

              Maintain last output.

              Opcode explanation:

              Load the CREG with the count for the number of nibbles to be
              shifted.
              "

sh_mop3  : 0110000001110001#b , if (eq = 0) then dec;

              "Output explanation:

              Maintain last output.

              Opcode explanation:

              Decrement the CREG so that the loop count for nibbles will be n-1.
              This is required since the CREG <> 0 comparison in a loop is only
              made at the end of a loop, thus count values must be the number of
              loops desired -1. It is assumed that the host loads the count for
              the actual number of nibbles to be shifted.
              "

sh_mop4  : 0110000001110001#b , if (eq = 0) then load pl(3), nested;

              "Output explanation:

              SDI_SSR_MUX is inactive to meet the set-up time to DCLKs that will
              occur in the next cycle. With MODE being inactive, SDI_SSR_MUX
              will control the multiplexer at the input of the ssr_port to
              select the MOP SSR chain as input. System clocks are still held.

              Opcode explanation:

              Load the inner loop count for shifting a nibble into CREG and push
              the nibble count into SREG.
              "

sh_mop5  : 0110110001110001#b , continue;

              "Output explanation:

              DCLK_MOP and DCLK_SSR go active together to shift one bit of a
              nibble from the ssr_port to the MOP SSR and also to shift one bit
              from the end of the MOP SSR chain into the ssr_port. SDI_SSR_MUX
              remains inactive to select the MOP SSR chain as the input to the
              ssr_port. System clocks are held.
```

```
                    Opcode explanation:

                    Continue to next instruction.
                    "

sh_mop6  : 0110000001110001#b , while (CREG <> 0) loop to pl(sh_mop5)   else nest;

                    "Output explanation:

                    DCLK_MOP and DCLK_SSR go inactive in preperation for going active
                    again at the next shift of the SSR. SDI_SSR_MUX remains inactive.
                    System clocks are halted.

                    Opcode explanation:

                    Loop on the SSR shift operation for 4 cycles then drop out.
                    "

sh_mop7  : 0110000001110001#b , while (CREG <> 0) loop to pl(sh_mop4);

                    "Output explanation:

                    Maintain last output.

                    Opcode explanation:

                    For the number of nibbles in the shift count loop, go through the
                    nibble shift routine. Then fall through.
                    "

sh_mop8  : 1110000001110000#b , if (not_select = 1) then goto pl(idle)  else wait;

                    "Output explanation:

                    Last instruction so READY is active.

                    All other outputs default.

                    Opcode explanation:

                    Wait for the host to release the bus then go to idle. "

End.
"
```

Memory Map:

Location			Label	Location			Label
Dec	Hex	Bin		Dec	Hex	Bin	
0	00	000000	memory	32	20	100000	idle
1	01	000001	memory2	33	21	100001	
2	02	000010	load_int2	34	22	100010	
3	03	000011	load_int3	35	23	100011	
4	04	000100	load_int4	36	24	100100	
5	05	000101	load_int5	37	25	100101	
6	06	000110	load_int6	38	26	100110	
7	07	000111	host_ack	39	27	100111	
8	08	001000	ssr_port	40	28	101000	idle2
9	09	001001	ssr_write	41	29	101001	sh_wcs2
10	0A	001010	ssr_read	42	2A	101010	sh_wcs3
11	0B	001011	load_wcs2	43	2B	101011	sh_wcs4
12	0C	001100	load_wcs3	44	2C	101100	sh_wcs5
13	0D	001101	load_wcs4	45	2D	101101	sh_wcs6
14	0E	001110	load_wcs5	46	2E	101110	sh_wcs7
15	0F	001111	load_wcs6	47	2F	101111	sh_wcs8
16	10	010000	cmd_0_7	48	30	110000	idle3
17	11	010001	haltmode	49	31	110001	sh_mop2
18	12	010010	runmode	50	32	110010	sh_mop3
19	13	010011	singlestp	51	33	110011	sh_mop4
20	14	010100	ss_contrl	52	34	110100	sh_mop5
21	15	010101	ss_data	53	35	110101	sh_mop6
22	16	010110	interrupt	54	36	110110	sh_mop7
23	17	010111	reset_cpu	55	37	110111	sh_mop8
24	18	011000	cmd_8_F	56	38	111000	idle4
25	19	011001	load_pipe	57	39	111001	load_ssr2
26	1A	011010	load_mop	58	3A	111010	load_ssr3
27	1B	011011	load_wcs	59	3B	111011	load_ssr4
28	1C	011100	load_init	60	3C	111100	
29	1D	011101	load_ssr	61	3D	111101	
30	1E	011110	sh_wcs	62	3E	111110	
31	1F	011111	sh_mop	63	3F	111111	reset

APPENDIX F
Memory Address Counter PAL Definition

```
"Advanced Micro Devices Application Note:
"
"Am29300 Demonstration System
"
"By Mark Mc Clain, Field Applications Engineer, San Diego, CA.
"(619)560-7030, Date = 1/87

Module
   Memory_Address_Counter_A;

Flag '-r3'

Title
'Memory Address Counter PAL for an Am29300 Demonstration System.';

memad_a device 'P22V10';

"declarations

X,Z,C,P = .X.,.Z.,.C.,.P.;

"Signal names that end in an underline indicate an active low signal.

CLK_D, P_MEM_3, P_MEM_2, P_MEM_1, P_MEM_0, CASIN_  Pin
   1,     2,       3,       4,       5,       6;

AIN_6, AIN_5, AIN_4, AIN_3, AIN_2, AIN_1, AIN_0  Pin
  7,     8,     9,     10,    11,    13,    14;

AOUT_6, AOUT_5, AOUT_4, AOUT_3, AOUT_2, AOUT_1, AOUT_0, CASOUT_  Pin
  18,     19,     20,     17,     21,     16,     22,     23;

ripple Pin
   15;

"  Some outputs are declared as active high. This requires that ABEL
"  version 2.0 or later be used to compile this definition. Earlier
"  versions of ABEL have a bug that assumes all programmable pins in the
"  22V10 are active low regardless of how they are declared. Earlier
"  versions of ABEL will generate an incorrect JEDEC file.

AOUT_6, AOUT_5, AOUT_4, AOUT_3, AOUT_2, AOUT_1, AOUT_0 Istype 'pos, reg';

CASOUT_
      Istype 'neg, com';

ripple
      Istype 'pos, com';

" declare some sets

count_msb =
      [AOUT_6, AOUT_5, AOUT_4];
```

```
count_lsb =
      [AOUT_3, AOUT_2, AOUT_1, AOUT_0];

data_in_msb =
      [AIN_6, AIN_5, AIN_4];

data_in_lsb =
      [AIN_3, AIN_2, AIN_1, AIN_0];

select =     [P_MEM_1, P_MEM_0];

cmd =        [P_MEM_3, P_MEM_2];

nop          = ^B00;
a_bus        = ^B01;
cntr_a       = ^B10;
cntr_b       = ^B11;

load_a       = ^B00;
load_b       = ^B01;
inc          = ^B10;
dec          = ^B11;

" declare a macro

EQUATIONS
"  This is a loadable, cascadable, 7 bit up/down counter that decodes
"  its own output enable, count enable, direction, and load signals
"  directly from the control pipeline bits. The counter is programmed as
"  either an A counter or a B counter and will only operate with the
"  appropriate command. Cascade-in and Cascade-out are used to cascade
"  the counters. As used in the Am29300 demonstration system, the data
"  input pins and output pins are tied together with the A_BUS.

"  The equations for the high order bits of a 7 bit counter require more
"  product terms than are available in any of the 22V10 outputs.So, the
"  counter is internally split into a 4 bit and 3 bit counter that have
"  a ripple carry between them.

ripple =

     (cmd == inc) and (count_lsb == ^b1111) #
     (cmd == dec) and (count_lsb == ^b0000) ;

"  The counter must be programmed as either an A counter or a B
"  counter because the output enable is limited to a single product
"  term.

enable count_msb = (select == cntr_a);
enable count_lsb = (select == cntr_a);
```

```
count_msb :=
                          (select == nop    )            & count_msb       #
   (cmd == load_a)    & (select == a_bus )  & data_in_msb                  #
   (cmd == load_b)    & (select == a_bus )             & count_msb       #
   (cmd == inc    )    & (select == a_bus )             & count_msb       #
   (cmd == dec    )    & (select == a_bus )             & count_msb       #
   (cmd == load_a)    & (select == cntr_a)             & count_msb       #
   (cmd == load_b)    & (select == cntr_a)             & count_msb       #
   (cmd == inc)       & (select == cntr_a) & ripple & ( count_msb + 1) #
   (cmd == dec)       & (select == cntr_a) & ripple & ( count_msb - 1) #
                        (select == cntr_b)  & count_msb                  ;

count_lsb :=

                          (select == nop    ) & count_lsb                  #
   (cmd == load_a)    & (select == a_bus )  & data_in_lsb                  #
   (cmd == load_b)    & (select == a_bus )             & count_lsb       #
   (cmd == inc    )    & (select == a_bus )             & count_lsb       #
   (cmd == dec    )    & (select == a_bus )             & count_lsb       #
   (cmd == load_a)    & (select == cntr_a)             & count_lsb       #
   (cmd == load_b)    & (select == cntr_a)             & count_lsb       #
   (cmd == inc)       & (select == cntr_a)  & !CASIN_  & (count_lsb + 1)#
   (cmd == dec)       & (select == cntr_a)  & !CASIN_  & (count_lsb - 1)#
                        (select == cntr_b)  & count_lsb                  ;

!CASOUT_ =

   (cmd == inc)   & (count_msb == ^b111)   & (count_lsb == ^b1111)    #
   (cmd == dec)   & (count_msb == ^b000)   & (count_lsb == ^b0000)    ;

"  Test_Vectors need to be defined.

End;

Module
   Memory_Address_Counter_B;

Flag '-r3'

Title
'Memory Address Counter PAL for an Am29300 Demonstration System.';

memad_b device 'P22V10';

"declarations

X,Z,C,P = .X.,.Z.,.C.,.P.;
```

```
"Signal names that end in an underline indicate an active low signal.

CLK_D, P_MEM_3, P_MEM_2, P_MEM_1, P_MEM_0, CASIN_  Pin
   1,      2,        3,        4,        5,       6;

AIN_6, AIN_5, AIN_4, AIN_3, AIN_2, AIN_1, AIN_0  Pin
  7,      8,      9,     10,     11,     13,     14;

AOUT_6, AOUT_5, AOUT_4, AOUT_3, AOUT_2, AOUT_1, AOUT_0, CASOUT_  Pin
  18,     19,      20,      17,      21,     16,      22,      23;

ripple Pin
   15;

"  Some outputs are declared as active high. This requires that ABEL
"  version 2.0 or later be used to compile this definition. Earlier
"  versions of ABEL have a bug that assumes all programmable pins in the
"  22V10 are active low regardless of how they are declared. Earlier
"  versions of ABEL will generate an incorrect JEDEC file.

AOUT_6, AOUT_5, AOUT_4, AOUT_3, AOUT_2, AOUT_1, AOUT_0 Istype 'pos, reg';

CASOUT_
      Istype 'neg, com';

ripple
      Istype 'pos, com';

" declare some sets

count_msb =
      [AOUT_6, AOUT_5, AOUT_4];

count_lsb =
      [AOUT_3, AOUT_2, AOUT_1, AOUT_0];

data_in_msb =
      [AIN_6, AIN_5, AIN_4];

data_in_lsb =
      [AIN_3, AIN_2, AIN_1, AIN_0];

select =     [P_MEM_1, P_MEM_0];

cmd =        [P_MEM_3, P_MEM_2];

nop          = ^B00;
a_bus        = ^B01;
cntr_a       = ^B10;
cntr_b       = ^B11;

load_a       = ^B00;
load_b       = ^B01;
inc          = ^B10;
dec          = ^B11;
```

```
" declare a macro

EQUATIONS
"  This is a loadable, cascadable, 7 bit up/down counter that decodes
"  its own output enable, count enable, direction, and load signals
"  directly from the control pipeline bits. The counter is programmed as
"  either an A counter or a B counter and will only operate with the
"  appropriate command. Cascade-in and Cascade-out are used to cascade
"  the counters. As used in the Am29300 demonstration system, the data
"  input pins and output pins are tied together with the A_BUS.

"  The equations for the high order bits of a 7 bit counter require more
"  product terms than are available in any of the 22V10 outputs, so the
"  counter is internally split into a 4 bit and 3 bit counter that have
"  a ripple carry between them.

ripple =

     (cmd == inc) and (count_lsb == ^b1111) #
     (cmd == dec) and (count_lsb == ^b0000) ;

"  The counter must be programmed as either an A counter or a B
"  counter because the output enable is limited to a single product
"  term.

enable count_msb = (select == cntr_b);
enable count_lsb = (select == cntr_b);

count_msb :=
                        (select == nop    )           & count_msb      #
   (cmd == load_b)    & (select == a_bus )  & data_in_msb              #
   (cmd == load_a)    & (select == a_bus )            & count_msb      #
   (cmd == inc    )   & (select == a_bus )            & count_msb      #
   (cmd == dec    )   & (select == a_bus )            & count_msb      #
   (cmd == load_a)    & (select == cntr_b)            & count_msb      #
   (cmd == load_b)    & (select == cntr_b)            & count_msb      #
   (cmd == inc)       & (select == cntr_b) & ripple & ( count_msb + 1) #
   (cmd == dec)       & (select == cntr_b) & ripple & ( count_msb - 1) #
                        (select == cntr_a)  & count_msb                ;

count_lsb :=

                        (select == nop    ) & count_lsb                #
   (cmd == load_b     & (select == a_bus )  & data_in_lsb              #
   (cmd == load_a)    & (select == a_bus )            & count_lsb      #
   (cmd == inc    )   & (select == a_bus )            & count_lsb      #
   (cmd == dec    )   & (select == a_bus )            & count_lsb      #
   (cmd == load_a)    & (select == cntr_b)            & count_lsb      #
   (cmd == load_b)    & (select == cntr_b)            & count_lsb      #
   (cmd == inc)       & (select == cntr_b)  & !CASIN_  & (count_lsb + 1)#
   (cmd == dec)       & (select == cntr_b)  & !CASIN_  & (count_lsb - 1)#
                        (select == cntr_a   & count_lsb                ;
```

```
!CASOUT_ =

   (cmd == inc)   & (count_msb == ^b111)   & (count_lsb == ^b1111)    #
   (cmd == dec)   & (count_msb == ^b000)   & (count_lsb == ^b0000)    ;

" Test_Vectors need to be defined.

End;
```

APPENDIX G

Macro Operand Counter PAL Definition

```
"Advanced Micro Devices Application Note:
"
"Am29300 Demonstration System
"
"By Mark Mc Clain, Field Applications Engineer, San Diego, CA.
"(619)560-7030, Date = 1/87

Module
   Macro_Operand_Counter;

Flag '-r3'

Title
'Macro Operand Counter PAL for an Am29300 Demonstration System.';

macop device 'P22V10';

"declarations

   X,Z,C,P = .X.,.Z.,.C.,.P.;

   "Signal names that end in an underline indicate an active low signal.

   CLK_CNTL, P_LD_CNT, P_SEL_0, P_SEL_1, P_UP_DN, P_CNT_EN Pin
       1,        2,        3,       4,       5,        6;

   AIN_5, AIN_4, AIN_3, AIN_2, AIN_1, AIN_0  Pin
     7,     8,     9,     10,    11,    13;

   AOUT_5, AOUT_4, AOUT_3, AOUT_2, AOUT_1, AOUT_0, reload  Pin
     18,     19,     20,     17,     21,     16,     22;

   valid_cmd Pin
      14;

"  Some outputs are declared as active high. This requires that ABEL
"  version 2.0 or later be used to compile this definition. Earlier
"  versions of ABEL have a bug that assumes all programmable pins in the
"  22V10 are active low regardless of how they are declared. Earlier
"  versions of ABEL will generate an incorrect JEDEC file.

AOUT_5, AOUT_4, AOUT_3, AOUT_2, AOUT_1, AOUT_0
   Istype 'pos, reg';

reload, valid_cmd
   Istype 'pos, com';

" declare some sets

   count =
      [AOUT_5, AOUT_4, AOUT_3, AOUT_2, AOUT_1, AOUT_0];
```

```
   data_in =
      [AIN_5, AIN_4, AIN_3, AIN_2, AIN_1, AIN_0];

   select = [P_SEL_1, P_SEL_0];

   nop      = ^B00;
   a_add    = ^B01;
   b_add    = ^B10;
   c_mac    = ^B11;

"  declare a macro for decrement - because ABEL is too dumb to do it
"  right for a counter bigger than 4 bits.

dec_count macro {

  [

   (!AOUT_5 & !AOUT_4 & !AOUT_3 & !AOUT_2 & !AOUT_1 & !AOUT_0 # AOUT_5 &
    AOUT_4 #
    AOUT_5 &  AOUT_3 #
    AOUT_5 &  AOUT_2 #
    AOUT_5 &  AOUT_1 #
    AOUT_5 &  AOUT_0 ),

   (!AOUT_4 & !AOUT_3 & !AOUT_2 & !AOUT_1 & !AOUT_0 # AOUT_4 &  AOUT_3 #
    AOUT_4 &  AOUT_2 #
    AOUT_4 &  AOUT_1 #
    AOUT_4 &  AOUT_0 ),

   (!AOUT_3 & !AOUT_2 & !AOUT_1 & !AOUT_0 # AOUT_3 &  AOUT_2 #
    AOUT_3 &  AOUT_1 #
    AOUT_3 &  AOUT_0 ),

   (!AOUT_2 & !AOUT_1 & !AOUT_0 #
    AOUT_2 &  AOUT_1 #
    AOUT_2 &  AOUT_0 ),

   (!AOUT_1 & !AOUT_0 #
    AOUT_1 &  AOUT_0 ),

   (!AOUT_0)

  ]};

EQUATIONS

"  This is a loadable, 6 bit up/down counter that decodes its own output
"  enable, count enable, direction, and load signals directly from the
"  control pipeline bits. When the max count is reached in increment
"  mode, the next increment will reload the counter from the input
"  value. When zero is reached in decrement mode, the next decrement
"  will also reload the counter.
```

```
"  The pipeline control bits for the counter come from an overlapped
"  field in the control word. It is necessary to disable (ignore) the
"  counter control bits when the field meaning is not valid. The enable
"  for the control bits is the OR of the P_C_SEL bits in the pipeline
"  (when the counter is used for the A or B address, these two select
"  inputs are simply both tied to the single control enable for the A or
"  B address). Since the enable is an OR function, it is necessary to
"  conserve product terms in the counter equations by performing the OR
"  function as a separate output which is fed back as an enable for the
"  counter command bits.

valid_cmd =

   P_SEL_0 # P_SEL_1;

"  The reload signal is active for the max or min count value in
"  increment or decrement mode, respectively.

reload =

   ( P_UP_DN & (count == ^b111111)) #
   (!P_UP_DN & (count == ^b000000)) ;

enable count = (select == c_mac);

count :=

      !valid_cmd & (count     )  &  !P_LD_CNT                                  #
       valid_cmd & (count     )  &  !P_LD_CNT   &  !P_CNT_EN                   #
       valid_cmd & (count + 1)   &   P_UP_DN    &  P_CNT_EN & !reload
                                 &  !P_LD_CNT                                  #
       valid_cmd & (dec_count)   & !P_UP_DN     &  P_CNT_EN & !reload
                                 & !P_LD_CNT                                   #
       valid_cmd & (data_in   )  & !P_LD_CNT    &  P_CNT_EN &  reload          #
                   (data_in   )  &  P_LD_CNT                                   ;

"  Test_Vectors need to be defined.

End;
```

APPENDIX H

Write Port A Multiplexer PAL Definition

```
" Advanced Micro Devices Application Note:
"
" Am29300 Demonstration System
"
" By Mark Mc Clain, Field Applications Engineer, San Diego, CA.
" (619)560-7030, Date = 1/87

Module
   Hex_Four_Input_Mux;

Title
'One half of a hex four input multiplexer PAL for an Am29300
 Demonstration System.';

hfmux device 'P18P8';

"declarations

   "Signal names that end in an underline indicate an active low signal.

   AIN_2, AIN_1, AIN_0  Pin
     1,      2,     3;
   BIN_2, BIN_1, BIN_0  Pin
     4,      5,     6;
   CIN_2, CIN_1, CIN_0  Pin
     7,      8,     9;
   DIN_2, DIN_1, DIN_0  Pin
     11,    12,    13;
   YOUT_2, YOUT_1, YOUT_0  Pin
     18,      17,      16;
   SELECT_0, SELECT_1 Pin
     14,        15;
   YOUT_2, YOUT_1, YOUT_0

     Istype 'pos, com';

" declare some sets

   Y =
      [YOUT_2, YOUT_1, YOUT_0];

   A =
      [AIN_2, AIN_1, AIN_0];

   B =
      [BIN_2, BIN_1, BIN_0];
```

```
    C =
        [CIN_2, CIN_1, CIN_0];

    D =
        [DIN_2, DIN_1, DIN_0];

    select =
        [SELECT_1, SELECT_0];

EQUATIONS

Y =

    A & (select == 0) # B & (select == 1)#
    C & (select == 2) # D & (select == 3);

" Test_Vectors need to be defined.

End;
```

APPENDIX I

Write Port B Multiplexer PAL Definition

```
" Advanced Micro Devices Application Note:
"
" Am29300 Demonstration System
"
" By Mark Mc Clain, Field Applications Engineer, San Diego, CA.
" (619)560-7030, Date = 1/87

Module
   Hex_Two_Input_Mux;

Title
'Hex two input multiplexer PAL for an Am29300 Demonstration System.';

htmux device 'P22V10';

"declarations

   X,Z,C,P = .X.,.Z.,.C.,.P.;

   "Signal names that end in an underline indicate an active low signal.

   AIN_5, AIN_4, AIN_3, AIN_2, AIN_1, AIN_0  Pin
     2,      3,      4,      5,      6,      7;

   BIN_5, BIN_4, BIN_3, BIN_2, BIN_1, BIN_0  Pin
     8,      9,      10,     11,     13,     14;

   YOUT_5, YOUT_4, YOUT_3, YOUT_2, YOUT_1, YOUT_0  Pin
     23,     22,        21,      20,      19,      18;

   SELECT Pin
      1;

"  Some outputs are declared as active high. This requires that ABEL
"  version 2.0 or later be used to compile this definition. Earlier
"  versions of ABEL have a bug that assumes all programmable pins in the
"  22V10 are active low regardless of how they are declared. Earlier
"  versions of ABEL will generate an incorrect JEDEC file.

   YOUT_5, YOUT_4, YOUT_3, YOUT_2, YOUT_1, YOUT_0
      Istype 'pos, com';
```

```
" declare some sets

   Y =
      [YOUT_5, YOUT_4, YOUT_3, YOUT_2, YOUT_1, YOUT_0];

   A =
      [AIN_5, AIN_4, AIN_3, AIN_2, AIN_1, AIN_0];

   B =
      [BIN_5, BIN_4, BIN_3, BIN_2, BIN_1, BIN_0];

EQUATIONS

   Y =

      A & !SELECT # B & SELECT;

" Test_Vectors need to be defined.

End;
```

APPENDIX J

Trap Logic PAL Definition

```
" Am29300 Demonstration System
"
" By Mark Mc Clain, Field Applications Engineer, San Diego, CA.
" (619)560-7030, Date = 1/87

Module
   Trap_Logic;

Title
'Trap Logic PAL for an Am29300 Demonstration System.';

trap device 'P22V10';

"declarations

   X,Z,C,P = .X.,.Z.,.C.,.P.;

   "Signal names that end in an underline indicate an active low signal.

   CLK_CNTL, MINTR_, EQUAL, P_FC_, RESET_300_, INTA_ Pin
       1,       2,       3,     4,        5,         6;

   INTR, TRAP, SEQ_FC, SEQ_CIN_, CASOUT2, MC_ADD_3, MC_ADD_2 Pin
    23,   22,     21,      20,        19,       18,        17;

   MC_ADD_1, MC_ADD_0 Pin
      16,       15;

"  Some outputs are declared as active high. This requires that ABEL
"  version 2.0 or later be used to compile this definition. Earlier
"  versions of ABEL have a bug that assumes all programmable pins in the
"  22V10 are active low regardless of how they are declared. Earlier
"  versions of ABEL will generate an incorrect JEDEC file.

   INTR, TRAP
      Istype 'pos, reg';

   SEQ_FC, CASOUT2, MC_ADD_3, MC_ADD_2, MC_ADD_1, MC_ADD_0
      Istype 'pos, com';

   SEQ_CIN_
      Istype 'neg, com';

" declare some sets

   mc_add =
      [MC_ADD_2, MC_ADD_1, MC_ADD_0];

EQUATIONS

"  OR the interrupt controller's interrupt request with the breakpoint
"  trap event signal to form the Sequencer's interrupt request.
"  Including  !TRAP with EQUAL will allow executing a breakpoint even on
"  the first instruction of the breakpoint trap routine. If this disable
"  of the trap were not included, and the trap vector address were the
"  same as the breakpoint address, then the system would get stuck
```

```
"   forever trying to trap on the trap vector address.

    INTR :=

       !MINTR_ # EQUAL & !TRAP;

"   Equal causes the breakpoint trap. Note that the Equal signal is
"   allowed or disallowed by setting or resetting the Equal comparator
"   register in the Sequencer. This allows breakpoint traps to be
"   disabled.

    TRAP :=

       EQUAL & !TRAP;

"   Disable the current Sequencer instruction on a TRAP or when the
"   pipeline Force Continue bit is active.

    SEQ_FC =

       TRAP # !P_FC_;

"   Don't increment the address stored on the stack after a trap so a
"   return from trap goes back to the trapped instruction. During reset
"   disable the incrementer so that the reset address will be zero
"   instead of one.

    !SEQ_CIN_ =

       TRAP # !RESET_300_;

"   When a trap comes, disable the interrupt controller from generating
"   an interrupt in the same cycle.

    CASOUT2 =

       EQUAL & !TRAP;

"   The trap logic always provides the fourth bit of the vector when the
"   interrupt or trap is acknowledged.

    enable MC_ADD_3 = !INTA_;

"   When a trap occurs, the fouth bit of the interrupt vector is active
"   otherwise it will be inactive.

    MC_ADD_3 =

       TRAP;

"   The LSB bits of the vector for the breakpoint trap are zero. Enable
"   these bits when a trap is acknowledged.

    enable mc_add = TRAP & !INTA_;

mc_add =

       !TRAP;

"   Test_Vectors need to be defined.

End;_
```

APPENDIX K

Clock Qualification PAL Definition

```
" Advanced Micro Devices Application Note:
"
" Am29300 Demonstration System
" Clock Qualification Logic Definition
"
" By Mark Mc Clain, Field Applications Engineer, San Diego, CA.
" (619)560-7030, Date = 1/87

Module
   Clock_Qualifier_1;

Title
'Clock Qualifying Control PAL for an Am29300 Demonstration System. This circuit
combines signals from the system control pipeline register and the host interface
controller to generate clock enables';

clkpal device 'P22V10';

"declarations

   X,Z,C,P = .X.,.Z.,.C.,.P.;

   "Signal names that end in an underline indicate an active low signal.

   CLK_FREE_RUN, CLK_CONTROL_3, CLK_CONTROL_2, CLK_CONTROL_1  Pin
         1,             2,             3,             4;

   CLK_CONTROL_0,  TRAP,  P_LD_MAC_STAT,  P_LD_MAC_OP  Pin
         5,           6,           7,             8;

   P_FC_,  P_LD_INT_BASE Pin
     9,          10;

   INTB_EN,  PIPE_EN,  MOP_EN, STAT_EN,   SEQ_EN, RESET_300_ Pin
      14,        15,       16,       17,       18,       19;

   INT_CPU_,  CNTL_EN, D_EN, haltmode Pin

      20,    21,    22,    23;

"  Some outputs are declared as active high. This requires that ABEL
"  version 2.0 or later be used to compile this definition. Earlier
"  versions of ABEL have a bug that assumes all programmable pins in the
"  22V10 are active low regardless of how they are declared.
"  Earlier versions of ABEL will generate an incorrect JEDEC file.

   haltmode
      Istype 'pos, reg';

   CNTL_EN, PIPE_EN, MOP_EN, STAT_EN,
   INTB_EN, SEQ_EN, D_EN
      Istype 'pos, com';
```

```
  RESET_300_, INT_CPU_
     Istype 'neg, com';

  clk_cntl = [CLK_CONTROL_3, CLK_CONTROL_2, CLK_CONTROL_1,
              CLK_CONTROL_0];

" CLK_CONTROL lines encoded meanings:

  nop            = ^b0000;
  ss_halt        = ^b0001;
  halt           = ^b0010;
  run            = ^b0011;
  ss             = ^b0100;
  ld_pipe        = ^b0101;
  ss_cntl        = ^b0110;
  ld_mop         = ^b0111;
  ss_data        = ^b1000;
  ss_reset       = ^b1001;
  int_cpu        = ^b1010;
  reserved1      = ^b1011;
  reserved2      = ^b1100;
  reserved3      = ^b1101;
  reserved4      = ^b1110;
  reserved5      = ^b1111;
  "

Equations

"  The haltmode signal is the output of a status flip flop which keeps
"  track of when the system has been placed in the halt mode, where all
"  clocks in the system are stopped. The flip flop is set when a halt
"  command appears on the CLK_CONTROL lines. It is reset by a run
"  command on the same lines.

   haltmode :=

      (clk_cntl == halt)         #
      haltmode & !(clk_cntl == run);

"  Each of the following outputs acts as an enable on the respective
"  qualified clock in the Am29300 system. Qualified clocks in this
"  system are held inactive in the high state.

"  The system clock generator produces an active low clock and the
"  enables are active high. By using negative logic OR gates (NAND
"  gates) the clock and enable signals are logically ORed together to
"  produce active high qualified clocks. The negative logic OR gates are
"  external to the PAL defined here.

"  The data section clock enable is active whenever haltmode is not
"  active and there is no single cycle halt command or trap operation
"  active. If haltmode is active, the enable can be forced active by a
"  single step command, a reset command, or a single step data section
"  command. If a trap operation is active the enable can be forced
"  active only by a reset command, or a single step data section
"  command.
```

```
   D_EN =
      !haltmode    &  !(clk_cntl == ss_halt)  &  !TRAP   #
       haltmode    &   (clk_cntl == ss      ) &  !TRAP   #
       haltmode    &   (clk_cntl == ss_reset)            #
       haltmode    &   (clk_cntl == ss_data )            ;

"  The control section clock enable is active whenever haltmode is not
"  active and there is no single cycle halt command or trap active. If
"  haltmode is active the enable can be forced active by a single step
"  command, a reset command, or a single step control section command.
"  If a trap is active the enable can be forced by a reset or single
"  step control section command.

   CNTL_EN =
      !haltmode    &  !(clk_cntl == ss_halt)  &  !TRAP   #
       haltmode    &   (clk_cntl == ss      ) &  !TRAP   #
       haltmode    &   (clk_cntl == ss_reset) #
       haltmode    &   (clk_cntl == ss_cntl ) ;

"  The Sequencer clock is the same as the control section clock except
"  that it is unaffected by a trap condition. The Sequencer continues to
"  be clocked during a trap.

SEQ_EN =
      !haltmode    &  !(clk_cntl == ss_halt)  &  !TRAP   #
       haltmode    &   (clk_cntl == ss      ) &  !TRAP   #
       haltmode    &   (clk_cntl == ss_reset) #
       haltmode    &   (clk_cntl == ss_cntl ) ;

"  The pipeline register clock is similar to the Sequencer clock but it
"  may be forced active by one additional condition when in the halt
"  mode. The condition is a load pipeline command on the CLK_CONTROL
"  lines.

   PIPE_EN =

      !haltmode & !(clk_cntl == ss_halt)  #
       haltmode &  (clk_cntl == ss      ) #
       haltmode &  (clk_cntl == ss_reset) #
       haltmode &  (clk_cntl == ss_cntl ) #
       haltmode &  (clk_cntl == ld_pipe ) ;

"  The macro status register clock is similar to the control section
"  clock but it is further qualified by the pipeline enable to load the
"  macro status register. The register will be loaded in any event if
"  there is a load macro opcode command on the CLK_CONTROL lines.

   STAT_EN =
      !haltmode  &  !(clk_cntl == ss_halt)  &  !TRAP & P_LD_MAC_STAT  #
       haltmode  &   (clk_cntl == ss      ) &  !TRAP & P_LD_MAC_STAT  #
       haltmode  &   (clk_cntl == ss_reset) &  P_LD_MAC_STAT  #
       haltmode  &   (clk_cntl == ss_cntl ) &  P_LD_MAC_STAT  #
                     (clk_cntl == ld_mop  )                   ;
```

```
"  The macro opcode register clock is very similar to the macro status
"  register clock but it is qualified by the pipeline load command for
"  the macro opcode.

"MOP_EN =
      !haltmode  &  !(clk_cntl == ss_halt)  &  !TRAP & P_LD_MAC_OP     #
      haltmode   &   (clk_cntl == ss       ) &  !TRAP & P_LD_MAC_OP     #
      haltmode   &   (clk_cntl == ss_reset) &  P_LD_MAC_OP     #
      haltmode   &   (clk_cntl == ss_cntl ) &  P_LD_MAC_OP     #
                     (clk_cntl == ld_mop  )                    ;

"  The interrupt base address register clock is similar to the control
"  section clock but it is further qualified by the pipeline enable to
"  load the interrupt base register. This signal is qualified by the
"  pipeline Force Continue being active. The register will be loaded in
"  any event if there is a load macro opcode command on the CLK_CONTROL
"  lines.

   INTB_EN =
      !haltmode  &  !(clk_cntl == ss_halt)  &
         !P_FC_  &  !TRAP & P_LD_INT_BASE #
      haltmode   &   (clk_cntl == ss       ) &
         !P_FC_  &  !TRAP & P_LD_INT_BASE #
      haltmode   &   (clk_cntl == ss_reset) &
         !P_FC_  &  P_LD_INT_BASE #
      haltmode   &   (clk_cntl == ss_cntl ) &
         !P_FC_  &  P_LD_INT_BASE #
      (clk_cntl == ld_mop  )                ;

"  The reset for the Am29300 is made active by a reset command from the
"  host interface controller.

   !RESET_300_ =

      (clk_cntl == ss_reset);

"  The interrupt to the Am29300 CPU is made active by an interrupt
"  command from the host interface controller.

   !INT_CPU_ =

      (clk_cntl ==int_cpu);

"  Test_Vectors need to be defined.

End;
```

APPENDIX L
Clock Generator PAL Definition

```
" Advanced Micro Devices Application Note:
"
" Am29300 Demonstration System
"
" By Mark Mc Clain, Field Applications Engineer, San Diego, CA.
" (619)560-7030, Date = 1/87

Module
   Clock_Generator;

Title
'Clock Generator PAL for an Am29300 Demonstration System.';

ckgen device 'P16R6';

"declarations

   "Signal names that end in an underline indicate an active low signal.

   CLOCK_MODULE, P_CLK_LEN_1, P_CLK_LEN_0 Pin
        1,            2,             3;

   CLK_FREE_RUN_, D_1_, D_2_, D_3_, D_4_ Pin
         18,        17,   16,   15,   14;

   CLK_FREE_RUN_, D_1_, D_2_, D_3_, D_4_

      Istype 'neg, reg';

" declare some sets

   cycle = [P_CLK_LEN_1, P_CLK_LEN_0];

EQUATIONS

   !CLK_FREE_RUN_ :=
   CLK_FREE_RUN_        #
   D_1_                 #
   D_2_ & (cycle == 1)  #
   D_2_ & (cycle == 2)  #
   D_3_ & (cycle == 2)  #
   D_2_ & (cycle == 3)  #
   D_3_ & (cycle == 3)  #
   D_4_ & (cycle == 3)  ;

!D_1_ :=

   !CLK_FREE_RUN_;
```

```
!D_2_ :=

   !CLK_FREE_RUN_ & !D_1_;

!D_3_ :=

   !CLK_FREE_RUN_ & !D_2_;

!D_4_ :=

   !CLK_FREE_RUN_ & !D_3_;

TEST_VECTORS

       ([CLOCK_MODULE, P_CLK_LEN_1, P_CLK_LEN_0]  ->
        [CLK_FREE_RUN_, D_1_, D_2_, D_3_, D_4_])

   [.C.,0,0] -> [0,1,1,1,1];
   [.C.,0,0] -> [0,0,1,1,1];
   [.C.,0,0] -> [1,0,0,1,1];
   [.C.,0,0] -> [0,1,1,1,1];
   [.C.,0,1] -> [0,0,1,1,1];
   [.C.,0,1] -> [0,0,0,1,1];
   [.C.,0,1] -> [1,0,0,0,1];
   [.C.,0,0] -> [0,1,1,1,1];
   [.C.,1,0] -> [0,0,1,1,1];
   [.C.,1,0] -> [0,0,0,1,1];
   [.C.,1,0] -> [0,0,0,0,1];
   [.C.,1,0] -> [1,0,0,0,0];
   [.C.,1,0] -> [0,1,1,1,1];
   [.C.,1,1] -> [0,0,1,1,1];
   [.C.,1,1] -> [0,0,0,1,1];
   [.C.,1,1] -> [0,0,0,0,1];
   [.C.,1,1] -> [0,0,0,0,0];
   [.C.,1,1] -> [1,0,0,0,0];
   [.C.,1,1] -> [0,1,1,1,1];
   [.C.,0,0] -> [0,0,1,1,1];
   [.C.,0,0] -> [1,0,0,1,1];

End;
```

APPENDIX M
Control Decode PALs Definition

```
" Advanced Micro Devices Application Note:
"
" Am29300 Demonstration System
"
" By Mark Mc Clain, Field Applications Engineer, San Diego, CA.
" (619)560-7030, Date = 1/87

Module
   Control_Decode_Data_Path;

Title
'Control Decode PAL for the data path,
for an Am29300 Demonstration System.';

condedp device 'P18P8';

"declarations

"Signal names that end in an underline indicate an active low signal.

P_DPS_1, P_DPS_0, P_OEA_, P_SEED_OE, P_FTP, P_FP_FT_1 Pin
   1,       2,       3,        4,        5,       6;

P_FP_FT_0 Pin
  7;

ALU_OE_, ALU_HOLD, PM_OE_, SEED_OE_, FTP, FP_FT_1, FP_FT_0 Pin
   19,      18,       17,        16,     15,     14,      13;

D_OER_ Pin
  12;

ALU_HOLD, FTP, FP_FT_1, FP_FT_0
   Istype 'pos, com';

ALU_OE_, PM_OE_, SEED_OE_, D_OER_
   Istype 'neg, com';

" declare some sets

   dps = [P_DPS_1, P_DPS_0];

EQUATIONS

   !ALU_OE_ =  (dps == 0) # (dps == 3);

   ALU_HOLD =  !(dps == 0);

   !PM_OE_  =  (dps == 1);

   !SEED_OE_ = ((dps == 2) # (dps == 3)) & P_SEED_OE;

   FTP = (dps == 1) & P_FTP;
```

```
   FP_FT_1 =  ((dps == 2) # (dps == 3)) & P_FP_FT_1;

   FP_FT_0 =  ((dps == 2) # (dps == 3)) & P_FP_FT_0;

   !D_OER_ = !(((dps == 2) # (dps == 3)) & P_SEED_OE) & P_OEA_;

" Test_Vectors need to be defined.

End;

Module
   Control_Decode_Memory;

Title
'Control Decode PAL for memory enables
for an Am29300 Demonstration System.';

condemem device 'P18P8';

"declarations

"Signal names that end in an underline indicate an active low signal.

P_MEM_3, P_MEM_2, P_MEM_1, P_MEM_0, P_FC_, P_INIT, WCS_INIT_ Pin
    1,       2,       3,       4,      5,      6,        7;

AD_MD_OE_, INIT_MC_ Pin
   19,        18;

AD_MD_OE_, INIT_MC_
   Istype 'neg, com';

" declare some sets

   select = [P_MEM_1, P_MEM_0];

EQUATIONS

   !AD_MD_OE_ = (select == 1);

   !INIT_MC_ = P_FC_ & P_INIT # WCS_INIT_;

" Test_Vectors need to be defined.

End;

Module
   Control_Decode_D_BUS;

Flag '-r3'

Title
'Control Decode for the D_BUS driver enables
for an Am29300 Demonstration System.';

condbus device 'P18P8';
```

```
"declarations

   "Signal names that end in an underline indicate an active low signal.

   P_BRANCH_EN_, P_FC_, P_INT_INST_3, P_INT_INST_2, P_INT_INST_1 Pin
        1,          2,           3,            4,            5;

   P_INT_INST_0, P_SEQ_INST_5, P_SEQ_INST_4, P_SEQ_INST_3 Pin
        6,              7,             8,             9;

   P_SEQ_INST_2, P_SEQ_INST_1, P_SEQ_INST_0 Pin
        11,             12,            13;

   D_OET_, SEQ_OED, IEN_, INT_CS_, D_SIGN_EX Pin
     19,      18,     17,    16,        15;

   D_OET_, INT_CS_,

   SEQ_OED,  D_SIGN_EX

      Istype 'pos, com';

   IEN_

      Istype 'neg, com';

" declare some sets

   seq_sp = [P_SEQ_INST_5, P_SEQ_INST_4];

   seq_inst = [P_SEQ_INST_5, P_SEQ_INST_4, P_SEQ_INST_3, P_SEQ_INST_2,
               P_SEQ_INST_1, P_SEQ_INST_0 ];

   int_inst = [P_INT_INST_3, P_INT_INST_2, P_INT_INST_1, P_INT_INST_0];

   pop_d     = ^h34;
   continue  = ^h30;
   rdmk      = ^h7;
   rdsr      = ^hB;
   rdir      = ^hF;

EQUATIONS

"  Enable A_BUS to D_BUS path if no one else will drive the D_BUS.

   D_OET_ = !P_BRANCH_EN_ #
             (P_FC_ &
             ((seq_inst == continue) # (seq_inst == pop_d) ) #
             ((((seq_sp == ^b11) & P_FC_) # !P_FC_) & ((int_inst == rdmk) #
             (int_inst == rdsr) # (int_inst == rdir)
             )
             )
            );
```

```
"   Enable sequencer output on continue to be able to read the stack
"   pointer and on pop_d to read the top of stack.

    SEQ_OED = P_BRANCH_EN_ & P_FC_ &
              ((seq_inst == continue) # (seq_inst == pop_d));

"   Unconditional instruction or when Force Continue.

    !IEN_ = (seq_sp == ^b11) & P_FC_ # !P_FC_;

" Disable if branch field active or if sequencer active and a interrupt
" controller read instruction is valid.

    INT_CS_ = !P_BRANCH_EN_ #
              (P_FC_ &
               ((seq_inst == continue) # (seq_inst == pop_d)) & ((int_inst == rdmk) #
               (int_inst == rdsr) # (int_inst == rdir))
               );

    D_SIGN_EX = !P_FC_ & P_SEQ_INST_4;

" Test_Vectors need to be defined.

End;
```

APPENDIX N
Components List

PART ID	PART NUMBER	# OF PINS	POWER SUPPLY REQUIREMENTS				FIGURE #
			STANDARD BIPOLAR		CMOS OR LOW POWER VERSION		
			Amps	Watts	Amps	Watts	
U 7	Am27S25	24	0.185	0.925	0.185	0.925	3-5
U 8	Am27S43	24	0.185	0.925	0.185	0.925	3-5
U 9	Am27S43	24	0.185	0.925	0.185	0.925	3-5
U 73	Am29114	40	0.395	1.975	0.395	1.975	5-11
U 10	Am2920	24	0.037	0.185	0.037	0.185	3-5
U 11	Am2920	24	0.037	0.185	0.037	0.185	3-5
U 12	Am2920	24	0.037	0.185	0.037	0.185	3-5
U 13	Am29C323	169	0.3	1.5	0.3	1.5	3-6
U 5	Am29325	145	1.743	8.715	0.3	1.5	3-3
U 67	Am29331	120	1	5	0.3	1.5	5-9
U 3	Am29332	169	1.36	6.8	0.3	1.5	3-2
U 1	Am29334	120	0.85	4.25	0.2	1	3-1
U 2	Am29334	120	0.85	4.25	0.2	1	3-1
U 14	Am29806	24	0.035	0.175	0.035	0.175	4-3
U 4	Am29818-1	24	0.155	0.775	0.155	0.775	3-2
U 18	Am29818-1	24	0.155	0.775	0.155	0.775	4-5
U 19	Am29818-1	24	0.155	0.775	0.155	0.775	4-5
U 20	Am29818-1	24	0.155	0.775	0.155	0.775	4-5
U 21	Am29818-1	24	0.155	0.775	0.155	0.775	4-5
U 48	Am29818-1	24	0.155	0.775	0.155	0.775	5-1
U 49	Am29818-1	24	0.155	0.775	0.155	0.775	5-1
U 50	Am29818-1	24	0.155	0.775	0.155	0.775	5-1
U 51	Am29818-1	24	0.155	0.775	0.155	0.775	5-1
U 55	Am29818-1	24	0.155	0.775	0.155	0.775	5-3
U 56	Am29818-1	24	0.155	0.775	0.155	0.775	5-3
U 74	Am29818-1	24	0.155	0.775	0.155	0.775	5-11
U 15	Am29825	24	0.13	0.65	0.0001	0.0005	4-3
U 36	Am29827	24	0.075	0.375	0.0001	0.0005	4-8
U 37	Am29827	24	0.075	0.375	0.0001	0.0005	4-8
U 38	Am29827	24	0.075	0.375	0.0001	0.0005	4-8
U 39	Am29827	24	0.075	0.375	0.0001	0.0005	4-8
U 40	Am29827	24	0.075	0.375	0.0001	0.0005	4-8
U 41	Am29827	24	0.075	0.375	0.0001	0.0005	4-8
U 42	Am29827	24	0.075	0.375	0.0001	0.0005	4-9
U 43	Am29827	24	0.075	0.375	0.0001	0.0005	4-9
U 60	Am29827	24	0.075	0.375	0.0001	0.0005	5-5
U 61	Am29827	24	0.075	0.375	0.0001	0.0005	5-5
U106	Am29827	24	0.075	0.375	0.0001	0.0005	4-3
U104	Am29828	24	0.075	0.375	0.0001	0.0005	5-11
U 68	Am29853	24	0.18	0.9	0.0001	0.0005	5-10
U 69	Am29853	24	0.18	0.9	0.0001	0.0005	5-10
U 70	Am29853	24	0.18	0.9	0.0001	0.0005	5-10
U 71	Am29853	24	0.18	0.9	0.0001	0.0005	5-10
U 72	Am29862	24	0.15	0.75	0.0001	0.0005	5-10

PART ID	PART NUMBER	# OF PINS	POWER SUPPLY REQUIREMENTS				FIGURE #
			STANDARD BIPOLAR		CMOS OR LOW POWER VERSION		
			Amps	Watts	Amps	Watts	
U 44	Am29863	24	0.15	0.75	0.0001	0.0005	4-9
U 45	Am29863	24	0.15	0.75	0.0001	0.0005	4-9
U 46	Am29863	24	0.15	0.75	0.0001	0.0005	4-9
U 47	Am29863	24	0.15	0.75	0.0001	0.0005	4-9
U 16	Am29PL141	28	0.4	2	0.4	2	4-3
U 52	Am9150-25	24	0.18	0.9	0.13	0.65	5-3
U 53	Am9150-25	24	0.18	0.9	0.13	0.65	5-3
U 54	Am9150-25	24	0.18	0.9	0.13	0.65	5-3
U 76	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 77	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 78	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 79	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 80	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 81	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 82	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 83	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 84	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 85	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 86	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 87	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 88	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 89	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 90	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 91	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 92	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 93	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 94	Am9151-50	24	0.18	0.9	0.18	0.9	5-13
U 95	Am9151-50	24	0.18	0.9	0.18	0.9	5-14
U 96	Am9151-50	24	0.18	0.9	0.18	0.9	5-14
U 97	Am9151-50	24	0.18	0.9	0.18	0.9	5-14
U 98	Am9151-50	24	0.18	0.9	0.18	0.9	5-14
U 22	Am99C165-35	24	0.11	0.55	0.11	0.55	4-6
U 23	Am99C165-35	24	0.11	0.55	0.11	0.55	4-6
U 24	Am99C165-35	24	0.11	0.55	0.11	0.55	4-6
U 25	Am99C165-35	24	0.11	0.55	0.11	0.55	4-6
U 26	Am99C165-35	24	0.11	0.55	0.11	0.55	4-6
U 27	Am99C165-35	24	0.11	0.55	0.11	0.55	4-6
U 28	Am99C165-35	24	0.11	0.55	0.11	0.55	4-6
U 29	Am99C165-35	24	0.11	0.55	0.11	0.55	4-6
U 30	Am99C165-35	24	0.11	0.55	0.11	0.55	4-6
U100	AmPAL16R6B	20	0.18	0.9	0.18	0.9	5-16
U 62	AmPAL18P8	20	0.18	0.9	0.055	0.275	5-6
U 63	AmPAL18P8	20	0.18	0.9	0.055	0.275	5-6
U101	AmPAL18P8B	20	0.18	0.9	0.18	0.9	5-21
U102	AmPAL18P8B	20	0.18	0.9	0.18	0.9	5-21
U103	AmPAL18P8B	20	0.18	0.9	0.18	0.9	5-22

PART ID	PART NUMBER	# OF PINS	POWER SUPPLY REQUIREMENTS				FIGURE #
			STANDARD BIPOLAR		CMOS OR LOW POWER VERSION		
			Amps	Watts	Amps	Watts	
U 32	AmPAL22V10	24	0.15	0.75	0.15	0.75	4-7
U 33	AmPAL22V10	24	0.15	0.75	0.15	0.75	4-7
U 34	AmPAL22V10	24	0.15	0.75	0.15	0.75	4-7
U 35	AmPAL22V10	24	0.15	0.75	0.15	0.75	4-7
U 64	AmPAL22V10	24	0.15	0.75	0.15	0.75	5-7
U 65	AmPAL22V10	24	0.15	0.75	0.15	0.75	5-8
U 66	AmPAL22V10	24	0.15	0.75	0.15	0.75	5-8
U 6	AmPAL22V10A	24	0.15	0.75	0.15	0.75	3-3
U 17	AmPAL22V10A	24	0.15	0.75	0.15	0.75	4-4
U 57	AmPAL22V10A	24	0.15	0.75	0.15	0.75	5-4
U 58	AmPAL22V10A	24	0.15	0.75	0.15	0.75	5-4
U 59	AmPAL22V10A	24	0.15	0.75	0.15	0.75	5-4
U 75	AmPAL22V10A	24	0.15	0.75	0.15	0.75	5-11
U 99	AmPAL22V10A	24	0.15	0.75	0.15	0.75	5-15
U 31	74AS32	14	0.0165	0.0825	0.0165	0.0825	4-6
U107	74AS804A	24	0.016	0.08	0.016	0.08	5-15
U108	74AS804A	24	0.016	0.08	0.016	0.08	5-15
U105	74ALS08	14	0.0022	0.011	0.0022	0.011	5-10
Total		3267	20.859	104.29	13.458	67.294	

APPENDIX O

Goals

The primary guidelines behind the design choices made in this application note are outlined below. Listing them here will help in understanding the design alternatives selected.

1. Illustrate the use of several of the 29300 family components in a typical system arrangement.
2. Show both macroprogram and microprogram approaches to processor design.
3. Make the design general purpose so that it may be copied in whole or in part by other engineers and used in a wide range of applications.
4. Provide hardware aids for Digital Signal Processing algorithms.
5. Illustrate the use of Serial Shadow Register (SSR) diagnostics.
6. Show the use of dual write data ports on the 29334 register file.
7. Work through the details of support logic design and system timing.

Disclaimer

Warning: This is a paper design. It has not been implemented in hardware. The design is therefore subject to the usual number of oversights, mistakes, and outright blunders that lie hidden in the depths of any complex and untried plan.

All AC & DC parameters quoted in this document were based on information available at the time of writing this document. Some of the parameters represent PRELIMINARY information, which is subject to change.