hyper**stone**

32-Bit-Microprocessor User's Manual

hyper**stone** **electronics** GmbH

This document contains information on a new product. Specifications and information herein are subject to change without notice. hyperstone electronics GmbH reserves the right to make changes to improve functioning. Although the information in this document has been carefully reviewed, hyperstone electronics GmbH does not assume any liability arising out of the use of the product or circuit described herein.
hyperstone electronics GmbH does not recommend the use of the Hyperstone microprocessor in life support applications wherein a failure or malfunction of the microprocessor may directly threaten life or cause injury. The user of the Hyperstone microprocessor in life support applications assumes all risks of such use and indemnifies hyperstone electronics GmbH against all damages.

For further information, please contact:

hyperstone electronics GmbH
Robert-Bosch-Str. 11
D-7750 Konstanz
West-Germany

Tel. 0 75 31 - 6 77 89
Fax. 0 75 31 - 5 17 25

©1987 Ingenieurbuero Otto Mueller
Issue April 1990

Table of Contents

1 Architecture

1.1 Introduction 1 - 1
1.2 Block Diagram 1 - 6
1.3 Register Model 1 - 7
1.4 Local Register Set 1 - 9
1.5 Global Register Set 1 - 9
1.5.1 Program Counter PC 1 - 9
1.5.2 Stack Pointer SP 1 - 10
1.5.3 Upper Stack Bound UB 1 - 10
1.5.4 Status Register SR 1 - 10
1.5.5 Bus Control Register BCR (see 5.2) 1 - 10
1.5.6 Status Information 1 - 11
1.5.7 Privilege States 1 - 14
1.6 Register Data Types 1 - 15
1.7 Memory Organization 1 - 16
1.8 Stack 1 - 18
1.9 Instruction Cache 1 - 21

2 Instructions General

2.1 Instruction Notation 2 - 1
2.2 Instruction Execution 2 - 2
2.3 Instruction Formats 2 - 3
2.3.1 Table of Immediate Values 2 - 5
2.3.2 Table of Instruction Codes 2 - 6
2.4 Entry Table 2 - 7
2.5 Instruction Timing 2 - 8

3 Instruction Set

3.1 Memory Instructions 3 - 1

3.1.1 Address Modes 3 - 2

3.1.2 Load Instructions 3 - 6

3.1.3 Store Instructions 3 - 8

3.1.4 Exchange Instruction 3 - 10

3.2 Move Word Instructions 3 - 11

3.3 Move Double-Word Instruction 3 - 11

3.4 Logical Instructions 3 - 12

3.5 Invert Instruction 3 - 13

3.6 Mask Instruction 3 - 13

3.7 Add Instructions 3 - 14

3.8 Sum Instructions 3 - 16

3.9 Subtract Instructions 3 - 17

3.10 Negate Instructions 3 - 18

3.11 Multiply Word Instruction 3 - 19

3.12 Multiply Double-Word Instructions 3 - 20

3.13 Divide Instructions 3 - 21

3.14 Shift Left Instructions 3 - 22

3.15 Shift Right Instructions 3 - 23

3.16 Rotate Left Instruction 3 - 24

3.17 Index Move Instructions 3 - 25

3.18 Check Instructions 3 - 26

3.19 No Operation Instruction 3 - 26

3.20 Compare Instructions 3 - 27

3.21 Compare Bit Instructions 3 - 28

3.22 Test Leading Zero's Instruction 3 - 28

3.23 Set Stack Address Instruction 3 - 29

3.24 Set Conditional Instructions 3 - 29

3.25 Branch Instructions 3 - 31

3.26 Delayed Branch Instructions 3 - 32

3.27 Call Instructions 3 - 34

3.28 Trap Instructions 3 - 35

3.29 Frame Instruction 3 - 37

3.30 Return Instruction 3 - 39

3 Instruction Set (continued)

3.31 Fetch Instruction 3 - 41
3.32 Software Instructions 3 - 42
3.32.1 Do Instruction 3 - 43
3.32.2 Extend Instruction 3 - 44
3.32.3 Floating-Point Instructions 3 - 45

4 Exceptions

4.1 Exception Processing 4 - 1
4.2 Exception Types 4 - 2
4.2.1 Reset 4 - 2
4.2.2 Pointer, Frame and Privilege Error 4 - 2
4.2.3 Data Page Fault 4 - 3
4.2.4 Range Error 4 - 3
4.2.5 Interrupt Exception 4 - 3
4.2.6 Trace Exception 4 - 4
4.2.7 Instruction Page Fault 4 - 4
4.3 Exception Backtracking 4 - 4

5 Bus Interface

5.1 Bus Control General 5 - 1
5.2 Bus Control Register BCR 5 - 3
5.3 I/O Bus Control 5 - 4
5.4 Bus Signals 5 - 5
5.5 Bus Cycles 5 - 11
5.5.1 Read Access 5 - 11
5.5.2 Write Access 5 - 12
5.5.3 Exchange Access 5 - 13
5.5.4 Read/Write Access 5 - 14
5.5.5 Write/Read Access 5 - 15
5.5.6 DRAM Access 5 - 16
5.5.7 DRAM Refresh 5 - 17
5.6 Bus Arbitration 5 - 18
5.7 D.C. Characteristics 5 - 19
5.8 A.C. Characteristics 5 - 20

6 Mechanical Data

6.1	Pin Configuration - View from Top Side	6 - 1
6.2	Pin Configuration - View from Pin Side	6 - 2
6.3	Pin Cross Reference by Pin Name	6 - 3
6.4	Pin Cross Reference by Location	6 - 4
6.5	Package Dimensions	6 - 5

1 Architecture

1.1 Introduction

The microprocessor presented here cannot be classed with the conventional RISC or CISC architectures. It constitutes a class of its own: The new class of RISC-like computers executing a large set of powerful and yet concise instructions in one cycle. The burst-rate speed of one cycle per instruction is almost sustained in many programs without an external cache memory. This high throughput is not achieved by raw clock speed, it is due to a sophisticated new architecture.
The speed is obtained by an optimized combination of the following features:

- The most recent stack frames are kept in a register set, thereby reducing data memory accesses to a minimum by keeping almost all local data in registers.
- Pipelined memory access allows overlapping of memory accesses with execution.
- On-chip instruction cache omits instruction fetch in inner loops and provides strategic prefetch.
- Variable-length instructions of 16, 32 or 48 bits provide a large, powerful instruction set, thereby reducing the number of instructions to be executed.
- Primarily used 16-bit instructions halve the memory accesses required for instruction fetch in comparison to conventional RISC architectures with fixed-length 32-bit instructions, yielding also even better code economy than conventional CISC architectures.
- Regular instruction set allows hardwiring of control logic at low component count.
- Most instructions execute in one cycle.
- Fast Call and Return by parameter passing via registers.
- An instruction pipeline depth of only two stages – decode/execute – provides branching without insertion of wait cycles in combination with Delayed Branch instructions.
- Range and pointer checks required by Pascal, Modula-2 and Ada are performed without speed penalty, thus, these checks need no longer be turned off, thereby providing higher runtime reliability.
- Separate address and data buses provide a throughput of one 32-bit word each cycle.

1.1 Introduction (continued)

The features noted above contribute to reduce the number of idle wait cycles to a bare minimum. Thus, the target of executing exactly one instruction in each cycle is almost met.
The processor is designed to sustain its execution rate without an external cache memory, only a standard DRAM memory with fast page mode reading or writing one word in each cycle is required.
The processor is implemented in 1.2μ CMOS on a chip with a core of 42 mm^2, the transistor count is ca. 85 000. Most of the transistors are used for the register stack and the on-chip instruction cache; only an insignificant number are required for the control logic.
Due to its low system cost – no external cache, small chip area – the processor can be used in high-speed yet cost- and speed-efficient embedded systems.

The following description gives a brief architectural overview:

Registers:
- 19 global and 64 local registers of 32 bits each
- Directly addressable are 16 global and up to 16 local registers

Flags:
- Zero(Z), negative(N), carry(C) and overflow(V) flag
- Interrupt-lock, trace-mode, trace-pending, supervisor state, cache-mode and high global flag

Register Data Types:
- Unsigned integer, signed integer, bitstring, IEEE-754 floating-point, each either 32 or 64 bits

Memory:
- Address space of 4 Gbytes
- Separate I/O address space
- Load/Store architecture
- Pipelined memory and I/O accesses
- High-order data consistently located and addressed at lower address
- Virtual memory by demand paging, assisted by page fault signals from external MMU
- Fault-causing memory instructions can easily be identified and repeated
- Instructions and double-word data may cross page boundaries

1.1 Introduction (continued)

Memory Data Types:

- Unsigned and signed byte (8 bit)
- Unsigned and signed halfword (16 bit), located on halfword boundary
- Undedicated word (32 bit), located on word boundary
- Undedicated double-word (64 bit), located on word boundary

Runtime Stack:

- Runtime stack is divided into memory part and register part
- Register part is implemented by the 64 local registers holding the most recent stack frame(s)
- Current stack frame (maximum 16 registers) is always kept in register part of the stack
- Data transfer between memory and register part of the stack is automatic
- Upper stack bound is guarded

Instruction Cache:

- An instruction cache of 128 bytes reduces instruction memory accesses substantially

Instructions General:

- Variable-length instructions of one, two or three halfwords halve required memory bandwidth
- Pipeline depth of only two stages, assures immediate refill after branches
- Register instructions of type "source operator destination -> destination" or "source operator immediate -> destination"
- All 32 or 64 bits participate in an operation
- Immediate operands of 5, 16 and 32 bits, zero- or sign-expanded
- Two sets of signed arithmetical instructions: instructions set or clear either only the overflow flag or trap additionally to a Range Error routine on overflow

1.1 Introduction (continued)

Instruction Summary:

- Memory instructions pipelined to a depth of two stages, trap on address register equal to zero (check for invalid pointers)
- Memory address modes: register address, register postincrement, register + displacement (including PC relative), register postincrement by displacement (next address), absolute, stack address, I/O absolute and I/O displacement
- Load, all data types, bytes and halfwords right adjusted and zero- or sign-expanded, execution proceeds after Load until data is needed
- Store, all data types, trap when unsigned or signed range of byte or halfword is exceeded
- Exchange word memory <-> register (for semaphores)
- Move, Move immediate, Move double-word
- Logical instructions AND, AND not, OR, XOR, NOT, AND not immediate, OR immediate, XOR immediate
- Mask source and immediate -> destination
- Add unsigned/signed, Add signed with trap on overflow, Add with carry
- Add unsigned/signed immediate, Add signed immediate with trap on overflow
- Sum source + immediate -> destination, unsigned/signed and signed with trap on overflow
- Subtract unsigned/signed, Subtract signed with trap on overflow, Subtract with carry
- Negate unsigned/signed, Negate signed with trap on overflow
- Multiply word * word -> low-order word signed with trap on low-order word overflow, Multiply word * word -> double-word unsigned and signed
- Divide double-word by word -> quotient and remainder, unsigned and signed
- Shift left unsigned/signed, single and double-word, by constant and by content of register, Shift left signed by constant with trap on loss of high-order bits
- Shift right unsigned and signed, single and double-word, by constant and by content of register
- Rotate left single word by content of register
- Index Move, check an index value for bounds and move it scaled by 1, 2, 4 or 8
- Check a value for an upper bound specified in a register or check for zero
- Compare unsigned/signed, Compare unsigned/signed immediate
- Compare bits, Compare bits immediate, Compare any byte zero
- Test number of leading zeros

1.1 **Introduction** (continued)

- Set Conditional, save conditions in a register
- Branch unconditional and conditional (12 conditions)
- Delayed Branch unconditional and conditional (12 conditions)
- Call subprogram, unconditional and on overflow
- Trap to supervisor subprogram, unconditional and conditional (11 conditions)
- Frame, structure a new stack frame, include parameters in frame addressing, set frame length, restore reserve length and check for upper stack bound
- Return from subprogram, restore program counter, status register and return-frame
- Software instructions, call an associated subprogram and pass a source operand and the address of a destination operand to it
- Floating-point instructions are architecturally fully integrated, they are executed as Software instructions by the present version. Floating-point Add, Subtract, Multiply, Divide, Compare and Compare unordered for single and double-precision, and Convert single <-> double are provided.

Note: The omission of packed BCD arithmetic is not an oversight. It is recommended that decimal numbers are represented as scaled two's complement integers. A decimal number is scaled by a factor of 10 for each digit right to the decimal point.

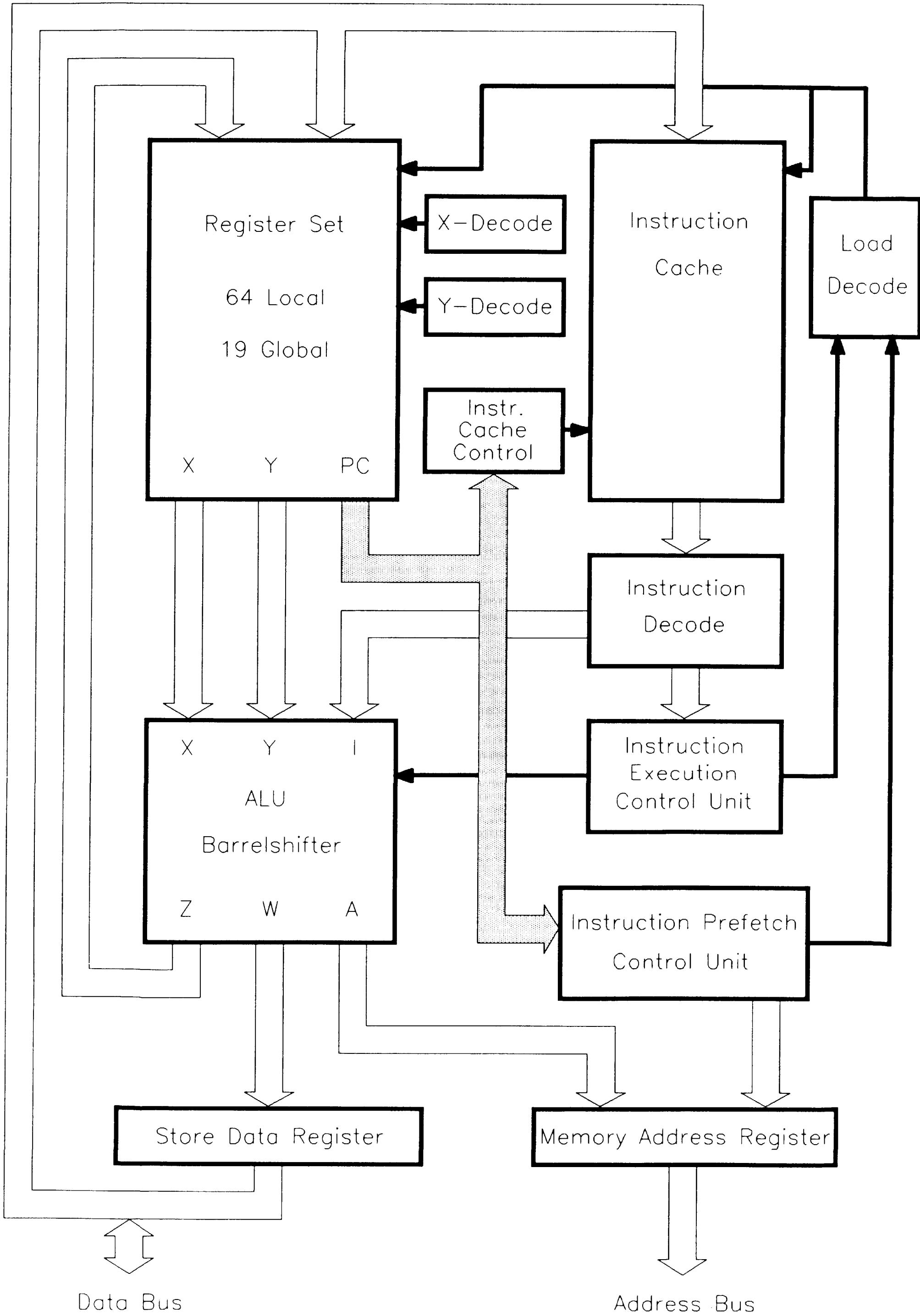
Exceptions:

- Pointer, Privilege, Frame and Range Error, Data and Instruction Page Fault, Interrupt and Trace mode exception
- Error- and fault-causing instructions can be identified by backtracking, allowing a very detailed error analysis

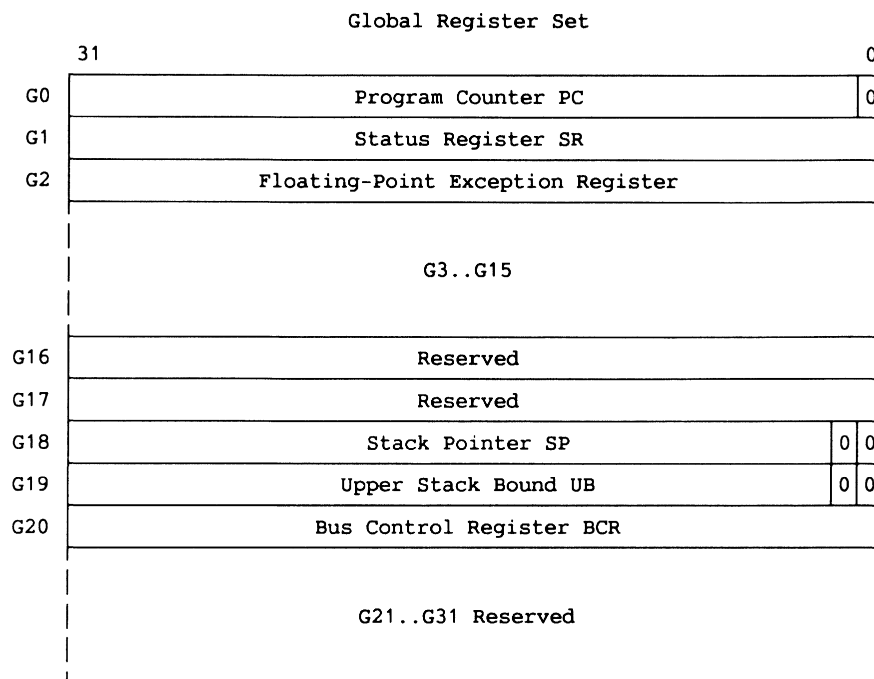
Bus Interface:

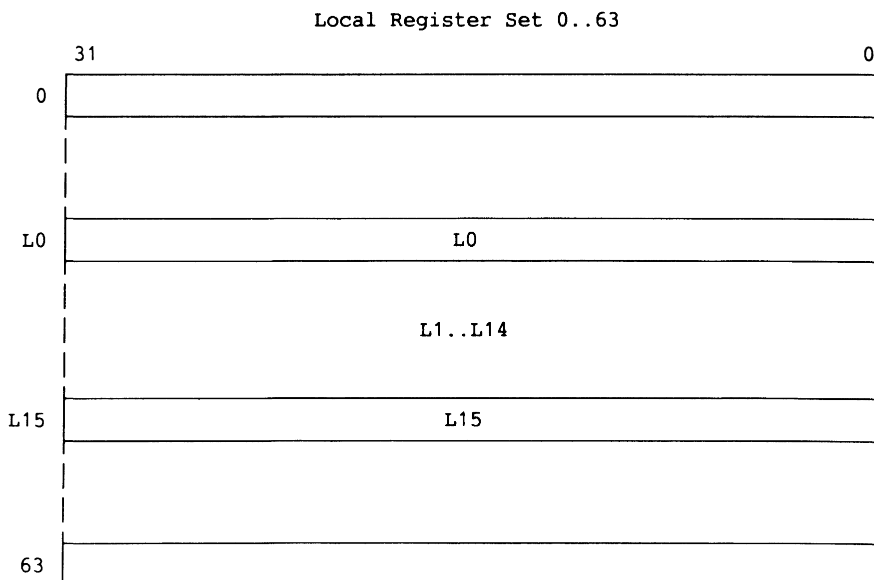
- Separate address and data buses of 30 and 32 bits respectively provide a throughput of four bytes at each clock cycle
- Alternating between different bus masters from cycle to cycle, with a loss of only one bus cycle

1.2 Block Diagram



1.3 Register Model

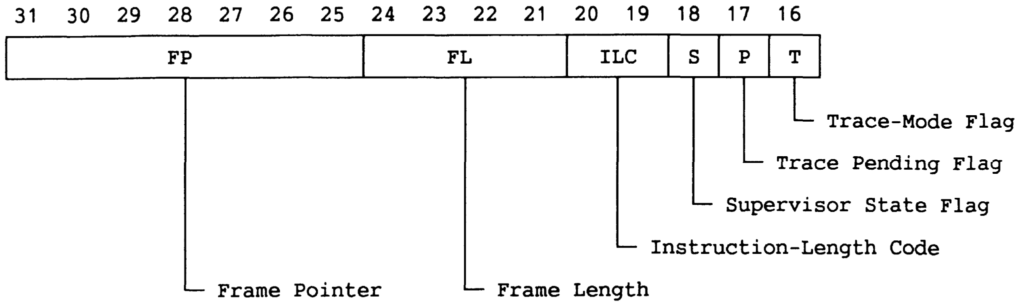




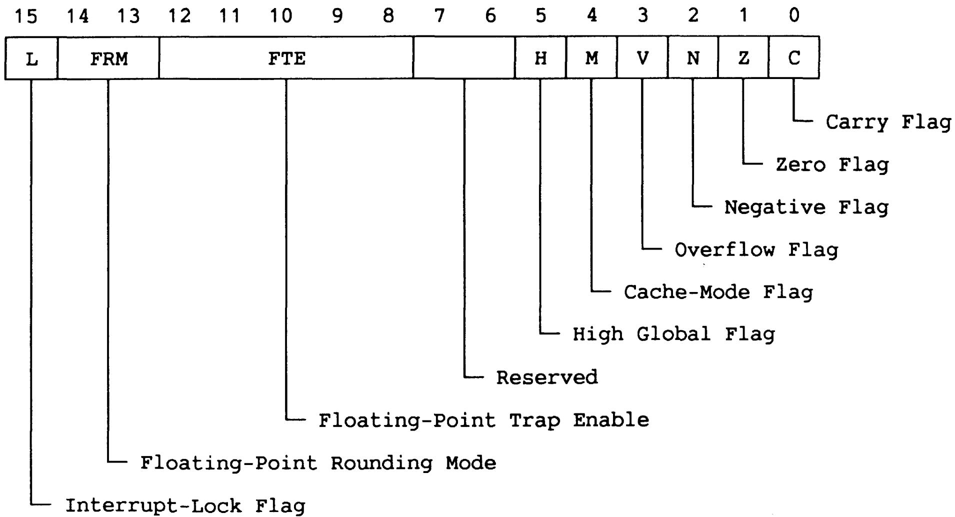
Bit 31 = most significant bit, bit 0 = least significant bit

1.3 Register Model (continued)

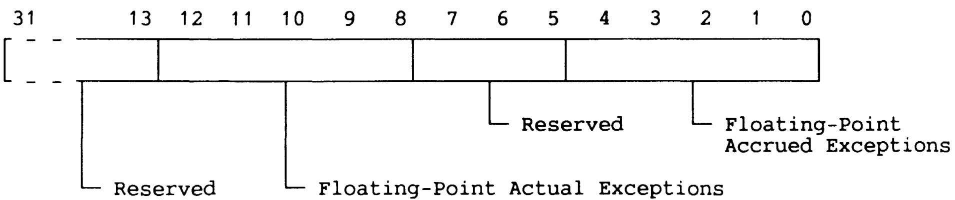
Status Register G1 (bits 31..16):



Status register G1 (bits 15..0):



Floating-Point Exception Register G2 (bits 31..0):



1.4 Local Register Set

The architecture provides a set of 64 local registers of 32 bits each. The local registers 0..63 represent the register part of the stack, containing the most recent stack frame(s).
The local registers can be addressed by the register code (0..15) of an instruction as L0..L15 only relative to the frame pointer FP; they can also be addressed absolutely as part of the stack in the stack address mode (see address modes).
The absolute local register address is calculated from the register code as:

absolute local register address := (FP + register code) modulo 64.

That is, only the least significant six bits of the sum FP + register code are used and thus, the absolute local register addresses for L0..L15 wrap around modulo 64.
The absolute local register addresses for FP + register code + 1 or FP + FL + offset are calculated accordingly.

1.5 Global Register Set

The architecture provides 19 global registers of 32 bits each. These are:

G0	Program counter PC
G1	Status register SR
G2	Floating-point exception register
G3..G15	General purpose registers
G16..G17	Reserved
G18	Stack Pointer SP
G19	Upper stack Bound UB
G20	Bus control register BCR (see bus interface)
G21..G31	Reserved

G0..G15 can be addressed directly by the register code (0..15) of an instruction. G18..G20 can be addressed only via the high global flag H by a MOV instruction.

1.5.1 Program Counter PC

G0 is the program counter PC. It is updated to the address of the next instruction through instruction execution. Besides this implicit updating, the PC can also be addressed like a regular source or destination register. When the PC is referenced as an operand, the value supplied is the address of the first byte after the instruction which references it, except when referenced by a delay instruction with a preceding delayed branch taken (see Delayed Branch instructions).

1.5.1 Program Counter PC (continued)

Placing a result in the PC has the effect of a branch taken. Bit zero of the PC is always zero, regardless of any value placed in the PC.

1.5.2 Stack Pointer SP

G18 is the stack pointer SP. The SP contains the top address + 4 of the memory part of the stack, that is the address of the first free memory location in which the first local register would be saved by a push operation (see Frame instruction for details). Stack growth is from low to high address.
Bits one and zero of the SP must always be cleared to zero. The SP can be addressed only via the high global flag H being set. Copying an operand to the SP is a privileged operation.

1.5.3 Upper Stack Bound UB

G19 is the upper stack bound UB. The UB contains the address beyond the highest legal memory stack location. It is used by the Frame instruction to inhibit stack overflow.
Bits one and zero of the UB must always be cleared to zero. The UB can be addressed only via the high global flag H being set. Copying an operand to the UB is a privileged operation.

1.5.4 Bus Control Register BCR

G20 is the bus control register BCR. Its content defines the options possible for bus cycle, parity and refresh control. It is described in detail in the bus interface description in chapter 5.

1.5.5 Status Register SR

G1 is the status register SR. Its content is updated by instruction execution. Besides this implicit updating, the SR can also be addressed like a regular register. When addressed as source or destination operand, all 32 bits are used as an operand. However, only bits 15..0 of a result can be placed in bits 15..0 of the SR, bits 31..16 of the result are discarded and bits 31..16 of the SR remain unchanged. The full content of the SR is replaced only by the Return Instruction. A result placed in the SR overrules any setting or clearing of the condition flags by conditions.

1.5.6 Status Information

The status register SR contains the following status information:

C Bit zero is the carry condition flag C. In general, when set it indicates that the unsigned integer range is exceeded. At add operations, it indicates a carry out of bit 31 of the result. At subtract operations, it indicates a borrow (inverse carry) into bit 31 of the result.

Z Bit one is the zero condition flag Z. When set, it indicates that all 32 or 64 result bits are equal to zero regardless of any carry, borrow or overflow.

N Bit two is the negative condition flag N. It indicates the arithmetic correct (true) sign of the result. At add and subtract operations it is derived as N := overflow xor result bit 31, which is the same as the sign bit 31 of the result when no overflow occurs. In the case of overflow, N still reflects the correct sign while bit 31 of the result reflects the inverted sign bit.

V Bit three is the overflow condition flag V. In general, when set it indicates a signed overflow. At the Move instructions, it indicates a floating-point NaN (Not a Number).

M Bit four is the cache-mode flag M. Besides being set or cleared under program control, it is also automatically cleared by a Frame instruction and by any branch taken except a delayed branch. See instruction cache for details.

H Bit five is the high global flag H. When H is set, denoting G0..G15 addresses G16..G31 instead. Thus, the SP, UB or BCR may be addressed by denoting G2, G3 or G4 respectively.
The H flag is effective only in the first cycle of the next instruction after it was set; then it is cleared automatically.
Only the MOV or MOVI instruction issued as the next instructions are to be used to copy the content of a local register or an immediate value to the SP, the UB or the BCR. The MOV instruction may be used to copy the content of the SP or the UB to a local register. (The content of BCR cannot be copied to any register). With all other instructions, the result may be invalid.
If the SP, UB or BCR is addressed as destination in user state (S = 0), the condition flags are undefined, the destination remains unchanged and a trap to Privilege Error occurs.

1.5.6 Status Information (continued)

Reserved Bits 7..6 are reserved for future use. They must always be zero.

FTE Bits 12..8 are the floating-point trap enable flags (see floating-point instructions).

FRM Bits 14..13 are the floating-point rounding modes (see floating-point instructions).

L Bit 15 is the interrupt-lock flag L. When the L flag is one, all interrupt exceptions are inhibited. The state of the L flag is effective immediately after any instruction which changed it.
The L flag can be cleared or kept set in any or on return to any privilege state (user or supervisor). Changing the L flag from zero to one is privileged to supervisor or return from supervisor to supervisor state. A trap to Privilege Error occurs if the L flag is set under program control from zero to one in user or on return to user state.

1.5.6 Status Information (continued)

The following status information can be changed only internally or replaced by the saved return value of the SR via a Return instruction:

T Bit 16 is the trace-mode flag T. When both the T flag and the trace pending flag P are one, a trace exception occurs after every instruction except after a Delayed Branch instruction.
Note: The T flag can only be changed in the saved return SR and is then effective after execution of a Return instruction.

P Bit 17 is the trace pending flag P. It is automatically set to one by all instructions except by the Return instruction, which restores the P flag from bit 17 of the saved return SR.
Since for a Trace exception both the P and the T flag must be one, the P flag determines whether a trace exception occurs (P = 1) or does not occur (P = 0) immediately after a Return instruction which restored the T flag to one.
Note: The P flag can only be changed in the saved SR. No program except the trace exception handler should affect the saved P flag. The trace exception handler must clear the saved P flag to prevent a trace exception on return, in order to avoid tracing the same instruction in an endless loop.

S Bit 18 is the supervisor state flag S (see privilege states).

ILC Bits 20 and 19 represent the instruction-length code ILC. It is updated by instruction execution. The ILC holds (in general) the length of the last instruction: ILC values of one, two or three represent an instruction length of one, two or three halfwords respectively. After a branch taken, the ILC is invalid. The Return instruction clears the ILC.
Note: Since a Return instruction following an exception clears the ILC, a program must not rely on the current value of the ILC.

FL Bits 24..21 represent the frame length FL. The FL holds the number of usable local registers (maximum 16) assigned to the current stack frame.
FL = 0 is always interpreted as FL = 16.

FP Bits 31..25 represent the frame pointer FP. The least significant six bits of the FP point to the beginning of the current stack frame in the local register set, that is, they point to L0.
The FP contains bit 8..2 of the address at which the content of L0 would be stored if pushed onto the memory part of the stack.

1.5.7 Privilege States

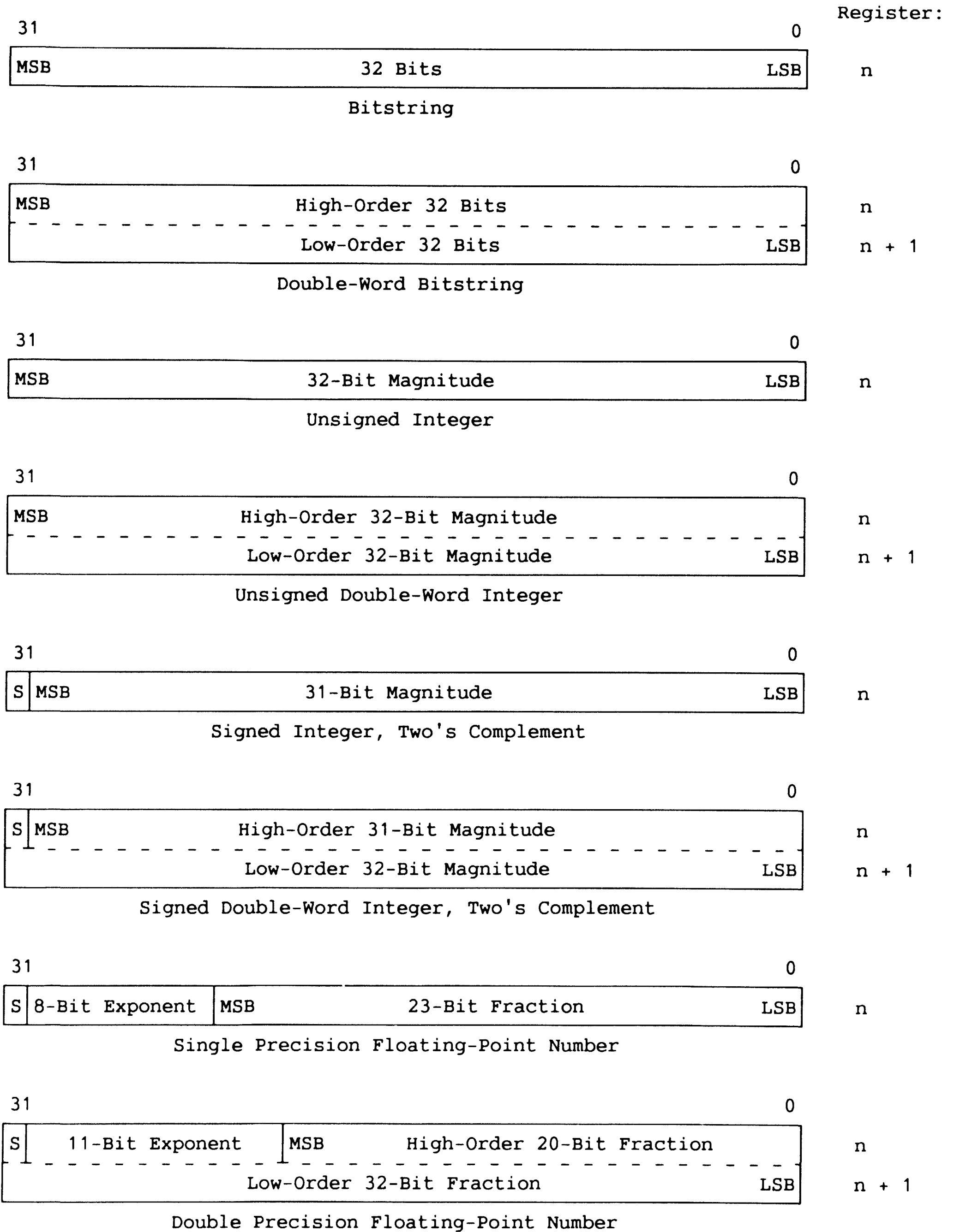
The architecture provides two privilege states, determined by the supervisor state flag S: User state (S = 0) and supervisor state (S = 1).
The privilege state is used by the (external) memory management to control memory and I/O accesses. The operating system kernel is executed in the higher privileged supervisor state, thereby restricting access to all sensitive data to a highly reliable system program. The following operations are also privileged to be executed only in the supervisor or on return from supervisor to supervisor state:

- Copying an operand to the SP, UB or BCR
- Changing the interrupt-lock flag L from zero to one
- Returning through a Return instruction to supervisor state

Any illegal attempt causes a trap to Privilege Error.
The S flag is also saved in bit zero of the saved return PC by the Call, Trap and Software instructions and by an exception. A Return instruction restores it from this bit position to the S flag in bit position 18 of the SR (thereby overwriting the bit 18 returned from the saved return SR).
If a Return instruction attempts a return from user to supervisor state, a trap to Privilege Error occurs (S = 1 is saved).
Returning from supervisor to user state is achieved by clearing the S flag in bit zero of the saved return PC before return. Switching from user to supervisor state is only possible by executing a Trap instruction or by exception processing through one of the 64 supervisor subprogram entries (see entry table).

Note: Since the Return instruction restores the PC first to enable the instruction fetch to start immediately, the restored S flag must also be available immediately to prevent any memory access with a false privilege state. The S flag is therefore packed in bit zero of the saved return PC.
The state of the S flag is signalled at the corresponding pin in each memory or I/O cycle.

1.6 Register Data Types



S = sign bit, MSB = most significant bit, LSB = least significant bit

1.7 Memory Organization

The architecture provides a memory address space in the range of $0..2^{32} - 1$ (0..4 294 967 295) 8-bit bytes. Memory is implied to be organized as 32-bit words. Besides the memory address space, a separate I/O address space is provided. The following memory data types are available:

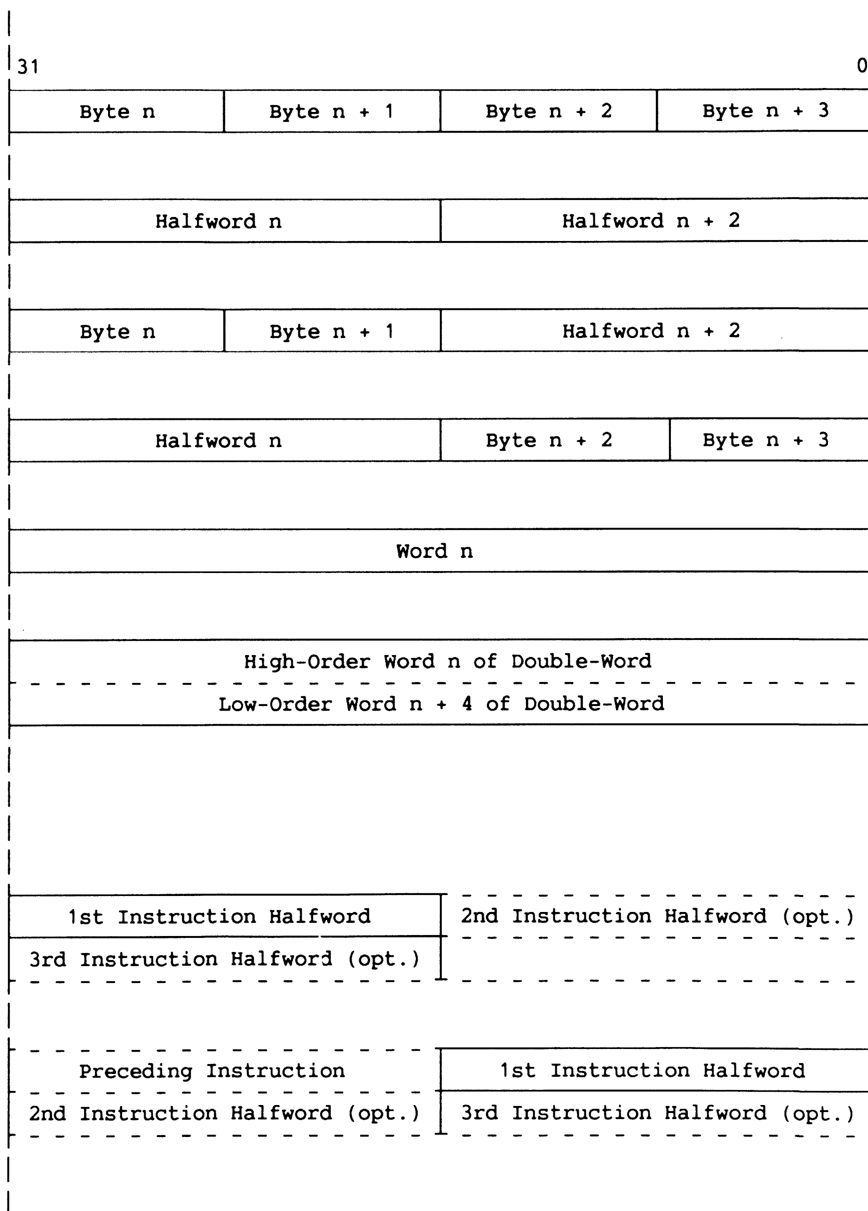
- Byte unsigned (unsigned 8-bit integer, bitstring or character)
- Byte signed (signed 8-bit integer, two's complement)
- Halfword unsigned (unsigned 16-bit integer or bitstring)
- Halfword signed (signed 16-bit integer, two's complement)
- Word (32-bit undedicated word)
- Double-Word (64-bit undedicated double-word)

At I/O address space, only word and double-word data types are available.
Words and double-words must be located at word boundaries, that is, their most significant byte must be located at an address whose two least significant bits are zero. Halfwords must be located at halfword boundaries, their most significant byte being located at an address whose least significant bit is zero. Bytes may be located at any address.
The variable-length instructions are located as contiguous sequences of one, two or three halfwords at halfword boundaries.
Memory accesses are pipelined to an implied depth of two addresses. Virtual memory using demand paging may be implemented by an off-chip MMU (memory management unit). A page fault signal from the MMU then causes a trap to the exception subprogram Data Page Fault or Instruction Page Fault immediately after the fault-causing memory instruction is executed or the missing instruction is decoded respectively. After loading the missing page from disk into memory and correcting a possibly updated memory address, the fault causing instruction can then be repeated.

Note: All data is located high to low order at addresses ascending from low to high, that is, the high order part of all data is located at the lower address. This scheme should also be used for the addressing of bit arrays. Though the most significant bit of a word is numbered as bit position 31 for convenience of use, it should be assigned the bit address zero to maintain consistent bit addressing in ascending order through word boundaries.

1.7 Memory Organization (continued)

The figure below shows the location of data and instructions in memory relative to a binary address n = ...xxx00 (x = 0 or 1).



At all data types, the most significant bit is located at the higher and the least significant bit at the lower bit position.

1.8 Stack

A runtime stack, called stack here, holds generations of local variables in last-in-first-out order. A generation of local variables, called stack frame or activation record, is created upon subprogram entry and released upon subprogram return.
The runtime stack provided by the architecture is divided into a memory part and a register part. The register part of the stack, implemented by a set of 64 local registers organized as a circular buffer, holds the most recent stack frame(s). The current stack frame is always kept in the register part of the stack. The frame pointer FP points to the beginning of the current stack frame (addressed as L0). The frame length FL indicates the number of registers (maximum 16) assigned to the current stack frame. The stack grows from low to high address. It is guarded by the upper stack bound UB.
The stack is maintained as follows:

- A Call, Trap or Software instruction increments the FP and sets FL to six, thus creating a new stack frame with a length of six registers (including the return PC and the return SR).
- An exception increments the FP by the value of FL and then sets FL to two.
- A Frame instruction restructures a stack frame to include (optionally) passed parameters by decrementing the FP and by resetting the FL, and restores a reserve of 10 local registers for the next subprogram call. If the required number of registers + 10 do not fit in the register part of the stack, the contents of the differential (required + 10 - available) number of local registers are pushed onto the memory part of the stack. A trap to Frame Error occurs after the push operation when the old value of the stack pointer SP exceeded the upper stack bound UB.
- A Return instruction releases the current stack frame and restores the preceding stack frame. If the restored stack frame is not fully contained in the register part of the stack, the content of the missing part of the stack frame is pulled from the memory part of the stack.

For more details see the descriptions of the specific instructions.
When the number of local registers required for a stack frame exceeds its maximum length of 16 (in rare cases), a second runtime stack in memory may be used. This second stack is also required to hold local record or array data.
The stack is used by routines in user or supervisor state, that is, supervisor stack frames are appended to user stack frames, and thus, parameters can be passed between user and supervisor state. A small stack space must be reserved above UB. UB can then be set to a higher value by the Frame Error handler to free stack space for error handling.

1.8 Stack (continued)

The figure below shows the creation and release of stack frames in the register part of the stack.

Return from B

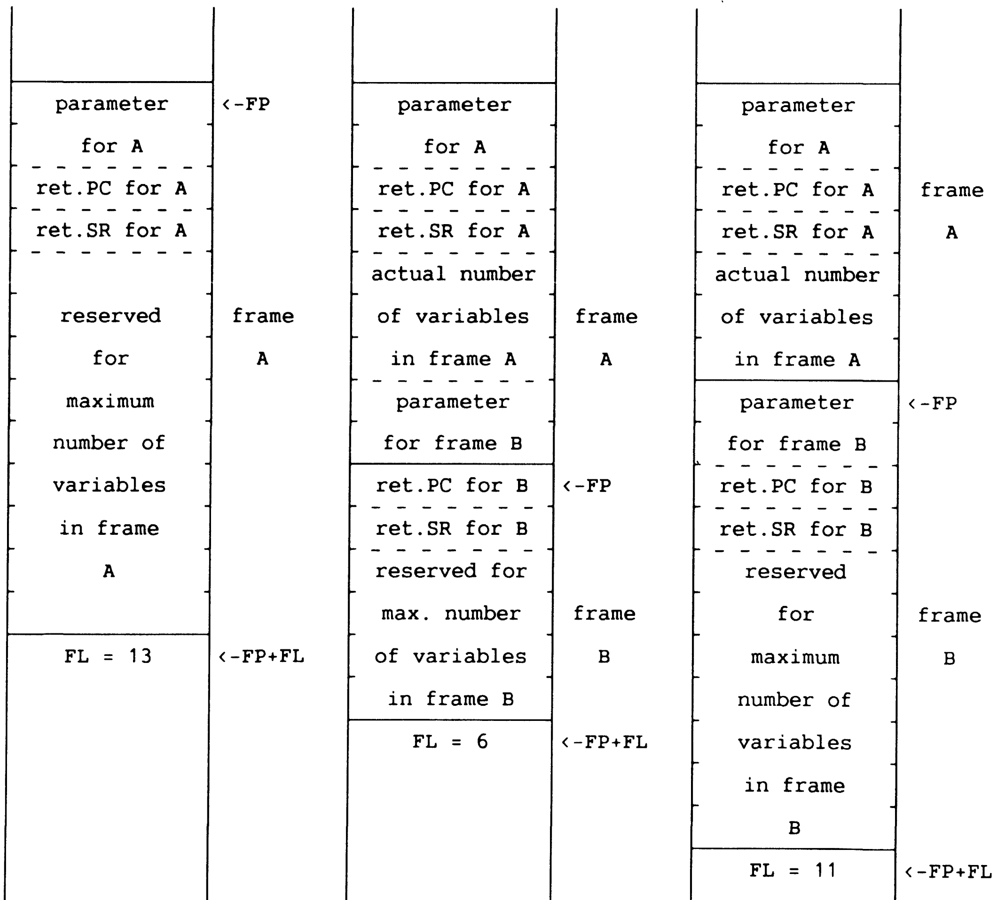
```
PC := ret.PC B;
SR := ret.SR B;
-- returns preceding
   stack frame
if stack frame contained
   in local registers
then next instruction;
else pull contents of
   differential words from
   memory part of stack;
```

Call B

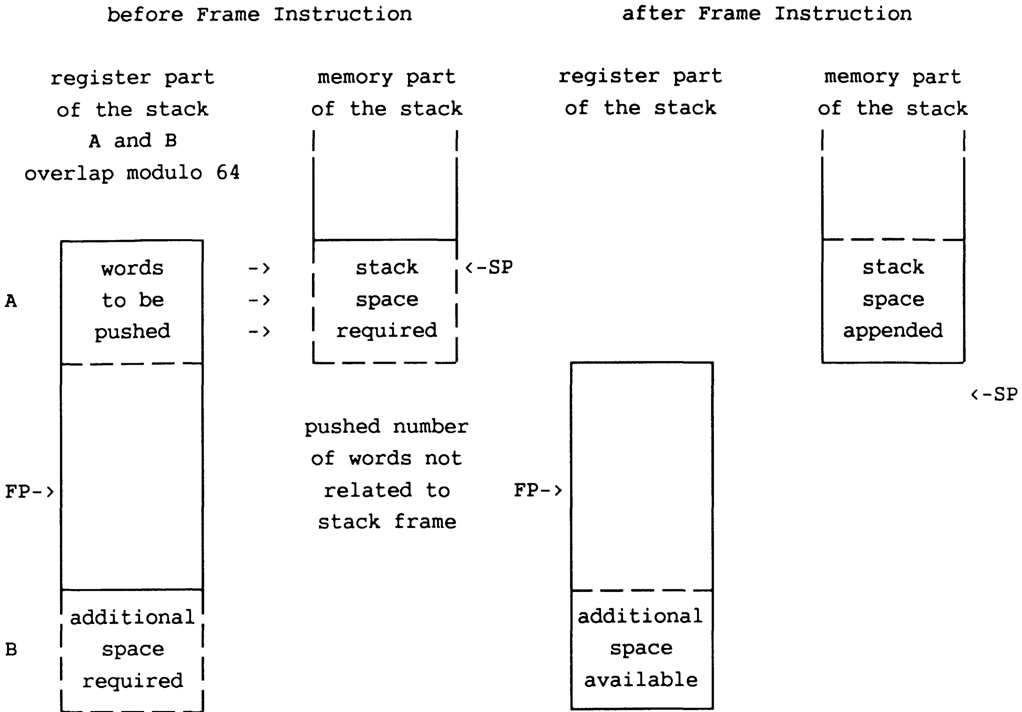
```
PC := branch address;
ret.PC B := old PC;
ret.SR B := old SR;
FP := FP +
   ret.PC reg.code;
FL := 6;
-- ret.PC reg.code = 9
```

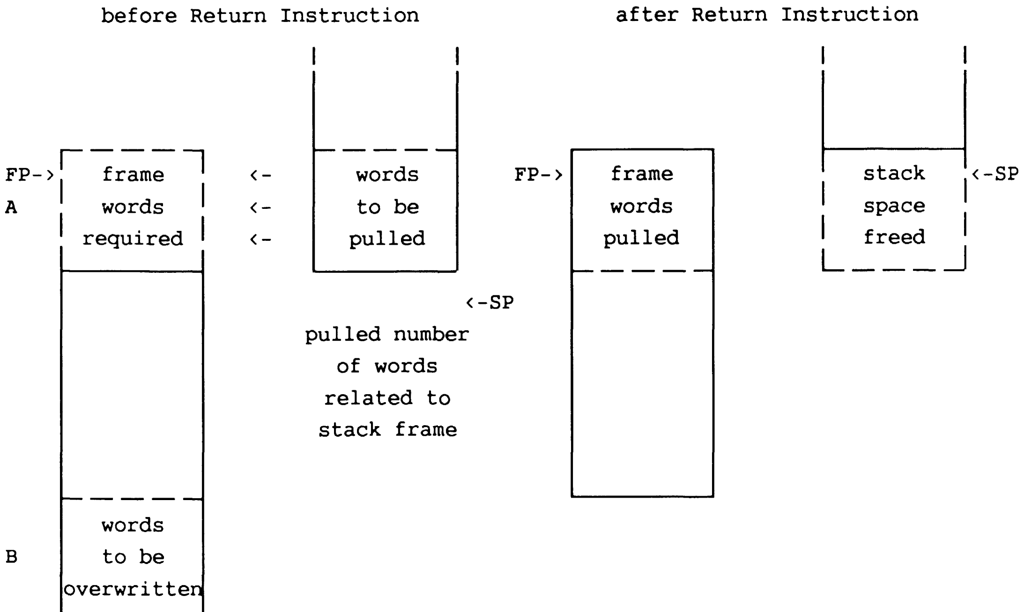
Frame in B

```
FP := FP - source reg.code;
FL := dest. reg.code;
if available registers >=
   (required + 10) registers
then next instruction;
else push contents of
   differential number of
   registers to memory
   part of stack;
-- source reg.code = 2
-- dest. reg.code = 11
```



1.8 Stack (continued)





1.9 Instruction Cache

The instruction cache holds a total of up to 128 bytes (32 unstructured 32-bit words of instructions). It is implemented as a circular buffer which is guarded by a look-ahead counter and a look-back counter. The look-ahead counter holds the highest and the look-back counter the lowest address of the instruction words available in the cache. The cache-mode flag M is used to optimize special cases in loops (see details below). The cache can be regarded as a temporary local window into the instruction sequence, moving along with instruction execution and being halted by the execution of a program loop.
Its function is as follows:
The prefetch control loads unstructured 32-bit instruction words (without regard to instruction boundaries) from memory into the cache. The load operation is pipelined to a depth of two stages (see memory instructions for details of the load pipeline). The look-ahead counter is incremented by four at each prefetch cycle. It always contains the address of the last instruction word for which an address bus cycle is initiated, regardless of whether the addressed instruction word is in the load pipeline or already loaded into the instruction cache.
The prefetched instruction word is placed in the cache word location addressed by bits 6..2 of the look-ahead counter. The look-back counter remains unchanged during prefetch unless the cache word location it addresses with its bits 6..2 is overwritten by a prefetched instruction word. In this case, it is incremented by four to point to the then lowest-addressed usable instruction word in the cache. Since the cache is implemented as a circular buffer, the cache word addresses derived from bits 6..2 of the look-ahead and look-back counter wrap around modulo 32.

The prefetch is halted:

- When eight words are prefetched, that is, eight words are available (including those pending in the load pipeline) in the prefetch sequence succeeding the instruction word addressed by the program counter PC through the instruction word addressed by the look-ahead counter. Prefetch is resumed when the PC is advanced by instruction execution.
- In the cycle preceding the execution cycle of a memory instruction or any potentially branch-causing instruction (regardless of whether the branch is taken) except a forward Branch or Delayed Branch instruction with an instruction length of one halfword and a branch target contained in the cache. Halting the prefetch in these cases avoids filling the load pipeline with demands for lower priority (compared to data) or potentially unnecessary instruction words. The pre-fetch is also halted during the execution cycle of any instruction accessing memory or I/O.

1.9 Instruction Cache (continued)

- When a Page Fault is signalled on the attempted fetch of an instruction word. In this case, the look-ahead counter holds the address of the fault-causing instruction word and an internal Page-Fault flag is set. The trap to Page Fault occurs only when the flagged instruction word address is met in the decoding stage prior to execution.

The cache is read in the decode cycle by using bits 6..1 of the PC as an address to the first halfword of the instruction presently being decoded. The instruction decode needs and uses only the number (1, 2 or 3) of instruction halfwords defined by the instruction format. Since only the bits 6..1 of the PC are used for addressing, the halfword addresses wrap around modulo 64. Idle wait cycles are inserted when the instruction is not or not fully available in the cache.
At an explicit Branch or Delayed Branch instruction (except when placed as delay instruction) with an instruction length of one halfword, the location of the branch target is checked. The branch target is treated as being in the cache when the target address of a backward branch is not lower than the address in the look-back counter and the target address of a forward branch is not higher than two words above the address in the look-ahead counter. That is, the two instruction words succeeding the instruction word addressed by the content of the look-ahead counter are treated by a forward branch as being in the cache. Their actual fetch overlaps in most cases with the execution of the branch instruction and thus, no cycles are wasted. When the branch target is in the cache, the look-back counter, the look-ahead counter and the internal Page Fault flag remain unchanged.
When a branch is taken by a Delayed Branch instruction with an instruction length of one halfword to a forward branch target not in the cache and the cache mode flag M is enabled (1), the look-back counter, the look-ahead counter and the internal Page Fault flag remain unchanged. Wait cycles are then inserted until the ongoing prefetch has loaded the branch target instruction into the cache. A Page Fault flag set or a Page Fault signal causes an immediate trap to Instruction Page Fault in this case.
Any other branch taken flushes the cache by also placing the branch address in the look-back and the look-ahead counter; the internal Page Fault flag is cleared. Prefetch then starts immediately at the branch address. Instruction decoding waits until the branch target instruction is fully available in the cache.
The cache mode flag M (bit four of the SR) can be set or cleared by logical instructions. It is automatically cleared by a Frame instruction and by any branch taken except a branch caused by a Delayed Branch or Return instruction; a Delayed Branch instruction leaves the M flag unchanged and a Return instruction restores the M flag from the saved status register SR.

1.9 Instruction Cache (continued)

Note: The instruction cache is transparent to programs. A program executes correctly even if it ignores the cache, whereby it is assumed that the instruction code is not modified in the local range contained in the cache.
Since up to eight instruction words can be loaded into the cache by the prefetch, only 24 instruction words are left to be contained in a program loop. Thus, a program loop can have a maximum length of 96 or 94 bytes including the branch instruction closing the loop, depending on the even or odd halfword address location of the first instruction of the loop respectively.
A forward Branch or Delayed Branch instruction with an instruction length of one halfword into up to two instruction words succeeding the word addressed by the look-ahead counter treats the branch target as being in the cache and does not flush the cache. Thus, three or four instruction halfwords, depending on the odd or even half-word address location of the branch instruction respectively, can always be skipped without flushing the cache.
Enabling the cache-mode flag M is only required when a program loop to be contained in the cache contains a forward branch to a branch target in the program loop and more than three (or four, see above) instruction halfwords are to be skipped. In this case, the enabled M flag in combination with a Delayed Branch instruction with an instruction length of one halfword inhibits flushing the cache when the branch target is not yet prefetched.
Since a single-word memory instruction halts the prefetch for two cycles, any sequence of memory instructions, even with interspersed one-cycle non-memory instructions, halts the prefetch during its execution. Thus, alternating between instruction and data memory pages is avoided. If the number of instruction halfwords required by such a sequence is not guaranteed to be in the cache at the beginning of the sequence, a Fetch instruction enforcing the prefetch of the sequence may be used. A Fetch instruction may also be used preceding a branch into a program loop; thus, flushing the cache by the first branch repeating the loop can be avoided.
A branch taken caused by a Branch or Delayed Branch instruction with an instruction length of two halfwords always flushes the instruction cache, even if the branch target is in the cache. Thus, branches can be forced to bypass the cache, thereby reducing the cache to a prefetch buffer. This reduced function can be used for testing.
The last nine words of a memory block containing instructions must not contain any instruction to be executed, otherwise the prefetch could overrun the memory limit.

2 Instructions General

2.1 Instruction Notation

In the following instruction-set presentation, an informal description of an instruction is followed by a formal description in the form:

Format Notation Operation

Format denotes the instruction format.

Notation gives the assembler notation of the instruction.

Operation describes the operation in a Pascal-like notation with the following symbols:

Ls denotes any of the local registers L0..L15 used as source register or as source operand. At memory Load instructions, Ls denotes the load destination register.

Ld denotes any of the local registers L0..L15 used as destination register or as destination operand.

Rs denotes any of the local registers L0..L15 or any of the global registers G0..G15 used as source register or as source operand. At memory Load, see Ls.

Rd denotes any of the local registers L0..L15 or any of the global registers G0..G15 used as destination register or as destination operand.

Lsf, Ldf, Rsf and Rdf denote the register or operand following after (with a register address one higher than) Ls, Ld, Rs and Rd respectively.

imm, const, dis, lim, rel, adr and n denote immediate operands (constants) of various formats and ranges.

Operand(x) denotes a single bit at the bit position x of an operand.

Example: Ld(31) denotes bit 31 of Ld.

Operand(x..y) denotes bits x through y of an operand.

Example: Ls(4..0) denotes bits 4 through 0 of Ls.

Expression^ denotes an operand at a location addressed by the value of the expression. Depending on the context, the expression addresses a memory location or a local register.

Example: Ld^ denotes a memory operand whose memory address is the operand Ld. (FP + FL)^ denotes a local register operand whose register address is FP + FL.

:= signifies the assignment symbol, read as "is replaced by".

// signifies the concatenation symbol. It denotes concatenation of two operand words to a double-word operand or concatenation of bits and bitstrings.

Example: Ld//Ldf denotes a double-word operand,

16 zeros//imm1 denotes zero expanding of an immediate halfword.

=, /=, > and < denote the equal, unequal, greater than and less than relations.

Example: The relation Ld = 0 evaluates to one if Ld is equal to zero, otherwise it evaluates to zero.

2.2 Instruction Execution

At instruction execution, all bits of the operands participate in the operations, except at the Shift and Rotate instructions (whereat only the 5 least significant bits of the source operand are used) and except at the byte and halfword Store instructions.
Instructions are executed by a two-stage pipeline. In the first stage, the instruction is fetched from the instruction cache and decoded. In the second stage, the instruction is executed while the next instruction in the first stage is already decoded.
At register instructions executing in one or two cycles, the corresponding source and destination operand words are read from their registers and evaluated in each cycle in which they are used. Then the result word is placed in the corresponding destination register in the same cycle. Thus, at all single-word register instructions executing in one cycle, the source operand register and the destination operand register may coincide without changing the effect of the instruction. At all other instructions, the effect of a register coincidence depends on execution order and must be examined specifically for each such instruction.
The content of a source register remains unchanged unless it is used coincidentally as a destination register (except at memory Load instructions).
Some instructions set or clear condition flags according to the result and special conditions occuring during their execution. The conditions may be expressed by single bits, relations or logical combinations of these. If a condition evaluates to one (true), the corresponding condition flag is set to one, if it evaluates to zero (false), the corresponding condition flag is cleared to zero. Unless specified otherwise, a trap to Range Error occurs after the flags and the destination are updated.
All instructions may use the result and any flags updated by the preceding instruction. A time penalty occurs only if the result of a memory Load or Exchange instruction is not yet available when needed as destination or source operand. In this case one or more (depending on the memory access time) idle wait cycles are enforced by a hardware interlock.
An instruction must not use any local register of the register sequence beginning with L0 beyond the number of usable registers specified by the current value of the frame length FL (FL = 0 is interpreted as FL = 16). That is, the value of the corresponding register code (0..15) addressing a local register must be lower than the interpreted value of the FL (except with a Call or Frame instruction or some restricted cases). Otherwise, an exception could overwrite the contents of such a register or the beginning of the register part of the stack at the SP could be overwritten without any warning when a result is placed in such a register.
Double-word instructions denote the high-order word (at the lower address). The low-order word adjacently following it (at the higher address) is implied.
"Old" denotes the state before the execution of an instruction.

2.3 Instruction Formats

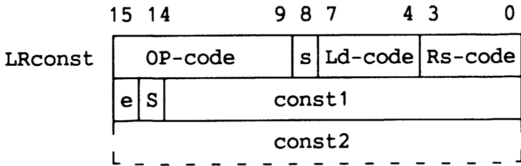
Instructions have a length of one, two or three halfwords and must be located on halfword boundaries. The following formats are provided:

Format	Configuration	
LL	15 … 8: OP-code | 7 … 4: Ld-code | 3 … 0: Ls-code	Ls-code encodes L0..L15 for Ls Ld-code encodes L0..L15 for Ld
LR	15 … 9: OP-code | 8: s | 7 … 4: Ld-code | 3 … 0: Rs-code	s = 0: Rs-code encodes G0..G15 for Rs s = 1: Rs-code encodes L0..L15 for Rs Ld-code encodes L0..L15 for Ld
RR	15 … 10: OP-code | 9: d | 8: s | 7 … 4: Rd-code | 3 … 0: Rs-code	s = 0: Rs-code encodes G0..G15 for Rs s = 1: Rs-code encodes L0..L15 for Rs d = 0: Rd-code encodes G0..G15 for Rd d = 1: Rd-code encodes L0..L15 for Rd
Ln	15 … 9: OP-code | 8: n | 7 … 4: Ld-code | 3 … 0: n	Ld-code encodes L0..L15 for Ld n: Bit 8//bits 3..0 encode n = 0..31
Rn	15 … 10: OP-code | 9: d | 8: n | 7 … 4: Rd-code | 3 … 0: n	d = 0: Rd-code encodes G0..G15 for Rd d = 1: Rd-code encodes L0..L15 for Rd n: Bit 8//bits 3..0 encode n = 0..31
PCadr	15 … 8: OP-code | 7 … 0: adr-byte	adr = 24 ones's//adr-byte(7..2)//00
PCrel	15 … 8: OP-code | 7: 0 | 6 … 1: low-rel | 0: S	S: sign bit of rel rel = 25 S//low-rel//0 range -128..126
PCrel	15 … 8: OP-code | 7: 1 | 6 … 0: high-rel low-rel | S	S: sign bit of rel rel = 9 S//high-rel//low-rel//0 range -8 388 608..8 388 606

2.3 Instruction Formats (continued)

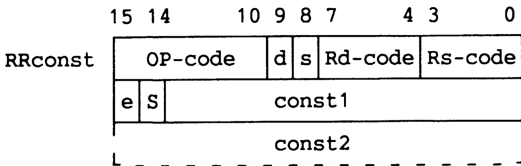
Format Configuration

LRconst



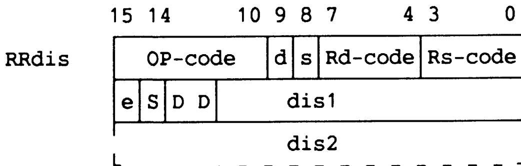
s = 0: Rs-code encodes G0..G15 for Rs
s = 1: Rs-code encodes L0..L15 for Rs
Ld-code encodes L0..L15 for Ld
S: Sign bit of const
e = 0: const = 18 S//const1
range -16 384..16 383
e = 1: const = 2 S//const1//const2
range -1 073 741 824..1 073 741 823

RRconst



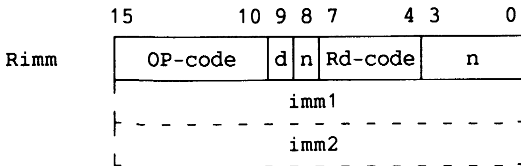
s = 0: Rs-code encodes G0..G15 for Rs
s = 1: Rs-code encodes L0..L15 for Rs
d = 0: Rd-code encodes G0..G15 for Rd
d = 1: Rd-code encodes L0..L15 for Rd
S: Sign bit of const
e = 0: const = 18 S//const1
range -16 384..16 383
e = 1: const = 2 S//const1//const2
range -1 073 741 824..1 073 741 823

RRdis



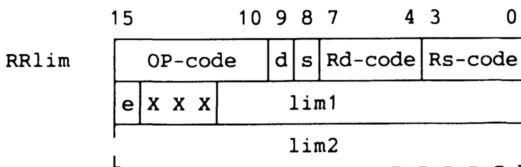
s = 0: Rs-code encodes G0..G15 for Rs
s = 1: Rs-code encodes L0..L15 for Rs
d = 0: Rd-code encodes G0..G15 for Rd
d = 1: Rd-code encodes L0..L15 for Rd
S: Sign bit of dis
e = 0: dis = 20 S//dis1
range -4 096..4 095
e = 1: dis = 4 S//dis1//dis2
range -268 435 456..268 435 455
DD: D-code, D13..D12 encode data types at memory instructions

Rimm



d = 0: Rd-code encodes G0..G15 for Rd
d = 1: Rd-code encodes L0..L15 for Rd
n: Bit 8//bits 3..0 encode n = 0..31
see table immediate values for encoding of imm

RRlim



s = 0: Rs-code encodes G0..G15 for Rs
s = 1: Rs-code encodes L0..L15 for Rs
d = 0: Rd-code encodes G0..G15 for Rd
d = 1: Rd-code encodes L0..L15 for Rd
XXX: X-code, X14..X12 encode Index instructions
e = 0: lim = 20 zeros//lim1
range 0..4 095
e = 1: lim = 4 zeros//lim1//lim2
range 0..286 435 455

2.3.1 Table of Immediate Values

n	immediate value imm	
0..16	0..16	-- at CMPBI, n = 0 encodes ANYBZ at ADDI and ADDSI n = 0 encodes CZ
17	imm1//imm2	-- range = $0..2^{32}-1$ or $-2^{31}..2^{31}-1$
18	16 zeros//imm1	-- range = 0..65 535
19	16 ones//imm1	-- range = -65 536..-1
20	32	-- bit 5 = 1, all other bits = 0
21	64	-- bit 6 = 1, all other bits = 0
22	128	-- bit 7 = 1, all other bits = 0
23	2^{31}	-- bit 31 = 1, all other bits = 0
24	-8	
25	-7	
26	-6	
27	-5	
28	-4	
29	-3	
30	-2	
31	$2^{31}-1$	at CMPBI and ANDNI -- bit 31 = 0, all other bits = 1
31	-1	at all other instructions using imm

Note: 2^{31} provides clear, set and invert of the floating-point sign bit at ANDNI, ORI and XORI respectively.
$2^{31}-1$ provides a test for floating-point zero at CMPBI and extraction of the sign bit at ANDNI.
See CMPBI for ANYBZ and ADDI, ADDSI for CZ.

2.3.2 Table of Instruction Codes

OP-code Bits 15..12 (rows) / OP-code Bits 11..8 (columns)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	CHK,CHKZ,NOP				MOVD,RET				DIVU				DIVS			
1	XMx,XMxZ				MASK				SUM				SUMS			
2	CMP				MOV				ADD				ADDS			
3	CMPB				ANDN				OR				XOR			
4	SUBC				NOT				SUB				SUBS			
5	ADDC				AND				NEG				NEGS			
6	CMPI				MOVI				ADDI				ADDSI			
7	CMPBI				ANDNI				ORI				XORI			
8	SHRDI		SHRD	SHR	SARDI		SARD	SAR	SHLDI		SHLD	SHL	SALDI		TESTLZ	ROL
9	LDxx.D/A/IOD/IOA				LDxx.N/S				STxx.D/A/IOD/IOA				STxx.N/S,XCHW			
A	SHRI				SARI				SHLI				SALI			
B	MULU				MULS				SETxx,SETADR,FETCH				MUL			
C	FADD	FADDD	FSUB	FSUBD	FMUL	FMULD	FDIV	FDIVD	FCMP	FCMPD	FCMPU	FCMPUD	FCVT	FCVTD	EX	DO
D	LDW.R		LDD.R		LDW.P		LDD.P		STW.R		STD.R		STW.P		STD.P	
E	DBV	DBNV	DBE	DBNE	DBC	DBNC	DBSE	DBHT	DBN	DBNN	DBLE	DBGT	DBR	FRAME	CALL,CALLV	
F	BV	BNV	BE	BNE	BC	BNC	BSE	BHT	BN	BNN	BLE	BGT	BR	TRAPxx,TRAP		

2.4 Entry Table

The table below shows the addresses of the first instruction of the subprogram associated with each Software instruction, Trap instruction and exception.

Address (Hex)	Entry	
FFFF FE00	FADD	
FFFF FE10	FADDD	
FFFF FE20	FSUB	
FFFF FE30	FSUBD	
FFFF FE40	FMUL	
FFFF FE50	FMULD	
FFFF FE60	FDIV	
FFFF FE70	FDIVD	
FFFF FE80	FCMP	
FFFF FE90	FCMPD	
FFFF FEA0	FCMPU	
FFFF FEB0	FCMPUD	
FFFF FEC0	FCVT	
FFFF FED0	FCVTD	
FFFF FEE0	EX	
FFFF FEF0	DO	
FFFF FF00	TRAP -> 0	
FFFF FF04	TRAP -> 1	
	.	
FFFF FFE0	TRAP -> 56	Instruction Page Fault -- lowest priority
FFFF FFE4	TRAP -> 57	Trace exception
FFFF FFE8	TRAP -> 58	Interrupt exception
FFFF FFEC		Reserved for interrupt handler instruction(s)
FFFF FFF0	TRAP -> 60	Range, Pointer, Frame and Privilege Error
FFFF FFF4	TRAP -> 61	Data Page Fault
FFFF FFF8	TRAP -> 62	Reset -- highest priority
FFFF FFFC	TRAP -> 63	-- error entry for instruction code of all ones

Note: Spacing of the entries for the Software instructions FADD..DO is 16 bytes. The associated subprogram head is intended to be placed directly at the corresponding entry.

Spacing of the entries for the Trap instructions and exceptions is four bytes. These entries are intended to each contain an instruction branching to the associated subprogram. The entries for the TRAPxx instructions are the same as for TRAP. The exception entries are ordered by priority.

Application Note: The Trap entries 32..63 are reserved for system software.

2.5 Instruction Timing

The following execution times are given in clock cycles.

All instructions not shown below: 1 cycle
Move Double-Word: 2 cycles
Shift Double-Word: 2 cycles
Test Leading Zeros: 2 cycles
Multiply Word and Multiply Double-Word signed: max. 24 cycles
 average (rounded up to nearest integer): 17 cycles
Multiply Double-Word unsigned: max. 34 cycles
 average (rounded up to nearest integer): 17 cycles
Divide unsigned and signed: 36 cycles
Branch instructions when branch not taken : 1 cycle
 when branch taken and target in on-chip cache : 2 cycles
 when branch taken and target in memory : 2 + memory read latency cycles
 (see 2-9)
Delayed Branch instructions when branch not taken : 1 cycle
 when branch taken and target in on-chip cache : 1 cycle
 when branch taken and target in memory : 1 + memory read latency cycles
 exceeding (delay instruction cycles - 1) cycles
Call and Trap instructions when branch not taken: 1 cycle
 when branch taken: 2 + memory read latency cycles
Software instructions: 6 + memory read latency cycles exceeding 4 cycles
Frame when not pushing words into the stack : 3 cycles
 additionally when pushing n words into the stack: memory write latency cycles
 + n cycles
 -- fast page mode storing a word in each cycle is implied
 -- write latency cycles = cycles elapsed until write access cycle of first word
 stored (minimum = 1 at a non-RAS access and no pipeline congestion)
Return:
 4 + memory read latency cycles exceeding 2 cycles
 additionally when pulling n words out of the stack: memory RAS latency cycles
 + n cycles
 (RAS latency applies only at n > 2, otherwise RAS latency is always 0)
 -- fast page mode reading a word in each cycle is implied
 -- RAS latency = RAS precharge cycles + RAS to CAS delay cycles

2.5 Instruction Timing (continued)

Memory word instructions, non-stack address mode:

Non-RAS access or Page Fault bit (in BCR) disabled: 1 cycle

RAS access and Page Fault bit enabled: 1 + RAS latency cycles

Memory word instructions, stack address mode:

Register access: 3 cycles

Non-RAS memory access or Page Fault bit disabled: 3 cycles

RAS access and Page Fault bit enabled: 3 + RAS latency cycles

Memory double word instructions:

Non-RAS access or Page Fault bit disabled: 2 cycles

RAS access and Page Fault bit enabled: 2 + RAS latency cycles

For timing calculations, double-word memory instructions are treated like a sequence of two single-word memory instructions.

Instruction execution proceeds after the execution of a Load or Exchange instruction until the data requested is needed (that is, the register into which the data is to be loaded is addressed) by a further instruction.

The cycles executed between the memory instruction cycle requesting the data and the first cycle at which the data are available are called read latency cycles. These read latency cycles can be filled with instructions which do not need the requested data. When, after the execution of these optional fill instruction cycles, the data is still not available in the cycle needing it, idle wait cycles are inserted until the data is available. The idle wait cycles are inserted transparently to the program by an on-chip hardware interlock.

Idle wait cycles are also transparently inserted when a memory instruction has to wait for execution because the two-stage address pipeline is full.

At a non-RAS memory or an I/O access, the read latency is:

read latency = address setup cycles + access cycles

At a RAS memory access, the read latency is:

read latency = RAS precharge cycles + RAS to CAS delay cycles
+ access cycles

Additional cycles are also inserted and add to the latency when the address pipeline is congested, these cycles must then also be taken into calculation.

A switch from a memory or I/O store operation to an immediately succeeding load operation or a bus master switch from a preceding load or store operation to an immediately succeeding load operation inserts one additional bus cycle.

A switch from a memory or I/O load operation, with or without a bus master switch, to an immediately succeeding store operation inserts two additional wait cycles.

3 Instruction Set

3.1 Memory Instructions

The memory instructions load data from memory in a register Rs (or a register pair Rs//Rsf) or store data from Rs (or Rs//Rsf) to memory using the data types byte unsigned/signed, halfword unsigned/signed, word or double-word, or exchange a data word. Since I/O devices are also addressed by memory instructions, "memory" stands here interchangeably also for I/O unless memory or I/O address space is specifically denoted.

The memory address is either specified by the operand Rd or Ld, by the sum Rd plus a signed displacement or by the displacement alone, depending on the address mode. Memory accesses to words and double-words ignore bits one and zero of the address, memory accesses to halfwords ignore bit zero of the address, (since these operands are located at word or halfword boundaries respectively, these address bits are redundant).

If the content of any register Rd except SR is zero, the memory is not accessed and a trap to Pointer Error occurs (see exceptions). Thus, uninitialized pointers are automatically checked.

Load, Store and Exchange instructions are pipelined to a total depth of two word entries for Load, Store and Exchange, thus, a double-word Load or a double-word Store instruction can be executed without halting the CPU in a wait state (The address pipeline provides a depth of two addresses common to load and store).

Double-word memory instructions enter two separate word entries into the pipeline and start two independent memory cycles. The first memory cycle, loading or storing the high-order word, uses the address specified by the address mode, the second cycle uses this address incremented by four and also places it on the address bus.

A Page Fault signal from the memory control circuits (off chip) causes a trap to the Data Page Fault routine.

Since a double-word instruction starts separate memory cycles, a Page Fault can be signalled for the first or second word in memory independently, enabling the handling of a Page Fault caused by a double-word instruction where the second word is in a non-resident page.

Accessing data in the same DRAM page by any number of succeeding memory cycles is performed in page mode, automatically directed by a control signal from the CPU.

Memory instructions leave all condition flags unchanged.

3.1.1 Address Modes

Register Address Mode:

Notation: LDxx.R, STxx.R -- xx: word or double word data type

The content of the destination register Ld is used as an address into memory address space.

Postincrement Address Mode:

Notation: LDxx.P, STxx.P -- xx: word or double-word data type

The content of the destination register Ld is used as an address into memory address space, then Ld is incremented according to the specified data size of a word or double-word memory instruction by 4 or 8 respectively, regardless of any exception occuring. In the case of a double-word data type, Ld is incremented by 8 at the first memory cycle.

Displacement Address Mode:

Notation: LDxx.D, STxx.D -- xx: any data type

The sum of the contents of the destination register Rd plus a signed displacement dis is used as an address into memory address space.
Rd may denote any register except the SR; Rd not denoting the SR differentiates this mode from the absolute address mode.
In the case of all data types except byte, bit zero of dis is treated as zero for the calculation of Rd + dis.

Note: Specification of the PC for Rd provides addressing relative to the address of the first byte after the memory instruction.

Absolute Address Mode:

Notation: LDxx.A, STxx.A -- xx: any data type

The displacement dis is used as an address into memory address space. Rd must denote the SR to differentiate this mode from the displacement address mode; the content of the SR is not used.
In the case of all data types except byte, address bit zero is supplied as zero.

Note: The displacement provides absolute addressing at the beginning and the end (ROM part) of the memory.

I/O Displacement Address Mode:

Notation: LDxx.**IOD**, STxx.**IOD** -- xx: word or double-word data type

The sum of the contents of the destination register Rd plus a signed displacement dis is used as an address into I/O address space.
Rd may denote any register except the SR; Rd not denoting the SR differentiates this mode from the I/O absolute address mode.
Bits one and zero of dis are treated as zero for the calculation of Rd + dis.
Execution of a memory instruction with I/O displacement address mode does not disrupt any page mode sequence and cannot cause any Data Page Fault; an erroneous Page Fault signal is ignored.

Note: The I/O displacement address mode provides dynamic addressing of peripheral devices.
When at a load instruction only a byte or halfword is placed on the (lower part) of the data bus, the higher-order bits are undefined and must be masked out before the loaded operand is used further.

I/O Absolute Address Mode:

Notation: LDxx.**IOA**, STxx.**IOA** -- xx: word or double-word data type

The displacement dis is used as an address into I/O address space.
Rd must denote the SR to differentiate this mode from the I/O displacement address mode; the content of the SR is not used.
Address bits one and zero are supplied as zero.
Execution of a memory instruction with I/O address mode does not disrupt any page mode sequence and cannot cause any Data Page Fault; an erroneous Page Fault signal is ignored.

Note: The I/O absolute address mode provides code efficient absolute addressing of peripheral devices and allows simple decoding of I/O addresses.
When at load instructions only a byte or a halfword is placed on the (lower part) of the data bus, the higher-order bits are undefined and must be masked out before the loaded operand is used further.

Application Note: For all I/O address modes, I/O addresses with bit 31 = 1 are reserved for hardware extensions and system modules.

Next Address Mode:

Notation: LDxx.N, STxx.N -- xx: any data type

The content of the destination register Rd is used as an address into memory address space, then Rd is incremented by the signed displacement dis regardless of any exception occuring. At a double-word data type, Rd is incremented at the first memory cycle.

Rd must not denote the PC or the SR.

In the case of all data types except byte, bit zero of dis is treated as zero for the calculation of Rd + dis.

Stack Address Mode:

Notation: LDW.S, STW.S -- only word data type

The content of the destination register Rd is used as stack address, then Rd is incremented by dis regardless of any exception occurred.

A stack address addresses memory address space if it is lower than the stack pointer SP; otherwise bits 7..2 of it (higher bits are ignored) address a register in the register part of the stack absolutely (not relative to the frame pointer FP).

Bits one and zero of dis are treated as zero for the calculation of Rd + dis.

Rd must not denote the PC or the SR.

Note: The stack address mode must be used to address an operand in the stack regardless of its present location either in the memory part or in the register part of the stack. Rd may be set by the Set Stack Address instruction.

Address Mode Encoding:

The encoding of the displacement and absolute address mode types of memory instructions is shown in the following table:

			LDxx.D/A/IOD/IOA		STxx.D/A/IOD/IOA	
D-code	dis(1)	dis(0)	Rd does not denote SR	Rd denotes SR	Rd does not denote SR	Rd denotes SR
0	X	X	LDBS.D	LDBS.A	STBS.D	STBS.A
1	X	X	LDBU.D	LDBU.A	STBU.D	STBU.A
2	X	0	LDHU.D	LDHU.A	STHU.D	STHU.A
2	X	1	LDHS.D	LDHS.A	STHS.D	STHS.A
3	0	0	LDW.D	LDW.A	STW.D	STW.A
3	0	1	LDD.D	LDD.A	STD.D	STD.A
3	1	0	LDW.IOD	LDW.IOA	STW.IOD	STW.IOA
3	1	1	LDD.IOD	LDD.IOA	STD.IOD	STD.IOA

The encoding of the Exchange instruction XCHW and of the next and stack address mode types of memory instructions is shown in the following table:

With the instructions below, Rd must not denote the PC or the SR

D-code	dis(1)	dis(0)	LDxx.N/S	STxx.N/S, XCHW
0	X	X	LDBS.N	STBS.N
1	X	X	LDBU.N	STBU.N
2	X	0	LDHU.N	STHU.N
2	X	1	LDHS.N	STHS.N
3	0	0	LDW.N	STW.N
3	0	1	LDD.N	STD.N
3	1	0	Reserved	XCHW
3	1	1	LDW.S	STW.S

3.1.2 Load Instructions

The Load instructions transfer data from the addressed memory location into a register Rs or a register pair Rs//Rsf.
In the case of data type word, one word, in the case of data type double-word, two succeeding words are read from memory and transferred unchanged into Rs or Rs//Rsf respectively.
In the case of byte and halfword data types, one word is read from memory, the byte or halfword addressed by bits one and zero or bit one of the memory address respectively is extracted, right adjusted, expanded to 32 bits and placed in Rs. Unsigned bytes and halfwords are expanded by leading zeros, signed bytes and halfwords are expanded by leading sign bits.
Execution of a Load instruction enters the register address of Rs, memory address bits one and zero and a code for the data type into the load pipeline, places the memory address onto the address bus and starts a memory cycle. A double-word Load instruction enters the register address of Rsf and the same control information into the load pipeline as a second entry, places the memory address incremented by four onto the address bus and starts a second memory cycle.
After execution of a Load instruction, the next instructions are executed without waiting for the data to be loaded. A wait is enforced only if an instruction uses a register whose register address is still in the load pipeline. The data read from memory is placed in the register whose register address is at the head of the load pipeline, its pipeline entry is then deleted.
Rs must not denote the PC or the SR; these registers cannot be loaded from memory.
Rs and Rsf must also not denote the same register as Rd (or Ld), otherwise the handling of a Data Page Fault could incorrectly use Rd (or Ld) already overwritten by the fault-causing memory cycle.

3.1.2 Load Instructions (continued)

Format	Notation	Operation	Data Type xx
LR	LDxx.R, Ld, Rs;	Rs := Ld^; [Rsf := (Ld + 4)^;] -- register address mode	W,D
LR	LDxx.P, Ld, Rs;	Rs := Ld^; Ld := Ld + size; [Rsf := (old Ld + 4)^;] -- postincrement address mode	W,D
RRdis	LDxx.D, Rd, Rs, dis;	Rs := (Rd + dis)^; [Rsf := (Rd + dis + 4)^;] -- displacement address mode	BU,BS,HU,HS,W,D
RRdis	LDxx.A, 0, Rs, dis;	Rs := dis^; [Rsf := (dis + 4)^;] -- absolute address mode	BU,BS,HU,HS,W,D
RRdis	LDxx.IOD, Rd, Rs, dis;	Rs := (Rd + dis)^; [Rsf := (Rd + dis + 4)^;] -- I/O displacement address mode	W,D
RRdis	LDxx.IOA, 0, Rs, dis;	Rs := dis^; [Rsf := (dis + 4)^;] -- I/O absolute address mode	W,D
RRdis	LDxx.N, Rd, Rs, dis;	Rs := Rd^; Rd := Rd + dis; [Rsf := (old Rd + 4)^;] -- next address mode	BU,BS,HU,HS,W,D
RRdis	LDxx.S, Rd, Rs, dis;	Rs := Rd^; Rd := Rd + dis; -- stack address mode	W

Size is 1, 2, 4 or 8 according to a data size of byte, halfword, word or double-word respectively.
The expressions in brackets are only executed at double-word data types.

Data Type xx is with:

BU: byte unsigned;	HU: halfword unsigned;	W: word;
BS: byte signed;	HS: halfword signed;	D: double-word;

3.1.3 Store Instructions

The Store instructions transfer data from the register Rs or the register pair Rs//Rsf to the addressed memory location.
In the case of data type word, one word, in the case of data type double-word, two succeeding words are placed unchanged from Rs or Rs//Rsf respectively onto the data bus to be stored in the memory.
In the case of byte and halfword data types, the low-order byte or halfword is placed onto the data bus at the byte or halfword position addressed by bits one and zero or bit one of the memory address respectively; it is implied to be merged (via byte write enable) with the other data in the same memory word.
In the case of unsigned/signed byte and unsigned/signed halfword data types, any content of Rs exceeding the value range of the specified data type causes a trap to Range Error. The byte or halfword is stored regardless of a Range Error.
If Rs denotes the SR, zero is stored regardless of the content of SR (or of SR//G2 at double-word).
Execution of a Store instruction enters the contents of Rs, memory address bits one and zero and a code for the data type into the store pipeline, places the memory address onto the address bus and starts a memory cycle. A double-word Store instruction enters the contents of Rsf and the same control information into the store pipeline as a second entry, places the memory address incremented by four onto the address bus and starts a second memory cycle.
After execution of a Store instruction, the next instructions are executed without waiting for the store memory cycle to finish. The data at the head of the store pipeline is put on the data bus on demand from the on-chip memory control logic and its pipeline entry is deleted.
When Rsf denotes the same register as Rd (or Ld) at double-word instructions with next address or postincrement address mode, the incremented content of Rsf is stored in the second memory cycle; in all other cases, the unchanged content of Rs or Rsf is stored.

3.1.3 Store Instructions (continued)

Format	Notation	Operation	Data Type xx
LR	STxx.R, Ld, Rs;	Ld^ := Rs; [(Ld + 4)^ := Rsf;] -- register address mode	W,D
LR	STxx.P, Ld, Rs;	Ld^ := Rs; Ld := Ld + size; [(old Ld + 4)^ := Rsf;] -- postincrement address mode	W,D
RRdis	STxx.D, Rd, Rs, dis;	(Rd + dis)^ := Rs; [(Rd + dis + 4)^ := Rsf;] -- displacement address mode	BU,BS,HU,HS,W,D
RRdis	STxx.A, 0, Rs, dis;	dis^ := Rs; [(dis + 4)^ := Rsf;] -- absolute address mode	BU,BS,HU,HS,W,D
RRdis	STxx.IOD, Rd, Rs, dis;	(Rd + dis)^ := Rs; [(Rd + dis + 4)^ := Rsf;] -- I/O displacement address mode	W,D
RRdis	STxx.IOA, 0, Rs, dis;	dis^ := Rs; [(dis + 4)^ := Rsf;] -- I/O absolute address mode	W,D
RRdis	Stxx.N, Rd, Rs, dis;	Rd^ := Rs; Rd := Rd + dis; [(old Rd + 4)^ := Rsf;] -- next address mode	BU,BS,HU,HS,W,D
RRdis	STxx.S, Rd, Rs, dis;	Rd^ := Rs; Rd := Rd + dis; -- stack address mode	W

Size is 1, 2, 4 or 8 according to a data size of byte, halfword, word or double-word respectively.
The expressions in brackets are only executed at double-word data types.
In the case of byte and halfword data types, a trap to Range Error occurs when the value of the operand to be stored exceeds the value range of the specified data type; the byte or halfword is stored regardless of a Range Error.

Data Type xx is with:

BU: byte unsigned;	HU: halfword unsigned;	W: word;
BS: byte signed;	HS: halfword signed;	D: double-word;

3.1.4 Exchange Instruction

The Exchange instruction exchanges the data word in the addressed memory location and Rs in one indivisible read-write cycle.

The Exchange instruction is executed analogous to a combination of a Load and Store instruction. The register address of Rs and a code for the data type (word) is entered into the load pipeline, the content of Rs and a code for the data type (word) is entered into the store pipeline, the memory address is placed onto the address bus and a memory cycle is started.

After execution of an Exchange instruction, the next instructions are executed without waiting for the data word to be loaded or the memory cycle to finish. A wait is enforced only if an instruction uses a register whose register address is still in the load pipeline. Data is transferred according to the execution of a Load and a Store instruction in sequence.

The content of Rd is used as an address into memory address space, then Rd is incremented by dis. Bit one of dis is treated as zero for the calculation of Rd + dis.

The Exchange instruction shares its basic OP-code with the instruction STxx.N/S, it is differentiated by the D-code = 3, dis(1) = 1 and dis(0) = 0.

At a Data Page Fault, the result of the exchange is undefined. Rs may only denote the same register as Rd without changing the effect of the exchange when a Data Page Fault cannot occur. Rd or Rs must not denote the PC or the SR.

Format	Notation	Operation	
RRdis	XCHW, Rd, Rs, dis;	Rs := Rd^; Rd^ := old Rs; Rd := Rd + dis;	-- exchange memory word

Note: The Exchange instruction is needed for programming semaphores. It provides the capability to set a value in a memory location and inspect its previous content in one indivisible memory cycle.

3.2 Move Word Instructions

The source operand or the immediate operand is copied to the destination register and the condition flags are set or cleared accordingly.

Format	Notation	Operation
RR	MOV, Rd, Rs;	Rd := Rs; Z := Rd = 0; N := Rd(31); V := Rd(30..19) = all ones; -- for floating-point NaN
Rimm	MOVI, Rd, imm;	Rd := imm; Z := Rd = 0; N := Rd(31); V := 0;

3.3 Move Double-Word Instruction

The double-word source operand is copied to the double-word destination register pair and the condition flags are set or cleared accordingly. The high-order word in Rs is copied first.

When the SR is denoted as a source operand, the source operand is supplied as zero regardless of the content of SR//G2. When the PC is denoted as destination, the Return instruction RET is executed instead of the Move Double-Word instruction.

Format	Notation	Operation
RR	MOVD, Rd, Rs;	if Rd does not denote PC and Rs does not denote SR then Rd := Rs; Rdf := Rsf; Z := Rd//Rdf = 0; N := Rd(31); V := Rd(30..19) = all ones; -- for floating-point NaN
RR	MOVD, Rd, 0;	if Rd does not denote PC and Rs denotes SR then Rd := 0; Rdf := 0; Z := 1; N := 0; V := 0;
RR	RET, PC, Rs;	if Rd denotes PC then execute the RET instruction;

3.4 Logical Instructions

The result of a bitwise logical AND, AND not (ANDN), OR or exclusive OR (XOR) of the source or immediate operand and the destination operand is placed in the destination register and the Z flag is set or cleared accordingly. At ANDN, the source operand is used inverted (itself remaining unchanged).
All operands and the result are interpreted as bitstrings of 32 bits each.

Format	Notation	Operation	
RR	AND, Rd, Rs;	Rd := Rd and Rs; Z := Rd = 0;	-- logical AND
RR	ANDN, Rd, Rs;	Rd := Rd and not Rs; Z := Rd = 0;	-- logical AND with source used inverted
RR	OR, Rd, Rs;	Rd := Rd or Rs; Z := Rd = 0;	-- logical OR
RR	XOR, Rd, Rs;	Rd := Rd xor Rs; Z := Rd = 0;	-- logical exclusive OR
Rimm	ANDNI, Rd, imm;	Rd := Rd and not imm; Z := Rd = 0;	-- logical AND with imm used inverted
Rimm	ORI, Rd, imm;	Rd := Rd or imm; Z := Rd = 0;	-- logical OR
Rimm	XORI, Rd, imm;	Rd := Rd xor imm; Z := Rd = 0;	-- logical exclusive OR

Note: ANDN and ANDNI are the instructions complementary to OR and ORI: Where OR and ORI set bits, ANDN and ANDNI clear bits at bit positions with a "one" bit in the source or immediate operand, thus obviating the need for an inverted mask in most cases.

3.5 Invert Instruction

The source operand is placed bitwise inverted in the destination register and the Z flag is set or cleared accordingly.
The source operand and the result are interpreted as bitstrings of 32 bits each.

Format	Notation	Operation
RR	NOT, Rd, Rs;	Rd := not Rs; Z := Rd = 0;

3.6 Mask Instruction

The result of a bitwise logical AND of the source operand and the immediate operand is placed in the destination register and the Z flag is set or cleared accordingly.
All operands and the result are interpreted as bitstrings of 32 bits each.

Format	Notation	Operation
RRconst	MASK, Rd, Rs, const;	Rd := Rs and const; Z := Rd = 0;

Note: The Mask instruction may be used to move a source operand with bits partly masked out by an immediate operand used as mask. The immediate operand const is constrained in its range by bits 31 and 30 being either both zero or both one (see format RRconst). If these bits are required to be different, the instruction pair MOVI, AND may be used instead of MASK.

3.7 Add Instructions

The source operand, the source operand + C or the immediate operand is added to the destination operand, the result is placed in the destination register and the condition flags are set or cleared accordingly.
At ADD, ADDC and ADDI, both operands and the result are interpreted as either all signed or all unsigned integers. At ADDS and ADDSI, both operands and the result are signed integers and a trap to Range Error occurs at overflow.

Format	Notation	Operation	
RR	ADD, Rd, Rs;	Rd := Rd + Rs;	-- signed or unsigned Add
		Z := Rd = 0;	
		N := Rd(31) xor overflow;	-- true sign
		V := overflow;	
		C := carry;	
RR	ADDS, Rd, Rs;	Rd := Rd + Rs;	-- signed Add with trap
		Z := Rd = 0;	
		N := Rd(31) xor overflow;	-- true sign
		V := overflow;	
		if overflow then trap -> Range Error;	
RR	ADDC, Rd, Rs;	Rd := Rd + Rs + C;	-- signed or unsigned Add
		Z := Z and (Rd = 0);	with carry
		N := Rd(31) xor overflow;	-- true sign
		V := overflow;	
		C := carry;	

When the SR is denoted as a source operand at ADD, ADDS and ADDC, C is added instead of the SR. The notation is then:

Format	Notation	Operation	
RR	ADD, Rd, C;	Rd := Rd + C;	-- signed or unsigned Add C
RR	ADDS, Rd, C;	Rd := Rd + C;	-- signed Add C with trap
RR	ADDC, Rd, C;	Rd := Rd + C;	

The flags and the trap condition are treated as defined by ADD, ADDS or ADDC.

3.7 Add Instructions (continued)

```
Format  Notation            Operation

Rimm    ADDI, Rd, imm;      Rd := Rd + imm;               -- signed or unsigned Add
                            Z := Rd = 0;
                            N := Rd(31) xor overflow;     -- true sign
                            V := overflow;
                            C := carry;

Rimm    ADDSI, Rd, imm;     Rd := Rd + imm;               -- signed Add with trap
                            Z := Rd = 0;
                            N := Rd(31) xor overflow;     -- true sign
                            V := overflow;
                            if overflow then trap -> Range Error;
```

The following instructions are special cases of ADDI and ADDSI differentiated by n = 0 (see table of immediate values):

```
Format  Notation            Operation

Rimm    ADDI, Rd, CZ;       Rd := Rd + (C and (Z = 0 or Rd(0)));     -- round to
                                                                        even

Rimm    ADDSI, Rd, CZ;      Rd := Rd + (C and (Z = 0 or Rd(0)));     -- round to
                                                                        even
```

The flags and the trap condition are treated as defined by ADDI or ADDSI.

Note: At ADDC, Z is cleared if Rd /= 0, otherwise left unchanged; thus, Z is evaluated correctly for multi-precision operands.
The effect of a Subtract immediate instruction can be obtained by using the negated 32-bit value of the immediate operand to be subtracted (except zero). At unsigned, C = 0 indicates then a borrow (the unsigned number range is exceeded below zero).
At "round to even", C is only added to the destination operand if Z = 0 or Rd(0) is one. The Z flag is assumed to be set or cleared by a preceding Shift Left instruction.
"Round to even" provides a better averaging of rounding errors than "add carry".
"Round to even" is equivalent to "round to nearest" at the Floating-Point instructions and may be used to implement them.

3.8 Sum Instructions

The sum of the source operand and the immediate operand is placed in the destination register and the condition flags are set or cleared accordingly. At SUM, both operands and the result are interpreted as either all signed or all unsigned integers. At SUMS, both operands and the result are signed integers and a trap to Range Error occurs at overflow.

Format	Notation	Operation
RRconst	SUM, Rd, Rs, const;	Rd := Rs + const; -- signed or unsigned Sum Z := Rd = 0; N := Rd(31) xor overflow; -- true sign V := overflow; C := carry;
RRconst	SUMS, Rd, Rs, const;	Rd := Rs + const; -- signed Sum with trap Z := Rd = 0; N := Rd(31) xor overflow; -- true sign V := overflow; if overflow then trap -> Range Error;

When the SR is denoted as a source operand at SUM and SUMS, C is added instead of the SR. The notation is then:

Format	Notation	Operation
RRconst	SUM, Rd, C, const;	Rd := C + const; -- signed or unsigned Sum C
RRconst	SUMS, Rd, C, const;	Rd := C + const; -- signed Sum C

The flags are treated as defined by SUM or SUMS. A trap cannot occur.

Note: The effect of a Subtract immediate instruction can be obtained by using the negated 32-bit value of the immediate operand to be subtracted (except zero). At unsigned, C = 0 indicates then a borrow (the unsigned number range is exceeded below zero).
The immediate operand is constrained to the range of const. The instruction pair MOV, ADDI or MOV, ADDSI may be used where the full integer range is required.

3.9 Subtract Instructions

The source operand or the source operand + C is subtracted from the destination operand, the result is placed in the destination register and the condition flags are set or cleared accordingly.
At SUB and SUBC, both operands and the result are interpreted as either all signed or all unsigned integers. At SUBS, both operands and the result are signed integers and a trap to Range Error occurs at overflow.

```
Format Notation         Operation

RR     SUB, Rd, Rs;     Rd := Rd - Rs;             -- signed or unsigned Subtract
                        Z := Rd = 0;
                        N := Rd(31) xor overflow;  -- true sign
                        V := overflow;
                        C := borrow;

RR     SUBS, Rd, Rs;    Rd := Rd - Rs;             -- signed Subtract with trap
                        Z := Rd = 0;
                        N := Rd(31) xor overflow;  -- true sign
                        V := overflow;
                        if overflow then trap -> Range Error;

RR     SUBC, Rd, Rs;    Rd := Rd - (Rs + C);       -- signed or unsigned Subtract
                        Z := Z and (Rd = 0);          with borrow
                        N := Rd(31) xor overflow;  -- true sign
                        V := overflow;
                        C := borrow;
```

When the SR is denoted as a source operand at SUB, SUBS and SUBC, C is subtracted instead of the SR. The notation is then:

```
Format Notation         Operation

RR     SUB, Rd, C;      Rd := Rd - C;          -- signed or unsigned Subtract C

RR     SUBS, Rd, C;     Rd := Rd - C;          -- signed Subtract C with trap

RR     SUBC, Rd, C;     Rd := Rd - C;
```

The flags and the trap condition are treated as defined by SUB, SUBS or SUBC.

Note: At SUBC, Z is cleared if Rd /= 0, otherwise left unchanged; thus, Z is evaluated correctly for multi-precision operands.

3.10 Negate Instructions

The source operand is subtracted from zero, the result is placed in the destination register and the condition flags are set or cleared accordingly.
At NEG and NEGC, the source operand and the result are interpreted as either both signed or both unsigned integers. At NEGS, the source operand and the result are signed integers and a trap to Range Error occurs at overflow.

```
Format  Notation          Operation

RR      NEG, Rd, Rs;      Rd := - Rs;                 -- signed or unsigned Negate
                          Z := Rd = 0;
                          N := Rd(31) xor overflow;   -- true sign
                          V := overflow;
                          C := borrow;

RR      NEGS, Rd, Rs;     Rd := - Rs;                 -- signed Negate with trap
                          Z := Rd = 0;
                          N := Rd(31) xor overflow;   -- true sign
                          V := overflow;
                          if overflow then trap -> Range Error;
```

When the SR is denoted as a source operand at NEG and NEGS, C is negated instead of the SR. The notation is then:

```
Format  Notation          Operation

RR      NEG, Rd, C;       Rd := - C;             -- signed or unsigned Negate C
                                                    if C is set, then Rd := -1;
                                                    else Rd := 0;

RR      NEGS, Rd, C;      Rd := - C;             -- signed Negate C
                                                    if C is set, then Rd := -1;
                                                    else Rd := 0;
```

The flags are treated as defined by NEG or NEGS. A trap cannot occur.

3.11 Multiply Word Instruction

The source operand and the destination operand are multiplied, the low-order word of the double-word product is placed in the destination register (the high-order product word is discarded) and the condition flags are set or cleared according to the double-word product (this is the same as according to the low-order word when no Range Error occurred).
After execution, a trap to Range Error occurs if the high-order word of the product is not the sign extension of the low-order word (that is, if any bit of the high-order word is different from the sign bit of the low-order word).
Both operands are signed integers, the product is a signed double-word integer and the low-order word of it is also a signed integer when no Range Error occurred.
The result is undefined if Rs denotes the same register as Rd or if the PC or the SR is denoted.

Format	Notation	Operation
RR	MUL, Rd, Rs;	Rd := low order word of doubleword product Rd * Rs; Z := doubleword product = 0; -- same as Rd = 0 if no trap; N := sign of doubleword product; -- same as Rd(31) if no trap; V := any bit of high order product word /= Rd(31); C := high order product word /= 0; if V = 1 then trap -> Range Error;

3.12 Multiply Double-Word Instructions

The source operand and the destination operand are multiplied, the double-word product is placed in the destination register pair (the destination register expanded by the register following it) and the condition flags are set or cleared according to the double-word product.
At MULS, both operands are signed integers and the product is a signed double-word integer. At MULU, both operands are unsigned integers and the product is an unsigned double-word integer.
The result is undefined if Rs denotes the same register as Rd or if the PC or the SR is denoted. However, Rs may denote the same register as Rdf; Rdf is overwritten by the low-order product word after the actual multiplication is executed.

Format	Notation	Operation
RR	MULS, Rd, Rs;	Rd//Rdf := signed doubleword product of Rd * Rs; Z := Rd//Rdf = 0; -- doubleword product is zero N := Rd(31); -- doubleword product is negative V := any bit of Rd /= Rdf(31); C := Rd /= 0;
RR	MULU, Rd, Rs;	Rd//Rdf := unsigned doubleword product of Rd * Rs; Z := Rd//Rdf = 0; -- doubleword product is zero N := Rd(31); V := any bit of Rd /= Rdf(31); C := Rd /= 0;

3.13 Divide Instructions

The double-word destination operand (dividend) is divided by the single-word source operand (divisor), the quotient is placed in the low-order destination register (Rdf), the remainder is placed in the high-order destination register (Rd) and the condition flags are set or cleared according to the quotient.
A trap to Range Error occurs if the divisor is zero or the value of the quotient exceeds the integer value range (quotient overflow). The result (in Rd//Rdf) is then undefined. At DIVS, a trap to Range Error also occurs and the result is undefined if the dividend is negative.
At DIVS, the dividend is a non-negative signed double-word integer, the divisor, the quotient and the remainder are signed integers; a non-zero remainder has the sign of the dividend.
At DIVU, the dividend is an unsigned double-word integer, the divisor, the quotient and the remainder are unsigned integers.
The result is undefined if Rs denotes the same register as Rd or Rdf or if the PC or the SR is denoted.

```
Format  Notation          Operation

RR      DIVS, Rd, Rs;     if Rs = 0 or quotient overflow
                              or Rd(31) = 1 then      -- dividend is negative
                              Rd//Rdf := undefined;
                              Z := undefined;
                              N := undefined;
                              V := 1;
                              trap -> Range Error;
                          else remainder Rd, quotient Rdf := (Rd//Rdf) / Rs;
                              Z := Rdf = 0;           -- quotient is zero
                              N := Rdf(31);           -- quotient is negative
                              V = 0;

RR      DIVU, Rd, Rs;     if Rs = 0 or quotient overflow then
                              Rd//Rdf := undefined;
                              Z := undefined;
                              N := undefined;
                              V := 1;
                              trap -> Range Error;
                          else remainder Rd, quotient Rdf := (Rd//Rdf) / Rs;
                              Z := Rdf = 0;           -- quotient is zero
                              N := Rdf(31);
                              V := 0;
```

3.14 Shift Left Instructions

The destination operand is shifted left by a number of bit positions specified
at SALI, SALDI, SHLI SHLDI by n = 0..31 as a shift by 0..31;
at SHL, SHLD by bits 4..0 of the source operand as a shift by 0..31.
The higher-order bits of the source operand are ignored.

The destination operand is interpreted
at SALI as a signed integer; a trap to Range Error can occur;
at SALDI as a signed double-word integer; a trap to Range Error can occur;
at SHL and SHLI as a bitstring of 32 bits or as a signed or unsigned integer;
at SHLD and SHLDI as a double-word bitstring of 64 bits or as a signed or unsigned double-word integer.

All Shift Left instructions insert zeros in the vacated bit positions at the right.
The double-word Shift Left instructions execute in two cycles. The low-order operand in Ldf is shifted first. At SHLD, the result is undefined if Ls denotes the same register as Ld or Ldf.

Format	Notation	Operation	insert
Rn	SALI, Rd, n;	Rd := Rd << by n; if any bit /= new Rd(31) is shifted out then trap -> Range Error;	-- 0..31 zeros
Ln	SALDI, Ld, n;	Ld//Ldf := Ld//Ldf << by n; if any bit /= new Ld(31) is shifted out then trap -> Range Error;	-- 0..31 zeros
Rn	SHLI, Rd, n;	Rd := Rd << by n;	-- 0..31 zeros
Ln	SHLDI, Ld, n;	Ld//Ldf := Ld//Ldf << by n;	-- 0..31 zeros
LL	SHL, Ld, Ls;	Ld := Ld << by Ls(4..0);	-- 0..31 zeros
LL	SHLD, Ld, Ls;	Ld//Ldf := Ld//Ldf << by Ls(4..0);	-- 0..31 zeros

The condition flags are set or cleared by all Shift Left instructions as follows:

```
Z := Ld = 0 or Rd = 0 at single-word;
Z := Ld//Ldf = 0 at double-word;
N := old Ld(31) or Rd(31) before shift; -- same as new Ld(31) or Rd(31) if V = 0
V := any bit /= new Ld(31) or Rd(31) is shifted out;    -- significant bits lost
C := any bit /= 0 is shifted out;                         -- leading ones lost
```

At SALI and SALDI, a trap to Range Error occurs if significant bits are shifted out (V = 1).

Note: The symbol << signifies "shifted left".

3.15 Shift Right Instructions

The destination operand is shifted right by a number of bit positions specified

at SARI, SARDI, SHRI, SHRDI by n = 0..31 as a shift by 0..31.

at SAR, SARD, SHR, SHRD by bits 4..0 of the source operand as a shift by 0..31.

The higher-order bits of the source operand are ignored.

The destination operand is interpreted

at SAR and SARI as a signed integer;

at SARD and SARDI as a signed double-word integer;

at SHR and SHRI as a bitstring of 32 bits or as an unsigned integer;

at SHRD and SHRDI as a double-word bitstring of 64 bits or as an unsigned double-word integer.

All Shift Right instructions which interpret the destination operand as signed insert sign bits, all others insert zeros in the vacated bit positions at the left.

The double-word Shift Right instructions execute in two cycles. The high-order operand in Ld is shifted first. At SARD and SHRD, the result is undefined if Ls denotes the same register as Ld or Ldf.

Format	Notation	Operation	insert
Rn	SARI, Rd, n;	Rd := Rd >> by n;	-- 0..31 sign bits
Ln	SARDI, Ld, n;	Ld//Ldf := Ld//Ldf >> by n;	-- 0..31 sign bits
LL	SAR, Ld, Ls;	Ld := Ld >> by Ls(4..0);	-- 0..31 sign bits
LL	SARD, Ld, Ls;	Ld//Ldf := Ld//Ldf >> by Ls(4..0);	-- 0..31 sign bits
Rn	SHRI, Rd, n;	Rd := Rd >> by n;	-- 0..31 zeros
Ln	SHRDI, Ld, n;	Ld//Ldf := Ld//Ldf >> by n;	-- 0..31 zeros
LL	SHR, Ld, Ls;	Ld := Ld >> by Ls(4..0);	-- 0..31 zeros
LL	SHRD, Ld, Ls;	Ld//Ldf := Ld//Ldf >> by Ls(4..0);	-- 0..31 zeros

The condition flags are set or cleared by all Shift Right instructions as follows:

Z := Ld = 0 or Rd = 0 at single-word;
Z := Ld//Ldf = 0 at double-word;
N := Ld(31) or Rd(31);
C := last bit shifted out is "one";

Note: The symbol >> signifies "shifted right".

3.16 Rotate Left Instruction

The destination operand is shifted left by a number of bit positions and the bits shifted out are inserted in the vacated bit positions; thus, the destination operand is rotated. The condition flags are set or cleared accordingly. Bits 4..0 of the source operand specify a rotation by 0..31 bit positions; bits 31..5 of the source operand are ignored.
The destination operand is interpreted as a bitstring of 32 bits.

Format	Notation	Operation
LL	ROL, Ld, Ls;	Ld := Ld rotated left by Ls(4..0); Z := Ld = 0; N := old Ld(31) before shift; -- same as new Ld(31) if V = 0 V := any bit /= new Ld(31) is shifted out; C := any bit /= 0 is shifted out;

Note: The condition flags are set or cleared by the same rules applying to the Shift Left instructions.

3.17 Index Move Instructions

The source operand is placed shifted left by 0, 1, 2 or 3 bit positions in the destination register, corresponding to a multiplication by 1, 2, 4 or 8. If the source operand is higher than the immediate operand lim (upper bound), a trap to Range Error occurs (after execution). At XM1Z, XM2Z, XM4Z and XM8Z, a trap to Range Error occurs also if the source operand is zero.
All condition flags remain unchanged. All operands and the result are interpreted as unsigned integers.
The SR must not be denoted as a source or a destination, nor the PC as a destination operand; these notations are reserved for future expansion. When the PC is denoted as a source operand, a trap to Range Error occurs if PC >= lim.

X-code	Format	Notation	Operation	
0	RRlim	XM1, Rd, Rs, lim;	Rd := Rs * 1;	if Rs > lim then trap -> Range Error;
1	RRlim	XM2, Rd, Rs, lim;	Rd := Rs * 2;	if Rs > lim then trap -> Range Error;
2	RRlim	XM4, Rd, Rs, lim;	Rd := Rs * 4;	if Rs > lim then trap -> Range Error;
3	RRlim	XM8, Rd, Rs, lim;	Rd := Rs * 8;	if Rs > lim then trap -> Range Error;
4	RRlim	XM1Z, Rd, Rs, lim;	Rd := Rs * 1;	if Rs > lim or Rs = 0 then trap -> Range Error;
5	RRlim	XM2Z, Rd, Rs, lim;	Rd := Rs * 2;	if Rs > lim or Rs = 0 then trap -> Range Error;
6	RRlim	XM4Z, Rd, Rs, lim;	Rd := Rs * 4;	if Rs > lim or Rs = 0 then trap -> Range Error;
7	RRlim	XM8Z, Rd, Rs, lim;	Rd := Rs * 8;	if Rs > lim or Rs = 0 then trap -> Range Error;

Note: The Index Move instructions move an index value scaled and check the unscaled value for an upper bound, optionally also excluding zero. If the lower bound is not zero or one, it may be mapped to zero by subtracting it from the index value before applying an Index Move instruction.

3.18 Check Instructions

The destination operand is checked and a trap to Range Error occurs

at CHK if the destination operand is higher than the source operand,

at CHKZ if the destination operand is zero.

All registers and all condition flags remain unchanged. All operands are interpreted as unsigned integers.

CHKZ shares its basic OP-code with CHK, it is differentiated by denoting the SR as source operand.

Format	Notation	Operation
RR	CHK, Rd, Rs;	if Rs does not denote SR and Rd > Rs then trap -> Range Error;
RR	CHKZ, Rd, 0;	if Rs denotes SR and Rd = 0 then trap -> Range Error;

When Rs denotes the PC, CHK traps if Rd >= PC. Thus, CHK, PC, PC always traps. Since CHK, PC, PC is encoded as 16 zeros, an erroneous jump into a string of zeros causes a trap to Range Error, thus trapping some address errors.

Note: CHK checks the upper bound of an unsigned value range, implying a lower bound of zero. If the lower bound is not zero, it can be mapped to zero by subtracting it from the value to be checked and then checking against a corrected upper bound (lower bound also subtracted). When the upper bound is a constant not exceeding the range of lim, the Index instructions may be used for bounds checks.

CHKZ may be used to trap on uninitialized pointers with the value zero.

3.19 No Operation Instruction

The instruction CHK, L0, L0 cannot cause any trap. Since CHK leaves all registers and condition flags unchanged, it can be used as a No Operation instruction with the notation:

Format	Notation	Operation
RR	NOP;	no operation;

Note: The NOP instruction may be used as a fill instruction.

3.20 Compare Instructions

Two operands are compared by subtracting the source operand or the immediate operand from the destination operand. The condition flags are set or cleared according to the result; the result itself is not retained.
All operands and the result are interpreted as either all signed or all unsigned integers.

Format	Notation	Operation	
RR	CMP, Rd, Rs;	result := Rd - Rs;	
		Z := Rd = Rs;	-- result is zero
		N := Rd < Rs signed;	-- result is true negative
		V := overflow;	
		C := Rd < Rs unsigned;	-- borrow
Rimm	CMPI, Rd, imm;	result := Rd - imm;	
		Z := Rd = imm;	-- result is zero
		N := Rd < imm signed;	-- result is true negative
		V := overflow;	
		C := Rd < imm unsigned;	-- borrow

When the SR is denoted as a source operand at CMP, C is subtracted instead of SR. The notation is then:

Format	Notation	Operation	
RR	CMP, Rd, C;	result := Rd - C;	
		Z := Rd = C;	-- result is zero
		N := Rd < C signed;	-- result is true negative
		V := overflow;	
		C := Rd < C unsigned;	-- borrow

3.21 Compare Bit Instructions

The result of a bitwise logical AND of the source or immediate operand and the destination operand is used to set or clear the Z flag accordingly; the result itself is not retained.
All operands and the result are interpreted as bitstrings of 32 bits each.

Format	Notation	Operation
RR	CMPB, Rd, Rs;	Z := (Rd and Rs) = 0;
Rimm	CMPBI, Rd, imm;	Z := (Rd and imm) = 0;

The following instruction is a special case of CMPBI differentiated by n = 0 (see table of immediate values):

Format	Notation	Operation
Rimm	CMPBI, Rd, ANYBZ;	Z := Rd(31..24) = 0 or Rd(23..16) = 0 or Rd(15..8) = 0 or Rd(7..0) = 0; -- any Byte of Rd = 0

3.22 Test Leading Zeros Instruction

The number of leading zeros in the source operand is tested and placed in the destination register. A source operand equal to zero yields 32 as a result. All condition flags remain unchanged.

Format	Notation	Operation
LL	TESTLZ, Ld, Ls;	Ld := number of leading zeros in Ls;

3.23 Set Stack Address Instruction

The frame pointer FP is placed, expanded to the stack address, in the destination register. The FP itself and all condition flags remain unchanged. The expanded FP address is the address at which the content of L0 would be stored if pushed onto the memory part of the stack.
The Set Stack Address instruction shares the basic OP-code SETxx, it is differentiated by n = 0 and not denoting the SR or the PC.

n	Format	Notation	Operation
0	Rn	SETADR, Rd;	Rd := SP(31..9)//SR(31..25)//00 + carry into bit 9 -- SR(31..25) is FP -- carry into bit 9 := (SP(8) = 1 and SR(31) = 0)

Note: The Set Stack Address instruction calculates the stack address of the beginning of the current stack frame. L0..L15 of this frame can then be addressed relative to this stack address in the stack address mode with displacement values of 0..60 respectively.
Provided the stack address of a stack frame has been saved, for example in a global register, any data in this stack frame can then be addressed also from within all younger generations of stack frames by using the saved stack address. (Addressing of local variables in older generations of stack frames is required by all block oriented programming languages like Pascal, Modula-2 and Ada.)

The basic OP-code SETxx is shared as indicated:

- n = 0 while not denoting the SR or the PC differentiates the Set Stack Address instruction.
- n = 1..31 while not denoting the SR or the PC differentiates the Set Conditional instructions.
- Denoting the SR differentiates the Fetch instruction.
- Denoting the PC is reserved for future use.

3.24 Set Conditional Instructions

The destination register is set or cleared according to the states of the condition flags specified by n. The condition flags themselves remain unchanged.
The Set Conditional instructions share the basic OP-code SETxx, they are differentiated by n = 1..31 and not denoting the SR or the PC.

3.24 Set Conditional Instructions (continued)

Format is Rn

n	Notation	or Alternative	Operation
1	Reserved		
2	SET1, Rd;		Rd := 1;
3	SET0, Rd;		Rd := 0;
4	SETLE, Rd;		if N = 1 or Z = 1 then Rd := 1 else Rd := 0;
5	SETGT, Rd;		if N = 0 and Z = 0 then Rd := 1 else Rd := 0;
6	SETLT, Rd;	SETN, Rd;	if N = 1 then Rd := 1 else Rd := 0;
7	SETGE, Rd	SETNN, Rd;	if N = 0 then Rd := 1 else Rd := 0;
8	SETSE, Rd;		if C = 1 or Z = 1 then Rd := 1 else Rd := 0;
9	SETHT, Rd;		if C = 0 and Z = 0 then Rd := 1 else Rd := 0;
10	SETST, Rd;	SETC, Rd;	if C = 1 then Rd := 1 else Rd := 0;
11	SETHE, Rd;	SETNC, Rd;	if C = 0 then Rd := 1 else Rd := 0;
12	SETE, Rd;	SETZ, Rd;	if Z = 1 then Rd := 1 else Rd := 0;
13	SETNE, Rd;	SETNZ, Rd;	if Z = 0 then Rd := 1 else Rd := 0;
14	SETV, Rd;		if V = 1 then Rd := 1 else Rd := 0;
15	SETNV, Rd;		if V = 0 then Rd := 1 else Rd := 0;
16	Reserved		
17	Reserved		
18	SET1M, Rd;		Rd := -1;
19	Reserved		
20	SETLEM, Rd;		if N = 1 or Z = 1 then Rd := -1 else Rd := 0;
21	SETGTM, Rd;		if N = 0 and Z = 0 then Rd := -1 else Rd := 0;
22	SETLTM, Rd;	SETNM, Rd;	if N = 1 then Rd := -1 else Rd := 0;
23	SETGEM, Rd;	SETNNM, Rd;	if N = 0 then Rd := -1 else Rd := 0;
24	SETSEM, Rd;		if C = 1 or Z = 1 then Rd := -1 else Rd := 0;
25	SETHTM, Rd;		if C = 0 and Z = 0 then Rd := -1 else Rd := 0;
26	SETSTM, Rd;	SETCM, Rd;	if C = 1 then Rd := -1 else Rd := 0;
27	SETHEM, Rd;	SETNCM, Rd;	if C = 0 then Rd := -1 else Rd := 0;
28	SETEM, Rd;	SETZM, Rd;	if Z = 1 then Rd := -1 else Rd := 0;
29	SETNEM, Rd;	SETNZM, Rd;	if Z = 0 then Rd := -1 else Rd := 0;
30	SETVM, Rd;		if V = 1 then Rd := -1 else Rd := 0;
31	SETNVM, Rd;		if V = 0 then Rd := -1 else Rd := 0;

3.25 Branch Instructions

The Branch instruction BR, and any of the conditional Branch instructions when the branch condition is met, place the branch address PC + rel (relative to the address of the first byte after the Branch instruction) in the program counter PC and clear the cache-mode flag M; all condition flags remain unchanged. Instruction execution proceeds then at the branch address placed in the PC.
When the branch condition is not met, the M flag and the condition flags remain unchanged and instruction execution proceeds sequentially.
Besides these explicit Branch instructions, the instructions MOV, MOVI, ADD, ADDI, SUM, SUB may denote the PC as a destination register and thus be executed as an implicit branch; the M flag is cleared and the condition flags are set or cleared according to the specified instruction.
All other non-comparing instructions must not be used with the PC as destination, otherwise possible Range Errors caused by these instructions would lead to ambiguous results on backtracking.

Format is PCrel

Notation	or alternative	Operation	Comment
BLE, rel;		if N = 1 or Z = 1 then BR;	-- Less or Equal signed
BGT, rel;		if N = 0 and Z = 0 then BR;	-- Greater Than signed
BLT, rel;	BN, rel;	if N = 1 then BR;	-- Less Than signed
BGE, rel;	BNN, rel;	if N = 0 then BR;	-- Greater or Equal signed
BSE, rel;		if C = 1 or Z = 1 then BR;	-- Smaller or Equal unsigned
BHT, rel;		if C = 0 and Z = 0 then BR;	-- Higher Than unsigned
BST, rel;	BC, rel;	if C = 1 then BR;	-- Smaller Than unsigned
BHE, rel;	BNC, rel;	if C = 0 then BR;	-- Higher or Equal unsigned
BE, rel;	BZ, rel;	if Z = 1 then BR;	-- Equal
BNE, rel;	BNZ, rel;	if Z = 0 then BR;	-- Not Equal
BV, rel;		if V = 1 then BR;	-- oVerflow
BNV, rel;		if V = 0 then BR;	-- Not oVerflow
BR, rel;		PC := PC + rel; M := 0;	

Note: rel is signed to allow forward or backward branches.

3.26 Delayed Branch Instructions

The Delayed Branch instruction DBR, and any of the conditional Delayed Branch instructions when the branch condition is met, place the branch address PC + rel (relative to the address of the first byte after the Delayed Branch instruction) in the program counter PC. All condition flags and the cache mode flag M remain unchanged.
Then the instruction after the Delayed Branch instruction, called the delay instruction, is executed regardless of whether the delayed branch is taken or not taken.
When the delayed branch is not taken, the delay instruction is executed like a regular instruction. The PC and the ILC are updated accordingly and instruction execution proceeds sequentially unless the delay instruction itself causes a branch.
When the delayed branch is taken, the delay instruction is executed before execution proceeds at the branch target. The PC (containing the delayed-branch target address) is not updated by the delay instruction unless the delay instruction itself causes a branch. Any reference to the PC by the delay instruction references the delayed-branch target address. Thus, a PC-relative branch placed as delay instruction becomes a branch relative to the delayed-branch target address. Exactly as after all branches taken, the ILC is invalid.
A Software instruction, a Branch, Call or Trap instruction with a branch taken or an implicit branch used as delay instruction overrules the target address of the preceding delayed branch. The target instruction of the overruling branch is always fetched from memory and the instruction cache is flushed. A Call, Trap or Software instruction saves its instruction length in the place of the saved ILC (the standard case), but saves the delayed-branch target address in the place of the saved PC. On return, instruction execution proceeds then at the delayed-branch target address.
In the case of an error or Data Page Fault exception caused by a delay instruction succeeding a delayed branch taken, the location of the saved return PC contains the address of the first byte of the delay instruction. The saved ILC contains the length (1 or 2 halfwords) of the Delayed Branch instruction. In the case of all other exceptions following a non-branch-taking delay instruction succeeding a delayed branch taken, the location of the saved return PC contains the branch target address of the delayed branch and the saved ILC is invalid. In the case of all exceptions following a branch-taking delay instruction, the location of the saved return PC contains the branch target address of the branching delay instruction and the saved ILC is invalid (like the standard case for an exception following a branch taken).

3.26 Delayed Branch Instructions (continued)

The following restrictions apply to delay instructions:
The sum of the length of the Delayed Branch instruction and the delay instruction must not exceed three halfwords, otherwise an arbitrary bit pattern may be supplied and erroneously used for the second or third halfword of the delay instruction without any warning.
The Delayed Branch instruction and the delay instruction are locked against any exception disrupting them (except Reset).
A Fetch, Do, Extend, Return or Delayed Branch instruction must not be placed as a delay instruction. Any such misplaced Delayed Branch instruction would be executed like the corresponding non-delayed Branch instruction to inhibit a permanent exception lock-out.

Format is PCrel

Notation	or alternative	Operation	Comment
DBLE, rel;		if N = 1 or Z = 1 then DBR;	-- Less or Equal signed
DBGT, rel;		if N = 0 and Z = 0 then DBR;	-- Greater Than signed
DBLT, rel;	DBN, rel;	if N = 1 then DBR;	-- Less Than signed
DBGE, rel;	DBNN, rel;	if N = 0 then DBR;	-- Greater or Equal signed
DBSE, rel;		if C = 1 or Z = 1 then DBR;	-- Smaller or Equal unsigned
DBHT, rel;		if C = 0 and Z = 0 then DBR;	-- Higher Than unsigned
DBST, rel;	DBC, rel;	if C = 1 then DBR;	-- Smaller Than unsigned
DBHE, rel;	DBNC, rel;	if C = 0 then DBR;	-- Higher or Equal unsigned
DBE, rel;	DBZ, rel;	if Z = 1 then DBR;	-- Equal
DBNE, rel;	DBNZ, rel;	if Z = 0 then DBR;	-- Not Equal
DBV, rel;		if V = 1 then DBR;	-- oVerflow
DBNV, rel;		if V = 0 then DBR;	-- Not oVerflow
DBR, rel;		PC := PC + rel;	

Note: rel is signed to allow forward or backward branches.
Using a branching instruction as delay instruction provides no timing advantage; it might be used however for supplying a non-sequential return address to a Call, Trap or Software instruction.
Since the PC seen by the delay instruction depends on the delayed branch taken or not taken, a delay instruction after a conditional Delayed Branch instruction should not reference the PC.

3.27 Call Instructions

The Call instructions CALL and CALLV when V = 1 cause a branch to a subprogram. When at CALLV V = 0, instruction execution proceeds sequentially.
When the subprogram branch is taken, the branch address Rs + const, or const alone if Rs denotes the SR, is placed in the program counter PC. The old PC containing the return address is saved in Ld; the old supervisor-state flag S is also saved in bit zero of Ld. The old status register SR is saved in Ldf; the saved instruction-length code ILC contains the length (2 or 3) of the Call instruction.
Then the frame pointer FP is incremented by the value of the Ld-code (Ld-code = 0 is interpreted as Ld-code = 16) and the frame length FL is set to six, thus creating a new stack frame. The cache-mode flag M is cleared. All condition flags remain unchanged.
The value of the Ld-code must not exceed the value of the old FL (FL = 0 is interpreted as FL = 16), otherwise the beginning of the register part of the stack at the SP could be overwritten without any warning.
Instruction execution then proceeds at the branch address placed in the PC.
CALL and CALLV share the same OP-code; they are differentiated by bit zero of const: zero indicates CALL, one indicates CALLV. Bit zero of const is treated as zero for calculating Rs + const regardless of its value.
Rs and Ld may denote the same register.

Format	Notation	Operation
LRconst	CALLV, Ld, Rs, const; or CALLV, Ld, 0, const;	if V = 1 then execute CALL; else next instruction;

LRconst	CALL, Ld, Rs, const;
	or CALL, Ld, 0, const;
		if Rs denotes not SR then PC := Rs + const;
		else PC := const;
		Ld := old PC(31..1)//old S;
		-- Ld-code 0 selects L16
		Ldf := old SR;
		FP := FP + Ld code;
		-- Ld-code 0 is treated as 16
		FL := 6;
		M := 0;

Note: At the new stack frame, the saved PC can be addressed as L0 and the saved SR as L1; L2..L5 are free for use as required.
A Frame instruction must be executed before executing any other Call, Trap or Software instruction, otherwise the beginning of the register part of the stack at the SP could be overwritten without any warning.

3.28 Trap Instructions

The Trap instructions TRAP and any of the conditional Trap instructions when the trap condition is met, cause a branch to one out of 64 supervisor subprogram entries. When the trap condition is not met, instruction execution proceeds sequentially.
When the subprogram branch is taken, the subprogram entry address adr is placed in the program counter PC and the supervisor-state flag S is set to one. The old PC containing the return address is saved in the register addressed by FP + FL; the old S flag is also saved in bit zero of this register. The old status register SR is saved in the register addressed by FP + FL + 1 (FL = 0 is interpreted as FL = 16); the saved instruction-length code ILC contains the length (1) of the Trap instruction.
Then the frame pointer FP is incremented by the old frame length FL and FL is set to six, thus creating a new stack frame. The cache-mode flag M and the trace-mode flag T are cleared, the interrupt-lock flag L is set to one. All condition flags remain unchanged.
Instruction execution then proceeds at the entry address placed in the PC.
The trap instructions are further differentiated by the 12 code values given by the bits 9 and 8 of the OP-code and bits 1 and 0 of the adr-byte (code = OP(9..8)//adr-byte(1..0)). The code values 0..3 are not available.

3.28 Trap Instructions (continued)

Format is PCadr

Code	Notation	Operation
4	TRAPLE, adr;	if N = 1 or Z = 1 then execute TRAP; else next instruction;
5	TRAPGT, adr;	if N = 0 and Z = 0 then execute TRAP; else next instruction;
6	TRAPLT, adr;	if N = 1 then execute TRAP; else next instruction;
7	TRAPGE, adr;	if N = 0 then execute TRAP; else next instruction;
8	TRAPSE, adr;	if C = 1 or Z = 1 then execute TRAP; else next instruction;
9	TRAPHT, adr;	if C = 0 and Z = 0 then execute TRAP; else next instruction;
10	TRAPST, adr;	if C = 1 then execute TRAP; else next instruction;
11	TRAPHE, adr;	if C = 0 then execute TRAP; else next instruction;
12	TRAPE, adr;	if Z = 1 then execute TRAP; else next instruction;
13	TRAPNE, adr;	if Z = 0 then execute TRAP; else next instruction;
14	TRAPV, adr;	if V = 1 then execute TRAP; else next instruction;
15	TRAP, adr;	PC := adr; S := 1; (FP + FL)^ := old PC(31..1)//old S; (FP + FL + 1)^ := old SR; FP := FP + FL; -- FL = 0 is treated as FL = 16 FL := 6; M := 0; T := 0; L := 1;

Note: At the new stack frame, the saved PC can be addressed as L0 and the saved SR as L1; L2..L5 are free for use as required. As long as the interrupt-lock flag L is kept at one and no Frame instruction or any instruction with the potential to cause an exception is executed, registers L6..L9 can also be used by those trap routines which do not share their entry with any exception entry.

A Frame instruction must be executed before executing any other Trap, Call or Software instruction, otherwise the beginning of the register part of the stack at the SP could be overwritten without any warning.

3.29 Frame Instruction

A Frame instruction restructures the current stack frame by

- decrementing the frame pointer FP to include (optionally) passed parameters in the local register addressing range; the first parameter passed is then addressable as L0;
- resetting the frame length FL to the actual number of registers needed for the current stack frame.

It also restores the reserve number of 10 registers in the register part of the stack to allow any further Call, Trap or Software instructions and clears the cache mode flag M.
The frame pointer FP is decremented by the Ls-code and the Ld-code is placed in the frame length FL (FL = 0 is always interpreted as FL = 16). Then the difference (available number of registers) - (required number of registers + 10) is evaluated and interpreted as a signed 7-bit integer.
If the difference is not negative, all the registers required plus the reserve of 10 fit into the register part of the stack; no further action is needed and the Frame instruction is finished.
If the difference is negative, the content of the old stack pointer SP is compared with the address in the upper stack bound UB. If the value in the SP is equal or higher than the value in the UB, a temporary flag is set. Then the contents of the number of local registers equal to the negative difference evaluated are pushed onto the memory part of the stack, beginning with the content of the local register addressed absolutely by SP(7..2) being pushed onto the location addressed by the SP. After each memory cycle, the SP is incremented by four until the difference is eliminated. A trap to Frame Error occurs after completion of the push operation when the temporary flag is set.
The upper stack bound UB must be set so that a Data Page Fault cannot occur; that is, the stack space up to the boundary of UB + 32 words must be in resident memory.
All condition flags remain unchanged.

3.29 Frame Instruction (continued)

Format	Notation	Operation
LL	FRAME, Ld, Ls;	see below

```
FP := FP - Ls code;
FL := Ld code;
M := 0;
difference(6..0) := SP(8..2) + (64 - 10) - (FP + FL);
    -- FL = 0 is treated as FL = 16
    -- difference is signed, difference(6) = sign bit
    -- 64 = number of local registers
    -- 10 = number of reserve registers
if difference >= 0 then continue at next instruction;
    -- Frame is finished
temporary flag := SP >= UB;
repeat
    memory SP^ := register SP(7..2)^;
        -- local register -> memory
    SP := SP + 4;
    difference := difference + 1;
until difference = 0;
if temporary flag = 1 then Trap -> Frame Error;
```

Note: Ls also identifies the same source operand which must be denoted by the Return instruction to address the saved return PC.
Ld (L0 is interpreted as L16) also identifies the register in which the return PC is being saved by a Trap or Software instruction or by an exception; therefore only local registers with a lower register code than the interpreted Ld-code of the Frame instruction must be used after execution of a Frame instruction.
The reserve of 10 registers is to be used as follows:

- A Call, Trap or Software instruction uses six registers.
- A subsequent exception, occurring before a Frame instruction is executed, uses another two registers.
- Two registers remain in reserve.

Note that the Frame instruction can write into the memory stack at address locations up to 32 words higher than indicated by the address in the UB. This is due to the fact that the upper bound is checked before the execution of the Frame instruction.

3.30 Return Instruction

The Return instruction returns control from a subprogram entered through a Call, Trap or Software instruction or an exception to the instruction located at the return address and restores the status from the saved return status.

The source operand pair Rs//Rsf is placed in the register pair PC//SR. The program counter PC is restored first from Rs. Then all bits of the status register SR are replaced by Rsf, except the supervisor flag S, which is restored from bit zero of Rs and except the instruction length code ILC, which is cleared to zero.

If the return occurred from user to supervisor state or if the interrupt-lock flag L was changed from zero to one on return from any state to user state, a trap to Privilege Error occurs. Exception processing saves the restored contents of the register pair PC//SR; an illegally set S or L flag is also saved.

Then the difference between frame pointer FP - stack pointer SP(8..2) is evaluated and interpreted as a signed 7-bit integer. If the difference is not negative, the register pointed to by FP(5..0) is in the register part of the stack; no further action is then required and the Return instruction is completed.

If the difference is negative, the number of words equal to the negative difference are pulled from the memory part of the stack and transferred to the register part of the stack, beginning with the contents of the memory location SP - 4 being transferred to the local register addressed absolutely by bits 7..2 of SP - 4. After each memory cycle, the SP is decremented by four until the difference is eliminated. In case of a Data Page Fault, the SP contains the address of the fault-causing memory location.

The Return instruction shares its basic OP-code with the Move Double-Word instruction. It is differentiated from it by denoting the PC as destination register Rd.

The PC or the SR must not be denoted as a source operand; these notations are reserved for future expansion.

3.30 Return Instruction (continued)

```
Format  Notation          Operation

RR      RET, PC, Rs;      old S := S;
                          old L := L;
                          PC := Rs(31..1);
                          SR := Rsf(31..21)//00//Rs(0)//Rsf(17..0);
                              -- ILC := 0;
                              -- S := Rs(0);
                          if old S = 0 and S = 1 or
                              S = 0 and old L = 0 and L = 1 then
                              trap -> Privilege Error;
                          difference(6..0) := FP - SP(8..2);
                              -- difference is signed, difference(6) = sign bit
                          if difference >= 0 then continue at next instruction;
                              -- RET is finished
                          repeat
                              SP := SP - 4;
                              register SP(8..2)^ := memory SP^;
                                  -- memory -> local register
                              if Page Fault then
                                  trap -> Data Page Fault;
                              difference := difference + 1;
                          until difference = 0;
```

3.31 Fetch Instruction

The instruction execution is halted until a number of at least n/2 + 1 (1..16) instruction halfwords succeeding the Fetch instruction are prefetched in the instruction cache. Since instruction words are fetched, one more halfword may be fetched. The number n/2 is derived by using bits 4..1 of n, bit 0 of n must be zero.
A Page Fault signalled during the prefetch marks the instruction prefetch address and causes instruction execution to resume; the trap to Instruction Page Fault is delayed until the instruction decode meets the marked instruction prefetch address.
The Fetch instruction must not be placed as a delay instruction; when the preceding branch is taken, the prefetch is undefined.
The Fetch instruction shares the basic OP-code SETxx, it is differentiated by denoting the SR for the Rd-code (see instruction formats).

n	Format	Notation	Operation
0	Rn	FETCH, 1;	Wait until n/2 + 1 instruction halfwords are fetched;
.	.	.	
.	.	.	
.	.	.	
30	Rn	FETCH, 16;	

Note: The Fetch instruction supplements the standard prefetch of instruction words. It may be used to speed up the execution of a sequence of memory instructions by avoiding alternating between instruction and data memory pages. By executing a Fetch instruction preceding a sequence of memory instructions addressing the same data memory page, the memory accesses can be constrained to the data memory page by prefetching all required instructions in advance.
A Fetch instruction may also be used preceding a branch into a program loop; thus, flushing the cache by the first branch repeating the loop can be avoided.

3.32 Software Instructions

The Software instructions cause a branch to the subprogram associated with each Software instruction. Its entry address (see entry table), deduced from the OP-code of the Software instruction, is placed in the program counter PC. Data is saved in the register sequence beginning at register address FP + FL (FL = 0 is interpreted as FL = 16) in ascending order as follows:

- Stack address of the destination operand
- High-order word of the source operand
- Low-order word of the source operand
- Old program counter PC, containing the return address and the old S flag in bit zero
- Old status Register SR, ILC contains the instruction-length code (1) of the software instruction

Then the frame pointer FP is incremented by the old frame length FL and FL is set to six, thus creating a new stack frame. The cache-mode flag M and the trace-mode flag T are cleared, the interrupt-lock flag L is set to one. All condition flags remain unchanged.
Instruction execution then proceeds at the entry address placed in the PC.
Ls or Lsf and Ld may denote the same register.

Format	Notation	Operation

LL	see specific instructions	PC := 23 ones//0//OP(11..8)//4 zeros;
		(FP + FL)^ := stack address of Ld;
		(FP + FL + 1)^ := Ls;
		(FP + FL + 2)^ := Lsf;
		(FP + FL + 3)^ := old PC(31..1)//old S;
		(FP + FL + 4)^ := old SR;
		FP := FP + FL; -- FL = 0 is treated as FL = 16
		FL := 6;
		M := 0;
		T := 0;
		L := 1;

Note: At the new stack frame, the stack address of the destination operand can be addressed as L0, the source operand as L1//L2, the saved PC as L3 and the saved SR as L4; L5 is free for use as required. As long as the interrupt-lock flag L flag is kept at one and no Frame instruction or any instruction with the potential to cause an exception is executed, registers L6..L9 can also be used.
A Frame instruction must be executed before executing any other Software instruction, Trap or Call instruction, otherwise the beginning of the register part of the stack at SP could be overwritten without any warning.

3.32.1 Do Instruction

The Do instruction is executed as a Software instruction. The associated subprogram is entered, the stack address of the destination operand and one double-word source operand are passed to it (see Software instructions for details).
The halfword succeeding the Do instruction will be used by the associated subprogram to differentiate branches to subordinate routines; the associated subprogram must increment the saved return program counter PC by two.

Format	Notation	Operation
LL	DO xx..., Ld, Ls;	execute Software instruction;

"xx..." stands for the mnemonic of the differentiating halfword after the OP-code of the Do instruction.
The Do instruction must not be placed as delay instruction since then xx... cannot be located.

Note: The Do instruction provides very code efficient passing of parameters to routines executing software implemented extensions of the instruction set, like string instructions for example.
Branching to unimplemented subordinate routines with the interrupt-lock flag L set to one must be excluded by bounds checks of the differentiating halfword at runtime; out-of-range values cannot be securely excluded at the assembly level.
The L flag must be cleared when the execution of a subordinate routine exceeds the regular interrupt latency time.

Application Note: The definition of subprograms entered via the Do instruction is reserved for system implementations. The values assigned to the differentiating halfword xx... after the OP-code of the Do instruction must be in ascending and contiguous order, starting with zero. This order enables fast range checking for an upper bound and also avoids unused space in the differentiating branch table.

3.32.2 Extend Instruction

The Extend instruction is strictly reserved for future expansion of the instruction set by instructions executed in hardware. The halfword succeeding the instruction is used for further differentiation.

In the present version, the Extend instruction is executed as a Software instruction to provide emulation of unimplemented hardware extensions.

The associated subprogram is entered, the stack address of the destination operand and the double-word source operand are passed to it (see Software instructions for details).

The halfword succeeding the EX instruction will be used by the associated subprogram to differentiate branches to subordinate routines; the associated subprogram must increment the saved return program counter PC by two.

Format	Notation	Operation
LL	EX xx..., Ld, Ls;	execute Software instruction;

"xx..." stands for the mnemonic of an instruction of the extended instruction set.

The Extend instruction must not be placed as delay instruction since then xx... cannot be located.

Note: Branching to unimplemented subordinate routines with the interrupt-lock flag L set to one must be excluded by bounds checks of the differentiating halfword at runtime; out-of-range values cannot be securely excluded at the assembly level.

3.32.3 Floating-Point Instructions

The Floating-Point instructions comply with the ANSI/IEEE standard 754-1985. In the present version, they are executed as Software instructions. The following description provides a general overview of the architectural integration.
The basic instructions use single-precision (single-word) and double-precision (double-word) operands, the instruction set may be extended by the Extend instruction to cover a comprehensive instruction set and extended-precision operands.
All Floating-Point instructions except those defined by an Extend instruction can be placed as delay instructions.
Except at the Floating-Point Compare instructions, all condition flags remain unchanged to allow future concurrent execution.
The rounding modes FRM are encoded as:

SR(14) = 0; SR(13) = 0;	Round to nearest
SR(14) = 0; SR(13) = 1;	Round toward zero;
SR(14) = 1; SR(13) = 0;	Round toward - infinity
SR(14) = 1; SR(13) = 1;	Round toward + infinity

The floating-point trap enable flags FTE and the exception flags are assigned as:

floating-point trap enable FTE	accrued exceptions	actual exceptions	exception type
SR(12)	G2(4)	G2(12)	Invalid Operation
SR(11)	G2(3)	G2(11)	Division by Zero
SR(10)	G2(2)	G2(10)	Overflow
SR(9)	G2(1)	G2(9)	Underflow
SR(8)	G2(0)	G2(8)	Inexact

The reserved bits G2(31..13) and G2(7..5) must be zero.
A floating-point Not a Number (NaN) is encoded by bits 30..19 = all ones in the operand word containing the exponent; all other bits of the operand are ignored for differentiating a NaN from a non-NaN.
In the case of an operand word containing a NaN, bit zero = 0 differentiates a quiet NaN, bit zero = 1 differentiates a signalling NaN; the bits 18..1 may be used to encode further information.

3.32.3 Floating-Point Instructions (continued)

Format	Notation	Operation
LL	FADD, Ld, Ls;	Ld := Ld + Ls;
LL	FADDD, Ld, Ls;	Ld//Ldf := (Ld//Ldf) + (Ls//Lsf);
LL	FSUB, Ld, Ls;	Ld := Ld - Ls;
LL	FSUBD, Ld, Ls;	Ld//Ldf := (Ld//Ldf) - (Ls//Lsf);
LL	FMUL, Ld, Ls;	Ld := Ld * Ls;
LL	FMULD, Ld, Ls;	Ld//Ldf := (Ld//Ldf) * (Ls//Lsf);
LL	FDIV, Ld, Ls;	Ld := Ld / Ls;
LL	FDIVD, Ld, Ls;	Ld//Ldf := (Ld//Ldf) / (Ls//Lsf);
LL	FCVT, Ld, Ls;	Ld := Ls//Lsf; -- Convert double -> single
LL	FCVTD, Ld, Ls;	Ld//Ldf := Ls; -- Convert single -> double
LL	FCMP, Ld, Ls;	result := Ld - Ls; Z := Ld = Ls and not unordered; N := Ld < Ls or unordered; C := Ld < Ls and not unordered; V := unordered; if unordered then Invalid Operation exception;
LL	FCMPD, Ld, Ls;	result := (Ld//Ldf) - (Ls//Lsf); Z := (Ld//Ldf) = (Ls//Lsf) and not unordered; N := (Ld//Ldf) < (Ls//Lsf) or unordered; C := (Ld//Ldf) < (Ls//Lsf) and not unordered; V := unordered; if unordered then Invalid Operation exception;
LL	FCMPU, Ld, Ls;	result := Ld - Ls; Z := Ld = Ls and not unordered; N := Ld < Ls or unordered; C := Ld < Ls and not unordered; V := unordered; -- no exception
LL	FCMPUD, Ld, Ls;	result := (Ld//Ldf) - (Ls//Lsf); Z := (Ld//Ldf) = (Ls//Lsf) and not unordered; N := (Ld//Ldf) < (Ls//Lsf) or unordered; C := (Ld//Ldf) < (Ls//Lsf) and not unordered; V := unordered; -- no exception

3.32.3 Floating-Point Instructions (continued)

A floating-point instruction, except a Floating-point Compare, can raise any of the exceptions Invalid Operation, Division by Zero, Overflow, Underflow or Inexact. FCMP and FCMPD can raise only the Invalid Operation exception (at unordered). FCMPU and FCMPUD cannot raise any exception.
At an exception, the following additional action is performed:

- Any corresponding accrued-exception flag whose corresponding trap-enable flag is zero (not enabled) is set to one; all other accrued-exception flags remain unchanged.
- If a corresponding trap-enable flag is one (enabled), any corresponding actual-exception flag is set to one; all other actual-exception flags are cleared. The destination remains unchanged.
  In the present software version, the software emulation routine must branch to the corresponding user-supplied trap handler. The (modified) result, the source operand, the stack address of the destination operand and the address of the floating-point instruction should be passed to the trap handler. In the future hardware version, a trap to Range Error will occur; the Range Error handler will then initiate re-execution of the floating-point instruction by branching to the entry of the corresponding software emulation routine, which will then act as described before.

The only exceptions that can coincide are Inexact with Overflow and Inexact with Underflow. An Overflow or Underflow trap, if enabled, takes precedence over an Inexact trap; the Inexact accrued-exception flag G2(0) must then be set as well.

3.32.3 Floating-Point Instructions (continued)

The table below shows the combinations of Floating-Point Compare and Branch instructions to test all 14 floating-point relations:

relation	Compare	Branch on true	Branch on false	exception if unordered
=	FCMPU	BE	BNE	--
?<>	FCMPU	BNE	BE	--
>	FCMP	BGT	BLE	x
>=	FCMP	BGE	BLT	x
<	FCMP	BLT	BGE	x
<=	FCMP	BLE	BGT	x
?	FCMPU	BV	BNV	--
<>	FCMP	BNE	BE	x
<=>	FCMP	--	--	x
?>	FCMPU	BHT	BSE	--
?>=	FCMPU	BHE	BST	--
?<	FCMPU	BLT	BGE	--
?<=	FCMPU	BLE	BGT	--
?=	FCMPU	BE,BV	BST,BGT	--

The symbol ? signifies unordered.

Note: At the test <=> (ordered), no branch after FCMP is required since the result of the test is an Invalid Operation exception occurred or not occurred.

4 Exceptions

4.1 Exception Processing

Exceptions are events which redirect the flow of control to a supervisor subprogram associated with the type of exception, that is, the program is "trapped" to respond to the exception. (See a detailed description of exceptions further below.) If exceptions coincide, the exception with the highest priority takes precedence over all exceptions with lower priority.

Processing of an exception proceeds as follows:

The entry address (see entry table) of the associated subprogram is placed in the program counter PC and the supervisor-state flag S is set to one. The old PC is saved in the register addressed by FP + FL; the old S flag is also saved in bit zero of this register. The old status register SR is saved in the register addressed by FP + FL + 1 (FL = 0 is interpreted as FL = 16); the saved instruction-length code ILC contains (in general, see backtracking) the instruction-length code of the preceding instruction.

Then the frame pointer FP is incremented by the old frame length FL and FL is set to two, thus creating a new stack frame. The cache-mode flag M and the trace-mode flag T are cleared, the interrupt-lock flag L is set to one. All condition flags remain unchanged.

Operation

```
PC := entry address of exception subprogram;
S := 1;
(FP + FL)^ := old PC(31..1)//old S;
(FP + FL + 1)^ := old SR;
FP := FP + FL;      -- FL = 0 is treated as FL = 16
FL := 2;
M := 0;
T := 0;
L := 1;
```

Note: At the new stack frame, the saved PC can be addressed as L0 and the saved SR as L1. Since FL = 2, no other local registers are free for use.

A Frame instruction must be executed before the interrupt-lock flag L is cleared, before any Call, Trap, Software instruction or any instruction with the potential to cause an exception is executed or before an Instruction Page Fault can occur. Otherwise, the beginning of the register part of the stack at the SP could be overwritten without any warning.

An entry caused by an exception can be differentiated from an entry caused by a Trap instruction by the value of FL: FL is set to two by an exception and set to six by a Trap instruction.

4.2 Exception Types

The exception types are distinguished by a separate trap entry (see entry table) for each type. They are ordered by priorities, Reset has the highest, Instruction Page Fault has the lowest priority.

4.2.1 Reset

The Reset exception occurs on a transition of the Reset signal from low to high. It overrules all other exceptions and is used to start execution at the Reset entry after power-on.
The load and store pipelines are cleared and all bits of the BCR are set to one by hardware; all other registers and flags, except those set or cleared explicitly by the exception processing itself, remain undefined and must be initialized by software. All reserved bits (except at the BCR) must be cleared.

Note: The frame pointer FP can only be set to a defined value by restoring it from the FP in the return SR through a Return instruction.

4.2.2 Data Page Fault

A Data Page Fault exception occurs when a Page Fault is signalled from the (external) memory management unit (MMU) on a memory access into data address space. The fault-causing instruction can be identified by backtracking. When the postincrement, next address or stack address mode is used by this instruction, the content of the address register (Ld or Rd) must be decremented by the specified data size or by the signed displacement value dis respectively. Except at byte data size, dis(0) must be treated as zero regardless of its value; at stack address mode, dis(1) must also be treated as zero.
After correction of the Data Page Fault (by loading the missing page), the instruction can be repeated. When the fault-causing instruction is placed as delay instruction and the preceding delayed branch is taken, the Delayed Branch instruction must be repeated. When the Data Page Fault is caused by a double-word memory instruction crossing a page boundary, the Data Page Fault could be caused by either the first or second memory cycle and thus, both pages must be checked.

4.2.3 Range, Pointer, Frame and Privilege Error

These exceptions share a common entry since they cannot occur coincidentally at the same instruction. The error-causing instruction can be identified by backtracking.
A Range Error exception occurs when an operand or result exceeds its value range.
A Pointer Error is caused by an attempted memory access using an address register (Rd or Ld) with the content zero. The memory is not accessed, but the content of the address register is updated in case of a postincrement or next address mode.
A Frame Error occurs when the restructuring of the stack frame reaches or exceeds the upper bound UB of the memory part of the stack. No further Frame instruction must be executed by the error routine for Pointer, Frame and Privilege Error before the UB is set to a higher value and thus, an expanded stack frame fits into the higher stack bound.
A Privilege Error occurs when a privileged operation is executed in user or on return to user state (see privilege states for details).

4.2.4 Interrupt Exception

An Interrupt exception is caused by an external interrupt signal. Since the interrupt-lock flag L is set, no further interrupts can occur until the L flag is cleared.

Note: Only a maskable interrupt is provided; a non-maskable interrupt occurring at a task exchange or at the handling of a Frame Error would cause an inextricable intermingling of stack data.
The single-entry interrupt scheme used yields a much shorter interrupt latency time than the conventional vectored interrupt schemes since pending tasks with higher priority can be tested after the present task context is saved in memory. As an additional advantage, no specific restricting interrupt protocol is required; task priorities can be assigned and reassigned dynamically.

4.2.5 Trace Exception

A Trace exception occurs after each execution of an instruction except a Delayed Branch instruction when the trace mode is enabled (trace flag T = 1) and the trace pending flag P is one. A Trace exception can also be enforced by a TRACE signal (see bus signals 5.4) when the interrupt lock flag L is zero and the trace pending flag P is one.
The P flag in the saved return status register SR must be cleared by the trace handler to prevent tracing the same instruction again.
The instruction preceding the Trace exception cannot be backtracked since only potentially error-causing instructions can and need be backtracked.
Note: Constraining the TRACE signal to be effective only when interrupts are locked out inhibits interrupt or trace handler programs to be further interrupted by Trace exceptions.

4.2.6 Instruction Page Fault

An Instruction Page Fault exception occurs when the instruction decode meets an instruction address which has been marked by the instruction prefetch control as having caused a Page Fault signal. Thus, a Page Fault signalled at an instruction fetch memory access causes no immediate trap; the trap occurs only when an attempt is made to decode for execution the instruction which could not be fetched.
The return address saved is the address of the missing instruction and thus, no backtracking is required. The instruction can be repeated after correction by using its address as return address.

4.3 Exception Backtracking

In the case of a Pointer, Frame, Privilege and Range Error and in the case of a Data Page Fault exception caused by a delay instruction succeeding a delayed branch taken, the location of the saved PC contains the address of the delay instruction and the saved instruction length code ILC contains the length of the Delayed Branch instruction (in halfwords).
In the case of all other exceptions, the location of the saved PC contains the return address, that is, the address of the instruction which would have been executed next if the exception had not occurred. The saved ILC contains the length of the last instruction except when the last instruction executed was a branch taken; a Return instruction clears the ILC and thus, the saved ILC after a Return instruction contains zero.

4.3 Exception Backtracking (continued)

An exception caused by a Pointer, Frame, Privilege or Range Error or by a Data Page Fault, except following a Return instruction, can be backtracked. For backtracking, the content of the adjusted saved ILC is subtracted from the address contained in the location of the saved PC.
If the backtrack-address calculated in this way points to a Delayed Branch instruction, the error- or fault-causing instruction is a delay instruction with a preceding delayed branch taken and the address contained in the location of the saved PC points to the address of this delay instruction.
If the backtrack-address calculated does not point to a Delayed Branch instruction, it points directly to the error- or fault-causing instruction. This instruction is then either not a delay instruction or a delay instruction with the preceding delayed branch not taken.
The error- or fault-causing instruction can then be inspected and the cause of an error analyzed in detail.
In the case of a Privilege Error or a Data Page Fault, the ILC must be tested for zero to single out an exception caused by a Return instruction before backtracking. Thus, an exception caused by a Return instruction can be identified. However, it cannot be backtracked to the instruction address of the Return instruction because the return address saved does not succeed the address of the Return instruction. All other branching instructions cannot be backtracked either. Since these instructions cause no errors, backtracking is not required.
The stack address of a local register denoted by a backtracked instruction can be calculated according to the following formula:

```
stack address of preceding stack frame := stack address of
current stack frame - (((FP - saved FP) modulo 64) * 4);
  -- bits 5..0 of the difference (FP - saved FP) are used zero-expanded
  -- * 4 converts word difference -> byte difference
  -- the stack address of the current stack frame is provided by the
     Set Stack Address instruction
stack address of local register := stack address of preceding
stack frame + (local register address code * 4);
  -- * 4 converts local register word offset -> byte offset
```

Note: Backtracking allows a much more detailed analysis of error causes than a more differentiated trapping could provide. Exception handlers can get more information about error causes and the precise messages required by most programming languages can be easily generated.

5 Bus Interface

5.1 Bus Control General

The processor provides on-chip all functions to control memory and peripheral devices, including RAS-CAS multiplexing, DRAM refresh and parity generation and checking. The number of bus cycles used for a memory or I/O access is also defined by the processor, thus, no external bus controllers are required.
The memory address space is divided into four equal partitions. Bits 31 and 30 of the address define these partitions as follows:

bits 31, 30:

0	0	DRAM address space MEM0
0	1	ROM or SRAM address space MEM1
1	0	ROM or SRAM address space MEM2
1	1	ROM address space MEM3

The bus timing, refresh control, page fault and parity error disable for memory access is defined in the bus control register BCR. The bus timing for I/O access is defined by address bits in the I/O address.
An access can be subdivided into three groups of cycles:

- Address setup time. During address setup time, none of the enable signals READEN or WRT0..WRT3 is activated high. RASEN is enabled to its last state. One or more address setup cycles (encoded by one bit) can be specified for MEM2 or I/O accesses.
- Access time. During access time, the corresponding CASEN, READEN or WRT0..WRT3 signals are activated high. At MEM0, RASEN is also high, otherwise enabled to the last state.
  One to eight access time cycles (encoded as 0..7 by a 3-bit code respectively) can be specified for memory or I/O accesses.
- Bus hold time. During bus hold time, none of the enable signals READEN or WRT0..WRT3 is activated high, RASEN is enabled to its last state. Zero to three bus hold cycles (encoded as 0..3 by a 2-bit code respectively) can be specified for MEM2, MEM3 and I/O accesses.

Additional bus cycles can be specified in BCR for the DRAM (MEM0):

- A RAS precharge time of zero to three cycles (encoded as 0..3 by a 2-bit code respectively) can be specified.
- A RAS to CAS delay time of zero to three cycles (encoded as 0..3 by a 2-bit code respectively) can be specified.

5.1 Bus Control General (continued)

The refresh timing for the DRAM (CAS-before-RAS refresh) can be specified by a 2-bit code in BCR as follows:

00	Refresh after 512 clock cycles
01	Refresh after 256 clock cycles
10	Refresh after 128 clock cycles
11	Refresh disabled

The physical page size of the DRAM is specified by a 3-bit code in BCR as follows:

Code	DRAM chip organization	Capacity of one memory block	Row address range	Column address range
0	256M x 1 / 256M x 4	1024 Mbyte	A29..A16	A15..A2
1	64M x 1 / 64M x 4	256 Mbyte	A27..A15	A14..A2
2	16M x 1 / 16M x 4	64 Mbyte	A25..A14	A13..A2
3	4M x 1 / 4M x 4	16 Mbyte	A23..A13	A12..A2
4	1M x 1 / 1M x 4	4 Mbyte	A21..A12	A11..A2
5	256K x 1 / 256K x 4	1 Mbyte	A19..A11	A10..A2
6	64K x 1 / 64K x 4	256 Kbyte	A17..A10	A9..A2
7	16K x 1 / 16K x 4	64 Kbyte	A15..A9	A8..A2

The high-order row address (RAS) bits multiplexed to the low-order (CAS) bit positions at a RAS access are not in any specific order.

For the DRAM, a bit in the BCR specifies also whether a page fault signal is accepted (enabled, bit = 0) or ignored (disabled, bit = 1) when a new page is accessed.
The parity checks can be enabled or disabled separately for each of the four address spaces MEM0..MEM3.

5.2 Bus Control Register BCR

All bits of the BCR are set to one by RESET#. They are intended to be initiated according to the hardware enviroment.
The content of the BCR cannot be read by any instruction.

BCR (31..28):	Parity check disable for address space MEM3..MEM0 (BCR(31) = 1 disables parity check for MEM3)
BCR (27):	reserved (must always be 1)
BCR (26..24):	Access time for address space MEM3 (1..8 cycles)
BCR (23):	reserved (must always be 1)
BCR (22..20):	Access time for address space MEM2 (1..8 cycles)
BCR (19..18):	Access time for address space MEM1 (1..4 cycles)
BCR (17..16):	Access time for address space MEM0 (1..4 cycles)
BCR (15):	reserved (must always be 1)
BCR (14):	Address setup time for address space MEM2 (0..1 cycles)
BCR (13..12):	Refresh select (see table)
BCR (11..10):	RAS Precharge time (0..3 cycles, select 0 when MEM0 is not a DRAM)
BCR (9..8):	RAS to CAS delay time (0..3 cycles, select 0 when MEM0 is not a DRAM)
BCR (7):	Page Fault disable (1 = disable)
BCR (6..4):	Page size code (see table)
BCR (3..2):	Bus hold time for address space 3 (0..3 cycles)
BCR (1..0):	Bus hold time for address space 2 (0..3 cycles)

5.3 I/O Bus Control

With I/O addresses, address setup, access and bus hold time can be specified by bits in the I/O address as follows:

<u>IO - Address Decoding</u>

Bits 2..0:	Reserved for internal use (bit 2 must be 0 at double word)
Bits 4..3:	Bus hold time (0..3 bus hold cycles after read or write access)
Bits 7..5:	Access time (1..8 read or write access cycles)
Bit 8:	Address setup time (0 or 1 cycle before read or write)
Bits 11..9:	3-bit short I/O address
Bits 27..9:	19-bit long I/O address

Bit 16 is suggested to be specified as zero at a short I/O address (halfword) and as one at a long I/O address (word).

5.4 Bus Signals

In the following signal description, state I = input, O = output and Z = three-state (inactive).

States	Names	Use
I	**CLK**	Clock signal. A clock cycle corresponds to a processor cycle, that is, the clock signal is used undivided.
O	**REQST**	Request address bus. REQST is signalled high to request the address bus for the next cycle(s).
O	**REQSTH**	Request address bus high priority. REQSTH is signalled high to request the address bus for a refresh cycle. It must be assigned the highest priority in a system with multiple bus masters.
I	**GRANT#**	Grant address bus. GRANT# must be signalled low by external logic to assert access to the address bus for the next cycle(s). Grant# must not be combined with a clock. (See recommended circuits for bus arbitration.) When the processor is the only busmaster, GRANT# can be tied low.
O/Z	**A31..A2**	The address bits A31..A2 represent the address bus. At an address bus cycle, A31..A2 are activated from three-state to either low or high; an active high bit signals a "one". A31 is the most significant bit.

The signals S, I/O, ACC, are the memory mode signals. They are activated (from three-state to either low or high) at the same cycle and with the same timing as the address bus.

O/Z	**S**	Supervisor state. Active high indicates supervisor state, active low indicates user state.
O/Z	**I/O**	I/O address space. Active high indicates I/O address space, active low indicates memory address space. I/O addresses with bit 31 = 1 are reserved for hardware extensions and system modules.
O/Z	**ACC**	Access code. Active high indicates instruction access, active low indicates data or refresh access.

5.4 Bus Signals (continued)

States	Names	Use
O	**AV#**	Address Valid. AV# low signals an address bus cycle, that is, it indicates that the address bus and memory mode signals are activated. AV# low precedes the address and memory mode signals by half a clock cycle.
O/Z	**AV1**	First address cycle. AV1 is activated at the same cycle and with the same timing as the address bus. AV1 is signalled active high at the first address cycle of a memory, refresh or I/O access and signalled active low at (optionally) succeeding address cycles of the same access. It may be used to synchronize peripheral devices in the (rare) cases where the timing provided by the processor must be supplemented.
I	**ACT**	Address bus active. ACT must be signalled high by the external control logic whenever the address bus is active (AV# = low), regardless which bus master (processor or DMA controller) performs the address bus cycle. Note: ACT is used internally to separate address bus cycles of different bus masters by an idle cycle. ACT is created by OR-ing the AV# signals of all potential bus masters. Since an active AV# is low, the OR-ing may be accomplished by connecting all AV# signals to the inputs of a Nand gate; the output signal of the Nand gate is ACT. No clock must be used for creating ACT.
I	**PGFLT**	Page Fault. PGFLT high signals a Page Fault from an external MMU to the processor. The PGFLT signal is only accepted when the processor accesses a new page in the DRAM. Acceptance of PGFLT can be disabled by a control bit in the bus control register BCR.

5.4 Bus Signals (continued)

States	Names	Use
O/I	**RASEN**	Row address enable. RASEN is activated in the same cycle(s) as the address bits. Active high indicates row address enable, active low indicates row address disable. Since RASEN is three-state, its state must be held in an external latch or flip-flop while RASEN is deactivated; the output of this latch may be used to control the RAS signal pin of a DRAM. RASEN is activated low and then again high when the processor accesses a new page in the DRAM address space, that is when any of the (high order) RAS address bits is different from the RAS address bits of the last DRAM access or when the processor-internal page state is "disabled". The internal page state is left "enabled" after any DRAM access. RASEN is activated low, high and then low by a refresh cycle. The internal page state is left "disabled". When the address bus is used by another bus master, RASEN is used as an input to the processor and a processor-internal RAS state is set or cleared according to the state of RASEN at the last active address cycle of that bus master. When RASEN was low at any such address cycle, the processor-internal page state is cleared to "disabled". At any non-RAS address cycle, RASEN is activated according to the internal RAS state, thus, the external RAS latch does not change its state and the RAS signal is not affected.
O/Z	**CASEN**	Column address enable. CASEN is activated in the same cycle(s) as the address bits. Active high indicates column address enable, active low indicates column address disable. CASEN is only used by a DRAM for column access cycles and for "CAS before RAS" refresh. CASEN must be forwarded by external latches to the corresponding CAS cycle(s).

5.4 Bus Signals (continued)

States	Names	Use
O/I	**READEN**	Read enable. READEN is activated in the same cycle(s) as the address bits. Active high indicates a read cycle. READEN is only high at the actual read access cycle(s); it is low at the (optional) address setup and hold cycle(s). Data from the data bus D31..D0 is transferred to the register set or instruction cache only at the cycle corresponding to the last actual read access cycle. READEN must be forwarded by external latches to the corresponding read cycle(s).
O/Z	**WRT0..WRT3**	Write enable Byte 0..3. The WRT0..WRT3 signals are activated in the same cycle(s) as the address bits at any memory or I/O access. Active high indicates a write enable for the corresponding byte, active low indicates write disable. Any WRT0..WRT3 signal is only high at the actual write access cycle(s); it is low at the (optional) address setup and hold cycle(s). WRT0..WRT3 must be forwarded by external latches to the corresponding write cycle(s).
O/I	**D31..D0**	Data bus. The signals D31..D0 represent the bidirectional data bus; active high indicates a "one". At a read access, data is transferred from the data bus to the register set or to the instruction cache only at the cycle corresponding to the last actual read access cycle, thus inhibiting garbled data from being transferred. At a write access, the corresponding data bus signals of the word, halfword or byte to be written are activated at the address setup and write cycles, they are deactivated at the (optional) address bus hold cycle(s). WRT0, WRT1, WRT2, WRT3 correspond to byte addresses 0,1,2,3 and activation of D31..D24, D23..D16, D15..D8, D7..D0 respectively. The data bus signals corresponding to non-addressed halfwords or bytes remain deactivated during all write access cycles.

5.4 Bus Signals (continued)

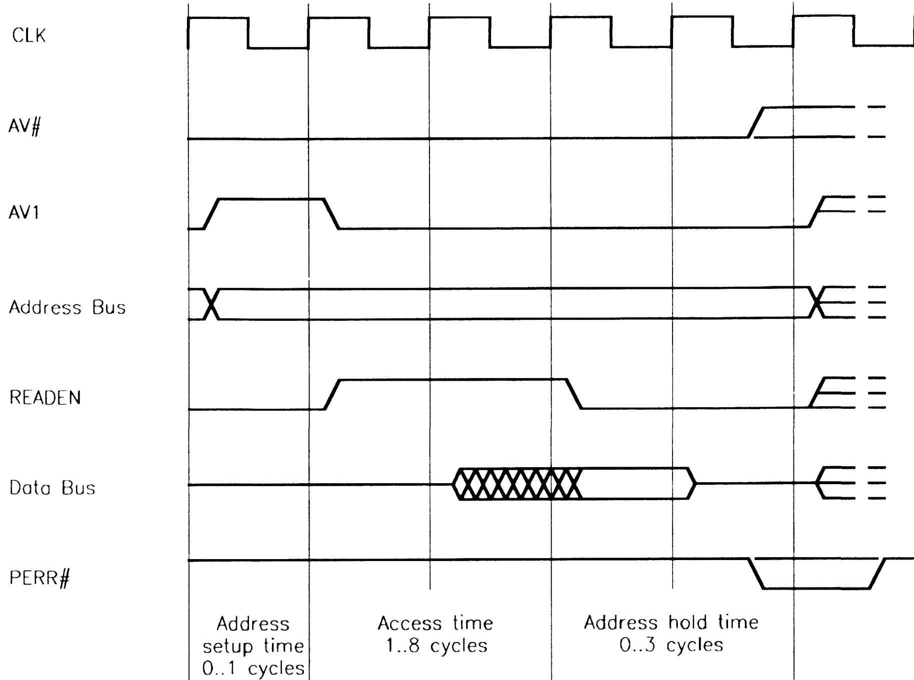
States	Names	Use
		Note: A halfword or byte to be written is placed from its right-adjusted position in a register to the addressed halfword or byte position. Thus, no external re-adjustment of data bit signals is required.
O/I	**DP0..DP3**	Data parity signals. DP0..DP3 represent the bidirectional parity signals; active high indicates a "one". The data parity signals DP0, DP1, DP2, DP3 correspond to the data bus signals D31..D24, D23..D16, D15..D8, D7..D0 respectively. At a write access, any of the signals DP0..DP3 is activated when its corresponding data bus signals are also activated. At a read access, DP0..DP3 are all evaluated at the last read access cycle regardless of the data size. Parity "odd" is used, that is, the correct parity bit is "one" when all bits of the corresponding byte are "zero".
O	**PERR#**	Parity error. PERR# low indicates a parity error occured in the preceding clock cycle. Since PERR# is signalled free of spikes, it can be fed directly into an interrupt controller. The signalling of PERR# can be disabled for each memory address space by the corresponding bit in the bus control register BCR; an I/O read access does not signal PERR#.
I	**RESET#**	Reset processor. RESET# low resets the processor to the initial state and halts all activity. RESET# must be low for at least two cycles. On a transition from low to high, a Reset exception occurs and the processor starts execution at the Reset entry (see entry table). The transition may occur asynchronously to the clock.

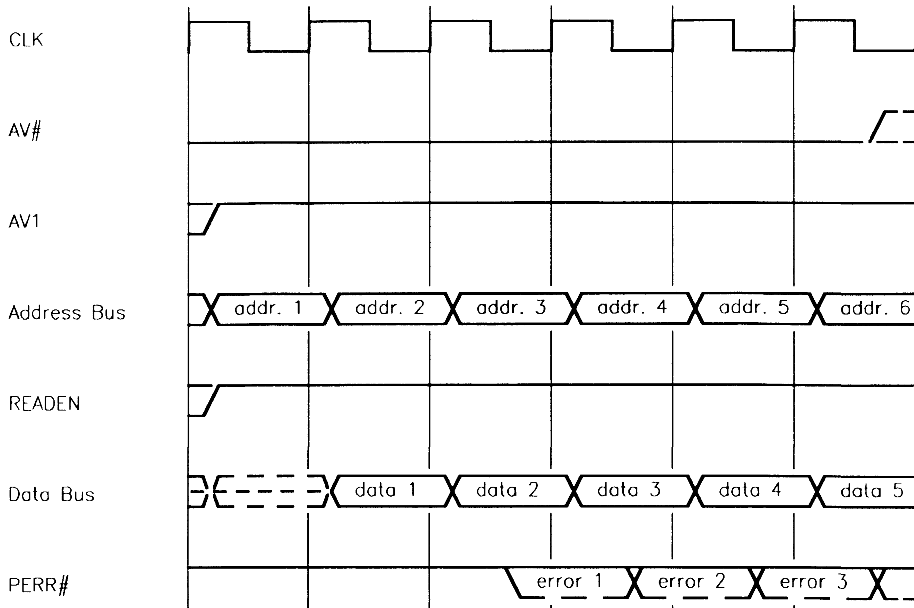
5.4 Bus Signals (continued)

States	Names	Use
I	**INT**	Interrupt request. INT high causes an Interrupt exception when the interrupt lock flag L is zero. INT may be signalled asynchronously to the clock; it is not stored internally. The delay time for INT is three cycles. That is, a transition of INT is effective after a minimum of three cycles. (The response time may be much higher depending on the number of cycles to the end of the current instruction or the number of cycles until the interrupt lock flag L is cleared.)
I	**TRACE**	Trace request. TRACE high causes a Trace exception when the trace pending flag P is one and the interrupt lock flag L is zero. TRACE must be signalled synchronously to the clock; it is not stored internally. TRACE has a delay time of one cycle. The next clock cycle of an instruction is always executed (unless interrupted otherwise); the Trace exception window is then effective after this next cycle until one cycle after TRACE transits to low. That is, a new instruction to be started in the Trace exception window is halted by the TRACE signal if the interrupt lock flag L is zero and the trace pending flag P is one. Trace exceptions can only occur between instructions, any executing instruction is run to completion before a Trace exception takes place.

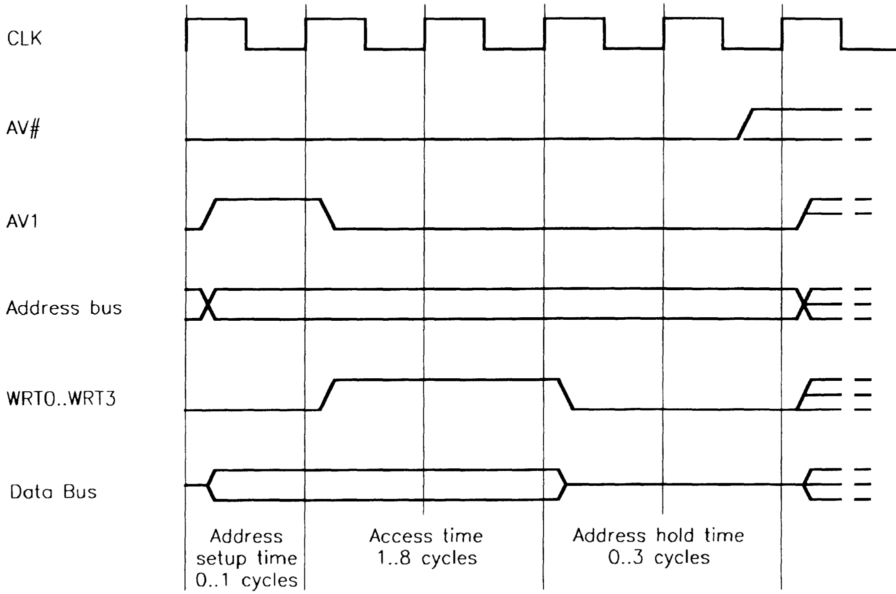
5.5 Bus Cycles

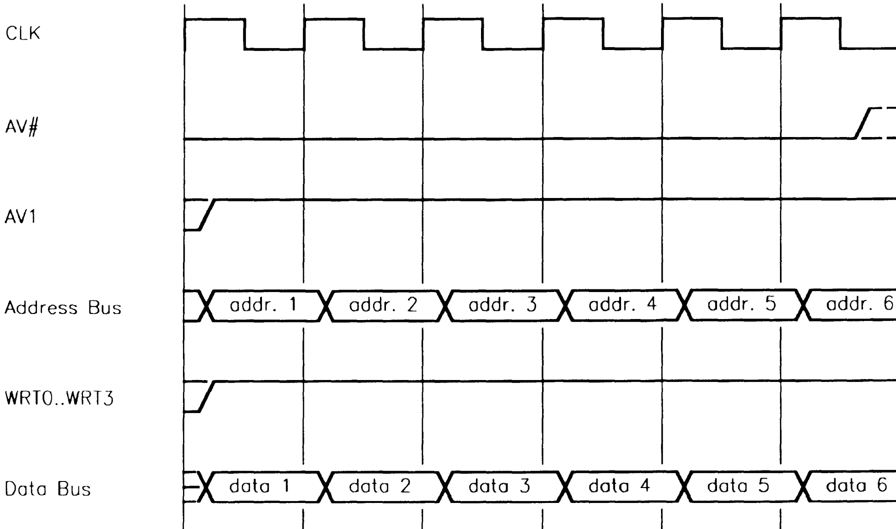
5.5.1 Read Access:



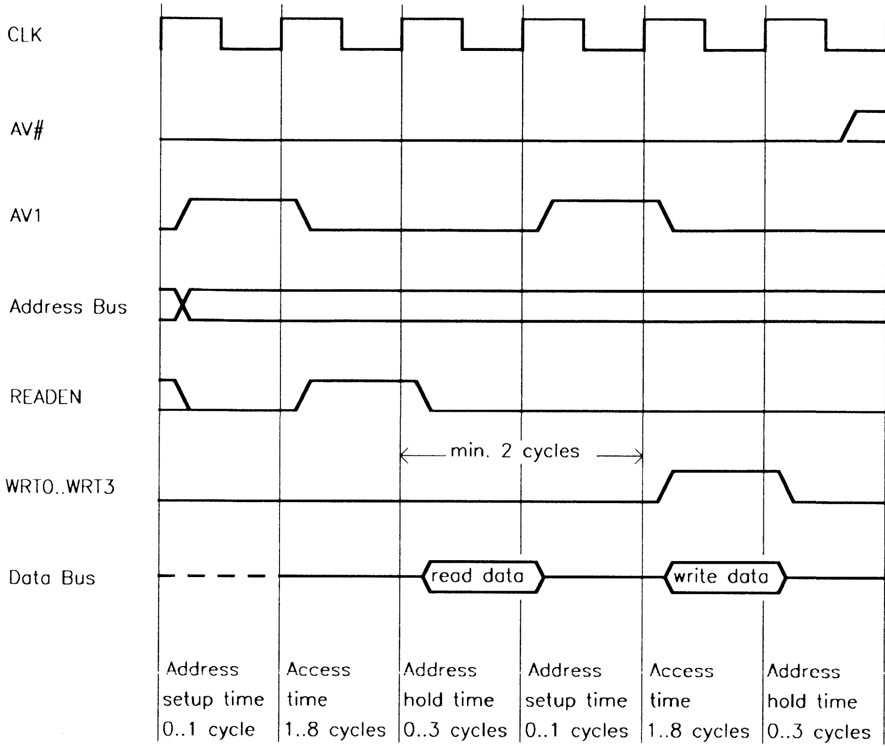


5.5.2 Write Access:



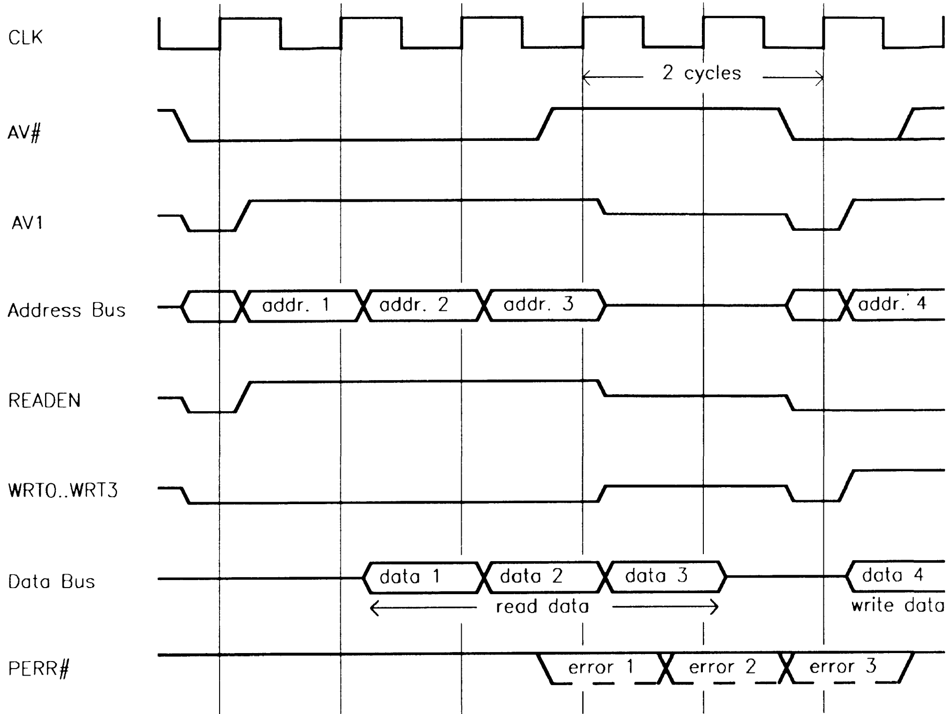


5.5.3 Exchange Access:



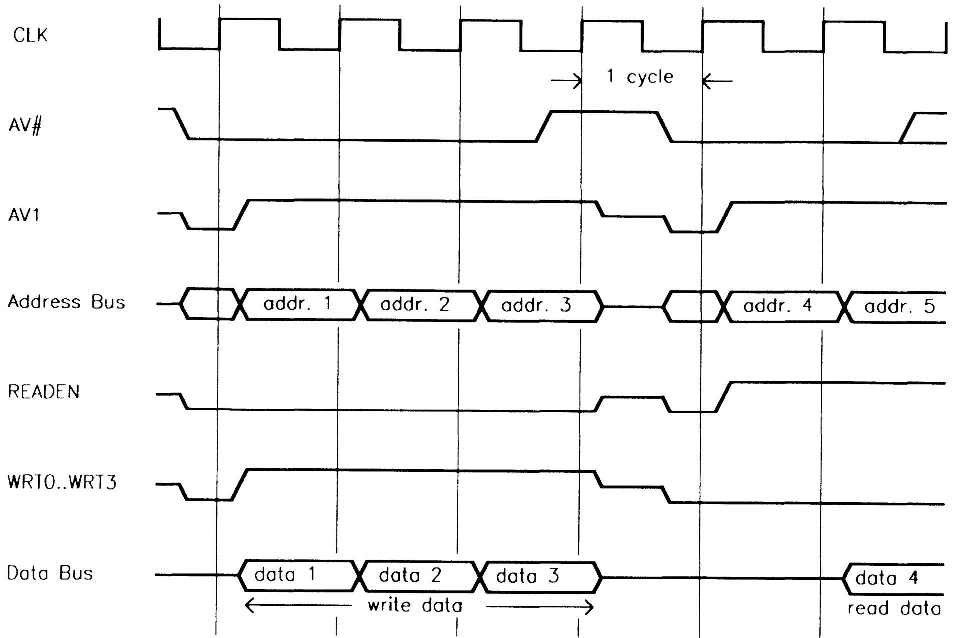
Note: Wait cycles are inserted automatically to guarantee a minimum of two idle cycles between the end of READEN high and the beginning of the activation of the data bus.

5.5.4 **Read/Write Access** (Read followed by Write)



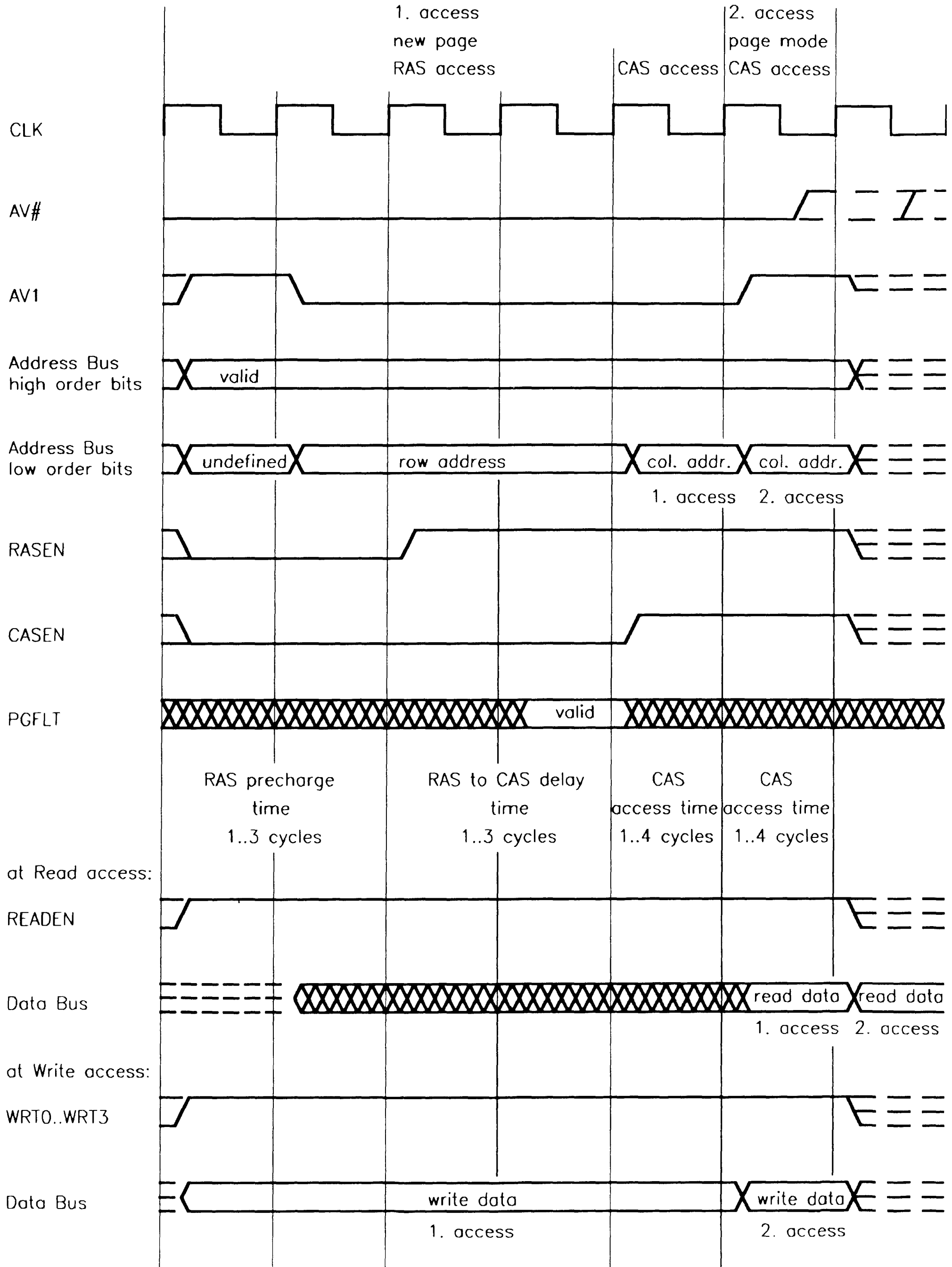
Note: Wait cycles are inserted automatically to guarantee a minimum of two idle cycles between the end of READEN high and the beginning of a subsequent write access.

5.5.5 Write/Read Access (Write followed by Read)



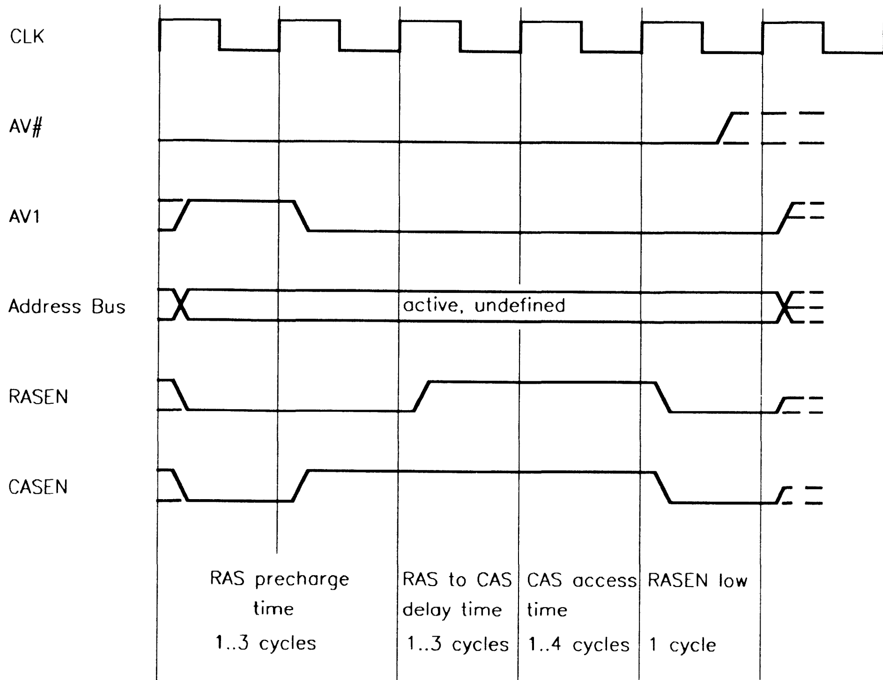
Note: A wait cycle is inserted automatically to guarantee a minimum of one idle cycle between the end of any WRT0..WRT3 high and the beginning of a subsequent read access.

5.5.6 DRAM Access:



Note: The window for PGFLT acceptance is always the last cycle of the RAS to CAS delay time.

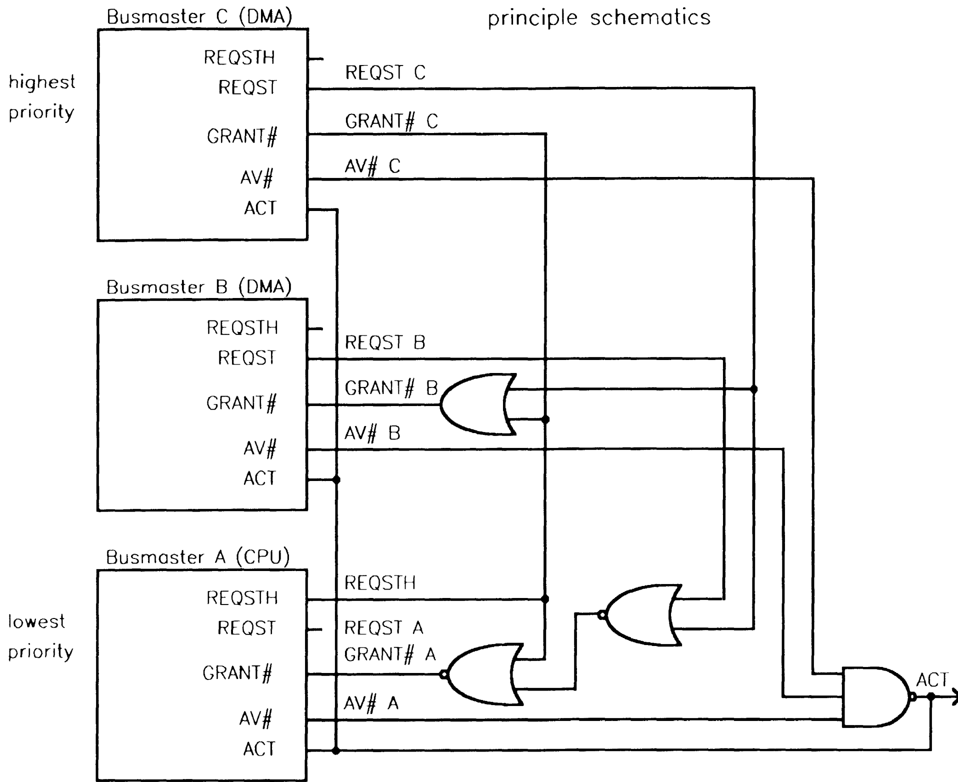
5.5.7 DRAM Refresh (CAS-before-RAS refresh)

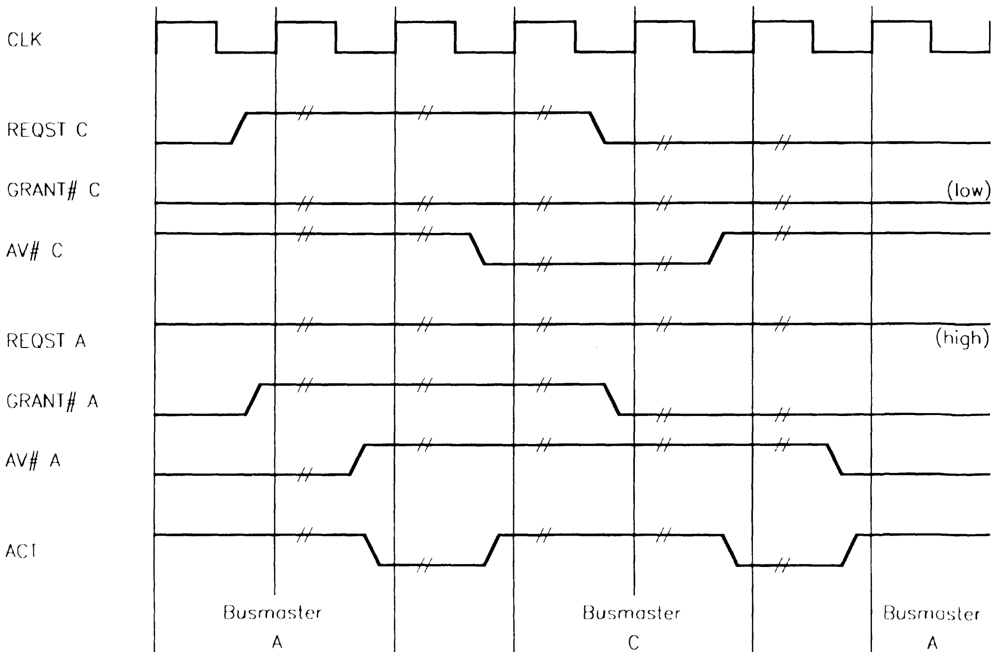


Note: The data bus and the parity signals are disactivated; READEN and WRT0..WRT3 are active low.

CASEN is activated high during the CAS access time, the RAS to CAS delay time and during the last cycle of the RAS precharge time provided that the RAS precharge time is longer than one cycle.

5.6 Bus Arbitration





5.7 D.C. Characteristics

Preliminary

Absolute Maximum Ratings:

Case temperature T_C under Bias	0°C to 85°C
Storage Temperature	-65°C to +150°C
Voltage on any Pin with respect to ground	-0.5 to VCC + 0.5V

D.C. Parameters:

Supply Voltage V_{CC}:	5V ± 0.25V
Case Temperature T_C:	0 to 85 C°

Symbol	Parameter	Min	Max	Units	Notes
V_{IL}	Input LOW Voltage	-0.3	+0.8	V	
V_{IH}	Input HIGH Voltage	2.0	V_{CC} + 0.3	V	
V_{OL}	Output LOW Voltage		0.45	V	at 4mA
V_{OH}	Output HIGH Voltage	2.4		V	at 1mA
I_{CC}	Power Supply Current				
	CLK = 25 MHz		250	mA	V_{CC} @5V
	CLK = 20 MHz		210	mA	V_{CC} @5V
I_{LI}	Input Leakage Current		± 20	μA	
I_{LO}	Output Leakage Current		± 20	μA	
C_{IN}	Input Capacitance		10	pF	
C_O	O/I or Output Capacitance		15	pF	
C_{CLK}	Clock Capacitance		15	pF	

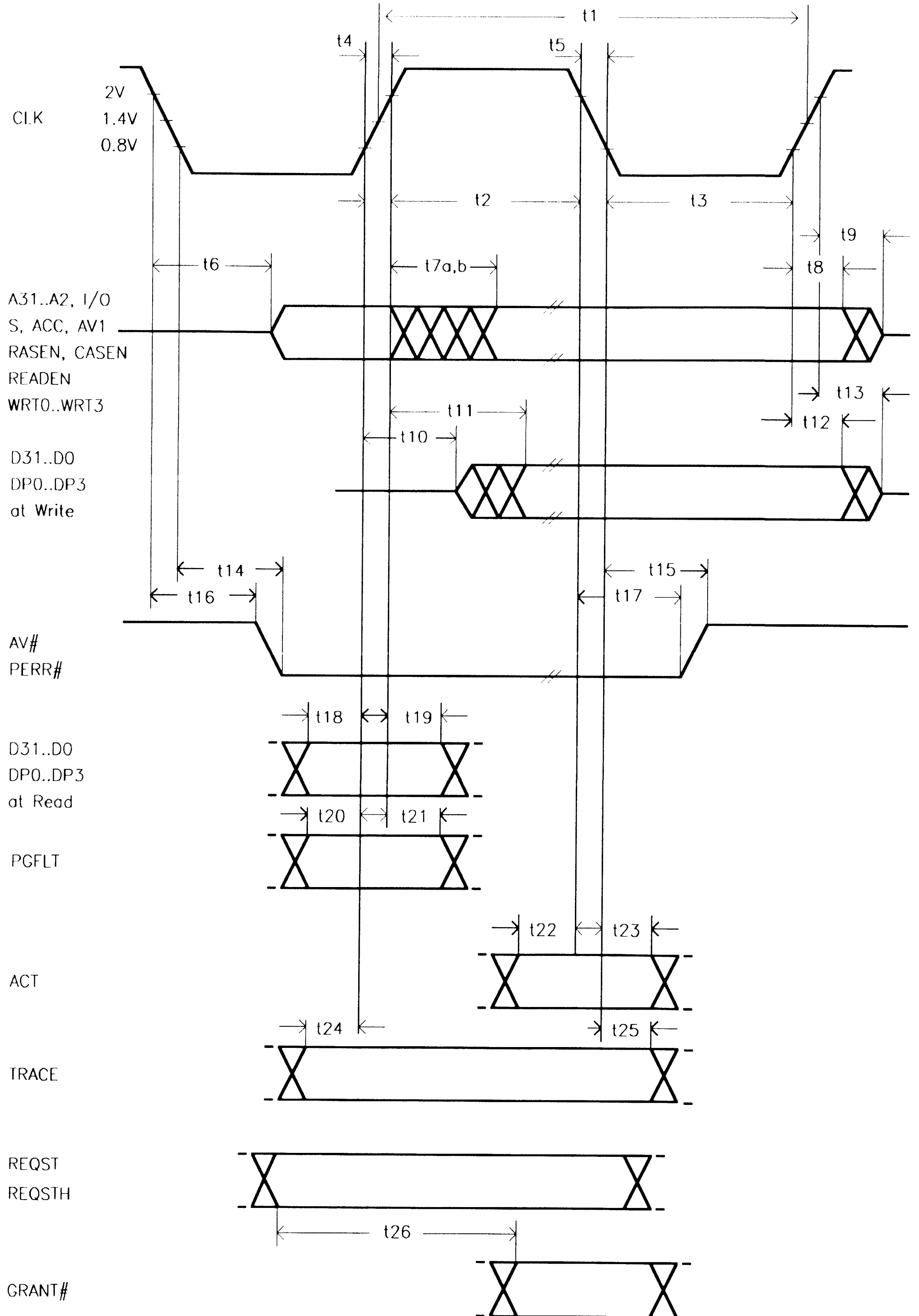
5.8 A.C. Characteristics

Preliminary

T_{CASE} = 0 to 85°C, V_{CC} = 5V ± 0.25V, C_L = 50 pF, unless otherwise specified

Symbol	Parameter	Min (ns)	Max (ns)	Notes
t1	CLK period	40		
t2	CLK high time	15		
t3	CLK low time	15		
t4	CLK rise time		4	
t5	CLK fall time		4	
t6	Address bus float hold time	5		
t7a	Address bus valid delay time (except READEN, RASEN)		25	(C_L=40pf)
t7b	Address bus valid delay time (READEN, RASEN)		27	(C_L=50pf)
t8	Address bus hold time	5		
t9	Address bus float delay time		23	
t10	Data bus float hold time	5		
t11	Write Data valid delay time		33	
t12	Write Data hold time	5		
t13	Data bus float delay time		23	
t14	AV#, PERR# low delay time		30	
t15	AV#, PERR# high delay time		29	
t16	AV#, PERR# high hold time	5		
t17	AV#, PERR# low hold time	5		
t18	Read data setup time	4		
t19	Read data hold time	6		
t20	PGFLT setup time	4		
t21	PGFLT hold time	8		
t22	ACT setup time	2		
t23	ACT hold time	5		
t24	TRACE setup time	2		
t25	TRACE hold time	9		
t26	REQST, REQSTH to GRANT# external delay time		t1 - 30	

5.8 A.C. Characteristics (continued)



6 Mechanical Data

6.1 Pin Configuration - View from Top Side

	R	P	N	M	L	K	J	H	G	F	E	D	C	B	A	
1	GNDe	VCCe	INT	DP2	DP0	WRT2	WRT0	D8	D10	D12	D14	D16	D18	GNDe	GNDe	1
2	GNDe	GNDe	RESET#	DP3	DP1	WRT3	WRT1	D9	D11	D13	D15	D17	D19	GNDe	VCCe	2
3	D7	PERR#	GNDe	VCCi	GNDi	VCCi	GNDi	VCCe	GNDe	VCCi	GNDi	VCCe	GNDe	D21	D22	3
4	D4	D5	VCCe										D20	D24	D25	4
5	D2	D3	D6										D23	D26	D27	5
6	D0	D1	GNDe										VCCe	D28	D29	6
7	NC	NC	VCCe										GNDe	D30	D31	7
8	ACC	CLK	GNDe										VCCe	A30	A31	8
9	S	I/O	VCCe										GNDe	A28	A29	9
10	AV1	A2	GNDe										VCCe	A26	A27	10
11	A3	A4	VCCe										GNDe	A24	A25	11
12	A5	A6	A9										VCCe	A22	A23	12
13	A7	A8	GNDe	VCCe	GNDi	VCCi	GNDi	VCCe	GNDe	VCCi	GNDi	VCCi	GNDe	A20	A21	13
14	VCCe	GNDe	NC	TRACE	CASEN	READEN	REQSTH	ACT	A11	A13	A15	A17	A19	GNDe	GNDe	14
15	GNDe	GNDe	NC	PGFLT	RASEN	AV#	REQST	GRANT#	A10	A12	A14	A16	A18	VCCe	GNDe	15
	R	P	N	M	L	K	J	H	G	F	E	D	C	B	A	

GNDi = GND internal (used for internal logic)

VCCi = VCC internal

GNDe = GND external (used for output drivers)

VCCe = VCC external

6.2 Pin Configuration - View from Pin Side

	A	B	C	D	E	F	G	H	J	K	L	M	N	P	R	
1	GNDe	GNDe	D18	D16	D14	D12	D10	D8	WRT0	WRT2	DP0	DP2	INT	VCCe	GNDe	1
2	VCCe	GNDe	D19	D17	D15	D13	D11	D9	WRT1	WRT3	DP1	DP3	RESET#	GNDe	GNDe	2
3	D22	D21	GNDe	VCCe	GNDi	VCCi	GNDe	VCCe	GNDi	VCCi	GNDi	VCCi	GNDe	PERR#	D7	3
4	D25	D24	D20										VCCe	D5	D4	4
5	D27	D26	D23										D6	D3	D2	5
6	D29	D28	VCCe										GNDe	D1	D0	6
7	D31	D30	GNDe										VCCe	NC	NC	7
8	A31	A30	VCCe										GNDe	CLK	ACC	8
9	A29	A28	GNDe										VCCe	I/O	S	9
10	A27	A26	VCCe										GNDe	A2	AV1	10
11	A25	A24	GNDe										VCCe	A4	A3	11
12	A23	A22	VCCe										A9	A6	A5	12
13	A21	A20	GNDe	VCCi	GNDi	VCCi	GNDe	VCCe	GNDi	VCCi	GNDi	VCCe	GNDe	A8	A7	13
14	GNDe	GNDe	A19	A17	A15	A13	A11	ACT	REQSTH	READEN	CASEN	TRACE	NC	GNDe	VCCe	14
15	GNDe	VCCe	A18	A16	A14	A12	A10	GRANT#	REQST	AV#	RASEN	PGFLT	NC	GNDe	GNDe	15
	A	B	C	D	E	F	G	H	J	K	L	M	N	P	R	

GNDi = GND internal (used for internal logic)

VCCi = VCC internal

GNDe = GND external (used for output drivers)

VCCe = VCC external

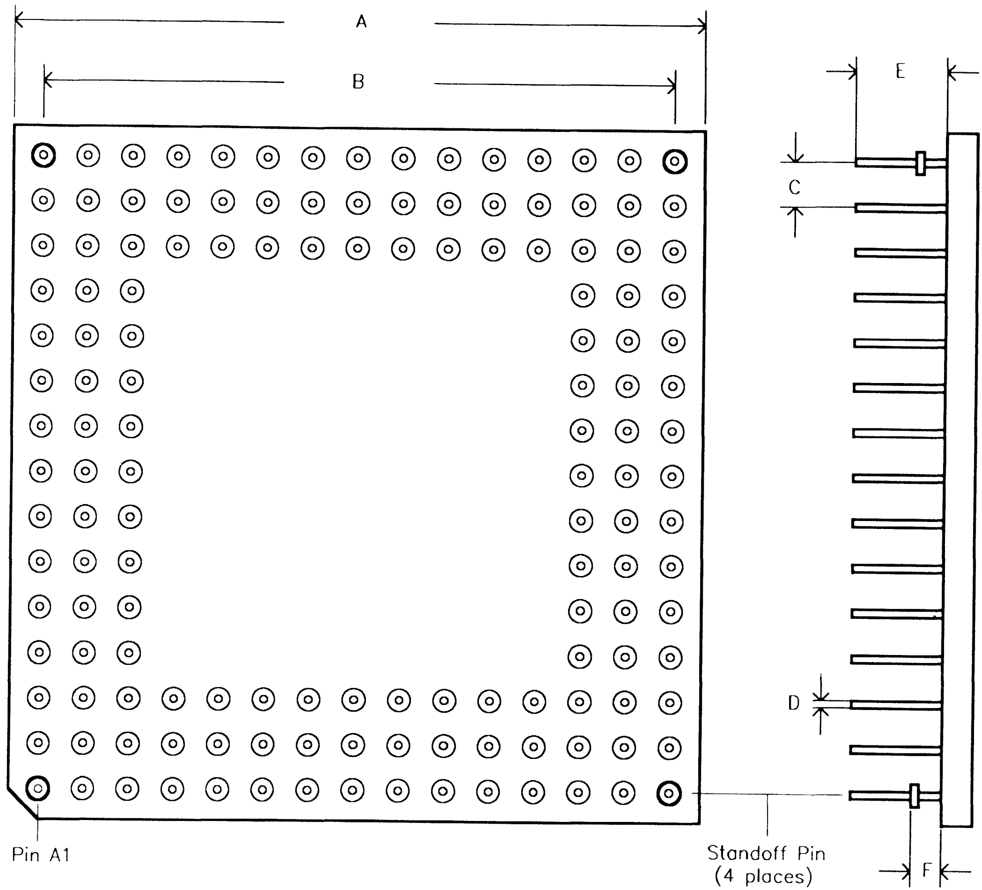
6.3 Pin Cross Reference by Pin Name

Signal	Location	Signal	Location	Signal	Location	Signal	Location
A2	P10	D0	R6	GNDe	A1	NC	R7
A3	R11	D1	P6	GNDe	A14	PERR#	P3
A4	P11	D2	R5	GNDe	A15	PGFLT	M15
A5	R12	D3	P5	GNDe	B1	RASEN	L15
A6	P12	D4	R4	GNDe	B2	READEN	K14
A7	R13	D5	P4	GNDe	B14	REQST	J15
A8	P13	D6	N5	GNDe	C3	REQSTH	J14
A9	N12	D7	R3	GNDe	C7	RESET#	N2
A10	G15	D8	H1	GNDe	C9	S	R9
A11	G14	D9	H2	GNDe	C11	TRACE	M14
A12	F15	D10	G1	GNDe	C13	VCCe	A2
A13	F14	D11	G2	GNDe	G3	VCCe	B15
A14	E15	D12	F1	GNDe	G13	VCCe	C6
A15	E14	D13	F2	GNDe	N3	VCCe	C8
A16	D15	D14	E1	GNDe	N6	VCCe	C10
A17	D14	D15	E2	GNDe	N8	VCCe	C12
A18	C15	D16	D1	GNDe	N10	VCCe	D3
A19	C14	D17	D2	GNDe	N13	VCCe	H3
A20	B13	D18	C1	GNDe	P2	VCCe	H13
A21	A13	D19	C2	GNDe	P14	VCCe	M13
A22	B12	D20	C4	GNDe	P15	VCCe	N4
A23	A12	D21	B3	GNDe	R1	VCCe	N7
A24	B11	D22	A3	GNDe	R2	VCCe	N9
A25	A11	D23	C5	GNDe	R15	VCCe	N11
A26	B10	D24	B4	GNDi	E3	VCCe	P1
A27	A10	D25	A4	GNDi	E13	VCCe	R14
A28	B9	D26	B5	GNDi	J3	VCCi	D13
A29	A9	D27	A5	GNDi	J13	VCCi	F3
A30	B8	D28	B6	GNDi	L3	VCCi	F13
A31	A8	D29	A6	GNDi	L13	VCCi	K3
ACC	R8	D30	B7	GRANT#	H15	VCCi	K13
ACT	H14	D31	A7	INT	N1	VCCi	M3
AV#	K15	DP0	L1	I/O	P9	WRT0	J1
AV1	R10	DP1	L2	NC	N14	WRT1	J2
CASEN	L14	DP2	M1	NC	N15	WRT2	K1
CLK	P8	DP3	M2	NC	P7	WRT3	K2

6.4 Pin Cross Reference by Location

Location	Signal	Location	Signal	Location	Signal	Location	Signal
A1	GNDe	C7	GNDe	H13	VCCe	N10	GNDe
A2	VCCe	C8	VCCe	H14	ACT	N11	VCCe
A3	D22	C9	GNDe	H15	GRANT#	N12	A9
A4	D25	C10	VCCe	J1	WRT0	N13	GNDe
A5	D27	C11	GNDe	J2	WRT1	N14	NC
A6	D29	C12	VCCe	J3	GNDi	N15	NC
A7	D31	C13	GNDe	J13	GNDi	P1	VCCe
A8	A31	C14	A19	J14	REQSTH	P2	GNDe
A9	A29	C15	A18	J15	REQST	P3	PERR#
A10	A27	D1	D16	K1	WRT2	P4	D5
A11	A25	D2	D17	K2	WRT3	P5	D3
A12	A23	D3	VCCe	K3	VCCi	P6	D1
A13	A21	D13	VCCi	K13	VCCi	P7	NC
A14	GNDe	D14	A17	K14	READEN	P8	CLK
A15	GNDe	D15	A16	K15	AV#	P9	I/O
B1	GNDe	E1	D14	L1	DP0	P10	A2
B2	GNDe	E2	D15	L2	DP1	P11	A4
B3	D21	E3	GNDi	L3	GNDi	P12	A6
B4	D24	E13	GNDi	L13	GNDi	P13	A8
B5	D26	E14	A15	L14	CASEN	P14	GNDe
B6	D28	E15	A14	L15	RASEN	P15	GNDe
B7	D30	F1	D12	M1	DP2	R1	GNDe
B8	A30	F2	D13	M2	DP3	R2	GNDe
B9	A28	F3	VCCi	M3	VCCi	R3	D7
B10	A26	F13	VCCi	M13	VCCe	R4	D4
B11	A24	F14	A13	M14	TRACE	R5	D2
B12	A22	F15	A12	M15	PGFLT	R6	D0
B13	A20	G1	D10	N1	INT	R7	NC
B14	GNDe	G2	D11	N2	RESET#	R8	ACC
B15	VCCe	G3	GNDe	N3	GNDe	R9	S
C1	D18	G13	GNDe	N4	VCCe	R10	AV1
C2	D19	G14	A11	N5	D6	R11	A3
C3	GNDe	G15	A10	N6	GNDe	R12	A5
C4	D20	H1	D8	N7	VCCe	R13	A7
C5	D23	H2	D9	N8	GNDe	R14	VCCe
C6	VCCe	H3	VCCe	N9	VCCe	R15	GNDe

6.5 Package Dimensions



Symbol	Inches	Millimeters
A	1.598 +- 0.006	40.6 +- 0.15
B	1.400 +- 0.012	35.56 +- 0.3
C	0.1 +- 0.012	2.54 +- 0.3
D	0.018 +- 0.002	0.46 +- 0.05
E	0.197 +- 0.008	5.0 +- 0.2
F	0.071 +- 0.006	1.8 +- 0.15