IGNITE™ Intellectual Property Reference Manual

Revision 1.0

PTSC
10989 Via Frontera
San Diego, CA 92127
1 (858) 674 5000 voice
1 (858) 674 5005 fax
www.ptsc.com

Printing Date: 2002 March 18

For company and product information, access www.ptsc.com. Patriot Scientific Corporation is publicly traded over the counter, symbol PTSC.

The technology discussed in this document may be covered by one or more of the following US patents:
5,440,749; 5,530,890; 5,604,915; 5,659,703; 5,784,584; 5,809,336. Other US and Foreign patents pending.

IMPORTANT NOTICE

Disclaimer

Patriot Scientific Corporation (PTSC) reserves the right to make changes to its products or specifications at any time, or to discontinue any product, without notice. PTSC advises its customers to obtain the latest product information available before designing-in or purchasing its products. PTSC assumes no responsibility for the use of any circuitry described other than the circuitry embodied in a PTSC product. PTSC makes no representations that the circuitry described herein is free from patent infringement or other rights of third parties, which may result from its use. No license is granted by implication or otherwise under any patent, patent rights or other rights, of PTSC. PTSC assumes no liability for any product designs, customer designs, design assistance, or use of its products.

Information within this document is subject to change without notice, but was believed to be accurate at the time of publication. No warranty of any kind, including but not limited to implied warranties of merchantability or fitness for a particular application, are stated or implied. PTSC and the author assume no responsibility for any errors or omissions, and disclaim responsibility for any consequences resulting from the use of the information included herein.

Critical Applications Policy

Some applications of semiconductor products involve potential risks of personal injury, death, severe property damage, or environmental damage. PTSC products are not authorized for use in such applications without a specific written agreement signed by the appropriate PTSC officer. Use of TSC products in such applications is understood to be fully at the risk of the customer.

Contents

Figures

Tables

PTSC

Purpose

This document describes the IGNITE processor. PTSC's IGNITE is a low-power, low-cost, stack-architecture processor targeted specifically for embedded applications. As a stack-architecture processor, the IGNITE processor is ideal for applications that must run Java™ at native speeds. These include laser printers, ignition controllers, network routers, personal digital assistants, set-top cable controllers, video games, pagers, cell phones, and many other applications. But since C++ is semantically similar to Java, the IGNITE processor also runs C and C++ efficiently, as well as stack-architecture languages such as Forth and Postscript.

This data book provides the information required to design products that use the IGNITE processor CPU.

Overview

The IGNITE processor is an implementation of the ShBoom™ microprocessor architecture. In its full implementation it is a highly integrated 32-bit RISC processor that executes at a peak performance of one instruction per CPU-clock cycle. The CPU is designed specifically for use in those embedded applications for which power consumption, CPU performance, and system cost are deciding selection factors.

The IGNITE processor CPU instruction set is hard-wired, allowing most instructions to execute in a single cycle, without the use of pipelines or superscalar architecture. A "flow-through" design allows the next instruction to start before the prior instruction completes, thus increasing performance.

The IGNITE processor contains 52 general-purpose registers, including 16 global data registers, an index register, a count register, a 16-deep addressable register/return stack, and an 18-deep operand stack. Both stacks contain an index register in the top element, are cached on chip, and, when required, automatically spill to and refill from external memory. The stacks minimize the data movement typical of register-based architectures, and also minimize memory accesses during procedure calls, parameter passing, and variable assignments. Additionally, the CPU contains a mode/status register, two stack pointers, and 7 locally addressed on-chip resource registers for I/O, control, configuration, and status.

Run Java at Native Speed: The stack architectures of the IGNITE processor and the Java Virtual Machine are very similar. This results in only a relatively simple byte code translator (20K) being required to produce executable native code from Java byte code, rather than a full Just-in-Time (JIT) compiler (200–400K) as is required for common processor architectures. The result is *much* faster initial execution of Java programs and significantly smaller memory requirements. Additionally, hundreds of kilobytes of memory are saved due to the reduced size of the translator itself.

Multiple Language Support: Most modern languages are implemented on a stack model. The features that allow the IGNITE processor to run Java efficiently apply similarly to other languages such as C, C++, Forth and Postscript.

Zero-Operand Architecture: Many RISC architectures waste valuable instruction space—often 15 bits or more per instruction—by specifying three possible operands for every instruction. Zero-operand (stack) architectures eliminate these operand bits, thus allowing much shorter instructions—typically one-fourth the size—and thus a higher instruction-execution bandwidth and smaller program size. Stacks also minimize register saves and loads within and across procedures, thus allowing shorter instruction sequences and faster-running code.

Fast, Simple Instructions: Instructions are less complex to decode and execute than those of conventional RISC processors, allowing the IGNITE processor to issue *and* complete instructions in a single CPU-clock cycle, as often as every CPU-clock cycle.

Four-Instruction Buffer: Using 8-bit opcodes, the CPU obtains up to four instructions from memory each time an instruction fetch or pre-fetch is performed. These instructions can be repeated without rereading them from memory. This maintains high performance when connected directly to DRAM, without the expense of a cache.

Local and Global Registers: Local and global registers minimize the number of accesses to data memory. The local-register stack automatically caches up to sixteen registers, and the operand stack up to eighteen registers. As stacks, any allocated data space efficiently nests and unnests across procedure calls. The sixteen global registers provide storage for shared data.

Posted Write: Decouples the processor from data writes to memory, allowing the processor to continue executing after a write is posted.

Fully Static Design: A fully static design allows running the clock from DC up to rated speed. Lower clock speeds can be used to drastically cut power consumption.

Hardware Debugging Support: Both breakpoint and single-step capability aid in debugging programs.

Floating-Point Support: Special instructions implement efficient single- and double-precision IEEE floating-point arithmetic.

Interrupt Controller: Supports up to eight prioritized levels with interrupt responses as fast as eight CPU-clock cycles.

Eight Bit Inputs and Eight Bit Outputs: I/O bits are available for CPU application use, thus reducing the requirement for external logic.

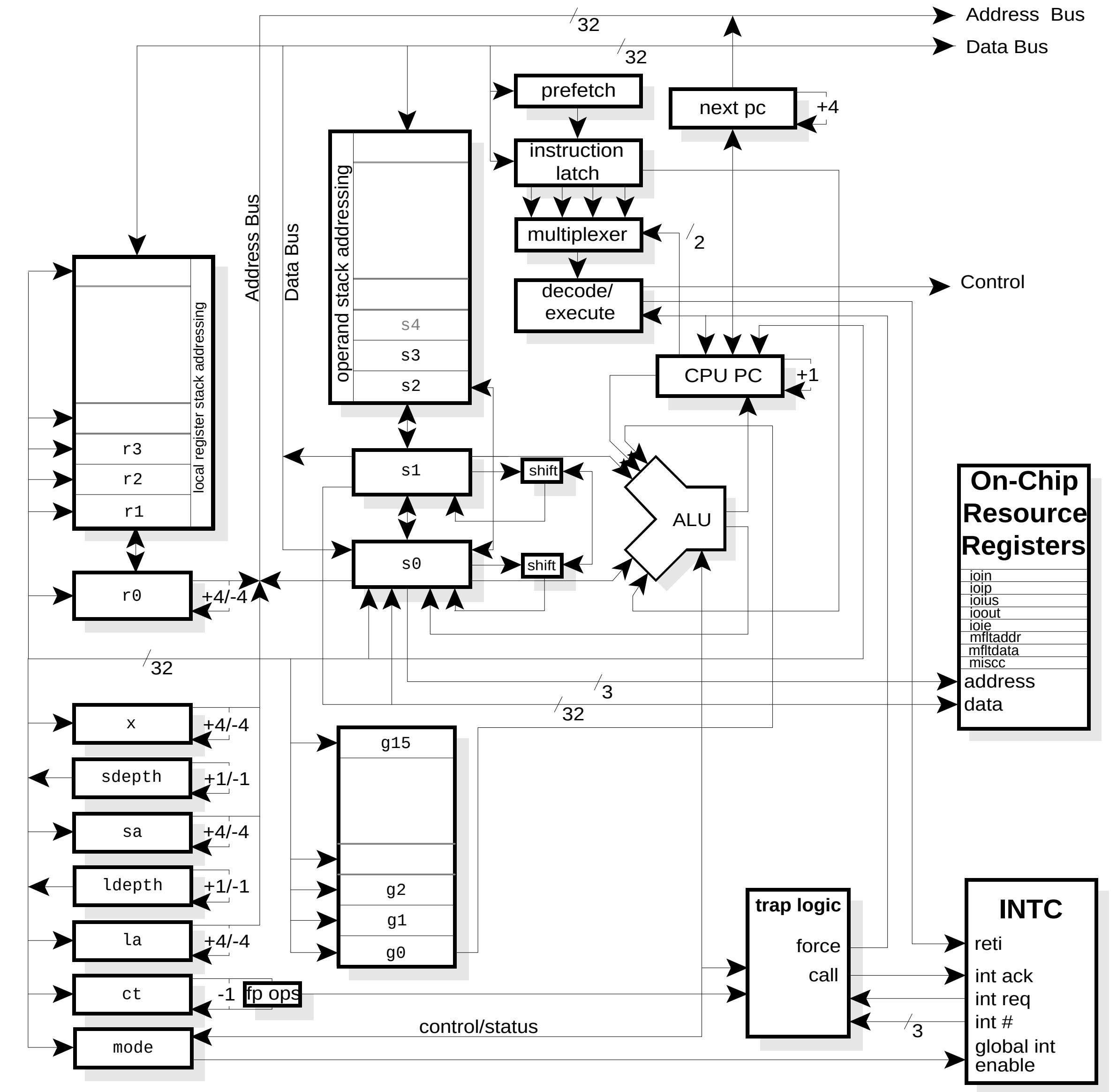


Figure 1 CPU Block Diagram

Microprocessor Unit

The CPU supports the ShBoom architectural philosophy of simplification and efficiency of use through its basic design in several interrelated ways.

Whereas most RISC processors use pipelines and superscalar execution to execute at high clock rates, the IGNITE processor uses neither. By having a simpler architecture, the IGNITE processor issues *and* completes most instructions in a single clock cycle. There are no pipelines to fill and none to flush during changes in program flow. Though more instructions are sometimes required to perform the same procedure in the IGNITE processor, the CPU operates at a higher clock frequency than other processors of similar silicon size and technology, thus giving comparable performance at significantly reduced cost.

A microprocessor's performance is often limited by how quickly it can be fed instructions from memory. The CPU reduces this bottleneck by using 8-bit instructions so that up to four instructions (an *instruction group*) can be obtained during each memory access. Each instruction typically takes one CPU-clock cycle to execute, thus requiring four CPU-clock cycles to execute the instruction group. Because a memory access can complete in four (or even fewer) CPU-clock cycles, the next instruction group can be available when execution of the previous group completes. This makes it possible to feed instructions to the processor at maximum instruction-execution bandwidth without the cost and complexity of an instruction cache.

The zero-operand (stack) architecture makes 8-bit instructions possible. The stack architecture eliminates the requirement to specify source and destination operands in every instruction. By not using opcode bits on every instruction for operand specification, a much greater bandwidth of functional operations—up to four times as high—is possible. Table 1 depicts an example IGNITE processor CPU instruction sequence that demonstrates twice the typical RISC CPU instruction bandwidth. The instruction sequence on the IGNITE processor requires one-half the instruction bits, and the uncached performance benefits from the resulting increase in instruction bandwidth.

```
 g5 = g1 - (g2 + 1) + g3 - (g4 * 2)

  Typical RISC MPU       IGNITE CPU

                         push   g1
                         push   g2
  add   #1,g2,g5         inc    #1

  sub   g1,g5,g5         sub

                         push   g3
  add   g5,g3,g5         add

                         push   g4
  shl   g4,#1,temp       shl    #1

                         sub
  sub   g5,temp,g5       pop    g5

  20 bytes               10 bytes

    Example of twice the instruction
bandwidth available on the IGNITE CPU
```

Table 1 Instruction Bandwidth Comparison

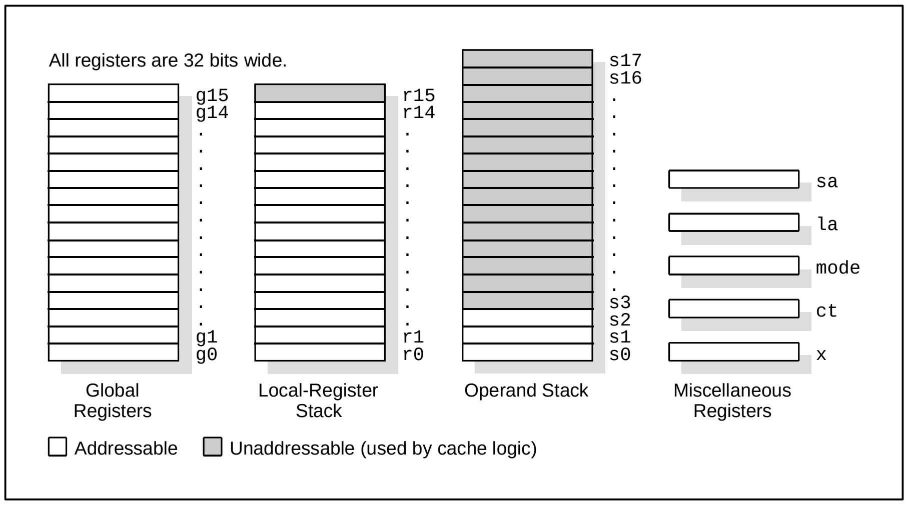


Figure 2 CPU Registers

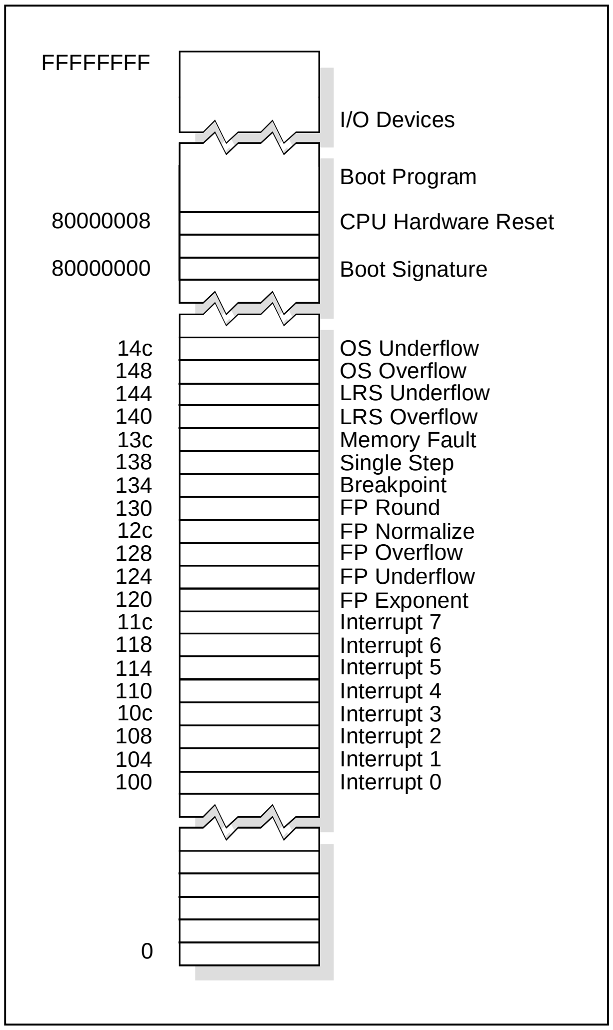


Figure 3 CPU Memory Map

Stack CPUs are thus simpler than register-based CPUs, and the IGNITE CPU has two hardware stacks to take advantage of this: the operand stack and the local-register stack. The simplicity is widespread and is reflected in the efficient ways stacks are used during execution.

The ALU processes data from primarily one source of inputs—the top of the operand stack. The ALU is also used for branch address calculations. Data bussing is thus greatly reduced and simplified. Intermediate results typically “stack up” to unlimited depth and are used directly when needed, rather than requiring specific register allocations and management. The stacks are individually cached and spill and refill automatically, eliminating software overhead for stack manipulation typical in other RISC processors. Function parameters are passed on, and consumed directly off of, the operand stack, eliminating the need for most stack frame management. When additional local storage is required, the local-register stack supplies registers that efficiently nest and unnest across functions. As stacks, the stack register spaces are only allocated for data actually stored, maximizing storage utilization and bus bandwidth when registers are spilled or refilled—unlike architectures using fixed-size register windows. Stacks speed context switches, such as interrupt servicing, because registers do not need to be explicitly saved before use—additional stack space is allocated as required. The stacks thus reduce the number of explicitly addressable registers otherwise required, and speed execution by reducing data location specification and movement. Stack storage is inherently local, so the global registers supply non-local register resources when required.

Eight-bit opcodes are too small to contain much associated data. Additional bytes are necessary for immediate values and branch offsets. However, variable-length instructions usually complicate decoding and complicate and lengthen the associated data access paths. To simplify the problem, byte literal data is taken only from the rightmost byte of the instruction group, regardless of the location of the byte literal opcode within the group. Similarly, branch offsets are taken as all bits to the right of the branch opcode, regardless of the opcode position. For 32-bit literal data, the data is taken from a subsequent memory cell. These design choices ensure that the required data is always right-justified for placement on the internal data busses, reducing interconnections and simplifying and speeding execution.

Since most instructions decode and execute in a single clock cycle, the same ALU that is used for data operations is also available, and is used, for branch address calculations. This eliminates an entire ALU often required for branch offset calculations.

Rather than consume the chip area for a single-cycle multiply-accumulate unit, the higher clock speed of the CPU reduces the execution time of conventional multi-cycle multiply and divide instructions. For efficiently multiplying by constants, a fast multiply instruction multiplies only by the specified number of bits.

Rather than consume the chip area for a barrel shifter, the counted bit-shift operation is “smart” to first shift by bytes, and then by bits, to minimize the cycles required. The shift operations can also shift double cells (64 bits), allowing bit-rotate instructions to be easily synthesized.

Although floating-point math is useful, and sometimes required, it is not heavily used in embedded applications. Rather than consume the chip area for a floating-point unit, CPU instructions to efficiently perform the most time-consuming aspects of basic IEEE floating-point math operations, in both single and double precision, are supplied. The operations use the “smart” shifter to reduce the cycles required.

Byte read and write operations are available, but cycling through individual bytes is slow when scanning for byte values. These types of operations are made more efficient by instructions that operate on all of the bytes within a cell at once.

Address Space

The CPU fully supports a linear four-gigabyte address space for all program and data operations.

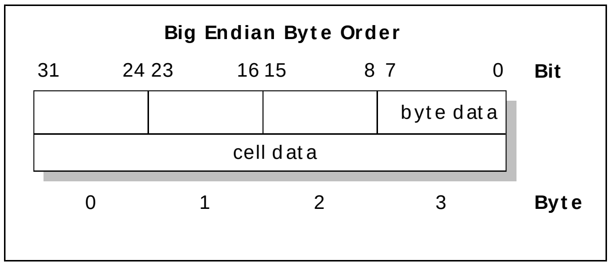


Figure 4 Byte Order

Several instructions or operations expect addresses aligned on four-byte (cell) boundaries. These addresses are referred to as *cell-aligned*. Only the upper 30 bits of the address are used to locate the data; the two least-significant address bits are ignored but appear externally. Within a cell, the high order byte is located at the low byte address. The next lower-order byte is at the next higher address, and so on. For example, the value 0x12345678 would exist at byte addresses in memory, from low to high address, as 12 34 56 78. See Figure 4.

Registers and Stacks

The register set contains 52 general-purpose registers, a mode/status register, and two stack pointers. See Figure 2. It also contains 7 local address-mapped on-chip resource registers used for I/O, configuration, and status.

The operand stack contains eighteen registers and operates as a push-down stack, with direct access to the top three registers (s0–s2). These registers and the remaining registers (s3–s17) operate together as a stack cache. Arithmetic, logical, and data-movement operations, as well as intermediate result processing, are performed on the operand stack. Parameters are passed to procedures and results are returned from procedures on the stack, without the requirement of building a stack frame or necessarily moving data between other registers and the frame. As a true stack, registers are allocated only as required, resulting in efficient use of available storage. The external operand stack is addressed by register sa.

The local-register stack contains sixteen registers and operates as a push-down stack with direct access to the first fifteen registers (r0–r14). Theses registers and the remaining register (r15) operate together as a stack cache. As a stack, they are used to hold subroutine return addresses and automatically nest local-register data. The external local-register stack is addressed by register la.

Both cached stacks automatically spill to memory and refill from memory, and can be arbitrarily deep. Additionally, s0 and r0 can be used for memory access. See *Stacks* and *Stack Caches*.

The use of stack-cached operand and local registers improve performance by eliminating the overhead required to save and restore context (when compared to processors with only global registers available). This allows for very efficient interrupt and subroutine processing.

In addition to the stacks are sixteen global registers and three other registers. The global registers (g0–g15) are used for data storage, and as operand storage for the CPU multiply and divide instructions (g0). Remaining are mode, which contains mode and status bits; x, which is an index register (in addition to s0 and r0); and ct, which is a loop counter and also participates in floating-point operations.

Programming Mode

For those familiar with the Java Virtual Machine, American National Standard Forth (ANS Forth), Postscript, or Hewlett-Packard calculators that use postfix notation, commonly known as Reverse Polish Notation (RPN), programming the IGNITE CPU will in many ways be very familiar.

A CPU architecture can be classified as to the number of operands specified within its instruction format. Typical 16-bit and 32-bit CISC and RISC CPUs are usually two- or three-operand architectures, whereas smaller microcontrollers are often one-operand architectures. In each instruction, two- and three-operand architectures specify a source and destination, or two sources and a destination, whereas one-operand architectures specify only one source and have an implicit destination, typically the accumulator. Architectures are also usually not pure. For example, one-operand architectures often have two-operand instructions to specify both a source and destination for data movement between registers.

The IGNITE CPU is a zero-operand architecture, known as a *stack computer*. Operand sources and destinations are assumed to be on the top of the operand stack, which is also the accumulator. An operation such as add uses both source operands from the top of the operand stack, adds them, and returns the result to the top of the operand stack, thus causing a net reduction of one in the operand stack depth. See Figure 5.

Most ALU operations behave similarly, using two source operands and returning one result operand to the operand stack. A few ALU operations use one source operand and return one result operand to the operand stack. Some ALU and other operations also require a non-stack register, and a very few do not use the operand stack at all.

Non-ALU operations are also similar. Loads (memory reads) either use an address on the operand stack or in a specified register, and place the retrieved data on the operand stack. Stores (memory writes) use either an address on the operand stack or in a register, and use data from the operand stack. Data movement operations push data from a register onto the operand stack, or pop data from the stack into a register.

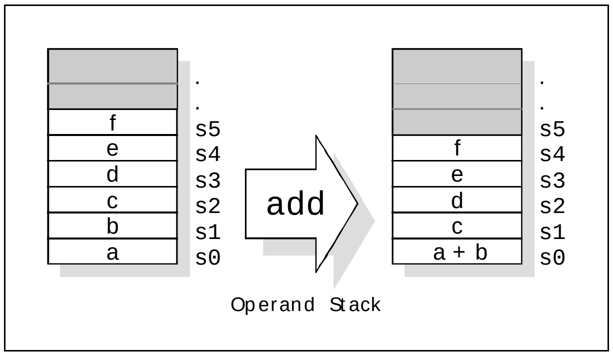


Figure 5 Add Execution Example

Once data is on the operand stack it can be used for any instruction that expects data there. The result of an add, for instance, can be left on the stack indefinitely, until used by a subsequent instruction. See Table 1. Instructions are also available to reorder the data in the top few cells of the operand stack so that prior results can be accessed when required. Data can also be removed from the operand stack and placed in local or global registers to minimize or eliminate later reordering of stack elements. Data can even be popped from the operand stack and restacked by pushing it onto the local-register stack.

Computations are usually most efficiently performed by executing the most deeply nested computations first, leaving the intermediate results on the operand stack, and then combining the intermediate results as the computation unnests. If the nesting of the computation is complex, or if the intermediate results are to be used some time later after other data would have been added to the operand stack, the intermediate results can be removed from the operand stack and stored in global or local registers.

Global registers are used directly and maintain their data indefinitely. Local registers are registers within the local-register stack cache and, as a stack, must first be allocated. Allocation can be performed by popping data from the operand stack and pushing it onto the local-register stack one cell at a time. It can also be preformed by allocating a block of uninitialized stack registers at one time; the uninitialized registers are then initialized by popping data, one cell at a time, into the registers in any order. The allocated local registers can be deallocated by pushing data onto the operand stack by popping it off of the local register stack one cell at a time, and then discarding from the operand stack the data that is not required. Alternatively, the allocated local registers can be deallocated by first saving any data required from the registers, and then deallocating a block of registers at one time. The method selected depends on the number of registers required and whether the data on the operand stack is in the required order.

Registers on both stacks are referenced relative to the tops of the stacks and are thus local in scope. What was accessible in r0, for example, after one cell has been push onto the local-register stack, is accessible as r1; the newly pushed value is accessible as r0.

Parameters are passed to and returned from subroutines on the operand stack. An unlimited number of parameters can be passed and returned in this manner. An unlimited number of local-register allocations can also be made. Parameters and allocated local registers thus conveniently nest and unnest across subroutines and program basic blocks.

Subroutine return addresses are pushed onto the local-register stack and thus appear as r0 on entry to the subroutine, with the previous r0 accessible as r1, and so on. As data is pushed onto the stacks and the available register space fills, registers are spilled to memory when required. Similarly, as data is removed from the stacks and the register space empties, the registers are refilled from memory as required. Thus from the program's perspective, the stack registers are always available.

Instruction Set Overview

Table 2 lists the CPU instructions; Table 35, page 66, and Table 36, page 67, list the mnemonics and opcodes. All instructions consist of eight bits, except for those that require immediate data. This allows up to four instructions (an instruction group) to be obtained on each instruction fetch, thus reducing memory-bandwidth requirements compared to typical RISC machines with 32-bit instructions. This characteristic also allows looping on an instruction group (a micro-loop) without additional instruction fetches from memory, further increasing efficiency. Instruction formats are depicted in Figure 6.

ARITHMETIC/SHIFT
ADD
ADD with carry
ADD ADDRESS
SUBTRACT
SUBTRACT with borrow
INCREMENT
DECREMENT
NEGATE
SIGN EXTEND BYTE
COMPARE
MAXIMUM
MULTIPLY SIGNED
MULTIPLY UNSIGNED
FAST MULTIPLY SIGNED
DIVIDE UNSIGNED
SHIFT LEFT/RIGHT
DOUBLE SHIFT LEFT/RIGHT
INVERT CARRY

MISCELLANEOUS
CACHE CONTROL
FRAME CONTROL
STACK DEPTH
NO OPERATION
ENABLE/DISABLE INTERRUPTS

CONTROL TRANSFER
BRANCH
BRANCH ON ZERO
BRANCH INDIRECT
CALL
CALL INDIRECT
DECREMENT AND BRANCH
SKIP
SKIP ON CONDITION
MICRO-LOOP
MICRO-LOOP ON CONDITION
RETURN
RETURN FROM INTERRUPT

FLOATING POINT
TEST EXPONENT
EXTRACT EXPONENT
EXTRACT SIGNIFICAND
REPLACE EXPONENT
DENORMALIZE
NORMALIZE RIGHT/LEFT
EXPONENT DIFFERENCE
ADD EXPONENTS
SUBTRACT EXPONENTS
ROUND

LOGICAL
AND
OR
XOR
NOT AND
TEST BYTES
EQUAL ZERO

DEBUGGING
STEP
BREAKPOINT

DATA MANAGEMENT
LOAD
STORE
STORE INDIRECT, pre-dec/post-inc
PUSH REGISTER/STACK
POP REGISTER/STACK
EXCHANGE
REVOLVE
SPLIT
REPLACE BYTE
PUSH LITERAL
STORE ON-CHIP RESOURCE
LOAD ON-CHIP RESOURCE

Table 2 CPU Instruction Set

add	add pc	adda	addc
and	cmp	dec #1	dec #4
dec ct,#1	divu	eqz	iand
inc #1	inc #4	mulfs	muls
mulu	mxm	neg	notc
or	sexb	shift	shiftd
shl #1	shl #8	shr #1	shr #8
shld #1	shrd #1	sub	subb
testb	xor		

Table 3 ALU Instructions

ALU Operations

Almost all ALU operations occur on the top of the operand stack in s0 and, if required, s1. A few operations also use g0, ct, or pc.

Only one ALU status bit, carry, is maintained and is stored in mode. Since there are no other ALU status bits, all other conditional operations are performed by testing s0 on the fly. eqz is used to reverse the zero/non-zero state of s0. Most arithmetic operations modify carry from the result produced out of bit 31 of s0. The instruction add pc is available to perform pc-relative data references. adda is available to perform address arithmetic without changing carry. Other operations modify carry as part of the result of the operation.

s0 and s1 can be used together for double-cell shifts, with s0 containing the more-significant cell and s1 the less-significant cell of the 64-bit value. Both single-cell and double-cell shifts transfer a bit between carry and bit 31 of s0. Code depicting single-cell rotates constructed from the double-cell shift is given in Table 4.

All ALU instruction opcodes are formatted as 8-bit values with no encoded fields.

```
; Rotate single cell left by specified number of bits
; ( n1 #bits -- n2 )

rotate_left::

        push    #0      ; space for bits
        xcg             ; get count
        shiftd
        or              ; combine parts
        ...

; Rotate single cell right by specified number of bits
; ( n1 #bits -- n2 )

rotate_right::

        push    #0      ; space for bits
        rev
        rev

        shl     #1      ; make a negative
        notc            ; sign magnitude
        shr     #1      ; number

        shiftd
        or
        ...
```

Table 4 Code example: Rotate

br	br []	bz	call
call []	dbr	mloop	mloopc
mloopn	mloopnc	mloopnn	mloopnz
mloopz	ret	reti	skip
skipc	skipn	skipnc	skipnn
skipnz	skipz		

Table 6 Branch, Loop and Skip Instructions

Offset Bits	Offset Range in Bytes
3	-16/+12
11	-4096/+4092
19	-1048576/+1048572
27	-268435456/+268435452

Note:
Encoded offset is in cells. Offset is added to the address of the beginning of the cell containing the branch to compute the destination address.

Table 5 CPU Branch Ranges

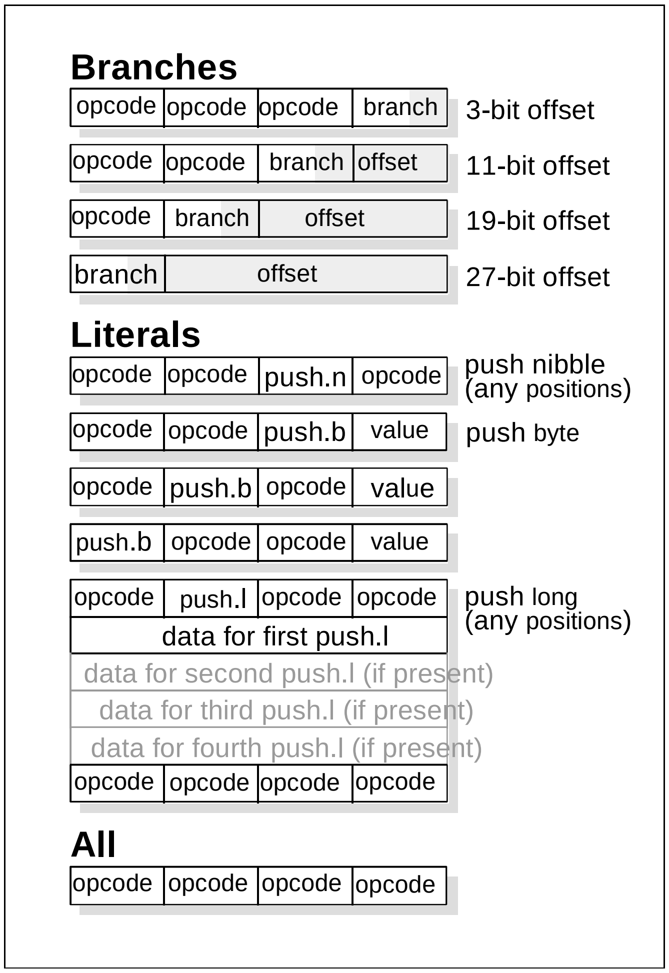

Branches
opcode opcode opcode branch
3-bit offset
opcode opcode branch offset
11-bit offset
opcode branch offset
19-bit offset
branch offset
27-bit offset
Literals
opcode opcode push.n opcode
push nibble (any positions)
opcode opcode push.b value
push byte
opcode push.b opcode value
push.b opcode opcode value
opcode push.l opcode opcode
push long (any positions)
data for first push.l
data for second push.l (if present)
data for third push.l (if present)
data for fourth push.l (if present)
opcode opcode opcode opcode
All
opcode opcode opcode opcode

Figure 6 CPU Instruction Format

Branches, Skips, and Loops

The instructions br, bz, call and dbr are variable-length. The three least-significant bits in the opcode and all of the bits in the current instruction group to the right of the opcode are used for the relative branch offset. See Figure 6 and Table 5. Branch destination addresses are cell-aligned to maximize the range of the offset and the number of instructions that are executed at the destination. If an offset is not of sufficient size for the branch to reach the destination, the branch must be moved to an instruction group where more offset bits are available, or a register indirect branch, br [] or call [], can be used. Register indirect branches use an absolute byte-aligned address from s0. The instruction add pc can be used if a computed pc-relative branch is required.

The mloop_ instructions are referred to as *micro-loops*. If specified, a condition is tested, and then ct is decremented. If a termination condition is not met, execution continues at the beginning of the current instruction group. Micro-loops are used to re-execute short instruction sequences without re-fetching the instructions from memory. See Table 11.

Other than branching on zero with bz, conditional branching is performed with the skip_ instructions. They terminate execution of the current instruction group and continue execution at the beginning of the next instruction group. They can be combined with the br, call, dbr, and ret (or other instructions) to create additional flow-of-control operations.

push.b	push.l	push.n

Table 7 Literal Instructions

Literals

To maximize opcode bandwidth, three sizes of literals are available. The data for four-bit (nibble) literals, with a range of -7 to +8, is encoded in the four least-significant bits of the opcode; the numbers are encoded as two's-complement values with the value 1000 binary decoded as +8. The data for eight-bit (byte) literals, with a range of 0–255, is located in the right-most byte of the instruction group, regardless of the position of the opcode within the instruction group. The data for 32-bit (long, or cell) literals is located in a cell following the instruction group in the instruction stream. Multiple push.l instructions in the same instruction group access consecutive cells immediately following the instruction group. See Figure 6.

pop ct	pop *gi*	pop *ri*	pop x
push ct	push *gi*	push *ri*	push *si*
push x			

Table 8 Data Movement Instructions

Data Movement

Register data is moved by first pushing the register onto the operand stack, and then popping it into the destination register. Memory data is moved similarly. See *Loads and Stores*, above.

The opcodes for the data-movement instructions that access *gi* and *ri* are 8-bit values with the register number encoded in the four least-significant bits. All other data-movement instruction opcodes are formatted as 8-bit values with no encoded fields.

ld [--r0]	ld [--x]	ld [r0++]	ld [r0]
ld [x++]	ld [x]	ld []	ld.b []
st [--r0]	st [--x]	st [r0++]	st [r0]
st [x++]	st [x]	st []	replb

Table 9 Load and Store Instructions

Loads and Stores

r0 and x support register-indirect addressing and also register-indirect addressing with predecrement by four or postincrement by four. These modes allow for efficient memory reference operations. Code depicting memory move and fill operations is given in Table 11.

Register indirect addressing can also be performed with the address in s0. Other addressing modes can be implemented using adda. Table 10 depicts the code for a complex memory reference operation.

The memory accesses depicted in the examples above are cell-aligned, with the two least-significant bits of the memory addresses ignored. Memory can also be read at byte addresses with ld.b [] and written at byte addresses using x and replb. Similar operations are available for 16-bit words. See *Byte and Word Operations*.

```
; addc [g0+g2+20],#8,[g0-g3-4]

        push    g0
        push    g2
        adda
        push.b  #20
        adda
        ld      []

        push.n  #8
        addc

        push    g0
        push    g3
        neg
        adda
        dec     #4
        st      []

; The carry into and out of addc is maintained.
```

Table 10 Code Example: Complex Addressing Mode

The CPU contains a one-level posted write. This allows the CPU to continue executing while the posted write is in progress and can significantly reduce execution time. Memory coherency is maintained by giving the posted write priority bus access over other CPU bus requests, thus writes are not indefinitely deferred. In the code examples in Table 11, the loop execution overhead is zero when using posted writes. Posted writes are enabled by setting mspwe in resource register miscc.

```
; Memory Move
; ( cell_source cell_dest cell_count -- )

move_cells::

        pop     ct      ; count
        pop     x       ; dest
        pop     lstack  ; source to r0

move_cell_loop::

        ld      [r0++]
        st      [x++]
        mloop   move_cell_loop

        push    lstack
        pop             ; discard source
        ...

; Memory Fill
; ( cell_dest cell_count cell_value  -- )

fill_cells::
        xcg
        pop     ct      ; count
        xcg
        pop     x       ; dest

fill_cells_loop::
        push            ; keep fill value
        st      [x++]
        mloop   fill_cells_loop

        pop             ; discard fill value
        ...
```

Table 11 Code Example: Memory Move and Fill

All load and store instruction opcodes are formatted as 8-bit values with no encoded fields.

lframe	pop	pop lstack	push
push lstack	rev	sframe	xcg

Table 12 Stack Data Management Instruction

Stack Data Management

Operand stack data is used from the top of the stack and is generally consumed when processed. This can require the use of instructions to duplicate, discard, or reorder the stack data. Data can also be moved to the local-register stack to place it temporarily out of the way, or to reverse its stack access order, or to place it in a local register for direct access. See the code examples in Table 11.

If more than a few stack data management instructions are required to access a given operand stack cell, performance usually improves by placing data in a local or global register. However, there is a finite supply of global registers, and local registers, at some point, spill to memory. Data should be maintained on the operand stack only while it is efficient to do so. In general, if the program requires frequent access to data in the operand stack deeper than s2, that data, or other more accessible data, should be placed in directly addressable registers to simplify access.

To use the local-register stack, data can be popped from the operand stack and pushed onto the local-register stack, or data can be popped from the local-register stack and pushed onto the operand stack. This mechanism is convenient to move a few cells when the resulting operand stack order is acceptable. When moving more data, or when the data order on the operand stack is not as desired, lframe can be used to allocate or deallocate the required local registers, and then the registers can be written and read directly. Using lframe also has the advantage of making the required local-register stack space available by spilling the stack as a continuous sequence of bus transactions, which minimizes the number of RAS cycles required when writing to DRAM. The instruction sframe behaves similarly to lframe, and is primarily used to discard a number of cells from the operand stack.

All stack data management instruction opcodes are formatted as 8-bit values with no encoded fields.

Stack Cache Management

Other than initialization, and possibly monitoring of overflow and underflow via the related traps, the stack caches do not require active management. Several instructions exist to efficiently manipulate the caches for context switching, status checking, and spill and refill scheduling.

The _depth instructions can be used to determine the number of cells in the SRAM part of the stack caches. This value can be used to discard the values currently in the cache, to later restore the cache depth with _cache, or to compute the total on-chip and external stack depth.

The _cache instructions can be used to ensure either that data is in the cache or that space for data exists in the cache, so that spills and refills occur at preferential times. This allows more control over the caching process and thus a greater degree of determinism during the program execution process. Scheduling stack spills and refills in this way can also improve performance by minimizing the RAS cycles required due to stack memory accesses.

The _frame instructions can be used to allocate a block of uninitialized register space at the top of the SRAM part of a stack, or to discard such a block of register space when no longer required. They, like the _cache instructions, can be used to group stack spills and refills to improve performance by minimizing the RAS cycles required due to stack memory accesses.

See *Stacks and Stack Caches* on page 15 for more information.

All stack cache management instruction opcodes are formatted as 8-bit values with no encoded fields.

`lcache`	`ldepth`	`pop la`	`pop sa`
`push la`	`push sa`	`scache`	`sdepth`

Table 13 Stack Cache Management Instruction

`copyb`	`ld.b []`	`replb`	`shl #8`
`shr #8`	`testb`		

Table 14 Byte and Word Operation Instructions

Byte and Word Operations

Bytes can be addressed and read from memory directly and can be addressed and written to memory with the code depicted in Table 15. Words (16-bit values) are handled similarly.

Instructions are available for manipulating bytes within cells. A byte can be replicated across a cell, the bytes within a cell can be tested for zero, and a cell can be shifted by left or right by one byte. Code examples depicting scanning for a specified byte, scanning for a null byte, and moving a null-terminated string in cell-sized units are given below.

All byte operation instruction opcodes are formatted as 8-bit values with no encoded fields.

```
; Byte store
; ( byte byte_addr -- )

byte_store::

        pop     x       ; address
        ld      [x]     ; get data
        replb           ; insert byte
        st      [x]     ; replace data
```

Table 15 Code Example: Byte Store

```
; Null character search
; ( cell_source -- )

null_search::

        pop     x       ; address

        push.n  #0
        pop     ct      ; a very long loop

        ; loop terminates when null found or after
        ; a long time if not found.

null_search_loop::

        ld      [x++]
        testb
        pop
        mloopnc         null_search_loop
        ...
```

Table 17 Code Example: Null Character Search

```
; Move cell-aligned null-terminated string
; ( cell_source cell_dest -- )

null_move::

        pop     x       ; destination
        pop     lstack  ; source

        push.n  #0
        pop     ct      ; a very long loop

null_move_loop::

        ld      [r0++]
        testb           ; check for zero
        st      [x++]
        mloopnc null_move_loop

        push    lstack
        pop             ; discard source
        ...
```

Table 16 Code Example: Null-Terminated String Move

```
; Byte search
; ( cell_source cell_count byte -- )

byte_search::

            xcg
            pop     ct      ; count

            xcg
            pop     x       ; source

            copyb

byte_search_loop::

            push            ; keep data pattern
            ld      [x++]
            xor

            testb
            pop

            skipnc
            dbr     byte_search_loop
            ; carry set if byte found

            pop             ; discard pattern
            ...
```

Table 18 Code Example: Byte Search

addexp	denorm	expdif	extexp
extsig	norml	normr	replexp
rnd	subexp	testexp	

Table 19 Floating Point Math Instruction

Floating-Point Math

The instructions above are used to implement efficient single- and double-precision IEEE floating-point software for basic math functions (+, -, *, /), and to aid in the development of floating-point library routines. The instructions perform primarily the normalization, denormalization, exponent arithmetic, rounding and detection of exceptional numbers and conditions that are otherwise execution-time-intensive when programmed conventionally. See *Floating-Point Math Support* on page 23.

All floating-point math instruction opcodes are formatted as 8-bit values with no encoded fields.

bkpt	step

Table 20 Debugging Instruction

Debugging Features

Each of these instructions signals an exception and traps to an application-supplied execution-monitoring program to assist in the debugging of programs. See *Debugging Support*.

Both debugging instruction opcodes are formatted as 8-bit values with no encoded fields.

ldo []	ldo.i []	sto []	sto.i []

Table 21 On-Chip Resources Instruction

On-Chip Resources

These instructions allow access to the on-chip peripherals, status registers, and configuration registers. All registers can be accessed with the ldo [] and sto [] instructions. The first six registers each contain eight bits, which are also bit addressable with ldo.i [] and sto.i []. See *On-Chip Resource Registers*.

All on-chip resource instruction opcodes are formatted as 8-bit values with no encoded fields.

All on-chip resource instruction opcodes are formatted as 8-bit values with no encoded fields.

di	ei	nop	pop mode
push mode	split		

Table 22 Miscellaneous Instructions

Miscellaneous

The disable- and enable-interrupt instructions are the only system control instructions; they are supplied to make interrupt processing more efficient. Other system control functions are performed by setting or clearing bits in mode, or in an on-chip resource register. The instruction split separates a 32-bit value into two cells, each containing 16 bits of the original value.

All miscellaneous instruction opcodes are formatted as 8-bit values with no encoded fields.

Stacks and Stack Caches

The stack caches optimize use of the stack register resources by minimizing the overhead required for the allocation and saving of registers during programmed or exceptional context switches (such as call subroutine execution and trap or interrupt servicing).

The local-register stack consists of an on-chip SRAM array that is addressed to behave as a conventional last-in, first-out queue. Local registers r0–r15 are addressed internally relative to the current top of stack. The registers r0–r14 are individually addressable and are always contiguously allocated and filled. If a register is accessed that is not in the cache, all the lower-ordinal registers are read in to ensure a contiguous data set.

The operand stack is constructed similarly, with the addition of two registers in front of the SRAM stack cache array to supply inputs to the ALU. These registers are designated s0 and s1, and the SRAM array is designated s2–s17. Only registers s0, s1 and s2 are individually addressable, but otherwise the operand stack behaves similarly to the local-register stack. Whereas the SRAM array, s2–s17, can become "empty" (see below), s0 and s1 are always considered to contain data.

The stack caches are designed to always allow the current operation to execute to completion before an implicit stack memory operation is required to occur. No instruction explicitly pushes or explicitly pops more than one cell from either stack (except for stack management instructions). Thus to allow execution to completion, the stack cache logic ensures that there is always one or more cells full and one or more cells empty in each stack cache (except immediately after reset, see *Stack Initialization*) before instruction execution. If, after the execution of an instruction, this is not the case on either stack, the corresponding stack cache is automatically spilled to memory or refilled from memory to reach this condition before the next instruction is allowed to execute. Similarly, the instructions _cache, _frame, pop sa, and pop la, which explicitly change the stack cache depth, execute to completion, and then ensure the above conditions exist.

Thus r15 or s17 can be filled by the execution of an instruction, but they are spilled before the next instruction executes. Similarly, r0 and s2 can be emptied by the execution of an instruction, but they are filled before the next instruction executes.

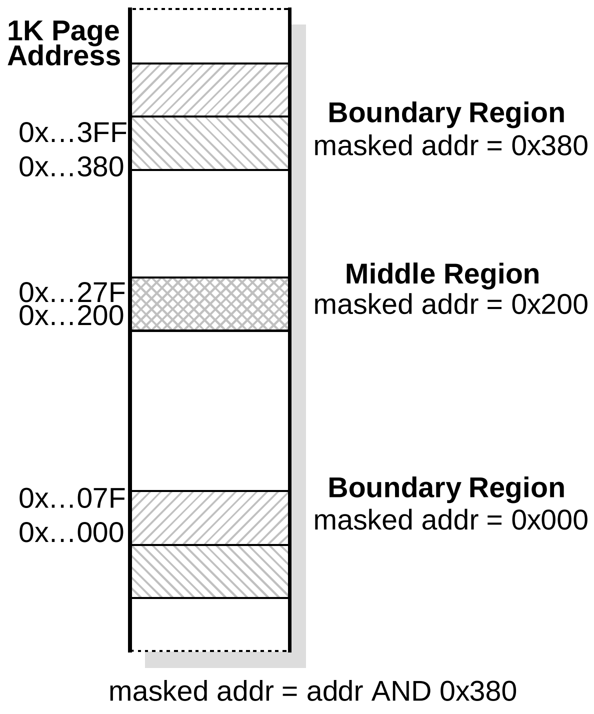


Figure 7 Stack Exception Region

The stacks can be arbitrarily deep. When a stack spills, data is written at the address in the stack pointer and then the stack pointer is decremented by four (postdecremented stack pointer). Conversely, when a stack refills, the stack pointer is incremented by four, and then data is read from memory (preincremented stack pointer). The stack pointer thus points to the next location to write and the stacks grow from higher to lower memory addresses. The stack pointer for the operand stack is sa, and the stack pointer for the local-register stack is la.

Since the stacks are dynamically allocated memory areas, some amount of planning or management is required to ensure the memory areas do not overflow or underflow. The simplest is to allocate a sufficiently large memory area so that overflow conditions won't occur. In this case, a correctly written program does not produce underflow. Alternatively, stack memory can be dynamically allocated or monitored through the use of stack-page exceptions.

Stack-Page Exceptions

Stack-page exceptions occur on any stack-cache memory access near the boundary of any 1024-byte memory page to allow overflow and underflow protection and stack memory management. To prevent thrashing stack-page exceptions near the margins of the page boundary areas, once a boundary area is accessed and the corresponding stack-page exception is signaled, the stack pointer must move to the middle region of the stack page before another stack-page exception can be signaled. See Figure 9.

Stack-page exceptions enable stack memory to be managed by allowing stack memory pages to be reallocated or relocated when the edges of the current stack page are approached. The boundary regions of the stack pages are located 32 cells from the ends of each page to allow even a _cache or _frame instruction to execute to completion and to allow for the corresponding stack cache to be emptied to memory. Using the stack-page exceptions requires that only 2 KB of addressable memory be allotted to each stack at any given time: the current stack page and the page near the most recently encroached boundary.

Each stack supports stack-page overflow and stack-page underflow exceptions. These exception conditions are tested against the memory address that is accessed when the corresponding stack spills or refills between the execution of instructions. mode contains bits that signal local-stack overflow, local-stack underflow, operand stack overflow and operand stack underflow, as well as the corresponding trap enable bits.

The stack-page exceptions have the highest priority of all of the traps. As this implies, it is important to consider carefully the stack effects of the stack trap handler code so that stack-page boundaries are not be violated during its execution. Additionally, a memory fault must not occur during a stack page access. The stack page exceptions are intended to be used to ensure valid stack pages can always be accessed without memory faults.

Since stack-page exceptions can occur on any stack spill or refill, usage of certain stack-cache management instructions (_depth and _cache) must be modified to ensure the expected result. A stack-page exception can occur after the stack-cache management instruction and thus modify the cache state. To prevent this, the instruction must complete without a stack spill or refill that would cause a stack-page exception. This can be accomplished by either causing a similar stack effect prior to executing the instruction, or by executing the instruction twice in immediate sequence. See the supplied stack management code examples in this section.

```
init_stacks::

        ; Create a stack area below xx_base in
        ; memory. One cell is read in to initialize s2/r0.

        push.l   #os_base-8   ; adjust for post incr and
                              ; one refill
        pop      sa           ; read os_base-4
        ; s0 and s1 are uninitialized

        push.l   #ls_base-8   ; adjust for post incr and
                              ; one refill
        pop      la           ; read ls_base-4
```

Table 23 Code Example: Stack Initialization

Stack Initialization

After CPU reset both of the CPU stacks should be considered uninitialized until the corresponding stack pointers are loaded, and this should be one of the first operations performed by the CPU.

After a reset, the stacks are abnormally empty. That is, r0 and s2 have not been allocated, and are allocated on the first push operation to, or stack pointer initialization of, the corresponding stack. However, popping the pushed cell causes that stack to be empty and require a refill. The first pushed cell should therefore be left on that stack, or the corresponding stack pointer should be initialized, before the stack is used further. See Table 23.

Stack Depth

The total number of cells on each stack can readily be determined by adding the number of cells that have spilled to memory and the number of cells in the on-chip caches. See Table 24.

```
; Operand stack depth

os_depth::

        push.n   #-2
        scache
        pop              ; ensure 3 spaces available

        .quad    2       ; keep up to push sa
        sdepth           ; uninterruptable

        push     sa
        push.l   #-(os_base-4)
        add              ; compute memory used

        shr      #1
        shr      #1      ; convert to cells

        add              ; total on-chip & off
        ...

ls_depth:
        push.n   #-2
        scache
        pop              ; ensure 3 spaces available

        .quad    2       ; keep up to push la
        ldepth           ; uninterruptable

        push     la
        push.l   #-(ls_base-4)
        add              ; compute memory used

        shr      #1
        shr      #1      ; convert to cells

        add              ; total on-chip & off
        ...
```

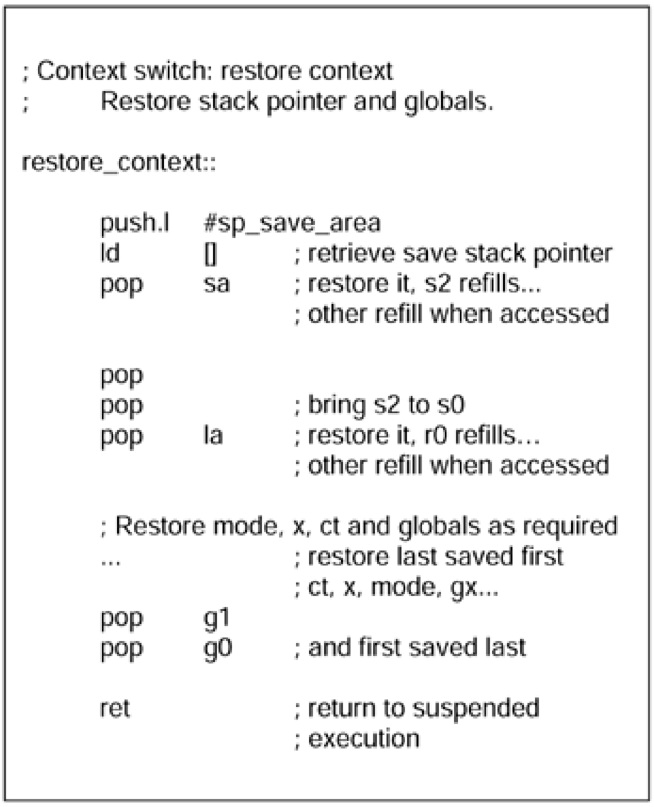
Table 24 Code Example: Stack Depth

```
; Context switch: save context
;     Save off any gloabls required and flush stacks

save_context::

      ; Save globals and mode, x, ct as required
      push     g0
      push     g1
      ...               ; save any others required
                        ; gx, mode, x, ct...

; Flush stacks to memory

      ; add one cell to local-register stack so on-chip
      ; part can spill.
      push.b   #-14     ; count for _cache
      pop      lstack

      push     r0       ; count for lcache

      ; ensure no interrupts between flush and la read
      .quad    4
      push              ; ensure space for lcache value
      pop               ; w/o overflow trap occurring
      lcache            ; write out spillable area
      push     la       ; save pointer

      ; add three cells to stack so on-chip part can spill
      push
      push
      push     r0       ; count for scache

      ; ensure no interrupts between flush and sa read
      .quad    4
      push              ; ensure space for scache value
      pop               ; w/o overflow trap occurring
      scache            ; write out all of spillable area
      push     sa

      push.l   #sp_save_area
      st       []       ; save off stack pointer

; Now load new context and continue
      ...
```

Table 25 Code Example: Save Context

Stack Flush and Restore

When performing a context switch, it is necessary to spill the data in the stack caches to memory so that the stack caches can be reloaded for the new context.



```
; Context switch: restore context
;         Restore stack pointer and globals.

restore_context::

          push.l    #sp_save_area
          ld        []          ; retrieve save stack pointer
          pop       sa          ; restore it, s2 refills...
                                ; other refill when accessed

          pop
          pop                   ; bring s2 to s0
          pop       la          ; restore it, r0 refills...
                                ; other refill when accessed

          ; Restore mode, x, ct and globals as required
          ...                   ; restore last saved first
                                ; ct, x, mode, gx...
          pop       g1
          pop       g0          ; and first saved last

          ret                   ; return to suspended
                                ; execution
```

Table 26 Code Example: Restore Context

Attention must be given to ensure that the parts of the
stack caches that are always maintained on-chip, r0 and s0–s2, are forced into the spillable area of the stack caches so that they can be written to memory. Code examples are given for context switches that include flushing and restoring the caches in Table 25 and Table 26, respectively.

Exceptions and Trapping

Exception handling is precise and is managed by trapping to executable-code vectors in low memory. Each 32-bit vector location can contain up to four instructions. This allows servicing the trap within those four instructions or branching to a longer trap routine. Traps are prioritized and nested to ensure proper handling. The trap names and executable vector locations are shown in Figure 3.

Stack Depth Change		
Operand Stack	Local-Register Stack	Traps
+n	0	Operand Stack Overflow
–n	0	Operand Stack Underflow
0	+1	Local Stack Overflow
0	–1	Local Stack Underflow
+1	-n	Local Stack Underflow Operand Stack Overflow Local Stack Underflow and Operand Stack Overflow
–1	+n	Local Stack Overflow Operand Stack Underflow Local Stack Overflow and Operand Stack Underflow
–1	–n	Local Stack Underflow Operand Stack Underflow Local Stack Underflow and Operand Stack Underflow
Notes: 1. +n > 0, –n < 0 2. If the instruction reads or writes memory or if a posted write is in progress, a memory fault can also occur. 3. If the instruction is single-stepped, a single-step trap also occurs. 4. If any trap occurs, a local-register stack overflow could also occur.		

Table 27 Traps Dependent on System State

An exception is said to be signaled when the defined conditions exist to cause the exception. If the trap is enabled, the trap is then processed. Traps are processed by the trap logic, which causes a call subroutine to the associated executable-code-vector address. When multiple traps occur concurrently, the lowest-priority trap is processed first, but before the executable-code vector is executed, the next-higher-priority trap is processed, and so on, until the highest-priority trap is processed. The highest-priority trap's executable-code vector then executes. The nested executable-code-vector return addresses unnest as each trap handler executes ret, thus producing the prioritized trap executions.

Interrupts are disabled during trap processing and nesting, until an instruction that begins in byte one of an instruction group is executed. Interrupts do not nest with the traps since their request state is maintained in the INTC registers.

Table 28 lists the priorities of each trap. Traps that can occur explicitly due to the data processed or instruction executed are listed in Table 29. Traps that can occur due to the current state of the system, concurrently with the traps in Table 29, are listed in Table 27.

Priority	Traps
1 (highest)	local-register stack overflow
2	operand stack overflow
3	local-register stack underflow
4	operand stack underflow
5	memory fault
6	floating-point exponent floating-point underflow floating-point overflow floating-point round
7	floating-point normalize
8	breakpoint
9 (lowest)	single step

Table 28 Trap Priorities

Instruction	Trap Combinations
addexp	Floating Point Underflow, Floating Point Overflow
bkpt	Breakpoint
denorm	Floating Point Normalize
norml	Floating Point Underflow, Floating Point Normalize, Floating Point Underflow and Floating Point Normalize
normr	Floating Point Overflow, Floating Point Normalize, Floating Point Overflow and Floating Point Normalize
rnd	Floating Point Round
step	Single Step
subexp	Floating Point Underflow, Floating Point Overflow
testexp	Floating Point Exponent

Table 29 Traps Independent of System State

Floating-Point Math Support

The CPU supports single-precision (32-bit) and double-precision (64-bit) IEEE floating-point math software. Rather than a floating-point unit and the silicon area it would require, the CPU contains instructions to perform most of the time-consuming operations required when programming basic floating-point math operations. Existing integer math operations are used to supply the core add, subtract, multiply, and divide functions, while special instructions are used to efficiently manipulate the exponents and detect exception conditions. Additionally, a three-bit extension to the top one or two stack cells (depending on the precision) is used to aid in rounding and to supply the required precision and exception signaling operations.

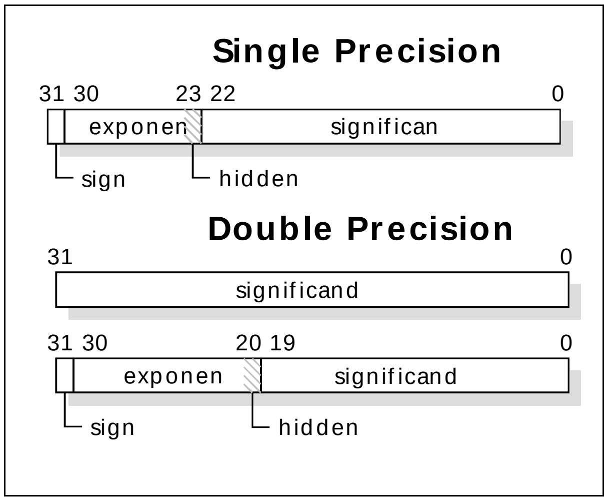


Figure 8 Floating-Point Number Formats

Data Formats

Though single- and double-precision IEEE formats are supported, from the perspective of the CPU, only 32-bit values are manipulated at any one time (except for double shifting). See Figure 8. The CPU instructions directly support the normalized data formats depicted. The related denormalized formats are detected by testexp and fully supportable in software.

Status and Control Bits

mode contains 13 bits that set floating-point precision, rounding mode, exception signals, and trap enables. See Figure 9.

cleared by:
`testexp` `replexp`

shifted into by:
`denorm` `normr` `shift` `shiftd`
`shr #1` `shr #8` `shrd #1`

shifted out of by:
`norml`

tested by:
`rnd`

read by:
`push mode`

written by:
`pop mode`

Table 30 GRS Extension Bit Manipulation Instructions

GRS Extension Bits

To maintain the precision required by the IEEE standard, more significand bits are required than are held in the IEEE format numbers. These extra bits are used to hold bits that have been shifted out of the right of the significand. They are used to maintain additional precision, to determine if any precision has been lost during processing, and to determine whether rounding should occur. The three bits appear in mode so they can be saved, restored and manipulated. Individually, the bits are named guard_bit, round_bit and sticky_bit. Several instructions manipulate or modify the bits. See Table 30.

When denorm and normr shift bits into the GRS extension, the source of the bits is always the least-significant bits of the significand. In single-precision mode the GRS extension bits are taken from s0, and in double-precision mode the bits are taken from s1. For conventional right shifts, the GRS extension bits always come from the least significant bits of the shift (i.e., s0 if a single shift and s1 if a double shift). The instruction norml is the only instruction to shift bits out of the GRS extension; it shifts into s0 in single-precision mode and into s1 in double-precision mode. Conventional left shifts always shift in zeros and do not affect the GRS extension bits.

Rounding

The GRS extension maintains three extra bits of precision while producing a floating-point result. These bits are used to decide how to round the result to fit the destination format. If one views the bits as if they were just to the right of the binary point, then guard_bit has a position value of one-half, round_bit has a positional value of one-quarter, and sticky_bit has a positional value of one-eighth. The rounding operation selected by fp_round_mode uses the GRS extension bits and the sign bit of ct to determine how rounding occurs. If guard_bit is zero the value of GRS extension is below one-half. If guard_bit is one the value of GRS extension is one-half or greater. Since the GRS extension bits are not part of the destination format they are discarded when the operation is complete. This information is the basis for the operation of the instruction rnd.

Sign of ct	G	R	S	Action
Round to nearest or even				
x	0	x	x	do nothing
x	1	0	0	increment s0, clear bit 0 of s0
x	1	any 1		increment s0
Round toward negative infinity				
0	x	x	x	do nothing
1	0	0	0	do nothing
1	any 1			increment s0
Round toward positive infinity				
0	0	0	0	do nothing
0	any 1			increment s0
1	x	x	x	do nothing
Round toward zero				
x	x	x	x	do nothing

Table 31 Rounding Mode Action

```
; Floating-Point Multiply
; ( r1 r2 -- product )
        ...
        testexp
        addexp

        pop     ct          ; save sign & exp sum

; A 24-bit x 24-bit multiply makes a 47 to 48-bit product,
; leaving 16-bits in the high cell. If we multiply 32-bit x
;  24-bit we get a 56-bit product with 24-bits in the high
; part, which is what we want.

; make into a 32-bit multiplier
        shl     #8
        pop     g0

        shl     #1
        push.n  #0

        mulu
        xcg
        pop                 ; discard low part

        normr
        rnd
        normr

        push    ct
        replexp
        ...
```

Table 32 Code Example: Floating-Point Multiply

Most rounding adjustments by rnd involve doing nothing or incrementing s0. Whether this is rounding down or rounding up depends on the sign of the floating-point result that is in ct. If the GRS extension bits are non-zero, then doing nothing has the effect of "rounding down" if the result is positive, and "rounding up" if the result is negative. Similarly, incrementing the result has the effect of "rounding up" if the result is positive and "rounding down" if the result is negative. If the GRS extension bits are zero then the result was exact and rounding is not required. See Table 31.

In practice, the significand (or the lower cell of a double-precision significand) is in s0, and the sign and exponent are in ct. carry is set if the increment from rnd carried out of bit 31 of s0; otherwise, carry is cleared. This allows carry to be propagated into the upper cell of a double-precision significand.

Exceptions

To speed processing, exception conditions detected by the floating-point instructions set exception signaling bits in mode and, if enabled, trap. The following traps are supported:

•	Exponent	signaled from testexp
•	Underflow	signaled from norml, addexp, subexp
•	Overflow	signaled from normr, addexp, subexp
•	Normalize	signaled from denorm, norml, normr
•	Rounded	signaled from rnd

Exceptions are prioritized when the instruction completes and are processed with any other system exceptions or traps that occur concurrently. See *Exceptions and Trapping*.

- Exponent Trap: Detects special-case exponents. If the tested exponent is all zeros or all ones, carry is set and the exception is signaled. Setting carry allows testing the result without processing a trap.
- Underflow Trap: Detects exponents that have become too small due to calculations or decrementing while shifting.
- Overflow Trap: Detects exponents that have become too large due to calculations or incrementing while shifting.
- Normalize Exception: Detects bits lost due to shifting into the GRS extension. The exception condition is tested at the end of instruction execution and is signaled if any of the bits in the GRS extension are set. Testing at this time allows normal right shifts to be used to set the GRS extension bits for later floating-point instructions to test and signal.
- Rounded Exception: Detects a change in bit zero of s0 due to rounding.

Hardware Debugging Support

The CPU contains a breakpoint instruction, bkpt, and a single-step instruction, step. The instruction bkpt executes the breakpoint trap and supplies the address of the bkpt opcode to the trap handler. This allows execution at full processor speed up to the breakpoint, and then execution in a program-controlled manner following the breakpoint. step executes the instruction at the supplied address, and then executes the single-step trap. The single-step trap can efficiently monitor execution on an instruction-by-instruction basis.

Breakpoint

The instruction bkpt performs an operation similar to a call subroutine to address 0x134, except that the return address is the address of the bkpt opcode. This behavior is required because, due to the instruction push.l, the address of a call subroutine cannot always be determined from its return address.

Commonly, bkpt is used to temporarily replace an instruction in an application at a point of interest for debugging. The trap handler for bkpt typically restores the original instruction, displays information for the user, and waits for a command. Or, the trap handler could be implemented as a conditional breakpoint to check for a termination condition (such as a register value or the number of executions of this particular breakpoint), continuing execution of the application until the condition is met. The advantage of bkpt over step is that the applications executes at full speed between breakpoints.

Single-Step

The instruction step is used to execute an application program one instruction at a time. It acts much like a return from subroutine, except that after executing one instruction at the return address, a trap to address 0x138 occurs. The return address from the trap is the address of the next instruction. The trap handler for step typically displays information for the user, and waits for a command. Or, the trap handler could instead check for a termination condition (such as a register value or the number of executions of this particular location), continuing execution of the application until the condition is met.

Step is processed and prioritized similarly to the other exception traps. This means that all traps execute before the step trap. The result is that step cannot directly single-step through the program code of other trap handlers. The instruction step is normally considered to be below the

```
; Memory-fault trap handler

memflt_handler::

        push    mode
        di

        ; Get data (if any) and fault address.

        push.l  #mfltdata   ; must be read first
        ldo     []
        push.l  #mfltaddr   ; must be read last
        ldo     []

; Now go and get the faulted page from disk
;  into memory, update the mapping SRAM, etc.
; ( mode data addr -- mode data addr )
        ...

; If memory fault occurred while attempting a
;  posted write, perform the write in the handler.

        ; check if fault was read or write
        push    s2                  ; duplicate mode
        push.l  #mflt_write
        and

        bz      discard_location    ; write fault?

        push.l  #miscc
        ldo     []

        push.b  #mspwe
        and                         ; posted write?

        .quad   3
        skipz   stack,discard_location
        st      []                  ; complete it
        push                        ; maintain 2 items

discard_location::

        pop                         ;discard "address"
        pop                         ; discard "data"

        ; Reset exception-signal bit.

        push.l  #mflt_exc_sig
        iand
        pop     mode

; For non-posted-write faults, the load/store/pre
;-fetch retries on return.

        ret
```

Table 33 Code example: Memory Fault Service Routine

operating-system level, thus operating-system functions such as stack-page traps must execute without its intervention.

Higher-priority trap handlers can be single-stepped by re-prioritizing them in software. Rather than directly executing a higher-priority trap handler from the corresponding executable trap vector, the vector would branch to code to rearrange the return addresses on the return stack to change the resulting execution sequence of the trap handlers. Various housekeeping tasks must also be performed, and the various handlers must ensure that the stack memory area boundaries are not violated by the re-prioritized handlers.

Register mode

mode contains a variety of bits that indicate the status and execution options of the CPU. Except as noted, all bits are writable. The register is shown in Figure 9.

mflt_write
After a memory-fault exception is signaled, indicates that the fault occurred due to a memory write.

guard_bit
The most-significant bit of a 3-bit extension below the least-significant bit of s0 (s1, if fp_precision is set) that is used to aid in rounding floating-point numbers.

round_bit
The middle bit of a 3-bit extension below the least-significant bit of s0 (s1, if fp_precision is set) that is used to aid in rounding floating-point numbers.

sticky_bit
The least-significant bit of a 3-bit extension below the least-significant bit of s0 (s1, if fp_precision is set) that is used to aid in rounding floating-point numbers. Once set due to shifting or writing the bit directly, the bit stays set even though zero bits are shifted right through it, until it is explicitly cleared or written to zero.

mflt_trap_en
If set, enables memory-fault traps.

mflt_exc_sig
Set if a memory fault is detected.

ls_boundary
Set if ls_ovf_exc_sig or ls_unf_exc_sig becomes set as the result of a stack spill or refill. Cleared when the address in la, as the result of a stack spill or refill, has entered the middle region of a 1024-byte memory page, and when la is written. Used by the local-register stack trap logic to prevent unnecessary stack overflow and underflow traps when repeated local-register stack spills and refills occur near a 1024-byte memory page boundary. Not writable.

ls_unf_trap_en
If set, enables a local-register stack underflow trap to occur after a local-register stack underflow exception is signaled.

ls_unf_exc_sig
Set if a local-register stack refill occurs, ls_boundary is clear, and the accessed memory address is in the last thirty-two cells of a 1024-byte memory page.

ls_ovf_trap_en
If set, enables a local-register stack overflow trap to occur after a local-register stack overflow exception is signaled.

ls_ovf_exc_sig
Set if a local-register stack spill occurs, ls_boundary is clear, and the accessed memory address is in the first thirty-two cells of a 1024-byte memory page.

os_boundary
Set if os_ovf_exc_sig or os_unf_exc_sig becomes set as the result of a stack spill or refill. Cleared when the address in sa, as the result of a stack spill or refill, has entered the middle region of a 1024-byte memory page, and when sa is written. Used by the operand stack trap logic to prevent unnecessary stack overflow and underflow traps when repeated operand stack spills and refills occur near a 1024-byte memory page boundary. Not writable.

os_unf_trap_en

If set, enables an operand stack underflow trap to occur after an operand stack underflow exception is signaled.

os_unf_exc_sig

Set if an operand stack refill occurs, os_boundary is clear, and the accessed memory address is in the last thirty-two cells of a 1024-byte memory page.

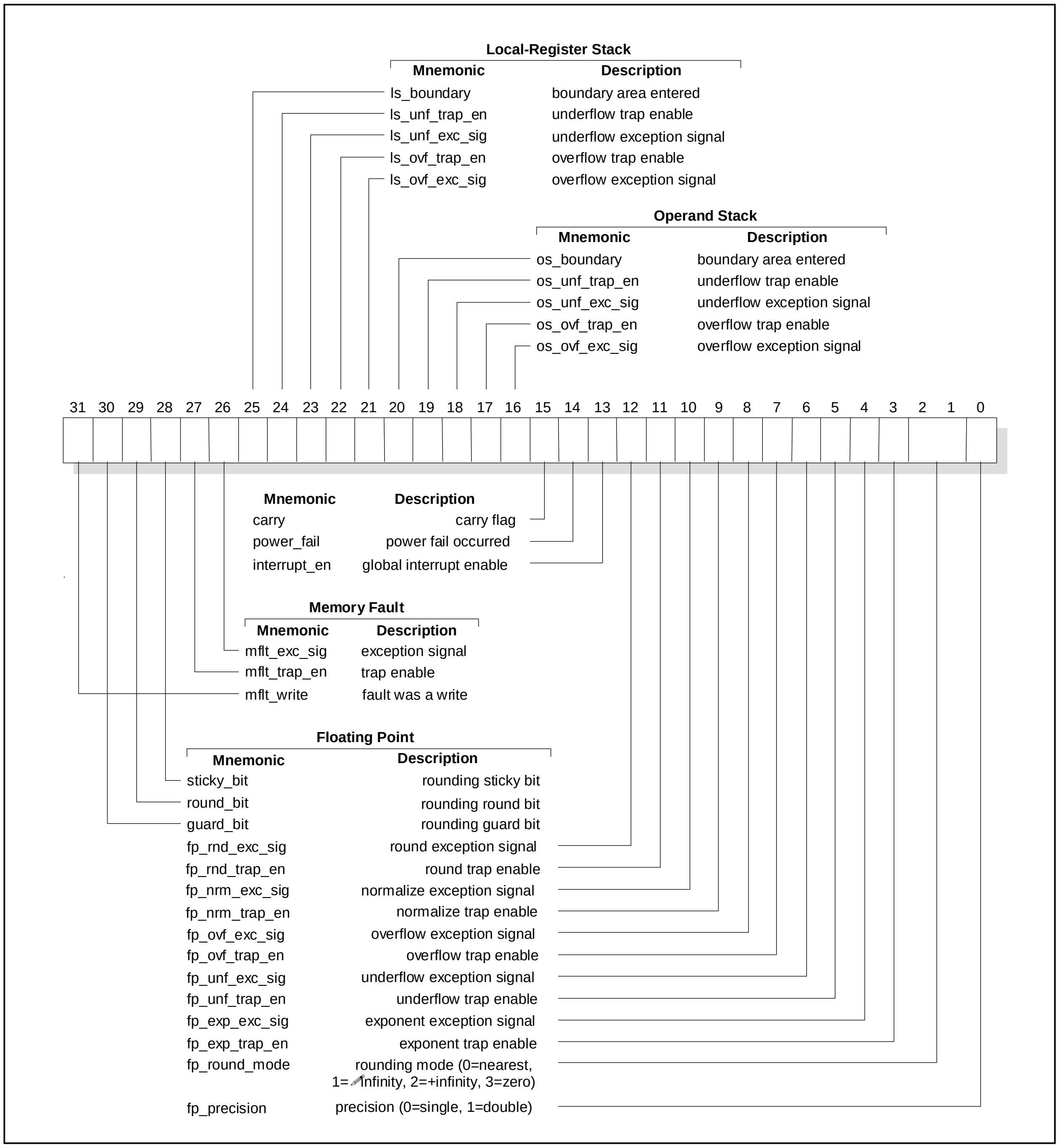


Figure 9 Register Mode

os_ovf_trap_en

If set, enables an operand stack overflow trap to occur after an operand stack overflow exception is signaled.

os_ovf_exc_sig

Set if an operand stack spill occurs, os_boundary is clear, and the accessed memory address is in the first thirty-two cells of a 1024-byte memory page.

carry

Contains the carry bit from the accumulator. Saving and restoring mode can be used to save and restore carry.

power_fail

Set during power-up to indicate that a power failure has occurred. Cleared by any write to mode. Otherwise, not writable.

interrupt_en

If set, interrupts are globally enabled. Set by the instruction ei, cleared by di.

fp_rnd_exc_sig

If set, a previous execution of rnd caused a change in the least significant bit of s0 (s1, if fp_precision is set).

fp_rnd_trap_en

If set, enables a floating-point round trap to occur after a floating-point round exception is signaled.

fp_nrm_exc_sig

If set, one or more of the guard_bit, round_bit and sticky_bit were set after a previous execution of denorm, norml or normr.

fp_nrm_trap_en

If set, enables a floating-point normalize trap to occur after a floating-point normalize exception is signaled.

fp_ovf_exc_sig

If set, a previous execution of normr, addexp or subexp caused the exponent field to increase to or beyond all ones.

fp_ovf_trap_en

If set, enables a floating-point overflow trap to occur after a floating-point overflow exception is signaled.

fp_unf_exc_sig

If set, a previous execution of norml, addexp or subexp caused the exponent field to decrease to or beyond all zeros.

fp_unf_trap_en

If set, enables a floating-point underflow trap to occur after a floating-point underflow exception is signaled.

fp_exp_exc_sig

If set, a previous execution of testexp detected an exponent field containing all ones or all zeros.

fp_exp_trap_en

If set, enables a floating-point exponent trap to occur after a floating-point exponent exception is signaled.

fp_round_mode

Contains the type of rounding to be performed by the CPU instruction rnd.

fp_precision

If clear, the floating-point instructions operate on stack values in IEEE single-precision (32-bit) format. If set, the floating-point instructions operate on stack values in IEEE double-precision (64-bit) format.

CPU Reset

The CPU begins executing at address 0x80000008 with the mode register set to all zeros.

Interrupts

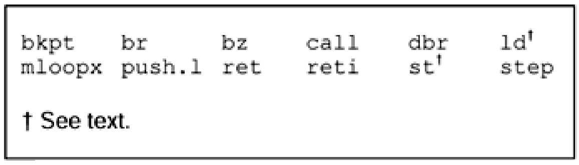
The CPU contains an on-chip prioritized interrupt controller that supports up to eight different interrupt levels. Interrupts can be received through the bit inputs or can be forced in software by writing to ioin. For complete details of interrupts and their servicing, see *Interrupt Controller*.

Bit Inputs

The CPU contains eight general-purpose bit inputs that are shared with the INTC as requests for those services. The bits are taken from IN[7:0]. See *Bit Inputs*.

Bit Outputs

The CPU contains eight general-purpose bit outputs which can be written by the CPU. The bits are output on OUT[7:0]. See *Bit Outputs*.



bkpt	br	bz	call	dbr	ld†
mloopx	push.l	ret	reti	st†	step

† See text.

Table 34 Instructions that Hold-off Pre-fetch

The CPU issues bus requests ordered to optimize execution. To keep executing instructions as much as possible, the next group of instructions are fetched while the current group executes. This is referred to as *instruction pre-fetch*. Instruction pre-fetch begins as soon as an instruction group begins to execute unless it is held off. Pre-fetch is held off if the executing instruction group contains one of the instruction in Table 34. ld and st only hold-off pre-fetch if they occur as the first instruction in the executing instruction group. Knowing which instruction hold-off pre-fetch is useful when programming bus configuration information.

Posted-Write

The CPU supports a one-level posted write. This allows CPU execution to continue unimpeded after the write is posted. To maintain memory coherency, posted writes have the highest priority of all CPU bus requests. This guarantees that memory reads following a posted write will always retrieve the most up-to-date data.

On-Chip Resources

The non-CPU hardware features of the CPU are generally accessed by the CPU through a set of 8 registers located in their own address space. Using a separate address space simplifies implementation, preserves opcodes, and prevents cluttering the normal memory address space with peripherals. Collectively known as the On-Chip Resources, these registers allow access to the bit inputs, bit outputs, INTC and system configuration. These registers and their functions are referenced throughout this manual and are described in detail in *On-Chip Resource Registers*.

Instruction Reference

As a stack-based CPU architecture, the IGNITE PROCESSOR CPU instructions have documentation requirements similar to other stack-based systems, such as the Java Virtual Machine (JVM) and American National Standard Forth (ANS Forth). Not surprisingly, many of the JVM and ANS Forth operations are instructions on the IGNITE CPU. As a result, the JVM and ANS Forth stack notation used for language documentation is useful for describing IGNITE CPU instructions. The basic notation adapted for the IGNITE CPU is:

(input_operands -- output_operands)

(L: input_operands -- output_operands)

where "--" indicates the execution of the instruction. "Input_operands" and "output_operands" are lists of values on the operand stack (the default) or local register stack (preceded by "L:"). These are similar, though not always identical, to the source and destination operands that can be represented within instruction mnemonics. The value held in the top-of-stack register (s0 or r0) is always on the right of the operand list with the values held in the higher ordinal registers appearing to the left (e.g., s2 s1 s0). The only items in the operand lists are those that are pertinent to the instruction; other values may exist under these on the stacks. All of the input_operands are considered to be popped off the stack, the operation performed, and the output_operands pushed on the stack. For example, a notational expression of:

n1 n2 -- n3

represents two input operands, n1 and n2, and one output operand, n3. For the instruction add, n1 (taken from s1) is added to n2 (taken from s0), and the result is n3 (left in s0). If the name of a value on the left of either diagram is the same as the name of a value on the right, then the value was required, but unchanged. The name represents the operand type. Numeric suffixes are added to indicate different or changed operands of the same type. The values may be bytes, integers, floating-point numbers, addresses, or any other type of value that can be placed in a single 32-bit cell.

addr	address
byte	character or byte (upper 24 bits zero)
n	integer or 32 arbitrary bits
other text	integer or 32 arbitrary bits

ANS Forth defines other operand types and operands that occupy more than one stack cell; those are not used here.

Note that typically all stack action is described by the notation and is not explicitly described in the text. If there are multiple possible outcomes then the outcome options are on separate lines and are to be considered as individual cases. If other registers or memory variables are modified, then that effect is documented in the text.

Also on the stack diagram line is an indication of the effect on carry, if any, as well as the opcode and execution time at the right margin.

A timing with an "M" indicates the specified number of bus requests and bus transactions (memory cycles) for the instruction to complete. The value used for "M" includes both the bus request and bus transaction times and depends on the memory interface implemented.

Timings do not include implied memory cycles such as stack spills and refills required to maintain the state of the stack caches. Any operation that pushes or pops a stack, or references a local register could cause a memory cycle. Operations that wait on the completion of instruction pre-fetch are labeled "Mprefetch." These are distinct in that pre-fetch occurs in parallel with execution so the wait time is probably not a full memory cycle.

ANS Forth Word Equivalents

Those IGNITE CPU instructions that are exact equivalents of ANS Forth words are indicated in the body text for the instruction. Many additional ANS Forth words simply require a short instruction sequence, but these are not indicated.

Java Byte Code Equivalents

Those IGNITE CPU instructions that are exact equivalents of Java byte codes are indicated in the body text for the IGNITE CPU instruction. Many additional Java byte codes simply require a short instruction sequence, though the most complex byte codes require a subroutine call. For detailed information contact PTSC.

add

add (*n1 n2* -- *n3*) carry± 1100 0000
0xC0
1 CPU-clock

Add *n1* and *n2* giving the sum *n3*. carry is set if there is a carry out of bit 31 of the sum and cleared otherwise.

Equivalent to Java byte code iadd.

Equivalent to ANS Forth word +.

add pc (*n1* -- *n2*) 1011 1011
0xBB
1 CPU-clock

Add the value of pc (the byte-aligned address of the add pc opcode) to *n1* giving the sum *n2*. carry is set if there is a carry out of bit 31 of the sum and cleared otherwise.

adda

Add Address

adda (*n1 n2* -- *n3*) 1110 1000
0xE8
1 CPU-clock

Add *n1* and *n2* giving the sum *n3*. carry is unaffected.

addc

Add with Carry

addc (*n1 n2* -- *n3*) carry± 1100 0010
0xC2
1 CPU-clock

Add *n1* and *n2* and carry giving the sum *n3*. carry is set if there is a carry out of bit 31 of the sum, otherwise carry is cleared.

addexp

Add Exponents

addexp (*n1 n2* -- *n3 n4 n5*) 1101 0010
0xD2
2 CPU-clocks
(L: -- *addr*) only when trap processed 4+M CPU-clocks

Perform the following:
Exponent_Field(*n5*) = Exponent_Field(*n1*) - BIAS + Exponent_Field(*n2*)
Sign_Bit(*n5*) = Sign_Bit(*n1*) XOR Sign_Bit(*n2*)
BIAS is 127 (0x3F800000 in position) for single precision and 1023 (0x3FF00000 in position) for double precision, as selected by fp_precision.

CoCPUte as described above. Clear the exponent field bits and sign bit and set the hidden bit of *n1* and *n2*, giving *n3* and *n4*, respectively. *n5* is the result of the coCPUtation. After completion, if the exponent-field calculation result equaled or exceeded the maximum value of the exponent field (exponent field result 255 for single, exponent field result 2047 for double) an overflow exception is signaled. If the exponent-field calculation result is less than or equal to zero an underflow exception is signaled. When an exception is signaled, the exponent field of *n5* contains as many low-order bits of the coCPUted exponent as it will hold.

and

Bitwise AND

and (*n1 n2* -- *n3*) carry clear 1110 0001
0xE1
1 CPU-clock

Perform a bitwise AND of *n1* and *n2* giving the result *n3*.

Equivalent to Java byte code iand.

Equivalent to the ANS Forth word AND.

bkpt

Breakpoint

bkpt	(--)	0011 1100
	(L: -- *addr*)	0x3C
		1+M CPU-clocks

Perform a call subroutine to the breakpoint trap location, 0x134. *addr* is the address of the bkpt instruction. Typically the breakpoint service routine replaces the bkpt opcode at *addr* with the original opcode, performs whatever debugging function desired, and ret to *addr*.

Equivalent to Java byte code breakpoint.

b

Branch if Condition

br *offset*	(--)	0000 0xxx
Branch Unconditionally		0x0?
		M CPU-clocks

Transfer execution to *offset* cells from the beginning of the current instruction group.

The instruction adds the two's-complement cell offset encoded within and following the br opcode to pc, and transfers execution to the resulting cell-aligned address.

Equivalent to Java byte codes goto, goto_w.

Equivalent to the run-time for the ANS Forth words AGAIN, AHEAD, ELSE.

br []	(*addr* --)	0100 1011
Branch Indirect		0x4B
		M CPU-clocks

Replace the value in pc with *addr* to transfer execution to *addr*. Note that *addr* is an absolute byte-aligned address and not an offset.

bz *offset*	(*n* --)	0001 0xxx
Branch if Zero		0x1?
		M CPU-clocks

If *n* is zero, transfer execution to *offset* cells from the beginning of the instruction group; otherwise, continue execution at the next instruction group.

If *n* is zero the instruction adds the two's-complement cell offset encoded within and following the bz opcode to pc, and transfers execution to the resulting cell-aligned address. If *n* is non-zero execution continues with the next instruction group.

Equivalent to Java byte codes ifeq, ifnull.

Equivalent to the run-time for the ANS Forth words IF, UNTIL, WHILE.

dbr *offset* (--) 0001 1xxx
Decrement CT and Branch 0x1?
M CPU-clocks

Decrement ct by one. If ct is non-zero, transfer execution to *offset* cells from the beginning of the current instruction group; otherwise, continue execution with the next instruction group.

The instruction decrements ct by one. If the resulting ct is non-zero the instruction then adds the two's-complement cell offset encoded within and following the dbr opcode to pc, and transfers execution to the resulting cell-aligned address. If the resulting ct is zero execution continues with the next instruction group.

cache

Fill/Empty Stack Cache

The cache instructions are used to optimize program execution, or to make program execution more deterministic. Stack cache spills and refills can be caused to occur at preferential times, and to occur in bursts to optimize memory access. Executing the instruction with both *n* and *n*-14 (*n*>0) ensures that an exact number of items are in the stack cache. Pushing dummy values onto the stack (one value for the local-register stack, three values for the operand stack) and then executing the instruction with *n* = -14 causes all previously held data to be spilled to memory. Note that if stack-page exceptions are enabled, a trap might occur and change the state of the stacks from that set by the cache instruction. See *Stack-Page Exceptions* on page ?.

lcache (*n* --) 0100 1101
0x4D
1 or (1M to 14M) CPU-clocks

If *n* > 0, ensure that at least *n* cells can be removed from the local-register stack without causing local-register stack cache refills. Cells are refilled from memory into the cache if required. (1 *n* 14).

If *n* < 0 (two's complement), ensure that at least *n* cells can be added to the local-register stack without causing local-register stack cache spills. Cells are spilled from the stack cache to memory if required. (-14 *n* -1).

If n = 0 the local-register stack cache is unchanged.

scache (*n* -- *n*) 0100 0101
0x45
1 or (1M to 14M) CPU-clocks

If *n* > 0, ensure that at least *n* cells can be removed from the operand stack without causing operand stack cache refills. Cells are refilled from memory into the cache if required. (1 n 14).

If *n* < 0 (two's complement), ensure that at least *n* cells can be added to the operand stack without causing operand stack cache spills. Cells are spilled from the stack cache to memory if required. (-14 n -1)

If *n* = 0 the operand stack cache is unchanged.

call

Call Subroutine

call *offset*	(--)	0000 1xxx
	(L: -- *addr*)	0x0?
Call Subroutine		1+M CPU-clocks

Transfer execution to *offset* cells from the beginning of the current instruction group. *addr* is the cell-aligned address of the next instruction group.

The instruction pushes *addr* on the local-register stack and then adds the two's-complement cell *offset* encoded within and following the call opcode to pc, and transfers execution to the resulting cell-aligned address. The *offset* is in the same form and follows the same rules as those for branches.

call []	(*addr1* --)	0100 1110
	(L: -- *addr2*)	0x4E
Call Subroutine Indirect		1+M CPU-clocks

Replace the value in pc with *addr1* to transfer execution there. *addr2* is the byte-aligned address of the next instruction following call []. Note that *addr1* is an absolute address and not an offset.

cmp

Compare

cmp	(*n1 n2* -- *n1 n2*)	carry±	1100 1011
			0xCB
			1 CPU-clock

Compare *n2* and *n1* as signed values. Set carry if *n1* < *n2*, otherwise clear carry.

copyb

Copy Byte Across Cell

copyb	(*n1* -- *n2*)	1101 0000
		0xD0
		1 CPU-clock

n2 is the result of copying the lowest byte of *n1* into each of the higher byte positions. For example, 0x12345678 becomes 0x78787878.

dbr

See _b_.

dec

Decrement

dec #1	(*n1* -- *n2*)	1100 1111 0xCF 1 CPU-clock

Subtract one from *n1* leaving the result *n2*.

Equivalent to ANS Forth word 1-.

dec #4	(*n1* -- *n2*)	1100 1101 0xCD 1 CPU-clock

Subtract four from *n1* leaving the result *n2*.

dec ct, #1	(--)	1100 0001 0xC1 1 CPU-clock

Subtract one from ct.

denorm

Denormalize

denorm	(*n1* -- *n2*) if single precision (*n1 n2* -- *n3 n4*) if double precision	1100 0101 0xC5 1 to 13 CPU-clocks
	(L: -- *addr*) only when trap processed	3+M to 15+M CPU-clocks

Shift *n1* (or *n2n1* if double) right by the bit count in the exponent field of ct. Bits shift out of the right into the GRS extension. If any bit in the GRS extension is set, a normalize exception is signaled. The location of the exponent field depends on fp_precision. The exponent field of ct is decremented to zero.

Shifting is performed by bytes or bits to minimize CPU-clock cycles required. If the count in the exponent bits of ct is larger than the width in bits of the significand field + 3 (for the guard_bit, round_bit and the hidden bit), the sticky_bit is set and the other bits are cleared, and execution requires one CPU-clock cycle.

depth

Depth of Stack

Note that if stack-page exceptions are enabled, a trap might occur and change the state of the stacks from that returned. See *Stack-Page Exceptions* on page ?.

ldepth (-- *n*) 1001 1011
0x9B
1 CPU-clock

n is exactly the number of cells that can be removed from the local-register stack without causing a local-register stack cache refill. (0 *n* 14).

sdepth (-- *n*) 1001 1111
0x9F
1 CPU-clock

n is exactly the number of cells, before *n* was pushed, that could be removed from the operand stack without causing an operand stack cache refill. (0 *n* 14). If *n* = 14, then an operand stack cache spill occurred when *n* was pushed and only 13 cells remain, excluding *n*, that can be removed from the operand stack without causing an operand stack cache refill.

di

Disable Interrupts

di (--) 1011 0111
0xB7
1 CPU-clock

Globally disable interrupts, clearing interrupt_en. The ioie bits are not changed.

divu

Divide Unsigned

divu (*n1 n2* -- *n3 n4*) 1101 1110
0xDE
32 CPU-clocks

Divide the double value *n2n1* by the value in g0 giving the quotient *n3* and remainder *n4*. All values are unsigned. If *n2* is greater than or equal to g0 then the quotient will overflow. If g0 is zero then *n3* equals *n1* and *n4* equals *n2*.

ei

Enable Interrupts

ei (--) 1011 0110
0xB6
1 CPU-clock

Globally enable interrupts, setting interrupt_en. The ioie bits are not changed.

eqz

Equal Zero

eqz (*n1* -- *n2*) 1110 0101
0xE5
1 CPU-clock

n2 is the logical inverse of *n1*. If *n1* is equal to zero *n2* is -1. If *n1* is non-zero *n2* is zero.

Equivalent to ANS Forth word 0=.

expdif

Exponent Difference

expdif (*n1 n2* -- *n3 n4*) 1100 0100
0xC4
1 CPU-clock

Clear the upper half of ct. Subtract the exponent field of *n2* from the exponent field in *n1* placing the result in the exponent-field bits of ct. Clear the exponent-field bits and sign bit and set the hidden bit of *n1* and *n2* giving *n3* and *n4*, respectively. The locations of the exponent field and hidden bit depend on fp_precision.

extexp

Extract Exponent

extexp (*n1* -- *n2*) 1101 1011
0xDB
1 CPU-clock

Clear the significand bits of *n1* leaving the exponent-field bits and sign bit unchanged, giving *n2*. The locations of the exponent field and significand field depend on fp_precision.

extsig

Extract Significand

extsig	(*n1* -- *n2*)	1101 1100
		0xDC
		1 CPU-clock

Clear the exponent and sign bits of *n1* leaving the significand-field bits unchanged. Then set the hidden bit of *n1*, giving *n2*. The locations of the exponent field and significand field depend on fp_precision.

frame

Allocate On-Chip Stack Frame

lframe	(n --)		1011 1110
	(L: -- x_n x_1)	($n > 0$)	0xBE
			1 or (1M to 15M) CPU-clocks
	(L: x_n x_1 --)	($n < 0$)	
			1 or (1 to 15) CPU-clocks
	(L: --)	($n = 0$)	1 CPU-clock

If $n > 0$, allocate n uninitialized cells, x_n x_1, at the top of the local-register stack cache. This causes r0 to move to rn, r1 to move to r(n+1), ri to move to r(n+i), etc. Those local registers for which (n+i) > 14 are written from the local-register stack cache to memory. (1 n 15).

If $n < 0$, discard n cells, x_n x_1, from the top of the local-register stack cache. This causes r0 through r(n -1) to be discarded, r n to become r0, r(n +1) to become r1, etc. (-15 n -1). Each cell discarded that is not in the stack cache requires one CPU-clock cycle.

If $n = 0$, no cells are allocated or discarded.

sframe			1011 1111
			0xBF
	(m n -- x_n x_1 m n)	($n > 0$)	
			1 or (1M to 15M) CPU-clocks
	(x_n x_1 m n -- m n)	($n < 0$)	
			1 or (1 to 15) CPU-clocks
	(n -- n)	($n = 0$)	1 CPU-clock

If $n > 0$, allocate n uninitialized cells, x_n x_1, in the operand stack cache after s0 and s1. This causes s2 to move to s(n+2), s3 to move to s(n+3), si to move to s(n+i), etc. Those stack cells for which (n+i) > 16 are written from the operand stack cache to memory. (1 n 15).

If $n < 0$, discard n cells, x_n x_1, from within the operand stack cache after s0 and s1. This causes s2 through s(n +1) to be discarded, s(n +2) to become s2, s(n +3) to become s3, etc. (-15 n -1). Each cell discarded that is not in the stack cache requires one CPU-clock cycle.

If $n = 0$, no cells are allocated or discarded.

iand

Bitwise Invert then AND

iand	(*n1 n2* -- *n3*)	clear carry	1110 1001 0xE9 1 CPU-clock

Clear the bits in *n1* that are set in *n2* leaving the result *n3*.

inc

Increment

inc #1	(*n1* -- *n2*)	1100 1110 0xCE 1 CPU-clock

Add one to *n1* giving the sum *n2*.

Equivalent to ANS Forth word 1+.

inc #4	(*n1* -- *n2*)	1100 1100 0xCC 1 CPU-clock

Add four to *n1* giving the sum *n2*.

lcache

See _cache.

ld

Load Indirect from Memory

ld [--r0] (-- *n*) 0100 0100
0x44
1+M CPU-clocks

Decrement the address in r0 by four. *n* is the value from the cell in memory at the new address in r0. The two least significant bits of the address are ignored and treated as zero.

ld [--x] (-- *n*) 0100 1010
0x4A
1+M CPU-clocks

Decrement the address in x by four. *n* is the value from the cell in memory at the new address in x. The two least significant bits of the address are ignored and treated as zero.

ld [r0++] (-- *n*) 0100 0110
0x46
M CPU-clocks

n is the value from the cell in memory at the address in r0. Increment r0 by four. The two least significant bits of the address are ignored and treated as zero.

ld [r0] (-- *n*) 0100 0010
0x42
M CPU-clocks

n is the value from the cell in memory at the address in r0. The two least significant bits of the address are ignored and treated as zero.

ld [x++] (-- *n*) 0100 1001
0x49
M CPU-clocks

n is the value from the cell in memory at the address in x. Increment x by four. The two least significant bits of the address are ignored and treated as zero.

ld [x] (-- *n*) 0100 0001
0x41
M CPU-clocks

n is the value from the cell in memory at the address in x. The two least significant bits of the address are ignored and treated as zero.

ld [] (*addr* -- *n*) 0100 0000
0x40
M CPU-clocks

n is the value from the cell in memory at the address *addr*. The two least significant bits of the address are ignored and treated as zero.

Equivalent to ANS Forth words @, F@, SF@.

ld.b [] (*addr* -- *byte*) 0100 1000
0x48
M CPU-clocks

byte is the value from the byte in memory at the address *addr*.

ld.w [] (*addr* -- *word*) 0100 1100
0x4C
M CPU_clocks

word is the 16-bit value from the word in memory at address *addr*. The least significant bit of the address is ignored and treated as zero.

Equivalent to ANS Forth word C@.

ldo

Load Indirect from On-Chip Resource

ldo [] (*addr* -- *n*) 1001 0110
0x96
1 CPU-clock

n is the value from the on-chip resource at *addr*. For valid values of *addr*, see *On-Chip Resource Registers*, page 89.

ldo.i [] (*bit_addr* -- *n*) 1001 0111
0x97
1 CPU-clock

n is all ones (-1) if the bit at the on-chip resource address *bit_addr* is one, otherwise *n* is zero. For valid values of *bit_addr*, see *On-Chip Resource Registers*, page 89.

ldepth

See _depth.

lframe

See _frame.

mloop_

Micro Loop on Condition

An mloop re-executes the current instruction group, beginning with the first instruction in the group, up to the mloop_ instruction, until a specified condition is not met or until ct is decremented to zero. When either termination condition occurs, execution continues with the instruction following the mloop_ opcode.

mloop	(--)	0011 1000
Micro Loop Unconditionally		0x38
		1 CPU-clock

Decrement ct by one. If ct is non-zero transfer execution to the beginning of the current instruction group. If ct is zero continue execution with the instruction following mloop.

mloopc	(--)	0011 1001
Micro Loop if Carry		0x39
		1 CPU-clock

Decrement ct by one. If ct is non-zero and carry is set transfer execution to the beginning of the current instruction group. If ct is zero or carry is clear continue execution with the instruction following mloopc.

mloopn		
mloopnp	(*n* -- *n*)	0011 1010
Micro Loop if Negative/Not Positive		0x3A
		1 CPU-clock

Decrement ct by one. If ct is non-zero and *n* is negative (neither positive nor zero) transfer execution to the beginning of the current instruction group. If ct is zero or *n* is not negative (either positive or zero) continue execution with the instruction following mloopn or mloopnp.

mloopnc	(--)	0011 1101
Micro Loop if Not Carry		0x3D
		1 CPU-clock

Decrement ct by one. If ct is non-zero and carry is clear transfer execution to the beginning of the current instruction group. If ct is zero or carry is set continue execution with the instruction following mloopnc.

mloopnn
mloopp (*n* -- *n*) 0011 1110
Micro Loop if Not Negative/Positive 0x3E
1 CPU-clock

Decrement ct by one. If ct is non-zero and *n* is not negative (either positive or zero) transfer execution to the beginning of the current instruction group. If ct is zero or *n* is negative (neither positive nor zero) continue execution with the instruction following mloopnn or mloopp.

mloopnz (*n* -- *n*) 0011 1111
Micro Loop if Not Zero 0x3F
1 CPU-clock

Decrement ct by one. If ct is non-zero and *n* is not zero transfer execution to the beginning of the current instruction group. If ct is zero or *n* is zero continue execution with the instruction following mloopnz.

mloopz (*n* -- *n*) 0011 1011
Micro Loop if Zero 0x3B
1 CPU-clock

Decrement ct by one. If ct is non-zero and *n* is zero transfer execution to the beginning of the current instruction group. If ct is zero or *n* is not zero continue execution with the instruction following mloopz.

mulfs

Multiply Fast Signed

mulfs (*n1 n2* -- *n3 n4*) 1101 0110
0xD6
2 to 32 CPU-clocks

Multiply the bit-order-reversed value *n1* by the value in g0 leaving the result *n4*. *n2* is usually zero and *n3* is garbage (see below). The number of significant bits in *n1* is indicated by the value in ct. All values are single-cell size and signed. ct is decremented to zero.

The program must supply *n1* in bit-order-reversed form (e.g., the binary value for decimal 13 is 01101 and bit-order reversed is 10110; note that the original high-order bit is zero as a sign bit and must be included.) The program must also load ct with the bit count and push a zero for *n2*. For the example number above, the count would be 5. *n3* is typically discarded.

n2 could be non-zero but its use in this form is questionable. The effect of *n2* on the result is that the value of *n2* shifted left by the bit count value in ct is added to the result, *n4*. *n3* contains the low cell of the value remaining after *n2n1* is shifted right by the number of bits in ct. Instruction execution time is limited to 65 CPU-clock cycles by the instruction expiration counter.

muls

Multiply Signed

muls	(*n1 n2* -- *n3 n4*)	1101 0101
		0xD5
		32 CPU-clocks

Multiply *n1* by the value in g0 and add *n2*, leaving the double result *n4n3*. All values are signed.

mulu

Multiply Unsigned

mulu	(*n1 n2* -- *n3 n4*)	1101 0111
		0xD7
		32 CPU-clocks

Multiply *n1* by the value in g0 and add *n2*, leaving the double result *n4n3*. All values are unsigned.

mxm

Maximum

mxm	(*n1 n2* -- *n1 n2*)	carry set	1101 1111
or	(*n1 n2* -- *n2 n1*)	carry clear	0xDF
			2 CPU-clocks

Compare *n2* and *n1* as signed values. Set carry if *n1* < *n2*, otherwise clear carry. Bring the larger of *n1* and *n2* to the top of stack. That is, if the resulting carry is set then *n2* is greater than *n1* and *n2* remains on top. If the resulting carry is clear then *n2* is less than or equal to *n1* and *n1* is exchanged with *n2*.

neg

Two's-Complement Negation

neg (*n1* -- *n2*)	1100 1001
	0xC9
	1 CPU-clock

n2 is the two's-complement negation of *n1*.

Equivalent to Java byte code ineg.

Equivalent to ANS Forth word NEGATE.

nop

No Operation

nop (--)	1110 1010 0xEA 1 CPU-clock

Do nothing.

Equivalent to Java byte code nop.

norml

Normalize Left

norml	(*n1* -- *n2*) if single precision (*n1 n2* -- *n3 n4*) if double precision	1100 0111 0xC7 1 to 13 CPU-clocks
	(L: -- *addr*) only when trap processed	3+M to 15+M CPU-clocks
	(L: -- *addr1 addr2*) only when both traps processed	5+2M to 17+2M CPU-clocks

While the hidden bit and the seven bits to the right of it in *n1* (*n2* if double) are zero, repeat the following:
   Shift *n1* (or *n2n1* if double) left by eight bits and decrement the exponent field in ct by eight.
Then, while the hidden bit of *n1* (*n2* if double) is zero, repeat the following:
   Shift *n1* (or *n2n1* if double) left by one bit and decrement the exponent field in ct by one.

In both steps, bits shifted into bit zero of *n1* come from the GRS extension.

When the operation is complete, if shifting was required and the decremented field in ct reached or passed all zero bits during the processing, an underflow exception is signaled. If no shifting is required an underflow exception is not signaled. Then, if any bit in the GRS extension is set, a normalize exception is signaled. The location of the exponent field depends on fp_precision. If both traps are processed, the underflow trap has higher priority. Instruction execution time is limited to 65 CPU-clock cycles by the instruction expiration counter.

normr

Normalize Right

normr	(*n1* -- *n2*) if single precision	1100 0110
	(*n1 n2* -- *n3 n4*) if double precision	0xC6
		1 to 11 CPU-clocks
	(L: -- *addr*) only when trap processed	
		3+M to 13+M CPU-clocks
	(L: -- *addr1 addr2*) only when both traps processed	
		5+2M to 15+2M CPU-clocks

While any bit except the first bit (the hidden bit) in the exponent field is non-zero, repeat the following:
Shift *n1* (or *n2n1* if double) right by one bit and increment the exponent field in ct by one. Bits shifted out of bit zero of *n1* shift into the GRS extension bits.

When the operation is complete, if shifting was required and the incremented field in ct reached or passed all one bits during the processing, an overflow exception is signaled. If no shifting is required an overflow exception is not signaled. Then, if the GRS extension is set, a normalization exception is signaled. The locations of the exponent field and hidden bit depend on fp_precision. If both traps are processed, the overflow trap has higher priority.

notc

Complement Carry

notc	(--)	carry inverted	1101 1101
			0xDD
			1 CPU-clock

Invert the state of carry.

or

Bitwise OR

or (*n1 n2* -- *n3*) carry clear

1110 0000
0xE0
1 CPU-clock

Perform a bitwise OR on *n1* and *n2* giving the result *n3*.

Equivalent to Java byte code ior.

Equivalent to ANS Forth word OR.

pop

pop (*n* --)

1011 0011
0xB3
1 CPU-clock

Discard *n*.

Equivalent to Java byte codes pop, l2i.
Equivalent when executed twice to Java byte code pop2.

Equivalent to ANS Forth word D>S, DROP, FDROP.
Equivalent when executed twice to ANS Forth word 2DROP.

pop ct (*n* --)

1011 0100
0xB4
1 CPU-clock

Replace the value in ct with *n*.

pop g*i* (*n* --)

0101 xxxx
0x5?
1 CPU-clock

Replace the value in g*i* (global register *i*, i.e., g0–g15) with *n*.

pop la (*addr* --)
(L: j_n j_1 --)

1011 1101
0xBD
1+M CPU-clocks

Replace the value in la with cell-aligned address *addr*. The contents of the local-register stack cache, j_n j_1, are discarded. The two least-significant bits of la are cleared. The bit ls_boundary is cleared. A stack refill is performed at *addr*+4 to initialize r0.

pop lstack (*n* --)
(L: -- *n*)

1011 1010
0xBA

1 CPU-clock

Remove *n* from the operand stack and push it onto the local-register stack (into r0). The previous contents of r0 are placed in r1, the previous contents of r1 are placed in r2, and so on.

Equivalent to ANS Forth word >R.
Equivalent when executed twice to ANS Forth word 2>R.

pop mode (*n* --) 1011 1001
0xB9
1 CPU-clock

Replace the value in mode with *n* and clear power_fail. The mode bits power_fail, ls_boundary and os_boundary are not writeable.

pop r*i* (*n* --) 1010 xxxx
0xA?
1 CPU-clock

Replace the value in r*i* (local register *i*, i.e., r0–r14) with *n*.

If r*i* is in the local-register stack cache (*i*   ldepth) the value in r*i* is replaced with *n*. If r*i* is not currently in the local-register stack cache (*i* > ldepth), cells starting at r(ldepth+1) are read from memory sequentially to fill the cache until r*i* is reached. r*i* is then replaced with the value *n*.

Equivalent to Java byte codes astore_0, astore_1, astore_2, astore_3, fstore_0, fstore_1, fstore_2, fstore_3, istore_0, istore_1, istore_2, istore_3.
Equivalent when executed twice to Java byte codes dstore_0, dstore_1, dstore_2, dstore_3, lstore_0, lstore_1, lstore_2, lstore_3.
Equivalent for indexes up to fourteen (almost all actual cases) to Java byte codes astore (vindex), fstore (vindex), istore (vindex).
Equivalent when executed twice for indexes up to thirteen (almost all actual cases) to Java byte codes dstore (vindex), lstore (vindex).

pop sa (j_n j_1 *m1 m2 addr* -- *m1 m2*) 1011 1100
0xBC
1+M CPU-clocks

Replace the value in sa with cell-aligned address *addr*. The contents of the operand stack cache, j_n j_1, are discarded. The two least-significant bits of sa are cleared. The bit os_boundary is cleared. A stack refill is performed at *addr*+4 to initialize s2.

pop x (*n* --) 1011 1000
0xB8
1 CPU-clock

Replace the value in x with *n*.

push

push	(*n* -- *n n*)	1001 0010 0x92 1 CPU-clock

Duplicate *n*.

Equivalent to Java byte code dup.

push ct	(-- *n*)	1001 0100 0x94 1 CPU-clock

n is the value in ct.

push g*i*	(-- *n*)	0111 xxxx 0x7? 1 CPU-clock

n is the value in g*i* (global register *i*, i.e., g0–g15).

push la	(-- *addr*)	1001 1101 0x9D 1 CPU-clock

addr is the value in la. Note that if stack-page exceptions are enabled, a trap might occur and change the state of the stacks from that returned. See *Stack-Page Exceptions* on page ?.

push lstack	(-- n) (L: n --)	1001 1010 0x9A 1 CPU-clock

Pop *n* from the local-register stack (from r0) and push it onto the operand stack. The previous contents of r1 are placed in r0, the previous contents of r2 are placed in r1, and so on.

Equivalent to ANS Forth word R>.
Equivalent when executed twice to ANS Forth word 2R>.

push mode	(-- *n*)	1001 0001 0x91 1 CPU-clock

n is the value in mode.

push r*i* (-- *n*) 1000 xxxx
0x8?
1 CPU-clock

n is the value in r*i* (local register *i*, i.e. r0–r14).

If r*i* is in the local-register stack cache (*i* ldepth) the value in r*i* is pushed onto the operand stack. If r*i* is not currently in the local-register stack cache (*i* > ldepth), cells starting at r(ldepth+1) are read from memory sequentially until r*i* is reached. The value in r*i* is then pushed onto the operand stack.

Equivalent to Java byte codes aload_0, aload_1, aload_2, aload_3, fload_0, fload_1, fload_2, fload_3, iload_0, iload_1, iload_2, iload_3.
Equivalent when executed twice to Java byte codes lload_0, lload_1, lload_2, lload_3, dload_0, dload_1, dload_2, dload_3.
Equivalent for indexes up to fourteen (almost all actual cases) to Java byte codes aload (vindex), fload (vindex), iload (vindex).
Equivalent when executed twice for indexes up to thirteen (almost all actual cases) to Java byte codes dload (vindex), lload (vindex).

Equivalent to ANS Forth word R@.
Equivalent when executed twice to ANS Forth word 2R@.

push s*i* (-- *n*) s0 1001 0010
0x92
s1 1001 0011
0x93
s2 1001 1110
0x9E
1 CPU-clock

n is the value in s*i* (operand stack register *i*, i.e., s0, s1 or s2)

Equivalent to Java byte code dup.
Equivalent when executed twice to Java byte code dup2.

Equivalent to ANS Forth words 2DUP, DUP, FDUP, FOVER, OVER.

push sa (-- *addr*) 1001 1100
0x9C
1 CPU-clock

addr is the value in sa. Note that if stack-page exceptions are enabled, a trap might occur and change the state of the stacks from that returned. See *Stack-Page Exceptions* on page ?.

push x (-- *n*) 1001 1000
0x98
1 CPU-clock

n is the value in x.

push.b #*n* (-- *n*) 1001 0000
0x90
1 CPU-clock

n is an eight-bit literal value in the range 0–255. The byte literal is encoded as the last byte in the instruction group. This allows only one unique push.b # value per instruction group. Multiple push.b # opcodes in the same instruction group push the same value.

Equivalent for positive values to Java byte code bipush.
Equivalent for some values to Java byte code sipush.

push.l #*n* (-- *n*) 0100 1111
0x4F
M CPU-clocks

n is a 32-bit literal value. The value is compiled as a full cell following the instruction group. Multiple push.l # in an instruction group are compiled with data in sequential cells following the instruction group in memory. As the push.l # opcodes are executed, the internally maintained next pc is incremented to move past each cell as it is fetched and pushed on the stack. Note that skipping a push.l # causes the CPU to execute the literal value because the skipped push.l # will not have incremented next pc to move past the value.

Equivalent to Java byte code fconst_1, fconst_2, ldc, ldc_w, sipush.
Equivalent when executed twice to Java byte code ldc2_w.

push.n #*n* (-- *n*) 0010 xxxx
0x2?
1 CPU-clock

n is a literal value in the range -7 to 8. The four least-significant bits of the opcode encode the value for *n*. The value is encoded as a two's-complement representation of *n* except that -8 (1000 binary) is decoded to be +8.

Equivalent to Java byte codes aconst_null, fconst_0, iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5.
Equivalent for some values to Java byte code bipush.
Equivalent when executed twice to Java byte codes dconst_0, lconst_0, lconst_1.

Equivalent to ANS Forth words FALSE, TRUE.

replb

Replace Byte

replb (*n1 n2* -- *n3*) 1101 1010
0xDA
1 CPU-clock

Replace the target byte of *n2* with the least-significant byte of *n1*, leaving the result *n3*. The target byte is selected by the two least-significant bits of x, as when accessing a byte in memory.

For example, if x = 0x121, *n1* = 0xCCDDEEFF, and *n2* = 0x12345678, then *n3* = 0x12FF5678.

replw

Replace Word

replw (*n1 n2* -- *n3*) 1110 1011
0xEB
1 CPU-clock

Replace the target 16-bit word of *n2* with the least-significant word of *n1*, leaving the result *n3*. The target word is selected by the next-to-least-significant bit of x, as when accessing a word in memory. The least-significant bit of x is ignored.

For example, if x = 0x121, *n1* = 0xCCDDEEFF, and *n2* = 0x12345678, then *n3* = 0xEEFF5678.

replexp

Replace Exponent

replexp (*n1 n2* -- *n3*) 1011 0101
0xB5
1 CPU-clock

Replace the exponent field and sign bits of *n1* with the corresponding bits of *n2*. Clear the GRS extension. The location of the exponent field depends on fp_precision.

ret

Return

ret (--)	0110 1110
(L: *addr* --)	0x6E
Return from Subroutine	M CPU-clocks

Pop *addr* from the local-register stack into pc to transfer execution to *addr*.

Equivalent to ANS Forth word EXIT.

reti (--)	0110 1111
(L: *addr* --)	0x6F
Return from Interrupt	M CPU-clocks

Pop *addr* from the local-register stack into pc to transfer execution to *addr*. Clear the current interrupt under-service bit.

rev

Revolve Operand Stack

rev (*n1 n2 n3* -- *n2 n3 n1*)	1110 0100
	0xE4
	1 CPU-clock

Rotate the top three cells of the stack to bring *n1* to the top.

Equivalent to the run-time for the ANS Forth words FROT, ROT.

rnd

Round

rnd (*n1* -- *n2*)	carry±	1101 0001
		0xD1
		1 CPU-clock
	(L: -- *addr*) only when trap processed	3+M CPU-clocks

Round *n1* giving *n2*. Rounding is based on fp_round_mode, the sign of ct, and the GRS extension. See *Rounding*, page 24. If an increment carried out of bit 31 then set carry, clear carry otherwise.

If the value of *n2* is different from *n1*, a rounded exception is signaled. The exception is detected as a change in the value of bit zero.

scache

See _cache.

sdepth

See _depth.

sexb

Sign-extend byte

sexb	(*n1* -- *n2*)	1101 1000
		0xD8
		1 CPU-clock

Copy the value of bit seven of *n1* into bits eight to thirty-one, leaving *n2*.

sexw

Sign-extend word

sexw (*n1* -- *n2*) 1001 0101
0x95
1 CPU-clock

Copy the value of bit fifteen of *n1* into bits sixteen to thirty-one, leaving *n2*

Equivalent to Java byte code i2b.

shift_

The number of CPU-clock cycles required to shift the specified number of bits depends on the number of bits requested. While the count eight the value (single or double) is shifted eight bits each CPU-clock cycle. When the count becomes less than eight the shifting is finished at one bit per CPU-clock cycle. For instance, the worst-case useful shift is 31 bits (either left or right) and takes eleven CPU-clock cycles—three 8-bit shifts and seven 1-bit shifts plus one CPU-clock cycle for setup. A 32-bit shift would take five CPU-clock cycles. The counts are modulo 64 in sign-magnitude representation using only the six least-significant bits for the magnitude and bit 31 for the sign. A zero in the six least-significant bits represents zero. (Sign-magnitude representation here is a positive integer count in the six least-significant bits, the middle bits ignored, and bit 31 indicating the sign, zero is positive, one is negative).

shift (*n1 n2* -- *n3*) carry± (*n2*>0) 1110 1110
0xEE
1 to 11 CPU-clocks

Shift *n1* by *n2* bits leaving the result *n3*. If *n2* is positive the shift is to the left, each bit is shifted out through carry, and zero is shifted into each bit on the right. If *n2* is negative the shift is to the right, each bit shifted out is shifted through the GRS extension, and carry is copied into each high order bit of *n1* vacated by the shift. See text above regarding execution time and format of negative counts.

Equivalent to ANS Forth word LSHIFT.

shiftd (*n1 n2 n3* -- *n4 n5*) carry± (*n3*>0) 1110 1111
Shift Double 0xEF
1 to 15 CPU-clocks

Shift the cell pair *n2n1* by *n3* bits leaving the resulting cell pair *n5n4*. If *n3* is positive the shift is to the left, each bit is shifted out of *n2* through carry, and zero is shifted into each bit on the right into *n1*. If *n3* is negative the shift is to the right, each bit shifted out of *n1* is shifted through the GRS extension, and carry is copied into each high order bit of *n2* vacated by the shift. See text above regarding execution time and format of negative counts.

shl_

Shift Left

shl #1	(*n1* -- *n2*)	carry±	1110 0010
Shift Left			0xE2
			1 CPU-clock

Shift *n1* one bit to the left leaving the result *n2*. The high order bit of *n1* shifted out goes into carry. The vacated bit on the right of *n1* is filled with zero.

Equivalent to ANS Forth word 2*.

shl #8	(*n1* -- *n2*)	carry±	1110 1100
Shift Left Byte			0xEC
			1 CPU-clock

Shift *n1* eight bits (one byte) to the left leaving *n2*. The last bit shifted out goes into carry. The vacated eight bits on the right are filled with zeros.

shld #1	(*n1 n2* -- *n3 n4*)	carry±	1110 0110
Shift Left Double			0xE6
			1 CPU-clock

Shift cell pair *n2n1* one bit to the left leaving the result *n4n3*. The high order bit of *n2* shifted out goes into carry. The vacated bit on the right of *n1* is filled with zero.

Equivalent to ANS Forth word D2*.

shr_

Shift Right

shr #1	(*n1* -- *n2*)	1110 0011
Shift Right		0xE3
		1 CPU-clock

Shift *n1* one bit to the right leaving the result *n2*. The bit shifted out is shifted into the GRS extension. The vacated bit on the left is filled with carry.

shr #8	(*n1* -- *n2*)	1110 1101
Shift Right Byte		0xED
		1 CPU-clock

Shift *n1* eight bits (one byte) to the right leaving the result *n2*. The bits shifted out are shifted into the GRS extension. The vacated eight bits on the left are filled with carry.

shrd #1	(*n1 n2* -- *n3 n4*)	1110 0111
Shift Right Double		0xE7
		1 CPU-clock

Shift cell pair *n2n1* one bit to the right leaving the result *n4n3*. The bit shifted out of *n1* is shifted into the GRS extension. The vacated bit in *n2* on the left is filled with carry.

skip

Skip if Condition

skip conditionally or unconditionally skips execution of the remainder of the instruction group. If the condition is true, skip the remainder of the instruction group and continue execution with the following instruction group. If condition is false, continue execution with the next instruction.

WARNING: Do not skip a push.l #. Since the CPU will not have executed the push.l # opcode, the corresponding literal cell is not skipped. The result will be the CPU executing the literal cell.

skip (--) 0011 0000
Skip Unconditionally 0x30
Mprefetch CPU-clocks

Unconditionally skip the remainder of the instruction group.

skipc (--) 0011 0011
Skip if Carry 0x31
1 (no carry) Mprefetch (carry) CPU-clocks

If carry is set, skip the remainder of the instruction group and continue execution with the next instruction group; otherwise, continue execution with the next instruction.

skipn
skipnp (*n* --) 0011 0010
Skip if Negative/Not Positive 0x32
1 (not neg) Mprefetch (neg) CPU-clocks

If *n* is negative (neither positive nor zero), skip the remainder of the instruction group and continue execution with the next instruction group; otherwise, continue execution with the next instruction.

skipnc (--) 0011 0111
Skip if Not Carry 0x35
1 (carry) Mprefetch (no carry) CPU-clocks

If carry is clear, skip the remainder of the instruction group and continue execution with the next instruction group; otherwise, continue execution with the next instruction.

skipnn
skipp (*n* --) 0011 0110
Skip if Not Negative/Positive 0x36
1 (neg) Mprefetch (not neg) CPU-clocks

If *n* is not negative (either positive or zero), skip the remainder of the instruction group and continue execution with the next instruction group; otherwise, continue execution with the next instruction.

skipnz (*n* --) 0011 0001
Skip if Not Zero 0x37
1 (zero) Mprefetch (non-zero) CPU-clocks

If *n* is not zero, skip the remainder of the instruction group and continue execution with the next instruction group; otherwise, continue execution with the next instruction.

skipz	(*n* --)	0011 0101
Skip if Zero		0x33
		1 (non-zero) Mprefetch (zero) CPU-clocks

If *n* is zero, skip the remainder of the instruction group and continue execution with the next instruction group; otherwise, continue execution with the next instruction.

split

Split Cell

split	(*n1* -- *n2* *n3*)	1001 1001
		0x99
		1 CPU-clock

Split *n1* into two parts so that the lower-half of *n1* is in the lower-half of *n2* and the upper-half of *n1* is in the lower-half of *n3*.

For example, if *n1* = 0x12345678 then *n2* = 0x5678 and *n3* = 0x1234.

st

Store Indirect to Memory

st [--r0]	(*n* --)	0110 0100 0x64 1+M CPU-clocks

Decrement r0 by four. Store the cell *n* into memory at the new address in r0. The two least-significant bits of the address are ignored and treated as zero.

st [--x]	(*n* --)	0110 1000 0x68 1+M CPU-clocks

Decrement x by four. Store the cell *n* into memory at the new address in x. The two least-significant bits of the address are ignored and treated as zero.

st [r0++]	(*n* --)	0110 0110 0x66 M CPU-clocks

Store the cell *n* into memory at the address in r0. Increment r0 by four. The two least-significant bits of the address are ignored and treated as zero.

st [r0]	(*n* --)	0110 0010 0x62 M CPU-clocks

Store the cell *n* into memory at the address in r0. The two least-significant bits of the address are ignored and treated as zero.

st [x++]	(*n* --)	0110 1001 0x69 M CPU-clocks

Store the cell *n* into memory at the address in x. Increment x by four. The two least-significant bits of the address are ignored and treated as zero.

st [x]	(*n* --)	0110 0001 0x61 M CPU-clocks

Store the cell *n* into memory at the address in x. The two least-significant bits of the address are ignored and treated as zero.

st [](*n addr* -- *n*)		0110 0000 0x60 M CPU-clocks

Store the cell *n* into memory at address *addr*. The two least-significant bits of the address are ignored and treated as zero.

step

Single-Step Processor

step(--)		0011 0100
	(L: *addr1* -- *addr2*)	0x34
		2M+2+inst CPU-clocks

Pop *addr1* from the local-register stack into pc and continue execution at *addr1* for one instruction. Then perform a call subroutine to the single-step trap location, 0x138. *addr2* is the address of the next instruction following *addr1*.

sto

Store Indirect to On-Chip Resource

sto []	(*n addr* -- *n*)	1011 0000
		0xB0
		1 CPU-clock

Store *n* into the on-chip resource register at address *addr*. The programmer must ensure that sto [] is not executed to access (even if not changed) any configuration register containing information for a memory group with a bus transaction in process. For valid values of *addr*, see *On-Chip Resource Registers*, page 89.

sto.i []	(*n bit_addr* -- *n*)	1011 0001
		0xB1
		1 CPU-clock

If *n* is non-zero, set the bit at the on-chip resource register address *bit_addr*; otherwise, clear the bit. For valid values of *addr*, see *On-Chip Resource Registers*, page 89.

sub

Subtract

sub (*n1 n2* -- *n3*) carry± 1100 1000
0xC8
1 CPU-clock

Subtract *n2* from *n1* leaving the difference *n3*. If computing the difference required a borrow, carry is set; otherwise, carry is cleared.

Equivalent to Java byte code isub.

Equivalent to ANS Forth word -.

subb

Subtract with Borrow

subb (*n1 n2* -- *n3*) carry± 1100 1010
0xCA
1 CPU-clock

Subtract *n2* and carry from *n1* leaving the difference *n3*. If computing the difference required a borrow, carry is set; otherwise, carry is cleared.

subexp

Subtract Exponents

subexp (*n1 n2* -- *n3 n4 n5*) 1101 0011
0xD3
2 CPU-clocks
(L: -- *addr*) only when trap processed 4+M CPU-clocks

Perform the following:
Exponent_Field(*n5*) = Exponent_Field(*n1*) - Exponent_Field(*n2*) + BIAS - 1
Sign_Bit(*n5*) = Sign_Bit(*n1*) XOR Sign_Bit(*n2*)
BIAS is 127 (0x3F800000 in bit position) for single precision and 1023 (0x3FF00000 in bit position) for double precision, as selected by fp_precision.

Compute as described above. Clear the exponent-field bits and sign bit and set the hidden bit of *n1* and *n2* giving *n3* and *n4*, respectively. *n5* is the result of the computation. After completion, if the exponent-field calculation result equaled or exceeded the maximum value of the exponent field (exponent result 255 for single, exponent result 2047 for double) an overflow exception is signaled. If the exponent-field calculation result is less than or equal to zero an underflow exception is signaled. When an exception is signaled, the exponent field of *n5* contains as low-order many bits of the result as it will hold.

testb

Test Bytes for Zero

testb	(*n* -- *n*)	carry±	1101 1001 0xD9 1 CPU-clock

If any byte of *n* is zero set carry, otherwise clear carry.

testexp

Test Exponent

testexp	(*n1 n2* -- *n1 n2*)	carry±	1101 0100 0xD4 1 CPU-clock
	(L: -- *addr*) only when trap processed		3+M CPU-clocks

Clear the GRS extension. If the exponent field in *n1* or *n2* is all zeros or all ones, an exponent exception is signaled and carry is set; otherwise, carry is cleared. The location of the exponent field depends on fp_precision.

xcg

Exchange

xcg (*n1 n2* -- *n2 n1*)	1011 0010 0xB2 1 CPU-clock

Exchange the top two operand stack cells.

Equivalent to Java byte code swap.

Equivalent to the ANS Forth words FSWAP, SWAP.

xor

Bitwise Exclusive OR

xor (*n1 n2* -- *n3*)	carry clear	1100 0011 0xC3 1 CPU-clock

Perform a bitwise EXCLUSIVE OR of *n1* and *n2* giving the result *n3*.

Equivalent to Java byte code ixor.

Equivalent to ANS Forth word XOR.

Mnemonic		Opcode
add	pc	bb
add		c0
adda		e8
addc		c2
addexp		d2
and		e1
bkpt		3c
br	*offset*	00…07
br	[]	4b
bz	*offset*	10…17
call	*offset*	08…0f
call	[]	4e
cmp		cb
copyb		d0
dbr	*offset*	18…1f
dec	ct,#1	c1
dec	#4	cd
dec	#1	cf
denorm		c5
di		b7
divu		de
ei		b6
eqz		e5
expdif		c4
extexp		db
extsig		dc
iand		e9
inc	#4	cc
inc	#1	ce
lcache		4d
ld	[]	40
ld	[x]	41
ld	[r0]	42
ld	[--r0]	44
ld	[r0++]	46
ld	[x++]	49
ld	[--x]	4a
ld.b	[]	48
ld.w	[]	4c
ldepth		9b
ldo	[]	96
ldo.i	[]	97
lframe		be
mloop		38
mloopc		39
mloopn		3a
mloopnc		3d
mloopnn		3e
mloopnz		3f
mloopp		3e
mloopz		3b
mulfs		d6
muls		d5
mulu		d7
mxm		df
neg		c9
nop		ea
norml		c7
normr		c6
notc		dd
or		e0
pop		b3
pop	ct	b4
pop	g0	50
pop	g1	51
pop	g2	52
pop	g3	53
pop	g4	54
pop	g5	55
pop	g6	56
pop	g7	57
pop	g8	58
pop	g9	59
pop	g10	5a
pop	g11	5b
pop	g12	5c
pop	g13	5d
pop	g14	5e
pop	g15	5f
pop	la	bd
pop	lstack	ba
pop	mode	b9
pop	r0	a0
pop	r1	a1
pop	r2	a2
pop	r3	a3
pop	r4	a4
pop	r5	a5
pop	r6	a6
pop	r7	a7
pop	r8	a8
pop	r9	a9
pop	r10	aa
pop	r11	ab
pop	r12	ac
pop	r13	ad
pop	r14	ae
pop	sa	bc
pop	x	b8
push		92
push	ct	94
push	g0	70
push	g1	71
push	g2	72
push	g3	73
push	g4	74
push	g5	75
push	g6	76
push	g7	77
push	g8	78
push	g9	79
push	g10	7a
push	g11	7b
push	g12	7c
push	g13	7d
push	g14	7e
push	g15	7f
push	mode	91
push	la	9d
push	lstack	9a
push	r0	80
push	r1	81
push	r2	82
push	r3	83
push	r4	84
push	r5	85
push	r6	86
push	r7	87
push	r8	88
push	r9	89
push	r10	8a
push	r11	8b
push	r12	8c
push	r13	8d
push	r14	8e
push	s0	92
push	s1	93
push	s2	9e
push	sa	9c
push	x	98
push.b	*# byte*	90
push.l	*# cell*	4f
push.n	#-7	29
push.n	#-6	2a
push.n	#-5	2b
push.n	#-4	2c
push.n	#-3	2d
push.n	#-2	2e
push.n	#-1	2f
push.n	#0	20
push.n	#1	21
push.n	#2	22
push.n	#3	23
push.n	#4	24
push.n	#5	25
push.n	#6	26
push.n	#7	27
push.n	#8	28
replb		da
replexp		b5
replw		eb
ret		6e
reti		6f
rev		e4
pnd		d1
scache		45
sdepth		9f
sexb		d8
sexw		95
sframe		bf
shift		ee
shiftd		ef
shl	#1	e2
shl	#8	ec
shld	#1	e6
shr	#1	e3
shr	#8	ed
shrd	#1	e7
skip		30
skipc		31
skipn		32
skipnc		35
skipnn		36
skipnp		32
skipnz		37
skipp		36
skipz		33
split		99
st	[]	60
st	[x]	61
st	[r0]	62
st	[--r0]	64
st	[r0++]	66
st	[--x]	68
st	[x++]	69
step		34
sto	[]	b0
sto.i	[]	b1
sub		c8
subb		ca
subexp		d3
testb		d9
testexp		d4
xcg		b2
xor		c3

Table 35 CPU Mnemonics and Opcodes (Mnemonic Order)

Opcode	Mnemonic		Opcode	Mnemonic		Opcode	Mnemonic		Opcode	Mnemonic	
00…07	br	*offset*	53	pop	g3	8d	push	r13	c7	norml	
08…0f	call	*offset*	54	pop	g4	8e	push	r14	c8	sub	
10…17	bz	*offset*	55	pop	g5	8f			c9	neg	
18…1f	dbr	*offset*	56	pop	g6	90	push.b	*# byte*	ca	subb	
20	push.n	#0	57	pop	g7	91	push	mode	cb	cmp	
21	push.n	#1	58	pop	g8	92	push	s0	cc	inc	#4
22	push.n	#2	59	pop	g9	93	push	s1	cd	dec	#4
23	push.n	#3	5a	pop	g10	94	push	ct	ce	inc	#1
24	push.n	#4	5b	pop	g11	95	sexw		cf	dec	#1
25	push.n	#5	5c	pop	g12	96	ldo	[]	d0	copyb	
26	push.n	#6	5d	pop	g13	97	ldo.i	[]	d1	rnd	
27	push.n	#7	5e	pop	g14	98	push	x	d2	addexp	
28	push.n	#8	5f	pop	g15	99	split		d3	subexp	
29	push.n	#-7	60	st	[]	9a	push	lstack	d4	testexp	
2a	push.n	#-6	61	st	[x]	9b	ldepth		d5	muls	
2b	push.n	#-5	62	st	[r0]	9c	push	sa	d6	mulfs	
2c	push.n	#-4	63			9d	push	la	d7	mulu	
2d	push.n	#-3	64	st	[--r0]	9e	push	s2	d8	sexb	
2e	push.n	#-2	65			9f	sdepth		d9	testb	
2f	push.n	#-1	66	st	[r0++]	a0	pop	r0	da	replb	
30	skip		67			a1	pop	r1	db	extexp	
31	skipc		68	st	[--x]	a2	pop	r2	dc	extsig	
32	skipn		69	st	[x++]	a3	pop	r3	dd	notc	
32	skipnp		6a			a4	pop	r4	de	divu	
33	skipz		6b			a5	pop	r5	df	mxm	
34	step		6c			a6	pop	r6	e0	or	
35	skipnc		6d			a7	pop	r7	e1	and	
36	skipnn		6e	ret		a8	pop	r8	e2	shl	#1
36	skipp		6f	reti		a9	pop	r9	e3	shr	#1
37	skipnz		70	push	g0	aa	pop	r10	e4	rev	
38	mloop		71	push	g1	ab	pop	r11	e5	eqz	
39	mloopc		72	push	g2	ac	pop	r12	e6	shld	#1
3a	mloopn		73	push	g3	ad	pop	r13	e7	shrd	#1
3b	mloopz		74	push	g4	ae	pop	r14	e8	adda	
3c	bkpt		75	push	g5	af			e9	iand	
3d	mloopnc		76	push	g6	b0	sto	[]	ea	nop	
3e	mloopnn		77	push	g7	b1	sto.i	[]	eb	replw	
3e	mlooppp		78	push	g8	b2	xcg		ec	shl	#8
3f	mloopnz		79	push	g9	b3	pop		ed	shr	#8
40	ld	[]	7a	push	g10	b4	pop	ct	ee	shift	
41	ld	[x]	7b	push	g11	b5	replexp		ef	shiftd	
42	ld	[r0]	7c	push	g12	b6	ei		f0		
43			7d	push	g13	b7	di		f1		
44	ld	[--r0]	7e	push	g14	b8	pop	x	f2		
45	scache		7f	push	g15	b9	pop	mode	f3		
46	ld	[r0++]	80	push	r0	ba	pop	lstack	f4		
47			81	push	r1	bb	add	pc	f5		
48	ld.b	[]	82	push	r2	bc	pop	sa	f6		
49	ld	[x++]	83	push	r3	bd	pop	la	f7		
4a	ld	[--x]	84	push	r4	be	lframe		f8		
4b	br	[]	85	push	r5	bf	sframe		f9		
4c	ld.w	[]	86	push	r6	c0	add		fa		
4d	lcache		87	push	r7	c1	dec	ct,#1	fb		
4e	call	[]	88	push	r8	c2	addc		fc		
4f	push.l	*# cell*	89	push	r9	c3	xor		fd		
50	pop	g0	8a	push	r10	c4	expdif		fe		
51	pop	g1	8b	push	r11	c5	denorm		ff		
52	pop	g2	8c	push	r12	c6	normr				

Table 36 CPU Mnemonics and Opcodes (Opcode Order)

Interrupt Controller

The Interrupt Controller (INTC) allows multiple requests to gain, in an orderly and prioritized manner, the attention of the CPU. The INTC supports up to eight prioritized interrupt requests. Interrupts are received from the bit inputs through ioin.

Resources

The INTC consists of several registers and associated control logic. Interrupt zero, which corresponds to bit zero of the registers, has the highest priority; interrupt seven, which corresponds to bit seven of the registers, has the lowest priority. The INTC and related registers include:

- Bit input register, ioin: bit inputs configured as interrupt requests or general bit inputs. See Figure 11.
- Interrupt pending register, ioip: indicates which interrupts have been recognized, but are waiting to be prioritized and serviced. See Figure 12.
- Interrupt under service register, ioius: indicates which interrupts are currently being serviced. See Figure 13.
- Interrupt enable register, ioie: indicates which ioin bits are to be recognized as interrupt requests. See Figure 15.

The bit inputs are low true used as interrupt requests or as directly readable bit inputs. Interrupt progress status is read as low true in ioin and as high true in ioie and ioius.

Operation

An interrupt request can arrive from a zero bit in ioin, typically from an external input low, or from the CPU writing the bit low. Interrupt request zero comes from ioin bit zero; interrupt request one comes from ioin bit one, the other interrupt requests are similarly assigned.

Associated with each of the eight interrupt requests is an interrupt service routine (ISR) executable-code vector located in memory. See Figure 3. A single ISR executable-code vector for a given interrupt request is used for all requests on that interrupt. It is programmed to contain executable code, typically a branch to the ISR.

Interrupt Request Servicing

When an interrupt request occurs, the corresponding bit in ioip is set, and the interrupt request is now a *pending interrupt*. Pending interrupts are prioritized each CPU-clock cycle. The interrupt_en bit in mode holds the current global interrupt enable state. It can be set with the CPU enable-interrupt instruction, ei; cleared with the disable-interrupt instruction, di; or changed by modifying mode. Globally disabling interrupts allows all interrupt requests to reach ioip, but prevents the pending interrupts in ioip from being serviced.

When interrupts are enabled, interrupts are recognized by the CPU between instruction groups, just before the execution of the first instruction in the group. This allows short, atomic, uninterruptable instruction sequences to be written easily without having to save, restore, and manipulate the interrupt state. The stack architecture allows interrupt service routines to be executed without requiring registers to be explicitly saved, and the stack caches minimize the memory accesses required when making additional register resources available.

If interrupts are globally enabled and the highest-priority ioip bit has a higher priority than the highest-priority ioius bit, the highest-priority ioip bit is cleared, the corresponding ioius bit is set, and the CPU is interrupted just before the next execution of the first instruction in an instruction group. This nests the interrupt servicing, and the pending interrupt is now the current *interrupt under service*. The ioip bits are not considered for interrupt servicing while interrupts are globally disabled, or while none of the ioip bits has a higher priority than the highest-priority ioius bit.

Unless software modifies ioius, the current interrupt under service is represented by the highest-priority ioius bit currently set. reti is used at the end of ISRs to clear the highest-priority ioius bit that is set and to return to the interrupted program. If the interrupted program was a lower-priority interrupt service routine, this effectively “unnests” the interrupt servicing.

Recognizing Interrupts

An ioin bit is configured to recognize an interrupt request source if the corresponding ioie bit is set. Once a zero reaches ioin, it is available to request an interrupt. An interrupt request is forced in software by clearing the corresponding ioin bit or by setting the corresponding ioip bit. Individually disabling an interrupt request by clearing its ioie bit prevents a corresponding zero bit in ioin from being recognized.

While an interrupt request is being processed, until its ISR terminates by executing reti, the corresponding ioin bit is not zero-persistent and follows the sampled level of the external input pin. Specifically, for a given interrupt request, while its ioie bit is set, and its ioip bit or ioius bit is set, its ioin bit is not zero-persistent. This effect can be used to disable zero-persistent behavior on non-interrupting bits. See *Zero Persistent*

ISR Processing

When an interrupt request is recognized by the CPU, a call to the corresponding ISR executable-code vector is performed, and interrupts are blocked until an instruction that begins in byte one of an instruction group is executed. To service an interrupt without being interrupted by a higher-priority interrupt:

- the ISR executable-code vector typically contains a four-byte branch, and
- the first instruction group of the interrupt service routine must globally disable interrupts. See the code example in Table 37.

```
; Interrupt Vectors

    .quad    4
    .text    vectors        ; org 0x100 set in linker

    br       int_0_ISR      ; highest-priority ISR
    br       int_1_ISR
    ...
    br       int_7_ISR      ; lowest-priority ISR
    ...

    .text    ISRs           ; org set in linker file

int_0_ISR::
    push     mode           ; save carry
    ; This ISR can't be interrupted because int 0
    ; has the highest priority.
    ...
    pop      mode           ; restore carry
    reti
    ...

int_A_ISR::
    push     mode           ; save carry
    ...
    ; This ISR can be interrupted by a higher
    ; priority interrupt.
    pop      mode
    reti
    ...

int_B_ISR::
    push     mode           ; save carry & ei state
    di
    ...
    ; Don't allow this ISR to be interrupted at all.
    ...
    ; ensure return before interrupts re-enabled
    .quad    2
    pop      mode
    reti
    ...

int_C_ISR::
    push     mode           ; save carry & ei state
    pop      lstack         ; place accessible
    di
    ; Don't allow this critical part of the ISR to be
    ; interrupted.
    ...
    push     r0
    pop      mode           ; restore ei state
    ...
    ; ISR can be interrupted by higher-priority
    ; interrupts now
    ...
    push     lstack
    pop      mode           ; restore carry
    reti
    ...
```

Table 37 Code Example: ISR Vectors

If interrupts are left globally enabled during ISR processing, a higher-priority interrupt can interrupt the CPU during processing of the current ISR. This allows devices with more immediate servicing requirements to be serviced promptly even when frequent interrupts at many priority levels are occurring.

Note that there is a delay of one CPU-clock cycle between the execution of ei, di, or pop mode and the change in the global interrupt enable state taking effect. To ensure the global interrupt enable state change takes effect before byte zero of the next instruction group, the state-changing instruction must not be the last instruction in the current instruction group.

If the global interrupt enable state is to be changed by the ISR, the prior global interrupt enable state can be saved with push mode and restored with pop mode within the ISR. Usually a pop mode, reti sequence is placed in the same instruction group at the end of the ISR to ensure that reti is executed, and the local-register stack unnests, before another interrupt is serviced. Since the return address from an ISR is always to byte zero of an instruction group (because of the way interrupts are recognized), another interrupt can be serviced immediately after execution of reti. See the code example in Table 37.

As described above for processing ISR executable-code vectors, interrupt requests are similarly blocked during the execution of all traps. This allows software to prevent, for example, further data from being pushed on the local-register stack due to interrupts during the servicing of a local-register-stack overflow exception. When resolving concurrent trap and interrupt requests, interrupts have the lowest priority.

Bit Inputs

Eight external bit inputs are available in bit input register ioin. They are shared for use as interrupt requests and as bit inputs for general use by the CPU.

Resources

The bit inputs consist of several registers, package pins, and associated input sampling circuitry. These resources include:

- Bit input register, ioin: bit inputs configured as interrupt requests or general bit inputs. See Figure 11.
- Interrupt enable register, ioie: indicates which ioin bits are to be recognized as interrupt requests. See Figure 15.
- Interrupt pending register, ioip: indicates which interrupts have been recognized, but are waiting to be prioritized and serviced. See Figure 12.
- Interrupt under service register, ioius: indicates which interrupts are currently being serviced. See Figure 13.
- Bit input pins, $\overline{\text{IN}}$[7:0].

Input Sampling

The bit inputs are sampled from $\overline{\text{IN}}$[7:0] every CPU-clock cycle and clocked into the IOIN register.

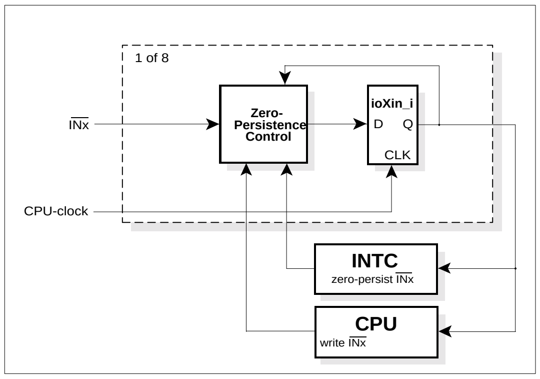


Figure 10 Bit Input Block Diagram

Zero Persistent

The bit inputs reaching ioin are normally zero-persistent. That is, once an ioin bit is zero, it stays zero regardless of the bit state at subsequent samplings until the bit is “consumed” and released, or is written with a one by the CPU. Zero-persistent bits have the advantage of both edge-sensitive and level-sensitive inputs, without the noise susceptibility and non-shareability of edge-sensitive inputs. Under certain conditions during ioin interrupt servicing, the ioin bits are not zero-persistent. An effect of the INTC can be used to disable zero-persistent behavior on the bits. See *General-Purpose Bits* below.

The code examples assume both zero persistence and input sampling. When both zero persistence and input sampling are disabled the inputs read read in the same manner and behave conventionally.

```
; Disable zero-persistence for bit input 7
      push.n   #-1         ; true flag

      push.b   #io7ius_i
      sto.i    []          ; set under service bit

      push.b   #io7ie_i
      sto.i    []          ; enable interrupt
      pop                  ; discard flag
      ...
```

Table 38 Code Example: Bit Input Without Zero-Persistence

Interrupt Usage

An ioin bit is configured as an interrupt request source when the corresponding ioie bit is set. While an interrupt request is being processed, until its ISR terminates by executing reti, the corresponding ioin bit is not zero-persistent and follows the sampled level of the external input. Specifically, for a given interrupt request, while its ioie bit is set, and its ioip bit or ioius bit is set, its ioin bit is not zero-persistent. This effect can be used to disable zero-persistent behavior on non-interrupting bits (see below).

General-Purpose Bits

If an ioin bit is not configured for interrupt requests then it is a zero-persistent general-purpose ioin bit. Alternatively, by using an effect of the INTC, general-purpose ioin bits can be configured without zero-persistence. Any bits so configured should be the lowest-priority ioin bits to prevent blocking a lower-priority interrupt. They are configured by setting their ioie and ioius bits. The ioius bit prevents the ioin bit from zero-persisting and from being prioritized and causing an interrupt request. See the code example in Table 38.

CPU Usage

Bits in ioin are read and written by the CPU as a group with ldo [ioin] and sto [ioin], or are read and written individually with ldo.i [ioXin_i] and sto.i [ioXin_i]. Writing zero bits to ioin has the same effect as though the external bit inputs had transitioned low for one sampling cycle, except that there is no sampling delay. This allows software to simulate events such as external interrupt requests. Writing one bits to ioin, unlike data from external inputs when the bits are zero-persistent, releases persisting zeros to accept the current sample. The written data is available immediately after the write completes. The CPU can read ioin at any time, without regard to the designations of the ioin bits, and with no effect on the state of the bits. The CPU does not consume the state of ioin bits during reads. See the code examples in Table 39.

```
; Read current state of zero-persistent input pins.
; (Assumes pkgio is set, and bits are zero-persistent)

; Assume we just tickled a device and we want to
; see if it just responded, but we have the bits
; configured as zero-persistent. The sample interval
; of four CPU-clock cycles and the sample holding
; delay of four CPU-clock cycles means there is a
; worst-case delay of eight CPU-clock cycles before
; the data is available in ioin. So...

    ; Put programming to tickle device here...

    nop                 ; wait the delay time
    nop
    nop
    nop
    nop
    nop                 ; 6 here, two below

; Read last sampled state of all zero-persistent
; bit inputs (Assumes all bits are configured as
; zero-persistent)

    push.n   #-1        ; all ones for all bits (7)

    push.n   #ioin      ; (CPU-clock cycle # 8)
                        ; ...data is now available
                        ; to ioin.
    sto  []             ; Temporarily remove
                        ; persistence, latest
                        ; sample latches,
    pop                 ; discard -1

    push.n   #ioin
    ldo  []             ; get last sample
    ...
```

Table 39 Code Example: CPU Usage of Bit Inputs

To perform a “real-time” external-bit-input read on zero-persistent bits, ones bits are written to the bits of interest in ioin before reading ioin. This releases any persisting zeros, latches the most recently resolved sample, and reads that value. Bits that are not configured as zero-persistent do not require this write. Note that any value read can be as much as two worst-case sample delays old. To read the values currently on the external inputs requires waiting two worst-case sample delays for the values to reach ioin. See the code example in Table 40.

```
; Force service on bit 5 (Interrupt or DMA, as
; configured)

    push.n  #0          ; false flag

    push.n  #io5in_i
    sto.i   []          ; clear input bit
    pop                 ; discard flag
    ...

; Read last sampled state of zero-persistent bit
; inputs. (Assumes all bits are configured as
; zero-persistent).

    push.n  #-1         ; all ones for all bits

    push.n  #ioin
    sto  []             ; temporarily remove
                        ; persistence, latest
                        ; sample latches,
    pop                 ; discard -1

    push.n  #ioin
    ldo  []             ; get last sample
    ...
```

Table 40 Code Example: CPU “Real-Time” Bit Input Read

Bit Outputs

Eight general-purpose bit outputs can be set high or low by the CPU. The bits are available in the bit output register, ioout.

Resources

The bit outputs consist of a register and pins. These resources include:

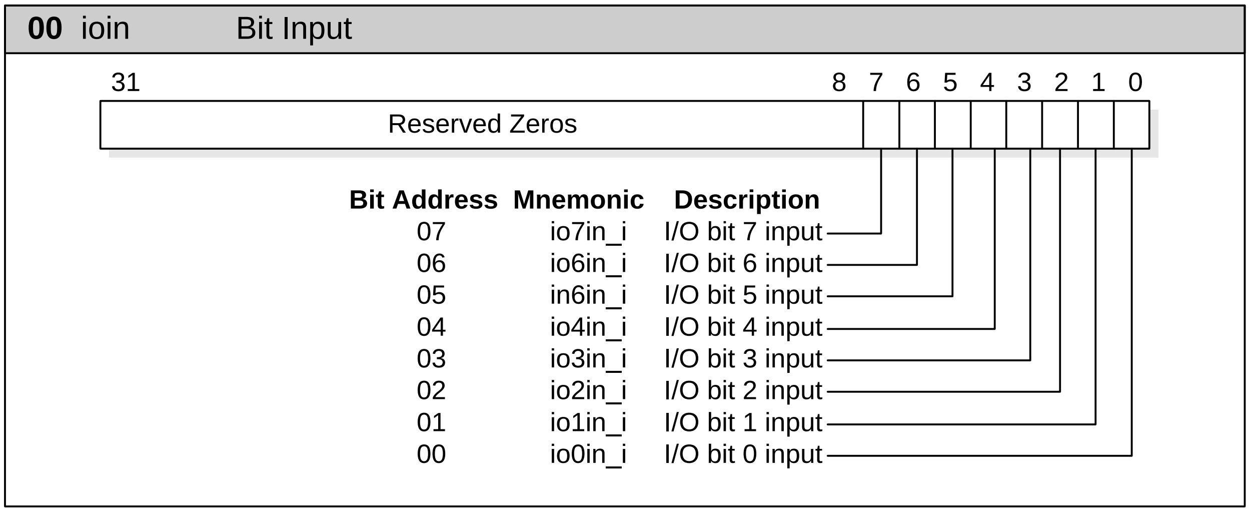
- Bit output register, ioout: bits that were last written by the CPU. See Figure 15.
- Bit outputs, out[7:0]

On-Chip Resource Registers

The on-chip resource registers comprise portions of various functional areas on the CPU including the CPU, INTC, and bit inputs. The registers are addressed from the CPU in their own address space using the instructions ldo and sto at the register level, or ldo. and sto. at the bit level (for those registers that have bit addresses). On other processors, resources of this type are often either memory-mapped or opcode-mapped. By using a separate address space for these resources, the normal address space remains uncluttered, and opcodes are preserved. Except as noted, all registers are readable and writable. Areas marked "Reserved Zeros" contain no programmable bits and always return zero. Areas marked "Reserved" contain unused programmable bits. Both areas might contain functional programmable bits in the future.

The first several registers are bit addressable in addition to being register addressable. This allows the CPU to modify individual bits without corrupting other bits that might be changed concurrently by INTC logic.

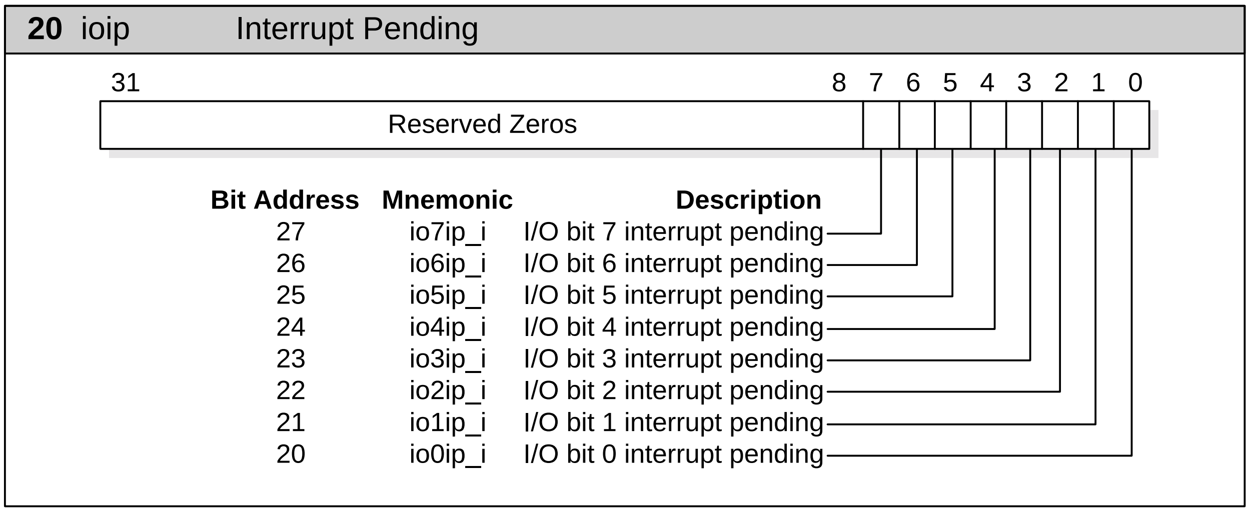
The bits are read and written by the CPU as a group with ldo [ioout] and sto [ioout], or are read and written individually with ldo.i [ioXout_i] and sto.i [ioXout_i]. When written, the new values are available immediately after the write completes.



Bit Address	Mnemonic	Description
07	io7in_i	I/O bit 7 input
06	io6in_i	I/O bit 6 input
05	in6in_i	I/O bit 5 input
04	io4in_i	I/O bit 4 input
03	io3in_i	I/O bit 3 input
02	io2in_i	I/O bit 2 input
01	io1in_i	I/O bit 1 input
00	io0in_i	I/O bit 0 input

Figure 11 Bit Input Register

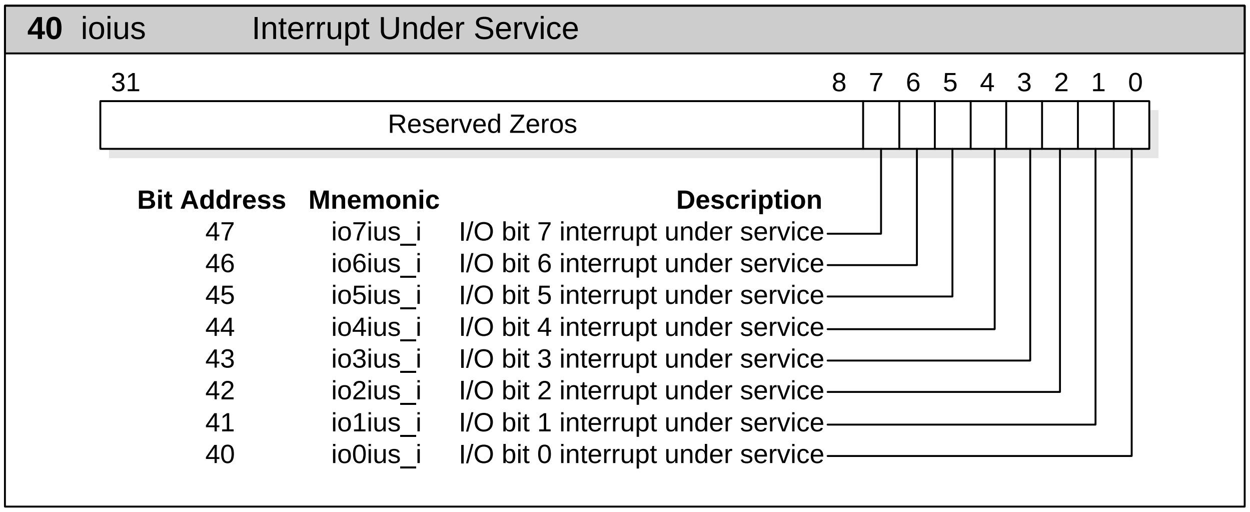
Contains sampled data from inputs[7:0]. ioin is the source of inputs for all consumers of bit inputs. Bits are zero-persistent: once a bit is zero in ioin it stays zero until consumed by the INTC, or written by the CPU with a one. Under certain conditions bits become not zero-persistent. See *Bit Inputs*. The bits can be individually read, set and cleared to prevent race conditions between the CPU and the interrupt controller logic.



Bit Address	Mnemonic	Description
27	io7ip_i	I/O bit 7 interrupt pending
26	io6ip_i	I/O bit 6 interrupt pending
25	io5ip_i	I/O bit 5 interrupt pending
24	io4ip_i	I/O bit 4 interrupt pending
23	io3ip_i	I/O bit 3 interrupt pending
22	io2ip_i	I/O bit 2 interrupt pending
21	io1ip_i	I/O bit 1 interrupt pending
20	io0ip_i	I/O bit 0 interrupt pending

Figure 12 Interrupt Pending Register

Contains interrupt requests that are waiting to be serviced. Interrupts are serviced in order of priority (0 = highest, 7 = lowest). An interrupt request from an I/O-channel transfer or from int occurs by the corresponding pending bit being set. Bits can be set or cleared to submit or withdraw interrupt requests. When an ioip bit and corresponding ioie bit are set, the corresponding ioin bit is not zero-persistent. See *Interrupt Controller*. The bits can be individually read, set and cleared to prevent race conditions between the CPU and the interrupt controller logic.

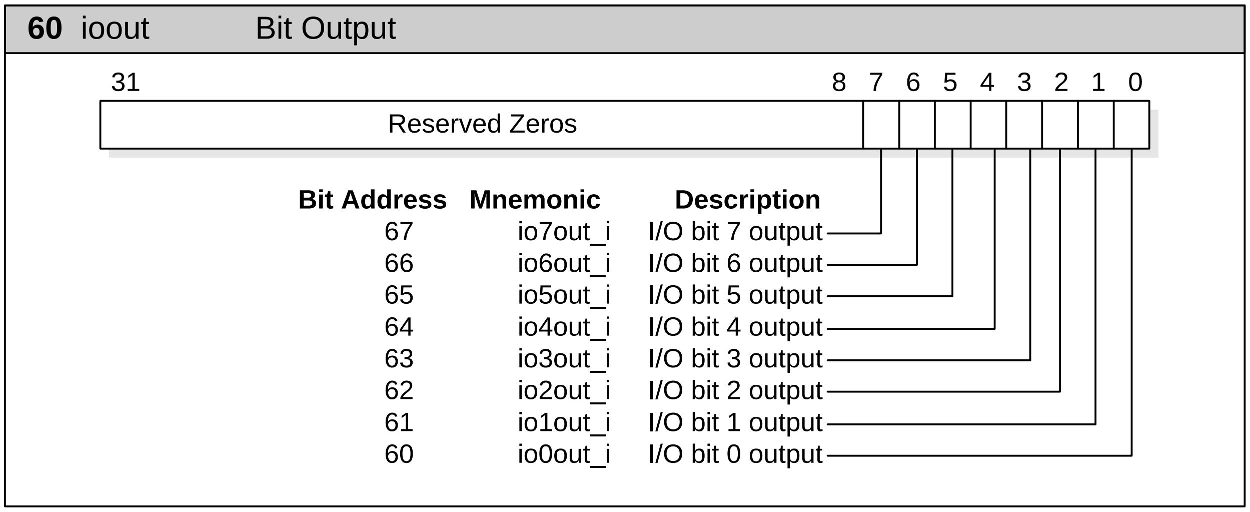


Bit Address	Mnemonic	Description
47	io7ius_i	I/O bit 7 interrupt under service
46	io6ius_i	I/O bit 6 interrupt under service
45	io5ius_i	I/O bit 5 interrupt under service
44	io4ius_i	I/O bit 4 interrupt under service
43	io3ius_i	I/O bit 3 interrupt under service
42	io2ius_i	I/O bit 2 interrupt under service
41	io1ius_i	I/O bit 1 interrupt under service
40	io0ius_i	I/O bit 0 interrupt under service

Figure 13 Interrupt Under Service Register

Contains the current interrupt service request and those that have been temporarily suspended to service a higher-priority request. When an ISR executable-code vector for an interrupt request is executed, the ioius bit for that interrupt request is set and the corresponding ioip bit is cleared. When an ISR executes reti, the highest-priority interrupt under-service bit is cleared. The bits are used to prevent interrupts from interrupting higher-priority ISRs. When an ioius bit and corresponding ioie bit are set, the corresponding ioin bit is not zero-persistent. See *Interrupt Controller*.

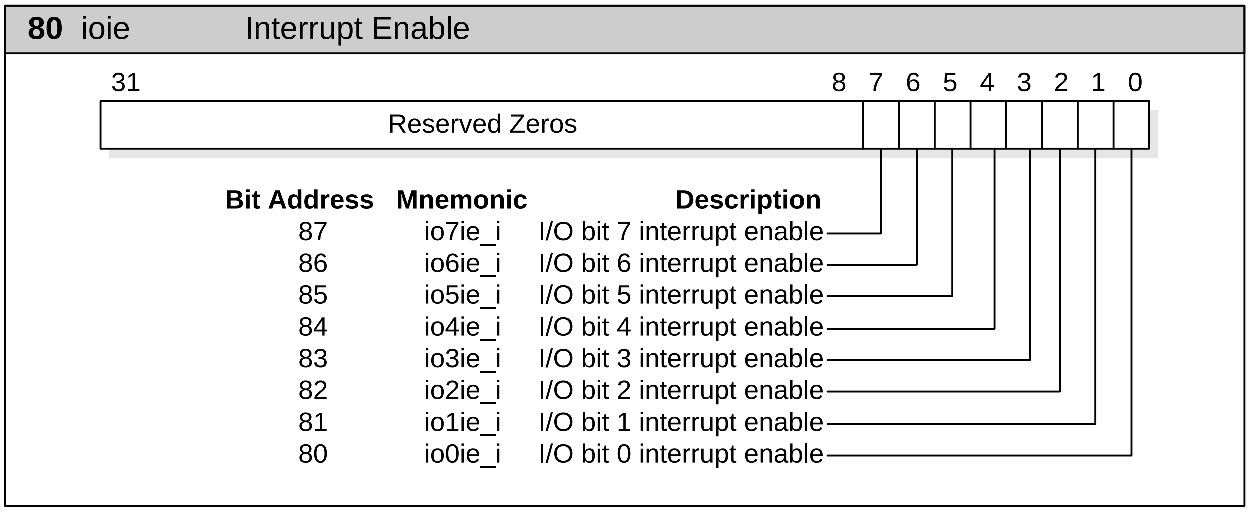
The bits can be individually read, set and cleared to prevent race conditions between the CPU and INTC logic.



Bit Address	Mnemonic	Description
67	io7out_i	I/O bit 7 output
66	io6out_i	I/O bit 6 output
65	io5out_i	I/O bit 5 output
64	io4out_i	I/O bit 4 output
63	io3out_i	I/O bit 3 output
62	io2out_i	I/O bit 2 output
61	io1out_i	I/O bit 1 output
60	io0out_i	I/O bit 0 output

Figure 14 Bit Output Register

Contains the bits from CPU bit-output operations. Bits appear on OUT[7:0] immediately after writing. The bits can be individually read, set and cleared.



Bit Address	Mnemonic	Description
87	io7ie_i	I/O bit 7 interrupt enable
86	io6ie_i	I/O bit 6 interrupt enable
85	io5ie_i	I/O bit 5 interrupt enable
84	io4ie_i	I/O bit 4 interrupt enable
83	io3ie_i	I/O bit 3 interrupt enable
82	io2ie_i	I/O bit 2 interrupt enable
81	io1ie_i	I/O bit 1 interrupt enable
80	io0ie_i	I/O bit 0 interrupt enable

Figure 15 Interrupt Enable Register

Allows a corresponding zero bit in ioin to request the corresponding interrupt service. When an enabled interrupt request is recognized, the corresponding ioip bit is set and the corresponding ioin bit is no longer zero-persistent. See *Interrupt Controller*, page 79. The bits can be individually read, set and cleared. Bit addressability for this register is an artifact of its position in the address space, and does not imply any race conditions on this register can exist.

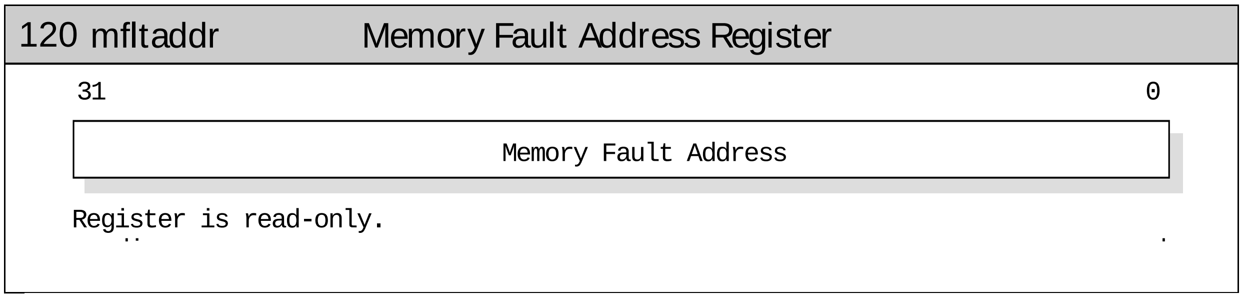


Figure 16 Memory Fault Address Register

When a memory page-fault exception occurs during a memory read or write, mfltaddr contains the address that caused the exception. The contents of mfltaddr and mfltdata are latched until the first read of mfltaddr after the fault. After reading mfltaddr, the data in mfltaddr and mfltdata are no longer valid.

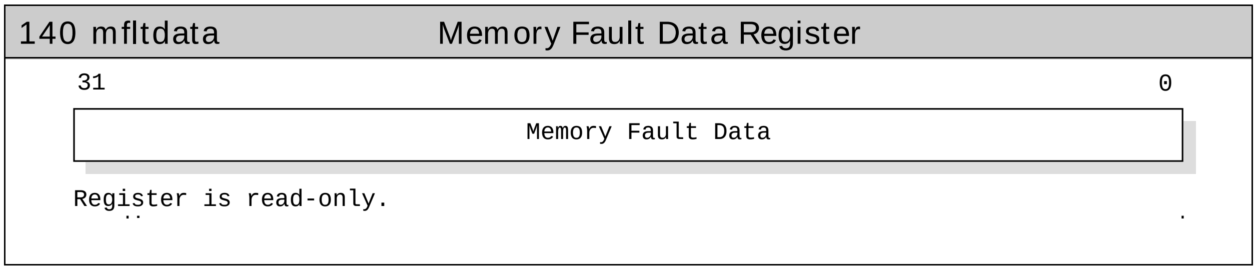


Figure 17 Memory Fault Data Register

When a memory page-fault exception occurs during a memory write, mfltdata contains the data to be stored at mfltaddr. The contents of mfltaddr and mfltdata are latched until the first read of mfltaddr after the fault.

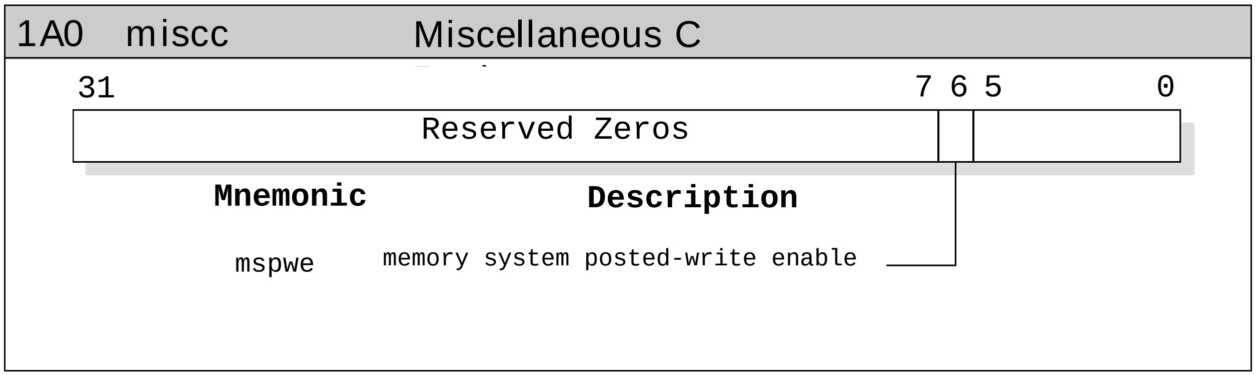


Figure 18 Miscellaneous C Register

If set, enables a one-level CPU posted-write buffer, which allows the CPU to continue executing after a write to memory occurs. A posted write has precedence over subsequent CPU reads to maintain memory coherency. If clear, the CPU must wait for writes to complete before continuing.

Onchip Resource Register values upon CPU reset:

Table 40 provides the values of all of the onchip registers upon the occurrence of a reset event to the IGNITE CPU.

Address	Register	Description	Initial value
000	ioin	Bit Input Register	0000 00FF
020	ioip	Interrupt Pending Register	0000 0000
040	ioius	Interrupt Under Service Register	0000 0000
060	ioout	Bit Output Register	0000 00FF
080	ioie	Interrupt enable Register	0000 0000
120	mfltaddr	Memory Fault Address Register	xxxx xxxx
140	mfltdata	Memory Fault Data Register	xxxx xxxx
1A0	misc	Miscellaneous C Register	0000 0000

Table 40 Resource Register Reset Values

This section of the document provides all of the information a designer will require designing the logic to interface with memory and other peripheral devices for the Ignite CPU processor core embodied as a net-list in EDIF file format.

Bus Interface

The bus interface of the Ignite CPU is relatively simple. There are no special requirements other than depicted in the timing diagrams.

Posted Writes

The Ignite CPU supports a one-deep posted write to allow it to continue execution while the write to the external device is in progress. Typically CPU execution will subsequently stall waiting for the next bus operation to start.

SYMBOL	TYPE	DESCRIPTION
*RESET	I	**RESET:** Asserting this signal (active low) causes the CPU to initialize all internal registers and begin execution at the hardware reset location
CLOCK	I	**CLOCK INPUT:** This is the clock input to the processor provided by a clock source. The processor runs at the same frequency of the clock input
MAR [31:0]	O	**ADDRESS OUTPUT:** This is the 32-bits of address bus produced by the processor. The address bus is non-multiplexed
MDR [31:0]	I/ O	**DATA OUTPUT:** This is 32-bits of data bus produced by the processor. The data bus is non-multiplexed and conforms to big-endian standard
*INB [7:0]	I	**BIT INPUTS:** These active low signals act as general or interrupts inputs to the processor
OUTB [7:0]	O	**BIT OUTPUTS:** These byte signals acts as general-purpose outputs from the processor. These are bit programmable.
WR	O	**READ/WRITE:** This acts as the Read/Write signal produced by the processor. A logic HIGH serves as Write. A logic LOW serves as Read.
REQ	O	**REQUEST:** This output signal indicates the beginning of a read or write transfer cycle of the processor from an idle state
DVAL	I	**DATA VALID:** This input signal generated by external indicates the completion of a read or write transfer to the processor
*FAULTB	I	**MEMORY FAULT:** This active low input signal generated by external logic indicates a faulty memory location access by the processor

Table 41 Signal Descriptions

Reset *RESET, input

When asserted active (low), completely initializes the CPU. When de-asserted, CPU execution begins at the address 0x80000008. This signal is internally synchronized with the CPU clock.

The *Reset signal must stay activate for at least 4 clock cycles for the processor to reach its quiescent state.

`Clock CLOCK`, input

There is no phase lock loop built into the Ignite IP and therefore all operations within the Ignite IP run off this clock input Baring a few, all instructions run in a single cycle clock as mentioned in the Ignite Reference Manual.

`Address MAR [31:0]`, output

The address bus provides non-multiplexed address for current CPU bus access. The rising edge of `request` signal indicates the start of bus read/write transfer cycle, which also indicates a valid address on the bus.

The address remains valid until the end of the rising edge of the CPU clock following a data valid dval input going active. The two least-significant bits of the address are ignored when fetching or writing cell-wide data. The first valid address after a `reset` has been active is the CPU reset address.

Data MDR [31:0], input/output

Provides 32 bit data input when `write` is inactive. Provides 32 bit data output when `write` is active.
The rising edge of `Request` signal indicates valid write data.

The write data remains valid until the end of the rising edge of the CPU clock following a data valid dval input going active. For read operations the read data needs to meet the setup and hold time with respect to rising edge of CPU clock after Data valid signal dval goes active.

The interface to the **ignite_ip** EDIF file logic consists of a 32-bit data in bus **mdi<31:0>** and a 32-bit data out bus **mdo<31:0>.** The bi-directional pin driver of the FPGA combines these to form MDR <31:0>.

`Input`, INB [7:0], input

Bit inputs can be used for general-purpose inputs or as interrupt requests. These inputs are accessible by the CPU through **`ioin`** register. These inputs need to be synchronized with the CPU clock before presenting to the Ignite IP FPGA device.

Output, OUTB [7:0], output

Bit outputs for general-purpose use. These bits are accessible by the CPU through the **`ioout`** register**.**

`Read/Write WR, output`

When active, indicates that the current bus cycle is a write cycle. When inactive, indicates the current bus cycle is a read cycle. This signal is active concurrent with the REQ signal that signifies the start of a bus transfer cycle. This signal goes active at the rising edge of the CPU clock.

CPU data transfer state, REQ, output

This signal goes active at the rising edge of the CPU clock indicating the beginning of a bus transfer cycle.

Data Valid DVAL, output

This signal generated by external logic indicates to the Ignite CPU as to when it is time to complete the current bus transfer cycle. This active High signal is sampled by the rising edge of the CPU clock. If there is a pending bus cycle, then the CPU will immediately start the next transfer on the rising edge of the CPU clock.

Memory Fault *FAULTB, input

If the pin `*faultb` is asserted (active low), and memory fault traps are enabled, following a request at the beginning of a bus transfer cycle, then the CPU will immediately transfer execution to the memory fault trap location to handle the memory fault. This signal is provided by an external logic implementing a memory manager function. Memory fault traps are enabled by bit 27 of the mode register. The address and write-data that caused the memory fault saved in internal registers and are retrieved allowing memory fault recovery. The `*faultb` going active has a required setup time and should also be driven inactive after the invalid memory cycle completes. The memory manager generating the `*faultb` signal must also generate `dval` to complete the current cycle.

If `*faultb` is asserted, and memory fault traps are not enabled, operation will be unaffected, provided that `*faultb` is removed in a timely manner.

The `*faultb` signal might be generated by external logic because of either memory errors detected by parity circuitry or memory non-availability caused by memory page swapping.

Bus Interface

The bus interface for the Ignite CPU employs a very simple request/acknowledge protocol that has been the traditional mechanism for most embedded processors.

There are two modes of bus transaction that are intended for single and multiple access mode of access respectively.

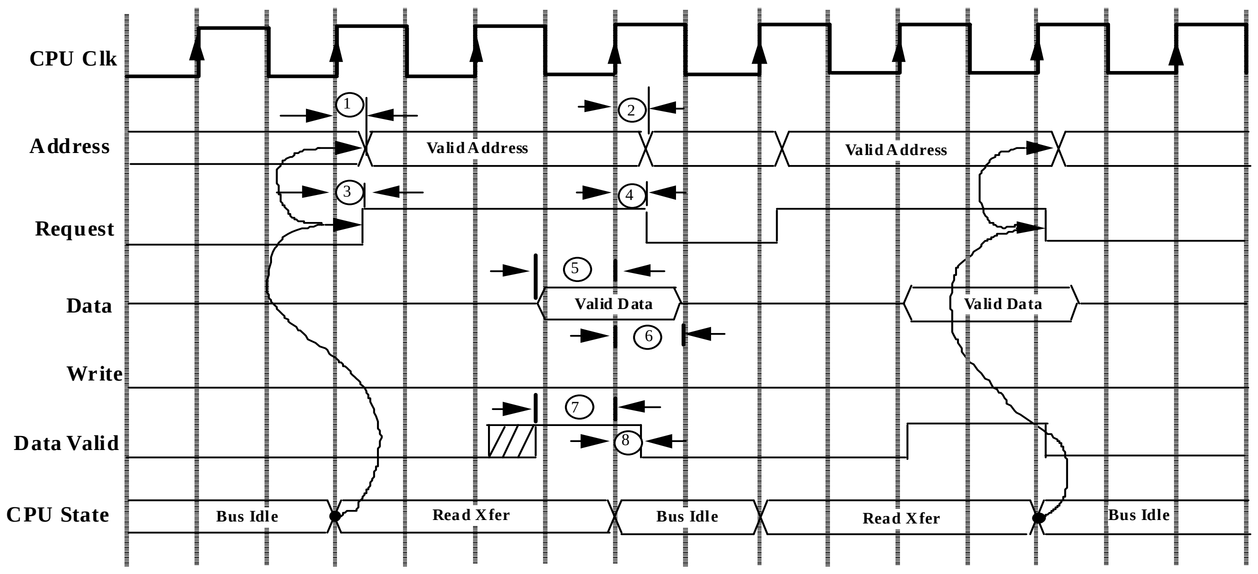
The Ignite processor IP is a completely synchronous design. All timing information will be stated with respect to the clock edge, period or duty cycle of the clock that it is operated from.

Timing Information

The timing specifications for the part as mentioned in the IP data sheet were derived post synthesis using TSMC library of parts for the 0.18-micron technology, and will be different for other technologies.

All output drivers will be specific to the user implementation.

All inputs have a setup time with respect to the clock input of the device. All outputs have a clock to output time delay referenced to the clock input of the device.



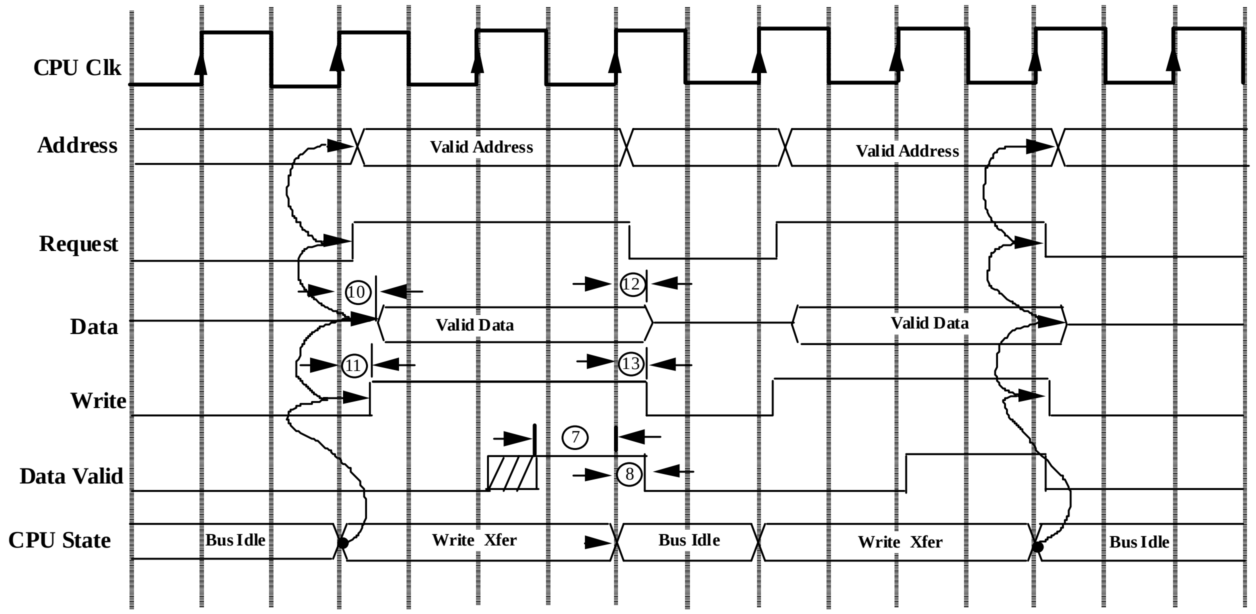
Ignite CPU Read

No	Symbol	Description	Min	Typical	Max	Notes
1	t_addrout	Address valid out from clock rise	T_{CHOH} Note 3 **or** T_{CHOL} Note 4			Foundry library specific Note2
2	t_addrinval	Address invalid from clock rise	T_{CHOH} Note 3 **or** T_{CHOL} Note 4			Foundry library specific Note2
3	t_reqvalout	Request valid out from clock rise	T_{CHOH} Note 3			Foundry library specific Note2
4	t_reqinval	Request invalid from clock rise	T_{CHOL} Note 4			
5	t_rdatasetup	Read Data setup	T_{IOOCK} Note 5			
6	t_rdatahold	Read Data Hold	$T_{IOHLDCK}$ Note 6			
7	t_dvalsetup	Data valid setup to clock rise	0.6T_clkperiod Note1			Meeting Min parameter assures 1 cycle memory access Note1
8	t_dvalhold	Data valid Hold	$T_{IOHLDCK}$ Note 6			

Table 42 CPU Read Timing Parameters

Notes:

Note1	T_clkperiod refers to the clock period of the CPU clock. This is an absolutely critical parameter to meet for 1 cycle memory access
Note 2	These parameters in this row are defined by the Foundry provided library for a specific semiconductor geometry and process
Note 3	This is the delay as specified by the component library for clock High to output High
Note 4	This is the delay as specified by the component library for clock High to output Low
Note 5	This is the Setup time before the clock active signal as specified by component library
Note 6	This is the Hold time after the clock active signal as specified by component library



Ignite CPU Write

No	Symbol	Description	Min	Typical	Max	Notes
10	t_dataout	Data valid out from clock rise	T_{CHOH} Note 3 **or** T_{CHOL} Note 4			Foundry library specific Note2
12	t_dataz	Data tri-state from clock rise	T_{IOCKP} Note 3 + T_{IOTHZ} Note7			Foundry library specific Note2
11	t_wrtvalout	Write valid out from clock rise	T_{CHOH} Note 3			Foundry library specific Note2
13	t_wrtinval	Write invalid from clock rise	T_{CHOL} Note 4			
7	t_dvalsetup	Data valid setup to clock rise	0.6T_clkperiod Note1			Meeting Min parameter assures 1 cycle memory access Note1
8	t_dvalhold	Data valid Hold	$T_{IOHLDCK}$ Note 6			

Table 43 CPU Write Timing Parameters

Notes:

Note1 T_clkperiod refers to the clock period of the CPU clock. This is an absolutely critical parameter to meet for 1 cycle memory access

Note 2 These parameters in this row are defined by the Foundry provided library for a specific semiconductor geometry and process

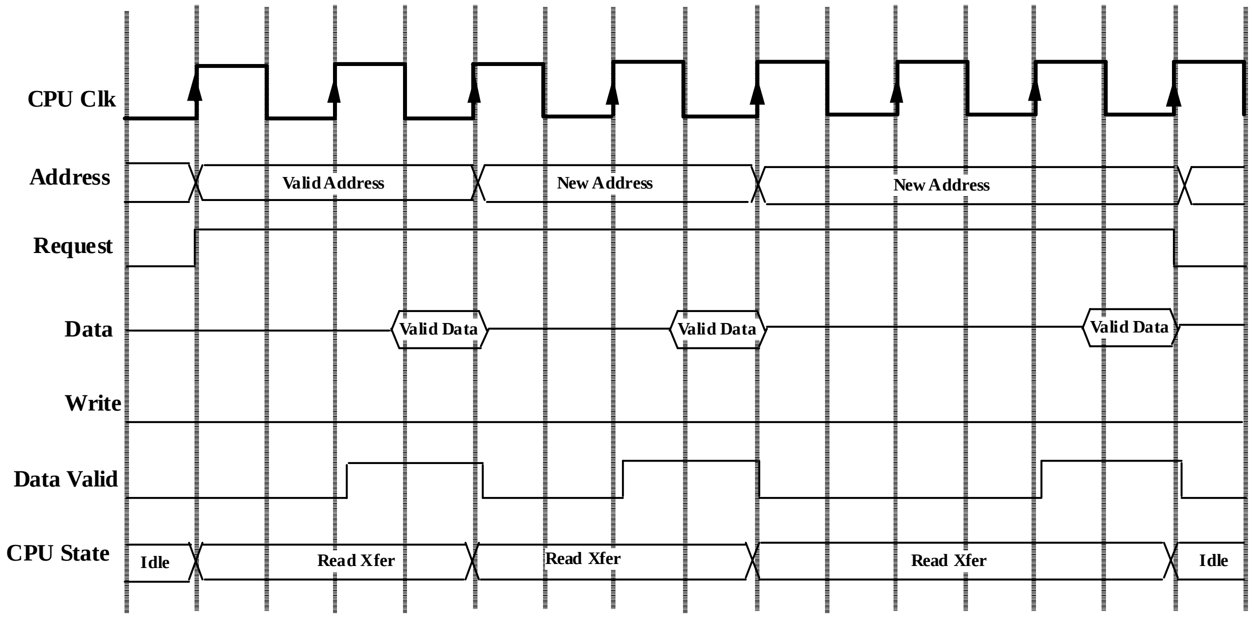
Note 3 This is the delay as specified by the component library for clock High to output High

Note 4 This is the delay as specified by the component library for clock High to output Low

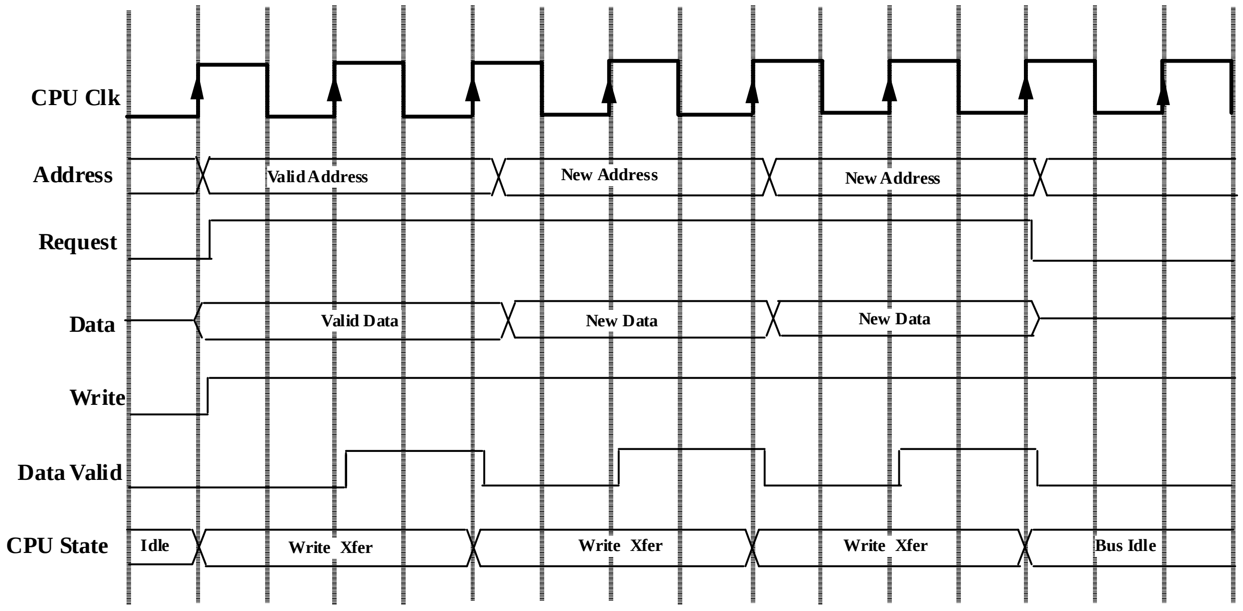
Note 5 This is the Setup time before the clock active signal as specified by component library

Note 6 This is the Hold time after the clock active signal as specified by component library

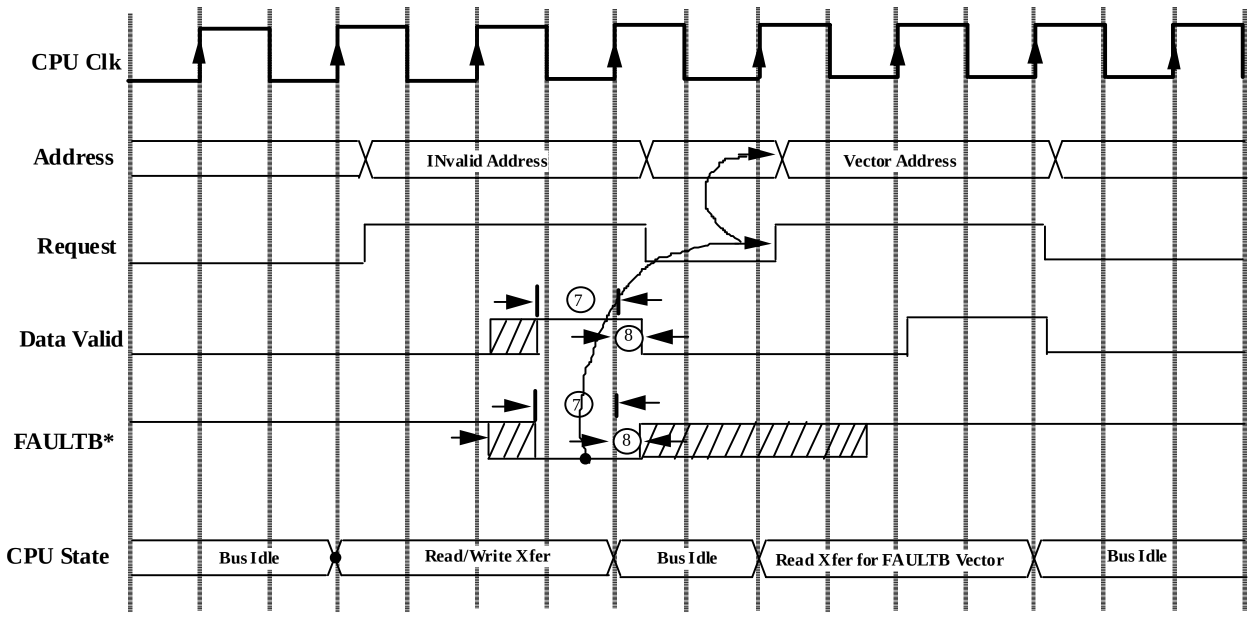
Note7 This is the input to high-impedance delay as specified by component library



Ignite CPU Multiple Access Read



Ignite CPU Multiple Access Write



Ignite Memory Fault

No	Symbol	Description	Min	Typical	Max	Notes
7	t_dvalsetup	Data valid setup to clock rise	0.6T_clkperiod Note1			Meeting Min parameter assures 1 cycle memory access Note1
8	t_dvalhold	Data valid Hold	$T_{IOHLDCK}$ Note 6			

Table 44 Memory Fault Operation Timing Parameters

Notes:

Note1 T_clkperiod refers to the clock period of the CPU clock. This is an absolutely critical parameter to meet for 1 cycle memory access

Note 6 This is the Hold time after the clock active signal as specified by component library