Stark

Table of Contents

Preface

Who This Book is For

This book is for the FPGA enthusiast who's looking to do a more complex project. It's advisable that one have a good background in digital electronics and computer systems before attempting a read. Examples are provided in the SystemVerilog language, it would be helpful to have some understanding of HDL languages. Finally, a lot about computer architecture is contained within these pages, some previous knowledge would also be helpful. If you're into electronics and computers as a

hobby FPGA's can be a lot of fun. This book primarily describes the Qupls2 ISA. It is for anyone interested in instruction set architectures.

About the Author

First a warning: I'm an enthusiastic hobbyist like yourself, with a ton of experience. I've spent a lot of time at home doing research and implementing several soft-core processors, almost maniacally. One of the first cores I worked on was a 6502 emulation. I then went on to develop the Butterfly32 core. Later the Raptor64. I have progressed slowly from the simple to the complex. I have about 25 years professional experience working on banking applications at a variety of language levels including assembler. So, I have some real-world experience developing complex applications. I also have a diploma in electronics engineering technology. Some of the cores I work on these days are too complex and too large to do at home on an inexpensive FPGA. I await bigger, better, faster boards yet to come. To some extent larger boards have arrived. The author is a bit wary of larger boards. Larger FPGAs increase build times by their nature.

Motivation

The author desired a CPU core supporting 128-bit floating-point operations for the precision. He also wanted a core he could develop himself. The simplest approach to supporting 128-bit floats is to use 128-bit wide registers, which leads to 128-bit wide busses in the CPU and just generally a 128-bit design. It was not the author's original goal to develop a 128-bit machine. There are good ways of obtaining 128-bit floating-point precision on 64-bit or even 32-bit machines, but it adds some complexity. Complexity is something the author must manage to get the project done and a flat 128-bit design is simpler.

Good single thread performance is also a goal.

Having worked on Qupls for several months, the author finally realized that it did not have very good code density. Having a reasonably good code density is desirable as it is unknown where the CPU will end up. Earlier designs were better in that regard. So, Qupls2 arrived and is a mix of the best from previous designs. Qupls2 aims to improve code density over earlier versions.

Some efficiency is being traded off for design simplicity. Some of the most efficient designs are 32-bit.

The processor presented here isn't the smallest, most efficient, and fastest RISC processor. It's also not a simple beginner's example. Those weren't my goals. Instead, it offers reasonable performance and hopefully design simplicity. It's also designed around the idea of using a simple compiler. Some operations like multiply and divide could have been left out and supported with software generated by a compiler rather than having hardware support. But I was after a simple compiler design. There's lots of room for expansion in the future. I chose a 64-bit design supporting 128-bit ops in part anticipating more than 4GB of memory available sometime down the road. A 64-bit architecture is doable in FPGA's today, although it uses two or more times the resources that a 32-bit design would.

History

Qupls2 is a work in progress beginning February 2025. It is a major re-write from earlier versions. Thor which originated from RiSC-16 by Dr. Bruce Jacob. RiSC-16 evolved from the Little Computer (LC-896) developed by Peter Chen at the University of Michigan. The author has tried to be innovative with this design borrowing ideas from many other processing cores.

Qupls's graphics engine originate from the ORSoC GFX accelerator core posted at opencores.org by Per Lenander, and Anton Fosselius.

Features of Qupls2

- Variable length instruction set with three sizes: 24/48/96-bit.
- Four way out-of-order superscalar operation
- Four operating modes, machine, hypervisor, supervisor, and user.
- 64-bit data path, support for 128-bit floats
- 16 (or more) entry re-order buffer
- 64 general purpose registers. The register file is unified; it may contain either integer or float data.
- Dedicated link registers.
- Independent control of sign for each register.
- Register renaming to remove dependencies, vector elements are also renamed.
- Dual operation instructions, Rt = Ra **op1** Rb **op2** Rc
- Standard suite of ALU operations, add subtract, compare, multiply and divide.
- Pair shifting instructions. Arithmetic right shift with rounding.
- Conditional branches with 19 effective displacement bits.

- 128 Entry Two-way TLB shared between data and code.

Programming Model

Register File

General Purpose Registers

The register file contains 64 64-bit general purpose registers.

The register file is *unified* and may hold either integer or floating-point values. The stack pointer is register 31.

Register r0 is special in that it always reads as a zero.

Register ABI

Regno	ABI	ABI Usage
0	0	Always zero – read only
1	A0	First argument / return value register
2	A1	Second argument / return value register
3	A2	Third argument register
4 to 8	A3 to A7	Argument registers
9 to 18	T0 to T9	Temporary register, caller save
19 to 27	S0 to S8	Saved register, register variables
28	LC	Loop counter
29	GP	Global Pointer – data
30	FP	Frame Pointer
31	SP	Stack pointer alias / Safe stack pointer (hidden from app)
32 to 35	xSP	Stack pointers for operating mode
36 to 39	MC0 to MC3	Micro-code temporaries
40	MCLR	Micro-code link register
41 to 43	LR1 to LR3	Subroutine link registers
44 to 63		high-order registers

Predicate Registers

Predicate registers are part of the general-purpose register file and may be manipulated using the same instructions as for other registers. Each bit of a predicate value corresponds to a byte in the target register. If the bit in the predicate register is set, then the corresponding byte of the target register is updated. Otherwise, the byte retains its value.

Predicate register #0 is preset to all ones, and read-only, which will allow all bytes of the target register to be updated. It is the default predicate if no predicate is supplied in the assembler code.

Code Address Registers

Many architectures have registers dedicated to addressing code. Almost every modern architecture has a program counter or instruction pointer register to identify the location of instructions. Many architectures also have at least one link register or return address register holding the address of the next instruction after a subroutine call. There are also dedicated branch address registers in some architectures. These are all code addressing registers.

It is possible to do an indirect method call using any register.

Link Registers

There are three registers in the Qupls2 ABI reserved for subroutine linkage. These registers are used to store the address after the calling instruction. They may be used to implement fast returns for three levels of subroutines or to used to call milli-code routines. The jump to subroutine, JSR, and branch to subroutine, BSR, instructions update a link register. The return from subroutine,. RTS, instruction is used to return to the next instruction. Note that the link register number is encoded into only two bits of the instruction.

Regno	ABI	Encode	ABI Usage
0	Zero	0	No linkage
36	LR1	1	Link register #1
37	LR2	2	Link register #2
38	LR3	3	Link register #3

Instruction Pointer

This register points to the currently executing instruction. The instruction pointer increments as instructions are fetched, unless overridden by another flow control

instruction. The instruction pointer may be set to any byte address. There is no alignment restriction. It is possible to write position independent code, PIC, using IP relative addressing.

SR - Status Register (CSR 0x?004)

The processor status register holds bits controlling the overall operation of the processor, state that needs to be saved and restored across interrupts. The bits have individual bit set / clear capability using the CSRRS, CSRRC instructions. Only the user interrupt enable bit is available in user mode, other bits will read as zero.

Bit		Usage
0	uie	User interrupt enable
1	sie	Supervisor interrupt enable
2	hie	Hypervisor interrupt enable
3	mie	Machine interrupt enable
4	die	Debug interrupt enable
5 to 10	ipl	Interrupt level
11	ssm	Single step mode
12	te	Trace enable
13 to 14	om	Operating mode
15 to 16	ps	Pointer size
17	ab	Absolute conditional branches
18	dbg	Debug mode
19	mprv	memory privilege
20 to 22	Swstk	Software stack
23		reserved
24 to 31	cpl	Current privilege level

CPL is the current privilege level the processor is operating at.

T indicates that trace mode is active.

OM processor operating mode.

PS: indicates the size of pointers in use. This may be one of 32, 64 or 128 bits.

AB: indicates that conditional branches should use absolute(1) or relative (0) addressing.

AR: Address Range indicates the number of address bits in use. 0 = near or short (32-bit) addressing is in use. When short addressing is in use only the low order 32-bit are significant and stored or loaded to or from the stack.

IPL is the interrupt mask level

MPRV Memory Privilege, indicates to use previous operating mode for memory privileges

Special Purpose Registers

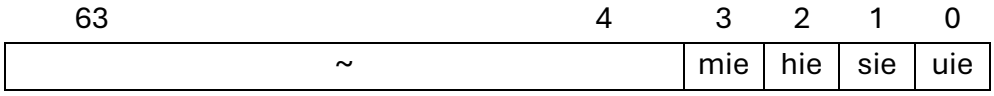
SC - Stack Canary (GPR 53)

This special purpose register is available in the general register file as register 53. The stack canary register is used to alleviate issues resulting from buffer overflows on the stack. The canary register contains a random value which remains consistent throughout the run-time of a program. In the right conditions, the canary register is written to the stack during the function's prolog code. In the function's epilog code, the value of the canary on stack is checked to ensure it is correct, if not a check exception occurs.

[U/S/H/M]_IE (0x?004)

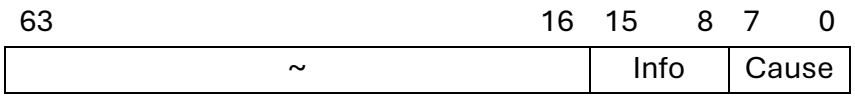
See status register.

This register contains interrupt enable bits. The register is present at all operating levels. Only enable bits at the current operating level or lower are visible and may be set or cleared. Other bits will read as zero and ignore writes. Only the lower four bits of this register are implemented. The bits have individual bit set / clear capability using the CSRRS, CSRRC instructions.



[U/S/H/M]_CAUSE (CSR- 0x?006)

This register contains a code indicating the cause of an exception or interrupt. The break handler will examine this code to determine what to do. Only the low order 16 bits are implemented. The high order bits read as zero and are not updateable. The info field, filled in by hardware, may supply additional information related to the exception.



U_REPBUF - (CSR – 0x008)

~~This register contains information needed for the REP instruction that must be saved and restored during context switches and interrupts. Note that the loop counter should also be saved.~~

~~127 112~~	~~121 48~~	~~47 44~~	~~43~~	~~42 40~~	~~39 8~~	~~7~~	~~6 0~~
~~Resv~~	~~pc~~	~~Resv2~~	~~V~~	~~ICnt~~	~~Limit~~	~~resv~~	~~Ins[15:9]~~

~~Pc: (64 bits) the address of the instruction following the REP~~
~~V: REP valid bit, 1 only if a REP instruction is active~~
~~ICnt: the current instruction count, distance from REP instruction.~~
~~Limit: a 32-bit amount to compare the loop counter against.~~
~~Ins: bits 9 to 15 of the REP instruction which contains the instruction count of instruction included in the repeat and condition under which the repeat occurs.~~

[U/S/H/M]_SCRATCH – CSR 0x?041

This is a scratchpad register. Useful when processing exceptions. There is a separate scratch register for each operating mode.

~~S_PTBR (CSR 0x1003)~~

This register is now located in the page table walker device.

S_ASID (CSR 0x101F)

This register contains the address space identifier (ASID) or memory map index (MMI). The ASID is used in this design to select (index into) a memory map in the paging tables. Only the low order sixteen bits of the register are implemented.

S_KEYS (CSR 0x1020 to 0x1027)

These eight registers contain the collection of keys associated with the process for the memory lot system. Each key is twenty-four bits in size. All eight registers are searched in parallel for keys matching the one associated with the memory page. Keyed memory enhances the security and reliability of the system.

			23 0
1020			key0
1021			key1
…			…
1027			key7

M_CORENO (CSR 0x3001)

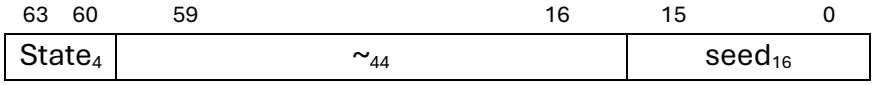
This register contains a number that is externally supplied on the coreno_i input bus to represent the hardware thread id or the core number. It should be non-zero.

M_TICK (CSR 0x3002)

This register contains a tick count of the number of clock cycles that have passed since the last reset. Note that this register should not be used for precise timing as the processor's clock frequency may vary for performance and power reasons. The TIME CSR may be used for wall-clock timing as it has its own timing source.

M_SEED (CSR 0x3003)

This register contains a random seed value based on an external entropy collector. The most significant bit of the state is a busy bit.



State$_4$ Bit	
0	dead
1	test
2	valid, the seed value is valid
3	Busy, the collector is busy collecting a new seed value

M_BADADDR (CSR 0x3007)

This register contains the address for a load / store operation that caused a memory management exception or a bus error. Note that the address of the instruction causing the exception is available in the EIP register.

M_BAD_INSTR (CSR 0x300B)

This register contains a copy of the exceptioned instruction.

M_SEMA (CSR 0x300C)

This register contains semaphores. The semaphores are shared between all cores in the MPU.

M_TVEC – CSR 0x3030 to 0x3034

These registers contain the address of the exception handler table for a given operating mode. TVEC[0] to TVEC[2] are used by the REX instruction.

A sync instruction should be used after modifying one of these registers to ensure the update is valid before continuing program execution.

Reg #	
0x3030	TVEC[0] – user mode
0x3031	TVEC[1] - supervisor mode
0x3032	TVEC[2] – hypervisor mode
0x3033	TVEC[3] – machine mode
0x3034	TVEC[4] - debug

M_SR_STACK (CSR 0x3080 to CSR 0x3087)

This set of registers contains a stack of the status register which is pushed during exception processing and popped on return from interrupt. There are only eight slots as that is the maximum nesting depth for interrupts.

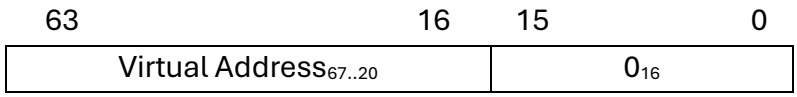
M_MC_STACK (CSR 0x3090 to CSR 0x3097)

This set of registers is a stack for the micro-code instruction register (MCIR) and the micro-code instruction pointer (MCIP). MCIR and MCIP need to be retained through exception processing.

Bits 52 to 63 of the register contain the MCIP. Bits 0 to 51 contain the MCIR.

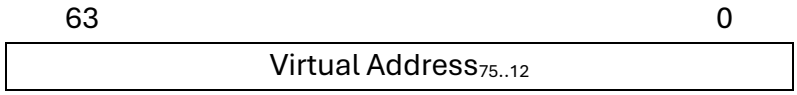
M_IOS – IO Select Register (CSR 0x3100)

The location of IO is determined by the contents of the IOS control register. The select is for a 1MB region. This address is a virtual address. The low order 16 bits of this register should be zero and are ignored.



M_CFGS – Configuration Space Register (CSR 0x3101)

The location of configuration space is determined by the contents of the CFGS control register. The select is for a 256MB region. This address is a virtual address. The low order 12 bits of this address are assumed to be zero. The default value of this registers is $FF…FD000



M_EIP (CSR 0x3108 to 0x310F)

This set of registers contains the address stack for the program counter used in exception handling.

Reg #	Name
0x3108	EIP0
…	
0x310F	EIP7

Operating Modes

The core operates in one of four basic modes: application/user mode, supervisor mode, hypervisor mode or machine mode. Each core may operate in only a single mode.

Most modern OSs require at least two modes of operation, a user mode, and a more secure system mode. It can be advantageous to have more operating modes as it eases the software implementation when dealing with multiple operating systems running on the same machine at the same time.

A subset of instructions is limited to machine mode.

Mode Bits	Mode
0	User / App
1	Supervisor
2	Hypervisor
3	Machine

Each operating mode has its own vector table. Different sets of CSR registers are visible to each operating mode.

	0	1	2	3	4	5	6	7
0								
1								
2	Load	Store						
3	Fp20	Fp40	Fp80					

Exceptions

External Interrupts

There is little difference between an externally generated exception and an internally generated one. An externally caused exception will set the exception cause code for the currently fetched instruction. A hardware interrupt displaces the instruction at the point the interrupt occurred with a TRAP.

There are eight priority interrupt levels for external interrupts. When an external interrupt occurs the mask level is set to the level of the current interrupt. A subsequent interrupt must exceed the mask level to be recognized.

Effect on Machine Status

The operating mode is always switched to machine mode on exception. It is up to the machine mode code to redirect the exception to a lower operating mode when desired. Further exceptions at the same or lower interrupt level are disabled automatically. Machine mode code must enable interrupts at some point.

Exception Stack

The status register and instruction pointer are pushed onto an internal stack when an exception occurs. This stack is at least 8 entries deep to allow for nested interrupts and multiply nested traps and exceptions. The stack pointer is also switched to one corresponding to the machine's operating mode.

Vector Table

The machine mode kernel vector is always used to locate the exception routine. The exception routine may then redirect the exception to a lower operating mode using the REX instruction. When an exception occurs the CPU just jumps to the entry in the vector table. The entry should contain a branch instruction to the exception handler.

Vector	Usage
0	Debug Breakpoint (BRK)
1	Debug breakpoint – single step
2	Bus Error
3	Address Error
4	Unimplemented Instruction
5	Privilege Violation

6	Page fault
7	Instruction trace
8	Stack Canary
9	Abort
10	Interrupt
11	Non-maskable interrupt
12	Reset
13	Alternate Cause
14, 15	Reserved

32

33	
34	Instruction Address
33 to 63	Trap #1 to 31
	Applications Usage
64	Divide by zero
65	Overflow
66	Table Limit
67 to 251	Unassigned usage
252	Reset value of stack pointer
253	Reset value of instruction pointer
254, 255	Reserved

Breakpoint Fault (0)

The breakpoint instruction, 0, was encountered.

Bus Error Fault (2)

The bus error fault is performed if the bus error signal was active during the bus transaction. This could be due to a bad or missing device.

Unimplemented Instruction Fault (4)

An unimplemented instruction causes this fault.

Page Fault (6)

The page table walker was unable to find a valid translation for the virtual address.

Stack Canary Fault (8)

This fault is caused if the stack canary was overwritten. A load instruction using the canary register did not match the value in the canary register.

Abort (9)

The external abort input signal was asserted.

Interrupt (10)

The external interrupt signal was asserted, and the interrupt level was greater than the current mask level.

Reset Vector (12)

This vector is the address that the processor begins running at.

Alternate Cause (13)

The alternate cause vector is jumped to if the cause code is greater than 15.

Reset

Reset is treated as an exception. The reset routine should exit using an RTE instruction. The status register should be setup appropriately for the return.

The core begins executing instructions at the address defined by the reset vector in the exception table. At reset the exception table is set to the last 512 bytes of memory $FF...FFC00. All registers are in an undefined state.

Precision

Exceptions in Qupls are precise. They are processed according to program order of the instructions. If an exception occurs during the execution of an instruction, then an exception field is set in the pipeline buffer. The exception is processed when the instruction commits which happens in program order. If the instruction was executed in a speculative fashion, then no exception processing will be invoked unless the instruction makes it to the commit stage.

Hardware Description

Caches

Overview

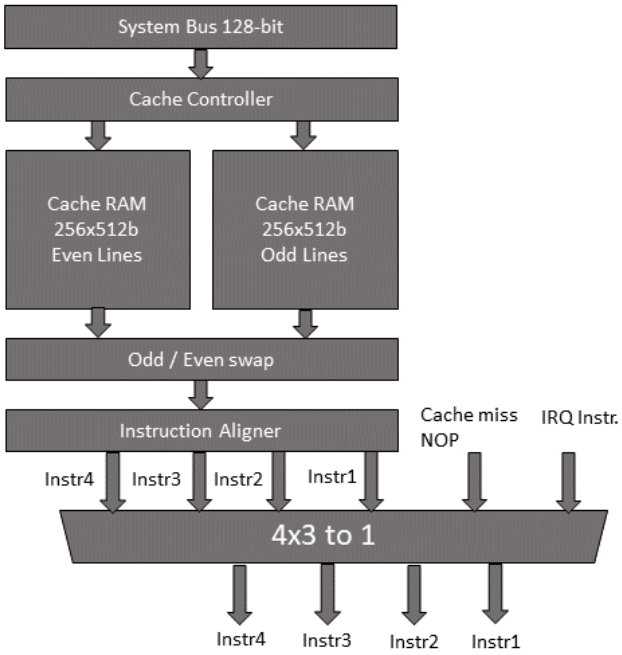
The core has both instruction and data caches to improve performance. Both caches are single level. The cache is four-way associative. The cache sizes of the instruction and data cache are available for reference from one of the info lines return by the CPUID instruction.

Instructions

Since the instruction format affects the cache design it is mentioned here. For this design instructions are of a variable length being 24, 48, 72 or 96 bits in size. Specific formats are listed under the instruction set description section of this book.

L1 Instruction Cache

L1 is 32kB in size and made from block RAM with a single cycle of latency. L1 is organized as an odd, even pair of 256 lines of 64 bytes. The following illustration shows the L1 cache organization for Qupls.



The cache is organized into odd and even lines to allow instructions to span a cache line. Two cache lines are fetched for every access; the one the instruction is located on, and the next one in case the instruction spans a line.

A 256-line cache was chosen as that matches the inherent size of block RAM component in the FPGA. It is the author's opinion that it would be better if the L1 cache were larger because it often misses due to its small size. In short the current design is an attempt to make it easy for the tools to create a fast implementation.

Note that supporting interrupts and cache misses, a requirement for a realistic processor design, adds complexity to the instruction stream. Reading the cache ram, selecting the correct instruction word and accounting for interrupts and cache misses must all be done in a single clock cycle.

While the L1 cache has single cycle reads it requires two clock cycles to update (write) the cache. The cache line to update needs to be provided by the tag memory which is unknown until after the tag updates.

Data Cache

The data cache organization is somewhat simpler than that of the instruction cache. Data is cached with a single level cache because it's not critical that the data be available within a single clock cycle at least not for the hobby design. Some of the latency of the data cache can be hidden by the presence of non-memory operating instructions in the instruction queue.

The data cache is organized as 512 lines of 64 bytes (32kB) and implemented with block ram. Access to the data cache is multicycle. The data cache may be replicated to allow more memory instructions to be processed at the same time; however, just a single cache is in use for the demo system. The policy for stores is write-through. Stores always write through to memory. Since stores follow a write-through policy the latency of the store operation depends on the external memory system. It isn't critical that the cache be able to update in single cycle as external memory access is bound to take many more cycles than a cache update. There is only a single write port on the data cache.

Capabilities Tag Cache

The capabilities tag cache supports the capability system. Every eight bytes of memory has a capabilities tag bit associated with it. If there is a valid capability stored at the address the tag bit will be set, otherwise it will be clear. The tag cache

is 512 lines of 16 bytes of tag bits for a capacity of 64k tags. It is a direct mapped cache.

Cache Enables

The instruction cache is always enabled to keep hardware simpler and faster. Otherwise, an additional multiplexor and control logic would be required in the instruction stream to read from external memory.

For some operations, it may desirable to disable the data cache so there is a data cache enable bit in control register #0. This bit may be set or cleared with one of the CSR instructions.

Cache Validation

A cache line is automatically marked as valid when loaded. The entire cache may be invalidated using the CACHE instruction. Invalidating a single line of the cache is not currently supported, but it is supported by the ISA. The cache may also be invalidated due to a write by another core via a snoop bus.

Un-cached Data Area

The address range $F...FDxxxxx is an un-cached 1MB data area. This area is reserved for I/O devices. The data cache may also be disabled in control register zero. There is also field in the load instructions that allows bypassing the data cache.

Fetch Rate

The fetch rate is four instructions per clock cycle.

Return Address Stack Predictor (RSB)

There is an address predictor for return addresses which can in some cases can eliminate the flushing of the instruction queue when a return instruction is executed. The RETD instruction is detected in the fetch stage of the core and a predicted return address used to fetch instructions following the return. The return address stack predictor has a stack depth of 64 entries. On stack overflow or underflow, the prediction will be wrong, however performance will be no worse than not having a predictor. The return address stack predictor checks the address of the instruction queued following the RET against the address fetched for the RET instruction to make sure that the address corresponds.

Branch Predictor

The branch predictor is a (2, 2) correlating predictor. The branch history is maintained in a 512- entry history table. It has four read ports for predicting branch outcomes, one port for each instruction fetched. The branch predictor may be disabled by a bit in control register zero. When disabled all branches are predicted as not taken, unless specified otherwise in the branch instruction.

To conserve hardware the branch predictor uses a fifo that can queue up to four branch outcomes at the same time. Outcomes are removed from the fifo one at a time and used to update the branch history table which has only a single write port. In an earlier implementation of the branch predictor, two write ports were provided on the history table. This turned out to be relatively large compared to its usefulness.

Correctly predicting a branch turns the branch into a single cycle operation. During execution of the branch instruction the address of the following instruction queued is checked against the address depending on the branch outcome. If the address does not match what is expected, then the queue will be flushed, and new instructions loaded from the correct program path.

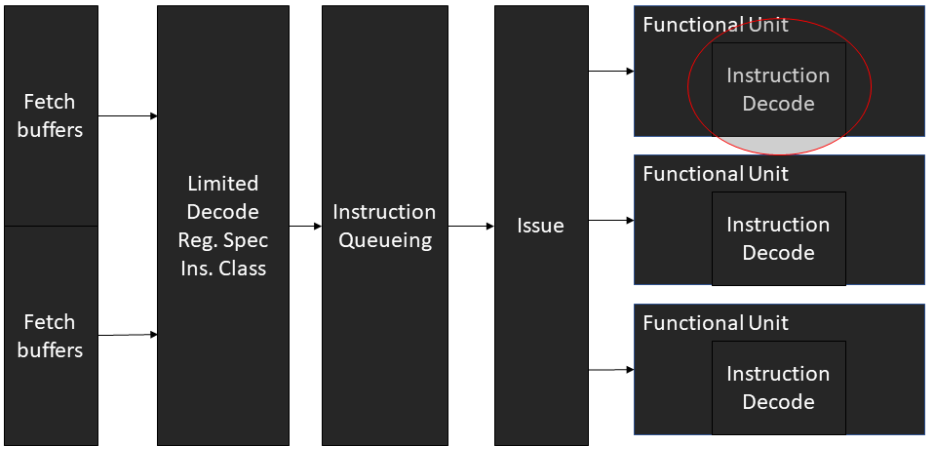
Branch Target Buffer (BTB)

The core has a 1k entry branch target buffer for predicting the target address of flow control instructions where the address is calculated and potentially unknown at time of fetch. Instructions covered by the BTB include jump-and-link, interrupt return and breakpoint instructions and branches to targets contained in a register.

Decode Logic

Instruction decode is distributed about the core. Although a number of decodes take place between fetch and instruction queue. Broad classes of instructions are decoded for the benefit of issue logic along with register specifications prior to instruction enqueue. Most of the decodes are done with modules under the decoder folder. Decoding typically involves reducing a wide input into a smaller number of output signals. Other decodes are done at instruction execution time with case statements.

Placement of Instruction Decode

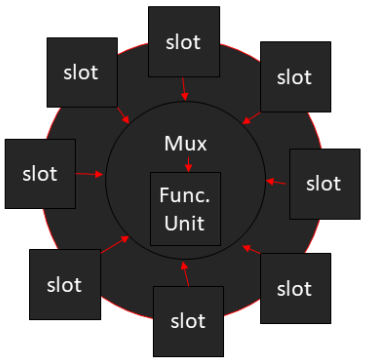


Limited decode takes place between fetch and queue. Between fetch and queue register specifications are decoded along with general instruction classes for the benefit of issue. A handful of additional signals (like sync) that control the overall operation of the core are also decoded. Much of the instruction decode is actually done in the functional unit. The instruction register is passed right through to the functional units in the core.

Instruction Queue (ROB)

The instruction queue is a 32-entry re-ordering buffer (ROB). The instruction queue tracks an instructions progress. Each instruction in queue may be in one of several different states. The instruction queue is a circular buffer with head and tail pointers. Instructions are queued onto the tail and committed to the machine state at the head. Queue and commit takes place in groups of up to four instructions.

Instruction Queue – Re-order Buffer



The instruction queue is circular with eight slots. Each slot feeds a multiplexor which in turn feeds a functional unit. Providing arguments to the functional unit is done under the vise of issue logic. Output from the functional unit is fed back to the same queue slot that issued to the functional unit.
The queue slots are fed from the fetch buffers.

Queue Rate

Up to four instructions may queue during the same clock cycle depending on the availability of queue slots.

Sequence Numbers

The queue maintains a 7-bit instruction sequence number which gives other operations in the core a clue as to the order of instructions. The sequence number is assigned when an instruction queues. Branch instructions need to know when the next instruction has queued to detect branch misses. The program counter cannot be used to determine the instruction sequence because there may be a software loop at work which causes the program counter to cycle backwards even though it's really the next instruction executing.

Input / Output Management

Before getting into memory management a word or two about I/O management is in order. Memory management depends on several I/O devices. I/O in Qupls is memory mapped or MMIO. Ordinary load and store instructions are used to access I/O registers. I/O is mapped as a non-cacheable memory area.

Device Configuration Blocks

I/O devices have a configuration block associated with them that allows the device to be discovered by the OS during bootup. All the device configuration blocks are located in the same 256MB region of memory in the address range $FF...D0000000 to $FF...DFFFFFFF. Each device configuration block is aligned on a 4kB boundary. There is thus a maximum of 64k device configuration blocks.

Reset

At reset the device configuration blocks are not accessible. They must be mapped into memory for access. However, the devices have default addresses assigned to them, so it may not be necessary to map the device control block into memory before accessing the device. The device itself also needs to be mapped into the memory space for access though.

Devices Built into the CPU / MPU

Devices present in the CPU itself include:

Device	Bus	Device	Func	IRQ	Config Block Address	Default Address
Interrupt Controller	0	6	0	~	$FF..FD0030000	$FF..FEE2xxxx
Interval Timers	0	4	0	29	$FF..FD0020000	$FF..FEE4xxxx
Memory Region Table	0	12	0	~	$FF..FD0060000	$FF..FEEFxxxx
Page Table Walker	0	14	0	~	$FF..FD0070000	$FF..FFF4xxxx
Hardware Card Table	0		0	~		

System Devices

Device	Bus	Device	Func	IRQ	Config Block Address	Default Address
Interrupt Reflector	0		0	~	TBD	TBD
Interrupt Logger	0		0	~	TBD	TBD

Function is mapped to address bits 12 to 14

Device is mapped to address bits 15 to 19

Bus is mapped to address bits 20 to 27

External Interrupts

Overview

External interrupts are interrupts external to the CPU and are usually generated by peripheral devices. External interrupts are usually events occurring asynchronously with respect to software running on a CPU. Q+ external interrupts make use of message signaling. Qupls does not follow the MSI / MSI-X standard exactly, although it is similar. The goal of Q+MSI is to be frugal with logic resources. Q+MSI Interrupts are signaled by peripheral devices placing an interrupt message on the peripheral slave response bus. This reuses the response bus pathway to the processing core. Slave peripherals do not need to include bus mastering logic that is normally present with MSI-X.

Interrupt Messages

Interrupt messages are placed on the response bus with an error status indicating an IRQ occurred. The interrupt message identifies the vector number, servicing operating mode, and servicing interrupt controller. This information is stored in a register in the peripheral. An additional 32-bit data word is present in the device to hold extended message information. Q+MSI differs from MSI-X in the storage location of the extended interrupt message information. MSI-X stores this information in the interrupt table whereas Q+ stores it in the device. MSI-X requires the device to perform a write operation to the interrupt table, whereas Q+MSI does not. MSI-X interrupts normally specify an I/O address to post to and a 32-bit data word. Unfortunately, in the Q+ system there are not enough bits in a 32-bit response bus to mimic MSI-X. The vector number combined with the interrupt controller number take the place of the I/O address. Additional information passed by the interrupt message (in the response address field) identifies the source of the interrupt, the desired priority level, and the software stack required for processing.

Interrupt Controller

The Q+ interrupt controller (QIC) is a slave peripheral device that detects interrupt messages occurring on the CPU response bus. It stores the interrupt message in a priority queue. The interrupt vector for the highest priority interrupt is looked up from an internal vector table. Information in the vector determines a list of possible target CPU cores and the software stack that must be available. Either the address of the interrupt subroutine (ISR) or, an instruction for the CPU to execute is provided. There may be multiple interrupt controllers in the system. Currently a six-bit controller

number is present in the interrupt message limiting the number of controllers to 63. With 63 interrupt controllers and each one servicing 63 CPU cores, a maximum of approximately 3900 CPU cores may be connected to interrupts.

The interrupt controller has some capacity to detect interrupt overruns. There is a “stuck interrupt” detector which flags an interrupt signal as being stuck if the same interrupt message is posted in a short time-frame. The queue full status flag is also available in the controller allowing software to detect if a queue is full. A full queue may also indicate a stuck interrupt.

There is more detail pertaining to QIC in the QIC device description later in this document.

Interrupt Vector Table

The interrupt vector table is internal to the interrupt controller. The table is laid out in four sections, one for each available operating mode. There are 2048 (512 in the demo system) vectors available for each operating mode. Note there may be multiple interrupt controllers in the system, and hence multiple vector tables. Which vector table to use is identified in a device control register in the form of specifying an interrupt controller number.

Interrupt Group Filter

There may be more than one CPU core connected to a QIC; up to 63 CPU cores may be connected to a QIC. Note that groups of CPU cores may be specified to handle an interrupt. There is a filter in the MPU that detects the lowest priority CPU core that is ready to handle an interrupt. The information from the QIC about the interrupt is passed to connected CPU cores.

To be ready to handle an interrupt, the current interrupt level of the CPU core must be less than that of the interrupting device, and the CPU core must be operating using the software stack appropriate for the interrupt.

Interrupt Reflector

The interrupt reflector is a peripheral device that allows a bus master to trigger an interrupt. Because interrupts are posted on the response bus for Q+ a bus master would not be able to trigger an interrupt directly. The reflector moves a request from the bus master request bus over the response bus. It can then be detected by the

interrupt controller. This allows IPI (inter-processor interrupts) generated by software to be used.

Interrupt Logger

Logging of interrupts can be useful for the system. It is handy for debugging. The interrupt logger is a peripheral device that monitors the CPU response bus for interrupts (like the QIC) and logs all interrupts to a file in memory. The file can be subsequently processed for system management purposes.

Memory Management

This section is somewhat pedantic and reviews technical approaches before getting into Qupls details.

Bank Swapping

About the simplest form of memory management is a single bank register that selects the active memory bank. This is the mechanism used on many early microcomputers. The bank register may be an eight bit I/O port supplying control over some number of upper address bits used to access memory.

The Page Map

The next simplest form of memory management is a single table map of virtual to physical addresses. The page map is often located in a high-speed dedicated memory. An example of a mapping table is the 74LS612 chip. It may map four address bits on the input side to twelve address bits on the output side. This allows a physical address range eight bits greater than the virtual address range. A more complicated page map is something like the MC6829 MMU. It may map 2kB pages in a 2MB physical address space for up to four different tasks.

Regions

In any processing system there are typically several different types of storage assigned to different physical address ranges. These include memory mapped I/O, MMIO, DRAM, ROM, configuration space, and possibly others. Qupls has a region table that supports up to eight separate regions.

The region table is a list of region entries. Each entry has a start address, an end address, an access type field, and a pointer to the PMT, page management table. To determine legal access types, the physical address is searched for in the region table, and the corresponding access type returned. The search takes place in parallel for all eight regions.

Once the region is identified the access rights for a particular page within the region can be found from the PMT corresponding to the region. Global access rights for the entire region are also specified in the region table. These rights are gated with value from the PMT and TLB to determine the final access rights.

Region Table Location

The region table in Q+ is a memory mapped I/O device and has a device configuration block associated with it. The default address of the device is $FF...FEEF0000.

Region Table Description

Reg	Bits	Field	Description
0000	128	Pmt	associated PMT address
0010	128	cta	Card table address
0020	128	at	Four groups of 32-bit memory attributes, 1 group for each of user, supervisor, hypervisor and machine.
0030	128	...	Not used
0040 to 01F0		...	7 more register sets

PMT Address

The PMT address specifies the location of the associated PMT.

CTA – Card Table Address

The card table address is used during the execution of the store pointer, STPTR instruction to locate the card table.

Attributes

Bitno		
0	X	may contain executable code
1	W	may be written to
2	R	may be read
3	~	reserved
4-7	C	Cache-ability bits
8-10	G	granularity G: 0 byte accessible; 1 wyde accessible; 2 tetra accessible; 3 octa accessible; 4 hexi accessible; 5 to 7 reserved
11	~	reserved
12-14	S	number of times to shift address to right and store for telescopic STPTR stores.
16-23	T	device type (rom, dram, eeprom, I/O, etc)
24-31	~	reserved

PMA - Physical Memory Attributes Checker

Overview

The physical memory attributes checker is a hardware module that ensures that memory is being accessed correctly according to its physical attributes.

Physical memory attributes are stored in an eight-entry region table. Three bits in the PTE select an entry from this table. The operating mode of the CPU also determines which 32-bit set of attributes to apply for the memory region.

Most of the entries in the table are hard-coded and configured when the system is built. However, they may be modified.

Physical memory attributes checking is applied in all operating modes.

The region table is accessible as a memory mapped IO, MMIO, device.

Page Management Table - PMT

Overview

For the first translation of a virtual to physical address, after the physical page number is retrieved from the TLB, the region is determined, and the page management table is referenced to obtain the access rights to the page. PMT information is loaded into the TLB entry for the page translation. The PMT contains an assortment of information most of which is managed by software. Pieces of information include the key needed to access the page, the privilege level, and read-write-execute permissions for the page. The table is organized as rows of access rights table entries (PMTEs). There are as many PMTEs as there are pages of memory in the region.

For subsequent virtual to physical address translations PMT information is retrieved from the TLB.

As the page is accessed in the TLB, the TLB may update the PMT.

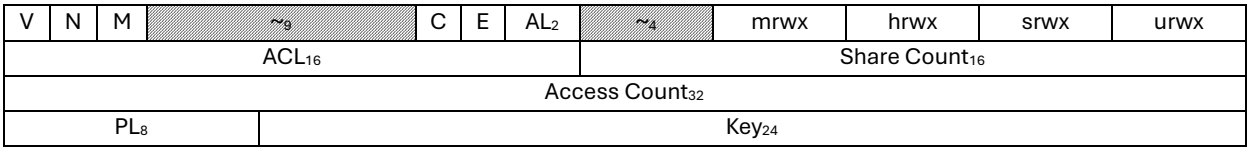
Location

The page management table is in main memory and may be accessed with ordinary load and store instructions. The PMT address is specified by the region table.

PMTE Description

There is a wide assortment of information that goes in the page management table. To accommodate all the information an entry size of 128-bits was chosen.

Page Management Table Entry



Access Control List

The ACL field is a reference to an associated access control list.

Share Count

The share count is the number of times the page has been shared to processes. A share count of zero means the page is free.

Access Count

This part uses the term ‘access count’ to refer to the number of times a page is accessed. This is usually called the reference count, but that phrase is confusing because reference counting may also refer to share counts. So, the phrase ‘reference count’ is avoided. Some texts use the term reference count to refer to the share count. Reference counting is used in many places in software and refers to the number of times something is referenced.

Every time the page of memory is accessed, the access count of the page is incremented. Periodically the access count is aged by shifting it to the right one bit.

The access count may be used by software to help manage the presence of pages of memory.

Key

The access key is a 24-bit value associated with the page and present in the key ring of processes. The keyset is maintained in the keys CSRs. The key size of 20 bits is a minimum size recommended for security purposes. To obtain access to the page it is necessary for the process to have a matching key OR if the key to match is set to zero in the PMTE then a key is not needed to access the page.

Privilege Level

The current privilege level is compared with the privilege level of the page, and if access is not appropriate then a privilege violation occurs. For data access, the current privilege level must be at least equal to the privilege level of the page. If the page privilege level is zero anybody can access the page.

N

indicates a conforming page of executable code. Conforming pages may execute at the current privilege level. In which case the PL field is ignored.

M

indicates if the page was modified, written to, since the last time the M bit was cleared. Hardware sets this bit during a write cycle.

E

indicates if the page is encrypted.

AL

indicates the compression algorithm used.

C

The C indicator bit indicates if the page is compressed.

urwx, srwx, hrwx, mrwx

These are read-write-execute flags for the page.

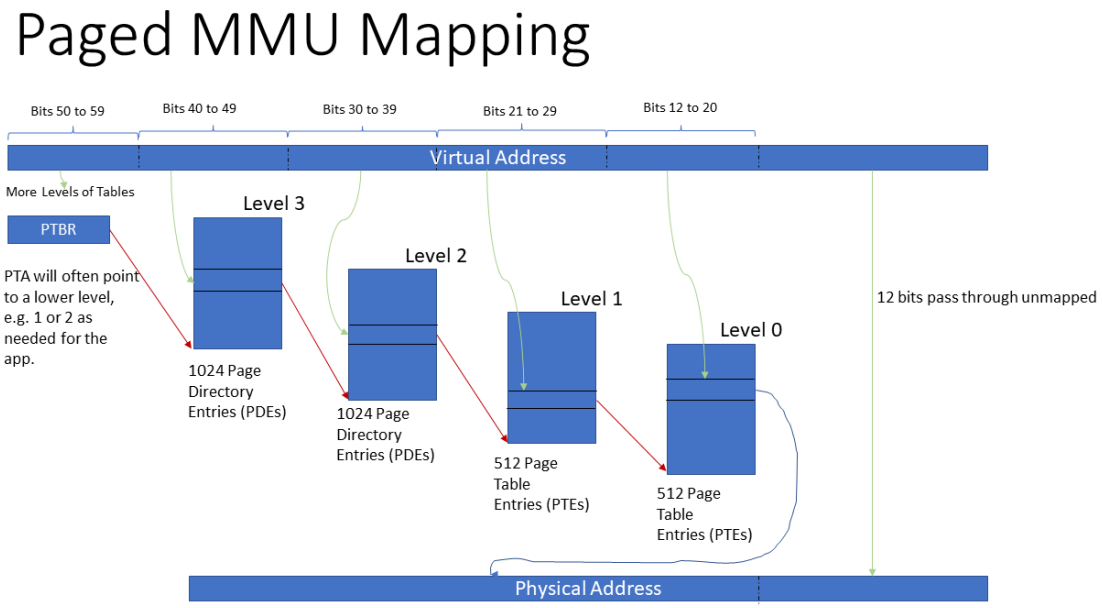
Page Tables

Intro

Page tables are part of the memory management system used map virtual addresses to real physical addresses. There are several types of page tables. Hierarchical page tables are probably the most common. Almost all page tables map only the upper bits of a virtual address, called a page. The lower bits of the virtual address are passed through without being altered. The page size often 4kB which means the low order 12-bits of a virtual address will be mapped to the same 12-bits for the physical address.

Hierarchical Page Tables

Hierarchical page tables organize page tables in a multi-level hierarchy. They can map the entire virtual address range but often only a subrange of the full virtual address space is mapped. This can be determined on an application basis. At the topmost level a register points to a page directory, that page directory points to a page directory at a lower level until finally a page directory points to a page containing page table entries. To map an entire 64-bit virtual address range approximately five levels of tables are required.



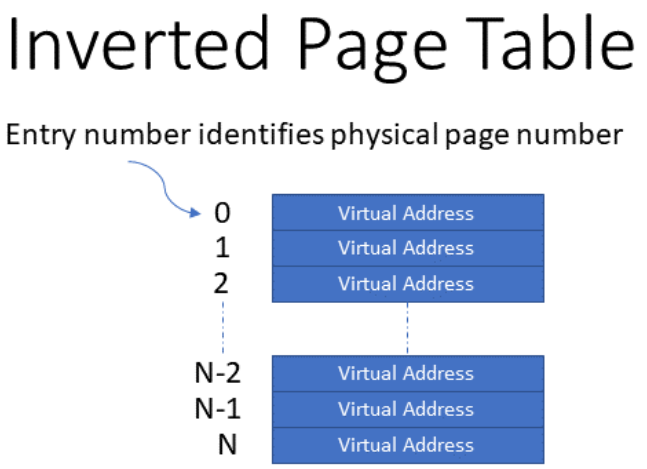
Inverted Page Tables

An inverted page table is a table used to store address translations for memory management. The idea behind an inverted page table is that there are a fixed

number of pages of memory no matter how it is mapped. It should not be necessary to provide for a map of every possible address, which is what the hierarchical table does, only addresses that correspond to real pages of memory need be mapped. Each page of memory can be allocated only once. It is either allocated or it is not. Compared to a non-inverted paged memory management system where tables are used to map potentially the entire address space an inverted page table uses less memory. There is typically only a single inverted page table supporting all applications in the system. This is a different approach than a non-inverted page table which may provide separate page tables for each process.

The Simple Inverted Page Table

The simplest inverted page table contains only a record of the virtual address mapped to the page, and the index into the table is used as the physical page number. There are only as many entries in the inverted page table as there are physical pages of memory. A translation can be made by scanning the table for a matching virtual address, then reading off the value of the table index. The attraction of an inverted page table is its small size compared to the typical hierarchical page table. Unfortunately, the simplest inverted page table is not practical when there are thousands or millions of pages of memory. It simply takes too long to scan the table. The alternative solution to scanning the table is to hash the virtual address to get a table index directly.



Hashed Page Tables

Hashed Table Access

Hashes are great for providing an index value immediately. The issue with hash functions is that they are just a hash. It is possible that two different virtual address

will hash to the same value. What is then needed is a way to deal with these hash collisions. There are a couple of different methods of dealing with collisions. One is to use a chain of links. The chain has each link in the chain pointing the to next page table entry to use in the event of a collision. The hash page table is slightly more complicated then as it needs to store links for hash chains. The second method is to use open addressing. Open addressing calculates the next page table entry to use in the event of a collision. The calculation may be linear, quadratic or some other function dreamed up. A linear probe simply chooses the next page table entry in succession from the previous one if no match occurred. Quadratic probing calculates the next page table entry to use based on squaring the count of misses.

Clustered Hash Tables

A clustered hash table works in the same manner as a hashed page table except that the hash is used to access a cluster of entries rather than a single entry. Hashed values may map to the same cluster which can store multiple translations. Once the cluster is identified, all the entries are searched in parallel for the correct one. A clustered hash table may be faster than a simple hash table as it makes use of parallel searches. Often accessing memory returns a cache line regardless of whether a single byte or the whole cached line is referenced. By using a cache line to store a cluster of entries it can turn what might be multiple memory accesses into a single access. For example, an ordinary hash table with open addressing may take up to 10 memory accesses to find the correct translation. With a clustered table that turns into 1.25 memory accesses on average.

Shared Memory

Another memory management issue to deal with is shared memory. Sometimes applications share memory with other apps for communication purposes, and to conserve memory space where there are common elements. The same shared library may be used by many apps running in the system. With a hierarchical paged memory management system, it is easy to share memory, just modify the page table entry to point to the same physical memory as is used by another process. With an inverted page table having only a single entry for each physical page is not sufficient to support shared memory. There needs to be multiple page table entries available for some physical pages but not others because multiple virtual addresses might map to the same physical address. One solution would be to have multiple buckets to store virtual addresses in for each physical address. However, this would waste a lot of memory because much of the time only a single mapped address is needed. There must be a better solution. Rather than reading off the table index as the

physical page number, the association of the virtual and physical address can be stored. Since we now need to record the physical address multiple times the simple mechanism of using the table index as the physical page number cannot be used. Instead, the physical page number needs to be stored in the table in addition to the virtual page number.

That means a table larger than the minimum is required. A minimally sized table would contain only one entry for each physical page of memory. So, to allow for shared memory the size of the table is doubled. This smells like a system configuration parameter.

Specifics: Qupls Page Tables

Qupls Hash Page Table Setup

Hash Page Table Entries - HPTE

We have determined that a page table entry needs to store both the physical page number and the virtual page number for the translations. To keep things simple, the page table stores only the information needed to perform an address translation. Other bits of information are stored in a secondary table called the page management table, PMT. The author did a significant amount of juggling around the sizes of various fields, mainly the size of the physical and virtual page numbers. Finally, the author decided on a 192-bit HPTE format.

V	LVL/BC$_5$	RGN$_3$	M	A	T	S	G	SW$_2$	CACHE$_4$	MRWX$_3$	HRWX$_3$	SRWX$_3$	URWX$_3$
PPN$_{31..0}$													
PPN$_{63..32}$													
VPN$_{37..6}$													
VPN$_{69..38}$													
~$_4$		ASID$_{11..0}$						~$_2$		VPN$_{83..70}$			

Fields Description

V	1	translation Valid
G	1	global translation
RGN	3	region
PPN	64	Physical page number
VPN	84	Virtual page number
RWX	3	readable, writeable, executable
ASID	12	address space identifier
LVL/BC	5	bounce count

M	1	modified
A	1	accessed
T	1	PTE type (not used)
S	1	Shared page indicator
SW	3	OS usage

The page table does not include everything needed to manage pages of memory. There is additional information such as share counts and privilege levels to take care of, but this information is better managed in a separate table.

Small Hash Page Table Entries - SHPTE

The small HPTE is used for the test system which contains only 512MB of physical RAM to conserve hardware resources. The SHPTE is 72-bits in size. A 32-bit physical address is probably sufficient for this system. So, the physical page number could be 18-bits or less depending on the page size.

V	LVL/BC$_5$	RGN$_3$	M	A	T	S	G	SW	CACHE$_4$	ASID$_{3..0}$	HRWX$_3$	SRWX$_3$	URWX$_3$
VPN$_{15..0}$									PPN$_{15..0}$				
											ASID$_{7..4}$	VPN$_{19..16}$	

Page Table Groups – PTG

We want the search for translations to be fast. That means being able to search in parallel. So, PTEs are stored in groups that are searched in parallel for translations. This is sometimes referred to as a clustered table approach. Access to the group should be as fast as possible. There are also hardware limits to how many entries can be searched at once while retaining a high clock rate. So, the convenient size of 1024 bits was chosen as the amount of memory to fetch.

A page table group then contains five HPTE entries. All entries in the group are searched in parallel for a match. Note that the entries are searched as the PTG is loaded, so that the PTG group load may be aborted early if a matching PTE is found before the load is finished.

191 0

PTE0
PTE1
PTE2
PTE3
PTE4

Small Page Table Group

For the small page table, a fetch size of 576 bits was chosen. This allows eight SHPTEs to fit into one group.

Size of Page Table

There are several conflicting elements to deal with, with regards to the size of the page table. Ideally, the hash page table is small enough to fit into the block RAM resources available in the FPGA. It may be practical to store the hash page table in block RAM as there would be only a single table for all apps in the system. This probably would not be practical for a hierarchical table.

About 1/6 of the block RAMs available are dedicated to MMU use. At the same time a multiple of the number of physical pages of memory should be supported to support page sharing and swapping pages to secondary storage. To support swapping pages, double the number of physical entries were chosen. To support page sharing, double that number again. Therefore, a minimum size of a page table would contain at least four times the number of physical pages for entries. By setting the size of the page table instead of the size of pages, it can be worked backwards how many pages of memory can be supported.

For a system using 256k block RAM to store PTEs. 256k / 8 = 32768 entries. 32,768 / 4 = 8,192 physical pages. Since the RAM size is 512MB, each page would be 512MB/8,192 = 64kB. Since half the pages may be in secondary storage, 1GB of address range is available.

Since there are 32,768 entries in the table and they are grouped into groups of eight, there are 4,096 PTGs. To get to a page table group fast a hash function is needed then that returns a 12-bit number.

Reworking things with a 64kB page size and 32,768 PTEs. The maximum memory size that can be supported is: 2.0 GB. This is only 4x the amount of RAM in the system, but may be okay for demo purposes.

Hash Function

The hash function needs to reduce the size of a virtual address down to a 10-bit number. The asid should be considered part of the virtual address. Including the asid of 10-bits and a 32-bit address is 42 bits. The first thing to do is to throw away the lowest eighteen bits as they pass through the MMU unaltered. We now have 24-

bits to deal with. We can probably throw away some high order bits too, as a process is not likely to use the full 32-bit address range.

The hash function chosen uses the asid combined with virtual address bits 20 to 29. This should space out the PTEs according to the asid. Address bits 18 and 19 select one of four address ranges. the PTG supports seven PTEs. The translations where address bits 18 and 19 are involved are likely consecutive pages that would show up in the same PTG. The hash is the asid exclusively or'd with address bis 20 to 29.

Collision Handling

Quadratic probing of the page table is used when a collision occurs. The next PTG to search is calculated as the hash plus the square of the miss count. On the first miss the PTG at the hash plus one is searched. Next the PTG at the hash plus four is searched. After that the PTG at the hash plus nine is searched, and so on.

Finding a Match

Once the PTG to be searched is located using the hash function, which PTE to use needs to be sorted out. The match operation must include both the virtual address bits and the asid, address space identifier, as part of the test for a match. It is possible that the same virtual address is used by two or more different address spaces, which is why it needs to be in the match.

Locality of Reference

The page table group may be cached in the system read cache for performance. It is likely that the same PTG group will be used multiple times due to the locality of reference exhibited by running software.

Access Rights

To avoid duplication of data the access rights are stored in another table called the PMT for access rights table. The first time a translation is loaded the access rights are looked-up from the PMT. A bit is set in the TLB entry indicating that the access rights are valid. On subsequent translations the access rights are not looked up, but instead they are read from values cached in the TLB.

Qupls2 Hierarchical Page Table Setup

Overview

Qupls2 hierarchical page tables are setup like a tree. Branch pages contain pointers to other pages and leaf pages contain pointers to block of memory that an application has access to. The entries in branch pages are referred to as page table pointers, PTPs, since they point at other page tables. The entries in leaf pages are referred to as page table entries, PTEs. Pages are 8kB in size. There may be from one to seven levels of page tables. A single page table level is sufficient to map 8MB of memory.

Page Table Pointer Format – PTP

The PTP occupies 64-bits. 1024 PTPs will fit into an 8kB page. A physical address range maximum of 2^{70} bytes of memory may be mapped.

LVL_3	S	Rgn_3	$PPN_{56...0}$

Page Table Entry Format – PTE

The PTE format may map up to 2^{58} bytes of contiguous memory. The upper address bits for the translation are supplied by bits 45 to 56 of the PTP. The PTE is eight bytes in size. 1024 PTEs will fit into an 8kB page.

LVL_3	S	Rgn_3	M	A	AVL_3	$CACHE_3$	U_1	RWX_3	$PPN_{44...0}$

Field	Size	Purpose
PPN	45	Physical page number
RWX	3	read-write-execute
U	1	1=User page, 0 = Supervisor
CACHE	3	Cache-location
AVL	3	OS software usage
A	1	1=accessed/used
M	1	1=modified
RGN	3	Memory region
S	1	1=shortcut
LVL	3	010 to 111 = PTP, 001 = PTE, 000 = invalid

Shortcut Translations

Translation mappings may be shortcut for the first three levels of page tables allowing the page table to map 8GB, 8TB, or 8XB of memory using just a single level table.

For a shortcut page, the low order bits of the page number indicate a limit on the size of the memory area mapped. For instance, if LVL=010 is a shortcut, the low order 10 bits of the PPN specify the limit in terms of number of 8kB pages. The upper bits of the PPN represent a map to an 8GB area of memory.

Location of Page Table

The page table walker contains a register specifying the base location of the page table. Please refer to the Qupls2 page table walker for more information.

TLB – Translation Lookaside Buffer

Overview

A simple page map is limited in the translations it can perform because of its size. The solution to allowing more memory to be mapped is to use main memory to store the translations tables.

However, if every memory access required two or three additional accesses to map the address to a final target access, memory access would be quite slow, slowed down by a factor or two or three, possibly more. To improve performance, the memory mapping translations are stored in another unit called the TLB standing for Translation Lookaside Buffer. This is sometimes also called an address translation cache ATC. The TLB offers a means of address virtualization and memory protection. A TLB works by caching address mappings between a real physical address and a virtual address used by software. The TLB deals with memory organized as pages. Typically, software manages a paging table whose entries are loaded into the TLB as translations are required.

The TLB is a cache specialized for address translations. Qupls's TLB is two level. The first level contains eight full associative entries making translations possible within one clock cycle. The second level contains 1024 three-way associative entries. If there is a miss on the first TLB level the second level will be searched for a translation. If available a translation is possible within two clock cycles. On a second level TLB miss the page table is searched for a translation by a hardware-based page table walker and if found the translation is stored in one of the ways of the TLB. The way selected is determined randomly.

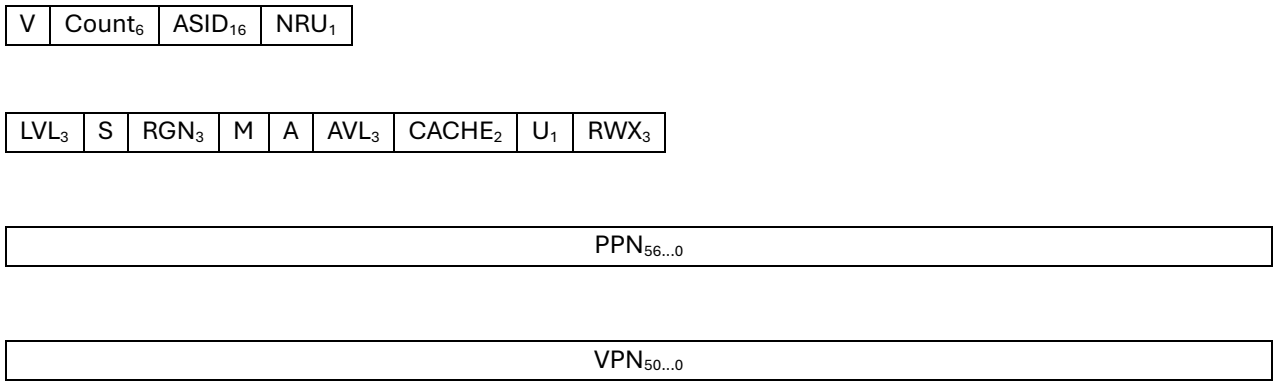
Size / Organization

The first level TLB has 8 fully associative entries.

The second level TLB has 1024 entries per set for 8kB pages and 128 entries per set for 8MB pages.

TLB Entries - TLBE

Closely related to page table entries are translation look-aside buffer, TLB, entries. TLB entries have additional fields to match against the virtual address. The count field is used to invalidate the entire TLB.



What is Translated?

The TLB processes addresses including both instruction and data addresses for all modes of operation. It is known as a *unified* TLB.

Page Size

Because the TLB caches address translations it can get away with a much smaller page size than the page map can for a larger memory system. 4kB is a common size for many systems. There are some indications in contemporary documentation that a larger page size would be better. In this case the TLB uses 8kB. For a 1GB system (the size of the memory in the test system) there are 131072 8kB pages.

Ways

The L1TLB is eight-way associative. The L2TLB is three-way associative.

Management

The TLB unit is updated by a hardware page table walker.

RWX_3

Otherwise RWX attributes are typically set by OS software and determined by the region table.

$CACHE_3$

The $cache_3$ field indicates the location of data in the cache hierarchy.

TLB Entry Replacement Policies

The TLB uses random replacement. Random replacement chooses a way to replace at random.

Flushing the TLB

The TLB maintains the address space (ASID) associated with a virtual address. This allows the TLB translations to be used without having to flush old translations from the TLB during a task switch.

Reset

On a reset the TLB is preloaded with translations that allow access to the system ROM.

PTW - Page Table Walker

The page table walker is a CPU device used to update the TLB. Whenever a TLB miss occurs the page table walker is triggered. The page table walker walks the page tables to find the translation. Once found the TLB is updated with the translation. If a translation cannot be found, then a page fault occurs.

The page table walker managers several variables associated with memory management. These include the page table base register, PT_BASE, page fault address and ASID. These registers are available to software using load and store instructions.

For a page fault the miss address and ASID are stored in a register in the page-table-walker. The PTW also contains the PT_BASE(page table base register) which is used to locate the page table.

The page table walker is a device located in the CPU and has a device configuration block associated with it. The default address of the device is $FF...FF40000.

Register	Name	Description
$FF00	PF_ADDR	Page fault address
$FF10	PF_ASID	Page fault asid
$FF20	PT_BASE	Page table base register
$FF30	PT_ATTR	Page table attributes

Page Table Base Register

The page table base register locates the page table in memory. Address bits 3 to 63 are specified. The page table must be octa-byte aligned. Normally the root page table will occupy 8kB of memory and be 8kB aligned. However, for smaller apps it may be desirable to share the memory page the page table is located in with multiple applications.

63 … 3	2 … 0
Page Table Address$_{63..3}$	0

Default Reset Value = *0xFFFFFFFFFFF80000*

Page Table Attributes Register

The attributes register contains attributes of the page table.

63 … 26	25	24 … 12	11 … 8	7 6	5 … 4	3	2 1	0
~$_{38}$	~	Root Page Table Limit$_{12..0}$	Levels	AL$_2$	~$_2$	S	~	Type

Type: 0 = inverted page table, 1 = hierarchical page table

S: 1=software managed TLB miss, 0 = hardware table walking, 0 is the only currently supported option.

AL_2: TLB entry replacement algorithm, 0=fixed,1=LRU,2=random,3=reserved, 2 is the only currently supported option.

Levels are ignored for the inverted page table. For a normal page table gives the top entry level.

Root Page Table Limit specifies the number of entries in the root page table. A maximum of 8192 entries is supported.

Default Reset Value = 0x1FFF081

63 26	25	24 12	11 8	7 6	5 4	3	2 1	0
$\sim_{38}$	~	1FFFh	0	2	$\sim_2$	0	~	1

Card Table

Overview

Also present in the memory system is the Card table. The card table is a telescopic memory which reflects with increasing detail where in the memory system a pointer write has occurred. This is for the benefit of garbage collection systems. Card table is updated using a write barrier when a pointer value is stored to memory, or it may be updated automatically using the STPTR instruction.

Organization

At the lowest level memory is divided into 256-byte card memory pages. Each card has a single byte recording whether a pointer store has taken place in the corresponding memory area.

To cover a 1GB memory system 4MB card memory is required at the outermost layer. A byte is used rather than a bit to allow byte store operations to update the table directly without having to resort to multiple instructions to perform a bit-field update.

To improve the performance of scanning a hardware card table, HCT, is present which divides memory at an upper level into 8192-byte pages. The hardware card table indicates if a pointer store operation has taken place in one of the 8192-byte pages. It is then necessary to scan only cards representing the 8192-byte page rather than having to scan the entire 4MB card table. Note that this memory is organized as 2048 64-bit words. Allowing 64-bits at a time to be tested.

To further improve performance a master card table, MCT, is present which divides memory at the uppermost layer into 16-MB pages.

Layer	Resolving Power	
0	4 MB	256B pages
1	128k bits	8kB pages
2	64 bits	16 MB pages

There is only a single card memory in the system, used by all tasks.

Location

The card memory location is stored in the region table. A card table may be setup for each region of memory.

Operation

As a program progresses it writes pointer values to memory using the write barrier. Storing a pointer triggers an update to all the layers of card memory corresponding to the main memory location written. A bit or byte is set in each layer of the card memory system corresponding to the memory location of the pointer store.

The garbage collection system can very quickly determine where pointer stores have occurred and skip over memory that has not been modified.

Sample Write Barrier

```
; Milli-code routine for garbage collect write barrier.
; This sequence is short enough to be used in-line.
; Three level card memory.
; a2 is a register pointing to the card table.
; STPTR will cause an update of the master card table, and hardware card table.
;
GCWriteBarrier:
    STPTR     a0,[a1]          ; store the pointer value to memory at a1
    LSR       t0,a1,#8         ; compute card address
    STIB      $0,[a2+t0]       ; clear byte in card memory
```

Instruction Set

Overview

Qupls2 is a variable length instruction set with lengths of 24, 48, 72 or 96 bits. There are several different classes of instructions including arithmetic, memory operate, branch, floating-point and others.

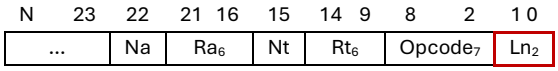
Code Alignment

Program code may be relocated at any byte address. However, within a subroutine code should be contiguous.

Instruction Length

A 24-bit instruction parcel size was chosen to try and capture as many instructions as possible using only 24-bits. This is more compact than a 32-bit parcel but is not capable of encoding as many instructions. Although individual instructions may be denser the length of program code may not be depending on the instruction mix. It may take more 24-bit instructions to encode a program than 32-bit ones.

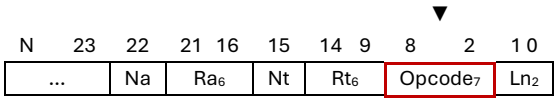
To simplify decoding, the length of an instruction is encoded directly as the two LSBs of every instruction.



Ln_2	Bits
0	24
1	48
2	72
3	96

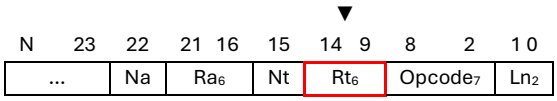
Root Opcode

The root opcode determines the class of instructions executed. Some commonly executed instructions are also encoded at the root level to make more bits available for the instruction. The root opcode is always present in all instructions as bits two to eight of the instruction.



Target Register Spec

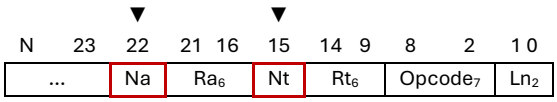
Most instructions have a target register. The register spec for the target register is always in the same position, bits 9 to 14 of an instruction. For some instructions, such as stores, the target register field acts as a source register.



Sign Control

Many instructions feature sign control for each register of the instruction. Loads / stores and branches do not have sign control.

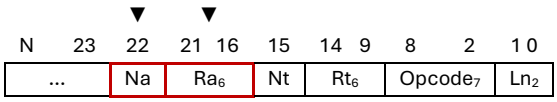
Each register may have its sign negated or complemented during the operation. Arithmetic operations will negate, bitwise operations will complement.



Nt/Na/Nb/Nc	
0	No effect
1	Negate (arithmetics) or Complement (logical)

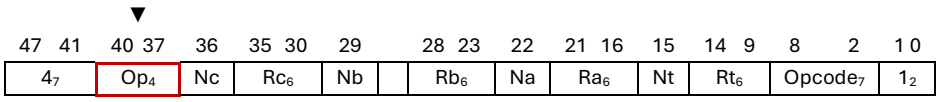
Source Register Spec

Most instructions have at least one source register. There may be as many a three source register specs. Please refer to individual instruction descriptions for the location of the source register specification fields.



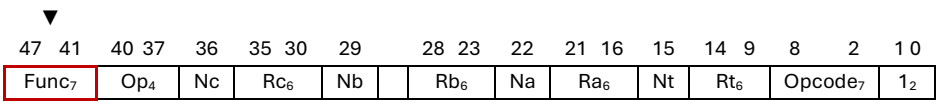
Secondary Opcode

Register-register operate instructions often have slightly different forms depending on a secondary opcode. The secondary opcode generally controls the operation between the result from the first two source register and the third source register.



Primary Function Code

For register to register operate instructions the primary function code is in the most significant seven bits of the instruction, bits 41 to 47. This function code typically controls the operation between registers Ra and Rb; sometimes Rc is also included.



Precision

The CPU supports multiple precisions for most operations. The precision selected is often controlled by the opcode. A register may be treated as 1 64-bit values, 2 32-bit values, or 4 16-bit values. The same operation is applied for each value. A pair of registers may be treated as a 128-bit value for some instructions.

$Opcode_7$	Values
2	1 x 128 bit
104	8 x 8-bit
105	4 x 16 bit
106	2 x 32-bit
107	1 x 64-bit

Instruction Descriptions

Instruction Length

Ln_2	Bits
0	24
1	48
2	96
3	

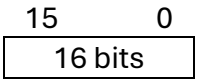
Arithmetic Operations

Representations

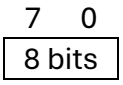
int

short int

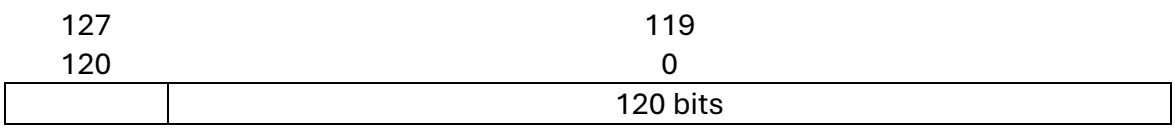
wyde



byte



decimal



Decimal integers use densely packed decimal format which provide 36 digits of precision.

Arithmetic Operations

Arithmetic operations include addition, subtraction, multiplication and division. These are available with the ADD, SUB, CMP, MUL, and DIV instructions. There are several variations of the instructions to deal with signed and unsigned values. Multiply may either multiply two values and add a third returning the low order bits, or return the entire product, referred to as a widening instruction. Divide may return both the quotient and the remainder with one instruction. The format of the typical immediate mode instruction is shown below:

ADD.sz Rt, Ra, Imm_{23}

Instruction Format: RI

47 25	24 23	22	21 16	15	14 9	8 2	1 0
$Immediate_{23}$	Prc_2	Na	Ra_6	Nt	Rt_6	4_7	1_2

Precision

Four different precisions are supported encoded by the Prc_2 field of an instruction. The precision of an operation may be specified with an instruction qualifier following the mnemonic as in ADD.T to add tetras together. The assembler assumes an octa-byte operation if the size is not specified.

Prc_2	Register treated as: Bits x lanes
0	8 x 8
1	16 x 4
2	32 x 2
3	64 x 1

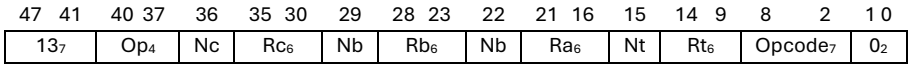
If the precision field is not present in the instruction, then an octa-byte operation is assumed.

ABS – Absolute Value

Description:

This instruction computes the absolute value of the contents of the source operand and places the result in Rt.

Instruction Format: R1



$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

OP_4		Mnemonic
3	Rt = ±ABS((±Ra ± Rb) ± Rc)	ABS_SUM

Operation:

```
If Source < 0
   Rt = -Source
else
   Rt = Source
```

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

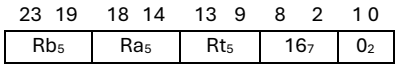
Notes:

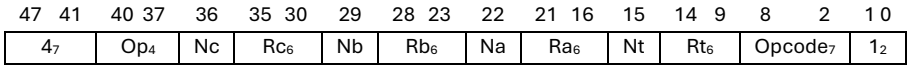
ADD - Register-Register

Description:

Add three registers and place the sum in the target register. All register values are integers. If Rc is not used, it is assumed to be zero.

Instruction Format: R3





Operation:

$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

OP_4		Mnemonic
0	Rt = (Ra ± Rb) & Rc	ADD_AND
1	Rt = (Ra ± Rb) | Rc	ADD_OR
2	Rt = (Ra ± Rb) ^ Rc	ADD_EOR
3	Rt = (Ra ± Rb) ± Rc	ADD_ADD
11	Rt = (Ra ± Rb) ± Rc	ADDGC
4 to 15	Reserved	

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

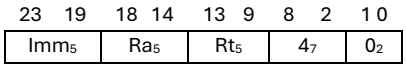
Notes:

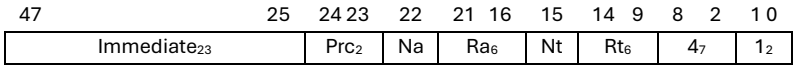
ADDI - Add Immediate

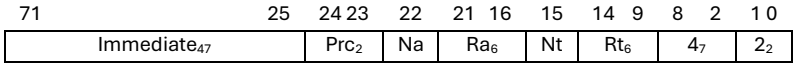
Description:

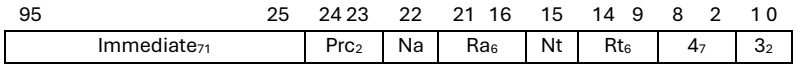
Add a register and immediate value and place the sum in the target register. The immediate is sign extended to the machine width.

Instruction Format: RI









Clock Cycles: 1

Execution Units: All ALUs, all FPUs

Operation:

Rt = Ra + immediate

Exceptions:

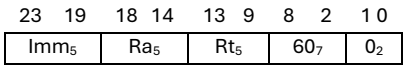
Notes:

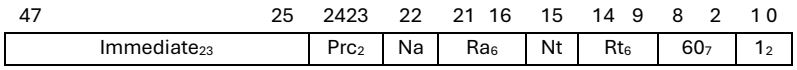
ADD2UI - Add Immediate

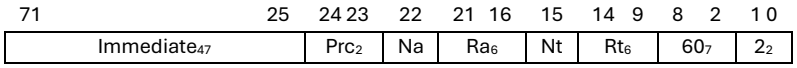
Description:

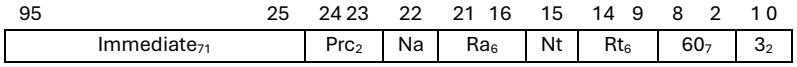
Add a register and immediate value and place the sum in the target register. The register value is shifted left once before the addition. The immediate is zero extended to the machine width.

Instruction Format: RI









Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = (Ra << 1) + immediate

Exceptions:

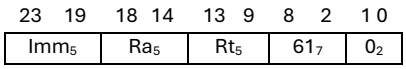
Notes:

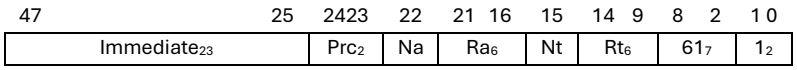
ADD4UI - Add Immediate

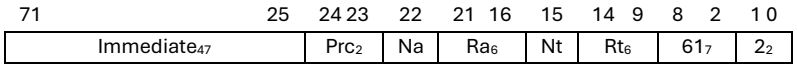
Description:

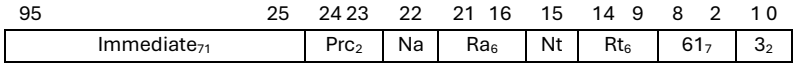
Add a register and immediate value and place the sum in the target register. The register value is shifted left twice before the addition. The immediate is zero extended to the machine width.

Instruction Format: RI









Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = (Ra << 2) + immediate

Exceptions:

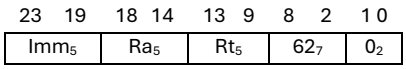
Notes:

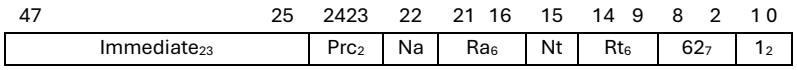
ADD8UI - Add Immediate

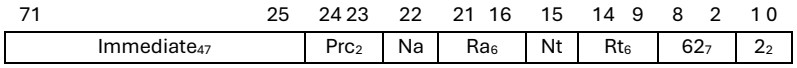
Description:

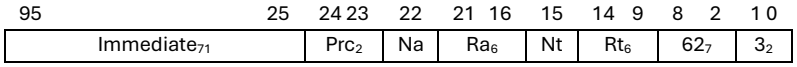
Add a register and immediate value and place the sum in the target register. The register value is shifted left three times before the addition. The immediate is zero extended to the machine width.

Instruction Format: RI









Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = (Ra << 3) + immediate

Exceptions:

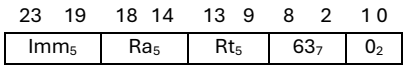
Notes:

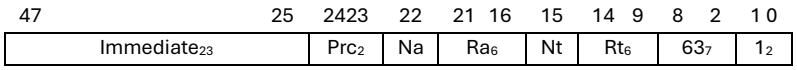
ADD16UI - Add Immediate

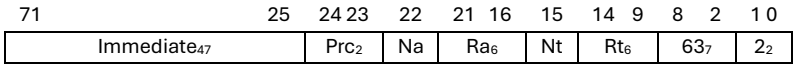
Description:

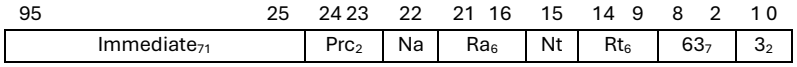
Add a register and immediate value and place the sum in the target register. The register value is shifted left four times before the addition. The immediate is zero extended to the machine width.

Instruction Format: RI









Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = (Ra << 4) + immediate

Exceptions:

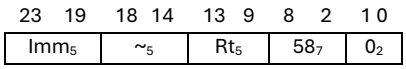
Notes:

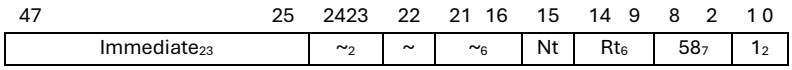
AUIIP - Add Unsigned Immediate to Instruction Pointer

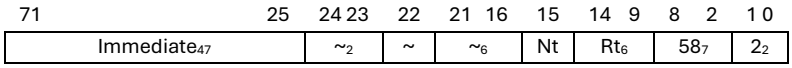
Description:

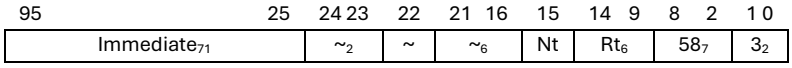
Add an immediate value to the instruction pointer and place the result in a target register. This instruction may be used in the formation of instruction pointer relative addresses.

Instruction Format: RI









Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = IP + immediate

Exceptions:

Notes:

BYTENDX – Character Index

Description:

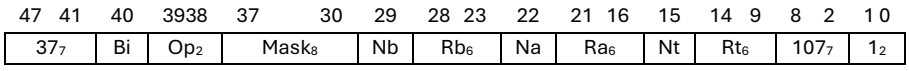
This instruction searches Ra, which is treated as an array of characters, for a character value specified by Rb and places the index of the character into the target register Rt. If the character is not found -1 is placed in the target register. A common use would be to search for a null character. The index result may vary from -1 to +7. The index of the first found byte is returned (closest to zero). The result is -1 if the character could not be found.

A masking operation may be performed on the Ra operand to allow searches for ranges of characters according to an immediate constant. For instance, the constant could be set to 0x78 and the mask 'anded' with Ra to search for any ascii control character.

Rb may be replaced by an immediate value.

Supported Operand Sizes: .b, .w, .t

Instruction Format: R3 (byte)



Op2	Mask Operation
0	a
1	a & imm
2	a | imm
3	a ^ imm

Operation:

Rt = Index of (Rb in Mask(Ra))

Execution Units: All Integer ALU's

Exceptions: none

Notes:

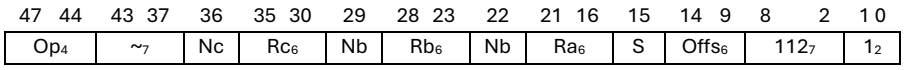
CHK/CHKU – Check Register Against Bounds

Description:

A register, Ra, is compared to two values. If the register is outside of the bounds defined by Rb and Rc then an exception specified by the cause code will occur. Comparisons may be signed (CHK) or unsigned (CHKU), indicated by 'S', 1 = signed, 0 = unsigned. The constant $Offs_6$ is multiplied by three and added to the instruction pointer address of the CHK instruction and stored on an internal stack. This allows a return to a point up to 192 bytes after the CHK. Typical values are zero or two.

The CHK type given by the Op_4 field is copied to the CAUSE CSR register as the info field.

Instruction Format: R3



Op_4	exception when not	
0	Ra >= Rb and Ra < Rc	
1	Ra >= Rb and Ra <= Rc	
2	Ra > Rb and Ra < Rc	
3	Ra > Rb and Ra <= Rc	
4	Not (Ra >= Rb and Ra < Rc)	
5	Not (Ra >= Rb and Ra <= Rc)	
6	Not (Ra > Rb and Ra < Rc)	
7	Not (Ra > Rb and Ra <= Rc)	
8	Ra >= CPL	CHKCPL – code privilege level
9	Ra <= CPL	CHKDPL – data privilege level
10	Ra == SC	Stack canary check

Operation:

IF check failed
    PUSH SR onto internal stack
    PUSH IP plus $Offs_6$ * 3 onto internal stack
    Cause = CHK
    Cause.info = Op4

IP = vector at (tvec[3] + cause*8)

Clock Cycles: 1

Execution Units: Integer ALU

Exceptions: bounds check

Notes:

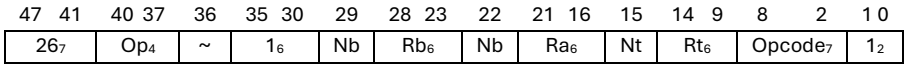
The system exception handler will typically transfer processing back to a local exception handler.

CNTLO – Count Leading Ones

Description:

This instruction counts the number of consecutive one bits beginning at the most significant bit towards the least significant bit for the register Ra.

Instruction Format: R3



$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

CNTLZ – Count Leading Zeros

Description:

This instruction counts the number of consecutive zero bits beginning at the most significant bit towards the least significant bit for the register Ra.

Instruction Format: R3

47 41	40 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
26_7	Op_4	~	0_6	Nb	Rb_6	Nb	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

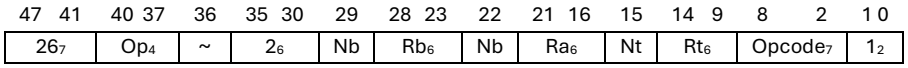
Notes:

CNTPOP – Count Population

Description:

This instruction counts the number of bits set in a register (Ra).

Instruction Format: R3



$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

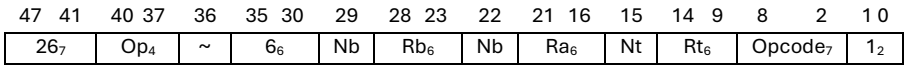
Notes:

CNTTZ – Count Trailing Zeros

Description:

This instruction counts the number of consecutive zero bits beginning at the least significant bit towards the most significant bit. This instruction can also be used to get the position of the first one bit from the right-hand side.

Instruction Format: R3



$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

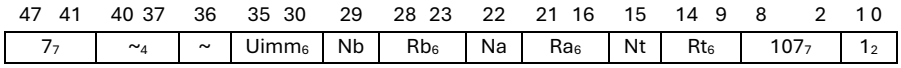
CPUID – Get CPU Info

Description:

This instruction returns general information about the core. The sum of Rb and register Ra is used as a table index to determine which row of information to return.

Supported Operand Sizes: N/A

Instruction Formats: R3



Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Info([imm+Ra+Rb])

Exceptions: none

Index	bits	Information Returned
0	0 to 63	The processor core identification number. This field is determined from an external input. It would be hard wired to the number of the core in a multi-core system.
2	0 to 63	Manufacturer name first eight chars “Finitron”
3	0 to 63	Manufacturer name last eight characters
4	0 to 63	CPU class “64BitSS”
5	0 to 63	CPU class
6	0 to 63	CPU Name “Qupls”
7	0 to 63	CPU Name
8	0 to 63	Model Number “M2”
9	0 to 63	Serial Number “1234”
10	0 to 63	Features bitmap
11	0 to 31	Instruction Cache Size (32kB)
11	32 to 63	Data cache size (64kB)
12	0 to 7	Maximum vector length – number of elements
13	0 to 7	Maximum vector length in bytes

CSR – Control and Special Registers Operations

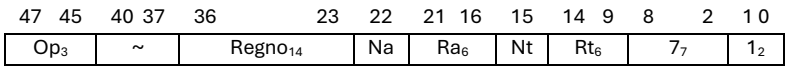
Description:

Perform an operation on a CSR. The previous value of the CSR is placed in the target register.

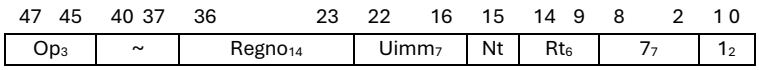
Operation	Op_3	Mnemonic
Read CSR	0	CSRRD
Write CSR	1	CSRRW
Or to CSR (set bits)	2	CSRRS
And complement to CSR (clear bits)	3	CSRRC
Reserved	4	
Write Immediate CSR	5	CSRRW
Or Immediate to CSR	6	CSRRS
And Immediate complement to CSR	7	CSRRC

Supported Operand Sizes: N/A

Instruction Formats: CSR



Instruction Formats: CSRI



Notes:

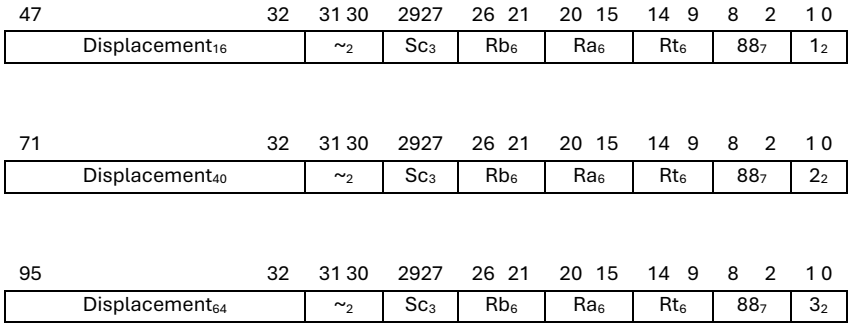
The top two bits of the Regno field correspond to the operating mode.

LOADA – Load Address

Description:

This instruction computes the scaled indexed virtual address and places it in the target register. It matches the format used by the load and store instructions.

Instruction Format: d[Ra+Rb*]



Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + Rb *Scale + displacement

Exceptions:

Notes:

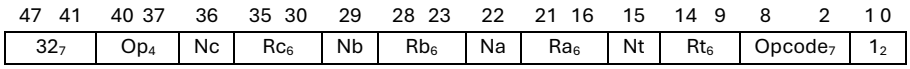
PTRDIF – Difference Between Pointers

Asm: PTRDIF Rt, Ra, Rb, Rc

Description:

Subtract two values then shift the result right. Both operands must be in a register. The right shift is provided to accommodate common object sizes. It may still be necessary to perform a divide operation after the PTRDIF to obtain an index into odd sized or large objects.

Instruction Format: R3



$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	Octa
2	Hexi

Op4		Operation	Comment
0	PTRDIF	Rt = abs(Ra – Rb) >> $Rc_{[5:0]}$	
1	AVG	Rt = (Ra + Rb) >> $Rc_{[5:0]}$, trunc	Arithmetic shift right
2	AVG	Rt = (Ra + Rb) >> $Rc_{[5:0]}$, round up	Arithmetic shift right
3		Reserved	
8	PTRDIF	Rt = abs(Ra – Rb) >> $Uimm_6$	
9	AVG	Rt = (Ra + Rb) >> $Uimm_6$, trunc	Arithmetic shift right
10	AVG	Rt = (Ra + Rb) >> $Uimm_6$, round up	Arithmetic shift right
11		Reserved	

Operation:

Rt = Abs(Ra – Rb) >> $Rc_{[5:0]}$

Clock Cycles: 1

Execution Units: ALU #0 only

Exceptions:

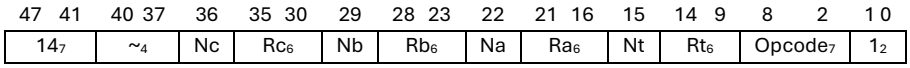
None

MAJ – Majority Logic

Description:

Determines the bitwise majority of three values in registers Ra, Rb and Rc and places the result in the target register Rt.

Instruction Format: R3



$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Execution Units: ALU #0 only

Operation:

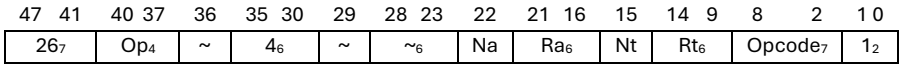
Rt = (Ra & Rb) | (Ra & Rc) | (Rb & Rc)

SQRT – Square Root

Description:

This instruction computes the integer square root of the contents of the source operand and places the result in Rt.

Instruction Format: R3



Opcode$_7$	Precision	Clocks
104	Byte parallel	
105	Wyde parallel	
106	Tetra parallel	
107	octa	72
2	hexi	

Operation:

Rt = SQRT(Ra)

Execution Units: Integer ALU #0 Only

Exceptions: none

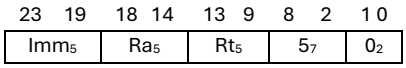
Notes:

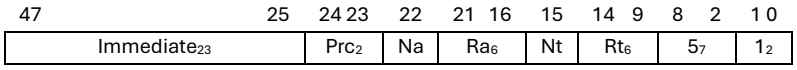
SUBFI – Subtract from Immediate

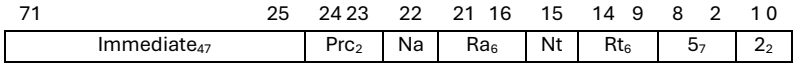
Description:

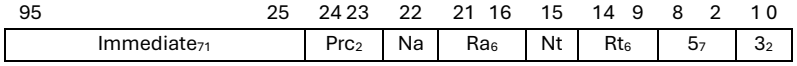
Subtract a register from an immediate value and place the difference in the target register. The immediate is sign extended to the machine width.

Instruction Format: RI









Clock Cycles: 1

Execution Units: All ALUs, all FPUs

Operation:

Rt = immediate - Ra

Exceptions:

Notes:

TETRANDX – Character Index

Description:

This instruction searches Ra, which is treated as an array of characters, for a character value specified by Rb and places the index of the character into the target register Rt. If the character is not found -1 is placed in the target register. A common use would be to search for a null character. The index result may vary from -1 to +1. The index of the first found tetra is returned (closest to zero). The result is -1 if the character could not be found.

A masking operation may be performed on the Ra operand to allow searches for ranges of characters according to an immediate constant. For instance, the constant could be set to 0x1F8 and the mask 'anded' with Ra to search for any ascii control character.

Supported Operand Sizes: .b, .w, .t

Instruction Format: R3 (tetra)

47 41	40	3938	37 30	29	28 23	22	21 16	15	14 9	8 2	1 0
39_7	Bi	Op_2	$Mask_8$	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	107_7	1_2

Op2	Mask Operation
0	a
1	a & imm
2	a | imm
3	a ^ imm

Operation:

Rt = Index of (Rb in Ra)

Execution Units: All Integer ALU's

Exceptions: none

Notes:

WYDENDX – Character Index

Description:

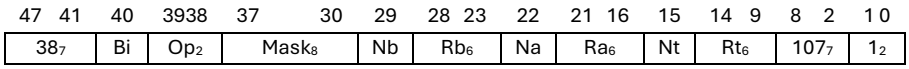
This instruction searches Ra, which is treated as an array of characters, for a character value specified by Rb and places the index of the character into the target register Rt. If the character is not found -1 is placed in the target register. A common use would be to search for a null character. The index result may vary from -1 to +3. The index of the first found wyde is returned (closest to zero). The result is -1 if the character could not be found.

A masking operation may be performed on the Ra operand to allow searches for ranges of characters according to an immediate constant. For instance, the constant could be set to 0xF8 and the mask 'anded' with Ra to search for any ascii control character.

Supported Operand Sizes: .b, .w, .t

Instruction Format: R3 (wyde)

Instruction Format: R3 (byte)



Op2	Mask Operation
0	a
1	a & imm
2	a | imm
3	a ^ imm

Operation:

Rt = Index of (Rb in Ra)

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Multiply / Divide

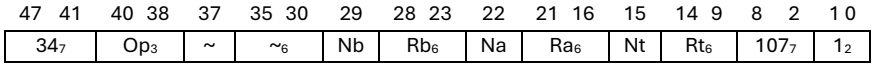
BMM – Bit Matrix Multiply

BMM Rt, Ra, Rb

Description:

The BMM instruction treats the bits of register Ra and register Rb as an 8x8 matrix and performs a bit matrix multiply of the two registers and stores the result in the target register. An alternate mnemonic for this instruction is MOR.

Instruction Format: R3



47 41	40 38	37	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
34_7	Op_3	~	$~_6$	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	107_7	1_2

Operation:

```
for I = 0 to 7
        for j = 0 to 7
                Rt.bit[i][j] = (Ra[i][0]&Rb[0][j]) | (Ra[i][1]&Rb[1][j]) | ... |
(Ra[i][7]&Rb[7][j])
```

Clock Cycles: 1

Execution Units: First Integer ALU

Exceptions: none

Notes:

The bits are numbered with bit 63 of a register representing I,j = 0,0 and bit 0 of the register representing I,j = 7,7.

DIV – Signed Division

Description:

Divide source dividend operand by divisor operand and place the quotient in the target register. All registers are integer registers. Arithmetic is signed twos-complement values.

Instruction Format: R3

47 41	40 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
17_7	Op_4	Nc	Rc_6	Nb	Rb_6	Nb	Ra_6	Nt	Rt_6	$Opcode_7$	0_2

$Opcode_7$	Precision	Clocks
104	Byte parallel	
105	Wyde parallel	
106	Tetra parallel	
107	octa	34
2	hexi	

OP_4		Mnemonic
0	Rt = (Ra / Rb) & Rc	DIV_AND
1	Rt = (Ra / Rb) | Rc	DIV_OR
2	Rt = (Ra / Rb) ^ Rc	DIV_EOR
3	Rt = (Ra / Rb) + Rc	DIV_ADD
others	Reserved	

Operation:

Rt = Ra / Rb

Execution Units: ALU #0 Only

Exceptions: DBZ

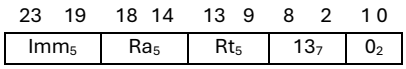
Notes:

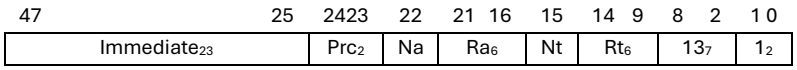
DIVI – Signed Immediate Division

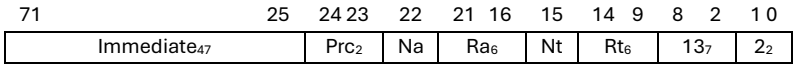
Description:

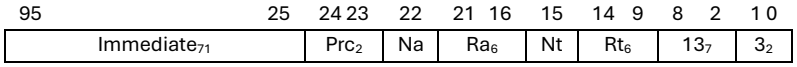
Divide source dividend operand by divisor operand and place the quotient in the target register. All registers are integer registers. Arithmetic is signed twos-complement values.

Instruction Format: RI









Operation:

Rt = Ra / Imm

Execution Units: ALU #0 Only

Exceptions: none

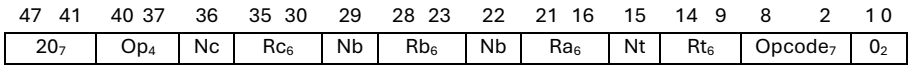
Notes:

DIVU – Unsigned Division

Description:

Divide source dividend operand by divisor operand and place the quotient in the target register. All registers are integer registers. Arithmetic is unsigned twos-complement values.

Instruction Format: R3



47 41	40 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
20_7	Op_4	Nc	Rc_6	Nb	Rb_6	Nb	Ra_6	Nt	Rt_6	$Opcode_7$	0_2

$Opcode_7$	Precision	Clocks
104	Byte parallel	
105	Wyde parallel	
106	Tetra parallel	
107	octa	34
2	hexi	

OP_4		Mnemonic
0	Rt = (Ra * Rb) & Rc	MUL_AND
1	Rt = (Ra * Rb) | Rc	MUL_OR
2	Rt = (Ra * Rb) ^ Rc	MUL_EOR
3	Rt = (Ra * Rb) + Rc	MUL_ADD
others	Reserved	

Operation:

Rt = Ra / Rb

Execution Units: ALU #0 Only

Exceptions: none

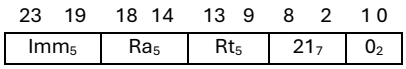
Notes:

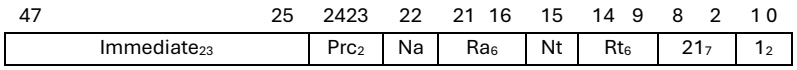
DIVUI – Unsigned Immediate Division

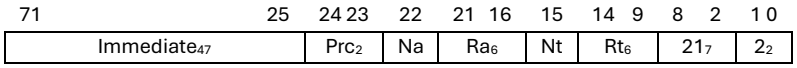
Description:

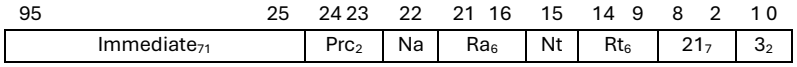
Divide source dividend operand by divisor operand and place the quotient in the target register. All registers are integer registers. Arithmetic is unsigned twos-complement values.

Instruction Format: RI









Operation:

Rt = Ra / Imm

Execution Units: ALU #0 Only

Exceptions: none

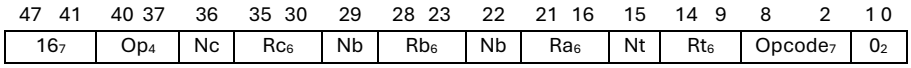
Notes:

MUL – Multiply Register-Register

Description:

Multiply two registers and place the product in the target register. All registers are integer registers. Values are treated as signed integers.

Instruction Format: R3



$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

OP_4		Mnemonic
0	Rt = (Ra * Rb) & Rc	MUL_AND
1	Rt = (Ra * Rb) | Rc	MUL_OR
2	Rt = (Ra * Rb) ^ Rc	MUL_EOR
3	Rt = (Ra * Rb) + Rc	MUL_ADD
others	Reserved	

Operation: R2

Rt = Ra * Rb + Rc

Clock Cycles: 4

Execution Units: All Integer ALUs

Exceptions: none

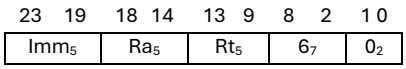
Notes:

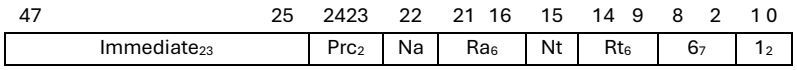
MULI - Multiply Immediate

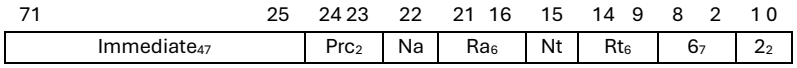
Description:

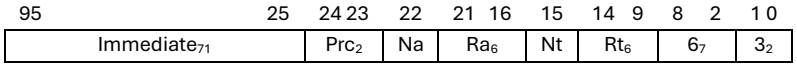
Multiply a register and immediate value and place the product in the target register. The immediate is sign extended to the machine width. Values are treated as signed integers.

Instruction Format: RI









Clock Cycles: 4

Execution Units: All ALUs

Operation:

Rt = Ra * immediate

Exceptions:

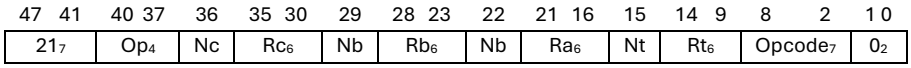
Notes:

MULSU – Multiply Signed Unsigned

Description:

Multiply two registers and place the product in the target register. All registers are integer registers. The first operand is signed, the second unsigned.

Instruction Format: R3



$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Operation: R2

Rt = Ra * Rb + Rc

Clock Cycles: 1

Execution Units: All Integer ALUs

Exceptions: none

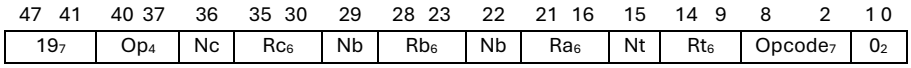
Notes:

MULU – Unsigned Multiply Register-Register

Description:

Multiply two registers and place the product in the target register. All registers are integer registers. Values are treated as unsigned integers.

Instruction Format: R3



$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

OP_4		Mnemonic
0	Rt = (Ra * Rb) & Rc	MUL_AND
1	Rt = (Ra * Rb) | Rc	MUL_OR
2	Rt = (Ra * Rb) ^ Rc	MUL_EOR
3	Rt = (Ra * Rb) + Rc	MUL_ADD
9	Rt = (Ra * Imm) | Rc	MUL_OR
others	Reserved	

Operation: R2

Rt = Ra * Rb + Rc

Clock Cycles: 4

Execution Units: All Integer ALUs

Exceptions: none

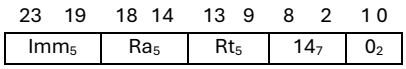
Notes:

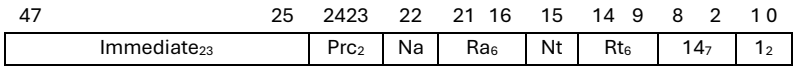
MULUI - Multiply Unsigned Immediate

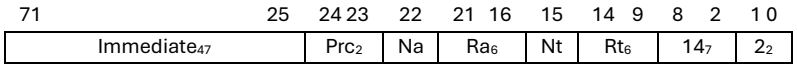
Description:

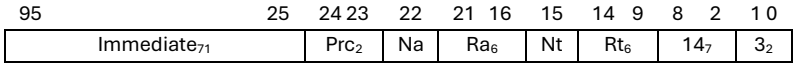
Multiply a register and immediate value and place the product in the target register. The immediate is sign extended to the machine width. Values are treated as unsigned integers.

Instruction Format: RI









Clock Cycles: 4

Execution Units: All ALUs

Operation:

Rt = Ra * immediate

Exceptions:

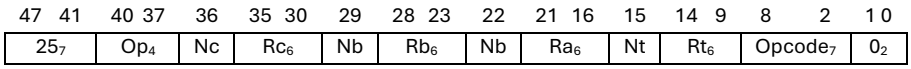
Notes:

REM – Signed Remainder

Description:

Divide source dividend operand by divisor operand and place the remainder in the target register. All registers are integer registers. Arithmetic is signed twos-complement values.

Instruction Format: R3



$Opcode_7$	Precision	Clocks
104	Byte parallel	
105	Wyde parallel	
106	Tetra parallel	
107	octa	34
2	hexi	

OP_4		Mnemonic
0	Rt = (Ra % Rb) & Rc	REM_AND
1	Rt = (Ra % Rb) | Rc	REM_OR
2	Rt = (Ra % Rb) ^ Rc	REM_EOR
3	Rt = (Ra % Rb) + Rc	REM_ADD
others	Reserved	

Operation:

Rt = Ra % Rb

Execution Units: ALU #0 Only

Exceptions: DBZ

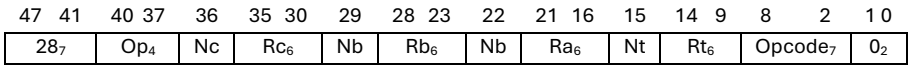
Notes:

REMU – Unsigned Remainder

Description:

Divide source dividend operand by divisor operand and place the remainder in the target register. All registers are integer registers. Arithmetic is unsigned twos-complement values.

Instruction Format: R3



$Opcode_7$	Precision	Clocks
104	Byte parallel	
105	Wyde parallel	
106	Tetra parallel	
107	octa	34
2	hexi	

OP_4		Mnemonic
0	Rt = (Ra % Rb) & Rc	REMU_AND
1	Rt = (Ra % Rb) | Rc	REMU_OR
2	Rt = (Ra % Rb) ^ Rc	REMU_EOR
3	Rt = (Ra % Rb) + Rc	REMU_ADD
others	Reserved	

Operation:

Rt = Ra % Rb

Execution Units: ALU #0 Only

Exceptions: none

Notes:

Data Movement

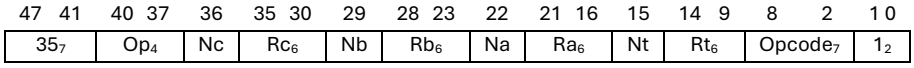
BMAP – Byte Map

Description:

First the target register is cleared, then bytes are mapped from the 8-byte source Ra into bytes in the target register. This instruction may be used to permute the bytes in register Ra and store the result in Rt. This instruction may also pack bytes, wydes or tetras. The map is determined by the low order 32-bits of register Rc. Bytes which are not mapped will end up as zero in the target register. Each nybble of the 32-bit value indicates the target byte in the target register.

Instruction Format: R3

BMAP Rt, Ra, Rb



$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Sz2	Unit Operated On
0	Bytes
1	wydes
2	tetras

Op4		
0	Byte map	Uses Rc
1	Reverse byte order	
2	Broadcast byte	
3	Mix	
4	Shuffle	
5	Alt	
6 to 15	reserved	

Operation:

Vector Operation

Execution Units: First Integer ALU

Exceptions: none

Notes:

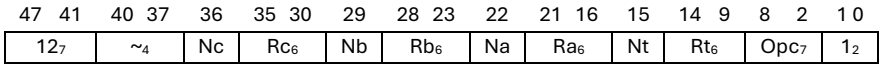
CMOV – Conditional Move if Non-Zero

CMOV Rt, Ra, Rb, Rc

Description:

If Ra is non-zero then the target register is set to Rb, otherwise the target register is to Rc.

Instruction Format: R3



47 41	40 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
12_7	$\sim_4$	Nc	Rc_6	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	Opc_7	1_2

$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

```
If Ra then
        Rt = Rb
else
        Rt = Rc
```

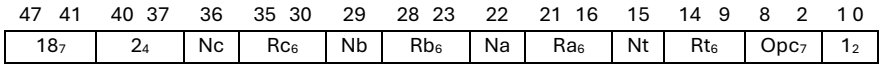
Exceptions: none

MAX3 – Maximum Signed Value

Description:

Determines the maximum of three values in registers Ra, Rb and Rc and places the result in the target register Rt. Operands values are treated as signed integers.

Instruction Format: R3



$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	Octa
2	Hexi

Execution Units: ALU #0 only

Operation:

```
IF (Ra > Rb and Ra > Rc)
        Rt = Ra
Else if (Rb > Rc)
        Rt = Rb
Else
        Rt = Rc
```

MAXU3 – Maximum Unsigned Value

Description:

Determines the maximum of three values in registers Ra, Rb and Rc and places the result in the target register Rt. Operands values are treated as unsigned integers.

Instruction Format: R3

47 41	40 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
23_7	2_4	Nc	Rc_6	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	Opc_7	1_2

$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Execution Units: ALU #0 only

Operation:

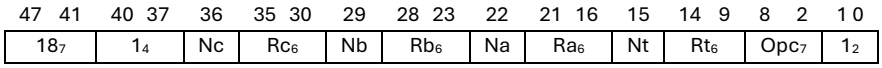
```
IF (Ra > Rb and Ra > Rc)
        Rt = Ra
Else if (Rb > Rc)
        Rt = Rb
Else
        Rt = Rc
```

MID3 – Middle Value

Description:

Determines the middle value of three values in registers Ra, Rb and Rc and places the result in the target register Rt.

Instruction Format: R3



$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Execution Units: ALU #0 only

Operation:

```
IF (Ra > Rb and Ra < Rc)
        Rt = Ra
Else if (Rb > Ra and Rb < Rc)
        Rt = Rb
Else
        Rt = Rc
```

MIDU3 – Middle Unsigned Value

Description:

Determines the middle value of three values in registers Ra, Rb and Rc and places the result in the target register Rt.

Instruction Format: R3

47 41	40 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
23_7	1_4	Nc	Rc_6	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	Opc_7	1_2

$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Execution Units: ALU #0 only

Operation:

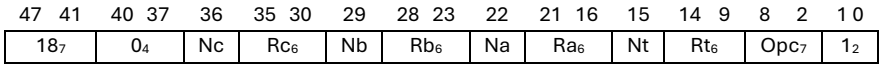
```
IF (Ra > Rb and Ra < Rc)
        Rt = Ra
Else if (Rb > Ra and Rb < Rc)
        Rt = Rb
Else
        Rt = Rc
```

MIN3 – Minimum Value

Description:

Determines the minimum of three values in registers Ra, Rb and Rc and places the result in the target register Rt. Values are treated as signed integers.

Instruction Format: R3



$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Execution Units: ALU #0 only

Operation:

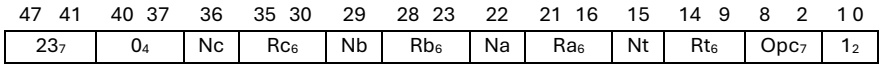
```
IF (Ra < Rb and Ra < Rc)
        Rt = Ra
Else if (Rb < Rc)
        Rt = Rb
Else
        Rt = Rc
```

MINU3 – Minimum Unsigned Value

Description:

Determines the minimum of three values in registers Ra, Rb and Rc and places the result in the target register Rt. Values are treated as unsigned integers.

Instruction Format: R3



$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Execution Units: ALU #0 only

Operation:

```
IF (Ra < Rb and Ra < Rc)
        Rt = Ra
Else if (Rb < Rc)
        Rt = Rb
Else
        Rt = Rc
```

MOVE – Move Register to Register

Description:

Move register-to-register. This instruction may move between different types of registers. Raw binary data is moved. No data conversions are applied. Some registers are accessible only in specific operating modes. Some registers are read-only. Normally referencing the stack pointer register r31 will map to the stack pointer according to the operating mode, however the 'A' bit of the instruction may be set to disable this.

Instruction Format: R1

23	22	21 16	15	14 9	8 2	1 0
A	Na	Ra_6	Nt	Rt_6	15_7	0_2

Operation: R2

Rt = Ra

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Ra_6 / Rt_6	Register file	Mode Access	RW
0 to 30	General purpose registers 0 to 30	USHM	RW
31	Safe stack pointer	SHM	RW
32	User stack pointer	USHM	RW
33	Supervisor stack pointer	SHM	RW
34	Hypervisor stack pointer	HM	RW
35	Machine stack pointer	M	RW
36 to 39	Micro-code temporaries #0 to #3	HM	RW
40	micro-code link register	HM	RW
41 to 43	Link registers	USHM	RW
44 to 63		USHM	RW

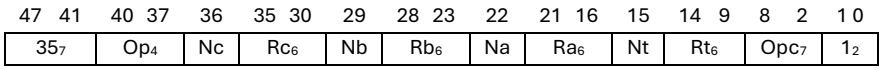
MUX – Multiplex

MUX Rt, Ra, Rb, Rc

Description:

If element Ra is set then the element of the target register is set to the corresponding element in Rb, otherwise the element in the target register is set to the corresponding element in Rc.

Instruction Format: R3



Mnemonic	$Opcode_7$	OP_4	Precision
MUXB	104	0	byte
MUXW	105	0	wyde
MUXT	106	0	tetra
MUXO	107	0	octa
MUXH	2	0	hexi
MUX	107	1	Bit

Clock Cycles: 1

Execution Units: ALU #0 only

Operation (bit):

For n = 0 to 63
    If $Ra_{[n]}$ is set then
        $Rt_{[n]} = Rb_{[n]}$
    else
        $Rt_{[n]} = Rc_{[n]}$

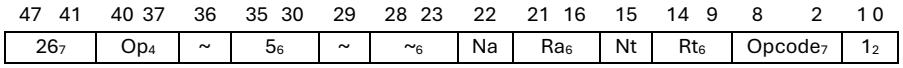
Exceptions: none

REV – Reverse Order

Description:

This instruction reverses the order of bits in Ra and stores the result in Rt.

Instruction Format: R1



$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Op_4	Precision	Mnemonic
0	bit	REVBIT
1	Bit pair	REVBITP
2	nybble	REVN
3	Byte	REVB
4	wyde	REVW
5	tetra	REVT
6	octa	REVO

Operation:

Execution Units: ALU #0 Only

Clock Cycles: 1

Exceptions: none

Notes:

SX – Sign Extend

SXB Rt, 7

Description:

A value in a target register is sign extended from the specified bit.

Instruction Format: R1

23	22	21 16	15	14 9	8 2	1 0
0	1	Imm_6	Nt	Rt_6	3_7	0_2

23	22	21 16	15	14 9	8 2	1 0
1	Na	Ra_6	Nt	Rt_6	3_7	0_2

Operation:

Clock Cycles:

Execution Units: All Integer ALUs

Exceptions: none

Notes:

ZX – Zero Extend

ZXB Rt, 7

Description:

A value in a target register is zero extended from the specified bit.

Instruction Format: R1

23	22	21 16	15	14 9	8 2	1 0
0	0	Imm_6	Nt	Rt_6	3_7	0_2

Operation:

Clock Cycles:

Execution Units: All Integer ALUs

Exceptions: none

Notes:

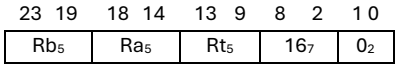
Logical Operations

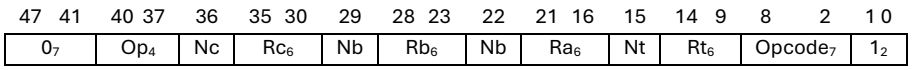
AND – Bitwise And

Description:

And three registers and place the result in the target register. All register values are integers. If Rc is not used, it is assumed to be zero.

Instruction Format: R3





Operation:

$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

OP_4		Mnemonic
0	Rt = (Ra & Rb) & Rc	AND_AND
1	Rt = (Ra & Rb) | Rc	AND_OR
2	Rt = (Ra & Rb) ^ Rc	AND_EOR
3	Rt = (Ra & Rb) + Rc	AND_ADD
4 to 7	Reserved	

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

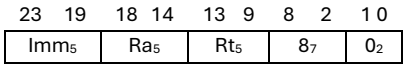
Notes:

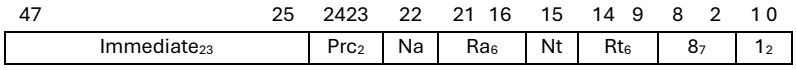
ANDI - Add Immediate

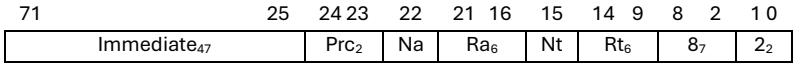
Description:

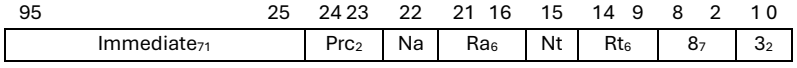
And a register and immediate value and place the result in the target register. The immediate is one extended to the machine width.

Instruction Format: RI









Clock Cycles: 1

Execution Units: All ALUs, all FPUs

Operation:

Rt = Ra + immediate

Exceptions:

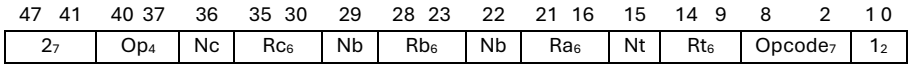
Notes:

EOR – Bitwise Exclusive Or

Description:

Bitwise exclusively or three registers and place the result in the target register. All registers are integer registers.

Instruction Format: R3



OP_4		Mnemonic
0	Rt = (Ra ^ Rb) & Rc	EOR_AND
1	Rt = (Ra ^ Rb) | Rc	EOR_OR
2	Rt = (Ra ^ Rb) ^ Rc	EOR_EOR
3	Rt = (Ra ^ Rb) + Rc	EOR_ADD
4 to 14	Reserved	
15	Rt = (^Ra) ^ (^Rb) ^ (^Rc)	PAR (triple parity)

Operation: R3

Rt = Ra ^ Rb ^ Rc

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

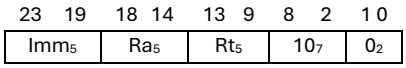
Notes:

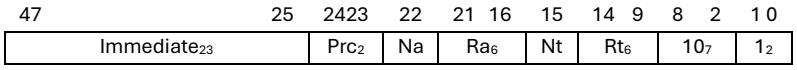
EORI – Exclusive Or Immediate

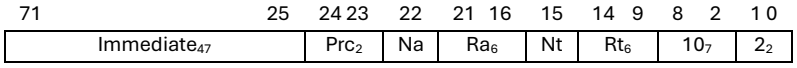
Description:

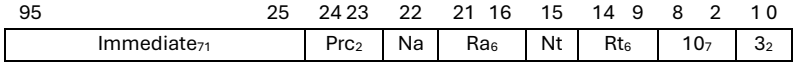
Exclusive Or a register and immediate value and place the sum in the target register. The immediate is zero extended to the machine width.

Instruction Format: RI









Clock Cycles: 1

Execution Units: All ALUs, all FPUs

Operation:

Rt = Ra ^ immediate

Exceptions:

Notes:

OR – Bitwise Or

Description:

Bitwise or three registers and place the result in the target register. All registers are integer registers.

Instruction Format: R3

47 41	40 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
1_7	Op_4	Nc	Rc_6	Nb	Rb_6	Nb	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Operation:

OP_4		Mnemonic
0	Rt = (Ra | Rb) & Rc	OR_AND
1	Rt = (Ra | Rb) | Rc	OR_OR
2	Rt = (Ra | Rb) ^ Rc	OR_EOR
3	Rt = (Ra | Rb) + Rc	OR_ADD
4 to 14	Reserved	
15	Rt = (Ra & Rb) | (Ra & Rc) | (Rb & Rc)	MAJ

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

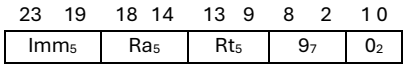
Notes:

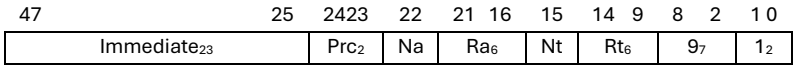
ORI – Inclusive Or Immediate

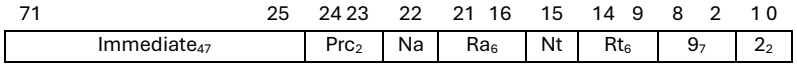
Description:

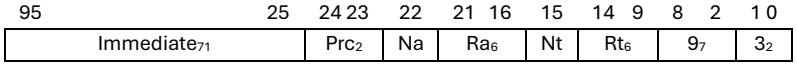
Inclusive Or a register and immediate value and place the sum in the target register. The immediate is zero extended to the machine width.

Instruction Format: RI









Clock Cycles: 1

Execution Units: All ALUs, all FPUs

Operation:

Rt = Ra | immediate

Exceptions:

Notes:

Comparison Operations

Overview

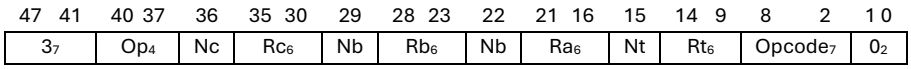
There are two basic types of comparison operators. The first type, compare, returns a bit vector indicating the relationship between the operands, the second type, set, returns a false or a constant depending on the result of the comparison.

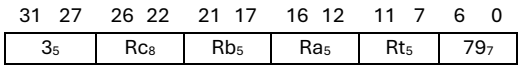
CMP - Comparison

Description:

Compare two source operands and place the result in the target register. The result is a bit vector identifying the relationship between the two source operands as signed integers. The compare may be cumulative by or'ing the result of previous comparisons with the current one. This may be used to test for the presence or absence of data in an array.

Instruction Format: R3





Opc_7	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

OP_4		Mnemonic
0	Rt = (Ra ? Rb) & Rc	CMP_AND
1	Rt = (Ra ? Rb) | Rc	CMP_OR
2	Rt = (Ra ? Rb) ^ Rc	CMP_EOR
3	Rt = (Ra ? Rb) + Rc	CMP_ADD
4 to 14	Reserved	
15	Range Check	CMP_RNG

Fn_4	Unsigned	Signed	Comparison Test
0	EQ	ENOR	Equal
1	NE	EOR	not equal
2	LTU	LT	less than
3	LEU	LE	less than or equal
4	GEU	GE	greater or equal
5	GTU	GT	greater than
6	BC		Bit clear
7	BS		Bit set
8	BC		Bit clear imm
9	BS		Bit set imm
10	NANDB	NAND	And zero
11	ANDB	AND	And non-zero
12	NORB	NOR	Or zero
13	ORB	OR	Or non-zero
15			
others			reserved

Operation:

Rt = Ra ? Rb

Rt = (Ra ? Rb) | Rc ; cumulative

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

Rt Bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	a == b
1	NE	< > not equal	a <> b
2	LT	< less than	a < b
3	LE	<= less than or equal	a <= b
4	GE	>= greater than or equal	a >= b
5	GT	> greater than	a > b
6	BC	Bit clear	!a[b]
7	BS	Bit set	a[b]

Range Check:

Rt Bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	GEL		a >=b and a < c
1	GELE		a >= b and a <= c
2	GL		a > b and a < c
3	GLE		a > b and a <= c
4	NGEL		Not (a >= b and a < c)
5	NGELE		Not (a >= b and a <= c)
6	NGL		Not (a > b and a < c)
7	NGLE		Not (a > b and a <= c)

CMPI – Compare Immediate

Description:

Compare two source operands and place the result in the target register. The result is a vector identifying the relationship between the two source operands as signed integers.

Operation:

Rt = Ra ? Imm

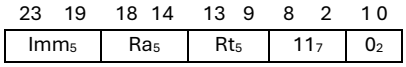
Clock Cycles: 1

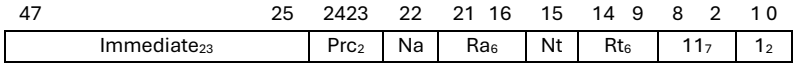
Execution Units: All Integer ALUs, all FPUs

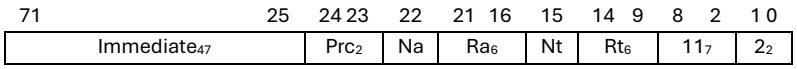
Exceptions: none

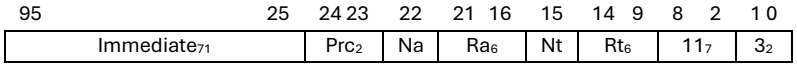
Notes:

Instruction Format: RI









Rt Bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	a == b
1	NE	< > not equal	a <> b
2	LT	< less than	a < b
3	LE	<= less than or equal	a <= b
4	GE	>= greater than or equal	a >= b
5	GT	> greater than	a > b
6	BC	Bit clear	!a[b]

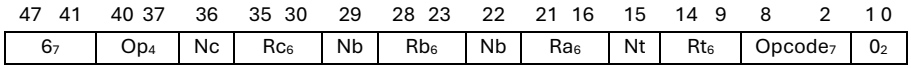
7	BS	Bit set	a[b]

CMPU – Unsigned Comparison

Description:

Compare two source operands and place the result in the target register. The result is a bit vector identifying the relationship between the two source operands as signed integers. The compare may be cumulative by or'ing the result of previous comparisons with the current one. This may be used to test for the presence or absence of data in an array.

Instruction Format: R3



Opc_7	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

OP_4		Mnemonic
0	Rt = (Ra ? Rb) & Rc	CMPU_AND
1	Rt = (Ra ? Rb) | Rc	CMPU_OR
2	Rt = (Ra ? Rb) ^ Rc	CMPU_EOR
3	Rt = (Ra ? Rb) + Rc	CMPU_ADD
4 to 14	Reserved	
15	Range Check	CMPU_RNG

Fn_4	Unsigned	Signed	Comparison Test
0	EQ	ENOR	Equal
1	NE	EOR	not equal
2	LTU	LT	less than
3	LEU	LE	less than or equal
4	GEU	GE	greater or equal
5	GTU	GT	greater than
6	BC		Bit clear
7	BS		Bit set
8	BC		Bit clear imm
9	BS		Bit set imm
10	NANDB	NAND	And zero
11	ANDB	AND	And non-zero
12	NORB	NOR	Or zero
13	ORB	OR	Or non-zero
15			
others			reserved

Operation:

Rt = Ra ? Rb

Rt = (Ra ? Rb) | Rc ; cumulative

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

Rt Bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	a == b
1	NE	< > not equal	a <> b
2	LT	< less than	a < b
3	LE	<= less than or equal	a <= b
4	GE	>= greater than or equal	a >= b
5	GT	> greater than	a > b
6	BC	Bit clear	!a[b]
7	BS	Bit set	a[b]

Range Check:

Rt Bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	GEL		a >=b and a < c
1	GELE		a >= b and a <= c
2	GL		a > b and a < c
3	GLE		a > b and a <= c
4	NGEL		Not (a >= b and a < c)
5	NGELE		Not (a >= b and a <= c)
6	NGL		Not (a > b and a < c)
7	NGLE		Not (a > b and a <= c)

CMPUI – Compare Immediate

Description:

Compare two source operands and place the result in the target register. The result is a vector identifying the relationship between the two source operands as unsigned integers.

Operation:

Rt = Ra ? Imm

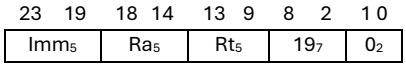
Clock Cycles: 1

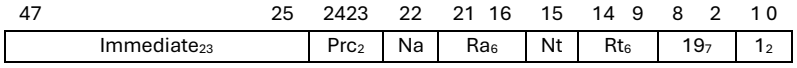
Execution Units: All Integer ALUs, all FPUs

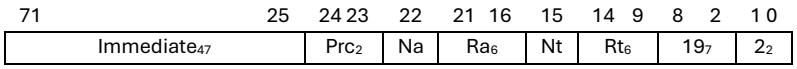
Exceptions: none

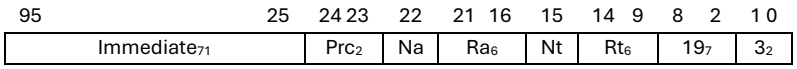
Notes:

Instruction Format: RI









Rt Bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	a == b
1	NE	< > not equal	a <> b
2	LT	< less than	a < b
3	LE	<= less than or equal	a <= b
4	GE	>= greater than or equal	a >= b
5	GT	> greater than	a > b
6	BC	Bit clear	!a[b]

7	BS	Bit set	a[b]

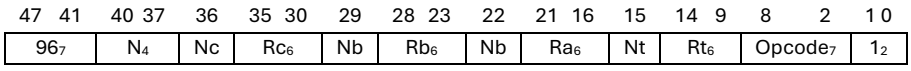
CMOVEQ – Conditional Move if Equal

CMOVEQ Rt, Ra, Rb, Rc, Imm_4

Description:

Compare two source operands {Ra, RB} for equality and if equal place Rc in target register, otherwise place N in target register. N may be either the original value of the target register, or a constant from -7 to +7.

Instruction Format: R3



Operation: R3

Rt = (Ra == Rb) ? Rc : N==8 ? Rt : N

Opc_7	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Clock Cycles: 1

Execution Units: All Integer ALUs

Exceptions: none

Notes:

CMOVLE – Conditional Move if Less Than or Equal

CMOVLE Rt, Ra, Rb, Rc, Imm_4

Description:

Compare two source operands {Ra, RB} for Ra less than or equal to Rb and if so place Rc in target register, otherwise place N in target register. N may be either the original value of the target register, or a constant from -7 to +7.

Instruction Format: R3

47 41	40 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
99_7	N_4	Nc	Rc_6	Nb	Rb_6	Nb	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Operation: R3

Rt = (Ra <= Rb) ? Rc : N==8 ? Rt : N

Opc_7	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Clock Cycles: 1

Execution Units: All Integer ALUs

Exceptions: none

Notes:

CMOVLT – Conditional Move if Less Than

CMOVLT Rt, Ra, Rb, Rc, Imm_4

Description:

Compare two source operands {Ra, RB} for Ra less than Rb and if so place Rc in target register, otherwise place N in target register. N may be either the original value of the target register, or a constant from -7 to +7.

Instruction Format: R3

47 41	40 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
98_7	N_4	Nc	Rc_6	Nb	Rb_6	Nb	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Operation: R3

Rt = (Ra < Rb) ? Rc : N==8 ? Rt : N

Opc_7	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Clock Cycles: 1

Execution Units: All Integer ALUs

Exceptions: none

Notes:

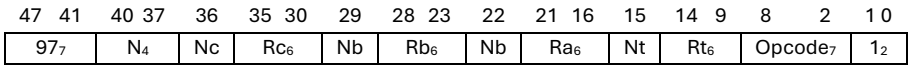
CMOVNE – Conditional Move if Not Equal

CMOVNE Rt, Ra, Rb, Rc, Imm_4

Description:

Compare two source operands {Ra, RB} for inequality and if not equal place Rc in target register, otherwise place N in target register. N may be either the original value of the target register, or a constant from -7 to +7.

Instruction Format: R3



Operation: R3

Rt = (Ra == Rb) ? Rc : N==8 ? Rt : N

Opc_7	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Clock Cycles: 1

Execution Units: All Integer ALUs

Exceptions: none

Notes:

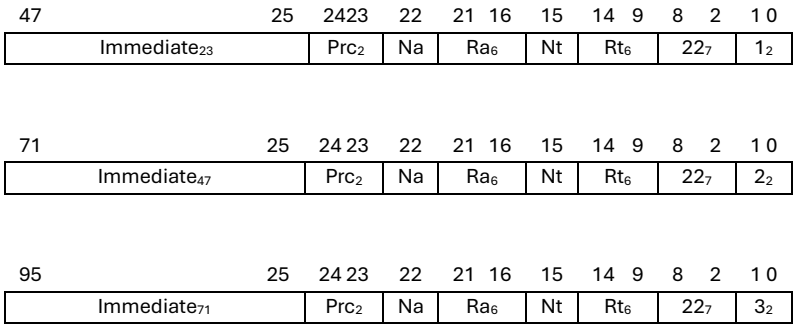
SEQI –Set if Equal

SEQ Rt, Ra, imm

Description:

Compare two source operands for equality and place the result in the target predicate register. The result is a Boolean value of one. The first operand is in a register, the second operand is an immediate constant.

Instruction Format: R3



Operation: R3

Rt = (Ra == Imm) ? 1 : Rt

Clock Cycles: 1

Execution Units: All Integer ALUs

Exceptions: none

Notes:

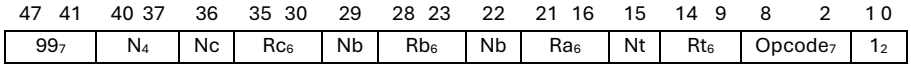
SLE – Set if Less or Equal

SLE Rt, Ra, Rb, Rc, Imm_4

Description:

Compare two source operands {Ra, RB} for signed less than or equal to and if Ra is less than or equal to Rb place Rc in target register, otherwise place N in target register. N may be either the original value of the target register, or a constant from -7 to +7.

Instruction Format: R3



Operation: R3

Rt = (Ra <= Rb) ? Rc : N==8 ? Rt : N

Opc_7	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa
2	hexi

Clock Cycles: 1

Execution Units: All Integer ALUs

Exceptions: none

Notes:

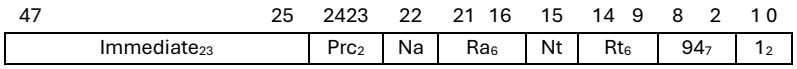
ZSEQI –Zero or Set if Equal

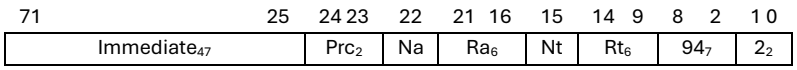
ZSEQ Rt, Ra, imm

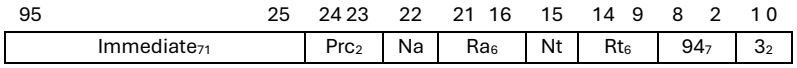
Description:

Compare two source operands for equality and place the result in the target predicate register. The result is a Boolean value of one or zero. The first operand is in a register, the second operand is an immediate constant.

Instruction Format: R3







Operation: R3

Rt = (Ra == Imm) ? 1 : 0

Clock Cycles: 1

Execution Units: All Integer ALUs

Exceptions: none

Notes:

Shift, Rotate and Bitfield Operations

Shift instructions can take the place of some multiplication and division instructions. Some architectures provide shifts that shift only by a single bit. Others use counted shifts, the original 80x88 used multiple clock cycles to shift by an amount stored in the CX register. Table888 and Thor use a barrel shifter to allow shifting by an arbitrary amount in a single clock cycle. Shifts are infrequently used, and a barrel (or funnel) shifter is relatively expensive in terms of hardware resources.

Qupls2 has a full complement of shift instructions including rotates.

Precision

Qupls2 supports four precisions for shift operations.

$Opcode_7$	Precision
80	Byte parallel
81	Wyde parallel
82	Tetra parallel
83	octa

CLR – Clear Bit Field

Description:

Clear a bitfield in a target register pair. This is an alternate mnemonic for the deposit, DEP, instruction where the value to deposit is zero.

Instruction Format: SHIFT

47 44	43 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
5_4	$\sim_7$	Nc	Rc_6	0	0_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Instruction Format: SHIFTI

47 44	43 37	36 30	29	28 23	22	21 16	15	14 9	8 2	1 0
13_4	ME_7	MB_7	0	0_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Operation:

{Ra, Rt}[ME:MB] = 0

Clock Cycles:

Execution Units: All Integer ALUs

Exceptions: none

Notes:

COM – Complement Bit Field

COM Rt, Ra, Rc

Description:

This is an alternate mnemonic for the DEPXOR instruction where the value to xor is -1.

A bit field in the source operand is one's complemented and the result placed in the target register. Rb specifies the first bit of the bitfield, Rc specifies the last bit of the bitfield. Immediate constants may be substituted for Rb and Rc.

Instruction Format: SHIFT

47 44	43 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
6_4	~$_7$	Nc	Rc_6	1	0_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Instruction Format: SHIFTI

47 44	43 37	36 30	29	28 23	22	21 16	15	14 9	8 2	1 0
14_4	ME_7	MB_7	1	0_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Operation:

{Ra, Rt}[ME:MB] = {Ra, Rt}[ME:MB] ^ -1

Clock Cycles: 1

Execution Units: All Integer ALUs

Exceptions: none

Notes:

DEP –Deposit Bitfield

Description:

Deposit a value into a bit-field which may span two words.

Left shift an operand value by an operand value and place the result in the target registers {Ra, Rt}. The register shifted is specified by Rb. The third operand may be either a register specified by the Rc field of the instruction, or an immediate value.

If Rc is used, mask-begin and mask-end are specified by bits 0 to 7 and 8 to 15 of the value in Rc respectively.

Instruction Format: SHIFT

47 44	43 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
5_4	~$_7$	Nc	Rc_6	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Instruction Format: SHIFTI

47 44	43 37	36 30	29	28 23	22	21 16	15	14 9	8 2	1 0
13_4	ME_7	MB_7	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Operation:

{Ra, Rt}[ME:MB] = Rb

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

DEPXOR –Deposit Bitfield

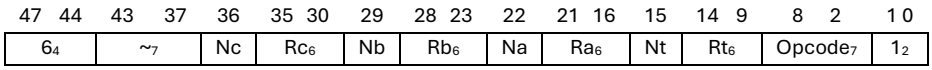
Description:

Exclusively Or deposit a value into a bit-field which may span two words.

Left shift an operand value by an operand value and xor the result to the target registers {Ra, Rt}. The register shifted is specified by Rb. The third operand may be either a register specified by the Rc field of the instruction, or an immediate value.

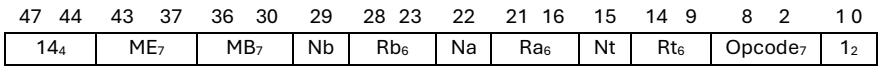
If Rc is used, mask-begin and mask-end are specified by bits 0 to 7 and 8 to 15 of the value in Rc respectively.

Instruction Format: SHIFT



47 44	43 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
6_4	~$_7$	Nc	Rc_6	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Instruction Format: SHIFTI



47 44	43 37	36 30	29	28 23	22	21 16	15	14 9	8 2	1 0
14_4	ME_7	MB_7	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Operation:

{Ra, Rt}[ME:MB] = {Ra, Rt}[ME:MB] ^ Rb

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

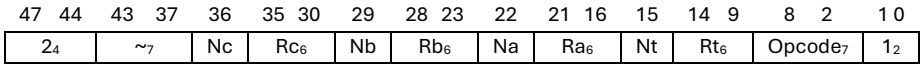
EXT – Extract Bit Field

EXT Rt, Ra, Rb, Rc

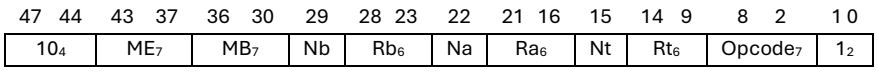
Description:

A bit field is extracted from the source operand, sign extended, and the result placed in the target register. This is an alternate mnemonic for the SRAP instruction.

Instruction Format: SHIFT



Instruction Format: SHIFTI



$Opcode_7$	Precision
80	Byte parallel
81	Wyde parallel
82	Tetra parallel
83	octa

Operation:

Rt = sign extend({Rb,Ra}[ME:MB])

Clock Cycles:

Execution Units: All Integer ALUs

Exceptions: none

Notes:

EXTU – Extract Unsigned Bit Field

EXTU Rt, Ra, Rb, Rc

Description:

A bit field is extracted from the source operand, zero extended, and the result placed in the target register. This is an alternate mnemonic for the SRLP instruction.

Instruction Format: SHIFT

47 44	43 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
1_4	~$_7$	Nc	Rc_6	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Instruction Format: SHIFTI

47 44	43 37	36 30	29	28 23	22	21 16	15	14 9	8 2	1 0
9_4	ME_7	MB_7	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

$Opcode_7$	Precision
80	Byte parallel
81	Wyde parallel
82	Tetra parallel
83	octa

Operation:

Rt = zero extend({Rb,Ra}[ME:MB])

Clock Cycles:

Execution Units: All Integer ALUs

Exceptions: none

Notes:

ROL –Rotate Left

ROL Rt, Ra, N

Description:

This is an alternate mnemonic for the SLLP instruction. Rotate left an operand value by an operand value and place the result in the target register. The most significant bits are shifted into the least significant bits. The first operand must be in a register specified by Ra and Rb. The second operand may be either a register specified by the Rc field of the instruction, or an immediate value.

Ra and Rb should specify the same register for a rotate to occur.

Instruction Format: SHIFT

47 44	43	42 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
0_4	H	~$_6$	Nc	Rc_6	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Instruction Format: SHIFTI

47 44	43	42 37	36 30	29	28 23	22	21 16	15	14 9	8 2	1 0
8_4	H	~$_6$	Imm_7	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Operation:

Rt = {Rb,Ra} << Rc

Or

Rt = {Rb,Ra} << Imm

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

SET – Set Bit Field

SET Rt, Ra, Rc

Description:

Set a bitfield in a target register pair. This is an alternate mnemonic for the deposit, DEP, instruction where the value to deposit is -1.

Instruction Format: SHIFT

47 44	43 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
5_4	~$_7$	Nc	Rc_6	1	0_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Instruction Format: SHIFTI

47 44	43 37	36 30	29	28 23	22	21 16	15	14 9	8 2	1 0
13_4	ME_7	MB_7	1	0_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Operation:

{Ra, Rt}[ME:MB] = -1

Clock Cycles: 1

Execution Units: All Integer ALUs

Exceptions: none

Notes:

ROR –Rotate Right

Description:

Rotate right an operand value by an operand value and place the result in the target register. The least significant bits are shifted into the most significant bits. The first operand must be in a register specified by Ra and Rb. The second operand may be either a register specified by the Rc field of the instruction, or an immediate value.

Ra and Rb should specify the same register for a rotate to occur.

Instruction Format: SHIFT

47 44	43 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
1_4	$\sim_7$	Nc	Rc_6	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Instruction Format: SHIFTI

47 44	43 37	36 30	29	28 23	22	21 16	15	14 9	8 2	1 0
9_4	ME_7	MB_7	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

$Opcode_7$	Precision
80	Byte parallel
81	Wyde parallel
82	Tetra parallel
83	octa

Operation:

Rt = {Rb,Ra} >> Rc

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

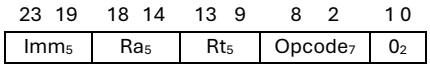
SLL –Shift Left Logical

SLL Rt, Ra, N

Description:

Left shift an operand value by an operand value and place the result in the target register. The second operand is an immediate value.

Instruction Format: SLL



Operation:

Rt = {Ra} << Imm

Operation Size: .b, .w, .t, .o

Execution Units: integer ALU

Exceptions: none

Example:

SLLP –Shift Left Logical Pair

Description:

Left shift a pair of operand values by an operand value and place the result in the target register. The pair of registers shifted is specified by Ra (lower bits), Rb (upper bits). The third operand may be either a register specified by the Rc field of the instruction, or an immediate value. If the 'H' bit is set, the upper 64-bits of the result are transferred to the target register, Rt.

This instruction may be used to perform a rotate operation by specifying the same register for Ra and Rb. It may also be used to implement a ring counter by inverting Ra during the shift.

Instruction Format: SHIFT

47 44	43	42 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
0_4	H	~$_6$	Nc	Rc_6	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Instruction Format: SHIFTI

47 44	43	42 37	36 30	29	28 23	22	21 16	15	14 9	8 2	1 0
8_4	H	~$_6$	Imm_7	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Operation:

Rt = {Rb,Ra} << Rc

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

SRAP –Shift Right Arithmetic Pair

Description:

Right shift an operand value by an operand value and place the sign extended result in the target register. The first operand must be in a pair of registers specified by {Rb,Ra}. The second operand may be either a register specified by the Rc field of the instruction, or an immediate value.

Instruction Format: SHIFT

47 44	43 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
2_4	$\sim_7$	Nc	Rc_6	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Instruction Format: SHIFTI

47 44	43 37	36 30	29	28 23	22	21 16	15	14 9	8 2	1 0
10_4	ME_7	MB_7	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

$Opcode_7$	Precision
80	Byte parallel
81	Wyde parallel
82	Tetra parallel
83	octa

Operation:

Rt = {Rb,Ra} >> Rc

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

SRAPRZ –Shift Right Arithmetic Pair, Round toward Zero

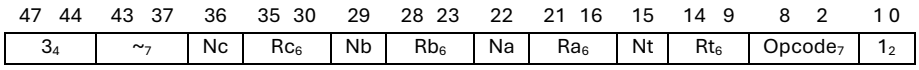
Description:

This instruction may be used to extract bit-fields spanning two words, or it may be used to perform a simple right shift operation with appropriate settings for the mask fields.

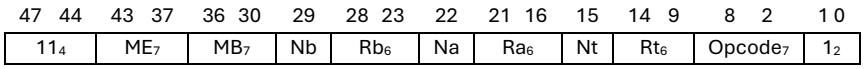
Right shift an operand value by an operand value and place the sign extended result in the target register. The first operand must be in a pair of registers specified by {Rb,Ra}. The second operand may be either a register specified by the Rc field of the instruction, or an immediate value.

If the result is negative, then it is rounded up.

Instruction Format: SHIFT



Instruction Format: SHIFTI



$Opcode_7$	Precision
80	Byte parallel
81	Wyde parallel
82	Tetra parallel
83	octa

Operation:

Rt = {Rb,Ra} >> Rc

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

SRAPRU –Shift Right Arithmetic Pair, Round Up

Description:

This instruction may be used to extract bit-fields spanning two words, or it may be used to perform a simple right shift operation with appropriate settings for the mask fields.

Right shift an operand value by an operand value and place the sign extended result in the target register. The first operand must be in a pair of registers specified by {Rb,Ra}. The second operand may be either a register specified by the Rc field of the instruction, or an immediate value.

One is added to the result if there was a carry out of the LSB.

Instruction Format: SHIFT

47 44	44 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
4_4	~$_7$	Nc	Rc_6	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Instruction Format: SHIFTI

47 44	43 37	36 30	29	28 23	22	21 16	15	14 9	8 2	1 0
12_4	ME_7	MB_7	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

$Opcode_7$	Precision
80	Byte parallel
81	Wyde parallel
82	Tetra parallel
83	octa

Operation:

Rt = {Rb,Ra} >> Rc

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

SRLP –Shift Right Logical Pair

Description:

This instruction may be used to extract bit-fields spanning two words, or it may be used to perform a simple right shift operation with appropriate settings for the mask fields.

Right shift an operand value by an operand value and place the result in the target register. The first operand must be in a pair of registers specified by {Rb,Ra}. The second operand may be either a register specified by the Rc field of the instruction, or an immediate value.

Instruction Format: SHIFT

47 44	43 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
1_4	$\sim_7$	Nc	Rc_6	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Instruction Format: SHIFTI

47 44	43 37	36 30	29	28 23	22	21 16	15	14 9	8 2	1 0
9_4	Me_7	MB_7	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

$Opcode_7$	Precision
80	Byte parallel
81	Wyde parallel
82	Tetra parallel
83	octa

Operation:

Rt = {Rb,Ra} >> Rc

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

Floating-Point Operations

Precision

Three storage formats are supported for binary floats: 64-bit double precision and 32-bit single precision.

$Opcode_7$	Qualifier	Precision
56	H	Half precision
48	S	Single precision
16	D	Double precision
90	Q	Quad precision

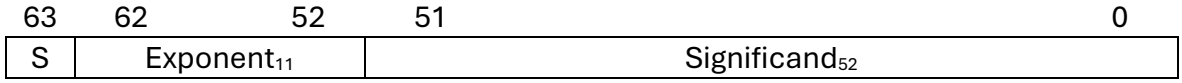
Representations

Binary Floats

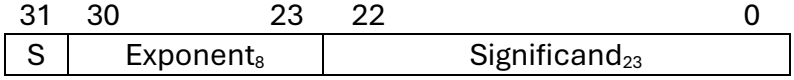
Double Precision, Float:64

The core uses a 64-bit double precision binary floating-point representation.

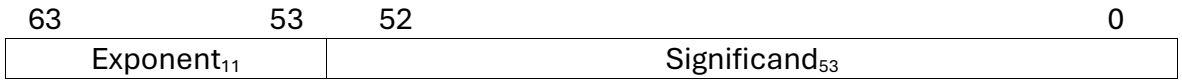
Double Precision



Single Precision, float



Double Precision, Two's Complement Form:



Decimal Floats

The core uses a 96-bit densely packed decimal double precision floating-point representation.

95	94 90	89 80	79 0
S	$Combo_5$	$Exponent_{10}$	$Significand_{80}$

The significand stores 25 densely packed decimal digits. One whole digit before the decimal point.

The exponent is a power of ten as a binary number with an offset of 1535. Range is 10^{-1535} to 10^{1536}

48-bit single precision decimal floating point:

47	46 42	41 34	33 0
S	$Combo_5$	$Exponent_8$	$Significand_{34}$

The significand stores 11 DPD digits. One whole digit before the decimal point.

NaN Boxing

Lower precision values are 'NaN boxed' meaning all the bits needed to extend the value to the width of the register are filled with ones. The sign bit of the number is preserved. Thus, lower precision values encountered in calculations are treated as NaNs.

Example: NaN boxed double precision value.

127	126 111	110 0	
S	$Exponent_{16}$	$Significand_{111}$	
S	FFFFh	7FFFFFFFFFFFh	Double Precision $Float_{64}$

Rounding Modes

Binary Float Rounding Modes

Rm3	Rounding Mode
000	Round to nearest ties to even
001	Round to zero (truncate)
010	Round towards plus infinity
011	Round towards minus infinity
100	Round to nearest ties away from zero
101	Reserved
110	Reserved

111	Use rounding mode in float control register

Decimal Float Rounding Modes

Rm3	Rounding Mode
000	Round ceiling
001	Round floor
010	Round half up
011	Round half even
100	Round down
101	Reserved
110	Reserved
111	Use rounding mode in float control register

1-7-12

Immediate$_{17}$	Ra$_6$	Func$_3$	Rt$_6$	Opcode$_5$	P$_3$

Opcode2$_5$	Rc$_6$	Rb$_6$	Ra$_6$	Func$_3$	Rt$_6$	Opcode$_5$	P$_3$

	0	1	2	3	4	5	6	7
0								
1								
2	Load	Store						
3	Fp20	Fp40	Fp80					

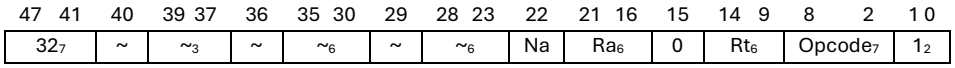
FABS – Absolute Value

Description:

This instruction computes the absolute value of the contents of the source operand and places the result in Rt. The sign bit of the value is cleared. No rounding occurs.

Integer Instruction Format: FLT1

FABS Rt, Ra



Operation:

Ft = Abs(Fa)

Execution Units: All FPUs, All ALUs

Clock Cycles: 1

Exceptions: none

Notes:

$Opcode_7$		Precision	Clocks
56	H	Half precision	1
57	S	Single precision	1
58	D	Double precision	1
90	Q	Quad precision	1

FADD –Float Addition

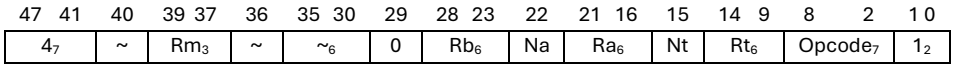
FADD Rt, Ra, Rb

Description:

Add two source operands and place the sum in the target register. Values are treated as floating-point values.

Supported Operand Sizes:

Instruction Format: FLT



Operation:

Rt = Ra + Rb

Execution Units: All FPU's

Exceptions: none

Notes:

$Opcode_7$		Precision	Clocks
56	H	Half precision	8
57	S	Single precision	8
58	D	Double precision	8
90	Q	Quad precision	8

FCMP - Comparison

FCMP Rt, Ra, Rb

Description:

Compare two source operands and place the result in the target register. The result is a vector identifying the relationship between the two source operands as floating-point values. This instruction may compare against lower precision immediate values to conserve code space. The source operands are floating-point values, the target operand is an integer. No rounding occurs.

Instruction Format: FLT

47 41	40	39 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
13_7	~	Rm_3	~	$\sim_6$	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Operation:

Rt = Ra ? Rb or Rt = Ra ? Imm

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Rt bit	Mnem.	Meaning	Test
		Float Compare Results	
0	EQ	equal	!nan & eq
1	NE	not equal	!eq
2	GT	greater than	!nan & !eq & !lt & !inf
3	UGT	Unordered or greater than	Nan || (!eq & !lt & !inf)
4	GE	greater than or equal	Eq || (!nan & !lt & !inf)
5	UGE	Unordered or greater than or equal	Nan || (!lt || eq)
6	LT	Less than	Lt & (!nan & !inf & !eq)
7	ULT	Unordered or less than	Nan | (!eq & lt)
8	LE	Less than or equal	Eq | (lt & !nan)
9	ULE	unordered less than or equal	Nan | (eq | lt)
10	GL	Greater than or less than	!nan & (!eq & !inf)
11	UGL	Unordered or greater than or less than	Nan | !eq

12	ORD	Greater than less than or equal / ordered	!nan
13	UN	Unordered	Nan
14		Reserved	
15		reserved	

FCONST – Load Float Constant

Description:

This instruction loads a constant from the constant ROM and places the value in Rt.

Integer Instruction Format: R1

FCONST Rt, N

47 41	40	39 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
$?_7$	~	$\sim_3$	~	0_6	~	4_6	~	N_6	Nt	Rt_6	$Opcode_7$	1_2

Clock Cycles: 1

Operation:

Ft = FConst[N]

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

N_6	Binary64	Decimal	
0	3fe0000000000000	0.5	
1	3ff0000000000000	1.0	
2	4000000000000000	2.0	
3	3ff8000000000000	1.5	
4	0x5FE6EB50C7B537A9		Lomont reciprocal square root magic
21			
22			
23			
57	7FF0000000000000		infinity
58	7FF0000000000001		Nan – infinity - infinity
59	7FF0000000000002		Nan – infinity / infinity
60	7FF0000000000003		Nan – zero / zero
61	7FF0000000000004		Nan – infinity * zero
62	7FF0000000000005		Nan – square root of infinity
63	7FF0000000000006		Nan – square root of negative

FCOS – Float Cosine

Description:

This instruction computes an approximation of the co-sine value of the contents of the source operand and places the result in Rt.

Integer Instruction Format: R1

FCOS Rt, Ra

47 41	40	39 37	36 33	32 30	29	28 23	22	21 16	15	14 9	8 2	1 0
65_7	~	Rm_3	$\sim_4$	Prc_3	~	$\sim_6$	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Operation:

Rt = cos(Ra)

Execution Units: FPU #0

Exceptions: none

Notes:

Prc_3		Precision	Clocks
0	H	Half precision	24
1	S	Single precision	36
2	D	Double precision	42
3	Q	Quad precision	

$Opcode_7$		Precision	Clocks
56	H	Half precision	1
57	S	Single precision	1
58	D	Double precision	1
90	Q	Quad precision	1

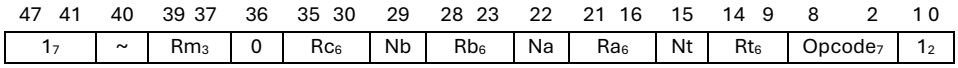
FMA –Float Multiply and Add

Description:

Multiply two source operands, add a third operand and place the result in the target register. All register values are treated as floating-point values.

Instruction Format: FLT3

FMA Rt, Ra, Rb, Rc



$Opcode_7$	Precision
56	
57	Half
58	Single
59	Double
90	Quad

Operation:

Rt = Ra * Rb + Rc

Clock Cycles: 8

Execution Units: All FPUs

Exceptions: none

Notes:

FMS –Float Multiply and Subtract

FMS Rt, Ra, Rb, Rc

Description:

Multiply two source operands, subtract a third operand and place the result in the target register. All register values are treated as quad precision floating-point values.

Instruction Format: FLT3

47 41	40	39 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
1_7	~	Rm_3	1	Rc_6	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Operation:

Rt = Ra * Rb - Rc

Clock Cycles: 8

Execution Units: All FPUs

Exceptions: none

Notes:

FNABS – Negative Absolute Value

FNABS Rt, Ra

Description:

This instruction computes the negative absolute value of the contents of the source operand and places the result in Rt. The sign bit of the value is set. No rounding occurs.

Integer Instruction Format: FLT1

47 41	40	39 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
32_7	~	$\sim_3$	~	$\sim_6$	~	$\sim_6$	Na	Ra_6	1	Rt_6	$Opcode_7$	1_2

Operation:

Ft = Fnabs(Fa)

Execution Units: All FPUs, All ALUs

Clock Cycles: 1

Exceptions: none

Notes:

$Opcode_7$		Precision	Clocks
56	H	Half precision	1
57	S	Single precision	1
58	D	Double precision	1
90	Q	Quad precision	1

FSLT – Float Set if Less Than

FSLT Rt, Ra, Rb

Description:

Compares two source operands for less than and places the result in the target register. The target register is a predicate register. The result is a Boolean true or false. Positive and negative zero are considered equal. For FSLT if either operand is a NaN zero the result is false. No rounding occurs. This instruction may also test for greater than by swapping operands.

Instruction Formats:

47 41	40	39 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
10_7	~	$\sim_3$	~	$\sim_6$	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Operation:

Rt = Ra < Rb

Clock Cycles: 1

Execution Units: All FPU's

Exceptions: none

Notes:

FSNE – Float Set if Not Equal

FSNE Rt, Ra, Bb

Description:

Compares two source operands for equality and places the result in the target predicate register. The result is a Boolean true or false. Positive and negative zero are considered equal. 16, 32, 64, and 128-bit immediates are supported. No rounding occurs.

Instruction Format:

47 41	40	39 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
9_7	~	$\sim_3$	~	$\sim_6$	Nb	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Operation:

Rt = Ra != Rb or Rt = Ra != Imm

Clock Cycles: 1

Execution Units: All FPU's

Exceptions: none

Notes:

FSQRT – Floating point square root

Description:

Take the square root of the floating-point number in register Ra and place the result into target register Rt. The sign bit (bit 63) of the register is set to zero. This instruction can generate NaNs.

Instruction Format: FLT

47 41	40	39 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
40_7	~	Rm_3	~	$\sim_6$	~	$\sim_6$	Na	Ra_6	0	Rt_6	$Opcode_7$	1_2

Operation:

Rt = fsqrt (Ra)

Clock Cycles: 72

Execution Units: Floating Point

FSUB –Float Subtraction

FSUB Rt, Ra, Rb

Description:

Subtract two source operands and place the difference in the target register.

Supported Operand Sizes:

Instruction Format: FLT

47 41	40	39 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
4_7	~	Rm_3	~	$~_6$	1	Rb_6	Na	Ra_6	Nt	Rt_6	$Opcode_7$	1_2

Operation:

Rt = Ra - Rb

Clock Cycles: 8

Execution Units: All FPUs

Exceptions: none

Notes:

FTRUNC – Truncate Value

Description:

The FTRUNC instruction truncates off the fractional portion of the number leaving only a whole value. For instance, ftrunc(1.5) equals 1.0. Ftrunc does not change the representation of the number. To convert a value to an integer in a fixed-point representation see the FTOI instruction.

Instruction Format: FLT

47 41	40	39 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
53_7	~	Rm_3	~	$~_6$	~	$~_6$	Na	Ra_6	0	Rt_6	$Opcode_7$	1_2

Clock Cycles: 1

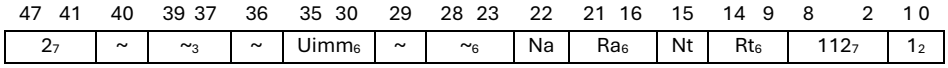
Execution Units: All FPUs

FTX – Trigger Floating Point Exceptions

Description:

This instruction triggers floating point exceptions. The Exceptions to trigger are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: EX



Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Enabled
0	global invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

Load / Store Instructions

Overview

Addressing Modes

Load and store instructions primary addressing mode is scaled indexed addressing. 24-bit forms of loads and stores use register indirect with displacement addressing.

Data Type

Six data types are supported:

Opcode Load	Opcode Store	Data Type
64	72	Integer
65	73	Unsigned integer
66	74	Floating point
67	75	Decimal floating point
68	76	Posit
69	77	Capability

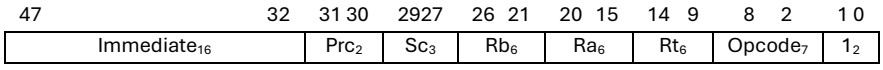
Precision

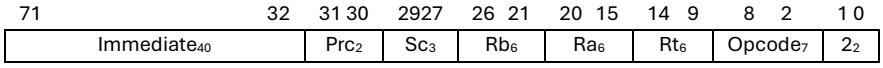
Prc_2	Integer	Float	Decimal Float	Posit	Capability
0	Byte		Double		64-bit
1	Wyde	half	Quad	half	128-bit
2	Tetra	single		Single	
3	Octa	double		double	

Scaled Indexed with Displacement Format

For scaled indexed with displacement format the load or store address is the sum of register Ra, scaled register Rb, and a displacement constant found in the instruction.

Instruction Format: d[Ra+Rb*]





95 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Immediate_{64}$	Prc_2	Sc_3	Rb_6	Ra_6	Rt_6	$Opcode_7$	3_2

CACHE <cmd>, <ea>

Description:

Issue command to cache controller.

Instruction Format: d[Ra+Rb*]

47 33	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Immediate_{16}$	Prc_2	Sc_3	Rb_6	Ra_6	Cmd_6	70_7	1_2

Notes:

Cmd_6	Cache	
???000	Ins.	Invalidate cache
???001	Ins.	Invalidate line
???010	TLB	Invalidate TLB
???011	TLB	Invalidate TLB entry
000???	Data	Invalidate cache
001???	Data	Invalidate line
010???	Data	Turn cache off
011???	Data	Turn cache on

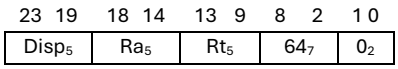
LDsz Rn, <ea> - Load Register

Description:

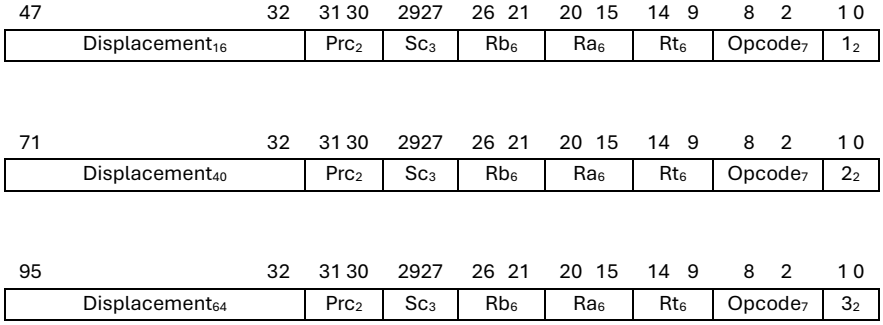
Load register Rt with data from source. The source value is sign extended to the machine width. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement.

The capabilities tag bit of the register is always cleared, unless a capabilities load instruction is taking place.

Instruction Format: d[Ra]



Instruction Format: d[Ra+Rb*Sc]



Prc_2

	Prc			
Opcode	0	1	2	3
64	LDB	LDW	LDT	LDO
65	LDBU	LDWU	LDTU	
66		FLDH	FLDS	FLDD
67	DFLDD	DFLDQ		
68		PLDW	PLDT	PLDO
69	LDCAPD	LDCAPQ		
70				CACHE

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDB Rn, <ea> - Load Byte

Description:

Load register Rt with a byte of data from source. The source value is sign extended to the machine width. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement.

The capabilities tag bit of the register is always cleared, unless a capabilities load instruction is taking place.

Instruction Format: d[Ra+Rb*Sc]

47 32	31 30	29 27	26 21	20 15	14 9	8 2	1 0
$Displacement_{16}$	0_2	Sc_3	Rb_6	Ra_6	Rt_6	64_7	1_2

71 32	31 30	29 27	26 21	20 15	14 9	8 2	1 0
$Displacement_{40}$	0_2	Sc_3	Rb_6	Ra_6	Rt_6	64_7	2_2

95 32	31 30	29 27	26 21	20 15	14 9	8 2	1 0
$Displacement_{64}$	0_2	Sc_3	Rb_6	Ra_6	Rt_6	64_7	3_2

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDBU Rn, <ea> - Load Unsigned Byte

Description:

Load register Rt with a byte of data from source. The source value is zero extended to the machine width. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement.

The capabilities tag bit of the register is always cleared, unless a capabilities load instruction is taking place.

Instruction Format: d[Ra+Rb*Sc]

47 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{16}$	0_2	Sc_3	Rb_6	Ra_6	Rt_6	65_7	1_2

71 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{40}$	0_2	Sc_3	Rb_6	Ra_6	Rt_6	65_7	2_2

95 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{64}$	0_2	Sc_3	Rb_6	Ra_6	Rt_6	65_7	3_2

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDO Rn, <ea> - Load Octa Byte

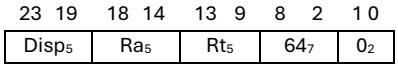
Description:

Load register Rt with an octa byte of data from source. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement.

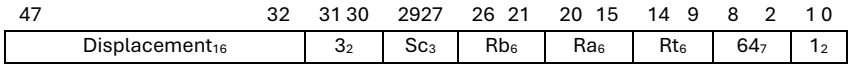
The capabilities tag bit of the register is always cleared, unless a capabilities load instruction is taking place.

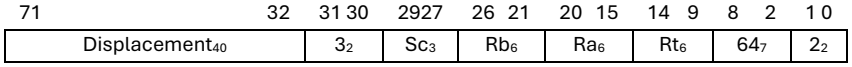
Instruction Format: d[Ra]

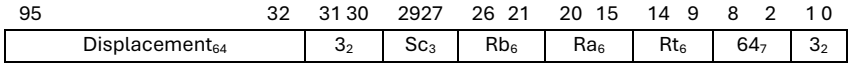
For this format, the displacement is shift left three times before use.



Instruction Format: d[Ra+Rb*Sc]







Execution Units: AGEN, MEM

Exceptions:

Notes:

LOAD Rn, <ea> - Load

Description:

This is an alternate mnemonic for the LDO instruction. Load register Rt with an octa byte of data from source. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement.

The capabilities tag bit of the register is always cleared, unless a capabilities load instruction is taking place.

Instruction Format: d[Ra]

For this format, the displacement is shift left three times before use.

23 19	18 14	13 9	8 2	1 0
$Disp_5$	Ra_5	Rt_5	64_7	0_2

Instruction Format: d[Ra+Rb*Sc]

47 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{16}$	3_2	Sc_3	Rb_6	Ra_6	Rt_6	64_7	1_2

71 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{40}$	3_2	Sc_3	Rb_6	Ra_6	Rt_6	64_7	2_2

95 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{64}$	3_2	Sc_3	Rb_6	Ra_6	Rt_6	64_7	3_2

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDT Rn, <ea> - Load Tetra

Description:

Load register Rt with a tetra byte (4 bytes) of data from source. The source value is sign extended to the machine width. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement.

The capabilities tag bit of the register is always cleared, unless a capabilities load instruction is taking place.

Instruction Format: d[Ra+Rb*Sc]

47 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{16}$	2_2	Sc_3	Rb_6	Ra_6	Rt_6	64_7	1_2

71 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{40}$	2_2	Sc_3	Rb_6	Ra_6	Rt_6	64_7	2_2

95 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{64}$	2_2	Sc_3	Rb_6	Ra_6	Rt_6	64_7	3_2

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDTU Rn, <ea> - Load Unsigned Tetra

Description:

Load register Rt with a tetra byte of data from source. The source value is zero extended to the machine width. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement.

The capabilities tag bit of the register is always cleared, unless a capabilities load instruction is taking place.

Instruction Format: d[Ra+Rb*Sc]

47 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{16}$	2_2	Sc_3	Rb_6	Ra_6	Rt_6	65_7	1_2

71 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{40}$	2_2	Sc_3	Rb_6	Ra_6	Rt_6	65_7	2_2

95 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{64}$	2_2	Sc_3	Rb_6	Ra_6	Rt_6	65_7	3_2

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDW Rn, <ea> - Load Wyde

Description:

Load register Rt with a wyde of data from source. The source value is sign extended to the machine width. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement.

The capabilities tag bit of the register is always cleared, unless a capabilities load instruction is taking place.

Instruction Format: d[Ra+Rb*Sc]

47 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{16}$	1_2	Sc_3	Rb_6	Ra_6	Rt_6	64_7	1_2

71 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{40}$	1_2	Sc_3	Rb_6	Ra_6	Rt_6	64_7	2_2

95 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{64}$	1_2	Sc_3	Rb_6	Ra_6	Rt_6	64_7	3_2

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDWU Rn, <ea> - Load Unsigned Wyde

Description:

Load register Rt with a wyde of data from source. The source value is zero extended to the machine width. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement.

The capabilities tag bit of the register is always cleared, unless a capabilities load instruction is taking place.

Instruction Format: d[Ra+Rb*Sc]

47 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{16}$	1_2	Sc_3	Rb_6	Ra_6	Rt_6	65_7	1_2

71 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{40}$	1_2	Sc_3	Rb_6	Ra_6	Rt_6	65_7	2_2

95 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{64}$	1_2	Sc_3	Rb_6	Ra_6	Rt_6	65_7	3_2

Execution Units: AGEN, MEM

Exceptions:

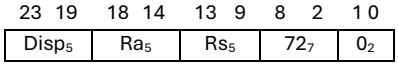
Notes:

STsz Rn, <ea> - Store Register

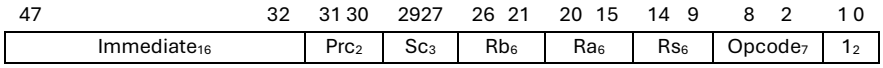
Description:

Store register Rs to memory. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement.

Instruction Format: d[Ra]



Instruction Format: d[Ra+Rb*Sc]



Prc_2

	Prc			
Opcode	0	1	2	3
72	STB	STW	STT	STO
73	STIB	STIW	STIT	STIO
74		FSTH	FSTS	FSTD
75				DFSTD
76		PSTW	PSTT	PSTO
77	STCAPD	STCAPQ		
78	STPTRT	STPTR		

Execution Units: AGEN, MEM

Exceptions:

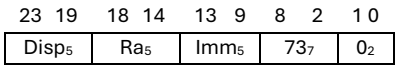
Notes:

STIsz $N, <ea> - Store Immediate to Memory

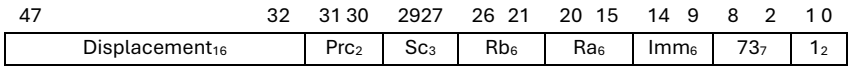
Description:

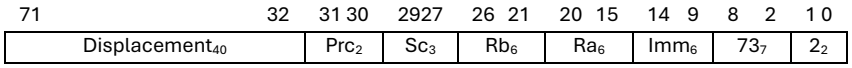
Store an immediate value to memory. The immediate value stored may be extended up to 64-bits with a postfix instruction. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement.

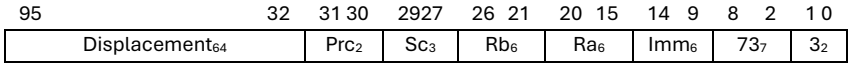
Instruction Format: d[Ra]



Instruction Format: d[Ra+Rb*Sc]







Prc_2

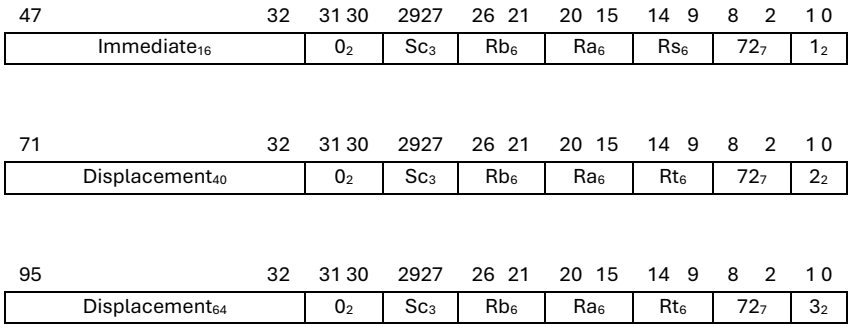
	Prc			
Opcode	0	1	2	3
72	STB	STW	STT	STO
73	STIB	STIW	STIT	STIO
74		FSTH	FSTS	FSTD
75				DFSTD
76		PSTW	PSTT	PSTO

STB Rn, <ea> - Store Register

Description:

Store the lowest byte from register Rs to memory. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement.

Instruction Format: d[Ra+Rb*Sc]



Execution Units: AGEN, MEM

Exceptions:

Notes:

STO Rn, <ea> - Store Register

Description:

Store register Rs to memory. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement.

Instruction Format: d[Ra]

For this format, the displacement is shift left three times before use.

23 19	18 14	13 9	8 2	1 0
$Disp_5$	Ra_5	Rs_5	72_7	0_2

Instruction Format: d[Ra+Rb*Sc]

47 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Immediate_{16}$	3_2	Sc_3	Rb_6	Ra_6	Rs_6	72_7	1_2

71 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{40}$	3_2	Sc_3	Rb_6	Ra_6	Rt_6	72_7	2_2

95 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{64}$	3_2	Sc_3	Rb_6	Ra_6	Rt_6	72_7	3_2

Execution Units: AGEN, MEM

Exceptions:

Notes:

STORE Rn, <ea> - Store Register

Description:

This is an alternate mnemonic for the STO instruction. Store register Rs to memory. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement.

Instruction Format: d[Ra]

For this format, the displacement is shift left three times before use.

23 19	18 14	13 9	8 2	1 0
$Disp_5$	Ra_5	Rs_5	72_7	0_2

Instruction Format: d[Ra+Rb*Sc]

47 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Immediate_{16}$	3_2	Sc_3	Rb_6	Ra_6	Rs_6	72_7	1_2

71 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{40}$	3_2	Sc_3	Rb_6	Ra_6	Rt_6	72_7	2_2

95 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{64}$	3_2	Sc_3	Rb_6	Ra_6	Rt_6	72_7	3_2

Execution Units: AGEN, MEM

Exceptions:

Notes:

STPTR Rn, <ea> - Store Pointer

Description:

Store a 64-bit pointer from register Rs to memory. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement. Storing a pointer to memory also updates the hardware card table, indicating roughly where the pointer was stored.

Instruction Format: d[Ra]

For this format, the displacement is shift left three times before use.

23 19	18 14	13 9	8 2	1 0
$Disp_5$	Ra_5	Rs_5	78_7	0_2

Instruction Format: d[Ra+Rb*Sc]

47 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Immediate_{16}$	1_2	Sc_3	Rb_6	Ra_6	Rs_6	78_7	1_2

71 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{40}$	1_2	Sc_3	Rb_6	Ra_6	Rt_6	78_7	2_2

95 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{64}$	1_2	Sc_3	Rb_6	Ra_6	Rt_6	78_7	3_2

Execution Units: AGEN, MEM

Exceptions:

Notes:

STT Rn, <ea> - Store Register

Description:

Store the lowest tetra from register Rs to memory. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement.

Instruction Format: d[Ra+Rb*Sc]

47 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Immediate_{16}$	2_2	Sc_3	Rb_6	Ra_6	Rs_6	72_7	1_2

71 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{40}$	2_2	Sc_3	Rb_6	Ra_6	Rt_6	72_7	2_2

95 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{64}$	2_2	Sc_3	Rb_6	Ra_6	Rt_6	72_7	3_2

Execution Units: AGEN, MEM

Exceptions:

Notes:

STW Rn, <ea> - Store Register

Description:

Store the lowest wyde from register Rs to memory. The memory address is the value in register Ra plus the value in register Rb scaled by 1,2,4,8,16,32, 64, or 128 plus a 16, 40 or 64-bit displacement.

Instruction Format: d[Ra+Rb*Sc]

47 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Immediate_{16}$	1_2	Sc_3	Rb_6	Ra_6	Rs_6	72_7	1_2

71 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{40}$	1_2	Sc_3	Rb_6	Ra_6	Rt_6	72_7	2_2

95 32	31 30	2927	26 21	20 15	14 9	8 2	1 0
$Displacement_{64}$	1_2	Sc_3	Rb_6	Ra_6	Rt_6	72_7	3_2

Execution Units: AGEN, MEM

Exceptions:

Notes:

Branch / Flow Control Instructions

Overview

Mnemonics

There are mnemonics for specifying the comparison method. Floating-point comparisons prefix the branch mnemonic with 'F' as in FBEQ. Decimal-floating point comparisons prefix the branch mnemonic with 'DF' as in DFBEQ. And finally posit comparisons prefix the branch mnemonic with a 'P' as in 'PBEQ'. There is no prefix for integer branches. For branches that increment register Ra, the mnemonic is prefixed with an 'I' as in 'IBNE'.

Predicated Execution

Branch instructions will execute only if both the predicate and branch condition are true.

Conditions

Conditional branches branch to the target address only if the condition is true. The condition is determined by the comparison or logical / arithmetic operation of two general-purpose registers.

> *The original Thor machine used instruction predicates to implement conditional branching. Another instruction was required to set the predicate before branching. Combining compare and branch in a single instruction may reduce the dynamic instruction count. An issue with comparing and branching in a single instruction is that it may lead to a wider instruction format.*

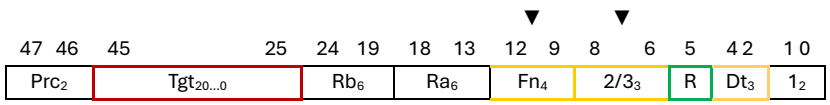
Conditional Branch Format

Branches are 48-bit opcodes.

> *A 32-bit opcode does not leave a large enough target field for all cases and would end up using two or more instructions to implement most branches. With the prospect of using two instructions to perform compare then branches as many architectures do, it is more space efficient to simply use a wider instruction format.*

Branch Conditions

The branch opcode determines the condition under which the branch will execute.



3x	2x	Data Type Compared
30h	28h	(Unsigned) Address
31h	29h	(Signed) Integers
32h	2Ah	Reserved
33h	2Bh	Decimal Float
34h	2Ch	Binary Float
35h	2Dh	Posit
36h	2Eh	Incr. Integer
37h	2Fh	Decr. Integer

Integer / Address Conditions

Fn_4	Unsigned	Signed	Comparison Test
2	LTU	LT	less than
4	GEU	GE	greater or equal
3	LEU	LE	less than or equal
5	GTU	GT	greater than
0	EQ / ENORB	ENOR	Equal
1	NE / EORB	EOR	not equal
6	BC		Bit clear
7	BS		Bit set
8	BC		Bit clear imm
9	BS		Bit set imm
10	NANDB	NAND	And zero
11	ANDB	AND	And non-zero
12	NORB	NOR	Or zero
13	ORB	OR	Or non-zero
15	IRQ	BADDO	IPL > SR.IM
others			reserved

Float Conditions

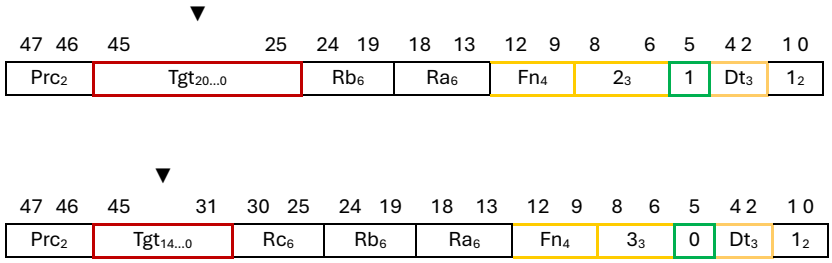
Fn_4	Mnem.	Meaning	Test
0	EQ	equal	!nan & eq
1	NE	not equal	!eq
2	GT	greater than	!nan & !eq & !lt & !inf
3	UGT	Unordered or greater than	Nan || (!eq & !lt & !inf)
4	GE	greater than or equal	Eq || (!nan & !lt & !inf)
5	UGE	Unordered or greater than or equal	Nan || (!lt || eq)
6	LT	Less than	Lt & (!nan & !inf & !eq)
7	ULT	Unordered or less than	Nan | (!eq & lt)
8	LE	Less than or equal	Eq | (lt & !nan)
9	ULE	unordered less than or equal	Nan | (eq | lt)
10	GL	Greater than or less than	!nan & (!eq & !inf)
11	UGL	Unordered or greater than or less than	Nan | !eq
12	ORD	Greater than less than or equal / ordered	!nan
13	UN	Unordered	Nan
14		Reserved	
15		reserved	

Branch Target

Conditional Branches

For conditional branches, the target address is formed in one of three ways. **One**, as the sum of the instruction pointer and a constant specified in the instruction. Relative branches have a range of approximately ±3MB or 22.5 displacement bits. The target field contains an instruction number relative displacement to the target location. This is the byte displacement divided by three. Encoding targets in this way allows fewer bits to be used to encode the target. Within a subroutine, instructions will always be a multiple of three bytes apart. **Two**, as the absolute address specified by the target address field multiplied by three. Absolute address branching must be selected by setting the 'ab' bit in the processor status register. **Three**, the target address may come from the contents of register Rc.

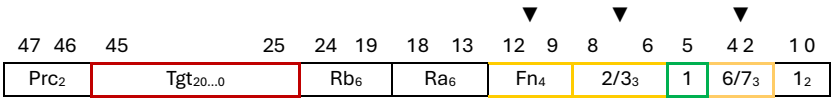
> *The target displacement field is recommended to be at least 16-bits. It is possible to get by with a displacement as small as 12-bits before a significant percentage of branches must be implemented as two or more instructions. The author decided to use a division by three since instructions are multiples of three bytes in size and the target must be a multiple of three bytes away from the branches IP. Dividing by three effectively adds 1 1/2 more displacement bits.*



R	Target
1	Displacement / Address
0	Register Indirect (Rc)

Incrementing / Decrementing Branches

Branches may increment or decrement the Ra register by one after performing the branch comparison or logical operation. The opcode field of the instruction indicates when a change should occur. Incrementing or decrementing branches make use of both the flow control unit and an ALU at the same time.



Opcode	Effect on Ra
Other	No change
46/54	Increment Ra
47/55	Decrement Ra

Unconditional Branches

Note that for unconditional branches the target displacement field is byte relative. This occurs because code functions or subroutines may be relocated at byte addresses. An unconditional subroutine branch call is usually performed to go outside of the current subroutine to a target routine that may be at any byte address. The target displacement field is large enough to accommodate a $\pm 2^{35}$ range.

Jumps and Subroutine Calls

The unconditional jump (JMP, CJMP) and jump-to-subroutine (JSR, CJSR) instructions use scaled indexed addressing to perform direct address jumps and memory indirect jumps. Refer to the documentation of the instructions for more details.

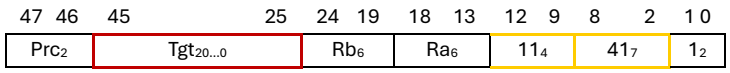
BAND –Branch if Logical And True

BAND Ra, Rb, label

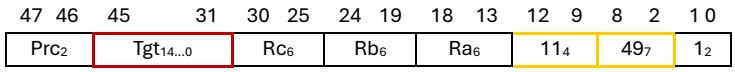
Description:

Branch if the logical and of source operands results in a non-zero value. The displacement is relative to the address of the branch instruction. This ‘and’ operation reduces the source operand values to a Boolean true or false before performing the operation. A non-zero value is considered true, zero is considered false.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

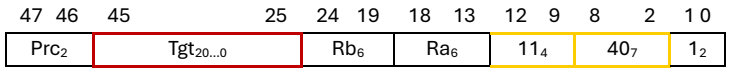
BANDB –Branch if Bitwise And True

BANDB Ra, Rb, label

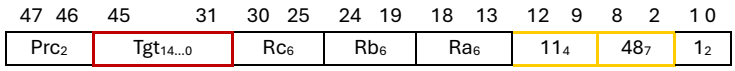
Description:

Branch if the bitwise and of source operands results in a non-zero value. The displacement is relative to the address of the branch instruction.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

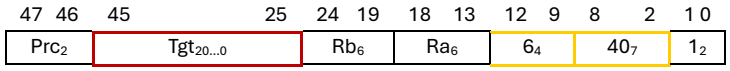
BBC – Branch if Bit Clear

Description:

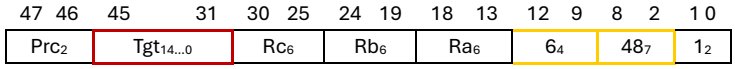
This instruction branches to the target address if bit Rb of Ra is clear, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Register Form

Formats Supported: BR



Register Indirect Target

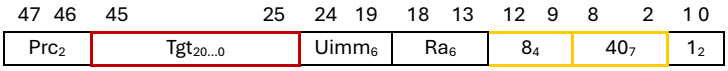


Operation:

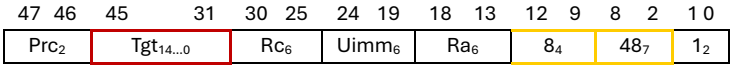
```
If (Ra.bit[Rb] == 0)
        IP = IP + Constant
```

Immediate Form

Formats Supported: BR



Register Indirect Target



Operation:

```
If (Ra.bit[imm] == 0)
        IP = IP + Constant
```

Clock Cycles: 13

Execution Units: Branch

Exceptions: none

Notes:

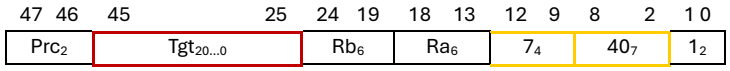
BBS – Branch if Bit Set

Description:

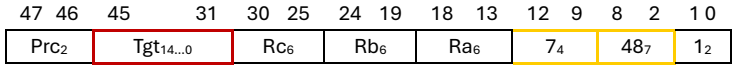
This instruction branches to the target address if bit Rb of Ra is set, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Register Form

Formats Supported: BR



Register Indirect Target

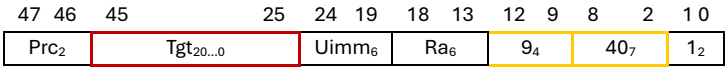


Operation:

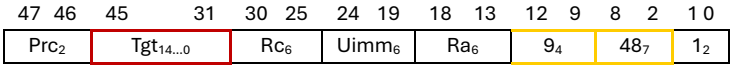
```
If (Ra.bit[Rb] == 1)
        IP = IP + Constant
```

Immediate Form

Formats Supported: BR



Register Indirect Target



Operation:

```
If (Ra.bit[imm] == 1)
        IP = IP + Constant
```

Clock Cycles: 13

Execution Units: Branch

Exceptions: none

Notes:

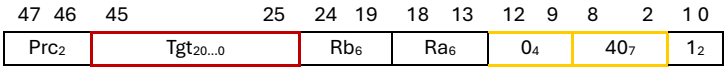
BEQ –Branch if Equal

BEQ Ra, Rb, label

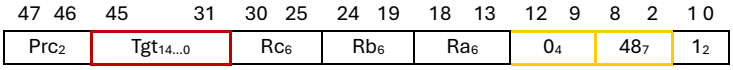
Description:

Branch if source operands are equal. Values are treated as unsigned integers.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

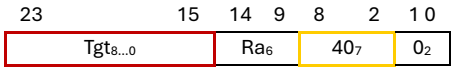
BEQZ –Branch if Equal Zero

BEQZ Ra, label

Description:

Branch if source operand is equal to zero. Values are treated as unsigned integers.

Formats Supported: BR



Clock Cycles: 13

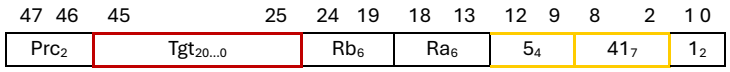
BGE –Branch if Greater than or Equal

BGE Ra, Rb, label

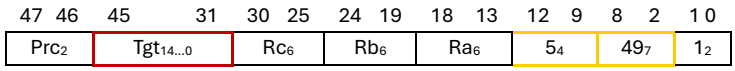
Description:

Branch if source operand Ra is greater than or equal to Rb. Values are treated as signed integers.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

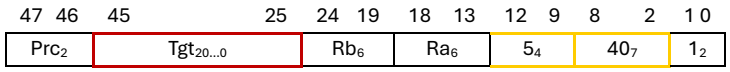
BGEU –Branch if Unsigned Greater than or Equal

BGEU Ra, Rb, label

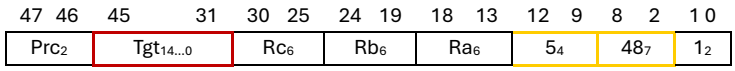
Description:

Branch if source operand Ra is greater than or equal to Rb. Values are treated as unsigned integers.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

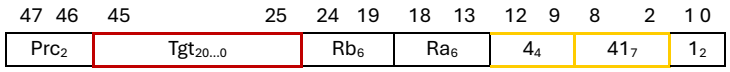
BGT –Branch if Greater Than

BGT Rm, Rn, label

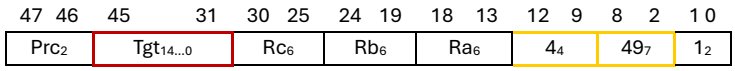
Description:

Branch if the first source operand is greater than the second. Both operands are treated as signed integer values.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

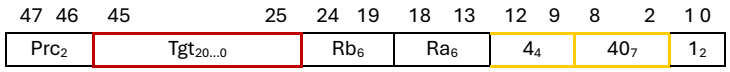
BGTU –Branch if Unsigned Greater Than

BGTU Rm, Rn, label

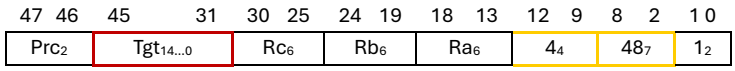
Description:

Branch if the first source operand is greater than the second. Both operands are treated as unsigned integer values.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

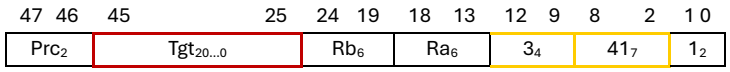
BLE –Branch if Less than or Equal

BLE Ra, Rb, label

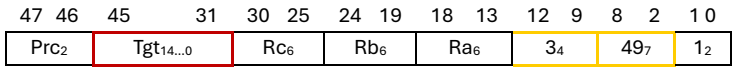
Description:

Branch if source operand Ra is less than or equal to Rb. Values are treated as signed integers.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

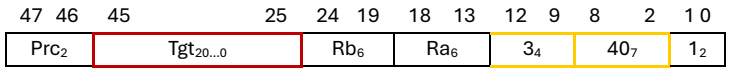
BLEU –Branch if Unsigned Less Than or Equal

BLEU Ra, Rb, label

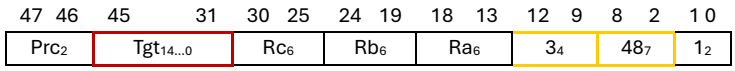
Description:

Branch if the first source operand is less than or equal to the second. Both operands are treated as unsigned integer values.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

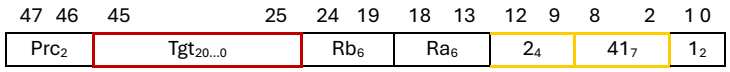
BLT –Branch if Less Than

BLT Ra, Rb, label

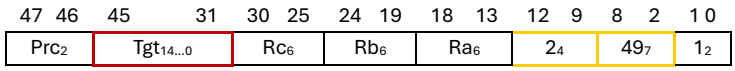
Description:

Branch if the first source operand is less than the second. Both operands are treated as signed integer values.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

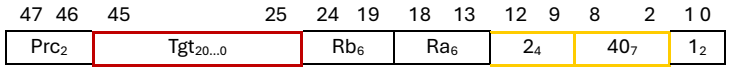
BLTU –Branch if Unsigned Less Than

BLTU Ra, Rb, label

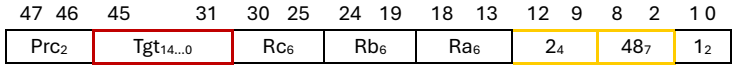
Description:

Branch if the first source operand is less than the second. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

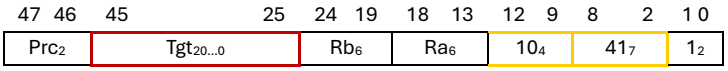
BNAND –Branch if Logical And False

BNAND Ra, Rb, label

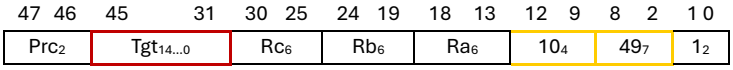
Description:

Branch if the logical ‘and’ of source operands results in a zero value.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

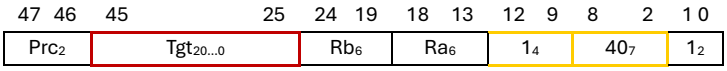
BNE –Branch if Not Equal

BNE Ra, Rb, label

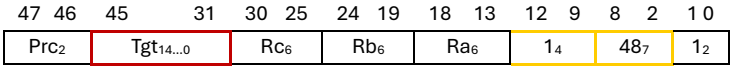
Description:

Branch if source operands are not equal. Values are treated as unsigned integers.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

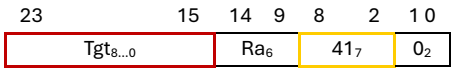
BNEZ –Branch if Not Equal Zero

BNEZ Ra, label

Description:

Branch if source operand is not equal to zero. Values are treated as unsigned integers.

Formats Supported: BR



Clock Cycles: 13

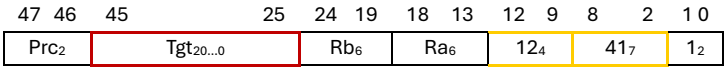
BNOR –Branch if Logical Or False

BNOR Ra, Rb, label

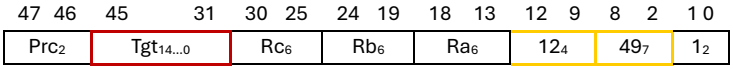
Description:

Branch if the logical ‘or’ of source operands results in a zero value.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

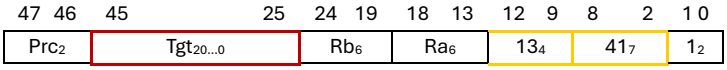
BOR –Branch if Logical Or True

BOR Ra, Rb, label

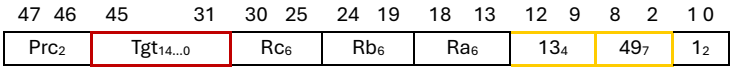
Description:

Branch if the logical or of source operands results in a non-zero value. This ‘or’ operation reduces the source operand values to a Boolean true or false before performing the operation. A non-zero value is considered true, zero is considered false.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

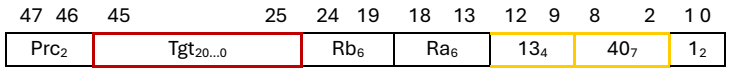
BORB –Branch if Bitwise Or True

BORB Ra, Rb, label

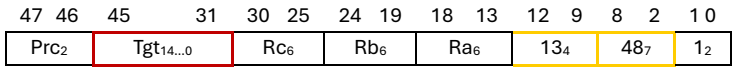
Description:

Branch if the bitwise ‘or’ of source operands results in a non-zero value. The displacement is relative to the address of the branch instruction.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

BRANCH – Branch Always

Description:

This instruction always jumps to the target address using relative addressing. The target address range is $\pm 2^{36}$ bits. This is an alternate mnemonic for the BSR instruction where the link register is r0.

Formats Supported: BSR

Operation:

IP = IP + Constant

Execution Units: Flow Control

Clock Cycles: 13

Exceptions: none

Notes:

BSR – Branch to Subroutine

Description:

This instruction always jumps to the target address using relative addressing. The address of the next instruction is stored in a link register. The target address range is $\pm 2^{36}$ bits.

Formats Supported: BSR

Operation:

Lkt = next IP
IP = IP + Constant

Execution Units: Flow Control

Exceptions: none

Notes:

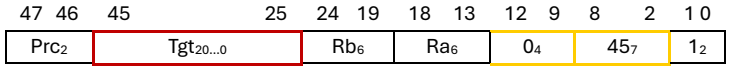
CBEQ –Branch if Capabilities Equal

CBEQ Ra, Rb, label

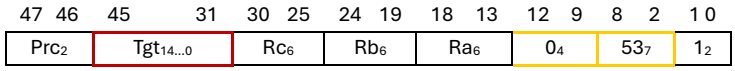
Description:

Branch if capabilities in registers Ra and Rb are exactly identical, including reserved and tag bits. Values are treated as capabilities. For 128-bit capabilities the quad extension prefix must be used. In that case register pairs {Rc,Ra} and {Rd,Rb} must be exactly identical.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

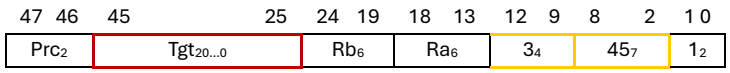
CBLE –Branch if Capability is a Subset or Equal

CBLE Ra, Rb, label

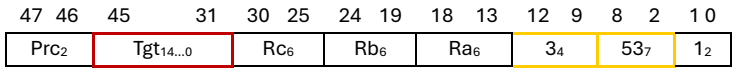
Description:

Branch if capability bounds and permissions in register Ra are a subset of capability bounds and permissions of Rb and the tags are the same OR capabilities in registers Ra and Rb are exactly identical, including reserved and tag bits. Values are treated as capabilities. For 128-bit capabilities the quad extension prefix must be used. In that case register pairs {Rc,Ra} and {Rd,Rb} are tested.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

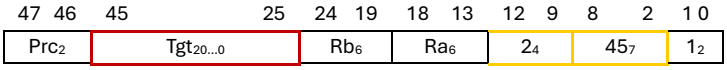
CBLT –Branch if Capability is a Subset

CBLT Ra, Rb, label

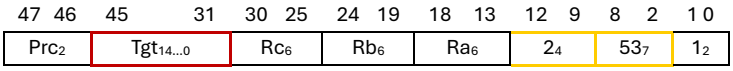
Description:

Branch if capability bounds and permissions in register Ra are a subset of capability bounds and permissions of Rb and the tags are the same. Values are treated as capabilities. For 128-bit capabilities the quad extension prefix must be used. In that case register pairs {Rc,Ra} and {Rd,Rb} are tested.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

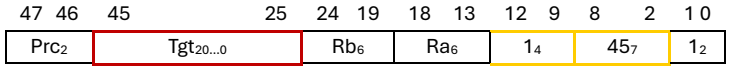
CBNE –Branch if Capabilities Not Equal

CBNE Ra, Rb, label

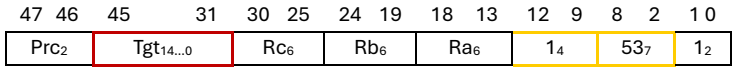
Description:

Branch if capabilities in registers Ra and Rb are not exactly identical, including reserved and tag bits. Values are treated as capabilities. For 128-bit capabilities the quad extension prefix must be used. In that case register pairs {Rc,Ra} and {Rd,Rb} must not be exactly identical.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

CJSR – Jump to Subroutine

Description:

Direct Address Form:

This instruction always jumps to the target address. The PCC of the next instruction is stored in a capability link register Lkt and sealed as a sentry. The target address range is 2^{37} bytes.

Formats Supported: BSR

Operation:

Lkt = next IP, sealed

PCC = Constant

Exceptions:

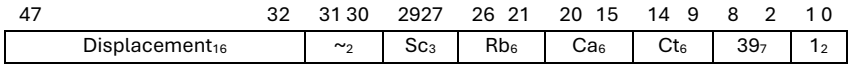
FLT_CAPBOUNDS

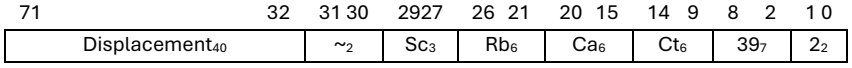
New PCC.address < PCC.base or PCC.address + max_intruction_bytes > PCC.top

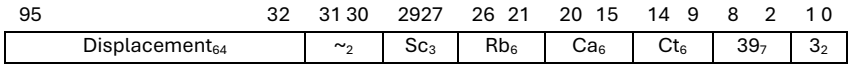
Scaled Indexed Addressing Form:

This instruction always jumps to the target address. The capability IP of the next instruction is stored in capability register Ct, which would normally be a link register, and sealed as a sentry. A constant is added to the sum of the address in capability Ca and scaled index register Rb and loaded into the capability IP.

Instruction Format: d[Ca.address+Rb*Sc]







Operation:

Lkt = next capability IP, sealed

IP.address = Ca.address + (Rb * scale) + Displacement

Exceptions:

FLT_CAPBOUNDS

New IP.address < IP.base or IP.address + max_intruction_bytes > IP.top

FLT_CAPTAG

Ca tag is clear

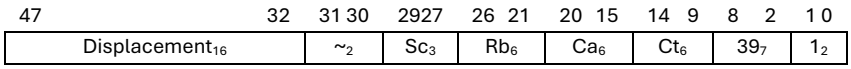
FLT_CAPPERMS

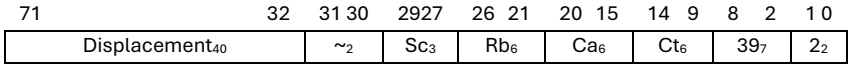
Ca does not grant PERMIT_EXECUTE

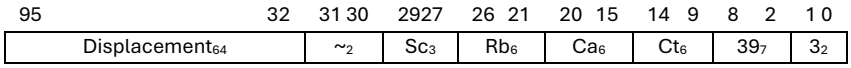
Memory Indirect Scaled Indexed Addressing Form:

This instruction always jumps to the target address. The capability IP of the next instruction is stored in capability register Ct, which would normally be a link register, and sealed as a sentry. The target address is calculated from a value loaded from a table in memory whose base address is contained in register Ca. The value is loaded from memory at the base address plus an index calculated as the scaled contents of register Rb. The table index must be greater than or equal to zero and less than the limit set in the instruction. If the index is greater than or equal to the limit, then the entry at the limit will be jumped to.

Instruction Format: d[Ca.address+Rb*Sc]







Sc_3	Operation
0	Load only the low order 8-bits of the instruction pointer, upper bits remain the same
1	Load only the low order 16-bits of the instruction pointer, upper bits remain the same

2	Load only the low order 32-bits of the instruction pointer, upper bits remain the same
3	Load only the low order 64-bits of the instruction pointer, upper bits remain the same
4	Load entire instruction pointer

Operation:

Lkt = next IP
IP = Memory[Ca.address + Rb * scale]

Execution Units: Branch

Exceptions:

FLT_CAPTAG

Ca tag is clear

FLT_CAPBOUNDS

New IP.address < IP.base or IP.address + max_intruction_bytes > IP.top

Notes:

Low order bits of the instruction pointer may be loaded while keeping the higher order bits constant. This allows efficient implementation of jump tables.

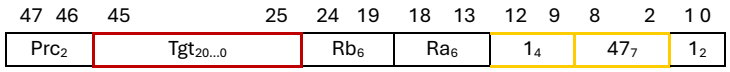
DBNE – Decrement and Branch if Not Equal

DBNE Ra, Rb, label

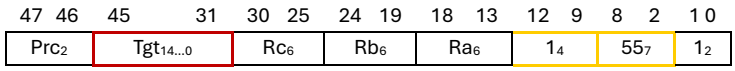
Description:

Branch if source operands are not equal. Values are treated as unsigned integers. Register Ra is decremented.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

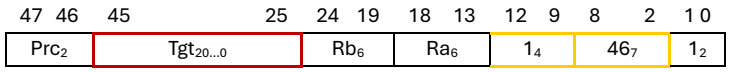
IBNE – Increment and Branch if Not Equal

DBNE Ra, Rb, label

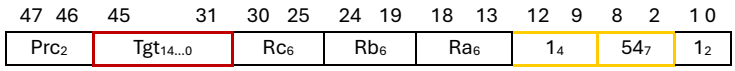
Description:

Branch if source operands are not equal. Values are treated as unsigned integers. Register Ra is decremented.

Formats Supported: BR



Register Indirect Target



Clock Cycles: 13

JMP – Jump to Address

Description:

This instruction always jumps to the target address using absolute addressing. See also BRANCH for position independent code. JMP is rarely used as most software uses relative branching.

Direct Address Form:

This instruction always jumps to the target address. The target address range is 2^{37} bits. If a greater address range is required, the scaled-indexed form of the instruction must be used.

Formats Supported: BSR

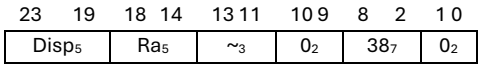
Operation:

IP = Constant

Register Indirect with Displacement Form:

This instruction always jumps to the target address. A displacement constant is added to Ra and loaded into the IP.

Instruction Format: d[Ra]



Scaled Indexed Addressing Form:

This instruction always jumps to the target address. A constant is added to the sum of Ra and scaled index register Rb and loaded into the IP.

Instruction Format: d[Ra+Rb*Sc]

47 32	31 30	29 27	26 21	20 15	14 9	8 2	1 0
$Displacement_{16}$	$\sim_2$	Sc_3	Rb_6	Ra_6	0_6	38_7	1_2

71 32	31 30	29 27	26 21	20 15	14 9	8 2	1 0
$Displacement_{40}$	$\sim_2$	Sc_3	Rb_6	Ra_6	0_6	38_7	2_2

95 32	31 30	29 27	26 21	20 15	14 9	8 2	1 0
$Displacement_{64}$	$\sim_2$	Sc_3	Rb_6	Ra_6	0_6	38_7	3_2

Operation:

Rt = next IP
IP = Ra + (Rb * scale) + displacement

Memory Indirect Form:

Instruction Format: d[Ra+Rb*Sc]

47 32	31 30	29 27	26 21	20 15	14 9	8 2	1 0
$Limit_{16}$	Ar_2	Sc_3	Rb_6	Ra_6	0_6	36_7	1_2

71 32	31 30	29 27	26 21	20 15	14 9	8 2	1 0
$Limit_{40}$	Ar_2	Sc_3	Rb_6	Ra_6	0_6	36_7	2_2

95 32	31 30	29 27	26 21	20 15	14 9	8 2	1 0
$Limit_{64}$	Ar_2	Sc_3	Rb_6	Ra_6	0_6	36_7	3_2

Operation:

```
If (Ar)
        If (Rb * scale > limit OR Rb < 0)
                IP = IP + Memory [Ra + (limit * scale)]
        else
                IP = IP + Memory [Ra + (Rb * scale)]
else
```

```
If (Rb * scale > limit OR Rb < 0)
        IP = Memory [Ra + (limit * scale)]
else
        IP = Memory [Ra + (Rb * scale)]
```

Exceptions: none

AR Abs/rel	Sc_3	Operation
0	0	Load only the low order 8-bits of the instruction pointer, upper bits remain the same
0	1	Load only the low order 16-bits of the instruction pointer, upper bits remain the same
0	2	Load only the low order 32-bits of the instruction pointer, upper bits remain the same
0	3	Load only the low order 64-bits of the instruction pointer, upper bits remain the same
0	4	Load entire instruction pointer
1	0	Add 8-bit value to instruction pointer
1	1	Add 16-bit value to instruction pointer
1	2	Add 32-bit value to instruction pointer
1	3	Add 64-bit value to instruction pointer
1	4	

Execution Units: Branch

Exceptions: none

Notes:

JSR – Jump to Subroutine

Description:

This instruction always jumps to the target address using absolute addressing. The address of the next instruction is stored in a link register. See also BSR for position independent code.

Direct Address Form:

This instruction always jumps to the target address. The address of the next instruction is stored in a link register. The target address range is 2^{37} bits. If a greater address range is required, the scaled-indexed form of the instruction must be used.

Formats Supported: BSR

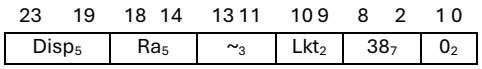
Operation:

Lkt = next IP
IP = Constant* 8

Register Indirect with Displacement Form:

This instruction always jumps to the target address. The IP of the next instruction is stored in link register Lkt. A displacement constant is added to Ra and loaded into the IP.

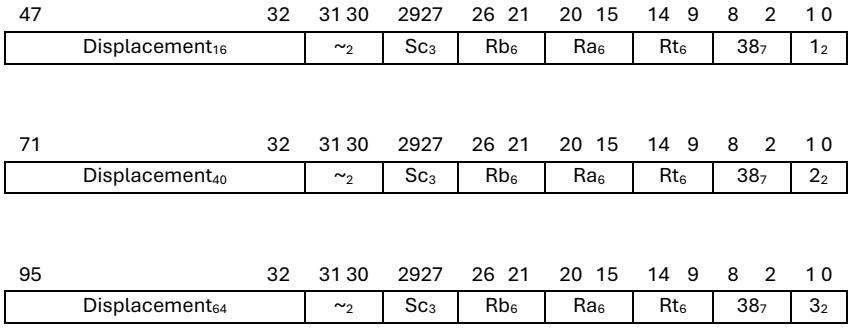
Instruction Format: d[Ra]



Scaled Indexed Addressing Form:

This instruction always jumps to the target address. The IP of the next instruction is stored in register Rt which would normally be a link register. A constant is added to the sum of Ra and scaled index register Rb and loaded into the IP.

Instruction Format: d[Ra+Rb*Sc]



Operation:

Rt = next IP
IP = Ra + (Rb * scale) + displacement

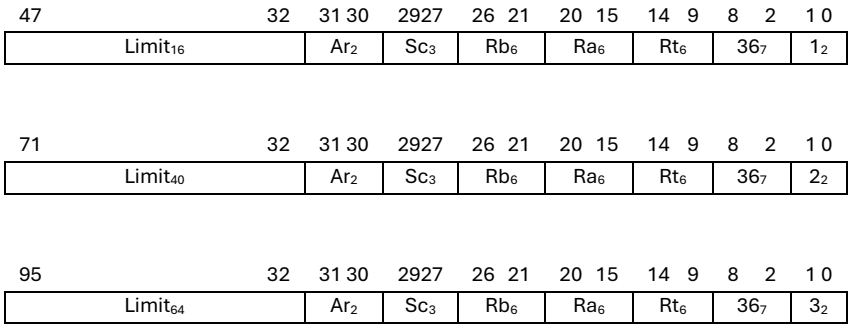
Exceptions: none

Memory Indirect Form:

Description:

This instruction jumps to the target address calculated from a value in a table. The IP of the next instruction is stored in register Rt which would normally be a link register. The target address is calculated from a value loaded from a table in memory whose base address is contained in register Ra. The value is loaded from memory at the base address plus an index calculated as the scaled contents of register Rb. The table index must be greater than or equal to zero and less than the limit set in the instruction. If the index is greater than or equal to the limit, then the entry at the limit will be jumped to.

Instruction Format: d[Ra+Rb*Sc]



Operation:

```
Rt = next IP
If (Ar)
        If (Rb * scale > limit OR Rb < 0)
                IP = IP + Memory [Ra + (limit * scale)]
        else
                IP = IP + Memory [Ra + (Rb * scale)]
else
        If (Rb * scale > limit OR Rb < 0)
                IP = Memory [Ra + (limit * scale)]
        else
                IP = Memory [Ra + (Rb * scale)]
```

Exceptions: none

AR Abs/rel	Sc_3	Operation
0	0	Load only the low order 8-bits of the instruction pointer, upper bits remain the same
0	1	Load only the low order 16-bits of the instruction pointer, upper bits remain the same
0	2	Load only the low order 32-bits of the instruction pointer, upper bits remain the same
0	3	Load only the low order 64-bits of the instruction pointer, upper bits remain the same
0	4	Load entire instruction pointer
1	0	Add 8-bit value to instruction pointer
1	1	Add 16-bit value to instruction pointer

1	2	Add 32-bit value to instruction pointer
1	3	Add 64-bit value to instruction pointer
1	4	

Execution Units: Branch

Exceptions: none

Notes:

NOP – No Operation

NOP

Description:

This instruction does not perform any operation. Any value for bits 9 to 23 may be used.

Instruction Format:

23 9	8 2	1 0
$0x7FFF_{15}$	127_7	0_2

Notes:

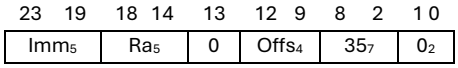
RET – Return from Subroutine and Deallocate

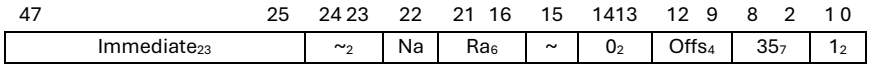
RET Ra, N

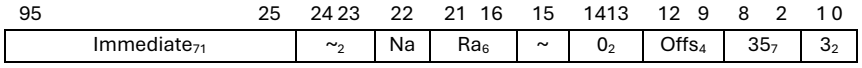
Description:

This instruction returns from a subroutine by transferring program execution to the address stored in a link register specified by Ra plus an offset amount. Additionally, the stack pointer is incremented by the amount specified.

Formats Supported: RET







Operation:

IP <= Ra + Offs * 3, Ra <> 0

SP = SP + Constant

Execution Units: Branch

Exceptions: none

Notes:

Return address prediction hardware may make use of the RTS instruction.

RTE – Return from Exception

Description:

This instruction returns from an exception routine by transferring program execution to the address stored in an internal stack. This instruction may perform a two-up level return.

Formats Supported: RTE

23 19	18 14	13	12 9	8 2	1 0
1_5	0_5	1	$Offs_4$	35_7	0_2

Formats Supported: RTE – Two up level return.

23 19	18 14	13	12 9	8 2	1 0
2_5	0_5	1	$Offs_4$	35_7	0_2

Operation:

Optionally pop the status register and always pop the instruction pointer from the internal stack. Add Offs * 3 bytes to the instruction pointer. If returning from an application trap the status register is not popped from the stack.

Execution Units: Branch

Exceptions: none

Notes:

Capabilities Instructions

Overview

The capabilities instruction set is modelled after the RISC-V capabilities instructions present in the capabilities document. It is very similar but there are some differences. A couple of the test instructions are replaced with a capability compare instruction. The opcodes are different to suit Qupls.

Please refer to: University of Cambridge technical report 987:

Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9) (cam.ac.uk)

Capability Register Representation

Capabilities are represented using a modified CHERI concentrate compression. The CHERI capability can resolve small regions to the byte. The format presented here has a minimum granularity of eight bytes.

Capability Register Format

31 20	19	18 15	14	13 11	10 8	7 3	2 0
P_{12}	f	$Otype_4$	Ie	T[8:6]	Te_3	B[10:6]	Be_3
$Address_{32}$							

Ie=0	Ie=1
E=0 T[2:0] = 0 B[2:0] = 0 T[5:3] = Te_3 B[5:3] = Be_3 Lcarryout = T[8:3] < B[8:3] Lmsb = 0	E= {Te_3, Be_3} T[5:0] = 0 B[5:0] = 0 Lcarryout = T[8:6] < B[8:6] Lmsb = 1

T[10:9] = B[10:9] + Lcarryout + Lmsb

Bounds decoding

Address, a =	Atop = a[63:E+14]	Amid = a[E+13:E]	Alow = a[E-1:0]
Top, t=	Atop + ct	T[13:0]	{E{0}}
Base, b=	Atop + cb	B[13:0]	{E{0}}

Calculating ct and cb

A3 = A[E+13:E+11]
B3 = B[13:11]
T3 = T[13:11]
R = B3 – 1

A3 < R	T3 < R	ct		A3 < R	B3 < R	cb
false	false	0		false	false	0
false	true	+1		false	true	+1
True	False	-1		True	False	-1
True	True	0		True	True	0

Permissions

0	Global	
1	Permit execute	
2	Permit load	
3	Permit store	
4	Permit load capability	
5	Permit store capability	
6	Permit store local capability	
7	Permit seal	
8	Permit invoke	
9	Permit unseal	
10	Permit access system registers	
11	Permit set CID	

64-bit pointer (requires 128-bit register pairs)

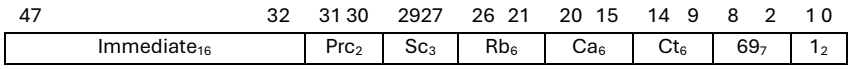
63 48	47	46 45	44 27	26	25 17	16 14	13 3	2 0
P_{16}	f		$Otype_{18}$	Ie	T[11:3]	Te_3	B[13:3]	Be_3
$Address_{64}$								

LDCAP Cn, <ea> - Load Capability

Description:

Load a capability from memory to Ct. If Ca.perms does not grant PERMIT_LOAD_CAPABILITY then Ct.tag is cleared. The quad extension prefix is needed to load 128-bit capabilities.

Instruction Format: d[Ra+Rb*Sc]



Execution Units: AGEN, MEM

Exceptions:

Ca tag not set
Ca is sealed
Ca.perms does not grant PERMIT_LOAD
Ca.address + displacement < Ca.base
Ca.address + displacement + CLEN/8 > Ca.top

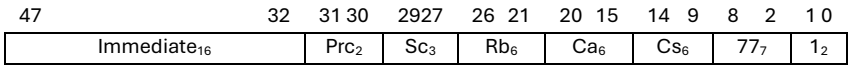
Notes:

STCAP Cn, <ea> - Store Capability

Description:

Store a capability from register Cs to memory. The capability at address Ca.address plus the displacement is replaced with the capability from Ca. The quad extension prefix is needed to store 128-bit capabilities.

Instruction Format: d[Ra+Rb*Sc]



Execution Units: AGEN, MEM

Exceptions:

Ra tag not set
Ra is sealed
Ra.perms does not grant PERMIT_STORE
Ra.perms does not grant PERMIT_STORE_CAPABILTY and Rs.tag is set
Ra.perms does not grant PERMIT_STORE_LOCAL_CAPABILITY and Rs.tag is set and Rs.perms does not grant GLOBAL
Ra.address + displacement < Ra.base
Ra.address + displacement + CLEN/8 > Ra.top

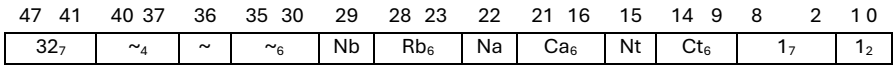
Notes:

CAndPerm

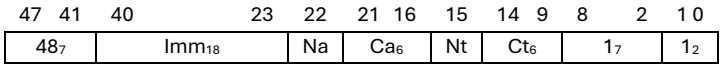
Description:

Capability Ct is replaced with Ca and sealed with the perms field set to the bitwise ‘and’ of its value and the value in register Rb. If Ca is sealed, then the tag field of Ct is cleared.

Instruction Format R3:



Instruction Format CRI:



Execution Units:

Exceptions:

Notes:

CBuildCap

Description:

Capability Ct is replaced with Ca with its base, address, length, perms, uperms and flags replaced with the value of those fields from Cb. If Cb is a sentry, then Ct is sealed as a sentry. If one of the following conditions is true:

- the resulting capability is not a subset of Ca in bounds or permissions, or is not a legally derivable capability,
- Ca does not have its tag field set,
- Ca is sealed

then Ct is replaced with Cb with its tag field clear.
The quad extension prefix is needed to build 128-bit capabilities.

Instruction Format:

47 41	40 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
33_7	$\sim_4$	~	$\sim_6$	~	Cb_6	~	Ca_6	~	Ct_6	1_7	1_2

Execution Units:

Exceptions:

Notes:

CClearTag

Description:

Capability Ct is replaced with Ca, the tag field of Ct is cleared.

Instruction Format:

47 41	40 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
64_7	$\sim_4$	~	$\sim_6$	~	$\sim_6$	~	Ca_6	~	Ct_6	1_7	1_2

Execution Units:

Exceptions:

Notes:

CCmp

Description:

Compares capabilities. This instruction replaces the CTestSubset, CSetEqualsExact, and CGetSealed instructions as outlined for RISC-V in the CHERI document.

If capabilities registers Ca and Cb are exactly identical, including reserved and tag bits, then the register Rt condition equals bit (bit zero) is set.

If capabilities register Ca bounds and permissions are a subset of capabilities Cb and the tags are the same, then the register Rt less than or equals bit is set.

If Ca is unsealed, the overflow bit of Rt is cleared; otherwise, the overflow bit is set.

Instruction Format:

47 41	40 37	36	35 30	29	28 23	22	21 16	15	14 9	8 2	1 0
44_7	$\sim_4$	~	$\sim_6$	~	Cb_6	~	Ca_6	~	Rt_6	1_7	1_2

Execution Units:

Clocks: 1

Exceptions: none

Notes:

Capabilities comparisons are also performed by the compare-and-branch instructions.

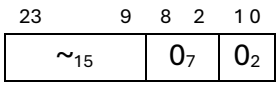
System Instructions

BRK – Break

Description:

This instruction initiates the processor debug routine. The processor enters debug mode. The cause code register is set to indicate execution of a BRK instruction. Interrupts are disabled. The instruction pointer is reset to the vector located from the contents of tvec[3] and instructions begin executing. There should be a jump instruction placed at the break vector location. The address of the BRK instruction is stored in the EIP.

Instruction Format: BRK



Operation:

PUSH SR
PUSH IP
EIP = IP
IP = vector at (tvec[3])

Execution Units: Branch

Clock Cycles:

Exceptions: none

Notes:

Modifiers

ATOM Modifier

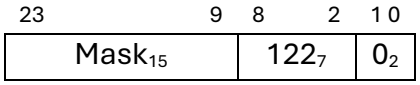
Description:

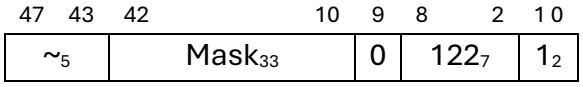
Treat the following sequence of instructions as an “atom”. The instruction sequence is executed with interrupts set to the specified mask level. Interrupts may be disabled for up to eleven instructions. The non-maskable interrupt may not be masked.

The 33-bit mask is broken into eleven three-bit interrupt level numbers. Bit 7 to 9 represent the interrupt level for the first instruction, bits 10 to 12 for the second and so on.

Note that since the processor fetches instructions in groups the mask effectively applies to the group. The mask guarantees that at least as many instructions as specified will be masked, but more may be masked depending on group boundaries.

Instruction Format: ATOM





Modifier Scope	Mask Bit	
	0 to 2	Instruction zero (always 7)
	3 to 5	Instruction one
	6 to 8	Instruction two
	9 to 11	Instruction three
	12 to 14	Instruction four
	15 to 17	Instruction five
	18 to 20	Instruction six
	21 to 23	Instruction seven
	24 to 27	Instruction eight
	28 to 30	Instruction nine
	31 to 33	Instruction ten

Assembler Syntax:

Example:

```
ATOM “777777”
LOAD a0,[a3]
SLT t0,a0,a1
PRED t0,~t0,r0,”AAB”
STORE a2,[a3]
LDI a0,1
LDI a0,0
```

```
ATOM “6666”
LOAD a1,[a3]
ADD t0,a0,a1
MOV a0,a1
STORE t0,[a3]
```

QEXT Prefix

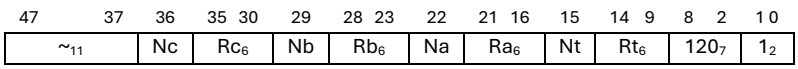
Description:

This prefix extends the register selection for quad precision. Quad precision operations need to use register pairs to contain a quad precision value. The QEXT prefix specifies the registers used to contain bits 64 to 127 of the quad precision values.

Quad precision values are calculated using the QEXT prefix before the quad precision instruction.

Note that any of 64 registers may be selected.

Instruction Format: QEXT



PFX[ABCT] – A/B/C/T Immediate Postfix

PFXA $1234

Description:

This instruction supplies immediate constant bits five to N for the preceding instruction, allowing a N-bit constant to be used in place of a register. The first five bits of the constant are specified by the register number field of the instruction. The A/B/C field of the instruction specifies which register is to be used as a constant.

ABC	Substitute Immediate for:
0	Ra
1	Rb
2	Rc
3	Rt

*Only one postfix is supported per instruction.

Instruction Format:

23 … 11	10 9	8 … 2	1 0
$Immediate_{13...5}$	ABC	124_7	0_2

47 … 11	10 9	8 … 2	1 0
$Immediate_{41...5}$	ABC	124_7	1_2

95 … 11	10 9	8 … 2	1 0
$Immediate_{89...5}$	ABC	124_7	2_2

Notes:

Qupls2 Opcodes

	0	1	2	3	4	5	6	7
0x	0 BRK	1 {CAP}	2 {R3.128}	3 {BFLD}	4 ADDI	5 SUBFI	6 MULI	7 CSR
	8 ANDI	9 ORI	10 EORI	11 CMPI	12	13 DIVI	14 MULUI	15 MOV
1x	16	17 {DFLT}	18 {PST}	19 CMPUI	20	21 DIVUI	22 SEQI	23 SNEI
	24 SLTI	25 SLEI	26 SGTI	27 SGEI	28 SLTUI	29 SLEUI	30 SGTUI	31 SGEUI
2x	32 BRA BSR	33 JMP addr JSR addr	34 CJMP addr CJSR addr	35 RET IRET JMPX	36 JMP ind JSR ind	37 CJMP ind CJSR ind	38	39
	40 BccU	41 Bcc	42 FBcc	43 DFBcc	44 PBcc	45 CBcc	46 IBcc	47 DBcc
3x	48 BccU r	49 Bcc r	50 FBcc r	51 DFBcc r	52 PBcc r	53 CBcc r	54 IBcc r	55 DBcc r
	56	57 {FLT.16}	58 {FLT.32}	59 {FLT.64}	60 ADD2UI	61 ADD4UI	62 ADD8UI	63 ADD16UI
4x	64 LDxU	65 LDx	66 FLDx	67 DFLDx	68 PLDx	69 LDCAP	70 CACHE	71 CHKSC
	72 STX	73 STI	74 FSTx	75 DFSTx	76 PSTx	77 STCAP	78 STPTR	79 STCTX
5x	80 {SHFT.8}	81 {SHFT.16}	82 {SHFT.32}	83 {SHFT.64}	84 ENTER	85 LEAVE	86 PUSH	87 POP
	88 LDA	89 BLEND	90 {FLT.128}	91	92 AMO	93 CAS	94 ZSEQI	95 ZSNEI
6x	96 ZSLTI	97 ZSLEI	98 ZSGTI	99 ZSGEI	100 ZSLTUI	101 ZSLEUI	102 ZSGTUI	103 ZSGEUI
	104 {R3.8}	105 {R3.16}	106 {R3.32}	107 {R3.64}	108 BFIND	109 BCMP	110	111 {BLOCK}
7x	112 CHK	113 STOP	114 FENCE	115 PFI	116	117	118	119
	120 QEXT	121 PRED	122 ATOM	123	124 PFXABC	125	126	127 NOP

{CAP} Map – Opcode 1

	0	1	2	3	4	5	6	7
	8 CRetd	9	10	11	12 CInvoke	13	14	15
	16	17	18	19	20	21	22	23
	24	25	26	27	28	29	30	31
	32 CAndPerm	33 CBuildCap	34 CCopyType	35 CIncOffs	36 CSeal	37 CSetAddr	38 CSetBounds	39 CSetFlags
	40 CSetHigh	41 CSetOffs	42 CSpeicalRW	43 CUnseal	44	45	46	47
	48 CAndPermi	49	50	51	52	53	54	55
	56	57	58	59	60	61	62	63
	64 CClearTag	65 CGetFlags	66 CGetHIgh	67 CGetLen	68 CGetOffs	69 CGetPerms	70 CGetTag	71 CGetTop
	72 CGetType	73 CLoadTags	74 CAlignMsk	75 CRoundLen	76 CSealEntry	77 CGetBase	78	79
	80	81	82	83	84	85	86	87
	88	89	90	91	92	93	94	95
	96	97	98	99	100	101	102	103
	104	105	106	107	108	109	110	111
	112	113	114	115	116	117	118	119
	120	121	122	123	124	125	126	127

{R1} Operations

	0	1	2	3	4	5	6	7
0x	0 CNTLZ	1 CNTLO	2 CNTPOP	3 ABS	4 SQRT	5 REVBIT	6 CNTTZ	7 NOT
	8 NNA_TRIG	9 NNA_STAT	10 NNA_MFACT	11	12 MKBOOL	13 REX	14 SM3P0	15 SM3P1
1x	16	17	18 AES64DS	19 AES64DSM	20 AES64ES	21 AES64ESM	22 AES64IM	23
	24 SHA256 SIG0	25 SHA256 SIG1	26 SHA256 SUM0	27 SHA256 SUM1	28 SHA512 SIG0	29 SHA512 SIG1	30 SHA512 SUM0	31 SHA512 SUM1

{R3} Operations

	0	1	2	3	4	5	6	7
0	0 AND	1 OR	2 EOR	3 CMP	4 ADD	5	6 CMPU	7 CPUID
	8 NAND	9 NOR	10 ENOR	11 CMOVNZ	12 CMOVNZ	13 ABS	14 MAJ	15
1	16 MUL	17 DIV	18 {MINMAX}	19 MULU	20 DIVU	21 MULSU	22 DIVSU	23 {MINMAXU}
	24 MULW	25 MOD	26 {R1}	27 MULUW	28 MODU	29 MULSUW	30 MODSU	31
2	32 PTRDIF	33 MUX	34 BMM	35 BMAP	36 DIF	37 CHARNDX	38 CHARNDX	39 CHARNDX
	40 NNA MTWT	41 NNA MTIN	42 NNA MTBIAS	43 NNA MTFB	44 NNA MTMC	45 NNA MTBC	46	47
3	48 V2BITS	49 BITS2V	50 VEX	51 VEINS	52 VGNDX	53	54 VSHLV	55 VSHRV
	56	57	58 VSETMASK	59	60	61	62 VSHLVI	63 VSHRVI
4	64 AES64K1I	65 AES64KS2	66 SM4ED	67 SM4KS	68	69	70 CLMUL	71
	72	73	74	75	76	77	78	79
5	80 SEQ	81 SNE	82 SLT	83 SLE	84 SLTU	85 SLEU	86	87 DIVMOD
	88	89	90	91	92	93	94	95 DIVMODU
6	96 SEQ	97 SNE	98 SLT	99 SLE	100 SLTU	101 SLEU	102	103
	104	105	106	107	108	109	110	111
7	112 ZSEQ	113 ZSNE	114 ZSLT	115 ZSLE	116 ZSLTU	117 ZSLEU	118	119
	120 ZSEQ	121 ZSNE	122 ZSLT	123 ZSLE	124 ZSLTU	125 ZSLEU	126	127 MVVR

{Shift} Operations

	0	1	2	3	4	5	6	7
1	0 SLLP	1 SRLP	2 SRAP	3 SRAPRU	4 SRAPRZ	5 DEP	6 DEPXOR	7
	8 SLLPI	9 SRLPI	10 SRAPI	11 SRAPRUI	12 SRAPRZI	13 DEPI	14 DEPXOR	15

{FLT} Operations

	0	1	2	3	4	5	6	7
16	0 FNOP	1 **{FMxx}**	2 FMIN	3 FMAX	4 FADD	5 FSUB	6 FMUL	7 FDIV
	8 FSEQ	9 FSNE	10 FSLT	11 FSLE	12	13 FCMP	14 FNXT	15 FREM
	16 FSGNJ	17 FSGNJN	18 FSGNJX	19	20 FSCALEB	21	22	23
	24	25	26	27	28	29	30	31
16	32 FABS	33 FNEG	34 FTOI	35 ITOF	36 FCONST	37	38 FSIGN	39 FSIG
	40 FSQRT	41 FCVTS2D	42 FCVTS2Q	43 FCVTD2Q	44 FCVTH2S	45 FCVTH2D	46 ISNAN	47 FINITE
	48 FCVTQ2H	49 FCVTQ2S	50 FCVTQ2D	51	52 FCVTH2Q	53 FTRUNC	54 FRSQRTE	55 FRES
	56	57 FCVTD2S	58	59	60	61	62 FCLASS	63
16	64 FSIN	65 FCOS	66 FTAN	67	68	69	70	71
	72	73	74 FATAN	75	76	77	78	79
	80 FSIGMOID	81	82	83	84	85	86	87
	88	89	90	91	92	93	94	95

{DFLT3} Operations

	0	1	2	3	4	5	6	7
17	0 FMA	1 FMS	2 FNMA	3 FNMS	4 VFMA	5 VFMS	6 VFNMA	7 VFNMS
	8 **{DFLT2}**	9	10	11	12 **{VDFLT2}**	13 **{VSFLT2}**	14	15

{FLT2} Operations

	0	1	2	3	4	5	6	7
17	0 DFSCALEB	1 **{DFLT1}**	2 DFMIN	3 DFMAX	4 DFADD	5 DFSUB	6 DFMUL	7 DFDIV
	8 DFSEQ	9 DFSNE	10 DFSLT	11 DFSLE	12	13 DFCMP	14 DFNXT	15 DFREM
	16 DFSGNJ	17 DFSGNJN	18 DFSGNJX	19	20	21	22	23
	24	25	26	27	28	29	30 FNMUL	31

{AMO} – Atomic Memory Ops

	0	1	2	3	4	5	6	7
92	0 AMOADD	1 AMOAND	2 AMOOR	3 AMOEOR	4 AMOMIN	5 AMOMAX	6 AMOSWAP	7
	8 AMOASL	9 AMOLSR	10 AMOROL	11 AMOROR	12 AMOMINU	13 AMOMAXU	14	15

{EX} Exception Instructions

	0	1	2	3	4	5	6	7
2	0 IRQ	1	2 FTX	3 FCX	4 FDX	5 FEX	6	7 REX
	8	9	10	11	12	13	14	15

MPU Hardware

QIC – Qupls Interrupt Controller

Overview

The Qupls system uses message-signaled interrupts (QMSI). QIC snoops the response bus going to the CPU core(s) for interrupt responses. Interrupt responses are stored in priority queues in the controller.

The Qupls interrupt controller presents an interrupt signal bus to the CPU core(s). The QIC may be used in a multi-CPU system as a shared interrupt controller. The QIC can guide the interrupt to the specified core(s). The QIC is a 64-bit slave I/O device.

System Usage

For the demo system there is just a single interrupt controller in the system. However, there may be up to 63 interrupt controllers in a system, numbered 1 to 63. Each interrupt controller may support up to 63 CPU cores, making the total number of CPU cores processing interrupts approximately 3900. QIC supports 63 different priority levels.

The QIC registers are located at an address determined by BAR0 in the configuration space. The interrupt table is located at a address determined by BAR1.

Priority Resolution

Interrupts have a fixed priority relationship with priority 63 having the highest priority and priority 1 the lowest. As interrupt messages are detected, they are placed in a queue according to their priority. (There are 63 small queues). The QIC sends the highest priority interrupt in the queues to the CPU. Periodically, once every 64 clock cycles, interrupt priorities are inverted.

Config Space

A 256-byte config space is supported. Most of the config space is unused. The only configuration is for the I/O address of the register set.

Regno	Width	R/W	Moniker	Description		
000	32	RO	REG_ID	Vendor and device ID		
004	32	R/W				
008	32	RO				

00C	32	R/W				
010	32	R/W	REG_BAR0	Base Address Register		
014	32	R/W	REG_BAR1	Base Address Register		
018	32	R/W	REG_BAR2	Base Address Register		
01C	32	R/W	REG_BAR3	Base Address Register		
020	32	R/W	REG_BAR4	Base Address Register		
024	32	R/W	REG_BAR5	Base Address Register		
028	32	R/W				
02C	32	RO		Subsystem ID		
030	32	R/W		Expansion ROM address		
034	32	RO				
038	32	R/W		Reserved		
03C	32	R/W		Interrupt		
040 to 0FF	32	R/W		Capabilities area		

REG_BAR0 defaults to $FEE20001 which is used to specify the address of the controller's registers in the I/O address space.

The controller will respond with a memory size request of 0MB (0xFFFFFFFF) when BAR0 is written with all ones. The controller contains its own dedicated memory and does not require memory allocated from the system.

Parameters

CFG_BUS defaults to zero
CFG_DEVICE defaults to six
CFG_FUNC defaults to zero
Config parameters must be set correctly. CFG device and vendors default to zero.

Registers

The QIC contains an interrupt vector table with a maximum of 2048 128-bit vectors available for each of four operating modes. (The number of vectors supported is parameterized). This vector table occupies 128kB of I/O space. An additional 522 registers are spread out through another 8k byte I/O region. All registers are 64-bit and only 64-bit accessible. The interrupt vector table is byte accessible.

Regno	Access	Moniker	Purpose
00	RW	UVTB	Base address for user interrupt vector table

<table>
<tr><td>08</td><td>RW</td><td>SVTB</td><td>Base address for supervisor interrupt vector table</td></tr>
<tr><td>10</td><td>RW</td><td>HVTB</td><td>Base address for hypervisor interrupt vector table</td></tr>
<tr><td>18</td><td>RW</td><td>MVTB</td><td>Base address for hypervisor machine vector table</td></tr>
<tr><td>20</td><td>RW</td><td>VTL</td><td>Vector table limit</td></tr>
<tr><td>28</td><td>RW</td><td>STAT</td><td><table><tr><td>Bit</td><td></td></tr><tr><td>0</td><td>Que full, set if any que is full, cleared by software if written with a zero</td></tr><tr><td>1</td><td>Set if stuck interrupt detected</td></tr><tr><td>2 to 62</td><td>reserved</td></tr><tr><td>63</td><td>Set if an interrupt is being requested</td></tr></table></td></tr>
<tr><td>30</td><td>R</td><td>QUEL</td><td>Output of the priority queues, bits 0 to 63</td></tr>
<tr><td>38</td><td>R</td><td>QUEH</td><td>Output of the priority queues, bits 64 to 127</td></tr>
<tr><td>40</td><td>R</td><td>EMP</td><td>Queue empty status, one bit for each queue, 1=empty</td></tr>
<tr><td>48</td><td>R</td><td>OVR</td><td>Queue overflow status, one bit for each queue, 1=overflowed</td></tr>
<tr><td>380</td><td>RW</td><td>GE</td><td>Bit 0 = global interrupt enable</td></tr>
<tr><td>390</td><td>RW</td><td>THRES</td><td>Interrupt threshold (0 to 63), IRQ priority must exceed this to be recognized.</td></tr>
<tr><td colspan="4">CPU affinity group table follows
There are 256 groups that may be set. The interrupt vector references one of these groups to determine which CPU cores should be notified of an interrupt.</td></tr>
<tr><td>800</td><td>RW</td><td>AFNx</td><td>CPU group, one bit for each CPU that should be notified</td></tr>
<tr><td>...</td><td>RW</td><td></td><td>More CPU groups</td></tr>
<tr><td>FF8</td><td>RW</td><td></td><td>Last CPU group</td></tr>
<tr><td colspan="4">Interrupt pending and enable tables follow. There are 128 64-bit entries for each table. This is enough to cover up to 2047 interrupts for each of four operating modes. User mode is entries 0 to 31, supervisor mode is entries 32 to 63, hypervisor 64 to 95 and machine 96 to 127.</td></tr>
<tr><td>1000</td><td>RW</td><td>IP</td><td>Interrupt enable bits</td></tr>
<tr><td>...</td><td></td><td></td><td>More IE bit registers</td></tr>
<tr><td>13F8</td><td>RW</td><td>IP</td><td></td></tr>
<tr><td>1400</td><td>RW</td><td>IE</td><td>Interrupt pending bits</td></tr>
<tr><td>...</td><td></td><td></td><td>More interrupt pending bits</td></tr>
<tr><td>17F8</td><td>RW</td><td>IE</td><td></td></tr>
</table>

Base Address Fields

The base address fields default to zero. The address fields are present should the controller be adapted to use main memory instead of dedicated BRAM. The address fields act as an index into the dedicated vector table for the location of the vectors for each operating mode.

CPU Affinity Group Table

This table is an array of groups of CPU cores that should be notified of an interrupt. The interrupt vector selects one of these groups for the group of CPUs to notify. Note that normally only a single CPU core will ultimately be selected to process the interrupt. If bit zero of the CPU group is set, then the interrupt will be broadcast to all CPU cores in the group.

Interrupt Enable Bits

The interrupt enable bit array offers a fast way to enable or disable interrupts without having to update the interrupt vector table. Both the enable bit in the enable bit array and the enable bit in the vector table must be set for an interrupt to be enabled.

Interrupt Pending Bits

Writing a pending bit register clears the bit specified by the write data. If the MSB of the value written is a 1 then the corresponding interrupt is immediately triggered.

Interrupt Vector Table

The interrupt vector table has a default address of $FF…FECC0000 to $FF…FECDFFFF. This address may be changed by altering the BAR1 register in the config space. The interrupt vector table has four consecutive sections to it, one for each CPU operating mode. There are 2048 vectors available for each mode. The vector format is as follows:

127 120	119 112	111 104	103 101	100 98	97	96	95 0
~$_8$	Data$_8$	CPU group$_8$	~$_3$	Swstk$_3$	IE	AI	Address$_{64}$ or Instruction$_{96}$

Field Description

AI: This field indicates that the vector contains an address (0) or an instruction (1)

IE: This field indicates if the interrupt is disabled (0) or enabled (1)

Swstk: This field contains the index of the software stack required to process the interrupt

CPU group: This field is an index into the CPU affinity group table which identifies which processor cores are candidates to receive the interrupt.

Data: This field is populated with data from the interrupt message.

QIT – Qupls Interval Timer

Overview

Many systems have at least one timer. The timing device may be built into the cpu, but it is frequently a separate component on its own. The programmable interval timer has many potential uses in the system. It can perform several different timing operations including pulse and waveform generation, along with measurements. While it is possible to manage timing events strictly through software it is quite challenging to perform in that manner. A hardware timer comes into play for the difficult to manage timing events. A hardware timer can supply precise timing. In the test system there are two groups of four timers. Timers are often grouped together in a single component. The QIT is a 64-bit peripheral. The QIT while powerful turns out to be one of the simpler peripherals in the system.

System Usage

One programmable timer component, which may include up 32 timers, is used to generate the system time slice interrupt and timing controls for system garbage collection. The second timer component is used to aid the paged memory management unit. There are free timing channels on the second timer component.

Each QIT is given an 8kB-byte memory range to respond to for I/O access. As is typical for I/O devices part of the address range is not decoded to conserve hardware.

PIT#1 is located at $FFFFFFFFFFEE40000 to $FFFFFFFFFFEE41FFF

PIT#2 is located at $FFFFFFFFFFEE50000 to $FFFFFFFFFFEE51FFF

Config Space

A 256-byte config space is supported. Most of the config space is unused. The only configuration is for the I/O address of the register set and the interrupt line used.

Regno	Width	R/W	Moniker	Description		
000	32	RO	REG_ID	Vendor and device ID		
004	32	R/W				
008	32	RO				
00C	32	R/W				
010	32	R/W	REG_BAR0	Base Address Register		
014	32	R/W	REG_BAR1	Base Address Register		
018	32	R/W	REG_BAR2	Base Address Register		

01C	32	R/W	REG_BAR3	Base Address Register		
020	32	R/W	REG_BAR4	Base Address Register		
024	32	R/W	REG_BAR5	Base Address Register		
028	32	R/W				
02C	32	RO		Subsystem ID		
030	32	R/W		Expansion ROM address		
034	32	RO				
038	32	R/W		Reserved		
03C	32	R/W		Interrupt		
040 to 0FF	32	R/W		Capabilities area		

REG_BAR0 defaults to $FEE40001 which is used to specify the address of the controller's registers in the I/O address space. Note for additional groups of timers the REG_BAR0 must be changed to point to a different I/O address range. Note the core uses only bits determined by the address mask in the address range comparison. It is assumed that the I/O address select input, cs_io, will have bits 24 and above in its decode and that a 8kB page is required for the device, matching the MMU page size.

The controller will respond with a mask of 0xFFFFFFFF when BAR0 is written with all ones.

Parameters

CFG_BUS defaults to zero
CFG_DEVICE defaults to four
CFG_FUNC defaults to zero
CFG_ADDR_MASK defaults to 0x00FF0000
CFG_IRQ_LINE defaults to 29
Config parameters must be set correctly. CFG device and vendors default to zero.

Parameters

NTIMER: This parameter controls the number of timers present. The default is eight. The maximum is 32.

BITS: This parameter controls the number of bits in the counters. The default is 48 bits. The maximum is 64.

PIT_ADDR: This parameter sets the I/O address that the PIT responds to. The default is $FEE40001.

PIT_ADDR_ALLOC: This parameter determines which bits of the address are significant during decoding. The default is $00FF0000 for an allocation of 64kB. To compute the address range allocation required, ‘or’ the value from the register with $FF000000, complement it then add 1.

Registers

The PIT has 134 registers addressed as 64-bit I/O cells. It occupies 2048 consecutive I/O locations. All registers are read-write except for the current counts which are read-only. All registers all 64-bit accessible; all 64 bits must be read or written. Values written to registers do not take effect until the synchronization register is written.

Note the core may be configured to implement fewer timers in which case timers that are not implemented will read as zero and ignore writes. The core may also be configured to support fewer bits per count register in which case the unimplemented bits will read as zero and ignore writes.

Regno	Access	Moniker	Purpose
00	R	CC0	Current Count
08	RW	MC0	Max count
10	RW	OT0	On Time
18	RW	CTRL0	Control
20 to 7F8	...	...	Groups of four registers for timer #1 to #63
800	RW	USTAT	Underflow status
808	RZW	SYNC	Synchronization register
810	RW	IE	Interrupt enable
818	RW	TMP	Temporary register
820	RO	OSTAT	Output status
828	RW	GATE	Gate register
830	RZW	GATEON	Gate on register
838	RZW	GATEOFF	Gate off register

Control Register

This register contains bits controlling the overall operation of the timer.

Bit		Purpose
0	LD	setting this bit will load max count into current count, this bit automatically resets to zero.
1	CE	count enable, if 1 counting will be enabled, if 0 counting is disabled and the current count register holds its value. On counter underflow this bit will be reset to zero causing the count to halt unless auto-reload is set.
2	AR	auto-reload, if 1 the max count will automatically be reloaded into the current count register when it underflows.
3	XC	external clock, if 1 the counter is clocked by an external clock source. The external clock source must be of lower frequency than the clock supplied to the PIT. The PIT contains edge detectors on the external clock source and counting occurs on the detection of a positive edge on the clock source. This bit is forced to 0 for timers 4 to 31.
4	GE	gating enable, if 1 an external gate signal will also be required to be active high for the counter to count, otherwise if 0 the external gate is ignored. Gating the counter using the external gate may allow pulse-width measurement. This bit is forced to 0 for timers 4 to 31.
5 to 63	~	not used, reserved

Current Count

This register reflects the current count value for the timer. The value in this register will change by counting downwards whenever a count signal is active. The current count may be automatically reloaded at underflow if the auto reload bit (bit #2) of the control byte is set. The current count may also be force loaded to the max count by setting the load bit (bit #0) of the counter control byte.

Max Count

This register holds onto the maximum count for the timer. It is loaded by software and otherwise does not change. When the counter underflows the current count may be automatically reloaded from the max count register.

On Time

The on-time register determines the output pulse width of the timer. The timer output is low until the on-time value is reached, at which point the timer output switches high. The timer output remains high until the counter reaches zero at which point the timer output is reset back to zero. So, the on time reflects the length

of time the timer output is high. The timer output is low for max count minus the on-time clock cycles.

Underflow Status

The underflow status register contains a record of which timers underflowed.

Writing the underflow register clears the underflows and disable further interrupts where bits are set in the incoming data. Interrupt processing should read the underflow register to determine which timers underflowed, then write back the value to the underflow register.

Synchronization Register

The synchronization register allows all the timers to be updated simultaneously. Values written to timer registers do not take effect until the synchronization register is written. The synchronization register must be written with a '1' bit in the bit position corresponding to the timer to update. For instance, writing all one's to the sync register will cause all timers to be updated. The synchronization register is write-only and reads as zero.

Interrupt Enable Register

Each bit of the interrupt enable register enables the interrupt for the corresponding timer. Interrupts must also be globally enabled by the interrupt enable bit in the config space for interrupts to occur. A '1' bit enables the interrupt, a '0' bit value disables it.

Temporary Register

This is merely a register that may be used to hold values temporarily.

Output Status

The output status register reflects the current status of the timers output (high or low). This register is read-only.

Gate Register

The internal gate register is used to temporarily halt or resume counting for the timer corresponding to the bit position of this register. Writing a value to this register will turn on all timers where there is a '1' bit in the value and turn off all timers where there is a '0' bit in the value.

Gate On Register

The internal gate 'on' register is used to resume counting for the timer corresponding to the bit position of this register. Writing a value to this register will turn on all timers

where there is a ‘1’ bit in the value. Where there is a ‘0’ in the value the timer will not be affected. This register reads as zero.

Gate Off Register

The internal gate ‘off’ register is used to halt counting for the timer corresponding to the bit position of this register. Writing a value to this register will turn off all timers where there is a ‘1’ bit in the value. Where there is a ‘0’ in the value the timer will not be affected. This register reads as zero.

Programming

The PIT is a memory mapped i/o device. The PIT is programmed using 64-bit load and store instructions (LDO and STO). Byte loads and stores (LDB, STB) may be used for control register access. It must reside in the non-cached address space of the system.

Interrupts

The core is configured use interrupt signal #29 by default. This may be changed with the CFG_IRQ_LINE parameter. Interrupts may be globally disabled by writing the interrupt disable bit in the config space with a ‘1’. Individual interrupts may be enabled or disabled by the setting of the interrupt enable register in the I/O space.

Glossary

ABI

An acronym for application binary interface. An ABI is a description of the interface between software and hardware, or between software modules. It includes things like the expected register usage by the compiler. Some registers hardware has specific requirements for are noted in the ABI, for instance r0 may always be zero or it may be a usable register. The stack pointer may need to be a specific register. A good ABI is an aid to guaranteeing that software works when coming from multiple sources.

AMO

AMO stands for atomic memory operation. An atomic memory operation typically reads then writes to memory in a fashion that may not be interrupted by another processor. Some examples of AMO operations are swap, add, and, and or. AMO operations are typically passed from the CPU to the memory controller and the memory controller performs the operation.

Assembler

A program that translates mnemonics and operands into machine code OR a low-level language used by programmers to conveniently translate programs into machine code. Compilers are often capable of generating assembler code as an output.

ATC

ATC stands for address translation cache. This buffer is used to cache address translations for fast memory access in a system with an mmu capable of performing address translations. The address translation cache is more commonly known as the TLB.

Base Pointer

An alternate term for frame pointer. The frame or base pointer is used by high-level languages to access variables on the stack.

Burst Access

A burst access is several bus accesses that occur rapidly in a row in a known sequence. If hardware supports burst access the cycle time for access to the

device is drastically reduced. For instance, dynamic RAM memory access is fast for sequential burst access, and somewhat slower for random access.

BTB

An acronym for Branch Target Buffer. The branch target buffer is used to improve the performance of a processing core. The BTB is a table that stores the branch target from previously executed branch instructions. A typical table may contain 1024 entries. The table is typically indexed by part of the branch address. Since the target address of a branch type instruction may not be known at fetch time, the address is speculated to be the address in the branch target buffer. This allows the machine to fetch instructions in a continuous fashion without pipeline bubbles. In many cases the calculated branch address from a previously executed instruction remains the same the next time the same instruction is executed. If the address from the BTB turns out to be incorrect, then the machine will have to flush the instruction queue or pipeline and begin fetching instructions from the correct address.

Card Memory

A card memory is a memory reserved to record the location of pointer stores in a garbage collection system. The card memory is much smaller than main memory; there may be card memory entry for a block of main memory addresses. Card memory covers memory in 128 to 512-byte sized blocks. Usually, a byte is dedicated to record the pointer store status even though a bit would be adequate, for performance reasons. The location of card memory to update is found by shifting the pointer value to the right some number of bits (7 to 9 bits) and then adding the base address of the table. The update to the card memory needs to be done with interrupts disabled.

Commit

As in commit stage of processor. This is the stage where the processor is dedicated or committed to performing the operation. There are no prior outstanding exceptions or flow control changes to prevent the instruction from executing. The instruction may execute in the commit stage, but registers and memory are not updated until the retire stage of the processor.

Decimal Floating Point

Floating point numbers encoded specially to allow processing as decimal numbers. Decimal floating point allows processing every-day decimal numbers rounding in the same manner as would be done by hand.

Decode

The stage in a processor where instructions are decoded or broken up into simpler control signals. For instance, there is often a register file write signal that must be decoded from instructions that update the register file.

Diadic

As in diadic instruction. An instruction with two operands.

Endian

Computing machines are often referred to as big endian or little endian. The endian of the machine has to do with the order bits and bytes are labeled. Little endian machines label bits from right to left with the lowest bit at the right. Big endian machines label bits from left to right with the lowest numbered bit at the left.

FIFO

An acronym standing for 'first-in first-out'. Fifo memories are used to aid data transfer when the rate of data exchange may have momentary differences. Usually when fifos transfer data the average data rate for input and output is the same. Data is stored in a buffer in order then retrieved from the buffer in order. Uarts often contain fifos.

FPGA

An acronym for Field Programmable Gate Array. FPGA's consist of a large number of small RAM tables, flip-flops, and other logic. These are all connected with a programmable connection network. FPGA's are 'in the field' programmable, and usually re-programmable. An FPGA's re-programmability is typically RAM based. They are often used with configuration PROM's so they may be loaded to perform specific functions.

Floating Point

A means of encoding numbers into binary code to allow processing. Floating point numbers have a range within which numbers may be processed, outside of this range the number will be marked as infinity or zero. The range is usually large enough that it is not a concern for most programs.

Frame Pointer

A pointer to the current working area on the stack for a function. Local variables and parameters may be accessed relative to the frame pointer. As a program progresses a series of "frames" may build up on the stack. In many cases the frame pointer may be omitted, and the stack pointer used for references instead. Often a register from the general register file is used as a frame pointer.

HDL

An acronym that stands for 'Hardware Description Language'. A hardware description language is used to describe hardware constructs at a high level.

HLL

An acronym that stands for "High Level Language"

Instruction Bundle

A group of instructions. It is sometimes required to group instructions together into bundle. For instance, all instructions in a bundle may be executed simultaneously on a processor as a unit. Instructions may also need to be grouped if they are oddball in size for example 41 bits, so that they can be fit evenly into memory. Typically, a bundle has some bits that are global to the bundle, such as template bits, in addition to the encoded instructions.

Instruction Pointers

A processor register dedicated to addressing instructions in memory. It is also often called a program counter. The program counter got its name because it usually increments (or counts) automatically after an instruction is fetched. In early machines in some rare cases the program counter did not count in a sequential binary fashion, but instead used other forms of a counter such as a grey counter or linear feedback shift register. In some

machines the program counter addresses bundles of instructions rather than individual instructions. This is common with some stack machines where multiple instructions are packed into a memory word.

Instruction Prefix

An instruction prefix applies to the following instruction to modify its operation. An instruction prefix may be used to add more bits to a following immediate constant, or to add additional register fields for the instruction. The prefix essentially extends the number of bits available to encode instructions. An instruction prefix usually locks out interrupts between the prefix and following instruction.

Instruction Modifier

An instruction modifier is similar to an instruction prefix except that the modifier may apply to multiple following instructions.

ISA

An acronym for Instruction Set Architecture. The group of instructions that an architecture supports. ISA's are sometimes categorized at extreme edges as RISC or CISC. RTF64 falls somewhere in between with features of both RISC and CISC architectures.

IPI

An acronym for Inter-Processor-Interrupt. An inter-processor interrupt is an interrupt sent from one processor to another.

JIT

An acronym standing for Just-In-Time. JIT compilers typically compile segments of a program just before usage, and hence are called JIT compilers.

Keyed Memory

A memory system that has a key associated with each page to protect access to the page. A process must have a matching key in its key list in order to access the memory page. The key is often 20 bits or larger. Keys for pages are usually cached in the processor for performance reasons. The key may be part of the paging tables.

Linear Address

A linear address is the resulting address from a virtual address after segmentation has been applied.

Machine Code

A code that the processing machine is able execute. Machine code is lowest form of code used for processing and is not usually delt with by programmers except in debugging cases. While it is possible to assemble machine code by hand usually a tool called an assembler is used for this purpose.

Milli-code

A short sequence of code that may be used to emulate a higher-level instruction. For instance, a garbage collection write barrier might be written as milli-code. Milli-code may use an alternate link register to return to obtain better performance.

Monadic

An instruction with just a single operand.

MSI

An acronym for Message Signaled Interrupt. A message signaled interrupt is an interrupt processed using a message sent to a CPU using in-band resources.

Opcode

A short form for operation code, a code that determines what operation the processor is going to perform. Instructions are typically made up of opcodes and operands.

Operand

The data that an opcode operates on, or the result produced by the operation. Operands are often located in registers. Inputs to an operation are referred to as source operands, the result of an operation is a destination operand.

Physical Address

A physical address is the final address seen by the memory system after both segmentation and paging have been applied to a virtual address. One can think of a physical address as one that is “physically” wired to the memory.

Physical Memory Attributes (PMA)

Memory usually has several characteristics associated with it. In the memory system there may be several different types of memory, rom, static ram, dynamic ram, eeprom, memory mapped I/O devices, and others. Each type of memory device is likely to have different characteristics. These characteristics are called the physical memory attributes. Physical memory attributes are associated with address ranges that the memory is located in. There may be a hardware unit dedicated to verifying software is adhering to the attributes associated with the memory range. The hardware unit is called a physical memory attributes checker (PMA checker).

PIC

An acronym for Position Independent Code. Position independent code is code that will execute properly no matter where it is located. The code may be moved in memory without needing to be modified.

Posits

An alternate representation of numbers.

Program Counter

A processor register dedicated to addressing instructions in memory. It is also often and perhaps more aptly called an instruction pointer. The program counter got its name because it usually increments (or counts) automatically after an instruction is fetched. In early machines in some rare cases the program counter did not count in a sequential binary fashion, but instead used other forms of a counter such as a grey counter or linear feedback shift register. In some machines the program counter addresses bundles of instructions rather than individual instructions. This is common with some stack machines where multiple instructions are packed into a memory word.

RAT

Anacronym for Register Alias Table. The RAT stores mappings of architectural registers to physical registers.

Retire

As in retire an instruction. This is the stage in processor in which the machine state is updated. Updates include the register file and memory. Buffers used for instruction storage are freed.

ROB

An acronym for ReOrder Buffer. The re-order buffer allows instructions to execute out of order yet update the machine's state in order by tracking instruction state and variables. In FT64 the re-order buffer is a circular queue with a head and tail pointers. Instructions at the head are committed if done to the machine's state then the head advanced. New instructions are queued at the buffer's tail as long as there is room in the queue. Instructions in the queue may be processed out of the order that they entered the queue in depending on the availability of resources (register values and functional units).

RSB

An acronym that stands for return stack buffer. A buffer of addresses used to predict the return address which increases processor performance. The RSB is usually small, typically 16 entries. When a return instruction is detected at time of fetch the RSB is accessed to determine the address of the next instruction to fetch. Predicting the return address allows the processing core to continuously fetch instructions in a speculative fashion without bubbles in the pipeline. The return address in the RSB may turn out to be detected as incorrect during execution of the return instruction, in which case the pipeline or instruction queue will need to be flushed and instructions fetched from the proper address.

SIMD

An acronym that stands for 'Single Instruction Multiple Data'. SIMD instructions are usually implemented with extra wide registers. The registers contain multiple data items, such as a 128-bit register containing four 32-bit numbers. The same instruction is applied to all the data items in the register

at the same time. For some applications SIMD instructions can enhance performance considerably.

Stack Pointer

A processor register dedicated to addressing stack memory. Sometimes this register is assigned by convention from the general register pool. This register may also sometimes index into a small dedicated stack memory that is not part of the main memory system. Sometimes machines have multiple stack pointers for different purposes, but they all work on the idea of a stack. For instance, in Forth machines there are typically two stacks, one for data and one for return addresses.

Telescopic Memory

A memory system composed of layers where each layer contains simplified data from the topmost layer downwards. At the topmost layer data is represented verbatim. At the bottom layer there may be only a single bit to represent the presence of data. Each layer of the telescopic memory uses far less memory than the layer above. A telescopic memory could be used in garbage collection systems. Normally however the extra overhead of updating multiple layers of memory is not warranted.

TLB

TLB stands for translation look-aside buffer. This buffer is used to store address translations for fast memory access in a system with an mmu capable of performing address translations.

Trace Memory

A memory that traces instructions or data. As instructions are executed the address of the executing instruction is stored in a trace memory. The trace memory may then be dumped to allow debugging of software. The trace memory may compress the storage of addresses by storing branch status (taken or not taken) for consecutive branches rather than storing all addresses. It typically requires only a single bit to store the branch status. However, even when branches are traced, periodically the entire address of the program executing is stored. Often trace buffers support tracing thousands of instructions.

Triadic

An instruction with three operands.

Vector Chaining

Vector chaining is a form of pipelining used with vector processors. A CPU that supports vector chaining can begin processing additional vector instructions before previous ones are complete. The processing of vector instructions is overlapped.

Vector Length (VL register)

The vector length register controls the maximum number of elements of a vector that are processed. The vector length register may not be set to a value greater than the number of elements supported by hardware. Vector registers often contain more elements than are required by program code. It would be wasteful to process all elements when only a few are needed. To improve the processing performance only the elements up to the vector length are examined.

Vector Mask (VM)

A vector mask is used to restrict which elements of a vector are processed during a vector operation. A one bit in a mask register enables the processing for that element, a zero bit disables it. The mask register is commonly set using a vector set operation.

Virtual Address

The address before segmentation and paging has been applied. This is the primary type of address a program will work with. Different programs may use the same virtual address range without being concerned about data being overwritten by another program. Although the virtual address may be the same the final physical addresses used will be different.

Writeback

A stage in a pipelined processing core where the machine state is updated. Values are ‘written back’ to the register file.

Miscellaneous

Reference Material

Below is a short list of some of the reading material the author has studied. The author has downloaded a fair number of documents on computer architecture from the web. Too many to list.

Modern Processor Design Fundamentals of Superscalar Processors by John Paul Shen, Mikko H. Lipasti. Waveland Press, Inc.

Computer Architecture A Quantitative Approach, Second Edition, by John L Hennessy & David Patterson, published by Morgan Kaufman Publishers, Inc. San Franciso, California is a good book on computer architecture. There is a newer edition of the book available.

Memory Systems Cache, DRAM, Disk by Bruce Jacob, Spencer W. Ng., David T. Wang, Samuel Rodriguez, Morgan Kaufman Publishers

PowerPC Microprocessor Developer's Guide, SAMS publishing. 201 West 103rd Street, Indianapolis, Indiana, 46290

80386/80486 Programming Guide by Ross P. Nelson, Microsoft Press

Programming the 286, C. Vieillefond, SYBEX, 2021 Challenger Drive #100, Alameda, CA 94501

Tech. Report UMD-SCA-2000-02 ENEE 446: Digital Computer Design — An Out-of-Order RiSC-16

Programming the 65C816, David Eyes and Ron Lichty, Western Design Centre Inc.

Microprocessor Manuals from Motorola, and Intel,

The SPARC Architecture Manual Version 8, SPARC International Inc, 535 Middlefield Road. Suite210 Menlo Park California, CA 94025

The SPARC Architecture Manual Version 9, SPARC International Inc, Sab Jose California, PTR Prentice Hall, Englewood Cliffs, New Jersey, 07632

The MMIX processor: 5

RISCV 2.0 Spec, Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovi´c CS Division, EECS Department, University of California, Berkeley {waterman|yunsup|pattrsn|krste}@eecs.berkeley.edu

The Garbage Collection Handbook, Richard Jones, Antony Hosking, Eliot Moss published by CRC Press 2012

RISC-V Cryptography Extensions Volume I Scalar & Entropy Source Instructions See github.com/riscv/riscv-crypto for more information.

Trademarks

WISHBONE Compatibility Datasheet

The Qupls core now uses the FTA bus which is not compatible with WISHBONE. Many signals serve a similar function to those on the WISHBONE bus so they are listed here. A bus bridge is required to interface FTA bus to WISHBONE as WISHBONE is a synchronous bus and FTA is asynchronous.

WISHBONE Datasheet WISHBONE SoC Architecture Specification, Revision B.3		
Description:	Specifications:	
General Description:	Central processing unit (CPU core)	
Supported Cycles:	MASTER, READ / WRITE MASTER, READ-MODIFY-WRITE MASTER, BLOCK READ / WRITE, BURST READ (FIXED ADDRESS)	
Data port, size: Data port, granularity: Data port, maximum operand size: Data transfer ordering: Data transfer sequencing	128 bit 8 bit 128 bit Little Endian any (undefined)	
Clock frequency constraints:	tm_clk_i must be >= 10MHz	
Supported signal list and cross reference to equivalent WISHBONE signals	Signal Name: Resp.ack_i Req.adr_o(31:0) clk_i resp.dat(127:0) req.dat(127:0) req.cyc req.stb req.wr	WISHBONE Equiv. ACK_I ADR_O() CLK_I DAT_I() DAT_O() CYC_O STB_O WE_O

	req.sel(7:0) req.cti(2:0) req.bte(1:0)	SEL_O CTI_O BTE_O
Special Requirements:		

FTA Bus

Overview

The FTA bus is an asynchronous bus meaning it does not wait for responses before beginning the next bus cycle. It is a request and response bus. Requests are outgoing from a bus master and incoming to a bus slave. Responses are output by a bus slave and input by a bus master. FTA bus includes standard signals for address, data, and control. These signals should be like those found on many other busses.

Bus Tags

The bus has tagged transactions; there is an id tag associated with each bus transaction. The id tag contains identifiers for the core, channel, and transaction. The core is a core number for a multi-core CPU. Channel selects a particular channel in the core which may for instance be a data channel or an instruction channel. Finally, the transaction id identifies the specific transaction. Incoming responses are matched against transactions that were outgoing. For instance, a bus master may issue a burst request for four bus transactions to fill a cache line. Each transaction will have an id associated with it. When the slave receives the transactions it sends back responses for each of the four requests with ids that match those in the request. The slave does not necessarily send back responses in the same order. Transaction requests from the master may not arrive in order.

*An id tag of all zeros is illegal – it represents the bus available state.

Single Cycle

The bus operates on a single cycle basis. Transaction requests and responses are routed through the soc interconnect network as the bus is available and are present for only a single clock cycle. Bus bridges may buffer the transactions for a short period of time. Generally, requests going out from masters do not need buffering as access to the bus will have been arbitrated before the bus cycle begins. Responses coming back from slaves may need to be buffered as two slaves may respond at the same time.

Retry

If the bus is unavailable the retry response signal is asserted to the master. The master must retry the transaction.

Signal Description

Following is a signal description for requests and responses for a 128-bit data version of the bus. Signal values have been chosen so that a value of zero represents a bus idle state. If nothing is on the bus it will be all zeros.

Requests

Signal	Width	Description	
Om	2	Operating mode	
Cmd	5	Command for bus controller or memory controller	
Bte	3	Burst type	
Cti	3	Cycle type	
Blen	6	Burst length -1 (0=1 to 63=64)	
sz	4	Transfer size	
Segment	3	Code, data, or stack	
Cyc	1	Bus cycle is valid	
Stb	1	Data strobe	
We	1	Write enable	
Asid	16	Address space id	
Vadr	32/64	Virtual address	
Padr	32/64	Physical address	
Sel	16	Byte lane selects	
Data1	128	First data item	
Data2	128	Second data item	
Tid	13	Transaction id	
Csr	1	Clear or set address reservation	
Pl	8	Privilege level	
Pri	4	Transaction priority (higher is better)	
Cache	4	Transaction cacheability	

Responses

Signal	Width	Description	
Tid	13	Transaction id	
Stall	1	Stall pipeline	
Next	1	Advance to next transaction	
Ack	1	Request acknowledgement (data is available)	
Rty	1	Retry transaction	
Err	1	An error occurred	
Pri	4	Transaction priority	

Adr	32/64	Physical address	
Dat	128	Response data	

Om

Operating mode, this corresponds to the operating mode of the CPU. Some devices are limited to specific modes.

Cmd

Command for memory controller. This is how the memory controller knows what to do with the data.

Oridinal		
0	CMD_NONE	No command
1	CMD_LOAD	Perform a sign extended data load operation
2	CMD_LOADZ	Perform a zero extended data load operation
3	CMD_STORE	Perform a data store operation
4	CMD_STOREPTR	Perform a pointer store operation
7	CMD_LEA	Load the effective address
10	CMD_DCACHE_LOAD	Perform load operation intended for data cache
11	CMD_ICACHE_LOAD	Perform load operation intended for instruction cache
13	CMD_CACHE	Issue a cache control command
16	CMD_SWAP	AMO swap operation
18	CMD_MIN	AMO min operation
19	CMD_MAX	AMO max operation
20	CMD_ADD	AMO add operation
22	CMD_ASL	AMO left shift operation
23	CMD_LSR	AMO right shift operation
24	CMD_AND	AMO and operation
25	CMD_OR	AMO or operation
26	CMD_EOR	AMO exclusive or operation
28	CMD_MINU	AMO unsigned minimum operation
29	CMD_MAXU	AMO unsigned maximum operation
31	CMD_CAS	AMO compare and swap
Others		reserved

BTE

Burst type extension.

Ordinal	
0	Linear

1	Wrap 4
2	Wrap 8
3	Wrap 16
4	Wrap 32
5	Wrap 64
6	Wrap 128
7	reserved

CTI

Cycle Type Indicator

Ordinal		Comment
0	Classic	
1	fixed	Constant data address
2	Incr	Incrementing data address
3	erc	Record errors on write
4	Irqa	Interrupt acknowledge
7	Eob	End of burst
others		reserved

Normally write cycles do not send a response back to the master. The ERC cycle type indicates that the master wants a response back from a write operation.

Blen

Burst length, this is the number of transactions in the burst minus one. There is a maximum of 64 transactions. With a 128-bit bus this is 1024 bytes of data.

Sz

Transfer size.

Ordinal		Transfer size
0	Nul	Nothing is transferred
1	Byt	A single byte
2	Wyde	Two bytes
3	Tetra	Four bytes
4	Penta	Five bytes
5	Octa	Eight bytes
6	Hexi	Sixteen bytes

10	vect	A vector 64 bytes (512 bit bus)
Others		Reserved

Segment

The memory segment associated with the transfer.

Ordinal	
0	data
6	stack
7	code
others	reserved

TID

Transaction ID. This is made up of three fields.

Size	Use
6	Core number
3	Channel
4	Tran id

Cache

Cache-ability of transaction. A transaction may be non-cacheable meaning as it progresses through the cache hierarchy it does not store data in the cache. It only stores data when it reaches the final memory destination.

Ordinal		
0	NC_NB	Non cacheable, non bufferable
1	NON_CACHEABLE	
2	CACHEABLE_NB	Cacheable, non bufferable
3	CACHEABLE	
8	WT_NO_ALLOCATE	Write-through without allocating
9	WT_READ_ALLOCATE	
10	WT_WRITE_ALLOCATE	
11	WT_READWRITE_ALLOCATE	
12	WB_NO_ALLOCATE	Write-back without allocating

13	WB_READ_ALLOCATE	
14	WB_WRITE_ALLOCATE	
15	WB_READWRITE_ALLOCATE	