Stark

Table of Contents

[Preface 1](#_Toc192189683)

[Who This Book is For 1](#_Toc192189684)

[About the Author 1](#_Toc192189685)

[Motivation 1](#_Toc192189686)

[History 2](#_Toc192189687)

[Features of Qupls2 2](#_Toc192189688)

[Programming Model 3](#_Toc192189689)

[Register File 3](#_Toc192189690)

[General Purpose Registers 3](#_Toc192189691)

[Predicate Registers 5](#_Toc192189692)

[Code Address Registers 5](#_Toc192189693)

[SR - Status Register (CSR 0x?004) 7](#_Toc192189694)

[Special Purpose Registers 9](#_Toc192189695)

[Operating Modes 14](#_Toc192189696)

[Instruction Descriptions 15](#_Toc192189697)

[Arithmetic Operations 15](#_Toc192189698)

[Representations 15](#_Toc192189699)

[Arithmetic Operations 17](#_Toc192189700)

[ADD - Register-Register 18](#_Toc192189701)

[ADDI - Add Immediate 19](#_Toc192189702)

[ADD2UI - Add Immediate 20](#_Toc192189703)

[ADD4UI - Add Immediate 21](#_Toc192189704)

[ADD8UI - Add Immediate 22](#_Toc192189705)

[BYTENDX – Character Index 23](#_Toc192189706)

[Multiply 24](#_Toc192189707)

[BMM – Bit Matrix Multiply 25](#_Toc192189708)

[MUL – Multiply Register-Register 26](#_Toc192189709)

[MULI - Multiply Immediate 27](#_Toc192189710)

[MULSU – Multiply Signed Unsigned 28](#_Toc192189711)

[MULU – Unsigned Multiply Register-Register 29](#_Toc192189712)

[MULUI - Multiply Unsigned Immediate 30](#_Toc192189713)

[Data Movement 31](#_Toc192189714)

[BMAP – Byte Map 31](#_Toc192189715)

[MOVE – Move Register to Register 33](#_Toc192189716)

[MUX – Multiplex 35](#_Toc192189717)

[Logical Operations 36](#_Toc192189718)

[AND – Bitwise And 36](#_Toc192189719)

[ANDI - Add Immediate 37](#_Toc192189720)

[EOR – Bitwise Exclusive Or 38](#_Toc192189721)

[EORI – Exclusive Or Immediate 39](#_Toc192189722)

[Shift and Rotate Operations 40](#_Toc192189723)

[ASLP –Arithmetic Shift Left Pair 41](#_Toc192189724)

[LSRP –Logic Shift Right Pair 42](#_Toc192189725)

[Floating-Point Operations 43](#_Toc192189726)

[Precision 43](#_Toc192189727)

[Representations 43](#_Toc192189728)

[NaN Boxing 44](#_Toc192189729)

[Rounding Modes 44](#_Toc192189730)

[FMA –Float Multiply and Add 46](#_Toc192189731)

[LDsz Rn, <ea> - Load Register 47](#_Toc192189732)

[Branch / Flow Control Instructions 49](#_Toc192189733)

[Overview 49](#_Toc192189734)

[Conditional Branch Format 49](#_Toc192189735)

[Branch Conditions 49](#_Toc192189736)

[Branch Target 52](#_Toc192189737)

[Incrementing / Decrementing Branches 53](#_Toc192189738)

[Unconditional Branches 53](#_Toc192189739)

[BEQ –Branch if Equal 54](#_Toc192189740)

[Modifiers 55](#_Toc192189741)

[ATOM 55](#_Toc192189742)

[QEXT Prefix 56](#_Toc192189743)

[PFX[ABC] – A/B/C Immediate Postfix 57](#_Toc192189744)

[Qupls2 Opcodes 58](#_Toc192189745)

# 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](#_JSR_–_Jump), and branch to subroutine, [BSR](#_BSR_–_Branch), instructions update a link register. The return from subroutine,. [RTS](#_RTS_–_Return), 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 7 | ipl | Interrupt level |
| 8 | ssm | Single step mode |
| 9 | te | Trace enable |
| 10 to 11 | om | Operating mode |
| 12 to 13 | ps | Pointer size |
| 14 | ~ | reserved |
| 15 | dbg | Debug mode |
| 16 | mprv | memory privilege |
| 17 to 19 | ~ | reserved |
| 20 to 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.

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.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 4 | 3 | 2 | 1 | 0 |
| ~ | mie | hie | sie | uie |

#### [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 8 bits are implemented. The high order bits read as zero and are not updateable.

#### 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.

|  |  |  |
| --- | --- | --- |
| 63 60 | 59 16 | 15 0 |
| State4 | ~44 | seed16 |

|  |  |
| --- | --- |
| State4 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.

|  |  |
| --- | --- |
| 63 16 | 15 0 |
| Virtual Address67..20 | 016 |

#### 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…FD0000

|  |
| --- |
| 63 0 |
| Virtual Address75..12 |

#### 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 |  |  |  |  |  |

# Instruction Descriptions

Instruction Length

|  |  |
| --- | --- |
| Ln2 | Bits |
| 0 | 24 |
| 1 | 48 |
| 2 | 96 |
| 3 |  |

## Arithmetic Operations

### Representations

#### int

|  |
| --- |
| 63 0 |
| 64 bits |

#### short int

|  |
| --- |
| 31 0 |
| 32 bits |

#### wyde

|  |
| --- |
| 15 0 |
| 16 bits |

#### byte

|  |
| --- |
| 7 0 |
| 8 bits |

#### decimal

|  |  |
| --- | --- |
| 127 120 | 119 0 |
|  | 120 bits |

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, Imm32**

**Instruction Format:** RI

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 47 21 | 20 19 | 18 14 | 13 9 | 8 2 | 1 0 |
| Immediate27 | Prc2 | Ra5 | Rt5 | 47 | 12 |

#### **Precision**

Four different precisions are supported encoded by the Prc2 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.

|  |  |
| --- | --- |
| Prc2 | 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.

### 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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 19 | 18 14 | 13 9 | 8 2 | 1 0 |
| Rb5 | Ra5 | Rt5 | 167 | 02 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 47 | Op4 | Nc | Rc6 | Nb | Rb6 | Nb | Ra6 | Nt | Rt6 | Opcode7 | 02 |

**Operation:**

|  |  |
| --- | --- |
| Opcode7 | Precision |
| 104 | Byte parallel |
| 105 | Wyde parallel |
| 106 | Tetra parallel |
| 107 | octa |
| 2 | hexi |

|  |  |  |
| --- | --- | --- |
| OP4 |  | 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 |
| 4 to 7 | 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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 19 | 18 14 | 13 9 | 8 2 | 1 0 |
| Imm5 | Ra5 | Rt5 | 47 | 02 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate23 | Prc2 | Na | Ra6 | Nt | Rt6 | 47 | 12 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 95 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate71 | Prc2 | Na | Ra6 | Nt | Rt6 | 47 | 22 |

**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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 25 | 2423 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate23 | Prc2 | Na | Ra6 | Nt | Rt6 | 607 | 12 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 95 25 | 2423 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate71 | Prc2 | Na | Ra6 | Nt | Rt6 | 607 | 22 |

**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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 25 | 2423 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate23 | Prc2 | Na | Ra6 | Nt | Rt6 | 617 | 12 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 95 25 | 2423 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate71 | Prc2 | Na | Ra6 | Nt | Rt6 | 617 | 22 |

**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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 25 | 2423 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate23 | Prc2 | Na | Ra6 | Nt | Rt6 | 627 | 12 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 95 25 | 2423 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate71 | Prc2 | Na | Ra6 | Nt | Rt6 | 627 | 22 |

**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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 25 | 2423 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate23 | Prc2 | Na | Ra6 | Nt | Rt6 | 637 | 12 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 95 25 | 2423 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate71 | Prc2 | Na | Ra6 | Nt | Rt6 | 637 | 22 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = (Ra << 4) + 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.

Rb may be replaced by an immediate value.

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

**Instruction Format: R3 (byte)**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 | 38 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 377 | Bi | Imm9 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | 1077 | 12 |

**Operation:**

Rt = Index of (Rb in Ra)

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

### 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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 147 | ~4 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | 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)**

## Multiply

### 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 |
| 347 | Op3 | ~ | ~6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | 1077 | 12 |

**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.

### 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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 167 | Op4 | Nc | Rc6 | Nb | Rb6 | Nb | Ra6 | Nt | Rt6 | Opcode7 | 02 |

|  |  |
| --- | --- |
| Opcode7 | Precision |
| 104 | Byte parallel |
| 105 | Wyde parallel |
| 106 | Tetra parallel |
| 107 | octa |
| 2 | hexi |

|  |  |  |
| --- | --- | --- |
| OP4 |  | 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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 19 | 18 14 | 13 9 | 8 2 | 1 0 |
| Imm5 | Ra5 | Rt5 | 67 | 02 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate23 | Prc2 | Na | Ra6 | Nt | Rt6 | 67 | 12 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 95 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate71 | Prc2 | Na | Ra6 | Nt | Rt6 | 67 | 22 |

**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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 217 | Op4 | Nc | Rc6 | Nb | Rb6 | Nb | Ra6 | Nt | Rt6 | Opcode7 | 02 |

|  |  |
| --- | --- |
| Opcode7 | 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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 197 | Op4 | Nc | Rc6 | Nb | Rb6 | Nb | Ra6 | Nt | Rt6 | Opcode7 | 02 |

|  |  |
| --- | --- |
| Opcode7 | Precision |
| 104 | Byte parallel |
| 105 | Wyde parallel |
| 106 | Tetra parallel |
| 107 | octa |
| 2 | hexi |

|  |  |  |
| --- | --- | --- |
| OP4 |  | 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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 19 | 18 14 | 13 9 | 8 2 | 1 0 |
| Imm5 | Ra5 | Rt5 | 147 | 02 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate23 | Prc2 | Na | Ra6 | Nt | Rt6 | 147 | 12 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 95 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate71 | Prc2 | Na | Ra6 | Nt | Rt6 | 147 | 22 |

**Clock Cycles:** 4

**Execution Units:** All ALUs

**Operation:**

Rt = Ra \* immediate

**Exceptions:**

**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**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 357 | Op4 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | 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:**

### CMOVNZ – Conditional Move if Non-Zero

**CMOVNZ 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 |
| 127 | ~4 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opc7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | 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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 187 | 24 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opc7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | 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 |
| 237 | 24 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opc7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | 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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 187 | 14 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opc7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | 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 |
| 237 | 14 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opc7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | 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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 187 | 04 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opc7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | 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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 237 | 04 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opc7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | 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 | Ra6 | Nt | Rt6 | 157 | 02 |

**Operation: R2**

Rt = Ra

**Clock Cycles:** 1

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

|  |  |  |  |
| --- | --- | --- | --- |
| Ra6 / Rt6 | 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 a bit in Ra is set then the bit of the target register is set to the corresponding bit in Rb, otherwise the bit in the target register is set to the corresponding bit in Rc.

**Instruction Format:** R3

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 357 | Op4 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opc7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | Precision |
| 104 | reserved |
| 105 | reserved |
| 106 | reserved |
| 107 | octa |
| 2 | hexi |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

For n = 0 to 63

If Ra[n] is set then

Rt[n] = Rb[n]

else

Rt[n] = Rc[n]

**Exceptions:** none

### 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 | Imm6 | Nt | Rt6 | 37 | 02 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 1 | Na | Ra6 | Nt | Rt6 | 37 | 02 |

**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 | Imm6 | Nt | Rt6 | 37 | 02 |

**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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 19 | 18 14 | 13 9 | 8 2 | 1 0 |
| Rb5 | Ra5 | Rt5 | 167 | 02 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 07 | Op4 | Nc | Rc6 | Nb | Rb6 | Nb | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Operation:**

|  |  |
| --- | --- |
| Opcode7 | Precision |
| 104 | Byte parallel |
| 105 | Wyde parallel |
| 106 | Tetra parallel |
| 107 | octa |
| 2 | hexi |

|  |  |  |
| --- | --- | --- |
| OP4 |  | 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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 19 | 18 14 | 13 9 | 8 2 | 1 0 |
| Imm5 | Ra5 | Rt5 | 87 | 02 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate23 | Prc2 | Na | Ra6 | Nt | Rt6 | 87 | 12 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 95 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate71 | Prc2 | Na | Ra6 | Nt | Rt6 | 87 | 22 |

**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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 27 | Op4 | Nc | Rc6 | Nb | Rb6 | Nb | Ra6 | Nt | Rt6 | Opcode7 | 12 |

|  |  |  |
| --- | --- | --- |
| OP4 |  | 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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 19 | 18 14 | 13 9 | 8 2 | 1 0 |
| Imm5 | Ra5 | Rt5 | 107 | 02 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate23 | Prc2 | Na | Ra6 | Nt | Rt6 | 107 | 12 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 95 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate71 | Prc2 | Na | Ra6 | Nt | Rt6 | 107 | 22 |

**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 |
| 17 | Op4 | Nc | Rc6 | Nb | Rb6 | Nb | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Operation:**

|  |  |  |
| --- | --- | --- |
| OP4 |  | 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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 19 | 18 14 | 13 9 | 8 2 | 1 0 |
| Imm5 | Ra5 | Rt5 | 97 | 02 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate23 | Prc2 | Na | Ra6 | Nt | Rt6 | 97 | 12 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 95 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate71 | Prc2 | Na | Ra6 | Nt | Rt6 | 97 | 22 |

**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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 37 | Op4 | Nc | Rc6 | Nb | Rb6 | Nb | Ra6 | Nt | Rt6 | Opcode7 | 02 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 27 | 26 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| 35 | Rc8 | Rb5 | Ra5 | Rt5 | 797 |

|  |  |
| --- | --- |
| Opc7 | Precision |
| 104 | Byte parallel |
| 105 | Wyde parallel |
| 106 | Tetra parallel |
| 107 | octa |
| 2 | hexi |

|  |  |  |
| --- | --- | --- |
| OP4 |  | 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 |

|  |  |  |  |
| --- | --- | --- | --- |
| Fn4 | 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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 19 | 18 14 | 13 9 | 8 2 | 1 0 |
| Imm5 | Ra5 | Rt5 | 117 | 02 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate23 | Prc2 | Na | Ra6 | Nt | Rt6 | 117 | 12 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 95 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate71 | Prc2 | Na | Ra6 | Nt | Rt6 | 117 | 22 |

|  |  |  |  |
| --- | --- | --- | --- |
| 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, Imm4

**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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 967 | N4 | Nc | Rc6 | Nb | Rb6 | Nb | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Operation: R3**

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

|  |  |
| --- | --- |
| Opc7 | 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, Imm4

**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 |
| 997 | N4 | Nc | Rc6 | Nb | Rb6 | Nb | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Operation: R3**

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

|  |  |
| --- | --- |
| Opc7 | 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, Imm4

**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 |
| 987 | N4 | Nc | Rc6 | Nb | Rb6 | Nb | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Operation: R3**

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

|  |  |
| --- | --- |
| Opc7 | 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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate23 | Prc2 | Na | Ra6 | Nt | Rt6 | 227 | 12 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 95 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate71 | Prc2 | Na | Ra6 | Nt | Rt6 | 227 | 22 |

**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, Imm4

**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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 997 | N4 | Nc | Rc6 | Nb | Rb6 | Nb | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Operation: R3**

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

|  |  |
| --- | --- |
| Opc7 | 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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate23 | Prc2 | Na | Ra6 | Nt | Rt6 | 947 | 12 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 95 25 | 24 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| Immediate71 | Prc2 | Na | Ra6 | Nt | Rt6 | 947 | 22 |

**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.

|  |  |
| --- | --- |
| Opcode7 | 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 |
| 54 | ~7 | Nc | Rc6 | 0 | 06 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Instruction Format:** SHIFTI

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 44 | 43 37 | 36 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 134 | ME7 | MB7 | 0 | 06 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**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 |
| 64 | ~7 | Nc | Rc6 | 1 | 06 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Instruction Format:** SHIFTI

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 44 | 43 37 | 36 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 144 | ME7 | MB7 | 1 | 06 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**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 |
| 54 | ~7 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Instruction Format:** SHIFTI

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 44 | 43 37 | 36 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 134 | ME7 | MB7 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**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 |
| 64 | ~7 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Instruction Format:** SHIFTI

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 44 | 43 37 | 36 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 144 | ME7 | MB7 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 44 | 43 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 24 | ~7 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Instruction Format:** SHIFTI

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 44 | 43 37 | 36 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 104 | ME7 | MB7 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | 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 |
| 14 | ~7 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Instruction Format:** SHIFTI

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 44 | 43 37 | 36 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 94 | ME7 | MB7 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | 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 |
| 04 | H | ~6 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Instruction Format:** SHIFTI

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 44 | 43 | 42 37 | 36 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 84 | H | ~6 | Imm7 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**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 |
| 54 | ~7 | Nc | Rc6 | 1 | 06 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Instruction Format:** SHIFTI

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 44 | 43 37 | 36 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 134 | ME7 | MB7 | 1 | 06 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**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 |
| 14 | ~7 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Instruction Format:** SHIFTI

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 44 | 43 37 | 36 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 94 | ME7 | MB7 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | 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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 19 | 18 14 | 13 9 | 8 2 | 1 0 |
| Imm5 | Ra5 | Rt5 | Opcode7 | 02 |

**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 |
| 04 | H | ~6 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Instruction Format:** SHIFTI

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 44 | 43 | 42 37 | 36 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 84 | H | ~6 | Imm7 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**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 |
| 24 | ~7 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Instruction Format:** SHIFTI

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 44 | 43 37 | 36 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 104 | ME7 | MB7 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | 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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 44 | 43 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 34 | ~7 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Instruction Format:** SHIFTI

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 44 | 43 37 | 36 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 114 | ME7 | MB7 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | 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 |
| 44 | ~7 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Instruction Format:** SHIFTI

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 44 | 43 37 | 36 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 124 | ME7 | MB7 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | 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 3 7 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 14 | ~7 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

**Instruction Format:** SHIFTI

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 44 | 43 37 | 36 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| 94 | Me7 | MB7 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | Opcode7 | 12 |

|  |  |
| --- | --- |
| Opcode7 | 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.

|  |  |  |
| --- | --- | --- |
| Opcode7 | 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

|  |  |  |
| --- | --- | --- |
| 63 | 62 52 | 51 0 |
| S | Exponent11 | Significand52 |

Single Precision, float

|  |  |  |
| --- | --- | --- |
| 31 | 30 23 | 22 0 |
| S | Exponent8 | Significand23 |

Double Precision, Two’s Complement Form:

|  |  |
| --- | --- |
| 63 53 | 52 0 |
| Exponent11 | Significand53 |

#### Decimal Floats

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

|  |  |  |  |
| --- | --- | --- | --- |
| 95 | 94 90 | 89 80 | 79 0 |
| S | Combo5 | Exponent10 | Significand80 |

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 101536

48-bit single precision decimal floating point:

|  |  |  |  |
| --- | --- | --- | --- |
| 47 | 46 42 | 41 34 | 33 0 |
| S | Combo5 | Exponent8 | Significand34 |

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 | Exponent16 | Significand111 | |
| S | FFFFh | 7FFFFFFFFFFFh | Double Precision Float64 |

### 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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Immediate17 | Ra6 | Func3 | Rt6 | Opcode5 | P3 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Opcode25 | Rc6 | Rb6 | Ra6 | Func3 | Rt6 | Opcode5 | P3 |

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

### 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**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 39 | 38 36 | 35 29 | 28 24 | 23 19 | 18 14 | 13 9 | 8 2 | 1 0 |
|  | Prc2 | Rm3 | ~7 | Rc5 | Rb5 | Ra5 | Rt5 | Opc7 | 12 |

**Operation:**

Rt = Ra \* Rb + Rc

**Clock Cycles:** 8

**Execution Units:** All FPUs

**Exceptions:** none

**Notes:**

### 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 eighteen-bit displacement.

The ‘H’ bit indicates to update the high-half of a 128-bit register.

The capabilities tag bit of the register is cleared.

**Instruction Format:** d[Ra]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 19 | 18 14 | 13 9 | 8 2 | 1 0 |
| Disp5 | Ra5 | Rt5 | 647 | 02 |

**Instruction Format:** d[Ra+Rb\*Sc]

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 30 | 29 28 | 2725 | 24 | 23 19 | 18 14 | 13 9 | 8 2 | 1 0 |
| Immediate18 | Prc2 | Sc3 | H | Rb5 | Ra5 | Rt5 | 647 | 12 |

Prc2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Prc | | | |
| Opcode | 0 | 1 | 2 | 3 |
| 64 | LDB | LDW | LDT | LDO |
| 65 | LDBU | LDWU | LDTU |  |
| 66 |  | FLDH | FLDS | FLDD |
| 67 |  |  |  | DFLDD |
| 68 |  | PLDW | PLDT | PLDO |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Prc | | | |
| Opcode | 0 | 1 | 2 | 3 |
| 72 | STB | STW | STT | STO |
| 73 |  |  |  |  |
| 74 |  | FSTH | FSTS | FSTD |
| 75 |  |  |  | DFSTD |
| 76 |  | PSTW | PSTT | PSTO |

**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.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | ▼ | ▼ |  |
| 47 46 | 45 | 44 26 | 25 20 | 19 14 | 13 | 12 9 | 8 2 | 1 0 |
| Prc2 | R | Tgt18…0 | Rb6 | Ra6 | A | Fn4 | 2xh7 | 12 |

|  |  |
| --- | --- |
| 2x | Data Type Compared |
| 28h | (Unsigned) Address |
| 29h | (Signed) Integers |
| 2Ah | Reserved |
| 2Bh | Decimal Float |
| 2Ch | Binary Float |
| 2Dh | Posit |
| 2Eh | Incr. Integer |
| 2Fh | Decr. Integer |

Integer / Address Conditions

|  |  |  |  |
| --- | --- | --- | --- |
| Fn4 | 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

|  |  |  |  |
| --- | --- | --- | --- |
| Fn4 | 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 ±750kB or 20.5 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. **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.*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | ▼ |  |  |  |  |  |  |
| 47 46 | 45 | 44 26 | 25 20 | 19 14 | 13 | 12 9 | 8 2 | 1 0 |
| Prc2 | R | Tgt18…0 | Rb6 | Ra6 | A | Fn4 | 2xh7 | 12 |

|  |  |
| --- | --- |
| A | Mode |
| 0 | Relative |
| 1 | Absolute |
| R | Register |
| 0 | Displacement / Address |
| 1 | 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.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | ▼ | ▼ |  |
| 47 46 | 45 | 44 26 | 25 20 | 19 14 | 13 | 12 9 | 8 2 | 1 0 |
| Prc2 | R | Tgt18…0 | Rb6 | Ra6 | A | Fn4 | 46/477 | 12 |

|  |  |
| --- | --- |
| Opcode | Effect on Ra |
| Other | No change |
| 46 | Increment Ra |
| 47 | 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 ±235 range.

|  |  |  |  |
| --- | --- | --- | --- |
| ▼ |  |  |  |
| 47 11 | 10 9 | 8 2 | 1 0 |
| Tgt36...0 | Lkt2 | 327 | 12 |

### 

### BEQ –Branch if Equal

BEQ Ra, Rb, label

**Description:**

Branch if source operands are equal.

**Formats Supported**: BR

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |
| 47 46 | 45 | 44 26 | 25 20 | 19 14 | 13 | 12 9 | 8 2 | 1 0 |
| Prc2 | 0 | Tgt18…0 | Rb6 | Ra6 | A | 04 | 467 | 12 |

**Register Indirect Target**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  | ▼ |  |
| 47 46 | 45 | 44 32 | 31 26 | 25 20 | 19 14 | 13 | 12 9 | 8 2 | 1 0 |
| Prc2 | 1 | Disp12…0 | Rc6 | Rb6 | Ra6 | A | 04 | 467 | 12 |

**Clock Cycles: 13**

## Modifiers

### ATOM

**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 thirty-two instructions. The non-maskable interrupt may not be masked.

Note that since the processor fetches instructions in groups the mask effectively applies to groups. 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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 35 | 34 32 | 21 17 | 16 12 | 11 8 | 7 3 | 2 0 |
| 05 | ~13 | Count5 | ~5 | LVL4 | 315 | P3 |

**Assembler Syntax:**

**Example:**

|  |
| --- |
| ATOM 6,7  LOAD a0,[a3]  SLT t0,a0,a1  PRED t0,”TTF”  STORE a2,[a3]  LDI a0,1  LDI a0,0 |

|  |
| --- |
| ATOM 4,6  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

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 37 | 36 | 35 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 2 | 1 0 |
| ~11 | Nc | Rc6 | Nb | Rb6 | Na | Ra6 | Nt | Rt6 | 1207 | 12 |

### PFX[ABC] – A/B/C 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. Ra if A/B/C =0, otherwise Rb if A/B/C=1, otherwise Rc.

\*Only one postfix is supported per instruction.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 23 11 | 10 9 | 8 2 | 1 0 |
| Immediate13...5 | ABC | 1247 | 02 |

|  |  |  |  |
| --- | --- | --- | --- |
| 47 11 | 10 9 | 8 2 | 1 0 |
| Immediate41...5 | ABC | 1247 | 12 |

|  |  |  |  |
| --- | --- | --- | --- |
| 95 11 | 10 9 | 8 2 | 1 0 |
| Immediate89...5 | ABC | 1247 | 22 |

**Notes:**

# Qupls2 Opcodes

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **0x** | 0  CHK  CHKCPL | 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  {FLT.64} | 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  JMP reg  JSR reg | 39  CJMP reg  CJSR reg |
|  | 40  BccU | 41  Bcc | 42 | 43  DFBcc | 44  FBcc | 45 | 46 | 47 |
| **3x** | 48  {FLT.32} | 49 | 50 | 51 | 52  ENTER | 53  LEAVE | 54  PUSH | 55  POP |
|  | 56  {FLT.16} | 57  LDA | 58  AIPSI | 59 | 60  ADD2UI | 61  ADD4UI | 62  ADD8UI | 63  ADD16UI |
| **4x** | 64  LDx | 65  LDxU | 66  FLDx | 67  DFLDx | 68  PLDx | 69  CLOAD | 70  CACHE | 71  LDCTX |
|  | 72  STX | 73 | 74  FSTx | 75  DFSTx | 76  PSTx | 77  CSTORE | 78  STPTR | 79  STCTX |
| **5x** | 80  {SHFT.8} | 81  {SHFT.16} | 82  {SHFT.32} | 83  {SHFT.64} | 84 | 85 | 86 | 87 |
|  | 88  {SHIFT} | 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 | 113  STOP | 114  FENCE | 115  PFI | 116 | 117 | 118 | 119 |
|  | 120  QEXT | 121  PRED | 122  ATOM | 123 | 124  PFXABC | 125 | 126 | 127  NOP |