Robert Finch

[Draw your reader in with an engaging abstract. It is typically a short summary of the document. When you’re ready to add your content, just click here and start typing.]

Thor2023

[Document subtitle]

Table of Contents

[RF65000 2](#_Toc120802753)

[CC - Condition Codes Register Group 3](#_Toc120802754)

[SR - Status Register 3](#_Toc120802755)

[AC – Application Control Register 3](#_Toc120802756)

[VB – Vector Base Register 4](#_Toc120802757)

[Exceptions 5](#_Toc120802758)

[Instruction Descriptions 6](#_Toc120802759)

[Major Opcode 6](#_Toc120802760)

[Operand Sizes 8](#_Toc120802761)

[Arithmetic Operations 9](#_Toc120802762)

[ABS – Absolute Value 9](#_Toc120802763)

[ADC – Add with Carry 10](#_Toc120802764)

[ADD - Addition 12](#_Toc120802765)

[AND – Bitwise And 14](#_Toc120802766)

[BIT – Bitwise And 15](#_Toc120802767)

[CMP - Comparison 16](#_Toc120802768)

[CSR – Control and Special Registers Operations 18](#_Toc120802769)

[DIVS – Signed Division 19](#_Toc120802770)

[DIVU – Unsigned Division 21](#_Toc120802771)

[EOR – Bitwise Exclusive Or 22](#_Toc120802772)

[LOADQ – Load Quick Immediate 23](#_Toc120802773)

[MULS – Multiply Signed 24](#_Toc120802774)

[MULU – Unsigned Multiplication 26](#_Toc120802775)

[OR – Bitwise Or 28](#_Toc120802776)

[SUB - Subtraction 29](#_Toc120802777)

[Floating-Point Operations 30](#_Toc120802778)

[FCMP - Comparison 30](#_Toc120802779)

[Bit Manipulation Operations 32](#_Toc120802780)

[BCLR – Clear Bit 33](#_Toc120802781)

[BCHG – Change Bit 34](#_Toc120802782)

[BPCHG – Change Bit Pair 35](#_Toc120802783)

[BPCLR – Clear Bit Pair 37](#_Toc120802784)

[BPTST – Test Bit Pair 38](#_Toc120802785)

[BSET – Set Bit 39](#_Toc120802786)

[BTST – Test Bit 40](#_Toc120802787)

[Shift and Rotate Operations 41](#_Toc120802788)

[ASL – Arithmetic Shift Left 41](#_Toc120802789)

[ASR – Arithmetic Shift Right 42](#_Toc120802790)

[LSL – Logical Shift Left 43](#_Toc120802791)

[LSR – Logical Shift Right 44](#_Toc120802792)

[ROL – Rotate Left 45](#_Toc120802793)

[ROLC – Rotate Left through Carry 46](#_Toc120802794)

[ROR – Rotate Right 47](#_Toc120802795)

[RORC – Rotate Right through Carry 48](#_Toc120802796)

[SWAP – Swap register halves 49](#_Toc120802797)

[Flow Control Instructions 50](#_Toc120802798)

[ATRAP – Application Trap 50](#_Toc120802799)

[Bcc – Conditional Branch 51](#_Toc120802800)

[BRA – Unconditional Branch 52](#_Toc120802801)

[BRK – Breakpoint 52](#_Toc120802802)

[BSR – Branch to Subroutine 52](#_Toc120802803)

[DBcc – Decrement and Branch 53](#_Toc120802804)

[JMP – Jump to Address 54](#_Toc120802805)

[JSR – Jump to Subroutine 55](#_Toc120802806)

[RTI – Return From Interrupt 57](#_Toc120802807)

[ARTI – Application Return from Interrupt 58](#_Toc120802808)

[RTS – Return from Subroutine 59](#_Toc120802809)

[TRAP – Trap 60](#_Toc120802810)

[TRAPV – Trap on Overflow 61](#_Toc120802811)

[Memory Operations 62](#_Toc120802812)

[Flag Updates 62](#_Toc120802813)

[LOAD Rn,<sea> 63](#_Toc120802814)

[LOAD Reglist,<sea> 64](#_Toc120802815)

[PEA <ea> 66](#_Toc120802816)

[STORE Rn,<dea> 67](#_Toc120802817)

[STORE #imm,<dea> 68](#_Toc120802818)

[STORE Reglist,<dea> 69](#_Toc120802819)

### Thor2023

## Register File

### Rn – General Purpose Registers

The register file contains 64 96-bit general purpose registers. The register file is *unified*; register may hold integer or floating-point values. The stack pointer is banked with a separate stack pointer for each operation mode.

|  |  |  |  |
| --- | --- | --- | --- |
| Regno | ABI | ABI Usage |  |
| 0 | 0 | Always zero |  |
| 1 | A0 | First argument / return value register |  |
| 2 | A1 | Second argument / return value register |  |
| 3 | T0 | Temporary register, caller save |  |
| 4 | T1 | Temporary register |  |
| 5 | T2 | Temporary register |  |
| 6 | T3 | Temporary register |  |
| 7 | T4 | Temporary register |  |
| 8 | T5 | Temporary register |  |
| 9 | T6 | Temporary register |  |
| 10 | T7 | Temporary register |  |
| 11 | T8 | Temporary register |  |
| 12 | T9 | Temporary register |  |
| 13 | S0 | Saved register, register variables |  |
| 14 | S1 | Saved register |  |
| 15 | S2 | Saved register |  |
| 16 | S3 | Saved register |  |
| 17 | S4 | Saved register |  |
| 18 | S5 | Saved register |  |
| 19 | S6 | Saved register |  |
| 20 | S7 | Saved register |  |
| 21 | S8 | Saved register |  |
| 22 | S9 | Saved register |  |
| 23 | A2 | Third argument register |  |
| 24 | A3 | Argument register |  |
| 25 | A4 | Argument register |  |
| 26 | A5 | Argument register |  |
| 27 | A6 | Argument register |  |
| 28 | A7 | Argument register |  |
| 29 | A8 | Argument register |  |
| 30 | A9 | Argument register |  |
| 31 |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 32 |  |  |  |
| 33 |  |  |  |
| 34 |  |  |  |
| 35 |  |  |  |
| 36 |  |  |  |
| 37 |  |  |  |
| 38 |  |  |  |
| 39 |  |  |  |
| 40 |  |  |  |
| 41 |  |  |  |
| 42 |  |  |  |
| 43 |  |  |  |
| 44 |  |  |  |
| 45 |  |  |  |
| 46 |  |  |  |
| 47 |  |  |  |
| 48 |  |  |  |
| 49 |  |  |  |
| 50 |  |  |  |
| 51 |  |  |  |
| 52 | PG | Predicate group / Status Register |  |
| 53 | PC | Program counter |  |
| 54 | SC | Stack canary; a LOAD does CCHK |  |
| 55 | LC | Loop counter |  |
| 56 | LR0 | Subroutine link register #0; branch subroutine specific |  |
| 57 | LR1 | Subroutine link register #1 |  |
| 58 | LR2 | Subroutine link register #2 |  |
| 59 | LR3 | Subroutine link register #3 |  |
| 60 | GP1 | Global Pointer #1 |  |
| 61 | GP0 | Global Pointer #0 |  |
| 62 | FP | Frame Pointer |  |
| 63 | SP | Stack Pointer |  |
| 63 | ASP | Application Stack pointer |  |
| 63 | SSP | System Stack pointer |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | AC | Application Control Register |  |

### 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.

*The original Thor lumped these registers together in a code address register array.*

#### LRn – Link Registers

There are four registers in the Thor2023 architecture reserved for subroutine linkage. These registers are used to store the address of the calling instruction. They may be used to implement fast returns for several 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 may reload the program counter with an offset from the value contained in a link register. Typically, this is used to return to the next instruction.

#### PC – Program Counter

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

### LC - Loop Counter

The loop counter register is used in counted loops along the decrement and branch, [DBcc](#_DBcc_–_Decrement), instruction.

### PG -Predicate Register Group

There is a set of eight predicate registers which may be referenced individually or together in a group called the PG. A predicate register holds eight flags which reflect the status of operations. The compare and bit test instructions update predicate registers.

*The original Thor design had sixteen predicate registers. This was probably overkill. Thor2023 reduces the number of predicate registers to eight. Predicated instructions are handled differently. An instruction modifier indicates which instructions are predicated. This contrasts with the original Thor where every instruction had a predicate. Thor2023 eliminates the bytes wasted representing unused predicates. This helps to increase code density.*

Lumped in with the predicate register group is the machine status register which often needs to be saved in addition to the PG. In application mode only the low order eight bits of the status register may be read or updated. The remaining bits will read as zero and ignore writes.

**Predicate Register Group Register**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 95 64 | 63 56 | 55 48 | 47 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| SR | Cr7 | Cr6 | Cr5 | Cr4 | Cr3 | Cr2 | Cr1 | Cr0 |

**Predicate Register Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| lt | eq | inf | Nan | nf | zf | vf | cf |

### SR - Status Register

The processor status register holds bits controlling the overall operation of the processor. The status register is not accessible in user mode.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 24 | 23 21 | 20 16 | 15 |  | 13 12 | 11 | 10 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| CPL | ~ | ~ | T |  | OM |  | IPL |  |  | D | D | AR | | RT | |

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

T indicates that trace mode is active.

S indicates the processor is in supervisor mode.

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

RT specifies the return type for an [RTI](#_RTI_–_Return) instruction.

#### Decimal Mode

Setting the ‘D’ flag bit 5 in the SR register sets the processor in decimal operating mode. Arithmetic operations will use BCD numbers for both source and destination operands.

Decimal mode, ‘D’ flag bit 4, may also be applied to floating-point which will use decimal floating-point operations instead of binary.

### Special Purpose Registers

#### SC - Stack Canary

This special purpose register is available in the general register file as register 54. 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.

#### AV – Application Vector Table Address

This register holds the address of the applications vector table. The vector table must be 16-byte aligned.

|  |  |
| --- | --- |
| 63 4 | 3 0 |
| App Vector Table Address63..4 | 0 |

#### VB – Vector Base Register

The vector base register provides the location of the vector table. The vector table must be octa aligned. On reset the VBR is loaded with zero. There is a separate vector base register for each operating mode.

|  |  |  |
| --- | --- | --- |
| 63 3 | 2 | 1 0 |
| Vector Table Address63..3 | ~ | ~ |

# Operating Modes

The core operates in one of four basic modes: application/user mode, supervisor mode, hypervisor mode or machine mode. Machine mode is switched to when an interrupt or exception occurs, or when debugging is triggered. On power-up the core is running in machine mode. An RTI instruction must be executed to leave machine mode after power-up.

A subset of instructions is limited to machine mode.

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

Tags

|  |  |
| --- | --- |
| Tag |  |
| 0 | Untagged |
| 1 | Address Pointer – 13 bit size + 60 bit pointer   |  |  | | --- | --- | | Subtype |  | | 0 | Unused | | 1 | Return address | | 2 | Frame Pointer | | 3 | Pointer | | 4 to 7 | Unassigned | |  |  | |
| 2 | Integer – 76 bits |
| 3 | Float – 76 bits |
| 4 | String Descriptor – 16 bit length, 60 bit virtual address pointer |
| 5 | Character data, three 24-bit characters |
| 6 | Unassigned |
| 7 | Instruction (2 \* 38 bits) |

|  |  |  |
| --- | --- | --- |
| 79 | 78 76 | 75 0 |
| P | Tag3 | Data76 |

# Exceptions

## External Interrupts

There is little difference between an externally generated exception and an internally generated one. An externally caused exception will set the exception cause code for the currently fetched instruction.

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

## Effect on Machine Status

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

## Exception Stack

The status register, program counter, and predicate group register are pushed onto an internal stack when an exception occurs. This stack is at least 16 entries deep to allow for nested interrupts and multiply nested traps and exceptions.

Exception Table

|  |  |
| --- | --- |
| Vector | Usage |
| 0 | Reset value for system stack pointer |
| 1 | Reset value for program counter |
| 2 | Bus Error |
| 3 | Address Error |
| 4 | Unimplemented Instruction |
| 5 |  |
| 6 |  |
| 7 |  |
| 8 | Privilege Violation |
| 9 | Instruction trace |
| 10 |  |
| 11 | Stack Canary |
| 12 to 23 | reserved |
| 24 | Spurious interrupt |
| 25 | Auto vector #1 |
| 26 | Auto vector #2 |
| 27 | Auto vector #3 |
| 28 | Auto vector #4 |
| 29 | Auto vector #5 |
| 30 | Auto vector #6 |
| 31 | Auto vector #7 |
| 32 | Breakpoint (BRK) |
| 33 to 63 | Trap #1 to 31 |
|  | Applications Usage |
| 64 | Divide by zero |
| 65 | Overflow |
| 65 to 511 | Unassigned usage |
|  |  |

## Reset

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

The core begins executing instructions at address $00…00. All registers are in an undefined state.

## Precision

Exceptions in Thor2023 are precise. They are processed according to program order of the instructions. If an exception occurs during the execution of an instruction, then an exception field is set in the pipeline buffer. The exception is processed when the instruction commits which happens in program order. If the instruction was executed in a speculative fashion, then no exception processing will be invoked unless the instruction makes it to the commit stage.

# Instruction Set

## Overview

Thor was a variable length instruction set with instructions varying in length from one to eight bytes. Thor2023 is primarily a fixed length instruction with provision for additional instruction words used for constants. Reducing the variety of instruction sizes makes implementation of decoders more economical.

# Instruction Descriptions

### Major Opcode

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0x | 0  TRAP | 1  {R1} | 2  {R2} | 3  {CSR} | 4  ADD | 5  CMP | 6  MUL | 7  DIV |
|  | 8  AND, BIT | 9  OR | 10  XOR | 11 | 12  {FLT} | 13  {BIT} | 14  SHIFT | 15  {RET} |
| 1x | 16  LOAD | 17 | 18  STORE | 19  STORE # | 20  DFMA | 21  DFMS | 22  DFNMA | 23  DFNMS |
|  | 24  JSR, JMP | 25  LOADI | 26 | 27 | 28  Bcc | 29  DBcc | 30 | 31  PFX / NOP |

### {R1} Operations

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0x | 0  PRED | 1 | 2 | 3 | 4  NOT | 5  NEG | 6  ABS | 7 |
|  | 8  CARRY | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1x | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|  | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 2x |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
| 3x |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

### {R2} Operations

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0x | 0 | 1 | 2 | 3 | 4  ADD | 5  SUB | 6  MUL | 7  DIV |
|  | 8  AND | 9  OR | 10  EOR | 11 | 12 | 13 | 14 | 15 |
| 1x | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|  | 24  JMP / JSR | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 2x |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
| 3x |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

### {FLT} Operations

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0x | 0 | 1 | 2 | 3  FSCALE | 4  FADD | 5  FSUB | 6  FMUL | 7  FDIV |
|  | 8  FAND | 9  FOR | 10  FEOR | 11  FCMP | 12 | 13 | 14 | 15  FDIVR |
| 1x | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|  | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 2x | 32 | 33 | 34 | 35  FSCALE | 36  FADD | 37  FSUB | 38  FMUL | 39  FDIV |
|  | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47  FDIVR |
| 3x |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

Operand Swapping

Most instructions allow first and second source operands to be swapped. This is indicated by the swap ‘S’ bit in the instruction.

**Load and Store Address Modes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Address Mode** | **Format** | **M2** | **Reg5** |
| Indexed1 with displacement | Disp(Rn, Rn) | 0 | Rn |
| Post Update indexed with displacement | Disp(Rn,Rn+) | 1 | Rn |
| Pre Update indexed indirect with displacement | Disp(+Rn,Rn) | 2 | Rn |
| Immediate Eleven | Imm11 | 3 | nnnnn |
| Immediate Thirty-Two | Imm32 | 3 | nnnnn |
| Immediate Sixty-Four | Imm64 | 3 | nnnnn |

1For indexed modes, R0 refers to the program counter

**Operation Size**

|  |  |  |
| --- | --- | --- |
| **Operation Size** | **Suffix** | **Sz2** |
| Byte | .b | 00 |
| Wyde | .w | 01 |
| Tetra | .t | 10 |
| Octa | .o | 11 |

Operand Swap

|  |  |
| --- | --- |
| **Operand Order** | **S** |
| Normal | 0 |
| 1st and 2nd Swapped | 1 |

### Operand Sizes

Many instructions support four different operand sizes: byte, wyde, tetra and octa. The operand size is selected by suffixing the mnemonic with ‘b’ for byte, ‘w’ for wyde, ‘t’ for tetra and ‘o’ for octa.

|  |  |
| --- | --- |
| Sz2 | Operand Size |
| 00 | Byte |
| 01 | Wyde |
| 10 | Tetra |
| 11 | Octa |

## Arithmetic Operations

### ABS – Absolute Value

**Description:**

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

**Integer Instruction Format: R1**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 23 | 22 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| 03 | ~6 | 66 | Ra5 | Rt5 | Sz2 | 15 |

**Operation:**

If Ra < 0

Rt = -Ra

else

Rt = Ra

**Execution Units:** Integer ALU #0

**Clock Cycles: 1**

**Exceptions:** none

**Notes:**

### ADD - Addition

**Description:**

Add two source operands and place the sum in the target register. All registers are treated as integer registers. Arithmetic is signed twos-complement values unless the decimal mode flag is set in which case values are treated as BCD numbers. This instruction may be used with the [CARRY](#_CARRY) modifier to perform extended precision addition.

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

**Operation:**

Rt = Ra + Rb or Rt = Ra + Imm

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

**Instruction Formats:**

**ADD Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 31 | 3029 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 46 | ~3 | ~2 | 0 | Sb | Rb6 | Sa | Ra6 | St | Rt6 | Sz2 | 25 |

**Clock Cycles: 1**

**ADD Rt,Ra,Imm18**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| Imm17..7 | S | I6 | Imm5..0 | Sa | Ra6 | St | Rt6 | Sz2 | 45 |

**Clock Cycles: 1**

### AND – Bitwise And

**Description:**

Bitwise ‘and’ two source operands and place the result in the target register.

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

**Clock Cycles: 1**

**Operation:**

Rt = Ra & Rb or Rt = Ra & Imm

**Instruction Formats:**

**AND Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 31 | 3029 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 86 | ~3 | ~2 | 0 | Sb | Rb6 | Sa | Ra6 | St | Rt6 | Sz2 | 25 |

**Clock Cycles: 1**

**AND Rt,Ra,Imm18**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| Imm17..7 | S | I6 | Imm5..0 | Sa | Ra6 | St | Rt6 | Sz2 | 85 |

**Clock Cycles: 1**

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

**Exceptions:** none

**Notes:**

### BIT – Bitwise And, Discard Result

**Description:**

Bitwise ‘and’ two source operands and discard the result. Predicate flags are updated according to the result. All registers are integer registers. Arithmetic is signed twos-complement values.

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

**Flag Updates:**

Predicate Register Pt3 is always updated.

If the result is zero the equals flag is set otherwise it is cleared.

If the result is negative the negative flag is set, otherwise it is cleared.

Overflow flag is not affected.

Carry flag is not affected.

**Clock Cycles:** 1

**Operation:**

Ct = Ra & Rb

**Instruction Formats:**

**BIT Pt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 23 | 22 | 21 17 | 16 12 | 1110 | 9 7 | 6 5 | 4 0 |
| 108h9 | 0 | Rb5 | Ra5 | ~ | Pt3 | Sz2 | 25 |

**Clock Cycles: 1**

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

**Exceptions:** none

**Notes:**

### CMP - Comparison

**Description:**

Compare two source operands and place the result in the target predicate register. The result is a vector identifying the relationship between the two source operands as signed and unsigned integers and as floating-point values. The operation performed is a subtract with the result discarded.

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

**Condition Register Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| lt | eq | inf | Nan | nf | zf | vf | cf |

**Operation:**

Ct = Ra ? Rb or Ct = Ra ? Imm or Ct = Imm ? Ra

**Clock Cycles:** 1

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

**Exceptions:** none

**Notes:**

**Instruction Formats:**

**CMP Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 31 | 30 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 56 | ~3 | ~2 | 0 | Sb | Rb6 | Sa | Ra6 | St | Rt6 | Sz2 | 25 |

**Clock Cycles: 1**

**CMP Rt,Ra,Imm13**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| Imm17..7 | S | I6 | Imm5..0 | Sa | Ra6 | St | Rt6 | Sz2 | 55 |

**Clock Cycles: 1**

### CSR – Control and Special Registers Operations

**Description:**

Perform an operation on a CSR. A CSR field value of zero indicates to use the next instruction word as the CSR register number.

|  |  |  |
| --- | --- | --- |
| **Operation** | **Op2** |  |
| Or to CSR (set bits) | 0 |  |
| And complement to CSR (clear bits) | 1 |  |
| Exclusive Or to CSR (flip bits) | 2 |  |

**Supported Operand Sizes:** .o

|  |  |  |
| --- | --- | --- |
| **Regno** |  |  |
| 000 | {ext} | Extended CSR number |
| 001 | ccr | Condition codes |
| 002 | sr | Status register (privileged) |
| 120 | Tick | Tick count (read only) |
| 121 | Coreno | Core number ( read only) (privileged) |
| 127 |  |  |

**Instruction Formats:**

**OR Rt, Ra, CSR**

**ANDC Rt, Ra, CSR**

**EOR Rt, Ra, CSR**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 23 | 22 | 21 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| 0 | ~ | CSR11..5 | 0 | CSR4..0 | Ra5 | Rt5 | Op2 | 35 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 23 | 22 | 21 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| 1 | I | CSR11..5 | 0 | CSR4..0 | Imm4..0 | Rt5 | Op2 | 35 |

### DIVS – Signed Division

**Description:**

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

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

**Operation:**

Rt = Ra / Rb or Rt = Ra / Imm or Rt = Imm / Ra

**Instruction Formats:**

**DIVS Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 23 | 22 | 21 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| 03 | 76 | 0 | Rb5 | Ra5 | Rt5 | Sz2 | 15 |

**Clock Cycles: 70**

**DIVS Rt,Ra,Imm14**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 23 | 22 | 21 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| Imm13..5 | S | Imm4..0 | Ra5 | Rt5 | Sz2 | 75 |

**Clock Cycles: 70**

**DIVS Rt,Ra,Imm32**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 2523 | 22 | 21 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| 20h6 | 03 | S | ~5 | Ra5 | Rt5 | Sz2 | 75 |
| Immediate32 | | | | | | | |

**Clock Cycles: 70**

**DIVS Rt,Ra,Imm64**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 2523 | 22 | 21 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| 21h6 | 03 | S | ~5 | Ra5 | Rt5 | Sz2 | 75 |
| Immediate31..0 | | | | | | | |
| Immediate63..32 | | | | | | | |

**Clock Cycles: 70**

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

**Exceptions:** none

**Notes:**

### DIVU – Unsigned Division

**Description:**

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

Ten-bit immediate mode is not available for this instruction.

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

**Operation:**

Rt = Ra / Rb or Rt = Ra / Imm or Rt = Imm / Ra

**Instruction Formats:**

**DIVU Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 23 | 22 | 21 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| 13 | 76 | 0 | Rb5 | Ra5 | Rt5 | Sz2 | 15 |

**Clock Cycles: 70**

**DIVS Rt,Ra,Imm32**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 2523 | 22 | 21 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| 20h6 | 13 | S | ~5 | Ra5 | Rt5 | Sz2 | 75 |
| Immediate32 | | | | | | | |

**Clock Cycles: 70**

**DIVS Rt,Ra,Imm64**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 2523 | 22 | 21 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| 21h6 | 13 | S | ~5 | Ra5 | Rt5 | Sz2 | 75 |
| Immediate31..0 | | | | | | | |
| Immediate63..32 | | | | | | | |

**Clock Cycles: 70**

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

**Exceptions:** none

**Notes:**

### EOR – Bitwise Exclusive Or

**Description:**

Bitwise exclusive ‘or’ two source operands and place the sum in the target register. All registers are integer registers. Arithmetic is signed twos-complement values.

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

**Operation:**

Rt = Ra ^ Rb or Rt = Ra ^ Imm

**Instruction Formats:**

**EOR Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 31 | 3029 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 106 | ~3 | ~2 | 0 | Sb | Rb6 | Sa | Ra6 | 0 | Rt6 | Sz2 | 25 |

**Clock Cycles: 1**

**EOR Rt,Ra,Imm18**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| Imm17..7 | S | I6 | Imm5..0 | Sa | Ra6 | 0 | Rt6 | Sz2 | 105 |

**Clock Cycles: 1**

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

**Exceptions:** none

**Notes:**

### ENOR – Bitwise Exclusive Nor

**Description:**

Bitwise exclusive ‘nor’ two source operands and place the result in the target register.

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

**Operation:**

Rt = ~(Ra ^ Rb) or Rt = ~(Ra ^ Imm)

**Instruction Formats:**

**ENOR Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 31 | 3029 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 106 | ~3 | ~2 | 0 | Sb | Rb6 | Sa | Ra6 | 1 | Rt6 | Sz2 | 25 |

**Clock Cycles: 1**

**ENOR Rt,Ra,Imm18**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| Imm17..7 | S | I6 | Imm5..0 | Sa | Ra6 | 1 | Rt6 | Sz2 | 105 |

**Clock Cycles: 1**

**Clock Cycles: 1**

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

**Exceptions:** none

**Notes:**

### PFX – Constant Postfix

**Description:**

The PFX instruction postfix is used to build large constants for use in the preceding instruction as the immediate constant for the instruction. There are three postfix instructions which extend the constant from different bit locations. They should be used in the order PFX0, PFX1, then PFX2. A postfix may be omitted if the omitted bits match what would be included.

Postfixes are normally caught at the decode stage and do not progress further in the pipeline. They are treated as a NOP instruction.

**Supported Operand Sizes:** N/A

**Instruction Format:**

This format extends the constant from bit 18 with the 32 bits specified in the instruction and sign extends the value to the width of the constant prefix buffer.

|  |  |  |
| --- | --- | --- |
| 39 8 | 7 5 | 4 0 |
| Immediate32 | 03 | 315 |

**Instruction Format:**

This format extends the previous constant value by 32 bits beginning at bit 50 and sign extends the value to the width of the machine.

|  |  |  |
| --- | --- | --- |
| 39 8 | 7 5 | 4 0 |
| Immediate32 | 13 | 315 |

**Instruction Format:**

This format extends the previous constant value by 32 bits beginning at bit 82 and sign extends the value to the width of the machine. Only the low order 14 bits of this value are used as values are limited to 96 bits.

|  |  |  |
| --- | --- | --- |
| 39 8 | 7 5 | 4 0 |
| Immediate32 | 23 | 315 |

### MULS – Multiply Signed

**Description:**

Multiply two source operands and place the sum in the target register. All registers are treated as integer registers. Arithmetic is signed twos-complement values. The ‘S’ flag indicates to perform an unsigned multiply.

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

**Operation:**

Rt = Ra \* Rb or Rt = Ra \* Imm

**Instruction Formats:**

**MULS Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 31 | 3029 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 66 | ~3 | ~2 | 0 | Sb | Rb6 | Sa | Ra6 | St | Rt6 | Sz2 | 25 |

**Clock Cycles: 12**

**MULS Rt,Ra,Imm18**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| Imm17..7 | 0 | I6 | Imm5..0 | Sa | Ra6 | St | Rt6 | Sz2 | 65 |

**Clock Cycles: 12**

**Clock Cycles:** 12

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

**Exceptions:** none

**Notes:**

### MULU – Unsigned Multiplication

**Description:**

Multiply two source operands and place the product in the target register. All registers are treated as integer registers. Arithmetic is signed twos-complement values. The ‘S’ flag indicates to perform an unsigned multiply. Unsigned multiply can be used during index calculations.

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

**Operation:**

Rt = Ra \* Rb or Rt = Ra \* Imm

**Instruction Formats:**

**MULU Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 31 | 3029 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 66 | ~3 | ~2 | 1 | Sb | Rb6 | Sa | Ra6 | St | Rt6 | Sz2 | 25 |

**Clock Cycles: 12**

**MULU Rt,Ra,Imm18**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| Imm17..7 | 1 | I6 | Imm5..0 | Sa | Ra6 | St | Rt6 | Sz2 | 65 |

**Clock Cycles: 12**

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

**Exceptions:** none

**Notes:**

### NAND – Bitwise And and Invert

**Description:**

Bitwise ‘nand’ two source operands and place the result in the target register.

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

**Clock Cycles: 1**

**Operation:**

Rt = ~(Ra & Rb)

**Instruction Formats:**

**NAND Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 25 | 24 | 23 | 22 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| 87 | 1 | Sb | Rb5 | Sa | Ra5 | Rt5 | Sz2 | 25 |

**Clock Cycles: 1**

**NAND Rt,Ra,Imm13**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 25 | 24 | 23 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| Imm12..6 | 1 | Imm5..0 | Sa | Ra5 | Rt5 | Sz2 | 85 |

**Clock Cycles: 1**

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

**Exceptions:** none

**Notes:**

### NOR – Bitwise Or and Invert

**Description:**

Bitwise ‘or’ two source operands invert the result and place the result in the target register. All registers are integer registers.

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

**Operation:**

Rt = ~(Ra | Rb)

**Instruction Formats:**

**NOR Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 31 | 3029 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 96 | ~3 | ~2 | 0 | Sb | Rb6 | Sa | Ra6 | 1 | Rt6 | Sz2 | 25 |

**Clock Cycles: 1**

**NOR Rt,Ra,Imm18**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| Imm17..7 | S | I6 | Imm5..0 | Sa | Ra6 | 1 | Rt6 | Sz2 | 95 |

**Clock Cycles: 1**

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

**Exceptions:** none

**Notes:**

### OR – Bitwise Or

**Description:**

Bitwise ‘or’ two source operands and place the sum in the target register. All registers are integer registers. Arithmetic is signed twos-complement values.

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

**Operation:**

Rt = Ra | Rb or Rt = Ra | Imm

**Instruction Formats:**

**OR Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 31 | 3029 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 96 | ~3 | ~2 | 0 | Sb | Rb6 | Sa | Ra6 | 0 | Rt6 | Sz2 | 25 |

**Clock Cycles: 1**

**OR Rt,Ra,Imm18**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| Imm17..7 | S | I6 | Imm5..0 | Sa | Ra6 | 0 | Rt6 | Sz2 | 95 |

**Clock Cycles: 1**

**Clock Cycles:** 2

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

**Exceptions:** none

**Notes:**

## Floating-Point Operations

### Precision

Floating point operations are always performed at the greatest precision available. Lower precision formats are available for storage.

For decimal floating-point three storage formats are supported. 88-bit extended precision, 64-bit double precision, and 32-bit single precision values.

### Representations

#### Long Integer

|  |  |
| --- | --- |
| 95 6 | 5 0 |
| 90 bits | 18 |

#### Integer Pair

|  |  |  |
| --- | --- | --- |
| 95 56 | 47 6 | 5 0 |
| 45 bits | 45 bits | 28 |

#### Short Integer Quad

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 93 72 | 71 50 | 49 28 | 27 6 | 5 0 |
| 22 bits | 22 bits | 22 bits | 22 bits | 38 |

#### Binary Floats

The core uses an 88-bit extended precision binary floating-point representation.

*96-bit values are more compact than 128-bit ones which reduces the amount of hardware required and data being transferred. They have enough significant digits for a wide variety of applications. 64-bit values are not sufficient for some applications. The question then is how much larger of a representation to use. 80-bits is popular, offering about 19 significant digits which is good for a wide variety of applications.*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 95 | 94 | 93 | 92 77 | 76 6 | 5 0 |
| ~ | ~ | S | Exponent16 | Significand71 | 4 |

Double Precision

|  |  |  |  |
| --- | --- | --- | --- |
| 69 | 68 58 | 57 6 | 5 0 |
| S | Exponent11 | Significand52 | 5 |

Single Precision Pair

|  |  |  |  |
| --- | --- | --- | --- |
| 37 | 36 29 | 28 6 | 5 0 |
| S | Exponent8 | Significand23 | 6 |

#### Decimal Floats

The core uses a 88-bit densely packed decimal triple precision floating-point representation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 95 | 94 90 | 89 82 | 81 8 | 7 0 |
| S | Combo5 | Exponent8 | Significand70 | 8 |

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

64-bit double precision decimal floating point:

|  |  |  |  |
| --- | --- | --- | --- |
| 63 | 62 58 | 57 50 | 49 0 |
| S | Combo5 | Exponent8 | Significand50 |

The significand stores 16 DPD digits. One whole digit before the decimal point.

32-bit single precision decimal floating point:

|  |  |  |  |
| --- | --- | --- | --- |
| 31 | 30 26 | 25 20 | 19 0 |
| S | Combo5 | Exponent6 | Significand20 |

The significand store 7 DPD digits. One whole digit before the decimal point.

#### Strings

|  |  |  |  |
| --- | --- | --- | --- |
| 9594 | 93 70 | 69 6 | 5 0 |
| Ty | Length24 | Pointer64 | 16 |

UTF8 Chars

|  |  |
| --- | --- |
| 95 6 | 5 0 |
| 11 characters | 17 |

UTF22 Chars

|  |  |
| --- | --- |
| 95 8 | 5 0 |
| 4 characters | 18 |

### Rounding Modes

#### Binary Float Rounding Modes

|  |  |
| --- | --- |
| R,Rm2 | 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

|  |  |
| --- | --- |
| R,Rm2 | 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 |

### FABS – Absolute Value

**Description:**

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

**Integer Instruction Format: R1**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 23 | 22 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| 23 | ~6 | 66 | FPa5 | FPt5 | Sz2 | 15 |

**Operation:**

FPt = Abs(FPa)

**Execution Units:** FPU #0

**Clock Cycles: 1**

**Exceptions:** none

**Notes:**

### FADD –Float Addition

**Description:**

Add two source operands and place the sum in the target register. All registers values are treated as 88-bit floating-point values.

**Supported Operand Sizes:**

**Operation:**

Rt = Ra + Rb or Rt = Ra + Imm

**Clock Cycles:** 8

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

**Exceptions:** none

**Notes:**

**Instruction Formats:**

**FADD Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 | 23 | 22 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| 46 | Rm | 0 | Sb | Rb5 | Sa | Ra5 | Rt5 | Rm2 | 125 |

**FADD Rt,Ra,Imm32**

Convert 32-bit single precision value to 88-bits then add

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 | 23 | 22 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| 366 | Rm | 0 | Sb | 15 | Sa | Ra5 | Rt5 | Rm2 | 125 |
| Imm32 | | | | | | | | | |

**FADD Rt,Ra,Imm64**

Convert 64-bit single precision value to 88-bits then add

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 | 23 | 22 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| 366 | Rm | 0 | Sb | 25 | Sa | Ra5 | Rt5 | Rm2 | 125 |
| Imm31..0 | | | | | | | | | |
| Imm64..32 | | | | | | | | | |

### FCMP - Comparison

**Description:**

This is an alternate mnemonic for the [CMP](#_CMP_-_Comparison) instruction. Compare two source operands and place the result in the target predicate register. The result is a vector identifying the relationship between the two source operands as floating-point values. This instruction may compare against lower precision immediate values to conserve code space.

**Supported Operand Sizes:**

**Operation:**

Pt = Ra ? Rb or Pt = Ra ? Imm or Pt = Imm ? Ra

**Clock Cycles:** 1

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

**Exceptions:** none

**Notes:**

**Instruction Formats:**

### FDIV –Float Division

**Description:**

Divide two source operands and place the quotient in the target register. All registers values are treated as 96-bit floating-point values.

**Supported Operand Sizes:**

**Operation:**

Rt = Ra / Rb or Rt = Ra / Imm or Rt = Imm / Ra

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

**Instruction Formats:**

**FDIV Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 76 | Rm | Rb6 | Ra6 | Rt6 | Rm2 | 125 |

**FDIV Rt,Ra,Imm32**

Convert 32-bit single precision value to 96-bits then add

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 396 | Rm | 15 | Ra6 | Rt6 | Rm2 | 125 |
| Immediate31..0 | | | | | | |

**FDIV Rt,Ra,Imm64**

Convert 64-bit double precision value to 96-bits then add

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 396 | Rm | 25 | Ra5 | Rt5 | Rm2 | 125 |
| Immediate31..0 | | | | | | |
| Immediate63..32 | | | | | | |

**FDIV Rt,Ra,Imm96**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 396 | Rm | 35 | Ra5 | Rt5 | Rm2 | 125 |
| Immediate31..0 | | | | | | |
| Immediate63..32 | | | | | | |
| Immediate95..64 | | | | | | |

**FDIV Rt, Ra, Rb – Register direct – Rt = Rb / Ra**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 156 | Rm | Rb6 | Ra6 | Rt6 | Rm2 | 125 |

**FDIV Rt,Ra,Imm32 – Rt = Imm32 / Ra**

Convert 32-bit single precision value to 96-bits then add

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 476 | Rm | 15 | Ra6 | Rt6 | Rm2 | 125 |
| Immediate31..0 | | | | | | |

**FDIV Rt,Ra,Imm64 – Rt = Imm64 / Ra**

Convert 64-bit double precision value to 96-bits then add

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 476 | Rm | 25 | Ra6 | Rt6 | Rm2 | 125 |
| Immediate31..0 | | | | | | |
| Immediate63..32 | | | | | | |

**FDIV Rt,Ra,Imm96 – Rt = Imm96 / Ra**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 476 | Rm | 35 | Ra6 | Rt6 | Rm2 | 125 |
| Immediate31..0 | | | | | | |
| Immediate63..32 | | | | | | |
| Immediate95..64 | | | | | | |

### FMUL –Float Multiplication

**Description:**

Multiply two source operands and place the product in the target register. All registers values are treated as 96-bit floating-point values.

**Supported Operand Sizes:**

**Operation:**

Rt = Ra \* Rb or Rt = Ra \* Imm

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

**Instruction Formats:**

**FMUL Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 66 | Rm | Rb6 | Ra6 | Rt6 | Rm2 | 125 |

**FMUL Rt,Ra,Imm32**

Convert 32-bit single precision value to 96-bits then add

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 386 | Rm | 15 | Ra6 | Rt6 | Rm2 | 125 |
| Immediate31..0 | | | | | | |

**FMUL Rt,Ra,Imm64**

Convert 64-bit double precision value to 96-bits then add

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 386 | Rm | 25 | Ra6 | Rt6 | Rm2 | 125 |
| Immediate31..0 | | | | | | |
| Immediate63..32 | | | | | | |

**FMUL Rt,Ra,Imm96**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 386 | Rm | 35 | Ra6 | Rt6 | Rm2 | 125 |
| Immediate31..0 | | | | | | |
| Immediate63..32 | | | | | | |
| Immediate95..64 | | | | | | |

### FNEG – Negate Value

**Description:**

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

**Integer Instruction Format: R1**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 23 | 22 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| 23 | ~6 | 56 | FPa5 | FPt5 | Sz2 | 15 |

**Operation:**

FPt = -FPa

**Execution Units:** FPU #0

**Clock Cycles: 1**

**Exceptions:** none

**Notes:**

### FSCALE –Scale Exponent

**Description:**

Add the source operand to the exponent.

**Supported Operand Sizes:**

**Operation:**

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

**Instruction Formats:**

**FSCALE Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 36 | Rm | Rb6 | Ra6 | Rt6 | Rm2 | 125 |

**FSCALE Rt,Ra,Imm16**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | | 12 7 | 6 5 | 4 0 |
| 356 | Rm | 05 | Ra6 | | Rt6 | Rm2 | 125 |
| ~16 | | | | Immediate15..0 | | | |

### FSUB –Float Subtraction

**Description:**

Subtract two source operands and place the difference in the target register. All registers values are treated as 96-bit floating-point values.

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

**Operation:**

Rt = Ra - Rb or Rt = Ra – Imm or Rt = Imm - Ra

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

**Instruction Formats:**

**FSUB Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 56 | Rm | Rb6 | Ra6 | Rt6 | Rm2 | 125 |

**FSUB Rt,Ra,Imm32**

Convert 32-bit single precision value to 96-bits then add

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 376 | Rm | 15 | Ra6 | Rt6 | Rm2 | 125 |
| Immediate31..0 | | | | | | |

**FSUB Rt,Ra,Imm64**

Convert 64-bit double precision value to 96-bits then add

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 376 | Rm | 25 | Ra6 | Rt6 | Rm2 | 125 |
| Immediate31..0 | | | | | | |
| Immediate63..32 | | | | | | |

**FSUB Rt,Ra,Imm96**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 24 19 | 18 13 | 12 7 | 6 5 | 4 0 |
| 376 | Rm | 35 | Ra6 | Rt6 | Rm2 | 125 |
| Immediate31..0 | | | | | | |
| Immediate63..32 | | | | | | |
| Immediate95..64 | | | | | | |

## Bit Manipulation Operations

### BCLR – Clear Bit

**Description:**

A bit in the source operand is cleared and the result placed in the target register. The specified bit to clear is modulo the operand size.

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

**Flag Updates:** none

**Operation:**

zf = Ra[Rb] or zf = Ra[Imm]

Rt = Ra &~bit Rb or Ra = Ra &~bit imm

**Instruction Formats:**

**BCLR Ct, Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 2524 | 23 | 22 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| ~3 | 03 | ~2 | 0 | Rb5 | 01 | Ra5 | Rt5 | Sz2 | 135 |

**Clock Cycles: 1**

**BCLR Ct, Rt, Ra,Imm7**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 24 | 23 | 22 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| ~3 | 03 | Im6..5 | 0 | Imm4..0 | 11 | Ra5 | Rt5 | Sz2 | 135 |

**Clock Cycles: 1**

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

### BCHG – Change Bit

**Description:**

A bit in the source operand is changed and placed in the target register. The specified bit to change is modulo the operand size.

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

**Flag Updates: none**

**Operation:**

zf = Ra[Rb] or zf = Ra[Imm]

Rt[Rb] = ~Ra[Rb] or Rt[Imm] = ~Ra[Imm]

**Instruction Formats:**

**BCLR Ct, Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 2524 | 23 | 22 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| ~3 | 23 | ~2 | 0 | Rb5 | 01 | Ra5 | Rt5 | Sz2 | 135 |

**Clock Cycles: 4**

**BCLR Ct, Rt, Ra,Imm7**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 24 | 23 | 22 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| ~3 | 23 | Im6..5 | 0 | Imm4..0 | 11 | Ra5 | Rt5 | Sz2 | 135 |

**Clock Cycles: 4**

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

### BPCHG – Change Bit Pair

**Description:**

A bit pair in the source operand is changed and placed in the target register. The pair is exclusively or’d with 11b. There are four bit-pairs per byte indicated as pair #0 to #3. The bit pair specified is taken modulo the operand size.

|  |  |
| --- | --- |
| Bit Pair Value | Updated Value |
| 00 | 11 |
| 01 | 10 |
| 10 | 01 |
| 11 | 00 |

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

**Flag Updates:** none

**Operation:**

zf = Ra[Rb] or zf = Ra[Imm]

Rt[Rb] = ~Ra[Rb] or Rt[Imm] = ~Ra[Imm]

**Instruction Formats:**

**BCLR Ct, Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 2524 | 23 | 22 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| ~3 | 63 | ~2 | 0 | Rb5 | 01 | Ra5 | Rt5 | Sz2 | 135 |

**Clock Cycles: 1**

**BCLR Ct, Rt, Ra,Imm7**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 24 | 23 | 22 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| ~3 | 63 | Im6..5 | 0 | Imm4..0 | 11 | Ra5 | Rt5 | Sz2 | 135 |

**Clock Cycles: 1**

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

### BPCLR – Clear Bit Pair

**Description:**

A pair of bits in the source operand is cleared and the result placed in the target register.

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

**Operation:**

Rt = Ra &~bit Rb or Ra = Ra &~bit imm

**Instruction Formats:**

**BPCLR Ct, Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 2524 | 23 | 22 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| ~3 | 43 | ~2 | 0 | Rb5 | 01 | Ra5 | Rt5 | Sz2 | 135 |

**Clock Cycles: 4**

**BPCLR Ct, Rt, Ra,Imm7**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 24 | 23 | 22 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| ~3 | 43 | Im6..5 | 0 | Imm4..0 | 11 | Ra5 | Rt5 | Sz2 | 135 |

**Clock Cycles: 4**

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

### BPTST – Test Bit Pair

**Description:**

Test a bit pair in the source operand and place the bit status in the flags of the predicate register. The bit tested is modulo the operation size. All combinations of bit pair value may be detected via flags.

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

**Flag Updates:**

Predicate Register Pt3 is always updated.

|  |  |
| --- | --- |
| Bit Pair Value | Flag Setting |
| 00 | Zero flag is set, cf, nf, and vf are cleared |
| 01 | Carry flag is set, zf, nf, and vf are cleared |
| 10 | Negative flag is set, zf, cf, and vf are cleared |
| 11 | Overflow flags is set, nf, zf and cf are cleared |

If the bit pair is zero the zero flag is set otherwise it is cleared.

If the bit pair is

The negative flag is set to the value of the high order bit of the pair.

The overflow flag is set to the exclusive or of the two bits in the pair.

**Operation:**

**Instruction Formats:**

**BPTST Pt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 2524 | 23 | 22 18 | 17 | 16 12 | 1110 | 9 7 | 6 5 | 4 0 |
| ~3 | 73 | ~2 | 0 | Rb5 | 01 | Ra5 | ~2 | Pt3 | Sz2 | 135 |

**Clock Cycles: 4**

**BPTST Pt,Ra,Imm7**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 24 | 23 | 22 18 | 17 | 16 12 | 1110 | 9 7 | 6 5 | 4 0 |
| ~3 | 73 | Im6..5 | 0 | Imm4..0 | 11 | Ra5 | ~2 | Pt3 | Sz2 | 135 |

**Clock Cycles: 4**

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

### BSET – Set Bit

**Description:**

A bit in the source operand is set and placed in the target register.

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

**Operation:**

zf = Ra[Rb] or zf = Ra[Imm]

Rt = Ra | bit Rb or Rt = Ra or Bit[Imm]

**Instruction Formats:**

**BSET Ct, Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 2524 | 23 | 22 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| ~3 | 13 | ~2 | 0 | Rb5 | 01 | Ra5 | Rt5 | Sz2 | 135 |

**Clock Cycles: 1**

**BSET Ct, Rt, Ra,Imm7**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 24 | 23 | 22 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| ~3 | 13 | Im6..5 | 0 | Imm4..0 | 11 | Ra5 | Rt5 | Sz2 | 135 |

**Clock Cycles: 1**

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

### BTST – Test Bit

**Description:**

Test a bit in the source operand and place the bit status in the zero flag of the predicate register. The bit tested is modulo the operation size.

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

**Flag Updates:**

Predicate Register Pt3 is always updated.

If the bit is zero the zero flag is set otherwise it is cleared.

**Operation:**

zf = Ra[Rb] or zf = Ra[Imm]

**Instruction Formats:**

**BTST Pt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 2524 | 23 | 22 18 | 17 | 16 12 | 1110 | 9 7 | 6 5 | 4 0 |
| ~3 | 33 | ~2 | 0 | Rb5 | 01 | Ra5 | ~2 | Pt3 | Sz2 | 135 |

**Clock Cycles: 4**

**BTST Pt,Ra,Imm7**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 24 | 23 | 22 18 | 17 | 16 12 | 1110 | 9 7 | 6 5 | 4 0 |
| ~3 | 33 | Im6..5 | 0 | Imm4..0 | 11 | Ra5 | ~2 | Pt3 | Sz2 | 135 |

**Clock Cycles: 4**

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

## Shift and Rotate Operations

### ASL – Arithmetic Shift Left

**Description:**

Shift the first source operand to the left by the number of bits specified by the second source operand and place the result in the target register. All registers are integer registers. Arithmetic is signed twos-complement values. The least significant bit is filled with the value of ‘N’ specified in the instruction.

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

**Operation:**

Rt = Ra << Rb or Rt = Ra << Imm

**Instruction Formats:**

**ASL Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 30 | 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 06 | ~4 | N | 0 | Sb | Rb6 | Sa | Ra6 | St | Rt6 | Sz2 | 145 |

**Clock Cycles: 1**

**ASL Rt,Ra,Imm7**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 30 | 29 | 28 | 27 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 326 | ~4 | N | 0 | Imm6..0 | Sa | Ra6 | St | Rt6 | Sz2 | 145 |

**Clock Cycles: 1**

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

### ASR – Arithmetic Shift Right

**Description:**

Shift the first source operand to the right, preserving the sign bit, by the number of bits specified by the second source operand and place the result in the target register. All registers are integer registers. Arithmetic is signed twos-complement values.

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

**Operation:**

Rt = Ra >> Rb or Rt = Ra >> Imm

**Instruction Formats:**

**ASR Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 30 | 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 16 | ~4 | N | 0 | Sb | Rb6 | Sa | Ra6 | St | Rt6 | Sz2 | 145 |

**Clock Cycles: 1**

**ASR Rt,Ra,Imm7**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 30 | 29 | 28 | 27 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 336 | ~4 | N | 0 | Imm6..0 | Sa | Ra6 | St | Rt6 | Sz2 | 145 |

**Clock Cycles: 1**

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

### SBX – Sign Bit Extend

**Description:**

Sign extend a value beginning at a specified bit to the width of the register and place the result in the target register. All registers are integer registers.

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

**Operation:**

**Instruction Formats:**

**SXB Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 30 | 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 96 | ~4 | N | 0 | Sb | Rb6 | Sa | Ra6 | St | Rt6 | Sz2 | 145 |

**Clock Cycles: 1**

**SXB Rt,Ra,Imm7**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 30 | 29 | 28 | 27 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 416 | ~4 | N | 0 | Imm6..0 | Sa | Ra6 | St | Rt6 | Sz2 | 145 |

**Clock Cycles: 1**

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

### LSL – Logical Shift Left

**Description:**

Shift the first source operand to the left by the number of bits specified by the second source operand and place the result in the target register. All registers are integer registers. Arithmetic is signed twos-complement values. Fill the least significant bit with the value specified by ‘N’ in the instruction.

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

**Operation:**

Rt = Ra << Rb or Rt = Ra << Imm

**Instruction Formats:**

**LSL Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 30 | 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 26 | ~4 | N | 0 | Sb | Rb6 | Sa | Ra6 | St | Rt6 | Sz2 | 145 |

**Clock Cycles: 1**

**LSL Rt,Ra,Imm7**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 30 | 29 | 28 | 27 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 346 | ~4 | N | 0 | Imm6..0 | Sa | Ra6 | St | Rt6 | Sz2 | 145 |

**Clock Cycles: 1**

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

### LSR – Logical Shift Right

**Description:**

Shift the first source operand to the right by the number of bits specified by the second source operand and place the result in the target register. All registers are integer registers. Arithmetic is signed twos-complement values. Fill the least significant bit with the value specified by ‘N’ in the instruction.

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

**Operation:**

Rt = Ra >> Rb or Rt = Ra >> Imm

**Instruction Formats:**

**LSR Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 30 | 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 36 | ~4 | N | 0 | Sb | Rb6 | Sa | Ra6 | St | Rt6 | Sz2 | 145 |

**Clock Cycles: 1**

**LSR Rt,Ra,Imm7**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 30 | 29 | 28 | 27 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 356 | ~4 | N | 0 | Imm6..0 | Sa | Ra6 | St | Rt6 | Sz2 | 145 |

**Clock Cycles: 1**

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

### ROL – Rotate Left

**Description:**

Rotate the first source operand to the left by the number of bits specified by the second source operand and place the result in the target register. All registers are integer registers. Arithmetic is signed twos-complement values. The least significant bit is set to the value of the most significant bit exclusively or’d with the value ‘N’ from the instruction.

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

**Operation:**

Rt = Ra << Rb or Rt = Ra << Imm

**Instruction Formats:**

**ROL Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 30 | 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 46 | ~4 | N | 0 | Sb | Rb6 | Sa | Ra6 | St | Rt6 | Sz2 | 145 |

**Clock Cycles: 1**

**ROL Rt,Ra,Imm7**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 30 | 29 | 28 | 27 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 366 | ~4 | N | 0 | Imm6..0 | Sa | Ra6 | St | Rt6 | Sz2 | 145 |

**Clock Cycles: 1**

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

### ROR – Rotate Right

**Description:**

Rotate the first source operand through the carry to the right by the number of bits specified by the second source operand and place the result in the target register. All registers are integer registers. Arithmetic is signed twos-complement values. The most significant bit is set to the value of the least significant bit exclusively or’d with the value ‘N’ from the instruction.

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

**Operation:**

Rt = Ra >> Rb or Rt = Ra >> Imm

**Instruction Formats:**

**ROR Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 30 | 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 56 | ~4 | N | 0 | Sb | Rb6 | Sa | Ra6 | St | Rt6 | Sz2 | 145 |

**Clock Cycles: 1**

**ROR Rt,Ra,Imm7**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 30 | 29 | 28 | 27 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 376 | ~4 | N | 0 | Imm6..0 | Sa | Ra6 | St | Rt6 | Sz2 | 145 |

**Clock Cycles: 1**

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

### SWAP – Swap register halves

**Description:**

SWAP.T: The bits 0 to 31 and 32 to 63 are swapped.

SWAP.W: The bits 0 to 15 and 16 to 31 are swapped.

SWAP.B: The bits 0 to 7 and 8 to 15 are swapped.

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

**Instruction Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 24 | 23 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| ~3 | ~5 | 166 | 01 | Ra5 | Rt5 | Sz2 | 15 |

### ZBX – Zero Bit Extend

**Description:**

Zero extend a value beginning at a specified bit to the width of the register and place the result in the target register. All registers are integer registers.

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

**Operation:**

Rt = Zero Extend(Ra)

**Instruction Formats:**

**ZXB Rt, Ra, Rb – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 30 | 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 86 | ~4 | N | 0 | Sb | Rb6 | Sa | Ra6 | St | Rt6 | Sz2 | 145 |

**Clock Cycles: 1**

**ZXB Rt,Ra,Imm7**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 30 | 29 | 28 | 27 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 406 | ~4 | N | 0 | Imm6..0 | Sa | Ra6 | St | Rt6 | Sz2 | 145 |

**Clock Cycles: 1**

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

## Flow Control Instructions

### Bcc – Conditional Branch

Bcc Pn, label

**Description:**

Branch if the predicate condition is met. The displacement is relative to the address of the branch instruction. The branch range is +/- 64MB.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 39 13 | 12 10 | 9 5 | 4 0 |
| Disp26..0 | Pn3 | Cond5 | 285 |

|  |  |  |
| --- | --- | --- |
| **Cond6** | **Mnemonic** | **Test** |
| 0 | RA | 1 |
| 1 | SR | 0 |
| 2 | HI | !cf & !zf |
| 3 | LS | cf | zf |
| 4 | CC / HS | !cf |
| 5 | CS / LO | cf |
| 6 | NE | !zf |
| 7 | EQ | zf |
| 8 | VC | !vf |
| 9 | VS | vf |
| A | PL | !nf |
| B | MI | nf |
| C | GE | (nf & vf) | (!nf & !vf) |
| D | LT | (nf & !vf) | (!nf & vf) |
| E | GT | (nf & vf & !zf) | (!nf & !vf & zf) |
| F | LE | zf | (nf & !vf) | (!nf & vf) |
| 10 | - |  |
| 11 | - |  |
| 12 | - |  |
| 13 | - |  |
| 14 | - |  |
| 15 | - |  |
| 16 | - |  |
| 17 | - |  |
| 18 | - |  |
| 19 | - |  |
| 1A | - |  |
| 1B | - |  |
| 1C | - |  |
| 1D | - |  |
| 1E | - |  |
| 1F | - |  |

**Clock Cycles: 4**

### BRA – Unconditional Branch

**Description:**

Unconditionally branch to a new program address. The displacement is relative to the address of the branch instruction. The branch range is +/- 64MB.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 39 13 | 12 10 | 9 5 | 4 0 |
| Disp26..0 | ~3 | 05 | 285 |

**Clock Cycles: 3**

### BRK – Breakpoint

**Description:**

Execute the breakpoint exception. This is a form of the TRAP instruction.

**Instruction Format:**

|  |  |
| --- | --- |
| 39 5 | 4 0 |
| 0 | 05 |

### BSR – Branch to Subroutine

**Description:**

Branch to a subroutine placing the address of the next instruction in a link register (register 56 to 59). The displacement is relative to the address of the branch instruction. The branch range is +/- 64MB.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 39 13 | 12 | 1110 | 9 5 | 4 0 |
| Disp26..0 | ~ | Lr2 | 16 | 285 |

**Clock Cycles: 3**

### DBcc – Decrement and Branch

DBcc Pn, label

**Description:**

Decrement the loop counter and branch if the condition is false and the loop counter is not equal to minus one. The displacement is relative to the address of the branch instruction. The branch range is +/- 1MB.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 13 | 12 10 | 9 5 | 4 0 |
| Disp20..2 | Pn3 | Cond5 | 295 |

### FBcc – Float Conditional Branch

FBcc Pn, label

**Description:**

Branch if the condition is met. The displacement is relative to the address of the branch instruction. The branch range is +/- 1MB.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 13 | 12 10 | 9 5 | 4 0 |
| Disp20..2 | Pn3 | Cond5 | 305 |

|  |  |  |
| --- | --- | --- |
| **Cond5** | **Mnemonic** | **Test** |
| 0 | F | 0 |
| 1 | EQ | Zf |
| 2 | OGT | !nanf && !zf && !vf && !nf |
| 3 | OGE | zf || (!nanf && !vf && !nf) |
| 4 | OLT | nf && (!nanf && !vf && !zf) |
| 5 | OLE | zf || (nf && !nanf) |
| 6 | OGL | !nanf && !vf && !zf |
| 7 | OR | !nanf |
| 8 | UN | nanf |
| 9 | UEQ | nanf || zf |
| A | UGT | nanf || (!nf && !vf && !zf) |
| B | UGE | nanf || zf || !nf |
| C | ULT | nanf || (nf && !zf) |
| D | ULE | nanf || zf || nf |
| E | NE | !zf |
| F | T | 1 |
| 10 | SF | 0 |
| 11 | SEQ | Zf |
| 12 | GT | !nanf && !zf && !vf && !nf |
| 13 | GE | zf || (!nanf && !vf && !nf) |
| 14 | LT | nf && (!nanf && !vf && !zf) |
| 15 | LE | zf || (nf && !nanf) |
| 16 | GL | !nanf && !vf && !zf |
| 17 | GLE | !nanf |
| 18 | NGLE | nanf |
| 19 | NGL | nanf || zf |
| 1A | NLE | nanf || (!nf && !vf && !zf) |
| 1B | NLT | nanf || (zf || !nf) |
| 1C | NGE | nanf || (nf && !zf) |
| 1D | NGT | nanf || zf || nf |
| 1E | SNE | !zf |
| 1F | ST | 1 |

**Clock Cycles: 4**

### JMP – Jump to Address

**Description:**

Compute the effective address and jump to it. If Ra=63 then the program counter is used.

*In the author’s opinion it does not make a lot of sense to be able to use the stack pointer as a jump target. It is more valuable to have the program counter value available.*

**Flag Updates:**

None.

**Operation:**

PC = Ra + Rb or PC = Ra + Imm

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

**Instruction Formats:**

**JMP (Ra, Rb) – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 246 | ~5 | 0 | Sb | Rb6 | Sa | Ra6 | 0 | 06 | Sz2 | 25 |

**Clock Cycles: 1**

**JMP Imm18 (Ra)**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| Imm17..7 | S | I6 | Imm5..0 | Sa | Ra6 | 0 | 06 | Sz2 | 245 |

**Clock Cycles: 1**

### JSR – Jump to Subroutine

**Description:**

Compute the effective address and jump to it. The address of the instruction is stored in a register. If Ra=63 then the program counter is used.

*In the author’s opinion it does not make a lot of sense to be able to use the stack pointer as a jump target. It is more valuable to have the program counter value available.*

**Flag Updates:**

None.

**Operation:**

Rt = PC

PC = Ra + Rb or PC = Ra + Imm

**Clock Cycles:**

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

**Exceptions:** none

**Notes:**

**Instruction Formats:**

**JSR (Ra, Rb) – Register direct**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| 246 | ~5 | 0 | Sb | Rb6 | Sa | Ra6 | St | Rt6 | Sz2 | 25 |

**Clock Cycles: 1**

**JSR Imm18 (Ra)**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| Imm17..7 | S | I6 | Imm5..0 | Sa | Ra6 | St | Rt6 | Sz2 | 245 |

**Clock Cycles: 1**

### NOP – No Operation

NOP

**Description:**

This instruction does not perform any operation. Ty3 0 to 3 indicates a postfix instruction and these codes should not be used for other NOPs.

**Instruction Format:**

|  |  |  |
| --- | --- | --- |
| 39 5 | 7 5 | 4 0 |
| Payload32 | Ty3 | 315 |

### RTI – Return From Interrupt

**Instruction Formats:**

**RTI #Arg,#Rpt**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 24 | 23 | 22 18 | 17 | 16 12 | 11 7 | 6 5 | 4 0 |
| ~3 | 15 | 0 | Arg11..7 | 01 | Arg6..2 | Rpt5 | D2 | 155 |

**Field Description:**

Rpt5 is the number of words to skip past the return address. This is to allow inline subroutine arguments. Up to 32 words may be skipped over. For externally triggered interrupts this field should be zero.

Arg11..2 is the number of words of subroutine arguments to remove from the stack. Up to 1024 words of arguments may be removed.

D2 specifies the number of internal stack entries to unstack. It may be used to perform a multi-level return. Legal values for D are 1,2 or 3. In most cases a single entry is unstacked. If two entries are unstack a two-up level return will occur.

**Operation:**

Optionally pop the status register, condition code group register, and program counter from the internal stack. Add Rpt tetras to the program counter, and Arg tetras to the stack pointer. If returning from an application trap the status register is not popped from the stack.

### RTS – Return from Subroutine

**Description:**

Return from a subroutine by transferring program flow to the address contained in a register.

**Instruction Formats:**

**RTS #Arg,#Rpt**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 43 31 | 30 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 8 | 7 5 | 4 0 |
| Arg19..7 | Wc2 | 0 | A6 | Arg5..0 | Sa | Ra6 | Rpt6 | Op3 | 155 |

**Field Description:**

Rpt6 is the number of tetras to skip past the return address. This is to allow inline subroutine arguments. Up to 32 tetras may be skipped over.

Arg11..2 is the number of tetras of subroutine arguments to remove from the stack. Up to 1024 tetras of arguments may be removed.

### TRAP – Trap

**Description:**

Execute trap. The data field is loaded into the specified target register, Rt. The trap number to execute comes from the contents of register Ra or an immediate value encoded in the instruction. The trap number must be between 1 and 511. Trap numbers below 64 are reserved for the system. Trap numbers 64 and above may be used by applications. Application traps do not store the status register on the stack.

Traps below 64 will use the vector base register to lookup the location of the service routine. Traps above 64 will use the application control register to lookup the location of the service routine.

**Instruction Format:**

**TRAP Rt, Ra, #Data**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 31 | 30 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| Imm15..7 | 02 | 0 | I6 | Imm5..0 | Sa6 | Ra6 | 0 | Rt6 | 02 | 05 |

**TRAP Rt, #Vec, #Data**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 31 | 30 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| Imm15..7 | 02 | 1 | I6 | Imm5..0 | V6 | Vec5..0 | 0 | Rt6 | V87 | 05 |

**Clock Cycles: 1**

**Operation:**

The program counter, condition code register group and the status register (if trap < 64) are pushed on an internal stack. Next the vector is fetched from the exception vector table and jumped to.

## Memory Operations

### LOAD Rn,<ea>

**Description:**

Load register Rt from source. Loading register r54, the stack canary placeholder, will cause a check trap if the value loaded is not equal to the current value of the stack canary register.

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

**Instruction Formats: NDX**

**LOAD Rt, d(Rb) – Register direct**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| D17..7 | 0 | Sb | Rb6 | D6 | D5..0 | St | Rt6 | Sz2 | 165 |

**Clock Cycles: 1**

Notes:

### STORE Rt,<ea>

**Description:**

Store register Ra to destination.

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

**Instruction Formats: NDX**

**STORE Ra, d(Rb) – Register direct**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 29 | 28 | 27 | 26 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| D17..7 | 0 | Sb | Rb6 | Sa | Ra5 | D6 | D5..0 | Sz2 | 165 |

**Clock Cycles: 1**

Notes:

### STORE #imm,<dea>

**Description:**

Store immediate constant to memory.

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

**Instruction Formats: d(Ra)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 29 | 28 21 | 20 | 19 14 | 13 | 12 7 | 6 5 | 4 0 |
| D17..7 | Constant8 | Sa | Ra5 | D6 | D5..0 | Sz2 | 195 |
| Opt Constant40 | | | | | | | |

Notes:

## Modifiers

### CARRY

**Description:**

Apply the carry modifier to following instructions according to a bit mask. This modifier may be used to perform extended precision addition. It may also be used to retrieve the high order multiplier bits or the divide remainder. Note that carry input is not available for the first instruction under the modifier’s shadow.

|  |  |  |
| --- | --- | --- |
| Carry Modifier  Scope | Mask Bit |  |
| 0,1 | Instruction zero |
| 2,3 | Instruction one |
| 4,5 | Instruction two |
| 6,7 | Instruction three |
| 8,9 | Instruction four |
| 10,11 | Instruction five |
| 12,13 | Instruction six |
| 14,15 | Instruction seven |

|  |  |
| --- | --- |
| Mask Bit | Meaning |
| 00 | No carry in or out |
| 01 | Use carry in |
| 10 | Generate carry out |
| 11 | Use carry in and generate carry out |

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 31 | 30 29 | 28 | 27 | 26 21 | 20 13 | 13 | 12 7 | 6 5 | 4 0 |
| Imm15..7 | 02 | 1 | ~ | 86 | Imm6..0 | 0 | Rt6 | ~2 | 15 |

### PRED

**Description:**

Apply the predicate to following instructions according to a bit mask.

|  |  |  |
| --- | --- | --- |
| Pred Modifier  Scope | Mask Bit |  |
| 0,1 | Instruction zero |
| 2,3 | Instruction one |
| 4,5 | Instruction two |
| 6,7 | Instruction three |
| 8,9 | Instruction four |
| 10,11 | Instruction five |
| 12,13 | Instruction six |
| 14,15 | Instruction seven |

|  |  |
| --- | --- |
| Mask Bit | Meaning |
| 00 | Always execute (ignore predicate) |
| 01 | Execute only if predicate is true |
| 10 | Execute only if predicate is false |
| 11 | Always execute (ignore predicate) |

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 31 | 30 29 | 28 | 27 | 26 21 | 20 13 | 13 | 12 10 | 9 5 | 4 0 |
| Imm15..7 | 02 | 1 | ~ | 06 | Imm6..0 | C6 | Pn3 | Cond5 | 15 |