Thor Guide

This document contains information pertaining to the Thor processor including the instruction set and formats and softcore interfacing.

2015

robfinch@Finitron.ca

Finitron

1/1/2015

Table of Contents

[Overview 7](#_Toc436904620)

[Design Objectives 7](#_Toc436904621)

[Programming Model 8](#_Toc436904622)

[General Registers 8](#_Toc436904623)

[Code Address Registers 9](#_Toc436904624)

[Predicates 10](#_Toc436904625)

[Predicate Conditions 10](#_Toc436904626)

[Compiler Usage 11](#_Toc436904627)

[Status Register (SR) 12](#_Toc436904628)

[Segmentation 13](#_Toc436904629)

[Software Support 13](#_Toc436904630)

[Address Formation: 13](#_Toc436904631)

[Selecting a segment register 13](#_Toc436904632)

[Non-Segmented Code Area 13](#_Toc436904633)

[Changing the Code Segment 14](#_Toc436904634)

[Segment Usage Conventions 14](#_Toc436904635)

[Power-up State 14](#_Toc436904636)

[Segment Registers 14](#_Toc436904637)

[TLB 15](#_Toc436904638)

[TLB Registers 17](#_Toc436904639)

[TLBWired (#0h) 17](#_Toc436904640)

[TLBIndex (#1h) 17](#_Toc436904641)

[TLBRandom (#2h) 17](#_Toc436904642)

[TLBPageSize (#3h) 17](#_Toc436904643)

[TLBPhysPage (#5h) 17](#_Toc436904644)

[TLBVirtPage (#4h) 18](#_Toc436904645)

[TLBASID (#7h) 18](#_Toc436904646)

[Vectors 19](#_Toc436904647)

[Vector table: 19](#_Toc436904648)

[Hardware Ports 20](#_Toc436904649)

[Reset 20](#_Toc436904650)

[Clock Cycle Counts 21](#_Toc436904651)

[Instruction Formats 23](#_Toc436904652)

[RR - Register-Register 23](#_Toc436904653)

[RI - Register-Immediate 23](#_Toc436904654)

[CMP Register-Register Compare 23](#_Toc436904655)

[CMPI Register-Immediate Compare 23](#_Toc436904656)

[TST - Register Test Compare 23](#_Toc436904657)

[CTRL- Control 23](#_Toc436904658)

[BR - Relative Branch 23](#_Toc436904659)

[BRK/NOP 23](#_Toc436904660)

[RTS 23](#_Toc436904661)

[JSR - Jump To Subroutine 23](#_Toc436904662)

[Instruction Set 25](#_Toc436904663)

[2ADDU - Register-Register 25](#_Toc436904664)

[2ADDUI - Register-Immediate 26](#_Toc436904665)

[4ADDU - Register-Register 27](#_Toc436904666)

[4ADDUI - Register-Immediate 28](#_Toc436904667)

[8ADDU - Register-Register 29](#_Toc436904668)

[8ADDUI - Register-Immediate 30](#_Toc436904669)

[16ADDU - Register-Register 31](#_Toc436904670)

[16ADDUI - Register-Immediate 32](#_Toc436904671)

[ABS – Absolute Value Register 33](#_Toc436904672)

[ADD - Register-Register 34](#_Toc436904673)

[ADDI - Register-Immediate 35](#_Toc436904674)

[ADDU - Register-Register 36](#_Toc436904675)

[ADDUI - Register-Immediate 37](#_Toc436904676)

[AND - Register-Register 38](#_Toc436904677)

[ANDI - Register-Immediate 39](#_Toc436904678)

[BCDADD - Register-Register 40](#_Toc436904679)

[BCDMUL - Register-Register 41](#_Toc436904680)

[BCDSUB - Register-Register 42](#_Toc436904681)

[BFCHG – Bit-field Change 43](#_Toc436904682)

[BFCLR – Bit-field Clear 44](#_Toc436904683)

[BFEXT – Bit-field Extract 45](#_Toc436904684)

[BFEXTU – Bit-field Extract Unsigned 46](#_Toc436904685)

[BFINS – Bit-field Insert 47](#_Toc436904686)

[BFSET – Bit-field Set 48](#_Toc436904687)

[BITI – Test bits Register-Immediate 49](#_Toc436904688)

[BR - Relative Branch 50](#_Toc436904689)

[BRK –Break 51](#_Toc436904690)

[BSR - Branch to Subroutine 52](#_Toc436904691)

[CACHE – Cache Command 53](#_Toc436904692)

[CAS – Compare and Swap 54](#_Toc436904693)

[CLI – Clear Interrupt Mask 55](#_Toc436904694)

[CMP Register-Register Compare 56](#_Toc436904695)

[CMPI Register-Immediate Compare 57](#_Toc436904696)

[DIV - Register-Register Divide 58](#_Toc436904697)

[DIVI - Register-Immediate Divide 63](#_Toc436904698)

[DIVIU – Unsigned Register-Immediate Divide 65](#_Toc436904699)

[DIVU – Unsigned Register-Register Divide 66](#_Toc436904700)

[ENOR - Register-Register 67](#_Toc436904701)

[EOR - Register-Register 68](#_Toc436904702)

[EORI - Register-Immediate 69](#_Toc436904703)

[FADD – Floating point addition 70](#_Toc436904704)

[FCMP Float Register-Register Compare 74](#_Toc436904705)

[FSUB – Floating point subtraction 75](#_Toc436904706)

[FTOI – Float to Integer 90](#_Toc436904707)

[FTST – Float Register Test Compare 92](#_Toc436904708)

[INC – Increment Memory 94](#_Toc436904709)

[IMM64,IMM56,IMM48,IMM40,IMM32,IMM24,IMM16 95](#_Toc436904710)

[INT –Interrupt 96](#_Toc436904711)

[ITOF – Integer to Float 97](#_Toc436904712)

[JMP - Jump To Address 99](#_Toc436904713)

[JSR - Jump To Subroutine Instruction 100](#_Toc436904714)

[LB – Load Byte 101](#_Toc436904715)

[LBU – Load Byte Unsigned 102](#_Toc436904716)

[LBUX – Load Byte Unsigned Indexed 103](#_Toc436904717)

[LBX – Load Byte Indexed 104](#_Toc436904718)

[LC – Load Character 105](#_Toc436904719)

[LCL – Load Cache Line 106](#_Toc436904720)

[LCU – Load Character Unsigned 107](#_Toc436904721)

[LCUX – Load Character Unsigned Indexed 108](#_Toc436904722)

[LCX – Load Character Indexed 109](#_Toc436904723)

[LDI - Load-Immediate 110](#_Toc436904724)

[LDIS - Load-Immediate Special 111](#_Toc436904725)

[LEA – Load Effective Address 112](#_Toc436904726)

[LH – Load Half-Word 113](#_Toc436904727)

[LHU – Load Half-word Unsigned 115](#_Toc436904728)

[LHUX – Load Half-word Unsigned Indexed 116](#_Toc436904729)

[LHX – Load Half-word Indexed 117](#_Toc436904730)

[LOOP – Loop Branch 118](#_Toc436904731)

[LVB – Load Volatile Byte 119](#_Toc436904732)

[LVC – Load Volatile Character 120](#_Toc436904733)

[LVH – Load Volatile Half-word 121](#_Toc436904734)

[LVW – Load Volatile Word 122](#_Toc436904735)

[LVWAR – Load Volatile Word and Reserve 123](#_Toc436904736)

[LW – Load Word 124](#_Toc436904737)

[LWS – Load Word Special 125](#_Toc436904738)

[LWX – Load Word Indexed 126](#_Toc436904739)

[MAX - Register-Register 127](#_Toc436904740)

[MEMDB – Memory Data Barrier 128](#_Toc436904741)

[MEMSB – Memory Synchronization Barrier 129](#_Toc436904742)

[MFSPR – Special Register-Register 130](#_Toc436904743)

[MIN - Register-Register 131](#_Toc436904744)

[MLO – Mystery Logical Operation 132](#_Toc436904745)

[MOV - Register-Register 133](#_Toc436904746)

[MOVS – Move Special Register- Special Register 134](#_Toc436904747)

[MTSPR –Register-Special Register 135](#_Toc436904748)

[MUL - Register-Register Multiply 136](#_Toc436904749)

[MULI - Register-Immediate Multiply 137](#_Toc436904750)

[MULU – Unsigned Register-Register Multiply 138](#_Toc436904751)

[MULUI – Unsigned Register-Immediate Multiply 139](#_Toc436904752)

[MUX – Multiplex 140](#_Toc436904753)

[NAND - Register-Register 141](#_Toc436904754)

[NEG - Negate Register 142](#_Toc436904755)

[NOP – No Operation 143](#_Toc436904756)

[NOR - Register-Register 144](#_Toc436904757)

[NOT – Logical Not 145](#_Toc436904758)

[OR - Register-Register 146](#_Toc436904759)

[ORI - Register-Immediate 147](#_Toc436904760)

[PUSH – Push Register 148](#_Toc436904761)

[ROL – Rotate Left 150](#_Toc436904762)

[ROLI – Rotate Left by Immediate 151](#_Toc436904763)

[ROR – Rotate Right 152](#_Toc436904764)

[RORI – Rotate Right by Immediate 153](#_Toc436904765)

[RTE – Return from Exception Routine 154](#_Toc436904766)

[RTI – Return from Interrupt Routine 155](#_Toc436904767)

[RTS – Return from Subroutine 156](#_Toc436904768)

[SB – Store Byte 158](#_Toc436904769)

[SBX – Store Byte Indexed 159](#_Toc436904770)

[SC – Store Character 160](#_Toc436904771)

[SCX – Store Character Indexed 161](#_Toc436904772)

[SEI – Set Interrupt Mask 162](#_Toc436904773)

[SH – Store Half-word 163](#_Toc436904774)

[SHL – Shift Left 164](#_Toc436904775)

[SHLI – Shift Left by Immediate 165](#_Toc436904776)

[SHLU – Shift Left Unsigned 166](#_Toc436904777)

[SHLUI – Shift Left Unsigned by Immediate 167](#_Toc436904778)

[SHR – Shift Right 168](#_Toc436904779)

[SHRI – Shift Right by Immediate 169](#_Toc436904780)

[SHRU – Shift Right Unsigned 170](#_Toc436904781)

[SHRUI – Shift Right Unsigned by Immediate 171](#_Toc436904782)

[SHX – Store Half-word Indexed 172](#_Toc436904783)

[STCMP – String Compare 173](#_Toc436904784)

[STFND – String Find 174](#_Toc436904785)

[STI – Store Immediate 175](#_Toc436904786)

[STIX – Store Immediate Indexed 176](#_Toc436904787)

[STMV – String Move 177](#_Toc436904788)

[STSB – Store String Byte 178](#_Toc436904789)

[STSC – Store String Character 179](#_Toc436904790)

[STSH – Store String Half-word 180](#_Toc436904791)

[STSW – Store String Word 181](#_Toc436904792)

[SUB - Register-Register 182](#_Toc436904793)

[SUBI - Register-Immediate 183](#_Toc436904794)

[SUBU - Register-Register 184](#_Toc436904795)

[SUBUI - Register-Immediate 185](#_Toc436904796)

[SW – Store Word 186](#_Toc436904797)

[SWCR – Store Word and Clear Reservation 187](#_Toc436904798)

[SWS – Store Word Special 188](#_Toc436904799)

[SWX – Store Word Indexed 189](#_Toc436904800)

[SYNC – Synchronization Barrier 190](#_Toc436904801)

[SYS –Call system routine 194](#_Toc436904802)

[TLB – TLB Command 195](#_Toc436904803)

[TST - Register Test Compare 197](#_Toc436904804)

[Opcode Map 201](#_Toc436904805)

# Overview

Thor is a powerful 64 bit superscalar processor that represents a generational refinement of processor architecture. The processor contains 64, 64 bit general purpose integer registers. Thor uses variable length instructions varying between one and eight bytes in length and handles 8, 16, 32, and 64 bit data within a 64 bit address space.

## Design Objectives

This processor is somewhat pedantic in nature and targeted towards high performance operation as a general purpose processor. Following are some of the criteria that were used on which to base the design.

|  |
| --- |
| * Designed for Superscalar operation - the ability to execute more than one instruction at a time. To achieve high performance it is generally accepted that a processor must be able to execute more than a single instruction in any given clock cycle. |
| * Simplicity - architectural simplicity leads to a design that is easy to implement resulting in reliability and assured correctness along with easy implementation of supporting tools such as compilers. Simplicity also makes it easier to obtain high performance and results in lower overall cost. |
| * Extensibility - the design must be extensible so that features not present in the first release can easily be added at a later date. |
| * Low Cost |

This design meets the above objectives in the following ways. The instruction set has been designed to minimize the interactions between instructions, allowing instructions to be executed as independent units for superscalar operation. There are a sufficient number of registers to allow the compiler to schedule parallel processing of code. A reasonably large general purpose register set is available making the design reasonably compatible with many existing compilers and assemblers. Where needed, additional specialized instructions have been added to the processor to support a sophisticated operating system and interrupt management.

# Programming Model

## General Registers

There are 64 general purpose registers. General purpose registers are 64 bits wide. The general registers may hold integer or floating point values.

Register #0 is always zero.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
| r0 | always zero |  |  | LC | Loop Counter |
| r1 | return value |  |  |  |  |
| r2 | return value |  |  | C0 |  |
| r3 |  |  |  | C1 | return address |
| r4 |  |  |  | C2 |  |
| r5 |  |  |  | C3 |  |
| r6 |  |  |  | C4 |  |
| r7 |  |  |  | C5 |  |
| r8 |  |  |  | C6 |  |
| r9 |  |  |  | C7 |  |
| r10 |  |  |  | C8 |  |
| r11 |  |  |  | C9 |  |
| r12 |  |  |  | C10 |  |
| r13 |  |  |  | C11 | catch link address |
| r14 |  |  |  | C12 | exception table pointer |
| r15 |  |  |  | C13 | exceptioned PC |
| r16 |  |  |  | C14 | interrupted PC |
| r17 |  |  |  | C15 | program counter, read only |
| r18 |  |  |  |  |  |
| r19 |  |  |  | ZS | zero segment |
| r20 |  |  |  | DS | data segment |
| r21 |  |  |  | ES | extra segment |
| r22 |  |  |  | FS |  |
| r23 |  |  |  | GS |  |
| r24 |  |  |  | HS |  |
| r26 | Base Pointer |  |  | SS | stack segment |
| r27 | User Stack Pointer1 |  |  | CS | code segment |
| r28 | Accessible only in kernel mode |  |  |  |  |
| r29 |  |  |  |  |  |
| r30 |  |  |  |  |  |
| r31 |  |  |  |  |  |
| r32/F0 | Floating point |  |  |  |  |
| … |  |  |  |  |  |
| r63/F31 |  |  |  |  |  |

1 this register is implied in the push and rts instructions, and updated by hardware

## Code Address Registers

The processor contains sixteen code address registers (C0-C15). Several of the registers are reserved for predefined purposes. A code address register is used in the formation and storage of code addresses.

|  |  |  |
| --- | --- | --- |
| Reg # |  | Usage |
| 0 | Always Zero | Absolute address formation |
| 1 |  | Subroutine return address |
| 2 |  | This register is available for general use. |
| 3 |  | This register is available for general use. |
| 4 |  | This register is available for general use. |
| 5 |  | This register is available for general use. |
| 6 |  | This register is available for general use. |
| 7 |  | This register is available for general use. |
| 8 |  | This register is available for general use. |
| 9 |  |  |
| 10 |  |  |
| 11 | Catch Link Register | Used by the compiler to link to try/catch handlers. |
| 12 | Exception Table Pointer | This register points to the exception table in memory. |
| 13 | Exceptioned PC | This register is set when an exception occurs |
| 14 | Interrupted PC | This register is automatically set during a hardware interrupt |
| 15 | Program Counter | Relative address formation. |
|  |  |  |

Code address registers may be used to point to a block of code from which the JSR instruction can index into with its 24 bit offset. For instance a register may contain a pointer to a class method jump list; the JSR instruction can then index into this list in order to invoke a method.

The presence of multiple code address registers allows multi-level return addresses to be used for performance. Leaf routines may use C1 as the return address. Next to leaf routines may use C2, etc. So that memory operations are avoided when implementing subroutine call and return.

The program counter register is read-only. The program counter cannot be modified by moving a value to this register.

## Predicates

The processor features predicated execution of all instructions. Whether or not an instruction is executed depends on the contents of a predicate register and the predicate condition specified in the predicate byte. There are 16 predicate registers each of which hold three flags. These flags are set as the result of a compare operation. The flags represent equality (eq) signed less than (lt) and unsigned less than (ltu).

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 2 | 1 | 0 |
| ~ | ltu | lt | eq |

All instructions are executed conditionally determined by the value of a predicate register. The special predicate 00 executes the break vector.

### Predicate Conditions

|  |  |  |  |
| --- | --- | --- | --- |
| Cond. |  | Test |  |
| 0 | PF | 0 | Always false – Instructions predicated with condition zero never execute regardless of the predicate register contents. This is used for extended immediate values as well. The false predicate byte for instructions is 90h. |
| 1 | PT | 1 | Always True – The instruction predicated with an always true condition always executes regardless of the predicate register contents. The always true predicate byte is 01h. Other true predicates are instruction short-forms. |
| 2 | PEQ | eq | Equal – instruction executes if the predicate register equal flag is set |
| 3 | PNE | !eq | Not Equal – instruction executes if the predicate register equal flag is clear |
| 4 | PLE | lt|eq | Less or Equal – predicate less or equal flag is set |
| 5 | PGT | !(lt|eq) | greater than |
| 6 | PGE | !lt | greater or equal |
| 7 | PLT | lt | less than |
| 8 | PLEU | ltu|eq | unsigned less or equal |
| 9 | PGTU | !(ltu|eq) | unsigned greater than |
| 10 | PGEU  POR | !ltu | unsigned greater or equal  Ordered for floating point |
| 11 | PLTU  PUN | ltu | unsigned less than  Unordered for floating point |
| 12 |  |  |  |
| 13 | PSIG | signal | execute if external signal is true |
| 14 |  |  |  |
| 15 |  |  |  |

### Compiler Usage

The compiler uses predicate register #15 to conditionally move TRUE / FALSE values to a register when evaluating a logical operation.

Predicate registers beginning with P0 and incrementing are applied for use as the control flow nesting level increases. The compiler does not support control flow nesting more than 14 levels in a single subroutine. Predicate registers beginning with P14 and decrementing are used in the evaluation of the hook operator. Care must be taken such that the number of predicate registers in use does not exceed the number available.

|  |  |  |
| --- | --- | --- |
| Pred. | Usage |  |
| P0 | control flow level 0 |  |
| P1 | control flow nesting level 1 |  |
| P2 | control flow nesting level 2 |  |
| … |  |  |
| Pn | control flow nesting level n (n not to exceed 14) |  |
| … |  |  |
| P12 | third hook operator in an expression |  |
| P13 | second hook operator in an expression |  |
| P14 | first hook operator in an expression |  |
| P15 | conditionally moves TRUE/FALSE for logical expressions |  |

## Status Register (SR)

This register contains bits that control the overall operation of the processor or reflect the processor’s state. Bits are included for interrupt masking, and system / application mode indicator. This register is split into two halves with both halves having the same format. The lower half of the register is what determines how the processor works. The upper half of the register maintains a backup copy of the lower half for interrupt processing. There are instructions provided for manipulating the interrupt mask.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31..16 | 15 | 14 | 13 | 12 | 11..8 | 7..0 |
| same format as 15..0 | Interrupt Mask | Reserved | Kernel / Application Mode Indicator | Float Except. Enable |  |  |
|  | IM | ~ | S | FXE |  |  |

 The Kernel / Application Mode indicator is read-only.

IM = interrupt mask

Maskable interrupts are disabled when this bit is set.

# Segmentation

The processor contains eight segment registers. The segment register to use during address formation for data addresses is identified by a field in the instruction. This field is set to default values by the assembler. For code addresses segment register #7 (the CS) is always used.

* If segmentation is not desired then segmentation can effectively be ignored by setting all the segment registers to zero. The processor can also be built without segmentation by commenting out the ‘SEGMENTATION’ definition.

## Software Support

Segment registers may only be transferred to or from one of the general purpose registers. The [mtspr](#_MTSPR_–Register-Special_Register) and [mfspr](#_MFSPR_–_Special) instructions can be used to perform the move. A segment register may also be loaded using the [LDIS](#_LDIS_-_Load-Immediate) instruction. After loading a segment register the instruction stream should be synchronized with a memory barrier ([MEMSB](#_MEMSB_–_Memory)) to ensure the segment value can be ready for a following memory operation.

## Address Formation:

Non-segmented address bits 0 to 11 pass through the segmentation module unchanged. Address bits 63 to 12 are added to the contents of the segment register to form the final segmented address. Note that there is no shift associated with the segment addition. Future implementations of the processor may include additional low order address bits in the segment register in order to allow a finer grain for memory page / paragraph size.

|  |  |
| --- | --- |
| Address[63:12] | Address[11:0] |
| + | + |
| Segment register value[63:12] | 00012 |
| = | |
| Segmented address[63:0] | |

## Selecting a segment register

A specific segment register for a memory operation may be selected using a segment prefix in assembler code. Segment prefixes apply to data addresses only. Code addresses always use segment register #7 – the code segment.

## Non-Segmented Code Area

The address range defined as 64’hFxxxxxxxxxxxxxxx (the top nibble is ‘F’) is a non-segmented code area. This area allows the operating system to work without paying attention to the code segment. Interrupt and exception vectors should vector into the non-segmented code area. The only way to change the code segment is by transferring to the operating system via a sys call instruction.

## Changing the Code Segment

The only way to change the code segment is by transferring to the operating system via a sys call instruction. The operating system, while operating in the non-segmented code area, can alter the code segment without causing a transfer of control. The operating system establishes the code segment for a task while running in the non-segmented code area.

## Segment Usage Conventions

Segment register #7 is the code segment (CS) register. All program counter addresses are formed with the code segment register unless the upper nibble of the address is ‘F’ in which case the code segment is ignored.

Segment register #6 is the stack segment (SS) register by convention. Segment register #1 is the data segment (DS) by convention.

## Power-up State

On reset the value in the segment registers are undefined. Note that the processor begins executing instructions out of the non-segmented code area as the reset address is 64’hFFFFFFFFFFFFFF80. One of the first tasks of the boot program would be to initialize the segment registers to known values. The segment register must be setup to perform data accesses properly.

### Segment Registers

|  |  |  |  |
| --- | --- | --- | --- |
| Num |  | Long name | Comment |
| 0 | ZS | zero (NULL) segment | by convention contains zero |
| 1 | DS | data segment | by convention – default for loads/stores |
| 2 | ES | extra segment | by convention |
| 3 | FS |  |  |
| 4 | GS |  |  |
| 5 | HS |  |  |
| 6 | SS | Stack segment | default for stack load/stores |
| 7 | CS | Code segment | always used for code addressing |

# TLB

The processor uses a 64 entry TLB (translation look-aside buffer) in order to support virtual memory. The TLB supports variable page sizes from 4kB to 1MB. The TLB is organized as an eight-way eight-set cache.

The TLB is updated by first placing values into the TLB holding registers using the TLB instruction, then issuing a TLB write command using the TLB command instruction.

Address translations will not take place until the TLB is enabled. An enable TLB command must be issued using the TLB command instruction.

TLB Entries:

C2..0

G

D

V

ASID7...0

Virtual Page51...0

Physical Page51...0

G = Global

The global bit marks the TLB entry as a global address translation where the ASID field is not used to match addresses.

ASID = address space identifier

The ASID field in the TLB entry must match the processor’s current ASID value in order for the translation to be considered valid, unless the G bit is set. If the G bit is set in the TLB entry, then the ASID field is ignored during the address comparison.

C = cachability bits

If the cachability bits are set to 001b then the page is uncached, otherwise the page is cached.

D = dirty bit

The dirty bit is set by hardware when a write occurs to the virtual memory page identified by the TLB entry.

V = valid bit

This bit must be set in order for the address translation to be considered valid. The entire TLB may be invalidated using the invalidate all command.

## TLB Registers

### TLBWired (#0h)

This register limits random updates to the TLB to a subset of the available number of ways. TLB ways below the value specified in the Wired register will not be updated randomly. Setting this register provides a means to create fixed translation settings. For instance if the wired register is set to two, the sixteen fixed entries will be available.

### TLBIndex (#1h)

This register contains the entry number of the TLB entry to be read from or written to.

### TLBRandom (#2h)

This register contains a random three bit value used to update a random TLB entry during a TLB write operation.

### TLBPageSize (#3h)

The TLBPageSize register controls which address bits are significant during a TLB lookup.

|  |  |  |
| --- | --- | --- |
| N | Page Size |  |
| 0 | 4KiB |  |
| 1 | 16kiB |  |
| 2 | 64kiB |  |
| 3 | 256kiB |  |
| 4 | 1MiB |  |
|  |  |  |
|  |  |  |

### TLBPhysPage (#5h)

The TLBPhysPage register is a holding register that contains the page number for an associated virtual address. This register is transferred to or from the TLB by TLB instructions.

|  |
| --- |
| 63 0 |
| Physical Page Number |

### TLBVirtPage (#4h)

The TLBVirtPage register is a holding register that contains the page number for an associated physical address. This register is transferred to or from the TLB by TLB instructions.

|  |
| --- |
| 63 0 |
| Virtual Page Number |

### TLBASID (#7h)

The TLBASID register is a holding register that contains the address space identifier (ASID) , valid, dirty, global, and cachability bits associated with a TLB entry. This register is transferred to or from the TLB by TLB instructions.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 63 16 | 15 8 | 6 4 | 2 | 1 | 0 |
| ----- | ASID | C | G | D | V |

# Memory Operations:

## Basic Operations

Basic memory operations include loads, stores, pushes, pops and string operates. Other than those operations there are no other instructions that access memory. Note that return addresses are not pushed onto the stack automatically.

## Memory Addressing Modes

The core supports both register indirect with displacement and scaled indexed addressing. Indexed addressing is supported only with the general purpose register load store operations.

## Pre-fetching data

The load instructions may be used to pre-fetch data by specifying a load into register R0. If R0 is used as the load target register then the load operation will not cause any exception.

## Bypassing the Data Cache

There are several load instructions that bypass the data-cache when loading – see the load volatile (LVx) instructions. These instructions are useful for I/O operations or for when it is better if the data cache is not loaded for performance reasons.

The volatile load instructions only offer sign extension and not zero extension. To zero extend data loaded by a volatile load operation follow it with one of the zero extension (ZXx) instructions.

## Push Operations

The core supports a data push to stack and data pop operation. The data push operation both decrements the stack pointer and stores data to the stack. Argument pushing is commonly used in high-level languages. Subroutine arguments pushed to the stack in high-level languages are usually popped off the stack simply by adding to the stack pointer.

An additional push operation includes pushing an effective address to the stack.

## Load Speculation

The core may load data speculatively in advance of its use provided there is no address overlap with a preceding store instruction.

## Store Issuing

Stores will only be issued if there are no instructions that can exception before the store in the instruction queue. Since many instructions do not cause any exceptions this happens fairly often.

## Address Reservation

The address reservation instructions rely on the external memory system to support address reservation. There are only two instruction (LVWAR, SWCR) associated with address reservation. the load instruction creates an address reservation and the store instruction clears it. In a multi-core system the reservation may be created or cleared by another processing core.

## Synchronization Operations

The core includes memory data barrier and memory instruction barrier instructions to allow data to be synchronized during program runs.

# Vectors

The processor vectors to $FFFFFFFFFFFFFF80 on a reset. All other vectoring is done through a vector table. The vector table allows for 256 entries. The vector table base address is established by code address register C12. During an external IRQ the processor looks at a vector number bus to determine the vector to use for the IRQ. This vector number may be hard-coded in which case all IRQ’s will be vectored to the same location. The address vectored to is the sum of C12 and an offset supplied in the instruction multiplied by sixteen. The contents of C12 are undefined at reset; this register must be loaded before interrupts can be processed.

### Vector table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Vector Number | Usage / Description | |  |  |
| 0 | BREAK instruction vector | |  |  |
| 1 | SLEEP vector (branch to self) | |  |  |
| 2 | Task reschedule interrupt | |  |  |
| … |  | |  |  |
| 192 | Spurious interrupt | |  |  |
| 193 | IRQ level 1 | 1000 Hz interrupt |  |  |
| 194 | IRQ level 2 | 100 Hz interrupt |  |  |
| … | Other IRQ levels |  |  |  |
| 207 | IRQ level 15 | keyboard interrupt |  |  |
| … |  | |  |  |
| 240 | overflow (integer) | |  |  |
| 241 | divide by zero (integer) | |  |  |
| 242 | floating point | |  |  |
| … |  | |  |  |
| 248 | DTLBMiss | |  |  |
| 249 | ITLB Miss | |  |  |
| 250 | Unimplemented instruction | |  |  |
| 251 | Bus error – data load / store | |  |  |
| 252 | Bus error – instruction fetch | |  |  |
| 253 | reserved | |  |  |
| 254 | NMI interrupt vector | |  |  |
| 255 | * reserved | |  |  |

# Hardware Ports

Thor uses a WISHBONE bus to communicate with the outside world.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | I/O | Width | WB |  |  |
| corenum | I | 32 |  | core number – this number is used to identify the core and is reflected in the cpuid register. Meant to be a hardcoded constant. |  |
| rst\_i | I | 1 | WB | reset signal |  |
| clk\_i | I | 1 | WB | clock |  |
| km | O | 1 |  | kernel mode indicator |  |
| nmi\_i | I | 1 |  | non-maskable interrupt input |  |
| irq\_i | I | 1 |  | maskable interrupt input |  |
| vec\_i | I | 8 |  | interrupt vector |  |
| bte\_o | O | 2 | WB | burst type extension |  |
| cti\_o | O | 3 | WB | cycle type indicator |  |
| bl\_o | O | 5 |  | burst length output |  |
| lock\_o | O | 1 | WB | bus lock |  |
| resv\_o | O | 1 |  | reserve address |  |
| resv\_i | I | 1 |  | address reservation status in |  |
| cres\_o | O | 1 |  | clear address reservation |  |
| cyc\_o | O | 1 | WB | cycle is valid |  |
| stb\_o | O | 1 | WB | data transfer is taking place |  |
| ack\_i | I | 1 | WB | data transfer acknowledge |  |
| err\_i | I | 1 | WB | bus error occurred input |  |
| we\_o | O | 1 | WB | write enable |  |
| sel\_o | O | 8 | WB | byte lane selects |  |
| adr\_o | O | 64 | WB | address output |  |
| dat\_i | I | 64 | WB | data input bus |  |
| dat\_o | O | 64 | WB | data output bus |  |
|  |  |  |  |  |  |

WB = see the WISHBONE spec rev B3

Notes:

Stores issue only from the head of the instruction queue when it is known that no exceptions have taken place.

# Reset

On reset the core begins fetching and executing instruction at address $FFFFFFFFFFFFFF80. Note that the last 32 bytes of memory should not be used to store instruction unless it is okay that the core “wraps” around to address zero when performing fetches. This is because the core is fetching cache lines in advance.

On power-up or reset interrupts are disabled automatically, In order to enable interrupts the RTI instruction must be executed. An [RTI](#_RTI_–_Return) automatically enables interrupts. Note that the interrupt mask must also be cleared with the CLI instruction to allow maskable interrupts to occur.

After reset or NMI the core begins processing at a half the maximum clock rate. The [STP](#_STP_–_Stop) instruction must be issued to get the processer running at full speed.

# Clock Cycle Counts

The core has a minimum CPI of 0.5 clocks per instruction running trivial sample code. Many instructions can be done in pairs provided there are no dependencies between the instructions. Due to the out of order execution ability of the core the latency of longer running instructions may be hidden. The core may be busy working on up to four instructions at once: two ALU or an ALU and memory op, a floating point op and a branch instruction.

# Core Parameters

|  |  |  |
| --- | --- | --- |
| DBW | 32 | The parameter controls the width of data processed by the core. Set to 64 for 64 bit processing. This parameter should be either 64 or 32. If the width is set to 32 bit then double precision floating point operations are unavailable. |
| ABW | 32 | This parameter controls the width of the external address bus. |
| ALU1BIG | 0 | This parameter controls whether or not ALU1 supports all instructions or only a subset of instructions. The default is to support only the most common instructions. (0 = limited, 1 = all) in order to reduce the size of the core.  Limiting the number of instructions supported may impact performance of the core because it may not be possible to issue two instructions in the same cycle. |

# Instruction Formats

Instructions vary in length from one to eight bytes. There are only a few of single byte instructions consisting of only a predicate. Some of the more common formats are shown below.

All instruction sequences begin with a predicate byte that determines the conditions under which the instruction executes. With the exception of special predicate values, the next field in the instruction is always the opcode byte. All opcodes may be preceded by an extended constant value.

### RR - Register-Register

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Func | Rt | Rb | Ra | Opcode | Predicate | |
| Func6 | Rt6 | Rb6 | Ra6 | Opcode8 | Pn4 | Pc4 |

### RI - Register-Immediate

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Immediate11..0 | Rt6 | Ra6 | Opcode8 | Pn4 | Pc4 |

### CMP Register-Register Compare

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 12 | 11 8 | 7 0 | |
| Opc4 | Rb6 | Ra6 | 14 | Pt4 | Pn4 | Pc4 |

### CMPI Register-Immediate Compare

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 22 | | | | 21 16 | | | | 15 12 | | | 11 8 | | 7 0 | | | |
| Immed9..0 | | | | Ra6 | | | | 24 | | | Pt4 | | Pn4 | | | Pc4 |
| TST - Register Test Compare | | | | | | | | | | | | |  | | | |  | | CTRL- Control | | | | | | |
| 2322 21 16 | | | 15 12 | | | 11 8 | | | 7 0 | | | |  | | | |  | | 15 8 | | 7 0 | | | | |
| O2 | Ra6 | | 04 | | | Pt4 | | | Pn4 | | Pc4 | |  | | | |  | | Opcode8 | | Pn4 | | | Pc4 | |
| BR - Relative Branch | | | | | | | | | | | |  | |  | | | | BRK/NOP | | | |  | RTS | | | |
| 23 16 | | 15 8 | | | | | 7 0 | | | | |  | |  | | | | 7 0 | | | |  | 7 0 | | | |
| Disp7..0 | | 34 | | | D11..8 | | Pn4 | | | Pc4 | |  | | |  | | | 0/14 | | 04 | |  | 14 | | 14 | |

### JSR - Jump To Subroutine

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 47 24 | 23 16 | | 15 8 | 7 0 | |
| Offset23..0 | Cr4 | Crt4 | Opcode8 | Pn4 | Pc4 |

# Instruction Set Summary

A number of rarely used instructions may only execute on ALU #0. These instructions are identified in the text.

## Branch Instructions

The core has only a single relative branch instruction which branches relative to the address of the next instruction. This single branch instruction may be used to implement branching on multiple complex conditions when combined with a predicate. The branch instruction supports a 12 bit displacement field.

### Branch Speculation

Branches are performed speculatively in the fetch stage of the core according to branch predictor output. Branches use a (2, 2) co-relating branch predictor with a 256 entry branch history table. Both global and local branch histories are maintained.

## Loops

There is a loop instruction and corresponding loop count register to support counted loops. The loop instruction is predicted as always taken and does not consume room in the branch history table. Like a branch instruction a loop instruction takes place at the fetch stage of the core. The loop instruction supports only and eight bit displacement field which may not be extended.

## Subroutine Call / Return

Program counter relative jumps and calls may be achieved using the program counter as the index register in jump instructions. The jump instruction directly supports up to 24 bit addressing. A shorter jump instruction is available that supports 16 bit addressing. The addressing capabilities of the jump instruction may be increased by applying an immediate prefix to the instruction. It is envisioned that the 16/24 bit jump addressing is sufficient for most cases when combined with usage of the code segment.

## Comparison Operations

Comparison operations include CMP and TST (compare to zero). Comparison operations set a predicate register to the result status of the comparison.

## Arithmetic Operations

|  |  |
| --- | --- |
| Mnemonic |  |
| ADD | addition |
| ADDU | unsigned addition |
| SUB | subtraction |
| SUBU | unsigned subtraction |
| MUL | multiplication |
| MULU |  |
| DIV | division |
| DIVU |  |
| NEG | negative |
| ABS | absolute value |
| MIN | minimum value |
| MAX | maximum value |

## Bitwise Operations

Bitwise operations include ‘and’, ‘or’ and exclusive ‘or’ along with their inverted versions.

|  |  |  |
| --- | --- | --- |
| Mnemonic | Has Immediate Form |  |
| AND | Y |  |
| OR | Y |  |
| EOR | Y |  |
| NAND | N |  |
| NOR | N |  |
| ENOR | N |  |
| COM | N | invert bits |

## Logical Operations

The core includes the logical ‘not’ (NOT) operation. The NOT operation reduces the value to a one or zero result.

# Detailed Instruction Set

### 2ADDU - Register-Register

**Description:**

Multiply Ra by two and add Rb and place the sum in the target register. This instruction will never cause an overflow exception.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 08h6 | Rt6 | Rb6 | Ra6 | 40h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra \* 2 + Rb

**Exceptions:** none

### 2ADDUI - Register-Immediate

**Description:**

Multiply Ra by two and add immediate and place the sum in the target register. This instruction will never cause an overflow exception.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Immediate11..0 | Rt6 | Ra6 | 6Bh8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra \* 2 + immediate

**Exceptions:** none

### 4ADDU - Register-Register

**Description:**

Multiply Ra by four and add Rb and place the sum in the target register. This instruction will never cause an exception.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 09h6 | Rt6 | Rb6 | Ra6 | 40h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra \* 4 + Rb

**Exceptions:** none

### 4ADDUI - Register-Immediate

**Description:**

Multiply Ra by four and add immediate and place the sum in the target register. This instruction will never cause an exception.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Immed11..0 | Rt6 | Ra6 | 6Ch8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra \* 4 + immediate

**Exceptions:** none

### 8ADDU - Register-Register

**Description:**

Multiply Ra by eight and add Rb and place the sum in the target register. This instruction will never cause an exception.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 0Ah6 | Rt6 | Rb6 | Ra6 | 40h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra \* 8 + Rb

**Exceptions:** none

### 8ADDUI - Register-Immediate

**Description:**

Multiply Ra by eight and add immediate and place the sum in the target register. This instruction will never cause an exception.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Immed11..0 | Rt6 | Ra6 | 6Dh8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra \* 8 + immediate

**Exceptions:** none

### 16ADDU - Register-Register

**Description:**

Multiply Ra by sixteen and add Rb and place the sum in the target register. This instruction will never cause an exception.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 0Bh6 | Rt6 | Rb6 | Ra6 | 40h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra \* 16 + Rb

**Exceptions:** none

### 16ADDUI - Register-Immediate

**Description:**

Multiply Ra by sixteen and add immediate and place the sum in the target register. This instruction will never cause an exception.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Immed11..0 | Rt6 | Ra6 | 6Eh8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra \* 16 + immediate

**Exceptions:** none

### ABS – Absolute Value Register

**Description:**

This instruction takes the absolute value of a register and places the result in a target register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 34 | Rt6 | Ra6 | A7h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

If Ra < 0

Rt = -Ra

else

Rt = Ra

**Exceptions:** none

### ADD - Register-Register

**Description:**

Add two registers and place the sum in the target register. This instruction may cause an overflow exception.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 00h6 | Rt6 | Rb6 | Ra6 | 40h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra + Rb

**Exceptions:** integer overflow

### ADDI - Register-Immediate

**Description:**

Add a register and immediate value and place the sum in the target register. This instruction may cause an overflow exception.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Immediate11..0 | Rt6 | Ra6 | 48h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra + immediate

**Exceptions:** integer overflow

### ADDU - Register-Register

**Description:**

Add registers Ra and Rb and place the result into register Rt. This instruction will never cause any exceptions.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 04h6 | Rt6 | Rb6 | Ra6 | 40h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra + Rb

**Exceptions:** none

### ADDUI - Register-Immediate

**Description:**

Add a register and immediate value and place the sum in the target register. This instruction will never cause an exception.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Immediate11..0 | Rt6 | Ra6 | 4Ch8 | Pn4 | Pc4 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 22 | 21 16 | 15 8 | 7 0 | |
| Immediate9..0 | Rt6 | 47h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra + Immediate

**Exceptions:** none

### AND - Register-Register

**Description:**

Bitwise and’s two registers and places the result in a target register.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 00h6 | Rt6 | Rb6 | Ra6 | 50h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra & Rb

**Exceptions:** none

### ANDC – And with Complement

**Description:**

Bitwise and’s a register Ra with the complement of register Rb and places the result in a target register.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 06h6 | Rt6 | Rb6 | Ra6 | 50h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra & ~Rb

**Exceptions:** none

### ANDI - Register-Immediate

**Description:**

Bitwise and’s register and an immediate value and places the result in a target register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Immediate11..0 | Rt6 | Ra6 | 53h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra & immediate

**Exceptions:** none

### BCDADD - Register-Register

**Description:**

Adds two registers using BCD arithmetic and places the result in a target register. Only the low order byte of the register is used. The result is an eight bit BCD number.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 00h6 | Rt6 | Rb6 | Ra6 | F5h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra + Rb

**Exceptions:** none

### BCDMUL - Register-Register

**Description:**

Multiplies two registers using BCD arithmetic and places the result in a target register. Only the low order byte of the register is used. The result is a 16 bit BCD value.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 02h6 | Rt6 | Rb6 | Ra6 | F5h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 Only

**Operation:**

Rt = Ra \* Rb

**Exceptions:** none

### BCDSUB - Register-Register

**Description:**

Subtracts two registers using BCD arithmetic and places the result in a target register. Only the low order byte of the register is used. The result is an eight bit BCD number.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 01h6 | Rt6 | Rb6 | Ra6 | F5h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra - Rb

**Exceptions:** none

### BFCHG – Bit-field Change

**Description:**

Inverts the bit-field in Ra located between the mask begin (mb) and mask end (me) bits and stores the result in the target register.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4744 | 43 40 39 34 | | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| ~4 | 34 | me6 | mb6 | Rt6 | Ra6 | AAh8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Exceptions:** none

### BFCLR – Bit-field Clear

**Description:**

Sets the bits to zero of the bit-field in Ra located between the mask begin (mb) and mask end (me) bits and stores the result in the target register.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4744 | 43 40 39 34 | | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| ~4 | 24 | me6 | mb6 | Rt6 | Ra6 | AAh8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Exceptions:** none

### BFEXT – Bit-field Extract

**Description:**

Extracts a bit-field from register Ra located between the mask begin (mb) and mask end (me) bits and places the sign extended result into the target register.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4744 | 43 40 39 34 | | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| ~4 | 54 | me6 | mb6 | Rt6 | Ra6 | AAh8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Exceptions:** none

### BFEXTU – Bit-field Extract Unsigned

**Description:**

Extracts a bit-field from register Ra located between the mask begin (mb) and mask end (me) bits and places the zero extended result into the target register.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4744 | 43 40 39 34 | | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| ~4 | 44 | me6 | mb6 | Rt6 | Ra6 | AAh8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Exceptions:** none

### BFINS – Bit-field Insert

**Description:**

Inserts a bit-field into the target register located between the mask begin (mb) and mask end (me) bits from the low order bits of Ra.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4744 | 43 40 39 34 | | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| ~4 | 04 | me6 | mb6 | Rt6 | Ra6 | AAh8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Exceptions:** none

### BFINSI – Bit-field Insert Immediate

**Description:**

Inserts a bit-field into the target register located between the mask begin (mb) and mask end (me) bits from the bits specified in the instruction.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4744 | 43 40 39 34 | | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| ~4 | 64 | me6 | mb6 | Rt6 | Imm6 | AAh8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Exceptions:** none

### BFSET – Bit-field Set

**Description:**

Sets the bits to one of the bit-field in Ra located between the mask begin (mb) and mask end (me) bits and stores the result in the target register.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4744 | 43 40 39 34 | | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| ~4 | 14 | me6 | mb6 | Rt6 | Ra6 | AAh8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Exceptions:** none

### BITI – Test bits Register-Immediate

**Description:**

Logically and’s register and an immediate value and places the result in a predicate register. If the result of the ‘and’ operation is zero the predicate register’s zero flag is set, otherwise it is cleared. If the result is negative the predicate’s less than flag is set, otherwise it is cleared.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 28 | 26 | 25 22 | 21 16 | 15 8 | 7 0 | |
| Immediate11..0 | ~2 | Pt4 | Ra6 | 46h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Pt = flag results( Ra & immediate)

**Predicate Results:**

|  |  |
| --- | --- |
| Predicate flag | Setting |
| eq | set if result is zero |
| lt | set if result is negative |
| ltu | set if result is odd (bit 0 is set) |
|  |  |

**Exceptions:** none

### BR - Relative Branch

**Description:**

A branch is made relative to the address of the next instruction.

* The twelve bit displacement field cannot be extended with an immediate constant prefix. Branches are executed immediately in the instruction fetch stage of the processor before it is known if there is a prefix present.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 16 | 15 8 | | 7 0 | |
| Disp7..0 | 3h4 | D11..8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s / Branch

**Operation:**

PC <= PC + displacement

**Exceptions:** none

### BRK –Break

**Description:**

This instruction contains only a predicate byte. The Break exception is executed.

**Instruction Format:**

|  |  |
| --- | --- |
| 7 0 | |
| 04 | 04 |

### BSR - Branch to Subroutine

**Description:**

This is an alternate mnemonic for the JSR instruction. A jump is made to the sum of the sign extended displacement supplied in the displacement field of the instruction and the specified code address register Cr.

The subroutine return address is stored in a code address register specified in the Crt field of the instruction.

Typically code address register #1 is used to store the return address.

**Instruction Formats:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 47 24 | 23 20 | 19 16 | 15 8 | 7 0 | |
| Displacement23..0 | 154 | Crt4 | A2h8 | Pn4 | Pc4 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 24 | 23 20 | 1916 | 15 8 | 7 0 | |
| Displacement15..0 | 154 | Crt4 | A1h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Exceptions:** none

### CACHE – Cache Command

**Description:**

This instruction issues a command to the cache.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 2524 | 23 22 | 21 16 | 15 8 | 7 0 | |
| Func6 | ~2 | ~2 | Ra6 | 9Fh8 | Pn4 | Pc4 |

**Operation:**

**Commands:**

|  |  |
| --- | --- |
| Func6 |  |
| 0 | Invalidate entire instruction cache |
| 1 | Invalidate instruction cache line (address in Ra) |
| 32 | Invalidate entire data cache |
| 33 | Invalidate data cache line (address in Ra) |
|  |  |

### CAS – Compare and Swap

**Description:**

If the contents of the addressed memory cell is equal to the contents of Rb then a sixty-four bit value is stored to memory from the source register Rc. The original contents of the memory cell are loaded into register Rt. The memory address is the sum of the sign extended displacement and register Ra. The memory address must be word aligned. If the operation was successful then Rt and Rb will be the same value. The compare and swap operation is an atomic operation; the bus is locked during the load and potential store operation. This operation assumes that the addressed memory location is part of the volatile region of memory and bypasses the data cache.

**Instruction Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 40 | 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Displacement7..0 | Rt6 | Rc6 | Rb6 | Ra6 | 97h8 | Pn4 | Pc4 |

**Operation:**

Rt = memory [Ra + displacement]

if memory[Ra + displacement] = Rb

memory[Ra + displacement] = Rc

**Assembler:**

CAS Rt,Rb,Rc,offset[Ra]

### CLI – Clear Interrupt Mask

**Description:**

This instruction is used to enable interrupts.

**Instruction Format:**

|  |  |  |
| --- | --- | --- |
| 15 8 | 7 0 | |
| FAh8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Operation:**

im = 0

**Exceptions:** none

### CMP Register-Register Compare

**Description:**

The register compare instruction compares two registers and sets the flags in the target predict register as a result.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3128 | 27 22 | 21 16 | 15 12 | 11 8 | 7 0 | |
| 04 | Rb6 | Ra6 | 14 | Pt4 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

if signed Ra < signed Rb

P.lt = true

else

P.lt = false

if unsigned Ra < unsigned Rb

P.ltu = true

else

P.ltu = false

if Ra = Rb

P.eq = true

else

P.eq = false

**Exceptions:** none

### CMPI Register-Immediate Compare

**Description:**

The register immediate compare instruction compares a register to an immediate value and sets the flags in the target predict register as a result. Both a signed and unsigned comparison take place at the same time.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 22 | 21 16 | 15 12 | 11 8 | 7 0 | |
| Immed10 | Ra6 | 24 | Pt4 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

if signed Ra < signed immediate

P.lt = true

else

P.lt = false

if unsigned Ra < unsigned immediate

P.ltu = true

else

P.ltu = false

if Ra = immediate

P.eq = true

else

P.eq = false

### CNTLO- Count Leading Ones

**Description:**

This instruction counts the number of leading ones in a register and places the result in a target register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 64 | Rt6 | Ra6 | A7h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

**Exceptions:** none

### CNTLZ- Count Leading Zeros

**Description:**

This instruction counts the number of leading zeros in a register and places the result in a target register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 54 | Rt6 | Ra6 | A7h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

**Exceptions:** none

### CNTPOP- Population Count

**Description:**

This instruction counts the number of one bits in a register and places the result in a target register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 74 | Rt6 | Ra6 | A7h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

**Exceptions:** none

### COM – Bitwise Complement

**Description:**

This instruction performs a bitwise complement on a register and places the result in a target register. If bit is a one then the bit is replaced with is zero otherwise it is replaced with a one.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| B4 | Rt6 | Ra6 | A7h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = ~ Ra

**Exceptions:** none

### CPUID – CPU Identification

**Description:**

This instruction returns general information about the core.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 04 | Rt6 | Ra6 | 41h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Info[Ra]

**Exceptions**: none

|  |  |  |
| --- | --- | --- |
| Index | bits | Information Returned |
| 0 | 63 to 0 | The processor core identification number. This field is determined from an external input. It would be hard wired to the number of the core in a multi-core system. |
| 2 | 63 to 0 | Manufacturer name first eight chars “Finitron” |
| 3 | 63 to 0 | Manufacturer name last eight characters |
| 4 | 63 to 0 | CPU class “64BitSS” |
| 5 | 63 to 0 | CPU class |
| 6 | 63 to 0 | CPU Name “Thor” |
| 7 | 63 to 0 | CPU Name |
| 8 | 63 to 0 | Model Number “M1” |
| 9 | 63 to 0 | Serial Number “1234” |
| 10 | 63 to 0 | Features bitmap |
| 11 | 31 to 0 | Instruction Cache Size (32kB) |
| 11 | 63 to 32 | Data cache size (16kB) |

### DIV - Register-Register Divide

**Description:**

Performs a signed division of two registers and places the quotient in the target register. This instruction may cause an overflow or divide by zero exception.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 03h6 | Rt6 | Rb6 | Ra6 | 40h8 | Pn4 | Pc4 |

**Clock Cycles:** 65

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra / Rb

### DIVI - Register-Immediate Divide

**Description:**

Performs a signed divide of a register and an immediate value and places the result in a target register. This instruction may cause an overflow or divide by zero exception.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Immediate11..0 | Rt6 | Ra6 | 4Bh8 | Pn4 | Pc4 |

**Clock Cycles:** 65

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra / immediate

### DIVIU – Unsigned Register-Immediate Divide

**Description:**

Performs an unsigned divide of a register and an immediate value and places the result in a target register. This instruction will not cause an overflow or divide by zero exception.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Immediate11..0 | Rt6 | Ra6 | 4Fh8 | Pn4 | Pc4 |

**Clock Cycles:** 65

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra / immediate

**Exceptions:** none

### DIVU – Unsigned Register-Register Divide

**Description:**

Performs an unsigned division of two registers and places the quotient in the target register. This instruction not cause an overflow or divide by zero exception.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 07h6 | Rt6 | Rb6 | Ra6 | 40h8 | Pn4 | Pc4 |

**Clock Cycles:** 65

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra / Rb

**Exceptions:** none

### ENOR - Register-Register

**Description:**

Bitwise exclusive or register with register and place inverted result in target register.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 05h6 | Rt6 | Rb6 | Ra6 | 50h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = ~(Ra ^ Rb)

### EOR - Register-Register

**Description:**

Bitwise exclusive or register with register and place result in target register.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 02h6 | Rt6 | Rb6 | Ra6 | 50h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra ^ Rb

### EORI - Register-Immediate

**Description:**

Bitwise exclusive or register with immediate and place result in target register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Immediate11..0 | Rt6 | Ra6 | 55h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra ^ immediate

### INC – Increment Memory

**Description:**

Memory is incremented by the amount specified in the instruction. The memory address is the sum of the sign extended displacement and register Ra. The amount is between -128 and +127. Note that the increment is not an atomic memory operation. The bus is not locked during the increment to allow cached data to be incremented. For atomic memory operations see the [CAS](#_CAS_–_Compare) instruction.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 40 | 3937 | 36 28 | 27 22 | | 21 16 | 15 8 | 7 0 | |
| Amt8 | Sg3 | Displacement8..0 | O3 | Sz3 | Ra6 | C7h8 | Pn4 | Pc4 |

**Execution Units:** All Memory

**Operation:**

(mem[Ra+offset]) = (mem[Ra+offset]) + amt

### IMM64,IMM56,IMM48,IMM40,IMM32,IMM24,IMM16

**Immediate Extensions**

The immediate extension predicates are used to extend the immediate constant of the following instruction. The extensions may add from one to seven bytes more to the constant. Most, but not all instructions can accept a predicated immediate.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Immediate | | | | | | | Predicate | |
| Immediate63..8 | | | | | | | 84 | 04 |
| Immediate55..8 | | | | | | 74 | 04 |
| Immediate47..8 | | | | | 64 | 04 |
| Immediate39..8 | | | | 54 | 04 |
| Immediate31..8 | | | 44 | 04 |
| Immediate23..8 | | 34 | 04 |
| Immediate15..8 | 24 | 04 |

**Clock Cycles:** 1

**Execution Units: Enqueue**

### INT –Interrupt

**Description:**

This instruction calls a system function located as the sum of the zero extended offset times 16 plus code address register 12. The return address is stored in the IPC register (code address register #14).

The offset field of this instruction cannot be extended.

Note that this instruction is automatically invoked for hardware interrupt processing. This instruction would not normally be used by software and is not supported by the assembler. The return address stored is the address of the interrupt instruction, not the address of the next instruction. To call system routines use the SYS instruction.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 24 | 23 20 | 19 16 | 15 8 | 7 0 | |
| Offset7..0 | Ch4 | Eh4 | A6h8 | Pn4 | Pc4 |

### JMP - Jump To Address

**Description:**

This is an alternate mnemonic for the JSR instruction.

A jump is made to the sum of the zero extended offset supplied in the offset field of the instruction and the specified code address register Cr. The JMP instruction may be used with an immediate predicate constant in order to extend the address range of the jump.

**Instruction Formats:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 47 24 | 23 20 | 19 16 | 15 8 | 7 0 | |
| Offset23..0 | Cr4 | 04 | A2h8 | Pn4 | Pc4 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 24 | 23 20 | 19 16 | 15 8 | 7 0 | |
| Offset15..0 | Cr4 | 04 | A1h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

pc = Cr[n] + offset

### JSR - Jump To Subroutine Instruction

**Description:**

A jump is made to the sum of the zero extended offset supplied in the offset field of the instruction and the specified code address register Cr. The JSR instruction may be used with an immediate predicate constant in order to extend the address range of the jump.

The subroutine return address is stored in a code address register specified in the Crt field of the instruction. Typically code address register #1 is used.

An immediate constant prefix applied to this instruction overrides offset bits 8 to 23 and acts like an eight bit immediate constant extension used by other instructions.

**Instruction Formats:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 47 24 | 23 20 | 19 16 | 15 8 | 7 0 | |
| Offset23..0 | Cr4 | Crt4 | A2h8 | Pn4 | Pc4 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 24 | 23 20 | 19 16 | 15 8 | 7 0 | |
| Offset15..0 | Cr4 | Crt4 | A1h8 | Pn4 | Pc4 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 20 | 19 16 | 15 8 | 7 0 | |
| Cr4 | Crt4 | A0h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Cr[t] = pc

pc = Cr[n] + offset

### LB – Load Byte

**Description:**

An eight bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction will load data from the cache and cause a cache load operation if the data isn’t in the cache. To bypass the cache use the [LVB](#_LVB_–_Load) instruction.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rt6 | Ra6 | 80h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (mem[Ra+offset])

### LBU – Load Byte Unsigned

**Description:**

An eight bit value is loaded from memory and zero extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rt6 | Ra6 | 81h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = zero extend (mem[Ra+offset])

### LBUX – Load Byte Unsigned Indexed

**Description:**

An eight bit value is loaded from memory zero extended and placed in the target register Rt. The memory address is the sum of register Ra and scaled register Rb.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 37 | 36 | 3534 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Seg3 | ~ | Sc2 | Rc6 | Rb6 | Ra6 | B1h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = mem[Ra+Rb]

### LBX – Load Byte Indexed

**Description:**

An eight bit value is loaded from memory and placed in the target register. The memory address is the sum of register Ra and scaled register Rb.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 37 | 36 | 3534 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Seg3 | ~ | Sc2 | Rc6 | Rb6 | Ra6 | B0h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (mem[Ra+Rb])

### LC – Load Character

**Description:**

A sixteen bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended displacement and register Ra. The memory address must be character aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rt6 | Ra6 | 82h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (mem[Ra + displacement])

### LCL – Load Cache Line

**Description:**

The cache line is loaded from memory into the cache (instruction or data). The memory address is the sum of the sign extended offset and register Ra.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Tgt6 | Ra6 | 8Fh8 | Pn4 | Pc4 |

**Execution Units:** Cache / Memory

**Operation:**

Rt = sign extend (mem[Ra+offset])

Target:

|  |  |
| --- | --- |
| Tgt6 | Cache |
| 0 | instruction cache |
| 1 | data cache |

### LCU – Load Character Unsigned

**Description:**

A sixteen bit value is loaded from memory and zero extended, then placed in the target register. The memory address is the sum of the sign extended displacement and register Ra. The memory address must be character aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rt6 | Ra6 | 83h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = zero extend (mem[Ra + displacement])

### LCUX – Load Character Unsigned Indexed

**Description:**

A sixteen bit value is loaded from memory, zero extended and placed in the target register Rt. The memory address is the sum of register Ra and scaled register Rb. The memory address must be character aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 37 | 36 | 3534 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Seg3 | ~ | Sc2 | Rc6 | Rb6 | Ra6 | B3h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = mem[Ra + Rb \* scale]

### LCX – Load Character Indexed

**Description:**

A sixteen bit value is loaded from memory, sign extended and placed in the target register Rt. The memory address is the sum of register Ra and scaled register Rb. The memory address must be character aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 37 | 36 | 3534 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Seg3 | ~ | Sc2 | Rc6 | Rb6 | Ra6 | B2h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = mem[Ra + Rb \* scale]

### LDI - Load-Immediate

**Description:**

This instruction loads a sign extended immediate constant into a register. The immediate constant may be extended by using an immediate prefix instruction.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 22 | 21 16 | 15 8 | 7 0 | |
| Immediate9..0 | Rt6 | 6Fh8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = immediate

### LDIS - Load-Immediate Special

**Description:**

This instruction loads a sign extended immediate constant into a special purpose register. The immediate constant may be extended by using an immediate prefix instruction. Typical usage is to initialize a code address register with a target address.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 22 | 21 16 | 15 8 | 7 0 | |
| Immediate9..0 | Spr6 | 9Dh8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Spr = immediate

### LEA – Load Effective Address

**Description:**

This is an alternate mnemonic for the ADDUI instruction. The memory address is placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Offset11..0 | Rt6 | Ra6 | 4Ch8 | Pn4 | Pc4 |

**Operation:**

Rt = Ra+offset

**Execution Units:** All ALU’s

### LH – Load Half-Word

**Description:**

A thirty-two bit value is loaded from memory and sign extended, then placed in the target register Rt. The memory address is the sum of the sign extended displacement and register Ra. The memory address must be half-word aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rt6 | Ra6 | 84h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (mem[Ra + displacement])

### LHU – Load Half-word Unsigned

**Description:**

A thirty-two bit value is loaded from memory and zero extended, then placed in the target register Rt. The memory address is the sum of the sign extended displacement and register Ra. The memory address must be half-word aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rt6 | Ra6 | 85h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = zero extend (mem[Ra + displacement])

### LHUX – Load Half-word Unsigned Indexed

**Description:**

A thirty-two bit value is loaded from memory, zero extended and placed in the target register. The memory address is the sum of register Ra and register Rb. The memory address must be half-word aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 37 | 36 | 3534 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Seg3 | ~ | Sc2 | Rc6 | Rb6 | Ra6 | B5h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = mem[Ra+Rb\*scale]

### LHX – Load Half-word Indexed

**Description:**

A thirty-two bit value is loaded from memory sign extended and placed in the target register Rt. The memory address is the sum of register Ra and scaled register Rb. The memory address must be half-word aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 37 | 36 | 3534 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Seg3 | ~ | Sc2 | Rc6 | Rb6 | Ra6 | B4h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (mem[Ra + Rb \* scale])

### LOOP – Loop Branch

**Description:**

A branch is made relative to the address of the next instruction if the loop count register is non-zero. The loop count register is decremented by this instruction. The predicate condition must also be met. The loop branch is predicted as always taken and does not consume room in the branch predication tables. The displacement constant may not be extended as the loop takes place in the instruction fetch stage of the core.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 23 16 | 15 8 | 7 0 | |
| Disp7..0 | A4h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s / Branch

**Operation:**

If LC <> 0

PC <= PC + displacement

LC = LC - 1

### LVB – Load Volatile Byte

**Description:**

An eight bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended displacement and register Ra. This instruction bypasses the data cache. Use this instruction to load data from volatile memory regions such as I/O devices. This instruction may also be used when it is known that the data is better not cached.

There is no indexed or unsigned form for this instruction. The value loaded may be zero extended rather than sign extended by following it with the [ZXB](#_ZXB_–_Zero) instruction.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rt6 | Ra6 | ACh8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (mem[Ra+offset])

### LVC – Load Volatile Character

**Description:**

A sixteen bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra. This instruction bypasses the data cache. Use this instruction to load data from volatile memory regions such as I/O devices.

There is no indexed or unsigned form for this instruction.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rt6 | Ra6 | ADh8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (mem[Ra+offset])

### LVH – Load Volatile Half-word

**Description:**

A thirty-two bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra. This instruction bypasses the data cache. Use this instruction to load data from volatile memory regions such as I/O devices.

There is no indexed or unsigned form for this instruction.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rt6 | Ra6 | AEh8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (mem[Ra+offset])

### LVW – Load Volatile Word

**Description:**

A sixty-four bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra. This instruction bypasses the data cache. Use this instruction to load data from volatile memory regions such as I/O devices.

There is no indexed or unsigned form for this instruction.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rt6 | Ra6 | AFh8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (mem[Ra + displacement])

### LVWAR – Load Volatile Word and Reserve

**Description:**

A sixty-four bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra. Additionally the reserve signal is activated on the bus to tell the memory system to place an address reservation. This instruction bypasses the data cache. Use this instruction to load data from volatile memory regions such as I/O devices. The primary purpose of this instruction is to setup semaphores. See also the [SWCR](#_SWCR_–_Store), [CAS](#_CAS_–_Compare) instructions.

There is no indexed form for this instruction.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rt6 | Ra6 | 8Bh8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (mem[Ra + displacement]); reserve = 1

### LW – Load Word

**Description:**

A sixty-four bit value is loaded from memory and placed in the target register. The memory address is the sum of the sign extended displacement and register Ra. The memory address must be word aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rt6 | Ra6 | 86h8 | Pn4 | Pc4 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | ~ | Rt6 | Ra6 | D6h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Exceptions:**

If the target register is R0 then this instruction will not cause an exception. Otherwise an exception may be caused by a data-bus error signal input or a TLB miss.

**Operation:**

Rt = mem[Ra + displacement]

### LWS – Load Word Special

**Description:**

A sixty-four bit value is loaded from memory and placed in the special purpose register. The memory address is the sum of the sign extended offset and register Ra. The memory address must be word aligned.

There is no indexed form for this instruction.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rt6 | Ra6 | 8Eh8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Spr = mem[Ra + displacement]

### LWX – Load Word Indexed

**Description:**

A sixty-four bit value is loaded from memory and placed in the target register. The memory address is the sum of register Ra and scaled register Rb. The memory address must be word aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 37 | 36 | 3534 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Seg3 | ~ | Sc2 | Rc6 | Rb6 | Ra6 | B6h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = mem[Ra + Rb\*scale]

### MAX - Register-Register

**Description:**

Determines the maximum of two values in registers Ra and Rb and places the result in the target register Rt.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 11h6 | Rt6 | Rb6 | Ra6 | 40h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

IF Ra < Rb

Rt = Rb

else

Rt = Ra

### MEMDB – Memory Data Barrier

**Description:**

All memory accesses before the MEMDB command are completed before any memory accesses after the data barrier are started. Note that this instruction has an effect even if the predicate is false; this does not affect the correct operation of the program, only performance is affected.

**Instruction Format:**

|  |  |  |
| --- | --- | --- |
| 15 8 | 7 0 | |
| F9h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** Memory

### MEMSB – Memory Synchronization Barrier

**Description:**

All instructions before the MEMSB command are completed before any memory access is started. Note that this instruction has an effect even if the predicate is false; this does not affect the correct operation of the program, only performance is affected.

**Instruction Format:**

|  |  |  |
| --- | --- | --- |
| 15 8 | 7 0 | |
| F8h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** Memory

### MFSPR – Special Register-Register

**Description:**

This instruction moves from a special purpose register into a general purpose one.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| ~4 | Rt6 | Spr6 | A8h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Spr[n]

**Special Purpose Registers**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reg # | R/W |  |  |  |
| 00-15 | RW | PRED | specific predicate register #0 to 15 |  |
| 16-31 | RW | CREGS | Code address register array (C0 to C15) |  |
| 32-39 | RW | SREGS | Segment base register array (zs,ds,es,fs,gs,hs,ss,cs) |  |
| 40-47 |  |  | * reserved for segmentation |  |
| 48 | R | MID | Machine ID |  |
| 49 | R | FEAT | Features |  |
| 50 | R | TICK | Tick count |  |
| 51 | RW | LC | Loop Counter |  |
| 52 | RW | PREGS | Predicate register array |  |
| 53 | RW | ASID | address space identifier |  |
| 59 | RW | EXC | exception cause register |  |
| 60 | W | BIR | Breakout index register |  |
| 61 | RW |  | Breakout register - additional spr’s |  |
| 63 |  |  | reserved |  |

Additional Spr’s are available by setting the breakout index register to an Sor index value, then accessing the Spr through the breakout register.

### MIN - Register-Register

**Description:**

Determines the minimum of two values in registers Ra and Rb and places the result in the target register Rt.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 10h6 | Rt6 | Rb6 | Ra6 | 40h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

IF Ra < Rb

Rt = Ra

else

Rt = Rb

### MLO – Mystery Logical Operation

**Description:**

The MLO instruction performs an operation that is determined at run-time as opposed to compile time. The operation to be performed is one of the register-register logical operations. Register Rc contains the function code for the operation. Registers Ra and Rb are the operands to the instruction. The result is placed in register Rt.

The MLO instruction is provided to help avoid writing self-modifying code for performance reasons.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Rt6 | Rc6 | Rb6 | Ra6 | 51h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra op(Rc) Rb

### MOV - Register-Register

**Description:**

This instruction moves one general purpose register to another. This instruction is shorter and uses one less register port than using the OR instruction to move between registers.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 04 | Rt6 | Ra6 | A78 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra

### MOVS – Move Special Register- Special Register

**Description:**

This instruction moves one special purpose register to another. The primary purpose of this instruction is to allow transfers directly between code address or segment registers.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| ~4 | Sprt6 | Spr6 | AB8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Sprt = Spra

### MTSPR –Register-Special Register

**Description:**

Move a general purpose register into a special purpose register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| ~4 | Spr6 | Ra6 | A9h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Spr[n] = Ra

### MUL - Register-Register Multiply

**Description:**

Performs a signed multiply of two registers and places the product in the target register. This instruction may cause an overflow exception.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 02h6 | Rt6 | Rb6 | Ra6 | 40h8 | Pn4 | Pc4 |

**Clock Cycles:** 5

**Execution Units:** ALU #0 Only

**Operation:**

Rt = Ra \* Rb

### MULI - Register-Immediate Multiply

**Description:**

Performs a signed multiply of a register and an immediate value and places the result in a target register. This instruction may cause an overflow exception.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Immediate11..0 | Rt6 | Ra6 | 4Ah8 | Pn4 | Pc4 |

**Clock Cycles:** 5

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra \* immediate

### MULU – Unsigned Register-Register Multiply

**Description:**

Performs an unsigned multiply of two registers and places the product in the target register. This instruction will never cause an overflow exception.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 06h6 | Rt6 | Rb6 | Ra6 | 40h8 | Pn4 | Pc4 |

**Clock Cycles:** 5

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra \* Rb

**Exceptions:** none

### MULUI – Unsigned Register-Immediate Multiply

**Description:**

Performs an unsigned multiply of a register and an immediate value and places the result in a target register. This instruction will never cause an overflow exception.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Immediate11..0 | Rt6 | Ra6 | 4Eh8 | Pn4 | Pc4 |

**Clock Cycles:** 5

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra \* immediate

**Exceptions:** none

### MUX – Multiplex

**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:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Rt6 | Rc6 | Rb6 | Ra6 | 72h8 | Pn4 | Pc4 |

**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

### NAND - Register-Register

**Description:**

Bitwise and’s two registers inverts the result and places the result in a target register.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 03h6 | Rt6 | Rb6 | Ra6 | 50h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = ~(Ra & Rb)

**Exceptions:** none

### NEG - Negate Register

**Description:**

This instruction negates a register and places the result in a target register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 14 | Rt6 | Ra6 | A7h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = - Ra

### NOP – No Operation

**Description:**

This instruction contains only a predicate byte. This is a single byte no-operation code. It can be used to align code addresses or as a fill byte.

The NOP operation is not queued by the processing core and is not present in the pipeline.

**Instruction Format:**

|  |  |
| --- | --- |
| 7 0 | |
| 14 | 04 |

**Two byte Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 18 8 | | 7 0 | |
| F4 | 14 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** None

**Operation:**

<none>

**Exceptions:** none

### NOR - Register-Register

**Description:**

Bitwise inclusively or two registers and place inverted result in the target register.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 04h6 | Rt6 | Rb6 | Ra6 | 50h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = ~(Ra | Rb)

**Exceptions:** none

### NOT – Logical Not

**Description:**

This instruction performs a logical NOT on a register and places the result in a target register. If the value in a register is non-zero then the result is zero. If the value in the register is zero then the result is one. This instruction results in either a one or zero being placed in the target register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 24 | Rt6 | Ra6 | A7h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = ! Ra

**Exceptions:** none

### OR - Register-Register

**Description:**

Bitwise inclusively or two registers and place the result in the target register.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 01h6 | Rt6 | Rb6 | Ra6 | 50h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra | Rb

**Exceptions:** none

### ORC – Or with Complement

**Description:**

Bitwise inclusively or register Ra and the complement of register Rb and place the result in the target register.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 07h6 | Rt6 | Rb6 | Ra6 | 50h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra | ~Rb

**Exceptions:** none

### ORI - Register-Immediate

**Description:**

Bitwise inclusively or register with immediate and place the result in the target register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Immediate11..0 | Rt6 | Ra6 | 54h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra | imm

**Exceptions:** none

### ROL – Rotate Left

**Description:**

Rotate register Ra left by Rb bits and place the result into register Rt. The most significant bit is shifted into the least significant bit. The rotation takes place modulo 64 of the value in register Rb (only the lower six bits of the register are used).

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 04h6 | Rt6 | Rb6 | Ra6 | 58h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra << Rb

**Exceptions:** none

### ROLI – Rotate Left by Immediate

**Description:**

Rotate register Ra left by n bits and place the result into register Rt. The most significant bit is shifted into the least significant bit.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 14h6 | Rt6 | Imm6 | Ra6 | 58h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra << #n

**Exceptions:** none

### ROR – Rotate Right

**Description:**

Rotate register Ra right by Rb bits and place the result into register Rt. The least significant bit is shifted into the most significant bit.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 05h6 | Rt6 | Rb6 | Ra6 | 58h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra >> Rb

**Exceptions:** none

### RORI – Rotate Right by Immediate

**Description:**

Rotate register Ra right by n bits and place the result into register Rt. The least significant bit is shifted into the most significant bit.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 15h6 | Rt6 | Imm6 | Ra6 | 58h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra >> #n

**Exceptions:** none

### RTE – Return from Exception Routine

**Description:**

The program counter is loaded with the value contained in code address register #13 which is the EPC register.

**Instruction Format:**

|  |  |  |
| --- | --- | --- |
| 15 8 | 7 0 | |
| F3h8 | Pn4 | Pc4 |

**Operation:**

PC = Cr[13]

Flags = FlagsBackup

**Exceptions:** none

### RTI – Return from Interrupt Routine

**Description:**

The program counter is loaded with the value contained in code address register #14 which is the IPC register. Additionally the interrupt mask is cleared to enable interrupts.

**Instruction Format:**

|  |  |  |
| --- | --- | --- |
| 15 8 | 7 0 | |
| F4h8 | Pn4 | Pc4 |

**Operation:**

pc = Cr[14]

Flags = FlagsBackup

Flags.im = 0

**Exceptions:** none

### RTS – Return from Subroutine

**Description:**

The program counter is loaded with the value contained in the specified code address register plus a zero extended four bit immediate constant. The constant may not be extended. This allows the return instruction to return a few bytes past the usual return address. This is used to allow static parameters to be passed to the subroutine in inline code. The stack pointer may also be adjusted using the proper form of the RTS instruction for which the immediate constant must be a multiple of eight.

Note that the JMP instruction may also be used to return from a subroutine. Similarly this instruction may also be used to perform a jump to one of the first sixteen addresses relative to a code address register.

This instruction has a single byte short form that always executes when encountered. For the short form the program counter is loaded from code address register one.

**Instruction Formats:**

**Return past calling address**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 20 | 19 16 | 15 8 | 7 0 | |
| Cr4 | Im4 | A3h8 | Pn4 | Pc4 |

**Stack pointer adjusting**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 24 | 23 20 | 19 16 | 15 8 | 7 0 | |
| Immed8 | Cr4 | Im4 | F2h8 | Pn4 | Pc4 |

**Short Form:**

|  |  |
| --- | --- |
| 7 0 | |
| 14 | 14 |

**Execution Units:** All ALU’s / Branch

**Operation:**

PC = Cr[N] +Imm4

**Short Form Operation:**

PC = Cr[1] +Imm4

**Stack Pointer Adjust:**

PC = Cr[1] +Imm4

SP = SP + Imm

**Exceptions:** none

### SB – Store Byte

**Description:**

An eight bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rt6 | Ra6 | 90h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

memory[Ra+offset] = Rb[7..0]

### SBX – Store Byte Indexed

**Description:**

An eight bit value is stored to memory from the source register Rc. The memory address is the sum of register Ra and Rb.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 37 | 36 | 3534 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Seg3 | ~ | Sc2 | Rc6 | Rb6 | Ra6 | C0h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

memory[Ra+Rb] = Rb

### SC – Store Character

**Description:**

A sixteen bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra. The memory address must be character aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rt6 | Ra6 | 91h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

memory[Ra+displacement] = Rb[15..0]

### SCX – Store Character Indexed

**Description:**

A sixteen bit value is stored to memory from the source register Rc. The memory address is the sum of register Ra and scaled register Rb. The memory address must be character aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 37 | 36 | 3534 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Seg3 | ~ | Sc2 | Rc6 | Rb6 | Ra6 | C1h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

memory[Ra+Rb\*scale] = Rb

### SEI – Set Interrupt Mask

**Description:**

The interrupt mask is set, disabling maskable interrupts.

**Instruction Format:**

|  |  |  |
| --- | --- | --- |
| 15 8 | 7 0 | |
| FBh8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Operation:**

im = 1

**Exceptions:** none

### SH – Store Half-word

**Description:**

A thirty-two bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra. The memory address must be half-word aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 37 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Seg3 | Displacement8..0 | Rb6 | Ra6 | 92h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

memory[Ra + displacement] = Rb[31..0]

### SHL – Shift Left

**Description:**

Shift register Ra left by Rb bits and place result into register Rt. A zero is shifted into the least significant bit.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 00h6 | Rt6 | Rb6 | Ra6 | 58h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra << Rb

**Exceptions:** none

### SHLI – Shift Left by Immediate

**Description:**

Shift register Ra left by n bits and place result into register Rt. A zero is shifted into the least significant bit.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 10h6 | Rt6 | Imm6 | Ra6 | 58h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra << #n

**Exceptions:** none

### SHLU – Shift Left Unsigned

**Description:**

Shift register Ra left by Rb bits and place the result into register Rt. A zero is shifted into the least significant bit.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 02h6 | Rt6 | Rb6 | Ra6 | 58h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra << Rb

**Exceptions:** none

### SHLUI – Shift Left Unsigned by Immediate

**Description:**

Shift register Ra left by n bits and place the result into register Rt. A zero is shifted into the least significant bit.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 12h6 | Rt6 | Imm6 | Ra6 | 58h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra << #n

**Exceptions:** none

### SHR – Shift Right

**Description:**

Shift register Ra right by Rb bits and place result in register Rt. The sign bit is preserved.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 01h6 | Rt6 | Rb6 | Ra6 | 58h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra >> Rb

**Exceptions:** none

### SHRI – Shift Right by Immediate

**Description:**

Shift register Ra right by n bits and place result into register Rt. The sign bit is preserved.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 11h6 | Rt6 | Imm6 | Ra6 | 58h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra >> #n

**Exceptions:** none

### SHRU – Shift Right Unsigned

**Description:**

Shift register Ra right by register Rb bits. A zero is shifted into the sign bit.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 03h6 | Rt6 | Rb6 | Ra6 | 58h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra >> Rb

**Exceptions:** none

### SHRUI – Shift Right Unsigned by Immediate

**Description:**

Shift register Ra right by n bits and place result into register Rt. A zero is shifted into the sign bit.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 13h6 | Rt6 | Imm6 | Ra6 | 58h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra >> #n

**Exceptions:** none

### SHX – Store Half-word Indexed

**Description:**

A thirty-two bit value is stored to memory from the source register Rb. The memory address is the sum of register Ra and scaled register Rb. The memory address must be half-word aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 37 | 36 | 3534 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Seg3 | ~ | Sc2 | Rc6 | Rb6 | Ra6 | C2h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

memory[Ra+Rb] = Rb

### STCMP – String Compare

**Description:**

This instruction compares data from the memory location addressed by Ra plus Rc to the memory location addressed by Rb plus Rc until the loop counter LC reaches zero or until a mismatch occurs. Rc acts as an index and increments or decrements by the size of the operation as the move takes place. This instruction is interruptible. The data must be in the same segment and appropriately aligned. The loop counter is set to zero when a mismatch occurs. The index of the mismatch is contained in register Rc.

**Instruction Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 37 34 | | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | O3 | Rc6 | Rb6 | Ra6 | 9Ah8 | Pn4 | Pc4 |

|  |  |  |
| --- | --- | --- |
| O3 | Assembler Mnemonic |  |
| 0 | STCMPBI | bytes incrementing |
| 1 | STCMPCI | characters incrementing |
| 2 | STCMPHI | half-word incrementing |
| 3 | STCMPWI | words incrementing |
| 4 | STCMPBD | bytes decrementing |
| 5 | STCMPCD | characters decrementing |
| 6 | STCMPHD | half-word decrementing |
| 7 | STCMPWD | word decrementing |

**Execution Units:** Memory

**Operation:**

temp = 0

while LC <> 0

mem[Rb + Rc] = mem[Ra + Rc]

Rc = Rc +/- amt

LC = LC – 1

### STFND – String Find

**Description:**

This instruction compares data from the memory location addressed by Ra plus Rc to the data in register Rb until the loop counter LC reaches zero or until a match occurs. Rc acts as an index and increments or decrements by the size of the operation as the move takes place. This instruction is interruptible. The data must be appropriately aligned. The loop counter is set to zero when a match occurs. The index of the match is contained in register Rc.

**Instruction Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 37 34 | | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | O3 | Rc6 | Rb6 | Ra6 | 9Bh8 | Pn4 | Pc4 |

|  |  |  |
| --- | --- | --- |
| O3 | Assembler Mnemonic |  |
| 0 | STFNDBI | bytes incrementing |
| 1 | SFNDCI | characters incrementing |
| 2 | STFNDHI | half-word incrementing |
| 3 | STFNDWI | words incrementing |
| 4 | STFNDBD | bytes decrementing |
| 5 | STFNDCD | characters decrementing |
| 6 | STFNDHD | half-word decrementing |
| 7 | STFNDWD | word decrementing |

**Execution Units:** Memory

**Operation:**

temp = 0

while LC <> 0

if (mem[Ra + Rc] = Rb)

stop

Rc = Rc +/- amt

LC = LC – 1

### STI – Store Immediate

**Description:**

A six bit value is zero extended to sixty-four bits and stored to memory. The memory address is the sum of the sign extended displacement and register Ra. The memory address must be word aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 37 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Seg3 | Displacement8..0 | Imm6 | Ra6 | 96h8 | Pn4 | Pc4 |

**Execution Units:** All ALU’s / Memory

**Operation:**

memory[Ra + displacement] = zero extend (Imm[5..0])

### STIX – Store Immediate Indexed

**Description:**

A ten bit value is zero extended to sixty-four bits and stored to memory. The memory address is the sum of register Ra and scaled register Rb. The memory address must be word aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 36 | 35 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Imm9..6 | Sc2 | Imm5..0 | Rb6 | Ra6 | C6h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

memory[Ra + Rb \* scale] = zero extend (Imm[9..0])

### STMV – String Move

**Description:**

This instruction moves a data from the memory location addressed by Ra plus Rc to the memory location addressed by Rb plus Rc until the loop counter LC reaches zero. Rc acts as an index and increments or decrements by the size of the operation as the move takes place. This instruction is interruptible. The data moved must be in the same segment and appropriately aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 37 34 | | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | O3 | Rc6 | Rb6 | Ra6 | 99h8 | Pn4 | Pc4 |

|  |  |  |
| --- | --- | --- |
| O3 | Assembler Mnemonic |  |
| 0 | STMVBI | move bytes incrementing |
| 1 | STMVCI | move characters incrementing |
| 2 | STMVHI | move half-word incrementing |
| 3 | STMVWI | move words incrementing |
| 4 | STMVBD | move bytes decrementing |
| 5 | STMVCD | move characters decrementing |
| 6 | STMVHD | move half-word decrementing |
| 7 | STMVWD | move word decrementing |

**Execution Units:** Memory

**Operation:**

temp = 0

while LC <> 0

mem[Rb + Rc] = mem[Ra + Rc]

Rc = Rc +/- amt

LC = LC – 1

### STP – Stop / Slow Down

**Description:**

This instruction controls the core clock rate which affects power consumption. The immediate constant is loaded into a shift register that controls the frequency of clock pulses seen by the processor. Setting the constant to FFFFh provides the maximum clock rate. Setting the constant to zero stops the clock completely. With the clock stopped completely the core must be reset or an NMI interrupt must occur before the core will continue processing. After reset or NMI the core begins processing at a half the maximum clock rate.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 16 | 15 8 | 7 0 | |
| Immediate16 | F6h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra[31:0]

**Typical Values For Shift Register**

|  |  |
| --- | --- |
| Value |  |
| 0000 | Stop clock completely |
| 8888 | 25% rate |
| AAAA | 50% rate |
| EEEE | 75% rate |
| FFFF | Full power, max clock rate |
|  |  |

**Exceptions:** none

### STSB – Store String Byte

**Description:**

This instruction stores a byte contained in register Rb to consecutive memory locations beginning at the address in Ra until the loop counter LC reaches zero. Ra is updated with by the number of bytes written. This instruction is interruptible.

**Instruction Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | 03 | ~6 | Rb6 | Ra6 | 98h8 | Pn4 | Pc4 |

**Execution Units:** Memory

**Operation:**

temp = 0

while LC <> 0

mem[Ra] = Rb[7:0]

Ra = Ra + 1

LC = LC – 1

### STSC – Store String Character

**Description:**

This instruction stores a character (16 bit value) to consecutive memory locations beginning at the address in Ra until the loop counter reaches zero. The memory address must be character aligned. Ra is updated by the number of bytes written. This instruction is interruptible.

**Instruction Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | 13 | ~6 | Rb6 | Ra6 | 98h8 | Pn4 | Pc4 |

**Execution Units:** Memory

**Operation:**

temp = 0

while LC <> 0

mem[Ra] = Rb[15:0]

Ra =Ra+ 2

LC = LC – 1

### STSH – Store String Half-word

**Description:**

This instruction stores a half-word (32 bit value) to consecutive memory locations beginning at the address in Ra until the loop counter reaches zero. The memory address must be half-word aligned. Ra is updated by the number of bytes written. This instruction is interruptible.

**Instruction Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | 23 | ~6 | Rb6 | Ra6 | 98h8 | Pn4 | Pc4 |

**Execution Units:** Memory

**Operation:**

temp = 0

while LC <> 0

mem[Ra] = Rb[31:0]

Ra =Ra + 4

LC = LC – 1

### STSW – Store String Word

**Description:**

This instruction stores a word (64 bit value) to consecutive memory locations beginning at the address in Ra until the loop counter reaches zero. The memory address must be half-word aligned. Ra is updated by the number of bytes written. This instruction is interruptible.

**Instruction Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | 33 | ~6 | Rb6 | Ra6 | 98h8 | Pn4 | Pc4 |

**Execution Units:** Memory

**Operation:**

temp = 0

while LC <> 0

mem[Ra] = Rb[63:0]

Ra =Ra + 8

LC = LC – 1

### SUB - Register-Register

**Description:**

This instruction subtracts one register from another and places the result into a third register. This instruction may cause an overflow exception.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 01h6 | Rt6 | Rb6 | Ra6 | 40h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra - Rb

### SUBI - Register-Immediate

**Description:**

This instruction subtracts an immediate value from a register and places the result into a register. This instruction may cause an overflow exception.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Immediate11..0 | Rt6 | Ra6 | 49h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra – Imm

### SUBU - Register-Register

**Description:**

This instruction subtracts one register from another and places the result into a third register. This instruction never causes an exception.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| 05h6 | Rt6 | Rb6 | Ra6 | 40h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra - Rb

### SUBUI - Register-Immediate

**Description:**

This instruction subtracts an immediate value from a register and places the result into a register. This instruction never causes an exception.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Immediate11..0 | Rt6 | Ra6 | 4Dh8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra – Imm

### SW – Store Word

**Description:**

A sixty-four bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended offset and register Ra. The memory address must be word aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rt6 | Ra6 | 93h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

memory[Ra+offset] = Rb

### SWCR – Store Word and Clear Reservation

**Description:**

If there is a reservation present on the memory address then a sixty-four bit value is stored to memory from the source register Rs and the reservation is cleared. If there is no reservation present then memory is not updated. If the update was successful then predicate register zero is set to ‘ne’ status, otherwise the predicate register is set to ‘eq’ status. The memory address is the sum of the sign extended offset and register Ra. The memory address must be word aligned. This instruction relies on the memory system for implementation.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Rs6 | Ra6 | 8Ch8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

memory[Ra+offset] = Rb, reservation cleared

### SWS – Store Word Special

**Description:**

A sixty-four bit value is stored to memory from the source special purpose register Spr. The memory address is the sum of the sign extended displacement and register Ra. The memory address must be word aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3937 | 36 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Sg3 | Displacement8..0 | Spr6 | Ra6 | 9Eh8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

memory[Ra + displacement] = Spr

### SWX – Store Word Indexed

**Description:**

A sixty-four bit value is stored to memory from the source register Rc. The memory address is the sum of register Ra and scaled register Rb. The memory address must be word aligned.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 37 | 36 | 3534 | 33 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| Seg3 | ~ | Sc2 | Rc6 | Rb6 | Ra6 | C3h8 | Pn4 | Pc4 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

memory[Ra+Rb] = Rc

### SXB – Sign Extend Byte

**Description:**

This instruction sign extends a register from bit 8 to 63 and places the result in a target register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| C4 | Rt6 | Ra6 | A7h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = {56{Ra[7]}},Ra[7:0]

**Exceptions:** none

### SXC – Sign Extend Character

**Description:**

This instruction sign extends a register from bit 16 to 63 and places the result in a target register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| D4 | Rt6 | Ra6 | A7h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = {48{Ra[15]}},Ra[15:0]

**Exceptions:** none

### SXH – Sign Extend Half-word

**Description:**

This instruction sign extends a register from bit 32 to 63 and places the result in a target register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| E4 | Rt6 | Ra6 | A7h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = {32{Ra[31]}},Ra[31:0]

**Exceptions:** none

### SYNC – Synchronization Barrier

**Description:**

All instructions before the SYNC command are completed before any following instructions are started. Note that this instruction has an effect even if the predicate is false; this does not affect the correct operation of the program, only performance is affected.

**Instruction Format:**

|  |  |  |
| --- | --- | --- |
| 15 8 | 7 0 | |
| F7h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Exceptions:** none

### SYS –Call system routine

**Description:**

This instruction calls a system function located as the sum of the offset times 16 plus code address register 12. The return address is stored in the EPC register (code address register #13).

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 24 | 23 20 | 19 16 | 15 8 | 7 0 | |
| Offset7..0 | Ch4 | Dh4 | A5h8 | Pn4 | Pc4 |

### TLB – TLB Command

**Description:**

The command is executed on the TLB unit. The command results are placed in internal TLB registers which can be read or written using TLB command instruction. If the operation is a read register operation then the register value is placed into Rt. If the operation is a write register operation, then the value for the register comes from Rb. Otherwise the Rb/Rt field in the instruction is ignored.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3130 | 29 24 | 23 16 | | 15 8 | 7 0 | |
| ~2 | Rb/Rt6 | Tn4 | Cmd4 | F0h8 | Pn4 | Pc4 |

**Clock Cycles:** 3

Tn4 – This field identifies which TLB register is being read or written.

|  |  |  |
| --- | --- | --- |
| Reg no. |  | Assembler |
| 0 | Wired | Wired |
| 1 | Index | Index |
| 2 | Random | Random |
| 3 | Page Size | PageSize |
| 4 | Virtual page | VirtPage |
| 5 | Physical page | PhysPage |
| 7 | ASID | ASID |
| 8 | Data miss address | DMA |
| 9 | Instruction miss address | IMA |
| 10 | Page Table Address | PTA |
| 11 | Page Table Control | PTC |

**TLB Commands**

|  |  |  |
| --- | --- | --- |
| Cmd | Description | Assembler |
| 0 | No operation |  |
| 1 | Probe TLB entry | TLBPB |
| 2 | Read TLB entry | TLBRD |
| 3 | Write TLB entry corresponding to random register | TLBWR |
| 4 | Write TLB entry corresponding to index register | TLBWI |
| 5 | Enable TLB | TLBEN |
| 6 | Disable TLB | TLBDIS |
| 7 | Read register | TLBRDREG |
| 8 | Write register | TLBWRREG |
| 9 | Invalidate all entries | TLBINV |

Probe TLB – The TLB will be tested to see if an address translation is present.

Read TLB – The TLB entry specified in the index register will be copied to TLB holding registers.

Write Random TLB – A random TLB entry will be written into from the TLB holding registers.

Write Indexed TLB – The TLB entry specified by the index register will be written from the TLB holding registers.

Disable TLB – TLB address translation is disabled so that the physical address will match the supplied virtual address.

Enable TLB – TLB address translation is enabled. Virtual address will be translated to physical addresses using the TLB lookup tables.

The TLB will automatically update the miss address registers when a TLB miss occurs only if the registers are zero to begin with. System software must reset the registers to zero after a miss is processed. This mechanism ensures the first miss that occurs is the one that is recorded by the TLB.

PageTableAddr – This is a scratchpad register available for use to store the address of the page table.

PageTableCtrl – This is a scratchpad register available for use to store control information associated with the page table.

### TST - Register Test Compare

**Description:**

The register test compare compares a register against the value zero and sets the predicate flags appropriately.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2322 | 21 16 | 15 12 | 11 8 | 7 0 | |
| 02 | Ra6 | 04 | Pt4 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Operation:**

if Ra < 0

Pt.lt = 1

else

Pt.lt = 0

if Ra = 0

Pt.eq = 1

else

Pt.eq = 0

Pt.ltu = 0

**Exceptions:** none

### ZXB – Zero Extend Byte

**Description:**

This instruction zero extends a register from bit 8 to 63 and places the result in a target register. This instruction is typically used to perform an unsigned load operation with the LVB instruction.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| C4 | Rt6 | Ra6 | A7h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra[7:0]

**Exceptions:** none

### ZXC – Zero Extend Character

**Description:**

This instruction zero extends a register from bit 16 to 63 and places the result in a target register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| D4 | Rt6 | Ra6 | A7h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra[15:0]

**Exceptions:** none

### ZXH – Zero Extend Half-word

**Description:**

This instruction zero extends a register from bit 32 to 63 and places the result in a target register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 22 | 21 16 | 15 8 | 7 0 | |
| E4 | Rt6 | Ra6 | A7h8 | Pn4 | Pc4 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra[31:0]

**Exceptions:** none

## Opcode Map

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | TST / FTST / FSTST | | | | | | | | | | | | | | | |
| 1x | CMP / FCMP / FSCMP | | | | | | | | | | | | | | | |
| 2x | CMPI | | | | | | | | | | | | | | | |
| 3x | BR | | | | | | | | | | | | | | | |
| 4x | {RR} | {R} |  |  |  |  | BITI | ADDUI | ADDI | SUBI | MULI | DIVI | ADDUI | SUBUI | MULUI | DIVUI |
| 5x | {logic} | MLO |  | ANDI | ORI | EORI |  |  | {shift} |  |  |  |  |  |  |  |
| 6x |  |  |  |  |  |  |  |  |  |  |  | \_2ADDUI | \_4ADDUI | \_8ADDUI | \_16ADDUI | LDI |
| 7x | NEG | NOT | MUX |  |  |  | {FMAC} | {double r} | {float rr} | {single r} |  |  |  |  |  |  |
| 8x | LB | LBU | LC | LCU | LH | LHU | LW | LFS | LFD |  |  | LVWAR | SWCR | LEA | LWS | LCL |
| 9x | SB | SC | SH | SW | SFS | SFD | STI | CAS | STS | STMV | STCMP | STFND |  | LDIS | SWS | CACHE |
| Ax | JSR | JSR | JSR | RTS | LOOP | SYS | INT | {R} | MFSPR | MTSPR | {bitfld} | MOVS | LVB | LVC | LVH | LVW |
| Bx | LBX | LBUX | LCX | LCUX | LHX | LHUX | LWX |  |  |  |  |  |  |  |  |  |
| Cx | SBX | SCX | SHX | SWX |  |  | STIX | INC | PUSH | PEA | POP | LINK | UNLINK |  |  |  |
| Dx |  |  |  |  |  |  | LW |  |  |  |  |  |  |  |  |  |
| Ex |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Fx | {TLB} | NOP | RTS | RTE | RTI | {BCD} | STP | SYNC | MEMSB | MEMDB | CLI | SEI |  |  |  | IMM |

{RR} Opcodes –Func6

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | ADD | SUB | MUL | DIV | ADDU | SUBU | MULU | DIVU | 2ADDU | 4ADDU | 8ADDU | 16ADDU |  |  |  |  |
| 1x | MIN | MAX |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

{logic} Opcodes – Func6

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | AND | OR | EOR | NAND | NOR | ENOR | ANDC | ORC |  |  |  |  |  |  |  |  |
| 1x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

{BCD} Opcodes – Func6

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | BCDADD | BCDSUB | BCDMUL |  |  |  |  |  |  |  |  |  |  |  |  |  |

{float -rr} Opcodes –Func6

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x |  |  |  |  |  |  |  | FCMP | FADD | FSUB | FMUL | FDIV |  |  |  |  |
| 1x |  |  |  |  |  |  |  | FCMPS | FADDS | FSUBS | FMULS | FDIVS |  |  |  |  |
| 2x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

77 - Double {R} Opcodes – Func4

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | FMOV |  | FTOI | ITOF | FNEG | FABS | FSIGN | FMAN | FNABS |  |  |  | FSTAT | FRM |  |  |

79 - Single {R} Opcodes – Func4

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | FMOVS |  | FTOIS | ITOFS | FNEGS | FABSS | FSIGNS | FMANS | FNABSS |  |  |  | FTX | FCX | FEX | FDX |

A7 {R} Opcodes – Func4

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | MOV | NEG | NOT | ABS | SGN | CNTLZ | CNTLO | CNTPOP | SXB | SXC | SXH | COM | ZXB | ZXC | ZXH |  |

41 {R} Opcodes – Func4

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | CPUID | REDOR | REDAND | PAR |  |  |  |  |  |  |  |  |  |  |  |  |