Table of Contents

[Considerations in the Design of DSD2’s Instruction Set. 2](#_Toc448248640)

[Segmentation 2](#_Toc448248641)

[Number of Registers 2](#_Toc448248642)

[Predicate Registers 2](#_Toc448248643)

[Immediate Prefixes 2](#_Toc448248644)

[Instruction Length 2](#_Toc448248645)

[16 Bit Compressed Instructions 4](#_Toc448248646)

[Detailed Instruction Set 6](#_Toc448248647)

[2ADD - Register-Register 6](#_Toc448248648)

[2ADDI - Register-Immediate 7](#_Toc448248649)

[4ADD - Register-Register 8](#_Toc448248650)

[4ADDI - Register-Immediate 9](#_Toc448248651)

[8ADD - Register-Register 10](#_Toc448248652)

[8ADDI - Register-Immediate 11](#_Toc448248653)

[16ADD - Register-Register 12](#_Toc448248654)

[16ADDI - Register-Immediate 13](#_Toc448248655)

[ABS – Absolute Value Register 14](#_Toc448248656)

[ADD - Register-Register 15](#_Toc448248657)

[ADDI - Register-Immediate 16](#_Toc448248658)

[ADDIS - Register-Shifted Immediate 17](#_Toc448248659)

[ADDO - Register-Register 18](#_Toc448248660)

[AND - Register-Register 19](#_Toc448248661)

[ANDC - Register-Register 20](#_Toc448248662)

[ANDI - Register-Immediate 21](#_Toc448248663)

[ANDIS - Register-Shifted Immediate 22](#_Toc448248664)

[Bcc- Conditional Relative Branch 23](#_Toc448248665)

[BFCHG – Bit-field Change 25](#_Toc448248666)

[BFCLR – Bit-field Clear 26](#_Toc448248667)

[BFEXT – Bit-field Extract 27](#_Toc448248668)

[BFEXTU – Bit-field Extract Unsigned 28](#_Toc448248669)

[BFINS – Bit-field Insert 29](#_Toc448248670)

[BFINSI – Bit-field Insert Immediate 30](#_Toc448248671)

[BFSET – Bit-field Set 31](#_Toc448248672)

[BITI – Test bits Register-Immediate 32](#_Toc448248673)

[CLI – Clear Interrupt Mask 33](#_Toc448248674)

[CMP Register-Register Compare 34](#_Toc448248675)

[CMPI Register-Immediate Compare 35](#_Toc448248676)

[LB – Load Byte 36](#_Toc448248677)

[LBU – Load Byte Unsigned 37](#_Toc448248678)

[LBUX – Load Byte Unsigned Indexed 38](#_Toc448248679)

[LBX – Load Byte Indexed 39](#_Toc448248680)

[LW – Load Word 40](#_Toc448248681)

[LDI - Load-Immediate 42](#_Toc448248682)

[MEMDB – Memory Data Barrier 43](#_Toc448248683)

[MEMSB – Memory Synchronization Barrier 44](#_Toc448248684)

[NAND - Register-Register 45](#_Toc448248685)

[NEG - Negate Register 46](#_Toc448248686)

[NOP – No Operation 47](#_Toc448248687)

[NOR - Register-Register 48](#_Toc448248688)

[NOT – Logical Not 49](#_Toc448248689)

[OR - Register-Register 50](#_Toc448248690)

[ORI - Register-Immediate 51](#_Toc448248691)

[ORIS - Register-Shifted Immediate 52](#_Toc448248692)

[RTS – Return from Subroutine 53](#_Toc448248693)

[SEI – Set Interrupt Mask 54](#_Toc448248694)

[SHL – Shift Left 55](#_Toc448248695)

[SHR – Shift Right 56](#_Toc448248696)

[SHRU – Shift Right Unsigned 57](#_Toc448248697)

[SUB - Register-Register 58](#_Toc448248698)

[SUBO - Register-Register 59](#_Toc448248699)

[SW – Store Word 60](#_Toc448248700)

[Opcode Map 61](#_Toc448248701)

## Considerations in the Design of DSD2’s Instruction Set.

Why develop yet another ISA ? The same justifications for developing something like RISC-V can be used as a basis for any open ISA. Competition is good. For DSD2 it was desired to offer inherent support for a segmented memory system, while at the same time simplifying some of the other aspects of RISC-V. This necessitates changes to the instruction set formats and programming model for the ISA.

### Comparison to RISC-V

Why not just use RISC-V ? RISC-V is very extendable and differences in the programming model compared to DSD2 could have been accommodated by developing a RISC-V extension, however code density would suffer.

RISC-V is optimized to the nth degree. A lot of thought was put into the design of the ISA. It tries to accommodate a broad spectrum of potential applications by providing many different optional ISA extensions. RISC-V has a lofty goal of being ubiquitous in nature. The architects envision RISC-V in use for everything from embedded controllers to powerful vector processors. An ideal of RISC-V being the development of binary compatible software libraries that can work on many different machines. DSD2 has the same lofty goals

Making an observation of nature and the animal kingdom, one size does not fit all very well. A single ISA used everywhere may be open to greater security risks. A malicious virus could impact a much greater range of software controlled devices when only a single ISA is in use.

The one-size fits all mentality arises from the observation that many processing cores are used outside of their original design domain. Micro-processors intended for print controllers for instance, found use in home computers. Just because something can be done, doesn’t mean that it should be done.

There are some short-comings to the RISC-V ISA. The base ISA is designed around a two read port, one write port architecture. The base ISA is designed for a minimal resource footprint, for use perhaps in embedded control applications. The ‘E’ version of the core may have only 16 registers present. Isn’t this really reflecting a different architecture ? For larger applications three or more read ports may be desirable. For instance three read ports are needed in order to support indexed addressing stores, the compare and swap operation and multiply-accumulate operations.

RISC-V allows for instruction set extensions in order to accommodate different classes of applications. Features “missing” from the base ISA can be added for specific application purposes at the expense of some code density. It is possible to extend RISC-V in almost any manner and that may be one of its future problems.

#### Instruction Length

RISC-V has a provision for variable length instructions in increments of a 16 bit parcel. RISC-V has supports many options for the instruction length. The encoding mechanism is reasonably done, but likely to be overkill for most applications. In RISC-V Encoding larger instruction formats impacts the remainder of the instruction format, particularly for register specification. DSD2’s provision for variable length instructions is simpler and there are correspondingly fewer choices available for instruction formats. For DSD2 two bits in the instruction opcode determine the instruction length.

Once the barrier of fixed sized instructions is broken some means (multiplexer) is required to support the variable instruction length.

#### Number of Registers

I’m of the opinion that it’s better to provide a fixed number of base architectural registers in an ISA design rather than allow it to vary. The number of registers present in an architecture has been something that’s defined different architectures. DSD2 is simpler than RISCV in that it must support a minimum of 32 general purpose integer registers.

The ‘E’ version of the RISCV core may have only 16 registers present. Isn’t this really reflecting a different architecture ?

For larger applications three or more read ports may be desirable. For instance three read ports are needed in order to support indexed addressing stores, the compare and swap operation and multiply-accumulate operations.

#### Segmentation

Segmented memory models are a feature of many ISA’s. Some support for a segmented memory model can be valuable at the operating system level. A segmented memory system isn’t that valuable at an applications level. Often different applications code, data, and stack areas are allocated to different memory segments by the OS. Unfortunately a segmented memory system impacts the ISA because instructions and registers are typically required in order to support it. Even instruction format may need to accommodate segmentation. RISCV had no support for segmentation. It appears that segmented memory model is being “bolted-on” to the RISCV architecture in supervisor mode. There is a provision for program base and limit registers in supervisor mode, which is the beginnings of a segmented model.

DSD2 ISA inherently supports segmentation. Segmented memory model registers are a part of the supervisory mode to the ISA. Space is reserved in load / store instructions in order to accommodate a segment register specification.

### Comparison to Thor

DSD2 represents a refinement of the Thor architecture which requires fewer logic resources. And hopefully offers better performance.

#### Segmentation

Segmentation adds a level of complexity to the instruction set. Segmentation must be designed in from the start. Instructions supporting segmentation must be planned. DSD2 does not support a segmented architecture.

#### Number of Registers

Thor uses the general purpose register array for both integer and floating point values and hence requires a larger number of general purpose registers. A handful of registers are also available only in kernel mode meaning more registers are needed for application or user mode. The bits used for the register spec are probably better used to increase the size of the immediate operand or as extra opcode bits. Hence DSD2 has fewer general purpose registers.

#### Predicate Registers

One important difference is the lack of predicate registers. This choice came about from realizing that predication isn’t that valuable until an entire basic block of statements can be predicated. For example eight or more instructions. The processor pipeline or queue size has to be fairly long for predication to have an effect, otherwise what happens is the use of predicated branch instructions combined with sets of instructions that always execute. This situation isn’t any better than non-predicated instructions with conditional branches. In the case of a shorter pipeline it’s probably better not to use predication as predication consumes memory and cache space.

Processing predicated instructions requires an extra argument to passed for every instruction. The argument being the unchanged value of the target register. The logic associated with passing this additional argument increased the size of the Thor core by about 20%. Not supporting predication should reduce the size of DSD2 correspondingly.

Taking the place of predicate registers are eight compare result registers. Limiting the result placement to eight registers allows shorter instructions to be used. Operation is similar to the PowerPC.

#### Immediate Prefixes

Thor uses immediate prefixes in order to support a full range of immediate values. This complicates pipeline design and impacts exception handling. A nice feature of prefixes is that they don’t require extra register usage for large constants. DSD2 doesn’t use prefix values. Instead DSD2 makes use of extra instructions to shift an immediate value before use. The wider instruction formats are useful for that purpose. For load / store operations if a large constant is required then it must be loaded into a register and indexed addressing used.

#### Instruction Length

In Thor’s ISA many instructions are five bytes long. This is partly due to the presence of a predicate byte in every instruction. Most instructions in DSD2 are shorter. Most architectures support some form of variable length instructions. The value of having shorter instructions in the cache is fairly high. Note the instructions must be able to do useful work. In Thor’s ISA instructions are variable in length but decoding for the length of an instruction isn’t that regular. DSD attempts to have fewer instruction lengths and to make it easier to tell the length of an instruction with minimal decoding.

It actually makes little difference whether instructions are byte-aligned, half-word aligned or word aligned once variable lengths are present. There’s a multiplexer involved in aligning instructions for variable length instructions. In terms of an FPGA a four to one multiplexer is often no more expensive than a two to one multiplexer.

Thor’s instruction finder for the second instruction has a nine to one multiplexer. This likely represents at least two levels of logic. The logic is further cascaded by the function to determine instruction length which is also several levels of logic.

|  |
| --- |
| Thor’s Second Instruction Finder |
| // Find the second instruction in the instruction line.  always @(insn)  case(fnInsnLength(insn))  4'd1: insn1a <= insn[71: 8];  4'd2: insn1a <= insn[79:16];  4'd3: insn1a <= insn[87:24];  4'd4: insn1a <= insn[95:32];  4'd5: insn1a <= insn[103:40];  4'd6: insn1a <= insn[111:48];  4'd7: insn1a <= insn[119:56];  4'd8: insn1a <= insn[127:64];  default: insn1a <= {8{8'h10}}; // NOPs  endcase |

Incorporating variable length instructions, DSD2 reduces this to a four-to-one multiplexer. One can’t have everything with a smaller multiplexer. So DSD2 supports the following instruction lengths: 16, 32, 48, and 64 bits.

|  |
| --- |
| DSD2’s Second Instruction Finder |
| // Find the second instruction in the instruction line.  always @(insn)  case(fnInsnLength(insn))  2'd0: insn1a <= insn[79:16];  2'd1: insn1a <= insn[95:32];  2'd2: insn1a <= insn[111:48];  2'd3: insn1a <= insn[127:64];  endcase |

The rationale behind this is that there aren’t that many eight bit instructions and they are only used infrequently. So the minimum parcel set to be processed is 16 bits.

Note that instructions that would otherwise be single byte instructions have the second byte of the instruction set to the NOP opcode value F1h. This is to provide for the eventuality that single byte opcodes are supported.

#### 16 Bit Compressed Instructions

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 15 | 11 | | 10 8 | | 7 2 | 1 0 |
| ADDI | I3 | | Rt5 | | | 000000b6 | I2 |
| ADDI SP | I8 | | | | | 000001b6 | I2 |
| ADDI BP | I8 | | | | | 001101b6 | I2 |
| LDI | I3 | | Rt5 | | | 110010b6 | I2 |
| ADD | Ra3 | | Rt5 | | | 110011b6 | 00 |
| AND | Ra3 | | Rt5 | | | 110011b6 | 10 |
| OR | Ra3 | | Rt5 | | | 110011b6 | 11 |
| BEQ | D5 | | | | P3 | 111000b6 | D2 |
| BNE | D5 | | | | P3 | 111001b6 | D2 |
| LW | D2 | Ra3 | | | Rt3 | 110100b6 | D2 |
| SW | D2 | Ra3 | | | Rs3 | 110101b6 | D2 |
| LW d[BP] | D3 | | Rt5 | | | 110110b6 | D2 |
| SW d[BP] | D3 | | Rs5 | | | 110111b6 | D2 |
| LW d[SP] | D3 | | Rt5 | | | 111010b6 | D2 |
| SW d[SP] | D3 | | Rs5 | | | 111011b6 | D2 |
| INT | I8 | | | | | 111101b6 | 0 I |
| RTI | 11110001 | | | | | 111101b6 | 10 |
| JAL | D8 | | | | | 111111b6 | D2 |

Rx3 Field

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| r8 | r9 | r10 | r11 | r12 | r13 | r14 | r15 |

All single byte instructions have bits four to seven of the instruction set to 1111. There aren’t that many instructions that are single byte in nature.

|  |  |
| --- | --- |
| Opcode | Opcode Size |
| 00xxxxxx | Twenty-four bit opcode |
| 0100xxxx | Forty-bit opcode |
| 11xxxxxx | Sixteen bit opcode |
| 1111xxxx | Sixteen bit Eight bit opcodes |
| All others | Thirty-two bit opcode |

|  |
| --- |
| DSD2’s Instruction Length Decode |
| function [1:0] fnInsnLength(isn);  input [39:0] isn;  casex(isn[7:4])  4'b00xx: fnInsnLength = 2'd1;  4'b0100: fnInsnLength = 2'd3;  4'b11xx: fnInsnLength = 2'd0;  default: fnInsnLength = 2'd2;  endcase  endfunction |

# Programming Model

|  |  |  |  |
| --- | --- | --- | --- |
| Reg | ABI Name |  | Saver |
| r0 | zero |  |  |
| r1 | ra | return address | caller |
| r2 | s0 / fp | frame pointer | callee |
| r3 | s1 |  | callee |
| r4 | s2 |  | callee |
| r5 | s3 |  | callee |
| r6 | s4 |  | callee |
| r7 | s5 |  | callee |
| r8 | s6 |  | callee |
| r9 | s7 |  | callee |
| r10 | s8 |  | callee |
| r11 | s9 |  | callee |
| r12 | s10 |  | callee |
| r13 | s11 |  | callee |
| r14 | sp | stack pointer | callee |
| r15 | tp | thread pointer | callee |
| r16 | v0 | return value #0 | caller |
| r17 | v1 | return value #1 | caller |
| r18 | a0 | function argument | caller |
| r19 | a1 |  |  |
| r20 | a2 |  |  |
| r21 | a3 |  |  |
| r22 | a4 |  |  |
| r23 | a5 |  |  |
| r24 | a6 |  |  |
| r25 | a7 |  |  |
| r26 | t0 | temporary |  |
| r27 | t1 |  |  |
| r28 | t2 |  |  |
| r29 | t3 |  |  |
| r30 | t4 |  |  |
| r31 | gp | global pointer |  |

# Detailed Instruction Set

### 2ADD - 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:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 168 | 7 0 |
| 0 | Rt5 | Rb5 | Ra5 | 2Ch8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra \* 2 + Rb

**Exceptions:** none

### 2ADDI - 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:**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Immediate13..0 | Rt5 | Ra5 | 6Ch8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra \* 2 + immediate

**Exceptions:** none

### 4ADD - Register-Register

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 168 | 7 0 |
| 0 | Rt5 | Rb5 | Ra5 | 2Dh8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra \* 4 + Rb

**Exceptions:** none

### 4ADDI - Register-Immediate

**Description:**

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

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Immediate13..0 | Rt5 | Ra5 | 6Dh8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra \* 4 + immediate

**Exceptions:** none

### 8ADD - Register-Register

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 168 | 7 0 |
| 0 | Rt5 | Rb5 | Ra5 | 2Eh8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra \* 8 + Rb

**Exceptions:** none

### 8ADDI - Register-Immediate

**Description:**

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

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Immediate13..0 | Rt5 | Ra5 | 6Eh8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra \* 8 + immediate

**Exceptions:** none

### 16ADD - Register-Register

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 168 | 7 0 |
| 0 | Rt5 | Rb5 | Ra5 | 2Fh8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra \* 16 + Rb

**Exceptions:** none

### 16ADDI - Register-Immediate

**Description:**

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

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Immediate13..0 | Rt5 | Ra5 | 6Fh8 |

**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:**

|  |  |  |  |
| --- | --- | --- | --- |
| 23 18 | 17 13 | 12 8 | 7 0 |
| 36 | Rt5 | Ra5 | 01h8 |

**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.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 2523 | 22 18 | 17 13 | 12 8 | 7 0 |
| 006 | 03 | Rt5 | Rb5 | Ra5 | 42h8 |

|  |  |  |  |
| --- | --- | --- | --- |
| 15 13 | 12 8 | 7 2 | 1 0 |
| Ra3 | Rt5 | 110011b6 | 00 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra + Rb

**Exceptions:** none

### ADDI - Register-Immediate

**Description:**

Add a register and immediate value and place the sum in the target register.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 47 18 | 17 13 | 12 8 | 7 0 |
| Immediate29..0 | Rt5 | Ra5 | 40h8 |

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Immediate13..0 | Rt5 | Ra5 | 52h8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 15 13 | 12 8 | 7 2 | 1 0 |
| ADDI | I3 | Rt5 | 000000b6 | I2 |
| ADDI SP | I8 | | 000001b6 | I2 |
| ADDI BP | I8 | | 001101b6 | I2 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra + immediate

**Exceptions:** none

Notes:

Adding to r0 is effectively a NOP operation.

### ADDIS - Register-Shifted Immediate

**Description:**

Add a register and shifted immediate value and place the sum in the target register. The immediate value is shifted left by 24 bits before the addition takes place.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 58 | 57 18 | 17 13 | 12 8 | 7 0 |
| ~6 | Immediate39..0 | Rt5 | Ra5 | E8h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra + immediate

**Exceptions:** none

### ADDO - Register-Register

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 2523 | 22 18 | 17 13 | 12 8 | 7 0 |
| 006 | 13 | Rt5 | Rb5 | Ra5 | 42h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra + Rb

**Exceptions:** integer overflow

### AND - Register-Register

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 2523 | 22 18 | 17 13 | 12 8 | 7 0 |
| 086 | ~3 | Rt5 | Rb5 | Ra5 | 42h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra & Rb

**Exceptions:** none

### ANDC - Register-Register

**Description:**

Bitwise and’s a registers and the compliment of a second register and places the result in a target register.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 0 | Rt5 | Rb5 | Ra5 | 07h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra & ~Rb

**Exceptions:** none

### ANDI - Register-Immediate

**Description:**

Bitwise and a register and immediate value and place the result in the target register. The immediate constant is sign extended before the operation.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 47 18 | 17 13 | 12 8 | 7 0 |
| Immediate29..0 | Rt5 | Ra5 | 48h8 |

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Immediate13..0 | Rt5 | Ra5 | 64h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra & immediate

**Exceptions:** none

### ANDIS - Register-Shifted Immediate

**Description:**

Bitwise and a register and shifted immediate value and place the sum in the target register. The immediate value is shifted left by 24 bits before the operation takes place. The low order bits of the shifted constant are set to one. This allows a 64 bit and operation to be performed by combining this instruction with a following ANDI instruction.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 58 | 57 18 | 17 13 | 12 8 | 7 0 |
| ~6 | Immediate39..0 | Rt5 | Ra5 | 41h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra & immediate

**Exceptions:** none

### Bcc- Conditional Relative Branch

**Description:**

A conditional branch is made relative to the address of the next instruction. There are 16 bit short form instructions for BEQ and BNE when the displacement can fit into 7 bits.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Displacement14..1 | Cnd5 | Ra5 | 52h8 |

**BEQS**

|  |  |  |
| --- | --- | --- |
| 15 11 | 10 8 | 7 2 |
| Disp6..2 | Pn3 | 111000b | D2 |

**BNES**

|  |  |  |
| --- | --- | --- |
| 15 11 | 10 8 | 7 2 |
| Disp6..2 | Pn3 | 111001b | D2 |

**Clock Cycles:** 1

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

**Operation:**

PC <= PC + displacement

**Exceptions:** none

**Branch Conditions**

|  |  |  |  |
| --- | --- | --- | --- |
| Cn4 |  | Test |  |
| 0 | BF | 0 | Always false – The branch never executes regardless of the compare register contents. |
| 1 | BT | 1 | Always True – The branch always executes regardless of the compare register contents. |
| 2 | BEQ | eq | Branches if compare register equals bit is set. |
| 3 | BNE | !eq | Branches if compare register equals bit is clear. |
| 4 | BLE | lt|eq | Less or Equal – compare less or equal flag is set |
| 5 | BGT | !(lt|eq) | greater than |
| 6 | BGE | !lt | greater or equal |
| 7 | BLT | lt | less than |
| 8 | BLEU | ltu|eq | unsigned less or equal |
| 9 | BGTU | !(ltu|eq) | unsigned greater than |
| 10 | BGEU  BOR | !ltu | unsigned greater or equal  Ordered for floating point |
| 11 | BLTU  BUN | ltu | unsigned less than  Unordered for floating point |
| 12 |  |  |  |
| 13 | BSIG | signal | branch if external signal is true |
| 14 |  |  |  |
| 15 |  |  |  |

### 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:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 30 29 24 | | 23 18 | 17 13 | 12 8 | 7 0 |
| 32 | me6 | mb6 | Rt5 | Ra5 | AAh8 |

**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:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 30 29 24 | | 23 18 | 17 13 | 12 8 | 7 0 |
| 22 | me6 | mb6 | Rt5 | Ra5 | AAh8 |

**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:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 30 29 24 | | 23 18 | 17 13 | 12 8 | 7 0 |
| 12 | me6 | mb6 | Rt5 | Ra5 | ABh8 |

**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:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 30 29 24 | | 23 18 | 17 13 | 12 8 | 7 0 |
| 02 | me6 | mb6 | Rt5 | Ra5 | ABh8 |

**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:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 30 29 24 | | 23 18 | 17 13 | 12 8 | 7 0 |
| 02 | me6 | mb6 | Rt5 | Ra5 | AAh8 |

**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:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 30 29 24 | | 23 18 | 17 13 | 12 8 | 7 0 |
| 22 | me6 | mb6 | Rt5 | Imm5 | ABh8 |

**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:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 30 29 24 | | 23 18 | 17 13 | 12 8 | 7 0 |
| 12 | me6 | mb6 | Rt5 | Ra5 | AAh8 |

**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 compare register. If the result of the ‘and’ operation is zero the register’s zero flag is set, otherwise it is cleared. If the result is negative the register’s less than flag is set, otherwise it is cleared.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 18 | 1716 | 15 13 | 12 8 | 7 0 |
| Immediate13..0 | ~2 | Pt3 | Ra6 | 54h8 |

**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

### CACHE – Cache Command

**Description:**

This instruction issues a command to the cache.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 23 18 | 17 13 | 12 8 | 7 0 |
| Func6 | ~5 | Ra5 | 1Bh8 |

**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 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.

The stack pointer cannot be used as the target register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 28 | 27 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| Displacement11..0 | Rt6 | Rc6 | Rb6 | Ra6 | 4Dh8 |

**Operation:**

Rt = memory [Ra + displacement]

if memory[Ra + displacement] = Rb

memory[Ra + displacement] = Rc

**Assembler:**

CAS Rt,Rb,Rc,offset[Ra]

### CHK – Register - Register

**Description:**

Check register against bounds. The comparisons are signed comparisons. If the register is inside the bounds, the target compare register equals flag is set, otherwise it is cleared.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 06 | Pt3 | Rc5 | Rb5 | Ra5 | 59h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

if (Ra < Rb or Ra >= Rc)

Pt.eq = 0

else

Pt.eq = 1

**Exceptions:** none

### CHKU – Register - Register

**Description:**

Check register against bounds. The comparisons are unsigned comparisons. If the register is inside the bounds, the target compare register equals flag is set, otherwise it is cleared.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 16 | Pt3 | Rc5 | Rb5 | Ra5 | 59h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

if (Ra < Rb or Ra >= Rc)

Pt.eq = 0

else

Pt.eq = 1

**Exceptions:** none

### CLI – Clear Interrupt Mask

**Description:**

This instruction is used to enable interrupts. This instruction is available only while operating in kernel mode. There are several delay cycles before this instruction takes effect. This is to allow code to run even if an interrupt line is stuck active.

**Instruction Format:**

|  |  |
| --- | --- |
| 15 8 | 7 0 |
| 90h8 | 33h8 |

**Clock Cycles:** 1

**Operation:**

im = 0

**Exceptions:** privilege violation

### CMP Register-Register Compare

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 2523 | 22 18 | 17 13 | 12 8 | 7 0 |
| 026 | ~3 | Rt5 | Rb5 | Ra5 | 42h8 |

**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 compare register as a result. Both a signed and unsigned comparison take place at the same time.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 47 18 | 17 13 | 12 8 | 7 0 |
| Immediate29..0 | Rt5 | Ra5 | 42h8 |

|  |  |  |  |
| --- | --- | --- | --- |
| 31 18 | 17 13 | 12 8 | 7 0 |
| Immediate13..0 | Rt5 | Ra5 | 68h8 |

**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

### CMPIL Register-Immediate Compare Lower

**Description:**

This instruction is present to allow 64 bit immediate comparisons to be performed when combined with a preceding CMPIU instruction.

If register Rt indicates a equals result then this instruction is executed, otherwise it is treated as a NOP operation and the target register remains unchanged. The register immediate compare equals instruction compares the low order 24 bits of a register to an immediate value and sets the flags in the target compare register as a result. Both a signed and unsigned comparison take place at the same time. Note that the number is sign extended by the most significant bit of Ra.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 47 42 | 41 18 | 17 13 | 12 8 | 7 0 |
| ~6 | Immediate23..0 | Rt5 | Ra5 | 42h8 |

**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

### CMPIU Register-Immediate Compare Upper

**Description:**

The register immediate compare shifted instruction compares the upper 40 bits of a register to an immediate value and sets the flags in the target compare register as a result. Both a signed and unsigned comparison take place at the same time.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 58 | 57 18 | 17 13 | 12 8 | 7 0 |
| ~6 | Immediate63..24 | Rt5 | Ra5 | 41h8 |

**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

### COM – Bitwise Compliment

**Description:**

This instruction performs a bitwise compliment 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:**

|  |  |  |  |
| --- | --- | --- | --- |
| 23 18 | 17 13 | 12 8 | 7 0 |
| B6 | Rt5 | Ra5 | 01h8 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = ~ Ra

**Exceptions:** none

### 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:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 0 | Rt5 | Rb5 | Ra5 | 1Eh8 |

**Clock Cycles:** 65

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra / Rb

**Exceptions**: divide by zero

### DIVU – Unsigned Register-Register Divide

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 1 | Rt5 | Rb5 | Ra5 | 1Eh8 |

**Clock Cycles:** 65

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra / Rb

**Exceptions:** none

### INT –Interrupt

**Description:**

This instruction calls a system function located as the sum of the zero extended offset times 16 plus the vector table address register. The return address is stored in the EPC register.

Note that this instruction is automatically invoked for hardware interrupt processing. The return address stored is the address of the interrupt instruction.

**Instruction Format:**

|  |  |  |
| --- | --- | --- |
| 15 8 | 7 1 | 0 |
| Offset8..1 | 0011000b8 | O |

### JAL - Jump And Link

**Description:**

A jump is made to the sum of the sign extended displacement supplied in the displacement field of the instruction and the program counter.

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

**Instruction Formats:**

|  |  |  |
| --- | --- | --- |
| 47 13 | 12 8 | 7 0 |
| Offset34..0 | Rt5 | 4Bh8 |

|  |  |  |  |
| --- | --- | --- | --- |
| 31 | 13 | 12 8 | 7 0 |
| Offset18..0 | | Rt5 | 6Bh8 |

**Instruction Formats – ra implied:**

|  |  |  |
| --- | --- | --- |
| 15 8 | 7 2 | 1 0 |
| Disp8 | 001110b6 | D2 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt[t] = pc

pc = pc + displacement

**Exceptions:** none

### JALR - Jump And Link Register

**Description:**

A jump is made to the contents of register Ra. The subroutine return address is stored in the register specified in the Rt field of the instruction. Typically register #1 is used.

**Instruction Formats:**

|  |  |  |  |
| --- | --- | --- | --- |
| 23 18 | 17 13 | 12 8 | 7 0 |
| 3Fh6 | Rt5 | Ra5 | 01h8 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt[t] = pc

pc = Ra[n]

**Exceptions:** none

### 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 displacement and register Ra.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4745 | 39 18 | 17 13 | 12 8 | 7 0 |
| Sg3 | Displacement26..0 | Rt5 | Ra5 | 80h8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3129 | 28 18 | 17 13 | 12 8 | 7 0 |
| Sg3 | Displacement10..0 | Rt5 | Ra5 | 80h8 |

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

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

**Operation:**

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

**Exceptions:** DBE, DBG, TLB

### 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:**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 18 | 17 13 | 12 8 | 7 0 |
| Displacement13..0 | Rt5 | Ra5 | 81h8 |

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

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

**Operation:**

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

**Exceptions:** DBE, DBG, TLB

### 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 register Rb.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 0 | Rt5 | Rb5 | Ra5 | 10h8 |

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

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

**Operation:**

Rt = mem[Ra+Rb]

**Exceptions:** DBE, DBG, TLB

### LBX – Load Byte Indexed

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 | 2423 | 22 18 | 17 13 | 12 8 | 7 0 |
| 1 | ~ | Sc2 | Rt5 | Rb5 | Ra5 | 10h8 |

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

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

**Operation:**

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

**Exceptions:** DBE, DBG, TLB

### LDI - Load-Immediate

**Description:**

This instruction loads a sign extended immediate constant into a register. Note to load a larger value into a register two instructions must be used. One of the arithmetic or logical instructions could be used along with their shifted immediate forms.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 15 13 | 12 8 | 7 2 | 10 |
| I3 | Rt5 | 000010b6 | I2 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = immediate

### LEA – Load Effective Address

**Description:**

The memory address is placed in the target register. The memory address is the sum of the sign extended displacement and register Ra. This is an alternate form of the ADDI instruction where the operand is specified in a memory operand format.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 39 18 | 17 13 | 12 8 | 7 0 |
| Immediate21..0 | Rt5 | Ra5 | 40h8 |

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Immediate13..0 | Rt5 | Ra5 | 52h8 |

|  |  |  |
| --- | --- | --- |
| 23 14 | 13 8 | 7 0 |
| Immediate10..0 | Rt5 | 0Ch8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 15 13 | 12 8 | 7 2 | 1 0 |
| ADDI | I3 | Rt5 | 110000b6 | I2 |
| ADDI SP | I8 | | 110001b6 | I2 |

**Operation:**

Rt = Ra + Displacement

**Execution Units:** All ALU’s

### LEAX – Load Effective Address Indexed

**Description:**

A memory address is computed and placed in the target register. The address is the sum of register Ra and scaled register Rb. This mnemonic is an alternate form of the ADD, \_2ADD,\_4ADD or \_8ADD instruction. The assembler will emit the add instruction according to the scale specified for Rb.

**Clock Cycles: 1**

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

**Operation:**

Rt = Ra + Rb \* scale

**Exceptions:** none

### LH – Load Half-Word

**Description:**

A sixteen 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:**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 18 | 17 13 | 12 8 | 7 0 |
| Displacement13..0 | Rt5 | Ra5 | 82h8 |

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

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

**Operation:**

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

**Exceptions:** DBE, DBG, TLB

### LHU – Load Half-word Unsigned

**Description:**

A sixteen 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:**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 18 | 17 13 | 12 8 | 7 0 |
| Displacement13..0 | Rt5 | Ra5 | 83h8 |

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

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

**Operation:**

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

**Exceptions:** DBE, DBG, LMT, TLB

### LHUX – Load Half-word Unsigned Indexed

**Description:**

A sixteen 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:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 0 | Rt5 | Rb5 | Ra5 | 11h8 |

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

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

**Operation:**

Rt = mem[Ra+Rb\*scale]

**Exceptions:** DBE, DBG, LMT, TLB

### LHX – Load Half-word 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 half-word aligned.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 1 | Rt5 | Rb5 | Ra5 | 11h8 |

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

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

**Operation:**

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

**Exceptions:** DBE, DBG, LMT, TLB

### LW – Load Word

**Description:**

A thirty-two 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.

There is a 16 bit form for this instruction where the target register is loaded relative to the stack pointer register which is implied by the instruction. The displacement field is shifted left twice before use in that case.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 18 | 17 13 | 12 8 | 7 0 |
| Displacement13..0 | Rt5 | Ra5 | 84h8 |

|  |  |  |  |
| --- | --- | --- | --- |
| 39 18 | 17 13 | 12 8 | 7 0 |
| Displacement21..0 | Rt5 | Ra5 | 4Eh8 |

**Instruction Format – SP implied:**

|  |  |  |  |
| --- | --- | --- | --- |
| 15 13 | 12 8 | 7 2 | 1 0 |
| Disp4..2 | Rt4..2 | 111010b6 | D2 |

**Instruction Format – BP implied:**

|  |  |  |  |
| --- | --- | --- | --- |
| 15 13 | 12 8 | 7 2 | 1 0 |
| Disp4..2 | Rt4..2 | 110110b6 | D2 |

**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]

**Exceptions:** DBE, DBG, TLB

### MEMDB – Memory Data Barrier

**Description:**

All memory accesses before the MEMDB command are completed before any memory accesses after the data barrier are started.

**Instruction Format:**

|  |  |
| --- | --- |
| 15 8 | 7 0 |
| 82h8 | 33h8 |

**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.

**Instruction Format:**

|  |  |
| --- | --- |
| 15 8 | 7 0 |
| 83h8 | 33h8 |

**Clock Cycles:** 1

**Execution Units:** Memory

### MF – Move From Special Register-Register

**Description:**

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

**Instruction Format:**

|  |  |  |
| --- | --- | --- |
| 23 13 | 12 8 | 7 0 |
| Spr11 | Rt5 | A8h8 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Spr[n]

**Special Purpose Registers**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reg # | R/W |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| 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.

### NAND - Register-Register

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 2523 | 22 18 | 17 13 | 12 8 | 7 0 |
| 0C6 | ~3 | Rt5 | Rb5 | Ra5 | 42h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’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:**

|  |  |  |  |
| --- | --- | --- | --- |
| 23 18 | 17 13 | 12 8 | 7 0 |
| 16 | Rt5 | Ra5 | 01h8 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = - Ra

### NOP – No Operation

**Description:**

NOP performs no operation. The NOP operation is not queued by the processing core and is not present in the pipeline.

**Instruction Format:**

|  |  |  |
| --- | --- | --- |
|  | 15 8 | 7 0 |
| NOP | E18 | 338 |
| NOP | EA8 | 338 |

**Clock Cycles:** 1

**Execution Units:** None

**Operation:**

<none>

**Exceptions:** none

Notes:

### NOR - Register-Register

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 2523 | 22 18 | 17 13 | 12 8 | 7 0 |
| 0D6 | ~3 | Rt5 | Rb5 | Ra5 | 42h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’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:**

|  |  |  |  |
| --- | --- | --- | --- |
| 23 18 | 17 13 | 12 8 | 7 0 |
| 26 | Rt5 | Ra5 | 01h8 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = ! Ra

**Exceptions:** none

### OR - Register-Register

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 2523 | 22 18 | 17 13 | 12 8 | 7 0 |
| 016 | ~3 | Rt5 | Rb5 | Ra5 | 42h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra | Rb

**Exceptions:** none

### ORI - Register-Immediate

**Description:**

Bitwise or a register and immediate value and place the result in the target register. The immediate constant is sign extended before the operation.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 47 18 | 17 13 | 12 8 | 7 0 |
| Immediate29..0 | Rt5 | Ra5 | 49h8 |

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Immediate13..0 | Rt5 | Ra5 | 65h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra | immediate

**Exceptions:** none

### ORIS - Register-Shifted Immediate

**Description:**

Bitwise or a register and shifted immediate value and place the sum in the target register. The immediate value is shifted left by 24 bits before the operation takes place. The low order bits of the shifted constant are set to zero. This allows a 64 bit and operation to be performed by combining this instruction with a following ORI instruction.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 58 | 57 18 | 17 13 | 12 8 | 7 0 |
| ~6 | Immediate39..0 | Rt5 | Ra5 | E9h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra | immediate

**Exceptions:** none

### RTS – Return from Subroutine

**Description:**

The program counter is loaded with the value contained in the return address register r1.

**Instruction Formats:**

|  |  |
| --- | --- |
| 15 8 | 7 0 |
| 00h8 | 33h8 |

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

**Operation:**

PC = r1

**Exceptions:** none

### SEI – Set Interrupt Mask

**Description:**

The interrupt mask is set, disabling maskable interrupts. This instruction is available only in kernel mode.

**Instruction Format:**

|  |  |
| --- | --- |
| 15 8 | 7 0 |
| 91h8 | 33h8 |

**Clock Cycles:** 1

**Operation:**

im = 1

**Exceptions:** none

### 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:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 0 | Rt5 | Rb5 | Ra5 | 2Ah8 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra << Rb

**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:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 1 | Rt5 | Rb5 | Ra5 | 2Bh8 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra >> Rb

**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:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 0 | Rt5 | Rb5 | Ra5 | 2Bh8 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra >> Rb

**Exceptions:** none

### SUB - Register-Register

**Description:**

Subtract two registers and place the difference in the target register.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 2523 | 22 18 | 17 13 | 12 8 | 7 0 |
| 016 | 03 | Rt5 | Rb5 | Ra5 | 42h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra - Rb

**Exceptions:** none

### SUBO - Register-Register

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 2523 | 22 18 | 17 13 | 12 8 | 7 0 |
| 016 | 13 | Rt5 | Rb5 | Ra5 | 42h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra - Rb

**Exceptions:** integer overflow

### 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.

There is a 16 bit form for this instruction where the source register is stored relative to the stack pointer register which is implied by the instruction. The displacement field is shifted left twice before use in that case.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 18 | 17 13 | 12 8 | 7 0 |
| Displacement13..0 | Rt5 | Ra5 | 92h8 |

|  |  |  |  |
| --- | --- | --- | --- |
| 39 18 | 17 13 | 12 8 | 7 0 |
| Displacement21..0 | Rt5 | Ra5 | 4Fh8 |

|  |  |  |  |
| --- | --- | --- | --- |
| 23 18 | 17 13 | 12 8 | 7 0 |
| Disp6 | Rt5 | Ra5 | 0Fh8 |

**Instruction Format – SP implied:**

|  |  |  |  |
| --- | --- | --- | --- |
| 15 13 | 12 8 | 7 2 | 1 0 |
| Disp4..2 | Rb5 | 111010b6 | D2 |

**Instruction Format – BP implied:**

|  |  |  |  |
| --- | --- | --- | --- |
| 15 13 | 12 8 | 7 2 | 1 0 |
| Disp4..2 | Rb5 | 110110b6 | D2 |

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

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

**Operation:**

memory[Ra+offset] = Rb

**Exceptions**: DBE, DBG, TLB

### SYNC – Synchronization Barrier

**Description:**

All instructions before the SYNC command are completed before any following instructions are started.

**Instruction Format:**

|  |  |
| --- | --- |
| 15 8 | 7 0 |
| 80h8 | 33h8 |

**Clock Cycles:** 1

**Exceptions:** none

### XNOR - Register-Register

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 | 22 18 | 17 13 | 12 8 | 7 0 |
| 1 | Rt5 | Rb5 | Ra5 | 06h8 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = ~(Ra ^ Rb)

**Exceptions:** none

### XOR - Register-Register

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 2523 | 22 18 | 17 13 | 12 8 | 7 0 |
| 0A6 | 03 | Rt5 | Rb5 | Ra5 | 42h8 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra ^ Rb

**Exceptions:** none

### XORI - Register-Immediate

**Description:**

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

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 47 18 | 17 13 | 12 8 | 7 0 |
| Immediate29..0 | Rt5 | Ra5 | 4Ah8 |

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Immediate13..0 | Rt5 | Ra5 | 66h8 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra ^ immediate

**Exceptions:** none

### XORIS - Register-Shifted Immediate

**Description:**

Bitwise exclusive or a register and shifted immediate value and place the sum in the target register. The immediate value is shifted left by 24 bits before the operation takes place. The low order bits of the shifted constant are set to zero. This allows a 64 bit and operation to be performed by combining this instruction with a following XORI instruction.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 58 | 57 18 | 17 13 | 12 8 | 7 0 |
| ~6 | Immediate39..0 | Rt5 | Ra5 | EAh8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra | immediate

**Exceptions:** none

## Opcode Map

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 4x24 | LDI | {R} | ADD / ADDO | SUB / SUBO | AND /NAND | OR /NOR | XOR / XNOR | ANDC / ORC | ANDI | ORI | XORI | NOP | ADDI |  | LW | SW |
| 5x24 | LBUX/ LBX | LHUX / LHX | LWX |  |  |  |  |  | SBX | SHX | SWX | CACHE |  | MUL / MULU | DIV / DIVU | Mod / MODU |
| 6x24 | CMPI | | | | | | | | CMP |  | SHL | SHR /SHRU | 2ADD | 4ADD | 8ADD | 16ADD |
| 7x24 | BF | BT / BRA | BEQ | BNE | BLE | BGT | BGE | BLT | BLEU | BGTU | BGEU | BLTU |  | BSIG |  |  |
| Cx40 | ADDI | ADDIS | CMPI |  | ANDIS | ORIS | XORIS |  | ANDI | ORI | XORI | JAL |  | CAS | LW | SW |
| Dx40 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Ex40 | LB | LBU | LC | LCU | LH | LHU | LW |  |  |  |  |  |  |  |  |  |
| Fx40 | SB | SC | SH | SW |  |  |  |  |  |  |  |  |  |  |  |  |
| 8x32 | LB | LBU | LC | LCU | LH | LHU | LW | LWAR |  |  |  | LVWAR | SWCR | JSRI | LWS | LCL |
| 9x32 | LB | LBU | LC | LCU | LH | LHU | LW |  |  |  |  |  |  |  |  |  |
| Ax32 | SB | SC | SH | SW | SWCR |  |  |  | MFSPR | MTSPR | {bitfld} | {bitfld} | LVB | LVC | LVH | LVW |
| Bx32 | SB | SC | SH | SW |  |  |  |  | LLAX |  |  |  |  |  |  |  |
| 0x16 | ADDI | | | | ADDI SP | | | | LDI | | | | ADD |  | AND | OR |
| 1x16 | LW | | | | SW | | | | LW d[BP] | | | | SW d[BP] | | | |
| 2x16 | BEQ | | | | BNE | | | | LW d[SP] | | | | SW d[SP] | | | |
| 3x16 | INT | | RTS / RTI / NP | SYNC / MEMxB | ADDI BP | | | | CLI / SEI |  |  |  | JAL | | | |

## Opcode Map

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x8 | BRK |  |  | INT3 | SYNC | MEMSB | MEMDB |  | ZS | DS | ES | FS | GS | HS | SS | CS |
| 1x8 | RTS | RTI | RTE | RTD | CLI | SEI | NOP |  | JAL | | | |  |  |  |  |
| 2x16 | ADDI | | | | ADDI SP | | | | LDI | | | | ADD |  | AND | OR |
| 3x16 | LW | | | | SW | | | | LW d[BP] | | | | SW d[BP] | | | |
| 4x16 | BEQ | | | | BNE | | | | LW D[SP] | | | | SW D[SP] | | | |
| 5x16 | INT | |  |  | ADDI BP | | | | CMPI | | | | | | | |
| 6x24 | LDI | LBU | LC | LCU | LH | LHU | LW |  |  |  |  |  |  |  |  |  |
| 7x24 | SB | SC | SH | SW |  |  |  |  |  |  |  |  |  |  |  |  |
| 8x32 | LB | LBU | LC | LCU | LH | LHU | LW | LWAR |  |  |  | LVWAR | SWCR | JSRI | LWS | LCL |
| 9x32 | LB | LBU | LC | LCU | LH | LHU | LW |  |  |  |  |  |  |  |  |  |
| Ax32 | SB | SC | SH | SW | SWCR |  |  |  | MFSPR | MTSPR | {bitfld} | {bitfld} | LVB | LVC | LVH | LVW |
| Bx32 | SB | SC | SH | SW |  |  |  |  | LLAX |  |  |  |  |  |  |  |
| 0x16 | ADDI | | | | ADDI SP | | | | LDI | | | | ADD |  | AND | OR |
| 1x16 | LW | | | | SW | | | | LW d[BP] | | | | SW d[BP] | | | |
| 2x16 | BEQ | | | | BNE | | | | LW d[SP] | | | | SW d[SP] | | | |
| 3x16 | INT | | RTS / RTI / NP | SYNC / MEMxB | ADDI BP | | | | CLI / SEI |  |  |  | JAL | | | |

## Opcode Map

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x16 | ADDI / NOP | | | | ADDI SP | | | | LDI | | | | ADD |  | AND | OR |
| 1x16 | LW | | | | SW | | | | LW d[BP] | | | | SW d[BP] | | | |
| 2x16 | BEQ | | | | BNE | | | | LW D[SP] | | | | SW D[SP] | | | |
| 3x16 | INT | |  | {33xx} | ADDI BP | | | | JAL | | | |  |  |  |  |
| 4x32 | ADDI | CMPI | {RR} |  |  |  |  |  | ANDI | ORI | XORI | BITI |  |  |  |  |
| 5x32 |  |  | Bcc |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 6x32 | LB | LBU | LC | LCU | LH | LHU | LW | LVWAR |  |  |  |  |  |  |  |  |
| 7x32 | SB | SC | SH | SW |  |  |  | SWCR |  |  |  |  |  |  |  |  |
| 8x48 | ADDI | CMPI | CMPIL |  |  |  |  |  | ANDI | ORI | XORI | BITI |  | JSRI | LWS | LCL |
| 9x48 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Ax48 | LB | LBU | LC | LCU | LH | LHU | LW | LWAR | MFSPR | MTSPR | {bitfld} | {bitfld} | LVB | LVC | LVH | LVW |
| Bx48 | SB | SC | SH | SW |  |  |  | SWCR | LLAX |  |  |  |  |  |  |  |
| Cx64 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Dx64 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Ex64 | ADDIS | CMPIU |  |  |  |  |  |  | ANDIS | ORIS | XORIS |  |  |  |  |  |
| Fx64 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 33x0 | 33x1 | 33x2 | 33x3 | 33x4 | 33x5 | 33x6 | 33x7 | 33x8 | 33x9 | 33xA | 33xB | 33xC | 33xD | 33xE | 33xF |
| 330x | RTS | RTI | RTD | RTE |  |  |  |  |  |  |  |  |  |  |  |  |
| 338x | SYNC |  | MEMDB | MEMSB |  |  |  |  |  |  |  |  |  |  |  |  |
| 339x | CLI | SEI |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 33Ex |  | NOP |  |  | MRK1 | MRK2 | MRK3 | MRK4 |  |  | NOP |  |  |  |  |  |