The MIPS-X project has been supported by the Defense Advanced Research Projects Agency under contract MDA903-83-C-0335. Paul Chow was partially supported by a Postdoctoral Fellowship from the Natural Sciences and Engineering Research Council of Canada.
MIPS-X Instruction Set
and Programmer's Manual

Paul Chow

Technical Report No. 86-289
May 1986

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract

MIPS-X is a high performance second generation reduced instruction set microprocessor. This document describes the visible architecture of the machine, the basic timing of the instructions, and the instruction set.

Keywords: MIPS-X processor, RISC, processor architecture, streamlined instruction set.
Copyright © 1986 Stanford University
# Table of Contents

1. Introduction

2. Architecture
   2.1. Memory Organization
   2.2. General Purpose Registers
   2.3. Special Registers
   2.4. The Processor Status Word
      2.4.1. Trap on Overflow
   2.5. Privilege Violations

3. Instruction Timing
   3.1. The Instruction Pipeline
   3.2. Delays and Bypassing
   3.3. Memory Instruction Interlocks
   3.4. Branch Delays
   3.5. Jump Delays
   3.6. Detailed Instruction Timings
      3.6.1. Notation
      3.6.2. A Normal Instruction
      3.6.3. Memory Instructions
      3.6.4. Branch Instructions
      3.6.5. Compute Instructions
         3.6.5.1. Special Instructions
      3.6.6. Jump Instructions
      3.6.7. Multiply Step $\times mstep$
      3.6.8. Divide Step $\div dstep$

4. Instruction Set
   4.1. Notation
   4.2. Memory Instructions
      4.2.1. Id - Load
      4.2.2. st - Store
      4.2.3. Idf - Load Floating Point
      4.2.4. stf - Store Floating Point
      4.2.5. Idt - Load Through
      4.2.6. stt - Store Through
      4.2.7. movfrc - Move From Coprocessor
      4.2.8. movtoc - Move To Coprocessor
      4.2.9. aluc - Coprocessor ALU
   4.3. Branch Instructions
      4.3.1. beq - Branch If Equal
      4.3.2. bge - Branch If Greater than or Equal
      4.3.3. bhs - Branch If Higher Or Same
      4.3.4. blo - Branch If Lower Than
      4.3.5. blt - Branch If Less Than
      4.3.6. bne - Branch If Not Equal
   4.4. Compute Instructions
      4.4.1. add - Add
      4.4.2. dstep - Divide Step
      4.4.3. mstart - Multiply Startup
      4.4.4. mstep - Multiply Step
      4.4.5. sub - Subtract
4.4.6. subnc - Subtract with No Carry In
4.4.7. and - Logical And
4.4.8. bic - Bit Clear
4.4.9. not - Ones Complement
4.4.10. or - Logical Or
4.4.11. xor - Exclusive Or
4.4.12. mov - Move Register to Register
4.4.13. asr - Arithmetic Shift Right
4.4.14. rolb - Rotate Left by Bytes
4.4.15. roticb - Rotate Left Complemented by Bytes
4.4.16. sh - Shift
4.4.17. nop - No Operation

4.5. Compute Immediate Instructions
4.5.1. addi - Add Immediate
4.5.2. jpc - Jump PC
4.5.3. jpcrs - Jump PC and Restore State
4.5.4. jspci - Jump Indexed and Store PC
4.5.5. movfrs - Move from Special Register
4.5.6. movtos - Move to Special Register
4.5.7. trap - Trap Unconditionally
4.5.8. hsc - Halt and Spontaneously Combust

Appendix I. Some Programming Issues
Appendix II. Opcode Map
   II.1. OP Field Bit Assignments
   II.2. Comp Func Field Bit Assignments
   II.3. Opcode Map of All Instructions
Appendix III. Floating Point Instructions
   III.1. Format
   III.2. Instruction Timing
   III.3. Load and Store Instructions
   III.4. Floating Point Compute Instructions
   III.5. Opcode Map of Floating Point Instructions
Appendix IV. Integer Multiplication and Division
   IV.1. Multiplication and Division Support
   IV.2. Multiplication
   IV.3. Division
Appendix V. Multiprecision Arithmetic
Appendix VI. Exception Handling
   VI.1. Interrupts
   VI.2. Trap On Overflow
   VI.3. Trap Instructions
Appendix VII. Assembler Macros and Directives
   VII.1. Macros
      VII.1.1. Branches
      VII.1.2. Shifts
      VII.1.3. Procedure Call and Return
   VII.2. Directives
   VII.3. Example
   VII.4. Grammar
List of Figures

Figure 2-1: Word Numbering in Memory                       3
Figure 2-2: Bit and Byte Numbering in a Word              3
Figure 2-3: The Processor Status Word                    5
Figure 3-1: Pipeline Sequence                            7
Figure III-1: Floating Point Number Format               73
Figure IV-1: Signed Integer Multiplication               77
Figure IV-2: Signed Integer Division                     79
Figure VI-1: Interrupt Sequence                          84
Figure VI-2: Trap Sequence                              86
List of Tables

Table 3-1:  MIPS-X Pipeline Stages  7
Table 3-2:  Delay Slots for MIPS-X Instruction Pairs  9
Table 4-1:  Branch Instructions  32
Table IV-1: Number of Cycles Needed to do a Multiplication  78
Table IV-2: Number of Cycles Needed to do a Divide  78
1. Introduction

This manual describes the visible architecture of the MIPS-X processor and the timing information required to execute correct programs. MIPS-X is a pipelined processor that has no hardware interlocks. Therefore, the software system is responsible for keeping track of the timing of the instructions.

The processor has a load/store architecture and supports a very small number of instructions. The instruction set of the processor will be described.

The processor supports two types of coprocessor interfaces. One interface is dedicated to the floating point unit (FPU) and the other will support up to 7 other coprocessors. These instructions will also be described.
2. Architecture

2.1. Memory Organization

The memory is composed of 32-bit words and it is a uniform address space starting at 0 and ending at $2^{32}-1$. Each memory location is a byte. Load/store addresses are manipulated as 32-bit byte addresses on-chip but only words can be read from memory (i.e., only the top 30 bits are sent to the memory system). The numbering of words in memory is shown in Figure 2-1. Bytes (characters) are accessed by sequences of instructions that can do insertion or extraction of characters into or from a word. (See Appendix I). Instructions that affect the program counter, such as branches and jumps, generate word addresses. This means that the offsets used for calculating load/store addresses are byte offsets, and displacements for branches and jumps are word displacements. The addressing is consistently Big Endian [1].

Bytes are numbered starting with the most significant byte at the most significant bit end of the word The bits in a word are numbered 0 to 31 starting at the most significant bit (MSB) and going to the least significant bit (LSB). Bit and byte numbering are shown in Figure 2-2.

![Figure 2-1: Word Numbering in Memory](image)

Figure 2-1: Word Numbering in Memory

![Figure 2-2: Bit and Byte Numbering in a Word](image)

Figure 2-2: Bit and Byte Numbering in a Word

The address space is divided into system and user space. An address with the high order bit (bit 0) set to one (1) will access user space. If the high order bit is zero (0) then a system space address is accessed. Programs executing in user space cannot access system space. Programs executing in system space can access both system and user space.

2.2. General Purpose Registers

There are 32 general purpose registers (GPRs) numbered 0 through 31. These are the registers named in the register fields of the instructions. All registers are 32 bits. Of these registers, one register is not general purpose. Register 0 (r-0) contains the constant 0 and thus cannot be changed. The constant 0 is used very frequently so it is the value that is
stored in the constant register. A constant register has one added advantage. One register is needed as a void
destination for instructions that do no writes or instructions that are being noped because they must be stopped for some
reason. This is implemented most easily by writing to a constant location.

2.3. Special Registers

There are several special registers that can be accessed with the Move Special instructions. They are:

**PSW**
The processor status word. This is described in more detail in Section 2.4.

**PC-4, PC-1**
Locations in the PC chain used for saving and restoring the state of the PC chain.

**MD**
The **mul/div** register. This is a special register used during multiplication and division.

2.4. The Processor Status Word

The Processor Status Word (**PSW**) holds some of the information pertaining to the current state of the machine. The
PSW actually contains two sets of bits that are called **PSWCurrent** and **PSWOther**. The current state of the machine is
always reflected in **PSWCurrent**. When an exception or trap occurs, the contents of **PSWCurrent** are copied into
**PSWOther**. The **e** bit is not saved **PSWOther** then contains the processor state from before the exception or trap so that
it can be saved. Interrupts are disabled, PC shifting is disabled, overflows are masked and the processor is put into
system state. The **I** bit is cleared if the exception was an interrupt. A jump PC and restore state instruction (**jpcsrs**)
causes **PSWOther** to be copied into **PSWCurrent**. After the ALU cycle of the **jpcsrs** instruction, the interrupts are enabled
and the processor returns to user state with its state restored. Appendix VI describes the trap and interrupt handling
mechanisms.

The PSW can be both read and written while in system space, but a write to the PSW while in user space has no
effect. To change the current state of the machine via the PSW, a move to special (**movtos**) instruction must be used to
write the bits in **PSWCurrent**. Before restoring the state of the machine, a move to special instruction must be used to
change the bits in **PSWOther**. All the bits are writable except the **e** bit and the E-bit shift chain.

The assignment of bits is shown in Figure 2-3. The bits corresponding to **PSWCurrent** are shown in upper case and
those in lower case correspond to the bits in **PSWOther**. The bits are:

- **I, i**
  The **I** bit should be checked by the exception handler. It is set to 0 when there is an interrupt request, otherwise it will be set to a 1. This bit never needs to be written but the value will be retained until the next interrupt or exception. The **i** bit contains the previous value of the **I** bit but in general has no meaning since only the **I** bit needs to be looked at when an exception occurs.

- **M, m**
  Interrupt mask. When set to 1, the processor will not recognize interrupts. Can only be changed by a system process, an interrupt or a trap instruction.

- **U, u**
  When set to 1, the processor is executing in user state. Can only be changed by a system process, an interrupt or a trap instruction.

- **S, s**
  Set to 1 when shifting of the PC chain is enabled.

- **e**
  Clear when doing an exception or trap return sequence. Used to determine whether state should be saved if another exception occurs during the return sequence. This bit only changes after an exception has occurred so the exception handler must be used to inspect this bit. See Appendix VI.

- **E**
  The **E** bits make up a shift chain that is used to determine whether the **e** bit needs to be cleared when an exception occurs. The **E** bits and the **e** bit are visible to the programmer but cannot be written.

Processor Status Word
V, v

The overflow mask bit. Traps on overflows are prevented when this bit is set. See Section 2.4.1.

O, o

This bit gets set or cleared on every exception. When a trap on overflow occurs, the O bit is set to 1 as seen by the exception handler. This bit never needs to be written. The o bit contains the previous value of the O bit but in general has no meaning.

```
0
```

U u | O o | , , , , , , , | E | E | E | e | v | V | m | M | i | I | s | S |

Figure 2-3: The Processor Status Word

### 2.4.1. Trap on Overflow

If the overflow mask bit in $PSW_{current}(V)$ is cleared, then the processor will trap to location 0 (the start of all exception and interrupt handling routines) when an overflow occurs during ALU or multiplication/division operations. The exception handling routine should begin the overflow trap handling routine if the overflow bit (0) is set in $PSW_{current}$.

The $V$ bit can only be changed while in system space so a system call will have to be provided for user space programs to set or clear this bit.

### 2.5. Privilege Violations

User programs cannot access system space. Any attempt to access system space will result in the address being mapped to user space. Bit 0 of the address will always be forced to 1 (a user space address) in user mode.

Attempting to write to the PSW while in user space will be the same as executing a nop instruction. The PSW is not changed and no other action is taken.

There are no illegal instructions, just strange results.
3. Instruction Timing

This chapter describes the MIPS-X instruction pipeline and the effects that pipelining has on the timing sequence for various instructions. A section is also included that describes in detail the timing of the various types of instructions.

3.1. The Instruction Pipeline

MIPS-X has a 5-stage pipeline with one instruction in each stage of the pipe once it has been filled. The clock is a two-phase clock with the phases called phase 1 ($\phi_1$) and phase 2 ($\phi_2$). The names of the pipe stages and the actions that take place in them are described in Table 3-1. The pipeline sequence is shown in Figure 3-1.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Name</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Instruction</td>
<td>Fetch the next instruction</td>
</tr>
<tr>
<td>RF</td>
<td>Register Fetch</td>
<td>The instruction is decoded. The register file is accessed during the second half of the cycle (Phase 2).</td>
</tr>
<tr>
<td>ALU</td>
<td>ALU Cycle</td>
<td>An ALU or shift operation is performed. Addresses go to memory at the end of the cycle.</td>
</tr>
<tr>
<td>MEM</td>
<td>Memory Cycle</td>
<td>Waiting for the memory (external cache) to come back on read. Data output for memory write.</td>
</tr>
<tr>
<td>WB</td>
<td>Write Back</td>
<td>The instruction result is written to the register file during the first half of the cycle (Phase 1).</td>
</tr>
</tbody>
</table>

Table 3-1: MIPS-X Pipeline Stages

---

Figure 3-1: Pipeline Sequence

Instruction Timing
3.2. Delays and Bypassing

A delay occurs because the result of a previous instruction is not available to be used by the current instruction. An example is a compute instruction that uses the result of a load instruction. If in Figure 3-1, instruction 1 is a load instruction, then the result of the load is not available to be read from the register file until the second half of WB in instruction 1. The first instruction that can access the value just loaded in the registers is instruction 4 because the registers are read on phase 2 of the cycle. This means that there is a delay of two instructions from a load instruction until the result can be used as an operand by the ALU. An instruction delay can also be called a delay slot where an instruction that does not depend on the previous instruction can be placed. This should be a nop if no useful instruction can be found. Delays between instructions can sometimes be reduced or eliminated by using bypassing.

Bypassing allows an instruction to use the result of a previous instruction before it is written back to the register file. This means that some of the delays can be reduced. Table 3-2 shows the number of delay slots that exist for various pairs of instructions in MIPS-X. The table takes into account bypassing on both the results of a compute instruction and a load instruction. For example, consider the load-address pair of instructions. This can occur if the result of the first load is used in the address calculation for the second load instruction. Without bypassing, there would be 2 delay slots. Table 3-2 shows only 1 delay slot because bypassing will take place.

The possible implementations for bypassing are bypassing only to Source 1 or to both Source 1 and Source 2. The implementation of bypassing in MIPS-X uses bypassing to both sources. Bypassing only to Source 1 means that the benefits of bypassing can only be achieved if the second instruction is accessing the value from the previous instruction via the Source I register. If the second instruction can only use the value from the previous instruction as the Source 2 register, then 2 delay slots are required. Bypassing to both Sources eliminates this asymmetry. The asymmetry is most noticeable in the number of delay slots between compute or load instructions and a following instruction that tries to store the results of the compute or load instruction. Branches are also a problem because the comparison is done with a subtraction of Source I - Source 2. Not all branch types have been implemented because it is assumed that the operands can be reversed. This means that it will not always be possible to bypass a result to a branch instruction. This asymmetry could be eliminated by taking one bit from the displacement field and using it to decide whether a subtraction or a reverse subtraction should be used. The tradeoff between the two types of bypassing is the ability to generate more efficient code in some places versus the hardware needed to implement more comparators. Table 3-2 shows the delays incurred for both implementations of bypassing. It is felt that bypassing to both Sources is preferable and the necessary hardware has been implemented.

Instructions in the slot of load instructions should not use the same register as the one that is the destination of the load instruction. Bypassing will occur and the instruction in the load slot will get the address being used for the load instead of the value from the desired register.

One other effect of bypassing should be described. Consider Figure 3-1. If instruction 1 is a load to r1 and instruction 2 is a compute instruction that puts its result also in r1, then there is an apparent conflict in instruction 3 if it wants to use r1 as its Source I register. Both the results from instructions 1 and 2 will want to bypass to instruction 3. This conflict is resolved by using the result of the second instruction. The reasoning is that this is how sequential instructions will behave. Therefore, in this example instruction 3 will use the result of the compute instruction.

Instruction Timing
3.3. Memory Instruction Interlocks

There are several instruction interlocks required because of the organization of the memory system. The external cache is a write-back cache so it requires two memory cycles to do a store operation, one to check that the location is in the cache and one to do the store. This means that a store instruction must be followed by a non-memory instruction so that there can be two memory cycles available. For example, a store followed by a compute instruction is okay because the compute instruction does not use its MEM cycle. The software should try to schedule non-memory instructions after all stores. If this is not possible, the processor will stall until the store can complete. Scheduling a *nop* instruction is not sufficient because an instruction cache miss will also generate a load cycle. This cannot be predicted so the hardware must be able to stall the processor.

There are no restrictions for instructions after a load instruction. There is a restriction that a load instruction cannot have as its destination the register being used to compute the *address* of the load. The reason is that if the load instruction misses in the external cache, it will still overwrite its destination register. This occurs because a late miss detect scheme is used in the external cache. The load instruction must be restartable.

3.4. Branch Delays

Besides the delays that can occur because one instruction must wait for the results of a previous instruction to be stored in a register or be bypassed, there are also delays because it takes time for a branch instruction to compute the destination for a taken branch. These are called *branch delays* or *branch slots*. MIPS-X has two branch slots after *every* branch instruction. Again, consider Figure 3-1. If instruction 1 is a branch instruction, then it is not until instruction 4 when the processor can decide that the branch is to be taken or not to be taken.
The branch slots can be filled with two types of instructions. They can either be ones that are always executed or ones that must be squashed if the branch does not go in the predicted direction. Squashing means that the instructions are converted into nops by preventing their write backs from occurring. This is used if the branch goes in a direction different from the one that was predicted. This mechanism is described in more detail in Section 4.3.

3.5. Jump Delays

The computation of a jump destination address means that there are two delay slots after a jump instruction before the program can begin executing at the new address. The computation uses the ALU to compute the jump address so the result is not available to the PC until the end of the ALU cycle. Unlike branches however, the instructions in the delay slots are always executed and never squashed.

3.6. Detailed Instruction Timings

This section describes the timing of the instructions as they flow through the data path. It does not describe the controls of the datapath and the timing required to set them up. These timing descriptions are intended to make more clear the programmer’s view of how each instruction is executed. The description of each instruction given in the later sections is generally insufficient when it is necessary to know the possible interactions of various instructions.

The timing for what happens during an exception is not described here. Appendix VI discusses the handling of exceptions.

The notation that will be used to describe the instruction timings will be shown first and then the execution of a normal instruction will be given. The timing for each type of instruction is then described in more detail. Finally, the timing for mstep and dstep are treated separately. These are the multiply and divide step instructions. They do not fit in with the other types of compute instructions because they use the MD register.

3.6.1. Notation

The description of each type of instruction will show what parts of the datapath are active and what they are doing for the instruction during each phase of execution. The notation that is used is:

IF,RF,ALU,MEM,WB

These are the names of the pipestages as described in Table 3- 1.

IF

This is the clock cycle before the IF cycle of the instruction being considered.

$1$ Phase 1 of the clock cycle.

$2$ Phase 2 of the clock cycle.

rSrc1,rSrc2 Register values on the Srcl and Src2 buses, corresponding to the Source 1 and Source 2 addresses specified in the instruction.

rDest Value to be written into the destination register specified by the Destination field of the instruction. The Src1 bus is used.

aluSrc1,aluSrc2 ALU latches corresponding to the values on the Srcl and Src2 buses, respectively.

IR The “instruction register.

MDRin Memory data register for values coming onto the chip.

MDRout Memory data register for values going off chip.

Instruction Timing
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rResult</td>
<td>The <em>result</em> register.</td>
</tr>
<tr>
<td>PC-source</td>
<td>The PC source to be used for this instruction. It will be one of: the displacement adder, the trap vector, the <em>incrementer</em>, the ALU or from the PC chain.</td>
</tr>
<tr>
<td>PCinc</td>
<td>The value from the PC incrementer.</td>
</tr>
<tr>
<td>PC-4</td>
<td>The last value in the PC chain.</td>
</tr>
<tr>
<td>Reg&lt;n&gt;, Reg&lt;n..m&gt;</td>
<td>Bit n or Bits n to m of register Reg.</td>
</tr>
<tr>
<td>Reg&lt;&lt; n</td>
<td>Reg is shifted left n bits.</td>
</tr>
<tr>
<td>Bypass source</td>
<td>Either <em>rResult</em> or <em>MDRin</em></td>
</tr>
<tr>
<td>Icache</td>
<td>The onchip instruction cache.</td>
</tr>
<tr>
<td>RFS</td>
<td>Reserved for Stanford.</td>
</tr>
</tbody>
</table>
### 3.6.2. A Normal Instruction

This section will show what each part of the **datapath** is doing during each phase of the execution of an instruction. The description of specific instruction types in the following sections will only describe the action of the relevant parts of the **datapath** pertaining to the instruction in question.

<table>
<thead>
<tr>
<th>Phase</th>
<th>( \phi_1 )</th>
<th>( \phi_2 )</th>
</tr>
</thead>
</table>
| IF-1  | RFS           | PC bus \( \Leftarrow \) PC,,
|       |               | Precharge tag comparators, valid bit store |
| IF    | Do tag compare
|       | Valid bit store access
|       | Icache address decoder \( \Leftarrow \) PC<26..31>
|       | Detect Icache hit
|       | Precharge Icache
|       | Do incrementer (calculate next sequential instruction address)
|       | Do Icache access
|       | IR \( \Leftarrow \) Icache |
| RF    | Do bypass comparisons
|       | \( \text{aluSrc1} \Leftarrow \text{rSrc1} \)
|       | or \( \text{aluSrc1} \Leftarrow \text{Bypass source} \)
|       | \( \text{aluSrc2} \Leftarrow \text{rSrc2} \)
|       | or \( \text{aluSrc2} \Leftarrow \text{Bypass source} \)
|       | or \( \text{aluSrc2} \Leftarrow \text{Offset value} \)
|       | Displacement adder latch \( \Leftarrow \) Displacement value
|       | MD Rout \( \Leftarrow \) r Src2
|       | or MD Rout \( \Leftarrow \) Bypass source |
| ALU   | Do ALU, do displacement adder (for branch and jump targets)
|       | Precharge Result bus
|       | Result bus \( \Leftarrow \) ALU
|       | r Result \( \Leftarrow \) Result bus
|       | Memory address pads \( \Leftarrow \) Result bus (There may be a latch here) |
| MEM   | RFS           | MDR in \( \Leftarrow \) r Result
|       |               | or MDR in \( \Leftarrow \) Memory data pads
|       |               | or Memory data pads \( \Leftarrow \) MDRout |
| WB    | r Dest \( \Leftarrow \) MDR in |
|       | RFS           |
3.6.3. Memory Instructions

These instructions do accesses to memory in the form of loads and stores. The coprocessor and floating point instructions have exactly the same timings. The only difference is that the processor may not always source an operand or use an operand during a coprocessor instruction.

The MDRout register is implemented as a series of registers to correctly time the output of data onto the memory data pads. These registers are labelled MDRout.RF\(_{\phi_2}\), MDRout.ALU\(_{\phi_1}\), MDRout.ALU\(_{\phi_2}\) and MDRout.MEM\(_{\phi_1}\).

<table>
<thead>
<tr>
<th>IF-1</th>
<th>(\phi_1)</th>
<th>RFS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\phi_2)</td>
<td>PC bus (\Leftarrow) PC, Precharge tag comparators, valid bit store</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IF</th>
<th>(\phi_1)</th>
<th>Do tag compare</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\phi_2)</td>
<td>Valid bit store access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Icache address decoder (\Leftarrow) PC&lt;26..31&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Detect Icache hit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Precharge Icache</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do incrementer (calculate next sequential instruction address)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RF</th>
<th>(\phi_1)</th>
<th>(\text{aluSrc}_1 \Leftarrow rSrc_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\phi_2)</td>
<td>or (\text{aluSrc}_1 \Leftarrow) Bypass source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{aluSrc}_2 \Leftarrow) Offset value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MDRout.RF(_{\phi_2}) (\Leftarrow) rSrc_2 (For stores)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or MDRout.RF(_{\phi_2}) (\Leftarrow) Bypass source (For stores)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALU</th>
<th>(\phi_1)</th>
<th>Do ALU(add)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\phi_2)</td>
<td>Precharge Result bus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MDRout.ALU(<em>{\phi_1}) (\Leftarrow) MDRout.RF(</em>{\phi_2}) (For stores)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result bus (\Leftarrow) ALU</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rResult (\Leftarrow) Result bus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Memory address pads (\Leftarrow) Result bus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MDRout.ALU(<em>{\phi_2}) (\Leftarrow) MDRout.ALU(</em>{\phi_1}) (For stores)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEM</th>
<th>(\phi_1)</th>
<th>MDRout.MEM(<em>{\phi_1}) (\Leftarrow) MDRout.ALU(</em>{\phi_2}) (For stores)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\phi_2)</td>
<td>MDRin (\Leftarrow) Memory data pads (For loads)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or Memory data pads (\Leftarrow) MDRout.MEM(_{\phi_1}) (For stores)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WB</th>
<th>(\phi_1)</th>
<th>rDest (\Leftarrow) MDRin (For loads)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\phi_2)</td>
<td>RFS</td>
</tr>
</tbody>
</table>

Instruction Timing
### 3.6.4. Branch Instructions

These instructions do a compare in the ALU. The PC value is taken from the displacement address when a branch is taken and from the incrementer when a branch is not taken.

<table>
<thead>
<tr>
<th>Stage</th>
<th>( \phi_1 )</th>
<th>( \phi_2 )</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IF-1</strong></td>
<td>RFS</td>
<td>PC bus ( \leftarrow ) PC,.. Precharge tag comparators, valid bit store</td>
<td></td>
</tr>
<tr>
<td><strong>IF</strong></td>
<td>Do tag compare Valid bit store access ( \text{Icache} ) address decoder ( \leftarrow \text{PC}&lt;26:31&gt; ) Detect ( \text{Icache} ) hit Precharge ( \text{Icache} ) Do incrementer (calculate next sequential instruction address)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RF</strong></td>
<td>Do bypass comparisons ( \text{aluSrc1} \leftarrow \text{rSrc1} ) or ( \text{aluSrc1} \leftarrow \text{Bypass source} ) ( \text{aluSrc2} \leftarrow \text{rSrc2} ) or ( \text{aluSrc2} \leftarrow \text{Bypass source} ) Displacement adder ( \leftarrow ) Displacement value</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ALU</strong></td>
<td>Do ( \text{ALU}(\text{Src1} - \text{Src2}) ), do displacement adder (for branch target) Precharge Result bus Evaluate condition at the end of ( \phi_1 ) before the rising edge of ( \phi_2 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td>( \text{MDRin} \leftarrow \text{rResult} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td>RFS</td>
<td>RFS</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Timing**
3.65 Compute Instructions

These instructions are mostly 3-operand instructions that use the ALU to do an operation. Some of them do traps or jumps. These are treated separately in Section 3.6.6. The timing for instructions that access the special registers is described in Section 3.6.5.1.

<table>
<thead>
<tr>
<th>Phase</th>
<th>φ₁</th>
<th>φ₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF₁</td>
<td>RFS</td>
<td>PC bus ⇐ PC,,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Precharge tag comparators, valid bit store</td>
</tr>
<tr>
<td>IF</td>
<td>Do tag compare</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Valid bit store access</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Icache address decoder ⇐ PC&lt;26..31&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Detect Icache hit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Precharge Icache</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Do incrementer (calculate next sequential instruction address)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Do Icache access</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IR ⇐ Icache</td>
<td></td>
</tr>
<tr>
<td>RF</td>
<td>Do bypass comparisons</td>
<td></td>
</tr>
<tr>
<td></td>
<td>aluSrc₁ ⇐ rSrc₁</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or aluSrc₁ ⇐ Bypass source</td>
<td></td>
</tr>
<tr>
<td></td>
<td>aluSrc₂ ⇐ rSrc₂</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or aluSrc₂ ⇐ Bypass source</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or aluSrc₂ ⇐ Immediate value (for Compute Immediate Instructions)</td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>Do ALU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Precharge Result bus</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Result bus ⇐ ALU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rResult ⇐ Result bus</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>RFS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MDRin ⇐ rResult</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>rDest ⇐ MDRin</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RFS</td>
<td></td>
</tr>
</tbody>
</table>
3.651. Special Instructions

These instructions (*movtos* and *movfrs*) access the *special registers* described in Section 2.3.

<table>
<thead>
<tr>
<th>Pipeline Stage</th>
<th>φ₁</th>
<th>φ₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF₁</td>
<td>RFS</td>
<td>PC bus ← PC&lt;&gt;, Precharge tag comparators, valid bit store</td>
</tr>
<tr>
<td>IF</td>
<td>φ₁</td>
<td>Do tag compare, Valid bit store access, Icache address decoder ← PC&lt;26..31&gt;, Detect Icache hit, Precharge Icache, Do incrementer (calculate next sequential instruction address)</td>
</tr>
<tr>
<td></td>
<td>φ₂</td>
<td>Do Icache access, IR ← Icache</td>
</tr>
<tr>
<td>RF</td>
<td>φ₁</td>
<td>Do bypass comparisons</td>
</tr>
<tr>
<td></td>
<td>φ₂</td>
<td>aluSrc₁ ← rSrc₁ (For <em>movtos</em>) or aluSrc₁ ← Bypass source (For <em>movtos</em>)</td>
</tr>
<tr>
<td>ALU</td>
<td>φ₁</td>
<td>Do ALU(pass Srcl), Precharge Result bus</td>
</tr>
<tr>
<td></td>
<td>φ₂</td>
<td>Result bus ← alu Srcl (For <em>movtos</em>) or Result bus ← Special Register (For <em>movfrs</em>) Special Register ← Result bus (For <em>movfrs</em>) rResult ← Result bus</td>
</tr>
<tr>
<td>MEM</td>
<td>φ₁</td>
<td>RFS</td>
</tr>
<tr>
<td></td>
<td>φ₂</td>
<td>MDRin ← rResult</td>
</tr>
<tr>
<td>WB</td>
<td>φ₁</td>
<td>rDest ← MDRin (For <em>movfrs</em>)</td>
</tr>
<tr>
<td></td>
<td>φ₂</td>
<td>RFS</td>
</tr>
</tbody>
</table>

Instruction Timing
### 3.6.6. Jump Instructions

<table>
<thead>
<tr>
<th></th>
<th>(\phi_1)</th>
<th>(\phi_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IF</strong></td>
<td>RFS</td>
<td>PC bus (\Leftrightarrow) PC,,,-&lt;br&gt;Precharge tag comparators, valid bit store</td>
</tr>
<tr>
<td><strong>IF</strong></td>
<td>Do tag compare</td>
<td>Valid bit store access&lt;br&gt;<strong>Icache</strong> address decoder (\Leftrightarrow) PC&lt;26..31&gt;&lt;br&gt;Detected <strong>Icache hit</strong>&lt;br&gt;Precharge <strong>Icache</strong>&lt;br&gt;Do <strong>incrementer</strong> (calculate next sequential instruction address)</td>
</tr>
<tr>
<td><strong>RF</strong></td>
<td>Do bypass comparisons</td>
<td><strong>aluSrc1</strong> (\Leftrightarrow) <strong>rSrc1</strong>&lt;br&gt;or <strong>aluSrc1</strong> (\Leftrightarrow) Bypass source&lt;br&gt;<strong>aluSrc2</strong> (\Leftrightarrow) Immediate value (For jspci)</td>
</tr>
<tr>
<td><strong>ALU</strong></td>
<td>Do <strong>ALU</strong> (add)</td>
<td>Precharge Result bus&lt;br&gt;Result bus (\Leftrightarrow) <strong>PCinc</strong> (For jspci)&lt;br&gt;PC bus (\Leftrightarrow) <strong>ALU</strong> (For jspci)&lt;br&gt;or PC bus (\Leftrightarrow) PC-4, shift PC chain (For jpc and jpcrs)&lt;br&gt;or PC bus (\Leftrightarrow) Trap vector (For trap)&lt;br&gt;<strong>PSWcurrent</strong> (\Leftrightarrow) <strong>PSWother</strong> (For jpcrs)&lt;br&gt;r<strong>Result</strong> (\Leftrightarrow) Result bus</td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td>RFS</td>
<td>M<strong>DRin</strong> (\Leftrightarrow) r<strong>Result</strong></td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td>r<strong>Dest</strong> (\Leftrightarrow) M<strong>DRin</strong> (For jspci)</td>
<td>RFS</td>
</tr>
</tbody>
</table>
3.6.7. Multiply Step - mstep

The MD register is implemented as a series of \( \phi_2-\phi_1 \) registers. They are called MDresult\( \phi_2 \), MDresult\( \phi_1 \), MDmdrin\( \phi_2 \), and MDwb\( \phi_1 \). The names reflect the names of the bypass registers used when bypassing to the register file. The special register that is visible for reading and writing is MDresult\( \phi_2 \). This chain of registers is necessary for restarting the sequence after an exception. MDwb\( \phi_1 \) contains the true value of MD. When an interrupt occurs, the write-back into this register is stopped just like write-backs to a register in the register file. The value in this register is needed to restart the sequence. One cycle after an interrupt is taken, the contents of MDwb\( \phi_1 \) are available in MDresult\( \phi_2 \). This value has to be saved if the interrupt routine does any multiplication or division.

The mstart instruction has similar timing with a different ALU operation.

There must be one instruction between the instruction that loads the MD register and the first instruction that uses the MD register. This occurs when starting a multiplication or division routine and when restarting after an interrupt.

<table>
<thead>
<tr>
<th>IF,1</th>
<th>( \phi_1 )</th>
<th>RFS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \phi_2 )</td>
<td>PC bus ( \Leftarrow ) PC_source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Precharge tag comparators, valid bit store</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IF</th>
<th>( \phi_1 )</th>
<th>Do tag compare</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \phi_2 )</td>
<td>Valid bit store access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Icache address decoder ( \Leftarrow ) PC_&lt;26..31&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Detect Icache hit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Precharge Icache</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do incrementer (calculate next sequential instruction address)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Do Icache access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IR ( \Leftarrow ) Icache</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RF</th>
<th>( \phi_1 )</th>
<th>Do bypass comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \phi_2 )</td>
<td>aluSrc1 ( \Leftarrow ) rSrc1&lt;&lt;1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or aluSrc1 ( \Leftarrow ) Bypass source&lt;&lt;1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>aluSrc2 ( \Leftarrow ) rSrc2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALU</th>
<th>( \phi_1 )</th>
<th>Do ALU(add)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \phi_2 )</td>
<td>Latch aluSrc1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Precharge Result bus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result bus ( \Leftarrow ) ALU (MSB (MDresult( \phi_1 )) is 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or Result bus ( \Leftarrow ) aluSrc1 (MSB (MDresult( \phi_1 )) is 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rResult ( \Leftarrow ) Result bus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MDresult( \phi_2 ) ( \Leftarrow ) = MDresult( \phi_1 )&lt;&lt;1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEM</th>
<th>( \phi_1 )</th>
<th>MDresult( \phi_1 ) ( \Leftarrow ) MDresult( \phi_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \phi_2 )</td>
<td>MDRin ( \Leftarrow ) rResult</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MDmdrin( \phi_2 ) ( \Leftarrow ) = MDresult( \phi_1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WB</th>
<th>( \phi_1 )</th>
<th>rDest ( \Leftarrow ) MDRin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \phi_2 )</td>
<td>MDwb( \phi_1 ) ( \Leftarrow ) MDmdrin( \phi_2 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RFS</td>
</tr>
</tbody>
</table>

Instruction Timing
### 3.6.8. Divide Step - *dstep*

The *MD* register is also used for this instruction. See Section 3.6.7 for a description of its implementation and the notation used.

<table>
<thead>
<tr>
<th>IF</th>
<th>1</th>
<th>( \phi_1 )</th>
<th>RFS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>PC bus ( \leftarrow \text{PC}_{\text{source}} )</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>Precharge tag comparators, valid bit store</td>
</tr>
</tbody>
</table>

| IF | \( \phi_1 \) | Do tag compare |
|    |          | Valid bit store access |
|    |          | \text{Icache} address decoder \( \leftarrow \text{PC}<26..31> \) |
|    |          | Detect \text{Icache} hit |
|    | \( \phi_2 \) | Precharge \text{Icache} |
|    |          | Do \text{Icache} access |
|    |          | Do \text{incrementer} (calculate next sequential instruction address) |
|    |          | IR \( \leftarrow \text{Icache} \) |

| RF | \( \phi_1 \) | Do bypass comparisons |
|    |          | \text{aluSrc1} \( \leftarrow \text{rSrc1} \ll 1 + \text{MSB(MDresult.}_\phi_1 \) |
|    |          | or \text{aluSrc1} \( \leftarrow \text{Bypass source} \ll 1 + \text{MSB(MDresult.}_\phi_1 \) |
|    |          | \text{aluSrc2} \( \leftarrow \text{rSrc2} \) |

| ALU | \( \phi_1 \) | Do \text{ALU(sub)} |
|     |          | Precharge Result bus |
|     | \( \phi_2 \) | Result bus \( \leftarrow \text{ALU (MSB (ALU result) is 0)} \) |
|     |          | or Result bus \( \leftarrow \text{aluSrc1 (MSB (ALU result) is 1)} \) |
|     |          | rResult \( \leftarrow \text{Result bus} \) |
|     |          | MDresult.\( \_\phi_2 \leftarrow \text{MDresult.}_\phi_1 \ll 1 + \text{Complement of MSB(AlU result)} \) |

| MEM | \( \phi_1 \) | MDresult.\( \_\phi_1 \leftarrow \text{MDresult.}_\phi_2 \) |
|     | \( \phi_2 \) | MDRin \( \leftarrow \text{rResult} \) |
|     |          | MDmdrin.\( \_\phi_2 \leftarrow \text{MDresult.}_\phi_1 \) |

| WB | \( \phi_1 \) | rDest \( \leftarrow \text{MDRin} \) |
|    |          | MDwb.\( \_\phi_1 \leftarrow \text{MDmdrin.}_\phi_2 \) |
|    | \( \phi_2 \) | RFS |
Instruction Timing
4. Instruction Set

There are four different types of instructions. They are memory instructions, branch instructions, compute instructions, and compute immediate instructions. Coprocessor instructions are part of the memory instructions.

4.1. Notation

This section explains the notation used in the descriptions of the instructions.

- **MSB(x)**: The most significant bit of x.
- **x<<y**: x is shifted left by y bits.
- **x>>y**: x is shifted right by y bits.
- **x#y**: x is a number represented in base y
- **x || Y**: x is concatenated with y.
- **PCcurrent**: Address of the instruction being fetched during the ALU cycle of an instruction
- **PCnext**: Address of the next instruction to be fetched.
- **Reg(n)**: The contents of CPU register n.
- **FReg(n)**: The contents of register n in the floating point unit (FPU).
- **Reg<n>, Reg<n..m>**: Bit n or Bits n to m of register Reg.
- **Memory[addr]**: The contents of memory at the location addr. The value accessed is always a word of 32 bits.
- **SignExtend**: The value of n sign extended to 32 bits. The size of n is specified by the field being sign extended.
- **rSrc1**: The register number used as the Source 1 operand
- **rSrc2**: The register number used as the Source 2 operand
- **rDest**: The register number used as the Destination location.
- **fSrc1**: The register number used as the Source 1 floating point operand.
- **fSrc2**: The register number used as the Source 2 floating point operand.
- **fDest**: The register number used as the Destination floating point register.
- **cop1**: Coprocessor instruction.
- **MAR**: The memory address register. The contents of this register are placed on the address pins of the processor.
- **MDR**: The memory data register. The address pads of the processor always reflect the contents of this register.

4.2. Memory Instructions

The memory instructions are the ones that do an external memory cycle. The most commonly used memory instructions are load and store. The other instructions that are part of the memory instructions are the coprocessor instructions. They do not always generate a memory cycle that is recognized by memory. Instead the coprocessor uses the cycle. This is explained in more detail in the individual instruction descriptions.
4.2.1. Id • Load

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Srcl</th>
<th>Dest</th>
<th>Offset(17)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Assembler**

`Id Offset[rSrl],rDest`

**Operation**

`Reg(Dest) ← Memory[SignExtend(Offset) + Reg(Srcl)]`

**Description**

The offset field is sign extended and added to the contents of the register specified by the Srcl field to compute a memory address. The contents of that memory location is put into `Reg(Dest)`.

Note: An instruction in the slot of a loud instruction that uses the same register as the load instruction is loading is not guaranteed to get the correct result. Do not try to use the `load` slots in this manner.
### 4.2.2. `st` - Store

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Src2</th>
<th>Offset(17)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>0 1</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

**Assembler**

`st Offset[rSrc1],rSrc2`

**Operation**

`Memory[SignExtend(Offset) + Reg(Src1)] ← Reg(Src2)`

**Description**

The offset field is sign extended and added to the contents of the register specified by the Src1 field to compute a memory address. The contents of Reg(Src2) are stored at that memory location.

This instruction requires 2 memory cycles, one to read the cache and then one to do the store. To obtain maximum performance, instructions that do not require a memory cycle should be scheduled after a store instruction if possible. Otherwise, the processor may stall for one cycle.
4.2.3. Idf - Load Floating Point

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Dest</th>
<th>Offset(17)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1 0</td>
<td>0 0</td>
<td></td>
</tr>
</tbody>
</table>

Assembler

Idf Offset[rSrc1],fDest

Operation

FReg(Dest) ← Memory[SignExtend(Offset) + Reg(Srcl)]

Description

The offset field is sign extended and added to the contents of the register specified by the Src1 field to compute a memory address. The contents of that memory location is put into the register specified by Dest in the floating point unit (FReg(Dest)). The CPU ignores the data returned in the memory cycle.

Note: An instruction in the slot of a load instruction that uses the same register as the load instruction is loading is not guaranteed to get the correct result. Do not try to use the load slots in this manner.

Note: If a processor configuration does not have an FPU then different code must be generated to emulate the floating point instructions. Any code that tries to use FPU instructions when there is no FPU will not execute correctly.
4.2.4. stf - Store Floating Point

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Srcl</th>
<th>Src2</th>
<th>Offset(17)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Assembler

`stf Offset[rSrcl],fSrc2`

Operation

Memory[SignExtend(Offset) + Reg(Srcl)] ⇐ FReg(Src2)

Description

The offset field is sign extended and added to the contents of the register specified by the Srcl field to compute a memory address. The contents of the floating point register specified by `Src2` are stored at that memory location. The CPU does not put out any data during this write memory cycle.

Note: If a processor configuration does not have an FPU then different code must be generated to emulate the floating point instructions. Any code that tries to use FPU instructions when there is no FPU will not execute correctly.
4.25. ld - Load Through

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Dest</th>
<th>Offset(17)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Assembler

ldt Offset[rSrc1],rDest

Operation

Reg(Dest) ← Memory[SignExtend(Offset) + Reg(Src1)]

Description

This instruction is the same as ld except that it is guaranteed to bypass the cache. There is no check to see whether the location being accessed currently exists in the cache.

The offset field is sign extended and added to the contents of the register specified by the Src1 field to compute a memory address. The contents of that memory location is put into Reg(Dest).

Note: An instruction in the slot of a load instruction that uses the same register as the load instruction is loading is not guaranteed to get the correct result. Do not try to use the load slots in this manner.
4.2.6. stt - Store Through

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Srl</th>
<th>Src2</th>
<th>Offset(17)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>010</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Assembler**

\texttt{stt \textit{Offset}[\textit{Src1}],rSrc2}

**Operation**

\texttt{Memory[SignExtend(Offset) + Reg(Src1)]\leftarrow Reg(Src2)}

**Description**

This instruction is the same as \texttt{st} except that it is guaranteed to bypass the cache. There is no check to see whether the location being accessed currently exists in the cache.

The offset field is sign extended and added to the contents of the register specified by the Srl field to compute a memory address. The contents of Reg(Src2) are stored at that memory location.
4.2.7. movfrc - Move From Coprocessor

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1(r0)</th>
<th>Dest</th>
<th>COP#</th>
<th>Func</th>
<th>CS1</th>
<th>CS2/CD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Assembler

movfrc Cop1,rDest

Operation

\[ \text{MAR} \leftarrow \text{SignExtend(Cop1)} + \text{Reg(Src 1)} \]
\[ \text{Reg(Dest)} \leftarrow \text{MDR} \]

Description

This instruction is used to do a Coprocessor register to CPU register move.

The Cop1 field is sign extended and added to the contents of the register specified by the Src1 field. The Srcl field should be Register 0 if the Cop1 field is to be unmodified (hackers take note). The Cop1 field will appear on the address lines of the processor where it can be read by the coprocessor. The coprocessor will place a value on the data bus that will be stored in Reg(Dest) of the CPU. The memory system will ignore this memory cycle.

The Cop1 field is decoded by the coprocessor to find the coprocessor being addressed (COP#) and the function to be performed. A possible format is shown above. The fields CS1 and CS2/CD show possible coprocessor register fields. The format is flexible except that all coprocessors should find the COP# in the same place.

Note: An instruction in the slot of a movfrc instruction that uses the same register that the movfrc instruction is loading is not guaranteed to get the correct result. Do not try to use the slots in this manner.
4.2.8. movtoc - Move To Coprocessor

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1(r0)</th>
<th>Src2</th>
<th>COP#</th>
<th>Func</th>
<th>CS1</th>
<th>CS2/CD</th>
</tr>
</thead>
<tbody>
<tr>
<td>J1</td>
<td>011</td>
<td>1 0 0 0 0</td>
<td>' ' '</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

Assembler

movtoc Cop1,rSrc2

Operation

MAR $\leftarrow$ SignExtend(Cop1) + Reg(Src1)
MDR $\leftarrow$ Reg(Src2)

Description

This instruction is used to do a CPU register to Coprocessor register move.

The Cop1 field is sign extended and added to the contents of the register specified by the Src1 field. The Src1 field should be Register 0 if the Cop1 field is to be unmodified (hackers take note). The Cop1 field will appear on the address lines of the processor where it can be read by the coprocessor. The contents of register Src2 are placed on the data lines so that the coprocessor can access the value. The memory system will ignore this memory cycle.

The Cop1 field is decoded by the coprocessors to find the coprocessor being addressed (COP#) and the function to be performed. A possible format is shown above. The fields CS1 and CS2/CD show possible coprocessor register fields. The format is flexible except that all coprocessors should find the COP# in the same place.
4.2.9. **aluc** - Coprocessor ALU

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Srl(r0)</th>
<th>COP#</th>
<th>Func</th>
<th>CS1</th>
<th>CS2/CD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Assembler**

`aluc Cop1`

**Operation**

\[
\text{MAR} \leftarrow \text{SignExtend(Cop1)} + \text{Reg(Src1)}
\]

**Description**

This instruction is used to execute a coprocessor instruction that does not require the transfer of data to or from the CPU.

This instruction is actually implemented as:

```
movfr Cop1, r0.
```

The Cop1 field is sign extended and added to the contents of the register specified by the Srl field. The Srl field should be Register 0 if the Cop1 field is to be unmodified (hackers take note). The Cop1 field will appear on the address lines of the processor where it can be read by the coprocessor. The memory system will ignore this memory cycle.

The Cop1 field is decoded by the coprocessor’s to find the coprocessor being addressed (COP#) and the function to be performed. A possible format is shown above. The fields CS1 and CS2/CD show possible coprocessor register fields. The format is flexible except that all coprocessor-s should find the COP# in the same place.

Note that this instruction is needed to perform floating point ALU operations. Only floating point loads and stores have special FPU instructions.
4.3. Branch Instructions

As described previously in Section 3.4, all branch instructions have two delay slots. The instructions placed in the slots can be either ones that must always execute or ones that should be executed if the branch is taken. There are two flavours of branch instructions that must be used depending on the type of instructions placed in the slots. They are:

No squash: The instructions in the slots are always executed. They are never squashed (turned into nops).

Squash if don’t go: All branches are statically predicted to go (be taken). This means that the instructions in the branch slots should be instructions from the target instruction stream. If the branch is not taken, then the instructions in the slots are squashed.

The instructions in the slots must be both of the same type. That is, they should both always execute or both be from the target instruction stream. If squashing takes place, both instructions in the slots are treated equally.

Note that for best performance, it is best to try to find instructions that can always execute and use the no squash branch types.

Branch instructions can be put in the slot of branches that can be squashed.

The branch conditions are established by testing the result of

\[
\text{Reg(Src 1) - Reg(Src2)}
\]

where Src1 and Src2 are specified in the branch instruction. The condition to be tested is specified in the COND field of the branch instruction. The expressions used to derive the conditions use the following notation:

- \( N \) Bit 0 of the result is a 1. The result is negative.
- \( Z \) The result is 0.
- \( V \) 32-bit two's-complement overflow has occurred in the result.
- \( C \) A carry bit was generated from bit 0 of the result in the ALU.
- \( \oplus \) Exclusive-Or

Some branch conditions that are usually found on other machines do not exist on MIPS-X. They can be synthesized by reversing the order of the operands or comparing with Reg(0) in Source 2 (Src2=0). These branches are shown in Table 4-1 along with the existing branches.
<table>
<thead>
<tr>
<th>Branch</th>
<th>Description</th>
<th>Expression</th>
<th>Branch To Use</th>
<th>IfSynthesized</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>Branch if equal</td>
<td>z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bge</td>
<td>Branch if greater than or equal</td>
<td>N ⊕ V</td>
<td>blt (rev ops)</td>
<td></td>
</tr>
<tr>
<td>bgt</td>
<td>Branch if greater than</td>
<td>(N ⊕ V) + Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bhi</td>
<td>Branch if higher</td>
<td>C + Z</td>
<td>blos (rev ops)</td>
<td></td>
</tr>
<tr>
<td>bhs</td>
<td>Branch if higher or same</td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ble</td>
<td>Branch if less than or equal</td>
<td>(N ⊕ V) + Z</td>
<td>bge (rev ops)</td>
<td></td>
</tr>
<tr>
<td>blo</td>
<td>Branch if lower than</td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>blos</td>
<td>Branch if lower or same</td>
<td>C + Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>blt</td>
<td>Branch if less than</td>
<td>N ⊕ V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>Branch if not equal</td>
<td>Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bpl</td>
<td>Branch if plus</td>
<td>N</td>
<td></td>
<td>bge (cmp to Src2=0)</td>
</tr>
<tr>
<td>bmi</td>
<td>Branch if minus</td>
<td>N</td>
<td></td>
<td>blt (cmp to Src2=0)</td>
</tr>
<tr>
<td>bra</td>
<td>Branch always</td>
<td></td>
<td></td>
<td>beq r0,r0</td>
</tr>
</tbody>
</table>

Table 4-1: Branch Instructions
4.3.1. beq - Branch If Equal

<table>
<thead>
<tr>
<th>TY</th>
<th>Cond</th>
<th>Src1</th>
<th>Src2</th>
<th>SQ</th>
<th>Disp(16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

s = 1 ⇒ Squash if don’t go
s = 0 ⇒ No squashing

**Assembler**

beq  rSrc1,rSrc2,Label    ; No squashing
beqsq rSrc1,rSrc2,Label   ; Squash if don’t go

**Operation**

If \([\text{Reg(Src1)} - \text{Reg(Src2)}] = Z\)
then

\[
\text{PCnext} \leftarrow \text{PCcurrent} + \text{SignExtend}@isp)
\]

**Description**

If \(\text{Reg(Src1)}\) equals \(\text{Reg(Src2)}\) then execution continues at \(\text{Label}\) and the two delay slot instructions are executed. The value of \(\text{Label}\) is computed by adding \(\text{PCcurrent} + \text{the signed displacement}\).

If \(\text{Reg(Src1)}\) does not equal \(\text{Reg(Src2)}\), then the delay slot instructions are executed for \text{beq} and squashed for \text{beqsq}. 

beq  Branch If Equal  beq
4.3.2. bge - Branch If Greater than or Equal

<table>
<thead>
<tr>
<th>TY</th>
<th>Cond</th>
<th>Src1</th>
<th>Src2</th>
<th>SQ</th>
<th>Disp(16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>s</td>
</tr>
</tbody>
</table>

- $s = 1 \Rightarrow$ Squash if don’t go
- $s = 0 \Rightarrow$ No squashing

**Assembler**

- bge rSrc1,rSrc2,Label ; No squashing
- bgesq rSrc1,rSrc2,Label ; Squash if don’t go

**Operation**

If $[\text{Reg(Srcl)} - \text{Reg(Src2)}] \Rightarrow \overline{N \oplus V}$
then

$\text{PCnext} \leftarrow \text{PCcurrent} + \text{SignExtend(Disp)}$

**Description**

This is a signed compare.

If Reg(Srcl) is greater than or equal to Reg(Src2) then execution continues at Label and the two delay slot instructions are executed. The value of Label is computed by adding PCcurrent + the signed displacement.

If Reg(Srcl) is less than Reg(Src2), then the delay slot instructions are executed for bge and squashed for bgesq.
4.3.3. bhs - Branch If Higher Or Same

<table>
<thead>
<tr>
<th>TY</th>
<th>Cond</th>
<th>Src1</th>
<th>Src2</th>
<th>SQ</th>
<th>Disp(16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>_1</td>
<td></td>
</tr>
</tbody>
</table>

\[ s = 1 \Rightarrow \text{Squash if don't go} \]

\[ s = 0 \Rightarrow \text{No squashing} \]

**Assembler**

bhs rSrc1,rSrc2,Label ; No squashing
bhssq rSrc1,rSrc2,Label ; Squash if don't go

**Operation**

If \( \text{Reg(Src1)} - \text{Reg(Src2)} \) \( \geq C \)
then
\[ \text{PCnext} \leftarrow \text{PCcurrent} + \text{SignExtend}(\text{isp}) \]

**Description**

This is an unsigned compare.

If Reg(Src1) is higher than or equal to Reg(Src2) then execution continues at Label and the two delay slot instructions are executed. The value of Label is computed by adding PCcurrent + the signed displacement.

If Reg(Src 1) is lower than Reg(Src2), then the delay slot instructions are executed for bhs and squashed for bhssq.
4.3.4. blo - Branch If Lower Than

<table>
<thead>
<tr>
<th>TY</th>
<th>Cond</th>
<th>Src1</th>
<th>Src2</th>
<th>SQ</th>
<th>Disp(16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- $s = 1 \Rightarrow$ Squash if don't go
- $s = 0 \Rightarrow$ No squashing

Assembler

- `blo rSrcl,rSrc2,Label` ; No squashing
- `blosq rSrcl,rSrc2,Label` ; Squash if don't go

Operation

If $[\text{Reg}(\text{Src1}) - \text{Reg}(\text{Src2})] \geq 0$
then
PCnext $\leftarrow PC_{\text{current}} + \text{SignExtend}(\text{isp})$

Description

This is an unsigned compare.

If \text{Reg}(\text{Src1}) is lower than \text{Reg}(\text{Src2}) then execution continues at Label and the two delay slot instructions are executed. The value of Label is computed by adding PC_{\text{current}} + the signed displacement.

If \text{Reg}(\text{Src1}) is higher than or equal to \text{Reg}(\text{Src2}) or if there was a carry generated, then the delay slot instructions are executed for blo and squashed for blosq.
4.3.5. \texttt{blt} - Branch If Less Than

<table>
<thead>
<tr>
<th>TY</th>
<th>Cond</th>
<th>Src1</th>
<th>Src2</th>
<th>SQ</th>
<th>Disp(16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.0</td>
<td>0_1</td>
<td>1</td>
<td></td>
<td>s</td>
<td></td>
</tr>
</tbody>
</table>

$s = 1 \Rightarrow$ Squash if don’t go
$s = 0 \Rightarrow$ No squashing

\textbf{Assembler}

\begin{align*}
\texttt{blt} & \quad r\text{Src}_1, r\text{Src}_2, \text{Label} \quad \text{; No squashing} \\
\texttt{bltsq} & \quad r\text{Src}_1, r\text{Src}_2, \text{Label} \quad \text{; Squash if don’t go}
\end{align*}

\textbf{Operation}

\begin{align*}
\text{If } \text{Reg(Src}_1) \text{ - Reg(Src}_2) & \Rightarrow \text{N} \oplus \text{V} \\
\text{then} & \\
\text{PC}_{\text{next}} & \Leftarrow \text{PC}_{\text{current}} + \text{SignExtend}(\text{isp})
\end{align*}

\textbf{Description}

This is a signed compare.

If $\text{Reg(Src}_1)$ is less than $\text{Reg(Src}_2)$ then execution continues at \textit{Label} and the two delay slot instructions are executed. The value of \textit{Label} is computed by adding $\text{PC}_{\text{current}} + \text{the signed displacement}$.

If $\text{Reg(Src}_1)$ is greater than or equal to $\text{Reg(Src}_2)$, then the delay slot instructions are executed for \texttt{blt} and squashed for \texttt{bltsq}.
### 4.3.6. bne - Branch If Not Equal

<table>
<thead>
<tr>
<th>TY</th>
<th>Cond</th>
<th>Src1</th>
<th>Src2</th>
<th>SQ</th>
<th>Disp(16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>s = 1 ⇒ Squash if don’t go</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s = 0 ⇒ No squashing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Assembler**

- `bne rSrc1,rSrc2,Label` ; No squashing
- `bnesq rSrc1,rSrc2,Label` ; Squash if don’t go

**Operation**

\[
\text{If } [\text{Reg(Src1)} - \text{Reg(Src2)}] \Rightarrow \overline{Z} \\
\text{then} \\
\text{PCnext} \Leftarrow \text{PCcurrent} + \text{SignExtend(Disp)}
\]

**Description**

If Reg(Src1) does not equal Reg(Src2) then execution continues at `Label` and the two delay slot instructions are executed. The value of `Label` is computed by adding `PCcurrent` + the signed displacement.

If Reg(Src1) equals Reg(Src2), then the delay slot instructions are executed for `bne` and squashed for `bnesq`. 
4.4. Compute Instructions

Most of the compute instructions are 3-operand instructions that use the ALU or the shifter to perform an operation on the contents of 2 registers and store the result in a third register.
4.4.1. add = Add

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Src2</th>
<th>Dest</th>
<th>Comp Func(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 0 0 0 0 0 1 1 0 0 1</td>
</tr>
</tbody>
</table>

**Assembler**

add rSrc1, rSrc2, rDest

**Operation**

Reg(Dest) ← Reg(Src1) + Reg(Src2)

**Description**

The sum of the contents of the two source registers is stored in the destination register.
4.4.2, dstep - Divide Step

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Src2</th>
<th>Dest</th>
<th>CompFunc(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1 0 0 1 1 0 0 1 1 0</td>
</tr>
</tbody>
</table>

**Assembler**

dstep rSrc1, rSrc2, rDest

**Operation**

Src1 should be the same as Dest.

\[
\begin{align*}
&\text{ALUsrc1} &:= &\text{Reg(Src1) }\ll 1 + \text{MSB(Reg(MD))} \\
&\text{ALUsrc2} &:= &\text{Reg(Src2)} \\
&\text{ALUoutput} &:= &\text{ALUsrc1} - \text{ALUsrc2}
\end{align*}
\]

If MSB(ALUoutput) is 1

then

\[
\begin{align*}
&\text{Reg(Dest)} &:= &\text{ALUsrc1} \\
&\text{Reg(MD)} &:= &\text{Reg(MD)}\ll 1
\end{align*}
\]

else

\[
\begin{align*}
&\text{Reg(Dest)} &:= &\text{ALUoutput} \\
&\text{Reg(MD)} &:= &\text{Reg(MD)}\ll 1 + 1
\end{align*}
\]

**Description**

This is one step of a 1-bit restoring division algorithm. The division scheme is described in Appendix IV.
4.4.3. mstart - Multiply Startup

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Src2</th>
<th>Dest</th>
<th>Comp</th>
<th>Func(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Assembler:
```
ms tart rSrc2,rDest
```

Operation:
- If MSB(Multiplier loaded in Reg(MD)) is 1
  - Reg(Dest) ← 0 - Reg(Src2)
  - Reg(MD) ← Reg(MD) << 1
- else
  - Reg(Dest) ← 0
  - Reg(MD) ← Reg(MD) << 1

Description:
This is the first step of a 1-bit shift and add multiplication algorithm used when doing signed multiplication. If the most significant bit of the multiplier is 1, then the multiplicand is subtracted from 0 and the result is stored in Reg(Dest). The multiplication scheme is described in Appendix IV.
### 4.4.4. mstep - Multiply Step

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Src2</th>
<th>Dest</th>
<th>Comp Func(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000010011001~</td>
</tr>
</tbody>
</table>

**Assembler**

```assembly
mstep rSrc1,rSrc2,rDest
```

**Operation**

Srcl should be the same as Dest.

If MSB(Reg(MD)) is 1

then

- `Reg(Dest) ← Reg(Src1)<< 1 + Reg(Src2)`
- `Reg(MD) ← Reg(MD)<< 1`

else

- `Reg(Dest) ← Reg(Src1)<< 1`
- `Reg(MD) ← Reg(MD)<< 1`

**Description**

This is one step of a 1-bit shift and add multiplication algorithm. The multiplication scheme is described in Appendix IV.
4.4.5. sub - Subtract

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Src2</th>
<th>Dest</th>
<th>Comp Func(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0 0 0 1 1 0 0 1 1 0</td>
</tr>
</tbody>
</table>

**Assembler**

```
sub rSrc1, rSrc2, rDest
```

**Operation**

```
Reg(Dest) ← Reg(Src1) - Reg(Src2)
```

**Description**

The Source 2 register is subtracted from the Source 1 register and the difference is stored in the Destination register.
4.4.6. subnc - Subtract with No Carry In

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Src2</th>
<th>Dest</th>
<th>CompFunc(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 0 0 0 0 1 0 1 1 0</td>
</tr>
</tbody>
</table>

Assembler
subnc rSrc1, rSrc2, rDest

Operation
Reg(Dest) ← Reg(Src1) + Reg(Src2)

Description
The 1’s complement of the Source 2 register is added to the Source 1 register and the result is stored in the Destination register. This instruction is used when doing multiprecision subtraction.

The following is an example of double precision subtraction. The operation required is \( C = A - B \), where \( A, B \) and \( C \) are double word values.

```
subnc rAhi, rBhi, rChi
bhssq rAlo, rBlo, ll
addi rChi, #1, rChi

11: sub rAlo, rBlo, Clo
```

subnc Subtract with No Carry In
subnc
4.4.7. and - Logical And

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Src2</th>
<th>Dest</th>
<th>Comp Func(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>' '</td>
<td>' '</td>
</tr>
</tbody>
</table>

Assembler

```
and rSrc1, rSrc2, rDest
```

Operation

```
Reg(Dest) <=> Reg(Src1) bitwise and Reg(Src2)
```

Description

This is a bitwise logical and of the bits in Source 1 and Source 2. The result is placed in Destination.
4.4.8. bic - Bit Clear

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>src2</th>
<th>Dest</th>
<th>Comp Func(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0 0 0 0 0 0 0 0 1 0 1 1</td>
</tr>
</tbody>
</table>

Assembler

bic rSrc1,rSrc2,rDest

Operation

Reg(Dest) ⇐ Reg(Src1) bitwise and Reg(Src2)

Description

Each bit that is set in Source 1 is cleared in Source 2. The result is placed in Destination.
4.4.9. not - Ones Complement

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Dest</th>
<th>Comp Func(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Assembler
not rSrc1,rDest

Operation
Reg(Dest) ← Reg(Src1)

Description
The ones complement of Source 1 is placed in Destination.
4.4.10. or - Logical Or

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Src2</th>
<th>Dest</th>
<th>Comp Func(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>1 0 0</td>
<td>' ' '</td>
<td>' ' '</td>
<td>' ' '</td>
<td>0 0 0 0 1 1 1 0 1 1</td>
</tr>
</tbody>
</table>

Assembler
or rSrc1, rSrc2, rDest

Operation
Reg(Dest) ← Reg(Src1) bitwise OR Reg(Src2)

Description
This is a bitwise logical or of the bits in Source 1 and Source 2. The result is placed in Destination.
4.4.11. xor - Exclusive Or

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Src2</th>
<th>Dest</th>
<th>Comp Func (12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 0 0 0 0 0 1 1 0 1</td>
</tr>
</tbody>
</table>

Assembler
xor rSrc1, rSrc2, rDest

Operation
Reg(Dest) ← Reg(Src1) bitwise exclusive-or Reg(Src2)

Description
This is a bitwise exclusive-or of the bits in Source 1 and Source 2. The result is placed in Destination.
4.4.12. mov - Move Register to Register

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Dest</th>
<th>Comp Func(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>00</td>
<td>' '</td>
<td>' '</td>
<td>00000000 ' ' '</td>
</tr>
</tbody>
</table>

**Assembler**

`mov rSrc1, rDest`

**Operation**

Reg(Dest) $\rightarrow$ Reg(Src1)

**Description**

This is a register to register move. It is implemented as `add rSrc1, rO, rDest`

This mnemonic is provided for convenience and clarity.
4.4.13. asr - Arithmetic Shift Right

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Dest</th>
<th>Comp Func(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0 1</td>
<td>0 0 0</td>
<td>0 b b b d d d</td>
</tr>
</tbody>
</table>

**Assembler**

asr rSrc1, rDest, #shift amount

**Operation**

Reg(Dest) ← Reg(Src1) >> shift amount (See below for explanation of shift amount)

The high order bits are sign extended.

**Description**

The contents of Source 1 are arithmetically shifted right by shift amount. The sign of the result is the same as the sign of Source 1. The result is stored in Destination. The range of shifts is from 1 to 32.

To determine the encoding for the shift amount, first subtract the shift amount from 32. The result can be encoded as 5 bits. Assume the 5-bit encoding is bbbef, where bbb is used in the final encoding. The bottom two bits (ef) are fully decoded to yield dddd in the following way:

<table>
<thead>
<tr>
<th>ef</th>
<th>dddd</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0001</td>
</tr>
<tr>
<td>01</td>
<td>0010</td>
</tr>
<tr>
<td>10</td>
<td>0100</td>
</tr>
<tr>
<td>11</td>
<td>1000</td>
</tr>
</tbody>
</table>

For example, to determine the bits required to specify the shift amount for the shift instruction

asr r4, r3, #5

first do (32-5) to get 27 and then encode 27 according to the above to get 1101000.
4.4.14. rotlb - Rotate Left by Bytes

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Src2</th>
<th>Dest</th>
<th>Comp Func(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0 0 0 1 1 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

**Assembler**

`rotlb rSrc1,rSrc2,rDest`

**Operation**

`Reg(Dest) ← Reg(Src1) rotated left by Reg(Src2)[30..31] bytes`

**Description**

This instruction rotates left the contents of Source 1 by the number of bytes specified in bit 30 and bit 31 of Source 2. For example,

`Reg(Src1) = AB01CD23#16`

`Reg(Src2) = 51#16`

`rotlb rSrc1,rSrc2,rDest`

`Reg(Dest) = 01CD23AB#16`
4.4.15. **rotlcb** - Rotate Left Complemented by Bytes

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Src2</th>
<th>Dest</th>
<th>Comp</th>
<th>Func(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1100</td>
<td>1</td>
<td>' '</td>
<td>' '</td>
<td>' '</td>
<td>' '</td>
</tr>
</tbody>
</table>

**Assembler**

`rotlcb rSrc1, rSrc2, rDest`

**Operation**

Reg(Dest) ← Reg(Src1) rotated left by BitComplement[Reg(Src2)<30..31>] bytes

**Description**

This instruction rotates left the contents of Source 1 by the number of bytes specified by using the bit complement of bits 30 and 31 in Source 2. For example,

Reg(Src1) = AB01CD23#16
Reg(Src2) = 51#16

`rotlcb rSrc1, rSrc2, rDest`

Rotate amount is Bit-Complement of `01#2 = 10#2 = 2`.
Reg(Dest) = CD23AB01#16
4.4.16. sh - Shift

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Src2</th>
<th>Dest</th>
<th>Comp Func(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Assembler

sh rSrc1,rSrc2,rDest,#shift amount

Operation

Reg(Dest) ← Bottom shift amount bits of Reg(Src2) || Top 32-shift amount bits of Reg(Src1)

Description

The shifter is a funnel shifter that concatenates Source 2 as the high order word with Source 1 and the shift amount is used to select a 32-bit field as the result. The range of shift amount is from 1 to 32.

The encoding of the shift amount is explained in the description of the asr instruction. For example, the instruction

sh r4,r12,r5,#7

places in r5 the bottom 7 bits of r2 (in the high order position) concatenated with the top 25 bits of r4. The bits to specify the shift amount are determined by first doing (32-7) to get 25. Then encode 25 to get 1100010.

The following table gives some more examples:

Assume

Reg(Src1) = 89ABCDEF#16
Reg(Src2) = 12345670#16

<table>
<thead>
<tr>
<th>Shift Amount</th>
<th>bbbdddd</th>
<th>Not Valid</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1111000</td>
<td></td>
<td>44D5E6F7</td>
</tr>
<tr>
<td>1</td>
<td>1110001</td>
<td></td>
<td>089ABCDE</td>
</tr>
<tr>
<td>4</td>
<td>1000001</td>
<td></td>
<td>567089AB</td>
</tr>
<tr>
<td>16</td>
<td>0010001</td>
<td></td>
<td>23456708</td>
</tr>
<tr>
<td>28</td>
<td>0000010</td>
<td></td>
<td>2468ACE1</td>
</tr>
<tr>
<td>31</td>
<td>0000001</td>
<td></td>
<td>12345670</td>
</tr>
<tr>
<td>32</td>
<td>0000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

shift sh - Shift shift
4.4.17. nop - No Operation

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>CompFunc(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>111</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1</td>
</tr>
</tbody>
</table>

Assembler
  nop

Operation
  Reg(0) ← Reg(0) + Reg(0)

Description
  This instruction does do much except take time and space. It is implemented as
  add r0,r0,r0
4.5. Compute Immediate Instructions

The compute immediate instructions have one source and one destination register. They provide a means to load a 17-bit constant that is stored as part of the instruction. Some of the instructions are used to access the special registers described in Section 2.3. In general, instructions that do not fit in with any of the other groups are placed here.
4.5.1. addi - Add Immediate

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Dest</th>
<th>Immed(17)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Assembler**

```
addi Src1, #Immed, Dest
```

**Operation**

```
Reg(Dest) ← SignExtend(Immed) + Reg(Src1)
```

**Description**

The value of the signed immediate constant is added to Source 1 and the result is stored in Destination.
4.5.2. jpc - Jump PC

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>CompFunc(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 1</td>
</tr>
</tbody>
</table>

**Assembler**

```
jpc
```

**Operation**

```
PCnext ← PC-4
```

**Description**

The PC chain should have been loaded with the 3 return addresses. PCnext is loaded with the contents of PC-4 which should contain a return address used for returning from an exception to user space.

This instruction should be the second and third of 3 jumps using the addresses in the PC chain. The first jump in the sequence should be `jpers` which also causes some state bits to change.
4.5.3. jpcrs - Jump PC and Restore State

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Comp Func(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1 110 6 0 0 0 10 0 0 0</td>
</tr>
</tbody>
</table>

Assembler

jpcrs

Operation

- PC shifting enabled
- \( \text{PSW}_{\text{current}} \leftarrow \text{PSW}_{\text{other}} \)
- \( \text{PC}_{\text{next}} \leftarrow \text{PC}_{-4} \)

Description

The PC chain should have been loaded with the 3 return addresses. \( \text{PC}_{\text{next}} \) is loaded with the contents of \( \text{PC}_{-4} \) which should contain the first return address when returning from an exception to user space.

This instruction should be the first of 3 jumps using the addresses in the PC chain. The next two instructions should be jpcs to jump to the 2 other instructions needed to restart the machine.

The machine changes from system to user state at the end of the ALU cycle of the jpcrs instruction. The PSW is changed at this time as well.

When this instruction is executed in user state, the PSW is not changed. The effective result is a jump using the contents of PC-4 as the destination address.
4.5.4. jspci - Jump Indexed and Store PC

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Src1</th>
<th>Dest</th>
<th>Immed(17)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Assembler

jspci rSrc1,#Immed,rDest

Operation

\[ \text{PC} \leftarrow \text{Reg(Src1)} + \text{SignExtend(Immed)} \]
\[ \text{Reg(Dest)} \leftarrow \text{PCcurrent + 1} \]

Description

This instruction has two delay slots. The address of the instruction after the two delay slots is stored in the Destination register. This is the return location. The immediate value is sign extended and added to the contents of Source 1. This is the jump destination so it is jammed into the PC. The displacement is a 17-bit signed word displacement.

This instruction provides a fast linking mechanism to subroutines that are called via a trap vector.
4.5.5. movfrs - Move from Special Register

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Dest</th>
<th>CompFunc(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Assembler
movfrs SpecialReg, rDest

Operation
Reg(Dest) ⇔ Reg(Spec)

Description
This instruction is used to copy the special registers described in Section 2.3 into a general register. The contents of the special register are put in the destination register. The value used in the Spec field for each of the special registers is shown in the table below along with the assembler mnemonic.

<table>
<thead>
<tr>
<th>SpecialReg</th>
<th>Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>psw</td>
<td>001</td>
</tr>
<tr>
<td>md</td>
<td>010</td>
</tr>
<tr>
<td>pcm4</td>
<td>100</td>
</tr>
</tbody>
</table>

The PSW (psw) can be read in both system and user state.

A move from pcm4 causes the PC chain to shift after the move.
4.5.6. movtos - Move to Special Register

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Srl</th>
<th>Comp Func(12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1 0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>spec</td>
</tr>
</tbody>
</table>

**Assembler**

movtos rSrcl.SpecialReg

**Operation**

Reg(Spec) = Reg(Srcl)

**Description**

This instruction is used to load the special registers described in Section 2.3. The contents of the Source 1 register is put in the special register. The value used in the Spec field for each of the special registers is shown in the table below along with the assembler mnemonic.

<table>
<thead>
<tr>
<th>Special Reg</th>
<th>Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>psw</td>
<td>001</td>
</tr>
<tr>
<td>md</td>
<td>010</td>
</tr>
<tr>
<td>pcml</td>
<td>100</td>
</tr>
</tbody>
</table>

Accessing the PSW (psw) requires the processor to be in system state. Otherwise the instruction is a nop in user state.

A move topcml causes the PC chain to shift after the move.

After a move to md, one cycle may be needed before an mstart or mstep instruction to settle some control lines to the ALU.
4.5.7. trap - Trap Unconditionally

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
<th>Vector(8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>1010 0000</td>
<td>[010000000001'1'1'1'9'9'9'9']</td>
</tr>
</tbody>
</table>

**Assembler**

\[
\text{trap Vector}
\]

**Operation**

Stop PC shifting
\[
\text{PC} \leftarrow \text{Vector} \ll 3
\]
\[
\text{PSW}\text{other} \leftarrow \text{PSW}\text{current}
\]

**Description**

The shifting of the PC chain is stopped and the PC is loaded with the contents of the Vector field shifted left by 3 bits. The PSW of the user space is saved.

This is an unconditional trap. The instruction is used to go to a system space routine from user space. The state of the machine changes from user to system after the ALU cycle of the trap instruction.

The trap instruction cannot be placed in the first delay slot of a branch, \text{jspci}, \text{jpc}, or \text{jpers} instruction. See Appendix VI for more details.

The assembler should convert Vector to its one’s complement form before generating the machine instruction. i.e., the machine instruction contains the one’s complement of the vector.
4.58. **hsc - Halt and Spontaneously Combust**

<table>
<thead>
<tr>
<th>TY</th>
<th>OP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 0 111 1 1 110 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

**Assembler**

```
    hsc
```

**Operation**

```
    Reg(31) <= PC
```

The processor stops fetching instructions and self destructs.

Note that the contents of Reg(31) are actually lost.

**Description**

This is executed by the processor when a protection violation is detected. It is a privileged instruction available only on the -NSA versions of the processor.
Appendix I
Some Programming Issues

This appendix contains some programming issues that must be stated but have not been included elsewhere in this document.

1. Address 0 in both system and user space should have a **nop** instruction. When an exception occurs during a squashed branch, the PCs for the instructions that have been squashed are set to 0 so that when these instructions are restarted they **will** not affect any state. The **nop** at address 0 is also convenient for some sequences when it is necessary to load a null instruction into the PC chain.

2. The instruction cache contains valid bits for each of the 32 buffers. There is also a bit to indicate whether the buffer contains system or user space instructions. When it is necessary to invalidate the instruction cache entries for a context switch between user processes, a system space routine is executed that jumps to 32 strategic locations to force all of the system bits to be set in the tags. Thus when the new user process begins, the cache is flushed of the previous user process. An example code sequence is shown at the end of this appendix.

3. After an interrupt occurs, no registers should be accessed for two instructions so that the tags in the bypass registers can be flushed. If a register access is done, then it is possible that the instruction will get values out of the bypass registers written by the previous context instead of the register file. This should not be a problem because the PCs must be saved first **anyways**. Since this happens in system space, the interrupt handler can just be written so that the improper bypassing does not occur.

4. There is no instruction that can be used to implement synchronization primitives such as test-and-set. The proposed method is to use Dekker’s algorithm or some other software scheme [3] but if this proves to be insufficient then a **load-locked** instruction can be implemented as a coprocessor **instruction for the cache controller**. This instruction will lock the bus until another coprocessor instruction is used to unlock it. This can be used to implement a read-modify-write cycle.

5. A long constant can be loaded with the following sequence:

   ```
   .data
   label1:
   .word 0xABCD1234
   .text
   ld  label1[r0],r5
   r5 now contains ABCD1234
   ```

6. If a privileged instruction is executed in user space none of the state bits can be changed. This means that writing the PSW becomes a **nop**. Reading the PSW returns the correct value. Trying to execute a **jpcrs** only does a jump to the address in PC-4 and does not change the PSW. There is no trap taken for a privilege violation.

7. Characters can be inserted and extracted with the following sequences:

   For each of these examples, assume
   r2 initially contains stuv
   r3 initially contains wxyz
   where s, t, u, v, w, x, y and z are byte values.

   ; Byte insertion - byte u gets replaced by w
   addi r0,#2,r1
   rotlb r2,r1,r2 ; r2 <-- uvst
   sh r3,r2,#24 ; r2 <-- vstw
   rotlcb r2,r1,r2 ; r2 <-- stvw
   ;
   ; Extract byte - extract byte u from r2 and place it in r3
   addi r0,#2,r1
   rotlb r2,r1,r3 ; r3 <-- uvst
   sh r3,r0,#24 ; r3 <-- u

Programming Issues
This routine will jump through low core to flush the cache by setting all the tags to be in system space. Note that this routine will also blow away any entry in the cache that called this routine but to make it general it will have to since you don't want to have to figure out where you came from. That is called from a trap as it knows where to return to.

The sequence of jump locations is designed to account for the behaviour of the ring counter that is used to determine the next instruction cache block to be replaced. It is not sufficient to access the locations in sequence.

The "makeup u" means that "u" mop instructions should be inserted.

This module should be loaded starting at address 0x1800.

Text

Dump

Quote

10x.1800:

J0pe1 r0,0x1810,

J0pe1 r0,0x1820,

makeup 15

10x.1810:

J0pe1 r0,0x1840,

J0pe1 r0,0x1850,

makeup 15

10x.1820:

J0pe1 r0,0x1860,

J0pe1 r0,0x1870,

makeup 15

10x.1830:

J0pe1 r0,0x1880,

makeup 15

10x.1840:

J0pe1 r0,0x1890,

makeup 15

10x.1850:

J0pe1 r0,0x18a0,

makeup 15

10x.1860:

J0pe1 r0,0x18b0,

makeup 15

10x.1870:

J0pe1 r0,0x18c0,

makeup 15

10x.1880:

J0pe1 r0,0x18d0,

makeup 15

10x.1890:

J0pe1 r0,0x18e0,

makeup 15

10x.18a0:

J0pe1 r0,0x18f0,

makeup 15

10x.18b0:

J0pe1 r0,0x1900,

makeup 15

10x.18c0:

J0pe1 r0,0x1910,

makeup 15

10x.18d0:

J0pe1 r0,0x1920,

makeup 15

10x.18e0:

J0pe1 r0,0x1930,

makeup 15

10x.18f0:

J0pe1 r0,0x1940,

makeup 15

10x.1900:

J0pe1 r0,0x1950,

makeup 15

10x.1910:

J0pe1 r0,0x1960,

makeup 15

10x.1920:

J0pe1 r0,0x1970,

makeup 15

10x.1930:

J0pe1 r0,0x1980,

makeup 15

10x.1940:

J0pe1 r0,0x1990,

makeup 15

10x.1950:

J0pe1 r0,0x19a0,

makeup 15

10x.1960:

J0pe1 r0,0x19b0,

makeup 15

10x.1970:

J0pe1 r0,0x19c0,

makeup 15

10x.1980:

J0pe1 r0,0x19d0,

makeup 15

10x.1990:

J0pe1 r0,0x19e0,

makeup 15

10x.19a0:

J0pe1 r0,0x19f0,

makeup 15

10x.19b0:

J0pe1 r0,0x1a0,

makeup 15

10x.19c0:

J0pe1 r0,0x1a1,

makeup 15

10x.19d0:

J0pe1 r0,0x1a2,

makeup 15

10x.19e0:

J0pe1 r0,0x1a3,

makeup 15

10x.19f0:

J0pe1 r0,0x1a4,

makeup 15

10x.1a0:

J0pe1 r0,0x1a5,

makeup 15

10x.1a1:

J0pe1 r0,0x1a6,

makeup 15

10x.1a2:

J0pe1 r0,0x1a7,

makeup 15

10x.1a3:

J0pe1 r0,0x1a8,

makeup 15

10x.1a4:

J0pe1 r0,0x1a9,

makeup 15

10x.1a5:

J0pe1 r0,0x1aa,

makeup 15

10x.1a6:

J0pe1 r0,0x1ab,

makeup 15

10x.1a7:

J0pe1 r0,0x1ac,

makeup 15

10x.1a8:

J0pe1 r0,0x1ad,

makeup 15

10x.1a9:

J0pe1 r0,0x1aa,

makeup 15

10x.1aa:

J0pe1 r0,0x1ab,

makeup 15

10x.1ab:

J0pe1 r0,0x1ac,

makeup 15

10x.1ac:

J0pe1 r0,0x1ad,

makeup 15

10x.1ae:

J0pe1 r0,0x1af,

makeup 15

10x.1af:

J0pe1 r0,0x1a0,

makeup 15

10x.1b0:

J0pe1 r0,0x1b1,

makeup 15

10x.1b1:

J0pe1 r0,0x1b2,

makeup 15

10x.1b2:

J0pe1 r0,0x1b3,

makeup 15

10x.1b3:

J0pe1 r0,0x1b4,

makeup 15

10x.1b4:

J0pe1 r0,0x1b5,

makeup 15

10x.1b5:

J0pe1 r0,0x1b6,

makeup 15

10x.1b6:

J0pe1 r0,0x1b7,

makeup 15

10x.1b7:

J0pe1 r0,0x1b8,

makeup 15

10x.1b8:

J0pe1 r0,0x1b9,

makeup 15

10x.1b9:

J0pe1 r0,0x1ba,

makeup 15

10x.1ba:

J0pe1 r0,0x1bb,

makeup 15

10x.1bb:

J0pe1 r0,0x1bc,

makeup 15

10x.1bc:

J0pe1 r0,0x1bd,

makeup 15

10x.1bd:

J0pe1 r0,0x1be,

makeup 15

10x.1be:

J0pe1 r0,0x1bf,

makeup 15

10x.1bf:

J0pe1 r0,0x1c0,

makeup 15

10x.1c0:

J0pe1 r0,0x1c1,

makeup 15

10x.1c1:

J0pe1 r0,0x1c2,

makeup 15

10x.1c2:

J0pe1 r0,0x1c3,

makeup 15

10x.1c3:

J0pe1 r0,0x1c4,

makeup 15

10x.1c4:

J0pe1 r0,0x1c5,
Appendix II
Opcode Map

This is a summary of how the bits in the instruction opcodes have been assigned. The first sections will show how the bits in the OP and Comp Func fields are assigned. Then the opcode map of the complete instruction set will be given.

11.1. OP Field Bit Assignments

The OP bits are bits 2-4 in all instructions. For memory type instructions the bits have no particular meaning by themselves. For branch type instructions the bits in the OP field (also known as the Cond field) are assigned as follows:

Bit 2 Set to 0 if branch on condition true, set to 1 if branch on condition false
Bits 3-4 Condition upon which the branch decision is made. 00 unused, 01 = Z, 10 = C, 11 = N ⊕ V

For compute type instructions the bits are assigned as follows:

Bit 2 Set to 1 if the ALU always drives the result bus for the instruction
Bit 3 Set to 0
Bit 4 Set to 1 if the shifter always drives the result bus for the instruction

For compute immediate type instructions the bits are assigned as follows:

Bit 2 Set to 1 if the ALU always drives the result bus for the instruction
Bits 3-4 These bits have no particular meaning by themselves

11.2. Comp Func Field Bit Assignments

The Comp Func bits are bits 20 through 31 in the compute and compute immediate type instructions. The bits are assigned according to whether they are being used by the ALU or the shifter. The bits for the ALU are assigned in the following way:

Bits 20-22 Unused
Bit 23 Set to 1 for dstep, 0 otherwise
Bit 24 Set to 1 for multiply instructions (mstart, mstep), 0 otherwise
Bit 25 Carry in to the ALU

Bits 26-29 Input to the \( P \) function block.

Bit 26 \( \text{Src1} \cdot \text{Src2} \)
Bit 27 \( \text{Src1} \cdot \text{Src2} \)
Bit 28 \( \text{Src1} \cdot \text{Src2} \)
Bit 29 \( \text{Src1} \cdot \text{Src2} \)

Bits 30-31 Input to the G function block.

Bit 30 0 for ALU add operation, 1 otherwise
Bit 31 0 for ALU subtract operation, 1 otherwise

The bits for the shifter are assigned as follows:

Bits 20-21 Unused
Bit 22 Set to 1 for funnel shift operation (sh instruction)
Bit 23 Set to 1 for arithmetic shift operation (asr instruction)
Bit 24 Set to 1 for byte rotate instructions (rotlb, rotlcb)
Bit 25
For byte rotate instructions, set to 1 if `rodlb`, 0 if `rotleb`

Bits 25-31
Shift amount for funnel and arithmetic shift operations (sh and asr instructions). The range is 0 to 31 bits. Although this can be encoded in five bits, the two low-order bits are fully decoded; therefore, the field is seven bits. The two low-order bits are decoded as follows: 0 = bit 31, 1 = bit 30, 2 = bit 29, 3 = bit 28. For example, a shift amount of 30 would become 1110100 in this seven-bit encoding scheme.
# 11.3. Opcode Map of All Instructions

## Memory Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>TY</th>
<th>OP</th>
<th>Func</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld</td>
<td>10</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>st</td>
<td>10</td>
<td>01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldf</td>
<td>10</td>
<td>10</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>stf</td>
<td>10</td>
<td>11</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>ldt</td>
<td>10</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stt</td>
<td>10</td>
<td>01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>movfrc</td>
<td>10</td>
<td>10</td>
<td></td>
<td>Srcl=0, *</td>
</tr>
<tr>
<td>movtoc</td>
<td>10</td>
<td>11</td>
<td></td>
<td>Srcl=0</td>
</tr>
<tr>
<td>aluc</td>
<td>10</td>
<td>11</td>
<td></td>
<td>Srcl=0, Dest=0, *</td>
</tr>
</tbody>
</table>

## Branch Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>TY</th>
<th>COND</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>00</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>bge</td>
<td>00</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>bhs</td>
<td>00</td>
<td>010</td>
<td></td>
</tr>
<tr>
<td>blo</td>
<td>00</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>blt</td>
<td>00</td>
<td>011</td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>00</td>
<td>101</td>
<td></td>
</tr>
</tbody>
</table>

## Compute Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>TY</th>
<th>OP</th>
<th>Func</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>01</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dstep</td>
<td>01</td>
<td>000</td>
<td></td>
<td>Src1=0</td>
</tr>
<tr>
<td>mstart</td>
<td>01</td>
<td>000</td>
<td></td>
<td>000101110110</td>
</tr>
<tr>
<td>mstep</td>
<td>01</td>
<td>000</td>
<td></td>
<td>00001110110</td>
</tr>
<tr>
<td>sub</td>
<td>01</td>
<td>100</td>
<td></td>
<td>00000110110</td>
</tr>
<tr>
<td>subnc</td>
<td>01</td>
<td>100</td>
<td></td>
<td>000001011010</td>
</tr>
<tr>
<td>and</td>
<td>01</td>
<td>100</td>
<td></td>
<td>000000010011</td>
</tr>
<tr>
<td>bic</td>
<td>01</td>
<td>100</td>
<td></td>
<td>000000011011</td>
</tr>
<tr>
<td>not</td>
<td>01</td>
<td>100</td>
<td></td>
<td>000000011111</td>
</tr>
<tr>
<td>or</td>
<td>01</td>
<td>100</td>
<td></td>
<td>000000011111</td>
</tr>
<tr>
<td>xor</td>
<td>01</td>
<td>100</td>
<td></td>
<td>000000011111</td>
</tr>
<tr>
<td>mov</td>
<td>01</td>
<td>100</td>
<td></td>
<td>000000011011</td>
</tr>
<tr>
<td>asr</td>
<td>01</td>
<td>001</td>
<td></td>
<td>000100bbddd</td>
</tr>
<tr>
<td>rctlb</td>
<td>01</td>
<td>001</td>
<td></td>
<td>00011000000</td>
</tr>
<tr>
<td>rotlcb</td>
<td>01</td>
<td>001</td>
<td></td>
<td>00010000000</td>
</tr>
<tr>
<td>sh</td>
<td>01</td>
<td>001</td>
<td></td>
<td>00100bbddd</td>
</tr>
<tr>
<td>nop</td>
<td>01</td>
<td>100</td>
<td></td>
<td>000000011001</td>
</tr>
</tbody>
</table>

## Compute Immediate Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>TY</th>
<th>OP</th>
<th>Func</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>11</td>
<td>100</td>
<td>Immed</td>
<td>* (Immed is a 17-bit signed constant)</td>
</tr>
<tr>
<td>jspeci</td>
<td>11</td>
<td>000</td>
<td>Immed</td>
<td>*</td>
</tr>
<tr>
<td>jpc</td>
<td>11</td>
<td>101</td>
<td>000000000011</td>
<td></td>
</tr>
<tr>
<td>jpcrs</td>
<td>11</td>
<td>111</td>
<td>000000000011</td>
<td></td>
</tr>
<tr>
<td>movfrs</td>
<td>11</td>
<td>011</td>
<td>0000000000rrrr</td>
<td></td>
</tr>
<tr>
<td>movtos</td>
<td>11</td>
<td>010</td>
<td>0000000000rrrr</td>
<td></td>
</tr>
<tr>
<td>trap</td>
<td>11</td>
<td>110</td>
<td>0vvvvvvvvvvv011</td>
<td></td>
</tr>
<tr>
<td>unused</td>
<td>11</td>
<td>001</td>
<td></td>
<td>Src1=0, vvvvvvvvvvv=vector</td>
</tr>
</tbody>
</table>

A star (*) indicates an instruction that has its Dest field in the position where the Src2 field normally sits. This can also be determined by decoding the MSB of the type field and the middle bit of the OP field.

---

**Opcode Map**
Appendix III
Floating Point Instructions

This describes the floating point opcodes and formats of the instructions implemented in the MIPS-X Instruction Level Simulator (milsx).

III.1. Format

All floating point numbers are represented in one 32-bit word as shown in Fig. III-1. The fields represent the following floating point number:

\((-1)^n \times 2^{exp-127} \times (1 + fraction)\)

This is an approximate IEEE floating point format.

\[
\begin{array}{ccccccccccccccccccc}
\text{exp (8 bits)} & & & & & & & & & & & & & \text{fraction (23 bits)}
\end{array}
\]

\[
\begin{array}{ccccccccccccccccccc}
\text{I} & & & & & & & & & & & & & \text{I}
\end{array}
\]

Figure III-1: Floating Point Number Format

111.2. Instruction Timing

All floating point instructions are assumed to take one cycle to execute. More realistic timing numbers can be derived by multiplying the number output by mils by an appropriate constant.

111.3. Load and Store Instructions

There are 16 floating point registers. They are loaded and stored using the \textit{ldf} and \textit{stf} instructions defined in the instruction set. Moves between the floating point registers and the main processor are done using the \textit{movif} and \textit{movfi} instructions. These use the movtoc and \textit{movfrc} formats defined in the instruction set. Note that only 4 of the 5 bits that specify a floating point register in the \textit{ldf}, \textit{stf}, \textit{movif} and \textit{movfi} instructions are used.

111.4. Floating Point Compute Instructions

The format of the floating point compute instructions is the one shown in the description of the \textit{aluc} coprocessor instruction. The coprocessor number (\textit{COP#}) is 0 for the floating point coprocessor. The \textit{Func} field specifies the floating point operation to be performed.
111.5. Opcode Map of Floating Point Instructions

In the following table:
- \texttt{rl}, \texttt{r2} are CPU registers from \texttt{r0..r31}
- \texttt{f1}, \texttt{f2} are floating point registers from \texttt{f0..f15}
- \( n \) is an integer expression

<table>
<thead>
<tr>
<th>Instruction</th>
<th>TY</th>
<th>OP</th>
<th>Func</th>
<th>Operation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>fadd</td>
<td>10</td>
<td>101</td>
<td>000000</td>
<td>f2 \rightarrow f1 + f2</td>
<td>Source=0, Destination=0</td>
</tr>
<tr>
<td>fsub</td>
<td>10</td>
<td>101</td>
<td>000001</td>
<td>f2 \rightarrow f1 - f2</td>
<td>Source=0, Destination=0</td>
</tr>
<tr>
<td>fmul</td>
<td>10</td>
<td>101</td>
<td>000010</td>
<td>f2 \rightarrow f1 \times f2</td>
<td>Source=0, Destination=0</td>
</tr>
<tr>
<td>fdiv</td>
<td>10</td>
<td>101</td>
<td>000111</td>
<td>f2 \rightarrow f1 / f2</td>
<td>Source=0, Destination=0</td>
</tr>
<tr>
<td>cvtif</td>
<td>10</td>
<td>101</td>
<td>000100</td>
<td>f2 \rightarrow \text{float}(f1)</td>
<td>Source=0, Destination=0</td>
</tr>
<tr>
<td>cvtfi</td>
<td>10</td>
<td>101</td>
<td>000101</td>
<td>f2 \rightarrow \text{int}(f1)</td>
<td>Convert integer to float</td>
</tr>
<tr>
<td>imul</td>
<td>10</td>
<td>101</td>
<td>000110</td>
<td>f2 \rightarrow f1 \times f2</td>
<td>Source=0, Destination=0</td>
</tr>
<tr>
<td>idiv</td>
<td>10</td>
<td>101</td>
<td>000111</td>
<td>f2 \rightarrow f1 / f2</td>
<td>Source=0, Destination=0</td>
</tr>
<tr>
<td>mod</td>
<td>10</td>
<td>101</td>
<td>001000</td>
<td>f2 \rightarrow f1 \text{ mod } f2</td>
<td>Source=0, Destination=0</td>
</tr>
<tr>
<td>movif</td>
<td>10</td>
<td>111</td>
<td>001001</td>
<td>f1 \rightarrow rl</td>
<td>Source=0, CS1=0</td>
</tr>
<tr>
<td>movfi</td>
<td>10</td>
<td>101</td>
<td>001010</td>
<td>rl \rightarrow f1</td>
<td>Source=0, CS2=0</td>
</tr>
<tr>
<td>ldf</td>
<td>10</td>
<td>101</td>
<td>001010</td>
<td>rl \rightarrow f1</td>
<td>Source=0, CS2=0</td>
</tr>
<tr>
<td>stf</td>
<td>10</td>
<td>110</td>
<td></td>
<td>rl \rightarrow f1</td>
<td>Source=0, CS2=0</td>
</tr>
</tbody>
</table>

Floating Point
Appendix IV

Integer Multiplication and Division

This appendix describes the multiplication and division support on MIPS-X. The philosophy behind why the current implementation was chosen is described first and then the instructions for doing multiplication and division are described.

IV.1. Multiplication and Division Support

The goal of the multiplication and division support in MIPS-X is to provide a reasonable amount of support with the smallest amount of hardware possible. Speed ups can be obtained by realizing that most integer multiplications are used to obtain a 32-bit result, not a 64-bit result. The result is usually the input to another operation, or it is the address of an array index. In either case a number larger than 32 bits would not make sense. Since the result is less than 32 bits, one of the operands is most likely to be less than 16 bits or there will be an overflow. In general this means that only about 16 l-bit multiplication or division steps are required to generate the final answer. For very small constants, instructions can be generated inline instead of using a general multiplication or division routine. Therefore, it was felt that there was no great advantage to implement a scheme that could do more than 1 bit at a time such as Booth multiplication.

The other advantage of only generating a 32-bit result is that it is possible to do multiplication starting at the MSB of the multiplier meaning that the same hardware can be used for multiplication and division. The required hardware is a single register, the MD register, that can shift left by one bit each cycle, and an additional multiplexer at the source 1 input of the ALU, that selects the input or two times the input for the source 1 operand.

IV.2. Multiplication

Multiplication is done with the simple l-bit shift and add algorithm except that the computation is started from the most significant bit instead of the least significant bit of the multiplier. The instruction that implements one step of the algorithm is called rnstep. For

\texttt{rnstep rSrc1,rSrc2,rDest}

the operation is:

If the MSB of the MD register is 1
then
\( r\text{Dest} \leftarrow 2 \times r\text{Src1} + r\text{Src2} \)
else
\( r\text{Dest} \leftarrow 2 \times r\text{Src1} \)

Shift left MD

For signed multiplication, the first step is different from the rest. If the MSB of the multiplier is 1, the multiplicand should be subtracted from 0. The instruction called \texttt{mstart} is provided for this purpose. For

\texttt{mstart rSrc2,rDest}

the operation is
If the MSB of the MD register is 1
then
\( r_{\text{Dest}} \leftarrow 0 - r_{\text{Src2}} \)
else
\( r_{\text{Dest}} \leftarrow 0 \)
Shift left MD

To show the simplest implementation of a multiplication routine assume that the following registers have been assigned and loaded:

- \( r_{\text{Mer}} \) is the multiplier,
- \( r_{\text{Mand}} \) is the multiplicand,
- \( r_{\text{Dest}} \) is the result register,
- \( r_{\text{Link}} \) is the jump linkage register.

Then,

\[
\text{movtos } r_{\text{Mer}}, r_{\text{MD}} \quad ; \text{Move the multiplier into MD}
\]
\[
\text{nop} \quad ; \text{Needed for hardware timing reasons--see movtos}
\]
\[
\text{mstart } r_{\text{Mand}}, r_{\text{Dest}} \quad ; \text{Do the first mstep. Result goes into } r_{\text{Dest}}
\]
\[
\text{mstep } r_{\text{Dest}}, r_{\text{Mand}}, r_{\text{Dest}} \quad ; \text{Repeat 31 times}
\]
\[
\text{jspci } r_{\text{Link}}, #0, r_{\text{0}} \quad ; \text{Return}
\]

It is possible to speed up the routine by using the assumption described previously that the numbers will not both be a full 32 bits long. The simplest scheme is to check to see if the multiplier is less than 8 bits long. Some statistics indicate that this occurs frequently.

The routine shown in Figure IV-1 implements multiplication with less than 32 \textit{msteps} on average. It will actually do a full 32 \textit{msteps} if it is necessary. In this case it is most likely that overflow will occur and this can be detected if the \textit{V} bit in the PSW is clear so that a trap on overflow will occur. Assume that the registers \( r_{\text{Mer}}, r_{\text{Mand}}, \text{and } r_{\text{Dest}} \) have been assigned and loaded as in the previous example. Two temporary registers, \( r_{\text{Temp1}} \) and \( r_{\text{Temp2}} \) are also required.

The number of cycles required, not including the instructions needed for the call sequence is shown in Table IV-1. Compare this with the simple routine using just 32 steps which requires 35 instructions to do the multiplication and a Booth 2-bit algorithm that will need about 19 instructions. It can be observed that if most multiplications require 8 or less \textit{msteps}, then this routine will be faster than just doing 32 \textit{msteps} all the time.

\section*{IV.3. Division}

For division, the same set of hardware is used, except the ALU is controlled differently. The algorithm is a restoring division algorithm. Both of the operands must be positive numbers. Signed division is not supported as it is too hard to do for the hardware required [2].

The dividend is loaded in the MD register and the register that will contain the remainder (\( r_{\text{Rem}} \)) is initialized to 0. The divisor is loaded into another register called (\( r_{\text{Dor}} \)). The result of the division (quotient) will be in MD. For

\[
\text{ds tep } r_{\text{Rem}}, r_{\text{Dor}}, r_{\text{Rem}}
\]

the operation is:

**Multiplication and Division**
FAST, UNCHECKED, SIGNED MUL:

```
; MUL:
; fast, unchecked, signed multiply
; 
; rLink = link
; rMand = src2
; rDest = rMer - src1/dest
; rTemp1 = temp
; rTemp2 = temp
; 
; Note: This code has been reorganized

MUL:

asr rMer, rTemp2, #7 ; Test for positive 8-bit number
bne rTemp2, r0, lnot8
sh r0, rMer, rTemp1, #24 ; assume 8 bit
movtos rTemp1, md
mstart rMand, rDest
mstep rDest, rMand, rDest

mul8bit:
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest

lnot8:
addi rTemp2, #1, rTemp2
beqsq rTemp2, r0, mul8bit
mstart rMand, rDest
mstep rDest, rMand, rDest
movtos rDest, md
mstart rMand, rDest
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest

mul8bit:
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest

lnot8:
addi rTemp2, #1, rTemp2
beqsq rTemp2, r0, mul8bit
mstart rMand, rDest
mstep rDest, rMand, rDest
movtos rDest, md
mstart rMand, rDest
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest

24 msteps

mstep rDest, rMand, rDest
jspci rLink, #0, r0
mstep rDest, rMand, rDest
mstep rDest, rMand, rDest
```

Figure IV-1: Signed Integer Multiplication

Multiplication and Division
Table IV-1: Number of Cycles Needed to do a Multiplication

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of msteps needed</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>Number of cycles with positive multiplier</td>
<td>13</td>
<td>42</td>
</tr>
<tr>
<td>Number of cycles with negative multiplier</td>
<td>15</td>
<td>42</td>
</tr>
</tbody>
</table>

Set ALUsrcl input to $2 \times r_{Rem} + \text{MSB}(r_{MD})$
Set ALUsrc2 input to $r_{Dor}$
$ALU_{output} \leftarrow ALUsrcl - ALUsrc2$

If $\text{MSB}(ALU_{output})$ is 1
then
$r_{Rem} \leftarrow ALUsrcl$
$r_{MD} \leftarrow 2 \times r_{MD}$
else
$r_{Rem} \leftarrow ALU_{output}$
$r_{MD} \leftarrow 2 \times r_{MD} + 1$

At the end of 32 dsteps the quotient will be in the MD register, and the remainder is in $r_{Rem}$.

A routine for doing division is shown in Figure IV-2. The dividend is passed in $r_{Dend}$ and the divisor in $r_{Dor}$. At the end, the quotient is in MD and $r_{Quot}$ and the remainder is in $r_{Rem}$. Note that $r_{Dend}$ and $r_{Rem}$ can be the same register, and $r_{Dor}$ and $r_{Quot}$ can be the same register. The dividend and divisor are checked to make sure they are positive. This routine does a 32-bit by 32-bit division so no overflow can occur.

The number of cycles needed, not including the calling sequence and assuming the operands are positive, is shown in Table IV-2.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of dsteps needed</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>Number of cycles needed</td>
<td>34</td>
<td>60</td>
</tr>
</tbody>
</table>

Table IV-2: Number of Cycles Needed to do a Divide

Multiplication and Division
DIV: fast, unchecked, signed divide (should check for zero divide)

DIV:

; 
; DIV
; 
; rDor = rQuot = src2/dst (divisor/quotient)
; rTemp1 = temp (trashed)
; rTemp2 = temp (trashed)
;
Note: This code has been reorganized
;
DIV: 

mov rDend,rTemp2 ; dividend > 0 ?
bge rDend,r0,lcinit1

nop

nop

sub r0,rDend,rDend ; make dividend > 0

bgesq rDor,r0,lcinit2 ; divisor > 0 ?

addi r0,#0xff,rTemp1 ; check for 8-bit dividend

nop

sub r0,rTemp2,rTemp2 ; rTemp2 > 0 if positive result

sub r0,rDor,rDor ; make divisor > 0

addi r0,#0xff,rTemp1

lcinit1:

bitsq rTemp1,rDend,ldivfull ; do 8-bit check
movtos rDend,md ; start 32-bit divide
mdv r0,rRem
sh r0,rDend,ldivfull,8 ; shift up divisor to do 8 bits
movtos rDend,md ; start 8-bit divide

beq r0,r0,ldivloop ; start 32-bit divide

mov r0,rRem

addi r0, #8, rTemp1 ; loop counter

ldivfull:

addi r0, #32, rTemp1 ; do full 32 dsteps

ldivloop:

dstep rRem,rDor,rRem

dstep rRem,rDor,rRem

ldivloopr:

dstep rRem,rDor,rRem

dstep rRem,rDor,rRem

dstep rRem,rDor,rRem

dstep rRem,rDor,rRem

addi rTemp1, #8, rTemp1 ; decrement loop counter

dstep rRem,rDor,rRem

bnesq rTemp1,r0,ldivloopr

dstep rRem,rDor,rRem

dstep rRem,rDor,rRem

movfrs md,rQuot ; get result

bge rTemp2,r0,lcinit3 ; check if need to adjust sign of result

nop

nop

sub r0,rQuot,rQuot ; adjust sign of result

lcinit3:

jpci rLink,#0,rLink ; return

Figure W-2: Signed Integer Division

Multiplication and Division
Multiplication and Division
Multiprecision arithmetic is not a high priority but it is desirable to make it possible to do. The minimal support necessary will be provided. The most straightforward way to do this would seem to be the addition of a carry bit to the PSW. However, this turns out to be extremely difficult.

The following program segments are examples of doing double precision addition and subtraction. The only addition required to the instruction set is the **Subtract with No Carry (subnc)** instruction. This is only an addition to the assembly language and not to the hardware.

Assume that there are 2 double precision operands (A and B) and a double precision result to be computed (C). Assume that the necessary registers have been loaded.

;Double precision addition
```
add rAhi, rBhi, rChi ;add high words
sub r0, rBlo, rClo ;get -rBlo; branch does subtract
bhssq rAlo, rClo, 11 ;branch if carry generated
addi rChi, #1, rChi ;add 1 to high word if carry
11: add rAlo, rBlo, rClo ;add low words
```

;Double precision subtraction
```
subnc rAhi, rBhi, rChi ;subtract high words
bhssq rAlo, rBlo, 11 ;branch if carry generated
addi rChi, #1, rChi ;add 1 to high word if carry
11: sub rAlo, rBlo, rClo ;subtract low words
```
Appendix VI
Exception Handling

An exception is defined as either an event that causes an interrupt or a trap instruction that can be thought of as a software interrupt. The two sequences cause similar actions in the processor hardware. Because there is a branch delay of 2, three PCs from the PC chain must be saved and restarted on an interrupt. Three PCs are needed in the event that a branch has occurred and fallen off the end of the chain. The two branch slot instructions and the branch destination are saved for restarting. Restarting a trap is slightly different and is explained later. See Section 2.4 for a description of the PSW during interrupts, exceptions, and traps.

VI.1. Interrupts

Interrupts are asynchronous events that the programmer has no control over. Because there are several instructions executing at the same time, it is necessary to save the PCs of all the instructions currently executing so that the machine can be properly restarted after an interrupt. The PCs are held in the PC chain. When an interrupt occurs, the PC chain is frozen (stops shifting in new values) to allow the interrupt routine to save the PCs of the three instructions that need to be restarted. These are the PCs of the instructions that are in the RF, ALU and MEM cycles of execution. This means that no further exceptions can occur while the PCs are being saved. When the interrupt sequence begins, the interrupts are disabled. PSWcurrent is copied into PSWother and the machine begins execution in system state. The contents of PSWother should be saved if interrupts are to be enabled before the return from the interrupt. The contents of the MD register must also be saved and restored if any multiplication or division is done. If the interrupt routine is very short and interrupts can be left off, it is possible to just leave the PC chain frozen, otherwise the three PCs must be saved. To save the PCs use movfrs with PC-4 as the source. The PC chain shifts after each read of PC-4.

The interrupt routine will start execution at location 0. It must look at a register in the interrupt controller to determine how to handle the interrupt. This sequence is yet to be specified.

To return from an interrupt, interrupts must first be disabled to allow the state of the machine to be restored. The PSW must be restored and the PC chain loaded with the return addresses. The PC chain is loaded by writing to PC-1 and it shifts after each write to PC-1. The instructions are restarted by doing three jumps to the address in PC-4 and having shifting of the PC chain enabled. This means that the addresses will come out of the end of the chain and be reloaded at the front in the desired order.

The first of the three jumps should be a jpcrs instruction. It will cause PSWother to be copied to PSWcurrent with the interrupts turned on and the state returned to user space. The machine state changes after the ALU cycle of the first jump. The last two instructions of the return jump sequence should be jpc instructions.

A problem arises because an exception could occur while restarting these instructions. The PC chain is now in a state that it is not possible to restart the sequence again using the standard sequence of first saving the PC chain. The start of an exception sequence should first check the e bit in the PSW to see whether it is cleared. The e bit will be set only when the PC chain is back in a normal state. If it is clear, then the state of the machine should not be resaved. The state to use for restart should still be available in the process descriptor for the process being restarted when the

Exception Handling
exception occurred. The sequence for interrupt handling is shown in Figure VI-1.

VI.2. Trap On Overflow

A trap on overflow (See Section 2.4.1) behaves exactly like an interrupt except that it is generated on-chip instead of externally. This interrupt can be masked by setting the V bit in the PSW.

When a trap on overflow occurs, the O bit is set in the PSW. The exception handling routine must check this bit to see if an overflow is the cause of the exception.

VI.3. Trap Instructions

Besides the Trap on Overflow, there is only one other type of trap available. It is an unconditional vectored trap to a system space routine in low order memory. After the ALU cycle of the trap instruction the processor goes into system state with the PC chain frozen. The instruction before the trap instruction will complete its WB cycle. The PSW is saved by copying PSWcurrent to PSWother as described in Section 2.4. PSWcurrent is loaded as if this were an interrupt.

Figure VI-1: Interrupt Sequence

Execution begins at label lret
Before interrupts can be turned on again, some processor state must be saved. The return PCs are currently in the PC chain. Three PCs must be read from the PC chain and the third one saved in the process descriptor. It is the instruction that is in the RF cycle. The instruction corresponding to the PC in MEM completes so it need not be restarted. The PC in the ALU cycle should not be restarted because it is the trap instruction. PSWother must be saved so that the state of the prior process is preserved. If PSWother is not saved before interrupts are enabled, then another interrupt will smash the PSW of the process that executed the trap before it can be saved.

All trap instructions have an 8-bit vector number attached to them. This provides 256 legal trap addresses in system space. These addresses are 8 locations apart to provide enough space to store some jump instructions to the correct handler. If this is not enough vectors, one of the traps can take a register as an argument to determine the action required.

The return sequence must disable interrupts, restore the contents of PSWother and MD if they were saved and then disable PC shifting so that the return address can be shifted into the PC chain. Two more addresses must be shifted in as well so that the restart will look the same as an interrupt. This can be done by loading the addresses of two nop instructions into the PC chain ahead of the return address. Three jumps to the addresses in the PC chain are then executed using jpcrs and twojpcs. The first jump will copy the contents of PSWother into PSWcurrent and turn on PC shifting. The processor state changes after the ALU cycle of the jpcrs. The change of state also enables interrupts and puts the processor in user space.

If an interrupt occurs during the return sequence then the interrupt handler will look at the e bit in the PSW to determine whether the state should be saved.

The flow of code for taking a trap and returning is shown in Figure VI-2.
Figure VI-2: Trap Sequence

Exception Handling
Appendix VII
Assembler Macros and Directives

This appendix describes the macros and directives used by the MIPS-X assembler. Also provided is a full grammar of the assembler for those that need more detail.

VII.1. Macros

Several macros are provided to ease the process of writing assembly code. These allow low level details to be hidden, and ease the generation of code for both compilers and assembly language programmers.

VII.1.1. Branches

bgt, ble  
The assembler synthesizes these instructions by reversing the operands and using a blt or a bge instruction.

VII.1.2. Shifts

lsr, lsl  
These instructions are synthesized from the sh instruction. For example:

\[ \text{lsr } r1, r2, #4 \]

shifts rl four bits right and puts the result in r2.

VII.1.3. Procedure Call and Return

pjsr subroutine,#exp1,reg2  
A simple procedure call. The stack pointer is decremented by exp1. The return address is stored on the stack. On return, the stack pointer is restored. Reg2 is used as a temporary. No registers are saved.

ipjsr reg 1 ,#exp 1 ,reg 2  
A call to a subroutine determined at run time. The particular subroutine address must be in a register (regl) or be addressable off a register (exp2 + regl). The stack pointer and the return address handling is identical to pjsr. Reg2 is used as a temporary.

ret  
Jump to the return address stored by a pjsr or ipjsr macro.

VII.2. Directives

.text  
Signals the beginning or resumption of the text segment. This allows code to be grouped into one area. Labels in the text segment have word values.

.data  
Signals the beginning or resumption of the data segment. Labels in the data segment have byte values. Ordering within the data segment is not changed.

.end  
Signals the end of the module.

.eop  
Signals the end of a procedure. No branches are allowed to cross procedure boundaries. This directive was added to reduce the memory requirements of the assembler. Reorganization can be done by procedure instead of by module.

.ascii “xxx”  
Allows a string literal to be put in the data segment.

.word exp  
Initializes a word of memory.

*Provided by Scott McFarling
.float number
id = exp

Initializes a floating point literal.

Sets an assembly-time constant. This allows a code generator to emit co& before the value of certain offsets and literals are known. The assembler will resolve expressions using this identifier for aliasing calculations etc.

def id = exp

Sets a link-time constant. The identifier will be global.

.noreorg

Allows reorganization to be turned off in local areas.

.reorg

Turns reorganization back on.

.comm id, n

Defines a labeled common area of n words. Common area names are always global.

glob id

Makes an identifier global or accessible outside the module. The .globl statement must appear before the id is otherwise used. All procedure entry points should be made global, otherwise the code may be removed as dead.

.lit r1, r2,...

.lif r5, r10,...

Give a list of registers that are live for the following branches. .lit is for registers live if the branch is taken and .lif is for registers live if the branch is not taken. Liveness information is used for interblock reorganization and branch scheduling.

VII.3. Example

;program 1+1 = 2?
.data
.label1:
.word 1
.text
.globl -main
.globl -main:
   id label1[r0], r1
   addi r1, #1, r1
   addi r0, #2, r2
   bne r0, r2, error
   ret
error:
   trap 1
   ret
.end

VII.4. Grammar

file
   | file line
line
   | label
   | binALUState
   | monALUState
   | specState
   | nopState
   | addiState
   | jmpciState
   | shiftState
   | loadState
   | storeState
   | branchState
   | copState
   | miscState
   | directState
| macroState       |
| ID : { ID must be in column 1 } |
| binALUState : binALUOp reg,reg,reg |
| binALUOp : ADD |
| SUB |
| AND |
| OR |
| XOR |
| ROTLB |
| ROTLBC |
| MSTEP |
| DSTEP |
| SUBNC |
| BIC |
| monALUState : monOp reg,reg |
| monOp : NOT |
| MOV |
| specState : MOVTS reg,specialReg |
| specialReg : MD |
| PCM4 |
| PCM1 |
| nopState : NOP |
| addiState : ADDI reg,#exp,reg |
| jspcistate : JSPCI reg,#exp,reg |
| shiftState : ASR reg,reg,#exp |
| SH reg,reg,reg,#exp |
| LSR reg,reg,#exp |
| LSL reg,reg,#exp |
| loadState : LD exp[reg],reg |
| LD #exp,reg |
| { adds constant to literal pool and loads it } |
| LDT exp[reg],reg |
| LDF exp[reg],freg |
| storeState : ST exp[reg],reg |
| STI exp[reg],reg |
| STF exp[reg],freg |
| branchState : branchOp reg,reg,ID |
| branchOp : BEQ |
| BNE |
| BGE |
| BGT |
| BHI |
| BHS |
| BLE |
| BLO |
| BLS |
| BLT |
| branchSqOp : BEQSQ |
| BNEQ |
| BGEQ |
| BGTQ |
| BHIQ |
| BHSQ |
| BLEQ |
| BLOQ |
| BLSQ |
| BLTQ |
| copState : MOVTOC exp,reg |

Assembler Macros and Directives
Assembler Macros and Directives

| MOVFRC  exp, reg  |
| ALUC  exp  |
| floatBinOp  freg, freg  |
| floatMonOp  freg, freg  |
| MOVIF  reg, freg  |
| MOVFI  freg, reg  |

**floatBinOp**:
- FADD
- FSUB
- FMUL
- FDIV
- IMUL
- IDIV
- MOD

**floatMonOp**:
- CVTIF
- CVTFI

**miscState**:
- TRAP exp
- JPC
- JPCRS

**directState**:
- TEXT
- DATA
- END
- EOP
- ASCII STRING { string: ".*" }
- WORD exp
- FLOAT FLOATCONSTANT
- ID = exp
- DEF ID = exp
- REORGON
- NOREORG
- COMM ID, INT
- GLOBL ID
- LIT liveList
- LIF liveList

**liveList**:
- reg
- liveList, reg

**macroState**:
- PJSR ID, #exp, reg
- IPJSR reg, #exp, reg
- IPJSR exp, reg, #exp, reg
- RET

**exp**:
- exp addOp term
  - factor
  - term

**addOp**:
- +

**term**:
- term multOp factor
  - factor

**multOp**:
- *

**factor**:
- { exp }
  - ID
  - INT
  - HEXINT { like C: 0x12fc }

**reg**:
- REG { r0..r31 }

**freg**:
- FREG { f0..f15 }

**Notes:**
1) only labels and directives may start in column 1
2) Keywords are shown in upper case just to make them stand out. In reality, they MUST be lower case.
3) directives begin with a ' '.
References

On Holy Wars and a Plea for Peace.

Summary of MIPS Instructions.
Technical Note 83-237, Stanford University, November, 1983.

A Fast Mutual Exclusion Algorithm.