MIPS-X INSTRUCTION SET and PROGRAMMER'S MANUAL

PAUL CHOW

Technical Report No. CSL-86-289

MAY 1988

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 processsor, RISC, processor architecture, streamlined instruction set.

Table of Contents

List of Figures

List of Tables

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 (ie., only the top 30 bits are sent to the memory system). The numbering of words in memory is shown in Figure 2-l. 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 [I].*

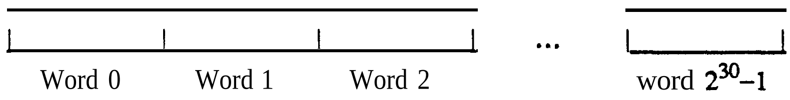

Figure 2-1: Word Numbering in Memory

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.

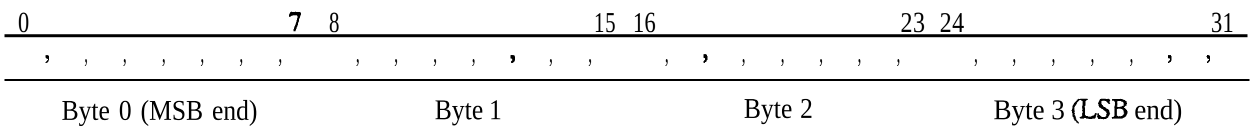

Figure 2-2: Bit and Byte Numbering in a Word

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 (*jpcrs*) causes *PSWother* to be copied into *PSWcurrent.* After the ALU cycle of the *jpcrs* 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.

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																															31
U	u	O	o															E	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 ***PSWcurrent (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 ovefflow bit (0) is set in ***PSWcurrent***.

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.

Instruction Timing

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 S-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* (ϕ_1) and *phase 2* (ϕ_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-l.

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

Table 3-1: MIPS-X Pipeline Stages

1.	IF	RF	ALU	MEM	WB				
2.		IF	RF	ALU	MEM	WB			
3.			IF	RF	ALU	MEM	WB		
4.				IF	RF	ALU	MEM	WB	
5.					IF	RF	ALU	MEM	WB

Figure 3-1: Pipeline Sequence

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 &lay 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 &lay 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 *1* 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 &lay 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 *1 - 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 implementions 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 *1* 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 Pair (Inst 1 - Inst 2)	Delay Slots with Bypassing Only to Source 1	Delay Slots with Src1/Src2 Bypassing	Comment
Load - Compute	1	1	
Load - Address	1	1	Loaded value used as address
Load - Data	2	1	Loaded value used for store data
Load - Branch	1	1	
Compute - Compute	0	0	
Compute - Address	0	0	Computed value used as address
Compute - Data	2	0	Compute result used for store data
Compute - Branch	0	0	

Table 3-2: Delay Slots for MIPS-X *Instruction* Pairs

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-l. 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_{-1}	This is the clock cycle before the IF cycle of the instruction being considered.
ϕ_1	Phase 1 of the clock cycle.
ϕ_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 Srcl 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.

rResult	The *result* register.
PC_{source}	The **PC** source to be used for this instruction. It will be one of: the displacement adder, the trap vector, the **incrementer,** the ALU or from the PC chain.
PCinc	The value from the PC incrementer.
PC-4	The last value in the PC chain.
Reg<n>, Reg<n..m>	Bit n or Bits n to *m* of register *Reg.*
Reg<< n	*Reg* is shifted left ***n*** **bits.**
Bypass source	Either ***rResult*** or ***MDRin***
Icache	The **onchip** instruction cache.
RFS	Reserved for Stanford.

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.

IF_{-1}	ϕ_1	RFS
	ϕ_2	PC bus ⇐ PC,,, Precharge tag comparators, valid bit store
IF	ϕ_1	Do tag compare Valid bit store access **Icache** address decoder ⇐ **PC<26..31>** Detect **Icache** hit Precharge **Icache** Do **incrementer** (calculate next sequential instruction address)
	ϕ_2	Do **Icache** access IR ⇐ **Icache**
RF	ϕ_1	Do bypass comparisons
	ϕ_2	**aluSrc1** ⇐ **rSrc1** or **aluSrc1** ⇐ Bypass source aluSrc2 c== **rSrc2** or aluSrc2 ⇐ Bypass source or aluSrc2 ⇐ Offset value Displacement adder latch ⇐ Displacement value MDRout ⇐ rSrc2 or MDRout ⇐ Bypass source
ALU	ϕ_1	Do ALU, do displacement adder (for branch and jump targets) **Precharge** Result bus
	ϕ_2	Result bus ⇐ ALU **rResult** ⇐ Result bus Memory address pads ⇐ Result bus (There may be a latch here)
MEM	ϕ_1	RFS
	ϕ_2	MDRin ⇐ **rResult** or MDRin ⇐ Memory data pads or Memory data pads ⇐ MDRout
WB	ϕ_1	**rDest** ⇐ MDRin
	ϕ_2	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.RFϕ_2, MDRout.ALUϕ_1, MDRout.ALUϕ_2 and MDRout.MEMϕ_1.

IF_{-1}	ϕ_1	RFS
	ϕ_2	PC bus $\Leftarrow$ PC,,,
		Precharge tag comparators, valid bit store
IF	ϕ_1	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)
	ϕ_2	Do Icache access
		IR $\Leftarrow$ Icache
RF	ϕ_1	Do bypass comparisons
	ϕ_2	aluSrc1 $\Leftarrow$ rSrc1
		or aluSrc1 $\Leftarrow$ Bypass source
		aluSrc2 $\Leftarrow$ Offset value
		MDRout.RFϕ_2 $\Leftarrow$ rSrc2 (For *stores)*
		or MDRout.RFϕ_2 $\Leftarrow$ Bypass source (For *stores)*
ALU	ϕ_1	Do ALU(add)
		Precharge Result bus
		MDRout.ALUϕ_1 $\Leftarrow$ MDRout.RFϕ_2 (For *stores)*
	ϕ_2	Result bus $\Leftarrow$ ALU
		rResult $\Leftarrow$ Result bus
		Memory address pads $\Leftarrow$ Result bus
		MDRout.ALUϕ_2 $\Leftarrow$ MDRout.ALUϕ_1 (For *stores)*
MEM	ϕ_1	MDRout.MEMϕ_1 $\Leftarrow$ MDRout.ALUϕ_2 (For *stores)*
	ϕ_2	MDRin $\Leftarrow$ Memory data pads (For *loads)*
		or Memory data pads $\Leftarrow$ MDRout.MEMϕ_1 (For *stores)*
WB	ϕ_1	rDest $\Leftarrow$ MDRin (For loads)
	ϕ_2	RFS

3.6.4. Branch Instructions

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

IF_{-1}	ϕ_1	RFS
	ϕ_2	PC bus $\Leftarrow$ PC,,, Precharge tag comparators, valid bit store
IF	ϕ_1	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)
	ϕ_2	Do Icache access IR $\Leftarrow$ Icache
RF	ϕ_1	Do bypass comparisons
	42	aluSrc 1 $\Leftarrow$ rSrc 1 or aluSrc1 $\Leftarrow$ Bypass source aluSrc2 $\Leftarrow$ rSrc2 or aluSrc2 $\Leftarrow$ Bypass source Displacement adder $\Leftarrow$ Displacement value
ALU	ϕ_1	Do ALU(Src1 – Src2), do displacement adder (for branch target) Precharge Result bus Evaluate condition at the end of ϕ_1 before the rising edge of ϕ_2
	ϕ_2	PC bus $\Leftarrow$ Displacement adder (Branch taken) or PC bus $\Leftarrow$ Incrementer (Branch not taken) Tag compare latch $\Leftarrow$ PC bus rResult $\Leftarrow$ Result bus
MEM	ϕ_1	RFS
	ϕ_2	MDRin $\Leftarrow$ rResult
WB	ϕ_1	RFS
	ϕ_2	RFS

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.

IF_{-1}	ϕ_1	RFS
	ϕ_2	PC bus ⇐ PC,,,
		Precharge tag comparators, valid bit store
IF	ϕ_1	Do tag compare
		Valid bit store access
		Icache address decoder ⇐ PC<26..31>
		Detect Icache hit
		Precharge Icache
		Do incrementer (calculate next sequential instruction address)
	ϕ_2	Do Icache access
		IR ⇐ Icache
RF	ϕ_1	Do bypass comparisons
	ϕ_2	aluSrc1 ⇐ rSrc 1
		or aluSrc1 ⇐ Bypass source
		aluSrc2 ⇐ rSrc2
		or aluSrc2 ⇐ Bypass source
		or aluSrc2 ⇐ Immediate value (for Compute Immediate Instructions)
ALU	ϕ_1	Do ALU
		Precharge Result bus
	ϕ_2	Result bus ⇐ ALU
		rResult ⇐ Result bus
MEM	ϕ_1	RFS
	ϕ_2	MDRin ⇐ rResult
WB	ϕ_1	rDest ⇐ MDRin
	ϕ_2	RFS

3.651. Special Instructions

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

IF_{-1}	ϕ_1	RFS
	ϕ_2	PC bus $\Leftarrow$ PC,,,
		Precharge tag comparators, valid bit store
IF	ϕ_1	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)
	ϕ_2	Do **Icache** access
		IR $\Leftarrow$ **Icache**
RF	ϕ_1	Do bypass comparisons
	ϕ_2	**aluSrc** 1 $\Leftarrow$ **rSrc** 1 (For ***movtos***)
		or **aluSrc1** $\Leftarrow$ Bypass source (For ***movtos***)
ALU	ϕ_1	Do **ALU(pass** Srcl)
		Precharge Result bus
	ϕ_2	Result bus $\Leftarrow$ alu Srcl (For ***movtos***)
		or Result bus $\Leftarrow$ Special Register (For ***movfrs***)
		Special Register $\Leftarrow$ Result bus (For ***movtos***)
		rResult $\Leftarrow$ Result bus
MEM	ϕ_1	RFS
	ϕ_2	**MDRin** $\Leftarrow$ **rResult**
WB	ϕ_1	**rDest** $\Leftarrow$ MDRin (For ***movfrs***)
	ϕ_2	RFS

3.6.6. Jump Instructions

IF_{-1}	ϕ_1	RFS
	ϕ_2	PC bus ⇐ PC,,, Precharge tag comparators, valid bit store
IF	ϕ_1	Do tag compare Valid bit store access Icache address decoder ⇐ PC<26..3 1> Detect Icache hit Precharge Icache Do incrementer (calculate next sequential instruction address)
	ϕ_2	Do Icache access IR ⇐ Icache
RF	ϕ_1	Do bypass comparisons
	ϕ_2	aluSrc1 ⇐ rSrc 1 or aluSrc1 ⇐ Bypass source aIuSrc2 ⇐ Immediate value (For jspci)
ALU	ϕ_1	Do ALU(add) Precharge Result bus
	ϕ_2	Result bus ⇐ PCinc (For *jspci*) PC bus ⇐ ALU (For *jspci*) or PC bus ⇐ PC-4, shift PC chain (For *jpc* and *jpcrs*) or PC bus ⇐ Trap vector (For *trap)* PSWcurrent ⇐ PSWother (For *jpcrs*) rResult ⇐ Result bus
MEM	ϕ_1	RFS
	ϕ_2	MDRin ⇐ rResult
WB	ϕ_1	rDes t ⇐ MDRin (For *jspci)*
	ϕ_2	RFS

3.6.7. Multiply Step - *mstep*

The *MD* register is implemented as a series of ϕ_2-ϕ_1 registers. They are called **MDresult.ϕ_2**, **MDresult.ϕ_1**, **MDmdrin.ϕ_2**, and **MDwb.ϕ_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.ϕ_2**. This chain of registers is necessary for restarting the sequence after an exception. **MDwb.ϕ_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.ϕ_1** are available in **MDresult.ϕ_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.

IF_{-1}	ϕ_1	RFS
	ϕ_2	PC bus $\Leftarrow$ **PC_{source}**
		Precharge tag comparators, valid bit store
IF	ϕ_1	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)
	ϕ_2	Do **Icache** access
		IR $\Leftarrow$ **Icache**
RF	ϕ_1	Do bypass comparisons
	ϕ_2	**aluSrc1** $\Leftarrow$ **rSrc1<<** 1
		or **aluSrc1** $\Leftarrow$ Bypass **source**<< 1
		aluSrc2 $\Leftarrow$ **rSrc2**
ALU	ϕ_1	Do **ALU(add)**
		Latch **aluSrc** 1
		Precharge Result bus
	ϕ_2	Result bus $\Leftarrow$ ALU (MSB (**MDresult.ϕ_1**) is 1)
		or Result bus $\Leftarrow$ **aluSrcl** (MSB (**MDresult.ϕ_1**) is 0)
		rResult $\Leftarrow$ Result bus
		MDresult.ϕ_2c==**MDresult.ϕ_1<<**1
MEM	ϕ_1	**MDresult.ϕ_1** $\Leftarrow$ **MDresult.ϕ_2**
	ϕ_2	MDRin $\Leftarrow$ **rResult**
		MDmdrin.ϕ_2 c= **MDresult.ϕ_1**
WB	ϕ_1	**rDest** $\Leftarrow$ MDRin
		MDwb.ϕ_1 $\Leftarrow$ **MDmdrin.ϕ_2**
	ϕ_2	RFS

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

IF_{-1}	ϕ_1	RFS
	ϕ_2	PC bus ⇐ PC_{source} Precharge tag comparators, valid bit store
IF	ϕ_1	Do tag compare Valid bit store access Icache address decoder ⇐ PC<26..31> Detect Icache hit Precharge Icache Do incrementer (calculate next sequential instruction address)
	ϕ_2	Do Icache access IR ⇐ Icache
RF	ϕ_1	Do bypass comparisons
	ϕ_2	aluSrc1 ⇐ rSrc1<< 1 + MSB(MDresult.ϕ_1) or aluSrc1 ⇐ Bypass source<< 1 + MSB(MDresult.ϕ_1) aluSrc2 ⇐ rSrc2
ALU	ϕ_1	Do ALU(sub) Precharge Result bus
	ϕ_2	Result bus ⇐ ALU (MSB (ALU result) is 0) or Result bus ⇐ aluSrcl (MSB (ALU result) is 1) rResult ⇐ Result bus MDresult.ϕ_2 ⇐ MDresult.ϕ_1<< 1 + Complement of MSB(ALU result)
MEM	ϕ_1	MDresult.ϕ_1 ⇐ MDresult.ϕ_2
	ϕ_2	MDRin ⇐ rResult MDmdrin.ϕ_2 ⇐ MDresult.ϕ_1
WB	ϕ_1	rDest ⇐ MDRin MDwb.ϕ_1 ⇐ MDmdrin.ϕ_2
	ϕ_2	RFS

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 II 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 &tail in the individual instruction descriptions.

4.2.1. ld - Load

TY	OP	Srcl	Dest	Offset(17)
1 0	0 0 0			

Assembler

ld Offset[rSrcl],rDest

Operation

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

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

TY	OP	Src1	Src2	Offset(17)
1 0	0 1 0			

Assembler

st Offset[rSrc1],rSrc2

Operation

Memory[SignExtend(Offset) + Reg(Src1)] $\Leftarrow$ Reg(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 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

TY	OP	Srcl	Dest	Offset(17)
1 0	1 0 0			

Assembler

ldf Offset [rSrc 1] ,fDes t

Operation

FReg(Dest) ⇐ Memory[SignExtend(Offset) + Reg(Src1)]

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 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

TY	OP	Srcl	Src2	Offset(17)
1 0	1 1 0			

Assembler

stf Offset[rSrc1],fSrc2

Operation

Memory[SignExtend(Offset) + Reg(Src1)] ⇐ 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. ldt - Load Through

TY	OP	Src1	Dest	Offset(17)
1 0	0 0 1			

Assembler

ldt Offset[rSrc1],rDest

Operation

Reg(Dest) $\Leftarrow$ 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 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 *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

TY	OP	Srcl	Src2	Offset(17)
1	010 1 1			

Assembler

stt Offset[rSrc1],rSrc2

Operation

Memory[SignExtend(Offset) + Reg(Src1)] ⇐ Reg(Src2)

Description

This instruction is the same as ***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 Srcl field to compute a memory address. The contents of Reg(Src2) are stored at **that** memory location

4.2.7. movfrc - Move From Coprocessor

TY	OP	Src1(r0)	Dest	COP#	Func	CS1	CS2/CD
1 0	1 0 1	0 0 0 0 0					
				cop1			

Assembler

movfrc CopI,rDest

Operation

MAR ⇐ SignExtend(CopI) + Reg(Src 1)
Reg(Dest) ⇐ 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 CopI field is decoded by the coprocessor-s to fmd 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

TY	OP	Src1(r0)	Src2	COP#	Func	CS1	CS2/CD
Jl	011 1 1	0 0 0 0 0					
				cop1			

Assembler

movtoc CopI,rSrc2

Operation

MAR ⇐ SignExtend(CopI) + Reg(Src1)
MDR ⇐ 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 Srcl field. The Srcl field should be Register 0 if the CopI 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

```
  TY    OP       Srcl (r0)               COP#      Func         CS1     CS2/CD
|1  0|1  0  1|0  0  0  0  0|0  0  0  0  0| ' ' | ' ' ' ' ' | ' ' ' | ' ' ' |
                                         |            cop1                 |
```

Assembler

aluc Cop1

Operation

MAR ⇐ SignExtend(CopI) + 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:

movfrc CopI,r0 .

The Cop1 field is sign extended and added to the contents of the register specified by the Srcl 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 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

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 2'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.

Branch	Description	Expression	Branch To Use If Synthesized
beq	Branch if equal	Z	
bge	Branch if greater than or equal	$\overline{N \oplus V}$	
bgt	Branch if greater than	$\overline{(N \oplus V) + Z}$	blt (rev ops)
bhi	Branch if higher	$\overline{\overline{C} + Z}$	blo (rev ops)
bhs	Branch if higher or same	C	
ble	Branch if less than or equal	$(N \oplus V) + Z$	bge (rev ops)
blo	Branch if lower than	$\overline{C}$	
blos	Branch if lower or same	$\overline{C} + Z$	bhs (rev ops)
blt	Branch if less than	$N \oplus V$	
bne	Branch if not equal	$\overline{Z}$	
bpl	Branch if plus	$\overline{N}$	bge (cmp to Src2=0)
bmi	Branch if minus	N	blt (cmp to Src2=0)
bra	Branch always		beq r0,r0

Table 4-1: Branch Instructions

4.3.1. beq - Branch If Equal

TY	Cond	Srcl	Src2	SQ	Disp(16)
0 0	0 0 1			s	

s = 1 $\Rightarrow$ Squash if don't go

s = 0 $\Rightarrow$ No squashing

Assembler

beq	rSrc1,rSrc2,Label	; No squashing
beqsq	rSrc 1,rSrc2,Label	; Squash if don't go

Operation

If [Reg(Src 1) – Reg(Src2)] $\Rightarrow$ Z
then
PCnext $\Leftarrow$ PCcurrent + SignExtend@isp)

Description

If Reg(Src1) equals 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) does not equal Reg(Src2), then the delay slot instructions are executed for *beq and* squashed for *beqsq.*

4.3.2. bge - Branch If Greater than or Equal

TY	Cond	Srcl	Src2	SQ	Disp(16)
0 0	1 1 1			s	

s = 1 ⇒ Squash if don't go

s = 0 ⇒ No squashing

Assembler

```
bge    rSrc1,rSrc2,Label      ; No squashing
bgesq  rSrc 1,rSrc2,Label     ; Squash if don't go
```

Operation

If $[\mathrm{Reg(Src1)} - \mathrm{Reg(Src2)}] \Rightarrow \overline{N \oplus V}$
then
PCnext ⇐ PCcurrent + SignExtend(Disp)

Description

This is a signed compare.

If Reg(Src1) 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(Src1) 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

TY	Cond	Srcl	Src2	SQ	Disp(16)
0 0	0 1 0			s	

s = 1 $\Rightarrow$ Squash if don’t go

s = 0 $\Rightarrow$ No squashing

Assembler

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

Operation

If [Reg(Src1) − Reg(Src2)] $\Rightarrow$ C
then
PCnext $\Leftarrow$ PCcurrent + SignExtend@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

TY	Cond	Srcl	Src2	SQ	Disp(16)
0 0	1 1 0			s	

s = 1 ⇒ Squash if don't go

s = 0 ⇒ No squashing

Assembler

```
blo    rSrc1,rSrc2,Label     ; No squashing
blosq  rSrc1,rSrc2,Label     ; Squash if don't go
```

Operation

If [Reg(Src1) – Reg(Src2)] ⇒ $\overline{C}$
  then
    PCnext ⇐ PCcurrent + SignExtend@isp)

Description

This is an unsigned compare.

If Reg(Src1) is lower than 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) is higher than or equal to Reg(Src2) or if there was a carry generated, then the delay slot instructions are executed for *blo* and squashed for *blosq*.

4.3.5. blt - Branch If Less Than

TY	Cond	Srcl	Src2	SQ	Disp(16)
0 0	0 1 1			s	

s = 1 ⇒ Squash if don't go

s = 0 ⇒ No squashing

Assembler

```
blt     rSrc1,rSrc2,Label     ; No squashing
bltsq   rSrc 1,rSrc2,Label    ; Squash if don't go
```

Operation

If [Reg(Src 1) – Reg(Src2)] ⇒ N ⊕ V
then
PCnext ⇐ PCcurrent + SignExtend@isp)

Description

This is a signed compare.

If Reg(Src1) is less than 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) is greater than or equal to Reg(Src2), then the delay slot instructions are executed for ***blt*** *and* squashed for *bltsq*.

4.3.6. bne - Branch If Not Equal

TY	Cond	Srcl	Src2	SQ	Disp(16)
0 0	1 0 1			s	

s = 1 ⇒ Squash if don't go

s = 0 ⇒ No squashing

Assembler

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

Operation

If [Reg(Src1) – Reg(Src2)] ⇒ $\overline{Z}$
then
PCnext ⇐ PCcurrent + 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

TY	OP	Srcl	Src2	Dest	Comp Func(12)
0 1	1 0	0			0 0 0 0 0 0 0 1 1 0 0 1

Assembler

add rSrc 1,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

TY	OP	Srcl	Src2	Dest	CompFunc(12)
0 1	0 0 0				0 0 0 1 0 1 1 0 0 1 1 0

Assembler

dstep rSrc1,rSrc2,rDest

Operation

Srcl should be the same as Dest.

ALUsrc1 $\Leftarrow$ Reg(Src1)<< 1 + MSB(Reg(MD))
ALUsrc2 $\Leftarrow$ Reg(Src2)
ALUoutput $\Leftarrow$ ALUsrc1 − ALUsrc2

If MSB(ALUoutput) is 1
  then
    Reg(Dest) $\Leftarrow$ ALUsrc1
    Reg(MD) $\Leftarrow$ Reg(MD)<< 1
  else
    Reg(Dest) $\Leftarrow$ ALUoutput
    Reg(MD) $\Leftarrow$ Reg(MD)<< 1 + 1

Description

This is one step of a l-bit restoring division algorithm. The division scheme is described in Appendix IV.

4.4.3. mstart - Multiply Startup

TY	OP	Srcl	Src2	Dest	Comp Func(12)
0 1	0 0 0	0 0 0 0 0	' ' ' '	' ' ' '	0 0 0 0 1 1 1 0 0 1 1 0

Assembler

ms tart rSrc2,rDest

Operation

If MSB(Multiplier loaded in Reg(MD)) is 1
then
Reg(Dest) $\Leftarrow$ 0 - Reg(Src2)
Reg(MD) $\Leftarrow$ Reg(MD)<< 1
else
Reg(Dest) $\Leftarrow$ 0
Reg(MD) $\Leftarrow$ Reg(MD)<< 1

Description

This is the first step of a l-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

TY	OP	Src1	Src2	Dest	Comp Func(12)
0 1	0 0 0	' ' ' '	' ' ' '	' ' ' '	1 0 0 0 0 1 0 0 1 1 0 0 1 ~

Assembler

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 l-bit shift and add multiplication algorithm. The multiplication scheme is described in Appendix IV.

4.4.5. sub - Subtract

TY	OP	Src1	Src2	Dest	Comp Func(12)
0 1	1 0 0				0 0 0 0 0 1 1 0 0 1 1 0

Assembler

sub rSrc1,rSrc2,rDest

Operation

Reg(Dest) $\Leftarrow$ 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

TY	OP	Src1	Src2	Dest	CompFunc(12)
0 1	1 0 0	“ “ I	“ “ I	’ ’ ’ ’	0 0 0 0 0 0 1 0 0 1 1 0

Assembler

subnc rSrc 1,rSrc2,rDest

Operation

Reg(Dest) $\Leftarrow$ Reg(Src1) + $\overline{\text{Reg(Src2)}}$

Description

The l’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        ;subtract high words
        bhssq   rAlo,rBlo,l1          ;check if subtract of low
                                      ;words generates a carry
                                      ;branch if carry set
        addi    rChi,#1,rChi          ;add 1 to high word if carry
        nop
11:     sub     rAlo,rBlo,Clo         ;subtract low words
```

4.4.7. and - Logical And

TY OP Srcl Src2 Dest Comp Func(12)

0 1 | 1 0 0 | “ “ I ’ ’ ’ ’ | ’ “ ‘ 1 0 0 0 0 0 0 1 0 0 0 1 1 ~

Assembler

and rSrc 1,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

TY	OP	Src1	src2	Dest	Comp Func(12)
0 1	1 0 0				0 0 0 0 0 0 0 0 1 0 1 1

Assembler

bic rSrc1,rSrc2,rDest

Operation

Reg(Dest) $\Leftarrow$ $\overline{\text{Reg(Src1)}}$ ***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

TY	OP	Src1		Dest	Comp Func(12)
0 1	1 0 0	' ' ' '	0 0 0 0 0	' ' ' '	0 0 0 0 0 0 0 0 1 1 1 1

Assembler

not rSrc1,rDest

Operation

Reg(Dest) $\Leftarrow$ $\overline{\text{Reg(Src1)}}$

Description

The *ones complement* of Source 1 is placed in Destination.

4.4.10. or - Logical Or

TY	OP	Src1	Src2	Dest	Comp Func(12)
0 1	1 0 0	, , , ,	, , , ,	, , , ,	0 0 0 0 0 0 1 1 1 0 1 1

Assembler

or rSrc 1,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

TY	OP	Srcl	Src2	Dest	Comp Func(12)
JO	1 1 1 0 0	' ' ' '	' ' ' '	' ' ' '	0 0 0 0 0 0 0 1 1 0 1 1

Assembler

xor rSrc1,rSrc2,rDest

Operation

Reg(Dest) $\Leftarrow$ 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

TY	OP	Srcl		Dest	Comp Func(12)
0 1	1 0 0	' ' ' '	0 0 0 0 0	' ' ' '	0 0 0 0 0 0 0 1 1 0 0 1

Assembler

mov **rSrc** 1 ,rDes t

Operation

Reg(Dest) ⇐ Reg(Srcl)

Description

This is a register **to** register move. It is implemented as

add **rSrc** l,rO,rDest .

This mnemonic is provided for convenience and clarity.

4.4.13. asr - Arithmetic Shift Right

TY	OP	Src1		Dest	Comp Func(12)	
0 1	0 0 1	' ' ' '	0 0 0 0 0	' ' ' '	0 0 0 1 0	b b b d d d d

Assembler

asr rSrcl,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:

ef	dddd
00	0001
01	0010
10	0100
11	1000

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

TY	OP	Src1	Src2	Dest	Comp Func(12)
0 1	0 0 1	, , , ,	, , , ,	, , , ,	0 0 0 0 1 1 0 0 0 0 0 0

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 rSrc 1,rSrc2,rDest

Reg(Dest) = 01CD23AB#16

4.4.15. rotlcb - Rotate Left Complemented by Bytes

TY	OP	Src1	Src2	Dest	Comp Func(12)	
0	1100	1			10000	10000000~

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

TY	OP	Src1	Src2	Dest	Comp Func(12)
0 1	0 0 1	' ' ' '	' ' ' '	' ' ' '	0 0 1 0 0 | b b b d d d d

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,r2,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
```

Shift Amount	bbbdddd	Result
0	Not Valid	
1	1111000	44D5E6F7
4	1110001	089ABCDE
16	1000001	567089AB
28	0010001	23456708
31	0000010	2468ACE1
32	0000001	12345670

4.4.17. nop - No Operation

TY	OP																										CompFunc(12)				
JO	111	0	010	0	0	0	010	0	0	0	010	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1			

Assembler

nop

Operation

Reg(0) ⇐ Reg(0) + Reg(0)

Description

This instruction does do not 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

TY	OP	Src1	Dest	Immed(17)
1 1	1 0 0			

Assembler

addi Src1,#Immed,Dest

Operation

Reg(Dest) $\Leftarrow$ 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

TY	OP					CompFunc(12)	
1 1	1 0 1	0 0 0 0 0	0 0 0 0 0	1 0 0 0 0 0	1 0 0 0 0 0 0 0 0 0 1 1		

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 *jpcrs* which also causes some state bits to change.

4.5.3. jpcrs - Jump PC and Restore State

TY		OP																								Comp Func(12)			
1	1	1	1	110	6	0	0	010	0	0	0	0	0	0	0	0	010	0	0	0	0	0	0	0	0	0	1	1	

Assembler

jpcrs

Operation

PC shifting enabled
PSWcurrent ⇐ PSWother
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 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

TY	OP	Src1	Dest	Immed(17)
1 1	0 0 0			

Assembler

jspci **rSrc1,#Immed,rDest**

Operation

PC ⇐ **Reg(Src1)** + **SignExtend(Immed)**
Reg(Dest) ⇐ **PCcurrent** + 1

Description

This instruction **has two delay slots. The address of the instruction after the two &lay 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

```
 TY    OP              Dest                      CompFunc(12)
|1 1 |0 1 1 |0 0 0 0 0 | ' ' ' ' |0 0 0 0 0 |0 0 0 0 0 0 0 0 0 | ' ' |
                                                              Spec
```

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.

SpecialReg	Spec
psw	001
md	010
pcm4	100

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

```
 TY    OP         Srcl                                          CompFunc(12)
|1 1|0 1 0| ' ' ' ' |0 0 0 0 010 0 0 0 0|0 0 0 0 0 0 0 0 0| ' ' |
                                                                  Spec
```

Assembler

movtos rSrc1,SpecialReg

Operation

Reg(Spec) ⇐ Reg(Src 1)

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.

SpecialReg	Spec
psw	001
md	010
pcm1	100

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

```
 TY    OP                                               Vector(8)
|1  111  1010  0  0  0  0|0  0  0  0  0|0  0  0  0  0  0|  '  '  '  '  '  '  '  |0  1  1|
```

Assembler

trap Vector

Operation

Stop PC shifting
PC ⇐ Vector << 3
PSWother ⇐ PSWcurrent

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, ***jspci, jpc,*** or *jpcrs* instruction. See Appendix VI for more details.

The assembler should convert ***Vector*** to its one's complement form before generating the machine instruction. ie., the machine instruction contains the one's complement of the vector.

4.58. **hsc - Halt and Spontaneously Combust**

TY OP

| 1 1 | 0 0 111 1 1 1 **110 0 0 0** 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |

Assembler

hsc

Operation

Reg(31) $\Leftarrow$ 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 id 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#16
```

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,r2,#24    ; r2 <-- vstw
        rotlcb  r2,r1,r2        ; r2 <-- stwv
;
; 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,r3,#24    ; r3 <-- u
```

Thu May 22 1986 09:08:40 -1- scltags.xs

```
;
; 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 the entry in the cache that called
;       this routine but to make it be general it will have to since you
;       don't want to have to figure out where you came from.
;       This is called from a trap so 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 ".makenop n" means that "n" nop instructions should be inserted.
;
; This module should be loaded starting at address 0x1800.
;
.text
.noreorg
cltags:
l0x1800:
        jspcl   r0,#0x1810,r0
        jspcl   r0,#0x1820,r0
        jspcl   r0,#0x1830,r0
        .makenop 13
l0x1810:
        jspcl   r0,#0x1840,r0
        .makenop 15
l0x1820:
        jspcl   r0,#0x1850,r0
        .makenop 15
l0x1830:
        jspcl   r0,#0x1860,r0
        .makenop 15
l0x1840:
        jspcl   r0,#0x1870,r0
        .makenop 15
l0x1850:
        jspcl   r0,#0x1890,r0
        .makenop 15
l0x1860:
        jspcl   r0,#0x18a0,r0
        .makenop 15
l0x1870:
        jspcl   r0,#0x18b0,r0
        .makenop 15
l0x1880:
        jspcl   r0,#0x1940,r0
        .makenop 15
l0x1890:
        jspcl   r0,#0x18c0,r0
        .makenop 15
l0x18a0:
        jspcl   r0,#0x18d0,r0
        .makenop 15
l0x18b0:
        jspcl   r0,#0x18e0,r0
        .makenop 15
l0x18c0:
        jspcl   r0,#0x18f0,r0
        .makenop 15
l0x18d0:
        jspcl   r0,#0x1880,r0
```

Thu May 22 1986 09:08:40 -2- scltags.xs

```
        .makenop 15
l0x18e0:
        jspcl   r0,#0x1920,r0
        .makenop 15
l0x18f0:
        jspcl   r0,#0x1930,r0
        .makenop 15
l0x1900:
        jspcl   r0,#0x19c0,r0
        .makenop 15
l0x1910:
        jspcl   r0,#0x19d0,r0
        .makenop 15
l0x1920:
        jspcl   r0,#0x1950,r0
        .makenop 15
l0x1930:
        jspcl   r0,#0x1960,r0
        .makenop 15
l0x1940:
        jspcl   r0,#0x1970,r0
        .makenop 15
l0x1950:
        jspcl   r0,#0x1900,r0
        .makenop 15
l0x1960:
        jspcl   r0,#0x1910,r0
        .makenop 15
l0x1970:
        jspcl   r0,#0x19b0,r0
        .makenop 15
l0x1980:
        ; start return
        movfrs  pcm4,r31                ; save this for restart
        .makenop 15
l0x1990:
        movtos  r0,pcm1                 ; prepare for return
        .makenop 15
l0x19a0:
        movtos  r0,pcm1
        movtos  r31,pcm1                ; restart instruction after trap
        jpcrs
        jpc
        jpc
        .makenop 11
0x19b0:
        jspcl   r0,#0x19e0,r0
        .makenop 15
l0x19c0:
        jspcl   r0,#0x19f0,r0
        .makenop 15
l0x19d0:
        jspcl   r0,#0x1980,r0
        .makenop 15
l0x19e0:
        jspcl   r0,#0x1990,r0
        .makenop 15
l0x19f0:
        jspcl   r0,#0x19a0,r0
end
```

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 = $N \oplus V$

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

II.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	Srcl · Src2
	Bit 27	Srcl . Src2
	Bit 2%	Srcl . Src2
	Bit 29	Srcl · Src2
3its 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 **rotlb,** 0 if rotlcb
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

Instruction	TY	OF	Comments
ld	10	000	*
st	10	010	
ldf	10	100	*
stf	10	110	
ldt	10	001	*
stt	10	011	
movfrc	10	101	Src1=0, *
movtoc	10	111	Src1=0
aluc	10	101	Src1=0, Dest=O, *

Branch Instructions

Instruction	TY	COND
beq	00	001
bge	00	111
bhs	00	010
blo	00	110
blt	00	011
bne	00	101

Compute Instructions

Instruction	TY	OP	Comp Func	Comments
add	01	100	000000011001	
dstep	01	000	000101100110	
mstart	01	000	000011100110	Src1=0
mstep	01	000	000010011001	
sub	01	100	000001100110	
subnc	01	100	000000100110	
and	01	100	000000100011	
bic	01	100	000000001011	Src2=0
not	01	100	000000001111	Src2=0
or	01	100	000000111011	
xor	01	100	000000011011	
mov	01	100	000000011001	Src2=0
asr	01	001	00010bbbdddd	Src2=0, bbbdddd=rotate amount
rotlb	01	001	000011000000	
rotlcb	01	001	000010000000	
sh	01	001	00100bbbdddd	bbbdddd=rotate amount
nop	01	100	000000011001	Src1=0, Src2=0, Dest=0

Compute Immediate Instructions

Instruction	TY	OP	Comp Func	Comments
addi	11	100	Immed	* (Immed is a 17-bit
jspci	11	000	Immed	* signed constant)
jpc	11	101	000000000011	*
jpcrs	11	111	000000000011	
movfrs	11	011	000000000rrr	rrr = special register
movtos	11	010	000000000rrr	rrr = special register
trap	11	110	0vvvvvvvv011	Src1=0, vvvvvvvv=vector
unused	11	001		

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-l. The fields represent the following floating point number:

$$(-1)^{s} \times 2^{exp - 127} \times (1 + fraction).$$

This is an approximate IEEE floating point format.

s	exp (8 bits)	fraction (23 bits)

Figure III-l: 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 ***ldf*** and ***stf*** instructions defined in the instruction set. Moves between the floating point registers and the main processor are done using the *movif* and ***movfi*** instructions. These use the movtoc and ***movfrc*** formats defined in the instruction set. Note that only 4 of the 5 bits that specify a floating point register in the *ldf*, ***stf***, ***movif*** and ***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 ***aluc*** coprocessor instruction. The coprocessor number (***COP#***) is 0 for the floating point coprocessor. The ***Func*** field specifies the floating point operation to be performed.

111.5. Opcode Map of Floating Point Instructions

In the following table:

r1,r2 are cpu registers from r0..r31
f1,f2 are floating point registers from f0..f15
n is an integer expression

Instruction		TY	OP	Func	Operation	Comments
fadd	f1,f2	10	101	000000	f2 ⇐ fl + f2	Src1=0, Dest=0
fsub	f1,f2	10	101	000001	f2 ⇐ fl − f2	Src1=0, Dest=0
fmul	f1,f2	10	101	000010	f2 ⇐ fl × f2	Srcl-0, Dest=0
fdiv	f1,f2	10	101	000011	f2 ⇐ fl / f2	Src1=0, Dest=0
cvtif	f1,f2	10	101	000100	f2 ⇐ float(f1)	Src1=0, Dest=0 Convert int to float
cvtfi	f1,f2	10	101	000101	f2 ⇐ int(f1)	Srcl-0, Dest=0 Convert float to int
imul	f1,f2	10	101	000110	f2 ⇐ fl x f2	Src1=0, Dest-0 Integer multiplication
idiv	f1,f2	10	101	000111	f2 ⇐ fl / f2	Src1=0, Dest=O Integer division
mod	f1,f2	10	101	001000	f2 ⇐ fl mod f2	Src1=0, Dest=O Integer mod
movif	r1,f1	10	111	001001	fl ⇐ rl	Src1=0, CS1=0
movfi	f1,r1	10	101	001010	rl ⇐ fl	Srcl-0, CS2=0
ldf	n[r1],f1	10	100			See instruction page
stf	n[r1],f1	10	110			See instruction page

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 *mstep*. For

mstep rSrc 1,rSrc2,rDest

the operation is:

If the MSB of the MD register is 1
then
rDest $\Leftarrow$ 2 x rSrc1 + rSrc2
else
rDest $\Leftarrow$ 2 x rSrc1

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 *mstart* is provided for this purpose. For

mstart rSrc2,rDes t

the operation is

If the MSB of the MD register is 1
then
rDest ⇐ 0 − rSrc2
else
rDest ⇐ 0

Shift left MD

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

rMer is the multiplier,
rMand is the multiplicand,
rDest is the result register
rLink is the jump linkage register.

Then,

```
movtos   rMer,rMD                  ;Move the multiplier into MD
nop                                ;Needed for hardware timing reasons--see movtos
mstart   rMand,rDest               ;Do the first mstep.  Result goes into rDest
mstep    rDest,rMand,rDest         ;Repeat 31 times
jspci    rLink,#0,r0               ;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 *msteps* on average. It will actually do a full 32 *msteps* if it is necessary. In this case it is most likely that overflow will occur and this can be detected if the ***V*** bit in the PSW is clear so that a trap on overflow will occur. Assume that the registers *rMer, rMand* and *rDest* have been assigned and loaded as in the previous example. Two temporary registers, *rTemp1* and ***rTemp2 are also*** required

The number of cycles required, not including the instructions needed for the call sequence is shown in Table IV-l. 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 *msteps*, then this routine will be faster than just doing 32 ***msteps*** all the time.

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 (*rRem*) is initialized to 0. The divisor is loaded into another register called (*rDor*). The result of the division (quotient) will be in ***MD.*** For

ds tep rRem,rDor,rRem

the operation is:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                       ;
; 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             ; may need nop before this
        mstep   rDest,rMand,rDest
lmul8bit:
        mstep   rDest ,rMand,rDest
        mstep   rDest,rMand,rDest
        mstep   rDest,rMand,rDest
        mstep   rDest,rMand,rDest
        jspci   rLink,#0,r0
        mstep   rDest ,rMand,rDest
        mstep   rDest,rMand, rDest
lnot8:
        addi    rTemp2,#1,rTemp2
        beqsq   rTemp2,r0,lmul8bit      ; 8 bit negative
        mstart  rMand,rDest
        mstep   rDest,rMand,rDest
        movtos  rDest,md                ; do full 32 bits
        mstart  rMand,rDest             ; may need nop before this
        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-l: Signed Integer Multiplication

Number of ***msteps*** needed	8	32
Number of cycles with positive multiplier	13	42
Number of cycles with negative multiplier	15	42

Table IV-l: Number of Cycles Needed to do a Multiplication

```
Set ALUsrcl input to 2 x rRem + MSB(rMD)
Set ALUsrc2 input to rDor
ALUoutput ⇐ ALUsrc1 - ALUsrc2

If MSB(ALUoutput) is 1
   then
      rRem ⇐ ALUsrcl
      rMD ⇐ 2 × rMD
   else
      rRem ⇐ ALUoutput
      rMD ⇐ 2 × rMD + 1
```

At ***the*** end of 32 ***dsteps*** the quotient will be in the ***MD*** register, and the remainder ***is in*** ***rRem.***

A routine for doing division is shown in Figure IV-2. The dividend is passed ***in rDend*** and the divisor in ***rDor***. At the end, the quotient is in ***MD*** and ***rQuot*** and the remainder is in ***rRem.*** ***Note*** that ***rDend*** and ***rRem can*** be the same register, and ***rDor*** and ***rQuot*** 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.

Number of ***dsteps*** needed	8	32
Number of cycles needed	34	60

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

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                           ;
; DIV                                                                       ;
;       fast,  unchecked,  signed divide (should check for zero divide)     ;
;               rLink = link                                                ;
;               rDend,rRem = srcl                  (dividend)               ;
;               rDor = rQuot = src2/dest           (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
lcinitl:
        bgesq   rDor,r0,lcinit2         ; divisor > 0 ?
        addi    r0,#0xff,rTempl         ; 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,rTempl
lcinit2:
        bltsq   rTempl,rDend,ldivfull   ; do 8-bit check
        movtos  rDend,md                ; start 32-bit divide
        mdv     r0 ,rRem
        sh      r0,rDend,rDend,#8       ; shift up divisor to do 8 bits
        movtos  rDend,md                ; start 8-bit divide
        beq     r0,r0,ldivloop
        mov     r0,rRem
        addi    r0,#8,rTempl            ; loop counter
ldivfull:
        addi    r0,#32,rTempl           ; 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
        dstep   rRem,rDor,rRem
        addi    rTempl,#-8,rTempl       ; decrement loop counter
        dstep   rRem,rDor,rRem
        bnesq   rTempl,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:
        jspci   rLink,#O,rLink          ; return
        nop
        nop
```

Figure W-2: Signed Integer Division

Multiplication and Division

Appendix V
Multiprecision Arithmetic

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,l1            ;check to see if carry generated
                                        ;branch if carry set
        addi    rChi,#l,rChi            ;add 1 to high word if carry
        nop
11:     add     rAlo,rBlo,rClo          ;add low words


;Double precision subtraction

        subnc   rAhi,rBhi,rChi          ;subtract high words
        bhssq   rAlo,rBlo,l1            ;check if subtract of low
                                        ;words generates a carry
                                        ;branch if carry set
        addi    rChi,#l,rChi            ;add 1 to high word if carry
        nop
11:     sub     rAlo,rBlo,Clo           ;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-l and it shifts after each write to PC-l. 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 bejpc 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

```
lret:   inst    a                       ;Instructions a, b and c are restarted
        inst    b
        inst    c
        --- interrupt ---
        inst    d
        inst    e



inthlr: bra to save if e bit set        ;Start of interrupt handler
        Do necessary fixes              ;e bit clear so don't save PC chain
        bra nosave
save:   Save PSWother                   ;do save if interrupts to be enabled
        Save MD                         ;if necessary
        movfrs  pcm4,rA                 ;save PCs if necessary
        movfrs  pcm4,rB
        movfrs  pcm4,rC
nosave: Enable interrupts               ;if necessary and above saving done
            .
        Process interrupts
            .
        Disable interrupts
        Restore MD                      ;if necessary
        Restore PSWother                ;if necessary
        movtos  rA,pcml                 ;restore PCs
        movtos  rB,pcml
        movtos  rC,pcml
        jpcrs                           ;This changes the PSW as well
        jpc                             ;Doesn't touch PSW
        jpc
        execution begins at label lret
```

Figure VI-l: Interrupt Sequence

exception occurred. The sequence for interrupt handling is shown in Figure **VI-l.**

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.

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.

```
                 .
         trap     vecnum
lret:



vecnum:  movfrs  pcm4,r0                   ;instruction before trap
         movfrs  pcm4,r0                   ,trap instruction
         movfrs  pcm4,r31                  ;save this one to restart
         Save PSWother                     ;if necessary
         Save MD                           ;if necessary
         Enable interrupts                 ;if necessary and above saving done


         Process requested trap

             .
         Disable interrupts                ;movtos x,pswc where x has M bit set
         Restore MD                        ;if necessary
         Restore PSWother                  ;if necessary
         movtos  r0,pcm1                   ;assume a nop at 0
         movtos  r0,pcm1
         movtos  r31,pcm1                  ,*instruction after trap
         jpcrs
         jpc
         jpc
         execution begins at label lret
```

Figure VI-2: Trap Sequence

Appendix VII
Assembler Macros and Directives

This appendix[1] 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:

```
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 ,reg2

ipj sr exp2,reg 1 ,#expl,reg2 — 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.

[1] Provided by Scott McFarling

.float number	Initializes a floating point literal.
id = exp	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.
.reorgon	Turns reorganization back on.
.comm id,n	Defines a labeled common area of n words. Common area names are always global.
.globl 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
_main:
        ld      label1[r0],r1
        addi    r1,#1,r1
        addi    r0,#2,r2
        bne     r1,r2,error
        ret
error:
        trap    1
        ret
 .end
```

VII.4. Grammar

```
file            :
                | file line
line            : \n
                | COMMENT \n           { comment = ;.* }
                | statement COMMENT \n
                | statement \n
statement       : label
                | binALUState
                | monALUState
                | specState
                | nopState
                | addiState
                | jspciState
                | shiftState
                | loadState
                | storeState
                | branchState
                | copState
                | miscState
                | directState
```

```
                 | macroState
label            : ID :          { ID must be in column 1 }
binALUState      : binALUOp reg,reg,reg
binALUOp         : ADD
                 | SUB
                 | AND
                 | OR
                 | XOR
                 | ROTLB
                 | ROTLCB
                 | MSTEP
                 | DSTEP
                 | SUBNC
                 | BIC
monALUState      : monOp reg,reg
                 | MSTART reg,reg
monOp            : NOT
                 | MOV
specState        : MOVTOS reg,specialReg
                 | MOVFRS specialReg,reg
specialReg       : MD
                 | PSW
                 | 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
                 | STT exp[reg],reg
                 | STF exp[reg],freg
branchState      : branchOp reg,reg,ID
                 | branchSqOp reg,reg,ID
                 | BRA ID
branchOp         : BEQ
                 | BNE
                 | BGE
                 | BGT
                 | BHI
                 | BHS
                 | BLE
                 | BLO
                 | BLS
                 | BLT
branchSqOp       : BEQSQ
                 | BNESQ
                 | BGESQ
                 | BGTSQ
                 | BHISQ
                 | BHSSQ
                 | BLESQ
                 | BLOSQ
                 | BLSSQ
                 | BLTSQ
copState         : MOVTOC exp,reg
```

```
                | 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

[1] Cohen, Danny.
On Holy Wars and a Plea for Peace.
IEEE Computer 14(10):48-54, October, 1981.

[2] Gill, J., Gross, T., Hennessy, J., Jouppi, N., Przybylski, S. and Rowen, C.
Summary of MIPS Instructions.
Technical Note 83-237, Stanford University, November, 1983.

[3] Lamport, Leslie.
A Fast Mutual Exclusion Algorithm.
Technical Report 7, DEC Systems Research Center, November, 1985.