MIPS

MIPS Instruction Set R5

www.mips.com

This section covers the MIPS instruction set.

Instruction Set Overview

- **This section covers the MIPS32R3 instruction set**
  - Instructions will be broken down into 2 types of instructions:
    - Machine instructions – These instruction are directly defined by the MIPS Architecture specification.
    - Macro instructions – instructions that are aliases for actual machine instructions or instructions that the assembler will translate into multiple machine instructions.

MIPS

+ I am going to break down the instructions into two types.

+ a machine instruction which is directly defined in the MIPS architecture and has a one to one correspondence with a single instruction bit encoding. Most of these instruction are designed to execute in a single cycle as long as there are on data dependencies.

+ then there are macro instructions. These instructions are supplied by the assembler to make assemble coding easier. These instructions can be a simple as an alias to a machine instruction or translated into a series of machine instructions.

While the macro instruction do make it easier to write assemble programs they will make it more confusing to debug an assemble program. This is because when a debugger dissembles a stream of instructions from memory it will list them out a actual machine instructions and not translate them back into the macro instruction they came from.

Instruction Set Overview

- **All instructions one-word length (32 Bits)**
- **32 General Purpose Registers (GPRs)**
  - 2 special: GPR0 (zero) and GPR31 (return address)
- **3 operand instructions**
  - 2 source (rs, rt) and 1 destination (rd) register
  - Can have a 16 bit immediate value instead of 1 of the source registers.
- **Register length arithmetic only**
  - No byte or halfword arithmetic.
  - Some instructions do have 32 and 64 bit versions
  - Double length results go to HI/LO registers

MIPS 3

Here are some facts you should know before I begin to tell you more about the instructions.

+ for the MIPS32 architecture Instructions have a fixed length of 32 bits and are always aligned in memory on a word boundary.

+ In the MIPs architecture there are 32 General purpose registers

+ two of which are special. GPR0 will always reads 0 which is useful when you are comparing a value to 0 because it is already in a register. GPR31 is reserved for the return address from a Jump and link instruction. All other registers can be used for either address or data.

+ Instructions can have three operands,

+ 2 source registers and one destination register.

+ The immediate instructions have 16 immediate value instead of one of the source registers. The assembler does allow for some instructions to be abbreviated to have only 2 operands I'll give examples when I get to those instructions.

+ Arithmetic instructions operate on whole registers only,

+ there is no half word or byte arithmetic.

+ Some instruction have 32 or 64 bit versions.

+ Double length operands and results will go into the Hi and Lo registers that are not part of the General purpose registers. These registers are both 32bits wide and HI will hold the high order bits Lo will hold the low order bits.

Instruction Set Overview – cont.

- **Hardware Floating Point is optional through Coprocessor 1**
  - Hardware Floating Point units have their own registers
    - Floating Point Registers are interlocked

MIPS 4

+ Hardware floating Point is an optional feature

+ this unit has its own registers

+ which are interlocked in the pipe line so instructions dependent on floating point results will stall and wait for completion of the floating point operation

Floating point instructions are not covered in this section.

Load/Store Instructions

- **One addressing mode**
  - GPR + signed 16-bit offset
- **Addresses aligned on size boundary**
  - *Load/stores must be aligned*: Memory operations can only load or store data from addresses aligned to suit the data type being transferred
  - Bytes can be transferred at any address
  - Halfwords must be even-aligned to 2-byte boundaries
  - Word transfers aligned to 4-byte boundaries.
  - All offsets are in bytes

MIPS 5

+ There is only one addressing mode and that is register plus a signed immediate 16 bit off set this plus or minus 32K covers a 64K range of addresses.

+ All address are aligned to their data type boundary.

+ Byte or a char of course are 1 byte and aligned to a byte boundary.

+ Halfwords or shorts are 2 bytes and aligned to a 2 byte boundary,

+ Words, int or long are 4 bytes and aligned to a 4 byte boundary.

+ all offset values are in bytes

Load/Store Instructions

- **Delayed loads**
  - Memory data cannot be used immediately by next instruction because it arrives too late in the pipeline.
  - Loads are interlocked to the data usage. Therefore the processor will stall for a cycle if the next instruction tries to use the load data.
  - Compilers will try to avoid using data right after a load.

MIPS 6

+ Load instructions will always incur at least a one cycle delay even if it hits in the data cache.

+ The core pipeline is interlocked on the completion of the load data which will cause the processor to stall an instruction if that instruction uses the register the data is being loaded into.

+ compliers will try to avoid using data right after a load by trying to find a non dependent instruction to execute after a load.

Load, Store Instructions

MIPS

All load instructions use a destination register and a source register with a signed immediate offset. Each type of load instruction has a signed and unsigned version. The sighed version is signed extended and the unsigned version designated by a U is zero extended.

+The Load Byte instruction loads a byte from any address into the least significant byte of the destination register.

+The Load Halfword instruction loads a halfword from a halfword aligned address into the least significant bytes of the destination register.

+The Load word instruction loads a word from a word aligned address the destination register.

All store instructions use a source register and a destination register with a signed immediate offset.

+ store byte stores the least significant byte in the source register to any address

+ store halfword stores the 2 least significant bytes in the source register to a halfword aligned address

+ store word stores the contents of the source register to a word aligned address.

Load, Store Instructions

Mnemonic	Instruction	Assemble	Pseudo Code	MIPS32	MIPS16e
LB, LBU	Load Byte	lb t0, x(t1)	t0 = Mem[t1+x]	☑	☑
LH, LHU	Load Halfword	lh t0, x(t1)	t0 = Mem[t1 +x]	☑	☑
LW	Load Word	lw t0, x(t1)	t0 = Mem[t1 +x]	☑	☑
li	Load Immediate	lw s0, *Immediate*	s0 = *Immediate*		☑
SB	Store Byte	sb t0, x(t1)	Mem[t1+x] = t0	☑	☑
SH	Store Halfword	sh, t0, x(t1)	Mem[t1] = t0	☑	☑
SW	Store Word	sw t0, x(t1)	Mem[t1] = t0	☑	☑

8

All load instructions use a destination register and a source register with a signed immediate offset. Each type of load instruction has a signed and unsigned version. The sighed version is signed extended and the unsigned version designated by a U is zero extended.

+The Load Byte instruction loads a byte from any address into the least significant byte of the destination register.

+The Load Halfword instruction loads a halfword from a halfword aligned address into the least significant bytes of the destination register.

+The Load word instruction loads a word from a word aligned address the destination register.

All store instructions use a source register and a destination register with a signed immediate offset.

+ store byte stores the least significant byte in the source register to any address

+ store halfword stores the 2 least significant bytes in the source register to a halfword aligned address

+ store word stores the contents of the source register to a word aligned address.

Load Link and Store Conditional

- **LL, SC Instructions**
  - Provide the primitives to implement atomic read-modify-write (RMW) operations for synchronized memory locations.
  - There can be only one active RMW sequence per processor.
  - When an LL is executed it starts an active RMW sequence replacing any other sequence that was active.
  - The RMW sequence is completed by a subsequent SC instruction that either completes the RMW sequence atomically, or does not and fails.
  - SC will fail if
    - An ERET instruction is executed
    - Coherent store completed by another processor or coherent I/O module

MIPS 9

Load Link and Store conditional instructions provide very basic semaphore operations. For example when a device driver is accessing a part of shared memory that it shares with an interrupt routine.

+ the Load Link and Store conditional instruction sequence provides a way to implement a atomic read modify write operation

+ you can have one active Load Link Store conditional sequence active at a time.

+ If another Load Link is executed between a Load Link Store conditional pair the Store conditional from the original pair will fail.

+ the Read modify write sequence is completed by a subsequent Store conditional which will either succeed in the store or fail depending on the detection of conditions in-between which may have cause the memory location to be altered.

+ For a single processor system any exception return or ERET in-between a Load Link and Store conditional pair will cause the Store conditional to fail.

LL and SC Additional Failure Conditions

- **MT enabled Cores**
  - Any TC completes a store within the same cacheline as the word accessed by LL and SC.
  - The TCRestart register for the TC executing the LL/SC instruction sequence is written.
- **Cores Using the Memory Coherence Manager**
  - A coherent store is completed by another processor or coherent I/O module into the cacheline containing the word.
- **M4K core**
  - The SYNC instruction is externalized on the SRAM interface of the M4K core. External logic can use this information in a system-dependent manner to enforce memory ordering between various memory elements in the system.

10

Here are some additional failure condition that depend on the type of MIPS Core being used

+ For a Multi threaded enabled core

+ if any thread completes a store into the same cache line as an active Load Link Store conditional sequence or the thread executing the sequence is restarted the Store conditional will fail

+ In a multi core system using the MIPS Memory Coherence Manager hardware option any processor that writes into the same cache line will cause the Store conditional to fail

+ In a M4K implementation the sync instruction is externalized to allow ordering of memory writes by other memory elements in the system. If the sync signal is utilized like this in your system then any write by any memory element to the 2K byte memory region that contains the word being written will cause the Store conditional to fail.

STOP! Do not include in Video presentation:

For a MT enabled core

If either of the following events occurs between the execution of LL and SC, the SC may succeed or it may fail; the success or failure is not predictable. Portable programs should not cause one of these events.

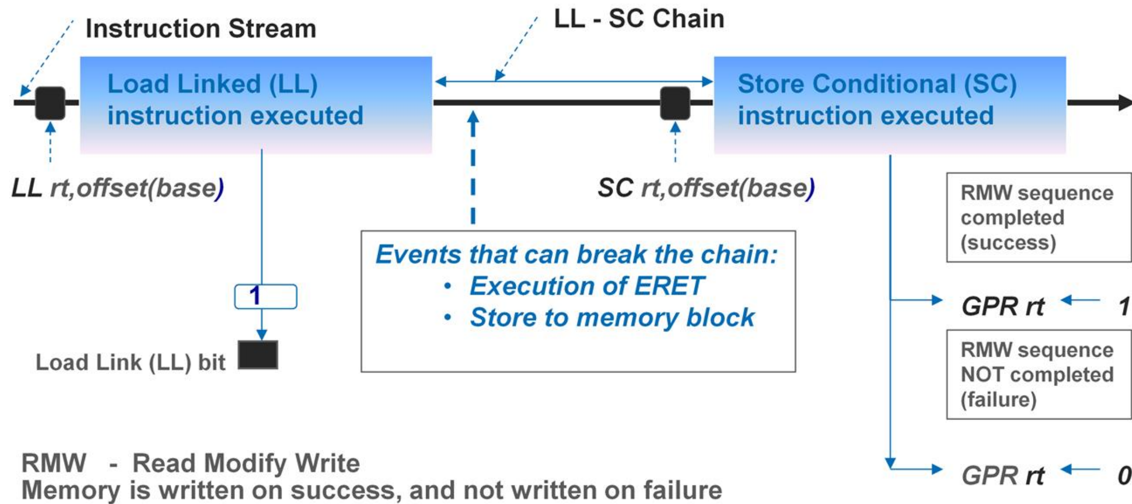
•A memory access instruction (load, store, or prefetch) is executed on the processor executing the LL/SC.

•The instructions executed starting with the LL and ending with the SC do not lie in a 2048-byte contiguous region of virtual memory. (The region does not have to be aligned, other than the alignment required for instruction words.)

The following conditions must be true or the result of the SC is **UNPREDICTABLE**:

•Execution of SC must have been preceded by execution of an LL instruction.

•On coherent core - An RMW sequence executed without intervening events that would cause the SC to fail must use the same address in the LL and SC. The address is the same if the virtual address, physical address, and cache-coherence algorithm are identical.



+ When an Load Link enters the instruction pipeline it starts an active Read Modify Write sequence replacing any other sequence that was active.

+ Load Link bit in the CP0 register is set

+ Execution continues until a Store conditional instruction is executed

+ If in-between the Load Link and Store conditional , there is no exception or a coherent store completed by another processor or coherent I/O module, into the block of synchronizable physical memory containing the word, the store will succeed.

+ A 1, indicating success, is written into source register .

+ Otherwise, memory is not modified and a 0, indicating failure, is written into source register.

Load Link and Store Conditional

Mnemonic	Instruction	Assemble	Pseudo Code	MIPS32	MIPS16e
LL	Load Link Word	ll t1, x(t0)	t1 = Mem[t0 +x]	☑	
SC	Store Conditional Word	sc t1, x(t0)	Mem[t0+x] = t1	☑	

- **Example of a wait for a binary semaphore:**

```
WaitForSem:
    la t0, sem
TryAgain:
    ll t1, 0(t0)
    bne t1, zero, WaitForSem
    li t1, 1
    sc t1, 0(t0)
    beq t1, zero, TryAgain
    /* got the semaphore... */
    jr ra
```

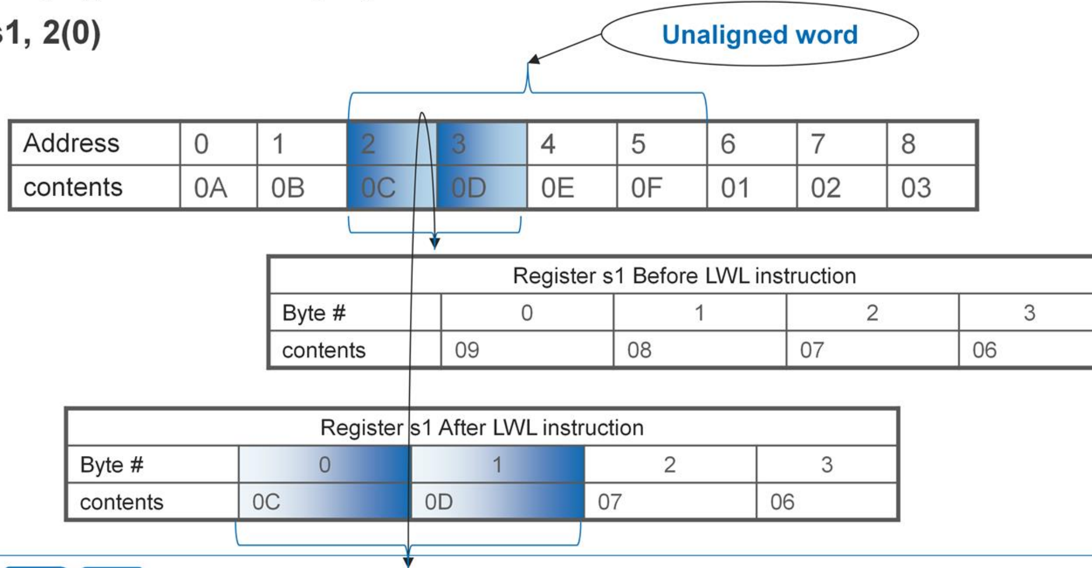
MIPS

13

Unaligned Loads and Stores

- **LWL (big endian example)**

lwl s1, 2(0)



Normal loads and stores in the MIPS architecture must be aligned; halfwords may be loaded only from 2-byte boundaries and words only from 4-byte boundaries. A load instruction with an unaligned address will produce an Address Error Exception so you should install a exception handler. This exception handler could emulate the desired load operation and hide the fact that the load was not aligned from the application. The exception handler could do this with a series of byte loads, shifts, and adds but the is a easier and quicker way.

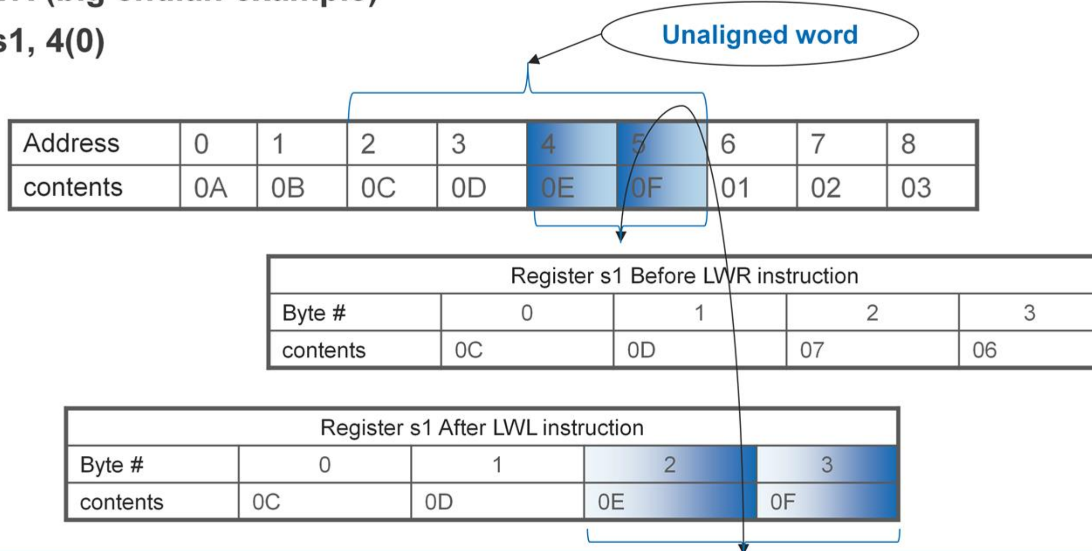
Here is an example of handling a unaligned store with a combination of Load Word Left and Load Word Right to get the job done with two instructions.

+ First use a Load Word left to load the most significant two bytes of the unaligned address into the most significant bytes of the destination register.

+ For this example the unaligned word is in bytes 2,3,4 and 5. S1 is the destination register and the address being loaded from, is address 0 with a offset of 2.

+ to make it clear about what is happening before the load starts S1 contains a hex 09 08 07 and 06

+ after the load word left you can see bytes 0 and 1 now contain 0c and 0D which was the value in the most significant bytes of the unaligned address.



+ Next use a Load Word right to load the least significant two bytes of the unaligned address into the least significant bytes of the destination register.

+ the least significant unaligned bytes are in are in bytes 4 and 5. S1 is the destination register and the address being loaded from is address 0 with a offset of 4.

+ to make it clear about what is happening before the load starts S1 contains a hex 0C 0D which we already moved there using the load work Left instruction and 07 and 06 in the least significant bytes

+ after the load word left you can see bytes 2 and 3 now contain 0E and 0F which was the value in the least significant bytes of the unaligned address.

Unaligned Load, Store

Mnemonic	Instruction	Assemble	Pseudo Code	MIPS32	MIPS16e
LWL	Load Word Left	lwl s1, x(a0	s1 = s1 merged Mem[a0 +x]	☑	
LWR	Load Word Right	lwr s1, x(a0)	s1 = s1 merged Mem[a0 +x]	☑	
SWL	Store Word Left	swl s1, x(a0)	Mem[a0 +x] = s1	☑	
SWR	Store Word Right	swr	Mem[a0 +x] = s1	☑	

MIPS

18

We see here the syntax of the Load word Left and right that I have already shown you in the example

+ Here is the syntax Store Word Right and Left instruction that will store a word from a register to an unaligned address.

Load Upper 16 bits

- **Load Upper Immediate loads an immediate value into the upper 16 bits of the destination register and zeros out the lower 16 bits.**
  - Commonly used in conjunction with ori to load an immediate address or 32 bit data value.
    - lui at x
    - ori at, at, 0x2345
  - Used in macro instructions li and la

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
LUI	Load Upper Immediate	lui s1, x	s1 = x << 16 | 0	☑	

MIPS

Arithmetic Instructions

- **Register length arithmetic only**
  - No byte or halfword arithmetic.
  - Double length results go to HI/LO registers
    - The low-order 32-bit word of the result is placed into special register LO.
    - High-order 32-bit word is placed into special register HI.

MIPS 20

I'll now talk about the Arithmetic Instructions.

+ The rule to remember is that all instructions use register length arithmetic so in the case of MIPS Core CPUs that means 32 bit arithmetic.

+ there is no byte or halfword arithmetic

+ double length results from multiply and divide instructions are placed in two special registers called HI and LO

+ the 4 least significant bytes go into the Lo register

+ and the 4 Most significant bytes go into the HI register

There are additional instructions to move the values from these register into a General Purpose Register which I will cover later

Arithmetic Instructions

Mnemonic	Instruction	Assemble	Pseudo Code	MIPS32	MIPS16e
ADD	Add	add s1, s2, s3	s1 = s2 + s3	☑	
ADDU	Add Unsigned	addu s1, s2, s3	s1 = s2 + s3	☑	☑
ADDI	Add Immediate	addi s1, s2, x	s1 = s2 + x	☑	
ADDIU	Add Immediate Unsigned	addiu s1, s2, x	s1 = s2 + x	☑	☑
SUB	Subtract	sub s1, s2, s3	s1 = s2 - s3	☑	
SUBU	Subtract Unsigned	subu s1, s2, s3	s1 = s2 - s3	☑	☑

I'll now go over the basic arithmetic instructions. Each one will have a signed and unsigned version. The unsigned version will end in a U.

+ First the simple add instruction which takes two registers and adds them together and places the result in a destination register.

+ Next is an add immediate which takes a 16 bit immediate value and adds it to the contents of a register and places the result in a destination register.

+ The subtract instruction subtraction subtracts one register value from another and places the result in a destination register. As you can see here there is no subtract immediate you can do that using the add immediate instruction and just supply a negative number.

+ The Multiply instruction multiplies the value in two registers and places the result in the HI and LO registers

+ The divide instruction divides the value in two registers and places the result in the HI and LO registers

One thing to note: what I am describing is here is actual machine instructions. The assembler allows a much broader range of these instructions through the use of instruction macros which the assembler will interrupt into actual machine instructions. For example The assembler will take either registers or immediate values up to 32 bits and do the right thing. I'll cover more on this in the assembler section.

Arithmetic Instructions

Mnemonic	Instruction	Assemble	Pseudo Code	MIPS32	MIPS16e
MULT	Multiply	mult s1, s2	HI LO = s1 x s2	☑	☑
MULTU	Multiply Unsigned	multu s1, s2	HI LO = s1 x s2	☑	☑
DIV	Divide	div s1, s2	HI LO = s1 / s2	☑	☑
DIVU	Divide Unsigned	divu s1, s2	HI LO = s1 / s2	☑	☑

MIPS

+ The Multiply instruction multiplies the value in two registers and places the result in the HI and LO registers

+ The divide instruction divides the value in two registers and places the result in the HI and LO registers

Arithmetic Instructions

Mnemonic	Instruction	Assemble	Pseudo Code	MIPS32	MIPS16e
SLT	Set on Less than	slt s0, s1, s2	If (s1 < s2) s0 = 1 else s0 = 0	☑	See MIPS16 only instructions
SLTU	Set on Less than Unsigned	sltu s0, s1, s2	If (s1 < s2) s0 = 1 else s0 = 0	☑	See MIPS16 only instructions
SLTI	Set on Less than Immediate	slti s0, s1, x	If (s1 < x) s0 = 1 else s0 = 0	☑	See MIPS16 only instructions
SLTIU	Set on Less than Immediate Unsigned	sltiu s0, s1, x	If (s1 < x) s0 = 1 else s0 = 0	☑	See MIPS16 only instructions

23

Here are the set instructions. These will test two values and place a 1 in the destination register if the first register is less than the second.

+ the other form of the instruction takes an immediate value instead of the second register so if the value in the register is less than the immediate value then the destination register is set to 1.

These machine instructions are used by the assemble to create macros for branch greater than, branch less than, branch equal and all combinations of the three.

Arithmetic Accumulate Instructions

- **Multiply add/subtract**
  - Multiply two 32 bit values and either add or subtract them from the HI/LO registers
  - The most significant 32 bits of the result are written into *HI* and the least significant 32 bits are written into *LO*.
  - No arithmetic exception occurs under any circumstances.

Mnemonic	Instruction	Assemble	Pseudo Code	MIPS32	MIPS16e
MADD MADDU unsigned	Multiply and add to HI/LO registers	madd s1, s2 maddu s1, s2	HI LO += s1 x s2	☑	
MSUB MSUBU unsigned	Multiply and subtract from HI/LO registers	msub s1, s2 msubu s1, s2	HI LO -= s1 x s2	☑	

MIPS

Arithmetic Instructions

- **Multiply – 32 bit product**
  - MUL - The 32-bit word value is multiplied by a 32-bit value, treating both operands as signed values, to produce a 64-bit result. The least significant 32 bits of the product are written to a GPR register. The contents of *HI* and *LO* are not preserved across this instruction

Mnemonic	Instruction	Assemble	Pseudo Code	MIPS32	MIPS16e
MUL	Multiply to GPR	mul s0, s1, s2	s0 = s1 x s2	☑	

MIPS 25

There is what I call a simple multiply instruction that will multiply two registers and place the least significant 32 bits into a general purpose register

You need to ensure that overflows are either avoided or irrelevant. Because there is no overflow detection .

Also you cannot intersperse MUL instructions with MADD instructions because the processor is allowed to use the HI and LO registers even though the results are stored in a General Purpose Register so any accumulated results are not preserved across the execution of a MUL instruction.

+ the machine instruction takes two registers multiplies them together and but the result in a third register

The assembler will allow you to use immediate values in place of the input registers.

Arithmetic Instructions

- **Count Leading Zeros Or Ones**

Mnemonic	Instruction	Assemble	Pseudo Code	MIPS32	MIPS16e
CLZ	Count leading 0s	clz s0, s1	s0 = number of leading Zeros in s1	☑	
CLO	Count leading 1s	clo s0, s1	s0 = number of leading Ones in s1	☑	

MIPS

There are two simple instructions to count either zeros or ones starting from the most significant bit.

+ Count Leading zeros starts at the most significant bit and travels backward counting zeros until it finds a one

+ Count Leading ones does the opposite starts at the most significant bit and travels backward counting ones until it finds a zero.

Logical Instructions

Mnemonic	Instruction	Assemble	Pseudo Code	MIPS32	MIPS16e
AND	Logical AND	and t0, t1, t2	t0 = t1 & t2	☑	☑
ANDI	Logical AND Immediate	andi t0, t1, x	t0 = t1 & x	☑	
OR	Logical OR	or t0, t1, t2	t0 = t1 | t2	☑	☑
ORI	Logical OR Immediate	ori t0, t1, x	t0 = t1 | x	☑	
NOR	Logical NOR	nor t0, t1, t2	If (!t1 && !t2) t0 = 1 else t0 = 0	☑	
XOR	Logical XOR	xor t0, t1 ,t2	if (t1 ^ t2) t0 = 1	☑	☑
XORI	Logical XOR Immediate	xori t0, t1, x	if (t1 ^ x) t0 = 1	☑	

27

Now I’ll cover the logical instructions

+ AND operation compares each bit in the two input registers if both bits are 1 then the corresponding bit in the destination register will be set to 1 other wise it will be set to 0.

+ OR operation compares each bit in the two input registers if both bits are 0 then the corresponding bit in the destination register will be set to 0 other wise it will be set to 1

+ NOR operation compares each bit in the two input registers if both bits are 0 then the corresponding bit in the destination register will be set to 1 other wise it will be set to 0

+ XOR operation compares each bit in the two input registers if one and only one bit is 1 then the corresponding bit in the destination register will be set to 1 other wise it will be set to 0

Move Instructions

Mnemonic	Instruction	Assemble	Pseudo Code	MIPS32	MIPS16e
MOVE	Move from one register to another	move a0, t0 (macro for: addu a0, zero, t0)	a0 = t0	☑	☑
MOVN	Move on not Zero	movn s0, s1, s2	If (s2 != 0) s0 = s1	☑	
MOVZ	Move on Zero	movz s0, s1, s2	If (s2 == 0) s0 = s1	☑	
MFHI	Move from HI	mfhi s0	s0 = HI	☑	☑
MFLO	Move from LO	mflo s0	s0 = LO	☑	☑
MTHI	Move to HI	mthi s0	HI = s0	☑	
MTLO	Move to LO	mtlo s0	LO = s0	☑	

Now for the move instructions

+ There is a move macro which would be used to copy the contents of one register to another. This translates to a unsigned add of register 0 to the register to be copied and placing the result in the destination register effectively making a copy.

+ Next are two instructions that test a register for true or false, 1 or 0 and conditionally do a register copy. These instructions are normally used in conjunction with the set instructions to test their result.

+ the last 4 move instructions either move from or to a Hi or Lo registers to a General purpose register.

Shift Instructions

Mnemonic	Instruction	Assemble	Pseudo Code	MIPS32	MIPS16e
SLL	Shift Left Logical	sll t0, t1, x	t0 = t1 << x Zero fill	☑	☑
SLLV	Shift Left Logical Variable	sll t0, t1, t2	t0 = t1 << t2 Zero fill	☑	☑
SRL	Shift right Logical	srl t0, t1, t2	t0 = t1 >> x Zero fill	☑	☑
SRLV	Shift right Logical Variable	srlv t0, t1, t2	t0 = t1 >> t2 Zero fill	☑	☑
SRA	Shift right Arithmetic	srat0, t1, t2	t0 = t1 >> x Sign bit duplication	☑	☑
SRAV	Shift right Arithmetic Variable	srav t0, t1, t2	t0 = t1 >> t2 Sign bit duplication	☑	☑

29

Shift instructions can be used to move bits around, or used instead of a multiply if multiplying by a power of 2. They were also used to isolate particular bits before MIPS32 release 2 but now we have insert and extract instructions that do that much more efficiently.

+ The shift left logical will shift the value in the source register to the left by the immediate value given, filling the bits with zeros. The Shift Left Logical is most notably looks the same as a nop because a Nop is instruction code is all zeros and the instruction code for SLL is also a zero so a Shift Left Logical where GPR 0 is both the source and destination register and the shift amount is 0 interrupts as the same as a nop. It also looks like the super scalar nop when the immediate value is 1 for much the same reason.

The shift left variable is almost the same but instead of a immediate value you provide a variable value in a register.

And of course there are the equivalent shift right instructions

There are also right shift arithmetic instructions that will preserve the sigh value with the shift instead of just zero filling.

Branch Instructions

- **Branch Delay Slot for branches (and jumps)**
  - There will be a 1 cycle delay while the new program counter is generated for a branch instruction.
  - To make it more efficient, the processor always executes the instruction immediately following branch/jump (except for Branch Likely).
  - The Assemblers will fill the BDS automatically unless the set .noreorder option is included in the assembler code.
  - Do not place branch, jump, ERET, DERET, or WAIT instructions in the BDS.
- **Branches have maximum of +-128KB displacement**
  - Conditional branches have only a 16-bit displacement field—giving a 2^{18}-byte range (instructions are 4 byte aligned) This immediate value is interpreted as a signed PC-relative displacement.

MIPS 30

Branches change the instruction stream to a different path usually based on a conditional evaluation.

+ The processor needs one cycle to install the new program counter for a taken branch

+ so instead of letting this cycle go to waist the processor will always execute the instruction that follows the branch. This is a good place to do a loop increment.

+The assembler will fill this Branch delay slot with an instruction of it’s choice. If that is not what you want you can use the set .noreorder directive to the assembler around areas of code where you want to hand fill the delay slots.

+ If you do this, be careful not to place any branch, Jump, Error return, Debug Error return or wait instruction in a delay slot.

+ Branches have a maximum of 128K plus or minus displacement from the current PC location .

+ The 16 bit immediate field is automatically shifted left two bits to align with a four byte instruction address giving it a range of 18 signed bits.

Branch Instructions

- **Branch Likely Instructions**
  - The instruction in the BDS is only executed if the branch is taken.
  - This instruction is useful in for loops where the code usually branches back to the beginning of the loop. The delay slot can be used to increment the loop counter.
  - The instruction is noted by the 'L' added to the normal branch instruction mnemonic.

MIPS 31

+ There is an exception to the rule that the branch always executes the instruction in the branch delay slot. In the case of a branch likely instruction the instruction in the branch delay slot is only executed if the branch is taken and not if the branch condition falls through.

+ Branch likely instructions are useful in loops where the code will loop several times through and the delay slot can be used to increment the counter.

+ Most branch instructions have a likely equivalent and are distinguished by ending in a "L"

Branch Instructions

Mnemonic	Instruction	Assemble	Pseudo Code	MIPS32	MIPS16e
BEQ BEQL	Branch if Equal	beq t0, t1, x	If (t0 == t1) PC = BDS address + x	☑	
BNE BNEL	Branch if Not Equal	bne t0, t1, x	If (t0 != t1) PC = BDS address + x	☑	
BLEZ BLEZL	Branch if Less than or Equal to Zero	blez t0, x	If (t0 <= 0) PC = BDS address + x	☑	
BGEZ BGEZL	Branch if Greater then or Equal to Zero	bgez t0, x	If (t0 >= 0) PC = BDS + x	☑	
BGEZAL BEGZALL	Branch if Greater then or Equal to Zero and Link	bgezal t0, x	If (t0 >= 0) PC = BDS address + x RA = PC + 8	☑	
BLEZAL BLEZALL	Branch if Less then or Equal to Zero and Link	blezal t0, x	If (t0 <= 0) PC = BDS address + x RA = PC + 8	☑	

32

Here are the machine branch instructions

+ Branch Equal will branch to the PC offset if the two source registers are equal

Branch Not Equal will branch to the PC offset if the two source registers are not equal

+ Branch Less than zero will branch to the PC offset if source register is less then or equal to zero

Branch Greater than zero branch to the PC offset if source register is greater then or equal zero

+ Branch Less than zero and Link will branch to the PC offset and set the return address register to the next instruction after the delay slot if source register is less then or equal to zero

Branch Greater than zero and link will branch to the PC offset and set the return address register to the next instruction after the delay slot if source register is greater then or equal to zero

Branch Instructions Macros

Mnemonic	Instruction	Assemble	Pseudo Code	MIPS32	MIPS16e
BEQZ BEQZL	Branch if Equal to Zero	beq t0,zero, x	If (t0 == 0) PC = BDS address + x	☑	☑
BGE BGEL	Branch Greater then or Equal	bge t0, t1, x	If (t0 >= t1) PC = BDS address + x	☑	
BGT BGTL	Branch if greater then	bgt t0, t1, x	If (t0 > t1) PC = BDS address + x	☑	
BLE BLEL	Branch if less then or Equal	ble t0, t1, x	If (t0 <= t1) PC = BDS + x	☑	
BLT BLTL	Branch if less then	blt t0, t1, x	If (t0 < t1) PC = BDS address + x	☑	
BLTZAL BLTZALL	Branch if Less then 0 and Link	bltzal t0, x	If (t0 < 0) PC = BDS address + x RA = PC + 8	☑	
B	Branch Unconditional	b x	PC = BDS address + x	☑	☑
BAL	Branch And Link Unconditional	bal x	PC = BDS address + x RA = PC +8	☑	

33

Here are the assembler macro branch instructions that fill in any other possible branch conditions. Remember these are made up of one or usually more machine instructions. I won't go over each one of these but you could pause now and review them at your leisure. These instructions will use the AT register GPR 1 so your code should leave this register alone. If your code uses the AT register the be sure to "set noat" the assembler will then give you a error if you use a macro instruction the uses AT.

Stop for video

+ branch to the PC offset if the two source registers are equal to zero

branch to the PC offset if first source register is greater then or equal the second source register

+ branch to the PC offset if first source register is greater then the second source register

branch to the PC offset if first source register is less then or equal the second source register

+ branch to the PC offset if first source register is less then the second source register

branch to the PC offset and set the return address register to the next instruction after the delay slot if source register is less then zero

+branch Unconditional does a PC relative jump

Branch and Link unconditional does a PC relative jump and set the return address in the return address register

Jump Instructions

- **Jump immediate instructions (J)**
  - A 26 bit immediate value to defines the target of a jump. Since all instructions are 4 byte aligned in memory, the two least-significant address bits need not be stored, allowing an address range of 2^{28} or 256MB.
  - The 28 bits of address is interpreted as an absolute address within a 256MB aligned segment, (not PC relative).
  - The jump address is computed by shifting the lower 26 bits of the instruction left by 2 and concatenating them with the upper 4 bits of the instruction in the branch delay slot.
- **Jumps beyond the 256M segment.**
  - A "jump register" (JR) instruction, can go to any 32-bit address.

MIPS 35

Lets talk about Jump instructions There are three types of jump instructions

+ Jump immediate instruction will jump to a absolute address with the current 256 megabyte PC-region.

+ You supply the instruction a 26 bit immediate value. This value is the target address of the jump shifted to the right by two. It is shifted to the right by two since all instructions are word aligned and the last two bits of the instruction address will always be 0. So this instruction uses that fact to pack a 28 bits address into 26 bits.

+ The current 256 megabyte PC region is defined by the 4 most significant bit of the address of the Branch Delay slot.

+ So the jump address is the immediate value shifted to the left by 2 and concatenated with the upper 4 bits of the address of the branch delay slot.

The good news is the assembler allows you to use the label of the jump target and it does the shifting and concatenating for you.

+ To reach out side the current 256 megabyte PC region you need to use the jump register instruction.

+ The jump register instruction uses a register that contains a 32 bit absolute address, so the jump target can be anywhere in the 4 gigabyte address range.

Jump Instructions

- **Procedure call by jump-and-link (JAL)**
  - Return address on linked branch/jump saved in ra (GPR31)
- **Delay Slot for Jumps (same as for Branches)**
  - There will be a 1 cycle delay while the new program counter is generated for a Jump instruction.
  - To make it more efficient, the processor always executes the instruction immediately following a Jump.
  - The assembler will fill the delay slot automatically unless the set .noreorder option is included in the assembler code.
  - Do not place branch, jump, ERET, DERET, or WAIT instructions in the delay slot.

MIPS 36

The Jump and link instruction use to call a subroutine, stores the return address into

a register. R**31** the return address register. Then when the subroutine wants to return it can execute a Jump Register instruction with R31 to return to the caller routine.

+ There is a delay slot for jumps. It is the same as it is for branches.

Jump Instructions

Mnemonic	Instruction	Assemble	Pseudo Code	MIPS32	MIPS16e
J	Jump target	j x	PC = ((PC +4) & 0xF000 0000) | (x << 2)	☑	
JAL	Jump target and link	jal x	PC = ((PC +4) & 0xF000 0000) | (x << 2)	☑	☑
JR JR.HB	Jump address in register	jr t0	RA = PC + 8 PC = t0	☑	☑
JALR JALR.HB	Jump address in register and Link	jal t0	PC = t0 RA = PC + 8	☑	☑

Adding .hb to JAL and JALR instructions will interlock the pipeline enabling it to stall while waiting to clear all execution and instruction hazards.

37

To summarize the jump instructions

+The Jump instruction jumps to and address using the immediate value, the label supplied with the instruction

+The Jump and link instruction does the same as the Jump instruction and in addition places a return address in the return address register.

+ The jump register instruction jumps to the address in the source register

+ and the jump and link register does the same as the Jump register instruction and in addition places a return address in the return address register.

+ adding the dot HB to the Jump and link or a jump and link register will cause the CPU to stall and wait for all execution and instruction hazards created by any CoProcessor 0 state changes. I'll talk more about these when I talk about the EHB instruction.

System Call Instruction

- **The SYSCALL instruction allows a user program to generate an identifiable exception.**
  - This exception enables switching from user mode to kernel mode. For example I/O devices are usually controlled by the OS, so when a user program needs to perform I/O it will do a system call to hand over control to the OS.
  - The OS uses a system call number to determine what system call is to be used.
  - A *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction. Not used in o32.

MIPS 39

The System call instruction is use

+ in a OS environment to transfer control from a user program to the to the OS running in Kernel mode.

+ The system call can a supply a unique number, usually referred to a the system call number, which the OS can use to determine which OS function is being requested.

+ This system call number can be retrieved by reading the instruction as data and extracting the code field. However As we will see the o32 standard uses an easier method to store and get the call number.

System Call Instruction Usage (o32)

- The syscall number is put in v0
- Arguments are passed in a0 through a3
- Return values are passed back in v0 and v1
- In keeping with the calling conventions, the syscall preserves the values of those registers which o32 defines as surviving function calls.

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
SYSCALL	System Call	Syscall x Note: x fills the code field but is not used for o32	Load system call number into v0 Load arguments into a0 - a3 Make the call Check return values in v0 and v1 as necessary	☑	

MIPS 40

Here is a run through of using a system call in Linux that uses the o32 standard.

+ The system call number is put into the v0 register. This makes it easier to retrieve instead of using the code field in the instruction.

+ The arguments to the system call are passed in the argument registers a0 through a3

+ Any return values can be passed back to the user code through registers v0 and v1

+ In keeping with the o32 standard the OS, Linux in this case, must preserve registers as if a function call was being preformed.

+ the instruction is called with a immediate value normally the system call number.

Usually a user program is not concerned with the system call interface because it would use a standard library to interface with the OS.

Trap Instructions

- Trap instructions are like conditional branches except they cause an exception.
- These are for compilers and interpreters that want to implement run-time checks. For example, checking array bounds. They are also used for assertions in the Linux kernel.
- All immediate values are 16 bits.

MIPS 41

Trap instructions are programmable conditional exceptions

+ Trap instructions evaluate a condition and can cause an exception.

+ For example an OS can use them for assertions, which are usually conditions that the OS assumes will never happen.

+ Some forms of the trap instruction can have a 16 bit immediate value to pass information back to the OS. The OS might use this information to determine what happened.

Trap Instructions

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
TEQ	Trap if Equal	teq s0, s1	If (s0 == s1) trap	☑	
TEQI	Trap if Equal Immediate	teqi s0, x	If (s0 == x) trap	☑	
TNE	Trap if not Equal	tne s0, s1	If (s0 != s1) trap	☑	
TNEI	Trap if not Equal Immediate	tnei s0, x	If (s0 != x) trap	☑	
TLT TLTU (unsigned)	Trap if Less than	tlt s0, s1 tltu s0, s1	If (s0 < s1) trap	☑	
TLTI TLTIU (unsigned)	Trap if Less than Immediate	tlti s0, x tltiu s0, x	If (s0 < x) trap	☑	
TGE TGEU (unsigned)	Trap if Greater than or Equal	tge s0, s1 tgeu s0, s1	If (s0 >= s1) trap	☑	
TGEI TGEIU (unsigned)	Trap if Greater than or Equal Immediate	tgei s0, x tgeiu s0, x	If (s0 >= x) trap	☑	

Since these are very similar to condition branches I have already covered I will not go into the details of each type of trap instruction.

BREAK Instruction

- **The BREAK instruction causes an unconditional exception. (unconditional Trap)**
  - Exception processing can determine the address where the break occurred by reading the CP0 EPC register.
  - The instruction can contain an immediate value (code) in bits 6 through 25. To get this value the programmer will need to load the instruction word stored at EPC and extract these bits.

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
BREAK	Breakpoint	break code	Execution break at EPC	☑	☑

MIPS 43

The Break instruction is usually use by debuggers to cause a break in execution. As the user instructs the debugger to set a breakpoint the debugger will replace the code at the breakpoint with a break instruction. Then when the break point happens the exception will turn control over the debugger. After the debugger has completed what the user wants and the user has instructed it to continue execution the debugger will put the program code back and execute it.

+ The break causes an unconditional exception

+ the Core puts address of the break instruction in the CP0 register EPC so, a debugger will know where the break happened.

+ the instruction can contain a twenty bit value that can be used to pass information to the exception. For example a debugger could use this as a indication of what type of break point it was or a pointer to the program instruction that the break replaced. This information will need to be extracted from the break instruction word.

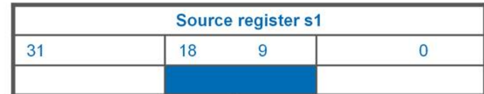
+ The break instruction takes one immediate value.

Extract Bit Field

- Easy way to read bits within a 32 bit value

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
EXT	Extract Bit Field	ext s0, s1, x, y		☑	

For example: ext s0, s1, 9, 10
Copy bits starting from bit 9 for 10 bits from register s1 to bits 0 through 9 in s0 and zero fill remaining bits.



MIPS 44

A new instruction introduced in MIPS32 Release 2 is the bit extract instruction.

+This instruction can replace a series of instruction that would have previously been needed to extract a bit or bits from a word. For example you would have previously need to do shift left and a shift right to extract a bit field to the least significant bits of a register.

+ The extract instruction takes 4 arguments, the destination register, the source register, the least significant bit of the field to be extracted and the number of bits to be extracted.

+ For example to extract 10 bits starting at bit 9 from register s1 to register s0 the assemble code would be “ext s0 s1 9 10”

+ this would start to copy from bit 9 of S1 to

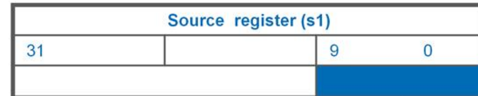
+ bit 0 of register s0 for 10 bits. The remaining bits in register s0 would be 0 filled.

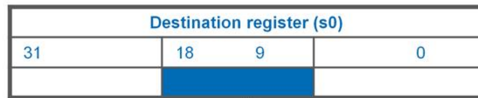
Insert Bit Field

- **Easy way to write bits within a 32 bit value**

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
INS	Insert Bit Field	ins s0, s1, x, y		☑	

For example: ins s0, s1, 9, 10
Copy bits starting from bit 0 for 10 bits in register s1 merge into bits 9 through 18 in s0.





MIPS 45

The counter part to the extract instruction is the insert instruction

+ as with the extract instruction this replaces several steps in merging a bit field into a word

+ the syntax for this instruction is the mnemonic ins, then the register to be merged , the source register that contains the bit field, the least significant bit in the destination register where the merge will start and the number of bytes to copy.

+ So to merge 10 bits into register s0 from register s1 starting at bit 9 the assemble code would be “ins s0 s1 9 10”

+ this would start to copy from bit 0 in register s1

+ to bit 9 in register s0 for 10 bits

Interrupt Control

- **DI - The interrupt-disabling sequence (requiring a read-modify-write sequence on SR) is itself not guaranteed to be atomic.**
  - Atomically clears the SR(IE) bit, returning the old value of SR in a general purpose register
- **EI - atomically sets the SR(IE) bit, returning the old value of SR in a general purpose register.**
  - Note of caution: It is better to restore the old value of SR returned by DI, so your “disable interrupts” code will not malfunction if you accidentally invoke it when interrupts were already disabled.

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
DI	Disable Interrupts	di s0	s0 = SR, SR[IE] = 0	☑	
EI	Enable Interrupts	ei s0	s0 = SR, SR[IE] = 1	☑	

MIPS 46

There two instructions that make interrupt control easier

+ First is the Disable interrupt instruction. This instruction helps by atomically writing the value of the Status register to a general purpose register and then clearing the Interrupt Enable bit in the CP0 Status register.

+ The Enable Interrupt instruction writes the value of the status register to a general purpose register and then sets the interrupt enable bit in the status register.

Note you might not always what to use the Enable interrupt instruction particularly if you are nesting interrupts. You might just what to restore the Status register with the value it had when it entered your interrupt function, so it would continue with processing any interrupt function that may have interrupted interrupted.

+ The assembler syntax for each is instruction the destination register where the Status register will be saved.

Software Debug Breakpoint

- **This instruction causes a debug exception, passing control to the debug exception handler.**
  - The code field can be used for passing information to the debug exception handler, and is retrieved by the debug exception handler only by loading the contents of the memory word containing the instruction, using the address in the DEPC register. The code field, bits 6 through 25 can then be extracted.

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
SDBBP	Software Debug Breakpoint	sdbbp code	Causes debug exception	☑	☑

MIPS

There is a software breakpoint instruction that works very much like the break instruction except it causes a debug mode exception which is tied into the EJTAG controller. A hardware EJTAG probe could use this instruction to set a breakpoint.

Exception Return

- **Returning from exceptions**
  - The instruction ERET, (exception return) clears the SR(EXL) bit and jumps to the address stored in EPC atomically.
  - DERET clears execution and instruction hazards, returns from Debug Mode and resumes non-debug execution at the instruction whose address is contained in the DEPC register.

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
ERET	Exception Return	eret	EXL = 0 PC = EPC	☑	
DERET	Debug Exception Return	deret	EXL = 0 PC = DEPC	☑	

MIPS 48

There are two instructions that can be used to return from exception handling.

+ The Exception Return instruction will atomically clear the exception bit EXL in the status register and then jump to the address stored in the Error Program Counter.

+ the Debug Exception Return instruction returns from a EJTAG debug exception and then jumps to the address stored in the Debug Error Program Counter.

+ the syntax for both is just the instruction mnemonic

Exception Return – Special Case

- **ERET will jump to the address stored in ErrorEPC when the ERL bit is set in the Status register.**
  - ERL- Error Level; Set by the processor when a Reset, Soft Reset, NMI or Cache Error exception are taken.
    - These exceptions can happen when the core is already taking an exception. This can prevent a infinite exception loop.

MIPS

There is a special case where the Error Return instruction will use the Error Error Program Counter instead of the Error Program Counter.

+ The Error Level bit is set in the status register during a Reset, Soft reset, Non Maskable Interrupt or a Cache error exception

+ exceptions can happen while processing these exceptions which are usually not recoverable so using the Error Error Program Counter is a way of not going into an infinite exception processing loop

Sign Extend bytes and halfwords

- **These instructions sign extend bytes and half-words so they are suitable for arithmetic instructions.**

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
SEB	Sign Extend Byte	seb t0, s1	Propagate bit 7 through out the entire word	☑	☑
SEH	Sign Extend Halfword	seh t0, s1	Propagate bit 15 through out the entire word	☑	☑

- **The SEH instruction can be used to convert two contiguous halfwords into two sign-extended word values in three instructions. For example:**

```
lw t0, 0(a1)     /* Read two contiguous halfwords */
seh t1, t0       /* t1 = lower halfword sign-extended to word */
sra t0, t0, 16   /* t0 = upper halfword sign-extended to word */
```

MIPS

50

The MIPs architecture does not allow arithmetic operations to be performed on anything other than integers. If there is a value that is less than a integer that needs to be used in an arithmetic expression in must first be made to look like an integer.

+ This can be done by sign extending a byte or half word.

+ The Sign Extend Byte instruction will propagate bit 7 from bit 8 through 31

+ the Sign Extend Half-word instruction will propagate bit 15 from bit 16 through bit 31

+ You can use the Sign extend half word instruction in-conjunction with Shift Right arithmetic to sign extend two contiguous half- words

+ first load the two contiguous half words into memory

+ next use the sign extend half-word instruction to sign extend the least significant half-work and copy the result to another register.

+ then use the shift right arithmetic with a value of 16 to shift the original value 16 bit and sign extend it.

Swap instructions

- **These instructions can be combined to swap the endianness of a word.**

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
WSBH	Word Swap Bytes Within Halfwords	wsbh t1, t2	Rotates by bytes each halfword of the word in t2 and places it into t1	☑	
ROTR	Rotate Word Right	rotr t2, t1, x	t2 = t1 <->(right) x	☑	
ROTRV	Rotate Word Right variable	rotrv t2, t1, t3	t2 = t1 <->(right) t3	☑	

```
lw t0, 0(a1)        /* 0xaabbccdd Read word value */
wsbh t0, t0         /* 0xbbaaddcc Convert endianness of halfwords */
rotr t0, t0, 16     /* 0xddccbbaa Swap the halfwords within the word */
```

51

There are three instructions that are used to manipulate bits within words.

+ these instructions can be combined to swap the endianess of a word with just two instructions.

+ The word swap within halfword instruction rotates bytes within both half-words

+ Rotate word right rotates bits in a word for a given number of bits set by the immediate value

+ Rotate word right variable is similar but it uses the value in a register instead of the immediate value to determine the amount of bits to rotate.

Let look at as example of an endianness swap

+ first load the word into a register

+ next swap the bytes in each half-word using the word-swap half-word instruction

+ then use the Rotate word instruction giving it a immediate value of 16 to swap the half-words

Moving to and from CP0 registers

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
MFC0	Move From CP0	mfco t0, CP0R, x	t0 = CP0R[x]	☑	
MTC0	Move To CP0	mtco t0, CP0R, x	CP0R[x] = t0	☑	
EHB	Execution Hazard Barrier	ehb	Stall till hazards cleared	☑	

- If you write a field of a CP0 register (MTC0) it may effect subsequent instructions, and the MIPS rules do not guarantee how long that might take.
- The EHB instruction alters the instruction issue behavior on a pipelined processor by stopping execution until all execution hazards have been cleared. All execution hazards created by previous instructions are cleared for instructions executed immediately following the EHB.

There are three instructions that are use in working with Co-Processor zero

+ Move From Co-processor zero moves from a Co-processor zero register, designated by a register and select numbers, to a general purpose register.

+ Move To Co-processor zero moves a value in a general purpose register to a Co-processor zero register designated by its register and select numbers

+ The EHB instruction will clear execution Hazard Barriers

+ Hazards are created when the Move to Co-processor zero instruction is used because there is no guarantee how long it might take for the value written to be usable by another instruction.

+ When the core executes a EHB instruction it will guarantee the value is safe to use by waiting for the value to be committed to the register.

You should consult your core’s Software Users Manual, for which actions cause hazards and how long the Hazard will last. Depending on the action not all Move to Co-processor zero instructions will cause a hazard and for others

as long as the value that was moved will not be used in a set period of time, a EHB may not be needed.

Reading CP0 Registers while in user mode

- **This instruction will allow a program in user mode to move the contents of a hardware register to a general purpose register (GPR) if that operation is enabled by software running in kernel mode. This is done by setting the appropriate bit in the CP0 HWREna register**

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
RDHWR	Read Hardware Register	rdhwr t0, x	t0 = HWR[x]	☑	

0	CPUNum – CPU number in a multi-processor system
1	SYNCI_Step - 0 no cache to sync or will indicate step size
2	CC – Cycle count
3	reserved
4	CCRes – Cycle count resolution
5-28	Reserved
29	ULR User Local Register
30 - 31	Reserved

MIPS

53

The read hardware register instruction allows access to some of the CP0 Registers while in user mode. Permission to do this is set in the HWREna register that needs to be setup while the cpu is in Kernel mode.

+ The assembler syntax for this instruction is a destination register and a operation code

I will use this instruction in a up coming example to get the sync step count, opcode 1

SYNC Instruction

- **A SYNC instruction defines a load/store barrier. You are guaranteed that all load/stores initiated before the SYNC will be completed before any load/store after the SYNC start.**
  - Affects only uncached and cached coherent loads and stores
  - To guarantee that memory reference results are visible across operating mode changes for example a sync is required on entry to and exit from Debug Mode to guarantee that memory affects are handled correctly.
  - The SYNC instruction stalls until all loads, stores, refills are completed and all write buffers are empty.

MIPS

54

The Sync instruction will stall any future loads and stores until all loads and stores are coherent through out the system. The use of the sync instruction is only necessary when memory is being shared between two or more devices within a system. For example if you are using a DMA controller to send data out of your system you need to make sure all stores to the uncached shared memory area between the CPU and the DMA controller have completed before you start the DMA. In a multi CPU system you may need to make sure before you load a value from a shared memory region into a CPU that another CPU doesn't have a newer value that has not yet been written to the shared memory for example it may have it in its cache.

+ a sync only affects uncached or cached coherent loads and stores. Cached loads and stores to non shared memory are no-coherent and you don't have to use the sync instruction for these areas.

+ A Sync is need when changing the operating mode between normal and EJTAG Debug and vise a versa to guarantee changes made to memory before and after the mode change is visible to both the EJTAG controller and the CPU.

+ the Sync instruction will stall the CPU until all uncached and cached coherent loads, stores and refills are completed and all the interim write buffers are empty.

SYNC Instruction continued

- If the Config7 ES (Externalize Sync) bit is set, executing a SYNC instruction will cause a synchronizing transaction on the external bus. How your system handles this transaction is system dependent. A typical system will flush any external write buffers and complete all pending transactions before completing the SYNC.
  - Note: This is not true for a 4KE, 4KSD or M4K, these do not have a Config7 register.

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
SYNC	To order loads and stores	sync		☑	

MIPS

55

The sync instruction does require external hardware to be able to propagate the sync signal to the external bus.

+ the Externalized Sync bit in the Config7 register of CoProcessor zero controls whether or not the sync signal is sent out to the external interface.

+ the exception to this is our 4KE, 4KSD and M4K cores where the signal is always sent.

+ the assembler syntax is simply sync

Synchronize Caches to Make Instruction Writes Effective

- **A SYNCI instruction is used after a new instruction stream is written to make the new instructions effective relative to an instruction fetch. It is available in all operating modes**
  - The operation occurs only on the cache line which contains the effective address. One SYNCI instruction is required for every cache line that was written.

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
SYNCI	Synchronize Caches	synci x(t0)	Synchronize cache line that contains t0[x]	☑	

MIPS

56

The SyncI instruction is used to synchronize a data cache line with a instruction cache line.

+ this is necessary when a program loads another program into memory because the load instruction only loads the data cache and to execute the code, the instruction cache for the data region needs to be invalidated so that when a fetch happens it will get an instruction cache miss and fetch the new instructions.

+ the SyncI instruction only effects the cache line of the address given so you will need to loop through all the effected cache lines.

+ You pass the address to the synci instruction using a register with a immediate offset

Synchronize Caches to Make Instruction Writes Effective - continued

- **The following example shows a routine which can be called after the new instruction stream is written to make those changes effective.**

```
/* Inputs: a0 = Start address of new instruction stream,  a1 = Size, in bytes, of new instruction stream */
        addu a1, a0, a1                 /* Calculate end address + 1 */
        rdhwr v0, SYNCI_Step            /* Get step size for SYNCI */
        beq v0, zero, 20f               /* If no caches require synchronization, */
        nop                             /* branch around */
10:     synci 0(a0)                     /* Synchronize all caches around address */
        sltu v1, a0, a1                 /* Compare current with end address */
        bne v1, zero, 10b               /* Branch if more to do */
        addu a0, a0, v0                 /* Add step size in delay slot */
        sync                            /* Clear memory hazards */
20:     jr.hb ra                        /* Return, clearing instruction hazards */
        nop
```

MIPS

57

Here is an example of a assemble function that can be used after code has been written into memory to synchronize the caches.

+ The function takes 2 inputs, the starting address in register a0 and the size of the area to sync in bytes

+ calculate the end address

+ get the size of a cache line

+ if the size is 0 we have no caches so no need to syncI so the code will jump to the return

+ Sync the first address

+ Check to see if it is the last cache line to sync

Use the delay slot to increment the byte count by the step size

And loop back if there is more to do

+ once we have looped through all the cache lines use the sync instruction to sync any coherent memory so other devices that might be sharing the memory will see the change

+ and return making sure that any instruction hazards have been cleared

Note the JR.HB instruction could be replaced with JALR.HB, ERET, or DERET instructions, as appropriate.

TLB Instructions

Mnemonic	Instruction	Assembly	MIPS32	MIPS16e
TLBR	Read Indexed TLB Entry	tlbr	☑	
TLBWI	Write Indexed TLB Entry	tlbwi	☑	
TLBWR	Write Random TLB Entry	tlbwr	☑	
TLBP	Probe TLB for matching Entry	tlbp	☑	

- **TLBR - The EntryHi, EntryLo0, EntryLo1, and PageMask registers are loaded with the contents of the TLB entry pointed to by the Index register.**
- **TLBWI - The TLB entry pointed to by the Index register is written from the contents of the EntryHi, EntryLo0, EntryLo1, and PageMask registers.**
- **TLBWR - The TLB entry pointed to by the Random register is written from the contents of the EntryHi, EntryLo0, EntryLo1, and PageMask registers.**
- **TLBP - The Index register is loaded with the address of the TLB entry whose contents match the contents of the EntryHi register. If no TLB entry matches, the high-order bit of the Index register is set.**
- **The TLB is covered in more detail later in this class.**

MIPS

58

There are four instructions the control the translation look aside buffer. These will be covered in detail in a separate TLB section.

Cache Instructions

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
CACHE	Perform a cache operation	cache op, x(t0)	Cache operation on address based at t0[x]	☑	
PREF	Move data between memory and cache	pref y, x(t0)	Prefetch memory t0[x] based on hint y	☑	

These instructions are covered in more detail later in this class

MIPS

Like wise there are two instructions the control the cache these will be covered in detail in a separate section on the Cache.

Wait Instruction

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
WAIT	Force core into low power mode	wait		☑	

- The WAIT instruction causes the CPU to enter a low-power sleep mode until woken by an interrupt. Most of the core logic is stopped, but the Count register, in particular, continues to run. The WAIT instruction is commonly used in the OS idle loop when there is nothing for the CPU to do.
- The WAIT instruction is covered in more detail later in this class

MIPS

60

The wait instruction can be used to control the power consumption of the CPU when there is nothing to do but wait for an interrupt. This instruction will be cover in detail in a separate section on power.

Instructions for Shadow Register Sets

Mnemonic	Instruction	Assembly	Pseudo code	MIPS32	MIPS16e
RDPGPR	Read Previous GPR	rdpgpr t0, 1	t0 = SGPR[pss][1]	☑	
WRPGPR	Write Previous GPR	wrpgpr 1, t0	SGPR[pss][1] = t0	☑	

- A core that does not support shadow sets will still implement these instructions, causing a register-to-register copy.
- These instructions are covered in more detail later in this class

MIPS 61

There are two instruction that control reading and writing Shadow register set since these work with interrupts they are covered in detail in a separate section for exceptions and interrupts.