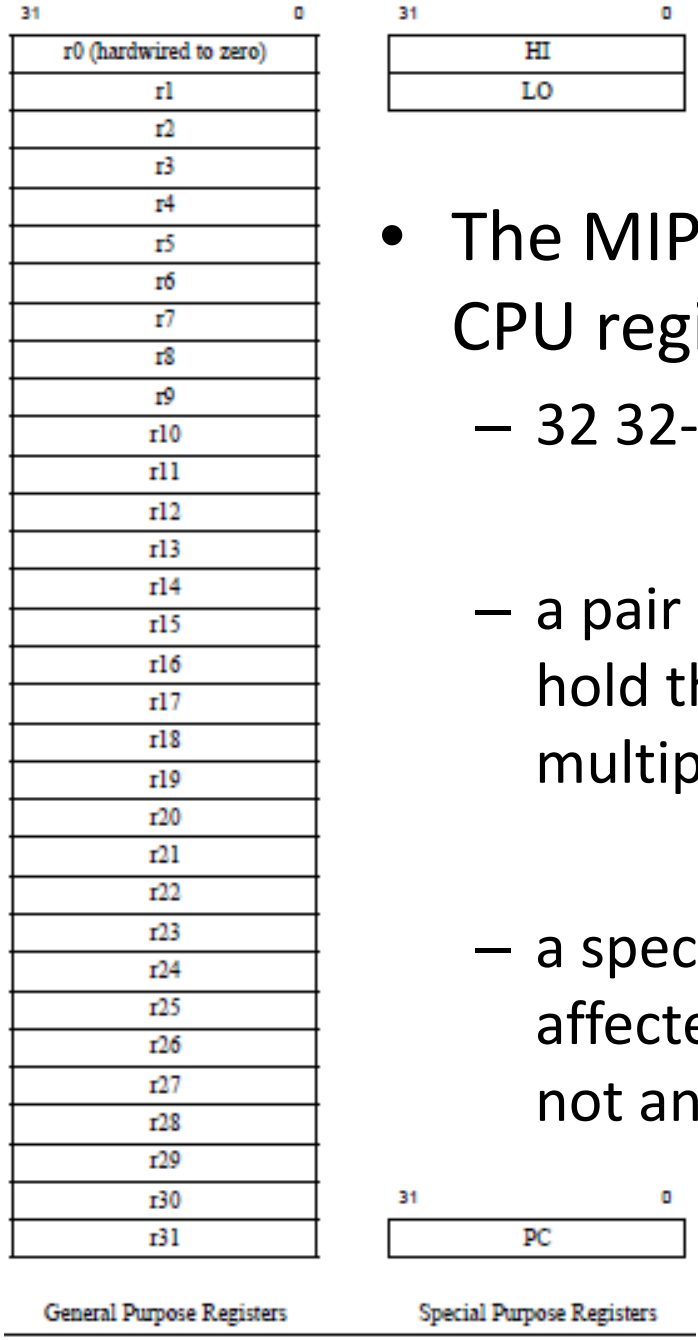
Introduction to the 32-bit MIPS processor architecture

Figure 2-8 CPU Registers



CPU registers

- The MIPS32 Architecture defines the following CPU registers:
  - 32 32-bit general purpose registers (GPRs)
  - a pair of special-purpose registers (each 32-bits) to hold the results of integer multiply, divide, and multiply-accumulate operations (HI and LO)
  - a special-purpose program counter (PC), which is affected only indirectly by certain instructions (it is not an architecturally-visible register)

CPU GPRs

- Two of the CPU general-purpose registers have assigned functions:
  1. r0 is hard-wired to a value of zero, and can be used as the target register for any instruction whose result is to be discarded. r0 can also be used as a source when a zero value is needed.
  2. r31 is the destination register used by JAL, BLTZAL, BLTZALL, BGEZAL, and BGEZALL without being explicitly specified in the instruction word. Otherwise r31 is used as a normal register.
- The remaining registers are available for general-purpose use.

CPU special purpose registers

- The CPU contains three special-purpose registers:
  1. PC—Program Counter register
  2. HI (multiply and divide register higher result)
  3. LO (multiply and divide register lower result)
     - During a multiply operation, the HI and LO registers store the product of integer multiply.
     - During a multiply-add or multiply-subtract operation, the HI and LO registers store the result of the integer multiply-add or multiply-subtract.
     - During a division, the HI and LO registers store the quotient (in LO) and remainder (in HI) of integer divide.
     - During a multiply-accumulate, the HI and LO registers store the accumulated result of the operation.

Suggested GPR uses:

- constant 0

Register name	Number	Usage
$zero	0	constant 0
$at	1	reserved for assembler
$v0	2	expression evaluation and results of a function
$v1	3	expression evaluation and results of a function
$a0	4	argument 1
$a1	5	argument 2
$a2	6	argument 3
$a3	7	argument 4
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (used by function call)

Figure 6.1 MIPS registers and usage convention.

Suggested GPR uses:

- **May be used as temporary register during macro expansion via assembler. (So avoid using it.)**

Example:

Load Address (used to initialize pointers): la Rd, Label

lui $at, Upper-16-bits-of-Label

ori Rd, $at, Lower-16-bits of-Label

Register name	Number	Usage
$at	1	reserved for assembler
$v1	3	expression evaluation and results of a function
$a0	4	argument 1
$a1	5	argument 2
$a2	6	argument 3
$a3	7	argument 4
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (used by function call)

Figure 6.1 MIPS registers and usage convention.

Suggested GPR uses:

- function return values (2), and
- expression evaluation

Register name	Number	Usage
$zero	0	constant 0
$v0	2	expression evaluation and results of a function
$v1	3	expression evaluation and results of a function
$a1	5	argument 2
$a2	6	argument 3
$a3	7	argument 4
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (used by function call)

Figure 6.1 MIPS registers and usage convention.

Suggested GPR uses:

- function arguments (4)

Register name	Number	Usage
$zero	0	constant 0
$at	1	reserved for assembler
$v0	2	expression evaluation and results of a function
$v1	3	expression evaluation and results of a function
$a0	4	argument 1
$a1	5	argument 2
$a2	6	argument 3
$a3	7	argument 4
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (used by function call)

Figure 6.1 MIPS registers and usage convention.

Suggested GPR uses:

- temporary (not preserved)

Register name	Number	Usage
$zero	0	constant 0
$at	1	reserved for assembler
$v0	2	expression evaluation and results of a function
$v1	3	expression evaluation and results of a function
$a0	4	argument 1
$a1	5	argument 2
$a2	6	argument 3
$a3	7	argument 4
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (used by function call)

Figure 6.1 MIPS registers and usage convention.

Suggested GPR uses:

- temporary (preserved)

Register name	Number	Usage
$zero	0	constant 0
$at	1	reserved for assembler
$v0	2	expression evaluation and results of a function
$v1	3	expression evaluation and results of a function
$a0	4	argument 1
$a1	5	argument 2
$a2	6	argument 3
$a3	7	argument 4
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (used by function call)

Figure 6.1 MIPS registers and usage convention.

Suggested GPR uses:

- global pointer
- stack pointer
- frame pointer
- return address

Register name	Number	Usage
$zero	0	constant 0
$at	1	reserved for assembler
$v0	2	expression evaluation and results of a function
$v1	3	expression evaluation and results of a function
$a0	4	argument 1
$a1	5	argument 2
$a2	6	argument 3
$a3	7	argument 4
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (used by function call)

Figure 6.1 MIPS registers and usage convention.

Overview of the CPU instruction set

- CPU instructions are organized into the following functional groups:
  1. Load and store
  2. Computational
  3. Jump and branch
  4. Miscellaneous
  5. Coprocessor
- Each instruction is 32 bits long.

CPU LOAD AND STORE INSTRUCTIONS

CPU load and store instructions

- MIPS processors use a load/store architecture; all operations are performed on operands held in processor registers and main memory is accessed only through load and store instructions.

Types of loads and stores

- There are several different types of load and store instructions, each designed for a different purpose:
  - transferring variously-sized fields (for example, LB, SW)
  - trading transferred data as signed or unsigned integers (for example, LHU)
  - accessing unaligned fields (for example, LWR, SWL)
  - selecting the addressing mode (for example, SDXC1, in the FPU)
  - atomic memory update (read-modify-write: for instance, LL/SC)

Types of loads and stores

- Regardless of the byte ordering (big- or little-endian), the address of a halfword, word, or doubleword is the lowest byte address among the bytes forming the object:
  - For big-endian ordering, this is the most-significant byte.
  - For a little-endian ordering, this is the least-significant byte.

4.1.1.2 Load and Store Access Types

Table 4.1 lists the data sizes that can be accessed through CPU load and store operations. These tables also indicate the particular ISA within which each operation is defined.

Table 4.1 Load and Store Operations Using Register + Offset Addressing Mode

Data Size	CPU			Coprocessors 1 and 2	
	Load Signed	Load Unsigned	Store	Load	Store
Byte	MIPS32	MIPS32	MIPS32		
Halfword	MIPS32	MIPS32	MIPS32		
Word	MIPS32	MIPS64	MIPS32	MIPS32	MIPS32
Doubleword (FPU)				MIPS32	MIPS32
Unaligned word	MIPS32		MIPS32		
Linked word (atomic modify)	MIPS32		MIPS32		

Table 4.2 Aligned CPU Load/Store Instructions

Mnemonic	Instruction	Defined in MIPS ISA
LB	Load Byte	MIPS32
LBU	Load Byte Unsigned	MIPS32
LH	Load Halfword	MIPS32
LHU	Load Halfword Unsigned	MIPS32
LW	Load Word	MIPS32

Table 4.2 Aligned CPU Load/Store Instructions (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
SB	Store Byte	MIPS32
SH	Store Halfword	MIPS32
SW	Store Word	MIPS32

Table 4.3 Unaligned CPU Load and Store Instructions

Mnemonic	Instruction	Defined in MIPS ISA
LWL	Load Word Left	MIPS32
LWR	Load Word Right	MIPS32
SWL	Store Word Left	MIPS32
SWR	Store Word Right	MIPS32

Table 4.4 Atomic Update CPU Load and Store Instructions

Mnemonic	Instruction	Defined in MIPS ISA
LL	Load Linked Word	MIPS32
SC	Store Conditional Word	MIPS32

COMPUTATIONAL INSTRUCTIONS

Computational instructions

- ALU Immediate
- ALU Three-Operand
- ALU Two-Operand
- Shift
- Multiply and Divide

ALU Immediate (16-bit) Instructions

Table 4.7 ALU Instructions With a 16-bit Immediate Operand

Mnemonic	Instruction	Defined in MIPS ISA
ADDI	Add Immediate Word	MIPS32
ADDIU[1]	Add Immediate Unsigned Word	MIPS32
ANDI	And Immediate	MIPS32
LUI	Load Upper Immediate	MIPS32
ORI	Or Immediate	MIPS32
SLTI	Set on Less Than Immediate	MIPS32
SLTIU	Set on Less Than Immediate Unsigned	MIPS32
XORI	Exclusive Or Immediate	MIPS32

ALU Three-Operand Instructions

Table 4.8 Three-Operand ALU Instructions

Mnemonic	Instruction	Defined in MIPS ISA
ADD	Add Word	MIPS32
ADDU[1]	Add Unsigned Word	MIPS32

ALU Three-Operand Instructions (cont'd.)

Table 4.8 Three-Operand ALU Instructions (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
AND	And	MIPS32
NOR	Nor	MIPS32
OR	Or	MIPS32
SLT	Set on Less Than	MIPS32
SLTU	Set on Less Than Unsigned	MIPS32
SUB	Subtract Word	MIPS32
SUBU[1]	Subtract Unsigned Word	MIPS32
XOR	Exclusive Or	MIPS32

ALU Two-Operand Instructions

Table 4.9 Two-Operand ALU Instructions

Mnemonic	Instruction	Defined in MIPS ISA
CLO	Count Leading Ones in Word	MIPS32
CLZ	Count Leading Zeros in Word	MIPS32

Shift Instructions

Table 4.10 Shift Instructions

Mnemonic	Instruction	Defined in MIPS ISA
ROTR	Rotate Word Right	MIPS32 Release 2
ROTRV	Rotate Word Right Variable	MIPS32 Release 2
SLL	Shift Word Left Logical	MIPS32
SLLV	Shift Word Left Logical Variable	MIPS32
SRA	Shift Word Right Arithmetic	MIPS32

Shift Instructions (cont'd.)

Table 4.10 Shift Instructions (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
SRAV	Shift Word Right Arithmetic Variable	MIPS32
SRL	Shift Word Right Logical	MIPS32
SRLV	Shift Word Right Logical Variable	MIPS32

Multiply and Divide Instructions

- The multiply and divide instructions produce twice as many result bits.
- With one exception (MUL), they deliver their results into the HI and LO special registers. (The MUL instruction delivers the lower half of the result directly to a GPR.)
  - Multiply produces a full-width product twice the width of the input operands; the low half is loaded into LO and the high half is loaded into HI.
  - Multiply-Add and Multiply-Subtract produce a full-width product twice the width of the input operations and adds or subtracts the product from the concatenated value of HI and LO. The low half of the addition is loaded into LO and the high half is loaded into HI.
  - Divide produces a quotient that is loaded into LO and a remainder that is loaded into HI.
- The results are accessed by instructions that transfer data between HI/LO and the general registers.

Multiply and Divide Instructions

Table 4.11 Multiply/Divide Instructions

Mnemonic	Instruction	Defined in MIPS ISA
DIV	Divide Word	MIPS32
DIVU	Divide Unsigned Word	MIPS32
MADD	Multiply and Add Word	MIPS32
MADDU	Multiply and Add Word Unsigned	MIPS32
MFHI	Move From HI	MIPS32
MFLO	Move From LO	MIPS32
MSUB	Multiply and Subtract Word	MIPS32
MSUBU	Multiply and Subtract Word Unsigned	MIPS32
MTHI	Move To HI	MIPS32
MTLO	Move To LO	MIPS32
MUL	Multiply Word to Register	MIPS32
MULT	Multiply Word	MIPS32
MULTU	Multiply Unsigned Word	MIPS32

JUMP AND BRANCH INSTRUCTIONS

Types of jump and branch instructions

- The architecture defines the following jump and branch instructions:
  1. PC-relative conditional branch
  2. PC-region unconditional jump
  3. Absolute (register) unconditional jump
  4. A set of procedure calls that record a return link address in a general register.

Table 4.12 Unconditional Jump Within a 256 Megabyte Region

Mnemonic	Instruction	Defined in MIPS ISA
J	Jump	MIPS32
JAL	Jump and Link	MIPS32
JALX	Jump and Link Exchange	MIPS16e MIPS32 Release 3

Table 4.13 Unconditional Jump using Absolute Address

Mnemonic	Instruction	Defined in MIPS ISA
JALR	Jump and Link Register	MIPS32
JALR.HB	Jump and Link Register with Hazard Barrier	MIPS32 Release 2
JR	Jump Register	MIPS32
JR.HB	Jump Register with Hazard Barrier	MIPS32 Release 2

Table 4.14 PC-Relative Conditional Branch Instructions Comparing Two Registers

Mnemonic	Instruction	Defined in MIPS ISA
BEQ	Branch on Equal	MIPS32
BNE	Branch on Not Equal	MIPS32

Table 4.15 PC-Relative Conditional Branch Instructions Comparing With Zero

Mnemonic	Instruction	Defined in MIPS ISA
BGEZ	Branch on Greater Than or Equal to Zero	MIPS32
BGEZAL	Branch on Greater Than or Equal to Zero and Link	MIPS32
BGTZ	Branch on Greater Than Zero	MIPS32
BLEZ	Branch on Less Than or Equal to Zero	MIPS32
BLTZ	Branch on Less Than Zero	MIPS32
BLTZAL	Branch on Less Than Zero and Link	MIPS32

Table 4.16 Deprecated Branch Likely Instructions

Mnemonic	Instruction	Defined in MIPS ISA
BEQL	Branch on Equal Likely	MIPS32
BGEZALL	Branch on Greater Than or Equal to Zero and Link Likely	MIPS32
BGEZL	Branch on Greater Than or Equal to Zero Likely	MIPS32
BGTZL	Branch on Greater Than Zero Likely	MIPS32
BLEZL	Branch on Less Than or Equal to Zero Likely	MIPS32
BLTZALL	Branch on Less Than Zero and Link Likely	MIPS32
BLTZL	Branch on Less Than Zero Likely	MIPS32
BNEL	Branch on Not Equal Likely	MIPS32

Branch delays and the branch delay slot

- **All branches have an architectural delay of one instruction.**

- The instruction immediately following a branch is said to be in the branch delay slot. (If a branch or jump instruction is placed in the branch delay slot, the operation of both instructions is UNPREDICTABLE!)

Branch delays and the branch delay slot

- There are two versions of branches and jumps; they differ in the manner in which they handle the instruction in the delay slot when the branch is not taken and execution falls through.
  1. Branch and Jump instructions execute the instruction in the delay slot.
  2. Branch Likely instructions do not execute the instruction in the delay slot if the branch is not taken.
     1. Software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

MISCELLANEOUS INSTRUCTIONS

Miscellaneous instructions

- Miscellaneous instructions include:
  1. instruction serialization (SYNC and SYNCI)
  2. exception instructions
  3. conditional move instructions
  4. prefetch instructions
  5. NOP instructions

1. Instruction serialization

2. Exception instructions

Table 4.18 System Call and Breakpoint Instructions

Mnemonic	Instruction	Defined in MIPS ISA
BREAK	Breakpoint	MIPS32
SYSCALL	System Call	MIPS32

Table 4.19 Trap-on-Condition Instructions Comparing Two Registers

Mnemonic	Instruction	Defined in MIPS ISA
TEQ	Trap if Equal	MIPS32
TGE	Trap if Greater Than or Equal	MIPS32
TGEU	Trap if Greater Than or Equal Unsigned	MIPS32
TLT	Trap if Less Than	MIPS32
TLTU	Trap if Less Than Unsigned	MIPS32
TNE	Trap if Not Equal	MIPS32

Table 4.20 Trap-on-Condition Instructions Comparing an Immediate Value

Mnemonic	Instruction	Defined in MIPS ISA
TEQI	Trap if Equal Immediate	MIPS32
TGEI	Trap if Greater Than or Equal Immediate	MIPS32
TGEIU	Trap if Greater Than or Equal Immediate Unsigned	MIPS32
TLTI	Trap if Less Than Immediate	MIPS32
TLTIU	Trap if Less Than Immediate Unsigned	MIPS32
TNEI	Trap if Not Equal Immediate	MIPS32

3. Conditional move instructions

Table 4.21 CPU Conditional Move Instructions

Mnemonic	Instruction	Defined in MIPS ISA
MOVF	Move Conditional on Floating Point False	MIPS32
MOVN	Move Conditional on Not Zero	MIPS32
MOVT	Move Conditional on Floating Point True	MIPS32
MOVZ	Move Conditional on Zero	MIPS32

4. Prefetch instructions

Table 4.22 Prefetch Instructions

Mnemonic	Instruction	Addressing Mode	Defined in MIPS ISA
PREF	Prefetch	Register+Offset	MIPS32
PREFX	Prefetch Indexed	Register+Register	MIPS64 MIPS32 Release 2

5. NOP instructions

Table 4.23 NOP Instructions

Mnemonic	Instruction	Defined in MIPS ISA
NOP	No Operation	MIPS32
SSNOP	Superscalar Inhibit NOP	MIPS32

SPIM – THE MIPS SIMULATOR

SPIM subset of MIPS assembler directives:

code

.align n

Align the next datum on a 2 n byte boundary. For example, .align 2 aligns the next value on a word boundary. .align 0 turns off automatic alignment of .half, .word, .float, and .double directives until the next .data or .kdata directive.

.extern sym size

Declare that the datum stored at sym is size bytes and is a global label. This directive enables assembler to store the datum in a portion of data segment that is efficiently accessed via register $gp.

.globl sym

Declare that label sym is global and can be referenced from other files.

.ktext

Subsequent items are put in the kernel text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument addr is present, subsequent items are stored starting at address addr.

.set noat and .set at

The first directive prevents SPIM from complaining about subsequent instructions that use register $at. The second directive reenables the warning. Since pseudoinstructions expand into code that uses register $at, programmers must be very careful about leaving values in this register.

.text <addr>

Subsequent items are put in the user text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument addr is present, subsequent items are stored starting at address addr.

SPIM subset of MIPS assembler directives:

data

.align n

Align the next datum on a 2n byte boundary. For example, .align 2 aligns the next value on a word boundary. .align 0 turns off automatic alignment of .half, .word, .float, and .double directives until the next .data or .kdata directive.

.data

Subsequent items are stored in the data segment. If the optional argument addr is present, subsequent items are stored starting at address addr. (SPIM does not distinguish various parts of the data segment (.data, .rdata, and .sdata).)

.kdata

Subsequent data items are stored in the kernel segment. If the optional argument addr is present, subsequent items are stored starting at address addr.

SPIM subset of MIPS assembler directives:

data allocation

.ascii str

Store the string str in memory, but do not null-terminate it.

.asciiz str

Store the string str in memory and null-terminated it.

.byte b1, ..., bn

Store the n values in successive bytes of memory.

.double d1, ..., dn

Store the n floating-point double precision numbers in successive memory locations.

.extern sym size

Declare that the datum stored at sym is size bytes and is a global label. This directive enables assembler to store the datum in a portion of data segment that is efficiently accessed via register $gp.

.float f1, ..., fn

Store the n floating-point single precision numbers in successive memory locations.

.half h1, ..., hn

Store the n 16-bit quantities in successive memory halfwords.

.space n

Allocate n bytes of space in the current segment (which must be the data segment in SPIM).

.word w1, ..., wn

Store the n 32-bit quantities in successive memory words.

System calls

Service	System Call Code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		

System calls – print_str

```
        .data
str:    .asciiz “Hello World”

        .text
        .globl   main
main:
        li   $v0,  4            # code for print_str
        la  $a0,  str           # argument
        syscall         # executes print_str
```

System calls – read _int

```
        .data
num:    .space 4

        .text
        .globl   main
main:
        li  $v0,  5                 # code for read_int
        syscall             # executes read_int
                                    # return value is stored in $v0
        la   $t0, num               # load address of num to $t0
        sw  $v0, 0($t0)             # store the number in num
```

- No mov $1, $2
  - use add $1, $2, 0
- No ld $1, 0x12345678
  - use lui $1, 0x1234
    ori $1, $1, 0x5678
- No jmp or branch unconditional (always).
  - use beq $zero, $zero, loop

References

1. **MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32® Architecture**, Revision 3.02, 2011, Document Number: MD00082, MIPS Technologies Inc.

2. **MIPS Assemby Language Programming**, Robert L. Britton, 2002.