MIPS Instruction Reference

CONTENTS

This is a **partial list** of the available MIPS32 instructions, system calls, and assembler directives. For more MIPS instructions, refer to the Assembly Programming section on the class Resources page.
In all examples, `$1`, `$2`, `$3` represent registers. For class, you should use the register names, not the corresponding register numbers.

Arithmetic Instructions

Instruction	Example	Meaning	Comments
add	`add $1,$2,$3`	$1=$2+$3	
subtract	`sub $1,$2,$3`	$1=$2-$3	
add immediate	`addi $1,$2,100`	$1=$2+100	"Immediate" means a constant number
add unsigned	`addu $1,$2,$3`	$1=$2+$3	Values are treated as unsigned integers, not two's complement integers
subtract unsigned	`subu $1,$2,$3`	$1=$2-$3	Values are treated as unsigned integers, not two's complement integers
add immediate unsigned	`addiu $1,$2,100`	$1=$2+100	Values are treated as unsigned integers, not two's complement integers
Multiply (without overflow)	`mul $1,$2,$3`	$1=$2*$3	Result is only 32 bits!

Multiply	`mult $2,$3`	$hi,$low=$2*$3	Upper 32 bits stored in special register `hi` Lower 32 bits stored in special register `lo`
Divide	`div $2,$3`	$hi,$low=$2/$3	Remainder stored in special register `hi` Quotient stored in special register `lo`
Unsigned Divide	`divu $2,$3`	$hi,$low=$2/$3	$2 and $3 store unsigned values. Remainder stored in special register `hi` Quotient stored in special register `lo`

Logical

Instruction	Example	Meaning	Comments
and	`and $1,$2,$3`	$1=$2&$3	Bitwise AND
or	`or $1,$2,$3`	$1=$2|$3	Bitwise OR
and immediate	`andi $1,$2,100`	$1=$2&100	Bitwise AND with immediate value
or immediate	`or $1,$2,100`	$1=$2|100	Bitwise OR with immediate value
shift left logical	`sll $1,$2,10`	$1=$2<<10	Shift left by constant number of bits
shift right logical	`srl $1,$2,10`	$1=$2>>10	Shift right by constant number of bits

Data Transfer

Instruction	Example	Meaning	Comments
load word	`lw $1,100($2)`	$1=Memory[$2+100]	Copy from memory to register
store word	`sw $1,100($2)`	Memory[$2+100]=$1	Copy from register to memory

load upper immediate	`lui $1,100`	$1=100x2^16	Load constant into upper 16 bits. Lower 16 bits are set to zero.
load address	`la $1,label`	$1=Address of label	*Pseudo-instruction* (provided by assembler, not processor!) Loads computed address of label (not its contents) into register
load immediate	`li $1,100`	$1=100	*Pseudo-instruction* (provided by assembler, not processor!) Loads immediate value into register
move from hi	`mfhi $2`	$2=hi	Copy from special register `hi` to general register
move from lo	`mflo $2`	$2=lo	Copy from special register `lo` to general register
move	`move $1,$2`	$1=$2	*Pseudo-instruction* (provided by assembler, not processor!) Copy from register to register.

Variations on load and store also exist for smaller data sizes:

- 16-bit halfword: `lh` and `sh`
- 8-bit byte: `lb` and `sb`

Conditional Branch

All conditional branch instructions compare the values in two registers together. If the comparison test is true, the branch is taken (i.e. the processor jumps to the new location). Otherwise, the processor continues on to the next instruction.

Instruction	Example	Meaning	Comments
branch on equal	`beq $1,$2,100`	if($1==$2) go to PC+4+100	Test if registers are equal
branch on not equal	`bne $1,$2,100`	if($1!=$2) go to PC+4+100	Test if registers are not equal
branch on greater than	`bgt $1,$2,100`	if($1>$2) go to PC+4+100	*Pseduo-instruction*
branch on greater than	`bge`	if($1>=$2) go to	*Pseduo-instruction*

or equal	`$1,$2,100`	PC+4+100	
branch on less than	`blt $1,$2,100`	if($1<$2) go to PC+4+100	*Pseduo-instruction*
branch on less than or equal	`ble $1,$2,100`	if($1<=$2) go to PC+4+100	*Pseduo-instruction*

Note 1: It is much easier to use a label for the branch instructions instead of an absolute number. For example: `beq $t0, $t1, equal`. The label "equal" should be defined somewhere else in the code.
Note 2: There are **many variations** of the above instructions that will **simplify writing programs**! Consult the Resources for further instructions, particularly H&P Appendix A.

Comparison

Instruction	Example	Meaning	Comments
set on less than	`slt $1,$2,$3`	if($2<$3)$1=1; else $1=0	Test if less than. If true, set $1 to 1. Otherwise, set $1 to 0.
set on less than immediate	`slti $1,$2,100`	if($2<100)$1=1; else $1=0	Test if less than. If true, set $1 to 1. Otherwise, set $1 to 0.

Note: There are **many variations** of the above instructions that will **simplify writing programs**! Consult the Resources for further instructions, particularly H&P Appendix A.

Unconditional Jump

Instruction	Example	Meaning	Comments
jump	`j 1000`	go to address 1000	Jump to target address
jump register	`jr $1`	go to address stored in $1	For switch, procedure return
jump and link	`jal 1000`	$ra=PC+4; go to address 1000	Use when making procedure call. This saves the return address in $ra

Note: It is much easier to use a label for the jump instructions instead of an absolute number. For example: `j loop`. That label should be defined somewhere else in the code.

System Calls

The SPIM simulator provides a number of useful system calls. These are **simulated**, and **do not represent MIPS processor instructions**. In a real computer, they would be implemented by the operating system and/or standard library.

System calls are used for input and output, and to exit the program. They are initiated by the `syscall` instruction. In order to use this instruction, you must first supply the appropriate arguments in registers $v0, $a0-$a1, or $f12, depending on the specific call desired. (In other words, not all registers are used by all system calls). The syscall will return the result value (if any) in register $v0 (integers) or $f0 (floating-point).
Available syscall services in SPIM:

Service	Operation	Code (in $v0)	Arguments	Results
print_int	Print integer number (32 bit)	1	$a0 = integer to be printed	None
print_float	Print floating-point number (32 bit)	2	$f12 = float to be printed	None
print_double	Print floating-point number (64 bit)	3	$f12 = double to be printed	None
print_string	Print null-terminated character string	4	$a0 = address of string in memory	None
read_int	Read integer number from user	5	None	Integer returned in $v0
read_float	Read floating-point number from user	6	None	Float returned in $f0
read_double	Read double floating-point number from user	7	None	Double returned in $f0
read_string	Works the same as Standard C Library `fgets()` function.	8	$a0 = memory address of string input buffer $a1 = length of string buffer (n)	None
sbrk	Returns the address to a block of memory containing n additional bytes. (Useful for dynamic memory allocation)	9	$a0 = amount	address in $v0

exit	Stop program from running	10	None	None
print_char	Print character	11	$a0 = character to be printed	None
read_char	Read character from user	12	None	Char returned in $v0
exit2	Stops program from running and returns an integer	17	$a0 = result (integer number)	None

Notes:

- The **print_string** service expects the address to start a null-terminated character string. The directive **.asciiz** creates a null-terminated character string.
- The **read_int**, **read_float** and **read_double** services read an entire line of input up to and including the newline character.
- The **read_string** service has the same semantics as the C Standard Library routine fgets().
  - The programmer must first allocate a buffer to receive the string
  - The read_string service reads up to *n-1* characters into a buffer and terminates the string with a null character.
  - If fewer than *n-1* characters are in the current line, the service reads up to and including the newline and terminates the string with a null character.
- There are a few additional system calls not shown above for file I/O:
  open, **read**, **write**, **close** (with codes 13-16)

Assembler Directives

An assembler directive allows you to request the assembler to do something when converting your source code to binary code.

Directive	Result
`.word w1, ..., wn`	Store *n* 32-bit values in successive memory words
`.half h1, ..., hn`	Store *n* 16-bit values in successive memory words
`.byte b1, ..., bn`	Store *n* 8-bit values in successive memory words
`.ascii str`	Store the ASCII string str in memory. Strings are in double-quotes, i.e. "Computer Science"

`.asciiz str`	Store the ASCII string str in memory and **null-terminate** it Strings are in double-quotes, i.e. "Computer Science"
`.space n`	Leave an empty *n*-byte region of memory for later use
`.align n`	Align the next datum on a 2^n byte boundary. For example, .align 2 aligns the next value on a word boundary

Registers

MIPS has 32 general-purpose registers that could, technically, be used in any manner the programmer desires. However, by convention, registers have been divided into groups and used for different purposes. Registers have both a *number* (used by the hardware) and a *name* (used by the assembly programmer).
This table **omits special-purpose registers** that will not be used in ECPE 170.

Register Number	Register Name	Description
0	**$zero**	The value 0
2-3	**$v0 - $v1**	(**v**alues) from expression evaluation and function results
4-7	**$a0 - $a3**	(**a**rguments) First four parameters for subroutine
8-15, 24-25	**$t0 - $t9**	**T**emporary variables
16-23	**$s0 - $s7**	**S**aved values representing final computed results
31	**$ra**	**R**eturn **a**ddress

This is a description of the MIPS instruction set, their meanings, syntax, semantics, and bit encodings. The syntax given for each instruction refers to the assembly language syntax supported by the MIPS assembler. Hyphens in the encoding indicate "don't care" bits which are not considered when an instruction is being decoded.

General purpose registers (GPRs) are indicated with a dollar sign ($). The words SWORD and UWORD refer to 32-bit signed and 32-bit unsigned data types, respectively.

The manner in which the processor executes an instruction and advances its program counters is as follows:

1. execute the instruction at *PC*
2. copy *nPC* to *PC*
3. add 4 or the branch offset to *nPC*

This behavior is indicated in the instruction specifications below. For brevity, the function advance_pc (int) is used in many of the instruction descriptions. This function is defined as follows:

```
void advance_pc (SWORD offset)
{
    PC   =  nPC;
   nPC  += offset;
}
```

Note: ALL arithmetic immediate values are sign-extended. After that, they are handled as signed or unsigned 32 bit numbers, depending upon the instruction. The only difference between signed and unsigned instructions is that signed instructions can generate an overflow exception and unsigned instructions can not.

The instruction descriptions are given below:

ADD – *Add (with overflow)*

Description:	Adds two registers and stores the result in a register
Operation:	$d = $s + $t; advance_pc (4);
Syntax:	add $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 0000`

ADDI -- *Add immediate (with overflow)*

Description:	Adds a register and a sign-extended immediate value and stores the result in a

	register
Operation:	$t = $s + imm; advance_pc (4);
Syntax:	addi $t, $s, imm
Encoding:	`0010 00ss ssst tttt iiii iiii iiii iiii`

ADDIU -- *Add immediate unsigned (no overflow)*

Description:	Adds a register and a sign-extended immediate value and stores the result in a register
Operation:	$t = $s + imm; advance_pc (4);
Syntax:	addiu $t, $s, imm
Encoding:	`0010 01ss ssst tttt iiii iiii iiii iiii`

ADDU -- *Add unsigned (no overflow)*

Description:	Adds two registers and stores the result in a register
Operation:	$d = $s + $t; advance_pc (4);
Syntax:	addu $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 0001`

AND -- *Bitwise and*

Description:	Bitwise ands two registers and stores the result in a register
Operation:	$d = $s & $t; advance_pc (4);
Syntax:	and $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 0100`

ANDI -- *Bitwise and immediate*

Description:	Bitwise ands a register and an immediate value and stores the result in a register
Operation:	$t = $s & imm; advance_pc (4);
Syntax:	andi $t, $s, imm
Encoding:	`0011 00ss ssst tttt iiii iiii iiii iiii`

BEQ -- *Branch on equal*

Description:	Branches if the two registers are equal

Operation:	if $s == $t advance_pc (offset << 2)); else advance_pc (4);
Syntax:	beq $s, $t, offset
Encoding:	`0001 00ss ssst tttt iiii iiii iiii iiii`

BGEZ -- *Branch on greater than or equal to zero*

Description:	Branches if the register is greater than or equal to zero
Operation:	if $s >= 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bgez $s, offset
Encoding:	`0000 01ss sss0 0001 iiii iiii iiii iiii`

BGEZAL -- *Branch on greater than or equal to zero and link*

Description:	Branches if the register is greater than or equal to zero and saves the return address in $31
Operation:	if $s >= 0 $31 = PC + 8 (or nPC + 4); advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bgezal $s, offset
Encoding:	`0000 01ss sss1 0001 iiii iiii iiii iiii`

BGTZ -- *Branch on greater than zero*

Description:	Branches if the register is greater than zero
Operation:	if $s > 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bgtz $s, offset
Encoding:	`0001 11ss sss0 0000 iiii iiii iiii iiii`

BLEZ -- *Branch on less than or equal to zero*

Description:	Branches if the register is less than or equal to zero
Operation:	if $s <= 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax:	blez $s, offset
Encoding:	`0001 10ss sss0 0000 iiii iiii iiii iiii`

BLTZ -- *Branch on less than zero*

Description:	Branches if the register is less than zero
Operation:	if $s < 0 advance_pc (offset << 2)); else advance_pc (4);

Syntax:	bltz $s, offset
Encoding:	`0000 01ss sss0 0000 iiii iiii iiii iiii`

BLTZAL -- *Branch on less than zero and link*

Description:	Branches if the register is less than zero and saves the return address in $31
Operation:	if $s < 0 $31 = PC + 8 (or nPC + 4); advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bltzal $s, offset
Encoding:	`0000 01ss sss1 0000 iiii iiii iiii iiii`

BNE -- *Branch on not equal*

Description:	Branches if the two registers are not equal
Operation:	if $s != $t advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bne $s, $t, offset
Encoding:	`0001 01ss ssst tttt iiii iiii iiii iiii`

DIV -- *Divide*

Description:	Divides $s by $t and stores the quotient in $LO and the remainder in $HI
Operation:	$LO = $s / $t; $HI = $s % $t; advance_pc (4);
Syntax:	div $s, $t
Encoding:	`0000 00ss ssst tttt 0000 0000 0001 1010`

DIVU -- *Divide unsigned*

Description:	Divides $s by $t and stores the quotient in $LO and the remainder in $HI
Operation:	$LO = $s / $t; $HI = $s % $t; advance_pc (4);
Syntax:	divu $s, $t
Encoding:	`0000 00ss ssst tttt 0000 0000 0001 1011`

J -- *Jump*

Description:	Jumps to the calculated address
Operation:	PC = nPC; nPC = (PC & 0xf0000000) | (target << 2);
Syntax:	j target

Encoding:	`0000 10ii iiii iiii iiii iiii iiii iiii`

JAL -- *Jump and link*

Description:	Jumps to the calculated address and stores the return address in $31
Operation:	$31 = PC + 8 (or nPC + 4); PC = nPC; nPC = (PC & 0xf0000000) | (target << 2);
Syntax:	jal target
Encoding:	`0000 11ii iiii iiii iiii iiii iiii iiii`

JR -- *Jump register*

Description:	Jump to the address contained in register $s
Operation:	PC = nPC; nPC = $s;
Syntax:	jr $s
Encoding:	`0000 00ss sss0 0000 0000 0000 0000 1000`

LB -- *Load byte*

Description:	A byte is loaded into a register from the specified address.
Operation:	$t = MEM[$s + offset]; advance_pc (4);
Syntax:	lb $t, offset($s)
Encoding:	`1000 00ss ssst tttt iiii iiii iiii iiii`

LUI -- *Load upper immediate*

Description:	The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes.
Operation:	$t = (imm << 16); advance_pc (4);
Syntax:	lui $t, imm
Encoding:	`0011 11-- ---t tttt iiii iiii iiii iiii`

LW -- *Load word*

Description:	A word is loaded into a register from the specified address.
Operation:	$t = MEM[$s + offset]; advance_pc (4);
Syntax:	lw $t, offset($s)
Encoding:	`1000 11ss ssst tttt iiii iiii iiii iiii`

MFHI -- *Move from HI*

Description:	The contents of register HI are moved to the specified register.
Operation:	$d = $HI; advance_pc (4);
Syntax:	mfhi $d
Encoding:	`0000 0000 0000 0000 dddd d000 0001 0000`

MFLO -- *Move from LO*

Description:	The contents of register LO are moved to the specified register.
Operation:	$d = $LO; advance_pc (4);
Syntax:	mflo $d
Encoding:	`0000 0000 0000 0000 dddd d000 0001 0010`

MULT -- *Multiply*

Description:	Multiplies $s by $t and stores the result in $LO.
Operation:	$LO = $s * $t; advance_pc (4);
Syntax:	mult $s, $t
Encoding:	`0000 00ss ssst tttt 0000 0000 0001 1000`

MULTU -- *Multiply unsigned*

Description:	Multiplies $s by $t and stores the result in $LO.
Operation:	$LO = $s * $t; advance_pc (4);
Syntax:	multu $s, $t
Encoding:	`0000 00ss ssst tttt 0000 0000 0001 1001`

NOOP -- *no operation*

Description:	Performs no operation.
Operation:	advance_pc (4);
Syntax:	noop
Encoding:	`0000 0000 0000 0000 0000 0000 0000 0000`

Note: The encoding for a NOOP represents the instruction SLL $0, $0, 0 which has no side effects. In fact, nearly every instruction that has $0 as its destination register will have no side effect and can thus be considered a NOOP instruction.

OR -- *Bitwise or*

Description:	Bitwise logical ors two registers and stores the result in a register
Operation:	$d = $s | $t; advance_pc (4);
Syntax:	or $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 0101`

ORI -- *Bitwise or immediate*

Description:	Bitwise ors a register and an immediate value and stores the result in a register
Operation:	$t = $s | imm; advance_pc (4);
Syntax:	ori $t, $s, imm
Encoding:	`0011 01ss ssst tttt iiii iiii iiii iiii`

SB -- *Store byte*

Description:	The least significant byte of $t is stored at the specified address.
Operation:	MEM[$s + offset] = (0xff & $t); advance_pc (4);
Syntax:	sb $t, offset($s)
Encoding:	`1010 00ss ssst tttt iiii iiii iiii iiii`

SLL -- *Shift left logical*

Description:	Shifts a register value left by the shift amount listed in the instruction and places the result in a third register. Zeroes are shifted in.
Operation:	$d = $t << h; advance_pc (4);
Syntax:	sll $d, $t, h
Encoding:	`0000 00ss ssst tttt dddd dhhh hh00 0000`

SLLV -- *Shift left logical variable*

Description:	Shifts a register value left by the value in a second register and places the result in a third register. Zeroes are shifted in.
Operation:	$d = $t << $s; advance_pc (4);
Syntax:	sllv $d, $t, $s
Encoding:	`0000 00ss ssst tttt dddd d--- --00 0100`

SLT -- *Set on less than (signed)*

Description:	If $s is less than $t, $d is set to one. It gets zero otherwise.
Operation:	if $s < $t $d = 1; advance_pc (4); else $d = 0; advance_pc (4);
Syntax:	slt $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 1010`

SLTI -- *Set on less than immediate (signed)*

Description:	If $s is less than immediate, $t is set to one. It gets zero otherwise.
Operation:	if $s < imm $t = 1; advance_pc (4); else $t = 0; advance_pc (4);
Syntax:	slti $t, $s, imm
Encoding:	`0010 10ss ssst tttt iiii iiii iiii iiii`

SLTIU -- *Set on less than immediate unsigned*

Description:	If $s is less than the unsigned immediate, $t is set to one. It gets zero otherwise.
Operation:	if $s < imm $t = 1; advance_pc (4); else $t = 0; advance_pc (4);
Syntax:	sltiu $t, $s, imm
Encoding:	`0010 11ss ssst tttt iiii iiii iiii iiii`

SLTU -- *Set on less than unsigned*

Description:	If $s is less than $t, $d is set to one. It gets zero otherwise.
Operation:	if $s < $t $d = 1; advance_pc (4); else $d = 0; advance_pc (4);
Syntax:	sltu $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 1011`

SRA -- *Shift right arithmetic*

Description:	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. The sign bit is shifted in.
Operation:	$d = $t >> h; advance_pc (4);
Syntax:	sra $d, $t, h
Encoding:	`0000 00-- ---t tttt dddd dhhh hh00 0011`

SRL -- *Shift right logical*

Description:	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. Zeroes are shifted in.

Operation:	$d = $t >> h; advance_pc (4);
Syntax:	srl $d, $t, h
Encoding:	`0000 00-- ---t tttt dddd dhhh hh00 0010`

SRLV -- *Shift right logical variable*

Description:	Shifts a register value right by the amount specified in $s and places the value in the destination register. Zeroes are shifted in.
Operation:	$d = $t >> $s; advance_pc (4);
Syntax:	srlv $d, $t, $s
Encoding:	`0000 00ss ssst tttt dddd d000 0000 0110`

SUB -- *Subtract*

Description:	Subtracts two registers and stores the result in a register
Operation:	$d = $s - $t; advance_pc (4);
Syntax:	sub $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 0010`

SUBU -- *Subtract unsigned*

Description:	Subtracts two registers and stores the result in a register
Operation:	$d = $s - $t; advance_pc (4);
Syntax:	subu $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 0011`

SW -- *Store word*

Description:	The contents of $t is stored at the specified address.
Operation:	MEM[$s + offset] = $t; advance_pc (4);
Syntax:	sw $t, offset($s)
Encoding:	`1010 11ss ssst tttt iiii iiii iiii iiii`

SYSCALL -- *System call*

Description:	Generates a software interrupt.
Operation:	advance_pc (4);

Syntax:	syscall
Encoding:	`0000 00-- ---- ---- ---- ---- --00 1100`

The syscall instruction is described in more detail on the System Calls page.

XOR -- *Bitwise exclusive or*

Description:	Exclusive ors two registers and stores the result in a register
Operation:	$d = $s ^ $t; advance_pc (4);
Syntax:	xor $d, $s, $t
Encoding:	`0000 00ss ssst tttt dddd d--- --10 0110`

XORI -- *Bitwise exclusive or immediate*

Description:	Bitwise exclusive ors a register and an immediate value and stores the result in a register
Operation:	$t = $s ^ imm; advance_pc (4);
Syntax:	xori $t, $s, imm
Encoding:	`0011 10ss ssst tttt iiii iiii iiii iiii`