MIPS Instruction Set

Registers

MIPS-32 microprocessor has 32 registers general-purpose registers, each of 32-bits in size. The most commonly used are mentioned in the below table.

Register Number	Register Name	Description
0	**$zero**	The value 0
2-3	**$v0 - $Sv1**	(**v**alues) from expression evaluation and function results
4-7	**$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

MIPS instructions

This is a **partial list** of the available MIPS-32 instructions, system calls, and assembler directives.

Arithmetic Instructions

Instruction	Example	Meaning	Comments
add	`add $S1,$S2,$S3`	$S1=$S2+$S3	
subtract	`sub $S1,$S2,$S3`	$S1=$S2-$S3	
add immediate	`addi $S1,$S2,100`	$S1=$S2+100	"Immediate" means a constant number
Multiply (without overflow)	`mul $S1,$S2,$S3`	$S1=$S2*$S3	Result is only 32 bits!

Divide	`div $S2,$S3`	$Shi,$Slow=$S2/$S3	Remainder stored in special register `hi` Quotient stored in special register `lo`

Logical

Instruction	Example	Meaning	Comments
and	`and $S1,$S2,$S3`	$S1=$S2&$S3	Bitwise AND
or	`or $S1,$S2,$S3`	$S1=$S2|$S3	Bitwise OR
and immediate	`andi $S1,$S2,100`	$S1=$S2&100	Bitwise AND with immediate value
or immediate	`ori $S1,$S2,100`	$S1=$S2|100	Bitwise OR with immediate value
not	`not $S1,$S2`	$S1=($S2 complement)	Bitwise NOT (inverter)
xor	`xor $S1,$S2,$S3`	$S1=$S2 ⊗ $S3	Bitwise XOR
shift left logical	`sll $S1,$S2,10`	$S1=$S2<<10	Shift left by constant number of bits
shift right logical	`srl $S1,$S2,10`	$S1=$S2>>10	Shift right by constant number of bits
Arithmetic shift right	`sra $S1,$S2,10`	$S1=$S2>>10	Shift right by constant number of bits with sign extender

Data Transfer

Instruction	Example	Meaning	Comments
load word	`lw $S1,100($S2)`	$S1=Memory[$S2+100]	Copy from memory to register

store word	`sw $S1,100($S2)`	Memory[$S2+100]=$S1	Copy from register to memory
load address	`la $S1,label`	$S1=Address of label	*Pseudo-instruction* (provided by assembler, not processor!) Loads computed address of label (not its contents) into register
load immediate	`li $S1,100`	$S1=100	*Pseudo-instruction* (provided by assembler, not processor!) Loads immediate value into register
move	`move $S1,$S2`	$S1=$S2	*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 $S1,$S2,100`	if($S1==$S2) go to PC+4+100	Test if registers are equal
branch on not equal	`bne $S1,$S2,100`	if($S1!=$S2) go to PC+4+100	Test if registers are not equal
branch on greater than	`bgt $S1,$S2,100`	if($S1>$S2) go to PC+4+100	*Pseduo-instruction*
branch on greater than or equal	`bge $S1,$S2,100`	if($S1>=$S2) go to PC+4+100	*Pseduo-instruction*

branch on less than	`blt $S1,$S2,100`	if($S1<$S2) go to PC+4+100	*Pseduo-instruction*
branch on less than or equal	`ble $S1,$S2,100`	if($S1<=$S2) go to PC+4+100	*Pseduo-instruction*

Note: It is much easier to use a label for the branch instructions instead of an absolute number. For example: `beq $St0, $St1, equal`. The label "equal" should be defined somewhere else in the code.

Comparison

Instruction	Example	Meaning	Comments
set on less than	`slt $S1,$S2,$S3`	if($S2<$S3)$S1=1; else $S1=0	Test if less than. If true, set $S1 to 1. Otherwise, set $S1 to 0.
set on less than immediate	`slti $S1,$S2,100`	if($S2<100)$S1=1; else $S1=0	Test if less than. If true, set $S1 to 1. Otherwise, set $S1 to 0.

Unconditional Jump

Instruction	Example	Meaning	Comments
Jump	`j 1000`	go to address 1000	Jump to target address
jump register	`jr $S1`	go to address stored in $S1	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:

In our module for now, we are only interested in the syscall to exit the program.

Service	Operation	Code (in $Sv0)	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 $Sf0
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 (For extra knowledge):

- 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

Assembler directives are directions to the assembler to take some action or change a setting. Assembler directives do not represent instructions, and are not translated into machine code.

For this assembler, all directives begin with a “.”, and the directive must exist on a separate line from any other assembler directive or assembler instruction.

An assembler directive also 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