**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 or equal** | bge $1,$2,100 | if($1>=$2) go to PC+4+100 | *Pseduo-instruction* |
| **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](https://ecs-network.serv.pacific.edu/ecpe-170/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](https://ecs-network.serv.pacific.edu/ecpe-170/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 |