**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 |

**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 |

**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 |

**J -- *Jump***

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

**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 |

**JR -- *Jump register***

|  |  |
| --- | --- |
| Description: | Jump to the address contained in register $s |
| Operation: | PC = nPC; nPC = $s; |
| Syntax: | jr $s |

**MULT -- *Multiply***

|  |  |
| --- | --- |
| Description: | Multiplies $s by $t and stores the result in $LO. |
| Operation: | $LO = $s \* $t; advance\_pc (4); |
| Syntax: | mult $s, $t |

**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 |

**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) |

**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) |

**BEQ -- *Branch on equal – BGT – Branch if greater than --- BLT – Branch if less than***

|  |  |
| --- | --- |
| 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 |

**Move** destination, what to move

**li** $t1,100 Load Immediate : Set $t1 to unsigned 16-bit immediate (zero-extended)

**.data directives**

* .asciiz Store the string in the Data segment and add null terminator
* .ascii Store the string in the Data segment but do not add null terminator
* .word Store the listed value(s) as 32 bit words on word boundary

How to do 8 bit two’s compliment.

Ex: -47

Step 1: Make the positive 00101111(+ 47)

Step 2: Compliment each bit (11010000)

Step 3: Add 1 (11010001)

Sixteen bit addition:

Ex:

1184

+ FF1C

---------

10A0

**True:** Two’s complement performs binary arithmetic on signed numbers just like unsigned arithmetic does with unsigned numbers. (The bits are the same)

**False:** In two’s complement, the maximum negative number equals the negative of the maximum positive number (The max negative is one greater)

**False:** In two’s complement, there are as many positive numbers as negative numbers (actually one less)

#### Big Endian --- STORING ~ 90 |AB | 12 | CD

|  |  |
| --- | --- |
| Address | Value |
| 1000 | 90 |
| 1001 | AB |
| 1002 | 12 |
| 1003 | CD |

In big endian, you store the most significant byte in the smallest address. Here's how it would look:

#### Little Endian

|  |  |
| --- | --- |
| Address | Value |
| 1000 | CD |
| 1001 | 12 |
| 1002 | AB |
| 1003 | 90 |

In little endian, you store the *least* significant byte in the smallest address. Here's how it would look:

Ex. program max that take two numbers in $a0 and $a1 and returns the max value in $v0

Max: move $v0,$a0 #assume $a0 is greater

blt $a0, $a1, a1\_greater # if $a0 < $a1, then return $a1 #if $a0 > $a1, then return $a1

jr $ra # jump to $ra (whoever called max)

a1\_ greater: move $v0, $a1 # $a1 was the max. Return it.

jr $ra # jump to $ra (whoever called a1\_greater)

Ex. Function sum that return the sum of all values in a word sized array the arguments passed to sum will be the size in words of the array in $a0, and a pointer of the array in #a1. Return the sum in $v0

Sum: li $v0, 0 #initialize sum to zero

ble $a0, $0, done #if size of array is 0 or less

top\_of\_loop: lw $t0, ($a1) #get current array element

add $v0, $v0, $t0 #add current element to running sum

add $a1, $a1,4 # print to next array element

add $a0, $a0, -1 #decrement counter of elements remaining

bgt $a0,$0,top\_of\_loop #loop back

done: jr $ra