4 | Assembly Language for MIPS (Pt. 2)

Immediate Instructions

One of the operands is in the instruction itself; think of it as a constant. Capabilities include (also known as **Immediate (I)** format instructions, not to be confused with the previous **Register (R)** format instructions):

- Add a constant value to a register
- Put a constant value into a register (other than 0)
- Move a value from one register into another register
- Perform a logical operation for which one of the operands is a constant (ex. AND, etc.)
- Move a number from RAM into a register
- Move a number from a register into RAM

There is a limitation; there can only be 16 bits (aka range of 0x8000 through 0x7fff or in integers, -32768 to 32767).

Add immediate operation:	**addi**	**$rsd, $rs, imm**	**# $rd ← $rs + imm**
Example:	addi	$t0, $a0, 17	# $t0 ← $a0 + 17

Load immediate operation: This is a pseudo operation; the assembler hands it for you, and translates it into the actual instruction that can be translated. (imm = decimal number)

	li	**$rd, imm**	**# $rd ← imm**
Example:	li	#v0, 35	# $v0 = 35

This example is translated by the assembler into: addi $v0, $0, 35

Move immediate operation: This is also a pseudo operation (not in the MIPS instruction set).

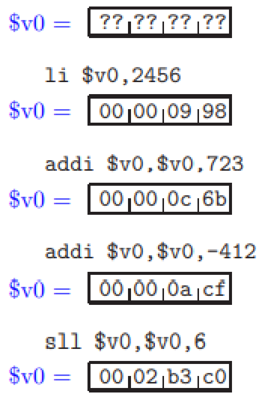
	mov	**$rd, $rs**	**# $rd ← $rs**
Example:	mov (use move in mars)	$t0, $a0	# $t0 = $a0

This example is translated by the assembler into: add $t0, $a0, $0

Example of a program: Calculate (2456 + 723 - 412) * 64.

This can be done with 4 immediate instructions and one shift instruction:

li	$v0, 2456	# $v0 = 2456
addi	$v0, $v0, 723	# $v0 = 2456 + 723
addi	$v0, $v0, -412	# $v0 = 2456 + 723 - 412
sll	$v0, $v0, 6	# $v0 = (2456 + 723 - 412) * 64
		# since 2^6 is 64, 6 is parameter



```
$v0 = ??|??|??|??
  li $v0,2456
$v0 = 00|00|09|98
  addi $v0,$v0,723
$v0 = 00|00|0c|6b
  addi $v0,$v0,-412
$v0 = 00|00|0a|cf
  sll $v0,$v0,6
$v0 = 00|02|b3|c0
```

The picture to the right is the trace of the mentioned program, where the values are in hexadecimal.

Note: There is no "subtract" immediate instruction; just use addi and a negative number as the "imm" parameter.

Logical Immediate Instructions

Logical immediate instructions are the same as logical instructions, but note that these have a 3rd operand, a constant, imm. This constant is represented by a 32-bit value, where 0's are extended to the highest order bit (leftmost) and the imm's are for the lower bits.

OR immediate:	**ori**	**$rd, $rs, imm**	**# $rd ← $rs $\vee$ imm**
AND immediate:	**andi**	**$rd, $rs, imm**	**# $rd ← $rs $\wedge$ imm**
XOR immediate:	**xori**	**$rd, $rs, imm**	**# $rd ← $rs $\oplus$ imm**

Example of a program: Clear the high order 16 bits of register $a0 and complement the low order 16 bits of that register.

```
andi    $a0, $a0, 0xffff        # Clear high order 16 bits
xori    $a0, $a0, 0xffff        # Complement low order 16 bits
```

Notice the immediate operand is being used as a mask. For andi, the mask's low order 16 bits are all represented by 1's (thanks to ffff), and the highest order bits are all 0's.

Load upper immediate instruction: Load the higher order bits from the immediate operand and clear the low order bits of the destination register.

	lui	**$rt, imm**	**# $\$rt_{16...31}$ ← imm, $\$rt_{0...15}$ ← 0**
Example:	lui	$t0, 17	# $t0 = 0x00110000
			# 17 is 0011_{16}, low bits all 0's

Example: Putting the constant 0x203a40bd into register $v0

```
lui     $v0, 0x203a     # High order 16 bits
xori    $v0, 0x40bd     # Low order 16 bits
```

Memory Reference Instructions

We're talking about transferring data between memory (RAM) and the CPU (registers).

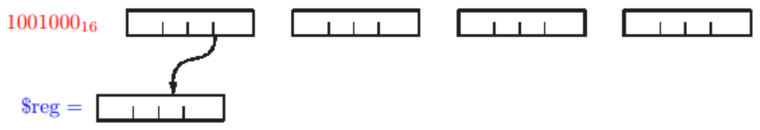
- A **word** is a 32-bit (4-byte) unit of data
- In assembly language, a word of memory is accessed by the word's *label* or by an explicit address

- **Load:** Transfer a word from memory to a register
- **Store:** Transfer a word from register to memory

Load word instruction: Loads a word from memory into a register; the full word of memory is moved into the specified register. It doesn't change that word at all, it just overwrites what was originally in the register.

lw **\$rt, label** **# \$rt ← memory**$_{\text{label}}$

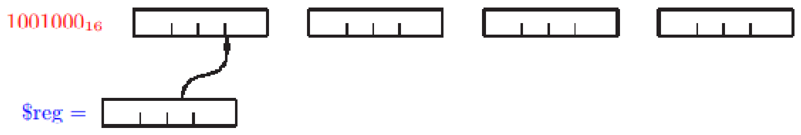
Example: lw \$t3, x # Register \$t3 is loaded from # memory word, *x*



Store word instruction: Stores contents of a register into a word of memory; the label indicates where the data is supposed to go. Take the contents of the register and put it into memory at the position indicated by the label.

sw **\$rt, label** **# \$rt → memory**$_{\text{label}}$

Example: sw \$t0, result # \$t0 is stored into result



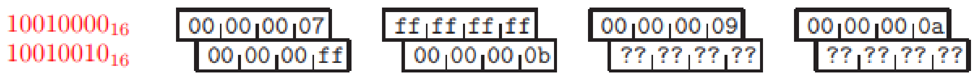
Symbolic memory addresses are symbols or labels that are used to identify particular words of memory. We can set up the data in assembly language using assembly directives:

.text	What follows are instructions, not data
.data	What follows are data, not instructions Format: The *s* stands for where that value is in memory.

.data
[label:]   .type   value(s)   [# comment]

Example: *x* is a label for the word of memory. The type represents that the amount of memory to be occupied is 1 full word. It's like saying int i = 7. Below is an image showing the data in memory.

```
            .data
x:          .word       7
y:          .word       -1
z:          .word       9, 10, 0xff     # 3 words; z refers to 9
            .word       11              # no label, so 11 follows 0xff
```



The data area begins at the hexadecimal address in red. The first box, left to right is *x*, the second box is *y*, the third is *z*, the next two are unlabelled, and the last is also unlabelled.

Example of a program: Compute result = (*x* - (*y* + *z*)) * 8 given (value of *x* is 77, etc., and here, we're storing result as a word without anything in it):

```
            .data
x:          .word       77
y:          .word       3
z:          .word       7
result:     .word

            .text                       # After this are instructions
            lw          $t0, y          # $t0 now contains 3
```

```
        lw      $t1, z              # $t1 now contains 7
        add     $t1, $t1, $t0       # $t1 = y + z
        lw      $t0, x
        sub     $t0, $t0, $t1       # $t0 = x - (y + z)
        sla     $t0, $t0, 3         # $t0 = (x - (y + z)) * 8
```

Example of a program: Add the 3 values beginning at start, putting the result at sum, given:

```
                .data
sum:            .word       0
start:          .word       3, 5, 13

                .text
                lw          $t0, start          # $t0 = 3
              * lw          $t1, start+4        # $t1 = 5
                add         $t1, $t0, $t1       # $t1 = 8
                lw          $t0, start+8        # $t0 = 13
                add         $t1, $t0, $t1       # $t1 = 21
                sw          $t1, sum            # sum = 21
```

** The symbolic label means that there are 4 bytes in a word, and start+4 means the word after the word "start" (which is the 3)*

Explicit (non-symbolic) memory addresses include registers, who may contain a memory address. Remember memory addresses are 32 bits, and registers store that amount.

Load address instruction: **la $rd, label** **$rd ← memory address of label**

Example:

```
                .data
x:              .word       7
y:              .word       3
z:              .word       4

## Assume data area begins at address 0x10010000
                .text
              * la          $t0, x          # $t0 = 0x10010000
             ** la          $t1, y          # $t1 = 0x10010004
                la          $t2, z          # $t2 = 0x10010008
            *** la          $t3, y+32       # $t3 = 0x10010024
```

** Since data area begins at the address, x has that address.*
*** Each word is 4 bytes, so the address increases by 4.*
**** This is 32 more bytes than y. 32, of course, in hexadecimal, is 20.*

Explicit memory references are where a memory reference may be the sum of a register plus a constant. Here's the format: **lw $rt, imm($rs)** **# $rt ← memory**$_{\text{imm + (\$rs)}}$

It means take a word of memory, and the word of memory is determined by the address, and you get the address by taking the contents of the register and adding the constant. That gives you a memory address, and whatever value at the memory address is loaded into $rt. This is NOT a symbolic address.

In order for this to work, $rs would have to be loaded with an address prior.

THIS IS NOT LOAD ADDRESS, KNOW THE DIFFERENCE!!

Example of a program: Sum the three words beginning at start (same assumption).

```
                  .data
sum:              .word      0
start:            .word      3, 5, 13          # Array

                  .text
                  la         $t0, start        # t0 = 0x10010004
                * lw         $t1, 0($t0)       # $t1 = 3
               ** lw         $t2, 4($t0)       # $t2 = 5
                  lw         $t3, 8($t0)       # $t3 = 13
                  add        $t1, $t1, $t2     # $t1 = 8
                  add        $t1, $t1, $t3     # $t1 = 21
                  sw         $t1, sum          # sum = 21
```

** Take what's in $t0, add the constant 0, and get address; value @ address is stored*
*** The constant is now 4 b/c it is the next word in the array*

Simple instructions on how to run and execute MIPS code on MARS!

1. Open a new file
2. Type your code, then save the file
3. Go to "Run", then "Assemble"
4. Click the green play button!

Printing out stuff: MIPS syscall functions available in MARS (missouristate.edu)