3 | Assembly Language for MIPS (Pt. 1)

In assembly language,

- Instead of using binary and hexadecimal representation, we'll be using **mnemonics**, or symbolic names, for operations (ex. add for addition, lw for load word, and j for jump)

- **Names for the CPU registers** - we have registers numbered from 0 to 31 (since there are 32), but we'll be using alternate names (ex. $t0, $a0, $sp)
    - Some registers have specific usages; these alt names will help us remember

- We also have **symbolic addresses** instead of hexadecimal/binary addresses
    - Identifiers for memory locations
    - Labels for instructions

- **Comments for documentation** are ignored at execution, not translated into machine language

- **Pseudo-ops and macros** for certain operations not supported by the MIP architecture; the assembler will translate them into actual MIPs instructions

Registers and Register Names

We know there are 32 general purpose registers in the CPU (each is 32 bits, word size, think of it as a Java int). Register name structure is $ followed by a number 0-31.

- Some registers *must* be used for designated purposes*
- Others *should* be used for designated purposes

Registers	**Alt Names**	**Usage**
$0	$zero	Always contains 0*
$1	$at	Reserved for the assembler* (pronounced "dollar A T")
$2 - $3	$v0 - $v1	Function return values
$4 - $7	$a0 - $a3	Function arguments

$8 - $15	$t0 - $t7	Temporary storage
$16 - $23	$s0 - $s7	Saved temporary storage
$24 - $25	$t8 - $t9	Temporary storage
$29	$sp	Stack pointer (used to implement functions)
$31	$ra	Return address*

Format of a Statement

All statements must be in this format (brackets indicate optionality):
[label:] operation operand[,operand,operand] [#comment]

- A statement label, followed by a colon (optional)
- An operation, such as 'add' (*mnemonic*)
- 1-3 operands, separated by commas (like registers or memory locations)
- Comment beginning with # (optional)

Examples:

```
done: add $t0, $t1, $zero
      jal   sqrt                # Jump to sqrt function
# This entire line is a comment, this won't get translated into machine language at all
```

Arithmetic Instructions

The concept here is that, pronounced "air-rith-metic", it is an adjective. **Arithmetic instructions** are instructions that perform simple arithmetic operations such as add, etc.

For example, the **add** instruction:

- Requires three operands:
  - Register storing the sum ($rd)
  - Register storing one of the values to be added ($rs)
  - Register storing the other value to be added ($rt)
  - *Note that these names are placeholders to represent* some *register*

- <u>Ex:</u>   add $s0, $t3, $a0   # $s0 = $t3 + $a0
    - Means take what's in register $t3, add to what's in $a3, then store that sum in $s0
    - Explained in the comments
    - *Note that whatever was in $s0 previously is overwritten by the sum calculated*

- <u>Ex:</u>   add $t0, $t0, $t0   # $t0 = $t0 + $t0
    - This is essentially doubling the value in $t0 and storing it in that
    - The format is **add $rd, $rs, $rt   # $rd = $rs + $rt**

For the **[subtraction]** instruction, it's similar:

- Requires three operands:
    - Register storing the difference ($rd)
    - Register storing the minuend ($rs) [the value getting subtracted from]
    - Register storing the subtrahend ($rt) [the value being subtracted]

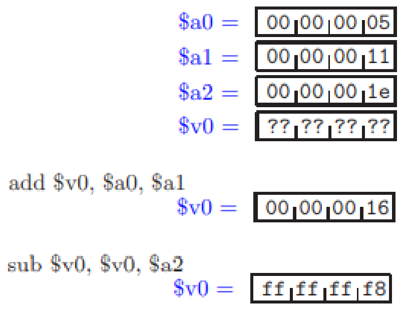
- <u>Ex:</u>   sub $s0, $t3, $a0   # $s0 = $t3 - $a0
    - Same comments apply as the add example, but with subtraction

- <u>Ex:</u>   sub $t0, $t0, $t0   # $t0 = 0
    - This is subtracting the value from itself, so $t0 in this case is always 0
    - Remember to be careful; flipping the $rs and $rt can change the result
    - Subtraction is not commutative
    - The format is **sub $rd, $rs, $rt   # $rd = $rs - $rt**

Let's do an example.

  Calculate 5 + 17 - 30.

  a. Assume register $a0 contains 5, $a1 contains 17, $a2 contains 30
  b. Remember these specific registers are for "*function arguments*" (top table)
  c. Let's leave the result in $v0 (this register is for "*function return values*")
  d. Firstly: add $v0, $a0, $a1   # $v0 = 5 + 17
  e. Then: sub $v0, $v0, $a2   # $v0 = 22 - 30



```
add $v0, $a0, $a1
```

```
sub $v0, $v0, $a2
```

f. Note the picture above; the values are stored as *hexadecimals*
   i. The first instance of $v0 has question marks
   ii. That's called "garbage" since it'll get overwritten anyway
   iii. If you're not convinced the last $v0 isn't -8, add 8 to it and you get 0
   iv. (After throwing the carry away, of course)

Next, the **load immediate** instruction (li) is used to put a value into a register. The format is **li $rd, constant**. The constant should be a number; we're setting ("loading") the register equal to the constant value specified.

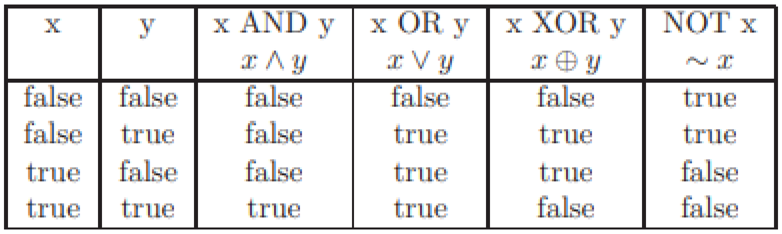
- Ex. li $t0, 37 # $t0 = 37
  - Register $t0 now contains: 00000025_x
  - *Note that this is 8 digits, 4 bytes, 32 bits, that's a full word register*

Lastly, the **set if less than** instruction is used by the assembler in conditional branches (Chapter 4). It's really "set if *only* less than". The format is **slt $rd, $rs, $rt**. How it works is that it makes the comparison: *If $rs < $rt, then $rd = 1; else $rd = 0 (clears the register)*

- Ex:
```
li      $t0, -19        # $t0 = -19
li      $t1, 5          # $t1 = 5
slt     $v0, $t0, $t1   # $v0 = 1
```
  - First, setting the variables $t0 and $t1 equal to certain constants
  - Makes the $v0 equal to 1 since $t0 < $t1 (it knows negative numbers are less)

- Also note that this is an *arithmetic comparison*, meaning that the operands are assumed to be two's complement representations (that negative numbers are smaller than positive). The command **sltu** performs *unsigned comparison*.

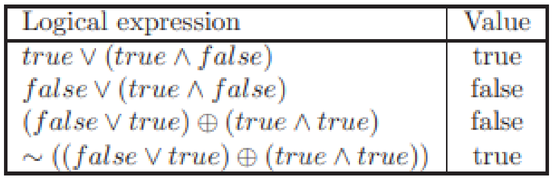
Logical Instructions



x	y	x AND y $x \wedge y$	x OR y $x \vee y$	x XOR y $x \oplus y$	NOT x $\sim x$
false	false	false	false	false	true
false	true	false	true	true	true
true	false	false	true	true	false
true	true	true	true	false	false

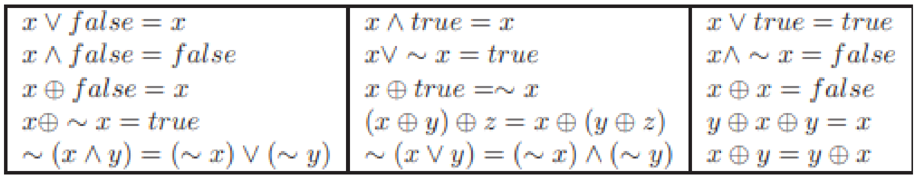
Logical operations are operations for which each operand must have one of the values: true or false. The result also must be a boolean value. In digital logic, **1 = true, 0 = false**. See the table above. Note that XOR means "EXCLUSIVE OR" and the NOT means "COMPLEMENT" basically.

See the below table for some examples of logical expressions and identities. Remember, in the identities, the *x* represents a boolean value, or a multiple bit thing.



Logical expression	Value
$true \vee (true \wedge false)$	true
$false \vee (true \wedge false)$	false
$(false \vee true) \oplus (true \wedge true)$	false
$\sim ((false \vee true) \oplus (true \wedge true))$	true

Figure 3.9: Examples of logical expressions



$x \vee false = x$	$x \wedge true = x$	$x \vee true = true$
$x \wedge false = false$	$x \vee \sim x = true$	$x \wedge \sim x = false$
$x \oplus false = x$	$x \oplus true = \sim x$	$x \oplus x = false$
$x \oplus \sim x = true$	$(x \oplus y) \oplus z = x \oplus (y \oplus z)$	$y \oplus x \oplus y = x$
$\sim (x \wedge y) = (\sim x) \vee (\sim y)$	$\sim (x \vee y) = (\sim x) \wedge (\sim y)$	$x \oplus y = y \oplus x$

Figure 3.10: Some logical identities

The first two identities in the last row of the table are known as **deMorgan's Laws**. The first two identities in the third row of the table are used extensively in cryptography. Each of these identities can be proved with a simple truth table.

Let's talk about bitwise operations.

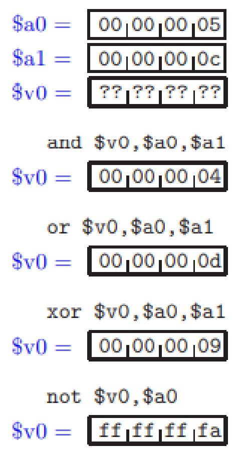
- If doing bitwise AND ($\wedge$) operation, we're doing 4 bit operation on a 4 bit word (when doing 0011 $\wedge$ 1010 for example, which results in 0010).
- If doing bitwise OR ($\vee$) operation, it would look similar (0011 $\vee$ 1010 = 1011).
- If doing bitwise XOR ($\oplus$) operation, it would look differently (0011 $\oplus$ 1010 = 1001).
- If doing bitwise NOT (~) operation, it would use only one operand (here, ~1010 = 0101).

MIPS architecture provides instructions for bitwise logical operations:

- For AND: **and** **\$rd, \$rs, \$rt** **# \$rd ← \$rs $\wedge$ \$rt**
  This would represent 32 AND operations cuz there are 32 bits in a register.
- For OR: **or** **\$rd, \$rs, \$rt** **# \$rd ← \$rs $\vee$ \$rt**
  This is 32 OR operations (reasoning stated above).
- For XOR: **xor** **\$rd, \$rs, \$rt** **# \$rd ← \$rs $\oplus$ \$rt**
  This is 32 XOR operations (reasoning stated above).
- For NOT: **not** **\$rd, \$rs** **# \$rd ← ~ \$rs**

This is 32 NOT operations (reasoning stated above).
This is also called one's complement.
Not an instruction in the MIPS set, but the assembler lets you use this anyway by substituting with its own instructions.

Take a look at this example of the bitwise operations being used. Note for the NOT operation, there are multiple "ff"s since there are 32 bits, and $a0 contains many 4 bit words "0"s (each number there is 4 bits each).



```
$a0 = 00|00|00|05
$a1 = 00|00|00|0c
$v0 = ??|??|??|??

      and $v0,$a0,$a1
$v0 = 00|00|00|04

      or $v0,$a0,$a1
$v0 = 00|00|00|0d

      xor $v0,$a0,$a1
$v0 = 00|00|00|09

      not $v0,$a0
$v0 = ff|ff|ff|fa
```

Figure 3.13: Examples of Logical Instructions

1) 0101 = 5
$\wedge$ 1100 = c
0100 = 4

2) 0101 = 5
$\vee$ 1101 = c
1101 = d

3) 0101 = 5
$\oplus$ 1100 = c
1001 = 9

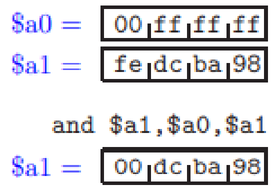
4) ~ 0000 0101 = 5
1111 1010 = fa

Applications of logical instructions:

- **Masks:** Used to select some bits in the register (change/sense, unchanged/unsensed)
- AND can be used to clear selected bits (properties used are $x \wedge 0 = 0$ and $x \wedge 1 = x$)
    - The notation used in the 0x0… shows that when writing a constant in assembly language, it doesn't have to be a decimal constant
    - if you precede it with 0x, it's a hexadecimal constant

Example: Clear the high order 8 bits of register $a1.

```
li      $a0, 0x00ffffff     # Mask
and     $a1, $a0, $a1       # Clear up high order 8 bits
```

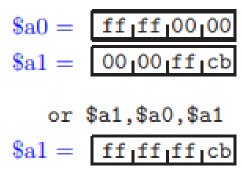


```
$a0 = 00|ff|ff|ff
$a1 = fe|dc|ba|98

      and $a1,$a0,$a1
$a1 = 00|dc|ba|98
```

This example is demonstrated to the right. Whenever you AND a 0 with something, you get a 0 result, as shown. The last 24 bits are all 1111 (1's), so they will leave the rest of the bits unchanged.

- OR can be used to set or sense selected bits (properties used are $x \vee 1 = 1$ and $x \vee 0 = x$)

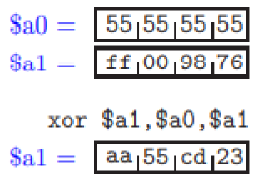
<u>Example:</u> Set the high order 16 bits of register $a1 (meaning set the specified bits to 1's).



```
li      $a0, 0xffff0000     # Mask
or      $a1, $a0, $a1       # Set high order 16 bits
```

- XOR can be used to complement selected bits (properties used here are $x \oplus 1 = {\sim}x$ and $x \oplus 0 = x$)

<u>Example:</u> Complement alternate bits of register $a1.

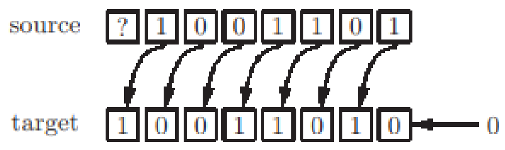


li $a0, 0x55555555   # Mask: $5_x = 0101_2$
xor $a1, $a0, $a1   # Complement alternate bits

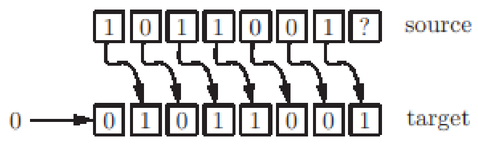
Shift Instructions

Shift instructions shift bits in a register: The process of moving each bit in a word to its right or left neighbor; with all operations, there is a source register and a target register. The target receives the result, the source is unchanged.

- Bits can be shifted in either direction
- Bits shifted off the end are lost
- To replace them, bits shifted in from the other end
    - Shift in zeros
    - Shift the sign bit, preserve the sign of an integer on a right shift?
- Number of bits can be specified in the instruction

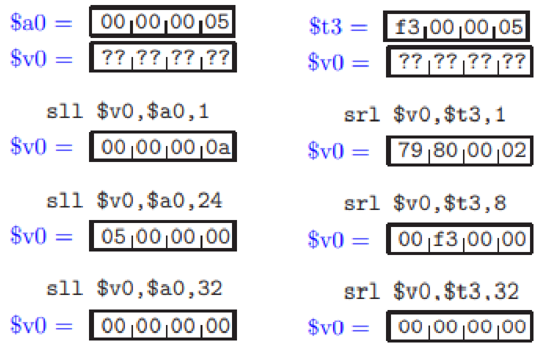


<u>Logical shift instructions:</u> The sign bit is not preserved; for example, shifting one bit to the left (see right figure). We will always be shifting in a 0 to the low order position when we do a left shift. These are on unsigned quantities. The format of shift left logical instruction: **sll $rd, $rs, shiftAmt**



Meanwhile, for a right shift, we'll be shifting a 0 into the high order position always, as demonstrated on the right. The format is **srl $rd, $rs, shiftAmt**.

Remember, when doing these shifts, the *source register does not get changed!* Note the examples to the right about some left shifts!

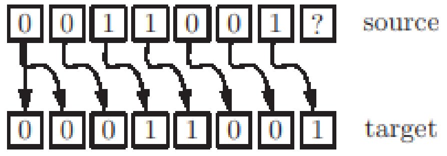


```
$a0 = 00 00 00 05        $t3 = f3 00 00 05
$v0 = ?? ?? ?? ??        $v0 = ?? ?? ?? ??

   sll $v0,$a0,1            srl $v0,$t3,1
$v0 = 00 00 00 0a        $v0 = 79 80 00 02

   sll $v0,$a0,24           srl $v0,$t3,8
$v0 = 05 00 00 00        $v0 = 00 f3 00 00

   sll $v0,$a0,32           srl $v0,$t3,32
$v0 = 00 00 00 00        $v0 = 00 00 00 00
```

For the part where they're only moving one position, 5 = 0101_2. If we move it by 1, we would get the binary number 1010_2 (the 0 in the highest order position in the 5 gets moved along with the rest of the 0's), and that is also equivalent to "a" in hexadecimal.

The same goes for the furthest right image for right shifts above.

Arithmetic shift instructions: Shifting right where the sign bit is preserved; an example is shifting one bit to the right, as shown in the right picture.



Note that the highest order bit is moved to the right BUT ALSO is copied to the highest order position as well (not shifting a 0 in). This will preserve the signs (remember the highest order bit is how we know if it's positive or negative). The format is **sra $rd, $rs, shiftAmt**.

Applications of shift instructions:

- Clear a register
- Fast way to multiply by a power of 2 (**sll** or **sla**)
  - $15 * 8 = 15 * 2^3$
  - Shift left 3 bits
  - Same way in binary!
  - $1111_2 * 1000_2 = 01111000_2 = 78_{16} = 120$
- Fast way to divide by a power of 2 (**srl** or **sra**)
  - $15 / 8 = 15 / 2^3$
  - Shift right 3 bits
  - $1111_2 / 1000_2 = 0001_2$
- To do these operations unsigned, use logical operations (arithmetic preserves the sign)