Lecture 13

Logical Operations

- **<u>New Perspective</u>:** View the contents of a register as 32 raw bits rather than as a single 32-bit number.
- Since registers are composed of 32 bits, we may want to access individual bits (or groups of bits) rather than the whole 32 bits.
- Two new classes of instructions: *Logical Operators* and *Shift Instructions*.
  - Essential for operations manipulating bits within a word.
    For example, characters within a word, each of which is 8 bits.
  - There are specific instructions in MIPS to simplify the pack/unpacking of bits into words.
    Also known as *bitwise operations*.

<table>
<tr><th>Logical ops</th><th>C operators</th><th>Java operators</th><th>MIPS instr</th></tr>
<tr><td>Shift Left</td><td><<</td><td><<</td><td>sll</td></tr>
<tr><td>Shift Right</td><td>>></td><td>>>></td><td>srl</td></tr>
<tr><td>Bit-by-bit AND</td><td>&</td><td>&</td><td>and, andi</td></tr>
<tr><td>Bit-by-bit OR</td><td>|</td><td>|</td><td>or, ori</td></tr>
<tr><td>Bit-by-bit NOT</td><td>~</td><td>~</td><td>nor</td></tr>
</table>

Other logical bitwise operations are: NOR, XOR, NAND, *etc*.

- Two basic logical operators:
  - <u>AND:</u> outputs 1 only if both inputs are 1.
  - <u>OR:</u> outputs 1 if at least one input is 1.
  - In general, they can be defined to accept > 2 inputs, but in the case of MIPS assembly, both of these accept exactly 2 inputs and produce 1 output.
    - Again, rigid syntax ⟶ simpler hardware.
- Bitwise operations, therefore, operate on each bit individually.
  - <u>Ex:</u> `$t3` = `$t1` AND `$t2`
    - Each bit of `$t1` and `$t2` is operated on with logical AND, and the resulting bits are stored in the corresponding positions of `$t3`.
  - OR works similarly.
  - NOT inverts the bits of a register. It is a pseudo instruction in MIPS.

- Note that ANDing a bit with 0 produces a 0 at the output, while ANDing a bit with 1 produces the original bit.
  - This can be used to create a *mask*.
  - Example:

```
        1011 0110 1010 0100 0011 | 1101 1001 1010
Mask:   0000 0000 0000 0000 0000 | 1111 1111 1111
```

  - The result of ANDing these:

```
        0000 0000 0000 0000 0000 | 1101 1001 1010
```

  - The second bit-string in the example above is called a *mask*. It is used to isolate the rightmost 12 bits of the first bit-string by masking out the rest of the string (*i.e.*, setting it to all 0s).
  - Thus, the *and* operator can be used to set certain portions of a bit-string to 0s, while leaving the rest alone.
  - In particular, if the first bit-string in were in `$t0`, then the following instruction would mask it as per the example (zero out the left 20 bits, and reproduce the pattern of the right 12 bits):

```
andi    $t0,$t0,0xFFF
```

- Similarly, note that ORing a bit with 1 produces a 1 at the output, while ORing a bit with 0 produces the original bit.
  - This can be used to force certain bits of a string to 1s.
  - For example, suppose `$t0` contains 0x12345678, and consider the following instruction:

```
ori  $t0, $t0, 0xFFFF
```

  - As a result, `$t0` will contain 0x1234FFFF (*i.e.*, the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).

- Examples of logical and shift instructions and their formats in MIPS.
  - AND and NOR are logical instructions; the rest are *shift* instructions.
    - *shift*: move the bits in a register left or right.
    - *sign extension* is a right arithmetic shift (retains sign).
    - *logical shift* injects zeros.

Instruction	Example	
and	and \$t1,\$t2,\$t3	# \$t1 = \$t2 & \$t3
and immediate	andi \$t1,\$t2,8	# \$t1 = \$t2 & 8
or	or \$t1,\$t2,\$t3	# \$t1 = \$t2 | \$t3
or immediate	ori \$t1,\$t2,15	# \$t1 = \$t2 | 15
xor	xor \$t1,\$t2,\$t3	# \$t1 = \$t2 ⊕ \$t3
xor immediate	xori \$t1,\$t2,9	# \$t1 = \$t2 ⊕ 9
nor	nor \$t1,\$t2,\$t3	# \$t1 = ~(\$t2 | \$t3)
shift left logical	sll \$t1,\$t2,3	# \$t1 = \$t2 << 3
shift left logical by variable	sllv \$t1,\$t2,\$t3	# \$t1 = \$t2 << \$t3
shift right logical	srl \$t1,\$t2,10	# \$t1 = \$t2 >> 10
shift right logical by variable	srlv \$t1,\$t2,\$t3	# \$t1 = \$t2 >> \$t3
shift right arithmetic	sra \$t1,\$t2,6	# \$t1 = \$t2 >> 6 # sign extend
shift right arithmetic by variable	srav \$t1,\$t2,\$t3	# \$t1 = \$t2 >> \$t3 # sign extend

Shift operations

- Move (shift) all the bits in a word to the left or right by a given number of bits.
  - Example: shift right by 8 bits

```
srl $t1, $t2, 8   # $t1 ← $t2 >> 8
0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110
```

  - Example: shift left by 8 bits

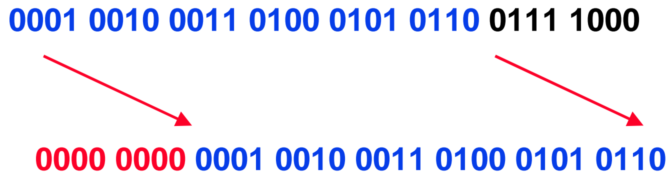
```
sll $t1, $t2, 8   # $t1 ← $t2 << 8
0001 0010 0011 0100 0101 0110 0111 1000

0011 0100 0101 0110 0111 1000 0000 0000
```

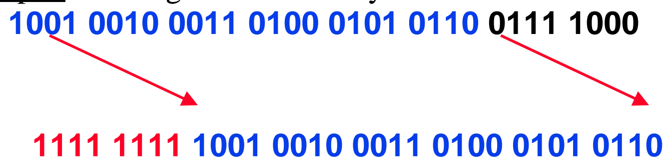
  - Shift right, logical: the rightmost bits are lost. The leftmost bits become 0 (called *zero fill*).
  - Shift left, logical: the leftmost bits are lost. The rightmost bits become 0.

- MIPS shift instructions:
    - sll (shift left logical): shifts left and fills emptied bits with 0s.
    - srl (shift right logical): shifts right and fills emptied bits with 0s.
    - sra (shift right arithmetic): shifts right and fills emptied bits by sign extending.

Example: shift right arithmetic by 8 bits



Example: shift right arithmetic by 8 bits



- Uses for Shift Instructions:
    - Suppose we want to isolate byte 0 (rightmost 8 bits) of a word in `$t0`. Simply use:

      ```
      andi $t0,$t0,0xFF
      ```

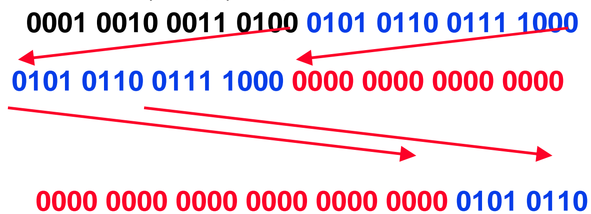
    - Suppose we want to isolate byte 1 (bit 8 to bit 15) of a word in `$t0`. We can use:

      ```
      andi $t0,$t0,0xFF00
      ```

      but then we still need to do a logical shift to the right by 8 bits (to get the byte we have isolated into byte 0 – the rightmost byte – of `$t0`, with the other three bytes all zero).

Could use instead:

```
sll   $t0,$t0,16
srl   $t0,$t0,24
```



- In decimal:
  - Multiplying by 10 is the same as shifting left by 1:
    - 714 x 10 = 7140
    - 56 x 10 = 560
  - Multiplying by 100 is same as shifting left by 2:
    - 714 x 100 = 71400
    - 56 x 100 = 5600
  - Multiplying by 10^n is same as shifting left by *n*.

- In binary:
  - Multiplying by 2 is same as shifting left by 1:
    - 11_2 x 10_2 = 110_2
    - 1010_2 x 10_2 = 10100_2
  - Multiplying by 4 is same as shifting left by 2:
    - 11_2 x 100_2 = 1100_2
    - 1010_2 x 100_2 = 101000_2
  - Multiplying by 2^n is same as shifting left by *n*.

- Since shifting may be faster than multiplication, a good compiler usually notices when *C* code multiplies by a power of 2 and compiles it to a shift instruction.

  a = a * 8;   (in *Java*, say)

  would compile to:

  `sll $s0,$s0,3`   (in MIPS)

- Likewise, shift right to do *integer division* by powers of 2.
  - Remember to use `sra`, not `srl`. Why?

- In addition to word data transfers (`lw`, `sw`), MIPS has byte data transfers:
  - load byte: `lb`.
  - store byte: `sb`.
    - same format as `lw`, `sw`.
  - What do with other 24 bits in the 32 bit register?
    - `lb`: sign extends to fill upper 24 bits.

- Normally with characters do not want to sign extend.
- MIPS instruction that does not sign extend when loading bytes:

  load byte unsigned: `lbu`.

<u>Rotate or Circular Shift</u>

- Bits are not lost when we rotate (*i.e.*, do a "circular shift").
- They wrap around and enter the register from the other end.
- These are pseudo-instructions:
  - `rol`: rotate left.
  - `ror`: rotate right.
  - <u>Ex:</u> `rol $t2, $t2, 4`
    - Rotate left bits of `$t2` by 4 positions:

      1101 0010 0011 0100 0101 0110 0111 1000

      0010 0011 0100 0101 0110 0111 1000 1101