The Basic Logical Operations

The three basic bit-wise logical operations are easily defined by tables:

A	NOT A
0	1
1	0

```
not  $t1, $t2
```

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

```
and  $t1, $t2, $t3
```

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

```
or   $t1, $t2, $t3
```

Bitwise NOT

The `not` operation simply flips the bits:

```
not  1011 0011  --->  0100 1100
```

The MIPS `not` instruction (pseudo-instruction, actually) simply flips the bits of the source register and stores them in the destination register.

Bitwise AND

The and operation yields 1 iff both the source bits are 1:

```
1101 1010 AND 1011 0011  --->  1001 0010
```

The MIPS and instruction simply ANDs the bits of the two source registers and stores the resulting bits in the destination register.

Bitwise OR

The `or` operation yields 1 unless both the source bits are 0:

```
1001 1010 OR 1011 0011  --->  1011 1011
```

The MIPS `and` instruction simply ORs the bits of the two source registers and stores the resulting bits in the destination register.

Why do we care?

We can use AND to extract any part of a bit sequence that we like:

```
x:  0101 1100 0000 1101 0011 0101 1010 0011
```

Suppose we want to extract the indicated bits.

Recall that for any value (0 or 1) of A, A and 0 equals 0 and A and 1 equals A.

So, the key to our problem is to create a suitable *mask*:

```
mask:  0000 0000 1111 1111 1111 1111 0000 0000
```

Note we put 1's where we want to capture bits and 0's where we want to annihilate them.

Now:

```
x and mask:  0000 0000 0000 1101 0011 0101 0000 0000
```

Of course, we might prefer to have the bits pushed to the right end (or the left)…

More Logical Operations

Some additional common logical operations are also supported:

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

```
xor  $t1, $t2, $t3
```

Note the similarity to not-equals!

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

```
nor  $t1, $t2, $t3
```

Why do we care?

Suppose we have the following values (8-bit for convenience):

```
x:  0101 1100        y:  1001 1000
```

Then:

```
       x:  0101 1100
       y:  1001 1000
x XOR y:  1100 0100  assign to x
y XOR x:  0101 1100  assign to y
x xor y:  1001 1000  assign to x
```

So? Take another look at the initial and final values of `x` and `y`.

Bitwise Shifts

There are three basic shift operations:

shift right logical:

```
srl  $t1, $t2, <imm>
```

```
      1101 0110   srl 1:   0110 1011
                  srl 3:   0001 1010
```

shift left logical:

```
sll  $t1, $t2, <imm>
```

```
      1101 0110   sll 1:   1010 1100
                  sll 3:   1011 0000
```

shift right arithmetic:

```
sra  $t1, $t2, <imm>
```

```
      1101 0110   sra 1:   1110 1011
                  sra 3:   1111 1010
```

Why do we care?

Suppose we have the following values (8-bit for convenience):

```
x:  0001 1100        y:  1001 1000
```

Then:

```
Note x == 28.
sll  x, x, 2  # x: 0111 0000  or 112 or 28 * 4
```

Be careful of overflow… what would `sll x, x, 4` yield?

Also:

```
Note y == -104.
sra  y, y, 4  # y: 1111 1001  or -7 or -104 / 16
```

What if we wanted to use a right-shift for division with an unsigned integer?

Why is there no `sla` (*shift left arithmetic*) instruction?

Other Bitwise Operations

There are also rotation instructions:

rotate left:

```
rol  $t1, $t2, <imm>
```

```
      1101 0110   rol 1:   1010 1101
                  rol 3:   1011 0110
```

rotate right:

```
ror  $t1, $t2, <imm>
```

```
      1101 0110   ror 1:   0110 1011
                  ror 3:   1101 1010
```