CSCI 260 Notes

Ralph Vente

Contents

2019 02 06

Section 2.3 cont.

Example

```
*p = arr[i] + y; // assume p is initialized properly
```

```
# i -> $s0, y -> $s1, p -> $s2, arr -> $s3
add $t3, $s0, $s0  # t3 <- 2i
add $t3, $t3, $t3  # t3 <- 2t3 = 4i
add $t3, $t3, $t3  # t3 <- address of arr[i]
lw $t3, 0($s3)     # t3 <- arr[i]
add $t3, $t3, $s1  # t3 <- arr[i] + y
sw $t3, 0($s2)     # *p <- arr[i] + y
```

Section 2.6

Logical Instructions

rd, rt, shamt

```
sll # shift left Logical
srl # shift right logical
```

rd, rs, rt

```
# rd <- rs op rt
and
or
nor
xor
```

rt, rs, imm

```
andi
ori
xori
```

`imm` needs to be converted from 16 bits to 32 bits. It uses 0-extended to do this.

Pseudo instructions

```
not rt, rs  # rt <- (rs)'
```

Pseudo instructions are kind of like a "mini-compiler" Don't use yet, may cause problems in the hardware side of things down the line.

Computers have 3 types of shifts: logical, arithmetic, and rotate.

Logical Shift L left <- o's, shift directly left

Arithmetic Shift L. same as logical shift, but does a special case for the sign bit

rotate shift left, moves all bits left moving the leftmost to the right (for the counter-clockwise leftward rotation)

Example: split `$s0` into 4 bytes assume `$s0` is 16 bits `$s1` is Most Significant Byte. `$s4` is Least Significant Byte

```
# bytes stored is $s1, $s2, $s3, $s4


add $t0, $s0, $zero  # t0 <- s0
andi $s4, $t0, 255   # s4 <- LSB of s0, 255 = 0x00FF (line 2)

srl $t0, $t0, 8     # t0 <- s0 >> 8
andi $s3, $t0, 255  # s3 <- 2nd LSB of s0
srl $t0, $t0, 8     # t0 <- s0 >> 16
andi $s2, $t0, 255  # s2 <- 2nd MSB of s0
srl $t0, $t0, 8     # t0 <- s0 >> 24
andi $s1, $t0, 255  # s1 <- MSB of s0
```

Line 2 explanation: Zero out all but the last 8 bits by `and`ing

```
  1100 1011 0010 1010 0011 0111 1110 1010 = $s0
& 0000 0000 0000 0000 0000 0000 1111 1111 = 0x00FF
-----------------------------------------
  0000 0000 0000 0000 0000 0000 1110 1010 -> $s4
```

0x00FF is operating as a bit mask. It *zero*s out all but the last 8 bits. What if we want to

Relevant identities for bit manipulation

Assume x is a bit.

```
x & 0 = 0
x & 1 = x
x | 1 = 1
x | 0 = x
x xor 0 = x
x xor 1 = ~x
```

$$x \wedge 0 = 0$$
$$x \wedge 1 = x$$
$$x \vee 1 = 1$$
$$x \oplus 0 = x$$
$$x \oplus 1 = \bar{x}$$

Ex: We want to do the following some bit manipulation. We can do this using the logical operations `ori`, `andi`, and `xori`. In this case, the `i` at the end of the command is short for `integer`. We're using the binary representations of the numbers as the underlying mechanism for the result.

1. set bits 3 & 8 of \$s5 (correspond to `0...0001 0000 1000`)
2. clear bits 2 & 11 of \$s5
3. flip bits 0, 1, 7 of \$s5
4. don't care for bits 31..16

```
ori $s5, $s5, 0x0108  # set bits 3, 8 of s5
andi $s5, s5, 0xF7FB  # clear bits 2, 4 of s5
xori $s5, $s5, 0x0083 # flip bits 0, 1, 7 of $s5
```

Use `xor` to flip bits. Or with 1 to set bit to 1.