Oliver Kim
Pulak Goyal

Scribe Notes 10/15/13

**Disassembly – figuring out how x86 works**

**f10.s**

type of f is a function, globally declared

**What does f do?**

Storing a = x + y

Function returns a

Prefixes for locations:

| | |
|---|---|
| $ | Immediate constant, i.e. the value 0 |
| % | Registers; the costliest, fastest form of memory |
| No prefix | Global |

x86 is a 2 address machine – most instructions refer to exactly 2 places

Basic commands:

| Command | Result |
|---|---|
| mov src, dst | dst = src |
| add src, dst | dst += src |

**What does g do?**

g is another function, subtracts y from x

| Command | Result |
|---|---|
| sub src, dst | dst -= src |

**What does h do?**

imull = integer multiplication

Can't distinguish between a function that returns a value and doesn't return a value

When a function ends, eax holds whatever's left over

**Suffixes**

- All our instructions used to end with an l, now they end with a w
- It's word as opposed to long
- Suffix for arithmetic instructions defines the size of data that's being manipulated
- Suffixes for sizes:
    - l for 32 bits
    - w for 16 bits
    - b for 8 bits (a byte)

**f13.s**

Interesting – why does `divl y` only have one argument?

- Division is one of the most expensive operations the processor can execute
- The x86 packs as much into its single division instruction as possible

Semantics of the division instruction:

`divl src` is the same as

```
uint 64.t x = %edx * 2^32 + %eax
%eax = x / src
%edx = x % src
xorl
```

Here, we are modifying `%eax`

**Bitwise operators**

| | |
|---|---|
| & | Bitwise and |
| \| | Bitwise or |
| ^ | Bitwise exclusive or ("xor") |
| ~ | Bitwise complement |

| & | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| ^ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

| \| | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

| ~ | |
|---|---|
| 0 | 1 |
| 0 | 0 |

`x^x = 0`

Quirk of the x86 architecture: instruction sequence for xor is only 1 byte long, while the explicit instruction is 5 bytes long

**f16.s**

moves eax into a global (sets a to –x)

**f17.s**

Sets `a` to `~x + 1`

What is x + ~x?

x is a 32-bit quantity, sums to 32 1s sum from i=0 to i=31 of $2^i = 2^{32} – 1$

`~x +1 =` $2^{32}$ `- x`

-x represented in the same way as 2^32 – x, thus –x is the same as ~x + 1

There are some patterns in the assembly that hint that there is a return value, the compiler will not calculate a value unless it has some use, which suggests that it will be returned

**What does f18.s do?**

Stores 1023 times the global value x in a, rare example of 3 address in instruction in x86

Shll is shift left – a << b means take the bit pattern

a31 a30 a 29 … a2 a1 a0

$a_{b+31}$ ab+30 ab+29 … ab+2 ab+1 ab 0 0 0 0 < b 0 bits, throw out first digits

**Example**

0001 << 1

0010 (2)

0101 << 2

0100 (4)

Left shift by 1 equivalent to multiplying by 2; left shift by *k* is equivalent to multiplying by $2^k$; right-shift is equivalent to division

For left shifts, you can throw away the sign bit and it doesn't matter. Take 1100 for example. When you shift left you get 1000 which is -8 (-4 * 2).

Shifting by bits is a lot cheaper than multiplication

For integers (32 bits), if the high bit is 1, the integer is negative

If the high bit is 0, the number is 0 or positive

The high bit is the sign bit; indicates if the number is negative or positive

In signed computer arithmetic, there are more negative numbers than there are positive numbers

Signed bit helps explain the difference between the sar and the shr instructions

4/4 = 1

0100 >> 2 = 0001

-4/4 = -1

1100 >> 2 = 1111 (arithmetic shift)

sar shifts to the right, but duplicates the signed bit in the new positions

x y x&y x&y <= x, x&y <= y

x y x|y x|y >= x, x|y >= y

x&(x-1), 0 if x = 2^i

not 0 if x != 2^i for any i

x&-x

0011 & 1101 = 0001

1101 (13) is -3 because 16-3 is 13

X = arbitrary string *a* 1 followed by k zeroes

~x = ~*a* 0 followed by k ones

-x = ~a 1 follwed by k zeroes