CSE 351

Midterm Review

Midterm Review

- Your midterm is next Wednesday
- Study past midterms (link on the website)
- Point of emphasis: **Registers are not memory**
  - Registers and caches are on the same piece of silicon as the processor
  - This is done to make things fast (farther away => takes more time)
  - Memory is an external storage bank of contiguous space
  - Registers are individual storage elements that are not "addressable"
    (i.e. there is no "offset" between %rax and %rdx)

Integer Representation

- Convert -39 and 99 to binary and add the two's complement 8-bit integers, then convert the result to hex

```
   -39 => 11011001
+   99 => 01100011
-------------------
    60 => 00111100

    60 => 0x3C
```

Integer Representation

- What is the difference between the carry flag (CF) and the overflow flag (OF)?
  - They differ in how they are triggered
  - CF is set during unsigned addition when a carry occurs at the most-significant bit
  - OF is set during signed arithmetic and it indicates that the addition yielded a number that was too large (either positive or negative direction)

Floating-Point Representation

- Suppose we have a 7-bit computer that uses IEEE floating-point arithmetic where a floating-point number has 1 sign bit, 3 exponent bits, and 3 fraction bits.

Number	Binary	Decimal
0	0 000 000	0.0
Smallest positive normalized number	0 001 000	0.25
Largest positive number < INF	0 110 111	15.0
-3.1	1 100 100	-3.0
12.25	0 110 100	12.0

Pointers

```
int a = 5, b = 15;
int *p1, *p2;

p1 = &a; // p1 -> a
p2 = &b; // p2 -> b
*p1 = 10; // a = 10
*p2 = *p1; // b = 10
p1 = p2; // p1 -> b
*p1 = 20; // b = 20
```

What are the values of a, b, p1, p2?

IA-32 vs. x86-64

Look at the following function:

```
int foo(int x, int y) {
      int c = x << (y + 3);
      if (x != 0) {
            return c;
      } else {
            return 1;
      }
}
```

What does this function look like in x86 assembly? 32-bit and 64-bit.

IA-32 version

```
push %ebp  // ??
mov    %esp, %ebp // ??
mov $0xc(%ebp), %ecx          // ??
add $0x3, %ecx
mov 0x8(%ebp), %eax  // ??
shl %ecx, %eax
cmp $0x8(%ebp), $0
jne $0x808472
mov $0x1, %eax
leave // ??
ret
```

This is what the code looks like in IA-32 assembly.

IA-32 version

```
push %ebp  // Save the base pointer
mov    %esp, %ebp // Move the base pointer up to the top
of the stack
mov $0xc(%ebp), %ecx          // Move y into a register
add $0x3, %ecx
mov 0x8(%ebp), %eax  // Move x into a register
shl %ecx, %eax
cmp $0x8(%ebp), $0
jne $0x808472
mov $0x1, %eax
leave // Restore the old base pointer
ret
```

This is what the code looks like in IA-32 assembly.

x86-64 version

```
push %ebp
mov %esp, %ebp
mov $0xc(%ebp), %ecx
add $0x3, %rsi
mov 0x8(%ebp), %eax
mov %rdi, %rax
shl %rsi, %rax
test %rdi,%rdi
jne $0x808472
mov $0x1, %rax
leave
ret
```

This is what the code looks like in x86-64 assembly. Differences?

Interpreting Assembly

Let `%eax` store x and `%ebx` store y. What does this compute?

```
  mov %ebx, %ecx // ??
  add %eax, %ebx // ??
  je .L1  // ??
  sub %eax, %ecx // ??
  je .L1 // ??
  xor %eax, %eax // ??
  jmp .L2 // ??
L1:
  mov $1, %eax // ??
L2 :
```

Interpreting Assembly

Let `%eax` store x and `%ebx` store y. What does this compute?

```
  mov %ebx, %ecx // %ecx = y
  add %eax, %ebx // ??
  je .L1  // ??
  sub %eax, %ecx // ??
  je .L1 // ??
  xor %eax, %eax // ??
  jmp .L2 // ??
L1:
  mov $1, %eax // ??
L2 :
```

Interpreting Assembly

Let `%eax` store x and `%ebx` store y. What does this compute?

```
  mov %ebx, %ecx // %ecx = y
  add %eax, %ebx // %ebx = x + y
  je .L1  // ??
  sub %eax, %ecx // ??
  je .L1 // ??
  xor %eax, %eax // ??
  jmp .L2 // ??
L1:
  mov $1, %eax // ??
L2 :
```

Interpreting Assembly

Let `%eax` store x and `%ebx` store y. What does this compute?

```
  mov %ebx, %ecx // %ecx = y
  add %eax, %ebx // %ebx = x + y
  je .L1  // jmp to L1 if x + y == 0
  sub %eax, %ecx // ??
  je .L1 // ??
  xor %eax, %eax // ??
  jmp .L2 // ??
L1:
  mov $1, %eax // ??
L2 :
```

Interpreting Assembly

Let `%eax` store x and `%ebx` store y. What does this compute?

```
  mov %ebx, %ecx // %ecx = y
  add %eax, %ebx // %ebx = x + y
  je .L1  // jmp to L1 if x + y == 0
  sub %eax, %ecx // %ecx = x - y
  je .L1 // ??
  xor %eax, %eax // ??
  jmp .L2 // ??
L1:
  mov $1, %eax // ??
L2 :
```

Interpreting Assembly

Let `%eax` store x and `%ebx` store y. What does this compute?

```
  mov %ebx, %ecx // %ecx = y
  add %eax, %ebx // %ebx = x + y
  je .L1  // jmp to L1 if x + y == 0
  sub %eax, %ecx // %ecx = x - y
  je .L1 // jmp to L1 if x – y == 0
  xor %eax, %eax // ??
  jmp .L2 // ??
L1:
  mov $1, %eax // ??
L2 :
```

Interpreting Assembly

Let `%eax` store x and `%ebx` store y. What does this compute?

```
  mov %ebx, %ecx // %ecx = y
  add %eax, %ebx // %ebx = x + y
  je .L1  // jmp to L1 if x + y == 0
  sub %eax, %ecx // %ecx = x - y
  je .L1 // jmp to L1 if x – y == 0
  xor %eax, %eax // %eax = 0
  jmp .L2 // ??
L1:
  mov $1, %eax // ??
L2 :
```

Interpreting Assembly

Let %eax store x and %ebx store y. What does this compute? **|x| == |y|**

```
  mov %ebx, %ecx // %ecx = y
  add %eax, %ebx // %ebx = x + y
  je .L1  // jmp to L1 if x + y == 0
  sub %eax, %ecx // %ecx = x - y
  je .L1 // jmp to L1 if x - y == 0
  xor %eax, %eax // %eax = 0
  jmp .L2 // return 0
L1:
  mov $1, %eax // return 1
L2 :
```