CSE 351

Midterm Review

Midterm Review

- Study past midterms (link on the website)
- Common point of confusion: **Registers vs. memory**
  - Registers are *named*, memory uses *addresses*
    - Can access *part* of a register with other names (%rax, %eax, %ax, %al)
  - Very few registers, lots of memory (big array of *bytes*)
  - Registers are *fast,* memory is much *slower*

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

- **Remember:** it's like *scientific notation* for binary
  - e.g. -1.01101 * 2^5
- 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;
p2 = &b;
*p1 = 10;
*p2 = *p1;
p1 = p2;
*p1 = 20;
```

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

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?

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

mov_ Interpretation Examples

```
movq %rax, (%rbx)
```
; move 8 byte value in %rax into memory starting at address %rbx

```
mov $0, %eax
```
; both operands are the same size, so we *infer* the instruction is really a movl

```
movw (%rax), %dx
```
; read 2 bytes starting at address %rax into %dx

```
movb (%rsp, %rdx, 4), %dl
```
; read 1 byte at address %rsp+%rdx*4 into %dl, (lowest byte of %rdx)

```
movzbq %al, %rbx
```
; moves byte %al into larger destination %rbx and zero extend

```
movl %eax, -12(%rsp)
```
; move 4 byte value stored in register %eax into memory starting at address
; %rsp - 12