Midterm Exam - Thurs Nov 10th, 7:30 - 9:30 pm

<table>
<thead>
<tr>
<th>If your Lecture number is</th>
<th>and the first letter of your family name is,</th>
<th>then, your assigned exam room is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>A-K</td>
<td>B130 Van Vleck</td>
</tr>
<tr>
<td>001</td>
<td>L-Z</td>
<td>B102 Van Vleck</td>
</tr>
<tr>
<td>002</td>
<td>A-R</td>
<td>B10 Ingraham</td>
</tr>
<tr>
<td>002</td>
<td>S-Z</td>
<td>19 Ingraham</td>
</tr>
</tbody>
</table>

- UW ID and #2 required
- closed book, no notes, no electronic devices (e.g., calculators, phones, watches)
  see “Midterm Exam 2” on course site Assignments for topics

Homework hw4: DUE on or before Monday, Nov 7
Homework hw5: will be DUE on or before Monday, Nov 14
Project p4A: DUE on or before Friday, Nov 4th
Project p4B: DUE on or before Friday, Nov 11th

Last Week

<table>
<thead>
<tr>
<th>Direct Mapped Caches - Restrictive</th>
<th>Writing to Caches (cont)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully Associative Caches - Unrestrictive</td>
<td>Cache Performance</td>
</tr>
<tr>
<td>Set Associative Caches - Sweet!</td>
<td>Impact of Stride</td>
</tr>
<tr>
<td>Replacement Policies</td>
<td>Memory Mountain</td>
</tr>
<tr>
<td>Writing to Caches</td>
<td>C, Assembly, &amp; Machine Code</td>
</tr>
</tbody>
</table>

This Week

<table>
<thead>
<tr>
<th>Low-level View of Data</th>
<th>Instructions - Arithmetic and Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Instructions - CMP and TEST, Condition</td>
</tr>
<tr>
<td>Operand Specifiers &amp; Practice L18-7</td>
<td>Codes</td>
</tr>
<tr>
<td>Instructions - MOV, PUSH, POP</td>
<td>Instructions - SET &amp; Jumps</td>
</tr>
<tr>
<td>Operand/Instruction Caveats</td>
<td>Encoding Targets &amp; Converting Loops</td>
</tr>
<tr>
<td>Instruction - LEAL</td>
<td></td>
</tr>
</tbody>
</table>

Next Week: Stack Frames and Exam 2
B&O 3.7 Intro - 3.7.5, 3.8 Array Allocation and Access
3.9 Heterogeneous Data Structures
C Function

```c
int accum = 0;
int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

Assembly (AT&T) Machine (hex)

```asm
sum:
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %eax
    addl 8(%ebp), %eax
    addl %eax, accum
    popl %ebp
    ret
```

→ What aspects of the machine does C hide from us?

Assembly (ASM)

→ What ISA (Instruction Set Architecture) are we studying?

→ What does assembly remove from C source?

→ Why Learn Assembly?
   1.
   2.
   3.

Machine Code (MC) is

→ How many bytes long is an IA-32 instruction?
Low-Level View of Data

C’s View

•

•

Machine’s View

Memory contains bits that do not

→ How does a machine know what it’s getting from memory?

1.

2.

Assembly Data Formats

<table>
<thead>
<tr>
<th>C</th>
<th>IA-32</th>
<th>Assembly Suffix</th>
<th>Size in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>double word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>long int</td>
<td>double word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>char*</td>
<td>double word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>single precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>double prec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>long double</td>
<td>extended prec</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In IA-32 a word
# Registers

## What? Registers

### General Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>bit 31</th>
<th>16 15</th>
<th>8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Program Counter

<table>
<thead>
<tr>
<th>Register</th>
<th>%eip</th>
</tr>
</thead>
</table>

### Condition Code Registers
Operand Specifiers

**What?** Operand specifiers are

- **S**
- **D**

**Why?**

**How?**

1. ) specifies an operand value that's
   
   **specifier**  **operand value**
   
   $l_{imm}$  $l_{imm}$

2. ) specifies an operand value that's
   
   **specifier**  **operand value**
   
   %E_a  R[%E_a]

3. ) specifies an operand value that's
   
   **specifier**  **operand value**  **effective address**  **addressing mode name**
   
   $l_{imm}$  M[EffAddr]  $l_{imm}$

   (%E_a)  M[EffAddr]  R[%E_a]

   $l_{imm}$(%E_b)  M[EffAddr]  $l_{imm}$+R[%E_b]

   (%E_b,%E_i)  M[EffAddr]  R[%E_b]+R[%E_i]

   $l_{imm}$(%E_b,%E_i)  M[EffAddr]  $l_{imm}$+R[%E_b]+R[%E_i]

   $l_{imm}$(%E_b,%E_i,s)M[EffAddr]  $l_{imm}$+R[%E_b]+R[%E_i]*s

   (%E_b,%E_i,s)  M[EffAddr]  R[%E_b]+R[%E_i]*s

   Imm,(%E_i,s)  M[EffAddr]  Im$m+R[%E_i]*s$

   ,(%E_i,s)  M[EffAddr]  R[%E_i]*s
Operands Practice

Given:

<table>
<thead>
<tr>
<th>Memory Addr</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x</td>
<td>%eax</td>
<td>0x</td>
</tr>
<tr>
<td>0x104</td>
<td>0x</td>
<td>%ecx</td>
<td>0x</td>
</tr>
<tr>
<td>0x108</td>
<td>0x</td>
<td>%edx</td>
<td>0x</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td>0x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What is the value being accessed? Also identify the type of operand, and for memory types name the addressing mode and determine the effective address.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Value</th>
<th>Type:Mode</th>
<th>Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. (%eax)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. 0xF8(,%ecx,8)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. %edx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. $0x108</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. -4(%eax)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. 4(%eax,%edx,2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. (%eax,%edx,2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. 0x108</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9. 259(%ecx,%edx)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instructions - MOV, PUSH, POP

What? These are instructions to

Why?

How?

<table>
<thead>
<tr>
<th>instruction class</th>
<th>operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV S, D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVS S, D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVZ S, D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushl S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>popl D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Practice with Data Formats

What data format suffix should replace the _ given the registers used?

1. mov_  %eax, %esp
2. push_  $0xFF
3. mov_  (%eax), %dx
4. mov_  (%esp, %edx, 4), %dh
5. mov_  0x800AFFE7, %bl
6. mov_  %dx, (%eax)
7. pop_  %edi

* Focus on register type operands
Operand/Instruction Caveats

Missing Combination?
→ Identify each source and destination operand type combinations.

1. movl $0xABCD,%ecx
2. movb $11,(%ebp)
3. movb %ah,%dl
4. movl %eax,-12(%esp)
5. movb (%ebx,%ecx,2),%al

→ What combination is missing?

Instruction Oops!
→ What is wrong with each instruction below?

1. movl %bl,(%ebp)

2. movl %ebx,$0xA1FF

3. movw %dx,%eax

4. movb $0x11,(%ax)

5. movw (%eax),(%ebx,%esi)

6. movb %sh, %bl
Instruction - LEAL

Load Effective Address

\[
\text{leal } S, D \quad D \leftarrow &S
\]

LEAL vs. MOV

```
struct Point {
    int x;
    int y;
} points[3];

int y = points[i].y; // mov 4(%ebx,%ecx,8),%eax
points[1].y;
```

```
int *py = &points[i].y; // leal 4(%ebx,%ecx,8),%eax
```

LEAL Simple Math

```
leal -3(%ebx), %eax  // subl $3, %ebx
movl %ebx, %eax
```

Suppose register %eax holds x and %ecx holds y. What value in terms of x and y is stored in %ebx for each instruction below?

1. \text{leal } (%eax, %ecx, 0), %ebx
2. \text{leal } 12(%eax, %eax, 4), %ebx
3. \text{leal } 11(%ecx), %ebx
4. \text{leal } 9(%eax, %ecx, 4), %ebx
Instructions - Arithmetic and Shift

Unary Operations

INC D \( D \leftarrow D + 1 \)
DEC D \( D \leftarrow D - 1 \)
NEG D \( D \leftarrow -D \)
NOT D \( D \leftarrow \sim D \)

Binary Operations

ADD S,D \( D \leftarrow D + S \)
SUB S,D \( D \leftarrow D - S \)
IMUL S,D \( D \leftarrow D \times S \)
XOR S,D \( D \leftarrow D \oplus S \)
OR S,D \( D \leftarrow D \mid S \)
AND S,D \( D \leftarrow D \& S \)

Given:

0x100 \%eax
0x104 \%ecx
0x108 \%edx

→ What is the destination and result for each? (do each independently)

1. incl 4(%eax)
2. addl %ecx, (%eax)
3. addl $32, (%eax, %edx, 4)
4. subl %edx, 0x104

Shift Operations

logical shift

\[ \text{SHL} \ k, \ D \leftarrow D \ll K \]
\[ \text{SHR} \ k, \ D \leftarrow D \gg K \]

arithmetic shift

\[ \text{SAL} \ k, \ D \leftarrow D \ll K \]
\[ \text{SAR} \ k, \ D \leftarrow D \gg K \]
Instructions - CMP and TEST, Condition Codes

What?

Why?

How?

\[ \text{CMP } S2, S1 \quad \text{CC} \leftarrow S1 - S2 \]

\[ \text{TEST } S2, S1 \quad \text{CC} \leftarrow S1 \& S2 \]

➢ What is done by \text{testl } %eax, %eax

Condition Codes (CC)

- ZF: zero flag
- CF: carry flag
- SF: sign flag
- OF: overflow flag
Instructions - SET

What?

set a byte register to 1 if a condition is true, 0 if false
specific condition is determined from CCs

How?

sete D  D <-- ZF
setne D  D <-- ~ZF
sets D  D <-- SF
setns D  D <-- ~SF

Unsigned Comparisons:

setb D  D <-- CF
setbe D  D <-- CF | ZF
seta D  D <-- ~CF & ~ZF
setae D  D <-- ~CF

Signed (2's Complement) Comparisons

setl D  D <-- SF ^ OF
setle D  D <-- (SF ^ OF) | ZF
setg D  D <-- ~(SF ^ OF) & ~ZF
setge D  D <-- ~(SF ^ OF)

Example: a < b (assume int a is in %eax, int b is in %ebx)

1. cmpl %ebx,%eax

2. setl %cl

3. movzbl %cl,%ecx
Instructions - Jumps

What?

*target*

Why?

How? Unconditional Jump

*indirect jump:*

    jmp *Operand

*direct jump:*

    jmp Label

How? Conditional Jumps

*both:*

    je Label   jne Label   js Label   jns Label

*unsigned:*

    jb Label   jbe Label   ja Label   jae Label

*signed:*

    jl Label   jle Label   jg Label   jge Label
Encoding Targets

What?

Absolute Encoding

Problems?

- code is not
- code cannot be

Solution?

IA-32:

→ What is the distance (in hex) encoded in the jne instruction?

<table>
<thead>
<tr>
<th>Assembly Code</th>
<th>Address</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpl %eax, %ecx</td>
<td>0x_B8</td>
<td>75 ??</td>
</tr>
<tr>
<td>jne .L1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>movl $11, %eax</td>
<td>0x_BA</td>
<td></td>
</tr>
<tr>
<td>movl $22, %edx</td>
<td>0x_BC</td>
<td></td>
</tr>
<tr>
<td>.L1:</td>
<td>0x_BE</td>
<td></td>
</tr>
</tbody>
</table>

→ If the jb instruction is 2 bytes in size and is at 0x08011357 and the target is at 0x8011340 then what is the distance (hex) encoded in the jb instruction?
Converting Loops

Identify which C loop statement (for, while, do-while) corresponds to each goto code fragment below.

```c
loop1:
  loop_body
  t = loop_condition
  if (!t) goto done:

loop2:
  loop_body
  t = loop_condition
  if (t) goto loop1:

done:
  loop_init
  t = loop_condition
  if (!t) goto done:

loop3:
  loop_body
  loop_update
  t = loop_condition
  if (t) goto loop3

done:
```

Most compilers (gcc included)