From last time...

```assembly
movl %ebp, %ecx
subl $16, %ecx
movl (%ecx), %eax
orl %eax, -8(%ebp)
negl %eax
movl %eax, 4(%ecx)
```

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x455C</td>
<td>7</td>
</tr>
<tr>
<td>0x4560</td>
<td>11</td>
</tr>
<tr>
<td>0x4564</td>
<td>5</td>
</tr>
<tr>
<td>0x4568</td>
<td>3</td>
</tr>
<tr>
<td>0x456C</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
How would you do this in IA32?

x is 2 at %ebp-8, y is 3 at %ebp-12, z is 2 at %ebp-16

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x1270</td>
</tr>
</tbody>
</table>

C code: \( z = x \ XOR \ y \)
How would you do this in IA32?

x is 2 at %ebp−8, y is 3 at %ebp−12, z is 2 at %ebp−16

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x1270</td>
</tr>
<tr>
<td>0x1260</td>
<td>2</td>
</tr>
<tr>
<td>0x1264</td>
<td>3</td>
</tr>
<tr>
<td>0x1268</td>
<td>2</td>
</tr>
<tr>
<td>0x126c</td>
<td></td>
</tr>
<tr>
<td>0x1270</td>
<td></td>
</tr>
</tbody>
</table>

C code: \(z = x \oplus y\)

A:    movl  -8(%ebp),  %eax
       movl  -12(%ebp), %edx
       xorl  %eax,  %edx
       movl  %eax,  -16(%ebp)

B:    movl  -8(%ebp),  %eax
       movl  -12(%ebp), %edx
       xorl  %edx,  %eax
       movl  %eax,  -16(%ebp)

C:    movl  -8(%ebp),  %eax
       movl  -12(%ebp), %edx
       xorl  %eax,  %edx
       movl  %eax,  -8(%ebp)

D:    movl  -16(%ebp), %eax
       movl  -12(%ebp), %edx
       xorl  %edx,  %eax
       movl  %eax,  -8(%ebp)

E: none of these implements \(z = x \oplus y\)
How would you do this in IA32?

x is 2 at %ebp−8, y is 3 at %ebp−12, z is 2 at %ebp−16

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x1270</td>
</tr>
</tbody>
</table>

x = y >> 3 | x * 8
(1) \( z = x^y \)

\[
\text{movl} \ -8(\%ebp), \ %eax \quad \# R[\%eax] \leftarrow x \\
\text{movl} \ -12(\%ebp), \ %edx \quad \# R[\%edx] \leftarrow y \\
xorl \ %edx, \ %eax \quad \# R[\%eax] \leftarrow x^y \\
\text{movl} \ %eax, \ -16(\%ebp) \quad \# M[R[\%ebp-16]] \leftarrow x^y
\]

(2) \( x = y \gg 3 \mid x \times 8 \)

\[
\text{movl} \ -8(\%ebp), \ %eax \quad \# R[\%eax] \leftarrow x \\
\text{imull} \ $8, \ %eax \quad \# R[\%eax] \leftarrow x \times 8 \\
\text{movl} \ -12(\%ebp), \ %edx \quad \# R[\%edx] \leftarrow y \\
\text{rshl} \ $3, \ %edx \quad \# R[\%edx] \leftarrow y \gg 3 \\
\text{orl} \ %eax, \ %edx \quad \# R[\%edx] \leftarrow y \gg 3 \mid x \times 8 \\
\text{movl} \ %edx, \ -8(\%ebp) \quad \# M[R[\%ebp-8]] \leftarrow \text{result}
\]
Recall Memory Operands

- \textit{displacement} (\%reg)
  - e.g., addl \%eax, -8(\%ebp)

- IA32 allows a memory operand as the source or destination, but NOT BOTH
  - One of the operands must be a register

- This would \textbf{not} be allowed:
  - addl -4(\%ebp), -8(\%ebp)
  - If you wanted this, \texttt{movl} one value into a register first
Unconditional Jumping / Goto

A label is a place you might jump to.

Labels are ignored except for goto/jumps.

(Skipped over if encountered)

```
int main() {
    int a = 10;
    int b = 20;

    goto label1;
    a = a + b;

label1:
    return;
```

```
int x = 20;
L1:
    int y = x + 30;
L2:
    printf("%d, %d\n", x, y);
```
Unconditional Jumping / Goto

```c
int main() {
    int a = 10;
    int b = 20;

goto label1;
    a = a + b;

label1:
    return;
}
```
jmp isn’t very useful by itself...

We’d like to use branch instructions for:
• if/else
• switch
• for loops
• while loops

But if jmp were our only branch instruction, the closest we could get would be an infinite loop.

We need conditional jumps.
Condition Codes (or Flags)

• Set in two ways:
  1. As “side effects” produced by ALU
  2. In response to explicit comparison instructions

• IA-32, condition codes tell you:
  • If the result is zero (ZF)
  • If the result’s first bit is set (negative if signed) (SF)
  • If the result overflowed (assuming unsigned) (CF)
  • If the result overflowed (assuming signed) (OF)
Processor State in Registers

- Temporary data
  \%eax - \%edi

- Location of runtime stack
  \%ebp, \%esp

- Location next instruction
  \%eip

- Status of recent tests
  \%EFLAGS:
  CF, ZF, SF, OF

\%eax
\%ecx
\%edx
\%ebx
\%esi
\%edi
\%esp
\%ebp
\%eip

General purpose registers

Current stack top
Current stack frame
Instruction pointer (PC)

Condition codes
Instructions that set condition codes

1. Arithmetic/logic side effects (addl, subl, orl, etc.)

2. \textbf{CMP and TEST:}

\texttt{cmpl b,a} like computing \texttt{a-b} without storing result
- Sets \texttt{OF} if overflow, Sets \texttt{CF} if carry-out,
- Sets \texttt{ZF} if result zero, Sets \texttt{SF} if results is negative

\texttt{testl b,a} like computing \texttt{a\&b} without storing result
- Sets \texttt{ZF} if result is zero, sets \texttt{SF} if \texttt{a\&b} < 0
- \texttt{OF} and \texttt{CF} flags are zero (no overflow with \&
Which flags would this `subl` set?

• Suppose `%eax` holds 5, `%ecx` holds 7

\[ \text{subl } \$5, \%eax \]

If the result is zero (ZF)
If the result’s first bit is set (negative if signed) (SF)
If the result overflowed (assuming unsigned) (CF)
If the result overflowed (assuming signed) (OF)

A. ZF
B. SF
C. CF and ZF
D. CF and SF
E. CF, SF, and CF
Which flags would this `cmp` set?

• Suppose `%eax` holds 5, `%ecx` holds 7

`cmp %ecx, %eax`

If the result is zero (ZF)
If the result’s first bit is set (negative if signed) (SF)
If the result overflowed (assuming unsigned) (CF)
If the result overflowed (assuming signed) (OF)

A. ZF
B. SF
C. CF and ZF
D. CF and SF
E. CF, SF, and CF
Conditional Jumping

• Jump based on which condition codes are set

<table>
<thead>
<tr>
<th>Jump Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned jg)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Example Scenario

```c
int userval;
scanf("%d", &userval);

if (userval == 42) {
    userval += 5;
} else {
    userval -= 10;
}
```

- Suppose user gives us a value via `scanf`
- We want to check to see if it equals 42
  - If so, add 5
  - If not, subtract 10

...
How would we use jumps/CCs for this?

```c
int userval;
scanf("%d", &userval);
if (userval == 42) {
    userval += 5;
} else {
    userval -= 10;
}
...
```

Assume userval is stored in %eax at this point.
How would we use jumps/CCs for this?

int userval;
scanf("%d", &userval);

if (userval == 42) {
    userval += 5;
} else {
    userval -= 10;
}

... (A) cmpl $42, %eax  jne L2
       L1:  subl $10, %eax  jmp DONE
       L2:  addl $5, %eax  DONE:
           ...

(B) cmpl $42, %eax  jne L2
       L1:  subl $10, %eax  jmp DONE
       L2:  addl $5, %eax  DONE:
           ...

(C) cmpl $42, %eax  jne L2
       L1:  addl $5, %eax  jmp DONE
       L2:  subl $10, %eax  DONE:
           ...

Assume userval is stored in %eax at this point.
Loops via goto

Goal: translate for loops and while loops to IA32.

• We know how to translate a for loop to a while loop, so let’s focus on while loops.

• Intermediate step: translate C code with a while loop into C code with goto statements.
Translate while $\rightarrow$ goto

```c
int i=1, j=100, k=0;
while(i < j){
    i *= 2;
    j -= i;
}
k = j + i;
```
Translate goto → IA32

```
int i=1, j=100, k=0;

L1:
    if(i >= j) goto L2;
    i *= 2;
    j -= i;
    goto L1;

L2:
    k = j + i;
```

```
| 0x8B00 | 2 | k |
| 0x8B04 | 3 | j |
| 0x8B08 | 2 | i |
| 0x8B0c |
| 0x8B10 | (%ebp) |
```

**Hint:**
cmpl,jge,jmp