Announcements

• Assignment 1 due today, 11:59pm
  • Hand in at front during break or email it to cs61-staff@seas.harvard.edu
  • If you need to use late days, you must email us before deadline

• Sections started yesterday
  • Contact course staff if you haven’t been assigned a section
  • Please try to attend your assigned section
  • On trial basis, we are allowing students to attend other sections.

• Infrastructure
  • Some issues yesterday and this morning, have been resolved
Office Hours and New Course Staff

• Office hours posted on website
• New course staff

Gabrielle Ehrlich

Randy Miller
Today

• Data types
  • Register names

• Control flow
  • jmp
  • Condition flags
  • Loops
  • Switch statements
Data types

• All examples have dealt with 4-byte integers
  • Instructions addl, subl, movl, etc.
  • The “l” (ell) at the end represents “long” ... which is the x86 data type that holds an int.

• Many instructions can operate on different data types:
  • addb – byte, 8 bits (C type: char, unsigned char)
  • addw – word, 16 bits (C type: short, unsigned short)
  • addl – long, 32 bits (C type: int, unsigned int, long, unsigned long)
Register names

- Registers `%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`, `%edi`, `%esp`, `%ebp` are all 32-bit
- Sometimes we handle data smaller than 32 bits
  - Have names for addressing just some bits of a register
    - Historical, due to development of IA32 from 8 and 16 bit architectures
Register names

General purpose registers

- %eax
- %ecx
- %edx
- %ebx
- %esi
- %edi
- %esp
- %ebp

16-bit virtual registers

(backwards compatibility)

Origin (mostly obsolete)
- accumulate
- counter
- data
- base
- source index
- destination index
- stack pointer
- base pointer
### Register name example

The diagram shows the register names and their corresponding values.

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0xdead0000</td>
</tr>
<tr>
<td>%ecx</td>
<td>0xdead0000</td>
</tr>
</tbody>
</table>

The corresponding assembly code is:

```assembly
movl $0xdead0000, %eax
movb $0xef, %al
movb $0xbe, %ah
movl %eax, %ecx
```
## Register name example

<table>
<thead>
<tr>
<th>%eax</th>
<th>dead</th>
<th>00</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ax</td>
<td></td>
<td>%ah</td>
<td>%al</td>
</tr>
<tr>
<td>%ah</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%al</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%ecx</th>
<th>0000</th>
<th>00</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>%cx</td>
<td></td>
<td>%cl</td>
<td>%ch</td>
</tr>
<tr>
<td>%ch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%cl</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```assembly
movl  $0xdead0000, %eax  
movb  $0xef, %al         
movb  $0xbe, %ah         
movl  %eax, %ecx
```

Stephen Chong, Harvard University
Register name example

\[ \begin{array}{c|c|c}
\%eax & \text{dead} & 00 \\
\%ah  & \%ax  & \%al \\
\end{array} \]

\[ \begin{array}{c|c|c}
\%ecx & 0000 & 00 \\
\%ch  & \%cl  & \%cx \\
\end{array} \]

\textbf{Assembly Code:}

\begin{verbatim}
movl $0xdead0000, %eax
movb $0xef, %al
movb $0xbe, %ah
movl %eax, %ecx
\end{verbatim}
## Register name example

<table>
<thead>
<tr>
<th>%eax</th>
<th>dead</th>
<th>%be</th>
<th>%ef</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>%ah</td>
<td>%al</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%ecx</th>
<th>0000</th>
<th>%00</th>
<th>%00</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>%ch</td>
<td>%cl</td>
</tr>
</tbody>
</table>

```
movl  $0xdead0000, %eax
movb  $0xef, %al
movb  $0xbe, %ah
movl  %eax, %ecx
```
Register name example

%eax

dead

%ah

%ax

%al

%eax

dead

%ah

%ax

%al

%ecx

dead

%ch

%cx

%cl

movl $0xdead0000, %eax
movb $0xef, %al
movb $0xbe, %ah
movl %eax, %ecx
Today

• Data types
  • Register names

• Control flow
  • jmp
  • Condition flags
  • Loops
  • Switch statements
Control flow

- **Control flow** is the general term for any code that controls which parts of a program are executed.

- Examples in C
  - if \((expr)\) \{ ... \} else \{ ... \}
  - do \{ .... \} while \((expr)\);
  - while \((expr)\) \{ .... \}
  - for \((expr1; expr2; expr3)\) \{ ... \}
  - C “goto” statement
  - switch \((expr)\) \{ case \texttt{val1}: ...; case \texttt{val1}: ...; default: ...; \}
Control flow is about how the value of the instruction pointer changes.
Simplest case: jmp instruction

- Operation jmp label causes processor to “jump” to a new instruction and execute from there

- The label (“.L6”) is just a symbolic reference to a specific instruction in the program.

- Once compiled to a binary, the .L6 will be replaced by a memory address.
Symbolic labels

- The assembler replaces symbolic labels with the actual memory address when converting to machine code.
- They are just “placeholders” for memory addresses.

Output from gcc -S

.L3:
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  movl %eax, -4(%ebp)
  jmp .L6

.L5:
  movl 12(%ebp), %eax
  imull 8(%ebp), %eax
  movl %eax, -4(%ebp)

.L6:
  movl -4(%ebp), %eax
  leave
  ret

Output from objdump

08048476: movl 12(%ebp), %eax
08048477: addl 8(%ebp), %eax
08040479: movl %eax, -4(%ebp)
0804847c: jmp 08048489
0804847f: movl 12(%ebp), %eax
08048481: imull 8(%ebp), %eax
08048487: movl %eax, -4(%ebp)
08048489: movl -4(%ebp), %eax
0804848d: leave
0804848e: ret
Conditional branching

- `jmp` basically implements goto
  - Always same control flow
- How do we implement if statements, loops, etc?
  - Not always the same control flow
- Two kinds of instructions
- Comparison instructions (`cmpl`, `testl`, etc.)
  - Compare values of two registers
  - Set **condition flags** based on result
- Conditional branch instructions (`je`, `jne`, `jg`, etc.)
  - Jump depending on current value of condition flags
Condition flags

- Bits maintained by the processor representing result of previous arithmetic instruction
- Used for many purposes: To determine if there has been overflow, whether the result is zero, etc.
- Stored in a special “EFLAGS” register within processor
Condition flags

- Bits maintained by the processor representing result of previous arithmetic instruction
- Used for many purposes: To determine if there has been overflow, whether the result is zero, etc.
- Stored in a special “EFLAGS” register within processor
Some x86 condition flags

- **CF: Carry Flag**
  - The most recent operation generated a carry bit out of the MSB
  - Indicates an overflow when performing unsigned integer arithmetic

- **OF: Overflow Flag**
  - The most recent operation caused a 2’s complement overflow (either positive or negative)
  - Indicates an overflow when performing signed integer arithmetic

- **SF: Sign flag**
  - The most recent operation yielded a negative value
  - Equal to MSB of result; which indicates the sign of a two's complement integer
  - 0 means result was positive, 1 means negative

- **ZF: Zero flag**
  - The most recent operation yielded zero

- Condition flags are set *implicitly* by every arithmetic operation.
- Can also be set *explicitly* by comparison instructions.
Comparison instructions

- **cmpl** $src1, src2
  - Compares value of $src1$ and $src2$
  - $src1$, $src2$ can be registers, immediate values, or contents of memory.
  - Computes ($src2 - src1$) without modifying either operand
    - like "subl $src1$, $src2$" without changing $src2$
  - But, sets the condition flags based on the result of the subtraction.

- **testl** $src1, src2$
  - Like cmpl, but computes ($src1$ & $src2$) instead of subtracting them.
Condition flags example

```
movl $42, %eax
movl $6, %ebx
mull $7, %ebx
cmpl %eax, %ebx
```
Condition flags example

movl $42, %eax
movl $6, %ebx
mull $7, %ebx
cmpl %eax, %ebx

%eax  42
%ebx  0

CF  ZF  SF  OF
0    0    0    0

Stephen Chong, Harvard University
Condition flags example

movl $42, %eax
movl $6, %ebx
mull $7, %ebx
cmpl %eax, %ebx

%eax  42
%ebx  6

<table>
<thead>
<tr>
<th>CF</th>
<th>ZF</th>
<th>SF</th>
<th>OF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Condition flags example

```
movl $42, %eax
movl $6, %ebx
mull $7, %ebx
cmpl %eax, %ebx
```
Condition flags example

- Recall: `cmpl %eax, %ebx` computes %ebx – %eax and sets the flags
  - %ebx is not modified by the `cmpl` instruction (unlike `subl`)
- Condition flags are set after every instruction!
  - See x86 manual or textbook for details of which flags are affected by each instruction
  - In this example, the flags were only changed by the `cmpl` instruction
Another example

• Consider `cmpl %eax, %ebx`
  • computes %ebx – %eax
• How do we determine the relationship between %ebx and %eax based on the condition flags?
• Suppose %ebx is equal to %eax
  • Then %ebx – %eax is zero, and so ZF = 1
Another example

• Suppose %ebx is greater than %eax
  • Since %ebx > %eax result cannot be zero ⇒ ZF = 0

• Suppose no overflow occurs (i.e., OF = 0)
  • %ebx > %eax if and only if result is positive
  • ⇒ SF = 0 (indicating positive)

• Suppose overflow occurs (i.e., OF = 1)
  • %ebx > %eax if and only if result is negative
  • ⇒ SF = 1 (indicating negative)

• %ebx is greater than %eax
  if and only if ZF=0, and SF equal to OF
  if and only if ~(SF ∧ OF) & ~ZF
### Reading condition flags

- **Operation `setX dest` sets single byte based on condition code**

<table>
<thead>
<tr>
<th><code>setX</code></th>
<th>Synonyms</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sete</code></td>
<td><code>setz</code></td>
<td><code>dest = ZF</code></td>
<td>Equal/zero</td>
</tr>
<tr>
<td><code>setne</code></td>
<td><code>setnz</code></td>
<td><code>dest = ~ZF</code></td>
<td>Not equal/non-zero</td>
</tr>
<tr>
<td><code>sets</code></td>
<td></td>
<td><code>dest = SF</code></td>
<td>Negative</td>
</tr>
<tr>
<td><code>setns</code></td>
<td></td>
<td><code>dest = ~SF</code></td>
<td>Not negative</td>
</tr>
<tr>
<td><code>setg</code></td>
<td><code>setnle</code></td>
<td><code>dest = ~(SF ^ OF) &amp; ~ZF</code></td>
<td>Greater than (signed &gt;)</td>
</tr>
<tr>
<td><code>setge</code></td>
<td><code>netnl</code></td>
<td><code>dest = ~(SF ^ OF)</code></td>
<td>Greater than or equal (signed ≥)</td>
</tr>
<tr>
<td><code>setl</code></td>
<td><code>setnge</code></td>
<td><code>dest = SF ^ OF</code></td>
<td>Less than (signed &lt;)</td>
</tr>
<tr>
<td><code>setle</code></td>
<td><code>setng</code></td>
<td>`dest = (SF ^ OF)</td>
<td>ZF`</td>
</tr>
<tr>
<td><code>seta</code></td>
<td><code>setnbe</code></td>
<td><code>dest = ~CF &amp; ~ZF</code></td>
<td>Above (unsigned &gt;)</td>
</tr>
<tr>
<td><code>setb</code></td>
<td><code>setnae</code></td>
<td><code>dest = CF</code></td>
<td>Below (unsigned &lt;)</td>
</tr>
</tbody>
</table>
# Conditional jumps

- Operation `jX label` jumps to label if condition X is satisfied

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonyms</th>
<th>Jump condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>jmp</code></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><code>je</code></td>
<td><code>jz</code></td>
<td><code>ZF</code></td>
<td>Equal/zero</td>
</tr>
<tr>
<td><code>jne</code></td>
<td><code>jnz</code></td>
<td><code>~ZF</code></td>
<td>Not equal/non-zero</td>
</tr>
<tr>
<td><code>js</code></td>
<td></td>
<td><code>SF</code></td>
<td>Negative</td>
</tr>
<tr>
<td><code>jns</code></td>
<td></td>
<td><code>~SF</code></td>
<td>Not negative</td>
</tr>
<tr>
<td><code>jg</code></td>
<td><code>jnle</code></td>
<td><code>~(SF ^ OF) &amp; ~ZF</code></td>
<td>Greater than (signed &gt;)</td>
</tr>
<tr>
<td><code>jge</code></td>
<td><code>jnl</code></td>
<td><code>~(SF ^ OF)</code></td>
<td>Greater than or equal (signed ≥)</td>
</tr>
<tr>
<td><code>jl</code></td>
<td><code>jnle</code></td>
<td><code>SF ^ OF</code></td>
<td>Less than (signed &lt;)</td>
</tr>
<tr>
<td><code>jle</code></td>
<td><code>jng</code></td>
<td>`(SF ^ OF)</td>
<td>ZF`</td>
</tr>
<tr>
<td><code>ja</code></td>
<td><code>jnbe</code></td>
<td><code>~CF &amp; ~ZF</code></td>
<td>Above (unsigned &gt;)</td>
</tr>
<tr>
<td><code>jb</code></td>
<td><code>jnae</code></td>
<td><code>CF</code></td>
<td>Below (unsigned &lt;)</td>
</tr>
</tbody>
</table>
Condition flags example

```assembly
movl $42, %eax
movl $6, %ebx
mull $7, %ebx
cmpl %eax, %ebx
```

```
%eax 42
%ebx 42
```

```
<table>
<thead>
<tr>
<th>CF</th>
<th>ZF</th>
<th>SF</th>
<th>OF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Condition flags example

```assembly
movl $42, %eax
movl $6, %ebx
mull $7, %ebx
cmpl %eax, %ebx
jz 0x80459845
movl $33, %eax
...
```

<table>
<thead>
<tr>
<th>%eax</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebx</td>
<td>42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CF</th>
<th>ZF</th>
<th>SF</th>
<th>OF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
**Condition flags example**

- Instruction `jz 0x80459845` will set instruction pointer to `0x80459845` if ZF=1.

- Otherwise, execution continues after the instruction
  - i.e., with `movl $33, %eax`
More examples

• What do the following examples do?

```assembly
movl $0xffffffff, %eax
addl $0x1, %eax
jz 0x08045900
...
```
Jump if -1 + 1 is zero

```assembly
movl $6, %eax
subl $10, %eax
jl 0x08045900
...
```
Jump if 6 is less than 10

```assembly
movl $0x42, %eax
movl $0x77, %ebx
subl %ebx, %eax
js 0x08045900
...
```
Jump if $0x42 – 0x77 is negative

Given `cmpl src, dest`, what is the relationship of `dest` to `src`?
Example: absdiff

```c
int absdiff(int x, int y) {
    int result;
    if (x > y) {
        result = x - y;
    } else {
        result = y - x;
    }
    return result;
}
```

```assembly
absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L7
    subl %eax, %edx
    movl %edx, %eax
.L8:
    leave
    ret
.L7:
    subl %edx, %eax
    jmp .L8
```

Set up

Body 1

Finish

Body 2
Example: absdiff

C allows "goto" as a control flow mechanism

- Closer to machine-level programming
- Generally considered bad programming style

```c
int absdiff_goto(int x, int y) {
    int result;
    if (x <= y) goto Else;
    result = x-y;
    Exit:
    return result;
Else:
    result = y-x;
goto Exit;
}
```

```assembly
absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L7
    subl %eax, %edx
    movl %edx, %eax
.L8:
    leave
    ret
.L7:
    subl %edx, %eax
    jmp .L8
```

Set up
Body 1
Finish
Body 2
Example: absdiff

- C allows "goto" as a control flow mechanism
  - Closer to machine-level programming
  - Generally considered bad programming style

```c
int absdiff_goto(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
    Exit:
    return result;
    Else:
    result = y-x;
    goto Exit;
}
```

```
absdiff:
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %edx
movl 12(%ebp), %eax

%edx = x
%eax = y

cmpl %eax, %edx
jle .L7

subl %eax, %edx
movl %edx, %eax
.L8:
leave
ret

.L7:
subl %edx, %eax
jmp .L8
```
Today

- Data types
  - Register names
- Control flow
  - jmp
  - Condition flags
  - Loops
  - Switch statements
Implementing loops

- Two equivalent programs to compute factorial
- Goto version uses backwards branch to continue loop
  - Only takes branch if while condition \((x > 1)\) is true

```c
int fact_do(int x)
{
    int result = 1;
    do {
        result *= x;
        x = x-1;
    } while (x > 1);
    return result;
}

int fact_goto(int x)
{
    int result = 1;
    Loop:
        result *= x;
        x = x-1;
        if (x > 1)
            goto Loop;
    return result;
}
```
int fact_goto(int x) {
    int result = 1;
    Loop:
    result *= x;
    x = x-1;
    if (x > 1)
        goto Loop;
    return result;
}
While loops version 1

C code

```c
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```

Goto version 1

```c
int fact_while_goto(int x)
{
    int result = 1;
    Loop:
    if (!(x > 1))
        goto Done;
    result *= x;
    x = x-1;
    goto Loop;
    Done:
    return result;
}
```

- How is this different from the do-while version?
While loops version 2

C code

```c
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```

Goto version 2

```c
int fact_while_goto2(int x)
{
    int result = 1;
    if (!(x > 1))
        goto Done;
    Loop:
    result *= x;
    x = x-1;
    if (x > 1)
        goto Loop;
   Done:
    return result;
}
```

- Historically used by GCC
- Uses same inner loop as do-while version
- Guards loop entry with extra test
While loops version 2

While version

\[ \text{while (test)} \]
\[ \quad \text{body} \]

Do-While version

\[ \text{if (!test) goto done; do} \]
\[ \quad \text{body} \]
\[ \quad \text{while (test)} \]
\[ \text{done:} \]

Goto version

\[ \text{if (!test) goto done; loop:} \]
\[ \quad \text{body} \]
\[ \quad \text{if (test) goto loop;} \]
\[ \text{done:} \]
While loops version 3

C code

```c
int fact_while(int x) {
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```

Goto version 3

```c
int fact_while_goto3(int x) {
    int result = 1;
    goto middle;
loop:
    result *= x;
    x = x-1;
middle:
    if (x > 1) {
        goto loop;
    }
    return result;
}
```
While loops version 3

While version

```c
while (test)
    body
```

Goto version

```c
goto middle;
loop:
    body
middle:
    if (test) goto loop;
```

done:

- "Jump to middle" compilation
- Recent technique used by GCC
- Avoids duplicating test code
- Unconditional goto incurs no performance penalty
Compiling for loops

For version

```c
for (init; test; update)
    body
```

While version

```c
init
while (test)
    body
    update
```

Do-While version

```c
init
if (!test) goto done;
do
    body
    update
while (test)
done:
```
Switch statements

- Switch statements can be complex...
  - Many cases to consider
  - Can have “fall through”
    - No break at end of case 2
  - Can have missing cases
  - Can have default case
- How to compile?
  - Series of conditionals?
    - Works, but a lot of code, and expensive
  - Jump table
    - List of jump targets indexed by x
    - Less code, and fast!

```c
int switchexample(int x) {
    int y;
    switch(x) {
        case 1:
            y = x; break;
        case 2:
            y = 2*x;
            /* Fall through! */
        case 3:
            y = 3*x; break;
            /* No case 4! */
        case 5:
            y = 5*x; break;
        default:
            y = x; break;
    }
    return y;
}
```
Jump table structure

Switch code

```plaintext
switch(x) {
  case val_0:
    Block_0
  case val_1:
    Block_1
  ...
  case val_n-1:
    Block_n-1
}
```

Jump table

```plaintext
jtab:
  Targ0
  Targ1
  ...
  Targn-1
```

Jump targets

```plaintext
Targ0: Code block 0
Targ1: Code block 1
Targn-1: Code block n-1
```

Approximate translation

```plaintext
target = jtab[x];
goto *target;
```
Using a jump table

```c
int switchexample(int x) {
    int y;
    switch(x) {
        case 1:  y =   x; break;
        case 2:  y = 2*x;
        case 3:  y = 2*x; break;
        /*no case 4*/
        case 5:
        case 6:  y = 2*x; break;
        default: y =   0; break;
    }
    return y;
}
```

**Why multiply x by 4?**

```
pushl %ebp    # Setup
movl %esp, %ebp
subl $16, %esp
cmpl $6, 8(%ebp)  # Check if 'x' is > 6
ja   .L38     # If so, jump to .L38 (default case)
movl 8(%ebp), %eax  # %eax = x
sall $2, %eax     # Shift left by 2 (multiply by 4)
movl .L39(%eax), %eax # Move jumptable[x] to eax
jmp *%eax    # Jump to this address
```

**jmp *src is an indirect jump.**
Always jumps to the address that src evaluates to.
Using a jump table

```c
int switchexample(int x) {
    int y;
    switch(x) {
        case 1:  y =   x; break;
        case 2:  y = 2*x;
        case 3:  y = 2*x; break;
        /*no case 4*/
        case 5:
        case 6:  y = 2*x; break;
        default: y =   0; break;
    }
    return y;
}
```

```
pushl   %ebp                # Setup
movl    %esp, %ebp
subl    $16, %esp
cmpl    $6, 8(%ebp)         # Check if 'x' is > 6
ja      .L38                # If so, jump to .L38 (default case)
movl    8(%ebp), %eax       # %eax = x
sall    $2, %eax            # Shift left by 2 (multiply by 4)
movl    .L39(%eax), %eax    # Move jumptable[x] to eax
jmp     *%eax               # Jump to this address

.L39:            # Jumptable starts here
    .long   .L38  # Entry 0 is symbol .L38 (default)
    .long   .L34  # Entry 1 is symbol .L34
    .long   .L35  # Entry 2 is symbol .L35
    .long   .L36  # Entry 3 is symbol .L36
    .long   .L38  # Entry 4 is symbol .L38 (default)
    .long   .L37  # Entry 5 is symbol .L37
    .long   .L37  # Entry 6 is symbol .L37
```
Using a jump table

```c
int switchexample(int x) {
    int y;
    switch(x) {
        case 1:  y =   x; break;
        case 2:  y = 2*x;
        case 3:  y = 2*x; break;
        /*no case 4*/
        case 5:
        case 6:  y = 2*x; break;
        default: y =   0; break;
    }
    return y;
}
```

### Jumptable

- Entry 0 is symbol `.L38` (default)
- Entry 1 is symbol `.L34`
- Entry 2 is symbol `.L35`
- Entry 3 is symbol `.L36`
- Entry 4 is symbol `.L38` (default)
- Entry 5 is symbol `.L37`
- Entry 6 is symbol `.L37`

```asm
.L34:    # Case for Entry 1 (x == 1)
    movl  8(%ebp), %eax   # %eax = x
    movl  %eax, -4(%ebp)  # y = %eax
    jmp   .L40            # Jump out of 'switch'
```

.L39:    # Jumptable starts here
    .long  .L38           # Entry 0 is symbol .L38 (default)
    .long  .L34           # Entry 1 is symbol .L34
    .long  .L35           # Entry 2 is symbol .L35
    .long  .L36           # Entry 3 is symbol .L36
    .long  .L38           # Entry 4 is symbol .L38 (default)
    .long  .L37           # Entry 5 is symbol .L37
    .long  .L37           # Entry 6 is symbol .L37

.L40:    # Return from 'switch'
    ret
Sparse switch statement

- Jump tables work fine when...
  - Small number of jump targets
  - Mostly contiguous (few missing cases)
- Inefficient to use a jump table if the switch space is “sparse”
  - Example would require a jump table of 1000 entries, 990 of which all point to the same “default” case.
- Could use simple translation to multiple conditional branches
  - No more than 9 tests would be required

```c
/* Return x/111 if x is multiple && <= 999. -1 otherwise */
int div111(int x)
{
    switch(x) {
    case   0: return 0;
    case 111: return 1;
    case 222: return 2;
    case 333: return 3;
    case 444: return 4;
    case 555: return 5;
    case 666: return 6;
    case 777: return 7;
    case 888: return 8;
    case 999: return 9;
    default: return -1;
    }
}
```
Better approach: branch tree

- Compare $x$ to possible case values
- Jump to different target depending on outcome of each test

```assembly
movl 8(%ebp),%eax    # get $x$
cmpl $444,%eax      # Compare to 444
je .L8
jg .L16
cmpl $111,%eax     # Compare to 111
je .L5
jg .L17
testl %eax,%eax    # Compare to 0
je .L4
jmp .L14
...
```

```
... .L5:
  movl $1,%eax
  jmp .L19
.L6:
  movl $2,%eax
  jmp .L19
.L7:
  movl $3,%eax
  jmp .L19
.L8:
  movl $4,%eax
  jmp .L19
...```

Branch tree structure

- Organizes cases as binary tree
- Logarithmic performance to find right case
  - Better than linear!
Next lecture

• Procedures
  • Implementing procedure calls
  • Using the stack
  • Storing and accessing local variables
  • Saving and restoring registers
  • Recursive procedures

• x86_64