Machine-Level Representation

CSCI 2021: Machine Architecture and Organization

Antonia Zhai
Department Computer Science and Engineering
University of Minnesota
http://www.cs.umn.edu/~zhai

With Slides from Bryant and O’Hallaron

Control Flow
### Processor State (IA32, Partial)

- Information about currently executing program
- Temporary data (%eax, ...)
- Location of runtime stack (%ebp, %esp)
- Location of current code control point (%eip, ...)
- Status of recent tests (CF, ZF, SF, OF)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>General purpose registers</td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>Current stack top</td>
</tr>
<tr>
<td>%ebp</td>
<td>Current stack frame</td>
</tr>
<tr>
<td>%eip</td>
<td>Instruction pointer</td>
</tr>
</tbody>
</table>

### Condition Codes (Implicit Setting)

- Single bit registers
  - CF: Carry Flag (for unsigned)
  - SF: Sign Flag (for signed)
  - ZF: Zero Flag
  - OF: Overflow Flag (for signed)

- Implicitly set (think of it as side effect) by arithmetic operations
  - Example: `addl/addq Src, Dest` $\leftrightarrow t = a+b$
    - **CF** set if carry out from most significant bit (unsigned overflow)
    - **ZF** set if $t == 0$
    - **SF** set if $t < 0$ (as signed)
    - **OF** set if two's-complement (signed) overflow
      - `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

- Not set by `lea` instruction
### Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  - `cmpl cmpq` `Src2, Src1`
  - `cmp l b, a` like computing `a-b` without setting destination

- **CF set** if carry out from most significant bit (used for unsigned comparisons)
- **ZF set** if `a == b`
- **SF set** if `(a-b) < 0` (as signed)
- **OF set** if two’s-complement (signed) overflow
  
  \[(a>0 \&\& b<0 \&\& (a-b)<0) || (a<0 \&\& b>0 \&\& (a-b)>0)\]

### Condition Codes (Explicit Setting: Test)

- **Explicit Setting by Test instruction**
  - `testl testq` `Src2, Src1`
  - `test l b, a` like computing `a\&b` without setting destination

- Sets condition codes based on value of `Src1 & Src2`
- Useful to have one of the operands be a mask

- **ZF set when** `a\&b == 0`
- **SF set when** `a\&b < 0`
Reading Condition Codes

- **SetX Instructions**
  - Set single byte based on combinations of condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setsns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF^OF)&amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>seta</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

Reading Condition Codes (Cont.)

- **SetX Instructions**
  - Set single byte based on combinations of condition codes
  - One of 8 addressable byte registers
    - Embedded within first 4 integer registers
    - Does not alter remaining 3 bytes
    - Typically use `movzbl` to finish job

```c
int gt (int x, int y) {
    return x > y;
}
```

```assembly
movl 12(%ebp),%eax  # eax = y
cmpl %eax,8(%ebp)   # Compare x : y
setg %al            # al = x > y
movzbl %al,%eax     # Zero rest of %eax
```

Note inverted ordering!
Reading Condition Codes: x86-64

- SetX Instructions:
  - Set single byte based on combination of condition codes
  - Does not alter remaining 3 bytes

```c
int gt (long x, long y)
{
    return x > y;
}

long lgt (long x, long y)
{
    return x > y;
}
```

```assembly
cmpl %esi, %edi
setg %al
movzbl %al, %eax
```

32-bit instructions set high order 32 bits to 0!

Jumping

jX Instructions: Jump to different part of code

<table>
<thead>
<tr>
<th>jX</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>ZF</td>
</tr>
<tr>
<td>ja</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Conditional Branch Example

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

absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L6
    subl %eax, %edx
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```

Setup

Body1

Body2a

Body2b

Finish
Conditional Branch Example (Cont.)

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

- C allows "goto" as means of transferring control
- Closer to machine-level programming style
- Generally considered bad coding style

```
absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L6
    subl %eax, %edx
    movl %edx, %eax
    jmp .L7
    .L6:
    subl %edx, %eax
    .L7:
    popl %ebp
    ret
```

With Slides from Bryant and O’Hallaron
Conditional Branch Example (Cont.)

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

```assembly
absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L6
    subl %eax, %edx
    movl %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
    .L7:
    popl %ebp
    ret
```

With Slides from Bryant and O’Hallaron
### General Conditional Expression Translation

**C Code**

```c
val = Test ? Then_Expr : Else_Expr;
```

**Goto Version**

```c
nt = !Test;
if (nt) goto Else;
val = Then_Expr;
goto Done;
else:
val = Else_Expr;
```

- **Test** is expression returning integer
  - `== 0` interpreted as false
  - `!= 0` interpreted as true
- **Create separate code regions for then & else expressions**
- **Execute appropriate one**

---

### Conditional Move Example: x86-64

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

```asm
absdiff:
    movl %edi, %edx  # x in %edi
    subl %esi, %edx  # tval = x-y
    movl %esi, %eax  # y in %esi
    subl %edi, %eax  # result = y-x
    cmpl %esi, %edi  # Compare x:y
    cmovg %edx, %eax  # If >, result = tval
    ret
```

---

With Slides from Bryant and O’Hallaron
Bad Cases for Conditional Move

Expensive Computations

```java
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```java
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

```java
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

Loops

With Slides from Bryant and O'Hallaron
"Do-While" Loop Example

C Code

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

Goto Version

```c
int fact_goto(int x) {
    int result = 1;
    loop:
        result *= x;
        x = x-1;
        if (x > 1)
            goto loop;
    return result;
}
```

- Use backward branch to continue looping
- Only take branch when "while" condition holds

"Do-While" Loop Compilation

Goto Version

```c
int fact_goto(int x) {
    int result = 1;
    loop:
        result *= x;
        x = x-1;
        if (x > 1)
            goto loop;
    return result;
}
```

Assembly

```asm
_fact_goto:
    pushl %ebp
    movl %esp,%ebp
    movl $1,%eax
    movl 8(%ebp),%edx
    L11:
        imull %edx,%eax
        decl %edx
        cmpl $1,%edx
        jg L11
        goto loop;
    ret
```

- Registers
  - %edx x
  - %eax result

With Slides from Bryant and O'Hallaron
General “Do-While” Translation

C Code

\[
\text{do} \\
\quad \text{Body} \\
\text{while (Test);} \\
\]

Goto Version

\[
\text{loop:} \\
\quad \text{Body} \\
\quad \text{if (Test)} \\
\quad \quad \text{goto loop} \\
\]

Body can be any C statement or compound statement:

\[
\{ \\
\quad \text{Statement}_1; \\
\quad \text{Statement}_2; \\
\quad \ldots \\
\quad \text{Statement}_n; \\
\}
\]

- Test is expression returning integer
  - = 0 interpreted as false
  - ≠0 interpreted as true

“While” Loop Example #1

C Code

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

First Goto Version

```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; 
}
```

- Is this code equivalent to the do-while version?
- Must jump out of loop if test fails
### General "While" Translation

#### C Code

```
while (Test)
  Body
```

#### Do-While Version

```
if (!Test) goto done;
do
  Body
while (Test);
done:
```

#### Goto Version

```
if (!Test) goto done;
loop:
  Body
  if (Test) goto loop;
done:
```

---

With Slides from Bryant and O'Hallaron
"For" Loop Example

C Code

```c
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
    int i;
    int result = 0;
    for (i = 0; i < WSIZE; i++) {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    return result;
}
```

- Is this code equivalent to other versions?

"For" Loop Form

**General Form**

```
for (Init; Test; Update)
Body
```

- **Init**
  - `i = 0`
- **Test**
  - `i < WSIZE`
- **Update**
  - `i++`
- **Body**
  ```c
  for (i = 0; i < WSIZE; i++) {
      unsigned mask = 1 << i;
      result += (x & mask) != 0;
  }
  ```
“For” Loop $\rightarrow$ While Loop

For Version

```plaintext
for (Init; Test; Update)  
  Body
```

While Version

```plaintext
Init;  
while (Test) {  
  Body  
  Update;
}
```

“For” Loop $\rightarrow$ ... $\rightarrow$ Goto

For Version

```plaintext
for (Init; Test; Update)  
  Body
```

While Version

```plaintext
Init;  
while (Test) {  
  Body  
  Update;
}
```

```
Init:  
  if (!Test)  
    goto done;  
loop:  
  Body  
  Update  
  if (Test)  
    goto loop;

done:
```
"For" Loop Conversion Example

C Code

```c
#define WSIZE 8*sizeof(int)

int pcount_for(unsigned x) {
    int i;
    int result = 0;
    for (i = 0; i < WSIZE; i++) {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    return result;
}
```

• Initial test can be optimized away

Goto Version

```c
int pcount_for_gt(unsigned x) {
    int i;
    int result = 0;
    i = 0;
    if (!i < WSIZE) goto done;
    loop:
    {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    i++;
    if (i < WSIZE) goto loop;
    done:
    return result;
}
```

Switch Statement

With Slides from Bryant and O'Hallaron

2/8/15 CSCI 2021
Switch Statement Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

long switch_eg
  (long x, long y, long z)
  {
    long w = 1;
    switch(x) {
      case 1:
        w = y*z;
        break;
      case 2:
        w = y/z;
        /* Fall Through */
      case 3:
        w += z;
        break;
      case 5:
      case 6:
        w -= z;
        break;
      default:
        w = 2;
    }
    return w;
  }

Jump Table Structure

Switch Form
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
  ... 
  case val_n-1:
    Block n-1
}

Jump Table

Jump Targets

Approximate Translation

target = JTab[x];
goto *target;
Switch Statement Example (IA32)

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

Setup:
```
switch_eg:
pushl %ebp          # Setup
movl %esp, %ebp     # Setup
movl 8(%ebp), %eax # %eax = x
cmpl $6, %eax       # Compare x:6
ja .L2              # If unsigned > goto default
jmp *.L7(%eax,4)    # Goto *JTab[x]
```

Note that w not initialized here

What range of values takes default?

Switch Statement Example (IA32)

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

```
Jump table
.sect .rodata
.align 4
.L7:
.long .L2 # x = 0
.long .L3 # x = 1
.long .L4 # x = 2
.long .L5 # x = 3
.long .L6 # x = 4
.long .L6 # x = 5
.long .L6 # x = 6
```

Indirect jump

With Slides from Bryant and O'Hallaron
Assembly Setup Explanation

- **Table Structure**
  - Each target requires 4 bytes
  - Base address at .L7

- **Jumping**
  - **Direct:** `jmp .L2`
  - Jump target is denoted by label `.L2`
  - **Indirect:** `jmp * .L7(%eax,4)`
  - Start of jump table: `.L7`
  - Must scale by factor of 4 (labels have 32-bits = 4 Bytes on IA32)
  - Fetch target from effective Address `.L7 + eax*4`
    - Only for $0 \leq x \leq 6$

Jump Table

```
switch(x) {
    case 1:    // .L3
        w = y*z;
        break;
    case 2:    // .L4
        w = y/z;
        /* Fall Through */
    case 3:    // .L5
        w += z;
        break;
    case 5:
    case 6:    // .L6
        w -= z;
        break;
    default:    // .L2
        w = 2;
}
```
Handling Fall-Through

```c
long w = 1;
  ...
switch(x) {
  ...
  case 2:
    w = y/z;
    /* Fall Through */
  case 3:
    w += z;
    break;
  ...
}
```

Code Blocks (Partial)

```assembly
.L2:       # Default
    movl $2, %eax # w = 2
    jmp .L8      # Goto done

.L5:       # x == 3
    movl $1, %eax # w = 1
    jmp .L9      # Goto merge

.L3:       # x == 1
    movl 16(%ebp), %eax # z
    imull 12(%ebp), %eax # w = y*z
    jmp .L8        # Goto done
```

With Slides from Bryant and O'Hallaron
### Code Blocks (Rest)

```
switch(x) {
    . . .
      case 2: // .L4
           w = y/z;
           /* Fall Through */
           merge:  // .L9
                   w += z;
                   break;
      case 5:
      case 6: // .L6
             w -= z;
             break;
}
```

```
.L4:      # x == 2
          movl 12(%ebp), %edx
          movl %edx, %eax
          sarl $31, %edx
          idivl 16(%ebp)  # w = y/z

.L9:      # merge:
          addl 16(%ebp), %eax  # w += z
          jmp .L8   # goto done

.L6:      # x == 5, 6
          movl $1, %eax  # w = 1
          subl 16(%ebp), %eax  # w = 1-z
```

### Switch Code (Finish)

```
return w;
```

```
.L8:      # done:
          popl %ebp
          ret
```

- **Noteworthy Features**
  - Jump table avoids sequencing through cases
  - Constant time, rather than linear
  - Use jump table to handle holes and duplicate tags
  - Use program sequencing to handle fall-through
  - Don’t initialize w = 1 unless really need it

With Slides from Bryant and O’Hallaron
x86-64 Switch Implementation

- Same general idea, adapted to 64-bit code
- Table entries 64 bits (pointers)
- Cases use revised code

```
switch(x) {
    case 1:  // .L3
        w = y*z;
        break;
    . . .
}
```

Jump Table

```
.section .rodata
.align 8
.L7:
    .quad .L2 # x = 0
    .quad .L3 # x = 1
    .quad .L4 # x = 2
    .quad .L5 # x = 3
    .quad .L2 # x = 4
    .quad .L6 # X = 5
    .quad .L6 # x = 6
```

IA32 Object Code

- Setup
  - Label .L2 becomes address 0x8048422
  - Label .L7 becomes address 0x8048660

Assembly Code

```
switch_eg:
    . . .
    ja .L2  # If unsigned > goto default
    jmp *.L7(%eax,4) # Goto *JTab[x]
```

Disassembled Object Code

```
08048410 <switch_eg>:
    . . .
08048419: 77 07 ja 08048422 <switch_eg+0x12>
0804841b: ff 24 85 60 86 04 08 jmp *0x8048660(%eax,4)
```
IA32 Object Code (cont.)

- Jump Table
  - Doesn't show up in disassembled code
  - Can inspect using GDB
    - `gdb switch`
    - `(gdb) x/7xw 0x8048660`
      - Examine 7 hexadecimal format "words" (4-bytes each)
      - Use command "help x" to get format documentation

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048660</td>
<td>0x8048422</td>
<td>0</td>
</tr>
<tr>
<td>0x8048664</td>
<td>0x8048432</td>
<td>1</td>
</tr>
<tr>
<td>0x8048668</td>
<td>0x804843b</td>
<td>2</td>
</tr>
<tr>
<td>0x804866c</td>
<td>0x8048429</td>
<td>3</td>
</tr>
<tr>
<td>0x8048670</td>
<td>0x8048422</td>
<td>4</td>
</tr>
<tr>
<td>0x8048674</td>
<td>0x804844b</td>
<td>5</td>
</tr>
<tr>
<td>0x8048678</td>
<td>0x804844b</td>
<td>6</td>
</tr>
</tbody>
</table>
Disassembled Targets

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048422:</td>
<td>b8 02 00 00 00</td>
<td>mov $0x2, %eax</td>
<td>8048422:</td>
</tr>
<tr>
<td>8048427:</td>
<td>eb 2a</td>
<td>jmp 8048453</td>
<td></td>
</tr>
<tr>
<td>8048429:</td>
<td>b8 01 00 00 00</td>
<td>mov $0x1, %eax</td>
<td></td>
</tr>
<tr>
<td>804842e:</td>
<td>66 90</td>
<td>xchg %eax, %ax</td>
<td></td>
</tr>
<tr>
<td>8048430:</td>
<td>eb 14</td>
<td>jmp 804846</td>
<td></td>
</tr>
<tr>
<td>8048432:</td>
<td>8b 45 10</td>
<td>mov 0x10(%ebp), %eax</td>
<td>8048435:</td>
</tr>
<tr>
<td>8048435:</td>
<td>0f af 45 0c</td>
<td>imul 0xc(%ebp), %eax</td>
<td>8048439:</td>
</tr>
<tr>
<td>8048439:</td>
<td>eb 18</td>
<td>jmp 8048453</td>
<td></td>
</tr>
<tr>
<td>804843b:</td>
<td>8b 55 0c</td>
<td>mov 0xc(%ebp), %edx</td>
<td>804843e:</td>
</tr>
<tr>
<td>804843e:</td>
<td>89 d0</td>
<td>mov %edx, %eax</td>
<td></td>
</tr>
<tr>
<td>8048440:</td>
<td>c1 fa 1f</td>
<td>sar $0x1f, %edx</td>
<td></td>
</tr>
<tr>
<td>8048443:</td>
<td>07 7d 10</td>
<td>idivl 0x10(%ebp)</td>
<td>8048446:</td>
</tr>
<tr>
<td>8048446:</td>
<td>03 45 10</td>
<td>add 0x10(%ebp), %eax</td>
<td>8048449:</td>
</tr>
<tr>
<td>8048449:</td>
<td>eb 08</td>
<td>jmp 8048453</td>
<td></td>
</tr>
<tr>
<td>804844b:</td>
<td>b8 01 00 00 00</td>
<td>mov $0x1, %eax</td>
<td></td>
</tr>
<tr>
<td>8048450:</td>
<td>2b 45 10</td>
<td>sub 0x10(%ebp), %eax</td>
<td>8048453:</td>
</tr>
<tr>
<td>8048453:</td>
<td>5d</td>
<td>pop %ebp</td>
<td></td>
</tr>
<tr>
<td>8048454:</td>
<td>c3</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

Matching Disassembled Targets

Value

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048422</td>
<td>mov $0x2, %eax</td>
<td>8048422:</td>
<td>mov $0x2, %eax</td>
</tr>
<tr>
<td>0x8048427</td>
<td>jmp 8048453</td>
<td>8048427:</td>
<td>jmp 8048453</td>
</tr>
<tr>
<td>0x8048429</td>
<td>mov $0x1, %eax</td>
<td>8048429:</td>
<td>mov $0x1, %eax</td>
</tr>
<tr>
<td>0x804842e</td>
<td>xchg %eax, %ax</td>
<td>804842e:</td>
<td>xchg %eax, %ax</td>
</tr>
<tr>
<td>0x8048430</td>
<td>jmp 804846</td>
<td>8048430:</td>
<td>jmp 804846</td>
</tr>
<tr>
<td>0x8048432</td>
<td>mov 0x10(%ebp), %eax</td>
<td>8048432:</td>
<td>mov 0x10(%ebp), %eax</td>
</tr>
<tr>
<td>0x8048435</td>
<td>imul 0xc(%ebp), %eax</td>
<td>8048435:</td>
<td>imul 0xc(%ebp), %eax</td>
</tr>
<tr>
<td>0x8048439</td>
<td>jmp 8048453</td>
<td>8048439:</td>
<td>jmp 8048453</td>
</tr>
<tr>
<td>0x804843b</td>
<td>mov 0xc(%ebp), %edx</td>
<td>804843b:</td>
<td>mov 0xc(%ebp), %edx</td>
</tr>
<tr>
<td>0x804844b</td>
<td>sar $0x1f, %edx</td>
<td>804844b:</td>
<td>sar $0x1f, %edx</td>
</tr>
<tr>
<td>0x8048443</td>
<td>idivl 0x10(%ebp)</td>
<td>8048443:</td>
<td>idivl 0x10(%ebp)</td>
</tr>
<tr>
<td>0x8048446</td>
<td>add 0x10(%ebp), %eax</td>
<td>8048446:</td>
<td>add 0x10(%ebp), %eax</td>
</tr>
<tr>
<td>0x8048449</td>
<td>jmp 8048453</td>
<td>8048449:</td>
<td>jmp 8048453</td>
</tr>
<tr>
<td>0x804844b</td>
<td>mov $0x1, %eax</td>
<td>804844b:</td>
<td>mov $0x1, %eax</td>
</tr>
<tr>
<td>0x8048450</td>
<td>sub 0x10(%ebp), %eax</td>
<td>8048450:</td>
<td>sub 0x10(%ebp), %eax</td>
</tr>
<tr>
<td>0x8048453</td>
<td>pop %ebp</td>
<td>8048453:</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>0x8048454</td>
<td>ret</td>
<td>8048454:</td>
<td>ret</td>
</tr>
</tbody>
</table>
Summarizing

- **C Control**
  - if-then-else
  - do-while
  - while
  - for
  - switch
- **Assembler Control**
  - jump
  - Conditional jump
- **Compiler**
  - Must generate assembly code to implement more complex control

*With Slides from Bryant and O'Halloran*