Machine-Level Programming II:
Arithmetic and Control

CSci 2021: Machine Architecture and Organization
Lectures #8-9, February 6th-9th, 2015
Your Instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O’Hallaron, Antonia Zhai

Arithmetic and Control

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- Control: Condition codes
- Conditional branches
- While loops
- Switch statements

Complete Memory Addressing Modes

- **Most General Form**
  - \( D(R_b, R_i, S) \) \[ Mem[Reg[R_b] + S*Reg[R_i] + D] \]
    - \( D \): Constant "displacement" 1, 2, or 4 bytes
    - \( R_b \): Base register: Any of 8 integer registers
    - \( R_i \): Index register: Any, except for \%esp
    - \( S \): Scale: 1, 2, 4, or 8 (why these numbers?)

- **Special Cases**
  - \( (R_b, R_i) \) \[ Mem[Reg[R_b] + Reg[R_i]] \]
  - \( D(R_b, R_i) \) \[ Mem[Reg[R_b] + Reg[R_i] + D] \]
  - \( (R_b, R_i, S) \) \[ Mem[Reg[R_b] + S*Reg[R_i]] \]

Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0x8(%edx, 2) )</td>
<td>( 2*0xf000 + 0x80 )</td>
<td>( 0x1e080 )</td>
</tr>
<tr>
<td>( 0x80(%edx, 1) )</td>
<td>( 2*0xf000 + 0x80 )</td>
<td>( 0xf0080 )</td>
</tr>
<tr>
<td>( 0x8(%edx) )</td>
<td>( 0xf000 + 0x8 )</td>
<td>( 0xf008 )</td>
</tr>
<tr>
<td>( (%edx, %ecx) )</td>
<td>( 0xf000 + 0x100 )</td>
<td>( 0xf100 )</td>
</tr>
<tr>
<td>( (%edx, %ecx, 4) )</td>
<td>( 0xf000 + 4*0x100 )</td>
<td>( 0xf400 )</td>
</tr>
</tbody>
</table>

Address Computation Instruction

- **leal** \( \text{Src, Dest} \)
  - \( \text{Src} \) is address mode expression
  - \( \text{Set Dest to address denoted by expression} \)

- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of \( p = 4x[1] \);
  - Computing arithmetic expressions of the form \( x + k*y \)
    - \( k = 1, 2, 4, \) or \( 8 \)

- **Example**

```c
int mul12(int x) {
    return x*12;
}
```

Converted to ASM by compiler:

```asm
leal (%eax,%eax,2), %eax  ;t < - x*x*2
sall $2, %eax             ;return t<<2
```

Arithmetic and Control

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- Control: Condition codes
- Conditional branches
- While loops
- Switch statements
Two Operand Instructions:

- add: Dest = Dest + Src
- sub: Dest = Dest - Src
- imul: Dest = Dest * Src
- sal: Dest = Dest << Src
- sar: Dest = Dest >> Src
- xor: Dest = Dest ^ Src
- or: Dest = Dest | Src

- Watch out for argument order!
- No distinction between signed and unsigned int (why?)

Arithmetic Expression Example:

```c
#include <stdio.h>

int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = x+z;  // Also called shift
    int t3 = x+4;
    int t4 = y*48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

Understanding arith:

```c
#include <stdio.h>

int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = x+z;
    int t3 = x+4;
    int t4 = y*48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

Observations about arith:

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code when compile:
  `(x+y+z)*(x+48+y)`
```c
int logical(int x, int y) {
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

```c
int logical(int x, int y) {
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

**Arithmetic and Control**

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- Control: Condition codes
- Conditional branches
- Loops
- Switch statements

**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)

- General purpose registers
- Current stack top
- Current stack frame
- Instruction pointer
- Condition codes
Condition Codes (Implicit Setting)

- Single bit registers
  - CF: Carry Flag (for unsigned)
  - SF: Sign Flag (for signed)
- Implicitly set (think of it as side effect) by arithmetic operations
  - Example: `addl`/`addq` `Src, Dest` → `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
- Intel documentation, others, have full details

Condition Codes (Explicit Setting: Compare)

- Explicit Setting by Compare Instruction
  - `cmpl/cmpq` `Src2, Src1`
  - `cmpl 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`
  - `testl 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 combination 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 (&quot;Sign&quot;)</td>
</tr>
<tr>
<td>setns</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 &gt;)</td>
</tr>
<tr>
<td>setae</td>
<td>~CF</td>
<td>Above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned &lt;)</td>
</tr>
<tr>
<td>setbe</td>
<td>CF</td>
<td>ZF</td>
</tr>
</tbody>
</table>

Reading Condition Codes: Cont.

- `SetX` Instructions:
  - Set single byte based on combination of condition codes
  - One of 8 addressable byte registers
    - Does not alter remaining 3 bytes
    - Typically use `movzbl` to finish job

  **Body**

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

  **Bodies**

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

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

  ```
  movq 12(%rdi), %rax # rax = y
  cmpl %rax,8(%rdi) # Compare x : y
  setg %al
  movzbl %al, %rax # Zero rest of %rax
  ```

  ```
  cmpq 12(%rdi), %rax # Compare x : y
  setg %al
  movzbl %al, %rax # Zero rest of %rax
  ```

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

  ```
  cmpq 12(%rdi), %rax # Compare x : y
  setg %al
  movzbl %al, %rax # Zero rest of %rax
  ```

  ```
  cmpq 12(%rdi), %rax # Compare x : y
  setg %al
  movzbl %al, %rax # Zero rest of %rax
  ```

  ```
  Is %rax zero?
  Yes: 32-bit instructions set high order 32 bits to 0!
  ```
2/11/2015

Exercise Break: More Conditions

- Every condition can be negated by putting "n" in the mnemonic, for "not"
  - We skipped some of these conditions in the previous tables, because they were equivalent to others
- Which other conditions are these equivalent to?
  1. setng: not greater than
  2. setnbe: not below or equal

Equivalents of More Conditions

- Intuition: cover three cases: <, =, >
- setng: not greater than (signed)
  - If not greater, then either less than or equal:
  - setle
  - Check conditions:
    - ~(~SF ^ OF) & ~ZF = ~ZF = (SF ^ OF) | ZF
- setnbe: not below or equal (unsigned)
  - If not below or equal, must be above:
  - seta
  - Check conditions:
    - ~CF & ~ZF

Arithmetic and Control

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches & Moves
- Loops
- Switch statements

Jumping

- JX instructions
  - Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>JX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>jeg</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)</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;
}
```

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
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;
}
```

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;
}
```

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;
}
```

Conditional Move Example: x86-64

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

Using Conditional Moves

- **Conditional Move Instructions**
  - Instruction supports:
    - if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC does not always use them
  - For compatibility with ancient processors:
    - Enabled for x86-64
    - Use switch -march=686 for IA32
  - Why?
    - Branches are very disruptive to instruction flow through pipelines
    - Conditional move do not require control transfer

### C Code

```c
C Code
Val = Test ? Then_Expr : Else_Expr;
Goto Version
nt = !Test;
if (nt) goto Else;
val = Then_Expr;
goto Done;
Else:
    val = Else_Expr;
    Done:
```

### Goto Version

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

### General Conditional Expression Translation

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

- x in %edi
- y in %esi
- absdiff movl %edi, %edx # tval = x-y
- subl %edi, %eax # result = y-x
- cmp %esi, %edi # Compare x-y
- cmovg ret # If >, result = tval
Bad Cases for Conditional Move

Expensive Computations

\[ \text{val} = \text{Test}(x) \oplus \text{Hard1}(x) : \text{Hard2}(x); \]
- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

\[ \text{val} = x > 0 ? x^7 : x^3; \]
- Both values get computed
- Must be side-effect free

Arithmetic and Control

- Complete addressing mode, address computation (lea)
- Arithmetic operations
- x86-64
- Control: condition codes
- Conditional branches and moves
- Loops
- Switch statements

"Do-While" Loop Example

C Code

```c
int pcount_do(unsigned x) {
  int result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```c
int pcount_do(unsigned x) {
  int result = 0;
  loop:
    result += x & 0x1;
    x >>= 1;
    if (x)
      goto loop;
  return result;
}
```

- Count number of 1's in argument x ("popcount")
- Use conditional branch to either continue looping or to exit loop

"Do-While" Loop Compilation

Goto Version

```c
int pcount_do(unsigned x) {
  int result = 0;
  loop:
    result += x & 0x1;
    x >>= 1;
    if (x)
      goto loop;
  return result;
}
```

- Registers:
  - %edx
  - %ecx
  - %ecx
  - %eax

```

```
movl \$0, %ecx # result = 0
.L2: # loop:
  movl %edx, %eax
  addl \$1, %eax # t = x & 1
  addl %eax, %ecx # result += t
  shrl %edx # x >>= 1
  jne .L2 # If !0, goto loop
```

General "Do-While" Translation

C Code

```c
do
  Body (Test);
while (Test);
```

- Body:
  - Statement;
  - Statement;
  - ...
  - Statement;

Goto Version

```c
loop:
  Body
  if (Test)
    goto loop
```

- Test returns integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true

"While" Loop Example

C Code

```c
int pcount_while(unsigned x) {
  int result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Goto Version

```c
int pcount_do(unsigned x) {
  int result = 0;
  if (!x)
    goto done;
  loop:
    result += x & 0x1;
    x >>= 1;
    if (x)
      goto loop;
  done:
    return result;
}
```

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

**“For” Loop Example**

C Code
```
#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
```
1 = 0
```

Test
```
1 < WSIZE
```

Update
```
i++
```

Body
```
unsigned mask = 1 << i;
result += (x & mask) != 0;
```

**“For” Loop → While Loop**

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

While Version
```
Init:
while (Test) {
    Body
}
```

Init
```
i = 0
```

Test
```
i < WSIZE
```

Update
```
i++;
```

Body
```
unsigned mask = 1 << i;
result += (x & mask) != 0;
```

**“For” Loop Conversion Example**

C Code
```
#define WSIZE 8*sizeof(int)
int pcount_for_gt(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

```
#define WSIZE 8*sizeof(int)
int pcount_for_gt(unsigned x) {
    int i;
    int result = 0;
    if (!Test)
        goto done;
    do
    Body
    while (Test);
done:
```

```
#define WSIZE 8*sizeof(int)
int pcount_for_gt(unsigned x) {
    int i;
    int result = 0;
    if (i < WSIZE)
        goto loop;
    do
    Body
    while (Test);
done:
```

```
#define WSIZE 8*sizeof(int)
int pcount_for_gt(unsigned x) {
    int i;
    int result = 0;
    if (i < WSIZE)
        goto loop;
    do
    Body
    while (Test);
done:
```

```
#define WSIZE 8*sizeof(int)
int pcount_for_gt(unsigned x) {
    int i;
    int result = 0;
    if (i < WSIZE)
        goto loop;
    do
    Body
    while (Test);
done:
```
Announcement Break: Bomb Lab Now Out

- Analyze malicious software with a debugger
  - Reverse engineering based on instructions, observation, and experiment
  - Find inputs to "defuse" a bomb program so it does not "explode"
- We've covered enough material for you to start working now
  - E.g., control flow structure and arithmetic
  - Will also cover in discussion sections tomorrow
- Like data lab, difficulty increases between parts
  - Last phase especially complex
  - Start early!

Arithmetic and Control

- Complete addressing mode, address computation (lea)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches and moves
- Loops
- Switch statements

Switch Statement Example

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

Switch Statement Example (IA32)

```
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 */
            break;
        case 5:
            case 6:
                w = x;
                break;
            default:
                w = 2;
    }
    return w;
}
```

Switch Statement Example (IA32)

```
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 = x;
                break;
        case 5:
        case 6:
            w = x;
            break;
        default:
            w = 2;
    }
    return w;
}
```

Jump Table Structure

```
Jump Form
switch(x) {
    case val_0:
        Block
    case val_1:
        Block
    ...
    case val_n-1:
        Block
}
```

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;
}
```

Indirect jump table
```
section .rodata
.align 4
.L7:
.long L2 # x = 0
.long L3 # x = 1
.long L4 # x = 2
.long L5 # x = 3
.long L6 # x = 4
```

Note that w not initialized here
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

```c
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:  // .L6
        w = 1;
        goto merge;
    case 2:  // .L4
        w = y/z;
        /* Fall Through */
    case 3:  // .L5
        w += z;
        break;
    case 5:  // .L6
        w = 1;
        goto merge;
    default:    // .L2
        w = 2;
        break;
}
```

Handling Fall-Through

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

Code Blocks (Partial)

```
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 # w = y+z
    jmp .L8 # Goto done
L6:  # w = 1
    subl 16(%ebp), %eax # w = 1-z
    jmp .L8 # Goto done
```

Code Blocks (Rest)

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

Switch Code (Finish)

```
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
x86-64 Switch Implementation

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

```
section .rodata
.align 8
.L7: .quad .12 # x = 0
    .quad .13 # x = 1
    .quad .14 # x = 2
    .quad .15 # x = 3
    .quad .16 # x = 4
    .quad .16 # x = 5
    .quad .16 # x = 6
```

Jump Table

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

Jump Table (cont.)

```
Label .L2 becomes address 0x8048422
Label .L7 becomes address 0x8048660
```

IA32 Object Code

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

Assembly Code

```
switch_ege:
    . . .

switch_eg:
    . . .
```

Disassembled Object Code

```
0x8048410 <switch_ege>:
   . . .

0x8048419: 77 07    ja 0x8048422 <switch_ege+0x12>
0x804841b: ff 24 85 60 86 04 08    jmp *0x8048660(,%eax,4)
```

Disassembled Targets

```
0x8048422: mov $0x2,teax
0x8048427: jmp 0x804843f <switch_eg+0x13>
0x8048429: mov $0x1,teax
0x804842e: xchg %ax,%ax  # noop
0x8048430: jmp 0x8048446 <switch_eg+0x36>
0x8048432: mov 0x10(%ebp),%eax
0x8048435: imulq 0xc(%ebp),%eax
0x8048439: add 0x10(%ebp),%eax
0x804843b: mov 0xc(%ebp),%edx
0x804843e: mov %edx,%eax
0x8048440: sar $0x1f,%edx
0x8048443: idivl 0x10(%ebp)
0x8048446: add 0x10(%ebp),%edx
```

Matching Disassembled Targets

```
0x8048422: mov $0x2,teax
0x8048427: jmp 0x804843f <switch_eg+0x13>
0x8048429: mov $0x1,teax
0x804842e: xchg %ax,%ax  # noop
0x8048430: jmp 0x8048446 <switch_eg+0x36>
0x8048432: mov 0x10(%ebp),%eax
0x8048435: imulq 0xc(%ebp),%eax
0x8048439: add 0x10(%ebp),%eax
0x804843b: mov 0xc(%ebp),%edx
0x804843e: mov %edx,%eax
0x8048440: sar $0x1f,%edx
0x8048443: idivl 0x10(%ebp)
0x8048446: add 0x10(%ebp),%edx
```

IA32 Object Code (cont.)

- Doesn’t show up in disassembled code
- Can inspect using GDB (or objdump -s)
- (gdb) x/7xw 0x8048660
  * Examine 7 hexadecimal format “words” (4-bytes each)
  * Use command “help x” to get format documentation

Assembly Code

```
switch_ege:
    . . .

switch_eg:
    . . .
```

Disassembled Object Code

```
0x8048419: 77 07    ja 0x8048422 <switch_ege+0x12>
0x804841b: ff 24 85 60 86 04 08    jmp *0x8048660(,%eax,4)
```

Disassembled Targets

```
0x8048422: mov $0x2,teax
0x8048427: jmp 0x804843f <switch_eg+0x13>
0x8048429: mov $0x1,teax
0x804842e: xchg %ax,%ax  # noop
0x8048430: jmp 0x8048446 <switch_eg+0x36>
0x8048432: mov 0x10(%ebp),%eax
0x8048435: imulq 0xc(%ebp),%eax
0x8048439: add 0x10(%ebp),%eax
0x804843b: mov 0xc(%ebp),%edx
0x804843e: mov %edx,%eax
0x8048440: sar $0x1f,%edx
0x8048443: idivl 0x10(%ebp)
0x8048446: add 0x10(%ebp),%edx
```

Matching Disassembled Targets

```
0x8048422: mov $0x2,teax
0x8048427: jmp 0x804843f <switch_eg+0x13>
0x8048429: mov $0x1,teax
0x804842e: xchg %ax,%ax  # noop
0x8048430: jmp 0x8048446 <switch_eg+0x36>
0x8048432: mov 0x10(%ebp),%eax
0x8048435: imulq 0xc(%ebp),%eax
0x8048439: add 0x10(%ebp),%eax
0x804843b: mov 0xc(%ebp),%edx
0x804843e: mov %edx,%eax
0x8048440: sar $0x1f,%edx
0x8048443: idivl 0x10(%ebp)
0x8048446: add 0x10(%ebp),%edx
```
Exercise Break: switch Bounds

- Every jump table needs to check that the index is in bounds
  - For each of these code patterns, what indexes are allowed?

- `cmpl $5, %eax
  ja .Ldefault
  jmp *.L1(%eax,4)`
  - Unsigned <= 5: 0 .. 5

- `andl $7, %eax
  jmp *.L2(%eax,4)`
  - Low 3 bits: 0 .. 7

- `movzlq 8(%ebp), %eax
  jmp *.L3(%eax,4)`
  - Low 8 bits: 0 .. 255

Summarizing

- **C Control**
  - if-then-else
  - do-while
  - while, for
  - switch

- **Assembler Control**
  - Conditional jump
  - Conditional move
  - Indirect jump
  - Compiler generates code sequence to implement more complex control

- **Standard Techniques**
  - Loops converted to do-while form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees

Summary

- **These slides**
  - Complete addressing mode, address computation (lea)
  - Arithmetic operations
  - Control: Condition codes
  - Conditional branches & conditional moves
  - Loops
  - Switch statements

- **Next Up**
  - Stack
  - Call / return
  - Procedure call discipline