15-213

"The course that gives CMU its Zip!"

Machine-Level Programming II: Control Flow Sept. 09, 2008

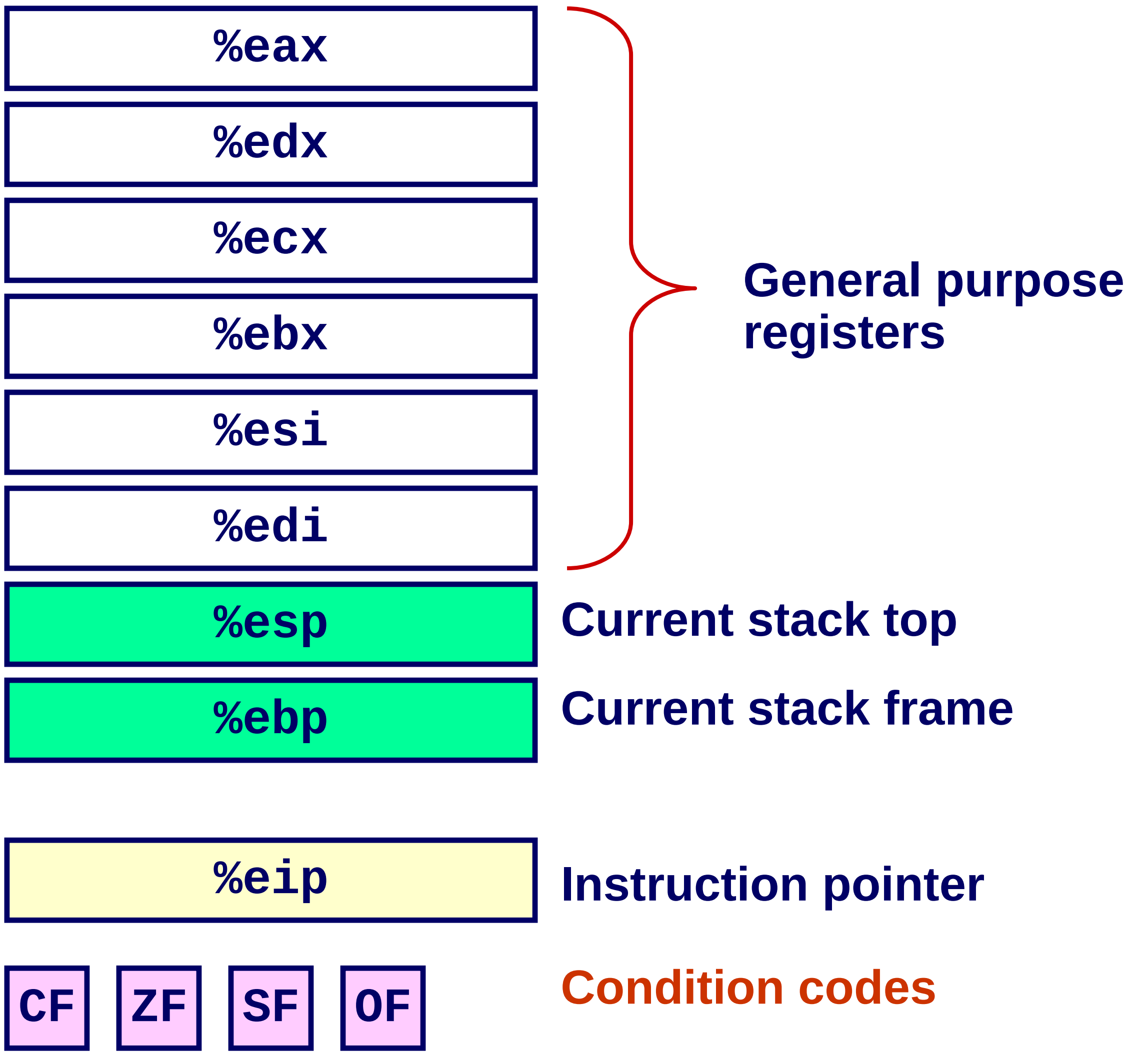
Topics

- Condition Codes
  - Setting
  - Testing
- Control Flow
  - If-then-else
  - Varieties of Loops
  - Switch Statements
- x86-64 features
  - conditional move
  - different loop implementation

Processor State (IA32, Partial)

Information about currently executing program

- Temporary data
- Location of current code control point
- Location of runtime stack
- Status of recent tests



Condition Codes

Single Bit Registers

CF Carry Flag **SF** Sign Flag

ZF Zero Flag **OF** Overflow Flag

Implicitly Set By Arithmetic Operations

`addl` *Src*,*Dest* `addq` *Src*,*Dest*

C analog: `t = a + b` (a = Src, b = Dest)

- CF set if carry out from most significant bit
  - Used to detect unsigned overflow
- ZF set if `t == 0`
- SF set if `t < 0`
- OF set if two's complement overflow

```
(a>0 && b>0 && t<0)
|| (a<0 && b<0 && t>=0)
```

Not set by `lea`, `inc`, or `dec` instructions

Setting Condition Codes (cont.)

Explicit Setting by Compare Instruction

`cmpl` ***Src2*,*Src1***   `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`
- OF set if two's complement overflow
  - `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

Setting Condition Codes (cont.)

Explicit Setting by Test instruction

`testl` ***Src2*,*Src1***

`testq` ***Src2*,*Src1***

- **Sets condition codes based on value of *Src1* & *Src2***
  - **Useful to have one of the operands be a mask**
- **`testl b,a` like computing a&b without setting destination**
- **ZF set when a&b `== 0`**
- **SF set when a&b `< 0`**

Reading Condition Codes

SetX Instructions

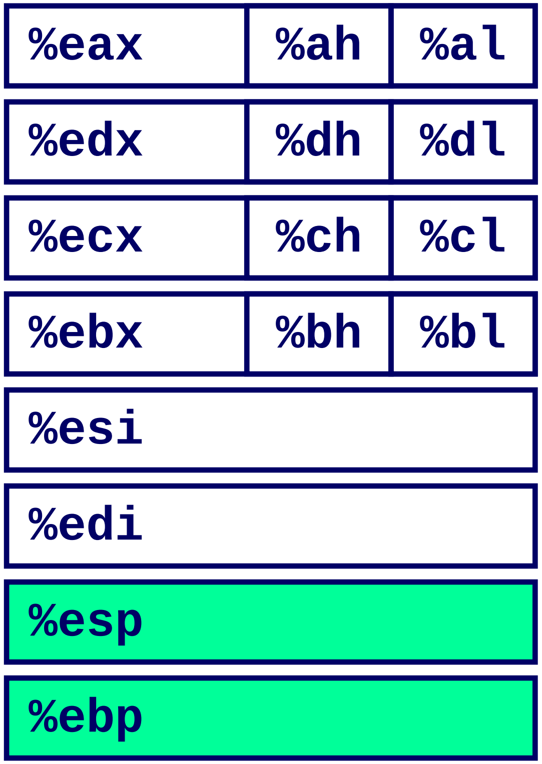
- Set single byte based on combinations of condition codes

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

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




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

Body

```
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 combinations of condition codes
  - Does not alter remaining 7 bytes

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

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

- x86-64 arguments
  - `x` in `%rdi`
  - `y` in `%rsi`

Body (same for both)

(32-bit instructions set high order 32 bits to 0)

```
xorl %eax, %eax      # eax = 0
cmpq %rsi, %rdi      # Compare x : y
setg %al             # al = x > y
```

Jumping

jX Instructions

- **Jump to different part of code depending on condition codes**

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

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           }  Set
   movl    8(%ebp), %edx        }  Up
   movl    12(%ebp), %eax       }
   cmpl    %eax, %edx           }
   jle     .L7                  }  Body1
   subl    %eax, %edx           }
   movl    %edx, %eax           }
.L8:
   leave                        }  Finish
   ret                          }
.L7:
   subl    %edx, %eax           }  Body2
   jmp     .L8                  }
```

Conditional Branch Example (Cont.)

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

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

Body1

```
   # x in %edx, y in %eax
   cmpl   %eax, %edx     # Compare x:y
   jle    .L7            # <= Goto Else
   subl   %eax, %edx     # x-= y
   movl   %edx, %eax     # result = x
.L8:  # Exit:
```

Body2

```
.L7:  # Else:
   subl   %edx, %eax     # result = y-x
   jmp    .L8            # Goto Exit
```

General Conditional Expression Translation

C Code

```
val = Test ? Then-Expr : Else-Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
  nt = !Test;
  if (nt) goto Else;
  val = Then-Expr;
Done:
  . . .
Else:
  val = Else-Expr;
  goto Done;
```

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

Conditionals: x86-64

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

```
absdiff: # x in %edi, y in %esi
  movl   %edi, %eax  # v   = x
  movl   %esi, %edx  # ve  = y
  subl   %esi, %eax  # v  -= y
  subl   %edi, %edx  # ve -= x
  cmpl   %esi, %edi  # x:y
  cmovle %edx, %eax  # v=ve if <=
  ret
```

- Conditional move instruction
  - cmov*C* src, dest
  - Move value from src to dest if condition *C* holds
  - More efficient than conditional branching
    - » Simple & predictable control flow

General Form with Conditional Move

C Code

```
val = Test ? Then-Expr ? Else-Expr;
```

Conditional Move Version

```
  val  = Then-Expr;
  vale = Else-Expr;
  val  = vale if !Test;
```

- **Both values get computed**
- **Overwrite then-value with else-value if condition doesn't hold**

Limitations of Conditional Move

```
  val  = Then-Expr;
  vale = Else-Expr;
  val  = vale if !Test;
```

```
int xgty  = 0, xltey = 0;

int absdiff_se(
    int x, int y)
{
    int result;
    if (x > y) {
        xgty++;  result = x-y;
    } else {
        xltey++; result = y-x;
    }
    return result;
}
```

Don’t use when:

- **Then-Expr or Else-Expr has side effect**
- **Then-Expr or Else-Expr requires significant computation**

Implementing Loops

IA32

- All loops translated into form based on “do-while”

x86-64

- Also make use of “jump to middle”

Why the Difference

- IA32 compiler developed for machine where all operations costly
- x86-64 compiler developed for machine where unconditional branches incur (almost) no overhead

"Do-While" Loop Example

C Code

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

  return result;
}
```

Goto Version

```
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

Registers

`%edx`	`x`
`%eax`	`result`

Goto Version

```
int
fact_goto(int x)
{
  int result = 1;


loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;

  return result;
}
```

Assembly

```
fact_goto:
  pushl %ebp            # Setup
  movl %esp,%ebp        # Setup
  movl $1,%eax          # eax = 1
  movl 8(%ebp),%edx     # edx = x

L11:
  imull %edx,%eax       # result *= x
  decl %edx             # x--
  cmpl $1,%edx          # Compare x : 1
  jg L11                # if > goto loop

  movl %ebp,%esp        # Finish
  popl %ebp             # Finish
  ret                   # Finish
```

General "Do-While" Translation

C Code

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

Goto Version

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

- *Body* can be any C statement
  - Typically compound statement:

```
{
  Statement1;
  Statement2;
    …
  Statementn;
}
```

- *Test* is expression returning integer

  = 0 interpreted as false   ≠0 interpreted as true

"While" Loop Example #1

C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {


    result *= x;
    x = x-1;
  };

  return result;
}
```

First Goto Version

```
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**

Alternative "While" Loop Translation

C Code

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

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

Second Goto Version

```
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;
}
```

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:
```

New Style "While" Loop Translation

C Code

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

Goto Version

```
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;
}
```

- Recent technique for GCC
  - Both IA32 & x86-64
- First iteration jumps over body computation within loop

Jump-to-Middle While Translation

C Code

```
while (Test)
  Body
```

Goto Version

```
goto middle;
loop:
  Body
middle:
  if (Test)
    goto loop;
```

- Avoids duplicating test code
- Unconditional `goto` incurs no performance penalty
- `for` loops compiled in similar fashion

Jump-to-Middle Example

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

- Most common strategy for recent IA32 & x86-64 code generation

```
# x in %edx, result in %eax
  jmp   L34          #   goto Middle
L35:                 # Loop:
  imull %edx, %eax   #   result *= x
  decl  %edx         #   x--
L34:                 # Middle:
  cmpl  $1, %edx     #   x:1
  jg    L35          #   if >, goto Loop
```

"For"→ "While"→ "Do-While"

For Version

```
for (Init; Test; Update )

    Body
```

While Version

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

Do-While Version

```
  Init;
  if (!Test)
    goto done;
  do {
    Body
    Update ;
  } while (Test)
done:
```

Goto Version

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

"For"→ "While" (Jump-to-Middle)

For Version

```
for (Init; Test; Update )

    Body
```

While Version

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

Goto Version

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

Switch Statements

Implementation Options

- Series of conditionals
  - Organize in tree structure
  - Logarithmic performance
- Jump Table
  - Lookup branch target
  - Constant time
  - Possible when cases are small integer constants
- GCC
  - Picks one based on case structure

Switch Statement Example

```
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;
}
```

Features

- Multiple case labels
- Fall through cases
- Missing cases

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

jtab:

Targ0
Targ1
Targ2
• • •
Targn-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

• • •

Targn-1: Code Block *n*–1

Approx. 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
   pushl %ebx              # Setup
   movl  $1, %ebx          # w = 1
   movl  8(%ebp), %edx     # edx = x
   movl  16(%ebp), %ecx    # ecx = z
   cmpl  $6, %edx          # x:6
   ja    .L61              # if > goto default
   jmp   *.L62(,%edx,4)    # goto JTab[x]
```

Assembly Setup Explanation

Table Structure

- **Each target requires 4 bytes**
- **Base address at `.L62`**

Jumping

`jmp .L61`

- Jump target is denoted by label **`.L61`**

`jmp *.L62(,%edx,4)`

- Start of jump table denoted by label **`.L62`**
- Register **`%edx`** holds **`x`**
- Must scale by factor of 4 to get offset into table
- Fetch target from effective Address **`.L61 + x*4`**
  - Only for $0 \leq \mathbf{x} \leq 6$

Jump Table

Table Contents

```
.section .rodata
   .align 4
.L62:
  .long   .L61  # x = 0
  .long   .L56  # x = 1
  .long   .L57  # x = 2
  .long   .L58  # x = 3
  .long   .L61  # x = 4
  .long   .L60  # x = 5
  .long   .L60  # x = 6
```

```
switch(x) {
case 1:         // .L56
    w = y*z;
    break;
case 2:         // .L57
    w = y/z;
    /* Fall Through */
case 3:         // .L58
    w += z;
    break;
case 5:
case 6:         // .L60
    w -= z;
    break;
default:        // .L61
    w = 2;
}
```

Code Blocks (Partial)

```
switch(x) {
  . . .
case 2:        // .L57
    w = y/z;
    /* Fall Through */
case 3:        // .L58
    w += z;
    break;
  . . .
default:       // .L61
    w = 2;
}
```

```
.L61:  // Default case
   movl  $2, %ebx     # w = 2
   movl  %ebx, %eax   # Return w
   popl  %ebx
   leave
   ret
.L57:  // Case 2:
   movl  12(%ebp), %eax  # y
   cltd               # Div prep
   idivl %ecx         # y/z
   movl  %eax, %ebx # w = y/z
# Fall through
.L58:  // Case 3:
   addl  %ecx, %ebx # w+= z
   movl  %ebx, %eax # Return w
   popl  %ebx
   leave
   ret
```

Code Blocks (Rest)

```
switch(x) {
case 1:         // .L56
    w = y*z;
    break;
  . . .
case 5:
case 6:         // .L60
    w -= z;
    break;
  . . .
}
```

```
.L60: // Cases 5&6:
   subl  %ecx, %ebx  # w -= z
   movl  %ebx, %eax  # Return w
   popl  %ebx
   leave
   ret
.L56: // Case 1:
   movl  12(%ebp), %ebx # w = y
   imull %ecx, %ebx     # w*= z
   movl  %ebx, %eax  # Return w
   popl  %ebx
   leave
   ret
```

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:       // .L50
    w = y*z;
    break;
  . . .
}
```

Jump Table

```
.section .rodata
   .align 8
.L62:
  .quad   .L55  # x = 0
  .quad   .L50  # x = 1
  .quad   .L51  # x = 2
  .quad   .L52  # x = 3
  .quad   .L55  # x = 4
  .quad   .L54  # x = 5
  .quad   .L54  # x = 6
```

```
.L50: // Case 1:
   movq   %rsi, %r8  # w = y
   imulq  %rdx, %r8  # w *= z
   movq   %r8, %rax  # Return w
   ret
```

IA32 Object Code

Setup

- Label `.L61` becomes address `0x8048630`
- Label `.L62` becomes address `0x80488dc`

Assembly Code

```
switch_eg:
  . . .
  ja    .L61              # if > goto default
  jmp   *.L62(,%edx,4)  # goto JTab[x]
```

Disassembled Object Code

```
08048610 <switch_eg>:
 . . .
 8048622:  77 0c                    ja    8048630
 8048624:  ff 24 95 dc 88 04 08    jmp   *0x80488dc(,%edx,4)
```

IA32 Object Code (cont.)

Jump Table

- Doesn't show up in disassembled code
- Can inspect using GDB

```
gdb asm-cntl
(gdb) x/7xw 0x80488dc
```

  - E*x*amine *7* he*x*adecimal format "*w*ords" (4-bytes each)
  - Use command "`help x`" to get format documentation

```
0x80488dc:
  0x08048630
  0x08048650
  0x0804863a
  0x08048642
  0x08048630
  0x08048649
  0x08048649
```

Disassembled Targets

```
8048630:    bb 02 00 00 00    mov    $0x2,%ebx
8048635:    89 d8             mov    %ebx,%eax
8048637:    5b                pop    %ebx
8048638:    c9                leave
8048639:    c3                ret
804863a:    8b 45 0c          mov    0xc(%ebp),%eax
804863d:    99                cltd
804863e:    f7 f9             idiv   %ecx
8048640:    89 c3             mov    %eax,%ebx
8048642:    01 cb             add    %ecx,%ebx
8048644:    89 d8             mov    %ebx,%eax
8048646:    5b                pop    %ebx
8048647:    c9                leave
8048648:    c3                ret
8048649:    29 cb             sub    %ecx,%ebx
804864b:    89 d8             mov    %ebx,%eax
804864d:    5b                pop    %ebx
804864e:    c9                leave
804864f:    c3                ret
8048650:    8b 5d 0c          mov    0xc(%ebp),%ebx
8048653:    0f af d9          imul   %ecx,%ebx
8048656:    89 d8             mov    %ebx,%eax
8048658:    5b                pop    %ebx
8048659:    c9                leave
804865a:    c3                ret
```

Matching Disassembled Targets

0x08048630
0x08048650
0x0804863a
0x08048642
0x08048630
0x08048649
0x08048649

```
8048630:   bb 02 00 00 00      mov
8048635:   89 d8               mov
8048637:   5b                  pop
8048638:   c9                  leave
8048639:   c3                  ret
804863a:   8b 45 0c            mov
804863d:   99                  cltd
804863e:   f7 f9               idiv
8048640:   89 c3               mov
8048642:   01 cb               add
8048644:   89 d8               mov
8048646:   5b                  pop
8048647:   c9                  leave
8048648:   c3                  ret
8048649:   29 cb               sub
804864b:   89 d8               mov
804864d:   5b                  pop
804864e:   c9                  leave
804864f:   c3                  ret
8048650:   8b 5d 0c            mov
8048653:   0f af d9            imul
8048656:   89 d8               mov
8048658:   5b                  pop
8048659:   c9                  leave
804865a:   c3                  ret
```

x86-64 Object Code

Setup

- Label `.L61` becomes address `0x0000000000400716`
- Label `.L62` becomes address `0x0000000000400990`

Assembly Code

```
switch_eg:
  . . .
   ja    .L55              # if > goto default
   jmp   *.L56(,%rdi,8)  # goto JTab[x]
```

Disassembled Object Code

```
0000000000400700 <switch_eg>:
 . . .
  40070d:  77 07                    ja     400716
  40070f:  ff 24 fd 90 09 40 00   jmpq   *0x400990(,%rdi,8)
```

x86-64 Object Code (cont.)

Jump Table

- Can inspect using GDB

```
gdb asm-cntl
(gdb) x/7xg 0x400990
```

  - E*x*amine 7 he*x*adecimal format "*g*iant words" (8-bytes each)
  - Use command "`help x`" to get format documentation

```
0x400990:
  0x0000000000400716
  0x0000000000400739
  0x0000000000400720
  0x000000000040072b
  0x0000000000400716
  0x0000000000400732
  0x0000000000400732
```

Sparse Switch Example

```
/* 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;
  }
}
```

- Not practical to use jump table
  - Would require 1000 entries
- Obvious translation into if-then-else would have max. of 9 tests

Sparse Switch Code (IA32)

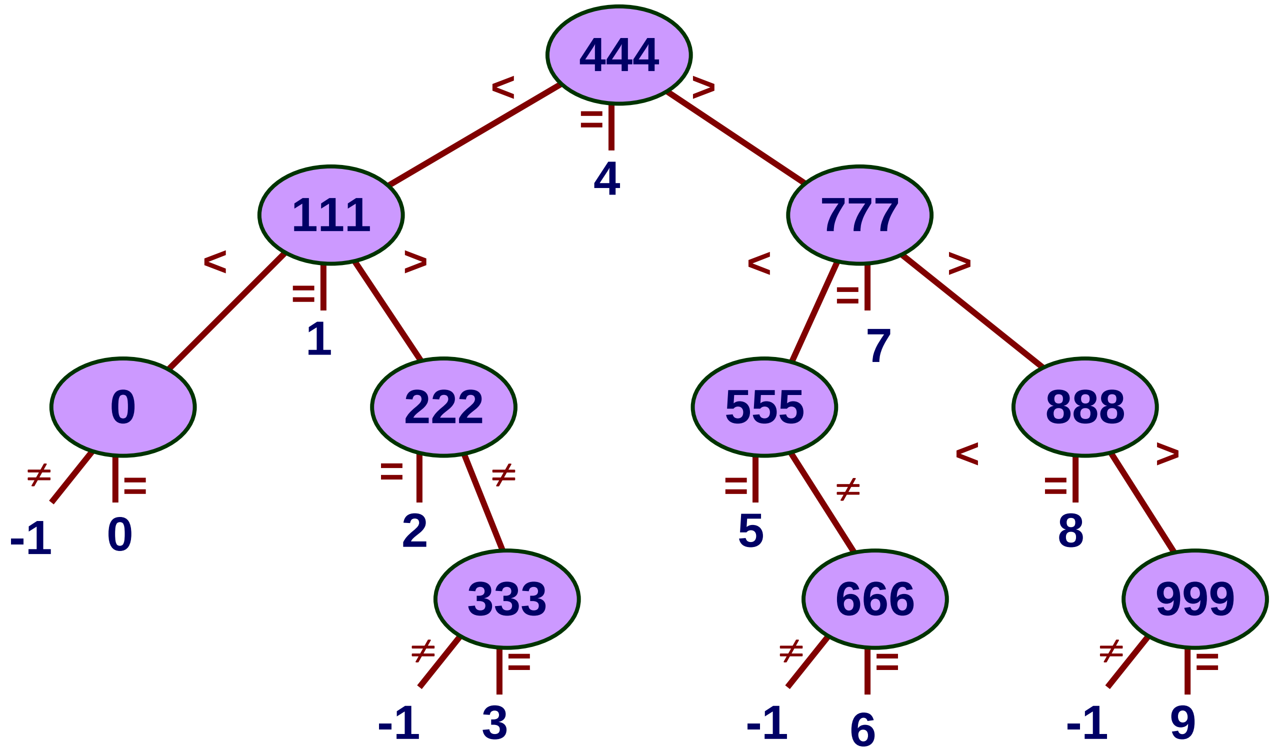
```
    movl 8(%ebp),%eax  # get x
    cmpl $444,%eax     # x:444
    je L8
    jg L16
    cmpl $111,%eax     # x:111
    je L5
    jg L17
    testl %eax,%eax    # x:0
    je L4
    jmp L14

    . . .
```

- Compares x to possible case values
- Jumps different places depending on outcomes

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

Sparse Switch Code Structure



- Organizes cases as binary tree
- Logarithmic performance

Summarizing

C Control

- if-then-else
- do-while
- while, for
- switch

Assembler Control

- Conditional jump
- Conditional move
- Indirect jump

Compiler

- Must generate assembly code to implement more complex control

Standard Techniques

- IA32 loops converted to do-while form
- x86-64 loops use jump-to-middle
- Large switch statements use jump tables

Conditions in CISC

- CISC machines generally have condition code registers