Machine-Level Programming: Control Flow

Chapter 3 of B&O

Some notes adopted from Bryant and O'Hallaron

Condition Codes

- Single Bit Registers
  - CF Carry Flag SF Sign Flag
  - ZF Zero Flag OF Overflow Flag

The carry flag and the overflow flag is for arithmetic overflo

- Implicitly Set By Arithmetic Operations
  - addl Src,Dest
  - C analog: t = a + b
  - 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 leal instruction

Setting Condition Codes (cont.)

- Explicit Setting by Compare Instruction
  - cmpl Src2,Src1
  - cmpl b,a like computing a-b without setting destination

    tests to see if they are equivalent without having to create a new register
  - 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
  - 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

%eax	%ah	%al
%edx	%dh	%dl
%ecx	%ch	%cl
%ebx	%bh	%bl
%esi		
%edi		
%esp		
%ebp		

```
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 combination of condition codes
  - Does not alter remaining 3 bytes

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

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

Body (same for both)

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

Is `%rax` zero?

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

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

*****HOW CAN YOU PULL INFORMATION FROM THE ASSEMLY CODE*****

Conditional Branch Example (Cont.)

```c
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;
}
```

```
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
```

x = edx

is x <= y if it is then jump to l7

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

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
  - ≠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   # eax = x
  movl    %esi, %edx   # edx = y
  subl    %esi, %eax   # eax = x-y
  subl    %edi, %edx   # edx = y-x
  cmpl    %esi, %edi   # x:y
  cmovle  %edx, %eax   # eax=edx 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 control flow)
  - But overhead: both branches are evaluated

General Form with Conditional Move

C Code

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

Conditional Move Version

```
val1  = Then-Expr;
val2  = Else-Expr;
val1  = val2 if !Test;
```

- Both values get computed
- Overwrite then-value with else-value if condition doesn’t hold
- Don’t use when:
  - Then or else expression have side effects
  - Then and else expression are too expensive

"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

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
```

- Registers

  `%edx` x

  `%eax` result

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 $\neq$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

Alternate "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
```

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

Goto Version

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

Goto (Previous) Version

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

Jump-to-Middle Example

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

```
# 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
```

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

"For" Loop Example: Square-and-Multiply

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p)
{
   int result;
      for (result = 1; p != 0; p = p>>1) {
         if (p & 0x1)
            result *= x;
         x = x*x;
      }
   return result;
}
```

- Algorithm
  - Exploit bit representation: $p = p_0 + 2p_1 + 2^2 p_2 + \ldots 2^{n-1} p_{n-1}$
  - Gives: $x^p = z_0 \cdot z_1{}^2 \cdot (z_2{}^2)^2 \cdot \ldots \cdot \underbrace{(\ldots((z_{n-1}{}^2)^2)\ldots)^2}_{n\text{–}1 \text{ times}}$

    $z_i = 1$ when $p_i = 0$

    $z_i = x$ when $p_i = 1$
  - Complexity $O(\log p)$

Example

$3^{10} = 3^2 * 3^8$

$= 3^2 * ((3^2)^2)^2$

ipwr Computation

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p)
{
   int result;
      for (result = 1; p != 0; p = p>>1) {
         if (p & 0x1)
            result *= x;
         x = x*x;
      }
   return result;
}
```

before iteration	result	x=3	p=10
1	1	3	$10=1010_2$
2	1	9	$5=101_2$
3	9	81	$2=10_2$
4	9	6561	$1=1_2$
5	59049	43046721	0

"For" Loop Example

```
int result;
for (result = 1; p != 0; p = p>>1)
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

General Form

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

Test

```
p != 0
```

Init

```
result = 1
```

Update

```
p = p >> 1
```

Body

```
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

"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-Loop: Compilation #1

For Version

```
for (Init; Test; Update )

    Body
```

Goto Version

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

```
for (result = 1; p != 0; p = p>>1)
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

```
  result = 1;
  if (p == 0)
    goto done;
loop:
  if (p & 0x1)
    result *= x;
  x = x*x;
  p = p >> 1;
  if (p != 0)
    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:
```

For-Loop: Compilation #2

For Version

```
for (Init; Test; Update )

    Body
```

Goto Version

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

```
for (result = 1; p != 0; p = p>>1)
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

```
  result = 1;
goto middle;
loop:
  if (p & 0x1)
    result *= x;
  x = x*x;
  p = p >> 1;
middle:
  if (p != 0)
    goto loop;
done:
```

Switch Statement Example

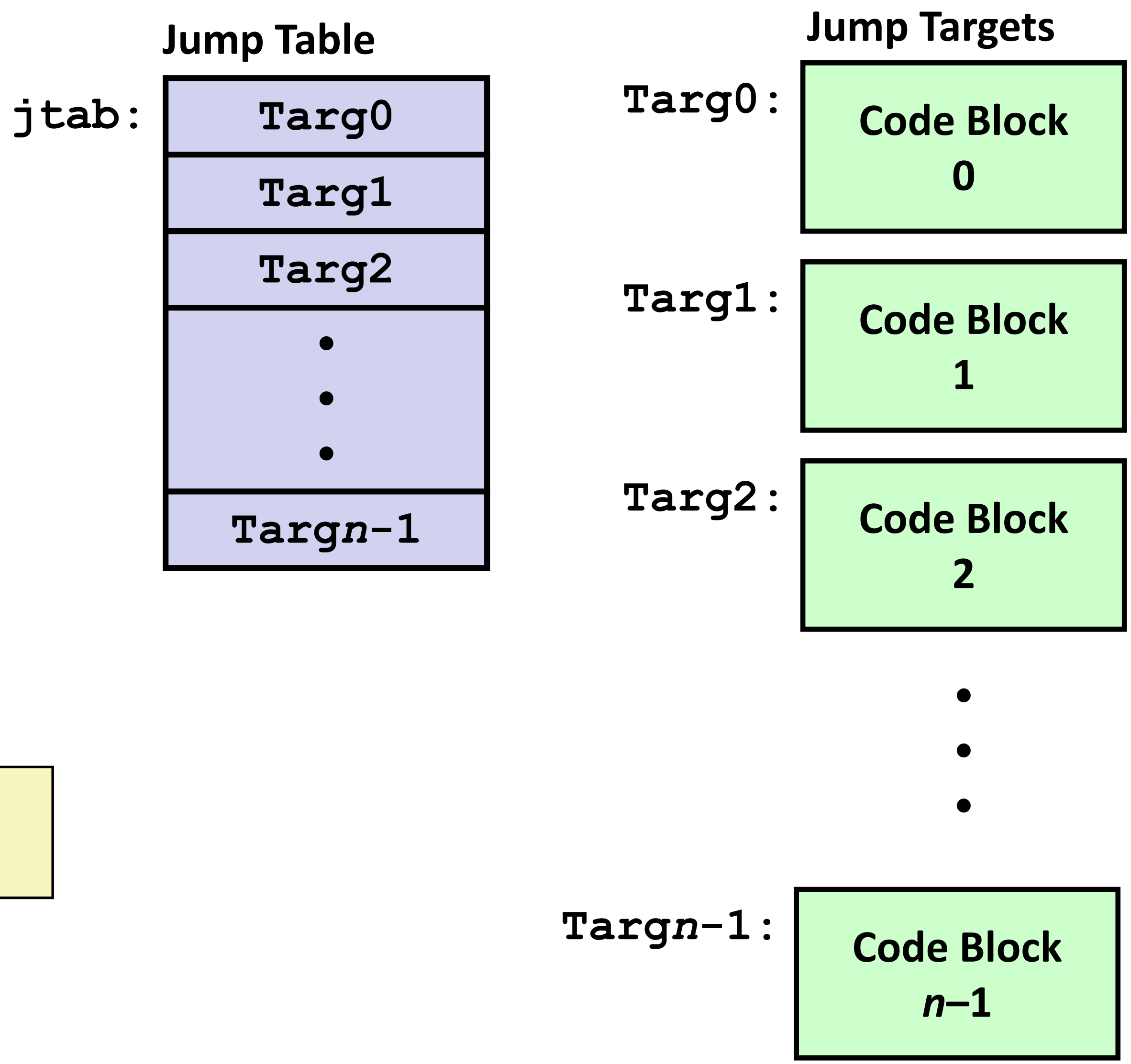
```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w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

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

Jump Table Structure

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```



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;
}
```

Jump table

```
        .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
```

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]
```

Indirect jump

Assembly Setup Explanation

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

- Jumping

  Direct: **`jmp .L61`**

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

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

  - Start of jump table: **`.L62`**
  - Must scale by factor of 4 (labels have 32-bit = 4 Bytes on IA32)
  - Fetch target from effective Address **`.L62 + edx*4`**
    - Only for $0 \le \mathbf{x} \le 6$

Jump table

```
.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
```

Jump Table

Jump table

```
.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;
  . . .
}
```

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

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
```

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
```

- Examine 7 hexadecimal format "words" (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
```

- Examine 7 hexadecimal format "giant 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)

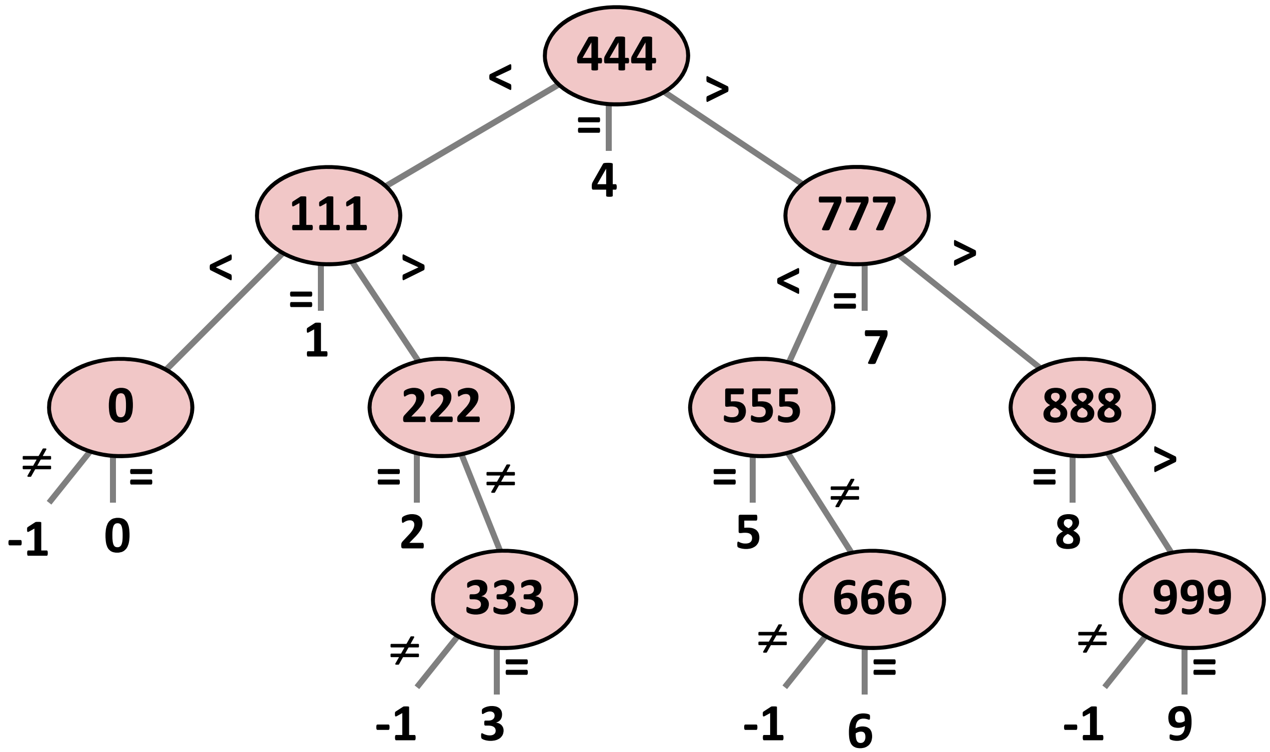
```
    movl 8(%ebp),%eax   # get x
    cmpl $444,%eax      # x:444
    je L8
    jg L16
    cmpl $111,%eax      # x:111
    je L5
    jg L17
    testl %eax,%eax     # x:0
    je L4
    jmp L14

    . . .
```

- 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
  - Sparse switch statements may use decision trees (not shown)
- Conditions in CISC
  - CISC machines generally have condition code registers