

# Today

- Lab 1 due
- Lab 2 handed out

# Last Time

- **IA32 history**
  - CISC vs. RISC
- **Assembly programmer's view of the machine**
- **Assembly language**
  - Addressing modes
  - Calling conventions
  - Registers
  - Arithmetic and logical operations
  - Machine control, OS support, I/O, etc.
- **Tools**
  - Preprocessor, compiler, assembler, linker
  - Debugger, disassembler

# Today:

## Machine-Level Programming II

### Control Flow

- **Condition Codes**
  - Setting
  - Testing
- **Control Flow in C and asm**
  - If-then-else
  - Varieties of loops
  - Switch statements

# One piece of Assembler Trickery

- **x86 instructions are 2 operand**
  - e.g. opcode S, D
    - » D is used twice in dyadic/binary operations (add, sub,...)
- **But a lot goes on w/ address modes**
  - e.g. `add 9(%eax, %edx, s), %ecx`
    - » causes  $9+eax+(s*edx)$  to be used as an address
  - `leal` instruction just saves that address somewhere
- **`leal (%edx, %ebx), %eax`**
  - is a tricky way to add two registers and place the value in a 3<sup>rd</sup> register
  - doesn't modify the CC's
- **You may see this in compiler output**

# x86 Condition Codes

- **Single Bit Registers**

CF	Carry Flag
ZF	Zero Flag
SF	Sign Flag
OF	Overflow Flag

- **All IA32 CPUs have these**

# x86 Condition Codes

- Implicitly set by arithmetic and logical operations
- Assembly: `addl a, b`
  - CF set if carry out from most significant bit
  - Used to detect unsigned overflow
- Analogous C code: `t = a + b`
- 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)`
- Condition codes *not* set by `leal` instruction

# Condition Codes

- **Explicit Setting by Compare Instruction**

`cmpl b, a`

- Effect is to compute `a-b` without setting destination
- CF set if carry out from most significant bit
  - » Used for unsigned comparisons
- ZF set if `a == b` (result of `a-b`  $\rightarrow$  0)
- SF set if `(a-b) < 0` (result is negative if `a < b`)
- OF set if two's complement overflow

`(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

- **Explicit Setting by Test instruction**

`testl b, a`

- Effect is to compute `(a & b)` without setting destination
- Sets condition codes based on value of `(a & b)`
  - » Useful to have one of the operands be a mask
- ZF set when `(a & b) == 0`
- SF set when `(a & b) < 0`

# Variants

- **x86 opcode options**

- **cmp for long, word, byte**

- » **cmpb S2, S1** → **S1 – S2**      byte subtract
    - » **cmpw S2, S1** → **S1 – S2**      word (16-bit) subtract
    - » **cmpl S2, S1** → **S1 – S2**      double word (32-bit) subtract

- **test for long, word, byte**

- » **testb S2, S1** → **S1 & S2**      byte
    - » **testw S2, S1** → **S1 & S2**      16-bit
    - » **testl S2, S1** → **S1 & S2**      32-bit

# Set Instructions

- **SetX Instructions (useful to read CC's): set<sub>xx</sub> D**
  - Set single byte based on combinations of condition codes

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

## cont'd

<b>SetX</b>	<b>Condition</b>	<b>Description</b>
setae	$\sim CF$	Above or Equal(unsigned)
setbe	$CF \ \& \ \sim ZF$	Below ore Equal (unsigned)

# Set Example

## • 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 `andl 0xFF,%eax` to finish job or `movzbl`

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

Body

```
movl 12(%ebp),%eax # eax = y
cmpl %eax,8(%ebp)  # compare x with eax
setg %al           # al = x > y
andl $0xFF,%eax   # zero rest of %eax
```

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

# Another Way

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

movzbl: 0 extend byte (%al) to a 32-bit value and place in eax

- **what's the difference? andl vs. movzbl**
  - both are simple instructions computationally
  - the & \$255 requires a byte long constant however
  - movzbl saves one byte of code size
  - compiler's choice...

# Control Flow in C

- **conditionals: if, if else, switch**
- **loops: for, while, do while**
- **disciplined gotos: break, continue**
  - **break: exit innermost loop or switch**
  - **continue: goto next iteration**
    - » **while & do: execute test now**
    - » **for: control passes to the increment step**
- **undisciplined gotos: goto**

# Control Flow in x86

- **jmp – unconditional jump**
  - **jmp .L1**                      **direct jump to a label (seen in .o)**
    - » we'll see what the linked version looks like shortly
  - **jmp \*operand**                **indirect jump via a pointer**
    - » operand is encoded the same as for a move
    - » \* marks the jump as indirect
- **jx – conditional jump where x is some condition specifier**
- **There are plenty of other control flow constructs but we're not talking about them today**

# Conditional Jumps

- **jX Instructions**

- Jump to different part of code depending on condition codes

<b>jX</b>	<b>Condition</b>	<b>Description</b>
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	$\sim ZF$	Not Equal / Not Zero
js	SF	Negative
jns	$\sim SF$	Nonnegative
jg	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
jge	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
jl	$(SF \wedge OF)$	Less (Signed)
jle	$(SF \wedge OF) \   \ ZF$	Less or Equal (Signed)
ja	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
jb	CF	Below (unsigned)

# More Conditional Jumps

<b>jX</b>	<b>Condition</b>	<b>Description</b>
<b>jae</b>	<b><math>\sim CF</math></b>	<b>Jump Above or Equal(unsigned)</b>
<b>jbe</b>	<b><math>CF \ \&amp; \ \sim ZF</math></b>	<b>Jump Below or Equal (unsigned)</b>

# If-Else Example

Why is it `_max` instead of `max`?

```
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

`_max:`

```
    pushl %ebp
    movl  %esp,%ebp
```

} prologue

```
    movl  8(%ebp),%edx
    movl  12(%ebp),%eax
    cmpl  %eax,%edx
    jle  L9
    movl  %edx,%eax
```

} body

`L9:`

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

} epilogue

Note: on entry return address is on the top of the stack

`%eax` holds return value when the return value is an int or pointer

What does `L9` mean?

# Same Example w/ goto

```
int goto_max(int x, int y)
{
    int rval = y;
    int ok = (x <= y);
    if (ok)
        goto done;
    rval = x;
done:
    return rval;
}
```

- C allows “goto” as means of transferring control
  - » Closer to machine-level programming style
- Generally considered bad coding style
  - » Sometimes gotos permit cleaner code (but rarely)

```
    movl 8(%ebp),%edx    # edx = x
    movl 12(%ebp),%eax   # eax = y
    cmpl %eax,%edx      # x : y (computes x - y)
    jle L9              # if <= goto L9
    movl %edx,%eax      # eax = x } Skipped when x ≤ y
L9:                    # done:
```

What does it mean when the compiler generates the same code?

# Do-While Loops

## “Good” C Code

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

## Using goto

```
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 → Assembly

## Goto Version

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

## • Registers

`%edx`    `x`  
`%eax`    `result`

## Assembly

```
_fact_goto:
    pushl %ebp                # prologue
    movl %esp,%ebp           # prologue
    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           # epilogue
    popl %ebp                 # epilogue
    ret                       # epilogue
```

# General Do-While's

## 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
  - Just like any other boolean value in C

# Examining While Loops

## 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

# Extra Branch vs. Extra Test

## C Code

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

- 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;
}
```

# While → Do-While

## C Code

```
while (Test)  
  Body
```

2 control statements in  
inner loop (executed most)



## Do-While Version

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

1 inner  
loop  
control  
statement



## Goto Version

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

# “For” Loop Example (powers the right way)

```
/* 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 property that**  $p = p_0 + 2p_1 + 4p_2 + \dots + 2^{n-1}p_{n-1}$
- **Gives:**  $x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \dots \cdot (\dots((z_{n-1}^2)^2)\dots)^2$ 
  - $z_i = 1$  when  $p_i = 0$
  - $z_i = x$  when  $p_i = 1$
- **Complexity**  $O(\log p)$

  
 $n-1$  times

## Example

$$\begin{aligned} 3^{10} &= 3^2 * 3^8 \\ &= 3^2 * ((3^2)^2)^2 \end{aligned}$$

# 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;
}
```

result	x	p
1	3	10
1	9	5
9	81	2
9	6561	1
531441	43046721	0

# “For” Loop Example

## General Form

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

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

*Init*

```
result = 1
```

*Test*

```
p != 0
```

*Update*

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

*Body*

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

# “For” → “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:
```

# Compiler's view of a "for" loop

## Goto Version

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



```
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:
```

***Init***

```
result = 1
```

***Test***

```
p != 0
```

***Update***

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

***Body***

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

# Switch Implementations

```
typedef enum
{ADD, MULT, MINUS, DIV, BAD}
  op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD:
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case BAD:
    return '?';
  }
}
```

- **Series of conditionals**
  - » Good if few cases
  - » Slow if many
- **Jump Table**
  - » Lookup branch target
  - » Avoids conditionals
    - Possible when cases are small integer constants
    - and rather densely packed
- **GCC picks one based on case structure**
- **Dangers in example code**
  - » No default given
- **In C cases fall through (argh!)**
  - » need break or goto
  - » or return (in this case)

# Using Jump Tables

## Switch code:

```
switch(op) {  
  case 0:  
    Block 0  
  case 1:  
    Block 1  
    . . .  
  case n-1:  
    Block n-1  
}
```

## Translated code:

```
target = JTab[op];  
goto *target;
```

## 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
```

# Jump Table Example

## Switch code:

```
typedef enum
  {ADD, MULT, MINUS, DIV, BAD}
  op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
    . . .
  }
}
```

## Enumerated Values

ADD	0
MULT	1
MINUS	2
DIV	3
MOD	4
BAD	5

## Setup:

```
unparse_symbol:
  pushl %ebp                # prologue
  movl %esp,%ebp           # prologue
  movl 8(%ebp),%eax         # eax = op
  cmpl $5,%eax             # Compare op : 5
  ja .L49                  # If > goto done
  jmp *.L57(,%eax,4)        # goto Table[op]
```

# Jump Table Continued

## Table Contents

```
.section .rodata
    .align 4
.L57:
    .long .L51    # Op = 0
    .long .L52    # Op = 1
    .long .L53    # Op = 2
    .long .L54    # Op = 3
    .long .L55    # Op = 4
    .long .L56    # Op = 5
```

## Enumerated Values

ADD	0
MULT	1
MINUS	2
DIV	3
MOD	4
BAD	5

## Targets & Completion

```
.L51:
    movl $43,%eax    # '+'
    jmp .L49         # .L49==done
.L52:
    movl $42,%eax    # '*'
    jmp .L49
.L53:
    movl $45,%eax    # '-'
    jmp .L49
.L54:
    movl $47,%eax    # '/'
    jmp .L49
.L55:
    movl $37,%eax    # '%'
    jmp .L49
.L56:
    movl $63,%eax    # '?'
    # Fall Through to .L49
```

# Jump Table Continued Again

```
.L49:                                # Done:
    movl %ebp,%esp                    # epilogue
    popl %ebp                          # epilogue
    ret                                # epilogue
```

- **Advantage of Jump Table**
  - Can do  $k$ -way branch in constant time
  - Repeated conditionals require  $O(N)$  time

# Jump Table Explanation

- **Symbolic Labels**

- Labels of form `.LXX` translated into addresses by assembler

- **Table Structure**

- Each target requires 4 bytes
- Base address at `.L57`

- **Jumping**

```
jmp .L49
```

- Jump target is denoted by label `.L49`

```
jmp *.L57(, %eax, 4)
```

- Start of jump table denoted by label `.L57`
- Register `%eax` holds `op`
- Must scale by factor of 4 to get offset into table
- Fetch target from effective Address `.L57 + op*4`

- **Puzzle**

- What value is returned when `op` is invalid?

# Even More Detail

- Setup

- Label `.L49` becomes address `0x804875c` (done label)
- Label `.L57` becomes address `0x8048bc0` (start of jump table)

```
08048718 <unparse_symbol>:  
8048718: 55                pushl   %ebp  
8048719: 89 e5            movl   %esp,%ebp  
804871b: 8b 45 08        movl   0x8(%ebp),%eax  
804871e: 83 f8 05        cmpl   $0x5,%eax  
8048721: 77 39           ja     804875c  
8048723: ff 24 85 c0 8b  jmp   *0x8048bc0(,%eax,4)
```

# Inspecting Jump Tables

- List of target addresses doesn't show up in disassembled code

- Can inspect using GDB

`gdb code-examples`

```
(gdb) x/6xw 0x8048bc0
```

- Examine 6 hexadecimal format "words" (4-bytes each)
- Use command "help x" to get format documentation

```
0x8048bc0 <_fini+32>:
```

```
0x08048730
```

```
0x08048737
```

```
0x08048740
```

```
0x08048747
```

```
0x08048750
```

```
0x08048757
```

# Another Way

- **Jump Table Stored in Read Only Data Segment (.rodata)**
  - Various fixed values needed by your code
- **Can examine with objdump**  
`objdump code-examples -s --section=.rodata`
  - Show everything in indicated segment.
- **Hard to read**
  - Jump table entries shown with reversed byte ordering

```
Contents of section .rodata:
```

```
8048bc0 30870408 37870408 40870408 47870408 0...7...@...G...
8048bd0 50870408 57870408 46616374 28256429 P...W...Fact(%d)
8048be0 203d2025 6c640a00 43686172 203d2025 = %ld..Char = %
...
```

- E.g., 30870408 really means 0x08048730

# Compiler Pads for Alignment

- No-operations (`movl %esi,%esi`) inserted to align target addresses

```
8048730: b8 2b 00 00 00  movl  $0x2b,%eax
8048735: eb 25             jmp   804875c <unparse_symbol+0x44>
8048737: b8 2a 00 00 00  movl  $0x2a,%eax
804873c: eb 1e             jmp   804875c <unparse_symbol+0x44>
804873e: 89 f6             movl  %esi,%esi
8048740: b8 2d 00 00 00  movl  $0x2d,%eax
8048745: eb 15             jmp   804875c <unparse_symbol+0x44>
8048747: b8 2f 00 00 00  movl  $0x2f,%eax
804874c: eb 0e             jmp   804875c <unparse_symbol+0x44>
804874e: 89 f6             movl  %esi,%esi #NOP alignment pad
8048750: b8 25 00 00 00  movl  $0x25,%eax
8048755: eb 05             jmp   804875c <unparse_symbol+0x44>
8048757: b8 3f 00 00 00  movl  $0x3f,%eax
```

# Resulting Binary Image

## Entry

0x08048730

0x08048737

0x08048740

0x08048747

0x08048750

0x08048757

8048730: b8 2b 00 00 00	movl
8048735: eb 25	jmp
8048737: b8 2a 00 00 00	movl
804873c: eb 1e	jmp
804873e: 89 f6	movl
8048740: b8 2d 00 00 00	movl
8048745: eb 15	jmp
8048747: b8 2f 00 00 00	movl
804874c: eb 0e	jmp
804874e: 89 f6	movl
8048750: b8 25 00 00 00	movl
8048755: eb 05	jmp
8048757: b8 3f 00 00 00	movl

# 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 efficient to use jump table
  - » Would require 1000 entries
- Obvious translation into if-then-else would use up to nine tests

# Sparse Switch Code

```
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

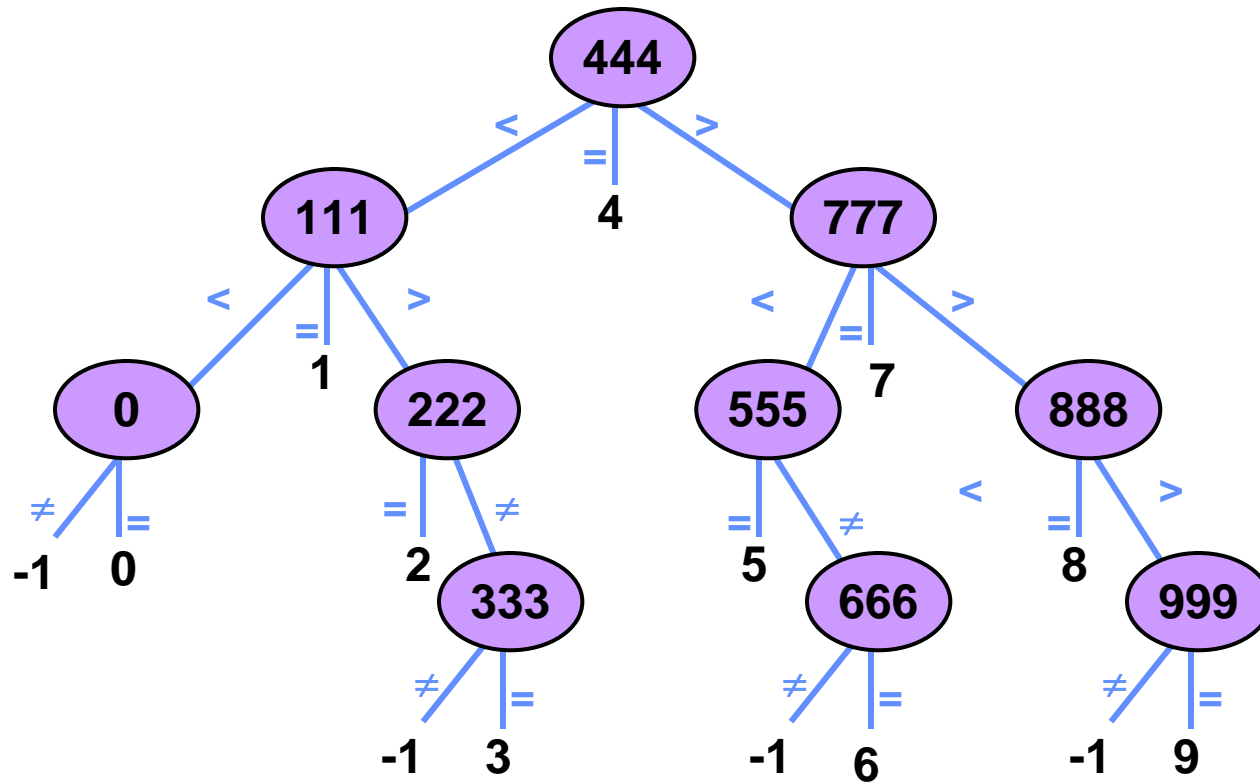
. . .
```

**Binary search: pick a middle value  
each time**

- 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
  - Avoids wasting too much space or time

# Summary

## •C Control

- if-then-else
- do-while
- while
- switch

## •Assembler Control

- jump
- Conditional jump

## •Compiler

- Must generate assembly code to implement more complex control

## •Standard Compiler Techniques

- All loops converted to do-while form
- Large switch statements use jump tables

## •Condition codes in CISC

- CISC machines generally have condition code registers

## •Condition codes in RISC

- Use general registers to store condition information
- Special comparison instructions
- E.g., on Alpha:

```
cmple $16,1,$1
```

- » Sets register \$1 to 1 when Register \$16  $\leq$  1