Machine-Level Programming: Loops, Switch Statements

Topics

- Control Flow
  - Do-While, While, and For Loops
  - Switch Statements

Announcements

- Data Lab grading this week
  - Sign up for grading time slots with your TA
- Bomb Lab available on moodle, due Friday Oct 3
  - Start early
  - Extra credit secret bomb: add 7 points to your final lab grade
- First midterm probably Tues Oct 7
- Recitation Exercises #2 will be released soon
  - Due Monday Sept 29 handed in at your recitation
- Essential that you read the textbook in detail & do the practice problems
  - Read Chapter 3.1-3.14, skip 3.12 for now

Recap…

- **Condition Codes**
  - **CF, OF, ZF, SF**
  - **Set due to various operations, e.g.** `addl, subl, cmpl, testl`
  - `sete, setns, setle`
- **Conditional Jumps based on these codes**
  - `jge` **means "jump if greater than or equal to"**
  - `jl` **means "jump if less than"**
  - **Usage:**

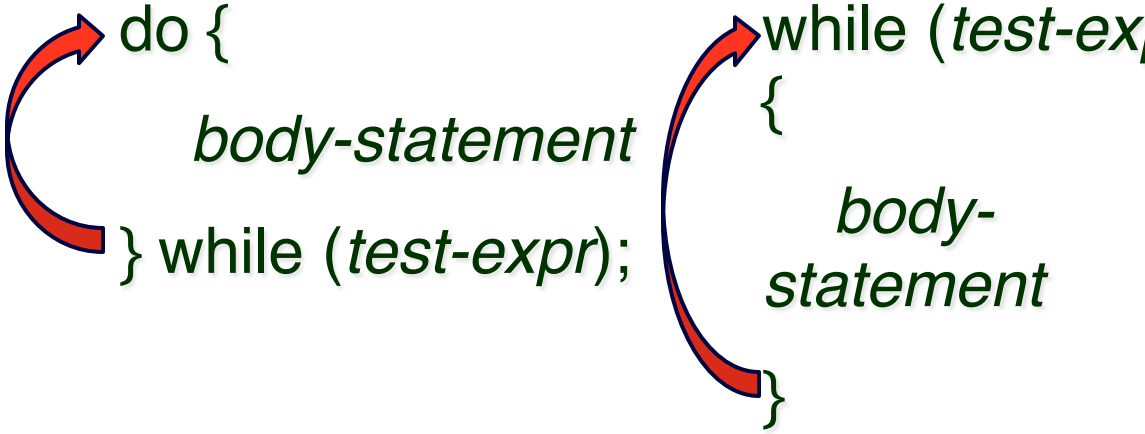
```
cmpl y, x      # sets flags
jl .L50        # if x<y, then jump to address of label .L50
```

  - **Use these conditional jumps to implement if-then-else conditional branching**

Recap…

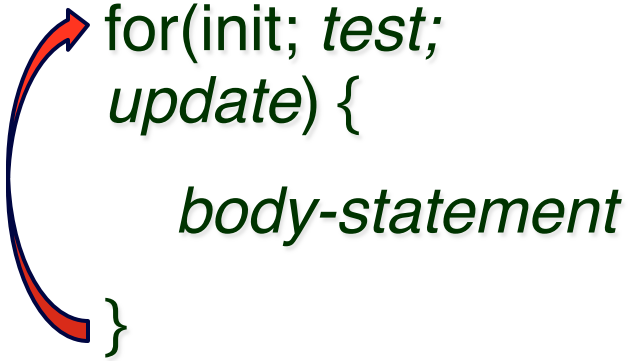
- **Conditional moves based on these codes**
  - `cmovl src, dst` **<--- sometimes better than conditional jumps**
  - **Execute both branches of if-then-else, and conditionally move at the very end based on the test condition – easier to pipeline**
    - **Sometimes this is risky**

Loops in C



- **Executes body-statement**
- **Then tests expression**
  - **If true, loops back to 'do'**
  - **Else exit**

- **Tests expression first**
  - **If true, executes body & loops back to 'while'**
  - **Else exit**



- **Initializes first**
- **If test is true**
  - **Execute body statement**
  - **Execute update & loop back to 'for'**
- **Else exit**

"Do-While" Loop Example

C Code for computing x!

```
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:*** {
  Statement$_1$;
  Statement$_2$;
  …
  Statement$_n$;
  }

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

"While" Loop Example

C Code

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

  return result;
}
```

Goto Version #1

```
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

Implementing "While" as"Do" Loop

Goto Version #1

```
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;
}
```

Goto Version #2

```
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;
}
```

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

General "While" Translation

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:
```

Jump-to-Middle "While" Loop Translation

Goto Version #2

```
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;
}
```

Goto Version #3

```
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 Loop 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

Goto (Previous) Version

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

Jump-to-Middle Assembly Example

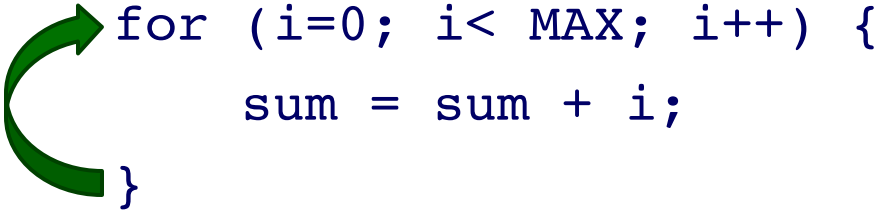
Goto Version #3

```
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;
}
```

```
# 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 Loops

- **Typical example:**



```
for (i=0; i< MAX; i++) {
     sum = sum + i;
}
```

We see that `i` is initialized to 0.

Then we check if `i` < MAX. If not, then we execute the body of the `for` loop sum = sum + `i`.

At this point, at the end of the loop, we increment `i`, and then jump back up to the top of the loop and check again if `i` < MAX. If so, we loop again, check again, loop again, check again, …

When `i`>=MAX, we exit out of the loop.

For Loops

- **General usage:**

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

This is equivalent to:

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

"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:
```

"For" Loop Example

Compute x^p, where x is a positive integer and p is a non-negative integer

- Could build a for loop:

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, int p) {
  int result=1, i;
  for (i=0; i<p; i++) {
    result *= x;
  }
  return result;
}
```

Problems

- For large p, this loop executes p times
  - Partial product is multiplied by x each time
- Can we do better?
  - Partial product is multiplied by *itself* each time (squared) until just < x^p, then deal with remaining powers
  - Executes for loop many fewer times for large p, i.e. O(log(p)) !

Faster "For" Loop Example

Example

$3^{18} = 3^2 * 3^{16}$

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

Multiply partial product times itself each time through the loop, which is equivalent to squaring

In this example, only have to square 4 times, but also have to cleanup remaining powers

Faster "For" Loop Example

```
/* Compute x raised to nonnegative power p */
int ipwr_for_fast(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** $p = p_0 + 2p_1 + 4p_2 + \ldots 2^{n-1}p_{n-1}$
- **Gives:** $x^p = z_0 \cdot {z_1}^2 \cdot ({z_2}^2)^2 \cdot \ldots \cdot \underbrace{(\ldots(({z_{n-1}}^2)^2)\ldots)^2}_{n-1 \text{ times}}$
  - $z_i = 1$ when $p_i = 0$
  - $z_i = x$ when $p_i = 1$
- **Complexity** $O(\log p)$

"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
```

Init

```
result = 1
```

Test

```
p != 0
```

Update

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

Body

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

"For" Loop Compilation

Goto Version

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

Proto-Assembly Version

```
  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;
}
```

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

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

switch() is missing default case

Switch Statements

Implementation Options

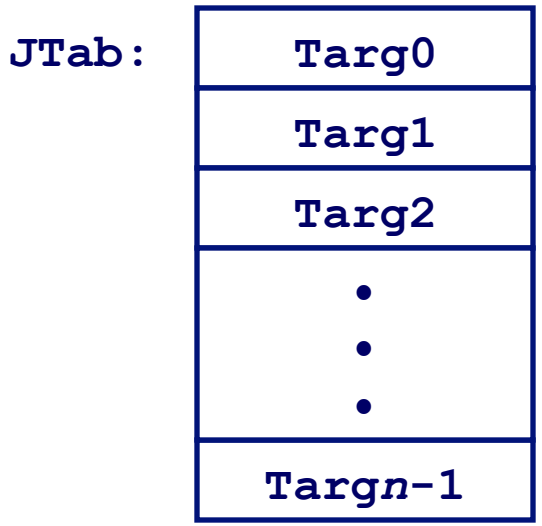
- Series of conditionals, e.g. if- else if - else if - else if…
  - Good if few cases
  - Slow if many cases, e.g. many compares and conditional jumps
- Jump Table (array of addresses)
  - Index into array and jump to branch target
  - Avoids conditionals
  - Good when cases are small integer constants
- GCC picks one based on case structure

Jump Table Structure

Switch Form

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

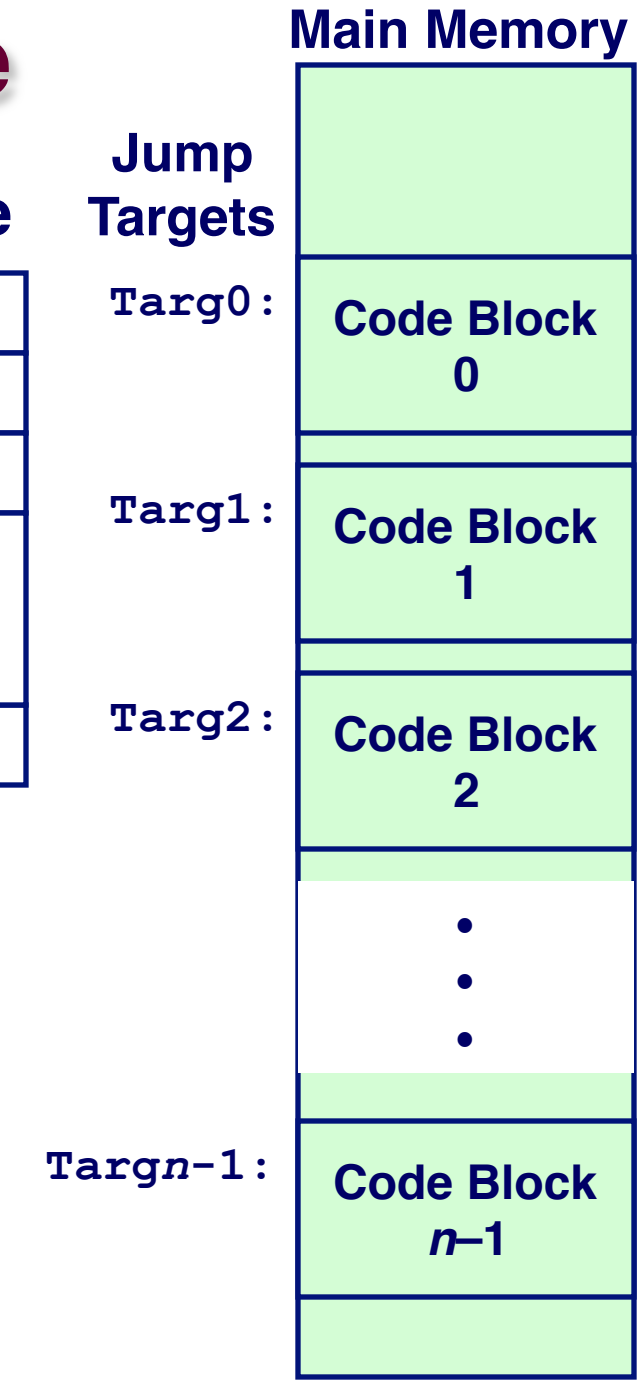
Jump Table



Approx. Translation

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

Note: no conditional evaluation needed, so very fast to jump to code block, but at cost of a potentially large jump table



Switch Statement Example

Branching Possibilities

```
typedef enum
  {ADD, MULT, MINUS, DIV, MOD, 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           # Setup
   movl %esp,%ebp       # Setup
   movl 8(%ebp),%eax    # eax = op
   cmpl $5,%eax         # Compare op : 5
   ja .L49              # If > goto done
   jmp *.L57(,%eax,4)   # goto Table[op]
```

Assembly Setup Explanation

```
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, as each address in jump table is 4 bytes wide (on a 32-bit system)
- Fetch target from effective Address **`.L57 + op*4`**

Table Structure

- **Array of target memory addresses, so each target requires 4 bytes**
- **Base address at `.L57`**

Symbolic Labels

```
jmp .L49
```

- Jump target is denoted by label **`.L49`**
- **Labels of form `.LXX` translated into addresses (absolute or PC-relative) by assembler**

Jump Table

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
.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
```

Switch Statement Completion

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

Puzzle

- What value returned when `op` is invalid?

Answer

- Register `%eax` set to `op` at beginning of procedure
- This becomes the returned value

Advantage of Jump Table

- Can do *k*-way branch in *O*(1) operations

Switch Object Code

Setup

- Label `.L49` becomes address `0x804875c`
- Label `.L57` becomes address `0x8048bc0`

```
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 <unparse_symbol+0x44>
8048723:ff 24 85 c0 8b      jmp     *0x8048bc0(,%eax,4)
```

Switch Object Code (cont.)

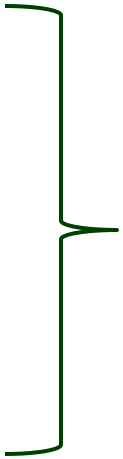
Jump Table

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

```
gdb code-examples
(gdb) x/6xw 0x8048bc0
```

  - E*x*amine 6 he*x*adecimal format "*w*ords" (4-bytes each)
  - Use command "`help x`" to get format documentation

```
0x8048bc0 <_fini+32>:
  0x08048730
  0x08048737
  0x08048740
  0x08048747
  0x08048750
  0x08048757
```



These are memory address pointers/jump targets to the code blocks for the different cases of the switch() statement

Extracting Jump Table from Binary

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 = %
 ...
```

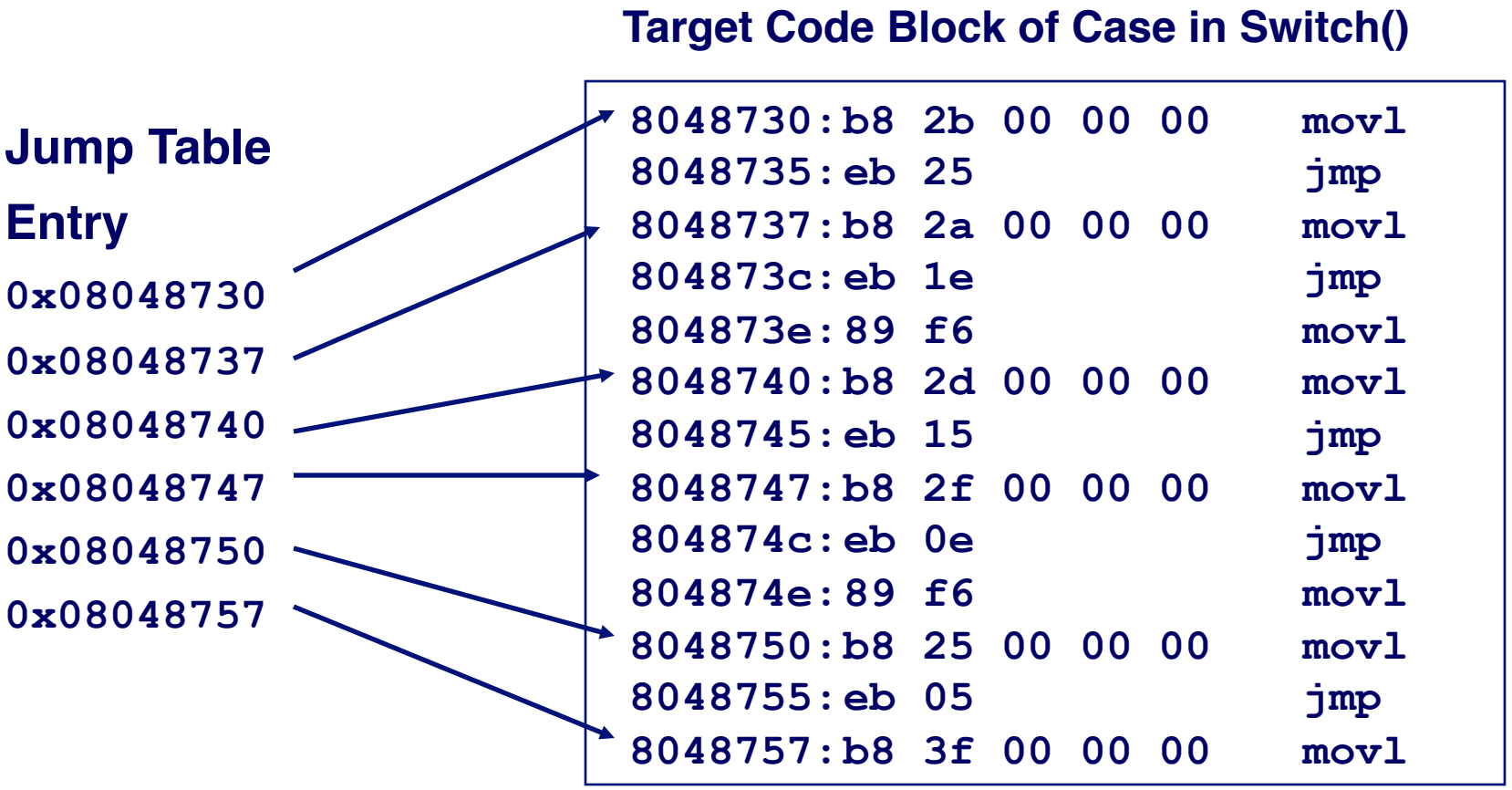
- E.g., `30870408` really means `0x08048730`

Disassembled Targets

```
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
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       #'?'
```

- `movl %esi,%esi` does nothing
- Inserted to align instructions for better cache performance

Matching Disassembled Targets

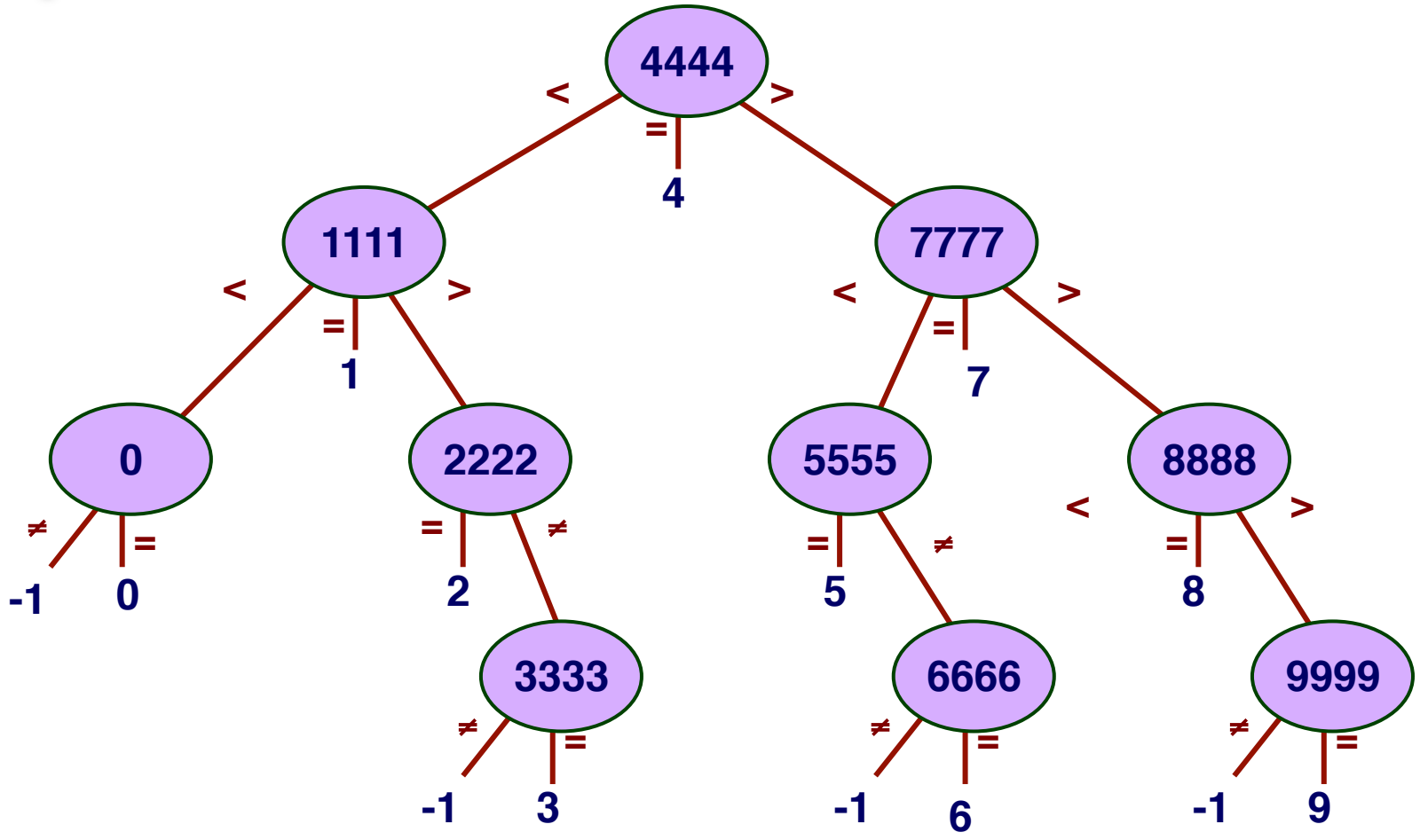


Sparse Switch Example

```
/* Return x/1111 if x is multiple
   && <= 9999.  -1 otherwise */
int div1111(int x)
{
  switch(x) {
  case    0: return 0;
  case 1111: return 1;
  case 2222: return 2;
  case 3333: return 3;
  case 4444: return 4;
  case 5555: return 5;
  case 6666: return 6;
  case 7777: return 7;
  case 8888: return 8;
  case 9999: return 9;
  default: return -1;
  }
}
```

- Not practical to use jump table
  - Would require 10000 entries
- Obvious translation into if-then-else would have maximum of 9 tests in the worst case
  - Can we do better?

Sparse Switch Code Structure



- Organizes the cases as a binary tree
- Logarithmic performance, i.e. order O(logN) in complexity
  - Worst case of 4 compares vs. ~9 for linear search of cases

Sparse Switch Code

```
    movl 8(%ebp),%eax  # get x
    cmpl $4444,%eax    # x:4444
    je L8
    jg L16
    cmpl $1111,%eax    # x:1111
    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
    . . .
```

Supplementary Slides

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

ipwr Computation

```
/* Compute x raised to nonnegative power p */
int ipwr_for_fast(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
59049	3	10
59049	9	5
6561	81	2
6561	6561	1
1	43046721	0

Summarizing

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 Techniques

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

Conditions in CISC

- CISC machines generally have condition code registers

Conditions 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 <= 1