UNM SCHOOL of ENGINEERING
Department of Computer Science

# Basics of Conditional Control Flow Loops and switch

## Lecture #10

Prof. Soraya Abad-Mota, PhD

(Chap. 3, Section 3.6.7 and 3.6.8 )

# Comments from last lecture

- About exercise 3.16
  - First check !test ➔ Negate the original condition:
  - `~(p && a > *p) = ~p || a <= *p`
  - The assembly code has two branches because it is a compound condition and it cannot do it in one
  - ~p is checked first and if p is NULL, it goes to done

# Remember

- The way to study the material in this chapter is to do the exercises, the core is in the details, not the general idea. Ideas → simple, learning happens in the details.
- Review the learning objectives on each set of slides (have you reached those objectives? requires practice)

# Previously

- Control: Condition codes (CCs) (Secs. 3.6.1 to 3.6.4)
- How did we use them?
- How does an if statement translate into assembly?

Today:

- Conditional branches (use the conditional codes to translate loops and the switch statement by setting and reading the CCs)
  - Loops (do-while, while, for)
  - Switch Statements

# Learning objectives

After this session and appropriate practice of indicated problems students should be able to:

1. Describe and practice the translation of loops (do-while, while, and for) into assembly code. (Sec. 3.6.7)
2. Describe and practice the translation of the switch statement (Sec. 3.6.8)

# "Do-While" Loop Example

C Code

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

Goto Version (mimics assembly)

```
long pcount_goto
  (unsigned long x) {
  long 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

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

| Register | Use(s) |
|---|---|
| %rdi | Argument **x** |
| %rax | **result** |

```
   movl    $0, %eax     #  result = 0
.L2:                    #  loop:
   movq    %rdi, %rdx
   andl    $1, %edx     #  t = x & 0x1
   addq    %rdx, %rax   #  result += t
   shrq    %rdi         #  x >>= 1
   jne     .L2          #  if (x) goto loop
   rep; ret             # p. 208 explains
```

# General "Do-While" Translation

C Code

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

Goto Version (closer to assembly)

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

- Body: { $\mathbf{Statement_1}$;
  $\mathbf{Statement_2}$;
  …
  $\mathbf{Statement_n}$;
  }

Note: find another example in the book, p. 221

# Translate Conditional branches

- **Loops** (Sec. 3.6.7)
  - do-while loop translation
  - **While loop** (p. 223) (two possible translations)
- Switch Statements (Sec. 3.6.8)

# General "While" Translation #1

- "Jump-to-middle" translation
- Used with **-Og**

While version

```
while (Test)
  Body
```

Goto Version

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

# While Loop Example #1

C Code

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

Jump to Middle

```
long pcount_goto_jtm
  (unsigned long x) {
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) goto loop;
  return result;
}
```

- Compare to do-while version of function (slides 46-47, same example)
- Initial goto starts loop at test

# General "While" Translation #2

While version

```
while (Test)
  Body
```

Do-While Version

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

- *"Guarded do" translation*
- "Do-while" conversion
- Used with **–O1**

Goto Version

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

# While Loop Example #2

C Code

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

Do-While Version

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

- Compare to do-while version of function
- Initial conditional *guards* entrance to loop

# Problem 3.24 (given assembly code)

```
loop_while:
   movl $1,%eax
   jmp  .L2
.L3:
   leaq  (%rdi,%rsi), %rdx
   imulq  %rdx,%rax
   addq   $1, %rdi
.L2:
   cmpq   %rsi, %rdi
   jl     .L3
   rep;  ret
```

# Complete C code (3.24 pp. 224-225)

```
long loop_while(long a, long b) {
    long result =
    while (                ) {
        result =
        a =
    }
    return result;
}
```

# Practice on your own with

- Problem 3.25, p. 226 (during lecture today?)
- Problem 3.26, p. 228

# Problem 3.25 (given assembly code)

```
loop_while2:
   testq %rsi, %rsi
   jle   .L8
   movq  %rsi, %rax
.L7:
   imulq  %rdi, %rax
   subq   %rdi, %rsi
   testq  %rsi, %rsi
   jg     .L7
   rep;  ret
.L8:
   movq   %rsi, %rax
   ret
```

# Complete C code (3.25 pp. 226-227)

```
long loop_while2(long a, long b){
    long result =
    while (                ) {
        result =
        b =
    }
    return result;
}
```

# "For" Loop Form

## General Form

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

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
      (x >> i) & 0x1;
  result += bit;
}
```

# "For" Loop → While Loop

For Version

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

While Version

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

# For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
      (x >> i) & 0x1;
  result += bit;
}
```

```
long pcount_for_while
  (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

# "For" Loop Do-While Conversion

C Code

```
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

- Initial test can be optimized away

Goto Version

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;                    Init
  if (!(i < WSIZE))
    goto done;              ! Test
 loop:
  {
    unsigned bit =
      (x >> i) & 0x1;       Body
    result += bit;
  }
  i++;  Update
  if (i < WSIZE)            Test
    goto loop;
 done:
  return result;
}
```

UNM SCHOOL of ENGINEERING
Department of Computer Science

# Basics of Conditional Control Flow

## Lecture #10 – part 2

Prof. Soraya Abad-Mota, PhD

# Translate Conditional branches

- Loops (Sec. 3.6.7)
  - do-while loop translation
  - While loop (p. 223) (two possible translations) which are the two patterns for translating while loops?
  - For loops, how are they translated?
- Switch Statements (Sec. 3.6.8)
  - Frequently uses a `jump table`

# Switch Statement Example

(pp. 233-236)

```
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

- Commonly translated into a *jump table* instead of a sequence of if/else statements

# Jump Table Structure

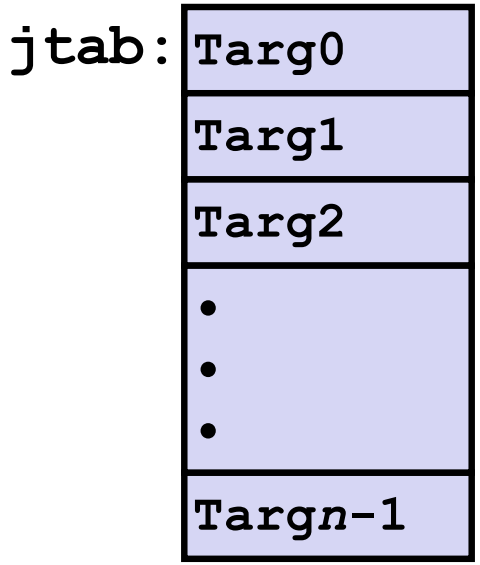
Switch General Form

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

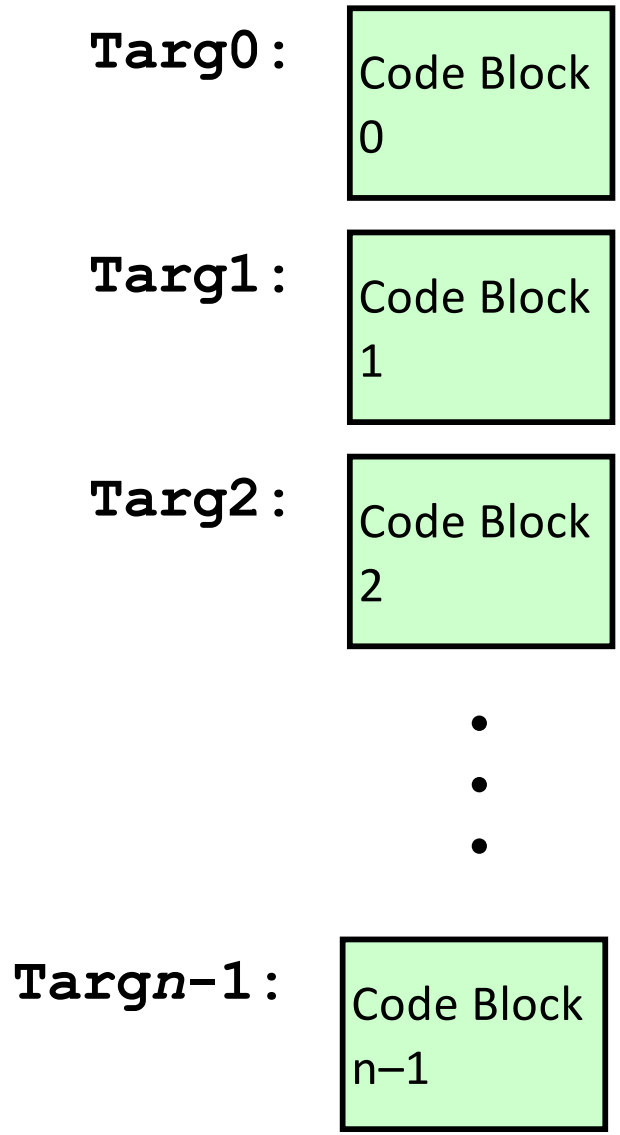
Translation (Extended C)

```
goto *JTab[x];
```

Jump Table




Jump Targets



# Switch Statement Example

```
long switch_eg(long x, long y,
               long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

| Register | Use(s) |
|---|---|
| `%rdi` | Argument **x** |
| `%rsi` | Argument **y** |
| `%rdx` | Argument **z** |
| `%rax` | Return value |

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja      .L8
    jmp     *.L4(,%rdi,8)
```

Note that **w** not initialized here

What range of values takes default?

# Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi          # x:6
    ja      .L8               # Use default
    jmp     *.L4(,%rdi,8)     # goto *JTab[x]
```

Indirect jump

# Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes
  - Base address at **`.L4`**

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

- Jumping
  - Direct: **`jmp .L8`**
  - Jump target is denoted by label **`.L8`**
  - Indirect: **`jmp *.L4(,%rdi,8)`**
  - Start of jump table: **`.L4`**
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective Address **`.L4 + x*8`**
    - Only for $0 \le \mathbf{x} \le 6$

# Jump Table

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

```
switch(x) {
case 1:         // .L3
    w = y*z;
    break;
case 2:         // .L5
    w = y/z;
    /* Fall Through */
case 3:         // .L9
    w += z;
    break;
case 5:
case 6:         // .L7
    w -= z;
    break;
default:        // .L8
    w = 2;
}
```

# What follows... (plan for studying this portion)

- Slides 74 to 77 (4 slides) show in detail how the cases of the switch are handled in the assembly code. Please follow them and ask questions.
- For the rest of this part of the lecture:
  - Conclusions about the use of jump tables (slide 72)
  - Summary which ends sec. 3.6 (slide 73)
- Later, on your own:
  - Follow the explanation in the textbook pp. 233-236.
  - Then try on your own, problems 3.30 (p 236) and 3.31 (p 237).

# Should we always use jump table?

- Compiler can ***choose*** whether to use jump table or a sequence of if/else statements (translated to assembly)
  - Or some combination of the two!
- What are the costs and benefits of a jump table versus a sequence of conditionals
  - Space for the jump table
  - Indirect memory access into the table
  - Jump table entry is 4(8) bytes vs. cmp/jmp statement
- What if there are *lots* of default entries?

# Summarizing

- C Control
  - if-then-else
  - do-while
  - while, for
  - switch
- Assembler Control
  - Conditional jump
  - Conditional move
  - Indirect jump (via jump tables)
  - Compiler generates code sequence to implement more complex control
- Standard Techniques
  - Loops (while and for) converted to do-while or jump-to-middle form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees (if-elseif-elseif-else)

# Code Blocks (x == 1)

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

```
.L3:
   movq     %rsi, %rax   # y
   imulq    %rdx, %rax   # y*z
   ret
```

| Register | Use(s) |
|---|---|
| `%rdi` | Argument **x** |
| `%rsi` | Argument **y** |
| `%rdx` | Argument **z** |
| `%rax` | Return value |

# Handling Fall-Through

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

```
case 2:
    w = y/z;
    goto merge;
```

```
case 3:
    w = 1;
merge:
    w += z;
```

# Code Blocks (x == 2, x == 3)

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

```
.L5:                        # Case 2
   movq    %rsi, %rax
   cqto
   idivq   %rcx             #  y/z
   jmp     .L6              #  goto merge
.L9:                        # Case 3
   movl    $1, %eax         #  w = 1
.L6:                        # merge:
   addq    %rcx, %rax       #  w += z
   ret
```

| Register | Use(s) |
|---|---|
| **%rdi** | Argument **x** |
| **%rsi** | Argument **y** |
| **%rdx** | Argument **z** |
| **%rax** | Return value |

# Code Blocks (x == 5, x == 6, default)

```
  switch(x) {
    . . .
    case 5:  // .L7
    case 6:  // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

```
.L7:                    # Case 5,6
  movl   $1, %eax       #  w = 1
  subq   %rdx, %rax     #  w -= z
  ret
.L8:                    # Default:
  movl   $2, %eax       #  2
  ret
```

| Register | Use(s) |
|---|---|
| `%rdi` | Argument **x** |
| `%rsi` | Argument **y** |
| `%rdx` | Argument **z** |
| `%rax` | Return value |

Ended section 3.6

go to Procedures (section 3.7)