UNM SCHOOL of ENGINEERING
Department of Computer Science

CS341L: Basics of Conditional Control Flow (section 3.6)

Prof. Soraya Abad-Mota, PhD

UNM SCHOOL of ENGINEERING
Department of Computer Science

Condition codes, Jumps, if-else translation

Lecture #9

Prof. Soraya Abad-Mota, PhD

(Chap. 3, Section 3.6 on)

Machine Programming I: Summary

(Covered so far: chapter 3, secs. 3.1 to 3.5)

- History of Intel processors and architectures
- C, assembly, machine code
  - New forms of visible state: program counter, registers, ...
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- Assembly Basics: Registers, operands, move
  - The x86-64 move instructions cover wide range of data movement forms
- Arithmetic
  - C compiler will figure out different instruction combinations to carry out computation

Learning objectives (start 3.6)

After this portion students should be able to:

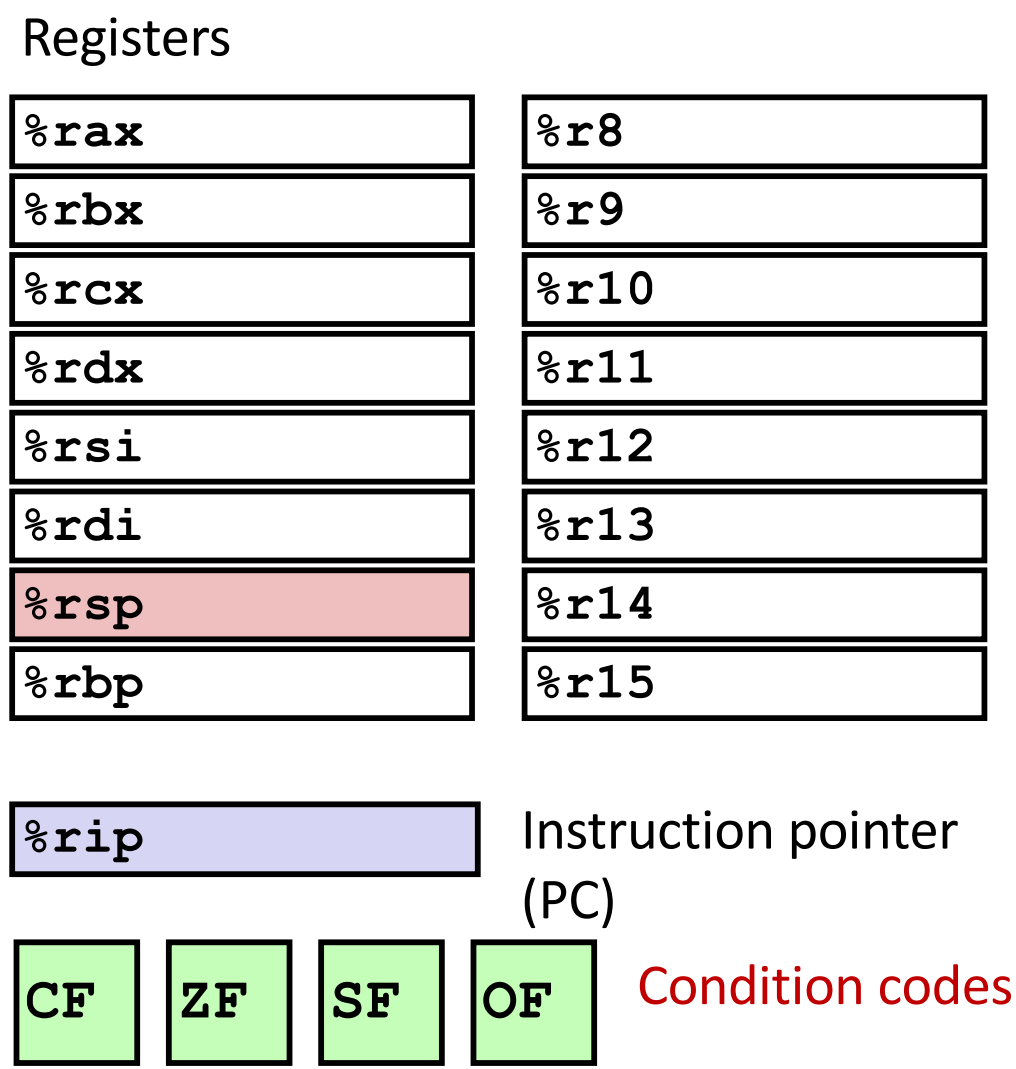
1. Describe condition codes. (Subsec. 3.6.1)
2. Summarize how to set and use condition codes.
3. Describe how are conditional branches implemented in assembly code. (If-then-else statements)
4. Describe and practice the translation of loops and the switch statement into assembly code. (Possibly week 6)

Look ahead

- Control: Condition codes (Secs. 3.6.1 to 3.6.4)
- Conditional branches
- Loops
- Switch Statements

Processor State (x86-64, Partial)

- Information about currently executing program
  - Temporary data (**%rax**, …)
  - Location of runtime stack (**%rsp**)
  - Location of current code control point (**%rip**, …)
  - Status of recent tests (CF, ZF, SF, OF)



Condition Codes (Implicit Setting)

- Single bit registers
  - CF Carry Flag (unsigned) SF Sign Flag (signed)
  - ZF Zero Flag OF Overflow Flag (signed)

- Implicitly set (think of it as side effect) by arithmetic operations

  Example: **`addq`** Src,Dest ↔ **`t = a+b`**

  CF set if carry out from most significant bit (unsigned overflow)

  ZF set if **`t == 0`**

  SF set if **`t < 0`** (as signed)

  OF set if two's-complement (signed) overflow

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

- Not set by **`leaq`** instruction (why?)

Explicitly setting the condition codes

Two instructions:

- cmp_
- test_
  - the _ is substituted by b, w, l, or q
  - the full set of 8 instructions is in Fig. 3.13 p. 202

Compare: Explicit Setting of Condition Codes

- Explicit Setting by Compare Instruction
  - **`cmpq`** Src2, Src1
  - **`cmpq 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`** (as signed)
  - OF set if two's-complement (signed) overflow
  `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

Test: Explicit Setting of Condition Codes

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

Reading Condition Codes

- SetX Instructions
  - Set low-order byte of destination to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 7 bytes

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)

Learning objectives

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

1. Describe condition codes.
2. Summarize how to set and use condition codes.
3. Describe how are conditional branches implemented in assembly code. (If-then-else statements)
4. Describe and practice the translation of loops and the switch statement into assembly code. (Possibly week 6)

Reading Condition Codes (Cont.)

- SetX Instructions:
  - Set single byte based on combination of condition codes
- One of addressable byte registers
  - Does not alter remaining bytes
  - Typically use **`movzbl`** to finish job
    - 32-bit instructions also set upper 32 bits to 0
    - (see movz__ instructions in sec. 3.4.2)

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

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

```
    cmpq   %rsi, %rdi    # Compare (x-y)
    setg   %al           # Set when >
    movzbl %al, %eax     # Zero rest of %rax
    ret
```

Practice problem 3.13-A

```
int comp(data_t a, data_t b) {
  return a COMP b;
}
```

Suppose a is in some portion of %rdi and b in %rsi, which data type data_t and which comparison COMP could cause the compiler to generate this code?

A.
```
cmpl %esi,%edi    # a - b
setl %al          # if a < b, set first byte of %rax
```

Answer: l: 32 bits➔ data type is int, COMP is < (signed op.)

Practice problem 3.13-B

```
int comp(data_t a, data_t b) {
  return a COMP b;
}
```

Suppose a is in some portion of %rdi and b in %rsi, which data type data_t and which comparison COMP could cause the compiler to generate this code?

B.
```
cmpw   %si,%di   # a - b
setge  %al       # if a >= b, set first byte of %rax
```

Answer: w: ? bits➔ data type is ??, COMP is ??? (signed op.)

Practice on your own

- Problem 3.13, parts C to D.
- Problem 3.14, p. 205, similar but uses `test` and `set` instructions. (The solution to 3.14.D has an error.)

Access to CC registers (summary of their use)

- Ultimate goal of the CC registers.: implement conditional branches in the assembly code.
- So, instead of reading them directly, they are used to:
  1. set a single byte to 0 or 1, depending on a combination of CCs (sec. 3.6.2)
  2. conditional jump to some other part of the program (sec. 3.6.3 to 3.6.5)
  3. conditional transfer (move) of data (sec. 3.6.6)

End of subsections 3.6.1 and 3.6.2

UNM SCHOOL of ENGINEERING
Department of Computer Science

Condition codes, Jumps, if-else translation

Lecture #9 – part 2

Prof. Soraya Abad-Mota, PhD

(Chap. 3, Section 3.6 on)

Summary of CC registers

- Four single-bit condition code registers
  - CF, ZF, SF, and OF
- These are set after an arithmetic or logical operation is executed (applies to the most recent).
- Except for `leaq` all instructions in fig. 3.10 p. 192 set the condition codes.
- Special cases:
  - Logical ops. set CF and OF to zero
  - shift ops. CF set to the last bit shifted out and OF to 0
  - INC and DEC ops. set OF and ZF, leave CF unchanged

Explicitly setting the CCs

- Arithmetic and logical ops set the CC implicitly
- The instructions in the groups `cmp` and `test` set them explicitly after performing a comparison cmp b,a (a-b) or a test s1,s2 (s1 & s2). (Fig. 3.13, p. 202)
- After doing a cmp or a test, we can use the `set` instructions (sec. 3.6.2, fig. 3.14, p. 203): *to set a single byte to 0 or 1, depending on a combination of the condition codes.*
- Explanation of the set instructions pp. 203-204

Jumping (Sec. 3.6.3)

- jX instructions (fig. 3.15 p. 206)
  - 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)

Understanding the *jump target*

- Study the jump instructions,
- study their encodings, i.e. how to encode where to jump (the jump target):
  - PC-relative: difference between the address of the target instruction and the address of instruction immediately following the jump.
  - Absolute address, using 4 bytes to directly specify the target.

Practice problem 3.15

Target of the je instruction below?

A.
```
4003fa: 74 02    je XXXXXX
4003fc: ff d0    callq *%rax
```

Answer: XXXXXX is?

target of jump? relative to the PC (or %rip)

PC = 4003fc

XXXXXX? 4003fc + 2 = 4003fe

Practice problem 3.15

Target of the je instruction below?

B.

```
40042f: 74 f4      je XXXXXX
400431: 5d         pop  %rbp
```

Answer: XXXXXX is?

xxxxxx = 400431 + f4 = 400425

0xf4 is -12 in 2's complement decimal (400431 – C)

$431_{16} = 1073_{10}$ ➔ 1073 - 12 = 1061 = 425_{16}

Practice problem 3.15

Target of the je instruction below?

c.
```
XXXXXX: 77 02    je 400547
YYYYYY: 5d       pop %rbp
```

Answer: XXXXXX is?

YYYYYY = 400547 - 2 = 400545

XXXXXX = YYYYYY – 2 = 400545 - 2 = 400543

Students finish last one on their own

Learning objectives

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

1. Describe condition codes.
2. Summarize how to set and use condition codes.
3. Describe how are conditional branches implemented in assembly code. (If-then-else statements) (Sec. 3.6.5 and 3.6.6)
4. Describe and practice the translation of loops and the switch statement into assembly code.

Conditional Branch (Old Style) (3.6.5)

- Generation

```
moons> gcc -Og -S -fno-if-conversion control.c
```

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

```
absdiff:
    cmpq    %rsi, %rdi   # x-y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:        # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

Expressing with Goto Code

- C allows **goto** statement
- Jump to position designated by label

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

```
long absdiff_j
  (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

If-then-else statement translation

(general pattern)

If (test-expr) then then-statement

else else-statement

translates using “go to” into: (the old style had 2 returns)

```
t = test-expr;
if (!t)
    go to false;
then-statement
go to done;
false:
    else-statement
done:
```

Practice problems (on your own)

- Problem 3.16, p. 212 (we can do this one now in groups, that way it is fresh in your minds)
- Problem 3.17, p.212
- Problem 3.18, p. 213

Practice Problem 3.16 (solution page 331)

When given the C code

```
void cond(long a, long *p)
{
    if (p && a > *p)
        *p = a;
}
```

GCC generates the following assembly code:

```
  void cond(long a, long *p)
  a in %rdi, p in %rsi
cond:
  testq   %rsi, %rsi
  je      .L1
  cmpq    %rdi, (%rsi)
  jge     .L1
  movq    %rdi, (%rsi)
.L1:
  rep; ret
```

A. Write a goto version in C that performs the same computation and mimics the control flow of the assembly code, in the style shown in Figure 3.16(b). You might find it helpful to first annotate the assembly code as we have done in our examples.

B. Explain why the assembly code contains two conditional branches, even though the C code has only one `if` statement.

Alternative translation

When the then-statement and the else-statement, both compute a value to be assigned to the same variable, the conditional transfer of data can be implemented, by using *conditional moves*

- Figure 3.18 p.217: lists conditional move instructions

Caveat: should only be used in restricted cases because it can have undesirable results (see Bad cases for conditional move slide).

General Conditional Expression Translation (Using Branches)

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

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

Goto Version

```
  ntest = !Test;
  if (ntest) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

Using Conditional Moves

- Conditional Move Instructions
  - Instruction supports:

    if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But, only when known to be safe
- Why use them?
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

C Code

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

Goto Version

```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

Conditional Move Example

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

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

```
absdiff:
    movq    %rdi, %rax   # x
    subq    %rsi, %rax   # result = x-y
    movq    %rsi, %rdx
    subq    %rdi, %rdx   # eval = y-x
    cmpq    %rsi, %rdi   # x:y
    cmovle  %rdx, %rax   # if <=, result = eval
    ret
```

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- makes sense when computations very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

Practice problem (on your own)

- Problem 3.21, p. 219