

# Machine Level Programming: Control

Computer Systems Organization (Spring 2017)  
CSCI-UA 201, Section 3

Instructor: Joanna Klukowska

Slides adapted from  
Randal E. Bryant and David R. O'Hallaron (CMU)  
Mohamed Zahran (NYU)

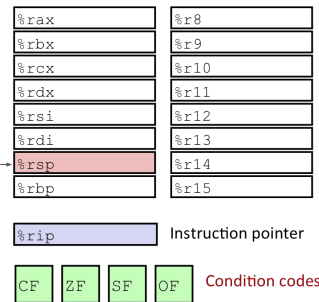
## Condition Codes

2

## Processor State (x86-64, first look)

### 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 )



Current top of the stack

3

## Condition Codes (Implicit Setting)

- Single bit registers
  - **CF** Carry Flag (for unsigned)    **SF** Sign Flag (for signed)
  - **ZF** Zero Flag    **OF** Overflow Flag (for signed)
- Implicitly set (think of it as side effect) by arithmetic operations
- Not set by `leaq` instruction
- Example: `t = a + b`  $\leftrightarrow$  `addq Src, Dest`
  - 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)`

4

## Condition Codes (Explicit Setting - `cmpq`)

- 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 (unsigned overflow)

**ZF** set if  $a == b$

**SF** set if  $(a-b) < 0$  (as signed)

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

$(a > 0 \ \&\& \ b < 0 \ \&\& \ (a-b) < 0) \ || \ (a < 0 \ \&\& \ b > 0 \ \&\& \ (a-b) > 0)$

5

## Condition Codes (Explicit Setting - `testq`)

- Explicit Setting by **Test instruction**

`testq Src2, Src1`

- `testq b, a` like computing  $a \& b$  without setting destination
  - Useful to have one of the operands be a mask

**ZF** set if  $a \& b == 0$

**SF** set if  $a \& b < 0$

6

## Reading Condition Codes

- setX** family of 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
<code>sete</code>	ZF	Equal / Zero
<code>setne</code>	$\sim$ ZF	Not Equal / Not Zero
<code>sets</code>	SF	Negative
<code>setns</code>	$\sim$ SF	Nonnegative
<code>setg</code>	$\sim(SF \wedge OF) \& \sim ZF$	Greater (signed $>$ )
<code>setge</code>	$\sim(SF \wedge OF)$	Greater or Equal (signed $\geq$ )
<code>setl</code>	$(SF \wedge OF)$	Less (signed $<$ )
<code>setle</code>	$(SF \wedge OF)   ZF$	Less or Equal (signed $\leq$ )
<code>seta</code>	$\sim CF \& \sim ZF$	Above (unsigned $>$ )
<code>setb</code>	CF	Below (unsigned $<$ )

7

## x86-64 Integer Registers

- We can reference low-order byte.

<code>%rax</code>	<code>%a1</code>	<code>%r8</code>	<code>%r8b</code>
<code>%rbx</code>	<code>%b1</code>	<code>%r9</code>	<code>%r9b</code>
<code>%rcx</code>	<code>%c1</code>	<code>%r10</code>	<code>%r10b</code>
<code>%rdx</code>	<code>%d1</code>	<code>%r11</code>	<code>%r11b</code>
<code>%rsi</code>	<code>%s11</code>	<code>%r12</code>	<code>%r12b</code>
<code>%rdi</code>	<code>%d11</code>	<code>%r13</code>	<code>%r13b</code>
<code>%rsp</code>	<code>%sp1</code>	<code>%r14</code>	<code>%r14b</code>
<code>%rbp</code>	<code>%bp1</code>	<code>%r15</code>	<code>%r15b</code>

8

## Reading Condition Codes

- **SetX** family of instructions
  - Set single byte based on combination of condition codes
  - Does not alter remaining bytes
  - One of addressable byte registers
- Typically use `movzbl` to finish job
  - 32-bit instructions also set upper 32 bits to 0

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

Register	Use(s)
<code>%rdi</code>	Argument <b>x</b>
<code>%rsi</code>	Argument <b>y</b>
<code>%rax</code>	Return value

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

9

## Conditional Branches

10

## Jumping (in the code)

- **jX** family of instructions
  - Jump to different part of code depending on condition codes

jX	Condition	Description
<code>jmp</code>	<code>1</code>	Unconditional
<code>je</code>	<code>ZF</code>	Equal / Zero
<code>jne</code>	<code>~ZF</code>	Not Equal / Not Zero
<code>js</code>	<code>SF</code>	Negative
<code>jns</code>	<code>~SF</code>	Nonnegative
<code>jg</code>	<code>~(SF^OF)&amp;~ZF</code>	Greater (signed >)
<code>jge</code>	<code>~(SF^OF)</code>	Greater or Equal (signed >=)
<code>jL</code>	<code>(SF^OF)</code>	Less (signed <)
<code>jle</code>	<code>(SF^OF)   ZF</code>	Less or Equal (signed <=)
<code>ja</code>	<code>~CF&amp;~ZF</code>	Above (unsigned >)
<code>jb</code>	<code>CF</code>	Below (unsigned <)

11

## Re-Writing Code with `goto` Statements

- 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;
}
```

Why do this?

- Because the "goto" code is closer to the assembly instructions.

12

## General Conditional Expression Translation

C code:

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

for example:

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

```
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

13

## Using Conditional Moves - cmovXX

- 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

C Code:

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

Goto Version:

```
result = Then_Expr;
eval   = Else_Expr;
nt     = !Test;

if (nt) result = eval;
return result;
```

- Why?
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

14

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

15

## Bad Cases of Conditional Move

- Expensive computations:

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

- both values get calculated
- only makes sense when computations are very simple

- Risky computations

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

- both values get calculated
- may have undesirable side effects (above it is dereferencing a pointer that may be 0)

- Computations with side effects

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

- both values get calculated
- must be side-effect free (unlike the example above)

16

# Loops

## do...while... loop example

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



```
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
- Use conditional branch to either continue looping or to exit loop

## do...while... loop compilation

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

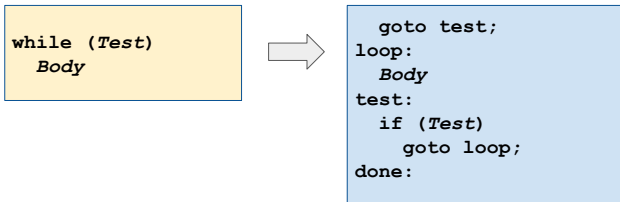
## General do...while... Translation

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



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

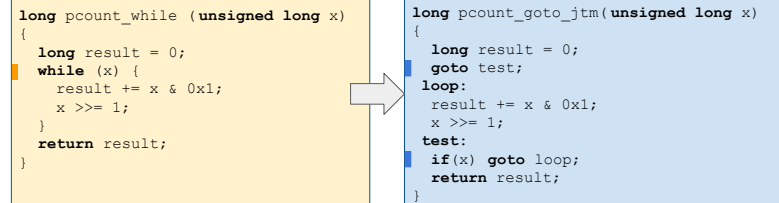
## General `while` Loop Translation (ver. 1)



- “Jump-to-middle” translation
- Used with `-Og` option to `gcc`

21

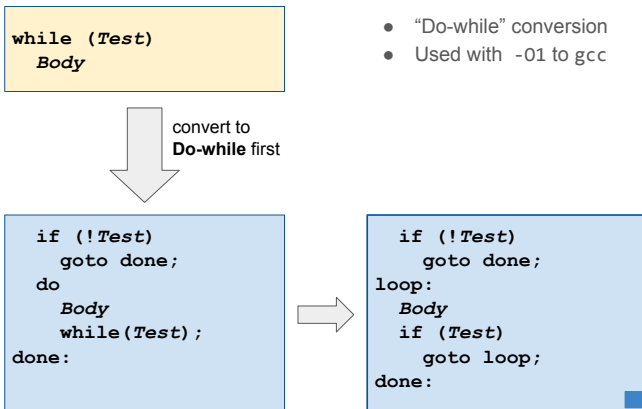
## General `while` Loop Translation (ver. 1)



- Compare to do-while version of function
- Initial `goto` starts loop at test

22

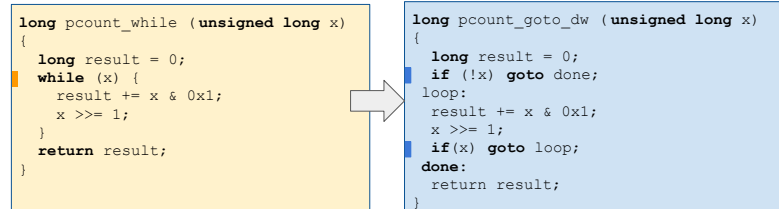
## General `while` Loop Translation (ver. 2)



- “Do-while” conversion
- Used with `-O1` to `gcc`

23

## General `while` Loop Translation (ver. 2)



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

24

## General for Loop Form

General form:

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

Example:

```
#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;

25

## for Loop ⇒ while loop

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



convert to  
while

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

26

## for Loop ⇒ while loop

```
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;
}
```

Init:	i = 0
Test:	i < WSIZE
Update:	i++
Body:	unsigned bit = (x >> i) & 0x1; result += bit;

27

## for Loop ⇒ while loop

```
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

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

Init
!Test
Body
Update
Test

28