Machine Level Programming: Control

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

Instructor: Joanna Klukowska

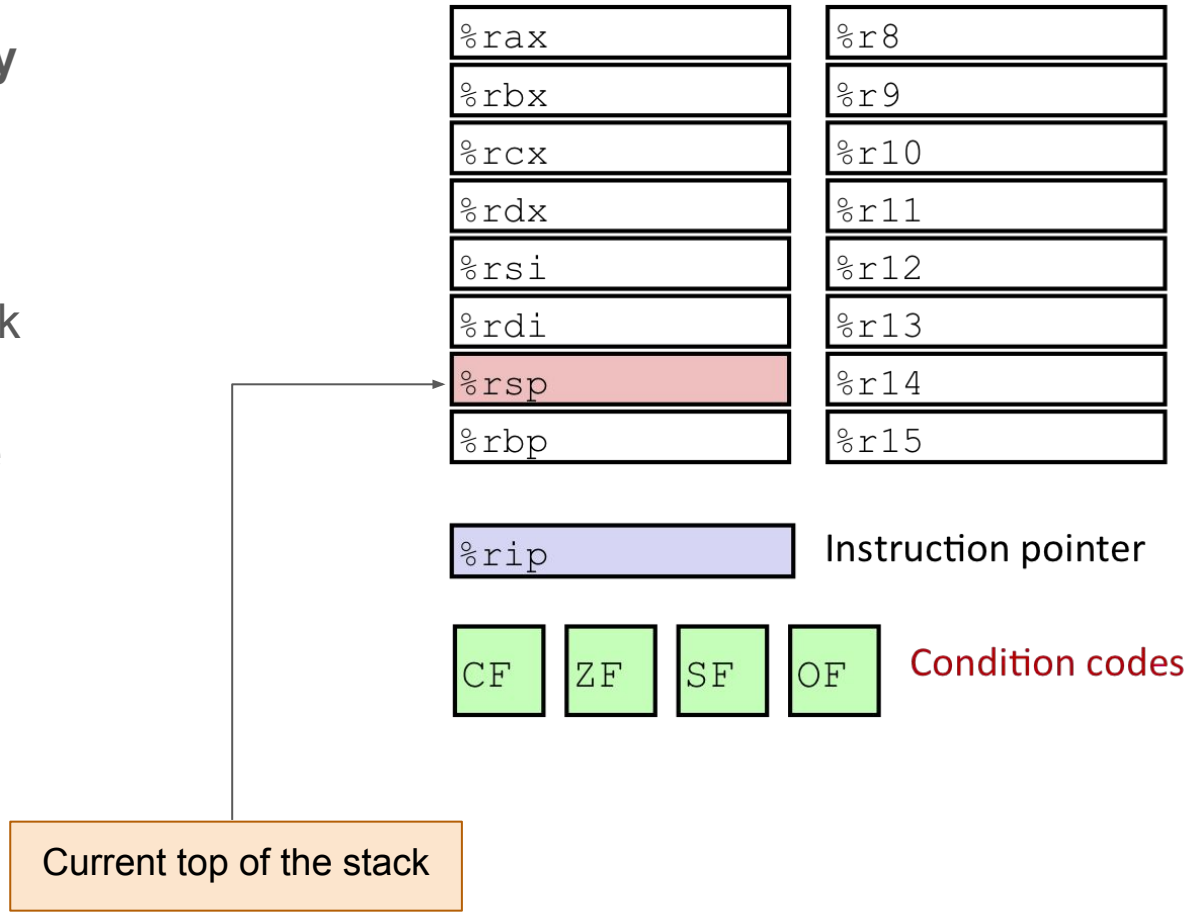
Slides adapted from
Randal E. Bryant and David R. O'Hallaron (CMU)
Mohamed Zahran (NYU)

Condition Codes

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)



Condition Codes (Implicit Setting)

- Single bit registers
  - **CF** Carry Flag (for unsigned)
  - **ZF** Zero Flag
  - **SF** Sign Flag (for signed)
  - **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 <-> 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)
```

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)`

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`

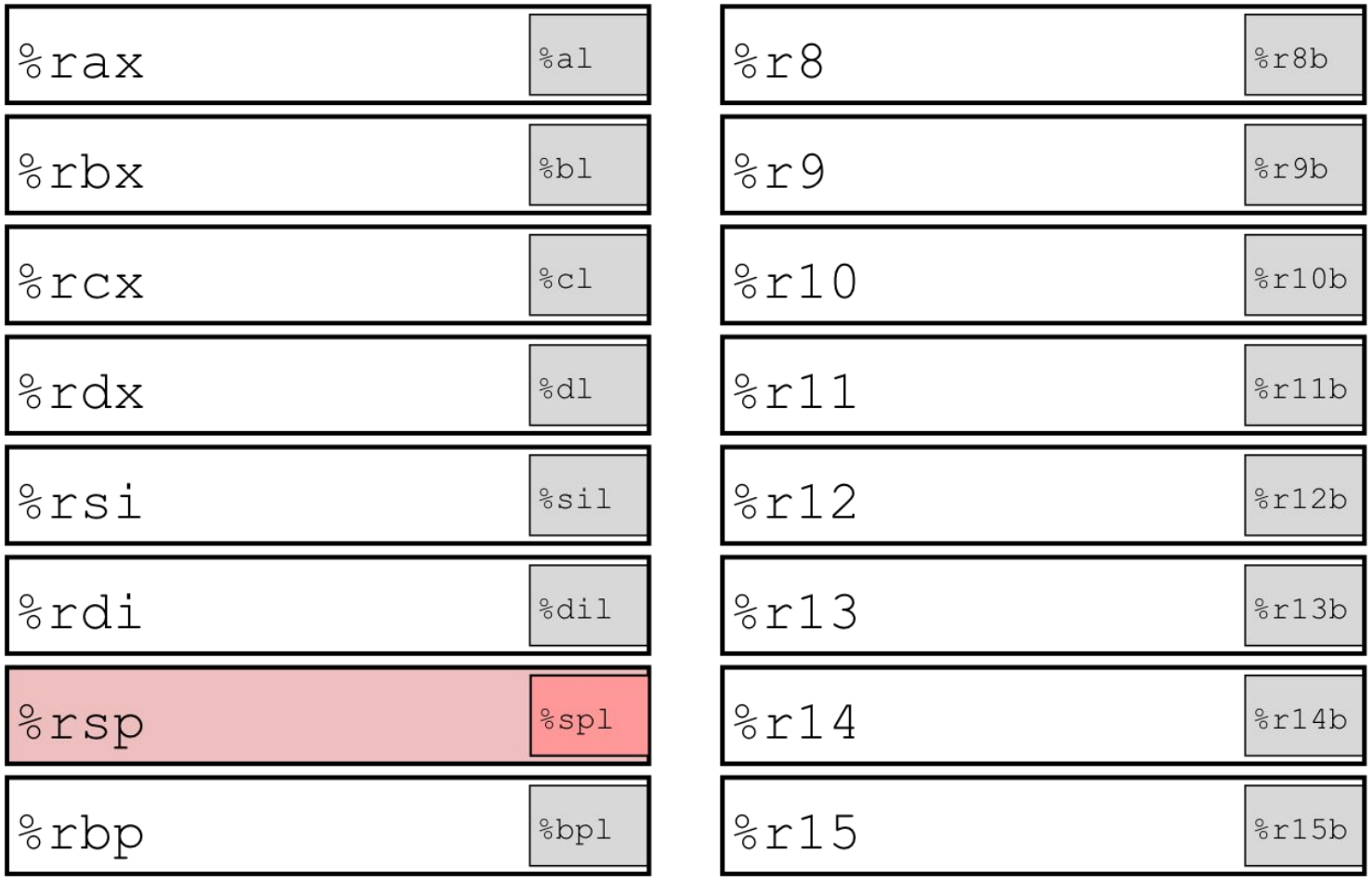
Reading Condition Codes

- `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
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 <)

x86-64 Integer Registers

- We can reference low-order byte.



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)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

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

Conditional Branches

Jumping (in the code)

- `jx` family of instructions
  - 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 <)

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.

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

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

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**

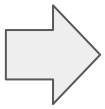
```
val = x > 0 ? x*=7 : x+=3;
```

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

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

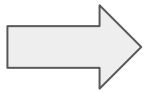
```
long pcount_goto (unsigned long x)
{
  long result = 0;
loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

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

```
      movl    $0, %eax          #  result = 0
   .L2:                         # 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
```

General `do...while`... Translation

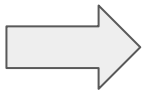
```
do
  Body
  while (Test);
```



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

General `while` Loop Translation (ver. 1)

```
while (Test)
  Body
```

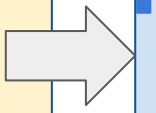


```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

- "Jump-to-middle" translation
- Used with `-Og` option to `gcc`

General `while` Loop Translation (ver. 1)

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

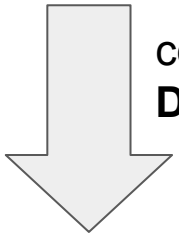


```
long pcount_goto_jtm(unsigned long x)
{
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) goto loop;
  return result;
}
```

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

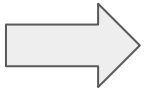
General `while` Loop Translation (ver. 2)

```
while (Test)
  Body
```



convert to
Do-while first

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

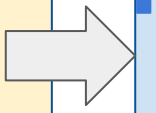


```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

- "Do-while" conversion
- Used with `-O1` to `gcc`

General `while` Loop Translation (ver. 2)

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



```
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

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
```

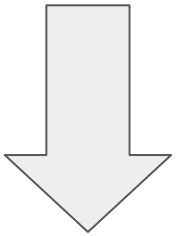
Test:
```
i < WSIZE
```

Update:
```
i++
```

Body:
```
unsigned bit =
     (x >> i) & 0x1;
result += bit;
```

`for` Loop ⇒ `while` loop

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

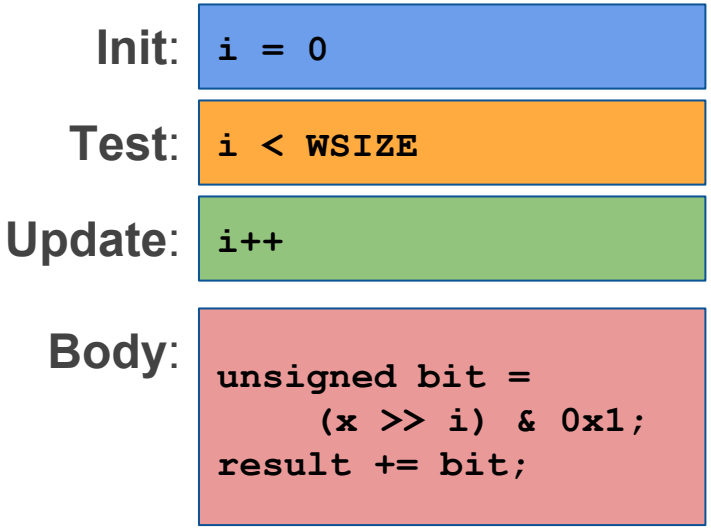


convert to **while**

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

`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;
}
```



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