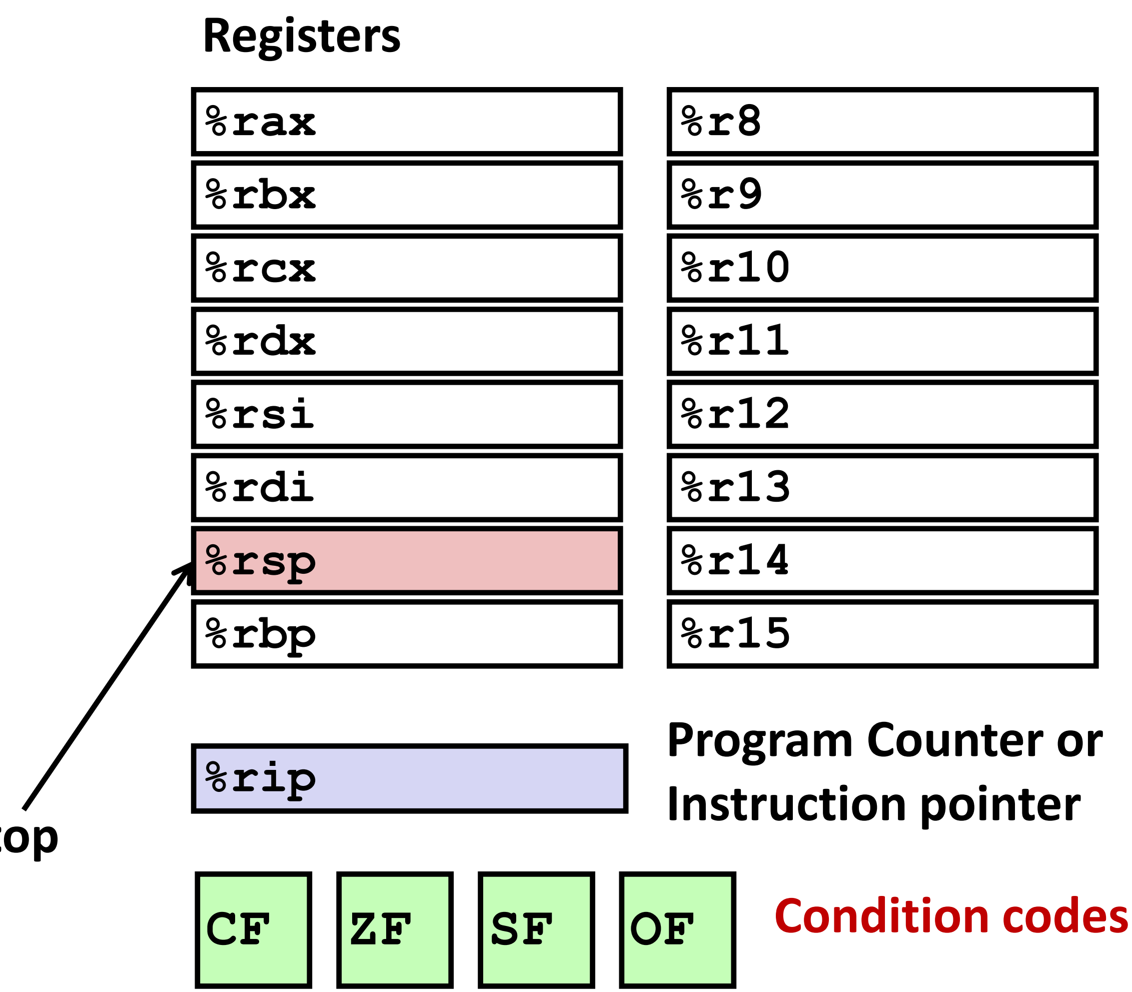
Computer Systems Organization

Topic 3 Contd.

Based on chapter 3 from Computer Systems by Randal E. Bryant and David R. O'Hallaron

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 (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
  - 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 (since intended to be used in address computations)

Example

- Carry flag enables numbers larger than a single ALU width to be added by carrying a binary digit
- Unsigned addition: 1111 + 0111 = 10110, CF = 1
- Unsigned addition: 0111 + 0001 = 1000, CF = 0
- Borrow flag for subtraction if borrow value (represented using Carry flag)
- Unsigned subtraction: 0000 – 0001 = 1111, CF = 1
- Unsigned subtraction: 1000 – 0001 = 0111, CF = 0

Example

- Overflow flag is set when the most significant bit (i.e., sign bit) is changed by adding two numbers with the same sign (or subtracting two numbers with opposite signs). Overflow cannot occur when the sign of two addition operands are different.
- Signed addition: 1111 + 1000 = 10111, CF = 1, SF = 0, OF = 1
- Signed addition: 0111 + 0111 = 1110, CF = 0, SF = 1, OF = 1
- Signed additon: 1111 + 1111 = 11110, CF = 1, SF = 1, OF = 0
- Overflow flag is meaningless for unsigned numbers and normally ignored. It is set for signed numbers so the program can be aware of the problem and mitigate or signal an error.
- Most instruction sets do not distinguish between signed and unsigned operands. Generate both (signed) overflow and (unsigned) carry flags on every operation, and allow to pick later whichever is of interest.

Compare Instruction

- Explicit setting of conditional code 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)`

Example

- Unsigned subtraction: 0000 – 0001 = 1111, CF = 1
- Unsigned subtraction: 1000 – 0001 = 0111, CF = 0
- Signed subtraction: 0001 – 1000 = 1001, CF = 1, SF = 1, OF = 1
- Signed subtraction: 0011 – 1100 = 0111, CF = 1, SF = 0, OF = 0
- Signed subtraction: 1000 – 0001 = 0111, CF = 0, SF = 0, OF = 1
- Signed subtraction: 1001 – 0001 = 1000, CF = 0, SF = 1, OF = 0

Explicitly Setting Condition Codes: Test

- Explicit setting of conditional codes by Test instruction
  - **`testq`** *Src2*, *Src1*
    - **`testq b,a`** like computing **`a&b`** without setting destination (performs bitwise AND)
  - 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`**

Example

- testq %rax %rax sets ZF or SF – hence used to test whether a value is negative, zero or positive
- One of the values can be a mask e.g., test the last bit
  - Mask would be 00..01 – If ZF set, last bit is 0
  - In general, can test nth bit or a subset of the expression in general

SET Instructions

- Rather than reading the conditional codes directly, set a single byte to 0 or 1 depending on some combination of the condition codes.
  - SET instructions are useful to model this
- For conditional codes set using cmpq i.e., t = a-b
  - If a, b and t are integers represented in 2's complement form
  - Consider sete or "set when equal" - when a == b, t = 0 and hence zero flag indicates equality

SET Instructions

- Consider setl or “set when less”
  - When no overflow occurs (OF set to 0), we will have a < b when a-b < 0 indicated by having SF set to 1. Similarly, we will have a >= b when a-b >= 0 indicated by having SF set to 0
  - When overflow occurs, we will have a < b when a-b > 0 (negative overflow) and a > b when a-b < 0 (positive overflow)
  - cannot have overflow when a = b
  - In summary , when OF is set to 1, we will have a < b only if SF is set to 0
  - Combining the EXCLUSIVE-OR of the overflow and sign bits provides a test for whether a < b
  - Signed comparison tests are based on combinations of SF, CF, OF and ZF

SET Instructions

- For unsigned comparisons of variables a and b, for t = a-b, carry flag will be set by CMP instruction when a-b < 0 (uses combinations of carry and zero flags)
- Machine code does not distinguish between signed and unsigned values since many arithmetic operations have the same bit level behavior for unsigned and 2’s complement arithmetic.
- Some circumstances can need handling of signed vs. unsigned operations e.g., right shifts [Sign extend for Arithmetic (or Signed) Shift while 0 extend for Logical Shift]

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 D	D ← ZF	Equal / Zero
setne D	D ← ~ZF	Not Equal / Not Zero
sets D	D ← SF	Negative
setns D	D ← ~SF	Nonnegative
setg D	D ← ~(SF^OF)&~ZF	Greater (Signed)
setge D	D ← ~(SF^OF)	Greater or Equal (Signed)
setl D	D ← (SF^OF)	Less (Signed)
setle D	D ← (SF^OF)|ZF	Less or Equal (Signed)
seta D	D ← ~CF&~ZF	Above (unsigned >)
setb D	D ← CF	Below (unsigned <)

x86-64 Integer Registers

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bpl	%r15	%r15b

- Can reference low-order byte

Reading Condition Codes

- 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
    - Move zero-extended byte to double word
    - 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 y:x
    setg   %al          # Set when >
    movzbl %al, %eax    # Zero rest of %eax (and %rax)
    ret
```

Details on ret

- By convention, %rax is used to store a function’s return value, if it exists and is no more than 64 bits long.
- Registers %rbx, %rbp, and %r12-r15 are callee-save registers, meaning that they are saved across function calls.
- Additionally, %rdi, %rsi, %rdx, %rcx, %r8, and %r9 are used to pass the first six integer or pointer parameters to called functions.

Conditional Branches: Jumping

- **jX 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)

Conditional Branch Example (Old Style)

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

```
absdiff:
   cmpq    %rsi, %rdi   # y:x
   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;
}
```

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?
  - 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

```c
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  # y:x
   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
- Only makes sense when computations are 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

Loops: "Do-While" Loop Example

C Code

```
long fact_do
  (long n) {
  long result = 1;
  do {
    result *= n;
    n = n-1;
  } while (n > 1);
  return result;
}
```

Goto Version

```
long fact_do_goto
  (long x) {
  long result = 1;
 loop:
  result *= n;
  n = n-1;
  if(n > 1) goto loop;
  return result;
}
```

- Compute n factorial
- Use conditional branch to either continue looping or to exit loop

"Do-While" Loop Compilation

Goto Version

```
long fact_goto
  (long x) {
  long result = 1;
 loop:
  result *= n;
  n = n-1;
  if(n > 1) goto loop;
  return result;
}
```

Register	Use(s)
%rdi	n
%rax	result

```
   movl    $1, %eax     #  result = 1
.L2:                   #  loop:
   imulq   %rdi, %rax
   subq    $1, %rdi     #  Decrement n
   cmpq    $1, %rdi     #  Compare n:1
   jg      .L2          #  if >, goto loop
   rep; ret
```

General "Do-While" Translation

C Code

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

Goto Version

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

- **Body:** {
  $\texttt{Statement}_1$;
  $\texttt{Statement}_2$;
  …
  $\texttt{Statement}_n$;
  }

General "While" Translation #1

- "Jump-to-middle" translation

While version

```
while (Test)
  Body
```

Goto Version

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

While Loop Example #1

C Code

```
long fact_while
  (long n) {
  long result = 1;
  while (n > 1) {
    result *= n;
    n = n-1;
  }
  return result;
}
```

Jump to Middle

```
long fact_while_jtm_goto
  (long n) {
  long result = 1;
  goto test;
 loop:
  result *= n;
  n = n-1;
 test:
  if(n > 1) goto loop;
  return result;
}
```

- Compare to do-while version of function
- 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:
```

- "Do-while" translation
- Used with `-O1 (higher level of optimization in GCC)`

Goto Version

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

While Loop Example #2

C Code

```
long fact_while
  (long n) {
  long result = 1;
  while (n > 1) {
    result *= n;
    n = n-1;
  }
  return result;
}
```

Do-While (or Guarded Do) Version

```
long fact_while_gd_goto
  (long n) {
  long result = 1;
  if (n <= 1) goto done;
 loop:
  result *= n;
  n = n-1;
  if(n != 1) goto loop;
 done:
  return result;
}
```

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

"For" Loop Form

General Form

for (*Init* ; *Test* ; *Update*)

    Body

```
long fact_for
  (long n)
{
  long i;
  long result = 1;
  for (i = 2; i <= n; i++)
    result *= i;
  return result;
}
```

Init

```
i = 2
```

Test

```
i <= n
```

Update

```
i++
```

Body

```
result *= i;
```

"For" Loop → While Loop

For Version

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

While Version

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

For-While Conversion

Init

```
i = 2
```

Test

```
i <= n
```

Update

```
i++
```

Body

```
result *= i
```

```
long fact_for_while
  (long n)
{
  long i = 2;
  long result = 1;
  while (i <= n)
  {
    result *= i;
    i++;
  }
  return result;
}
```

"For" Loop Do-While Conversion

C Code

```
long fact_for (long n)
{
  long i;
  long result = 1;
  for (i = 2; i <= n; i++)
  {
    result *= i;
  }
  return result;
}
```

Goto Version

```
long fact_for_jm_goto
   (long n) {
   long i = 2;
   long result = 1;
   goto test;
 loop:
   result *= i;
   i++;
 test:
   if (i <= n)
     goto loop;
   return result;
}
```

Switch Statement: An example

```
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

Jump Table Structure

Switch 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

jtab:
Targ0
Targ1
Targ2
• • •
Targ*n*-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

•
•
•

Targ*n*-1: Code Block *n*–1

Jump Table jt

- Array where entry i is the address of a code segment implementing the action the program should take when the switch index equals i
- Advantage of using jt is the time taken to perform the switch is independent of the number of switch cases
- Jt used when >= 4 cases

Switch Statement Example

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

Setup:

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

What range of values takes default?

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

Note that w not initialized here

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    Align address
to multiple of 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`

- 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 $0 \le$ **x** ≤ 6

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
```

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;
}
```

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`**
`%rcx`	Argument **`z`**
`%rax`	Return value

Code Blocks

- cqto: sign extend rax to rdx:rax
- idivq S:
  - Signed divide %rdx:%rax by S
  - Quotient stored in %rax
  - Remainder stored in %rdx
- Figure 3.12 of book

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   %rcx, %rax #   w -= z
  ret
.L8:                       # Default:
  movl   $2, %eax     #   2
  ret
```

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

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 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)