

CS 33

Machine Programming (2)

Processor State (x86-64, Partial)

| | |
|-------------|-------------|
| %rax | %eax |
| %rbx | %ebx |
| %rcx | %ecx |
| %rdx | %edx |
| %rsi | %esi |
| %rdi | %edi |
| %rsp | %esp |
| %rbp | %ebp |
| %rip | |

| | |
|-------------|--------------|
| %r8 | %r8d |
| %r9 | %r9d |
| %r10 | %r10d |
| %r11 | %r11d |
| %r12 | %r12d |
| %r13 | %r13d |
| %r14 | %r14d |
| %r15 | %r15d |

| | | | |
|-----------|-----------|-----------|-----------|
| CF | ZF | SF | OF |
|-----------|-----------|-----------|-----------|

condition codes

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: *addl/addq* Src, Dest \leftrightarrow *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 *leal* instruction**

Condition Codes (Explicit Setting: Compare)

- **Explicit setting by compare instruction**

`cmpl/cmpq src2, src1`

`cmpl 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)`

Condition Codes (Explicit Setting: Test)

- **Explicit setting by test instruction**

`testl/testq src2, src1`

`testl 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 single byte based on combinations of condition codes

| SetX | Condition | Description |
|-------|--------------------------------------|---------------------------|
| sete | ZF | Equal / Zero |
| setne | $\sim ZF$ | Not Equal / Not Zero |
| sets | SF | Negative |
| setns | $\sim SF$ | Nonnegative |
| setg | $\sim (SF \wedge OF) \ \& \ \sim ZF$ | Greater (Signed) |
| setge | $\sim (SF \wedge OF)$ | Greater or Equal (Signed) |
| setl | $(SF \wedge OF)$ | Less (Signed) |
| setle | $(SF \wedge OF) \mid ZF$ | Less or Equal (Signed) |
| seta | $\sim CF \ \& \ \sim ZF$ | Above (unsigned) |
| setb | CF | Below (unsigned) |

Reading Condition Codes (Cont.)

- **SetX instructions:**
 - set single byte based on combination of condition codes
- **Uses one of 8 addressable byte registers**
 - does not alter remaining 7 bytes
 - typically use `movzbl` to finish job

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

| | | | |
|------|------|-----|-----|
| %rax | %eax | %ah | %al |
|------|------|-----|-----|

Body

```
cmpl %esi, %edi    # compare x : y
setg %al           # %al = x > y
movzbl %al, %eax    # zero rest of %eax/%rax
```

Jumping

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

| jX | Condition | Description |
|-----|--------------------------------------|---------------------------|
| jmp | 1 | Unconditional |
| je | ZF | Equal / Zero |
| jne | $\sim ZF$ | Not Equal / Not Zero |
| js | SF | Negative |
| jns | $\sim SF$ | Nonnegative |
| jg | $\sim (SF \wedge OF) \ \& \ \sim ZF$ | Greater (Signed) |
| jge | $\sim (SF \wedge OF)$ | Greater or Equal (Signed) |
| jl | $(SF \wedge OF)$ | Less (Signed) |
| jle | $(SF \wedge OF) \mid ZF$ | Less or Equal (Signed) |
| ja | $\sim CF \ \& \ \sim ZF$ | Above (unsigned) |
| jb | CF | Below (unsigned) |

Jumping

- **jX instructions**
 - Jump to different

Quiz 1

What would be an appropriate description if the condition is $\sim CF$?

- a) above or equal (unsigned)
- b) not less (signed)
- c) incomparable

| jX | Condition | Description |
|-----|--------------------------------------|---------------------------|
| jmp | 1 | Unconditional |
| je | ZF | Equal / Zero |
| jne | $\sim ZF$ | Not Equal / Not Zero |
| js | SF | Negative |
| jns | $\sim SF$ | Nonnegative |
| jg | $\sim (SF \wedge OF) \ \& \ \sim ZF$ | Greater (Signed) |
| jge | $\sim (SF \wedge OF)$ | Greater or Equal (Signed) |
| jl | $(SF \wedge OF)$ | Less (Signed) |
| jle | $(SF \wedge OF) \mid ZF$ | Less or Equal (Signed) |
| ja | $\sim CF \ \& \ \sim ZF$ | Above (unsigned) |
| jb | CF | Below (unsigned) |

Conditional-Branch Example

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

```
absdiff:
    movl    %esi, %eax
    cmpl    %esi, %edi
    jle     .L6
    subl    %eax, %edx
    movl    %edx, %eax
    jmp     .L7
.L6:
    subl    %edx, %eax
.L7:
    ret
```

Body1

Body2a

Body2b

Conditional-Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
    goto Exit;
Else:
    result = y-x;
Exit:
    return result;
}
```

```
absdiff:
    movl    %esi, %eax
    cmpl    %esi, %edi
    jle     .L6
    subl    %eax, %edx
    movl    %edx, %eax
    jmp     .L7
.L6:
    subl    %edx, %eax
.L7:
    ret
```

Body1

Body2a

Body2b

- **C allows “goto” as means of transferring control**
 - closer to machine-level programming style
- **Generally considered bad coding style**

General Conditional-Expression Translation

C Code

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

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

Goto Version

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

- Test is expression returning integer
 - == 0 interpreted as false
 - ≠ 0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one

Using Conditional Moves

- **Conditional move instructions**

- instruction supports:
if (Test) Dest \leftarrow Src
- supported in post-1995 x86 processors
- gcc does not always use them
 - » wants to preserve compatibility with ancient processors
 - » enabled for x86-64
 - » use switch `-march=686` for IA32

- **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

```
tval = Then_Expr;  
result = Else_Expr;  
t = Test;  
if (t) result = tval;  
return result;
```

Conditional Move Example: x86-64

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

x in %edi

y in %esi

absdiff:

```
movl    %edi, %eax  
subl    %esi, %eax    # result = x-y  
movl    %esi, %edx  
subl    %edi, %edx    # tval = y-x  
cmpl    %esi, %edi    # compare x:y  
cmovle  %edx, %eax    # if <=, result = tval  
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

“Do-While” Loop Example

C Code

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

Goto Version

```
int pcount_do(unsigned x)
{
    int 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 either to continue looping or to exit loop

“Do-While” Loop Compilation

Goto Version

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

Registers:

| | |
|------|--------|
| %edx | x |
| %eax | result |

| | | | |
|------|------------|---|------------------|
| movl | \$0, %eax | # | result = 0 |
| .L2: | | # | loop: |
| movl | %edx, %ecx | | |
| andl | \$1, %ecx | # | t = x & 1 |
| addl | %ecx, %eax | # | result += t |
| shrl | %edx | # | x >>= 1 |
| jne | .L2 | # | if !0, goto loop |

General “Do-While” Translation

C Code

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

- **Body:**

```
{  
    Statement1;  
    Statement2;  
    ...  
    Statementn;  
}
```
- **Test returns integer**
 - = 0 interpreted as false
 - ≠ 0 interpreted as true

Goto Version

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

“While” Loop Example

C Code

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

Goto Version

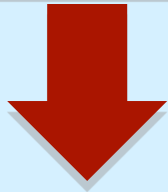
```
int pcount_do(unsigned x) {  
    int result = 0;  
    if (!x) goto done;  
loop:  
    result += x & 0x1;  
    x >>= 1;  
    if (x)  
        goto loop;  
done:  
    return result;  
}
```

- Is this code equivalent to the do-while version?
 - must jump out of loop if test fails

General “While” Translation

While version

```
while (Test)  
  Body
```



Do-While Version

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



Goto Version

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

“For” Loop Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
    int i;
    int result = 0;
    for (i = 0; i < WSIZE; i++) {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    return result;
}
```

- Is this code equivalent to other versions?

“For” Loop Form

General Form

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

```
for (i = 0; i < WSIZE; i++) {  
    unsigned mask = 1 << i;  
    result += (x & mask) != 0;  
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

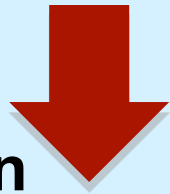
Body

```
{  
    unsigned mask = 1 << i;  
    result += (x & mask) != 0;  
}
```

“For” Loop → While Loop

For Version

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



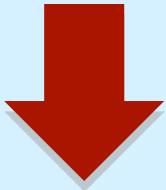
While Version

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

“For” Loop → ... → Goto

For Version

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

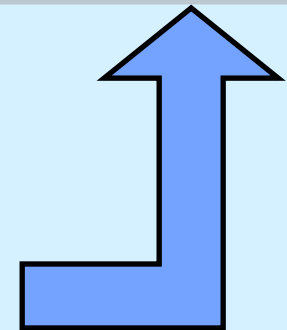


While Version

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



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



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


“For” Loop Conversion Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
    int i;
    int result = 0;
    for (i = 0; i < WSIZE; i++) {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    return result;
}
```

Initial test can be optimized away

Goto Version

```
int pcount_for_gt(unsigned x) {
    int i;
    int result = 0;      Init
    i = 0;
    if (!(i < WSIZE)) !Test
    goto done;
loop:
    Body
    {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    i++; Update
    if (i < WSIZE) Test
        goto loop;
done:
    return result;
}
```

Switch-Statement 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  
}
```

Approximate Translation

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

Jump Table

| | |
|-------|---------|
| jtab: | Targ0 |
| | Targ1 |
| | Targ2 |
| | • |
| | • |
| | • |
| | Targn-1 |

Jump Targets

Targ0:

Code Block 0

Targ1:

Code Block 1

Targ2:

Code Block 2

•
•
•

Targn-1:

Code Block n-1

Switch-Statement Example (x86-64)

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

What range of values is covered by the default case?

Setup:

switch_eg:

```
...                               # Setup
movq    %rdx, %rcx                # %rcx = z
cmpq    $6, %rdi                  # Compare x:6
ja      .L8                        # If unsigned > goto default
jmp     *.L7(, %rdi, 8)            # Goto *JTab[x]
```

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;
}
```

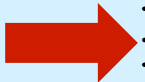
Jump table

```
.section      .rodata
    .align 4
.L7:
    .quad     .L8 # x = 0
    .quad     .L3 # x = 1
    .quad     .L4 # x = 2
    .quad     .L9 # x = 3
    .quad     .L8 # x = 4
    .quad     .L6 # x = 5
    .quad     .L6 # x = 6
```

Setup:

```
switch_eg:
    ...
    movq      %rdx, %rcx      # Setup
                                # %rcx = z
    cmpq      $6, %rdi        # Compare x:6
    ja        .L8              # If unsigned > goto default
    jmp       *.L7(, %rdi, 8)   # Goto *JTab[x]
```

Indirect
jump



Assembly-Setup Explanation

- Table structure

- each target requires 8 bytes
- base address at .L7

- Jumping

direct: `jmp .L8`

- jump target is denoted by label .L8

indirect: `jmp *.L7(, %rdi, 8)`

- start of jump table: .L7
- must scale by factor of 8 (labels have 8 bytes on x86-64)
- fetch target from effective address `.L7 + rdi*8`
 - » only for $0 \leq x \leq 6$

Jump table

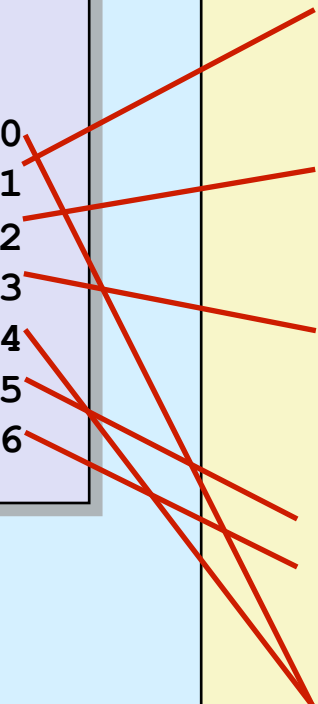
```
.section .rodata
.align 4
.L7:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L4 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L6 # x = 5
    .quad .L6 # x = 6
```

Jump Table

Jump table

```
.section .rodata
.align 4
.L7:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L4 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L6 # x = 5
.quad .L6 # x = 6
```

```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
case 2:      // .L4
    w = y/z;
    /* Fall Through */
case 3:      // .L9
    w += z;
    break;
case 5:
case 6:      // .L6
    w -= z;
    break;
default:    // .L8
    w = 2;
}
```



Code Blocks (Partial)

```
switch(x) {  
  case 1:          // .L3  
    w = y*z;  
    break;  
  
  . . .  
  case 5:          // .L6  
  case 6:          // .L6  
    w -= z;  
    break;  
  default:         // .L8  
    w = 2;  
}
```

```
.L3:                # x == 1  
    movl %rsi, %rax # y  
    imulq %rdx, %rax # w = y*z  
    ret  
  
.L6:                # x == 5, x == 6  
    movl $1, %eax # w = 1  
    subq %rdx, %rax # w -= z  
    ret  
  
.L8:                # Default  
    movl $2, %eax # w = 2  
    ret
```


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;

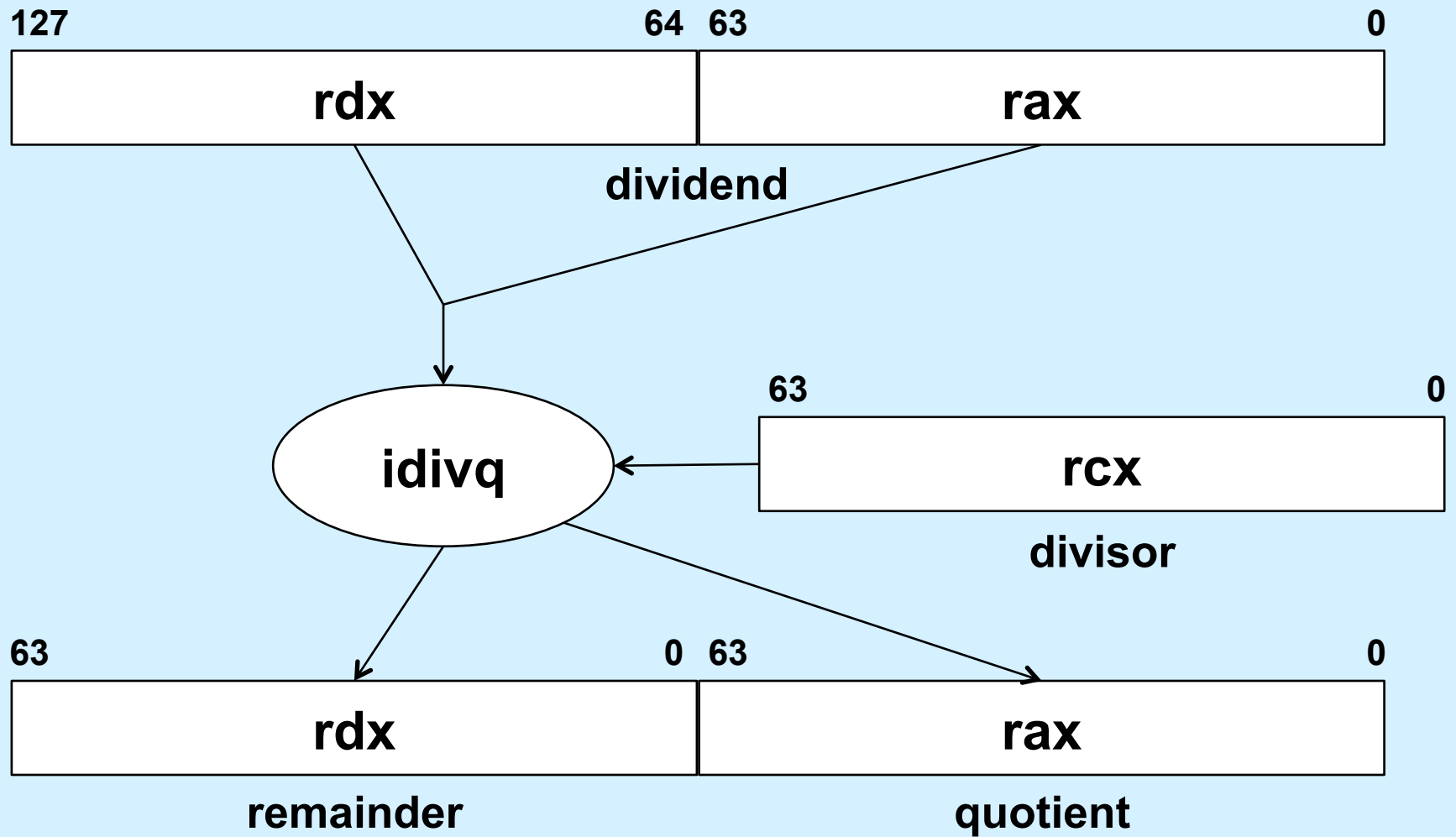
| |
|-------------------|
| case 3: w = 1; |
| merge: w += z; |

Code Blocks (Rest)

```
switch(x) {  
    . . .  
    case 2:  // .L4  
        w = y/z;  
        /* Fall Through */  
    case 3:  // .L9  
        w += z;  
        break;  
    . . .  
}
```

```
.L4:                # x == 2  
    movq    %rsi, %rax  
    movq    %rsi, %rdx  
    sarq    $63, %rdx  
    idivq   %rcx      # w = y/z  
    jmp     .L5  
.L9:                # x == 3  
    movl    $1, %eax  # w = 1  
.L5:                # merge:  
    addq    %rcx, %rax # w += z  
    ret
```

idivq



x86-64 Object Code

- **Setup**

- label `.L8` becomes address `0x4004e5`
- label `.L7` becomes address `0x4005c0`

Assembly code

```
switch_eg:
    . . .
    ja      .L8          # If unsigned > goto default
    jmp     *.L7(, %rdi, 8) # Goto *JTab[x]
```

Disassembled object code

```
00000000004004ac <switch_eg>:
    . . .
4004b3: 77 30                      ja      4004e5 <switch_eg+0x39>
4004b5: ff 24 fd c0 05 40 00      jmpq    *0x4005c0(, %rdi, 8)
```

x86-64 Object Code (cont.)

- **Jump table**
 - doesn't show up in disassembled code
 - can inspect using gdb

`gdb switch`

`(gdb) x/7xg 0x4005c0`

- » examine 7 hexadecimal format “giant” words (8-bytes each)
- » use command “`help x`” to get format documentation

| | | |
|------------------------|----------------------------------|----------------------------------|
| <code>0x4005c0:</code> | <code>0x000000000004004e5</code> | <code>0x000000000004004bc</code> |
| <code>0x4005d0:</code> | <code>0x000000000004004c4</code> | <code>0x000000000004004d3</code> |
| <code>0x4005e0:</code> | <code>0x000000000004004e5</code> | <code>0x000000000004004dc</code> |
| <code>0x4005f0:</code> | <code>0x000000000004004dc</code> | |

x86-64 Object Code (cont.)

- Deciphering jump table

| | | |
|-----------|--------------------|--------------------|
| 0x4005c0: | 0x00000000004004e5 | 0x00000000004004bc |
| 0x4005d0: | 0x00000000004004c4 | 0x00000000004004d3 |
| 0x4005e0: | 0x00000000004004e5 | 0x00000000004004dc |
| 0x4005f0: | 0x00000000004004dc | |

| Address | Value | x |
|----------|----------|---|
| 0x4005c0 | 0x4004e5 | 0 |
| 0x4005c8 | 0x4004bc | 1 |
| 0x4005d0 | 0x4004c4 | 2 |
| 0x4005d8 | 0x4004d3 | 3 |
| 0x4005e0 | 0x4004e5 | 4 |
| 0x4005e8 | 0x4004dc | 5 |
| 0x4005f0 | 0x4004dc | 6 |

Disassembled Targets

```
(gdb) disassemble 0x4004bc,0x4004eb
```

```
Dump of assembler code from 0x4004bc to 0x4004eb
```

```
0x00000000004004bc <switch_eg+16>:  mov    %rsi,%rax
0x00000000004004bf <switch_eg+19>:  imul   %rdx,%rax
0x00000000004004c3 <switch_eg+23>:  retq
0x00000000004004c4 <switch_eg+24>:  mov    %rsi,%rax
0x00000000004004c7 <switch_eg+27>:  mov    %rsi,%rdx
0x00000000004004ca <switch_eg+30>:  sar    $0x3f,%rdx
0x00000000004004ce <switch_eg+34>:  idiv   %rcx
0x00000000004004d1 <switch_eg+37>:  jmp    0x4004d8 <switch_eg+44>
0x00000000004004d3 <switch_eg+39>:  mov    $0x1,%eax
0x00000000004004d8 <switch_eg+44>:  add    %rcx,%rax
0x00000000004004db <switch_eg+47>:  retq
0x00000000004004dc <switch_eg+48>:  mov    $0x1,%eax
0x00000000004004e1 <switch_eg+53>:  sub    %rdx,%rax
0x00000000004004e4 <switch_eg+56>:  retq
0x00000000004004e5 <switch_eg+57>:  mov    $0x2,%eax
0x00000000004004ea <switch_eg+62>:  retq
```

Matching Disassembled Targets

| Value | x |
|----------|---|
| 0x4004e5 | 0 |
| 0x4004bc | 1 |
| 0x4004c4 | 2 |
| 0x4004d3 | 3 |
| 0x4004e5 | 4 |
| 0x4004dc | 5 |
| 0x4004dc | 6 |

```
0x00000000004004bc:  mov    %rsi,%rax
0x00000000004004bf:  imul   %rdx,%rax
0x00000000004004c3:  retq
0x00000000004004c4:  mov    %rsi,%rax
0x00000000004004c7:  mov    %rsi,%rdx
0x00000000004004ca:  sar    $0x3f,%rdx
0x00000000004004ce:  idiv   %rcx
0x00000000004004d1:  jmp    0x4004d8
0x00000000004004d3:  mov    $0x1,%eax
0x00000000004004d8:  add    %rcx,%rax
0x00000000004004db:  retq
0x00000000004004dc:  mov    $0x1,%eax
0x00000000004004e1:  sub    %rdx,%rax
0x00000000004004e4:  retq
0x00000000004004e5:  mov    $0x2,%eax
0x00000000004004ea:  retq
```