Machine-Level Programming: Loops and switches statements

Arquitectura de Computadores
Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Luís Nogueira (lmn@isep.ipp.pt)

Today

- Loops
- Switch statements

Loops

- **C provides several looping constructs - namely, *do-while*, *while*, and *for***

- **No corresponding instructions exist in machine code**
  - Instead, combinations of conditional tests and jumps are used to implement the effect of loops

- **We will study the translation of loops as a progression, starting with *do-while* and then working toward ones with more complex implementations**
  - Most compilers generate loop code based on the *do-while* form of a loop

“Do-While” loop example

- Count the number of bits 1 in argument **x**

C code

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

Goto version

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

- Use conditional branch to either continue or exit loop

“Do-While” loop in Assembly

Register	Use
%rdi	Argument x
%rax	result

```
  movq   $0,%rax          #  result = 0
.L2:                      # loop:
  movq   %rdi,%rdx
  andq   $1,%rdx          #  t = x & 1
  addq   %rdx,%rax        #  result += t
  shrq   %rdi             #  x >>= 1
  jnz    .L2              #  if !0, goto loop
```

Goto version

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

General "Do-While" translation

C code

```
do
  statement1
  ...
  statementn
while(Test);
```

Body

Goto version

```
loop:
  statement1
  ...
  statementn
  if(Test)
    goto loop
```

Body

- **Test returns integer**
  - = 0 interpreted as false
  - ≠ 0 interpreted as true

“While” loop example

C code

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

- **It differs from *do-while* in that *test-expr* is first evaluated**
  - The loop is potentially terminated before the first execution of *body-statement*
- **There are a number of ways to translate a *while* loop into machine code**

General "While" translation #1

- **"Jump-to-middle" translation**
  - Used with **–Og**

While version

```
while(Test){
   Body
}
```

Goto version

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

- **Avoids duplicating test code**
- **Unconditional jump incurs no performance penalty on modern CPUs**
  - It occupies a decode unit but never makes it into the main pipeline

While loop example #1

C code

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

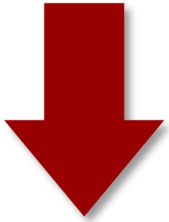
Jump to middle version

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

General "While" translation #2

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:
```

- **"Do-while" conversion**
  - Used with `-O1`

"While" loop example #2

C code

```
long pcount_while(unsigned long x)
{
  int result = 0;

  while(x){
    result += x & 0x1;
    x >>= 1;
  }

  return result;
}
```

Goto Version

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

- **The compiler can often optimize the initial test**
  - For example, determining that the test condition will always hold

General “For” loop form

General form

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

Example

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

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

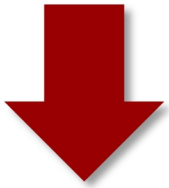
Body

```
{
  unsigned long 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;
}
```

Do-While version

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

Goto Version

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

"For" loop conversion example

C code

```
#define WSIZE 8*sizeof(int)

long pcount_for(unsigned long x)
{
  int i;
  long result = 0;

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

  return result;
}
```

Goto version

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

The *loop* instructions

- **Use the RCX register as a counter and automatically decrease its value as the loop instruction is executed**
  - Without affecting the EFLAGS register flag bits when RCX reaches zero

- **Support only an 8-bit offset, so only short jumps can be performed**

loopX	Condition	Description
`loop`	`RCX != 0`	Loop until the RCX register is zero
`loope/loopz`	`RCX != 0 or ZF`	Loop until either the RCX register is zero, or the ZF flag is not set
`loopne/loopnz`	`RCX != 0 and ~ZF`	Loop until either the RCX register is zero, or the ZF flag is set

The *loop* instructions example

C code

```
for (i = 100; i > 0; i--)
{
    …
}
```

Assembly *loop* version

```
  movq $100,%rcx
for_loop:
  …
  loop for_loop
```

- **Be careful with code inside the loop**
  - If the RCX register is modified, it will affect the operation of the loop
  - Function calls within the loop can easily trash the value of the RCX register without you knowing it
  - If RCX is already <= 0 before the loop, it will eventually exit when the register overflows

Today

- Complete addressing mode, address computation with *leal*
- Control: Condition codes
- Accessing the condition codes
- Loops
- **Switch statements**

Switch statements

- **Provide a multi-way branching capability based on the value of an integer index**
  - Particularly useful when dealing with tests where there can be a large number of possible outcome
- **Large blocks are implemented using a *jump table***
  - An array where entry *i* is the address of a code segment implementing the action the program should take when the switch index equals *i*
- **The time taken to perform the switch is independent of the number of switch cases**
  - As opposed to a long sequence of if-else statements

```
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;
}
```

Example

- **Multiple case labels**
  - case 5 & 6
- **Fall through cases**
  - case 2
- **Missing cases**
  - case 4

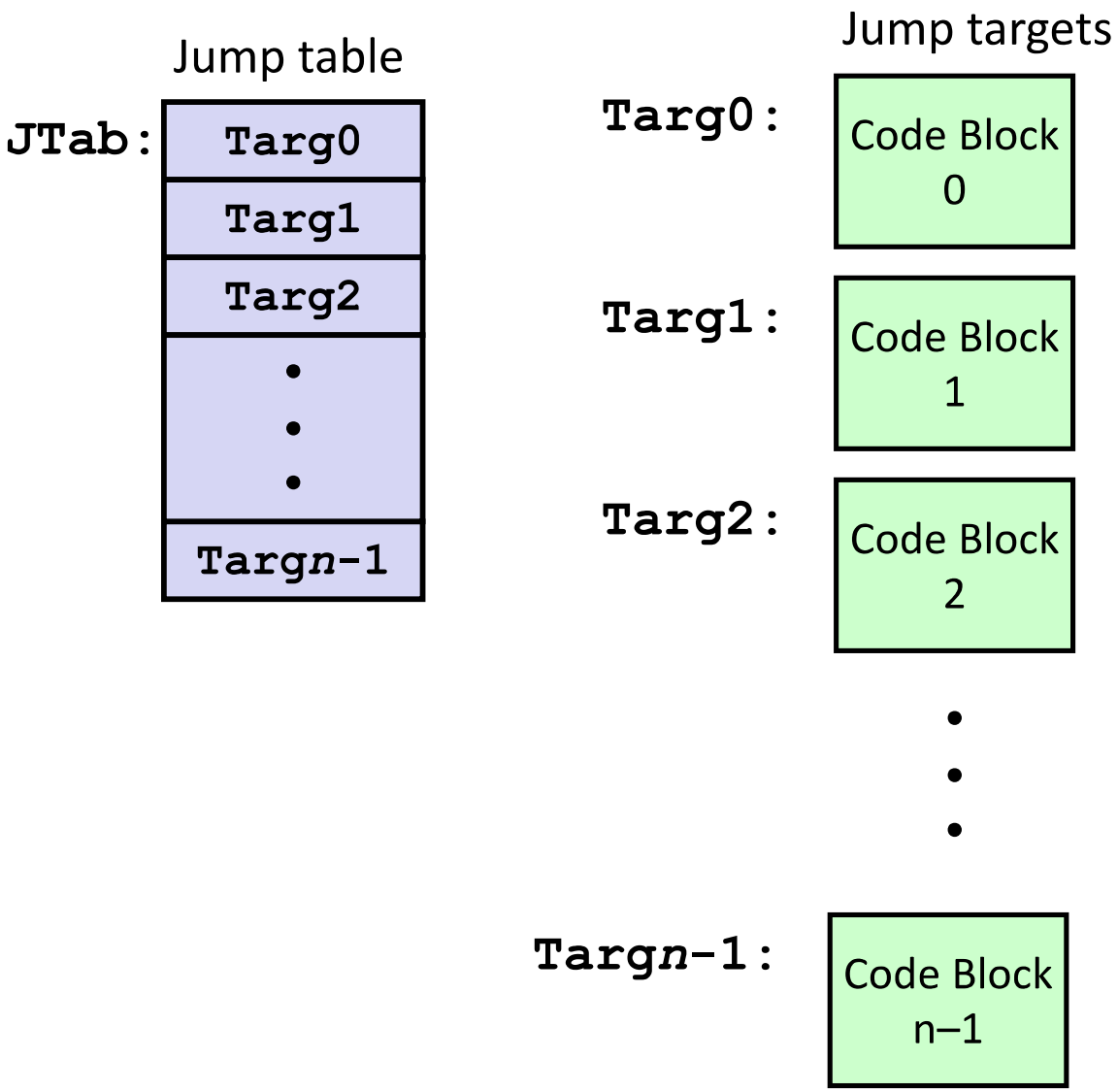
Jump table structure

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;
```



Switch statement example

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

Setup:

```
switch_eg:
  pushq %rbp
  movq  %rsp,%rbp
  movl  %rdi,%rcx
  cmpl  $6,%rdi          # Compare x:6
  ja    .L8              # if > goto default
  jmp   *.L4(,%rdi,8)    # goto *JTab[x]
```

What range of values takes *default*?

Register	Use
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	Return value

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
.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:
  pushq %rbp
  movq  %rsp,%rbp
  movl  %rdi,%rcx
  cmpl  $6,%rdi          # Compare x:6
  ja    .L8              # if > goto default
  jmp   *.L4(,%rdi,8)    # goto *JTab[x]
```

← Indirect jump

Setup explanation

- **Table structure**
  - Each target requires 8 bytes
  - Base address at **`.L4`**

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
```

- **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 on x86-64)
  - Fetch target from effective address **`.L4 + x*8`**
    - Only for $0 \le \mathbf{x} \le 6$

Jump table

```
.section .rodata
  .align 8
.L7:
  .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;
}
```

- **Duplicates have same label**
- **Missing cases use label for the default case**

Code blocks (x == 1, default)

```
 switch(x) {
 case 1:        /* .L3 */
        w = y*z;
        break;
  ...
 default:       /* .L2 */
        w = 2;
}
```

```
.L3:
   movq  %rsi,%rax        # y
   imulq %rdx,%rax        # w = y*z
   jmp   .L10             # Goto done

.L8:
   movq $2,%rax           # w = 2
   jmp .L10               # Goto done
```

- **Jump table avoids sequencing through cases**
  - Constant time, rather than linear

Code blocks (x == 2, x == 3)

```
    int w = 1;
       . . .
    switch(x) {
     . . .
    case 2:       /* .L5 */
        w = y/z;
        /* Fall Through */

    case 3:       /* .L9 */
        w += z;
        break;
     . . .
    }
```

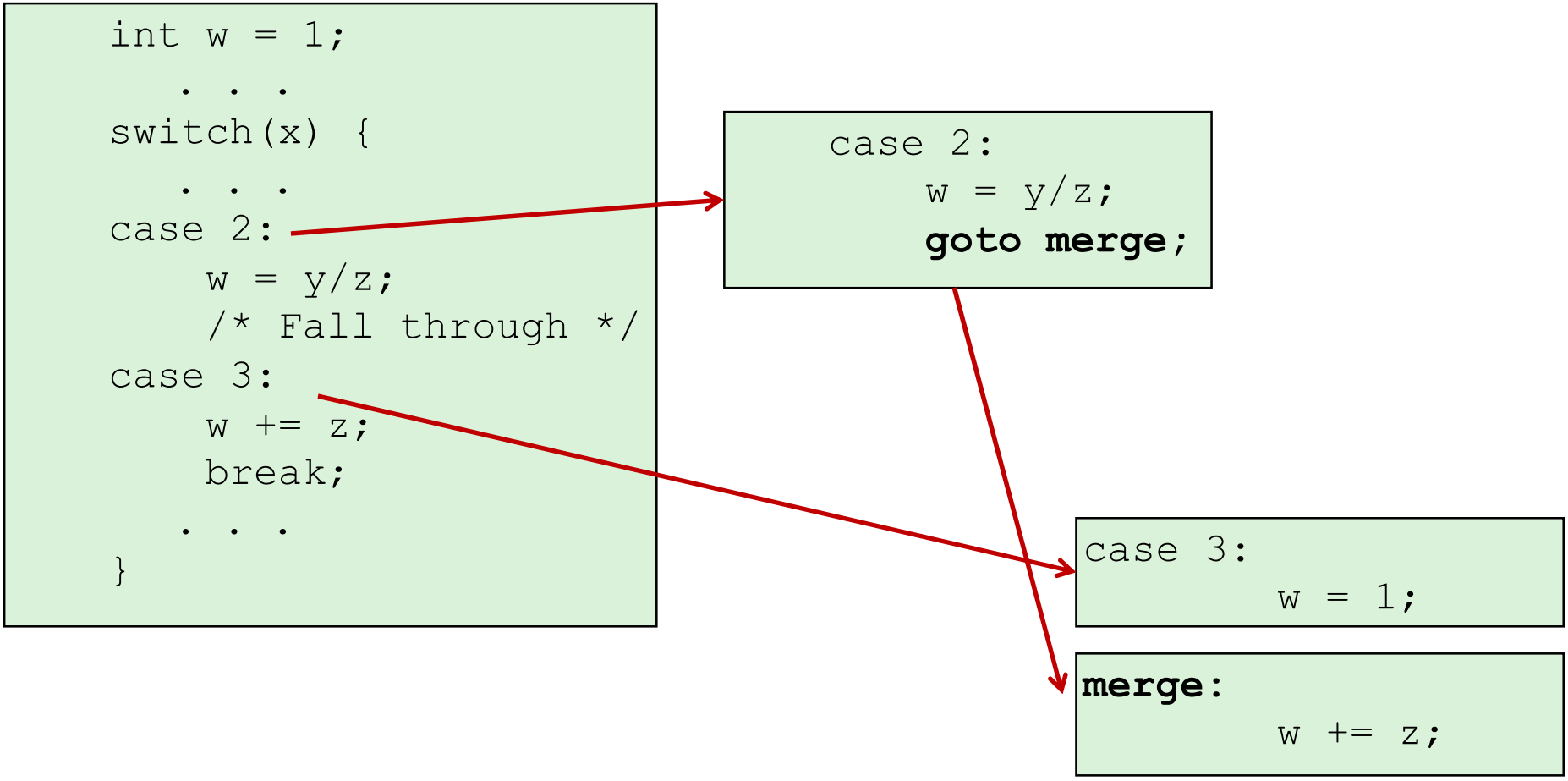
```
.L4:
    movq   %rsi,%rax        # y
    cqto
    idivq  %rcx             # w = y/z
    jmp    .L6

.L9:
    movq   $1, %rax         # w = 1

.L6:
    addq   %rcx,%rax        #  w += z
    jmp    .L10             #  goto done
```

- **Do not initialize `w = 1` unless really need it**
- **Use program sequencing to handle fall-through**

Handling fall-through



Code blocks (x == 5, x == 6)

```
switch(x) {
  . . .
  case 5:     /* .L7 */
  case 6:     /* .L7 */
      w -= z;
      break;
  . . .
}

return w;
```

```
.L7:
  movq  $1,%rax          # w = 1
  subq  %rdx,%rax        # w -= z

.L10:                    # done
  movq %rbp,%rsp
  popl %rbp
  ret
```

- **Use jump table to handle holes and duplicate tags**

Summary

- **C control**
  - if-then-else
  - do-while
  - while, for
  - switch
- **Assembler control**
  - Conditional jump
  - Conditional move
  - Indirect jump (via jump tables)
- **Standard techniques**
  - Loops converted to do-while form
  - Large switch statements use jump tables

Disassemble - Finding jump table

```
 00000000004005e0 <switch_eg>:
4005e0:       48 89 d1                mov    %rdx,%rcx
4005e3:       48 83 ff 06             cmp    $0x6,%rdi
4005e7:       77 2b                   ja     400614 <switch_eg+0x34>
4005e9:       ff 24 fd f0 07 40 00    jmpq   *0x4007f0(,%rdi,8)
4005f0:       48 89 f0                mov    %rsi,%rax
4005f3:       48 0f af c2             imul   %rdx,%rax
4005f7:       c3                      retq
4005f8:       48 89 f0                mov    %rsi,%rax
4005fb:       48 99                   cqto
4005fd:       48 f7 f9                idiv   %rcx
400600:       eb 05                   jmp    400607 <switch_eg+0x27>
400602:       b8 01 00 00 00          mov    $0x1,%eax
400607:       48 01 c8                add    %rcx,%rax
40060a:       c3                      retq
40060b:       b8 01 00 00 00          mov    $0x1,%eax
400610:       48 29 d0                sub    %rdx,%rax
400613:       c3                      retq
400614:       b8 02 00 00 00          mov    $0x2,%eax
400619:       c3                      retq
```

Disassemble - Finding jump table (cont.)

```
 00000000004005e0 <switch_eg>:
. . .
4005e9:       ff 24 fd f0 07 40 00    jmpq   *0x4007f0(,%rdi,8)
. . .
```

```
 % gdb switch
(gdb) x /8xg 0x4007f0
0x4007f0:       0x0000000000400614      0x00000000004005f0
0x400800:       0x00000000004005f8      0x0000000000400602
0x400810:       0x0000000000400614      0x000000000040060b
0x400820:       0x000000000040060b      0x2c646c25203d2078
(gdb)
```

Disassemble - Finding jump table (cont.)

```
 % gdb switch
(gdb) x /8xg 0x4007f0
0x4007f0:       0x0000000000400614      0x00000000004005f0
0x400800:       0x00000000004005f8      0x0000000000400602
0x400810:       0x0000000000400614      0x000000000040060b
0x400820:       0x000000000040060b      0x2c646c25203d2078
```

```
  . . .
  4005f0:       48 89 f0                mov    %rsi,%rax
  4005f3:       48 0f af c2             imul   %rdx,%rax
  4005f7:       c3                      retq
  4005f8:       48 89 f0                mov    %rsi,%rax
  4005fb:       48 99                   cqto
  4005fd:       48 f7 f9                idiv   %rcx
  400600:       eb 05                   jmp    400607 <switch_eg+0x27>
  400602:       b8 01 00 00 00          mov    $0x1,%eax
  400607:       48 01 c8                add    %rcx,%rax
  40060a:       c3                      retq
  40060b:       b8 01 00 00 00          mov    $0x1,%eax
  400610:       48 29 d0                sub    %rdx,%rax
  400613:       c3                      retq
  400614:       b8 02 00 00 00          mov    $0x2,%eax
  400619:       c3                      retq
```