CS 33

Machine Programming (2)
### Processor State (x86-64, Partial)

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
</tr>
<tr>
<td>%rip</td>
<td></td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

**Condition codes**

- 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: \texttt{addl/addq Src, Dest} $\leftrightarrow t = a+b$
  - \textbf{CF set} if carry out from most significant bit (unsigned overflow)
  - \textbf{ZF set} if $t == 0$
  - \textbf{SF set} if $t < 0$ (as signed)
  - \textbf{OF set} if two’s-complement (signed) overflow
    - $(a>0 \ & \ & b>0 \ & \ & t<0) \ \lor \ (a<0 \ & \ & b<0 \ & \ & t>=0)$

- \textbf{Not set by lea instruction}
Condition Codes (Explicit Setting: Compare)

- Explicit setting by compare instruction
  
  `cmp1/cmpq src2, src1`
  
  `cmp1 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
  
  \texttt{testl/testq \ src2, src1}
  
  \texttt{testl b,a} like computing \( a \& b \) without setting destination

  - sets condition codes based on value of \( \text{Src1} \) & \( \text{Src2} \)
  - useful to have one of the operands be a mask

\[ \begin{align*}
  \text{ZF \ set \ when \ } a \& b \ &= \ 0 \\
  \text{SF \ set \ when \ } a \& b \ &= \ < \ 0
\end{align*} \]
Reading Condition Codes

- **SetX instructions**
  - set single byte based on combinations of condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
## 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

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

### Body

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

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%ah</th>
<th>%al</th>
</tr>
</thead>
</table>

```plaintext
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

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

### Body

<table>
<thead>
<tr>
<th>cmpl %esi, %edi</th>
<th>setg %al</th>
<th>movzbl %al, %eax</th>
</tr>
</thead>
<tbody>
<tr>
<td># compare x : y</td>
<td># %al = x &gt; y</td>
<td># zero rest of %eax/%rax</td>
</tr>
</tbody>
</table>
Jumping

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

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF)&amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>ja</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Jumping

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

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF) &amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>ja</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

Quiz 1

What would be an appropriate description if the condition is ~CF?

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

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

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

x in %edi
y in %esi
Conditional-Branch Example (Cont.)

```c
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;
}
```

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

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

Body1

Body2a

Body2b
General Conditional-Expression Translation

C Code

```c
val = Test ? Then_Expr : Else_Expr;
val = x>y ? x-y : y-x;
```

Goto Version

```c
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 ← 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

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

Goto Version

```c
tval = Then_Expr;
result = Else_Expr;
t = Test;
if (t) result = tval;
return result;
```
Conditional Move Example: x86-64

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

absdiff:

- x in %edi
- y in %esi

```
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

\[
\text{val} = \text{Test}(x) \ ? \ \text{Hard1}(x) : \text{Hard2}(x);
\]

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

Risky Computations

\[
\text{val} = p \ ? \ *p : 0;
\]

- both values get computed
- may have undesirable effects

Computations with side effects

\[
\text{val} = x > 0 \ ? \ x*=7 : x+=3;
\]

- both values get computed
- must be side-effect free
“Do-While” Loop Example

C Code

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

Goto Version

```c
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

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

```
movl $0, %eax      # result = 0
.L2:  
    movl %edi, %ecx  # t = x & 1
    addl %ecx, %eax  # result += t
    shrl %edi        # x >>= 1
    jne .L2          # if !0, goto loop
```

Registers:
- %edi  x
- %eax  result
General “Do-While” Translation

C Code

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

- **Body:**
  
  ```
  { 
    Statement_1;
    Statement_2;
    ...
    Statement_n;
  }
  ```

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

Goto Version

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

Copyright © 2015 Thomas W. Doeppner. All rights reserved.
“While” Loop Example

C Code

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

Goto Version

```c
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:
  done:
“For” Loop Example

C Code

```c
#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

\[
\text{for} \ (\text{Init}; \ \text{Test}; \ \text{Update}) \\
\text{Body}
\]

\[
\text{for} \ (i = 0; \ i < \ \text{WSIZE}; \ i++) \ {
    \begin{array}{l}
    \text{unsigned} \ mask = 1 << i; \\
    \text{result} += (x & mask) \neq 0;
    \end{array}
\}
\]
“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;
loop:
  Body
  Update
  if (Test)
goto loop;
done:

Init;
if (!Test)
goto done;
done:

while (Test);
"For" Loop Conversion Example

C Code

```c
#include 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

```c
int pcount_for_gt(unsigned x) {
    int i;
    int result = 0;
    i = 0;
    if (!(i < WSIZE)) goto done;
    loop:
    {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    i++; if (i < WSIZE) goto loop;
    done:
    return result;
}
```
Switch-Statement Example

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

```c
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;
}
```
Jump-Table Structure

Switch Form

```java
switch(x) {
    case val_0:
        Block 0
    case val_1:
        Block 1
    ...
    case val_n-1:
        Block n-1
}
```

Jump Table

<table>
<thead>
<tr>
<th>Case</th>
<th>Jump Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>val_0</td>
<td>Targ0</td>
</tr>
<tr>
<td>val_1</td>
<td>Targ1</td>
</tr>
<tr>
<td>val_n-1</td>
<td>Targn-1</td>
</tr>
</tbody>
</table>

Jump Targets

- Targ0: Code Block 0
- Targ1: Code Block 1
- Targ2: Code Block 2
- Targn-1: Code Block n-1

Approximate Translation

```java
target = JTab[x];
goto *target;
```
Switch-Statement Example (x86-64)

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

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]
```

What range of values is covered by the default case?

Note that w not initialized here
Switch-Statement Example

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

**Jump table**

```plaintext
.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:**

```plaintext
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]
```
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 ≤ x ≤ 6
Jump Table

Jump table

```
.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)

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

```assembly
.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

```c
long w = 1;
    ...
switch(x) {
    ...
    case 2:
        w = y/z;
        /* Fall Through */
        case 3:
            w += z;
            break;
    ...
}
```
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
  divq %rcx    # w = y/z
  jmp .L5
.L9:      # x == 3
  movl $1, %eax # w = 1
.L5:      # merge:
  addq %rcx, %rax # w += z
  ret
The diagram illustrates the operation of the `idivq` instruction in a computer system. It involves the following registers:

- **rdx**: Dividend register
- **rax**: Quotient register
- **rcx**: Divisor register
- **Quotient**
- **Remainder**

The diagram shows the flow of data from the dividend to the quotient and remainder through the `idivq` instruction. The values 127 and 0 are involved in the process, representing the dividend and the quotient, respectively.
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

```
0000000000004004ac <switch_eg>:
    ...
00004004b3: 77 30     ja   4004e5 <switch_eg+0x39>
00004004b5: 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
```

```
0x4005c0:      0x000000000004004e5      0x00000000004004bc
0x4005d0:      0x000000000004004c4      0x00000000004004d3
0x4005e0:      0x000000000004004e5      0x00000000004004dc
0x4005f0:      0x000000000004004dc
```
x86-64 Object Code (cont.)

• Deciphering jump table

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4005c0</td>
<td>0x00000000004004e5</td>
<td>0</td>
</tr>
<tr>
<td>0x4005d0</td>
<td>0x00000000004004c4</td>
<td>1</td>
</tr>
<tr>
<td>0x4005e0</td>
<td>0x00000000004004e5</td>
<td>2</td>
</tr>
<tr>
<td>0x4005f0</td>
<td>0x00000000004004dc</td>
<td>3</td>
</tr>
<tr>
<td>0x4005c8</td>
<td>0x04004bc</td>
<td>4</td>
</tr>
<tr>
<td>0x4005d8</td>
<td>0x04004d3</td>
<td>5</td>
</tr>
<tr>
<td>0x4005e8</td>
<td>0x04004dc</td>
<td>6</td>
</tr>
</tbody>
</table>
Disassembled Targets

(gdb) disassemble 0x4004bc,0x4004eb
Dump of assembler code from 0x4004bc to 0x4004eb

0x0000000000000000 <switch_eg+16>:
  mov  %rsi,%rax

0x0000000000000000 <switch_eg+19>:
  imul %rdx,%rax

0x0000000000000000 <switch_eg+23>:
  retq

0x0000000000000000 <switch_eg+24>:
  mov  %rsi,%rax

0x0000000000000000 <switch_eg+27>:
  mov  %rsi,%rdx

0x0000000000000000 <switch_eg+30>:
  sar  $0x3f,%rdx

0x0000000000000000 <switch_eg+34>:
  idiv %rcx

0x0000000000000000 <switch_eg+37>:
  jmp  0x4004d8 <switch_eg+44>

0x0000000000000000 <switch_eg+39>:
  mov  $0x1,%eax

0x0000000000000000 <switch_eg+44>:
  add  %rcx,%rax

0x0000000000000000 <switch_eg+47>:
  retq

0x0000000000000000 <switch_eg+48>:
  mov  $0x1,%eax

0x0000000000000000 <switch_eg+53>:
  sub  %rdx,%rax

0x0000000000000000 <switch_eg+56>:
  retq

0x0000000000000000 <switch_eg+57>:
  mov  $0x2,%eax

0x0000000000000000 <switch_eg+62>:
  retq
## Matching Disassembled Targets

<table>
<thead>
<tr>
<th>Value</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4004e5</td>
<td>0</td>
</tr>
<tr>
<td>0x4004bc</td>
<td>1</td>
</tr>
<tr>
<td>0x4004c4</td>
<td>2</td>
</tr>
<tr>
<td>0x4004d3</td>
<td>3</td>
</tr>
<tr>
<td>0x4004e5</td>
<td>4</td>
</tr>
<tr>
<td>0x4004dc</td>
<td>5</td>
</tr>
<tr>
<td>0x4004dc</td>
<td>6</td>
</tr>
</tbody>
</table>

- \(0x00000000004004bc\): 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
Quiz 2

What C code would you compile to get the following assembler code?

```
movl $0, %eax
.L2:
    movl %eax, a(,%rax,4)
    addq $1, %rax
    cmpq $10, %rax
    jne .L2
ret
```

```
int a[10];
void func() {
    int i;
    for (i=0; i<10; i++)
        a[i] = i;
}
```

```
int a[10];
void func() {
    int i;
    while (i<10)
        a[i] = i++;
}
```

```
int a[10];
void func() {
    int i;
    switch (i) {
        case 0:
            a[i] = 0;
            break;
        default:
            a[i] = 10
    }
}
```

a  b  c