Most of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.
Supplied by CMU, but converted to x86-64.

```c
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

- Instructions in different order from C code
- Some expressions might require multiple instructions
- Some instructions might cover multiple expressions

```
leal (%rdi,%rsi), %eax    # eax = x+y     (t1)
addl %edx, %eax           # eax = t1+z     (t2)
leal (%rsi,%rsi,2), %edx  # edx = 3*y      (t4)
sall $4, %edx             # edx = t4*16    (t4)
leal 4(%rdi,%rdx), %ecx   # ecx = x+4+t4   (t5)
imull %ecx, %eax          # eax *= t5     (rval)
ret
```
Another Example

```c
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

$2^{13} = 8192$, $2^{13} - 7 = 8185$

```asm
xorl %esi, %edi      # edi = x^y       (t1)
sarl $17, %edi       # edi = t1>>17   (t2)
movl %edi, %eax     # eax = edi
andl $8185, %eax    # eax = t2 & mask (rval)
```

Supplied by CMU, but converted to x86-64.
Quiz 1

- What is the final value in %ecx?

  xorl %ecx, %ecx  
  inc1 %ecx  
  sal1 %cl, %ecx  # %cl is the low byte of %ecx  
  addl %ecx, %ecx  

  a)  2  
  b)  4  
  c)  8  
  d)  indeterminate
### Processor State (x86-64, Partial)

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

**Condition Codes:**
- CF (Carry Flag)
- ZF (Zero Flag)
- SF (Sign Flag)
- OF (Overflow Flag)
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 ← t = a+b
  
  - CF set if carry out from most significant bit or borrow (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 lea instruction**
Condition Codes (Explicit Setting: Compare)

- Explicit setting by compare instruction
  
  \[ \text{cmp} l/\text{cmp} q \quad \text{src} 2, \text{src} 1 \]
  \[ \text{compares src} 1:\text{src} 2 \]
  \[ \text{cmp} b, a \text{ like computing } a-b \text{ without setting destination} \]

  **CF set** if carry out from most significant bit or borrow (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
  \[
  (a>0 \&\& b<0 \&\& (a-b)<0) \text{ } || \text{ } (a<0 \&\& b>0 \&\& (a-b)>0)
  \]
Condition Codes (Explicit Setting: Test)

- Explicit setting by test instruction
  \[
  \text{test}\ l/\text{test}q\ \text{src}2,\ \text{src}1
  \quad\text{test}\ l\ \text{b},\ \text{a like computing}\ a\&b\ \text{without setting destination}
  \]
  
  - sets condition codes based on value of Src1 & Src2
  - useful to have one of the operands be a mask

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

Supplied by CMU.
Recall that the first argument to a function is passed in %rdi (%edi) and the second in %rsi (%esi).
### 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>Less or Equal (Signed)</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>

Supplied by CMU.
Jumping

• jX instructions:
  – Jump to different locations

Quiz 2

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

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

<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>
Conditional-Branch Example

```c
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, %edi
    movl %edi, %eax
    jmp  .L7
    .L6:
    subl %edi, %eax
    .L7:
    ret
```

x in %edi
y in %esi

Supplied by CMU, but converted to x86-64.
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
```

Supplied by CMU, but converted to x86-64.
General Conditional-Expression Translation

C Code

```c
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

Supplied by CMU.
Conditional Moves

- Conditional move instructions
  - instruction supports:
    
    ```
    if (Test) Dest ← Src
    ```
- 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

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

Supplied by CMU.

What’s important here is that the goto version is different from the original C version. In the original, first Test is computed. If it’s true, then Then_Expr is evaluated, otherwise Else_Expr is evaluated.

But in the Goto version both Then_Expr and Else_Expr are computed before the test is performed.
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;
}
```

```
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
```

Supplied by CMU.
Bad Cases for Conditional Move

Expensive Computations

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

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

Risky Computations

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

- both values get computed
- may have undesirable effects

Computations with side effects

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

- both values get computed
- must be side-effect free

Supplied by CMU.
“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

Supplied by CMU.
“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:** # loop:
- **movl %edi, %ecx**
- **andl $1, %ecx** # t = x & 1
- **addl %ecx, %eax** # result += t
- **shrl %edi** # x >>= 1
- **jne .L2** # if !0, goto loop

Supplied by CMU.

Note that the condition codes are set as part of the execution of the shrl instruction.
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

Supplied by CMU.
“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:
```
“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?

Supplied by CMU.
"For" Loop Form

General Form

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

- **Init**
  - `i = 0`

- **Test**
  - `i < WSIZE`

- **Update**
  - `i++`

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

---

Supplied by CMU.

“For” Loop $\Rightarrow$ 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:
```

Supplied by CMU.
“For” Loop Conversion 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;
}
```

Goto Version

```c
int pcount_for_gt(unsigned x) {
    int i;
    int result = 0;
    i = 0;
    if (!(i < WSIZE)) !Test
        goto done;
    loop:
        Body
        if (i < WSIZE) Test
            goto loop;
        else
            return result;
    done:
}
```

Initial test can be optimized away

Supplied by CMU.
long switch_eq
    (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;
}

Switch-Statement Example

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

Supplied by CMU.
The translation is “approximate” because C doesn’t have the notion of the target of a goto being a variable. But, if it did, then the translation is what we’d want!
Supplied by CMU, but converted to x86-64.

Note that the \textit{ja} in the slide causes a jump to occur if the previous comparison is interpreted as being performed on unsigned values, and the result is that \textit{x} is greater than (above) 6. Given that \textit{x} is declared to be a \textit{signed} value, for what range of values of \textit{x} will \textit{ja} cause a jump to take place?

The assembler code shown in the examples was produced by compiling the C code using gcc with the “-O1” flag, which instructs gcc to do some minor optimizations of the code.
Supplied by CMU, but converted to x86-64.
The `jmp` instruction is doing a couple things that require explanation: The asterisk means it’s an *indirect jump* (such indirection is allowed only in jumps). The address specified after the asterisk is the address of an entry in the *jump table*. The asterisk means, rather than jumping directly to that entry, jump to the address that’s in that table entry. “.L7” is a label that’s being used as a displacement in the address computation. The value of .L7 is the address of the area of memory it labels. In this case, it’s the address of the jump table. Thus, an unconditional jump is to take place to the address contained in the 8-byte entry of the jump table indexed by the contents of `%rdi`. Thus, if `%rdi` is, say, 2, then a jump will take place to address in the location starting 16 bytes beyond the beginning of the table. This will be a jump to .L4. .L4 itself is a label of code specified elsewhere, the reference to the label is replaced by the assembler with the address of the code labelled with .L4.

The jump table is separate from the code (it’s not executable). This is specified by the “.section” directive, which also specifies that it should be placed in memory that’s made read-only (“.rodata” indicates this). The “.align 4” says that the address of the start of the table should be divisible by four (why this is important is something we’ll get to in a week or two).
Supplied by CMU, but converted to x86-64.
Supplied by CMU, but converted to x86-64.
Handling Fall-Through

```c
long w = 1;
    ...
switch(x) {
    ...
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    ...
}
```

Supplied by CMU, but converted to x86-64.
The code following the `.L4` label requires some explanation. The `idivq` instruction is special in that it takes a 128-bit dividend that is implicitly assumed to reside in registers `rdx` and `rax`. Its single operand specifies the divisor. The quotient is always placed in the `rax` register, and the remainder in the `rdx` register. In our example, `y`, which we want to be the dividend, is copied into both the `rax` and `rdx` registers. The `sarq` (shift arithmetic right quadword) instruction propagates the sign bit of `rdx` across the entire register, replacing its original contents. Thus, if one considers `rdx` to contain the most-significant bits of the dividend and `rax` to contain the least-significant bits, the pair of registers now contains the 128-bit version of `y`. The `idivq` instruction computes the quotient from dividing this 128-bit value by the 64-bit value contained in register `rcx` (containing `z`). The quotient is stored register `rax` (implicitly) and the remainder is stored in register `rdx` (and is ignored in our example). This illustrated in the next slide.
idivq

127 64 63 0

rdx rax

dividend

idivq

63 63 0

rcx

divisor

63 0 63

rdx rax

remainder quotient
Disassembly was accomplished using “objdump –d”. Note that the text enclosed in angle brackets (“<“, “>”) is essentially a comment, relating the address (4004e5) to a symbolic location (0x39 bytes after the beginning of switch_eg).
Supplied by CMU, but converted to x86-64. We assume that the switch_eg function was
included in a program whose name is `switch`. Hence, `gdb` is invoked from the shell with
the argument “switch”.

```plaintext
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: 0x00000000004004e5 0x00000000004004bc
    0x4005d0: 0x00000000004004c4 0x00000000004004d3
    0x4005e0: 0x00000000004004e5 0x00000000004004dc
    0x4005f0: 0x00000000004004dc
```
**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>0x4004e5</td>
<td>0</td>
</tr>
<tr>
<td>0x4005c8</td>
<td>0x4004bc</td>
<td>1</td>
</tr>
<tr>
<td>0x4005d0</td>
<td>0x4004c4</td>
<td>2</td>
</tr>
<tr>
<td>0x4005d8</td>
<td>0x4004d3</td>
<td>3</td>
</tr>
<tr>
<td>0x4005e0</td>
<td>0x4004e5</td>
<td>4</td>
</tr>
<tr>
<td>0x4005e8</td>
<td>0x4004dc</td>
<td>5</td>
</tr>
<tr>
<td>0x4005f0</td>
<td>0x4004dc</td>
<td>6</td>
</tr>
</tbody>
</table>

Supplied by CMU, but converted to x86-64.
### Disassembled Targets

```plaintext
(gdb) disassemble 0x4004bc,0x4004eb
Dump of assembler code from 0x4004bc to 0x4004eb
0x0000000000004004bc <switch_eg+16>:  mov    %rsi,%rax
0x0000000000004004bf <switch_eg+19>:  imul   %rdx,%rax
0x0000000000004004c3 <switch_eg+23>:  retq
0x0000000000004004c4 <switch_eg+24>:  mov    %rsi,%rax
0x0000000000004004c7 <switch_eg+27>:  mov    %rsi,%rdx
0x0000000000004004ca <switch_eg+30>:  sar    $0x3f,%rdx
0x0000000000004004ce <switch_eg+34>:  idiv   %rcx
0x0000000000004004d1 <switch_eg+37>:  jmp    0x4004d8 <switch_eg+44>
0x0000000000004004d3 <switch_eg+39>:  mov    $0x1,%eax
0x0000000000004004d8 <switch_eg+44>:  add    %rcx,%rax
0x0000000000004004db <switch_eg+47>:  retq
0x0000000000004004dc <switch_eg+48>:  mov    $0x1,%eax
0x0000000000004004e1 <switch_eg+53>:  sub    %rdx,%rax
0x0000000000004004e4 <switch_eg+56>:  retq
0x0000000000004004e5 <switch_eg+57>:  mov    $0x2,%eax
0x0000000000004004ea <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 3

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
    jl    .L2
    ret
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
</table>
| int a[10];
void func() {
   int i;
   for (i=0; i<10; i++)
      a[i]= 1;
}
| int a[10];
void func() {
   int i=0;
   while (i<10)
      a[i]= i++;
}
| int a[10];
void func() {
   int i=0;
   switch (i) {
      case 0:
         a[i] = 0;
         break;
      default:
         a[i] = 10
    }
} |