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.
### Processor State (x86-64, Partial)

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

<table>
<thead>
<tr>
<th><strong>condition codes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
</tr>
</tbody>
</table>
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: \( addl/addq \) \( \text{Src,Dest} \rightarrow t = a+b \)
  - \( CF \) 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
    \( (a>0 \land \land b>0 \land t<0) \lor (a<0 \land \land b<0 \land t>=0) \)

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

- Explicit setting by compare instruction
  
  \texttt{cmp} \texttt{src1} / \texttt{cmpq} \texttt{src2}, \texttt{src1}
  
  compares \texttt{src1:src2}
  
  \texttt{cmp} \texttt{b,a} like computing \texttt{a-b} without setting destination
  
  \textbf{CF set} if carry out from most significant bit or borrow (used for unsigned comparisons)
  
  \textbf{ZF set} if \texttt{a} == \texttt{b}
  
  \textbf{SF set} if \texttt{(a-b)} < 0 (as signed)
  
  \textbf{OF set} if two's-complement (signed) overflow
  
  \((\texttt{a}>0 \land \texttt{b}<0 \land \texttt{(a-b)}<0) \lor (\texttt{a}<0 \land \texttt{b}>0 \land \texttt{(a-b)}>0)\)
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`

Supplied by CMU.
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.
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
```

Supplied by CMU, but converted to x86-64.
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^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 based on conditions.

### Quiz 1

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
```

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

```
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
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

Supplied by CMU.
Using 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**

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

Supplied by CMU.

Note that, as shown 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;
}
```

```assembly
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
    cmp1 %esi, %edi  # compare x:y
    cmovle %edx, %eax  # if <=, result = tval
ret
```

Supplied by CMU.
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

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;
}
```

```assembly
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
  while (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.
Supplied by CMU.
“For” Loop $\rightarrow$ While Loop

For Version

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

While Version

\[
\text{Init}; \\
\text{while (Test) } \{ \\
\quad \text{Body} \\
\quad \text{Update}; \\
\}
\]
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:

CS33 Intro to Computer Systems

Supplied by CMU.
“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;
    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;
}
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 = z;
    }
    return w;
  }

Switch-Statement Example

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

Supplied by CMU.
Jump-Table Structure

Switch Form

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

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

Approximate Translation

```c
target = JTab[x];
goto *target;
```
Supplied by CMU, but converted to x86-64.

Note that the `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 `x` is greater than (above) 6. Given that `x` is declared to be a `signed` value, for what range of values of `x` will `ja` cause a jump to take place?

Note that the assembler code shown in the examples was produced by compiling the C code using gcc with the “-O1” flag.
Supplied by CMU, but converted to x86-64.
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
```

Supplied by CMU, but converted to x86-64.
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;
}
```
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;
  . .
}
case 2:
  w = y/z;
  goto merge;
```

```c
case 3:
  w = 1;
merge:
  w += z;
```
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.
Supplied by CMU, but converted to x86-64.

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).
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
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4005c0</td>
<td>0x0000000000040045</td>
<td>0x000000000004004bc</td>
</tr>
<tr>
<td>0x4005d0</td>
<td>0x000000000004004c4</td>
<td>0x000000000004004d3</td>
</tr>
<tr>
<td>0x4005e0</td>
<td>0x000000000004004c5</td>
<td>0x000000000004004dc</td>
</tr>
<tr>
<td>0x4005f0</td>
<td>0x000000000004004dc</td>
<td></td>
</tr>
</tbody>
</table>

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”.

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>0x0000000000004004e5</td>
<td></td>
</tr>
<tr>
<td>0x4005d0</td>
<td>0x0000000000004004c4</td>
<td></td>
</tr>
<tr>
<td>0x4005e0</td>
<td>0x0000000000004004e5</td>
<td></td>
</tr>
<tr>
<td>0x4005f0</td>
<td>0x0000000000004004dc</td>
<td></td>
</tr>
</tbody>
</table>

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

(gdb) disassemble 0x4004bc, 0x4004eb
Dump of assembler code from 0x4004bc to 0x4004eb
0x000000000004004bc <switch_eg+16>:   mov   %rsi, %rax
0x000000000004004bf <switch_eg+19>:   imul  %rdx, %rax
0x000000000004004c3 <switch_eg+23>:   retq
0x000000000004004c4 <switch_eg+24>:   mov   %rsi, %rax
0x000000000004004c7 <switch_eg+27>:   mov   %rsi, %rdx
0x000000000004004ca <switch_eg+30>:   sar   $0x3f, %rdx
0x000000000004004ce <switch_eg+34>:   div   %rcx
0x000000000004004d1 <switch_eg+37>:   jmp   0x4004d8 <switch_eg+44>
0x000000000004004d3 <switch_eg+39>:   mov   $0x1, %eax
0x000000000004004d8 <switch_eg+44>:   add   %rcx, %rax
0x000000000004004db <switch_eg+47>:   retq
0x000000000004004dc <switch_eg+48>:   mov   $0x1, %eax
0x000000000004004e1 <switch_eg+53>:   sub   %rdx, %rax
0x000000000004004e4 <switch_eg+56>:   retq
0x000000000004004e5 <switch_eg+57>:   mov   $0x2, %eax
0x000000000004004ea <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>

0x000000000004000b: mov %rsi,%rax
0x000000000004000f: imul %rdx,%rax
0x000000000004003c: retq
0x000000000004004c: mov %rsi,%rax
0x00000000000400c7: mov %rsi,%rdx
0x00000000000400ca: sar $0x3f,%rdx
0x00000000000400ce: idiv %rcx
0x00000000000400d1: jmp 0x4004d8
0x00000000000400d3: mov $0x1,%eax
0x00000000000400d8: add %rcx,%rax
0x00000000000400db: retq
0x00000000000400dc: mov $0x1,%eax
0x00000000000400e1: sub %rdx,%rax
0x00000000000400e4: retq
0x00000000000400e5: mov $0x2,%eax
0x00000000000400ea: retq
Quiz 2

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

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

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

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

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