CS 33

Machine Programming (3)

Not a Quiz …

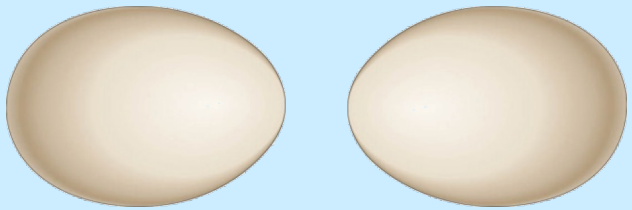
What value ends up in %ecx?

```
movl $1000,%eax
movl $1,%ebx
movl 2(%eax,%ebx,4),%ecx
```

a) 0x02030405
b) 0x05040302
c) 0x06070809
d) 0x09080706

	1009:	0x09
	1008:	0x08
	1007:	0x07
	1006:	0x06
	1005:	0x05
	1004:	0x04
	1003:	0x03
	1002:	0x02
	1001:	0x01
%eax →	1000:	0x00

Hint:



x86-64 General-Purpose Registers

	%rax	%eax	%r8	%r8d	a5
	%rbx	%ebx	%r9	%r9d	a6
a4	%rcx	%ecx	%r10	%r10d	
a3	%rdx	%edx	%r11	%r11d	
a2	%rsi	%esi	%r12	%r12d	
a1	%rdi	%edi	%r13	%r13d	
	%rsp	%esp	%r14	%r14d	
	%rbp	%ebp	%r15	%r15d	

- Extend existing registers to 64 bits. Add 8 new ones.
- No special purpose for %ebp/%rbp

32-bit Instructions on x86-64

- addl 4(%rdx), %eax
  - memory address must be 64 bits
  - operands (in this case) are 32-bit
    - » result goes into %eax
      - lower half of %rax
      - upper half is filled with zeroes

Bytes

- Each register has a byte version
  - e.g., %r10: %r10b
- Needed for byte instructions
  - movb (%rax, %rsi), %r10b
  - sets *only* the low byte in %r10
    - » other seven bytes are unchanged
- Alternatives
  - movzbq (%rax, %rsi), %r10
    - » copies byte to low byte of %r10
    - » zeroes go to higher bytes
  - movsbq (%rax, %rsi), %r10
    - » copies byte to low byte of %r10
    - » sign is extended to all higher bits

32-bit code for swap

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   pushl %ebp
   movl  %esp,%ebp
   pushl %ebx

   movl   8(%ebp), %edx
   movl   12(%ebp), %ecx
   movl   (%edx), %ebx
   movl   (%ecx), %eax
   movl   %eax, (%edx)
   movl   %ebx, (%ecx)

   popl   %ebx
   popl   %ebp
   ret
```

Set Up

Body

Finish

64-bit code for swap

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
                                    } Set Up
    movl    (%rdi), %edx
    movl    (%rsi), %eax            } Body
    movl    %eax, (%rdi)
    movl    %edx, (%rsi)
    ret                             } Finish
```

- **Arguments passed in registers**
  - **first (`xp`) in `%rdi`, second (`yp`) in `%rsi`**
  - **64-bit pointers**
- **No stack operations required**
- **32-bit data**
  - **data held in registers `%eax` and `%edx`**
  - **`movl` operation**

64-bit code for long int swap

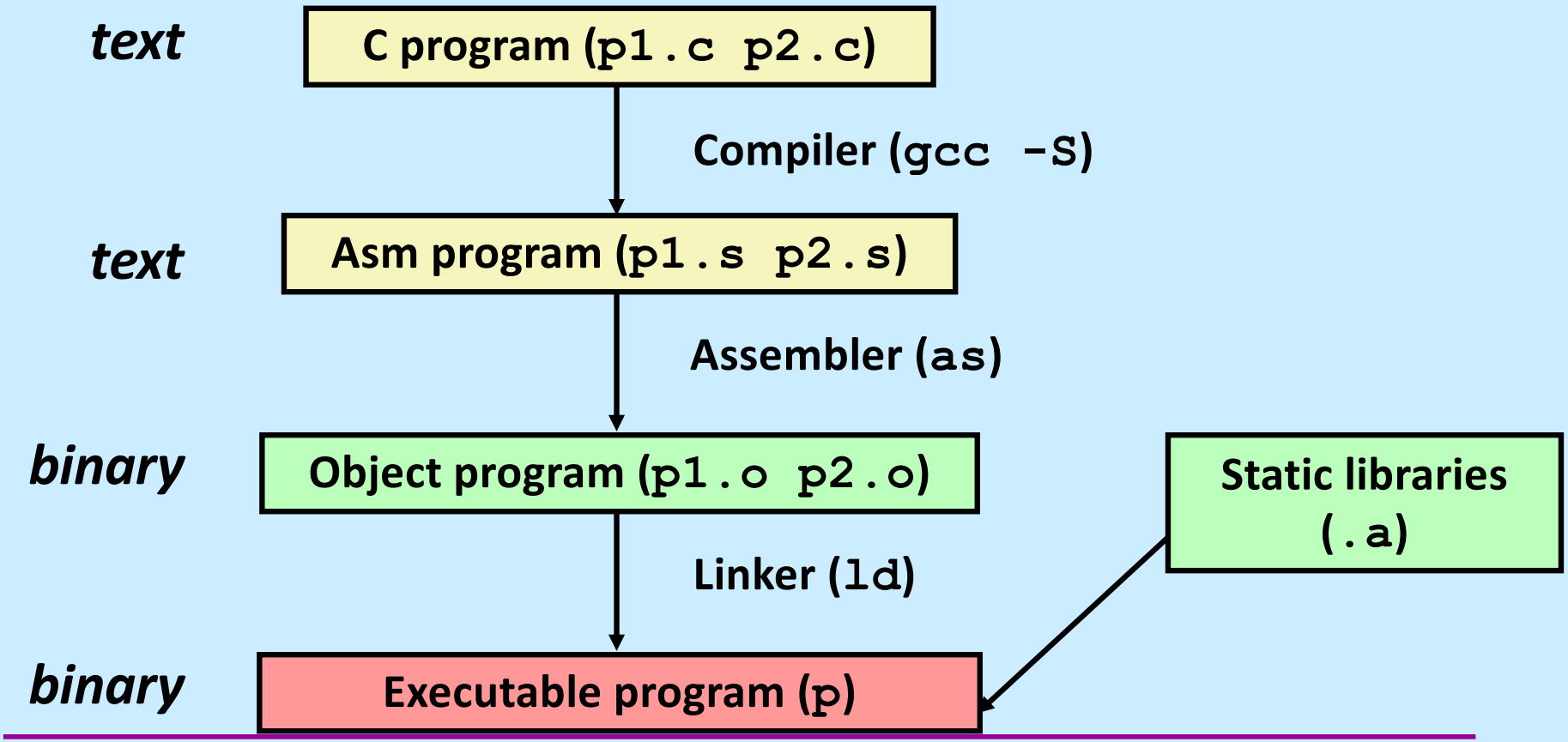
```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap_l:
                                    } Set Up
movq    (%rdi), %rdx
movq    (%rsi), %rax                } Body
movq    %rax, (%rdi)
movq    %rdx, (%rsi)
ret                                 } Finish
```

- **64-bit data**
  - **data held in registers `%rax` and `%rdx`**
  - **`movq` operation**
    - » **"q" stands for quad-word**

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - » use basic optimizations (`-O1`)
  - » put resulting binary in file `p`



Example

```
int sum(int a, int b) {
    return(a+b);
}
```

Object Code

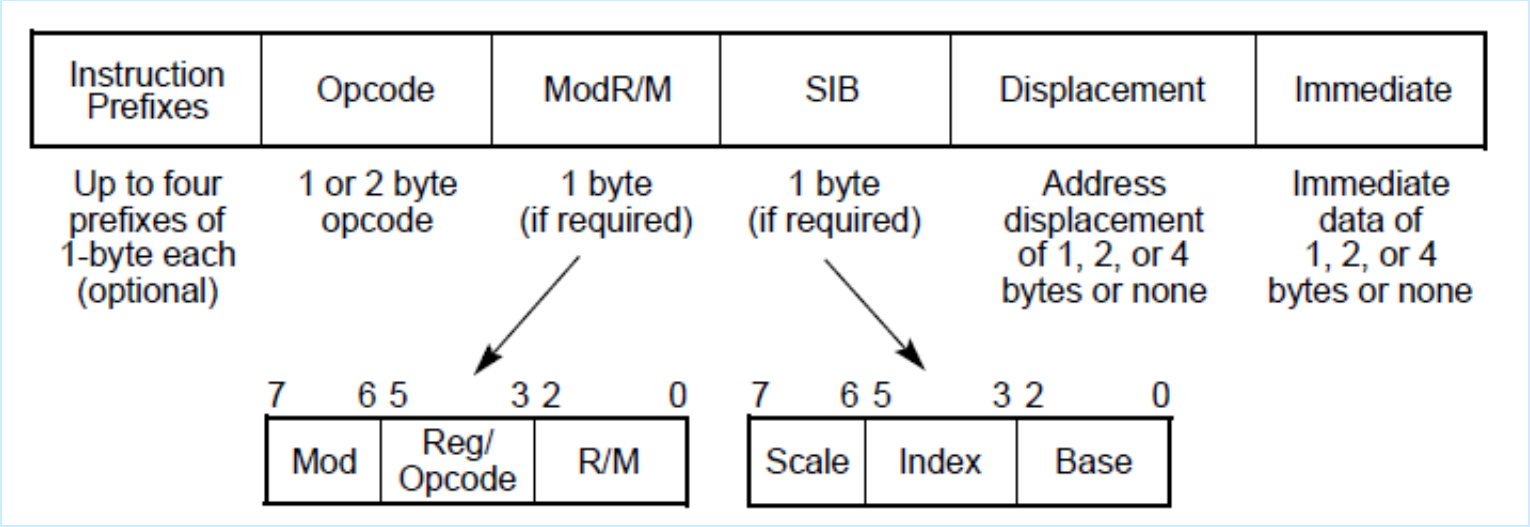
Code for `sum`

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x5d
    0xc3
```

- Total of 11 bytes
- Each instruction: 1, 2, or 3 bytes
- Starts at address `0x401040`

- **Assembler**
  - translates `.s` into `.o`
  - binary encoding of each instruction
  - nearly-complete image of executable code
  - missing linkages between code in different files
- **Linker**
  - resolves references between files
  - combines with static run-time libraries
    - » e.g., code for `printf`
  - some libraries are *dynamically linked*
    - » linking occurs when program begins execution

Instruction Format



Disassembling Object Code

Disassembled

```
080483c4 <sum>:
 80483c4:   55          push   %ebp
 80483c5:   89 e5       mov    %esp,%ebp
 80483c7:   8b 45 0c    mov    0xc(%ebp),%eax
 80483ca:   03 45 08    add    0x8(%ebp),%eax
 80483cd:   5d          pop    %ebp
 80483ce:   c3          ret
```

- **Disassembler**

  `objdump -d <file>`

  – useful tool for examining object code

  – analyzes bit pattern of series of instructions

  – produces approximate rendition of assembly code

  – can be run on either executable or object (`.o`) file

Alternate Disassembly

Object

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x5d
    0xc3
```

Disassembled

```
Dump of assembler code for function sum:
0x080483c4 <sum+0>:     push   %ebp
0x080483c5 <sum+1>:     mov    %esp,%ebp
0x080483c7 <sum+3>:     mov    0xc(%ebp),%eax
0x080483ca <sum+6>:     add    0x8(%ebp),%eax
0x080483cd <sum+9>:     pop    %ebp
0x080483ce <sum+10>:    ret
```

- **Within gdb debugger**
  - `gdb <file>`
  - `disassemble sum`
  - – disassemble procedure
  - `x/11xb sum`
  - – examine the 11 bytes starting at `sum`

How Many Instructions are There?

- We cover ~30
- Implemented by Intel:
  - 80 in original 8086 architecture
  - 7 added with 80186
  - 17 added with 80286
  - 33 added with 386
  - 6 added with 486
  - 6 added with Pentium
  - 1 added with Pentium MMX
  - 4 added with Pentium Pro
  - 8 added with SSE
  - 8 added with SSE2
  - 2 added with SSE3
  - 14 added with x86-64
  - 10 added with VT-x
  - 2 added with SSE4a
- Total: 198
- Doesn't count:
  - floating-point instructions
    - ~100
  - SIMD instructions
    - lots
  - AMD-added instructions
  - undocumented instructions

Some Arithmetic Operations

- **Two-operand instructions:**

Format		Computation	
`addl`	Src,Dest	Dest = Dest + Src	
`subl`	Src,Dest	Dest = Dest – Src	
`imull`	Src,Dest	Dest = Dest * Src	
`sall`	Src,Dest	Dest = Dest << Src	Also called shll
`sarl`	Src,Dest	Dest = Dest >> Src	Arithmetic
`shrl`	Src,Dest	Dest = Dest >> Src	Logical
`xorl`	Src,Dest	Dest = Dest ^ Src	
`andl`	Src,Dest	Dest = Dest & Src	
`orl`	Src,Dest	Dest = Dest | Src	

  - watch out for argument order!
  - no distinction between signed and unsigned int (why?)

Some Arithmetic Operations

- **One-operand Instructions**

`incl`	Dest	= Dest + 1
`decl`	Dest	= Dest – 1
`negl`	Dest	= – Dest
`notl`	Dest	= ~Dest

- **See book for more instructions**

Arithmetic Expression Example

```
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;
}
```

```
arith:
    leal   (%rdi,%rsi), %eax
    addl   %edx, %eax
    leal   (%rsi,%rsi,2), %edx
    sall   $4, %edx
    leal   4(%rdi,%rdx), %ecx
    imull  %ecx, %eax
    ret
```

Understanding `arith`

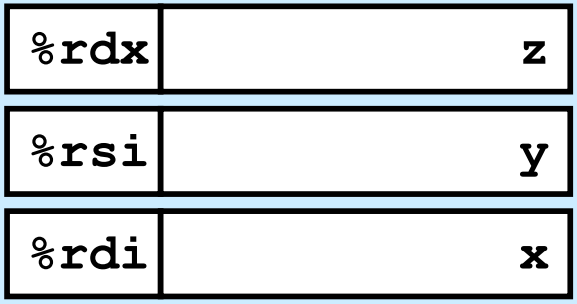
```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

%rdx	z
%rsi	y
%rdi	x

```
leal   (%rdi,%rsi), %eax
addl   %edx, %eax
leal   (%rsi,%rsi,2), %edx
sall   $4, %edx
leal   4(%rdi,%rdx), %ecx
imull  %ecx, %eax
ret
```

Understanding `arith`

```
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;
}
```




```
  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
```

Observations about `arith`

```
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

```
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;
}
```

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

```
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)
```

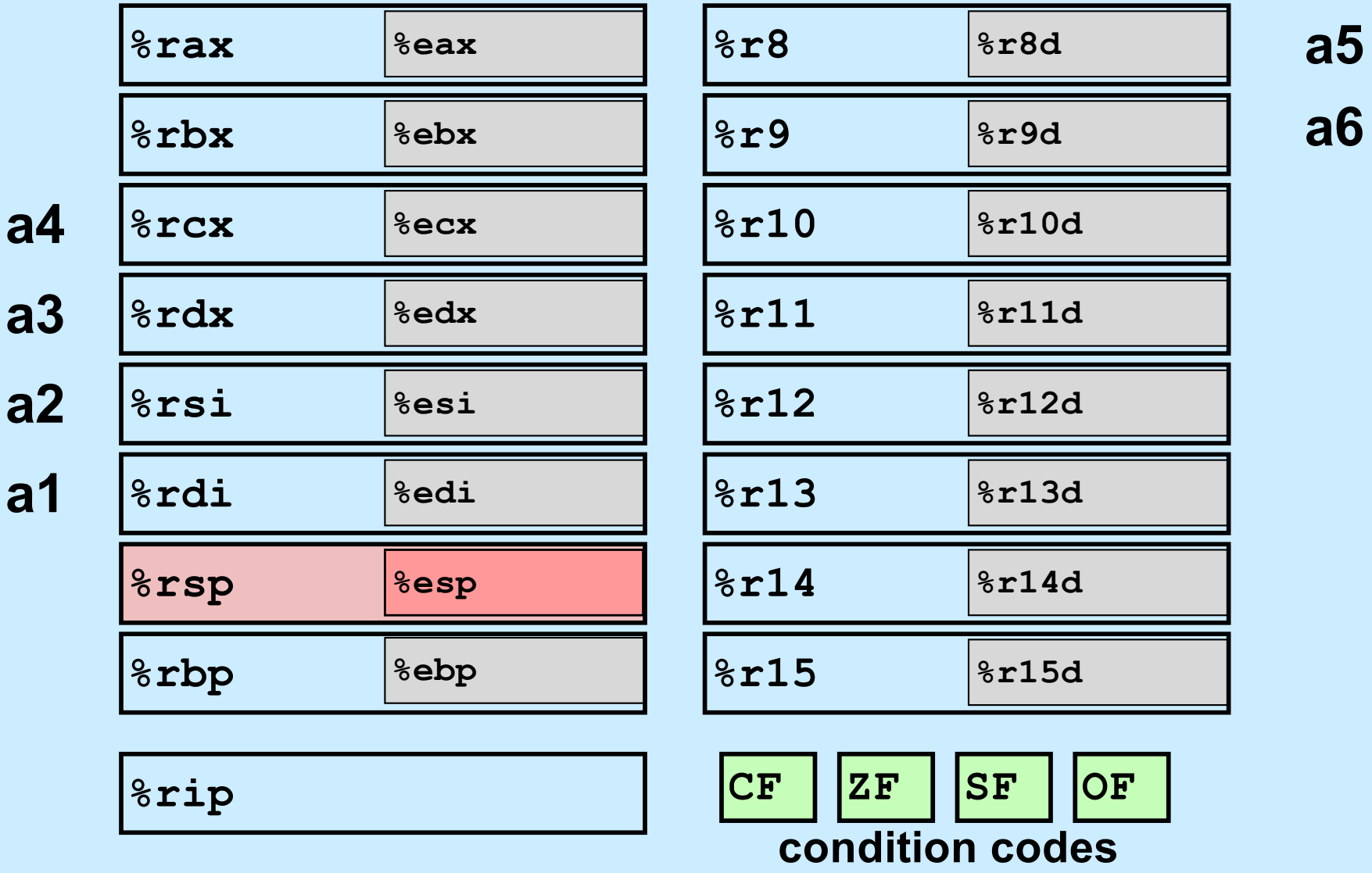
Quiz 1

- **What is the final value in %ecx?**

```
xorl %ecx, %ecx
incl %ecx
sall %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)



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**

  `cmpl/cmpq` src2, src1

  compares src1:src2

  `cmpl b,a` like computing `a-b` 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) || (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`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

Reading Condition Codes (Cont.)

- **SetX instructions:**
  - set single byte based on combination of condition codes
- **Uses 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`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~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, %edi
    movl    %edi, %eax
    jmp     .L7
.L6:
    subl %edi, %eax
.L7:
    ret
```

Body1

Body2a

Body2b

x in %edi

y in %esi

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, %edi
    movl    %edi, %eax
    jmp .L7
.L6:
    subl %edi, %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

"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:

%edi	x
%eax	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
```

General "Do-While" Translation

C Code

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

Goto Version

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

- **Body:**

```
{
   Statement1;
   Statement2;
      …
   Statementn;
}
```

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

"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**

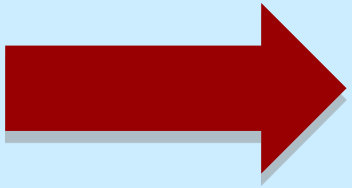
General "While" Translation

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++
```

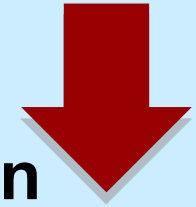
Body

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

"For" Loop → While Loop

For Version

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



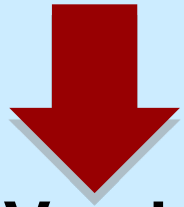
While Version

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

"For" Loop → … → Goto

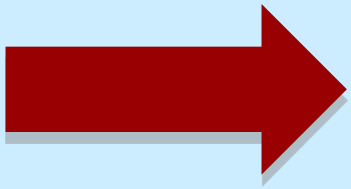
For Version

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

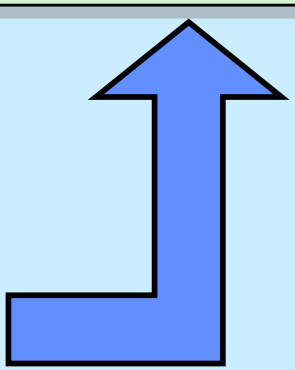


While Version

```
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;
  i = 0;                          Init
  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;
}
```

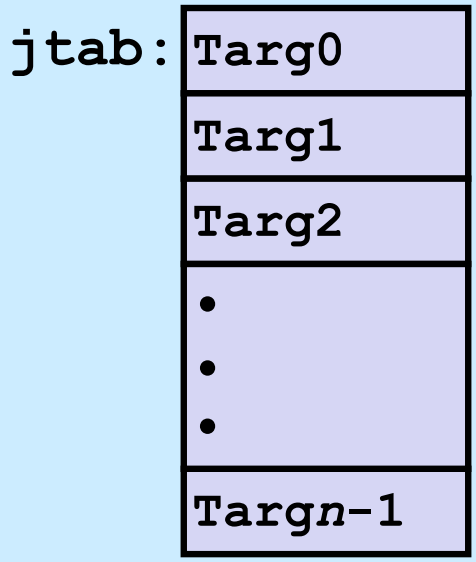
- **Multiple case labels**
  - **here: 5 & 6**
- **Fall-through cases**
  - **here: 2**
- **Missing cases**
  - **here: 4**

Jump-Table Structure

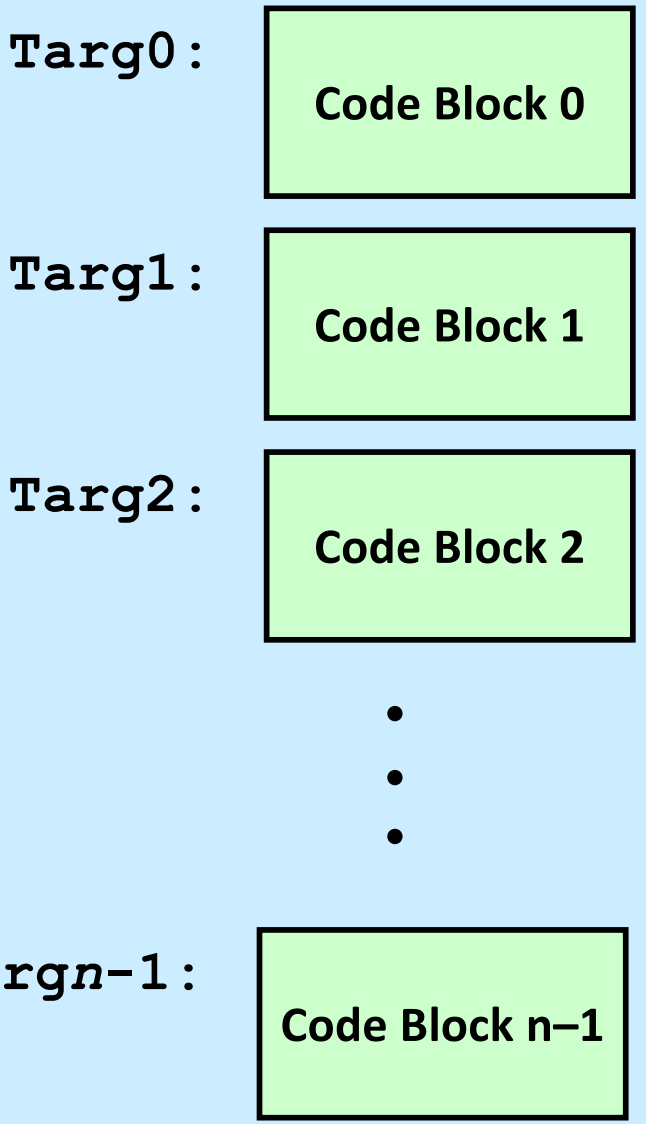
Switch Form

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

Jump Table



Jump Targets



Approximate Translation

```
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;
}
```

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;
}
```

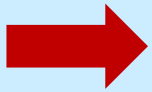
Jump table

```
.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:
    ...     # 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]
```

Indirect jump



Assembly-Setup Explanation

- **Table structure**
  - each target requires 8 bytes
  - base address at `.L7`

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
```

- **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 $0 \le x \le 6$

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;
```

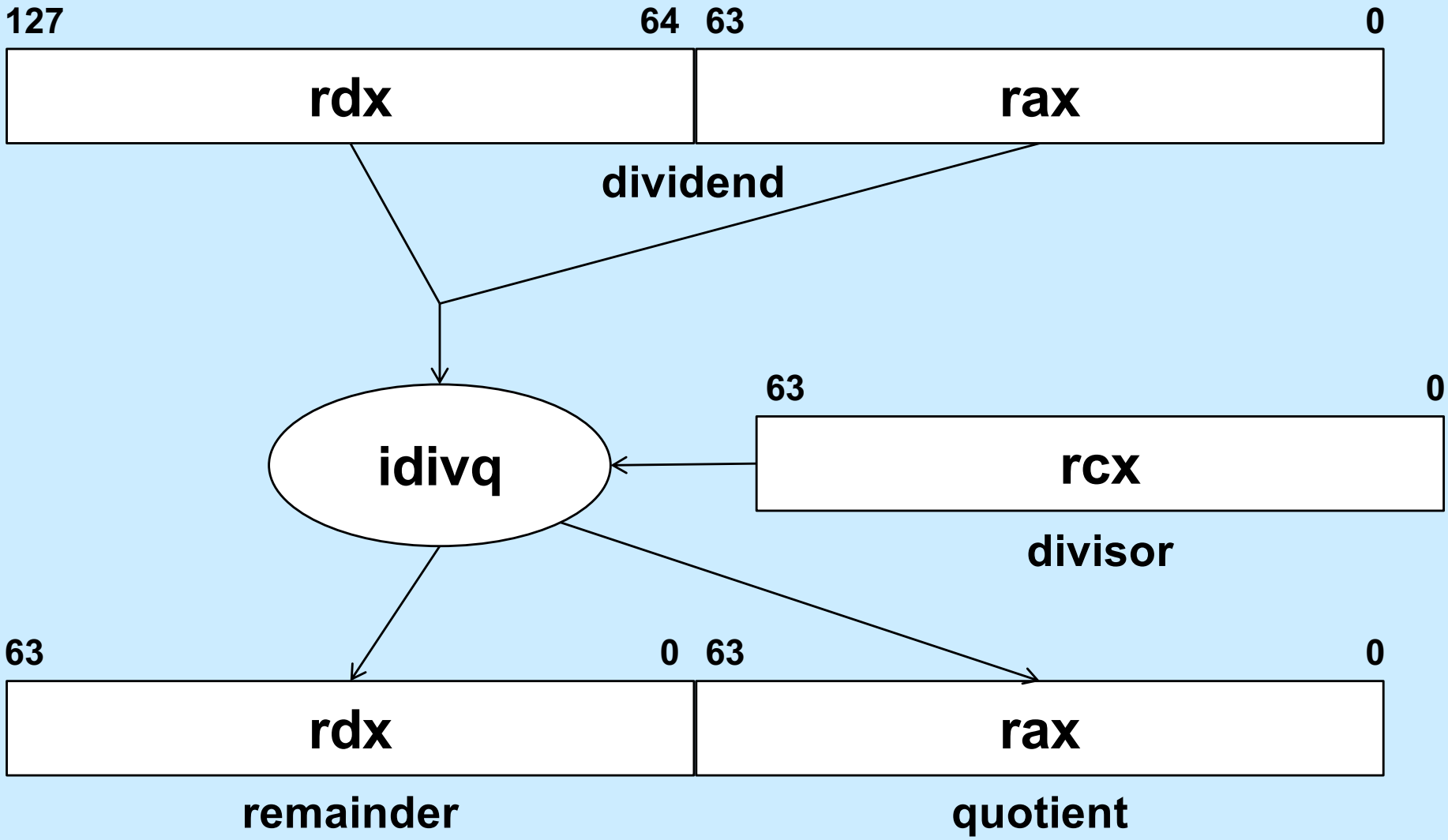
```
     case 3:
          w = 1;
merge:
          w += z;
```

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

```
0x4005c0:       0x00000000004004e5      0x00000000004004bc
0x4005d0:       0x00000000004004c4      0x00000000004004d3
0x4005e0:       0x00000000004004e5      0x00000000004004dc
0x4005f0:       0x00000000004004dc
```

x86-64 Object Code (cont.)

- **Deciphering jump table**

```
0x4005c0:    0x00000000004004e5    0x00000000004004bc
0x4005d0:    0x00000000004004c4    0x00000000004004d3
0x4005e0:    0x00000000004004e5    0x00000000004004dc
0x4005f0:    0x00000000004004dc
```

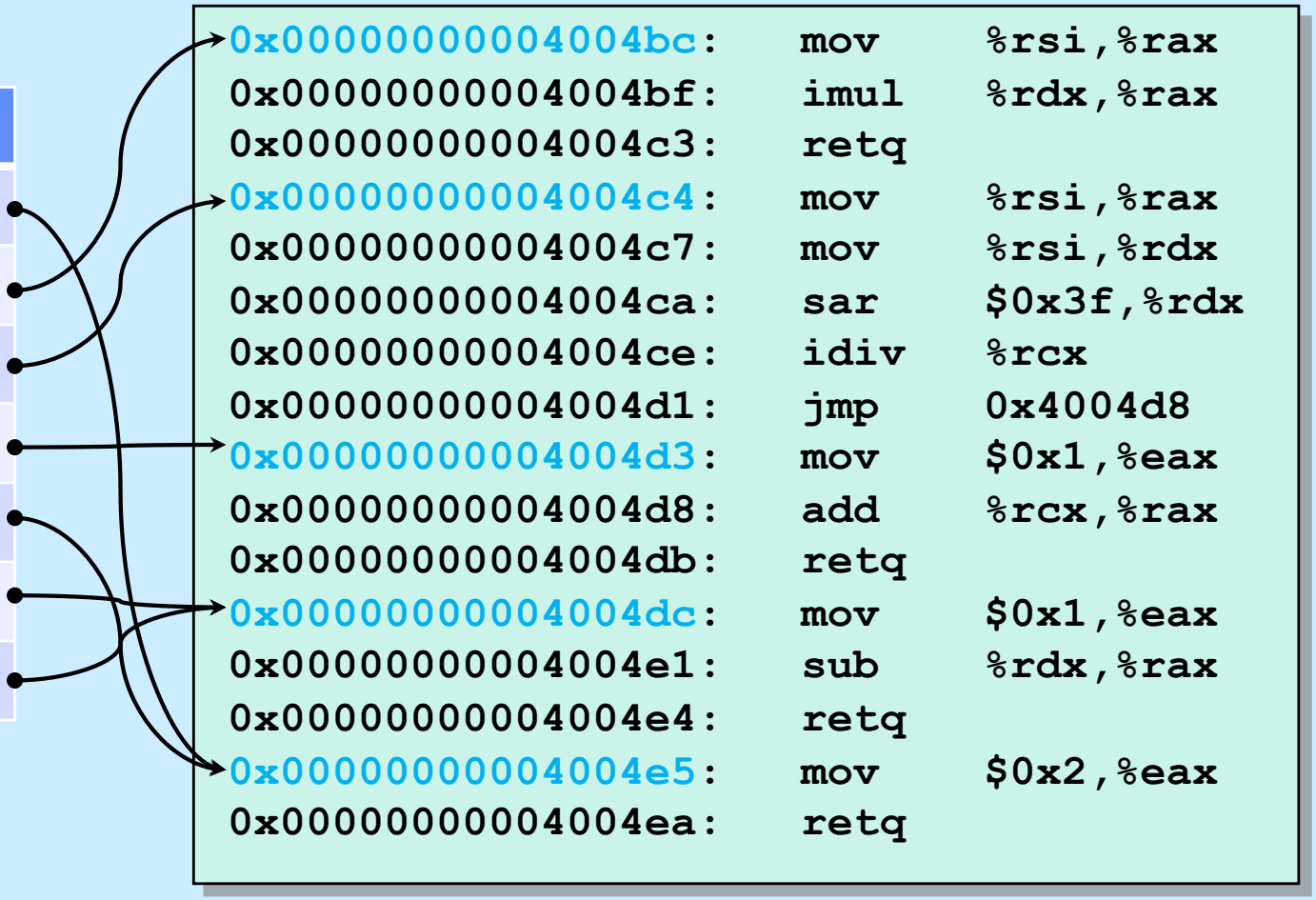
Address	Value	x
0x4005c0	0x4004e5	0
0x4005c8	0x4004bc	1
0x4005d0	0x4004c4	2
0x4005d8	0x4004d3	3
0x4005e0	0x4004e5	4
0x4005e8	0x4004dc	5
0x4005f0	0x4004dc	6

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



Quiz 3

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

```
        movq    $0, %rax
.L2:
        movq    %rax, a(,%rax,8)
        addq    $1, %rax
        cmpq    $10, %rax
        jne     .L2
        ret
```

```
long a[10];
void func() {
  long i;
  for (i=0; i<10; i++)
    a[i]= 1;
}
```

a

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

b

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

c