

# CS 33

## Machine Programming (3)

# Swapxy for Ints

```
struct xy {  
    int x;  
    int y;  
}  
  
void swapxy(struct xy *p) {  
    int temp = p->x;  
    p->x = p->y;  
    p->y = temp;  
}
```

swap:

```
movl (%rdi), %eax  
movl 4(%rdi), %edx  
movl %edx, (%rdi)  
movl %eax, 4(%rdi)  
ret
```

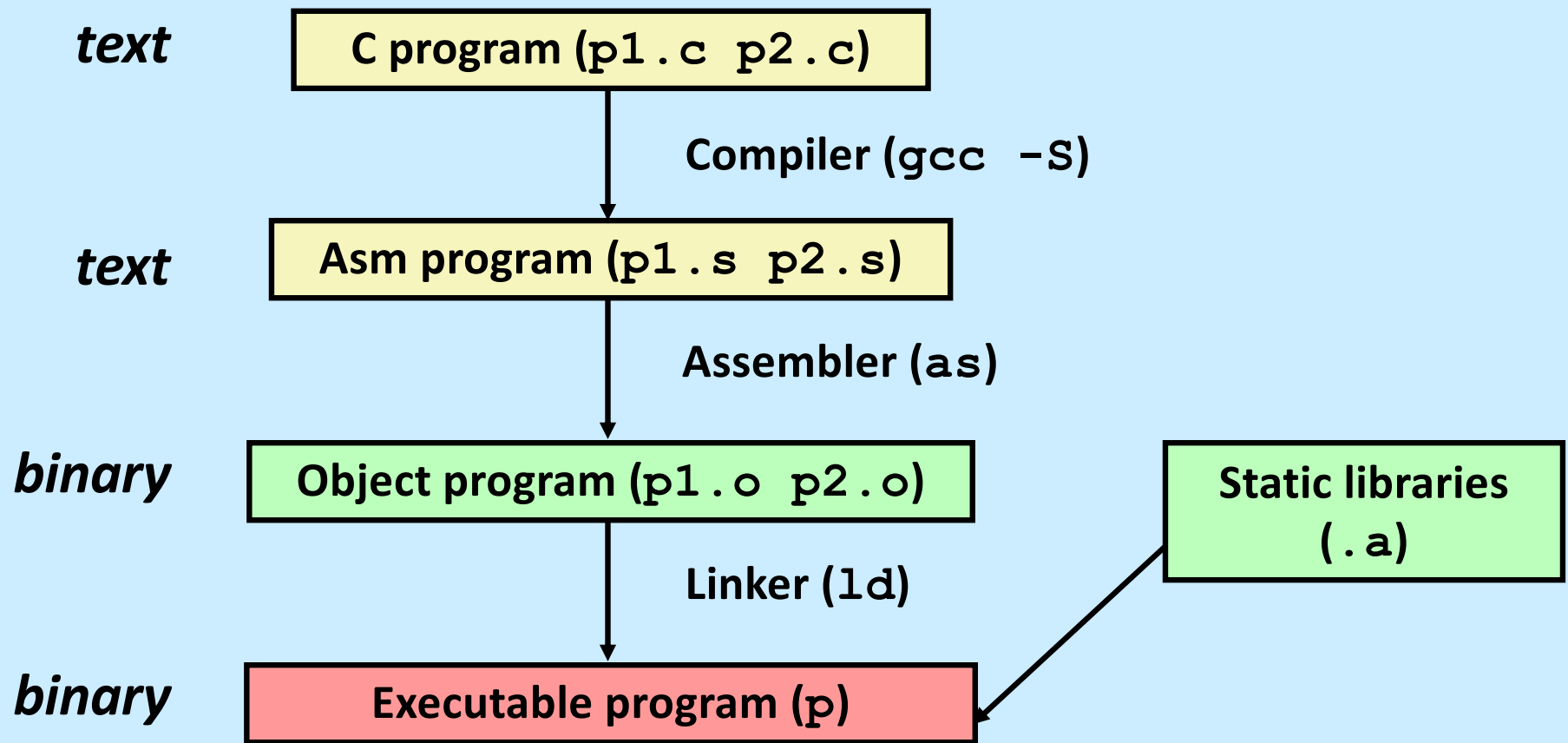
- **Pointers are 64 bits**
- **What they point to are 32 bits**

# Bytes

- **Each register has a byte version**
  - e.g., `%r10: %r10b`; see earlier slide for x86 registers
- **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

# 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

```
long ASum(long *a, unsigned long size) {  
    long i, sum = 0;  
    for (i=0; i<size; i++)  
        sum += a[i];  
    return sum;  
}
```

```
int main() {  
    long array[3] = {2,117,-6};  
    long sum = ASum(array, 3);  
    return sum;  
}
```

# Assembler Code

ASum:

```
    testq    %rsi, %rsi
    je       .L4
    movq     %rdi, %rax
    leaq     (%rdi,%rsi,8), %rcx
    movl     $0, %edx
```

.L3:

```
    addq     (%rax), %rdx
    addq     $8, %rax
    cmpq     %rcx, %rax
    jne      .L3
```

.L1:

```
    movq     %rdx, %rax
    ret
```

.L4:

```
    movl     $0, %edx
    jmp      .L1
```

main:

```
    subq     $32, %rsp
    movq     $2, (%rsp)
    movq     $117, 8(%rsp)
    movq     $-6, 16(%rsp)
    movq     %rsp, %rdi
    movl     $3, %esi
    call     ASum
    addq     $32, %rsp
    ret
```

# Object Code

## Code for ASum

0x1125 <ASum>:

0x48

0x85

0xf6

0x74

0x1c

0x48

0x89

0xf8

0x48

0x8d

0x0c

0xf7

.

.

.

- Total of 39 bytes

- Each instruction:  
1, 2, or 3 bytes

- Starts at address  
0x1125

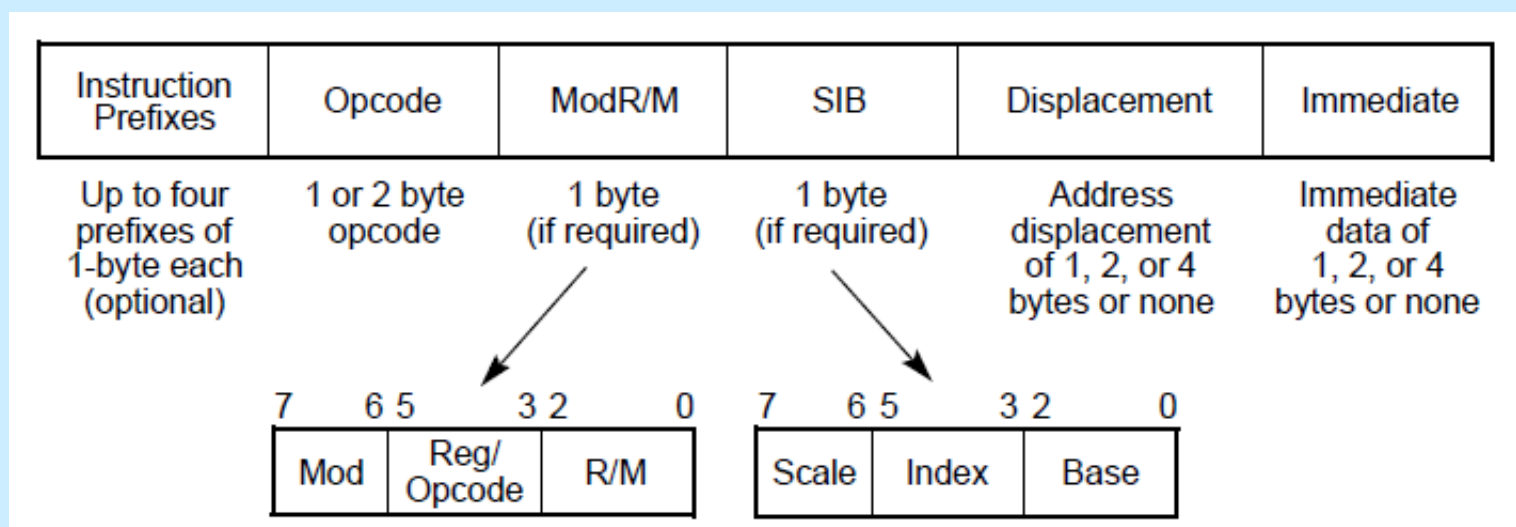
- **Assembler**

- 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

```
0000000000001125 <ASum>:
    1125:      48 85 f6          test    %rsi,%rsi
    1128:      74 1c             je      1146 <ASum+0x21>
    112a:      48 89 f8          mov     %rdi,%rax
    112d:      48 8d 0c f7     lea     (%rdi,%rsi,8),%rcx
    1131:      ba 00 00 00 00    mov     $0x0,%edx
    1136:      48 03 10          add     (%rax),%rdx
    1139:      48 83 c0 08       add     $0x8,%rax
    113d:      48 39 c8          cmp     %rcx,%rax
    1140:      75 f4             jne     1136 <ASum+0x11>
    1142:      48 89 d0          mov     %rdx,%rax
    1145:      c3               retq
    1146:      ba 00 00 00 00    mov     $0x0,%edx
    114b:      eb f5             jmp     1142 <ASum+0x1d>
```

- **Disassembler**

`objdump -d <file>`

- useful tool for examining object code
- produces approximate rendition of assembly code

# Alternate Disassembly

## Object

0x1125:

0x48

0x85

0xf6

0x74

0x1c

0x48

0x89

0xf8

0x48

0x8d

0x0c

0xf7

.

.

.

## Disassembled

Dump of assembler code for function ASum:

0x1125 <+0>: test %rsi,%rsi

0x1128 <+3>: je 0x1146 <ASum+33>

0x112a <+5>: mov %rdi,%rax

0x112d <+8>: lea (%rdi,%rsi,8),%rcx

0x1131 <+12>: mov \$0x0,%edx

...

- **Within gdb debugger**

`gdb <file>`

`disassemble ASum`

– disassemble the ASum object code

`x/39xb ASum`

– examine the 39 bytes starting at ASum

# 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	
<code>addl</code>	<code>Src, Dest</code>	<code>Dest = Dest + Src</code>
<code>subl</code>	<code>Src, Dest</code>	<code>Dest = Dest - Src</code>
<code>imull</code>	<code>Src, Dest</code>	<code>Dest = Dest * Src</code>
<code>shll</code>	<code>Src, Dest</code>	<code>Dest = Dest &lt;&lt; Src</code>
<code>sarl</code>	<code>Src, Dest</code>	<code>Dest = Dest &gt;&gt; Src</code>
<code>shrl</code>	<code>Src, Dest</code>	<code>Dest = Dest &gt;&gt; Src</code>
<code>xorl</code>	<code>Src, Dest</code>	<code>Dest = Dest ^ Src</code>
<code>andl</code>	<code>Src, Dest</code>	<code>Dest = Dest &amp; Src</code>
<code>orl</code>	<code>Src, Dest</code>	<code>Dest = Dest   Src</code>

Also called `sall`  
Arithmetic  
Logical

– watch out for argument order!

# Some Arithmetic Operations

- **One-operand Instructions**

`incl`      `Dest`       $= \text{Dest} + 1$

`decl`      `Dest`       $= \text{Dest} - 1$

`negl`      `Dest`       $= -\text{Dest}$

`notl`      `Dest`       $= \sim\text{Dest}$

- **See textbook for more instructions**
- **See Intel documentation for even more**

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

%rdx	z
%rsi	y
%rdi	x

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

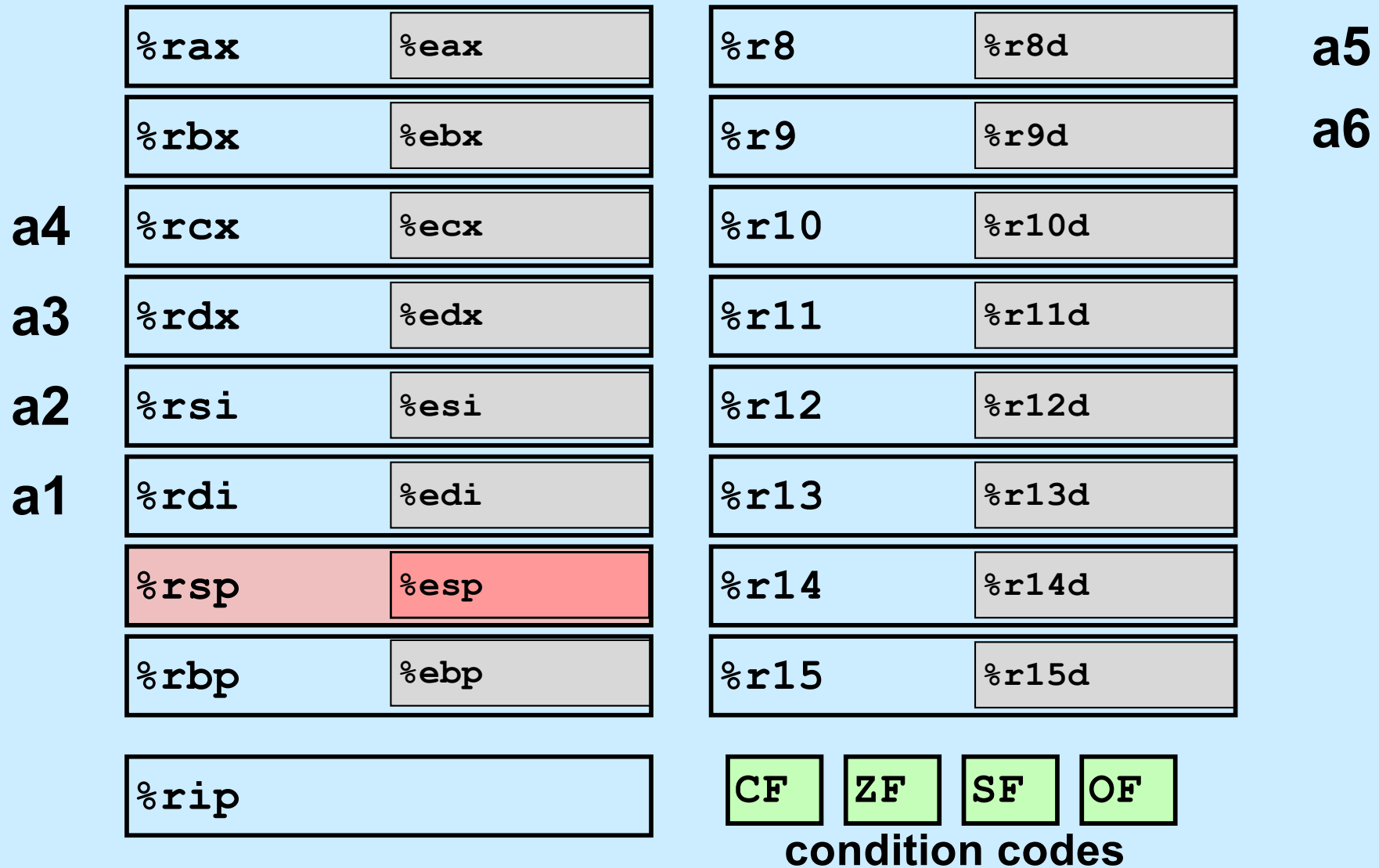
# Another Example

```
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;
}
```

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

<code>xorl %esi, %edi</code>	<code># edi = x^y</code>	<code>(t1)</code>
<code>sarl \$17, %edi</code>	<code># edi = t1&gt;&gt;17</code>	<code>(t2)</code>
<code>movl %edi, %eax</code>	<code># eax = edi</code>	
<code>andl \$8185, %eax</code>	<code># eax = t2 &amp; mask</code>	<code>(rval)</code>

# 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  $\leftrightarrow$  *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 *leal* instruction**

# Condition Codes (Explicit Setting: Compare)

- **Explicit setting by compare instruction**

`cmp1/cmpq src2, src1`

compares `src1:src2`

`cmp1 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
<b>sete</b>	<b>ZF</b>	Equal / Zero
<b>setne</b>	<b>~ZF</b>	Not Equal / Not Zero
<b>sets</b>	<b>SF</b>	Negative
<b>setns</b>	<b>~SF</b>	Nonnegative
<b>setg</b>	<b>~ (SF^OF) &amp; ~ZF</b>	Greater (Signed)
<b>setge</b>	<b>~ (SF^OF)</b>	Greater or Equal (Signed)
<b>setl</b>	<b>(SF^OF)</b>	Less (Signed)
<b>setle</b>	<b>(SF^OF)   ZF</b>	Less or Equal (Signed)
<b>seta</b>	<b>~CF &amp; ~ZF</b>	Above (unsigned)
<b>setb</b>	<b>CF</b>	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;
}
```

<b>%rax</b>	<b>%eax</b>	<b>%ah</b>	<b>%al</b>
-------------	-------------	------------	------------

## Body

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



# Jumping

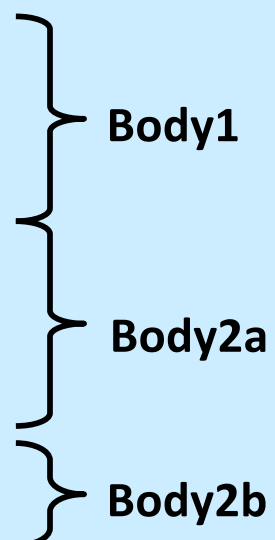
- **jX instructions**
  - Jump to different part of program depending on condition codes

jX	Condition	Description
<b>jmp</b>	<b>1</b>	Unconditional
<b>j<sub>e</sub></b>	<b>ZF</b>	Equal / Zero
<b>j<sub>ne</sub></b>	<b>~ZF</b>	Not Equal / Not Zero
<b>j<sub>s</sub></b>	<b>SF</b>	Negative
<b>j<sub>ns</sub></b>	<b>~SF</b>	Nonnegative
<b>j<sub>g</sub></b>	<b>~ (SF^OF) &amp; ~ZF</b>	Greater (Signed)
<b>j<sub>ge</sub></b>	<b>~ (SF^OF)</b>	Greater or Equal (Signed)
<b>j<sub>l</sub></b>	<b>(SF^OF)</b>	Less (Signed)
<b>j<sub>le</sub></b>	<b>(SF^OF)   ZF</b>	Less or Equal (Signed)
<b>j<sub>a</sub></b>	<b>~CF &amp; ~ZF</b>	Above (unsigned)
<b>j<sub>b</sub></b>	<b>CF</b>	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

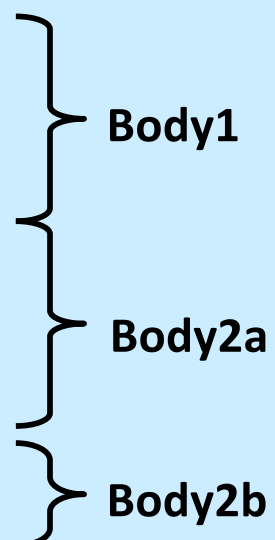
x in %edi

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 and 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);
```

- **Body:** {  
    Statement<sub>1</sub>;  
    Statement<sub>2</sub>;  
    ...  
    Statement<sub>n</sub>;  
}
- **Test returns integer**  
    = 0 interpreted as false  
    ≠ 0 interpreted as true

## Goto Version

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

# “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;  
}
```

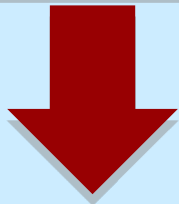
- Is this code equivalent to the do-while version?
  - must jump out of loop if test fails



# 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++
```

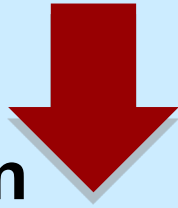
## Body

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

# “For” Loop → While Loop

## For Version

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



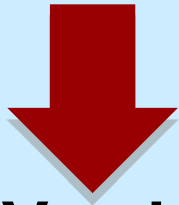
## While Version

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

# “For” Loop → ... → Goto

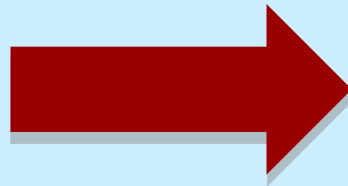
## For Version

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

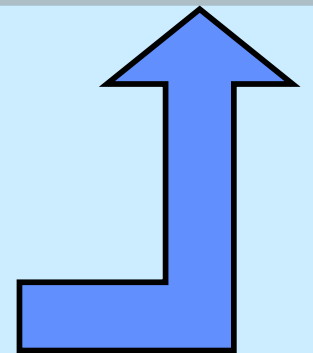


## While Version

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

# Switch-Statement Example

```
long switch_eg (long m, long d) {  
    if (d < 1) return 0;  
    switch(m) {  
        case 1: case 3: case 5:  
        case 7: case 8: case 10:  
        case 12:  
            if (d > 31) return 0;  
            else return 1;  
        case 2:  
            if (d > 28) return 0;  
            else return 1;  
        case 4: case 6: case 9:  
        case 11:  
            if (d > 30) return 0;  
            else return 1;  
        default:  
            return 0;  
    }  
    return 0;  
}
```

# Offset Structure

## Switch Form

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

## Jump Offset Table

Otab:	Targ0 Offset
	Targ1 Offset
	Targ2 Offset
	•
	•
	•
	Targn-1 Offset

## Jump Targets

Targ0:

Code Block 0

Targ1:

Code Block 1

Targ2:

Code Block 2

•  
•  
•

Targn-1:

Code Block n-1

## Approximate Translation

```
target = Otab + OTab[x];  
goto *target;
```



# Assembler Code (1)

```
switch_eg:                                .section      .rodata
    movl    $0, %eax                      .align 4
    testq   %rsi, %rsi                    .L4:
    jle     .L1                            .long    .L8-.L4
    cmpq    $12, %rdi                     .long    .L3-.L4
    ja      .L8                            .long    .L6-.L4
    leaq    .L4(%rip), %rdx               .long    .L3-.L4
    movslq   (%rdx,%rdi,4), %rax          .long    .L5-.L4
    addq     %rdx, %rax                   .long    .L3-.L4
    jmp      *%rax                        .long    .L5-.L4
                                              .long    .L3-.L4
                                              .long    .L3-.L4
                                              .long    .L5-.L4
                                              .long    .L3-.L4
                                              .long    .L5-.L4
                                              .long    .L3-.L4
                                              .text
```

# Assembler Code (2)

.L3:

```
    cmpq    $31, %rsi
    setle   %al
    movzbl  %al, %eax
    ret
```

.L6:

```
    cmpq    $28, %rsi
    setle   %al
    movzbl  %al, %eax
    ret
```

.L5:

```
    cmpq    $30, %rsi
    setle   %al
    movzbl  %al, %eax
    ret
```

.L8:

```
    movl    $0, %eax
```

.L1:

```
    ret
```

# Assembler Code Explanation (1)

switch\_eg:

```
    movl    $0, %eax    # return value set to 0
    testq   %rsi, %rsi  # sets cc based on %rsi & %rsi
    jle     .L1         # go to L1, where it returns 0
    cmpq    $12, %rdi
    ja      .L8
    leaq    .L4(%rip), %rdx
    movslq  (%rdx,%rdi,4), %rax
    addq    %rdx, %rax
    jmp     *%rax
```

- **testq %rsi, %rsi**
  - **sets cc based on the contents of %rsi (d)**
  - **jle**
    - **jumps if  $(SF \oplus OF) \mid ZF$**
    - **OF is not set**
    - **jumps if SF or ZF is set (i.e.,  $< 1$ )**

# Assembler Code Explanation (2)

switch\_eg:

```
    movl    $0, %eax        # return value set to 0
    testq   %rsi, %rsi      # sets cc based on %rsi & %rsi
    jle     .L1             # go to L1, where it returns 0
    cmpq     $12, %rdi      # %rdi : 12
    ja       .L8           # go to L8 if %rdi > 12 or < 0
    leaq    .L4(%rip), %rdx
    movslq  (%rdx,%rdi,4), %rax
    addq    %rdx, %rax
    jmp     *%rax
```

- **ja .L8**
  - **unsigned comparison, though m is signed!**
  - **jumps if %rdi > 12**
  - **also jumps if %rdi is negative**

# Assembler Code Explanation (3)

```
switch_eg:                                     .section      .rodata
    movl    $0, %eax                           .align 4
    testq   %rsi, %rsi                          .L4:
    jle     .L1                                .long      .L8-.L4 # m=0
    cmpq    $12, %rdi                           .long      .L3-.L4 # m=1
    ja      .L8                                .long      .L6-.L4 # m=2
    leaq    .L4(%rip), %rdx                     .long      .L3-.L4 # m=3
    movslq   (%rdx,%rdi,4), %rax                 .long      .L5-.L4 # m=4
    addq     %rdx, %rax                         .long      .L3-.L4 # m=5
    jmp      *%rax                             .long      .L5-.L4 # m=6
                                                .long      .L3-.L4 # m=7
                                                .long      .L3-.L4 # m=8
                                                .long      .L5-.L4 # m=9
                                                .long      .L3-.L4 # m=10
                                                .long      .L5-.L4 # m=11
                                                .long      .L3-.L4 # m=12
                                                .text
```

# Assembler Code Explanation (4)

```
switch_eg:
    movl    $0, %eax
    testq   %rsi, %rsi
    jle     .L1
    cmpq    $12, %rdi
    ja      .L8
    leaq    .L4(%rip), %rdx
    movslq   (%rdx,%rdi,4), %rax
    addq    %rdx, %rax
    jmp     *%rax
.L4:
    .long   .L8-.L4 # m=0
    .long   .L3-.L4 # m=1
    .long   .L6-.L4 # m=2
    .long   .L3-.L4 # m=3
    .long   .L5-.L4 # m=4
    .long   .L3-.L4 # m=5
    .long   .L5-.L4 # m=6
    .long   .L3-.L4 # m=7
    .long   .L3-.L4 # m=8
    .long   .L5-.L4 # m=9
    .long   .L3-.L4 # m=10
    .long   .L5-.L4 # m=11
    .long   .L3-.L4 # m=12
    .text
```

**indirect jump**

# Assembler Code Explanation (5)

```
switch_eg:
    movl    $0, %eax
    testq   %rsi, %rsi
    jle     .L1
    cmpq    $12, %rdi
    ja      .L8
    leaq    .L4(%rip), %rdx
    movslq   (%rdx,%rdi,4), %rax
    addq     %rdx, %rax
    jmp      *%rax

.L4:
    .section      .rodata
    .align 4
    .long    .L8-.L4 # m=0
    .long    .L3-.L4 # m=1
    .long    .L6-.L4 # m=2
    .long    .L3-.L4 # m=3
    .long    .L5-.L4 # m=4
    .long    .L3-.L4 # m=5
    .long    .L5-.L4 # m=6
    .long    .L3-.L4 # m=7
    .long    .L3-.L4 # m=8
    .long    .L5-.L4 # m=9
    .long    .L3-.L4 # m=10
    .long    .L5-.L4 # m=11
    .long    .L3-.L4 # m=12
    .text
```

# Assembler Code Explanation (6)

```
switch_eg:
    movl    $0, %eax
    testq   %rsi, %rsi
    jle     .L1
    cmpq    $12, %rdi
    ja      .L8
    leaq    .L4(%rip), %rdx
    movslq   (%rdx,%rdi,4), %rax
    addq    %rdx, %rax
    jmp     *%rax

.L4:
    .section .rodata
    .align 4
    .long    .L8-.L4 # m=0
    .long    .L3-.L4 # m=1
    .long    .L6-.L4 # m=2
    .long    .L3-.L4 # m=3
    .long    .L5-.L4 # m=4
    .long    .L3-.L4 # m=5
    .long    .L5-.L4 # m=6
    .long    .L3-.L4 # m=7
    .long    .L3-.L4 # m=8
    .long    .L5-.L4 # m=9
    .long    .L3-.L4 # m=10
    .long    .L5-.L4 # m=11
    .long    .L3-.L4 # m=12
    .text
```



# Assembler Code Explanation (7)

```
switch_eg:
    movl    $0, %eax
    testq   %rsi, %rsi
    jle     .L1
    cmpq    $12, %rdi
    ja      .L8
    leaq    .L4(%rip), %rdx
    movslq   (%rdx,%rdi,4), %rax
    addq     %rdx, %rax
    jmp      *%rax

.L4:
    .long    .L8-.L4 # m=0
    .long    .L3-.L4 # m=1
    .long    .L6-.L4 # m=2
    .long    .L3-.L4 # m=3
    .long    .L5-.L4 # m=4
    .long    .L3-.L4 # m=5
    .long    .L5-.L4 # m=6
    .long    .L3-.L4 # m=7
    .long    .L3-.L4 # m=8
    .long    .L5-.L4 # m=9
    .long    .L3-.L4 # m=10
    .long    .L5-.L4 # m=11
    .long    .L3-.L4 # m=12
    .text
```

**.section .rodata**  
**.align 4**

# Switch Statements and Traps

- The code we just looked at was compiled with gcc's O1 flag
  - a moderate amount of “optimization”
- Traps is compiled with the O0 flag
  - no optimization
- O0 often produces easier-to-read (but less efficient) code
  - not so for switch

# O1 vs. O0 Code

switch\_egO1:

```
movl    $0, %eax
testq   %rsi, %rsi
jle     .L1
cmpq    $12, %rdi
ja      .L8
leaq    .L4(%rip), %rdx
movslq  (%rdx,%rdi,4), %rax
addq    %rdx, %rax
jmp     *%rax
```

switch\_eg00:

```
pushq   %rbp
movq    %rsp, %rbp
movq    %rdi, -8(%rbp)
movq    %rsi, -16(%rbp)
cmpq    $0, -16(%rbp)
jg      .L2
movl    $0, %eax
jmp     .L3
.L2:
cmpq    $12, -8(%rbp)
ja      .L4
movq    -8(%rbp), %rax
leaq    0(,%rax,4), %rdx
leaq    .L6(%rip), %rax
movl    (%rdx,%rax), %eax
cltq
leaq    .L6(%rip), %rdx
addq    %rdx, %rax
jmp     *%rax
```

# O1 vs. O0 Code Explanation (1)

switch\_eg01:

```
movl    $0, %eax
testq   %rsi, %rsi
jle     .L1
cmpq    $12, %rdi
ja      .L8
leaq    .L4(%rip), %rdx
movslq  (%rdx,%rdi,4), %rax
addq    %rdx, %rax
jmp     *%rax
```

switch\_eg00:

```
pushq   %rbp
movq    %rsp, %rbp
movq    %rdi, -8(%rbp)
movq    %rsi, -16(%rbp)
cmpq    $0, -16(%rbp)
jg      .L2
movl    $0, %eax
jmp     .L3
.L2:
cmpq    $12, -8(%rbp)
ja      .L4
movq    -8(%rbp), %rax
leaq    0(,%rax,4), %rdx
leaq    .L6(%rip), %rax
movl    (%rdx,%rax), %eax
cltq
leaq    .L6(%rip), %rdx
addq    %rdx, %rax
jmp     *%rax
```

# O1 vs. O0 Code Explanation (2)

switch\_eg01:

```
movl    $0, %eax
testq   %rsi, %rsi
jle     .L1
cmpq    $12, %rdi
ja      .L8
leaq    .L4(%rip), %rdx
movslq  (%rdx,%rdi,4), %rax
addq    %rdx, %rax
jmp     *%rax
```

switch\_eg00:

```
pushq   %rbp
movq    %rsp, %rbp
movq    %rdi, -8(%rbp)
movq    %rsi, -16(%rbp)
cmpq    $0, -16(%rbp)
jg      .L2
movl    $0, %eax
jmp     .L3
```

.L2:

```
cmpq    $12, -8(%rbp)
ja      .L4
movq    -8(%rbp), %rax
leaq    0(,%rax,4), %rdx
leaq    .L6(%rip), %rax
movl    (%rdx,%rax), %eax
cltq
leaq    .L6(%rip), %rdx
addq    %rdx, %rax
jmp     *%rax
```

# O1 vs. O0 Code Explanation (3)

switch\_eg01:

```
movl    $0, %eax
testq   %rsi, %rsi
jle     .L1
cmpq    $12, %rdi
ja      .L8
leaq    .L4(%rip), %rdx
movslq  (%rdx,%rdi,4), %rax
addq    %rdx, %rax
jmp     *%rax
```

switch\_eg00:

```
pushq   %rbp
movq    %rsp, %rbp
movq    %rdi, -8(%rbp)
movq    %rsi, -16(%rbp)
cmpq    $0, -16(%rbp)
jg      .L2
movl    $0, %eax
jmp     .L3
```

.L2:

```
cmpq    $12, -8(%rbp)
ja      .L4
movq    -8(%rbp), %rax
leaq    0(,%rax,4), %rdx
leaq    .L6(%rip), %rax
movl    (%rdx,%rax), %eax
cltq
leaq    .L6(%rip), %rdx
addq    %rdx, %rax
jmp     *%rax
```

# O1 vs. O0 Code Explanation (4)

switch\_eg01:

```
movl    $0, %eax
testq   %rsi, %rsi
jle     .L1
cmpq    $12, %rdi
ja      .L8
leaq    .L4(%rip), %rdx
movslq  (%rdx,%rdi,4), %rax
addq    %rdx, %rax
jmp     *%rax
```

switch\_eg00:

```
pushq   %rbp
movq    %rsp, %rbp
movq    %rdi, -8(%rbp)
movq    %rsi, -16(%rbp)
cmpq    $0, -16(%rbp)
jg      .L2
movl    $0, %eax
jmp     .L3
```

.L2:

```
cmpq    $12, -8(%rbp)
ja      .L4
movq    -8(%rbp), %rax
leaq    0(,%rax,4), %rdx
leaq    .L6(%rip), %rax
movl    (%rdx,%rax), %eax
cltq
leaq    .L6(%rip), %rdx
addq    %rdx, %rax
jmp     *%rax
```

# O1 vs. O0 Code Explanation (5)

switch\_eg01:

```
movl    $0, %eax
testq   %rsi, %rsi
jle     .L1
cmpq    $12, %rdi
ja      .L8
leaq    .L4(%rip), %rdx
movslq  (%rdx,%rdi,4), %rax
addq    %rdx, %rax
jmp     *%rax
```

switch\_eg00:

```
pushq   %rbp
movq    %rsp, %rbp
movq    %rdi, -8(%rbp)
movq    %rsi, -16(%rbp)
cmpq    $0, -16(%rbp)
jg      .L2
movl    $0, %eax
jmp     .L3
```

.L2:

```
cmpq    $12, -8(%rbp)
ja      .L4
movq    -8(%rbp), %rax
leaq    0(,%rax,4), %rdx
leaq    .L6(%rip), %rax
movl    (%rdx,%rax), %eax
cltq
leaq    .L6(%rip), %rdx
addq    %rdx, %rax
jmp     *%rax
```



# Gdb and Switch (1)

```
B+ 0x55555555169 <switch_eg+4>      mov     %rdi,-0x8(%rbp)
    0x5555555516d <switch_eg+8>      mov     %rsi,-0x10(%rbp)
    0x55555555171 <switch_eg+12>     cmpq    $0x0,-0x10(%rbp)
    0x55555555176 <switch_eg+17>     jg      0x5555555517f <nswitch+26>
    0x55555555178 <switch_eg+19>     mov     $0x0,%eax
    0x5555555517d <switch_eg+24>     jmp     0x555555551ee <nswitch+137>
    0x5555555517f <switch_eg+26>     cmpq    $0xc,-0x8(%rbp)
    0x55555555184 <switch_eg+31>     ja      0x555555551e9 <nswitch+132>
    0x55555555186 <switch_eg+33>     mov     -0x8(%rbp),%rax
    0x5555555518a <switch_eg+37>     lea     0x0(,%rax,4),%rdx
    0x55555555192 <switch_eg+45>     lea     0xe6b(%rip),%rax          # 0x55555555
    0x55555555199 <switch_eg+52>     mov     (%rdx,%rax,1),%eax
    0x5555555519c <switch_eg+55>     cltq
    0x5555555519e <switch_eg+57>     lea     0xe5f(%rip),%rdx          # 0x55555555
    0x555555551a5 <switch_eg+64>     add     %rdx,%rax
>0x555555551a8 <switch_eg+67>     jmp     *%rax
```

```
(gdb) x/14dw $rdx
```

```
0x555555556004: -3611    -3674    -3653    -3674
0x555555556014: -3632    -3674    -3632    -3674
0x555555556024: -3674    -3632    -3674    -3632
0x555555556034: -3674    1734439765
```

# Gdb and Switch (2)

```
0x555555551a5 <switch_eg+64>    add    %rdx,%rax
>0x555555551a8 <switch_eg+67>    jmp     *%rax
0x555555551aa <switch_eg+69>    cmpq   $0x1f,-0x10(%rbp)
0x555555551af <switch_eg+74>    jle    0x555555551b8 <nswitch+83>
0x555555551b1 <switch_eg+76>    mov     $0x0,%eax
0x555555551b6 <switch_eg+81>    jmp     0x555555551ee <nswitch+137>
0x555555551b8 <switch_eg+83>    mov     $0x1,%eax
0x555555551bd <switch_eg+88>    jmp     0x555555551ee <nswitch+137>
0x555555551bf <switch_eg+90>    cmpq   $0x1c,-0x10(%rbp)
0x555555551c4 <switch_eg+95>    jle    0x555555551cd <nswitch+104>
```

```
(gdb) x/14dw $rdx
```

```
0x555555556004: -3611    -3674    -3653    -3674
0x555555556014: -3632    -3674    -3632    -3674
0x555555556024: -3674    -3632    -3674    -3632
0x555555556034: -3674    1734439765
```

# Quiz 1

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=0;
    while (i<10)
        a[i]= i++;
}
```

**a**

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

**b**

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

**c**