CS 33

Machine Programming (3)

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.

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
```

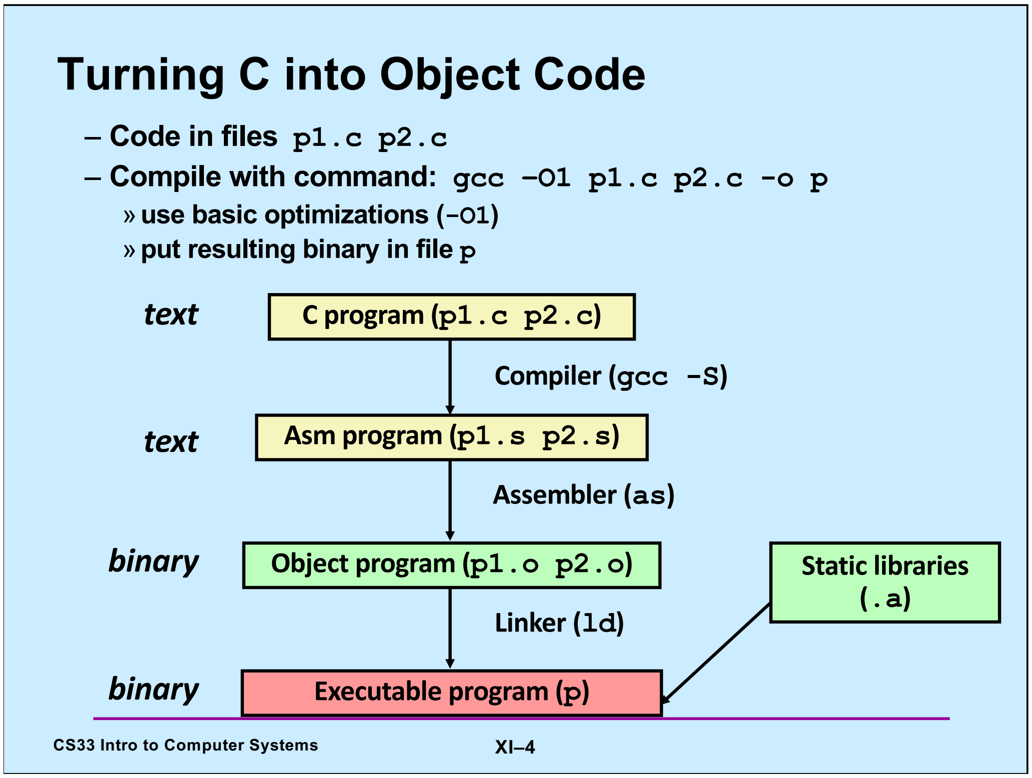
- **Pointers are 64 bits**
- **What they point to are 32 bits**

Here we have a simple function that swaps the two components of a structure that's passed to it. (Assume that %rdi contains the argument.) Note that even though we use the "e" form of the registers to hold the (32-bit) data, we need the "r" form to hold the 64-bit addresses.

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

Note that using single-byte versions of registers has a different behavior from using 4-byte versions of registers. Putting data into the latter using **mov** causes the upper bytes to be zeroed. But with the byte versions, putting data into them does not affect the upper bytes.



Supplied by CMU.

Note that normally one does not ask gcc to produce assembler code, but instead it compiles C code directly into machine code (producing an object file). Note also that the gcc command actually invokes a script; the compiler (also known as gcc) compiles code into either assembler code or machine code; if necessary, the assembler (as) assembles assembler code into object code. The linker (ld) links together multiple object files (containing object code) into an executable program.

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;
}
```

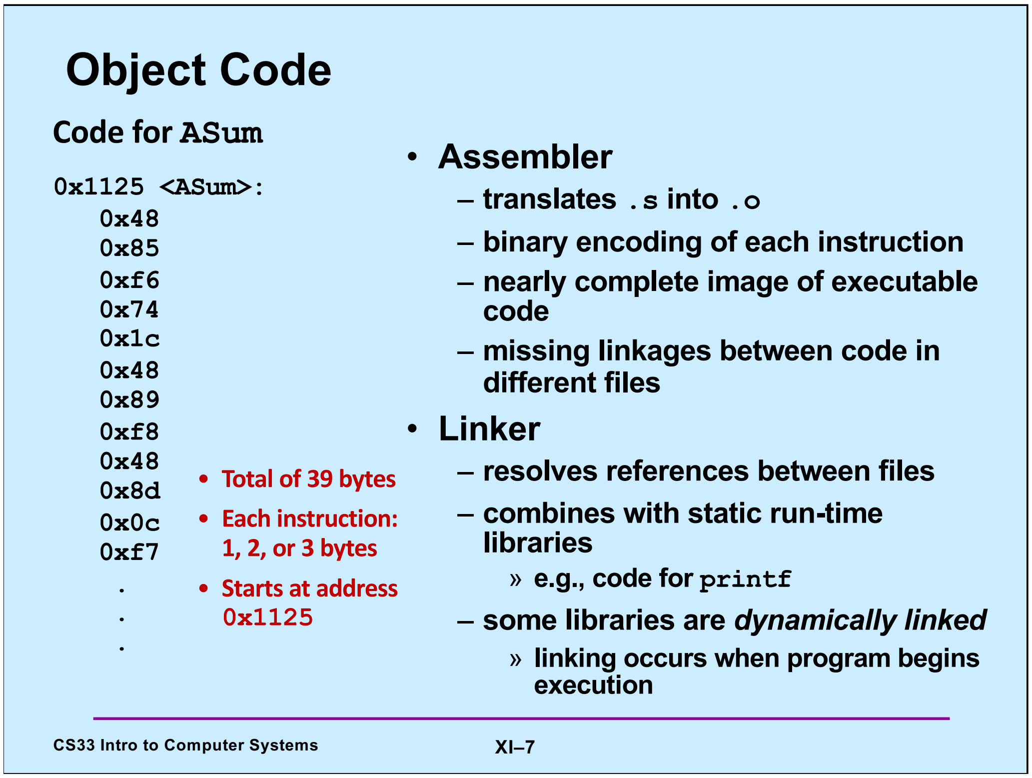
Assembler Code

```
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
```


Here is the assembler code produced by gcc from the C code of the previous slide. The testq instruction is checking whether the size (second argument) of the array is 0. If it is, then there will be no iterations of the for loop and the program returns right away.

Note that the two movl instructions are ostensibly just copying a zero into %edx (a 32-bit register). However, what it's really doing is copying a zero in the 64-bit register %rdx (the 64-bit extension of %edx). This happens because, as we discussed earlier, when one copies something into a 32-bit register, the high-order 32 bits of its extension is filled with 0s.

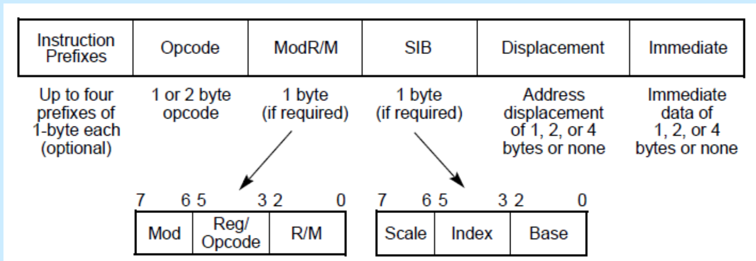


Adapted from a slide supplied by CMU.

The lefthand column shows the object code produced by gcc. This was produced either by assembling the code of the previous slide, or by compiling the C code of the slide before that.

Suppose that all we have is the object code (the output of the assembler) – we don’t have the assembler code and the C code. Can we translate for object code to assembler code? (This is known as disassembling.)

Instruction Format



This is taken from Intel 64 and IA-32 Architecture Software Developer's Manual, Volume 2: Instruction Set Reference; Order Number 325462-043US, Intel Corporation, May 2012 (https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4)

The point of the slide is that the instruction format is complicated, too much so for a human to deal with. Which is why we talk about **disassemblers** in the next slides.

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**

Adapted from a slide supplied by CMU.

objdump's rendition is approximate because it assumes everything in the file is assembly code, and thus translates data into (often really weird) assembly code. Also, it leaves off the suffix at the end of each instruction, assuming it can be determined from context.

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`

Adapted from a slide supplied by CMU.

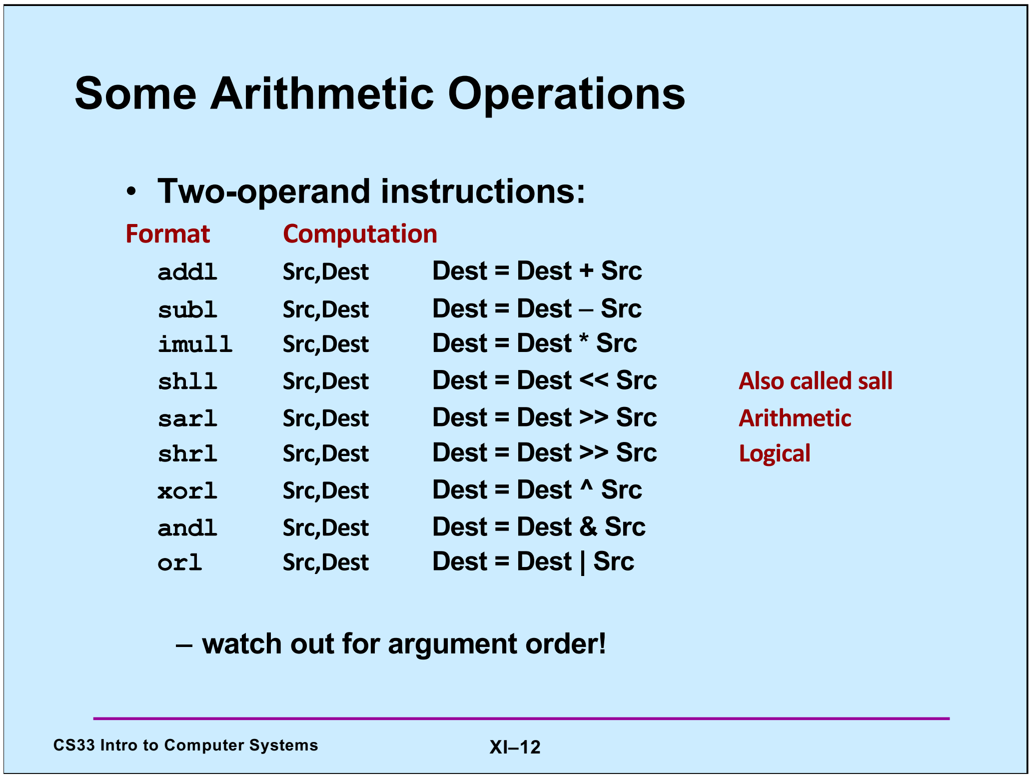
The "x/35xb" directive to gdb says to examine (first x, meaning print) 35 bytes (b) viewed as hexadecimal (second x) starting at ASum.

The format of the output has been modified a bit from what gdb actually produces, so that it will fit on the slide. In the dump of the assembler code, the addresses are actually 64-bit values (in hex) – we have removed the leading 0s. The output of the x command is actually displayed in multiple columns. We have reorganized it into one column.

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

The source for this is http://en.wikipedia.org/wiki/X86_instruction_listings, viewed on 6/20/2017, which came with the caveat that it may be out of date. While it's likely that more instructions have been added since then, we won't be covering them in 33!



Supplied by CMU.

Note that for shift instructions, the Src operand (which is the size of the shift) must either be an immediate operand or be a designator for a one-byte register (e.g., %cl – see the slide on general-purpose registers for IA32).

Also note that what's given in the slide are the versions for 32-bit operands. There are also versions for 8-, 16-, and 64-bit operands, with the "l" replaced with the appropriate letter ("b", "s", or "q").

The **imul** instruction performs a signed multiply; the **mul** instruction performs an unsigned multiply. This is one of the few instances in which different instructions are required for signed and unsigned integers. The reason for this is to make certain, for the signed case, that the sign of the result is correct (see slides VIII-13 and VIII-14).

Some Arithmetic Operations

- **One-operand Instructions**

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

- **See textbook for more instructions**

- **See Intel documentation for even more**

Adapted from a slide supplied by CMU.

Quiz 1

- **What is the value stored in %r9 after the following code is execute?**

```
movq $8, %r8
movq $9, %r9
addq %r9, %r8
addq %r8, %r9
addq %r9, %r8
```

a) 17
b) 26
c) 42
d) 43

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
```


Supplied by CMU, but converted to x86-64.

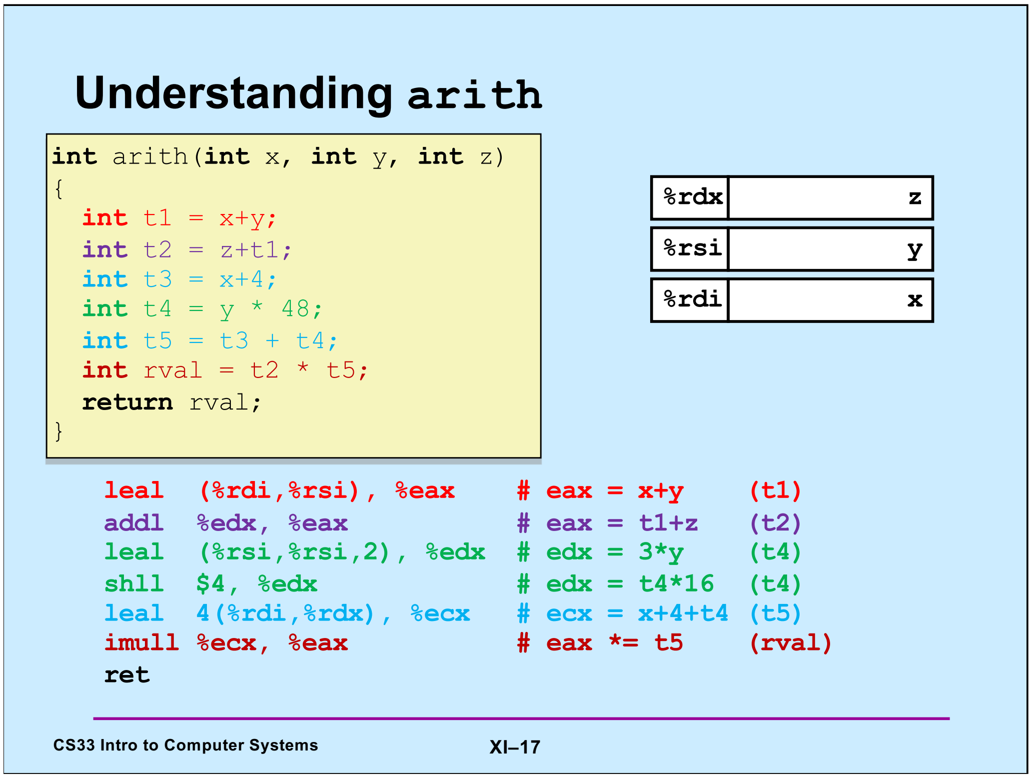
Understanding `arith`

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


Supplied by CMU, but converted to x86-64.



Supplied by CMU, but converted to x86-64.

By convention, the first three arguments to a function are placed in registers **rdi**, **rsi**, and **rdx**, respectively. Note that, also by convention, functions put their return values in register **eax**/**rax**.

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
```


Supplied by CMU, but converted to x86-64.

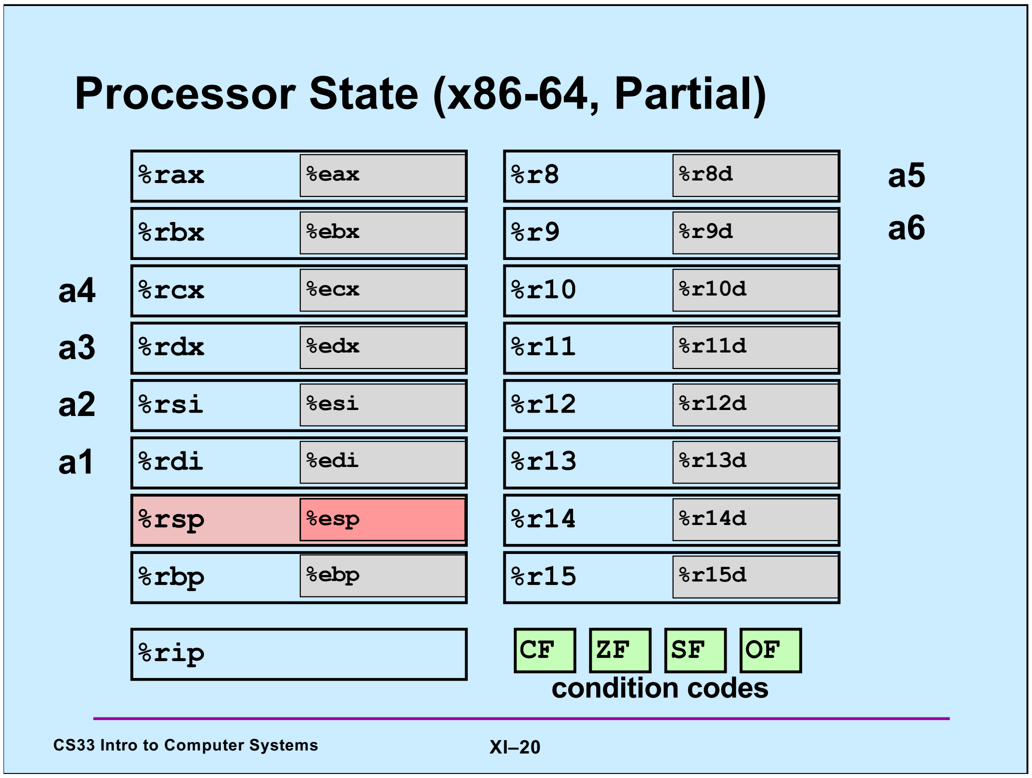
Another Example

```
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)
ret
```


Supplied by CMU, but converted to x86-64.



Supplied by CMU, but converted to x86-64.

%rip is the instruction-pointer register. It contains the address of the next instruction to be executed. CF, ZF, SF, and OF are the condition codes, referring to carry flag, zero flag, sign flag, and 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**

Supplied by CMU.

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)`

Supplied by CMU.

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.

Note that if a&b<0, what is meant is that the most-significant bit is 1.

Quiz 2

- **The following code is executed:**

```
movq $10, %r8
movq $-11, %r9
cmpq %r8, %r9
```

Which of the condition codes will be 1?

a) CF, SF, ZF, OF
b) SF, OF
c) SF
d) CF, OF

Reading Condition Codes

- **SetX instructions**
  - **set single byte (to 1 or 0) 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)

Supplied by CMU.

These operations allow one to set a byte depending on the values of the condition codes.

Some of these conditions aren't all that obvious. Suppose we are comparing A with B (**cmpl** B,A). Thus the condition codes would be set as if we computed A-B. For signed arithmetic, If A >= B, then the true result is non-negative. But some issues come up because of two's complement arithmetic with a finite word size. If overflow does not occur, then the sign flag should not be set. If overflow does occur (because A is positive, B is negative, and A-B is a large positive number that does not fit in an int), then even though the true result should have been positive, the actual result is negative. So, if both the sign flag and the overflow flag are not set, we know that A >= B. If both flags are set, we know the true result of the subtraction is positive and thus A>=B. But if one of the two flags is set and the other isn't, then A must be less than B. Thus if ~(SF^OF) is 1, we know that A>=B. If ZF (zero flag) is set, we know that A==B. Thus for A>B, ZF is not set.

For unsigned arithmetic, if A>B, then subtracting B from A doesn't require a borrow and thus CF is not set; and since A is not equal to B, ZF is not set. If A<B, then subtracting B from A requires a borrow and thus CF is set.

The other cases can be worked out similarly.

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
```


Supplied by CMU, but converted to x86-64.

Recall that the first argument to a function is passed in %rdi (%edi) and the second in %rsi (%esi).

Also recall that **movzbl** copies the byte in the first operand to the second operand, and zeroes the higher-order bytes. Since %eax is a 32-bit register, this operation zeroes the higher-order bytes of %rax, the containing 64-bit register.

Jumping

- jX instructions
  - Jump to different part of program 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)

Supplied by CMU.

See the notes for slide 25.

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: movl, cmpl, jle
Body2a: subl, movl, jmp
Body2b: subl %edi, %eax

x in %edi

y in %esi

Supplied by CMU, but converted to x86-64.

The function computes the absolute value of the difference between its two arguments.

Conditional-Branch Example (Cont.)

```c
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

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

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

Body1

Body2a

Body2b

Supplied by CMU, but converted to x86-64.

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:
  . . .
```

- **<u>Test</u> is expression returning integer**
  - **== 0 interpreted as false**
  - **≠ 0 interpreted as true**
- **Create separate code regions for <u>then</u> and <u>else</u> expressions**
- **Execute appropriate one**

Supplied by CMU.

C's conditional expression, as shown in the slide, is sometimes useful, but often results in really difficult-to-read code.

(There's an "International Obfuscated C Code Contest" (IOCCC) that awards prizes to those who use valid syntax to write the most difficult-to-understand implementations of simple functions. The conditional expression features prominently in winners' code. See https://www.ioccc.org/.)

“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**

Supplied by CMU.

"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   $1, %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);
```

Goto Version

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

- **Body:**
```
{
  Statement1;
  Statement2;
    …
  Statementn;
}
```
- **Test returns integer**
  - = 0 interpreted as false
  - ≠ 0 interpreted as true

Supplied by CMU.

"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**

Supplied by CMU.

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:
```


Supplied by CMU.

“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?**

Supplied by CMU.

"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;
}
```


Supplied by CMU.

"For" Loop → While Loop

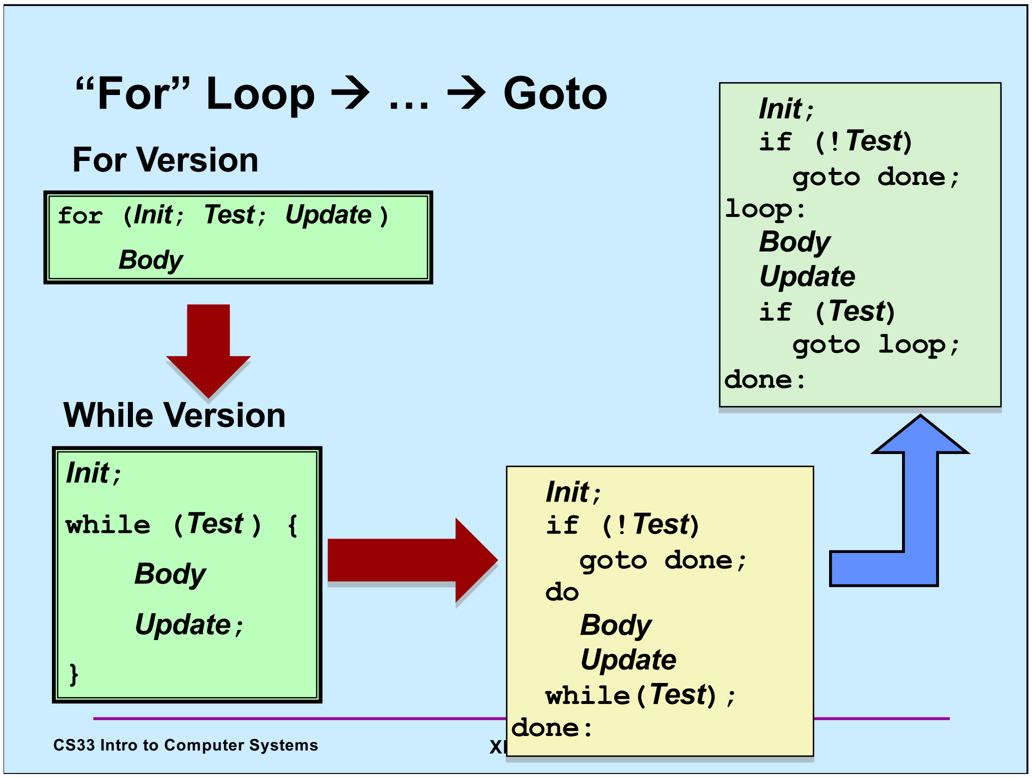
For Version

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

While Version

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


Supplied by CMU.



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;                    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;
}
```


Supplied by CMU.