Machine-Level Programming II: Arithmetic Ops and Control Flow

CS 105: Computer Systems
Lecture 06

Melissa O'Neill

February 9, 2026

Lecture 06 Learning Goals

- **Understand common arithmetic/logical integer operations of the x86-64**
  - E.g., `leaq`, `addq`, `andq`…
- **Describe the four condition codes and explain how they can be set and used**
  - Set implicitly vs. explicitly
  - Conditional control flow
- **Practice interpreting assembly code and its impact on state of memory and registers**

Address Computation Instruction

`leaq` *Src*, *Dst*

- *Src* is address mode expression; i.e. $D(R_1, R_2, S)$
- Set *Dst* to address denoted by expression $D + v_{R1} + S*v_{R2}$

Uses

- Computing addresses without a memory reference
  - E.g., translation of `p = &x[i];`
- Computing arithmetic expressions of the form x + k*y
  - k = 1, 2, 4, or 8

```
long m12(long x)
{
  return x*12;
}
```

```
movq $12, %rax
imult %rdi, %rax
ret
```

Multiplication is slower!

Compiler produces

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```


Examples: `leaq` vs. `movq`

`leaq src, %rax` vs. `movq src, %rax`

	src	dest %rax becomes:
leaq movq	(%rdi)	
leaq movq	8(%rdi)	
leaq movq	(%rdi,%rsi,2)	
leaq movq	%rdi	

Registers

%rdi	0xF000
%rsi	0x8

Memory

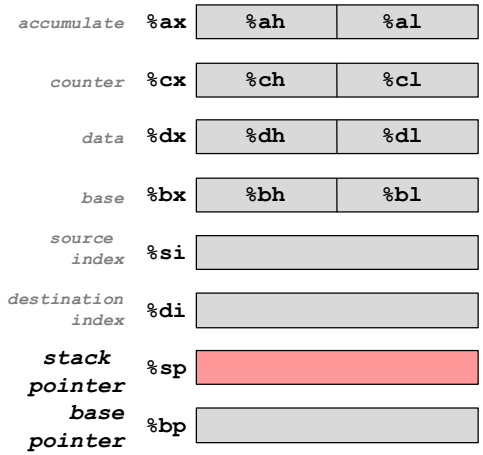
0xF010	0xBCDEF
0xF008	0x6789A
0xF000	0x12345

Some History: 8086 16 bit registers

Naming origin
(mostly obsolete)

```
move word:
movw %ax %bx

move byte:
movb %al %bl
```

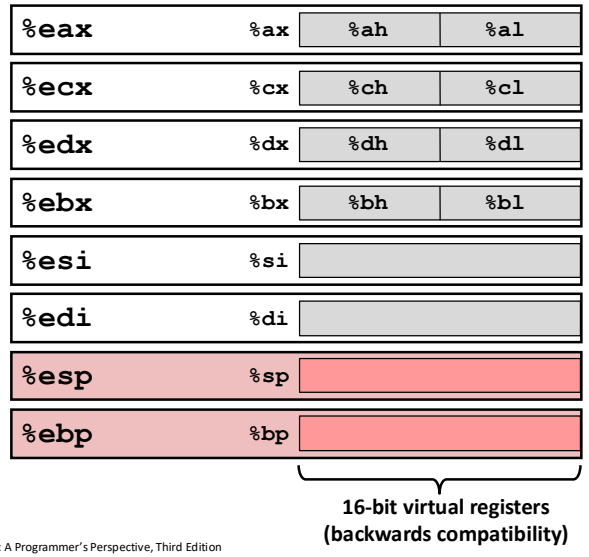


Some History: 32 bit Registers

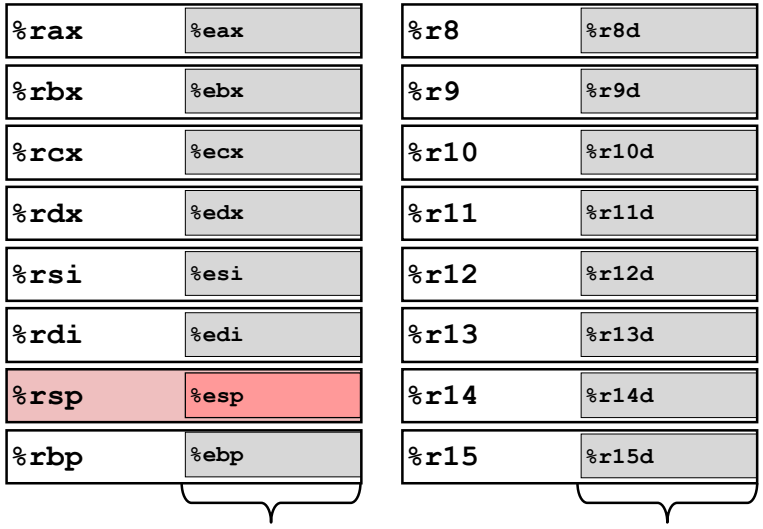
32 bits
```
move double word:
movl %eax %ebx
```

16 bits
```
move word:
movw %ax %bx

move byte:
movb %al %bl
```



x86-64 Integer Registers (64 bits)



movl zeroes out top 32 bits!

Referencing registers (64 bit processor)

- `movq %rax %rbx` // move the ***quad*** word (long-8 byte) in register %rax to %rbx
- `movl %eax %ebx` // move the ***double*** word (int-4 byte) in register %eax to %ebx
- `movw %ax %bx` // move the ***word*** (short-2 byte) in register %ax to %bx
- `movb %al %bl` // move the ***byte*** (char-1 byte) in register %al to %bl

Common Arithmetic/Logical Operations

- **Two Operand Instructions:**

Instruction Format		*Computation*
`addq`	*Src,Dest*	Dest = Dest + Src
`subq`	*Src,Dest*	Dest = Dest – Src
`imulq`	*Src,Dest*	Dest = Dest * Src
`salq`	*Src,Dest*	Dest = Dest << Src
`sarq`	*Src,Dest*	Dest = Dest >> Src
`shrq`	*Src,Dest*	Dest = Dest >> Src
`xorq`	*Src,Dest*	Dest = Dest ^ Src
`andq`	*Src,Dest*	Dest = Dest & Src
`orq`	*Src,Dest*	Dest = Dest | Src

Variations of `addq`:
`addl` for int
`addw` for short
`addb` for byte

(*Arithmetic*)
(*Logical*)

No distinction between signed and unsigned…

Common Arithmetic/Logical Operations

- **One Operand Instructions**

Instruction Format		*Computation*
`incq`	*Dest*	*Dest = Dest + 1*
`decq`	*Dest*	*Dest = Dest – 1*
`negq`	*Dest*	*Dest = – Dest*
`notq`	*Dest*	*Dest = ~Dest*

- **See book for more instructions**

Example: Arithmetic Expressions

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

```
arith:
  leaq    (%rdi,%rsi), %rax    # t1
  addq    %rdx, %rax           # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

Register	Values
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	~~t1~~ t2

Example: Arithmetic Expressions

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

```
arith:
  leaq    (%rdi,%rsi), %rax    # t1
  addq    %rdx, %rax           # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx             # t4
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

Register	Values
%rdi	Argument x
%rsi	Argument y
%rdx	~~Argument z~~ ~~3y~~ t4
%rax	~~t1~~ t2

Example: Arithmetic Expressions

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

```
arith:
  leaq    (%rdi,%rsi), %rax    # t1
  addq    %rdx, %rax           # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx             # t4
  leaq    4(%rdi,%rdx), %rcx   # t5
  imulq   %rcx, %rax
  ret
```

Register	Values
%rdi	Argument x
%rsi	Argument y
%rdx	~~Argument z~~ ~~3y~~ t4
%rax	~~t1~~ t2
%rcx	t5

Example: Arithmetic Expressions

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

```
arith:
  leaq    (%rdi,%rsi), %rax    # t1
  addq    %rdx, %rax           # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx             # t4
  leaq    4(%rdi,%rdx), %rcx   # t5
  imulq   %rcx, %rax           # rval
  ret
```

Register	Values
%rdi	Argument x
%rsi	Argument y
%rdx	~~Argument z~~ ~~3y~~ t4
%rax	~~t1~~ ~~t2~~ rval
%rcx	t5

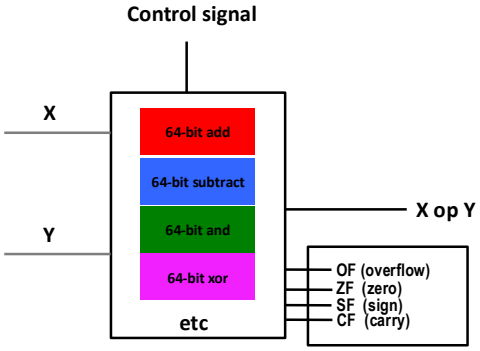
Exercise

What are in the registers right before the `ret` instruction executes?
(i.e., imagine a breakpoint is set at the `ret`)

```
arith:
1  leaq    (%rdi,%rsi,2), %rax
2  addq    %rdx, %rax
3  leaq    (,%rsi,4), %rdx
4  salq    $2, %rdx
5  ret
```

Register	*Value*
%rdi	5
%rsi	3
%rdx	7
%rax	

Arithmetic/logical ops performed on ALU



Most arithmetic/logical ops set condition codes

Setting Condition Codes

- **Single bits of the CPU's `flags` register**
- **Set**
  - Implicitly – as a result of an arithmetic/logical operations
  - Explicitly
    - Setting by ***Compare*** Instruction
      E.g., `cmpq b,a` like computing `a-b` without setting destination
    - Setting by ***Test*** instruction
      `testq b,a` like computing `a&b` without setting destination
- **Use**
  - Save (e.g., `sete`)
  - Jump (e.g., `je`) and conditional moves (e.g., `cmove`)

Condition codes example

Suppose we have a 4 bit word size and we add 0110 and 1101.

Result is 0011

```
  0110
+ 1101
 10011
```

=0000?

Carry Flag CF: 1
Zero Flag ZF: 0
Sign Flag SF:
Overflow Flag OF:

CF Interpretation for unsigned:
CF=0 result is true sum
CF=1 result is not true sum

Example above: 6+13=3

CF Interpretation for signed: ***not meaningful***

Condition codes example

Suppose we have a 4 bit word size and we add 0110 and 1101.

Result is 0011

```
  0110
+ 1101
 10011
```

Carry Flag CF: 1
Zero Flag ZF: 0
Sign Flag SF: 0
Overflow Flag OF:

SF Interpretation for unsigned: ***not meaningful***

SF Interpretation for signed:
SF=0 result non-negative
SF=1 result negative

Condition codes example

Suppose we have a 4 bit word size and we add 0110 and 1101.

Result is 0011

```
  0110
+ 1101
 10011
```

Carry Flag CF: 1
Zero Flag ZF: 0
Sign Flag SF: 0
Overflow Flag OF: 0

OF Interpretation for unsigned: ***not meaningful***

OF Interpretation for signed
OF = 0 result is true sum
OF = 1 result is not true sum

if input msb bits are equal but not equal to the msb of output

Example above: 6 + -3 = 3

Summary: Condition codes for addition

- CF: Carry Flag Detects unsigned overflow
- ZF: Zero Flag Detects result equal to zero
- SF: Sign Flag Detects signed negative
- OF: Overflow Flag Detects signed overflow

Result not equal real sum for, respectively, unsigned and two's complement

- *Some condition codes are not meaningful for some operations*

Exercise:

Compute the carry and overflow condition flags

```
  1001        0111        0110
 +1000       + 0111      +1101
 10001       01110       10011
```

CF= CF= CF=
OF= OF= OF=

Which sums not correct?

Condition codes interpretation after cmpq

`cmpq b,a` computing a−b without setting destination

Condition	Description
ZF	Equal / Zero: a==b
~ZF	Not Equal / Not Zero: a!=b
SF	Negative
~SF	Nonnegative
~ (SF^OF) &~ZF	Greater (Signed): a>b
~ (SF^OF)	Greater or Equal (Signed): a≥b
(SF^OF)	Less (Signed): a<b
(SF^OF) |ZF	Less or Equal (Signed): a≤b
~CF&~ZF	Above (unsigned): a>b
~CF	Above or Equal (unsigned) a≥b
CF	Below (unsigned): a<b
CF|ZF	Below or Equal (unsigned) a≤b

Note: result of a-b may not be the true sum. (The sign of a-b does not tell us if a≤b or a>b.)

a≥b? (SF==OF)

a-b≥0 and result is true sum

a-b<0 but result is not true sum

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)
jae	~CF	Above or Equal (unsigned) a≥b
jb	CF	Below (unsigned)
jbe	CF|ZF	Below or Equal (unsigned) a≤b

Jump instructions

```
long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

compute x-y without setting destination

```
absdiff:
   cmpq     %rsi, %rdi   # x:y
   jle      .L4          # jump if x <= y
   movq     %rdi, %rax
   subq     %rsi, %rax
   ret
.L4:                     # x <= y
   movq     %rsi, %rax
   subq     %rdi, %rax
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

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;
  }
  …
```

```
switch_eg:
    …
    cmpq     $6, %rdi    # x:6
    ja       .L8         # jump if x >6
    jmp      *.L4(,%rdi,8)
```

Indirect jump

Jump table

```
.section  .rodata
  .align 8
.L4:
  .quad   .L8  # x = 0
  .quad   .L3  # x = 1
  .quad   .L5  # x = 2
  .quad   .L9  # x = 3
  .quad   .L8  # x = 4
  .quad   .L7  # x = 5
  .quad   .L7  # x = 6
```


Alternative approach to `absdiff`

previous

```
long absdiff(long x, long y){
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
   cmpq     %rsi, %rdi
   jle      .L4
   movq     %rdi, %rax
   subq     %rsi, %rax
   ret
.L4:                     # x <= y
   movq     %rsi, %rax
   subq     %rdi, %rax
   ret
```

alternative

```
long absdiff(long x, long y){
  long result = x-y;
  long altResult = y-x;
  if (x <= y)
    result = altResult;
  return result;
}
```

```
absdiff:
   movq     %rdi, %rax
   subq     %rsi, %rax
   movq     %rsi, %rdx
   subq     %rdi, %rdx
   cmpq     %rsi, %rdi
   cmovle   %rdx, %rax
```

Conditional move