x86-64 Programming II

CSE 351 Spring 2022

Instructor:
Ruth Anderson

Teaching Assistants:
Melissa Birchfield
Jacob Christy
Alena Dickmann
Kyrie Dowling
Ellis Haker
Maggie Jiang
Diya Joy
Anirudh Kumar
Jim Limprasert
Armin Magness
Hamsa Shankar
Dara Stotland
Jeffery Tian
Assaf Vayner
Tom Wu
Angela Xu
Effie Zheng

http://xkcd.com/99/

Relevant Course Information

- hw7 due TONIGHT (4/15) @ 11:59 pm
- Lab 1b, due Monday 4/18
  - Submit `aisle_manager.c`, `store_client.c`, and `lab1Bsynthesis.txt`
- Lab 2 (x86-64) coming soon!
  - Learn to read x86-64 assembly and use GDB
- Midterm:
  - Released Mon 5/2 11:59pm, due Wed 5/4 11:59pm
  - Take home, Individual but some discussion permitted
  - More details later

Extra Credit

- All labs starting with Lab 2 have extra credit portions
  - These are meant to be fun extensions to the labs
- Extra credit points *don't* affect your lab grades
  - From the course policies: "they will be accumulated over the course and will be used to bump up borderline grades at the end of the quarter."
  - Make sure you finish the rest of the lab before attempting any extra credit

Example of Basic Addressing Modes

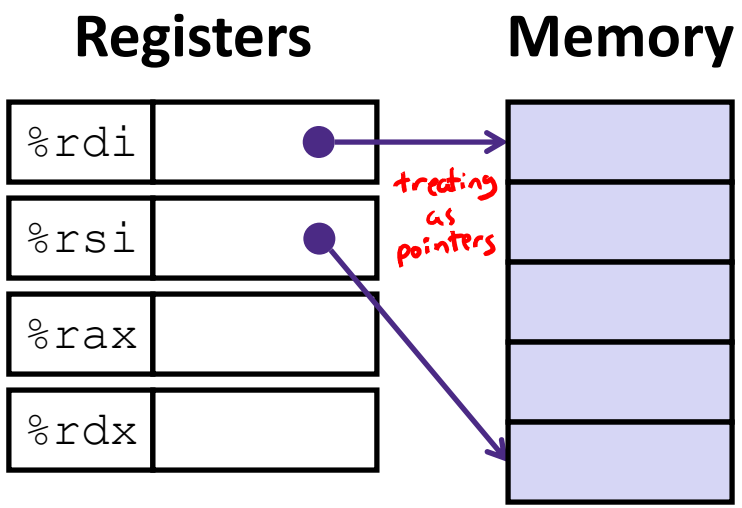
```
void swap(long* xp, long* yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movq   (%rdi), %rax
    movq   (%rsi), %rdx
    movq   %rdx, (%rdi)
    movq   %rax, (%rsi)
    ret
```

Compiler Explorer:
https://godbolt.org/z/zc4Pcq

Understanding swap()

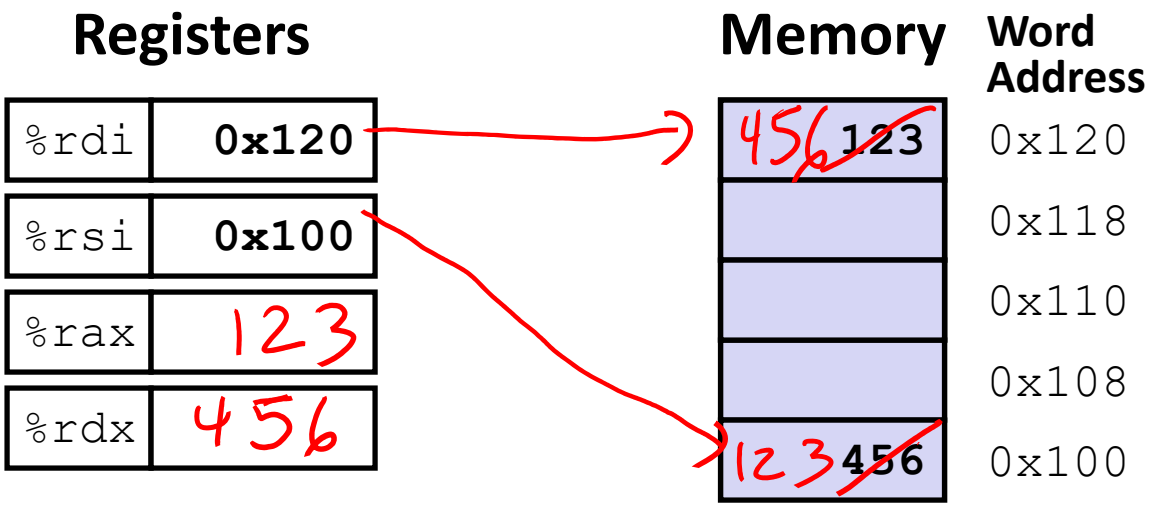
```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```




```
swap:
    movq   (%rdi), %rax
    movq   (%rsi), %rdx
    movq   %rdx, (%rdi)
    movq   %rax, (%rsi)
    ret
```

Register		Variable
%rdi	⇔	xp
%rsi	⇔	yp
%rax	⇔	t0
%rdx	⇔	t1

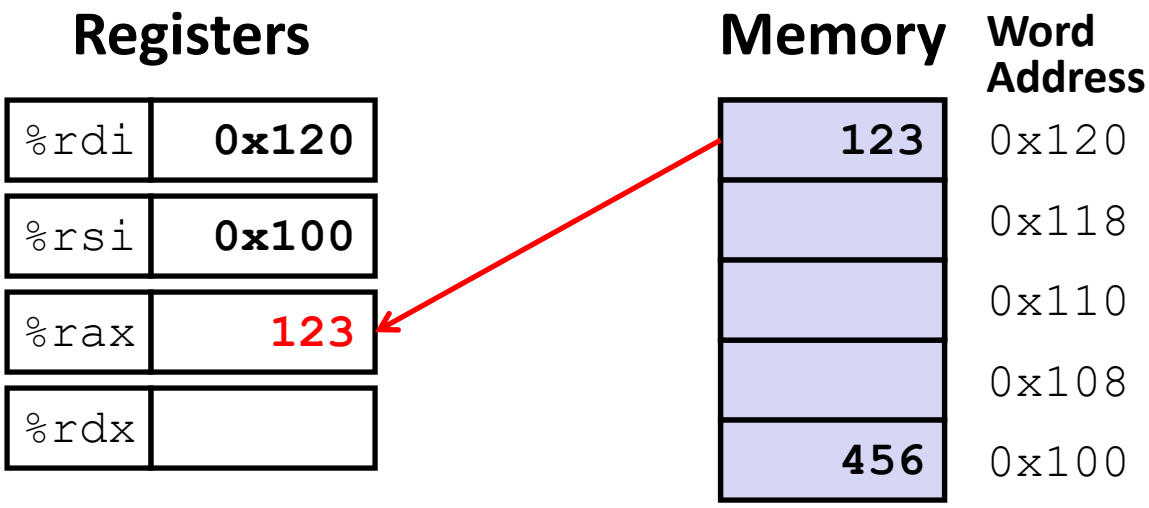
Understanding swap()




```
swap:
    movq    (%rdi), %rax    #  t0 = *xp
    movq    (%rsi), %rdx    #  t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

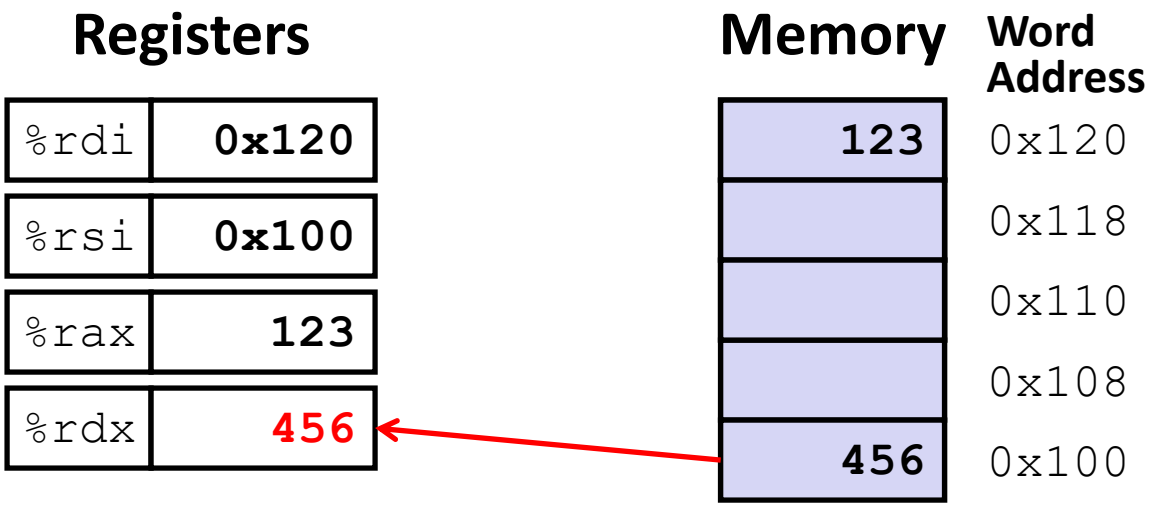
src → dest

Understanding swap()



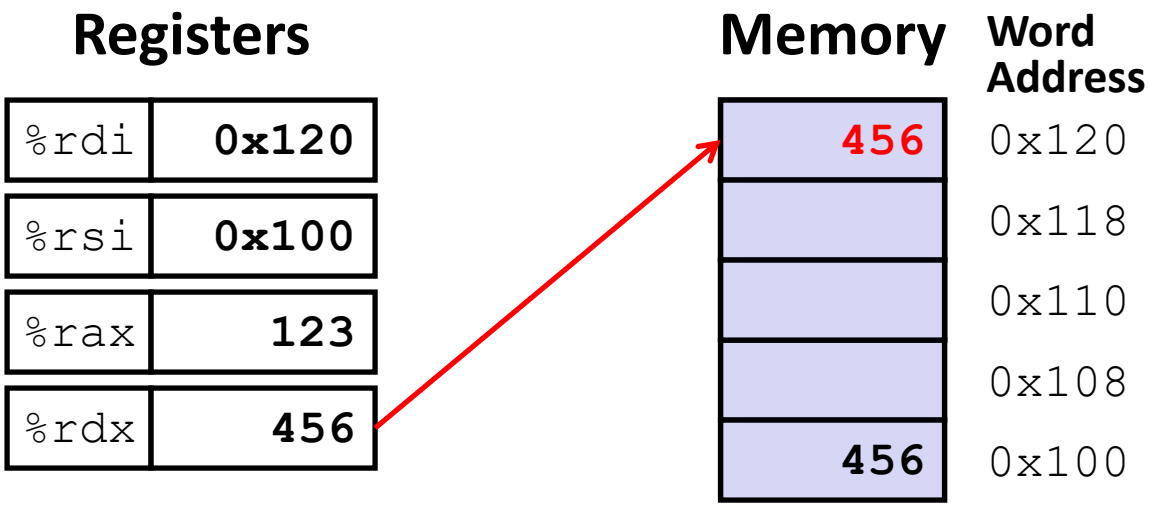

```
swap:
    movq   (%rdi), %rax   #  t0 = *xp
    movq   (%rsi), %rdx   #  t1 = *yp
    movq   %rdx, (%rdi)   # *xp =  t1
    movq   %rax, (%rsi)   # *yp =  t0
    ret
```

Understanding swap()



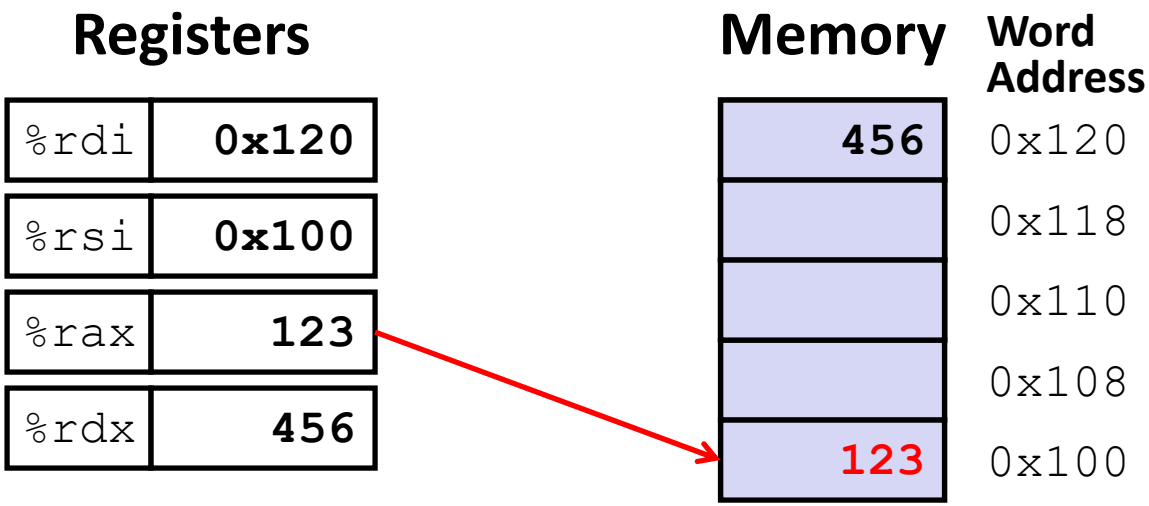

```
swap:
    movq   (%rdi), %rax   #  t0 = *xp
    movq   (%rsi), %rdx   #  t1 = *yp
    movq   %rdx, (%rdi)   # *xp =  t1
    movq   %rax, (%rsi)   # *yp =  t0
    ret
```

Understanding swap()




```
swap:
    movq   (%rdi), %rax   #  t0 = *xp
    movq   (%rsi), %rdx   #  t1 = *yp
    movq   %rdx, (%rdi)   # *xp =  t1
    movq   %rax, (%rsi)   # *yp =  t0
    ret
```

Understanding swap()




```
swap:
    movq   (%rdi), %rax   #  t0 = *xp
    movq   (%rsi), %rdx   #  t1 = *yp
    movq   %rdx, (%rdi)   # *xp =  t1
    movq   %rax, (%rsi)   # *yp =  t0
    ret
```

Memory Addressing Modes: Basic

- **Indirect:** `(R)` Mem[Reg[R]]
  - Data in register `R` specifies the memory address
  - Like pointer dereference in C
  - Example: `movq (%rcx), %rax`

- **Displacement:** `D(R)` Mem[Reg[R]+D]
  - Data in register `R` specifies the *start* of some memory region
  - Constant displacement `D` specifies the offset from that address
  - Example: `movq 8(%rbp), %rdx`

Complete Memory Addressing Modes

ar[i] ⟷ *(ar + i) → Mem[ar + i * sizeof(data type)]

- **General:**
  - `D(Rb,Ri,S)` Mem[Reg[Rb]+Reg[Ri]*S+D]
    - `Rb`: Base register (any register)
    - `Ri`: Index register (any register except `%rsp`)
    - `S`: Scale factor (1, 2, 4, 8) – *why these numbers?* data type widths
    - `D`: Constant displacement value (a.k.a. immediate)

- **Special cases** (see CSPP Figure 3.3 on p.181)
  - `D(Rb,Ri)` Mem[Reg[Rb]+Reg[Ri]+D] `(S=1)`
  - `(Rb,Ri,S)` Mem[Reg[Rb]+Reg[Ri]*S] `(D=0)`
  - `(Rb,Ri)` Mem[Reg[Rb]+Reg[Ri]] `(S=1,D=0)`
  - `(,Ri,S)` Mem[Reg[Ri]*S] `(Rb=0,D=0)`

↑ so reg name not interpreted as Rb

default values:
S = 1
D = 0
Reg[Rb] = 0
Reg[Ri] = 0

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

D(Rb,Ri,S) →
Mem[Reg[Rb]+Reg[Ri]*S+D]
└ ignore the memory access for now

Expression	Address Computation	Address (8 bytes wide)
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x0100	0xf100
(%rdx,%rcx,4)	0xf000 + (0x100 * 4)	0xf400
0x80(,%rdx,2)	f000 * 2 = 1 1110 0000 0000 = 0x1e000	0x1e080

Reading Review

- Terminology:
  - Address Computation Instruction (`lea`)
  - Condition codes: Carry Flag (CF), Zero Flag (ZF), Sign Flag (SF), and Overflow Flag (OF)
  - Test (`test`) and compare (`cmp`) assembly instructions
  - Jump (`j*`) and set (`set*`) families of assembly instructions

Review Questions

- Which of the following x86-64 instructions correctly calculates: `%rax = 9 * %rdi`

  no memory access, so must be lea

  $S \in \{1, 2, 4, 8\}$

  A. `leaq (,%rdi,9), %rax` — invalid syntax

  B. ~~`movq`~~ `(,%rdi,9), %rax` — invalid syntax

  C. `leaq (%rdi,%rdi,8), %rax` — %rax = 9 * %rdi

  D. ~~`movq`~~ `(%rdi,%rdi,8), %rax` — %rax = Mem[9 * %rdi]

- If `%rsi` is 0x B0BACAFE 1EE7 F0 0D, what is its value after executing **`movswl %si, %esi`**?

  %esi — MSB of %si is a 1

  %si

  sign extension

  destination is 4 bytes

  source is 2 bytes

  0x 0000 0000 FFFF FOOD

  x86-64 rule when destination is 32 bits

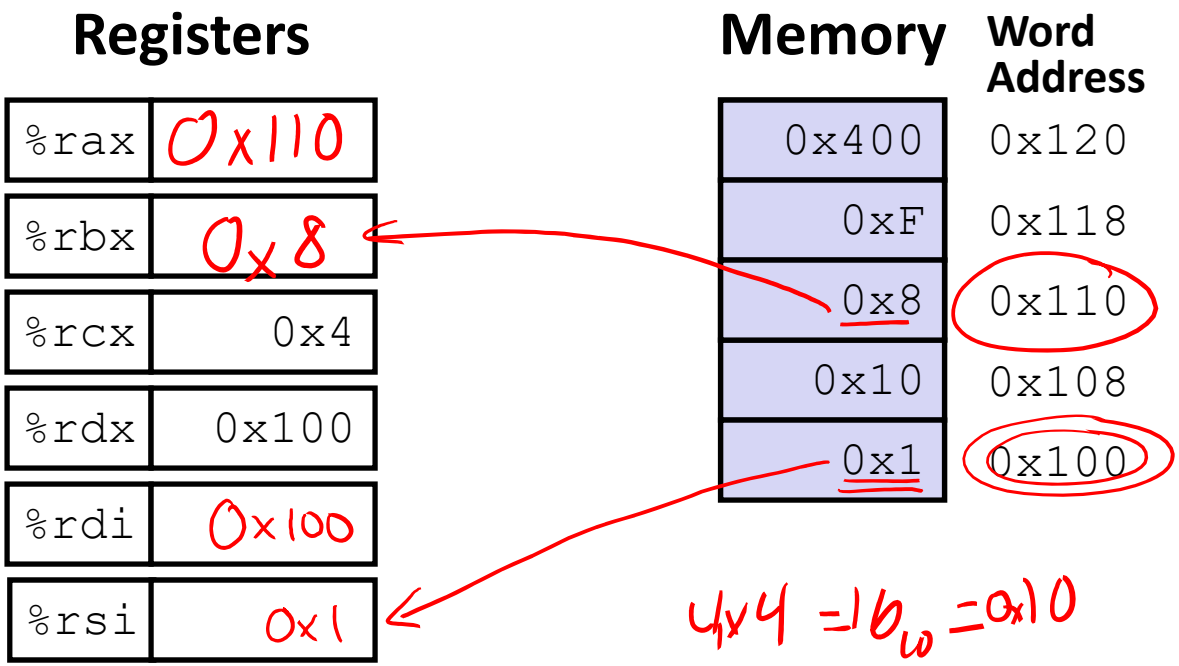
  sign extension

  original data

Address Computation Instruction

- `leaq src, dst`
  - "`lea`" stands for *load effective address*
  - `src` is address expression (any of the formats we've seen)
  - `dst` is a register
  - Sets `dst` to the *address* computed by the `src` expression (does not go to memory! – it just does math)
  - Example: `leaq (%rdx,%rcx,4), %rax`
- Uses:
  - Computing addresses without a memory reference
    - *e.g.* translation of `p = &x[i];`
  - Computing arithmetic expressions of the form `x+k*i+d`
    - Though `k` can only be 1, 2, 4, or 8

Example: `lea` vs. `mov`




```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

0x100

→ 0x110 ("addr")
→ 0x8 (data)
→ 0x100 ("addr")
→ 0x1 (data)

`lea` – "It just does math"

Arithmetic Example

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rdx	3rd argument (z)

```
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;
}
```

← replaced by lea & shift

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

rax = x+y (t1)
rax = x+y+z (t2)
rdx = 3y
rdx = 48y (t4)

multiplying two variables

- Interesting Instructions
  - leaq: "address" computation
  - salq: shift
  - imulq: multiplication
    - Only used once!

Arithmetic Example

```
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;
}
```

Register	Use(s)
%rdi	x
%rsi	y
%rdx	z, t4
%rax	t1, t2, rval
%rcx	t5

limited registers means they often get reused!

```
arith:
  leaq    (%rdi,%rsi), %rax     # rax/t1    = x + y
  addq    %rdx, %rax            # rax/t2    = t1 + z
  leaq    (%rsi,%rsi,2), %rdx   # rdx       = 3 * y
  salq    $4, %rdx              # rdx/t4    = (3*y) * 16
  leaq    4(%rdi,%rdx), %rcx    # rcx/t5    = x + t4 + 4
  imulq   %rcx, %rax            # rax/rval  = t5 * t2
  ret
```

comment (AT&T syntax)

S ∈ {1,2,4,8}

Control Flow

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rax	return value

```
long max(long x, long y)
{
  long max;
  if (x > y) {
    max = x;
  } else {
    max = y;
  }
  return max;
}
```

```
max:
   ???
   movq    %rdi, %rax  # if case
   ???
   ???
   movq    %rsi, %rax  # else case
   ???
   ret
```

Control Flow

Register	Use(s)
%rdi	1st argument (x)
%rsi	2nd argument (y)
%rax	return value

```
long max(long x, long y)
{
  long max;
  if (x > y) {
    max = x;
  } else {
    max = y;
  }
  return max;
}
```

Conditional jump

Unconditional jump

```
max:
  if x <= y then jump to else     (if TRUE)
  movq    %rdi, %rax              (if FALSE)
  jump to done
else:
  movq    %rsi, %rax
done:
  ret
```

Conditionals and Control Flow

- Conditional branch/*jump*
  - Jump to somewhere else if some *condition* is true, otherwise execute next instruction
- Unconditional branch/*jump*
  - *Always* jump when you get to this instruction

- Together, they can implement most control flow constructs in high-level languages:
  - **`if`** (*condition*) **`then`** {…} **`else`** {…}
  - **`while`** (*condition*) {…}
  - **`do`** {…} **`while`** (*condition*)
  - **`for`** (*initialization*`;` *condition*`;` *iterative*) {…}
  - **`switch`** {…}

Summary

- **Memory Addressing Modes:** The addresses used for accessing memory in `mov` (and other) instructions can be computed in several different ways
  - *Base register*, *index register*, *scale factor*, and *displacement* map well to pointer arithmetic operations
- Control flow in x86 determined by Condition Codes