# x86-64 Programming I

CSE 351 Winter 2019

**Instructors:**

Max Willsey

Luis Ceze

**Teaching Assistants:**

Britt Henderson

Lukas Joswiak

Josie Lee

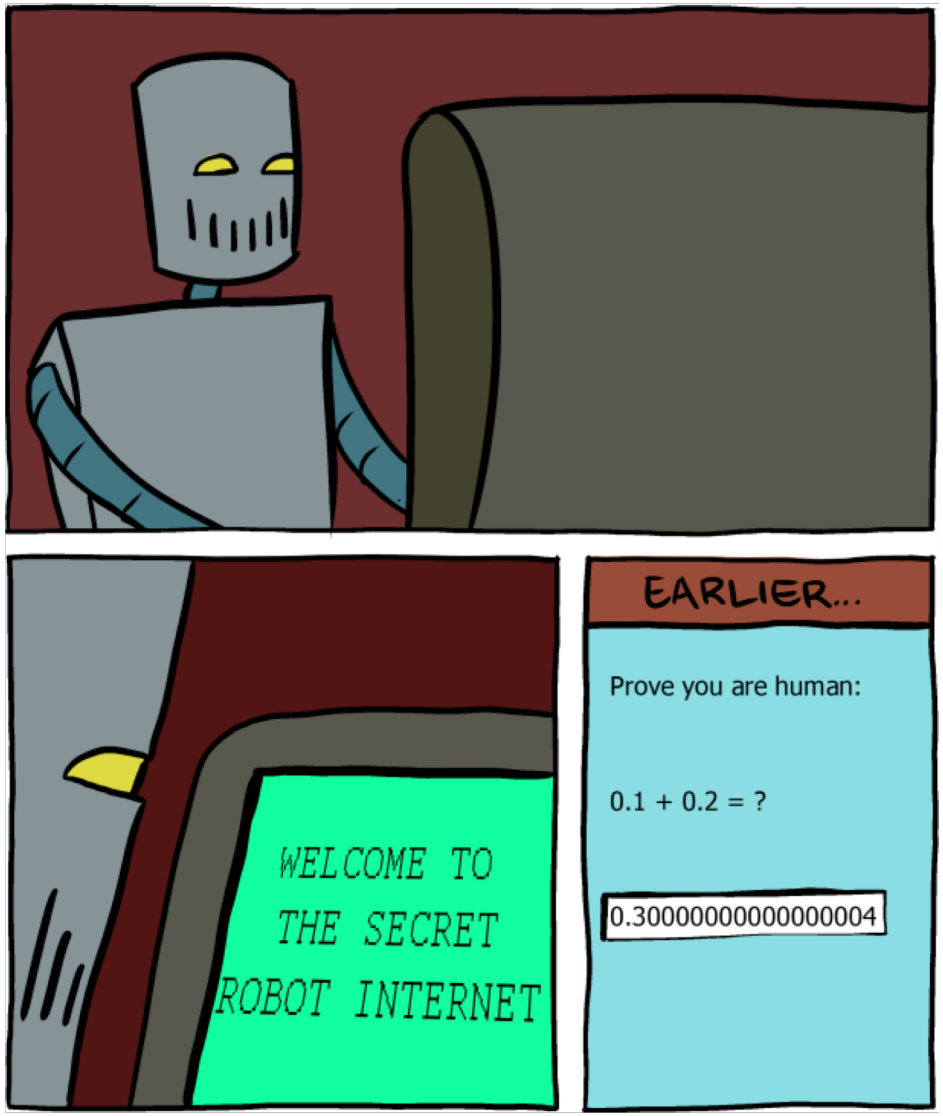
Wei Lin

Daniel Snitkovsky

Luis Vega

Kory Watson

Ivy Yu




http://www.smbc-comics.com/?id=2999

# Administrivia

- Lab 1b due tonight at 11:59 pm
  - You have *late day tokens* available
- Homework 2 due next Friday (Feb 1)
- Lab 2 (x86-64) released
  - Due on Feb 8

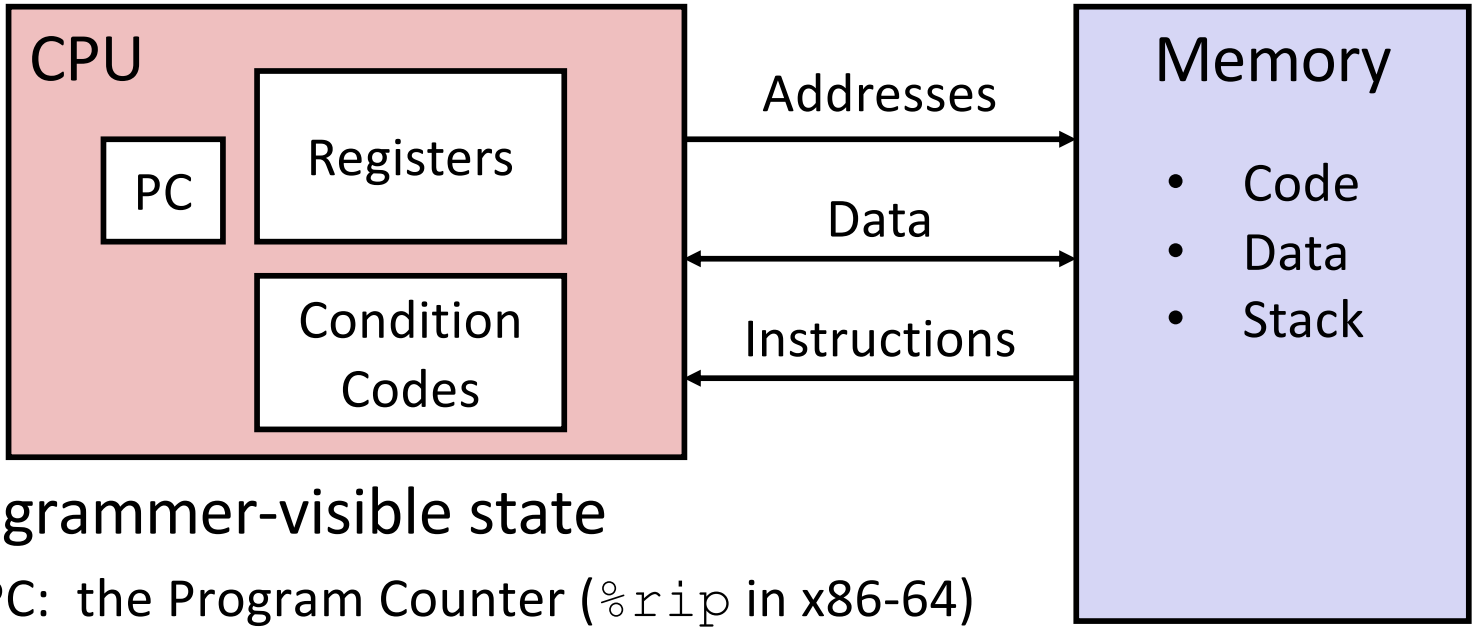
# Non-Compiling Code

- You get a zero on the assignment
  - No excuses – you have access to our grading environment
- Some leeway was given on Lab 1a, do not expect the same leniency moving forward

# Writing Assembly Code? In 2018???

- Chances are, you'll never write a program in assembly, but understanding assembly is the key to the machine-level execution model:
  - Behavior of programs in the presence of bugs
    - When high-level language model breaks down
  - Tuning program performance
    - Understand optimizations done/not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing systems software
    - What are the "states" of processes that the OS must manage
    - Using special units (timers, I/O co-processors, etc.) inside processor!
  - Fighting malicious software
    - Distributed software is in binary form

# Assembly Programmer's View




- Programmer-visible state
  - PC: the Program Counter (`%rip` in x86-64)
    - Address of next instruction
  - Named registers
    - Together in "register file"
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching
- Memory
  - Byte-addressable array
  - Code and user data
  - Includes *the Stack* (for supporting procedures)

# x86-64 Assembly "Data Types"

- Integral data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses
- Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
  - Different registers for those (*e.g.* `%xmm1`, `%ymm2`)
  - Come from *extensions to x86* (SSE, AVX, …)

Not covered In 351

- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
- Two common syntaxes
  - "AT&T": used by our course, slides, textbook, gnu tools, …
  - "Intel": used by Intel documentation, Intel tools, …
  - Must know which you're reading

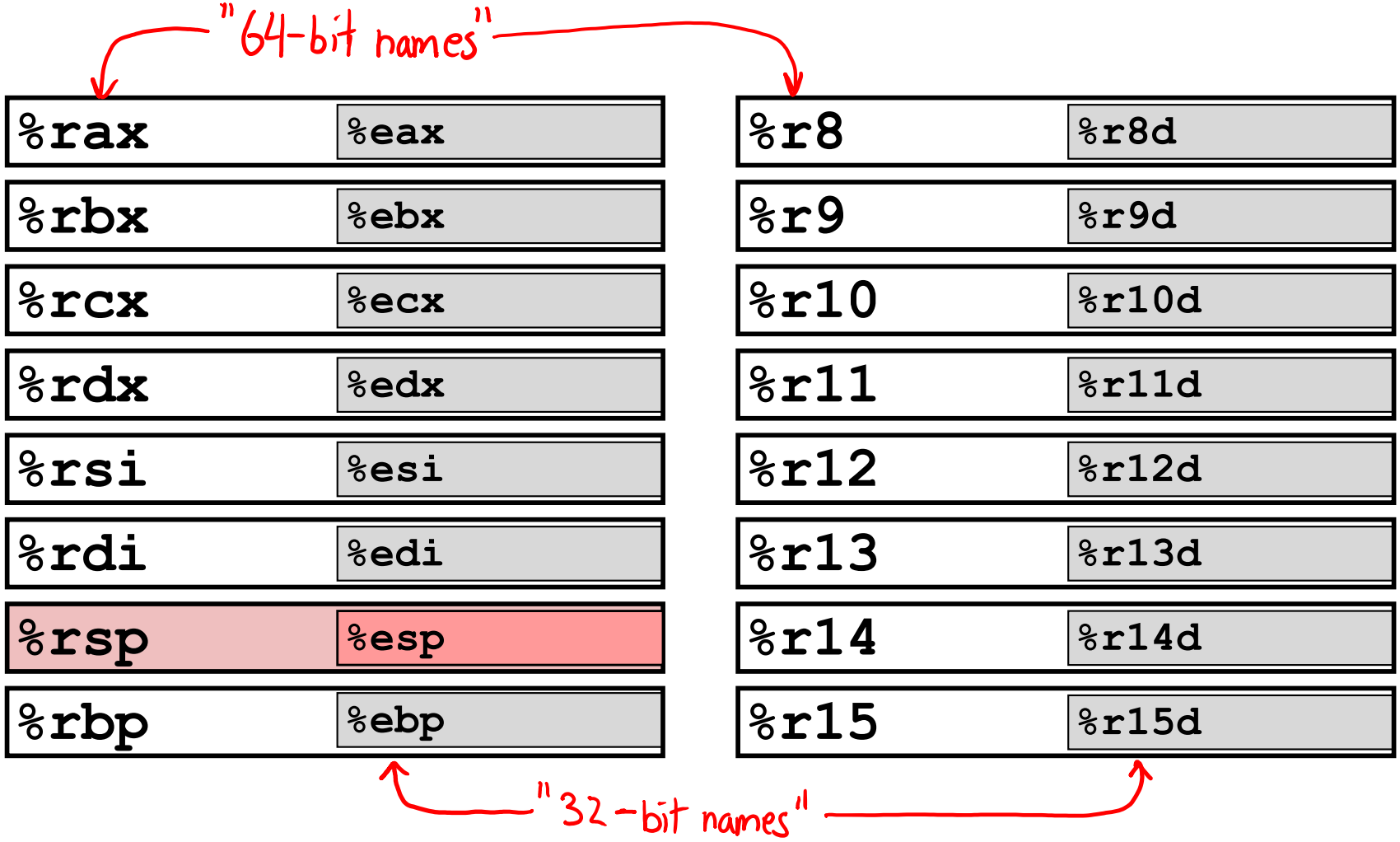
# What is a Register?

- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)

- Registers have *names*, not *addresses*
  - In assembly, they start with `%` (*e.g.* `%rsi`)

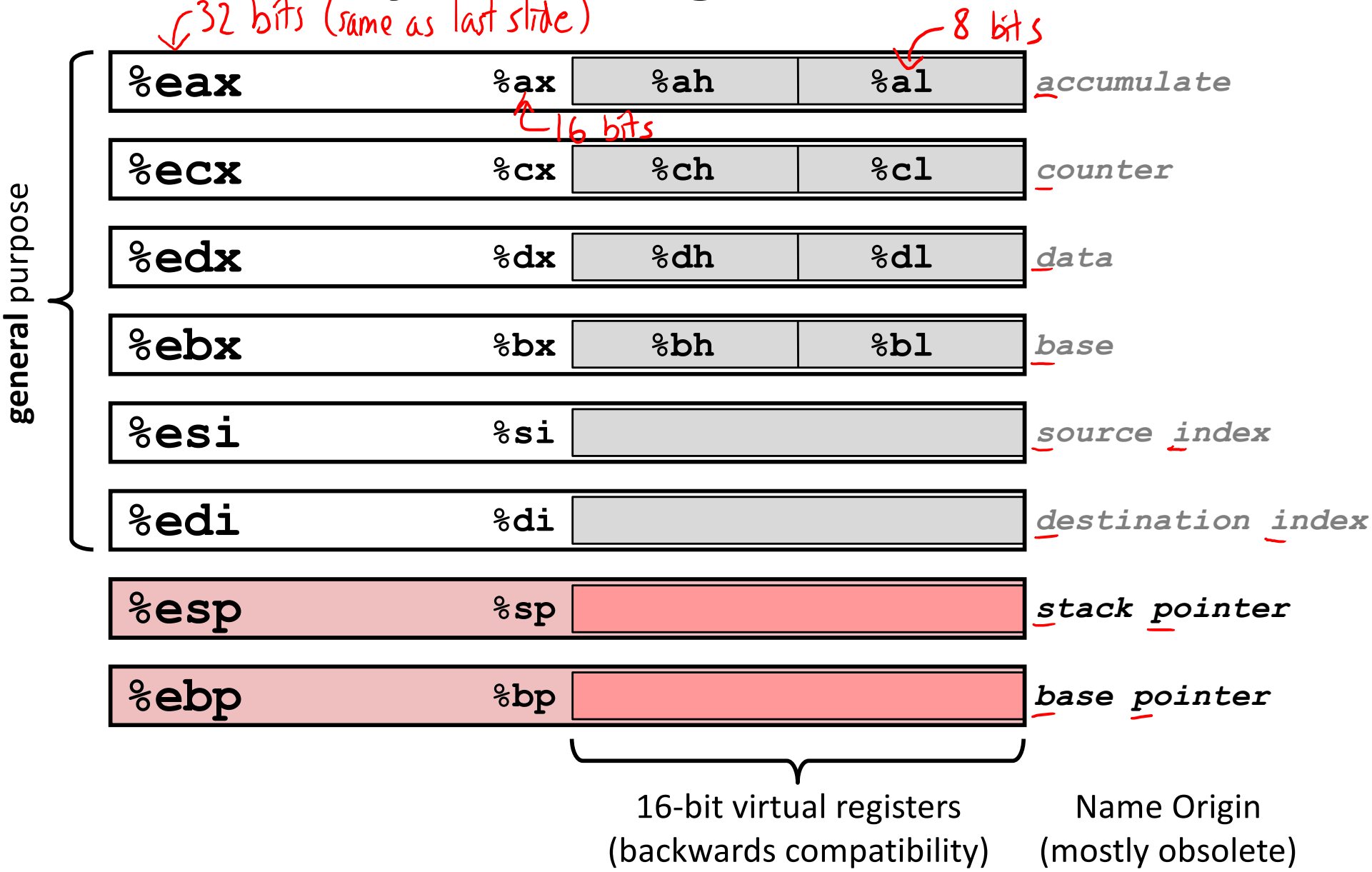
- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but *especially* x86 only 16 of them...

# x86-64 Integer Registers – 64 bits wide




- Can reference low-order 4 bytes (also low-order 2 & 1 bytes)

# Some History: IA32 Registers – 32 bits wide



# Memory vs. Registers

| Memory | | Registers |
|---|---|---|
| Addresses | **vs.** | Names |
| `0x7FFFD024C3DC` | | `%rdi` |
| Big | **vs.** | Small |
| ~ 8 GiB | | (16 x 8 B) = 128 B |
| Slow | **vs.** | Fast |
| ~50-100 ns !!! | | sub-nanosecond timescale |
| Dynamic | **vs.** | Static |
| Can "grow" as needed while program runs | | fixed number in hardware |

# Three Basic Kinds of Instructions

1) Transfer data between memory and register
   - *Load* data from memory into register
     - `%reg` = Mem[address]
   - *Store* register data into memory
     - Mem[address] = `%reg`

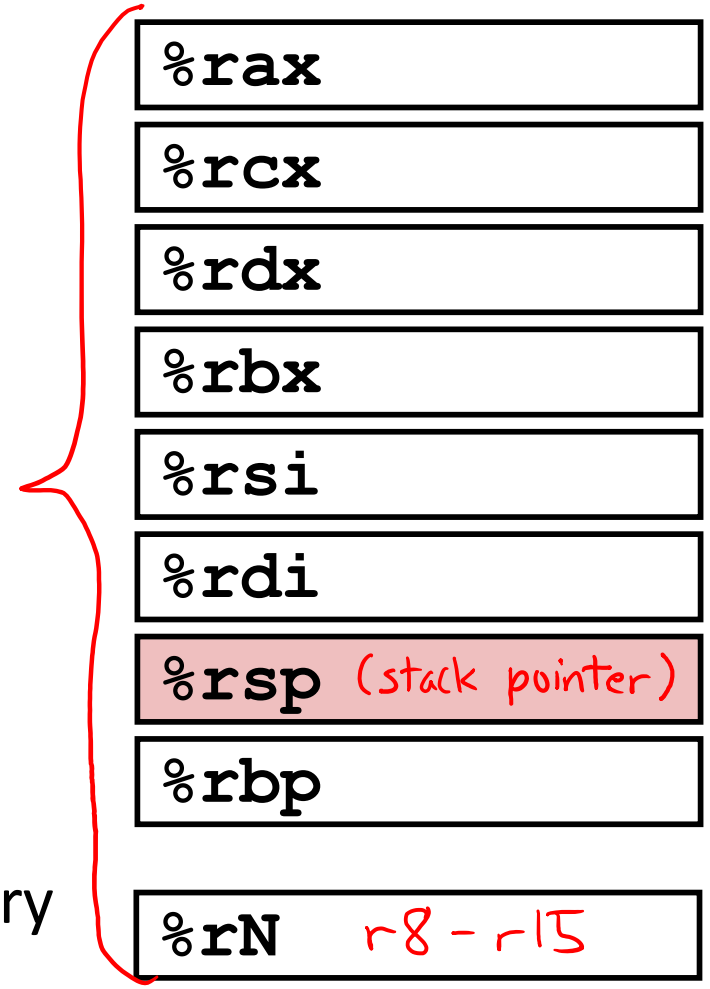
**Remember:** Memory is indexed just like an array of bytes!

2) Perform arithmetic operation on register or memory data
   - `c = a + b;`  `z = x << y;`  `i = h & g;`

3) Control flow: what instruction to execute next
   - Unconditional jumps to/from procedures
   - Conditional branches

# Operand types

- ***Immediate:*** Constant integer data
  - Examples: **`$0x400`**, **`$-533`** (hex, decimal)
  - Like C literal, but prefixed with **`'$'`**
  - Encoded with 1, 2, 4, or 8 bytes *depending on the instruction*
- ***Register:*** 1 of 16 integer registers
  - Examples: **`%rax`**, **`%r13`**
  - But **`%rsp`** reserved for special use
  - Others have special uses for particular instructions
- ***Memory:*** Consecutive bytes of memory at a computed address
  - Simplest example: **`(%rax)`** ← take data in %rax, treat as address, pull data at that address
  - Various other "address modes"



# x86-64 Introduction

- Data transfer instruction (`mov`)
- Arithmetic operations
- Memory addressing modes
  - `swap` example
- Address computation instruction (`lea`)

# Moving Data


- General form: `mov_ source, destination`
  - Missing letter (_) specifies size of operands
  - Note that due to backwards-compatible support for 8086 programs (16-bit machines!), "word" means 16 bits = 2 bytes in x86 instruction names
  - Lots of these in typical code

- `movb src, dst`
  - Move 1-byte "**b**yte"
- `movw src, dst`
  - Move 2-byte "**w**ord"
- `movl src, dst`
  - Move 4-byte "**l**ong word"
- `movq src, dst`
  - Move 8-byte "**q**uad word"

# movq Operand Combinations

x86 C

Imm ⟷ Constant
Reg ⟷ Variable
Mem ⟷ dereferencing a pointer

| | Source | Dest | Src, Dest | C Analog |
|---|---|---|---|---|
| movq | Imm | Reg | `movq $0x4, %rax` | `var_a = 0x4;` |
| | | Mem | `movq $-147, (%rax)` | `*p_a = -147;` |
| | Reg | Reg | `movq %rax, %rdx` | `var_d = var_a;` |
| | | Mem | `movq %rax, (%rdx)` | `*p_d = var_a;` |
| | Mem | Reg | `movq (%rax), %rdx` | `var_d = *p_a;` |

- *Cannot do memory-memory transfer with a single instruction*
  - How would you do it?

(1) Mem → Reg  movq (%rax), %rdx
(2) Reg → Mem  movq %rdx, (%rbx)

other ways to set to 0:
subq %rcx, %rcx
andq $0, %rcx
xorq %rcx, %rcx
imulq $0, %rcx

# Some Arithmetic Operations

- Binary (two-operand) Instructions:
  - **Maximum of one memory operand**
  - Beware argument order!
  - No distinction between signed and unsigned
    - Only arithmetic vs. logical shifts
  - How do you implement "r3 = r1 + r2"? (%rcx = %rax + %rbx)

Imm, Reg, or Mem

| Format | Computation | |
|---|---|---|
| **addq** src, dst | dst = dst + src | (dst += src) |
| **subq** src, dst | dst = dst – src | |
| **imulq** src, dst | dst = dst * src | signed mult |
| **sarq** src, dst | dst = dst >> src | **A**rithmetic |
| **shrq** src, dst | dst = dst >> src | **L**ogical |
| **shlq** src, dst | dst = dst << src | (same as salq) |
| **xorq** src, dst | dst = dst ^ src | |
| **andq** src, dst | dst = dst & src | |
| **orq** src, dst | dst = dst \| src | |

operation

operand size specifier (b, w, l, q)

① clear r3
② add r1 to r3 ⇒
③ add r2 to r3

```
movq $0, %rcx
addq %rax, %rcx
addq %rbx, %rcx
```

```
movq %rax, %rcx
addq %rbx, %rcx
```

# Some Arithmetic Operations

- Unary (one-operand) Instructions:

| Format | Computation | |
|---|---|---|
| **incq** *dst* | *dst = dst + 1* | increment |
| **decq** *dst* | *dst = dst – 1* | decrement |
| **negq** *dst* | *dst = –dst* | negate |
| **notq** *dst* | *dst = ~dst* | bitwise complement |

- See CSPP Section 3.5.5 for more instructions:
  `mulq`, `cqto`, `idivq`, `divq`

# Arithmetic Example

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

convention!

```
long simple_arith(long x, long y)
{
    long t1 = x + y;
    long t2 = t1 * 3;
    return t2;
}
```

don't actually need new variables!

```
    y += x;
    y *= 3;
    long r = y;
    return r;
```

must return in %rax

```
simple_arith:
    addq    %rdi, %rsi
    imulq   $3, %rsi
    movq    %rsi, %rax
    ret     # return
```

# Example of Basic Addressing Modes

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

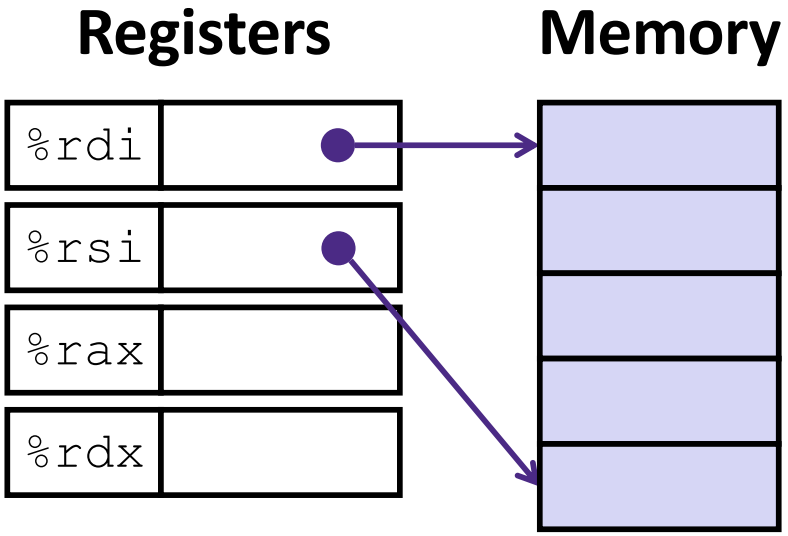
```
swap:
    movq   (%rdi), %rax
    movq   (%rsi), %rdx
    movq   %rdx, (%rdi)
    movq   %rax, (%rsi)
    ret
```

src , dst (AT&T syntax)

Mem operands

# Understanding swap()

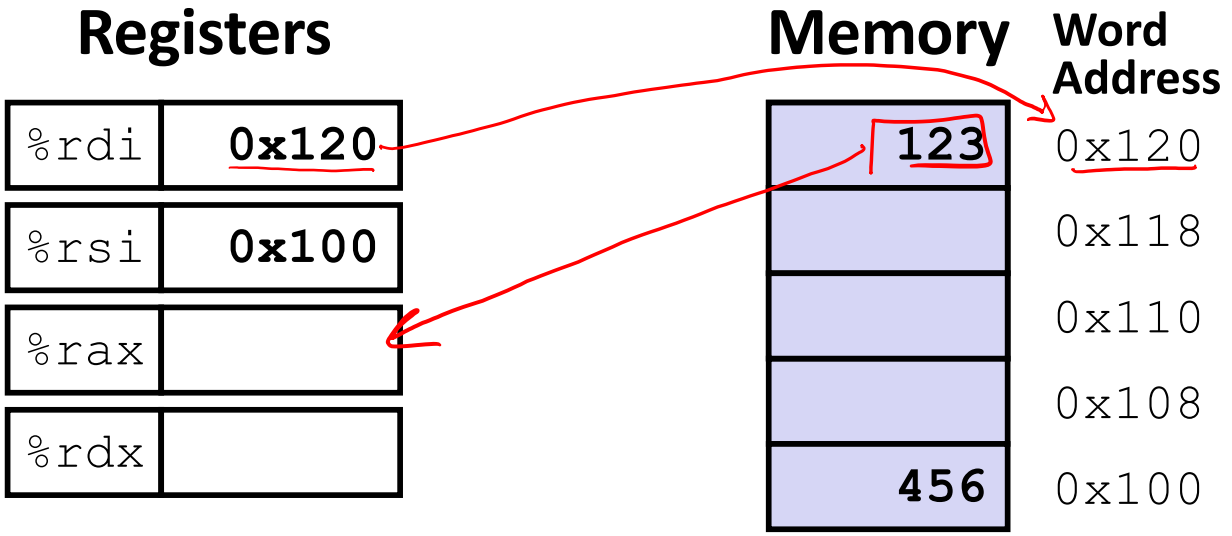
```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```




```
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()

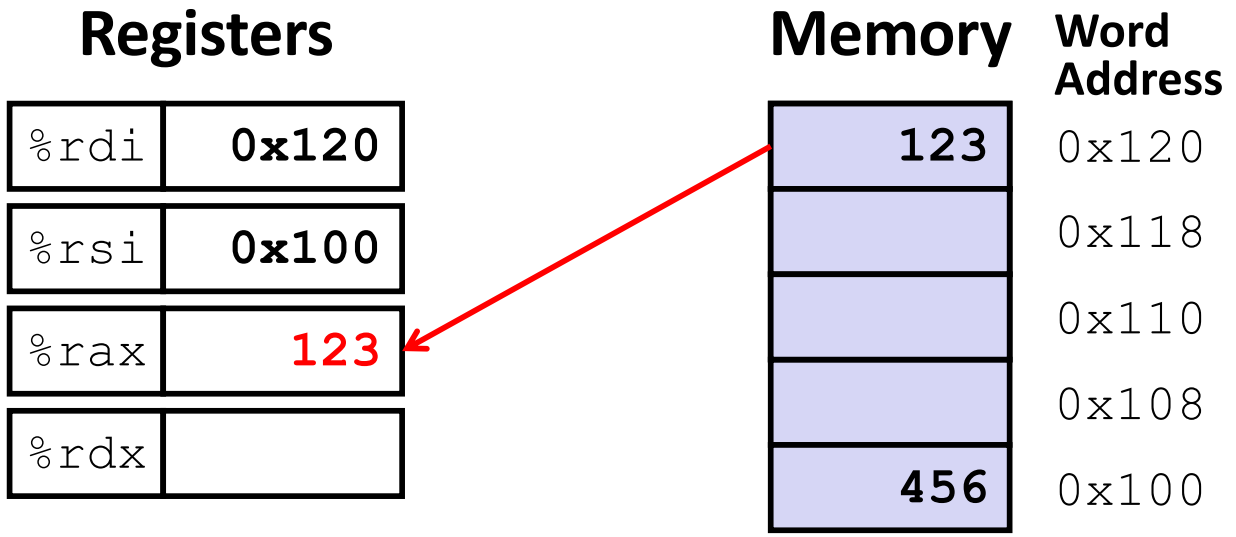



src dst

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

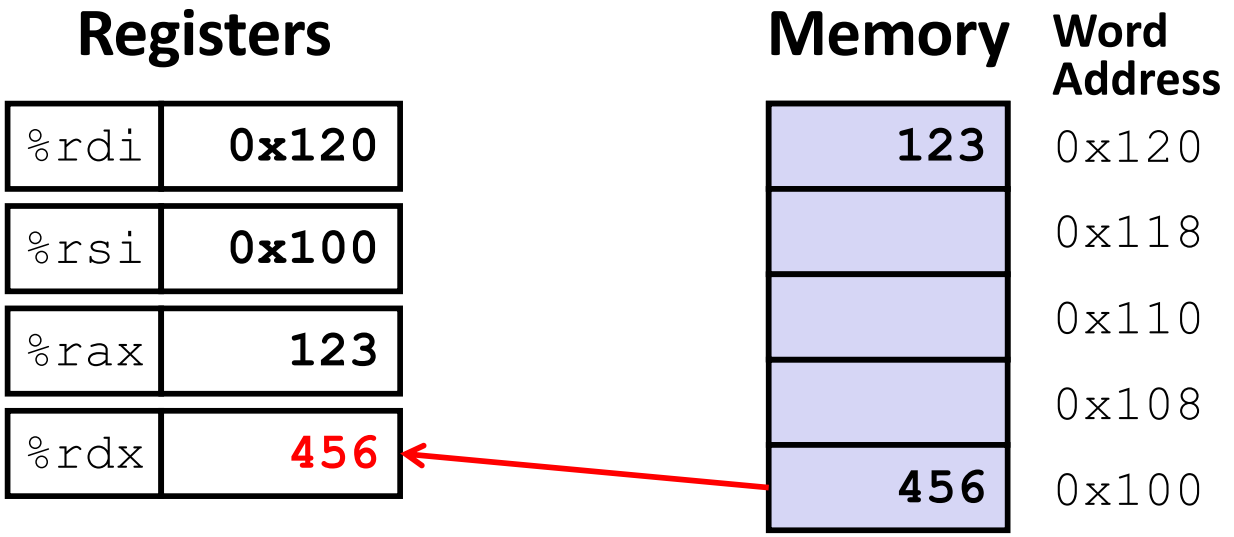
Comment

# Understanding swap()



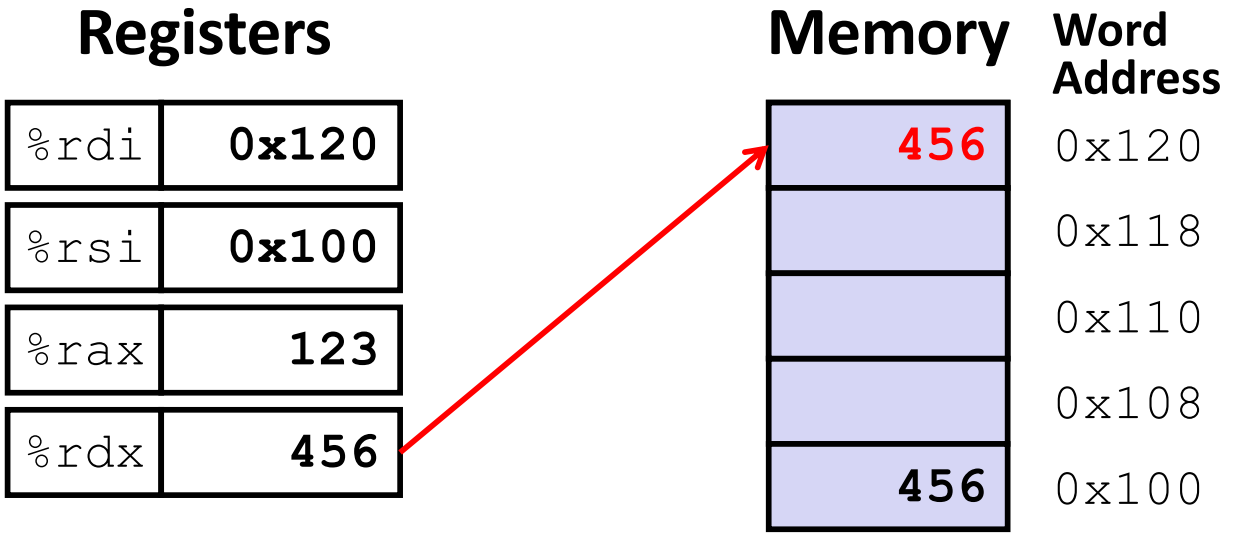

```
swap:
    movq   (%rdi), %rax   #  t0 = *xp
    movq   (%rsi), %rdx   #  t1 = *yp
    movq   %rdx, (%rdi)   # *xp =  t1
    movq   %rax, (%rsi)   # *yp =  t0
    ret
```

# Understanding swap()



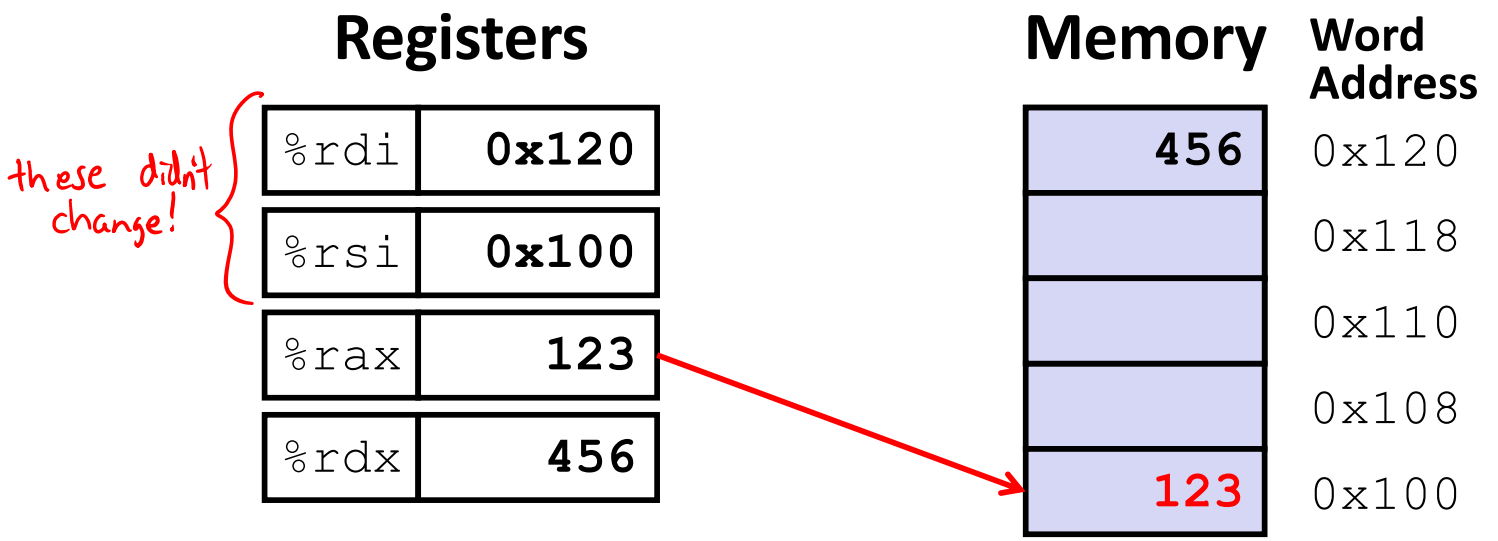

```
swap:
    movq   (%rdi), %rax   #  t0 = *xp
    movq   (%rsi), %rdx   #  t1 = *yp
    movq   %rdx, (%rdi)   # *xp =  t1
    movq   %rax, (%rsi)   # *yp =  t0
    ret
```

# 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(datatype)]

- **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

(if not specified) 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) | Reg[Rb]+D = 0xf000 + 0x8 | 0xf008 |
| (%rdx,%rcx) | Reg[Rb]+Reg[Ri]*1 | 0xf100 |
| (%rdx,%rcx,4) | *4 | 0xf400 |
| 0x80(,%rdx,2) | Reg[Ri]*2+0x80 | 0x1e080 |

0xf000*2
0xf000 << 1 = 0x1e000
1111 0000
1 1110 0000...0

# Summary

- There are 3 types of operands in x86-64
  - Immediate, Register, Memory
- There are 3 types of instructions in x86-64
  - Data transfer, Arithmetic, Control Flow

- **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