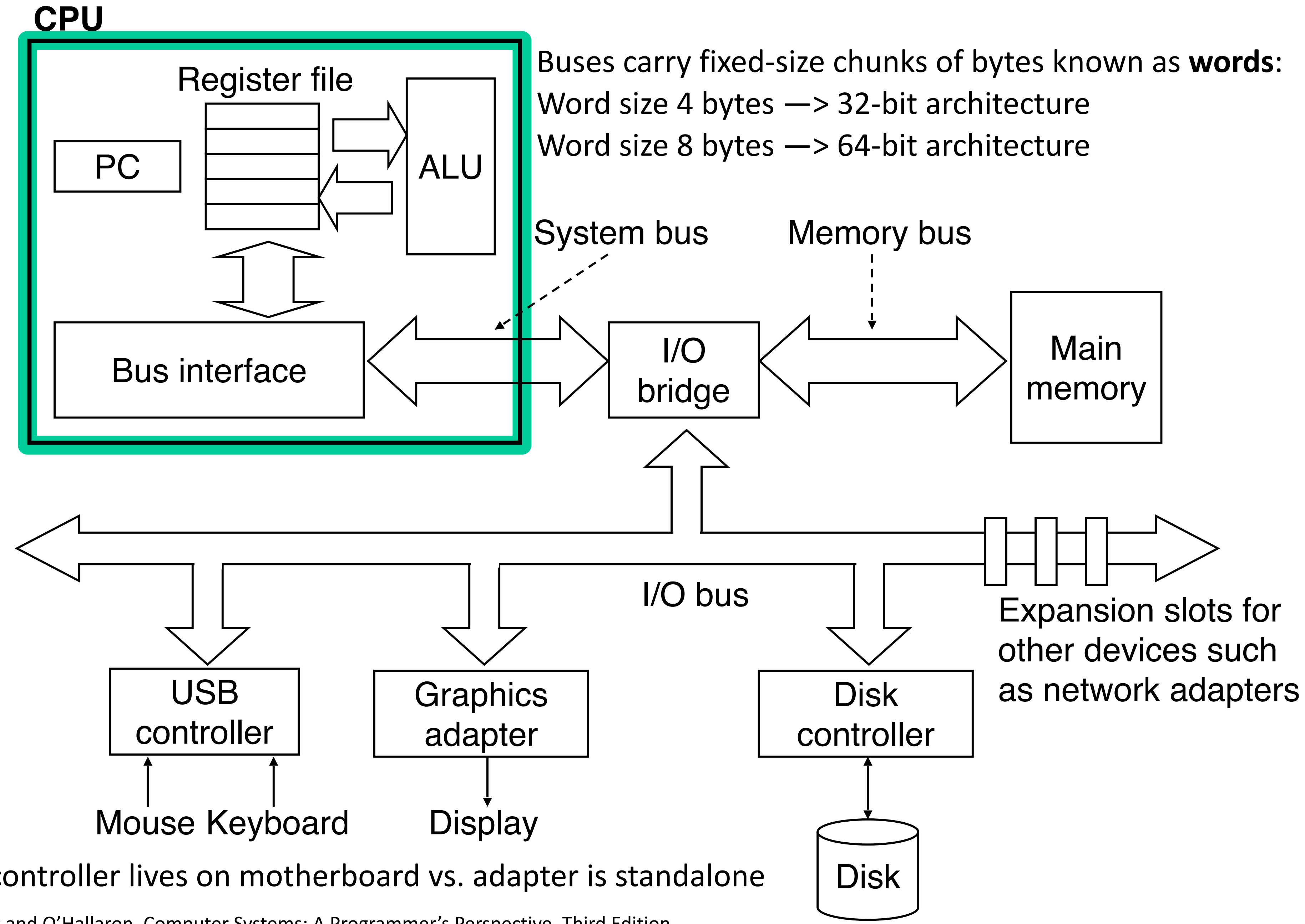
Machine-Level Programming I: Basics

CS2011: Introduction to Computer Systems
Lecture 6 (3.1, 3.2, 3.3, 3.4, 3.5)

Machine Programming I: Basics

- **History of Intel processors and architectures**
- Assembly Basics: Registers, operands, data movement operations
- Arithmetic & logical operations
- C, assembly, machine code

Hardware Organization (Review)



Buses carry fixed-size chunks of bytes known as **words**:
Word size 4 bytes —> 32-bit architecture
Word size 8 bytes —> 64-bit architecture

controller lives on motherboard vs. adapter is standalone

Intel x86 Processors

- **Dominate laptop/desktop/server market**
  - ARM architecture is a promising newcomer
- **Evolutionary design**
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
    - Now 3 volumes, about 5,000 pages of documentation
    - https://cdrdv2.intel.com/v1/dl/getContent/671200
- **Complex instruction set computer (CISC)**
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (**RISC**)
  - But, Intel has done just that (i.e., matching performance or RISC)!
    - In terms of speed. Less so for low power.

Intel x86 Evolution: Milestones

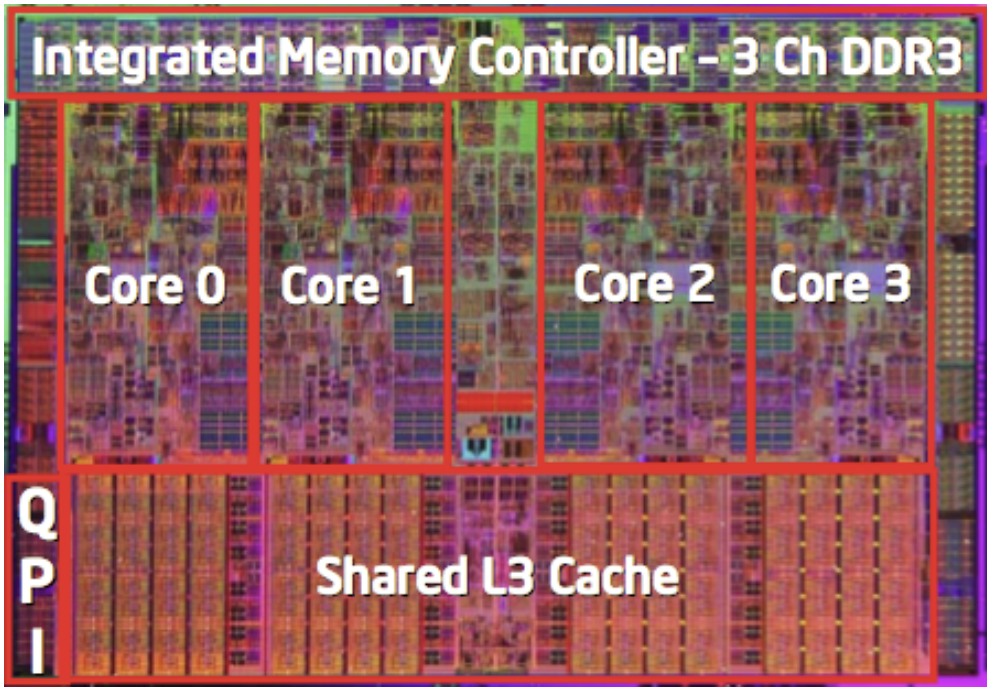
Name	*Date*	*Transistors*	*MHz*
8086	**1978**	**29K**	**5-10**
386	**1985**	**275K**	**16-33**
Pentium 4E	**2004**	**125M**	**2800-3800**
Core 2	**2006**	**291M**	**1060-3333**
Core i7	**2008**	**731M**	**1600-4400**

- **8086**
  - First **16-bit Intel processor**. Basis for IBM PC & DOS
  - 1MB address space
- **386**
  - First **32 bit Intel processor** , referred to as **IA32**
  - Added “flat addressing”, capable of running Unix
- **Pentium 4E**
  - First **64-bit Intel x86 processor**, referred to as **x86-64**
- **Core 2**
  - First multi-core Intel processor (much less power hungry than single processor)
- **Core i7**
  - Four cores

Intel x86 Processors, briefly

Machine Evolution

8086 (16 bit)	1978	29k
386 (32 bit!)	1985	0.3M
Pentium 4e (64 bit)	2004	55M
Core 2 (multicore!)	2006	291M
Core i7	2008	731M
Core i9	2019	3.5B



Added Features

- Instructions to support multimedia operations
- Instructions to enable ***more efficient conditional operations***
- Transition from 32 bits to 64 bits
- More cores

Intel x86 Processors, cont.

Past Generations

		Process technology
1st Pentium Pro	1995	600 nm
1st Pentium III	1999	250 nm
1st Pentium 4	2000	180 nm
1st Core 2 Duo	2006	65 nm

Recent & Upcoming Generations

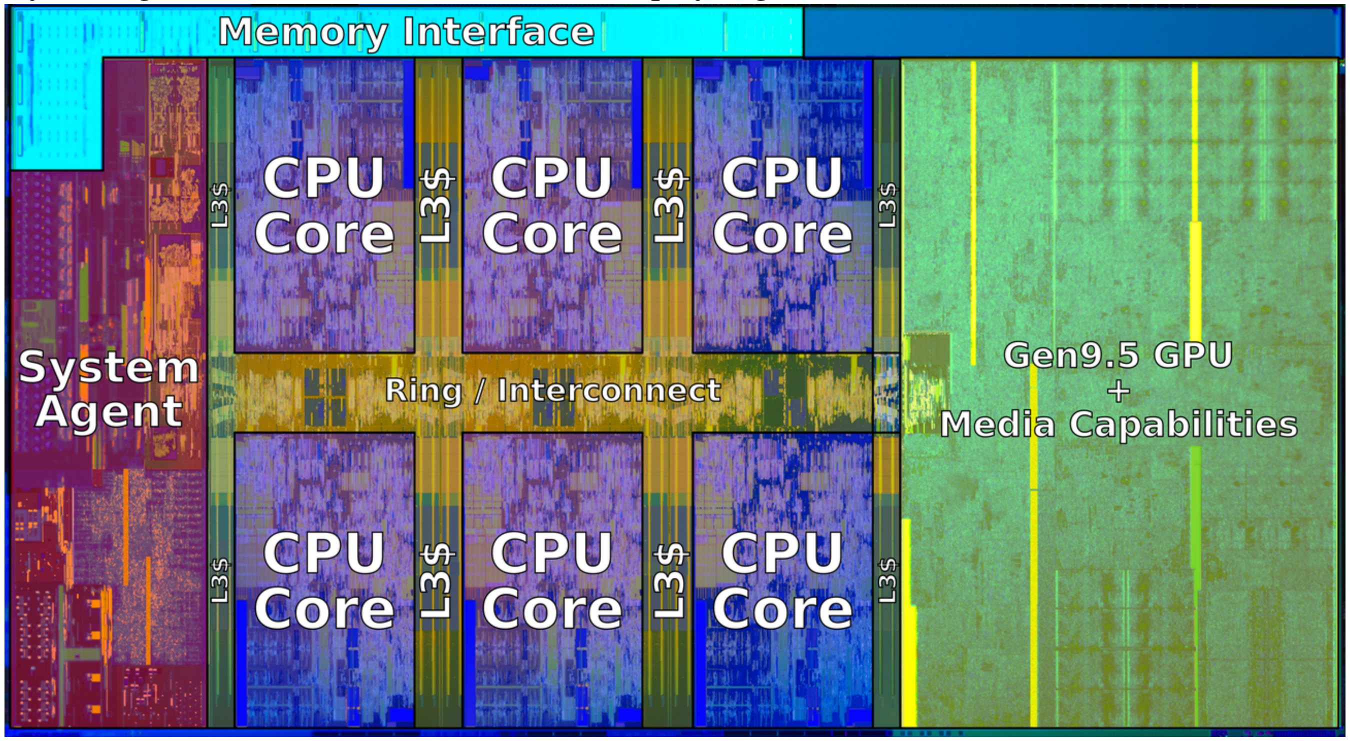
1.	Nehalem	2008	45 nm
2.	Sandy Bridge	2011	32 nm
3.	Ivy Bridge	2012	22 nm
4.	Haswell	2013	22 nm
5.	Broadwell	2014	14 nm
6.	Skylake	2015	14 nm
7.	Kaby Lake	2016	14 nm
8.	Coffee Lake	2017	14 nm
9.	Cannon Lake	2018	10 nm
10.	Ice Lake	2019	10 nm
11.	Tiger Lake	2020	10 nm
12.	Alder Lake	2022	“intel 7” (10nm+++)

Process technology dimension
= width of narrowest wires
(10 nm ≈ 100 atoms wide)

(But this is changing now.)

2018 State of the Art: Coffee Lake

System agent contains the I/O bus and the display engine



- **Mobile Model: Core i7**
  - 2.2-3.2 GHz
  - 45 Watt
- **Desktop Model: Core i7**
  - Integrated graphics
  - 2.4-4.0 GHz
  - 35-95 Watt
- **Server Model: Xeon E**
  - Integrated graphics
  - Multi-socket enabled
  - 3.3-3.8 GHz
  - 80-95 Watt

x86 Clones: Advanced Micro Devices (AMD)

- **Historically**
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- **Then**
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed **x86-64, their own extension to 64 bits – which won!**
- **Recent Years**
  - Intel got its act together
    - 1995-2011: Lead semiconductor "fab" in world
    - 2018: #2 largest by $$ (#1 is Samsung)
    - 2019: reclaimed #1
  - AMD fell behind: Spun off GlobalFoundaries
  - 2019-20: Pulled ahead! Used TSMC for part of fab
  - 2022: Intel re-took the lead

Intel's 64-Bit History

- **2001: Intel Attempts Radical Shift from IA32 to IA64**
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
    - Compiler technology couldn't keep its promises of doing all the hard work to make 64bits work
- **2003: AMD Steps in with Evolutionary Solution**
  - AMD makes x86 clones but then .. started innovating
  - **x86-64** (now called "**AMD64**")
- **Intel Felt Obligated to Focus on IA64**
  - Hard to admit mistake or that AMD is better
- **2004: Intel Announces EM64T extension to IA32**
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!
- **All but low-end x86 processors support x86-64**
  - But, ***lots of code still runs in 32-bit mode***

Book Coverage

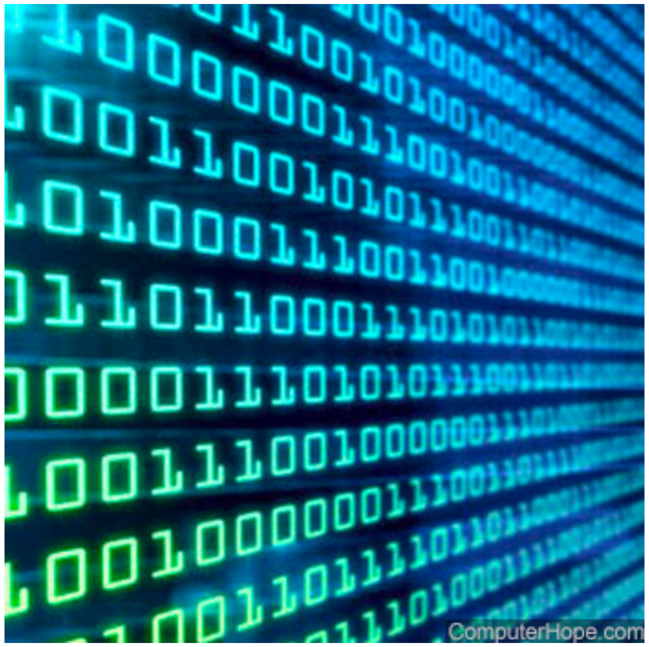
- **IA32**
  - The traditional x86
  - Second Edition
- **x86-64**
  - The standard
  - `linux> gcc hello.c`
  - `linux> gcc -m64 hello.c`
- **Presentation**
  - Book covers x86-64
  - *Web aside on IA32*
  - **We will only cover x86-64**

Machine Programming I: Basics

- History of Intel processors and architectures
- **Assembly Basics: Registers, operands, data movement operations**
- Arithmetic & logical operations
- C, assembly, machine code

How Does a Computer Understand Code?

- Short answer: Bits!
- Not the complete picture, let’s take it bit by bit



How Does a Person Understand English?

- **Let’s see how humans understand English**
  - English uses an alphabet
    - A, B, C …
  - Examine a **sentence**:
    - Subject: (**I**) core noun of the sentence
    - Object: (**CS2011**) supporting noun
    - Verbs: (**love**) actions associating subject and object

I love CS2011!

How Does a CPU Understand Code (cont)?

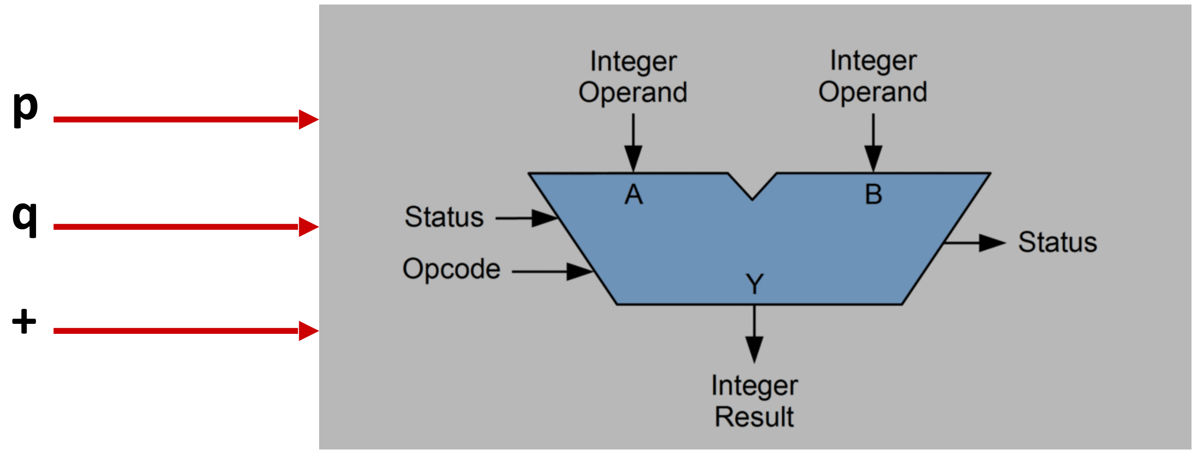
- **What's the language of the CPU?** ***Instructions***
  - Basic building block is bits
  - A sentence in English is like a program
  - **Subject and Object?**
    - **Data**: some specific Integers, Floats, …
  - **Verb?**
    - **Arithmetic Operations**: +, -, /, *, <<, >>, …

- **CPU: Take data, apply action, use result**
  - Let's encode: 1 + 2
    - (+, 1, 2)

Generalize Our Encoding

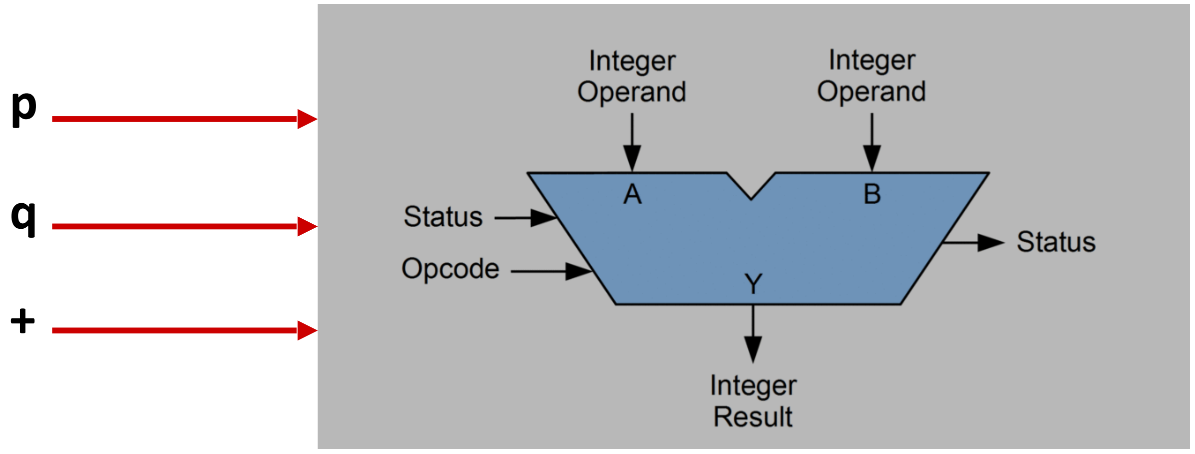
- 1 + 2 -> (+, 1, 2)
- 3 + 4 -> (+, 3, 4)
- Let’s make specialized hardware in our CPU for +, -, …

1 + 2



- Separate operations from data
  - p = 1
  - q = 2
  - (+, p, q)

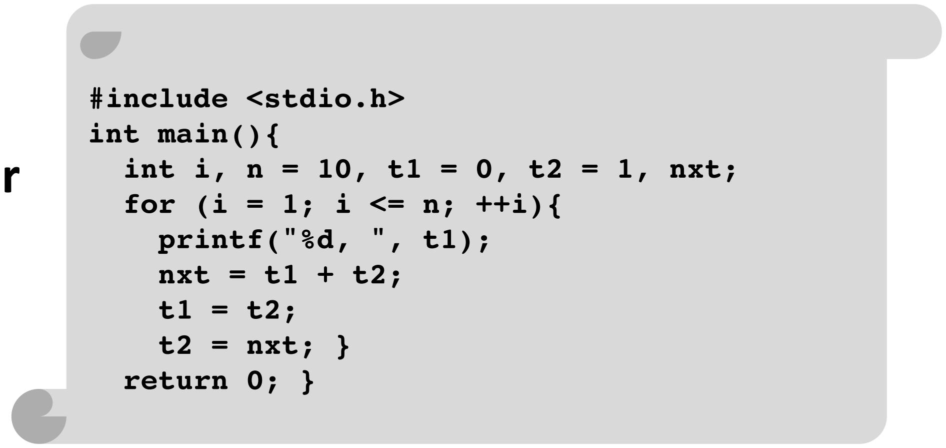
Generalize Our Data Pipeline



- CPU needs to supply p, q to our arithmetic unit
- CPU uses **registers** to store information for the ALU
  - (+, %p, %q)
- **But where do the registers get information?**
  - From the program, and in memory
    - (Load, **0x1**, %p)
    - (Load, **&0x7FFFF7AFCBA0**, %q)

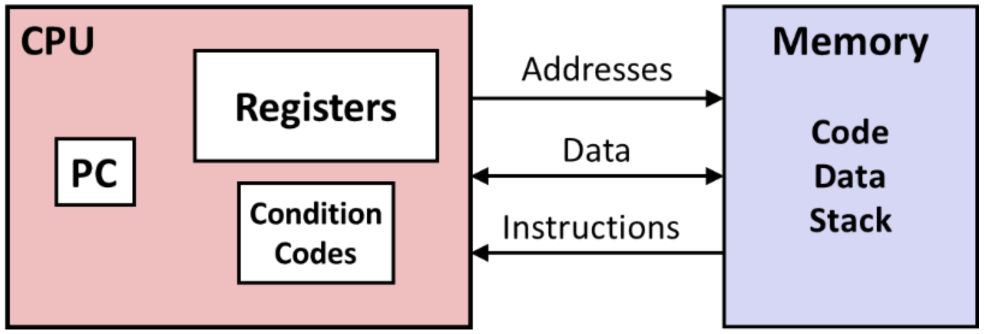
Levels of Abstraction

C programmer

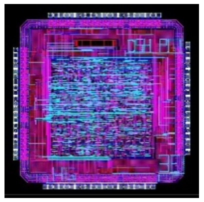


```
#include <stdio.h>
int main(){
  int i, n = 10, t1 = 0, t2 = 1, nxt;
  for (i = 1; i <= n; ++i){
    printf("%d, ", t1);
    nxt = t1 + t2;
    t1 = t2;
    t2 = nxt; }
  return 0; }
```

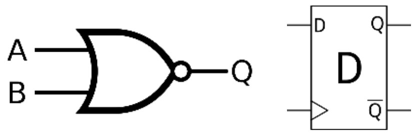
Assembly programmer



Computer Designer



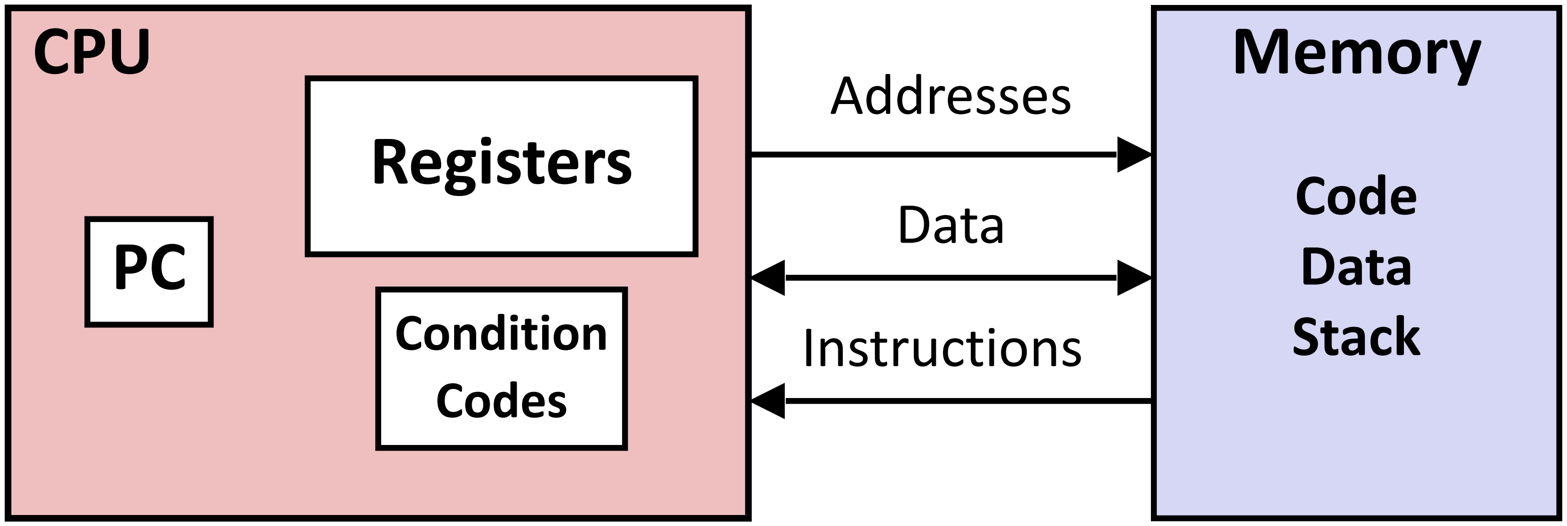
Gates, clocks, circuit layout, …



Definitions

- **Architecture: (also *ISA*: instruction set architecture) The parts of a processor design that one needs to understand for writing assembly/machine code.**
  - Examples: instruction set specification, registers
  - Impacts ***correctness*** of program
- **Microarchitecture: Implementation of the architecture**
  - Examples: cache sizes and core frequency
  - Impacts ***performance*** of program (speed, power usage, etc.)
- **Code Forms:**
  - Machine Code: The **byte-level** programs that a **processor executes**
  - Assembly Code: A **text representation** of machine code
    - Human readable
- **Example ISAs:**
  - Intel: x86, IA32, Itanium, **x86-64**
  - **ARM**: Used in almost all mobile phones
    - Based on RISC
  - **RISC V (risk five)**: New open-source ISA

Assembly/Machine Code View



Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called "**RIP**" (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most ***recent arithmetic or logical operation***
  - Used for **conditional branching**

- **Memory**
  - ***Byte addressable array***
  - Code and user data
  - Stack to support procedures

Assembly Characteristics: Data Types

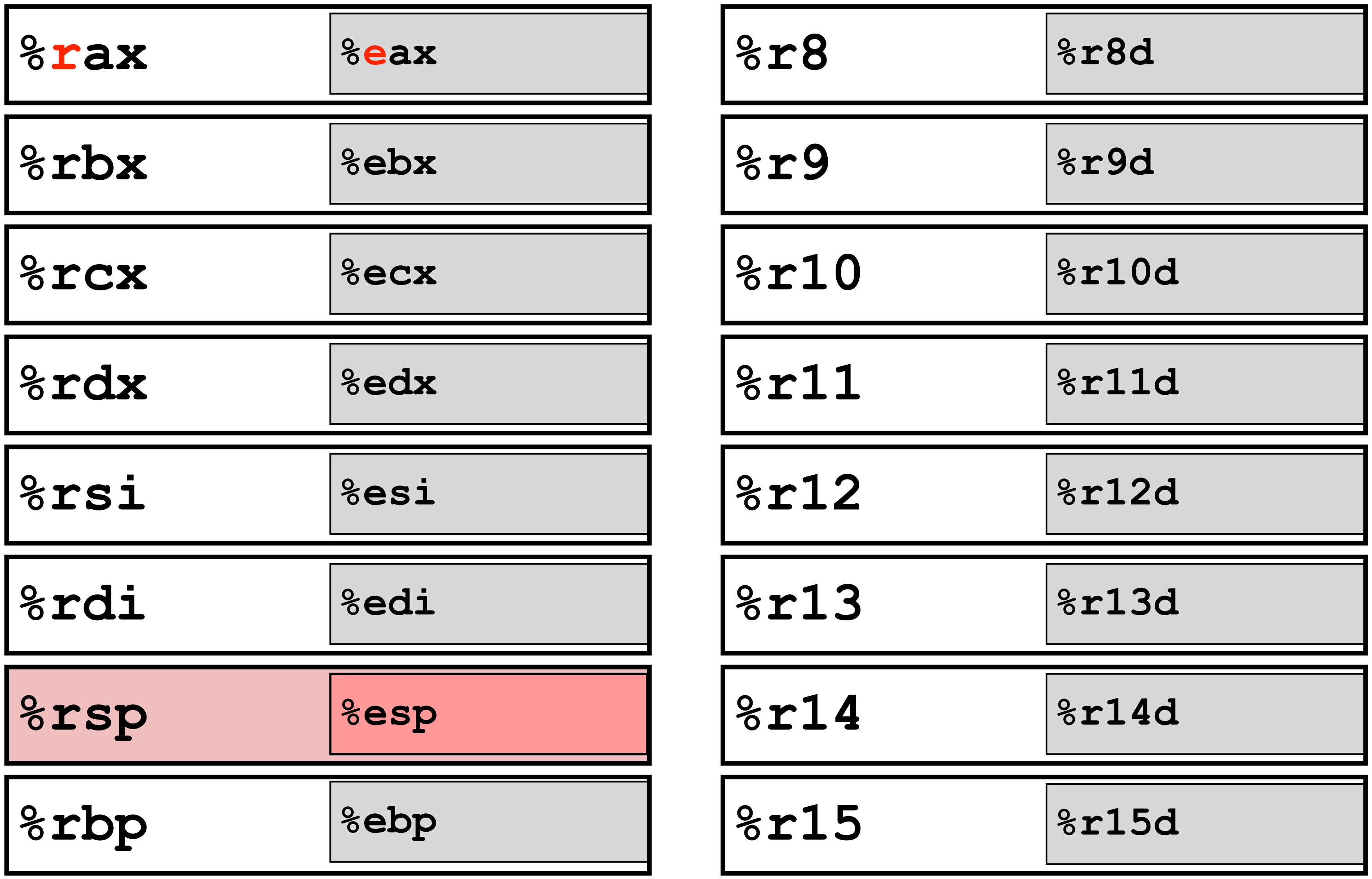
- **"Integer" data of 1 (char), 2 (short), 4 (int), or 8 (long, ptr) bytes**
  - Data values
  - Addresses (untyped pointers)
  - BYTE (1), WORD (2), **D**WORD "Double Word" (4), **Q**WORD "Quad Word"(8)
    - The **original 8086 16-bit** arch referred to the 16-bit data type as **word**
- **Floating point data of 4 (float), 8 (double), or 10 bytes**
  - Historically all floating point data in x86 was represented in 10 bytes. Continuing to use 10 bytes is not recommended due to:
    - 1) not portable to other classes of machines
    - 2) not implemented with the same high-performance hardware as single- and double-precision arithmetic
- **Code: Byte sequences encoding series of instructions**
- **No aggregate types such as arrays or structures**
  - Just contiguously allocated bytes in memory

Assembly Characteristics: Data Types

C Declaration	Intel Data Type	Assembly Code Suffic	Size (bytes)
char	Byte	b	1
short	Word	w	2
int	**Double** word	l	4
long	**Quad** word	q	8
char *	Quad word	q	8
float	Single precision	s	4
double	Double precision	l	8

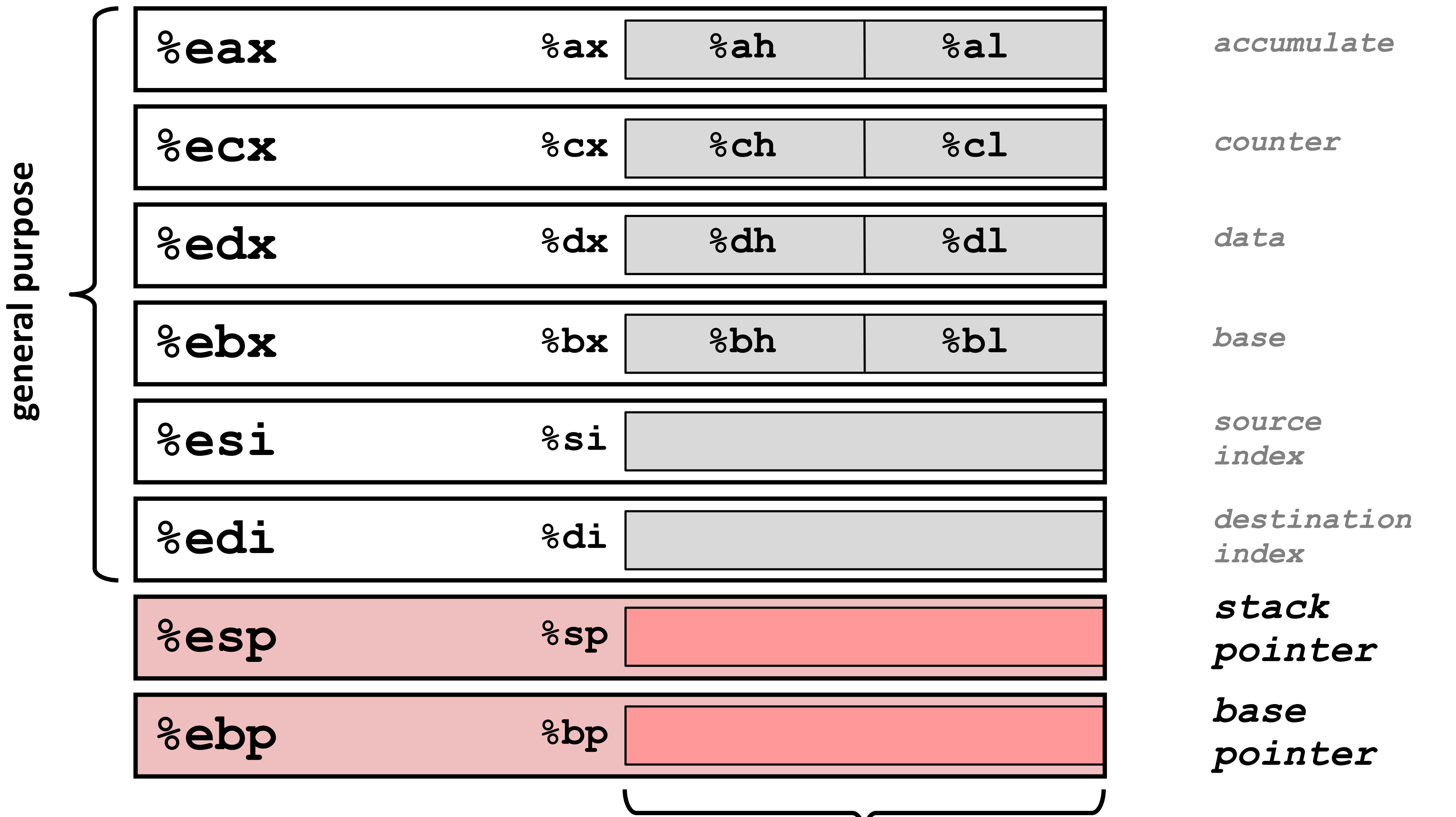
- Assembly Code suffix: e.g., mov**b**, mov**w**, mov**l**, mov**q**
  - No ambiguity between int and double (both use l as suffix) since int and floating point have different instructions

x86-64 16 Integer Registers



- Backward Compatibility: Can reference ***low-order 4 bytes*** (also low-order 1 & 2 bytes)
- Not part of memory (or cache)

Some History: IA32 Registers



16-bit virtual registers (backwards compatibility)

Assembly Characteristics: Operations

- **Transfer data between memory and register**
  - Load data from memory into register
  - Store register data into memory

- **Perform arithmetic function on register or memory data**

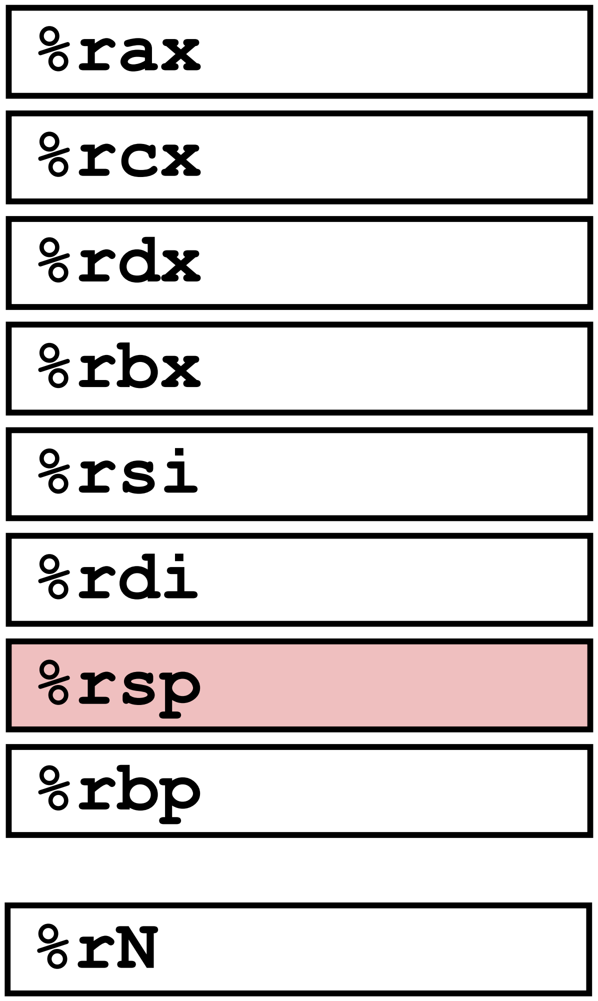
- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches
  - Indirect branches

Moving Data

- **Moving Data**

  `movq` *SourceOperand, DestOperand*

- **Operand Types**
  - ***Immediate:*** Constant integer data
    - Example: **`$0x400, $-533`**
    - Like C constant, but prefixed with **`‘$’`**
    - Encoded with 1, 2, or 4 bytes
  - ***Register:*** One of 16 integer registers
    - Example: **`%rax, %r13`**
    - But **`%rsp`** reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 8 consecutive bytes of memory at address given by register
    - Simplest example: **`(%rax)`**
    - Various other “addressing modes”



Warning: Intel docs use mov *Dest, Source*

`movq` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction

Instruction suffix (e.g., movq) must match size of operand

Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C (*p)

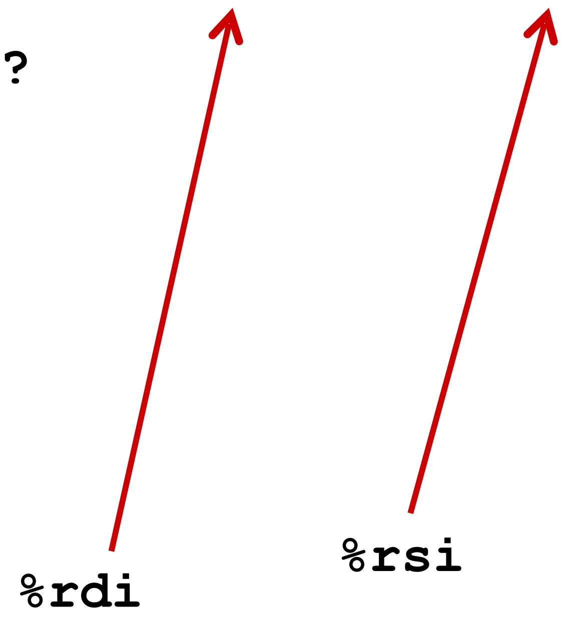
```
movq (%rcx),%rax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies **offset**
  - E.g., accessing arrays

```
movq 8(%rbp),%rdx
```

Example of Simple Addressing Modes

```
void
whatAmI(<type> a, <type> b)
{
      ????
}
```



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

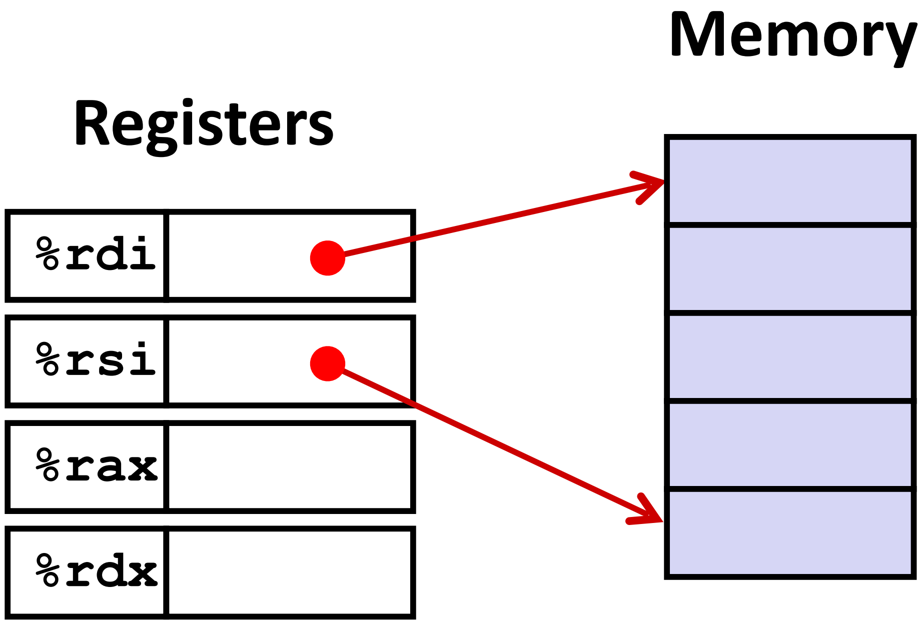
Example of Simple Addressing Modes

```
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
```

Understanding Swap()

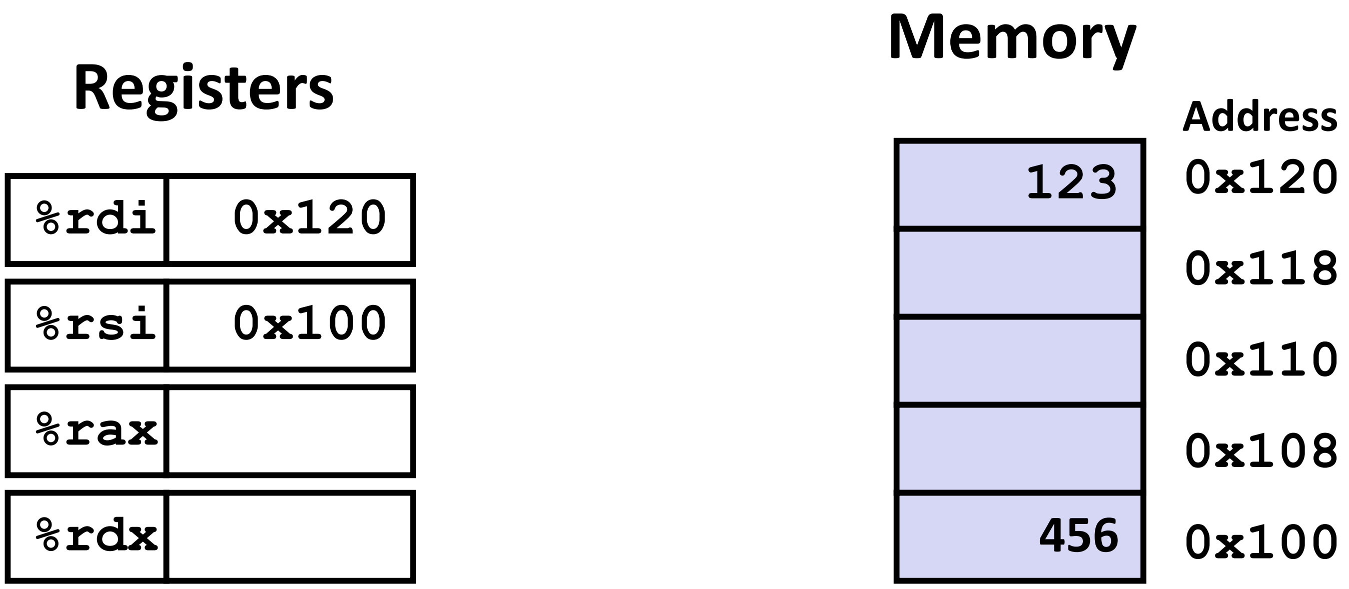
```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

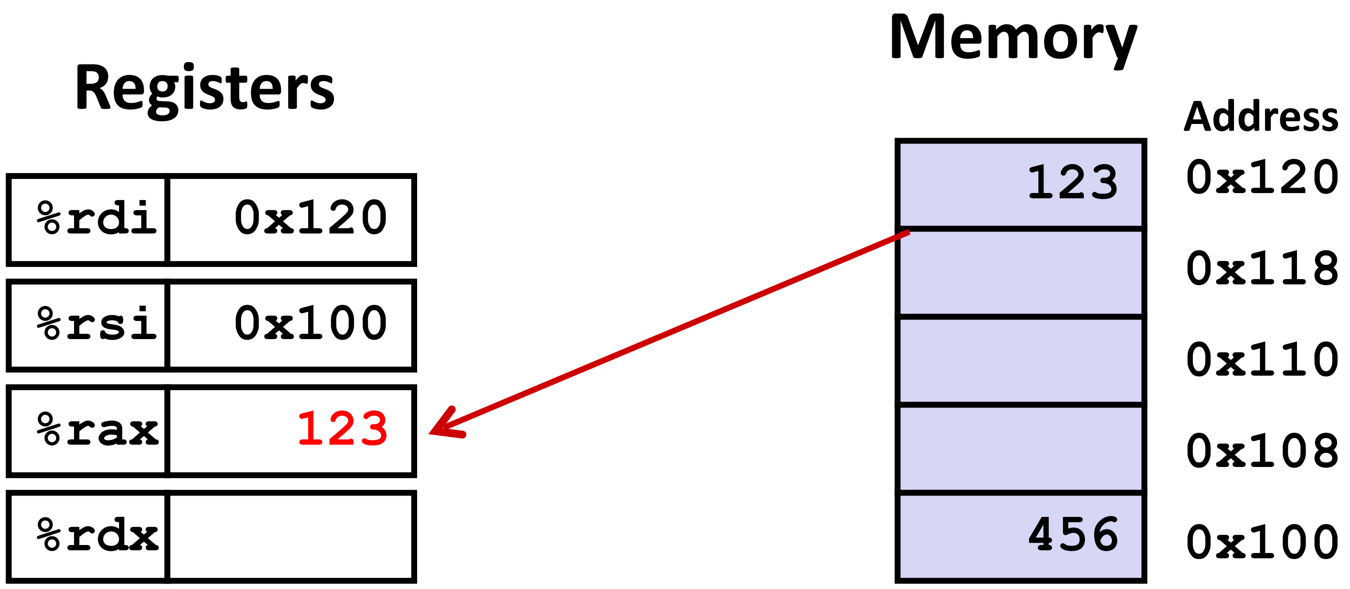
```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



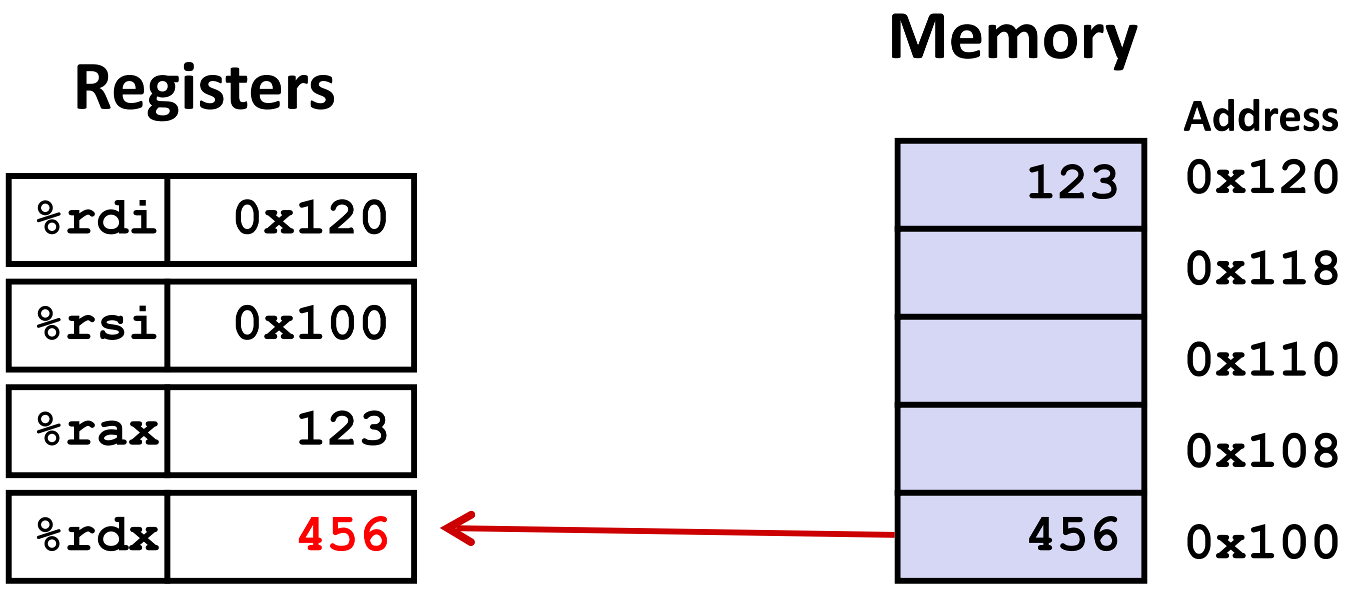
```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



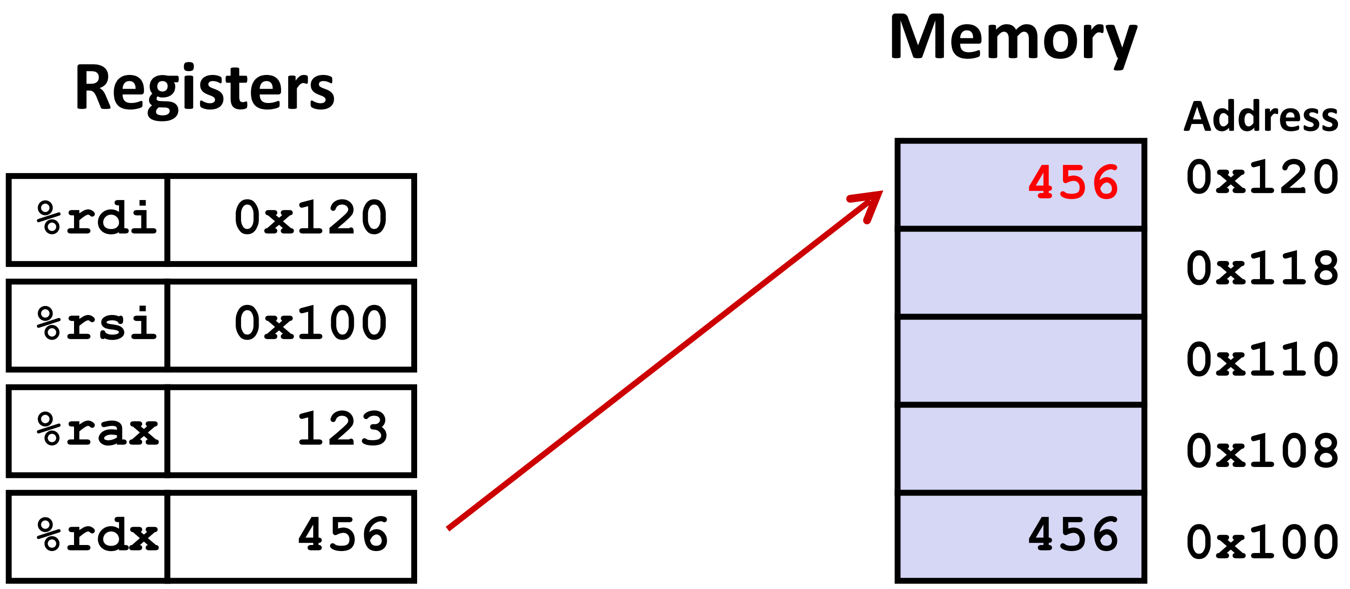
```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



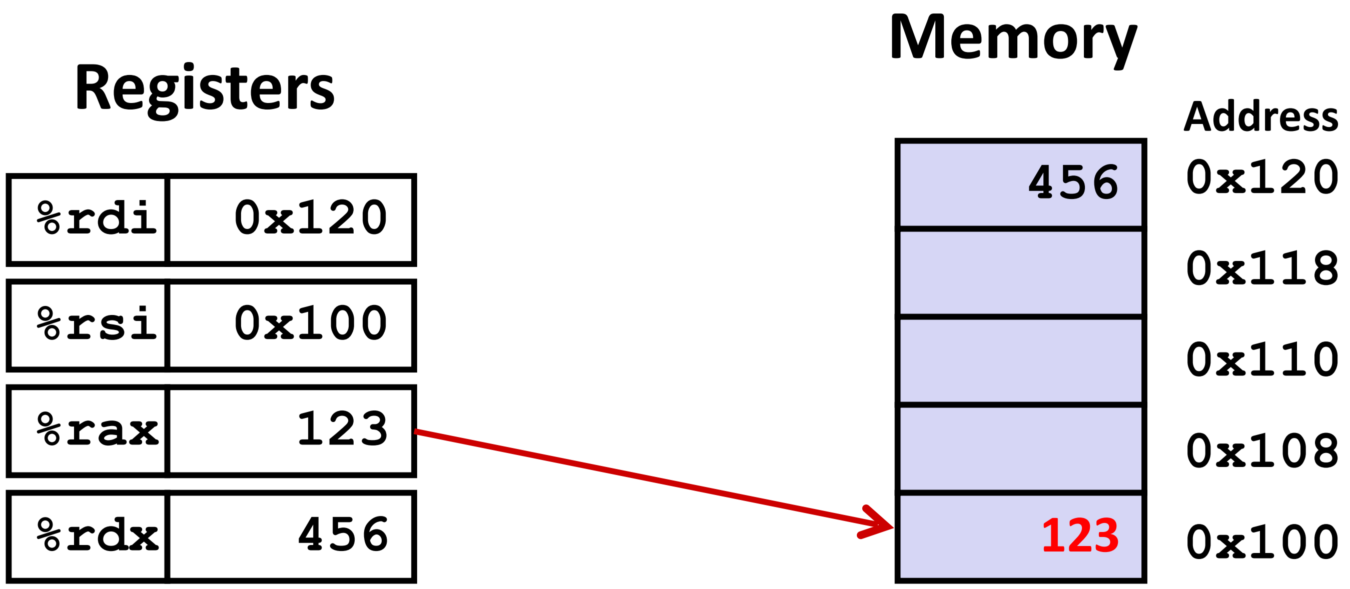
```
swap:
    movq     (%rdi), %rax   # t0 = *xp
    movq     (%rsi), %rdx   # t1 = *yp
    movq     %rdx, (%rdi)   # *xp = t1
    movq     %rax, (%rsi)   # *yp = t0
    ret
```

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
```

Simple Memory Addressing Modes (Review)

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

Complete Memory Addressing Modes

- **Most General Form**

D(Rb,Ri,S)	**Mem[Reg[Rb]+S*Reg[Ri]+ D]**

  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: **Base** register: Any of 16 integer registers
  - Ri: **Index** register: Any, except for **`%rsp`**
  - **S**: **Scale**: 1, 2, 4, or 8 (*why these numbers?*)
  - E,g, Multidimensional array:
    - **Reg[Ri]** is size of **row**
    - **S** is **row index** / and **D** is **column index**

- **Special Cases**

(Rb,Ri)	**Mem[Reg[Rb]+Reg[Ri]]**
D(Rb,Ri)	**Mem[Reg[Rb]+Reg[Ri]+D]**
(Rb,Ri,S)	**Mem[Reg[Rb]+S*Reg[Ri]]**

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

D(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

Expression	Address Computation	Address
`0x8(%rdx)`		
`(%rdx,%rcx)`		
`(%rdx,%rcx,4)`		
`0x80(,%rdx,2)`		

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Movabsq Instruction

Instruction	Description
mov**x** *S*, *D*	Move a byte (x = b), word (x = w), double word (x = l), or quad word (x = q) from source *S* to destination *D*
movabsq *I*, *R*	Move absolute quad word immediate *I* to register *R*

- **movabsq** deals with 64-bit immediate data
    - Regular **movq** only allows 32-bit immediate "two's complement numbers"
    - mov**q** **sign-extends** 32-bit immediate into **64-bit destination**
    - movabsq can have any ***arbitrary*** 64-bit immediate, but can only have a Register as its destination operand

Zero-extending MOVZ Instruction

Instruction	Description
movzbw *S*, *R*	Zero extend **byte** S and move into **word** R
movzbl *S*, *R*	Zero extend **byte** S and move into **double word** R
movzwl *S*, *R*	Zero extend **word** S and move into **double word** R
movzbq *S*, *R*	Zero extend **byte** S and move into **quad word** R
movzwq *S*, *R*	Zero extend **word** S and move into **quad word** R

- **First suffix** denotes the **source size** and **second suffix** denotes the **destination size**
  - e.g., movzbw —> b: source size is byte, w: destination size is word
- Notice that movzlq does NOT exist
  - Can be implemented using **movl** with **register as destination**
  - Any instruction generating a 4-byte value with register as destination will always fill the upper 4 bytes with zeros (equivalent to **zero extension**)

Zero-extending MOVZ Instruction: Example

```
Void cast()
{
 uint8_t uint8_val = 255;
 uint32_t uint32_val = (uint32_t) uint8_val;
}
```

```
  movb          $-1, -5(%rbp)
  movzbl         -5(%rbp), %eax
  movl          %eax, -4(%rbp)
```

Sign-extending MOVS Instruction

Instruction	Description
movsbw *S, R*	Sign extend byte S and move into word R
movsbl *S, R*	Sign extend byte S and move into double word R
movswl *S, R*	Sign extend word S and move into double word R
movsbq *S, R*	Sign extend byte S and move into quad word R
movswq *S, R*	Sign extend word S and move into quad word R
movslq *S, R*	Sign extend double word S and move into quad word R
cltq	Sign extend %eax and move into %rax

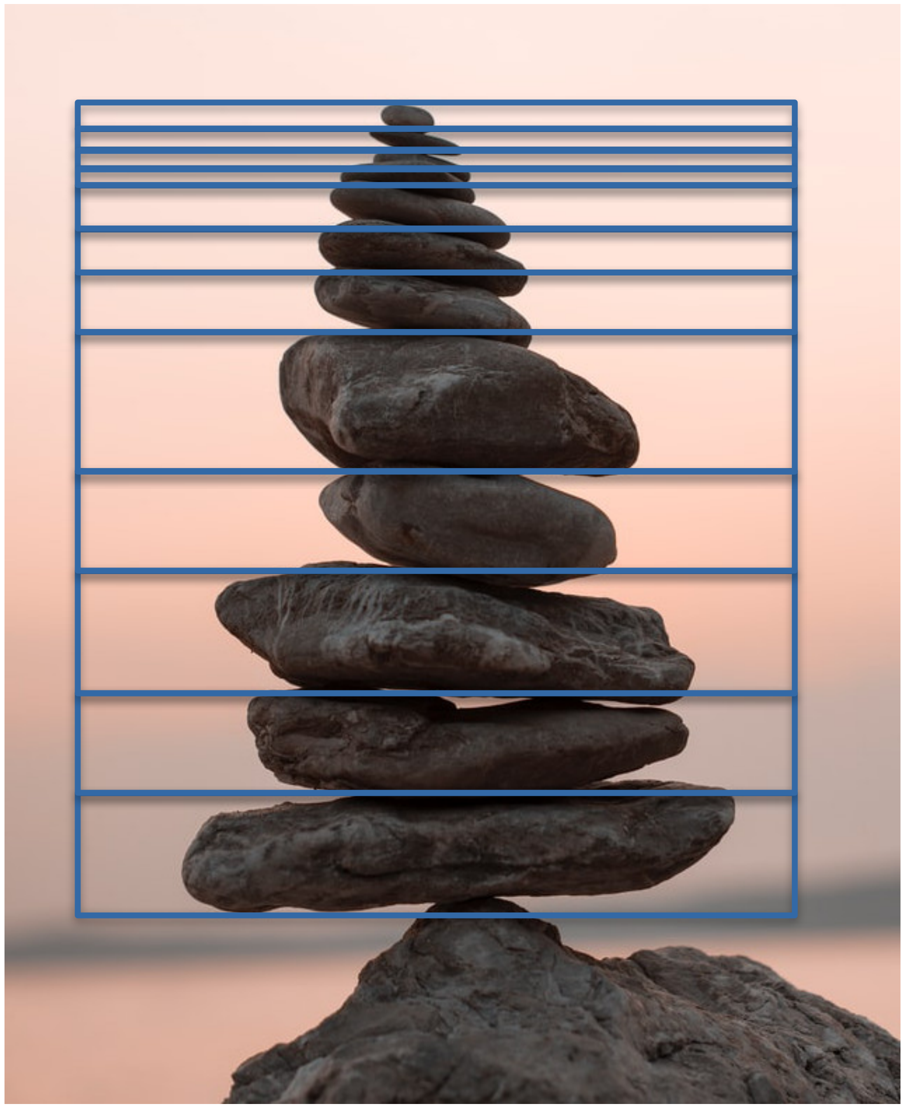
- cltq == movslq %eax, %rax

Sign-extending MOVS Instruction: Example

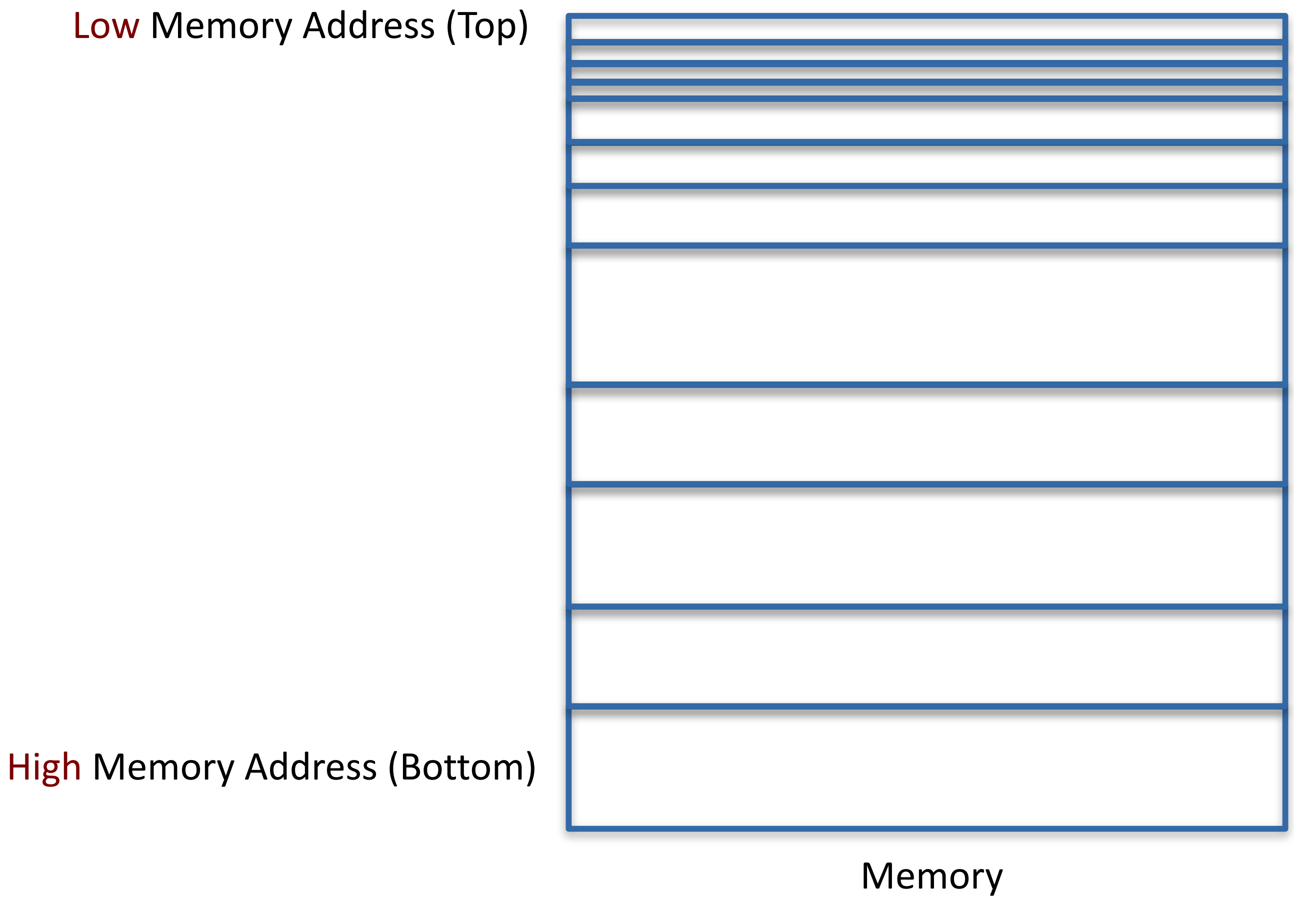
```
Void cast()
{
 int8_t int8_val = 127;
 int16_t int16_val = (int16_t) int8_val;

 int8_t int8_neg_val = -128;
 int16_t int16_neg_val = (int16_t) int8_neg_val;
}
```

```
  movb          $127, -6(%rbp)
  movsbw        -6(%rbp), %ax
  movw          %ax, -4(%rbp)
  movb          $-128, -5(%rbp)
  movsbw        -5(%rbp), %ax
  movw          %ax, -2(%rbp)
```

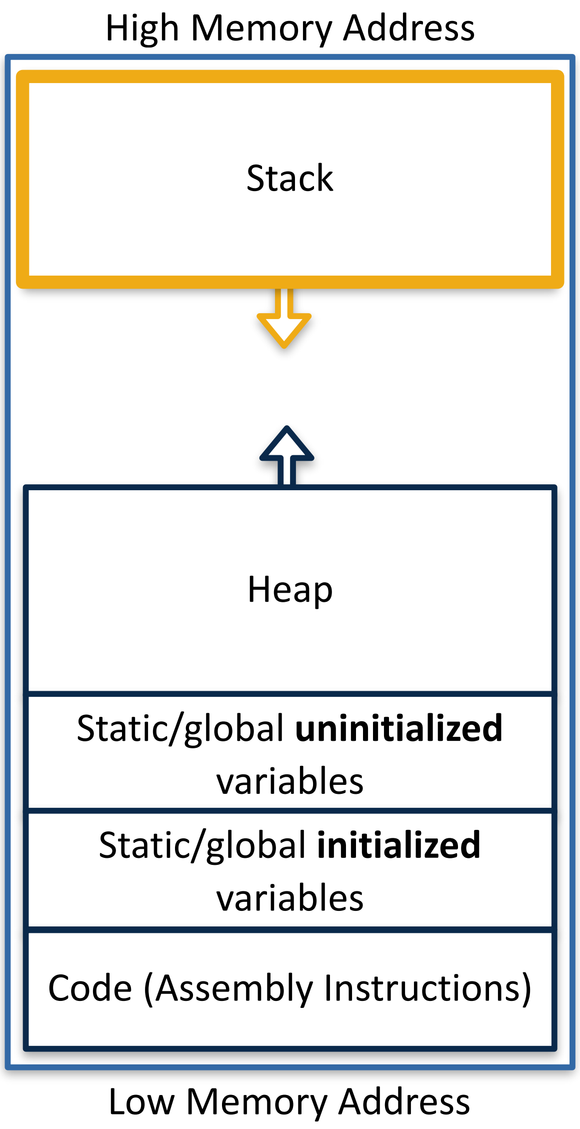
Stack Push/Pop Data Instructions (Stack)

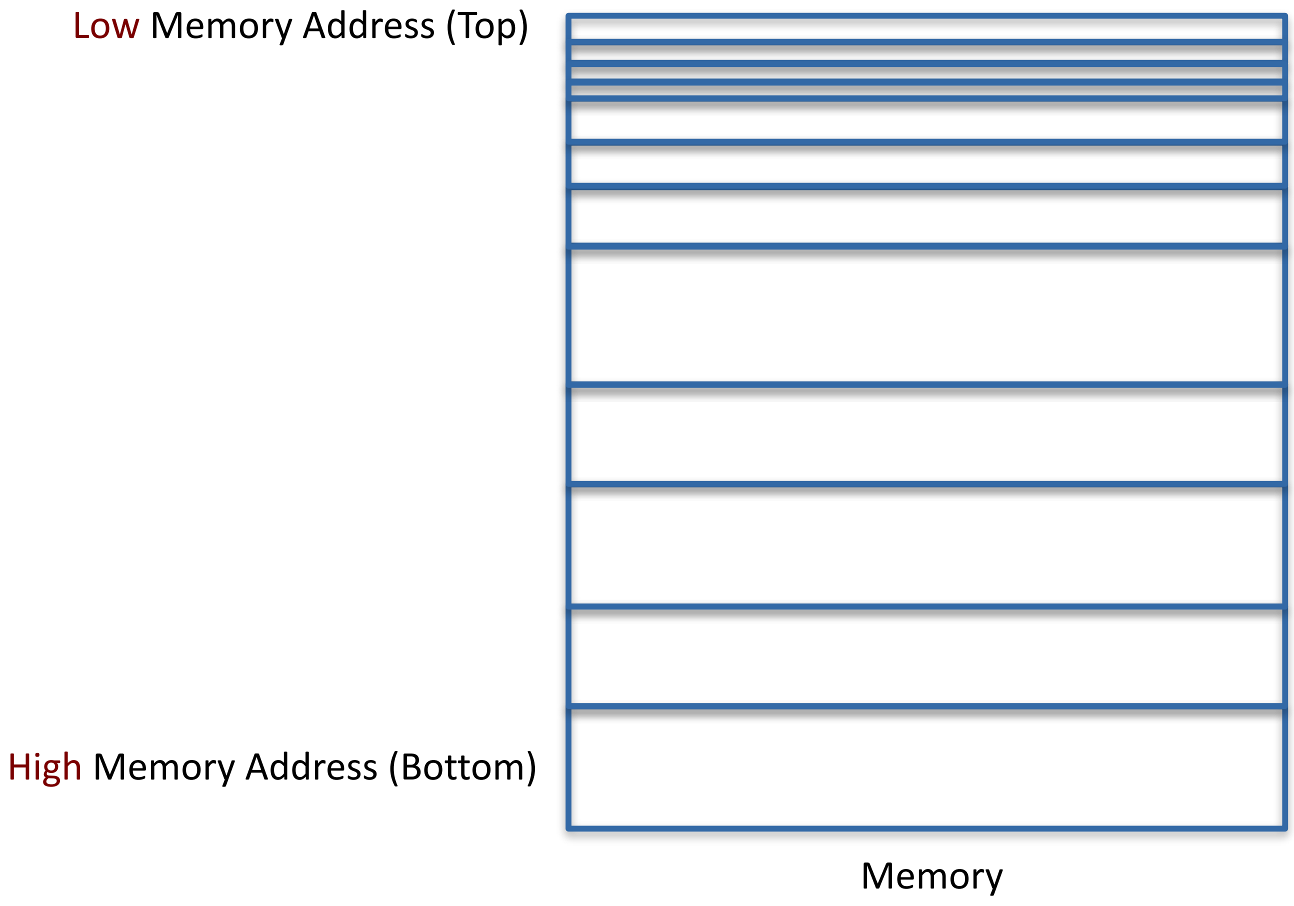


Stack Push/Pop Data Instructions (Stack)

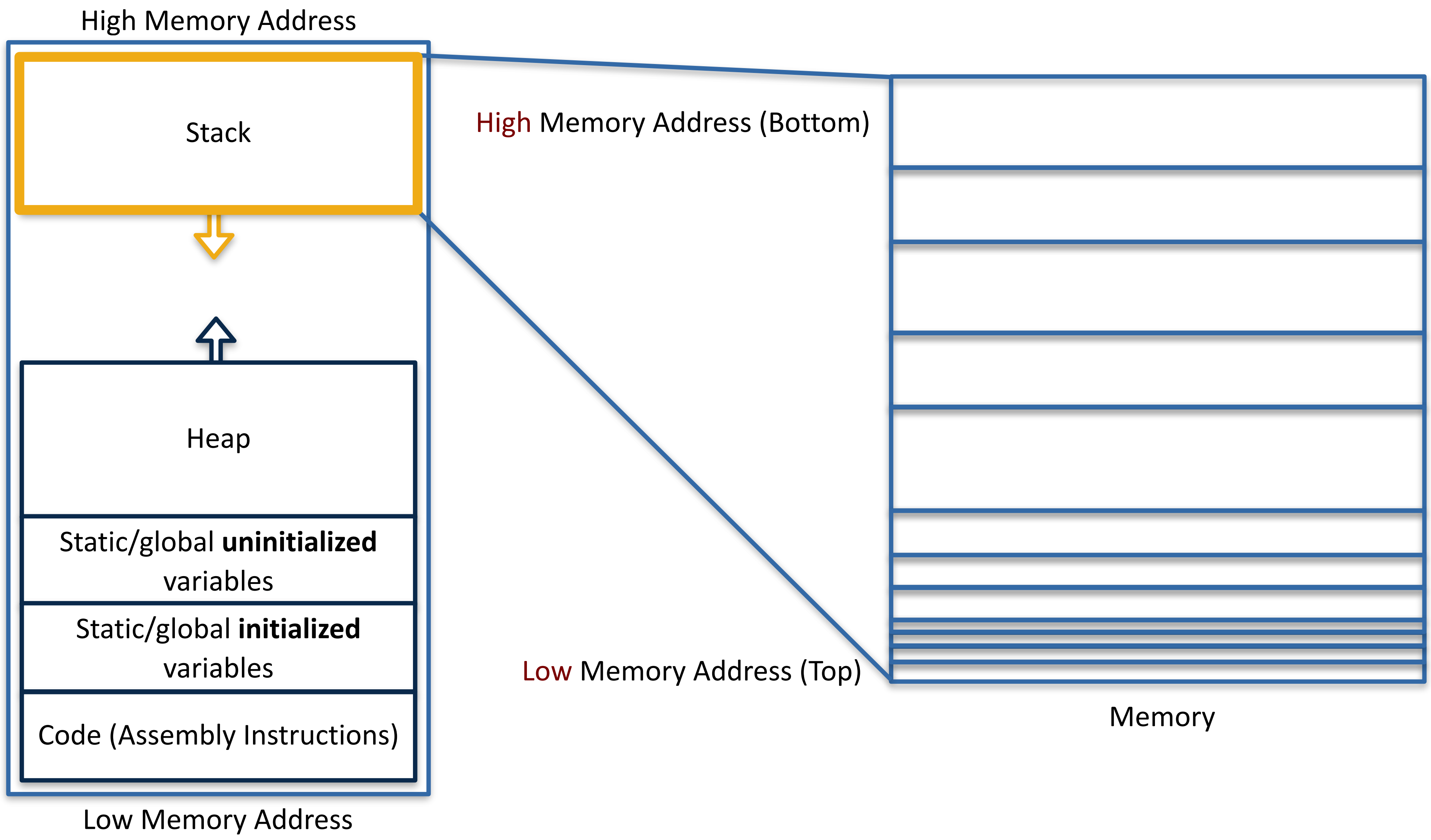


Stack Push/Pop Data Instructions (Stack)

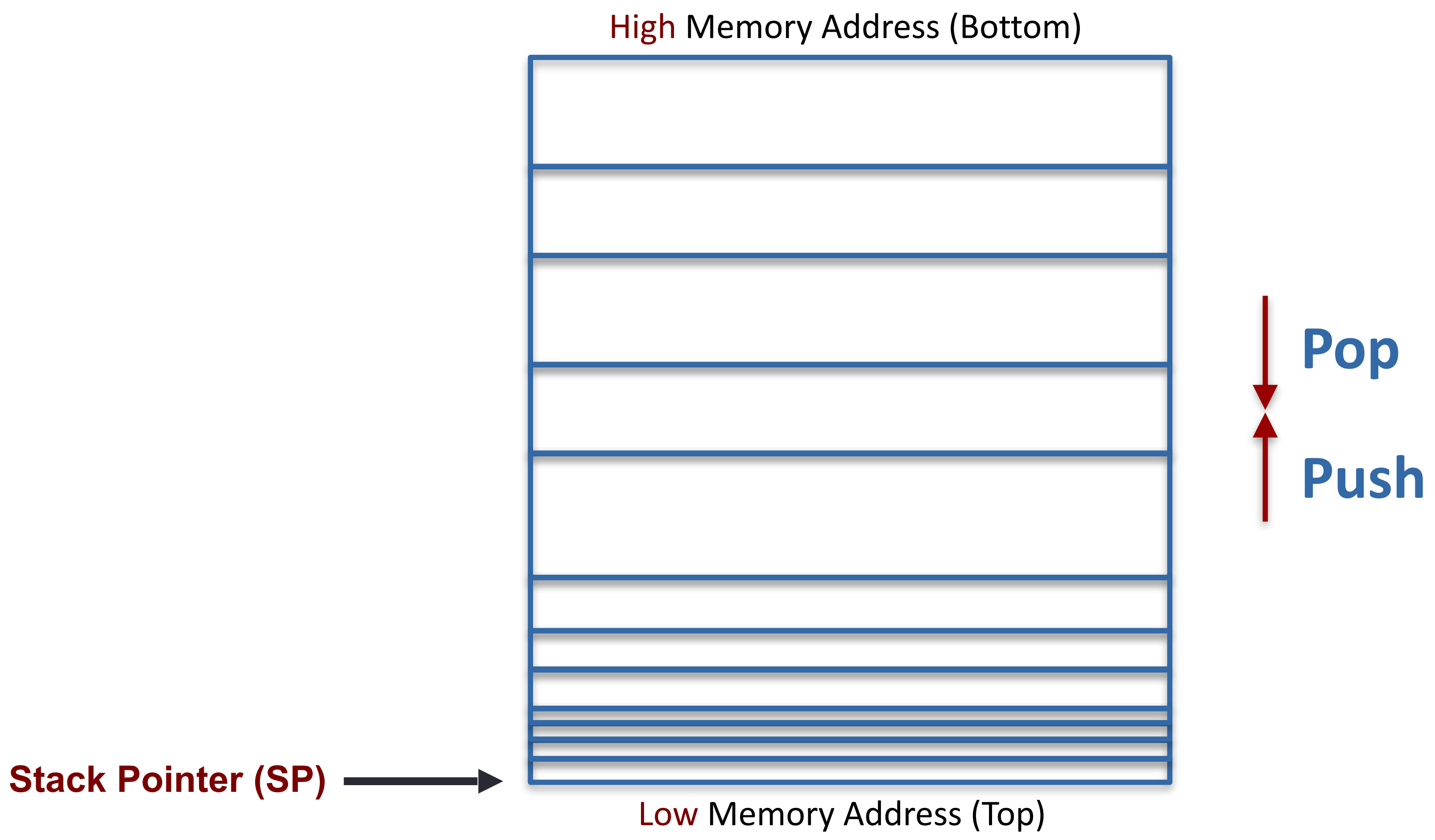

High Memory Address
Stack
Heap
Static/global **uninitialized** variables
Static/global **initialized** variables
Code (Assembly Instructions)
Low Memory Address



Stack Push/Pop Data Instructions (Stack)


High Memory Address
Stack
Heap
Static/global **uninitialized** variables
Static/global **initialized** variables
Code (Assembly Instructions)
Low Memory Address
High Memory Address (Bottom)
Low Memory Address (Top)
Memory

Stack Push/Pop Data Instructions (Stack)



Stack Push/Pop Data Instructions

Initially

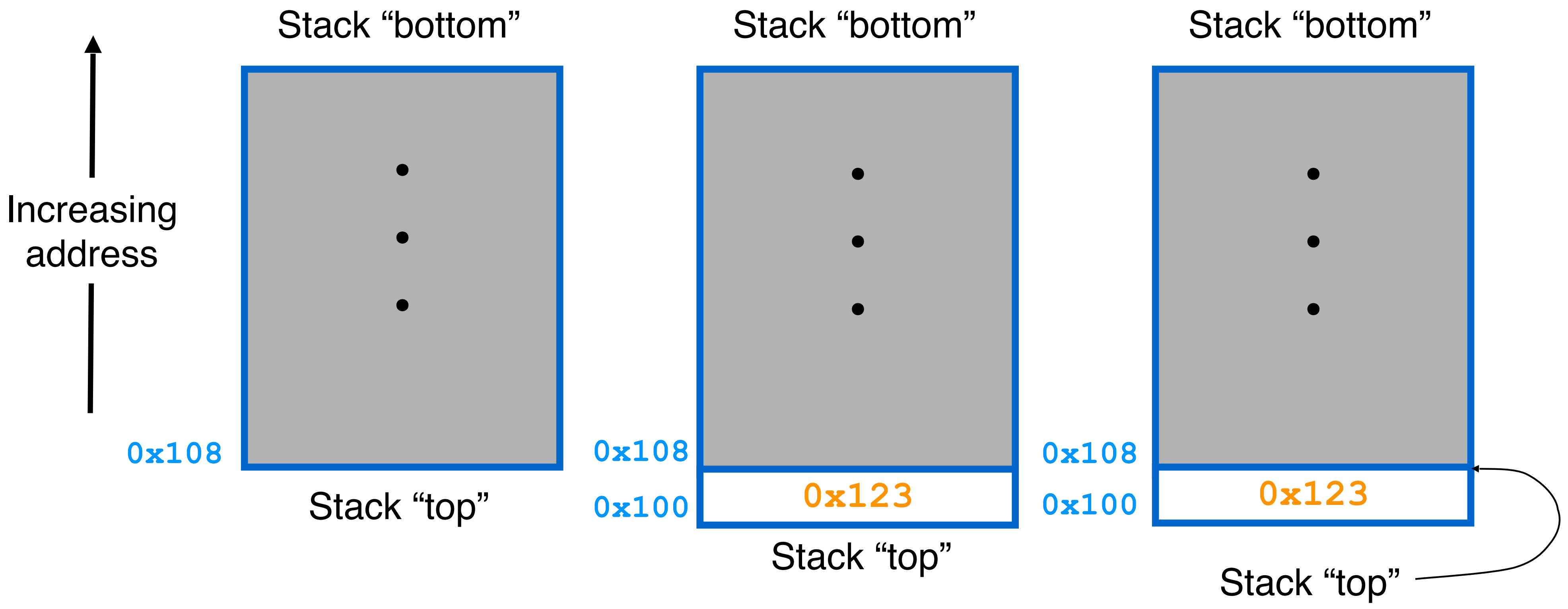
`%rax`	`0x123`
`%rdx`	`0`
`%rsp`	`0x108`

`pushq %rax`

`%rax`	`0x123`
`%rdx`	`0`
`%rsp`	`0x100`

`popq %rdx`

`%rax`	`0x123`
`%rdx`	`0x123`
`%rsp`	`0x108`



Machine Programming I: Basics

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, data movement operations
- **Arithmetic & logical operations**
- C, assembly, machine code

Address Computation Instruction

- `leaq` *Src, Dst*
  - *lea(**L**oad **E**ffective **A**ddress)*
  - *Put a memory address into a destination register (**must be a register**)*
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression
- **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

LEA used for Arithmetic Calculations

- **Example 1**

```
Long calc(long x)
{
   return 5*x+7;
}
```

Converted to ASM by compiler:

```
leaq 7(%rdi,%rdi,4), %rax
```

- **Example 2**

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

Converted to ASM by compiler:

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

Memory operands and LEA

- **In most instructions, a memory operand accesses memory (dereferences a pointer, accesses data/value at memory location)**

Assembly	C equivalent
mov 6(%rbx,%rdi,8), %ax	ax = *(rbx + rdi*8 + 6)
add 6(%rbx,%rdi,8), %ax	ax += *(rbx + rdi*8 + 6)
xor %ax, 6(%rbx,%rdi,8)	*(rbx + rdi*8 + 6) ^= ax

- **LEA is special: it *doesn't* access memory**

Assembly	C equivalent
lea 6(%rbx,%rdi,8), %rax	rax = rbx + rdi*8 + 6

Why use LEA?

- **CPU designers' intended use: calculate a pointer to an object**
  - An array element, perhaps
  - For instance, to pass just one array element to another function

Assembly	C equivalent
lea (%rbx,%rdi,8), %rax	**rax = &rbx[rdi]**

- **Compiler authors like to use it for ordinary arithmetic**
  - It can do complex calculations in one instruction
  - It's one of the only three-operand instructions the x86 has (slow operation)
  - It doesn't touch the *condition codes* (we'll come back to this)

Assembly	C equivalent
lea (%rbx,%rbx,2), %rax	**rax = rbx * 3**

Some Arithmetic Operations

- **Two Operand Instructions (binary operations):**

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	***Also called shlq***
`sarq`	*Src,Dest*	Dest = Dest >> Src	***Arithmetic***
`shrq`	*Src,Dest*	Dest = Dest >> Src	***Logical***
`xorq`	*Src,Dest*	Dest = Dest ^ Src	
`andq`	*Src,Dest*	Dest = Dest & Src	
`orq`	*Src,Dest*	Dest = Dest | Src	

- **Watch out for argument order! *Src,Dest***
  (Warning: Intel docs use "op *Dest,Src*")
- **No distinction between signed and unsigned int (why?)**

Some Arithmetic Operations

- One Operand Instructions (unary operations)

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

- See book for more instructions

Arithmetic Expression 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;
}
```

```
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
```

Initially:

x in rdi

y in rsi

z in rdx

Interesting Instructions

- `leaq`: address computation
- `salq`: shift
- `imulq`: multiplication
  - But, only used once

Understanding Arithmetic Expression 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;
}
```

```
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
```

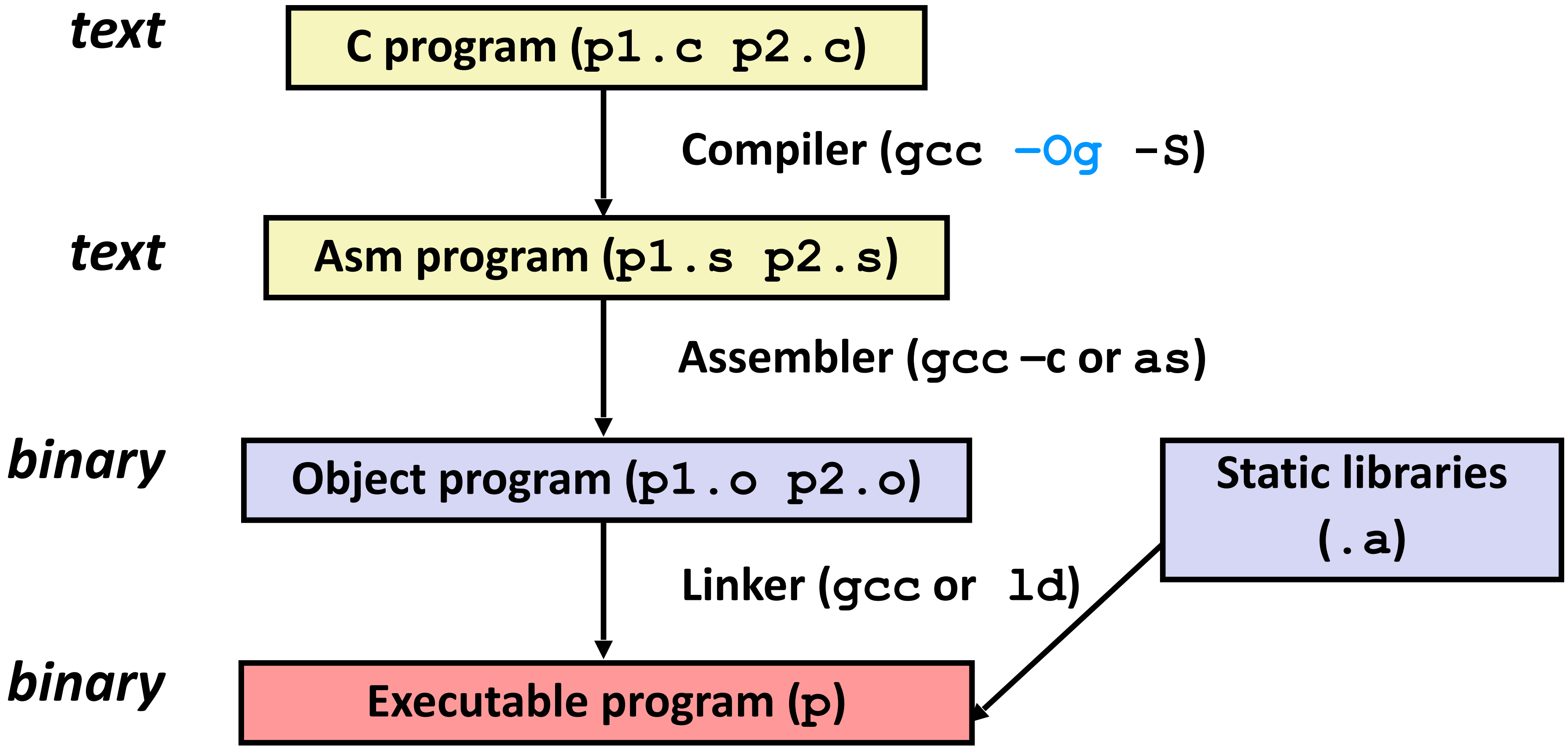
Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rdx`	Argument **`z`**, **`t4`**
`%rax`	**`t1`**, **`t2`**, **`rval`**
`%rcx`	**`t5`**

Machine Programming I: Basics

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, data movement operations
- Arithmetic & logical operations
- **C, assembly, machine code**

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use **debugging-friendly** optimizations (`-Og`)
  - Put resulting binary in file `p`



Compiling Into Assembly

C Code (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

Generated x86-64 Assembly

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

Obtain with command

```
gcc –Og –S sum.c
```

Produces file `sum.s`

***Warning*: Will get very different results on different machines (Andrew Linux, Mac OS-X, …) due to different versions of gcc and different compiler settings.**

What it really looks like

```
        .globl  sumstore
        .type   sumstore, @function
sumstore:
.LFB35:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    plus
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE35:
        .size   sumstore, .-sumstore
```

Things that look weird and are preceded by a '.' are generally directives (notes to the assembler, not translated to machine code).

```
sumstore:
   pushq    %rbx
   movq     %rdx, %rbx
   call     plus
   movq     %rax, (%rbx)
   popq     %rbx
   ret
```

Object Code

Code for `sumstore`

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

- **Total of 14 bytes**
- **Each instruction 1, 3, or 5 bytes**
- **Starts at address `0x0400595`**

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 **`malloc`**, **`printf`**
- Some libraries are ***dynamically linked***
  - Linking occurs when program begins execution

Machine Instructions

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e:   48 89 03
```

- **C**
  - Store value `t` where designated by `dest`
- **Assembly**
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:

`t:`	Register `%rax`
`dest:`	Register `%rbx`
`*dest:`	Memory **M[`%rbx`]**

- **Machine**
  - 3 bytes at address `0x40059e`
  - Compact representation of the assembly instruction
  - (Relatively) easy for hardware to interpret

Machine Instructions

```
*dest = t;
```

- **C**
  - Store value `t` where designated by `dest`

```
movq %rax, (%rbx)
```

- **Assembly**
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:

`t:`	Register `%rax`
`dest:`	Register `%rbx`
`*dest:`	Memory **M[`%rbx`]**

```
0x40059e:  48 89 03
```

```
0100 1 0 0 0  10001001   00 000 011
REX  W R X B  MOV  r->x  Mod R    M
```

Assembly instruction translates to **unique opcode**
Rest encodes arguments (e.g., *R=0000 means RAX*)

- **Machine**
  - 3 bytes at address **`0x40059e`**
  - Compact representation of the assembly instruction
  - (Relatively) easy for hardware to interpret

Disassembling Object Code

Disassembled

```
0000000000400595 <sumstore>:
  400595:  53                push   %rbx
  400596:  48 89 d3          mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff    callq  400590 <plus>
  40059e:  48 89 03          mov    %rax,(%rbx)
  4005a1:  5b                pop    %rbx
  4005a2:  c3                retq
```

- **Disassembler**

  `objdump -d sum`

  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces **approximate** rendition of **assembly** code
  - *Can be run on either `a.out` (complete executable) or `.o` file*

Alternate Disassembly

Disassembled from within gdb

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- **Within gdb Debugger**
  - Disassemble procedure

```
gdb sum
disassemble sumstore
```

Alternate Disassembly

Object Code

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

Disassembled from within gdb

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- **Within gdb Debugger**
  - Disassemble procedure

```
gdb sum
disassemble sumstore
```

  - Examine the 14 bytes starting at `sumstore`

```
x/14xb sumstore
```

What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:   file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55               push   %ebp
30001001:  8
30001003:  0
30001005:  0
3000100a:  0
```

Reverse engineering forbidden by Microsoft End User License Agreement

- **Anything that can be interpreted as executable code**
- **Disassembler examines bytes and reconstructs assembly source**

AT&T format vs Intel format

```
% gcc -Og -S sumstore.c

sumstore:
   pushq    %rbx
   movq     %rdx, %rbx
   call     plus
   movq     %rax, (%rbx)
   popq     %rbx
   ret
```

```
% gcc -Og -S -masm=intel sumstore.c

sumstore:
   push     rbx #no '%' as prefix
   mov      rbx, rdx #source: rdx, destination: rbx
   call     plus
   mov      QWORD PTR [rbx], rax
   pop      rbx
   ret
```

Which numbers are pointers?

- They aren’t labeled
- You have to figure it out from context

```
(gdb)  info registers
rax        0x40057d          4195709
rbx        0x0               0
rcx        0x4005e0          4195808
rdx        0x7fffffffdc28    140737488346152
rsi        0x7fffffffdc18    140737488346136
rdi        0x1               1
rbp        0x0               0x0
rsp        0x7fffffffdb38    0x7fffffffdb38
r8         0x7ffff7dd5e80    140737351868032
r9         0x0               0
r10        0x7fffffffd7c0    140737488345024
r11        0x7ffff7a2f460    140737348039776
r12        0x400490          4195472
r13        0x7fffffffdc10    140737488346128
r14        0x0               0
r15        0x0               0
rip        0x40057d          0x40057d
```

Which numbers are pointers?

- They aren't labeled
- You have to figure it out from context
- %rsp and %rip always hold pointers

```
(gdb) info registers
rax      0x40057d          4195709
rbx      0x0               0
rcx      0x4005e0          4195808
rdx      0x7fffffffdc28    140737488346152
rsi      0x7fffffffdc18    140737488346136
rdi      0x1               1
rbp      0x0               0x0
rsp      0x7fffffffdb38    0x7fffffffdb38
r8       0x7ffff7dd5e80    140737351868032
r9       0x0               0
r10      0x7fffffffd7c0    140737488345024
r11      0x7ffff7a2f460    140737348039776
r12      0x400490          4195472
r13      0x7fffffffdc10    140737488346128
r14      0x0               0
r15      0x0               0
rip      0x40057d          0x40057d
```

Which numbers are pointers?

- They aren't labeled
- You have to figure it out from context
- %rsp and %rip always hold pointers
  - Register values that are "close" to %rsp or %rip are *probably* also pointers

```
(gdb) info registers
rax       0x40057d         4195709
rbx       0x0              0
rcx       0x4005e0         4195808
rdx       0x7fffffffdc28   140737488346152
rsi       0x7fffffffdc18   140737488346136
rdi       0x1              1
rbp       0x0              0x0
rsp       0x7fffffffdb38   0x7fffffffdb38
r8        0x7ffff7dd5e80   140737351868032
r9        0x0              0
r10       0x7fffffffd7c0   140737488345024
r11       0x7ffff7a2f460   140737348039776
r12       0x400490         4195472
r13       0x7fffffffdc10   140737488346128
r14       0x0              0
r15       0x0              0
rip       0x40057d         0x40057d
```

Which numbers are pointers?

- If a register is being *used* as a pointer…

```
Dump of assembler code for function main:
=> 0x40057d <+0>:   sub    $0x8,%rsp
   0x400581 <+4>:   mov    (%rsi),%rsi
   0x400584 <+7>:   mov    $0x400670,%edi
   0x400589 <+12>:  mov    $0x0,%eax
   0x40058e <+17>:  call   0x400460
```

Which numbers are pointers?

- **If a register is being *used* as a pointer…**
  - mov (%rsi), %rsi
  - …Then its value is *expected* to be a pointer.
    - There might be a bug that makes its value incorrect.

```
Dump of assembler code for function main:
=> 0x40057d <+0>:  sub   $0x8,%rsp
   0x400581 <+4>:  mov   (%rsi),%rsi
   0x400584 <+7>:  mov   $0x400670,%edi
   0x400589 <+12>: mov   $0x0,%eax
   0x40058e <+17>: call  0x400460
```

Which numbers are pointers?

- **If a register is being *used* as a pointer…**
  - mov (%rsi), %rsi
  - …Then its value is *expected* to be a pointer.
    - There might be a bug that makes its value incorrect.

```
Dump of assembler code for function main:
=> 0x40057d <+0>:   sub    $0x8,%rsp
   0x400581 <+4>:   mov    (%rsi),%rsi
   0x400584 <+7>:   mov    $0x400670,%edi
   0x400589 <+12>:  mov    $0x0,%eax
   0x40058e <+17>:  call   0x400460
```

- **Not as obvious with complicated address “modes”**
  - (%rsi, %rbx) – *One* of these is a pointer, we don’t know which.
  - (%rsi, %rbx, 2) – %rsi is a pointer, %rbx isn’t (why?)
  - 0x400570(, %rbx, 2) – 0x400570 is a pointer, %rbx isn’t (why?)
  - lea (anything), %rax – (anything) *may or may not* be a pointer

Machine Programming I: Summary

- **History of Intel processors and architectures**
  - Evolutionary design leads to many quirks and artifacts
- **C, assembly, machine code**
  - New forms of visible state: program counter, registers, ...
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- **Assembly Basics: Registers, operands, move**
  - The x86-64 move instructions cover wide range of data movement forms
- **Arithmetic**
  - C compiler will figure out different instruction combinations to carry out computation