CS 3650 Computer Systems – Spring 2026

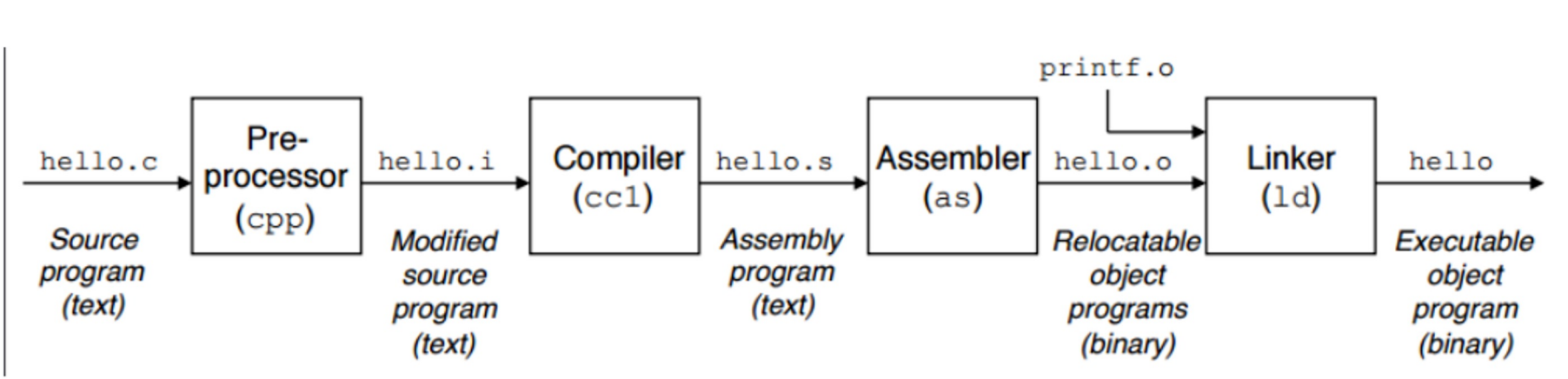
Assembly

Week 2

* Acknowledgements: created based on Mike Shah, Christo Wilson, Ferdinand Vesely, and Alden Jackson's lecture slides for the same course.

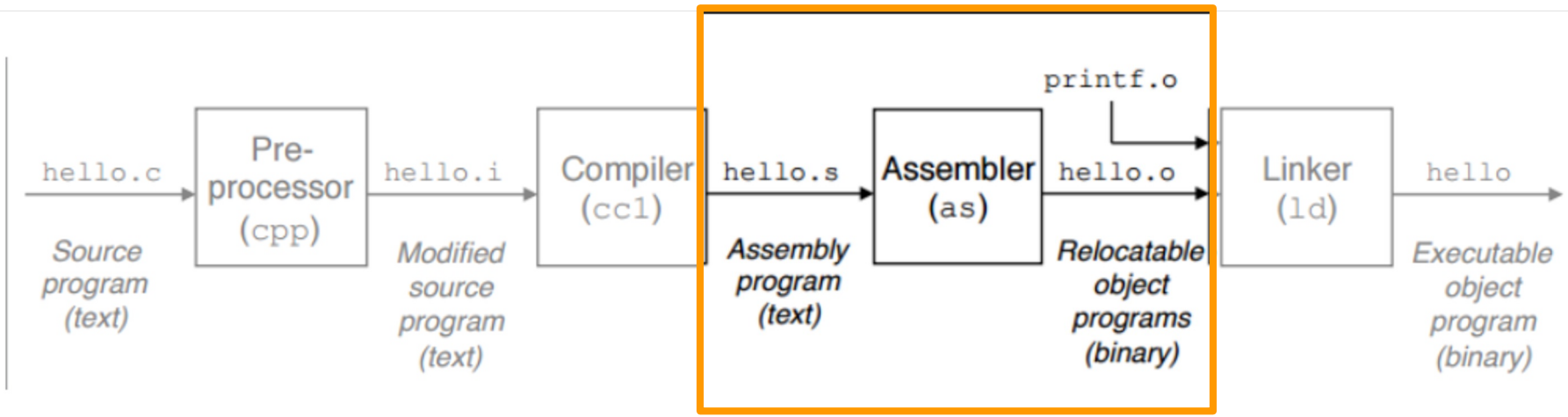
Recall the C toolchain pipeline

- All C programs go through this transformation of C --> Assembly --> Machine Code



Assembly is important in our toolchain

- Even if the step is often hidden from us!

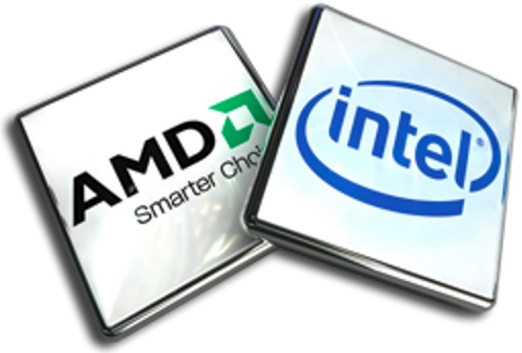


Intel and x86 Instruction set

- There is a specific instruction set to program CPUs
- Popularized by Intel
- Other companies have contributed.
  - AMD has been the main competitor
- (AMD was first to really nail 64 bit architecture around 2001)
- Intel followed up a few years later (2004)
- Intel remains the dominant architecture
- x86 is a CISC architecture
  - (CISC pronounced /ˈsɪsk/)

CISC versus RISC

- Complex Instruction Set Computer (CISC)
  - Instructions do more per operation
  - Architecture understands a series of operations
- Performance can be nearly as fast or equal to RISC



- Reduced Instruction Set Computer (RISC)
  - Instructions are very small
  - Performance is extremely fast
  - Generally a simpler architecture

Introduction to Assembly

How are programs created?

- Compile a program to an executable
  - gcc main.c -o program
- Compile a program to assembly
  - gcc main.c -S -o main.s
- Compile a program to an object file (.o file)
  - gcc -c main.c
- Linker (A program called ld) then takes all of your object files and makes a binary executable.

Focus on this step today -- pretend C does not exist

- ~~Compile a program to an executable~~
  - ~~gcc main.c -o program~~

- Compile a assembly program to an executable
  - gcc main.s -o main

- Compile an assembly program to an object file (.o file)
  - gcc -c main.s

- Linker (A program called ld) then takes all of your object files and makes a binary executable.

Layers of Abstraction

- As a C programmer you worry about C code
  - You work with variables, do some memory management using malloc and free, etc.

- As an assembly programmer, you worry about assembly
  - You also maintain the registers, condition codes, and memory

- As a hardware engineer (programmer)
  - You worry about cache levels, layout, clocks, etc.

Assembly Abstraction layer

- With Assembly, we lose some of the information we have in C

- In higher-order languages we have many different data types which help protect us from errors.
  - For example: int, long, boolean, char, string, float, double, complex, …
  - In C there are custom data types (structs for example)
  - Type systems help us avoid inconsistencies in how we pass data around.

- In Assembly we lose unsigned/signed information as well!
  - However, we do have two data types
  - Types for integers (1,2,4,8 bytes) and floats (4,8, or 10 bytes) [byte = 8 bits]
    - We are going to focus on integers in this course

Sizes of data types (C to assembly)

C Declaration	Intel Data Type	Assembly-code suffix	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

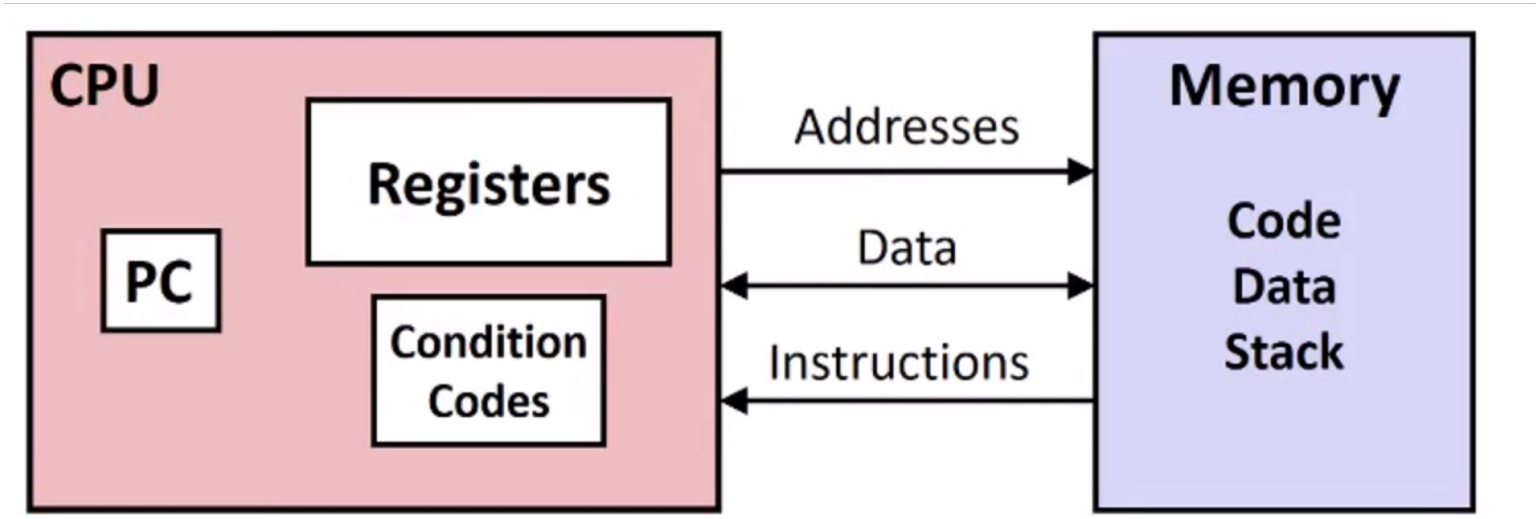
Sizes of data types (C to assembly)

C Declaration	Intel Data Type	Assembly-code suffix	Size (bytes)
char	Byte	b	1
short	Word		
int	Double word		
long	Quad word		
char *	Quad word		
float	Single precision	s	4
double	Double Precision	l	8

For us, one word of data is 64 bits
[8 bytes] but may vary on other hardware

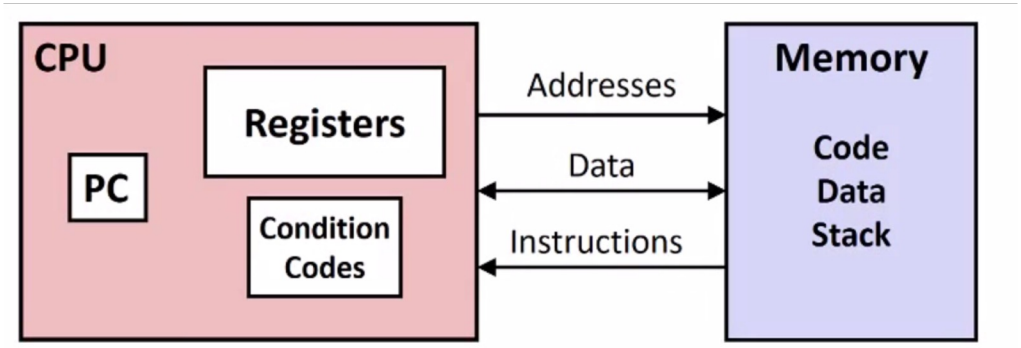
View as an assembly programmer

- Register - where we store data (heavily used data)
- PC - gives us address of next instruction
- Condition codes - some status information
- Memory – where the program (code) resides and data is stored



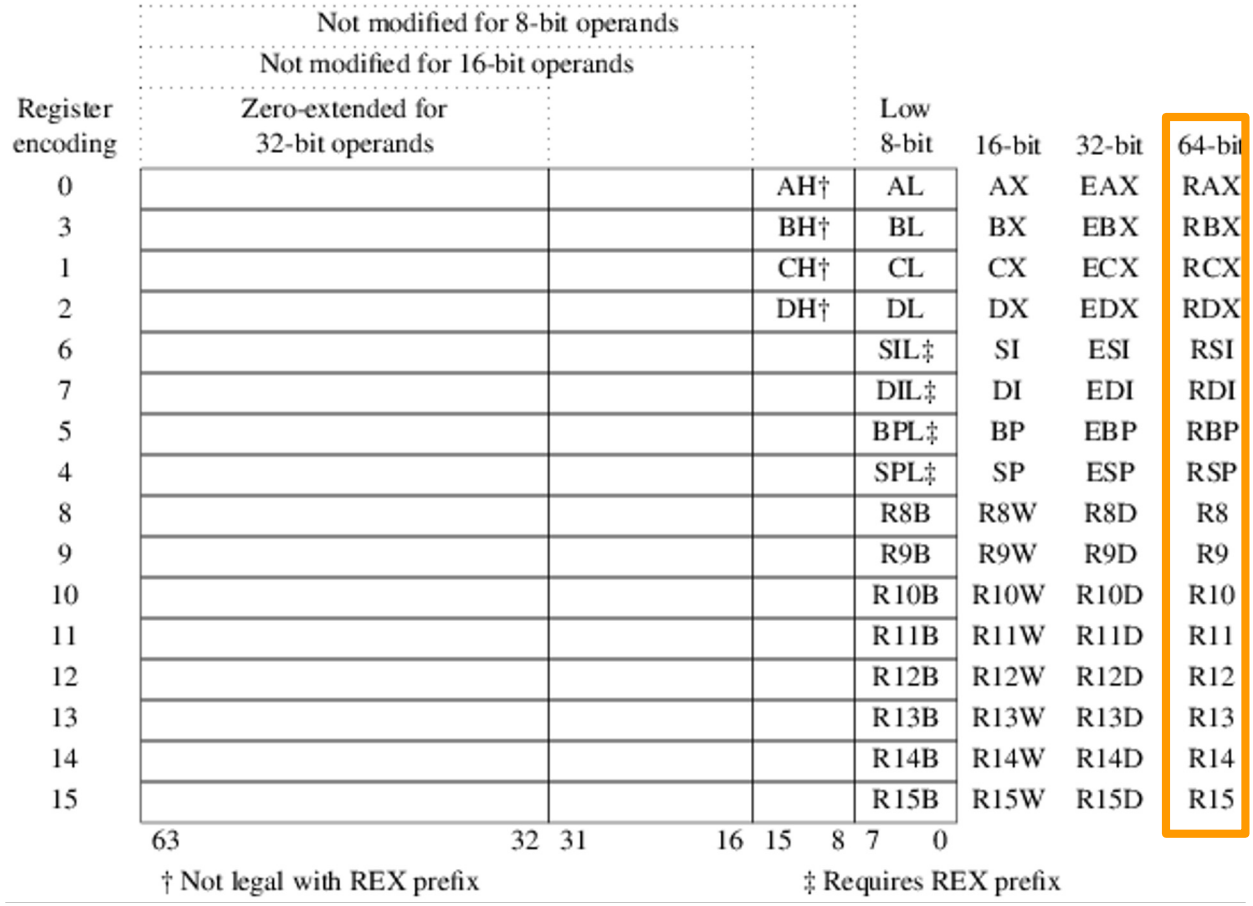
Assembly Operations (i.e. Our instruction set)

- Things we can do with assembly (and this is about it!)
  - Transfer data between memory and register
    - Load data from memory to register
    - Store register data back into memory
  - Perform arithmetic/logical operations on registers and memory
  - Transfer Control
    - Jumps
    - Branches (conditional statements)



x86-64 Registers

- Focus on the 64-bit column.
- These are 16 general purpose registers for storing bytes
  - (Note sometimes we do not always have access to all 16 registers)
- Registers are similar to variables where we store values



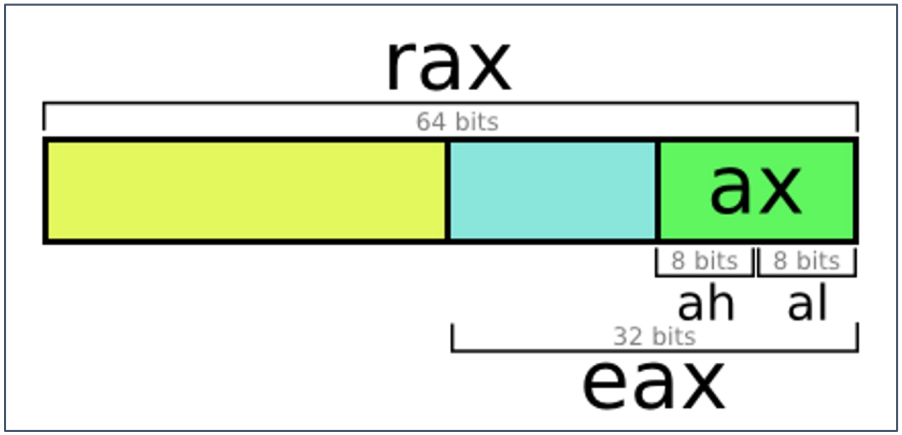
Register encoding	Zero-extended for 32-bit operands	Not modified for 8-bit operands	Not modified for 16-bit operands		Low 8-bit	16-bit	32-bit	64-bit
0				AH†	AL	AX	EAX	RAX
3				BH†	BL	BX	EBX	RBX
1				CH†	CL	CX	ECX	RCX
2				DH†	DL	DX	EDX	RDX
6					SIL‡	SI	ESI	RSI
7					DIL‡	DI	EDI	RDI
5					BPL‡	BP	EBP	RBP
4					SPL‡	SP	ESP	RSP
8					R8B	R8W	R8D	R8
9					R9B	R9W	R9D	R9
10					R10B	R10W	R10D	R10
11					R11B	R11W	R11D	R11
12					R12B	R12W	R12D	R12
13					R13B	R13W	R13D	R13
14					R14B	R14W	R14D	R14
15					R15B	R15W	R15D	R15
	63 32	31	16	15 8	7 0			

† Not legal with REX prefix

‡ Requires REX prefix

x86-64 Register (zooming in)

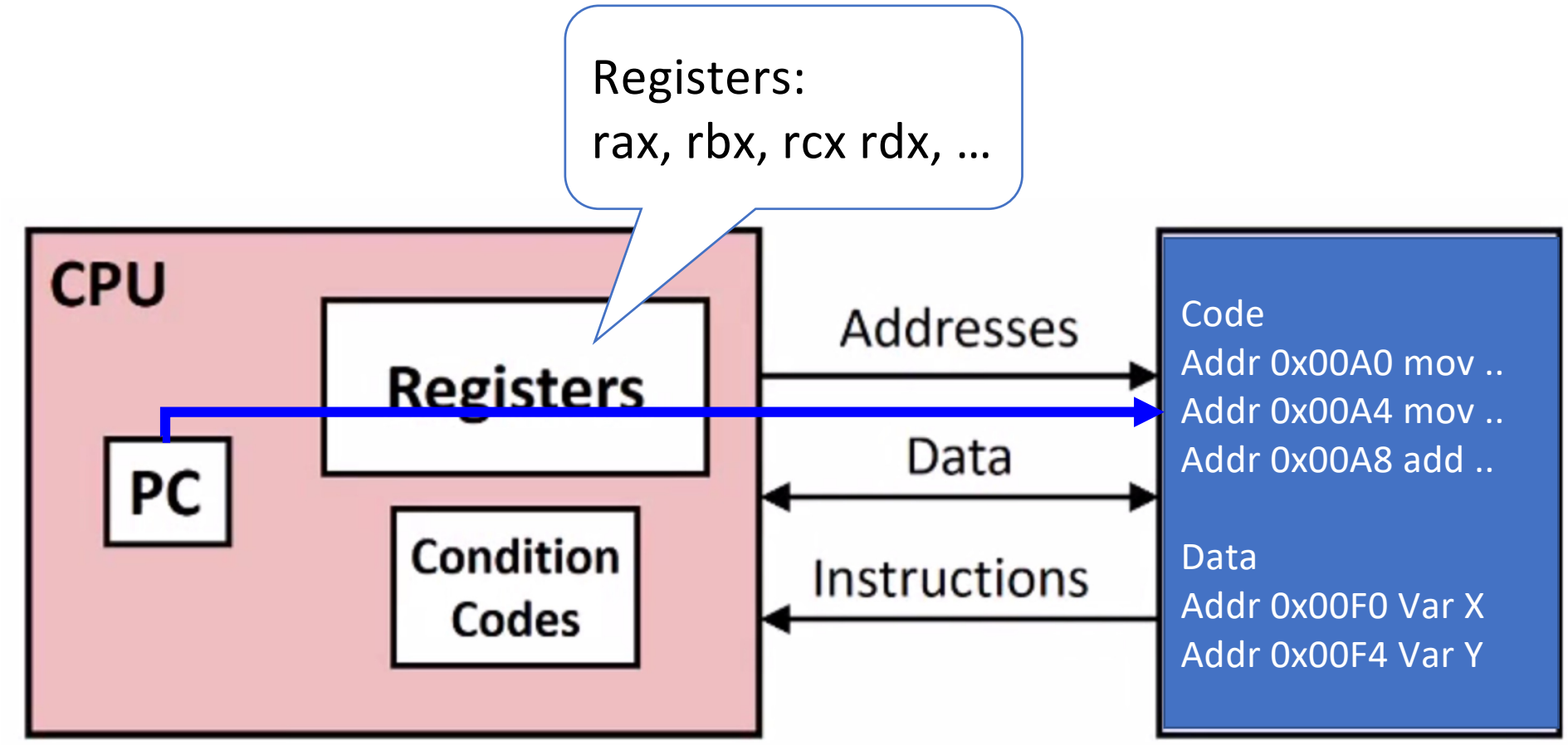
- Note register eax addresses the lower 32 bits of rax
- Note register ax addresses the lower 16 bits of eax
- Note register ah addresses the high 8 bits of ax
- Note register al (lowercase L) addresses the low 8 bits of ax



Some registers are reserved for special use (More to come)

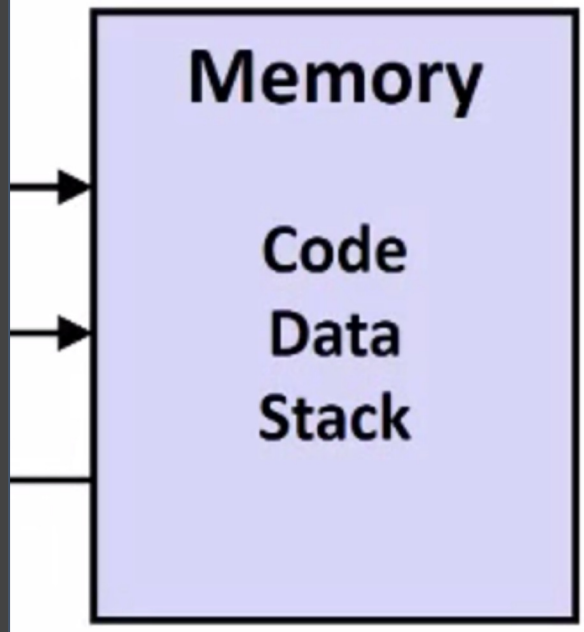
- This can be dependent on the instruction being used
  - %rsp - keeps track of where the stack pointer is
  - (We will do an example with the stack and what this means soon)

Program Counter and Memory Addresses



Memory Addresses

- Note that we are looking at virtual addresses in our assembly when we see addresses.
- This makes us think of the program as a large byte array.
  - The operating system takes care of managing this for us with virtual memory.
  - This is one of the key jobs of the operating system

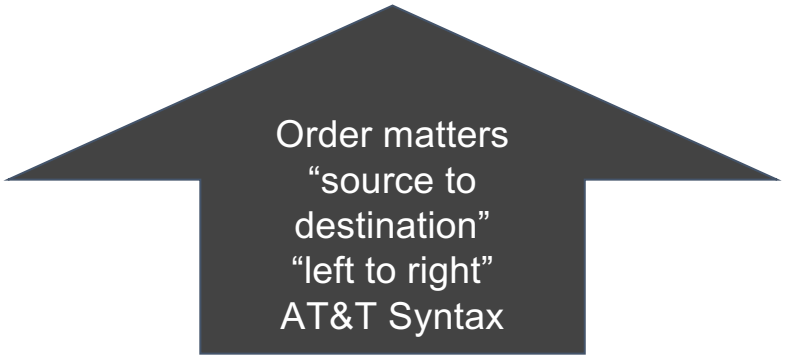


A First Assembly Instruction

Moving data around | mov instruction

- (Remember moving data is all machines do!)
- movq - moves a quad word (8 bytes) of data
- movd - move a double word (4 bytes) of data

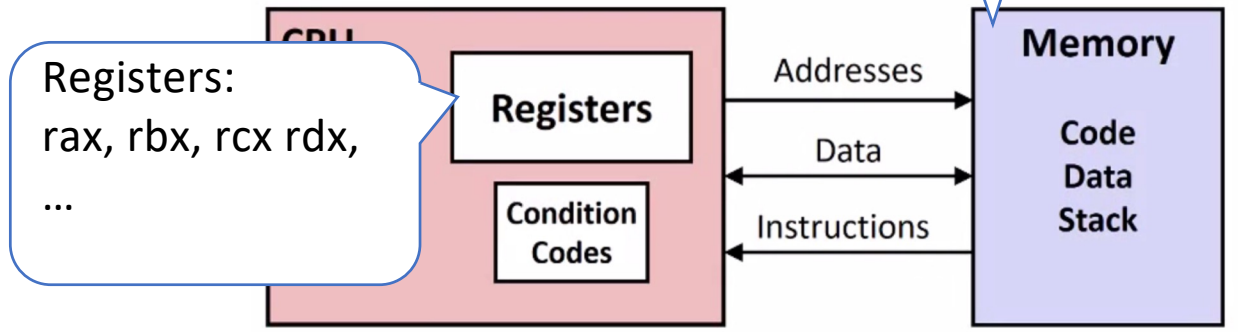
```
movq Source, Dest
```



Moving data around | mov instruction

- (Remember moving data is all machines do!)
- movq - moves a quad word (8 bytes) of data
- movd - move a double word (4 bytes) of data

```
movq Source, Dest
```



- Source or Dest Operands can have different addressing modes
  - Immediate - some memory address **$**0x333 or **$**-900
  - Memory - **(**%rax**)** dereferences gets the value in the register and use it as address
  - Register - Just **%**rax

Full List of Memory Addressing Modes

Mode	Example
Global Symbol	MOVQ x, %rax
Immediate	MOVQ $56, %rax
Register	MOVQ %rbx, %rax
Indirect	MOVQ (%rsp), %r
Base-Relative	MOVQ -8(%rbp), %rax
Offset-Scaled-Base-Relative	MOVQ -16(%rbx, %rcx, 8), %rax (*base, index, scale*)

Copy data from addr pointed by rbp minus 8 to rax

(rbx + rcx * 8) - 16

C equivalent of movq instructions | movq src, dest

```
movq $0x4, %rax
movq $-150, (%rax)
movq %rax, %rdx
movq %rax, (%rdx)
movq (%rax), %rdx
```

What does each movq do?

C equivalent of movq instructions | movq src, dest

`movq $0x4, %rax`	**`%rax`** `= 0x4;` (Moving in literal value into register)
`movq $-150, (%rax)`	use value of rax as memory location and set that location to -150 (`*p = -150`)
`movq %rax, %rdx`	`%rdx = %rax` (copy src into dest)
`movq %rax, (%rdx)`	use value of rdx as memory location and set that location to value stored in rax (`*p = %rax`)
`movq (%rax), %rdx`	Set value of rdx to value of rax as memory location (`%rdx = *p`)

Some registers are reserved for special use (More to come)

- This can be dependent on the instruction being used
- %rsp - keeps track of where the stack is for example
- %rdi - the first program argument in a function
- %rsi - the second argument in a function
- %rdx - the third argument of a function
- %rax – return value of a function

These conventions are especially useful for functions known as system calls.

1	write	sys_write	fs/read_write.c
%rdi **unsigned int** fd		%rsi **const char __user** * buf	%rdx **size_t** count

https://filippo.io/linux-syscall-table/

Some registers are reserved for special use (More to come)

- This can be dependent on the instruction being used
- %rsp - keeps track of where the stack is for example
- %rdi - the first program argument in a function
- %rsi - the second argument in a function
- %rdx - the third argument of a function
- %rax – return value of a function
- **%rip - the Program Counter**

Some registers are reserved for special use

- This can be dependent on the instruction being used
- %rsp - keeps track of where the stack is for example
- %rdi - the first program argument in a function
- %rsi - the second argument in a function
- %rdx - the third argument of a function
- %rax – return value of a function
- %rip - the Program Counter
- **%r8-%r15 - These eight registers are general purpose registers**

Success strategies

- Review the Resources posted on each Week of the Schedule
- Review my slides and the resource listing weekly
- Organize a study group
- Write (lots of) tiny test code

A little example

What does this function do? (take a few moments to think)

- void mystery(<type> a, <type> b) {

  ????

  }

- mystery:

  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret

Cheat Sheet

(Note: This can be dependent on the instruction being used)
%rsp - keeps track of where the stack is for example
%rdi - the first program argument in a function
%rsi - The second argument in a function
%rdx - the third argument of a function
%rip - the Program Counter
%r8-%r15 - These ones are actually the general purpose registers

swap of long

```
void mystery(long *a, long *b) {
        long t0 = *a;
        long t1 = *b;
        *a = t1;
        *b = t0;
}
```

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

Cheat Sheet

(Note: This can be dependent on the instruction being used)
%rsp - keeps track of where the stack is for example
%rdi - the first program argument in a function
%rsi - The second argument in a function
%rdx - the third argument of a function
%rip - the Program Counter
%r8-%r15 - These ones are actually the general purpose registers

END Lecture

More assembly instructions

```
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
shlq  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
```

- Note on order:
  We use AT&T syntax: op Src, Dest
  Intel syntax: op Dest, Src

	Value 1	Value 2
x	0110 0011	1001 0101
x>>4 (arithmetic)	**0000** 0110	**1111** 1001
x>>4 (logical)	**0000** 0110	**0000** 1001

Exercise

- If I have the expression

  c = b*(b+a)

Hint: Use BODMAS (Brackets, Orders, Division & Multiplication, Addition & Subtraction)

- How should I write this is ASM?

Cheat Sheet

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

Exercise

- If I have the expression

  c = b*(b+a)

- How should I write this in ASM?

Cheat Sheet

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

- movq a, %rax
  movq b, %rbx
  addq %rbx, %rax
  imulq %rbx
  movq %rax, c

IMULQ has a variant with one operand which multiplies by whatever is in %rax and stores result in %rax

imulq has three forms

- **imulq X : rax = X * rax**
- **imulq X Y : Y = X * Y**
- **imulq X Y Z : Z = X * Y**

Some common operations with one-operand

- incq Dest Dest = Dest + 1
- decq Dest Dest = Dest - 1
- negq Dest Dest = -Dest
- notq Dest Dest = ~Dest

More Anatomy of Assembly Programs

Assembly output of hello.c

- Lines that start with “.” are compiler directives.
  - This tells the assembler something about the program
  - .text is where the actual code starts.
- Lines that end with “:” are labels
  - Useful for control flow
  - Lines that start with . and end with : are usually temporary locals generated by the compiler.
- Reminder that lines that start with % are registers
- (.cfi stands for call frame information)

```
        .file   "hello.c"
        .text
        .globl  main
        .align  16, 0x90
        .type   main,@function
main:                                   # @main
        .cfi_startproc
# BB#0:
        pushq   %rbp
.Ltmp2:
        .cfi_def_cfa_offset 16
.Ltmp3:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
.Ltmp4:
        .cfi_def_cfa_register %rbp
        subq    $16, %rsp
        leaq    .L.str, %rdi
        movl    $0, -4(%rbp)
        callq   puts
        movl    $0, %ecx
        movl    %eax, -8(%rbp)          # 4-byte Spill
        movl    %ecx, %eax
        addq    $16, %rsp
        popq    %rbp
        ret
.Ltmp5:
        .size   main, .Ltmp5-main
        .cfi_endproc

        .type   .L.str,@object          # @.str
        .section        .rodata.str1.1,"aMS",@progbits,1
.L.str:
        .asciz  "Hello Computer Systems Fall 2022"
        .size   .L.str, 33


        .ident  "clang version 3.4.2 (tags/RELEASE_34/dot2-final)"
        .section        ".note.GNU-stack","",@progbits
```

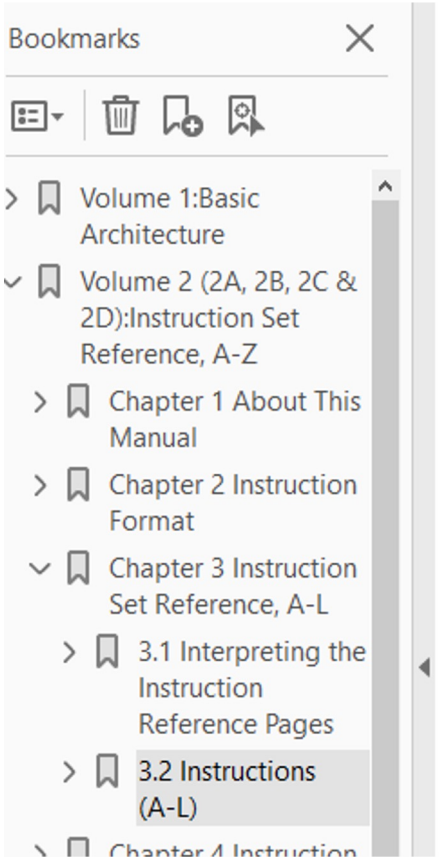
Where to Learn more?

- https://diveintosystems.org/

- Intel® 64 and IA-32 Architectures Software Developer Manuals

Document	Description
Intel® 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4	This document contains the following: **Volume 1:** Describes the architecture and programming environment of processors supporting IA-32 and Intel® 64 architectures. **Volume 2:** Includes the full instruction set reference, A-Z. Describes the format of the instruction and provides reference pages for instructions. **Volume 3:** Includes the full system programming guide, parts 1, 2, 3, and 4. Describes the operating-system support environment of Intel® 64 and IA-32 architectures, including: memory management, protection, task management, interrupt and exception handling, multi-processor support, thermal and power management features, debugging, performance monitoring, system management mode, virtual machine extensions (VMX) instructions, Intel® Virtualization Technology (Intel® VT), and Intel® Software Guard Extensions (Intel® SGX). **Volume 4:** Describes the model-specific registers of processors supporting IA-32 and Intel® 64 architectures.

(Volume 2 Instruction set reference)



INC—Increment by 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FE /0	INC *r/m8*	M	Valid	Valid	Increment *r/m* byte by 1.
REX + FE /0	INC *r/m8**	M	Valid	N.E.	Increment *r/m* byte by 1.
FF /0	INC *r/m16*	M	Valid	Valid	Increment *r/m* word by 1.
FF /0	INC *r/m32*	M	Valid	Valid	Increment *r/m* doubleword by 1.
REX.W + FF /0	INC *r/m64*	M	Valid	N.E.	Increment *r/m* quadword by 1.
40+ *rw***	INC *r16*	O	N.E.	Valid	Increment word register by 1.
40+ *rd*	INC *r32*	O	N.E.	Valid	Increment doubleword register by 1.

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

** 40H through 47H are REX prefixes in 64-bit mode.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA
O	opcode + rd (r, w)	NA	NA	NA

Description

Adds 1 to the destination operand, while preserving the state of the CF flag. The destination operand can be a

So far we looked at moving data and doing some operations on data

What's missing?

Comparisons

Compare operands: cmp_, jmp_, set__

- Often we want to compare the values of two registers
  - Think if, then, else constructs or loop exit or switch conditions

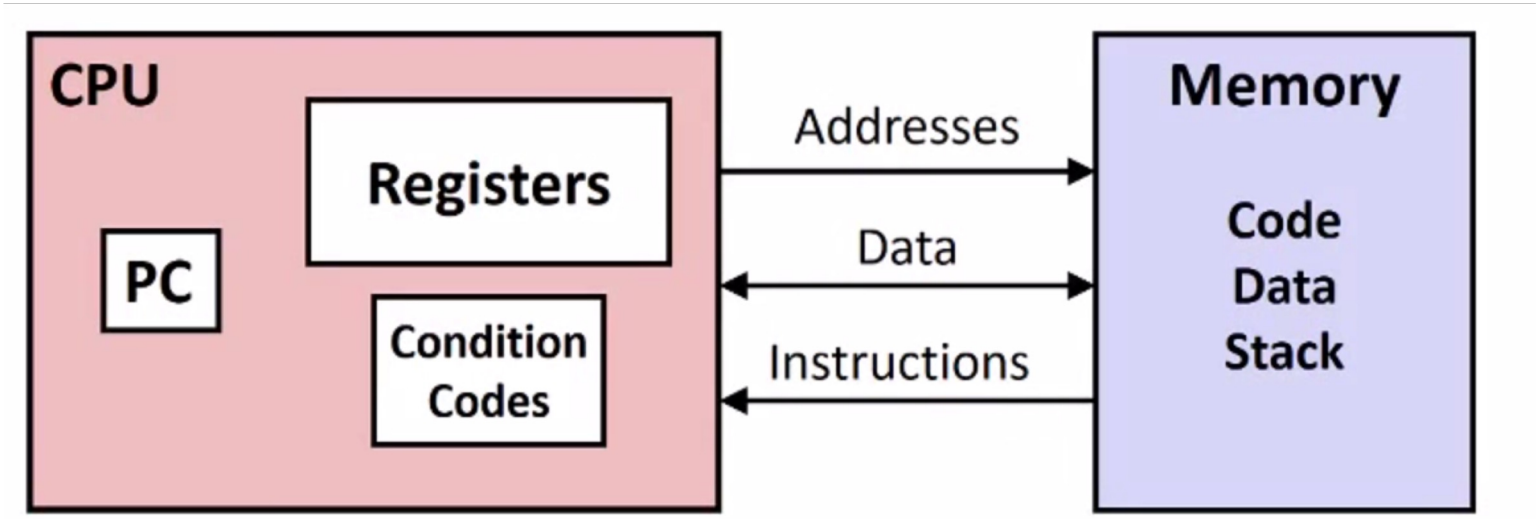
- cmpq Src2, Src1
  - cmpq Src2, Src1 is equivalent to computing Src1-Src2 (but there is no destination register)

- Now we need a method to use the result of compare, but there is not destination to find the result.

What do we do?

Remember condition codes?

- Register - where we store data (heavily used data)
- PC - gives us address of next instruction
- **Condition codes - some status information**
- Memory – where the program (code) resides and data is stored



FLAGS registers

- CF (carry flag)
  - Set to 1 when there is a carry out in an unsigned arithmetic operation
  - Otherwise set to 0
- ZF (zero flag)
  - Set to 1 when the result of an arithmetic operation is zero
  - Otherwise set to 0
- SF (signed flag)
  - Set to 1 when there is a carry out in a signed arithmetic operation
  - Otherwise set to 0
- OF (overflow flag)
  - Set to 1 when signed arithmetic operations overflow
  - Otherwise set to 0

Conditional Branches (jumps)

Using the result from cmp => jmp instructions

- In order to read result from cmp, we use jmp to a label

Instruction		Description
jmp	*Label*	Jump to label
jmp	**Operand*	Jump to specified location
je / jz	*Label*	Jump if equal/zero
jne / jnz	*Label*	Jump if not equal/nonzero
js	*Label*	Jump if negative
jns	*Label*	Jump if nonnegative
jg / jnle	*Label*	Jump if greater (signed)
jge / jnl	*Label*	Jump if greater or equal (signed)
jl / jnge	*Label*	Jump if less (signed)
jle / jng	*Label*	Jump if less or equal
ja / jnbe	*Label*	Jump if above (unsigned)
jae / jnb	*Label*	Jump if above or equal (unsigned)
jb / jnae	*Label*	Jump if below (unsigned)
jbe / jna	*Label*	Jump if below or equal (unsigned)

Jump instructions | Typically used after a compare

	Condition	Description
jmp	1	unconditional
je	ZF	jump if equal to 0
jne	~ZF	jump if not equal to 0
js	SF	Negative
jns	~SF	non-negative
jg	~(SF^OF) & ~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal
jl	(SF^OF)	Less (Signed)
jle	(SF ^ OF) | ZF	Less or Equal
ja	~CF & ~ZF	Above (unsigned)
jb	CF	Below (unsigned)

Conditional Branch | if-else

- long absoluteDifference (long x, long y) {
  long result;

  if (x > y)
  result = x-y;

  else
  result = y-x;

  }

Take a moment to think about the ASM code

- absoluteDifference:

```
        cmpq    %rsi, %rdi
        jle     .else
        movq    %rdi,   %rax
        subq    %rsi,   %rax
        ret
.else:
        movq    %rsi,   %rax
        subq    %rdi,   %rax
        ret
```

Some reminders:

%rdi = argument x (first argument)
%rsi = argument y (second argument)
%rax = return value
cmpq src2, src1 = src1 – src2 and sets flags
jle x = jump to x if less than or equal

Code Exercise
(Take a moment to think what this assembly does)

```
      movq      $0, %rax
mystery:
      incq      %rax
      cmpq      $5, %rax
      jl        mystery
```

Code Exercise | Annotated (while loop example)

```
    movq    $0, %rax
mystery:
    incq    %rax
    cmpq    $5, %rax
    jl      mystery
```

- Move the value 0 into %rax (temp = 0)
- Increment %rax (temp = temp + 1;)
- Compare %rax with 5
- If %rax is smaller than 5 then
  jump to ‘mystery’
  If not then
  proceed

Code Exercise | Annotated (while loop example)

```
      movq      $0, %rax
mystery:
      incq      %rax
      cmpq      $5, %rax
      jl        mystery
```

Equivalent C Code

```
long temp = 0;
do {
        temp = temp + 1;
   }
while(temp < 5);
```

- Move the value 0 into %rax (temp = 0)
- Label of a location
- Increment %rax (temp = temp + 1;)
- Compare %rax with 5
- If %rax is smaller than 5 then
  jump to ‘mystery’
  If not then
  proceed