

# CSE 2421

## X86 Assembly Language – Part 1

- Required Reading: *Computer Systems: A Programmer's Perspective, 3<sup>rd</sup> Edition*
  - Chapter 3 thru 3.2.1 (inclusive), Section 3.3 through 3.4.2 (inclusive)

# Quotes from previous semesters\*

- ▶ “I ain’t gonna lie, I really like this stuff”
- ▶ “Don’t tell anyone I said this, but I really like X86!”
- ▶ “There’s so much more control! This is neat!”
- ▶ “I hate it! I hate it! I hate it!”
- ▶ “Once you figure it out, it’s actually kinda fun!”

▶ \*One of these I made up, guess which one

# Computer Architecture

- The modern meaning of the term *computer architecture* covers three aspects of computer design:
  - *instruction set architecture*,
  - *computer organization* and
  - *computer hardware*.
- *Instruction set architecture* – **ISA** refers to the actual programmer visible machine interface such as instruction set, registers, memory organization, and exception (i.e. interrupt) handling.

# Instruction Set Architecture

## Assembly Language View

Processor state

Registers, memory, ...

Instructions

`addq, pushq, ret, ...`

How instructions are encoded as bytes

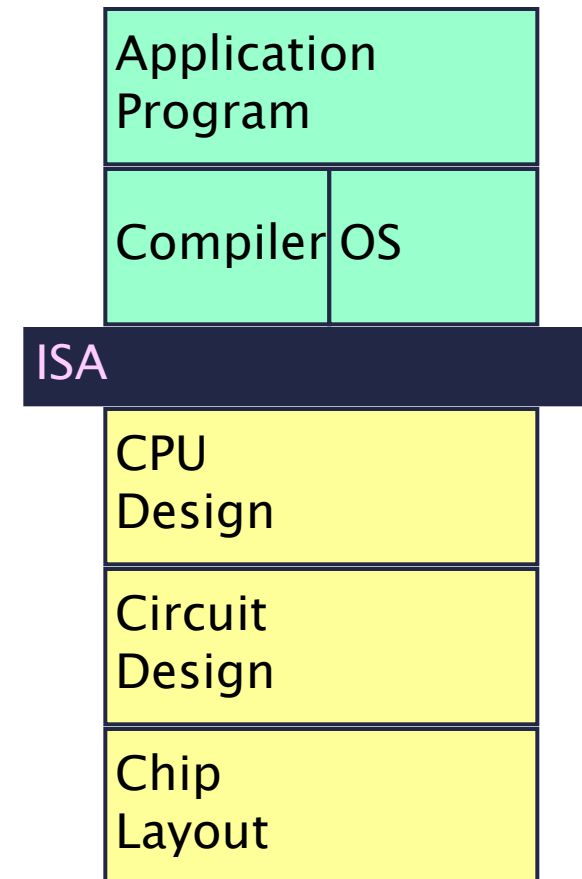
## Layer of Abstraction

Above: how to program machine


Processor executes instructions in a sequence

Below: what needs to be built

Use variety of tricks to make it run fast  
E.g., execute multiple instructions simultaneously



# Instruction Set Architecture

- Types of instruction set architectures
    - RISC (Reduced Instruction Set Computer) architecture
    - CISC (Complex Instruction Set Computer) architecture
- 


# RISC ISA

- Fixed length encoding (All instructions are the same length)
- Simple addressing modes, typically only base and displacement
- Arithmetic & logical operations (ALU operations) work on data in registers only
- The only instructions that can affect memory are load and store instructions
  - Move data from memory to a register (load) or to memory from a register (store), respectively; this is called load–store architecture
- No condition registers on early designs
- RISC processors tend to use less power and thus generate less heat
- Large number of registers (32, 64 or 128 is typical)
- Register intensive procedural linkage
  - Registers used for procedure arguments, return values and addresses
- **All processors in smart phones and tablets are of RISC architecture.**

# CISC ISA

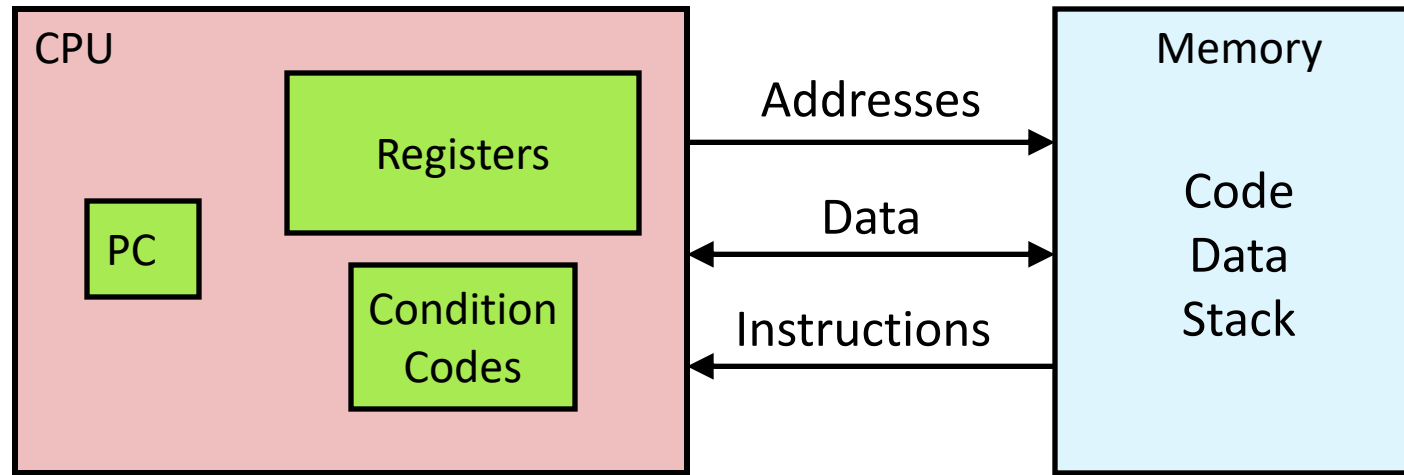
- Variable length encoding (instruction length varies)
  - Intel Architecture 32 bit (IA32) instruction length ranges from 1 to 15 bytes
- More addressing modes
  - X64 supports displacement, base, index, registers, scale factors, etc.
- Arithmetic and logic operations can be performed on registers *or directly on memory*
- Condition codes hold the side effects of instructions
- Use more power and generate more heat, but no one cares unless it's in a laptop  
(well, maybe the person who pays the electric bill)
- Stack-intensive procedure linkages
  - Stack is used for procedural arguments and return address/values
- This is the ISA for **Intel IA-32 processors**, and also the basis for the related 64 bit versions of the ISA (these processors are now found in over 90% of laptop and desktop computers).

# Assembly

- There are many different kinds of assembly languages
  - Each different type of processor can have a different one
  - We're going to use X86-64 because it's the processor that stdlinux is based on.
- 



# 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

# 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

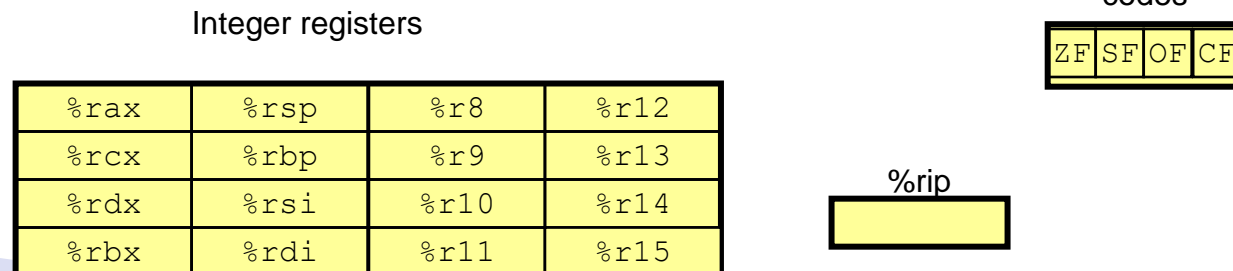
*Warning:* Will get very different results on different machines (Linux, Mac OS-X, ...) due to different versions of gcc and different compiler settings.

**WARNING: Do not turn in machine generated assembly code in labs. Doing so is a CoAM offense.**

# X86 programmer-visible state

The X86-64 CPU has:

- . 16 64-bit integer registers (we are ignoring floating point)
- . 4 condition codes (RFLAGS/EFLAGS) that we care about:
  - . They are single-bit flags set by arithmetic or logical instruction
  - . ZF(zero), SF(negative), OF(overflow) , CF (carry)
- . – A program counter (%rip)
  - . – Holds the address of the next instruction
- . – Memory: Consider it to be a byte-addressable storage array. Words stored in little-endian byte order



# Condition Codes

- 4 condition codes in x86 of concern to us
  - ZF – Set if the result of the last ALU operation (arithmetic/logical operation) is 0
  - SF – Set if the result of the last ALU operation resulted in the sign bit (msb, or most significant bit) of the result being set (that is, equal to 1)
  - OF – Set if the result of the last ALU operation resulted in overflow for a signed operation
  - CF – Set if the result of the last ALU operation resulted in overflow for an unsigned operation

NOTE: 1) only ALU operations set condition codes

2) ALU doesn't know if operation is signed or unsigned


# The Bigger Picture

- ▶ Four General Categories of Statements in Computer Languages
  - Declarations (optional in some languages like Python)
  - Data Movement
    - Memory to function variables (registers in assembler)
    - Function variables (register) to function variables (register)
    - Function variables (register) to memory
  - Arithmetic/Logical Operations
    - Compare something
    - Calculate something
  - Control-Flow
    - Procedure/function calls
    - Looping
    - Conditionals

# Syntax – AT&T vs. Intel

- ▶ We will only learn AT&T syntax for x86.
- ▶ Intel syntax is “backwards” to AT&T syntax
- ▶ The gcc compiler uses AT&T syntax
- ▶ Be careful about syntax when using other resources such as from the web

# x86 Instructions

- ▶ Three categories
    - Data Movement
    - Arithmetic/Logical (ALU) operations
    - Control-Flow
  - ▶ We will not cover ALL x86 instructions (there are hundreds); there are numerous obscure ones.
  - ▶ We will cover many of the common instructions.
  - ▶ For full list of instructions see Intel's instruction set reference.
- 

# x86 Instruction Overview

- ▶ Operands can be of different sizes: 1, 2, 4 or 8 bytes
  - Data Movement and Arithmetic/Logic operations in AT&T format use an instruction opcode *suffix* (q, b, w, or l) to specify operand size.
  - Because of this, we also need registers (or parts of registers) which have a size of 1, 2, 4, or 8 bytes (see later slide).
- ▶ X86-64 ISA uses one bit *flags* (z, s, c and o) in the RFLAGS register (more on this later); to signify that the result of the most recent ALU operation was 0, negative, carry out or resulted in overflow.



# Sizes of operands

- ▶ In AT&T format, the size of the operands is specified with suffixes, q, l, w, and b appended to the opcode
  - q – quad word, or 64 bit operand
  - l – long word, or 32 bit operand
  - w – word, or 16 bit operand
  - b – byte, or 8 bit operand
- ▶ Most instructions that we will use will have the q suffix for 64 bit operands, but we will make some use of 1 byte, 2 byte and 4 byte operands.
- ▶ If you omit the suffix, the assembler will attempt to determine it from the operands involved, but this is *dangerous* (the assembler will not always generate a machine instruction that operates on data of the size you want). This can cause bugs which are extremely difficult to track down, so be sure to include the operand size suffix *always*!
- ▶ *Opcodes in the syntax information below are given without suffixes; you must add the appropriate suffix when you write assembly language instructions. The examples show opcodes with operand size suffixes.*


# Verbiage

- ▶ Opcode operand1, operand2, operand3
  - Opcode is the “name” of the instruction in assembly language, which does a certain kind of operation on the processor: for example, *mov* (which moves data), *add*, *jmp*, etc.
  - We will use this shorthand for instruction operands: op1, op2, op3
- ▶ Which operands are required or optional depends on the opcode (covered in the slides below for each of the opcodes or instructions that we will look at)
- ▶ *movq %rax, %rbx*
  - Opcode *with operand size suffix* – *movq*
  - Operand1 (op1) – *rax*
  - Operand2 (op2) – *rbx*

# Assembly Characteristics: Data Types

- ▶ “Integer” data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (*untyped* pointers)
- ▶ Floating point data of 4, 8, or 10 bytes
- ▶ Code: Byte sequences encoding series of instructions
- ▶ No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
  - That doesn't mean we can't work something out. 😊

# Assembly Characteristics: Operations

- ▶ Perform arithmetic function on register or memory data
  - ▶ Transfer data between memory and register
    - Load data from memory into register
    - Store register data into memory
  - ▶ Transfer control
    - Unconditional jumps to/from procedures
    - Conditional branches
- 

# Hold This Thought...



# x86-64 Integer Registers\*

<b>%rax</b>	<b>%eax</b>	<b>%ax</b>
<b>%rbx</b>	<b>%ebx</b>	<b>%bx</b>
<b>%rcx</b>	<b>%ecx</b>	<b>%cx</b>
<b>%rdx</b>	<b>%edx</b>	<b>%dx</b>
<b>%rsi</b>	<b>%esi</b>	<b>%si</b>
<b>%rdi</b>	<b>%edi</b>	<b>%di</b>
<b>%rsp</b>	<b>%esp</b>	<b>%sp</b>
<b>%rbp</b>	<b>%ebp</b>	<b>%bp</b>

<b>%r8</b>	<b>%r8d</b>	<b>%r8w</b>
<b>%r9</b>	<b>%r9d</b>	<b>%r9w</b>
<b>%r10</b>	<b>%r10d</b>	<b>%r10w</b>
<b>%r11</b>	<b>%r11d</b>	<b>%r11w</b>
<b>%r12</b>	<b>%r12d</b>	<b>%r12w</b>
<b>%r13</b>	<b>%r13d</b>	<b>%r13w</b>
<b>%r14</b>	<b>%r14d</b>	<b>%r14w</b>
<b>%r15</b>	<b>%r15d</b>	<b>%r15w</b>

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- \*See Figure 3.2, page 180 of Bryant/O'Halloran for 1 byte register names

# History: IA32 Registers

Origin  
(mostly obsolete)

general purpose	%eax	%ax	%ah	%al	accumulate
	%ecx	%cx	%ch	%cl	counter
	%edx	%dx	%dh	%dl	data
	%ebx	%bx	%bh	%bl	base
	%esi	%si			source index
	%edi	%di			destination index
	%esp	%sp			stack pointer
	%ebp	%bp			base pointer
16-bit virtual registers (backwards compatibility)					

# X86 Registers

- ▶ rax, rbx, rcx, rdx
  - These registers can be accessed in parts in x86-64 (replace R below by a, b, c, or d):
    - *rRx* – Refers to 64 bit register
    - *eRx* – Refers to 32 bit register
    - *Rx* – Refers to the lower (least significant) 16 bits of *eRx*
    - *Rh* – Refers to the top (most significant) 8 bits of the *Rx* bit register (buggy)
    - *Al* – Refers to the lower 8 bits of the *Rx* register
- ▶ The least significant (or lower) 16 bits of rdi and rsi can be accessed also in x86-64
  - rdi, rsi: 64 bit registers



# X86 Registers

- ▶ rax, rbx, rcx, rdx, rdi and rsi, and their sub-parts are used as general purpose registers
  - Can be used to store integer and character data, and also addresses
- ▶ rsp and rbp are used as stack management registers, and should not be used for any other purpose
  - As discussed previously, rsp is the address of the top of the stack (it points to the last value pushed onto the stack)
  - rbp is a frame pointer: it points to the bottom of the stack frame of the function which is currently executing.
    - rbp must be set at the beginning of the function, after preserving the value of rbp for the caller function.
    - We will see how this is done in X86-64 soon.

# Moving Data

## ▶ Moving Data

`movq Source, Dest`

## ▶ Operand Types

- **Immediate:** Constant integer data
  - Example: `$0x400`, `$-533`
  - Like C constant, but prefixed with ``$'`
  - Encoded with 1, 2, 4 or 8 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 “address modes”

`%rax`

`%rcx`

`%rdx`

`%rbx`

`%rsi`

`%rdi`

`%rsp`

`%rbp`

`%rN`

# 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*

# Simple Memory Addressing Modes

## ▶ Normal (R) $\text{Mem}[\text{Reg}[R]]$

- Register R specifies memory address
- Aha! Pointer dereferencing in C

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

## ▶ Displacement D(R) $\text{Mem}[\text{Reg}[R]+D]$

- Register R specifies start of memory region
- Constant displacement D specifies offset

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

# Run X86 program

**A first x86  
program:**

```
movq $55,%rdx
movq %rdx, %rbx
movq $Array, %rax
movq %rbx,8(%rax)
movq 0(%rax),%rcx
.align 8
Array:
.quad 0x6f
.quad 0x84
```

**NOTE: 55 = 0x37**