NEU CS 3650 Computer Systems

Instructor: Dr. Ziming Zhao

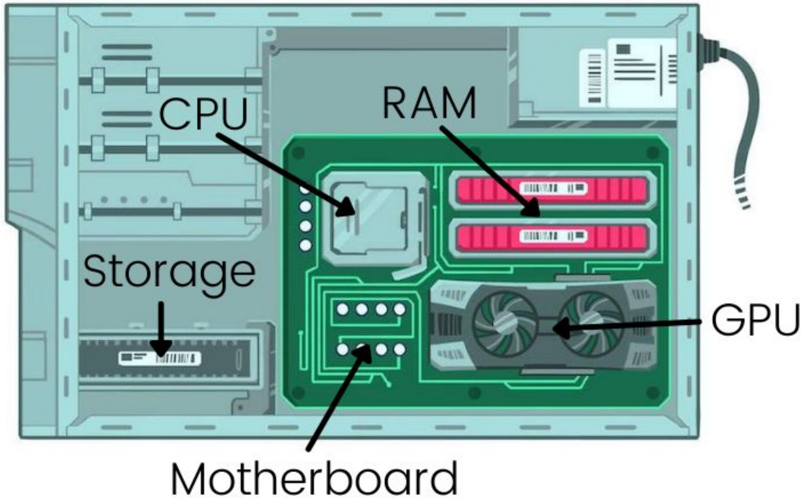
* Acknowledgements: created based on Christo Wilson, Ferdinand Vesely, Alden Jackson, Ben Weintraub, Gene Cooperman, Peter Desnoyers' lecture slides for the same course.

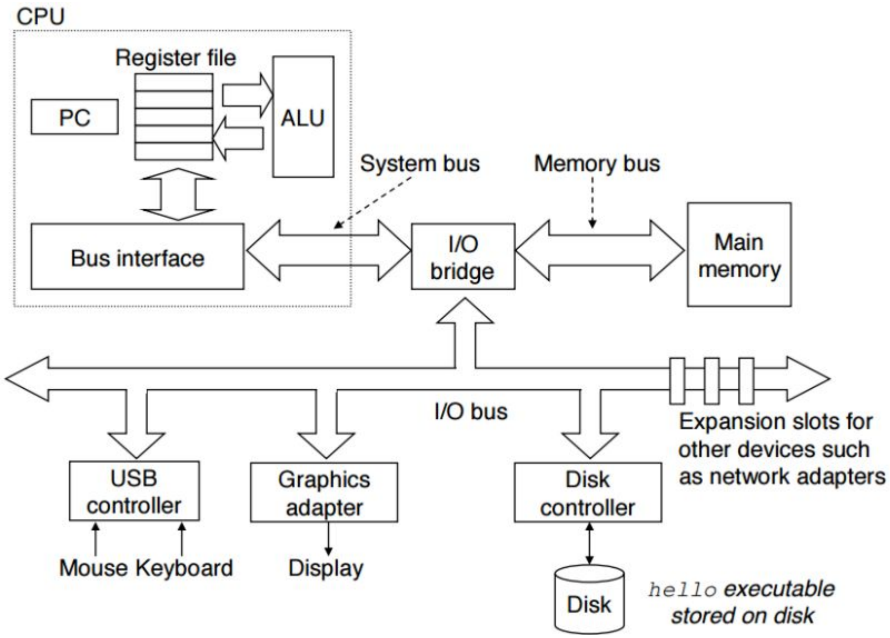
CPU Architectures and Assembly Languages

Agenda

1. Overview - Architecture and Assembly
2. The x86 32-bit architecture and its instruction set (Complex instruction set computer; CISC)
3. The x86-64 architectures and its instruction set (CISC)
4. The ARM Cortex-A and Cortex-M and their instruction sets (Reduced instruction set computer; RISC)

Computer Organization





What is Computer Architecture?

- Defines the design and organization of a processor (CPU, GPU, etc.)
- Specifies how the processor executes instructions, handles data, and interacts with memory and I/O

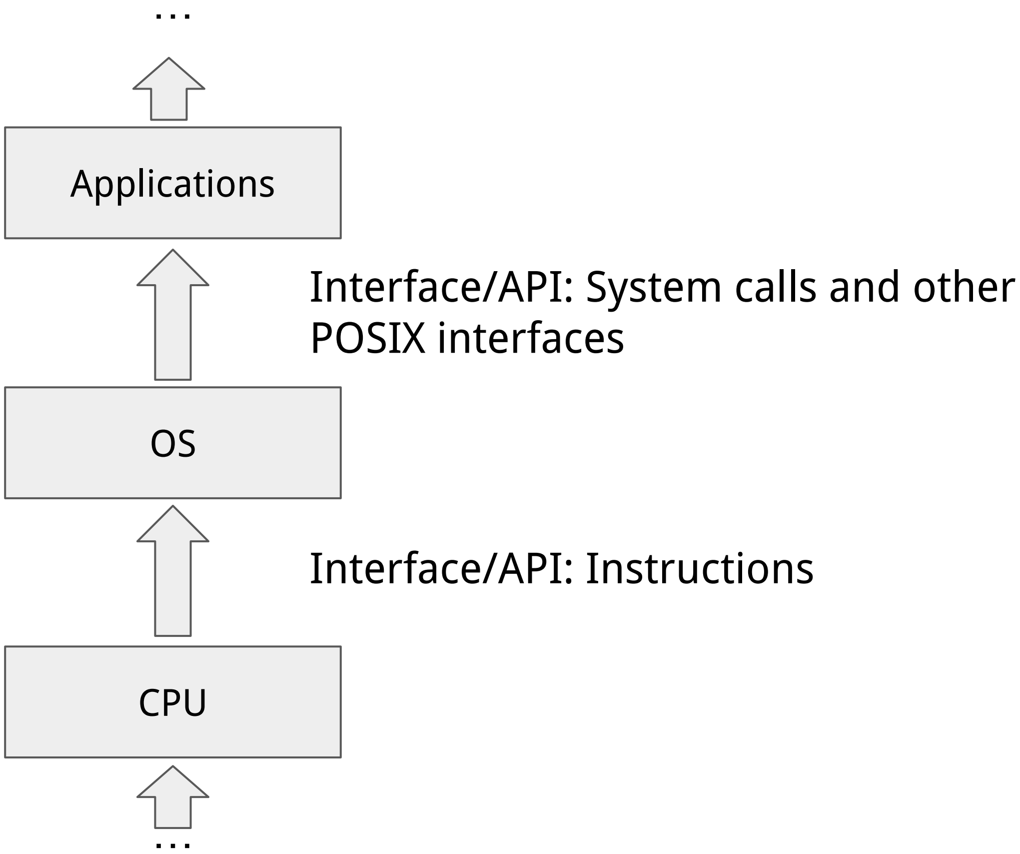
What are Instructions?

- Binary-encoded operations the processor understands directly.
- Tell the CPU what action to perform
- Form the Instruction Set Architecture (ISA), the “API” between hardware and software

What is Assembly Language?

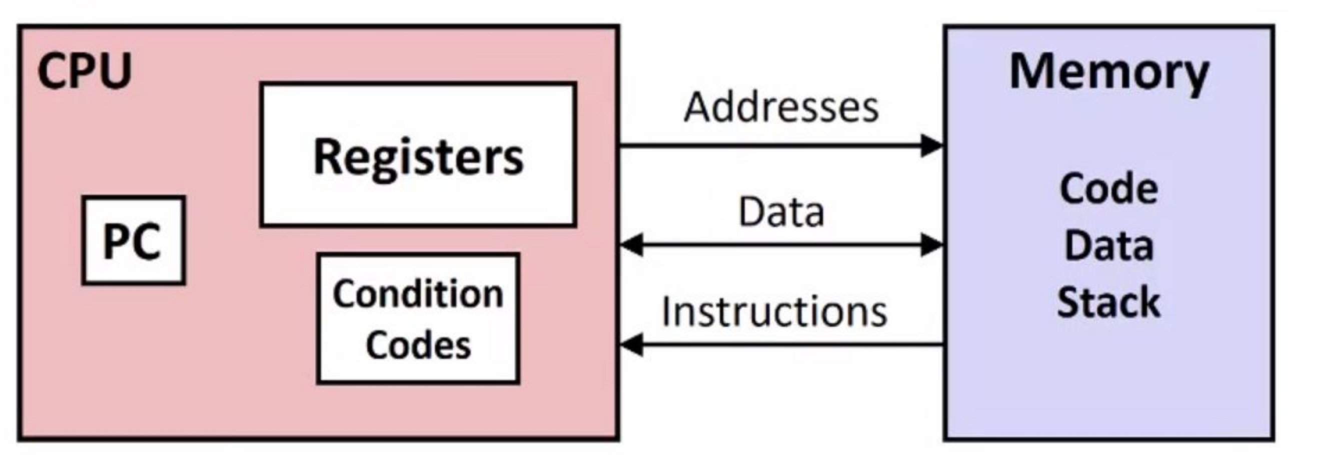
- A human-readable representation of machine instructions
- Uses mnemonics (e.g., ADD, MOV, JMP) instead of raw binary opcodes
- Provides a low-level programming interface closely tied to the CPU’s ISA
- Still requires an assembler to convert into machine code

The CPU provides an interface to software in the form of its *instruction* set architecture (ISA)



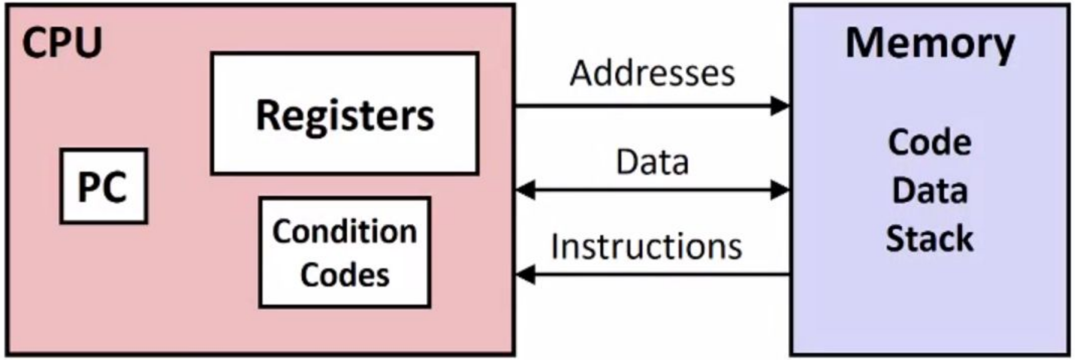
CPU and RAM (Main Memory)

CPU does the actual computation by executing instructions
RAM holds the data and instructions while the computer is running



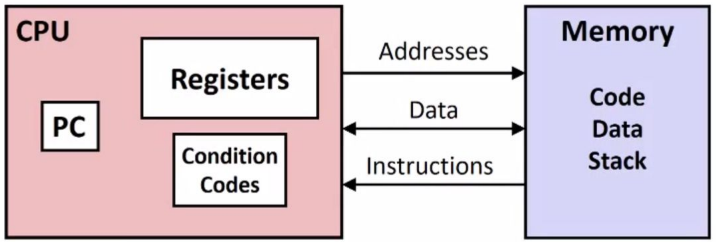
View as an assembly programmer

- 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



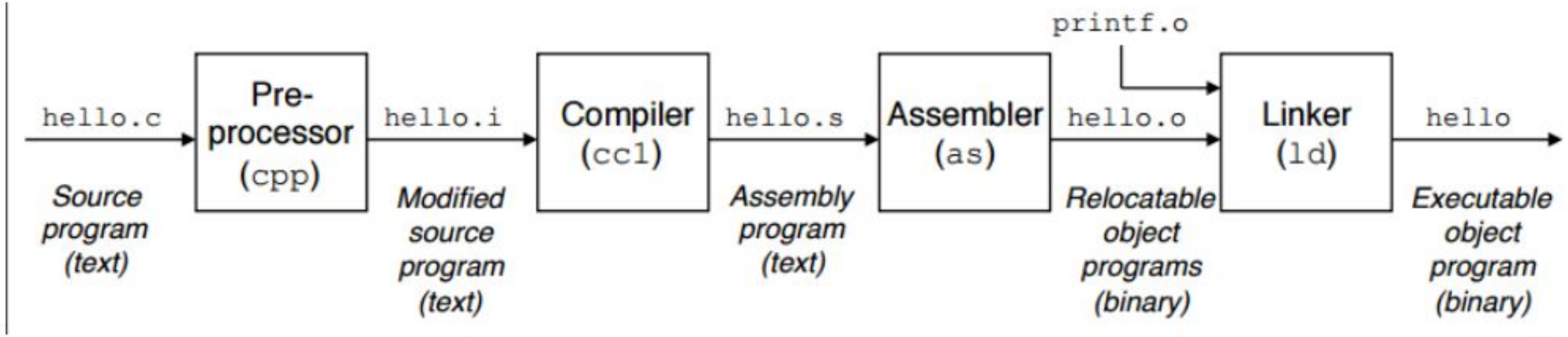
Assembly Operations (i.e. Our instruction set)

- 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)



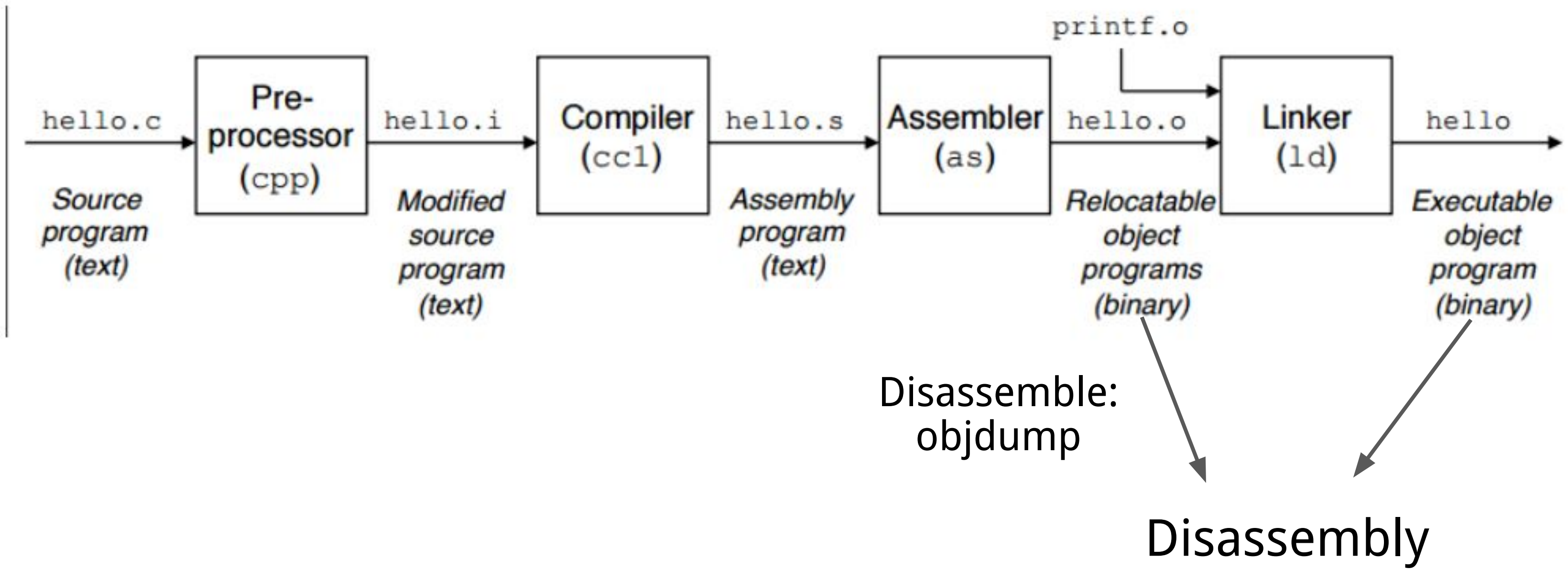
Recall the C toolchain pipeline

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



Recall the C toolchain pipeline

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



So we have gone back in time in a way!

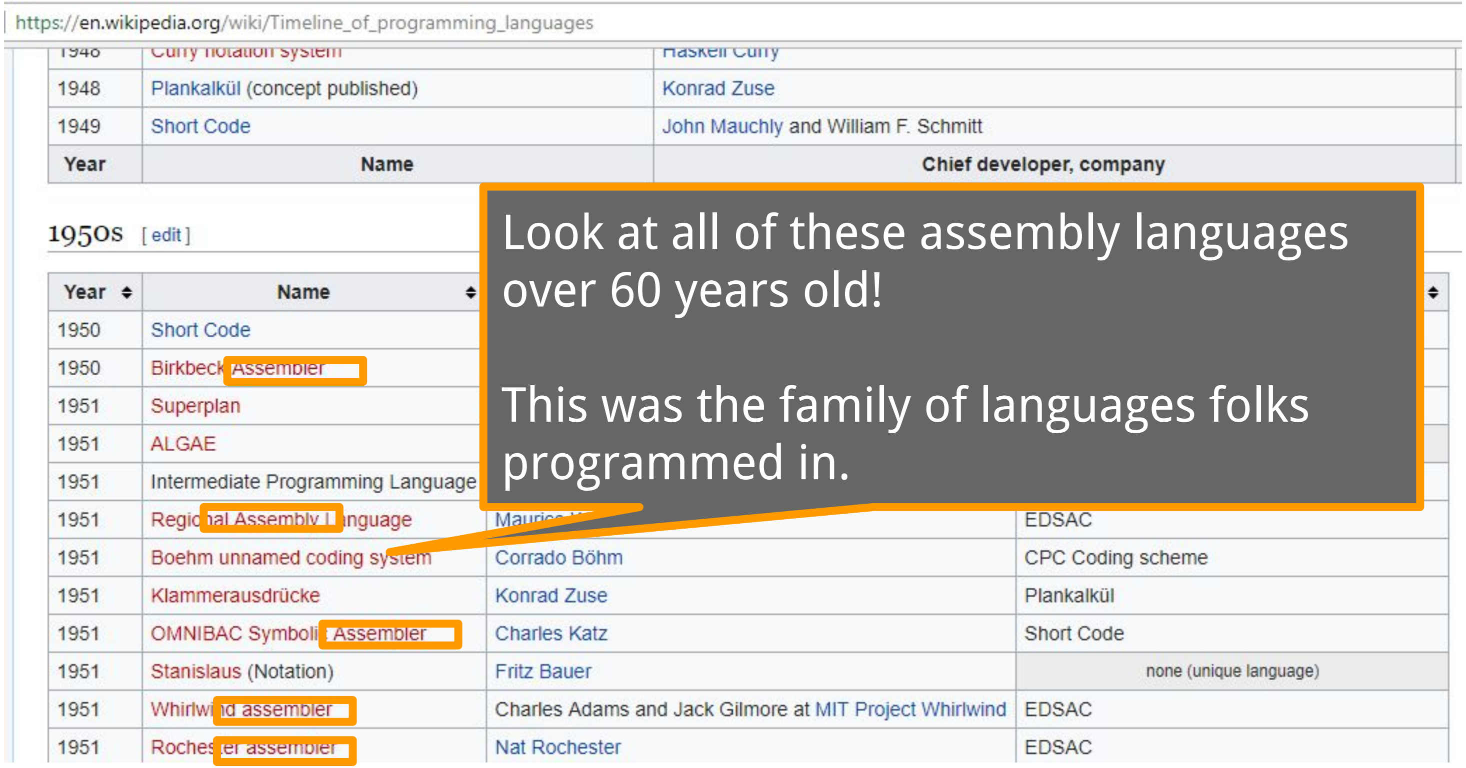
https://en.wikipedia.org/wiki/Timeline_of_programming_languages

Year	Name	Chief developer, company
1948	Curry notation system	Haskell Curry
1948	Plankalkül (concept published)	Konrad Zuse
1949	Short Code	John Mauchly and William F. Schmitt
Year	Name	Chief developer, company

1950s [edit]

Year	Name	Chief developer, company	Predecessor(s)
1950	Short Code	William F Schmidt, Albert B. Tonik,[3] J.R. Logan	Brief Code
1950	Birkbeck Assembler	Kathleen Booth	ARC
1951	Superplan	Heinz Rutishauser	Plankalkül
1951	ALGAE	Edward A Voorhees and Karl Balke	none (unique language)
1951	Intermediate Programming Language	Arthur Burks	Short Code
1951	Regional Assembly Language	Maurice Wilkes	EDSAC
1951	Boehm unnamed coding system	Corrado Böhm	CPC Coding scheme
1951	Klammerausdrücke	Konrad Zuse	Plankalkül
1951	OMNIBAC Symbolic Assembler	Charles Katz	Short Code
1951	Stanislaus (Notation)	Fritz Bauer	none (unique language)
1951	Whirlwind assembler	Charles Adams and Jack Gilmore at MIT Project Whirlwind	EDSAC
1951	Rochester assembler	Nat Rochester	EDSAC

So we have gone back in time!



https://en.wikipedia.org/wiki/Timeline_of_programming_languages

Year	Name	Chief developer, company
1948	Curry notation system	Haskell Curry
1948	Plankalkül (concept published)	Konrad Zuse
1949	Short Code	John Mauchly and William F. Schmitt
Year	Name	Chief developer, company

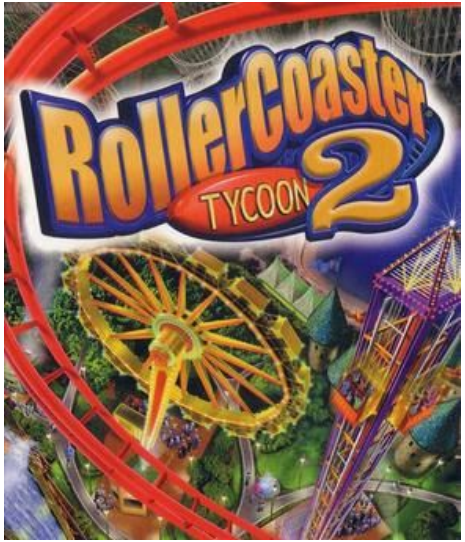
1950s [edit]

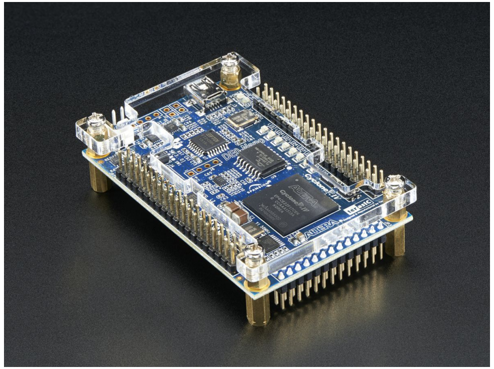
Year	Name	[illegible]	[illegible]
1950	Short Code	[illegible]	[illegible]
1950	Birkbeck Assembler	[illegible]	[illegible]
1951	Superplan	[illegible]	[illegible]
1951	ALGAE	[illegible]	[illegible]
1951	Intermediate Programming Language	[illegible]	[illegible]
1951	Regional Assembly Language	Maurice [illegible]	EDSAC
1951	Boehm unnamed coding system	Corrado Böhm	CPC Coding scheme
1951	Klammerausdrücke	Konrad Zuse	Plankalkül
1951	OMNIBAC Symbolic Assembler	Charles Katz	Short Code
1951	Stanislaus (Notation)	Fritz Bauer	none (unique language)
1951	Whirlwind assembler	Charles Adams and Jack Gilmore at MIT Project Whirlwind	EDSAC
1951	Rochester assembler	Nat Rochester	EDSAC

C was created by Dennis Ritchie at Bell Labs in the early 1970s as an augmented version of Ken Thompson's B.[

Modern Day Assembly is of course still in use

- Still used in games (console games specifically)
    - In hot loops where code must run fast
- Still used on embedded systems
- Useful for debugging any compiled language
- Useful for even non-compiled or Just-In-Time Compiled languages
    - Python has its own bytecode
    - Java’s bytecode (which is eventually compiled) is assembly-like
- Being used on the web
    - webassembly
- Still relevant after 60+ years!





Aside: Java(left) and Python(right) bytecode examples

```
 0 aload_0
 1 new #3 <acceptanceTests/treeset_personOK/Main$A>
 4 dup
 5 new #8 <java/lang/Object>
 8 dup
 9 invokespecial #10 <java/lang/Object.<init>>
12 new #12 <java/lang/Integer>
15 dup
16 iconst_2
17 invokespecial #14 <java/lang/Integer.<init>>
20 invokespecial #17 <acceptanceTests/treeset_personOK/Main$A.<init>>
23 new #12 <java/lang/Integer>
26 dup
27 iconst_1
28 invokespecial #14 <java/lang/Integer.<init>>
31 invokespecial #17 <acceptanceTests/treeset_personOK/Main$A.<init>>
34 getstatic #20 <java/lang/System.out>
37 new #3 <acceptanceTests/treeset_personOK/Main$A>
40 dup
41 new #8 <java/lang/Object>
44 dup
45 invokespecial #10 <java/lang/Object.<init>>
48 new #12 <java/lang/Integer>
51 dup
52 iconst_2
53 invokespecial #14 <java/lang/Integer.<init>>
56 invokespecial #17 <acceptanceTests/treeset_personOK/Main$A.<init>>
59 invokevirtual #26 <java/io/PrintStream.println>
62 return
```

```
def f(n, accum):
  if n <= 1:
    return accum
  else:
    return f(n-1, accum*n)
```

```
>>> import dis
>>> dis.dis(f)
  2           0 LOAD_FAST                0 (n)
              3 LOAD_CONST               1 (1)
              6 COMPARE_OP               1 (<=)
              9 POP_JUMP_IF_FALSE       16

  3          12 LOAD_FAST                1 (accum)
             15 RETURN_VALUE

  5     >>   16 LOAD_GLOBAL              0 (f)
             19 LOAD_FAST                0 (n)
             22 LOAD_CONST               1 (1)
             25 BINARY_SUBTRACT
             26 LOAD_FAST                1 (accum)
             29 LOAD_FAST                0 (n)
             32 BINARY_MULTIPLY
             33 CALL_FUNCTION            2
             36 RETURN_VALUE
             37 LOAD_CONST               0 (None)
             40 RETURN_VALUE
```

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 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.

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

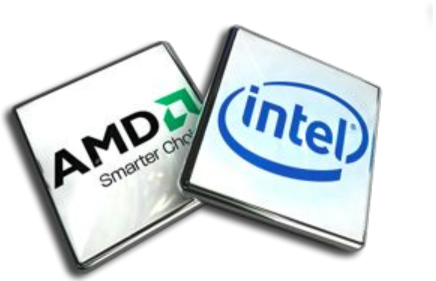
- With Assembly, we lose some of the information we have in C
- In higher-order languages we have many different data types which help protect us from errors.
  - For example: int, long, boolean, char, string, float, double, complex, …
  - In C there are custom data types (structs for example)
  - Type 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]

Intel and x86 Instruction set

- In order to program these chips, there is a specific instruction set we will use
- 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

x86 Architecture

Chronology of x86 processors

Era		Introduction	Prominent CPU models	Address space: Linear	Address space: Virtual	Address space: Physical	Notable features
x86-16	1st	1978	Intel 8086, Intel 8088 (1979)	16-bit	NA	20-bit	16-bit ISA, IBM PC (8088), IBM PC/XT (8088)
		1982	Intel 80186, Intel 80188 NEC V20/V30 (1983)				8086-2 ISA, embedded (80186/80188)
	2nd		Intel 80286 and clones		30-bit	24-bit	protected mode, IBM PC/XT 286, IBM PC/AT
IA-32	3rd	1985	Intel 80386, AMD Am386 (1991)	32-bit	46-bit	32-bit	32-bit ISA, paging, IBM PS/2
	4th (pipelining, cache)	1989	Intel 80486 Cyrix Cx486S, DLC (1992) AMD Am486 (1993), Am5x86 (1995)				pipelining, on-die x87 FPU (486DX), on-die cache
	5th (Superscalar)	1993	Intel Pentium, Pentium MMX (1996)				Superscalar, 64-bit databus, faster FPU, MMX (Pentium MMX), APIC, SMP
		1994	NexGen Nx586 AMD 5k86/K5 (1996)				Discrete microarchitecture (µ-op translation)
		1995	Cyrix Cx5x86 Cyrix 6x86/MX (1997)/MII (1998)				dynamic execution
	6th (PAE, µ-op translation)	1995	Intel Pentium Pro			36-bit (PAE)	µ-op translation, conditional move instructions, dynamic execution, speculative execution, 3-way x86 superscalar, superscalar FPU, PAE, on-chip L2 cache
		1997	Intel Pentium II, Pentium III (1999) Celeron (1998), Xeon				on-package (Pentium II) or on-die (Celeron) L2 Cache, SSE (Pentium III),

Intel released a series of processors in the late 1970s and 1980s whose names all ended with **“86”**.

https://en.wikipedia.org/wiki/X86

Intel Data Types

There are 5 integer data types:

Byte – 8 bits.
Word – 2 bytes; 16 bits.
Dword, Doubleword – 4 bytes; 32 bits.
Quadword – 64 bits.
Double quadword – 128 bits.
Single precision - 32 bits.
Double precision - 64 bits.

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

Endianness
Ordering of Bytes in Memory or Transmission

- Little Endian (Intel, ARM)

Least significant byte has lowest address

Dword address: 0x0

Value: 0x78563412

- Big Endian

Least significant byte has highest address

Dword address: 0x0

Value: 0x12345678

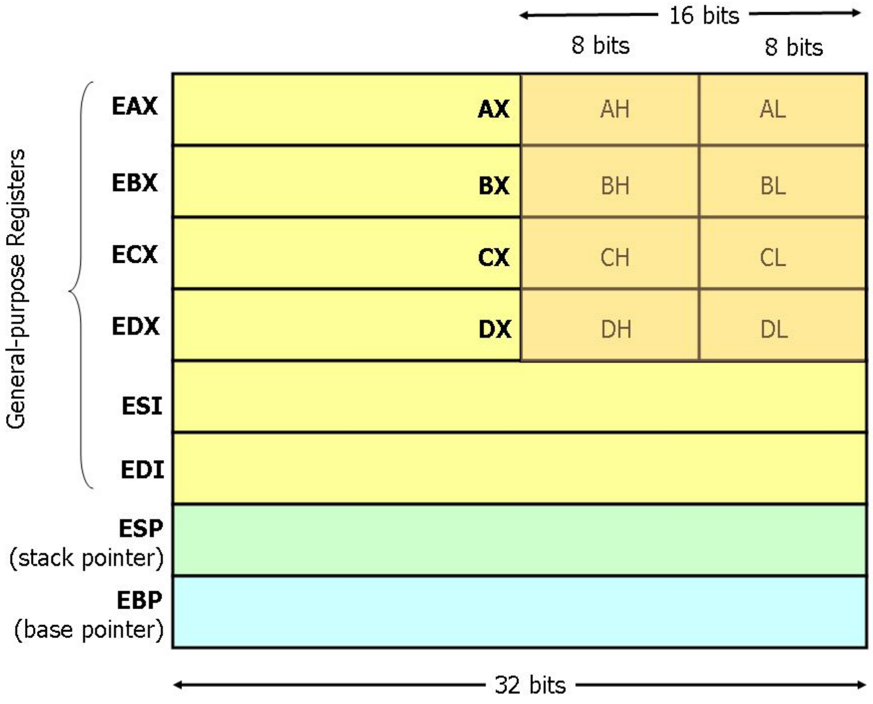
Address 0	0x12
Address 1	0x34
Address 2	0x56
Address 3	0x78

Base Registers

There are

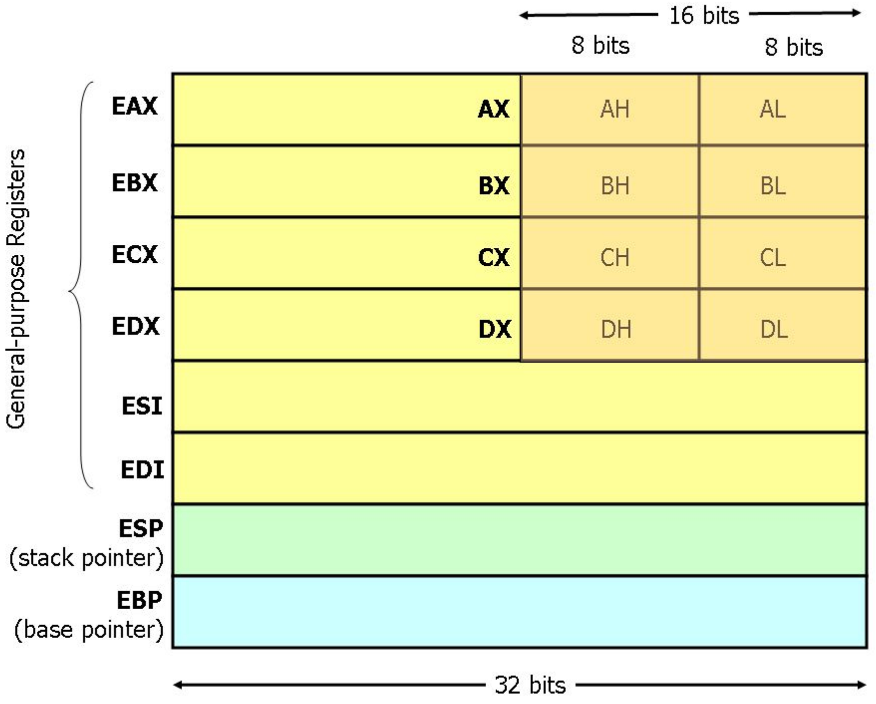
- SIX 32-bit “general-purpose” registers,
- One 32-bit EFLAGS register,
- One 32-bit instruction pointer register (eip), and
- Other special-purpose registers.

The General-Purpose Registers



- 6 general-purpose registers
- esp is the stack pointer
- ebp is the base pointer
- esi and edi are source and destination index registers for array and string operations

The General-Purpose Registers



- The registers eax, ebx, ecx, and edx may be accessed as 32-bit, 16-bit, or 8-bit registers.
- The other four registers can be accessed as 32-bit or 16-bit.

EFLAGS Register

The various bits of the 32-bit EFLAGS register are set (1) or reset/clear (0) according to the results of certain operations.

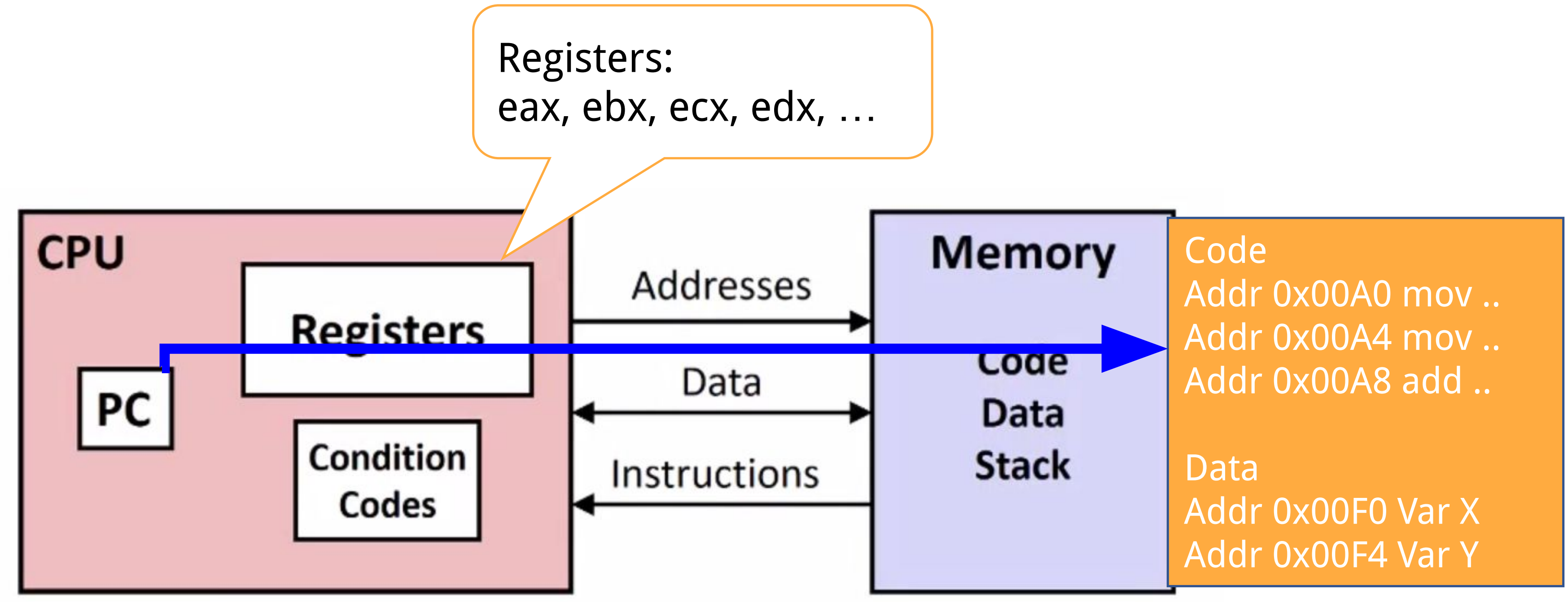
We will be interested in, at most, the bits

CF – carry flag
PF – parity flag
ZF – zero flag
SF – sign flag

Instruction Pointer (EIP)

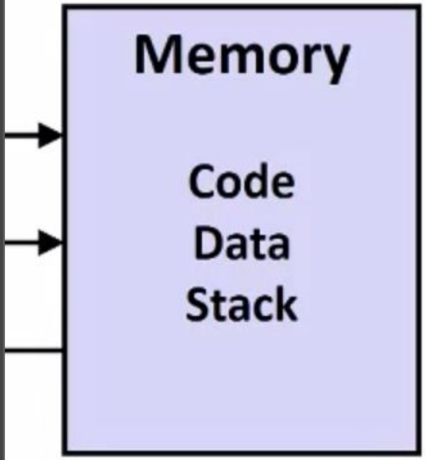
Finally, there is the EIP register, which is the instruction pointer (program counter). Register EIP holds the address of the **next** instruction to be executed.

Program Counter and Memory Addresses

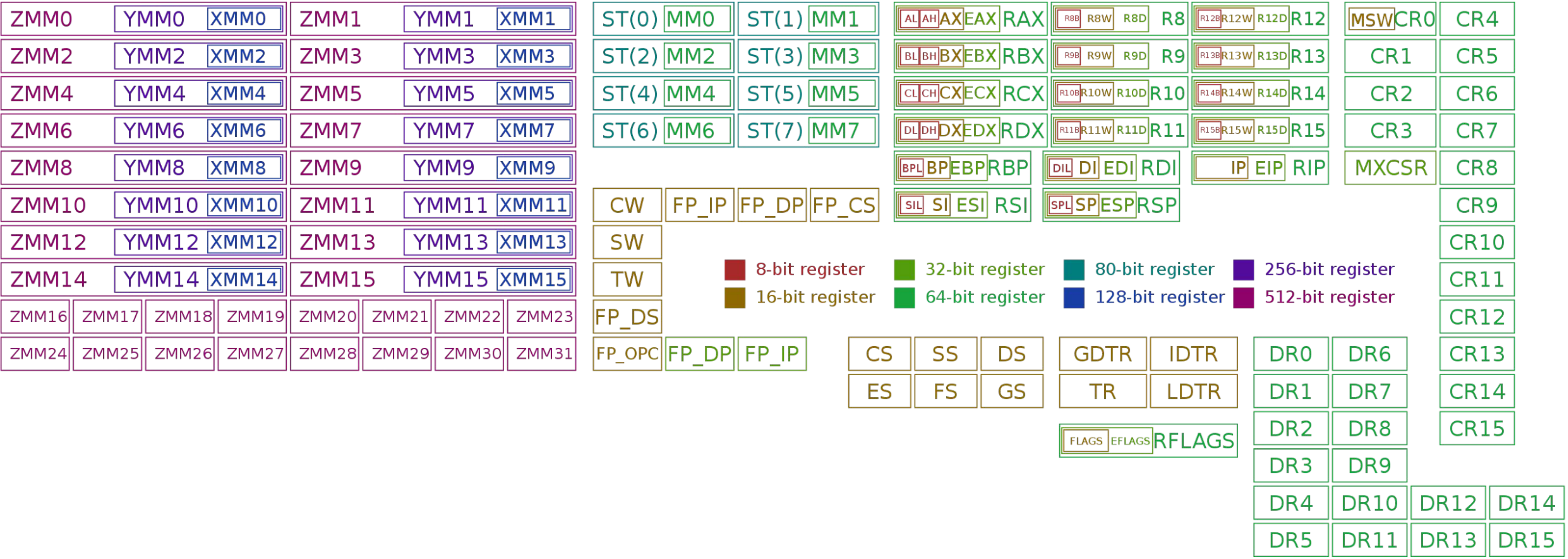


Memory Addresses

- When programming applications 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



Registers on x86 and amd64



https://en.wikipedia.org/wiki/X86

Instructions

Each instruction is of the form

label: mnemonic operand1, operand2, operand3

The label is optional. Operand 1 is the source, operand 2 is the destination in AT&T syntax

The number of operands is 0, 1, 2, or 3, depending on the mnemonic .

Each operand is either

- An immediate value,
- A register, or
- A memory address.

Source and Destination Operands

Each operand is either a source operand or a destination operand.

A source operand, in general, may be

- An immediate value,
- A register, or
- A memory address.

A destination operand, in general, may be

- A register, or
- A memory address.

Instructions

hlt – 0 operands
halts the central processing unit (CPU) until the next external interrupt is fired

inc – 1 operand; inc <reg>, inc <mem>

add – 2 operands; add <reg>,<reg>

imul – 1, 2, or 3 operands; imul <reg32>,<reg32>,<con>

In Intel syntax the first operand is the destination

AT&T Syntax Assembly and Disassembly

Machine instructions generally fall into three categories: data movement, arithmetic/logic, and control-flow.

<reg32> Any 32-bit register (%eax, %ebx, %ecx, %edx, %esi, %edi, %esp, or %ebp)
<reg16> Any 16-bit register (%ax, %bx, %cx, or %dx)
<reg8> Any 8-bit register (%ah, %bh, %ch, %dh, %al, %bl, %cl, or %dl)
<reg> Any register
<mem> A memory address (e.g., (%eax) or (%eax,%ebx,1))
<con32> Any 32-bit immediate
<con16> Any 16-bit immediate
<con8> Any 8-bit immediate
<con> Any 8-, 16-, or 32-bit immediate

Key Points of AT&T Syntax Assembly and Disassembly

- instruction source, destination
  - movl %eax, %ebx   # move contents of EAX into EBX
- Register Prefix %. All registers start with %
  - %eax, %ebx, %rsp, %rdi
- Immediate Values with $. Immediate constants (literal values) use the $ prefix.
  - movl $5, %eax   # put the constant 5 into EAX
- Memory Operands. Memory references are written in parentheses.
  - disp(base, index, scale)
  - movl 8(%ebp), %eax   # load value at [EBP+8] into EAX

Addressing Memory

Move from source (operand 1) to destination (operand 2)

mov (%ebx), %eax (read as MOVE FROM x to y) Load 4 bytes from the memory address in EBX into EAX.

mov -4(%esi), %eax Move 4 bytes at memory address ESI - 4 into EAX.

mov %cl, (%esi,%eax,1) Move the contents of CL into the byte at address ESI+EAX*1.

mov (%esi,%ebx,4), %edx Move the 4 bytes of data at address ESI+4*EBX into EDX.

Addressing Memory

The size prefixes b, w, l, q (x86-64) indicate sizes of 1, 2, 4, and 8 (x86-64) bytes respectively.

mov $2, (%ebx) isn't this ambiguous? We can have a default.

movb $2, (%ebx) Move 2 into the single byte at the address stored in EBX.

movw $2, (%ebx) Move the 16-bit integer representation of 2 into the 2 bytes starting at the address in EBX.

movl $2, (%ebx) Move the 32-bit integer representation of 2 into the 4 bytes starting at the address in EBX.

Data Movement Instructions

mov — Move

Syntax
mov <reg>, <reg>
mov <reg>, <mem>
mov <mem>, <reg>
mov <con>, <reg>
mov <con>, <mem>

Examples
mov %ebx, %eax — copy the value in EBX into EAX
movb $5, var(,1) — store the value 5 into the byte at location var

Data Movement Instructions

push — Push on stack; decrements ESP by 4, then places the operand at the location ESP points to.

Syntax
push <reg32>
push <mem>
push <con32>

Examples
push %eax — push eax on the stack

Data Movement Instructions

pop — Pop from stack

Syntax
pop <reg32>
pop <mem>

Examples
pop %edi — pop the top element of the stack into EDI.
pop (%ebx) — pop the top element of the stack into memory at the four bytes starting at location EBX.

Data Movement Instructions

lea — Load effective address; used for quick calculation

Syntax
lea <mem>, <reg32>

Examples
lea (%ebx,%esi,8), %edi — the quantity EBX+8*ESI is placed in EDI.

Arithmetic and Logic Instructions

add $10, %eax — EAX is set to EAX + 10
addb $10, (%eax) — add 10 to the single byte stored at memory address stored in EAX

sub %ah, %al — AL is set to AL - AH
sub $216, %eax — subtract 216 from the value stored in EAX

dec %eax — subtract one from the contents of EAX

imul (%ebx), %eax — multiply the contents of EAX by the 32-bit contents of the memory at location EBX. Store the result in EAX.

shr %cl, %ebx — Store in EBX the floor of result of dividing the value of EBX by 2n where n is the value in CL.

Control Flow Instructions

jmp — Jump

Transfers program control flow to the instruction at the memory location indicated by the operand.

Syntax

jmp <label> # direct jump

jmp <reg32> # indirect jump

Example

jmp begin — Jump to the instruction labeled begin.

Control Flow Instructions

jcondition — Conditional jump

Syntax

je <label> (jump when equal)
jne <label> (jump when not equal)
jz <label> (jump when last result was zero)
jg <label> (jump when greater than)
jge <label> (jump when greater than or equal to)
jl <label> (jump when less than)
jle <label> (jump when less than or equal to)

Example

```
cmp %ebx, %eax
jle done
```

Control Flow Instructions

cmp — Compare

Syntax
cmp <reg>, <reg>
cmp <mem>, <reg>
cmp <reg>, <mem>
cmp <con>, <reg>

Example
cmpb $10, (%ebx)
jeq loop

If the byte stored at the memory location in EBX is equal to the integer constant 10, jump to the location labeled loop.

Control Flow Instructions

call — Subroutine call

The call instruction first **pushes the current code location onto the hardware supported stack** in memory, and then performs **an unconditional jump to the code** location indicated by the label operand. *Unlike the simple jump instructions, the call instruction saves the location to return to when the subroutine completes*.

Syntax
call <label>
call <reg32>
Call <mem>

Control Flow Instructions

ret — Subroutine return

The ret instruction implements a subroutine return mechanism. This instruction pops a code location off the hardware supported in-memory stack to the program counter.

Syntax
ret

The Run-time Stack

The run-time stack supports procedure calls and the passing of parameters between procedures.

The stack is located in memory.

The stack grows towards **low memory**.

When we push a value, esp is decremented.

When we pop a value, esp is incremented.

Stack Instructions

enter — Create a function frame

Equivalent to:

push %ebp
mov %esp, %ebp
Sub #imm, %esp

Stack Instructions

leave — Releases the function frame set up by an earlier ENTER instruction.

Equivalent to:

```
mov %ebp, %esp
pop %ebp
```

xv6 bootasm.S

xv6-public / bootasm.S

Robert Morris nits

Code Blame 88 lines (73 loc) · 2.92 KB

```
#include "asm.h"
#include "memlayout.h"
#include "mmu.h"

# Start the first CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.code16                       # Assemble for 16-bit mode
.globl start
start:
  cli                         # BIOS enabled interrupts; disable

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax             # Set %ax to zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment

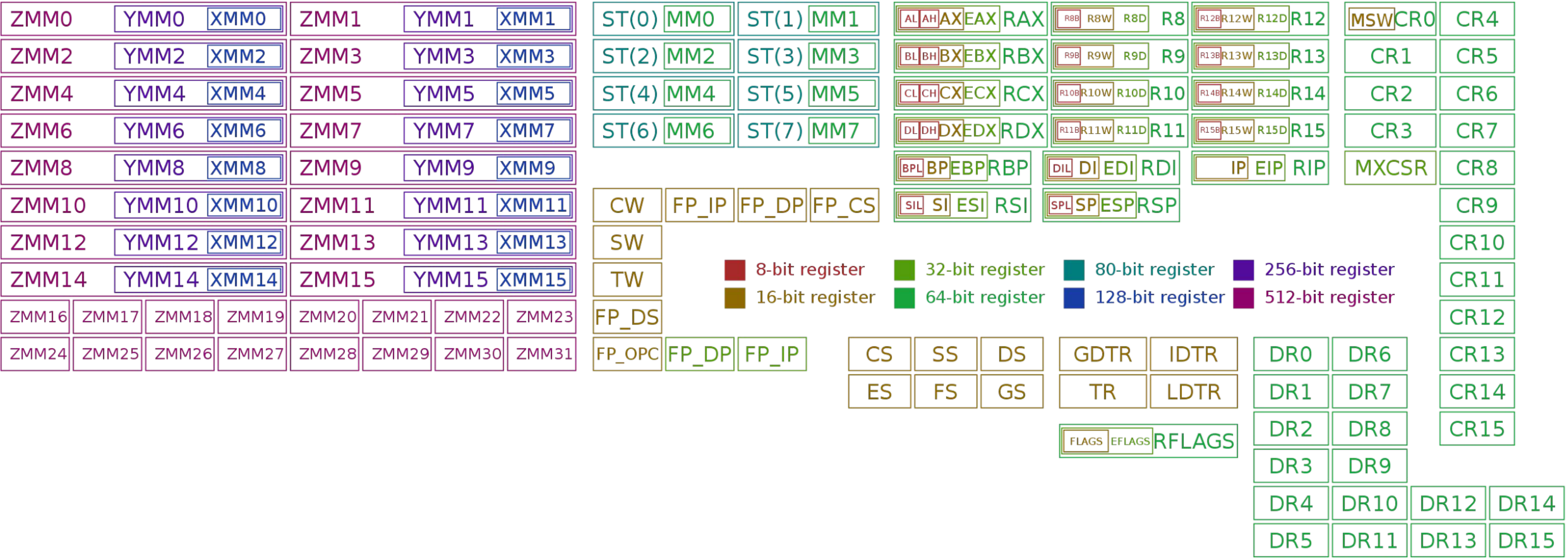
  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64
```

https://github.com/mit-pdos/xv6-public/blob/master/bootasm.S

x86-64/amd64 architecture

Registers on x86 and x86-64

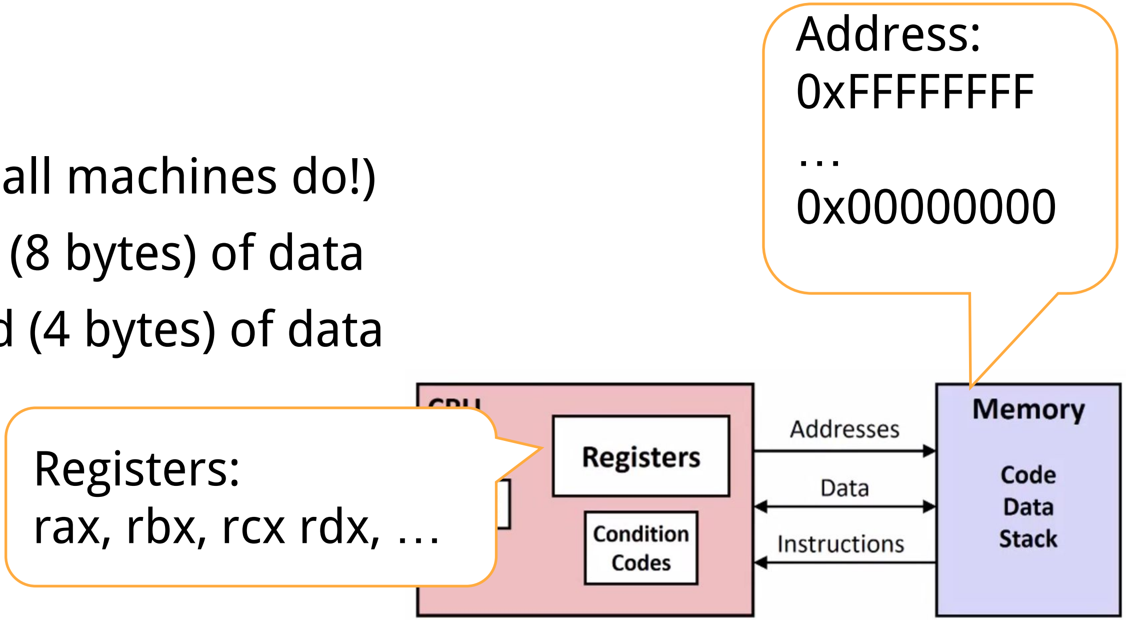


https://en.wikipedia.org/wiki/X86

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), %rax
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	%rsi	%rdx
unsigned int fd	**const char __user** * buf	**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**

X86 Linux Calling Convention (cdecl)

Caller (in this order)

- Pushes arguments onto the stack (in right to left order)
- Execute the **call** instruction (pushes address of instruction after call, then moves dest to %eip)

Callee

- Pushes previous frame pointer onto stack (%ebp)
- Setup new frame pointer (mov %esp, %ebp)
- Creates space on stack for local variables (sub #imm, %esp)
- Ensures that stack is consistent on return
- Return value in %eax register

amd64 Linux Calling Convention

Caller

- Use registers to pass arguments to callee. Register order (1st, 2nd, 3rd, 4th, 5th, 6th, etc.) %rdi, %rsi, %rdx, %rcx, %r8, %r9, ... (use stack for more arguments)

x86 vs. x86-64 (code/ladd)

main.c

```
/*
This program has an integer overflow vulnerability.
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

long long ladd(long long *xp, long long y)
{
  long long t = *xp + y;
  return t;
}
```

```
int main(int argc, char *argv[])
{
  long long a = 0;
  long long b = 0;

  if (argc != 3)
    {
      printf("Usage: ladd a b\n");
      return 0;
    }

  printf("The sizeof(long long) is %d\n", sizeof(long long));

  a = atoll(argv[1]);
  b = atoll(argv[2]);

  printf("%lld + %lld = %lld\n", a, b, ladd(&a, b));
}
```

```
gcc -Wall -m32 -O2 main.c -o ladd
```

```
gcc -Wall -O2 main.c -o ladd64
```

x86 vs. x86-64 (code/ladd)

x86

```
00000640 <ladd>:
 640:  8b 44 24 04        mov    0x4(%esp),%eax
 644:  8b 50 04           mov    0x4(%eax),%edx
 647:  8b 00              mov    (%eax),%eax
 649:  03 44 24 08        add    0x8(%esp),%eax
 64d:  13 54 24 0c        adc    0xc(%esp),%edx
 651:  c3                 ret
```

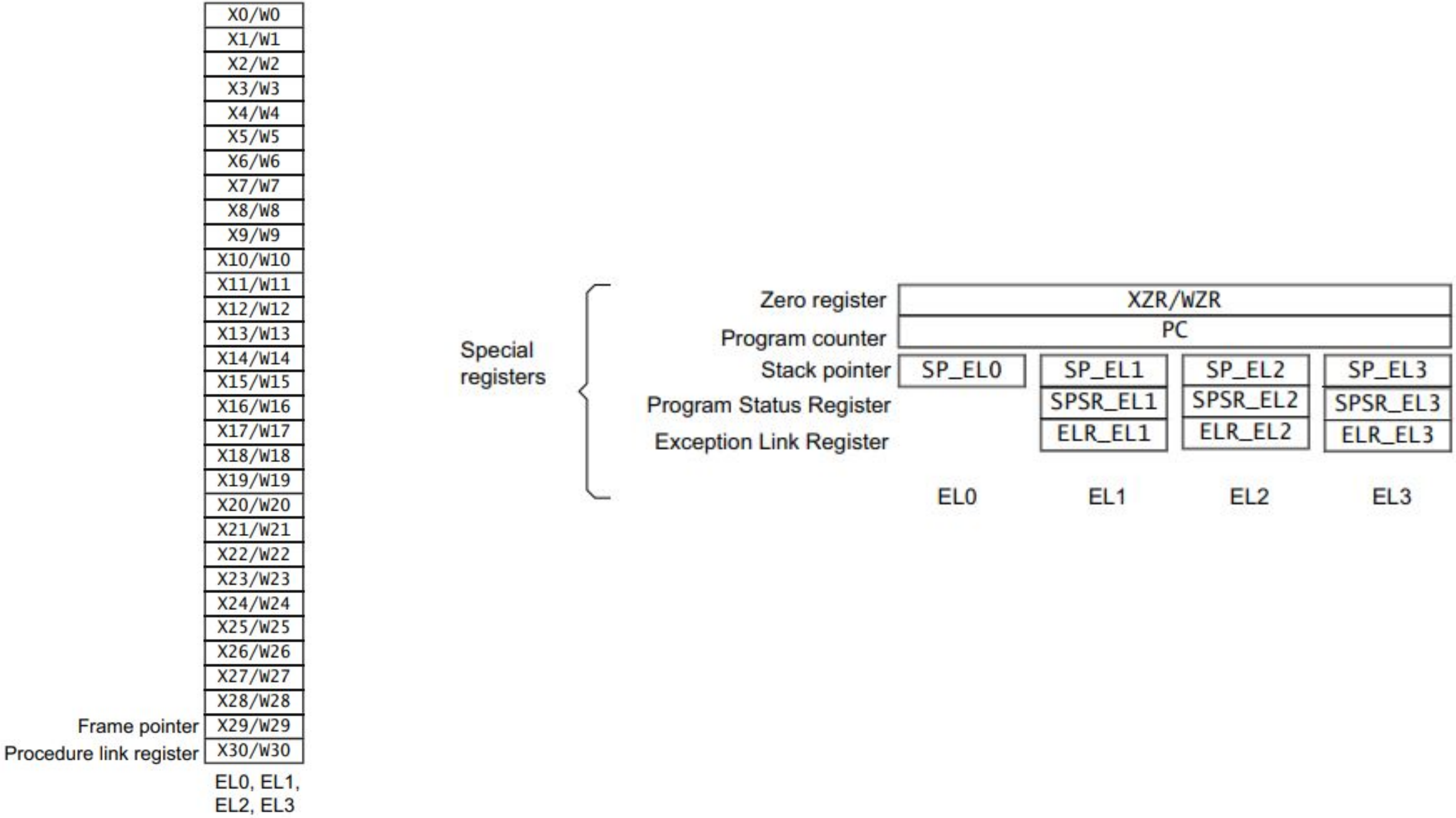
x86-64

```
0000000000000780 <ladd>:
 780:  48 8b 07           mov    (%rdi),%rax
 783:  48 01 f0           add    %rsi,%rax
 786:  c3                 retq
```

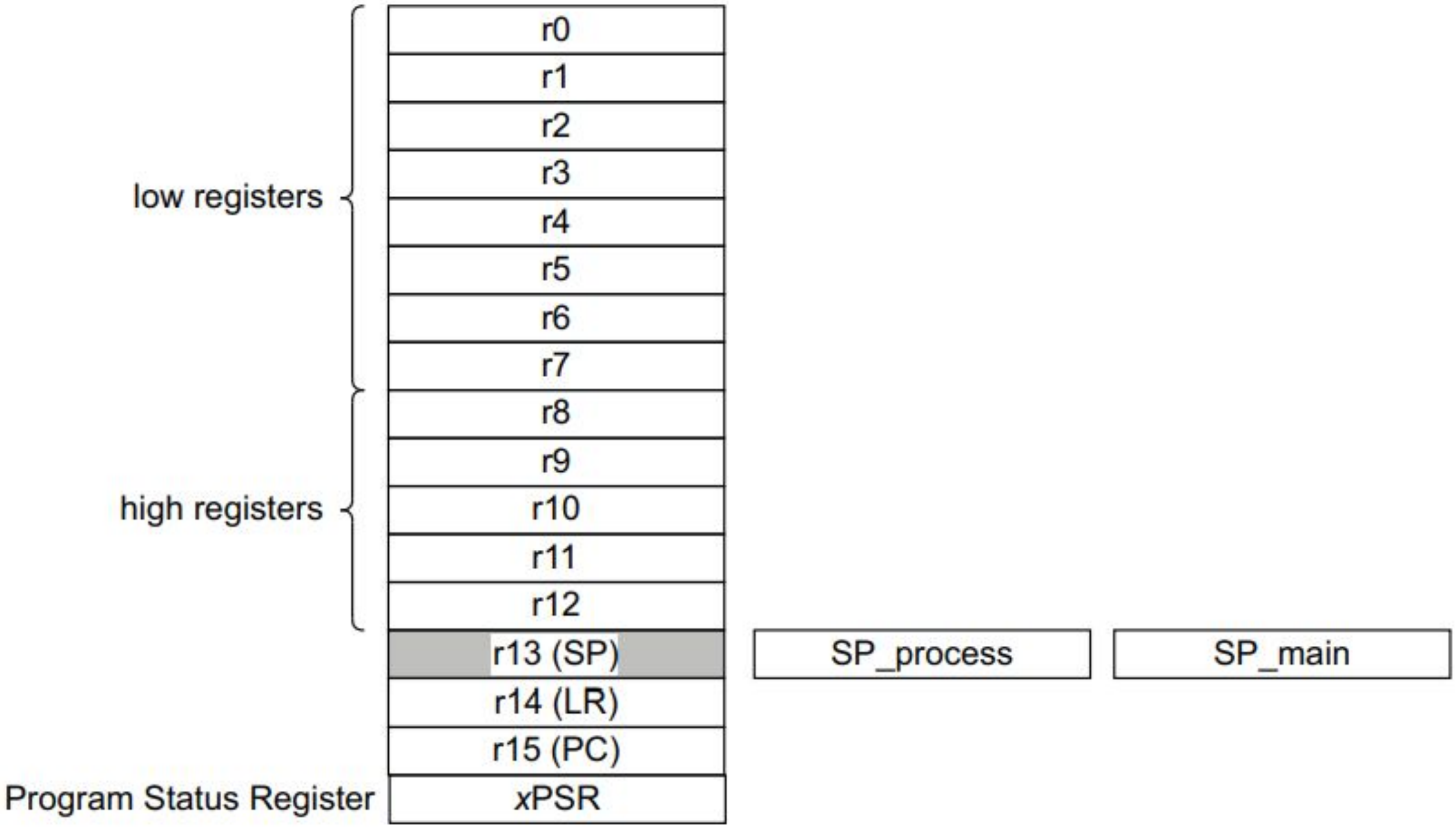
```
objdump -d ladd
objdump -d ladd64
```

ARM Cortex-A/M Architecture

Cortex-A 64 bit



Cortex-M 32 bit



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

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)

- 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 is ASM?

- ```
  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

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
  ```

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)

Bookmarks

- Volume 1:Basic Architecture
- Volume 2 (2A, 2B, 2C & 2D):Instruction Set Reference, A-Z
  - Chapter 1 About This Manual
  - Chapter 2 Instruction Format
  - Chapter 3 Instruction Set Reference, A-L
    - 3.1 Interpreting the Instruction Reference Pages
    - 3.2 Instructions (A-L)
  - Chapter 4 Instruction

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

Comparisons

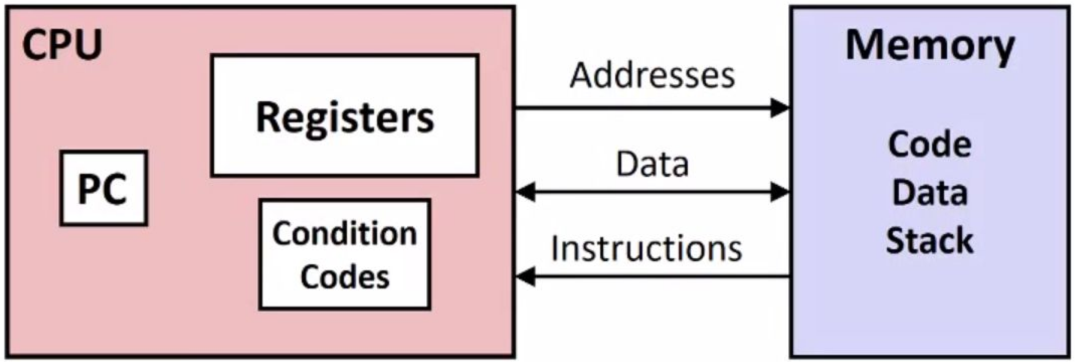
Compare operands: cmp_, jmp_, set__

- 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
```

- 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

Equivalent C Code

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