System Programming

06. Machine-Level Programming I: Basics (ch 3.1-3.5)- part1

2019. Fall

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

datalab

These slides are based on your text book on the slides prepared by Dan Grossman, Ruth Anderson.

data science laboratory

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);  fl
oat mpg =
     c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

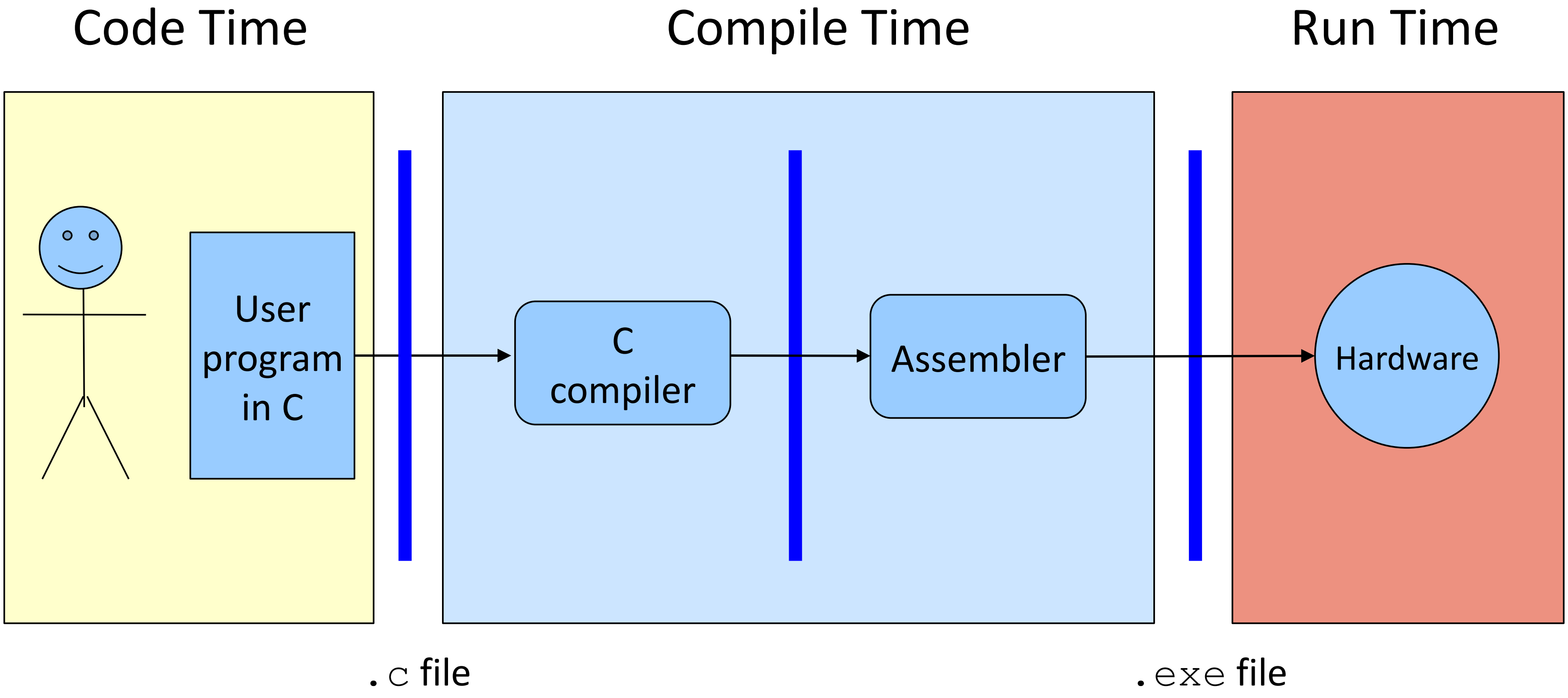
Computer system:

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

Machine Programming I: Basics

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

Translation



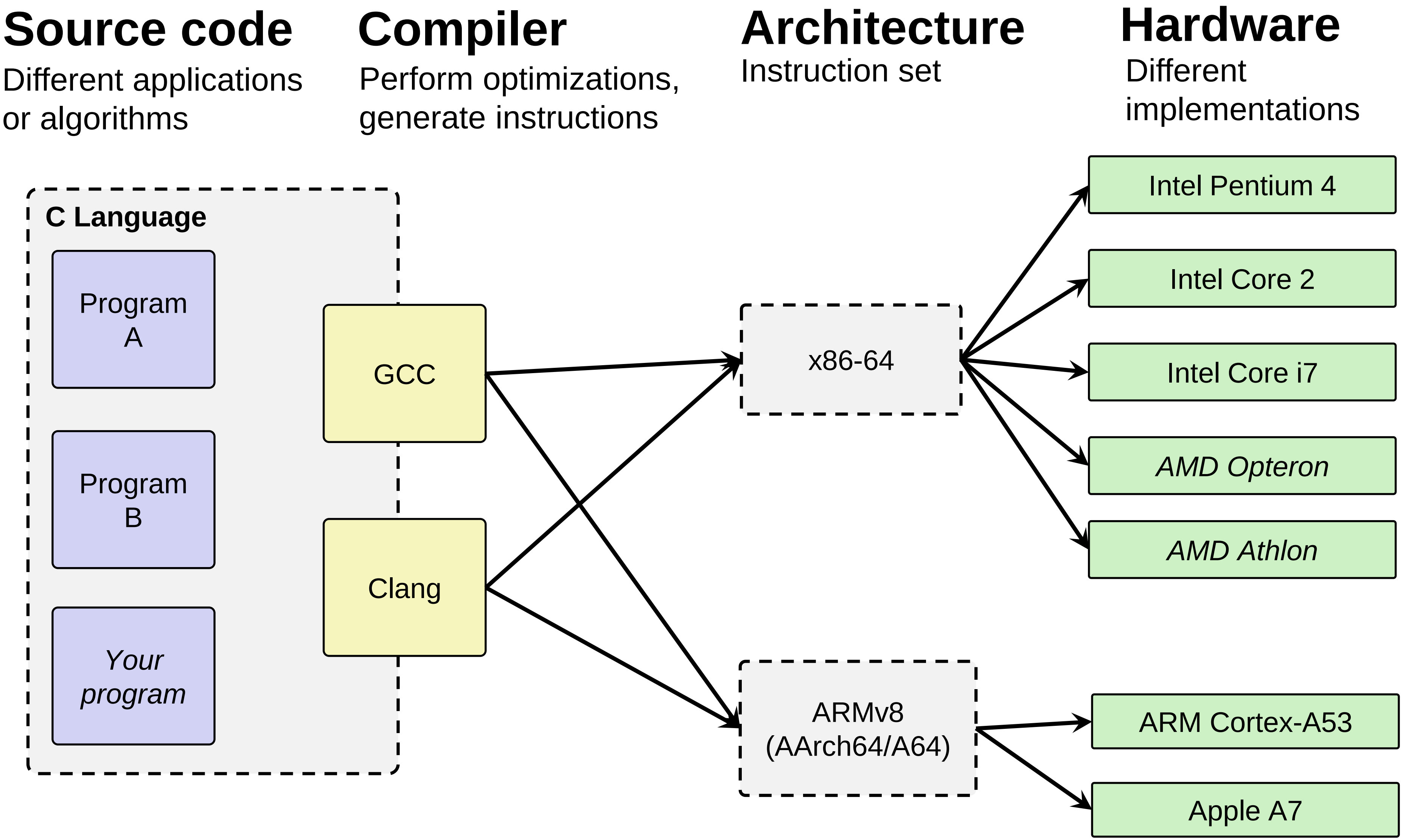
What makes programs run fast(er)?

Translation Impacts Performance

- The time required to execute a program depends on:
  - **The program** (as written in C, for instance)
  - **The compiler**: what set of assembler instructions it translates the C program into
  - **The instruction set architecture** (ISA): what set of instructions it makes available to the compiler
  - **The hardware implementation**: how much time it takes to execute an instruction

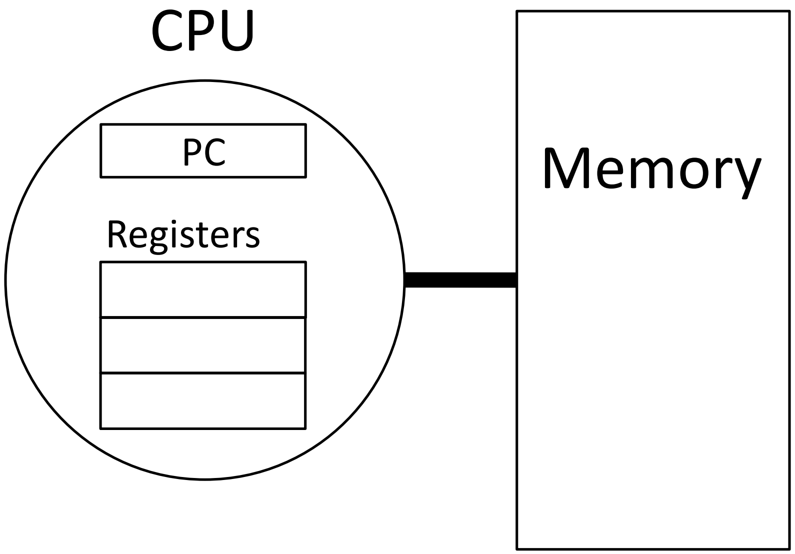
What should the HW/SW interface contain?

HW Interface Affects Performance



Instruction Set Architectures

- The ISA defines:
  - The system’s state (e.g. registers, memory, program counter)
  - The instructions the CPU can execute
  - The effect that each of these instructions will have on the system state



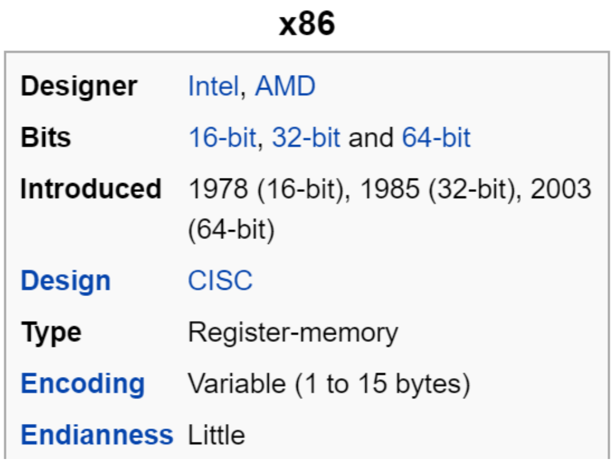
Instruction Set Philosophies

- *Complex Instruction Set Computing* (CISC):
  - Add more and more elaborate and specialized instructions as needed
    - Lots of tools for programmers to use, but hardware must be able to handle all instructions
    - x86-64 is CISC, but only a small subset of instructions encountered with Linux programs
- *Reduced Instruction Set Computing* (RISC):
  - Keep instruction set small and regular
    - Easier to build fast hardware
    - Let software do the complicated operations by composing simpler ones

General ISA Design Decisions

- Instructions
  - What instructions are available? What do they do?
  - How are they encoded? Instructions are data!
- Registers
  - How many registers are there?
  - How wide are they? Size of a word
- Memory
  - How do you specify a memory location? Different ways to build up an address

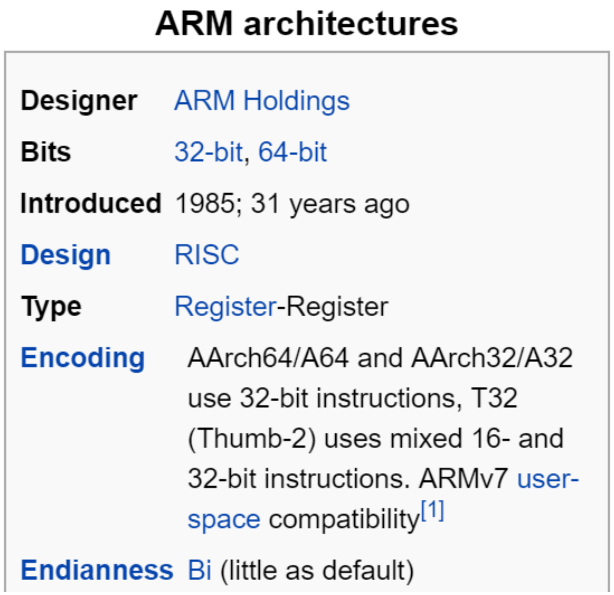
Mainstream ISAs



x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register-memory
Encoding	Variable (1 to 15 bytes)
Endianness	Little

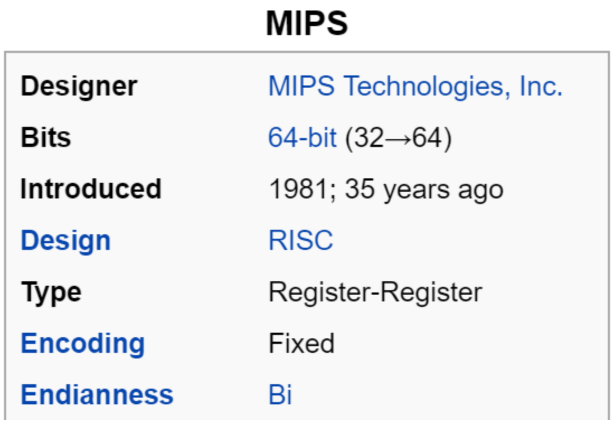
Macbooks & PCs
(Core i3, i5, i7, M)
[x86-64 Instruction Set]



ARM architectures

Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985; 31 years ago
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[1]
Endianness	Bi (little as default)

Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
[ARM Instruction Set]



MIPS

Designer	MIPS Technologies, Inc.
Bits	64-bit (32→64)
Introduced	1981; 35 years ago
Design	RISC
Type	Register-Register
Encoding	Fixed
Endianness	Bi

Digital home & networking equipment
(Blu-ray, PlayStation 2)
[MIPS Instruction Set]

X86 ISA

- Processors that implement the x86 ISA completely dominate the server, desktop and laptop markets

- **Evolutionary design**
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on

- **Complex instruction set computer (CISC)**
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - (as opposed to Reduced Instruction Set Computers (RISC), which use simpler instructions)

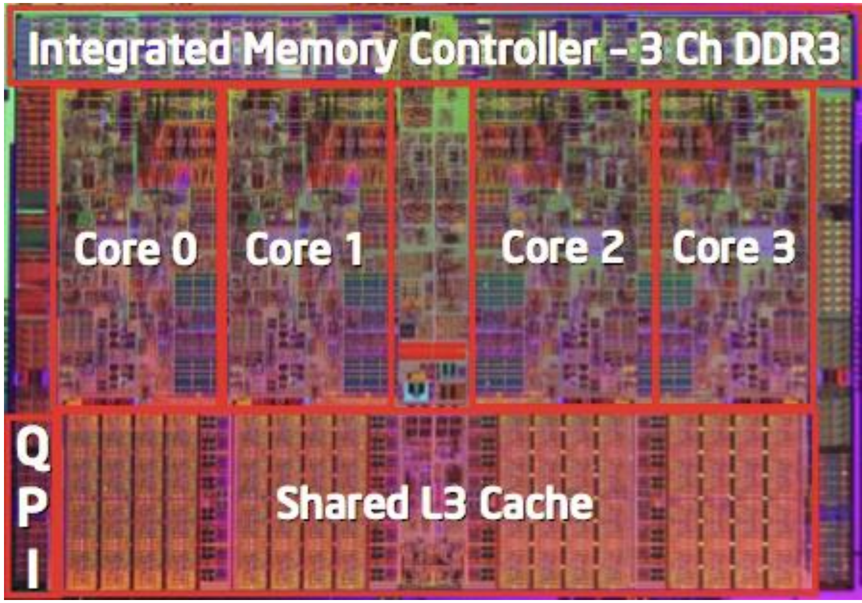
Intel x86 Evolution: Milestones

Name	*Date*	*Transistors*	*MHz*
8086	1978	29K	5-10
First 16-bit Intel processor. Basis for IBM PC & DOS 1MB address space			
386	1985	275K	16-33
First 32 bit Intel processor , referred to as IA32 Added “flat addressing”, capable of running Unix			
Pentium 4E	2004	125M	2800-3800
First 64-bit Intel x86 processor, referred to as x86-64			
Core 2	2006	291M	1060-3500
First multi-core Intel processor			
Core i7	2008	731M	1700-3900
Four cores (our shark machines)			

Intel x86 Processors, cont.

- Machine Evolution

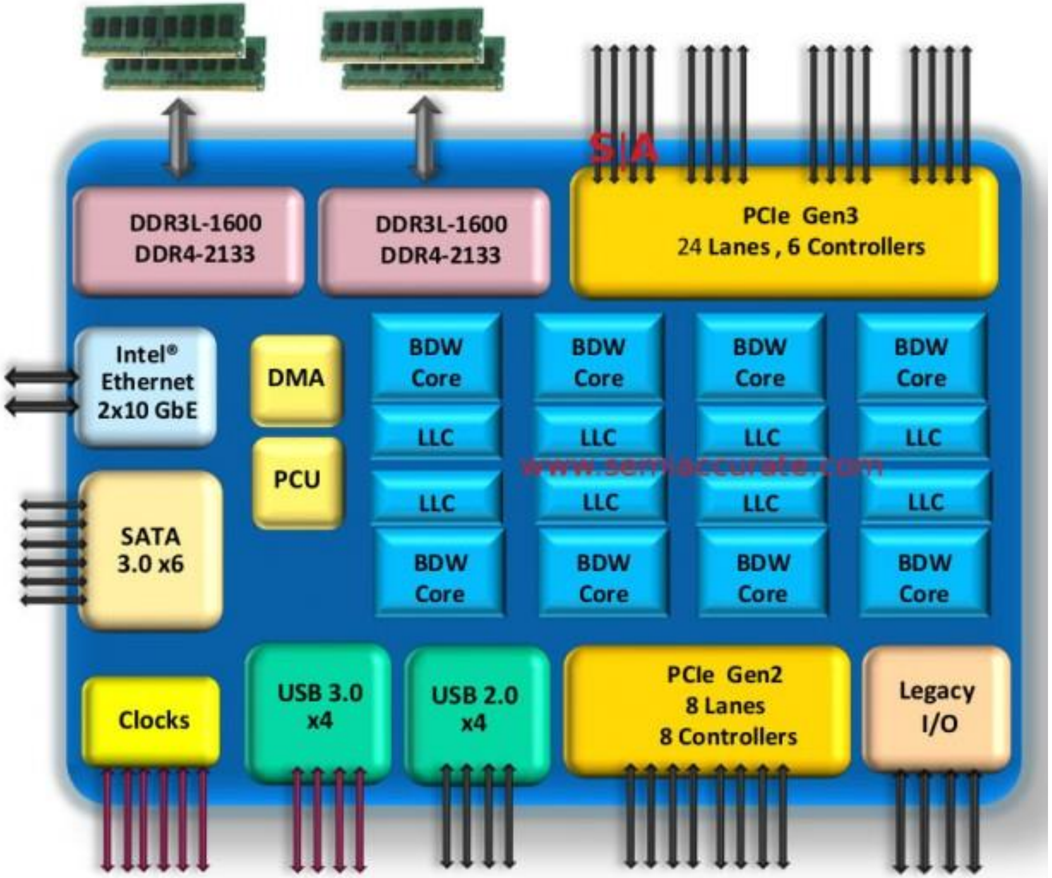
386	1985	0.3M
Pentium	1993	3.1M
Pentium/MMX	1997	4.5M
PentiumPro	1995	6.5M
Pentium III	1999	8.2M
Pentium 4	2001	42M
Core 2 Duo	2006	291M
Core i7	2008	731M



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

2015 State of the Art

- Core i7 Broadwell 2015
- Desktop Model
  - 4 cores
  - Integrated graphics
  - 3.3-3.8 GHz
  - 65W
- Server Model
  - 8 cores
  - Integrated I/O
  - 2-2.6 GHz
  - 45W



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
- Recent Years
  - Intel got its act together
    - Leads the world in semiconductor technology
  - AMD has fallen behind
    - Relies on external semiconductor manufacturer

Intel's Transition to 64-Bit

- **2001: Intel Attempts Radical Shift from IA32 to IA64**
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- **2003: AMD Steps in with Evolutionary Solution**
  - 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

Our Coverage

- IA32
  - The traditional x86

- x86-64
  - The standard
  - `$ gcc hello.c`
  - `$ gcc –m64 hello.c`

- Presentation
  - Book covers x86-64
  - Web aside on IA32
  - We will only cover x86-64

Roadmap

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

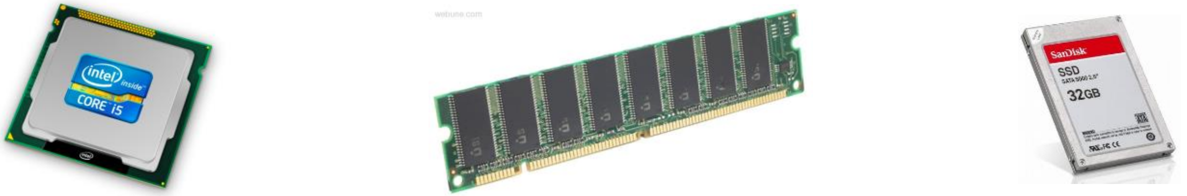
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8 Mac

Computer system:



Machine Programming I: Basics

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

Definitions

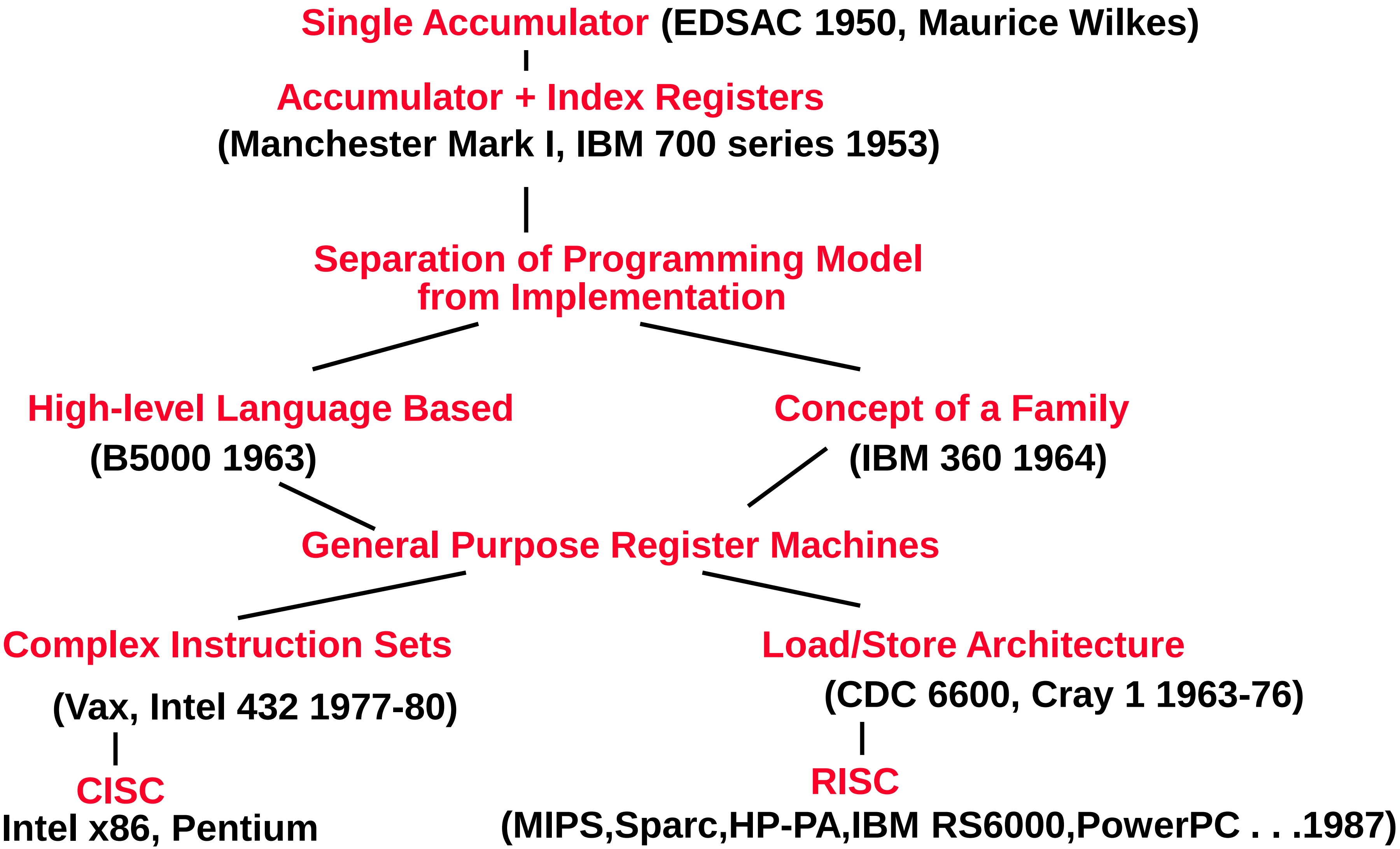
- Architecture: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write assembly/machine code.
  - Examples: instruction set specification, registers.
- Microarchitecture: Implementation of the architecture.
  - Examples: cache sizes and core frequency.
- Code Forms:
  - Machine Code: The byte-level programs that a processor executes
  - Assembly Code: A text representation of machine code

- Is cache size "architecture"?
- How about CPU frequency?
- And number of registers?

Detour: Instruction Set Design Issues

- Where are operands stored?
  - registers, memory, stack, accumulator
- How many explicit operands are there?
  - 0, 1, 2, or 3
- How is the operand location specified?
  - register, immediate, indirect, . . .
- What type & size of operands are supported?
  - byte, int, float, double, string, vector. . .
- What operations are supported?
  - add, sub, mul, move, compare . . .

Detour: Evolution of Instruction Sets



Single Accumulator (EDSAC 1950, Maurice Wilkes)

Accumulator + Index Registers

(Manchester Mark I, IBM 700 series 1953)

Separation of Programming Model from Implementation

High-level Language Based

(B5000 1963)

Concept of a Family

(IBM 360 1964)

General Purpose Register Machines

Complex Instruction Sets

(Vax, Intel 432 1977-80)

CISC

Intel x86, Pentium

Load/Store Architecture

(CDC 6600, Cray 1 1963-76)

RISC

(MIPS,Sparc,HP-PA,IBM RS6000,PowerPC . . .1987)

Detour: Classifying ISAs

Accumulator (before 1960, e.g. 68HC11):

1-address	add A	acc ← acc + mem[A]

Stack (1960s to 1970s):

0-address	add	tos ← tos + next

Memory-Memory (1970s to 1980s):

2-address	add A, B	mem[A] ← mem[A] + mem[B]
3-address	add A, B, C	mem[A] ← mem[B] + mem[C]

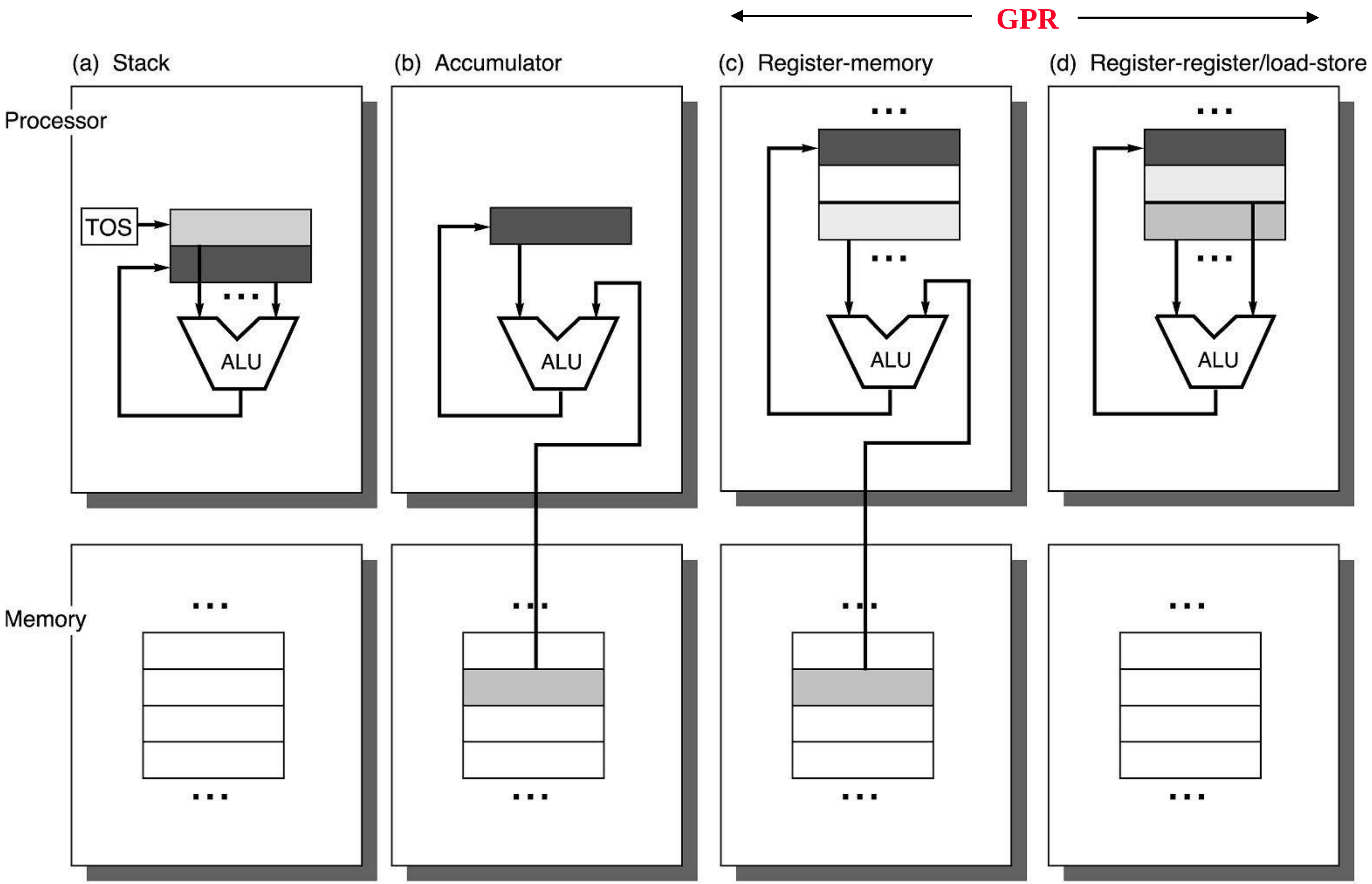
Register-Memory (1970s to present, e.g. 80x86):

2-address	add R1, A	R1 ← R1 + mem[A]
	load R1, A	R1 ← mem[A]

Register-Register (Load/Store) (1960s to present, e.g. MIPS):

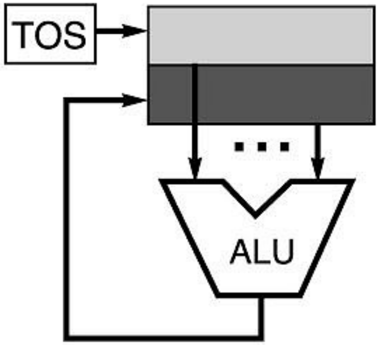
3-address	add R1, R2, R3	R1 ← R2 + R3
	load R1, R2	R1 ← mem[R2]
	store R1, R2	mem[R1] ← R2

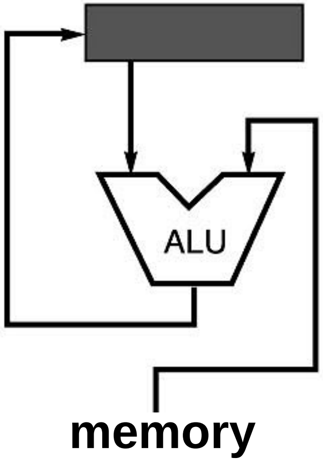
Detour: Operand Locations in Four ISA Classes



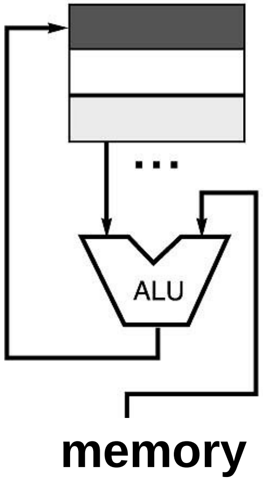
Detour: Code Sequence C = A + B for Four Instruction Sets

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1,A Load R2, B Add R3, R1, R2 Store C, R3

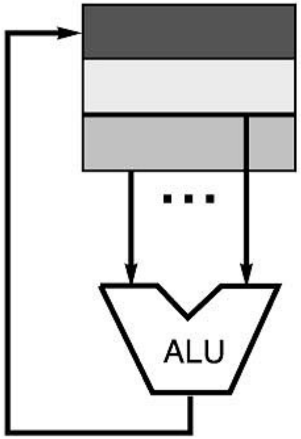




acc = acc + mem[C]



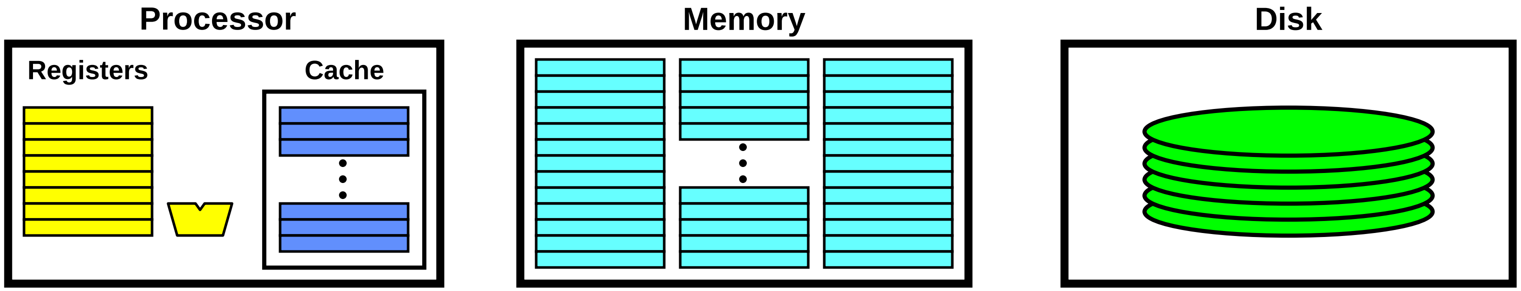
R1 = R1 + mem[C]



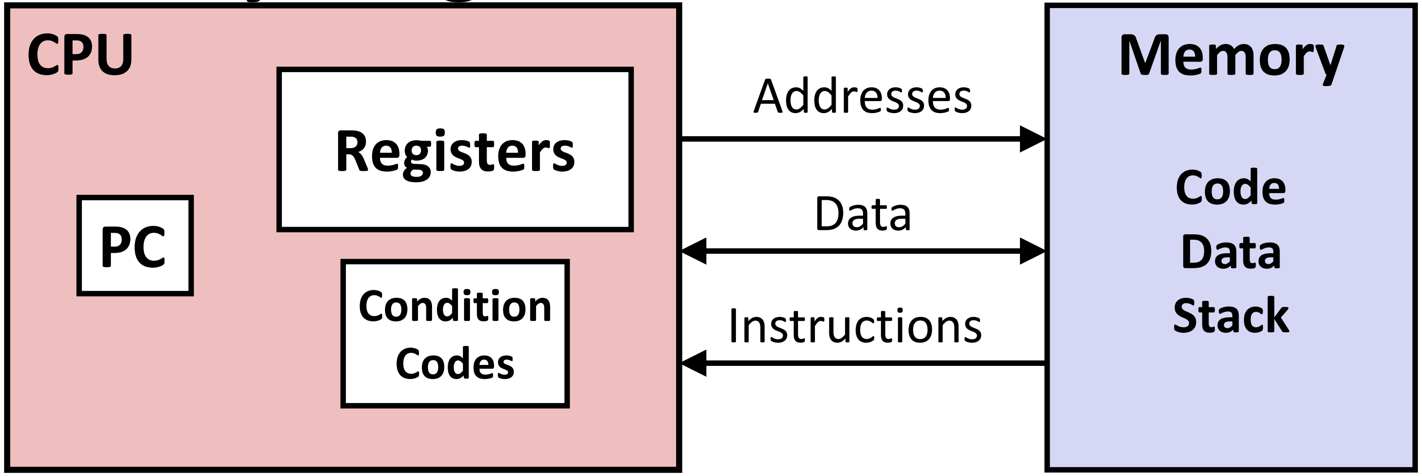
R3 = R1 + R2

Detour: More About General Purpose Regist ers

- Why do almost all new architectures use GPRs?
  - Registers are much faster than memory (even cache)
    - Register values are available immediately
    - When memory isn’t ready, processor must wait (“stall”)
  - Registers are convenient for variable storage
    - Compiler assigns some variables just to registers
    - More compact code since small fields specify registers (compared to memory addresses)



Assembly Programmer's 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

- Integral data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
  - Different registers for those (*e.g.* `%xmm1`, `%ymm2`)
  - Come from *extensions to x86* (SSE, AVX, …)
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
- Two common syntaxes
  - "AT&T": used by our course, slides, textbook, gnu tools, …
  - "Intel": used by Intel documentation, Intel tools, …
  - Must know which you're reading

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

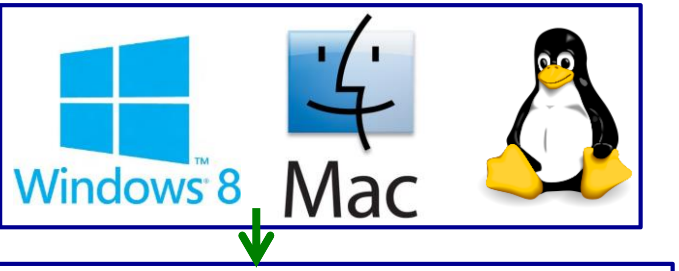
Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

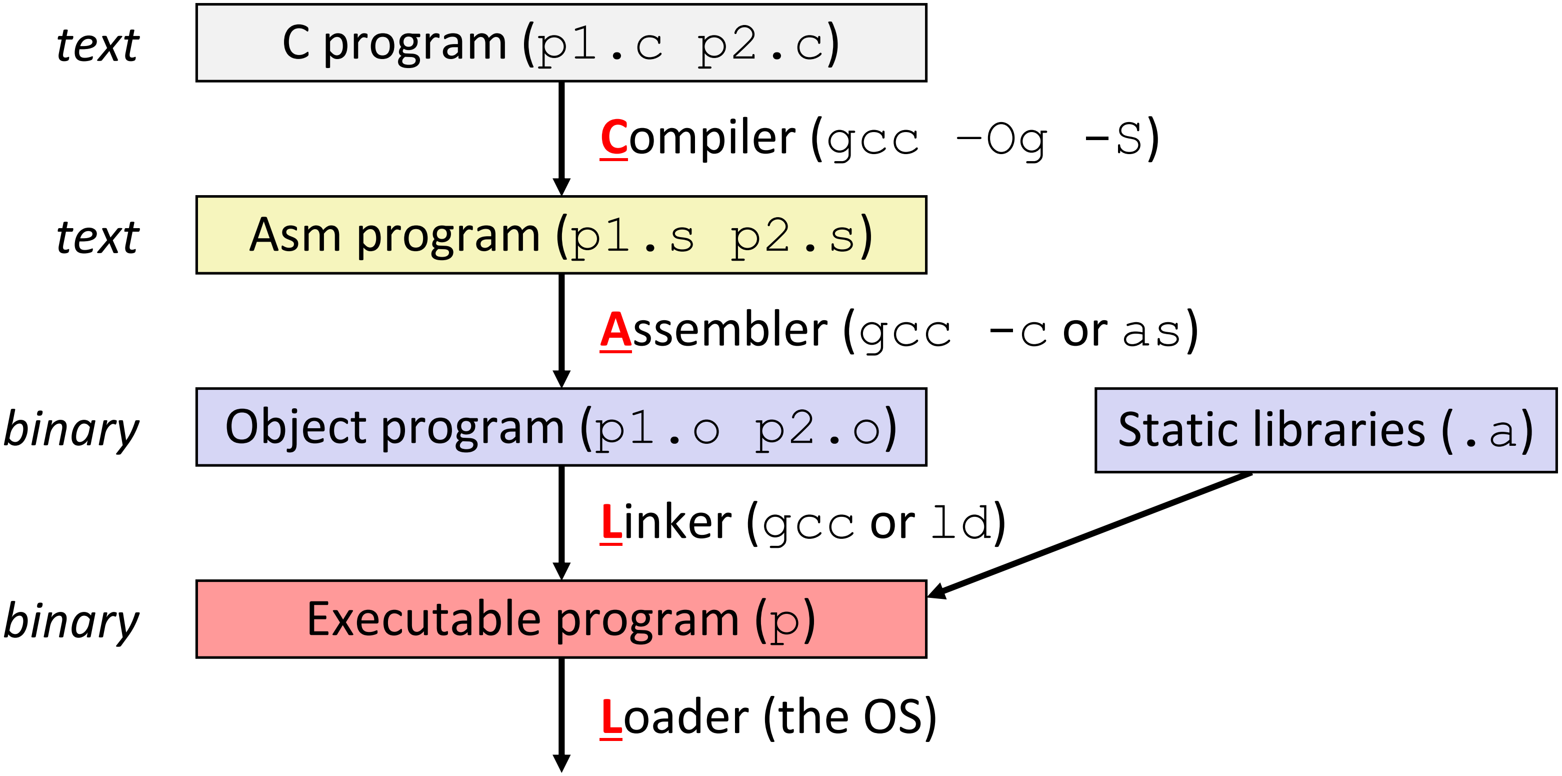


Computer system:

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

Building an Executable from a C File

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Put resulting machine code in file `p`
- Run with command: `./p`



text C program (`p1.c p2.c`)

Compiler (`gcc -Og -S`)

text Asm program (`p1.s p2.s`)

Assembler (`gcc -c` or `as`)

binary Object program (`p1.o p2.o`)

Static libraries (`.a`)

Linker (`gcc` or `ld`)

binary Executable program (`p`)

Loader (the OS)

Compiler

- **Input:** Higher-level language code (*e.g.* C, Java)
  - `foo.c`
- **Output:** Assembly language code (*e.g.* x86, ARM, MIPS)
  - `foo.s`

- First there's a preprocessor step to handle #directives
  - Macro substitution, plus other specialty directives
  - If curious/interested: http://tigcc.ticalc.org/doc/cpp.html
- Compiler is super complex, whole courses devoted to these!
- Compiler optimizations
  - "Level" of optimization specified by capital '`O`' flag (*e.g.* `-Og`, `-O3`)
  - Options: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Compiling Into Assembly

- C Code (`sum.c`)

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

- x86-64 assembly (`gcc –Og –S sum.c`)
  - Generates file `sum.s` (see https://godbolt.org/g/pQUhIZ)

```
sumstore(long, long, long*):
  addq    %rdi, %rsi
  movq    %rsi, (%rdx)
  ret
```

Warning: You may get different results with other versions of `gcc` and different compiler settings

Assembler

- **Input:** Assembly language code (*e.g.* x86, ARM, MIPS)
  - `foo.s`
- **Output:** Object files (*e.g.* ELF, COFF)
  - `foo.o`
  - Contains *object code* and *information tables*
- Reads and uses *assembly directives*
  - *e.g.* `.text`, `.data`, `.quad`
  - x86: https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html
- Produces "machine language"
  - Does its best, but object file is *not* a completed binary
- Example: `gcc -c foo.s`

Producing Machine Language

- **Simple cases:** arithmetic and logical operations, shifts, etc.
  - All necessary information is contained in the instruction itself

- What about the following?
  - Conditional jump
  - Accessing static data (*e.g.* global var or jump table)
  - `call`

- Addresses and labels are problematic because final executable hasn't been constructed yet!
  - So how do we deal with these in the meantime?

Object File Information Tables

- **Symbol Table** holds list of "items" that may be used by other files
    - *Non-local labels* – function names for `call`
    - *Static Data* – variables & literals that might be accessed across files

- **Relocation Table** holds list of "items" that this file needs the address of later (currently undetermined)
    - Any *label* or piece of *static data* referenced in an instruction in this file
        - Both internal and external

- Each file has its own symbol and relocation tables

Object File Format

1) object file header: size and position of the other pieces of the object file
2) text segment: the machine code
3) data segment: data in the source file (binary)
4) relocation table: identifies lines of code that need to be "handled"
5) symbol table: list of this file's labels and data that can be referenced
6) debugging information

- More info: ELF format
  - http://www.skyfree.org/linux/references/ELF_Format.pdf

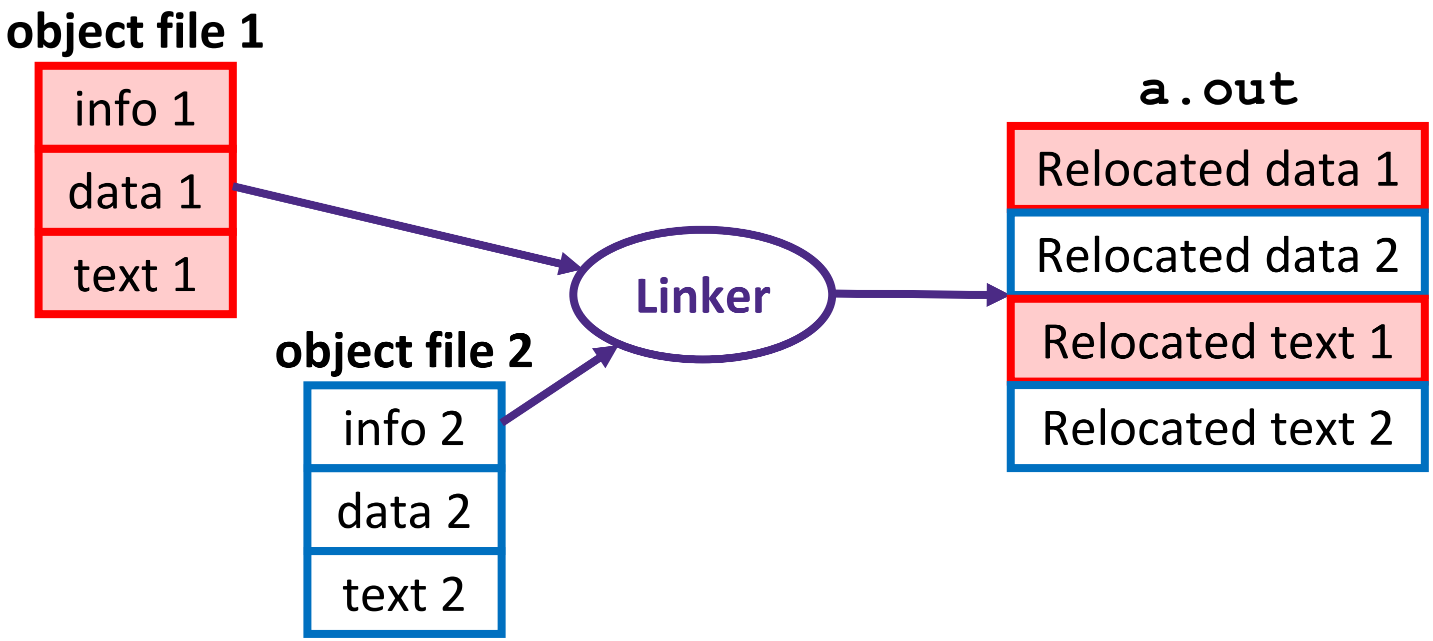
Linker

- **Input:** Object files (e.g. ELF, COFF)
  - `foo.o`
- **Output:** executable binary program
  - `a.out`

- Combines several object files into a single executable (*linking*)
- Enables separate compilation/assembling of files
  - Changes to one file do not require recompiling of whole program

Linking

1) Take text segment from each `.o` file and put them together
2) Take data segment from each `.o` file, put them together, and concatenate this onto end of text segments
3) Resolve References
   - Go through Relocation Table; handle each entry



Machine Instruction Example

```
*dest = t;
```

- C Code
  - 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
```

- Object Code
  - 3-byte instruction
  - Stored at address **0x40059e**

Object Code

Code for `sumstore`

```
0x0400536:
    0x48
    0x01
    0xfe
    0x48
    0x89
    0x32
    0xc3
```

- **Total of 7 bytes**
- **Each instruction 1, 3, or 5 bytes (not shown)**
- **Starts at address `0x0400536`**

Disassembling Object Code

- Disassembled:

```
0000000000400536 <sumstore>:
              36  37  38
  400536:     48  01  fe        add    %rdi,%rsi
              39  3a  3b
  400539:     48  89  32        mov    %rsi,(%rdx)
              3c
  40053c:     c3                retq
```

address of instruction   object code bytes (hex)   interpreted assembly instructions

- **Disassembler** (`objdump -d sum`)
  - Useful tool for examining object code (`man 1 objdump`)
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can run on either `a.out` (complete executable) or `.o` file

Alternate Disassembly in GDB

```
$ gdb sum
(gdb) disassemble sumstore
Dump of assembler code for function sumstore:
   0x0000000000400536 <+0>:       add    %rdi,%rsi
   0x0000000000400539 <+3>:       mov    %rsi,(%rdx)
   0x000000000040053c <+6>:       retq
End of assembler dump.

(gdb) x/7bx sumstore
0x400536 <sumstore>:0x48   0x01   0xfe   0x48   0x89   0x32   0xc3
```

- Within gdb debugger (`gdb sum`):
  - `disassemble sumstore`: disassemble procedure
  - `x/7bx sumstore`: show 7 bytes starting at `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:
30001001:
30001003:
30001005:
3000100a:
```

Reverse engineering forbidden by Microsoft End User License Agreement

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

Loader

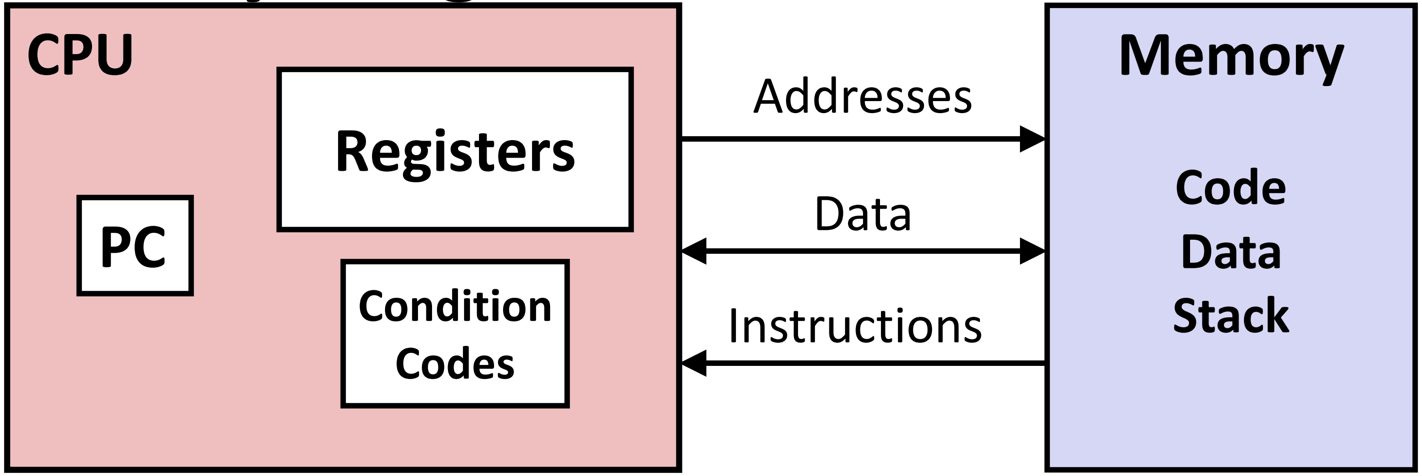
- **Input:** executable binary program, command-line arguments
  - `./a.out arg1 arg2`
- **Output:** <program is run>

- Loader duties primarily handled by OS/kernel
  - More about this when we learn about processes
- Memory sections (Instructions, Static Data, Stack) are set up
- Registers are initialized

Machine Programming I: Basics

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

Assembly Programmer’s 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

- Integral data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
  - Different registers for those (*e.g.* `%xmm1`, `%ymm2`)
  - Come from *extensions to x86* (SSE, AVX, …)
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
- Two common syntaxes
  - "AT&T": used by our course, slides, textbook, gnu tools, …
  - "Intel": used by Intel documentation, Intel tools, …
  - Must know which you're reading

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

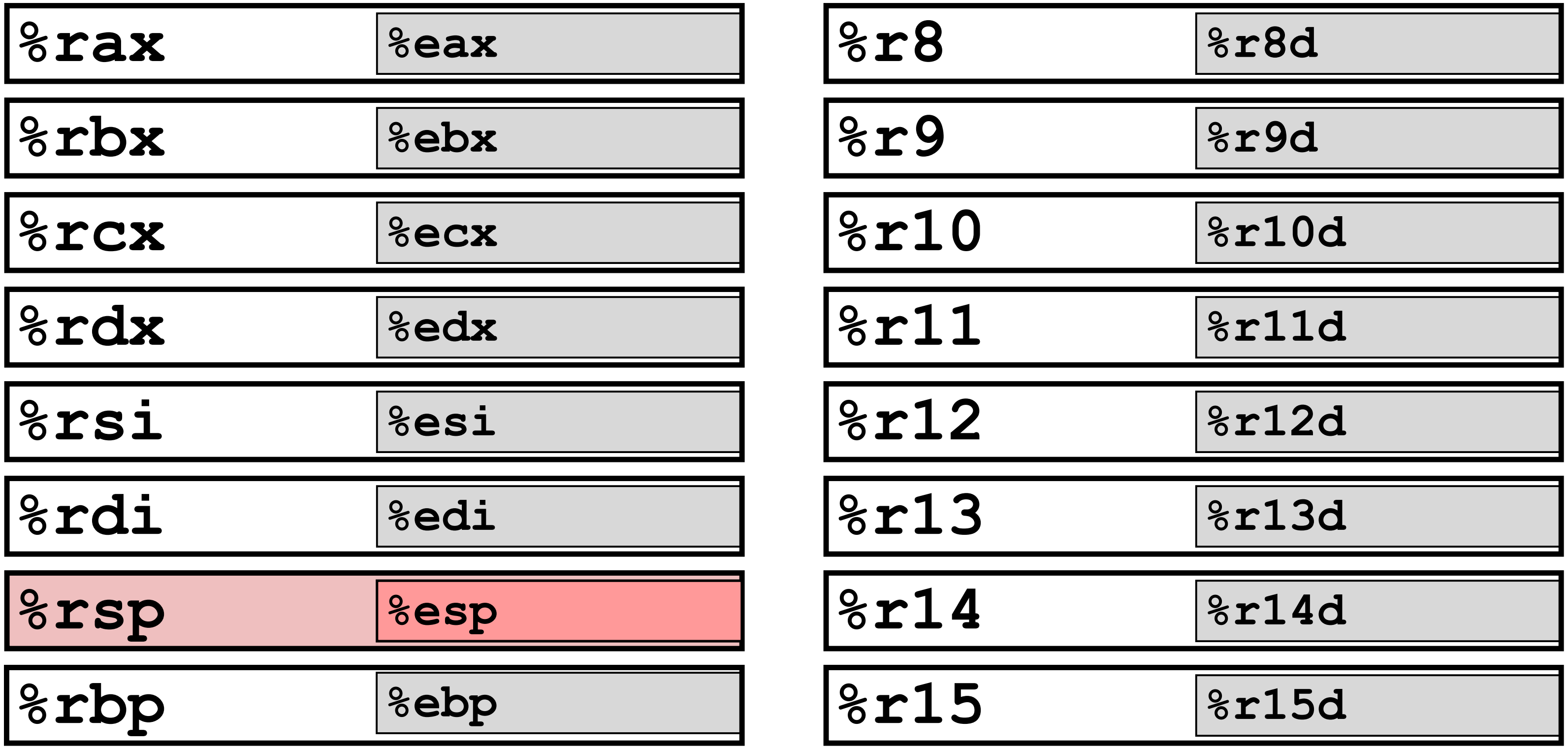
What is a Register?

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

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

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

x86-64 Integer Registers – 64 bits wide

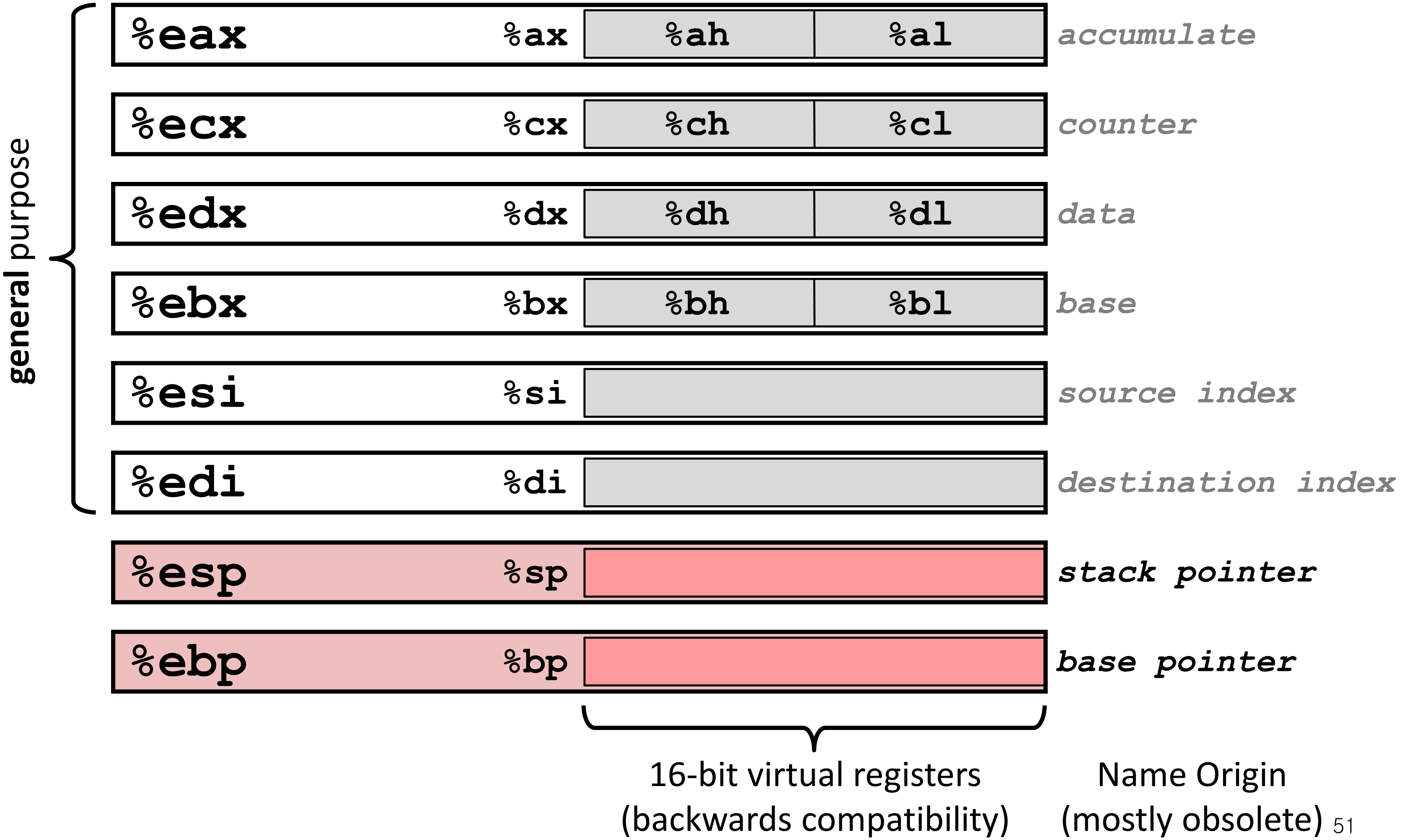


%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

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

Some History: IA32 Registers – 32 bits wide

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)

Name Origin
(mostly obsolete)

Memory vs. Registers

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

Three Basic Kinds of Instructions

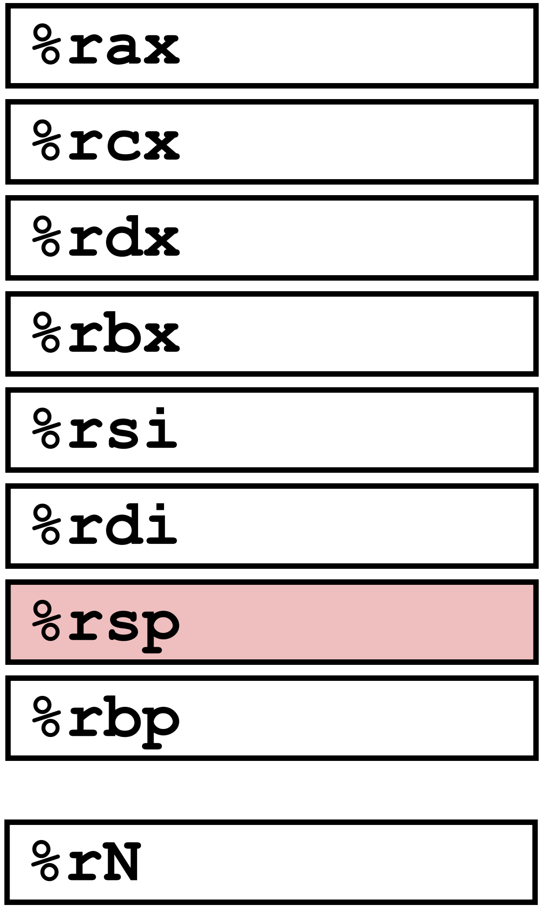
- **Transfer data between memory and register**
  - ***Load*** data from memory into register
    - %reg = Mem[address]
  - ***Store*** register data into memory
    - Mem[address] = %reg

Remember: memory is indexed just like an array[] of bytes!

- **Perform arithmetic function on register or memory data**
  - c = a + b; z = x << y; i = h & g;
- **Control flow: what instruction to execute next**
  - Unconditional jumps to/from procedures
  - Conditional branches

Operand types

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



%rax

%rcx

%rdx

%rbx

%rsi

%rdi

%rsp

%rbp

%rN

Moving Data

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

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

`movq` Operand Combinations

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

Cannot do memory-memory transfer with a single instruction

How would you do it?

Question

- Which of the following statements is TRUE?

A. For `float f`, `(f+2 == f+1+1)` always returns TRUE

B. The width of a "word" is part of a system's *architecture* (as opposed to *microarchitecture*)

C. Having more registers increases the performance of the hardware, but decreases the performance of the software

D. Mem to Mem (src to dst) is the only disallowed operand combination in x86-64

Summary

- x86-64 is a complex instruction set computing (CISC) architecture
- **Registers** are named locations in the CPU for holding and manipulating data
  - x86-64 uses 16 64-bit wide registers
- Assembly operands include immediates, registers, and data at specified memory locations
- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts
- Assembly Basics: Registers, operands, move
  - The x86-64 move instructions cover wide range of data movement forms

Q&A