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
```

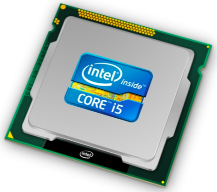
Machine code:

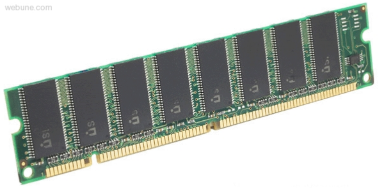
```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8 Mac

Computer system:



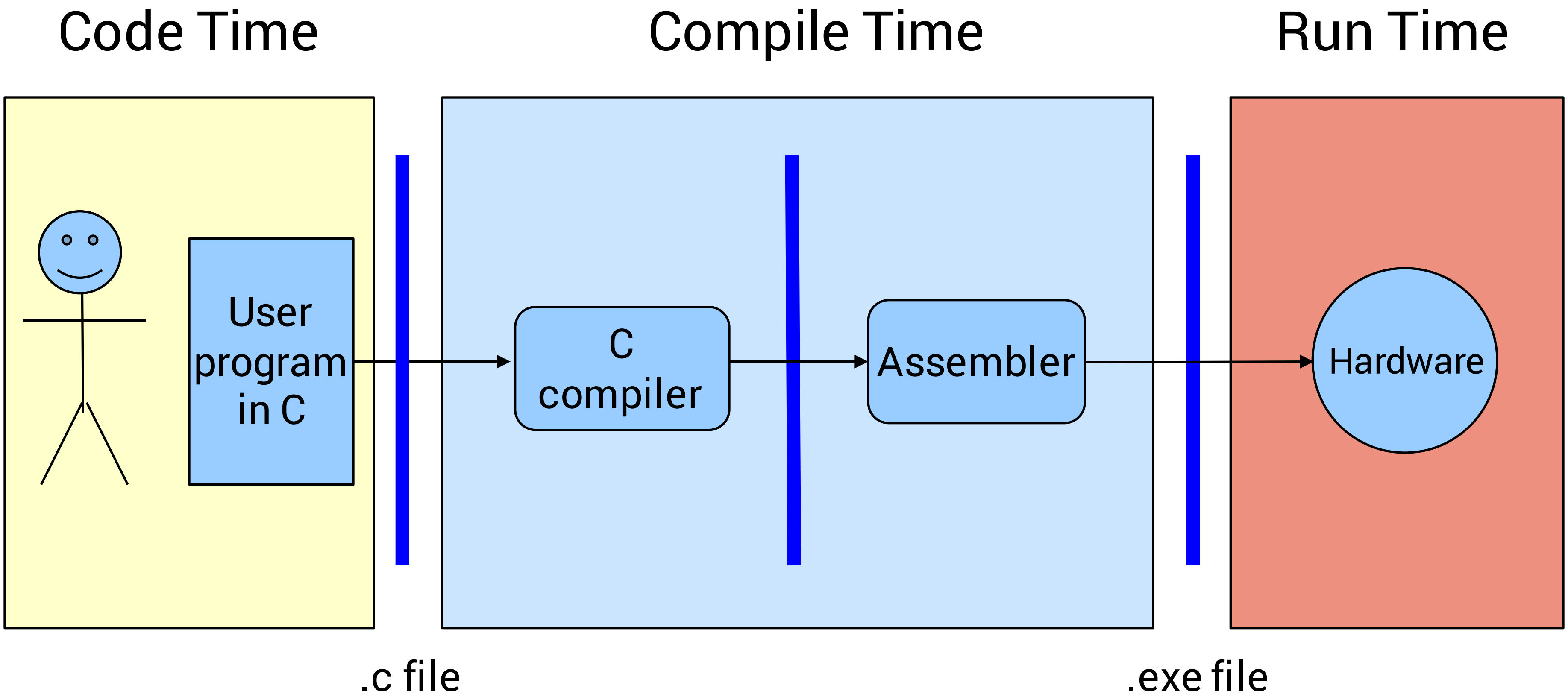


Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

Basics of Machine Programming & Architecture

- **What is an ISA (Instruction Set Architecture)?**
- **A brief history of Intel processors and architectures**
- **C, assembly, machine code**

Translation



What makes programs run fast?

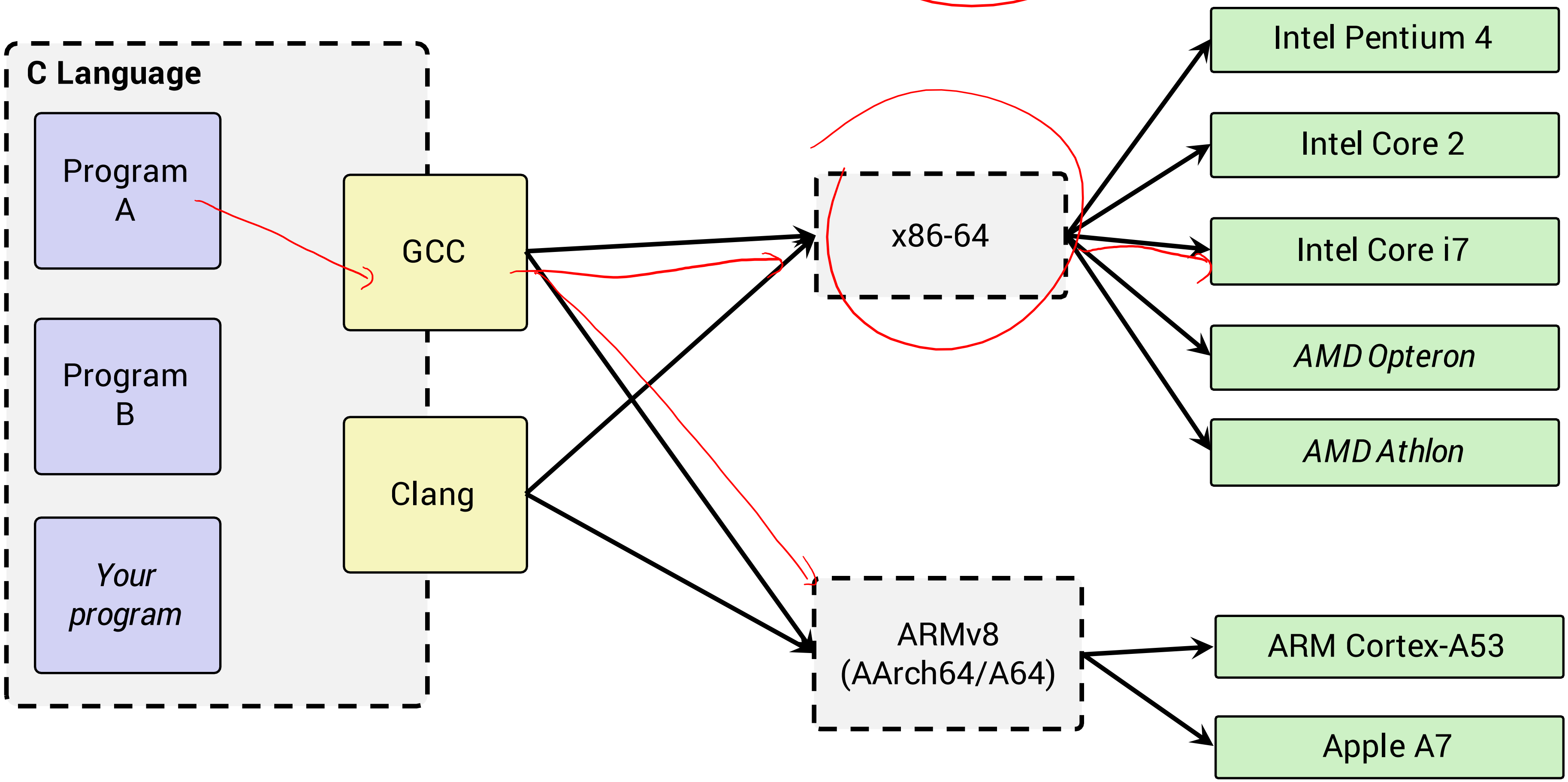
HW Interface Affects Performance

Source code
Different applications or algorithms

Compiler
Perform optimizations, generate instructions

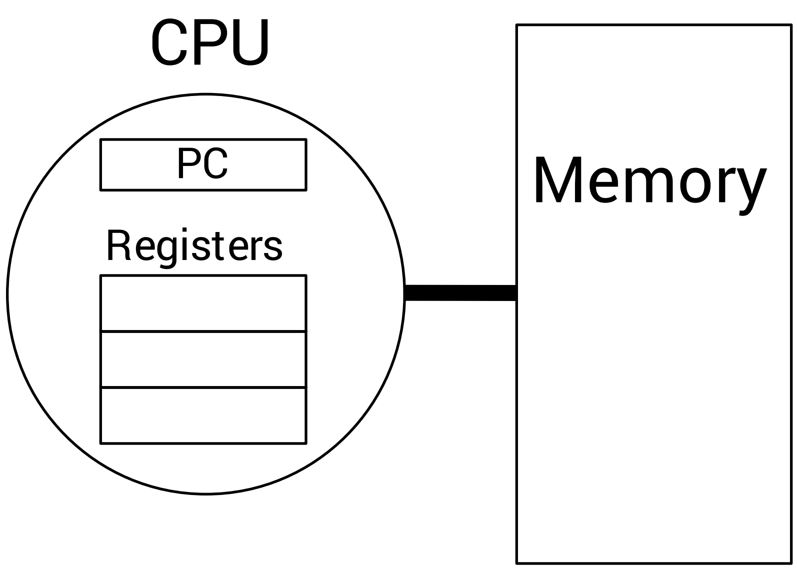
Architecture
Instruction set

Hardware
Different implementations



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



General ISA Design Decisions

- **Instructions**
  - What instructions are available? What do they do?
  - How are they encoded?

e.g. 1011 = add

- **Registers**
  - How many registers are there?
  - How wide are they?

word size

- **Memory**
  - How do you specify a memory location?

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, highly specialized instructions
    - 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**
386	**1985**	**275K**	**16-33**
Pentium 4E	**2004**	**125M**	**2800-3800**
Core 2	**2006**	**291M**	**1060-3500**
Core i7	**2008**	**731M**	**1700-3900**

- **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

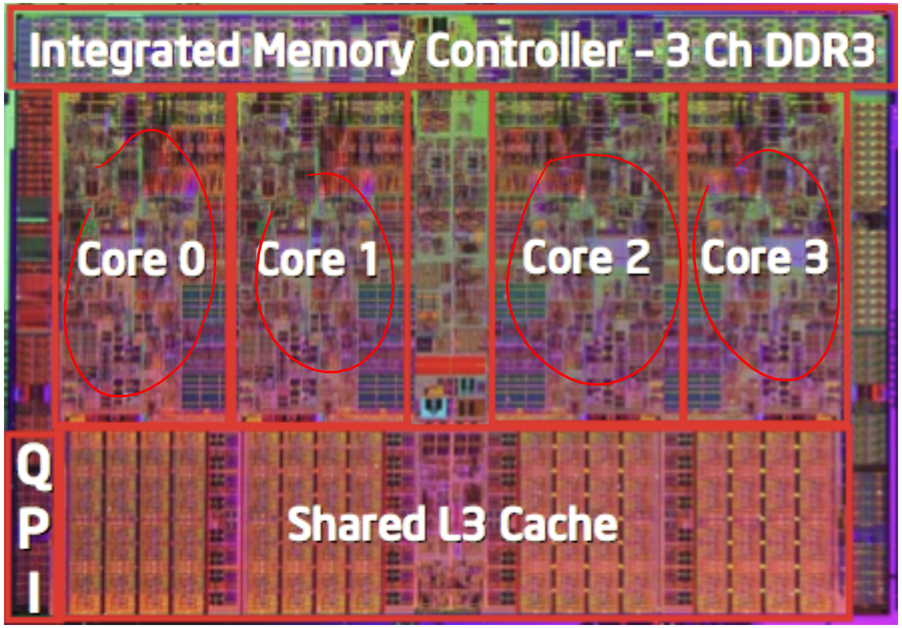
same speed

Intel x86 Processors

- **Machine Evolution**

486	1989	1.9M
Pentium	1993	3.1M
Pentium/MMX	1997	4.5M
Pentium Pro	1995	6.5M
Pentium III	1999	8.2M
Pentium 4	2001	42M
Core 2 Duo	2006	291M
Core i7	2008	731M

Intel Core i7



- **Added Features**
  - Instructions to support multimedia operations
    - Parallel operations on 1, 2, and 4-byte data ("SIMD")
  - Instructions to enable more efficient conditional operations
  - Hardware support for virtualization (virtual machines)
  - More cores!

VTX

More information

- **References for Intel processor specifications:**
  - Intel's "automated relational knowledgebase":
    - http://ark.intel.com/
  - Wikipedia:
    - http://en.wikipedia.org/wiki/List_of_Intel_microprocessors

x86 Clones: Advanced Micro Devices (AMD)

- **Same ISA, different implementation**
- **Historically**
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- **Then**
  - Recruited top circuit designers from Digital Equipment and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension of x86 to 64 bits

Intel's Transition to 64-Bit

- **Intel attempted radical shift from IA32 to IA64 (2001)**
  - Totally different architecture (Itanium) and ISA than x86
  - Executes IA32 code only as legacy
  - Performance disappointing
- **AMD stepped in with *evolutionary* solution (2003)**
  - x86-64 (also called "AMD64")
- **Intel felt obligated to focus on IA64**
  - Hard to admit mistake or that AMD is better
- **Intel announces "EM64T" extension to IA32 (2004)**
  - Extended Memory 64-bit Technology
  - Almost identical to AMD64!
- **Today: all but low-end x86 processors support x86-64**
  - But, lots of code out there is still just IA32

Our Coverage in 351

- **x86-64**
  - The new 64-bit x86 ISA – all lab assignments use x86-64!
  - Book covers x86-64

- **Previous versions of CSE 351 and 2nd edition of textbook covered IA32 (traditional 32-bit x86 ISA) <u>and</u> x86-64**

- **We will only cover x86-64 this quarter**

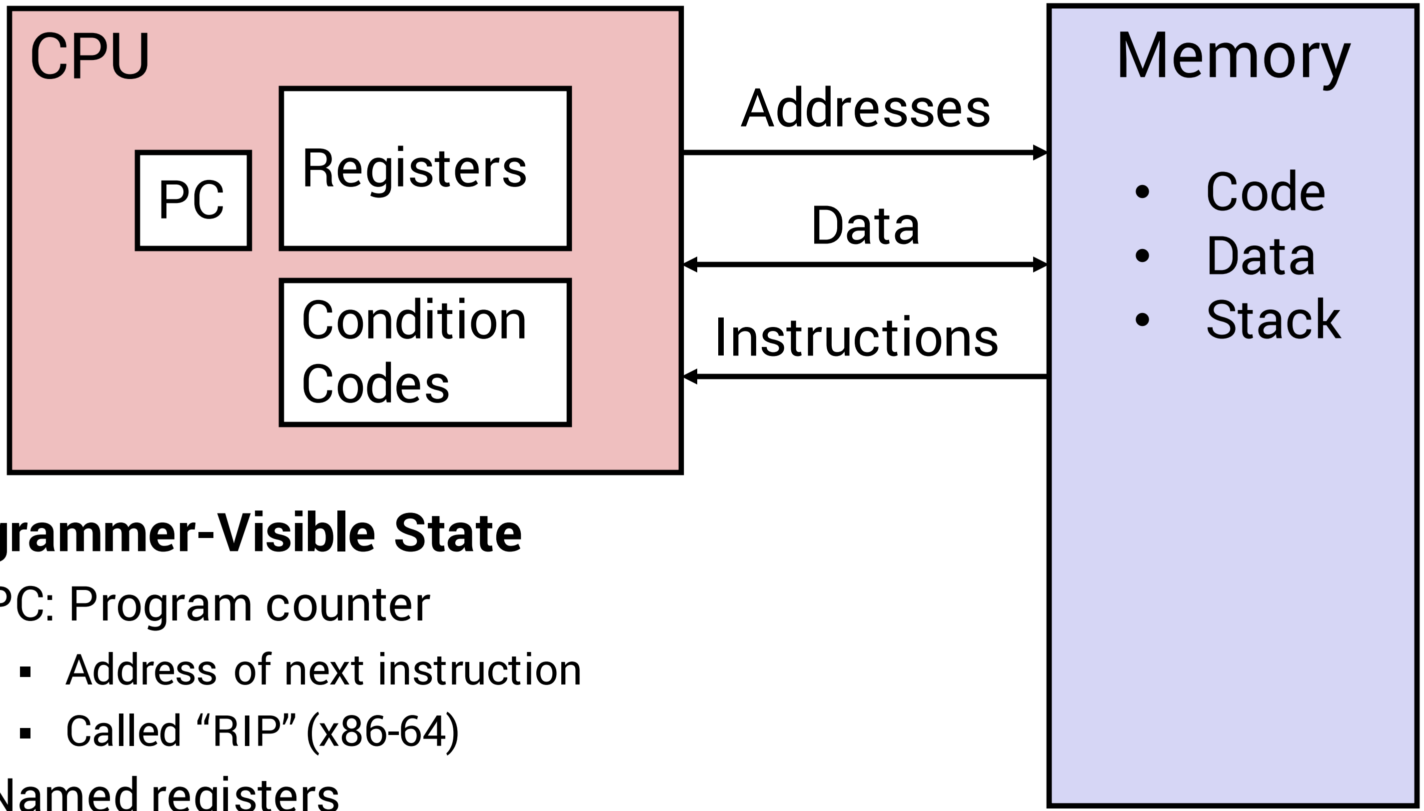
Definitions

- **Architecture: (also instruction set architecture or ISA) The parts of a processor design that one needs to understand to write assembly code**
  - "What is directly visible to software"
- ~~**Microarchitecture**~~**: Implementation of the architecture**
  - CSE/EE 469, 470

- Number of registers? Yes – names
- How about CPU frequency? No
- Cache size? Memory size? No

address space = 16 EB

Assembly Programmer's View

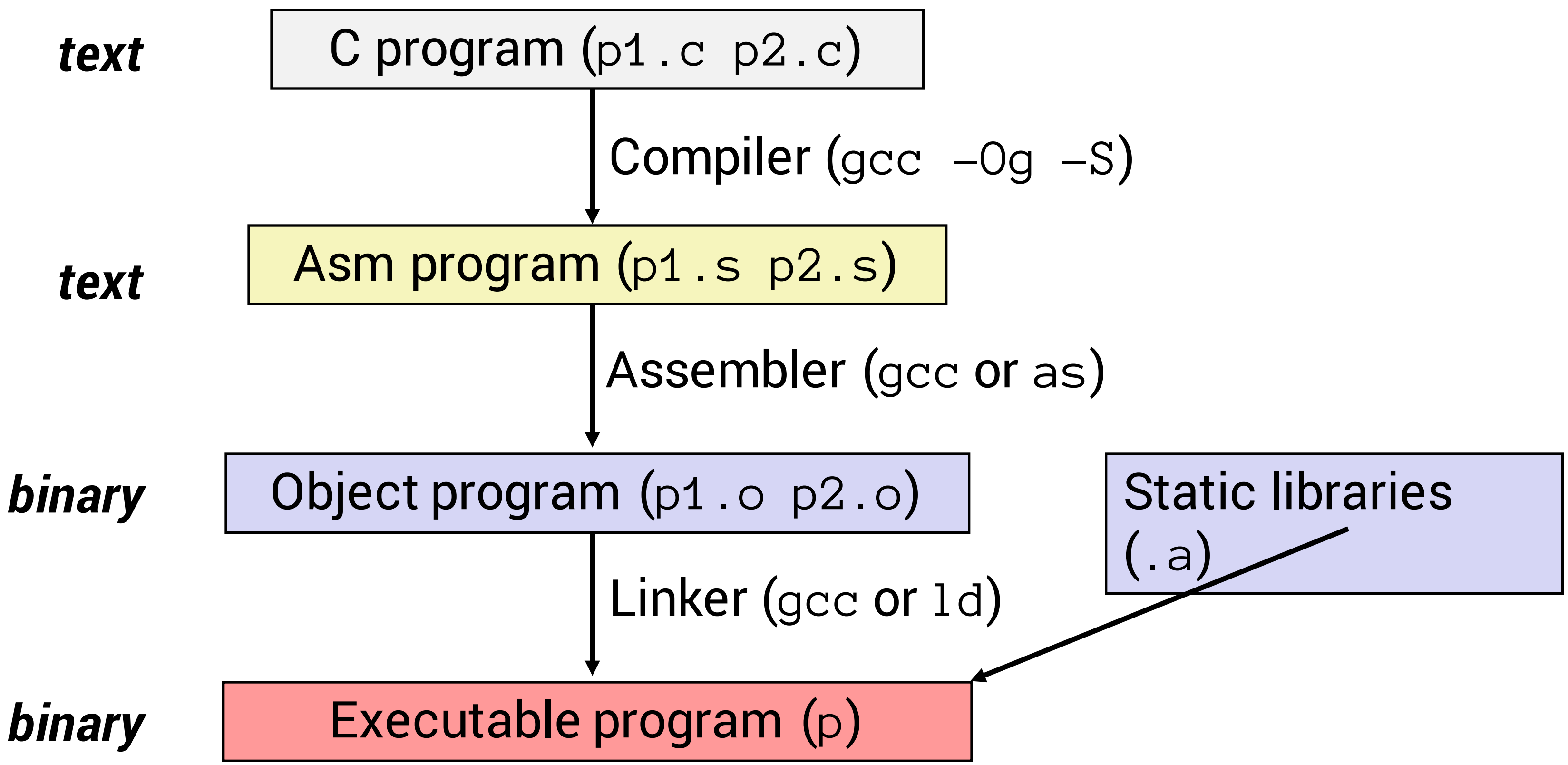


- **Programmer-Visible State**
  - PC: Program counter
    - Address of next instruction
    - Called "RIP" (x86-64)
  - Named registers
    - Heavily used program data
    - Together, called "register file"
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

- Memory
  - Byte addressable array
  - Code and user data
  - Includes *Stack* (for supporting procedures, we'll come back to that)

Turning C into Object Code

- **Code in files `p1.c p2.c`**
- **Compile with command: `gcc –Og p1.c p2.c -o p`**
  - Use basic optimizations (`–Og`) [New to recent versions of GCC]
  - Put resulting machine code in file `p`



Compiling Into Assembly

C Code (sum.c)

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

y += x
*dest = y

Generated x86-64 Assembly

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

addq ⇔ y += x

Produced by command:

```
gcc -Og -S sum.c
```

Generates file: `sum.s`

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

https://godbolt.org/g/5hQk8a

Machine Instruction Example

```
*dest = t;
```

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

```
movq %rsi, (%rdx)
```

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

`t`:	Register	`%rsi`
`dest`:	Register	`%rdx`
`*dest`:	Memory	**M[**`%rdx`]

```
0x400539:   48 89 32
```

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

Object Code

Code for `sumstore`

```
0x00400536 <sumstore>:
   0x48
   0x01
   0xfe
   0x48
   0x89
   0x32
   0xc3
```

- Total of 7 bytes
- Each instruction here 1 or 3 bytes
- Starts at address `0x00400536`

- **Assembler**
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
- **Linker**
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for `malloc`, `printf`
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution

Disassembling Object Code

Disassembled

```
0000000000400536 <sumstore>:
  400536:   48 01 fe        add    %rdi,%rsi
  400539:   48 89 32        mov    %rsi,(%rdx)
  40053c:   c3              retq
```

- **Disassembler**

  `objdump -d sum`

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

Alternate Disassembly 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 sumstore0x400536 <sumstore>:    0x48
    0x01     0xfe     0x48    0x89    0x32    0xc3
```

- **Within gdb Debugger**

  `gdb sum`

  `disassemble sumstore`

  - Disassemble procedure

  `x/14bx sumstore`

  - Examine the 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:  5
30001001:  8
30001003:  6
30001005:  6
3000100a:  6
```

Reverse engineering forbidden by Microsoft End User License Agreement

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