

# CS 134

# Operating Systems

---

January 28, 2019

PC Hardware and x86

# Outline

---

- **PC Architecture**
- **x86 Instruction Set**
- **gcc Calling Conventions**
- **Emulation**

# PC

---

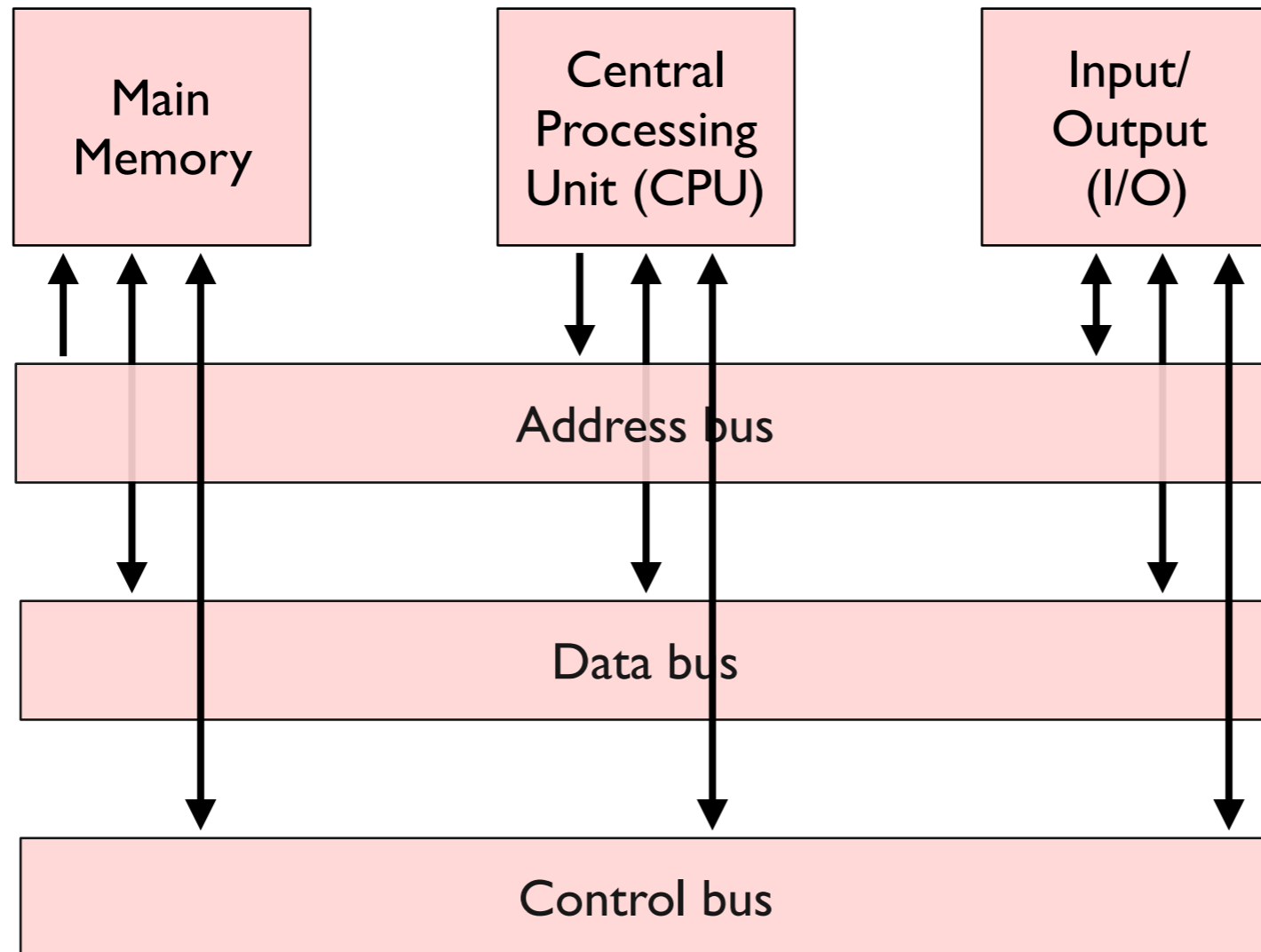


# PC Board



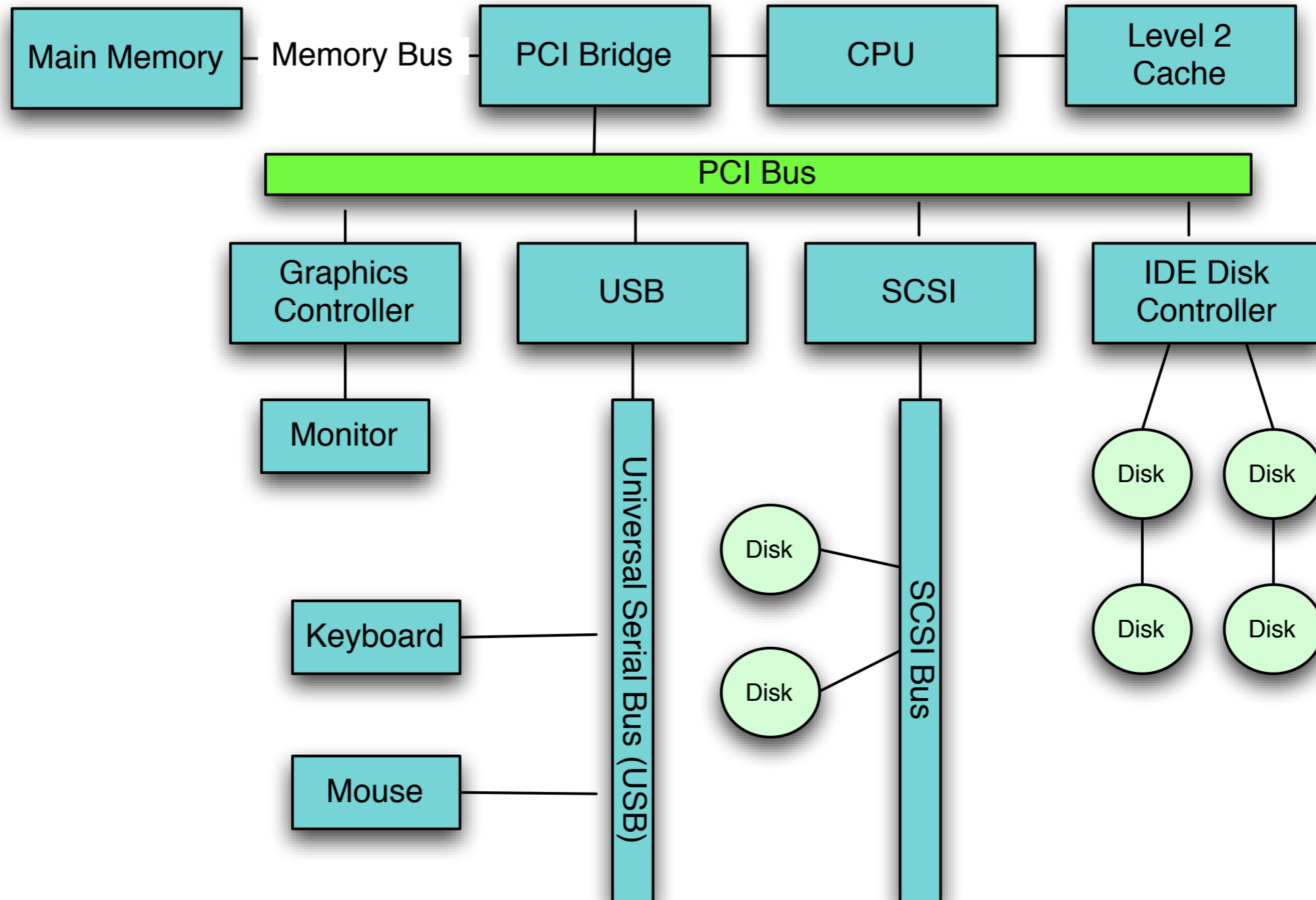
# High-Level

---



# Bus

Bus: a set of wires and a protocol for communicating



# Stored-Program Computer

---

## CPU

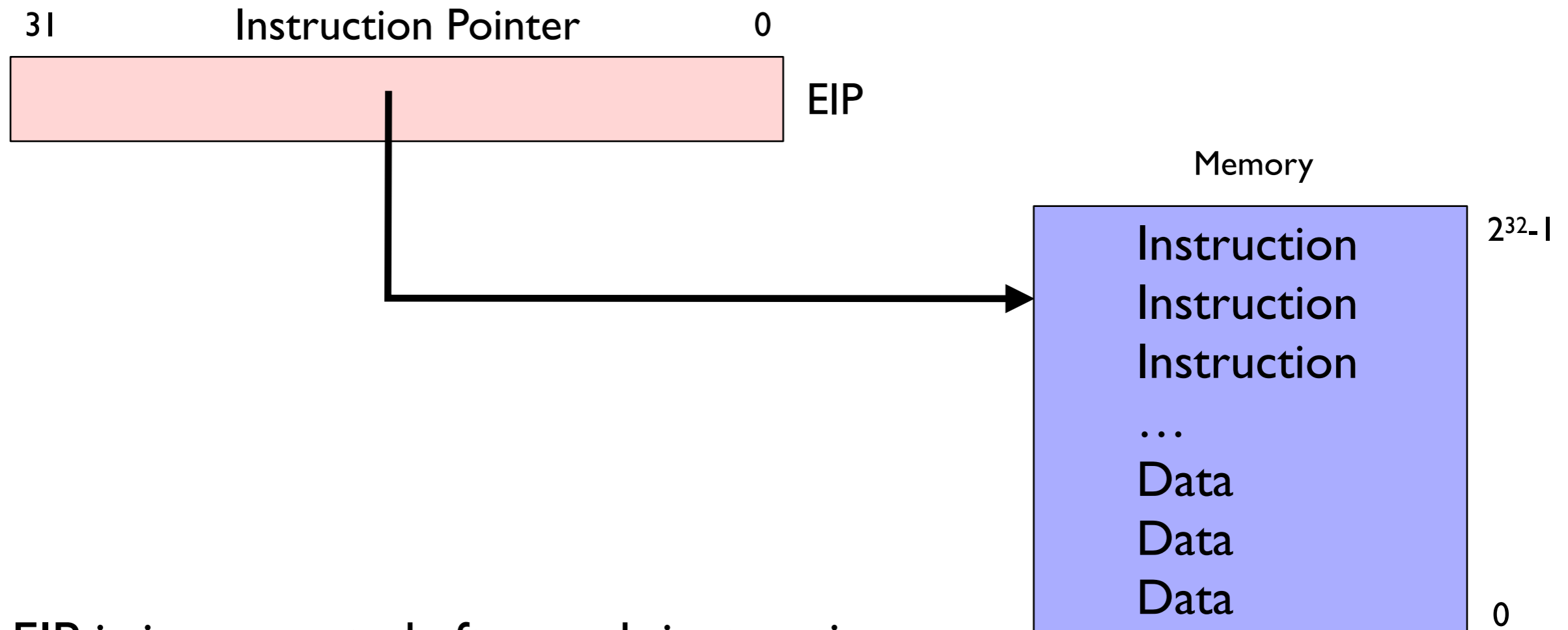
```
for (;;) {  
  Fetch next instruction  
  Execute next instruction  
}
```

## Memory

```
Instruction  
Instruction  
Instruction  
...  
Data  
Data  
Data
```

- CPU interprets instructions
- Instructions read/write data

# x86 implementation



- EIP is incremented after each instruction
- Instructions are different length
- EIP modified by CALL, RET, JMP, and conditional JMP

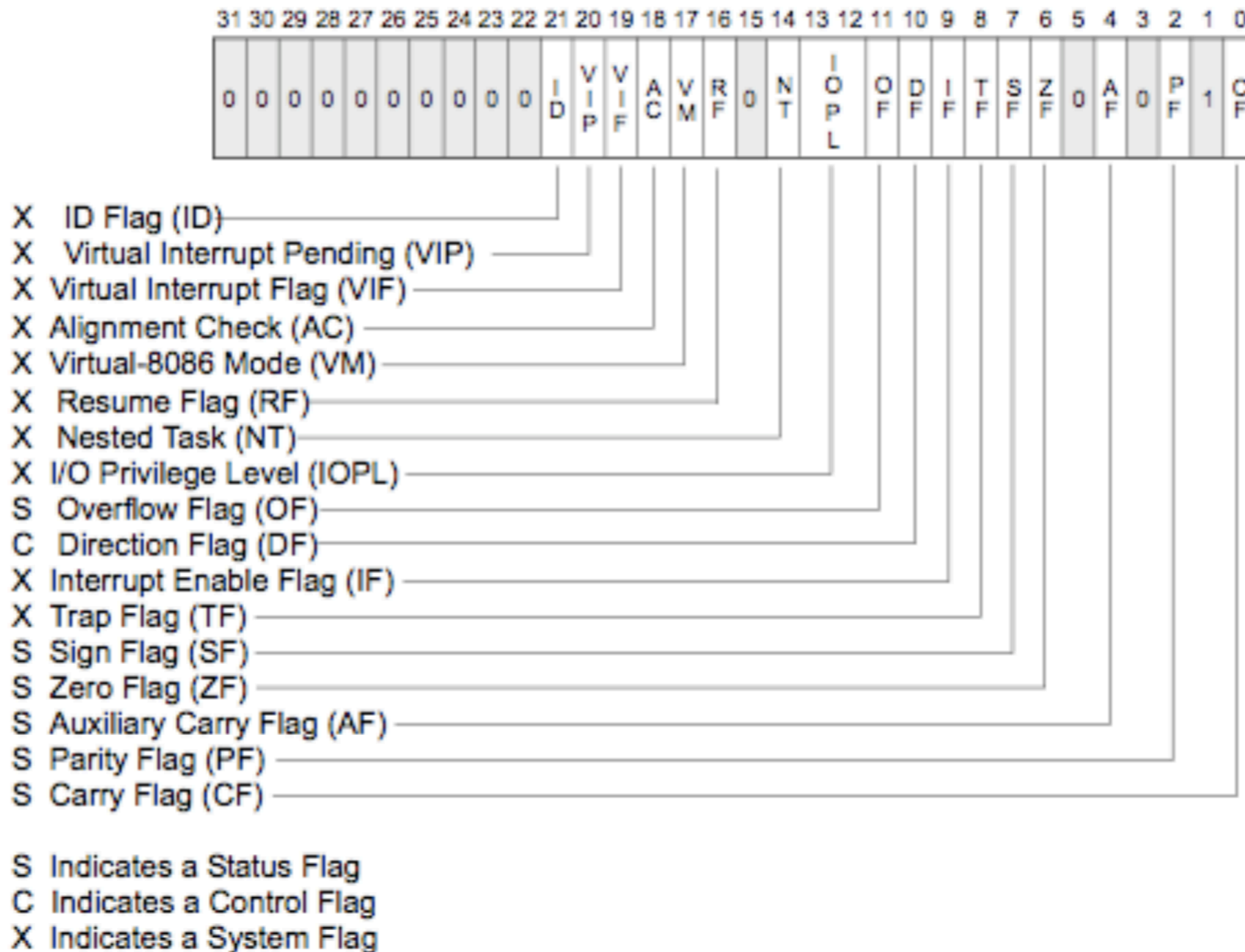


# Registers for work space

	31	16	15	8	7	0	<b>16-bit</b>	<b>32-bit</b>
Data {		AH		AL			AX	EAX
		BH		BL			BX	EBX
		CH		CL			CX	ECX
		DH		DL			DX	EDX
Address {		BP						EBP
		SI						ESI
		DI						EDI
		SP						ESP

- 8-, 16-, and 32-bit versions
- By convention, some registers are for specific purposes.
- Example: `ADD EAX, 10`

# EFLAGS register



- Test instructions (TEST EAX, 0)

## Memory: more work space

---

<code>movl %eax, %edx</code>	<code>edx = eax;</code>	<i>register mode</i>
<code>movl \$0x123, %edx</code>	<code>edx = 0x123;</code>	<i>immediate</i>
<code>movl 0x123, %edx</code>	<code>edx = *(int32_t*)0x123;</code>	<i>direct</i>
<code>movl (%ebx), %edx</code>	<code>edx = *(int32_t*)ebx;</code>	<i>indirect</i>
<code>movl 4(%ebx), %edx</code>	<code>edx = *(int32_t*)(ebx+4);</code>	<i>displaced</i>

- Memory instructions: MOV, PUSH, POP, etc.
- Most instructions can take a memory address

# Stack memory and operations

---

<u>Example instruction</u>	<u>What it does</u>
<code>pushl %eax</code>	<code>subl \$4, %esp movl %eax, (%esp)</code>
<code>popl %eax</code>	<code>movl (%esp), %eax addl \$4, %esp</code>
<code>call 0x12345</code>	<code>pushl %eip (*) movl \$0x12345, %eip (*)</code>
<code>ret</code>	<code>popl %eip (*)</code>

- The stack grows down
- Used to implement procedure calls

# More memory

---

- 8086: 16-bit registers, and 20-bit address bus
- Extra 4 bits come from *segment registers*:
  - CS: Code segment (for EIP)
  - SS: Stack segment (for SP, and BP)
  - DS: Data segment (for load/store via other registers)
  - ES: Another data segment (for string operations)
- **virtual->physical translation:**
  - $pa = va + seg * 16$ 
    - for example, CS of 4096, with an EIP of 0 executes code at 65536

# And More Memory

---

- 80386: 32-bit data and address busses
- Early 2000's: x86-64: 64-bit data and address busses
- Backwards compatibility:
  - Boot in 16-bit mode (*real mode*). Boot.S switches to *protected mode* with 32-bit addresses.
  - Prefix 0x66 gives you 32-bit addresses:
    - MOVW = 0x66 MOVW
  - .code32 in boot.S tells assembler to insert 0x66
- 80386 also added virtual memory addresses:
  - Segment registers are indices into a page table
  - Page table hardware

# I/O Space and Instructions

---

```
#define DATA_PORT    0x378
#define STATUS_PORT   0x379
#define BUSY          0x80
#define CONTROL_PORT  0x37A
#define STROBE        0x01

void lpt_putc(int c) {
    /* wait for printer to consume previous byte */
    while ((inb(STATUS_PORT) & BUSY) == 0)
        ;
    /* put the data on the parallel port */
    outb(DATA_PORT, c);

    /* tell the printer to look at the data */
    outb(CONTROL_PORT, STROBE);
    outb(CONTROL_PORT, 0);
}
```

- 8086: only 1024 I/O addresses

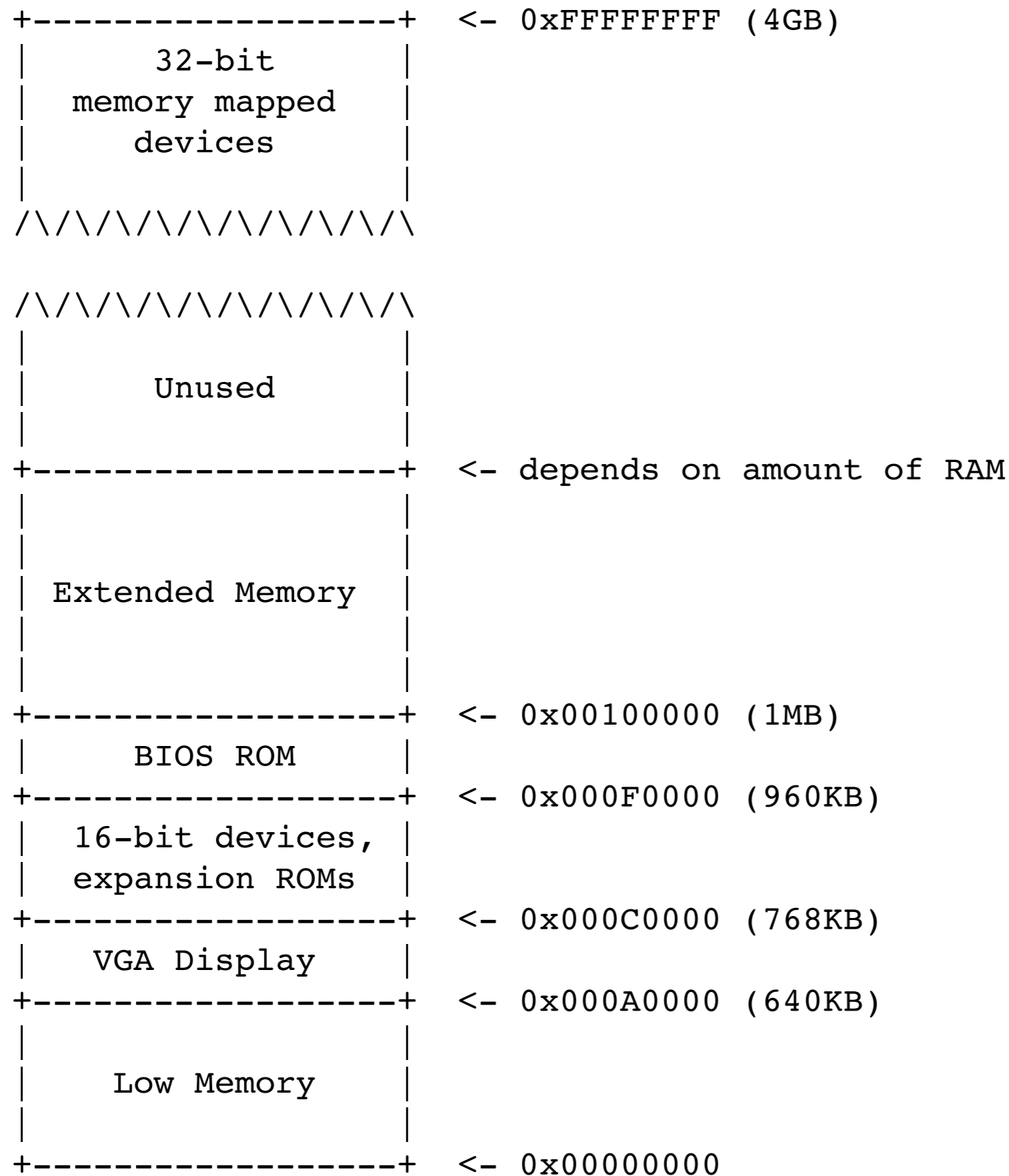
# Memory-Mapped I/O

---

- **Use normal addresses**
  - No need for special instructions
  - No 1024 limit
  - System controller routes to device
- **Works like “magic” memory**
  - Addressed and accessed like normal memory
  - But does not behave like real memory
  - Reads and writes have “side effects”
  - Read result can change due to external events



# Physical Memory Layout



# Outline

---

- PC Architecture
- **x86 Instruction Set**
- gcc Calling Conventions
- Emulation

# x86 Instruction Set

---

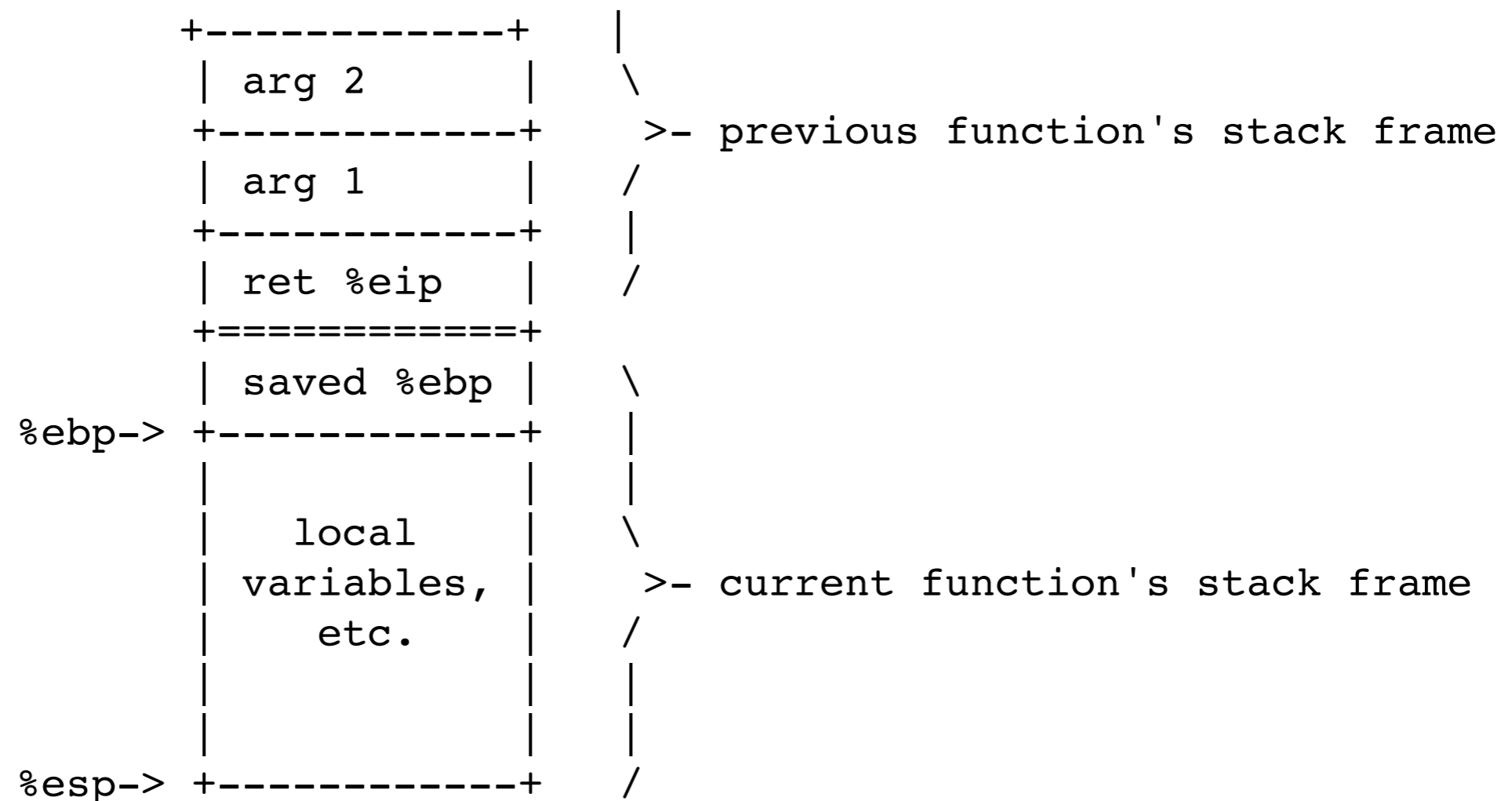
- **Instruction classes:**
  - Data movement: MOV, PUSH, POP, ...
  - Arithmetic: TEST, SHL, ADD, ...
  - I/O: IN/OUT
  - Control: JMP, JZ, JNZ, CALL, RET, ...
  - String: REP, MOVSB, ...
  - System: INT, IRET
- **Intel Architecture Manual volume 2: *the reference***
  - Intel syntax: `op dst, src`
  - AT&T (gcc/gas) syntax: `op src, dst`
    - uses b, w, l suffix on instructions to specify size of operands

# Outline

---

- PC Architecture
- x86 Instruction Set
- **gcc Calling Conventions**
- Emulation

# GCC Calling Conventions



- Saved EBPs form a chain: can walk stack
- Arguments and locals at fixed offsets from EBP

# GCC Use of the Stack

---

- GCC dictates how the stack is used: contract between caller and callee
  - At entry (just after CALL):
    - `%eip` points to the first instruction of the function
    - `%esp+4` points at the first argument
    - `%esp` points at the return address
  - At exit (just after RET):
    - `%eip` contains the return address
    - `%esp` points at the arguments pushed by caller
    - called function may have trashed arguments
    - `%eax`, (and `%edx` if 64-bit) contain return value
    - `%eax`, `%edx`, and `%ecx` may be trashed *Caller-saved registers*
    - `%ebp`, `%ebx`, `%esi`, and `%edi` have the values they did as of the CALL *Callee-saved registers*

# GCC Function Prologue

```
int main(void) { return f(8)+1; }
int f(int x) { return g(x); }
int g(int x) { return x+3; }
```

```
_main:
    prologue
    pushl %ebp
    movl %esp, %ebp
    body
    pushl $8
    call _f
    addl $1, %eax
    epilogue
    movl %ebp, %esp
    popl %ebp
    ret

_f:
    prologue
    pushl %ebp
    movl %esp, %ebp
    body
    pushl 8(%esp)
    call _g
    epilogue
    movl %ebp, %esp
    popl %ebp
    ret
```

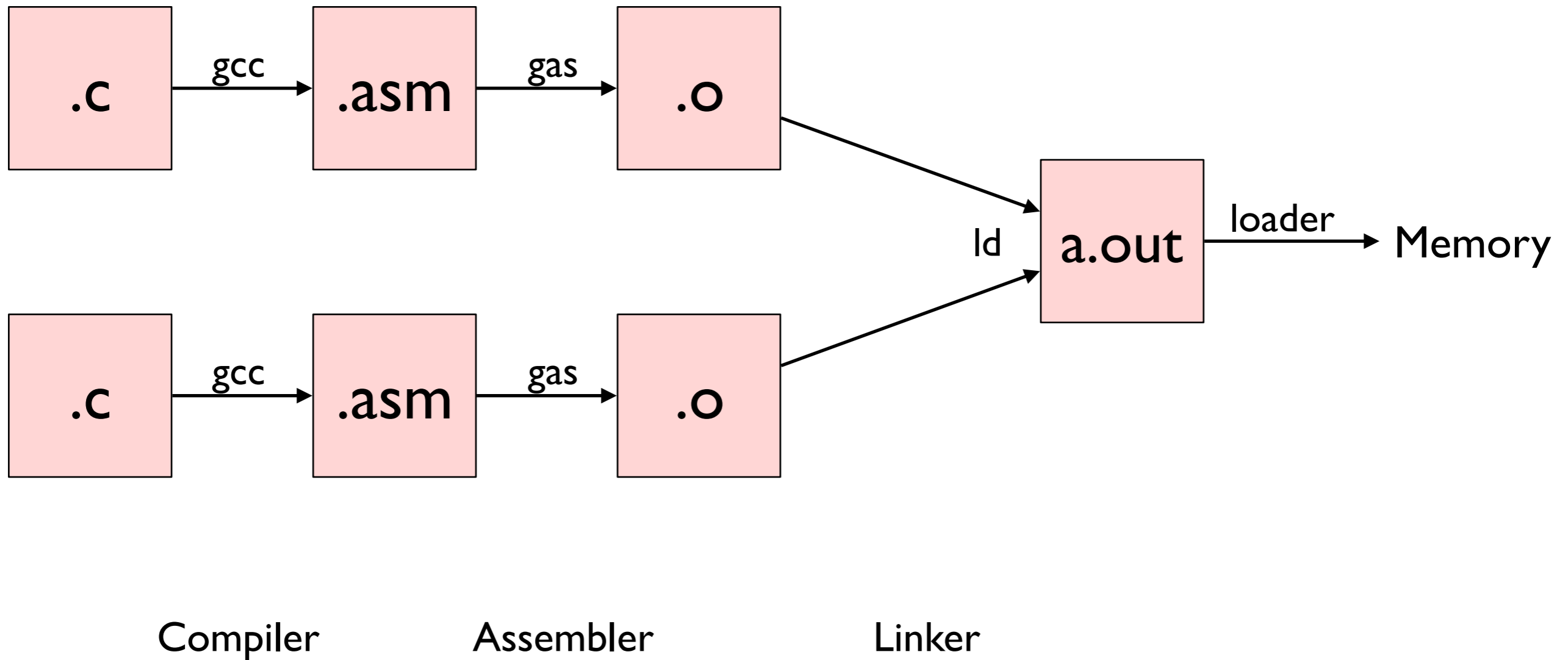
```
_g:
    prologue
    pushl %ebp
    movl %esp, %ebp
    save %ebx
    pushl %ebx
    body
    movl 8(%ebp), %ebx
    addl $3, %ebx
    movl %ebx, %eax
    restore %ebx
    popl %ebx
    epilogue
    movl %ebp, %esp
    popl %ebp
    ret
```

g optimized for space

```
_g:
    movl 4(%esp), %eax
    addl $3, %eax
    ret
```

# From C to Running Program

---





# Outline

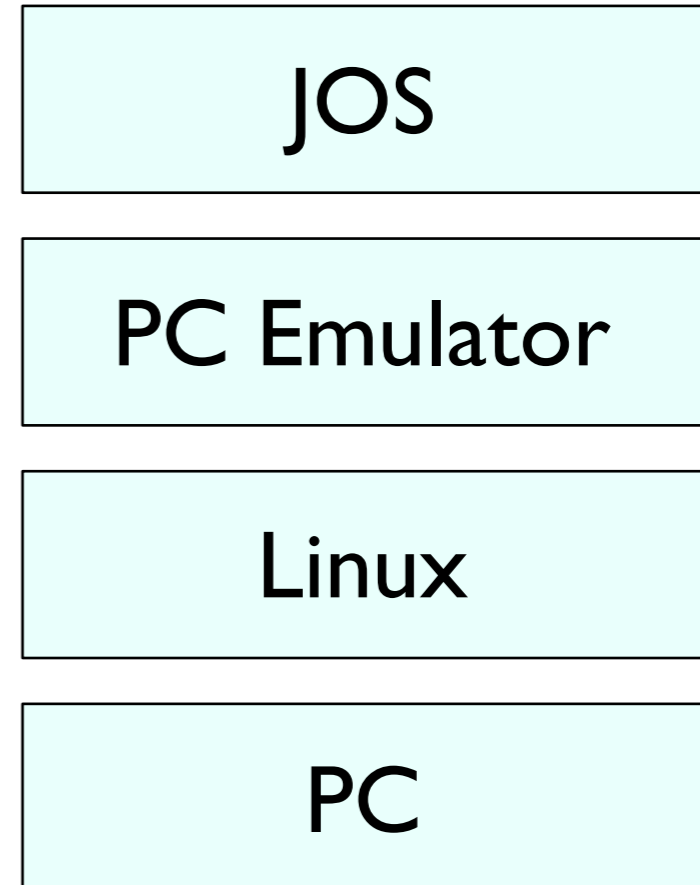
---

- PC Architecture
- x86 Instruction Set
- gcc Calling Conventions
- **Emulation**

# Development using PC Emulator

---

- **QEMU PC Emulator**
  - Does what a real PC does
  - Only implemented in software
- **Runs like a normal program on “host” operating system**



# Emulation of Memory

---

```
int32_t regs[8];  
#define REG_EAX 1;  
#define REG_EBX 2;  
#define REG_ECX 3;  
...  
int32_t eip;  
int16_t segregs[4];  
...
```

Registers

```
char mem[256*1024*1024];
```

Main memory

# Emulation of CPU

```
for (;;) {
    read_instruction();
    switch (decode_instruction_opcode()) {
    case OPACODE_ADD:
        int src = decode_src_reg();
        int dst = decode_dst_reg();
        regs[dst] = regs[dst] + regs[src];
        break;
    case OPACODE_SUB:
        int src = decode_src_reg();
        int dst = decode_dst_reg();
        regs[dst] = regs[dst] - regs[src];
        break;
        ...
    }
    eip += instruction_length;
}
```

# Emulation x86 Memory

```
#define LOW_MEMORY 640*KB
#define EXT_MEMORY 10*MB

uint8_t low_mem[LOW_MEMORY];
uint8_t ext_mem[EXT_MEMORY];
uint8_t bios_rom[64*KB];

uint8_t read_byte(uint32_t phys_addr) {
    if (phys_addr < LOW_MEMORY)
        return low_mem[phys_addr];
    else if (phys_addr >= 960*KB && phys_addr < 1*MB)
        return rom_bios[phys_addr - 960*KB];
    else if (phys_addr >= 1*MB && phys_addr < 1*MB+EXT_MEMORY) {
        return ext_mem[phys_addr-1*MB];
    }
    else ...
}

void write_byte(uint32_t phys_addr, uint8_t val) {
    if (phys_addr < LOW_MEMORY)
        low_mem[phys_addr] = val;
    else if (phys_addr >= 960*KB && phys_addr < 1*MB)
        ; /* ignore attempted write to ROM! */
    else if (phys_addr >= 1*MB && phys_addr < 1*MB+EXT_MEMORY) {
        ext_mem[phys_addr-1*MB] = val;
    }
    else ...
}
```

# Emulating Devices

---

- **Hard disk:** using a file of the host
- **VGA display:** draw in a host window
- **Keyboard:** host's keyboard API
- **Clock chip:** host's clock chip
- **etc.**