

6.828: PC hardware and x86

Frans Kaashoek
kaashoek@mit.edu

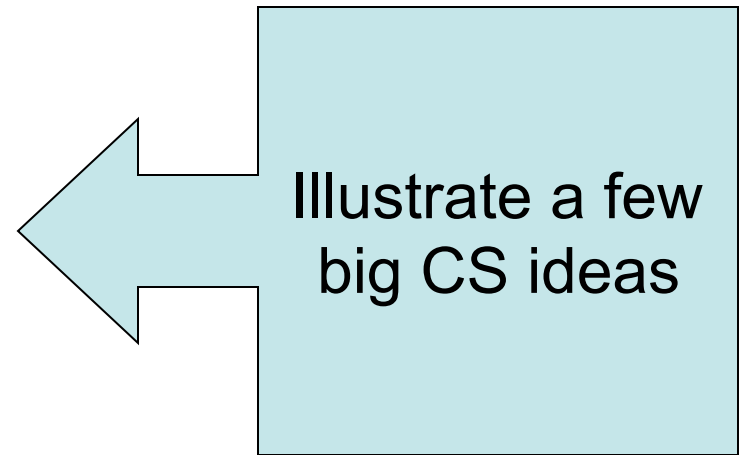
A PC



how to make it to do something useful?

Outline

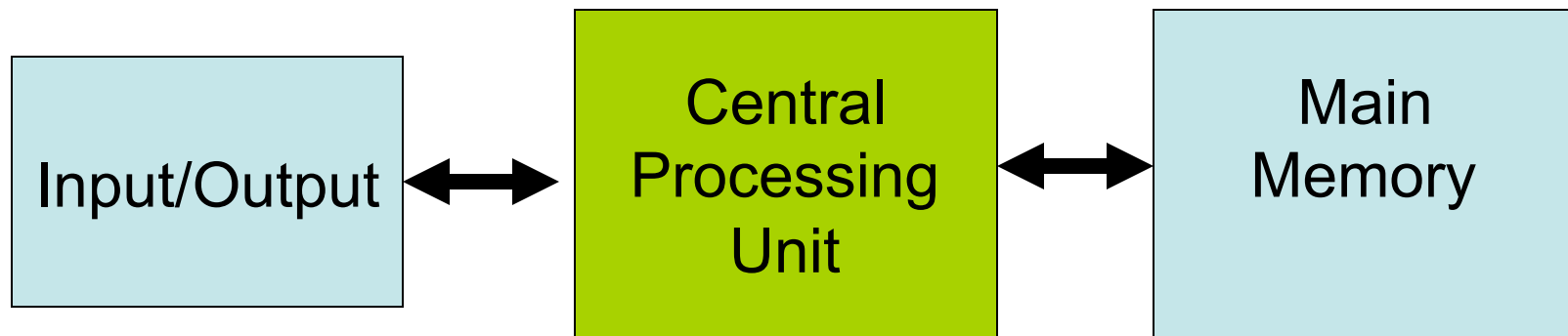
- PC architecture
- x86 instruction set
- gcc calling conventions
- PC emulation



PC board

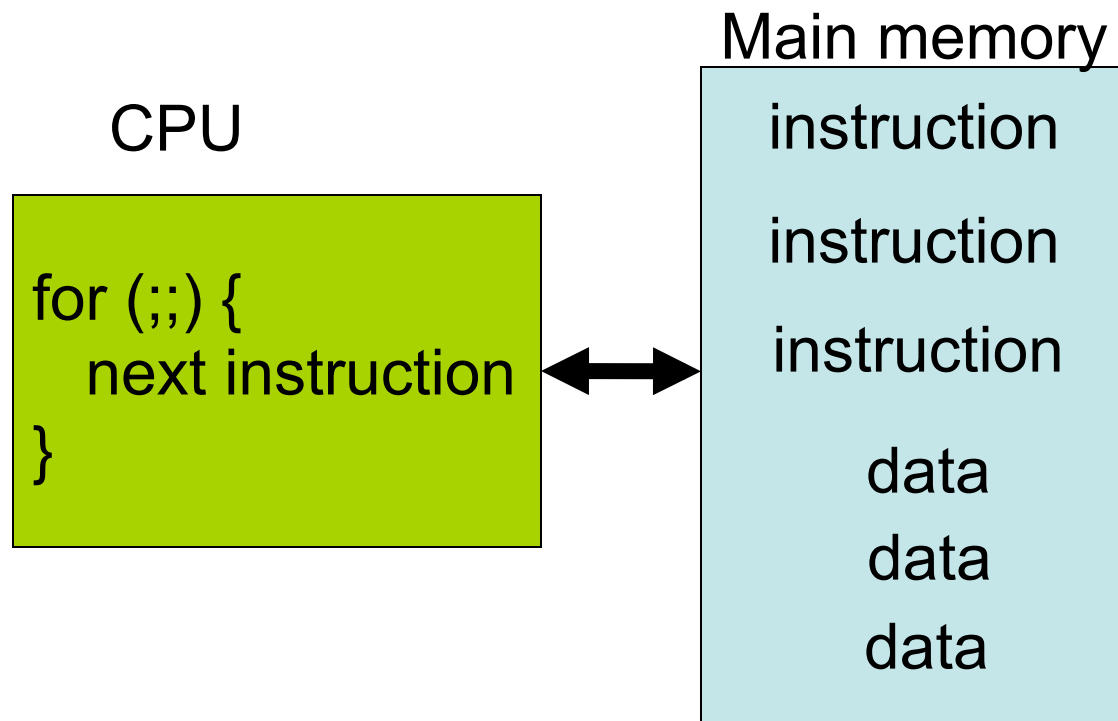


Abstract model



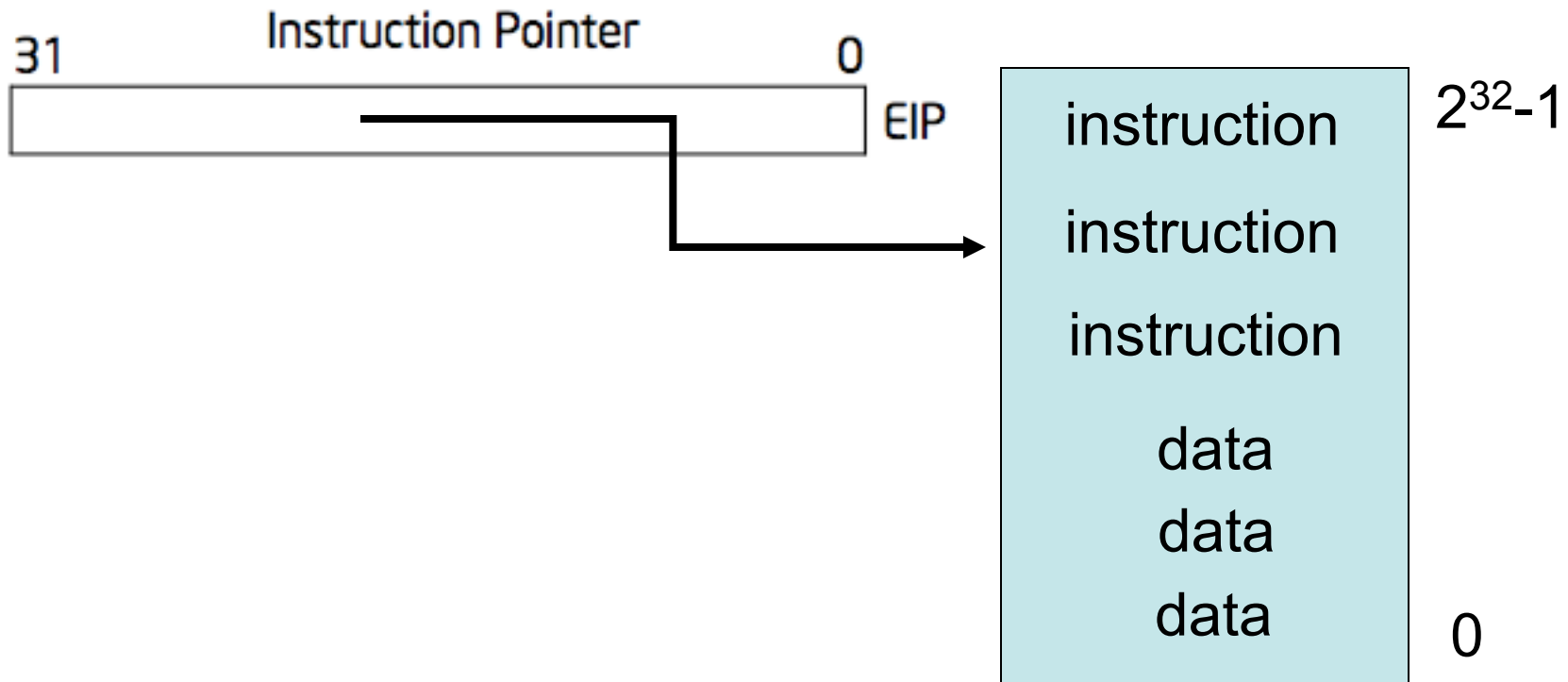
- I/O: communicating data to and from devices
- CPU: digital logic for performing computation
- Memory: N words of B bits

The stored program computer



- Memory holds *instructions* and *data*
- CPU *interpreter* of instructions

x86 implementation



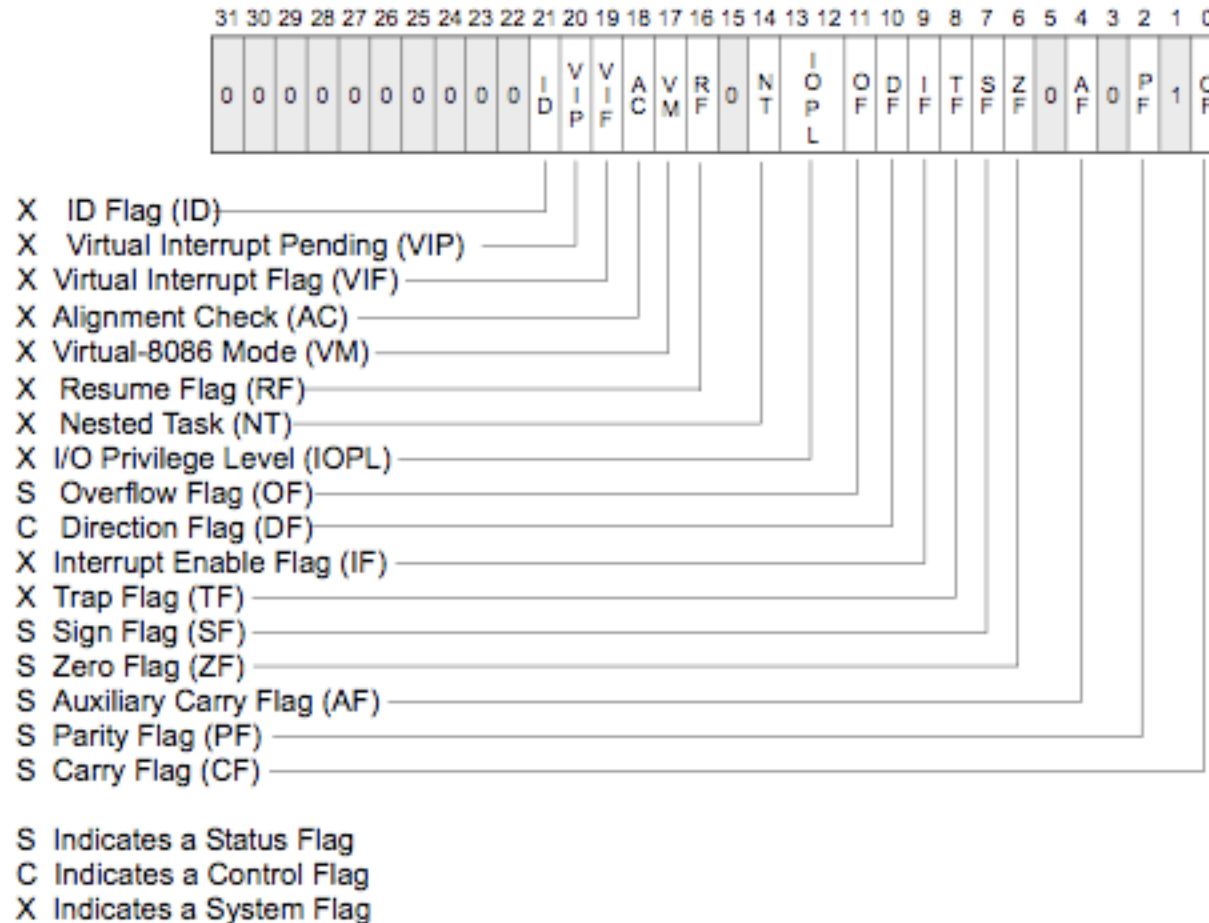
- EIP is incremented after each instruction
- Instructions are different length
- EIP modified by CALL, RET, JMP, and conditional JMP

Registers for work space

General-Purpose Registers						
31	16	15	8	7	0	
		AH		AL		16-bit 32-bit
		BH		BL		AX EAX
		CH		CL		BX EBX
		DH		DL		CX ECX
		BP				DX EDX
		SI				EBP
		DI				ESI
		SP				EDI
						ESP

- 8, 16, and 32 bit versions
- By convention some registers for special purposes
- Example: ADD EAX, 10
- Other instructions: SUB, AND, etc.

EFLAGS register



- Test instructions: TEST EAX, 0
- Conditional JMP instructions: JNZ address

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 + operations

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

- Stack grows down
- Use to implement procedure calls

More memory

- 8086 16 registers and 20-bit bus addresses
- The extra 4 bits come *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, destination for string ops
 - For example: CS=4096 to start executing at 65536
- Makes life more complicated
 - Cannot use 16 bit address of stack variable as pointer
 - Pointer arithmetic and array indexing across segment boundaries
 - For a far pointer programmer must include segment reg

And more memory

- 80386: 32 bit data and bus addresses
- Now: the transition to 64 bit addresses
- Backwards compatibility:
 - Boots in 16-bit mode, and boot.S switches to protected mode with 32-bit addresses
 - Prefix 0x66 gets you 32-bit:
 - MOVW = 0x66 MOVW
 - .code32 in boot.S tells assembler to insert 0x66
- 80386 also added virtual memory addresses
 - Segment registers are indices into a 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 byte on the parallel lines */
    outb(DATA_PORT, c);

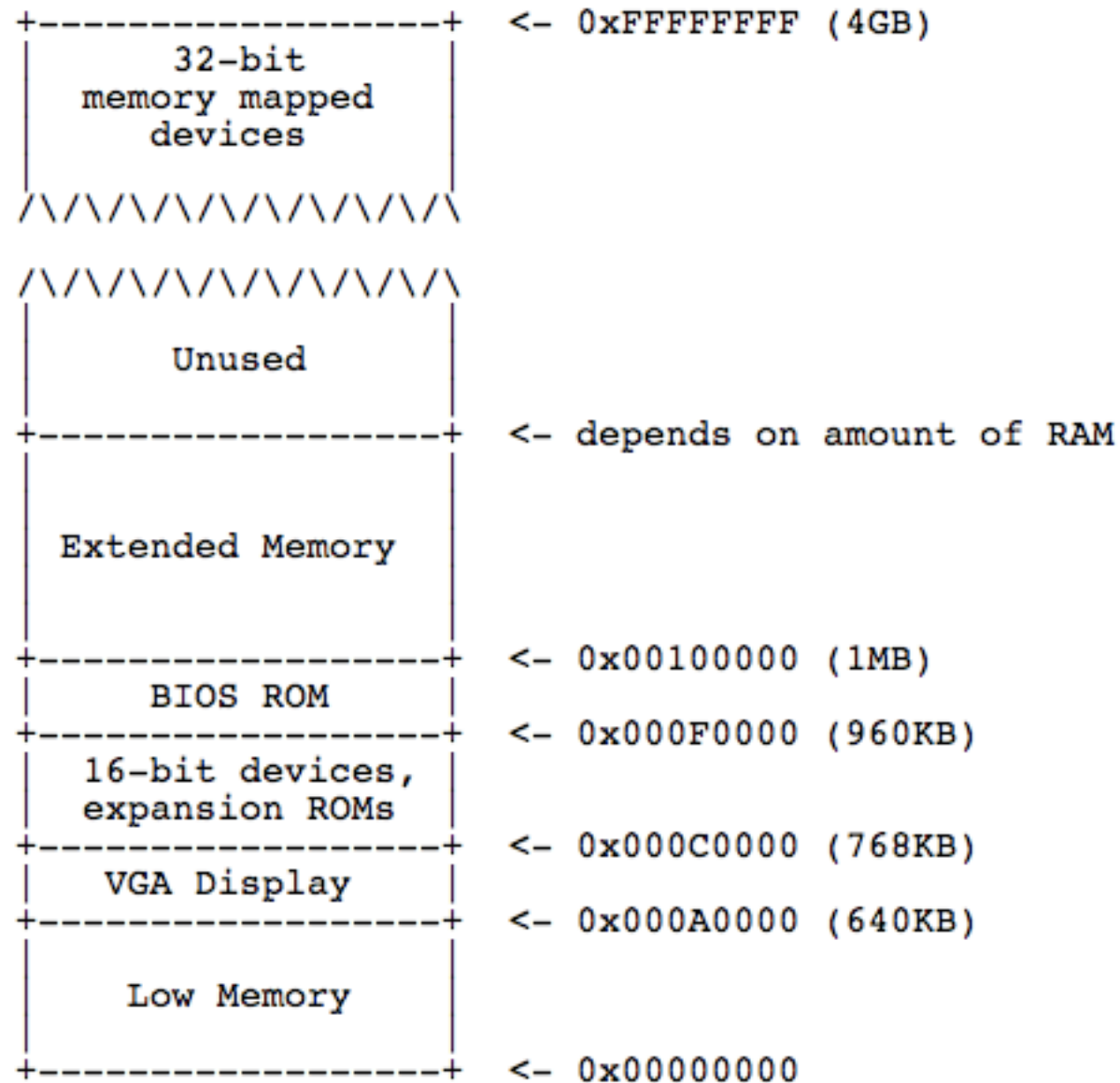
    /* tell the printer to look at the data */
    outb(CONTROL_PORT, STROBE);
    outb(CONTROL_PORT, 0);
}
```

- 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 memory
 - But does not behave like memory
 - Reads and writes have “side effects”
 - Read result can change due to external events

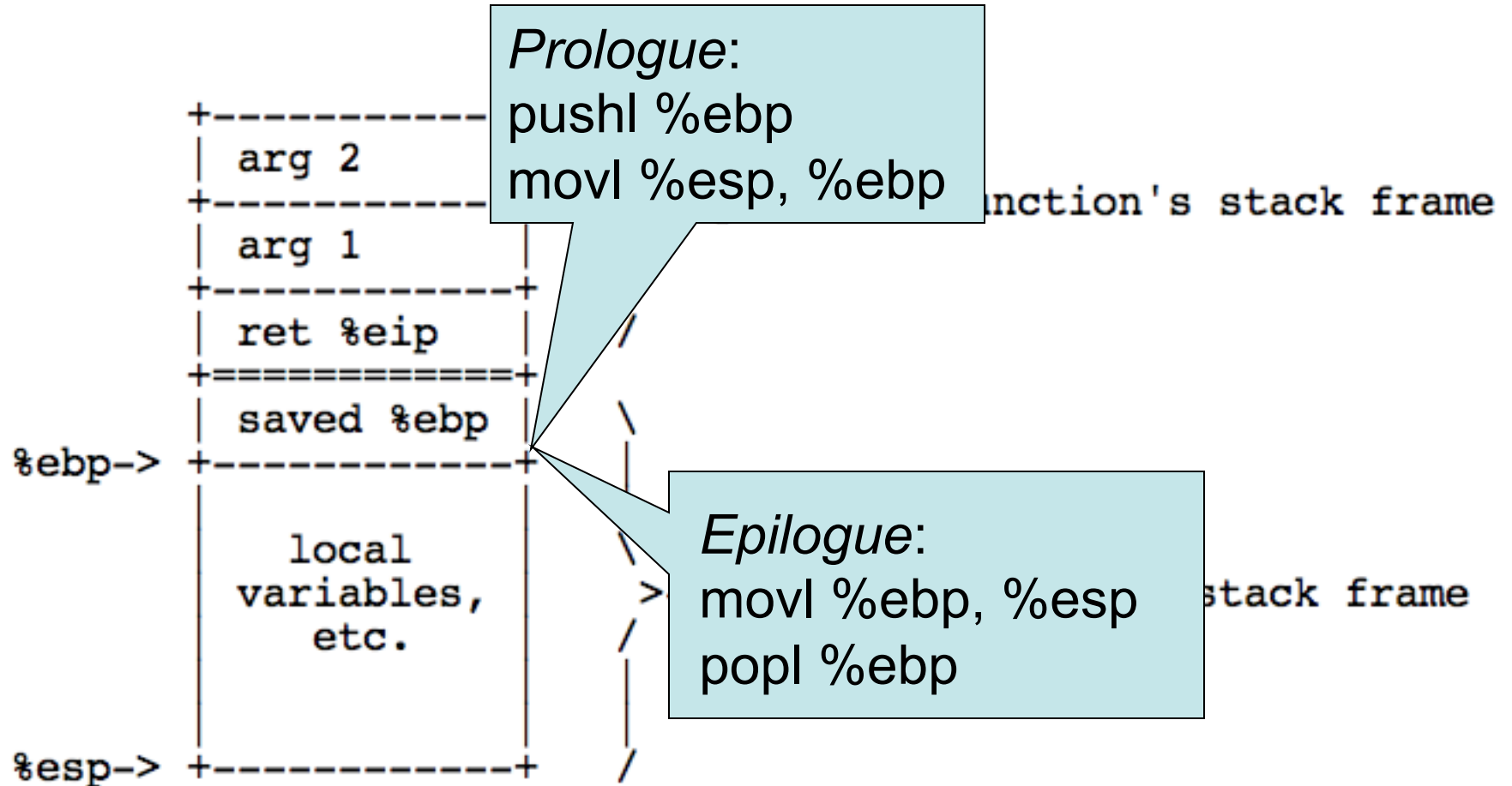
Physical memory layout



x86 instruction set

- Instructions classes:
 - Data movement: MOV, PUSH, POP, ...
 - Arithmetic: TEST, SHL, ADD, ...
 - I/O: IN, OUT, ...
 - Control: JMP, JZ, JNZ, CALL, RET
 - String: REP, MOVSB, ...
 - System: IRET, INT, ...
- Intel architecture manual Volume 2
 - Intel syntax: op dst, src
 - AT&T (gcc/gas) syntax: op src, dst

Gcc calling conventions for JOS



- Saved %ebp' s form a chain, can walk stack
- Arguments and locals at fixed offsets from EBP

gcc procedure calling convention

Caller saved

- ~~%eax~~ contains return value, %ecx, %edx may be trashed
- %ebp, ~~%ebx~~, %esi, %edi must be as before call

Callee saved

- Note that %ebp isn't strictly necessary. We compile JOS and xv6 this way for convenience of walking up the stack.

Example

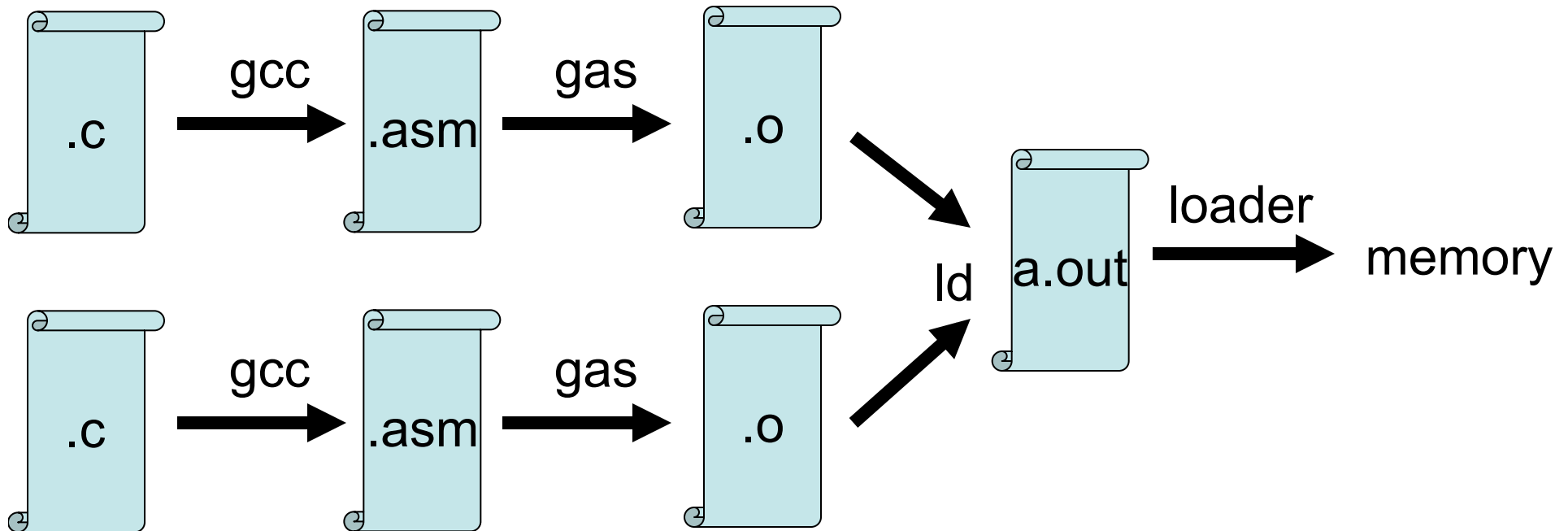
```
int main(void) { return f(8)+1; }
int f(int x) { return g(x); }
int g(int x) { return x+3; }
```

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

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

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

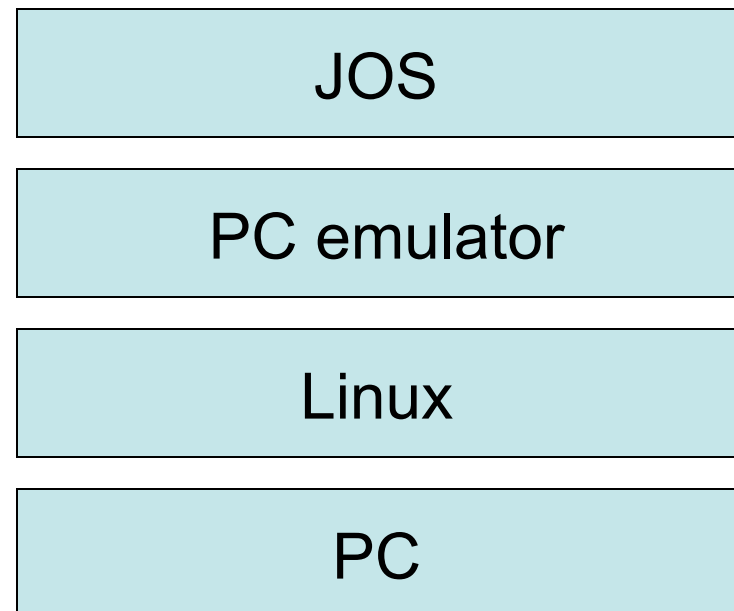
From C to running program



- Compiler, assembler, linker, and loader

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];
...

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

Emulation of CPU

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


Emulation x86 memory

```
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: hosts' s keyboard API
- Clock chip: host' s clock
- Etc.

Summary

- For lab: PC and x86
- Illustrate several big ideas:
 - Stored program computer
 - Stack
 - Memory-mapped I/O
 - Software = hardware