6.828: PC hardware and x86

Frans Kaashoek
kaashoek@mit.edu

A PC

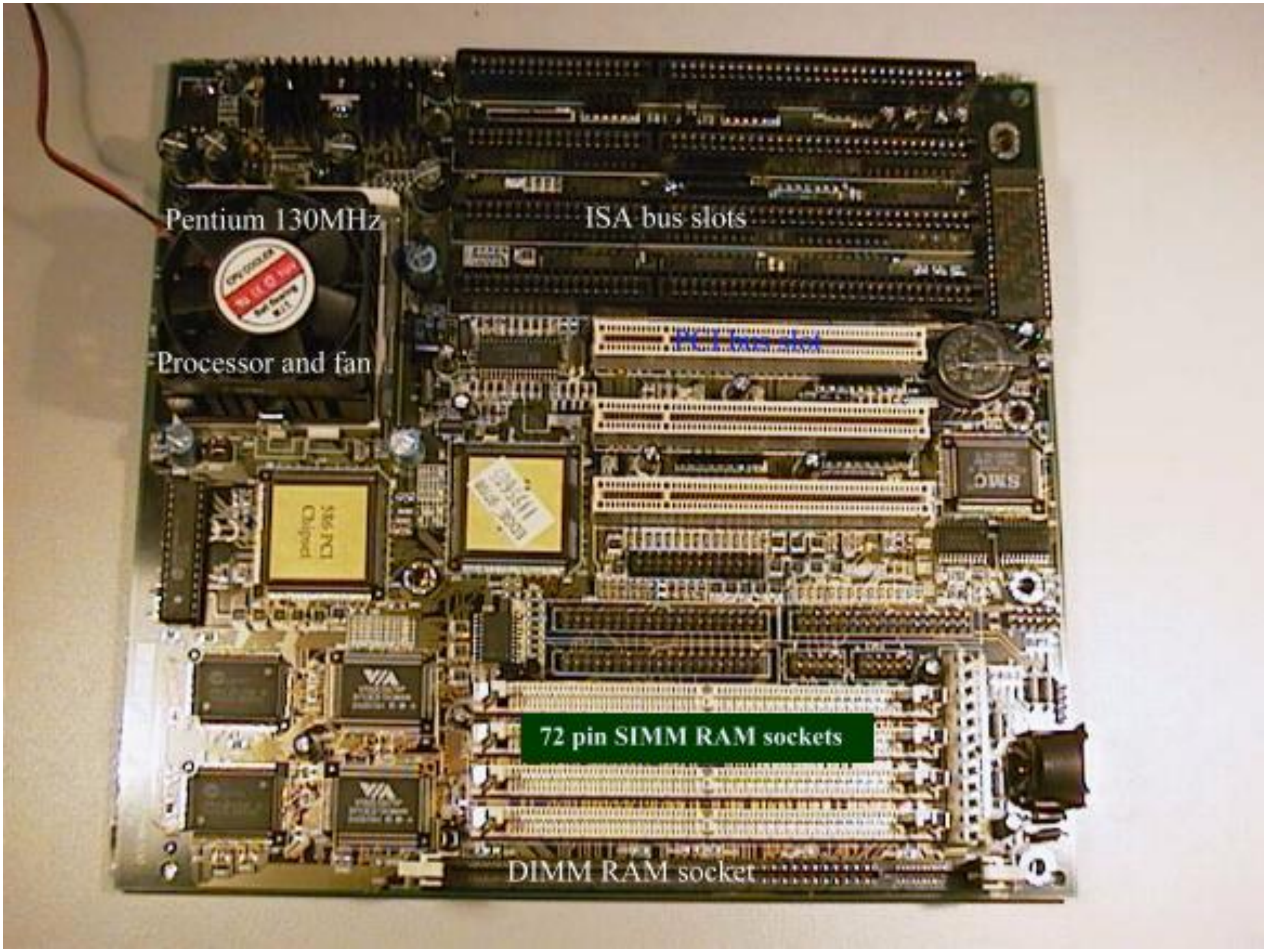
how to make it to do something useful?

Outline

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

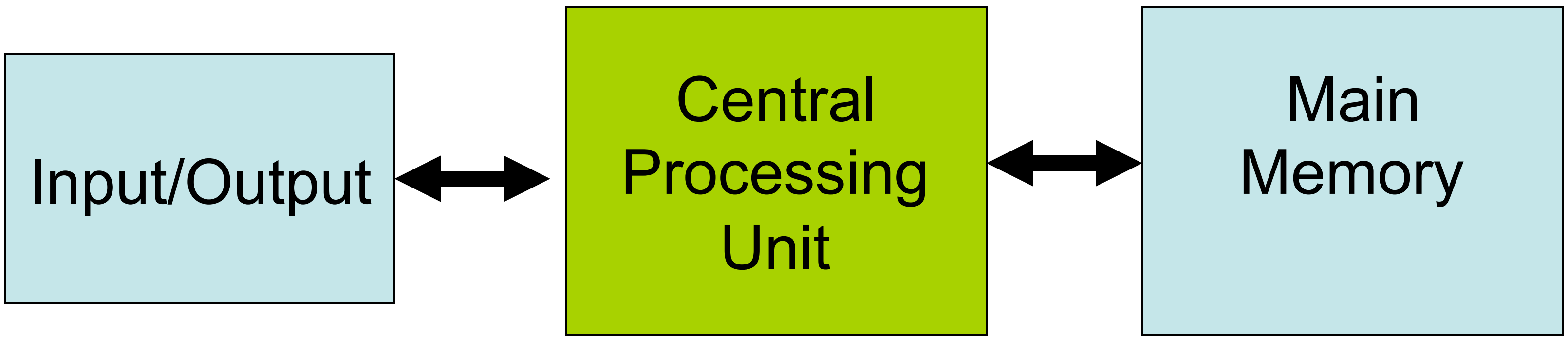
Illustrate a few big CS ideas

PC board



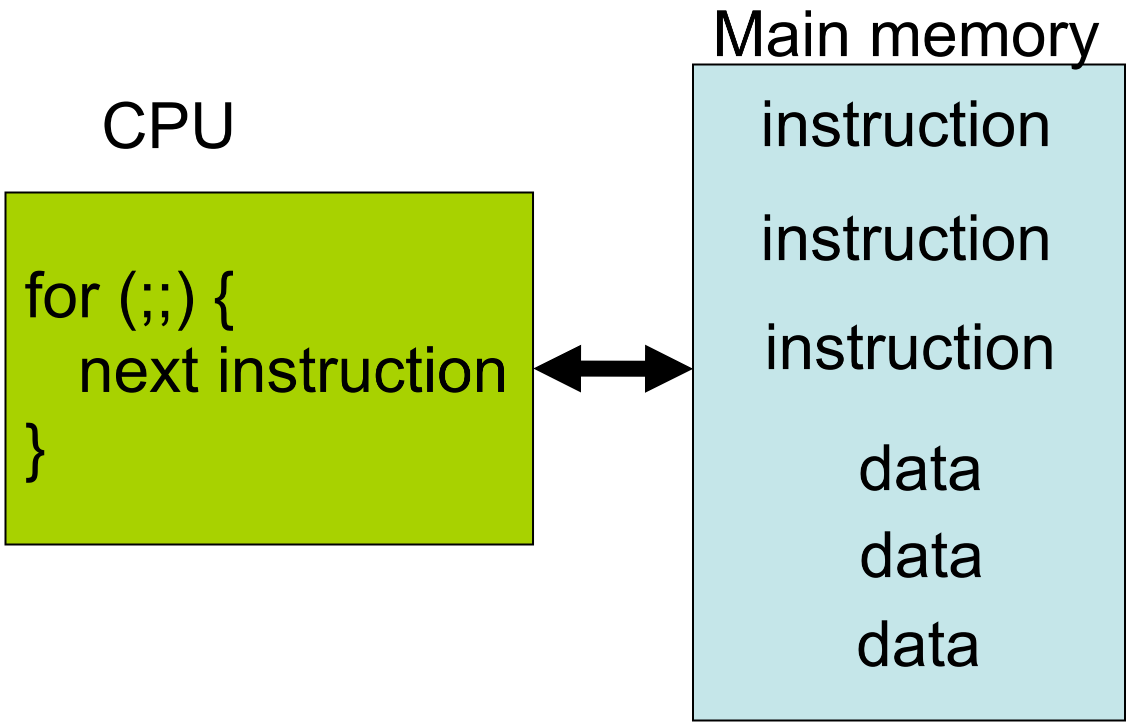
Pentium 130MHz
ISA bus slots
PCI bus slot
Processor and fan
72 pin SIMM RAM sockets
DIMM RAM socket

Abstract model



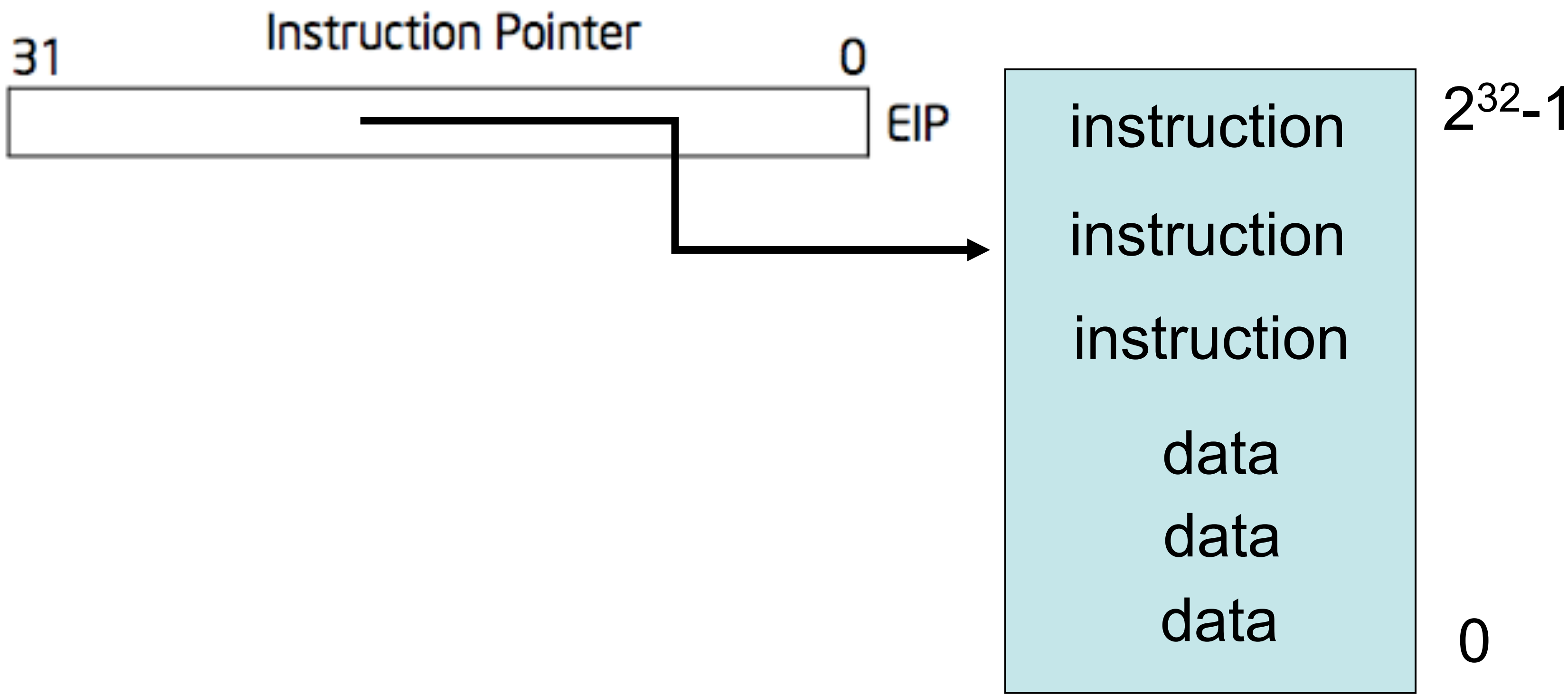
- I/O: communicating data to and from devices
- CPU: digital logic for performing computation
- Memory: *N* words of *B* bits

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

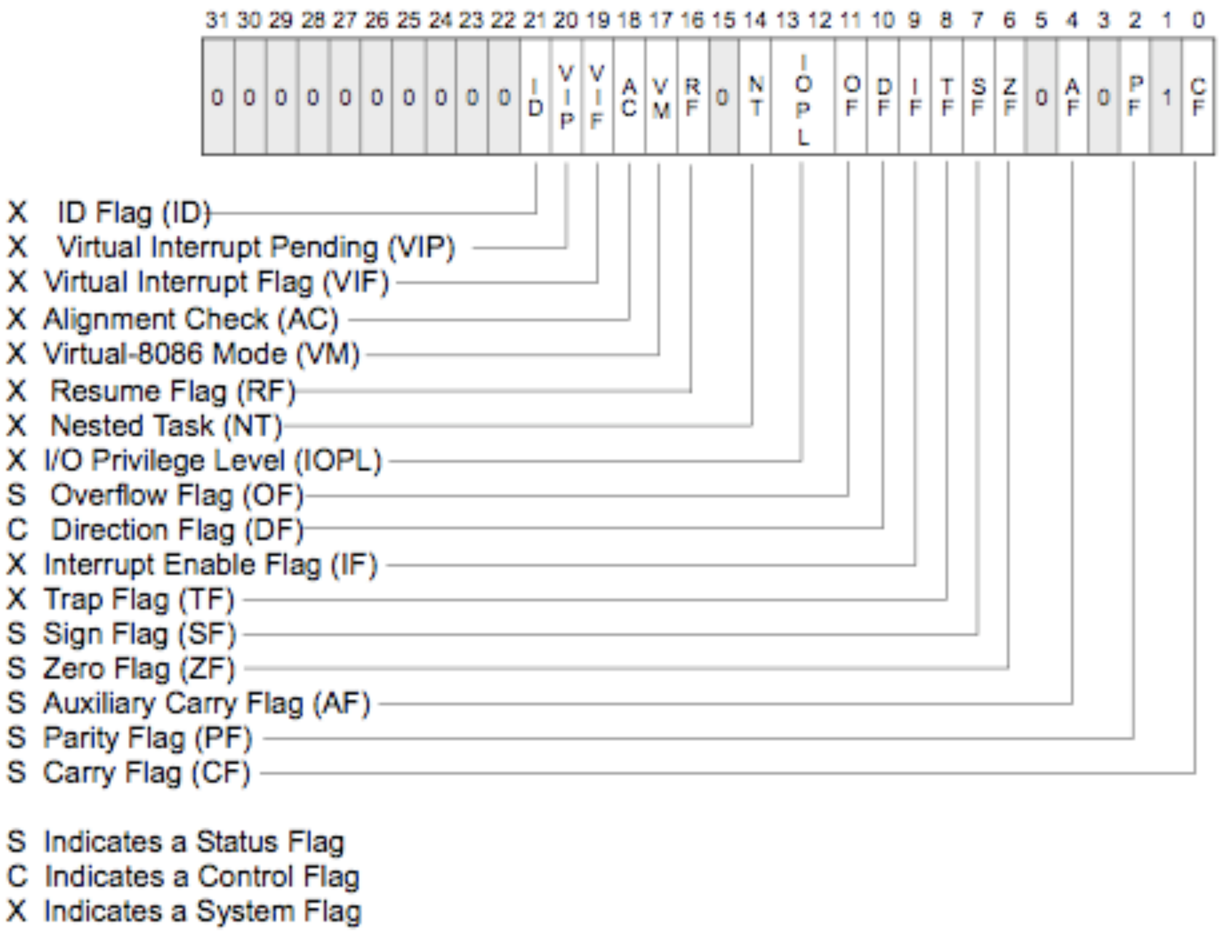
Registers for work space

General-Purpose Registers

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

movl %eax, %edx	edx = eax;	*register mode*
movl $0x123, %edx	edx = 0x123;	*immediate*
movl 0x123, %edx	edx = *(int32_t*)0x123;	*direct*
movl (%ebx), %edx	edx = *(int32_t*)ebx;	*indirect*
movl 4(%ebx), %edx	edx = *(int32_t*)(ebx+4);	*displaced*

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

Stack memory + operations

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

- 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

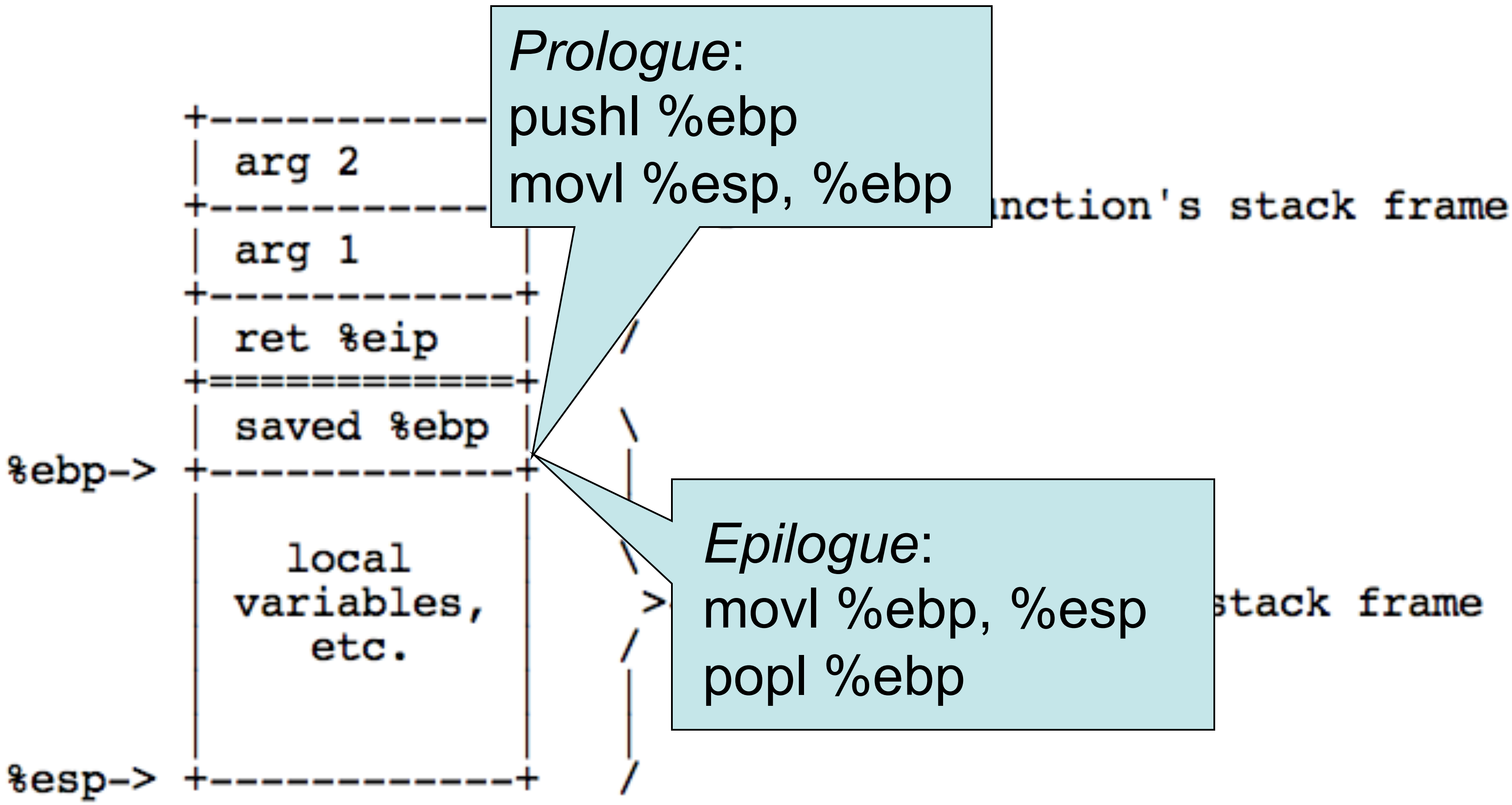
```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

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 co s

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 neces
  compile JOS and xv6 this way for convenience of walking up the stack.

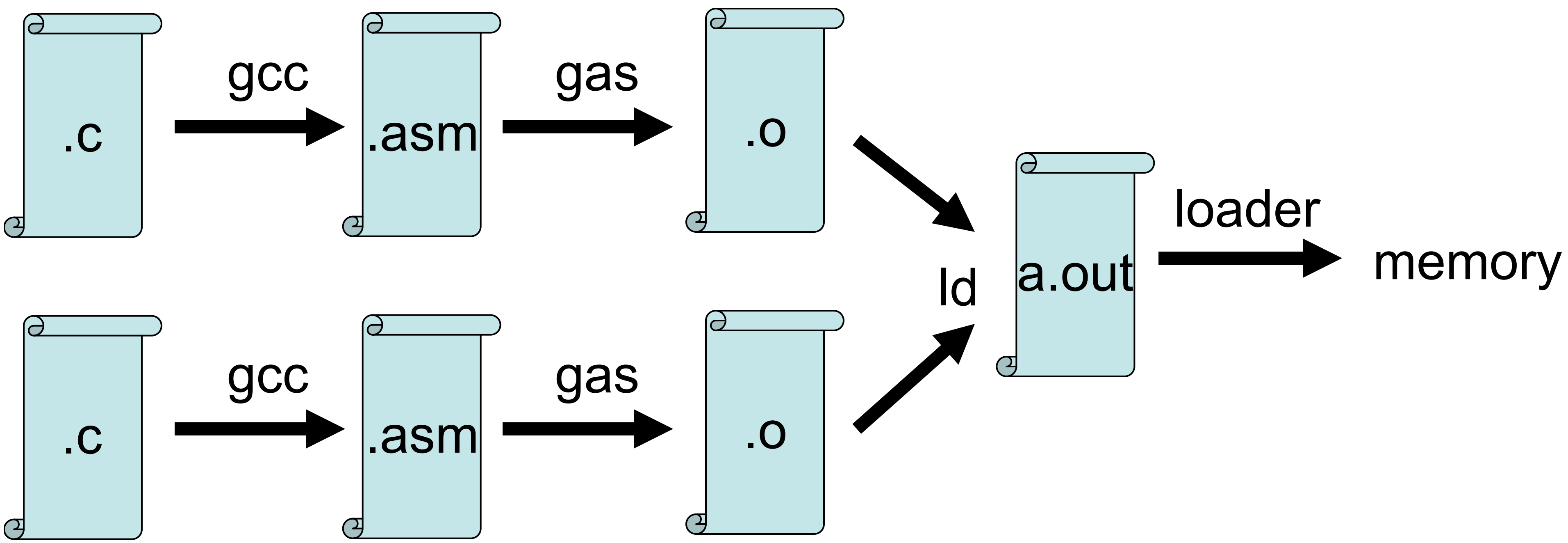
Example

```c
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
```

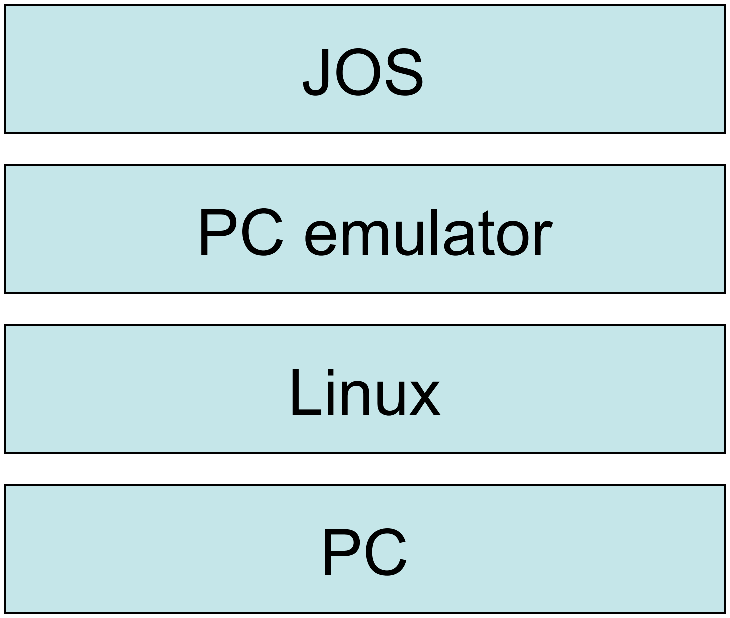
From C to running program



- 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 OPCODE_ADD:
                int src = decode_src_reg();
                int dst = decode_dst_reg();
                regs[dst] = regs[dst] + regs[src];
                break;
        case OPCODE_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