W4118: PC Hardware and x86

Junfeng Yang

A PC

How to make it do something useful?
Outline

- PC organization
- x86 instruction set
- gcc calling conventions
- PC emulation
PC board

- Pentium 130MHz
- ISA bus slots
- Processor and fan
- 72 pin SIMM RAM sockets
- DIMM RAM socket
PC organization

- One or more CPUs, memory, and device controllers connected through system bus
Abstract model

- I/O: communicating data to and from devices
- CPU: digital logic for doing computation
- Memory: N words of B bits
The stored program computer

- Memory holds both *instructions* and *data*
- CPU interprets instructions
- Instructions read/write data

```
for(;;) {
    fetch next instruction
    run next instruction
}
```
x86 implementation

- EIP incremented after each instruction
- Variable length instructions
- EIP modified by CALL, RET, JMP, conditional JMP
Registers: work space

8, 16, and 32 bit versions

Example: ADD EAX, 10
  - More: SUB, AND, etc

By convention some for special purposes

<table>
<thead>
<tr>
<th>General-Purpose Registers</th>
<th>16-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 16 15 8 7 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AH</td>
<td>AL</td>
<td>AX</td>
</tr>
<tr>
<td>BH</td>
<td>BL</td>
<td>BX</td>
</tr>
<tr>
<td>CH</td>
<td>CL</td>
<td>CX</td>
</tr>
<tr>
<td>DH</td>
<td>DL</td>
<td>DX</td>
</tr>
<tr>
<td>BP</td>
<td></td>
<td>EBP</td>
</tr>
<tr>
<td>SI</td>
<td></td>
<td>ESI</td>
</tr>
<tr>
<td>DI</td>
<td></td>
<td>EDI</td>
</tr>
<tr>
<td>SP</td>
<td></td>
<td>ESP</td>
</tr>
</tbody>
</table>

ESP: stack pointer
EBP: frame base pointer
ESI: source index
EDI: destination index
EFLAGS register

- Track current CPU status

```
TEST EAX, EBX
JNZ address
```
Memory: more work space

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

```
movl %eax, %edx   edx = eax;           \textit{register mode}
movl $0x123, %edx  edx = 0x123;        \textit{immediate}
movl 0x123, %edx   edx = *(int32_t*)0x123; \textit{direct}
movl (%ebx), %edx  edx = *(int32_t*)ebx;  \textit{indirect}
movl 4(%ebx), %edx edx = *(int32_t*)(ebx+4); \textit{displaced}
```
Stack memory + operations

<table>
<thead>
<tr>
<th>Example instruction</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl %eax</td>
<td>subl $4, %esp</td>
</tr>
<tr>
<td></td>
<td>movl %eax, (%esp)</td>
</tr>
<tr>
<td>popl %eax</td>
<td>movl (%esp), %eax</td>
</tr>
<tr>
<td></td>
<td>addl $4, %esp</td>
</tr>
<tr>
<td>call 0x12345</td>
<td>pushl %eip (*)</td>
</tr>
<tr>
<td></td>
<td>movl $0x12345, %eip (*)</td>
</tr>
<tr>
<td>ret</td>
<td>popl %eip (*)</td>
</tr>
</tbody>
</table>

- For implementing function calls
- Stack grows "down" on x86
More memory

- 8086 16-bit register and 20-bit bus addresses
  - These extra 4 bits come from segment register
    - CS: code segment, for IP
      - Instruction address: $CS \times 16 + IP$
    - SS: stack segment, for ESP and EBP
    - DS: data segment for load/store via other registers
    - ES: another data segment, destination for string ops

- Make life more complicated
  - Cannot directly use 16-bit stack address as pointer
  - For a far pointer programmer must specify segment reg
  - Pointer arithmetic and array indexing across seg bound
And more memory

- 80386: 32 bit register and addresses (1985)
- AMD k8: 64 bit (2003)
  - RAX instead of EAX
  - x86-64, x64, amd64, intel64: all same thing

- Backward compatibility
  - Boots in 16-bit mode; bootasm.S switches to 32
  - Prefix 0x66 gets 32-bit mode instructions
    - MOVW in 32-bit mode = 0x66 + MOVW in 16-bit mode
  - .code32 in bootasm.S tells assembler to insert 0x66

- 80386 also added virtual memory addresses
I/O space and instructions

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

void lpt_putchar(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 addresses
Memory-mapped I/O

- Use normal addresses for I/O
  - No special instructions
  - No 1024 limit
  - Hardware routes to device

- Works like “magic” memory
  - I/O device addressed and accessed like memory
  - However, reads and writes have “side effects”
  - Read result can change due to external events
Memory layout

- 32-bit memory mapped devices
  \( \rightarrow 0xFFFFFFFF \) (4GB)

- Unused
  \( \rightarrow \) depends on amount of RAM

- Extended Memory
  \( \rightarrow 0x00100000 \) (1MB)

- BIOS ROM
  \( \rightarrow 0x000F0000 \) (960KB)

- 16-bit devices, expansion ROMs
  \( \rightarrow 0x000C0000 \) (768KB)

- VGA Display
  \( \rightarrow 0x000A0000 \) (640KB)

- Low Memory
  \( \rightarrow 0x00000000 \)
Instruction classes

- Instruction classes
  - Data movement: MOV, PUSH, POP, ...
  - Arithmetic: TEST, SHL, ADD, AND, ...
  - I/O: IN, OUT, ...
  - Control: JMP, JZ, JNZ, CALL, RET
  - String: MOVSB, REP, ...
  - System: INT, IRET

- Instruction syntax
  - Intel manual Volume 2: op dst, src
  - AT&T (gcc/gas): op src, dst
    - op uses suffix b, w, l for 8, 16, 32-bit operands
gcc inline assembly

- Can embed assembly code in C code
  - Many examples in xv6

- Basic syntax: `asm ("assembly code")`
  e.g., `asm ("movl %%eax %%ebx")`

- Advanced syntax:
  ```c
  asm ( assembler template
       : output operands /* optional */
       : input operands   /* optional */
       : list of clobbered registers /* optional */ );
  ```
  e.g.,
  ```c
  int val;
  asm ("movl %%ebp,%0" : "=r" (val));
  ```
gcc calling conventions

- Args, ret addr, locals: fixed offsets from EBP
- Saved EBPs form a chain, can walk stack

Prologue:
- pushl %ebp
- movl %esp, %ebp

Epilogue:
- movl %ebp, %esp
- popl %ebp
main() {
    return foo(10, 20);
}
int foo(int x, int y) {
    return x+y;
}

_main:
pushl %ebp
movl %esp, %ebp
pushl $20
pushl $10
call foo
movl %ebp, %esp  //addr X
popl %ebp
ret

_foo:
pushl %ebp
movl %esp, %ebp
movl 0xc(%ebp), %eax
add 0x8(%ebp), %eax
movl %ebp, %esp
popl %ebp
ret
gcc calling conventions (cont.)

- %eax contains return value, %ecx, %edx may be trashed
  - 64 bit return value: %eax + %edx

- %ebp, %ebx, %esi, %edi must be as before call

- Caller saved: %eax, %ecx, %edx

- Callee saved: %ebp, %ebx, %esi, %edi
From C to running program

- Compiler, assembler, linker, and loader
Development using PC emulator

- QEMU pc emulator
  - Does what a real PC does
  - Except implemented in s/w!

- Run like a normal program on “host” OS
Emulator of Registers

```c
int32_t regs[8];
#define REG_EAX 1;
#define REG_EBX 2;
#define REG(ECX) 3;
...
int32_t eip;
int16_t segregs[4];
...
```
Emulator of CPU logic

```c
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 of 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: use file of the host
- VGA display: draw in a host window
- Keyboard: host’s keyboard API
- Clock chip: host’s clock
- Etc.
Summary

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

**Next lecture**

- Processes and address spaces
gcc inline assembly example

int a=10, b;
asm ("movl %1, %eax;
    movl %eax, %0;"
    :="r"(b)  /* output operands */
    :"r"(a)  /* input operands */
    :"%eax"  /* clobbered registers */ );

- Equivalent to b = a
- Operand number: %0, %1, ... %n-1, n = the total number of operand
  - b is output, referred to by %0
  - a is input, referred to by %1

- "r" store in registers
- "=" write only
Example

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

```assembly
_main:  prologue
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx  
    pushl %ebp
    call _f  
    movl %ebp, %esp
    popl %ebp
    ret

_f:  prologue
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    movl %ebp, %esp
    popl %ebp
    ret

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