CS5600 -
PC H/W & Assembly
Overview

• Hardware basics
• PC Bootup Sequence
• x86 basics
• Intro to OS
Hardware Basics

- PC compatible, “Wintel"
  - alternatives: Amiga, PowerPC, DEC Alpha, SPARC, etc.
- 1981 IBM PC (compete with Apple)
- 1982 Compaq IBM-compatible PC
- 1985 IBM clones everywhere!
- 1986 Compaq 80386-based PC
- 1990s Wintel
  - x86, Pentium I, II, III …
  - x86_64 AMD … tomorrow?
Motherboard

CPU
I/O
Memory
BIOS
South-Bridge
• I/O between CPU, devices and MM
North-Bridge
• Coordinates access to MM
Storage
Connectors
• (S)ATA
Conceptually

All devices compete for access to memory

L1, L2, L3 Cache

CPU(s)

North/South Bridge

Graphics

Memory

Graphics Memory

I/O

I/O

I/O

...
Simplified CPU Layout

- Control Unit
- ALU
- ALU
- FPU
- Registers
- Cache L1, L2, L3
- Instruction Decode
- Instruction Fetch
- System Bus
Registers

- Storage built into the CPU
- Can hold valued or pointer
- Instructions operate directly on registers
- Load from memory
- Load to memory
Registers

• Some registers are special
  • point to the current instruction in memory
  • point to top of the stack
  • configure low-level CPU features
  • …
Memory Hierarchy

- More Costly
  - Registers
  - Level 1 Cache
  - Level 2 Cache
  - Main Memory
  - Fixed Rigid Disk
  - Optical Disk (Jukeboxes)
  - Magnetic Tape (Robotic Libraries)

- Less Costly
  - System
  - Online
  - Near Line
  - Offline

Access Times:
- 1ns → 2ns
- 3ns → 10ns
- 25ns → 50ns
- 30ns → 90ns
- 5ms → 20ms
- 100ms → 5s *
- 10s → 3m *

* If volume is mounted.
x86_32 Registers

General-purpose Registers

- EAX
- EBX
- ECX
- EDX
- ESI
- EDI

Stack-related Registers

- ESP (stack pointer)
- EBP (base pointer)
x86 Registers

- EIP
  - Points to currently executing instruction

- EFLAGS
  - Think of it as scratch register, e.g., results after comparison, carry after addition.
  - Sometimes referred to as the *machine status word register*
### x86 instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mov</strong></td>
<td>Move data src -&gt; dst</td>
<td>mov eax, 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov edx, [0xF0FF]</td>
</tr>
<tr>
<td><strong>add/sub</strong></td>
<td>Add/subtract vals in reg.</td>
<td>add eax, eab</td>
</tr>
<tr>
<td><strong>inc/dec</strong></td>
<td>Increment/decrement value in reg.</td>
<td>inc eax</td>
</tr>
<tr>
<td><strong>call</strong></td>
<td>Push EIP onto stack &amp; jump to func</td>
<td>call 0x80FEAC</td>
</tr>
<tr>
<td><strong>ret</strong></td>
<td>Pop the stack into EIP</td>
<td>ret</td>
</tr>
<tr>
<td><strong>push/pop</strong></td>
<td>Push/pop onto stack</td>
<td>push eax</td>
</tr>
<tr>
<td><strong>int</strong></td>
<td>Execute interrupt handler</td>
<td>int 0x70</td>
</tr>
<tr>
<td><strong>jmp</strong></td>
<td>Load value into EIP</td>
<td>jmp 0x80FEAC</td>
</tr>
<tr>
<td><strong>cmp</strong></td>
<td>Compare 2 regs, put result in flags register</td>
<td>cmp ebx, edx</td>
</tr>
<tr>
<td><strong>jz/jnz/jXXX</strong></td>
<td>Load value in EIP if zero or non-zero in flags register</td>
<td>jnz 0x80FEAC</td>
</tr>
</tbody>
</table>
Example x86 assembly

for (i = 0; i < a; i++)
    sum += i;

xorl %edx,%edx  # i = 0 (more compact than movl)
cmpl %ecx,%edx  # test (i - a)
jge .L4         # >= 0 ? jump to end
movl sum,%eax   # cache value of sum in register

.L6:
    addl %edx,%eax  # sum += i
    incl %edx      # i++
cmpl %ecx,%edx  # test (i - a)
jl .L6           # < 0 ? go to top of loop
movl %eax,sum   # store value of sum back in memory

.L4
Memory Layout

- **Memory mapped devices**
- **Free Memory**
- **BIOS Code**
- **Interrupt Vector**

- **Top 0xFFFF**
- **0xF000**
- **0x0DFF**
- **0x00FF**
- **0x0000**

- **High BIOS (2 MB)**
- ~1 GB for PCI space, APIC space, DMI interface, etc.
- Accessible RAM Memory (nearly 3 GB, not to scale)
- System BIOS
- Extended System BIOS
- Expansion Area (maps ROMs for old peripheral cards)
- Legacy Video Card Memory Access
- Accessible RAM Memory (640KB is enough for anyone - old DOS area)
CPU and Device Communication

• CPU and devices execute concurrently

• Communication happens

  1. I/O ports
     • Specific addresses on I/O Bus
  
  2. Memory mapping
     • RAM region shared by device and CPU
  
  3. Direct Memory Map
     • Device writes directly to share region in RAM
  
  4. Interrupts
     • Signal from device to CPU. OS has to switch to handler code
Examples

Shared Memory

Interrupt

I/O Ports
Device, CPU communication

- I/O Ports
  - virtual memory shared between them
  - Synchronous + CPU has to copy data over
  - SLOW!
- Memory Mapped
  - RAM shared between them, CPU involved in all memory transactions
- Direct Memory Access (DMA)
  - device reads/writes to memory without involving the CPU
Interrupts

- Interrupt Vector
  - Maps interrupts to handler’s address
  - Interrupt causes context switch

<table>
<thead>
<tr>
<th>Number</th>
<th>Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0xAAA1</td>
</tr>
<tr>
<td>0x02</td>
<td>0xBBB1</td>
</tr>
</tbody>
</table>
PC Bootup Process

1. **BIOS Initialization**
2. **Master Boot Record**
3. **Boot Loader**
4. **Early Kernel Initialization**
   - **Switch to Protected Mode**
5. **Full Kernel Initialization**
   - **First User-Mode Process**
6. **CPU in Protected Mode**
7. **BIOS Services**
8. **Kernel Services**
9. **Hardware**

**Time Flow**
Power On

- Start the BIOS (Basic Input/Output System)
  - code from BIOS gets copied to RAM
  - load EIP register with starting address
- Load setting from CMOS
- Initialize devices
  - CPU, MEM, Keyboard, Video
  - Install Interrupt Vector Table
- Run POST (Power On Self Test)
- Initiate the bootstrap sequence (configurable, HD, CD, net)
MBR

N-sector disk drive. Each sector has 512 bytes.

Master Boot Record (512 bytes)

- Code (440 bytes)
- Disk Signature (4 bytes)
- Nulls (2 bytes)
- Partition Table (four 16-byte entries, 64 bytes total)
- MBR Signature (2 bytes)
MBR

- Special 512 byte file in sector 1 address 0
- Too small for a full OS
  - points to another section of your drive
  - starts chain loading
The Kernel

• The program that always runs on your machine

• Started by the boot loader

• Features
  • Device management
  • loading and executing your programs
  • System calls and APIs
  • Protection
  • Fault tolerance
  • Security
Kernel Architectures

- Monolithic
  - one big code base, one big binary
  - Code Runs in privileged Kernel-space

- Microkernels
  - Only core components in the kernel
  - Rest of kernel components run in user space

- Hybrid kernels
  - Most components run in the kernel
  - Some loaded dynamically
Monolithic

Kernel Space

- MMM
- Program Loader
- Device Drivers
- File Systems
- Security Policies
- Scheduler

User Space

- System APIs
- User Program
Microkernel

Kernel Space
- MMM
- Program Loader
- Scheduler
- Security Policies
- Interprocess Communication

User Space
- User Program
- File Systems
- Device Drivers
Hybrid

Kernel Space
- MMM
- Program Loader
- Security Policies
- Scheduler
- System APIs
- File Systems
- 3rd-Party Code
- Device Drivers

Kernel Code

User Space
- User Program
## Examples

<table>
<thead>
<tr>
<th>Microkernels</th>
<th>Hybrid</th>
<th>Monolithic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mach</td>
<td>Windows</td>
<td>DOS</td>
</tr>
<tr>
<td>L4</td>
<td>iOS</td>
<td>SunOS</td>
</tr>
<tr>
<td>GNU Hurd</td>
<td>OS/2</td>
<td>Linux</td>
</tr>
<tr>
<td>QNX</td>
<td>BeOS</td>
<td>OpenVMS</td>
</tr>
</tbody>
</table>