CS5600 - PC H/W & Assembly
Overview

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

• PC compatible, “Wintel"
  • alternatives: Amiga, PowePC, 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, Pemtium 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

L1, L2, L3 Cache
CPU(s)
Graphics
Memory
North/South Bridge
Graphics Memory
I/O I/O ...
I/O

All devices compete for access to memory
Simplified CPU Layout

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

- Storage build 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
  • …
Toy CPU

- ALU
- ACC
- Control Unit
- Status Register
- SR
- MDR
- PC
- MAR
- CIR
- BUS
- RAM
- Addr
Toy CPU

- ALU
- Control Unit
- MAR
- Arithmetic and Logic Unit
- PC
- ACC
- SR
- MDR
- CIR
- BUS
- RAM
- Addr
Toy CPU

PC

MAR

ALU

Control Unit

CIR

General Purpose Registers r1 .. rn

SR

MDR

RAM Addr

BUS
It’s all bits!
Use Assembly instruction names and numbers
It all starts with the PC
Copy the contents of PC into MAR
To fetch the contents of the address, place it on the bus
Control unit notifies MDR to be on the look out
MDR picks the result of fetch from the bus and copies to CIR
Toy CPU

CU decodes to figure out which instruction we have
It's a load of number 1, execute it and load 1 in ACC
Toy CPU

Check if there is an interrupt
Toy CPU

PC moves to the next instruction (next address), repeat!
Repeating the process for the next instruction with less detail
Repeating the process for the next instruction with less detail
Repeating the process for the next instruction with less detail.
Toy CPU

Repeating the process for the next instruction with less detail
Toy CPU

Repeating the process for the next instruction with less detail
Repeating the process for the next instruction with less detail
Toy CPU

Repeating the process for the next instruction with less detail
Repeating the process for the next instruction with less detail
Repeating the process for the next instruction with less detail
Toy CPU

Repeating the process for the next instruction with less detail
Toy CPU

Repeating the process for the next instruction with less detail
Toy CPU

Repeating the process for the next instruction with less detail
Repeating the process for the next instruction with less detail.
Toy CPU

Repeating the process for the next instruction with less detail
Actual CPU

- Architectures Vary (Intel, DEC Alpha, AMIGA etc.)
  - different layouts, numbers of registers etc.
  - registers for stack points (sp), frame base pointer (sb), flag registers (EFLAGS)
- Instruction Set Architecture varies
  - CISC vs RISC (X86, Sparc)
  - branching and jumping instructions manipulate the PC
- The course will use X86, 32bit
Memory Hierarchy
x86_32 Registers

- General-purpose Registers
  - EAX
  - EBX
  - ECX
  - EDX
  - ESI
  - EDI
  - ESP (stack pointer)
  - EBP (base pointer)

- AX
- BX
- CX
- DX
- AH
- BH
- CH
- DH
- AL
- BL
- CL
- DL
x86 Registers

- EIP (PC)
  - 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>
</table>
| **mov**     | Move data src -> dst | mov eax,7  
              mov edx,[0xF0FF] |
| **add/sub** | Add/subtract vals in reg. | add eax,eab |
| **inc/dec** | Increment/decrement value in reg. | inc eax |
| **call**    | Push EIP onto stack & jump to func | call 0x80FEAC |
| **ret**     | Pop the stack into EIP | ret |
| **push/pop**| Push/pop onto stack | push eax |
| **int**     | Execute interrupt handler | int 0x70 |
| **jmp**     | Load value into EIP | jmp 0x80FEAC |
| **cmp**     | Compare 2 regs, put result in flags register | cmp ebx,edx |
| **jz/jnz/jXXX** | Load value in EIP if zero or non-zero in flags register | jnz 0x80FEAC |
Assembly Language

- Contents of the file is text. Much like your C programs. It’s a programming language (not high level)
- assembler takes text file with assembly instructions and creates a file with 0s and 1s see (as, as86, nasm)
- syntax is simple
  - label: instruction ; comment
  - sections and directives; .data .bss .text
Assembly Language

- Contents of the file is text. Much like your C programs. It’s a programming language (not high level)

- assembler takes text file with assembly instructions and creates a file with 0s and 1s see (as, as86, nasm)

- syntax is simple
  - label: instruction ; comment
  - sections and directives; .data .bss .text

Initialize data, constants
Assembly Language

• Contents of the file is text. Much like your C programs. It’s a programming language (not high level)

• assembler takes text file with assembly instructions and creates a file with 0s and 1s see (as, as86, nasm)

• syntax is simple
  • label: instruction ; comment
  • sections and directives; .data .bss .text

Un-initialized data, local vars
Assembly Language

• Contents of the file is text. Much like your C programs. It’s a programming language (not high level)

• assembler takes text file with assembly instructions and creates a file with 0s and 1s see (as, as86, nasm)

• syntax is simple
  
  • label: instruction ; comment

  • sections and directives; .data .bss .text
Example x86 assembly

```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
```
Main Memory

• CPU is fast and dumb

• Manipulations involve memory

• Memory is used by more components!
  • Devices
  • Kernel
  • more than one program

• Shared resource!
Memory Layout

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

0x0000 - 0x0DFF
- High BIOS (2 MB)
- ~1 GB for PCI space, APIC space, DMI interface, etc.

0x0DFF - 0xFFFF
- Accessible RAM Memory (nearly 3GB, 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 Acces (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>
How does it all start?
Big Bang!
PC Bootup Process
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

System APIs

User Space
- 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

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>
Monolithic Vs Micro Vs...

- Linux Vs Tanenbaum