PC Architecture from inside the CPU

Russ Cox

2002 USACO Training Camp
Just run instructions, one at a time.

```c
for(;;){
    run next instruction
}
```

What should the instructions do?
Add some data locations to work in.
Add some data locations to work in.

**AX**, **BX**, **CX**, and **DX** – 32-bit variables
Add some data locations to work in.

\textbf{AX, BX, CX, and DX} – 32-bit variables

Very fast but not very many.
Memory

Send out address (one bit per wire).

After a while, data comes back from memory.

or

Data written to address in memory.
Add registers to hold pointers.
Memory Registers

SI  source index
DI  destination index
SP  stack pointer
BP  base pointer
Memory Registers

SI  source index
DI  destination index
SP  stack pointer
BP  base pointer

Registers are 16 bits, memory addresses are 20.
Use *two* variables to index memory.

\[
\text{addr}(\text{seg:off}) = \text{segment} \ll 4 + \text{off}
\]

\[
1:ABCD = 1ABC:D = 1A0C:B0D = 1ABCD
\]
Where do instructions come from?

Memory, of course.
**Instruction pointer**

CS:IP points at instructions in memory.

Normally increment after each instruction.

Jump instructions change instruction pointer.
Flags

Want conditional jumps:
- If condition, goto there.
Flags

Want conditional jumps:
- If condition, goto there.

Flags register holds result of various checks.

JG    jump if greater
JZ    jump if zero
JNZ   jump if non-zero
...

<table>
<thead>
<tr>
<th>AX</th>
<th>SI</th>
</tr>
</thead>
<tbody>
<tr>
<td>BX</td>
<td>DI</td>
</tr>
<tr>
<td>CX</td>
<td>SP</td>
</tr>
<tr>
<td>DX</td>
<td>BP</td>
</tr>
<tr>
<td>CS</td>
<td>FS</td>
</tr>
<tr>
<td>DS</td>
<td>GS</td>
</tr>
<tr>
<td>ES</td>
<td>SS</td>
</tr>
<tr>
<td>FLAGS</td>
<td>IP</td>
</tr>
</tbody>
</table>
I/O Ports

Want to interface with devices.

<table>
<thead>
<tr>
<th>AX</th>
<th>SI</th>
</tr>
</thead>
<tbody>
<tr>
<td>BX</td>
<td>DI</td>
</tr>
<tr>
<td>CX</td>
<td>SP</td>
</tr>
<tr>
<td>DX</td>
<td>BP</td>
</tr>
<tr>
<td>CS</td>
<td>FS</td>
</tr>
<tr>
<td>DS</td>
<td>GS</td>
</tr>
<tr>
<td>ES</td>
<td>SS</td>
</tr>
<tr>
<td>FLAGS</td>
<td>IP</td>
</tr>
</tbody>
</table>
I/O Ports

Want to interface with devices.

Generic interface: same as memory, but signal I/O on line.
I/O Ports: Parallel Port

Three I/O ports:

<table>
<thead>
<tr>
<th>Port</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>0x378</td>
<td>CPU to printer: data</td>
</tr>
<tr>
<td>Status</td>
<td>0x379</td>
<td>printer to CPU: status bits</td>
</tr>
<tr>
<td>Ctl</td>
<td>0x380</td>
<td>CPU to printer: control bits</td>
</tr>
</tbody>
</table>

sendbyte(int c)
{
    while(!(ioread(Status)&NotBusy))
    ;
    iowrite(Data, c);
    iowrite(Ctl, Strobe);
    iowrite(Ctl, 0);
}
Memory-Mapped I/O (MMIO)

Only 1024 I/O ports.

No reason devices can’t respond to otherwise unused memory addresses.

No need for special I/O instructions.
sendbyte(int c) {
    while(!(ioread(Status)&NotBusy));
    iowrite(Data, c);
    iowrite(Ctl, Strobe);
    iowrite(Ctl, 0);
}

sendbyte(int c) {
    while(!(*status&NotBusy));
    *data = c;
    *ctl = Strobe;
    *ctl = 0;
}

Still waiting on slow devices!
Interrupts

```c
for(;;){
    run next instruction
}
```
for(;;){
    run next instruction
    if(interrupts enabled && interrupt line){
        push current registers onto stack
        push next inst ptr onto stack
        disable interrupts
        set next inst ptr to IDTR[type]
    }
}

Interrupt handler can send next byte to printer.

We can do other work instead of waiting to send the next byte.
FLAGS controls interrupt enable/disable.
for(;;){
    run next instruction
    if(interrupts enabled && interrupt line){
        push current registers onto stack
        push next inst ptr onto stack
        disable interrupts
        set next inst ptr to IDTR[type]
    }
}

Interrupt handler returns with IRET:
- pop next instruction pointer from stack
- pop register set off stack
Direct Memory Access (DMA)

Allow devices to read and write main memory too.

CPU uses (MM)IO to point device at a buffer.

Device interrupts CPU when done with entire buffer.

Used by audio cards, ethernet cards.
Protected Mode

Memory addresses have been direct mapped to memory.

It's convenient to allow arbitrary mappings.
- Most systems leave 0 (NULL) unmapped.
- Write protect in-memory copy of program binary.
- Can swap unused virtual pages out to disk.
Protected Mode

Page table translates from virtual address to physical address.

Page Directory: 1024 32-bit entries

Page Table: 1024 32-bit entries

Page: 4096 bytes

10 bits | 10 bits | 12 bits
Segment registers facilitate use of different maps.
Protected Mode

GDTR contains (physical) pointer to GDT.

TLB (translation lookaside buffer) is a page table cache.
  - Avoids walking page tables for every memory access.
Privilege Levels

Want operating system to be protected from user processes.

Two bits of code segment selector specify privilege level.

- Ring 0: operating system kernel
- Ring 1: device drivers
- Ring 2: more device drivers
- Ring 3: user programs

No one uses ring 1 or ring 2.

Ring 0 can fiddle with:
- interrupt handler table
- global descriptor table register
Want to simulate enough of a PC to run an OS in a window.
- Host OS: operating system running on bare hardware and running VMware.
- Guest OS: operating system running on virtual hardware provided by VMware.

Could use entire-machine simulator (Mac’s VirtualPC).
- Very portable: can run on non-Intel systems.
- Very slow: executing instructions by simulating CPU.

Could let the CPU execute the instructions directly?
- Non-portable: requires Intel CPU to run simulated PC.
- Very fast: CPU can run at near-full speed.
- How do we deal with privilege levels, I/O, memory paging, etc.?
VMware CPU

Run guest OS in ring 3 (user-level).

Privileged (ring 0 only) operations (e.g., fiddling with page tables) cause interrupts.

Interrupt handler runs VMware code in ring 0.

VMware simulates privileged operation, returns to guest OS.
VMware I/O

Arrange that guest OS I/O traps into VMware
- I/O port operations are privileged.
- Leave MMIO pages unreadable/unwritable.

VMware simulates I/O operation, returns to guest OS.

DMA is much faster than (MM)I/O.
- Reduces number of traps into VMware.
  (trap per 4096 bytes instead of trap per byte)
VMware Paging

Single-OS CPU page table mapping:

VMware page table mapping:

In-CPU VMware mapping:
VMware Paging

Single-OS CPU page table mapping:

Virtual ➔ Physical

VMware page table mapping:

Guest Virtual ➔ Guest Physical ➔ Host Physical

In-CPU VMware mapping:

Guest Virtual ➔ Host Physical

Maintain in-CPU mapping as composition of guest OS map and VMware map.

Guest OS only sees guest OS map.

- Whenever guest updates its map, need to update in-CPU map.

Make guest page tables read-only.

- Writes trap into VMware, which can update both the guest and in-CPU maps.
Non-Virtualizable Instructions

Not all privileged instructions cause interrupts in ring 3.

Some just behave differently.
- E.g., changing the “‘interrupt enable’” bit in the \texttt{FLAGS} register is ignored in user mode.

  \begin{verbatim}
  FLAGS ^= InterruptEnable;
  x = FLAGS;
  FLAGS ^= InterruptEnable;
  if(x != FLAGS)
      printf("running in ring 0 (kernel mode)\n");
  else
      printf("running in ring 3 (user mode)\n");
  \end{verbatim}

  - Called “‘non-virtualizable instructions.’”

Guest OS can see that it’s not really running in ring 0.
- This is no good.
Non-Virtualizable Instructions

Need to make sure an instruction is virtualizable before running it.
- Kernel cannot run at full speed anymore.

Non-virtualizable instructions are rare and often clustered.

Check kernel code at page granularity.
- Unchecked pages are not readable, not writable.
- Checked pages are readable, not writable.
- Writes to checked pages are simulated and the page rechecked.
VMware Summary

Provide virtual CPU.
- Use page table tricks to allow CPU to run guest OS code directly.

Provide virtual devices by simulating I/O:
- Standard parallel, serial, USB, disk
- AMD 79C970 ethernet card
- Buslogic BT-958 SCSI adapter
- SoundBlaster 16 sound card
- Home-grown VGA card (much simpler)

Virtual devices are independent of host hardware.
- Run operating systems with old drivers on new machines!