Virtualization : Machine Architectures

CMPE283
Week 2

Agenda

- Machine Architectures
- OS / Hardware Integration
- Device Support

But First...

- A review...

- What is virtualization?
- Why is it important?
- Who is it important to?

Basic Definitions

- Virtualization is
  - A hardware and/or software technology
  - ...that provides for isolation of architectural layers in a computer system
  - ...wherein that isolation is performed in an efficient manner
  - …and that isolation is assumed to be inviolate

The x86 Architecture

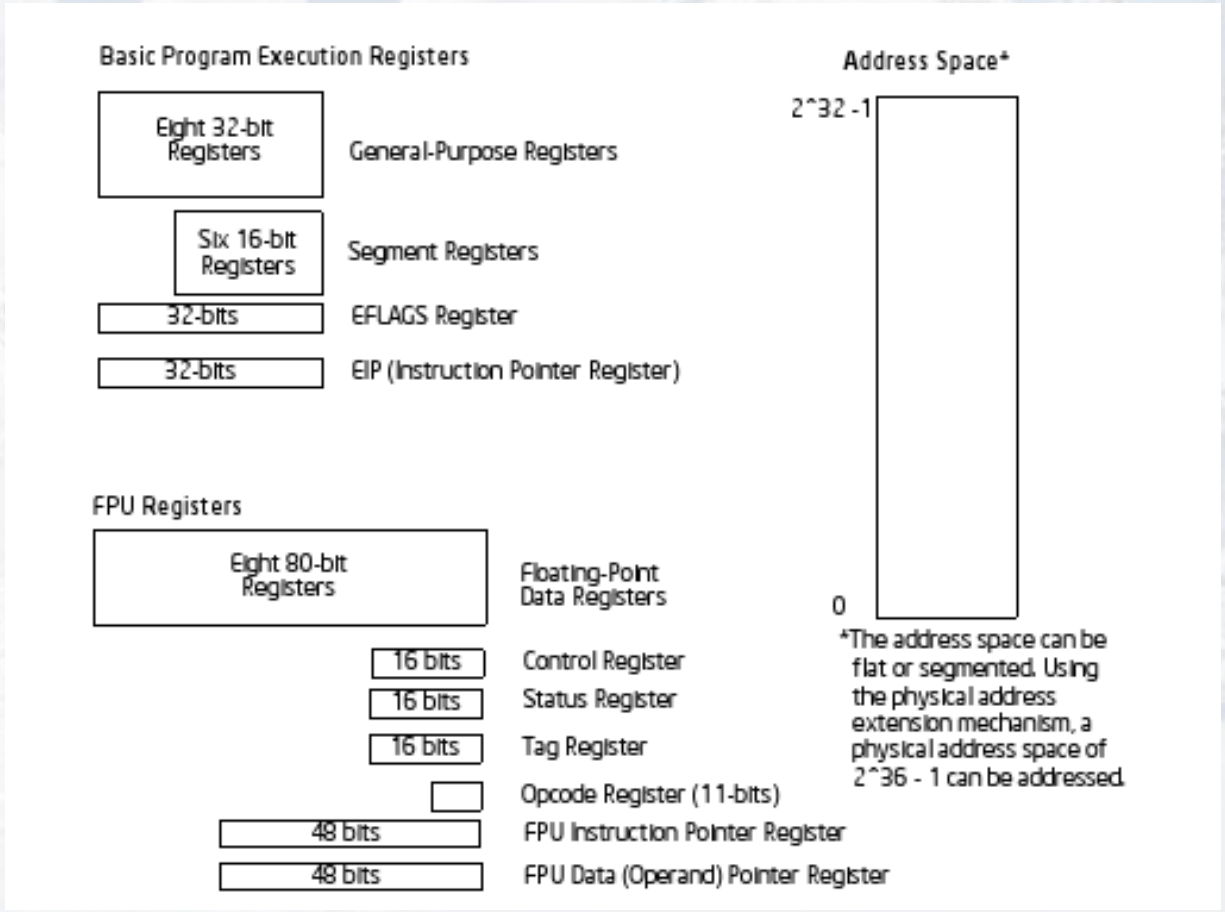
- The industry standard x86 architecture has not changed since 1985
  - Hamstrung by backward compatibility requirements
- Built up by adding new functionality, from the days of the 8080
  - 8080
  - 8088 / 8086
  - 80286, 80386 .. 80486, Pentium, etc

The x86 CPU

- Specifically talking about the CPU …
  - (there is more to the architecture than only the CPU)
- From the SDM ...

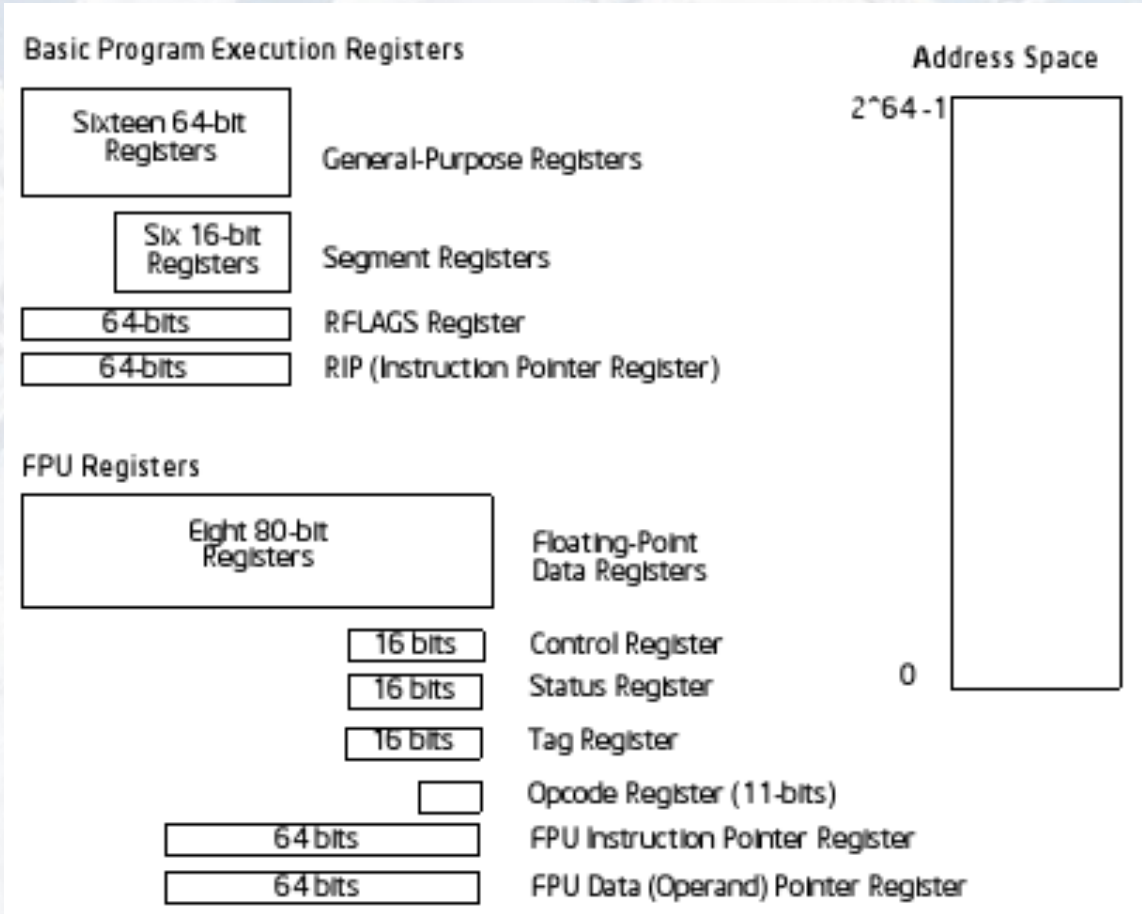
The 32-bit x86 CPU

- Registers
  - GP
  - FP
  - Special
- Address space
  - PA?
  - VA?
- Segments



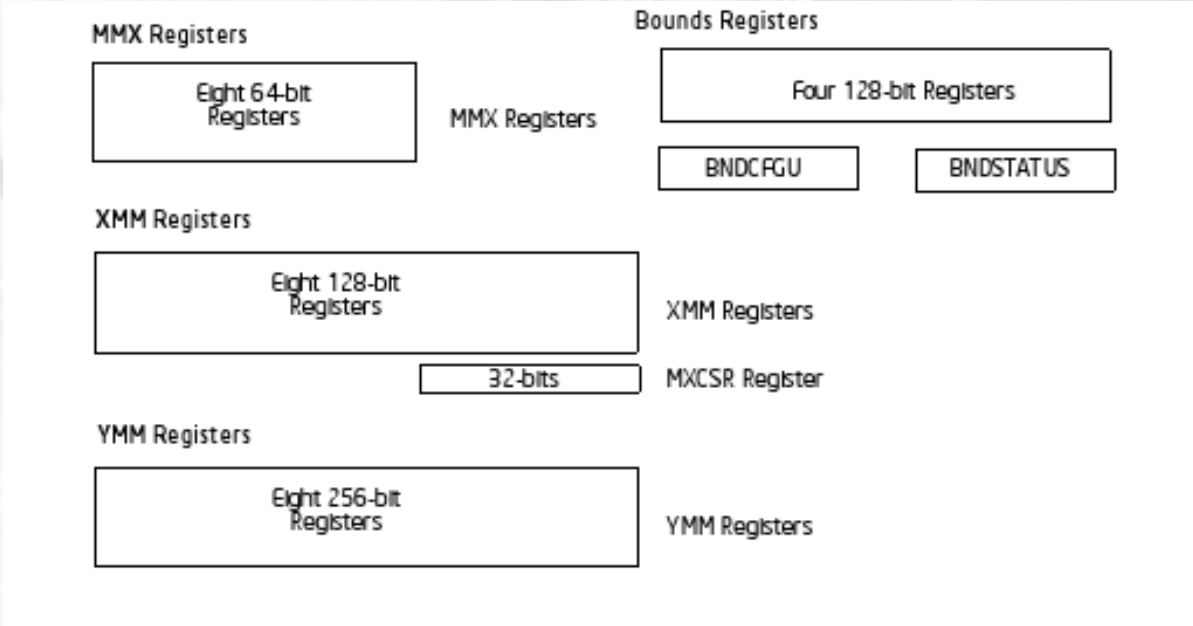
The 64-bit x86 CPU

- Registers
  - Changes?
  - Same?
- Address Space
  - VA/PA ?



The x86 CPU (cont'd)

- Registers
  - More FP
- More on these later...



The x86 CPU

- General purpose registers
  - What are these used for?
  - Example
- Special purpose registers
  - Control registers
  - Debug registers
  - MSRs

The x86 CPU : Memory

- MMU Modes
  - Real
  - Protected
  - 32 / 64-bit paged
- How is memory accessed?
  - Segment : Offset
  - Paging
  - Example

The x86 CPU

- How is physical memory arranged?
  - Contiguous, with 'holes' for various purposes
    - Devices, special purpose regions, etc.
- Does a modern operating system use physical mode addressing?

The x86 CPU (cont'd)

- Why is all this important?
- In a virtualized environment, you must present the same CPU features to each VM as would be present in a non-virtualized environment
  - You need to handle emulating/simulating anything that isn't natively virtualizable!

The x86 Architecture

- Fundamentally, we're doing things the same way we have been since the first 32 bit operating systems appeared
  - With lots of extra "glue" around to handle special cases
- Quiz – what is the first thing that happens when an x86 PC cold boots?

The x86 Architecture

- Boot sequence (uniprocessor)
  - CPU registers set to “sane values”
  - CPU in real mode (16 bit mode)
  - CPU starts executing at CS:IP
    - Real address 0xFFFF0 or 0xFFFFFFF0
- So, what’s at 0xFFFF0 / 0xFFFFFFF0?
  - System BIOS?
  - Something else?

The x86 Architecture

- Generally, it’s the system BIOS
- What does the system BIOS do?
  - Powers up power planes on the system board
  - Identifies buses
  - Powers up buses
  - Identifies basic (legacy) devices
  - Sets up clocks/timers

The x86 Architecture

- System BIOS (cont'd)
  - Initializes PICs / APICs
  - Initializes memory controllers
  - POSTs the video device and other add-on cards
- Once this "hardware init" phase is done...
  - Loads the boot sector from the defined boot device
  - Starts executing boot sector code (in real mode)

The x86 Architecture

- The boot sector (or boot block)...
  - Is only 512 bytes long...
  - What can you do with only 512 bytes?
  - What does the boot sector **need** to do?
- Example..

The x86 Architecture

- What about EFI / UEFI?
  - Replacement for system BIOS
  - Intended to remove much of the legacy goo
  - Provides common API and execution environment for bootloaders
- Goal is the same ... get a second stage boot loader into memory

The x86 Architecture

- Second stage boot loader...
  - Has a little more intelligence
  - What does **it** need to do?
- Example...

The x86 Architecture

- Ok, so now you've got a kernel loaded into RAM.. Now what?
  - Second stage boot loader transfers control to kernel.
  - Might still be running in real mode!

x86 OS Kernels

- For a 32 bit OS (i386 through modern), the kernel is presented with a primitive environment...
  - It needs to set up everything to get started and properly work
  - Setting up addressing modes and the MMU
  - Enabling paging
  - Configuring interrupts

x86 OS Kernels

- Kernel initialization (cont'd)
  - Initializing extended CPU features
  - Setting up buses and configuring devices
  - …
  - …
  - …
  - Finally, start the first process!

x86 32-bit OS Architectures

- Anything from the 80386 to the most modern CPUs support a similar overall architecture
  - Four privilege levels
    - Enables code running in a lower privilege to be effectively isolated from higher privilege level code
  - Segmented addressing
    - A hold-over from old 16-bit legacy systems
    - Almost everyone turns this “off” (more shortly)

x86 32-bit OS Architectures

- 32-bit environments also provide...
  - Provisions for mapping interrupts to code in the kernel
    - These are called interrupt handlers
  - Provisions for handling exceptions
    - Page faults, FPU errors, protection violations, etc
  - Up to 4GB of virtual address space
    - Note that this does not mean **physical** address space!
    - Is this 4GB number entirely accurate?

x86 64-bit OS Architectures

- A 64 bit OS gives the environment
  - More registers to work with
  - 4 billion times more memory to work with
    - Is this entirely accurate?
  - Segmented mode
    - (but it’s just one big flat segment!)

x86 OS Architectures

- There are of course some nuances...
- Multiple processors
  - How are these started and managed?
  - IPIs
  - Locks, contention and other fun stuff

OS Operation

- Fast forward in time, now the OS is started
- The OS sits in a loop, waiting for something to do
  - What external events can trigger the OS to do something?
  - What internal events can trigger the OS to do something?

OS Operation

- Devices can interrupt the OS and instruct it do do something
- A hardware device changes the voltage on a pin connected to the interrupt controler (xPIC)
  - The interrupt controller then interrupts the CPU and lets it know “something needs to be done”
  - This hardware interrupt causes the CPU to do what?

Devices and Interrupts

- The CPU responds to interrupts by examining something called the IDT (remember where this was set up?)
  - What's in the IDT?
- The OS' interrupt dispatch routine is then invoked

Devices and Interrupts

- There are different philosophies of device interrupt management
  - Monolithic
    - “Do all the work now”
  - Two halves/Deferred
    - “Do some of the work now and some later”

Devices and Interrupts

- Monolithic interrupt handlers
  - … handle all device I/Os in one operation
  - … acknowledge the interrupt
  - … return the OS, all in one code path
- Two-half / deferred interrupt handlers
  - … extract the relevant information from the interrupt source, store in memory
  - … acknowledge the interrupt
  - … schedule a deferred handling of the interrupt payload “for later”

Devices and Interrupts

- Acknowledging the interrupt tells the xPIC (and later, the device) that the interrupt has been seen
  - … but not necessarily processed
- It is the responsibility of the deferral code to ensure that the correct thing is done with the payload!

Devices and Interrupts

- Interrupts handled by the OS
  - … disk block writes
  - … transmission of network packets
  - … etc
- What about software interrupts?
  - … system calls and the like
  - Do these cause interruptions in normal code execution?

Devices, CPUs, and MMUs

- There is a tight (very tight) coupling between devices, CPU operations, and the MMU in an x86 system
- The MMU is a key participant in all of this
  - … why?

MMUs

- Intel has spent a lot of time (and money) to ensure that the MMU in modern x86 CPUs is as fast as possible
  - Historically, why did this occur?
  - Is it better with 64-bit CPUs?
- The MMU is implemented in fast silicon, but as we will see, silicon is hard to emulate!
  - And you'll need to do at least *some* emulation in your VMM

Other Stuff

- DMA
  - Fast memory transfer from devices
  - What page protections does this use?
  - Is the MMU involved?
- Timers (and time in general)

Shutdown

- Fast forward to the end of time …
- How does the machine shut down?
  - Shut down (for real)
  - Shut down (temporarily)

Shutdown

- Shutdown processing
  - … stop all the user applications
  - … stop all new I/Os
  - … flush existing I/Os
  - … power down/shutdown devices
  - … power down/shutdown buses
  - … quiesce CPUs
  - … “somehow” shutdown

A Glimpse Forward...

- Out of all this, what does a system virtualization software product need to deal with?
  - How is all that hardware interaction managed?
    - Much (nearly all) is highly timing dependent!
  - How is all that MMU interaction managed?
  - What kinds of performance do such systems yield?
- Is there a better way?

Reading

- https://github.com/0xAX/linux-insides/tree/master/Booting
- Clone the repo or just read online

- Intel SDM
  - Volume 3, Ch. 6 and 9 (omit 9.2 and 9.11)