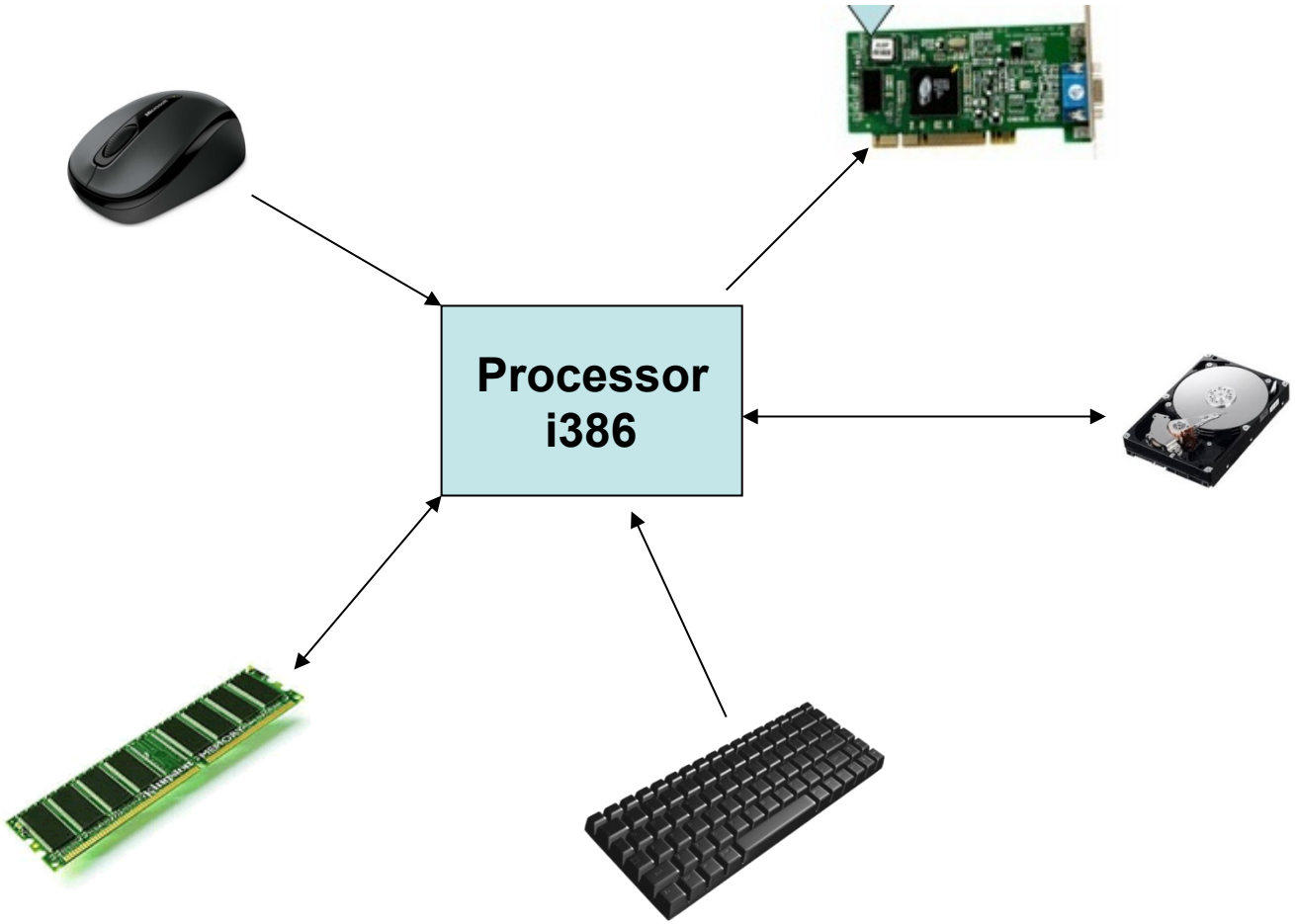
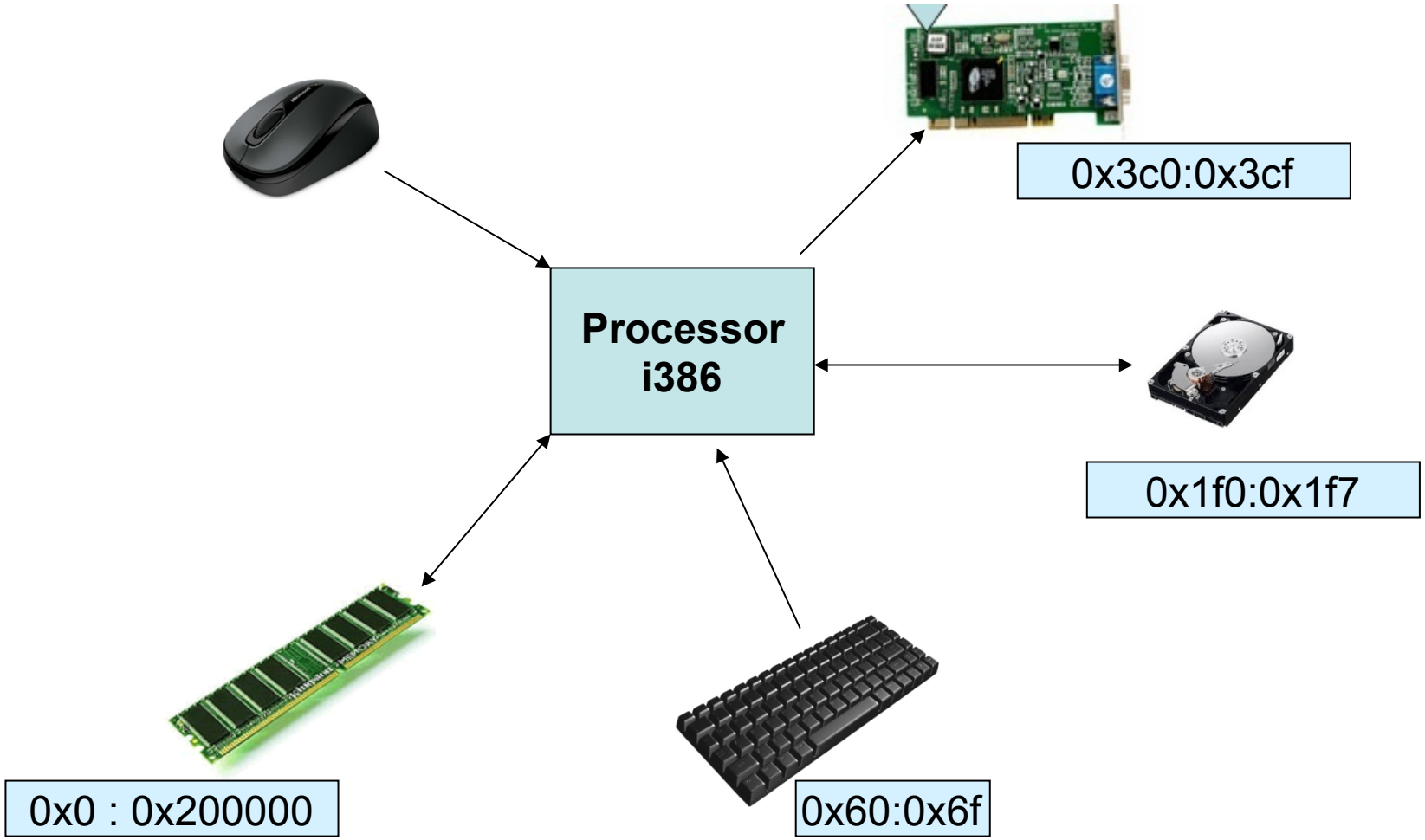
PC Hardware and Booting

Chester Rebeiro
IIT Madras

CPUs



Everything has an address

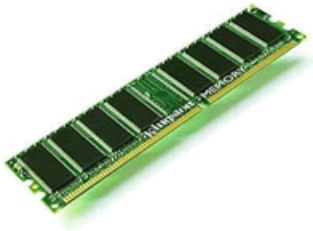


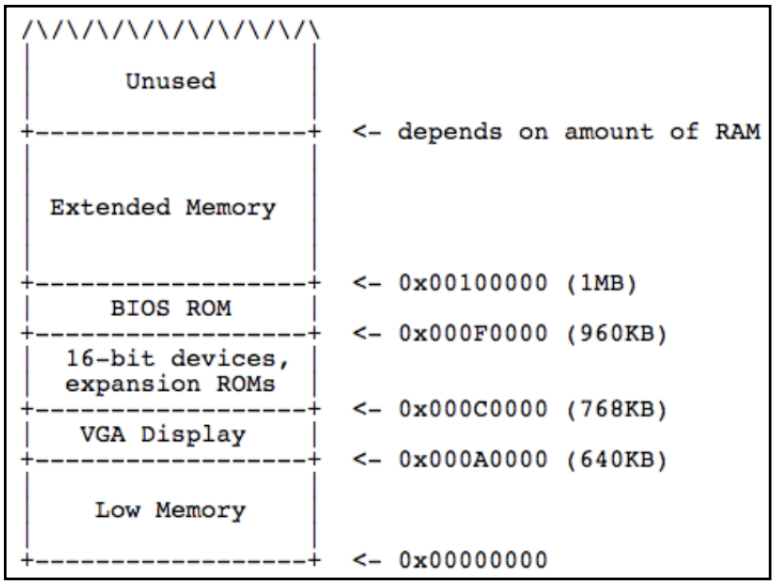
Address Types

- Memory Addresses
- IO Addresses
- Memory Mapped IO Addresses

Address Types : (Memory Addresses)

- Range : 0 to (RAM size or 2^{32}-1)
- Where main memory is mapped
  - Used to store data for code, heap, stack, OS, etc.
- Accessed by load/store instructions





Memory Address Map

Low and Extended Memory (Legacy Issues)

- Why study it?
  - Backward compatibility
- 8088 has 20 address lines; can address 2^{20} bytes (1MB)
- Memory Ranges
  - 0 to 640KB used by IBM PC MSDOS
    - Other DOS versions have a different memory limit
  - 640 KB to 1MB used by video buffers, expansion ROMS, BIOS ROMs
  - 1 MB onwards called extended memory
- Modern processors have more usable memory
  - OSes like Linux and x86 simply ignore the first 1MB and load kernel in extended memory

Address Types : (IO Ports)

- Range : 0 to 2^{16}-1
- Used to access devices
- Uses a different bus compared to RAM memory access
  - Completely isolated from memory
- Accessed by in/out instructions

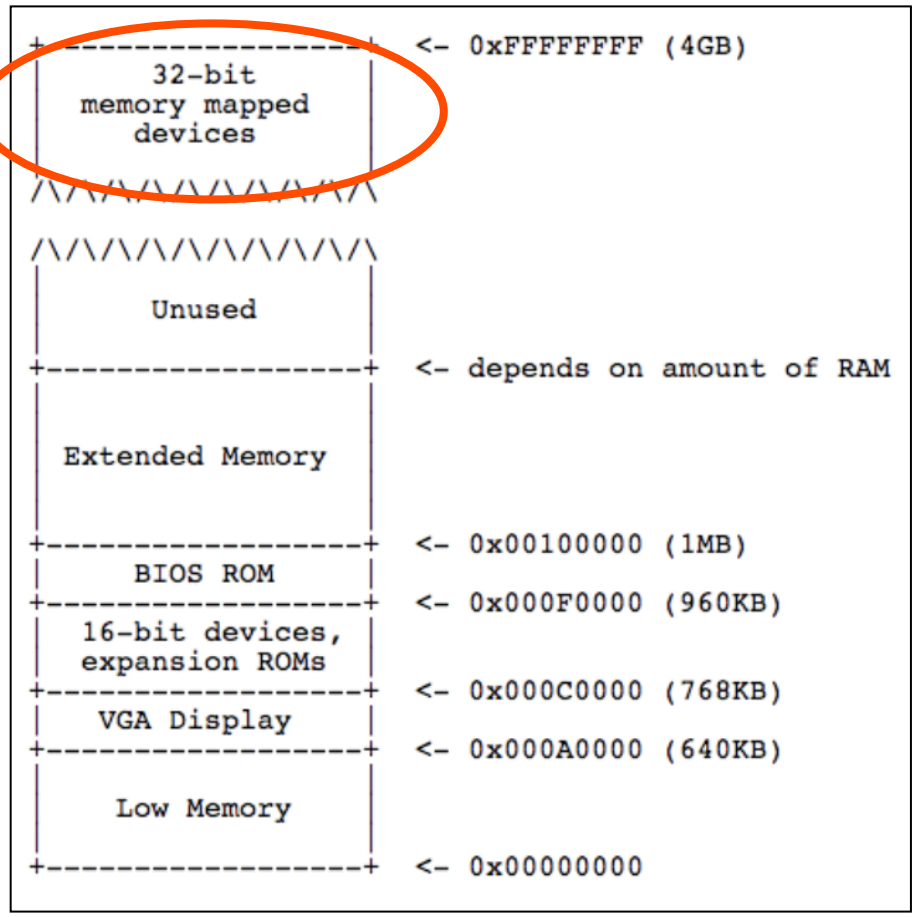
inb $0x64, %al
outb %al, $0x64

I/O address range	Device
00 – 1F	First DMA controller 8237 A-5
20 – 3F	First Programmable Interrupt Controller, 8259A, Master
40 – 5F	Programmable Interval Timer (System Timer), 8254
60 – 6F	Keyboard, 8042
70 – 7F	Real Time Clock, NMI mask
80 – 9F	DMA Page Register, 74LS612
87	DMA Channel 0
83	DMA Channel 1
81	DMA Channel 2
82	DMA Channel 3
8B	DMA Channel 5
89	DMA Channel 6
8A	DMA Channel 7
8F	Refresh
A0 – BF	Second Programmable Interrupt Controller, 8259A, Slave
C0 – DF	Second DMA controller 8237 A-5
F0	Clear 80287 Busy
F1	Reset 80287
F8 – FF	Math coprocessor, 80287
F0 – F5	PCjr Disk Controller
F8 – FF	Reserved for future microprocessor extensions
100 – 10F	POS Programmable Option Select (PS/2)
110 – 1EF	System I/O channel
140 – 15F	Secondary SCSI host adapter
170 – 177	Secondary Parallel ATA Disk Controller
1F0 – 1F7	Primary Parallel ATA Hard Disk Controller
200 – 20F	Game port
210 – 217	Expansion Unit
220 – 233	Sound Blaster and most other sound cards

ref : http://bochs.sourceforge.net/techspec/PORTS.LST

Memory Mapped I/O

- Why?
  - More space
- Devices and RAM share the same address space
- Instructions used to access RAM can also be used to access devices.
  - Eg load/store

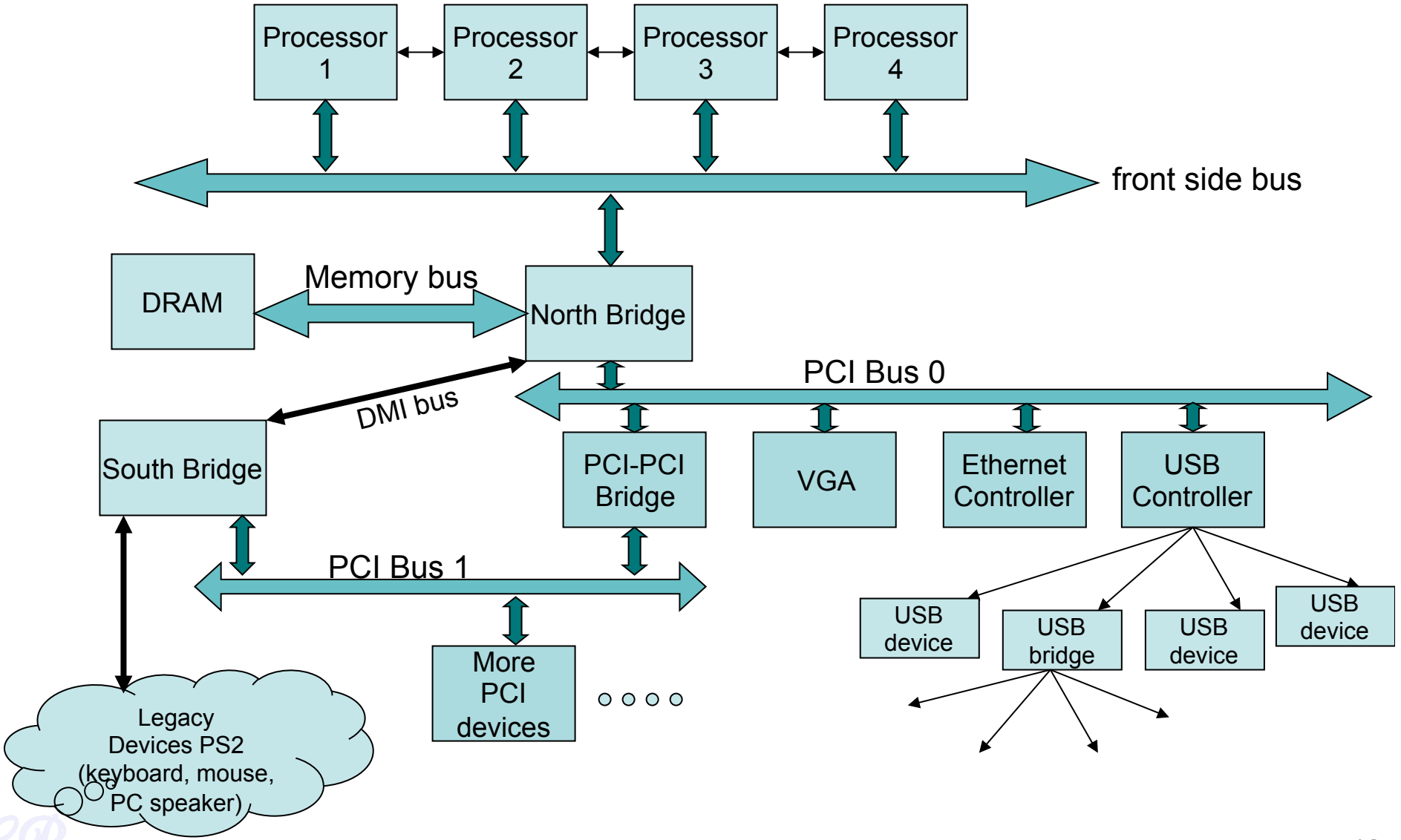


Memory Map

Who decides the address ranges?

- Standards / Legacy
  - Such as the IBM PC standard
  - Fixed for all PCs.
  - Ensures BIOS and OS to be portable across platforms
- Plug and Play devices
  - Address range set by BIOS or OS
  - A device address range may vary every time the system is restarted

PC Organization



The x86 Evolution (8088)

- **8088**
  - 16 bit microprocessor
  - 20 bit external address bus
    - **Can address 1MB of memory**
  - Registers are 16 bit

    General Purpose Registers

    AX, BX, CD, DX,

    Pointer Registers

    BP, SI, DI, SP

    Instruction Pointer : IP

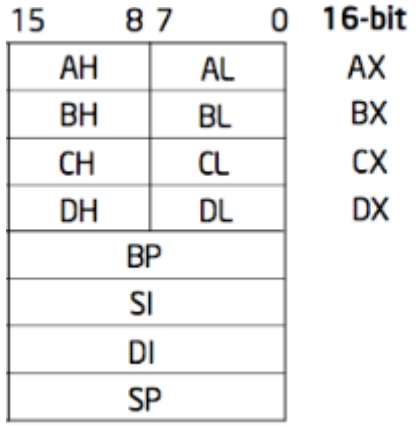
    Segment Registers

    CS, SS, DS, ES
  - Accessing memory

    (segment_base << 4) + offset

    eg: (CS << 4) + IP

General Purpose Registers



GPRs can be accessed as 8 bit or 16 bit registers

Eg.

```
mov $0x1, %ah  ; 8 bit move
mov $0x1, %ax  ; 16 bit move
```

The x86 Evolution (80386)

- **80386** (1995)
  - 32 bit microprocessor
  - 32 bit external address bus
    - **Can address 4GB of memory**
  - Registers are 32 bit

    General Purpose Registers

    EAX, EBX, ECD, EDX,

    Pointer Registers

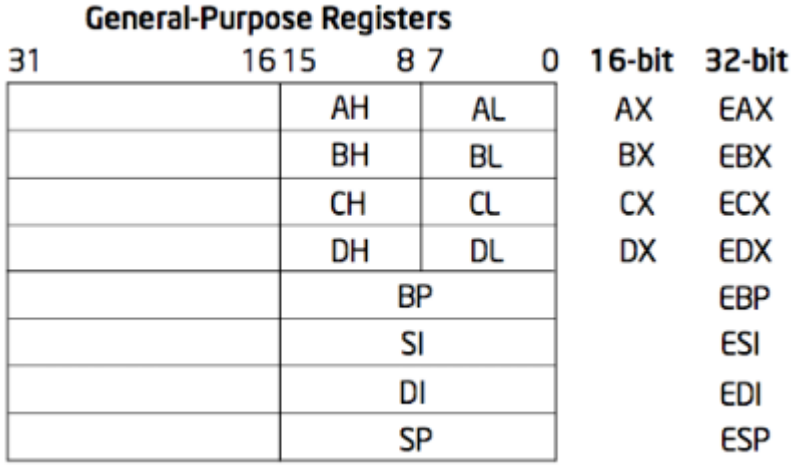
    EBP, ESI, EDI, ESP

    Instruction Pointer : IP

    Segment Registers

    CS, SS, DS, ES
  - Lot more features
    - Protected operating mode
    - Virtual addresses

General Purpose Registers



GPRs can be accessed as 8, 16, 32 bit registers

e.g.

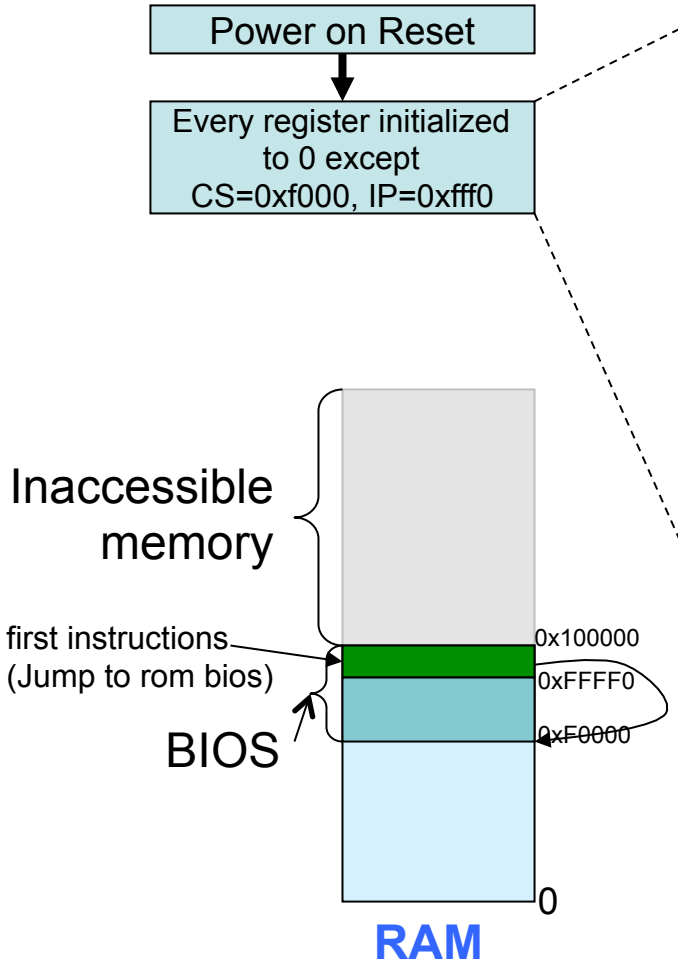
```
mov $0x1, %ah  ; 8 bit move
mov $0x1, %ax  ; 16 bit move
mov $0x1, %eax ; 32 bit move
```

The x86 Evolution (k8)

- **AMD k8** (2003)
  - RAX instead of EAX
  - X86-64, x64, amd64, intel64: all same thing
- **Backward compatibility**
  - All systems backward compatible with 8088

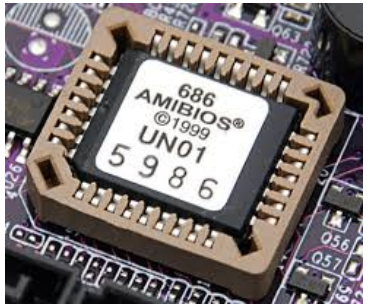
Powering Up

Powering up : Reset

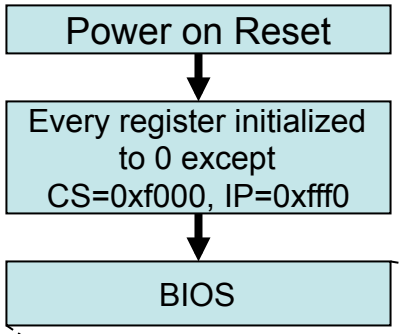


Physical address = (CS << 4) + IP
= 0xffff0

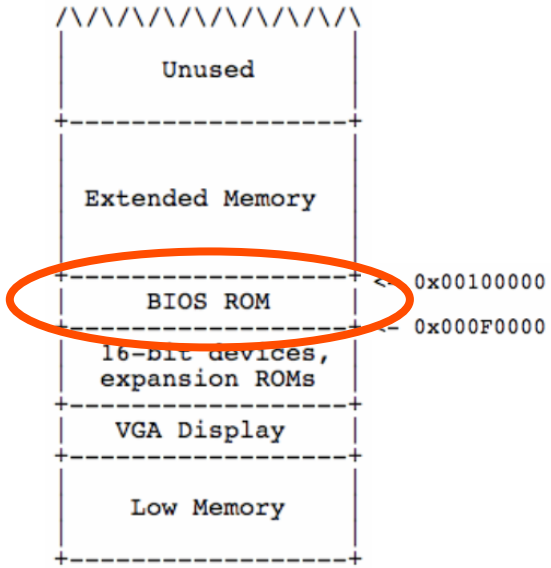
- first instruction fetched from location 0xffff0.
- Processor in real mode (backward compatible to 8088)
  - Limited to 1MB addresses
  - No protection; no privilege levels
  - Direct access to all memory
  - No multi-tasking
- First instruction is right on top of accessible memory
  - Should jump to another location

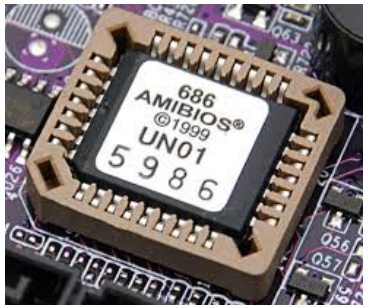


Powering up : BIOS

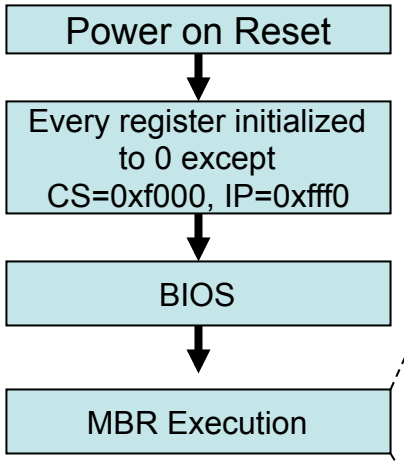


- Present in a small chip connected to the processor
  - Flash/EPROM/E^{2}PROM
- Does the following
  - Power on self test
  - Initialize video card and other devices
  - Display BIOS screen
  - Perform brief memory test
  - Set DRAM memory parameters
  - Configure Plug & Play devices
  - Assign resources (DMA channels & IRQs)
  - Identify the boot device
    - Read sector 0 from boot device into memory location **0x7c00**
    - Jumps to 0x7c00



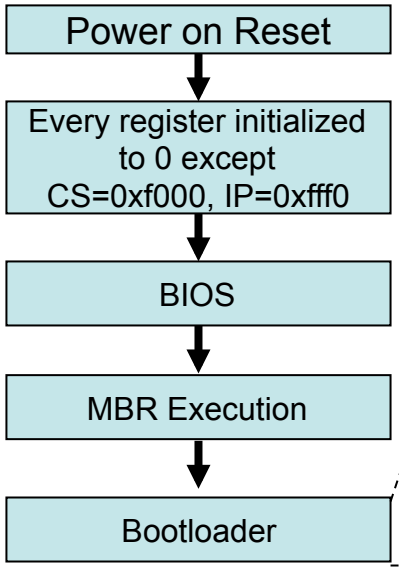


Powering up : MBR



- Sector 0 in the disk called Master Boot Record (MBR)
- Contains code that boots the OS or another boot loader
- Copied from disk to RAM (@0x7c00) by BIOS and then begins to execute
- Size 512 bytes
  - 446 bytes bootable code
  - 64 bytes disk partition information (16 bytes per partition)
  - 2 bytes signature
- Typically, MBR code looks through partition table and loads the bootloader (such as Linux or Windows)
- or, it may directly load the OS

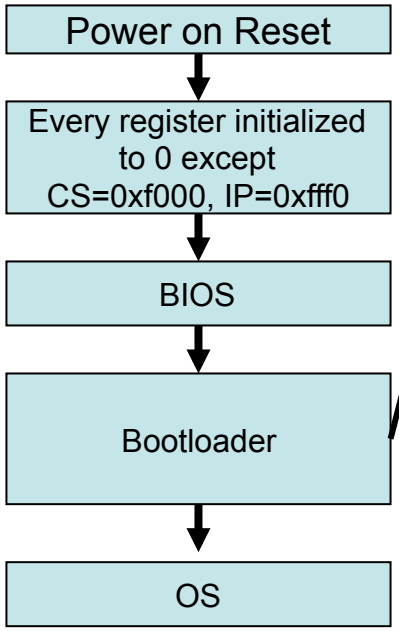
Powering Up : bootloader



- Loads the operating system
  - May also allow the user to select which OS to load (eg. Windows or Linux)
- Other jobs done
  - Disable interrupts :
    - Don't want to bother with interrupts at this stage
    - Interrupts re-enabled by xv6 when ready
  - Setup GDT
  - Switch from real mode to protected mode
  - Read operating system from disk

The bootloader may be present in the MBR (sector 0) itself

Powering Up : xv6



- Bootloader
  - Present in sector 0 of disk.
  - 512 bytes
  - 2 parts:
    - bootasm.S (*8900*)
      - Enters in 16 bit real mode, leaves in 32 bit protected mode
      - Disables interrupts
        - We don't want to use BIOS ISRs
      - Enable A20 line
      - Load GDT (only segmentation, no paging)
      - Set stack to 0x7c00
      - Invoke bootmain
      - Never returns
    - bootmain.c (*9017*)
      - Loads the xv6 kernel from sector 1 to RAM starting at 0x100000 (1MB)
      - Invoke the xv6 kernel entry
        - _start present in entry.S (sheet 10)
        - This entry point is known from the ELF header

Gets loaded into
0x7c00
by the BIOS?

xv6 : bootasm.S

```
8909 .code16                       # Assemble for 16-bit mode
8910 .globl start
8911 start:
8912   cli                         # BIOS enabled interrupts; disable
8913
8914   # Zero data segment registers DS, ES, and SS.
8915   xorw    %ax,%ax             # Set %ax to zero
8916   movw    %ax,%ds             # -> Data Segment
8917   movw    %ax,%es             # -> Extra Segment
8918   movw    %ax,%ss             # -> Stack Segment
8919
8920   # Physical address line A20 is tied to zero so that the first PCs
8921   # with 2 MB would run software that assumed 1 MB.  Undo that.
8922 seta20.1:
8923   inb     $0x64,%al               # Wait for not busy
8924   testb   $0x2,%al
8925   jnz     seta20.1
8926
8927   movb    $0xd1,%al               # 0xd1 -> port 0x64
8928   outb    %al,$0x64
8929
8930 seta20.2:
8931   inb     $0x64,%al               # Wait for not busy
8932   testb   $0x2,%al
8933   jnz     seta20.2
8934
8935   movb    $0xdf,%al               # 0xdf -> port 0x60
8936   outb    %al,$0x60
```

Note 16 bit code (compatible with 8088)

Loading :
Handled by the BIOS

Linking :
What linker options need to be set?

xv6 : bootasm.S

```
8909 .code16                       # Assemble for 16-bit mode
8910 .globl start
8911 start:
8912   cli                         # BIOS enabled interrupts; disable
8913
8914   # Zero data segment registers DS, ES, and SS.
8915   xorw    %ax,%ax              # Set %ax to zero
8916   movw    %ax,%ds              # -> Data Segment
8917   movw    %ax,%es              # -> Extra Segment
8918   movw    %ax,%ss              # -> Stack Segment
8919
8920   # Physical address line A20 is tied to zero so that the first PCs
8921   # with 2 MB would run software that assumed 1 MB.  Undo that.
8922 seta20.1:
8923   inb     $0x64,%al               # Wait for not busy
8924   testb   $0x2,%al
8925   jnz     seta20.1
8926
8927   movb    $0xd1,%al               # 0xd1 -> port 0x64
8928   outb    %al,$0x64
8929
8930 seta20.2:
8931   inb     $0x64,%al               # Wait for not busy
8932   testb   $0x2,%al
8933   jnz     seta20.2
8934
8935   movb    $0xdf,%al               # 0xdf -> port 0x60
8936   outb    %al,$0x60
```

Disable interrupts.
Initialize registers to 0

xv6 : bootasm.S

```
8909 .code16                       # Assemble for 16-bit mode
8910 .globl start
8911 start:
8912   cli                         # BIOS enabled interrupts; disable
8913
8914   # Zero data segment registers DS, ES, and SS.
8915   xorw    %ax,%ax              # Set %ax to zero
8916   movw    %ax,%ds              # -> Data Segment
8917   movw    %ax,%es              # -> Extra Segment
8918   movw    %ax,%ss              # -> Stack Segment
8919
8920   # Physical address line A20 is tied to zero so that the first PCs
8921   # with 2 MB would run software that assumed 1 MB.  Undo that.
8922 seta20.1:
8923   inb     $0x64,%al               # Wait for not busy
8924   testb   $0x2,%al
8925   jnz     seta20.1
8926
8927   movb    $0xd1,%al               # 0xd1 -> port 0x64
8928   outb    %al,$0x64
8929
8930 seta20.2:
8931   inb     $0x64,%al               # Wait for not busy
8932   testb   $0x2,%al
8933   jnz     seta20.2
8934
8935   movb    $0xdf,%al               # 0xdf -> port 0x60
8936   outb    %al,$0x60
```

Enable A20 line.

Why do we have it?

xv6 : bootasm.S

Switch from real to protected mode

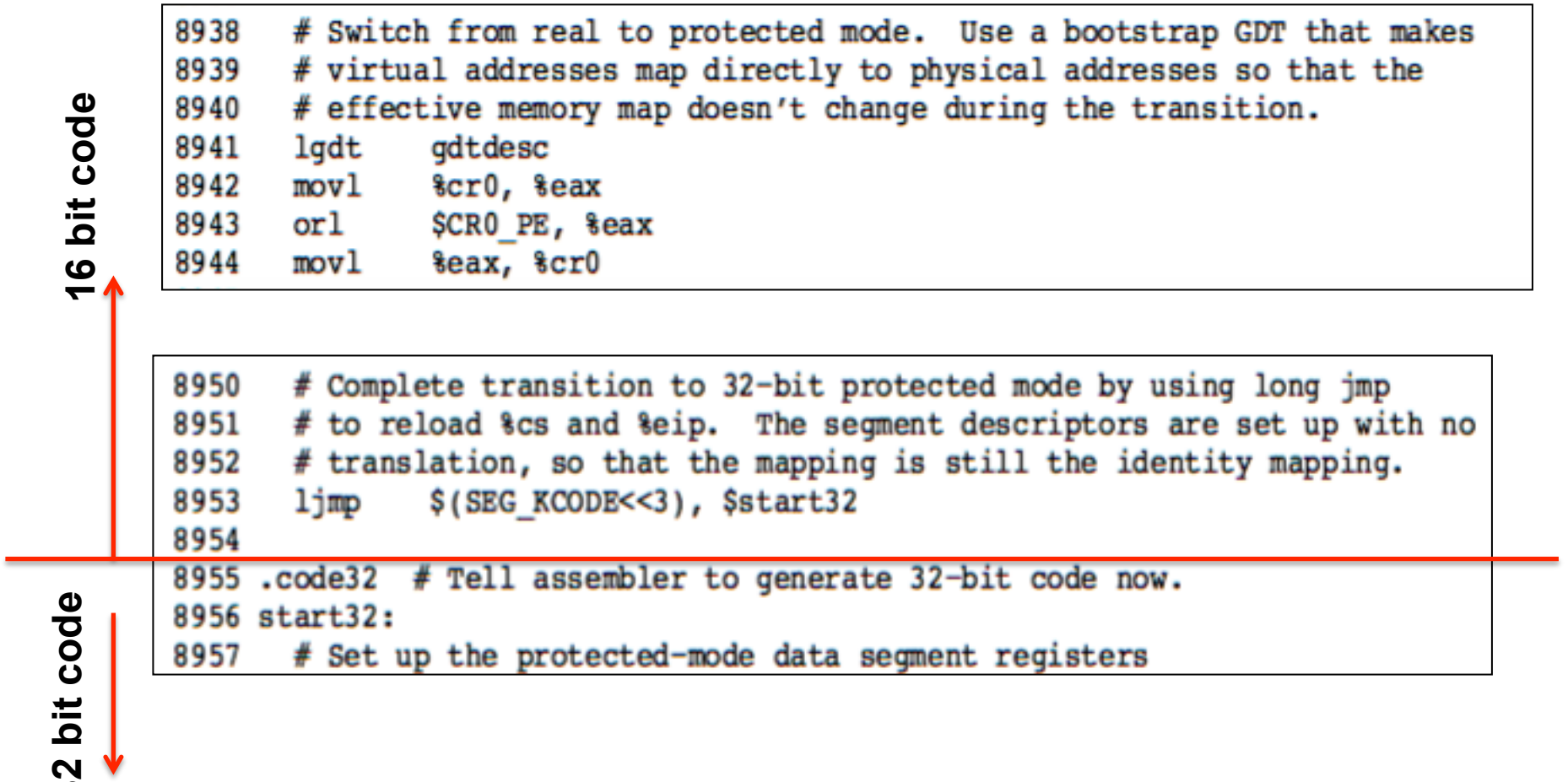
```
8938    # Switch from real to protected mode.  Use a bootstrap GDT that makes
8939    # virtual addresses map directly to physical addresses so that the
8940    # effective memory map doesn't change during the transition.
8941    lgdt    gdtdesc
8942    movl    %cr0, %eax
8943    orl     $CR0_PE, %eax
8944    movl    %eax, %cr0
```

```
8980 # Bootstrap GDT
8981 .p2align 2                                # force 4 byte alignment
8982 gdt:
8983   SEG_NULLASM                             # null seg
8984   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
8985   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
8986
8987 gdtdesc:
8988   .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
8989   .long   gdt                             # address gdt
8990
```

GDT related information

xv6 : bootasm.S

Switch from real to protected mode



xv6 : bootasm.S

```
8909 .code16                       # Assemble for 16-bit mode
8910 .globl start
8911 start:
8912   cli                         # BIOS enabled interrupts; disable
8913
8914   # Zero data segment registers DS, ES, and SS.
8915   xorw    %ax,%ax              # Set %ax to zero
8916   movw    %ax,%ds              # -> Data Segment
8917   movw    %ax,%es              # -> Extra Segment
8918   movw    %ax,%ss              # -> Stack Segment
8919
8920   # Physical address line A20 is tied to zero so that the first PCs
8921   # with 2 MB would run software that assumed 1 MB.  Undo that.
8922 seta20.1:
8923   inb     $0x64,%al               # Wait for not busy
8924   testb   $0x2,%al
8925   jnz     seta20.1
8926
8927   movb    $0xd1,%al               # 0xd1 -> port 0x64
8928   outb    %al,$0x64
8929
8930 seta20.2:
8931   inb     $0x64,%al               # Wait for not busy
8932   testb   $0x2,%al
8933   jnz     seta20.2
8934
8935   movb    $0xdf,%al               # 0xdf -> port 0x60
8936   outb    %al,$0x60
```

Enable A20 line.

Why do we have it?

xv6 : bootasm.S

```
8956 start32:
8957   # Set up the protected-mode data segment registers
8958   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
8959   movw    %ax, %ds                # -> DS: Data Segment
8960   movw    %ax, %es                # -> ES: Extra Segment
8961   movw    %ax, %ss                # -> SS: Stack Segment
8962   movw    $0, %ax                 # Zero segments not ready for use
8963   movw    %ax, %fs                # -> FS
8964   movw    %ax, %gs                # -> GS
8965
8966   # Set up the stack pointer and call into C.
8967   movl    $start, %esp
8968   call    bootmain
```

Set up stack and call a C function.

Note the stack pointer points to 0x7c00.
This means the stack grows downwards
from 0x7c00. why?

xv6 : bootasm.S

```
8956 start32:
8957   # Set up the protected-mode data segment registers
8958   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
8959   movw    %ax, %ds                # -> DS: Data Segment
8960   movw    %ax, %es                # -> ES: Extra Segment
8961   movw    %ax, %ss                # -> SS: Stack Segment
8962   movw    $0, %ax                 # Zero segments not ready for use
8963   movw    %ax, %fs                # -> FS
8964   movw    %ax, %gs                # -> GS
8965
8966   # Set up the stack pointer and call into C.
8967   movl    $start, %esp
8968   call    bootmain
```

Set up stack and call a C function.

Note the stack pointer points to 0x7c00.
This means the stack grows downwards
from 0x7c00. why?

bootmain

```
9016 void
9017 bootmain(void)
9018 {
9019   struct elfhdr *elf;
9020   struct proghdr *ph, *eph;
9021   void (*entry)(void);
9022   uchar* pa;
9023
9024   elf = (struct elfhdr*)0x10000;  // scratch space
9025
9026   // Read 1st page off disk
9027   readseg((uchar*)elf, 4096, 0);
9028
9029   // Is this an ELF executable?
9030   if(elf->magic != ELF_MAGIC)
9031     return;  // let bootasm.S handle error
9032
9033   // Load each program segment (ignores ph flags).
9034   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
9035   eph = ph + elf->phnum;
9036   for(; ph < eph; ph++){
9037     pa = (uchar*)ph->paddr;
9038     readseg(pa, ph->filesz, ph->off);
9039     if(ph->memsz > ph->filesz)
9040       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
9041   }
9042
9043   // Call the entry point from the ELF header.
9044   // Does not return!
9045   entry = (void(*)(void))(elf->entry);
9046   entry();
9047 }
```

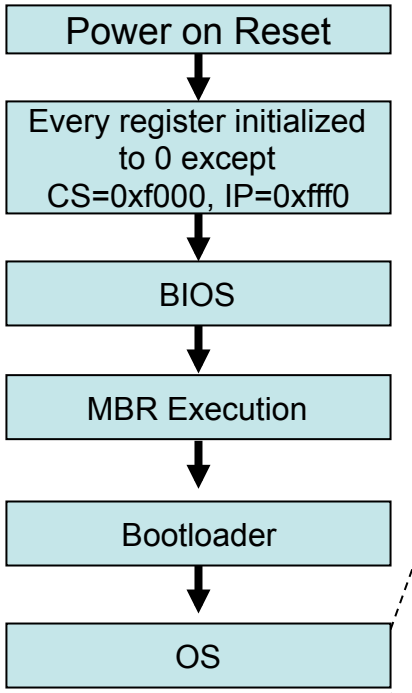
Load in 1MB region

The xv6 kernel is stored as an ELF image.

Read kernel from the disk (sector 1) to RAM.

Read the entry function in the kernel and Invoke it.
This starts the OS

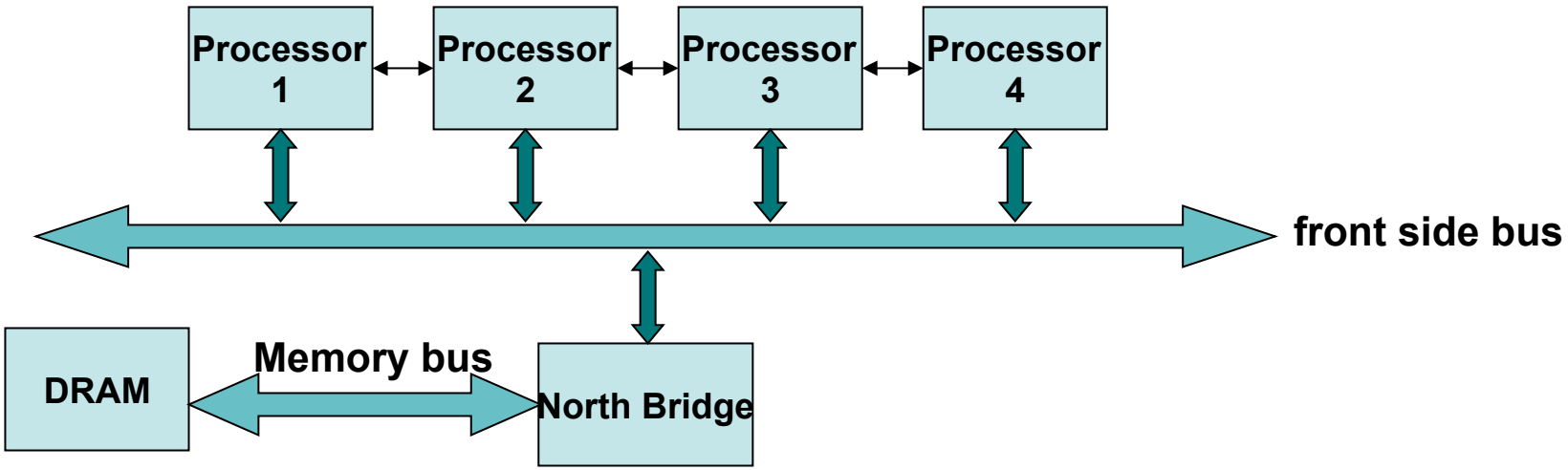
Powering Up : OS



- Set up virtual memory
- Initialize interrupt vectors
- Initilize
  - timers,
  - monitors,
  - hard disks,
  - consoles,
  - filesystems,
- Initialized other processors (if any)
- Startup user process

Multi-processor Bootup

Multiprocessor Organization



- Memory Symmetry
  - All processors in the system share the same memory space
  - Advantage : Common operating system code
- I/O Symmetry
  - All processors share the same I/O subsystem
  - Every processor can receive interrupt from any I/O device

Multiprocessor Booting

- One processor designated as ‘Boot Processor’ (BSP)
  - Designation done either by Hardware or BIOS
  - All other processors are designated AP (Application Processors)
- BIOS boots the BSP
- BSP learns system configuration
- BSP triggers boot of other AP
  - Done by sending an Startup IPI (inter processor interrupt) signal to the AP

http://www.intel.com/design/pentium/datashts/24201606.pdf

xv6 Multiprocessor Boot

- mpinit (7001) invoked from main (1221)
  - Searches for an MP table in memory
    - (generally put there by the BIOS)
    - Contains information about processors in system along with other details such as IO-APICs, Processor buses, etc.
    - Extracts system information from MP table
  - Fills in the cpu id (7024)
    - CPU is a structure which contains CPU specific data (2304)

Booting APs

- startothers (1274) invoked from main(1237)
  - copy '*entryother*' to location 0x7000
  - For each CPU found
    - Allocate a stack (1295)
    - Set C entry point to *mpenter* (1252)
    - Send a Startup IPI (1299)
      - Pass the entryother.S location to the new processor (40:67 ← 0x7000 >> 4)
      - Send inter processor interrupt to the AP processor using its apicid
    - Wait until CPU has started

for next class

- Read / revise about memory management in x86 especially
  - Segmentation (GDT)
  - Virtual memory (page tables, CR3 register, etc)