

CS3210: Booting and x86

Lecture 2

Instructor: Dr. Tim Andersen

Today: Bootstrapping

- CPU -> needs a first instruction
- Memory -> needs initial code/data
- I/O -> needs to know how to communicate

What happens after power on?

- High-level: Firmware -> Bootloader -> OS kernel
 - e.g., jos: BIOS -> boot/* -> kern/*
 - e.g., xv6: BIOS -> bootblock -> kernel
 - e.g., Linux: BIOS/UEFI -> LILO/GRUB/syslinux -> vmlinuz
- Why three steps?
- What are the handover protocols?

BIOS: Basic Input/Output System

- QEMU uses an opensource BIOS, called **SeaBIOS**
 - e.g., try to run `qemu-system-i386` (with no arguments)

```
SeaBIOS (version 1.9.0-20151117_093733-anatol)
```

```
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+
```

```
Booting from Hard Disk...
```

```
Boot failed: could not read the boot disk
```

```
Booting from Floppy...
```

```
Boot failed: could not read the boot disk
```

```
Booting from DVD/CD...
```

```
Boot failed: Could not read from CDROM (code 0003)
```

```
Booting from ROM...
```

```
iPXE (PCI 00:03.0) starting execution...ok
```

```
iPXE initialising devices...ok
```

From power-on to BIOS in x86 (miniboot)

- Set IP -> 4GB - 16B (0xffffffff0)
 - e.g., 80286: 1MB - 16B (0xffff0)
 - e.g., SPARCS v8: 0x00 (reset vector)

[[DEMO]]: x86 initial state on QEMU

The first instruction

- To understand, we first need to understand:
 1. x86 state (e.g., registers)
 2. Memory referencing model (e.g., segmentation)
 3. BIOS features (e.g., memory aliasing)

```
(gdb) x/1i 0xffffffff0  
0xffffffff0:  ljmp  $0xf000,$0xe05b
```

x86 execution environment (IA32, Ch.3.2)

- CPU Registers:
 - instruction pointer (IP)
 - 8 general purpose regs
 - 6 segment regs
 - status regs
 - control regs
- Address space: 0-2³², 4GB
- I/O ports: 1024 space

x86 registers (IA32, Ch.3.4)

- 8, 16, 32 bit version
- General purpose, but with convention

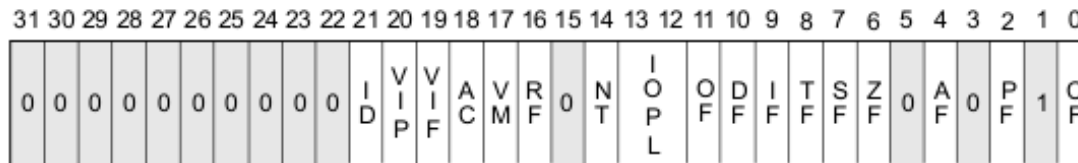
General-Purpose Registers				16-bit	32-bit
31	16 15	8 7	0		
	AH	AL		AX	EAX
	BH	BL		BX	EBX
	CH	CL		CX	ECX
	DH	DL		DX	EDX
	BP				EBP
	SI				ESI
	DI				EDI
	SP				ESP

Figure 3-5. Alternate General-Purpose Register Names

x86 registers (IA32, Ch.3.4)

- [[EAX]]: **accumulator register**
- [[EBX]]: **base register**
- [[ECX]]: **count register**
- [[EDX]]: **data register**
- [[ESI]]: **source index**
- [[EDI]]: **destination index**
- [[EBP]]: **base pointer**
- [[ESP]]: **stack pointer**
- [[EIP]]: **instruction pointer**

x86 EFLAGS



- X ID Flag (ID)
- X Virtual Interrupt Pending (VIP)
- X Virtual Interrupt Flag (VIF)
- X Alignment Check / Access Control (AC)
- X Virtual-8086 Mode (VM)
- X Resume Flag (RF)
- X Nested Task (NT)
- X I/O Privilege Level (IOPL)
- S Overflow Flag (OF)
- C Direction Flag (DF)
- X Interrupt Enable Flag (IF)
- X Trap Flag (TF)
- S Sign Flag (SF)
- S Zero Flag (ZF)
- S Auxiliary Carry Flag (AF)
- S Parity Flag (PF)
- S Carry Flag (CF)

QEMU's internal state

(qemu) info registers

```
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000663
ESI=00000000 EDI=00000000 EBP=00000000 ESP=00000000
EIP=0000ffff EFL=00000002 [-----] CPL=0  II=0  A20=1  SMM=0  HLT=0
ES =0000 00000000 0000ffff 00009300
CS =f000 ffff0000 0000ffff 00009b00
SS =0000 00000000 0000ffff 00009300
DS =0000 00000000 0000ffff 00009300
FS =0000 00000000 0000ffff 00009300
GS =0000 00000000 0000ffff 00009300
LDT=0000 00000000 0000ffff 00008200
TR =0000 00000000 0000ffff 00008b00
GDT=      00000000 0000ffff
IDT=      00000000 0000ffff
CR0=60000010 CR2=00000000 CR3=00000000 CR4=00000000
```

Memory referencing model: Segmentation

- Initial IP -> 0xffffffff, but EIP=0x000fff0?
- Interpreted by segmentation:
 - e.g., 0xffffffff = 0xffff0000 (base) + 0x000fff0 (EIP)

```
ES =0000 00000000 0000ffff 00009300
CS =f000 ffff0000 0000ffff 00009b00
SS =0000 00000000 0000ffff 00009300
DS =0000 00000000 0000ffff 00009300
FS =0000 00000000 0000ffff 00009300
GS =0000 00000000 0000ffff 00009300
```

x86 Segmentation

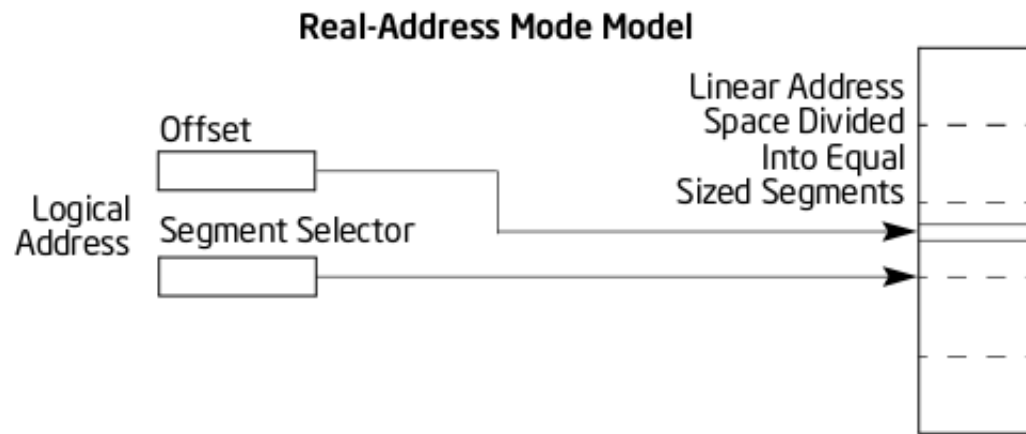
- Segment selectors (16 bits): hold "segment selector"
 - `[[CS]]`: Code Segment
 - `[[DS]]`: Data Segment
 - `[[SS]]`: Stack Segment
 - `[[ES]]`: Extra Segment
 - `[[FS]]`: F Segment
 - `[[GS]]`: G Segment

History: Segmentation in 8086

- 16-bit micro-processor (2^{16} , 64KB)
- Has 20-bit physical addr (2^{20} , 1MB)
- Uses extra four bits from a 16-bit "segment register"
 - CS: for fetches via IP
 - DS: for load/store via other registers
 - SS: for load/store via SP and BP
 - ES: another data segment, destination for string operations,
- Address translation: $pa = va + seg * 16$

Segmentation in x86

- Segment registers hold "segment selector" (in real mode)



Segmentation in x86

- Protection through segmentation (in protected mode)

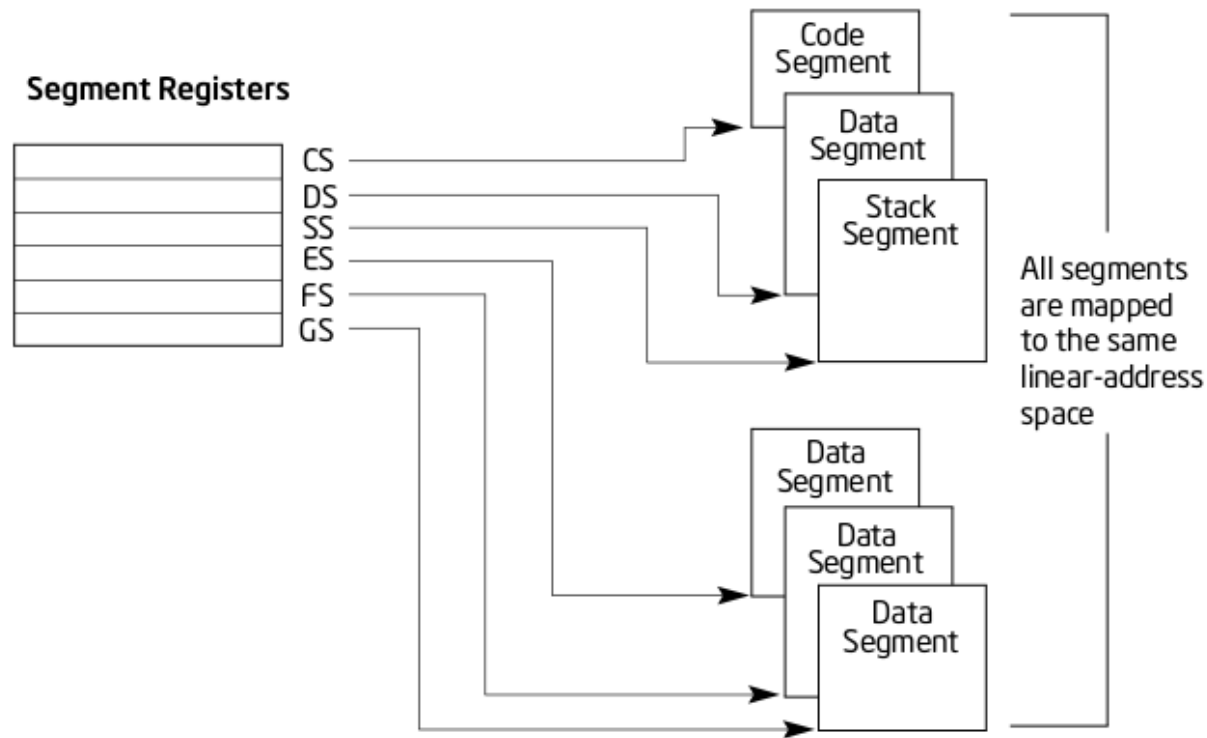
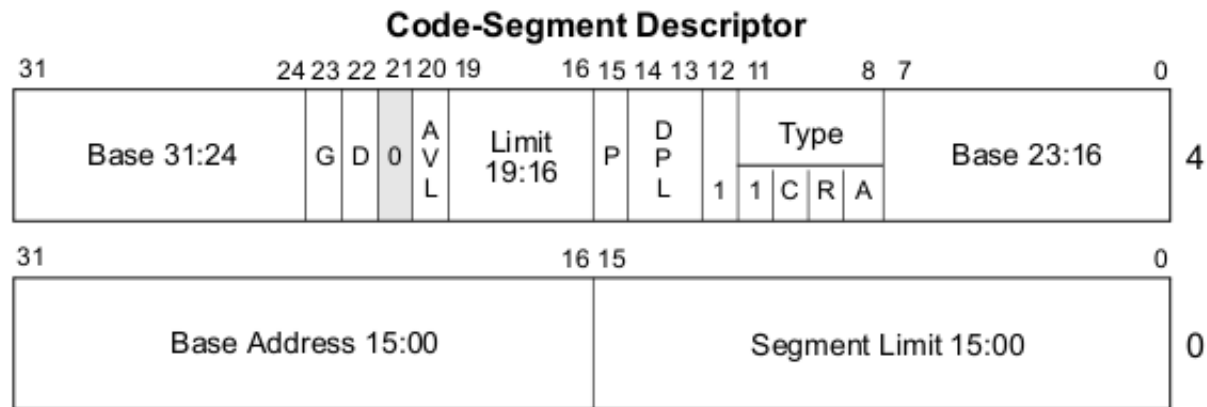



Figure 3-7. Use of Segment Registers in Segmented Memory Model

Segment descriptor

- e.g., 0x009b -> 0000 0000 1001 1011 (CS in QEMU)



A	Accessed	E	Expansion Direction
AVL	Available to Sys. Programmers	G	Granularity
B	Big	R	Readable
C	Conforming	LIMIT	Segment Limit
D	Default	W	Writable
DPL	Descriptor Privilege Level	P	Present
	Reserved		

Flat memory through segmentation

- jos, xv6: relies on paging for protection (later)

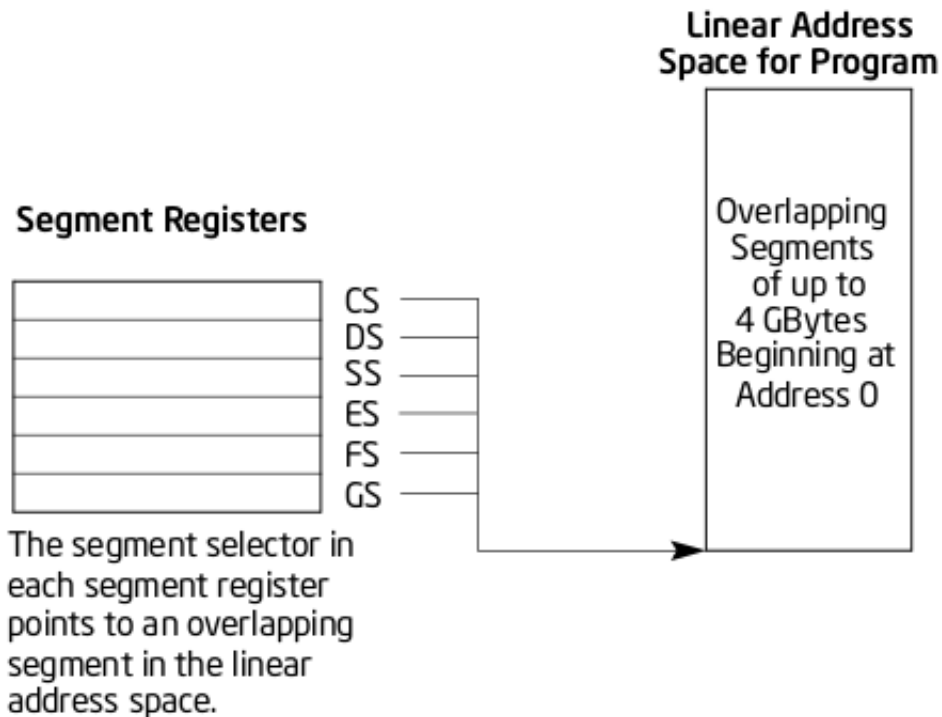


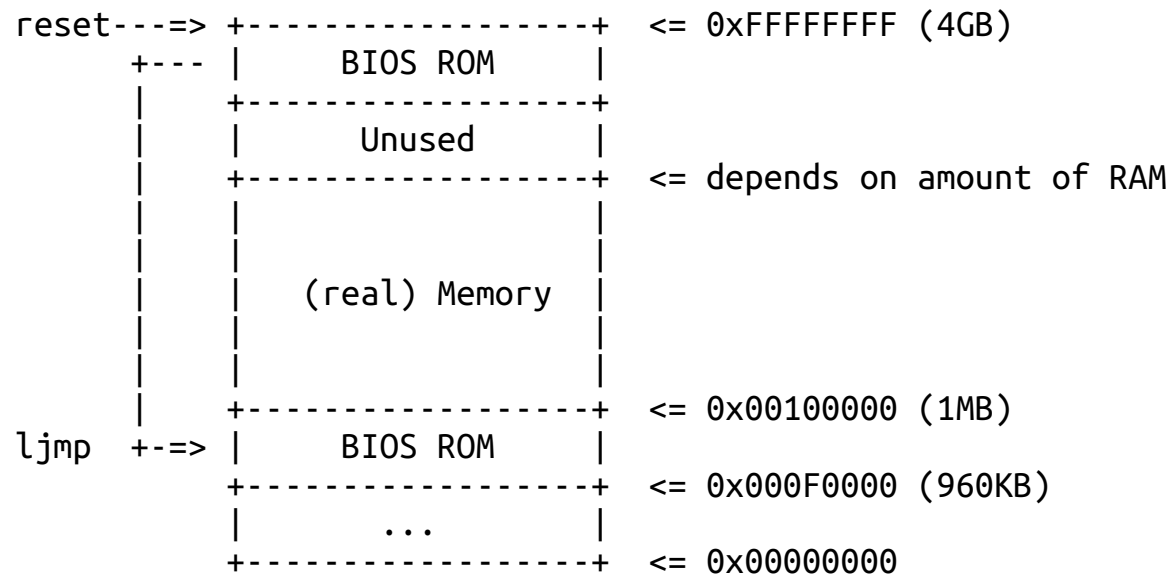
Figure 3-6. Use of Segment Registers for Flat Memory Model

Example: segmentation on xv6

- CS/DS/ES/SS -> 0x00000000 (base) - 0xffffffff (limit)
 - 0xcf9a: 1100 1111 1001 1010

```
ES =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
CS =0008 00000000 ffffffff 00cf9a00 DPL=0 CS32 [-R-]
SS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
DS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
```

Memory aliasing



The first instruction (again)

- A far jump: CS=0xf000, IP=0xe05b
- Next IP: $0xfe05b = 0xf000 * 16 + 0xe05b$ (real mode)

```
(gdb) x/1i 0xfffffffff0
0xfffffffff0:  ljmp    $0xf000,$0xe05b
```

SeaBIOS: reset vector

- **SeaBIOS**: an open source implementation of x86 BIOS
- `src/romlayout.S`

```
reset_vector:  
    ljmpw $SEG_BIOS, $entry_post
```

BIOS -> Bootloader

1. System startup (e.g., I2C to RAM)
2. Power-On Self-Test (POST)
3. **Handover to bootloader**
 - Search bootable devices (looking for a signature, 0x55 0xAA)
 - Load the first sector (MBR, 512B) -> 0x0000:0x7c00 (32KB - 1KB)
 - DEMO: where/how SeaBIOS loads the MBR?

Why not loading the operating system?

- Hard to fit the entire OS in 512 bytes
- More options for users: boot from cdrom, network, usb, etc.

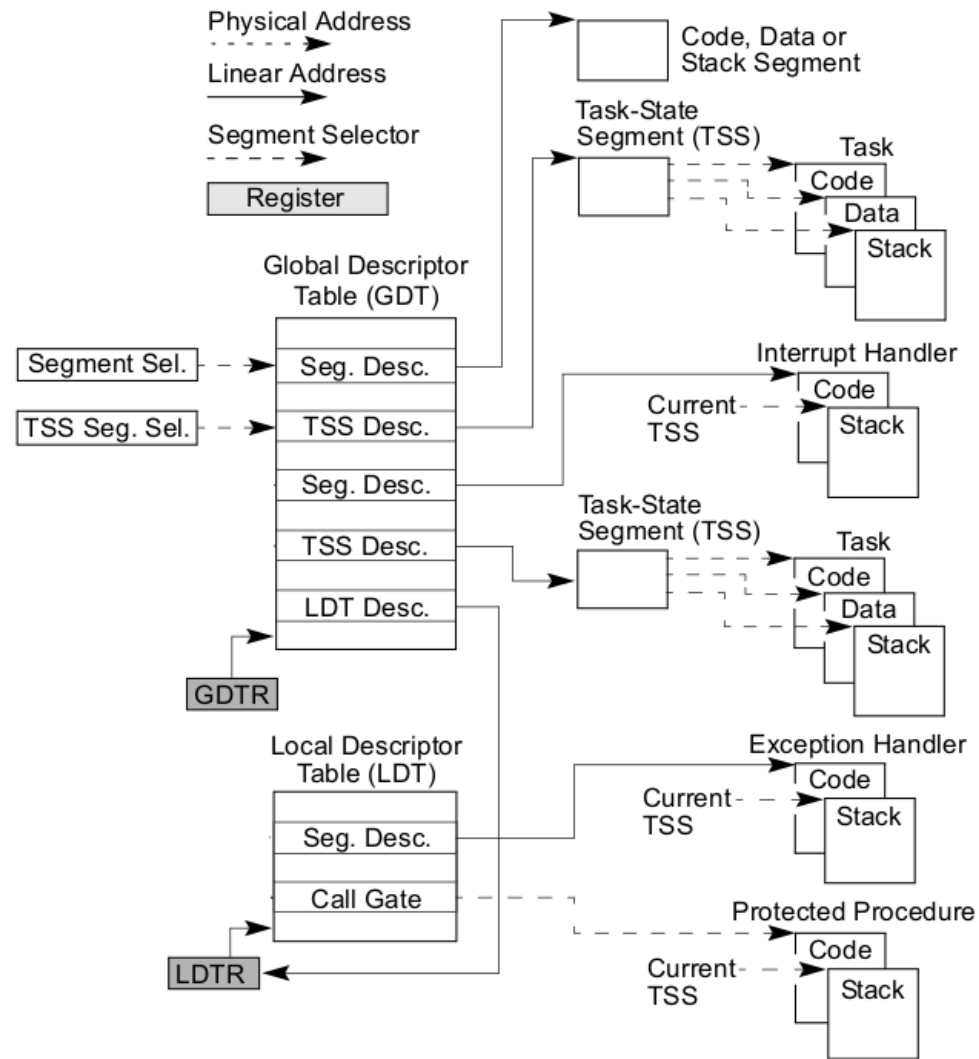
Code review: xv6 bootloader

- `Makefile`
- `bootasm.S`
- `bootmain.c`

About A20 line: a quirk for compatibility

- 8086 -> 1MB (20-bit), 80286 -> 16MB (24-bit)
- In 8086, due to segmentation, it could address over 1MB
 - $0xffff * 16 + 0xffff = 0x10ffef$ (1MB + 64KB - 16B)
 - Wrap around "feature": $0x10ffef \rightarrow 0x00ffef$
- In 20286, latch A20 line for compatibility
 - Using a spare pin in Intel's 8042 keyboard controller
- How to disable (so, enabling A20 line)?
 - Talking to 8042, A20 Gate in IBM PC, or a BIOS call

Segment descriptor tables



I/O: communicating to external world

- Works same as memory accesses, but set I/O signal
- Only 1024 I/O addresses
- Accessed with special instructions (IN, OUT)
- Example: `readsect()` in xv6, `console.c` in jos

I/O: read a sector from a disk (ATA/IDE)

```
void readsect(void *dst, uint offset) {
    waitdisk();
    outb(0x1F2, 1);           // count = 1
    outb(0x1F3, offset);     // set offset
    outb(0x1F4, offset >> 8);
    outb(0x1F5, offset >> 16);
    outb(0x1F6, (offset >> 24) | 0xE0); // master/slave
    outb(0x1F7, 0x20);       // cmd: read sectors

    waitdisk();
    insl(0x1F0, dst, SECTSIZE/4); // read data
}
```

ATA Command (ATA8-ACS)

- Yet another protocol (462 pages)

7.36 READ SECTOR(S) EXT - 24h, PIO data-in

7.36.1 Feature Set

This command is mandatory for devices implementing the 48-bit Address feature set

7.36.2 Description

This command reads from 1 to 65,536 logical sectors as specified in the Count field. The transfer shall begin at the logical sector specified in the LBA field.

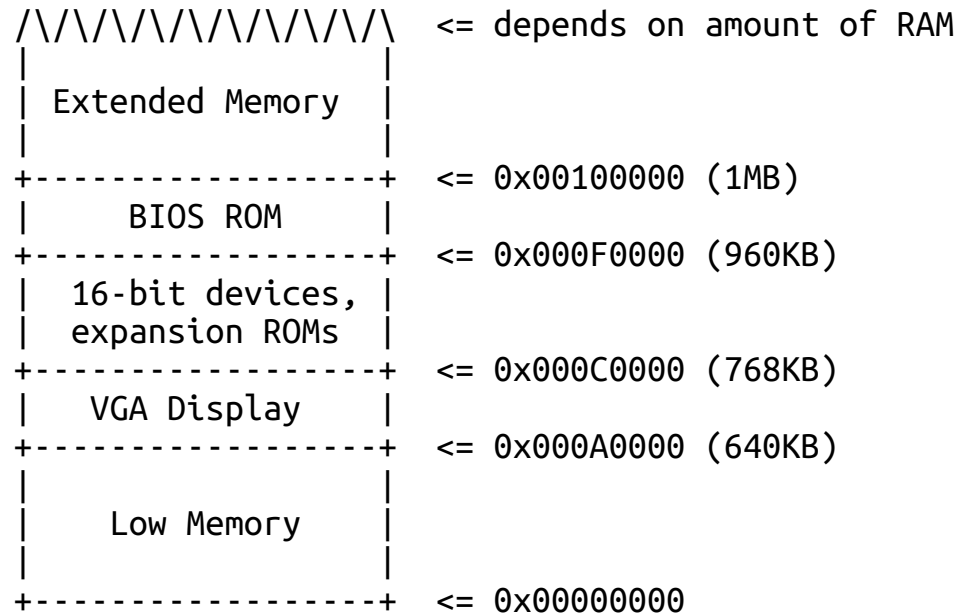
7.36.3 Inputs

Word	Name	Description
00h	Feature	Reserved
01h	Count	The number of logical sectors to be transferred. A value of 0000h indicates that 65,536 logical sectors are to be transferred.
02h	LBA	(MSB)
03h		Address of first logical sector to be transferred.
04h		
05h	Device	Bit Description 15 Obsolete 14 Shall be set to one 13 Obsolete 12 Transport Dependent - See 6.1.9 11:8 Reserved
	Command	7:0 24h

Memory-mapped I/O

- Use normal physical memory addresses
 - Gets around limited size of I/O address space
 - No need for special instructions (i.e., no 1024 limitation)
 - System controller routes to appropriate device
- Works like "magic" memory:
 - Addressed and accessed like memory
 - Reads and writes can have "side effects"

Physical memory layout (again)



Exercise: booting xv6

```
$ git clone git@gatech.github.edu:cs3210-spring2017/cs3210-pub
```

or

```
$ cd cs3210-pub  
$ git pull
```

References

- Intel Manual
- UW CSE 451
- OSPP
- MIT 6.828
- Wikipedia
- The Internet