Booting

Backward compatibility: x86 segments, descriptor tables
BIOS, Bootloader, Read/write disk sectors, ELF format

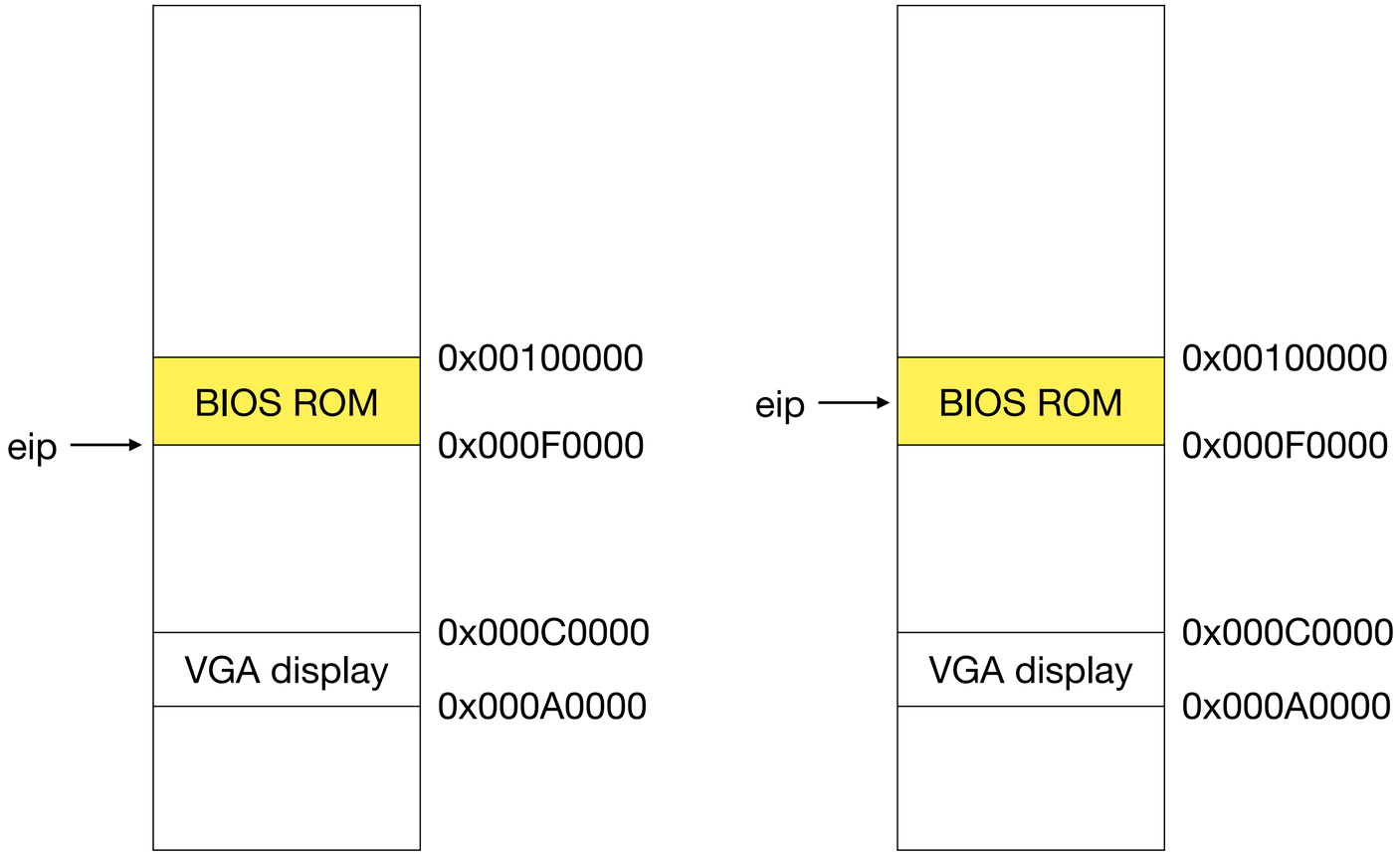
Abhilash Jindal

Reference. xv6 book: Appendix B

Agenda

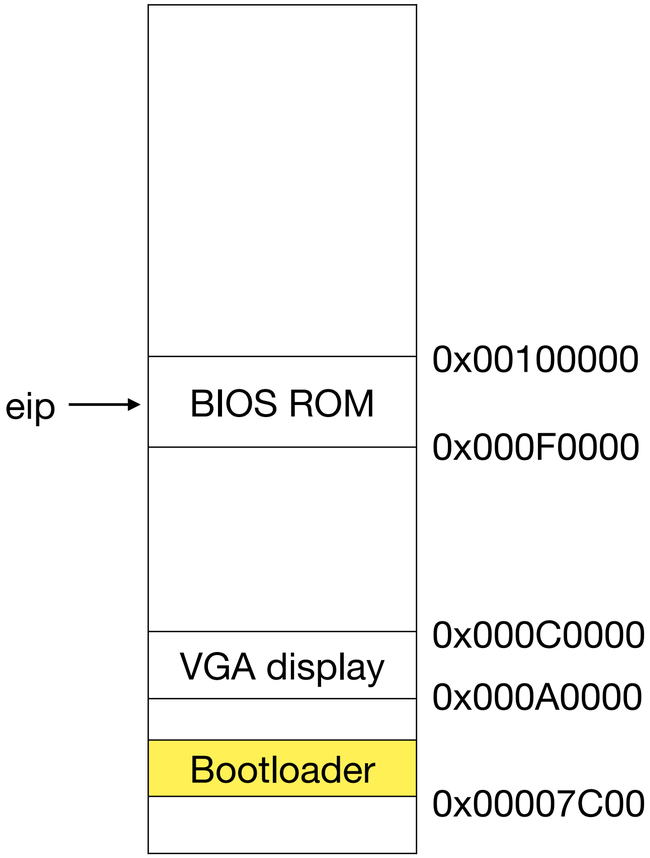
- Boot a minimal OS!
- Understand code
  - Learn what happens when we power on a computer
  - Learn few more x86 details required for booting

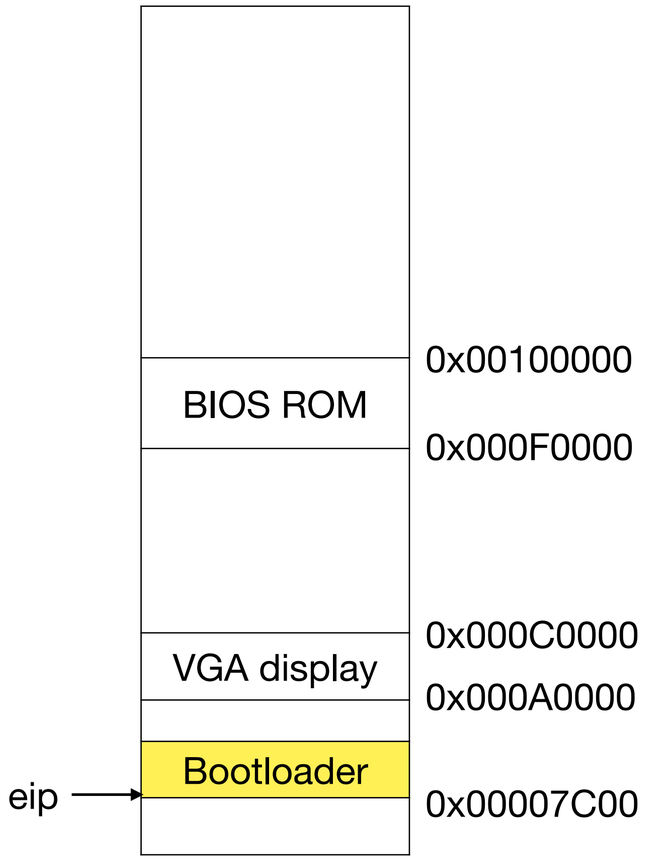
Boot up sequence (1): BIOS



- For backward compatibility, PC boots in 16-bit mode
- BIOS does initial hardware check: are CPUs, memory, disk functional?

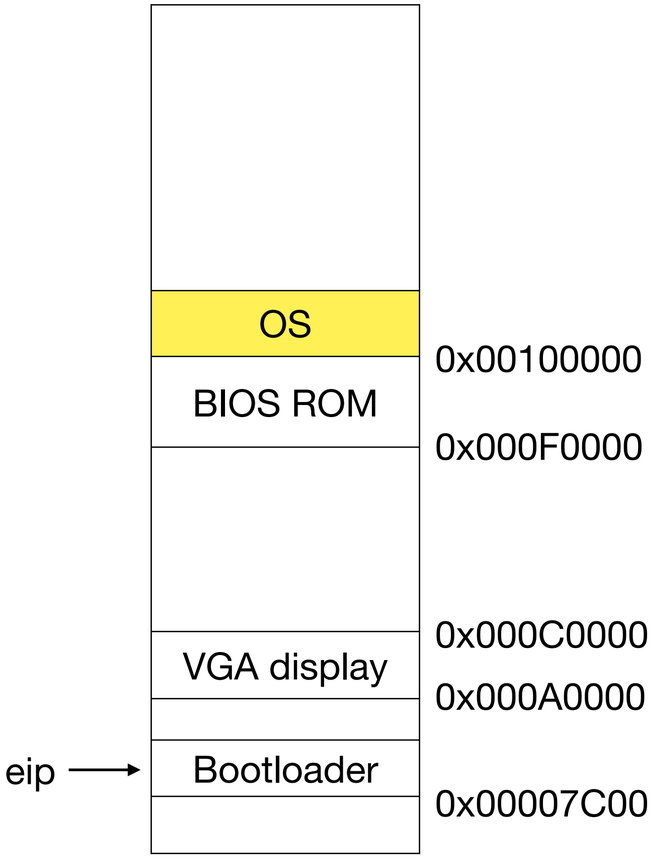
Boot up sequence (2): Bootloader

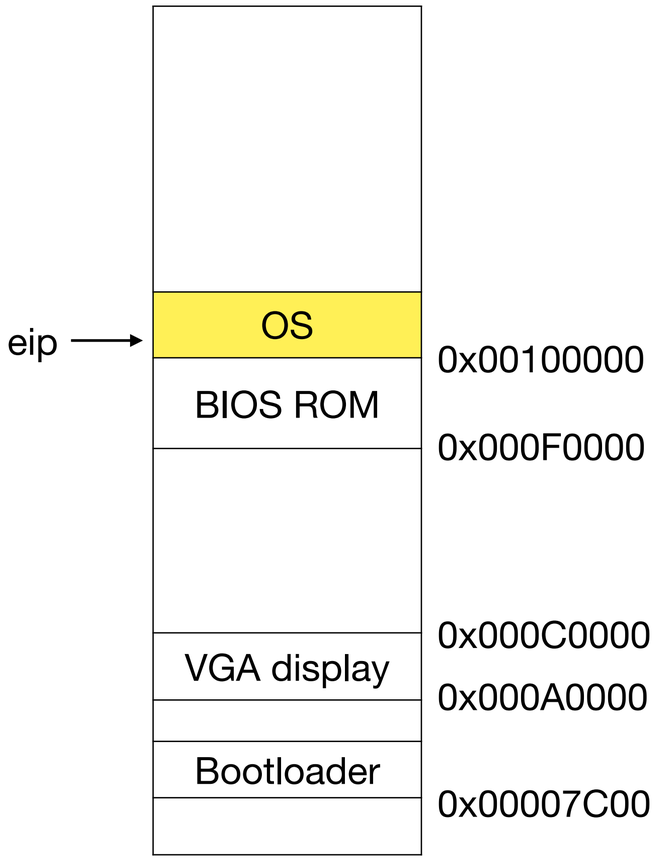




- BIOS loads first disk sector (512 bytes) at 0x7c00 and gives control
  - We need to write a boot loader that fits in the first sector of disk
- Boot loader changes to 32 bit mode

Boot up sequence (3): OS





- gcc prepares OS image in Executable and Linkable Format (ELF)
- Bootloader copies OS image starting from disk sector 1 to 0x100000 and transfers control to it
- We need to tell gcc that image will be loaded at 0x100000

Backward compatibility

- When boot loader gets control, the CPU is in *16-bit mode*
  - This is for backward compatibility. OS written for 16-bit mode should just work for 32-bit and 64-bit machines
- Bootloader explicitly switches from 16-bit mode to 32-bit mode
- Understand how hardware provides backward compatibility and some historical details of 16-bit architecture

16-bit registers

- All registers were 16-bit on 16-bit CPU
- movw %ax, %bx : move low 16 bits of %eax into 16 bits of %ebx
- movb %al, %bl : move low 8 bits of %eax into 8 bits of %ebx
- When CPU is in 16-bit mode, 32-bit and 64-bit machines continue to support same opcode for these instructions

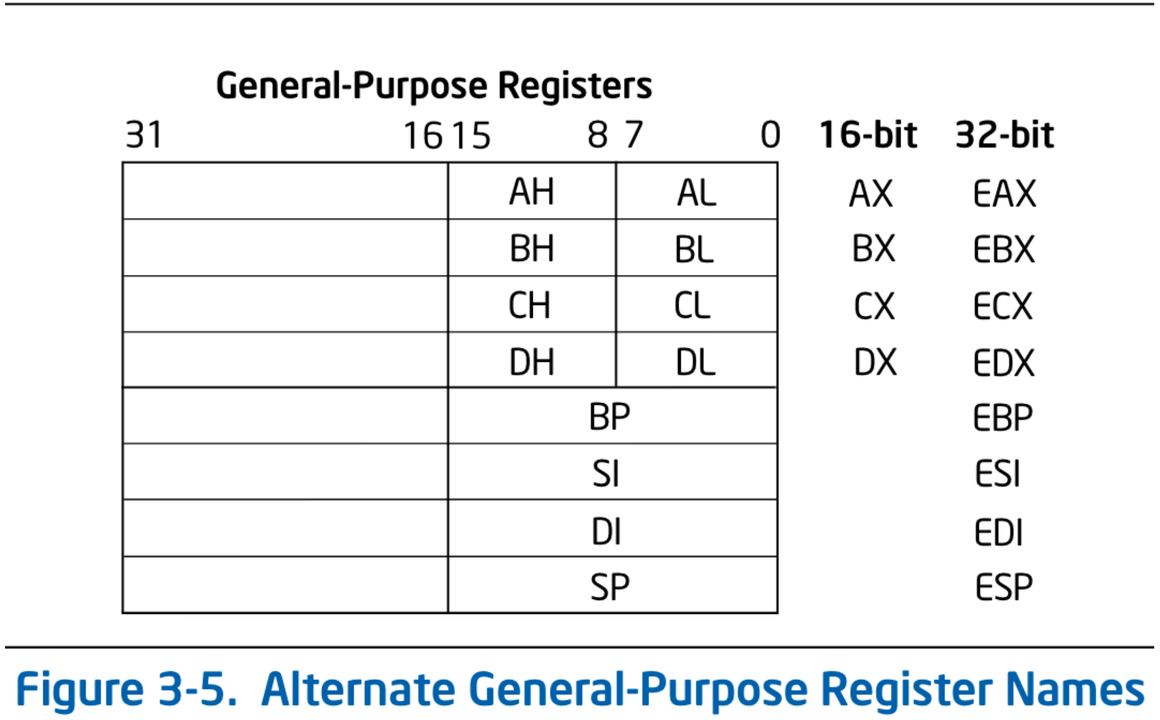
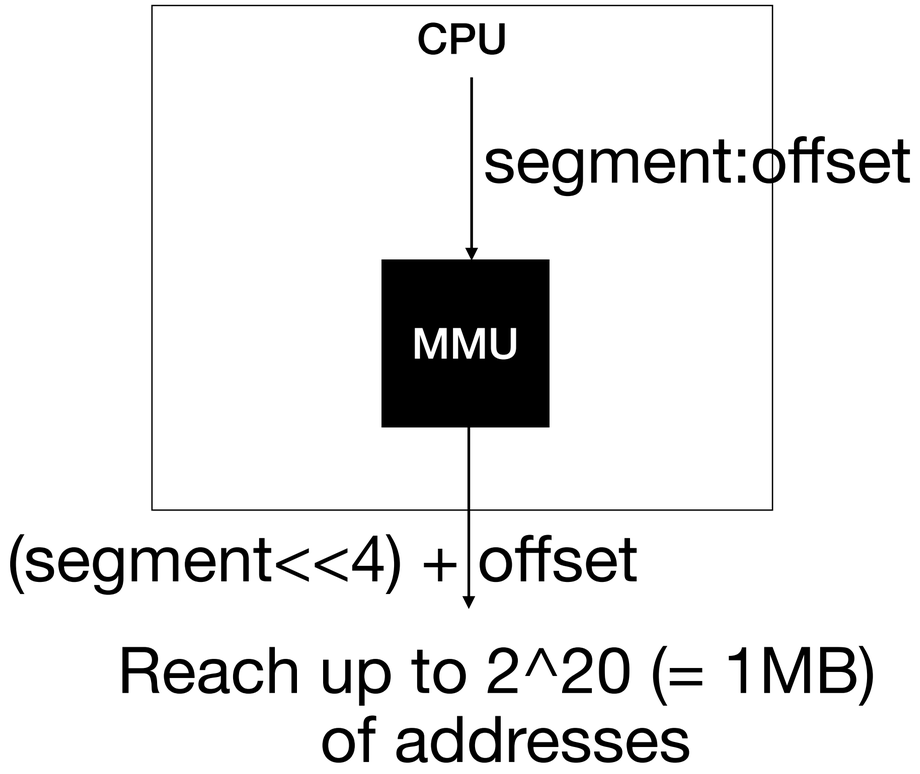


Figure 3-5. Alternate General-Purpose Register Names

Segment registers

- 16-bit registers can only point up to 2^16 (=64KB) addresses in DRAM
- Full address = (segment registers : offset)
  - code segment (cs): ip
  - stack segment (ss): sp / bp.
    push, pop
  - data segment (ds): ax, bx, cx, dx.
    mov (%bx) %ax
  - extra segment (es): si, di
    movsb



Far pointers in 16-bit x86

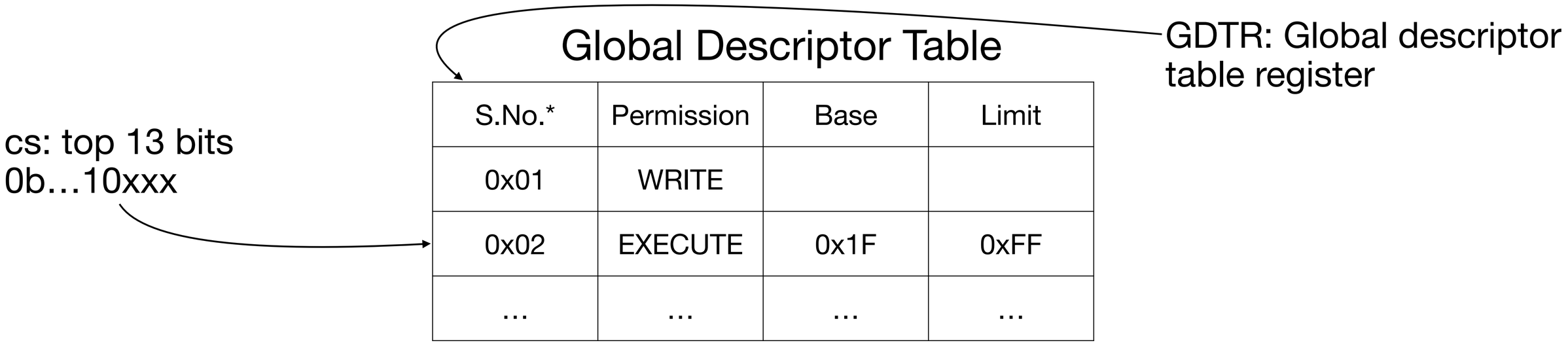
```
#include <stdio.h>
int foo() {
    char far *p =(char far *)0x55550005;
    char far *q =(char far *)0x53332225;
    *p = 80;
    (*p)++;
    printf("%d",*q);
    return 0;
}

Outputs 81
```

- p points to (0x5555 << 4) + 0x0005 = 0x55555
- q points to (0x5333 << 4) + 0x2225 = 0x55555
- Multiple ways of referencing same address making them awkward to control

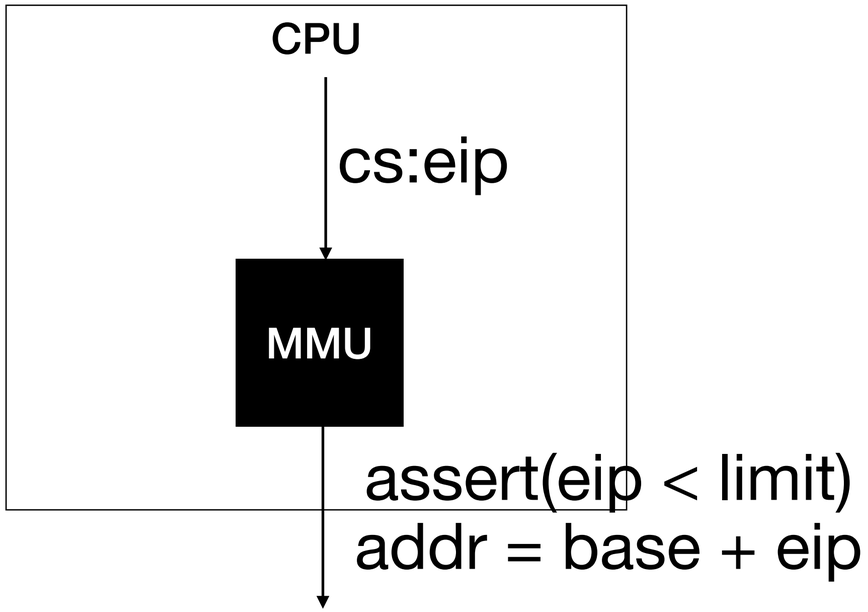
Segment registers in 32-bit

- 32-bit registers can point to 2^32 (=4GB) memory.
- “Protected mode”: extend segment registers for protection



*: S.No. added only for illustration

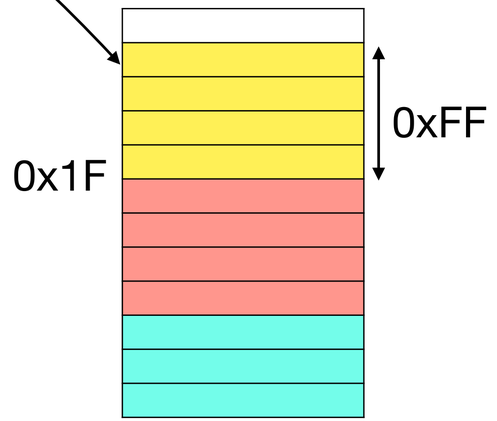
Address translation



Global Descriptor Table

cs: top 13 bits
0b…10xxx

S.No.*	Permission	Base	Limit
0x01	WRITE		
0x02	EXECUTE	0x1F	0xFF
…	…	…	…



- Can “protect” different segments from each other

Global descriptor table

- Upto 2^13 (=8192) segment descriptors
- Segment descriptor:
  - 32 bit base
  - 20 bit limit
  - If G=1, granularity=4KB.
  - Max memory within 1 segment = 2^20*2^12 = 4GB

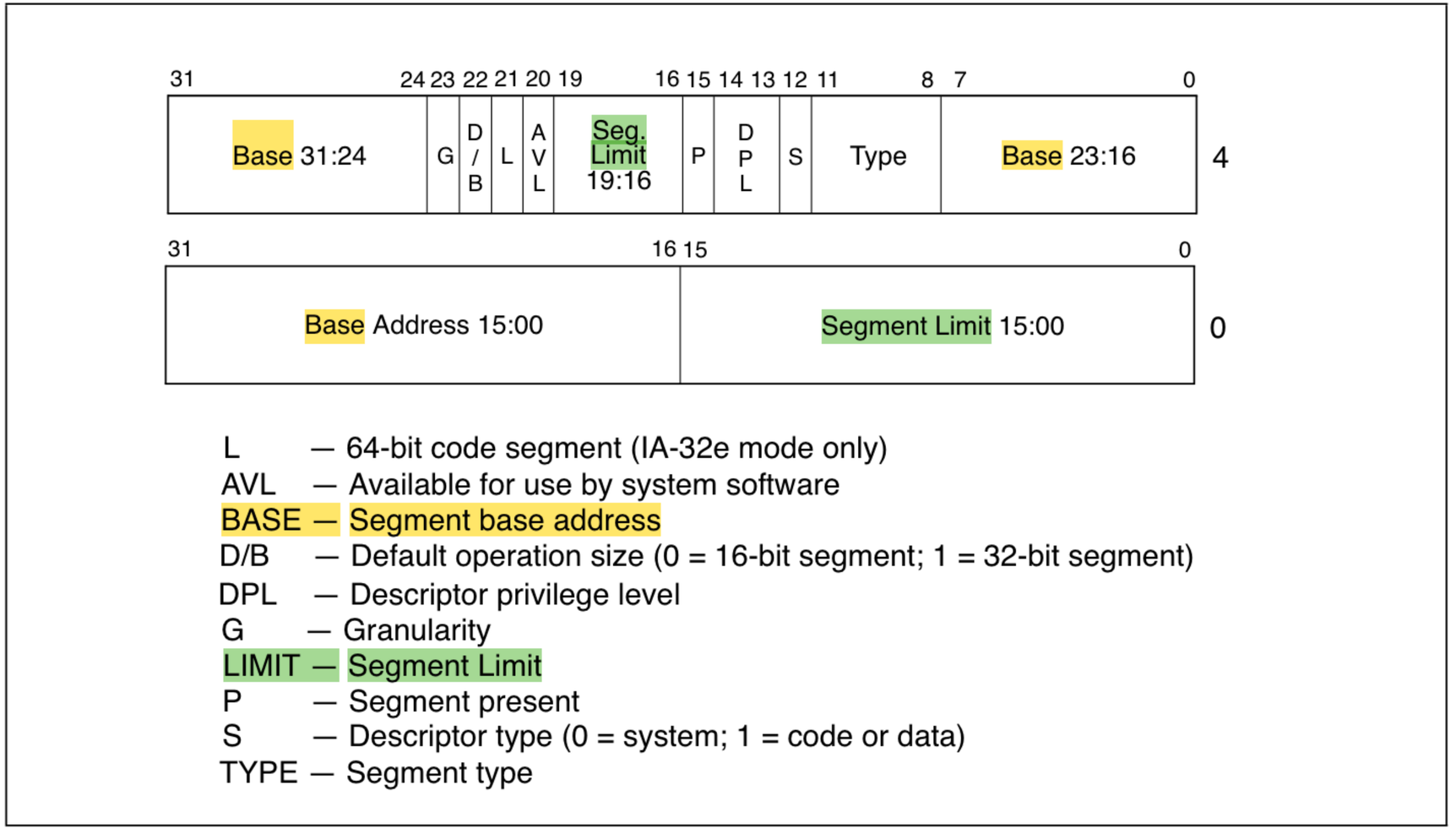


Figure 3-8. Segment Descriptor

Segmented memory model

- Stack cannot grow into code section

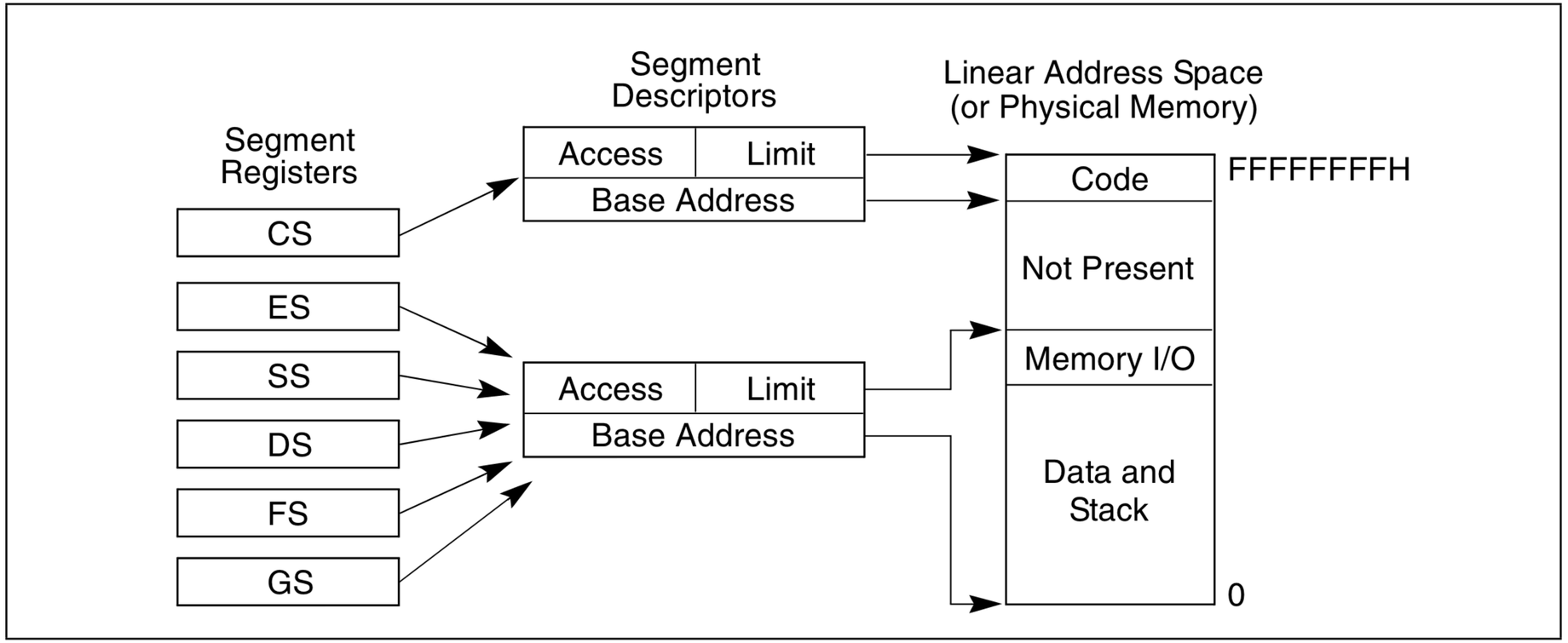


Figure 3-3. Protected Flat Model

Multi-segment model

- Best protection
- Difficult to program

movl %esp %ecx

addl $1 (%ecx)

Does not add 1 to the value at top of the stack!

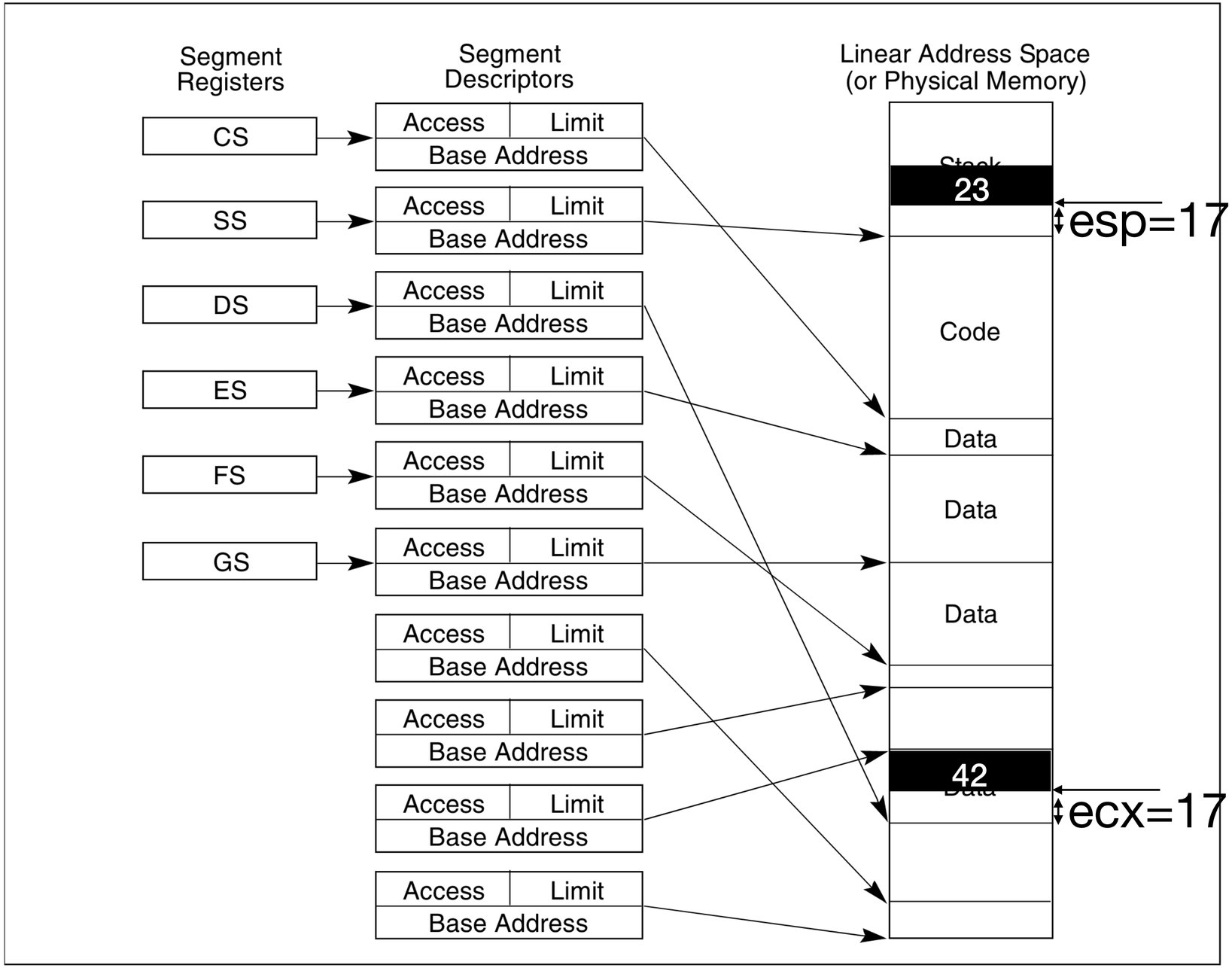


Figure 3-4. Multi-Segment Model

Flat memory model

- Easier to program
- Used by xv6

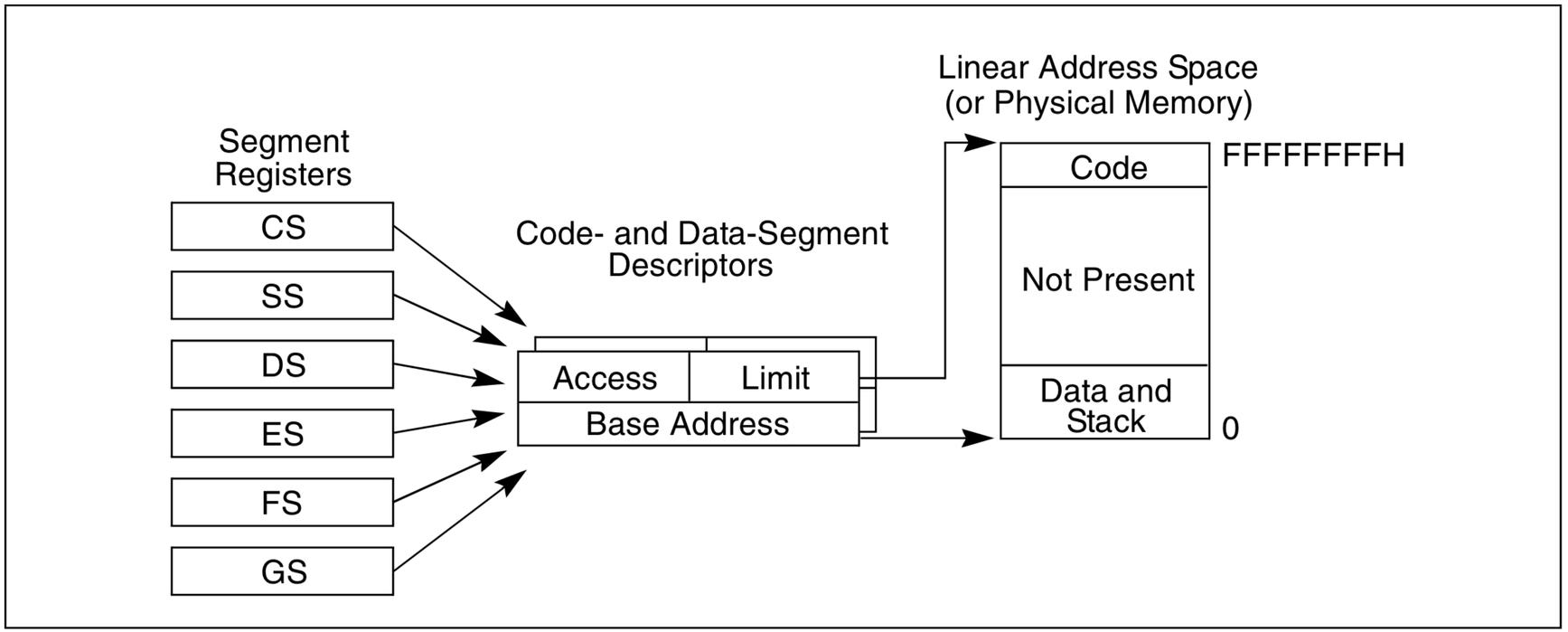


Figure 3-2. Flat Model

Bootloader in action (bootasm.S)

Preparing address offsets

- Makefile
  - bootblock tells linker that I want to start at 0x7c00
  - kernel tells linker that I want to start at 0x100000 in kernel.ld
  - xv6.img copies bootblock at sector 0, kernel starting sector 1

Bootloader in action bootasm.S

- Starts in 16 bit
- Tell assembler that we are in 16 bit mode
- BIOS may have interrupts enabled, setup its own segment registers
  - Clear interrupt flag
  - Clear segment registers
- Switch to 32 bit
- Set up stack pointer to grow below .start (stack grows downward)

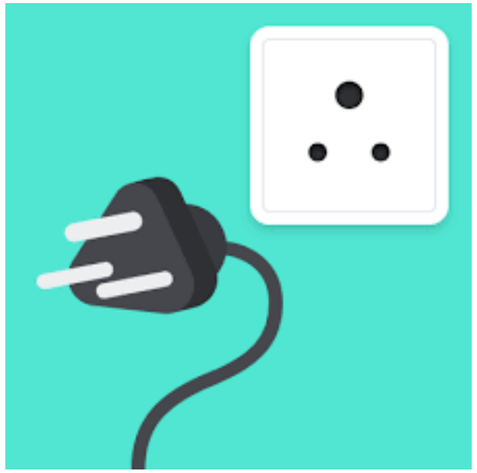
Switching from 16-bit to 32-bit

- Setup GDT with three entries: null entry, code segment entry, and data/stack segment entry
- LGDT to point GDTR to GDT. The operand of the LGDT instruction is a GDT descriptor gdtdesc, which contains the size of the GDT (first two bytes) and the base address of the GDT (next four bytes).
- Set protected mode bit in CR0 register
- Unlike other selectors, code selector cannot be directly set (similar to how eip cannot be set directly).
- Do ljmp to set code selector
- Tell assembler to start assemble assuming 32 bit mode
- Set data segment, extra segment, and stack segment to data segment. These segments are used by default in data access, string instructions, stack instruction respectively.
- Set FS and GS to null segment. They are never default. Programmer can explicitly use FS, GS. xv6 does not want to use them.

Loading OS in Executable and Linkable Format (ELF)

Design principle

Use interfaces, not implementation



Why ELF?

- Port executables from one machine to another[1], one OS to another[2]
- Our OS is just an executable!

1: Same architecture

2. Same system calls

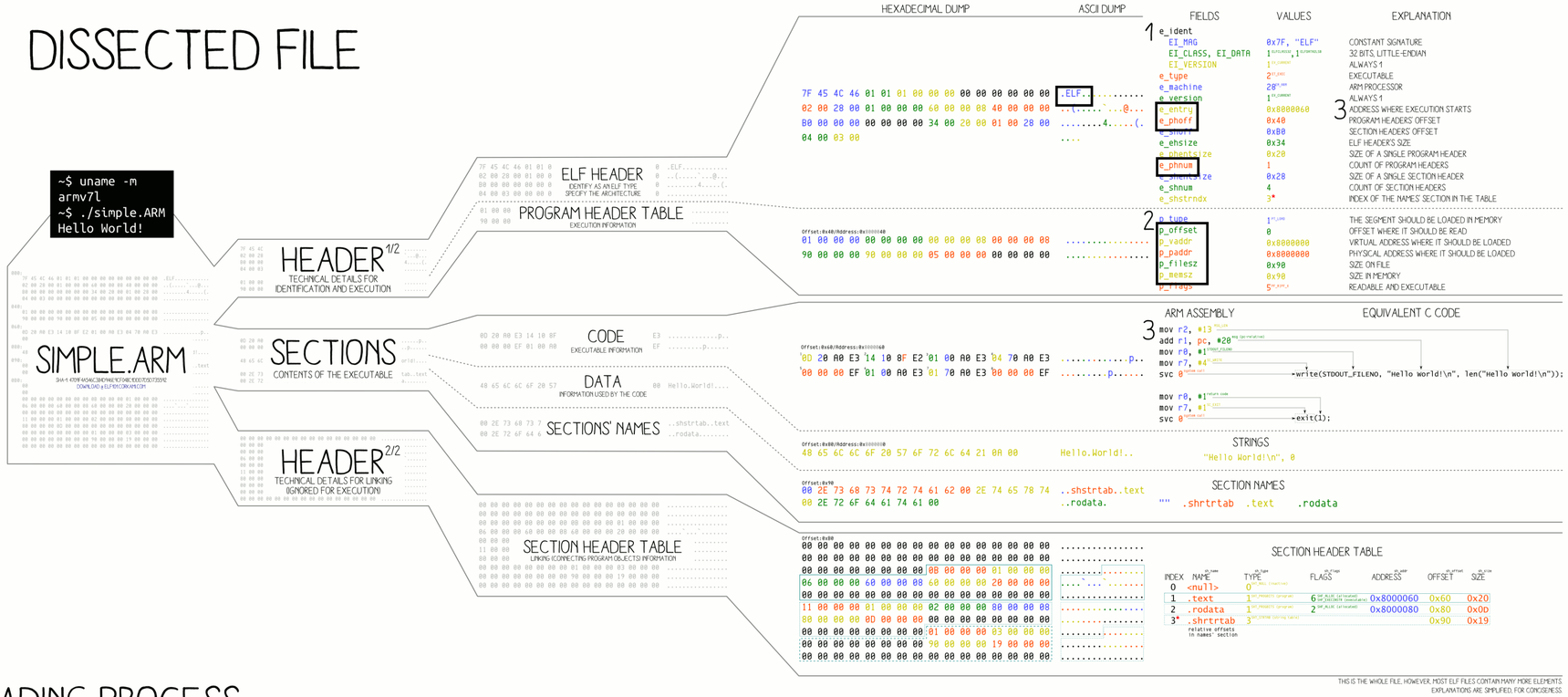
ELF

- ELF header: magic constant, program header offset, count of program headers, entry
- Program header: offset, size in file and in memory, address where to load

ELF[101] a Linux executable walk-through

ANGE ALBERTINI
CORKAMI.COM

DISSECTED FILE



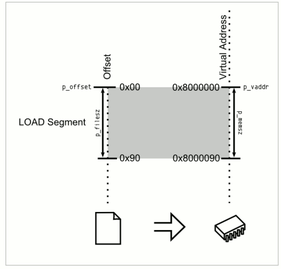
LOADING PROCESS

1 HEADER

THE ELF HEADER IS PARSED
THE PROGRAM HEADER IS PARSED
(SECTIONS ARE NOT USED)

2 MAPPING

THE FILE IS MAPPED IN MEMORY
ACCORDING TO ITS SEGMENT(S)



3 EXECUTION

ENTRY IS CALLED
SYSCALLS[KERNEL SPACE] ARE ACCESSED VIA:
- SYSCALL NUMBER IN THE R7 REGISTER[ARM]
- CALLING INSTRUCTION SVC

TRIVIA

THE ELF WAS FIRST SPECIFIED BY U.S.L. AND U.I. FOR UNIX SYSTEM V, IN 1989

THE ELF IS USED, AMONG OTHERS, IN:
- LINUX, ANDROID, *BSD, SOLARIS, BEOS
- PSP, PLAYSTATION 2-4, DREAMCAST, GAMECUBE, WII
- VARIOUS OSES MADE BY SAMSUNG, ERICSSON, NOKIA,
- MICROCONTROLLERS FROM ATMEL, TEXAS INSTRUMENTS

VERSION 1.0A
2013/12/06

ELF in action

- Open xv6.img in hex editor (vim, :%!xxd)
  - ELF magic word is written at 0x200,00 (byte number 512)
  - At 0x210,08 we have 0x0C0010000 which is little endian for 0x0001000C
  - In kernel.asm, entry is 0x0001000C

ELF in action

(p1-booting)$ readelf -l kernel

Elf file type is EXEC (Executable file)

Entry point **0x10000c**

There are **3** program headers, starting at offset **52**

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	**0x001000**	0x00100000	**0x00100000**	**0x005A**	0x005A	RE	0x1000
LOAD	0x000000	0x00101000	**0x00101000**	**0x0000**	**0x01000**	RW	0x1000
GNU_STACK	0x000000	0x00000000	0x00000000	0x0000	0x00000	RWE	0x10

Section to Segment mapping:

Segment Sections...

00 .text

01 .bss

02

1. Program headers start from 52

2. Read the text region of size 0x5A from file offset 0x1000 and load it at physical address 0x100000

3. bss region (uninitialised global variables) need not be read since file size is zero. But allocate 0x1000 memory at 0x101000.

4. Jump to address 0x10000c to start executing

Loading ELF file

- bootmain calls readseg to read 4KB (ELF file header)
  - readseg adds 1 since kernel starts at sector 1 and reads sectors 1 by 1.
- bootmain checks if magic header is present. If not, something is wrong: the OS image is corrupted, *etc.* abort
- Read program segment header one by one
  - Read ph->filesz bytes from ph->off on the disk into ph->paddr
  - Zero (memsz - filesz) values from pa + ph->filesz
- Finally jump to the entry point as per ELF

Reading from disk

- 0x1F7 is status for read, command for write
- If status is ready and not busy, we can write
  - command (0x20 of read, 0x30 for write)
  - or data (from port 0x1F0)

Register	Directio	Function	Description
0x1F0	R/W	Data register	Read/write data bytes
0x1F2	R/W	Sector count register	Number of sectors to read/write
0x1F3-0x1F6	R/W	Sector number registers	Sector address
0x1F7	R	Status register	Read current status
0x1F7	W	Command register	Send commands to device

Status registers bit	Function
6 (Ready)	Bit is clear when drive is spun down
7 (Busy)	Drive is preparing to send/receive data

Head register bits 0x1F6	Function
4 (Ready)	Select drive number
5, 6, 7	Always set

Command registers	Function
0x20	Read sector
0x30	Write sector

From: https://wiki.osdev.org/ATA_PIO_Mode
https://wiki.osdev.org/ATA_Command_Matrix

Reading from disk readseg

- readseg repeatedly calls readsect
- readsect calls waitdisk which waits for the disk to get ready: busy bit is zero and ready bit is one
    - This approach is called *polling*. It wastes CPU cycles since disks can take a long time (~ms). Bootloader has nothing else to do so it is fine here.
- Then it says that I am reading one sector, gives sector offset, and gives command as read
- When the disk is ready, read out 512 bytes using insl from data port to dst

Understanding insl

```
asm volatile("cld; rep insl" :
             "=D" (addr), "=c" (cnt) :
             "d" (port), "0" (addr), "1" (cnt) :
             "memory", "cc");
```

```
asm volatile("cld; rep insl" :
  7ce8: 8b 7d 08        mov    0x8(%ebp),%edi
  7ceb: b9 80 00 00 00  mov    $0x80,%ecx
  7cf0: ba f0 01 00 00  mov    $0x1f0,%edx
  7cf5: fc              cld
  7cf6: f3 6d           rep insl (%dx),%es:(%edi)
```

```
cld;
I=0;
while(I < ecx) {
     insl %dx, %edi
}
```

- cld: clear direction flag (a bit in EFLAGS): read in forward direction. (std for reads in backward direction)
- “=D” (adds): move addr into edi
- “=c” (cnt): move cnt into ecx
- “d” (port): move port into edx
- memory: instruction will clobber memory
- cc: instruction will clobber condition code (EFLAGS) register
- asm volatile: don’t optimise it away

Our simple OS

- entry.S allocated 4KB space in the bss section with “.comm stack, KSTACKSIZE”
- Then it sets the stack pointer to the top of the allocation (since stack grows downwards) and calls main.c
- main.c sends shutdown signal to QEMU (https://wiki.osdev.org/Shutdown)

Modern systems use two-step booting procedure

- Our Bootloader fits in 512 bytes and assumes there is one OS contiguously stored from sector 1
  - Bootloader boots into a temporary OS
  - Temporary OS can download OS from network, boot from USB, etc. i.e, it is a full-fledged OS with support for reading file systems, TCP/IP to talk to network, etc.