

# **Booting**

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

# Agenda

# Agenda

- Boot a minimal OS!

# Agenda

- Boot a minimal OS!
- Understand code

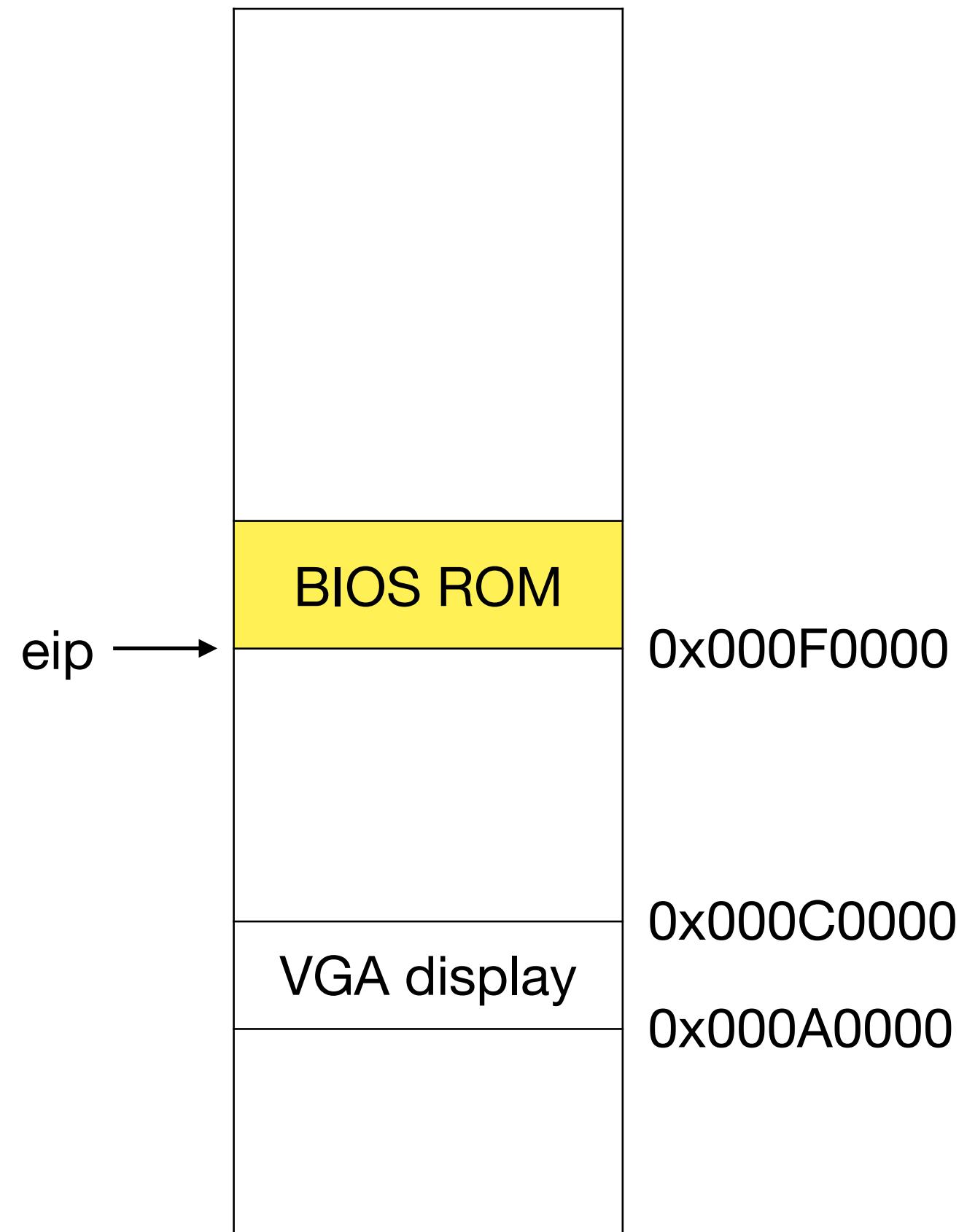
# Agenda

- Boot a minimal OS!
- Understand code
  - Learn what happens when we power on a computer

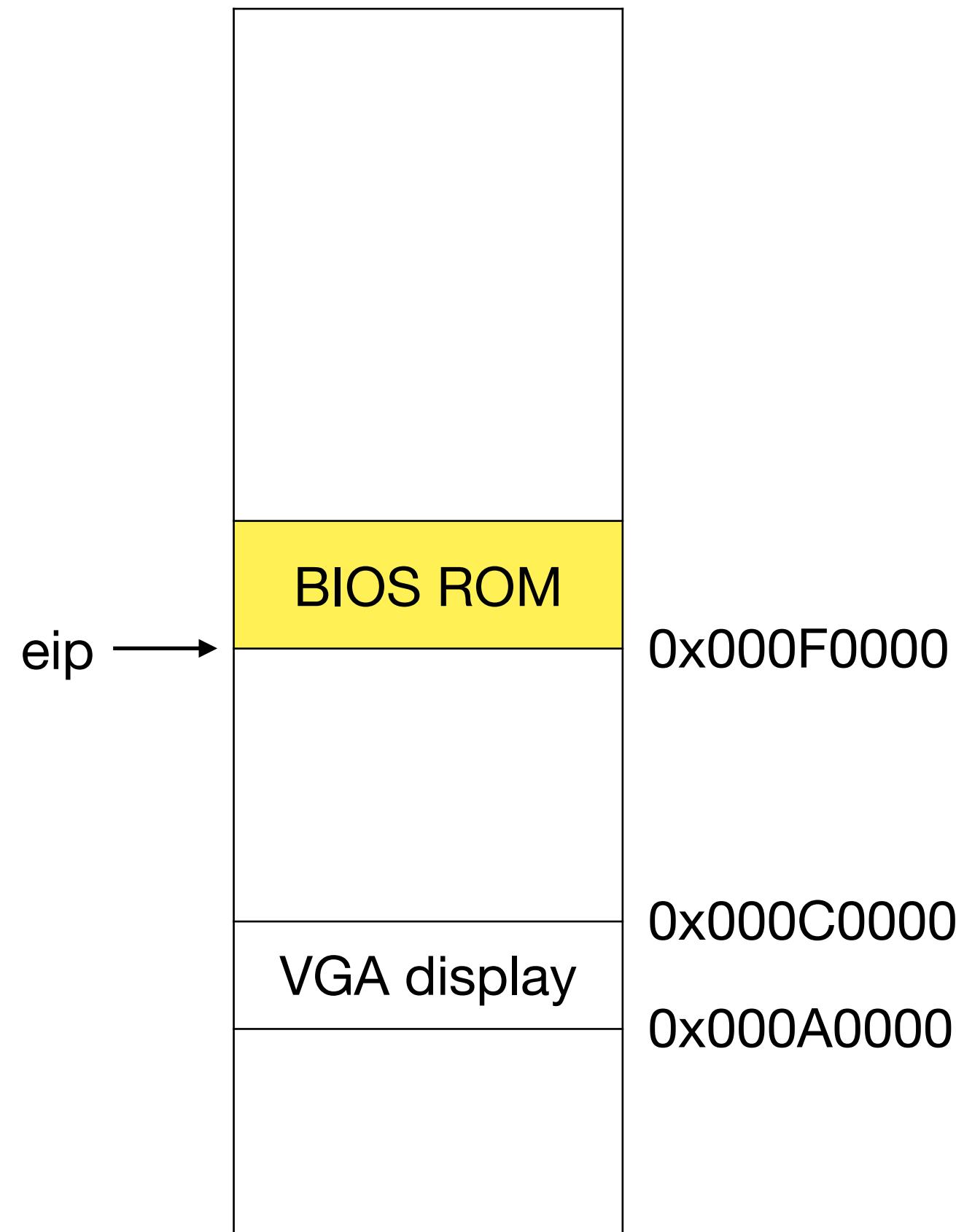
# Agenda

- 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

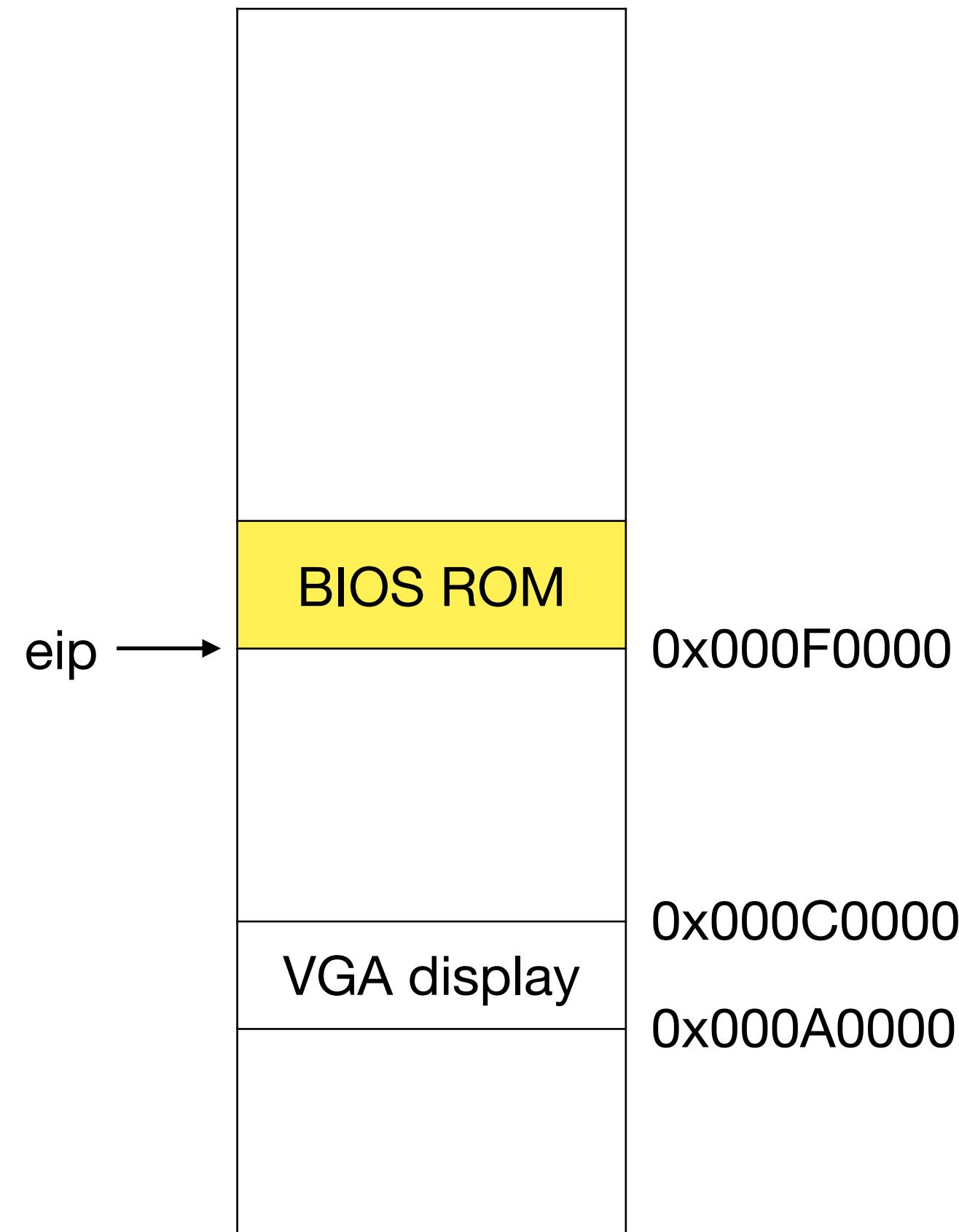


# Boot up sequence (1): BIOS



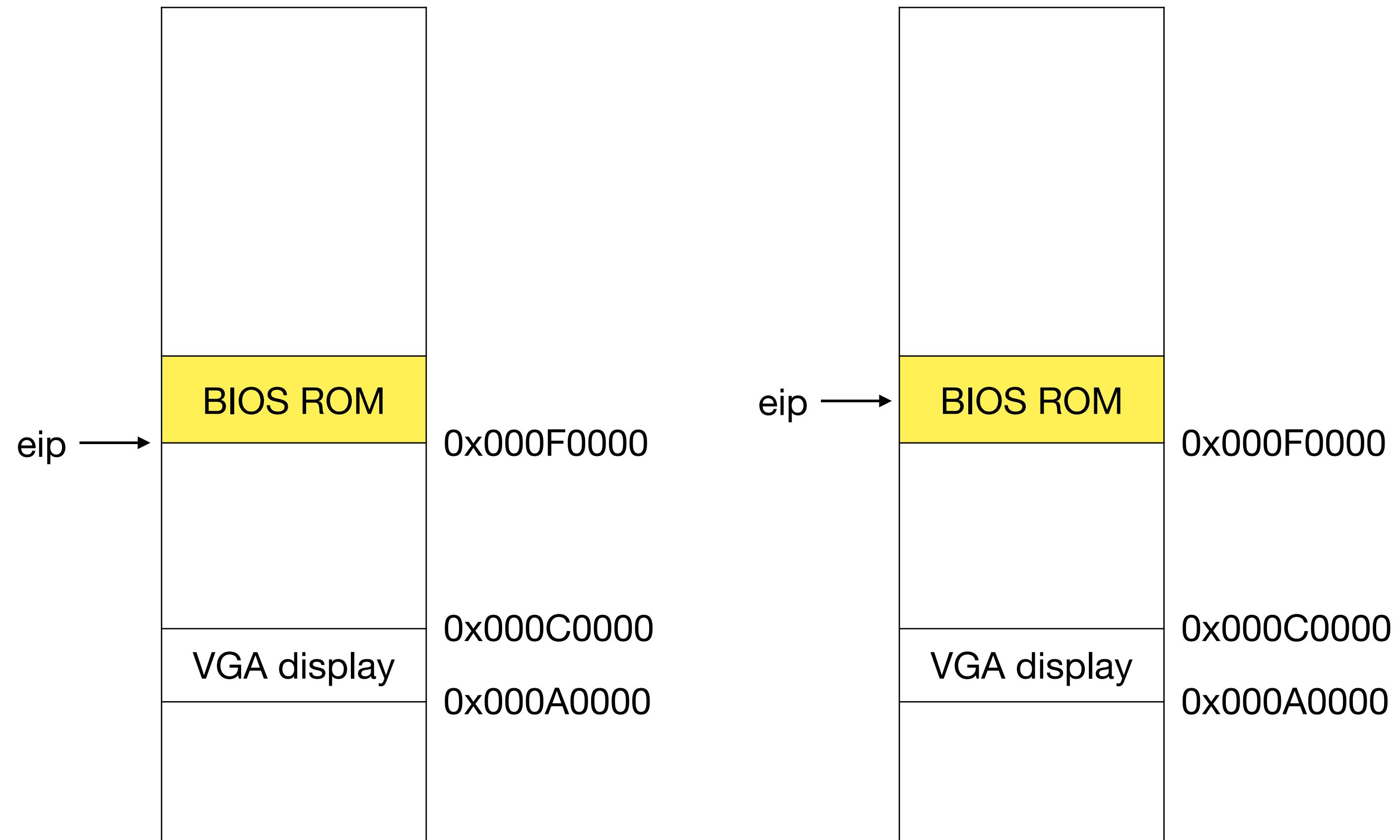
- For backward compatibility, PC boots in 16-bit mode

# Boot up sequence (1): BIOS



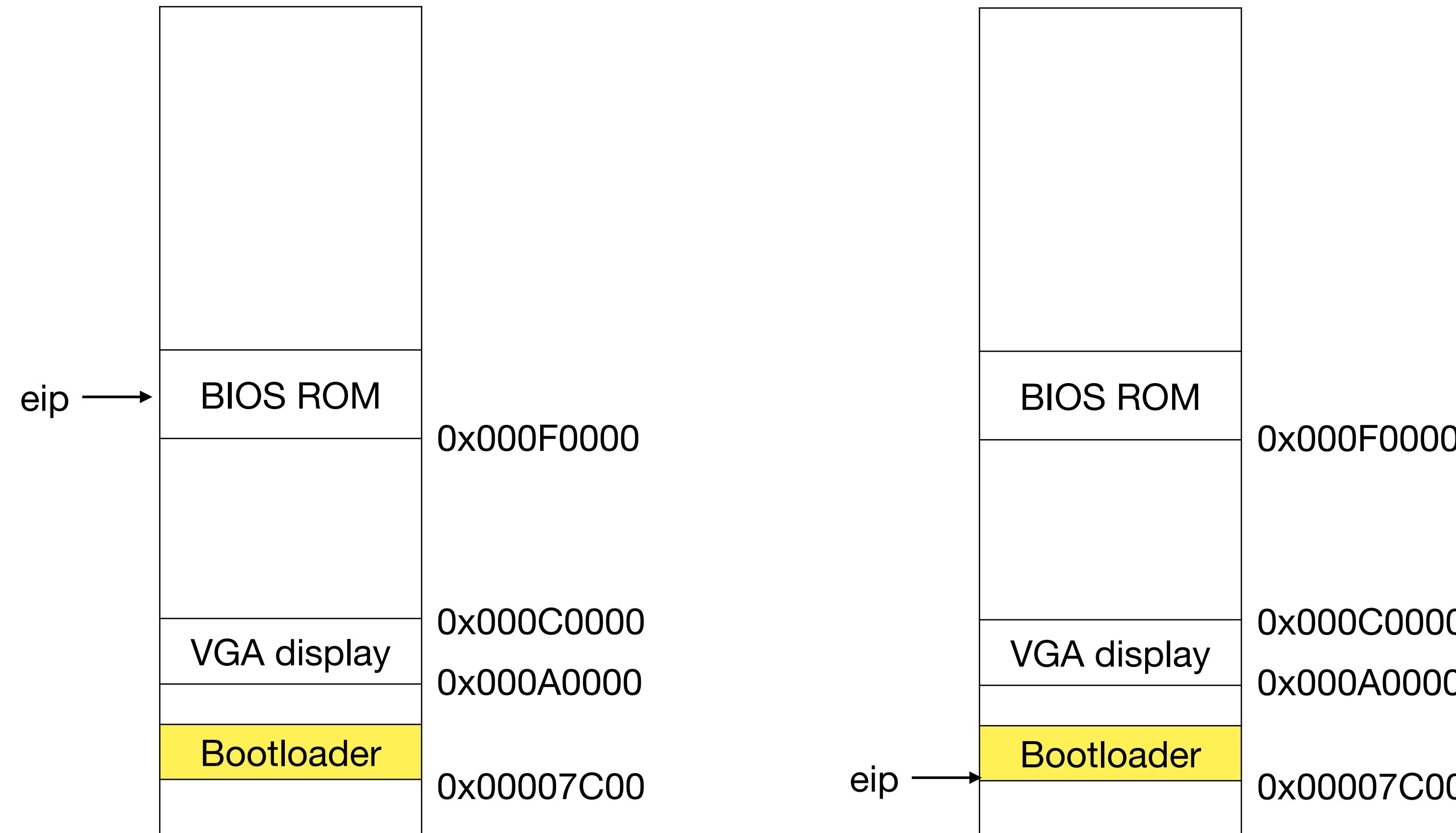
- For backward compatibility, PC boots in 16-bit mode
- BIOS does initial hardware check: are CPUs, memory, disk functional?

# Boot up sequence (1): BIOS



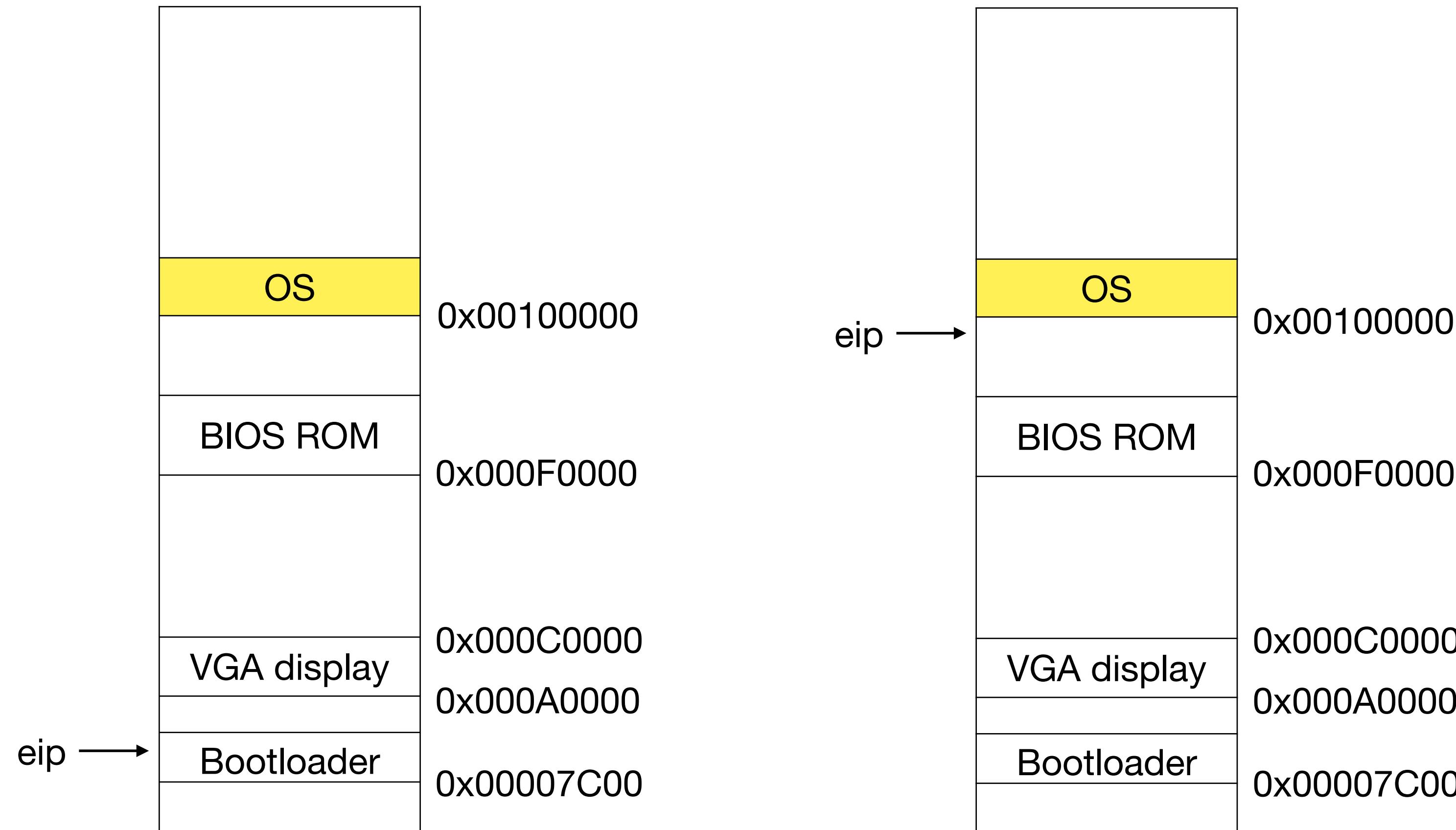
- For backward compatibility, PC boots in 16-bit mode
- BIOS does initial hardware check: are CPUs, memory, disk functional?

# Boot up sequence (2): Bootloader

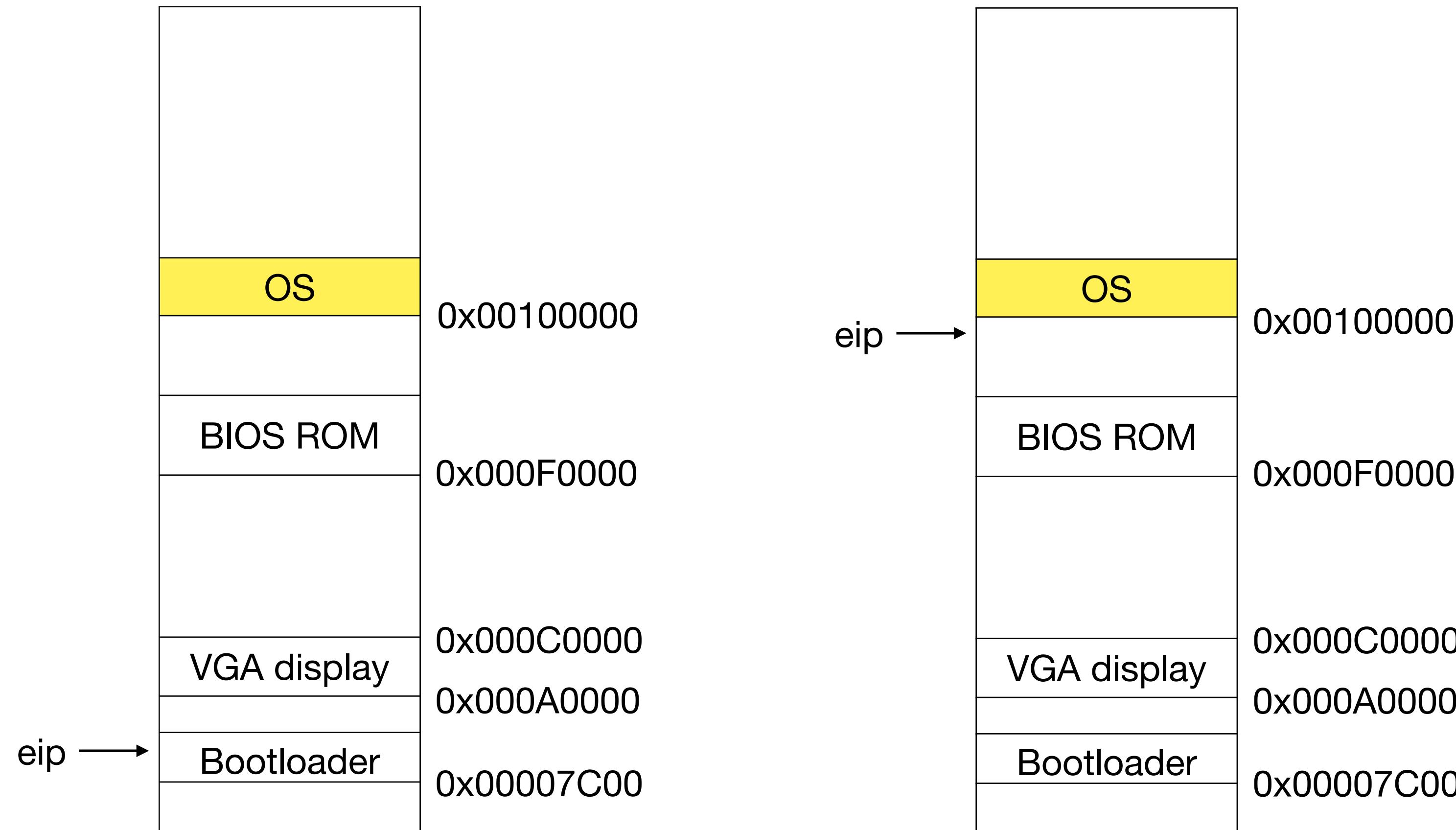


- BIOS loads first disk sector (512 bytes) at 0x7c00 and gives control
- We need to write a boot loader that fits in the first sector of disk
- Boot loader changes to 32 bit mode

# Boot up sequence (3): OS

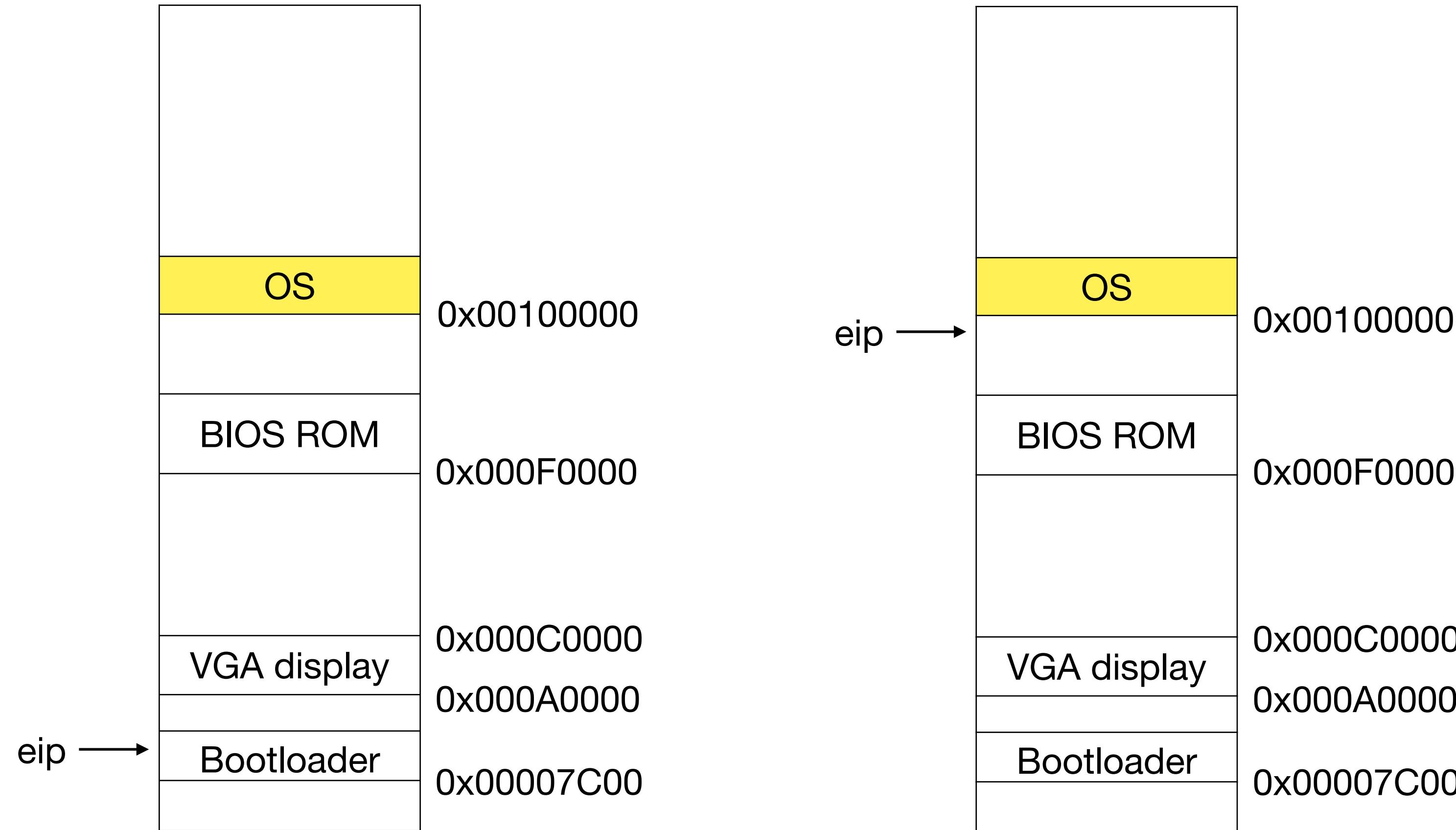


# Boot up sequence (3): OS



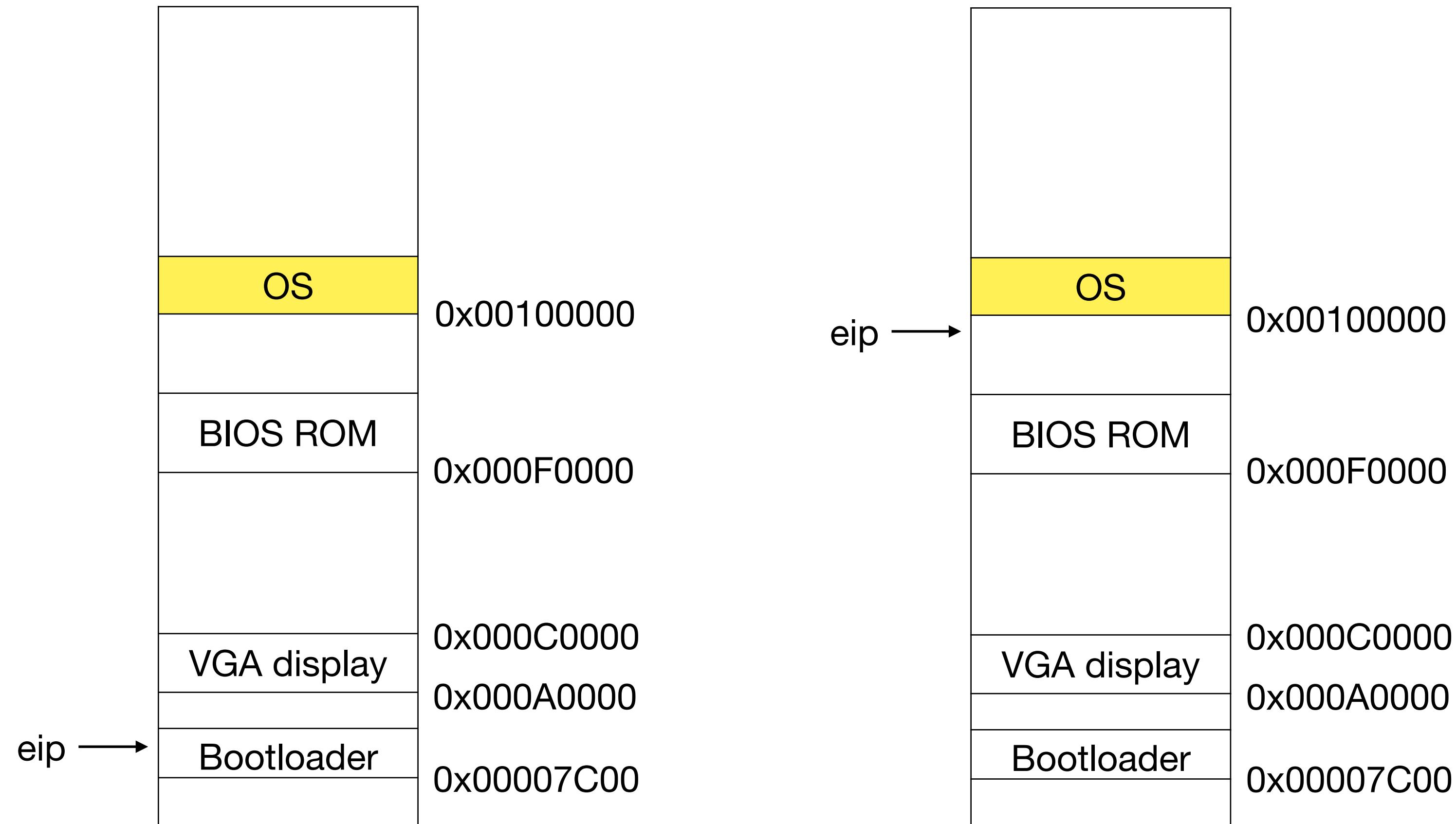
- gcc prepares OS image in Executable and Linkable Format (ELF)

# 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

# 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

# Backward compatibility

- When boot loader gets control, the CPU is in *16-bit mode*

# 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

# 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

# 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

General-Purpose Registers

31	16	15	8	7	0	16-bit	32-bit
			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

# 16-bit registers

- All registers were 16-bit on 16-bit CPU

---

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

# 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`

---

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

# 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`

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

# 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

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

# Segment registers

# Segment registers

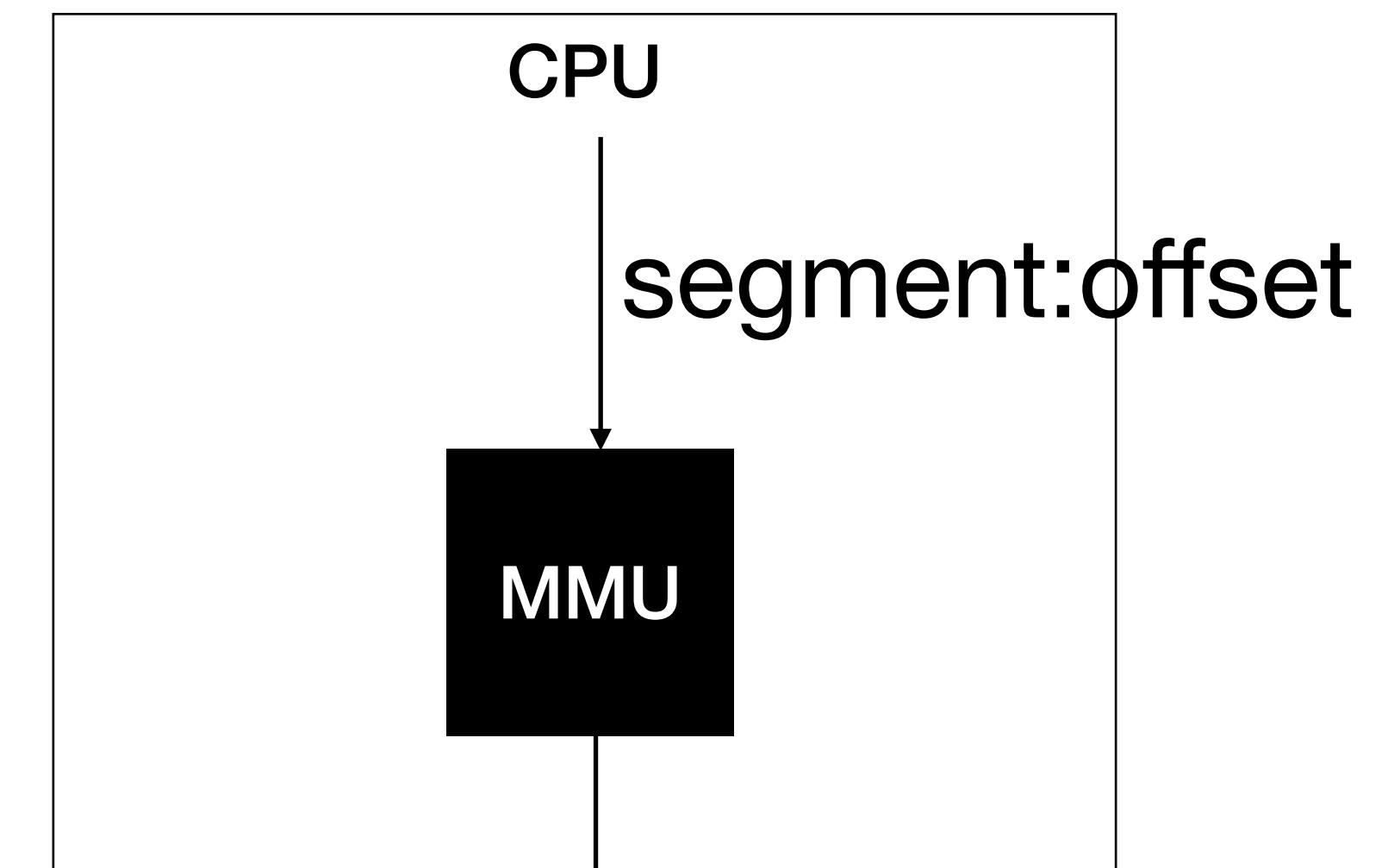
- 16-bit registers can only point up to  $2^{16}$  (=64KB) addresses in DRAM

# Segment registers

- 16-bit registers can only point up to  $2^{16}$  (=64KB) addresses in DRAM
- Full address = (segment registers : offset)

# Segment registers

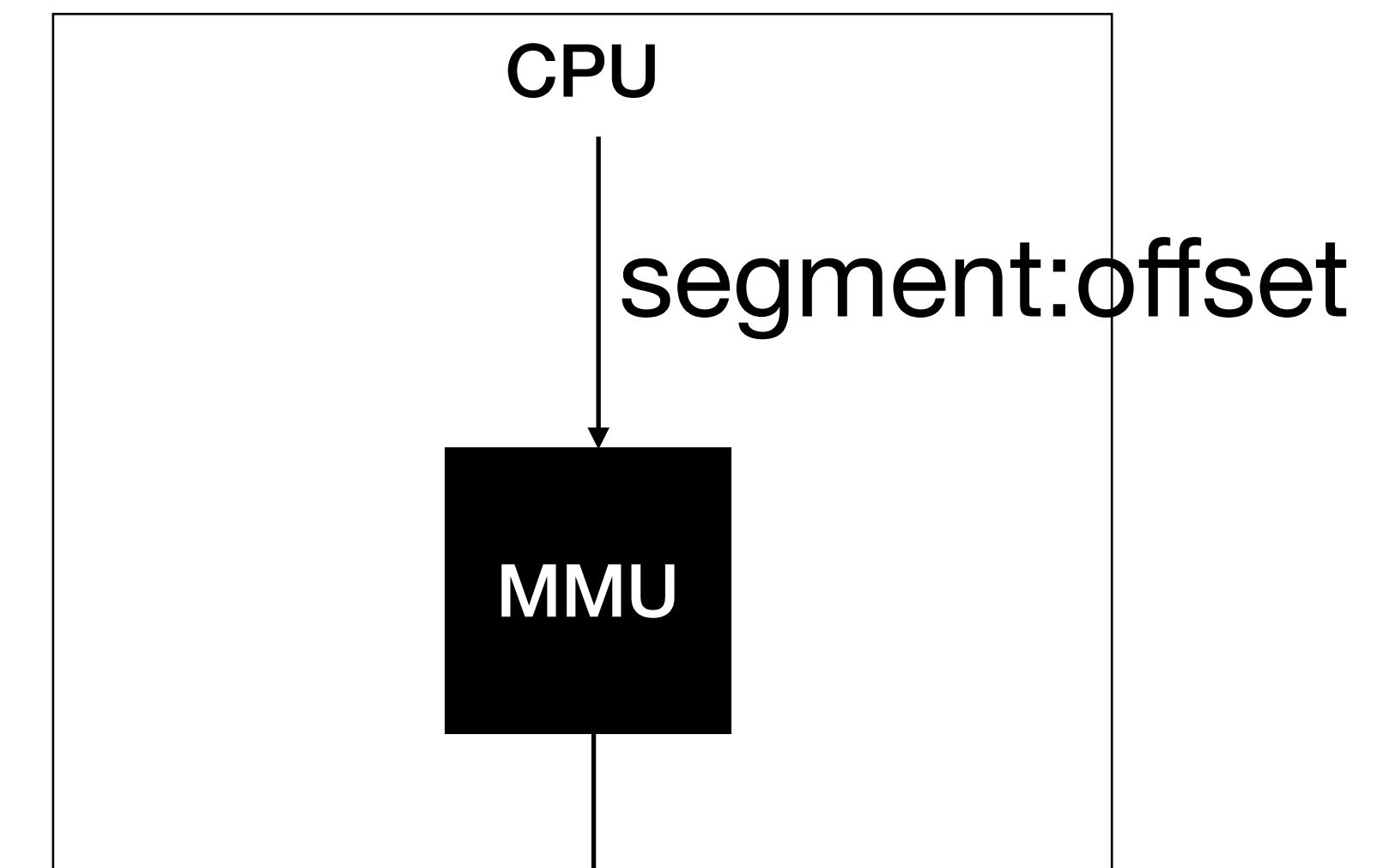
- 16-bit registers can only point up to  $2^{16}$  (=64KB) addresses in DRAM
- Full address = (segment registers : offset)



Reach up to  $2^{20}$  (= 1MB)  
of addresses

# Segment registers

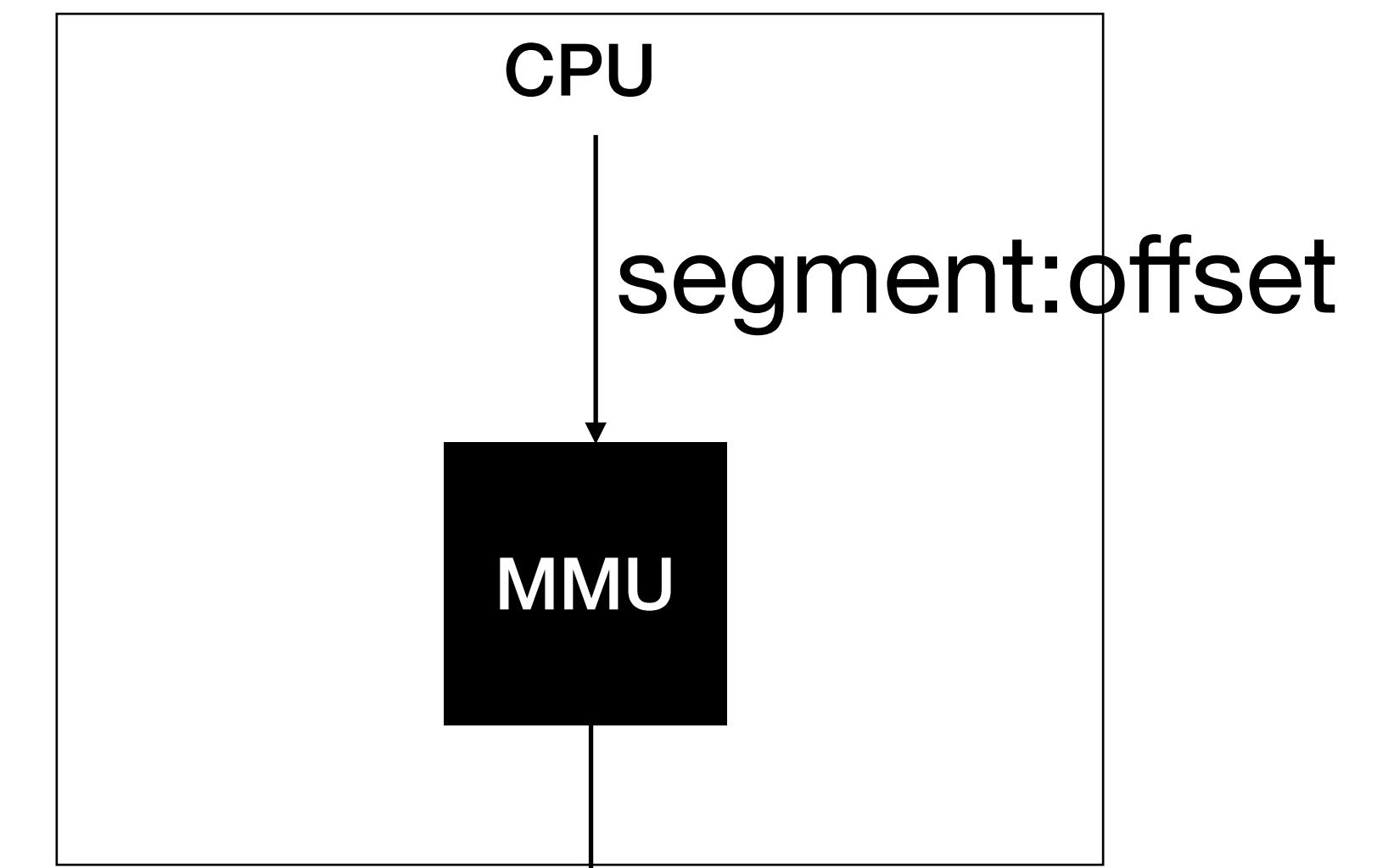
- 16-bit registers can only point up to  $2^{16}$  (=64KB) addresses in DRAM
- Full address = (segment registers : offset)
  - code segment (cs): ip



Reach up to  $2^{20}$  (= 1MB)  
of addresses

# Segment registers

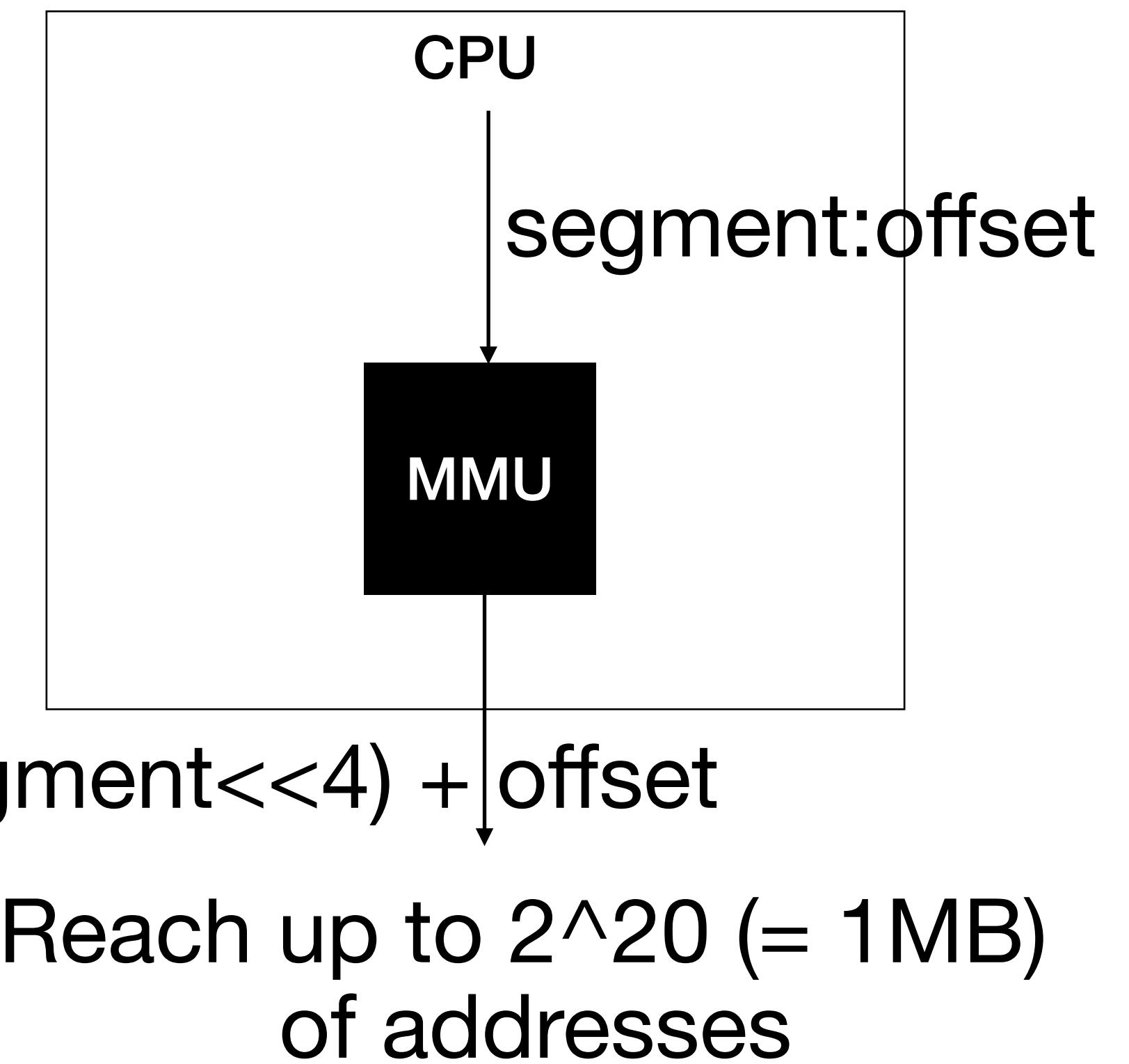
- 16-bit registers can only point up to  $2^{16}$  (=64KB) addresses in DRAM
- Full address = (segment registers : offset)
  - code segment (cs): ip
  - stack segment (ss): sp / bp.  
*push, pop*



Reach up to  $2^{20}$  (= 1MB)  
of addresses

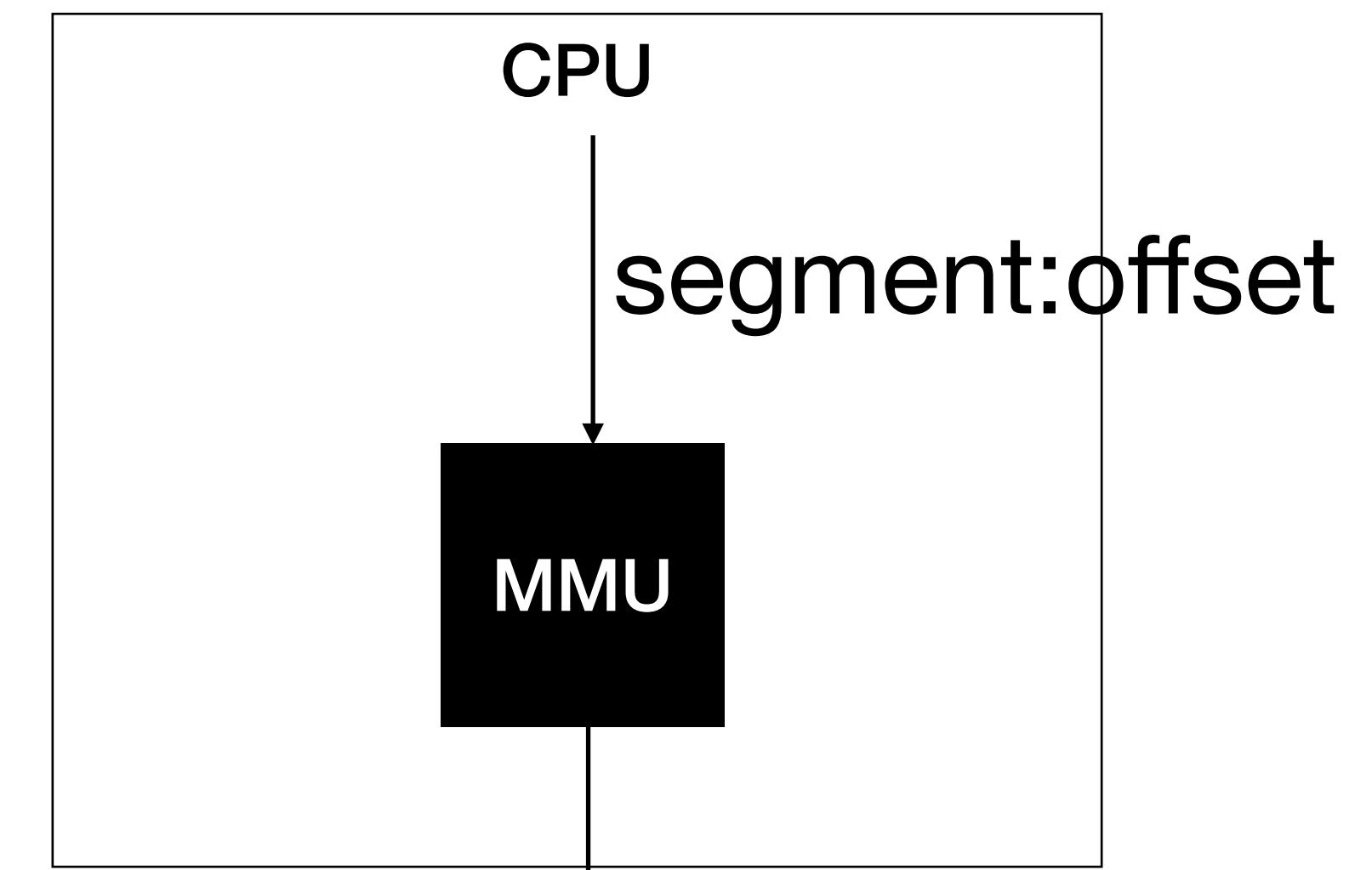
# Segment registers

- 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*



# 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*



Reach up to  $2^{20}$  (= 1MB)  
of addresses

# 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

# 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  $(0x5555 \ll 4) + 0x0005$   
 $= 0x5555$

# 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  $(0x5555 \ll 4) + 0x0005$   
= 0x55555
- q points to  $(0x5333 \ll 4) + 0x2225$   
= 0x55555

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

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

# **Segment registers in 32-bit**

\*: S.No. added only for illustration

# Segment registers in 32-bit

- 32-bit registers can point to  $2^{32}$  (=4GB) memory.

\*: S.No. added only for illustration

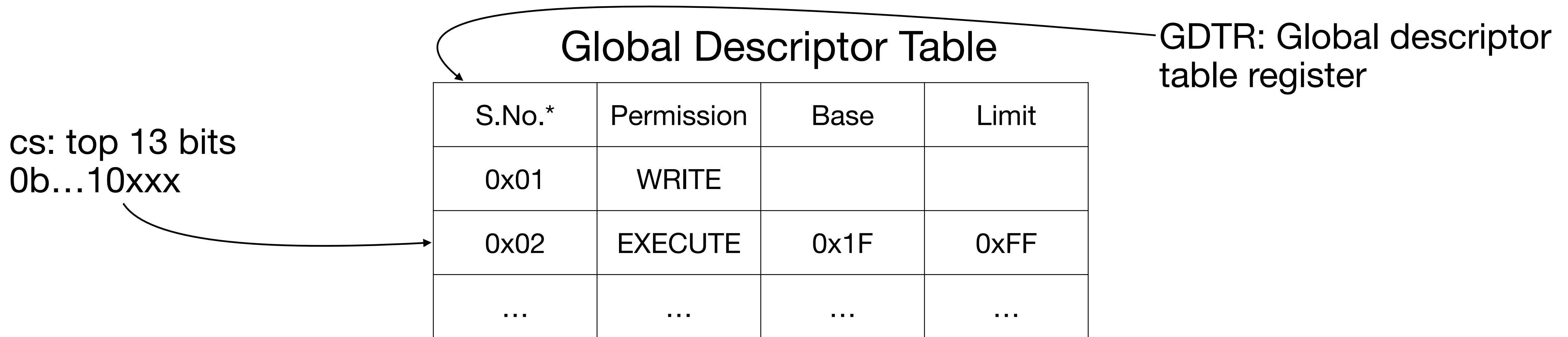
# Segment registers in 32-bit

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

\*: S.No. added only for illustration

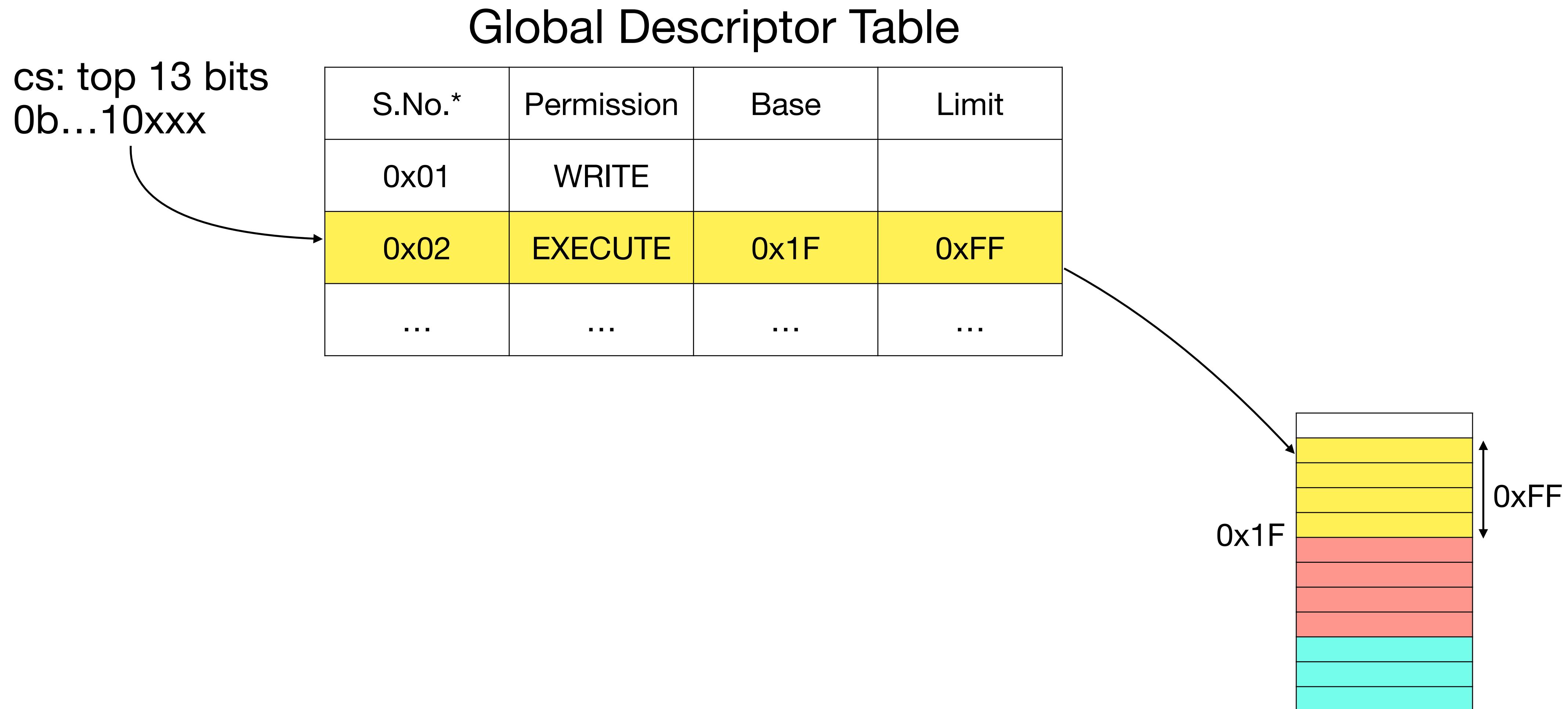
# Segment registers in 32-bit

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

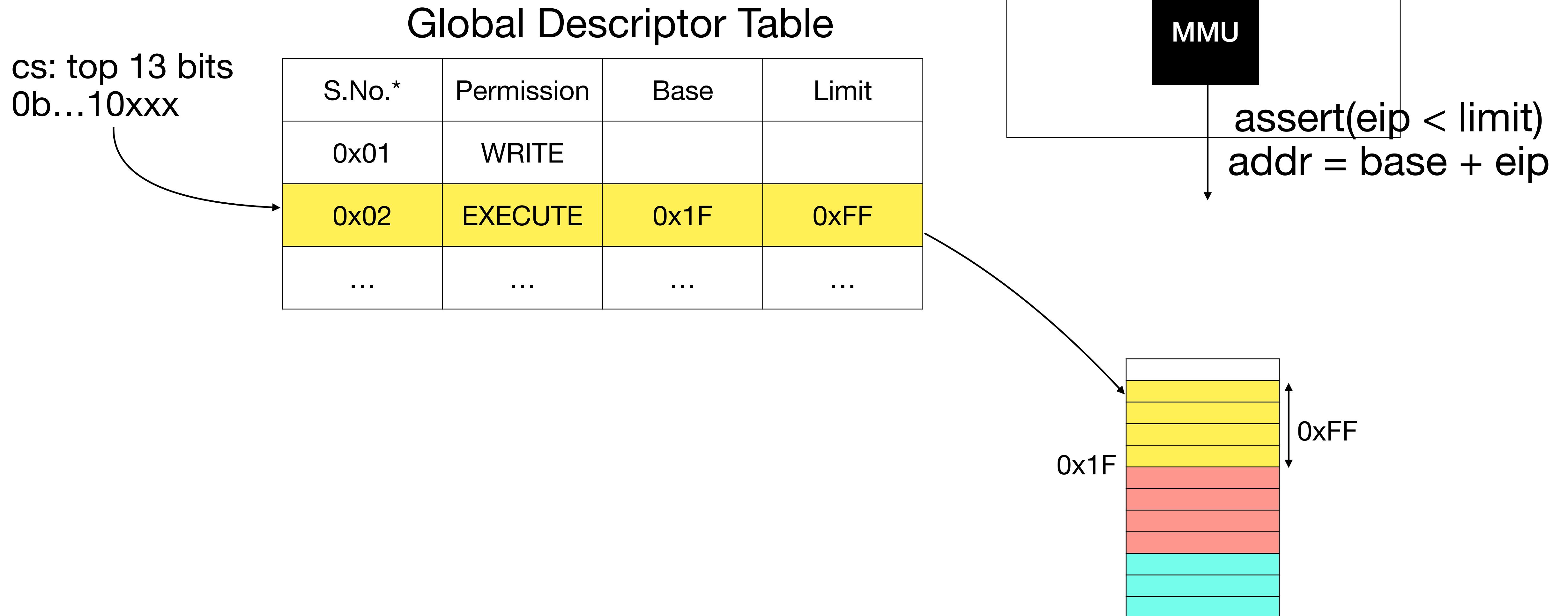


\*: S.No. added only for illustration

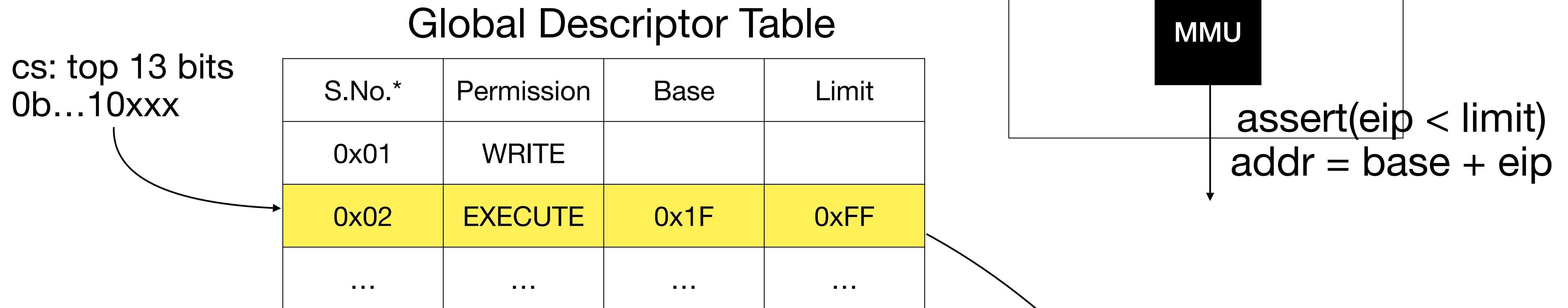
# Address translation



# Address translation



# Address translation



- Can “protect” different segments from each other

# Global descriptor table

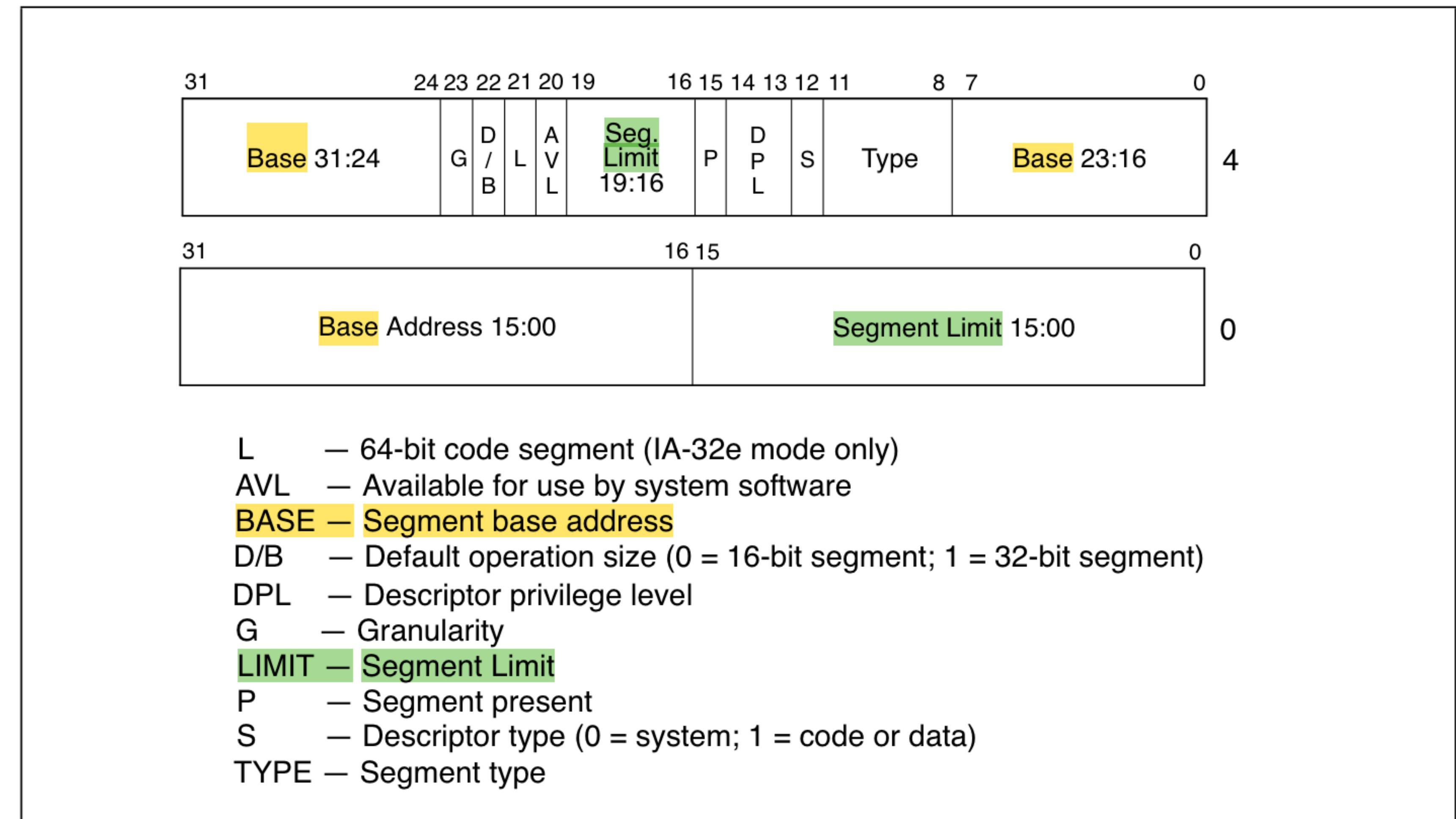


Figure 3-8. Segment Descriptor

# Global descriptor table

- Up to  $2^{13}$  (=8192) segment descriptors

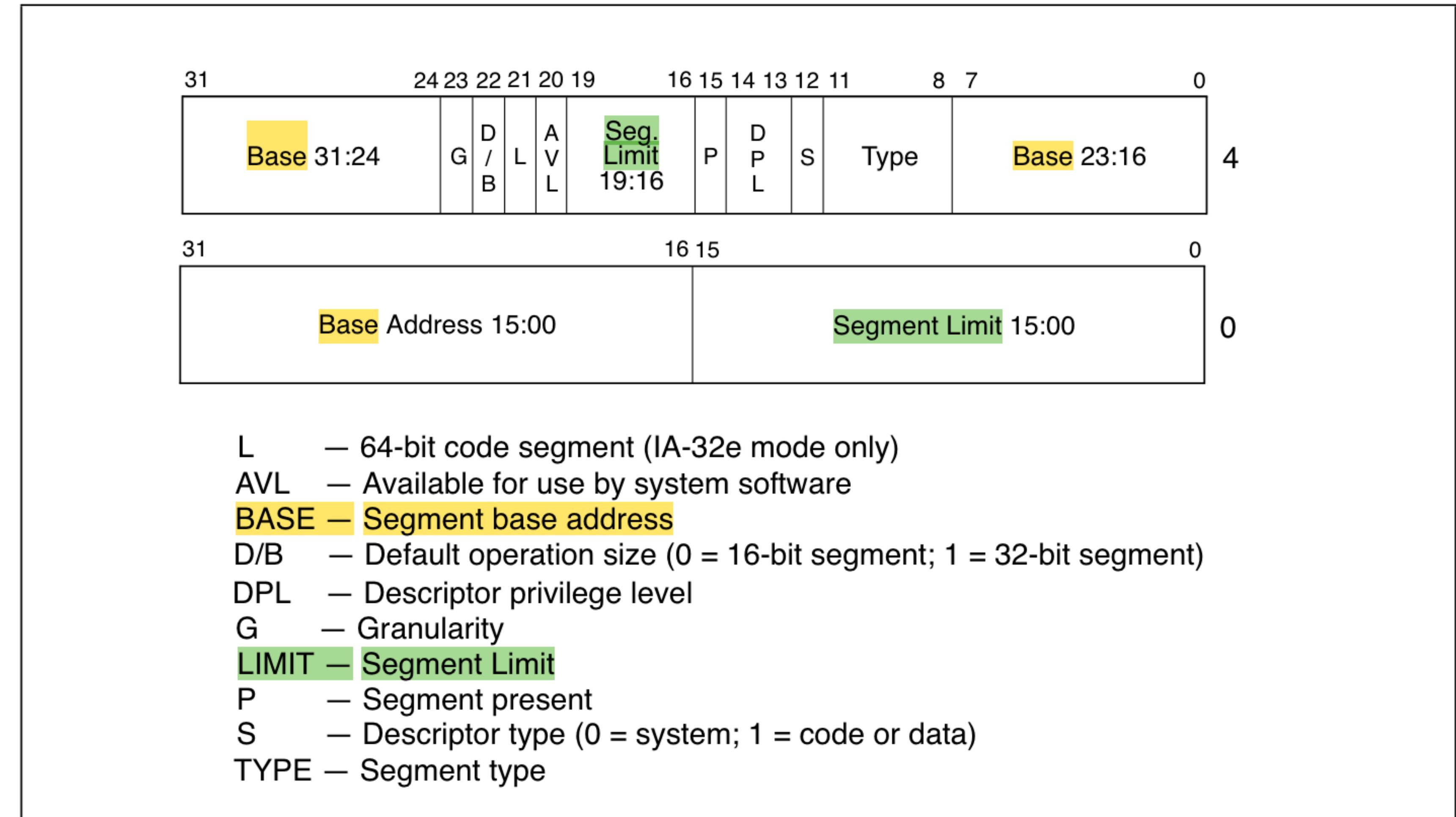


Figure 3-8. Segment Descriptor

# Global descriptor table

- Up to  $2^{13}$  (=8192) segment descriptors
- Segment descriptor:

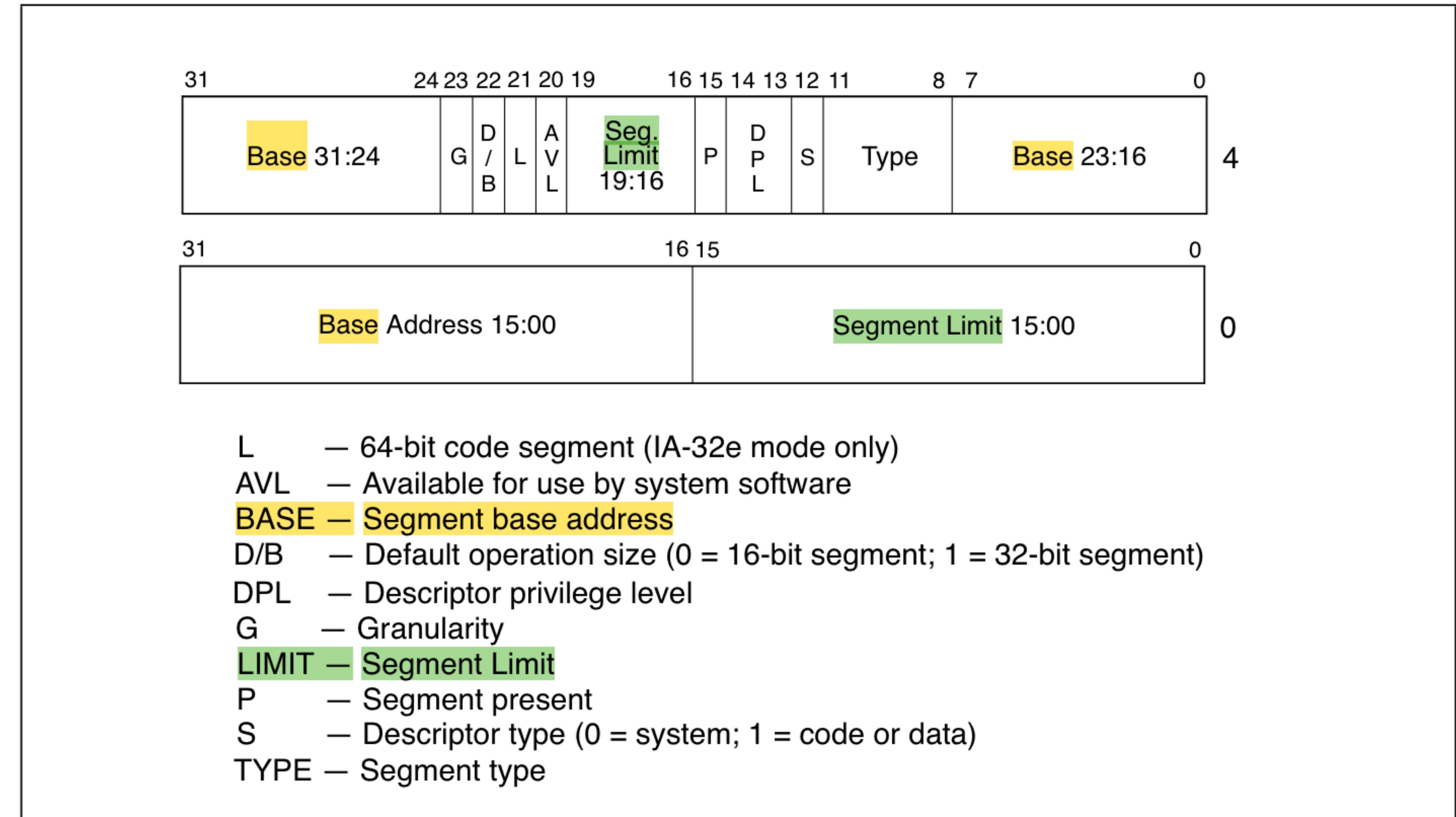


Figure 3-8. Segment Descriptor

# Global descriptor table

- Up to  $2^{13}$  (=8192) segment descriptors
- Segment descriptor:
  - 32 bit base

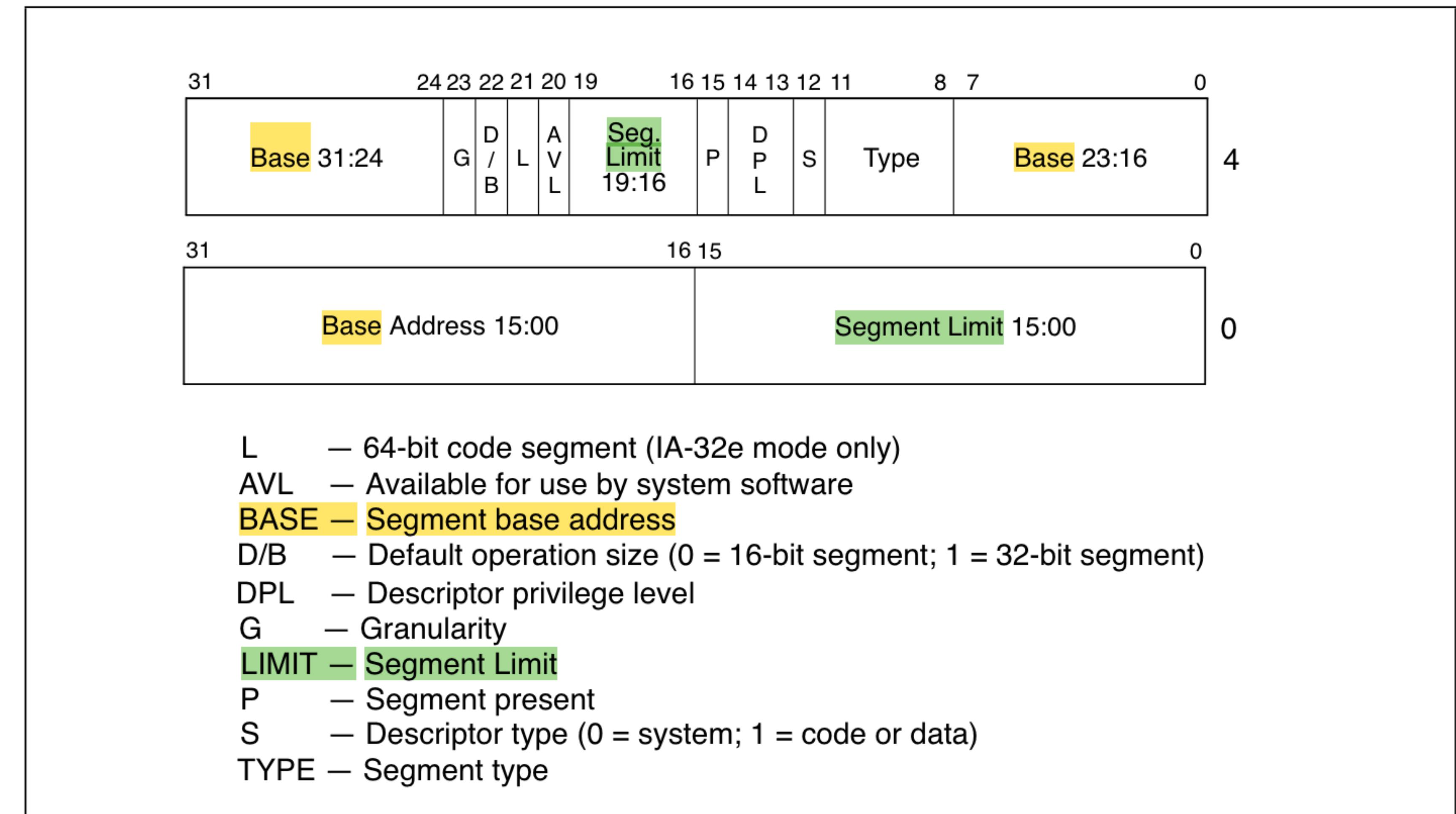


Figure 3-8. Segment Descriptor

# Global descriptor table

- Up to  $2^{13}$  (=8192) segment descriptors

- Segment descriptor:
  - 32 bit base
  - 20 bit limit

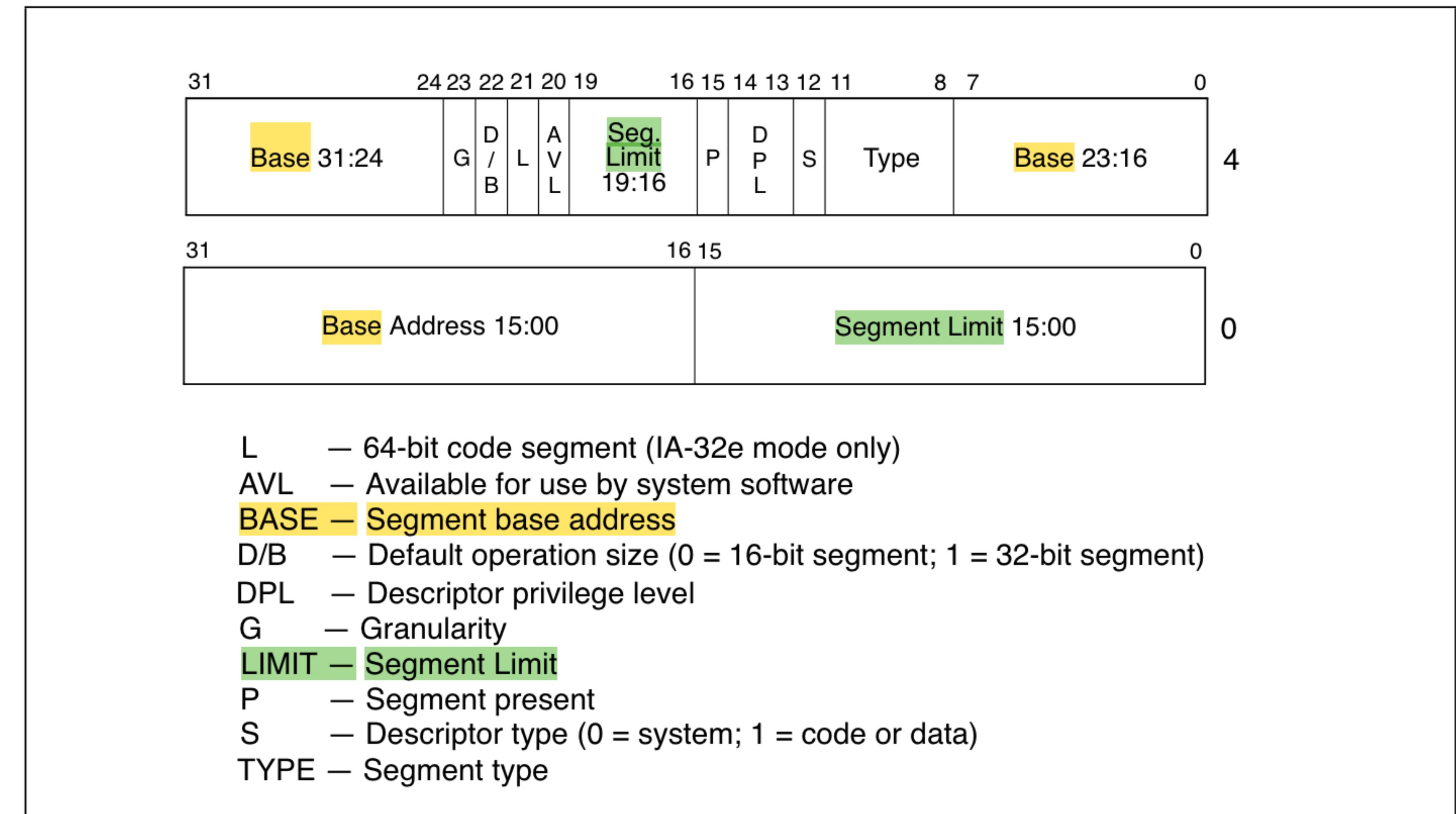


Figure 3-8. Segment Descriptor

# Global descriptor table

- Up to  $2^{13}$  (=8192) segment descriptors
- Segment descriptor:
  - 32 bit base
  - 20 bit limit
  - If G=1, granularity=4KB.

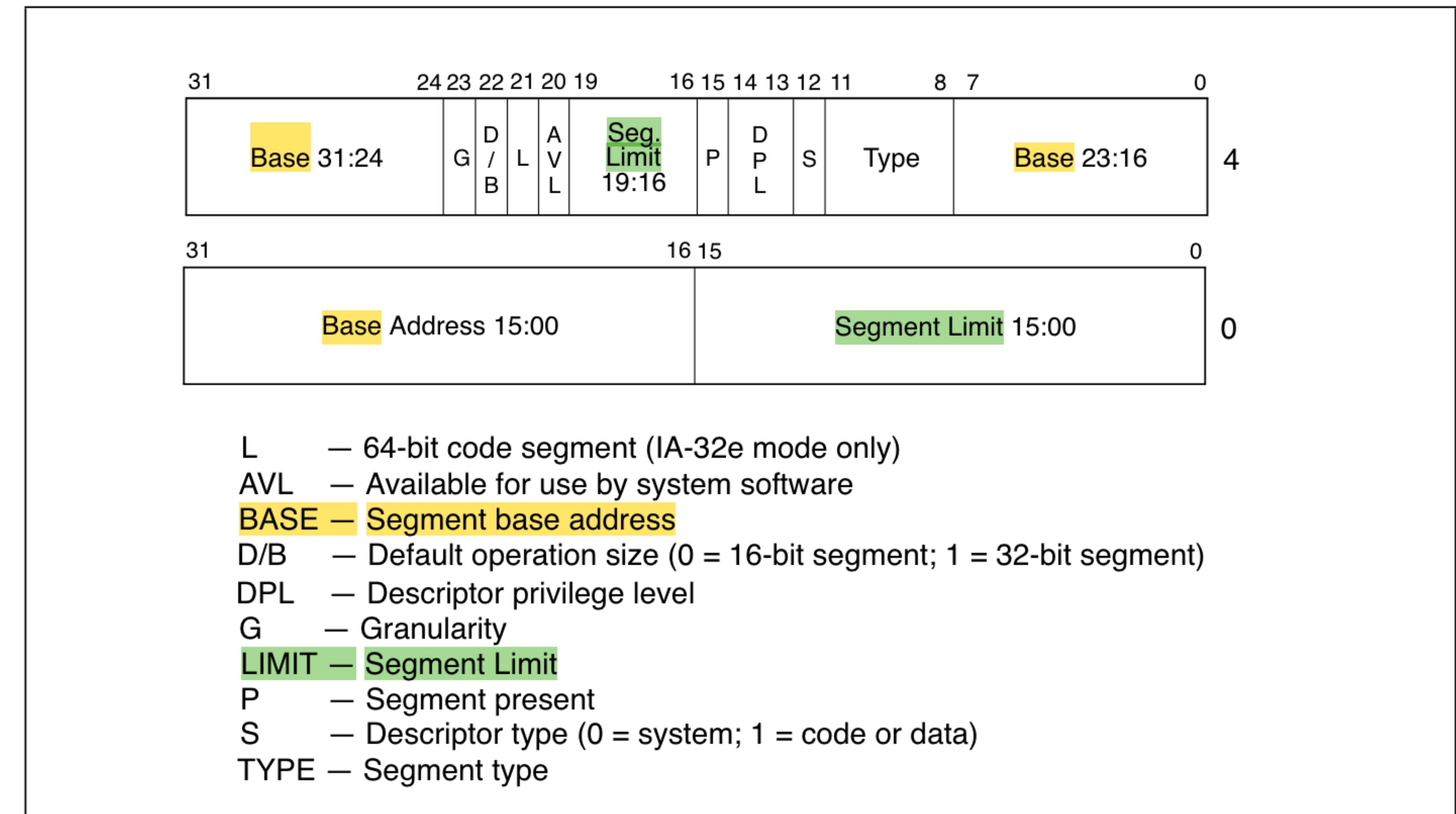


Figure 3-8. Segment Descriptor

# Global descriptor table

- Up to  $2^{13}$  (=8192) segment descriptors
- Segment descriptor:
  - 32 bit base
  - 20 bit limit
  - If G=1, granularity=4KB.
  - Max memory within 1 segment =  $2^{20} \cdot 2^{12} = 4\text{GB}$

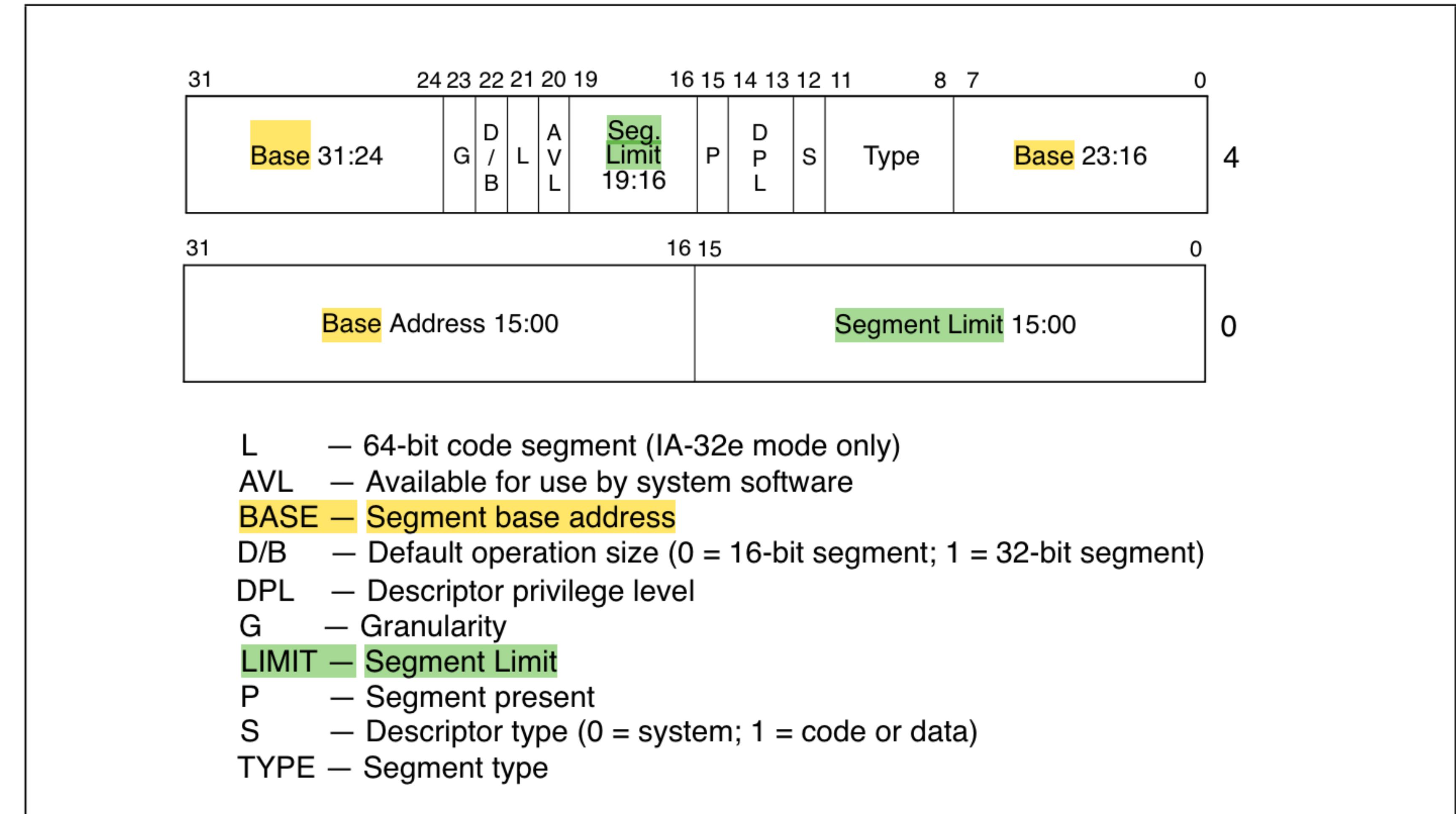


Figure 3-8. Segment Descriptor

# Segmented memory model

- Stack cannot grow into code section

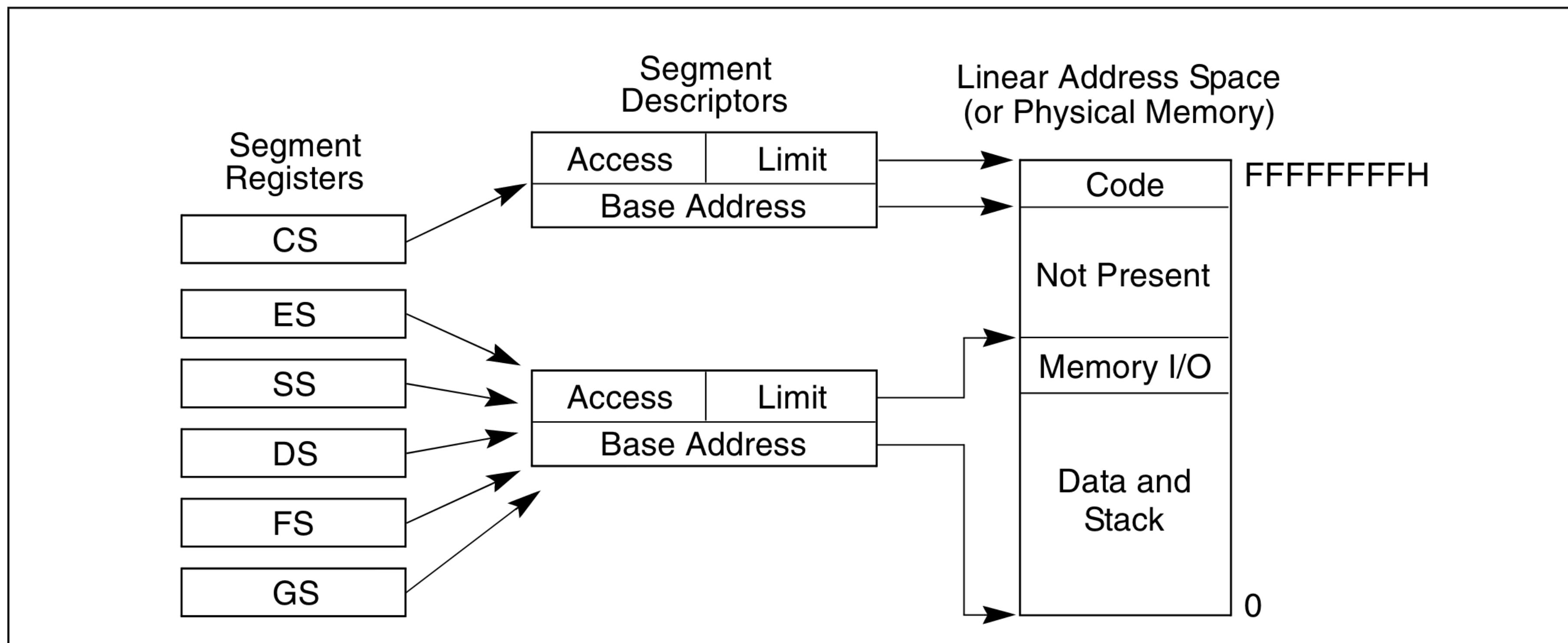


Figure 3-3. Protected Flat Model

# Multi-segment model

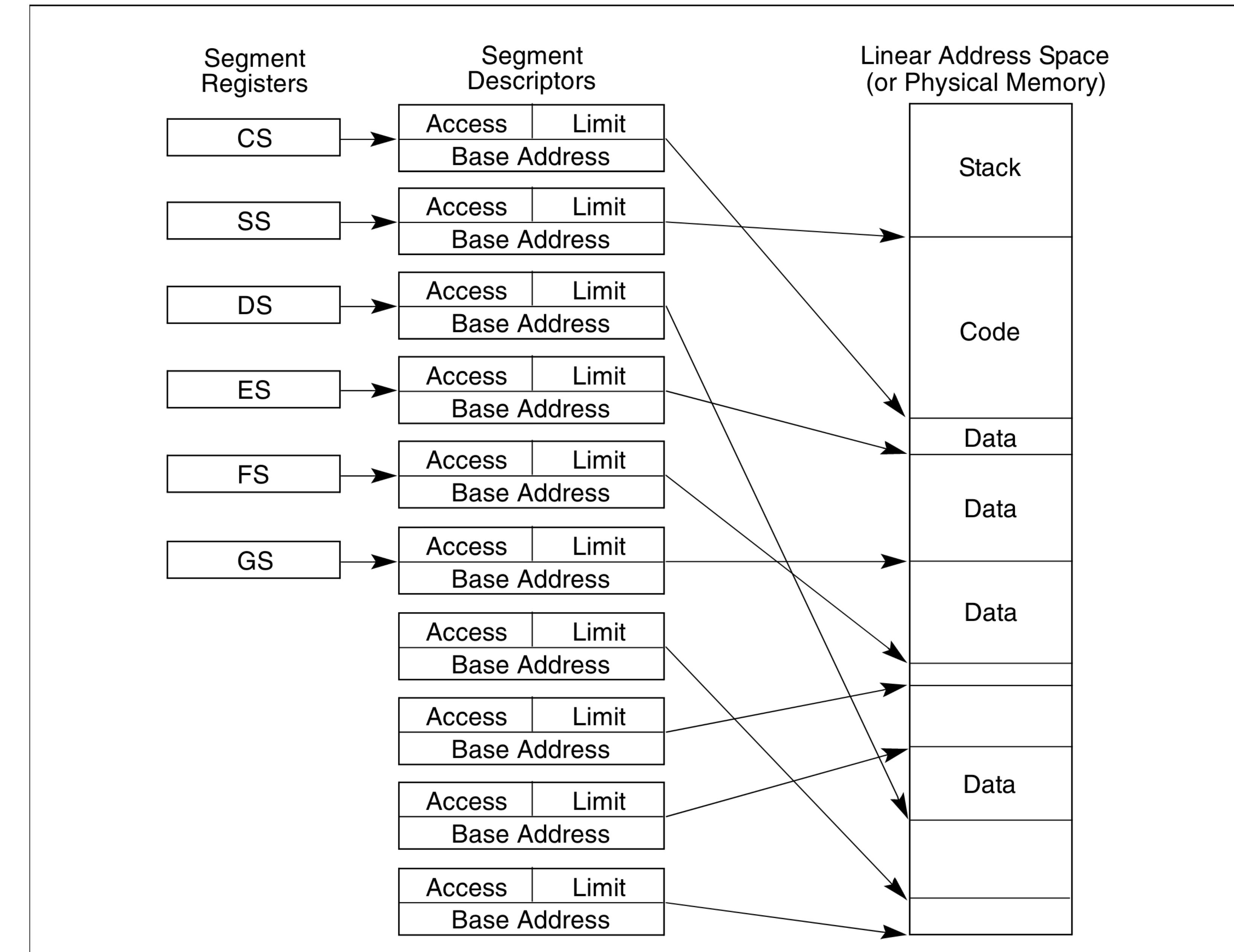


Figure 3-4. Multi-Segment Model

# Multi-segment model

- Best protection

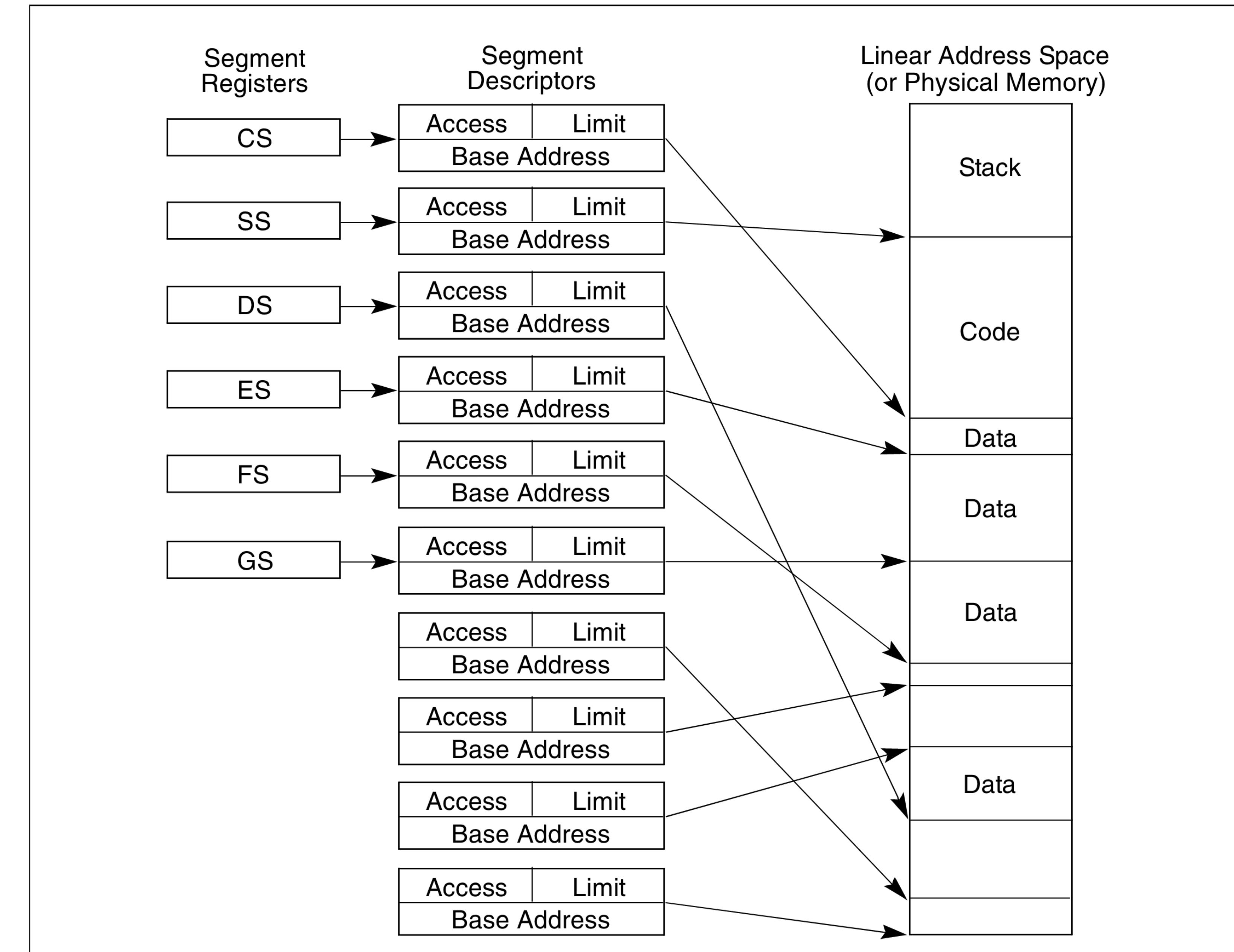


Figure 3-4. Multi-Segment Model

# Multi-segment model

- Best protection
- Difficult to program

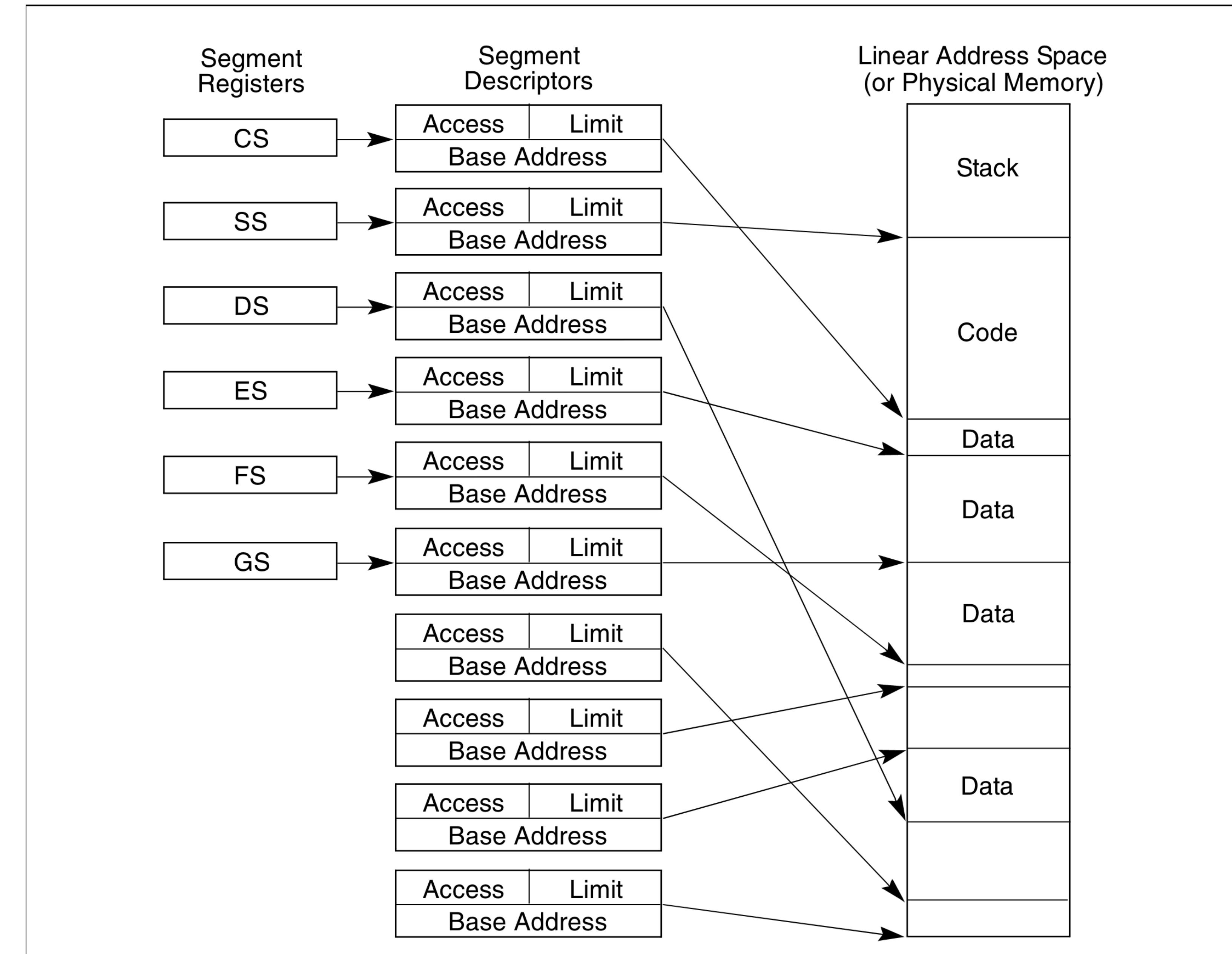


Figure 3-4. Multi-Segment Model

# Multi-segment model

- Best protection
- Difficult to program

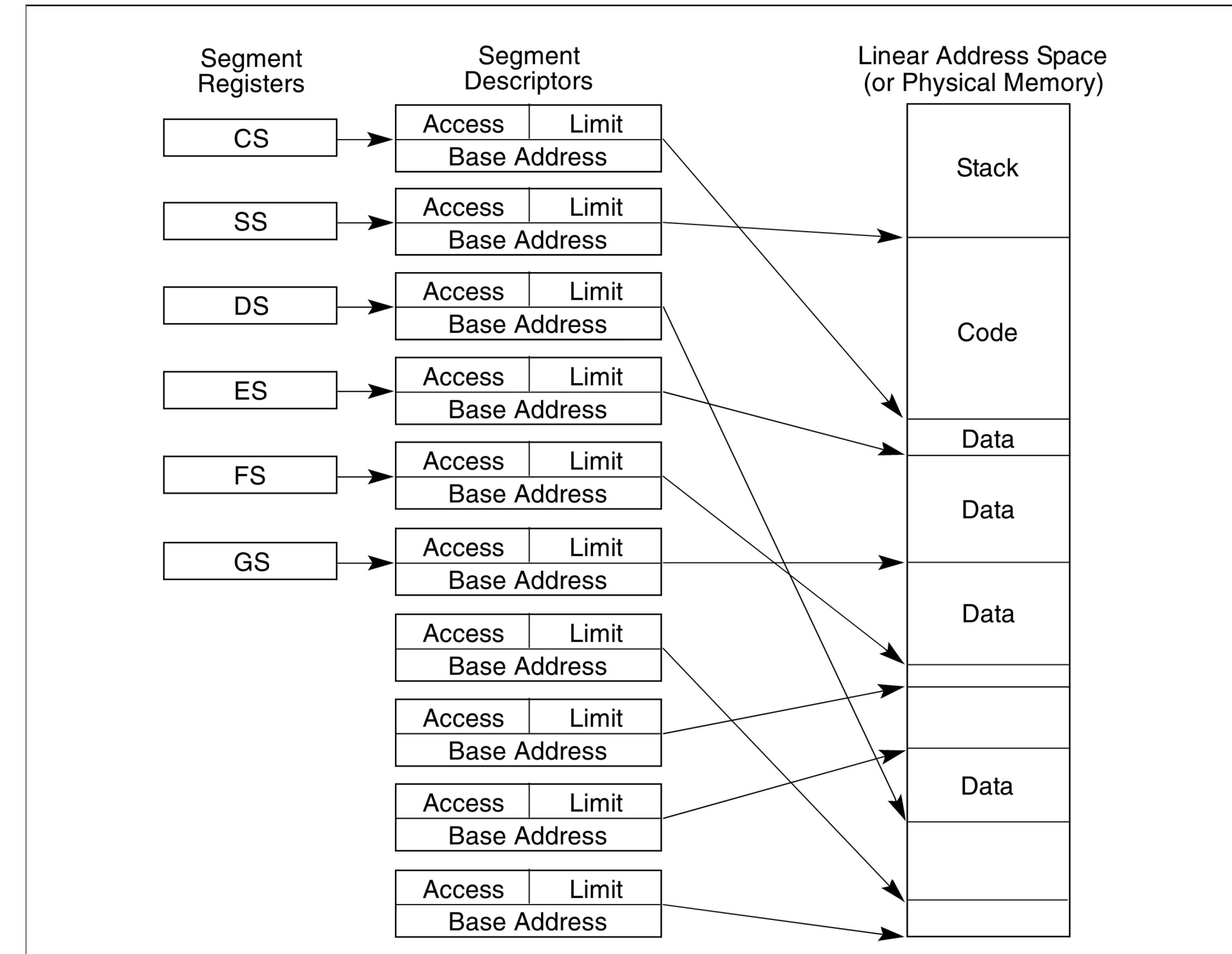


Figure 3-4. Multi-Segment Model

# Multi-segment model

- Best protection
- Difficult to program

`movl %esp %ecx`

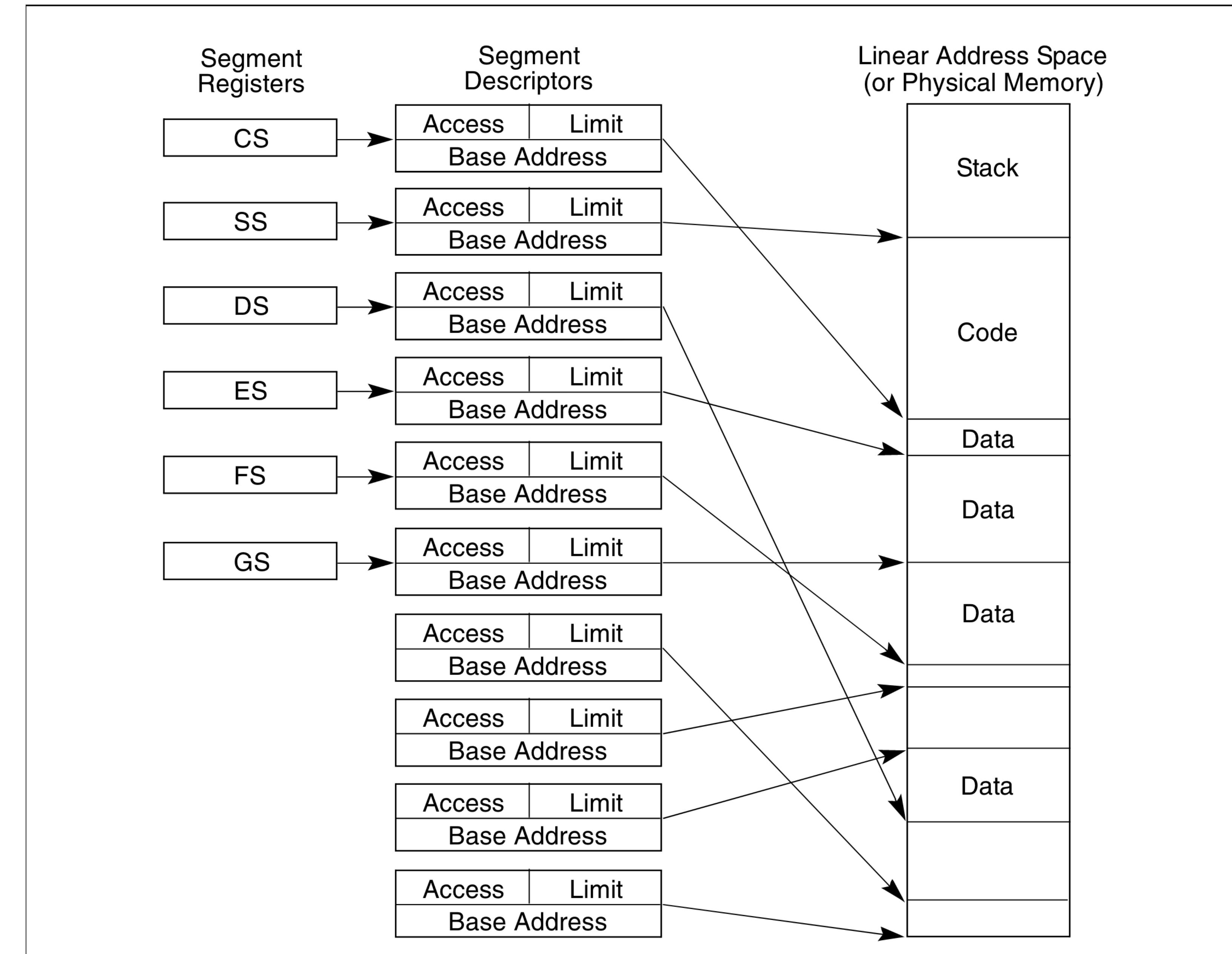


Figure 3-4. Multi-Segment Model

# Multi-segment model

- Best protection
- Difficult to program

`movl %esp %ecx`

`addl $1 (%ecx)`

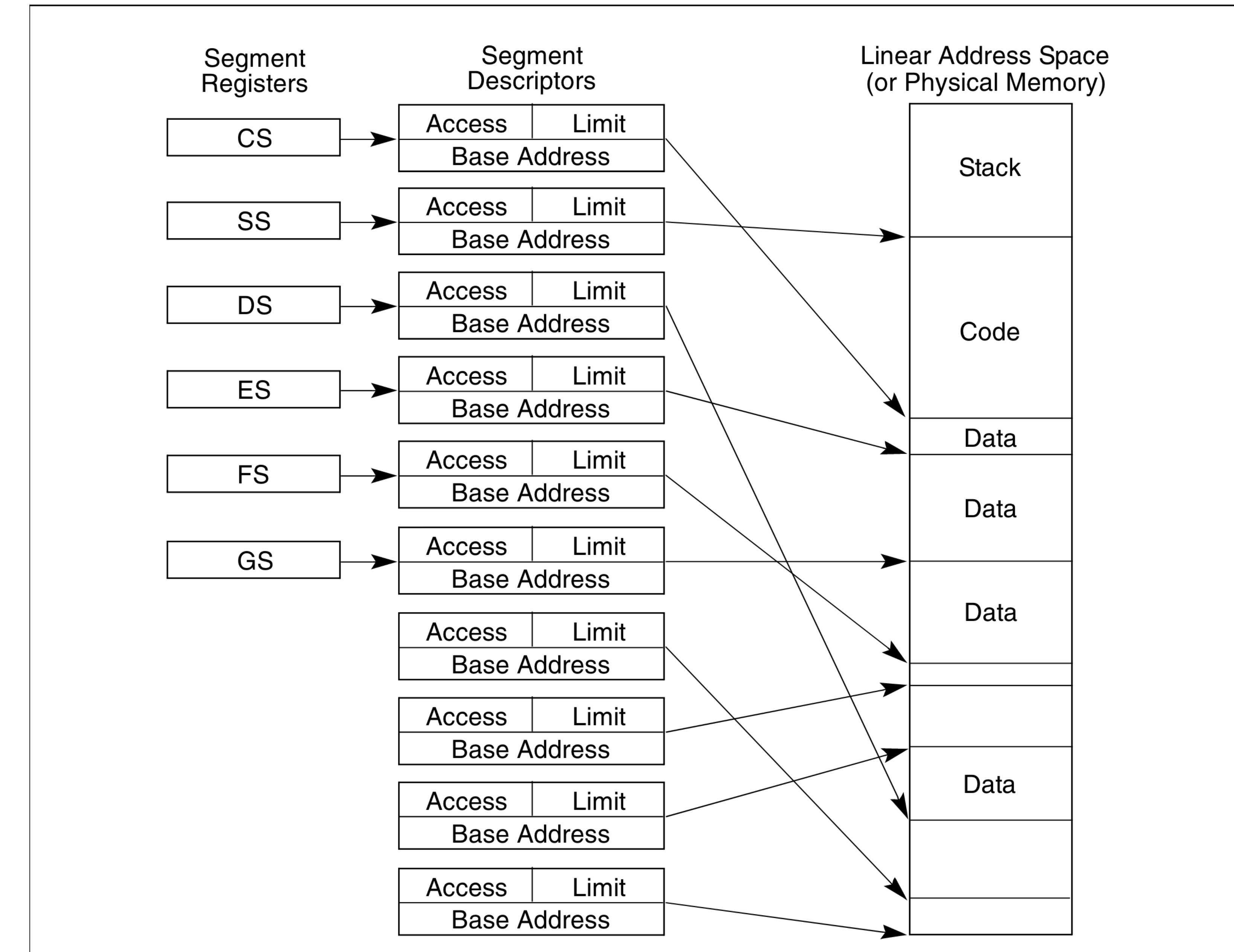


Figure 3-4. Multi-Segment Model

# 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!

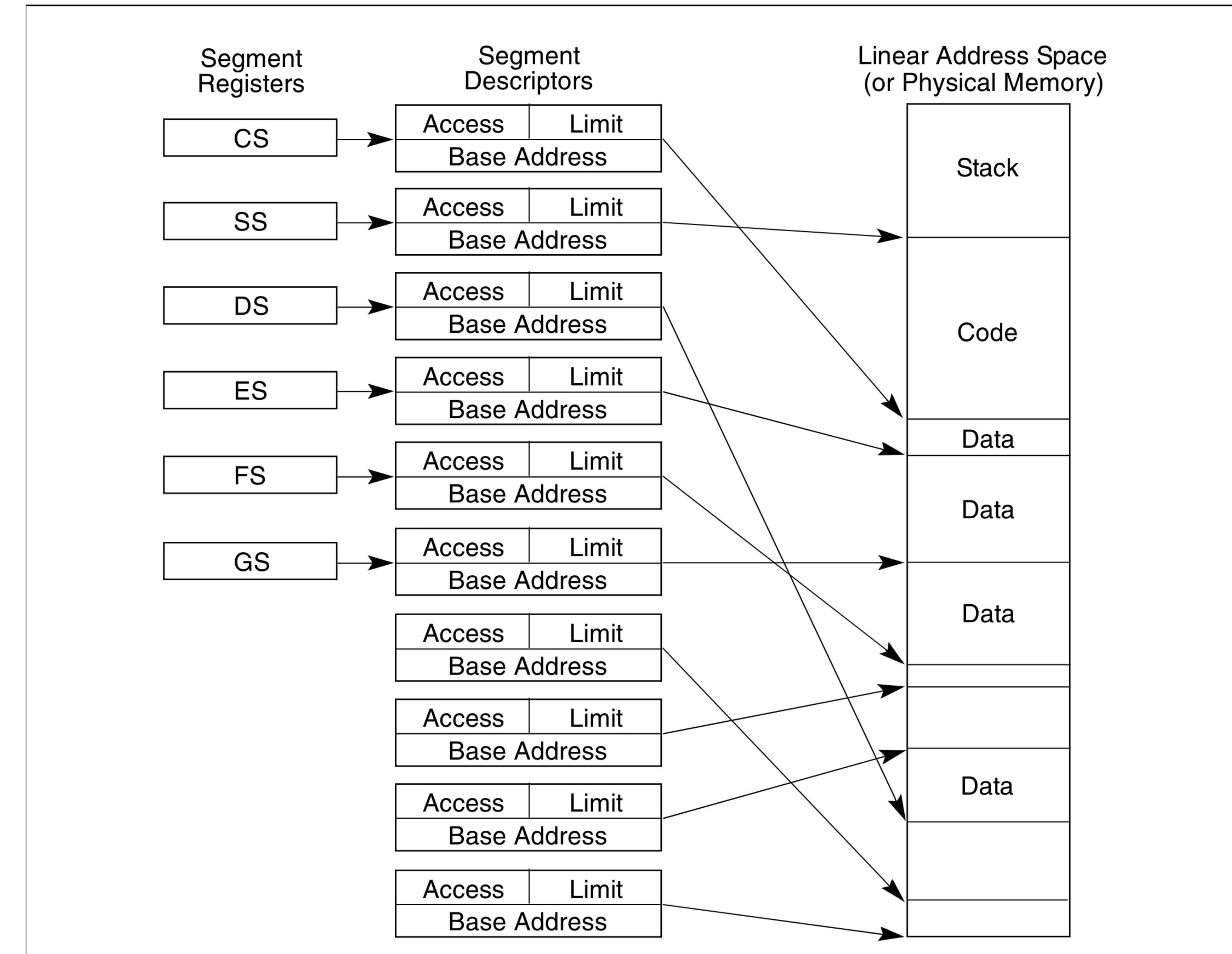


Figure 3-4. Multi-Segment Model

# 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!

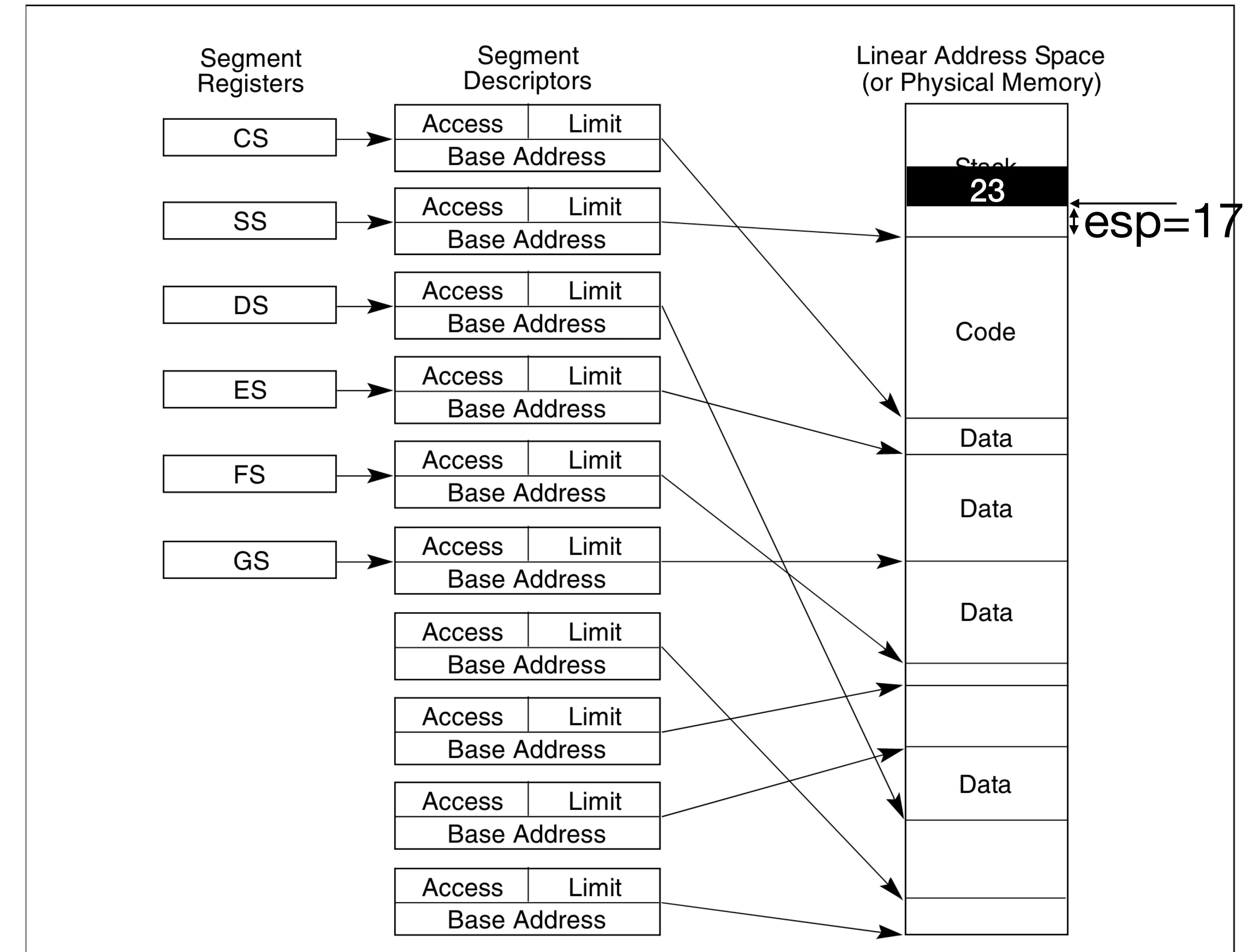


Figure 3-4. Multi-Segment Model

# 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!

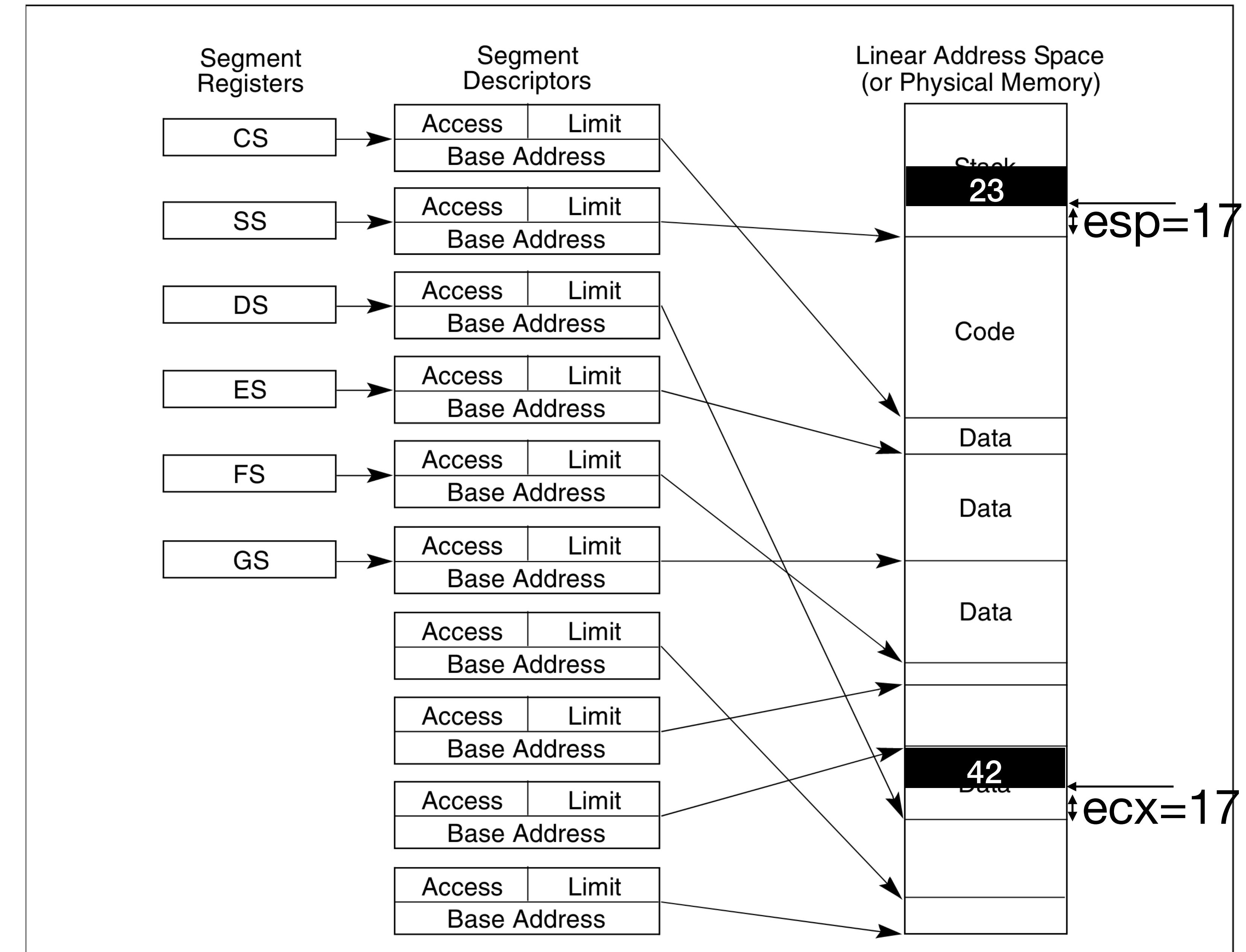


Figure 3-4. Multi-Segment Model

# Flat memory model

- Easier to program
- Used by xv6

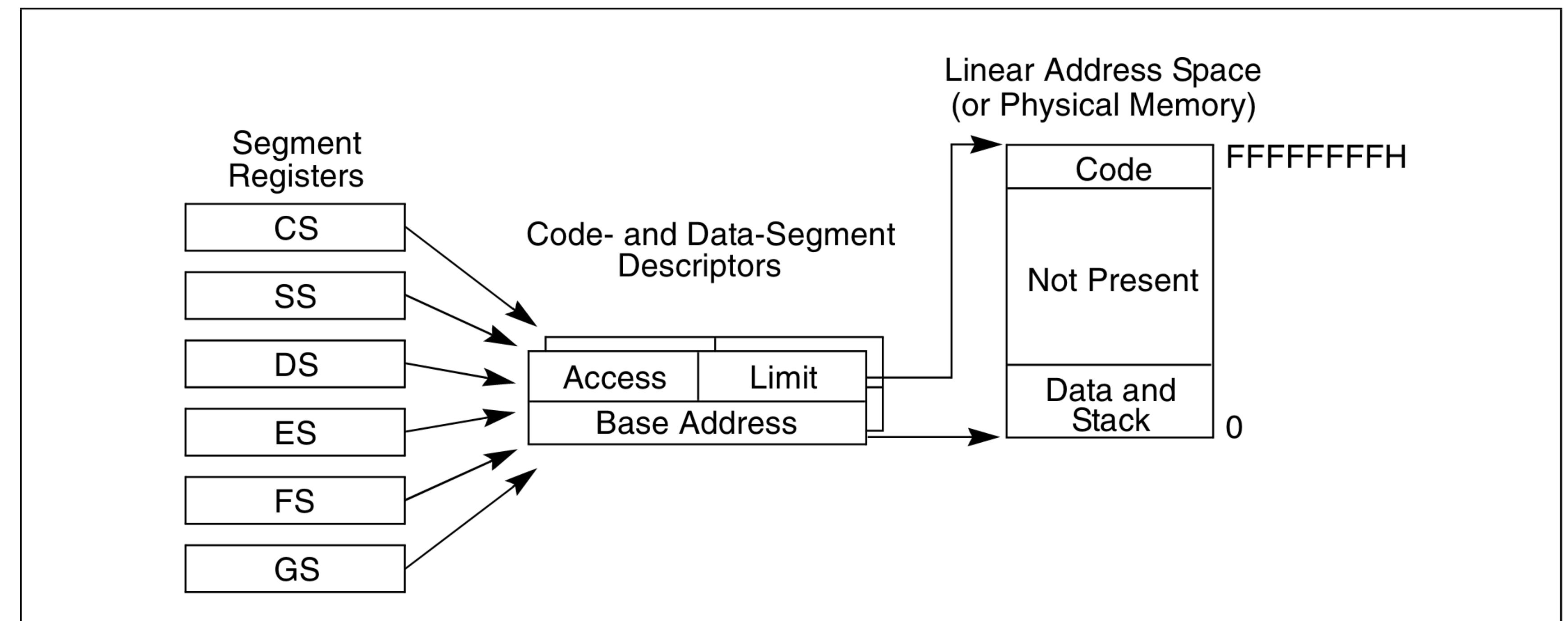


Figure 3-2. Flat Model

# **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?

gcc v10.0

gcc v12.0

clang

ELF format

Loader

# Why ELF?

- Port executables from one machine to another<sup>1</sup>, one OS to another<sup>2</sup>

gcc v10.0

gcc v12.0

clang

ELF format

Loader

# Why ELF?

- Port executables from one machine to another<sup>1</sup>, one OS to another<sup>2</sup>
- Our OS is just an executable!

gcc v10.0

gcc v12.0

clang

ELF format

Loader

# Why ELF?

- Port executables from one machine to another<sup>1</sup>, one OS to another<sup>2</sup>
- Our OS is just an executable!

gcc v10.0

gcc v12.0

clang

ELF format

Loader

# Why ELF?

- Port executables from one machine to another<sup>1</sup>, one OS to another<sup>2</sup>
- Our OS is just an executable!

gcc v10.0

gcc v12.0

ELF format

Loader

clang

1: Same architecture

# Why ELF?

- Port executables from one machine to another<sup>1</sup>, one OS to another<sup>2</sup>
- Our OS is just an executable!

gcc v10.0

gcc v12.0

ELF format

Loader

clang

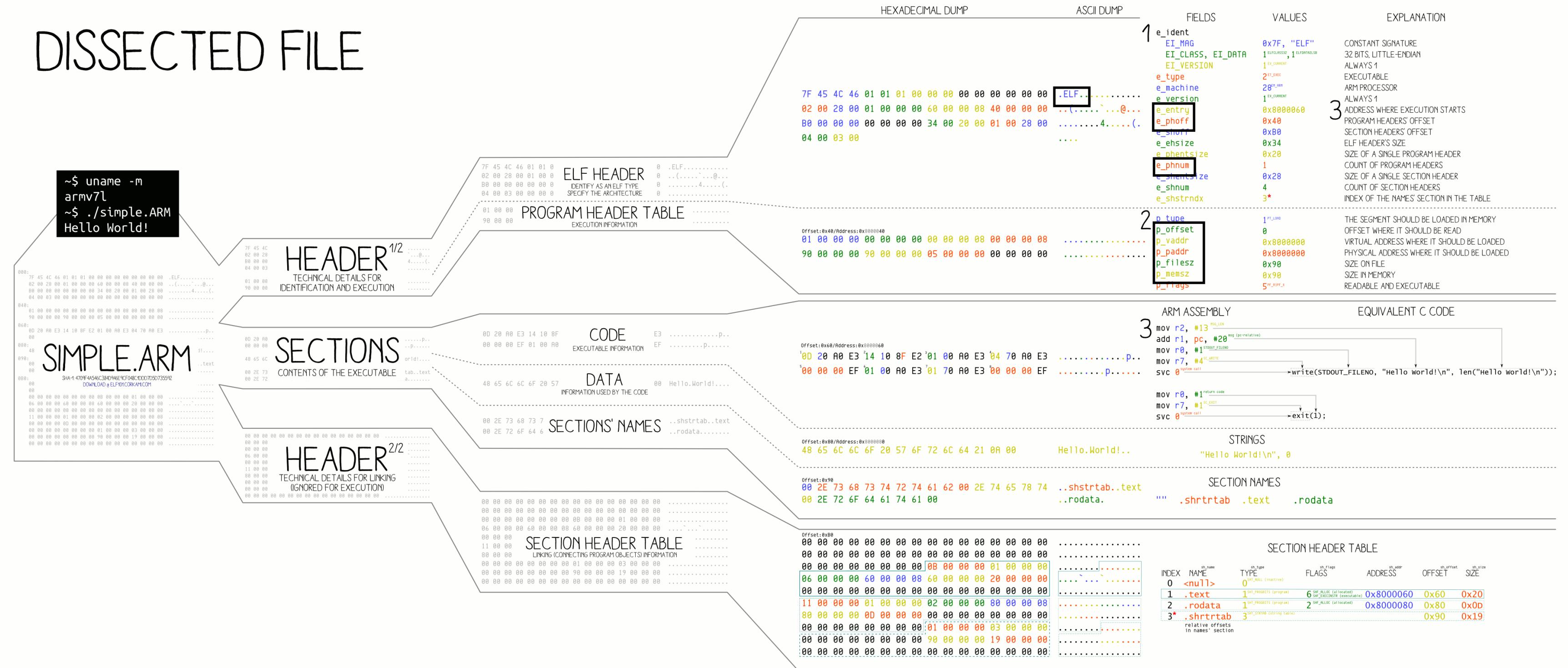
1: Same architecture

2. Same system calls

# ELF

- ELF header: magic constant, program header offset, count of program headers, entry

## DISSECTED FILE



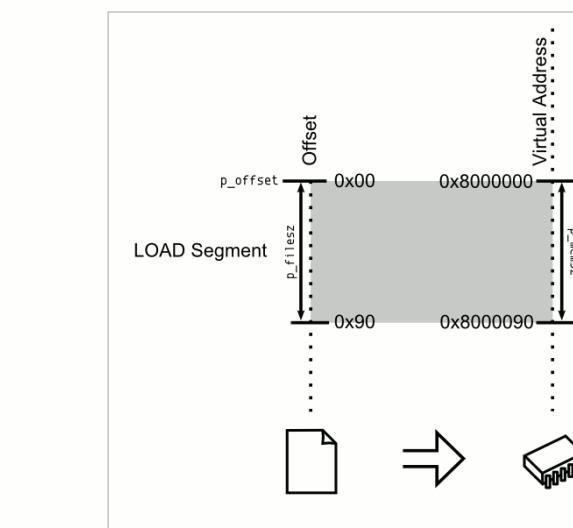
## LOADING PROCESS

### 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 ARE ACCESSED VIA:  
- SYSCALL NUMBER IN THE R7 REGISTER  
- 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



# 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 0x0C001000 which is little endian for 0x0010000C
  - In kernel.asm, entry is 0x0010000C

# 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	<b>0x001000</b>	0x00100000	<b>0x00100000</b>	<b>0x005A</b>	0x005A	RE	0x1000
LOAD	0x000000	0x00101000	<b>0x00101000</b>	<b>0x0000</b>	<b>0x01000</b>	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

4. Jump to address 0x10000c to start executing

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.

# 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	Direction	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
5, 6, 7	Always set

Command registers	Function
0x20	Read sector
0x30	Write sector

# 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

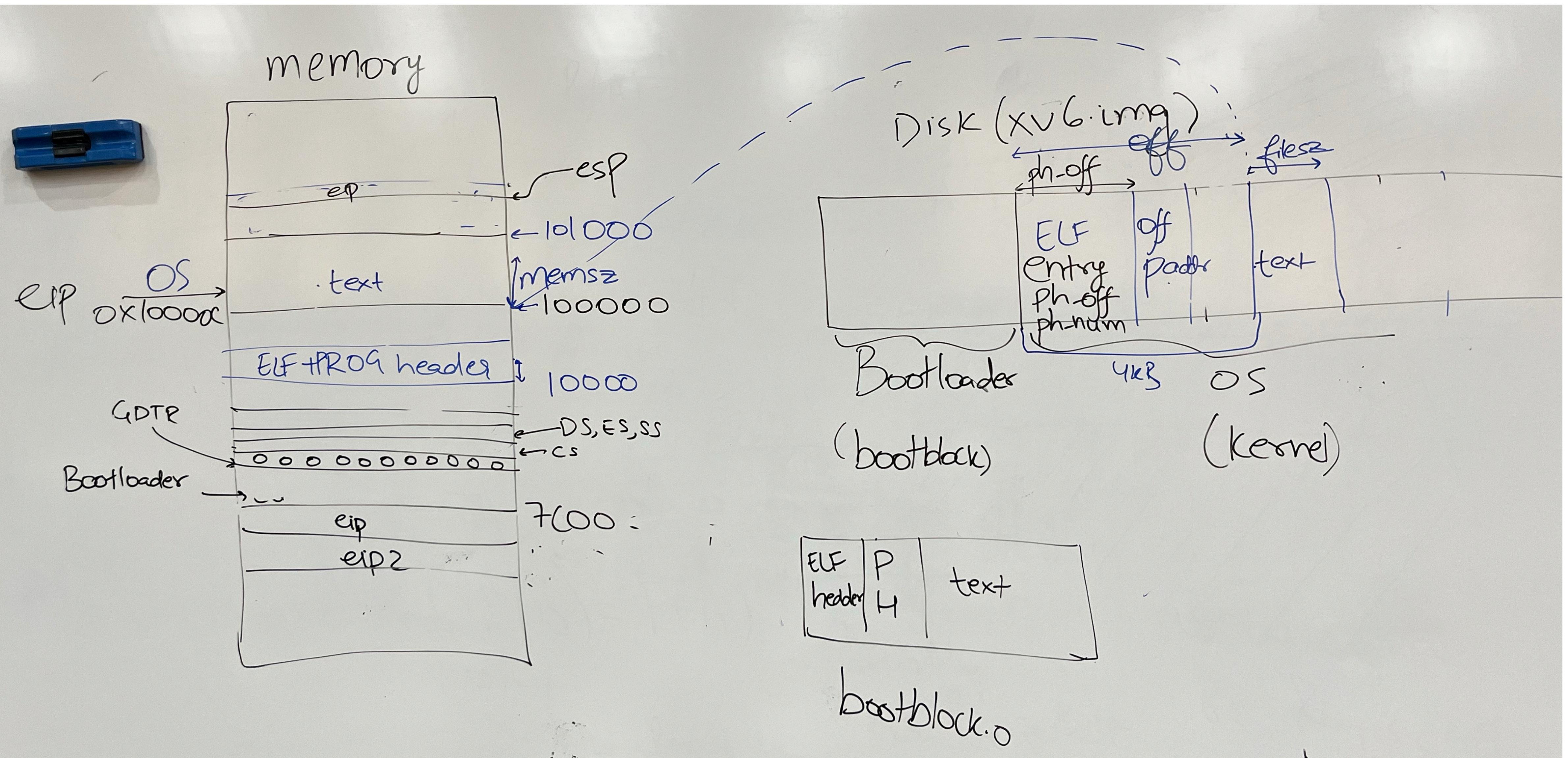
asm volatile("cld; rep insl" : "=D" (addr), "=c" (cnt) : "d" (port), "0" (addr), "1" (cnt) : "memory", "cc");	asm volatile("cld; rep insl" : 7ce8: 8b 7d 08      mov    0x8(%ebp),%edi 7ceb: b9 80 00 00 00  mov    \$0x80,%ecx 7cf0: ba f0 01 00 00  mov    \$0x1f0,%edx 7cf5: fc             cld 7cf6: f3 6d          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>)

# Code walkthrough



# 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.