

238P Operating Systems, Fall 2018

xv6 Boot Recap: Transitioning from 16 bit mode to 32 bit mode

3 November 2018

Aftab Hussain

University of California, Irvine

BIOS → xv6 Boot loader

what it does

Sets up the hardware.

Transfers control to the Boot Loader.

BIOS → xv6 Boot loader

what it does

Sets up the hardware.

Transfers control to the Boot Loader.

how it transfers control to the Boot Loader

Boot loader is **loaded from** the 1st 512-byte sector of the boot disk.

This 512-byte sector is known as the **boot sector**.

Boot loader is **loaded at** 0x7c00.

Sets processor's **ip** register to 0x7c00.

BIOS → xv6 Boot loader

2 source source files

bootasm.S - 16 and 32 bit assembly code.

bootmain.c - C code.

BIOS → xv6 Boot loader

2 source source files

bootasm.S - 16 and 32 bit assembly code.

bootmain.c - C code.

executing bootasm.S

1. **Disable interrupts** using `cli` instruction. ([Code](#)).
 - > Done in case BIOS has initialized any of its interrupt handlers while setting up the hardware. Also, BIOS is not running anymore, so better to disable them.
 - > Clear segment registers. Use xor for %ax, and copy it to the rest ([Code](#)).
2. **Switch from real mode to protected mode.** (References: [a](#), [b](#)).
 - > Note the difference between processor modes and kernel privilege modes
 - > We do the above switch to increase the size of the memory we can address.

BIOS → xv6 Boot loader

executing bootasm.S

Let's expand on this a little bit

2. Switch from real mode to protected mode.

Addressing in Real Mode

In real mode, the processor sends 20-bit addresses to the memory.
(e.g. Intel processors 8086, 8088).

However, it has eight general 16-bit registers + 16 bit segment registers.

How to generate a 20 bit address from a 16 bit register?

Say processor fetches a data read/write instruction.
The processor would then use the data segment register (%ds).

To pass the 16-bit address obtained from %ds
to the memory (which accepts 20-bit addresses) we left
shift the 16-bit register by 4 bits.

This is equivalent to **adding the register
to itself 16 times.**
(try it out in a bit calculator - e.g. gnu calculator)

*Use other kinds of
segment registers
for other kinds of instructions.
For example for stack reads/writes use %ss.*

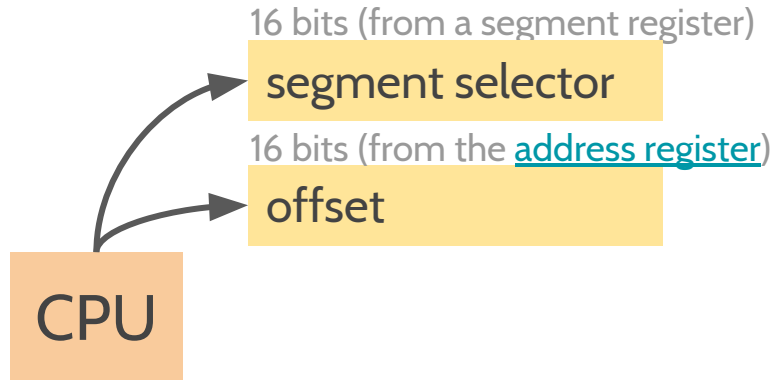
Say processor fetches a data read/write instruction.
The processor would then use the data segment register (%ds).

To pass the 16-bit address obtained from %ds
to the memory (which accepts 20-bit addresses) we left
shift the 16-bit register by 4 bits.

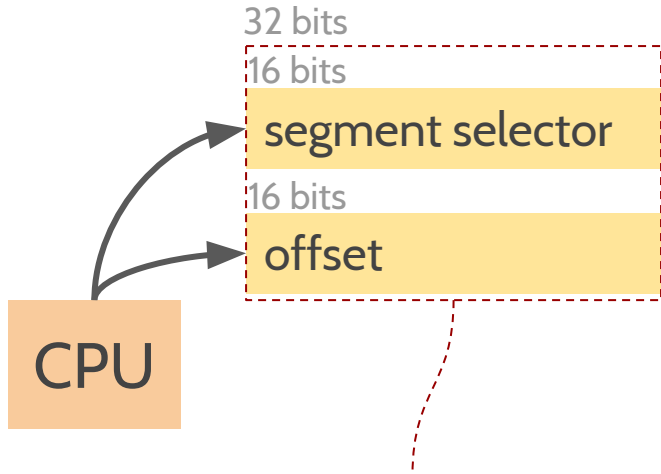
This is equivalent to **adding the register
to itself 16 times.**
(try it out in a bit calculator - e.g. gnu calculator)

So let's look at how the
address translation process takes place in real mode...

Segmentation in Real Mode

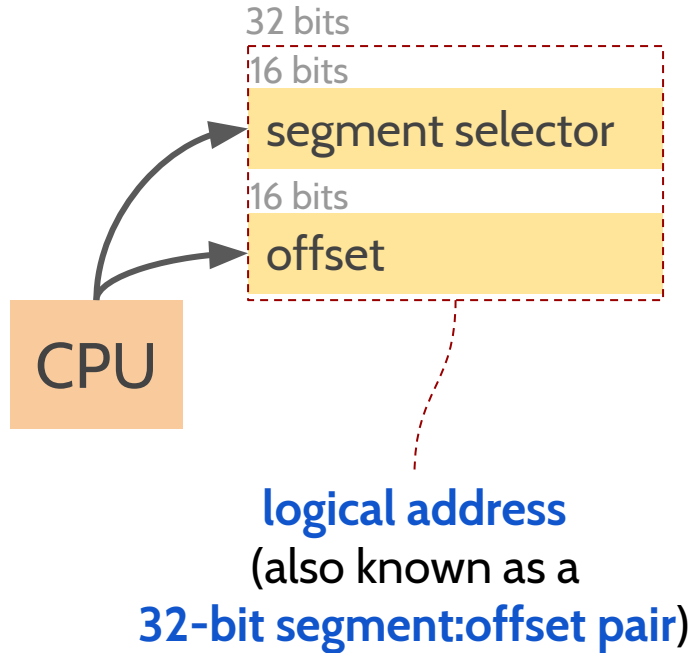


Segmentation in Real Mode



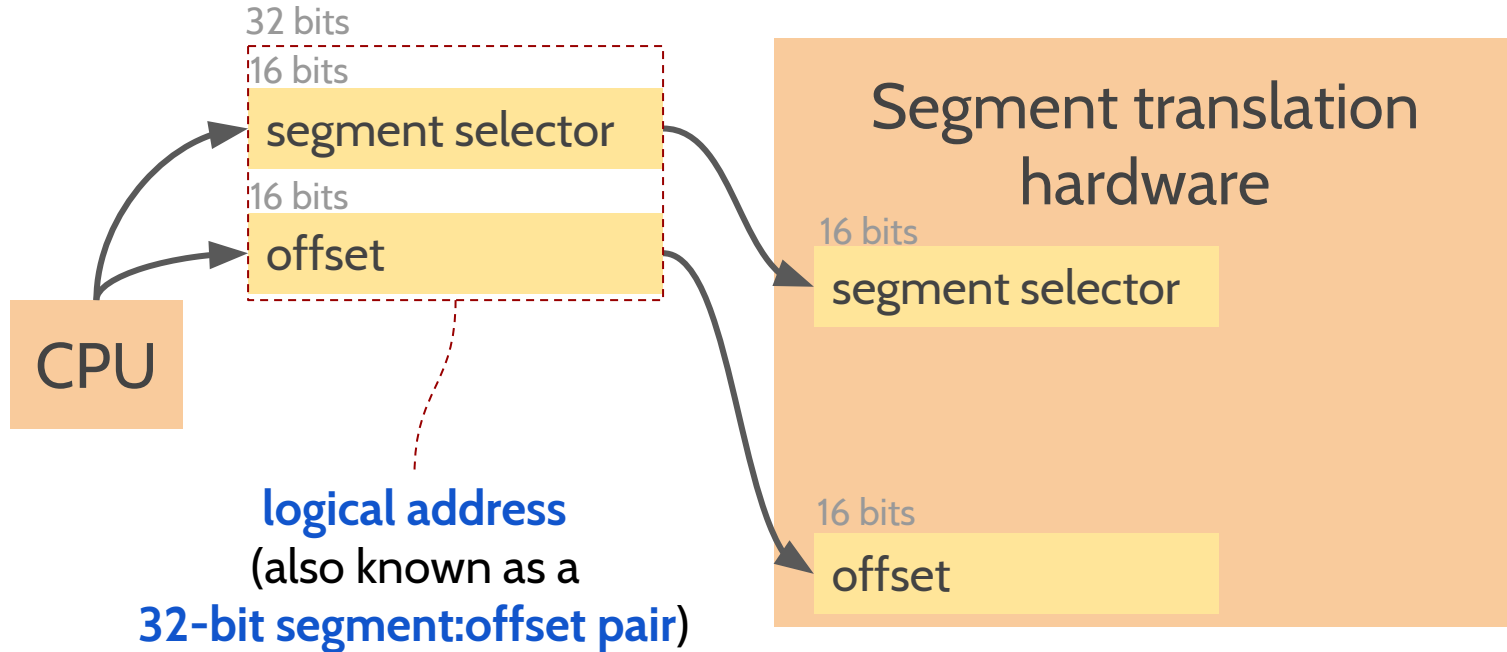
logical address
(also known as a
32-bit segment:offset pair)

Segmentation in Real Mode

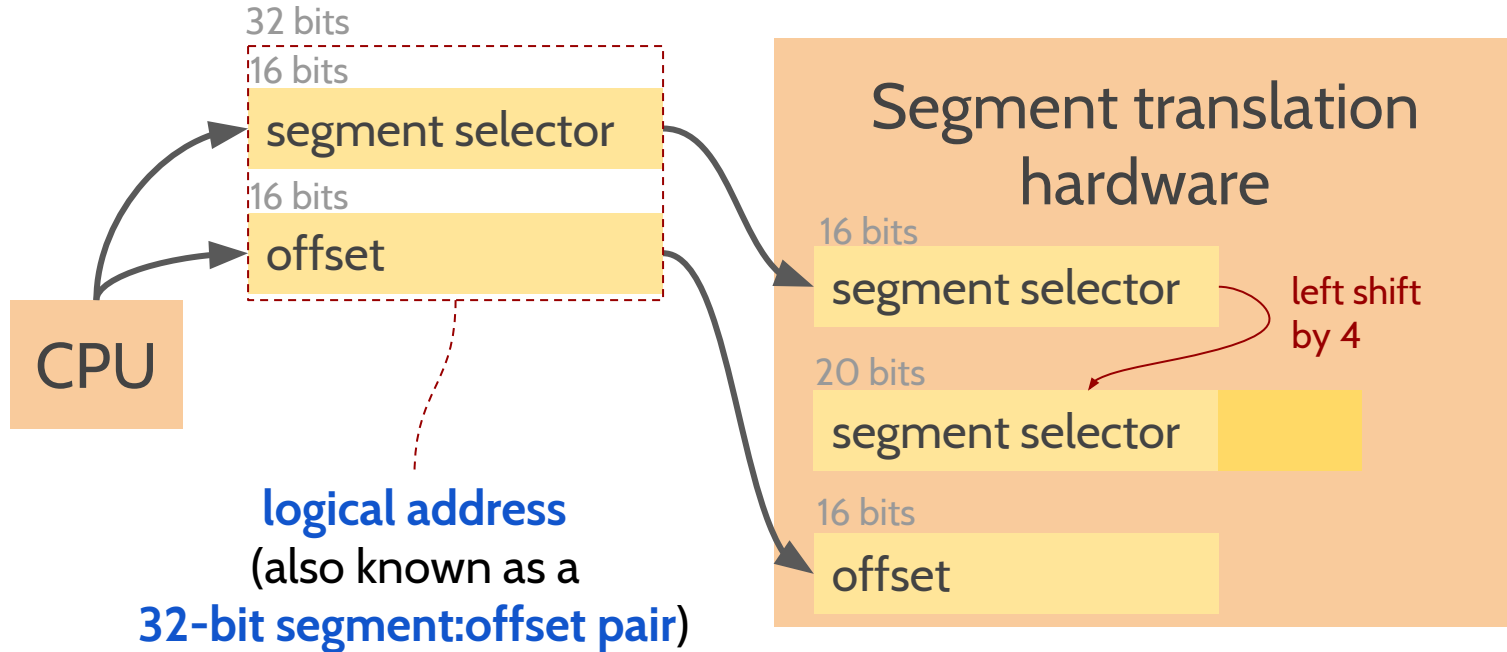


xv6 refers to this x86 logical address as a **virtual address**

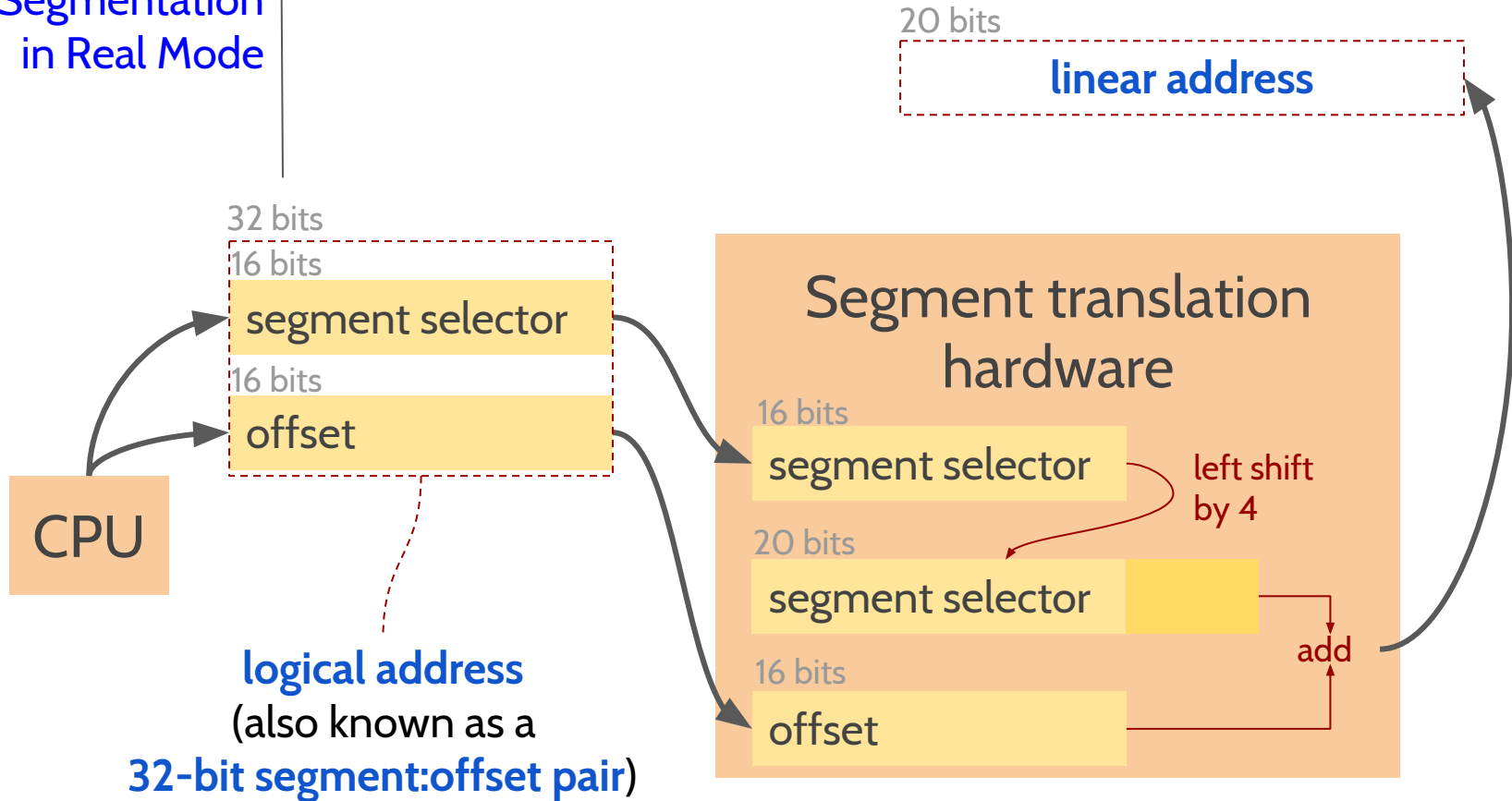
Segmentation in Real Mode



Segmentation in Real Mode



Segmentation in Real Mode



Segmentation in Real Mode

directly corresponds to the **physical address**

20 bits

linear address

32 bits

16 bits

segment selector

16 bits

offset

CPU

Segment translation hardware

16 bits

segment selector

left shift by 4

20 bits

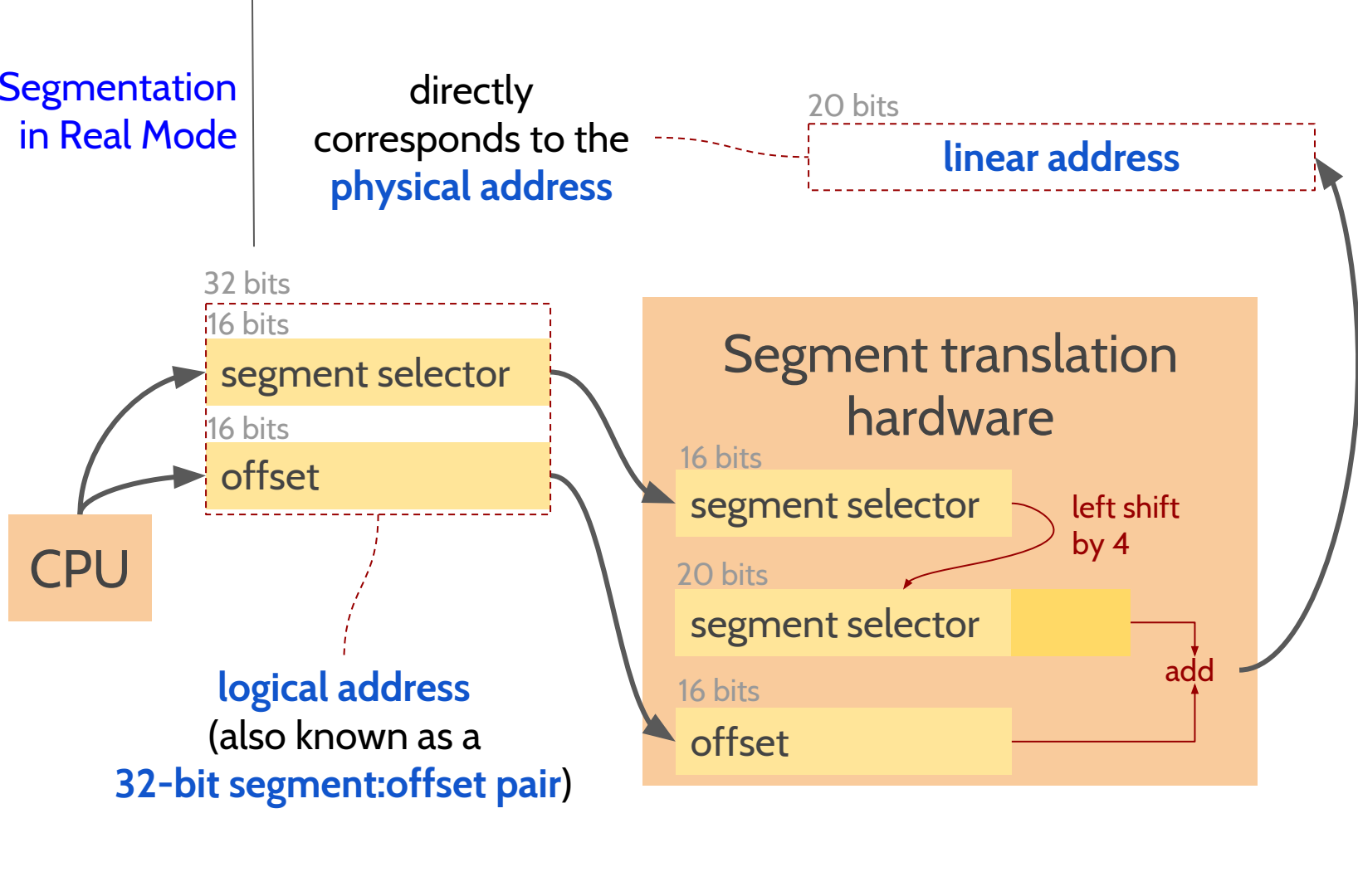
segment selector

16 bits

offset

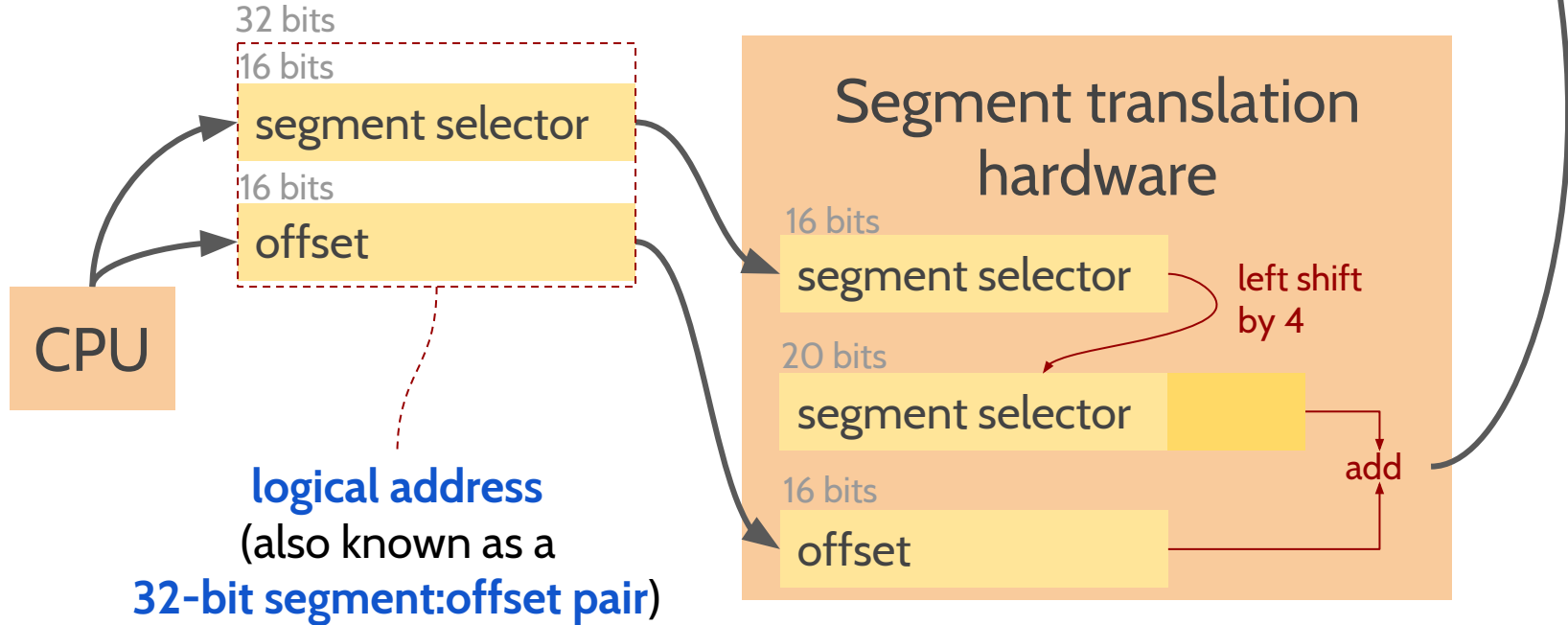
add

logical address
(also known as a **32-bit segment:offset pair**)

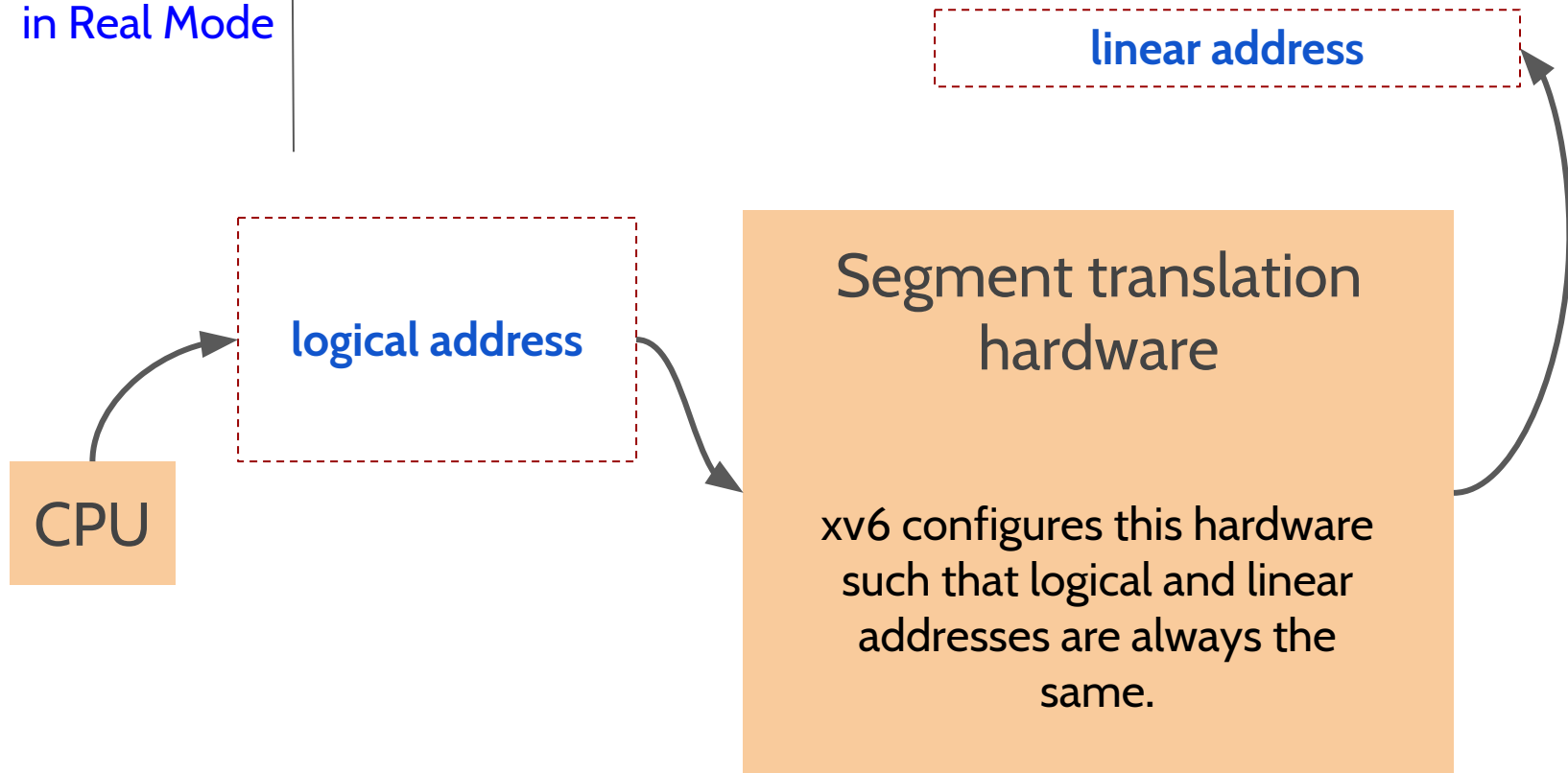


Segmentation in Real Mode

if paging is enabled, this address would go through a further translation process within the paging hardware to generate a physical address



Segmentation in Real Mode



Segmentation in Real Mode

*It follows (without paging) in
xv6,
logical address
= linear address
= physical address*

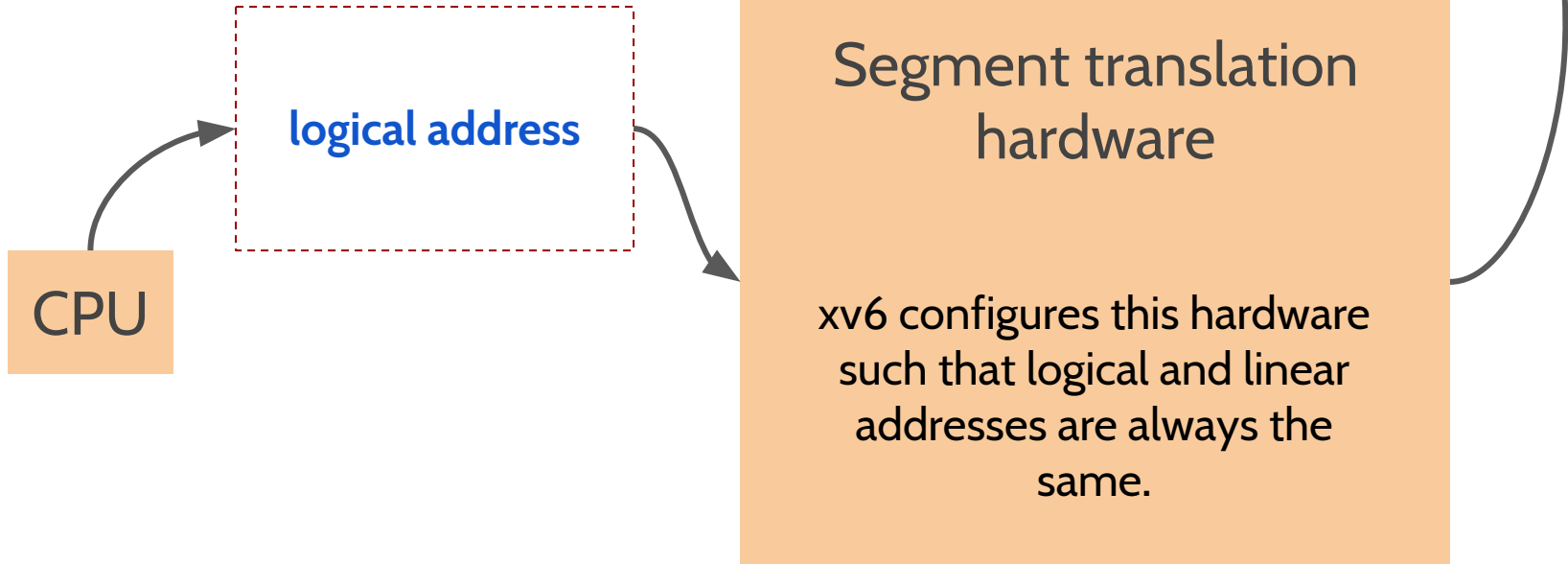
linear address

logical address

Segment translation
hardware

CPU

xv6 configures this hardware
such that logical and linear
addresses are always the
same.



Why the switch to Protected Mode?

With the processor being able to send **20 bit linear addresses**,
the maximum size of addressable
memory is 2^{20} bytes which is **only 1MB**.

In Protected Mode, the processor can send **32 bit linear addresses**,
which allows us to address a memory of size **4GB**.

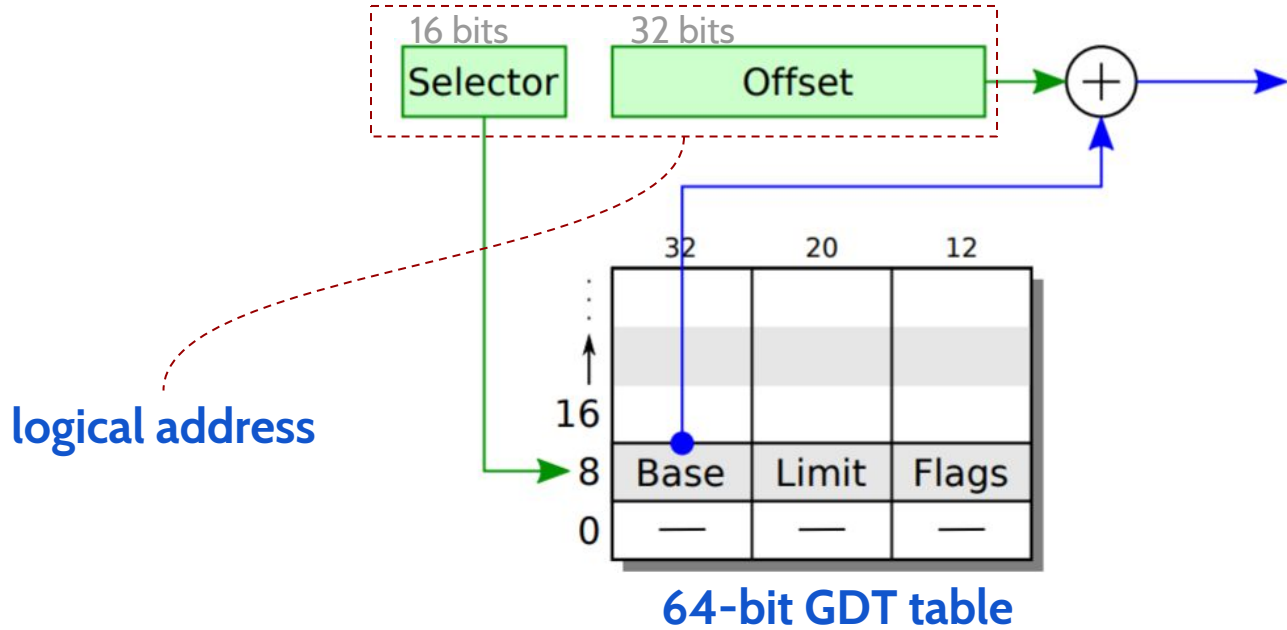
Paging hardware is also enabled in Protected Mode.

BIOS → xv6 Boot loader

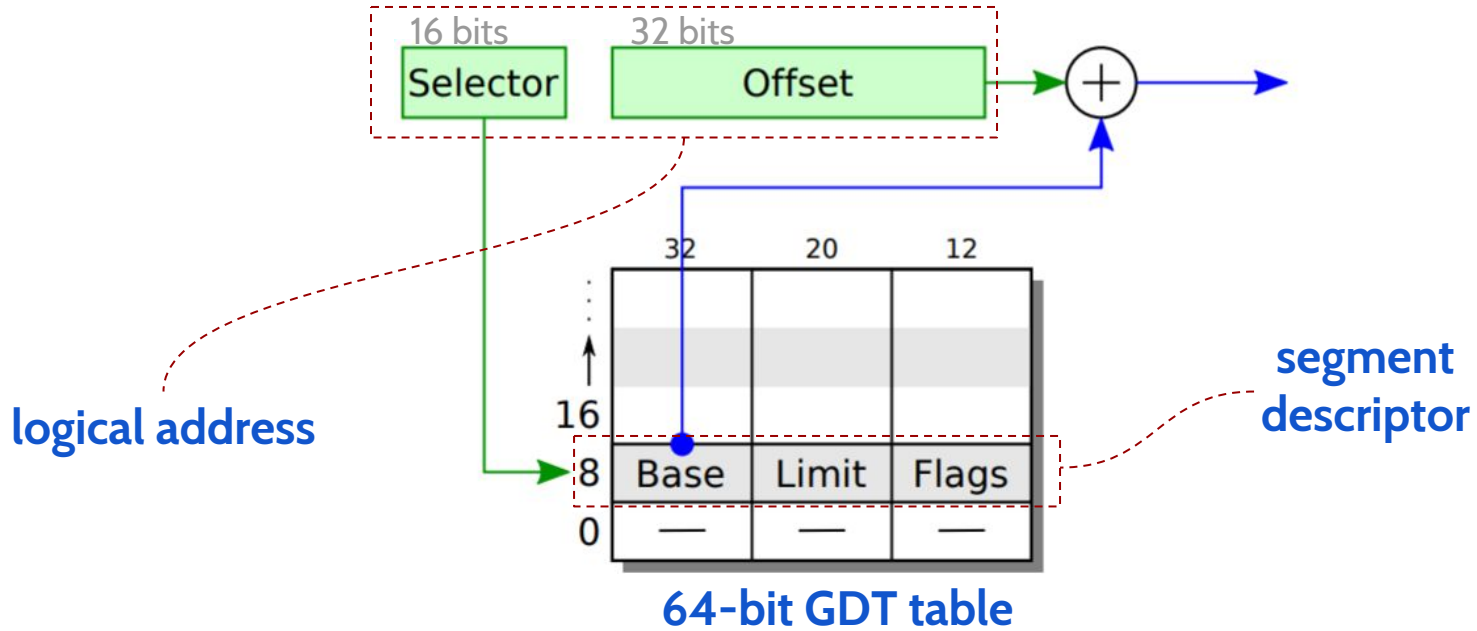
executing bootasm.S

2. **Switch from real mode to protected mode.**
 - > To setup the **protected mode**, the boot loader sets up the **segment descriptor table**, with three entries. Each entry is [a base physical address, max virtual address limit (4GB), permission bits]

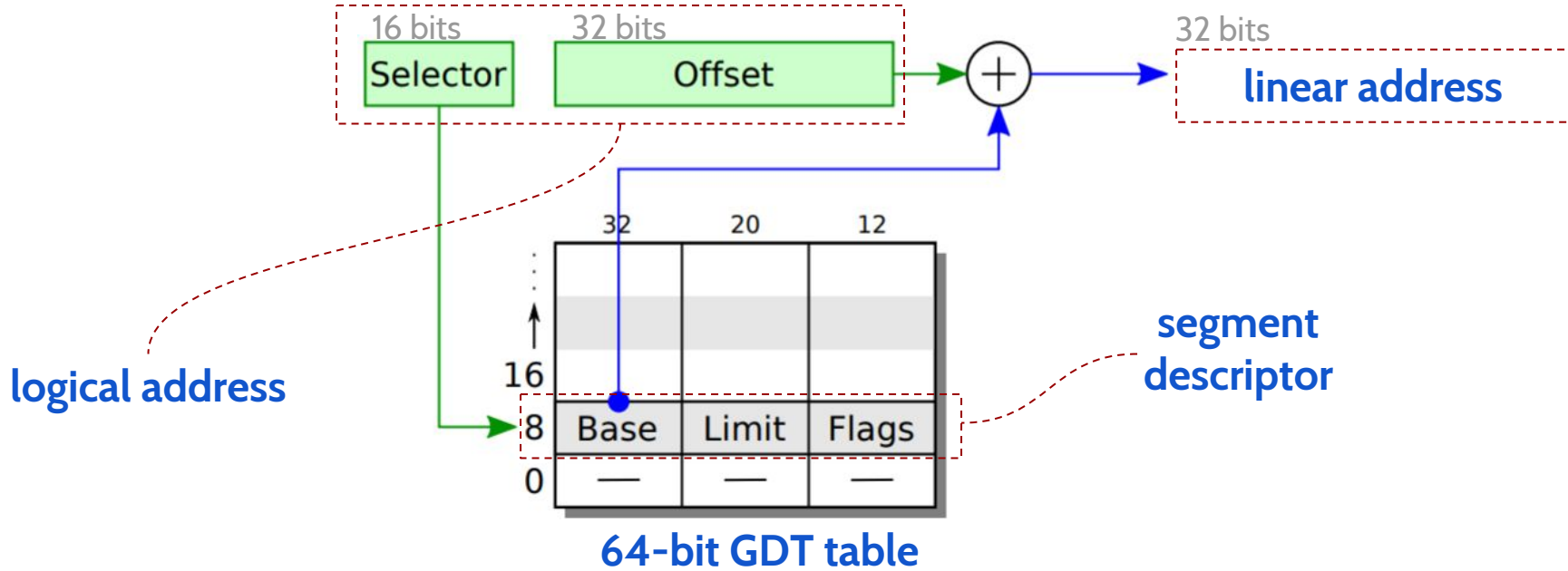
Segmentation in Protected Mode



Segmentation in Protected Mode



Segmentation in Protected Mode



BIOS → xv6 Boot loader

executing bootasm.S

```
lgdt    gdtdesc
```

```
gdtdesc:
```

```
.word   (gdtdesc - gdt - 1)    # sizeof(gdt) - 1
```

```
.long   gdt                    # address gdt
```

setting up segment
descriptor table

the gdt table

the macros
used are
defined here

BIOS → xv6 Boot loader

executing bootasm.S

Assembly Tip:

All assembler directives begin with a period. ([Ref.](#))

setting up segment descriptor table

```
lgdt    gdtdesc
```

```
gdtdesc:
```

```
.word   (gdtdesc - gdt - 1)    # sizeof(gdt) - 1
```

```
.long   gdt                    # address gdt
```

the gdt table

the macros
used are
defined [here](#)

BIOS → xv6 Boot loader

executing bootasm.S

2. **Switch from real mode to protected mode.**
 - > Now enable protected mode by setting the 1 bit in control register %cr0.
([Code](#)).

BIOS → xv6 Boot loader

executing bootasm.S

2. **Switch from real mode to protected mode.**
 - > To fully enable protected mode, we need to load a new value into %cs.
 - > Since %cs cannot be modified directly we setup %cs here: ([code](#)), which results in %cs to refer to the code descriptor entry in gdt.

BIOS → xv6 Boot loader

executing bootasm.S

Assembly Tip:

Long Jump:

```
ljmp $0xfebc, $0x12345678
```

Use 0xfebc for the CS register and 0x12345678 for the EIP register. ([Ref.](#))

2. Switch from real mode to protected mode.

- > To fully enable protected mode, we need to load a new value into %cs.
- > Since %cs cannot be modified directly we `ljmp (code)`, which results in %cs to refer to the code descriptor entry in gdt (a 32 bit code

seg.)

BIOS → xv6 Boot loader

**We have completed transitioning from 16 bit (real) mode to
32 bit (protected) mode**

BIOS → xv6 Boot loader

To see details on how the 32 bit addresses are translated in the segmentation hardware in protected mode look [here](#).

BIOS → xv6 Boot loader

The next steps in booting....

- bootasm.S makes necessary initializations to call bootmain.c, and calls bootmain
- bootmain loads the kernel into memory

thank you !