Crafting a ‘boot time’ program

How we can utilize some standard ‘real-mode’ routines that reside in the PC’s ROM-BIOS firmware

Our ‘bare machine’

- If we want to do a “hands on” study of our CPU, without any operating system getting in our way, we have to begin by exploring ‘Real Mode’ (it’s the CPU’s startup state)
- We will need to devise a mechanism by which our program-code can get loaded into memory (since we won’t have an OS)
- This means we must write a ‘boot loader’

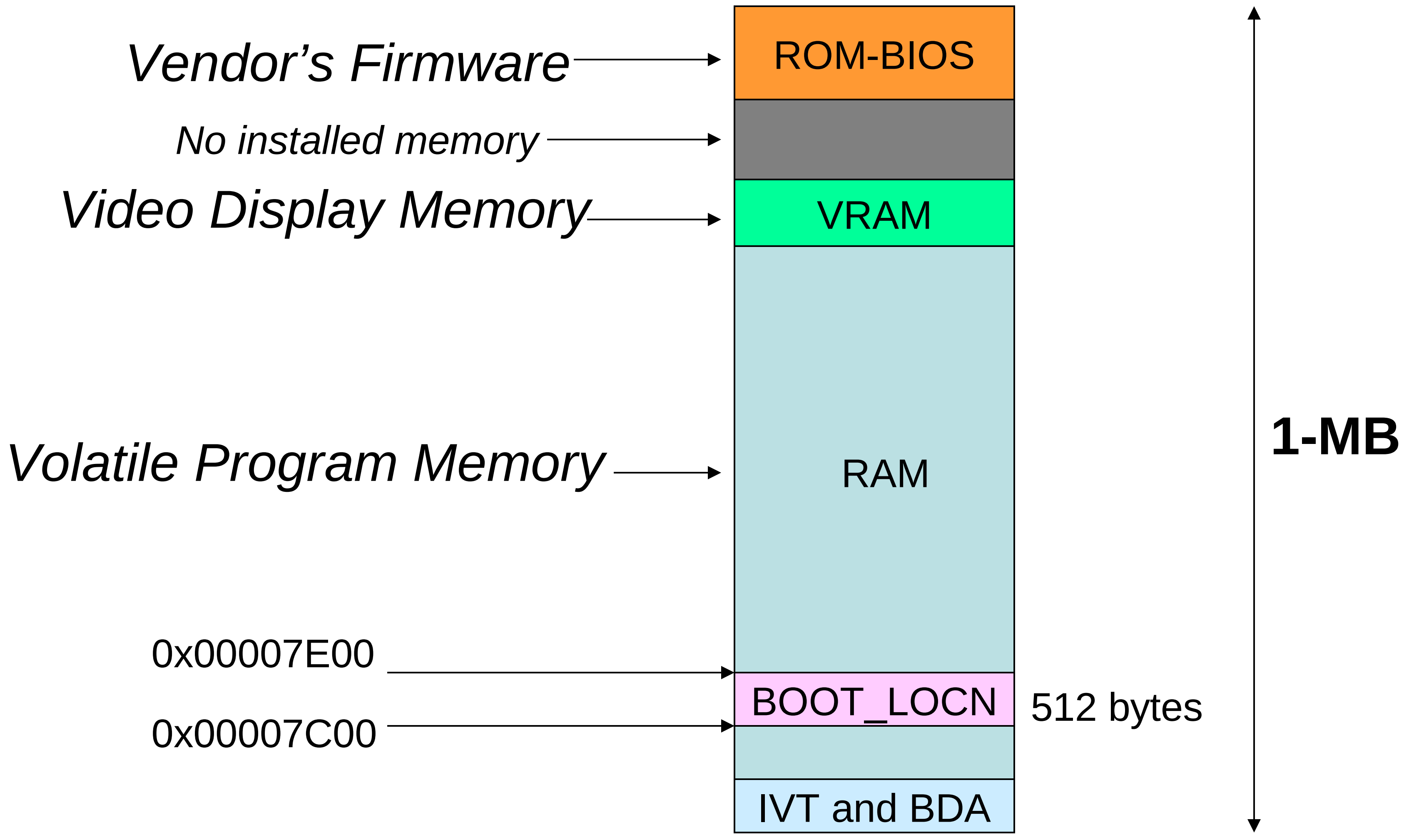
Our ‘bare machine’

- If we want to do a “hands on” study of our CPU, without any operating system getting in our way, we have to begin by exploring ‘Real Mode’ (it’s the CPU’s startup state)
- We will need to devise a mechanism by which our program-code can get loaded into memory (since we won’t have an OS)
- So we utilize the ROM-BIOS ‘boot loader’

What’s a ‘boot loader’

- After testing and initializing the machine’s essential hardware devices, the startup program in the ROM-BIOS firmware will execute its ‘boot loader’, to read a small amount of code and data into memory from some type of non-volatile storage medium (e.g, floppy-diskette, hard-disk, magnetic-tape, CD-ROM, etc.) and then ‘jump’ to that location in memory

PC ROM-BIOS BOOT_LOCN



8086 memory-map

Some Requirements

- A ‘boot loader’ has to be 512 bytes in size (because it has to fit within a disk sector)
- Must begin with executable machine-code
- Must end with a special ‘boot signature’
- Depending on the type of storage medium, it may need to share its limited space with certain other data-structures (such as the ‘partition table’ on a hard disk, or the Bios Parameter Block’ on a MS-DOS diskette)

Writing a ‘boot-sector’ program

- Not practical to use a high-level language
- We need to use 8086 assembly language
- We can use the Linux development tools:
  - The ‘vi’ editor
  - The ‘as’ assembler
  - The ‘ld’ linker
  - The ‘dd’ copy-and-convert utility

Using ROM-BIOS functions

- Our system firmware provides many basic service-functions that real mode programs can invoke (this includes ‘boot-loaders’):
  - Video display functions
  - Keyboard input functions
  - Disk access functions
  - System query functions
  - A machine ‘re-boot’ function

ROM-BIOS documentation

- The standard ROM-BIOS service-routines are described in numerous books that deal with assembly language programming for Intel-based Personal Computers
- They are also documented online (e.g., see the link on our class website to Ralf Brown’s Interrupt List)

A ‘typical’ calling convention

- A service-function ID-number is placed in the processor’s AH register
- Other registers are then loaded with the values of any parameters that are needed
- Then a ‘software interrupt’ instruction is executed which transfers control to the code in the Read-Only Memory (ROM)
- Often, upon return, the **carry-flag** is used for indicating that the function succeeded

Example: Write_String function

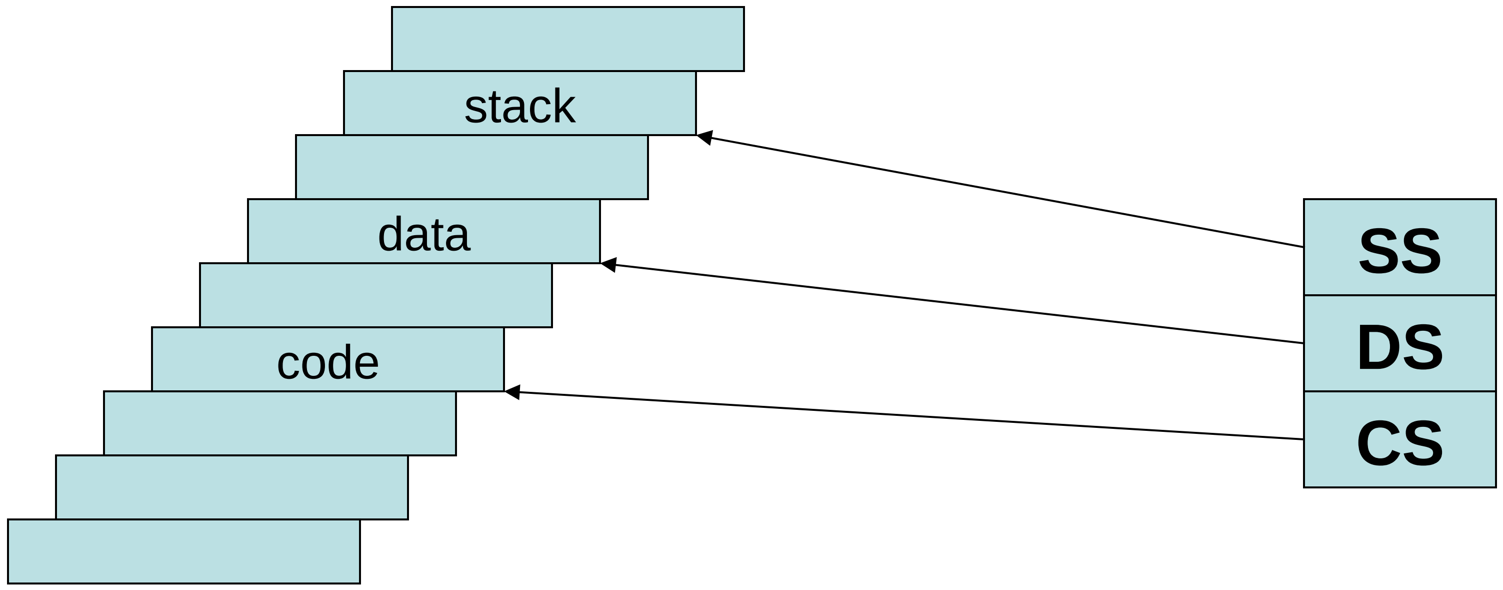
- Setup parameters in designated registers
  - AH = function ID-number (e.g. 0x13)
  - AL = cursor handling method (e.g. 0x01)
  - BH = display page-number (e.g., 0x00)
  - BL = color attributes (e.g., 0x0A)
  - CX = length of the character-string
  - DH, DL = row-number, column-number
  - ES:BP = string's starting-address (seg:off)
- Call BIOS via software interrupt (int-0x10)

Real Mode

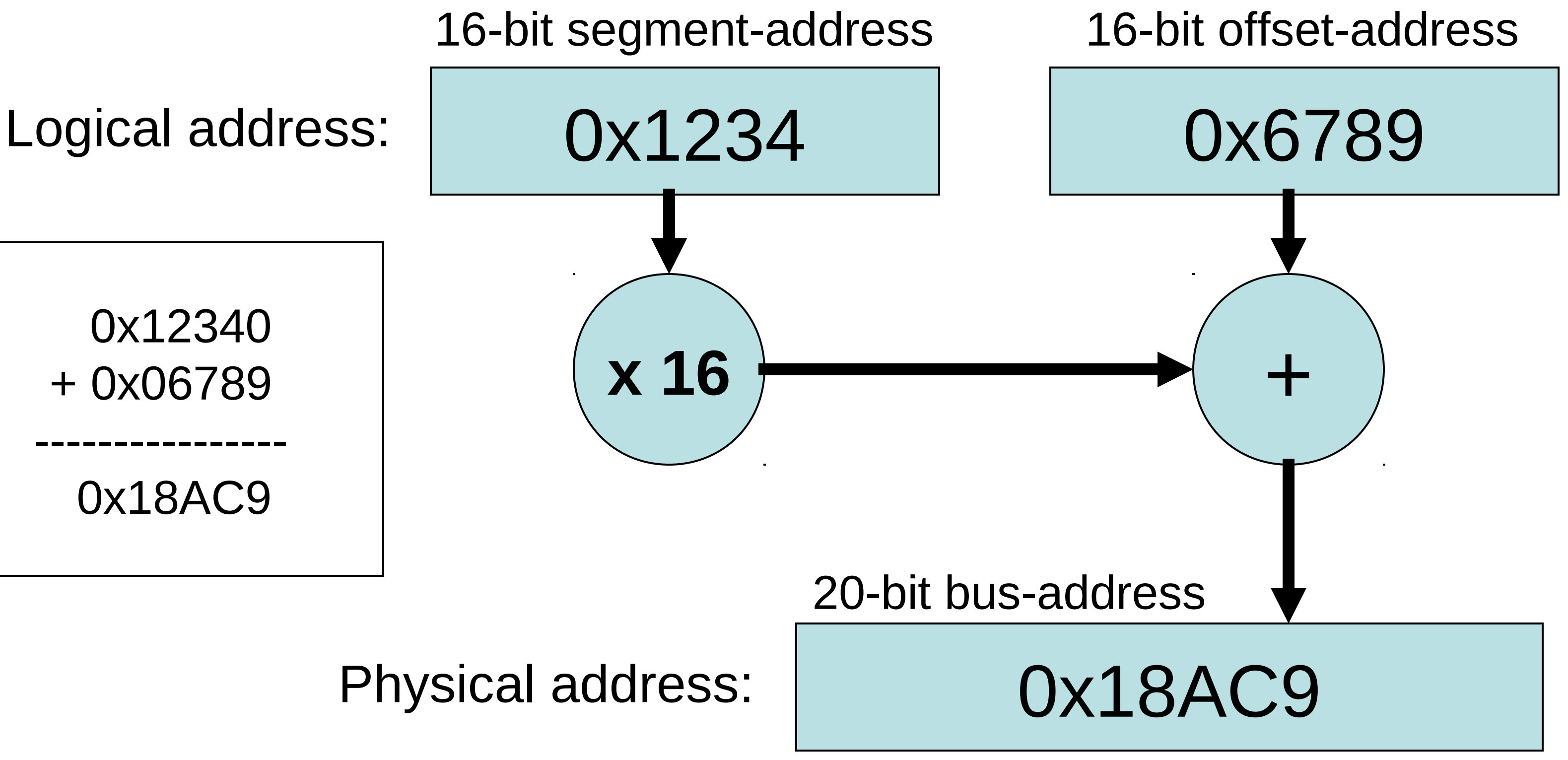
- 8086/8088 had only one execution mode
- It used “segmented” memory-addressing
- Physical memory on 8086 was subdivided into overlapping “segments” of fixed-size
- The length of any memory “segment” was fixed at 64KB until the processor has been switched out of its initial ‘real-mode’ and its segment-registers reprogrammed

64KB Memory-Segments

- Fixed-size segments partially overlap
- Segments start on paragraph boundaries
- Segment-registers serve as “selectors”

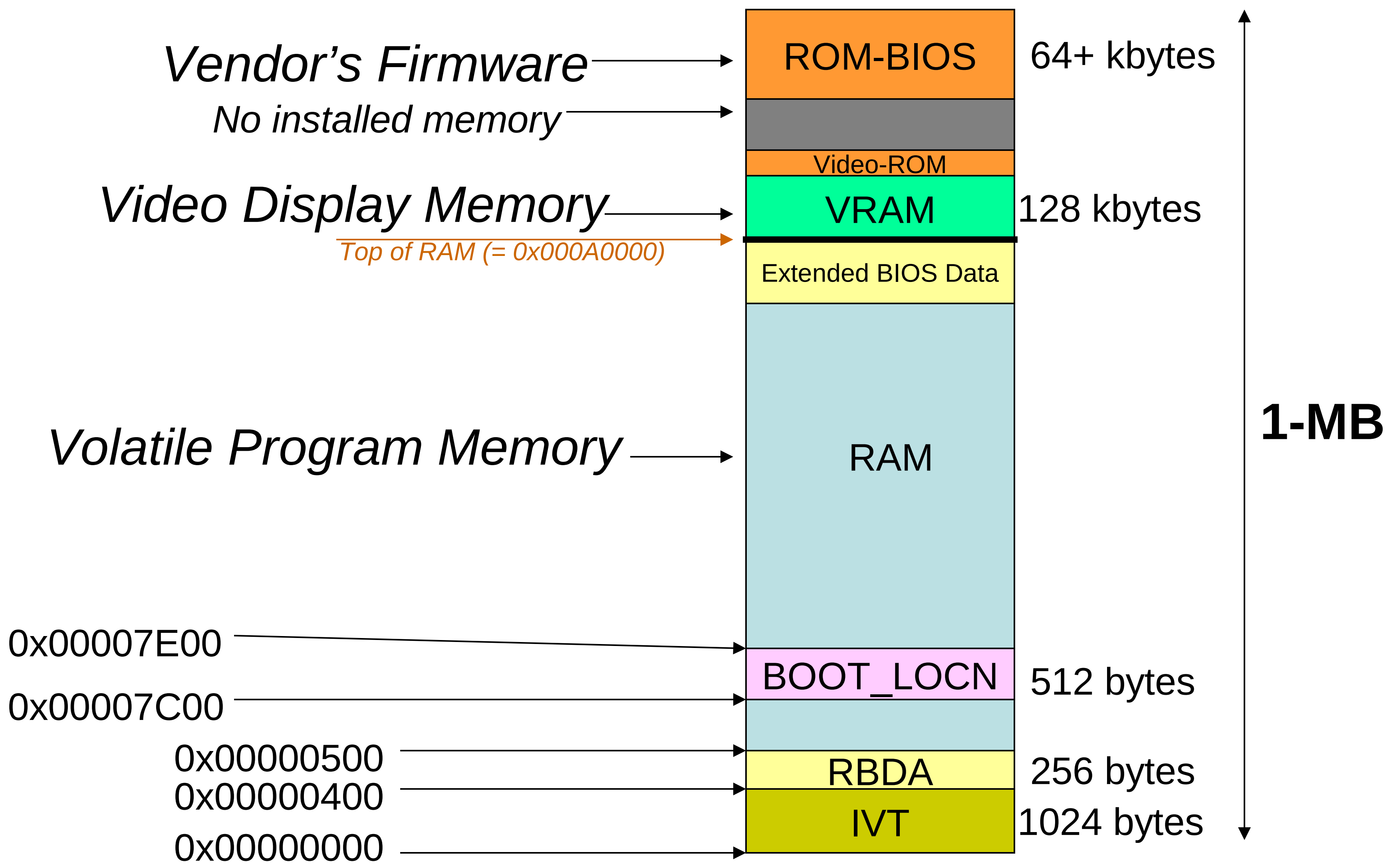


Real-Mode Address-Translation



```
   0x12340
 + 0x06789
----------------
   0x18AC9
```

‘Real-Mode’ Memory Map



Downloading a class demo

- You can ‘download’ a program source-file from our CS 630 course-website to your own ‘present working directory’ by using the Linux file-copy command, like this:

```
$ cp  /home/web/cruse/cs630/bootdemo.s  .
```

(Here the final period-character (‘.’) is the Linux shell’s symbol for your ‘current directory’).

Compiling and Installing

- Compiling our ‘bootdemo.s’ using ‘as’ is a two-step operation (and requires use of a linker-script, named ‘ldscript’):

```
$  as bootdemo.s  –o bootdemo.o
$  ld bootdemo.o  –T ldscript  –o bootdemo.b
```

- Installing our bootdemo.b into the starting sector of a hard-disk partition is very simple:

```
$  dd  if=bootload.b  of=/dev/sda4
```

Executing our ‘bootdemo’

- You’ll need to perform a system ‘reboot’
- Our classroom machines will load GRUB (the Linux GRand Unified Boot-loader)
- GRUB will display a menu of Boot Options
- You can choose ‘boot from a disk-partition’

In-class exercise

- Can you modify our ‘bootdemo.s’ program so that it will display some information that is more useful?
- For example, can you display the Vendor Identification String and the Brand String?
- (Remember, you code and data must fit within a single 512-byte disk-sector)
- You can use ‘fileview’ to check on space