CS 410/510
Languages & Low-Level Programming

Mark P Jones
Portland State University

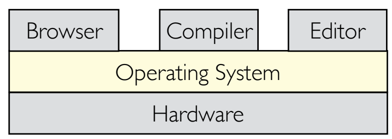
Fall 2018

Week 2: Bare Metal and the Boot Process

Bare Metal Programming

A conventional computing environment

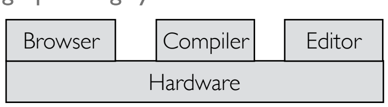
- The standard applications that we run on our computers do so with the support of an underlying operating system:



- These applications benefit enormously from the functionality that the operating system provides:
  - Memory management
  - I/O
  - File systems
  - Networking
  - etc…

A bare metal environment

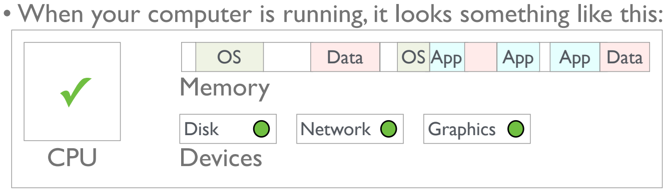
- What if we ran applications directly on the hardware, without an underlying operating system?



- Applications like this are said to be "running on bare metal"
  - Direct access to and manipulation of hardware
  - Potential to reduce complexity and cost
  - Less suited to general purpose computing …
  - Although conventional operating systems are bare metal systems that enable a general purpose environment ….

The boot process

- When your computer is running, it looks something like this:



- The CPU is initialized
- The memory contains the apps and data that we need
- The devices are initialized and operational
- How did that happen?

Initializing the CPU

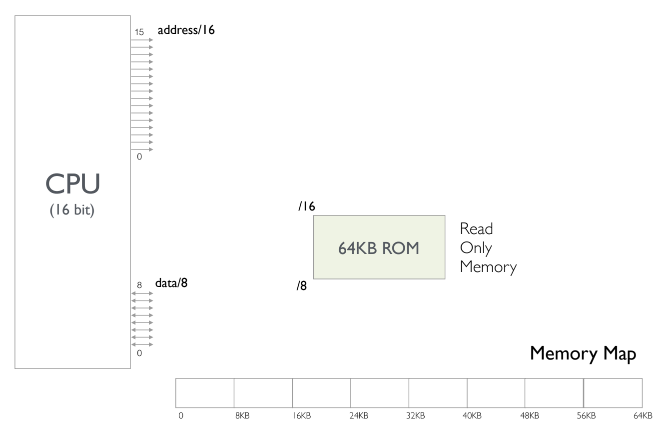
- The CPU will typically initialize itself when power is first applied or when the system is reset:
  - Basic self-test
  - Initialize registers to known states
  - … including the instruction pointer/program counter
    - On IA32, for example, execution starts at 0xFFFFFFF0
- So the computer can begin executing programs …
- And those programs can initialize the devices …
- But only if those programs are in memory!

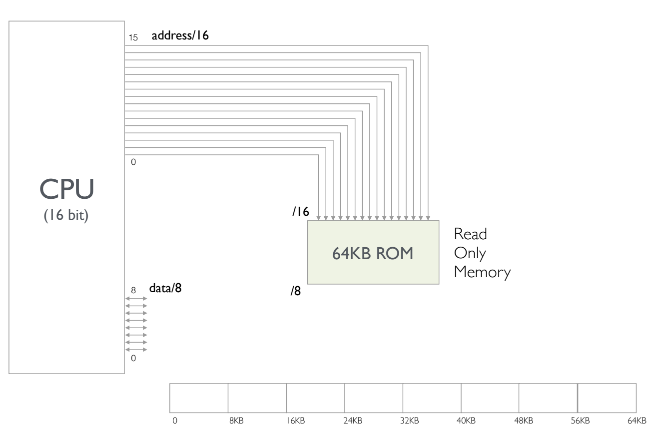
where does this information come from?

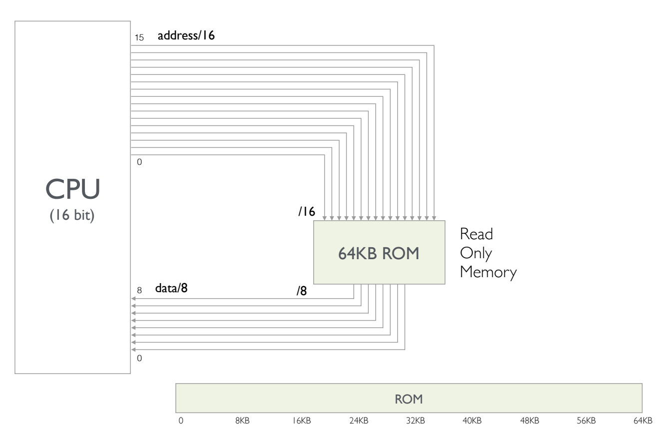
Building a Simple Computer System

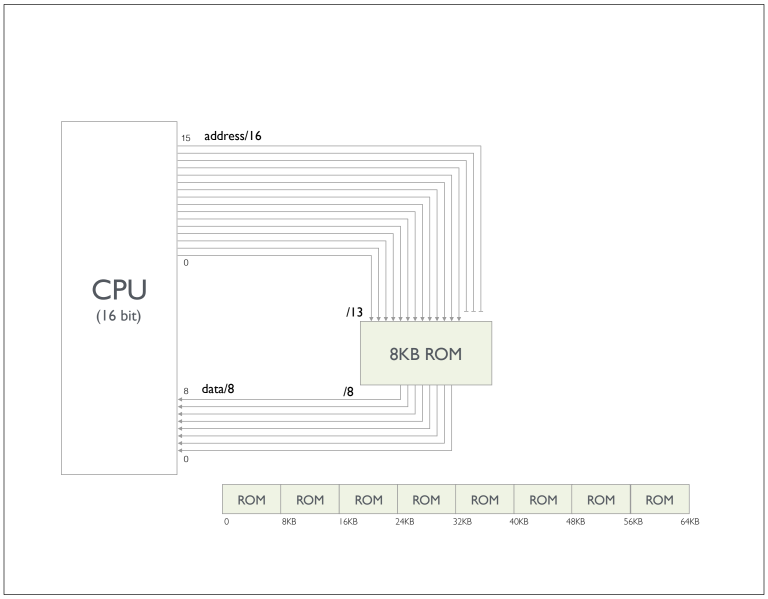
Building a basic computer system

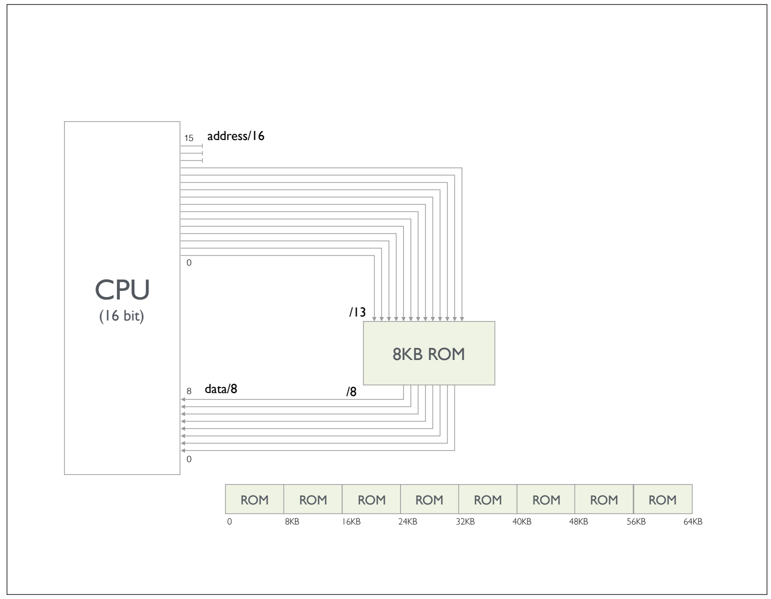
- Let's review some basic techniques that are used to construct a typical computer
- For the purposes of this exercise, we'll assume a 16 bit processor … but the same ideas apply to other architectures
- Key goal: understand how physical memory might be organized and addressed

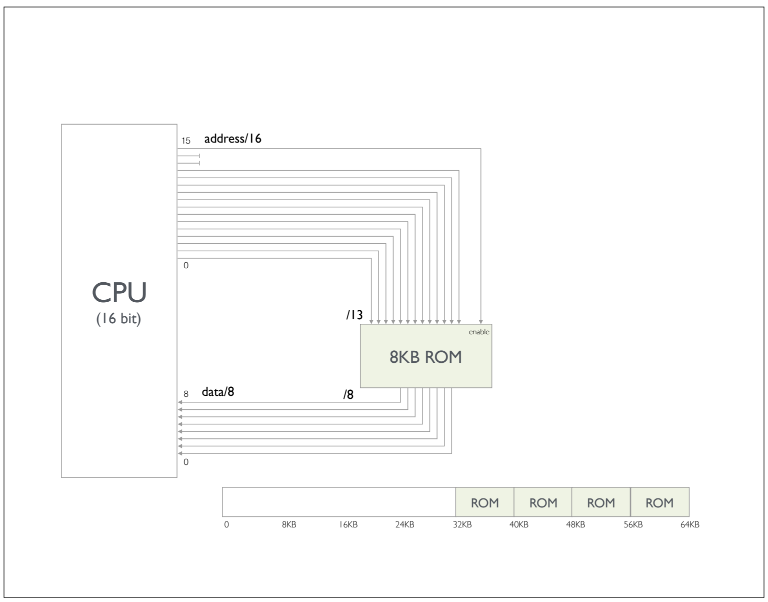


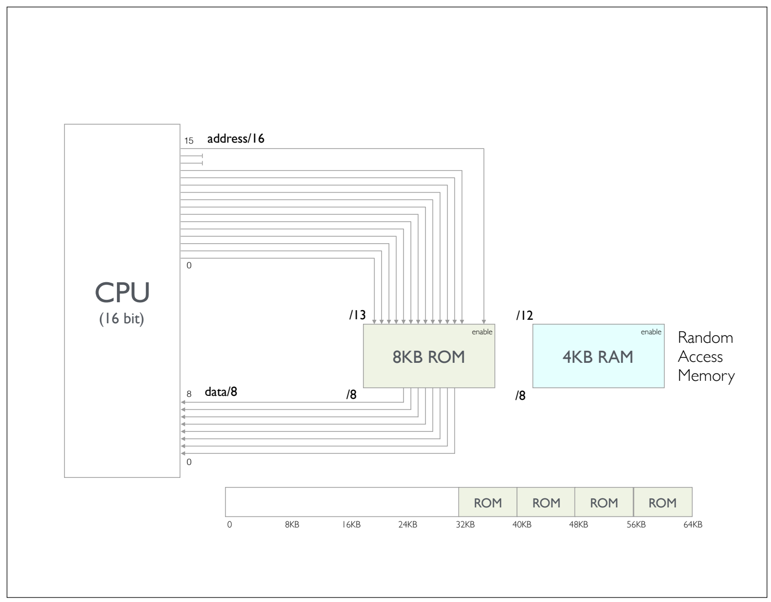


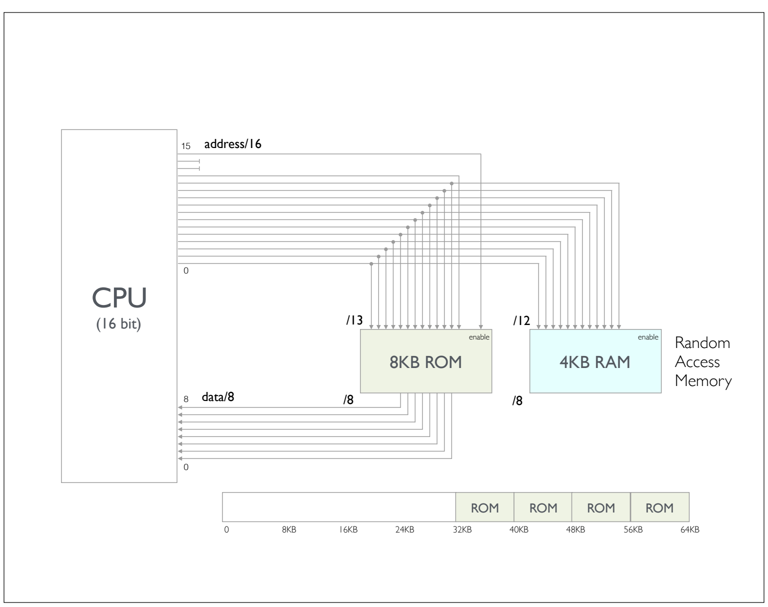


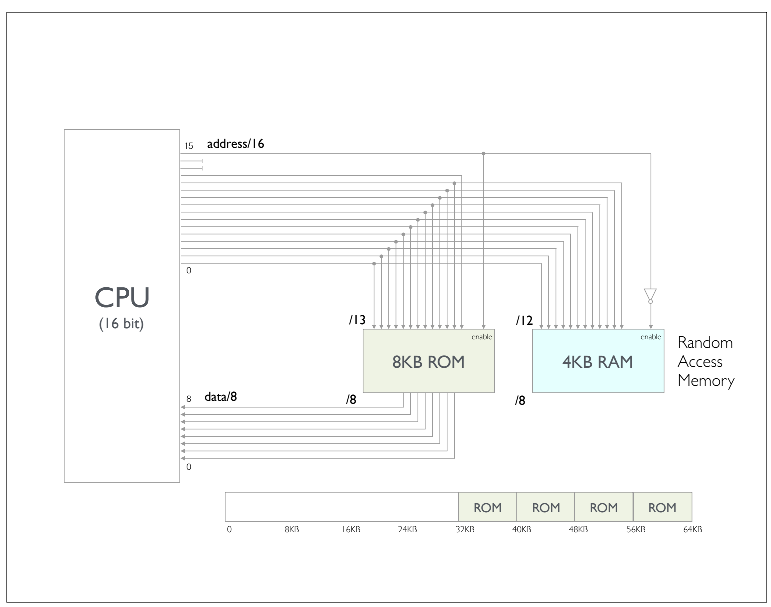

CPU
(16 bit)
15
address/16
0
/13
8KB ROM
8
data/8
/8
0
ROM ROM ROM ROM ROM ROM ROM ROM
0 8KB 16KB 24KB 32KB 40KB 48KB 56KB 64KB

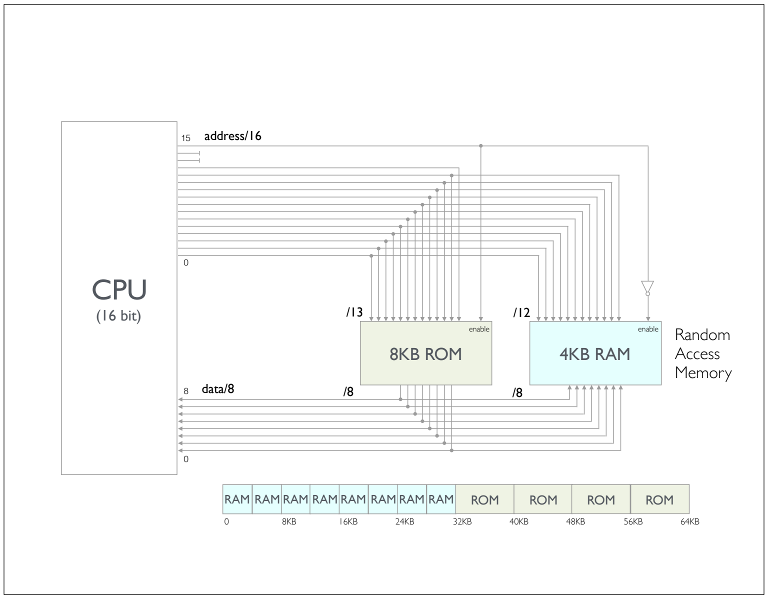

CPU
(16 bit)
15
address/16
0
/13
8KB ROM
8
data/8
/8
0
ROM ROM ROM ROM ROM ROM ROM ROM
0 8KB 16KB 24KB 32KB 40KB 48KB 56KB 64KB

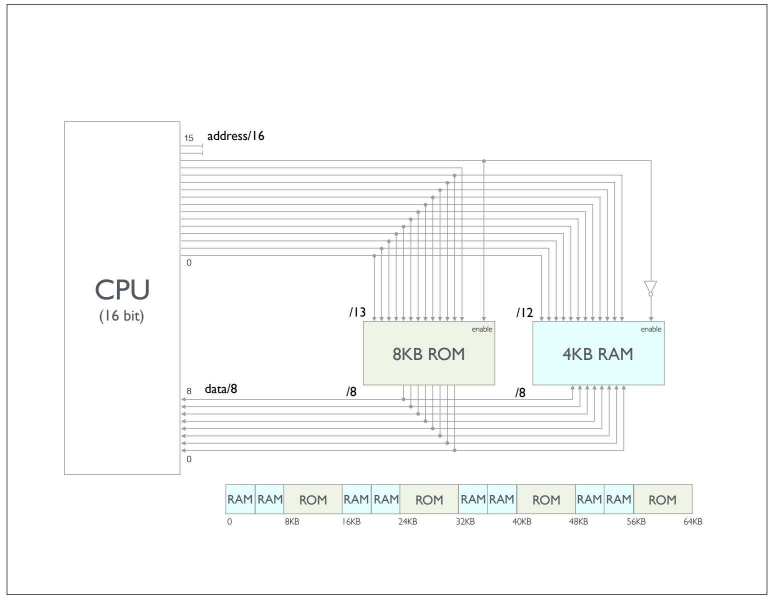

CPU
(16 bit)
15
address/16
0
/13
enable
8KB ROM
8
data/8
/8
0
ROM ROM ROM ROM
0 8KB 16KB 24KB 32KB 40KB 48KB 56KB 64KB

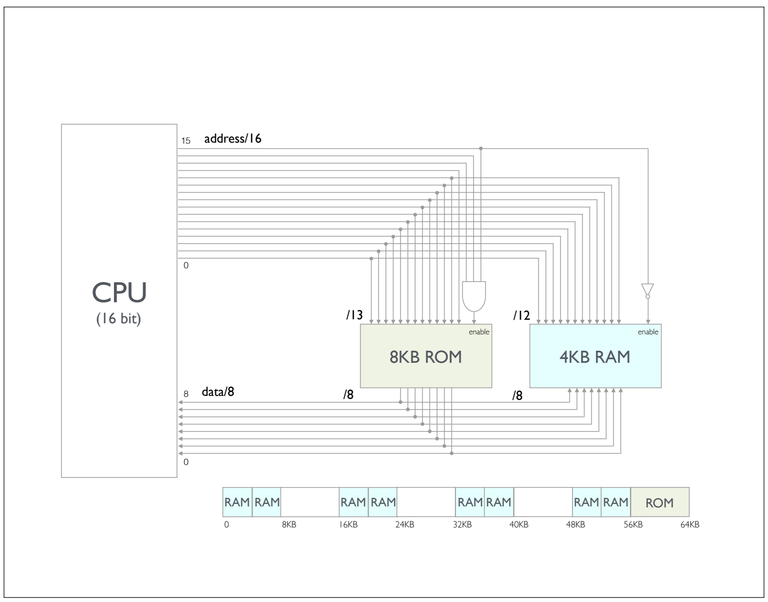

CPU
(16 bit)
15
address/16
0
/13
enable
8KB ROM
/12
enable
4KB RAM
Random
Access
Memory
8
data/8
/8
/8
0
ROM ROM ROM ROM
0 8KB 16KB 24KB 32KB 40KB 48KB 56KB 64KB

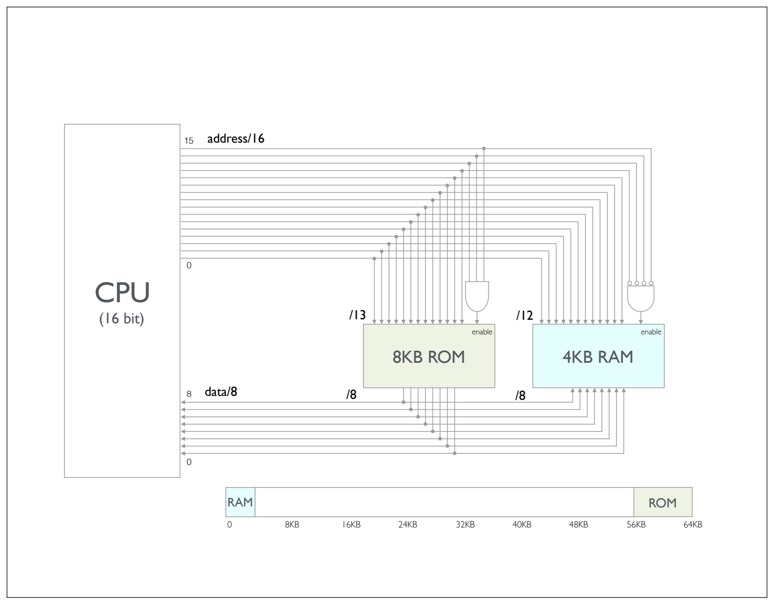

CPU
(16 bit)
15
address/16
0
/13
enable
8KB ROM
/12
enable
4KB RAM
Random
Access
Memory
8
data/8
/8
/8
0
ROM ROM ROM ROM
0 8KB 16KB 24KB 32KB 40KB 48KB 56KB 64KB

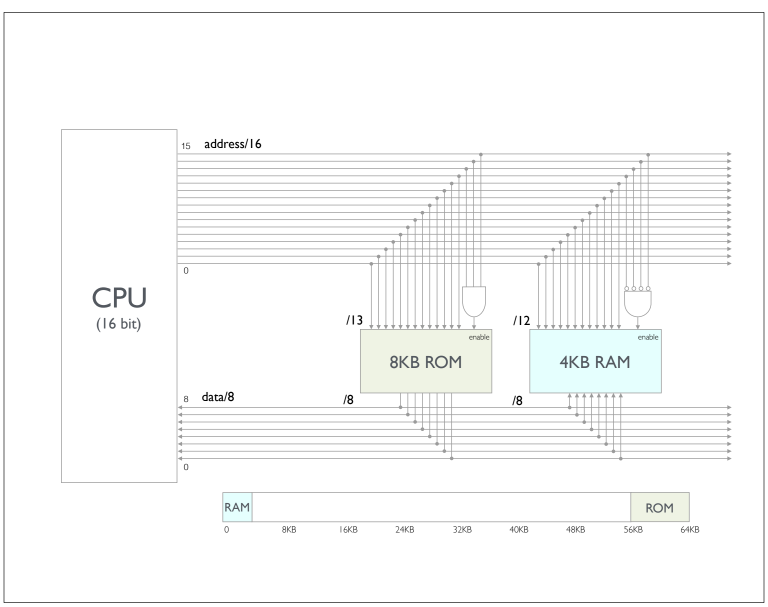

CPU
(16 bit)
15
address/16
0
/13
enable
8KB ROM
/12
enable
4KB RAM
Random
Access
Memory
8
data/8
/8
/8
0
ROM ROM ROM ROM
0 8KB 16KB 24KB 32KB 40KB 48KB 56KB 64KB

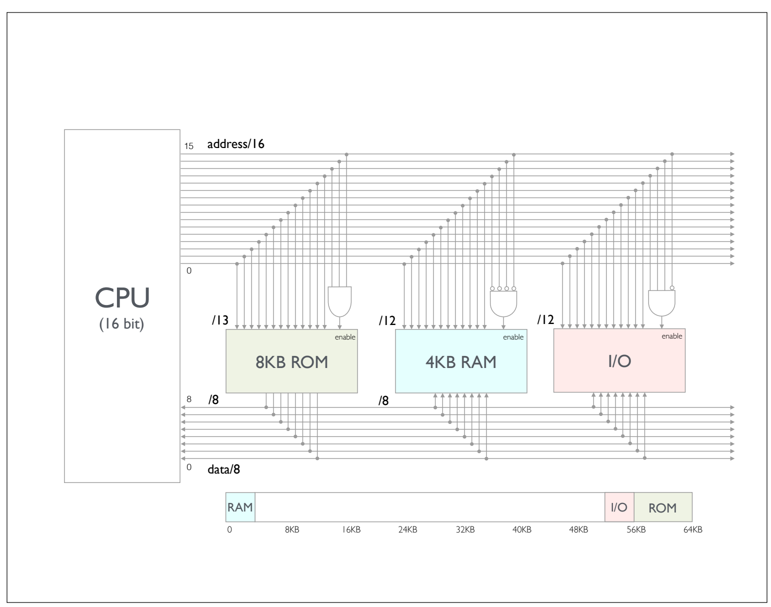

CPU
(16 bit)
15
address/16
0
/13
/12
8KB ROM
enable
4KB RAM
enable
Random
Access
Memory
8
data/8
/8
/8
0
RAM RAM RAM RAM RAM RAM RAM RAM ROM ROM ROM ROM
0 8KB 16KB 24KB 32KB 40KB 48KB 56KB 64KB

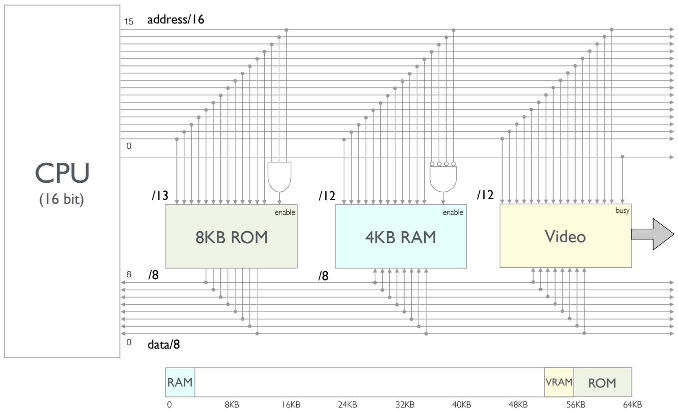

CPU
(16 bit)
15
address/16
0
/13
/12
8KB ROM
enable
4KB RAM
enable
8
data/8
/8
/8
0
RAM RAM ROM RAM RAM ROM RAM RAM ROM RAM RAM ROM
0 8KB 16KB 24KB 32KB 40KB 48KB 56KB 64KB


CPU
(16 bit)
15
address/16
0
/13
/12
8KB ROM
enable
4KB RAM
enable
8
data/8
/8
/8
0
RAM RAM RAM RAM RAM RAM RAM RAM ROM
0 8KB 16KB 24KB 32KB 40KB 48KB 56KB 64KB


CPU
(16 bit)
15
address/16
0
/13
/12
8KB ROM
enable
4KB RAM
enable
8
data/8
/8
/8
0
RAM ROM
0 8KB 16KB 24KB 32KB 40KB 48KB 56KB 64KB


CPU
(16 bit)
15
address/16
0
/13
/12
8KB ROM
enable
4KB RAM
enable
8
data/8
/8
/8
0
RAM ROM
0 8KB 16KB 24KB 32KB 40KB 48KB 56KB 64KB


CPU
(16 bit)
15
address/16
0
/13
/12
/12
8KB ROM
enable
4KB RAM
enable
I/O
enable
8
/8
/8
0
data/8
RAM I/O ROM
0 8KB 16KB 24KB 32KB 40KB 48KB 56KB 64KB



Booting a PC

Introducing the BIOS (Basic I/O System)

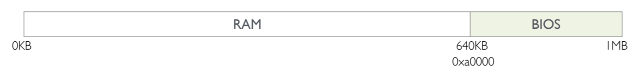
- The original IBM PC had a 20 bit address bus, so it could address up to 1MB of data:

- The CPU starts executing programs at an address close to the top of the address space …
- … so we can install a ROM at that address:

- The ROM contains the BIOS, or basic I/O system, for the computer

… continued

- The rest of the address space can be used for RAM (who would need more than 640KB, eh?):

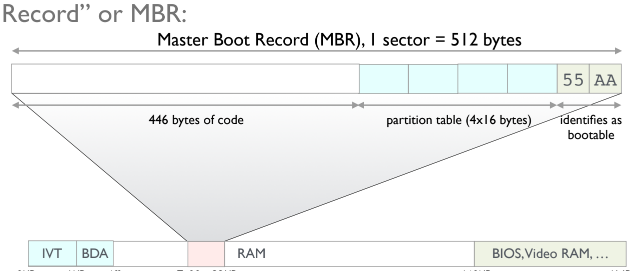


- Video RAM is also mapped within the region above 640KB (at address 0xb8000), so it doesn't interfere with lower memory:

- But the BIOS does need to use some of that memory for its own purposes:

The master boot record (MBR)

- We don't want the BIOS to make too many assumptions about the operating system that it is booting
- Instead, the BIOS searches the available hardware for a "bootable" disk that contains a 512 byte "Master Boot Record" or MBR:



… booting, continued

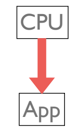
- Now the program from the MBR can continue the process of loading the rest of the operating system …
- … taking advantage of BIOS routines …
- … but without relying on a BIOS that is hardwired to that particular OS

Boot loaders

Boot scenarios

- Custom hardware

- Simple, single purpose programmed system, app in ROM

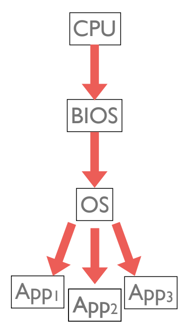


- Single purpose programmed system, app on disk or other media

… continued

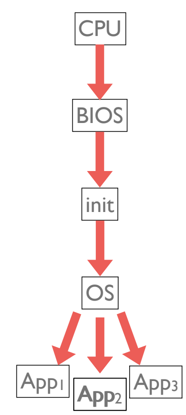
- App on disk or media, leveraging an underlying operating system

- … possibly supporting multiple applications …



… continued

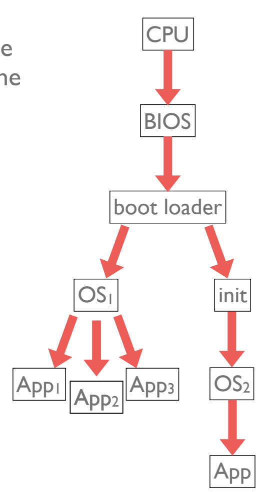
- Boot time configuration that is not required once the system is properly initialized
- Typical uses:
  - initialize and test a device
  - decrypt/decompress a file system
  - free resources (e.g., memory) that are not required once the system is booted



… continued

- Potential to boot into one of multiple operating systems, selected at runtime

- The role of a boot loader is to:
  - prepare the next stage to run (includes selecting between multiple possible “next stages”)
  - collect and pass on configuration details

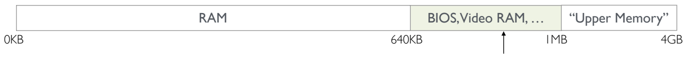


Introducing GRUB

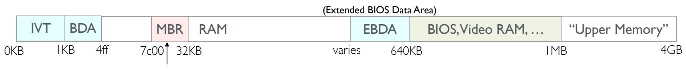
(The GNU **GR**and **U**nified **B**ootloader)

Booting via GRUB

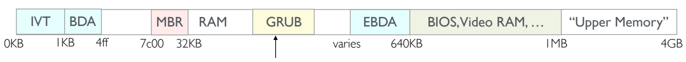
- After reset, the CPU starts executing code in the BIOS ROM



- The BIOS loads and transfers control to the MBR code

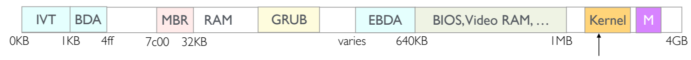


- The MBR code loads GRUB from a known location on the disk (using BIOS routines)



… continued

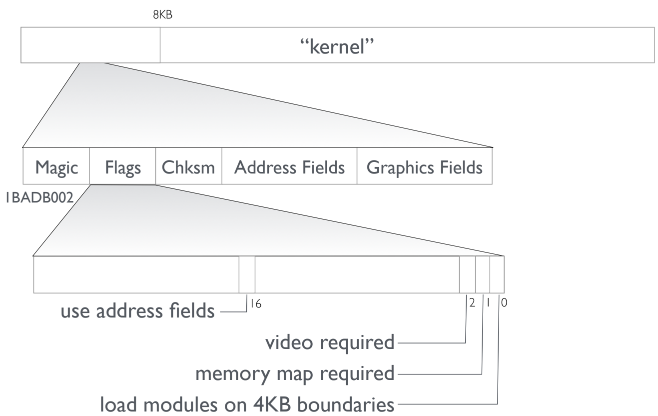
- The main GRUB program (interpreting the higher-level file system on the boot disk) searches for a configuration file, reading and acting on its contents
- Once a boot option has been identified (possibly with user input), GRUB will load an appropriate "kernel" file, together with a sequence of zero or more "modules", in to memory and then transfer control to the kernel



- The kernel begins the process of initializing the OS/App/…

Details: Loading the kernel

- The kernel must contain a "multiboot header" in the first 8KB



Where should the kernel be loaded?

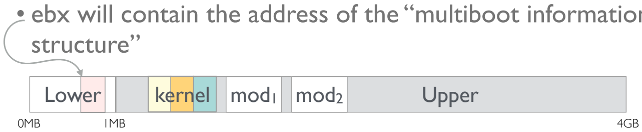
- GRUB is able to parse kernel files in ELF format, and will load the different sections of the file in to the appropriate addresses

7F	E	L	F	Header	$Section_1$	$Section_2$	$Section_3$

- If the kernel is not in ELF format, then flag bit 16 must be set and the address fields must be used to specify where the kernel will be loaded
- Either way, GRUB will not allow the kernel to be loaded below 1MB (so GRUB is free to use that memory)
- End result:

Kernel in control!

- GRUB's work is done, and it jumps to the specified entry point for the kernel:
  - eax will contain 0x2BADB002
  - ebx will contain the address of the "multiboot information structure"

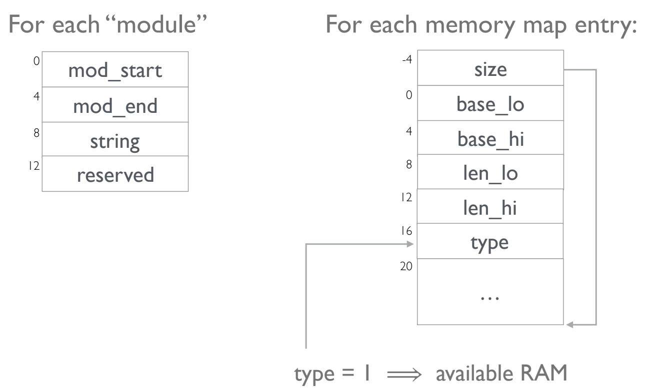


  - Values in other registers are also set to appropriate constant values, as described by the multiboot specification
- What will the "kernel" do next?

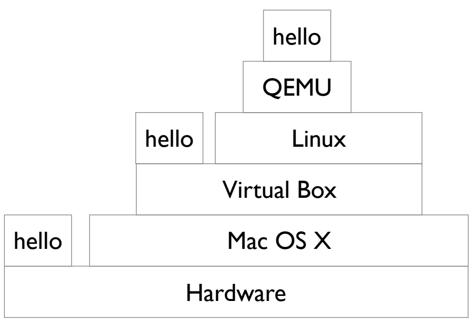
Multiboot Information

0	flags	
4	mem_lower	lower memory in KB (if flags[0])
8	mem_upper	upper memory in KB (if flags[0])
12	boot_device	
16	cmdline	pointer to command line string (if flags[2])
20	mods_count	number of modules (if flags[3])
24	mods_addr	address of first module descriptor (if flags[3])
28	symbols	
44	mmap_length	length of memory map buffer (if flags[6])
48	mmap_addr	address of first memory map entry (if flags[6])
52	etc…	

Multiboot Information, continued



Let's look at this in practice …



Introducing mimgload and mimgmake

GRUB is great …

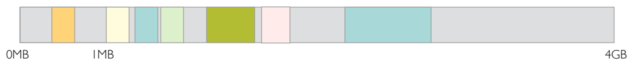
- It can load a "kernel" in one of several executable formats, as well as a collection of uninterpreted "modules"
- It supports booting from a variety of different media and file systems
- It supports network booting
- It can load from compressed kernel/module images
- It provides a boot-time menu and allows customization
- It gathers useful data about the machine and makes it available to the kernel
- Widely used, "multiboot standard", open source, …

But, of course, it has limits too …

- It can only load one executable
  - Possible workarounds include merging multiple ELF files into a single file, or using a kernel that can unpack executables from modules …
- The address at which modules are loaded cannot be controlled or predicted
- The location of the multiboot information structures is not specified, and is not even guaranteed to be stored in a contiguous block of memory
- There are limits on where GRUB can load data (e.g., it does not appear to be able to load into lower memory)

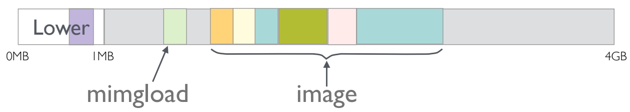
Memory Images

- Think about what we want the memory layout to look like immediately after the boot process completes:



- Package up those components in a (compressed) module:

- Boot from GRUB into a small program that can unpack the image, move the pieces to the required locations (including boot data), and transfer control to the main program:



mimgmake and mimgload

- mimgmake builds image files (in a full Linux environment):

```
../mimg/mimgmake image             \
        noload:../mimg/mimgload \
        bootdata:0x0000-0x3fff  \
        $(KERNEL)/pork          \
        user/sigma0/sigma0      \
        user/l4ka-pingpong/pingpong
```

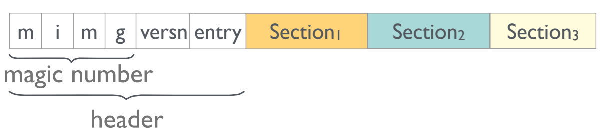
- mimgload loads images (on bare metal):

```
menuentry "InsertKernelNameHere" {
  multiboot /mimgload
  module    /image.gz
}
```

- No particular claim to originality: this was just a tool that I built as a learning experience/to meet a practical need

The mimg file format

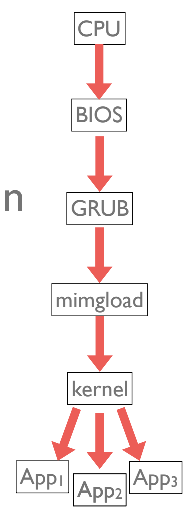
- Memory images are stored as binary files using a simple format that is like a greatly simplified version of ELF:



- Individual sections:

  - if type is DATA (1) or BOOTDATA(2), payload will contain (last-first+1) bytes
  - if type is ZERO (0), or RESERVED (3), payload is empty

A quick look at mimg in practice



Exercises

- Add a function to the code for "hello" that can be used to output an integer value (hexadecimal notation is probably easiest, and most useful too). Test to make sure it works correctly
- Integrate your assembly code for cls into "hello" …
- Adapt the code from "hello" or "bootinfo" to print out a summary of the details that GRUB passes on to the "kernel" via the multiboot information structure. (Start simple, and add more fields as you go.)
- Experiment with different virtual machine settings to see what impact this has on the information in the multiboot structure.