CS221
Irvine – Chapter 2

Let's switch gears a little bit and touch on a few items specific to the Intel 8086-based computers. Shortly we will discuss Intel assembly language in more detail.

General Concepts

Chapter 2.1 reviews general concepts regarding computer architecture. We have already discussed this in more detail using the Null and Lobur textbook, but it is a good idea to review chapter 2.1 to get an idea of these concepts.

Intel Hardware and Software Architecture – 8086

The Intel family of microprocessors is quite diverse. For the majority of this class we will focus on the 32 bit Intel processor. However we will look a little bit at older versions of the chip x86 family of chips and see how the design of the original chip affected the design of later chips. The latest incarnation, the Itanium, is the first of a completely new architecture that emulates the 8086.

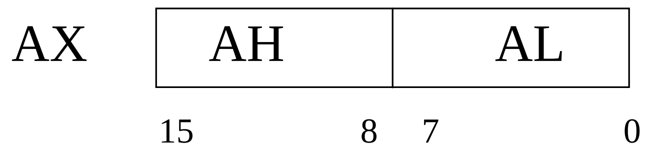
The first CPU in the Intel family is the 8086. The registers inside the 8086 are all 16 bits. They are split up into four categories: General Purpose, Index, Status & Control, and Segment.

General Purpose Registers

The four general purpose registers are the AX, BX, CX, and DX registers.

- AX - accumulator, and preferred for most operations.
- BX - base register, typically used to hold the address of a procedure or variable.
- CX - count register, typically used for looping.
- DX - data register, typically used for multiplication and division.

All of the general purpose registers can be treated as a 16 bit quantity or as two 8 bit quantities. The high byte is referenced by replacing the X with H. The low byte is referenced by replacing the X with L:



For example, putting 0xF100 (the leading 0x indicates what follows is in hex) into AX is the same as putting F1 into AH, and then 00 into AL. We can reference DX in terms of DH, DL, CX in terms of CH, CL, and BX in terms of BH, BL as well.

Segment Registers

The CPU contains four segment registers, used as base locations for program instructions, data, or the stack. In fact, all references to memory on the IBM PC involve a segment register as a base location.

The registers are:

- CS – Code Segment, base location of program code
- DS – Data Segment, base location for variables
- SS – Stack Segment. Base location of the stack
- ES – Extra Segment. Additional base location for variables in memory.

Index Registers

The index registers contain offsets from a segment register for information we are interested about. There are four index registers:

- BP – Base Pointer, offset from SS register to locate variables on the stack
- SP – Stack Pointer, offset from SS register as to the location of the stack's top
- SI – Source Index, used for copying strings, segment register varies
- DI – Destination Index, used for destination for copying strings

Calculating Addresses

How are these registers used to calculate addresses? On the Intel 8086, there is a 20 bit address bus giving us the capability to address up to 1MB of memory. However, registers are only 16 bits long. This means we can only address 64K of memory using a 16 bit register, leaving the entire 1Mb of memory seemingly un-addressable.

To solve this problem, the Intel engineers designed the CPU so that it computes 20 bit *absolute* addresses by combining the segment with the offset (i.e. index).

Segmented addresses are represented by:

BaseSegment:Offset

e.g.: 08F0:0010

Absolute addresses are represented directly by the hex representing the 20 bits:

e.g.: 0AB41

How do we compute an absolute address given a segmented address? Note that in the segmented address we have a total of 32 bits. This is an additional 12 bits than we really need! The scheme to compute the absolute address is to multiply the segment by 16 (i.e. slide it over four bits to the left) and then add the result to the offset:

e.g. given 08F1:0010
  here the segment is 08F1
  multiply by 16 or shift four bits / one hex digit to the left, inserting a zero on the right:
  giving: 08F10
  now add to the offset:

```
        08F10
 +        0010
------------------
        08F20          ← Absolute address
```

By sliding the segment over by four bits, we get a total addressable space of 20 bits. Note that there is **overlap** among segments:

```
        08F10                08F00
+         0000         +       0010
------------------     ------------------
        08F10                08F10
```

That is, 08F1:0000 and 08F0:0010 both reference the same absolute address.

You might wonder why such a complicated addressing method is used. One reason is surely to confuse CS students. Another is that this allows the CPU to access the 20 bit address bus using the 16 bit registers. A benefit of the scheme is that it becomes compact and easy to reference data within a single segment. Another benefit is that it allows programs to be loaded into any segment address without having to recalculate the individual addresses of variables – they are merely references as offsets from the segment. Finally, programs can access large data structures by gradually modifying the segment portion of the address to access large blocks of memory. In this case, the overlap feature of the segment becomes useful. A downside of the scheme, in addition to the complexity, is the confusion that can result if a program needs to access data beyond a 64K block of memory.

When we visit the 32-bit Intel processor architecture, this problem goes away and we have a much cleaner addressing system.

Status and Control Registers

The last set of registers is the Instruction Pointer (i.e. Program Counter) and the Flags:
  IP – Instruction pointer, offset from the CS for the next instruction to execute
  Flags – contains status flags

These flags are:

Direction: Used for block transfer of data, may be 0=up or 1=down
Interrupt: 1 = enabled, 0=disabled
Trap: Determines if the CPU should halt after each instruction. 1=on, 0=off.
Carry: Set if the last unsigned arithmetic operation had a carry
Overflow: Set if the last signed arithmetic operation overflowed
Sign: Set if the last arithmetic operation was negative. 1=negative, 0=positive
Zero: Set if the last arithmetic operation generated a result of zero. 1=zero
Aux Carry: Set when a carry from bit 3 to bit 4
Parity: Used for a parity check, number of 1's

Operating System and Memory

The 1Mb of accessible memory in the 8086 ranges from 00000 to FFFFF. RAM occupies 0000 – BFFFF. ROM occupies C0000 to FFFFF. DOS uses the first 640K of memory, 0000 to 9FFFF.

Memory usage is illustrated in the table below:

Address	**Contents**
00000	Interrupt Vector Table
00400	DOS Data
	Software BIOS
	DOS Kernel Device Drivers
	COMMAND.COM
	Available to programs
9FFFF	Used by COMMAND.COM
A0000	Video Graphics Buffer
B8000	Text Buffer
C0000	Reserved
F0000	ROM BIOS

The interrupt vectors are used to process hardware and software interrupts. This comprises the first 1024 bytes of memory. These locations store addresses of interrupt handlers for the respective routines.

The BIOS (Basic Input/Output System) includes routines for managing the keyboard, screen, clock, detecting PlugAndPlay devices, and other hardware devices. This is typically done through firmware – software burned onto the BIOS chip as hardware.

The DOS kernel includes services for disk and program functions, such as loading and running user programs. Above the kernel are the resident parts of MS-DOS to interpret basic commands typed from the DOS prompt.

All of these are loaded when your system boots up. The ROM BIOS tells how to load the boot record from the device, which in turn loads the COMMAND.COM information into memory along with all of the device drivers.

The video graphics buffer is memory for what is called a **memory-mapped** video display. Locations in memory correspond to pixels on the graphics screen or to characters on the text screen. The video memory area is special high-speed VRAM (video RAM). Text mode VRAM is at address B8000 and graphics mode VRAM is at address A0000. Many applications write directly to the VRAM to output data. Other programs invoke DOS routines that output data to the appropriate VRAM for you.

Intel IA-32 Processor Architecture

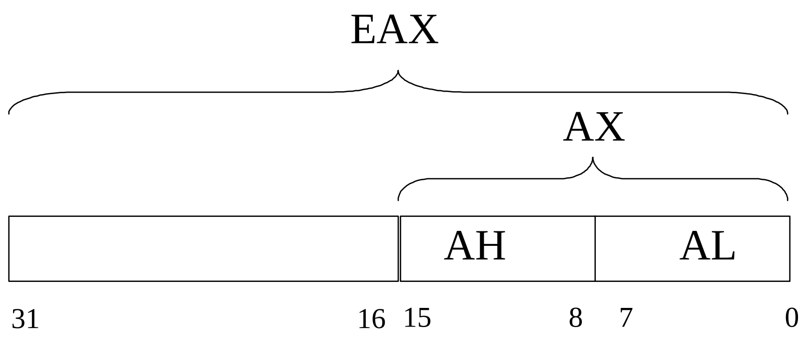
The 32-bit family of Intel processors includes the Pentiums and equivalents that are most common in PC's today. These processors operate in one of four modes:

- Protected Mode. This is the native state of the processor in which all instructions and features are available. Programs run in their own segment, not to be confused with the definition of segment defined earlier with the 8086.
- Virtual 8086 mode. This is an emulation of the 8086 while the IA-32 processor is in native mode. When you create a DOS window in Windows, it is using virtual 8086 mode.
- Real-address mode. This is the programming mode of the original 8086 processor with a few extra's (e.g., a way to switch to protected mode).
- System Management mode. This mode is used by computer manufacturers to customize the processor.

Registers and Addressing

The 32 bit processors avoid the ugly addressing used in the 8086 by supporting a 32 bit address bus and a 32 bit data bus. With 32 bits, this means each processor can access up to 4 Gb of memory. For compatibility with 8086 programs, the 32 bit registers were designed to overlap with the 16 bit registers of the 8086. The new registers are prefaced with an "E" to indicate that they are the "Extended" registers.

For example, just as we had the AX, BX, CX, and DX registers, we now have the EAX, EBX, ECX, and EDX registers. The lower 16 bits of the EAX register corresponds to the AX register, and similarly for the other registers. This means that AH corresponds to the second low-order byte of EAX, and that AL corresponds to the low-order byte of EAX.



We also have 32 bit versions of FLAGS, IP, SI, DI, BP, and SP that are now referenced by EFLAGS, EIP, ESI, EDI, EBP, and ESP.

The CS, ES, SS, and DS registers are still 16 bits. They will be used to index into a descriptor table as we will see shortly..

Real Address Mode

This mode is the programming environment of the original Intel 8086 processor. Newer processors boot into real mode and then have instructions allowing them to switching into protected mode. We have already described how the 1Mb of accessible memory is organized in real mode and how addresses are calculated in the section under 8086.

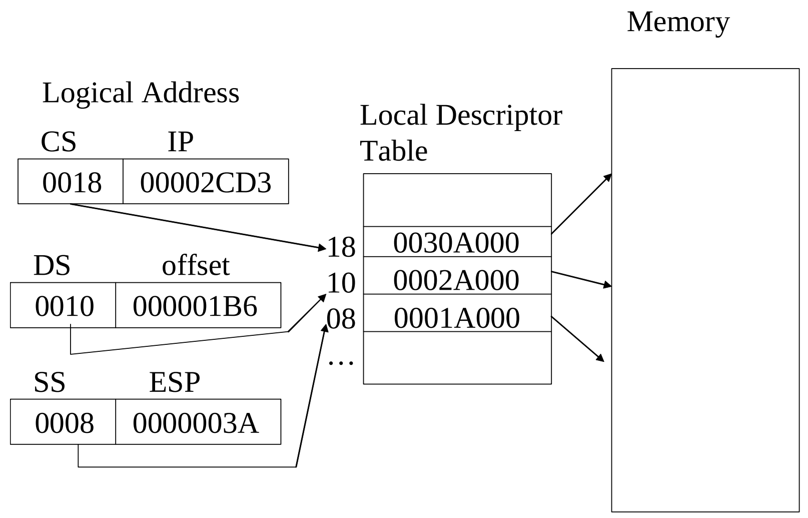
Protected Mode

In protected mode, each *process* (a running program) can access all memory, which we saw is up to 4Gb due to the 32 bit address space. Protected memory access uses a *flat memory model* because we no longer need to use the segment offset scheme; a single 32 bit register can hold the address of any instruction or variable.

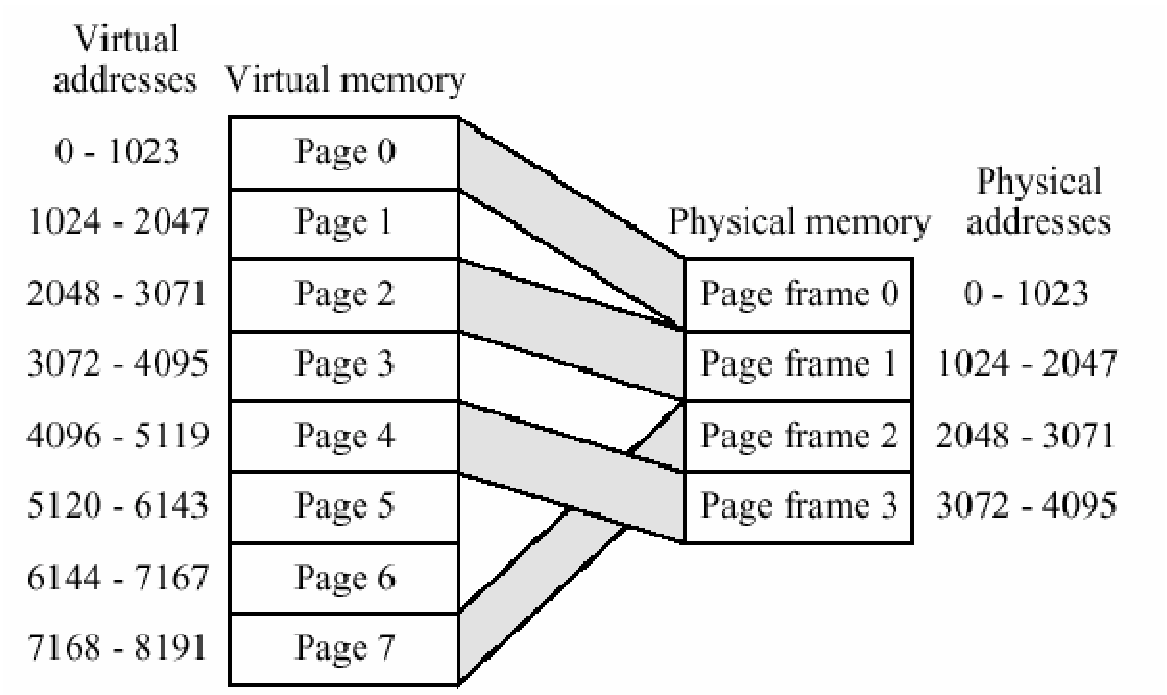
Programs are written in segments. Generally a program will declare a code segment (place in memory where your code goes), a data segment (where your data will be stored such as variables) and a stack segment (where data can be pushed and popped on the stack).

To keep one program from trampling on the memory used by another program, the operating system does a lot of work to ensure that memory remains separate even though from the viewpoint of each program, it has access to the entire range of memory. The first bit of work is that each process is given its own table of segment descriptors, called a *local descriptor table* (LDT).

The segment registers (e.g., CS, SS, DS) reference locations in the LDT which in turn contains the base address in physical memory for that segment. This is similar to the idea of indirect addressing that we saw earlier. The figure below shows the LDT for a particular process. Each segment indexes into the LDT which stores the actual address in memory. The operating system manages the tables to ensure that memory remains consistent. Each LDT entry may also contain bits that indicate access privileges for memory locations.



IA-32 also supports a feature called paging. In paging, a segment is divided into 4K blocks where each block is called a *page*. We can allow there to be more pages than will actually fit in memory, as shown in the figure below. This is called *virtual memory* since the memory doesn't actually exist in RAM:



In this case, we are using 1K blocks for each page (IA-32 uses 4K blocks). Here we are allowing a total of 8K of memory in 8 pages, but physical memory only stores enough for 4K. In this case, physical memory is actually storing virtual pages number 0, 2, 4, and 7. What if we tried to access something in virtual page 1? What the OS will do is have the memory that corresponds to page 1 stored somewhere on disk. The OS will copy that page from disk into a physical frame in memory, for example, it might throw out page frame 0 and copy virtual page 1 into the physical memory that was used for frame 0.

This technique allows us to access more memory than we actually have in RAM. Paging also allows us to start executing a program before it has entirely been loaded; all we need to do is make sure that the first page containing the code to begin executing has been loaded.

You will study paging in much more detail in the Operating Systems class.