

## The Intel x86 Family

- Key features are summarized in the table below

CPU	REG SIZE	Data BUS	MEM (Addr Bus)
8088	16	8	1MB (20 bits)
8086	16	16	1MB
80188	16	8	1MB
80186	16	16	1MB
80286	16	16	16MB (24 bits)
80386SX	32	16	4G (32 bits)
80386DX	32	32	4G
80486SX	32	32	4G
80486	32	32	4G

## The Intel x86 Family

With the 80486 Intel makes the stunning discovery that numbers cannot be trademarked.

CPU	REG SIZE	Data BUS	MEM (Addr Bus)
Pentium	32	64	4G (32 bits)

Pent.Pro, Pentium II, Celeron, Xeon, Pentium III, Pentium 4, Pentium M ... the same

x86-64 64 64  $2^{52}$  phys.  $2^{64}$  virt

Note:

In 2001 Intel Itanium abandoned the core x86 architecture while AMD continued with x86-64 (now called AMD64)

2004: Intel clones AMD architecture and releases 64-bit versions of Xeon, Pentium 4 and Pentium D

## Major Advances

- The above list can be shortened if we are considering major changes only:
- 8086 16-bit regs, "real mode", 1MB physical memory
- 80286 16-bit regs, 16 bit "protected mode", 16MB physical mem
- 80386 32-bit regs, 32 bit "protected mode", 4G physical mem
- x86-64 64-bit regs, 64-bit virtual memory space, segmentation abandoned in "long" mode.

## Incremental Changes

- Processors since the 80386 have been incremental improvements:
- 80486 added integrated caching, x87 coprocessor on same chip
- Pentium parallel execution of some instructions (two integer processors), separate code and data caching
- Pentium Pro optimized for 32 bit execution, improvements in caching
- Pent. MMX (Multimedia) Single-Issue Multiple Data (SIMD) instruction set, MMX registers aliased to x87 registers (can't use MMX and floating point instructions in same block of code).
- Pent.II Parallel bus access, data-flow analysis
- Pent.III SSE (Streaming SIMD Extensions) instruction set with 8 128-bit registers xmm0-xmm7
- Pentium 4 SSE 2/3 Extensions, execution caches, deeper pipelines

## Numeric Coprocessors – the X87s

- The 80x86 processors can operate directly on integers only.  
Floating point computation using an integer architecture is several orders of magnitude (1000x) slower than direct floating point computation in hardware.
- The 8086 and 8088 were designed to work with a floating point coprocessor called an 8087  
The 8087 has operations such as multiplying and dividing floating point numbers, computing sines, cosines, logarithms, and similar functions
- The floating point companions for the 286 and 386 are the 80287 and 80387
- Among the improvements of the 486 was a built-in floating point coprocessor
- The x87 architecture is virtually unchanged since the 8087.  
About a half-dozen new instructions in the Pentium family

## SIMD Instruction Set Extensions

- SIMD: Single-Issue Multiple Data means certain operations can operate on multiple values simultaneously
- MMX - Integer SIMD  
8 64-bit registers (mapped to x87 regs)  
useful for 2D graphics and blending images
- allows 8-8bit operations or 4 16-bit operations
- cannot coexist with x87 code.

### SSE (Streaming SIMD Extensions)

- added 8 128 bit regs xmm0 - xmm7
- can do integer arithmetic or 32-bit single precision FP
- up to 4 32-bit operands per register
- can coexist with x87 code
- FP operations are restricted to simple arithmetic
- added with Celeron/Pentium III

### SSE2/SSE3

- SSE 2 uses the same 128-bit regs as SSE
  - allows 2 64-bit double-precision FP ops/reg
  - maps MMX ops to xmm0 - xmm7 regs
  - added with Pentium 4
- SSE3 is an instruction set extension to SSE2
  - supports arithmetic with complex numbers, integer conversions, specialized graphic operations

### Technology and Economics

- Adrian King (former Microsoft VP) called the Intel segmented architecture "the triumph of marketing over engineering."
- Technical decisions based on the economics of the computer marketplace have left us stuck with many things that would have been better consigned to oblivion
- Backwards compatibility is both a blessing and a curse. It makes life easy because we can still do the old things and use the old stuff but it makes life miserable because we are still burdened by all of the bad design decisions that people made earlier.
- Intel processors and Microsoft operating systems are prime examples.

### Register Size

- The 8086 is said to be a 16-bit CPU because its registers can hold 16 bits. The 80286 is also a 16-bit processor.
  - The 8086 can add, subtract, divide and multiply 16-bit quantities with a single instruction
  - To handle larger numbers multiple instructions are required.
- 80386 and later chips are 32-bit processors.
- 64-bit processors have been on the market since late 2001
  - Intel - Itanium
  - AMD's x86-64 processor was released early 2003

### 16-bit Code

- All x86 family processors can run 16-bit code developed in 1979.
- 16-bit code runs in "real mode."
- Full capabilities of more advanced processors can only be used when running in "protected mode."
- But real mode gives full and remarkably simple access to all hardware
- Later we will see the rationale for these terms.

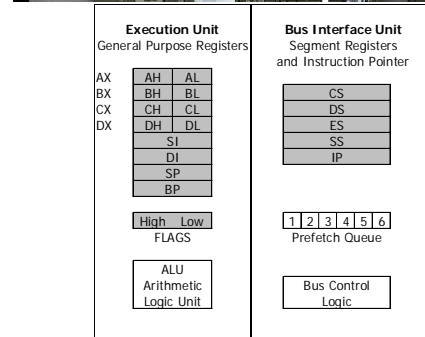
### Systems and Application Programming

- "Systems programming" refers to Operating System programming; applications to apps running under an OS
- What are the differences?
  - In applications programming, we rely on the services of an operating system to perform basic tasks such as reading the keyboard, putting characters on the screen, opening and closing files.
  - Direct control of the hardware is rare
- In systems programming, we may be writing programs that control hardware, control other programs, manage memory, and implement other functions of an operating system.

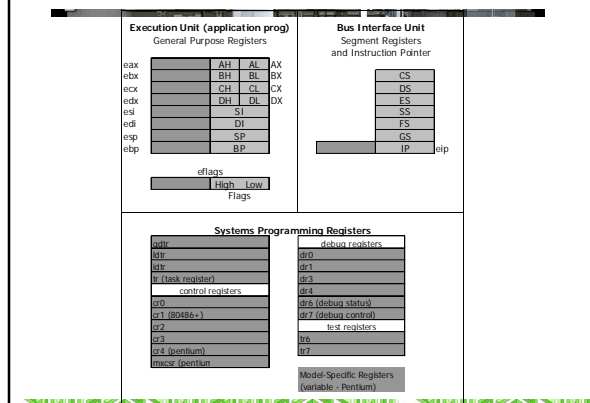
## Processor Support for OS Programs

- The 8086 made no distinction between OS and applications. Full control of the machine is open to applications
- Starting with the 80286, the processor defined 4 levels of privilege
- Only programs running on the highest level (PL=0, often called "ring 0") have access to hardware and I/O devices.
- Applications are the least privileged (PL=3)

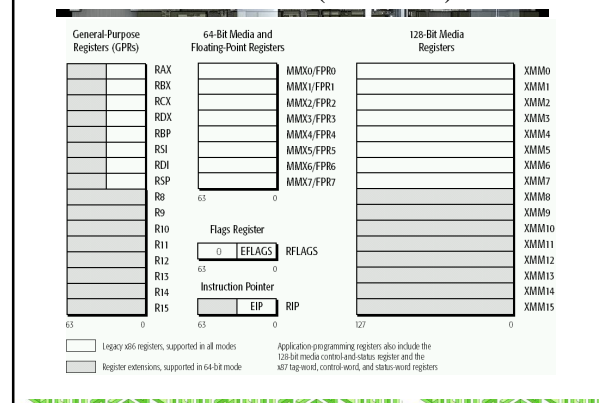
## The Intel 8086 Processor



## Intel 80386+ Processors



## AMD x86-64 (Athlon 64)



## 8086

- The figure is incomplete since it does not show the interconnections among the units and many other details
- It is the programmer's view of the processor
- For our purposes, it is enough to describe what the different pieces of the processor are without getting into details of how the pieces are wired together
- The 8086 is a much simpler processor than its descendants
  - All registers are "program visible"
  - All registers are equally accessible to applications and to the operating system
  - There are no "privileged instructions"; most CPUs today provide at least two levels of privilege
  - Privileged instructions may only be executed by a privileged process. This is the hardware foundation for safe operating systems
- Starting with the 80286, Intel processors were built with privileged instructions that are normally only available to the OS and with registers that can only be accessed with these instructions

## EU and BIU

- The first distinction that is worth noting is between the Execution Unit, sometimes referred to as the EU, and the Bus Interface Unit, sometimes referred to as the BIU
- The Execution Unit (EU) carries out the instructions that the processor receives and is generally the unit that is directly controlled by the programmer
- The Bus Interface Unit (BIU) handles address computations and movement of instructions and data between the EU and the outside world which includes memory and peripheral devices
- The programmer generally lacks direct control over this unit

## The Machine Cycle

1. Fetch
  2. Decode
  3. Execute
  4. Repeat forever...
- The EU and BIU were designed to operate in parallel.
  - As the EU was executing one instruction, the BIU would fetch the next from memory
  - Predecessor of today's pipelining technology that allows execution of two or more instructions simultaneously

## The Execution Unit

- The EU is shown with 3 main components
  1. Registers `eax`, `ebx`, `ecx`, `edx`, `ebp`, `edi`, `esi`, `esp` (AX, BX, CX, DX, BP, DI, SI, SP)
  2. The EFLAGS (FLAGS) register
  3. The ALU
- We will not discuss the details of the ALU in this track of the course; it will be covered in the architecture track
- We will focus on how the registers and memory are affected by different instructions – the programmer's view

## General Purpose Registers

- The registers `eax`, `ebx`, `ecx`, and `edx` (AX, BX, CX, and DX) are often referred to as DATA REGISTERS. The 32 bit registers have four separately addressable parts and the 16-bit registers have 3:
  - 32-bit `eax`, 16-bit AX, 8-bit AH and AL
  - 32-bit `ebx`, 16-bit BX, 8-bit BH and BL
  - 32-bit `ecx`, 16-bit CX, 8-bit CH and CL
  - 32-bit `edx`, 16-bit DX, 8-bit DH and DL
- Any of these registers can be used for arithmetic and logical operations on data
  - Many have specialized purposes
- But there is no name for the upper 16-bits of EAX and it cannot be accessed as a 16-bit register
- x86-64 64-bit registers follow the same scheme:
  - `rAX`, `EAX`, `AX`, `AH`, `AL`
  - Where `rAX` is a 64-bit register, etc.
- AMD 64 bit machines also have `r8` ... `r15` (pure 64 bit registers)

## Index Registers

- The registers `ebp`, `esp`, `edi`, `esi` (BP, SP, DI, SI) can only be used as 32-bit (16-bit) registers and are often referred to as POINTER and INDEX REGISTERS
  - These registers are used to reference addresses in memory
    - BP and SP by convention are used to address the stack
    - DI and SI are used in string processing or block memory operations
- Note that 32-bit machines allow use of both 16 and 32 bit registers e.g. `ESI` and `SI` but while `esi` can be used to address memory in a 32-bit machine `SI` cannot.
- 16-bit processors have significant limitations on the use of index registers to address memory
  - These limitations disappeared in the 32-bit architecture
- We will soon discuss the special functions of individual registers on the 80x86 but they are interchangeable for many of the instructions

## 16 and 32 bit Register Usage

- The 16 and 32 bit machines have parallel register sets
- Most instructions can operate using 32 bit registers just as they operate using 16 bit registers
- The one exception concerns the use of index registers to address memory
- The other significant difference between the 16 and 32 instruction sets concerns the amount of memory addressable by a conditional jump
- More later

## The eflags / FLAGS Register

- The eflags/Flags register contains two groups of bits:
  1. Bits that specify the status of the machine (Status Flags)
  2. Bits that affect the operation of the processor (Control Flags)
- The Flags register is always treated as a collection of bits and is never used as a unit on its own
- Status Flags are used to indicate various conditions that might occur when an arithmetic operation or comparison is performed
- The Flags register is like a global variable in a program. It is shared by all instructions and processes

## Instruction Semantics

- When we examine instruction semantics, we have to ask two questions:
  - How does the instruction affect its operands?
  - How does the instruction affect the flags?
- It is important to remember that flags are a "shared global resource."

## The Data Registers

- Although the names probably came from A,B,C and D they also have English names that are reminiscent of their functions:

16-bit	32-bit	Name
AX	eax	Accumulator
BX	ebx	Base Register
CX	ecx	Count Register
DX	edx	Data Register

## The Accumulator

- Eax, AX, and AL are all referred to as "the accumulator"  
Sometimes AH and AL are called accumulator high and accumulator low
- Eax, AL, or AX are used primarily for arithmetic and logical operations on data. It corresponds to the "accumulator" found in older architectures  
Many instructions are optimized to be shorter or faster with the accumulator as an operator  
These instructions operate on eax, AX, AL but not AH
- Sometimes eax and edx (AX and DX) together form one large register.
- When multiplying two 32 bit numbers the product is 64 bits and is stored in edx:eax (edx is the high part, eax is low part)
  - for 16-bit multiplication the result is stored in DX:AX
  - for 8-bit multiplication the result is stored in AX alone

## Special Uses of the Accumulator

- The accumulator **MUST** be used in arithmetic operations such as full-precision multiplication and any division  
The 386 and later processors have multiplication instructions that are more general but discard the upper bits of the product
- Other specialized instructions such as sign-extension and table-lookup use the accumulator
- Input and Output require the use of the accumulator. It is the only register that can store receive data from an I/O device or send data to an I/O device

## Colon Notation

- Register "pairs" are frequently written with a colon:  
eax:edx    CX:BX    es:edi    ES:DI    eax:edx
- In some cases, a register pair refers to a 64-bit (32-bit) quantity and in other cases, a register pair refers to a 32-bit (20 bit) quantity  
DX:AX   CX:BX   32 bits    edx:eax    ecx:ebx   64 bits  
CS:IP   ES:DI   20 bits    cs:eip    es:edi   32 bits
- The rule is that if addresses are involved, it is a 32-bit (20-bit) quantity.
- If two general purpose registers are involved it is a 64-bit (32-bit) quantity.
- This will make more sense after we discuss effective address formation and segment registers later

## The Base Register (ebx / BX)

- ebx and BX are known as the base register
- BH and BL are base high and base low registers
- BX was called the base register because it is used in 16-bit code as an index register to store the base address for array accesses  
The use of bx as an index register is a 16-bit peculiarity.  
AX, CX and DX cannot be used as index registers while eax, ecx and edx can be so used
- Note that 8 bit registers are **NEVER** used to store addresses and 16-bit registers are not used to store addresses in a 32-bit program
- BX also has some other specialized uses such as the table-lookup instruction XLAT

### The Count Register

- ecx and CX are known as the count register
- Many iterative instructions work with a count in CX or ecx:
  - Loops: loopd, loop, loope, loopne, etc
  - Block memory ops: movs, cmps, stos, etc
  - CL is used as counter by some rotation or shift instructions
  - Block I/O instructions: ins, outs

### The Data Register

- edx (DX) is probably called the data register because they couldn't think of a better word that starts with D
- DX is used most often for the same purposes as AX – arithmetic and logical operations – because CX and BX have specialized purposes
- DX does have some specialized purposes, but fewer than other registers:
  - Multiplication and Division use both eax and edx (AX and DX) and cannot use other registers
  - DX is the only register that can hold an address in I/O space for the IN and OUT instructions. Even in 32-bit processors the 16-bit DX is used for this purpose
  - Some sign-extension instructions use DX or edx

### General Purpose??

- We have seen that all of the so-called “general purpose” registers actually have specialized uses (some are more general-purpose than others!)
- This is characteristic of CISC (complex instruction set) computers
- Contrast with the Motorola 68000 series (also CISC, but less so than Intel)
  - 8 data registers named D0 – D7
  - 9 address registers A0 – A7 and A7'
  - Any operation that can be done on a data register can be done in any data register
  - Any operation that can be done with an address can be done with any address register (except A7 and A7')

### The Pointer and Index Registers

- This group of four registers is used almost exclusively for memory access
- edi, esi, ebp, esp (DI, SI, BP, SP)
  - Note that 16-bit code uses the 16-bit index registers while 32-bit code uses the 32-bit index registers
- esp/SP can ONLY be used with the stack
- ebp/BP is intended for use the stack, but can (with extra instructions) be used for general memory access
- edi/DI and esi/SI are general purpose index registers but are also associated with string or block memory instructions

### The “Index” Registers edi (DI) and esi (SI)

- DI is “Destination Index” and SI is “Source Index”
- The names derive from their association with the string or block memory instructions MOVs, CMPS, SCAS, LODS and STOS where they are implied operands
- In 16-bit code only BX, SI and DI may be used as general purpose index registers (pointers to memory)
- In 32-bit code eax, ebx, ecx, edx, esi and edi may all be used
- Note that esp (sp) and ebp (bp) are also pointer registers but are associated with the stack.

### The “Pointer” Registers ebp (BP) and esp (SP)

- BP stands for Base Pointer and is used for addressing data on the stack (more on this topic later)
- It might be better named the ‘Frame Pointer’ because its primary usage is to point to structures called stack frames (aka activation records)
- SP stands for Stack Pointer and always points to the top of the stack
  - We will explain the stack and its significance later in much greater detail
  - Hardware stacks were developed to provide hardware support for programming languages

- The following figure illustrates the structure of the flag register (bits are numbered using hex)

- A useful mnemonic for flags is to mentally pronounce "ODITSZAPC"
- OZAPC bits are STATUS FLAGS and are set by the processor as instructions are executed
- DIT bits are CONTROL flags and are set by the programmer to control the way that certain instructions execute or the way that the processor itself works

- The 32-bit equivalent of the FLAGS register is called the EFLAGS register

- Changes to the Flags since the 8086 are relevant only to Systems programming.
- Use of the Flags by applications is unchanged since 1979.

The PARITY FLAG (PF) (bit 2 of F) is set to 1 if the result of an operation has even parity, to 0 if it has odd parity. Obviously this flag is useful for communications applications

The CARRY FLAG (CF) (bit 0 of F) is set to 1, if there has been a carry or borrow out of the MSB following an arithmetic operation. Otherwise it is set to 0.

CF also is affected by many other instructions; for example following a shift or rotate it has a copy of the last bit affected

CF is also used extensively to pass Boolean parameters

- Control Flags affect the operation of the processor itself
- D= Direction Flag; T = Trap Flag; I= Interrupt Enable Flag
- If the DIRECTION FLAG (DF) (bit A of F) is set to 0 all string/block memory instructions operate from low to high memory
- Example: REP MOVSD copies 4 bytes of memory whose location is given by [ds:esi] to memory whose location is given by [es:edi]
- Then both edi and esi are incremented by four.
- The instructions continues for the number of operations specified in ecx
- If DF is set to 1, then edi and esi are decremented rather than incremented

- If the TRAP FLAG (TF) (bit 8 of F) is set to 1, the CPU "single-steps," pausing after each instruction. If is set to 0, the CPU works as usual.
- Clearly the processor doesn't actually pause.
- What happens is that control is turned over to a debugger whose address is found in Interrupt Vector 3 (more on this shortly)
- TF is used only by debuggers.



- IF enables and disables interrupts
  - IF  $= 0$  interrupts are disabled and the machine will not respond to any external hardware event
  - IF  $= 1$  interrupts are enabled and machine operation proceeds normally
- Examples of hardware that generate interrupts are the keyboard, the clock, disk drivers, printers etc.
- Interrupts may be disabled so that the operating system can execute critical regions of code that cannot be interrupted without disrupting the operating system
- When an interrupt occurs IF is set to 0 automatically
- Normally an interrupt handler will enable interrupts as the first instruction; if critical handling is necessary then interrupts will be enabled as soon as the critical section has executed

- The following assembler code

```
cli
    jmp short $
```
- Can be assembled to a .com file of three bytes (e.g., halt.com)

```
FA EB FE
```
- When executed this code will immediately halt any machine running Windows 98, Windows ME, any older version of Windows or MS-DOS
- The CLI instruction disables interrupts
- The next instruction jumps to itself (a very tight loop indeed)
- The machine is dead; it can only be powered off

- Two of the bits added to the eflags register in the 80486 are:
  - VM Machine is in virtual-8086 mode
  - VIF Virtual Interrupt Flag
- Because IF is a critical system flag, it should not be accessible to anything other than the operating system
- The purpose of VIF is to simulate the operation of IF in virtual-8086 mode
- When Microsoft created Windows 95 as a 32-bit protected-mode operating system they found that protecting IF with VIF resulting is about a 15% decrease in speed for 16-bit code
- They felt that this would be unacceptable particularly for games (notable for direct control of hardware) and might cause consumer rejection of the OS
- By contrast the Windows NT operating system was intended for business and always protected IF
- The NT lineage includes Windows 2000, Windows XP, and Vista

Flag		NT	IOP	O	D	I	T	S	Z	A	P	C				
Bit#	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Flag										ID	VIP	VIF	AC	VM	RF	
Bit#	1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10

- eip/IP cannot be changed directly (i.e., as an operand of an instruction), but changes ONLY as the result of executing instructions such as Jumps, Calls, and Interrupts



## Segment Registers

- The segment registers (segregs) are 16-bit registers only and are used to fully access the memory of the 8086
  - Without using the segregs in the 8086, a program can only access 64 KB of memory
  - With segment registers, a program can access the full 1 MB of memory that the 8086 can address
- The segment registers have the following names:
  - CS Code Segment
  - DS Data Segment
  - SS Stack Segment
  - ES Extra Segment
  - FS, GS (No special names)
- Any address register is normally associated with a specific segment register when the bus interface unit calculates an address
- This also applies to 32-bit processors and 64-bit processors that are not operating in "long" mode
- Note that segment registers are 16 bits even in 32 or 64 bit processors

## The SegRegs

- **CS - Code Segment**
  - This register is used in conjunction with eip/IP to determine the address of the next instruction to be executed
  - No other registers are associated with CS
- **SS - Stack Segment**
  - This register is used in conjunction with esp/SP (Stack Pointer) for stack operations. ebp/BP also is associated with SS.
- **DS - Data Segment**
  - By default, when the general purpose registers eax, ebx, ecx, edx, esi and edi are used as index registers, they refer to this segment (In 16-bit processors this includes only BX, SI and DI)
  - Any direct references to memory (e.g. mov ax, count) is DS - relative
- **ES - Extra Segment**
  - Can be used as needed by the programmer
  - But with string instructions, edi/DI is associated with ES rather than with DS

## SegRegs in 32-bit Processors

- The 80386 added FS and GS registers. These have no assigned special purposes or even names
  - Segment registers remain 16 bits even in 32-bit processors
  - Segment registers in 32-bit processors are often referred to as "Selectors" rather than segment registers
- In 16-bit processors, segment registers contain part of the address used for a memory reference
- In 32-bit processors, segment registers contain pointers into one of two possible tables of 64-bit segment descriptors:
  - Local Descriptor Table (LDT)
  - Global Descriptor Table (GDT)
- Thus, in 16-bit mode, the segment registers have "real" addresses
- More on this when we discuss effective address (EA) computation

## Segment Register Operations

- Only a very few instructions can affect the segment registers directly
- The only operations that can be performed are loads and stores, and even these are restricted.
- It is not possible to perform arithmetic, logical or bit manipulation operations directly on a segment register

## SegRegs and 16-bit Executables

- MS-DOS provides two types of executable programs: EXE and COM files
  - COM files ignore the segment registers and the entire program is contained in 64KB
  - 64KB of memory is a lot of memory for an assembly language program
    - The A86 assembler is only about 24KB and is a very sophisticated piece of code
    - The original Turbo Pascal was only about 35 KB of code and included a compiler, a text editor and a debugger
- All 4 segment registers have the same values when a .COM program is executed
- EXE files (even small ones) have separate data, code and stack segments.
- Segment registers normally have different values when an EXE program is executed.

## 32-Bit Flat Model Programming

- .COM files are restricted to the 16-bit MS-DOS operating system
- .COM files contain a pure "load image" - as opposed to .EXE files, which have some "blank addresses" that are filled in at load time.
- .COM files have an unsegmented (flat) 64-KB address space
- Although this might appear to be an obsolete technology, 32-bit .EXEs are normally constructed in a similar manner
- In the 32-bit flat model, DS, ES and SS are identical and refer to the same area of memory (ALL of it) referred to by CS: a 4GB flat address space
- Segment registers can be ignored and you can program in a manner similar to DOS .COM programming

## 20-Bit Effective Address (EA) Computation

- The standard 8086/8088 chip can address 1 MB of memory. Addresses are 20 bits and range from 00000h to FFFFFh
- Since address registers are 16-bits, how do you address 20-bit memory?
- Answer: combine two 16-bit register so  $16 + 16 = 20$
- The Segment registers mentioned above supply the "first four" digits and the other registers supply the "last four".
- In plain English, to get the complete address take the number in a segment register, multiply by 10H (shift left 4 bits) and add the value in a regular register ("the offset").
- Addresses are often given in the form Segment:Offset.  
For example, 2349:1AB8 gives  

$$\begin{array}{r} 23490 \\ + \quad 1AB8 \\ \hline = 024F48 \end{array}$$

## Segmented Addresses

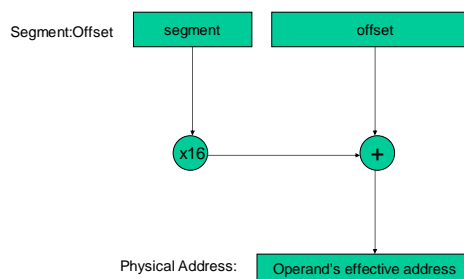
- The main idea here is how to express an N-place number using numbers with fewer than N digits.
- For example, how would you express 3 digit decimal numbers using two decimal digits?  
 $999 = 99:9 = 93:69 = 90:99 = 94:59$  etc.
- In general, a single memory address can have many different equivalent expressions.  
 Consider the address 0800:0100  
 In 5-digit hex this is the address 08100  
 But it can also be expressed as:
 

0000:8100	0100:7100
0400:4100	0700:1100
0810:0000	080F:0010
....	....
- The fact that a single address has many valid representations makes operations such as comparing pointers for equality problematic

## Real and Protected Mode

- Protected Mode addressing (where the CPU can access more than 1MB of memory) is radically different  
 The Segment Registers are used to point into tables of descriptors  
 A descriptor describes a segment's location, length and access rights
- "Real mode" derives its name from the fact the segment registers in this mode contain "real" addresses
- You can do address arithmetic with segment register values (but not in the registers themselves)
- In protected mode, the numerical value of the segment register has NO relation to the actual address
- Address arithmetic is not possible with protected mode segment registers

## Real-Mode Addresses



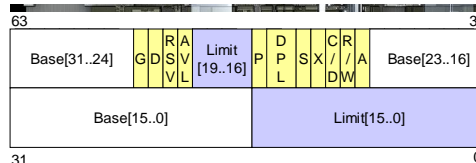
## EA Computation in 32-Bit Protected Mode

- Selector (segment register) layout:



- Index (Bits 3-15) 13-bit values allows 8,192 entries in a descriptor table
- Table Indicator (Bit 2) (1=Local DT, 0 = Global DT)
- Requested Privilege Level (Bits 0-1) must be <= privilege level stored in segment descriptor

## Segment Descriptor Format



- Address computation
  - Selector indexes into GDT or LDT
  - Base Address is fetched from Descriptor
  - Offset is added to base address to determine "linear" 32-bit address
  - Linear address compared against Limit (in 4KB or 4MB pages) to determine if address is legal

[Home](#)
[About Us](#)
[Contact Us](#)
[Privacy Policy](#)
[Terms of Service](#)

[Home](#)
[About Us](#)
[Contact Us](#)
[Privacy Policy](#)
[Terms of Service](#)

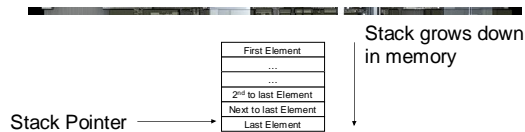
[Home](#)
[About Us](#)
[Contact Us](#)
[Privacy Policy](#)
[Terms of Service](#)

[Home](#)
[About Us](#)
[Contact Us](#)
[Privacy Policy](#)
[Terms of Service](#)

[Home](#)
[About Us](#)
[Contact Us](#)
[Privacy Policy](#)
[Terms of Service](#)

[Home](#)
[About Us](#)
[Contact Us](#)
[Privacy Policy](#)
[Terms of Service](#)

## The Stack

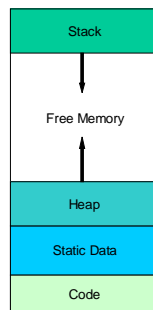


- Operations are Push and Pop
- Push adds an item to the stack (decrement ESP, copy data to top of stack)
- Pop copies an item from the top of stack and increments ESP

## Stack Pointer

- Stack is located in segment pointed to by SS  
ESP/SP (Stack Pointer) points to the top of stack  
SS:SP or SS:ESP
- Be careful about direct manipulation of ESP  
Because return addresses (among other things) are stored on the stack, failure to remove temporary data before executing a POP will pop garbage into eip  
The CPU will attempt to execute whatever eip points to as if it were a program

## Typical Program Memory Organization



## I/O Ports

- Besides memory, the x86 processors can access up to 65,536 I/O ports. This is called "I/O space" (distinguished from Address or Memory Space)  
Memory space is accessed via MOV and other instructions.  
I/O space can only be accessed with IN and OUT instructions
- I/O ports act like memory, except that they are typically connected to hardware devices such as printers, disk drives and serial ports  
Other architectures may use memory-mapped I/O ports, where hardware registers of various devices appear to exist at certain memory addresses
- Some I/O ports are read-only, some are write-only, and others are read/write
- Note that if no device exists at a given I/O address, there is no effect from a read or write to that address
- Text screens (command prompts) use memory-mapped video. You can "output" to the monitor by storing data directly into certain addresses

## I/O Space

- All x86 processors allow for a 64K I/O port space (16-bit addresses)  
Some early machines only implemented 10 bits on the bus (1024 ports)
- A "port" in I/O space is always one byte wide just as memory is always one byte wide
- If you load a 32-bit value from memory you are accessing bytes at 4 contiguous addresses in memory
- If you read a 32-bit quantity from I/O space you are accessing 4 contiguous ports

## Peripheral Devices

- Other than memory-mapped video, communications with peripheral devices takes place in I/O space:
  - keyboard
  - clock
  - disk drives
  - serial and parallel ports
  - mouse
  - video adapters
  - USB hubs
  - ....
- Note the usage of the term "port" in two different contexts:  
An I/O port is an address in I/O space  
A serial or parallel port is a physical device; a simple processor whose registers occupy several addresses in I/O space

## Units of Memory in x86 Systems

- x86/x87 processors use the following "units of memory"

Byte		8 bits
Word	2 bytes	16 bits
Double Word	4 bytes	32 bits
Fword		
(Far pointer)	6 bytes	48 bits
Quad Word	8 bytes	64 bits
Tenbyte	10 bytes	80 bits

- Also occasionally encountered is the term "paragraph" (16 bytes)
- Double word is usually abbreviated to dword and quad words to qword
- Terms are still used on 32 and 64 bit machines even the word size is 4 or 8 bytes on these machines

## Data Storage Conventions

- Processors are classified as:
  - Big-Endian (Motorola, Sparc, etc)
  - Little-Endian (Intel x86, Z80, Vax etc.)
  - Bi-Endian (Power PC, Sparc V9, MIPS)
- Endian storage conventions apply to numbers or other objects that are larger than one byte
- For example 12345678h can be stored as 12 34 56 78.
- BIG-ENDIAN processors expect data to be stored with the most significant byte in the LOWEST address  
This is a natural order for Westerners who read left to right
- Intel chips are LITTLE-ENDIAN. Intel chips store numbers backWORDS, so the above number would be stored as 78 56 34 12  
The MSB is stored in the highest address
- Note that the hex digits (nibbles) within a byte come in the normal order!

## Endianness

- "Big-endian" was first used in Gulliver's Travels (1726).
- A royal decree in Lilliput required one to open a soft-boiled egg at the small end, while the "big-endian" subjects of Blefescu had to crack them at the big end
- Consider the value 7Fh as a byte, a 16-bit word and a 32-bit double word

	Byte	Word	DWord
BIG	7F	00 F7	00 00 00 7F
LITTLE	7F	7F 00	7F 00 00 00

- Now consider 017Fh as an array of two bytes, as a 16-bit word and a 32-bit dword

	Byte	Word	DWord
BIG	01 7F	01 7F	00 00 01 7F
LITTLE	01 7F	7F 01	7F 01 00 00

## Objects in Memory

- Endianness only affects "objects" that are known to the processor (e.g., 32-bit integers)
- Other objects such as strings or arrays are fictions imposed by humans on memory
- Strings of single-byte characters are stored left-to-right (low-to-high) on both little and big endian machines

## Memory Dumps

- You can examine a memory dump of a big-endian machine and read naturally from left to right

- It's not so easy with little-endian memory

```
msg db "ASM is fun!",0dh,0ah
db "(for masochists)"
byt db 12h, 34h, 56h, 78h
wrd dw 1234h, 5678h
dwd dd 12345678h
qwd dq 12345678h
one dd 1.0
```

- Displayed by debug:

```
0EB4:0100 41 53 4D 20 69 73 20 66 ASM is f
0EB4:0108 75 6E 21 0D 0A 28 66 6F un!..(fo
0EB4:0110 72 20 6D 61 73 6F 63 68 r masoch
0EB4:0118 69 73 74 73 29 12 34 56 ists).4V
0EB4:0120 78 34 12 78 56 78 56 34 x4.xVxV4
0EB4:0128 12 78 56 34 12 00 00 00 .xV4....
0EB4:0130 00 00 00 80 3F 72 6F 6D ....?rom
```

## Endianness and Networks

- Note that endianness makes it possible to write non-portable programs in high-level languages such as C if they address memory directly
- This should not be a problem if you do not engage in "tricky" programming that relies on intimate knowledge of the machine.
- A program that accesses a single byte of a long integer is violating the abstraction
- Networks generally use big-endian order.
- Files containing binary data have to be manipulated to have the correct endianness when transmitted over a network or exchanged between machines

## Interrupts

- Computers have to handle hardware and external events as well as perform computations. Sometimes events require immediate action.  
Example: a key has been pressed
- Hardware generates a special signal called an interrupt request to signal an event that requires handling
- CPU saves state of machine (pushes registers on stack), suspends current process and transfer control to a process that handles the event
- The process that handles the interrupt is called an Interrupt Service Routine or Interrupt Handler
- When an ISR is finished control returns to interrupted process. Previous state of registers must be restored

## Types of Interrupts

- **External**  
Generated by input, output storage and other devices
  - Examples: clock, keyboard, mouse, hard disk
- **Internal**  
Generated by processor. Subclassified as Traps, Exceptions, and Faults.
  - Exceptions or aborts: serious errors, cannot restart instruction ex. Integer divide by 0
  - Faults: can restart faulting instruction. Ex. Page fault (referenced page is not in memory)
  - Traps: debugger breakpoints
- **Software**  
Generated by software during the course of program execution
  - This type of interrupt is actually an instruction; more like a procedure call than the interrupts described above

## 80x86 Interrupts

- 80x86 processors recognize 256 different interrupts
  - Generally 16 or 32 are assigned to hardware
  - The rest are available for use as software interrupts
  - Invoked in Intel processors via an INT instruction
- Note the difference between an INT and IRQ (Interrupt Request).
- An interrupt-driven device is assigned an IRQ line to use.
- It raises a signal on the IRQ line which is directed to another device called a Programmable Interrupt Controller (PIC)
- The PIC signals the processor on the INTR line and also signals the processor which INT handler to invoke by placing the interrupt number on the bus.  
This is normally NOT the same as the IRQ  
Example: Keyboard IRQ 1 is handled by INT 9

## Interrupt Vectors: 32-bit machines

- 32 and 64 bit machines also recognize 256 different interrupts
- The IDTR (Interrupt Descriptor Table Register) contains the address of IDT
- The IDT contains 8-byte interrupt descriptors
- Interrupts are vectored through IDT entries

## Interrupt Vectors: 16-bit machine

- The first 1K (400H) bytes of physical memory are reserved for interrupt vectors--addresses of the ISRs that handle each interrupt
- MS-DOS OS services are invoked through software interrupts
- Intel machines have part of the 16-bit OS stored in ROM  
BIOS -- Basic Input/Output System - contains simple hardware routines and power-on code

## Real-Mode Memory Map

00000-003FF	Interrupt Vector Table
00400-004FF	BIOS Data Area
00500-006FF	DOS Data Area
00700-0xxxx	MS-DOS Device Drivers resident command.com or cmd.exe
00xxx-A0000	Application programs Transient command.com or cmd.exe
A0000-BFFFF	Display adapter RAM memory (text mode and low-res graphics)
C0000-DFFFF	Peripheral ROM (e.g., disk controller)
E0000-EFFFF	BIOS Expansion ROM
F0000-FFFFF	BIOS ROM

- Note: FFFF0 is hardwired in the processor as the startup address
- Entry point to ROM BIOS Power-on Self Test (POST) and boot code

## Calling MS-DOS Services

- All DOS and BIOS Services have an INT interface
- In real mode, applications can call OS, BIOS, or even control hardware directly.
- All 32-bit operating systems (Windows and Unix) provide a procedure or function-call interface.
  - Applications CANNOT issue an INT instruction (causes GPF)
  - Applications CANNOT issue I/O instructions
  - Device drivers however may use INT to communicate with OS.
- 

## Int 21h

- Most MS-DOS functions are available via the DOS Function interrupt (Think of it as a "master interrupt") INT 21H
- These functions are all activated as follows:
  1. Put the number of the function you want number into AH.
  2. Put whatever other parameters are required into the appropriate registers.
  3. Execute the instruction INT 21H.
- For many functions (especially file I/O the carry flag (CF) is used a Boolean success flag.
  - If the flag is cleared, no error occurred.
  - If the flag is set, an error occurred and the corresponding error code is in AX.

## Terminating a Program

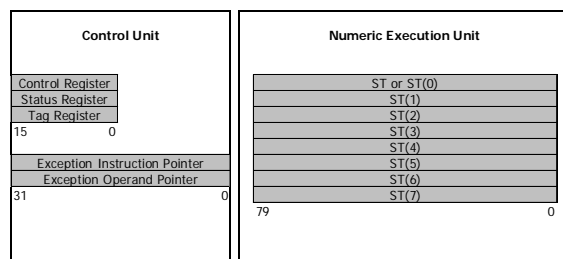
- When a program ends, what does the CPU do next?
- Control is returned to the Operating System
- To terminate a program and return control to DOS, use DOS function 4Ch.
  - load AH with 4C
  - if you wish to return an errorlevel to DOS, put that value in AL; otherwise set AL to 0
  - issue an INT 21h instruction
- Example:
 

```
MOV AX, 4C00h
INT 21h
```

## 80x87 Architecture Overview

- 80x87 Coprocessors are designed for floating point operations. Processor consists of two units:
  - Control Unit (CU): responsible for interfacing with 80x86 processor
    - 3 16-bit control and status registers
    - 2 32-bit exception pointers
  - Numeric Execution Unit (NEU)
    - responsible for numeric execution
    - 8 80-bit data registers for computation

## The 80x87 FPU



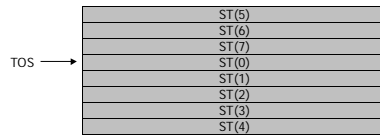
## 80x87

- The 80x87 is a stack machine
  - 8 Data Registers form a stack
  - Addressed as ST(0), ST(1), ..., ST(7)
  - ST can be used to refer to ST(0)
- Top of stack is maintained by a 3-bit field in the status register
- However, it is easiest to think about this machine by pretending that pushes and pops cause data in other registers to "move"
  - After loading a value into ST(0), the data that was formerly in ST(0) is now in ST(1); ST(1) in ST(2) etc.



## TOS Pointer

- A three bit field in the status register



## Data Registers

- Data registers are all 80 bits wide
- All variables loaded into an 80x87 register are automatically converted to IEEE 80-bit temporary (extended) real format
- This has significant implications for high-level languages
- Computations (for example, with 32-bit singles) may have different results depending on whether results have been stored to memory

## Tag Register

8087 Tag Register							
tag 7	tag 6	tag 5	tag 4	tag 3	tag 2	tag 1	tag 0
15	14	13	12	11	10	9	8
7	6	5	4	3	2	1	0

- Tag field describes state of each register
  - 00 valid (finite nonzero number)
  - 01 zero
  - 10 invalid (infinity or NaN)
  - 11 empty

## Status Register

8087 Status Register															
B	C <sub>2</sub>		ST			C <sub>2</sub>		C <sub>0</sub>	IR	PE	UE	OE	ZE	DE	IE
15	14	13	11	10	9	8	7	6	5	4	3	2	1	0	0

- B** Busy bit. Indicates if 8087 currently busy (see WAIT instruction)
- C<sub>0</sub>-C<sub>3</sub>** Condition codes. Set by comparison instructions (and some others)
- ST** 3-bit top of stack pointer
- IR** Interrupt Request (8087 only)
- On 287 and above bit 7 is ES (Error Summary) = 1 if any of bits 0 - 5 set
- Remaining Flags are Exception Flags (1 = Exception Occurred)

## Control Register

8087 Control Register															
IC				RC				PC				PM			
12	11	10	9	8	7	6	5	4	3	2	1	0	0	0	0

- Bits 0-5 are exception mask bits
  - If Mask bit clear when exception occurs interrupt is generated for 8086
  - If mask bit is set exception value is produced as described above
- Control fields
  - PC** (Precision Control, default extended)
    - 00 single precision
    - 01 (not used)
    - 10 double precision
    - 11 extended precision
  - RC** (Rounding Control, default to Nearest)
    - 00 to nearest
    - 01 toward negative infinity
    - 10 toward positive infinity
    - 11 toward 0
  - IC** (infinity control, default projective mode)
    - 0 projective mode (infinity unsigned)
    - 1 affine mode (infinity is signed)