# The Intel Pentium: Design Considerations

The name “Pentium” represents a line of central processing units developed by Intel, beginning in 1992 and continuing to this day.

Most of the existing Pentium chips fall into the IA-32 family and, thus can be seen as extensions of the Intel 80386.

Unlike the earlier IA-32 models, the Pentium was designed to support MS-Windows, which was beginning to evolve into a true operating system.

A number of Pentium design features are best seen in the light of the requirements placed by a modern operating system.

We begin this lecture by discussing a few topics from the study of Operating Systems.

# Processes in an Operating System

The idea of a process is one of the most fundamental in the study of operating systems.

The term “process” is necessarily defined somewhat vaguely.

The term “**process**” refers to the executable image of a program, along with the assets and resources required to support that execution.

The **general purpose register** set can be considered one of the assets that are a part of the process. Each process must have unique access to the registers when it executes.

The term “**resource**” may refer either to an asset such as a file, or a data structure used to control access to that asset.

Note that, in general, memory is not a resource belonging to any one process; it is shared. One big job of the Operating System is to manage this sharing.

The idea of a process arose in the era of single-CPU computers.
It can be generalized to multi-core computers.

# Resource Sharing/ Time Sharing

The idea of time sharing arose in the early days of computing in order to allow the sharing of expensive resources, such as the CPU, by many programs.

The idea of time sharing is based on the relative speeds of (even older) computers and humans. It is possible to support many programs that appear to run simultaneously.

The key idea is to protect each process in execution on the computer, and not to allow other processes to interfere, either maliciously or unintentionally.

The operating system has access to all system resources, but must be protected from all other processes.

One sharable asset of particular interest is physical memory.

In a time-sharing system with many processes loaded, the operating system will allocate areas of memory to each process.

In the rare situations in which processes share information through main memory, the operating system will manage that sharing.

NOTE: The name for this slide is based on an operating system for the PDP–11:
RSTS/E (Resource Sharing – Time Sharing / Extended)

# The PDP–11

The 1970's was a decade in which many things happened, including

1. The initial design for the Intel IA–32 architecture.
2. The maturation of the design of modern operating systems.
3. The beginning of a reduction of the cost of a moderately powerful computer.

The PDP–11 was typical of the time period. The computer and its peripheral devices (tape drives, disk drives, printers, etc.) were very expensive.

**Time sharing** was devised as a way to allow multiple users to access this expensive resource.

Because humans were much slower than computers, a reasonable number of users could share use of the computer with the belief that no other users were active.

Each user had a dedicated terminal connected to an I/O processor that managed communication between multiple users and the CPU.

One key design issue was sizing the system to the expected number of users. Too many users could cause the system to be very slow.

# The Microsoft Operating Systems

Many of the design features seen in modern operating systems had their origins in decisions made in the 1970's.

The concept of time sharing has evolved in two directions, not necessarily distinct.

1. The modern server systems, which do serve a large number of users at once. (Think of mail servers, e–commerce, etc.).

2. The individual personal computer, with one human user. Think of the many processes running on the computer as "users", who are not humans. (Think of e–mail clients, web clients, display management, the clock, etc.)

The point is that algorithms and data structures developed to support the sharing of a single computer by many human users have proven very useful for use in the modern single–user computer.

In particular, the many special–purpose registers seen in a modern IA–32 design reflect the need for fast access to data used in process management.

# IA-32 Modes

It was noted in an earlier slide that the concept of a process as representing all of the resources required for execution does not explicitly include the memory.

Memory is a resource shared by many processes. The process image hold only a set of pointers to memory or a table of memory access data structures. In other words:

1. The process image holds descriptors indicating what areas of physical memory it may access and, optionally, its access rights to each area.
2. The operating system manages memory on behalf of each process.

Memory management involves hardware resources to support the operating system. This includes a set of specialized registers.

Quite often the hardware and OS designs are based on a number of modes, each representing a fixed set of strategies to support a particular use.

The later IA–32 implementations, including all Pentium models, supported three memory segmentation modes to facilitate memory management by the operating system. These are **real mode**, **protected mode**, and **virtual 8086 mode.**

# Real Mode

**Real mode,** also called "real–address mode" implements the programming mode of the Intel 8086 almost exactly, with a few extra features to allow switching to other modes.

In real mode, only one megabyte can be addressed, from 0x00000 to 0xFFFFF. This is a 20–bit address formed using the segment registers paired with pointer registers.

For example, the 16–bit CS register is paired with the 16–bit IP register.

| | | | | |
|---|---|---|---|---|
| CS = **`0x1234`** | Shift left by 4 bits | **`0x12340`** | | |
| IP = **`0x5001`** | Add 4 leading 0 bits | **`0x05001`** | Address = **`0x17341`** | |

In this mode, the segment registers are used only to compute 20-bit addresses.

This mode, when available, can be used to run MS–DOS programs that require direct access to system memory and hardware devices.

If a program running in real mode crashes, it can take the entire operating system with it. All other programs crash also, and the computer ceases to respond to input.

Modern operating systems "boot up" in real mode to allow for direct access to the hardware resources, before changing over to protected mode for program execution.

# Protected Mode

**Protected mode** is the native state of the Pentium processor, in which all instructions and features are available.

As opposed to the 20–bit **segment:offse**t addressing mode used in real mode, the native addressing for protected mode is called the **flat segmentation model**.

In the flat segmentation model on the IA–32:

1. Memory addresses are treated as single 32–bit integers.
2. The segment registers point to segment descriptor tables, and are not directly used in address calculations.

CS now contains the address of the descriptor table for the code segment.

DS now contains the address of the descriptor table for the data segment.

SS now contains the address of the descriptor table for the stack segment.

Programs are given separate memory areas called **segments**; the processor uses the segment registers and descriptor tables to manage access to memory, so that no program can reference memory outside its assigned area.

# Virtual 8086 Mode

**Virtual 8086** mode is a sub–mode of protected mode. In this mode, many of the protection features of protected mode are active.

In this mode, each process has its own 1MB address space that simulates that provided by real–address mode.

Unlike real–address mode, the address space in virtual 8086 mode comprises virtual addresses that are managed by the MMU (Memory Management Unit) for conversion into physical addresses.

The MMU is controlled by the operating system, and should be viewed as a hardware component of the OS.

In modern MS–Windows systems, this can be accessed the **command window**.

The processor can execute most real–mode software in a safe multitasking environment.

If a virtual 8086 mode process crashes or attempts to access memory in areas reserved for other processes or the operating system, it can be terminated without adversely affecting any other process.

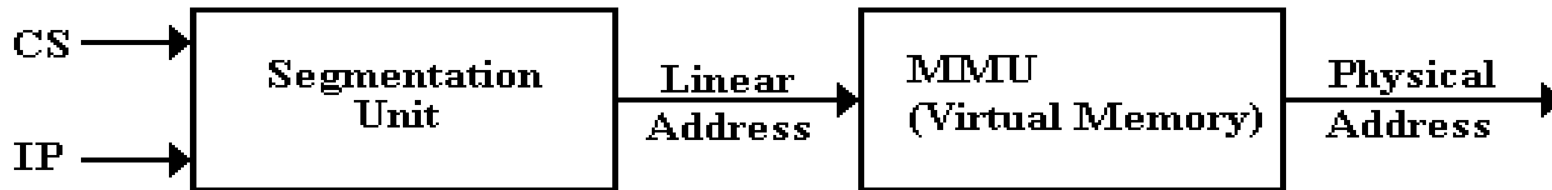
# Types of Addresses in the IA–32

We have mentioned two methods for address generation used in the IA–32 designs.

The segment:offset address method is used to convert a 16–bit segment address and a 16–bit pointer address into a 20–bit address.

The flat address model uses a 32–bit address.

Here is some terminology commonly used when discussing IA–32 designs.
Here we imagine an address of an executable statement.




The **segmentation uni**t translates the older segment:offset addresses into the more modern linear addresses.

A **linear address** is a single 32–bit unsigned integer.
Presumably all addresses in the flat segmentation model are linear addresses.

The MMU, part of the virtual memory system of the OS, converts this 32–bit linear address into a 32–bit **physical address** that is the actual address in main memory.

# Segment Descriptor Tables

Each segment is represented by an 8–byte (64 bit) **segment descriptor**.

Segment descriptors are stored in either the **GDT** (**Global Descriptor Table**) or **LDT** (**Local Descriptor Table**).

Usually, only one GDT is defined, presumably for use by the operating system. Each process commonly has an associated LDT.

Commonly each descriptor table would contain at least three descriptors, one each for a code segment, a data segment, and a stack segment.

Access to these tables is through two processor registers.

The **GDTR** (Global Descriptor Table Register) contains the address of the GDT.

The **LDTR** (Local Descriptor Table Register) contains the address of the LDT in current use by the executing process.

# Segment Descriptors

Each 64–bit segment descriptor has the following fields.

The **32–bit Base field** contains the linear address of the first byte of the segment.

The **1–bit G** (granularity) flag is used in computing the limit address.

The **20–bit Limit field** denotes the maximum size of the segment.
    If G = 0, this is the maximum size in bytes, varying from 1 byte to 1 MB.
    If G = 1, this is the maximum size in units of 4 KB; 4 KB to 4 GB.

The **1–bit S** (system flag)
    If S = 0, the segment stores kernel (operating system) data structures.
    If S = 1, the segment stores user program data structures.

The **4–bit Type field**, characterizing the segment. Common types include:

| | |
|---|---|
| Code Segment Descriptor, | used to refer to a code segment. |
| Data Segment Descriptor, | used to refer to either a data segment or stack segment. |

The **2–bit DPL** (Descriptor Privilege Level), used to restrict access to the segment.
    If DPL = 0, only kernel mode (operating system) code can access this segment.
    If DPL = 3, any code can access the segment.

# Virtual Memory

The term "**virtual memory**" refers to a mechanism by which the operating system can manage physical memory and provide increased security.

The term "**virtual memory**" is commonly defined operationally in terms of disk drives used as secondary memory. We shall explore that useful definition later.

Here, we shall give the precise definition. **Virtual memory** is a mechanism by which the Operating System translates addresses generated by an executing process into actual physical memory addresses.

Consider a program running in Virtual 8086 mode. The program issues an address that is then combined with the contents of a segment register to create a **linear address**.

This 32-bit linear address has the form of an IA-32 memory address, but it is not.

This linear address is passed to the **MMU** (Memory Management Unit), controlled by the Operating System, for conversion into a physical memory address.

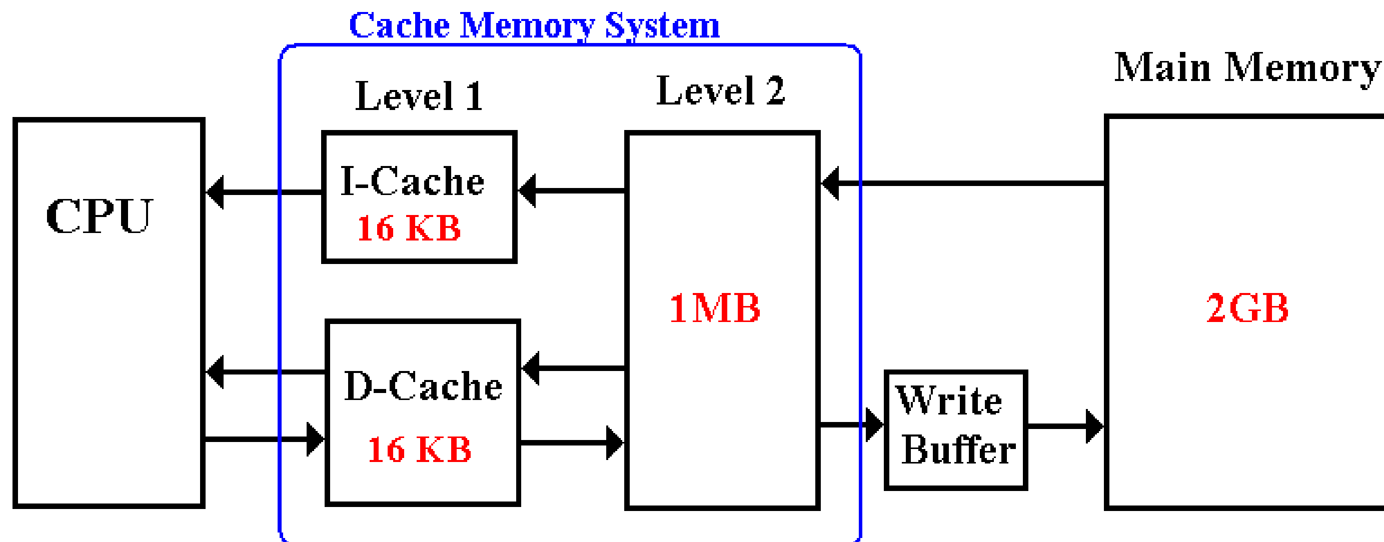
Suppose two memory-resident programs each issue linear address 0x1000. The OS can map one of these to physical address 0x101000 and the other to 0x201000.

For security, the OS can set aside a block of physical memory and not map any user program linear address to that block.

# Cache Memory

At this point, we describe the cache configuration found on a Pentium and give a very general description of its advantages. A future lecture will cover the topic more fully.

Each Pentium product is packaged with a cache memory system designed to optimize memory access to both data and instructions.

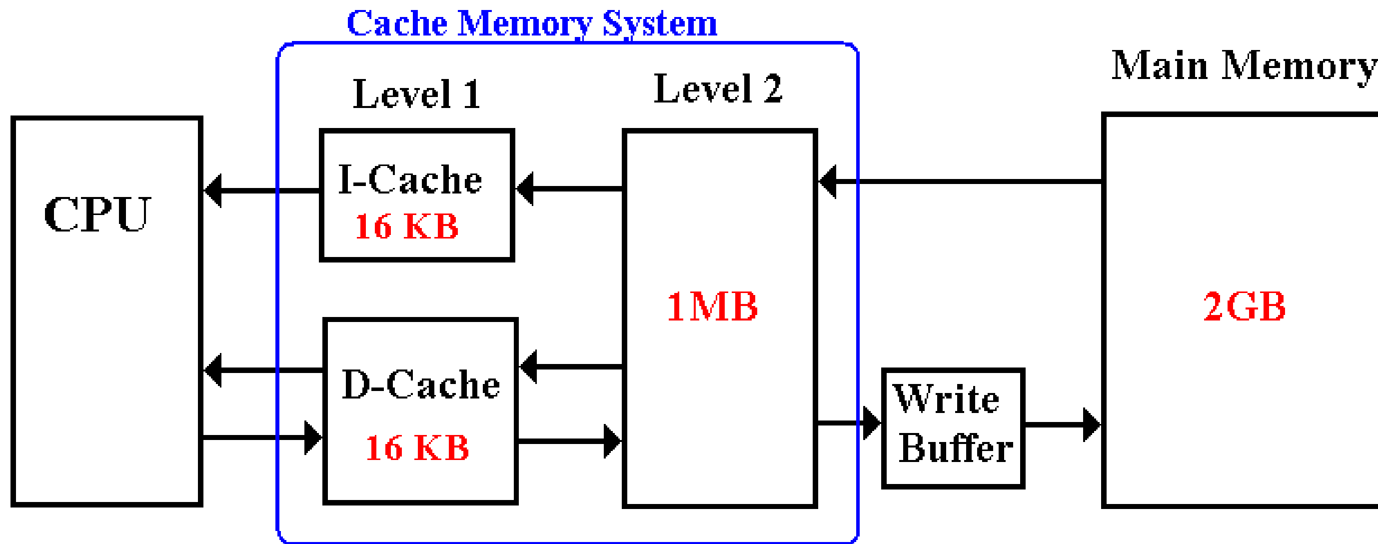



Most of the Pentium designs have a 32kb split L1 (level 1) cache.

The split is between the I-cache for instructions and D-cache for data. This allows a pipelined execution unit to access one instruction and one data item at the same time.

# Cache Memory (Part 2)

We now explain the multi-level nature of the cache.




Because it is smaller, the L1 cache is faster than the L2 cache.
Due to locality, the cache acts as if it is as large as the L2, and only a bit slower than L1.

The cache memory is faster than the main memory.
Due to locality, the system acts as if it is a large memory with the cache access speed.

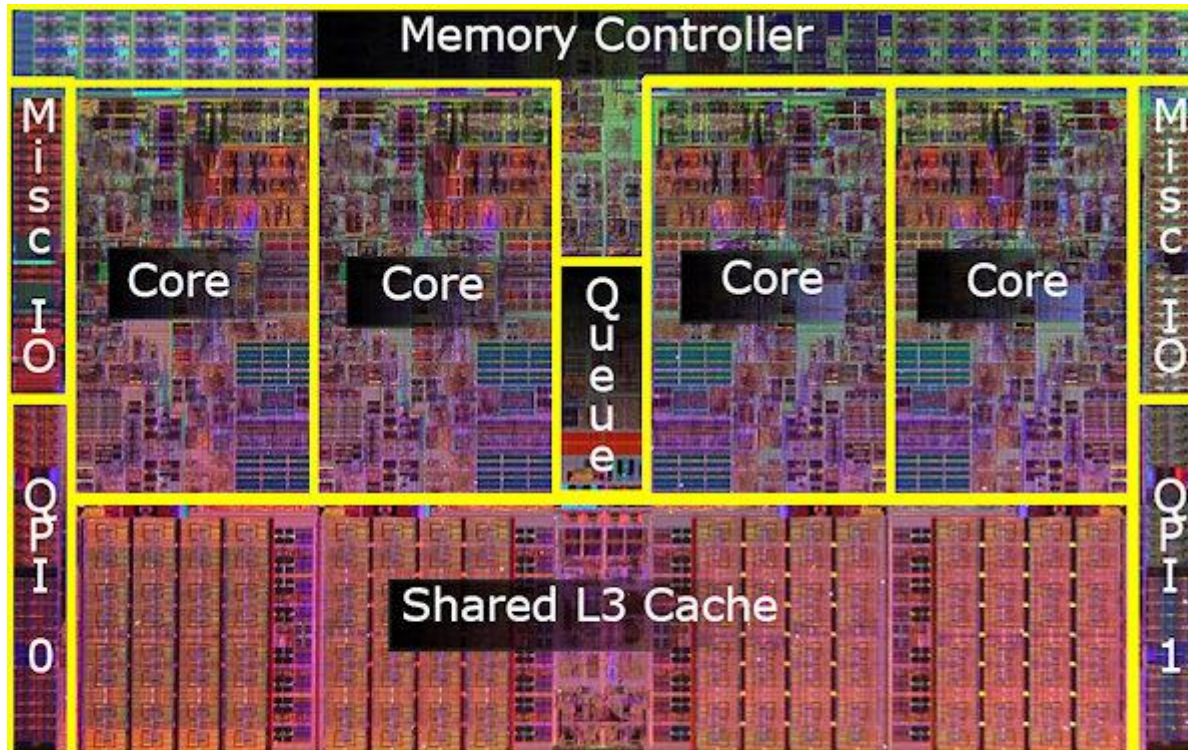
The **write buffer** between the cache memory and main memory allows for short bursts of writing to the main memory at speeds faster than the main memory can take it in.

# Another Level of Cache

Modern multi-core designs add a third level of cache memory.

Each core is a complete CPU with its standard two-level cache.

The L3 cache is shared by all of the cores in the chip.  Here is a quad-core.




On-chip cache speeds up program execution and uses less power than logic chips.

# The Register File

The standard stored program computer design calls for multiple levels of data storage

1. A set of very fast registers, associated with the CPU.
2. The primary memory, which may include cache memory.
3. Backing storage, such as magnetic tapes and disks.

Those registers that can be accessed directly by an assembly language program are called general purpose registers.

This collection of registers is sometimes called the **register file**.

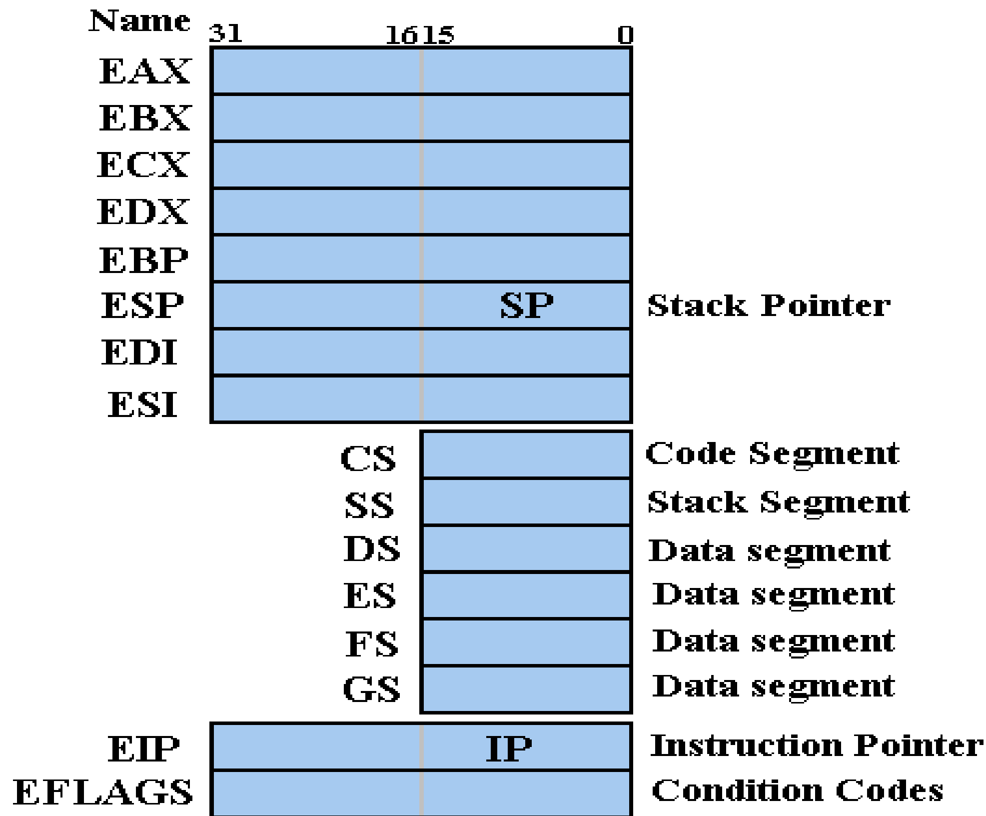
Older designs, such as early Pentiums, had only four general purpose registers. These are EAX, EBX, ECX, and EDX.

It is more common for a CPU to have 16 or 32 registers. Some designs have larger numbers, possibly 128 or 256 registers.

In general, access to register memory is much faster than access to cache or standard memory. For this reason, compilers favor registers for storage when possible.

# The Intel 80386 Register Set

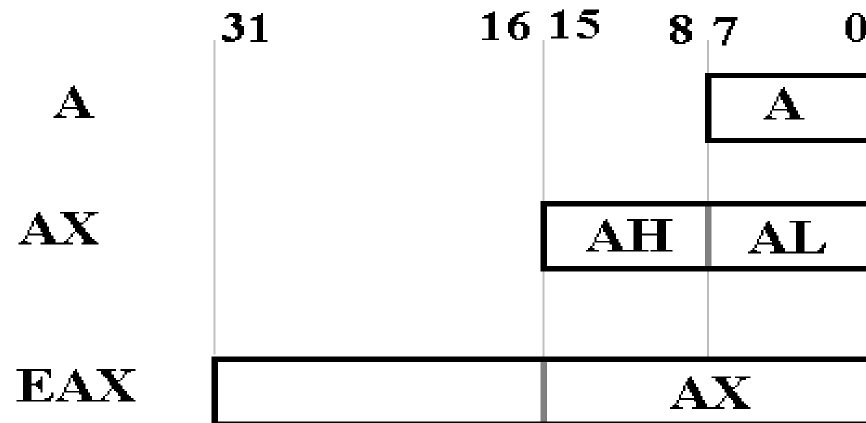
The basic IA-32 design is built around this set of registers.




Note the continued existence of the 16-bit segment registers, allowing 8086 code to run.

# The IA-32 Registers: EAX

**EAX:** This is the general–purpose register used for arithmetic and logical operations. Recall from the previous chapter that parts of this register can be separately accessed. This division is seen also in the EBX, ECX, and EDX registers; the code can reference BX, BH, CX, CL, etc.




This register has an implied role in both multiplication and division.

In addition, the A register (AL in the Intel 80386 usage) is involved in all data transfers to and from the I/O ports.

# EAX Code Samples (Part 1)

```
MOV  EAX, 1234H   ; Set value of EAX to hexadecimal 1234
                  ; The format is destination, source.

CMP  AL, ‘Q’      ; Compare the value in AL (the low
                  ; order 8 bits of EAX to 81,
                  ; the ASCII code for ‘Q’

MOV  ZZ, EAX      ; Copy the value in EAX to memory
                  ; location at address ZZ

DIV  DX           ; Divide the 32-bit value in EAX by the
                  ; 16-bit value in DX.
```

The last example shows the common practice of having an implicit operand; here the accumulator EAX is implicitly referenced by the instruction.

# EAX Code Samples (Part 2)

The EAX register is implicitly a part of any MUL (multiplication) operation.

Here, the product of two integers in 8-bit registers is placed into a 16-bit register.

```
MOV AL,  5H    ; Move decimal 5 to AL
MOV BL, 10H    ; Decimal 16 to BL
MUL BL         ; AX gets the 16-bit number 0050H
               ; (80 decimal).  The MUL instruction
               ; says multiply the value in AL by that
               ; in BL and put the product in AX.
               ; Only BL is explicitly mentioned.
```

16–bit multiplications use AX as a 16–bit register. For compatibility with the Intel 8086, the full 32 bits of EAX are not used to hold the product.

The two 16–bit registers AX and DX are viewed as forming a 32–bit pair and serve to store it. Again, note that AX implicitly holds one of the integers to be multiplied.

```
MOV AX, 6000H  ;
    MOV BX, 4000H  ;
    MUL BX         ; DX:AX = 1800 0000H.
```

# EAX Code Samples (Part 3)

Here is an example showing the use of the AX register (AH and AL) in character input.

```
MOV AH, 1     ; Set AH to 1 to indicate the desired I/O
              ; function:  read a character
              ; from standard input.
INT 21H       ; Software interrupt to invoke an Operating
              ; System function, here the value 21H (33 in
              ; decimal) indicates a standard I/O call.
MOV XX, AL    ; On return from the function call, register
              ; AL contains the ASCII code for a single
              ; character that has just been input.
              ; Store this in memory location XX.
```

Note that this code is not likely to work in Pentium Protected Mode.

# EBX Code Samples

**EBX:** This can be used as a general–purpose register, but was originally designed to be the base register, holding the address of the base of a data structure.

The easiest example of such a data structure is a singly dimensioned array.

```
LEA EBX, ARR   ; The LEA instruction loads the address
               ; associated with a label and not the value
               ; stored at that location.
MOV AX, [EBX]  ; Using EBX as a memory pointer, get the
               ; 16-bit value at that address and load
               ; it into AX (bits 15 – 0 of EAX).
ADD EAX, EBX   ; Add the 32-bit value in EBX to
               ; the 32-bit value in EAX.
```

# ECX Code Samples

**ECX:** This can be used as a general–purpose register, but it is often used in its special role as a counter register for loops or bit shifting operations. This code fragment illustrates its use.

```
      MOV EAX, 0     ; Clear the accumulator EAX
      MOV ECX, 100   ; Set the count to 100 for
                     ; 100 repetitions of the loop
TOP:  ADD EAX, ECX   ; Add the count value to EAX
      LOOP TOP       ; Decrement ECX, test for zero, and jump
                     ; back to TOP if non-zero.
```

At the end of this loop, EAX contains the value 5,050.

# EDX

This can be used as a general–purpose register.

It also plays a special part in executing integer multiplication and division.
For multiplication, DX or EDX store the more significant bits of the product.

The 16–bit implementation of multiplication uses AX to hold one of the integers to be multiplied and uses the register pair DX:AX to hold the 32-bit product.

```
MOV AX, 6000H  ;
MOV BX, 4000H  ;
MUL BX         ; DX:AX = 1800 0000H.
```

The 32–bit implementation of multiplication uses EAX to hold one of the integers to be multiplied and uses the register pair EDX:EAX to hold the 64-bit product.

```
MOV EAX, 12345H
MOV EBX, 10000H
MUL EBX         ; Form the product EAX times EBX
                ; EDX:EAX = 0000 0001 2345 0000H
```

Register DX can also hold the 16–bit port number of an I/O port.

```
MOV DX, 0200H   ; Load port address 200H
IN  AL, DX      ; Get a byte from the port, put in AL
```

# Some Other Registers

**The Index Registers**

The **ESI** and **EDI** registers are used as source and destination addresses for string and array operations.

The ESI "**Extended Source Index**" and EDI "**Extended Destination Index**" facilitate high–speed memory transfers.

**The Instruction Pointer**

The **EIP** register is the 32-bit instruction pointer. Other designs call this register the **PC** (Program Counter), which is the standard terminology.

**The Other Pointers**

The **ESP** register is the 32-bit stack pointer, used to manage push and pop operations.

The **EBP** register is the 32-bit base pointer. In connection with the ESP, the EBP is used to manage the stack frame, that part of the stack used to communicate with subprograms and store local variables.

The **EFLAGS** register holds a collection of at most 32 Boolean flags.

The flags are divided into two broad categories: **control flags** and **status flags**.

# Addressing Modes

Here is a list of some of the more common addressing modes used in IA-32.

**Data Register Direct Mode**
The simplest mode is also the fastest to execute.

```
MOV EAX, EBX    ; Copy the value from EBX into EAX
                ; The value in EBX is not changed.
```

**Immediate Mode**
In this mode, one of the arguments is the value to be used.
Here are some examples, a few of which are not valid.

```
MOV EBX, 1234H  ; EBX gets the value 01234H.
MOV 123H, EBX   ; NOT VALID.  The destination of any
                ; move must be a memory location.
MOV AL, 1234H   ; NOT VALID.  Only one byte can be moved
                ; into an 8-bit register.
                ; This intermediate value is 2 bytes.
```

# More Addressing Modes

**Memory Direct Mode**

In this mode, one of the arguments is a memory location. Here are some examples.

```
MOV ECX, [1234H]   ; Move the value at address 1234H to
                   ; ECX.  Note the square brackets around
                   ; the value.  Not immediate mode.
MOV EDX, WORD1     ; Move the contents of address WORD1
MOV WORD2, EDX     ; Move the contents of the 32-bit
                   ; EDX to memory location WORD2.
MOV X, Y           ; NOT VALID.  Memory to memory moves
                   ; not allowed in this architecture.
```

# More Addressing Modes

**Address Register Direct**

Here, the address associated with a label is loaded into a register. Here are two examples, one of which is memory direct and one of which is address register direct.

```
LEA  EBX, VAR1    ; Load the address associated with VAR1
                  ; into register EBX.
                  ; This is address register direct.

MOV  EBX, VAR1    ; Load value at address VAR1 into EBX.
                  ; This is memory direct addressing.
```

**Register Indirect.**

Here the register contains the address of the argument. Here are some examples.

```
MOV EAX, [EBX]    ; EBX contains the address of a value
                  ; to be moved to EAX.
```

Note that the following two code fragments do the same thing to EAX. Only the first fragment changes the value in EBX.

```
LEA EBX, VAR1    ; Load the address VAR1 into EBX
MOV EAX, [EBX]   ; Load the value at that address into EAX

MOV EAX, VAR1    ; Load the value at address VAR1 into EAX
```

# Based and Indexed Addressing

## Direct Offset Addressing

Suppose an array of 16–bit entries at address AR16. We may employ direct offset in two ways to access members of the array. Here are a number of examples.

```
MOV CX,AR16+2      ; Load the 16-bit value at address
                   ; AR16 + 2 into CX.  For a zero-based
                   ; array, this might be AR16[1].
MOV CX,AR16[2]     ; Does the same thing.  Computes the
                   ; address (AR16 + 2).
```

## Base Index Addressing

This mode combines a base register with an index register to form an address.

```
MOV EAX, [EBP+ESI] ; Add the contents of ESI to that of
                   ; EBP to form the source address.
                   ; Move the 32-bit value at that
                   ; address to EAX.
```

# More Indexed Addressing

**Index Register with Displacement**

There are two equivalent versions of this, due to the way the assembler interprets the second way. Each uses an address, here **`TABLE`**, as a base address.

```
MOV EAX, [TABLE+EBP+ESI] ; Add the contents of ESI to
                         ; that of EBP to form an
                         ; offset, then add that to the
                         ; address associated
                         ; with the label TABLE to get
                         ; the address of the source.

MOV EAX TABLE[ESI]       ; Interpreted as the same
                         ; as the above instruction.
```