Trends in processor architecture: the Intel Pentium

This chapter attempts to trace the evolution of the modern Intel Pentium from the earliest CPU chip, the Intel 4004. The real evolution begins with the Intel 8080, which is an 8–bit design having features that permeate the entire line. Our discussion focuses on three organizations.

- IA–16 The 16–bit architecture found in the Intel 8086 and Intel 80286.
- IA–32 The 32–bit architecture found in the Intel 80386, Intel 80486, and most variants of the Pentium design.
- IA–64 The 64–bit architecture found in some high–end later model Pentiums.

The IA–32 has evolved from an early 4–bit design (the Intel 4004) that was first announced in November 1971. At that time, memory came in chips no larger than 64 kilobits (8 KB) and cost about $1,600 per megabyte. Before moving on with the timeline, it is worth recalling the early history of Intel. Here, we quote extensively from Tanenbaum [R002].

> "In 1968, Robert Noyce, inventor of the silicon integrated circuit, Gordon Moore, of Moore's law fame, and Arthur Rock, a San Francisco venture capitalist, formed the Intel Corporation to make memory chips. In the first year of operation, Intel sold only $3,000 worth of chips, but business has picked up since then."

> "In September 1969, a Japanese company, Busicom, approached Intel with a request for it to manufacture twelve custom chips for a proposed electronic calculator. The Intel engineer assigned to this project, Ted Hoff, looked at the plan and realized that he could put a 4–bit general–purpose CPU on a single chip that would do the same thing and be simpler and cheaper as well. Thus in 1970, the first single–chip CPU, the 2300–transistor 4004 was born."

> "It is worth note that neither Intel nor Busicom had any idea what they had just done. When Intel decided that it might be worth a try to use the 4004 in other projects, it offered to buy back all the rights to the new chip from Busicom by returning the $60,000 Busicom had paid Intel to develop it. Intel's offer was quickly accepted, at which point it began working on an 8–bit version of the chip, the 8008, introduced in 1972."

> "Intel did not expect much demand for the 8008, so it set up a low–volume production line. Much to everyone's amazement, there was an enormous amount of interest, so Intel set about designing a new CPU chip that got around the 8008's limit of 16 kilobytes of memory (imposed by the number of pins of the chip). This design resulted in the 8080, a small, general–purpose CPU, introduced in 1974. Much like the PDP–8, this product took the industry by storm, and instantly became a mass market item. Only instead of selling thousands, as DEC had, Intel sold millions."

The 4004 was designed as a 4–bit chip in order to perform arithmetic on numbers stored in BCD format, which required 4 bits per digit stored. It ran at a clock speed of 108 KHz and could address up to 1Kb (128 bytes) of program memory and up to 4Kb (512 bytes) of data memory.

The later history of the CPU evolution that lead to the Pentium is one of backward compatibility with an earlier processor, in that the binary machine code written for that early model would run unchanged on all models after it. There are two claims to the identity of this early model, some say it was the Intel 8080, and some say the Intel 8086. We begin the story with the 8080.

1974 The Intel 8080 processor is released in April 1974. It has a 2 MHz clock. It had 8–bit registers, and 8–bit data bus, and a 16–bit address bus.
The accumulator was called the "A register".

1978 The Intel 8086 and related 8088 processors are released. Each has 16–bit registers, 16–bit internal data busses, and a 20–bit address bus. Each had a 5MHz clock; the 8088 ran at 4.7 MHz for compatibility with the scan rate of a standard TV, which could be used as an output device. The main difference between the 8086 and the 8088 is the data bus connection to other devices. The 8086 used a 16–bit data bus, while the 8088 used a cheaper and slower 8–bit data bus.

The 16–bit accumulator was called the "AX register". It was divided into two smaller registers: the AH register and AL register so it could run 8080 code.

Neither the 8086 nor the 8088 could address more than one megabyte of memory. Remember that in 1978, one megabyte of memory cost $10,520. According to Bill Gates "Who would need more than 1 megabyte of memory?"

1980 The Intel 8087 floating–point coprocessor is announced. Each of the 80x86 series (8088, 8086, 80286, 80386, and 80486) will use a floating–point coprocessor on a separate chip. A later variant of the 80486, called the 80486DX was the first of the series to including floating–point math on the CPU chip itself. The 80486SX was a lower cost variant of the 80486, without the FPU.

1982 The Intel 80186 was announced. It had a clock speed of 6 MHz, and a 16–bit external data bus. It might have been the successor to the 8086 in personal computers, but its design was not compatible with the hardware in the original IBM PC, so the Intel 80286 was used in the next generation of personal computers.

1982 The Intel 80286 was announced. It extended the address space to 24 bits, for an astounding 16 Megabytes allowed. (Intel should have jumped to 32–bit addressing, but had convincing financial reasons not to do so). The 80286 originally had a 6 MHz clock.

A number of innovations, now considered to be mistakes, were introduced with the Intel 80286. The first was a set of bizarre memory mapping options, which allowed larger programs to run. These were called **"extended memory"** and **"expanded memory"**. We are fortunate that these are now history.

Each of these memory mapping options was based on the use of 64 KB segments. Unfortunately, it was hard to write code for data structures that crossed a segment boundary, possibly due to being larger than 64 KB. The other innovation was a memory protection system, allowing the CPU to run in one of two modes: **real** or **protected**. The only problem is that no software developer elected to make use of these modes.

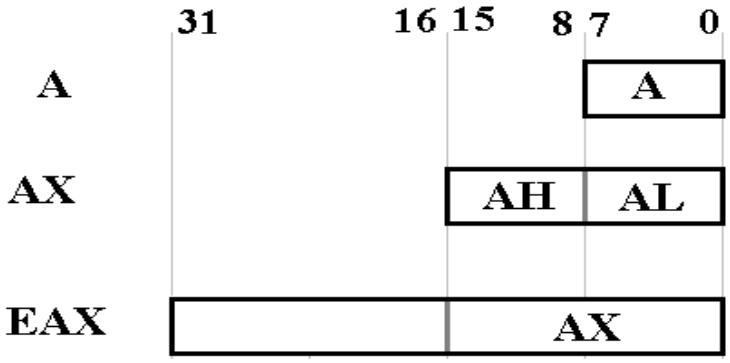
As a result of the requirement for backward compatibility, every IA–32 processor since the 80286 must include this mechanism, even if it is not used.

1983 The introduction of the Intel 80386, the first of the IA–32 family. This CPU had 32–bit registers, 32–bit data busses, and a 32–bit address bus.
The 32–bit accumulator was called the "EAX register".

The Intel 80386 was introduced with a 16 MHz clock. It had three memory protection modes: **protected**, **real**, and **virtual**. We now have three protection modes to ignore.

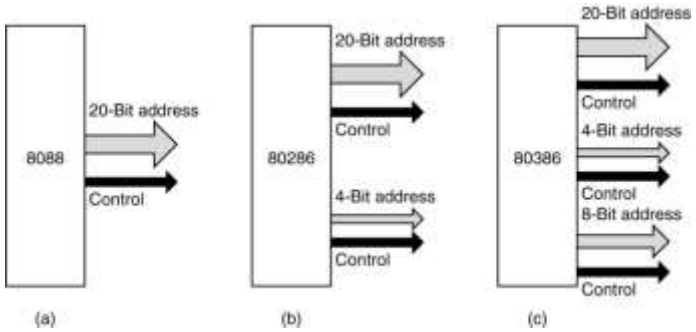
Lesson: The hardware should evolve along with the system software (Operating Systems, Run–Time Systems, and Compilers) that uses it.

Here is the structure of the EAX register in the Intel 80386 and all of the following computers in the IA–32 line. This structure shows the necessity to have backward compatibility with the earlier models. The 16–bit models had a 16–bit accumulator, called AX. The 8–bit model had an accumulator, called A, that is now equivalent to the AL 8–bit register.



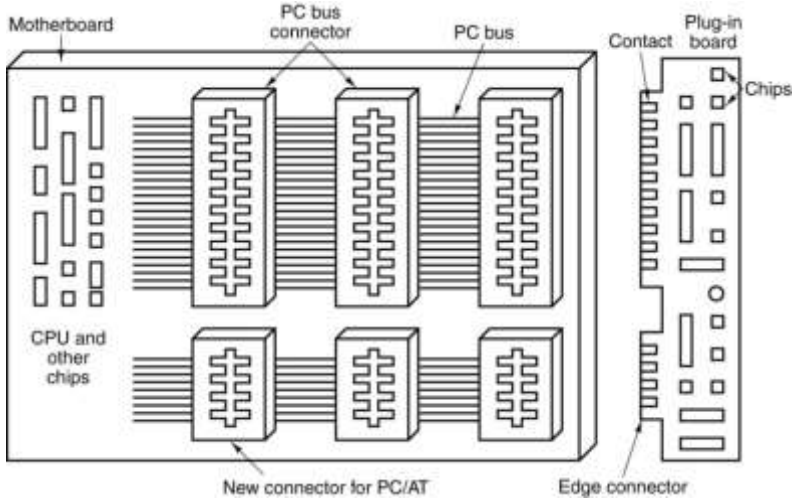
Structure of the EAX register in the Intel 80386. There is no name for the high–order 16 bits of EAX. The AX, AH, and AL registers can all be referenced directly.

Backward Compatibility in the I/O Busses
Here is a figure that shows how the PC bus grew from a 20–bit address through a 24–bit address to a 32–bit address while retaining backward compatibility.



The requirement is that I/O components (printers, disk drives, etc.) purchased for the Intel 8088 should be plug–compatible with both the Intel 80286 and Intel 80386. Those purchased for the Intel 80286 should be plug–compatible with the Intel 80386. The basic idea is that one is more likely to buy a new computer if the old peripheral devices can still be used.

Here is a picture of the PC/AT (Intel 80286) bus, showing how the original configuration was kept and augmented, rather than totally revised. Note that the top slots can be used by the older 8088 cards, which do not have the "extra long" edge connectors. This cannot be used with cards for the Intel 80386; that would be "forward compatibility"



The Intel 80386 was the first of the IA–32 series. In you instructor's opinion, it was the first "real computer CPU" produced by Intel. The reason for this opinion is that it was the first of the series that had enough memory and a large enough address space to remove the need for some silly patches and kludges, such as extended and expanded memory.

1989 The Intel 80486 is introduced. It was aimed at higher performance. It was the first of the Intel microprocessors to contain one million transistors. As noted above, later variants of the 80486 were the first to incorporate the floating point unit in the CPU core.

1992 Intel attempts to introduce the Intel 80586. Finding that it could not get a trademark on a number, Intel changed the name to "Pentium". The name "80586" was used briefly as a generic name for the Pentium and its clones by manufacturers such as AMD.

1995 The Pentium Pro, a higher performance variant of the Pentium was introduced. It had four new instructions, three to support multiprocessing.

1997 The MMX (Multimedia Extensions) set of 57 instructions was added to both the Pentium and the Pentium Pro. These facilitate graphical and other multimedia computations.

1999 The Pentium III was introduced, with the SSE (Streaming SIMD Extensions) instruction set. This involved the addition of eight 128–bit registers, each of which could hold four independent 32–bit floating point numbers. Thus four floating point computations could be performed in parallel.

2001 The Pentium 4 was shipped, with another 144 instructions, called SSE2.

2003 AMD, a producer of Pentium clones, announced its AMD64 architecture to expand the address space to 64 bits. All integer registers are widened to 64 bits. New execution modes were added to allow execution of 32–bit code written for earlier models.

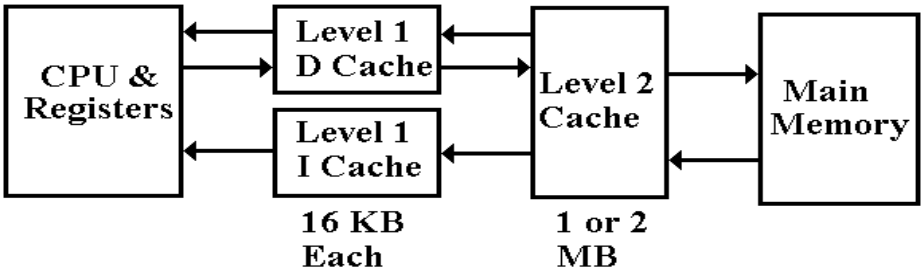
2004 Intel adopts the AMD64 memory model, relabeling it EM64T (Extended Memory 64 Technology).

Most of the IA–32 improvements since this time have focused on providing graphical services to the game playing community. Your instructor is grateful to the gamers; they have turned high–end graphical coprocessors into commodity items. One can get very good graphics card really cheap. Consider the NVIDIA GeForce 8600 graphics processor with 512 MB of 400 MHz graphics memory (DDR transferring 32 bytes per clock cycle), a 675 MHz graphics processor, supporting 2048 by 1536 resolution. It costs $210 bundled with software.

The Trace Cache

All implementations of the Pentium architecture include at least two levels of cache memory. While we plan to discuss this topic in some detail later in this text, we must bring it up now in order to focus on a development in the architecture that began with the Pentium III in 1999.

The earlier Pentium designs called for a two–level cache, with a split L1 cache. There was a 16 KB L1 instruction cache and a 16 KB L1 data cache. Having the split L1 cache allowed the CPU to fetch an instruction and access data in the same clock pulse. (Memory can do only one thing at a time, but two independent memories can do a total of two things at a time.) Here is a figure showing a typical setup. Note that the CPU does not write to the Instruction Cache.



By the time that the Pentium III was introduced, Intel was having increasing difficulty in obtaining fast execution of its increasingly complex machine language instructions. The solution was to include a step that converted each of the complex instructions into a sequence of simpler instructions, called **micro–operations** in Intel terminology. These simpler operations seem to be designed following the RISC (Reduced Instruction Set Computer) approach. Because these micro–operations are simpler than the original, the CPU control unit to interpret them can be hardwired, simpler, and faster.

By the time the Pentium 4 was introduced, this new design approach had lead to the replacement of the 16 KB Level–1 Instruction Cache with the **ETC** (**E**xecution **T**race **C**ache). Unlike the Instruction cache, which holds the original Pentium machine language instructions, the ETC holds the micro–operations that implement these instructions.

Sixteen–bit Addressing

The Intel 8086 and later use a segmented address system in order to generate addresses from 16–bit registers. Each of the main address registers was paired with an offset. The **IP** (Instruction Pointer) register is paired with the **CS** (Code Segment) register.
Each of the IP and CS is a 16–bit register in the earlier designs.

NOTE:The Intel terminology is far superior to the standard name, the PC (Program Counter), which is so named because it does not count anything.

The **SP** (Stack Pointer) register is paired with the **SS** (Stack Segment) register.

The Intel 8086 used the segment:offset approach to generating a 20–bit address.
The steps are as follows.

1. The 16–bit value in the segment register is treated as a 20–bit number with four leading binary zeroes. This is one hexadecimal 0.
2. This 20 bit value is left shifted by four, shifting out the high order four 0 bits and shifting in four low order 0 bits. This is equivalent to adding one hexadecimal 0.
3. The 16–bit offset is expanded to a 20–bit number with four leading 0's and added to the shifted segment value. The result is a 20–bit address.

Example: CS = 0x1234 and IP = 0x2004.

CS with 4 trailing 0's:	**0001 0010 0011 0100 0000**	or **0x12340**
IP with 4 leading 0's:	**0000 0010 0000 0000 0100**	or **0x02004**
Effective address:	**0001 0100 0011 0100 0100**	or **0x13344**

Thirty–Two Bit Addressing
All computers in the IA–32 series must support the segment:offset method of addressing in order to run legacy code. This is **"backwards compatibility"**.

The native addressing mode in the IA–32 series is called a "flat address space". The 16–bit IP (Instruction Pointer) is now the lower order 16 bits of the EIP (Extended Instruction Pointer), which can be used without a segment. The 16–bit SP (Stack Pointer) is now the lower order 16 bits of the ESP (Extended Stack Pointer), which also can be used without a segment.

This diversity of addressing modes has given rise to a variety of "memory models" based on the addressing needed for code and data.

Memory Models: These are conventional assembly language models based on the size of the code and the size of the data.

Code Size	**Data Size**	**Model to Use**
Under 64 KB	Under 64 KB	Small or Tiny
Over 64KB	Under 64 KB	Medium
Under 64 KB	Over 64 KB	Compact
Over 64 KB	Over 64 KB	Large

The smaller memory models give rise to code that is more compact and efficient.

The IA–32 Register Set
The IA–32 register set contains eight 32–bit registers that might be called "general purpose", though they retain some special functions. These registers are: EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI. These are the 32–bit extensions of the 16–bit registers AX, BX, CD, DX, SP, BP, SI, and DI.

The 16–bit segment registers (CS, DS, SS, ES, FS and GS) appear to be retained only for compatibility with earlier code.

In the original Intel 8086 design, the AX register was considered as a single accumulator, with the other registers assigned supporting roles. It is likely that most IA–32 code maintains this distinction, though it is not required.

The IA–64 Architecture

The IA–64 architecture is a design that evolved from the Pentium 4 implementation of the IA–32 architecture. The basic issues involve efficient handling of the complex instruction set that has evolved over the 35 year evolution of the basic design. The IA–64 architecture is the outcome of collaboration between Intel and the Hewlett–Packard Corporation. In some sense, it is an outgrowth of the Pentium 4.

The IA–64 architecture has many features similar to RISC, but with one major exception: it expects a sophisticated compiler to issue machine language that can be exploited by the superscalar architecture. (Again, we shall discuss this in the future.) The current implementations of the IA–64 are called the "Itanium" and "Itanium 2". One wonders if the name is based on that of the element Titanium. In any case, the geeks soon started to call the design the "Itanic", after the ship "Titanic", which sank in 1912. The Itanium was released in June 2001: the Itanium 2 in 2002.

Here are some of the features of the IA–64 design.

1. The IA–64 has 128 64–bit integer registers and 128 82–bit floating–point registers.
2. The IA–64 translates the binary machine language into 128–bit instruction words that represent up to three assembly language instructions that can be executed during one clock pulse. A sophisticated compile emits these 128–bit instructions and is responsible for handling data and control dependencies. More on this later.
3. The design might be called "VLIW" (Very Long Instruction Word) except that Intel seems to prefer "EPIC" (Explicitly Parallel Instruction Computing).
4. The design allows for predicated execution, which is a technique that can eliminate branching by making the execution of the instruction dependent on the predicate. There are sixty–four 1–bit predicate registers, numbered 0 through 63. With one exception, each can hold a 0 (false) or a 1 (true). Predicate register **pr0** is fixed at 1 (true). Any instruction predicated on **pr0** will always execute.

For a full appreciation of predication, one would have to understand the design of a pipelined CPU, especially the handling of control hazards. This is a topic for a graduate course. Here we shall just give a simple code example to show how it works. Consider the statement: **`if (p) then S1 else S2`** ; where S1 and S2 are statements in the high level language.

Under normal compilation, this would be converted to a statement to test the **predicate** (Boolean expression that can be either true or false), execute a conditional branch around statement S1, and follow S1 by an unconditional branch around S2. In predication, the compilation is simpler, and equivalent to the following two statements.

```
(p)  S1 ;   // Do this if the predicate is true.
(~p) S2 ;   // Do this if the predicate is false.
```

The execution of this pair of statements is done together, in parallel with evaluation of the predicate. Depending on the value of the predicate, the effect of one of the instructions is committed to memory, and the results of the other statement are discarded.

One of the goals of advanced architectures is the execution of more than one instruction at a time. This approach, called **"superscalar"**, must detect which operations can be executed in parallel, and which have data dependencies that force sequential execution.

In the design of pipelined control units, these data dependencies are called "**data hazards**". Here is an example of a pair of instructions that present a data hazard.

```
x = y – z ;
w = u + x ;
```

Note that these two instructions cannot be executed in parallel, while the next pair can be so executed. In the first set, the first instruction changes the value of **x**. Parallel execution would use the old value of **x** and hence yield an incorrect result.

```
x = y – z ;
w = u + z ;   // This pair can be executed in parallel.
```

Early designs, dating back to the CDC-6600, used a hardware mechanism to detect which instructions could be executed in parallel. The difficulty with this approach is the increasing complexity of such a control unit, leading to slower execution. The IA–64 strategy is called "**explicit parallelism**", in which the compiler statically schedules instructions for parallel execution at compile time, rather than the control unit do so dynamically at run time.

IA–64 calls for the compiler to emit 128–bit **bundles**, each containing three instructions, and a template that defines which of the parallel execution units are to be used. Each of the three instructions in a bundle is called a **syllable**. Here is the structure of a bundle.

Instruction Slot 2	Instruction Slot 1	Instruction Slot 0	Template
41 bits	41 bits	41 bits	5 bits

Here is the instruction of a syllable, fit into an instruction slot.

Major Opcode	More of the instruction	Predicate Register
4 bits	31 bits	6 bits

Multicore Processors

The CPU design problem, called the "**power wall**" was discussed in chapter 1 of this text. The commonly accepted solution to this problem, in which designers try to get more performance from a CPU without overheating it, is called a **multicore CPU**. This is basically a misnomer, because each core in a multicore CPU is an independent CPU. Thus a quad–core Pentium chip actually contains four CPUs. Examples of this strategy include the Intel iCore3, iCore5, and iCore7 (sometimes called "Core i3", "Core i5", and "Core i7") designs. The i3 is the entry level processor, with two cores. The i5 is a mid–range processor with 2 to 4 cores. The i7 is considered to be the high–end processor, with 2 to 6 cores.

Motherboards and slots

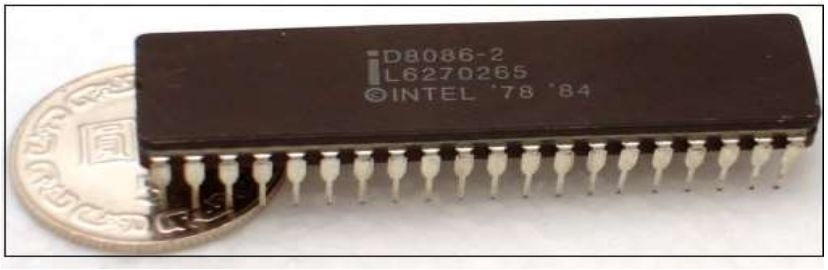
Along with the evolution of the CPU chips, we see an evolution of the support hardware. Here we study the hardware used to integrate the CPU into the system as a whole.

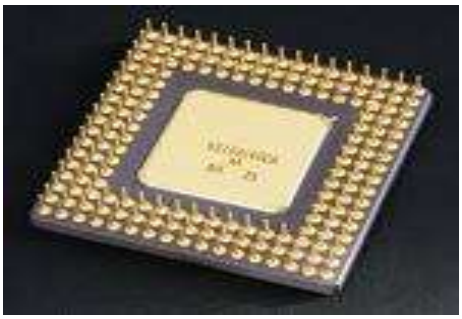
Sockets

A **CPU socket** or **CPU slot** is a mechanical component that provides mechanical and electrical connections between a device (usually a microprocessor) and a printed circuit board (PCB), or motherboard. This allows the CPU to be replaced without risking the damage typically introduced when using soldering tools [R016]. Common sockets utilize retention clips that are designed to apply a constant force, which must be overcome when a device is inserted. For chips that sport a high number of pinouts, either zero-insertion force (ZIF) sockets or land grid array (LGA) sockets are used instead. These designs apply a compression force once either a handle (for ZIF type) or a surface plate (LGA type) is put into place. This provides superior mechanical retention while avoiding the added risk of bending pins when inserting the chip into the socket.

CPU sockets are used in desktop computers (laptops typically use surface mount CPUs) because they allow easy swapping of components, they are also used for prototyping new circuits.

The earliest sockets were quite simple, in fact they were DIP (Dual In–line Pin) devices. A typical CPU for such a slot might be an Intel 4004 or an Intel 8086 (with different pin count). Here is a picture of the Intel 8086, showing one of the two rows of pins.





The complexity of the IA–32 series processors grew as the series evolved, and the number of pin–outs required grew with it. By the time of the Intel 80486, a DIP arrangement was impossible. Here is a picture of the Intel 80486DX2.

The Intel 80486 had 196 pins arranged as a hollow rectangle. It should be obvious that it required more than a DIP socket.

The sockets for the late Intel 80x86 series and the early Pentium series came in a number of sizes in order to accommodate the number of pins on the chip.

Here is a table of some of the early sockets used for the IA–32 series.

Socket name	Year introduced	CPU families	Package	Pin count	Bus speed
DIP	1970s	Intel 8086, Intel 8088	DIP	40	5/10 MHz
Socket 1	1989	Intel 80486	PGA	169	16–50 MHz
Socket 2	?	Intel 80486	PGA	238	16–50 MHz
Socket 3	1991	Intel 80486	PGA	237	16–50 MHz
Socket 4	?	Intel Pentium	PGA	273	60–66 MHz
Socket 5	?	Intel Pentium, AMD K5	PGA	320	50–66 MHz
Socket 6	?	Intel 80486	PGA	235	?
Socket 7	1994	Intel Pentium, Intel Pentium MMX, AMD K6	PGA	321	50–66 MHz
Socket 8	1995	Intel Pentium Pro	PGA	387	60–66 MHz

Slots

With the introduction of the Pentium II CPU, the transition from socket to slot had become necessary. With the Pentium Pro, Intel had combined processor and cache dies in the same package, connected by a full-speed bus, resulting in significant performance benefits. Unfortunately, this method required that the two components be bonded together early in the production process, before testing was possible. As a result, a single, tiny flaw in either die made it necessary to discard the entire assembly, causing low production yield and high cost.

Intel subsequently designed a circuit board where the CPU and cache remained closely integrated, but were mounted on a printed circuit board, called a Single-Edged Contact Cartridge (SECC). The CPU and cache could be tested separately, before final assembly into a package, reducing cost and making the CPU more attractive to markets other than that of high-end servers. These cards could also be easily plugged into a Slot 1, thereby eliminating the chance for pins of a typical CPU to be bent or broken when installing in a socket.

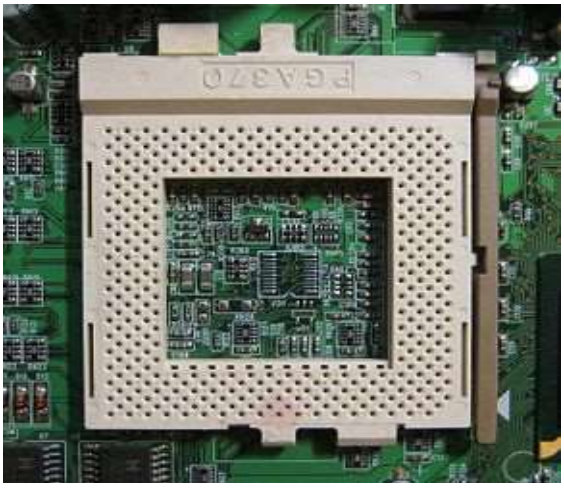
Slot 1 refers to the physical and electrical specification for the connector used by some of Intel's microprocessors, including the Pentium Pro, Celeron, Pentium II and the Pentium III. Both single and dual processor configurations were implemented. Slot 1 (also Slot1 or SC242) is a Slot-type connector with 242 contacts. This connector was designed for Pentium II family of processors, and later used for Celeron budget line of processors. Pentium III was the last microprocessor family that used the Slot 1. For its next generation of Pentium processors - Pentium 4, Intel completely abandoned the Slot1 architecture. The fastest processor that can be used in the Slot 1 motherboards is the Pentium III 1133 MHz with 133 MHz FSB [R012]. The picture on the left shows a typical slot 1 connector mounted on a motherboard. The picture at right shows a CPU mounted in the slot, along with its rather large cooling fans.

Slot 1 connector is 5.23" long (13.29 cm). Besides the actual connector, the Slot 1 also includes SEC cartridge retention mechanism required to support a processor in SEC cartridge and a heatsink. Maximum supported weight of the processor with the heatsink is 400 grams.

Slot 2 refers to the physical and electrical specification for the 330-lead Single Edge Contact Cartridge (or edge-connector) used by some of Intel's Pentium II Xeon and certain models of the Pentium III Xeon. When first introduced, Slot 1 Pentium IIs were intended to replace the Pentium and Pentium Pro processors in the home, desktop, and low-end SMP markets. The Pentium II Xeon, which was aimed at multiprocessor workstations and servers, was largely similar to the later Pentium IIs, being based on the same P6 Deschutes core, aside from a wider choice of L2 cache ranging from 512 to 2048 KB and a full-speed off-die L2 cache (the Pentium 2 used cheaper 3rd party SRAM chips, running at 50% of CPU speed, to reduce cost).

Because the design of the 242-lead Slot 1 connector did not support the full-speed L2 cache of the Xeon, an extended 330-lead connector was developed. This new connector, dubbed 'Slot 2', was used for Pentium 2 Xeons and the first two Pentium III Xeon cores, codenamed 'Tanner' and 'Cascades'. Slot 2 was finally replaced with the Socket 370 with the Pentium III Tualatin; some of the Tualatin Pentium IIIs were packaged as 'Pentium III' and some as 'Xeon', despite the fact they were identical [R014].

Socket 370 (also known as the **PGA370 socket**) is a common format of CPU socket first used by Intel for Pentium III and Celeron processors to replace the older **Slot 1** CPU interface on personal computers. The "370" refers to the number of pin holes in the socket for CPU pins. Modern Socket 370 fittings are usually found on Mini-ITX motherboards and embedded systems [R015]. Here is a picture of the PGA370 socket.



The socket is a ZIF (Zero Insertion Force) type, designed for easy insertion. As noted, it has 370 pin connectors.

The dimensions are 1.95 inches by 1.95 inches, or approximately 5 centimeters on a side.

This was designed to work with a Front Side Bus operating at 66, 100, or 133 MHz. The design voltage range is 1.05 to 2.10 volts.

The mass of the Socket 370 CPU cooler should not exceed 180 grams (a weight of about 6.3 ounces) or damage to the die may occur.

The **LGA 775**, also known as **Socket T**, is one of the latest and largest Intel CPU sockets . LGA stands for **land grid array.** Unlike earlier common CPU sockets, such as its predecessor Socket 478, the LGA 775 has no socket holes; instead, it has 775 protruding pins which touch contact points on the underside of the processor (CPU).

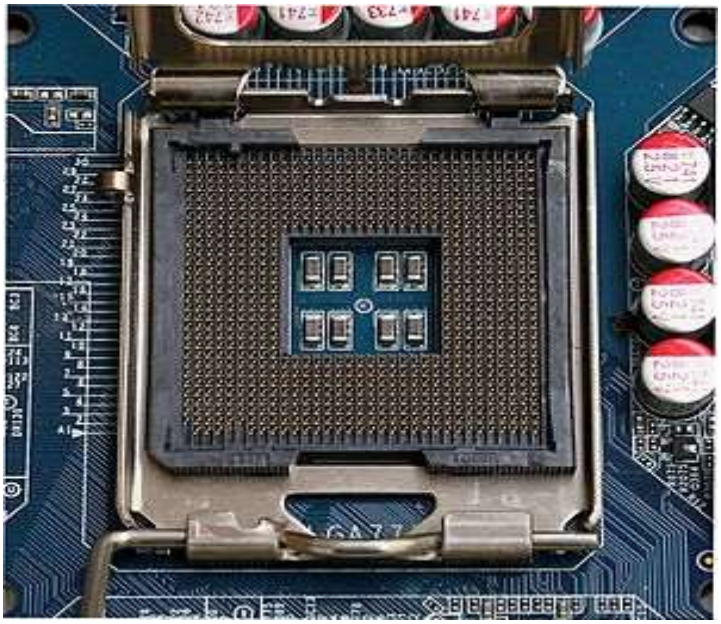
The Prescott and Cedar Mill Pentium 4 cores, as well as the Smithfield and Presler Pentium D cores, used the LGA 775 socket. In July 2006, Intel released the desktop version of the Core 2 Duo (codenamed Conroe), which also uses this socket, as does the subsequent Core 2 Quad. Intel changed from Socket 478 to LGA 775 because the new pin type offers better power distribution to the processor, allowing the front side bus to be raised to 1600 MT/s. The 'T' in Socket T was derived from the now cancelled Tejas core, which was to replace the Prescott core. Another advantage for Intel with this newer architecture is that it is now the motherboard which has the pins, rather than the CPU, transferring the risk of pins being bent from the CPU to the motherboard.

The CPU is pressed into place by a "load plate", rather than human fingers directly. The installing technician lifts the hinged "weld plate", inserts the processor, closes the load plate over the top of the processor, and pushes down a locking lever. The pressure of the locking lever on the load plate clamps the processor's 775 copper contact points firmly down onto the motherboard's 775 pins, ensuring a good connection. The load plate only covers the edges of the top surface of the CPU (processor heatspreader). The center is free to make contact with the cooling device placed on top of the CPU.

An examination of the relevant Intel data sheets shows that LGA 775 which is used for consumer level desktops and LGA 771 used for (Xeon based) workstation and server class computers appear to differ only in the placement of the indexing notches and the swap of two address pins. Many pins devoted to functions such as interfacing multiple CPUs are not clearly defined in the LGA 775 specifications, but from the information available appear to be consistent with those of LGA 771. Considering that LGA 775 predated LGA 771 by nearly a year and a half, it would seem that LGA 771 was adapted from LGA 775 rather than the other way around.

The socket has been superseded by the LGA 1156 (Socket H) and LGA 1366 (Socket B) sockets.

Here is a picture from [R017] of the LGA 775 mounted on some sort of motherboard.



IA–32 Memory Segmentation

Early computers ran programs mostly written by the users, with only a small amount of system software to support the user programs. The logical model of execution was that of a single program under execution; the user program would call the needed system routines as needed.

As the operating system evolved, the execution model became more one of parallel processes, perhaps executing sequentially but better considered logically as executing in parallel. The system processes were best seen as separate from the user process, requiring protection from accidental corruption by the user program. Such protection requires some sort of hardware support for memory management.

Basic to the idea of memory management is the definition of ranges of the address space that a particular process can access. In many modern computers, the address space is divided into logical segments. For each logical segment that a process can access, the hardware defines the starting address of that segment, the size of the segment, and access rights owned by the process.

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** [R018, page 586; R019, page 36].

Real mode implements the programming mode of the Intel 8086 almost exactly, with a few extra features to allow switching to other modes. This mode, when available, can be used to run MS–DOS programs that require direct access to system memory and hardware devices. Programs run in real mode can cause the operating system to crash. If a real mode program is one of many running on the computer at the time, all of the other programs crash as well. There is no protection among programs; the computer just stops responding to input. In this mode, the segment registers are used purely to calculate addresses; see the previous sub–chapter.

There is one real–mode data structure that requires discussion, as it will lead to a more general data structure used in protected mode. This is the **IVT** (**I**nterrupt **V**ector **T**able), which is used to activate software associated with a specific I/O device. We shall discuss I/O management, including I/O interrupts and I/O vectors in chapter 9 of this text. Here is the brief description of an input I/O operation to show the significance of the IVT.

1. An I/O device signals the CPU that it is ready to transfer data by asserting a signal called an "**interrupt**". This is asserted low.
2. When the CPU is ready to handle the transfer, it sends out a signal, called an "**acknowledge**" to initiate the I/O process itself.
3. As a first step to the I/O process, the device that asserted the interrupt identifies itself to the CPU. It does this by sending a **vector**, which is merely an address to select an entry in the IVT. The IVT should be considered as an array of entries, each of which contains the address of the program to handle a specific I/O device.
4. The **ISR** (**I**nterrupt **S**ervice **R**outine) appropriate for the device begins to execute.

There is more to the story than this, but we have hit the essential idea of a single IVT to manage the input and output for all executing programs.

Protected mode is the native state of the Pentium processor, in which all instructions and features are available. Programs are given separate memory areas called **segments**, and the processor uses the segment registers and associated other registers to manage access to memory, so that no program can reference memory outside its assigned area. The operating system is thus protected from intrusion by user programs. The operating system operates in a privileged state in which it can change the segment registers in order to access any area of memory.

Virtual 8086 mode is a sub–mode of protected mode. In this mode, many of the protection features of protected mode are active. The processor can execute 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.

In protected mode, and its sub–mode virtual 8086 mode, each process is assigned a separate **session**, which allows for proper management of its resources. Part of that management involves creation of a separate IVT for that session, allowing the Pentium to allocate different I/O services to separate sessions. More importantly it provides protection against software crashes.

Windows XP can manage multiple separate virtual 8086 sessions at the same time, possibly in parallel with execution of programs in protected mode. This idea has been extended successfully to that of a **virtual machine**, in which a number of programs can execute on a given machine without affecting other programs in any way. The large IBM mainframes, including the z/9 and z/10, call this idea an **LPAR** (**L**ogical **Par**tition).

One key logical component of the virtual machine idea has yet to be discussed; this is called **virtual memory**. This will be discussed fully in chapter 12 of this textbook. There is one important point that can be restated even at this early stage. The program generates addresses that are modified by the operating system into actual addresses into physical memory. As a result, the operating system controls access to real physical memory and can use that control to enhance security.

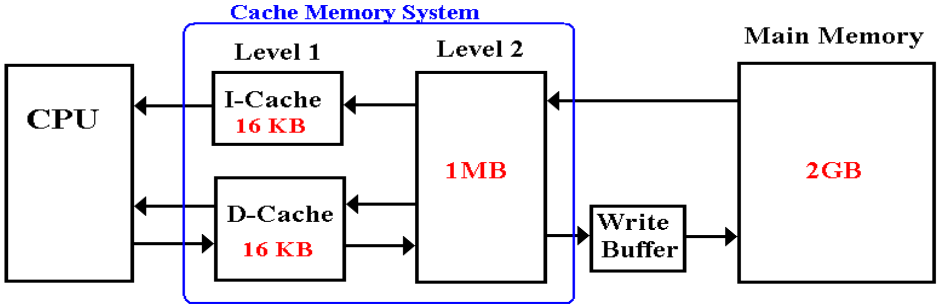
In protected mode, as well as in its sub–mode virtual 8086, addresses to physical memory are generated in a number of steps. Three terms related to this process are worth mention: the effective address, linear address, and physical address. With the exception of the term "**physical address**", which references the actual address in the computer memory, the terms are somewhat contrived. In the IA–32 designs, the **effective address** is the address generated by the program before modification by the memory management unit. The rules for generation of this address are specified by the syntax of the assembly language.

The effective address is passed to the memory management unit, first to the segmentation unit, which accesses the segment registers to create the **linear address** and then accesses a number of other **MMU** (**M**emory **M**anagement **U**nit) registers to determine the validity of the address value and the validity of the access: read, write, execute, etc. The translation from linear address to physical address is controlled by the virtual memory system, the topic of a later chapter.

Cache Memory

Here is another topic that we continue to mention in passing with a promise to discuss it more fully at a later time. For the moment, we shall describe the advantages of such a system, and again postpone a full discussion for another chapter.

Each Pentium product is packaged with a cache memory system designed to optimize memory access in a system that is referencing both data memory and instruction memory at the same time. We should note that it is the general practice to keep both data and executable instructions in the same main memory, and differentiate the two only in the cache. This is one example of the common use of cache: cause the memory system to act as if it has a certain desirable attribute without having to alter the large main memory to actually have that attribute.



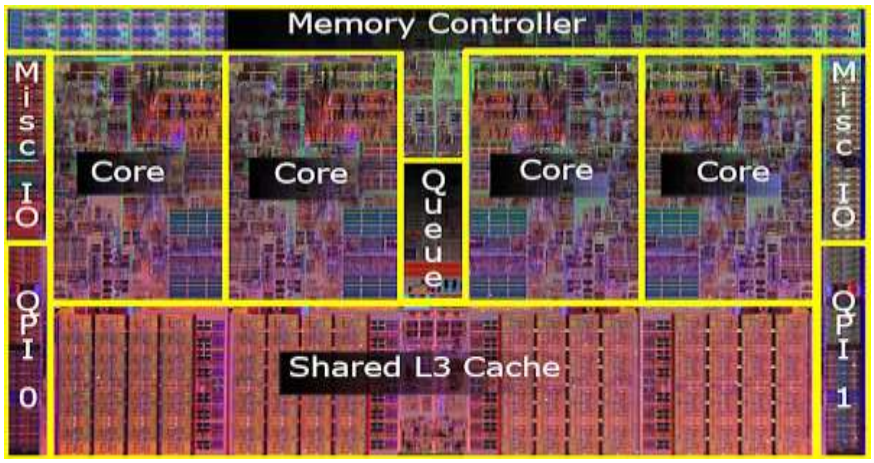
At this time, let's state a few facts. Because it is smaller, the Level 1 cache (L1 cache) is faster than the L2 cache. Because it is smaller than main memory, the L2 cache is faster than the main memory. This multilevel cache applies the same trick twice. In the above example, the 32 KB L1 cache combined with the 1 MB L2 cache acts as if it were a single cache memory with an access time only slightly slower than the actual L1 cache. Then the combination of cache memory and the main memory acts as if it were a single large memory (2 GB) with an access time only slightly slower than the cache memory. Now we have a memory that functionally is both large and fast, while no single element actually has both attributes.

Recent main memory designs have added a write buffer, allowing for short bursts of memory writes at a rate much higher than the main memory can sustain. Suppose that the main memory has a cycle time of 80 nanoseconds, requiring a 80 nanosecond time interval between two

independent writes to memory. A fast write buffer might be able to accept eight memory writes in that time span, sending each to main memory at a slower rate.

We mention in passing that some multi–core Pentium designs have three levels of cache memory. Here is a picture of the Intel Core i7 die. This CPU has four cores, each with its L1 and L2 caches. In addition, there is a Level 3 cache that is shared by the four cores. This design illustrates two realities of CPU design in regards to cache memory.

1. The placement of cache memory on the CPU chip significantly increases execution speed, as on –chip accesses are faster than accesses to another chip.
2. Better power management, due to the fact that memory uses less power per unit area than does the CPU logic.



Register Sets

Almost all modern computers divide storage devices into three classes: registers, memory, and external storage (such as disks and magnetic tape). In earlier times, the register set (also called the register file) was distinctly associated with the CPU, while main memory was obviously separate from the CPU. Now that designs have on–chip cache memory, the distinction between register memory and other memory is purely logical. We shall see that difference when we study a few fragments of IA–32 assembly language.

One of the first steps in designing a CPU is the determination of the number and naming of the registers to be associated with the CPU. There are many general approaches, and then there is the approach seen on the Pentium. The design used in all IA–32 and some IA–64 designs is a reflection of the original Intel 8080 register set.

Register set of the Intel 8080 and 8086

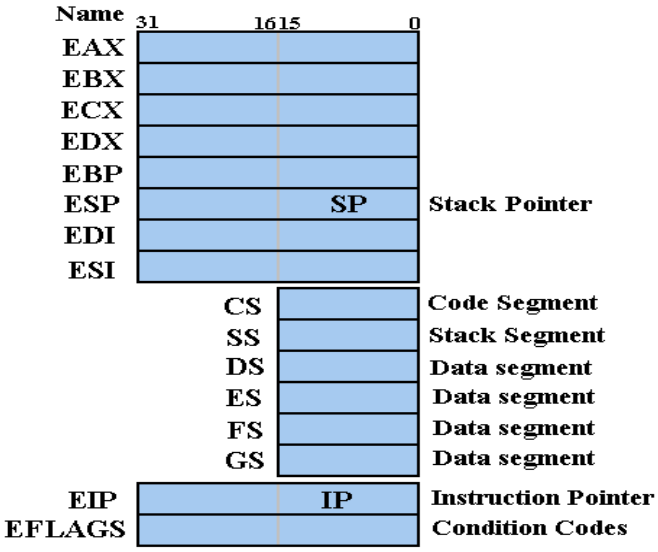
The original Intel 8080 and Intel 8086 designs date from a time when single accumulator machines were still common. As mentioned in a previous chapter, it is quite possible to design a CPU with only one general–purpose register; this is called the **accumulator**. The provision of seven general–purpose registers in the Intel 8080 design was a step up from existing practice.

We have already discussed the evolution of the register set design in the evolution of the IA–32 line. The Intel 8080 had 8–bit registers; the Intel 8086, 80186, and 80286 each has 16–bit registers, and the IA–32 line (beginning with the Intel 80386) all have 32–bit registers. The

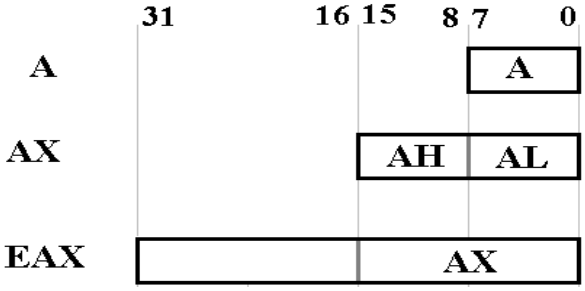
Intel 8080 set the trend; newer models might have additional registers, but each one had to have the original register set in some fashion.

Register set of the Intel 80386

The Intel 80386 was the first member of the IA–32 design line. It is a convenient example for purposes of discussion. In fact, it is common practice for introductory courses in Pentium assembly language to focus almost exclusively on the Intel 80386 Instruction Set Architecture (register set and assembly language instructions), and to treat the full Pentium ISA as an extension. Here is a figure showing the Intel 80386 register set.



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.

Here are some examples of IA–32 assembly language involving the EAX register. Note that the assembly language syntax denotes hexadecimal numbers by appending an "H".

```
MOV  EAX, 1234H  ; Set the 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 ZZ
```

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

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.
             ; Store this in memory location XX.
```

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.
ADD EAX, EBX  ; Add the 32-bit value in EBX to that in EAX.
```

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
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, but it can also support input and output data transfers. It also plays a special part in executing integer multiplication and division. In general, the product of two 8–bit integers is a 16–bit integer, the product of two 16–bit integers is a 32–bit integer, and the product of two 32–bit integers is a 64–bit integer. Remember that register AL is the 8 low–order bits of EAX, and AX is the 16 low–order bits.

One item that is important to note is that the EAX register, or whatever part is used in the MUL operation, is implicitly a part of the operation, without being called out explicitly.

```
    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 instruction says multiply the value in
```

```
                    ; AL by that in BL and put the product in AX.
                    ; Only BL is explicitly mentioned.
```

The 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. Rather the two 16–bit registers AX and DX are viewed as forming a 32–bit pair and serve to store it. Again, note that the 16–bit version of the MUL automatically takes AX as holding one of the integers to be multiplied.

```
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 product. Here is an example.

```
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
IN  AL, DX      ; Get a byte from the port at address 200H.
```

The **ESI** and **EDI** registers are used as source and destination addresses for string and array operations. These are sometimes called "**Extended Source Index**" and "**Extended Destination Index**". They facilitate high–speed memory transfers.

The **EBP** register is used to support the call stack for high level language procedure calls. We shall discuss this more in the next chapter, in which we discuss subroutines. Briefly put, it functions much like a stack pointer, but does not point to the top of the stack.

The next two registers, **EIP** and **ESP**, are 32–bit versions of the older 16–bit counterparts. We discuss these here, and then introduce the 16–bit variants by discussing segments again.

The **EIP** is the 32–bit **Instruction Pointer**, so called because it points to the instruction likely to be executed next. Many other architectures call this register by the more traditional, if less appropriate, name "**Program Counter**". Jump and branch instructions, unconditional or conditional (if the condition is true), achieve their affect by forcing a target address into the EIP.

The **ESP** is the 32–bit **Stack Pointer**, used to hold the address of the top of the stack. This register is not commonly accessed directly except as a part of a procedure call. We must make the point here that the stack is not always treated as an **ADT** (Abstract Data Type) with PUSH as the only way to place an item on the stack. We shall investigate direct manipulation of the ESP in more detail when we discuss allocation of dynamic memory for local variables.

The **EFLAGS** register holds a collection of at most 32 Boolean flags with various meanings. The flags are divided into two broad categories: **control flags** and **status flags**. Control flags can cause the CPU to break after every instruction (good for debugging), interrupt execution on detecting arithmetic overflow, enter protected mode, or enter virtual 8086 mode.

The status flags reflect the state of the execution and include CF (the carry flag, indicating a carry out of the last arithmetic operation), OF (the overflow flag, indicating that the result is

too large or too small to be represented), SF (the sign flag, indicating that the last result was negative), ZF (the zero flag, indicating that the last result was zero), and several more.

There are six 16–bit segment registers (CS, SS, DS, ES, FS, and GS), which are hold overs from the 16–bit Intel 8086. As discussed in the previous chapter, these are used to allow generation of 20–bit addresses from 16–bit registers. The two standard register pairings are CS:IP (Code Segment and Instruction Pointer) and SS:SP (Stack Segment and Stack Pointer). In the more modern Pentium usage, these segment registers are used in combination with descriptor registers to support memory management.

Register set of the Pentium

In addition to the above register set, the Pentium architecture calls for six 64–bit registers to support memory management (CSDCR, SSDCR, DSDCR, ESDCR, FSDCR, and GSDCR), the TR (Task Register), the IDTR (Interrupt Descriptor Table Register), two descriptor registers (GDTR – Global Descriptor Task Register and LDTR – Local Descriptor Task Register) and a few more. Then there are the sixteen specialized data registers (MM0 – MM7 for the multimedia instructions, and FP0 – FP7 for floating point arithmetic). Newer versions of the architecture almost certainly contain still more registers.

Especially in the case of memory management, it is important to remember that the Operating System functions by setting up and then using some fairly elaborate data structures. Each of these structures has a base address stored in one of these registers for fast access.

Addressing Modes

We now discuss some of the addressing modes used in the Pentium architecture. We shall use two–argument instructions to illustrate this, as that is easier. The simplest mode is also the fastest to execute. This is the **data register direct mode**. Here is an example.

```
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 is 2 bytes.
```

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.
                 ; Not the same as the above example.

MOV EDX, WORD1   ; Move the contents of address WORD1 to EDX

MOV WORD2, EDX   ; Move the contents of the 32-bit register
                 ; EDX to memory location WORD2.
```

```
MOV X, Y            ; NOT VALID.  Memory to memory moves are
                    ; not allowed in this architecture.
```

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 the 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
```

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.
```

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 above.
```