# CHAPTER 2
# INTRODUCTION TO THE IA-32 INTEL ARCHITECTURE

The exponential growth of computing power and personal computer ownership made the computer one of the most important forces that shaped business and society in the second half of the twentieth century. Furthermore, computers are expected to continue to play crucial roles in the growth of technology, business, and other new arenas in the future, because new applications (such as, the Internet, digital media, and genetics research) are strongly dependent on ever increasing computing power for their growth.

The IA-32 Intel Architecture has been at the forefront of the computer revolution and is today clearly the preferred computer architecture, as measured by the number of computers in use and total computing power available in the world. Two of the major factors that may be the cause of the popularity of IA-32 architecture are: compatibility of software written to run on IA-32 processors, and the fact that each generation of IA-32 processors deliver significantly higher performance than the previous generation. As such, this chapter provides a brief historical summary of the IA-32 architecture, from its origin in the Intel 8086 processor to the latest version implemented in the Pentium 4 processor.

## 2.1. BRIEF HISTORY OF THE IA-32 ARCHITECTURE

The developments leading to the latest version of the IA-32 architecture can be traced back to the Intel 8085 and 8080 microprocessors and to the Intel 4004 microprocessor (the first microprocessor, designed by Intel in 1969). Before the IA-32 architecture family introduced 32-bit processors, it was preceded by 16-bit processors including the 8086 processor, and quickly followed by a more cost-effective version, the 8088. From a historic perspective, the IA-32 architecture contains both 16-bit processors and 32-bit processors. At present, the 32-bit IA-32 architecture is a very popular computer architecture for many operating systems and a very wide range of applications.

One of the most important achievements of the IA-32 architecture is that the object code programs created for these processors starting in 1978 still execute on the latest processor in the IA-32 architecture family.

The 8086 has 16-bit registers and a 16-bit external data bus, with 20-bit addressing giving a 1-MByte address space. The 8088 is identical except for a smaller external data bus of 8 bits. These processors introduced segmentation to the IA-32 architecture. With segmentation, a 16-bit segment register contains a pointer to a memory segment of up to 64 KBytes in size. Using four segment registers at a time, the 8086/8088 processors are able to address up to 256 KBytes without switching between segments. The 20-bit addresses that can be formed using a segment register pointer and an additional 16-bit pointer provide a total address range of 1 MByte.

The Intel 286 processor introduced the protected mode operation into the IA-32 architecture. This new mode of operation uses the segment register contents as selectors or pointers into descriptor tables. The descriptors provide 24-bit base addresses, allowing a maximum physical memory size of up to 16 MBytes, support for virtual memory management on a segment swapping basis, and various protection mechanisms. These protection mechanisms include segment limit checking, read-only and execute-only segment options, and up to four privilege levels to protect operating system code (in several subdivisions, if desired) from application or user programs. In addition, hardware task switching and local descriptor tables allow the operating system to protect application or user programs from each other.

The Intel386 processor was the first 32-bit processor in the IA-32 architecture family. It introduced 32-bit registers into the architecture, for use both to hold operands and for addressing. The lower half of each 32-bit register retained the properties of the 16-bit registers of the two earlier generations, to provide complete backward compatibility. A new virtual-8086 mode was provided to yield greater efficiency when executing programs created for the 8086 and 8088 processors on the new 32-bit processors. The Intel386 processor has a 32-bit address bus, and can support up to 4 GBytes of physical memory. The 32-bit architecture provides logical address space for each software process. The 32-bit architecture supports both a segmented-memory model and a "flat"[1] memory model. In the "flat" memory model, the segment registers point to the same address, and all 4 GBytes addressable space within each segment are accessible to the software programmer. The original 16-bit instructions were enhanced with new 32-bit operand and addressing forms, and completely new instructions were provided, including those for bit manipulation. The Intel386 processor also introduced paging into the IA-32 architecture, with the fixed 4-KByte page size providing a method for virtual memory management that was significantly superior compared to using segments for the purpose. This paging system was much more efficient for operating systems, and completely transparent to the applications, without significant sacrifice in execution speed. The ability to support 4 GBytes of virtual address space, memory protection, together with paging support, enabled the IA-32 architecture to be a popular choice for advanced operating systems and wide variety of applications.

The IA-32 architecture has been and is committed to the task of maintaining backward compatibility at the object code level to preserve Intel customers' large investment in software. At the same time, in each generation of the architecture, the latest most effective micro-architecture and silicon fabrication technologies have been used to produce high-performance processors. In each generation of IA-32 processors, Intel has conceived and incorporated increasingly sophisticated techniques into its microarchitecture in pursuit of ever faster computers. Various forms of parallel processing have been the most performance enhancing of these techniques, and the Intel386 processor was the first IA-32 architecture processor to include a number of parallel stages. These six stages are the Bus Interface Unit (accesses memory and I/O for the other units), the Code Prefetch Unit (receives object code from the Bus Unit and puts it into a 16-byte queue), the Instruction Decode Unit (decodes object code from the Prefetch unit into microcode), the Execution Unit (executes the microcode instructions), the Segment Unit (translates logical addresses to linear addresses and does protection checks), and the Paging Unit (translates linear addresses to physical addresses, does page based protection checks, and contains a cache with information for up to 32 most recently accessed pages).

[1] Requires only one 32-bit address component to access anywhere in the linear address space.

The Intel486 processor added more parallel execution capability by expanding the Intel386 processor's Instruction Decode and Execution Units into five pipelined stages, where each stage (when needed) operates in parallel with the others on up to five instructions in different stages of execution. Each stage can do its work on one instruction in one clock, and so the Intel486 processor can execute as rapidly as one instruction per clock cycle. An 8-KByte on-chip first-level cache was added to the Intel486 processor to greatly increase the percent of instructions that could execute at the scalar rate of one per clock: memory access instructions were now included if the operand was in the first-level cache. The Intel486 processor also for the first time integrated the x87 FPU onto the processor and added new pins, bits and instructions to support more complex and powerful systems (second-level cache support and multiprocessor support).

Late in the Intel486 processor generation, Intel incorporated features designed to support power savings and other system management capabilities into the IA-32 architecture mainstream with the Intel486 SL Enhanced processors. These features were developed in the Intel386 SL and Intel486 SL processors, which were specialized for the rapidly growing battery-operated notebook PC market segment. The features include the new System Management Mode, triggered by its own dedicated interrupt pin, which allows complex system management features (such as power management of various subsystems within the PC), to be added to a system transparently to the main operating system and all applications. The Stop Clock and Auto Halt Powerdown features allow the processor itself to execute at a reduced clock rate to save power, or to be shut down (with state preserved) to save even more power.

The Intel Pentium processor added a second execution pipeline to achieve superscalar performance (two pipelines, known as u and v, together can execute two instructions per clock). The on-chip first-level cache was also doubled, with 8 KBytes devoted to code, and another 8 KBytes devoted to data. The data cache uses the MESI protocol to support the more efficient write-back mode, as well as the write-through mode that is used by the Intel486 processor. Branch prediction with an on-chip branch table was added to increase performance in looping constructs. Extensions were added to make the virtual-8086 mode more efficient, and to allow for 4-MByte as well as 4-KByte pages. The main registers are still 32 bits, but internal data paths of 128 and 256 bits were added to speed internal data transfers, and the burstable external data bus has been increased to 64 bits. The Advanced Programmable Interrupt Controller (APIC) was added to support systems with multiple Pentium processors, and new pins and a special mode (dual processing) was designed in to support glueless two processor systems.

The last processor in the Pentium family (the Pentium Processor with MMX™ Technology) introduced the Intel MMX technology to the IA-32 architecture. The Intel MMX technology uses the single-instruction, multiple-data (SIMD) execution model to perform parallel computations on packed integer data contained in the 64-bit MMX registers. This technology greatly enhanced the performance of the IA-32 processors in advanced media, image processing, and data compression applications.

In 1995, Intel introduced the P6 family of processors. This processor family was based on a new superscalar micro-architecture that established new performance standards. One of the primary goals in the design of the P6 family micro-architecture was to exceed the performance of the Pentium processor significantly while still using the same 0.6-micrometer, four-layer, metal BICMOS manufacturing process. Using the same manufacturing process as the Pentium processor meant that performance gains could only be achieved through substantial advances in the micro-architecture.

The Intel Pentium Pro processor was the first processor based on the P6 micro-architecture. Subsequent members of the P6 processor family are: the Intel Pentium II, Intel Pentium® II Xeon™, Intel Celeron™, Intel Pentium III, and Intel Pentium® III Xeon™ processors. A brief description of each of these processor members follows.

The Pentium Pro processor is three-way superscalar, permitting it to execute up to three instructions per clock cycle. It also introduced the concept of dynamic execution (micro-data flow analysis, out-of-order execution, superior branch prediction, and speculative execution) in a superscalar implementation. Three instruction decode units worked in parallel to decode object code into smaller operations called micro-ops (micro-architecture op-codes). These micro-ops are fed into an instruction pool, and (when interdependencies permit) can be executed out of order by the five parallel execution units (two integer, two FPU and one memory interface unit). The Retirement Unit retires completed micro-ops in their original program order, taking account of any branches. The power of the Pentium Pro processor was further enhanced by its caches: it had the same two on-chip 8-KByte 1st-Level caches as did the Pentium processor, and also had a 256-KByte 2nd-Level cache that was in the same package as, and closely coupled to, the processor, using a dedicated 64-bit backside (cache-bus) full clock speed bus. The 1st-Level cache was dual-ported, the 2nd-Level cache supported up to 4 concurrent accesses, and the 64-bit external data bus was transaction-oriented, meaning that each access was handled as a separate request and response, with numerous requests allowed while awaiting a response. These parallel features for data access enhanced the performance of the processor by providing a non-blocking architecture in which the processor's parallel execution units can be better utilized. The Pentium Pro processor also has an expanded 36-bit address bus, giving a maximum physical address space of 64 GBytes.

The Intel Pentium II processor added the Intel MMX technology to the P6 family processors along with new packaging and several hardware enhancements. The processor core is packaged in the Single Edge Contact cartridge (SECC), enabling ease of design and flexible motherboard architecture. The first-level data and instruction caches are enlarged to 16 KBytes each, and second-level cache sizes of 256 KBytes, 512 KBytes, and 1 MByte are supported. A "half clock speed" backside bus that connects the second-level cache to the processor. Multiple low-power states such as AutoHALT, Stop-Grant, Sleep, and Deep Sleep are supported to conserve power when idling.

The Pentium II Xeon processor combined several premium characteristics of previous generation of Intel processors such as 4-way, 8-way (and up) scalability a 2-MByte second-level cache running on a "full-clock speed" backside bus to meet the demands of mid-range and higher performance servers and workstations.

The Intel Celeron processor family focused the IA-32 architecture on the desktop or value PC market segment. It offers features such as an integrated 128 KByte of second-level cache, a plastic pin grid array (P.P.G.A.) form factor to lower system design cost.

The Pentium III processor introduced the Streaming SIMD Extensions (SSE) into the IA-32 architecture. The SSE extensions extend the SIMD execution model introduced with the Intel MMX technology with a new set of 128-bit registers and the ability to perform SIMD operations on packed single-precision floating-point values.

The Pentium III Xeon processor extended the performance levels of the IA-32 processors with the enhancement of a full-speed, on-die, Advanced Transfer Cache using Intel' s 0.18 micron process technology.

## 2.2. THE INTEL PENTIUM 4 PROCESSOR

The Intel Pentium 4 processor is the latest generation of IA-32 processor that is based on the Intel NetBurst$^{TM}$ micro-architecture. The Intel NetBurst micro-architecture is a new 32-bit micro-architecture that allows processors to operate at significantly higher clock speeds and performance levels than previous IA-32 processors. The Intel Pentium 4 processor family has the following advanced features:

- Rapid Execution Engine:
  - — Arithmetic Logic Units (ALUs) run at twice the processor frequency.
  - — Basic integer operations executes in 1/2 processor clock tick.
  - — Higher throughput and reduced latency of execution.
- Hyper Pipelined Technology:
  - — Twenty-stage pipeline to enable breakthrough clock rates for desktop PCs and servers.
  - — Frequency headroom and performance scalability to continue leadership into the future.
- Advanced Dynamic Execution
  - — Very deep, out-of-order, speculative execution engine.
    - Up to 126 instructions in flight (3 times larger than the Pentium III processor).
    - Up to 48 loads and 24 stores in pipeline (2 times larger than the Pentium III processor).
  - — Enhanced branch prediction capability
    - Reduces the mis-prediction penalty associated with deeper pipelines.
    - Advanced branch prediction algorithm.
    - 4K entry branch target array (8 times larger than the Pentium III processor).
- Innovative new cache subsystem:
  - — First level caches.
    - 12 K micro-op Execution Trace Cache.
    - Execution Trace Cache that removes decoder latency from main execution loops.
    - Execution Trace Cache integrates path of program execution flow into a single line.
    - Low latency 8-KByte data cache with 2 cycle latency.
  - — Second level caches.
    - Full-speed, unified 8-way 2nd-level on-die Advance Transfer Cache.
    - Delivers ~45 GB/s data throughput (at 1.4GHz processor frequency).
    - Bandwidth and performance increases with processor frequency.

- Streaming SIMD Extensions 2 (SSE2) Technology:
    - — SSE2 Extends MMX and SSE technology with the addition of 144 new instructions, which include support for:
        - 128-bit SIMD integer arithmetic operations.
        - 128-bit SIMD double precision floating point operations.
        - Cache and memory management operations.
    - — Further enhances and accelerates video, speech, encryption, image and photo processing.
- 400 MHz Intel NetBurst micro-architecture system bus.
    - — Provides 3.2 GBytes per second throughput (3 times faster than the Pentium III processor).
    - — Quad-pumped 100MHz scalable bus clock to achieve 400 MHz effective speed.
    - — Split-transaction, deeply pipelined.
    - — 128-byte lines with 64-byte accesses.
- Compatible with existing IA-32 applications and operating systems.

### 2.2.1. Streaming SIMD Extensions 2 (SSE2) Technology

The Intel Pentium 4 processor introduces the SSE2 extensions, which offer several enhancements to the Intel MMX technology and SSE extensions. These enhancements include operations on new packed data formats and increased SIMD computational performance using 128-bit wide registers for integer SIMD operation. A packed double-precision floating-point data type is introduced along with several packed 128-bit integer data types. These new data types allow packed double-precision and single-precision floating-point and packed integer computations to be performed in the XMM registers.

New SIMD instructions introduced in the IA-32 architecture include floating-point SIMD instructions, integer SIMD instructions, conversion between SIMD floating-point data and SIMD integer data, and conversion of packed data between XMM registers and MMX registers. New floating-point SIMD instructions allow computations to be performed on packed double-precision floating-point values (two double-precision values per XMM register). The computation of SIMD floating-point instructions and the single-precision and double-precision floating-point formats are compatible with IEEE Standard 754 for Binary Floating-Point Arithmetic. New integer SIMD instructions provide flexible and higher dynamic range computational power by supporting arithmetic operations on packed doubleword and quadword data as well as other operations on packed byte, word, doubleword, quadword and double quadword data.

In addition to new 128-bit SIMD instructions described in the previous paragraph, there are 128-bit enhancement to 68 integer SIMD instructions, which operated solely on 64-bit MMX registers in the Pentium II and Pentium III processors. Those 64-bit integer SIMD instructions are enhanced to support operation on 128-bit XMM registers in the Pentium 4 processor. These enhanced integer SIMD instructions allow software developers to deliver new performance levels when implementing floating-point and integer algorithms, and to have maximum flexibility by writing SIMD code with either XMM registers or MMX registers.

The Intel Pentium 4 processor offers new features that enable software developers to deliver new levels of performance in multimedia applications ranging from 3-D graphics, video decoding/encoding to speech recognition. The new packed double-precision floating-point instructions enhance performance for applications that require greater range and precision, including scientific and engineering applications and advanced 3-D geometry techniques, such as ray tracing.

To speed up processing and improve cache usage, the SSE2 extensions offers several new instructions that allow application programmers to control the cacheability of data. These instructions provide the ability to stream data in and out of the registers without disrupting the caches and the ability to prefetch data before it is actually used.

The new architectural features introduced with the SSE2 extensions do not require new operating system support. This is because the SSE2 extensions do not introduce new architectural states, and the FXSAVE/FXRSTOR instructions, which supports the SSE extensions, also supports SSE2 extensions and are sufficient for saving and restoring the state of the XMM registers, the MMX registers, and the x87 FPU registers during a context switch. The CPUID instruction has been enhanced to allow operating system or applications to identify for the existence of the SSE and SSE2 features.

The SSE2 extensions are accessible in all IA-32 architecture operating modes in the Intel Pentium 4 processor. The Pentium 4 processor maintains IA-32 software compatibility. All existing software continues to run correctly, without modification on the Pentium 4 processor and future IA-32 processors that incorporate the SSE2 extensions. Also, existing software continues to run correctly in the presence of applications that make use of the SSE2 instructions.

## 2.3. MOORE’S LAW AND IA-32 PROCESSOR GENERATIONS

In the mid-1960s, Intel Chairman of the Board Gordon Moore made an observation: “the number of transistors that would be incorporated on a silicon die would double every 18 months for the next several years”. Over the past three and half decades, this prediction has continued to hold true that it is often referred to as “Moore's Law .”

The computing power and the complexity (or roughly, the number of transistors per processor) of Intel architecture processors has grown, over the years, in close relation to Moore's law. By taking advantage of new process technology and new micro-architecture designs, each new generations of IA-32 processors have demonstrated frequency-scaling headroom and new performance levels over the previous generation processors. The key features of the Intel Pentium 4 processor and Pentium III processor with Advanced Transfer Cache are shown in Table 2-1. Older generation of IA-32 processors, which do not employ on-die second-level cache, are shown in Table 2-2.

Table 2-1. Key Features of contemporary IA-32 processors

| Intel Processor | Date Intro-duced | Micro-architecture | Clock Frequency at Intro-duction | Transis-tors per Die | Register Sizes[1] | System Bus Bandwidth | Max. Extern. Addr. Space | On-die Caches [2] |
|---|---|---|---|---|---|---|---|---|
| Pentium III processor[3] | 1999 | P6 | 700 MHz | 28 M | GP: 32<br>FPU: 80<br>MMX: 64<br>XMM: 128 | Up to 1.06 GB/s | 64 GB | 32KB L1;<br>256KB L2 |
| Pentium 4 processor | 2000 | Intel NetBurst™ micro-architecture | 1.50 GHz | 42 M | GP: 32<br>FPU: 80<br>MMX: 64<br>XMM: 128 | 3.2 GB/s | 64 GB | 12K µop Execution Trace Cache;<br>8KB L1;<br>256KB L2 |

NOTES:

1. The register size and external data bus size are given in bits.
2. First level cache is denoted using the abbreviation L1, 2nd level cache is denoted as L2.
3. Intel Pentium III and Pentium III Xeon processors, with Advanced Transfer Cache and built on 0.18 micron process technology, were introduced in October 1999.

Table 2-2. Key Features of previous generations of IA-32 Processor

| Intel Processor | Date Intro-duced | Max. Clock Frequency at Intro-duction | Transis-tors per Die | Register Sizes[1] | Ext. Data Bus Size[2] | Max. Extern. Addr. Space | Caches[2] |
|---|---|---|---|---|---|---|---|
| 8086 | 1978 | 8 MHz | 29 K | 16 GP | 16 | 1 MB | None |
| Intel 286 | 1982 | 12.5 MHz | 134 K | 16 GP | 16 | 16 MB | Note 3 |
| Intel386 DX Processor | 1985 | 20 MHz | 275 K | 32 GP | 32 | 4 GB | Note 3 |
| Intel486 DX Processor | 1989 | 25 MHz | 1.2 M | 32 GP<br>80 FPU | 32 | 4 GB | 8KB L1 |
| Pentium Processor | 1993 | 60 MHz | 3.1 M | 32 GP<br>80 FPU | 64 | 4 GB | 16KB L1 |
| Pentium Pro Processor | 1995 | 200 MHz | 5.5 M | 32 GP<br>80 FPU | 64 | 64 GB | 16KB L1;<br>256KB or<br>512KB L2 |
| Pentium II Processor | 1997 | 266 MHz | 7 M | 32 GP<br>80 FPU<br>64 MMX | 64 | 64 GB | 32KB L1;<br>256KB or<br>512KB L2 |
| Pentium III Processor[3] | 1999 | 500 MHz | 8.2 M | 32 GP<br>80 FPU<br>64 MMX<br>128 XMM | 64 | 64 GB | 32KB L1;<br>512KB L2 |

NOTES:

1. The register size and external data bus size are given in bits. Note also that each 32-bit general-purpose (GP) registers can be addressed as an 8- or a 16-bit data registers in all of the processors, and there are internal data paths that are 2 to 4 times wider than the external data bus for each processor.

2. In addition to the general-purpose caches listed in the table for the Intel486 processor (8 KBytes of combined code and data) and the Intel Pentium and Pentium Pro processors (8 KBytes each for separate code cache and data cache), there are smaller special purpose caches. The Intel 286 processor has 6 byte descriptor caches for each segment register. The Intel386 processor has 8 byte descriptor caches for each segment register, and also a 32-entry, 4-way set associative Translation Lookaside Buffer (cache) to store access information for recently used pages on the chip. The Intel486 processor has the same caches described for the Intel386 processor, as well as its 8K L1 general-purpose cache. The Intel Pentium and Pentium Pro processors have their general purpose caches, descriptor caches, and two Translation Lookaside Buffers each (one for each 8K L1 cache). The Pentium II and Pentium III processors have the same cache structure as the Pentium Pro processor except that the size of each cache is 16 KBytes. The 2nd level caches in Pentium Pro, Pentium II and Pentium III processors are off-die, but inside the processor package.

3. Intel Pentium III processor was introduced in February1999, it included an off-die, 512 KB L2 cache.

## 2.4. THE P6 FAMILY MICRO-ARCHITECTURE

The Pentium Pro processor introduced a new micro-architecture for the Intel IA-32 processors, commonly referred to as P6 processor microarchitecture. The P6 processor micro-architecture was later enhanced with an on-die, 2nd level cache, called Advanced Transfer Cache. This micro-architecture is a three-way superscalar, pipelined architecture. The term "three-way superscalar" means that using parallel processing techniques, the processor is able on average to decode, dispatch, and complete execution of (retire) three instructions per clock cycle. To handle this level of instruction throughput, the P6 processor family use a decoupled, 12-stage superpipeline that supports out-of-order instruction execution. Figure 2-1 shows a conceptual view of the P6 processor micro-architecture pipeline with the Advanced Transfer Cache enhancement. The micro-architecture pipeline is divided into four sections (the 1st level and 2nd level caches, the front end, the out-of-order execution core, and the retire section). Instructions and data are supplied to these units through the bus interface unit.

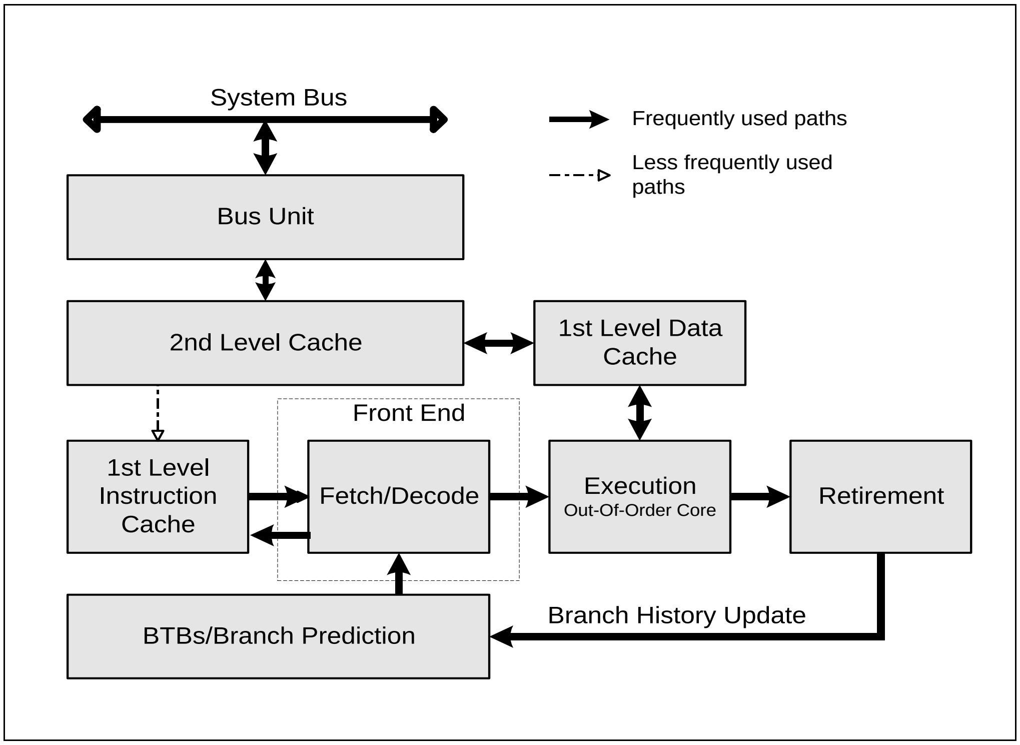


Figure 2-1. The P6 Processor Micro-Architecture with Advanced Transfer Cache enhancement

To insure a steady supply of instructions and data to the instruction execution pipeline, the P6 processor micro-architecture incorporates two cache levels. The first-level cache provides an 8-

KByte instruction cache and an 8-KByte data cache, both closely coupled to the pipeline. The second-level cache is a 256-KByte, 512-KByte, or 1-MByte static RAM that is coupled to the core processor through a full clock-speed 64-bit cache bus.

The centerpiece of the P6 processor micro-architecture is an innovative out-of-order execution mechanism called "dynamic execution." Dynamic execution incorporates three data-processing concepts:

- Deep branch prediction.
- Dynamic data flow analysis.
- Speculative execution.

Branch prediction is a modern technique to deliver high performance in pipelined micro-architectures. It allows the processor to decode instructions beyond branches to keep the instruction pipeline full. The P6 processor family implements highly optimized branch prediction algorithm to predict the direction of the instruction stream through multiple levels of branches, procedure calls, and returns.

Dynamic data flow analysis involves real-time analysis of the flow of data through the processor to determine data and register dependencies and to detect opportunities for out-of-order instruction execution. The out-of-order execution core can simultaneously monitor many instructions and execute these instructions in the order that optimizes the use of the processor's multiple execution units, while maintaining the data integrity. This out-of-order execution keeps the execution units busy even when cache misses and data dependencies among instructions occur.

Speculative execution refers to the processor's ability to execute instructions that lie beyond a conditional branch that has not yet been resolved, and ultimately to commit the results in the order of the original instruction stream. To make speculative execution possible, the P6 processor micro-architecture decouples the dispatch and execution of instructions from the commitment of results. The processor's out-of-order execution core uses data-flow analysis to execute all available instructions in the instruction pool and store the results in temporary registers. The retirement unit then linearly searches the instruction pool for completed instructions that no longer have data dependencies with other instructions or unresolved branch predictions. When completed instructions are found, the retirement unit commits the results of these instructions to memory and/or the IA-32 registers (the processor's eight general-purpose registers and eight x87 FPU data registers) in the order they were originally issued and retires the instructions from the instruction pool.

Combining branch prediction, dynamic data-flow analysis and speculative execution, the dynamic execution capability of the P6 micro-architecture removes the constraint of linear instruction sequencing between the traditional fetch and execute phases of instruction execution. Thus, the processor can continue to decode instructions even when there are multiple levels of branches. Branch prediction and advanced decoder implementation work together to keep the instruction pipeline full. Subsequently, the out-of-order, speculative execution engine can take advantage of the processor' s six execution units to execute instructions in parallel. And finally, it commits the results of executed instructions in original program order to maintain data integrity and program coherency.

## 2.5. THE INTEL NETBURST MICRO-ARCHITECTURE

The Intel Pentium 4 processor is the newest member of the 32-bit Intel architecture family. It is the first implementation of the Intel NetBurst micro-architecture and provides the following significant features:

- Rapid Execution Engine:
  - — Arithmetic Logic Units (ALUs) run at twice the processor frequency.
  - — Basic integer operations executes in 1/2 processor clock tick.
  - — Provides higher throughput and reduced latency of execution.
- Hyper Pipelined Technology:
  - — Twenty-stage pipeline to enable industry-leading clock rates for desktop PCs and servers.
  - — Provides frequency headroom and scalability to continue leadership into the future.
- Advanced Dynamic Execution:
  - — Very deep, out-of-order, speculative execution engine.
    - Up to 126 instructions in flight.
    - Up to 48 loads and 24 stores in pipeline.
  - — Enhanced branch prediction capability.
    - Reduces the mis-prediction penalty associated with deeper pipelines.
    - Advanced branch prediction algorithm.
    - 4K-entry branch target array.
- New cache subsystem:
  - — First level caches.
    - Advanced Execution Trace Cache stores decoded instructions.
    - Execution Trace Cache removes decoder latency from main execution loops.
    - Execution Trace Cache integrates path of program execution flow into a single line.
    - Low latency data cache with 2 cycle latency.
  - — second level cache.
    - Full-speed, unified 8-way 2nd-Level on-die Advance Transfer Cache.
    - Bandwidth and performance increases with processor frequency.
- High-performance, quad-pumped bus interface to the Intel NetBurst micro-architecture system bus.

  - — Support quad-pumped, scalable bus clock to achieve 4X effective speed.
  - — Capable of delivering up to 3.2 GB of bandwidth per second for Pentium 4 processor.
- Superscalar issue to enable parallelism.
- Expanded hardware registers with renaming to avoid register name space limitations.
- 128-byte cache line size.
  - — Two 64-byte sectors.

Figure 2-2 gives an overview of the Intel NetBurst micro-architecture. This micro-architecture pipeline is made up of three sections: an in-order issue front end, an out-of-order superscalar execution core, and an in-order retirement unit. The following sections provide an overview of each of these pipeline sections.

### 2.5.1. The Front End Pipeline

The front end supplies instructions in program order to the out-of-order core which has very high execution bandwidth and can execute basic integer operations with 1/2 clock cycle latency. The front end fetches and decodes IA-32 instructions, and breaks them down into simple operations called micro-ops (µops). It can issue multiple µops per cycle, in original program order, to the out-of-order core.

The front end performs several basic functions:

- Prefetch IA-32 instructions that are likely to be executed.
- Fetch instructions that have not already been prefetched.
- Decode IA-32 instructions into micro-operations.
- Generate microcode for complex instructions and special-purpose code.
- Deliver decoded instructions from the execution trace cache.
- Predict branches using highly advanced algorithm.

The front end of the Intel NetBurst micro-architecture is designed to address some of the common problems in high-speed, pipelined microprocessors. Two of these problems contribute to major sources of delays:

- the time to decode instructions fetched from the target
- wasted decode bandwidth due to branches or branch target in the middle of cache lines.

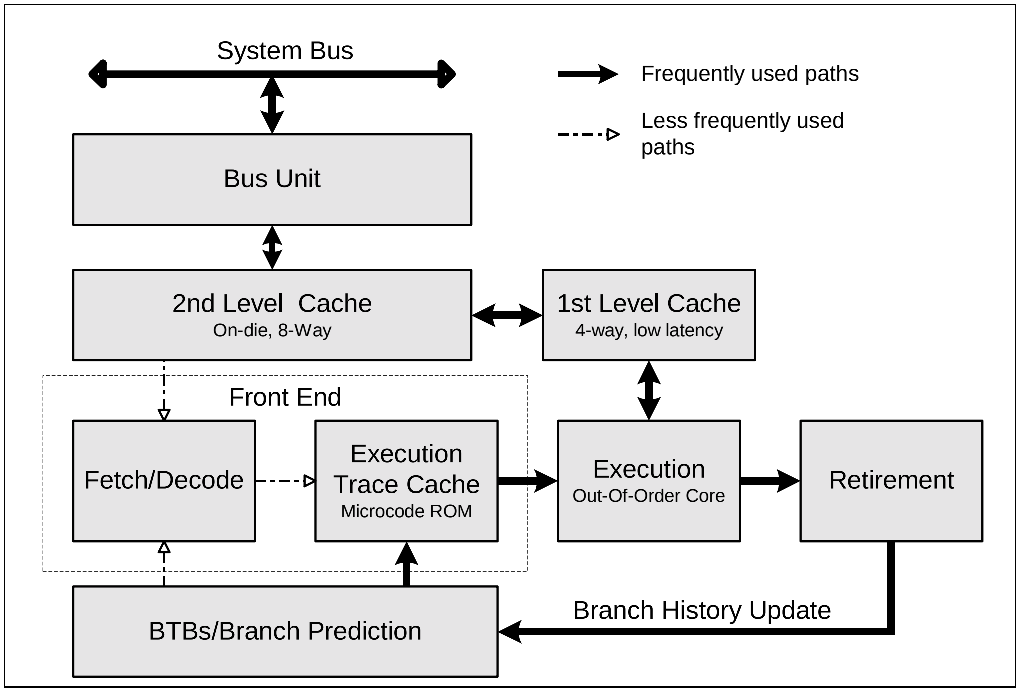


Figure 2-2. The Intel NetBurst Micro-Architecture

The execution trace cache addresses both of these issues by storing decoded instructions. Instructions are fetched and decoded by the translation engine and built into sequences of µops called traces. These traces of µops are stored in the trace cache. The instructions from the most likely target of a branch immediately follow the branch without regard for contiguity of instruction addresses. Once a trace is built, the trace cache is searched for the instruction that follows that trace. If that instruction appears as the first instruction in an existing trace, the fetch and decode of instructions from the memory hierarchy ceases and the trace cache becomes the new source of instructions. The critical execution loop in the Intel NetBurst micro-architecture is illustrated in Figure 2-2, it is simpler than the execution loop in the P6 micro-architecture that is shown in Figure 2-1.

The execution trace cache and the translation engine have cooperating branch prediction hardware. Branch targets are predicted based on their linear addresses using branch target buffers (BTBs) and fetched as soon as possible. Branch targets are fetched from the trace cache if they are indeed cached there, otherwise they are fetched from the memory hierarchy. The translation engine's branch prediction information is used to form traces along the most likely paths.

### 2.5.2. The Out-of-order Core

The core's ability to execute instructions out of order is a key factor in enabling parallelism. This feature enables the processor to reorder instructions so that if one µop is delayed while waiting for data or a contended execution resource, other µops that are later in program order may proceed around it. The processor employs several buffers to smooth the flow of µops. This implies that when one portion of the pipeline experiences a delay, that delay may be covered by other operations executing in parallel or by the execution of µops which were previously queued up in a buffer.

The core is designed to facilitate parallel execution. It can dispatch up to six µops per cycle; note that this exceeds the trace cache and retirement µop bandwidth. Most pipelines can start executing a new µop every cycle, so that several instructions can be in flight at a time for each pipeline. A number of arithmetic logical unit (ALU) instructions can start two per cycle, and many floating-point instructions can start one every two cycles. Finally, µops can begin execution, out of order, as soon as their data inputs are ready and resources are available.

### 2.5.3. Retirement

The retirement section receives the results of the executed µops from the execution core and processes the results so that the proper architectural state is updated according to the original program order. For semantically-correct execution, the results of IA-32 instructions must be committed in original program order before it is retired. Exceptions may be raised as instructions retired. Thus, exceptions cannot occur speculatively, they occur in the correct order, and the machine can be correctly restarted after an exception.

When a µop completes and writes its result to the destination, it is retired. Up to three µops may be retired per cycle. The Reorder Buffer (ROB) is the unit in the processor which buffers completed µops, updates the architectural state in order, and manages the ordering of exceptions.

The retirement section also keeps track of branches and sends updated branch target information to the BTB to update branch history. In this manner, traces that are no longer needed can be purged from the trace cache and new branch paths can be fetched, based on updated branch history information.

# 3

# IA-32 Execution Environment

# CHAPTER 3
# BASIC EXECUTION ENVIRONMENT

This chapter describes the basic execution environment of an IA-32 processor as seen by assembly-language programmers. It describes how the processor executes instructions and how it stores and manipulates data. The parts of the execution environment described here include memory (the address space), the general-purpose data registers, the segment registers, the EFLAGS register, and the instruction pointer register.

## 3.1. MODES OF OPERATION

The IA-32 architecture supports three operating modes: protected mode, real-address mode, and system management mode. The operating mode determines which instructions and architectural features are accessible:

- Protected mode. This mode is the native state of the processor. In this mode all instructions and architectural features are available, providing the highest performance and capability. This is the recommended mode for all new applications and operating systems. Among the capabilities of protected mode is the ability to directly execute "real-address mode" 8086 software in a protected, multi-tasking environment. This feature is called virtual-8086 mode , although it is not actually a processor mode. Virtual-8086 mode is actually a protected mode attribute that can be enabled for any task.
- Real-address mode. This mode implements the programming environment of the Intel 8086 processor with a few extensions (such as the ability to switch to protected or system management mode). The processor is placed in real-address mode following power-up or a reset.
- System management mode (SSM). This mode provides an operating system or executive with a transparent mechanism for implementing platform-specific functions such as power management and system security. The processor enters SMM when the external SMM interrupt pin (SMI#) is activated or an SMI is received from the advanced programmable interrupt controller (APIC). In SMM, the processor switches to a separate address space while saving the entire context of the currently running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the system management interrupt. SSM was introduced with the Intel386$^{TM}$ SL and Intel486$^{TM}$ SL processors and became a standard IA-32 feature with the Pentium processor family.

The basic execution environment is the same for each of these operating modes, as is described in the remaining sections of this chapter.

## 3.2. OVERVIEW OF THE BASIC EXECUTION ENVIRONMENT

Any program or task running on an IA-32 processor is given a set of resources for executing instructions and for storing code, data, and state information. These resources (described briefly in the following paragraphs and shown in Figure 3-1) make up the basic execution environment for an IA-32 processor. This basic execution environment is used jointly by the application programs and the operating-system or executive running on the processor.

- Address Space. Any task or program running on an IA-32 processor can address a linear address space of up to 4 GBytes ($2^{32}$ bytes) and a physical address space of up to 64 GBytes ($2^{36}$ bytes). (See Section 3.3.3., “Extended Physical Addressing” for more information about addressing an address space greater than 4 GBytes.)
- Basic program execution registers . The eight general-purpose registers, the six segment registers, the EFLAGS register, and the EIP (instruction pointer) register comprise a basic execution environment in which to execute a set of general-purpose instructions. These instructions perform basic integer arithmetic on byte, word, and doubleword integers, handle program flow control, operate on bit and byte strings, and address memory.
- x87 FPU registers. The eight x87 FPU data registers, the x87 FPU control register, the status register, the x87 FPU instruction pointer register, the x87 FPU operand (data) pointer register, the x87 FPU tag register, and the x87 FPU opcode register provide an execution environment for operating on single-precision, double-precision, and double extended-precision floating-point values, word-, doubleword, and quadword integers, and binary coded decimal (BCD) values.
- MMX™ registers. The eight MMX registers support execution of single-instruction, multiple-data (SIMD) operations on 64-bit packed byte, word, and doubleword integers.
- XMM registers. The eight XMM data registers and the MXCSR register support execution of SIMD operations on 128-bit packed single-precision and double-precision floating-point values and on 128-bit packed byte, word, doubleword, and quadword integers.
- Stack. To support procedure or subroutine calls and the passing of parameters between procedures or subroutines, a stack and stack management resources are included in the execution environment. The stack (not shown in Figure 3-1) is located in memory.

In addition to the resources provided in the basic execution environment, the IA-32 architecture provides the following system resources. These resources are part of the IA-32 architecture’s system-level architecture. They provide extensive support for operating-system and system-development software. Except for the I/O ports, the system resources are described in detail in the Intel Architecture Software Developer’s Manual, Volume 3: System Programming Guide.

- I/O Ports. The IA-32 architecture supports a transfers of data to and from input/output (I/O) ports (see Chapter 12, Input/Output, in this volume).
- Control registers. The five control registers (CR0 through CR5) determine the operating mode of the processor and the characteristics of the currently executing task (see the section titled “Control Registers” in Chapter 3 of the Intel Architecture Software Developer’s Manual, Volume 3).

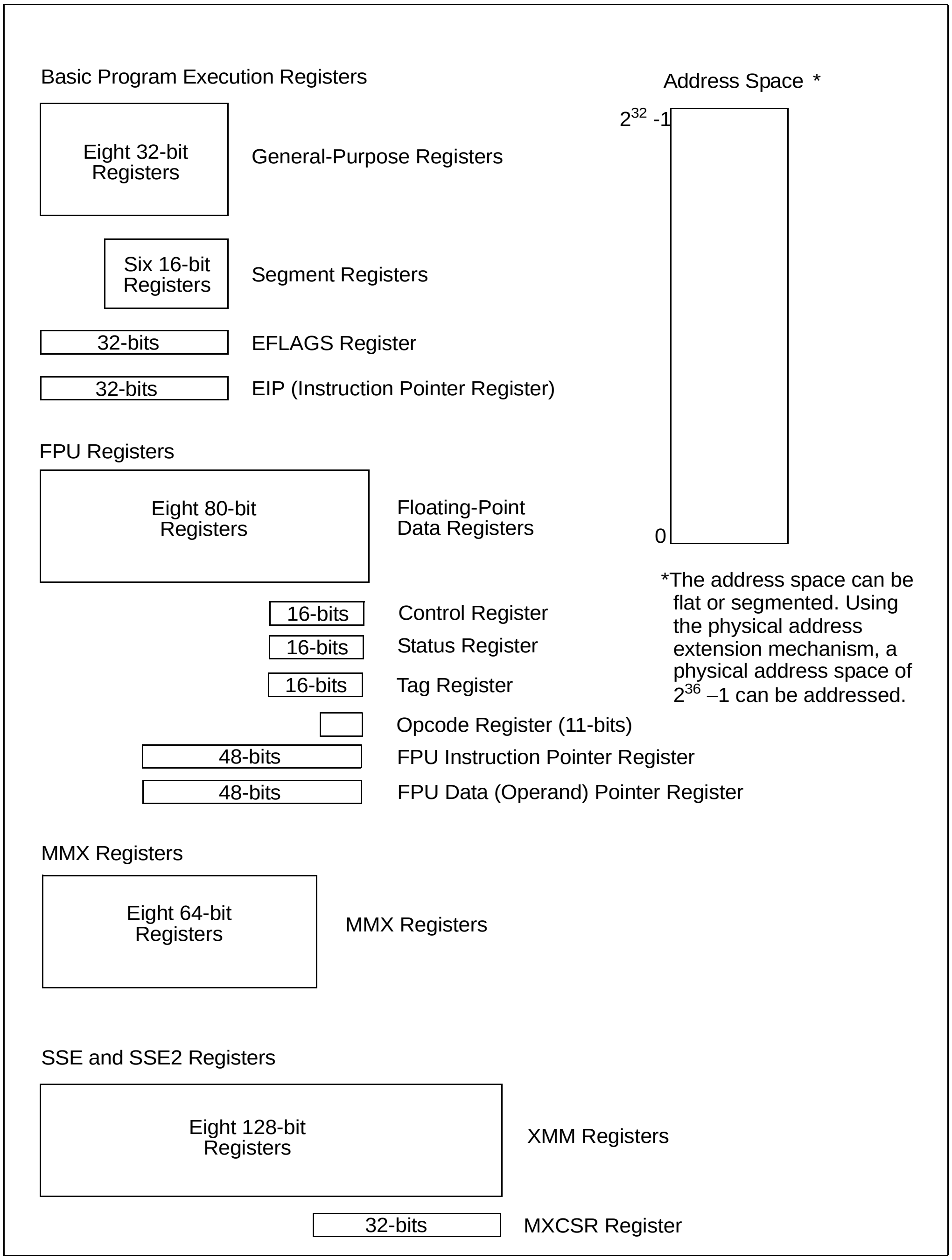

Basic Program Execution Registers
Eight 32-bit Registers
General-Purpose Registers
Six 16-bit Registers
Segment Registers
32-bits
EFLAGS Register
32-bits
EIP (Instruction Pointer Register)
FPU Registers
Eight 80-bit Registers
Floating-Point Data Registers
16-bits
Control Register
16-bits
Status Register
16-bits
Tag Register
Opcode Register (11-bits)
48-bits
FPU Instruction Pointer Register
48-bits
FPU Data (Operand) Pointer Register
MMX Registers
Eight 64-bit Registers
MMX Registers
SSE and SSE2 Registers
Eight 128-bit Registers
XMM Registers
32-bits
MXCSR Register
Address Space *
2^32 -1
0
*The address space can be flat or segmented. Using the physical address extension mechanism, a physical address space of 2^36 −1 can be addressed.

Figure 3-1.  IA-32 Basic Execution Environment

- Memory management registers. The GDTR, IDTR, task register, and LDTR specify the locations of data structures used in protected mode memory management (see the section titled "Memory-Management Registers" in Chapter 3 of the Intel Architecture Software Developer's Manual, Volume 3).
- Debug registers. The debug registers (DR0 through DR7) control and allow monitoring of the processor's debugging operations (see the section titled "Debug Registers" in Chapter 14 of the Intel Architecture Software Developer's Manual, Volume 3).
- Memory type range registers (MTRRs). The MTRRs are used to assign memory types to regions of memory (see the section titled "Memory Type Range Registers [MTRRs]" in Chapter 9 of the Intel Architecture Software Developer's Manual, Volume 3).
- Machine check registers (MCRs). The MCR registers consist of a set of control, status, and error-reporting registers that are used to detect and report on hardware (machine) errors (see the section titled "Machine-Check MSRs" in Chapter 12 of the Intel Architecture Software Developer's Manual, Volume 3).
- Performance monitoring counters. The performance monitoring counters allow processor performance events to be monitored (see the section titled "Performance-Monitoring Counters" in Chapter 14 of the Intel Architecture Software Developer's Manual, Volume 3).

The remainder of this chapter describes the organization of memory and the address space, the basic program execution registers, and addressing modes. Refer to the following chapters in this volume for descriptions of the other program execution resources shown in Figure 3-1:

- x87 FPU registers—See Chapter 8, Programming with the x87 FPU.
- MMX Registers—See Chapter 9, Programming With the Intel MMX Technology
- XMM registers—See Chapter 10, Programming with the Streaming SiMD Extensions (SSE) and Chapter 11, Programming With the Streaming SIMD Extensions 2 (SSE2) respectively.
- Stack implementation and procedure calls—See Chapter 6, Procedure Calls, Interrupts, and Exceptions.

## 3.3. MEMORY ORGANIZATION

The memory that the processor addresses on its bus is called physical memory. Physical memory is organized as a sequence of 8-bit bytes. Each byte is assigned a unique address, called a physical address. The physical address space ranges from zero to a maximum of $2^{36} - 1$ (64 GBytes).

Virtually any operating system or executive designed to work with an IA-32 processor will use the processor's memory management facilities to access memory. These facilities provide features such as segmentation and paging, which allow memory to be managed efficiently and reliably. Memory management is described in detail in Chapter 3, Protected-Mode Memory Management, in the Intel Architecture Software Developer's Manual, Volume 3. The following paragraphs describe the basic methods of addressing memory when memory management is used.

When employing the processor's memory management facilities, programs do not directly address physical memory. Instead, they access memory using any of three memory models: flat, segmented, or real-address mode.

With the flat memory model (see Figure 3-2), memory appears to a program as a single, continuous address space, called a linear address space . Code (a program's instructions), data, and the procedure stack are all contained in this address space. The linear address space is byte addressable, with addresses running contiguously from 0 to $2^{32} - 1$. An address for any byte in the linear address space is called a linear address .

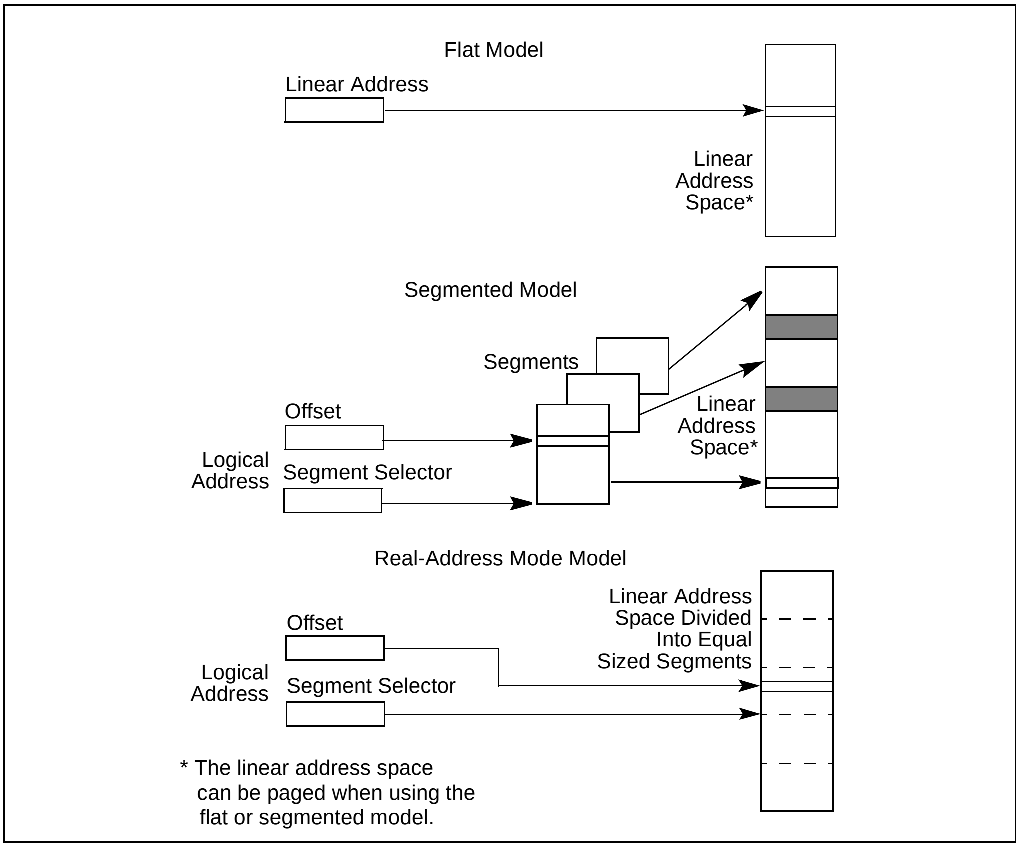


Figure 3-2. Three Memory Management Models

With the segmented memory model, memory appears to a program as a group of independent address spaces called segments. When using this model, code, data, and stacks are typically contained in separate segments. To address a byte in a segment, a program must issue a logical address , which consists of a segment selector and an offset. (A logical address is often referred to as a far pointer .) The segment selector identifies the segment to be accessed and the offset identifies a byte in the address space of the segment. The programs running on an IA-32 processor can address up to 16,383 segments of different sizes and types, and each segment can be as large as $2^{32}$ bytes.

Internally, all the segments that are defined for a system are mapped into the processor's linear address space. To access a memory location, the processor thus translates each logical address into a linear address. This translation is transparent to the application program.

The primary reason for using segmented memory is to increase the reliability of programs and systems. For example, placing a program's stack in a separate segment prevents the stack from growing into the code or data space and overwriting instructions or data, respectively. Placing the operating system's or executive's code, data, and stack in separate segments also protects them from the application program and vice versa.

With the flat or the segmented memory model, the linear address space is mapped into the processor's physical address space either directly or through paging. When using direct mapping (paging disabled), each linear address has a one-to-one correspondence with a physical address (that is, linear addresses are sent out on the processor's address lines without translation). When using the IA-32 architecture's paging mechanism (paging enabled), the linear address space is divided into pages, which are mapped into virtual memory. The pages of virtual memory are then mapped as needed into physical memory. When an operating system or executive uses paging, the paging mechanism is transparent to an application program; that is, all the application program sees is the linear address space.

The real-address mode model uses the memory model for the Intel 8086 processor. This memory model is supported in the IA-32 architecture for compatibility with existing programs written to run on the Intel 8086 processor. The real-address mode uses a specific implementation of segmented memory in which the linear address space for the program and the operating system/executive consists of an array of segments of up to 64 KBytes in size each. The maximum size of the linear address space in real-address mode is $2^{20}$ bytes. (See Chapter 15, 8086 Emulation, in the Intel Architecture Software Developer's Manual, Volume 3, for more information on this memory model.)

### 3.3.1. Modes of Operation vs. Memory Model

When writing code for an IA-32 processor, a programmer needs to know the operating mode the processor is going to be in when executing the code and the memory model being used. The relationship between operating modes and memory models is as follows:

- Protected mode. When in protected mode, the processor can use any of the memory models described in this section. (The real-addressing mode memory model is ordinarily used only when the processor is in the virtual-8086 mode.) The memory model used depends on the design of the operating system or executive. When multitasking is implemented, individual tasks can use different memory models.

- Real-address mode. When in real-address mode, the processor only supports the real-address mode memory model.
- System management mode. When in SMM, the processor switches to a separate address space, called the system management RAM (SMRAM). The memory model used to address bytes in this address space is similar to the real-address mode model. (See Chapter 11, in the Intel Architecture Software Developer's Manual, Volume 3, for more information on the memory model used in SMM.)

### 3.3.2. 32-Bit vs. 16-Bit Address and Operand Sizes

The processor can be configured for 32-bit or 16-bit address and operand sizes. With 32-bit address and operand sizes, the maximum linear address or segment offset is FFFFFFFFH ($2^{32}$-1), and operand sizes are typically 8 bits or 32 bits. With 16-bit address and operand sizes, the maximum linear address or segment offset is FFFFH ($2^{16}$-1), and operand sizes are typically 8 bits or 16 bits.

When using 32-bit addressing, a logical address (or far pointer) consists of a 16-bit segment selector and a 32-bit offset; when using 16-bit addressing, it consists of a 16-bit segment selector and a 16-bit offset.

Instruction prefixes allow temporary overrides of the default address and/or operand sizes from within a program.

When operating in protected mode, the segment descriptor for the currently executing code segment defines the default address and operand size. A segment descriptor is a system data structure not normally visible to application code. Assembler directives allow the default addressing and operand size to be chosen for a program. The assembler and other tools then set up the segment descriptor for the code segment appropriately.

When operating in real-address mode, the default addressing and operand size is 16 bits. An address-size override can be used in real-address mode to enable 32-bit addressing; however, the maximum allowable 32-bit linear address is still 000FFFFFH ($2^{20}$-1).

### 3.3.3. Extended Physical Addressing

Beginning with the Pentium Pro processor, the IA-32 architecture supports addressing of up to 64 GBytes ($2^{36}$ bytes) of physical memory. A program or task cannot address locations in this address space directly. Instead it addresses individual linear address spaces of up to 4 GBytes that are mapped to the larger 64-GByte physical address space through the processor's virtual memory management mechanism. A program can switch between linear address spaces within this 64-GByte physical address space by changing segment selectors in the segment registers. The use of extended physical addressing requires the processor to operate in protected mode and the operating system to provide a virtual memory management system. (See "Physical Address Extension" in Chapter 3 of the Intel Architecture Software Developer's Manual, Volume 3 for more information about this addressing mechanism.)

## 3.4. BASIC PROGRAM EXECUTION REGISTERS

The processor provides 16 registers basic program execution registers for use in general system and application programing. As shown in Figure 3-3, these registers can be grouped as follows:

- General-purpose registers . These eight registers are available for storing operands and pointers.
- Segment registers . These registers hold up to six segment selectors.
- EFLAGS (program status and control) register . The EFLAGS register report on the status of the program being executed and allows limited (application-program level) control of the processor.
- EIP (instruction pointer) register. The EIP register contains a 32-bit pointer to the next instruction to be executed.

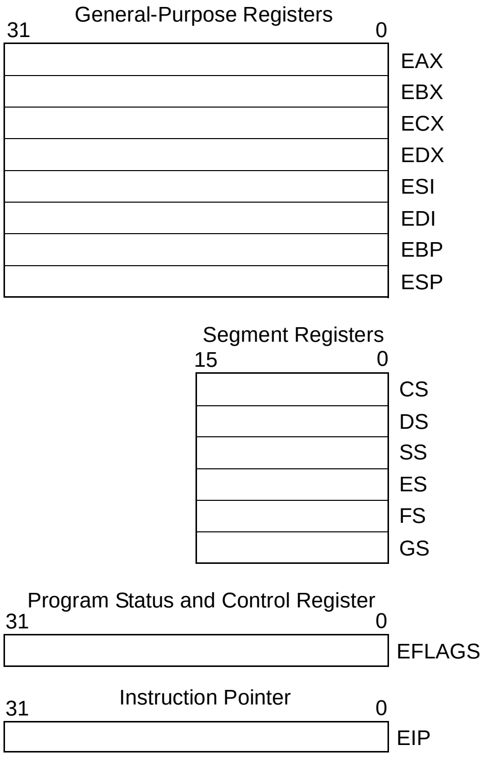


Figure 3-3. Application Programming Registers

### 3.4.1. General-Purpose Registers

The 32-bit general-purpose registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for holding the following items:

- Operands for logical and arithmetic operations
- Operands for address calculations
- Memory pointers.

Although all of these registers are available for general storage of operands, results, and pointers, caution should be used when referencing the ESP register. The ESP register holds the stack pointer and as a general rule should not be used for any other purpose.

Many instructions assign specific registers to hold operands. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands. When using a segmented memory model, some instructions assume that pointers in certain registers are relative to specific segments. For instance, some instructions assume that a pointer in the EBX register points to a memory location in the DS segment.

The special uses of general-purpose registers by instructions are described in Chapter 5, Instruction Set Summary, in this volume and Chapter 3, Instruction Set Reference, in the Intel Architecture Software Developer's Manual, Volume 2. The following is a summary of these special uses:

- EAX—Accumulator for operands and results data.
- EBX—Pointer to data in the DS segment.
- ECX—Counter for string and loop operations.
- EDX—I/O pointer.
- ESI—Pointer to data in the segment pointed to by the DS register; source pointer for string operations.
- EDI—Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations.
- ESP—Stack pointer (in the SS segment).
- EBP—Pointer to data on the stack (in the SS segment).

As shown in Figure 3-4, the lower 16 bits of the general-purpose registers map directly to the register set found in the 8086 and Intel 286 processors and can be referenced with the names AX, BX, CX, DX, BP, SP, SI, and DI. Each of the lower two bytes of the EAX, EBX, ECX, and EDX registers can be referenced by the names AH, BH, CH, and DH (high bytes) and AL, BL, CL, and DL (low bytes).

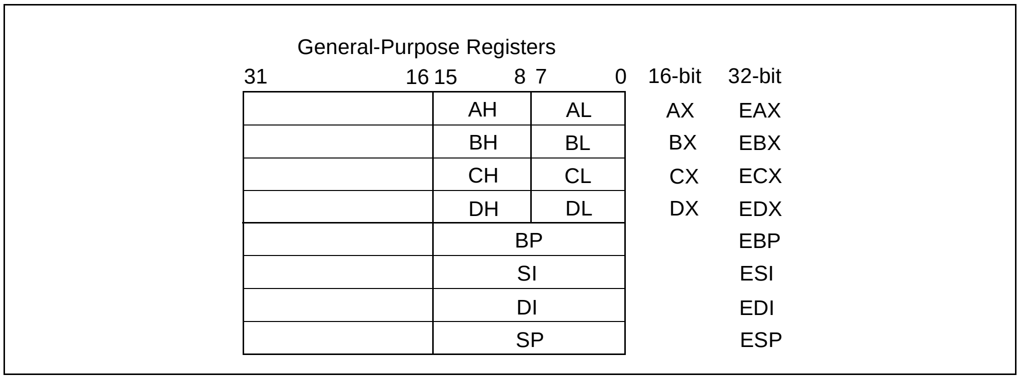


Figure 3-4. Alternate General-Purpose Register Names

## 3.4.2. Segment Registers

The segment registers (CS, DS, SS, ES, FS, and GS) hold 16-bit segment selectors. A segment selector is a special pointer that identifies a segment in memory. To access a particular segment in memory, the segment selector for that segment must be present in the appropriate segment register.

When writing application code, programmers generally create segment selectors with assembler directives and symbols. The assembler and other tools then create the actual segment selector values associated with these directives and symbols. If writing system code, programmers may need to create segment selectors directly. (A detailed description of the segment-selector data structure is given in Chapter 3, Protected-Mode Memory Management, of the Intel Architecture Software Developer's Manual, Volume 3.)

How segment registers are used depends on the type of memory management model that the operating system or executive is using. When using the flat (unsegmented) memory model, the segment registers are loaded with segment selectors that point to overlapping segments, each of which begins at address 0 of the linear address space (as shown in Figure 3-5). These overlapping segments then comprise the linear address space for the program. (Typically, two overlapping segments are defined: one for code and another for data and stacks. The CS segment register points to the code segment and all the other segment registers point to the data and stack segment.)

When using the segmented memory model, each segment register is ordinarily loaded with a different segment selector so that each segment register points to a different segment within the linear address space (as shown in Figure 3-6). At any time, a program can thus access up to six segments in the linear address space. To access a segment not pointed to by one of the segment registers, a program must first load the segment selector for the segment to be accessed into a segment register.

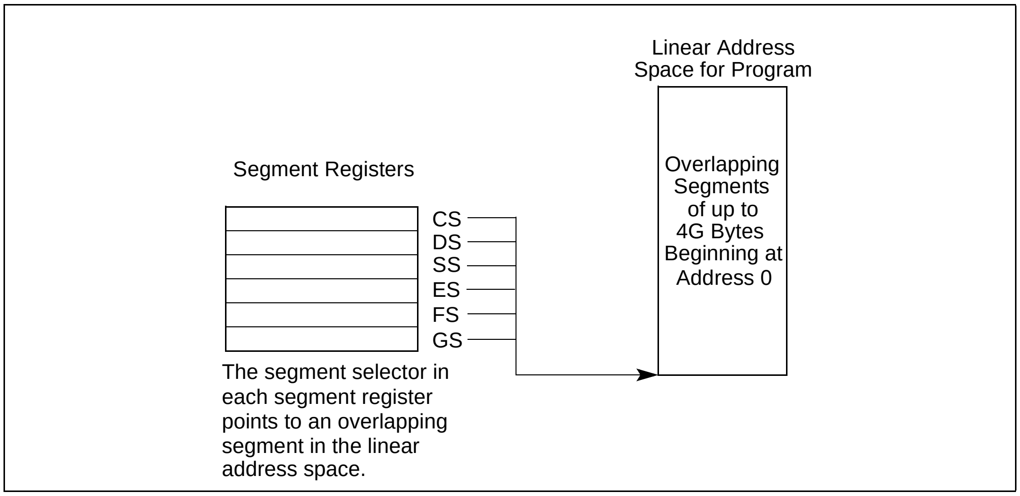

Linear Address
Space for Program
Segment Registers
CS
DS
SS
ES
FS
GS
Overlapping Segments of up to 4G Bytes Beginning at Address 0
The segment selector in each segment register points to an overlapping segment in the linear address space.

Figure 3-5. Use of Segment Registers for Flat Memory Model

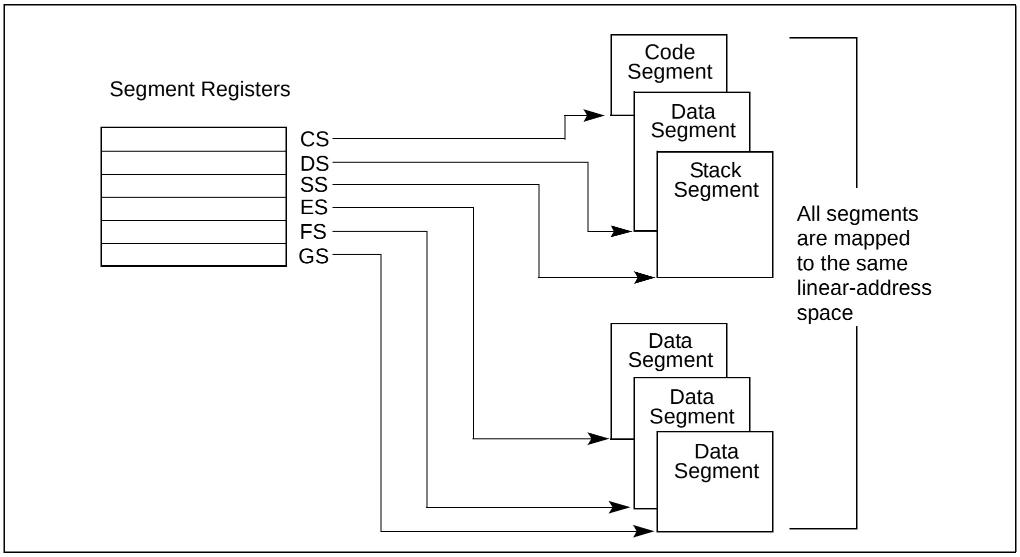


Figure 3-6. Use of Segment Registers in Segmented Memory Model

Each of the segment registers is associated with one of three types of storage: code, data, or stack). For example, the CS register contains the segment selector for the code segment , where the instructions being executed are stored. The processor fetches instructions from the code segment, using a logical address that consists of the segment selector in the CS register and the contents of the EIP register. The EIP register contains the offset within the code segment of the next instruction to be executed. The CS register cannot be loaded explicitly by an application program. Instead, it is loaded implicitly by instructions or internal processor operations that change program control (such as, procedure calls, interrupt handling, or task switching).

The DS, ES, FS, and GS registers point to four data segments . The availability of four data segments permits efficient and secure access to different types of data structures. For example, four separate data segments might be created: one for the data structures of the current module, another for the data exported from a higher-level module, a third for a dynamically created data structure, and a fourth for data shared with another program. To access additional data segments, the application program must load segment selectors for these segments into the DS, ES, FS, and GS registers, as needed.

The SS register contains the segment selector for a stack segment , where the procedure stack is stored for the program, task, or handler currently being executed. All stack operations use the SS register to find the stack segment. Unlike the CS register, the SS register can be loaded explicitly, which permits application programs to set up multiple stacks and switch among them.

See Section 3.3., “Memory Organization”, for an overview of how the segment registers are used in real-address mode.

The four segment registers CS, DS, SS, and ES are the same as the segment registers found in the Intel 8086 and Intel 286 processors and the FS and GS registers were introduced into the IA-32 Architecture with the Intel386™ family of processors.

### 3.4.3. EFLAGS Register

The 32-bit EFLAGS register contains a group of status flags, a control flag, and a group of system flags. Figure 3-7 defines the flags within this register. Following initialization of the processor (either by asserting the RESET pin or the INIT pin), the state of the EFLAGS register is 00000002H. Bits 1, 3, 5, 15, and 22 through 31 of this register are reserved. Software should not use or depend on the states of any of these bits.

Some of the flags in the EFLAGS register can be modified directly, using special-purpose instructions (described in the following sections). There are no instructions that allow the whole register to be examined or modified directly. However, the following instructions can be used to move groups of flags to and from the procedure stack or the EAX register: LAHF, SAHF, PUSHF, PUSHFD, POPF, and POPFD. After the contents of the EFLAGS register have been transferred to the procedure stack or EAX register, the flags can be examined and modified using the processor’s bit manipulation instructions (BT, BTS, BTR, and BTC).

When suspending a task (using the processor’s multitasking facilities), the processor automatically saves the state of the EFLAGS register in the task state segment (TSS) for the task being suspended. When binding itself to a new task, the processor loads the EFLAGS register with data from the new task’s TSS.

When a call is made to an interrupt or exception handler procedure, the processor automatically saves the state of the EFLAGS registers on the procedure stack. When an interrupt or exception is handled with a task switch, the state of the EFLAGS register is saved in the TSS for the task being suspended.

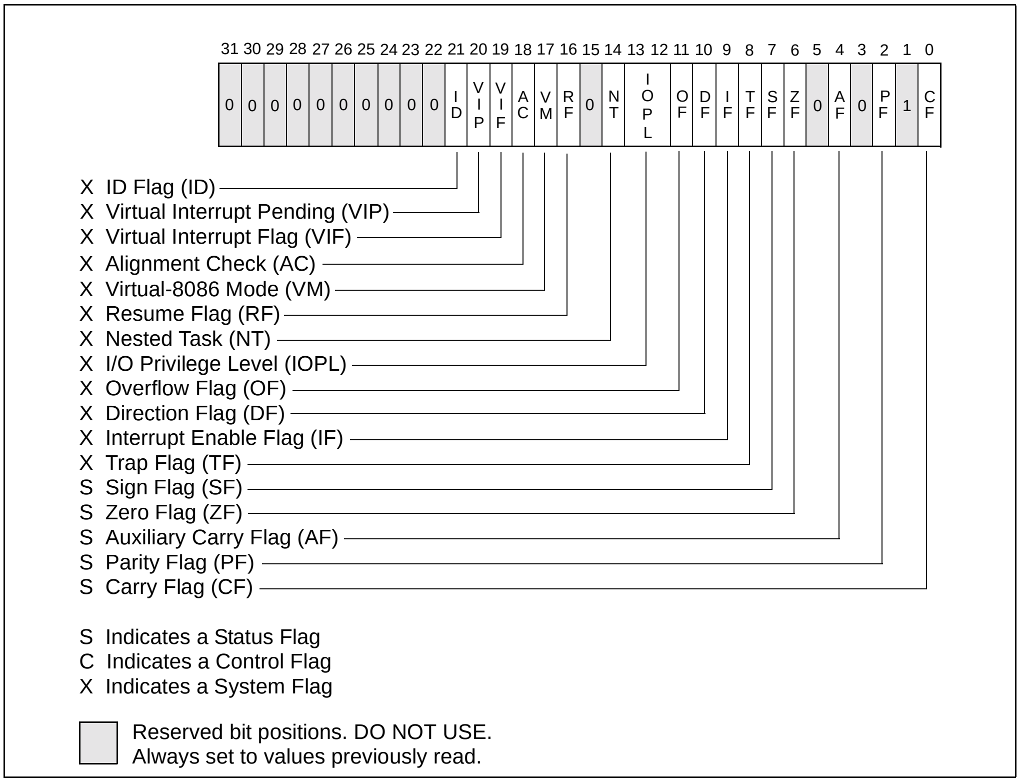

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
X Overflow Flag (OF)
X Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE.
Always set to values previously read.

Figure 3-7. EFLAGS Register

As the IA-32 Architecture has evolved, flags have been added to the EFLAGS register, but the function and placement of existing flags have remained the same from one family of the IA-32 processors to the next. As a result, code that accesses or modifies these flags for one family of IA-32 processors works as expected when run on later families of processors.

### 3.4.3.1. STATUS FLAGS

The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions. The functions of the status flags are as follows:

CF (bit 0) Carry flag. Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.

PF (bit 2) Parity flag. Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.

AF (bit 4) Adjust flag. Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.

ZF (bit 6) Zero flag. Set if the result is zero; cleared otherwise.

SF (bit 7) Sign flag. Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

OF (bit 11) Overflow flag. Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.

Of these status flags, only the CF flag can be modified directly, using the STC, CLC, and CMC instructions. Also the bit instructions (BT, BTS, BTR, and BTC) copy a specified bit into the CF flag.

The status flags allow a single arithmetic operation to produce results for three different data types: unsigned integers, signed integers, and BCD integers. If the result of an arithmetic operation is treated as an unsigned integer, the CF flag indicates an out-of-range condition (carry or a borrow); if treated as a signed integer (two's complement number), the OF flag indicates a carry or borrow; and if treated as a BCD digit, the AF flag indicates a carry or borrow. The SF flag indicates the sign of a signed integer. The ZF flag indicates either a signed- or an unsigned-integer zero.

When performing multiple-precision arithmetic on integers, the CF flag is used in conjunction with the add with carry (ADC) and subtract with borrow (SBB) instructions to propagate a carry or borrow from one computation to the next.

The condition instructions Jcc (jump on condition code cc), SETcc (byte set on condition code cc), LOOPcc, and CMOVcc (conditional move) use one or more of the status flags as condition codes and test them for branch, set-byte, or end-loop conditions.

### 3.4.3.2. DF FLAG

The direction flag (DF, located in bit 10 of the EFLAGS register) controls the string instructions (MOVS, CMPS, SCAS, LODS, and STOS). Setting the DF flag causes the string instructions to auto-decrement (that is, to process strings from high addresses to low addresses). Clearing the DF flag causes the string instructions to auto-increment (process strings from low addresses to high addresses).

The STD and CLD instructions set and clear the DF flag, respectively.

## 3.4.4. System Flags and IOPL Field

The system flags and IOPL field in the EFLAGS register control operating-system or executive operations. They should not be modified by application programs. The functions of the system flags are as follows:

IF (bit 9) Interrupt enable flag. Controls the response of the processor to maskable interrupt requests. Set to respond to maskable interrupts; cleared to inhibit maskable interrupts.

TF (bit 8) Trap flag. Set to enable single-step mode for debugging; clear to disable single-step mode.

IOPL (bits 12 and 13) I/O privilege level field. Indicates the I/O privilege level of the currently running program or task. The current privilege level (CPL) of the currently running program or task must be less than or equal to the I/O privilege level to access the I/O address space. This field can only be modified by the POPF and IRET instructions when operating at a CPL of 0.

NT (bit 14) Nested task flag. Controls the chaining of interrupted and called tasks. Set when the current task is linked to the previously executed task; cleared when the current task is not linked to another task.

RF (bit 16) Resume flag. Controls the processor's response to debug exceptions.

VM (bit 17) Virtual-8086 mode flag. Set to enable virtual-8086 mode; clear to return to protected mode.

AC (bit 18) Alignment check flag. Set this flag and the AM bit in the CR0 register to enable alignment checking of memory references; clear the AC flag and/or the AM bit to disable alignment checking.

VIF (bit 19) Virtual interrupt flag. Virtual image of the IF flag. Used in conjunction with the VIP flag. (To use this flag and the VIP flag the virtual mode extensions are enabled by setting the VME flag in control register CR4.)

VIP (bit 20) Virtual interrupt pending flag. Set to indicate that an interrupt is pending; clear when no interrupt is pending. (Software sets and clears this flag; the processor only reads it.) Used in conjunction with the VIF flag.

ID (bit 21) Identification flag. The ability of a program to set or clear this flag indicates support for the CPUID instruction.

See Chapter 3, Protected-Mode Memory Management, in the Intel Architecture Software Developer's Manual, Volume 3, for a detail description of these flags.

## 3.5. INSTRUCTION POINTER

The instruction pointer (EIP) register contains the offset in the current code segment for the next instruction to be executed. It is advanced from one instruction boundary to the next in straight-line code or it is moved ahead or backwards by a number of instructions when executing JMP, Jcc, CALL, RET, and IRET instructions.

The EIP register cannot be accessed directly by software; it is controlled implicitly by control-transfer instructions (such as JMP, Jcc, CALL, and RET), interrupts, and exceptions. The only way to read the EIP register is to execute a CALL instruction and then read the value of the return instruction pointer from the procedure stack. The EIP register can be loaded indirectly by modifying the value of a return instruction pointer on the procedure stack and executing a return instruction (RET or IRET). See Section 6.2.4.2., "Return Instruction Pointer".

All IA-32 processors prefetch instructions. Because of instruction prefetching, an instruction address read from the bus during an instruction load does not match the value in the EIP register. Even though different processor generations use different prefetching mechanisms, the function of EIP register to direct program flow remains fully compatible with all software written to run on IA-32 processors.

## 3.6. OPERAND-SIZE AND ADDRESS-SIZE ATTRIBUTES

When the processor is executing in protected mode, every code segment has a default operand-size attribute and address-size attribute. These attributes are selected with the D (default size) flag in the segment descriptor for the code segment (see Chapter 3, Protected-Mode Memory Management, in the Intel Architecture Software Developer's Manual, Volume 3). When the D flag is set, the 32-bit operand-size and address-size attributes are selected; when the flag is clear, the 16-bit size attributes are selected. When the processor is executing in real-address mode, virtual-8086 mode, or SMM, the default operand-size and address-size attributes are always 16 bits.

The operand-size attribute selects the sizes of operands that instructions operate on. When the 16-bit operand-size attribute is in force, operands can generally be either 8 bits or 16 bits, and when the 32-bit operand-size attribute is in force, operands can generally be 8 bits or 32 bits.

The address-size attribute selects the sizes of addresses used to address memory: 16 bits or 32 bits. When the 16-bit address-size attribute is in force, segment offsets and displacements are 16 bits. This restriction limits the size of a segment that can be addressed to 64 KBytes. When the 32-bit address-size attribute is in force, segment offsets and displacements are 32 bits, allowing segments of up to 4 GBytes to be addressed.

The default operand-size attribute and/or address-size attribute can be overridden for a particular instruction by adding an operand-size and/or address-size prefix to an instruction (see "Instruction Prefixes" in Chapter 2 of the Intel Architecture Software Developer's Manual, Volume 3). The effect of this prefix applies only to the instruction it is attached to.

Table 3-1 shows effective operand size and address size (when executing in protected mode) depending on the settings of the D flag and the operand-size and address-size prefixes.

Table 3-1. Effective Operand- and Address-Size Attributes

| D Flag in Code Segment Descriptor | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|
| Operand-Size Prefix 66H | N | N | Y | Y | N | N | Y | Y |
| Address-Size Prefix 67H | N | Y | N | Y | N | Y | N | Y |

Table 3-1. Effective Operand- and Address-Size Attributes

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Effective Operand Size | 16 | 16 | 32 | 32 | 32 | 32 | 16 | 16 |
| Effective Address Size | 16 | 32 | 16 | 32 | 32 | 16 | 32 | 16 |

NOTES:

Y Yes, this instruction prefix is present.

N No, this instruction prefix is not present.

## 3.7. OPERAND ADDRESSING

IA32 machine-instructions acts on zero or more operands. Some operands are specified explicitly in an instruction and others are implicit to an instruction. An operand can be located in any of the following places:

- The instruction itself (an immediate operand).
- A register.
- A memory location.
- An I/O port.

### 3.7.1. Immediate Operands

Some instructions use data encoded in the instruction itself as a source operand. These operands are called immediate operands (or simply immediates). For example, the following ADD instruction adds an immediate value of 14 to the contents of the EAX register:

```
ADD EAX, 14
```

All the arithmetic instructions (except the DIV and IDIV instructions) allow the source operand to be an immediate value. The maximum value allowed for an immediate operand varies among instructions, but can never be greater than the maximum value of an unsigned doubleword integer ($2^{32}$).

### 3.7.2. Register Operands

Source and destination operands can be located in any of the following registers, depending on the instruction being executed:

- The 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP).
- The 16-bit general-purpose registers (AX, BX, CX, DX, SI, DI, SP, or BP).
- The 8-bit general-purpose registers (AH, BH, CH, DH, AL, BL, CL, or DL).
- The segment registers (CS, DS, SS, ES, FS, and GS).

- The EFLAGS register.
- System registers, such as the global descriptor table (GDTR) or the interrupt descriptor table register (IDTR).

Some instructions (such as the DIV and MUL instructions) use quadword operands contained in a pair of 32-bit registers. Register pairs are represented with a colon separating them. For example, in the register pair EDX:EAX, EDX contains the high order bits and EAX contains the low order bits of a quadword operand.

Several instructions (such as the PUSHFD and POPFD instructions) are provided to load and store the contents of the EFLAGS register or to set or clear individual flags in this register. Other instructions (such as the Jcc instructions) use the state of the status flags in the EFLAGS register as condition codes for branching or other decision making operations.

The processor contains a selection of system registers that are used to control memory management, interrupt and exception handling, task management, processor management, and debugging activities. Some of these system registers are accessible by an application program, the operating system, or the executive through a set of system instructions. When accessing a system register with a system instruction, the register is generally an implied operand of the instruction.

## 3.7.3. Memory Operands

Source and destination operands in memory are referenced by means of a segment selector and an offset (see Figure 3-8). The segment selector specifies the segment containing the operand and the offset (the number of bytes from the beginning of the segment to the first byte of the operand) specifies the linear or effective address of the operand.

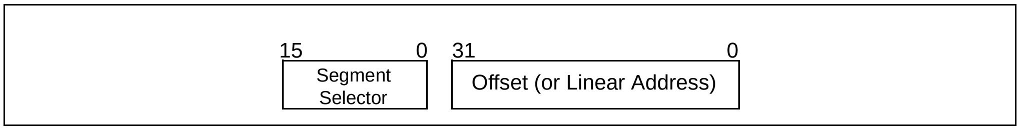


Figure 3-8. Memory Operand Address

### 3.7.3.1. SPECIFYING A SEGMENT SELECTOR

The segment selector can be specified either implicitly or explicitly. The most common method of specifying a segment selector is to load it in a segment register and then allow the processor to select the register implicitly, depending on the type of operation being performed. The processor automatically chooses a segment according to the rules given in Table 3-2.

Table 3-2. Default Segment Selection Rules

| Type of Reference | Register Used | Segment Used | Default Selection Rule |
|---|---|---|---|
| Instructions | CS | Code Segment | All instruction fetches. |

Table 3-2. Default Segment Selection Rules

| | | | |
|---|---|---|---|
| Stack | SS | Stack Segment | All stack pushes and pops.<br>Any memory reference which uses the ESP or EBP register as a base register. |
| Local Data | DS | Data Segment | All data references, except when relative to stack or string destination. |
| Destination Strings | ES | Data Segment pointed to with the ES register | Destination of string instructions. |

When storing data in or loading data from memory, the DS segment default can be overridden to allow other segments to be accessed. Within an assembler, the segment override is generally handled with a colon ":" operator. For example, the following MOV instruction moves a value from register EAX into the segment pointed to by the ES register. The offset into the segment is contained in the EBX register:

```
MOV ES:[EBX], EAX;
```

(At the machine level, a segment override is specified with a segment-override prefix, which is a byte placed at the beginning of an instruction.) The following default segment selections cannot be overridden:

- Instruction fetches must be made from the code segment.
- Destination strings in string instructions must be stored in the data segment pointed to by the ES register.
- Push and pop operations must always reference the SS segment.

Some instructions require a segment selector to be specified explicitly. In these cases, the 16-bit segment selector can be located in a memory location or in a 16-bit register. For example, the following MOV instruction moves a segment selector located in register BX into segment register DS:

```
MOV DS, BX
```

Segment selectors can also be specified explicitly as part of a 48-bit far pointer in memory. Here, the first doubleword in memory contains the offset and the next word contains the segment selector.

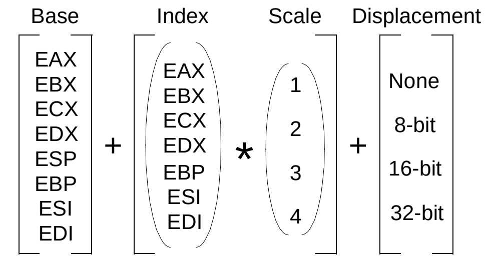
### 3.7.3.2. SPECIFYING AN OFFSET

The offset part of a memory address can be specified either directly as an static value (called a displacement ) or through an address computation made up of one or more of the following components:

- Displacement—An 8-, 16-, or 32-bit value.
- Base—The value in a general-purpose register.
- Index—The value in a general-purpose register.

- Scale factor—A value of 2, 4, or 8 that is multiplied by the index value.

The offset which results from adding these components is called an effective address . Each of these components can have either a positive or negative (2s complement) value, with the exception of the scaling factor. Figure 3-9 shows all the possible ways that these components can be combined to create an effective address in the selected segment.



Offset = Base + (Index ∗ Scale) + Displacement

Figure 3-9. Offset (or Effective Address) Computation

The uses of general-purpose registers as base or index components are restricted in the following manner:

- The ESP register cannot be used as an index register.
- When the ESP or EBP register is used as the base, the SS segment is the default segment. In all other cases, the DS segment is the default segment.

The base, index, and displacement components can be used in any combination, and any of these components can be null. A scale factor may be used only when an index also is used. Each possible combination is useful for data structures commonly used by programmers in high-level languages and assembly language. The following addressing modes suggest uses for common combinations of address components.

### Displacement

A displacement alone represents a direct (uncomputed) offset to the operand. Because the displacement is encoded in the instruction, this form of an address is sometimes called an absolute or static address. It is commonly used to access a statically allocated scalar operand.

### Base

A base alone represents an indirect offset to the operand. Since the value in the base register can change, it can be used for dynamic storage of variables and data structures.

### Base + Displacement

A base register and a displacement can be used together for two distinct purposes:

- As an index into an array when the element size is not 2, 4, or 8 bytes—The displacement component encodes the static offset to the beginning of the array. The base register holds the results of a calculation to determine the offset to a specific element within the array.
- To access a field of a record—The base register holds the address of the beginning of the record, while the displacement is an static offset to the field.

An important special case of this combination is access to parameters in a procedure activation record. A procedure activation record is the stack frame created when a procedure is entered. Here, the EBP register is the best choice for the base register, because it automatically selects the stack segment. This is a compact encoding for this common function.

### (Index ∗ Scale) + Displacement

This address mode offers an efficient way to index into a static array when the element size is 2, 4, or 8 bytes. The displacement locates the beginning of the array, the index register holds the subscript of the desired array element, and the processor automatically converts the subscript into an index by applying the scaling factor.

### Base + Index + Displacement

Using two registers together supports either a two-dimensional array (the displacement holds the address of the beginning of the array) or one of several instances of an array of records (the displacement is an offset to a field within the record).

### Base + (Index ∗ Scale) + Displacement

Using all the addressing components together allows efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size.

### 3.7.3.3. ASSEMBLER AND COMPILER ADDRESSING MODES

At the machine-code level, the selected combination of displacement, base register, index register, and scale factor is encoded in an instruction. All assemblers permit a programmer to use any of the allowable combinations of these addressing components to address operands. High-level language compilers will select an appropriate combination of these components based on the language construct a programmer defines.

## 3.7.4. I/O Port Addressing

The processor supports an I/O address space that contains up to 65,536 8-bit I/O ports. Ports that are 16-bit and 32-bit may also be defined in the I/O address space. An I/O port can be addressed with either an immediate operand or a value in the DX register. See Chapter 12, Input/Output, for more information about I/O port addressing.