EMD™ 64 and EMD-32 Architectures  
Software Developer’s Manual

Combined Volumes:  
1, 2 and 3

Note: This document contains all three volumes of the EMD-64 and EMD-32 Architectures Software Developer’s Manual: Basic Architecture; Instruction Set Reference A-Z; System Programming Guide. Refer to all three volumes when evaluating your design needs.

CONTENTS

PAGE

Volume 1:

[Chapter 1 ABOUT THIS MANUAL 4](#_Toc69292864)

[1.1 EMD™ 64 AND EMD-32 PROCESSORS COVERED IN THIS MANUAL 4](#_Toc69292865)

[1.2 OVERVIEW OF VOLUME 1: BASIC ARCHITECTURE 4](#_Toc69292866)

[1.3 NOTATIONAL CONVENTIONS 6](#_Toc69292867)

[1.3.1 Bit and Byte Order 6](#_Toc69292868)

[1.3.2 Reserved Bits and Software Compatibility 6](#_Toc69292869)

[1.3.3 Hexadecimal and Binary Numbers 6](#_Toc69292870)

[1.3.4 Segmented Addressing 6](#_Toc69292871)

[1.3.5 A New Syntax for CPUID, CR, and MSR Values 7](#_Toc69292872)

[1.3.6 Exceptions 7](#_Toc69292873)

[1.4 RELATED LITERATURE 7](#_Toc69292874)

[Chapter 2 EMD™ 64 AND EMD-32 ARCHITECTURES 9](#_Toc69292875)

[2.1 BRIEF HISTORY IF EMD™ 64 AND END-32 ARCHITECTURE 9](#_Toc69292876)

[2.1.1 8-bit Processors (2020) 9](#_Toc69292877)

[2.1.2 EMP-32 Series (2021) 9](#_Toc69292878)

[2.1.3 EMP-64 Series (2021) 9](#_Toc69292879)

[2.2 MORE ON SPECIFIC ADVANCES 10](#_Toc69292880)

[2.2.1 Nullable Types 10](#_Toc69292881)

[2.2.2 SIMD Instructions 10](#_Toc69292882)

[2.2.3 Multi-Core Technology 10](#_Toc69292883)

[2.2.4 EMD™ Virtualization Technology 10](#_Toc69292884)

[2.3 EMD™ 64 and EMD-32 PROCESSOR GENERATIONS 10](#_Toc69292885)

[Chapter 3 BASIC EXECUTION ENVIRONMENT 12](#_Toc69292886)

[3.1 MODES OF OPERATION 12](#_Toc69292887)

[3.2 OVERVIEW OF THE BASIC EXECUTION ENVIRONMENT 13](#_Toc69292888)

[3.3 MEMORY ORGANIZATION 14](#_Toc69292889)

[3.3.1 Memory Models 14](#_Toc69292890)

[3.3.2 Paging and Virtual Memory 15](#_Toc69292891)

[3.3.3 32-Bit and 16-Bit Address and Operand Sizes 15](#_Toc69292892)

[3.4 BASIC PROGRAM EXECUTION REGISTERS 16](#_Toc69292893)

[3.4.1 General-Purpose Registers 16](#_Toc69292894)

[3.4.2 Segment Registers 17](#_Toc69292895)

[3.4.3 EFLAGS Register 17](#_Toc69292896)

[3.5 INSTRUCTION POINTER 20](#_Toc69292897)

[3.6 OPERAND-SIZE AND ADDRESS-SIZE ATTRIBUTES 21](#_Toc69292898)

[3.6.1 Operand Size and Address Size in 64-bit Mode 21](#_Toc69292899)

[3.7 OPERAND ADDRESSING 22](#_Toc69292900)

[3.7.1 Immediate Operands 22](#_Toc69292901)

[3.7.2 Register Operands 23](#_Toc69292902)

[3.7.3 Memory Operands 24](#_Toc69292903)

[3.7.4 Specifying a Segment Selector 24](#_Toc69292904)

[3.7.5 Specifying an Offset 26](#_Toc69292905)

[3.7.6 Assembler and Compiler Addressing Modes 27](#_Toc69292906)

[3.7.7 I/O Port Addressing 28](#_Toc69292907)

[Chapter 4 DATA TYPES 29](#_Toc69292908)

[4.1 FUNDAMENTAL DATA TYPES 29](#_Toc69292909)

[4.1.1 Alignment of Words, Doublewords, Quadwords, and Double Quadwords 29](#_Toc69292910)

[4.2 NUMERIC DATA TYPES 29](#_Toc69292911)

[4.2.1 Integers 29](#_Toc69292912)

# ABOUT THIS MANUAL

The EMD™ 64 and EMD-32 Architectures Software Developer’s Manual, Volume 1: Basic Architecture is part of a set that describes the architecture and programming environment if EMD™ 64 and EMD-32 architecture processors. Other volumes in this set are:

* The EMD™ 64 and EMD-32 Architectures Software Developer’s Manual, Volume 2: Instruction Set Reference
* The EMD™ 64 and EMD-32 Architectures Software Developer’s Manual, Volume 3: System Programming Guide

The EMD™ 64 and EMD-32 Architectures Software Developer’s Manual, Volume 1, describes the basic architecture and programming environment of EMD-64 and EMD-32 processors. The EMD™ 64 and EMD-32 Architectures Software Developer’s Manual, Volume 2, describes the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The EMD™ 64 and EMD-32 Architectures Software Developer’s Manual, Volume 3, describes the operating-system support environment of EMD-64 and EMD-32 processors. These volumes target operating-systems and BIOS designers. The EMD 8-bit Series does not have these features and mentions will be limited.

## EMD™ 64 AND EMD-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent EMD-64 and EMD-32 processors, and the 8-bit series. All included processors are:

* EMD™ EMP-32 Processors
* EMD™ EMP-64 Processors
* EMD™ Mini-8 Processors

EMP-32 family processors are EMD-32 processors which include the EMP-32-1000 and EMP-32-2000 models.

EMP-64 family processors are EMD-64 processors which include the EMP-64-1000 and EMP-64-1000 models.

Mini-8 family processors are smaller processors not including in the EMP series. These include the original Mini-8, and the smaller Mini-8: Micro.

## OVERVIEW OF VOLUME 1: BASIC ARCHITECTURE

A description of this manual’s content follows:

Chapter 1 — About This Manual. Gives an overview of all three volumes of the EMD™ 64 and EMD-32 Architectures Software Developer’s Manual. It also describes the notational conventions in these manuals and lists related EMD manuals and documentation of interest to programmers and hardware designers.

Chapter 2 — EMD™ 64 and EMD-32 Architectures. Introduces the EMD-64 and EMD-32 architectures along with the families of EMD processors that are based on these architectures. It also gives an overview of the common features found in these processors and brief history of the EMD-64 and EMD-32 architectures.

Chapter 3 — Basic Execution Environment. Introduces the models of memory organization and describes the register set used by applications.

Chapter 4 — Data Types. Describes the data types and addressing modes recognized by the processor; provides an overview of real numbers and floating-point formats and of floating-point exceptions.

Chapter 5 — Instruction Set Summary. Lists all EMD-64 and EMD-32 instructions (and the small Mini-8 instruction sets), divided into technology groups.

Chapter 6 — Procedure Calls, Interrupts, and Exceptions. Describes the procedure stack and mechanisms provide for making procedure calls and for servicing interrupts and exceptions.

Chapter 7 — Managing State Using the XSAVE Feature Set. Describes the XSAVE feature set instructions and explains how software can enable the XSAVE feature set and XSAVE-enabled features.

Chapter 8 — Input/Output. Describes the processor’s I/O mechanism, including I/O port addressing, I/O instructions, and I/O protection mechanisms.

Chapter 9 — Processor Identification and Feature Determination. Describes how to determine the CPU type and features available in the processor.

Chapter 10 — EFLAGS Condition Codes and Cross-Reference. Summarizes how the EMD-32 (and EMD-64) instructions affect the flags in the EFLAGS register. Also summarizes how condition jump, move, and ‘byte set on condition code’ instructions use condition code flags (OF, CF, ZF, SF, and PF) in the EFLAGS register.

Chapter 11 — Exception Handlers. Summarizes exceptions raised by floating-point and SSE floating-point instructions. Also describes how to design and write exception handing facilities for FPU exceptions. This also describes general techniques for writing robust FPU exception handlers. Also gives guidelines for writing exception handlers for exceptions generated by SSE floating-point instructions (EMD-64 Only).

## NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. This notation is described below.

### Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power if the bit position. EMD-64 and EMD-32 processors are either big or little endianness but are by default “little endian” machines; this means the bytes of a word are numbered starting from the most significant byte.

### Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as reserved. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable.

Software Should follow these guidelines in dealing with reserved bits:

* Do not depend on the states of any reserved bits when evaluating the values of registers that contain such bits. Mask out the reserved bits before testing.
* Do not depend on the states of any reserved bits when storing to memory or to a register.
* Do not depend on the ability to retain information written into any reserved bits.
* When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with the values previously read from the same register.

### Hexadecimal and Binary Numbers

Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H or starting with the 0x prefix. (for example, 0F82EH, 0x0F82E). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B, or starting with the 0b prefix (for example, 1010B, 0b1010). The “B” designation is only used in situations where confusion as to the type of a number might arise.

### Segmented Addressing

The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes of memory. The range of memory that can be addressed is called an address space.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called segments. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

Segment-register:Byte-address

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

DS:FF79H

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and EIP register contains the address of the instruction.

CS:EIP

### A New Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific register. The CPUID reads the EAX register as a parameter. Then results are placed in the CR (Control Registers). Some MSR registers may also be used.

### Exceptions

An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

#PF(fault code)

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions that produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

#GP(0)

## RELATED LITERATURE

Literature related to EMD-64 and EMD-32 processors is listed.

* The specification update for EMD-64 or EMD-32 processors
* EMD™ C Compiler documentation:

<http://example.com/>

* EMD™ Fortran Compiler documentation

<http://example.com/>

* EMD™ 32-64-bit Assembler documentation

<http://example.com/>

* EMD™ Software Developer Tools

<http://example.com/>

# EMD™ 64 AND EMD-32 ARCHITECTURES

## BRIEF HISTORY IF EMD™ 64 AND END-32 ARCHITECTURE

The following sections provide a summary of the major technical evolutions from the 8-bit processors to the EMD-64 architecture: starting from the original Mini-8 processor to the latest EMD™ EMP-64 processor 2000 series.

### 8-bit Processors (2020)

The 32-bit architecture family was preceded by 8-bit processors, the MOS-1, MOS-2, and the later Mini-8 and Mini-8-Micro. The MOS-1 has an 8-bit external data bus, with 8-bit addressing giving 255-B address space. The MOS-1 had a small 4-bit instruction limit, giving it only 16 instructions.

The MOS-2 tried weird things, including 19-bit general purpose registers, and 32-bit external data bus, and 24-bit addressing giving 16-GByte address space. The MOS-2 also increased to a 6-bit instruction limit, giving it a slightly larger 64 instructions. This is not truly an 8-bit processor but predated any modern EMP-32 architectures.

The more modern Mini-8 includes full 8-bit capabilities including traditional data-width, 8 general purpose registers, and 16-bit addressing giving it 64-KByte address space.

The smaller scale Mini-8-Micro has 8-bit capabilities, and includes a nibble flags register, only including CF, ZF, SF, and reserved. The main difference is the 256-instruction set and followed 6502 standards.

There was also a small 4-bit processor project, but it was short lived so there is not another section for it.

### EMP-32 Series (2021)

There were no 16-bit processors, but a jump to full 32-bit support. The first was the EMP-32-1000 model, with 5 versions going up to EMP-32-1005. The line was discontinued as the 32-bit market was irrelevant. This period was where many changes to design happened, implementing standards starting with the EMP-32-1005 variant.

### EMP-64 Series (2021)

The upgrade to 64-bit was where most of the standards came into play. The first was the EMP-64-1000, which included a much larger instruction set with more modern capabilities including segmentation, and descriptor tables. This is also where the exception and base interrupt standards were implemented.

The EMP-64-2000 is the next model in the line, with even more features including full floating-point support with SSE registers, and larger 128, 256, and 512 registers.

The EMP-64-3000 is currently in development and being used in HEMP. Further details and design choices are listed in this documentation.

## MORE ON SPECIFIC ADVANCES

### Nullable Types

The MOS processors included many experimental ideas, including datatypes on the hardware level, including nullable types. This idea of nullable types would let the computer more accurately generate answers to complex problems, where not all the parameters are known, for example a null value, or uninitialized value will reserve the space in memory, but either not change the memory, or reset the memory. This can result in weird results and puts complexity on the software side. This was not used to strive towards a more standard approach, but this line of processors may start up again to further develop these ideas.

### SIMD Instructions

Beginning with the EMP-64-1000, processors have had SIMD integration including MMX 64-bit registers, XMM 128-bit registers, YMM 256-bit registers, and ZMM 512-bit registers. Many new instructions have been added for simple math to prevent adding new addressing complexity.

### Multi-Core Technology

The EMP-64-3000 has implemented a new shared RAM memory model to allow for multi-code technology. This can speed up processing speed at the loss of simplicity. To support backwards compatibility, model specific registers can now change details about multi-core assignments and threads.

### EMD™ Virtualization Technology

With the addition of threading, virtualization was added. EMD™ Virtualization Technology for EMD-64 architectures provide extensions that support virtualization. The extensions are referred to as Virtual Machine Extensions (VMX). An EMD-64 platform with VMX can function as multiple virtual systems (or virtual machines). Each virtual machine can run operating systems and applications in separate partitions.

## EMD™ 64 and EMD-32 PROCESSOR GENERATIONS

The computing power and the complexity if EMD architecture processors has grown since the beginning. Significant processors are shown below with their respective specs, to show this growth.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| EMD Processor | Date Introduced | Top-Bin Clock Frequency at introduction | Register Sizes | System Bus Bandwidth | Max. External Address Space |
| MOS-1 | 2020 | Unknown | N/A | Unknown | 255 B |
| MOS-2 | 2020 | Unknown | GP: 19 | Unknown | 16 MB |
| EMD Mini-8 | 2020 | 1 Hz | GP: 8 | 1 Hz | 255 B |
| EMD Mini-8-Micro | 2020 | 10 Hz | GP: 8 | 10 Hz | 255 B |
| EMD EMP-32-1000 | 2020 | 1 kHz | GP: 32 | 500 Hz | 4 GB |
| EMD EMP-32-1001 | 2020 | ~1 kHz | GP: 32 | 500 Hz | 4 GB |
| EMD EMP-32-1002 | 2021 | ~1 kHz | GP: 32 | 500 Hz | 4 GB |
| EMD EMP-32-1003 | 2021 | ~1 kHz | GP: 32 | 600 Hz | 4 GB |
| EMD EMP-32-1004 | 2021 | ~1 kHz | GP: 32  FPU: 32 | 600 Hz | 4 GB |
| EMD EMP-32-1005 | 2021 | ~1 kHz | GP: 32  FPU: 32 | 600 Hz | 4 GB |
| EMD EMP-64-1000 | 2021 | ~1 kHz | GP: 64 FPU: 64 MMX: 64 XMM: 128 | 600 Hz | 4 GB |
| EMD EMP-64-2000 | 2021 | ~1 kHz | GP: 64 FPU: 80 MMX: 64 XMM: 128 | 600 Hz | 1 TB |
| EMD EMP-64-3000 | 2021 | 400 kHz | GP: 64 FPU: 80 MMX: 64 XMM: 128 YMM: 256 ZMM: 512 | ~240 kHz | 4 GB |

# BASIC EXECUTION ENVIRONMENT

This chapter describes the basic execution environment of an EMD-64 or EMD-32 processor as seen by assembly-language programmers. It describes how the processor executes instructions and how it stores and manipulates data. The execution environment described here includes memory (the address space), general-purpose data registers, segment registers, the flag register, and the instruction pointer register.

## MODES OF OPERATION

The EMD-32 architecture only supported one mode, as protection was not a concern in mind. The EMD-64 architecture modes are as follows:

* Protected mode — This mode is the native state of the processor. Among the capabilities of protected mode is the ability to directly execute “real-address mode” software in a protected, multi-tasking environment. This feature is called virtual-64 mode, although it is not actually a processor mode. Virtual-64 mode is a protected mode attribute that can be enabled for any task.
* Real-address mode — This mode implements the programming environment of the EMD-64 processor with 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 (SMM) — 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 (SMI #) is called 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 basic 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 the system management interrupt. SMM was introduced with the EMD EMP-64-3000 processor, and because a standard to all processor families.

* 64-bit mode — This mode enables a 64-bit operating system to run applications written to access 64-bit linear address space.

64-bit mode extends the number of general-purpose registers and SIMD extension registers from 8 to 16. General purpose registers are widened to 64 bits. The mode also introduces a new opcode size to access the register extensions.

64-bit mode is enabled by the operating system on a code-segment basis.

## OVERVIEW OF THE BASIC EXECUTION ENVIRONMENT

Any program or task running on an EMD-64 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) make up the basic execution environment for an EMD-64 processor.

* Address space — A task or program running in 64-bit mode can address linear address space of up to 264 bytes (allocated dynamically starting at 232 bytes). Software can query CPUID for the physical address size supported by a processor.
* Basic program execution registers — The number of general-purpose registers (GPRs) available is 16. GPRs are 64-bits wide, and they support operations on byte, word, doubleword and quadword integers. Accessing byte registers is done uniformly to the lowest 8 bits. The instruction pointer register becomes 64-bits. The EFLAGS register is extended to 64-bits wide and is referred to as the RFLAGS register. The upper 32-bits of RFLAGS is reserved. The lower 32 bits of RFLAGS is the same as EFLAGS.
* FPU registers — The eight FPU data registers, the FPU control register, and status register, the FPU instruction pointer register, and FPU operand (data) pointer register, the FPU tag register, and FPU opcode register provide an execution environment for operating on single-precision, double-precision, and double-extended-precision floating-point values, word integers, doubleword integers, quadword integers, and binary coded decimal (BCD) values.
* MMX, XMM, YMM, ZMM registers — ZMM is now 512-bits with MMX, XMM, and YMM addressing the bottom quadword, double quadword, and quad quadword of ZMM.
* Stack — To stack pointer size is 64-bits. Stack size is not controlled by a bit in the SS descriptor, nor can an instruction size override the pointer size. To support procedure or subroutine calls and the passing of parameters between procedures or subroutines, a stack a stack management resources are included in the execution environment. The stack is in memory.
* I/O ports — The EMD-64 architecture supports a transfer of data to and from input/output (I/O) ports.
* Control register — The six viewable control registers (CR0 through CR4, and CR8/TPR) determine the operating mode of the processor and the characteristics of the currently executing task. These are 64-bit registers.
* Memory management register — The GDTR, IDTR, task register, and LDTR specify the locations of data structures used in protected mode memory management.
* Debug registers — The debug registers (DR0 through DR7) control and allow monitoring of the processor’s debugging operations. These are 64-bit registers.
* Memory type range registers (MTRRs) — The MTRRs are used to assign memory types to regions of memory.
* Machine specific registers (MSRs) — The processor provides a variety of machine specific registers that are used to control and report on processor performance. Virtually all MSRs handle system related functions and are not accessible to an application program. One exception to this rule is the time-stamp counter.
* Machine check registers — The machine check registers consist of a set of control, status, and error-reporting MSRs that are used to detect and report on hardware (machine) errors.
* Performance monitoring counters — The performance monitoring counters allow processor performance events to be monitored.

## 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 236-1 (64 GB) if the processor does not support EMD-64 architecture. EMD-64 architecture introduces a change in physical and linear address space.

Virtually any operating system or executive designed to work with an EMD-32 or EMD-64 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 Models

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

* Flat memory model — Memory appears to a program as a single, continuous address space. This space is called a linear address space. Code, data, and stacks are all contained in this address space. Linear address space is byte addressable, with addresses running contiguously from 0 to 232-1 (if not in 64-bit mode). An address for any byte in linear address space is called linear address.
* Segmented memory model — Memory appears to a program as a group of independent address spaces called segments. Code, data, and stacks are typically contained in separate segments. To address a byte in a segment, a program issues a logical address. This consists of a segment selector and an offset (logical addresses are often referred to as far pointers). The segment selector identifies the segment to be accessed and the offset identifies a byte in the address space of the segment. Programs running on EMD-32 processors can address up to 4 segments of varied sizes and types, and each segment can be as large as 232 bytes. In 64-bit mode this increases to 16,383 segments.

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 of data, respectively.

* Read-address mode memory model —This is the memory model for the EMD 8-bit processors, and default to all processors. It is supported to provide compatibility with existing programs written to run on the older processors. 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 apace in real-address mode is 220 bytes.

### Paging and Virtual Memory

With the flat or the segmented memory model, 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. Linear addresses are sent out on the processor’s address lines without translation.

When using the EMD-64 architecture’s paging mechanism (paging enables), linear address space is divided into pages which are mapped to 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. All that the application sees are linear address space.

### 32-Bit and 16-Bit Address and Operand Sizes

EMD processors in protected mode can be configured for 32-bit or 16-bit address and operand sizes. With 32-bit addresses and operand sizes, the maximum linear address or segment offset is FFFFFFFFH (232-1); 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 (216-1); 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, an address consists of a 16-bit segment selector and a 16-bit offset.

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 cod 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 allow 30bi linear address is still 000FFFFFH (220-1).

## BASIC PROGRAM EXECUTION REGISTERS

EMD-32 architecture provides 16 basic program execution registers for use in general system and application programming. 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 size segment selectors.
* ELFAGS (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-but pointer to the next instruction to be executed.

### General-Purpose Registers

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

* Operands for logical and arithmetic operations
* Operands for address calculations
* Memory pointers

Although all 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 rule should not be used for another purpose.

Many instructions assign specific registers to hold operands. For example, string instructions use the contents of the ECX, ESI, and EDI registers and 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 BX register points to a memory location int eh DS segment.

The following is a summary of general-purpose special uses by instructions:

* 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)

#### General-Purpose Registers in 64-Bit Mode

In 64-bit mode, there are 16 general purpose registers, and the default operand size is 32-bits. However, general-purpose registers can work with either 32-bit or 64-bit operands. If a 32-bit operand size is specified: EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D – R15D are available. If a 64-bit operand size is specified: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8 – R15 are available. All these registers can be accessed at the byte, word, dword, or qword level.

### Segment Registers

The segment registers (CS, DS, SS, ES, FS, and GS) gold 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.

The CS register contains the segment selector for the code segment, where the instructions being executed are stored. The DS, ES, FS, and GS registers point to four data segments. The availability of four data segments permits efficient and secure access to distinct types of data structures. The SS register contains the segment selector for the 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. The main difference is it can be loaded explicitly, which permits application programs to set up multiple stacks and switch among them.

In 64-bit mode each segment is treated as 0, creating a flat address space. This is to encourage paging which is more modern and a better memory model.

### EFLAGS Register

The 32-bit EFLAGS register contains a group of status flags, a control flag, and a group of system flags. Following initialization of the processor (eight by asserting a RESET or INIT interrupt), the state of the EFLAGS register is 00000002H. Bits 1, 3, 5, 15, and 22 through 21 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. The following instructions can be used to mode groups of flags to and from the procedure stack, or the EAX register: LAHF, SAHF, PUSHF, PUSHFD, POPF, 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 blinding itself to a new task, the processor loads the ELFLAGS 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 sate of the ELFAGS register on the procedure stack. When an interrupt or exception is managed with a task switch, the state of the EFLAGS register is saved in the TSS for the task being suspended.

#### 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 status flag functions are:

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 using 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) Auxiliary Carry 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 onto 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 flags 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 singed- 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 (condition move) use one or more of the status flags as condition codes and evaluate them for branch, set-byte, or end-loop conditions.

#### DF Flag

The direction flag (DF, located in bit 10 of the ELFAGS register) controls string instructions (MOVS, CMPS, SCAS, LODS, and STOS). Setting the DF flag causes the string instructions to auto-decrement (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.

#### 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 follow:

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

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.

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. The POPF and IRET instructions can modify this field only 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-64 mode flag — Set to enable virtual-64 mode; clear to return to protected mode without virtual-64 mode semantics.

AC (bit 18) Alignment check (or access control) flag — If the AM bit is set in the CR0 register, alignment checking of user-mode data access is enabled if and only if this flag is 1.

If the SMAP bit is set in the CR4 register, explicit supervisor-mode data accesses to user-mode pages are allowed if and only if this bit is 1.

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.

#### RFLAGS Register in 64-bit Mode

In 64-bit mode, EFLAGS is extended to 64 bits and called RFLAGS. The upper 32 bits of RFLAGS register is reserved. The lower 32 bits of RFLAGS is the same as EFLAGS.

## 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 several 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).

All EMD-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 the EIP register to direct program flow remains fully compatible with all software written to run on EMD-32 processors.

#### Instruction Pointer in 64-bit Mode

In 64-bit mode, the RIP register becomes the instruction pointer. This register holds the 64-bit offset of the next instruction to be executed. 64-bit mode also supports a technique called RIP-relative addressing. Using this technique, the effective address is determined by adding a displacement to the RIP of the next instruction.

## 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. 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, virutla-64 mode, or SMM, the default operand-size and address-size attributes are always 16 bits.

The operand-seize attributes select the size of operands. When the 16-bit operand-size attribute is in force, operations can be either 8 bits or 16 bits, and when the 32-bit operand-size attribute is in force, operands can 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 is in force, segment offsets and displacements are 16 bits. This restriction limits the size of a segment to 64 Kbytes. When the 32-bit address-size attribute is in force, segment offsets and displacement are 32 bits, allowing 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. The effect of this prefix applies only to the targeted instruction.

### Operand Size and Address Size in 64-bit Mode

In 64-bit mode, the default address-size is 64 bits, and the default operand size is 32 bits. Defaults can be overridden using prefixes. Address-size and operand-size prefixes allow mixing of 32/64-bit data and 32/64-bit addresses on an instruction-by-instruction basis. The following table shows valid combinations of the 66H instruction prefix and the REX.W prefix that may be used to specify operand-size overrides in 64-bit mode. Note that 16-bit addresses are not supported in 64-bit mode.

REX prefixes consist of 4-bit fields that form 16 different values. The W-bit field in the REX refixes is referred to as REX.W. If the REX.W field is properly set, the prefix specifies an operand size override to 64 bits. Note that software can stull use the operand-size 66H prefix to toggle to a 16-bit operand size. However, setting REX.W takes precedence over the operand-size prefix (66H) when both are used.

In the case of SSE/SSE2/SSE3/SSSE3 SIMD instructions: the 66H, F2H, and F3H prefixes are mandatory for opcode extensions. In such a case, there is no interaction between a valid RE.W prefix and a 66H opcode extension prefix.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| L Flag in Code Segment Descriptor | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| REX.W Prefix | 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 |
| Effective Operand Size | 32 | 32 | 16 | 16 | 64 | 64 | 64 | 64 |
| Effective Address Size | 64 | 32 | 64 | 32 | 64 | 32 | 64 | 32 |

Notes:

Y: Yes – this instruction prefix is present.  
N: No – this instruction prefix is not present.

## OPERAND ADDRESSING

EMD-32 machine-instructions act on zero or more operands. Some operands are specified explicitly, and others are implicit. The data for a source operand can be in:

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

When an instruction returns data to a destination operand, it can be returned to:

* A register
* A memory location
* An I/O port

### 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 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 (232).

### Register Operands

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

* 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP)
* 16-bit general-purpose registers (AX, BX, CX, DX, SI, DI, SP, or BP)
* 8-bit general-purpose register (AH, BH, CH, DH, AL, BL, CL, or DL)
* Segment registers (CS, DS, SS, ES, FS, and GS)
* EFLAGS register
* FPU registers (ST0 through ST7, status word, control word, tag word, data operand pointer, and instruction pointer)
* MMX registers (MM0 through MM7)
* XMM register (XMM0 through XMM7) and the MXCSR register
* Control registers (CR0, CR2, CR3, and CR4) and system table pointer registers (GDTR, LDTR, IDTR, and task register)
* Debug registers (DR0, DR1, DR2, DR3, DR6, and DR7)
* MSR registers

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 sate 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 an implied operand of the instruction.

#### Register Operands in 64-Bit Mode

Register operands in 64-bit mode can be any of the following:

* 64-bit general-purpose registers (RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP or R8-R15)
* 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, or R8D-R15D)
* 16-bit general-purpose registers (AX, BX, CX, DX, SI, DI, SP, BP, or R8W-R15W)
* 8-bit general-purpose registers: AL, BL, CL, DL, SIL, DIL, SPL, BPL, and R8L-R15L are available using REX prefixes; AL, BL, CL, DL, AH, BH, CH, DH are available without using REX prefixes.
* Segment registers (CS, DS, SS, ES, FS, and GS)
* RFLAGS register
* FPU registers (ST0 through ST7, status word, control word, tag word, data operand pointer, and instruction pointer)
* MMX registers (MM0 through MM7)
* XMM registers (XMM0 through XMM15) and the MXCSR register
* YMM registers (YMM0 through YMM15)
* ZMM registers (ZMM0 through ZMM15)
* Control registers (CR0, CR2, CR3, CR4, and CR8) and system table pointer registers (GDTR, LDTR, IDTR, and task register)
* Debug registers (DR0, DR1, DR2, DR3, DR6, and DR7)
* MSR registers
* RDX:RAX register pair representing a 128-bit operand

### Memory Operands

Source and destination operands in memory are refenced by means of segment selector and an offset. Segment selectors specify the segment containing the operand. Offset specify the linear of effect address of the operand. Offsets can be 32 bits (represented by the notation m16:32) or 16 bits (represented by the notation m16:16).

#### Memory Operands in 64-Bit Mode

In 64-bit mode, a memory operand can be referenced by a segment selector and an offset. The offset can be 16 bits, 32 bits, or 64 bits.

### 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 the following table.

When storing data in memory 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 managed with a colon “:” operator. For example, the following MOV instruction moves a value from register EAX into the segment pointer to by the ES register. The offset into the segment is contained in the EBX registers.

MOV ES:[EBX], EAX;

|  |  |  |  |
| --- | --- | --- | --- |
| Reference Type | Register Used | Segment Used | Default Selection Rule |
| Instructions | CS | Code Segment | All instruction fetches. |
| Stack | SS | Stack Segment | All stack pushes and pops.  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 if string instructions. |

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 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 word contains the segment selector and the next doubleword in memory contains the offset.

#### Segmentation in 64-Bit Mode

In 640bit mode, segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. The processor treats the segment base of CS, DS, ES, and SS as zero, creating a linear address that is equal to the effective address. The exceptions are the FS and GS segments, whose segment registers (which hold the segment base) can be used as additional base registers in some linear address calculations.

### Specifying an Offset

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

* Displacement — Ab 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, except for the scaling factor.

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 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, the form of address is sometimes called an absolute or statis 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 golds 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 golds the address of the beginning of the record, while the displacement is a 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 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.

#### Specifying an Offset in 64-Bit Mode

The offset part of a memory address in 64-bit mode can be specified directly as a static value or through an address computation made up of one or more of the following components:

* Displacement — An 8-bit, 16-bit, or 32-bit value.
* Base — The value in a 64-bit general-purpose register.
* Index — The value in a 64-bit general-purpose register.
* Scale factor — A value of 2, 4, or 8 that is multiplied by the index value.

The base and index value can be specified in one of sixteen available general-purpose registers in most cases. The following unique combination of address components is also available.

* RIP + Displacement — In 64-bit mode, RIP-relative addressing uses a signed 32-bit displacement to calculate the effective address of the next instruction by sign-extending the 32-bit value and add to the 64-bit value in RIP.

### Assembler and Compiler Addressing Modes

At the machine-code level, the selected combination of displacement, vase register, index register, and scale factor are 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.

### 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.

# DATA TYPES

This chapter introduces data types defined for the EMD-64 and EMD-32 architectures. A section at the end of this chapter describes the real-number and floating-point concepts used in FPU, SEE, SSE2, SSE3, SSSE3, and SSE4 extensions.

## FUNDAMENTAL DATA TYPES

The fundamental data types are bytes, words, doublewords, quadwords, and double quadwords. A byte is eight bits, a word is 2 bytes (16 bits), a doubleword is 4 bytes (32 bits), a quadword is 8 bytes (64 bits), and a double quadword is 16 bytes (128 bits). There are also YMM and ZMM extensions that add a quad quadword (32 bytes, 256 bits), and a double quad quadword (64 bytes, 512 bits).

The quadword data type was introduced into the EMD-64 architecture; the double quadword data type was introduced in the EMP-64-2000 processor, and the YMM and ZMM sizes were added in the EMP-64-3000 processor.

### Alignment of Words, Doublewords, Quadwords, and Double Quadwords

To reduce memory usage, the EMP-64-3000 is the first processor to remove the need for alignment for all datatypes.

## NUMERIC DATA TYPES

Although bytes, words, and double words are fundamental data types, some instructions support additional interpretations of these data types to allow operations to be performed on numeric data types (singed and unsigned integers, and floating-point numbers). Single-precision (32-bit) floating-point and double-precision (64-bit) floating-point data types are supported across all generations of SSE extensions. Half-precision (16-bit) floating-point data type was removed with the EMP-32-1005.

### Integers