Intel 64 and IA-32 Architectures: Basic Execution Environment

Diego Linares

How the processor executes instructions aand how it stores and manipulates data. Includes: memory, general-purpose data, segment, flag and intruction pointer register.

I. Modes of operation

- **Protected:** native state. Directly execute real address software in a virtual-8086 environment, which can be enabled for any task.
- **Real-adress:** implements the environment of the Intel 8086 processor with extensions. Gets this mode after a reset.
- **System management:** transparent mechanism for implementing platform-specific functions. Enter this mode through an SMM interrupt pin. Saves the context of a task, but switches to a special adress space.

A. *Intel 64 Architecture*

Adds IA-32e mode with 2 submodes:

- **Compatibility:** permits to run 16-bit and 32-bit programs without recompilation on a 64-bit system. Supports all privilege levels of the 64-bit protected mode. However, virtual-8086 mode won't work. It's enabled in a code-segment basis (it can support both 64-bit and 32-bit apps at the same time). Similar to protected mode.
- **64-bit:** allows to use 64-bit linear address space. 8 more general purpose (which are widened too) and SIMD registers. New opcode prefix, which allows to promote instructions to work in 64-bit mode.

II. Overview of the Basic Execution Environment

Resources given to each program or task. 64-bit supports the basic BEE of an IA-32 processor.

- **Address space:** in IA-32, you have a linear space of 2^{32} bytes and a physical of 2^{36}.
- **Basic program execution registers:** divided in 8 GP, 6 segment, 1 EFLAGS and 1 EIP register, in which instructions are done.
- **x87 FPU registers:** 8 of data, 1 of control, status, instruction pointer, operand pointer and tag opcode register. Allow to work with various degrees of precision values.
- **MMX registers:** 8, for support of single-instruction, multiple-data operations.
- **XMM registers:** 8 + 1 MXCSR register. Like the previous, but allow for 128-bit packed byte operations.
- **YMM registers:** Same as previous, but 256-bit packed.
- **Bounds registers:** 4 (64-bit) that store lower and upper bounds of the memory buffer.
- **BNDCFGU - BNDSTATUS:** configures user mode MPX operations and additional information of these operations.
- **Stack:** located in memory, supports procedure calls and passing of parameters between them.

As a part of its system-level architecture, some resources are provided for software development.

- **I/O ports:** for data transfer.
- **Control registers:** 5, which determine the operating mode of the prrocessors, and the characteristics of the executing task.
- **Memory management registers:** locations of data structures used in protected mode.
- **Debug registers:** 8, monitor the processor's debugging operations.
- **Memory type range registers:** assign memory types to regions of memory.
- **Machine specific registers:** control and report performance. Apart from the time-stamp counter, they are not accessible by the app.
- **Machine check registers:** set of control, status, and error-reporting.
- **Performance monitoring counters:** monitor processor performance events.

A. *64-bit mode execution Environment*

Very similar to the previous one, with some differences:

- 2^{64} linear addresses and 2^{46} physical adresses.
- 16 general purpose registers now (64-bit wide), up to quadword integers. 64 bit EIP and EFLAGS, now referred as the RFLAGS.
- 16 XMM and YMM registers for SIMD operations.
- Stack pointer is now 64-bit. It's size not controlled by a bit anymore
- 64-bit control and debug registers.
- Descriptor table registers are expanded to 10 bytes so they can gold a full 64-bit address.

III. Memory Organization

Physical memory is adressed by the bus of the processor. Each byte gets assigned an adress up to $2^{36} - 1$ bytes.
Any OS that works with these processors will use the facilities like segmentation and paging.

A. *IA-32 Memory Models*

Physical memory is not addressed directly, instead, these are the alternatives:

- **Flat:** the program sees a linear adress sàce where code, data and stacks are contained. Byte addressable through linear address.
- **Segmented:** group of independent address spaces called segments. To adress aa byte aa logical address (segment selector + offset). Each segment can be 2^{32} bytes. Segments are mapped into linear adress space. It's much more reliable.
- **Real address mode:** used in Intel8086, similar to segmented. The linear adress space for the program and OS is an array of segments.

B. *Paging and Virtual memory*

Way of mapping the linear address space to physical memory. Without paging there's a one-to-one correspondence.
With paging the linear address is divided into segments that are mapped to virtual memory, which is then mapped to physical. The program only sees linear space.

C. *Memory Organization in 64-bit*

More than 64 GB of physical address space, but same paging mechanism.

D. *Modes of operation vs. Memory Model*

- In protected mode any models can be used, apps can even use different models, it is handled by the OS.
- In real address mode, only the equivalent memory model is available
- In SMM, the System Management RAM uses a model similar to the real-address one.
- In compatibility, you should observe the same model as in 32-bit.
- 64-bit means segmentation is generally disbaled, so linear adress space is used.

E. *32 and 16-bit Address Operand Sizes*

The maximum size of an address is $2^n - 1$ and the operand sizes are usually $\frac{n}{2}$. Both have 16-bit segment selector and n size offset.
In protected mode the code might define default address and operand size through assembler directives.

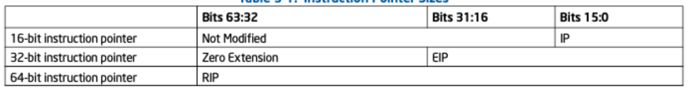
F. *Extended Physical Addressing in Protected Mode*

Individual 4GB linear addressed that are mapped into the 64GB physical address space through a virtual memory management mechanism. Requires to use protected mode.

G. *Adress Calculations in 64-Bit mode*

Flat address space for code, data and stacks. Same size for address calculations. Linear addresses are equal to effective addresses unless FS or GS segements are used with a non-zero base. Instruction pointer size might vary:

Table 3-1. Instruction Pointer Sizes



	Bits 63:32	Bits 31:16	Bits 15:0
16-bit instruction pointer	Not Modified		IP
32-bit instruction pointer	Zero Extension	EIP	
64-bit instruction pointer	RIP		

Support is provided for 64-bit displacement but it is not usual. 16 and 32-bit address are 0 extended to work in 64. But they can only access the first parts of the memory that correspond to it.

1) Canonical Addressing: Canonical means that address bits 63 to most significant are set to either all 0 or 1. Implementations can support less (so 63 to implemented must be set that way).
If the address is not in canonical form an exception (general purpose or stack) is generated.
Implied stack references generate the SS fault. And the ones that use the base registers generate the GP fault.

IV. Basic program execution registers

- **GP:** 8, for storing operands and pointers
- **Segment:** 6 selectors.
- **EFLAGS:** reports on the status of the program and has limited control of the processor.
- **EIP:** 32-bit pointer to the next instruction.

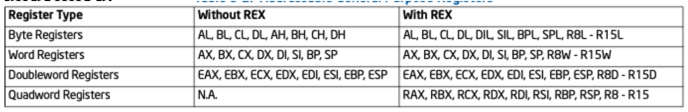
A. *General Purpose Registers:*

32-bit (EAX-EDX, ESI, EDI, EBP, ESP). Holds operands for logical, arithmetic and adrerss calculations, and memory pointers.
ESP should only be used for stack pointer.
Specific registers hold certain operands. Also some instructions asasume that pointers in certain ergisters are relative to specific segments.

- EAX - Accumulator for operands and results data.
- EBX - Pointer to data in the DS segment
- ECX - Counter for iterations
- EDX - I/O pointer
- ESI - Pointer to data in the segment pointed to by the DS register.
- EDI - Same, but pointed to by the ES register.
- EBP - Pointer to data on the stack.

Alternative names remove the E.

1) General-Purpose Registers in 64-bit mode: 8 new general purpose, that can be aaccessed at byte, word, dword and qword level. REX prefixes are used for the 64bit operand size. They are kept in compatibility mode but their values are undefined.



Register Type	Without REX	With REX
Byte Registers	AL, BL, CL, DL, AH, BH, CH, DH	AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L - R15L
Word Registers	AX, BX, CX, DX, DI, SI, BP, SP	AX, BX, CX, DX, DI, SI, BP, SP, R8W - R15W
Doubleword Registers	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D
Quadword Registers	N.A.	RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8 - R15

Can't access legacy high bytes and the new byte registers at the same time, only the legacy low bytes. This is solved by forcing the high byte references to low ones.
64 bit operands give a 64 bit result, 32 bit give a 32 one, zero extended to 64. 8 and 16 results are not modified.
Software must not depend on the 32 top bits to maaintain a value after 64 to 32 mode switch.

B. Segment Registers

(CS-GS, SS). 16-bit that identify a segment in memory. Segment selectors are often created with directives. Or programmers might create them directly.
In flat memory, segment registers are loaded with selectors that point to overlapping segments. They comprise the linear aaddress space of the program. They are overlapping (one for code and one for data, stacks).
Each register loaded with a different selector, up to 6 segments in the linear address space accessed at a certain time.
Each segment register is asssociated with one of 3 types of storage:

- **Code:** CS, stores executed instructions. The processor fetches the contents of the EIP so it knows the offset of the instruction to do. It doesn't access this register directly.
- **Data:** DS-GS, secure access to different data structures. The app loads segment selectors to access these.
- **Stack:** SS, the procedure stack of the program is stored. Can be loaded explicitly

1) In 64-Bit mode: CS-ES and SS are treated as their base was 0, so there's a flat adress for each storage.
Segmentation is generally disabled, but the processor might still perform legacy checks.
Limit checks for some registers are disabled.

C. EFLAGS Register

32 bit, contains status flags, a control one, and system ones. Some are reserved and software shouldn't depend on the. Some values can be modified through instructions.
When a task is interrupted its EFLAGS value is saved in the task state segment. When a call is made to an exception handler procedure, the EFLAGS is saved on the procedure stack.

1) Status flags:

- Carry (0): if an operation carries most significant bit, indicates also overflow condition when unsigned.
- Parity (2): to see if there's an even number of 1.
- Auxiliary Carry (4): used in binary-coded decimal arithmetic.
- Zero (6): if the result is 0.
- Sign (7): Equal to the most-significant bit of the result.
- Overflow (11): if the integer is too large or too small.

Only the carry can be modified directly. It also plays an important role in multiple precision.
Arithmetic operation can produce results for 3 data types (unsigned, signed and BCD).

2) DF Flag: Bit 10. Controls string instructions.

3) System Flags and IOPL field:

- Trap (8): single step mode for debugging.
- Interrupt enable (9): set to respond to maskable interrupts.
- IOPL (12-13): privilege level (I/O) of the current program. Must be less or equal than the privilege level to access the I/O space.
- Nested task (14): controls chaining of interrupted and called tasks
- Virtual8086 (17): self-explanatory
- Allignment check (18): set if the AM bit is set in the CR0 register.
- Virtual interrupt (19): virtual image of the 9 flag.
- Virtual interrupt pending (20): set to indicate that an interrupt is pending.
- Identification: support for CPUID instruction

4) RFLAGS in 64: The 32 extra bits are reserved.

V. Instruction Pointer

Contains the offset of the code segment for the next instruction to be executed. Can't be modified by software and can only be read through a instruction call.
Because of prefetching in IA32, the value in the bus might not match the one in the register.

A. In 64 bit

Added RIP-Relative addressing, by addding a displacement to the RIP of the next instruction (RIP is the new name of the register)

VI. Operand Size and Address Size Attrributes

Every code segment has these. If its flag is selected, they are 32 bit, if not 16 (also in v8086).
Size of operands depend of the number of bits, it can be 8 or n bits. Address Size is the size used to address memory.
Can be overrriden by aadding a prefix to an instruction.

A. In 64 bit

Default address is 64bit, and operand is 32. 32/64 bit data and addresses can be mixed and they are supported. 16bit addresses are not supported.
REX prefixes (allow to override) are 4bit long.

VII. Operand Addressing

Data for a source operand can be in: the instruction, register, memory locaation or I/O port, and can be returned to the 3 latter.

A. Immediate Operands

The ones in the instruction. Most arithmetic instruction allow for the operand to be an immediate value. Can never be bigger than 2^{32}

B. Register Operands

Great variety of destination registers, depending on instruction. Some instructions (like quadword) use register pairs notated with a :.
Several instructions can load the EFLAGS register and change values in it, or use it as conditionals.
When accessing a system register with a system instruction, the register is generally an implied operand of it.

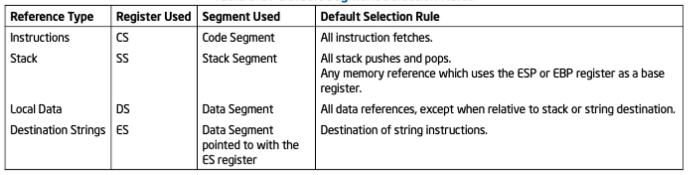
C. *Memory Operands*

As mentioned before, referred to through a segment selector (16 bit) and offset (32 - 64 bit).

D. *Specifying a Segment Selector*

Can be done implicitly and explicitly. Most common way to do it is to specify it through a register. The DS segment default can be overriden to allow other ssegments to be accessd.

Table 3-5. Default Segment Selection Rules



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 of string instructions.

A segment override is done through a prefix. But the following default segment seleections can't be:

- Instruction fetches must be done from code segment.
- Destination strings must be stored in the data segment provided.
- Push and pop operations.

Some instructions, like `MOV` require a segment selector to be specified explicitly.

1) Segmentation in 64bit: In compatibility mode it works just as 32-bit. In 64-bit it's generally disabled, which creates a flat linear space.

E. *Specifying an offset*

Can be a static value or through an aaddress computation made from: displacement, base, index, and scale factor. The first 3 can have negative or positive value.
Creates an **effective address**.
The ESP can't be used as index register, and when it or the EBP is used as base, the SS segment is the default one, in all other cases, it is the DS one.
Here are some common addressing modes:

- Displacement: direct offset to the operand, also called staatic address. Used for scalar operands
- Base: indirect offset, used for dynamic storage.
- Base + Displacement: 2 purposes. Index to an aarray when the element size is not a power of 2, or access the field of a record. EBP register is best choice for base one because it selects stack segment.
- $Index * Scale$ + Displacement: Index to an array when it is a power of 2.
- Base + Index + Displacement: 2 dimentional array or several instances of an array of records.
- Base + $Index + Scale$ + Displacement: efficient indexing in 2D array when elements are power of 2 in size.

Powers of 2 up to 8.

1) Specifying an Offset in 64Bit: Displacement is 8, 16 or 64bit value. Base and index arevalues in a 64bit GP register, and Scale is a power of 2 value multipled by the index.
RIP + Displacement: 32-bit displacement to calculate the effective address.

F. *Aassembler and Compiler Addressing Modes*

The selected combination of values is encoded in the instruction, assemblers permit all combinations. And high-level compilers will choose the appropiate one.

G. *I/O Port Addressing*

Can be addressed with either an immediate operand or a value in the DX register.