This work is licensed under a Creative Commons license

Attribution-NonCommercial-NoDerivatives 4.0 International
(CC BY-NC-ND 4.0)

You are free to:

Share copy and redistribute the material in any medium or format.

Under the following terms:

Attribution You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial You may not use the material for commercial purposes.

NoDerivatives If you remix, transform, or build upon the material, you may not distribute the modified material.

Intel x86/x64

An overview of the Instruction Set Architecture

Giovanni Lagorio

giovanni.lagorio@unige.it
https://csec.it/people/giovanni_lagorio
Twitter & GitHub: zxgio

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova, Italy

www.zenhack.it

A little history of Intel CPUs

1978 16-bit processors: 8086 and 8088 (8-bit bus); segmentation, $2^{20} = 1$ MB address space. So, 16 bit words, forever ☺

1982 286, protected mode using segment registers as selector, $2^{24} = 16$ MB address space

1985 386, 32-bit processor, virtual-8086 mode, $2^{32} = 4$ GB address space, segmented-memory model and flat memory model, paging with 4k pages

1989 486, integrated x87 FPU

1993 Pentium, 4k and 4M pages

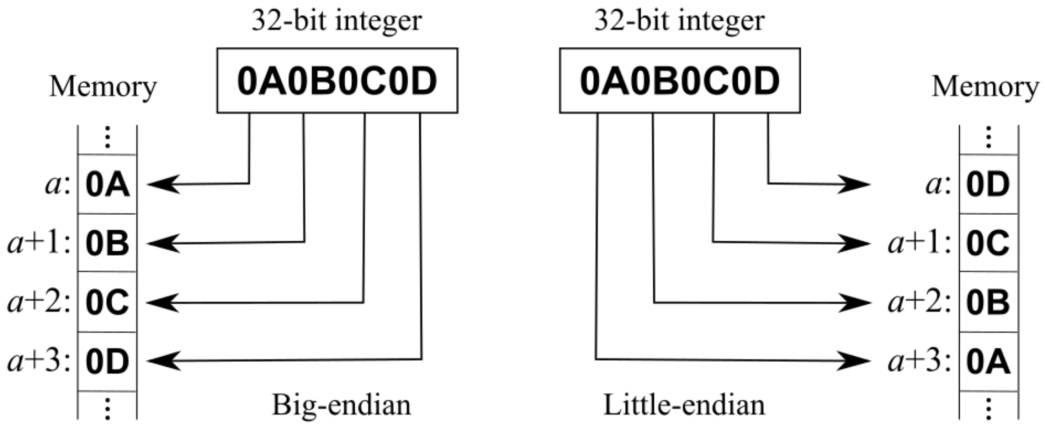
1995-1999 P6 family, MMX and SSE $\rightarrow$ SIMD parallelism

2000-2007 Pentium 4/Xeon family, SSE2 and SS3; AMD64/Intel 64, hyperthreading and VT

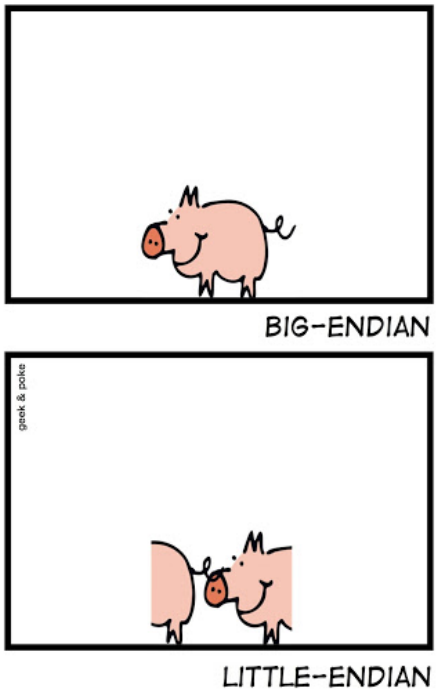
2008 Core i7 family, SSE4.2, 2nd generation Virtualization Technology

...

Endianness



source: Wikipedia



source: Simply Explained

Intel CPUs use little endian

CPU: the Intel x86/x64

We mostly deal with user-mode of:

- x86/IA-32 ("Intel Architecture, 32-bits", sometimes called i386)
- x86-64/x64/Intel 64/AMD64 is an extension to original IA-32
  - Beware: IA-64 (Intel Architecture 64)/Itanium is an almost completely unrelated 64 bit architecture

documented in Intel Software Developer Manuals

- at the time of writing, a handy 5000+ (!) page reference
- hundreds of opcodes, *however*
- 5 of them cover about 64% of opcodes in "normal programs" [B+06]
  - namely: `mov`, `push`, `pop`, `call`, `cmp`

  14 opcodes cover about 90%, and 72 cover $> 99.8\%$

x86/64 documentation

- Cheat-sheet: `http://www.jegerlehner.ch/intel/` (32 bits only)
- Architecture 1001: x86-64 Assembly in *Open Security Training 2* `https://p.ost2.fyi/courses/course-v1:OpenSecurityTraining2+Arch1001_x86-64_Asm+2021_v1/course/`
- chapters 7, 8 and 9 of *x86-64 Assembly Language Programming with Ubuntu* [Jor18] — `http://www.egr.unlv.edu/~ed/x86.html`
- *PC Assembly Language [Car07] (32 bits only)*— `https://pacman128.github.io/pcasm/`
- `https://en.wikipedia.org/wiki/X86_instruction_listings`

Modes of operation

16/32/64 16-bit real-address mode, the "8086 mode"; activated at power-up. If interested, watch: 35C3 - A deep dive into the world of DOS viruses

32 protected mode, the "normal" mode

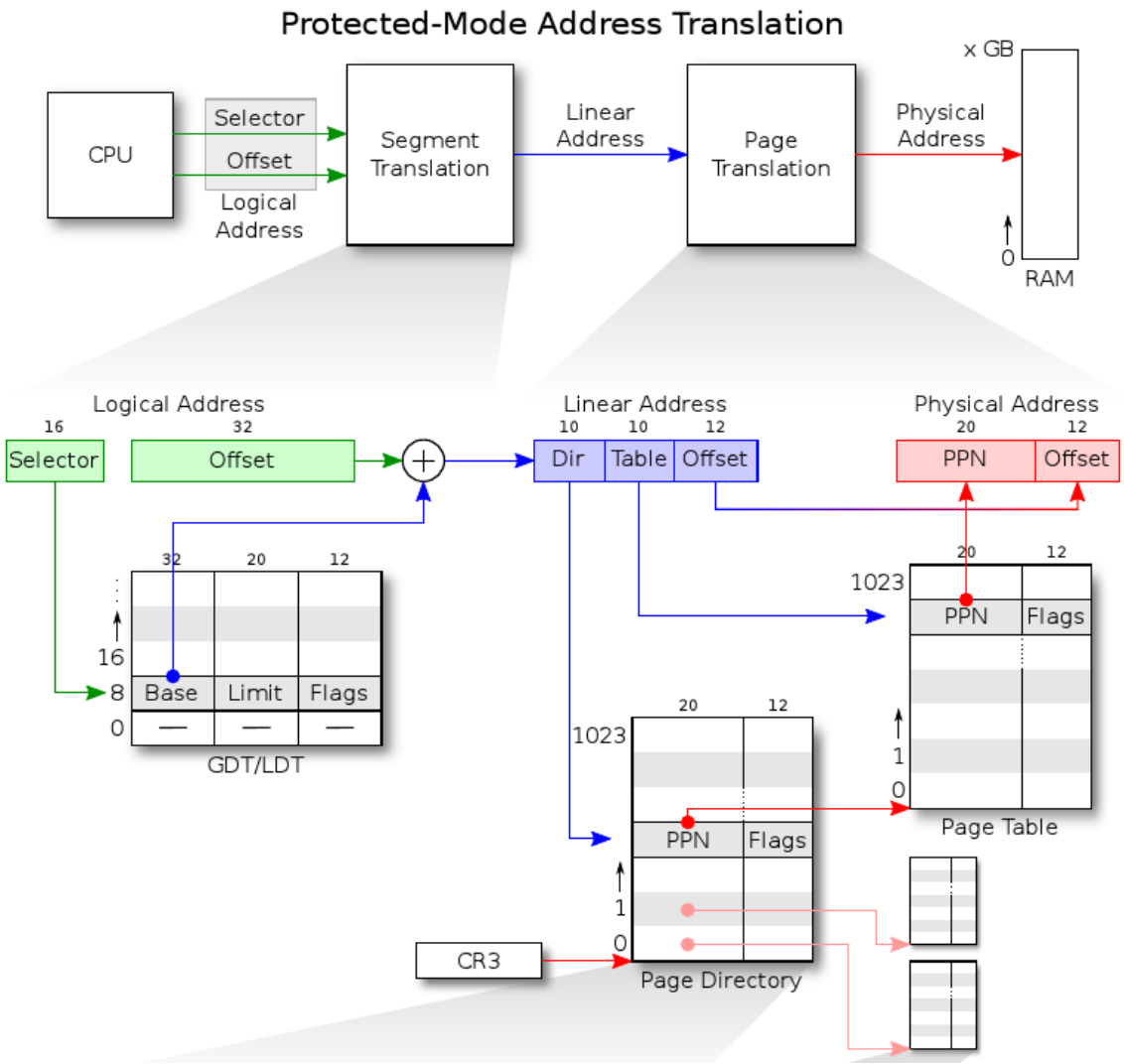
- four protection rings, two used: 0 (kernel) and 3 (user)
- applications run with a paged 32-bit flat address space

64 IA32e, with two submodes:

- compatibility mode, similar to 32-bit protected mode, permits legacy 16/32-bit application to run without recompilation on a 64-bit OS
- 64-bit mode, allows to run 64-bit applications
  - enabled by the OS on a code-segment basis

There is also a special mode, *system-management mode*, intended for use only by firmware for implementing platform-specific functions

Memory translation



https://pdos.csail.mit.edu/6.828/2009/lec/x86_translation.pdf

Segment registers

Technically, six 16-bit segment registers: `CS`, `DS`, `ES`, `SS`, `FS` and `GS`.
However, modern OSes use paging (only) $\rightarrow$ virtual flat address space

- in 32-bit mode, segment registers CS/DS/ES and SS hold 16-bit "useless" segment selectors
  - i.e. base$=0$, limit$=$"$+\infty$"
- in 64-bit mode,
  - `CS`, `DS`, `ES`, `SS` are treated as if each segment base is 0
  - all limit checks are disabled
- `FS` and `GS` used for TLS

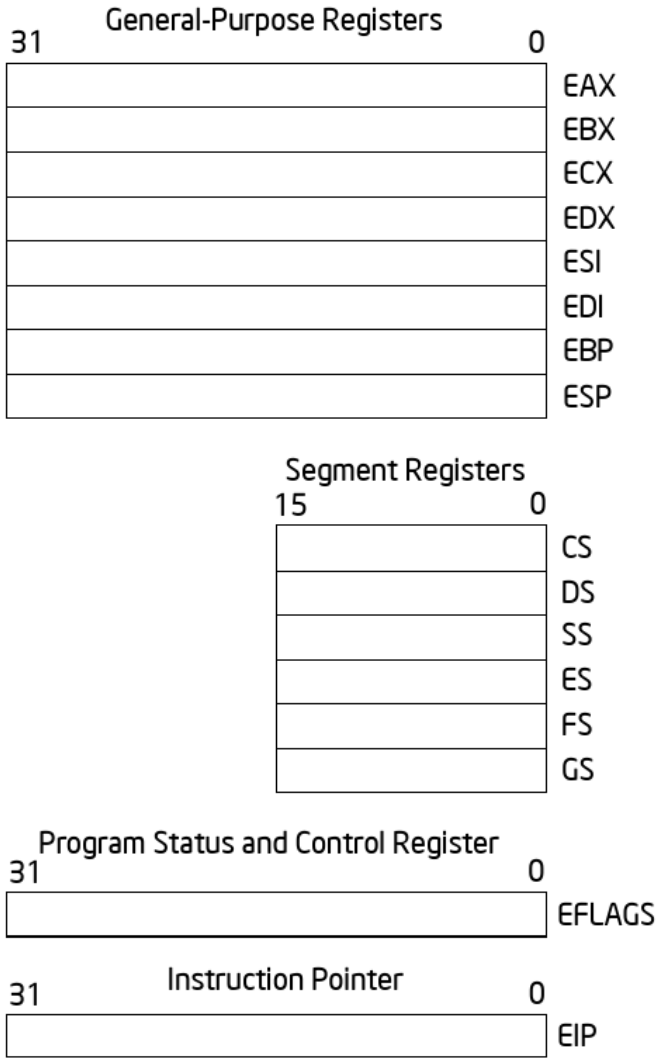
  Linux `FS` in 64-bit mode, `GS` in 32-bit mode

  `https://wiki.osdev.org/Thread_Local_Storage`

  Windows viceversa, `FS` in 32-bit mode, `GS` in 64-bit mode

  `https://en.wikipedia.org/wiki/Win32_Thread_Information_Block`

Basic execution registers in 32 bit modes



(Mostly) General purpose:

- `EAX` – accumulator
- `EBX` – pointer to data
- `ECX` – counter
- `EDX` – I/O pointer
- `ESI` – source pointer for string ops
- `EDI` – destination pointer

Execution registers:

- `EIP` – instruction pointer
- `ESP` – stack pointer
- (`EBP` – base/frame pointer)
- `EFLAGS` – carry, sign, zero, parity, ...

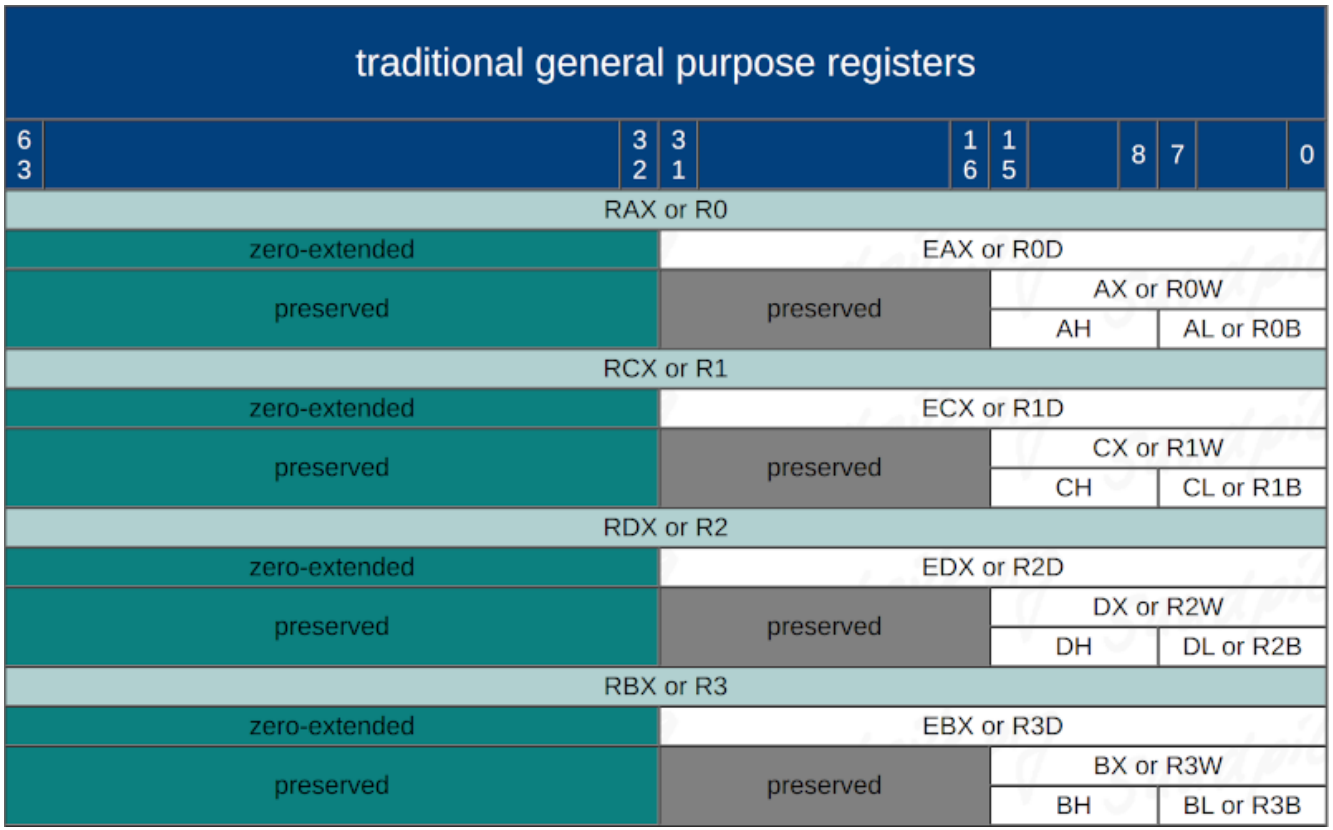
GPRs in 32 bit modes

General-Purpose Registers

31 16	15 8	7 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

GPRs in 64 bit modes

- sixteen 64-bit registers `R0-R15[D,W,B]`; `R0-R7` are aliases to old ones
  - `R0=RAX`, `R0D=EAX`, `R0W=AX`, `R0B=AL` (no `R0...` for `AH`)
  - ...
- *beware:* assigning the 32 lower bits, zero-extends the upper ones



Source: `https://www.sandpile.org/x86/gpr.htm`

Operands

Machine-instructions act on zero or more operands
The data for an operand can be located in:

- the instruction itself, in case of an immediate operand
- a register
- a memory location
- (an I/O port)

when there are more operands, only one can refer to memory

Addressing modes

Addressing modes for 16-bit x86 processors can be summarized by the formula:[15][16]

$$\left\{\begin{matrix}\texttt{CS}:\\ \texttt{DS}:\\ \texttt{SS}:\\ \texttt{ES}:\end{matrix}\right\}\left[\left\{\begin{matrix}\texttt{BX}\\ \texttt{BP}\end{matrix}\right\}\right]+\left[\left\{\begin{matrix}\texttt{SI}\\ \texttt{DI}\end{matrix}\right\}\right]+[\text{displacement}]$$

Addressing modes for 32-bit x86 processors,[17] and for 32-bit code on 64-bit x86 processors, can be summarized by the formula:[18]

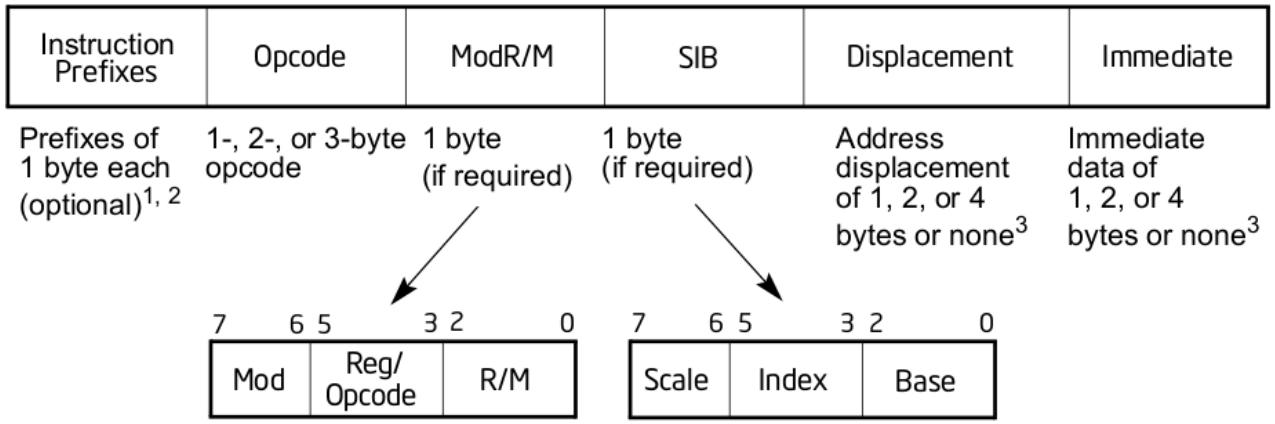
$$\left\{\begin{matrix}\texttt{CS}:\\ \texttt{DS}:\\ \texttt{SS}:\\ \texttt{ES}:\\ \texttt{FS}:\\ \texttt{GS}:\end{matrix}\right\}\left[\left\{\begin{matrix}\texttt{EAX}\\ \texttt{EBX}\\ \texttt{ECX}\\ \texttt{EDX}\\ \texttt{ESP}\\ \texttt{EBP}\\ \texttt{ESI}\\ \texttt{EDI}\end{matrix}\right\}\right]+\left[\left\{\begin{matrix}\texttt{EAX}\\ \texttt{EBX}\\ \texttt{ECX}\\ \texttt{EDX}\\ \texttt{EBP}\\ \texttt{ESI}\\ \texttt{EDI}\end{matrix}\right\}*\left\{\begin{matrix}1\\ 2\\ 4\\ 8\end{matrix}\right\}\right]+[\text{displacement}]$$

Addressing modes for 64-bit code on 64-bit x86 processors can be summarized by the formula:[18]

$$\left\{\begin{matrix}\left\{\begin{matrix}\texttt{FS}:\\ \texttt{GS}:\end{matrix}\right\}[\text{general register}]+\left[\text{general register}*\left\{\begin{matrix}1\\ 2\\ 4\\ 8\end{matrix}\right\}\right]\\ \\ \texttt{RIP}\end{matrix}\right\}+[\text{displacement}]$$

https://en.wikipedia.org/wiki/X86#Addressing_modes

Instruction format: from one byte to 15 bytes (!)



- Prefixes: optional, overrides for segment/operand sizes and so on
  - Eg. REX prefixes are used to generate 64-bit operand sizes or to reference registers R8-R15 (See 2.2.1.1 in Volume 2)
- ModR/M: specifies whether instruction accesses memory or registers
  - In 64-bit mode ModR/M and SIB displacement sizes do not change, they remain 8/32 bits and are sign-extended to 64 bits
- Immediates: typical size remains 32 bits. When 64 bits are needed, operands are sign-extended
  - A REX prefix allows us to load a 64-bit immediate into a GPR
  - Instructions that encode an 8-byte immediate field do not use a displacement field and vice versa

Two syntaxes for assembly: AT&T vs Intel

- Source-destination ordering
- Register naming: AT&T prefixes register names with %
- Immediate operands: AT&T prefixes immediate operands with $
- Operand size
  - in AT&T determined from the last character of the op-code name
  - Intel prefixes memory operands with *size* `ptr`
    - optional when there are no ambiguities
- Memory operands
  - in Intel syntax the base register is enclosed in `[` and `]`
  - in AT&T they change to `(` and `)`

  moreover, Intel indirect memory references like `[base + index*scale + disp]`, changes to `disp(base, index, scale)` in AT&T

`http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s3`

AT&T vs Intel: examples

Intel Code	AT&T Code
`mov eax,1`	`movl $1,%eax`
`mov ebx,0ffh`	`movl $0xff,%ebx`
`int 80h`	`int $0x80`
`mov ebx, eax`	`movl %eax, %ebx`
`mov eax,[ecx]`	`movl (%ecx),%eax`
`mov eax,[ebx+3]`	`movl 3(%ebx),%eax`
`mov eax,[ebx+20h]`	`movl 0x20(%ebx),%eax`
`add eax,[ebx+ecx*2h]`	`addl (%ebx,%ecx,0x2),%eax`
`lea eax,[ebx+ecx]`	`leal (%ebx,%ecx),%eax`
`sub eax,[ebx+ecx*4h-20h]`	`subl -0x20(%ebx,%ecx,0x4),%eax`

http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s3

References

[B+06] Daniel Bilar et al.
Statistical structures: Fingerprinting malware for classification and analysis.
Proceedings of Black Hat Federal 2006, 2006.

[Car07] Paul A Carter.
PC assembly language, 2007.
https://pacman128.github.io/pcasm/.

[Jor18] Ed Jorgensen.
x86-64 assembly language programming with Ubuntu, 2018.
http://www.egr.unlv.edu/~ed/x86.html.