

# Chapter 30 - The x86 Processor

---

The x86 family began as the Intel 8086 in 1978 and grew into the most widely deployed processor architecture in history. Intuition Engine's x86 processor is 32-bit but not a PC-compatible 386: it implements the complete 8086/8088 instruction set, the 386-era 32-bit register and operand extensions, the x87 floating-point stack, and selected later flat-mode opcodes such as BSWAP and CMOVcc. It runs permanently in a flat real-address mode. Protected mode, paging, segment descriptors, control registers, task-state segments, and the LDT/GDT/IDT machinery are not present.

This makes it a clean target for programmers who already know x86 opcodes but do not need protected mode.

## 30.1 Register set

---

### 30.1.1 General registers

Eight 32-bit general registers, each with 16-bit and 8-bit sub-names that overlap the low bits of the 32-bit form:

| 32-bit | 16-bit | 8-bit high | 8-bit low | Common use                                  |
|--------|--------|------------|-----------|---|
| EAX    | AX     | AH         | AL        | Accumulator                                 |
| EBX    | BX     | BH         | BL        | Base register                               |
| ECX    | CX     | CH         | CL        | Count register                              |
| EDX    | DX     | DH         | DL        | Data register; high half of multiply/divide |
| ESI    | SI     | -          | -         | Source index                                |
| EDI    | DI     | -          | -         | Destination index                           |
| EBP    | BP     | -          | -         | Base pointer (stack frame)                  |
| ESP    | SP     | -          | -         | Stack pointer                               |

Writes to the 8-bit or 16-bit form leave the unused high bits of the 32-bit register untouched. The four "common use" labels are conventions, not architectural roles, but several instructions do require specific registers (MUL/DIV use EAX/EDX, string ops use ESI/EDI/ECX, etc.).

### 30.1.2 Segment registers

Six 16-bit segment registers:

| Register | Conventional use          |
|----------|---------------------------|
| CS       | Code segment              |
| DS       | Default data segment      |
| ES       | Extra data segment        |
| SS       | Stack segment             |
| FS       | Extra data segment (386+) |
| GS       | Extra data segment (386+) |

On Intuition Engine all six segments are cleared to 0 on reset, which gives a flat memory model: physical address is just the 32-bit offset. Programs that load a non-zero segment selector get  $\text{seg} * 16 + \text{offset}$  (the real-address-mode rule), but the typical case is to leave the segments at zero and address all of memory directly with 32-bit offsets.

### 30.1.3 Instruction pointer

EIP is the 32-bit instruction pointer. IP is its low 16 bits, used by code that runs in 16-bit mode.

On reset  $\text{EIP} = \$00000000$ . The historical 8086 reset vector at  $\text{F000:F000}$  does not apply in the flat model. Monitor examples in this chapter place code at  $\$1000$  and set EIP there explicitly.

Loaded .ie86 flat images use the reset convention. The image bytes are placed starting at physical address  $\$00000000$ , and execution begins with  $\text{EIP} = 0$ . If the useful body of a program lives higher in memory, put a small jump or entry stub at address 0 to reach it. That image rule is separate from IE Mon work, where you may set EIP to any address by hand before running.

### 30.1.4 EFLAGS

The 32-bit EFLAGS register. The low 16 bits are the historical FLAGS register and remain compatible with 8086 and 286 code; the upper bits hold 386-era extensions.

| Bit   | Mnemonic | Name                             |
|-------|----------|----------------------------------|
| 0     | CF       | Carry - status                   |
| 2     | PF       | Parity - status                  |
| 4     | AF       | Auxiliary carry (BCD) - status   |
| 6     | ZF       | Zero - status                    |
| 7     | SF       | Sign - status                    |
| 8     | TF       | Trap (single-step) - system      |
| 9     | IF       | Interrupt enable - system        |
| 10    | DF       | Direction (string ops) - control |
| 11    | OF       | Overflow - status                |
| 12-13 | IOPL     | I/O privilege level - system     |
| 14    | NT       | Nested task - system             |
| 16    | RF       | Resume - system                  |
| 17    | V86      | Virtual-8086 mode - system       |
| 18    | AC       | Alignment check - system         |

Bits 1, 3, 5, and 15 are reserved (bit 1 reads as 1). The IOPL, NT, RF, V86, and AC bits are decoded but have no functional effect because IE has no protected mode.

## 30.2 Memory model

Address space is 32-bit flat. Multi-byte data is **little-endian**: a 32-bit value  $\$12345678$  written to address  $\$1000$  places  $\$78$  at  $\$1000$ ,  $\$56$  at  $\$1001$ ,  $\$34$  at  $\$1002$ ,  $\$12$  at  $\$1003$ .

Word and doubleword operands need not be aligned, but unaligned accesses are slower. The MMIO map of Chapter 24 is accessible directly; an `OUT DX, AL` and a `MOV BYTE PTR [F0700], AL` both reach the terminal.

Absolute data operands in `$000F0000-$000FFFF` reach native MMIO directly. x86 also keeps a smaller compatibility mirror for data accesses in `$F000-$FFFF`, which maps to `$000F0000-$000F0FFF`. That mirror is not used for instruction fetch. If `EIP = $F000`, the CPU fetches code bytes from flat RAM at `$0000F000`, not from the MMIO block.

## 30.3 Addressing modes

x86 uses the ModR/M and (when scaled-index is needed) SIB byte to encode operand locations. The expressible forms:

| Form                              | Example                              |
|-----------------------------------|--------------------------------------|
| Register                          | <code>MOV EAX, EBX</code>            |
| Immediate                         | <code>MOV EAX, \$12345678</code>     |
| Direct (absolute)                 | <code>MOV EAX, [F0700]</code>        |
| Register indirect                 | <code>MOV EAX, [EBX]</code>          |
| Base + displacement               | <code>MOV EAX, [EBX+8]</code>        |
| Base + index                      | <code>MOV EAX, [EBX+ECX]</code>      |
| Base + index*scale                | <code>MOV EAX, [EBX+ECX*4]</code>    |
| Base + index*scale + displacement | <code>MOV EAX, [EBX+ECX*4+16]</code> |

16-bit addressing forms (`[BX+SI]`, `[BP+DI]`, etc.) are still available with an address-size prefix. 32-bit addressing is the default when no prefix is used.

A segment override prefix (`CS:`, `DS:`, `ES:`, `SS:`, `FS:`, `GS:`) forces a particular segment register. Because all segments resolve to base zero on IE, the override changes which segment register is used in the formula but not the resulting address.

## 30.4 Instruction set

Appendix G has the full opcode table. The groups below outline what is available.

### 30.4.1 Data movement

`MOV`, `MOVZX` (zero-extend), `MOVSX` (sign-extend), `CMOVcc` (conditional move to a register), `XCHG`, `BSWAP` (32-bit byte swap), `XLAT` (`AL = [EBX + AL]`), `PUSH/POP`, `PUSHA/POPA`, `PUSHF/POPF`, `LEA`, `LDS/LES/LFS/LGS/LSS` (load far pointer).

`CMOVcc` uses the same condition names as `Jcc`: `CMOVZ`, `CMOVNZ`, `CMOVB`, `CMOVNB`, `CMOVL`, `CMOVNL`, and the other flag tests. The destination is always a register and the source may be a register or memory. A memory source is read even when the condition is false, so do not use an untaken `CMOVcc` to avoid reading a read-sensitive MMIO register.

### 30.4.2 Arithmetic

`ADD`, `ADC`, `SUB`, `SBB`, `INC`, `DEC`, `NEG`, `CMP`. Multiply: `MUL` (unsigned), `IMUL` (signed, three forms). Divide: `DIV` (unsigned), `IDIV` (signed). Decimal: `DAA`, `DAS`, `AAA`, `AAS`, `AAM`, `AAD`. Sign/zero extend `AL` to `AX`, `AX` to `EAX`: `CBW/CWDE`. Sign-extend `EAX` to `EDX`: `EAX`: `CDQ`.

### 30.4.3 Logic, shifts, rotates

AND, OR, XOR, NOT, TEST. Shifts: SHL, SHR, SAL, SAR, SHLD, SHRD (386+ double-precision shifts). Rotates: ROL, ROR, RCL, RCR.

### 30.4.4 Bit and bit-string

BT, BTS, BTR, BTC: bit test / test-and-set/reset/complement. BSF, BSR: bit scan forward / reverse. SETcc: set byte on condition.

### 30.4.5 String

MOVS, CMPS, SCAS, LODS, STOS, INS, OUTS. Each has explicit (MOVSB/MOVSW/MOVSD) and implicit forms. The REP, REPE, and REPNE prefixes repeat using ECX as a counter.

### 30.4.6 Control transfer

| Mnemonic     | Effect   |
|--------------|--|
| JMP          | Unconditional jump (relative, register-indirect, memory-indirect, far)                                       |
| Jcc          | Conditional jump (JE/JZ, JNE/JNZ, JL, JG, JC, JNC, JO, JNO, JS, JNS, JP, JNP, JA, JAE, JB, JBE, JCXZ, JECXZ) |
| LOOP         | Decrement ECX, jump if non-zero  |
| LOOPE/LOOPNE | Decrement and conditionally jump   |
| CALL         | Call subroutine (near or far)  |
| RET          | Return   |
| INT n        | Software interrupt (n = 0 . . 255)   |
| INT0         | Trap if 0F set   |
| IRET/IRETD   | Return from interrupt  |

### 30.4.7 Port I/O

| Mnemonic     | Operation                                     |
|--------------|---|
| IN AL, imm8  | Read 8-bit port imm8 into AL                  |
| IN AX, imm8  | Read 16-bit port                              |
| IN EAX, imm8 | Read 32-bit port                              |
| IN AL, DX    | Read port DX (allows full 16-bit port number) |
| OUT imm8, AL | Write AL to port imm8                         |
| OUT DX, AL   | Write AL to port DX                           |
| INS/OUTS     | Block port I/O (with REP prefix)              |

Intuition Engine assigns ports per device; the device chapters list the relevant port numbers. Many devices are also reachable through memory-mapped registers and most x86 code uses those directly.

The useful x86 port assignments are:

| Ports     | Device                              |
|-----------|-------------------------------------|
| \$60-\$69 | POKEY direct register window        |
| \$A0-\$AD | VGA registers                       |
| \$B0-\$B7 | Voodoo register and texture gateway |
| \$D4/\$D5 | ANTIC select/data                   |
| \$D6/\$D7 | GTIA select/data                    |
| \$E0/\$E1 | SID select/data                     |
| \$F0/\$F1 | PSG select/data                     |
| \$F2/\$F3 | TED select/data                     |
| \$FE      | ULA border port                     |

### 30.4.8 Flag manipulation

CLC, STC, CMC (clear/set/complement carry); CLI, STI (disable/enable interrupts); CLD, STD (clear/set direction); LAHF, SAHF (load/store AH from/to flags); PUSHF/POPF and their 32-bit PUSHFD/POPFD siblings.

### 30.4.9 Other

NOP, HLT, WAIT, CPUID (returns vendor and feature bits).

The x87 floating-point coprocessor is present; FPU opcodes D8-DF execute against an 80-bit stack of eight registers ST(0)-ST(7). The full FPU instruction set is in Appendix G.

## 30.5 Interrupts

The interrupt vector table sits at physical address \$00000000, just like the 8086 real-mode IVT. Each entry is four bytes: a 16-bit IP followed by a 16-bit CS.

When INT *n* executes or a hardware interrupt fires:

1. The CPU pushes the low 16 bits of EFLAGS, then CS, then IP.
2. Clears IF and TF.
3. Loads IP from [4\*n] and CS from [4\*n + 2].

IRET reverses the push order, popping IP, CS, and FLAGS.

NMI is vector 2. Interrupt vectors 0-7 and 16-19 are reserved by the architecture (divide error, debug, NMI, breakpoint, overflow, bounds, invalid opcode, device-not-available, and FPU diagnostics).

## 30.6 What is not implemented

By design IE's x86 omits:

- Protected mode and the descriptor tables (GDT, IDT, LDT, TSS).
- Paging and the CR0-CR4 control registers.
- Real-mode BIOS interrupts. The IVT exists but contains zero vectors at reset; a program is free to install its own.
- The historical reset vector at F000:F000. IE starts at EIP = 0; monitor-entered examples normally set EIP to their chosen start address.

- SMM, VMX, all post-486 MSR machinery.

Programs that need any of these would have to be written for a different processor. For ordinary applications, the flat 32-bit real-address model is the most pleasant x86 environment available.

### 30.7 A small example

This x86 byte-entry program uses byte stores to make TED play two detuned square voices. In flat IE x86 mode, MOV [addr],AL reaches the MMIO address directly:

```
(x86)> w 1000 B0 01 A2 00 08 0F 00 B0 1C A2 00 0F 0F 00 B0 02
(x86)> w 1010 A2 04 0F 0F 00 B0 58 A2 01 0F 0F 00 B0 02 A2 02
(x86)> w 1020 0F 0F 00 B0 38 A2 03 0F 0F 00 EB FE
(x86)> d 1000 #13
001000: B0 01          MOV AL, $01
001002: A2 00 08 0F 00  MOV [$000F0800], AL
001007: B0 1C          MOV AL, $1C
001009: A2 00 0F 0F 00  MOV [$000F0F00], AL
00100E: B0 02          MOV AL, $02
001010: A2 04 0F 0F 00  MOV [$000F0F04], AL
001015: B0 58          MOV AL, $58
001017: A2 01 0F 0F 00  MOV [$000F0F01], AL
00101C: B0 02          MOV AL, $02
00101E: A2 02 0F 0F 00  MOV [$000F0F02], AL
001023: B0 38          MOV AL, $38
001025: A2 03 0F 0F 00  MOV [$000F0F03], AL
T 00102A: EB FE          JMP SHORT $0000102A
(x86)> r eip 1000
(x86)> b 102A
(x86)> g
(x86)> m F0F00 1
000F0F00: 1C 58 02 38 02 00 00 00 00 00 00 00 00 00 00 00 .X.8.....
(x86)> bc 102A
```

The byte stream is small enough to type and inspect directly:

| Address | Bytes          | Meaning   |
|---------|----------------|---|
| \$1000  | B0 01          | MOV AL, 1; opcode \$B0 loads an immediate byte into AL.   |
| \$1002  | A2 00 08 0F 00 | MOV [\$000F0800], AL; opcode \$A2 stores AL at an absolute little-endian address. This enables audio. |
| \$1007  | B0 1C          | Loads TED voice 1 low divider byte \$1C.  |
| \$1009  | A2 00 0F 0F 00 | Writes to \$F0F00, TED_FREQ1_LO.  |
| \$100E  | B0 02          | Loads TED voice 1 high divider bits \$02.   |
| \$1010  | A2 04 0F 0F 00 | Writes to \$F0F04, TED_FREQ1_HI.  |
| \$1015  | B0 58          | Loads TED voice 2 low divider byte \$58.  |
| \$1017  | A2 01 0F 0F 00 | Writes to \$F0F01, TED_FREQ2_LO.  |
| \$101C  | B0 02          | Loads TED voice 2 high divider bits \$02.   |
| \$101E  | A2 02 0F 0F 00 | Writes to \$F0F02, TED_FREQ2_HI.  |
| \$1023  | B0 38          | Loads TED sound control \$38, both voices on with volume 8.   |

| Address | Bytes          | Meaning  |
|---------|----------------|--|
| \$1025  | A2 03 0F 0F 00 | Writes to \$F0F03, TED_SND_CTRL.   |
| \$102A  | EB FE          | Short jump back by 2, which branches to \$102A and keeps the sound active. |

The first store enables audio. TED voice 1 gets divider \$21C through low byte \$1C and high bits \$02; voice 2 gets divider \$258 through low byte \$58 and high bits \$02. \$38 in TED\_SND\_CTRL enables both voices and sets volume 8, the chip's maximum. The address fields after each A2 opcode are little-endian: \$000F0F03 is entered as 03 0F 0F 00.

## 30.8 TED video example

---

The same two x86 instruction forms can draw on the TED video chip. This program enables TED video, sets the border and background colours, puts character code 1 in the top-left matrix cell, assigns that cell a foreground colour, and replaces character 1 with an 8-byte motif.

```

(x86)> w 1100 B0 01 A2 58 0F 0F 00 B0 18 A2 20 0F 0F 00
(x86)> w 110E B0 08 A2 24 0F 0F 00 B0 06 A2 30 0F 0F 00
(x86)> w 111C B0 2E A2 40 0F 0F 00 B0 01 A2 00 30 0F 00
(x86)> w 112A B0 4E A2 00 34 0F 00 B0 FF A2 08 38 0F 00
(x86)> w 1138 B0 81 A2 09 38 0F 00 B0 BD A2 0A 38 0F 00
(x86)> w 1146 B0 A5 A2 0B 38 0F 00 B0 A5 A2 0C 38 0F 00
(x86)> w 1154 B0 BD A2 0D 38 0F 00 B0 81 A2 0E 38 0F 00
(x86)> w 1162 B0 FF A2 0F 38 0F 00 EB FE
(x86)> d 1100 #31
001100: B0 01          MOV AL, $01
001102: A2 58 0F 0F 00  MOV [$000F0F58], AL
001107: B0 18          MOV AL, $18
001109: A2 20 0F 0F 00  MOV [$000F0F20], AL
00110E: B0 08          MOV AL, $08
001110: A2 24 0F 0F 00  MOV [$000F0F24], AL
001115: B0 06          MOV AL, $06
001117: A2 30 0F 0F 00  MOV [$000F0F30], AL
00111C: B0 2E          MOV AL, $2E
00111E: A2 40 0F 0F 00  MOV [$000F0F40], AL
001123: B0 01          MOV AL, $01
001125: A2 00 30 0F 00  MOV [$000F3000], AL
00112A: B0 4E          MOV AL, $4E
00112C: A2 00 34 0F 00  MOV [$000F3400], AL
001131: B0 FF          MOV AL, $FF
001133: A2 08 38 0F 00  MOV [$000F3808], AL
001138: B0 81          MOV AL, $81
00113A: A2 09 38 0F 00  MOV [$000F3809], AL
00113F: B0 BD          MOV AL, $BD
001141: A2 0A 38 0F 00  MOV [$000F380A], AL
001146: B0 A5          MOV AL, $A5
001148: A2 0B 38 0F 00  MOV [$000F380B], AL
00114D: B0 A5          MOV AL, $A5
00114F: A2 0C 38 0F 00  MOV [$000F380C], AL
001154: B0 BD          MOV AL, $BD
001156: A2 0D 38 0F 00  MOV [$000F380D], AL
00115B: B0 81          MOV AL, $81
00115D: A2 0E 38 0F 00  MOV [$000F380E], AL
001162: B0 FF          MOV AL, $FF
001164: A2 0F 38 0F 00  MOV [$000F380F], AL
T 001169: EB FE          JMP SHORT $00001169
(x86)> r eip 1100
(x86)> b 1169
(x86)> g
(x86)> m F0F20 1
000F0F20: 18 00 00 00 08 00 00 00 20 00 00 00 00 00 00 00 .....
(x86)> m F0F24 1
000F0F24: 08 00 00 00 20 00 00 00 00 00 00 00 06 00 00 00 ....
(x86)> m F3000 1
000F3000: 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
(x86)> m F3400 1
000F3400: 4E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 N.....
(x86)> m F3808 1
000F3808: FF 81 BD A5 A5 BD 81 FF 00 00 00 00 00 00 00 00 .....
(x86)> bc 1169

```

The first five stores configure TED video:

| Address | Value | Meaning  |
|---------|-------|--|
| \$F0F58 | \$01  | Enable TED video output.                             |
| \$F0F20 | \$18  | DEN + RSEL: display enabled, 25-row character field. |
| \$F0F24 | \$08  | CSEL: 40-column character field.                     |
| \$F0F30 | \$06  | Background colour 0.                                 |
| \$F0F40 | \$2E  | Border colour.                                       |

The next two stores set up the top-left cell. \$F3000 is the first byte of the video matrix, and \$F3400 is the matching colour RAM byte. The eight stores at \$F3808-\$F380F replace the glyph for character 1; the bytes form a small block motif.

After the breakpoint is reached, the top-left TED character cell shows that motif in colour \$4E on background colour \$06, with the border set to \$2E.

## 30.9 What comes next

---

Chapter 31 returns to material that applies across every CPU: how each processor handles timing, traps, and exceptions; how the shared timer counter is exposed to each one; and how the IRQ priority levels intersect with the auto-vector and MMU exception paths.