Materials

W8 Współczesna architektury procesorów i zestawy instrukcji: CISC i x86

Omawiane zagadnienia

- x86 architecture and instruction set
  - 8086: 16-bit world
  - 80386: the 32-bit world
  - Real and protected mode
  - The x86 register set
  - x86 addressing modes
  - x86 instruction categories
  - x86 instruction formats
- x64 architecture and instruction set
  - x64 instruction categories and formats

x86 architecture and instruction set

In 1978 Intel introduced the Intel 8086 processor. The 8088, released in 1979, is functionally very similar to the 8086, except it has an 8-bit data bus instead of the 16-bit bus of the 8086. The 8088 was the central processor in the original IBM PC. Subsequent generations of this processor series were named 80186, 80286, 80386, and 80486, leading to the term "x86" as shorthand for the family. Despite the fact that later subsequent generations dropped the numeric naming convention and were given the names Pentium, Core, i Series, Celeron, and Xeon the term "x86" is used to refere them describing their architecture as all of them are still backwards compatible and design of the newest chip inherits features of the most simple Intel 8086 which is not necessary a good thing.

That is why in most cases whenever you see the term *x86* it refers to the 16-bit and 32-bit instruction set architecture of the series of processors that began with the Intel 8086.

Note: AMD

Advanced Micro Devices (AMD), a semiconductor manufacturing company that competes with Intel, has been producing x86-compatible processors since 1982.

8086: 16-bit world

The 8086 and 8088 are 16-bit processors, despite the 8-bit data bus of the 8088. Internal registers in these processors are 16 bits wide and the instruction set operates on 16-bit data values. The 8088 **transparently executes** two bus cycles to transfer each 16-bit value between the processor and memory.

These processors have only 20 address lines, limiting the addressable memory to 1 MB. A 20-bit address cannot fit in a 16-bit register, so it is necessary to use a somewhat complicated system of segment registers and offsets to access the full 1 MB address space. This was discussed in the *Note: stack registers in action* of *Processor structure and function* chapter.

80386: the 32-bit world

In 1985, Intel released the 80386 with enhancements to meet the needs of programmers and users at that time. The 80386 introduced these features:

- **32-bit architecture**: Addresses, registers, and the ALU are 32 bits wide and instructions operate natively on operands up to 32 bits wide.
- **Protected mode**: This mode enables the multi-level privilege mechanism, consisting of ring numbers from 0 to 3. In Windows and Linux, ring 0 is kernel mode and ring 3 is user mode. Rings 1 and 2 are not used in these operating systems.
- **Huge, unique address space**: The 80386 on-chip MMU supports a **flat memory model**, enabling any location in the 4 GB ($2^{32} = 4294967296$) space to be accessed with a one unique 32-bit address. Manipulation of segment registers and offsets is no longer required. The MMU supports paged virtual memory.
- **3-stage instruction pipeline**: The pipeline, as you know from previous parts accelerates instruction execution.
- **Hardware debug registers**: The debug registers support setting up to four breakpoints that stop code execution at a specified virtual address when the address is accessed and a selected condition is satisfied. The available break conditions are:

  - code execution,
  - data write,
  - data read,
  - and data write.

  These registers are only available to code running in ring 0.

Real and protected mode

Modern x86 processors boot into the 16-bit operating mode of the original 8086, which is now called **real mode**. This mode retains compatibility with sofware written for the 8086/8088 environment, such as the MS-DOS operating system. In most modern systems running on x86 processors, a transition to protected mode occurs during system startup. Once in protected mode, the operating system generally remains in protected mode until the computer shuts down.

These days, the main use for 16-bit mode in x86 processors is to serve for a very short time as a bootloader for a protected mode operating system. Everything else takes place in a protected mode and it is very unlikely that you will need to implement any functionality in real mode. That is why it is justiffied to have in mind protected mode and the associated 32-bit flat memory model whenever you say *x86*.

The x86 architecture supports **unsigned and signed two's complement integer data** types with widths of 8 (named as *byte*), 16 (*word*), 32 (*doubleword*), 64 (*quadword*), and 128 (*double quadword*) bits. In most cases, the x86 architecture does not mandate storage of these data types on **natural boundaries**.

Note: natural boundary of a data type

The natural boundary of a data type is any address evenly divisible by the size of the data type in bytes.

Storing any of the multi-byte types at unaligned boundaries is permitted, but is discouraged because it causes a negative performance impact: instructions operating on unaligned data require additional clock cycles. A new instructions that operate on double quadwords require naturally aligned storage and will generate a general protection fault if unaligned access is attempted.

Modern x86 natively **supports floating-point** data types in widths of 16, 32, 64, and 80 bits. The 16-bit format is called half-precision floating-point and has an 11-bit mantissa, an implied leading 1 bit, and a 5-bit exponent. The half-precision floating-point format is used extensively in GPU processing.

Because numbers of type half are stored using 16 bits, they require less memory than numbers of type single, which uses 32 bits, or double, which uses 64 bits. However, because they are stored with fewer bits, numbers of type half are represented to less precision than numbers of type single or double.

The range, bias, and precision for supported floating-point data types are given in the table

below [3]:

Data type	Low limit	High limit	Exponent bias	Precision
Half	2^{-14} $\approx 6.1 \cdot 10^{-5}$	$(2 - 2^{-10}) \cdot 2^{15}$ $\approx 6.5 \cdot 104$	15	2^{-10} $\approx 10^{-3}$
Single	2^{-126} $\approx 10^{-38}$	2^{128} $\approx 3 \cdot 10^{38}$	127	2^{-23} $\approx 10^{-7}$
Double	2^{-1022} $\approx 2 \cdot 10^{-308}$	2^{1024} $\approx 2 \cdot 10^{308}$	1023	2^{-52} $\approx 10^{-16}$

The x86 register set

In protected mode, the x86 architecture has eight 32-bit wide general-purpose registers, a flags register, and an instruction pointer. There are also six segment registers and additional processor model-specific configuration registers. Nowadays the segment registers and model-specific registers are used only by very low leve code for example during startup and are, in general, not relevant to the developers of applications and device drivers.

The 16-bit general-purpose registers in the original 8086 architecture are named `AX`, `BX`, `CX`, `DX`, `SP`, `BP`, `SI`, and `DI`. All four `X` registers can be further accessed as individual bytes using the names `AH` (high-order byte) and `AL` (low-order byte) (see section *Example of microprocessor register organizations* of *Processor structure and function chapter*).

In the transition to the 32-bit architecture of the 80386, each register "grew" to 32 bits. The 32-bit version of a register's name is prefixed with the letter "E" to indicate this *extension*. It is possible to access portions of 32-bit registers in smaller bit widths: for example you still have an access to two low-order bytes of `EAX` via `AX` and (in case of `X` registers) also to those bytes via `AH` and `AL`.

```
33222222222111111111
10987654321098765432109876543210
|_____________EAX_____________|
               |______AX______|
               |__AH__||__AL__|
```

In keeping with the x86's complex instruction set computer (CISC) architecture, several functions associated with **various instructions are tied to specific registers**. The following table provides a description of the functions associated with each of the x86 general-purpose registers:

```
Register  Name           Function
EAX       Accumulator    Arithmetic operations
EBX       Base           Pointer to data
ECX       Counter        Loop counter, shift/rotate counter
EDX       Data           Arithmetic and I/O operations

ESP       Stack pointer  Pointer to the top of the stack
EBP       Base pointer   Pointer to the stack base within a function

ESI       Source index   Pointer to the source location in array
                         operation
EDI       Destination    Pointer to the destination location in array
          index          operation
```

These register-specific functions contrast with the architectures of many reduced instruction set computer (RISC) processors, which tend to provide a greater number of general-purpose registers. Registers within a RISC processor are, for the most part, functionally equivalent to one another.

Te x86 flags register, `EFLAGS`, contains the processor status bits (bits that are not listed in the table are reserved):

```
Bit Name Function
0   CF   Carry flag. Indicates if addition produced a carry
         or subtraction produced a borrow.
2   PF   Parity flag. Set if the low eight bits of the result
         contain an even number of 1 bits.
4   AF   Adjust flag. Indicates if addition produced a carry
         or subtraction produced a borrow from the lower 4 bits.
         Used in BCD arithmetic.
6   ZF   Zero flag. Set if the result of an opperation is zero.
7   SF   Sign flag. Set if the result of an opperation is negative.
8   TF   Trap flag. Used in single-step debugging.
9   IF   Interrupt enable flag. Settng this bit enables hardware
         interrupts.
10  DF   Direction flag. Controls the direction of string (block
         of data) processing. When set, the order is highest to
         lowest addresses.
11  OF   Overflow flag. Set if an operation resulted in a signed
         overflow.
12  IOPL (along with bit 13) I/O privilege level. The privilege
         level of the currently executing thread. IOPL 0 is kernel
         mode and 3 is user mode.
13  IOPL See above.
14  NT   Nested task flag. Controls the chaining of interrupts.
16  RF   Resume flag. Used for processing exceptions during
         debugging.
17  VM   Virtual 8086 mode flag. If set, 8086 compatibility mode
         is active. This mode allows running some MS-DOS
         applications in the context of a protected mode
         operating system.
18  AC   Alignment check flag. If set, memory alignment checking
         is active. For example, if the AC flag is set, storing
         a 16-bit value to an odd address triggers an Alignment Check
         exception. x86 processor can perform unaligned memory
         accesses when this flag is not set, but the number
         of instruction cycles required may increase.
19  VIF  Virtual interrupt flag. Virtual version of the IF flag
         in virtual 8086 mode.
20  VIP  Virtual interrupt pending flag. Set when an interrupt
         is pending in virtual 8086 mode.
21  ID   ID flag. If this bit can be set, the `cpuid` instruction
         returning processor identification and feature information,
         is supported.
```

Note: carry vs. overflow

Overflow flags get set when the register cannot properly represent the result as a signed

value (you overflowed into the sign bit).

Carry flags is set when is is not possible to represent the result as an unsigned value because result does not fit in this representation.

The 32-bit instruction pointer, `EIP`, contains the address of the next instruction to execute, unless a branch is taken. The x86 architecture is **little-endian**, meaning multi-byte values are stored in memory with the least significant byte at the lowest address (and the most significant byte at the highest address).

x86 addressing modes

As befits a a well-known representative of CISC architecture, x86 supports a variety of addressing modes. There are several rules associated with addressing source and destination operands that must be followed to create valid instructions.

For instance, the sizes of the source and destination operands of a instruction must be equal. The assembler will attempt to select a suitable size for an operand of ambiguous size (for example, an immediate value of 7) to match the width of a destination location (such as the 32-bit register EAX). In cases where the size of an operand cannot be inferred, size keywords must be provided.

The x86 architecture supports a variety of addressing modes, which we will look at now.

The assembly language in these examples uses the Intel syntax, which places the operands in destination-source order. The Intel syntax is used primarily in the Windows and MS-DOS contexts. An alternative notation, known as the AT&T syntax, places operands in source-destination order. Te AT&T syntax is used in Unix-based operating systems. All examples in this book will use the Intel syntax.

- Implied addressing

  The register is implied by the instruction opcode: `clc` – clear the carry flag `CF` in the `EFLAGS` register.

- Register addressing

  One or both source and destination registers are encoded in the instruction:

  `mov eax, ecx` – copy the contents of register `ECX` to `EAX`

- Immediate addressing

  An immediate value is provided as an instruction operand:

`mov eax, 7` – move the 32-bit value 7 into `EAX`

`mov ax, 7` – move the 16-bit value 7 into `AX` (the lower 16 bits of `EAX`)

- Direct memory addressing

  The address of the value is provided as an instruction operand:

  `mov eax, [078bch]` – copy the 32-bit value at hex address `78BC` to `EAX`.

  In x86 assembly code, square brackets around an expression indicate the expression is an address.

- Register indirect addressing

  The operand identifies a register containing the address of the data value:

  `mov eax, [esi]` - copy the 32-bit value at the address contained in `ESI` to `EAX`.

- Indexed addressing

  The operand indicates a register plus offset that calculates the address of the data value:

  `mov eax, [esi + 0bh]` – copy the 32-bit value at the address `ESI + 0bh` to `EAX`

  This mode is useful for accessing the elements of a data structure. In such scenario, the `ESI` register contains the address of the structure and the added constant is the byte offset of the element from the beginning of the structure.

- Based indexed addressing

  The operand indicates a base register, an index register, and an offset that sum together to calculate the address of the data value:

  `mov eax, [ebx + esi + 10]` - copy the 32-bit value starting at the address `EBX + ESI + 10` to `EAX`

  This mode is useful for accessing individual data elements within an array of structures. In this example, the `EBX` register contains the address of the beginning of the structure array, `ESI` contains the offset of the referenced structure within the array, and the constant value `10` is the offset of the desired element from the beginning of the selected structure.

- Based indexed addressing with scaling

The operand is composed of a base register, an index register multiplied by a scale factor, and an offset that sum together to calculate the address of the data value:

`mov eax, [ebx + esi*4 + 10]` - copy the 32-bit value starting at the address `EBX + ESI*4 + 10` to `EAX`.

In this mode, the value in the index register can be multiplied by 1 (the default), 2, 4, or 8 before being summed with the other components of the operand address. There is no performance penalty associated with using the scaling multiplier.

This feature is helpful when iterating over arrays containing elements with sizes of 2, 4, or 8 bytes. Most of the general-purpose registers can be used as the base or index register in the based addressing modes.

x86 instruction categories

The x86 instruction set was introduced with the Intel 8086 and has been extended several times over the years. Some of the most significant changes relate to the extension of the architecture from 16 to 32 bits, which added protected mode and paged virtual memory. In almost all cases, the new capabilities were added while retaining full backward compatibility.

The x86 instruction set contains several hundred instructions. It can be divided into a few general categories:

- data movement,
- stack manipulation,
- arithmetic and logic,
- conversions,
- control flows,
- string and flag manipulations,
- input/output,
- and protected mode.

- Data movement

  Data movement instructions do not affect the processor flags.

  `cmovcc` Conditionally moves the second operand's data to the register provided as the first operand if the `cc` condition is true. The condition is determined from one or more of the processor flags: `CF`, `ZF`, `SF`, `OF`, and `PF`. The condition codes are `e` (equal), `ne` (not equal), `g` (greater), `ge` (greater or equal), `a` (above), `ae` (above or equal), `l` (less), `le` (less or equal), `b` (below), `be` (below or equal), `o` (overflow), `no` (no overflow), `z` (zero), `nz` (not zero), `s` (`SF=1`), `ns` (`SF=0`), `cxz` (register `CX` is zero), and `ecxz` (the `ECX` register is zero).

`cmovnae dx, ax` – move data from `AX` to `DX` if *not above or equal* (`CF=1`).

To have flags set, before executing `cmovcc` , you very often make a comparision of two values with the `cmp` instruction:

```
cmp minuend, subtrahend  ; (Intel syntax)
```

`cmp` performs a comparison operation between *minuend* and *subtrahend*. The comparison is performed by a (signed) subtraction of subtrahend from minuend, the results of which can be called *difference*. Difference is then discarded as it is neede only to set the `EFLAGS` register.

- Stack manipulation

  Stack manipulation instructions do not affect the processor flags.

  `push eax` Decrements the stack pointer and then stores the source operand on the top of the stack. In this case decrements `ESP` by 4 (because 32 bit are needed to preserve value form 32-bit `EAX` register), then places the 32-bit operand into the stack location pointed to by `ESP` .

- Arithmetic and logic

  Arithmetic and logic instructions modify the processor flags.

  `mul cx` Performs an unsigned integer multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register `AL` , `AX` or `EAX` (depending on the size of the second operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size. A *byte* operand is multiplied by `AL` and the result is placed in `AX` . A *word* operand is multiplied by `AX` and the result is placed in `DX:AX` , with the upper 16 bits in `DX` . A *doubleword* is multiplied by `EAX` and the result is placed in `EDX:EAX` .

  `sal al, 2` Performs an arithmetic shift of the argument left by 1 to 31 bit positions. An arithmetic shift preserves the number's sign and clear the least significant bit:

  ```
  mov al, -40 ;  -40_(10) = 11011000_(2)
  sal al, 2   ;  11 (111000 00)_(2) = -32_(10)
  ```

- Conversions

Conversion instructions extend a smaller data size to a larger size.

`cbw` Converts a byte (register `AL`) into a word (`AX`). In this instruction the (source) operand is an implied operand located in register `AL`, `AX` or `EAX`.

- Control fow

Control flow instructions conditionally or unconditionally transfer execution to an address other than that resulting from the natural order of instruction execution, i.e. other than `IP+1`.

`jmp address` Transfers control to the instruction at the address provided as the operand, `address`.

```
label: inc ax
       ...
       do processing
       ...
       jmp label
```

`jcc` Transfers control to the instruction at the address provided as the operand if the condition `cc` is true. The condition is determined from one or more of the processor flags: `CF`, `ZF`, `SF`, `OF`, and `PF` as it was described in case of `cmovcc`.

```
ja label ; Jump if Above (unsigned comparison)
         ; Jump to address given by label
         ; if CF = 0 and ZF = 0
```

Loads `EIP` with the specified address, if the minuend of the previous `cmp` instruction is greater than the subtrahend:

```
difference = minuend - subtrahend
if difference > 0 then
    EIP = label
```

For example the following piece of code always jumps except the case when `rbx` is -1, too:

```
mov rax, -1    ;  rax := -1
cmp rax, rbx
ja label
```

This is because -1 is represented as all bits set in the two's complement. Interpreting all bits set, without treating any bit as the sign, has the largest unsigned value a register can hold.

`call` Pushes the current value of `EIP` onto the stack and transfers control to the instruction at the address provided as the operand.

`ret` Pops the top-of-stack value and stores it in `EIP`. If an operand is provided, it pops the given number of bytes from the stack to clear parameters.

```
loop:     call function
          ...
          do processing in a loop
          ...
          jmp loop
function: ...
          do processing in a function
          ...
          ret
```

`loop` Decrements the loop counter in `ECX` and, if not zero, transfers control to the instruction at the address provided as the operand.

Here is a loop that repeats ten times:

```
       mov ecx, 10
       mov eax, 0
start: inc eax
       ...
       do processing
       ...
       loop start      ; Jump to start
```

When you exit loop, `eax=10` and `ecx=0`. Note that you have to be careful not to change `ecx` inside the loop by mistake. The `loop` instruction decrements automatically `ecx`. If you touch `ecx` inside loop, remember to restore its value before every end of iteration inside loop. See example below:

```
       mov ecx, 10   ; Initialize outer loop
loop1: ...           ; Start inner loop
      do processing in outer loop
      ...
      push ecx      ; Save ECX value used by outer loop
      mov ecx, 5    ; Initialize inner loop

loop2: ...           ; Start inner loop
      do processing in inner loop
      ...
      loop loop2    ; End inner loop

      pop ecx       ; Restore ECX value to be used by outer loop
      ...
      continue processing in outer loop
      ...
      loop loop1    ; End outer loop
```

- String manipulation

  `cmpsb` Compares value (byte, for word use `cmpsw`) at memory location referenced by `DS:SI` with value at `ES:DI`, sets appropriate flags depending of result and, depending of value of `DF` (direction flag) before instruction is executed either increase `SI` and `DI` registers by one (when `DF` is set) or decrease `SI` and `DI` registers by one (when `DF` is cleared). `cmps` is usually used in conjuction with one of `rep` family prefix and length of string/array in `CX` register.

  The `rep` (repeat), `repe` (repeat while equal), `repne` (repeat while not equal), `repz` (repeat while zero), and `repnz` (repeat while not zero) mnemonics are prefixes resulting in repeating a string instruction the number of times specified in the count register `(E)CX` or until the indicated condition of the `ZF` flag is no longer met.

  If you want to compare two buffers, each of 64 bytes (one starting at `DS:1234`, another starting at `ES:5678`) in size you will have something like below:

```
mov SI, 1234
mov DI, 5678
mov CX, 64
cld ; clear DF
repe cmpsb
```

  Above code will execute until difference is found or until it compare all 64 bytes.

- Flag manipulation

Flag manipulation instructions modify bits in the `EFLAGS` register.

`stc`, `clc`, `cmc` Sets, clears, or complements the carry flag bit, `CF`.

- Input/output

  Input/output instructions read data from or write data to peripheral devices.

  `in dest, src`, `out dest, src` Moves 1, 2, or 4 bytes between `AL`, `AX`, or `EAX` and an I/O port, depending on the operand size.

  The `in` instruction almost always has the operands `AX` and `DX` associated with it. `DX` (src) frequently holds the *port address to read*, and `AX` (dest) *receives the data from the port*.

  ```
  in where_to_put_data, port_to_read
  ```

  The `out` instruction is very similar to the `in` instruction. `out` outputs data from a given register (src) to a given output port (dest).

  ```
  out port_to_write, data_to_output
  ```

  In protected mode, both instructions are frequently locked so normal users can't use it.

  ```
  .equ RTCaddress, 0x70
  .equ RTCdata, 0x71

  ; Read seconds from RTC
  mov al, 0
  out RTCaddress, al
  in al, RTCdata         ; al contains seconds.

  ; Read minutes from RTC
  mov al, 0x02
  out RTCaddress, al
  in al, RTCdata         ; al contains minutes.

  ; Read hours from RTC
  mov al, 0x04
  out RTCaddress, al
  in al, RTCdata         ; al contains hour.
  ```

- Protected mode

The following instructions access the features of the protected mode:

`sysenter`, `sysexit` Transfers control from ring 3 to ring 0 (`sysenter`) or from ring 0 to ring 3 (`sysexit`) in Intel processors.

`syscall`, `sysret` Transfers control from ring 3 to ring 0 (`syscall`) or from ring 0 to ring 3 (`sysret`) in AMD processors.

- Miscellaneous instructions

  `int` Initiates a sofware interrupt. The operand is the interrupt vector number.

  `nop` No operation.

  `cpuid` Provides information about the processor model and its capabilities.

- Other instruction categories

  - Floating-point instructions
  - SIMD instructions: This category includes the MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2, and AVX-512 instructions.
  - AES instructions: These instructions support encryption and decryption using the Advanced Encryption Standard (AES).
  - MPX instructions: The memory protection extensions (MPX) enhance memory integrity by preventing errors such as buffer overruns.
  - SMX instructions: The safer mode extensions (SMX) improve system security.

  Additional processor registers are provided for use by the floating-point and SIMD instructions:

  FPU Register Stack

  The FPU has an array of **eight registers** that can be accessed as a stack. There is one top index indicating the current top of the stack. Pushing or popping items to or from the stack will only change the top index and store or wipe data respectively.

  `st(0)` or simply `st` refers to the register that is currently at the top of the stack. If eight values were stored on the stack, `st(7)` refers to last element on the stack (i. e. the bottom).

  Numbers are pushed onto the stack from memory, and are popped off the stack back to memory. There is no instruction allowing to transfer values directly to or from ALU registers. The x87 stack can only be accessed by FPU instructions - you cannot write `mov eax, st(0)` - it is necessary to store values to memory if you want to print them, for example.

FPU instructions generally will pop the first two items off the stack, act on them, and push the answer back on to the top of the stack.

MMX Registers

MMX is a supplemental instruction set introduced by Intel in 1996. Most of the new instructions are *single instruction, multiple data* (SIMD), meaning that single instructions work with multiple pieces of data in parallel.

There are **eight** 64-bit MMX registers (form `MM0` to `MM7`). To avoid having to add new registers, they were made to overlap with the FPU stack register. This means that **the MMX instructions and the FPU instructions cannot be used simultaneously**. MMX registers are addressed directly, and do not need to be accessed by pushing and popping in the same way as the FPU registers.

Usually when you initiate an assembly block in your code that contains MMX instructions, the CPU automatically will disallow floating point instructions. To re-allow FPU operations you must end all MMX code with `emms`.

Note: saturation arithmetic

If you ues a regular register like `AX` or `BX` and you add 1 to 255 you get 0. This is because the regular registers "roll-over" to the next value. MMX registers get around this by a technique called **Saturation Arithmetic**. In saturation arithmetic, the value of the register never rolls over to 0 again. This means that in the MMX world, we have the following equations:

```
255 + 100 = 255
200 + 100 = 255
0 - 100 = 0;
99 - 100 = 0;
```

This may seem counter-intuitive at first to people who are used to their registers rolling over, but it makes sense in some situations, especially if you work with multimedia data and for example try to make white brighter - it should never become black.

SSE Registers

SSE, introduced by Intel in 1999 with the Pentium III, creates eight new 128-bit registers (from `XMM0` to `XMM7`). SSE stands for *Streaming SIMD Extensions* and it is essentially the floating-point equivalent of the MMX instructions.

Originally, an SSE register could only be used as four 32-bit single precision floating

point numbers (the equivalent of a float in C). SSE2 expanded the capabilities of the XMM registers, so they can now be used as:

- 2 64-bit floating points (double precision)
- 2 64-bit integers
- 4 32-bit floating points (single-precision)
- 4 32-bit integers
- 8 16-bit integers
- 16 8-bit characters (bytes)

x86 instruction formats

Individual x86 instructions are of variable length, and can range in size from 1 to 15 bytes. The components of a single instruction, including any optional bytes, are laid out in memory in the following sequence [4]:

- **Opcode with prefixex bytes**: One or more optional prefix bytes provide auxiliary opcode execution information.
- **Opcode bytes** An x86 opcode, consisting of 1 to 3 bytes, follows any prefix bytes. For some opcodes, an additional 3 opcode bits are encoded in a ModR/M byte following the opcode.
- **ModR/M byte**: Not all instructions require this byte. The ModR/M byte contains three information fields providing address mode and operand register information. The upper two bits of this byte (the Mod field) and the lower three bits (the R/M field) combine to form a 5-bit field with 32 possible values. Of these, 8 values identify register operands and the other 24 specify addressing modes. The remaining 3 bits (the reg/opcode field) either indicate a register or provide three additional opcode bits, depending on the instruction.
- **Address displacement bytes**: Either 0, 1, 2, or 4 bytes provide an address displacement (offset added to the calculated address) used in computing the operand address.
- **Immediate value bytes**: If the instruction includes an immediate value, it is located in the last 1, 2, or 4 bytes of the instruction.

The variable-length nature of x86 instructions makes the process of instruction decoding quite complex. It is also challenging for debugging tools to disassemble a sequence of instructions in reverse order, perhaps to display the code leading up to a breakpoint. This difficulty arises because it is possible for a trailing subset of bytes within a lengthy instruction to form a complete, valid instruction. This complexity is a notable difference from the more regular instruction formats used in RISC architectures.

x64 architecture and instruction set

The original specification for a processor architecture extending the x86 processor and instruction set to 64 bits, named AMD64, was introduced by AMD in 2000. The first AMD64 processor, the Opteron, was released in 2003. Intel found itself following AMD's lead and developed an AMD64-compatible architecture, eventually given the name Intel 64. The first Intel processor that implemented their 64-bit architecture was the Xeon, introduced in 2004. The name of the architecture shared by AMD and Intel came to be called x86-64, reflecting the evolution of x86 to 64 bits, and in popular usage, this term has been shortened to x64.

The first Linux version supporting the x64 architecture was released in 2001, well before the first x64 processors were even available. Windows began supporting the x64 architecture in 2005. Processors implementing the AMD64 and Intel 64 architectures are largely compatible at the instruction level of user mode programs. There are a few differences between the architectures, the most significant of which is the difference in support of the `sysenter` / `sysexit` Intel instructions and the `syscall` / `sysret` AMD instructions. In general, operating systems and programming language compilers manage these differences, making them rarely an issue of concern to sofware and system developers. Developers of kernel sofware, drivers, and assembly code must take these differences into account.

The principal features of the x64 architecture are as follows:

- x64 is a mostly compatible 64-bit extension of the 32-bit x86 architecture. Most sofware, particularly user mode applications, written for the 32-bit environment should execute without modification in a processor running in 64-bit mode. 64-bit mode is also referred to as **long mode**.
- The eight 32-bit general-purpose registers of x86 are extended to 64 bits in x64. The register name prefix `R` indicates 64-bit registers. For example, in x64, the extended x86 `EAX` register is called `RAX`. The x86 register subcomponents `EAX`, `AX`, `AH`, and `AL` are still available in x64.
- The instruction pointer, `RIP`, is now 64 bits. The flags register, `RFLAGS`, also extends to 64 bits, though the upper 32 bits are reserved. The lower 32 bits of `RFLAGS` are the same as `EFLAGS` in the x86 architecture.
- Eight 64-bit general-purpose registers have been added, named `R8` through `R15`. The new 64-bit registers can be accessed in smaller widths using the appropriate suffix letter:
    - Suffix `D` accesses the lower 32 bits of the register: `R11D`.
    - Suffix `W` accesses the lower 16 bits of the register: `R11W`.
    - Suffix `B` accesses the lower 8 bits of the register: `R11B`.

    Unlike the x86 registers, the new registers in the x64 architecture are truly general purpose and do not perform any special functions at the processor instruction level.

- 64-bit integers are supported as a native data type.
- x64 processors retain the option of running in x86 compatibility mode. This mode enables the use of 32-bit operating systems and allows any application built for x86 to run on x64 processors. In 32-bit compatibility mode, the 64-bit extensions are unavailable.

Virtual addresses in the x64 architecture are 64 bits wide, supporting an address space of 16 exbiytes (EiB), equivalent to 2^{64} bytes. Current processors from AMD and Intel, however, support only 48 bits of virtual address space. This restriction reduces processor hardware complexity while still supporting up to 256 tebibytes (TiB) of virtual address space. Current-generation processors also support a maximum of 48 bits of physical address space. This permits a processor to address 256 TiB of physical RAM, though modern motherboards do not support the quantity of DRAM devices such a system would require.

x64 instruction categories and formats

The x64 architecture implements essentially the same instruction set as x86, with 64-bit extensions. When operating in 64-bit mode, the x64 architecture uses a default address size of 64 bits and a default operand size of 32 bits. A new opcode prefix byte, `rex`, is provided to specify the use of 64-bit operands. The format of x64 instructions in memory matches that of the x86 architecture, with some minor exceptions that are not significant at this stage of considerations. The addition of support for the `rex` prefix byte is the most significant variation from the x86 instruction format. Address displacements and immediate values within some instructions can be 64 bits wide, in addition to all the bit widths supported in x86. Although it is possible to define instructions longer than 15 bytes, the processor instruction decoder will raise a general protection fault if an attempt is made to decode an instruction longer than 15 bytes.

Bibliography

1. William Stallings, *Computer Organization and Architecture*, Pearson, 11th edition (2019)
2. Jim Ledin, *Modern Computer Architecture and Organization*, Packt, April 2020
3. What is Half Precision? https://www.mathworks.com/help/coder/ug/what-is-half-precision.html, Retrieved 2023- 12-02.
4. X86-64 Instruction Encoding https://wiki.osdev.org/X86-64*Instruction*Encoding, Retrieved 2023- 12-03.