x86 reference

Written by Peter S in 16 Apr 2021

x86 registers reference table

Size	4bit	8bit	16bit	32bit	64bit
Accumulator	AL	AH	AX	EAX	RAX
Base index[1]	BL	BH	BX	EBX	RBX
Counter[2]	CL	CH	CX	ECX	RCX
Data[3]	DL	DH	DX	EDX	RDX
Source index	*	*	SI	ESI	RSI
Destination index	*	*	DI	EDI	RDI
Stack pointer	*	*	SP	ESP	RSP
Stack base pointer	*	*	BP	EBP	RBP
Instruction pointer	*	*	IP	EIP	RIP

x86 instructions reference

Syntax used

<reg32>	Any 32-bit register (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP)
<reg16>	Any 16-bit register (AX, BX, CX, or DX)
<reg8>	Any 8-bit register (AH, BH, CH, DH, AL, BL, CL, or DL)
<reg>	Any register
<mem>	A memory address (e.g., [eax], [var + 4], or dword ptr [eax+ebx])
<con32>	Any 32-bit constant
<con16>	Any 16-bit constant
<con8>	Any 8-bit constant
<con>	Any 8-, 16-, or 32-bit constant

1 Used for arrays.
2 Used for loops and strings.
3 Extend the precision of the accumulator.

Data movement

mov — Move (Opcodes: 88, 89, 8A, 8B, 8C, 8E, ...)

The mov instruction copies the data item referred to by its second operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its first operand (i.e. a register or memory). While register-to-register moves are possible, direct memory-to-memory moves are not. In cases where memory transfers are desired, the source memory contents must first be loaded into a register, then can be stored to the destination memory address.

Syntax
mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>
mov <reg>,<const>
mov <mem>,<const>

Examples
mov eax, ebx — copy the value in ebx into eax
mov byte ptr [var], 5 — store the value 5 into the byte at location var

push — Push stack (Opcodes: FF, 89, 8A, 8B, 8C, 8E, ...)

The push instruction places its operand onto the top of the hardware supported stack in memory. Specifically, push first decrements ESP by 4, then places its operand into the contents of the 32-bit location at address [ESP]. ESP (the stack pointer) is decremented by push since the x86 stack grows down - i.e. the stack grows from high addresses to lower addresses.

Syntax
push <reg32>
push <mem>
push <con32>

Examples
push eax — push eax on the stack
push [var] — push the 4 bytes at address *var* onto the stack

pop — Pop stack

The pop instruction removes the 4-byte data element from the top of the hardware-supported stack into the specified operand (i.e. register or memory location). It first moves the 4 bytes located at memory location [SP] into the specified register or memory location, and then increments SP by 4.

Syntax
pop <reg32>
pop <mem>

Examples
pop edi — pop the top element of the stack into EDI.
pop [ebx] — pop the top element of the stack into memory at the four bytes starting at location EBX.

lea — Load effective address

The lea instruction places the *address* specified by its second operand into the register specified by its first operand. Note, the *contents* of the memory location are not loaded, only the effective address is computed and placed into the register. This is useful for obtaining a pointer into a memory region.

Syntax
lea <reg32>,<mem>

Examples
lea edi, [ebx+4*esi] — the quantity EBX+4*ESI is placed in EDI.
lea eax, [var] — the value in *var* is placed in EAX.
lea eax, [val] — the value *val* is placed in EAX.

Arithmetic and logic

add — Integer Addition

The add instruction adds together its two operands, storing the result in its first operand. Note, whereas both operands may be registers, at most one operand may be a memory location.

Syntax
add <reg>,<reg>
add <reg>,<mem>
add <mem>,<reg>
add <reg>,<con>
add <mem>,<con>

Examples
add eax, 10 — EAX ← EAX + 10
add BYTE PTR [var], 10 — add 10 to the single byte stored at memory address var

sub — Integer Subtraction

The sub instruction stores in the value of its first operand the result of subtracting the value of its second operand from the value of its first operand. As with add

Syntax
sub <reg>,<reg>
sub <reg>,<mem>
sub <mem>,<reg>

sub <reg>,<con>
sub <mem>,<con>

Examples
sub al, ah — AL ← AL - AH
sub eax, 216 — subtract 216 from the value stored in EAX

inc, dec — Increment, Decrement

The inc instruction increments the contents of its operand by one. The dec instruction decrements the contents of its operand by one.

Syntax
inc <reg>
inc <mem>
dec <reg>
dec <mem>

Examples
dec eax — subtract one from the contents of EAX.
inc DWORD PTR [var] — add one to the 32-bit integer stored at location *var*

imul — Integer Multiplication

The imul instruction has two basic formats: two-operand (first two syntax listings above) and three-operand (last two syntax listings above).

The two-operand form multiplies its two operands together and stores the result in the first operand. The result (i.e. first) operand must be a register.

The three operand form multiplies its second and third operands together and stores the result in its first operand. Again, the result operand must be a register. Furthermore, the third operand is restricted to being a constant value.

Syntax
imul <reg32>,<reg32>
imul <reg32>,<mem>
imul <reg32>,<reg32>,<con>
imul <reg32>,<mem>,<con>

Examples

imul eax, [var] — multiply the contents of EAX by the 32-bit contents of the memory location *var*. Store the result in EAX.
imul esi, edi, 25 — ESI → EDI * 25

idiv — Integer Division

The idiv instruction divides the contents of the 64 bit integer EDX:EAX (constructed by viewing EDX as the most significant four bytes and EAX as the least significant four bytes) by the specified operand value. The quotient result of the division is stored into EAX, while the remainder is placed in EDX.

Syntax
idiv <reg32>
idiv <mem>

Examples

idiv ebx — divide the contents of EDX:EAX by the contents of EBX. Place the quotient in EAX and the remainder in EDX.
idiv DWORD PTR [var] — divide the contents of EDX:EAX by the 32-bit value stored at memory location *var*. Place the quotient in EAX and the remainder in EDX.
and, or, xor — Bitwise logical and, or and exclusive or

These instructions perform the specified logical operation (logical bitwise and, or, and exclusive or, respectively) on their operands, placing the result in the first operand location.

Syntax
and <reg>,<reg>
and <reg>,<mem>
and <mem>,<reg>
and <reg>,<con>
and <mem>,<con>

or <reg>,<reg>
or <reg>,<mem>
or <mem>,<reg>
or <reg>,<con>
or <mem>,<con>

xor <reg>,<reg>
xor <reg>,<mem>
xor <mem>,<reg>
xor <reg>,<con>
xor <mem>,<con>

Examples
and eax, 0fH — clear all but the last 4 bits of EAX.
xor edx, edx — set the contents of EDX to zero.

not — Bitwise Logical Not

Logically negates the operand contents (that is, flips all bit values in the operand).

Syntax
not <reg>
not <mem>

Example
not BYTE PTR [var] — negate all bits in the byte at the memory location *var*.

neg — Negate

Performs the two's complement negation of the operand contents.

Syntax
neg <reg>
neg <mem>

Example
neg eax — EAX → - EAX

shl, shr — Shift Left, Shift Right

These instructions shift the bits in their first operand's contents left and right, padding the resulting empty bit positions with zeros. The shifted operand can be shifted up to 31 places. The number of bits to shift is specified by the second operand, which can be either an 8-bit constant or the register CL. In either case, shifts counts of greater then 31 are performed modulo 32.

Syntax
shl <reg>,<con8>
shl <mem>,<con8>
shl <reg>,<cl>
shl <mem>,<cl>

shr <reg>,<con8>
shr <mem>,<con8>
shr <reg>,<cl>
shr <mem>,<cl>

Examples

shl eax, 1 — Multiply the value of EAX by 2 (if the most significant bit is 0)

shr ebx, cl — Store in EBX the floor of result of dividing the value of EBX by 2^n where*n* is the value in CL.

Flow control

jmp — Jump

Transfers program control flow to the instruction at the memory location indicated by the operand.

Syntax
jmp <label>

Example
jmp begin — Jump to the instruction labeled begin.

j*condition* — Conditional Jump

These instructions are conditional jumps that are based on the status of a set of condition codes that are stored in a special register called the *machine status word*. The contents of the machine status word include information about the last arithmetic operation performed. For example, one bit of this word indicates if the last result was zero. Another indicates if the last result was negative. Based on these condition codes, a number of conditional jumps can be performed. For example, the jz instruction performs a jump to the specified operand label if the result of the last arithmetic operation was zero. Otherwise, control proceeds to the next instruction in sequence.

A number of the conditional branches are given names that are intuitively based on the last operation performed being a special compare instruction, cmp (see below). For example, conditional branches such as jle and jne are based on first performing a cmp operation on the desired operands.

Syntax
je <label> (jump when equal)
jne <label> (jump when not equal)
jz <label> (jump when last result was zero)
jg <label> (jump when greater than)
jge <label> (jump when greater than or equal to)
jl <label> (jump when less than)
jle <label> (jump when less than or equal to)

Example
cmp eax, ebx
jle done

If the contents of EAX are less than or equal to the contents of EBX, jump to the label *done*. Otherwise, continue to the next instruction.

cmp — Compare

Compare the values of the two specified operands, setting the condition codes in the machine status word appropriately. This instruction is equivalent to the sub instruction, except the result of the subtraction is discarded instead of replacing the first operand.

Syntax
cmp <reg>,<reg>
cmp <reg>,<mem>
cmp <mem>,<reg>
cmp <reg>,<con>

Example
cmp DWORD PTR [var], 10
jeq loop

If the 4 bytes stored at location *var* are equal to the 4-byte integer constant 10, jump to the location labeled *loop*.

call, **ret** — Subroutine call and return

These instructions implement a subroutine call and return. The call instruction first pushes the current code location onto the hardware supported stack in memory (see the push instruction for details), and then performs an unconditional jump to the code location indicated by the label operand. Unlike the simple jump instructions, the call instruction saves the location to return to when the subroutine completes.

The ret instruction implements a subroutine return mechanism. This instruction first pops a code location off the hardware supported in-memory stack (see the pop instruction for details). It then performs an unconditional jump to the retrieved code location.

Syntax
call <label>
ret

References

Wikipedia, x86, https://en.wikipedia.org/wiki/X86

University of Virginia Computer Science, x86 Assembly Guide, 19 nov 2018, https://www.cs.virginia.edu/~evans/cs216/guides/x86.html#instructions