x86 Instructions

05/23/2017 • 9 minutes to read •

In this article

In the lists in this section, instructions marked with an asterisk (*****) are particularly important. Instructions not so marked are not critical.

On the x86 processor, instructions are variable-sized, so disassembling backward is an exercise in pattern matching. To disassemble backward from an address, you should start disassembling at a point further back than you really want to go, then look forward until the instructions start making sense. The first few instructions may not make any sense because you may have started disassembling in the middle of an instruction. There is a possibility, unfortunately, that the disassembly will never synchronize with the instruction stream and you will have to try disassembling at a different starting point until you find a starting point that works.

For well-packed **switch** statements, the compiler emits data directly into the code stream, so disassembling through a **switch** statement will usually stumble across instructions that make no sense (because they are really data). Find the end of the data and continue disassembling there.

Instruction Notation

The general notation for instructions is to put the destination register on the left and the source on the right. However, there can be some exceptions to this rule.

Arithmetic instructions are typically two-register with the source and destination registers combining. The result is stored into the destination.

Some of the instructions have both 16-bit and 32-bit versions, but only the 32-bit versions are listed here. Not listed here are floating-point instructions, privileged instructions, and instructions that are used only in segmented models (which Microsoft Win32 does not use).

To save space, many of the instructions are expressed in combined form, as shown in the following example.

*	MOV	**r1**, **r**/m/#n	**r1** = **r**/m/#n

means that the first parameter must be a register, but the second can be a register, a memory reference, or an immediate value.

To save even more space, instructions can also be expressed as shown in the following.

*	MOV	**r1**/m, **r**/m/#n	**r1**/m = **r**/m/#n

which means that the first parameter can be a register or a memory reference, and the second can be a register, memory reference, or immediate value.

Unless otherwise noted, when this abbreviation is used, you cannot choose memory for both source and destination.

Furthermore, a bit-size suffix (8, 16, 32) can be appended to the source or destination to indicate that the parameter must be of that size. For example, r8 means an 8-bit register.

Memory, Data Transfer, and Data Conversion

Memory and data transfer instructions do not affect flags.

Effective Address

*	LEA	**r**, m	Load effective address. (r = address of m)

For example, **LEA eax, [esi+4]** means **eax** = **esi** + 4. This instruction is often used to perform arithmetic.

Data Transfer

	MOV	**r1**/m, **r2**/m/#n	**r1**/m = **r**/m/#n
	MOVSX	**r1**, **r**/m	Move with sign extension.
*	MOVZX	**r1**, **r**/m	Move with zero extension.

MOVSX and **MOVZX** are special versions of the **mov** instruction that perform sign extension or zero extension from the source to the destination. This is the only instruction that allows the source and destination to be different sizes. (And in fact, they must be different sizes.

Stack Manipulation

The stack is pointed to by the **esp** register. The value at **esp** is the top of the stack (most recently pushed, first to be popped); older stack elements reside at higher addresses.

	PUSH	**r**/m/#n	Push value onto stack.
	POP	**r**/m	Pop value from stack.
	PUSHFD		Push flags onto stack.
	POPFD		Pop flags from stack.
	PUSHAD		Push all integer registers.
	POPAD		Pop all integer registers.
	ENTER	#n, #n	Build stack frame.
*	LEAVE		Tear down stack frame

The C/C++ compiler does not use the **enter** instruction. (The **enter** instruction is used to implement nested procedures in languages like Algol or Pascal.)

The **leave** instruction is equivalent to:

asm Copy
```asm
mov esp, ebp
pop ebp
```

Data Conversion

CBW	Convert byte (**al**) to word (**ax**).

CWD	Convert word (**ax**) to dword (**dx**:**ax**).
CWDE	Convert word (**ax**) to dword (**eax**).
CDQ	convert dword (**eax**) to qword (**edx**:**eax**).

All conversions perform sign extension.

Arithmetic and Bit Manipulation

All arithmetic and bit manipulation instructions modify flags.

Arithmetic

	ADD	**r1**/m, **r2**/m/#n	**r1**/m += **r2**/m/#n
	ADC	**r1**/m, **r2**/m/#n	**r1**/m += **r2**/m/#n + carry
	SUB	**r1**/m, **r2**/m/#n	**r1**/m -= **r2**/m/#n
	SBB	**r1**/m, **r2**/m/#n	**r1**/m -= **r2**/m/#n + carry
	NEG	**r1**/m	**r1**/m = -**r1**/m
	INC	**r**/m	**r**/m += 1
	DEC	**r**/m	**r**/m -= 1
	CMP	**r1**/m, **r2**/m/#n	Compute **r1**/m - **r2**/m/#n

The **cmp** instruction computes the subtraction and sets flags according to the result, but throws the result away. It is typically followed by a conditional **jump** instruction that tests the result of the subtraction.

	MUL	**r**/m8	**ax** = **al** * **r**/m8
	MUL	**r**/m16	**dx**:**ax** = **ax** * **r**/m16
	MUL	**r**/m32	**edx**:**eax** = **eax** * **r**/m32

	IMUL	**r**/m8	**ax** = **al** * **r**/m8
	IMUL	**r**/m16	**dx**:**ax** = **ax** * **r**/m16
	IMUL	**r**/m32	**edx**:**eax** = **eax** * **r**/m32
	IMUL	**r1**, **r2**/m	**r1** *= **r2**/m
	IMUL	**r1**, **r2**/m, #n	**r1** = **r2**/m * #n

Unsigned and signed multiplication. The state of flags after multiplication is undefined.

	DIV	**r**/m8	(**ah**, **al**) = (**ax** % **r**/m8, **ax** / **r**/m8)
	DIV	**r**/m16	(**dx**, **ax**) = **dx**:**ax** / **r**/m16
	DIV	**r**/m32	(**edx**, **eax**) = **edx**:**eax** / **r**/m32
	IDIV	**r**/m8	(**ah**, **al**) = **ax** / **r**/m8
	IDIV	**r**/m16	(**dx**, **ax**) = **dx**:**ax** / **r**/m16
	IDIV	**r**/m32	(**edx**, **eax**) = **edx**:**eax** / **r**/m32

Unsigned and signed division. The first register in the pseudocode explanation receives the remainder and the second receives the quotient. If the result overflows the destination, a division overflow exception is generated.

The state of flags after division is undefined.

*	SET*cc*	**r**/m8	Set **r**/m8 to 0 or 1

If the condition *cc* is true, then the 8-bit value is set to 1. Otherwise, the 8-bit value is set to zero.

Binary-coded Decimal

You will not see these instructions unless you are debugging code written in COBOL.

	DAA	Decimal adjust after addition.
	DAS	Decimal adjust after subtraction.

These instructions adjust the **al** register after performing a packed binary-coded decimal operation.

AAA	ASCII adjust after addition.
AAS	ASCII adjust after subtraction.

These instructions adjust the **al** register after performing an unpacked binary-coded decimal operation.

AAM	ASCII adjust after multiplication.
AAD	ASCII adjust after division.

These instructions adjust the **al** and **ah** registers after performing an unpacked binary-coded decimal operation.

Bits

	AND	**r1**/m, **r2**/m/#n	**r1**/m = **r1**/m and **r2**/m/#n
	OR	**r1**/m, **r2**/m/#n	**r1**/m = **r1**/m or **r2**/m/#n
	XOR	**r1**/m, **r2**/m/#n	**r1**/m = **r1**/m xor **r2**/m/#n
	NOT	**r1**/m	**r1**/m = bitwise not **r1**/m
*	TEST	**r1**/m, **r2**/m/#n	Compute **r1**/m and **r2**/m/#n

The **test** instruction computes the logical AND operator and sets flags according to the result, but throws the result away. It is typically followed by a conditional jump instruction that tests the result of the logical AND.

	SHL	**r1**/m, **cl**/#n	**r1**/m <<= **cl**/#n

	SHR	**r1**/m, **cl**/#n	**r1**/m >>= **cl**/#n zero-fill
*	SAR	**r1**/m, **cl**/#n	**r1**/m >>= **cl**/#n sign-fill

The last bit shifted out is placed in the carry.

	SHLD	**r1**, **r2**/m, **cl**/#n	Shift left double.

Shift **r1** left by **cl**/#n, filling with the top bits of **r2**/m. The last bit shifted out is placed in the carry.

SHRD	**r1**, **r2**/m, **cl**/#n	Shift right double.

Shift **r1** right by **cl**/#n, filling with the bottom bits of **r2**/m. The last bit shifted out is placed in the carry.

ROL	**r1**, **cl**/#n	Rotate **r1** left by **cl**/#n.
ROR	**r1**, **cl**/#n	Rotate **r1** right by **cl**/#n.
RCL	**r1**, **cl**/#n	Rotate **r1**/C left by **cl**/#n.
RCR	**r1**, **cl**/#n	Rotate **r1**/C right by **cl**/#n.

Rotation is like shifting, except that the bits that are shifted out reappear as the incoming fill bits. The C-language version of the rotation instructions incorporate the carry bit into the rotation.

BT	**r1**, **r2**/#n	Copy bit **r2**/#n of **r1** into carry.
BTS	**r1**, **r2**/#n	Set bit **r2**/#n of **r1**, copy previous value into carry.
BTC	**r1**, **r2**/#n	Clear bit **r2**/#n of **r1**, copy previous value into carry.

Control Flow

	J*cc*	dest	Branch conditional.
	JMP	dest	Jump direct.

	JMP	**r**/m	Jump indirect.
	CALL	dest	Call direct.
*	CALL	**r**/m	Call indirect.

The **call** instruction pushes the return address onto the stack then jumps to the destination.

*	RET	*#n*	Return

The **ret** instruction pops and jumps to the return address on the stack. A nonzero *#n* in the **RET** instruction indicates that after popping the return address, the value *#n* should be added to the stack pointer.

LOOP	Decrement **ecx** and jump if result is nonzero.
LOOPZ	Decrement **ecx** and jump if result is nonzero and **zr** was set.
LOOPNZ	Decrement **ecx** and jump if result is nonzero and **zr** was clear.
JECXZ	Jump if **ecx** is zero.

These instructions are remnants of the x86's CISC heritage and in recent processors are actually slower than the equivalent instructions written out the long way.

String Manipulation

	MOVS*T*	Move *T* from **esi** to **edi.**
	CMPS*T*	Compare *T* from **esi** with **edi.**
	SCAS*T*	Scan *T* from **edi** for acc*T.*
	LODS*T*	Load *T* from **esi** into acc*T.*
	STOS*T*	Store *T* to **edi** from acc*T.*

After performing the operation, the source and destination register are incremented or decremented by sizeof(*T*), according to the setting of the direction flag (up or down).

The instruction can be prefixed by **REP** to repeat the operation the number of times specified by the **ecx** register.

The **rep mov** instruction is used to copy blocks of memory.

The **rep stos** instruction is used to fill a block of memory with acc*T*.

Flags

LAHF	Load **ah** from flags.
SAHF	Store **ah** to flags.
STC	Set carry.
CLC	Clear carry.
CMC	Complement carry.
STD	Set direction to *down.*
CLD	Set direction to *up.*
STI	Enable interrupts.
CLI	Disable interrupts.

Interlocked Instructions

XCHG	**r1**, **r**/m	Swap **r1** and **r**/m.
XADD	**r1**, **r**/m	Add **r1** to **r**/m, put original value in **r1.**
CMPXCHG	**r1**, **r**/m	Compare and exchange conditional.

The **cmpxchg** instruction is the atomic version of the following:

```asm
   cmp     accT, r/m
   jz      match
   mov     accT, r/m
```

```asm
   jmp     done
match:
   mov     r/m, r1
done:
```

Miscellaneous

	INT	#n	Trap to kernel.
	BOUND	**r**, m	Trap if **r** not in range.
*	NOP		No operation.
	XLATB		**al** = [**ebx** + **al**]
	BSWAP	**r**	Swap byte order in register.

Here is a special case of the **int** instruction.

INT	3	Debugger breakpoint trap.

The opcode for **INT 3** is 0xCC. The opcode for **NOP** is 0x90.

When debugging code, you may need to patch out some code. You can do this by replacing the offending bytes with 0x90.

Idioms

	XOR	**r**, **r**	**r** = 0
	TEST	**r**, **r**	Check if **r** = 0.
*	ADD	**r**, **r**	Shift **r** left by 1.