Instruction Set

In general, the features of the modern x86 instruction set are:

- A compact encoding
  - Variable length and alignment independent (encoded as little endian, as is all data in the x86 architecture)
  - Mainly one-address and two-address instructions, that is to say, the first operand is also the destination.
  - Memory operands as both source and destination are supported (frequently used to read/write stack elements addressed using small immediate offsets).
  - Both general and implicit register usage; although all seven (counting ebp) general registers in 32-bit mode, and all fifteen (counting rbp) general registers in 64-bit mode, can be freely used as accumulators or for addressing, most of them are also *implicitly* used by certain (more or less) special instructions; affected registers must therefore be temporarily preserved (normally stacked), if active during such instruction sequences.
- Produces conditional flags implicitly through most integer ALU instructions.
- Supports various addressing modes including immediate, offset, and scaled index but not PC-relative, except jumps (introduced as an improvement in the x86-64 architecture).
- Includes floating point to a stack of registers.
- Contains special support for atomic instructions (xchg, cmpxchg/cmpxchg8b, xadd, and integer instructions which combine with the lock prefix)
- SIMD instructions (instructions which perform parallel simultaneous single instructions on many operands encoded in adjacent cells of wider registers).

TYPES OF INSTRUCTION
Instructions vary from one CPU to another
General groupings possible...
Arithmetic/Logic
* Add, subtract, AND, OR, shifts
* Performed by ALU Data Movement
* Load, Store (to/from registers/memory) Transfer of Control
* Jump, Branch, procedure call Test/Compare
* Set condition flags Input/Output
* In, Out
* Only on some CPU's Others
* Halt, NOP

Stack instructions

The x86 architecture has hardware support for an execution stack mechanism. Instructions such as push, pop, call and ret are used with the properly set up stack to pass parameters, to allocate space for local data, and to save and restore call-return points. The ret *size* instruction is very useful for implementing space efficient (and fast) calling conventions where the callee is responsible for reclaiming stack space occupied by parameters.

When setting up a stack frame to hold local data of a recursive procedure there are several choices; the high level enter instruction takes a *procedure-nesting-depth* argument as well as a *local size* argument, and may be faster than more explicit manipulation of the registers (such as push bp, mov bp, sp, sub sp, *size*) but it is generally not used. Whether it is faster depends on the particular x86 implementation (i.e. processor) as well as the calling convention and code intended to run on multiple processors will usually run faster on most targets without it.

The full range of addressing modes (including *immediate* and *base+offset*) even for instructions such as push and pop, makes direct usage of the stack for integer, floating point and address data simple, as well as keeping the ABI specifications and mechanisms relatively simple compared to some RISC architectures (require more explicit call stack details).

Integer ALU instructions

x86 assembly has the standard mathematical operations, add, sub, mul, with idiv; the logical operators and, or, xor, neg; bitshift arithmetic and logical, sal/sar, shl/shr; rotate with and without carry, rcl/rcr, rol/ror, a complement of BCD arithmetic instructions, aaa, aad, daa and others.

Floating point instructions

x86 assembly language includes instructions for a stack-based floating point unit. They include addition, subtraction, negation, multiplication, division, remainder, square roots, integer truncation, fraction truncation, and scale by power of two. The operations also include conversion instructions which can load or store a value from memory in any of the following formats: Binary coded decimal, 32-bit integer, 64-bit integer, 32-bit floating point, 64-bit floating point or 80-bit floating point (upon loading, the value is converted to the currently used floating point mode). x86 also includes a number of transcendental functions including sine, cosine, tangent, arctangent, exponentiation with the base 2 and logarithms to bases 2, 10, or e.

The stack register to stack register format of the instructions is usually f*op* st, st(*) or f*op* st(*), st, where st is equivalent to st(0), and st(*) is one of the 8 stack registers (st(0), st(1), ..., st(7)). Like the integers, the first operand is both the first source operand and the destination operand. fsubr and fdivr should be singled out as first swapping the source operands before performing the subtraction or division. The addition, subtraction, multiplication, division, store and comparison instructions include instruction modes that will pop the top of the stack after their operation is complete. So for example faddp st(1), st performs the calculation st(1) = st(1) + st(0), then removes st(0) from the top of stack, thus making what was the result in st(1) the top of the stack in st(0).

SIMD instructions

Modern x86 CPUs contain SIMD instructions, which largely perform the same operation in parallel on many values encoded in a wide SIMD register. Various instruction technologies support different operations on different register sets, but taken as complete whole (from MMX to SSE4.2) they include general computations on integer or floating point arithmetic (addition, subtraction, multiplication, shift, minimization, maximization, comparison, division or square root). So for example, paddw mm0, mm1 performs 4 parallel 16-bit (indicated by the w) integer adds (indicated by the padd) of mm0 values to mm1 and stores the result in mm0. SSE also includes a floating point mode in which only the very first value of the registers is actually modified (expanded in SSE2). Some other unusual instructions have been added including a sum of absolute differences (used for motion estimation in video compression, such as is done in MPEG) and a 16-bit multiply accumulation instruction (useful for software-based alpha-blending and digital filtering). SSE (since SSE3) and 3DNow! extensions include addition and subtraction instructions for treating paired floating point values like complex numbers.

These instruction sets also include numerous fixed sub-word instructions for shuffling, inserting and extracting the values around within the registers. In addition there are instructions for moving data between the integer registers and XMM (used in SSE)/FPU (used in MMX) registers.

Data manipulation instructions

The x86 processor also includes complex addressing modes for addressing memory with an immediate offset, a register, a register with an offset, a scaled register with or without an offset, and a register with an optional offset and another scaled register. So for example, one can encode mov eax, [Table + ebx + esi*4] as a single instruction which loads 32 bits of data from the address computed as (Table + ebx + esi * 4) offset from the ds selector, and stores it to the eax register. In general x86 processors can load and use memory matched to the size of any register it is operating on. (The SIMD instructions also include half-load instructions.)

The x86 instruction set includes string load, store and move instructions (lods, stos, and movs) which perform each operation to a specified size (b for 8-bit byte, w for 16-bit word, d for 32-bit double word) then increments/decrements (depending on DF, direction flag) the implicit address register (si for lods, di for stos, and both for movs). For the load and store, the implicit target/source register is in the al, ax or eax register (depending on size). The implicit segment used is ds for lods, es for stos and both for movs.

The stack is implemented with an implicitly decrementing (push) and incrementing (pop) stack pointer. In 16-bit mode, this implicit stack pointer is addressed as SS:[SP], in 32-bit mode it is SS:[ESP], and in 64-bit mode it is [RSP]. The stack pointer actually points to the last value that was stored, under the assumption that its size will match the operating mode of the processor (i.e., 16, 32, or 64 bits) to match the default width of the push/pop/call/ret instructions. Also included are the instructions enter and leave which reserve and remove data from the top of the stack while setting up a stack frame pointer in bp/ebp/rbp. However, direct setting, or addition and subtraction to the sp/esp/rsp register is also supported, so the enter/leave instructions are generally unnecessary.

This code in the beginning of a function:

```
pushl   %ebp       /* save calling function's stack frame (%ebp) */
movl    %esp,%ebp  /* make a new stack frame on top of our caller's stack */
subl    $4,%esp    /* allocate 4 bytes of stack space for this function's local variables */
```

...is functionally equivalent to just:

```
enter   $4,$0
```

Other instructions for manipulating the stack include pushf/popf for storing and retrieving the (E)FLAGS register. The pusha/popa instructions will store and retrieve the entire integer register state to and from the stack.

Values for a SIMD load or store are assumed to be packed in adjacent positions for the SIMD register and will align them in sequential little-endian order. Some SSE load and store instructions require 16-byte alignment to function properly. The SIMD instruction sets also include "prefetch" instructions which perform the load but do not target any register, used for cache loading. The SSE instruction sets also include non-temporal store instructions which will perform stores straight to memory without performing a cache allocate if the destination is not already cached (otherwise it will behave like a regular store.)

Most generic integer and floating point (but no SIMD) instructions can use one parameter as a complex address as the second source parameter. Integer instructions can also accept one memory parameter as a destination operand.

8086 instruction set (79 basic instructions)

AAA	AAD	AAM	AAS
ADC	ADD	AND	CALL
CBW	CLC	CLD	CLI
CMC	CMP	CMPS	CMPXCHG
CWD	DAA	DAS	DEC
DIV	ESC	HLT	IDIV
IMUL	IN	INC	INT
INTO	IRET/IRETD	Jxx	JCXZ/JECXZ
JMP	LAHF	LDS	LEA
LES	LOCK	LODS	LOOP
MOV	MOVS	LOOPE/LOOPZ	LOOPNZ/LOOPNE
MUL	NEG	NOP	NOT
OR	OUT	POP	POPF/POPFD
PUSH	RCL	RCR	PUSHF/PUSHFD
REP	REPE/REPZ	RET/RETF	REPNE/REPNZ
ROL	ROR	SAHF	SAL/SHL
SAR	SBB	SCAS	SHL
SHR	STC	STD	STI
STOS	SUB	TEST	WAIT/FWAIT
XCHG	XLAT/XLATB	XOR	