X86 assembly language

x86 assembly language is a family of backward-compatible assembly languages, which provide some level of compatibility all the way back to the Intel 8008 introduced in April 1972.[1][2] x86 assembly languages are used to produce object code for the x86 class of processors. Like all assembly languages, it uses short mnemonics to represent the fundamental instructions that the CPU in a computer can understand and follow.[3] Compilers sometimes produce assembly code as an intermediate step when translating a high level program into machine code. Regarded as a programming language, assembly coding is *machine-specific* and *low level*. Assembly languages are more typically used for detailed and time critical applications such as small real-time embedded systems or operating system kernels and device drivers.

Mnemonics and opcodes[edit]

Each x86 assembly instruction is represented by a mnemonic which, often combined with one or more operands, translates to one or more bytes called an opcode; the NOP instruction translates to 0x90, for instance and the HLT instruction translates to 0xF4.[3] There are potential opcodes with no documented mnemonic which different processors may interpret differently, making a program using them behave inconsistently or even generate an exception on some processors. These opcodes often turn up in code writing competitions as a way to make the code smaller, faster, more elegant or just show off the author's prowess.

Syntax[edit]

x86 assembly language has two main syntax branches: *Intel syntax* and *AT&T syntax*.[4] *Intel syntax* is dominant in the DOS and Windows world, and AT&T syntax is dominant in the Unix world, since Unix was created at AT&T Bell Labs.[5] Here is a summary of the main differences between *Intel syntax* and *AT&T syntax*:

	AT&T	**Intel**
Parameter order	Source before the destination. `movl $5, %eax`	Destination before source. `mov eax, 5`

Parameter size	Mnemonics are suffixed with a letter indicating the size of the operands: *q* for qword, *l* for long (dword), *w* for word, and *b* for byte.[4] `addl $4, %esp`	Derived from the name of the register that is used (e.g. *rax*, *eax*, *ax*, *al* imply *q*, *l*, *w*, *b*, respectively). `add esp, 4`
Sigils	Immediate values prefixed with a '$', registers prefixed with a '%'.[4]	The assembler automatically detects the type of symbols; i.e., whether they are registers, constants or something else.
Effective addresses	General syntax of *DISP(BASE,INDEX,SCALE)*. Example: `movl mem_location(%ebx,%ecx,4), %eax`	Arithmetic expressions in square brackets; additionally, size keywords like *byte*, *word*, or *dword* have to be used if the size cannot be determined from the operands.[4] Example: `mov eax, [ebx + ecx*4 + mem_location]`

Many x86 assemblers use *Intel syntax*, including NASM, FASM, MASM, TASM, and YASM. GAS, which originally used *AT&T syntax*, has supported both syntaxes since version 2.10 via the *.intel_syntax* directive.[4][6][7] A quirk in the AT&T syntax for x86 is that x87 operands are reversed, an inherited bug from the original AT&T assembler.[8]

The AT&T syntax is nearly universal to all other architectures with the same `mov` order; it was originally a syntax for PDP-11 assembly. The Intel syntax is specific to the x86 architecture, and is the one used in the x86 platform's documentation.

Registers[edit]

x86 processors have a collection of registers available to be used as stores for binary data. Collectively the data and address registers are called the general registers. Each register has a special purpose in addition to what they can all do: [3]

- AX multiply/divide, string load & store
- BX index register for MOVE
- CX count for string operations & shifts
- DX port address for IN and OUT
- SP points to top of the stack

- BP points to base of the stack frame
- SI points to a source in stream operations
- DI points to a destination in stream operations

Along with the general registers there are additionally the:

- IP instruction pointer
- FLAGS
- segment registers (CS, DS, ES, FS, GS, SS) which determine where a 64k segment starts (no FS & GS in 80286 & earlier)
- extra extension registers (MMX, 3DNow!, SSE, etc.) (Pentium & later only).

The IP register points to the memory offset of the next instruction in the code segment (it points to the first byte of the instruction). The IP register cannot be accessed by the programmer directly.

The x86 registers can be used by using the MOV instructions. For example, in Intel syntax:

```
mov ax, 1234h ; copies the value 1234hex (4660d) into register AX
```

```
mov bx, ax    ; copies the value of the AX register into the BX register
```

Segmented addressing[edit]

The x86 architecture in real and virtual 8086 mode uses a process known as **segmentation** to address memory, not the **flat memory model** used in many other environments. Segmentation involves composing a memory address from two parts, a *segment* and an *offset*; the segment points to the beginning of a 64 KB (64×2^{10}) group of addresses and the offset determines how far from this beginning address the desired address is. In segmented addressing, two registers are required for a complete memory address. One to hold the segment, the other to hold the offset. In order to translate back into a flat address, the segment value is shifted four bits left (equivalent to multiplication by 2^4 or 16) then added to the offset to form the full address, which allows breaking the 64k barrier through clever choice of addresses, though it makes programming considerably more complex.

In real mode/protected only, for example, if DS contains the hexadecimal number 0xDEAD and DX contains the number 0xCAFE they would together point to the memory address 0xDEAD * 0x10 + 0xCAFE = 0xEB5CE. Therefore, the CPU can address up to 1,048,576 bytes (1 MB) in real mode. By combining *segment* and *offset* values we find a 20-bit address.

The original IBM PC restricted programs to 640 KB but an expanded memory specification was used to implement a bank switching scheme that fell out of use when later operating systems, such as Windows, used the larger address ranges of newer processors and implemented their own virtual memory schemes.

Protected mode, starting with the Intel 80286, was utilized by OS/2. Several shortcomings, such as the inability to access the BIOS and the inability to switch back to real mode without resetting the processor, prevented widespread usage.[9] The 80286 was also still limited to addressing memory in 16-bit segments, meaning only 2^{16} bytes (64 kilobytes) could be accessed at a time. To access the extended functionality of the 80286, the operating system would set the processor into protected mode, enabling 24-bit addressing and thus 2^{24} bytes of memory (16 megabytes).

In protected mode, the segment selector can be broken down into three parts: a 13-bit index, a *Table Indicator* bit that determines whether the entry is in the GDT or LDT and a 2-bit *Requested Privilege Level*; see x86 memory segmentation.

When referring to an address with a segment and an offset the notation of *segment:offset* is used, so in the above example the *flat address* 0xEB5CE can be written as 0xDEAD:0xCAFE or as a segment and offset register pair; DS:DX.

There are some special combinations of segment registers and general registers that point to important addresses:

- CS:IP (CS is *Code Segment*, IP is *Instruction Pointer*) points to the address where the processor will fetch the next byte of code.
- SS:SP (SS is *Stack Segment*, SP is *Stack Pointer*) points to the address of the top of the stack, i.e. the most recently pushed byte.
- DS:SI (DS is *Data Segment*, SI is *Source Index*) is often used to point to string data that is about to be copied to ES:DI.
- ES:DI (ES is *Extra Segment*, DI is *Destination Index*) is typically used to point to the destination for a string copy, as mentioned above.

The Intel 80386 featured three operating modes: real mode, protected mode and virtual mode. The protected mode which debuted in the 80286 was extended to allow the 80386 to address up to 4 GB of memory, the all new virtual 8086 mode (*VM86*) made it possible to run one or more real mode programs in a protected environment which largely emulated real mode, though some programs were not compatible (typically as a result of memory addressing tricks or using unspecified op-codes).

The 32-bit flat memory model of the 80386's extended protected mode may be the most important feature change for the x86 processor family until AMD released x86-64 in 2003, as it helped drive large scale adoption of Windows 3.1 (which relied on protected mode) since Windows could now run many applications at once, including DOS applications, by using virtual memory and simple multitasking.

Execution modes[edit]

The x86 processors support five modes of operation for x86 code, **Real Mode**, **Protected Mode**, **Long Mode**, **Virtual 86 Mode**, and **System Management Mode**, in which some instructions are available and others are not. A 16-bit subset of instructions are available on the 16-bit x86 processors, which are the 8086, 8088, 80186, 80188, and 80286. These instructions are available in real mode on all x86 processors, and in 16-bit protected mode (80286 onwards), additional instructions relating to protected mode are available. On the 80386 and later, 32-bit instructions (including later extensions) are also available in all modes, including real mode; on these CPUs, V86 mode and 32-bit protected mode are added, with additional instructions provided in these modes to manage their features. SMM, with some of its own special instructions, is available on some Intel i386SL, i486 and later CPUs. Finally, in long mode (AMD Opteron onwards), 64-bit instructions, and more registers, are also available. The instruction set is similar in each mode but memory addressing and word size vary, requiring different programming strategies.

The modes in which x86 code can be executed in are:

- Real mode (16-bit)
  - 20-bit segmented memory address space (meaning that only 1 MB of memory can be addressed—actually, slightly more), direct software access to peripheral hardware, and no concept of memory protection or multitasking at the hardware level. Computers that use BIOS start up in this mode.
- Protected mode (16-bit and 32-bit)
  - Expands addressable physical memory to 16 MB and addressable virtual memory to 1 GB. Provides privilege levels and protected memory, which prevents programs from corrupting one another. 16-bit protected mode (used during the end of the DOS era) used a complex, multi-segmented memory model. 32-bit protected mode uses a simple, flat memory model.
- Long mode (64-bit)
  - Mostly an extension of the 32-bit (protected mode) instruction set, but unlike the 16 – to – 32-bit transition, many instructions were dropped in the 64-bit mode. Pioneered by AMD.

- Virtual 8086 mode (16-bit)
    - A special hybrid operating mode that allows real mode programs and operating systems to run while under the control of a protected mode supervisor operating system
- System Management Mode (16-bit)
    - Handles system-wide functions like power management, system hardware control, and proprietary OEM designed code. It is intended for use only by system firmware,. All normal execution, including the operating system, is suspended. An alternate software system (which usually resides in the computer's firmware, or a hardware-assisted debugger) is then executed with high privileges.

Switching modes[edit]

The processor runs in real mode immediately after power on, so an operating system kernel, or other program, must explicitly switch to another mode if it wishes to run in anything but real mode. Switching modes is accomplished by modifying certain bits of the processor's control registers after some preparation, and some additional setup may be required after the switch.

Examples[edit]

With a computer running legacy BIOS, the BIOS and the boot loader is running in Real mode, then the 64-bit operating system kernel checks and switches the CPU into Long mode and then starts new kernel-mode threads running 64-bit code.

With a computer running UEFI, the UEFI firmware (except CSM and legacy Option ROM), the UEFI boot loader and the UEFI operating system kernel is all running in Long mode.

Instruction types[edit]

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 read-modify-write 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).

Stack instructions[edit]

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 (introduced with the 80186) 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`). Whether it is faster or slower depends on the particular x86-processor implementation as well as the calling convention used by the compiler, programmer or particular program code; most x86 code is intended to run on x86-processors from several manufacturers and on different technological generations of processors, which implies highly varying microarchitectures and microcode solutions as well as varying gate- and transistor-level design choices.

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[edit]

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[edit]

x86 assembly language includes instructions for a stack-based floating-point unit (FPU). The FPU was an optional separate coprocessor for the 8086 through the 80386, it was an on-chip option for the 80486 series, and it is a standard feature in every Intel x86 CPU since the 80486, starting with the Pentium. The FPU instructions 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 `fop st, st(n)` or `fop st(n), st`, where `st` is equivalent to `st(0)`, and `st(n)` 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 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[edit]

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`. Streaming SIMD Extensions or

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[edit]

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, move, scan and compare instructions (`lods` , `stos` , `movs` , `scas` and `cmps`) 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 `scas` , and both for `movs` and `cmps`). For the load, store and scan operations, the implicit target/source/comparison register is in the `al` , `ax` or `eax` register (depending on size). The implicit segment registers used are `ds` for `si` and `es` for `di` . The `cx` or `ecx` register is used as a decrementing counter, and the operation stops when the counter reaches zero or (for scans and comparisons) when inequality is detected.

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:

```
push    ebp       ; save calling function's stack frame (ebp)
mov     ebp, esp  ; make a new stack frame on top of our caller's stack
sub     esp, 4    ; allocate 4 bytes of stack space for this function's local variables
```

...is functionally equivalent to just:

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.

Program flow[edit]

The x86 assembly has an unconditional jump operation, `jmp`, which can take an immediate address, a register or an indirect address as a parameter (note that most RISC processors only support a link register or short immediate displacement for jumping).

Also supported are several conditional jumps, including `jz` (jump on zero), `jnz` (jump on non-zero), `jg` (jump on greater than, signed), `jl` (jump on less than, signed), `ja` (jump on above/greater than, unsigned), `jb` (jump on below/less than, unsigned). These conditional operations are based on the state of specific bits in the (E)FLAGS register. Many arithmetic and logic operations set, clear or complement these flags depending on their result. The comparison `cmp` (compare) and `test` instructions set the flags as if they had performed a subtraction or a bitwise AND operation, respectively, without altering the values of the operands. There are also instructions such as `clc` (clear carry flag) and `cmc` (complement carry flag) which work on the flags directly. Floating point comparisons are performed via `fcom` or `ficom` instructions which eventually have to be converted to integer flags.

Each jump operation has three different forms, depending on the size of the operand. A *short* jump uses an 8-bit signed operand, which is a relative offset from the current instruction. A *near* jump is similar to a short jump but uses a 16-bit signed operand (in real or protected mode) or a 32-bit signed operand (in 32-bit protected mode only). A *far* jump is one that uses the full segment base:offset value as an absolute address. There are also indirect and indexed forms of each of these.

In addition to the simple jump operations, there are the `call` (call a subroutine) and `ret` (return from subroutine) instructions. Before transferring control to the subroutine, `call` pushes the segment offset address of the instruction following the `call` onto the stack; `ret` pops this value off the stack, and jumps to it, effectively returning the flow of control to that part of the program. In the case of a `far call`, the segment base is pushed following the offset; `far ret` pops the offset and then the segment base to return.

There are also two similar instructions, `int` (interrupt), which saves the current (E)FLAGS register value on the stack, then performs a `far call`, except that instead of an address, it uses an *interrupt vector*, an index into a table of interrupt handler addresses. Typically, the interrupt handler saves all other CPU registers it uses, unless they are used to return the result of an operation to the calling program (in software called interrupts). The matching return from interrupt instruction is `iret`, which restores the flags after returning. *Soft Interrupts* of the type described above are used by some operating systems for system calls, and can also be used in debugging hard interrupt handlers. *Hard interrupts* are triggered by external hardware events, and must preserve all register values as the state of the currently executing program is unknown. In Protected Mode, interrupts may be set up by the OS to trigger a task switch, which will automatically save all registers of the active task.

Examples[edit]

'Hello world!' program for DOS in MASM style assembly[edit]

Using interrupt 21h for output – other samples use libc's printf to print to stdout.[10]

```
.model small
.stack 100h

.data
msg        db        'Hello world!$'

.code
start:
        mov        ah, 09h   ; Display the message
        lea        dx, msg
        int        21h
        mov        ax, 4C00h  ; Terminate the executable
        int        21h

end start
```

'Hello world!' program for Windows in MASM style assembly[edit]

```
; requires /coff switch on 6.15 and earlier versions
.386
.model small,c
.stack 1000h

.data
msg     db 'Hello world!',0

.code
includelib libcmt.lib
includelib libvcruntime.lib
includelib libucrt.lib
includelib legacy_stdio_definitions.lib

extrn printf:near
extrn exit:near

public main
main proc
        push    offset msg
        call    printf
        push    0
        call    exit
main endp

end
```

'Hello world!' program for Windows in NASM style assembly[edit]

```
; Image base = 0x00400000
%define RVA(x) (x-0x00400000)
section .text
push dword hello
call dword [printf]
push byte +0
call dword [exit]
ret

section .data
hello db 'Hello world!'

section .idata
dd RVA(msvcrt_LookupTable)
dd -1
dd 0
dd RVA(msvcrt_string)
dd RVA(msvcrt_imports)
times 5 dd 0 ; ends the descriptor table

msvcrt_string dd 'msvcrt.dll', 0
msvcrt_LookupTable:
dd RVA(msvcrt_printf)
dd RVA(msvcrt_exit)
dd 0

msvcrt_imports:
printf dd RVA(msvcrt_printf)
exit dd RVA(msvcrt_exit)
dd 0

msvcrt_printf:
dw 1
dw 'printf', 0
msvcrt_exit:
dw 2
dw 'exit', 0
dd 0
```

'Hello world!' program for Linux in NASM style assembly[edit]

```
;
; This program runs in 32-bit protected mode.
;  build: nasm -f elf -F stabs name.asm
;  link:  ld -o name name.o
;
; In 64-bit long mode you can use 64-bit registers (e.g. rax instead of eax, rbx instead of
ebx, etc.)
; Also change '-f elf ' for '-f elf64' in build command.
;
section .data                           ; section for initialized data
str:     db 'Hello world!', 0Ah         ; message string with new-line char at the end (10
decimal)
str_len: equ $ - str                    ; calcs length of string (bytes) by subtracting the
str's start address
                                            ; from this address ($ symbol)

section .text                           ; this is the code section
global _start                           ; _start is the entry point and needs global scope to
be 'seen' by the
                                            ; linker --equivalent to main() in C/C++
_start:                                 ; definition of _start procedure begins here
        mov        eax, 4                   ; specify the sys_write function code (from OS
vector table)
        mov        ebx, 1                   ; specify file descriptor stdout --in gnu/linux,
everything's treated as a file,
                                             ; even hardware devices
        mov        ecx, str                 ; move start _address_ of string message to ecx
register
        mov        edx, str_len             ; move length of message (in bytes)
        int        80h                      ; interrupt kernel to perform the system call we
just set up -
                                             ; in gnu/linux services are requested through
the kernel
        mov        eax, 1                   ; specify sys_exit function code (from OS vector
table)
        mov        ebx, 0                   ; specify return code for OS (zero tells OS
everything went fine)
        int        80h                      ; interrupt kernel to perform system call (to
exit)
```

'Hello world!' program for Linux in NASM style assembly using the C standard library[edit]

```
;
;  This program runs in 32-bit protected mode.
;  gcc links the standard-C library by default

;  build: nasm -f elf -F stabs name.asm
;  link:  gcc -o name name.o
;
; In 64-bit long mode you can use 64-bit registers (e.g. rax instead of eax, rbx instead of
ebx, etc..)
; Also change '-f elf ' for '-f elf64' in build command.
;
        global  main                                    ;main must be defined as it being
compiled against the C-Standard Library
        extern  printf                                   ;declares use of external symbol as
printf is declared in a different object-module.
                                                               ;Linker resolves this symbol later.

segment .data                                           ;section for initialized data
        string db 'Hello world!', 0Ah, 0h               ;message string with new-line char (10
decimal) and the NULL terminator
                                                        ;string now refers to the starting
address at which 'Hello, World' is stored.

segment .text
main:
        push    string                                  ;push the address of first character of
string onto stack. This will be argument to printf
        call    printf                                  ;calls printf
        add     esp, 4                                  ;advances stack-pointer by 4 flushing out
the pushed string argument
        ret                                             ;return
```

'Hello world!' program for 64-bit mode Linux in NASM style assembly[edit]

```
;  build: nasm -f elf64 -F dwarf hello.asm
;  link:  ld -o hello hello.o

DEFAULT REL                        ; use RIP-relative addressing modes by default, so [foo] =
[rel foo]

SECTION .rodata                          ; read-only data can go in the .rodata section on GNU/
Linux, like .rdata on Windows
Hello:                db 'Hello world!',10        ; 10 = `\n`.
len_Hello:        equ $-Hello                ; get NASM to calculate the length as an
assemble-time constant
;;  write() takes a length so a 0-terminated C-style string isn't needed. It would be for puts

SECTION .text

global _start
_start:
        mov eax, 1                                ; __NR_write syscall number from Linux asm/
unistd_64.h (x86_64)
        mov edi, 1                                ; int fd = STDOUT_FILENO
        lea rsi, [rel Hello]                        ; x86-64 uses RIP-relative LEA to put
static addresses into regs
        mov rdx, len_Hello               ; size_t count = len_Hello
        syscall                                          ; write(1, Hello, len_Hello);  call
into the kernel to actually do the system call
     ;; return value in RAX.  RCX and R11 are also overwritten by syscall

        mov eax, 60                                ; __NR_exit call number (x86_64)
        xor edi, edi                                ; status = 0 (exit normally)
        syscall                                          ; _exit(0)
```

Running it under `strace` verifies that no extra system calls are made in the process. The printf version would make many more system calls to initialize libc and do dynamic linking. But this is a static executable because we linked using ld without -pie or any shared libraries; the only instructions that run in user-space are the ones you provide.

```
$ strace ./hello > /dev/null                    # without a redirect, your program's stdout
is mixed strace's logging on stderr.  Which is normally fine
execve('./hello', ['./hello'], 0x7ffc8b0b3570 /* 51 vars */) = 0
write(1, 'Hello world!\n', 13)          = 13
exit(0)                                 = ?
+++ exited with 0 +++
```

Using the flags register[edit]

Flags are heavily used for comparisons in the x86 architecture. When a comparison is made between two data, the CPU sets the relevant flag or flags. Following this, conditional jump instructions can be used to check the flags and branch to code that should run, e.g.:

```
        cmp         eax, ebx
        jne         do_something
        ; ...
do_something:
        ; do something here
```

Flags are also used in the x86 architecture to turn on and off certain features or execution modes. For example, to disable all maskable interrupts, you can use the instruction:

The flags register can also be directly accessed. The low 8 bits of the flag register can be loaded into `ah` using the `lahf` instruction. The entire flags register can also be moved on and off the stack using the instructions `pushf`, `popf`, `int` (including `into`) and `iret`.

Using the instruction pointer register[edit]

The instruction pointer is called `ip` in 16-bit mode, `eip` in 32-bit mode, and `rip` in 64-bit mode. The instruction pointer register points to the memory address which the processor will next attempt to execute; it cannot be directly accessed in 16-bit or 32-bit mode, but a sequence like the following can be written to put the address of `next_line` into `eax`:

```
        call        next_line
next_line:
        pop         eax
```

This sequence of instructions generates position-independent code because `call` takes an instruction-pointer-relative immediate operand describing the offset in bytes of the target instruction from the next instruction (in this case 0).

Writing to the instruction pointer is simple — a `jmp` instruction sets the instruction pointer to the target address, so, for example, a sequence like the following will put the contents of `eax` into `eip`:

In 64-bit mode, instructions can reference data relative to the instruction pointer, so there is less need to copy the value of the instruction pointer to another register.

See also[edit]

- Assembly language

- X86 instruction listings
- X86 architecture
- CPU design
- List of assemblers
- Self-modifying code
- DOS

References[edit]

Further reading[edit]

Manuals[edit]

- Intel 64 and IA-32 Software Developer Manuals
- AMD64 Architecture Programmer's Manual (Volume 1-5)

Books[edit]

- *Ed, Jorgensen (May 2018). x86-64 Assembly Language Programming with Ubuntu (PDF) (1.0.97 ed.). p. 367.*
- Dennis Yurichev: *Understanding Assembly Language*

External links[edit]