Assembly Language

Armine Hayrapetyan

Instigate Mobile

Outline

1. Basic Concepts
    a. Questions to ask
    b. Data representation
2. General Concepts
    a. X86 architecture details
    b. X86 memory management
    c. Components of a typical x86 computer
3. Assembly Fundamentals
    a. Basic elements of assembly language
    b. Example: Adding and Subtracting Integers
    c. Assembling, linking and running programs
    d. Defining Data

Basic Concepts: Questions To Ask

Why Learn Assembly?

- If you are planning to be a C or C++ developer, you need to develop an understanding of how memory, address, and instructions work at a low level. A lot of programming errors are not easily recognized at the high-level language level. You will often find it necessary to “drill down” into your program’s internals to find out why it isn’t working.
- Quote from a leading computer scientist, Donald Knuth, in discussing his famous book series, The Art of Computer Programming:
    - *Some people [say] that having machine language, at all, was the great mistake that I made.I really don’t think you can write a book for serious computer programmers unless you are able to discuss low-level detail.*

Question

What is the compilation stages of C program?

What Are Assemblers and Linkers?

An ***assembler*** is a utility program that converts source code programs from assembly language into machine language.

A ***linker*** is a utility program that combines individual files created by an assembler into a single executable program.

A related utility, called a ***debugger***, lets you to step through a program while it's running and examine registers and memory.

How Does Assembly Language Relate to Machine Language?

Machine language is a numeric language specifically understood by a computer's processor (the CPU). All x86 processors understand a common machine language.

Assembly language consists of statements written with short mnemonics such as ADD, MOV, SUB, and CALL. Assembly language has a one-to-one relationship with machine language: Each assembly language instruction corresponds to a single machine-language instruction.

How Do C++ and Java Relate to Assembly Language?

High-level languages such as C++ and Java have a one-to-many relationship with assembly language and machine language. A single statement in C++ expands into multiple assembly language or machine instructions. Ex:

```
int   Y;
int   X = (Y + 4) * 3;
```

Question

Where is loaded c program when we give the command to execute a program?

How Do C++ and Java Relate to Assembly Language?

Following is the equivalent translation to assembly language. The translation requires multiple statements because assembly language works at a detailed level:

```
mov   eax,Y                ; move Y to the EAX register
add   eax,4                ; add 4 to the EAX register
mov   ebx,3                ; move 3 to the EBX register
imul  ebx                  ; multiply EAX by EBX
mov   X,eax                ; move EAX to X
```

Is Assembly Language Portable?

A language whose source programs can be compiled and run on a wide variety of computer systems is said to be ***portable***. A C++ program, for example, should compile and run on just about any computer, unless it makes specific references to library functions that exist under a single operating system. A major feature of the Java language is that compiled programs run on nearly any computer system.

Is Assembly Language Portable?

Assembly language is not portable because it is designed for a specific processor family. There are a number of different assembly languages widely used today, each based on a processor family.

Some well-known processor families are Motorola 68x00, x86, SUN Sparc, Vax, and IBM-370.

The instructions in assembly language may directly match the computer's architecture or they may be translated during execution by a program inside the processor known as a microcode interpreter.

Why Learn Assembly Language?

- **Embedded programs**: They are short programs stored in a small amount of memory in single-purpose devices such as telephones, automobile fuel, etc. Assembly language is an ideal tool for writing embedded programs because of its economical use of memory.
- Computer **game** consoles require their software to be highly optimized for small code size and fast execution. Game programmers are experts at writing code that takes full advantage of hardware features in a target system. They use assembly language as their tool of choice because it permits direct access to computer hardware, and code can be hand optimized for speed.

Why Learn Assembly Language?

- Assembly language helps you to gain an overall **understanding of the interaction between computer hardware, operating systems, and application programs**. Using assembly language, you can apply and test theoretical information you are given in computer architecture and operating systems courses.
- Hardware manufacturers create ***device drivers*** for the equipment they sell. Device drivers are programs that translate general operating system commands into specific references to hardware details. Printer manufacturers, for example, create a different MS-Windows device driver for each model they sell. The same is true for Mac OS, Linux, and other operating systems.

Are There Rules in Assembly Language?

Most rules in assembly language are based on physical limitations of the target processor and its machine language. The CPU, for example, requires two instruction operands to be the same size.

Assembly language, on the other hand, can access any memory address. The price for such freedom is high: Assembly language programmers spend a lot of time debugging!

Assembly Language Applications

In the early days of programming, most applications were written partially or entirely in assembly language. They had to fit in a small area of memory and run as efficiently as possible on slow processors. As memory became more plentiful and processors dramatically increased in speed, programs became more complex. Programmers switched to high-level languages such as C, FORTRAN, and COBOL that contained a certain amount of structuring capability. More recently, object-oriented languages such as C++, C#, and Java have made it possible to write complex programs containing millions of lines of code

Assembly Language Applications

It is rare to see large application programs coded completely in assembly language because they would take too much time to write and maintain. Instead, assembly language is used to optimize certain sections of application programs for speed and to access computer hardware.

Comparison of Assembly Language to High-Level Languages

Type of Application	High-Level Languages	Assembly Language
Commercial or scientific application, written for single platform, medium to large size.	Formal structures make it easy to organize and maintain large sections of code.	Minimal formal structure, so one must be imposed by programmers who have varying levels of experience. This leads to difficulties maintaining existing code.
Hardware device driver.	The language may not provide for direct hardware access. Even if it does, awkward coding techniques may be required, resulting in maintenance difficulties.	Hardware access is straightforward and simple. Easy to maintain when programs are short and well documented.
Commercial or scientific application written for multiple platforms (different operating systems).	Usually portable. The source code can be recompiled on each target operating system with minimal changes.	Must be recoded separately for each platform, using an assembler with a different syntax. Difficult to maintain.
Embedded systems and computer games requiring direct hardware access.	Produces too much executable code, and may not run efficiently.	Ideal, because the executable code is small and runs quickly.

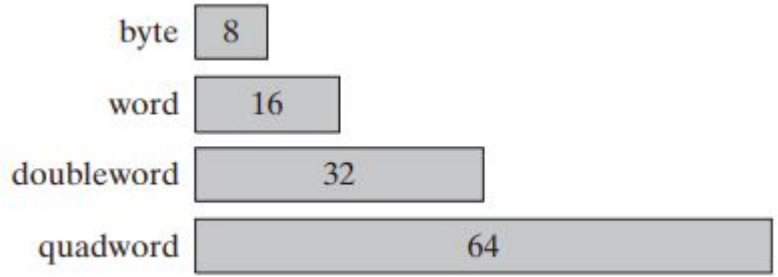
Data Representation

Question

Convert 1010 from binary/octal/hexadecimal system to decimal

Integer Storage Sizes

The basic storage unit for all data in an x86 computer is a **byte**, containing 8 bits. Other storage sizes are **word (2 bytes), doubleword (4 bytes), and quadword (8 bytes)**. In the following figure, the number of bits is shown for each size:



Binary, Octal, Decimal, and Hexadecimal Digits.

System	Base	Possible Digits
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

Storage Sizes and Ranges of Integers

Storage Type	Range (Low to High)	Powers of 2
Signed byte	−128 to +127	-2^7 to $(2^7 - 1)$
Signed word	−32,768 to +32,767	-2^{15} to $(2^{15} - 1)$
Signed doubleword	−2,147,483,648 to 2,147,483,647	-2^{31} to $(2^{31} - 1)$
Signed quadword	−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	-2^{63} to $(2^{63} - 1)$

Character Storage

If computers only store binary data, how do they represent characters? They use a character set, which is a mapping of characters to integers.

- ASCII
- ANSI
- Unicode Standard
  - UTF-8, UTF-16, UTF-32

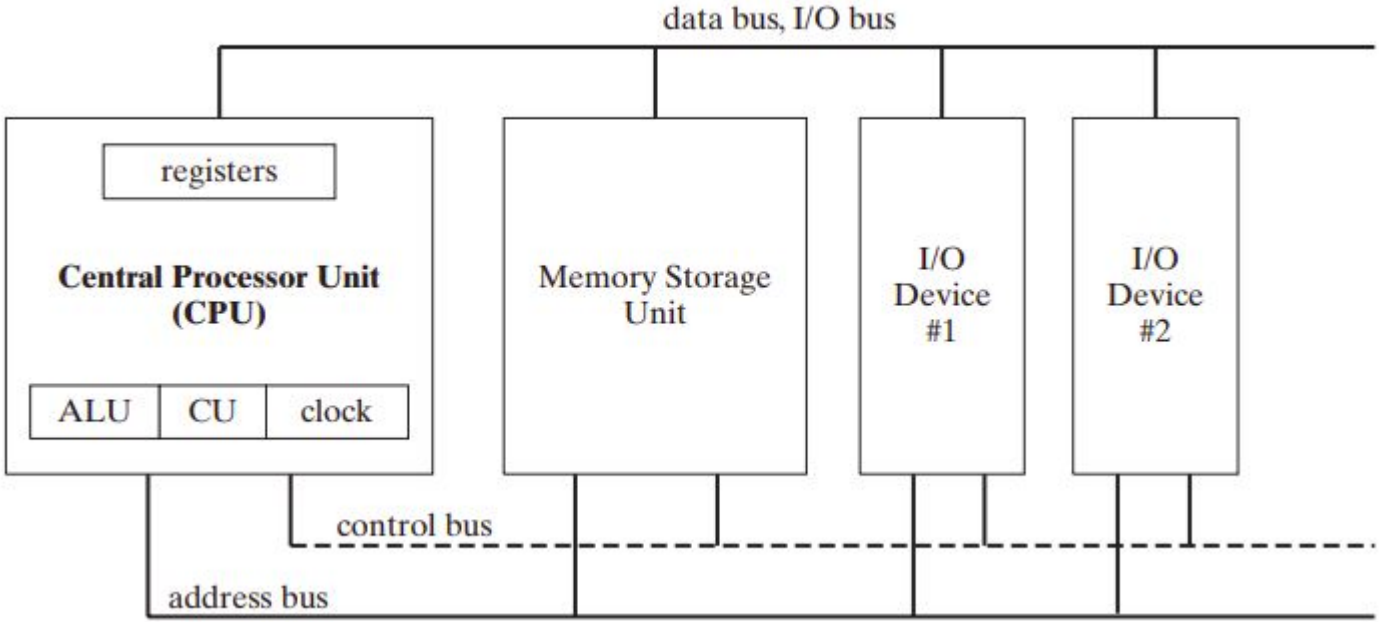
Letter ‘A’ in ASCII standard

Format	Value
ASCII binary	“01000001”
ASCII decimal	“65”
ASCII hexadecimal	“41”
ASCII octal	“101”

x86 Processor Architecture

Basic Microcomputer Design

Assembly language requires you to have a working knowledge of computer hardware. To that end, the concepts and details in this chapter will help you to understand the assembly language code you write.



Central Processor Unit (CPU)

The **central processor unit (CPU)**, where calculations and logic operations take place, contains a limited number of storage locations named registers, a high-frequency clock, a control unit, and an arithmetic logic unit.

- The *clock* synchronizes the internal operations of the CPU with other system components.
- The *control unit (CU)* coordinates the sequencing of steps involved in executing machine instructions.
- The *arithmetic logic unit (ALU)* performs arithmetic operations such as addition and subtraction and logical operations such as AND, OR, and NOT.

Memory Storage Unit

The memory storage unit is where instructions and data are held while a computer program is running. The storage unit receives requests for data from the CPU, transfers data from random access memory (RAM) to the CPU, and transfers data from the CPU into memory. All processing of data takes place within the CPU, so programs residing in memory must be copied into the CPU before they can execute. Individual program instructions can be copied into the CPU one at a time, or groups of instructions can be copied together

Bus

A bus is a group of parallel wires that transfer data from one part of the computer to another. A computer system usually contains four bus types: **data, I/O, control, and address**.

- The ***data bus*** transfers instructions and data between the CPU and memory
- The ***I/O bus*** transfers data between the CPU and the system input/output devices
- The ***control bus*** uses binary signals to synchronize actions of all devices attached to the system bus
- The ***address bus*** holds the addresses of instructions and data when the currently executing instruction transfers data between the CPU and memory

Instruction Execution Cycle

The execution of a single machine instruction can be divided into a sequence of individual operations called the ***instruction execution cycle***. Before executing, a program is loaded into memory. The **instruction pointer** contains the address of the next instruction. The instruction queue holds a group of instructions about to be executed. Executing a machine instruction requires three basic steps: **fetch, decode, and execute.**

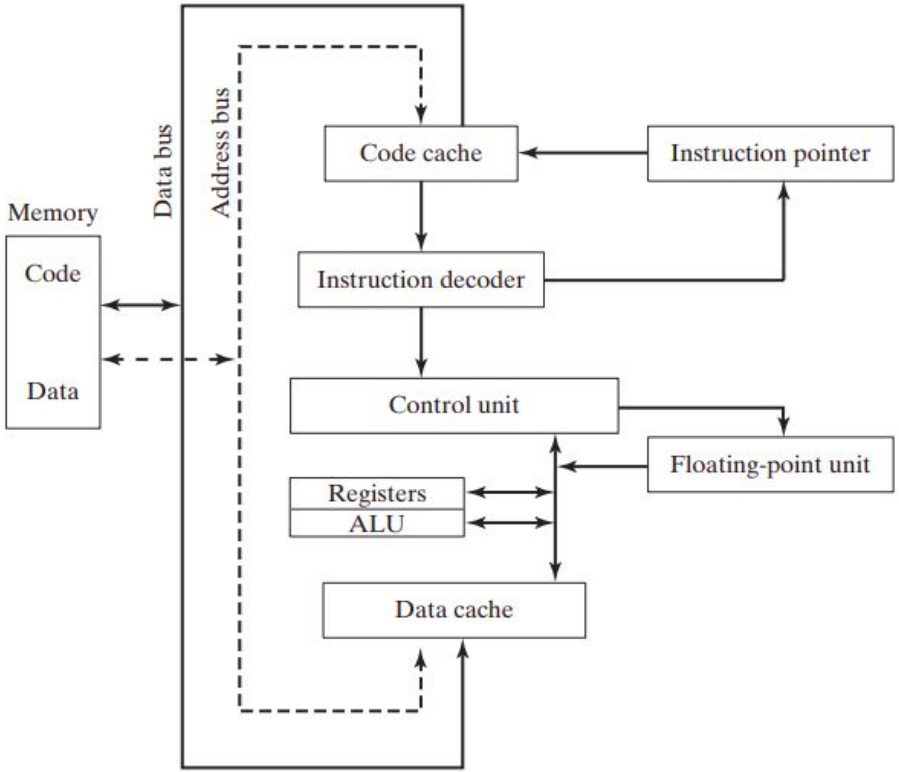
Fetch, Decode, Fetch Operands

- The control unit **fetches** the next instruction from the instruction queue and increments the **instruction pointer (IP)**. The IP is also known as the **program counter**.
- The control unit **decodes** the instruction's function to determine what the instruction will do. The instruction's input operands are passed to the ALU, and signals are sent to the ALU indicating the operation to be performed.
- If the instruction uses an input operand located in memory, the control unit uses a read operation to retrieve the operand and copy it into internal registers. Internal registers are not visible to user programs.

Execute, Store Output Operand

- The ALU executes the instruction using the named registers and internal registers as operands and sends the output to named registers and/or memory. The ALU updates status flags providing information about the processor state
- If the output operand is in memory, the control unit uses a write operation to store the data.

Simplified CPU Block Diagram



Clock

Each operation involving the CPU and the system bus is synchronized by an internal clock pulsing at a constant rate. The basic unit of time for machine instructions is a machine cycle (or clock cycle).

A clock that oscillates 1 billion times per second (1 GHz), for example, produces a clock cycle with a duration of one billionth of a second (1 nanosecond).

A machine instruction requires at least one clock cycle to execute, and a few require in excess of 50 clocks. Instructions requiring memory access often have empty clock cycles called wait states because of the differences in the speeds of the CPU, the system bus, and memory circuits.

Reading From Memory

Program throughput is often dependent on the speed of memory access. CPU clock speed might be several gigahertz, whereas access to memory occurs over a system bus running at a much slower speed. The CPU must wait one or more clock cycles until operands have been fetched from memory before the current instruction can complete its execution. The wasted clock cycles are called **wait states**.

Cache Memory

Because conventional memory is so much slower than the CPU, computers use high-speed cache memory to hold the most recently used instructions and data. The first time a program reads a block of data, it leaves a copy in the cache. If the program needs to read the same data

a second time, it looks for the data in cache. A cache hit indicates the data is in cache; a cache miss indicates the data is not in cache and must be read from conventional memory. In general, cache

memory has a noticeable effect on improving access to data, particularly when the cache is large.

How Programs Run

Load and Execute Process

- The operating system (OS) searches for the program's filename in the current disk directory. If it cannot find the name there, it searches a predetermined list of directories (called paths) for the filename. If the OS fails to find the program filename, it issues an error message.
- If the program file is found, the OS retrieves basic information about the program's file from the disk directory, including the file size and its physical location on the disk drive.
- The OS determines the next available location in memory and loads the program file into memory. It allocates a block of memory to the program and enters information about the program's size and location into a table (sometimes called a descriptor table). Additionally, the OS may adjust the values of pointers within the program so they contain addresses of program data.

Load and Execute Process

- The OS begins execution of the program’s first machine instruction. As soon as the program begins running, it is called a process. The OS assigns the process an identification number (process ID), which is used to keep track of it while running.
- The process runs by itself. It is the OS’s job to track the execution of the process and to respond to requests for system resources. Examples of resources are memory, disk files, and input-output devices
- When the process ends, it is removed from memory.

Multitasking

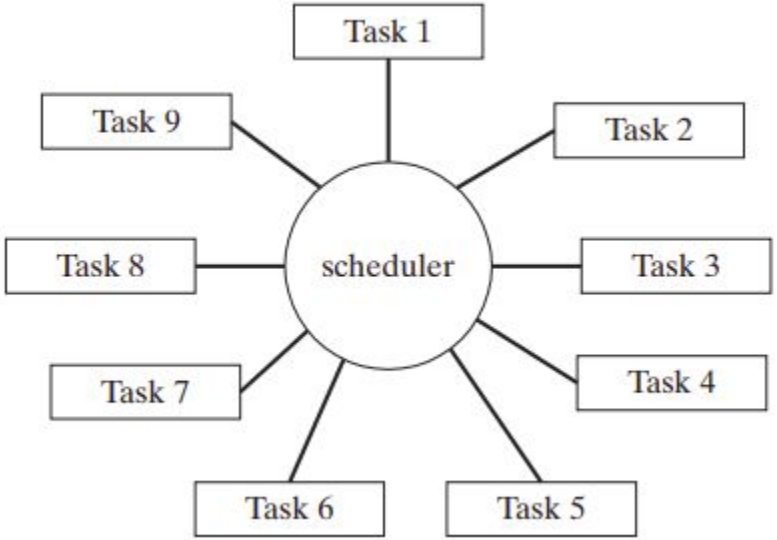
A **multitasking** operating system is able to run multiple tasks at the same time. A task is defined as either a program (a process) or a thread of execution. A process has its own memory area and may contain multiple threads. A thread shares its memory with other threads belonging to the same process. Game programs, for example, often use individual threads to simultaneously control multiple graphic objects. Web browsers use separate threads to simultaneously load graphic images and respond to user input.

Multitasking

A CPU can really execute only one instruction at a time, so a component of the operating system named the ***scheduler*** allocates a slice of CPU time (called a time slice) to each task. During a single time slice, the CPU executes a block of instructions, stopping when the time slice has ended.

By rapidly switching tasks, the processor creates the illusion they are running simultaneously. One type of scheduling used by the OS is called round-robin scheduling. In Figure 2–4, nine tasks are active. Suppose the scheduler arbitrarily assigned 100 milliseconds to each task, and switching between tasks consumed 8 milliseconds. One full circuit of the task list would require 972 milliseconds (9 X 100) + (9 X 8) to complete.

Multitasking



Multitasking

A multitasking OS runs on a processor (such as the x86) that supports task switching. The processor saves the state of each task before switching to a new one. A task's state consists of the contents of the processor registers, program counter, and status flags, along with references to the task's memory segments. A multitasking OS will usually assign varying priorities to tasks, giving them relatively larger or smaller time slices. A preemptive multitasking OS (such as Windows XP or Linux) permits a higher-priority task to interrupt a lower-priority one, leading to better system stability. Suppose an application program is locked in loop and has stopped responding to input. The keyboard handler (a high-priority OS task) can respond to the user's Ctrl-Alt-Del command and shut down the buggy application program.

x86 Architecture Details

Modes of Operation

- **Protected mode** is the native state of the processor, in which all instructions and features are available. Programs are given separate memory areas named segments, and the processor prevents programs from referencing memory outside their assigned segments.
- **Virtual-8086 Mode**: While in protected mode, the processor can directly execute real-address mode software such as MS-DOS programs in a safe multitasking environment. In other words, if an MS-DOS program crashes or attempts to write data into the system memory area, it will not affect other programs running at the same time. Windows XP can execute multiple separate virtual-8086 sessions at the same time.

Modes of Operation

- **Real-address mode** implements the programming environment of the Intel 8086 processor with a few extra features, such as the ability to switch into other modes. This mode is available in Windows 98, and can be used to run an MS-DOS program that requires direct access to system memory and hardware devices. Programs running in real-address mode can cause the operating system to crash (stop responding to commands).
- **System Management mode (SMM)** provides an operating system with a mechanism for implementing functions such as power management and system security. These functions are usually implemented by computer manufacturers who customize the processor for a particular system setup.

Basic Execution Environment

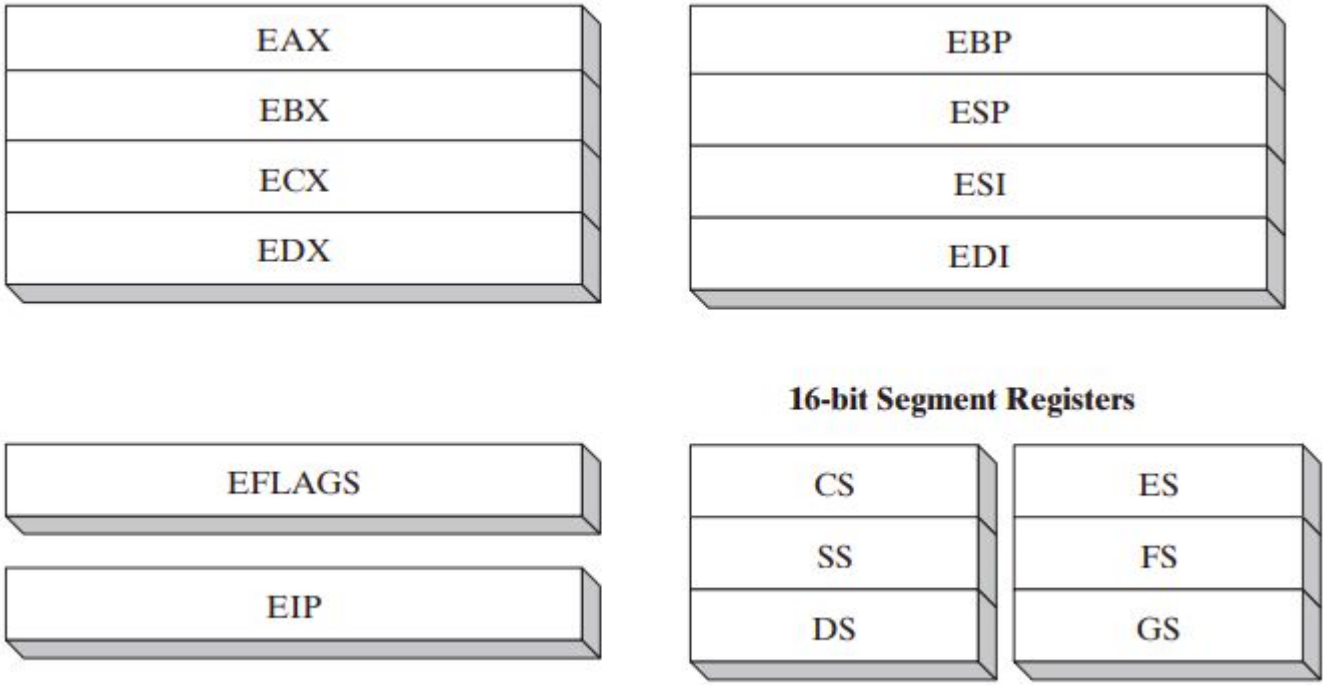
Address Space

- In 32-bit **protected mode**, a task or program can address a linear address space of up to 4 GBytes. Beginning with the P6 processor, a technique called Extended Physical Addressing allows a total of 64 GBytes of physical memory to be addressed.
- **Real-address mode** programs, can only address a range of 1 MByte
- If the processor is in protected mode and running multiple programs in **virtual-8086 mode**, each program has its own 1-MByte memory area.

Basic Program Execution Registers

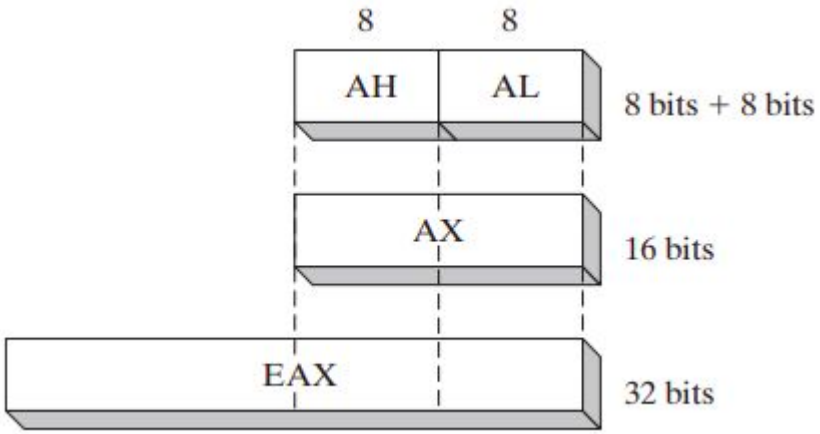
- Registers are high-speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory. When a processing loop is optimized for speed, for example, loop counters are held in registers rather than variables.
- There are eight general-purpose registers, six segment registers, a processor status flags register (EFLAGS), and an instruction pointer (EIP).

32-Bit General Purpose Registers



EAX Register

The general-purpose registers are primarily used for arithmetic and data movement.



32-Bit General Purpose Registers

32-Bit	16-Bit	8-Bit (High)	8-Bit (Low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

32-Bit General Purpose Registers

The remaining general-purpose registers can only be accessed using 32-bit or 16-bit names.

32-Bit	16-Bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Specialized Uses

- **EAX** is automatically used by multiplication and division instructions. It is often called the extended accumulator register
- The CPU automatically uses **ECX** as a loop counter.
- **ESP** addresses data on the stack (a system memory structure). It is rarely used for ordinary arithmetic or data transfer. It is often called the extended stack pointer register.
- **ESI** and **EDI** are used by high-speed memory transfer instructions. They are sometimes called the extended source index and extended destination index registers
- **EBP** is used by high-level languages to reference function parameters and local variables on the stack. It should not be used for ordinary arithmetic or data transfer except at an advanced level of programming. It is often called the extended frame pointer register.

Segment Registers

- In real-address mode, 16-bit **segment registers** indicate base addresses of preassigned memory areas named segments.
- In protected mode, **segment registers** hold pointers to segment descriptor tables. Some segments hold **program instructions (code)**, others hold **variables (data)**, and another segment named the **stack segment** holds **local function variables and function parameters**.

Instruction Pointer

The EIP, or instruction pointer, register contains the address of the next instruction to be executed. Certain machine instructions manipulate EIP, causing the program to branch to a new location.

EFLAGS Register

The EFLAGS (or just Flags) register consists of individual binary bits that control the operation of the CPU or reflect the outcome of some CPU operation. Some instructions test and manipulate individual processor flags.

A ***flag*** is set when it equals 1; it is clear (or reset) when it equals 0.

Control Flags, Status Flags

- **Control flags** control the CPU’s operation. For example, they can cause the CPU to break after every instruction executes, interrupt when arithmetic overflow is detected, enter virtual-8086 mode, and enter protected mode.
- **The Status flags** reflect the outcomes of arithmetic and logical operations performed by the CPU. They are the Overflow, Sign, Zero, Auxiliary Carry, Parity, and Carry flags.

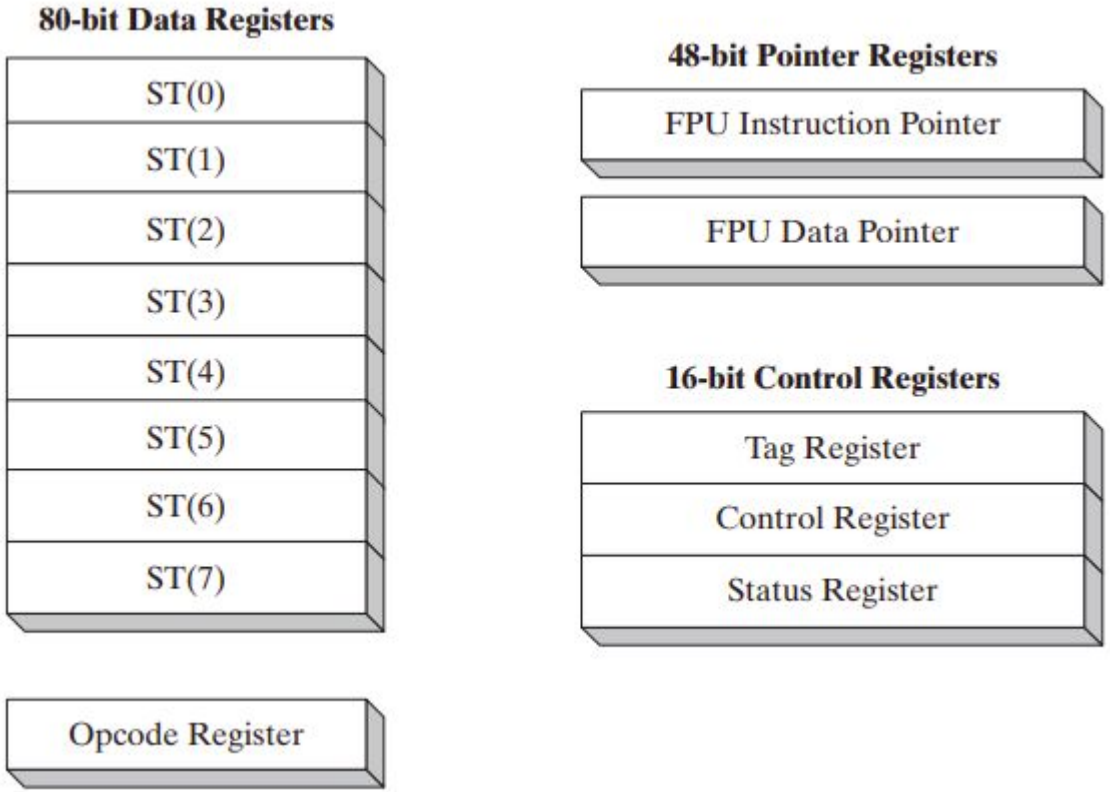
Status Flags

- The **Carry flag (CF)** is set when the result of an unsigned arithmetic operation is too large to fit into the destination
- The **Overflow flag (OF)** is set when the result of a signed arithmetic operation is too large or too small to fit into the destination.
- The **Sign flag (SF)** is set when the result of an arithmetic or logical operation generates a negative result.
- The **Zero flag (ZF)** is set when the result of an arithmetic or logical operation generates a result of zero.
- The **Auxiliary Carry flag (AC)** is set when an arithmetic operation causes a carry from bit 3 to bit 4 in an 8-bit operand.
- The **Parity flag (PF)** is set if the least-significant byte in the result contains an even number of 1 bits. Otherwise, PF is clear. In general, it is used for error checking when there is a possibility that data might be altered or corrupted.

Floating-Point Unit

- The **floating-point unit (FPU)** performs high-speed floating-point arithmetic. At one time a separate coprocessor chip was required for this. From the Intel486 onward, the FPU has been integrated into the main processor chip. There are eight floating-point data registers in the FPU, named **ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), and ST(7)**.

Floating-Point Unit Registers



x86 Memory Management

Real-Address Mode

In **real-address mode**, only 1 MByte of memory can be addressed, from hexadecimal 00000 to FFFFF. The processor can run only one program at a time, but it can momentarily interrupt that program to process requests (called interrupts) from peripherals. Application programs are permitted to access any memory location, including addresses that are linked directly to system hardware. The **MS-DOS** operating system runs in real-address mode, and **Windows 95** and **98** can be booted into this mode.

Protected Mode

In **protected mode**, the processor can run multiple programs at the same time. It assigns each process (running program) a total of 4 GByte of memory. Each program can be assigned its own reserved memory area, and programs are prevented from accidentally accessing each other's code and data. **MS-Windows** and **Linux** run in protected mode.

Virtual-8086 Mode

In **virtual-8086 mode**, the computer runs in protected mode and creates a virtual 8086 machine with its own 1-MByte address space that simulates an 80x86 computer running in **real address mode**. Windows NT and 2000, for example, create a virtual 8086 machine when you open a Command window. You can run many such windows at the same time, and each is protected from the actions of the others. Some MS-DOS programs that make direct references to computer hardware will not run in this mode under Windows NT, 2000, and XP.

Segment Registers In Protected Mode

Segment registers (CS, DS, SS, ES, FS, GS) point to segment descriptor tables, which the operating system uses to keep track of locations of individual program segments. A typical protected-mode program has three segments: code, data, and stack, using the CS, DS, and SS segment registers:

- **CS** references the descriptor table for the **code** segment
- **DS** references the descriptor table for the **data** segment
- **SS** references the descriptor table for the **stack** segment

Basic Elements of Assembly Language

Assembling, Linking, and Running Programs

- **Assembler** - a source program written in assembly language cannot be executed directly on its target computer. It must be translated, or assembled into executable code.
- The assembler produces a file containing machine language called an **object file**. This file isn't quite ready to execute. It must be passed to another program called a **linker**, which in turn produces an executable file.

The Assemble-Link-Execute Cycle

- Step 1: A programmer uses a text editor to create an ASCII text file named the ***source file***.
- Step 2: The assembler reads the source file and produces an ***object file***, a machine-language translation of the program.
- Step 3: The linker reads the object file and checks to see if the program contains any calls to procedures in a link library. The linker copies any required procedures from the link library, combines them with the object file, and produces the ***executable file***.
- Step 4: The ***operating system loader*** utility reads the executable file into memory and branches the CPU to the program's starting address, and the program begins to execute.

Compiling and Linking an Assembly Program in NASM

- To assemble the program, type **nasm -f elf64 -o hello.o hello.asm**
- If there is any error, you will be prompted about that at this stage. Otherwise, an object file of your program named **hello.o** will be created.
- To link the object file and create an executable file named **hello**, type **ld hello.o -o hello**
- Execute the program by typing **./hello**

NASM Assembly Sections

- The **data section** is used for declaring initialized data or constants. This data does not change at runtime - ***section.data***
- The **bss section** is used for declaring variables - ***section.bss***
- The **text section** is used for keeping the actual code. This section must begin with the declaration **global _start**, which tells the kernel where the program execution begins.

```
section.text
   global _start
_start:
```

NASM: Comments

Assembly language comment begins with a semicolon (;) - `; This program displays a message on screen`

add eax, ebx ; adds ebx to eax

Assembly Language Statements

Assembly language programs consist of three types of statements −

- Executable instructions or instructions,
- Assembler directives or pseudo-ops, and
- Macros.

Executable Instruction

The **executable instructions** or simply instructions tell the processor what to do. Each instruction consists of an **operation code (opcode)**. Each executable instruction generates one machine language instruction.

Assembler Directives, Macros

- The **assembler directives** or pseudo-ops tell the assembler about the various aspects of the assembly process. These are non-executable and do not generate machine language instructions.
- **Macros** are basically a text substitution mechanism.

Data Types

Type	Usage
BYTE	8-bit unsigned integer. B stands for byte
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer (can also be a Near pointer in real-address mode)
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer (can also be a Near pointer in protected mode). D stands for double
SDWORD	32-bit signed integer. SD stands for signed double
FWORD	48-bit integer (Far pointer in protected mode)
QWORD	64-bit integer. Q stands for quad

Data Directives

Directive	Usage
DB	8-bit integer
DW	16-bit integer
DD	32-bit integer or real
DQ	64-bit integer or real
DT	define 80-bit (10-byte) integer