

Module III

Computer abstractions and technology - Introduction, Computer architecture -8 Design features, Application program - layers of abstraction, Five key components of a computer, Technologies for building processors and memory, Performance, Instruction set principles – Introduction, Classifying instruction set architectures, Memory addressing, Encoding an instruction set.

Introduction - Classes of Computing Applications and Their Characteristics

☐ Personal computers

- o A computer designed for use by an individual, usually incorporating a graphics display, a keyboard, and a mouse.
- o Personal computers emphasize delivery of good performance to single users at low cost and usually execute third-party software.
- o This class of computing drove the evolution of many computing technologies, which is only about 35 years old!

☐ Server computers

- o A computer used for running larger programs for multiple users, often simultaneously, and typically accessed only via a network.
- o Servers are built from the same basic technology as desktop computers, but provide for greater computing, storage, and input/output capacity.

☐ Supercomputers

- o A class of computers with the highest performance and cost
- o Supercomputers consist of tens of thousands of processors and many terabytes of memory, and cost tens to hundreds of millions of dollars.
- o Supercomputers are usually used for high-end scientific and engineering calculations

☐ Embedded computers

- o A computer inside another device used for running one predetermined application or collection of software.
- o Embedded computers include the microprocessors found in your car, the computers in a television set, and the networks of processors that control a modern airplane or cargo ship.
- o Embedded computing systems are designed to run one application or one set of related applications that are normally integrated with the hardware and delivered as a single system; thus, despite the large number of embedded computers, most users never really see that they are using a computer!

8 Great Ideas in Computer Architecture

Application of the following great ideas has accounted for much of the tremendous growth in computing capabilities over the past 50 years.

- Design for Moore's law
- Use abstraction to simplify design
- Make the common case fast
- Performance via parallelism
- Performance via pipelining
- Performance via prediction
- Hierarchy of memories
- Dependability via redundancy

Design for Moore's Law

Gordon Moore, one of the founders of Intel made a prediction in 1965 that integrated circuit resources would double every 18–24 months. This prediction has held approximately true for the past 50 years. It is now known as Moore's Law.

When computer architects are designing or upgrading the design of a processor they must anticipate where the competition will be in 3-5 years when the new processor reaches the market. Targeting the design to be just a little bit better than today's competition is not good enough.

Use Abstraction to Simplify Design

Abstraction uses multiple levels with each level hiding the details of levels below it. For example:

- The instruction set of a processor hides the details of the activities involved in executing an instruction.
- High-level languages hide the details of the sequence of instructions need to accomplish a task.
- Operating systems hide the details involved in handling input and output devices.

Make the Common Case Fast

The most significant improvements in computer performance come from improvements to the common case areas where the current design is spending the most time.

This idea is sometimes called Amdahl's law, though it is preferable to use that term to refer to a mathematical law for analyzing improvements. The mathematical law also is closely related to the law of diminishing returns.

Performance via Parallelism

Doing different parts of a task in parallel accomplishes the task in less time than doing them sequentially. A processor engages in several activities in the execution of an instruction. It runs faster if it can do these activities in parallel.

Performance via Pipelining

This idea is an extension of the idea of parallelism. It is essentially handling the activities involved in instruction execution as an assembly line. As soon as the first activity of an instruction is done you move it to the second activity **and** start the first activity of a new instruction. This results in executing more instructions per unit time compared to waiting for all activities of the first instruction to complete before starting the second instruction.

Performance via Prediction

A *conditional branch* is a type of instruction determines the next instruction to be executed based on a condition test. Conditional branches are essential for implementing high-level language **if** statements and loops.

Unfortunately, conditional branches interfere with the smooth operation of a pipeline — the processor does not know where to fetch the next instruction until after the condition has been tested.

Many modern processors reduce the impact of branches with *speculative execution*: make an informed guess about the outcome of the condition test and start executing the indicated instruction. Performance is improved if the guesses are reasonably accurate and the penalty of wrong guesses is not too severe.

Hierarchy of Memories

The principle of locality states that memory that has been accessed recently is likely to be accessed again in the near future. That is, accessing recently accessed data is a common case for memory accesses. To make this common case faster you need a *cache* — a small high-speed memory designed to hold recently accessed data.

Modern processors use as many as 3 levels of caches. This is motivated by the large difference in speed between processors and memory.

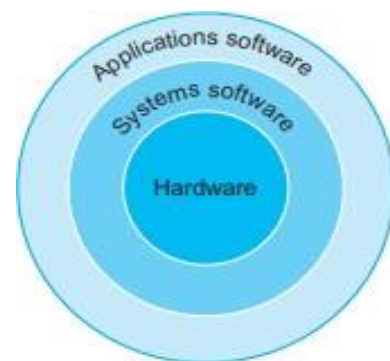
Dependability via Redundancy

One of the most important ideas in data storage is the Redundant Array of Inexpensive Disks (RAID) concept. In most versions of RAID, data is stored redundantly on multiple disks. The redundancy insures that if one disk fails the data can be recovered from other disks.

Layers of Abstraction

Layers of software are organized primarily in a hierarchical fashion, with applications being the outermost ring and a variety of systems software sitting between the hardware and applications software.

These layers of software are organized in hierarchical fashion, with applications being the outermost ring and a variety of system software sitting between the hardware and application software.



Hardware in a computer can only execute extremely simpler low level instructions . To go from complex applications to simple instructions involves several layers of software that interpret/translate high level operations into simple computer instructions.

Application programm :

A typical application, such as a word processor or a large database system, may consist of millions of lines of code and rely on sophisticated software libraries that implement complex functions in support of the application. Application programming aims to produce software which provides services to the user directly (e.g. word processor).

System software:

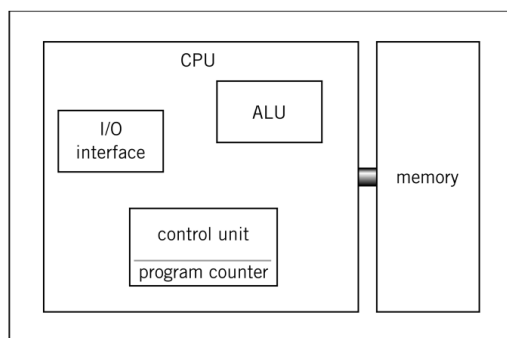
It is a type of computer program that is designed to run computer's hardware and application programs. It provides a platform to other software. It is the interface between hardware and use applications. Eg: Operating Systems, Compilers ,

Assemblers. Operating System is a Supervising program that manages the resources of a computer for the benefit of the programs that run on that machine. Compiler is a program that translates high level language statements into assembly language statements. Assembly language is a symbolic representation of machine instructions . Assembler is a program that translates a symbolic version of instructions into binary version.

Five key components of a computer

- INPUT
- OUTPUT
- MEMORY
- DATAPATH
- CONTROL

System Block Diagram



DATAPATH

A data path (also written as datapath) is a set of functional units ,such as arithmetic logic units or multipliers that perform data processing operations, registers, and buses. Functional units carry out data processing operations.

Datapath, along with a control unit, make up the CPU (central processing unit) of a computer system.

CPU

- ALU (arithmetic logic unit) – Performs calculations and comparisons (data changes)
- CU (control unit): performs fetch/execute cycle
 - Functions:
 - Moves data to and from CPU registers and other hardware components (no change in data)
 - Accesses program instructions and issues commands to the ALU
 - Subparts:
 - Memory management unit: supervises fetching instructions and data
 - I/O Interface: sometimes combined with memory management unit as Bust Interface Unit
 - Registers
 - Example: Program counter (PC) or instruction pointer determines next instruction for execution

-ALU - Arithmetic/Logic Unit

Actual computation (the manipulation of data to generate “new” data) takes place in the ALU. Because data is represented in binary form (1’s and 0’s), the ALU is mainly comprised of logic gates, circuits made from transistors that take inputs (combinations of highs and lows) and produce outputs (different combinations of highs and lows). Logic in which the output depends solely on the input is called combinatorial.

Registers

- Small, permanent storage locations within the CPU used for a particular purpose. Manipulated directly by the Control Unit .Wired for specific function. Size in bits or bytes (not MB like memory) . Can hold data, an address or an instruction
- Use of Registers
 - Scratchpad for currently executing program
 - Holds data needed quickly or frequently. Stores information about status of CPU and currently executing program. Address of next program instruction. Signals from external devices
 - General Purpose Registers
 - User-visible registers
 - Hold intermediate results or data values, e.g., loop counters
 - Typically several dozen in current CPUs

Special-Purpose Registers

- Program Count Register (PC)
 - Also called instruction pointer
- Instruction Register (IR)
 - Stores instruction fetched from memory
- Memory Address Register (MAR)
- Memory Data Register (MDR)
- Status Registers
 - Status of CPU and currently executing program. Flags (one bit Boolean variable) to track condition like arithmetic carry and overflow, power failure, internal computer error.

Control Unit

- Control is responsible for determining what action is to be performed on what data. If the action is a calculation, then control will deliver the necessary data to the ALU, inform the ALU what particular action is to be performed, and then directs the output to the appropriate location.

All of the other units of a computer have “control inputs” that determine whether or not they are active and what particular action they will perform (if they can do more than one thing). Control takes code down to this lowest level of making the appropriate control inputs high or low, active or inactive. – Not all devices are active when the control input is high, some devices are “active low.”

Memory

Memory consists of circuits whose primary purpose is to hold information, but only temporarily. Memory holds the data that the processor has just acted on, is acting on or will soon act on. Each memory location has a unique address • Address from an instruction is copied to the MAR which finds the location in memory .CPU determines if it is a store or retrieval . Transfer takes place between the MDR and memory

Types of memory :

RAM: Random Access Memory

- DRAM (Dynamic RAM)
 - Most common, cheap
 - Volatile: must be refreshed (recharged with power) 1000's of times each second
- SRAM (static RAM)
 - Faster than DRAM and more expensive than DRAM
 - Volatile
 - Frequently small amount used in cache memory for high speed access use

ROM - Read Only Memory

- Non-volatile memory to hold software that is not expected to change over the life of the system
- Magnetic core memory
- EEPROM
 - Electrically Erasable Programmable ROM
 - Slower and less flexible than Flash ROM
- Flash ROM
 - Faster than disks but more expensive
 - Uses
 - BIOS: initial boot instructions and diagnostics
 - Digital cameras

Technologies for building processors and memory

Processors and memory have improved at an incredible rate, because computer designers have long embraced the latest in electronic technology to try to win the race to design a better computer. Table below shows the technologies that have been used over time, with an estimate of the relative performance per unit cost for each technology.

| Year | Technology used in computers | Relative performance/unit cost |
|------|--------------------------------------|--------------------------------|
| 1951 | Vacuum tube | 1 |
| 1965 | Transistor | 35 |
| 1975 | Integrated circuit | 900 |
| 1995 | Very large-scale integrated circuit | 2,400,000 |
| 2013 | Ultra large-scale integrated circuit | 250,000,000,000 |

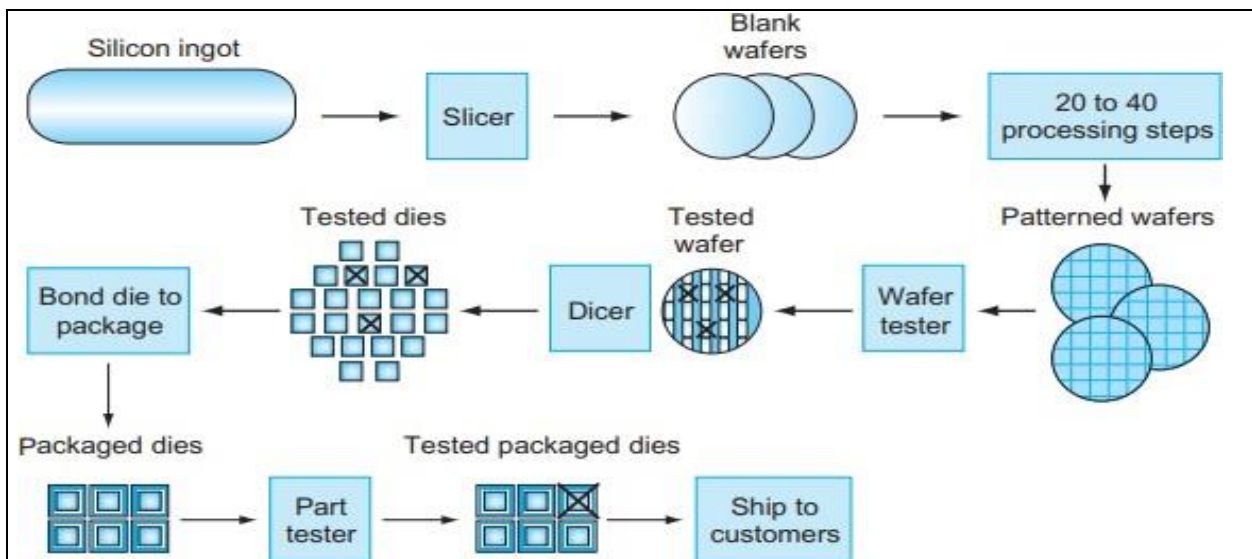
To understand how manufacture integrated circuits, we start at the beginning. The manufacture of a chip begins with silicon, a substance found in sand. Because silicon does not conduct electricity well, it is called a semiconductor. With a special chemical process, it is possible to add materials to silicon that allow tiny areas to transform into one of three devices:

- Excellent conductors of electricity (using either microscopic copper or aluminum wire).
- Excellent insulators from electricity (like plastic sheathing or glass) .
- Areas that can conduct or insulate under special conditions (as a switch).

Manufacturing ICs

Silicon crystal igot: A rod composed of a silicon crystal that is between 8 and 12 inches in diameter and about 12 to 24 inches long .

Wafer: A slice from a silicon igot no more than 0.1 inches thick, used to create chips.



Wafers then go through a series of processing steps, during which patterns of chemicals are placed on each wafer.

A single microscopic flaw in the wafer itself or in one of the dozens of patterning steps can result in that area of the wafer failing. These **defects**, as they are called, make it virtually impossible to manufacture a perfect wafer.

The simplest way to cope with imperfection is to place many independent components on a single wafer. The patterned wafer is then chopped up, or diced, into these components, called **dies** and more informally known as chips.

❑ **die** → The individual rectangular sections that are cut from a wafer, more informally known as chips.

❑ **Yield** → The percentage of good dies from the total number of dies on the wafer.

Once you've found good dies, they are connected to the input/output pins of a package, using a process called **bonding**. These packaged parts are tested a final time, since mistakes can occur in packaging, and then they are shipped to customers.

Elaboration: The cost of an integrated circuit can be expressed in three simple equations:

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{yield}}$$

$$\text{Dies per wafer} \approx \frac{\text{Wafer area}}{\text{Die area}}$$

$$\text{Yield} = \frac{1}{(1 + (\text{Defects per area} \times \text{Die area}/2))^2}$$

Performance

In most cases, we will need different performance metrics as well as different sets of applications to benchmark personal mobile devices, which are more focused on response time, versus servers, which are more focused on throughput.

Response time : Also called execution time. The total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time, and so on.

Throughput : Also called bandwidth. Another measure of performance, it is the number of tasks completed per unit time.

To maximize performance, we want to minimize response time or execution time for some task. Thus, we can relate performance and execution time for a computer X:

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\begin{aligned}\text{Performance}_X &> \text{Performance}_Y \\ \frac{1}{\text{Execution time}_X} &> \frac{1}{\text{Execution time}_Y} \\ \text{Execution time}_Y &> \text{Execution time}_X\end{aligned}$$

That is, the execution time on Y is longer than that on X, if X is faster than Y.

In discussing a computer design, we often want to relate the performance of two different computers quantitatively. We will use the phrase “X is n times faster than Y”—or equivalently “X is n times as fast as Y”—to mean

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

If X is n times as fast as Y, then the execution time on Y is n times as long as it is on X:

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?

We know that A is n times as fast as B if

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Thus the performance ratio is

$$\frac{15}{10} = 1.5$$

and A is therefore 1.5 times as fast as B.

In the above example, we could also say that computer B is 1.5 times *slower than* computer A, since

$$\frac{\text{Performance}_A}{\text{Performance}_B} = 1.5$$

means that

$$\frac{\text{Performance}_A}{1.5} = \text{Performance}_B$$

Measuring Performance

Time is the measure of computer performance: the computer that performs the same amount of work in the least time is the fastest. Program execution time is measured in seconds per

program. Computers are often shared, however, and a processor may work on several programs simultaneously.

CPU execution time or simply CPU time is the time the CPU spends computing for the task and does not include time spent waiting for I/O or running other programs. CPU time can be further divided into the CPU time spent in the program, called user CPU time, and the CPU time spent in the operating system performing tasks on behalf of the program, called system CPU time.

All computers are constructed using a clock that determines when events take place in the hardware. These discrete time intervals are called clock cycles (or ticks, clock ticks, clock periods, clocks, cycles). Designers refer to the length of a clock period both as the time for a complete clock cycle (e.g., 250 picoseconds, or 250 ps) and as the clock rate (e.g., 4 gigahertz, or 4 GHz), which is the inverse of the clock period.

$$\text{CPU execution time for a program} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}} \times \text{Clock cycle time}$$

Alternatively, because clock rate and clock cycle time are inverses,

$$\text{CPU execution time for a program} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

This formula makes it clear that the hardware designer can improve performance by **reducing the number of clock cycles** required for a program or the length of the clock cycle.

EXAMPLE

Our favorite program runs in 10 seconds on computer A, which has a 2 GHz clock. We are trying to help a computer designer build a computer, B, which will run this program in 6 seconds. The designer has determined that a substantial increase in the clock rate is possible, but this increase will affect the rest of the CPU design, causing computer B to require 1.2 times as many clock cycles as computer A for this program. What clock rate should we tell the designer to target?

ANSWER

Let's first find the number of clock cycles required for the program on A:

$$\text{CPU time}_A = \frac{\text{CPU clock cycles}_A}{\text{Clock rate}_A}$$

$$10 \text{ seconds} = \frac{\text{CPU clock cycles}_A}{2 \times 10^9 \frac{\text{cycles}}{\text{second}}}$$

$$\text{CPU clock cycles}_A = 10 \text{ seconds} \times 2 \times 10^9 \frac{\text{cycles}}{\text{second}} = 20 \times 10^9 \text{ cycles}$$

$$\text{CPU time}_B = \frac{1.2 \times \text{CPU clock cycles}_A}{\text{Clock rate}_B}$$

$$6 \text{ seconds} = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{\text{Clock rate}_B}$$

$$\text{Clock rate}_B = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{6 \text{ seconds}} = \frac{0.2 \times 20 \times 10^9 \text{ cycles}}{\text{second}} = \frac{4 \times 10^9 \text{ cycles}}{\text{second}} = 4 \text{ GHz}$$

To run the program in 6 seconds, B must have twice the clock rate of A.

Instruction Performance

The performance equations above did not include any reference to the number of instructions needed for the program. However, since the compiler clearly generated instructions to execute, and the computer had to execute the instructions to run the program, the execution time must depend on the number of instructions in a program.

One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction. Therefore, the number of clock cycles required for a program can be written as,

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \text{Average clock cycles per instruction}$$

The term clock cycles per instruction, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as CPI. Since different instructions may take different amounts of time depending on what they do, CPI is an average of all the instructions executed in the program.

Using the Performance Equation

Suppose we have two implementations of the same instruction set architecture. Computer A has a clock cycle time of 250 ps and a CPI of 2.0 for some program, and computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program. Which computer is faster for this program and by how much?

EXAMPLE

ANSWER

We know that each computer executes the same number of instructions for the program; let's call this number I . First, find the number of processor clock cycles for each computer:

$$\text{CPU clock cycles}_A = I \times 2.0$$

$$\text{CPU clock cycles}_B = I \times 1.2$$

Now we can compute the CPU time for each computer:

$$\begin{aligned}\text{CPU time}_A &= \text{CPU clock cycles}_A \times \text{Clock cycle time} \\ &= I \times 2.0 \times 250 \text{ ps} = 500 \times I \text{ ps}\end{aligned}$$

Likewise, for B:

$$\text{CPU time}_B = I \times 1.2 \times 500 \text{ ps} = 600 \times I \text{ ps}$$

Clearly, computer A is faster. The amount faster is given by the ratio of the execution times:

$$\frac{\text{CPU performance}_A}{\text{CPU performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{600 \times I \text{ ps}}{500 \times I \text{ ps}} = 1.2$$

We can conclude that computer A is 1.2 times as fast as computer B for this program.

The Classic CPU Performance Equation

The basic performance equation in terms of instruction count (the number of instructions executed by the program), CPI, and clock cycle time:

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

or, since the clock rate is the inverse of clock cycle time:

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

In order to use the hardware of a computer, we must speak its language. The words of a computer language are called instructions, and its vocabulary is called an instruction set.

Instruction set of a computer is the portion of the computer visible to the assembly level programmer or to the compiler writer.

The ISA serves as the boundary between software and hardware.

INSTRUCTION SET ARCHITECTURES

The 3 most common types of ISAs are:

- ❑ **Stack** - The operands are implicitly on top of the stack.
- ❑ **Accumulator** - One operand is implicitly the accumulator.
- ❑ **General Purpose Register (GPR)** - All operands are explicitly mentioned, they are either registers or memory locations.

Accumulator Based Machines

Accumulator based machines use special registers called the accumulators to hold one source operand and also the result of the arithmetic or logic operations performed. Thus the accumulator registers collect (or 'accumulate') data.

Since the accumulator holds one of the operands, one more register may be required to hold the address of another operand. Accumulator machines employ a very small number of accumulator registers, generally only one. These machines were useful at the time when memory was quite expensive.

However, now that the memory is relatively inexpensive, these are not considered very useful, and their use is severely limited for the computation of expressions with many operands.

Stack Based Machines

A stack is a group of registers organized as a last-in-first-out (LIFO) structure.

In such a structure, the operands stored first, through the push operation, can only be accessed last, through a pop operation; the order of access to the operands is reverse of the storage operation.

Arithmetic and logic operations successively pick operands from the top-of-the-stack (TOS), and push the results on the TOS at the end of the operation.

In stack based machines, operand addresses need not be specified during the arithmetic or logical operations. Therefore, these machines are also called 0-address machines.

General-Purpose-Register Machines

In general purpose register machines, a number of registers are available within the CPU. These registers do not have dedicated functions, and can be employed for a variety of purposes.

To identify the register within an instruction, a small number of bits are required in an instruction word. For example, to identify one of the 64 registers of the CPU, a 6-bit field is required in the instruction.

CPU registers are faster than cache memory. Registers are also easily and more effectively used by the compiler compared to other forms of internal storage. Registers can also be used to hold variables, thereby reducing memory traffic.

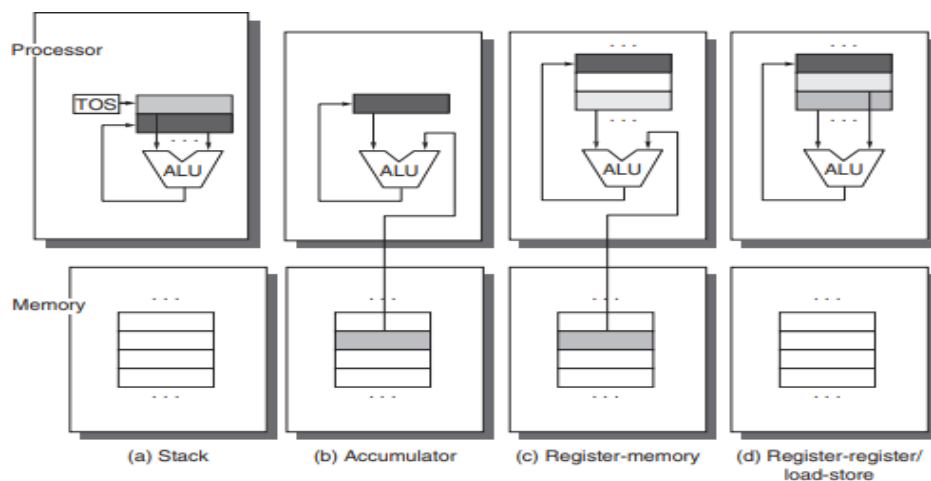


Figure shows how the code sequence $C = A + B$ would typically appear in these three classes of instruction sets. The explicit operands may be accessed directly from memory or may need to be first loaded into temporary storage, depending on the class of architecture and choice of specific instruction.

| Stack | Accumulator | Register (register-memory) | Register (load-store) |
|--------|-------------|-------------------------------|-----------------------|
| Push A | Load A | Load R1,A | Load R1,A |
| Push B | Add B | Add R3,R1,B | Load R2,B |
| Add | Store C | Store R3,C | Add R3,R1,R2 |
| Pop C | | | Store R3,C |

As the figures show, there are really two classes of register computers.

One class can access memory as part of any instruction, called register- memory architecture, and the other can access memory only with load and store instructions, called load-store architecture.

A third class, not found in computers shipping today, keeps all operands in memory and is called a memory-memory architecture. Although most early computers used stack or accumulator-style architectures, virtually every new architecture designed after 1980 uses a load-store register architecture.

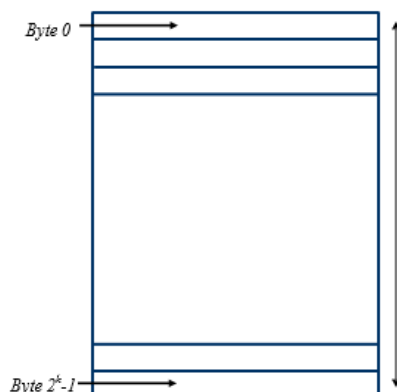
The major reasons for the emergence of general-purpose register (GPR) computers are twofold. First, registers—like other forms of storage internal to the processor—are faster than memory. Second, registers are more efficient.

Advantages And Disadvantages

| Type | Advantages | Disadvantages |
|-----------------------------------|---|---|
| Register-register (0, 3) | Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see Appendix C). | Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density lead to larger programs. |
| Register-memory (1, 2) | Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density. | Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location. |
| Memory-memory (2, 2) or (3, 3) | Most compact. Doesn't waste registers for temporaries. | Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.) |

MEMORY ADDRESSING

Information is stored in the memory as a collection of bits. Collection of 8 bits known as a “byte”. Bytes are grouped into words □ Collection of bits stored or retrieved simultaneously.

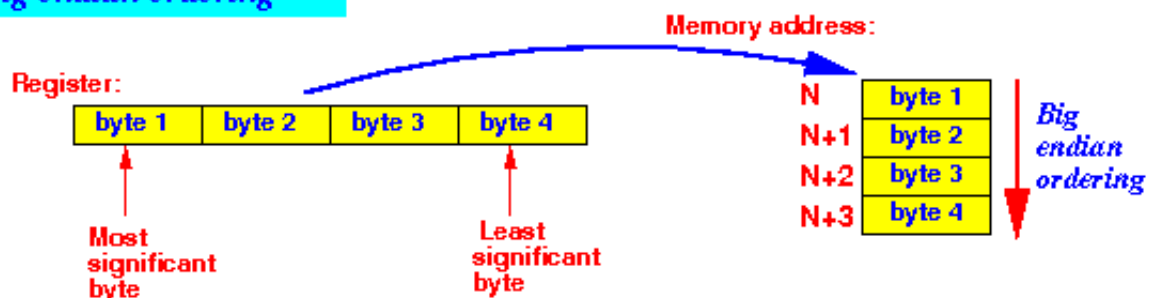


- Memory is viewed as a sequence of bytes.
- Address of the first byte is 0
- Address of the last byte is $2^k - 1$, where k is the number of bits used to hold memory address
- E.g. when $k = 16$,
Address of the first byte is 0
Address of the last byte is 65535

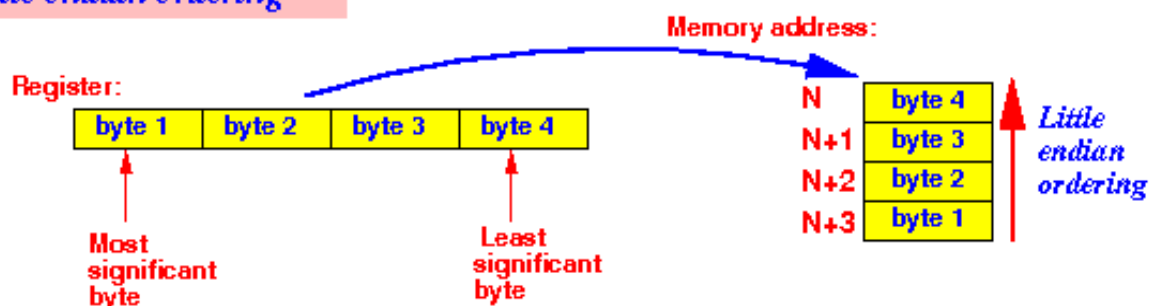
BIG-ENDIAN & LITTLE-ENDIAN ASSIGNMENTS

- ❑ Endianness is a term that describes the order in which a sequence of bytes is stored in computer memory.
- ❑ Byte address can be assigned across words in 2 ways:
 1. Big-endian : Lower byte addresses are used for the most significant bytes of the words.
 2. Little-endian: Lower byte addresses are used for the less significant bytes of the word.
- ❑ Big-endian is an order in which the "big end" (most significant value in the sequence) is stored first, at the lowest storage address. Little-endian is an order in which the "little end" (least significant value in the sequence) is stored first.
- ❑ In a big-endian computer, the two bytes required for the hexadecimal number 4F52 would be stored as 4F52 in storage.
- ❑ For example, if 4F is stored at storage address 1000, 52 will be at address 1001.
- ❑ In a little-endian system, it would be stored as 524F, with 52 at address 1000 and 4F at 1001.

Big endian ordering



Little endian ordering



ADDRESSING MODES

The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes. When a memory location is used, the actual memory address specified by the addressing mode is called the effective address. The operands of the instructions can be located either in the main memory or in the CPU registers.

If the operand is placed in the main memory, then the instruction provides the location address in the operand field. Many methods are followed to specify the operand address. They are called addressing modes.

Addressing Modes– The term addressing modes refers to the way in which the operand of an instruction is specified. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually executed.

Addressing modes for 8086 instructions are divided into two categories:




- 1) Addressing modes for data
- 2) Addressing modes for branch

Generic Addressing Modes:

- ☐ Register mode
- ☐ Absolute mode
- ☐ Immediate mode
- ☐ Indirect mode
- ☐ Index mode
- ☐ Relative mode
- ☐ Auto-increment mode
- ☐ Auto-decrement mode

| Name | Assembler syntax | Addressing function |
|----------------------------|------------------|-----------------------|
| Immediate | #Value | Operand=Value |
| Register | Ri | EA=Ri |
| Absolute (Direct) | LOC | EA=LOC |
| Indirect | (Ri) | EA=[Ri] |
| | (LOC) | EA=[LOC] |
| Index | X(Ri) | EA=[Ri]+X |
| Base with index | (Ri, Rj) | EA=[Ri]+[Rj] |
| Base with index and offset | X(Ri, Rj) | EA=[Ri]+[Rj]+X |
| Relative | X(PC) | EA=[PC]+X |
| Autoincrement | (Ri)+ | EA=[Ri]; Increment Ri |
| Autodecrement | -(Ri) | Decrement Ri; EA=[Ri] |

1. Register mode

-  Operand is the contents of a processor register.
-  Name (Address) of the register (its Name) is given in the instruction.
-  E.g. Clear R1 or Move R1, R2

In this mode, the operands are in registers that reside within the CPU. The specific register is selected from a register field in the instruction. In register addressing the operand is placed in one of 8 bit or 16 bit general purpose registers. The data is in the register that is specified by the instruction.

Here one register reference is required to access the data.



Example:

MOV AX,CX (move the contents of CX register to AX register)

2. Absolute mode(Direct Mode) –symbol []

- ❑ Operand is in a memory location.
- ❑ Address of the memory location is given explicitly in the instruction.
- ❑ E.g. **Clear A** or **Move LOC, R2**
- ❑ Also called as “Direct mode” in some assembly languages.

The operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction, the address field specifies the actual branch address.

Here only one memory reference operation is required to access the data.



Example:

ADD AL,[0301] / /add the contents of offset address 0301 to AL

3. Immediate mode –Symbol

- Operand is given explicitly in the instruction.
- Move R0,200 immediate

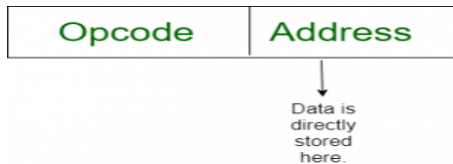
It places the value 200 in the register R0. The immediate mode used to specify the value of source operand. In assembly language, the immediate subscript is not appropriate so ‘#’ symbol is used. It can be re-written as,

Move R0 ,#200

Register, Absolute and Immediate modes contained either the address of the operand or the operand itself.

In this mode data is present in address field of instruction .Designed like one address instruction format.

Note:Limitation in the immediate mode is that the range of constants are restricted by size of address field. In other words, an immediate-mode instruction has an operand field instead of an address field. Immediate-mode instructions are beneficial for initializing registers to a constant value.



Example:

MOV AL, 35H (move the data 35H into AL register)

❑ Immediate mode

❑ Eg: A=B+6

❑ MOV R1,B

❑ ADD R1, #6

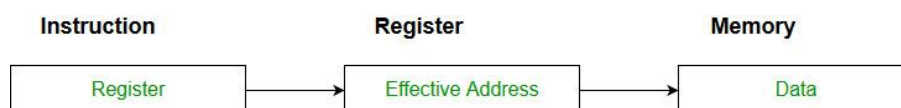
❑ MOV A,R1

4. Indirect mode

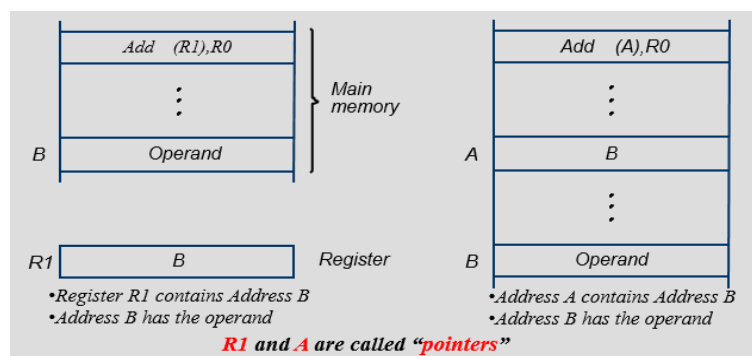
The effective address of the operand is the contents of a register or memory location appears in the instruction. Indirection is denoted by placing the name of the register or the memory address given in the instruction in parentheses. The register or memory location that contains the address of an operand is called a pointer.

❑ (Ri) -> EA=[Ri]

❑ (LOC) -> EA=[LOC]



MOV AX, [BX](move the contents of memory location addressed by the register BX to the register AX)



5. Index mode:

The effective address of the operand is generated by adding a constant value to the contents of a register. The constant value uses either special purpose or general purpose register. This register is referred to as an index register.

The operand's offset is the sum of the content of an index register SI or DI and an 8 bit or 16 bit displacement.

Example:

MOV AX, [SI + 05]

Index mode symbolically denoted as $X(R_i)$.

X is constant value contained in the instruction (offset or displacement).

R_i is the name of the register involved.

The effective address of the operand is given by, $EA = X + [R_i]$

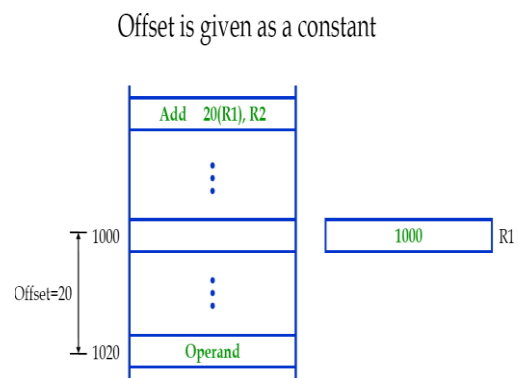
The index register R₁ contains the address of a new location and the value of X defines an offset (also called a displacement).

The contents of the index register are not changed in the process of generating the effective address.

MOV R0, 20(R1)

To find operand,

- First goto Reg R₁ (using the content address)
- Read from R₁-1000
- Add the content 1000 with offset 20 get the result.
- $1000 + 20 = 1020$



6. Relative Mode

Effective Address of the operand is generated by adding a constant value to the contents of the Program Counter (PC).

$X(PC) \rightarrow EA = [PC] + X$

Same as Index Mode à the index register is the PC instead of a general purpose register. Useful for specifying target addresses in branch instructions. Addressed location is "relative" to the PC à "Relative Mode"

7. Auto-increment/decrement mode

In this mode, the instruction specifies a register which points to a memory address that contains the operand. However, after the address stored in the register is accessed, the address is

incremented or decremented, as specified. The next operand is found by the new value stored in the register.

Autodecrement mode

Effective address of the operand is the contents of a register specified in the instruction.

Decrement Ri;

EA=[Ri]

Before accessing the operand, the contents of this register are automatically decremented to point to the previous consecutive memory location.

-(R1)

Encoding An Instruction Set

How to encode instructions as binary values?

- ❑ Instructions consist of:
 - ❑ operation (opcode) e.g. MOV
 - ❑ operands (number depends on operation)
- ❑ operands specified using addressing modes.
- ❑ addressing mode may include **addressing information**.
 - ❑ e.g. registers, constant values
- ❑ **Encoding of instruction must include opcode, operands & addressing information.**

Encoding:

Represent entire instruction as a **binary value**. **Number of bytes** needed depends on how much information must be encoded. Instructions are **encoded by assembler**.

Encoding of instruction depends on the range of addressing modes and the degree of independence between opcodes and addressing modes. Some older computers have one to five operands with 10 addressing modes for each operand. For such a large number of combinations, typically a separate address specifier is needed for each operand. The address specifier tells what addressing mode is used to access the operand. When encoding the instructions, **the number of registers and the number of addressing modes both have a significant impact** on the size of instructions, as the register field and addressing mode field may appear many times in a single instruction. In fact, for most instructions many more bits are consumed in encoding addressing modes and register fields than in specifying the opcode.

The architect must balance several competing forces when encoding the instruction set:

The desire to have as many registers and addressing modes as possible. The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average

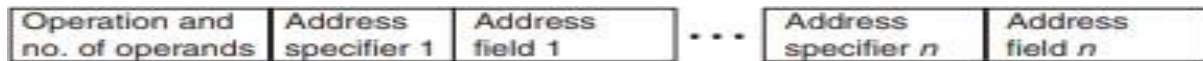
program size. A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation. As a minimum, the architect wants instructions to be in multiples of bytes, rather than an arbitrary bit length.

☐ Three basic variations in instruction encoding:

☐ variable length

☐ fixed length

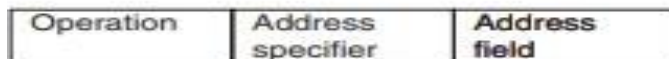
☐ hybrid



(a) Variable (e.g., Intel 80x86, VAX)



(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)



(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)