BCA Sem-3 PAPER: BCAB2103T

Computer System Organization And Architecture

UNIT No. 1

Center for Distance and Online Education, PunjabiUniversity, Patiala

Lesson No:

BCAB2103T: Computer System Organisation and Architecture

Max Marks: 75 **Maximum Time: 3 Hrs**
Min Pass Marks: 35

(A.) INSTRUCTION FOR THE PAPER SETTER

The question paper will consist of three sections A, B and C. Section A and B will have four questions from the respective section of the syllabus carrying 15 marks for each question. Section C will consist of 5-10 short answer type questions carrying a total of 15 marks, which will cover the entire syllabus uniformly. Candidates are required to attempt five questions in all by selecting at least two questions each from the section A and B. Section C is compulsory.

(B.) INSTRUCTIONS FOR THE CANDIDATES

Candidates are required to attempt five questions in all by selecting at least two questions each from the section A and B. Section C is compulsory.

Section-A

Computer System Organisation: CPU Organisation, Instruction Execution, (instruction cycle, types of instructions), RISC v/s CISC, Design principles of modern computers, Instruction level parallelism, Processor level parallelism

Primary Memory: Memory address, byte ordering, Error correcting codes, Cache memory.

Secondary Memory: Memory hierarchy, SCSI disk, RAID.

Instruction Set Architecture: Instruction formats, Expanding opcodes, types of addressing modes, data transfer and manipulation instructions, Program control (status-bit conditions, conditional branch instructions, program interrupt, type of interrupt)

Section-B

Register Transfer Language: Register Transfer, Bus and memory transfer, Arithmetic micro-operations, Logic micro-operations, Shift micro-operations, Arithmetic logic shift unit

Micro-programmed control, control word, control memory (concepts only)

Input-Output Organisation- I/O interfaces (I/O bus and interface modules, I/O versus memory bus, isolated versus memory-mapped I/O)

Asynchronous Data Transfer (strobe control, handshaking), modes of transfer (programmed I/O, interrupt initiated I/O, software considerations), Direct memory access.

Text Books:

1. Jyotsna Sengupta, Fundamentals of Computer Organisation and Architecture, NuTech Books, Deep and Deep Publications, New Delhi.

Reference Books

1. M.Morris Mano, Digital Logic and Computer Design, Prentice Hall of India.
2. Andrew S. Tanenbaum, “Structured Computer Organisation” 4th edition, PHI.
3. J.P.Hayes Tata-Mcgraw Hill, Computer Organisation and Architecture TMH.
4. William Stallings, “Computer System Architecture”, PHI.

BCA SEM-3 **PAPER BCAB2103T COMPUTER SYSTEM ORGANIZATION AND ARCHITECTURE**

LESSON No. 1 **AUTHOR: Dr.VISHAL SINGH**

Introduction

Objectives

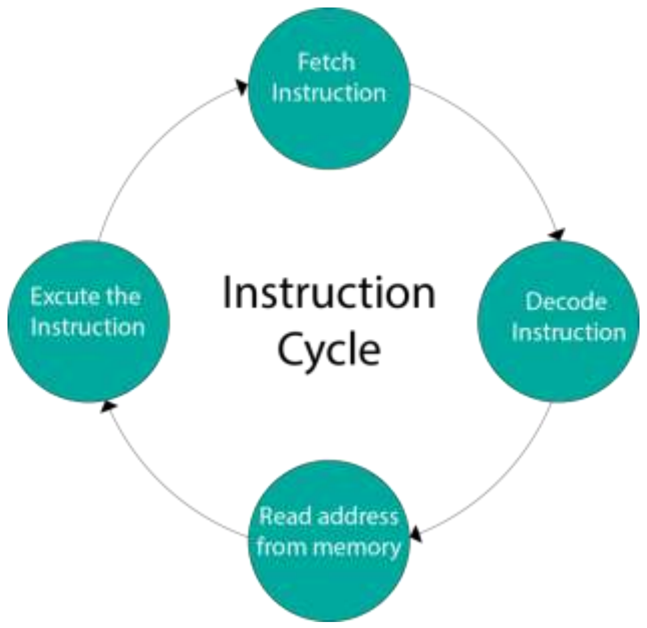
In this lesson we will discuss computer instructions and its types. We will also study RISC and CISC, and the concept of parallelism in detail.

1.1 Introduction: Instruction Cycle

A program residing in the memory unit of a computer consists of a sequence of instructions. These instructions are executed by the processor by going through a cycle for each instruction.

In a basic computer, each instruction cycle consists of the following phases:

1. Fetch instruction from memory.
2. Decode the instruction.
3. Read the effective address from memory.
4. Execute the instruction.

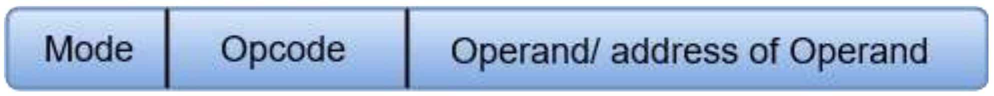


1.2 Types of Computer Instructions

Computer instructions are a set of machine language instructions that a particular processor understands and executes. A computer performs tasks on the basis of the instruction provided.

An instruction comprises of groups called fields. These fields include:

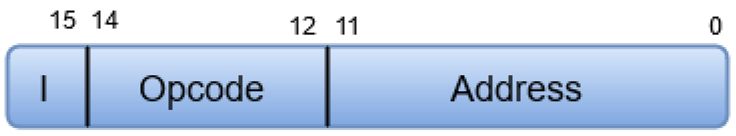
- The Operation code (Opcode) field which specifies the operation to be performed.
- The Address field which contains the location of the operand, i.e., register or memory location.
- The Mode field which specifies how the operand will be located.



A basic computer has three instruction code formats which are:

1. Memory - reference instruction
2. Register - reference instruction
3. Input-Output instruction

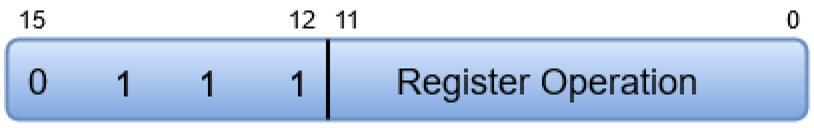
Memory - reference instruction



(Opcode = 000 through 110)

In Memory-reference instruction, 12 bits of memory is used to specify an address and one bit to specify the addressing mode 'I'.

Register - reference instruction



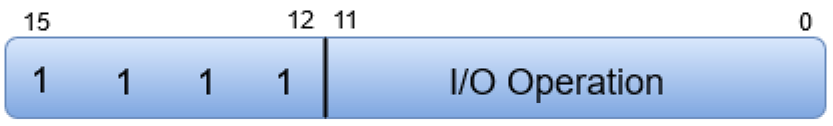
(Opcode = 111, I = 0)

The Register-reference instructions are represented by the Opcode 111 with a 0 in the leftmost bit (bit 15) of the instruction.

Note: The Operation code (Opcode) of an instruction refers to a group of bits that define arithmetic and logic operations such as add, subtract, multiply, shift, and compliment.

A Register-reference instruction specifies an operation on or a test of the AC (Accumulator) register.

Input-Output instruction



(Opcode = 111, I = 1)

Just like the Register-reference instruction, an Input-Output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of the input-output operation or test performed.

Note

- The three operation code bits in positions 12 through 14 should be equal to 111. Otherwise, the instruction is a memory-reference type, and the bit in position 15 is taken as the addressing mode I.
- When the three operation code bits are equal to 111, control unit inspects the bit in position 15. If the bit is 0, the instruction is a register-reference type. Otherwise, the instruction is an input-output type having bit 1 at position 15.

Instruction Set Completeness

A set of instructions is said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

- Arithmetic, logical and shift instructions
- A set of instructions for moving information to and from memory and processor registers.
- Instructions which controls the program together with instructions that check status conditions.
- Input and Output instructions

Arithmetic, logic and shift instructions provide computational capabilities for processing the type of data the user may wish to employ.

A huge amount of binary information is stored in the memory unit, but all computations are done in processor registers. Therefore, one must possess the capability of moving information between these two units.

Program control instructions such as branch instructions are used change the sequence in which the program is executed.

Input and Output instructions act as an interface between the computer and the user. Programs and data must be transferred into memory, and the results of computations must be transferred back to the user.

1.3 CISC

Pronounced sisk, and stands for **C**omplex **I**nstruction **S**et **C**omputer. Most PC's use CPU based on this architecture. For instance Intel and AMD CPU's are based on CISC architectures.

Typically CISC chips have a large amount of different and complex instructions. The philosophy behind it is that hardware is always faster than software, therefore one should make a powerful instruction set, which provides programmers with assembly instructions to do a lot with short programs.

In common CISC chips are relatively slow (compared to RISC chips) per instruction, but use little (less than RISC) instructions.

1.4 RISC

Pronounced risk, and stands for **R**educed **I**nstruction **S**et **C**omputer. RISC chips evolved around the mid-1980 as a reaction at CISC chips. The philosophy behind it is that almost no one uses complex assembly language instructions as used by CISC, and people mostly use compilers which never use complex instructions. Apple for instance uses RISC chips.

Therefore fewer, simpler and faster instructions would be better, than the large, complex and slower CISC instructions. However, more instructions are needed to accomplish a task.

An other advantage of RISC is that - in theory - because of the more simple instructions, RISC chips require fewer transistors, which makes them easier to design and cheaper to produce.

Finally, it's easier to write powerful optimized compilers, since fewer instructions exist.

1.5 RISC vs. CISC

There is still considerable controversy among experts about which architecture is better. Some say that RISC is cheaper and faster and therefore the architecture of the future.

Others note that by making the hardware simpler, RISC puts a greater burden on the software. Software needs to become more complex. Software developers need to write more lines for the same tasks.

Therefore they argue that RISC is not the architecture of the future, since conventional CISC chips are becoming faster and cheaper anyway.

RISC has now existed more than 10 years and hasn't been able to kick CISC out of the market. If we forget about the embedded market and mainly look at the market for PC's, workstations and servers I guess a least 75% of the processors are based on the CISC architecture. Most of them the x86 standard (Intel, AMD, etc.), but even in the mainframe territory CISC is dominant via the IBM/390 chip. Looks like CISC is here to stay …

Is RISC than really not better? The answer isn't quite that simple. RISC and CISC architectures are becoming more and more alike. Many of today's RISC chips support just as many instructions as yesterday's CISC chips. The PowerPC 601, for example, supports *more* instructions than the Pentium. Yet the 601 is considered a RISC chip, while the Pentium is definitely CISC. Further more today's CISC chips use many techniques formerly associated with RISC chips.

So simply said: RISC and CISC are growing to each other.

x86

An important factor is also that the x86 standard, as used by for instance Intel and AMD, is based on CISC architecture. X86 is the standard for home based PC's. Windows 95 and 98 won't run at any other platform. Therefore companies like AMD an Intel will not abandoning the x86 market just overnight even if RISC was more powerful.

Changing their chips in such a way that on the outside they stay compatible with the CISC x86 standard, but use RISC architecture inside is difficult and gives all kinds of overhead which could undo all the possible gains. Nevertheless Intel and AMD are doing this more or less with their current CPU's. Most acceleration mechanisms available to RISC CPUs are now available to the x86 CPU's as well.

Since in the x86 the competition is killing, prices are low, even lower than for most RISC CPU's. Although RISC prices are dropping also a, for instance, SUN UltraSPARC is still more expensive than an equal performing PII workstation is.

Equal that is in terms of integer performance. In the floating point-area RISC still holds the crown. However CISC's 7th generation x86 chips like the K7 will catch up with that.

The one exception to this might be the Alpha EV-6. Those machines are overall about twice as fast as the fastest x86 CPU available. However this Alpha chip costs about €20000, not something you're willing to pay for a home PC.

Maybe interesting to mention is that it's no coincidence that AMD's K7 is developed in co-operation with Alpha and is for al large part based on the same Alpha EV-6 technology.

1.6 Design Principles for Modern Computers

There is a set of design principles, sometimes called the RISC design principles that architects of general-purpose CPUs do their best to follow:

- All Instructions Are Directly Executed by Hardware
    - eliminates a level of interpretation
- Maximize the Rate at Which Instructions are Issued

        - **MIPS** = millions of instructions per second
        - MIPS speed related to the number of instructions issued per second
        - Parallelism can play a role
- Instructions Should be Easy to Decode
    - a critical limit on the rate of issue of instructions
    - make instructions regular, fixed length, with a small number of fields.
    - the fewer different formats for instructions the better.
- Only Loads and Stores Should Reference Memory
    - **–** operands for most instructions should come from- and return to- registers.
    - access to memory can take a long time
    - thus, only LOAD and STORE instructions should reference memory.
- Provide Plenty of Registers

– accessing memory is relatively slow, many registers (at least 32) need to be provided, so that once a word is fetched, it can be kept in a register until it is no longer needed.

1.7 Parallelism

- Computer architects are constantly striving to improve performance of the machines they design.
- Making the chips run faster by increasing their clock speed is one way,
- However, most computer architects look to parallelism (doing two or more things at once) as a way to get even more performance for a given clock speed.
- Parallelism comes in two general forms:
    - Instruction-level parallelism, and
    - Processor-level parallelism.

1.8 Instruction-level parallelism:

(ILP) is a measure of how many of the operations in a computer program can be performed simultaneously. The potential overlap among instructions is called instruction level parallelism. ILP is to achieve not only instruction overlap, but the actual execution of more than one instruction at a time through dynamic scheduling and how to maximize the throughput of a processor. For typical RISC processors, instructions usually depend on each other too and as a result the amount of overlap is limited.

1.9 Processor level parallelism:

Multiprocessing is the use of two or more central processing units within a single computer system. It refers to the ability of a system to support more than processor and the ability to allocate task between them.

1.10 Summary

In this lesson we have discussed computer instructions and its types. We have also studied Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC) and the difference between these two. The concept of parallelism is also discussed in detail.

1.11 Self Check Exercise

1. What are instruction and its types? How many types of instructions set are there?
2. What is CISC and RISC? What is the difference between RISC and CISC processors?
3. What is parallelism and its types? What are the types of parallelism in parallel computer architecture?

1.12 Suggested Reading

1. M. Morris Mano, Computer System Architecture, Third Edition, Pearson Education (Singapore), 1993
2. William Stalling, "Computer Organization and Architecture", 6th edition, Pearson Education, 2000.
3. Hill, F.J. and G.R. Peterson, Digital Systems: Hardware Organization and Design, 3rd ed. New York: John Wiley, 1987.
4. Tanenbaum, Andrew S., Structured Computer Organization, 4th ed, Prentice-Hall of India, New Delhi, 2001

LESSON No. 2 **AUTHOR: Dr.VISHAL SINGH**

MEMORY

Objectives

In this lesson we will discuss memory and its hierarchy and storage architecture of the system.

2.1 Introduction

The system memory is the place where the computer holds current programs and data that are in use. There are various levels of computer memory, including ROM, RAM, cache, page and graphics, each with specific objectives for system operation.

Although memory is used in many different forms around modern PC systems, it can be divided into two essential types: RAM and ROM. ROM, or Read Only Memory, is relatively small, but essential to how a computer works. ROM is always found on motherboards, but is increasingly found on graphics cards and some other expansion cards and peripherals. Generally speaking, ROM does not change. It forms the basic instruction set for operating the hardware in the system, and the data within remains intact even when the computer is shut down. It is possible to update ROM, but it's only done rarely, and at need. If ROM is damaged, the computer system simply cannot function.

2.2 The Memory Hierarchy

Most modern programs can benefit greatly from a large amount of very fast memory. A physical reality, however, is that as a memory device gets larger, it tends to get slower. For example, cache memories are very fast but are also small and expensive. Main memory is inexpensive and large, but is slow (requiring wait states). The memory hierarchy is a mechanism of comparing the cost and performance of the various places we can store data and instructions.

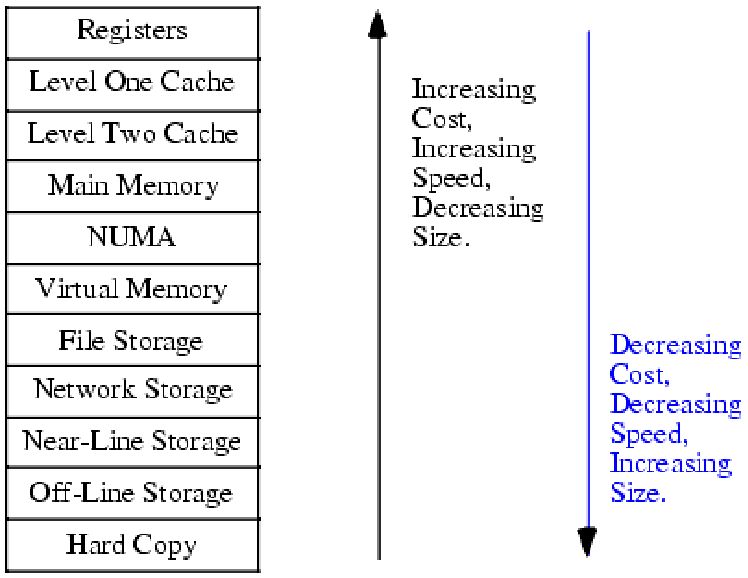


Figure-1 The Memory Hierarchy

At the top level of the memory hierarchy are the CPU's general purpose registers. The registers provide the fastest access to data possible on the 80x86 CPU. The register file is also the smallest memory object in the memory hierarchy (with just eight general purpose registers available). By virtue of the fact that it is virtually impossible to add more registers to the 80x86, registers are also the most expensive memory locations. Note that we can include FPU, MMX, SIMD, and other CPU registers in this class as well. These additional registers do not change the fact that there are a very limited number of registers and the cost per byte is quite high (figuring the cost of the CPU divided by the number of bytes of register available).

Working our way down, the Level One Cache system is the next highest performance subsystem in the memory hierarchy. On the 80x86 CPUs, the Level One Cache is provided on-chip by Intel and cannot be expanded. The size is usually quite small (typically between 4Kbytes and 32Kbytes), though much larger than the registers available on the CPU chip. Although the Level One Cache size is fixed on the CPU and you cannot expand it, the cost per byte of cache memory is much lower than that of the registers because the cache contains far more storage than is available in all the combined registers.

The Level Two Cache is present on some CPUs, on other CPUs it is the system designer's task to incorporate this cache (if it is present at all). For example, most Pentium II, III, and

IV CPUs have a level two cache as part of the CPU package, but many of Intel's Celeron chips do not. The Level Two Cache is generally much larger than the level one cache (e.g., 256 or 512KBytes versus 16 Kilobytes). On CPUs where Intel includes the Level Two Cache as part of the CPU package, the cache is not expandable. It is still lower cost than the Level One Cache because we amortize the cost of the CPU across all the bytes in the Level Two Cache. On systems where the Level Two Cache is external, many system designers let the end user select the cache size and upgrade the size. For economic reasons, external caches are actually more expensive than caches that are part of the CPU package, but the cost per bit at the transistor level is still equivalent to the in-package caches.

Below the Level Two Cache system in the memory hierarchy falls the main memory subsystem. This is the general-purpose, relatively low-cost memory found in most computer systems. Typically, this is DRAM or some similar inexpensive memory technology.

Below main memory is the NUMA category. NUMA, which stands for NonUniform Memory Access is a bit of a misnomer here. NUMA means that different types of memory have different access times. Therefore, the term NUMA is fairly descriptive of the entire memory hierarchy. In Figure a, however, we'll use the term NUMA to describe blocks of memory that are electronically similar to main memory but for one reason or another operate significantly slower than main memory. A good example is the memory on a video display card. Access to memory on video display cards is often much slower than access to main memory. Other peripheral devices that provide a block of shared memory between the CPU and the peripheral probably have similar access times as this video card example. Another example of NUMA includes certain slower memory technologies like Flash Memory that have significant slower access and transfers times than standard semiconductor RAM. We'll use the term NUMA in this lesson to describe these blocks of memory that look like main memory but run at slower speeds.

Most modern computer systems implement a Virtual Memory scheme that lets them simulate main memory using storage on a disk drive. While disks are significantly slower than main memory, the cost per bit is also significantly lower. Therefore, it is far less expensive (by three orders of magnitude) to keep some data on magnetic storage rather than in main memory. A Virtual Memory subsystem is responsible for transparently copying data between the disk and main memory as needed by a program.

File Storage also uses disk media to store program data. However, it is the program's responsibility to store and retrieve files data. In many instances, this is a bit slower than using Virtual Memory, hence the lower position in the memory hierarchy.

Below File Storage in the memory hierarchy comes Network Storage. At this level a program is keeping data on a different system that connects the program's system via a network. With Network Storage you can implement Virtual Memory, File Storage, and a system known as Distributed Shared Memory (where processes running on different computer systems share data in a common block of memory and communicate changes to that block across the network).

Virtual Memory, File Storage, and Network Storage are examples of so-called *on-line memory subsystems*. Memory access via this mechanism is slower than main memory access, but when a program requests data from one of these memory devices, the device is ready and able to respond to the request as quickly as is physically possible. This is not true for the remaining levels in the memory hierarchy.

The Near-Line and Off-Line Storage subsystems are not immediately ready to respond to a program's request for data. An Off-Line Storage system keeps its data in electronic form (usually magnetic or optical) but on media that is not (necessarily) connected to the computer system while the program that needs the data is running. Examples of Off-Line Storage include magnetic tapes, disk cartridges, optical disks, and floppy diskettes. When a program needs data from an off-line medium, the program must stop and wait for a someone or something to mount the appropriate media on the computer system. This delay can be quite long (perhaps the computer operator decided to take a coffee break?). Near-Line Storage uses the same media as Off-Line Storage, the difference is that the system holds the media in a special robotic jukebox device that can automatically mount the desired media when some program requests it. Tapes and removable media are among the most inexpensive electronic data storage formats available. Hence, these media are great for storing large amounts of data for long time periods.

Hard Copy storage is simply a print-out (in one form or another) of some data. If a program requests some data and that data is present only in hard copy form, someone will have to manually enter the data into the computer. Paper (or other hard copy media) is probably the least expensive form of memory, at least for certain data types.

2.3 Main Memory

The maximum size of the Main Memory (MM) that can be used in any computer is determined by its addressing scheme. For example, a 16-bit computer that generates 16-bit addresses is capable of addressing upto 2^{16} =64K memory locations. If a machine generates 32-bit addresses, it can access upto 2^{32} = 4G memory locations. This number represents the size of address space of the computer.

If the smallest addressable unit of information is a memory word, the machine is called word-addressable. If individual memory bytes are assigned distinct addresses, the computer is called byte-addressable. Most of the commercial machines are byte-addressable. For example in a byte-addressable 32-bit computer, each memory word contains 4 bytes. A possible word-address assignment would be:

Word Address	**Byte Address**			
0	0	1	2	3
4	4	5	6	7
8	8	9	10	11
.		…..		
.		…..		
.		…..		

With the above structure a READ or WRITE may involve an entire memory word or it may involve only a byte. In the case of byte read, other bytes can also be read but ignored by the CPU. However, during a write cycle, the control circuitry of the MM must ensure that only the specified byte is altered. In this case, the higher-order 30 bits can specify the word and the lower-order 2 bits can specify the byte within the word.

2.4 CPU-Main Memory Connection – A block schematic

From the system standpoint, the Main Memory (MM) unit can be viewed as a "block box". Data transfer between CPU and MM takes place through the use of two CPU registers, usually called MAR (Memory Address Register) and MDR (Memory Data Register). If MAR is K bits long and MDR is 'n' bits long, then the MM unit may contain upto 2^k addressable locations and each location will be 'n' bits wide, while the word length is equal to 'n' bits. During a "memory cycle", n bits of data may be transferred between the MM and CPU. This transfer takes place over the processor bus, which has k address lines (address bus), n data lines (data bus) and control lines like Read, Write, Memory Function completed (MFC), Bytes specifiers etc (control bus). For a read operation, the CPU loads the address into MAR, set READ to 1 and sets other control signals if required. The data from the MM is loaded into MDR and MFC is set to 1. For a write operation, MAR, MDR are suitably loaded by the CPU, write is set to 1 and other control signals are set suitably. The MM control circuitry loads the data into appropriate locations and sets MFC to 1. This organization is shown in the following block schematic Figure-2.

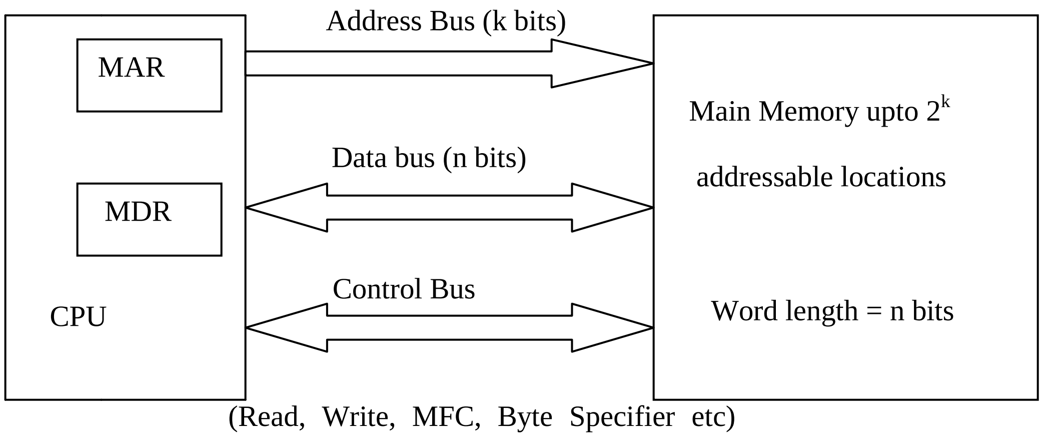

Address Bus (k bits)
MAR
Main Memory upto 2^k addressable locations
Data bus (n bits)
MDR
Control Bus
CPU
Word length = n bits
(Read, Write, MFC, Byte Specifier etc)

Figure-2

2.5 Some Basic Concepts

Memory Access Times

It is a useful measure of the speed of the memory unit. It is the time that elapses between the initiation of an operation and the completion of that operation (for example, the time between READ and MFC).

Memory Cycle Time

It is an important measure of the memory system. It is the minimum time delay required between the initiations of two successive memory operations (for example, the time between two successive READ operations). The cycle time is usually slightly longer than the access time.

Random Access Memory (RAM)

A memory unit is called a Random Access Memory if any location can be accessed for a READ or WRITE operation in some fixed amount of time that is independent of the location's address. Main memory units are of this type. This distinguishes them from serial or partly serial access storage devices such as magnetic tapes and disks which are used as the secondary storage device.

Cache Memory

The CPU of a computer can usually process instructions and data faster than they can be fetched from compatibly priced main memory unit. Thus the memory cycle time becomes the bottleneck in the system. One way to reduce the memory access time is to use cache memory. This is a small and fast memory that is inserted between the larger, slower main memory and the CPU. This holds the currently active segments of a program and its data. Because of the locality of address references, the CPU can, most of the time, find the relevant information in the cache memory itself (cache hit) and infrequently needs access to the main memory (cache miss) with suitable size of the cache memory, cache hit rates of over 90% are possible leading to a cost-effective increase in the performance of the system.

Memory Interleaving

This technique divides the memory system into a number of memory modules and arranges addressing so that successive words in the address space are placed in different modules. When requests for memory access involve consecutive addresses, the access will be to different modules. Since parallel access to these modules is possible, the average rate of fetching words from the Main Memory can be increased.

Virtual Memory

In a virtual memory System, the address generated by the CPU is referred to as a virtual or logical address. The corresponding physical address can be different and the required mapping is implemented by a special memory control unit, often called the memory management unit. The mapping function itself may be changed during program execution according to system requirements.

Because of the distinction made between the logical (virtual) address space and the physical address space; while the former can be as large as the addressing capability of the CPU, the actual physical memory can be much smaller. Only the active portion of the virtual address space is mapped onto the physical memory and the rest of the virtual address space is mapped onto the bulk storage device used. If the addressed information is in the Main Memory (MM), it is accessed and execution proceeds. Otherwise, an exception is generated, in response to which the memory management unit transfers a contiguous block of words containing the desired word from the bulk storage unit to the MM, displacing some block that is currently inactive. If the memory is managed in such a way that, such transfers are required relatively infrequency (ie the CPU will generally find the required information in the MM), the virtual memory system can provide a reasonably good performance and succeed in creating an illusion of a large memory with a small, in expensive MM.

Internal Organization of Semiconductor Memory Chips

Memory chips are usually organized in the form of an array of cells, in which each cell is capable of storing one bit of information. A row of cells constitutes a memory word, and the cells of a row are connected to a common line referred to as the word line, and this line is driven by the address decoder on the chip. The cells in each column are connected to a sense/write circuit by two lines known as bit lines. The sense/write circuits are connected

to the data input/output lines of the chip. During a READ operation, the Sense/Write circuits sense, or read, the information stored in the cells selected by a word line and transmit this information to the output lines. During a write operation, they receive input information and store it in the cells of the selected word.
The following figure shows such an organization of a memory chip consisting of 16 words of 8 bits each, which is usually referred to as a 16 x 8 organization.
The data input and the data output of each Sense/Write circuit are connected to a single bi-directional data line in order to reduce the number of pins required. One control line, the R/W (Read/Write) input is used a specify the required operation and another control line, the CS (Chip Select) input is used to select a given chip in a multichip memory system. This circuit requires 14 external connections, and allowing 2 pins for power supply and ground connections, can be manufactured in the form of a 16-pin chip. It can store 16 x 8 = 128 bits.
Another type of organization for 1k x 1 format is shown below (Figure-3)

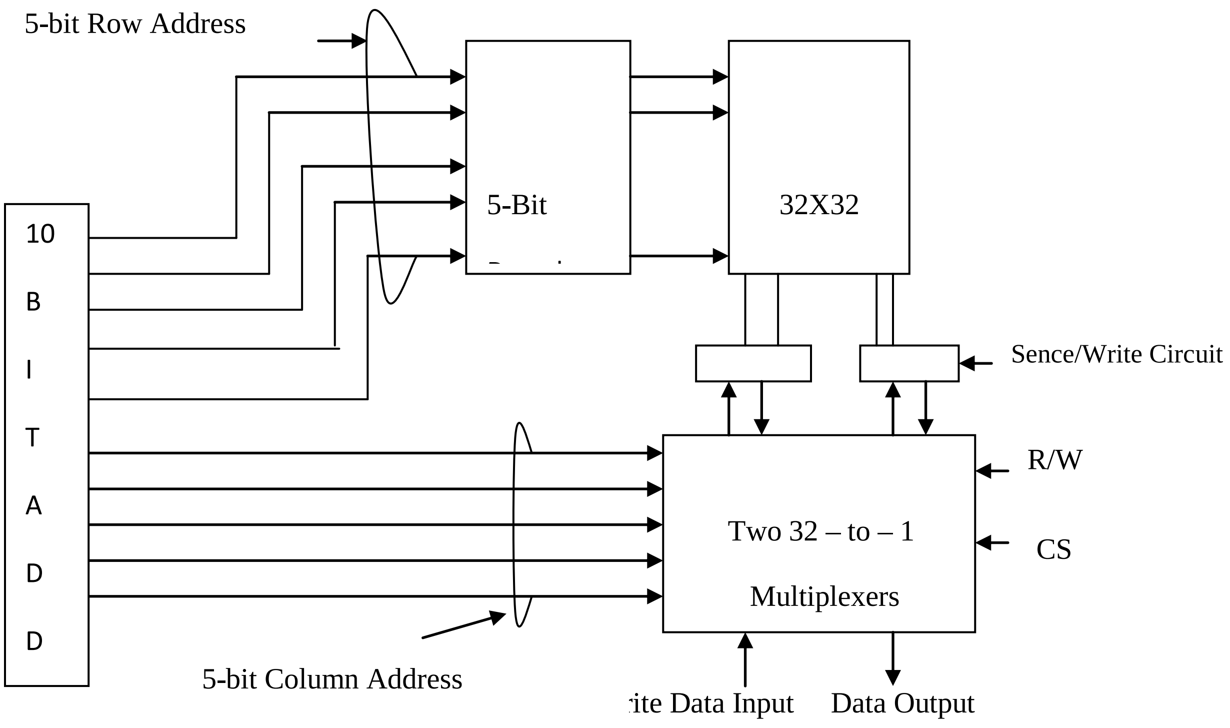


Figure-3

The 10-bit address is divided into two groups of 5 bits each to form the row and column addresses for the cell array. A row address selects a row of 32 cells, all of which are accessed in parallel. One of these, selected by the column address, is connected to the external data lines by the input and output multiplexers. This structure can store 1024 bits, can be implemented in a 16-pin chip.

2.6 A Typical Memory Cell

Semiconductor memories may be divided into bipolar and MOS types. They may be compared as follows:

Characteristic	**Bipolar**	**MOS**
Power Dissipation	More	Less
Bit Density	Less	More
Impedance	Lower	Higher
Speed	More	Less

2.6.1 Bipolar Memory Cell

A typical bipolar storage cell is shown below (Figure-4)

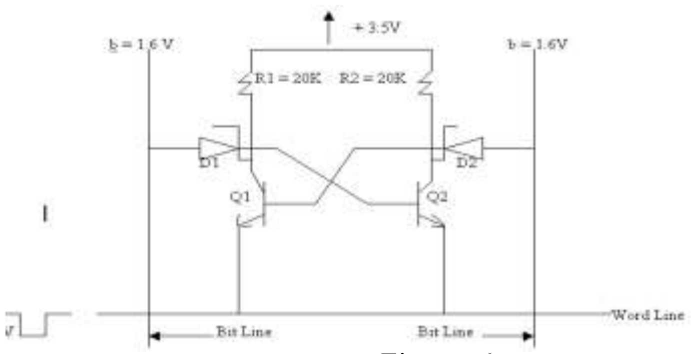


Figure-4

Two transistor inverters connected to implement a basic flip-flop. The cell is connected to one word line and two bits lines as shown. Normally, the bit lines are kept at about 1.6V, and the word line is kept at a slightly higher voltage of about 2.5V. Under these conditions, the two diodes D1 and D2 are reverse biased. Thus, because no current flows through the diodes, the cell is isolated from the bit lines.

Read Operation

Let us assume the Q1 on and Q2 off represents a 1 to read the contents of a given cell, the voltage on the corresponding word line is reduced from 2.5 V to approximately 0.3 V. This causes one of the diodes D1 or D2 to become forward-biased, depending on whether the transistor Q1 or Q2 is conducting. As a result, current flows from bit line b when the cell is in the 1 state and from bit line b when the cell is in the 0 state. The Sense/Write circuit at the end of each pair of bit lines monitors the current on lines b and b' and sets the output bit line accordingly.

Write Operation

While a given row of bits is selected, that is, while the voltage on the corresponding word line is 0.3V, the cells can be individually forced to either the 1 state by applying a positive voltage of about 3V to line b' or to the 0 state by driving line b. This function is performed by the Sense/Write circuit.

2.6.2 MOS Memory Cell

MOS technology is used extensively in Main Memory Units. As in the case of bipolar memories, many MOS cell configurations are possible. The simplest of these is a flip-flop circuit. Two transistors T1 and T2 are connected to implement a flip-flop. Active pull-up to VCC is provided through T3 and T4. Transistors T5 and T6 act as switches that can be opened or closed under control of the word line. For a read operation, when the cell is selected, T5 or T6 is closed and the corresponding flow of current through b or b' is sensed by the sense/write circuits to set the output bit line accordingly. For a write operation, the bit is selected and a positive voltage is applied on the appropriate bit line, to store a 0 or 1. This configuration is shown below (Figure-5)

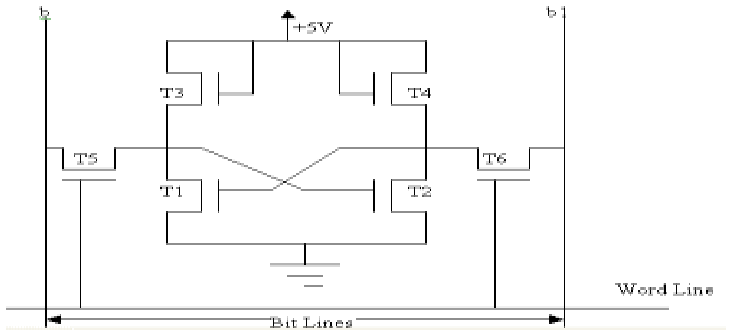


Figure-5

2.7 Auxiliary Memory

It also known as auxiliary memory or secondary storage, is the memory that supplements the main storage. This is a long-term, non-volatile memory. The term non-volatile means it stores and retains the programs and data even after the computer is switched off. Unlike RAM which looses the contents when the computer is turned off and ROM to which it is not possible to add anything new, auxiliary storage device allows a computer to record information semi-permanently. This is to ensure that this information can be read later by the same computer or by another computer. Auxiliary storage devices are also useful in transferring data or programs from one computer to another. They also function as backup devices which allows backup of the valuable information that you are working on. So, even if by some accident your computer crashes and the data in it is unrecoverable mode, you can restore it from your backups. The most common types of auxiliary storage devices are magnetic tapes, magnetic disks, floppy disks and hard disks.

There are two types of auxiliary storage devices. This classification is based on the type of data access-- sequential and random. Based on the type of access, they are called sequential access media and random media. In case of sequential access media, data stored in media can only be read in sequence. To get to a particular point on media, you have to go through all the preceding points.

Magnetic tapes are examples of sequential access media. In contrast, disks are random access media, also called direct access media, because a disk drive can access any point at random without passing through intervening points. Other examples of direct access media are magnetic disks, optical disks, zip disks etc.

2.8 Static Memories Vs Dynamic Memories

Bipolar as well as MOS memory cells using a flip-flop like structure to store information can maintain the information as long as current flow to the cell is maintained. Such memories are called static memories. In contracts, Dynamic memories require not only the maintaining of a power supply, but also a periodic "refresh" to maintain the information stored in them. Dynamic memories can have very high bit densities and very lower power consumption relative to static memories and are thus generally used to realize the main memory unit.

2.8.1 Dynamic Memories

The basic idea of dynamic memory is that information is stored in the form of a charge on the capacitor. An example of a dynamic memory cell is shown Figure-6.

When the transistor T is turned on and an appropriate voltage is applied to the bit line, information is stored in the cell, in the form of a known amount of charge stored on the capacitor. After the transistor is turned off, the capacitor begins to discharge. This is caused by the capacitor's own leakage resistance and the very

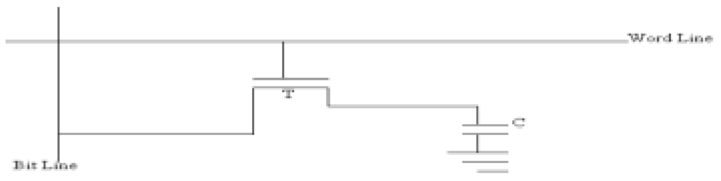
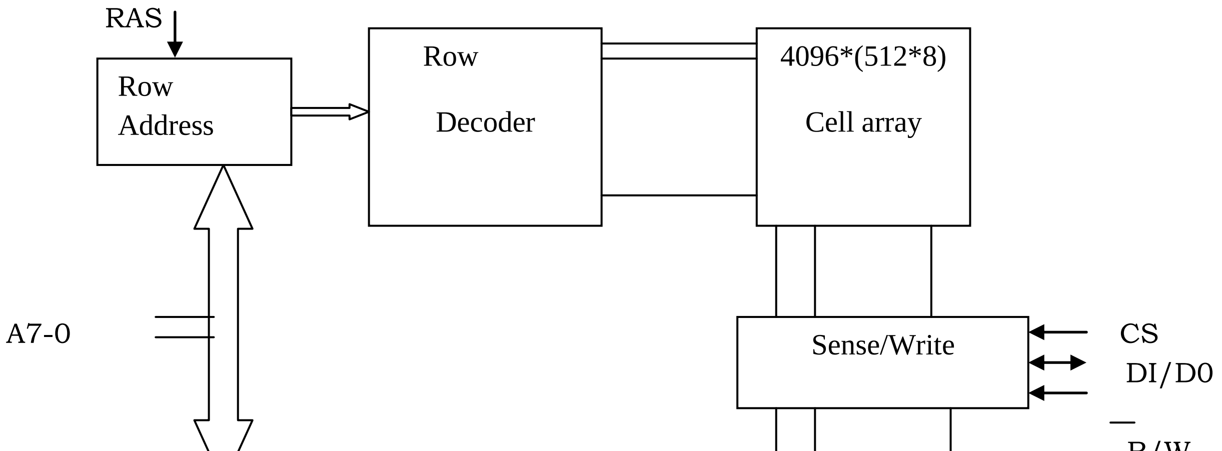


Figure-6

small amount of current that still flows through the transistor. Hence the data is read correctly only if is read before the charge on the capacitor drops below some threshold value. During a Read operation, the bit line is placed in a high-impedance state, the transistor is turned on and a sense circuit connected to the bit line is used to determine whether the charge on the capacitor is above or below the threshold value. During such a Read, the charge on the capacitor is restored to its original value and thus the cell is refreshed with every read operation.

Figure-7 Typical Organization of a Dynamic Memory Chip



A typical organization of a 64k x 1 dynamic memory chip is shown Figure-7.

The cells are organized in the form of a square array such that the high-and lower-order 8 bits of the 16-bit address constitute the row and column addresses of a cell, respectively. In order to reduce the number of pins needed for external connections, the row and column address are multiplexed on 8 pins. To access a cell, the row address is applied first. It is loaded into the row address latch in response to a single pulse on the Row Address Strobe (RAS) input. This selects a row of cells. Now, the column address is applied to the address pins and is loaded into the column address latch under the control of the

Column Address Strobe (CAS) input and this address selects the appropriate sense/write circuit. If the R/W signal indicates a Read operation, the output of the selected circuit is transferred to the data output. Do. For a write operation, the data on the DI line is used to overwrite the cell selected.
It is important to note that the application of a row address causes all the cells on the corresponding row to be read and refreshed during both Read and Write operations. To ensure that the contents of a dynamic memory are maintained, each row of cells must be addressed periodically, typically once every two milliseconds. A Refresh circuit performs this function. Some dynamic memory chips incorporate a refresh facility the chips themselves and hence they appear as static memories to the user! Such chips are often referred to as Pseudostatic.
Another feature available on many dynamic memory chips is that once the row address is loaded, successive locations can be accessed by loading only column addresses. Such block transfers can be carried out typically at a rate that is double that for transfers involving random addresses. Such a feature is useful when memory access follow a regular pattern, for example, in a graphics terminal.
Because of their high density and low cost, dynamic memories are widely used in the main memory units of computers. Commercially available chips range in size from 1k to 4M bits or more, and are available in various organizations like 64k x 1, 16k x 4, 1MB x 1 etc.

2.9 Design Consideration for Memory Systems

The choice of a RAM chip for a given application depends on several factors like speed, power dissipation, size of the chip, availability of block transfer feature etc.
Bipolar memories are generally used when very fast operation is the primary requirement. High power dissipation in bipolar circuits makes it difficult to realize high bit densities.
Dynamic MOS memory is the predominant technology used in the main memories of computer, because their high bit-density makes it possible to implement large memories economically.
Static MOS memory chips have higher densities and slightly longer access times compared to bipolar chips. They have lower densities than dynamic memories but are easier to use because they do not require refreshing.

2.9.1 Design using static Memory Chips

Consider the design of a memory system of 64k x 16 using 16k x 1 static memory chips. We can use a column of 4 chips to implement one bit position. Sixteen such sets provide the required 64k x 16 memories. Each chip has a control input called chip select, which should be set to 1 to enable the chip to participate in a read or write operation. When this signal is 0, the chip is electrically isolated from the rest of the system. The high-order 2 bits of the 16-bit address bus are decoded to obtain the four chip select control signals, and the remaining 14address bits are used to access specific bit locations inside each chip of the selected row. The $\overline{R}$ /W input of all chips are fed together to provide a common Read/$\overline{Write}$ control. This organization is shown in the following Figure-8.

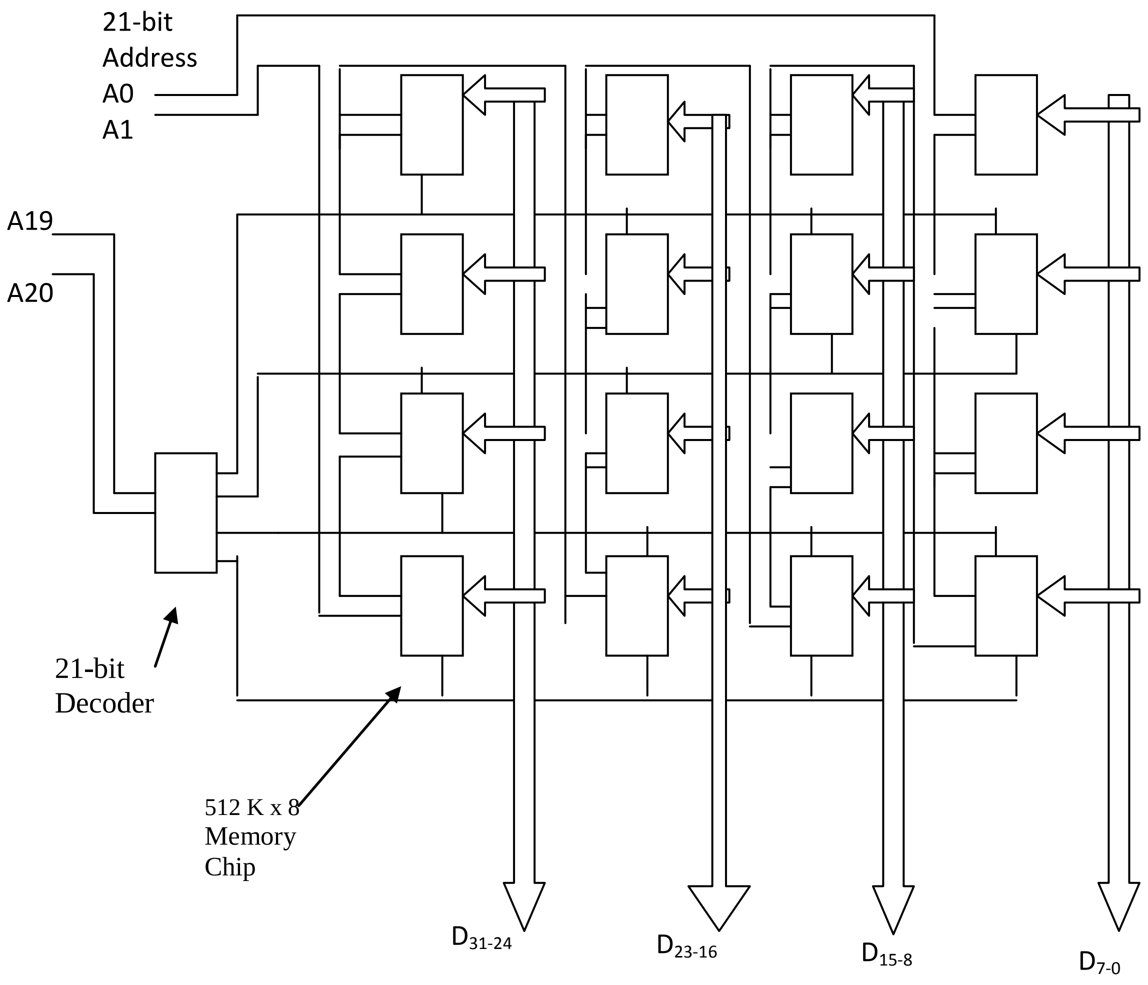


Figure-8

2.9.2 Design using Dynamic Memory Chips

The design of a memory system using dynamic memory chips is similar to the design using static memory chips, but with the following important differences in the control circuitry.

- The row and column parts of the address of each chip have to be multiplexed;
- A refresh circuit is needed; and
- The timing of various steps of a memory cycle must be carefully controlled.

A memory system of 256k x 16 designed using 64k x 1 DRAM chips, is shown in Figure-9

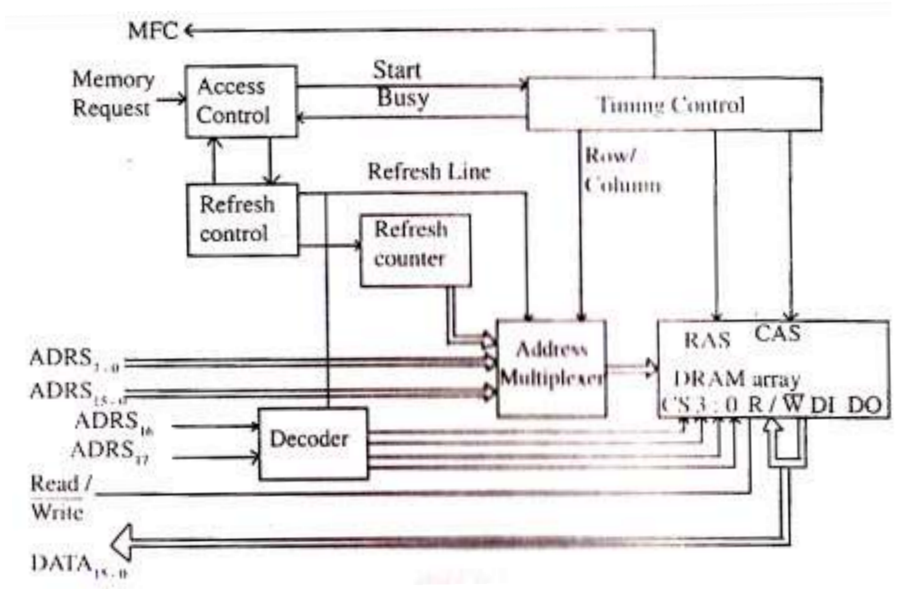


Figure-9

The memory unit is assumed to be connected to an asynchronous memory bus that has 18 address lines (ADRS $_{17-0}$), 16 data lines (DATA$_{15-0}$), two handshake signals (Memory request and MFC), and a Read/$\overline{Write}$ line to specify the operation (read to Write).

The memory chips are organized into 4 rows, with each row having 16 chips. Thus each column of the 16 columns implements one bit position. The higher order 12 bits of the address are decoded to get four chip select control signals which are used to select one of the four rows. The remaining 16 bits, after multiplexing, are used to access specific bit locations inside each chip of the selected row. The R/W inputs of all chips are tied together to provide a common Read/Write control. The DI and DO lines, together, provide $D_{15} - D_0$ i.e. the data bus DATA$_{15-0}$.

The operation of the control circuit can be described by considering a normal memory read cycle. The cycle begins when the CPU activates the address, the Read/$\overline{Write}$ and the Memory Request lines. When the memory request line becomes active, the Access control block sets the start signal to 1. The timing control block responds by setting busy lines to 1, in order to prevent the access control block from accepting new requests until the current cycle ends. The timing control block then loads the row and column addresses into the memory chips by activating the RAS and CAS lines. During this time, it uses the Row/$\overline{Column}$ line to select first the row address, ADRS$_{15-8}$, followed by the column address, ADRS$_{(7-0)}$.

2.10 Semi Conductor Rom Memories

Semiconductor read-only memory (ROM) units are well suited as the control store components in micro programmed processors and also as the parts of the main memory that contain fixed programs or data.

The following Figure-10 shows a possible configuration for a bipolar ROM cell.

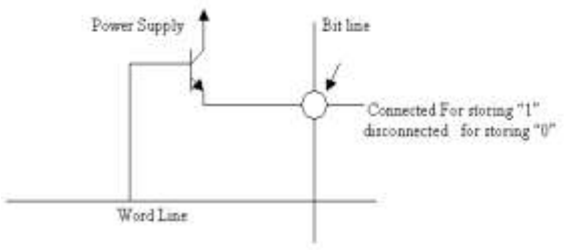


Figure-10

The word line is normally held at a low voltage. If a word is to be selected, the voltage of the corresponding word line is momentarily raised, which causes all transistors whose emitters are connected to their corresponding bit lines to be turned on. The current that flows from the voltage supply to the bit line can be detected by a sense circuit. The bit positions in which current is detected are read as 1s, and the remaining bits are read as O_s. Therefore, the contents of a given word are determined by the pattern of emitter to bit-line connections similar configurations are possible in MOS technology.

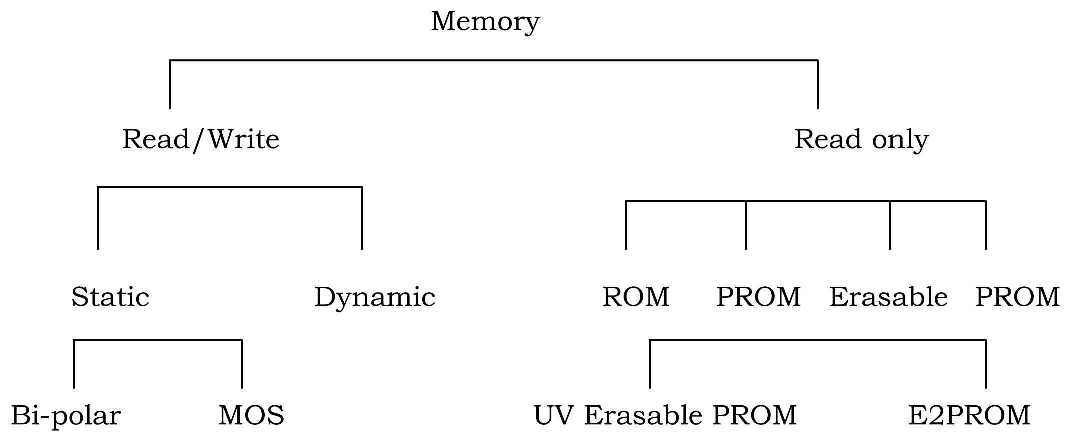
Data are written into a ROM at the time of manufacture programmable ROM (PROM) devices allow the data to be loaded by the user. Programmability is achieved by connecting a fuse between the emitter and the bit line. Thus, prior to programming, the memory contains all 1s. The user can insert O_s at the required locations by burning out the fuses at these locations using high-current pulses. This process is irreversible.

ROMs are attractive when high production volumes are involved. For smaller numbers, PROMs provide a faster and considerably less expensive approach.

Chips which allow the stored data to be erased and new data to be loaded. Such a chip is an erasable, programmable ROM, usually called an EPROM. It provides considerable flexibility during the development phase. An EPROM cell bears considerable resemblance to the dynamic memory cell. As in the case of dynamic memory, information is stored in the form of a charge on a capacitor. The main difference is that the capacitor in an EPROM cell is very well insulated. Its rate of discharge is so low that it retains the stored information for very long periods. To write information, allowing charge to be stored on the capacitor.

The contents of EPROM cells can be erased by increasing the discharge rate of the storage capacitor by several orders of magnitude. This can be accomplished by allowing ultraviolet light into the chip through a window provided for that purpose, or by the application of a high voltage similar to that used in a write operation. If ultraviolet light is used, all cells in the chip are erased at the same time. When electrical erasure is used, however, the process can be made selective. An electrically erasable EPROM, often referred to as EEPROM. However, the circuit must now include high voltage generation. Some E^2PROM chips incorporate the circuitry for generating these voltages o the chip itself. Depending on the requirements, suitable device can be selected.

Figure-11 Classification of memory devices



2.11 Summary

In this lesson we have discussed about the memory which is an essential component that stores data and instructions for the CPU to access quickly. We have studied in detail about the computer memory hierarchy which refers to the organization of memory into different levels based on speed, capacity, and cost. At the top of the hierarchy is the CPU cache, the fastest and smallest memory that holds frequently accessed data. Next is the RAM, providing quick access to data during active tasks. Lower in the hierarchy are secondary storage devices like SSDs and HDDs, offering larger but slower storage for long-term data retention. The hierarchy enables efficient data management, optimizing performance and cost-effectiveness in modern computing systems.

2.12 Self Check Exercise

1. A block set associative cache consists of a total of 64 blocks divided into 4 block sets. The MM contains 4096 blocks each containing 128 words.
   a) How many bits are there in MM address?
   b) How many bits are there in each of the TAG, SET & word fields
2. Describe Metal Oxide Semiconductor memory.
3. What is memory and storage in digital computer?

2.13 Suggested Readings

1. "Structured Computer Organization" by Tanenbaum.
2. R.P. Jain, “Modern Digital Electronics”, TMH.
3. M. Morris Mano, “Computer System Architecture”, PHI.

BCA SEM-3 | **PAPER BCAB2103T COMPUTER SYSTEM ORGANIZATION AND ARCHITECTURE**

LESSON No. 3 | **AUTHOR: Dr.VISHAL SINGH**

MEMORY-II

Objectives

In this lesson we will discuss the concept of cache memory. We will learn about virtual memory in detail along with the concept of paging and segmentation.

3.1 Cache Memory

Analysis of a large number of typical programs has shown that most of their execution time is spent on a few main row lines in which a number of instructions are executed repeatedly. These instructions may constitute a simple loop, nested loops or few procedures that repeatedly call each other. The main observation is that many instructions in a few localized are as of the program are repeatedly executed and that the remainder of the program is accessed relatively infrequently. This phenomenon is referred to as locality of reference.

If the active segments of a program can be placed in a fast memory, then the total execution time can be significantly reduced, such a memory is referred as a cache memory which is in served between the CPU and the main memory as shown in Figure-1.

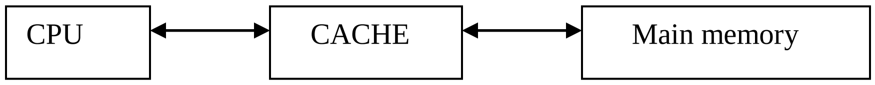


Figure-1: cache memory between main memory & CPU

3.2 Primary and Secondary addresses

A two level hierarchy and its addressing are illustrated in following figure. A system address is applied to the memory management unit (MMU) that handles the mapping function for the particular pair in the hierarchy. If the MMU finds the address in the Primary level, it provides Primary address, which selects the item from the Primary memory. This translation must be fast, because every time memory is accessed, the system address must be translated. The translation may fail to produce a Primary address because the requested items is not found, so that information can be retrieved from the secondary level and transferred to the Primary level.

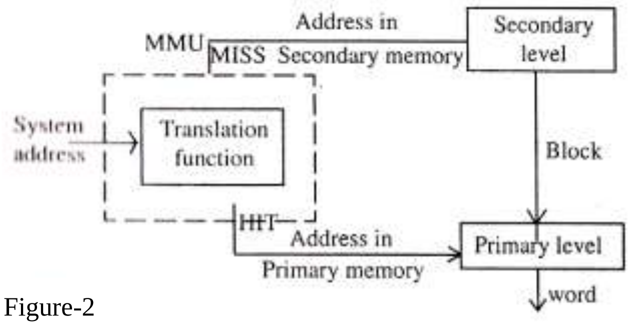


Figure-2

Hits and Misses

Successful translation of reference into Primary address is called a hit, and failure is a miss. The hit ratio is (1-miss ratio). If t_p is the Primary memory access time and t_s is the secondary access time, the average access time for the two level hierarchy is

$$t_a = h\,t_p + (1-h)t_s$$

The Cache:- The Mapping Function

Figure-3 Shows a schematic view of the cache mapping function. The mapping function is responsible for all cache operations. It is implemented in hardware, because of the required high speed operation. The mapping function determines the following.

- Placement strategies – Where to place an incoming block in the cache
- Replacement strategies – Which block to replace when a cache miss occurs
- How to handle Read and Writes up as cache hit and misses

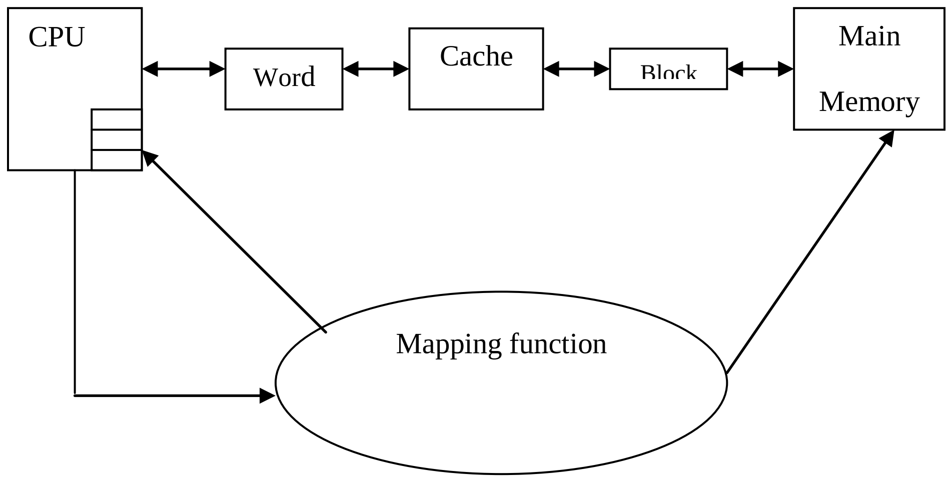


Figure-3

Three different types of mapping functions are in common use

1. Associative mapping cache
2. Direct mapping cache
3. Block-set-associative mapping cache

1. Associative mapped caches:- In this any block from main memory can be placed any where in the cache. After being placed in the cache, a given block is identified uniquely by its main memory block number, referred to as the tag, which is stored inside a separate tag memory in the cache.

Regardless of the kind of cache, a given block in the cache may or may not contain valid information. For example, when the system has just been powered up add before the cache has had any blocks loaded into it, all the information there is invalid. The cache maintains a valid bit for each block to keep track of whether the information in the corresponding block is valid.

Figure-4 shows the various memory structures in an associative cache, The cache itself contain 256, 8byte blocks, a 256 x 13 bit tag memory for holding the tags of the blocks currently stored in the cache, and a 256 x 1 bit memory for storing the valid bits, Main memory contains 8192, 8 byte blocks. The figure indicates that main memory address references are partition into two fields, a 3 bit word field describing the location of the desired word in the cache line, and a 13 bit tag field describing the main memory block number desired. The 3 bit word field becomes essentially a “cache address” specifying where to find the word if indeed it is in the cache.

The remaining 13 bits must be compared against every 13 bit tag in the tag memory to see if the desired word is present.

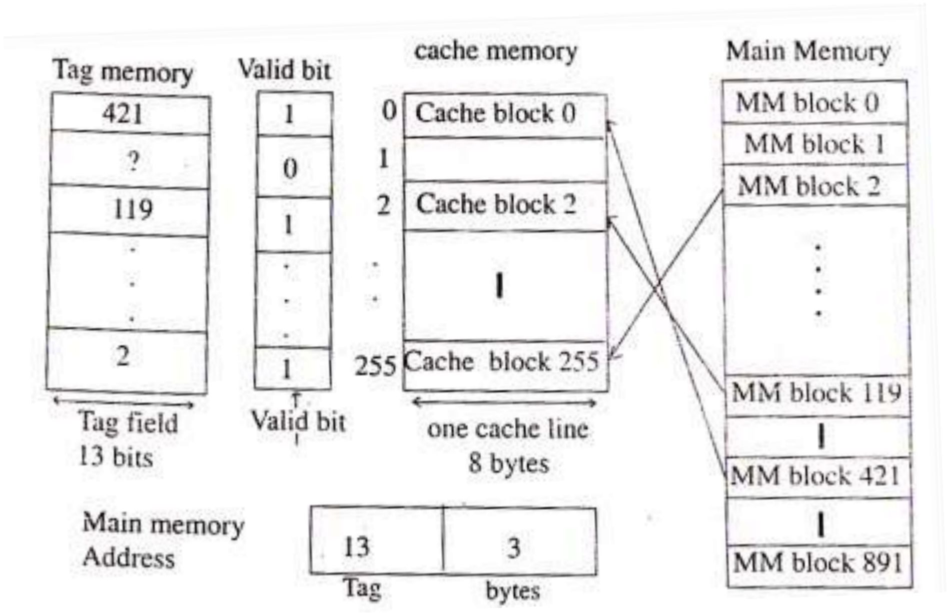


Figure-4

In the Figure-4 above, main memory block 2 has been stored in the 256 cache block and so the 256th tag entry is 2 mm block 119 has been stored in the second cache block corresponding entry in tag memory is 119 mm block 421 has been stored in cache block 0 and tag memory location 0 has been set to 421. Three valid bits have also been set, indicating valid information in these locations.

The associative cache makes the most flexible and complete use of its capacity, storing blocks wherever it needs to, but there is a penalty to be paid for this flexibility the tag memory must be searched in for each memory reference.

Figure-5 shows the mechanism of operation of the associative cache. The process begins with the main memory address being placed in the argument register of the (associative) tag memory (1) if there is a match (hit), (2) and if the ratio bit for the block is set (3), then the block is gated out of the cache (4), and the 3 bit offset field is used to select the byte corresponding to the block offset field of the main memory address (5) That byte is forwarded to the CPU, (6)

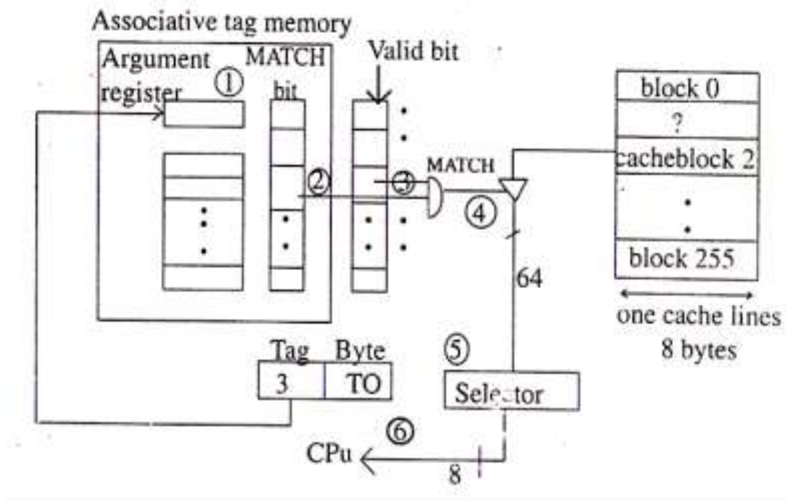


Figure-5

2. Direct mapped caches:- In this a given main memory block can be placed in one and only one place in the cache. Fig. Shows an example of a direct – mapped cache. For simplicity, the example again uses a 256 block x 8 byte cache and a 16 bit main memory address. The main memory in the Figure-6 has 256 rows x 32 columns, still fielding 256 x 32 = 8192 = 2^{13} total blocks as before. Notice that the main memory address is partitioned into 3 fields. The word field still specifies the word in the block. The group field specifies which of the 256 cache locations the block will be in, if it is indeed in the cache. The tag field specifies which of the 32 blocks from main memory is actually present in the cache. Now the cache address is composed of the group field, which specifies the address of the block location in the cache and the word field, which specifies the address of the word in the block. There is also valid bit specifying whether the information in the selected block in valid.

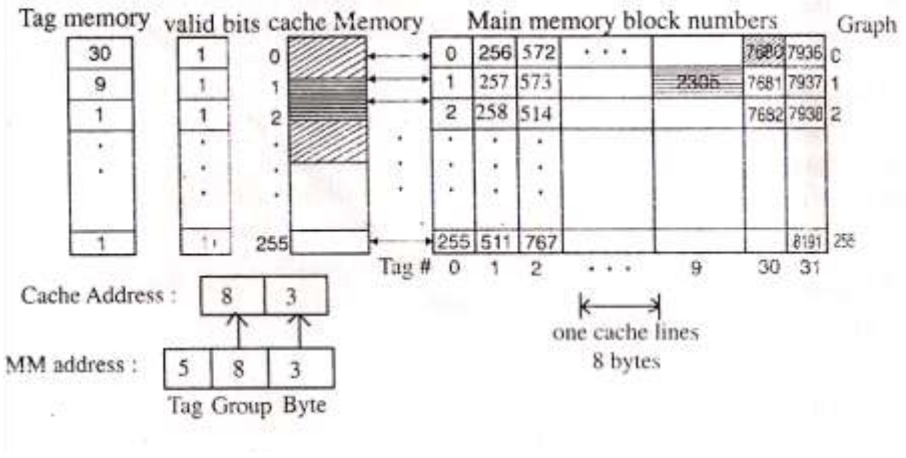


Figure-6

The figure shows block 7680, from group 0 of MM placed in block location 0 of the cache and the corresponding tag set to 30. Similarly MM block 259 is in MM group 2, column 1, it is placed in block location 2 of the cache and the corresponding tag memory entry is 1.

The tasks required of the direct – mapped cache in servicing a memory request are shown in figure.

The figure shows the group field of the memory address being decoded 1 and used to select the tag of the one cache block location in which the block must be stored if it is the cache. If the valid bit for that block location is gated (2), then that tag is gated out, (3) and compared with the tag of the incoming memory address (4), A cache hit gates the cache block out (5) and the word field selects the specified word from the block (6), only one tag needs to be compared, resulting in considerably less hardware than in the associative memory case.

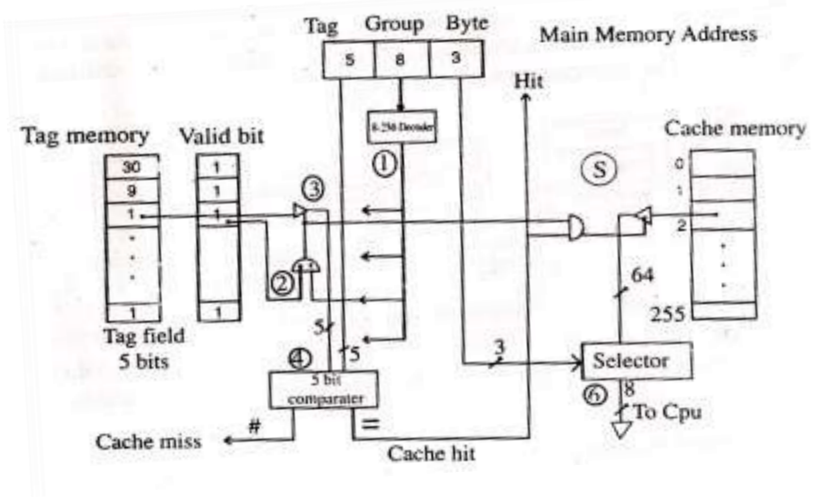


Figure-7

The direct mapped cache has the advantage of simplicity, but the obvious disadvantage that only a single block from a given group can be present in the cache at any given time.

3. Block-set-Associative cache:- Block-set-Associative caches share properties of both of the previous mapping functions. It is similar to the direct-mapped cache, but now more than one block from a given group in main memory can occupy the same group in the cache. Assume the same main memory and block structure as before, but with the cache being twice as large, so that a set of two main memory blocks from the same group can occupy a given cache group.

Figure-8 shows a 2 way set associative cache that is similar to the direct mapped cache in the previous example, but with twice as many blocks in the cache, arranged so that a set of any two blocks from each main memory group can be stored in the cache. MM is still partitioned into an 8 bit set field and a 5 bit tag field, but now there are two possible places in which a given block can reside and both must be searched associatively.

The cache group address is the same as that of the direct matched cache, an 8 bit block location and a 3 bit word location. Figure shows that the cache entries' corresponding to the second group contains blocks 513 and 2304. the group field now called the set field, is again decoded and directs the search to the correct group and new only the tags in the selected group must be searched. So instead of 256 compares, the cache only needs to do 2.

For simplicity, the valid bits are not shown, but they must be present. The cache hard ware would be similar so that shown in figure but there would be two simultaneous comparisons of the two blocks in the set.

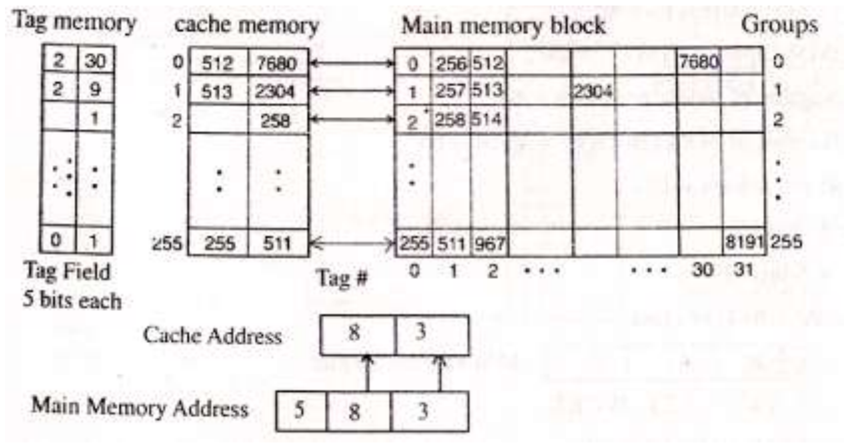


Figure-8

If the presence bit indicates a hit, then the page field of the page table entry will contain the physical page number. If the presence bit is a miss, which is page fault, then the page field of the page table entry which contains an address is secondary memory where the page is stored. This miss condition also generates an interrupt. The interrupt service routine will initiate the page fetch from secondary memory and with also suspended the requesting process until the page has been bought into main memory. If the CPU operation is a write hit, then the dirty bit is set. If the CPU operation is a write miss, then the MMU with begin a write allocate process.

3.3 Virtual Memory

Virtual memory is the technique of using secondary storage such as disks to enter the apparent size of accessible memory beyond its actual physical size. Virtual memory is implemented by employing a memory-management unit (MMU) to translate every logical address reference into a physical address reference as shown in Figure-9. The MMU is imposed between the CPU and the physical memory where it performs these translations under the control of the operating system. Each memory reference is used by the CPU is translated from the logical address space to the physical address space. Mapping tables guide the translation, again under the control of the operating system.

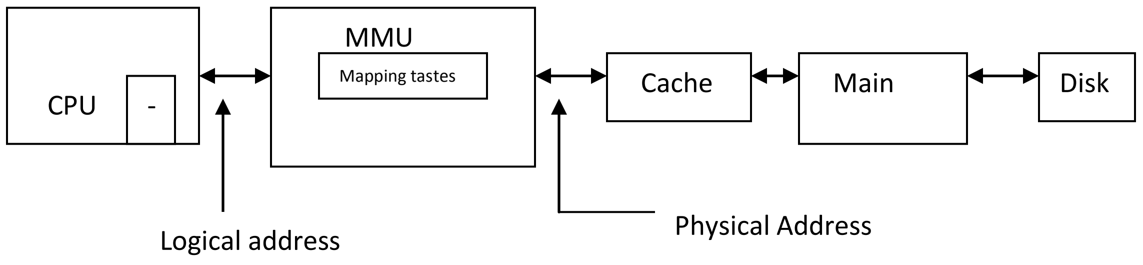


Figure-9

Virtual memory usually demands paging, which means that a Page is moved from disk into main memory only when the processor accesses a word on that page. Virtual memory pages always have a place on the disk once they are created, but are copied to main memory only on a miss or page fault.

Virtual memory is a catch-all phrase for the abstraction of physical memory via a virtual address space. In all cases a virtual memory manager is responsible for maintaining this virtual address translation; generally speaking the definition also includes the responsibility of operating a large virtual address range within a smaller available physical memory. This process involves the transfer of blocks of memory from a secondary source (usually a slower hard-disk) to the primary memory whenever necessary for program execution. It is the virtual memory manager's responsibility to provide a level of abstraction for programs, allowing them to operate seamlessly without any "knowledge" of the underlying system. If implemented properly, software can be written for these systems using a relatively large virtual address range, while running on a small amount of physical memory with little reduction in speed.

3.3.1 Advantages of Virtual memory

1. **Simplified addressing:-** Each program unit can be compiled into its own memory space, beginning at address O and extending far beyond the limits of physical memory. Programs and data structures do not require address relocation at load time, nor must they be broken into fragments merely to accommodate memory limitations.
2. **Cost effective use of memory:**- Less expensive disk storage can replace more expensive RAM memory, since the entire program does not need to occupy physical memory at one time.
3. **Access control:**- Since each memory reference must be translated, it can be simultaneously checked for read, write and execute privileges. This allows hardware level control of access to system resources and also prevents and also prevents buggy programs or intruders from causing damage to the resources of other users or the system.

3.3.2 Memory management by segmentation

Segmentation allows memory to be divided into segments of varying sizes depending upon requirements. Figure-10 shows a main memory containing five segments identified by segment numbers. Each segment beings at a virtual address 0, regardless of where it is located in physical memory.

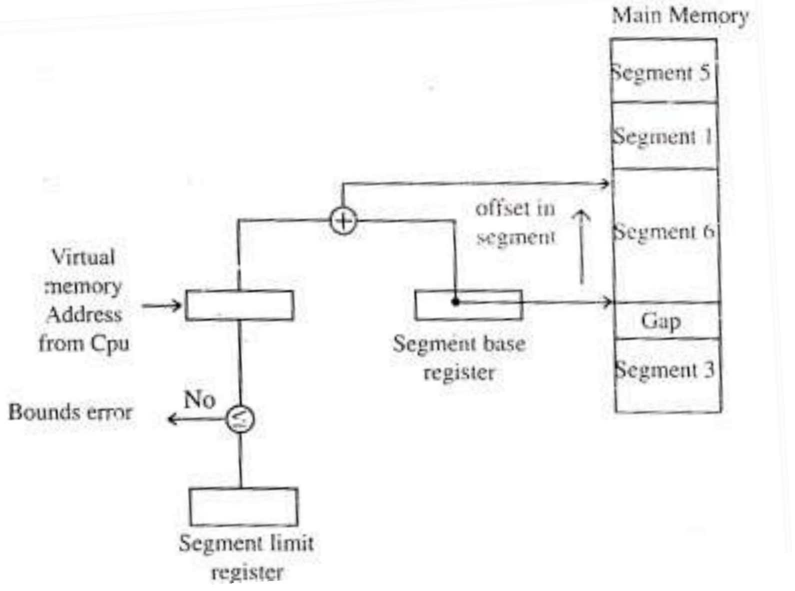


Figure-10

Each virtual address arriving from the CPU is added to the contents of the segment base register in the MMU to form the physical address. The virtual address may also optionally be compared to a segment limit register to trap reference beyond a specified limit.

3.3.3 Memory management by paging

Figure-11 shows a simplified mechanism for virtual address translation in a paged MMU. The process begins in a manner similar to the segmentation process. The virtual address composed of a high order page number and a low order word number is applied to MMU. The virtual page number is limit checked to be certain that the page is within the page table, and if it is, it is added to the page table base to yield the page table entry. The page table entry contains several control fields in addition to the page field. The control fields may include access control bits, a presence bit, a dirty bit and one or more use bits, typically the access control field will include bits specifying read, write and perhaps execute permission. The presence bit indicates whether the page is currently in main memory. The use bit is set upon a read or writes to the specified page, as an indication to the replaced algorithm in case a page must be replaced.

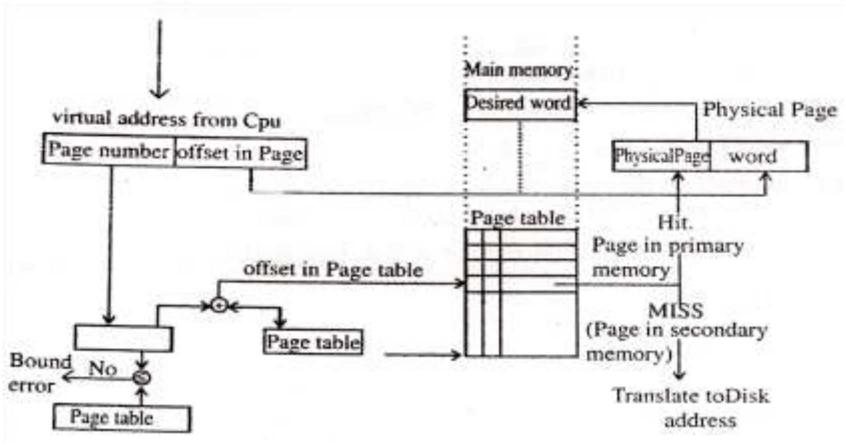


Figure-11

If the presence bit indicates a hit, then the page field of the page table entry will contain the physical page number. If the presence bit is a miss, which is page fault, then the page field of the page table entry which contains an address is secondary memory where the page is stored. This miss condition also generates an interrupt. The interrupt service routine will initiate the page fetch from secondary memory and with also suspended the requesting process until the page has been bought into main memory. If the CPU operation is a write hit, then the dirty bit is set. If the CPU operation is a write miss, then the MMU with begin a write allocate process.

3.4 Virtual Memory

Virtual memory is a system by which the machine or operating system fools processes running on the machine into thinking that they have a lot more memory to work with than the capacity of RAM would indicate. It does this by storing the most recently used items in RAM, and storing the lesser used items in the slower disk memory, and interchanging data between the two whenever a disk access is made. In this way, memory appears to programs to be a full 32 bit address space, when it fact memory space is probably only a mere fraction of that.

3.4.1 The Virtual Address

How can this be done? We are able to address byte 0xFFFFFFFF. But there is no byte 0xFFFFFFFF in our main memory. So where is byte 0xFFFFFFFF? Somewhere other than byte 0xFFFFFFFF in RAM, since such an index that large does not exist in RAM. Obviously, the relation between an address and its actual location in memory is not as simple as previously thought. Your friend told you the memory you're looking for at address 0x00000010 is stored in the seventeenth byte in main memory. Your "friend" lied to you.

The way it works: The programs request memory at a certain address. Now, this address isn't the REAL address for this data; it is a virtual address. The memory system then checks to see where the data for this virtual address really is, based on a reference to something called the page table. After looking up this address in the page table to tell where it is, it does one of two things. If the memory is already in main memory, it returns the real address in main memory, at which point the write to or reading from memory is done. If the data we're looking for is not yet in RAM, then the memory system retrieves the data from the disk (where data that cannot fit in RAM is stored), puts it into RAM, and THEN returns the address at which it can be found.

3.4.2 Pages, and the Virtual Page Number

To take advantage of special locality as well as hardware issues with disk mass storage, it is important to know that just as lines of data in cache would often be larger than one byte or four bytes (in some systems cache lines are thirty-two or sixty-four bytes), "lines" of data in a virtual memory system are often 2048 bytes, 4096 bytes, or 8192 bytes. Also, these "lines" are called pages. The page is the quantum unit of transfer between disk and main memory in a virtual memory system, just as a line was the quantum unit of transfer between cache and RAM. If you ask for a word of memory not in RAM, not just the word, but a whole page's worth of data from the disk is loaded into RAM.

Pages in a virtual address space are contiguous. In a 4096 byte page virtual memory system, the first page in virtual memory would be addresses 0x00000000 to 0x00000FFF. The last page in virtual memory would be 0xFFFFF000 to 0xFFFFFFFF. In this example, the first twenty bits uniquely identify each page in the virtual address space. Therefore, since each page in the virtual address space may be uniquely identified by a certain number of the leading bits of the virtual address, we call that number the "virtual page number."

If we have an N bit virtual address space (in our example it is thirty two), and we have 2^M page size (in our example, M = 12), then the leading N-M bits (in our example the leading 32-12 = 20 bits) comprise the virtual page number that uniquely identifies each page in the virtual address space.

The lower M bits tell us which byte of this page is the byte we're looking for, and is called the "page offset." The page offset is more or less the exact analogy of the cache's "byte select field." It tells us which byte of our 2^M byte page we're trying to access.

3.4.3 The Virtual Page Number through Examples

For some examples of this, I provide some virtual addresses with the page size of the system. I then tell what the page is, and what the page offset is.

Virtual address 0x000A502F, with page size of 4K bytes.

A page size of 4K = 4096 = 2^{12}, so the last 12 bits are used as the page offset to tell just what byte of the page we want. Therefore the first 20 bits uniquely identify this page, and comprise the virtual page number.

0000 0000 0000 1010 0101	0000 0010 1111

So the virtual page number is 165, and 47 is the page offset. So we're looking for the 48th byte (since 0 page offset indicates the 1st byte of the page) of the 166th page (similarly, since a page number of 0 indicates the 1st page).

Virtual address 0x000A502F, with page size of 2K bytes

The same address, but different page size. A page size of 2K = 2048 = 2^{11}, so the last 11 bits are used as the page offset to tell just what byte of the page we want. Therefore the first 21 bits uniquely identify this page, and comprise the virtual page number.

0000 0000 0000 1010 0101 0	000 0010 1111

So the virtual page number is 330, and 47 is the page offset. So we're looking for the 48th byte of the 331th page.

Virtual address 0x009B18A0, with page size of 4K bytes.

A page size of 4K = 4096 = 2^{12}, so the last 12 bits are used as the page offset to tell just what byte of the page we want. Therefore the first 20 bits uniquely identify this page, and comprise the virtual page number.

0000 0000 1001 1011 0001	1000 1010 0000

So the virtual page number is 2481, and 2208 is the page offset. So we're looking for the 2208th byte of the 2481th page.

Virtual address 0x009B18A0, with page size of 2K bytes.

The same address, but different page size. A page size of 2K = 2048 = 2^{11}, so the last 11 bits are used as the page offset to tell just what byte of the page we want. Therefore the first 21 bits uniquely identify this page, and comprise the virtual page number.

0000 0000 1001 1011 0001 1	000 1010 0000

So the virtual page number is 4963, and 160 is the page offset. So we're looking for the 4964th byte of the 161th page.

3.5 Three Main Parts of Virtual Memory

As implied above, virtual memory consists of three main parts. First, there is the main memory (RAM), which holds recently used chunks (pages) of memory. Second, there is the secondary memory (disk), which stores the chunks (pages) not currently being used. Thirdly, there is the page table, which tells us just where on the disk or in RAM the particular chunk of data we're looking for is.

Main Memory (RAM)

The main memory system is where the more recently used pages are stored. Each page is stored into subdivisions of memory called "frames." A frame size is the same as our page size. If we have a 4096 byte page size, then each frame will hold 4096 bytes to accomodate the pages. The first frame will take up actual addresses 0x00000000 to 0x00000FFF. The second frame will take up address 0x00001000 to 0x00001FFF. Etc. If physical memory is 256 megabytes as it is on my computer, the last page would be from 0x0FFFF000 to 0x0FFFFFFF. Notice that the range of addresses covers 4096 bytes. (It actually be a little more complicated than that, but for our purposes this will suffice.)

Also, though I used similar numbers in my example of different pages in the virtual memory space, it is important to remember that in these frames, the frame that consists of physical addresses 0x00001000 to 0x00001FFF probably does not consist of the same data contained in the virtual addresses 0x00001000 to 0x00001FFF.

A concept that will become important later is the "frame number." In the example above, the frame number of the first frame will be 0. The second frame will have frame number 1. Similarly, the frame number of the physical address 0x0005ACC0 in a system with a 4K page size will be frame number 0x5A. However, the same address in a system with a 2K page size will have frame number 0xB5 (comprised of the leading 21 bits rather than the first 20 bits).

Think of main memory as short term fast storage of the pages we are currently working on, like a cache.

Secondary Memory (Disk)

The disk memory is a repository for pages not currently in use. When a page needs to be brought to memory, the appropriate page is found and transferred to main memory. Whenever a page that has been modified during its time in main memory, it is written to disk. Think of the pages on the disk as being in long term storage.

The Page Table

The page table is what keeps track of where pages are, and what their properties are. The system updates the page table as changes in the state of the system warrant. It makes sense that there are as many entries in the page table as there are pages in our virtual address space. Therefore, in a virtual address space addressed by 32 bits, at 4K per page that's 2^{20} pages, and hence 2^{20} entries in the page table. The first entry in the page table contains information about the first page. The second entry contains information about the second page. Etc. If we have a virtual address with virtual page number 56 (and hence is the 57th page), we can find information about that page in the 57th entry in the page table.

Entry of the Page Table

So what is in an entry of the page table? Since the virtual address and the physical address need not have much relation to each other, it is important to know just where we can find memory. Also, we may want to be able to tell certain things about memory: Is this page in main memory or on the disk? Has this memory been written to, in which case we'll have to write it back to disk before discarding it? What are we allowed to do to this data?

Valid Bit

> The valid bit tells us if the memory is currently in main memory or if it must be retrieved from secondary memory. When a page is taken from disk and put into main memory, the valid bit is set to 1. When a page is overwritten in main memory once the system feels we no longer need it, the valid bit is set to 0.

Dirty Bit

> The dirty bit tells us if memory has been written do during its time in main memory. If it has been, then once we discard the memory we must write it back to disk. If it hasn't been modified, then we're saved a disk write when we get rid of the page in this frame, which speeds the system up. When the page is first loaded into memory, its dirty bit is set to 0. If a write instruction to that page occurs during its time in memory, then the dirty bit is set to 1.

Access Control

> A rather primitive form of memory protection, access control indicates what we may do to this frame. Do we have access to read, read/write, or execute from this frame? In this way programs are prevented from doing damage to other data that may belong to other processes.

Frame Number

If the data is in main memory, since any page can be loaded into any frame, we have to have some data that tells us just which frame the data is at. Hence, the frame number.

Disk Location

If we need to write back data currently in main memory, or if we need to read a page that is not in main memory, then we need to know where on the disk that data can be found. This is a close analogy to the frame number.

3.6 Finding the Real Address

Given a virtual address, we can find its virtual page number by selecting a certain number of leading bits (the number selected depends on both the sizes of our virtual address space and the size of our page). From that, the memory system can access the corresponding entry in the page table to find out just where this data is, retrieve it from disk if necessary, and then, using the frame number of the frame to which it was loaded, derive the corresponding physical address of this virtual address.

When a memory reference is made, you're given a virtual address. With a page size of 4K, or 2^{12}, and our virtual address space of 2^{32}, we have 2^{20} possible pages. The first 20 bits will therefore be the "virtual page number" for this address.

As an example, suppose we're working with a 4K page size. Take the 32 bit virtual address 0x00072A4C, which in 32-bit binary is:

0000 0000 0000 0111 0010 1010 0100 1100

The leading 20 bits of that is our page number, since the rightmost 12 bits must be used to determine where in the page the byte we're trying to access is. The page number is therefore $1110010_2 = 114_{10}$. The rest of the bits, $101001001100_2 = 2636_{10}$, is our page offset.

For the purposes of simplicity, let's say the page table has as many entries as there are possible pages... that is, $2^{20} = 1048576$ entries. There is a direct correspondence between the entires in the page table and the pages in DISK, e.g., the 23rd page corresponds to the 23rd entry in the page table.

If it IS valid, then another field in this entry tells us what frame in MEMORY it is at (the frame number). We now have the frame. Using the frame number and the page offset, we can now compute the physical address.

If the page we seek is not already loaded into main memory (that is, the valid bit is not set so that we encounter a page fault), then the page must be loaded into a frame in main memory before any reading or writing by the program may occur. We must find a frame to load into.

Figure-12 Conversion from virtual address to physical address

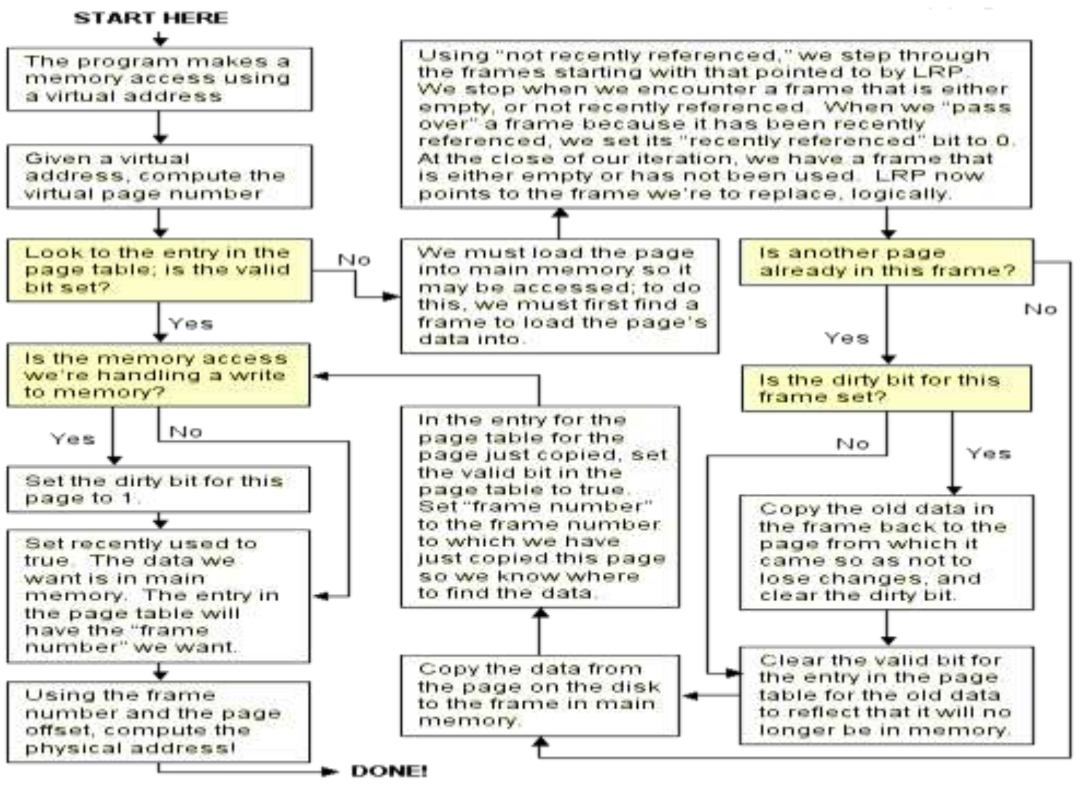

START HERE
The program makes a memory access using a virtual address
Given a virtual address, compute the virtual page number
Look to the entry in the page table; is the valid bit set?
No
Yes
Is the memory access we're handling a write to memory?
Yes
No
Set the dirty bit for this page to 1.
Set recently used to true. The data we want is in main memory. The entry in the page table will have the "frame number" we want.
Using the frame number and the page offset, compute the physical address!
DONE!
Using "not recently referenced," we step through the frames starting with that pointed to by LRP. We stop when we encounter a frame that is either empty, or not recently referenced. When we "pass over" a frame because it has been recently referenced, we set its "recently referenced" bit to 0. At the close of our iteration, we have a frame that is either empty or has not been used. LRP now points to the frame we're to replace, logically.
We must load the page into main memory so it may be accessed; to do this, we must first find a frame to load the page's data into.
Is another page already in this frame?
No
Yes
Is the dirty bit for this frame set?
No
Yes
In the entry for the page table for the page just copied, set the valid bit in the page table to true. Set "frame number" to the frame number to which we have just copied this page so we know where to find the data.
Copy the old data in the frame back to the page from which it came so as not to lose changes, and clear the dirty bit.
Copy the data from the page on the disk to the frame in main memory.
Clear the valid bit for the entry in the page table for the old data to reflect that it will no longer be in memory.

Figure-12

3.7 Selecting a Frame to Replace on Page Fault (NRU and LRU)

Through what mechanism do we select a page that we want to replace? I will describe the "not recently used" algorithm in this section. The flow chart in Figure-12 has a brief summary of it in its upper righthand box.

Suppose that we have an additional data structure that defines a bit for each frame in main memory, and suppose this bit is called "recently used" (RU). Also suppose we have a pointer to the frame that was last replaced called the "last replaced pointer" (LRP). What not recently used does is it looks at the LRP, increments LRP by one (looping around if we're past the last frame), and checks the frame. Is the RU bit set? If it is not (or if the frame is still empty), we have a frame that we can load data into. If it is set, we skip over this frame, but we set its recently used bit false before repeating this process and incrementing LRP to the next frame.

RU is set to true whenever a memory access (read or write) to that address occurs, to tell the machine that yes, someone is using the page in this frame, so don't get rid of it yet.

There are optimizations to this algorithm. For example, in the assignment PC, we adjusted this algorithm so that it favors replacing pages whose dirty bits are not set, to avoid the extra time of a drive write whenever possible. I'm not going to go into it aside from mentioning that such an optimization exists.

There is another replacement algorithm called "least recently used" (LRU), and it is exactly analogous to the replacement policy used by fully associative cache. Essentially, you keep track of the exact history of frame accesses, and replace the one that was accessed least recently.

3.8 Replacing the Frame

Ideally this frame will be empty, but if the frame already holds data, several steps must be taken. I go through the basics of the steps here.

1. First, we find a suitable frame in main memory to copy the desired page to. This can be done any number of ways, but the way that we've had the most experience with is the not recently used algorithm.
2. Once a frame is found that we can load into, we check to see if there is data in this frame; your method for doing this may vary. If there isn't, we can copy our page from secondary to main memory without consideration of what was there... since there wasn't anything there in the first place... so skip to step 5.
3. If there is data already in this page frame, the "valid bit" in the page table for which this page is a copy must be cleared to signify that the page we're replacing may no longer be found in main memory.
4. Further, if the data we're to get rid of has been modified (ie, the "dirty bit" is active), the changed page must be written back to DISK so that changes are not lost.
5. Copy the data we want to load from the page in DISK to the appropriate frame in main memory. In the page table, for the element of this page, set the valid bit to 1 (since the page is now held in main memory), the dirty bit to 0 (because this is new virgin data exactly as it is in secondary memory), and put the frame number of the frame to which we loaded it in "frame number."

3.9 Memory Management Hardware

In a multiprogramming environment where many programs reside in memory it becomes necessary to move programs and data around the memory, to vary the amount of memory in use by a given program, and to prevent a program from changing other programs. The demands on computer memory brought about my multiprogramming have created the need for a memory management system. A memory management system is a collection of hardware and software procedures for managing the various programs residing in memory. The memory management software is part of an overall operating system available in many computers. Here we are concerned with the hardware unit associated with the memory management system.

The basic components of a memory management unit are:

1. A facility for dynamic storage relocation that maps logical memory references into physical memory address
2. A provision for sharing common programs stored in memory by different users.
3. Protection of information against unauthorized access between users and preventing users from changing operating system functions.

The dynamic storage relocation hardware is a mapping process similar to the paging system. The fixed page size used in the virtual memory system causes certain difficulties with respect to program size and the logical structure of programs. It is more convenient to divide programs and data into logical parts called segments. A segment is a set of logically related instructions or data elements associated with a given name. Segments may be generated by the programmer or by the operating system. Examples of segments are subroutine, an array of data, a table of symbols, or a user's program.

The sharing of common programs is an integral part of a multiprogramming system. For example, several users wishing to compiler their FORTRAN programs should be able to share a single copy of the compiler rather than each user having a separate copy in memory. Other system programs residing in memory are also shared by all users in a multiprogramming system without having to produce multiple copies.

The third issue in multiprogramming is protecting one program from unwanted interaction with another. An example of unwanted interaction is one user's unauthorized copying of another user's program. Another aspect of protection is concerned with preventing the occasional user from performing operating system functions and thereby interrupting the orderly sequence of operations in a computer installation. The secrecy of certain programs must by keep from unauthorized personnel to prevent abuses in the confidential activities of an organization.

The address generated by a segmented program is called a logical address. This is similar to a virtual address except that logical address space is associated with variable-length segments rather than fixed-length pages. The logical address may be larger than the physical memory address as in virtual memory, but it may also be equal, and sometimes even smaller than the length of the physical memory address. In addition to relocation information, each segment has protection information associated with it. Shared programs are places in a unique segment in each user's logical address space so that a single physical copy can be shared. The function of the memory management unit is to map logical addresses into physical addresses similar to the virtual memory mapping concept.

3.10 Summary

In this lesson we have discussed about virtual memory, which is a memory management technique used in computer architecture to extend the available RAM artificially. It utilizes the hard disk as an extension of RAM, allowing the system to store less frequently used data in virtual memory when the physical RAM becomes full. When needed, data is swapped between RAM and virtual memory, enabling larger program execution and multitasking without running out of physical memory. Virtual memory plays a crucial role in enhancing system performance and enabling the execution of more substantial programs than the available physical RAM would allow.

3.11 Self Check Exercise

1. An address space is specified by 24 bits & the corresponding memory space is 16 bits.
   a) How many words are there in address space?
   b) How many words are there in memory space?
   c) If a page has 2k words, how many pages & blocks are in the system?
2. What is Virtual memory?
3. Explain the various part of the Virtual memory?

3.12 Suggested Readings

1. "Structured Computer Organization" by Tanenbaum.
3. M. Morris Mano, “Computer System Architecture”, PHI.

BCA SEM-3 **PAPER BCAB2103T COMPUTER SYSTEM ORGANIZATION AND ARCHITECTURE**

LESSON No. 4 **Author: Dr. Jyotsna Sen Gupta**

Converted into SLM by: Dr. Vishal Singh

INSTRUCTION IMPLEMENTATION

4.0 Objectives

In this lesson, the student will learn about the implementation of an instruction. The student will also learn as to how the instructions are stored in the memory. This lesson will also explain about the format of and types of instructions. In the end, various addressing modes used while implementing an instruction will be discussed.

4.1 Introduction

Implementation of an instruction requires many computations to take place at the hardware level. The instructions are stored in memory. The Central Processing Unit (CPU) should know in which addresses the instructions are stored and in what addresses, data is stored. The instructions are stored in various formats. The instructions are stored in various parts of the memory. There are various methods by which the program will address the instructions.

4.2 Functional Units of a Computer

Computer organization is the structure of high level parts of the computer like - CPU, memory and input-output devices, whereas computer design is the structure of low level parts like gates, flip-flops, logic arrays etc. The organization of a computer consists of study of three main functional units which are: Processor, Memory and Input/Output devices. The control unit fetches the instructions from the main memory and determines their type. It generates control signals and microoperations. The arithmetic and logic unit (ALU) performs operations like addition, subtraction, Boolean AND, OR etc. that are needed to carry out instructions. Registers are a very high speed memory within the CPU that stores temporary results, data for computation and certain control information. The most commonly used registers are the Program counter, Instruction register, Memory address register, Memory Buffer register.

The organization of the CPU is based on the functions it performs which are to fetch an instruction, decode it and execute it. They are explained as follows:

1. **Fetch Instruction:** CPU reads instructions from memory
2. **Change Program Counter (PC)**
3. **Interpret instruction:** Determine the type of instruction fetched.
4. **Decode Instruction:** Instruction is decoded to determine what action is required.
5. **Fetch data:** If the instruction uses data, fetch it from memory or I/O devices
6. **Execute instruction:** The execution of the instruction may require arithmetic or logical operations on data

7. **Write data:** The results are stored in the memory location.

To do these operations, CPU needs to store some data temporarily in the CPU registers. Figure below shows the block diagram of a processor. The input to the processor is the instructions from the memory that are to be executed, and data from the memory that is needed for executing these instructions. The output of the processor is the resulting data from execution and is stored in the memory. The three units are interconnected through internal buses, which are different from the system buses.

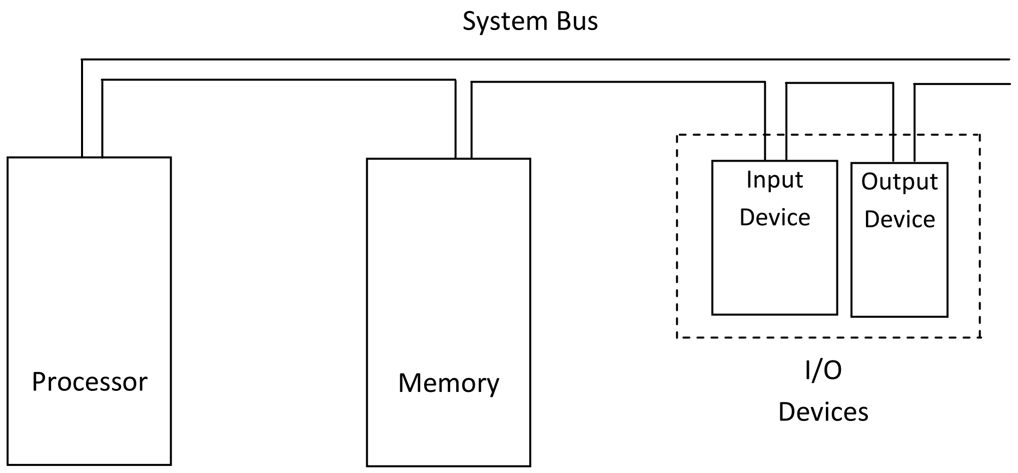


Figure: Functional Units of a simple computer with two I/O devices

4.3 Instruction Implementation

A computer program comprises of a sequence of instructions. Each instruction is executed by the CPU through a sequence of steps. This process consists of execution of a set of instructions stored in the memory. The instructions constituting the program are made to go through various execution cycles, one for every instruction. Each step is executed as a set of microoperations. These instruction cycles consists of the following phases:

1. Fetch an instruction
2. Decode the fetched instruction
3. Get data from the effective address of operands (if any)
4. Execute the instruction

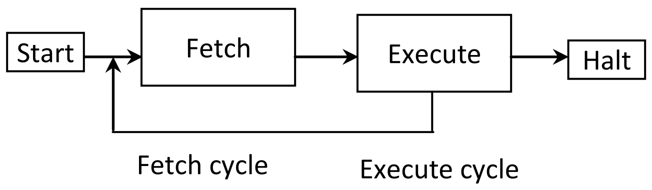


Figure: Basic Instruction cycle

In the beginning of each instruction cycle, the processor fetches the instruction from the memory. Program Counter (PC) is a register of the processor that contains the address of next instruction to be fetched. In the beginning, the address of the first instruction is entered into PC. After the instruction is fetched, the PC is incremented. The fetched instruction is loaded in the instruction register (IR). This instruction is interpreted by the decoder and the required action is performed. Accumulator (AC) is a special register which holds one of the operands for the operation to be performed, and is directly connected to the ALU. After this, the instruction is fetched from the address, decoded and executed by the processor. Similarly, other instructions are also executed one by one.

The execution of an instruction that adds two numbers is illustrated below:

Two 16-bit numbers stored in memory locations with addresses 100 and 101 have to be added. The instruction is 16 bits: 4 bits for the opcode, 12 bits for the operand. The memory words at addresses 830 and 831 are to be added. Instructions of a program add the contents of the memory word at address 830 to the contents of the memory word stored at address 831. The results are stored at address 831. The implementation requires three fetch and three execute cycles, as described below:

1. PC contains address 100, which is the address of the first instruction. The instruction is loaded in the IR.
2. The first 4 bits of the IR are for the opcode which indicates that AC is to be loaded. The remaining 12 bits hold the address 830.
3. The PC is incremented.
4. The contents of AC (location 830) and the contents of 831 are added. The result is stored in AC.
5. PC is incremented and the next instruction is fetched.
6. The contents of AC are stored in location 831.

As can be seen from the above mentioned 6 steps for the addition of two numbers, three instruction cycles are used. Each of this instruction cycle consists of a ‘fetch’ and ‘execute’ cycle.

4.4 Timing and Control

The various steps of instruction cycle take place in a particular time slot. The timing is handled by a master clock generator, which manages the timing for all operations in a basic computer. All electronic elements have an ENABLE pin activated by the control unit to obey the timer. These control signals have their origin in the control unit and act as control input to multiplexers of shared bus, registers etc. To understand the various control signals necessary for timing in a computer, it is very important to first understand the structure of a control unit of that particular computer.

4.5 Stored Program

A computer instruction is a binary code that represents a sequence of microinstruction for the computer. All the instructions of the program that have to be executed are stored in the computer memory. Each instruction is then read from the memory and placed in a control register. The binary code of this instruction is interpreted, and finally it is executed by the control unit. The **operation code (also known as opcode)** of an instruction is a group of bits that define operations such as add, subtract, multiply, shift, complement etc. The opcode must consist of at least 'n' bits for a given 2^n (or less) different operations to be performed. The operation which the opcode specifies must be performed on some data stored in the memory. An instruction code, hence, must specify not only the operation but also the registers or the memory words where the operands are to be found. The register or memory words where the operands are to be stored need also be specified.

Figure 1 is an illustration of the stored program in the memory of size 4096x16 bits. A part of this memory is used to store the instructions. Operands are stored in a different region of the memory. Processor registers are temporary storage and also hold instructions and operands. Figure 2 depicts instructions and data stored in the storage memory and registers. The format of the instruction in Figure 2 shows that it is a 16 bit instruction, with bits 12-15 used for the opcode and bits 0-11 used for the address of the operand.

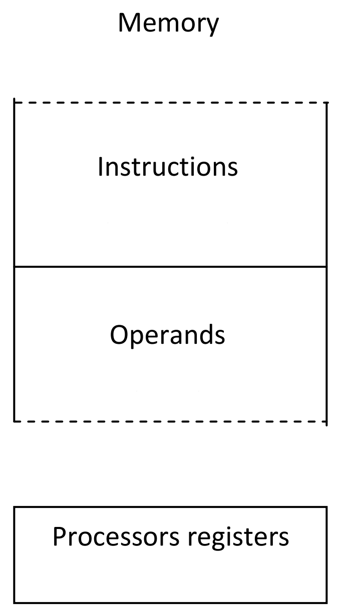


Figure 1 Stored Program

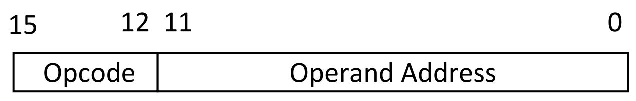


Figure 2 Instruction Format

4.6 Instruction Format

Each instruction is represented as a sequence of binary digits within a computer. The instruction is divided into different fields, corresponding to the elements of the instruction. The format of a simple instruction is given in Figure 1.10.4, where the instruction has 16 bits that are divided as: 4 bits for opcode, 6 bits for source address, and 6 bits for destination address.

The instruction contains the information required by the CPU for its execution. To accomplish this, the instruction must contain the following essential parts:

1. Operation Code: To specify the operation to be performed, like ADD, Read etc. The operation is specified by binary code, called operation code or opcode.
2. Source operand: The operation may involve one or more source operands that are inputs to the operation.
3. Result operand: The operation may produce a result operand.
4. Next instruction reference: After the execution of the current instruction, the CPU fetches next instruction from this location.

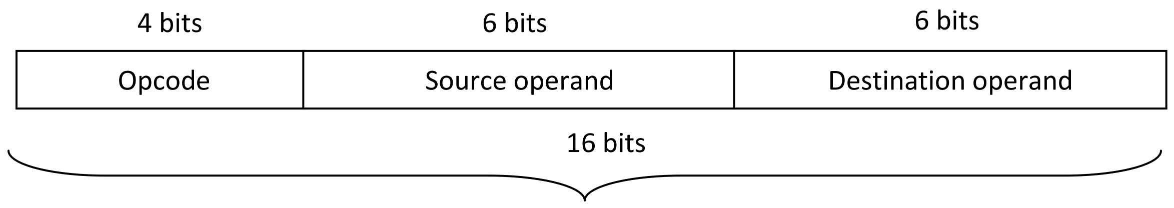


Figure: General Instruction Format

4.7 Instruction Cycle

A central processing unit normally contains three main components: a control unit, an arithmetic and logic unit and a register section. It is the control unit which is responsible for the control and synchronization of the actions of the CPU. The program consists of a sequence of instructions which is executed by going through a cycle for each instruction.

Every instruction cycle is further comprised of a sequence of subcycles. An instruction cycle consists of the following subcycles or phases.

1. Fetch instruction from memory.
2. Decode the instruction.
3. Read effective address from memory.
4. Execute the instruction.

After step 4, control goes to step 1 again.

4.8 Instruction Types

Different types of instructions are available in the computer to perform various actions. The instruction that is used the most is for copying data from one place to another. Copying means creating an identical bit pattern as the original at the destination location. The data is copied for two reasons. First is the assignment of values to variables, such as X = Y, where the value in memory location Y is copied to memory location X. The second reason is to make the data available at some other memory location where it is more easy to use it. For example, variables are copied from memory locations to registers.

Consider the operation that adds two variables X and Y to produce the third variable Z. The statement Z = X + Y in a high-level language program is a command to the processor to add the current values of the two variables called X and Y, and to assign the sum to a third variable Z. When the program containing this statement is compiled, the three variables X, Y, and Z are assigned to different locations in the memory. The contents of these locations represent the values of the three variables. Hence, the above high-level language statement requires the following action to take place in the computer.

$$[Z] \leftarrow [X] + [Y]$$

For the execution of this statement, the contents of memory locations X and Y are fetched from the memory into processor registers where their sum is calculated. This result is then stored in location C. This operation can be accomplished in various ways. The address provided by the program to access the memory may be provided in three separate ways. These methods are explained as follows:

Three-address instruction

Three-address instruction provides three separate addresses for three operands in a single machine instruction. The following instruction is a three address instruction with addresses X, Y and Z. This three-address instruction can be represented symbolically as:

ADD X, Y, Z

Operands X and Y are called the source operands, Z is called the destination operand, and ADD is the operation to be performed on the operands. A general three-address instruction has the format:

Operation Addr1, Addr2, Addr3

Where Addr1 and Addr2 are the source addresses and Addr3 is the destination address.

Two address instruction

A two-address instruction has only two operands and has the form

Operation Addr1, Addr2

Where Addr1 and Addr2 are the source and destination addresses respectively.

A two address ADD instruction is of the type:

ADD X, Y

The above statement performs the operation [Y] ← [X] + [Y]. When the sum is calculated, the result is sent to the memory and stored in location X, replacing the original contents of this location. This means that operand Y is both a source and a destination.

One address instruction

One-address instruction specifies only one memory operand. When a second operand is needed, as in the case of an ADD instruction, it is understood implicitly to be a unique location. A processor register, usually called the accumulator, may be used for this purpose. Thus, the one-address instruction is:

ADD X

Which means the following: Add the contents of memory location X to the contents of the accumulator register and place the sum back into the accumulator.

Using only one-address instructions, the operation [Z] ← [X] + [Y] can be performed by executing the sequence of instructions:

```
LDA   X   {Load X}
ADD   Y   {Add Y to X}
STO   Z   {Store Z}
```

The operand specified in the one address instruction may be a source or a destination, depending on the instruction. In the LDA instruction, address X specifies the source operand and the destination operand is accumulator, which is implied. Address Z specifies the destination location in the STO instruction and here the accumulator is the implied source.

Move instruction

The movement of data between different memory locations is the most used instruction by a programmer. The transfer of data between different locations of memory is achieved with the instruction:

MOV Source, Destination

This instruction places a copy of the contents of Source into Destination. When data are moved to or from a processor register, the MOV instruction can be used rather than the LDA or STO instructions because the order of the source and destination operands determines which operation was intended.

Zero Address Instructions

Zero address instructions are those in which the locations of all operands are defined implicitly. The execution of these instructions is the fastest as the speed with which a given task is carried out depends on the time it takes to transfer instructions from memory into the processor and to access the operand. These instructions are generally used for stacks.

4.9 Addressing Modes

Data used for computation can be organized in various ways in different locations of the memory. Different addressing modes are used to address the memory to access the required operand(s) stored in various ways in different memory locations. The special way in which the location of an operand is specified in an instruction is referred to as 'addressing mode'.

The various data types represented in a computer program are constants, variables, pointers, and arrays. In assembly language, a variable is represented by allocating a register or memory location to hold its value. The address field of a typical instruction is relatively small. The memory locations to be addressed may be in the main memory or in the virtual memory. The aim is to reference a large number of memory locations. To achieve this, a variety of addressing methods are employed. Some methods are able to access more number of locations but to access such a location will involve extra time. Thus, to choose an addressing mode involves trade-offs between address range and number of memory references. Some of the common addressing techniques are as under:

i) Immediate addressing
ii) Direct addressing
iii) Indirect addressing
iv) Register addressing
v) Register Indirect addressing
vi) Displacement addressing
vii) Stack addressing

The following notations are used for explanation:

i) M – Contents of address field in the instruction
ii) R – Contents of address field that refer to a register
iii) EA – Effective Address of memory location containing the operand
iv) (X) – Contents of location X

One or more bits in the instruction format are used as the mode field for the control unit to determine which addressing mode is being used. The effective address will either be a main memory address, virtual memory address or a register.

4.9.1 Immediate addressing

Immediate addressing is the simplest form of addressing. In this form of addressing, the operand is actually present in the instruction.

OPERAND = M

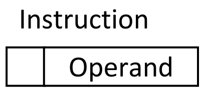


Figure: Immediate Addressing

This mode is used to set initial values of variables, or to define constants. The number is set as two's complement and the left-most bit of the operand is used as the sign bit. Advantage of this mode is that only the instruction-fetch is required and no memory reference is needed, thus saving one instruction cycle. Disadvantage is that the size of the number is restricted to the size of the address field.

4.9.2 Direct addressing

This is also a very simple form of addressing. In direct addressing, the address field contains the effective address of the operand, as shown in Figure below.

EA = M

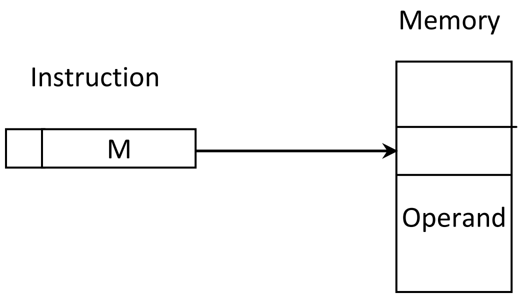


Figure: Direct Addressing

Only one memory reference is required in this addressing mode. The limitation is that it makes available only limited address spaces and is usually less than the word length.

4.9.3 Indirect addressing

In indirect addressing, the address field refers to address of a word in memory, which in turn contains the full address of the operand. This is demonstrated in Figure below.

A = [M]

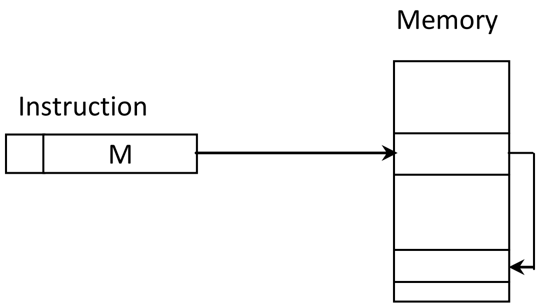


Figure: Indirect Addressing

The advantage of this mode is that for word length of N, address space of 2^N is available. The disadvantage is that two memory references are needed to fetch the operand. One reference is required to get the address of the operand and the second reference gets its value.

4.9.4 Register Addressing

Register addressing is similar to direct addressing. The difference is that the address field refers to a register rather than a memory location. This form of addressing is exhibit in Figure below.

The advantage of register addressing is that only small address field is needed in the instruction. This means that no memory reference is required. Thus, register addressing is very fast. The disadvantage of this mode is that address space very limited.

EA = R

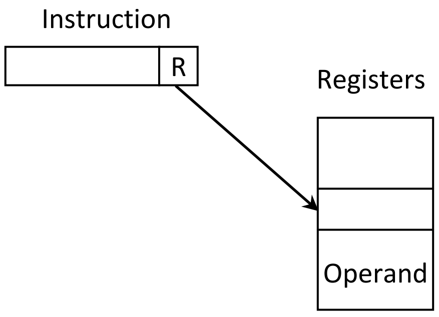


Figure: Register Addressing

4.9.5 Register Indirect Addressing

Register indirect addressing is similar to indirect addressing but address field refers to a register rather than a memory location. Figure below shows this form of addressing.

$$EA = [R]$$

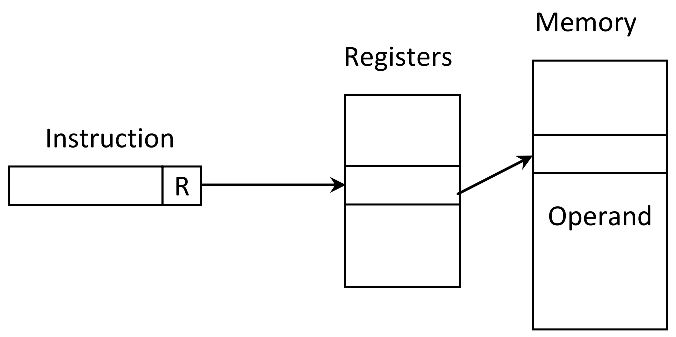


Figure: Register Indirect Addressing

The advantage of this form of addressing is that though it is the same as indirect addressing, but one less memory reference is used. The disadvantage is that it has to access the registers twice.

4.9.6 Relative Addressing

This form of addressing combines the capabilities of direct addressing and register indirect addressing. The effective address is given as:

$$EA = M + [R]$$

The implied register is the program counter. Current instruction address is added to the address field to produce effective address. The effective address is a displacement relative to the address of the instruction.

4.9.7 Base Register Addressing

Base register addressing has a register in reference which is known as the base register. The effective address is the memory address and a displacement from that address stored in the base register. The register reference may be explicit or implicit. This kind of addressing is quite convenient to implement segmentations. In some implementations, a single segment-base register is used. In other cases, a programmer may choose a register to hold the base address of the segment.

4.9.8 Index Addressing

In this form of memory reference, the address field references a main memory address, and the referenced register contains a positive displacement from that address. Figure below depicts Index addressing. This method is just the opposite of 'base-register' addressing. Here the address field is considered to be a memory address. The method of calculating the effective address is the same as base register addressing.

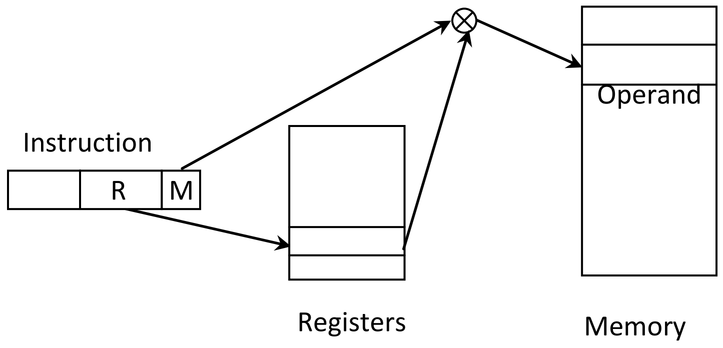


Figure: Index Addressing

Indexing is used in iterative operations and index registers are used for these operations. Here the index register is just incremented or decremented.

$$EA = EA + (R)$$

$$R \leftarrow (R) + 1$$

First the contents of the address field are used to access a memory location containing a direct address. The value of the index register is then added to the register value.

4.9.10 Stack addressing

Stack is a special block of contiguous memory locations. A special register, known as stack pointer is used to point to the address at one end of these locations, generally known as the top of stack. Stack addressing is similar to register indirect addressing is depicted in Figure below. Stack Pointer maintained in a register

$$EA = [R]$$

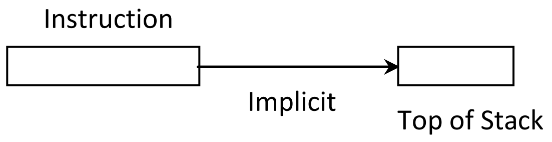


Figure: Stack Addressing

Stack mode of addressing is a form of implied addressing. The machine instructions do not include a memory reference. The top of the stack is the implied address.

4.10 Summary

In this lesson, we have discussed about the implementation of an instruction which is a single operation or command that a computer's central processing unit can execute. We have learned how the instructions are stored in the memory and have also studied about the format of and types of instructions. In the end, various addressing modes used while implementing an instruction were discussed.

4.11 Self-Check Exercise

1. Represent the structure of a computer showing the basic blocks and their interconnection through buses.
2. Explain the format of a computer instruction.
3. What is the significance of PC, DR, AR, IR registers in the instruction implementation of a computer?
4. What is the procedure of the execution of an instruction in a processor?
5. Which instruction is used the most in computations? Explain the different formats of instructions.
6. List the main number formats in which data can be represented.
7. Write an algorithm for multiplying two 4-bit positive integers. Assuming that each register transfer is performed in 1 microsecond, how much time will be taken by:
    i. zero-address instruction
    ii. one-address instruction
    iii. two-address instruction
    iv. three-address instruction.
8. How many references are needed for: (i) direct address instruction, (ii) indirect address instruction.
9. What is the difference between register addressing and register indirect addressing in terms of memory references. Give an example of their use.
10. Explain the difference between Base Register addressing and Index Addressing.

4.12 Suggested Reading

5. M. Morris Mano, Computer System Architecture, Third Edition, Pearson Education (Singapore), 1993
6. William Stalling, “Computer Organization and Architecture”, 6th edition, Pearson Education, 2000.
7. Hill, F.J. and G.R. Peterson, Digital Systems: Hardware Organization and Design, 3rd ed. New York: John Wiley, 1987.
8. Tanenbaum, Andrew S., Structured Computer Organization, 4th ed, Prentice-Hall of India, New Delhi, 2001

Last Updated on April 2023

Mandatory Student Feedback Form

https://forms.gle/KS5CLhvpwrpgjwN98

Note: Students, kindly click this google form link, and fill this feedback form once.