Computer Organization & Architecture

Reference Books:

1. Computer System Architecture 3rd Edition, By M Morris Mano
2. Computer Organization 5th Edition, by Carl Hamacher

This PDF contains the notes from the standards books and are only meant for GATE CSE aspirants.

Notes Compiled By-
Manu Thakur
Mtech CSE, IIT Delhi
worstguymanu@gmail.com
https://www.facebook.com/Worstguymanu

Central Processing Unit

T**he register set stores intermediate data used during the execution of the instructions** ALU performs the required micro-operations for executing the instructions. The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

A **computer instruction** is a binary code that specifies a sequence of micro-operations for the computer. Instruction codes together with data are stored in memory. The computer reads each instruction from memory and places it in a control register. The control then interprets the binary code of the instruction and proceeds to execute it by issuing a sequence of micro-operations.
Every computer has its own unique instruction set.

Instruction code
An instruction code is a **group of bits** that instruct the computer to perform a specific operation. It is usually divided into parts, each having its own particular interpretation.

Operation Code
The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement. The number of bits required for the operation code of an instruction depends on the total number of operations available in the computer. The operation code must consist of **at least n bits for a given 2^n (or less) distinct operations**.

For example, consider a computer with 64 distinct operations, one of them being an ADD operation. The operation code consists of six bits, with a bit configuration 110010 assigned to the ADD operation. When this operation code is decoded in the control unit, the computer issues control signals to read an operand from memory and add the operand to a processor register.

An operation is part of an instruction stored in computer memory. It is a binary code that tells the computer to perform a specific operation. The control unit receives the instruction from memory and interprets the operation code bits. It then issues a sequence of control signals to initiate micro-operations in internal computer registers. For every operation code, the control issues a sequence of micro-operations needed for the hardware implementation of the specified operation.

Note:-
An instruction code must the operation, registers or the memory words where the operands are to be found, as well as the register or memory word where the result is to be stored. Memory words can be specified in instruction codes by their address**. Processor registers can be specified by assigning to the instruction another binary code of k bits that specifies one of 2^k registers**.

Note: - Each computer has its own particular instruction code format. Instruction code formats are conceived by computer designers who specify the architecture of the computer.

Memory locations and addresses

The memory consists of many millions of storage cells, e**ach of which can store a bit of information having the value 0 or 1**. Because a single bit represents a very small amount of information, bits are seldom handled individually. The usual approach is to deal with them in groups of fixed size. For this purpose, the memory is organized so that a group of n bits can be stored or retrieved in a single, basic operation. Each group of n bits is referred to as a word of information, and n is called **the word length**.

If the word length of a computer is 32 bits, a single word can store a 32-bit 2's complement number or four ASCII characters, each occupying 8 bits. A unit of 8 bits is called a byte. Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct names or addresses for each item location.

BYTE ADDRESSABILITY:-

A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits. Successive addresses refer to successive byte.

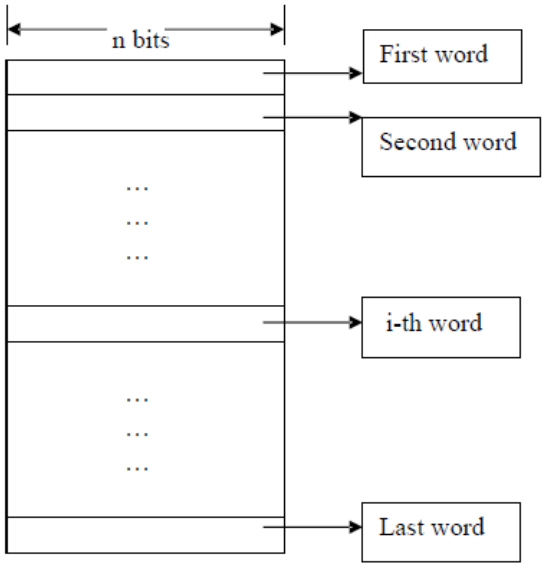


Fig: Memory Words

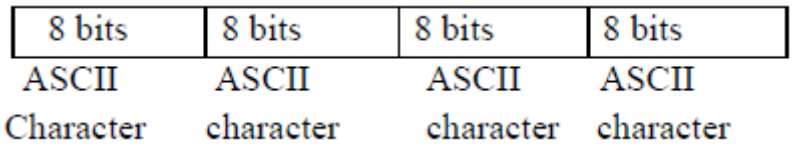


Fig: Four Characters

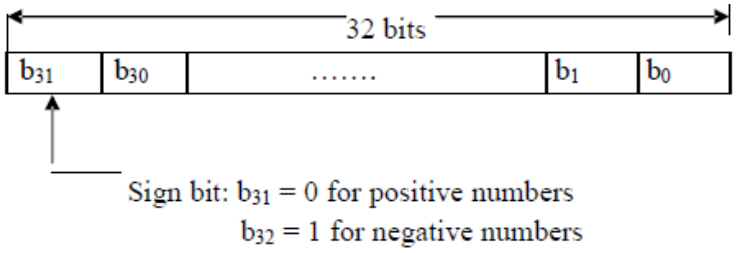


Fig: Signed Integer

Note:-
If the memory is byte addressable, then byte locations have addresses 0, 1, 2… and if the memory is word addressable and the length of each word is 32 bits, then successive words are located at addresses 0,4,8,…., with each word consisting of four bytes.

Big-endian & Little-endian Assignments:-
The name **big-endian** is used when lower byte addresses are used for the more significant bytes. The name little-endian is used for where the lower byte addresses are used for the less significant bytes of the word.

Instructions and instruction sequencing
A computer must have instructions capable of performing four types of operations.

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

Instruction execution and Straight-line sequencing:-

We assume that computer allows one memory operand per instruction and has multiple of processor registers. We assume that **word length is 32bits and memory is byte-addressable.**

C = [A] + [B] // three instructions in this program each of length 4 bytes or 1 word

The three instructions of the program are in successive word locations, starting at location i. **since each instruction is 4 bytes long, the second and third instructions start at addresses i + 4 and i + 8**, we also assume that full memory address can be directly specified by a **single-word** instruction.

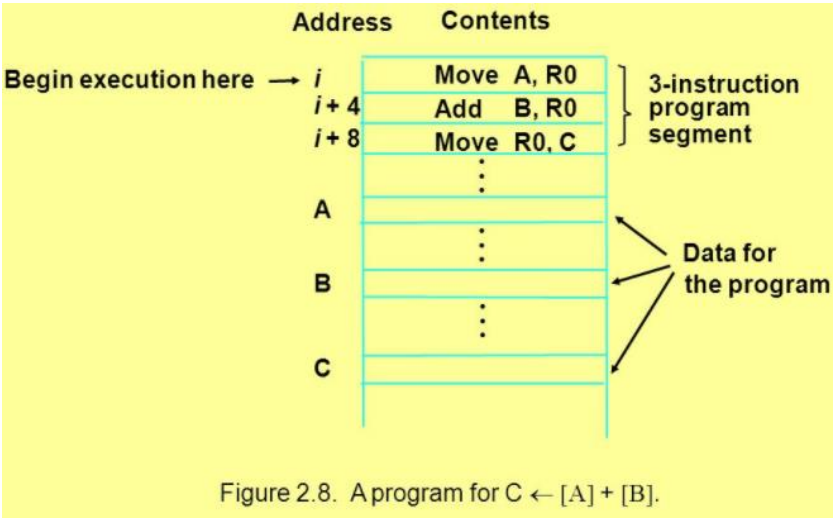


Figure 2.8. A program for C ← [A] + [B].

Let us consider how this program is executed. The processor contains a register called the **program counter** (PC), which holds the address of the instruction to be executed next. **To begin executing a program, the address of its first instruction must be placed into the PC**. Then, the processor control circuits use the information in the PC to fetch and execute instructions. **During the execution of each instruction, the PC is incremented by 4 to point to the next instruction**. Thus, after the Move instruction at location i + 8 is executed, the PC contains the value i + 12, which is the address of the first instruction of the next program segment.

Executing a given instruction is a **two-phase procedure**. In the first phase, called instruction fetch, the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the **instruction register** (IR) in the processor. The instruction in IR is examined to determine

which operation is to be performed. The specified operation is then performed by the processor. This often involves fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location.

Branching:-

Consider the task of adding a **list of n numbers**. Instead of using a long list of add instructions.

i	Move NUM1, R0
i+4	Add NUM2, R0
i+8	Add NUM3, R0
	…
	…
i+4n-4	Add NUMn, R0
i+4n	Move R0, SUM
	….
	….
SUM	
NUM1	
NUM2	
	….
	….
NUMn	

It is possible to place a single add **instruction in a program loop**. It starts at location LOOP and ends at the instruction Branch > 0.

	Move N, R1
	Clear R0
LOOP Program loop	Determine address of "Next" number and add "Next" number to R0
	Decrement R1
	Branch >0 LOOP
	Move R0, SUM
	…….
	…….
	…….
SUM	
N	n
NUM1	
NUM2	
	……
	……
	……
NUMn	

Using a loop to add n numbers

Store the value of N into R1, decrement R1 each time and check if its value > 0. When loop is completed store the value of R0 into SUM. **This type of instruction loads a new value into the program counter**. As a result, the processor fetches and executes the instruction at this new

address, called the branch target, instead of the instruction at the location that follows the branch instruction in sequential address order.

Addressing Modes:-

The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words. The way the operands are chosen during program execution is dependent on the addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction Before the operand is actually referenced. Addressing modes are used:

1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
2. To reduce the number of bits in the addressing field of the instruction.

The control unit of a computer is designed to go through an instruction cycle that is divided into three major phases:

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Execute the instruction

Program counter (PC)

PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory. The decoding done in step 2 determines the operation to be performed, the addressing mode of the instruction, and the location of the operands.

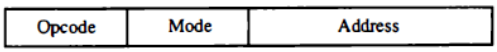
Note: - In some computers

1. The addressing mode of the instruction is specified with a distinct binary code, just like the opcode is specified.
2. A single binary code that designates both the operation and the mode of the instruction.
3. Sometimes, two or more addressing modes are combined in one instruction.

Mode field

The mode field is used to locate the operands needed for the operation. There may or may not be an address field in the instruction. Each address field may be associated with its own particular addressing mode.

Figure 8-6 Instruction format with mode field.



Implied Mode:

In this mode the operands are **specified implicitly in the definition of the instruction**.

1. All register reference instructions that use an accumulator are implied-mode instructions.
2. **Zero-address** instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

Immediate Mode:

In this mode the operand is specified in the instruction itself. An immediate-mode instruction has an operand field rather than an address field.

Note: When the address field specifies a processor register, the instruction is said to be in the register mode.

Direct Address Mode:
In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction the address field specifies the actual branch address.

Indirect Address Mode:
In this mode the address field of the instruction gives the address where the effective address is stored in memory. Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

Register Mode:
In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction. A k-bit field can specify any one of 2^k registers.

Register Indirect Mode:
In this mode the instruction specifies a register in **the CPU whose contents give the address of the operand in memory**. In other words, the selected register contains the address of the operand rather than the operand itself.
The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than a memory address directly.

Autoincrement or Autodecrement Mode:
This is similar to the register indirect mode except that the register is **incremented (after)** or **decremented (before) after (or before)** its value is used to access memory.
Auto-increment mode automatically increments the contents of a register by a value that corresponds to the size of the accessed operand. Thus, the increment is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for 32-bit operands. The size of the operand is usually specified as part of the operation code of an instruction
Note: - The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode.

A few addressing modes require that the address field of the **instruction be added to the content of a specific register in the CPU**.

Effective address = address part of instruction + content of CPU register

Note: The CPU register used in the computation may be the **program counter**, an **index register**, or a **base register**.

Relative Address Mode:
In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address. The address part of the instruction is usually a signed number (in 2's complement representation) which can be **either positive or negative**.
For example, assume that the PC contains the number 825 and the address part of the instruction contains the number 24. The instruction at location 825 is read from memory during the fetch phase and the program counter is then incremented by one to 826. The effective address computation for the relative address mode is 826 + 24 = 850. **It results in a shorter address field** in the instruction format since the relative address can be specified with a smaller number of bits compared to the entire memory address. It's generally used in **Branch-Type** instructions.

Indexed Addressing Mode:

In this mode the content of an index register is added to the address part of the instruction to obtain the effective address. **The index register is a special CPU register that contains an index value**. **The Address field of the instruction defines the beginning address of a data array in memory**. Each Operand in the array is stored in memory relative to the beginning address. The index register can be incremented to facilitate access to consecutive operands.

Base Register Addressing Mode:-

In this mode the content of a base register is added to the address part of the instruction to obtain the effective address. This is similar to the indexed addressing mode except that the register is now Called a base register instead of an index register.

An index register is assumed to hold an index number that is relative to the address part of the instruction. **A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address**. The base register addressing mode is used in computers to facilitate the relocation of programs in memory. When programs and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of instructions must reflect this change of position. With a base register, the displacement values of instructions do not have to change. Only the value of the base register requires updating to reflect the beginning of a new memory segment.

Numerical Example:-

The two-word instruction at address 200 and 201 is a "load to AC" instruction with an address field equal to 500.

The first word of the instruction specifies the operation code and mode, and the second word specifies the address part. PC has the value 200 for fetching this instruction. The content of processor register R1 is 400, and the content of an index register XR is 100. **AC** receives the operand after the instruction is executed.

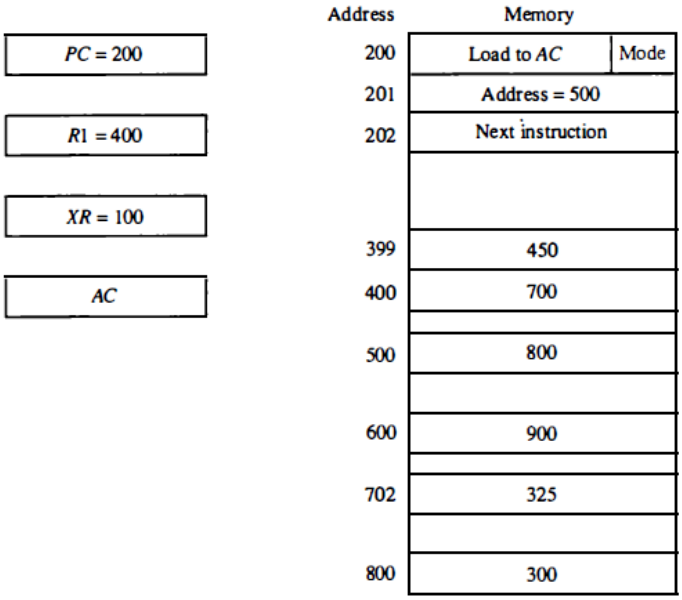


Figure 8-7 Numerical example for addressing modes.

- **Direct address mode** effective address: 500, 800 will be loaded in **AC**.
- **Immediate mode** AC: 500 (effective address in this case is 201)
- **Indirect mode** EA: 800, AC:300
- **Relative mode:** EA:500+202(PC) = 702, AC: 325
- **Index mode:** EA: XR + 500 = 100+500=600, AC: 900

- **Register mode**: The operand is in R1 and 400 is loaded into AC. (There is no effective address in this case.)
- **Register indirect mode**: EA=400 (content of R1), AC:700
- **Autoincrement mode:** same as the register indirect mode except that R1 is incremented to 401 **after the execution** of the instruction mode. AC: 700
- **Autodecrement mode**: decrements R1 to 399 **prior to the execution** of the instruction. AC:450

Table 2.1 Generic addressing modes

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

EA = effective address
Value = a signed number

Note: - Where X denotes the constant value contained in the instruction. [Ri] means content of the register Ri.

Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC + ADR]$
Immediate operand	LD #NBR	$AC \leftarrow NBR$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR + XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1], R1 \leftarrow R1 + 1$

Stack Organization:

A stack is a list of data elements, usually **words or bytes**, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the **top of the stack**, and the other end is called the **bottom**. The structure is sometimes referred as a **pushdown** stack.

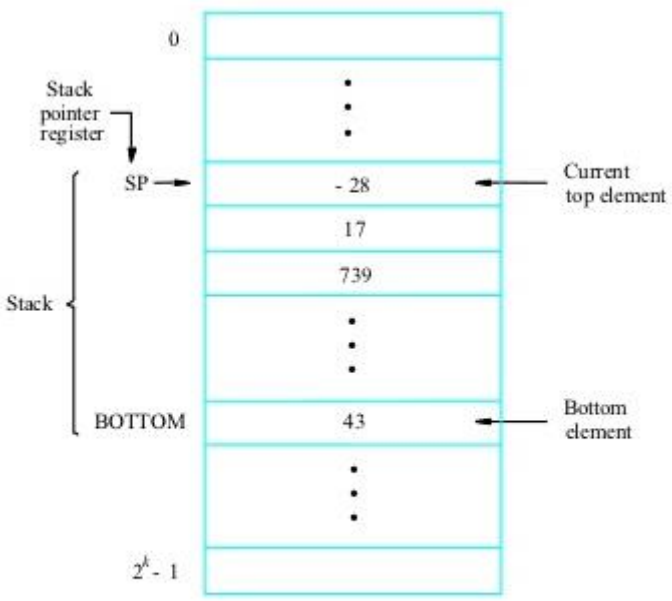


Figure 2.21. A stack of words in the memory.

Fig 2.21, Stack contains numerical values, with 43 at the bottom and -28 at the top. A processor register is used to keep track of the address of the element of the stack that is at the top at any given time. This register is called the **stack pointer** (SP). It could be one of the general-purpose registers or a register dedicated to this function.

If we assume a **byte-addressable memory with a 32-bit word length**, the **push operation** can be implemented as

Subtract #4, SP
Move NEWITEM, (SP)

These two instructions move the word from the location NEWITEM onto the top of the stack, decrementing the stack pointer by 4 before the move.
The **POP operation** can be implemented as

Move (SP), ITEM
Add #4, SP

These two instructions move the top value from the stack into location ITEM, and then increment the stack pointer by 4.

If the processor has **autoincrement** and **autodecrement** addressing modes:

Move NEWITEM, -SP (Push operation, pre-decrement)
Move (SP)+, ITEM (Pop Operation, post-decrement)

Note: - We should avoid attempting to pop an item off an empty stack, and push into a full stack.

Queue Organization:

Data are stored in and retrieved from a queue on a **first-in-first-out** (FIFO) basis. Thus, if we assume that the queue grows in the direction of increasing addresses in the memory, which is a common practice, new data are added at the back (high-address end) and retrieved from the front (low-address end) of the queue.

Both ends of a queue move to higher addresses as data are added at the back and removed from the front. So two pointers are needed to keep track of the two ends of the queue.

Another difference between a stack and a queue is that, without further control**, a queue would continuously move through the memory of a computer in the direction of higher addresses**. One way to limit the queue to a fixed region in memory is to use a **circular buffer**.

Instruction Formats

A computer will usually have **a variety of instruction code formats**. It is the **function of the control unit within the CPU to interpret each instruction code** and provide the necessary control functions needed to process the instruction.

The most common fields found in instruction formats are:

1. An operation code field that specifies the operation to be performed.
2. An address field that designates a memory address or a processor register.
3. A mode field that specifies the way the operand or the effective address is determined.

Note:

1. Other special fields are sometimes employed under certain circumstances.
2. Computers may have instructions of several different lengths containing varying number of addresses.

Most computers fall into one of three types of CPU organizations:

1. Single accumulator organization.
2. General register organization.
3. Stack organization.

a. **Accumulator-Type Organization** All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field.

ADD X

AC ← AC + M[X]

[Where X is the address of the operand]

b. **General Register Type Organization** The instruction format in this type of computer needs three register address fields.
   - **i.** **ADD R1, R2, R3**
   - **ii.** **ADD R 1, R 2** (if destination register is same as source reg.)
   - **iii.** **MOV R1, R2 (**R1←R2 or R2←R1, depending on the particular computer).

General register-type computers employ two or three address fields in their instruction format. Each address field may specify a processor register or a memory word. An instruction symbolized by

ADD R1, X

R1 ← R1 + M[X]

c. **Stack organization**
Computers with stack organization would have PUSH and POP instructions which require an address field.
PUSH X (push the word at address X to the top of the stack.)

The instruction **ADD**, in a stack computer consists of **an operation code only with no address field**. This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack.

Three-Address Instructions

Computers with three-address instruction formats can use each address field to specify either a **processor register or a memory operand**.

X = (A + B) * (C + D)

```
ADD   R1, A, B     R1←M[A] + M[B]
ADD   R2, C, D     R2←M[C] + M[D]
MUL   X, R1, R2    M[X]←R1*R2
```

The advantage of the three-address format is that it results in **short programs** when evaluating arithmetic expressions.

Two-Address Instructions

Each address field can specify either a processor register or a memory word. The program to evaluate X = (A + B) * (C + D) is as follows

```
MOV   R1, A     R1←M[A]
ADD   R1, B     R1←R1 + M[B]
MOV   R2, C     R2←M[C]
ADD   R2, D     R2←R2 + M[D]
MUL   R1,R2     R1←R1*R2
MOV   X, R1     M[X]←R1
```

Note: - The MOV instruction moves or transfers the operands **to and from memory** and processor registers.

One-Address Instructions

One-address instructions use an implied accumulator (AC) register for all data manipulation. The program to evaluate X = (A + B) * (C + D) is

```
LOAD    A    AC←M[A]
ADD     B    AC←AC + M[B]
STORE   T    M[T]←AC
LOAD    C    AC←M[C]
ADD     D    AC←AC + M[D]
MUL     T    AC←AC*M[T]
STORE   X    M[X]←AC
```

All operations are done between the AC register and a memory operand. T is the address of a temporary memory location required for storing the intermediate result.

Zero-Address Instructions

A stack-organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack. The following program shows how X = (A + B) * (C + D) will be written for a stack organized computer.

```
PUSH    A    TOS ← A
PUSH    B    TOS ← B
ADD          TOS ← (A + B)
PUSH    C    TOS ← C
PUSH    D    TOS ← D
ADD          TOS ← (C + D)
MUL          TOS ← (C + D) * (A + B)
POP     X    M[X] ← TOS
```

Note: - To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation.

RISC Instructions

The instruction set of a typical RISC processor is **restricted** to the use of **load** and **store** instructions when communicating between memory and CPU.

All other instructions are executed within the registers of the CPU without referring to memory.

Following is a program to evaluate X = (A + B) * (C + D)

```
LOAD    R1, A          R1 ← M[A]
LOAD    R2, B          R2 ← M[B]
LOAD    R3, C          R3 ← M[C]
LOAD    R4, D          R4 ← M[D]
ADD     R1, R1, R2     R1 ← R1 + R2
ADD     R3, R3, R2     R3 ← R3 + R4
MUL     R1, R1, R3     R1 ← R1 * R3
STORE   X, R1          M[X] ← R1
```

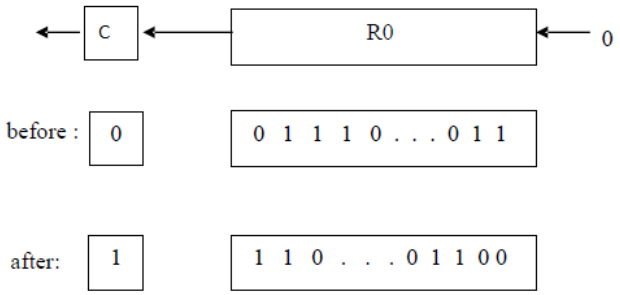
Shift and Rotate Instructions:-

The details of how the **shifts** are performed depend on whether the operand is a signed number or some more general binary-coded information. For general operands, we use a **logical shift**. For a number, we use an arithmetic shift, which preserves the sign of the number.

1. **Logical Shifts**

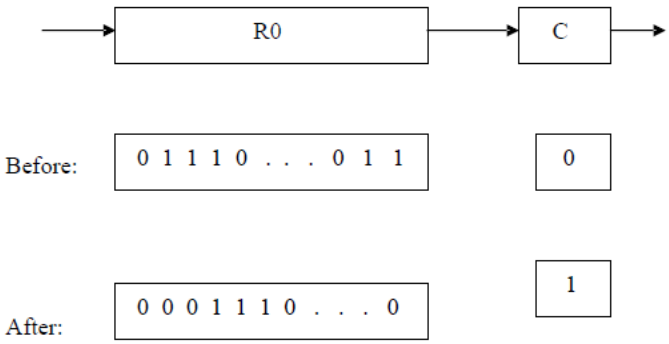
a. **Logical Left shifts (LShiftL)** shifts left an operand over a number of bit positions specified in a count operand contained in the instruction.

(a) Logical shift left LShiftL #2, R0



b. **Logical Right Shifts (LShiftR)** LShiftR works in the same manner except it shifts to right.

(b) Logical shift right LShiftR #2, R0

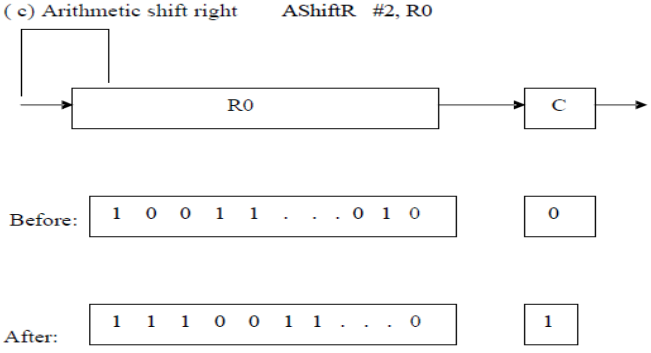


2. Arithmetic Shifts It preserves the sign bit.

a. **Arithmetic Left Shift (AshiftL)** is exactly same as the Logical left shift. Shifting a number one bit position to left is equivalent to multiplying it by 2.
Note: - Overflow might occur on shifting left.

b. **Arithmetic Right Shift (AShiftR)** on a right shift the sign bit must be repeated as the fill-in bit for the vacated position. **Remainder is lost in shifting right**.

(c) Arithmetic shift right AShiftR #2, R0



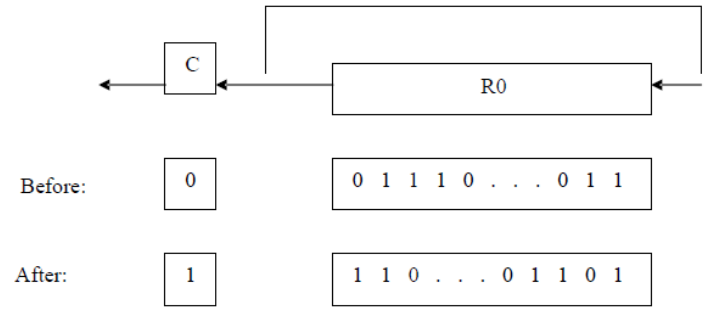
Rotate operations:

In the shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry flag C. **To preserve all bits, a set of rotate instructions can be used**. They move the bits that are shifted out of one end of the operand back into the other end.

Two versions of both the left and right rotate instructions are usually provided.

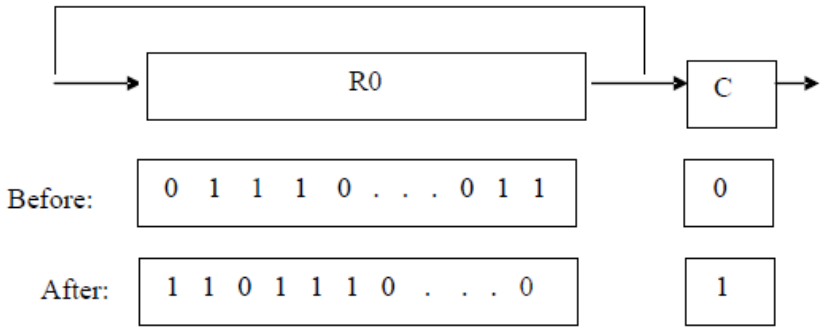
1. The bits of the operand are simply rotated.
2. In the other version, the rotation includes the C flag.

a. **Rotate left without carry**



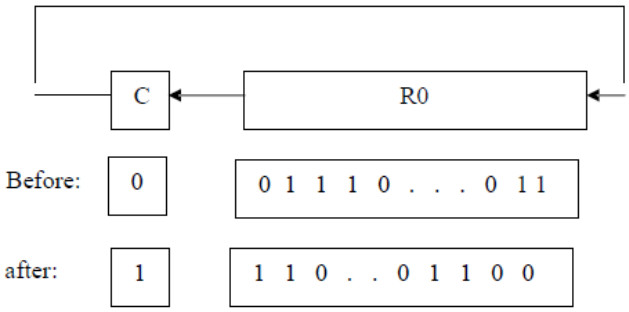
RotateL #2, R0

b. **Rotate right without carry**



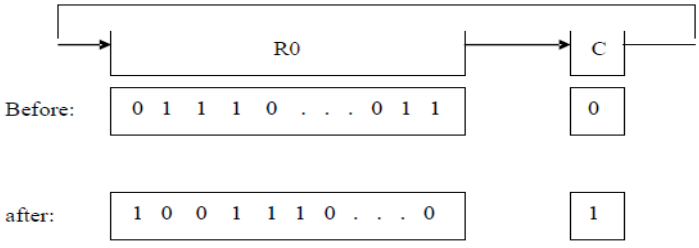
RoateR #2, R0

c. **Rotate left with carry**



RotateLC #R, R0

d. **Rotate right with carry**



Rotate RC #2, R0

Reduced Instruction Set Computer (RISC)

The instruction set chosen for a particular computer determines the way that machine language programs are constructed.
RISC computers use **fewer instructions with simple constructs** so they can be executed much faster within the CPU without having to use memory as often.

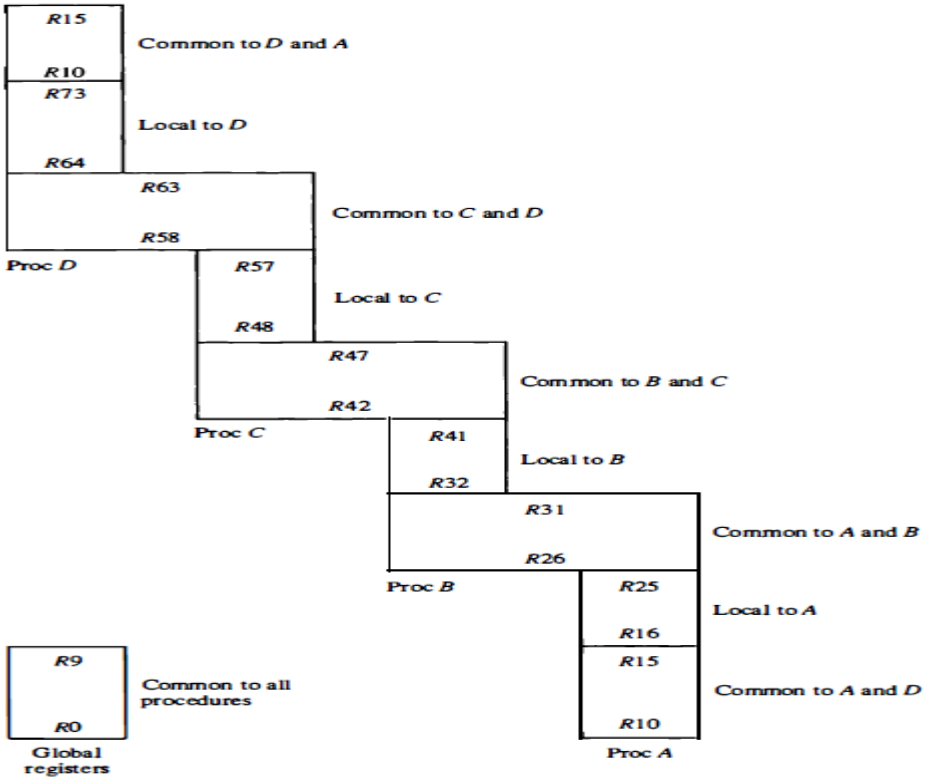
RISC Characteristics:-
The concept of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer. The major characteristics of a RISC processor are:

1. Relatively few instructions
2. Relatively few addressing modes
3. Memory access limited to load and store instructions
4. All operations done within the registers of the CPU
5. Fixed-length, easily decoded instruction format
6. Single-cycle instruction execution, CPI=1
7. Hardwired rather than microprogrammed control
8. A relatively large number of registers in the processor unit
9. Use of overlapped register windows to speed-up procedure call and return.
10. Efficient instruction pipeline, CPI=1
11. Compiler support for efficient translation of high-level language programs into machine language programs.

Overlapped Register Windows

A characteristic of some RISC processors is their use of **overlapped register windows** to provide the passing of parameters and avoid the need for saving and restoring register values.

i. number of global registers = G
ii. number of local registers in each window = L
iii. number of registers common to two windows = C
iv. number of windows = W



The number of registers available for **each window** is calculated as follows:

Window size = L + 2C + G

The total number of registers needed in the processor is

Register file = (L + C)W + G

Example: we have G = 10, L = 10, C = 6, and W = 4. The window size is 10 + 12 + 10 = 32 registers, and the register file consists of (10 + 6) x 4 + 10 = 74 registers.

Complex Instruction Set Computer (CISC)

A computer with a large number of instructions is classified as a complex instruction set computer. Characteristic of CISC architecture is the incorporation of **variable- length instruction** formats. The instructions in a typical CISC processor provide **direct manipulation of operands residing in memory**.

CISC Characteristics: -

The major characteristics of CISC architecture are:

1. A large number of instructions-typically from 100 to 250 instructions
2. Some instructions that perform specialized tasks and are used infrequently
3. A large variety of addressing modes-typically from 5 to 20 different modes
4. Variable-length instruction formats
5. Instructions that manipulate operands in memory

Subroutine Call and Return

A subroutine is a self-contained sequence of instructions that performs a given computational task. During the execution of a program, a subroutine may be called to perform its function many times at various points in the main program. Each time a subroutine is called, a branch is executed to the beginning of the subroutine to start executing its set of instructions. After the subroutine has been executed, a branch is made back to the main program.

A **call subroutine** instruction consists of an operation code together with an address that specifies the beginning of the subroutine. The instruction is executed by performing two operations:

1. The address of the next instruction available in the program counter (the return address) is stored in a temporary location so the subroutine knows where to return.
2. Control is transferred to the beginning of the subroutine.

The last instruction of every subroutine, commonly called return from subroutine, transfers the return address from the temporary location into the program counter. This results in a transfer of program control to the instruction whose address was originally stored in the temporary location.

Different computers use a different temporary location for storing the return address. Some store the return address in the first memory location of the subroutine, some store it in a fixed location in memory, some store it in a processor register, and some store it in a memory stack. The most efficient way is to store the return address in a memory stack. **The advantage of using a stack for the return address is that when a succession of subroutines is called, the sequential return addresses can be pushed into the stack**. A subroutine call uses following operations:

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Push content of PC onto the stack
$PC \leftarrow$ effective address	Transfer control to the subroutine

If another subroutine is called by the current subroutine, the new return address is pushed into the stack, and so on. The instruction that returns from the last subroutine is implemented by the microoperations:

$PC \leftarrow M[SP]$ Pop stack and transfer to PC

$SP \leftarrow SP + 1$ Increment stack pointer

Recursive Sub Routine:-

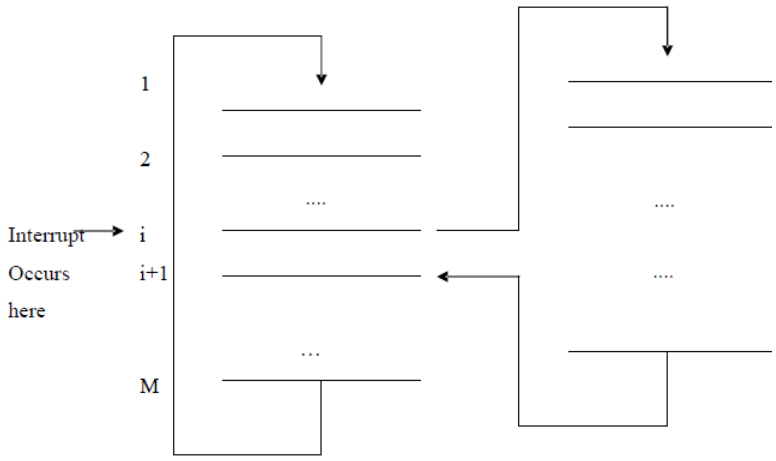
A recursive subroutine is a subroutine that calls itself. If only one register or memory location is used to store the return address, and the recursive subroutine calls itself, it destroys the previous return address. This is undesirable because vital information is destroyed. This problem can be solved if different storage locations are employed for each use of the subroutine while another lighter-level use is still active. When a stack is used, each return address can be pushed into the stack without destroying any previous values. This solves the problem of recursive subroutines because the next subroutine to exit is always the last subroutine that was called.

Program Interrupt:-

Program interrupt refers to the transfer of program control **from a currently running program to another service program as a result of an external or internal generated request**. Control returns to the original program after the service program is executed.

The **interrupt cycle is a hardware implementation of a branch** and save return address operation. The return address available in PC is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted.

The routine executed in response to an interrupt request is called the **interrupt- service routine**.



Transfer of Control through the use of interrupts

The processor first completes execution of instruction i. Then, it loads the program counter with the address of the first instruction of the interrupt-service routine. After execution of the interrupt-service routine, the processor has to come back to instruction i +1.

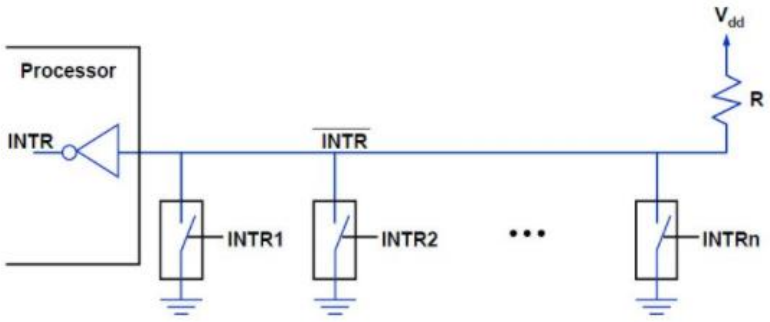
We should note that as part of handling interrupts, the processor must inform the device that its request has been recognized so that it may remove its **interrupt-request signal**. This may be accomplished by means of a special control signal on the bus. An **interrupt-acknowledge signal**. The execution of an instruction in the interrupt-service routine that accesses a status or data register in the device interface implicitly informs that device that its interrupt request has been recognized.

Note:- A subroutine performs a function required by the program from which it is called.

Saving registers also increases the delay between the time an interrupt request is received and the start of execution of the interrupt-service routine. This delay is called **Interrupt Latency.**

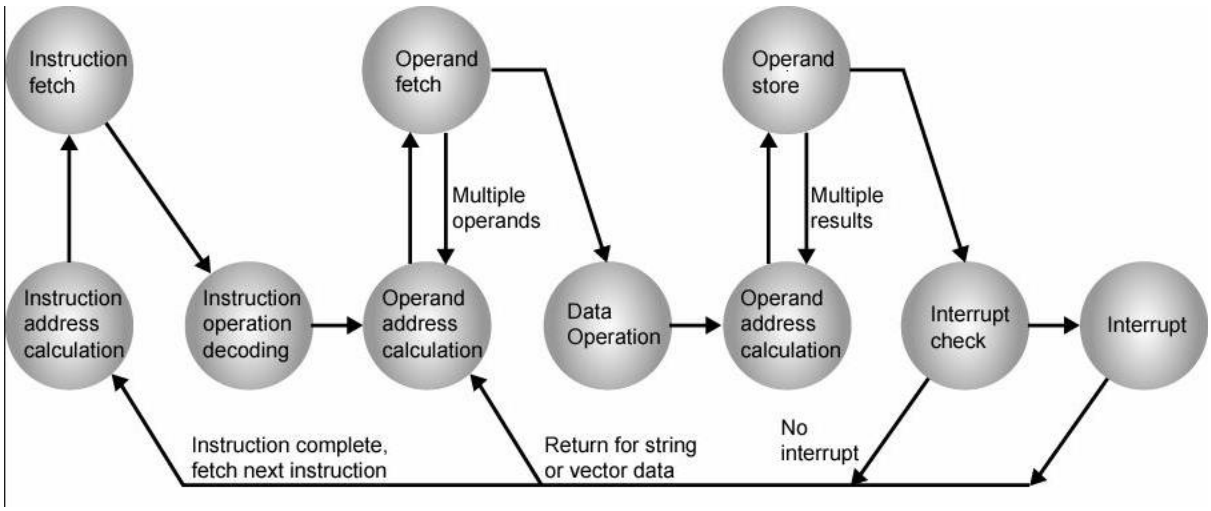
Interrupt Hardware:-

I/O device requests an interrupt by activating a bus line called **interrupt-request**. Most computers are likely to have several I/O devices that can request an interrupt. A single **interrupt-request line** may be used to serve n devices. All devices are connected to the line via switches to ground. **To request an interrupt, a device closes its associated switch**.



INTR=INTR1+INTR2+…+INTRn

Open-drain bus used to implement a common interrupt-request line



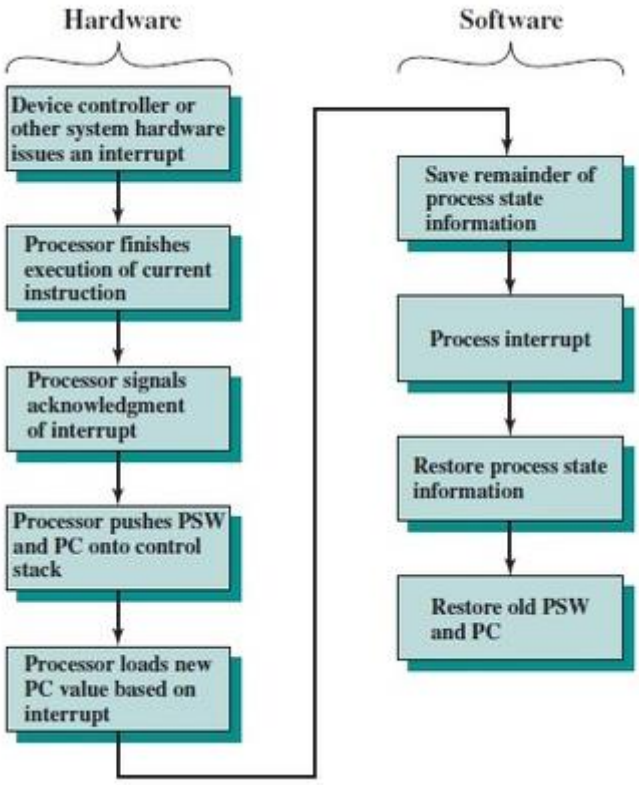
Difference Interrupts and Exceptions

- **Exceptions** are caused by software executing instructions.
  Eg: a page fault, or an attempted write to read only page. An expected exception is **'trap'**, unexpected is a **"fault"**.
- **Interrupts** are caused by hardware devices
  Eg: device finishes I/O, timer fires.

How interrupts are handled?

Different routines handle different interrupts – called Interrupt Service Routines (ISR). When CPU is interrupted it stops what it is doing. A generic routine called **Interrupt Handling Routine (IHR)** is run which examines the nature of interrupt and calls the corresponding **Interrupt Service Routine (ISR)** stored in the lower part of memory.

After servicing the interrupt, the saved address is loaded again to PC to resume the process again.



Simple Interrupt Processing

Enabling and Disabling Interrupts

There are many situations in which the processor should ignore interrupt requests. A particular sequence of instructions is executed without the interrupt because the intercept-service routine may change some of the data used by the instructions.

For enabling and disabling interrupts must be available to the programmer. A simple way is to provide machine instructions such as Interrupt-enable and Interrupt-disable.

When a device activates the interrupt-request signal, it keeps this signal activated until it learns that the processor has accepted its request. This means that the interrupt- request signal will be active during execution of the interrupt-service routine. **It is essential to ensure that this active request signal doesn't lead to successive interruptions.**

The following sequence of events involved in handling an interrupt request from a single device.

1. The device raises an interrupt request.
2. The processor interrupts the program currently being executed.
3. Interrupts are **disabled by changing the control bits** in the PS (except in the case of edge-triggered interrupts).
4. The device is informed by **sending INTA signal to peripheral** that its request has been recognized, and in response, it deactivates the interrupt-request signal.
5. The action requested by the interrupt is performed by the interrupt-service routine.
6. Interrupts are enabled and execution of the interrupted program is resumed.

Handling Multiple Devices-

Let us now consider the situation where a number of devices capable of initiating interrupts are connected to the processor. Devices are operationally independent, there is no definite order in which they will generate interrupts.

1. How can the processor recognize the device requesting an interrupts?
2. Given that different devices are likely to require different interrupt-service routines, how can the processor obtain the starting address of the appropriate routine in each case?
3. Should a device be allowed to interrupt the processor while another interrupt is being serviced?
4. How should two or more simultaneous interrupt requests be handled?

If two devices have activated the line at the same time, it must be possible to break the tie and select one of the two requests for service.

Note: - When a device raises an interrupt request, it sets to 1 one of the bits in its status register.

Vectored Interrupts:-

To reduce the time involved in the polling process**, a device requesting an interrupt may identify itself directly to the processor. Then, the processor can immediately start executing the corresponding interrupt-service routine**. The term vectored interrupts refers to all interrupt-handling schemes based on this approach.

In a **non-vectored interrupt**, **the branch address is assigned to a fixed location in memory**. In a **vectored interrupt**, **the source that interrupts supplies the branch information to the computer**. **This information is called the interrupt vector**.

Types of Interrupts:

1. External interrupts
2. Internal Interrupts
3. Software Interrupts

External interrupts (Asynchronous) come from input-output (l/0) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source. Examples that cause external interrupts are l/0 device requesting transfer of data, l/0 device finished transfer of data, elapsed time of an event, or power failure. Timeout interrupt may result from a program that is in an endless loop and thus exceeded its time allocation.

The 8085 has five hardware interrupts

(1) TRAP (2) RST 7.5 (3) RST 6.5 (4) RST 5.5 (5) INTR

Internal interrupts (Synchronous) arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called **traps**. Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation. The difference between internal and external interrupts is that the internal interrupt is initiated by some exceptional condition caused by the program itself rather than by an external event.

Note:-

1. **If the program is rerun, the internal interrupts will occur in the same place each time.**
2. **External/internal interrupts are initiated from signals that occur in the H/W of the CPU**

A software interrupt is initiated by executing an instruction. Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program.

- The software interrupts are program instructions. These instructions are inserted at desired locations in a program.
- The 8085 has eight software interrupts from RST0 to RST7. The vector address for these interrupts can be calculated as follows.
- Interrupt number * 8 = vector address For RST 5,5 * 8 = 40 = 28H
- Vector address for interrupt RST5 is 0028H

The Table shows the vector addresses of all interrupts.

Interrupt	Vector address
RST 0 RST 1	0000_H 0008_H
RST 2 RST 3	0010_H 0018_H
RST 4 RST 5	0020_H 0028_H
RST 6 RST 7	0030_H 0038_H

Instruction Cycle

A program residing in the memory unit of the computer consists of a sequence of instructions. The program is executed in the computer by going through a cycle for each instruction. Each instruction cycle in turn is subdivided into a sequence of subcycles or phases. In the basic computer each instruction cycle consists of the following phases:

1. Fetch an instruction from memory.
2. Decode the instruction.
3. Read the effective address from memory if the instruction has an indirect address.
4. Execute the instruction.

Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction. This process continues indefinitely unless a HALT instruction is encountered.

Fetch and Decode:

Initially, the program counter PC is loaded with the address of the first instruction in the program. Timing signals go through a sequence T0, T1, T2, and so on.

T_0: $AR \leftarrow PC$
T_1: $IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
T_2: $D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{–}14)$, $AR \leftarrow IR(0\text{–}11)$, $I \leftarrow IR(15)$

Program Status Word:

PSW is the collection of all status bit conditions in the CPU. The PSW is stored in a separate **hardware register.** Typically, it includes the status bits from **the last ALU operation**.

Status Bit Conditions:

1. Bit C (carry) is set to 1 of the end carry C8 is 1. It is cleared to 0 if the carry is 0.
2. Bit S (sign) is set to 1 if the highest-order bit F7 is 1. It is set to 0 if the bit is 0.
3. Bit Z (zero) is set to 1 if the output of the ALU contains all 0's. it's cleared to 0 else.
4. Bit V (overflow) is set to 1 if the EX-OR of the last two carries is equal to 1.
5. Bit Au (auxiliary) is set to 1 if there is a carry from lower nibble to higher nibble in BCD.

Microprogrammed Control

Control Memory:
The function of the **control unit** in a digital computer is to initiate sequences of micro-operations. During any given time, certain micro-operations are to be initiated, while others remain idle. The control variables at any given time can be represented by a string of 1's and 0's called a **control word**. As such, control words can be programmed to perform various operations on the components of the system. A control unit whose binary control variables are stored in memory is called a **microprogrammed control unit**. Control memory can be a **read-only memory (ROM)**.

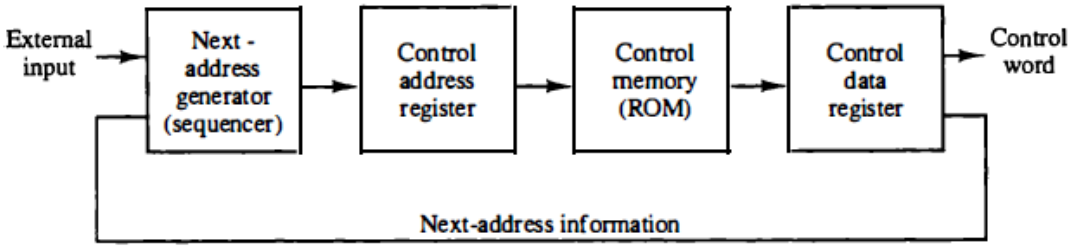
Microinstruction (Control word)
Each word in control memory contains within it a **microinstruction**. The microinstruction specifies one or more microoperations for the system.

Microprogram
A sequence of microinstructions constitutes a **microprogram**.

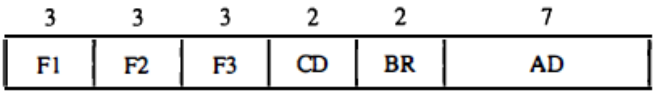
Note: - **A memory that is part of a control unit is referred to as a control memory.**

A computer that employs a microprogrammed control unit will have two separate memories: **a main memory** and **a control memory**. The main memory is available to the user for storing the programs. The user's program in main memory consists of machine instructions and data. In contrast, the control memory holds a fixed microprogram that cannot be altered by the occasional user. **The microprogram consists of microinstructions that specify various internal control signals** for execution of register microoperations.

Figure 7-1 Microprogrammed control organization.



Control address register
The control memory is a ROM, within which all control information is permanently stored. The **control memory address register (CAR)** specifies the address of the microinstruction, and the **control data register (CDR)** holds the microinstruction read from memory. The microinstruction contains a control word that specifies one or more microoperations for the data processor. Once these operations are executed, the control must determine the next address. The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in The control memory.



Microinstruction code format

F1, F2, F3 micro-operations fields, CD: Condition for branching, BR: Branch Field
AD: Address field.

Basic Processing Unit (from Hamacher)

Fundamental Concepts:-

Execution of a program by the processor starts with the fetching of instructions one at a time, decoding the instruction and performing the operations specified. From memory, instructions are fetched from successive locations until a branch or a jump. The processor keeps track of the address of the memory location containing the next instruction to be fetched using the **program counter**. After fetching an instruction, the contents of the PC are updated to point to the next instruction in the sequence.

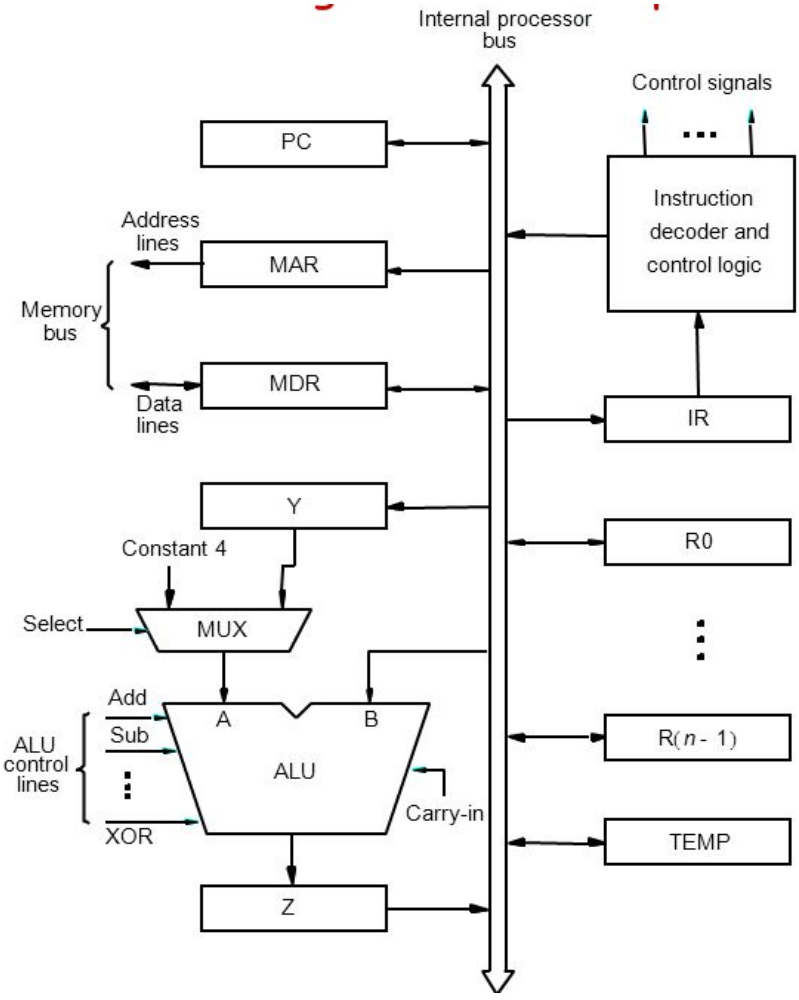
Another key register in the processor is **Instruction Register**. Suppose that each instruction comprises 4 bytes, and that it is stored in one memory word. To execute an instruction, the processor has to perform the following three steps:

1. Fetch the contents of the memory location pointed to by the PC, and loaded into IR.

   IR → [(PC)]
2. Assuming that the memory is byte addressable, increment the contents of the PC by 4

   PC ← [PC] +4
3. Decode the instruction to understand the operation & generate the control signals necessary to carry out the operation.
4. Carry out the actions specified by the instruction in the IR

Note: - In cases where an instruction occupies more than one word, steps 1 and 2 must be repeated as many times as necessary to fetch the complete instruction.



Datapath in which the arithmetic and logic unit (ALU) and all the registers are interconnected through a single common bus, which **is internal to the processor** and should not be confused with the **external bus** that connects the processor to the memory and I/O devices.
The **data** and **address lines** of the external memory bus are shown above connected to the internal processor bus via the **memory data register**, MDR, and the **memory address register**, MAR, respectively.
Register MDR has two inputs and two outputs. Data may be loaded into MDR either from the memory bus or from the internal processor bus. The data stored in MDR may be placed on either bus. The input of MAR is connected to the internal bus, and its output is connected to the external bus.
The control lines of the memory bus are connected to the instruction decoder and control logic block.
Registers from R0 to Rn-1 may be provided for general-purpose use by the programmer. Some may be dedicated as special-purpose registers, such as **index registers** or **stack pointers**.

The multiplexer **MUX** selects either the output of register Y or a constant value 4 to be provided as input A of the ALU. The constant 4 is used to increment the contents of the program counter.
The instruction **decoder and control logic** unit is responsible for implementing the actions specified by the instruction loaded in the IR register. The decoder generates the control signals needed to select the registers involved and direct the transfer of data.

The registers, the ALU, and the interconnecting bus are collectively referred to as the data path.

An instruction can be executed by performing one or more of the following operations in some specified sequence:

1. Transfer a word of data from one processor register to another or to the ALU.
2. Perform an arithmetic or a logic operation and store the result in a processor register.
3. Fetch the contents of a given memory location and load them into a processor register.
4. Store a word of data from a processor register into a given memory location

Instruction execution involves a sequence of steps in which data are transferred from one register to another. For each register, **two control signals are used to place the contents** of that register on the bus or to load the data on the bus into the register.

> The input and output of register Ri are connected to the bus via switches controlled by the signals Ri_{in} and Ri_{out} respectively. When Ri_{in} is set to 1, the data on the bus are loaded into Ri. Similarly, when Ri_{out}, is set to 1, the contents of register Ri_{out} are placed on the bus. While Ri_{out} is equal to 0, the bus can be used for transferring data from other registers.

Suppose that we wish to transfer the contents of register R1 to register R4. This can be accomplished as follows:

1. Enable the output of register R1 out by setting R1out to 1. This places the contents of R1 on the processor bus.
2. Enable the input of register R4 by setting R4in to 1. This loads data from the processor bus into register R4.

Fetching a word from Memory:-
To fetch a word of information from memory, the processor has to specify the address of the memory location where this information is stored and request a read operation.
The processor transfers the required address to the MAR, whose output is connected to the address lines of the memory bus. At the same time the processor uses the control lines of the memory bus to indicate that a read operation is needed, and data is stored in MDR.

Pipelining

[Read pipeline from Hamacher's Book]
[Notes from Moris Mano]

Pipelining

Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments.

The overlapping of computation is made possible by associating a register with each segment in the pipeline. The registers provide isolation between each segment so that each can operate on distinct data simultaneously.

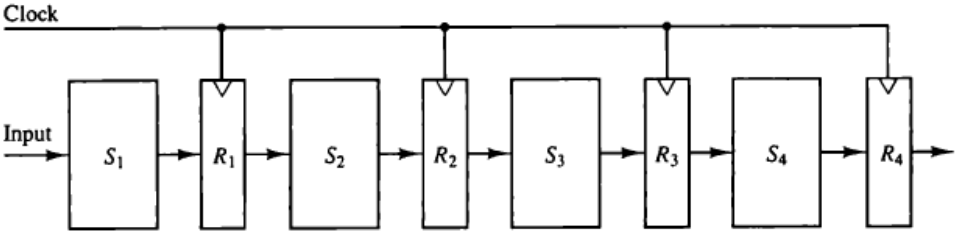


Figure 9-3 Four-segment pipeline.

Suppose there are 6 tasks T1 through T6 executed in four segments. Initially, task T1 is handled by segment 1. After the first clock, segment 2 is busy with T1 while segment 1 is busy with task T2 Continuing in this manner, the first task T1 is completed after the fourth clock cycle. From then on, the pipe completes a task every clock cycle. No matter how many segments there are in the system, **once the pipeline is full, it takes only one clock period to obtain an output**.

Consider a **k-segment pipeline** with a clock **cycle time tp**, is used to execute n tasks. The first task T1 requires a time equal to **k*tp**, to complete its operation since there are k segments in the pipe. The remaining n - 1 tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal **to (n - 1)*tp**.

Note: - To complete n tasks using a k-segment pipeline requires **k + (n - 1) clock cycles**.

Next consider a **nonpipeline** unit that performs the same operation and takes a time equal to tn to complete each task. **The total time required for n tasks is n*tn**.

a. **The speedup of a pipeline processing over an equivalent non-pipeline processing** is defined by the ratio:

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

Figure 9-4 Space-time diagram for pipeline.

Segment:	1	2	3	4	5	6	7	8	9	→ Clock cycles
1	T_1	T_2	T_3	T_4	T_5	T_6				
2		T_1	T_2	T_3	T_4	T_5	T_6			
3			T_1	T_2	T_3	T_4	T_5	T_6		
4				T_1	T_2	T_3	T_4	T_5	T_6	

b. As the number of tasks increases, **n becomes much larger than k - 1**, and k + n - 1 approaches the value of n. Under this condition, the speedup becomes

$$S = \frac{t_n}{t_p}$$

c. If we assume that the time it takes to process a task is the same in the pipeline and non-pipeline circuits, we will have **tn = k*tp**

$$S = \frac{kt_p}{t_p} = k$$

Note: - Maximum speedup that a pipeline can provide is k, where k is the number of segments in the pipeline.

Example: Let the time it takes to process a sub-operation in each segment be equal to tp = 20 ns. Assume that the pipeline has k = 4 segments and executes n = 100 tasks in sequence.
Solution:-
tn = k*tp = 4*20=80ns,

S = 80*100/(4+100-1)*20 = 8000/2060 = 3.88

As the number of tasks increases, the speedup will approach 4, which is equal to the number of segments in the pipeline.

Why pipeline cannot operate at its maximum theoretical rate?

a. Different segments may take different times to complete their sub-operation.
b. The clock cycle must be chosen to equal the time delay of the segment with the maximum propagation time. This causes all other segments to waste time while waiting for the next clock.
c. Intermediate registers will not be needed in a non-pipeline processor.

There are two areas of computer design where the pipeline organization is applicable.

a. An **arithmetic pipeline** divides an arithmetic operation into sub-operations for execution in the pipeline segments.
b. An **instruction pipeline** operates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle.

Arithmetic Pipeline Example:

the floating-point pipeline are implemented with combinational circuits. Suppose that the time delays of the four segments are $t_1 = 60$ ns, $t_2 = 70$ ns, $t_3 = 100$ ns, $t_4 = 80$ ns, and the interface registers have a delay of $t_r = 10$ ns. The clock cycle is chosen to be $t_p = t_3 + t_r = 110$ ns. An equivalent nonpipeline floating-point adder-subtractor will have a delay time $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 320$ ns. In this case the pipelined adder has a speedup of 320/110 = 2.9 over the nonpipelined adder.

Note: Here one register delay is added into non-pipeline total delay.

Instruction Pipeline

Pipeline processing can occur not only in the data stream but in the instruction stream as well.
An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and execute phases to overlap and perform simultaneous operations.
An instruction may cause a branch out of sequence. In that case the pipeline must be emptied & all the instructions that have been read from memory after the branch instruction must be discarded.

In the most general case, the computer needs to process each instruction with the following sequence of steps

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

Note: instruction pipeline will be most efficient if the instruction cycle is divided into segments of equal duration.

Consider a four stage pipeline as follows:

1. FI is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction:	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	–	–	FI	DA	FO	EX			
	5					–	–	–	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

Figure 9-8 Timing of instruction pipeline.

Note:-
It is assumed that the processor has separate instruction and data memories so that the operation in FI and FO can proceed at the same time.
In step 4, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched in segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FI.

Assume now that **instruction 3 is a branch instruction**. As soon as this instruction is decoded in segment DA in step 4. The transfer from FI to DA of the other instructions is halted until the branch instruction is executed in step 6. If the branch is taken, a new instruction is fetched in step 7.

If the branch is not taken, the instruction fetched previously in step 4 can be used. The pipeline then continues until a new branch instruction is encountered.

Pipeline conflicts

In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

1. **Resource conflicts** caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.
2. **Data dependency conflicts** arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
3. **Branch difficulties** arise from branch and other instructions that change the value of PC.

Data Dependency

A data dependency occurs when an instruction needs data that are not yet available.
Pipelined computers deal with such conflicts between data dependencies in a variety of ways.

a. **hardware interlocks**
   The most straightforward method is to insert **hardware interlocks**. An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline. Detection of this situation causes the instruction whose source is not available **to be delayed by enough clock cycles to resolve the conflict**. **This approach maintains the program sequence**.

b. **Operand forwarding** it uses special hardware to detect a conflict and then avoid it **by routing the data through special paths between pipeline segments**.

c. **Delayed load/ No operation** a procedure employed in some computers is to give the responsibility for solving data conflicts problems **to the compiler** that translates the high-level programming language into a machine language program. The compiler for such computers is designed **to detect a data conflict and reorder the instructions** as necessary to delay the loading of the conflicting data by inserting **no-operation** instructions. This method is referred to as **delayed load**.

Handling of Branch Instructions

The branch instruction breaks the normal sequence of the instruction stream.

a. **Pre-fetch target instruction** pre-fetch the target instruction in addition to the instruction following the branch. Both are saved until the branch is executed. If the branch condition is successful, the pipeline continues from the branch target instruction.

b. **Branch target buffer** the BTB is an associative memory included in the fetch segment of the pipeline. Each entry in the BTB consists of the address of a previously executed branch instruction and the target instruction for that branch it also stores the next few instructions after the branch target instruction. When the pipeline decodes a branch instruction, it searches the associative memory BTB for the address of the instruction. The advantage of this scheme is that branch instructions that have occurred previously are readily available in the pipeline without interruption.

c. **Loop buffer** a variation of the BTB is the loop buffer. This is a small very high speed register file maintained by the instruction fetch segment of the pipeline. When a program loop is detected in the program, it is stored in the loop buffer in its entirety, including all branches.

d. **Branch prediction** a pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed. The pipeline then begins prefetching the instruction stream from the predicted path. A correct prediction eliminates the wasted time caused by branch penalties**.**

e. **Delayed branch** a procedure employed in most RISC processors is the **delayed branch**. The compiler detects the branch instructions and rearranges the machine language code sequence by inserting useful instructions that keep the pipeline operating without interruptions.

Example: Three-Segment Instruction Pipeline
I: Instruction fetch (fetches the instruction from program memory)
A: ALU operation (decodes and performs ALU operation)
E: Execute instruction (E segment directs the output of the ALU to destination)

Delayed Load
Consider now the operation of the following four instructions:

1. LOAD: $R1 \leftarrow M$[address 1]
2. LOAD: $R2 \leftarrow M$[address 2]
3. ADD: $R3 \leftarrow R1 + R2$
4. STORE: M[address 3] $\leftarrow R3$

Clock cycles:	1	2	3	4	5	6
1. Load $R1$	I	A	E			
2. Load $R2$		I	A	E		
3. Add $R1 + R2$			I	A	E	
4. Store $R3$				I	A	E

(a) Pipeline timing with data conflict

There will be a data conflict in instruction 3 because the operand in R2 is not yet available in the A segment. The E segment in clock cycle 4 is in a process of placing the memory data into R2. The A segment in clock cycle 4 is using the data from R2, but the value in R2 will not be the correct value since it has not yet been transferred from memory.
It is up to the compiler to make sure that the instruction following the load instruction uses the data fetched from memory. **If the compiler cannot find a useful instruction to put after the load, it inserts a no-op (no-operation) instruction**.
This is a type of instruction that is fetched from memory but has no operation, thus wasting a clock cycle. This concept of delaying the use of the data loaded from memory is referred to as **delayed load**.

Clock cycle:	1	2	3	4	5	6	7
1. Load *R*1	I	A	E				
2. Load *R*2		I	A	E			
3. No-operation			I	A	E		
4. Add *R*1 + *R*2				I	A	E	
5. Store *R*3					I	A	E

(b) Pipeline timing with delayed load

Figure (b) shows the same program with a no-op instruction inserted after the load to R2 instruction. The data is loaded into R2 in clock cycle 4. The add instruction uses the value of R2 in step 5. Thus the no-op instruction is used to advance one clock cycle in order to compensate for the data conflict.

Advantage: the delayed load approach is that the data dependency is **taken care of by the compiler** rather than the hardware.

Delayed Branch

The method used in most RISC processors is to **rely on the compiler** to redefine the branches so that they take effect at the proper time in the pipeline. This method is referred to as **delayed branch**. The effect is the same as if the instructions were executed in their original order, except that the branch delay is removed. **It is up to the compiler to find useful instructions to put after the branch instruction**. Failing that, the compiler can insert **no-op** instructions.

The program for this example consists of five instructions:

Load from memory to *R*1
Increment *R*2
Add *R*3 to *R*4
Subtract *R*5 from *R*6
Branch to address *X*

Clock cycles:	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to *X*					I	A	E			
6. No-operation						I	A	E		
7. No-operation							I	A	E	
8. Instruction in *X*								I	A	E

(a) Using no-operation instructions

In Figure (a) the compiler inserts two no-op instructions after the branch. The branch address X is transferred to PC in clock cycle 7. The fetching of the instruction at X is delayed by two clock cycles by the no-op instructions. The instruction at X starts the fetch phase at clock cycle 8 after the program counter PC has been updated.

Clock cycles:	1	2	3	4	5	6	7	8
1. Load	I	A	E					
2. Increment		I	A	E				
3. Branch to *X*			I	A	E			
4. Add				I	A	E		
5. Subtract					I	A	E	
6. Instruction in *X*						I	A	E

(b) Rearranging the instructions

The program in Fig (b) is rearranged by placing the add and subtract instructions after the branch instruction instead of before as in the original program.
Inspection of the pipeline timing shows that PC is updated to the value of X in clock cycle 5, but the add and subtract instructions are fetched from memory and executed in the proper sequence.

Cache Memory (William Stallings)

Types of Memory: Based on the methods of accessing.

1. **Sequential access** memory is organized into units of data, called records. Access must be made in a specific linear sequence. A shared read - write mechanism is used, and this must be moved from its current location to the desired location, passing and rejecting each intermediate record. **E.g. Magnetic tape**

2. **Direct access** it involves a shared read-write mechanism. Individual blocks or records have a unique address based on physical location. Access is accomplished by direct access to reach a general sector plus sequential searching, counting, or waiting to reach the final location. **E.g. Compact disks**

3. **Random Access** each addressable location in memory has a unique, physically wired-in addressing mechanism. The time to access a given location is independent of the sequence of prior access and is **constant**. Thus, any location can be selected at random and **directly addressed and accessed**. E.g. RAM and some cache memories

4. **Associative:** This is a random access type of memory that enables one to make a comparison of desired bit locations within a word for a specified match, and to do this for all words **simultaneously**. Thus, **a word is retrieved based on a portion of its contents rather than its address**. Retrieval time is constant. E.g. TLB

Access time (latency):
For **random-access memory**, this is the time it takes to perform a read or write operation, that is, the time from the instant that an address is presented to the memory to the instant that data have been stored or made available for use. For **non-random-access memory**, access time is the time it takes to position the read–write mechanism at the desired location.

Memory cycle time
This concept is primarily applied to random-access memory and consists of the access time plus any additional time required before a second access can commence. **Note that memory cycle time is concerned with the system bus, not the processor**.

Transfer rate:
This is the rate at which data can be transferred into or out of a memory unit. For **random-access memory**, it is equal to **1/(cycle time)**.

Example: - Suppose that the processor has access to two levels of memory. Level 1 contains 1000 words and has an access time of 0.01 microsec; level 2 contains 100,000 words and has an access time of 0.1 microsec. Assume that if a word to be accessed is in level 1, then the processor accesses it directly. If it is in level 2, then the word is first transferred to level 1 and then accessed by the processor. For simplicity, we ignore the time required for the processor to determine whether the word is in level 1 or level 2. Suppose 95% of the memory accesses are found in the cache. Then The average time to access a word can be expressed as

Solution: $(0.95)(0.01\ \mu s) + (0.05)(0.01\ \mu s + 0.1\ \mu s) = 0.0095 + 0.0055 = 0.015\ \mu s$

Cache Memory Principles

The cache contains a copy of portions of main memory. When the processor attempts to read a word of memory, a check is made to determine if the word is in the cache. If so, the word is delivered to the processor. If not, a block of main memory, consisting of some fixed number of words, is read into the cache and then the word is delivered to the processor.

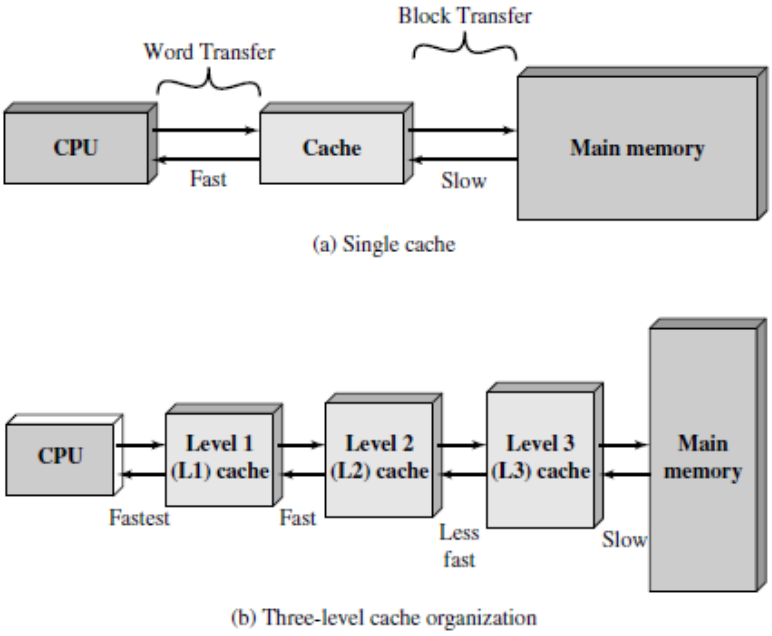


Figure 4.3 Cache and Main Memory

Main memory consists of up to 2^n addressable words, with each word having a unique n-bit address. For mapping purposes, this memory is considered to consist of a number of **fixed length blocks of K words each**. That is, there are **M= 2^n/K blocks** in main memory.

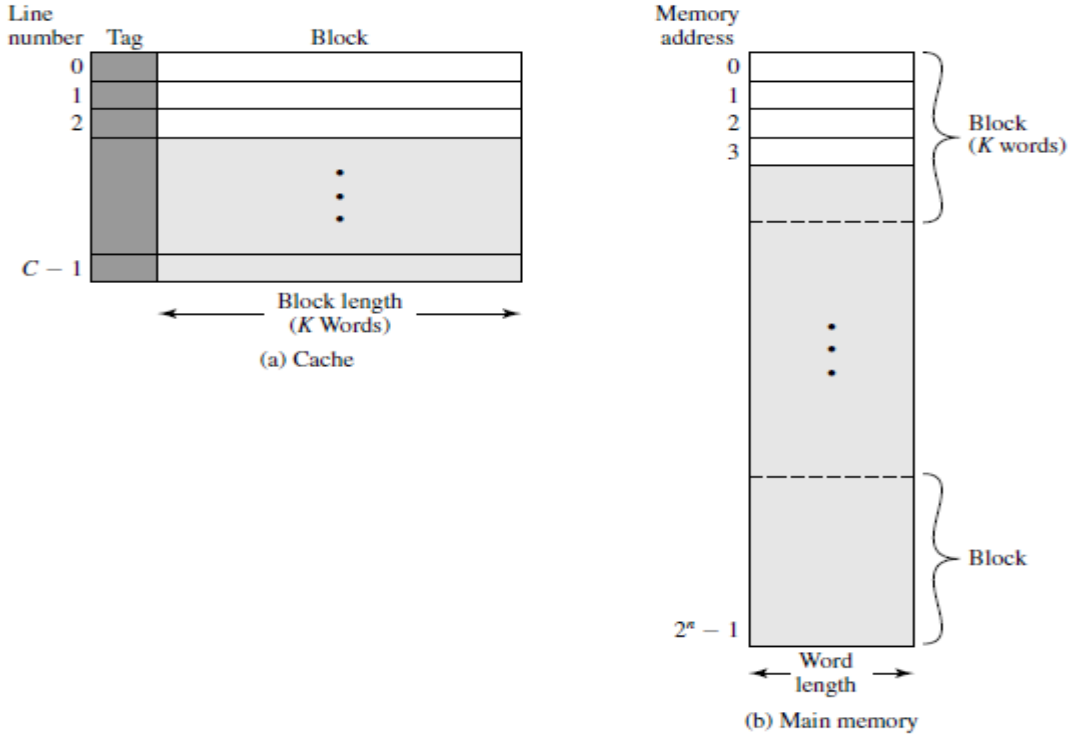


Figure 4.4 Cache/Main Memory Structure

The cache consists of m blocks, called **lines**. Each line contains K words, plus a tag of a few bits. **Each line also includes control bits** (not shown) such as to indicate as whether the line has been modified since being loaded into the cache.
Note: - The length of a line, not including tag and control bits, is the **line size.** The line size may be as small as 32 bits, with each "word" being a single byte; in this case the line size is **4 bytes**. Because there are more memory blocks than lines, an individual line cannot be uniquely and permanently dedicated to a particular block. Thus, each line includes a tag that identifies which particular block is currently being stored. The tag is usually a portion of the main memory address.

The processor generates the read address (RA) of a word to be read. If the word is contained in the cache, it is delivered to the processor. Otherwise, the block containing that word is loaded into the cache, and the word is delivered to the processor.

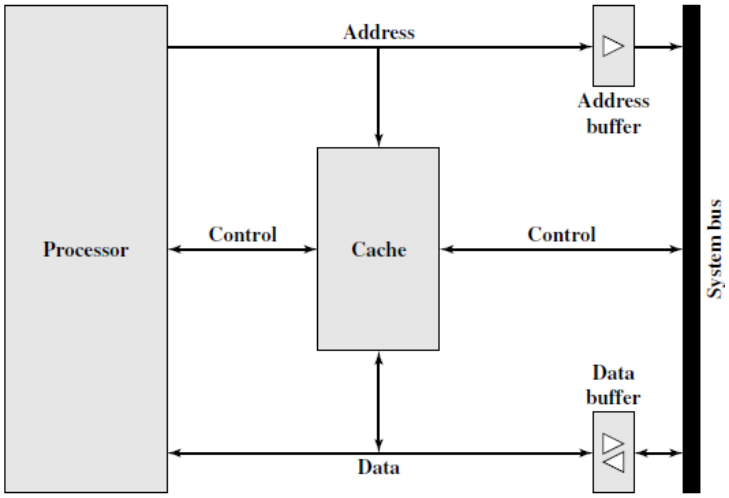


Figure 4.6 Typical Cache Organization

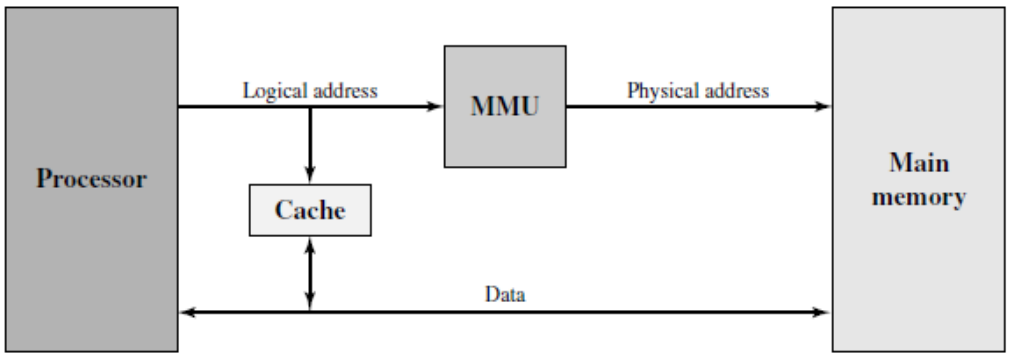
In this organization, the cache connects to the processor via data, control, and address lines. The data and address lines also attach to data and address buffers, which attach to a system bus from which main memory is reached. **When a cache hit occurs, the data and address buffers are disabled and communication is only between processor and cache**, with no system bus traffic. When a cache miss occurs, the desired address is loaded onto the system bus and the data are returned through the data buffer to both the cache and the processor.
In other organizations, the cache is physically interposed between the processor and the main memory for all data, address, and control lines. In this latter case, for a cache miss, **the desired word is first read into the cache and then transferred from cache to processor**.

Memory Hierarchy:-
A typical hierarchy from top to bottom will be:

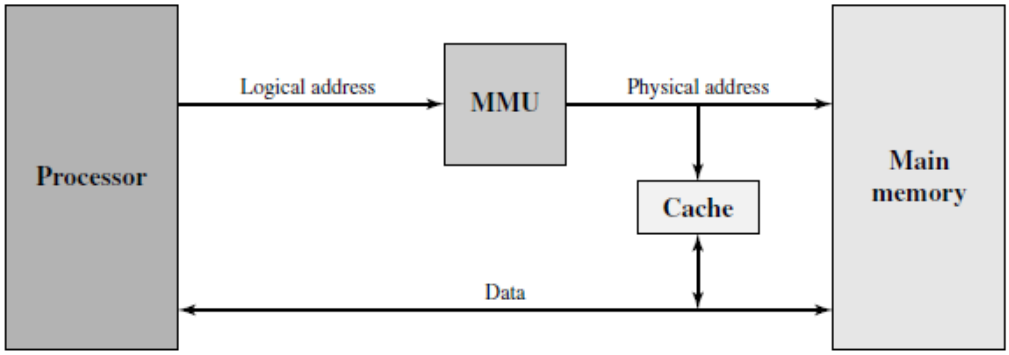
a. Decreasing cost per bit
b. Increasing capacity
c. Increasing access time
d. Decreasing frequency of access of the memory by the processor

Cache Addresses

When virtual addresses are used, the system designer ma y choose to **place the cache** between the **processor and the MMU** or between the **MMU and main memory**. A logical cache, also known as a **virtual cache**, stores data using virtual addresses. The processor accesses the cache directly, without going through the MMU. A **physical cache** stores data using main memory physical addresses.



(a) Logical cache



(b) Physical cache

Figure 4.7 Logical and Physical Caches

Advantage of the logical cache is that cache access speed is faster than for a physical cache. The **disadvantage** has to do with the fact that most virtual memory systems supply each application with the same virtual memory address space. That is, each application sees a virtual memory that starts at address 0. Thus, the same virtual address in two different applications refers to two different physical addresses. The cache memory must therefore be completely **flushed** with each application context switch, or extra bits must be added to each line of the cache to identify which virtual address space this address refers to.

Mapping Function

Direct Mapping:-

It maps each block of main memory into only one possible cache line. Where

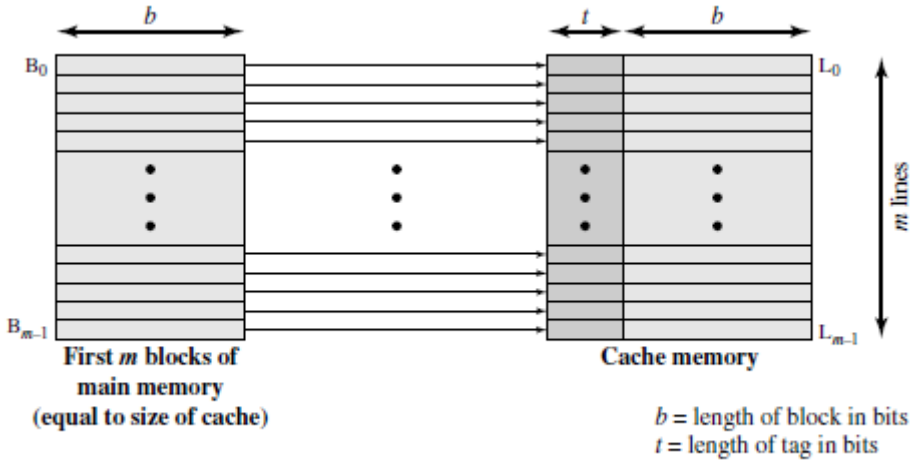
$$i = j \text{ modulo } m$$

where

i = cache line number
j = main memory block number
m = number of lines in the cache

Each main memory address can be viewed as consisting of **three fields**. The least significant **w bits** identify a unique word or byte within a block of main memory. The remaining s bits specify one of the 2^s blocks of main memory. The cache logic interprets these s bits as a tag of **s-r bits** (most significant portion) and a line field of r bits. This latter field identifies one of the **m= 2^r** lines of the cache.

- Address length = $(s + w)$ bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = $\frac{2^{s+w}}{2^w} = 2^s$
- Number of lines in cache = $m = 2^r$
- Size of cache = 2^{r+w} words or bytes
- Size of tag = $(s - r)$ bits



(a) Direct mapping

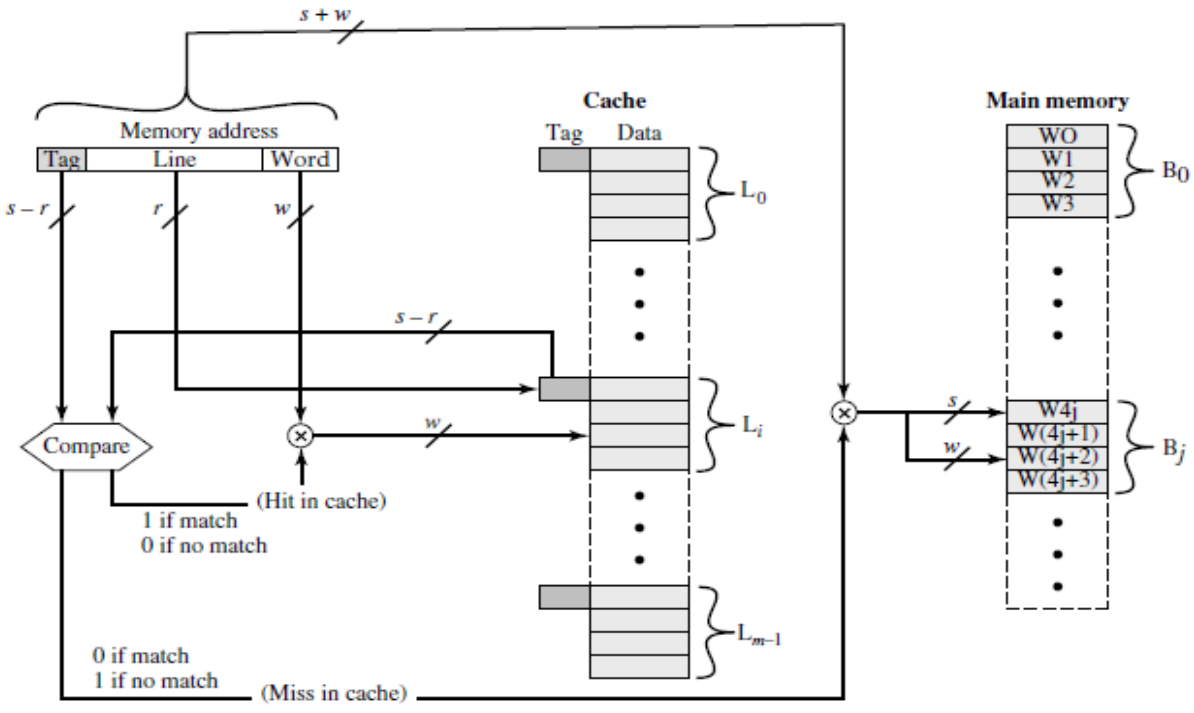


Figure 4.9 Direct-Mapping Cache Organization

Note: - No two blocks that map into the same line number have the same tag number.

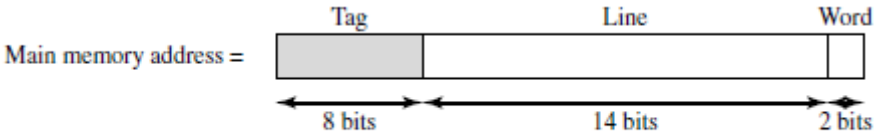


Figure 4.10 Direct Mapping Example

The cache system is presented with a **24-bit** address. The 14-bit line number is used as an index into the cache to access a particular line. If the 8-bit tag number matches the tag number currently stored in that line, then the 2-bit word number is used to select one of the 4 bytes in that line. Otherwise, the 22-bit tag-plus-line field is used to fetch a **block from main memory**.
The direct mapping technique is simple and inexpensive to implement. Its main disadvantage is that there is a fixed cache location for any given block. Thus, if a program happens to reference words repeatedly from two different blocks that map into the same line, then the blocks will be continually swapped in the cache, and the hit ratio will be low (a phenomenon known as **thrashing**).

Note:- We need only one comparator of size equals to tag bits.

Associative Mapping-

Associative mapping overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded **into any line of the cache**. In this case, **the cache control logic interprets a memory address simply as a Tag and a Word field**. The Tag field uniquely identifies a block of main memory. To determine whether a block is in the cache, t**he cache control logic must simultaneously examine every line's tag for a match**.

Note: No field in the address corresponds to the line number, so that the number of lines in the cache is not determined by the address format.

- Address length = $(s + w)$ bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = $\frac{2^{s+w}}{2^w} = 2^s$
- Number of lines in cache = undetermined
- Size of tag = s bits

A main memory address consists of a 22-bit tag and a 2-bit byte number. The 22-bit tag must be stored with the 32-bit block (4 Bytes) of data for each line in the cache. Note that it is the leftmost (most Significant) 22 bits of the address that form the tag.

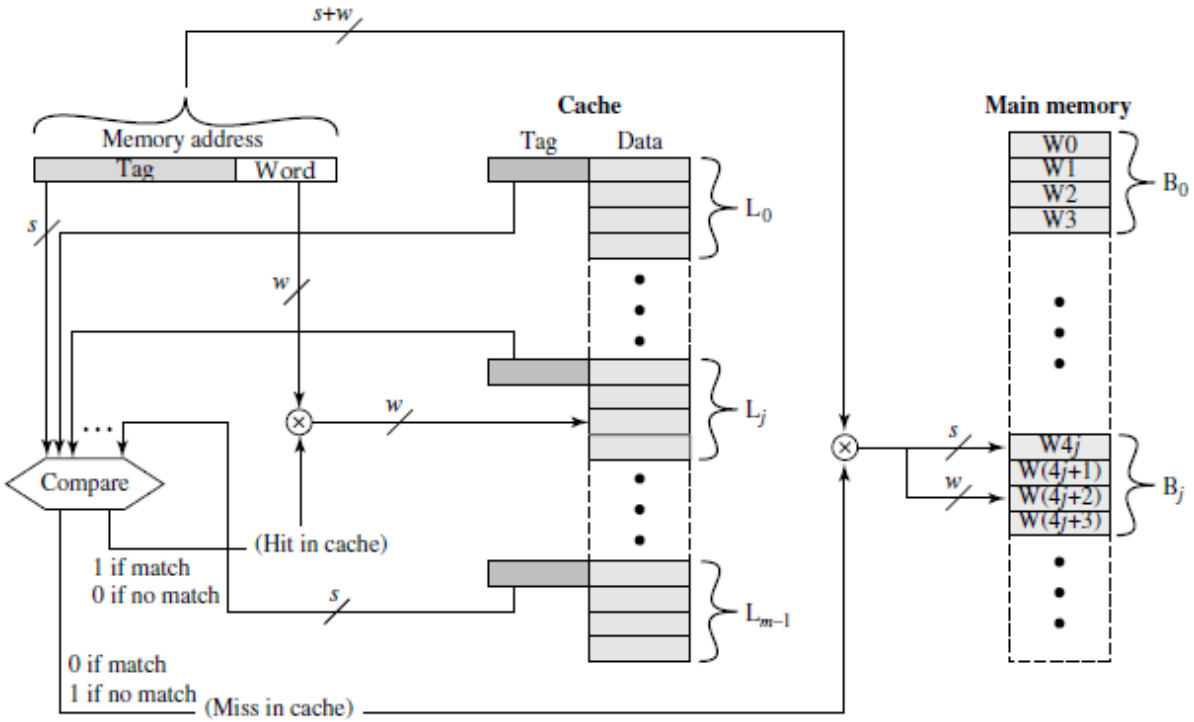


Figure 4.11 Fully Associative Cache Organization

Note-

1. It needs number of comparators equal to the number of cache lines given, each of size equal to the number of tag bits.
2. The principal disadvantage of associative mapping is the complex circuitry required to examine the tags of all cache lines in parallel.

Set Associative Memory:-

Set-associative mapping is a compromise that exhibits the strengths of both the direct and associative approaches while reducing their disadvantages.

In this case, the cache consists of a number sets, **each set which consists of a number of lines**. The relationships are:

$$m = v \times k$$
$$i = j \text{ modulo } v$$

where

i = cache set number
j = main memory block number
m = number of lines in the cache
v = number of sets
k = number of lines in each set

(k – Associative cache memory)

Note: - **With set-associative mapping, block Bj can be mapped into any of the lines of set j**.

As with associative mapping, each word maps into multiple cache lines. **For set-associative mapping, each word maps into all the cache lines in a specific set**, so that main memory block B0 maps into set 0, and so on. Thus, the set-associative cache can be physically implemented as **v associative caches**.

Note:-

a. 1 – way set associative memory = direct mapping cache.
b. n – way set associative memory = full associative memory
c. number of comparators required will be equal to k each of size equal to tag bits
d. kx1 MUX is required to output one tag out of k stored k tags in 1 set.

The cache control logic interprets a memory address as three fields: **Tag**, **Set**, and **Word**.

With fully associative mapping, the tag in a memory address is quite large and must be compared to the tag of every line in the cache. With k-way set-associative mapping, the tag in a memory address is much smaller and is only compared to the k tags within a single set.

- Address length $= (s + w)$ bits
- Number of addressable units $= 2^{s+w}$ words or bytes
- Block size $=$ line size $= 2^w$ words or bytes
- Number of blocks in main memory $= \frac{2^{s+w}}{2^w} = 2^s$
- Number of lines in set $= k$
- Number of sets $= v = 2^d$

The d set bits specify one of **v=2^d sets**. The **s bits of the Tag and Set fields specify one of the 2^s blocks of main memory**.

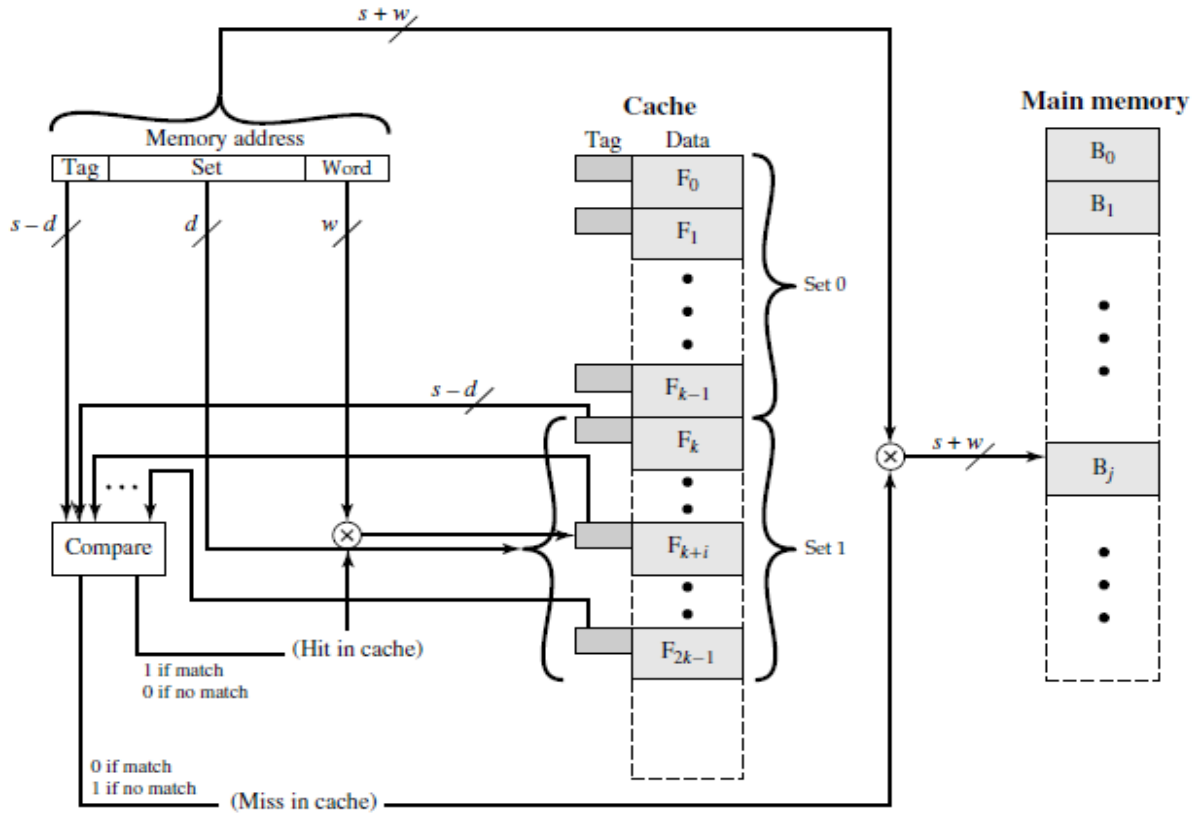
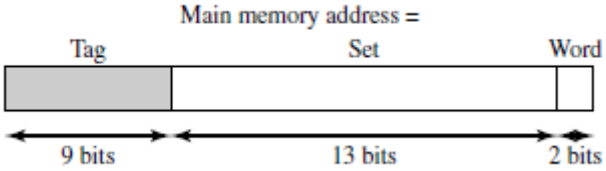


Figure 4.14 *K*-Way Set Associative Cache Organization

- Number of lines in cache $= m = kv = k \times 2^d$
- Size of cache $= k \times 2^{d+w}$ words or bytes
- Size of tag $= (s - d)$ bits

For example, using set-associative mapping with two lines in each set, referred to as two-way set-associative. The 13-bit set number identifies a unique set of two lines within the cache. It also gives the number of the block in main memory, modulo 2^13.



Cache Write Policy

If the old block in the cache has not been altered, then it may be overwritten with a new block without first writing out the old block. If at least one write operation has been performed on a word in that line of the cache, then main memory must be updated by writing the line of cache out to the block of memory before bringing in the new block.

An I/O module may be able to read-write directly to memory. **If a word has been altered only in the Cache, then the corresponding memory word is invalid**. Further, **if the I/O device has altered main memory, then the cache word is invalid**. A more complex problem occurs when multiple processors are attached to the same bus and **each processor has its own local cache**. Then, if a word is altered in one cache, it could conceivably invalidate a word in other caches.

Write through all write operations are made to main memory as well as to the cache, ensuring that main memory is always valid. This technique generates substantial memory traffic and may create a Bottleneck.

Write back `minimizes memory` writes. With write back, updates are made only in the cache. When an update occurs, a **dirty bit**, or **use bit**, associated with the line is set. Then, when a block is replaced it is written back to main memory if and only if the dirty bit is set. The problem with write back is that portions of main memory are invalid, and he**nce accesses by I/O modules can be allowed only through the cache**. This makes for complex circuitry and a potential bottleneck.

How to deal with write misses?

a. **Write Allocate** loads the corresponding block from the next lower level into the cache and then updates the cache block. **Write-allocate tries to exploit spatial locality of writes**, but it has the disadvantage that **every miss results in a block transfer from the next lower level to cache**.
b. **No-write-allocate** bypasses the cache and writes the word directly to the next lower level.

Note:-

1. **Write through caches are typically no-write-allocate**.
2. **Write-back caches are typically write-allocate**.
3. **Caches at lower levels of the memory hierarchy are more likely to use write-back instead of write-through because of the larger transfer times.**

Types of Locality:

1. **Temporal Locality**: - Same word in the same block can be referenced in near future by CPU2.
2. **Spatial Locality**: - Adjacent word in same block can be referenced by CPU in near future.

Cache Coherence:
Same address at different locations have different data.

Cache Line Size: - when a block of data is retrieved and place in the cache not only the desired word but also some number of adjacent words are retrieved.

1. As the block size increases from very small to larger size**, the hit ratio will at first increase because of the principle of locality.** As block size increases more useful data is brought into the cache.
2. **The hit ratio will begin to decrease** however as the block becomes even bigger and the possibility of using the newly fetched info less than the probability of reusing the info that has to be replaced.

Note: - Larger blocks reduce number of blocks that fit into a cache.

Points to be remembered

a. A smaller block size means during a memory fetch only a smaller part of nearby address are brought into cache, **spatial locality is reduced.**
b. Smaller block size more number of cache blocks and **larger tag memory** as there will be more number of cache lines and each cache line contains tag bits.
c. A **larger cache** will have a **lower miss rate** and **higher delay,**
d. A cache with more **associativity** will have a lower miss rate but **higher delay** due to MUX.
e. A cache with **larger block size** may have lower delay (higher block size needs less decoder circuiting and fewer tag bits memory) but **miss rate will vary** Program with **higher spatial locality** will do well, but program with **low spatial locality** will perform poor.
f. A smaller block size implying **larger cache tag** but it can't **lower cache hit** smaller block size incurs **lower miss penalty** and reduces **miss overhead** because on a miss less data will be brought from lower level.

Types of Cache misses: -

a. **Compulsory Miss** the very first access to a block that can't be in the cache, so the block must be brought into cache. These are also called **cold start misses** or **first reference misses**.
b. **Capacity Miss** If the cache cannot contain all the blocks needed to during execution of a program. Capacity misses will occur because of blocks being discarded and later retrieved.
c. **Conflict Miss** If the block placement strategy is **set associative** or **direct mapped** conflict misses (in addition to compulsory and capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. There are also called **collision misses** or **interference misses.**

Replacement Algorithm when cache is full

I. FIFO
II. LRU
III. Random

Performance:

Cost average per bit = (C1*S1 + C2*S2)/ (S1 + S2)
C1 = average cost per bit of upper level memory m1
C2 = average cost per bit of upper level memory m2
S1 = size of m1
S2 = size of m2

Hit Rate and Miss Penalty (Hamacher)

Hit rate: the number of hits stated as a fraction of all attempted accesses is called the hit rate.
Miss rate: the number of misses stated as a fraction of attempted accesses is called the miss rate.
Miss Penalty: The extra time needed to bring the desired information into the cache is called miss penalty.

From Hamacher:

Let h be the hit rate, M the miss penalty, that is, the time to access information in the main memory, and C the time to access information in cache

tavg = h*C + (1-h)*M

Example: If the computer has no cache, processor takes 10 cycles for each memory read access and 17 cycles are required to load a block into the cache. Assume that 30% of the instructions in a typical program perform a read operation. Assume that the hit rates in the cache are 0.95 for instructions and 0.9 for data. Let us further assume that the miss penalty is the same for both read and write accesses. Performance improvement

There 130 memory access for every 100 instructions executed. It means there are 100 instructions and 30 instructions are memory read.

$$\frac{Timewithoutcache}{Timewithcache} = \frac{130 * 10}{100(0.95 * 1 + 0.05 * 17) + 30(0.9 * 1 + 0.1 * 17)}$$

= 5.04
The computer with cache performs 5 times faster.

How effective the cache is compared to an ideal cache?
Ideal cache there is no miss, all memory references take 1 cycles

$$= \frac{100(0.95 * 1 + 0.05 * 17) + 30(0.9 * 1 + 0.1 * 17)}{130 * 1} = 1.98$$

Two Level Cache: (Hamacher)

Tavg = h1C1 + (1-h1)h2C2 + (1-h1)(1-h2)M (if memory is simultaneous)

Where
h1 is the hit rate in the L1 cache
h2 is the hit rate in the L2 cache
C1 is the time to access information in the cache L1
C2 is the time to access information in the cache L2
M is the time to access information in the memory

From GO and Gatebook

If memory is hierarchical then"

Tavg = h1C1 + (1-h1)h2(C1 + C2) + (1-h1)(1-h2)(C1 + C2 + M) can further be simplified to

Tavg = C1 + (1-h1)*C2 + (1-h1)(1-h2)*M

Lockup free cache:
Prefetching stops other accesses to the cache until the pre-fetch is completed. A cache of this type is said to be **locked** while it serves a miss.
A cache that can support multiple outstanding miss information it is also called as **Lockup free.** It can service only one miss at a time, it must include circuitry that keeps track of all information about these misses. That circuit is called as **miss information handling register.**

From Notes and Gateoverflow (Multilevel Cache)

a. **Local Miss Rate**
= (number of misses in that cache)/ (no. of access request to that cache)

b. Global Cache Rate
= (number of misses in that cache)/ (no. of CPU generated requests)

From A Quantitative Approach by Peterson

1.

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Hit time is the time to hit in cache.

2.

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

Final Formula from (GO)

Number of memory stalls/memory reference = MissRateL1*MissPenaltyL1
MissPenaltyL1 = Hit Time L2 + MissRateL2*Miss PenaltyL2

# of stalls/instruction = (no. of memory stalls/mm reference)*(memory reference/instruction)

Question (GO)

Suppose there are 500 memory references in which 50 misses in the 1st level cache and 20 misses in the 2nd level cache . Let the miss penalty from L2 cache to memory is 100 cycles .

Hit time in L2 cache is 20 cycles and hit time in L1 cache is 10 cycles . If there are 2.5 memory reference/instruction , average number of stall cycles per instruction will be _____

Solution:

no. of memory stalls/ memory reference: Miss in L1* MissPenaltyL1

Miss Penalty L1 = Hit Time L2 + Miss in L2*MissPenaltyL2

no. of memory stalls/memory reference $= \frac{50}{500} * (20 + \frac{20}{50} * 100) = 6$

Avg. stall/ instruction = (no. of memory stalls/memory reference)*memory reference/Instruction

= 6*2.5 = 15

Hence, 15 is correct answer!

Que:- (Peterson)

Find the AMAT for a processor with a 1 ns clock cycle time, a miss penalty of 20 clock cycles, a miss rate of 0.05 misses per instruction, and a cache access time (including hit detection) of 1 clock cycle. Assume that the read and write miss penalties are the same and ignore other write stalls.

The average memory access time per instruction is

$$\begin{aligned} \text{AMAT} &= \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty} \\ &= 1 + 0.05 \times 20 \\ &= 2 \text{ clock cycles} \end{aligned}$$

or 2 ns.

Que: - (GO Question)

suppose that in 1000 memory reference there are 40 misses in the first level cache and 20 misses in the second level cache. Assume miss penalty from the L2 cache to memory is 100 cycles the hit time of the L2 cache is 10 clock cycles.the hit time of the L1 cache is 1 clock cycle.

Ques. if there are 1.5 memory references per instruction. What is the average stall cycles per instruction

a. 3.4 cycles b. 3.5 cycles c. 3.2 cycles d. 3.6 cycles

GO Solutions:

1.5 memory references per instructions. There are 1000 memory references hence there will be

x*1.5 = 1000

x = 1000/1.5 = 10000/15 = 2000/3 instructions

No. of stall cycles = miss in L1*MissPenaltyL1 + Miss in L2*Miss PenaltyL2

= 40*10 + 20*100 = 400 + 2000 = 2400 memory stall cycles

No. of stall cycles / Instructions = 2400/(2000/3) = 7200/2000 = 3.6

Que:- (GO Question)

Suppose there are 500 memory references in which 50 misses in the 1st level cache and 20 misses in the 2nd level cache . Let the miss penalty from L2 cache to memory is 100 cycles .

Hit time in L2 cache is 20 cycles and hit time in L1 cache is 10 cycles . If there are 2.5 memory reference/instruction , average number of stall cycles per instruction will be _____

Solution:-

There are 500 memory references and there are 2.5 memory reference/instruction hence there are x*2.5 = 500;
x=500/2.5 = 200 instructions.

no. of memory stalls = Miss in L1* MissPenaltyL1 + Miss in L2* Miss PenaltyL2
=50*20 + 20*100 = 1000 + 2000 = 30000 memory stall cycles

No. of stall cycles/instruction = 3000/200 = 15

Flynn's Classifications:

1. The normal operation of a computer is to fetch instruction memory and execute them in the processor.
2. The sequence of instruction read from memory and constitutes an **instruction stream.**
3. The operations performed on the data in the processor constitutes **data stream**
4.

NOTE: - Parallel processing may occur in the instruction stream, in the data stream or in both

Flynn's classification divides computers into four major groups as follows:

1. Single instruction stream, single data stream (SISD)
2. Single instruction stream, multiple data stream (SIMD)
3. Multiple instruction stream, single data stream (MISD)
4. Multiple instruction stream, multiple data stream (MIMD)

SISD:-

It represents the organization of a single computer containing a control unit, a processor unit, and a memory unit. Instructions are executed **sequentially** and the system may or may not have internal parallel processing capabilities (in the form of pipelining).

SIMD:-

An organization that includes many processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data.

MISD:-

It's only a theoretically concept. No practical system implemented it.

MIMD:-

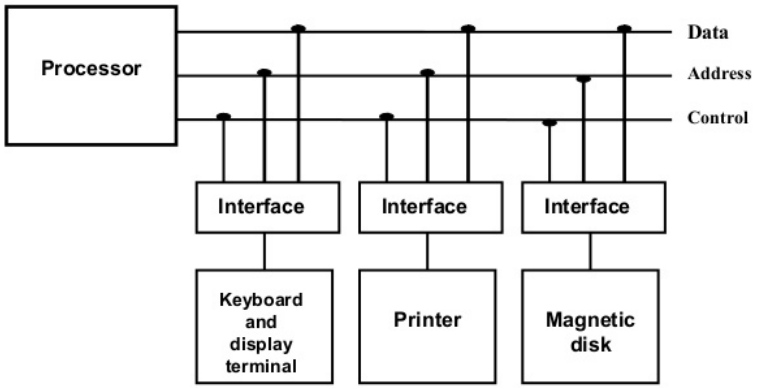
A computer system capable of processing several programs at the same time. **Most multiprocessor and multicomputer systems can be classified in this category**.

Input-Output Interface

Input-output interface provides a method for transferring information between internal storage and external I/O devices.

a. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices.
b. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU.
c. **Data codes and formats in peripherals differ from the word format in the CPU and memory**.

To resolve these differences, computer systems include special hardware components between the CPU and peripherals to supervise and synchronize all input and output transfers. These components are called **interface units** because they interface between the processor bus and the peripheral device.



Connection of I/O bus and input-output device

There are three ways that computer buses can be used to communicate with **memory and I/O**:

1. Use two separate buses, one for memory and the other for I/O.
2. Use one common bus for both memory and IO but have separate control lines for each.
3. Use one common bus for memory and I/O with common control lines

1. **I/O processor (IOP)**
   In the first method, the computer has independent sets of data, address, and control buses, one for accessing memory and the other for I/O. This is done IOP in computers that provide **a separate I/O processor (IOP) in addition to the central processing unit (CPU)**. The memory communicates with both the CPU and the IOP through a memory bus. The IOP communicates also with the **input and output devices through a separate I/O bus with its own address, data and control lines**.

2. **Isolated I/O** (Common address space and different control lines)
   In the isolated I/O configuration, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register. When the CPU fetches and decodes the operation code of an input or output instruction**, it places the address associated with the instruction into the common address lines. At the same time, it enables the I/O read (for input) or I/O write (for output) control line**. This informs the external components that are attached to the common bus that the address in the address lines is for an

interface register and not for a memory word. On the other hand, when the CPU is fetching an instruction or an operand from memory, it places the memory address on the address lines and **enables the Memory read or memory write control line**. This informs the external components that the address is for a memory word and not for an I/O interface.

3. **Memory mapped I/O** (take few addresses from memory address space)
   It uses the same address space for both memory and I/O. This is the case in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as memory-mapped I/O. The computer treats an interface register as being part of the memory system. **The assigned addresses for interface registers cannot be used for memory words, which reduces the memory address range available**.
   It allows the computer to use the **same instructions** for either input-output transfers or for memory transfers.

I/O Interface:-

1. Programmed I/O
2. Interrupt driven I/O
3. Direct Memory Access (DMA)

1. Programmed I/O

With programmed I/O, data are exchanged between the processor and the I/O module. **The processor executes a program that gives it direct control of the I/O operation**, including sensing device status, sending a read or write command, and transferring the data. When the processor issues a command to the I/O module, it must wait until the I/O operation Is complete. If the processor is faster than the I/O module, this is wasteful of processor time.

	No Interrupts	Use of Interrupts
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

The I/O module takes no further action to alert the processor. In particular, it **does not interrupt The processor**. Thus, it is the responsibility of the processor periodically to check the status of the I/O module until it finds that the operation is complete.

2. Interrupt Driven I/O

In programmed I/O the processor, while waiting, must repeatedly interrogate the status of the I/O module. As a result, the level of the performance of the entire system is severely degraded.

An alternative is for the processor to issue an I/O command to a module and then go on to do some other useful work. The I/O module will then interrupt the processor to request service when it is ready to exchange data with the processor. The processor then executes the data transfer, as before, and then resumes its former processing.

For input, the I/O module receives a READ command from the processor. The I/O module then proceeds to read data in from an associated peripheral. Once the data are in the module's data register, the module signals an interrupt to the processor over a control line. The module then

waits until its data are requested by the processor. Then the request is made, **the module places its data on the data bus and is then ready for another I/O operation**.

The processor issues a READ command. It then goes off and does something else. At the end of each instruction cycle, the processor checks for interrupts. When the interrupt from the I/O module occurs, the processor saves the context of the current program and processes the interrupt. In this case, the processor reads the word of data from the I/O module and stores it in memory. It then restores the context of the program it was working on and resumes execution.

Interrupt I/O is **more efficient than programmed I/O** because it eliminates needless waiting. However, **interrupt I/O still consumes a lot of processor time**, because every word of data that goes from memory to I/O module or from I/O module to memory must pass through the processor.

Note:-
1. In Programmed I/O data will be transferred based on the speed of peripheral device.
2. In Interrupt Driven I/O, data will be transferred by the speed of I/O module.

Direct Memory Access (DMA)

When large amount of data to be moved, DMA is required.

DMA Function

The DMA module is capable of **mimicking** the processor and, indeed, of taking over control of the system from the processor. It needs to do this to transfer data to and from memory over the system bus. For this purpose, the DMA module must use the bus only when the processor does not need it, or it must force the processor to suspend operation temporarily.

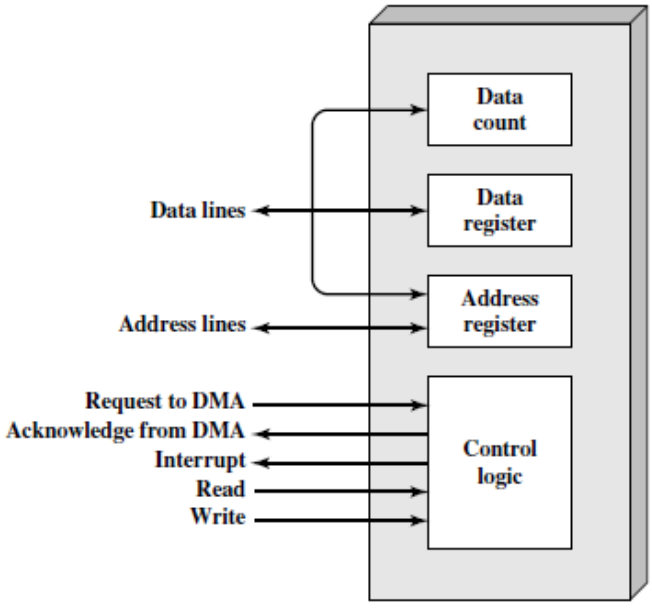


Figure 7.11 Typical DMA Block Diagram

When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information:

- Whether a read or write is requested, using the read or write control line between the processor and the DMA module
- The address of the I/O device involved, communicated on the data lines
- The starting location in memory to read from or write to, communicated on the data lines and stored by the DMA module in its **address register**.
- The number of words to be read or written, again communicated via the data lines and stored in the **data count register**.

The processor then continues with other work**. It has delegated this I/O operation to the DMA module**. The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor. Thus, **the processor is involved only at the beginning and end of the transfer**.

If the DMA module is to transfer a block of data from memory to disk, it will do the following:

1. 1. The peripheral device (such as the disk controller) will request the service of DMA by pulling DREQ (DMA request) high.
2. The DMA will put a high on its HRQ (hold request), signalling the CPU through its HOLD pin that it needs to use the buses.
3. The CPU will finish the present bus cycle (not necessarily the present instruction) and respond to the DMA request by putting high on its HDLA (hold acknowledge), thus telling the DMA that it can go ahead and use the buses to perform its task. HOLD must remain active high as long as DMA is performing its task.
4. DMA will activate DACK (DMA acknowledge), which tells the peripheral device that it will start to transfer the data.
5. DMA starts to transfer the data from memory to peripheral by putting the address of the first byte of the block on the address bus and activating MEMR.
6. After the DMA has finished its job it will deactivate HRQ, signalling the CPU that it can regain control over its buses.

Frequency: - cycles per second was officially replaced by the hertz. Frequency means oscillations (cycles) per second in Hz.

Write-Back, Write-Through with Hierarchical and Simultaneous Access (GO)

For hierarchical access and write-through:

$$T_{read} = H \times T_{cache} + (1 - H) \times (T_{cache} + T_{memory\_block})$$
$$= T_{cache} + (1 - H) \times T_{memory\_block}$$

For simultaneous access and write-through:

$$T_{read} = H \times T_{cache} + (1 - H) \times (T_{memory\_block})$$

Due to hit rates being larger and cache being much faster, the above 2 times are almost the same.

> Hierarchical access and write-though case for memory write is actually not present in practical as since memory is always accessed in case of write through, it makes sense to provide simultaneous write to cache and memory.

For simultaneous access and write-through:

$T_{write} = T_{memory\_word}$

Notice that on write, a *word* is being updated and on read a *block* is retrieved from memory.

From My Notes

Average Read Time(Write Back)

TAvgRead = HrTc + (1-Hr)[%dirtybits(Write back + Read Allocate + Read from Cache) + %clearbits(read allocate + read from cache)

TAvgRead =Hr*Tc + (1-Hr)[%dirtybits(Tm + Tm + Tc) + %clearbits(Tm + Tc)]

Hr = Cache Read hit
Tc = Cache Read Time
Tm = Memory Write time / Write Allocate Time

Average Write Time(Write Back)

TAvgWrite = HwTc + (1-Hw)[%dirtybits(Tm + Tm + Tc) + %clearbits(Tm + Tc)]

Tc = Write time in cache
Tm = Write Allocate/Write time in m/m

Constant Linear Velocity with CLV, the density of bits is uniform from cylinder to cylinder. Because there are more sectors in outer cylinders, the disk spins slower when reading those cylinders, causing the rate of bits passing under the read-write head to remain constant. This is the approach used by modern CDs and DVDs.

Constant Angular Velocity with CAV, the disk rotates at a constant angular speed, with the bit density decreasing on outer cylinders. (These disks would have a constant number of sectors per track on all cylinders.)

Track Capacity = storage density*circumference
Circumference=2*pi*r

Que: A disk has 8 equidistant tracks. The diameters of the innermost and outermost tracks are 1 cm and 8 cm respectively. The *innermost track* has a storage capacity of 10 MB.
What is the total amount of data that can be stored on the disk if it is used with a drive that rotates it with?

1. Constant Linear Velocity
2. Constant Angular Velocity?

Solution:
Diameter of innermost = 1cm, radius=0.5cm
Diameter of outermost = 8cm, radius=4cm

1. **CLV**

Track Capacity = storage density*circumference
10 MB = density*2pi*r = density*diameter
density=10MB/2*pi*0.5 = 10MB/pi MB/com

For 2nd track capacity = density*circumference
= (10MB/pi)*2*pi*1 = 20MB

For 3rd Track capacity = (10MB/pi)*2*pi*1.5 = 30MB

[1] (1) If disk rotates with constant velocity then total amount of data that can be stored on the tracks is linearly related to track length. The track with diameter has a storage capacity of 10 MB and length π (diameter 1 cm). Hence, if diameter of the track equals d then it has length πd and a storage capacity of $10d$ MB. The amount of data that can be stored equals

$$\sum_{d=1}^{8} 10d = 10 \sum_{d=1}^{8} d = 10 \cdot \frac{8 \cdot 9}{2} = 360 \text{ MB}.$$

2. **CAV**
   Track capacity remains same for all tracks = 8*10MB = **80MB**

Conflict/ Capacity/ Capacity Misses:-

1. **Compulsory miss/Cold Start Miss/First Reference Miss** the very first access to a memory Block that must be brought into cache. When a block of main memory is trying to occupy fresh empty line of cache.

2. **Conflict Miss/Collision Miss/Interference Miss** in addition to Compulsory and Capacity misses In Direct Mapped and in Set Associative Memories. When still there are empty lines in the cache, block of main memory is conflicting with the already filled line of cache, i.e. even when empty place is available, and block is trying to occupy already filled line.

3. **Capacity miss** if the cache CANNOT contain all the blocks needed during the execution of a program, Capacity Misses will occur because of blocks are discarded and retrieved later.

Question (GO)

Consider a 2 - way set associative cache memory with 4 sets and total 8 cache blocks (0 - 7) .Main memory has 64 blocks (0 - 63). If LRU policy is used for replacement and cache is initially empty then total number of conflict cache misses for the following sequence of memory block references is :

0 5 9 13 7 0 15 25

A. 2
B. 3
C. 0
D. 1

GO Solution:
Reference string: 0 5 9 13 7 0 15 25
0 - set 0 - Compulsory miss
5 - Set 1 - Compulsory miss
9 - Set 1 - Compulsory miss (set 1 now full)
13 - set 1 - Compulsory miss (5 replaced, but this is not conflict miss; if 5 is again accessed then that access causes conflict miss by its definition. Remember that conflict miss is something which can be avoided if the cache is fully associative and hence it can never happen with compulsory miss)
7 - Set 3 - Compulsory miss
0 - set 0 - Hit
15 - Set 3 - Compulsory miss
25 - Set 1 - Compulsory miss - 9 replaced. Again this is not a conflict miss.
So, number of conflict misses = 0.

QUE GATE 2017-1-51

Consider a $2-$way set associative cache with 256 blocks and uses LRU replacement. Initially the cache is empty. Conflict misses are those misses which occur due to the contention of multiple blocks for the same cache set. Compulsory misses occur due to first time access to the block. The following sequence of access to memory blocks :

$$\{0, 128, 256, 128, 0, 128, 256, 128, 1, 129, 257, 129, 1, 129, 257, 129\}$$

is repeated 10 times. The number of *conflict misses* experienced by the cache is _____ .

GO Solutions

I am reiterating the same thing what Arjun already explained.

{0,128,256,128,0,128,256,128,1,129,257,129,1,129,257,129}

1st Iteration:

for {0,128,256,128,0,128,256,128}

Block Id	Type	Seto content
0	Compulsory Miss	0
128	Compulsory Miss	0 128
256	Compulsory Miss	128 256
128	hit	256 128
0	Conflict Miss	128 0
128	hit	0 128
256	Conflict Miss	128 256
128	hit	256 128

Total number of conflict misses =2;

Similarly for {1,129,257,129,1,129,257,129}, total number of conflict misses in set1 = 2

Total number of conflict misses in 1st iteration = 2+2=4

2nd iteration:

for {0,128,256,128,0,128,256,128}

Block Id	Type	Seto content
0	Conflict Miss	128 0
128	hit	0 128
256	Conflict Miss	128 256
128	hit	256 128
0	Conflict Miss	128 0
128	hit	0 128
256	Conflict Miss	128 256
128	hit	256 128

Total number of conflict misses = 4

Similarly for {1,129,257,129,1,129,257,129}, total number of conflict misses in set1 = 4

Total Number of conflict misses in 2nd iteration = 4+4=8

Note that content of each set is same, before and after 2nd iteration. Therefore each of the remaining iterations will also have 8 conflict misses.

Therefore, overall conflict misses = 4+8*9 = 76

RAW/WAR/WAW Data Dependencies

Que 15 (GO)
I1: R1 = 100
I2: R1 = R2 + R4
I3: R2 = R4 + 25
I4: R4 = R1 + R3
I5: R1 = R1 +30
Calculate no. of RAW (True dependency), WAR (Anti dependency), and WAW (Output dependency) dependencies:

Solution:
RAW: J wants to read before I can write

1. I2 -> I4 (R1)
2. I2 -> I5 (R1)

3. I1 -> I4 (R1)
4. I1 -> I5(R1)

Here 3 & 4 are data dependencies but **not data hazards** as I2 will cover I1. There is a dependency between I1 and I2.

WAR: J wants to Write before I can read

1. I2 -> I3 (R2)
2. I4 -> I2 (R4)
3. I3 -> I4 (R4)
4. I5 -> I4 (R1)

WAW: J wants to write before I can write

1. I1->I2 (R1)
2. I2->I5 (R1)
3. I1->I5 (R1)

Here, no WAW Hazard between I1 and I5, I2 will cover it

Hence, Data Hazards and there will be 11 data dependencies.

GATE2006-IT-78

A pipelined processor uses a 4-stage instruction pipeline with the following stages: Instruction fetch (IF), Instruction decode (ID), Execute (EX) and Writeback (WB). The arithmetic operations as well as the load and store operations are carried out in the EX stage. The sequence of instructions corresponding to the statement X = (S - R * (P + Q))/T is given below. The values of variables P, Q, R, S and T are available in the registers R_0, R_1, R_2, R_3 and R_4 respectively, before the execution of the instruction sequence.

ADD	R_5, R_0, R_1	; $R_5 \leftarrow R_0 + R_1$
MUL	R_6, R_2, R_5	; $R_6 \leftarrow R_2 * R_5$
SUB	R_5, R_3, R_6	; $R_5 \leftarrow R_3 - R_6$
DIV	R_6, R_5, R_4	; $R_6 \leftarrow R_5/R_4$
STORE	R_6, X	; X $\leftarrow R_6$

The number of Read-After-Write (RAW) dependencies, Write-After-Read(WAR) dependencies, and Write-After-Write (WAW) dependencies in the sequence of instructions are, respectively,

A. 2, 2, 4
B. 3, 2, 3
C. 4, 2, 2
D. 3, 3, 2

Solution: - definition mentioned in above question is from standard resources, it is ambiguous term hence solve by looking at options.

RAW:
I1 – I2 (R5)
I2 – I3 (R6)
I3 – I4 (R5)
I4 – I5 (R6)

WAR:
I2 – I3 (R5)
I3 – I4 (R6)

WAW:
I1 – I3 (R5)
I2 – I4 (R6)