Internal Assessment Test 2– JAN 2026

Sub:	**Digital Design and Computer Organization**	Sub Code:	BCS302	Branch:	ISE

	Scheme and Solutions	MARKS	CO	RBT
1	a. Mention four types of operations to be performed by instructions in a computer **Scheme:**	5+5	CO3	L2

a. Mention four types of operations to be performed by instructions in a computer **Scheme:**

Each operations -1 marks

Solution:

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

To perform these operations a computer requires a set of instructions (Instruction Set Architecture), it is denoted with two notations namely,

 Register Transfer Notation

 Assembly-Language Notation

REGISTER TRANSFER NOTATION:

•Possible locations that may be involved in transfer of information from one location in a computer to another are Memory locations, Processor registers, Registers in the I/O subsystem.

•Locations are represented in capital letters.

R=1/P

•Memory locations are represented like LOC A, LOC B etc.,

•Processor registers are represented like R0, R1 etc.,

•I/O are represented like DATAIN, DATAOUT, OUTBUFFER, INBUFFER etc.,

•Contents /Data of the locations are denoted with square brackets [].

•Ex: C=A+B to be written in RTN, then it is denoted with an

•LHS of the arrow is destination, RHS is the source.

•$C <- [A] + [B]$

•If A is to move to R0 then in RTN it can be denoted as $R0$ $[A]$

•R4 < − [R2]+[R3] Here, the contents of registers R2 and R3 are added and their sum is placed into register R4. This type of notation is known as Register Transfer Notation (RTN).

ASSEMBLY-LANGUAGE NOTATION

•The notation to represent machine instructions and programs is assembly language.

Ex: MOVE LOC, R1

•Here, the contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.

•The second example of adding two numbers contained in processor registers R2 and R3 and placing their sum in R4 can be specified by the assembly-language statement

Ex: Add R4, R2, R3

•In this case, registers R2 and R3 hold the source operands, while R4 is the destination.

b. Explain with basic types of instructions formats to carry out C<-[A]+[B].

Scheme:

	Each Instructions -1 Marks **Solution: □BASIC INSTRUCTION TYPES** The types of instruction depends upon the number of operands used in the instruction. The instruction types are as follows: •Three-address instruction •Two-address instruction •One-address instruction •Zero-address instruction **THREE-ADDRESS INSTRUCTION** Syntax: **Operation Source1, Source 2, Destination** •Consider an example C=A+B Operation is add, Source1 is A, Source 2 is B, Destination is C Move LOCA, R0 Move LOC B, R1 **Add R0,R1,R2** **TWO-ADDRESS INSTRUCTION** •General two-address instruction has the format, Operation Source, Destination **Add R2,R3** • Contents of R2,R3 are added and stored into R3 EX : C=A+B • Move A, R0 #from mem loc A data move to R0 • Add B,R0 #data at mem loc B is added with R0, result in R0 • Store R0,C # result store in mem loc C from R0 **ONE-ADDRESS INSTRUCTION** •General one-address instruction has the format, Operation Operand EX : C=A+B Move A, # content in A move to accumulator Add B # Add content in B with the content in accumulator Store C **ZERO-ADDRESS INSTRUCTION** •Both source and destination defined implicitly. •Here the contents of operands are stored in PUSH DOWN stack. EX: C=A+B PUSH A (Pushes the value of A onto the stack) PUSH B (Pushes the value of B onto the stack) ADD (Pops A and B, adds them, and pushes the sum back onto the stack) POP X (Pops the result from the stack and stores it in location X).			
2	Explain in detail the various addressing modes with an example **Scheme:** **Explaining each addressing modes 1 marks** **Solution:**	10	CO3	L2

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	Ri	EA = Ri
Absolute	LOC	EA = LOC
Register indirect	(Ri)	EA = [Ri]
Index	X(Ri)	EA = [Ri] + X
Base with index	(Ri,Rj)	EA = [Ri] + [Rj]

EA = effective address
Value = a signed number
X = index value

The different ways for specifying the locations of instruction operands are known as *addressing modes*.

Immediate mode: The operand is given explicitly in the instruction.

Add R4, R6, #200

- **Register mode—The** operand is the contents of a processor register; the name of the register is given in the instruction.
  - **Add R4, R2, R3**
- Registers R2 and R3 hold the two source operands, while R4 is the destination.
- **Absolute mode**—The operand is in a memory location; the address of this location is given explicitly in the instruction.

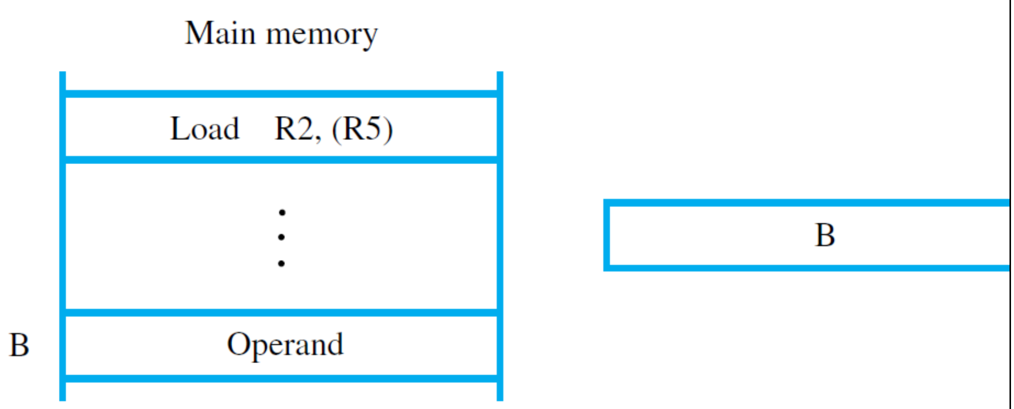
The Absolute mode can represent global variables in a program.

Integer NUM1, NUM2, SUM;

in a high-level language program will cause the compiler to allocate a memory location to each of the variables NUM1, NUM2, and SUM. Whenever they are referenced later in the

program, the compiler can generate assembly-language instructions that use the Absolute mode to access these variables.

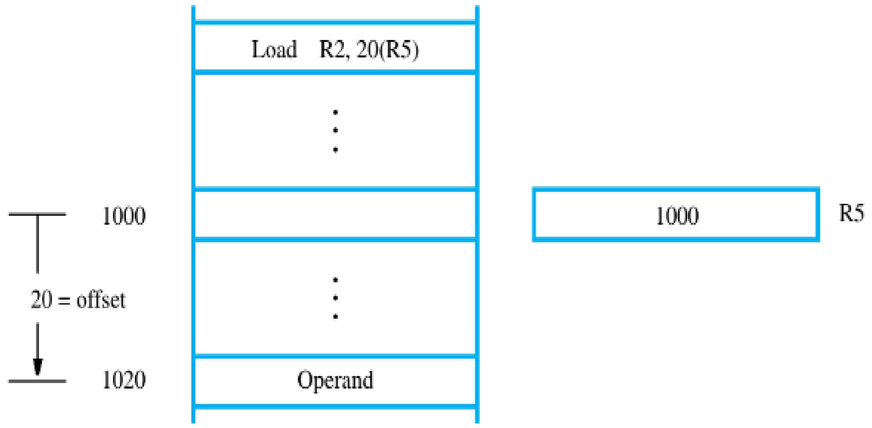
- **Register indirect addressing**



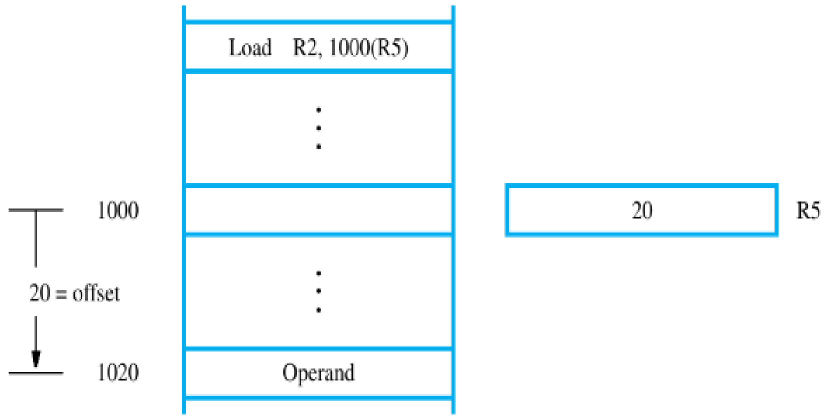
The register acts as a pointer to the list, and we say that an item in the list is accessed indirectly by using the address in the register. The desired capability is provided by the indirect addressing mode.

Indirect mode—The effective address of the operand is the contents of a register that is specified in the instruction.

- Index mode—The effective address of the operand is generated by adding a constant value to the contents of a register.
- the register used in this mode is the **index register. this is general-purpose register.**
- **X(Ri)**
- where X denotes a constant signed integer value contained in the instruction and Ri is the name of the register involved.
- The effective address of the operand is given by
- EA = X + [Ri]
- The contents of the register are not changed in the process of generating the effective address.



(a) Offset is given as a constant



(b) Offset is in the index register

6. Base with Index mode

The effective address of the operand is sum of the contents of registers Ri and Rj.

Syntax: (Ri,Rj)

EA = [Ri]+ [Rj]

Example: Add (R1,R2),R3

Here, the index registers, R1 and R2, contains the address of a memory location, which is summed up to give the address to the location where the operand is found. This operand is added to the content in Register R3 and the resultant sum is placed inside Register R3.

8.Relative mode:

The effective address is determined by the Index mode using the program counter in place of the general-purpose register Ri.

Syntax: X(PC)

EA = X + [PC]

Example: Branch > 0 LOOP

This instruction causes program execution to go to the branch target location identified by the name LOOP if the branch condition is satisfied.

This location can be computed by specifying it as an offset from the current value of the program counter. Since the branch target may be either before or after the branch instruction, the offset is given as a signed number.

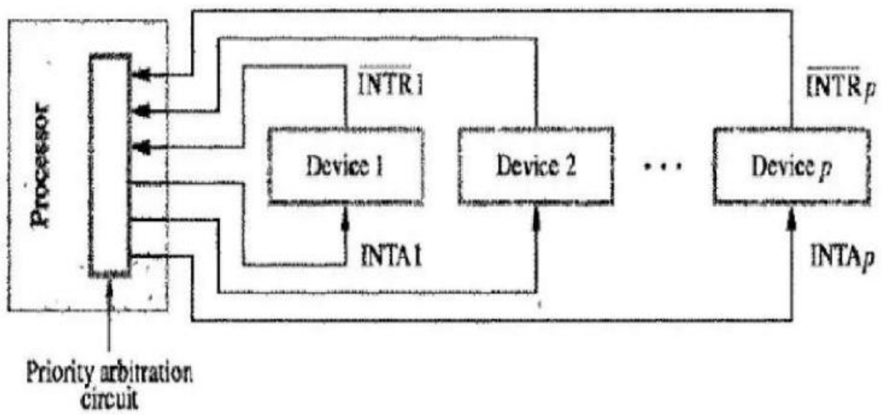
	9. Auto-increment mode: The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically to point to the next item in a list. **Syntax: (Ri)+** **EA = [Ri]; Increment Ri** **Example: Add (R2)+,R1** **10. Auto-decrement mode:** The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand. **Syntax: -(Ri)** **Decrement Ri ; EA = [Ri]** **Example: Add -(R2),R1**			
3	With neat sketches explain various methods for handling multiple interrupt requests **Scheme:** Definition: 2 Marks Classification: 1marks Explanation: 5 Marks Diagram: 2 marks **Solution:** Let us now consider the situation where a number of devices capable of initiating interrupts are connected to the processor. When an interrupt request is received it is necessary to identify the particular device that raised the request. Furthermore, if two devices raise interrupt requests at the same time it must be possible to break the tie and select one of the two requests for service. When the interrupt-service routine for the selected device has been completed, the second request can be serviced. When the device raises an interrupt request, it sets to 1 a bit in its status register, which we will call the IRQ bit. **Polling scheme** The simplest way to identify the interrupting device is to have the interrupt-service routine poll all I/O devices in the system. The first device encountered with its IRQ bit set to 1 is the device that should be serviced. An appropriate subroutine is then called to provide the requested service. The polling scheme is easy to implement. Its main disadvantage is the time spent interrogating the IRQ bits of devices that may not be requesting any service. **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	10	CO4	L2

start executing the corresponding interrupt-service routine. A device requesting an interrupt can identify itself if it has its own interrupt-request signal, or if it can send a special code to the processor through the interconnection network. The processor's circuits determine the memory address of the required interrupt-service routine.

A commonly used scheme is to allocate permanently an area in the memory to hold the addresses of interrupt-service routines. These addresses are usually referred to as interrupt vectors, and they are said to constitute the interrupt-vector table. For example, 128 bytes may be allocated to hold a table of 32 interrupt vectors. Typically, the interrupt vector table is in the lowest-address range. The interrupt-service routines may be located anywhere in the memory. When an interrupt request arrives, the information provided by the requesting device is used as a pointer into the interrupt-vector table, and the address in the corresponding interrupt vector is automatically loaded into the program counter 4.3.3

Interrupt Nesting

It is necessary to accept an interrupt request from the clock during the execution of an interrupt-service routine for another device, i.e., to nest interrupts. I/O devices should be organized in a priority structure. An interrupt request from a high-priority device should be accepted while the processor is servicing a request from a lower-priority device. Multiple Priority Scheme: A multiple-level priority organization means that during execution of an Interrupt Service Routine, interrupt requests will be accepted from some devices but not from others, depending upon the device's priority. To implement this scheme, we can assign a priority level to the processor that can be changed under program control. The priority level of the processor is the priority of the program that is currently being executed. The processor accepts interrupts only from devices that have priorities higher than its own. At the time the execution of an ISR for some device is started, the priority of the processor is raised to that of the device. The action disables interrupts from devices at the same level of priority or lower.



Privileged Instruction:

The processor priority is usually encoded in a few bits of the Processor Status word. It can also be changed by program instruction & then it is write into PS. These instructions are called privileged instruction. This can be executed only when the processor is in supervisor mode. The processor is in supervisor mode only when executing OS routines. It switches to the user mode before beginning to execute application program.

Privileged Exception: User program cannot accidently or intentionally change the priority of the processor &disrupts the system operation. An attempt to execute a privileged instruction while in user mode, leads to a special type of interrupt called the privileged exception. Each of the interrupt request line is assigned a different priority level. Interrupt request received over these lines are sent to a priority arbitration circuit in the processor. A request is accepted only if it has a higher priority level than that currently assigned to the processor.

Simultaneous Requests

We also need to consider the problem of simultaneous arrivals of interrupt requests from two or more devices. The processor must have some means of deciding which request to service first. Polling the status registers of the I/O devices is the simplest such mechanism. In this case, priority is determined by the order in which the devices are polled. When vectored interrupts are used, we must ensure that only one device is selected to send its interrupt vector code.

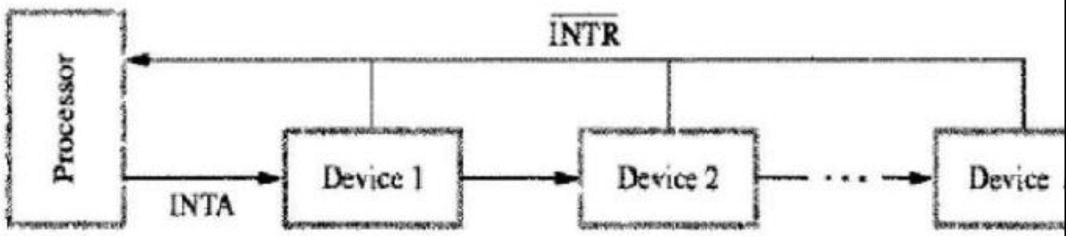


Figure 4.8:Daisy Chain

Here, the interrupt request line INTR is common to all devices. The interrupt acknowledge line INTA is connected in a daisy chain fashion such that INTA signal propagates serially through the devices. When several devices raise an interrupt request, the INTR line is activated & the processor responds by setting INTA line to 1. This signal is received by device 1 Devicel passes the signal on to device2 only if it does not require any service. If devices1 has a pending request for interrupt blocks that INTA signal & proceeds to put its identification code on the data lines. Therefore, the device that is electrically closest to the processor has the highest priority. The advantage of using this scheme is that it requires fewer wires than the individual connections. In Figure 4.9, the devices are organized in groups & each group is connected at a different priority level. Within a group, devices are connected in a daisy chain.

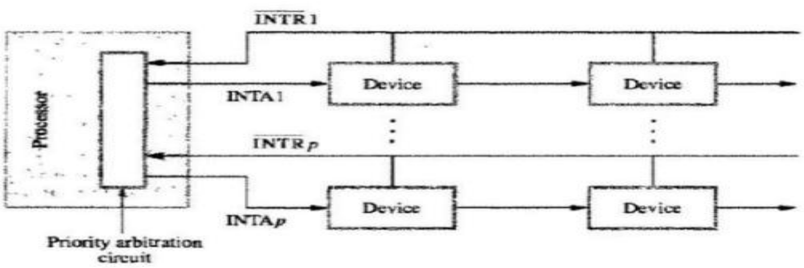


Figure 4.9: Arrangements of priority groups

4	Explain different mapping functions used in cache memory.	10	CO4	L2

Scheme:
Definition: 2 Marks
Classification: 1marks
Explanation: 5 Marks
Diagram: 2 marks
Solution:

Processor is much faster than the main memory. As a result, the processor has to spend much of its time waiting while instructions and data are being fetched from the main memory. Speed of the main memory cannot be increased beyond a certain point. Cache memory is an architectural arrangement which makes the main memory appear faster to the processor than it really is. Cache memory is based on the property of computer programs known as "**locality of reference".**

Analysis of programs indicates that many instructions in localized areas of a program are executed repeatedly during some period of time, while the others are accessed relatively less frequently. These instructions may be the ones in a loop, nested loop or few procedures calling each other repeatedly. This is called "**locality of reference**".

Temporal locality of reference: Recently executed instruction is likely to be executed again very soon. Spatial locality of reference: Instructions with addresses close to a recently instruction are likely to be executed soon.

When processor issues a Read request, a block of words is transferred from the main memory to the cache, one word at a time. Subsequent references to the data in this block of words are found in the cache. At any given time, only some blocks in the main memory are held in the cache. Which blocks in the main memory are in the cache is determined by a "**mapping function".**

When the cache is full, and a block of words needs to be transferred from the main memory, some block of words in the cache must be replaced. This is determined by a **"replacement algorithm".**

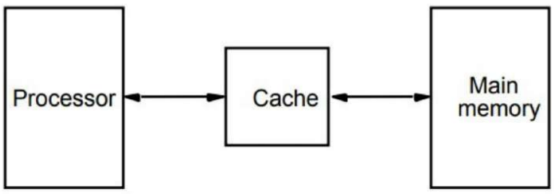


Figure 4.17: Use of a cache memory

Cache Hit:

The processor issues Read and Write requests, If the data is in the cache it is called a Read or Write hit.

Read hit:

The data is obtained from the cache.

Write hit:

Cache has a replica of the contents of the main memory.

Write-through Protocol

Contents of the cache and the main memory may be updated simultaneously.
This is the **write-through protocol**.

Dirty bit:

- Update the contents of the cache, and mark it as updated by setting a bit known as the **dirty bit** or modified bit.

Writeback protocol:

- The contents of the main memory are updated when this block is replaced. This is **writeback or copy-back protocol**.

Cache Miss:
If the data is not present in the cache, then a Read miss or Write miss occurs.
Read miss:
Block of words containing this requested word is transferred from the memory. After the block is transferred, the desired word is forwarded to the processor.
Load-through:
The desired word is forwarded to the processor as soon as it is transferred without waiting for the entire block to be transferred. This is called load-through or early-restart.
Write-miss:
Write-through protocol is used, then the contents of the main memory are updated directly. If write-back protocol is used, the block containing the addressed word is first brought into the cache. The desired word is overwritten with new information.

Cache Coherence problem:
- A bit called as "**valid bit**" is provided for each block.
- If the block contains valid data, then the bit is set to 1, else it is 0.
- The copies of the data in the cache, and the main memory are different. This is called the cache coherence problem.

Cache Mapping:
Mapping functions determine how memory blocks are placed in the cache.
Three mapping functions are:
1. Direct mapping >
2. Associative mapping
3. Set-associative mapping.

Direct mapping:
It is the Simplest mapping technique where each block of main memory maps to only one cache line i.e. if a block is in cache, it must be in one specific place. Main memory locations can only be copied into one location in the cache, accomplished by dividing main memory into blocks that correspond in size with the cache.
Formula to map a memory block to a cache line:

$$\mathbf{i = j \bmod c}$$

i→Cache Line Number (block address of cache)
j→Main Memory Block Number(block address of main memory)
c→Number of Lines in Cache(no.of cache blocks) i.e. we divide the memory block by the number of cache lines and the remainder is the cache line address
In the below figure, Block j of the main memory maps to j modulo 128 of the cache. 0 maps to 0, 129 maps to 1. More than one memory block is mapped onto the same position in the cache. This may lead to contention for cache blocks even if the cache is not full.We can resolve the contention by allowing new block to replace the old block, leading to a trivial replacement algorithm.

Memory address is divided into three fields:

Low order 4 bits determine one of the 16 words in a block. –

When a new block is brought into the cache, the next 7 bits determine which cache block this new block is placed in.

High order 5 bits determine which of the possible 32 blocks is currently present in the cache. These are tag bits. This mapping is simple to implement but not very flexible.

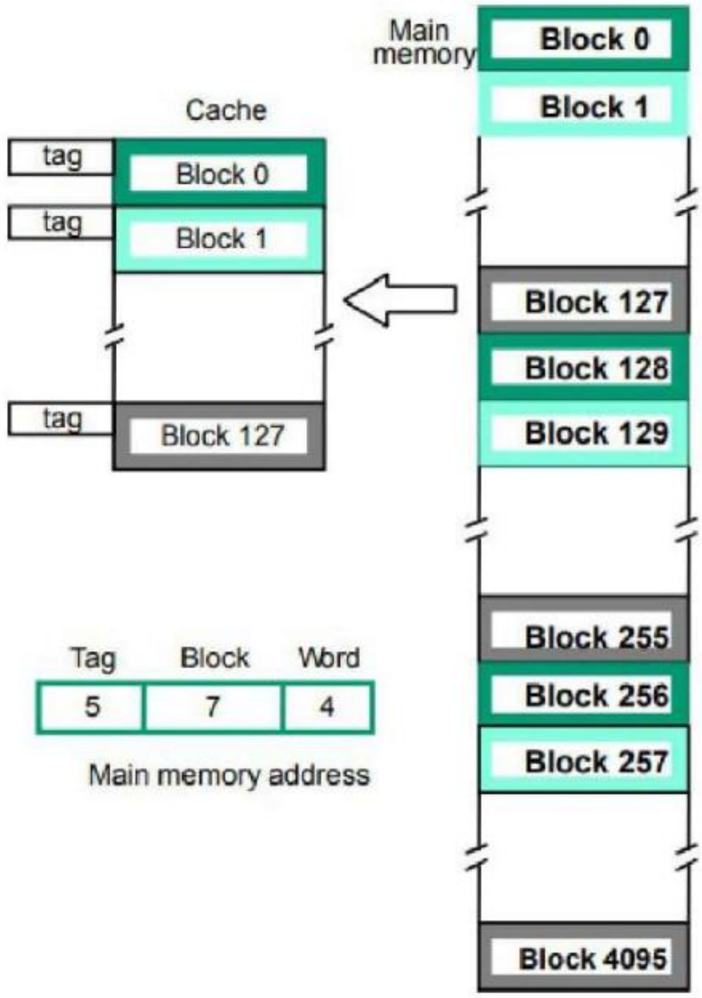
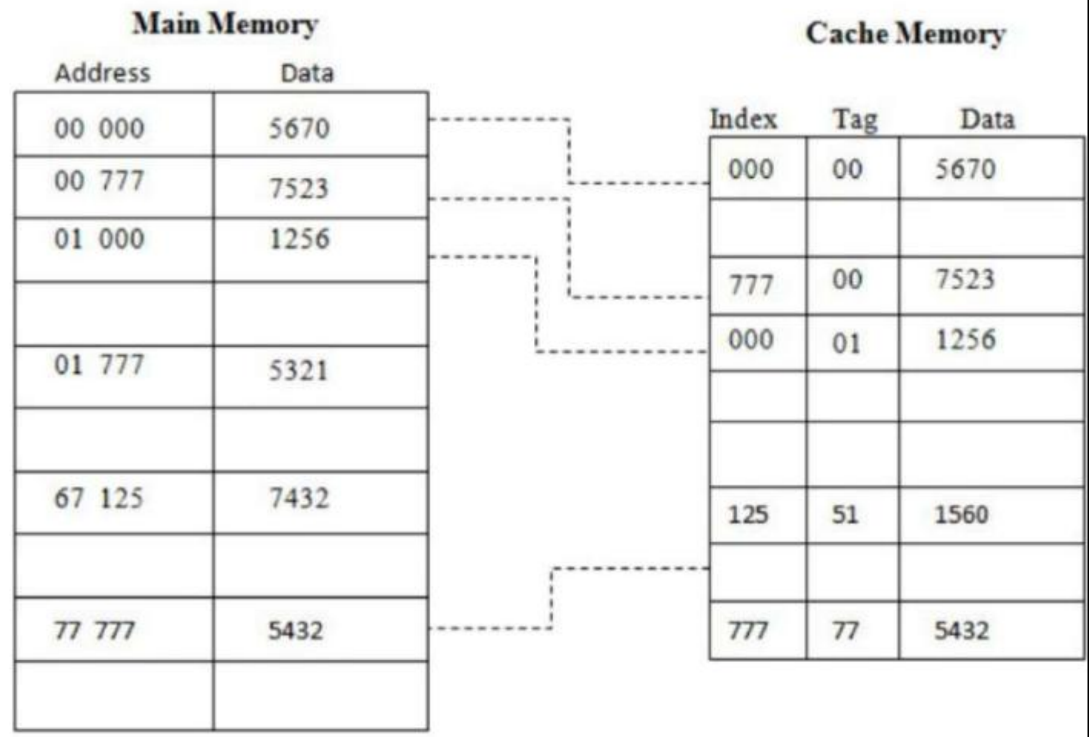


Figure 4.18:Direct Mapping

The direct mapping technique is simple and inexpensive to implement. When the CPU wants to access data from memory, it places a address. The index field of CPU address is used to access address. The tag field of CPU address is compared with the associated tag in the word read from the cache. If the tag-bits of CPU address is matched with the tag-bits of cache, then there is a hit and the required data word is read from cache. If there is no match, then there is a miss and the required data word is stored in main memory. It is then transferred from main memory to cache memory with the new tag.



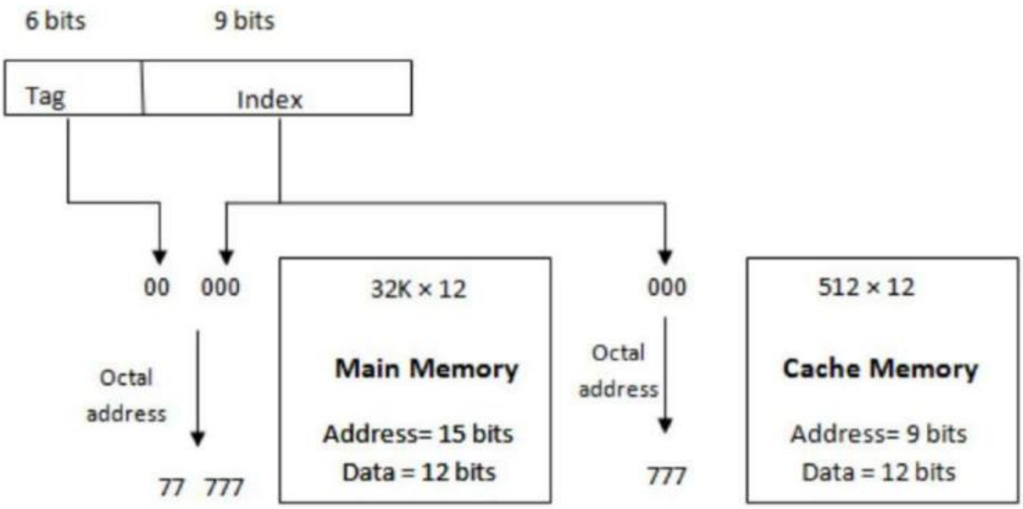


Figure 4.19 :Mapping concept for direct mapping

Fully Associative Mapping:

An associative mapping uses an associative memory. This memory is being accessed using its contents. Each line of cache memory will accommodate the address (main memory) and the contents of that address from the main memory. That is why this memory is also called Content Addressable Memory (CAM). It allows each block of main memory to be stored in the cache. is A fully associative mapping scheme can overcome the problems of the direct mapping scheme. A main memory block can load into any line of cache. Memory address interpreted as tag and word. Tag uniquely identifies block of memory. Every line's tag is examined for a match which also need a Dirty and Valid bit. But Cache searching gets expensive! Ideally need circuitry that can simultaneously examine all tags for a match. Hence lots of circuitry needed, high cost. It is more Flexible, and uses cache space efficiently. Replacement algorithms can be used to replace an existing block in the cache when the cache is full. Cost is higher than directmapped cache because of the need to search all 128 patterns to determine whether a given block is in the cache.

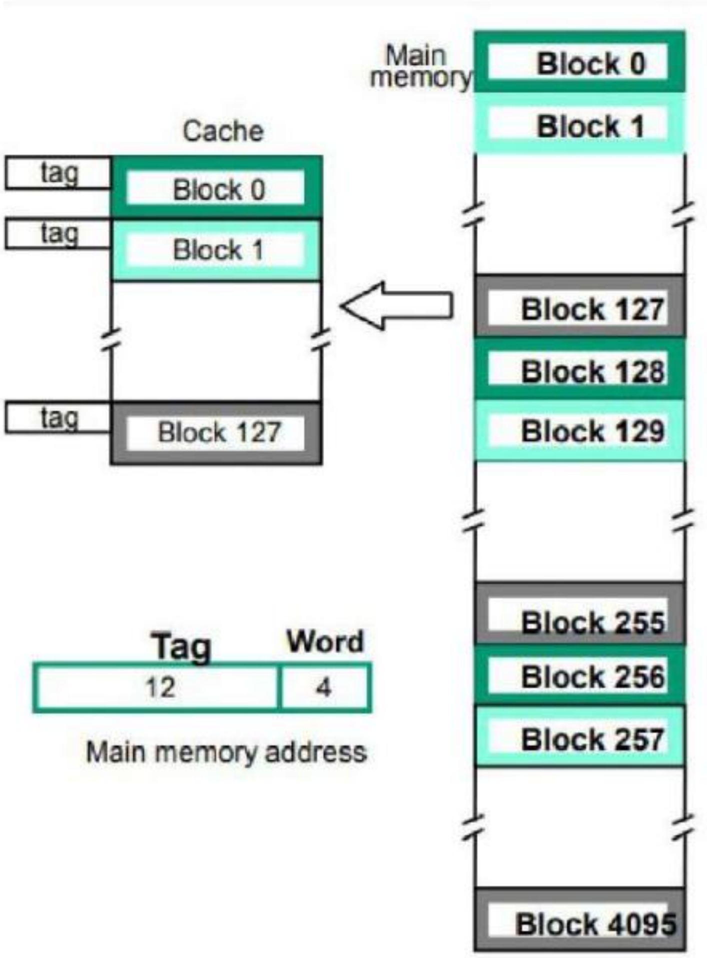
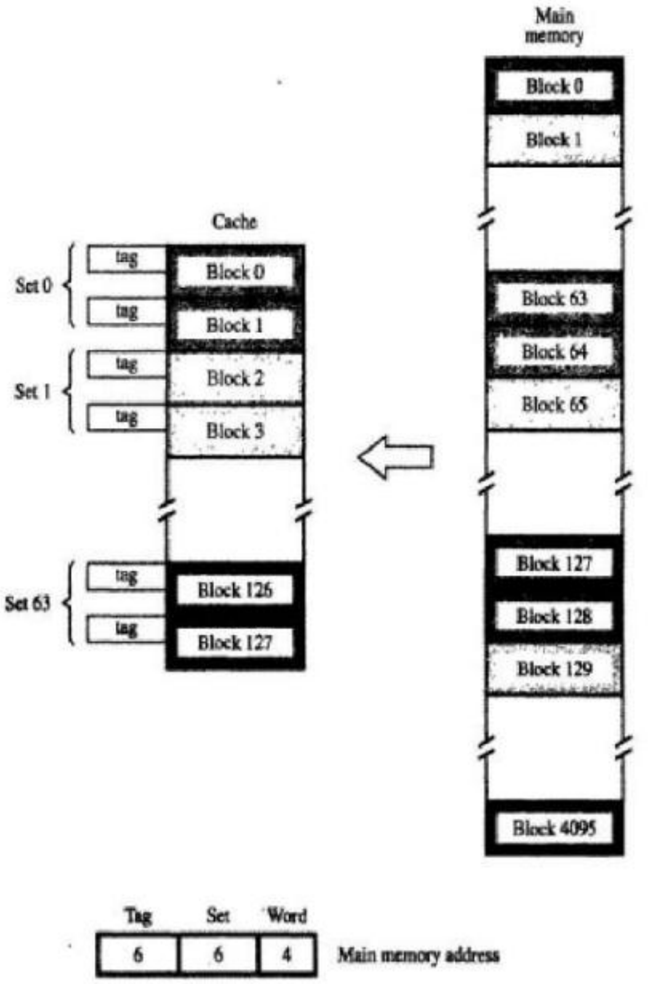


Figure 4.20:Fully Asscociative

Set Associative:

It is a Compromise between fully-associative and direct-mapped cache. Cache is divided into a number of sets. Each set contains a number of lines. A given block maps to any line in a specific set .Use direct-mapping to determine which set in the cache corresponds to a set in memory. Memory block could then be in any line of that set. e.g. 2 lines per set - 2 way associative mapping A given block can be in either of 2 lines in a specific set e.g. K lines per set -K way associative mapping A given block can be in one of K lines in a specific set Much easier to simultaneously search one set than all lines.

	 **Figure 4.21: Set Associative** Blocks of cache are grouped into sets. Mapping function allows a block of the main memory to reside in any block of a specific set. Divide the cache into 64 sets, with two blocks per set. Memory block 0, 64, 128 etc. map to block 0, and they can occupy either of the two positions. Memory address is divided into three fields: - 6 bit field determines the set number. High order 6 bit fields are compared to the tag fields of the two blocks in a set. Set-associative mapping is a combination of direct and associative mapping. Number of blocks per set is a design parameter. - One extreme is to have all the blocks in one set, requiring no set bits (fully associative mapping). - Other extreme is to have one block per set, is the same as direct mapping.			
5	Explain with need diagram a single-bus structure connecting the memory to the processor. **Scheme:** Diagram: 3marks Explanation: 7marks **Solution:**	10	CO5	L2

To execute an instruction, processor has to perform following 3 steps:

1) Fetch contents of memory-location pointed to by PC. Content of this location is an instruction to be executed. The instructions are loaded into IR, Symbolically, this operation can be written as IR←[[PC]]

2) Increment PC by 4 P,← [PC] +4

3) Carry out the actions specified by instruction (in the IR).

The first 2 steps are referred to as **fetch phase**; Step 3 is referred to as **execution phase.**

The operations specified by an instruction can be carried out by performing one or more of the following actions:

1.Read the content of a given memory-location and load them into a register.

2.Read data from one or more register

3.Perform ALU operations and place the result into the register.

4.Store data from a register into a given memory location.

Figure 5.1 shows the single bus organization. ALU and all the registers are interconnected via a single common bus. Data and address line of the external memory bus is connected to the internal processor bus via MDR and MAR respectively(MDR-Memory Data Register and MAR-Memory Address Register).

→ into MDR either from memory-bus (external) or

→ from processor-bus (internal).

MAR's input is connected to internal-bus, and MAR"s output is connected to external-bus.Instruction-decoder & control-unit is responsible for

→ issuing the signals that control the operation of all the units inside the processor (and for interacting with memory bus).

→ implementing the actions specified by the instruction (loaded in the IR)

Registers **R0 through R(n**-1) are provided for general purpose use by programmer.

Three registers **Y, Z & TEMP** are used by processor for temporary storage during execution of some instructions.

These are transparent to the programmer i.e. programmer need not be concerned with them because they are never referenced explicitly by any instruction.

MUX(Multiplexer) selects either

→ output of Y or

→ constant-value 4(is used to increment PC content).This is provided as **input A of ALU.**

B input of ALU is obtained directly from processor-bus.

As instruction execution progresses, data are transferred from one register to another, often passing through **ALU to perform arithmetic or logic operation.**

An instruction can be executed by performing one or more of the following operations:

1) **Transfer a word of data from one processor-register** to another or to the ALU.

2) **Perform 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**.

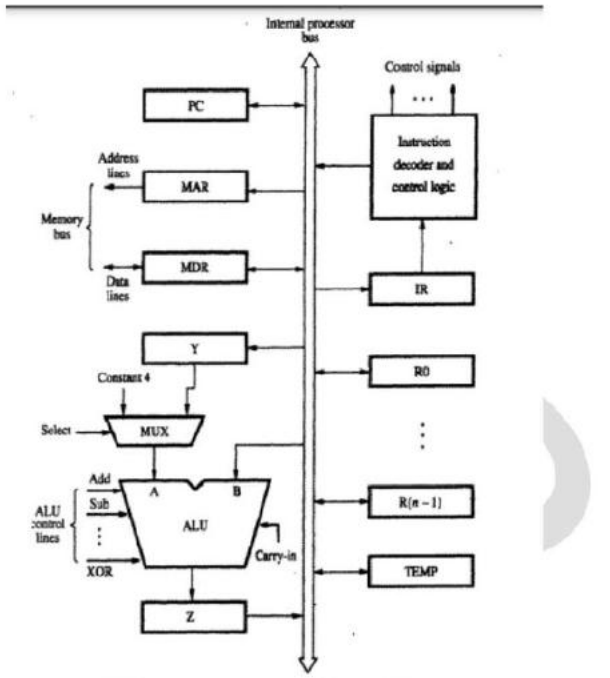


Figure 5.1: Single bus organization of the data path inside a processor

Disadvantage: Only one data word can be transferred over the bus in a clock cycle.

Solution: Providing multiple data-paths allows several data transfer to take place in parallel.

6 a) Write control sequence for the fetching a word with neat timing diagram. 5+5 CO5 L2

Scheme:

Diagram: 3marks

Explanation: 7marks

Solution:

To fetch instruction/data from memory, processor transfers required address to MAR (whose output is connected to address-lines of memory-bus). At the same time, processor issues Read signal on control-lines of memory-bus. When requested-data are received from memory, they are stored in MDR. From MDR, they are transferred to other registers.

The response time of each memory access varies. For this MFC (Memory Function Completed): is used. It is the signal sent from Addressed-device to the processor. MFC informs the processor that the requested operation is completed by addressed device.

Thus MFC is set to 1 to indicate that the contents of the specified location

→ have been read &

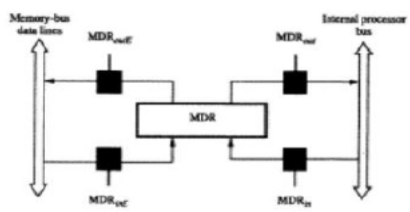
→ are available on data-lines of memory-bus

Consider the instruction **Move (R1),R2.** The sequence of steps is:

1) **$R1_{out}$, MAR_{in}, Read** ;desired address is loaded into MAR & Read command is issued

2) **MDR_{inE}, WMFC** ;load MDR from memory bus & Wait for MFC response from memory.

3) **MDR_{out}, $R2_{in}$** ;load R2 from MDR where WMFC=control signal that causes processor's control circuitry to wait for arrival of MFC signal



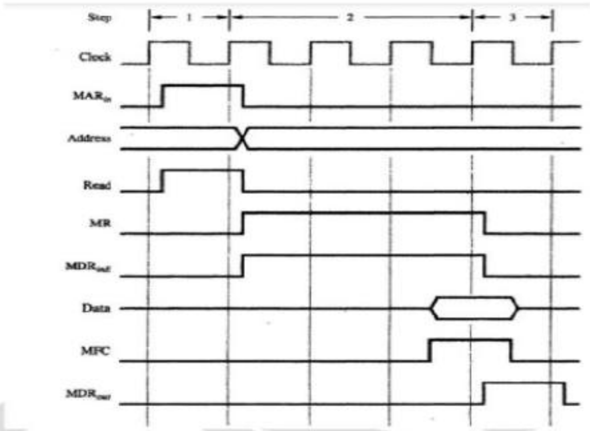


Figure 5.6: Timing of a memory Read operation

b) Explain pipelining performance.

Scheme:

Diagram: 3marks

Explanation: 7marks

Solution:

5.3.2 Pipeline performance

The pipelined processor in Figure 5.6 completes the processing of one instruction in each clock cycle, which means that the rate of instruction processing is four times that of sequential operation. The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages.

Let us consider an example of, one of the pipeline stages may not be able to complete its processing task for a given instruction in the time allotted as in **Figure 5.13.**

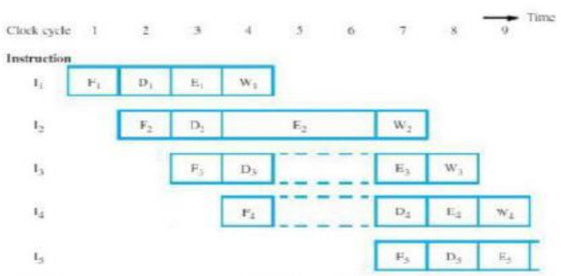


Figure 5.13: Execution unit takes more than one cycle for execution

Here instruction I2 requires three cycles to complete, from cycle 4 through cycle 6. Thus, in cycles 5 and 6, the Write stage must be told to do nothing, because it has no data to work with. Meanwhile, the information in buffer B2 must remain intact until the Execute stage has completed its operation. This means that stage 2 and, in turn, stage1 are blocked from accepting new instructions because the information in B1 cannot be overwritten. Thus, steps D4 and F5 must be postponed.

Pipelined operation in **Figure 5.13** is said to have been stalled for two clock cycles. Normal pipelined operation resumes in cycle 7. Any condition that causes the pipeline to stall is called **a hazard**.

There are three types of Hazards:

1. Data hazard

2. Instruction or control hazard

3. Structural hazard

CI CCI HOD