

## Module No-4: Memory and I/O Organization:

- Memory system overview,
- Cache memory organizations
- Techniques for reducing cache misses
- Hierarchical memory technology
- Coherence and locality properties,
- Virtual Memory
- Memory mapped IO.
- Introduction to I/O interfaces.
- Interrupts
- Interrupt hardware
- Enabling and Disabling interrupts
- Concept of handshaking,
- Polled I/O,
- Priorities, Daisy Chaining.
- Vectored interrupts
- Direct memory access, DMA controller

### Concept of Memory

We have already mentioned that digital computer works on stored programmed concept introduced by Von Neumann. We use memory to store the information, which includes both program and data.

Due to several reasons, we have different kind of memories. We use different kind of memory at different level.

The memory of computer is broadly categories into two categories:

- Internal and
- external

Internal memory is used by CPU to perform task and external memory is used to store bulk information, which includes large software and data.

Memory is used to store the information in digital form. The memory hierarchy is given by:

- Register
- Cache Memory
- Main Memory
- Magnetic Disk
- Removable media (Magnetic tape)

### Register:

This is a part of Central Processor Unit, so they reside inside the CPU. The information from main memory is brought to CPU and keep the information in register. Due to space and cost constraints, we have got a limited number of registers in a CPU. These are basically faster devices.

### Cache Memory:

Cache memory is a storage device placed in between CPU and main memory. These are semiconductor memories. These are basically fast memory device, faster than main memory.

We can not have a big volume of cache memory due to its higher cost and some constraints of the CPU. Due to higher cost we can not replace the whole main memory by faster memory. Generally, the most recently used information is kept in the cache memory. It is brought from the main memory and placed in the cache memory. Now a days, we get CPU with internal cache.

### Main Memory:

Like cache memory, main memory is also semiconductor memory. But the main memory is relatively slower memory. We have to first bring the information (whether it is data or program), to main memory. CPU can work with the information available in main memory only.

### Magnetic Disk:

This is bulk storage device. We have to deal with huge amount of data in many application. But we don't have so much semiconductor memory to keep these information in our computer. On the other hand, semiconductor memories are volatile in nature. It loses its content once we switch off the computer. For permanent storage, we use magnetic disk. The storage capacity of magnetic disk is very high.

### Removable media:

For different application, we use different data. It may not be possible to keep all the information in magnetic disk. So, which ever data we are not using currently, can be kept in removable media. Magnetic tape is one kind of removable medium. CD is also a removable media, which is an optical device.

Register, cache memory and main memory are internal memory. Magnetic Disk, removable media are external memory. Internal memories are semiconductor memory. Semiconductor memories are categorized as volatile memory and non-volatile memory.

**RAM:** Random Access Memories are **volatile** in nature. As soon as the computer is switched off, the contents of memory are also lost.

**ROM:** Read only memories are **non volatile** in nature. The storage is permanent, but it is read only memory. We can not store new information in ROM.

Several types of ROM are available:

- **PROM:** Programmable Read Only Memory; it can be programmed once as per user requirements.
- **EPROM:** Erasable Programmable Read Only Memory; the contents of the memory can be erased and store new data into the memory. In this case, we have to erase whole information.

- **EEPROM:** Electrically Erasable Programmable Read Only Memory; in this type of memory the contents of a particular location can be changed without effecting the contents of other location.

## Main Memory

The main memory of a computer is semiconductor memory. The main memory unit of computer is basically consists of two kinds of memory:

**RAM :** Random access memory; which is volatile in nature.

**ROM :** Read only memory; which is non-volatile.

The permanent information are kept in ROM and the user space is basically in RAM.

The smallest unit of information is known as bit (binary digit), and in one memory cell we can store one bit of information. 8 bit together is termed as a byte.

The maximum size of main memory that can be used in any computer is determined by the addressing scheme.

A computer that generates 16-bit address is capable of addressing upto  $2^{16}$  which is equal to 64K memory location. Similarly, for 32 bit addresses, the total capacity will be  $2^{32}$  which is equal to 4G memory location.

In some computer, the smallest addressable unit of information is a memory word and the machine is called word-addressable.

In some computer, individual address is assigned for each byte of information, and it is called **byte-addressable computer**. In this computer, one memory word contains one or more memory bytes which can be addressed individually.

A byte addressable 32-bit computer, each memory word contains 4 bytes. A possible way of address assignment is shown in figure3.1. The address of a word is always integer multiple of 4.

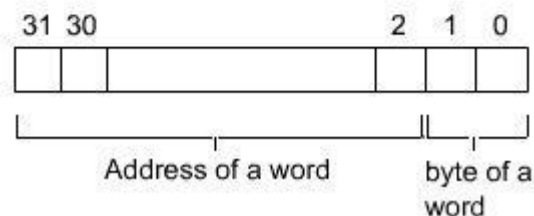
The main memory is usually designed to store and retrieve data in word length quantities. The word length of a computer is generally defined by the number of bits actually stored or retrieved in one main memory access.

Consider a machine with 32 bit address bus. If the word size is 32 bit, then the high order 30

Word Address	Byte Address			
0	0	1	2	3
4	4	5	6	7
8	8	9	10	11
12	12	13	14	15
⋮	⋮	⋮	⋮	⋮

Organization of the main memory in a 32-bit byte addressable computer

32 bit address bus/word size is 32 bit



**Figure 3.1:**Address assignment to a  
4-byte word

bit will specify the address of a word. If we want to access any byte of the word, then it can be specified by the lower two bit of the address bus.

The data transfer between main memory and the CPU takes place through two CPU registers.

- **MAR :** Memory Address Register
- **MDR :** Memory Data Register.

If the MAR is  $k$ -bit long, then the total addressable memory location will be  $2^k$ .

If the MDR is  $n$ -bit long, then the  $n$  bit of data is transferred in one memory cycle.

The transfer of data takes place through memory bus, which consist of address bus and data bus. In the above example, size of data bus is  $n$ -bit and size of address bus is  $k$  bit.

It also includes control lines like Read, Write and Memory Function Complete (MFC) for coordinating data transfer. In the case of byte addressable computer, another control line to be added to indicate the byte transfer instead of the whole word.

For memory operation, the CPU initiates a memory operation by loading the appropriate data i.e., address to MAR.

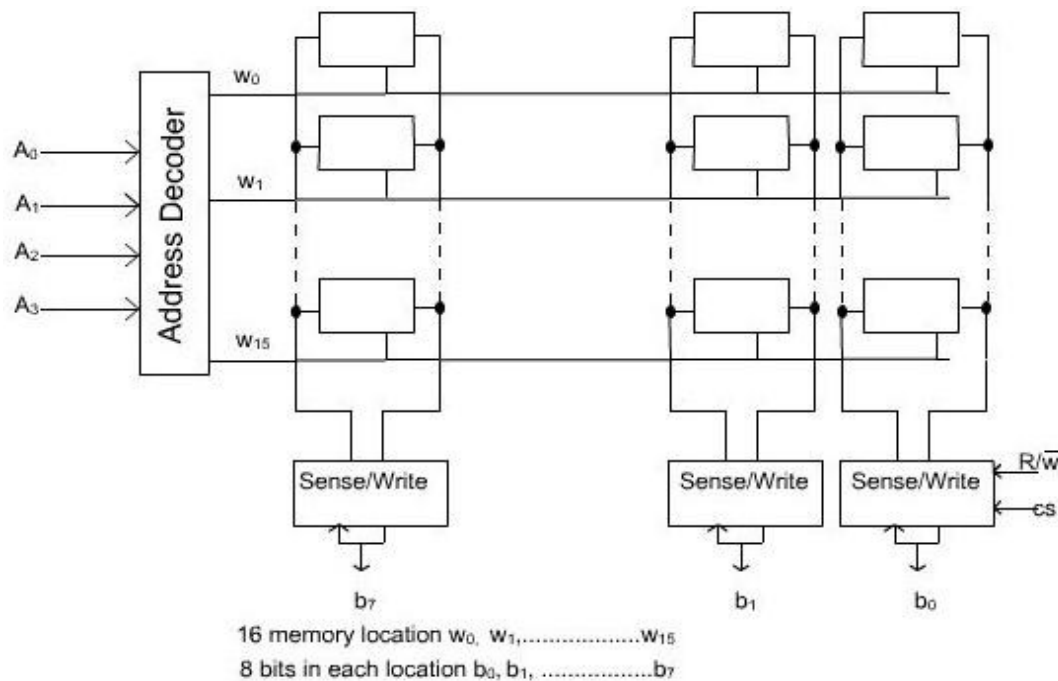
If it is a memory read operation, then it sets the read memory control line to 1. Then the contents of the memory location is brought to MDR and the memory control circuitry indicates this to the CPU by setting MFC to 1.

If the operation is a memory write operation, then the CPU places the data into MDR and sets the write memory control line to 1. Once the contents of MDR are stored in specified memory location, then the memory control circuitry indicates the end of operation by setting MFC to 1.

A useful measure of the speed of memory unit is the time that elapses between the initiation of an operation and the completion of the operation (for example, the time between Read and MFC). This is referred to as **Memory Access Time**. Another measure is memory cycle time. This is the minimum time delay between the initiation two independent memory operations (for example, two successive memory read operation). Memory cycle time is slightly larger than memory access time.

### Internal Organization of Memory Chips

A memory cell is capable of storing 1-bit of information. A number of memory cells are organized in the form of a matrix to form the memory chip. One such organization is shown in the Figure 3.5.



**Figure 3.5: 16 X 8 Memory Organization**

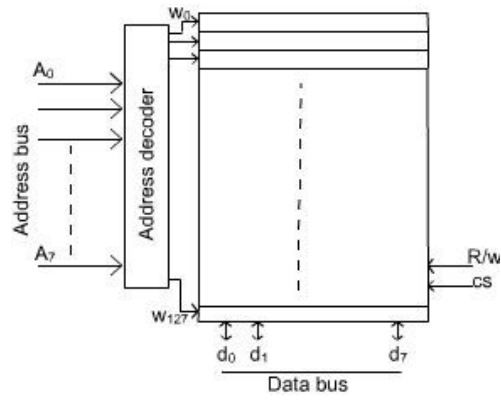
Each row of cells constitutes a memory word, and all cells of a row are connected to a common line which is referred to as word line. An address decoder is used to drive the word line. At a particular instant, one word line is enabled depending on the address present in the address bus. The cells in each column are connected by two lines. These are known as bit lines. These bit lines are connected to data input line and data output line through a Sense/Write circuit. During a Read operation, the Sense/Write circuit senses, or reads the information stored in the cells selected by a word line and transmits this information to the output data line. During a write operation, the sense/write circuit receives information and stores it in the cells of the selected word.

A memory chip consisting of 16 words of 8 bits each, usually referred to as **16 x 8 organization**. The data input and data output line of each Sense/Write circuit are connected to a single bidirectional data line in order to reduce the pins required. For 16 words, we need an address bus of size 4. In addition to address and data lines, two control lines,  $R/\overline{W}$  and CS, are provided. The  $R/\overline{W}$  line is used to specify the required operation about read or write. The CS (Chip Select) line is required to select a given chip in a multi-chip memory system.

**Consider a slightly larger memory unit that has 1K (1024) memory cells...**

**128 x 8 memory chips:**

If it is organized as a **128 x 8 memory chips**, then it has got 128 memory words of size 8 bits. So the size of data bus is 8 bits and the size of address bus is 7 bits ( $2^7 = 128$ ). The storage organization of 128 x 8 memory chip is shown in the figure 3.6.



**Figure 3.6:** 128 x 8 Memory Chip

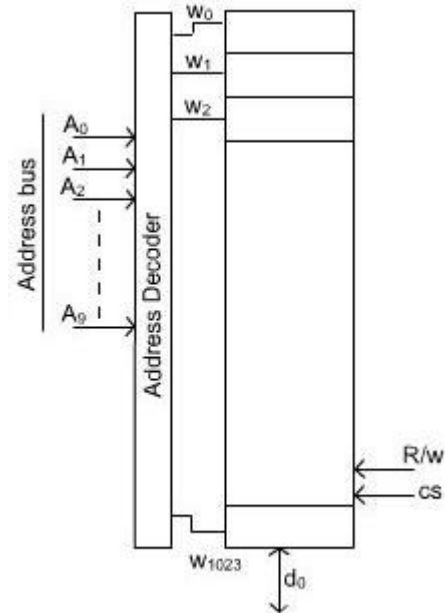
1024 x 1 memory chips:

If it is organized as a **1024 x 1 memory chips**, then it has got 1024 memory words of size 1 bit only.

Therefore, the size of data bus is 1 bit and the size of address bus is 10 bits ( $2^{10} = 1024$ ).

A particular memory location is identified by the contents of memory address bus. A decoder is used to decode the memory address. There are two ways of decoding of a memory address depending upon the organization of the memory module.

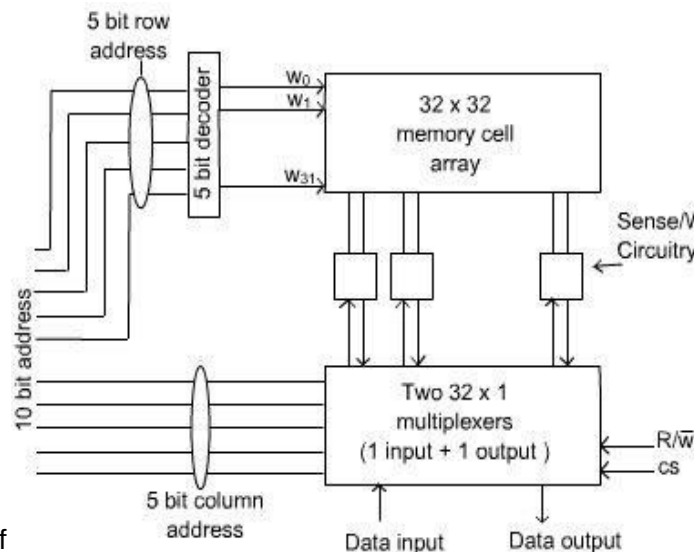
In one case, each memory word is organized in a row. In this case whole memory address bus is used together to decode the address of the specified location. The memory organization of 1024 x 1 memory chip is shown in the figure 3.7.



**Figure 3.7:** 1024 x 1 Memory chip

In second case, several memory words are organized in one row. In this case, address bus is divided into two groups.

One group is used to form the row address and the second group is used to form the column address. Consider the memory organization of 1024 x 1 memory chip. The required 10-bit address is divided into two groups of 5 bits each to form the row and column address of the cell array. A row address selects a row of 32 cells, all of which are accessed in parallel. However, according to the column address, only one of these cells is connected to the external data line via the input output multiplexers. The arrangement for row address and column address decoders is shown in the figure 3.8.

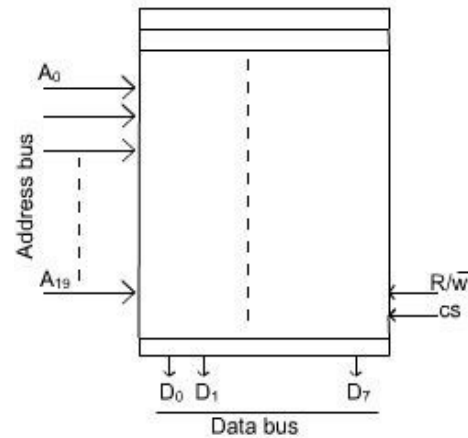


**Figure 3.8:** Organization of 1k x 1 Memory chip

The commercially available memory chips contain a much larger number of cells. As for example, a memory unit of 1MB (mega byte)

size, organised as  $1\text{M} \times 8$ , contains  $2^{20} \times 8$  memory cells. It has got 220 memory location and each memory location contains 8 bits information. The size of address bus is 20 and the size of data bus is 8.

The number of pins of a memory chip depends on the data bus and address bus of the memory module. To reduce the number of pins required for the chip, we use another scheme for address decoding. The cells are organized in the form of a square array. The address bus is divided into two groups, one for column address and other one is for row address. In this case, high- and low-order 10 bits of 20-bit address constitute of row and column address of a given cell, respectively. In order to reduce the number of pin needed for external connections, the row and column addresses are multiplexed on ten pins. During a Read or a Write operation, the row address is applied first. In response to a signal pulse on the **Row Address Strobe (RAS)** input of the chip, this part of the address is loaded into the row address latch. All cell of this particular row is selected. Shortly after the row address is latched, the column address is applied to the address pins. It is loaded into the column address latch with the help of Column Address Strobe (CAS) signal, similar to RAS. The information in this latch is decoded and the appropriate **Sense/Write** circuit is selected.

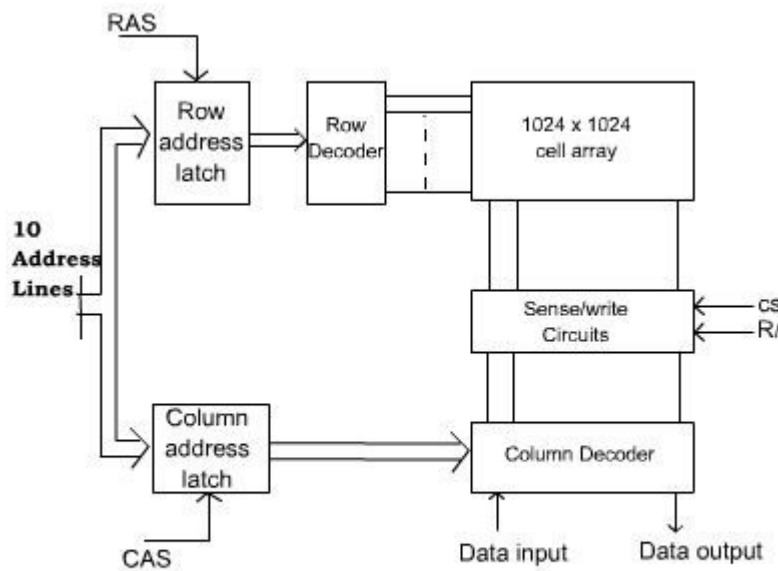


**Figure 3.9:** 1 MB(Mega Byte) Memory Chip

For a Write operation, the information at the input lines are transferred to the selected circuits.

The 1MB (Mega byte) memory chip with 20 address lines as shown in the figure 3.9. The same memory chip (1MB) with 10 address lines ( where

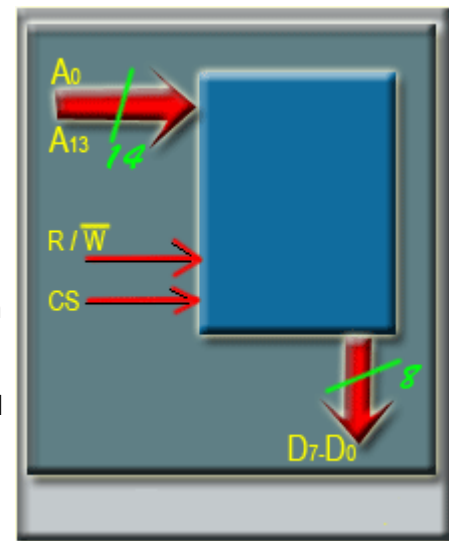




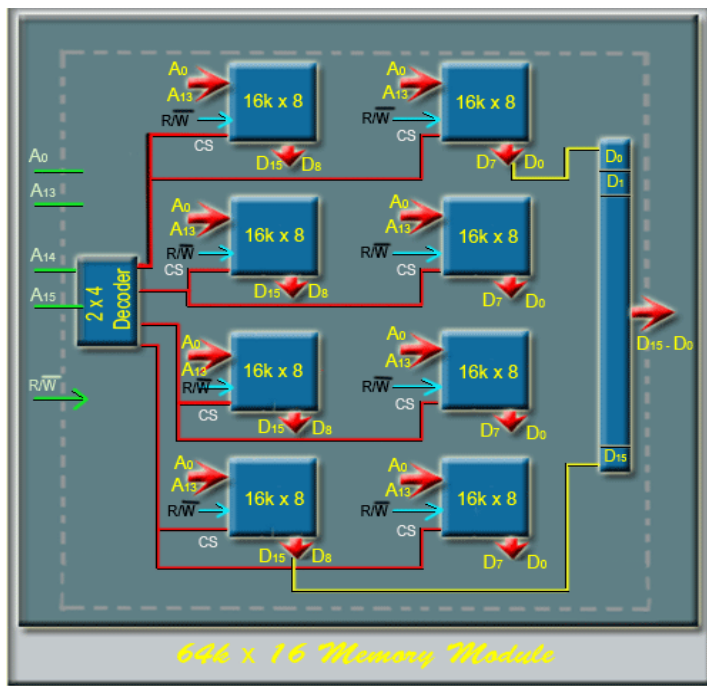
row & column address are multiplexed) is shown in Figure 3.10.

**Figure 3.10:** Organization of a 1M x 1 Memory chip.

Now we discuss the design of memory subsystem using memory chips. Consider a memory chips of capacity 16K x 8. The requirement is to design a memory subsystem of capacity 64K x 16. Each memory chip has got eight lines for data bus, but the data bus size of memory subsystem is 16 bits. The total requirement is for 64K memory location, so four such units are required to get the 64K memory location. For 64K memory location, the size of address bus is 16. On the other hand, for 16K memory location, size of address bus is 14 bits. Each chip has a control input line called Chip Select (CS). A chip can be enabled to accept data input or to place the data on the output bus by setting its Chip Select input to 1. The address bus for the 64K memory is 16 bits wide. The high order two bits of the address are decoded to obtain the four chip select control signals. The remaining 14 address bits are connected to the address lines of all the chips. They are used to access a specific location inside each chip of the selected row. The  $R/\overline{W}$  inputs of all chips are tied together to provide a common  $READ/\overline{WRITE}$  control.



**Figure 3.11:** 16k x 8 Memory chip



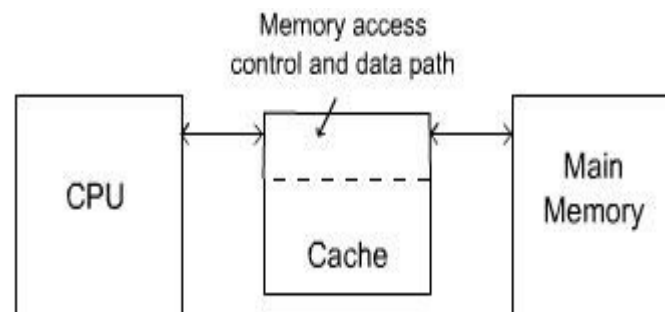
[Click on Image To View Large Image](#)

**Figure 3.12:** 64k x 16 Memory chip

The block diagram of a 16k x 8 memory chip is shown in the figure 3.11. The block diagram of a 64k x 16 memory module constructed with the help of eight 16k x 8 memory chips is shown in the figure 3.12.

## Cache Memory

Analysis of large number of programs has shown that a number of instructions are executed repeatedly. This may be in the form of a simple loops, nested loops, or a few procedures that repeatedly call each other. It is observed that many instructions in each of a few localized areas of the program are repeatedly executed, while the remainder of the program is accessed relatively less. This phenomenon is referred to as locality of reference.



**Figure 3.13:** Cache memory between CPU and the main memory

Now, if it can be arranged to have the active segments of a program in a fast memory, then the total execution time can be significantly reduced. It is the fact that CPU is a faster device and memory is a relatively slower device. Memory access is the main bottleneck for the performance efficiency. If a faster memory device can be inserted between main memory and CPU, the efficiency can be

increased. The faster memory that is inserted between CPU and Main Memory is termed as Cache memory. To make this arrangement effective, the cache must be considerably faster than the main memory, and typically it is 5 to 10 times faster than the main memory. This approach is more economical than the use of fast memory device to implement the entire main memory. This is also feasible due to the locality of reference that is present in most of the program, which reduces the frequent data transfer between main memory and cache memory. The inclusion of cache memory between CPU and main memory is shown in Figure 3.13.

### Operation of Cache Memory

The memory control circuitry is designed to take advantage of the property of locality of reference. Some assumptions are made while designing the memory control circuitry:

1. The CPU does not need to know explicitly about the existence of the cache.
2. The CPU simply makes Read and Write request. The nature of these two operations are same whether cache is present or not.
3. The address generated by the CPU always refers to location of main memory.
4. The memory access control circuitry determines whether or not the requested word currently exists in the cache.

When a Read request is received from the CPU, the contents of a block of memory words containing the location specified are transferred into the cache. When any of the locations in this block is referenced by the program, its contents are read directly from the cache.

Consider the case where the addressed word is not in the cache and the operation is a read. First the block of words is brought to the cache and then the requested word is forwarded to the CPU. But it can be forwarded to the CPU as soon as it is available to the cache, instead of the whole block to be loaded in the cache. This is called load through, and there is some scope to save time while using

During a write operation, if the address word is not in the cache, the information is written directly into the main memory. A write operation normally refers to the location of data areas and the property of locality of reference is not as pronounced in accessing data when write operation is involved. Therefore, it is not advantageous to bring the data block to the cache when there is a write operation, and the addressed word is not present in cache.

### Mapping Functions

The mapping functions are used to map a particular block of main memory to a particular block of cache. This mapping function is used to transfer the block from main memory to cache memory. Three different mapping functions are available:

#### Direct mapping:

A particular block of main memory can be brought to a particular block of cache memory. So, it is not flexible.

### **Associative mapping:**

In this mapping function, any block of Main memory can potentially reside in any cache block position. This is much more flexible mapping method.

### **Block-set-associative mapping:**

In this method, blocks of cache are grouped into sets, and the mapping allows a block of main memory to reside in any block of a specific set. From the flexibility point of view, it is in between to the other two methods.

[All these three mapping methods are explained with the help of an example.](#)

Consider a cache of 4096 (4K) words with a block size of 32 words. Therefore, the cache is organized as 128 blocks. For 4K words, required address lines are 12 bits. To select one of the block out of 128 blocks, we need 7 bits of address lines and to select one word out of 32 words, we need 5 bits of address lines. So the total 12 bits of address is divided for two groups, lower 5 bits are used to select a word within a block, and higher 7 bits of address are used to select any block of cache memory.

Let us consider a main memory system consisting 64K words. The size of address bus is 16 bits. Since the block size of cache is 32 words, so the main memory is also organized as block size of 32 words. Therefore, the total number of blocks in main memory is 2048 ( $2K \times 32 \text{ words} = 64K \text{ words}$ ). To identify any one block of 2K blocks, we need 11 address lines. Out of 16 address lines of main memory, lower 5 bits are used to select a word within a block and higher 11 bits are used to select a block out of 2048 blocks.

Number of blocks in cache memory is 128 and number of blocks in main memory is 2048, so at any instant of time only 128 blocks out of 2048 blocks can reside in cache memory. Therefore, we need mapping function to put a particular block of main memory into appropriate block of cache memory.

### **Direct Mapping Technique:**

The simplest way of associating main memory blocks with cache block is the direct mapping technique. In this technique, block  $k$  of main memory maps into block  $k \text{ modulo } m$  of the cache, where  $m$  is the total number of blocks in cache. In this example, the value of  $m$  is 128. In direct mapping technique, one particular block of main memory can be transferred to a particular block of cache which is derived by the modulo function.

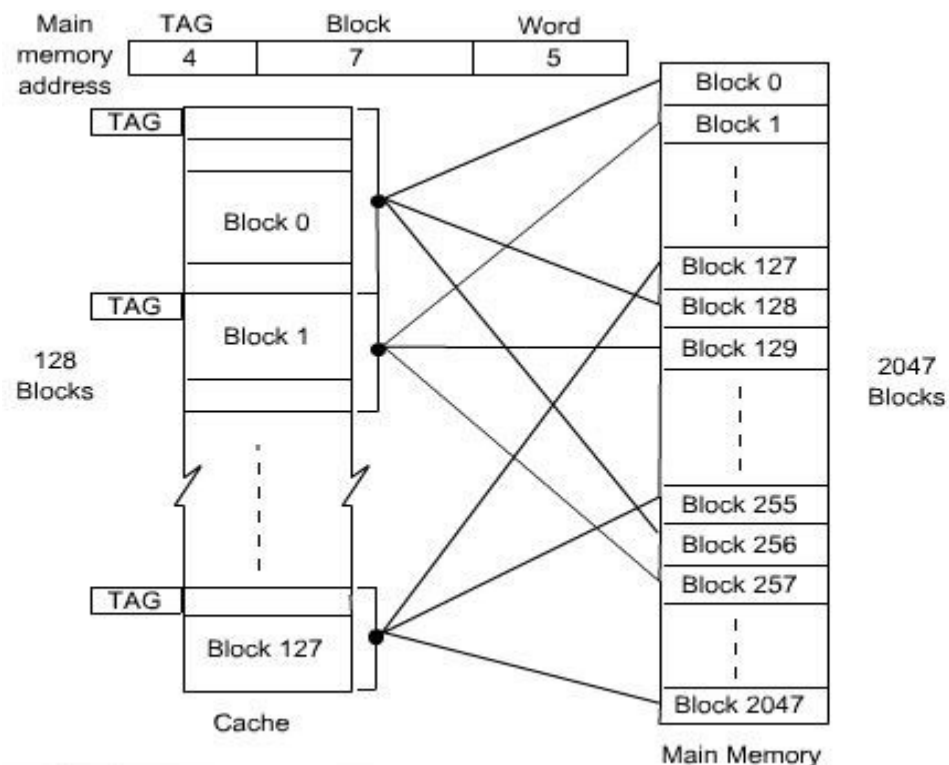
Since more than one main memory block is mapped onto a given cache block position, contention may arise for that position. This situation may occur even when the cache is not full. Contention is resolved by allowing the new block to overwrite the currently resident block. So the replacement algorithm is trivial.

The detail operation of direct mapping technique is as follows:

The main memory address is divided into three fields. The field size depends on the memory capacity and the block size of cache. In this example, the lower 5 bits of address is used to identify a word within a block. Next 7 bits are used to select a block out of 128 blocks (which is the capacity of the cache). The remaining 4 bits are used as a TAG to identify the proper block of main memory that is mapped to cache.

When a new block is first brought into the cache, the high order 4 bits of the main memory address are stored in four TAG bits associated with its location in the cache. When the CPU generates a memory request, the 7-bit block address determines the corresponding cache block. The TAG field of that block is compared to the TAG field of the address. If they match, the desired word specified by the low-order 5 bits of the address is in that block of the cache.

If there is no match, the required word must be accessed from the main memory, that is, the contents of that block of the cache is replaced by the new block that is specified by the new address generated by the CPU and correspondingly the TAG bit will also be changed by the high order 4 bits of the address. The whole arrangement for direct mapping technique is shown in the figure 3.14.

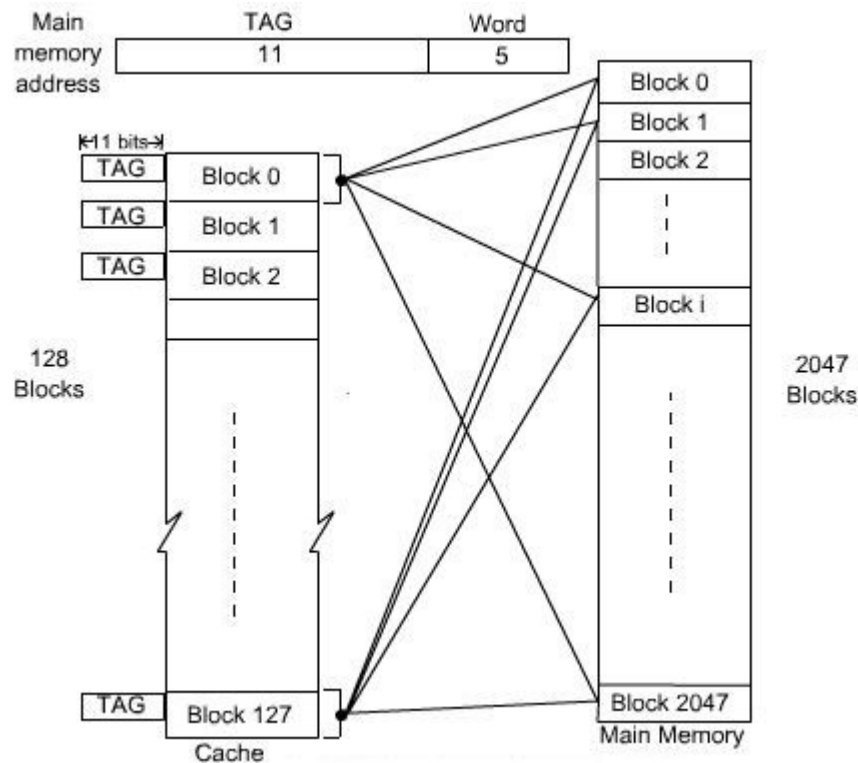


**Figure 3.14:** Direct-mapping cache

### Associated Mapping Technique:

In the associative mapping technique, a main memory block can potentially reside in any cache block position. In this case, the main memory address is divided into two groups, low-order bits identifies the location of a word within a block and high-order bits identifies the block. In the example here, 11 bits are required to identify a main memory block when it is resident in the cache, high-order 11 bits are used as TAG bits and low-order 5 bits are used to identify a word within a block. The TAG bits of an address received from the CPU must be compared to the TAG bits of each block of the cache to see if the desired block is present.

In the associative mapping, any block of main memory can go to any block of cache, so it has got the complete flexibility and we have to use proper replacement policy to replace a block from cache if the currently accessed block of main memory is not present in cache. It might not be practical to use this complete flexibility of associative mapping technique due to searching overhead, because the TAG field of main memory address has to be compared with the TAG field of all the cache block. In this example, there are 128 blocks in cache and the size of TAG is 11 bits. The whole arrangement of Associative Mapping Technique is shown in the figure 3.15.



**Figure 3.15:** Associated Mapping Cacke

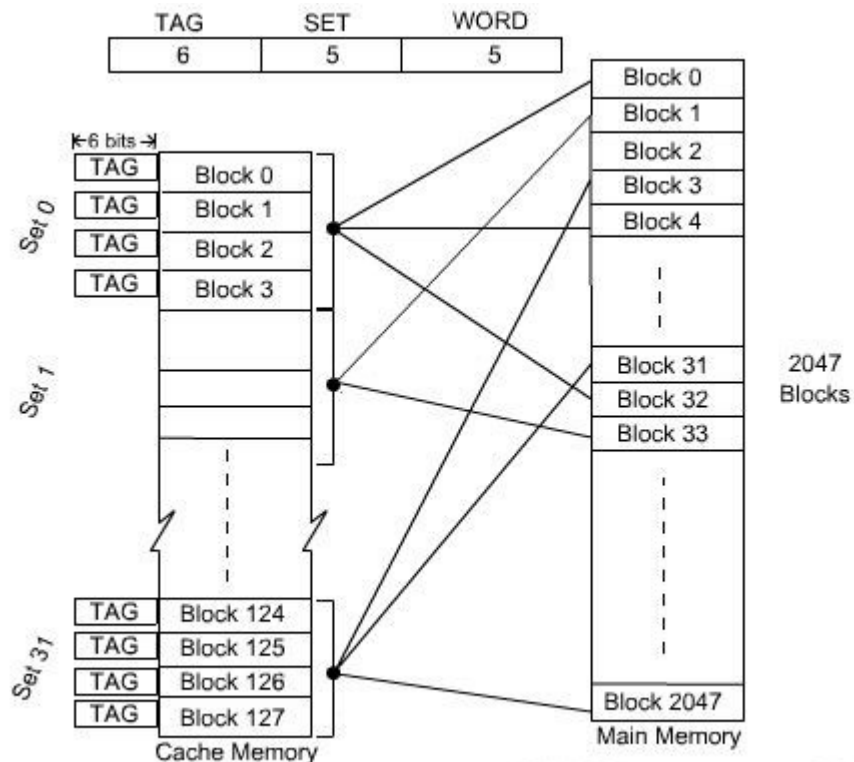
### Block-Set-Associative Mapping Technique:

This mapping technique is intermediate to the previous two techniques. Blocks of the cache are grouped into sets, and the mapping allows a block of main memory to reside in any block of a specific set. Therefore, the flexibility of associative mapping is reduced from full freedom to a set of specific blocks. This also reduces the searching overhead, because the search is restricted to number of sets, instead of number of blocks. Also the contention problem of the direct mapping is eased by having a few choices for block replacement.

Consider the same cache memory and main memory organization of the previous example. Organize the cache with 4 blocks in each set. The TAG field of associative mapping technique is divided into two groups, one is termed as SET bit and the second one is termed as TAG bit. Each set contains 4 blocks, total number of set is 32. The main memory address is grouped into three parts: low-order 5 bits are used to identifies a word within a block. Since there are total 32 sets present, next 5 bits are used to identify the set. High-order 6 bits are used as TAG bits.

The 5-bit set field of the address determines which set of the cache might contain the desired block. This is similar to direct mapping technique, in case of direct mapping, it looks for block, but in case of block-set-associative mapping, it looks for set. The TAG field of the address must then be compared with the TAGs of the four blocks of that set. If a match occurs, then the block is present in the cache; otherwise the block containing the addressed word must be brought to the cache. This block will potentially come to the cooresponding set only. Since, there are four blocks in the set, we have to choose appropriately which block to be replaced if all the blocks are occupied. Since the search is restricted to four block only, so the searching complexity is reduced. The whole arrangement of block-set-associative mapping technique is shown in the figure 3.15.

It is clear that if we increase the number of blocks per set, then the number of bits in SET field is reduced. Due to the increase of blocks per set, complexity of search is also increased. The extreme condition of 128 blocks per set requires no set bits and corresponds to the fully associative mapping technique with 11 TAG bits. The other extreme of one block per set is the direct mapping method.



**Figure 3.15:** Block-set Associated mapping Cache with 4 blocks per set

## Replacement Algorithms

When a new block must be brought into the cache and all the positions that it may occupy are full, a decision must be made as to which of the old blocks is to be overwritten. In general, a policy is required to keep the block in cache when they are likely to be referenced in near future. However, it is not easy to determine directly which of the block in the cache are about to be referenced. The property of locality of reference gives some clue to design good replacement policy.

Least Recently Used (LRU) Replacement policy:

Since program usually stay in localized areas for reasonable periods of time, it can be assumed that there is a high probability that blocks which have been referenced recently will also be referenced in the near future. Therefore, when a block is to be overwritten, it is a good decision to overwrite the one that has gone for longest time without being referenced. This is defined as the least recently used (LRU) block. Keeping track of LRU block must be done as computation proceeds.

Consider a specific example of a four-block set. It is required to track the LRU block of this four-block set. A 2-bit counter may be used for each block.

When a hit occurs, that is, when a read request is received for a word that is in the cache, the counter of the block that is referenced is set to 0. All counters which values originally lower than the referenced one are incremented by 1 and all other counters remain unchanged.

When a miss occurs, that is, when a read request is received for a word and the word is not present in the cache, we have to bring the block to cache.

There are two possibilities in case of a miss:

If the set is not full, the counter associated with the new block loaded from the main memory is set to 0, and the values of all other counters are incremented by 1.

If the set is full and a miss occurs, the block with the counter value 3 is removed, and the new block is put in its place. The counter value is set to zero. The other three block counters are incremented by 1.

It is easy to verify that the counter values of occupied blocks are always distinct. Also it is trivial that highest counter value indicates least recently used block.

### **First In First Out (FIFO) replacement policy:**

A reasonable rule may be to remove the oldest from a full set when a new block must be brought in. While using this technique, no updation is required when a hit occurs. When a miss occurs and the set is not full, the new block is put into an empty block and the counter values of the occupied block will be incremented by one. When a miss occurs and the set is full, the block with highest counter value is replaced by new block and counter is set to 0, counter value of all other blocks of that set is incremented by 1. The overhead of the policy is less, since no updation is required during hit.

### **Random replacement policy:**

The simplest algorithm is to choose the block to be overwritten at random. Interestingly enough, this simple algorithm has been found to be very effective in practice.

### **Reducing Cache Miss Rate**

- *Compiler-controlled prefetch*
  - o *An alternative to hardware prefetching.*
  - o *Some CPUs include prefetching instructions .*
    - *These instructions request that data be moved into either a register or cache.*
  - o *These special instructions can either be faulting or non-faulting .*
    - *Non-faulting instructions do nothing (no-op) if the memory access would cause an exception.*



- o Of course, prefetching does **not** help if it interferes with normal CPU memory access or operation.

- Thus, the cache must be **nonblocking** (also called **lockup-free** ).

- o This allows the CPU to overlap execution with the prefetching of data.

#### Reducing Cache Miss Rate

- **Compiler-controlled prefetch**

- o While this approach yields better prefetch "hit" rates than hardware prefetch, it does so at the expense of executing more instructions.

- o Thus, the compiler tends to concentrate on prefetching data that are likely to be cache misses anyway.

- Loops are key targets since they operate over large data spaces and their data accesses can be inferred from the loop index in advance.

- **Compiler optimizations**

- o This method does NOT require any hardware modifications.

- Yet it can be the most efficient way to eliminate cache misses.

- o The improvement results from better code and data organizations.


- For example, code can be rearranged to avoid conflicts in a direct-mapped cache, and accesses to arrays can be reordered to operate on **blocks of data** rather than processing **rows** of the array.

#### Reducing Cache Miss Rate

- **Compiler optimizations**

- o **Merging arrays**

- This method combines two separate arrays (that might conflict for a single block in the cache) into a single interleaved array.

<pre>int val[SIZE]; int key[SIZE];</pre>		<pre>struct merge {     int val;     int key; }; struct merge MA[SIZE];</pre>
--	---	---

- This brings together corresponding elements in both arrays, which are likely to be referenced together.

- Reorganizing and fetching them at the same time can reduce misses.

- This technique reduces misses by improving spatial locality.

#### Reducing Cache Miss Rate

- **Compiler optimizations**

### o Loop interchange

- By switching the order in which loops execute, misses can be reduced due to improvements in spatial locality.
- For example,

```
for (i = 0; i < 100; i++) {  
    for (j = 0; j < 100; j++) {  
        a[j][i] = a[j][i] * 2;  
    }  
}
```

- These loops cause a miss on each memory access because of the long stride given by index *j* in the inner loop.
- By switching the order of the loops, the stride is changed to 1, allowing the elements to be accessed in sequential order.

Reducing Cache Miss Rate

### • Compiler optimizations

#### o Loop Fusion

- Many programs have separate loops that operate on the same data.
- Combining these loops allows a program to take advantage of **temporal locality** by grouping operations on the same (cached) data together.

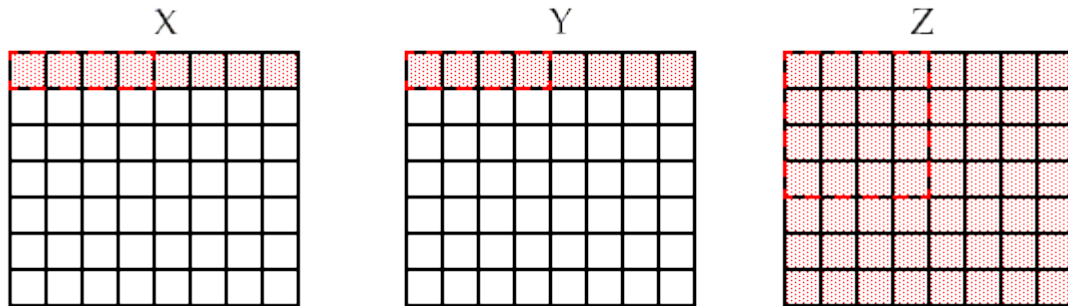
#### o Blocking

- The above methods work well on array accesses that occur along one dimension only.
- However, loops that access both rows and columns, such as matrix multiplication, are problems.

Reducing Cache Miss Rate

#### o Blocking

- Unoptimized matrix multiplication requires the cache to hold the minimum elements shown shaded below.



Data accessed to compute a row of X using  $X = Y * Z$

- Capacity misses can occur for large matrices since it may not be possible to store all the elements of Z in the cache.
- Blocking operates on blocks (submatrices) as shown by the dotted line, and reduces the total number of memory words accessed by a factor of B (the blocking factor).  
Reducing Cache Miss Rate
- Compiler optimizations
  - o **Blocking**
    - Therefore, matrix multiplication is performed by multiplying the submatrices first.
    - Matrix Y benefits from spatial locality and Z benefits from temporal locality.
    - This method is also used to reduce the number of blocks that must be transferred between disk and main memory.
    - Therefore, the technique is effective for several levels of the hierarchy.
- Given the increasing speed gap in processor speed and memory access times, these last two techniques will only increase in importance over time.  
Reducing Cache Miss Penalty
- Giving read misses priority
  - o If a system has a write buffer, writes can be delayed to come after reads.
  - o The system must, however, be careful to check the write buffer to see if the value being read is about to be written.

```
SW 512(R0), R3
LW R1, 1024(R0)
LW R2, 512(R0)
```

Assume the write-through  
cache maps 512 to the same  
cache location as 1024.  
**Will R2 = R3 ?**

- o A simple method of dealing with this problem:
  - Stall reads until the write buffer is empty.

- o However, this method increases the read miss penalty considerably since the write buffer in write-through is likely to have blocks waiting to be written.

- o An alternative is to check the write buffer for conflicts.

- In cases like this, the write buffer acts as a victim cache.

Reducing Cache Miss Penalty

- Using subblocks to reduce fetch time

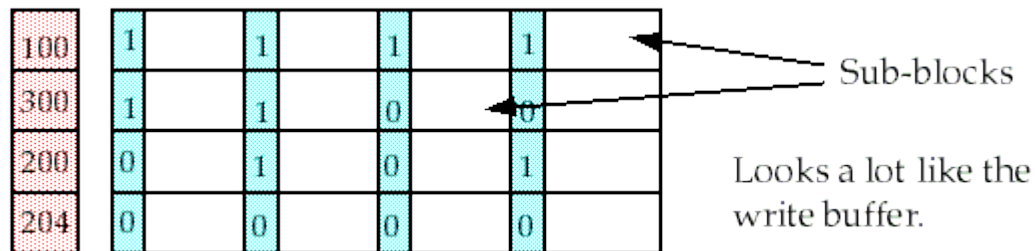
- o Tags can hurt performance by occupying too much space or by slowing down caches.

- o Using large blocks reduces the amount of storage for tags (and makes them shorter), optimizing space on the chip.

- o This may even reduce **miss rate** by reducing compulsory misses.

- However, the miss penalty for large blocks is high, since the entire block must be moved between the cache and memory.

- o The solution is to divide each block into subblocks, each of which has a valid bit.



Reducing Cache Miss Penalty

- Using subblocks to reduce fetch time

- o The tag is valid for the entire block, but only a sub-block needs to be read on a miss.
- Therefore, a block can no longer be defined as the minimum unit transferred between cache and memory.

- o This results in a smaller miss penalty.

- Early restart & critical word first

- o This strategy does NOT require extra hardware (like the previous two techniques).

- o It optimizes the order in which the words of a block are fetched and when the desired word is delivered to the CPU.

- o

- Early restart

- o With early restart, the CPU gets its data (and thus resumes execution) as soon as it arrives in the cache without waiting for the rest of the block.

Reducing Cache Miss Penalty

- Early restart & critical word first

- Critical word first

o *Instead of starting the fetch of a block with the first word, the cache can fetch the requested word first and then fetch the rest afterwards.*

o *In conjunction with early restart, this reduces the miss penalty by allowing the CPU to continue execution while most of the block is still being fetched.*

- *Nonblocking caches*

o *A **nonblocking** cache, in conjunction with out-of-order execution, can allow the CPU to continue executing instructions after a data cache miss.*

- *The cache continues to supply hits while processing read misses ( hit under miss ).*

- *The instruction needing the missed data waits for the data to arrive.*

#### *Reducing Cache Miss Penalty*

- *Nonblocking caches*

o *Complex caches can even have multiple outstanding misses ( miss under miss ).*

- *But this greatly increases cache complexity.*

- *Second-level caches*

o *This method focuses on the interface between the cache and main memory.*

o *We can add an second-level cache between main memory and a small, fast first-level cache.*

- *This helps satisfy the desire to make the cache fast and large.*

o *The second-level cache allows:*

- *The smaller first-level cache to fit on the chip with the CPU and fast enough to service requests in one or two CPU clock cycles.*

- *Hits for many memory accesses that would go to main memory, lessening the effective miss penalty.*

#### *Reducing Cache Miss Penalty*

- *Second-level caches*

- **Performance of a multi-level cache:**

- o *The performance of a two-level cache is calculated in a similar way to the performance for a single level cache.*

$$\text{Avg mem access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}$$

$$\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

- o *So the miss penalty for level 1 is calculated using the hit time, miss rate, and miss penalty for the level 2 cache.*

- o *For two level caches, there are two miss rates:*

- *Global miss rate*

- The number of misses in the cache divided by the total number of memory accesses generated by the CPU ( $\text{Miss rate}_{L1} \times \text{Miss rate}_{L2}$ ).
  - The number of misses in the cache divided by the total number of memory accesses to this cache ( $\text{Miss rate}_{L2}$  for the 2nd-level cache).

#### Reducing Cache Miss Penalty

- **Performance of a multi-level cache:**

- o Note that the local miss rate for L2 is high because it's only getting the misses from the L1 cache (instead of all memory accesses).

- o In general, the global miss rate is a more useful measure since it indicates what fraction of the memory accesses that leave the CPU go all the way to memory.

- **Desirable characteristics for an L2 cache:**

- Much larger than the L1 cache
  - Since L2 contains the same data as L1, making L2 about the same size as L1 causes it to have a high local miss rate.
  - This is true since if we miss in L1, it is likely that we'll miss in L2 as well resulting in performance that is not much better than using main memory alone.
  - Therefore, it must be much larger.

#### Reducing Cache Miss Penalty

- **Desirable characteristics for an L2 cache:**

- Higher associativity
  - The main reason for low associativity was fast, small caches.
  - The L2 cache need be **neither** , and will benefit from the higher hit rate that more blocks per set provides.
- Larger block size
  - This has the advantage of reducing compulsory misses that must go all the way to main memory.
  - Since the L2 cache is large, the effect of increasing conflict misses (as is true for a smaller cache) is minimal.

#### Reducing Cache Miss Penalty

- **Inclusion**

- o If all of the data in the L1 cache is also in the L2 cache, the L2 cache has the multilevel inclusion property .

- o *Most caches enforce this property since it is easier to deal with cache consistency.*
  - *Consistency between I/O and caches (and between caches in a multiprocessor) can be determined by checking second-level cache.*

- **Design of L1 and L2 caches**

- o *Although they can be designed separately, it is often helpful to know if there is going to be an L2 cache.*
- o *For example, write-through in L1 is much more effective if there is an L2 writeback cache to buffer repeated writes.*
- o *Similarly, a direct-mapped L1 cache can work fine if the L2 cache satisfies most of the conflict misses.*

#### Reducing Cache Miss Penalty

- **L2 cache summary**

- o *In general, cache design trades fast hits for few misses.*
- o *For an L1 cache, fast hits are more important.*
- o *For L2, there are many fewer hits, so fewer misses becomes more important.*
- o *Therefore, larger caches with higher associativity and larger blocks are beneficial in L2 caches.*

## Main Memory

The main working principle of digital computer is [Von-Neumann](#) stored program principle. First of all we have to keep all the information in some storage, mainly known as main memory, and CPU interacts with the main memory only. Therefore, memory management is an important issue while designing a computer system.

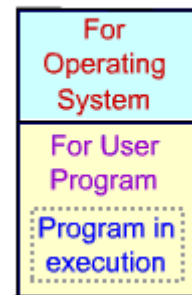
On the otherhand, everything cannot be implemented in hardware, otherwise the cost of system will be very high. Therefore some of the tasks are performed by software program. Collection of such software programs are basically known as operating systems. So operating system is viewed as extended machine. Many more functions or instructions are implemented through software routine. The operating system is mainly memory resistant, i.e., the operating system is loaded into main memory.

Due to that, the main memory of a computer is divided into two parts. One part is reserved for operating system. The other part is for user program. The program currently being executed by the CPU is loaded into the user part of the memory. The two parts of the main memory are shown in the figure 3.17.

In a uni-programming system, the program currently being executed is loaded into the user part of the memory.

In a multiprogramming system, the user part of memory is subdivided to accomodate multiple process. The task of subdivision is carried out dynamically by opearting system and is known as memory management.

### Main Memory



**Figure 3.17 :** Partition of main memory.

Efficient memory management is vital in a multiprogramming system. If only a few process are in memory, then for much of the time all of the process will be waiting for I/O and the processor will idle. Thus memory needs to be allocated efficiently to pack as many processes into main memory as possible.

When memory holds multiple processes, then the process can move from one process to another process when one process is waiting. But the processor is so much faster then I/O that it will be common for all the processes in memory to be waiting for I/O. Thus, even with multiprogramming, a processor could be idle most of the time.

### Memory Management



In an uniprogramming system, main memory is divided into two parts : one part for the **operating system** and the other part for the **program currently being executed**.

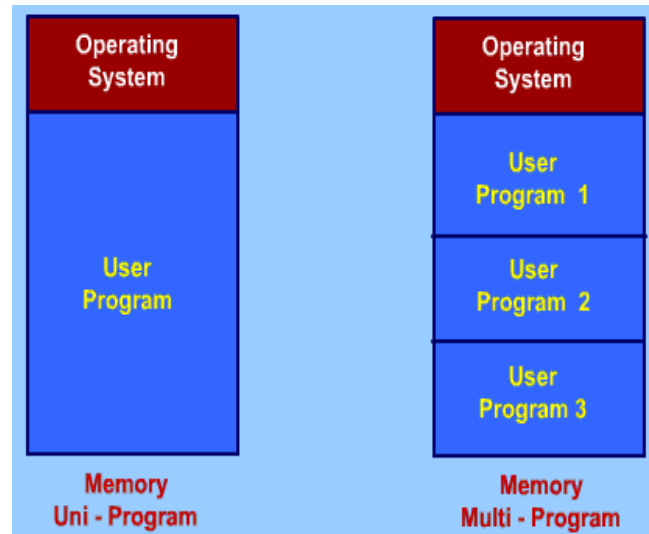
In multiprogramming system, the user part of memory is subdivided to accomodate multiple processes.

The task of subdivision is carried out dynamically by the operating system and is known as **memory management**.

In uniprogramming system, only one program is in execution. After completion of one program, another program may start.

In general, most of the programs involve I/O operation. It must take input from some input device and place the result in some output device.

Partition of main memory for uni-program and multi program is shown in figure 3.19.



**Figure 3.19:** Partition of Main Memory

To utilize the idle time of CPU, we are shifting the paradigm from uniprogram environment to multiprogram environment.

Since the size of main memory is fixed, it is possible to accomodate only few process in the main memory. If all are waiting for I/O operation, then again CPU remains idle.

To utilize the idle time of CPU, some of the process must be off loaded from the memory and new process must be brought to this memory place. This is known **swapping**.

### **What is swapping :**

1. The process waiting for some I/O to complete, must stored back in disk.
2. New ready process is swapped in to main memory as space becomes available.
3. As process completes, it is moved out of main memory.
4. If none of the processes in memory are ready,

- Swapped out a block process to intermediate queue of blocked process.
- Swapped in a ready process from the ready queue.

But swapping is an I/O process, so it also takes time. Instead of remain in idle state of CPU, sometimes it is advantageous to swapped in a ready process and start executing it. The main question arises where to put a new process in the main memory. It must be done in such a way that the memory is utilized properly.

## Partitioning

Splitting of memory into sections to allocate processes including operating system. There are two scheme for partitioning :

- o Fixed size partitions
- o Variable size partitions

### Fixed sized partitions:

The memory is partitioned to fixed size partition. Although the partitions are of fixed size, they need not be of equal size.

There is a problem of wastage of memory in fixed size even with unequal size. When a process is brought into memory, it is placed in the smallest available partition that will hold it.

Equal size and unequal size partition of fixed size partitions of main memory is shown in Figure 3.20.

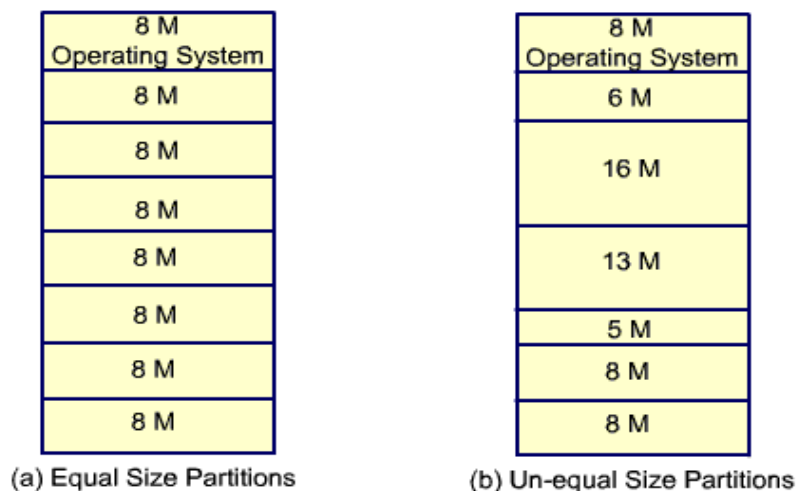


Figure 3.20: Equal and unequal size partition.

Even with the use of unequal size of partitions, there will be wastage of memory. In most cases, a process will not require exactly as much memory as provided by the partition.

For example, a process that require 5-MB of memory would be placed in the 6-MB partition which is the smallest available partition. In this partition, only 5-MB is used, the remaining 1-MB can not be used by any other process, so it is a wastage. Like this, in every partition we may have some unused memory. *The unused portion of memory in each partition is termed as hole.*

### Variable size Partition:

When a process is brought into memory, it is allocated exactly as much memory as it requires and no more. In this process it leads to a hole at the end of the memory, which is too small to use. It seems that there will be only one hole at the end, so the waste is less.

But, this is not the only hole that will be present in variable size partition. When all processes are blocked then swap out a process and bring in another process. The new swapped in process may be smaller than the swapped out process. Most likely we will not get two processes of same size. So, it will create another hole. If the swap-out and swap-in is occurring more time, then more and more holes will be created, which will lead to more wastage of memory.

There are two simple ways to slightly remove the problem of memory wastage:

**Coalesce** : Join the adjacent holes into one large hole, so that some process can be accommodated into the hole.

**Compaction** : From time to time go through memory and move all holes into one free block of memory.

During the execution of process, a process may be swapped in or swapped out many times. It is obvious that a process is not likely to be loaded into the same place in main memory each time it is swapped in. Furthermore, if compaction is used, a process may be shifted while in main memory.

A process in memory consists of instructions plus data. The instruction will contain addresses for memory locations of two types:

- o Address of data item
- o Address of instructions used for branching instructions

These addresses will change each time a process is swapped in. To solve this problem, a distinction is made between logical address and physical address.

- o *Logical address is expressed as a location relative to the beginning of the program. Instructions in the program contain only logical address.*
- o *Physical address is an actual location in main memory.*

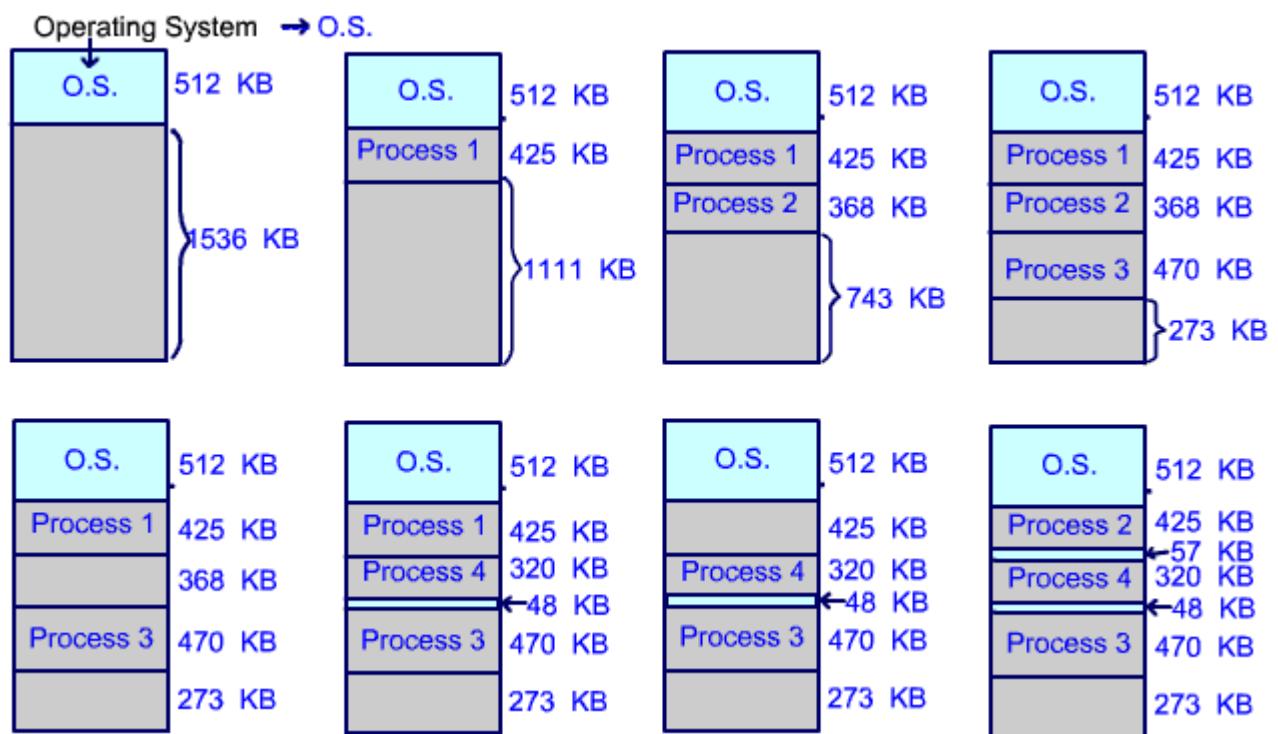
When the processor executes a process, it automatically converts from logical to physical address by adding the current starting location of the process, called its base address to each logical address.

Every time the process is swapped in to main memory, the base address may be different depending

on the allocation of memory to the process.

Consider a main memory of 2-MB out of which 512-KB is used by the Operating System. Consider three process of size 425-KB, 368-KB and 470-KB and these three process are loaded into the memory. This leaves a hole at the end of the memory. That is too small for a fourth process. At some point none of the process in main memory is ready. The operating system swaps out process-2 which leaves sufficient room for new process of size 320-KB. Since process-4 is smaller then process-2, another hole is created. Later a point is reached at which none of the processes in the main memory is ready, but proces-2, so process-1 is swapped out and process-2 is swapped in there. It will create another hole. In this way it will create lot of small holes in the momory system which will lead to more memory wastage.

The effect of dynamic partitionining that careates more whole during the execution of processes is shown in the Figure 3.21.



**Figure 3.21:** The effect of dynamic partitioning

## Virtual Memory

### Paging

Both unequal fixed size and variable size partitions are inefficient in the use of memory. It has been observed that both schemes lead to memory wastage. Therefore we are not using the memory efficiently.

There is another scheme for use of memory which is known as **paging**. In this scheme,

*The memory is partitioned into equal fixed size chunks that are relatively small. This chunk of memory is known as **frames or page frames**.*

*Each process is also divided into small fixed chunks of same size. The chunks of a program is known as **pages**.*

A page of a program could be assigned to available page frame.

In this scheme, the wastage space in memory for a process is a fraction of a page frame which corresponds to the last page of the program.

At a given point of time some of the frames in memory are in use and some are free. The list of free frame is maintained by the operating system.

Process A , stored in disk , consists of pages . At the time of execution of the process A, the operating system finds six free frames and loads the six pages of the process A into six frames.

These six frames need not be contiguous frames in main memory. The operating system maintains a page table for each process.

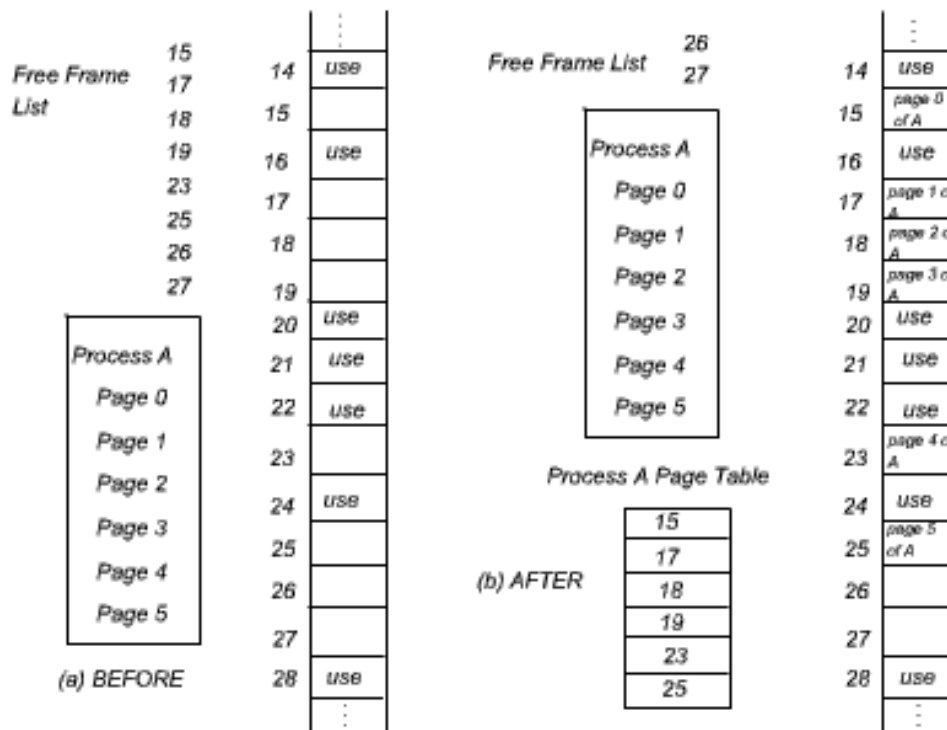
Within the program, each logical address consists of page number and a relative address within the page.

In case of simple partitioning, a logical address is the location of a word relative to the beginning of the program; the processor translates that into a physical address.

With paging, a logical address is a location of the word relative to the beginning of the page of the program, because the whole program is divided into several pages of equal length and the length of a page is same with the length of a page frame.

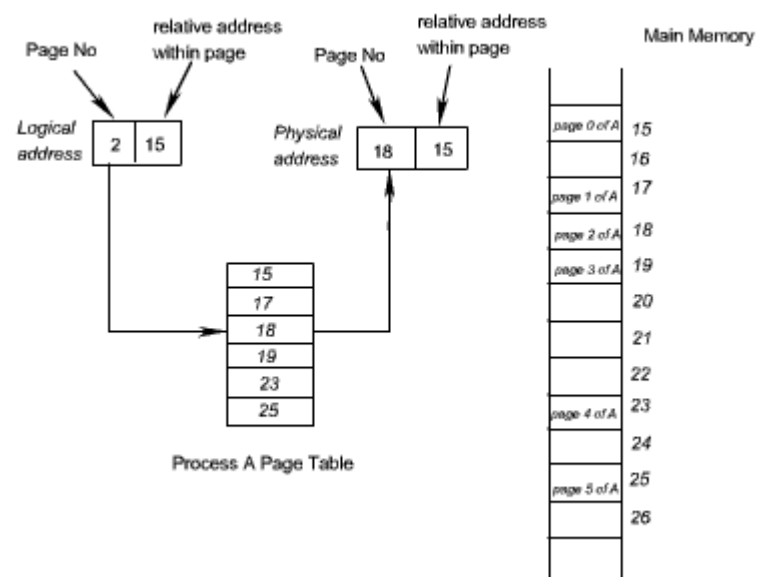
A logical address consists of page number and relative address within the page, the process uses the page table to produce the physical address which consists of frame number and relative address within the frame.

The Figure 3.22 shows the allocation of frames to a new process in the main memory. A page table is maintained for each process. This page table helps us to find the physical address in a frame which corresponds to a logical address within a process.



**Figure 3.22:** Allocation of free frames

The conversion of logical address to physical address is shown in the figure for the Process A.



**Figure 3.23:** Translation of Logical Address to Physical

Address

This approach solves the problems. Main memory is divided into many small equal size frames. Each process is divided into frame size pages. Smaller process requires fewer pages, larger process requires more. When a process is brought in, its pages are loaded into available frames and a page table is set up.

The translation of logical addresses to physical address is shown in the Figure 3.23.

## Virtual Memory

The concept of paging helps us to develop truly effective multiprogramming systems.

Since a process need not be loaded into contiguous memory locations, it helps us to put a page of a process in any free page frame. On the other hand, it is not required to load the whole process to the main memory, because the execution may be confined to a small section of the program. (eg. a subroutine).

It would clearly be wasteful to load in many pages for a process when only a few pages will be used before the program is suspended.

Instead of loading all the pages of a process, each page of process is brought in only when it is needed, i.e on demand. This scheme is known as **demand paging** .

Demand paging also allows us to accommodate more process in the main memory, since we are not going to load the whole process in the main memory, pages will be brought into the main memory as and when it is required.

With demand paging, it is not necessary to load an entire process into main memory.

This concept leads us to an important consequence ? It is possible for a process to be larger than the size of main memory. So, while developing a new process, it is not required to look for the main memory available in the machine. Because, the process will be divided into pages and pages will be brought to memory on demand.

Because a process executes only in main memory, so the main memory is referred to as real memory or physical memory.

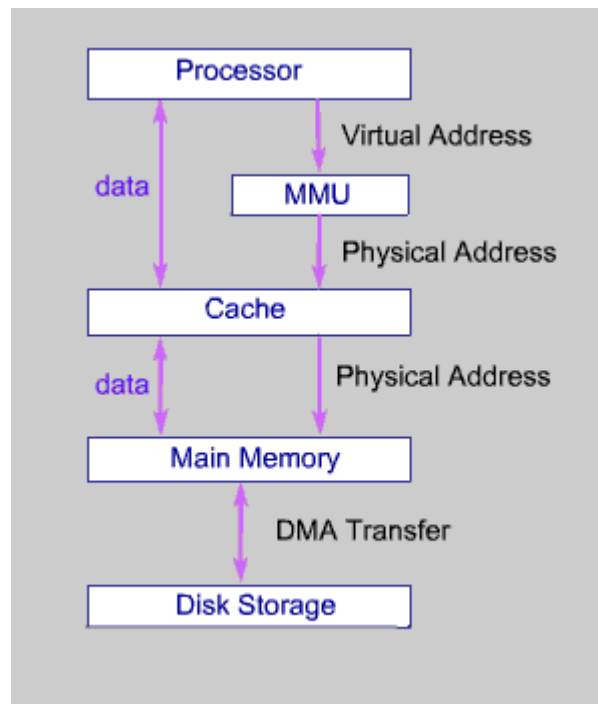
A programmer or user perceives a much larger memory that is allocated on the disk. This memory is referred to as virtual memory. The program enjoys a huge virtual memory space to develop his or her program or software.

The execution of a program is the job of operating system and the underlying hardware. To improve the performance some special hardware is added to the system. This hardware unit is known as Memory Management Unit (MMU).

In paging system, we make a page table for the process. Page table helps us to find the physical address from virtual address.

The virtual address space is used to develop a process. The special hardware unit, called **Memory Management Unit (MMU)** translates virtual address to physical address. When the desired data is in the main memory, the CPU can work with these data. If the data are not in the main memory, the MMU causes the operating system to bring into the memory from the disk.

A typical virtual memory organization is shown in the Figure 3.24.



**Figure 3.24:** Virtual Memory Organization

### Address Translation

The basic mechanism for reading a word from memory involves the translation of a virtual or logical address, consisting of page number and offset, into a physical address, consisting of frame number and offset, using a page table.

There is one page table for each process. But each process can occupy huge amount of virtual memory. But the virtual memory of a process cannot go beyond a certain limit which is restricted by the underlying hardware of the MMU. One of such component may be the size of the virtual address register.

The sizes of pages are relatively small and so the size of page table increases as the size of process increases. Therefore, size of page table could be unacceptably high.

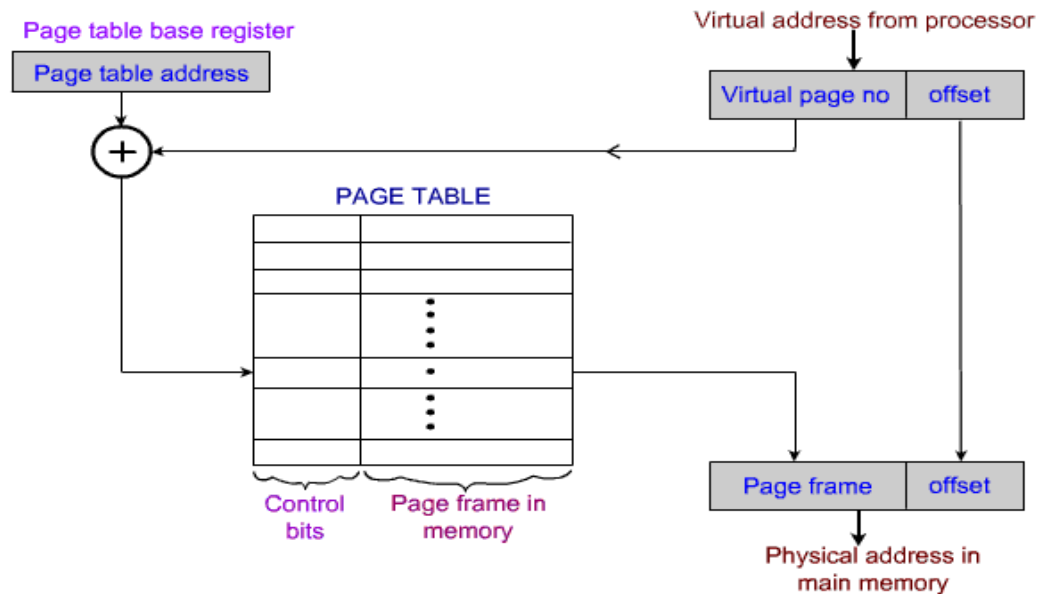
To overcome this problem, most virtual memory scheme store page table in virtual memory rather than in real memory.

This means that the page table is subject to paging just as other pages are.



When a process is running, at least a part of its page table must be in main memory, including the page table entry of the currently executing page.

A virtual address translation scheme by using page table is shown in the Figure 3.25.



**Figure 3.25:** Virtual Address Translation Method

Each virtual address generated by the processor is interpreted as virtual page number (high order list) followed by an offset (lower order bits) that specifies the location of a particular word within a page. Information about the main memory location of each page kept in a page table.

Some processors make use of a two level scheme to organize large page tables.

In this scheme, there is a page directory, in which each entry points to a page table.

Thus, if the length of the page directory is  $X$ , and if the maximum length of a page table is  $Y$ , then the process can consist of up to  $X * Y$  pages.

Typically, the maximum length of page table is restricted to the size of one page frame.

### Inverted page table structures

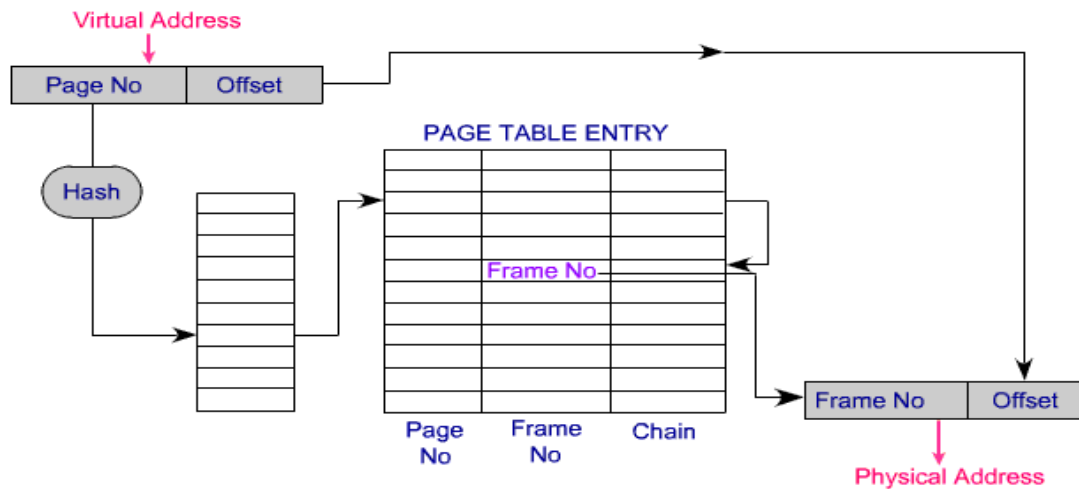
There is one entry in the hash table and the inverted page table for each real memory page rather than one per virtual page.

Thus a fixed portion of real memory is required for the page table, regardless of the number of processes or virtual page supported.

Because more than one virtual address may map into the hash table entry, a chaining technique is used for managing the overflow.

The hashing techniques results in chains that are typically short ? either one or two entries.

The inverted page table structure for address translation is shown in the Figure 3.26.



**Figure 3.26:** Inverted Page table structure

### Translation Lookaside Buffer (TLB)

Every virtual memory reference can cause two physical memory accesses.

One to fetch the appropriate page table entry

One to fetch the desired data.

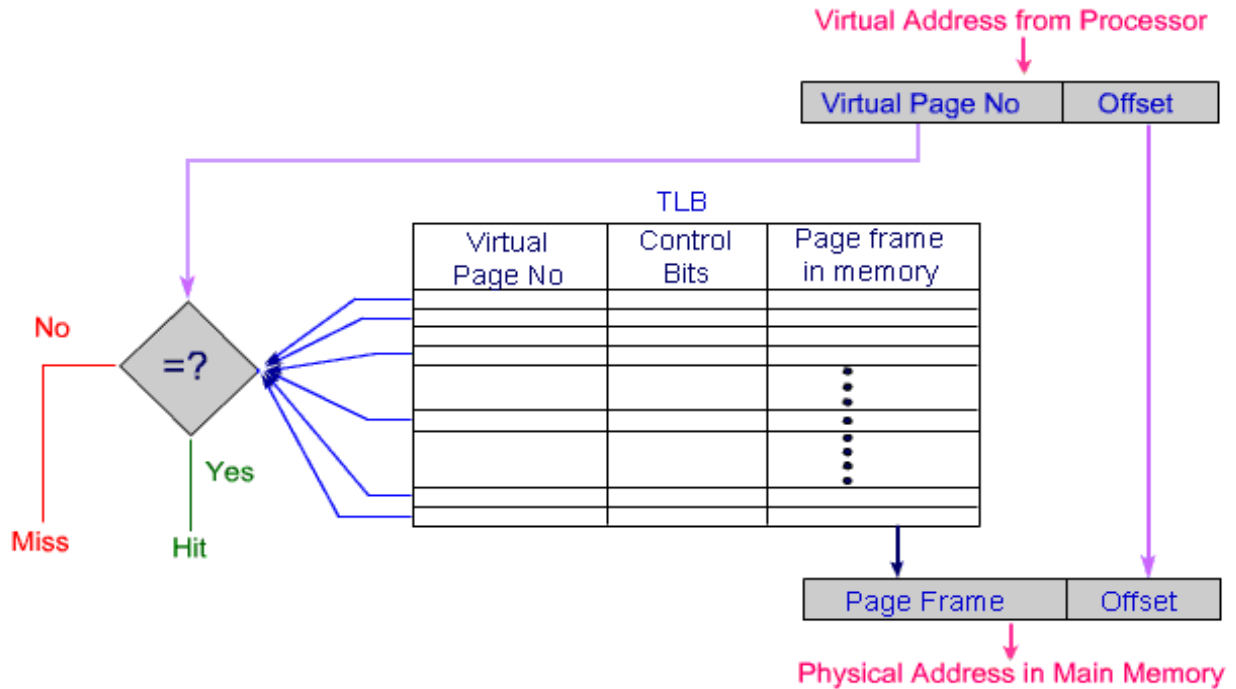
Thus a straight forward virtual memory scheme would have the effect of doubling the memory access time.

To overcome this problem, most virtual memory schemes make use of a special cache for page table entries, usually called *Translation Lookaside Buffer (TLB)*.

This cache functions in the same way as a memory cache and contains those page table entries that have been most recently used.

In addition to the information that constitutes a page table entry, the TLB must also include the virtual address of the entry.

The Figure 3.27 shows a possible organization of a TLB whwere the associative mapping technique is used.



**Figure 3.27:** Use of an associative mapped TLB

Set-associative mapped TLBs are also found in commercial products.

An essential requirement is that the contents of the TLB be coherent with the contents of the page table in the main memory.

When the operating system changes the contents of the page table it must simultaneously invalidate the corresponding entries in the TLB. One of the control bits in the TLB is provided for this purpose.

#### Address Translation proceeds as follows:

- Given a virtual address, the MMU looks in the TLB for the reference page.
- If the page table entry for this page is found in the TLB, the physical address is obtained immediately.
- If there is a miss in the TLB, then the required entry is obtained from the page table in the main memory and the TLB is updated.
- When a program generates an access request to a page that is not in the main memory, a page fault is said to have occurred.
- The whole page must be brought from the disk into the memory before access can proceed.
- When it detects a page fault, the MMU asks the operating system to intervene by raising an exception.(interrupt).
- Processing of active task is interrupted, and control is transferred to the operating system.
- The operating system then copies the requested page from the disk into the main memory and returns control to the interrupted task. Because a long delay occurs due to a page transfer takes place, the operating system may suspend execution of the task that caused the page fault and begin execution of another task whose page are in the main memory.

**Refer COMPUTER ORGANIZATION AND DESIGN By P. PAL CHAUDHURI**  
**for write back and write through policy of cache memory**

## **Input / Output**

1. **Introduction to I/O**
2. **Program Controlled I/O**
3. **Interrupt Controlled I/O**
4. **Direct Memory Access**

## **Input/Output Organization**

- The computer system's **input/output** (I/O) architecture is its interface to the outside world.
- Till now we have discussed the two important modules of the computer system -
  - o **The processor** and
  - o **The memory** module.
- The third key component of a computer system is a set of **I/O modules**
- Each I/O module interfaces to the system bus and controls one or more peripheral devices.

There are several reasons why an I/O device or peripheral device is not directly connected to the system bus. Some of them are as follows -

- There are a wide variety of peripherals with various methods of operation. It would be impractical to include the necessary logic within the processor to control several devices.
- The data transfer rate of peripherals is often much slower than that of the memory or processor. Thus, it is impractical to use the high-speed system bus to communicate directly with a peripheral.
- Peripherals often use different data formats and word lengths than the computer to which they are attached.

Thus, an I/O module is required.

## **Input/Output Modules**

The major functions of an I/O module are categorized as follows ?

- **Control and timing**

- Processor Communication
- Device Communication
- Data Buffering
- Error Detection

During any period of time, the processor may communicate with one or more external devices in unpredictable manner, depending on the program's need for I/O.

The internal resources, such as main memory and the system bus, must be shared among a number of activities, including data I/O.

### **Control & timings:**

The I/O function includes a control and timing requirement to co-ordinate the flow of traffic between internal resources and external devices.

For example, the control of the transfer of data from an external device to the processor might involve the following sequence of steps ?

1. The processor interacts with the I/O module to check the status of the attached device.
2. The I/O module returns the device status.
3. If the device is operational and ready to transmit, the processor requests the transfer of data, by means of a command to the I/O module.
4. The I/O module obtains a unit of data from external device.
5. The data are transferred from the I/O module to the processor.

If the system employs a bus, then each of the interactions between the processor and the I/O module involves one or more bus arbitrations.

### **Processor & Device Communication**

During the I/O operation, the I/O module must communicate with the processor and with the external device.

Processor communication involves the following -

#### **Command decoding :**

The I/O module accepts command from the processor, typically sent as signals on control bus.

#### **Data :**

Data are exchanged between the processor and the I/O module over the data bus.

#### **Status Reporting :**

Because peripherals are so slow, it is important to know the status of the I/O module. For example, if an I/O module is asked to send data to the processor(read), it may not be ready to do so because it is still working on the previous I/O command. This fact can be reported with a status signal. Common status signals are **BUSY** and **READY**.

#### Address Recognition :

Just as each word of memory has an address, so thus each of the I/O devices. Thus an I/O module must recognize one unique address for each peripheral it controls.

On the other hand, the I/O must be able to perform device communication. This communication involves command, status information and data.

#### Data Buffering:

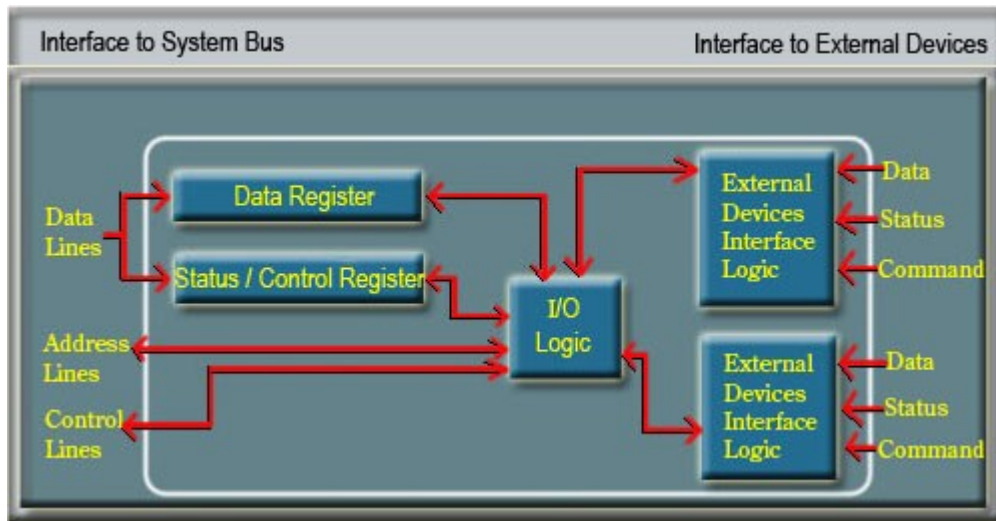
An essential task of an I/O module is data buffering. The data buffering is required due to the mismatch of the speed of CPU, memory and other peripheral devices. In general, the speed of CPU is higher than the speed of the other peripheral devices. So, the I/O modules store the data in a data buffer and regulate the transfer of data as per the speed of the devices.

In the opposite direction, data are buffered so as not to tie up the memory in a slow transfer operation. Thus the I/O module must be able to operate at both device and memory speed.

#### Error Detection:

Another task of I/O module is error detection and for subsequently reporting error to the processor. One class of error includes mechanical and electrical malfunctions reported by the device (e.g. paper jam). Another class consists of unintentional changes to the bit pattern as it is transmitted from devices to the I/O module.

Block diagram of I/O Module is shown in the Figure 6.1.



**Figure 6.1:** Block diagram of I/O Module.

There will be many I/O devices connected through I/O modules to the system. Each device will be identified by a unique address.

When the processor issues an I/O command, the command contains the address of the device that is used by the command. The I/O module must interpret the address lines to check if the command is for itself.

Generally in most of the processors, the processor, main memory and I/O share a common bus(data address and control bus).

Two types of addressing are possible -

- Memory-mapped I/O
- Isolated or I/O mapped I/O

### Memory-mapped I/O:

There is a single address space for memory locations and I/O devices.

The processor treats the status and address register of the I/O modules as memory location.

For example, if the size of address bus of a processor is 16, then there are  $2^{16}$  combinations and all together  $2^{16}$  address locations can be addressed with these 16 address lines.

Out of these  $2^{16}$  address locations, some address locations can be used to address I/O devices and other locations are used to address memory locations.

Since I/O devices are included in the same memory address space, so the status and address registers of I/O modules are treated as memory location by the processor. Therefore, the same machine instructions are used to access both memory and I/O devices.

### **Isolated or I/O -mapped I/O:**

In this scheme, the full range of addresses may be available for both.

The address refers to a memory location or an I/O device is specified with the help of a command line.

In general  $IO/\overline{M}$  command line is used to identify a memory location or an I/O device.

if  $IO/\overline{M}=1$ , it indicates that the address present in address bus is the address of an I/O device.

if  $IO/\overline{M}=0$ , it indicates that the address present in address bus is the address of a memory location.

Since full range of address is available for both memory and I/O devices, so, with 16 address lines, the system may now support both  $2^{16}$  memory locations and  $2^{16}$  I/O addresses.

### **Input / Output Subsystem**

There are three basic forms of input and output systems ?

- **Programmed I/O**
- **Interrupt driven I/O**
- **Direct Memory Access(DMA)**

With programmed I/O, the processor executes a program that gives its direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data.

With interrupt driven I/O, the processor issues an I/O command, continues to execute other instructions, and is interrupted by the I/O module when the I/O module completes its work.

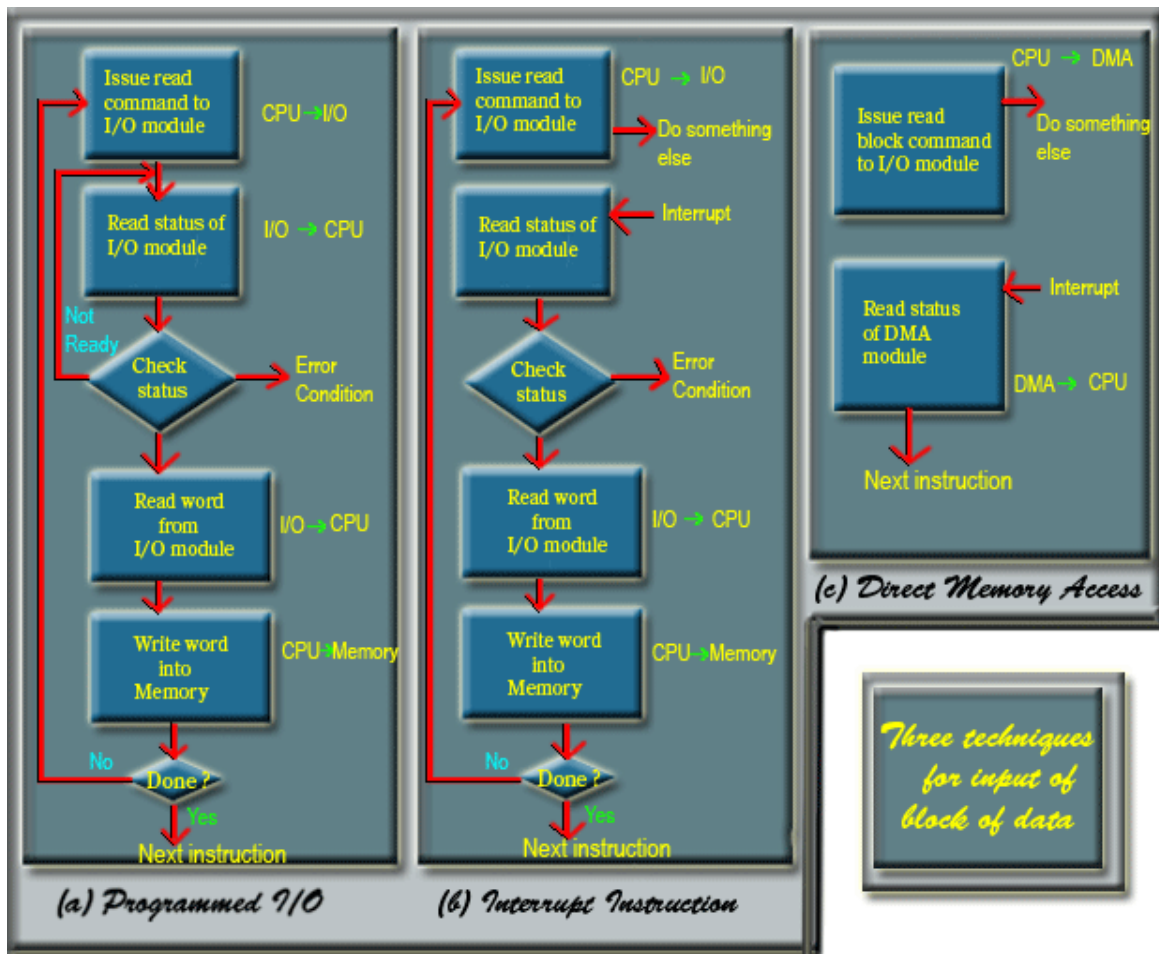


In Direct Memory Access (DMA), the I/O module and main memory exchange data directly without processor involvement.

With both programmed I/O and Interrupt driven I/O, the processor is responsible for extracting data from main memory for output operation and storing data in main memory for input operation. To send data to an output device, the CPU simply moves that data to a *special memory location* in the I/O address space if I/O mapped input/output is used or to an address in the memory address space if memory mapped I/O is used.

To read data from an input device, the CPU simply moves data from the address (I/O or memory) of that device into the CPU.

**Input/Output Operation:** The input and output operation looks very similar to a memory read or write operation except it usually takes *more time* since peripheral devices are slow in speed than main memory modules. The working principle of the three methods for input of a Block of Data is shown in the Figure 6.2.



[Click on Image To View Large Image](#)

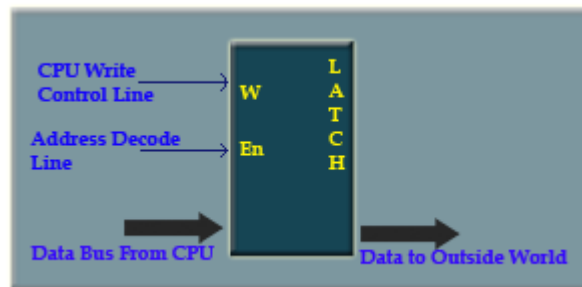
**Figure 6.2:** Working of three techniques for input of block of data.

## Input/Output Port

An *I/O port* is a device that looks like a memory cell to the computer but contains connection to the outside world.

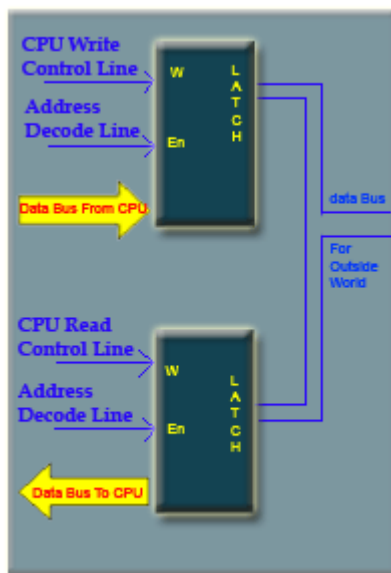
An *I/O port* typically uses a *latch*. When the CPU writes to the address associated with the latch, the latch device captures the data and makes it available on a set of wires external to the CPU and memory system.

The *I/O ports* can be *read-only*, *write-only*, or *read/write*. The *write-only* port is shown in the Figure 6.3.



**Figure 6.3:** The write only port

First, the CPU will place the address of the device on the *I/O address bus* and with the help of *address decoder* a signal is generated which will enable the latch.



**Figure 6.4:** Read / Write port.

Next, the CPU will indicate the operation is a write operation by putting the appropriate signal in CPU write control line.

Then the data to be transferred will be placed in the CPU bus, which will be stored in the latch for the onward transmission to the device.

Both the address decode and write control lines must be active for the latch to operate.

The *read/write* or *input/output* port is shown in the Figure 6.4.

The device is identified by putting the appropriate address in the I/O address lines. The address decoder will generate the signal for the address decode lines. According to the operation, *read* or *write*, it will select either of the latch.

If it is a write operation, then data will be placed in the latch from CPU for onward transmission to the output device.

If it is in a read operation, the data that are already stored in the latch will be transferred to the CPU.

A *read only* (input) port is simply the lower half of the Figure 6.4.

In case of I/O mapped I/O, a different address space is used for I/O devices. The address space for memory is different. In case of memory mapped I/O, same address space is used for both memory and I/O devices. Some of the memory address space are kept reserved for I/O devices.

To the programmer, the difference between I/O-mapped and memory-mapped input/output operation is the instruction to be used.

For memory-mapped I/O, any instruction that accessed memory can access a memory-mapped I/O port.

I/O-mapped input/output uses special instruction to access I/O port.

Generally, a given peripheral device will use more than a single I/O port. A typical PC parallel printer interface, for example, uses three ports, a *read/write port*, and *input port* and an *output port*.

The read/write port is the data port ( it is read/write to allow the CPU to read the last ASCII character it wrote to the printer port ).

The input port returns control signals from the printer.

- These signals indicate whether the printer is ready to accept another character, is off-line, is out of paper, etc.

The output port transmits control information to the printer such as

- whether data is available to print.

Memory-mapped I/O subsystems and I/O-mapped subsystems both require the CPU to move data between the peripheral device and main memory.

For example, to input a sequence of 20 bytes from an input port and store these bytes into memory, the CPU must send each value and store it into memory.

### **Programmed I/O:**

In programmed I/O, the data transfer between CPU and I/O device is carried out with the help of a software routine.

When a processor is executing a program and encounters an instruction relating to I/O, it executes that I/O instruction by issuing a command to the appropriate I/O module.

The I/O module will perform the requested action and then set the appropriate bits in the I/O status register.

The I/O module takes no further action to alert the processor.

It is the responsibility of the processor to check periodically the status of the I/O module until it finds that the operation is complete.

In programmed I/O, when the processor issues a command to a I/O module, it must wait until the I/O operation is complete.

Generally, the I/O devices are slower than the processor, so in this scheme CPU time is wasted. CPU is checking the status of the I/O module periodically without doing any other work.

## **I/O Commands**

To execute an I/O-related instruction, the processor issues an address, specifying the particular I/O module and external device, and an I/O command. There are four types of I/O commands that an I/O module will receive when it is addressed by a processor ?

- **Control** : Used to activate a peripheral device and instruct it what to do. For example, a magnetic tape unit may be instructed to rewind or to move forward one record. These commands are specific to a particular type of peripheral device.
- **Test** : Used to test various status conditions associated with an I/O module and its peripherals. The processor will want to know if the most recent I/O operation is completed or any error has occurred.
- **Read** : Causes the I/O module to obtain an item of data from the peripheral and place it in the internal buffer.
- **Write** : Causes the I/O module to take an item of data ( byte or word ) from the data bus and subsequently transmit the data item to the peripheral.

## **Interrupt driven I/O**

The problem with programmed I/O is that the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of data. The processor, while waiting, must repeatedly interrogate the status of the I/O module.

This type of I/O operation, where the CPU constantly tests a part to see if data is available, is polling, that is, the CPU Polls (asks) the port if it has data available or if it is capable of accepting data. Polled I/O is inherently inefficient.

The solution to this problem is to provide an interrupt mechanism. In this approach the processor issues an I/O command to a module and then go on to do some other useful work. The I/O module then interrupt the processor to request service when it is ready to exchange data with the processor. The processor then executes the data transfer. Once the data transfer is over, the processor then resumes its former processing.

## **Let us consider how it works**

### **A. From the point of view of the I/O module:**

- o For input, the I/O module services a READ command from the processor.
- o The I/O module then proceeds to read data from an associated peripheral device.
- o Once the data are in the modules data register, the module issues an interrupt to the processor over a control line.
- o The module then waits until its data are requested by the processor.
- o When the request is made, the module places its data on the data bus and is then ready for another I/O operation.

#### **B. From the processor point of view; the action for an input is as follows :**

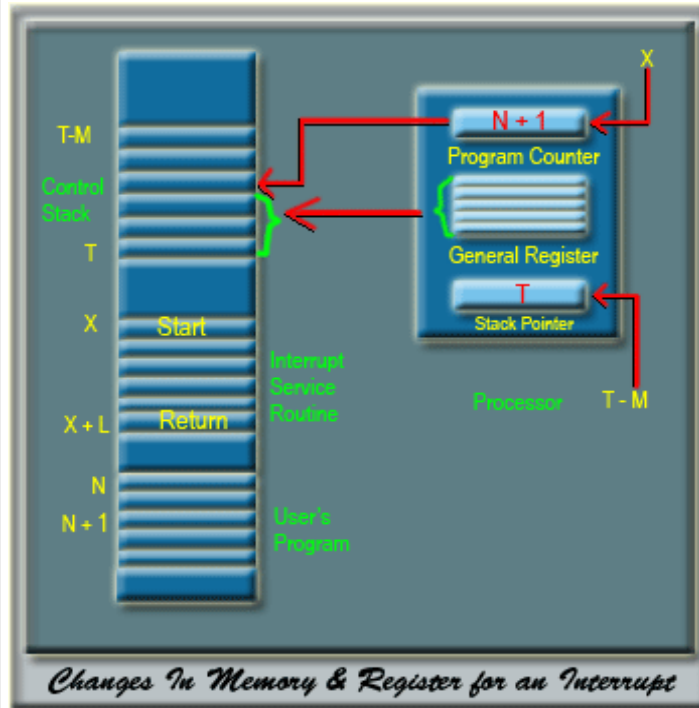
- The processor issues a READ command.
- It then does something else  
(e.g. the processor may be working on several different programs at the same time)
- At the end of each instruction cycle, the processor checks for interrupts
- When the interrupt from an I/O module occurs, the processor saves the context  
(e.g. program counter & processor registers) 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 Processing**

The occurrence of an interrupt triggers a number of events, both in the processor hardware and in software.

When an I/O device completes an I/O operation, the following sequences of hardware events occurs:

1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt.
3. The processor tests for the interrupt; if there is one interrupt pending, then the processor sends an acknowledgement signal to the device which issued the interrupt. After getting acknowledgement, the device removes its interrupt signals.
4. The processor now needs to prepare to transfer control to the interrupt routine. It needs to save the information needed to resume the current program at the point of interrupt. The minimum information required to save is the processor status word (PSW) and the location of the next instruction to be executed which is nothing but the contents of program counter. These can be pushed into the system control stack.
5. The processor now loads the program counter with the entry location of the interrupt handling program that will respond to the interrupt.

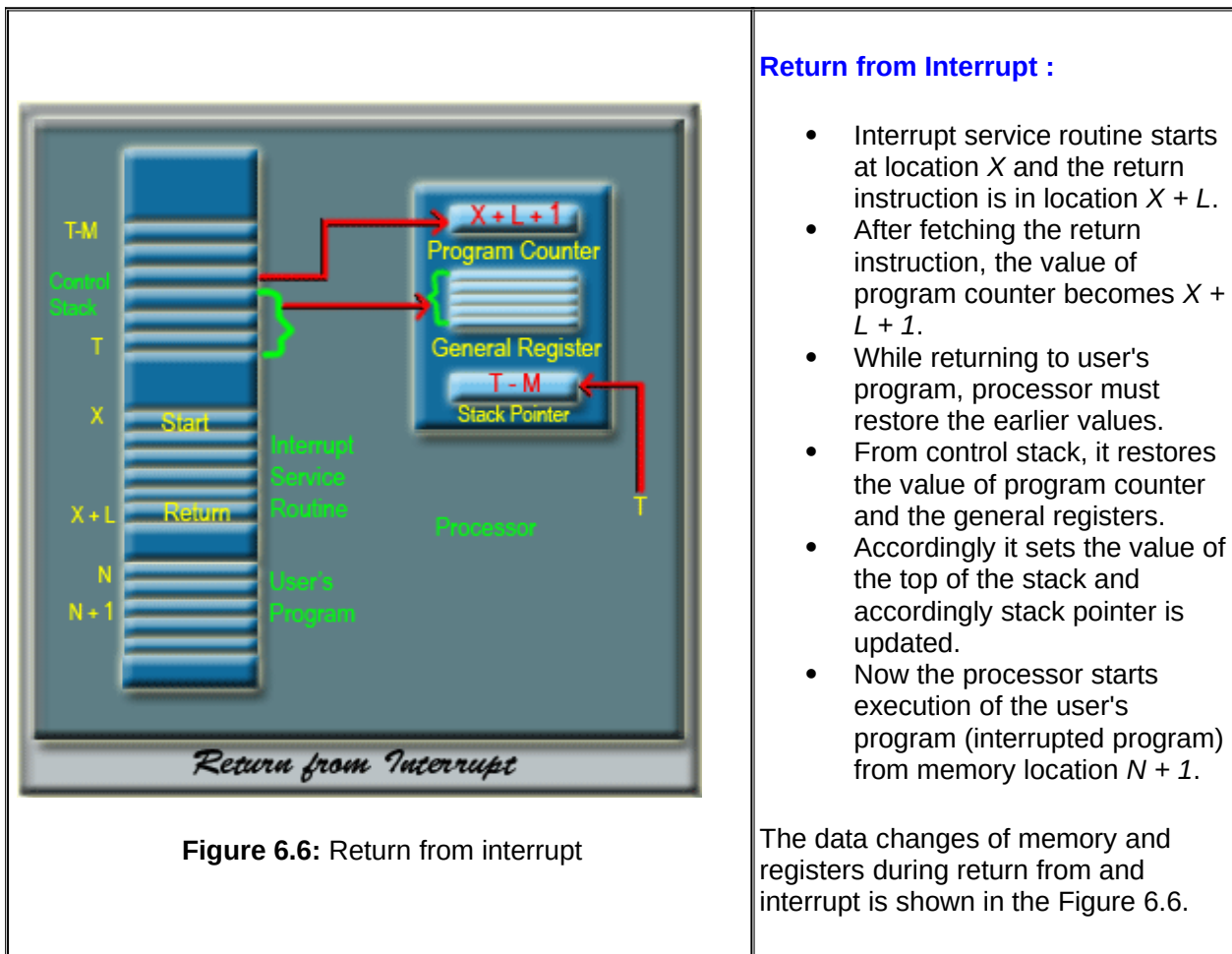


**Figure 6.5:** Changes of memory and register for an interrupt

### Interrupt Processing:

- An interrupt occurs when the processor is executing the instruction of location  $N$ .
- At that point, the value of program counter is  $N+1$ .
- Processor services the interrupt after completion of current instruction execution.
- First, it moves the content of general registers to system stack.
- Then it moves the program counter value to the system stack.
- Top of the system stack is maintained by stack pointer.
- The value of stack pointer is modified to point to the top of the stack.
- If  $M$  elements are moved to the system stack, the value of stack pointer is changed from  $T$  to  $T-M$ .
- Next, the program counter is loaded with the starting address of the interrupt service routine.
- Processor starts executing the interrupt service routine.

The data changes of memory and registers during interrupt service is shown in the Figure 6.5.



Once the program counter has been loaded, the processor proceeds to the next instruction cycle, which begins with an interrupt fetch. The control will transfer to interrupt handler routine for the current interrupt.

The following operations are performed at this point.

1. At the point, the program counter and PSW relating to the interrupted program have been saved on the system stack. In addition to that some more information must be saved related to the current processor state which includes the control of the processor registers, because these registers may be used by the interrupt handler. Typically, the interrupt handler will begin by saving the contents of all registers on stack.
2. The interrupt handles next processes the interrupt. This includes an examination of status information relating to the I/O operation or, other event that caused an interrupt.
3. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers.



4. The final act is to restore the PSW and program counter values from the stack. As a result, the next instruction to be executed will be from the previously interrupted program.

## **Design Issues for Interrupt**

Two design issues arise in implementing interrupt I/O.

- o There will almost invariably be multiple I/O modules, how does the processor determine which device issued the interrupt?
- o If multiple interrupts have occurred how the processor does decide which one to process?

## **Device Identification**

Four general categories of techniques are in common use:

- **Multiple interrupt lines**
- **Software poll**
- **Daisy chain (hardware poll, vectored)**
- **Bus arbitration ( vectored)**

## **Multiple Interrupts Lines:**

The most straight forward approach is to provide multiple interrupt lines between the processor and the I/O modules.

It is impractical to dedicate more than a few bus lines or processor pins to interrupt lines.

Thus, though multiple interrupt lines are used, it is most likely that each line will have multiple I/O modules attached to it. Thus one of the other three techniques must be used on each line.

## **Software Poll :**

When the processor detects an interrupt, it branches to an interrupt service routine whose job is to poll each I/O module to determine which module caused the interrupt.

The poll could be implemented with the help of a separate command line (e.g. TEST I/O). In this case, the processor raises TEST I/O and place the address of a particular I/O module on the address lines. The I/O module responds positively if it set the interrupt.

Alternatively, each I/O module could contain an addressable status register. The processor then reads the status register of each I/O module to identify the interrupting module.

Once the correct module is identified, the processor branches to a device service routine specific to that device.

The main disadvantage of software poll is that it is time consuming. Processor has to check the status of each I/O module and in the worst case it is equal to the number of I/O modules.

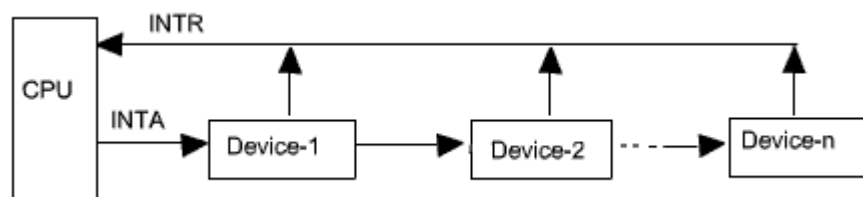
### Daisy Chain :

In this method for interrupts all I/O modules share a common interrupt request lines. However the interrupt acknowledge line is connected in a daisy chain fashion. When the processor senses an interrupt, it sends out an interrupt acknowledgement.

The interrupt acknowledge signal propagates through a series of I/O module until it gets to a requesting module.

The requesting module typically responds by placing a word on the data lines. This word is referred to as a vector and is either the address of the I/O module or some other unique identification.

In either case, the processor uses the vector as a pointer to the appropriate device service routine. This avoids the need to execute a general interrupt service routine first. This technique is referred to as a *vectored interrupt*. The daisy chain arrangement is shown in the Figure 6.7.



### Bus Arbitration :

In bus arbitration method, an I/O module must first gain control of the bus before it can raise the interrupt request line. Thus, only one module can raise the interrupt line at a time. When the processor detects the interrupt, it responds on the interrupt acknowledge line. The requesting module then places its vector on the data line.

### Handling multiple interrupts

There are several techniques to identify the requesting I/O module. These techniques also provide a way of assigning priorities when more than one device is requesting interrupt service.

*With multiple lines*, the processor just picks the interrupt line with highest priority. During the processor design phase itself priorities may be assigned to each interrupt lines.

*With software polling*, the order in which modules are polled determines their priority.

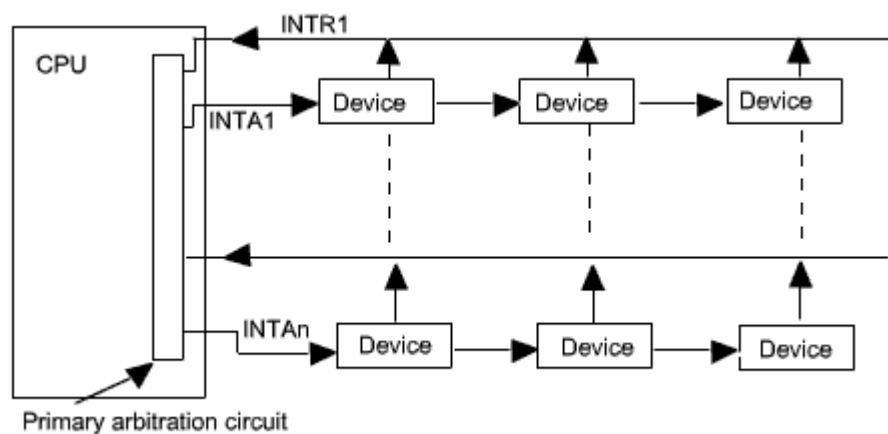
*In case of daisy chain configuration*, the priority of a module is determined by the position of the module in the daisy chain. The module nearer to the processor in the chain has got higher priority, because this is the first module to receive the acknowledge signal that is generated by the processor.

*In case of bus arbitration method*, more than one module may need control of the bus. Since only one module at a time can successfully transmit over the bus, some method of arbitration is needed. The various methods can be classified into two group ? centralized and distributed.

**In a centralized scheme**, a single hardware device, referred to as a bus controller or arbiter is responsible for allocating time on the bus. The device may be a separate module or part of the processor.

**In distributed scheme**, there is no central controller. Rather, each module contains access control logic and the modules act together to share the bus.

It is also possible to combine different device identification techniques to identify the devices and to set the priorities of the devices. As for example multiple interrupt lines and daisy chain technologies can be combined together to give access for more devices.



**Figure 6.8:** Possible arrangement to handle multiple interrupt

In one interrupt line, more than one device can be connected in daisy chain fashion. The High priorities devices should be

connected to the interrupt lines that has got higher priority.

A possible arrangement is shown in the Figure 6.8.

## Interrupt Nesting

The arrival of an interrupt request from an external device causes the processor to suspend the execution of one program and starts the execution of another. The execution of this another program is nothing but the interrupt service routine for that specified device.

Interrupt may arrive at any time. So during the execution of an interrupt service routine, another interrupt may arrive. This kind of interrupts are known as *nesting of interrupt*.

Whether interrupt nesting is allowed or not? This is a design issue. Generally nesting of interrupt is allowed, but with some restrictions. The common notion is that a high priority device may interrupt a low priority device, but not the vice-versa.

To accomodate such type of restrictions, all computer provide the programmer with the ability to enable and disable such interruptions at various time during program execution. The processor provides some instructions to enable the interrupt and disable the interrupt. If interrupt is disabled, the CPU will not respond to any interrupt signal.

On the other hand, when multiple lines are used for interrupt and priorities are assigned to these lines, then the interrupt received in a low priority line will not be served if an interrupt routine is in execution for a high priority device. After completion of the interrupt service routine of high priority devices, processor will respond to the interrupt request of low priority devices

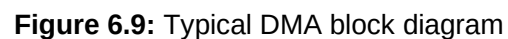
## Direct Memory Access

We have discussed the data transfer between the processor and I/O devices. We have discussed two different approaches namely programmed I/O and Interrupt-driven I/O. Both the methods require the active intervention of the processor to transfer data between memory and the I/O module, and any data transfer must transverse a path through the processor. Thus both these forms of I/O suffer from two inherent drawbacks.

- o The I/O transfer rate is limited by the speed with which the processor can test and service a device.
- o The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

DMA transfers are performed by a control circuit associated with the I/O device and this circuit is referred as DMA controller. The DMA controller allows direct data transfer between the device and the main memory without involving the processor.

The typical block diagram of a DMA controller is shown in the Figure 6.9.



sending to the DMA module the following information.

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

- o 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 works. It has delegated this I/O operation to the DMA module.

The DMA module checks the status of the I/O device whose address is communicated to DMA controller by the processor. If the specified I/O device is ready for data transfer, then DMA module generates the DMA request to the processor. Then the processor indicates the release of the system bus through DMA acknowledge.

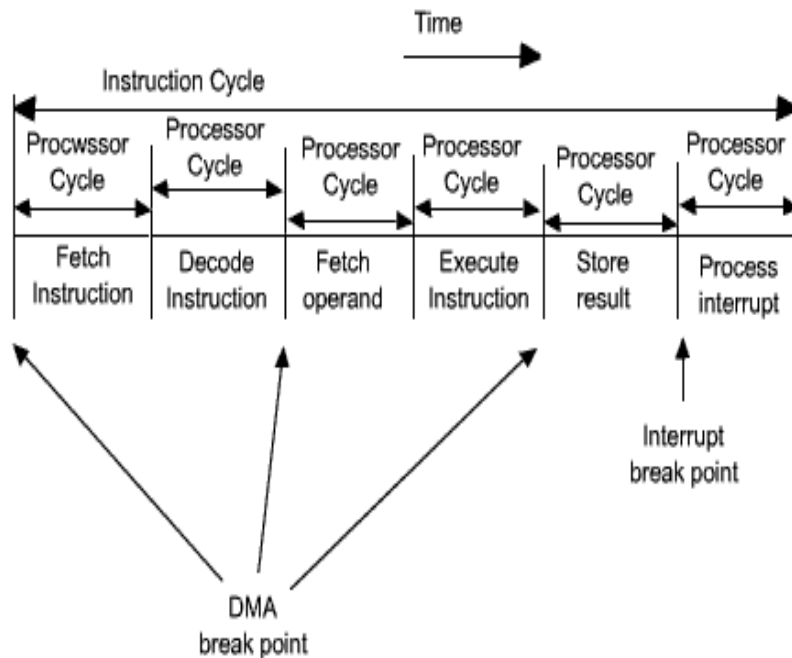
The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor.

When the transfer is completed, the DMA module sends an interrupt signal to the processor. After receiving the interrupt signal, processor takes over the system bus.

Thus the processor is involved only at the beginning and end of the transfer. During that time the processor is suspended.

It is not required to complete the current instruction to suspend the processor. The processor may be suspended just after the completion of the current bus cycle. On the other hand, the processor can be suspended just before the need of the system bus by the processor, because DMA controller is going to use the system bus, it will not use the processor.

The point where in the instruction cycle the processor may



**Figure 6.10 : DMA break point**

be suspended  
shown in the Figure  
6.10.

When the processor is suspended, then the DMA module transfer one word and return control to the processor.

Note that, this is not an interrupt, the processor does not save a context and do something else. Rather, the processor pauses for one bus cycle.

During that time processor may perform some other task which does not involve the system bus. In the worst situation processor will wait for some time, till the DMA releases the bus.

The net effect is that the processor will go slow. But the net effect is the enhancement of performance, because for a multiple word I/O transfer, DMA is far more efficient than interrupt driven or programmed I/O.

The DMA mechanism can be configured in different ways. The most common amongst them are:

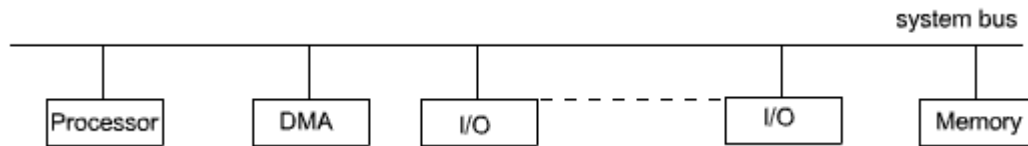
- **Single bus, detached DMA - I/O configuration.**
- **Single bus, Integrated DMA - I/O configuration.**
- **Using separate I/O bus.**

### **Single bus, detached DMA - I/O configuration**

In this organization all modules share the same system bus. The DMA module here acts as a surrogate processor. This method uses programmed I/O to exchange data between memory and an I/O module through the DMA module.

For each transfer it uses the bus twice. The first one is when transferring the data between I/O and DMA and the second one is when transferring the data between DMA and memory. Since the bus is used twice while transferring data, so the bus will be suspended twice. The transfer consumes two bus cycle.

The interconnection organization is shown in the Figure 6.11.



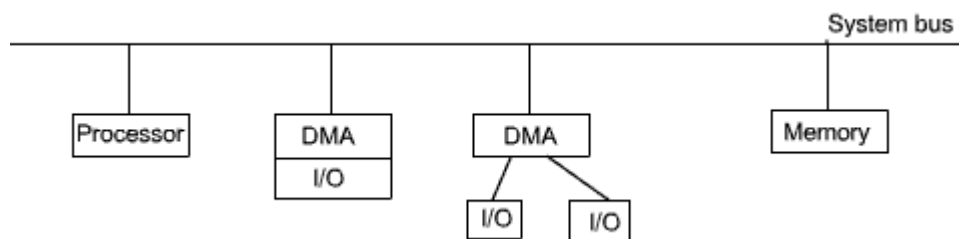
**Figure 6.11:** Single bus arrangement for DMA transfer

### Single bus, Integrated DMA - I/O configuration

By integrating the DMA and I/O function the number of required bus cycle can be reduced. In this configuration, the DMA module and one or more I/O modules are integrated together in such a way that the system bus is not involved. In this case DMA logic may actually be a part of an I/O module, or it may be a separate module that controls one or more I/O modules.

The DMA module, processor and the memory module are connected through the system bus. In this configuration each transfer will use the system bus only once and so the processor is suspended only once.

The system bus is not involved when transferring data between DMA and I/O device, so processor is not suspended. Processor is suspended when data is transferred between DMA and memory. The configuration is shown in the Figure 6.12.



**Figure 6.12:** Single bus integrated DMA transfer

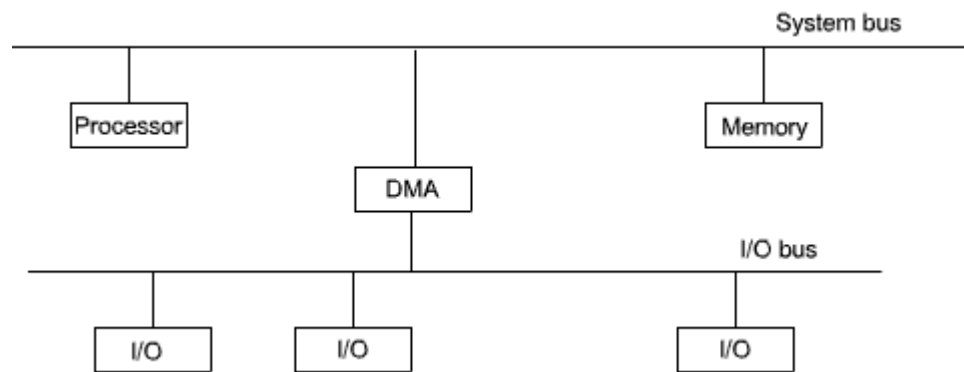
### Using separate I/O bus

In this configuration the I/O modules are connected to the DMA through another I/O bus. In this case the DMA module is reduced to one.

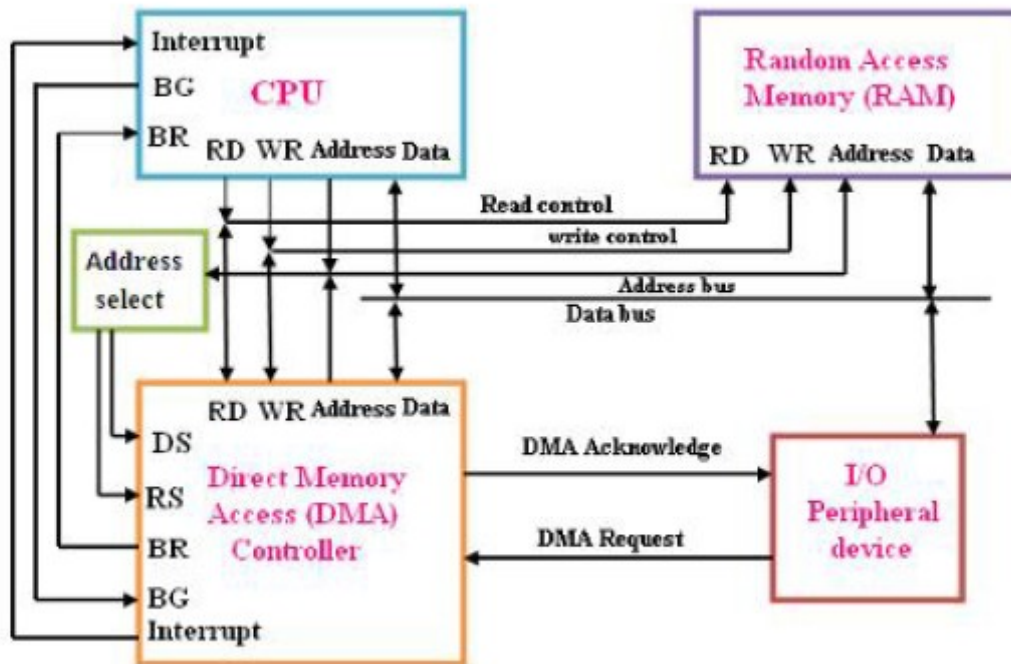
Transfer of data between I/O module and DMA module is carried out through this I/O bus. In this transfer, system bus is not in use and so it is not needed to suspend the processor.

There is another transfer phase between DMA module and memory. In this time system bus is needed for transfer and processor will be suspended for one bus cycle. The configuration is shown in the Figure 6.13.





**Figure 6.13:** Seperate I/O bus for DMA transfer



### DMA Transfer In A Computer System

- The DMA request line is used to request a DMA transfer.
- The bus request (BR) signal is used by the DMA controller to request the CPU to relinquish control of the buses.
- The CPU activates the bus grant (BG) output to inform the external DMA that its buses are in a high-impedance state (so that they can be used in the DMA transfer.)
- The address bus is used to address the DMA controller and memory at given location.
- The Device select (DS) and register select (RS) lines are activated by addressing the DMA controller.
- The RD and WR lines are used to specify either a read (RD) or write (WR) operation on the given memory location.
- The DMA acknowledge line is set when the system is ready to initiate data transfer.
- The data bus is used to transfer data between the I/O device and memory.
- When the last word of data in the DMA transfer is transferred, the DMA controller informs the termination of the transfer to the CPU by means of the interrupt line.