UNIT V

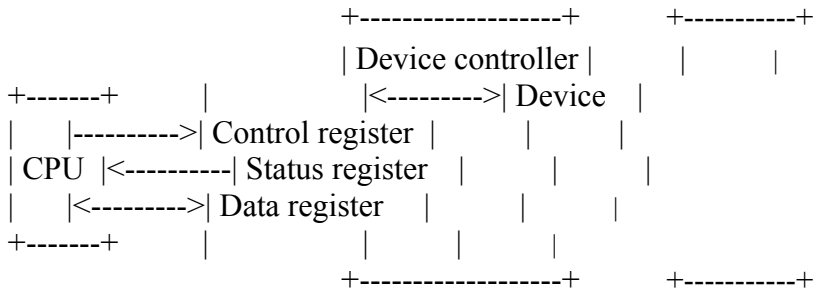
MEMORY AND I/O SYSTEMS

From the CPU's perspective, an I/O device appears as a set of special-purpose registers, of three general types:

- Status registers provide status information to the CPU about the I/O device. These registers are often read-only, i.e. the CPU can only read their bits, and cannot change them.
- Configuration/control registers are used by the CPU to configure and control the device. Bits in these configuration registers may be write-only, so the CPU can alter them, but not read them back. Most bits in control registers can be both read and written.
- Data registers are used to read data from or send data to the I/O device.

In some instances, a given register may fit more than one of the above categories, e.g. some bits are used for configuration while other bits in the same register provide status information.

The logic circuit that contains these registers is called the *device controller*, and the software that communicates with the controller is called a *device driver*.

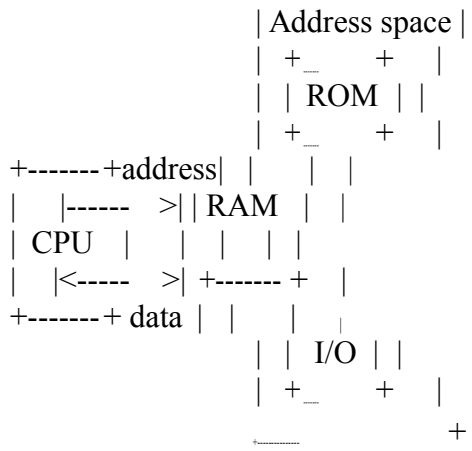


Simple devices such as keyboards and mice may be represented by only a few registers, while more complex ones such as disk drives and graphics adapters may have dozens.

Each of the I/O registers, like memory, must have an address so that the CPU can read or write specific registers.

Some CPUs have a separate address space for I/O devices. This requires separate instructions to perform I/O operations.

Other architectures, like the MIPS, use *memory- mapped I/O*. When using memory-mapped I/O, the same address space is shared by memory and I/O devices. Some addresses represent memory cells, while others represent registers in I/O devices. No separate I/O instructions are needed in a CPU that uses memory-mapped I/O. Instead, we can perform I/O operations using any instruction that can reference memory.



On the MIPS, we would access ROM, RAM, and I/O devices using load and store instructions. Which type of device we access depends only on the address used!

```
lw      $t0, 0x00000004     # Read ROM
sw      $t0, 0x00000004   # Write ROM (bus error!)

lbu     $t0, 0x0000ffc1      # Read RAM
sb      $t0, 0x0000ffc1      # Write RAM

lbu     $t0, 0xffff0000     # Read an I/O device
sb      $t0, 0xffff0004     # Write to an I/O device
```

The 32-bit MIPS architecture has a 32-bit address, and hence an address space of 4 gigabytes. Addresses 0x00000000 through 0xfffeffff are used for memory, and addresses 0xffff0000 - 0xffffffff (the last 64 kilobytes) are reserved for I/O device registers. This is a very small fraction of the total address space, and yet far more space than is needed for I/O devices on any one computer.

Each register within an I/O controller must be assigned a unique address within the address space. This address may be fixed for certain devices, and auto-assigned for others. (PC plug-and-play devices have auto-assigned I/O addresses, which are determined during boot-up.)

MEMORY HIERARCHY

Primary memory – costly & have limited size. mainly used for storing the currently processing data

Rom

RAM.

Secondary Memory

used to store data & instructions (programs when they are not being processed

Eg: hard disks, floppies, CD-Roms, magnetic tapes etc.

Memory Hierarchy :-

→ Ideally, computer memory should be fast, large and inexpensive. Unfortunately it is impossible to meet all the three of these requirements using one type of memory.

→ Increased speed and size are achieved at increased cost.

→ Very fast memory system can be achieved if SRAM chips are used. These chips are expensive and for the cost reason it is impracticable to build a large main memory using SRAM chips. The only alternative is to use DRAM chips for large main memories.

~~Based on the~~ d

→ DRAM should insert wait states in memory read/write cycles. This reduces the speed of execution.

→ In the memory system, small section of SRAM is added along with main memory, referred to as cache memory.

→ The program which is to be executed is loaded in the main memory, but the part of program (code) and data that work at a particular time is usually accessed from the cache memory.

→ The size of memory is still small compared to the demands of large programs with voluminous data. Very large disks are available at a resonable price, sacrificing the speed.

→ Thus it is realized that to make efficient computer system it is not possible to rely on single memory component, but to employ a memory hierarchy.

→ Using Memory hierarchy, all of different types of memory units are employed to give efficient computer system.

The typical Memory Hierachy is given in Fig below.

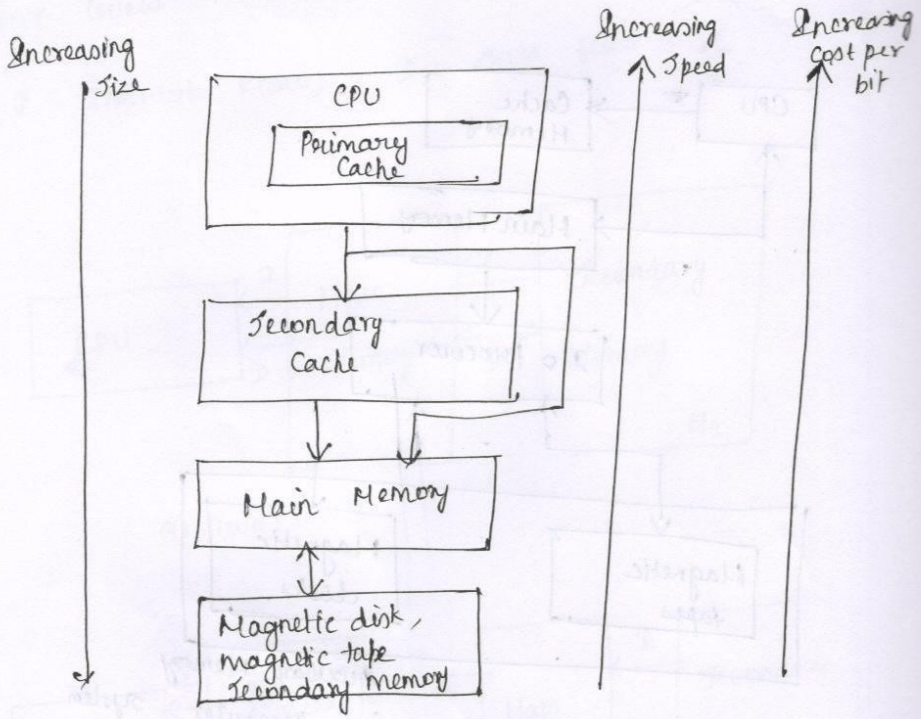


Fig : Typical memory hierarchy.

∴ Huge amount of cost effective storage can be provided by magnetic disks. Memory with DRAM technology can be built alongwith cache memory to achieve better performance.

Memory Hierarchy can be employed in a computer system is shown below

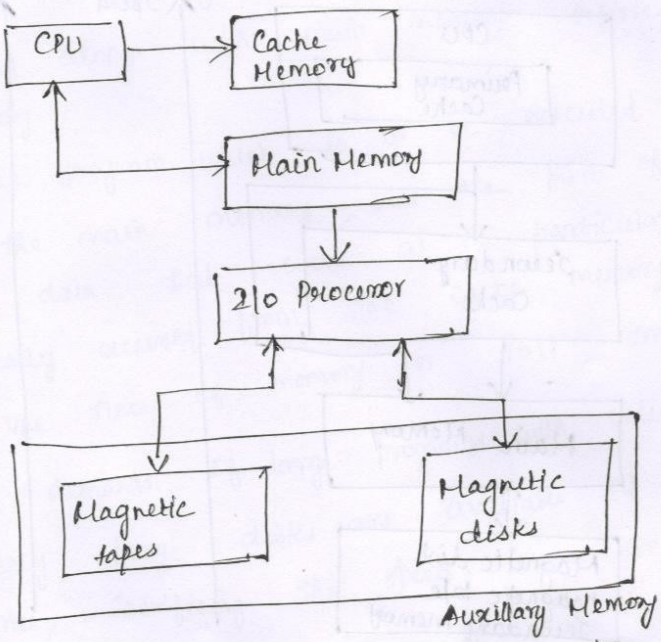


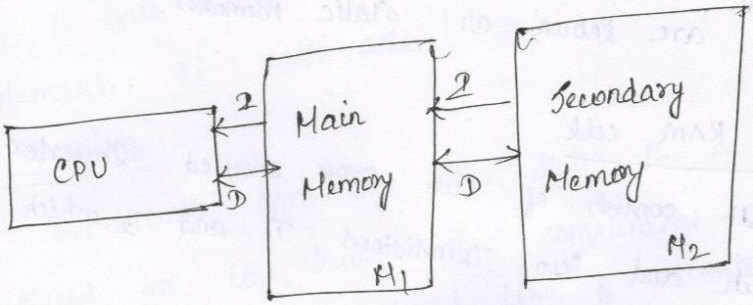
Fig: Memory Hierarchy in computer system

Magnetic tapes and Magnetic disks are used as secondary memory. This memory is also known as auxiliary memory.

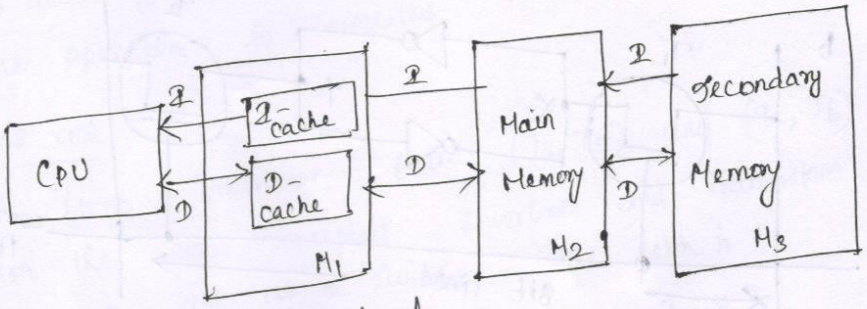
(4)

Common Memory hierarchies with two, three and four levels

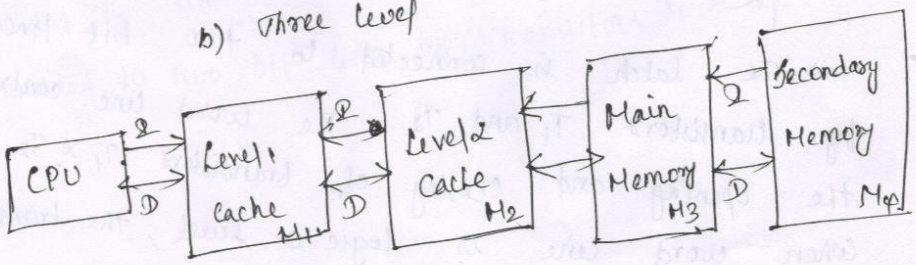
I - Instruction flow, D : Data flow



a) Two level



b) Three level



c) Four level

MEMORY TECHNOLOGIES

Much of the success of computer technology stems from the tremendous progress in storage technology.

Early computers had a few kilobytes of random-access memory. The earliest IBM PCs didn't even have a hard disk.

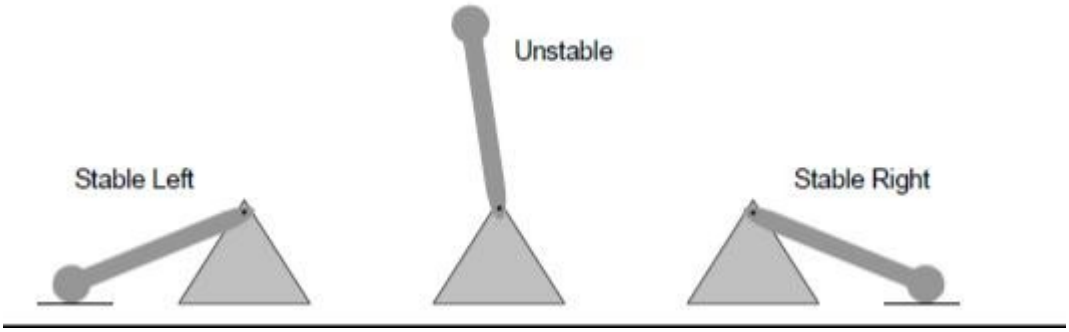
That changed with the introduction of the IBM PC-XT in 1982, with its 10-megabyte disk. By the year 2010, typical machines had 150,000 times as much disk storage, and the amount of storage was increasing by a factor of 2 every couple of years.

Random-Access Memory

Random-access memory (RAM) comes in two varieties—*static* and *dynamic*. *Static RAM* (SRAM) is faster and significantly more expensive than *Dynamic RAM* (DRAM). SRAM is used for cache memories, both on and off the CPU chip. DRAM is used for the main memory plus the frame buffer of a graphics system. Typically, a desktop system will have no more than a few megabytes of SRAM, but hundreds or thousands of megabytes of DRAM.

Static RAM

SRAMstores each bit in a *bistable* memory cell. Each cell is implemented with a six-transistor circuit. This circuit has the property that it can stay indefinitely in either of two different voltage configurations, or *states.* Any other state will be unstable—starting from there, the circuit will quickly move toward one of the stable



Dynamic RAM

DRAM stores each bit as charge on a capacitor. This capacitor is very small—typically around 30 femtofarads,that is, 30 × 10−15 farads. Recall, however, that a farad is a very large unit of measure. DRAM storage can be made very dense—each cell consists of a capacitor and a single access-transistor. Unlike SRAM, however, a DRAM memory cell is very sensitive to any disturbance. When the capacitor voltage is disturbed, it will never recover. Exposure to light rays will cause the capacitor voltages to change. In fact, the sensors in digital cameras and camcorders are essentially arrays of DRAM cells.

	Transistors per bit	Relative access time	Persistent?	Sensitive?	Relative cost	Applications
SRAM	6	1X	Yes	No	100X	Cache memory
DRAM	1	10X	No	Yes	1X	Main mem, frame buffers

Conventional DRAMs

The cells (bits) in a DRAM chip are partitioned into d *supercells*, each consisting of w DRAM cells. A d × w DRAM stores a total of dw bits of information. The supercells are organized as a rectangular array with r rows and c columns, where rc = d. Each supercell has an address of the form (i, j), where i denotes the row, and j denotes the column.

For example, Figure 6.3 shows the organization of a 16 × 8 DRAM chip with d = 16 supercells, w = 8 534 bits per supercell, r = 4 rows, and c = 4 columns. The shaded box denotes the supercell at address (2, 1).

Information flows in and out of the chip via external connectors called *pins*. Each pin carries a 1-bit signal.

Figure shows two of these sets of pins: eight data pins that can transfer 1 byte in or out of the chip, and two addr pins that carry two-bit row and column supercell addresses. Other pins that carry control information are not shown.

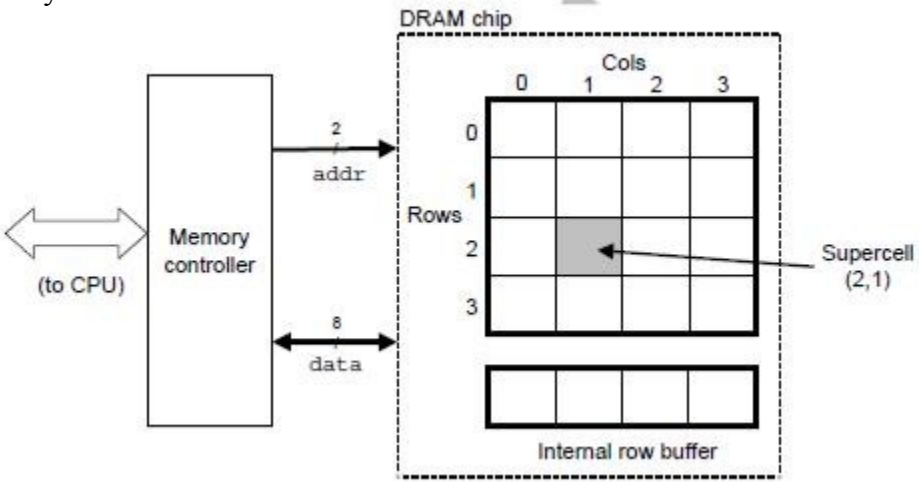
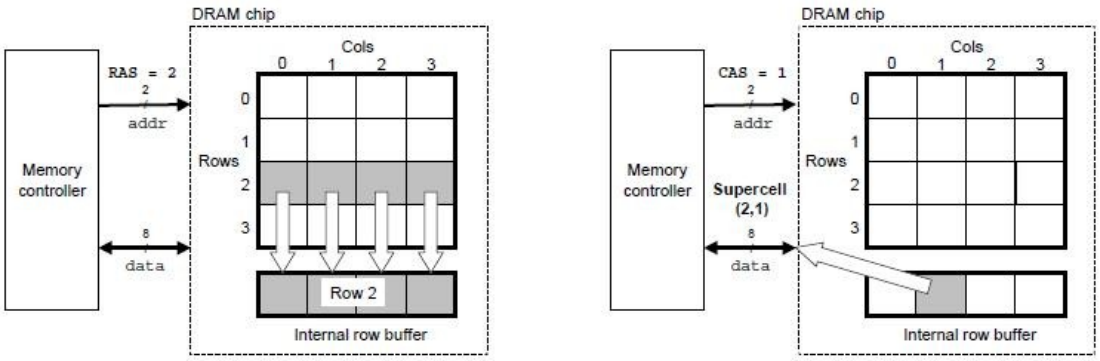


Fig: Conventionall DRAM

One reason circuit designers organize DRAMs as two-dimensional arrays instead of linear arrays is to reduce the number of address pins on the chip. For example, if our example 128-bit DRAM were organized as a linear array of 16 supercells with addresses 0 to 15, then the chip would need four address pins instead of two. The disadvantage of the two-dimensional array organization is that addresses must be sent in two distinct steps, which increases the access time.



(a) Select row 2 (RAS request). (b) Select column 1 (CAS request).

Enhanced DRAMs

There are many kinds of DRAM memories, and new kinds appear on the market with regularity as manufacturers attempt to keep up with rapidly increasing processor speeds. Each is based on the conventional DRAM cell, with optimizations that improve the speed with which the basic DRAM cells can be accessed.

Accessing Main Memory

Data flows back and forth between the processor and the DRAM main memory over shared electrical conduits called *buses*. Each transfer of data between the CPU and memory is accomplished with a series of steps called a *bus transaction*. A *read transaction* transfers data from the main memory to the CPU. A *write transaction* transfers data from the CPU to the main memory.

A *bus* is a collection of parallel wires that carry address, data, and control signals. Depending on the particular bus design, data and address signals can share the same set of wires, or they can use different sets. Also, more than two devices can share the same bus. The control wires carry signals that synchronize the transaction and identify what kind of transaction is currently being performed**.**

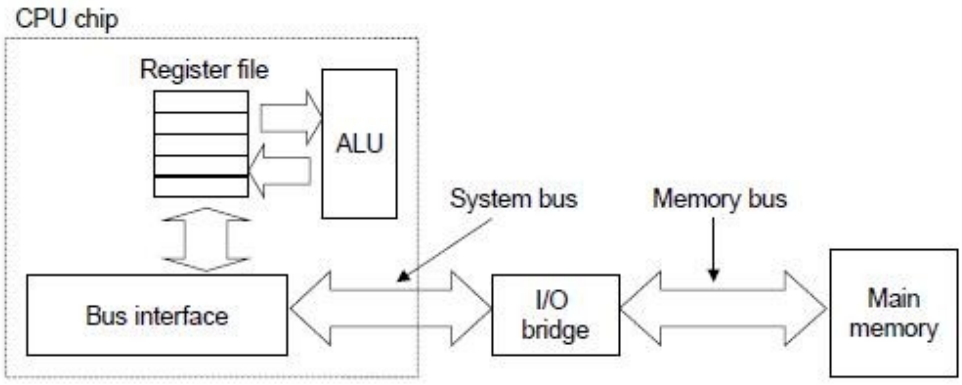
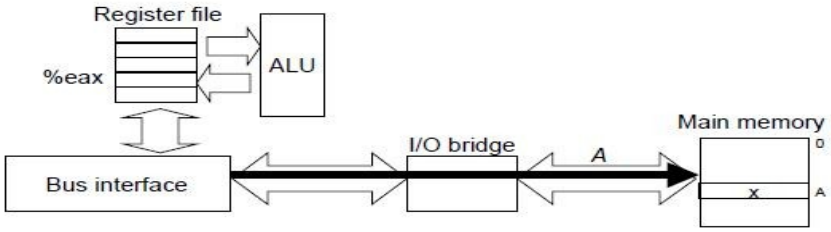


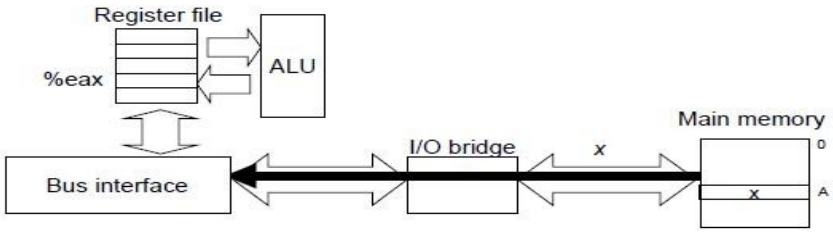
Figure : **Example bus structure that connects the CPU and main memory.**

Disk Storage

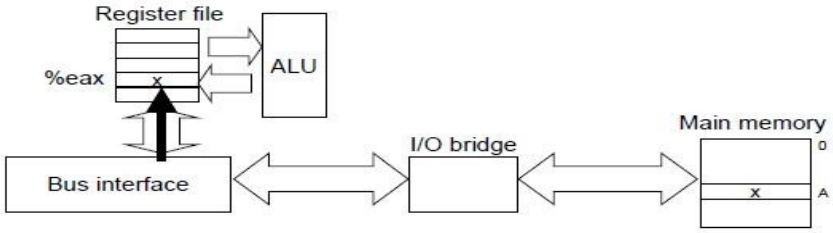
Disks are workhorse storage devices that hold enormous amounts of data, on the order of hundreds to thousands of gigabytes, as opposed to the hundreds or thousands of megabytes in a RAM-based memory. However, it takes on the order of milliseconds to read information from a disk, a hundred thousand times longer than from DRAM and a million times longer than from SRAM.



(a) CPU places address A on the memory bus.



(b) Main memory reads A from the bus, retrieves word x, and places it on the bus.



(c) CPU reads word x from the bus, and copies it into register `%eax`.

Figure: **Memory read transaction for a load operation**

CACHE BASICS – MEASURING AND IMPROVING CACHE PERFORMANCE

One focuses on reducing the miss rate by reducing the probability that two different memory blocks will contend for the same cache location. The second technique reduces the miss penalty by adding an additional level to the hierarchy. This technique, called *multilevel caching*, first appeared in high-end computers selling for more than $100,000 in 1990; since then it has become common on desktop computers selling for less than $500! CPU time can be divided into the clock cycles that the CPU spends executing the program and the clock cycles that the CPU spends waiting for the memory

system. Normally, we assume that the costs of cache accesses that are hits are part of the normal CPU execution cycles. Thus,

CPU time = (CPU execution clock cycles T Memory-stall clock cycles)

The memory-stall clock cycles come primarily from cache misses, and we make that assumption here. We also restrict the discussion to a simplified model of the memory system. In real processors, the stalls generated by reads and writes can be quite complex, and accurate performance prediction usually requires very detailed simulations of the processor and memory system.

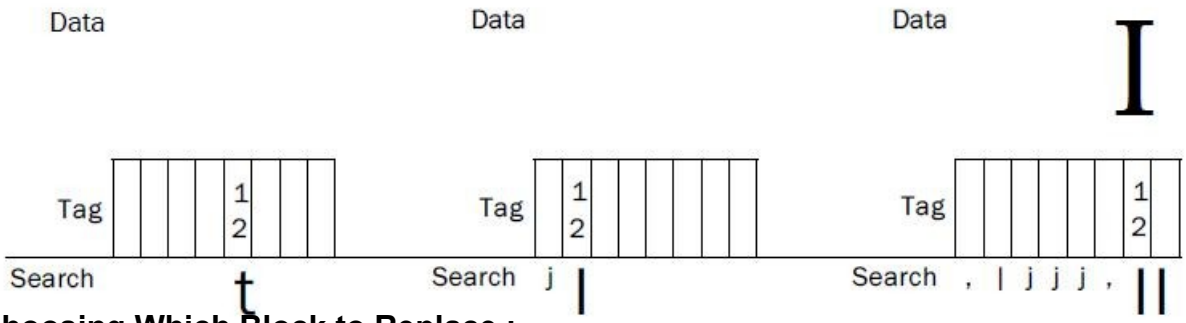
Reads Read-stall cycles =Pr ogram x Read miss rate x Read miss penalty Writes are more complicated. For a write-through scheme, we have two sources of stalls: write misses, which usually require that we fetch the block before continuing the write (see the *Elaboration* on page 467 for more details on dealing with writes), and write buffer stalls, which occur when the write buffer is full when a write occurs.

Calculating Cache Performance:

Assume the miss rate of an instruction cache is 2% and the miss rate of the data cache is 4%. If a processor has a CPI of 2 without any memory stalls and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect cache that never missed. Assume the frequency of all loads and stores is 36%.

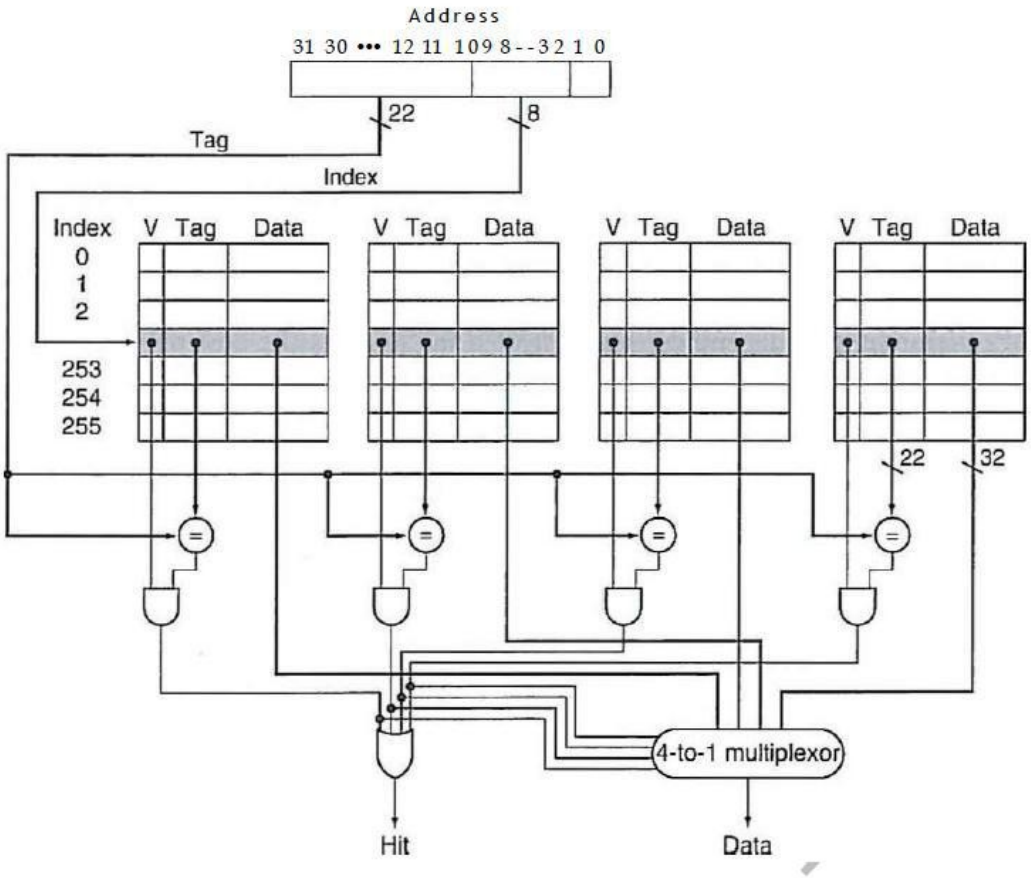
Reducing Cache Misses by *Move* Flexibfle Placement of Blocks

So far, when we place a block in the cache, we have used a simple placement scheme: A block can go in exactly one place in the cache. As mentioned earlier, it is called *direct mapped* because there is a direct mapping from any block address in memory to a single location in the upper level of the hierarchy. However, there is actually a whole range of schemes for placing blocks. Direct mapped, where a block can be placed in exactly one location, is at one extreme. At the other extreme is a scheme where a block can be placed in *any* location in the cache. Such a scheme is called **fully associative,** because a block in memory may be associated with any entry in the cache. To find a given block in a fully associative cache, all the entries in the cache must be searched because a block can be placed in any one. To make the search practical, it is done in parallel with a comparator associated with each cache entry. These comparators significantly increase the hardware cost, effectively making fully associative placement practical only forcaches with small numbers of blocks.



Choosing Which Block to Replace :

When a miss occurs in a direct-mapped cache, the requested block can go in exactly one position, and the block occupying that position must be replaced. In an associative cache, we have a choice of where to place the requested block, and hence a choice of which block to replace. In a fully associative cache, all blocks are candidates for replacement. In a set-associative cache, we must choose among the blocks in the selected set. The most commonly used scheme is **least recently used** (LRU), which we used in the previous example. In an LRU scheme, the block replaced is the one that has been unused for the longest time. The set associative example on page 482 uses LRU, which is why we replaced Memory(O) instead of Memory(6). LRU replacement is implemented by keeping track of when each element in a set was used relative to the other elements in the set. For a two-way set-associative cache, tracking when the two elements were used can be implemented by keeping a single bit in each set and setting the bit to indicate an element whenever that element is referenced. As associativity increases, implementing LRU gets harder; in Section 5.5, we will see an alternative scheme for replacement.



Two techniques can be used to improve cache performance are.

→ Reducing the miss rate by reducing the probability that two different memory blocks will contend for same cache location

→ Reducing the miss penalty by adding an additional level to hierarchy. This technique is called multilevel caching

$$\text{CPU time} = \left(\text{CPU execution clock cycles} + \text{Memory stall clock cycles}\right) \times \text{clock cycle time.}$$

→ cache hit indicates normal CPU execution cycles

→ cache misses indicates memory stall clock cycles.

→ Memory stall clock cycles are sum of stall cycles coming from read plus those coming from writes.

$$\text{Memory stall clock cycle} = \left(\text{Read stall cycles} + \text{write stall cycles}\right)$$

$$\text{Read stall cycles} = \frac{\text{Reads}}{\text{program}} \times \text{Read miss rate} \times \text{Read miss penalty.}$$

→ For a write through scheme, two sources of stalls : write misses and write buffer stalls.

Write Stall cycles = (Writes/program × write miss rate × write miss penalty) + write buffer stalls

One can combine read and write penalties together if they are same as

Memory stall clock cycles = Memory accesses/program × Miss rate × Miss penalty.

Also represented as

Memory Stall clock cycles = Instructions/program × Misses/Instruction × Miss penalty

Average Memory access time (AMAT)

It is the average time to access memory considering both hits and misses and the frequency of different accesses.

AMAT = Time for a hit + miss rate × miss penalty.

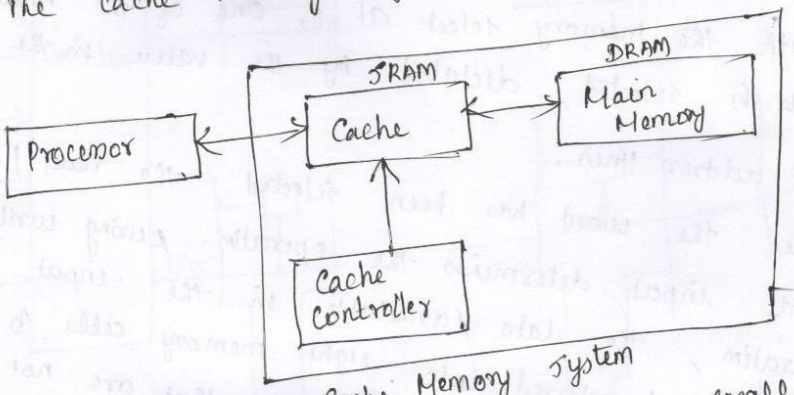
Reducing Cache misses by more flexible placement of blocks

In direct mapping, the position of memory block is given by

Cache Basics

The small section of SRAM memory added between processor and main memory to speed up execution process is known as cache memory.

The cache memory system is shown below



Cache Memory System

→ The cache memory system includes a small amount of fast memory (SRAM) and a large amount of slow memory (DRAM).

→ Cache controller implements the cache logic. If processor finds that the addressed code or data is not available in the cache - the condition referred as cache miss, the desired memory block is copied from main memory to cache using cache controller.

→ The cache block also known as cache slot or line

(11)

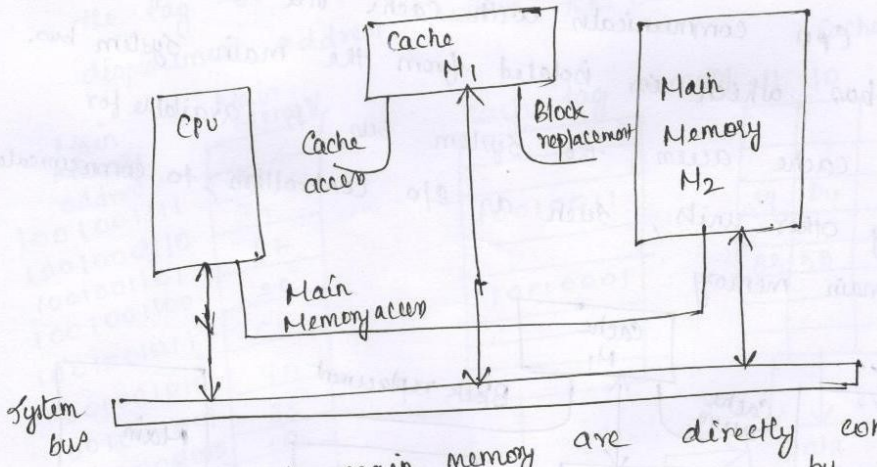
→ the percentage of accesses where the processor finds the code or data word it needs in the cache memory is called hit rate / hit ratio.

$$\text{Hit rate} = \frac{\text{Number of hits}}{\text{Total no of bus cycles}} \times 100\%.$$

Most commonly Used Cache Organization

1. Look aside
2. Look Through.

Look aside System organization



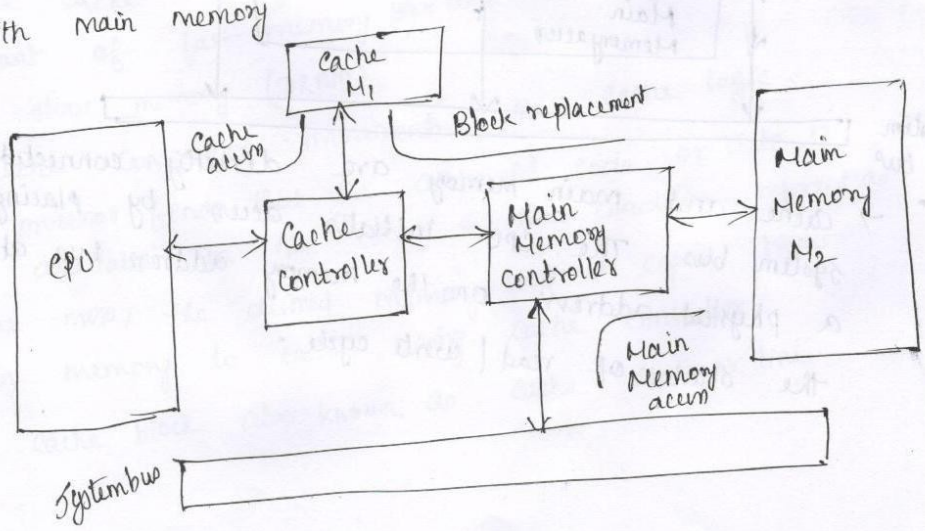
→ Cache and main memory are directly connected to system bus. The CPU initiates accen by placing a physical address on the memory addren bus at the start of read / write cycle.

→ M_1 immediately compares physical address to tag address currently residing in tag memory. If match is found, ~~the cache~~ in case of cache hit, the access is completed by read / write operation executed in the cache.

→ If ~~matbe~~ match is not found, in case of cache miss, the desired access is completed by a read / write operation directed to M_2. The system bus is used to transfer ~~the~~ target address from M_2 to M_1.

Look Through System Organization

CPU communicates with cache via a separate (local) bus which is isolated from the main system bus. During cache access, the system bus is available for use by other units, such as I/O controllers, to communicate with main memory



(12)

→ Look Through cache system does not automatically send all memory request to main memory; it does so only after a cache miss

→ Look Through cache systems use wider local bus to link M_1 & M_2, thus speeding up cache main memory transfers. It is faster.

Cache Read operation

→ Here each cache block is 4 bytes and each memory is 10 bit long. 8 higher order bits form the tag or block addrn & 2 lower order bits define displacement addrn within the block.

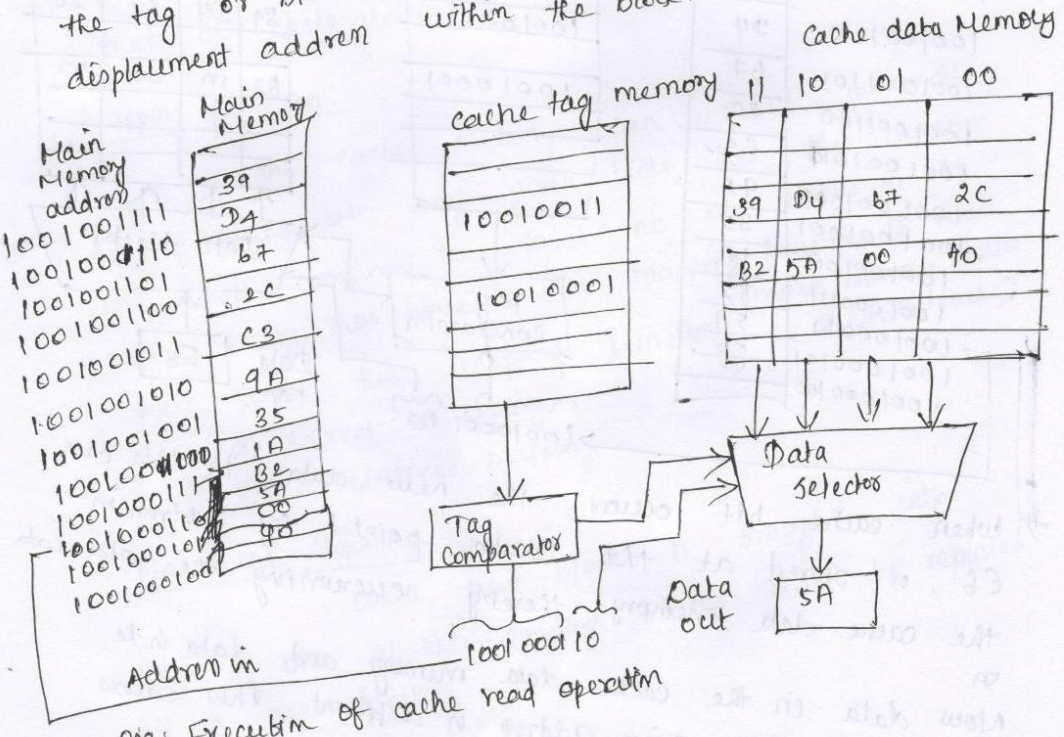
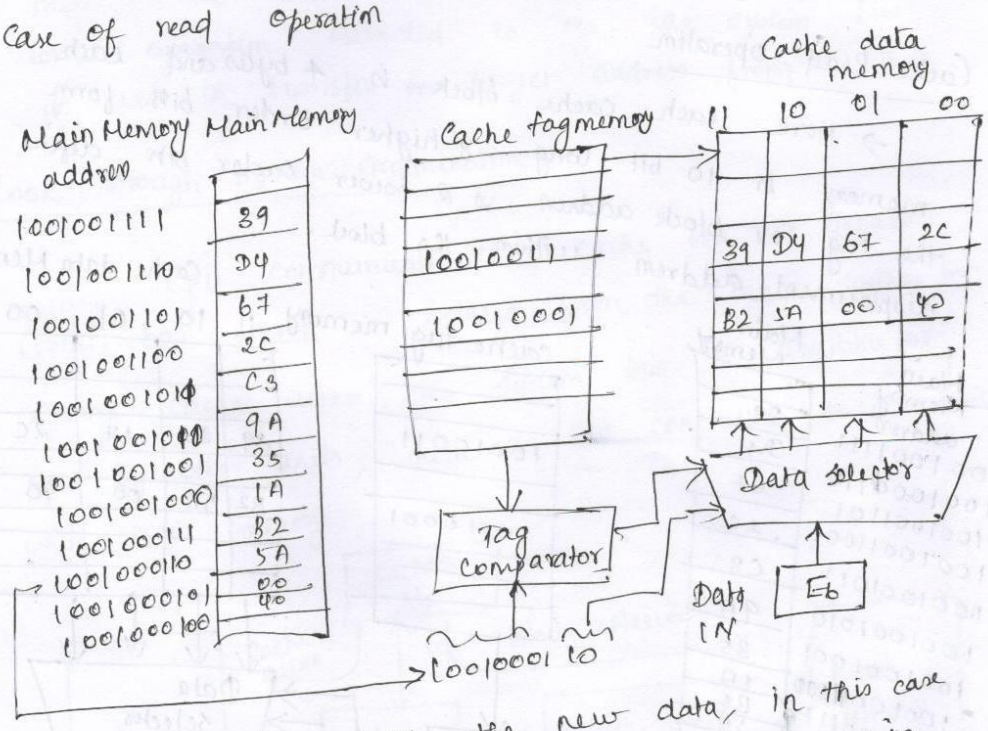


Fig: Execution of cache read operation

→ The stored tag pinpoints the corresponding block in cache data memory and 2-bit displacement is used to read the target word.

Cache write operation

It uses same addressing technique as in case of read operation



→ When cache hit occurs, the new data, in this case E6, is stored at the location pointed by address in the cache data memory, thereby overwriting the old data 5A

Now data in the cache data memory and data in the main memory for given address is different. This causes cache consistency problem

Elements of cache design

The cache design elements include cache size, mapping function, replacement algorithm write policy, block size and number of caches.

Cache size:-

The size of the cache should be small enough so that the overall average cost per bit is close to that of main memory and large enough so that the overall average access time is close to that of cache alone

Mapping function

The cache memory can store reasonable no of blocks at any given time, but this number is small compared to total no of blocks in the main memory. Thus have to use mapping functions to relate (main memory blocks & cache blocks)

There are two mapping function

a) Direct mapping
b) Associative mapping

Replacement Algorithm

When new block is brought into cache, one of the existing blocks must be replaced by a new block.

There are four most common replacement algorithms.

1. Least Recently Used (LRU)
2. First IN First OUT (FIFO)
3. Least Frequently Used (LFU)
4. Random

Cache Coherency

In a single CPU system, two copies of same data, one in cache memory and another in main memory may become different. This data inconsistency is called as cache coherence problem.

To protect cache coherence, there are four different approaches

1. Bus watching (snooping) – main memory resides in cache memory
2. Hardware transparency – access to all devices to main memory are routed through same cache
3. Non-cacheable memory
4. Cache flushing

By designing shared memory as noncacheable memory cache coherency can be maintained, since shared memory never copied to cache

To avoid data inconsistency, a cache flush writes any altered data to main memory and caches in the system are flushed before a device writes to shared memory

Cache Updating policies

Two different sets of data become associated with same address. To prevent this, the cache system has updating systems such as write through system, buffered write through system and write back system.

Write through system

The cache controller copies data to main memory immediately after it is written to cache. Due to this, main memory always contains a valid data and thus any block in the cache can be overwritten immediately without data loss.

Buffered write through system

In buffered write through system, the processor can start a new cycle before the write cycle to the main memory is completed. This means that the write accesses to main memory are buffered.

Write back system

All altered blocks must be written to main memory before another device can access these blocks in main memory.

Measuring and Improving Cache Performance

The performance of a memory system depends mainly on the following

Addren reference statistics

It means the order and frequency of the logic addren generated by programs that use the memory hierarchy.

Accen time :- the accen time (t_A) of each memory level relative to the CPU

Storage capacity

Storage capacity of each memory level.

Block size

The size of blocks (pages) transferred between adjacent levels.

Allocatin Algorithm

The algorithm used to determine the regions of memory to which blocks are transferred by the block replacement procen.

(Block number) modulo (Number of blocks in the cache)

→ In set associative mapping, there is a set that contains the memory block and it is given as

(Block number) modulo (Number of sets in the cache)

Thus to access a particular data all the tags of all elements within the particular set must be searched.

→ In a fully associative mapping, as the block goes anywhere, all the tags of all the blocks must be searched.

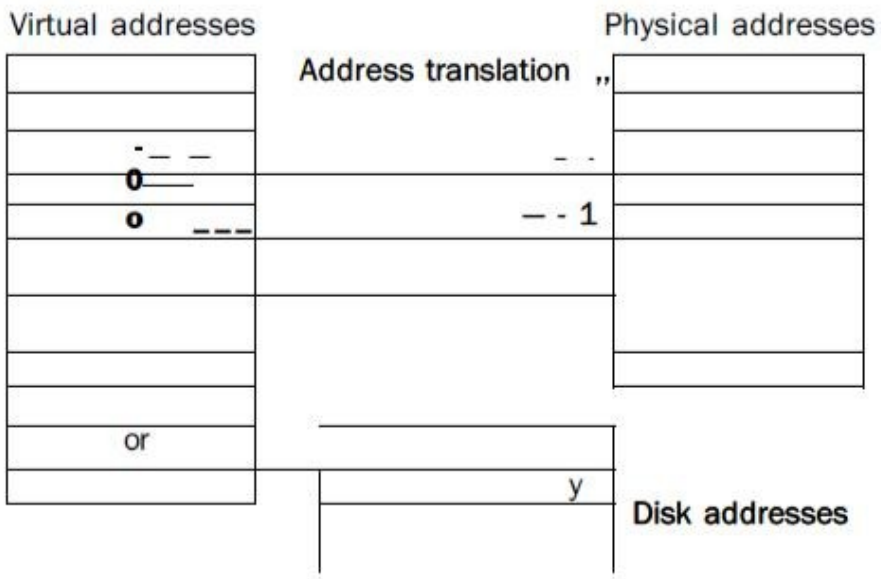
Thus increase in associativity decreases the miss rate and hence increases the performance of cache.

VIRTUAL MEMORY

Similarly, the main memory can act as a "cache" for the secondary storage, usually implemented with magnetic disks. This technique is called virtual memory. Historically, there were two major motivations for virtual memory: to allow efficient and safe sharing of memory among multiple programs, and to remove the programming burdens of a small, limited amount of main memory. Four decades after its invention, it's the former reason that reigns today.

Consider a collection of programs running all at once on a computer. Of course, to allow multiple programs to share the same memory, we must be able to protect the programs from each other, ensuring that a program can only read and write the portions of main memory that have been assigned to it. Main memory need contain only the active portions of the many programs, just as a cache contains only the active portion of one program. Thus, the principle of locality enables virtual memory as well as caches, and virtual memory allows us to efficiently share the processor as well as the main memory.

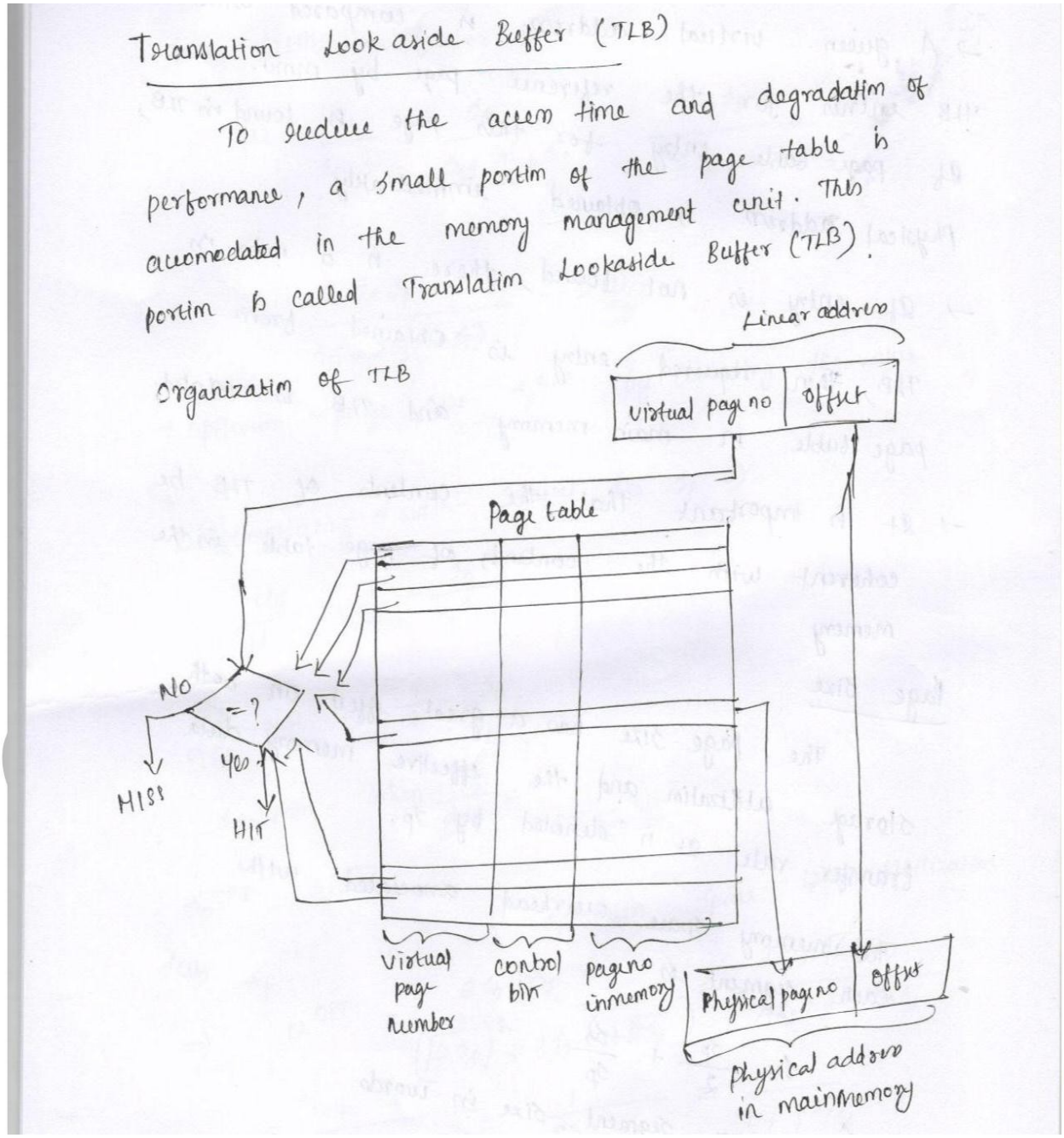
The second motivation for virtual memory is to allow a single user program to exceed the size of primary memory. Formerly, if a program became too large for memory, it was up to the programmer to make it fit. Programmers divided programs into pieces and then identified the pieces that were mutually exclusive. These *overlays* were loaded or unloaded under user program control during execution, with the programmer ensuring that the program never tried to access an overlay that was not loaded and that the overlays loaded never exceeded the total size of the memory. Overlays were traditionally organized as modules, each containing both code and data.



In virtual memory, the address is broken into a *virtual page number* and a *page offset.* Figure 5.20 shows the translation of the virtual page number to a *physical page number.* The physical page number constitutes the upper portion of the physical address, while the page offset, which is not changed, constitutes the lower portion. The number of bits in the page offset field determines

the page size. The number of pages addressable with the virtual address need not match the number of pages addressable with the physical address. Having a larger number of virtual pages than physical pages is the basis for the illusion of an essentially unbounded amount of virtual memory.

TLBS - INPUT/OUTPUT SYSTEM



→ A given virtual address is compared with TLB entries for the reference page by mmu.

If page table entry for this page is found in TLB, physical address is obtained immediately.

→ If entry is not found, there is a miss in TLB, then required entry is obtained from page table in main memory and TLB is updated.

→ It is important that the contents of TLB be coherent with the contents of page table in the memory

Page size

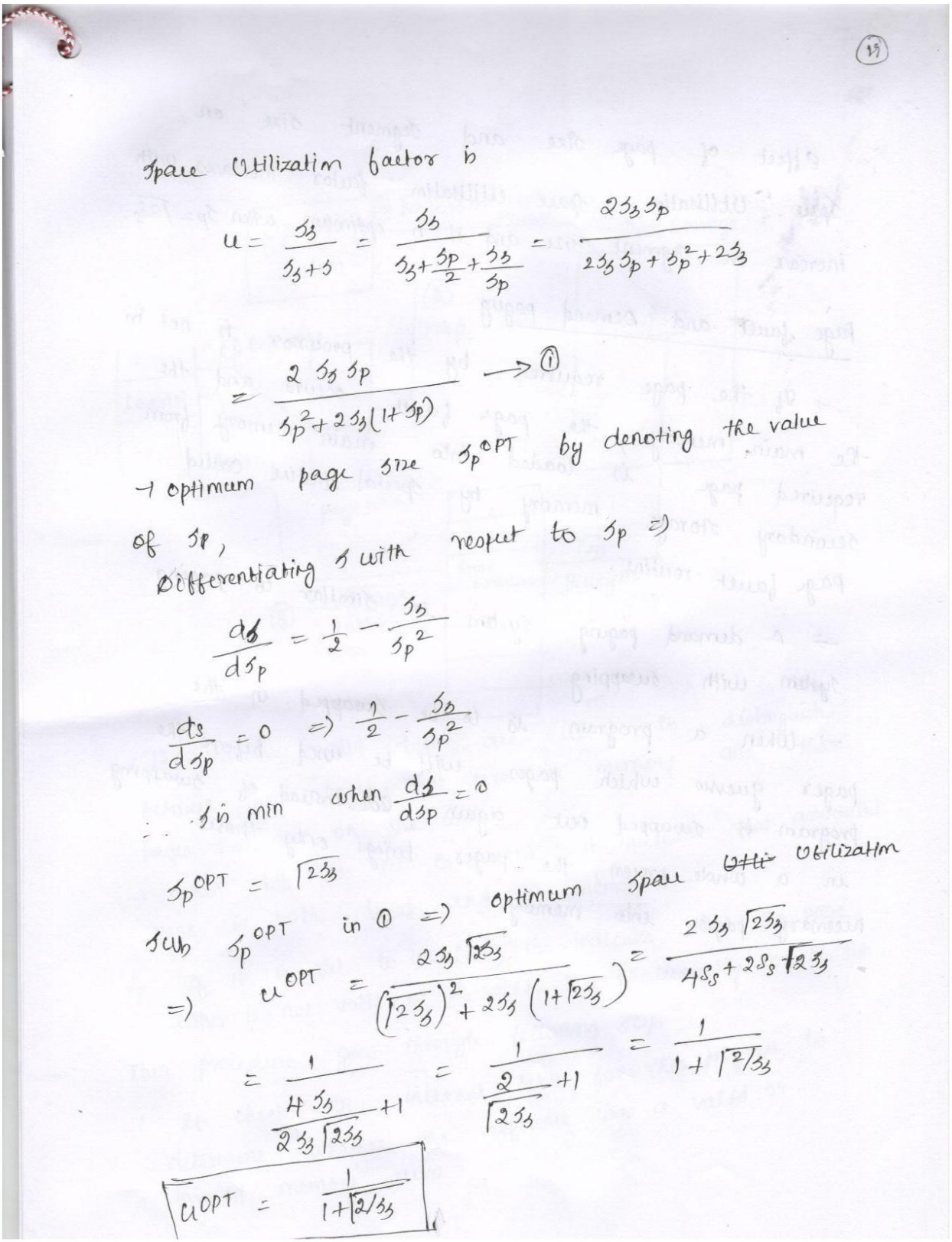
The page size has a great effect on both storage utilization and the effective memory data transfer rate. It is denoted by s_p.

The memory space overhead associated with each segment is

$$s = \frac{s_p}{2} + \frac{s_s}{s_p}$$

s_s - average segment size in words

$\frac{s_s}{s_p}$ - size of the page table



Space Utilization factor is

$$u = \frac{s_s}{s_s + s} = \frac{s_s}{s_s + \frac{s_p}{2} + \frac{s_s}{s_p}} = \frac{2 s_s s_p}{2 s_s s_p + s_p^2 + 2 s_s}$$

$$= \frac{2 s_s s_p}{s_p^2 + 2 s_s (1 + s_p)} \rightarrow (1)$$

→ optimum page size s_p^{OPT} by denoting the value of s_p,

Differentiating s with respect to s_p ⇒

$$\frac{ds}{ds_p} = \frac{1}{2} - \frac{s_s}{s_p^2}$$

$$\frac{ds}{ds_p} = 0 \Rightarrow \frac{1}{2} - \frac{s_s}{s_p^2}$$

∴ s is min when $\frac{ds}{ds_p} = 0$

$$s_p^{OPT} = \sqrt{2 s_s}$$

Sub s_p^{OPT} in (1) ⇒ optimum Space Utilization

$$\Rightarrow u^{OPT} = \frac{2 s_s \sqrt{2 s_s}}{(\sqrt{2 s_s})^2 + 2 s_s (1 + \sqrt{2 s_s})} = \frac{2 s_s \sqrt{2 s_s}}{4 s_s + 2 s_s \sqrt{2 s_s}}$$

$$= \frac{1}{\frac{4 s_s}{2 s_s \sqrt{2 s_s}} + 1} = \frac{1}{\frac{2}{\sqrt{2 s_s}} + 1} = \frac{1}{1 + \sqrt{2/s_s}}$$

$$\boxed{u^{OPT} = \frac{1}{1 + \sqrt{2/s_s}}}$$

Effect of page size and segment size on space utilisation.

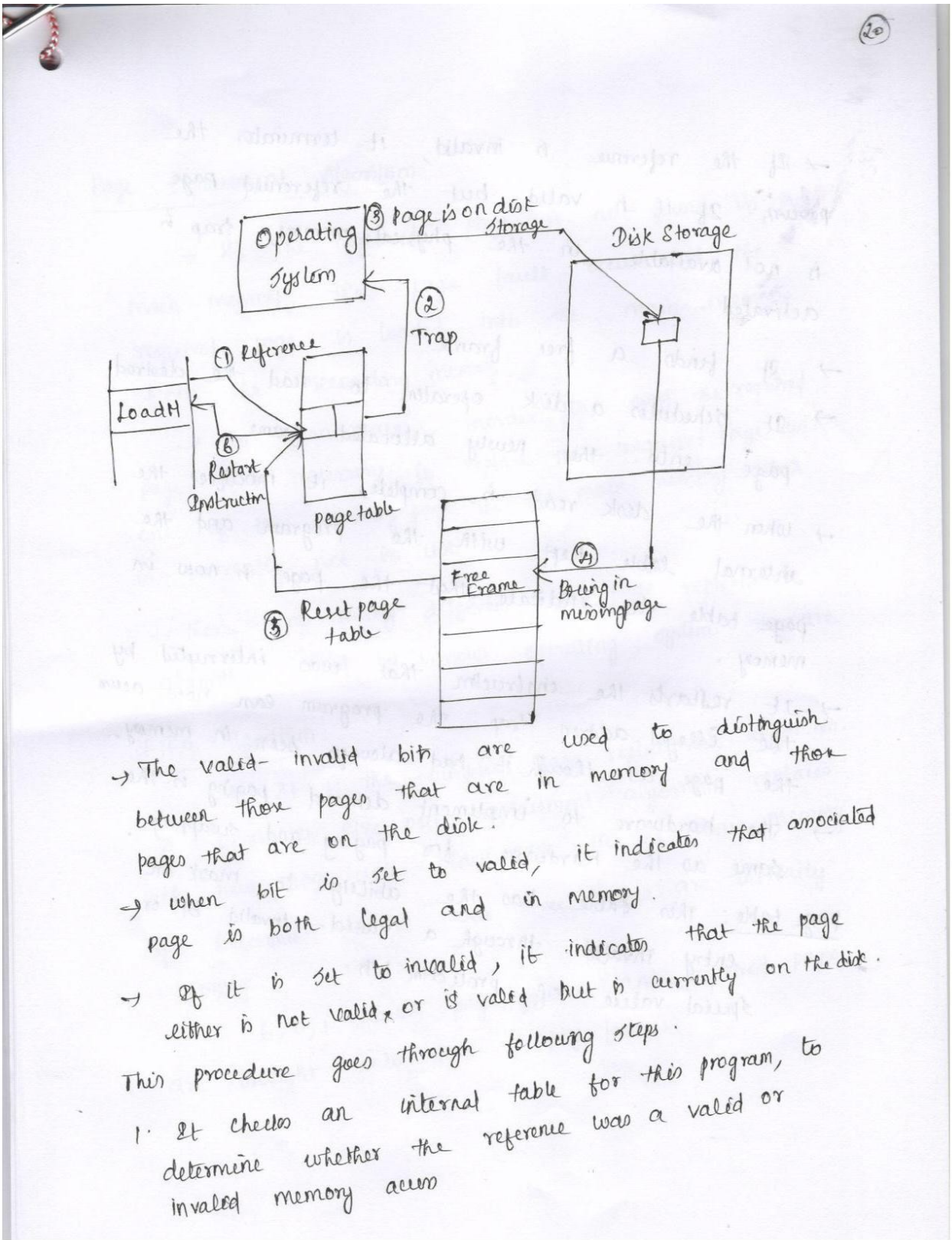
Space utilisation factor increases with increase in segment size and it is optimum when $S_p = \sqrt{2S_s}$.

Page fault and demand paging

→ If the page required by the processor is not in the main memory, the page fault occurs and the required page is loaded into main memory from secondary storage memory by special routine called page fault routine.

→ A demand paging system is similar to paging system with swapping

→ When a program is to be swapped in the pager guesses which pages will be used before the program is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory.



→ The valid- invalid bit are used to distinguish between those pages that are in memory and those pages that are on the disk.

→ when bit is set to valid, it indicates that associated page is both legal and in memory.

→ If it is set to invalid, it indicates that the page either is not valid or is valid but is currently on the disk.

This procedure goes through following steps.

1. It checks an internal table for this program, to determine whether the reference was a valid or invalid memory acces

→ If the reference is invalid, it terminates the process. If it is valid, but the referenced page is not available in the physical memory, trap is activated.

→ It finds a free frame

→ It schedules a disk operation to read the desired page into the newly allocated frame.

→ When the disk read is complete, it modifies the internal table kept with the program and the page table to indicate that the page is now in memory.

→ It restarts the instruction that was interrupted by the illegal address trap. The program can now access the page as though it had always been in memory.

→ The hardware to implement demand paging is the same as the hardware for paging and swapping.

Page table: This table has the ability to mark an entry invalid through a valid – invalid bit or special value of protection bit

Page Replacement Algorithm

→ If the required page is not found in the main memory, the page fault occurs and the required page is loaded into the main memory from the secondary memory.

→ no vacant space, inorder to copy the required page, it is necessary to replace the required page with one of the existing page in the main memory which is currently not in use.

→ There are many different page replacement algorithms used by various operating systems. They are

FIFO Algorithm

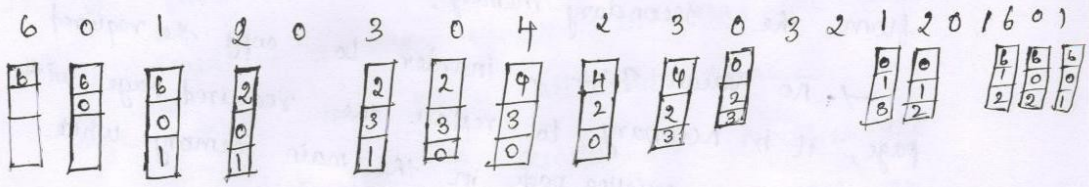
→ It is the simplest page replacement algorithm. A First In First out replacement algorithm replaces the new page with oldest page in the main memory

Eg: Reference string, our three frames are initially empty.

6, 0, 1 cause page faults and required pages are brought into these empty frames.

Next reference (2) replaces page 6, because page 6 was brought in first. Since 0 is next reference and 0 is already in memory, no fault for this reference

Reference String



page frames

Fig: FIFO page replacement Algorithm

→ First reference to 3 results in page 0 being replaced, since it was the first of the three pages in memory (0, 1, 2) to be brought in. This process continues as in Fig above ~~shown from~~ shown above.

→ FIFO page replacement algorithm is easy to understand and program. It may replace the most needed page as its oldest page.

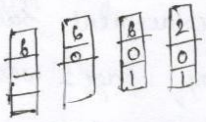
Optimal Algorithm

An optimal page replacement algorithm has the lowest page fault rate of all algorithms. It has been called OPT or MIN

It states that Replace the page that will not be used for the longest period of time.

Eg:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
6	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	6	0	1



page frames

The reference to page 2 replaces page 6, 6 will not be used untill reference 18, whereas page 0 will be used at 5 and page 1 at 14. page 3 replaces page 1 as page 1 will be the last of three pages in memory to be referenced again. No replacement algorithm can process the reference string in three frames with less than nine faults.

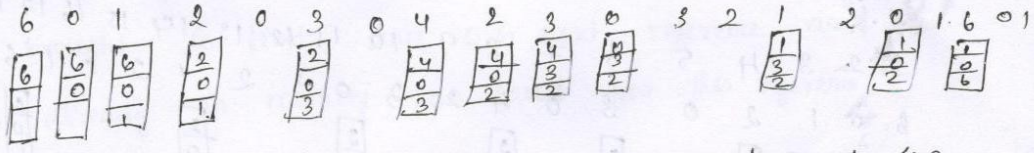
Disadvantage

It is difficult to implement, because it requires future knowledge of reference string.

LRU algorithm

The algorithm which replaces the page that has not been used for the longest period of time is referred as Least Recently Used algorithm (LRU).

Reference String



→ when reference to page 4 occurs, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. recently used page is 0 and just before that page 3 was used. LRU algorithm replaces page 2 not knowing that page 2

→ when page 2 fault, LRU replaces page 3, among (0,3,4) page 3 is least recently used

→ LRU with 12 faults is still much better than FIFO replacement with 15.

Counting algorithm

In the counting algorithm, a counter of the number of references that have been made to each pages are kept, and based on these counts, the following two schemes work.

LFU algorithm

The least frequently used page replacement algorithm requires page with smallest count be replaced

MFU algorithm :- Most frequently Used page replacement algorithm is based on argument that page with smallest count was probably just brought in and yet to be used

Input/Output

The computer system's I/O architecture is its interface to the outside world. This architecture is designed to provide a systematic means of controlling interaction with the outside world and to provide the operating system with the information it needs to manage I/O activity effectively.

There are three principal I/O techniques: programmed I/O, in which I/O occurs under he direct and continuous control of the program requesting the I/O operation; interrupt-driven I/O, in which a program issues an I/O command and then continues to execute, until it is interrupted by the I/O hardware to signal the end of the I/O operations; and direct memory access (DMA), in which a specialized I/O processor takes over control of an I/O operation to move a large block of data.

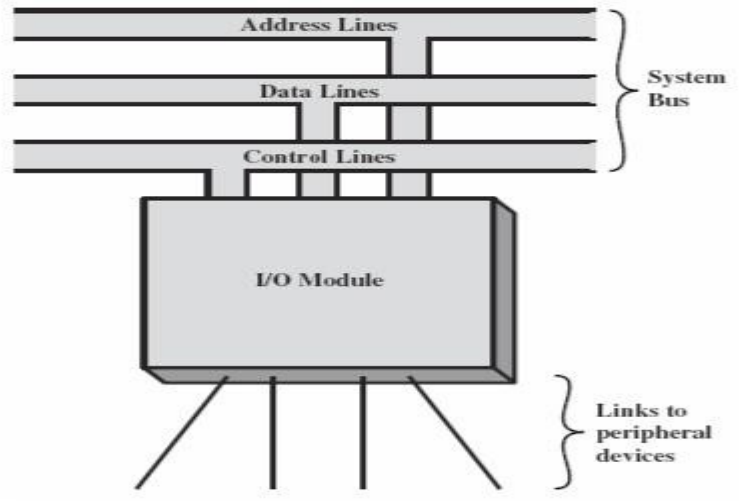
Two important examples of external I/O interfaces are FireWire and Infiniband.

Peripherals and the System Bus

There are a wide variety of peripherals each with varying methods of operation
Impractical to for the processor to accommodate all
Data transfer rates are often slower than the processor and/or memory
Impractical to use the high-speed system bus to communicate directly
Data transfer rates may be faster than that of the processor and/or memoryThis mismatch may lead to inefficiencies if improperly managed
Peripheral often use different data formats and word lengths Purpose of I/O Modules
Interface to the processor and memory via the system bus or control switch Interface to one or more peripheral devices

Purpose of I/O Modules

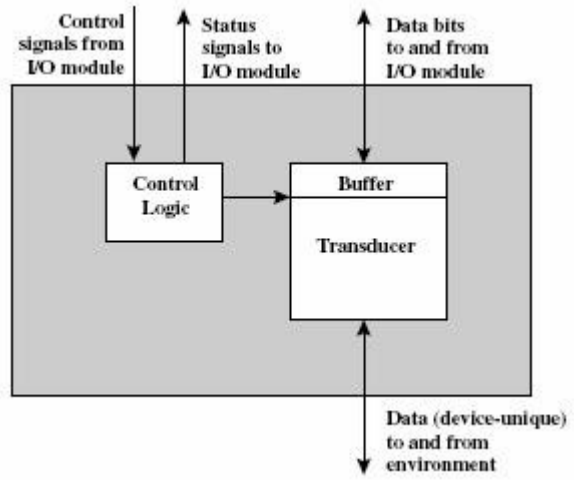
- Interface to the processor and memory via the system bus or control switch
- Interface to one or more peripheral devices



External Devices:

External device categories

- **Human readable:** communicate with the computer user – CRT
- **Machine readable:** communicate with equipment – disk drive or tape drive
  - **Communication:** communicate with remote devices – may be human readable or machine readable



The External Device – I/O Module

- **Control signals:** determine the function that will be performed
- **Data:** set of bits to be sent of received
- **Status signals:** indicate the state of the device
- **Control logic:** controls the device's operations
- **Transducer:** converts data from electrical to other forms of energy
- **Buffer:** temporarily holds data being transferred

Keyboard/Monitor

- Most common means of computer/user interaction
- Keyboard provides input that is transmitted to the computer
- Monitor displays data provided by the computer
- The character is the basic unit of exchange
- Each character is associated with a 7 or 8 bit code

Disk Drive

- Contains electronics for exchanging data, control, and status signals with an I/O module
- Contains electronics for controlling the disk read/write mechanism
  - Fixed-head disk – transducer converts between magnetic patterns on the disk surface and bits in the buffer
- Moving-head disk – must move the disk arm rapidly across the surface

I/O Modules

Module Function

- Control and timing
- Processor communication

- Device communication
- Data buffering
- Error detection

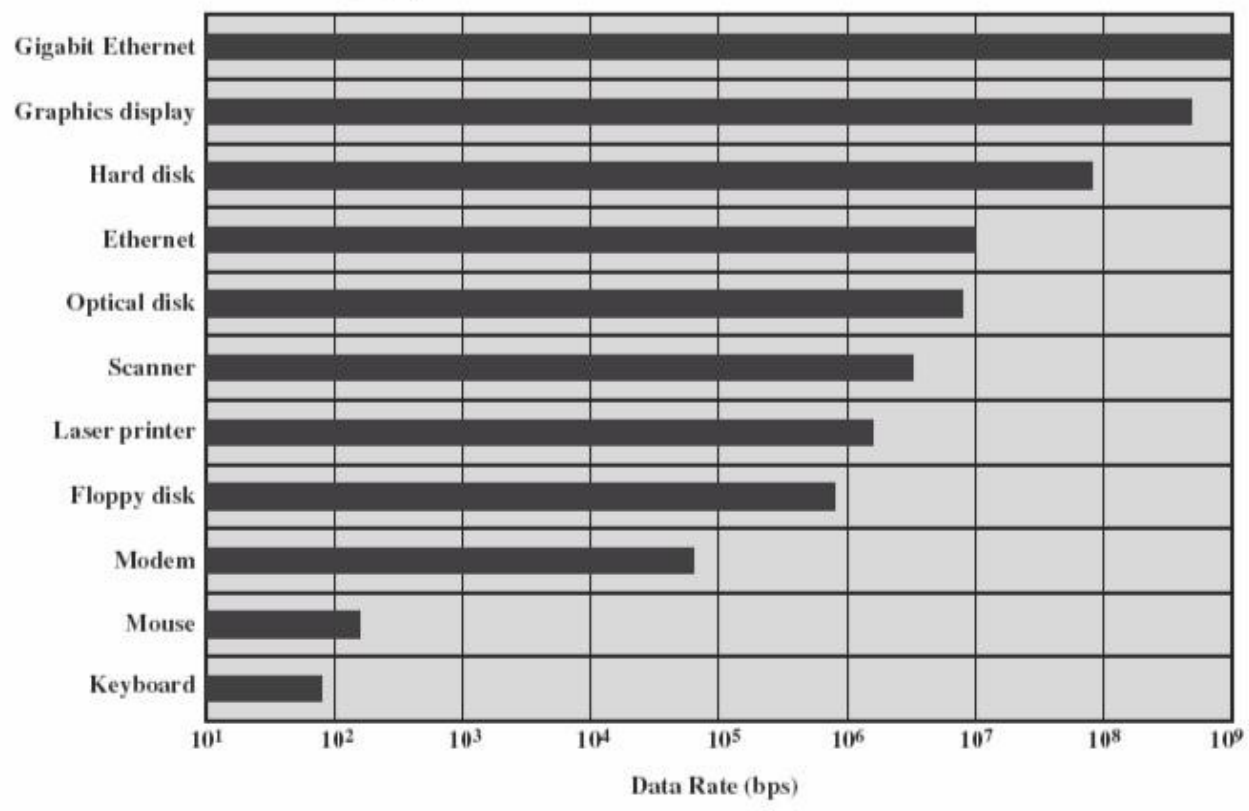
I/O control steps

- Processor checks I/O module for external device status
- I/O module returns status
- If device ready, processor gives I/O module command to request data transfer
- I/O module gets a unit of data from device
- Data transferred from the I/O module to the processor

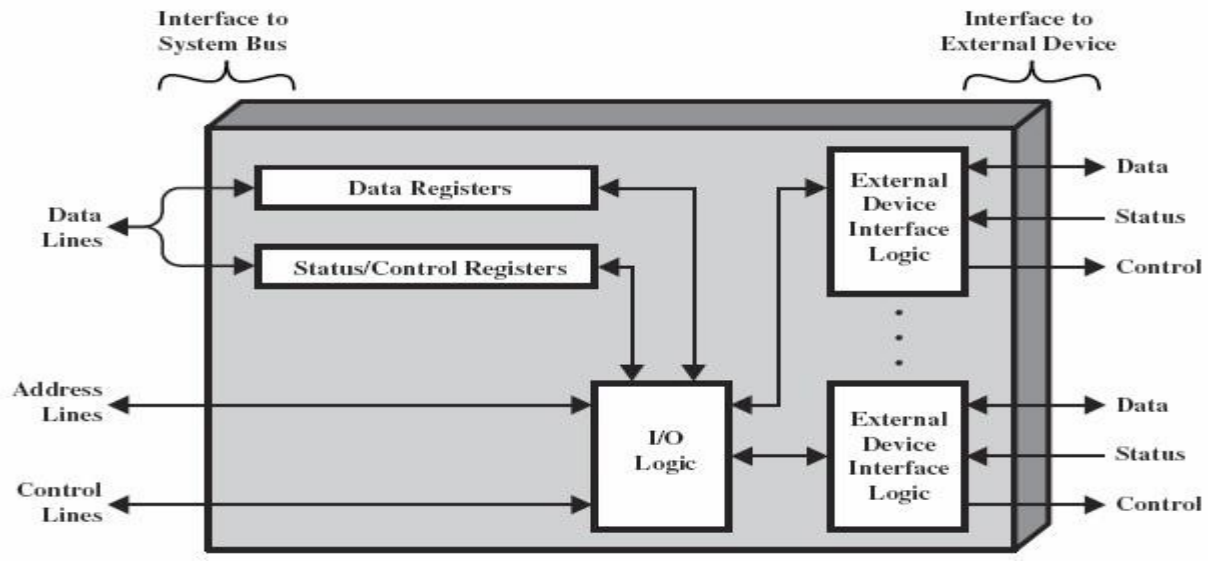
Processor communication

Command decoding: I/O module accepts commands from the processor sent as signals on the control bus
Data: data exchanged between the processor and I/O module over the data bus Status reporting: common status signals BUSY and READY are used because peripherals are slow
Address recognition: I/O module must recognize a unique address for each peripheral that it controls I/O module communication
Device communication: commands, status information, and data
Data buffering: data comes from main memory in rapid burst and must be buffered by the I/O module and then sent to the device at the device's rate
Error detection: responsible for reporting errors to the processor

Typical I/O Device Data Rates



I/O Module Structure: Block Diagram of an I/O Module



Module connects to the computer through a set of signal lines – system bus

- Data transferred to and from the module are buffered with data registers
- Status provided through status registers – may also act as control registers
- Module logic interacts with processor via a set of control signal lines
- Processor uses control signal lines to issue commands to the I/O module
- Module must recognize and generate addresses for devices it controls
- Module contains logic for device interfaces to the devices it controls

- I/O module functions allow the processor to view devices is a simple-minded way
  - I/O module may hide device details from the processor so the processor only functions in terms of simple read and write operations – timing, formats, etc…
  - I/O module may leave much of the work of controlling a device visible to the processor – rewind a tape, etc…

I/O channel or I/O processor

- I/O module that takes on most of the detailed processing burden
- Used on mainframe computers

I/O controller of device controller

- Primitive I/O module that requires detailed control
- Used on microcomputers

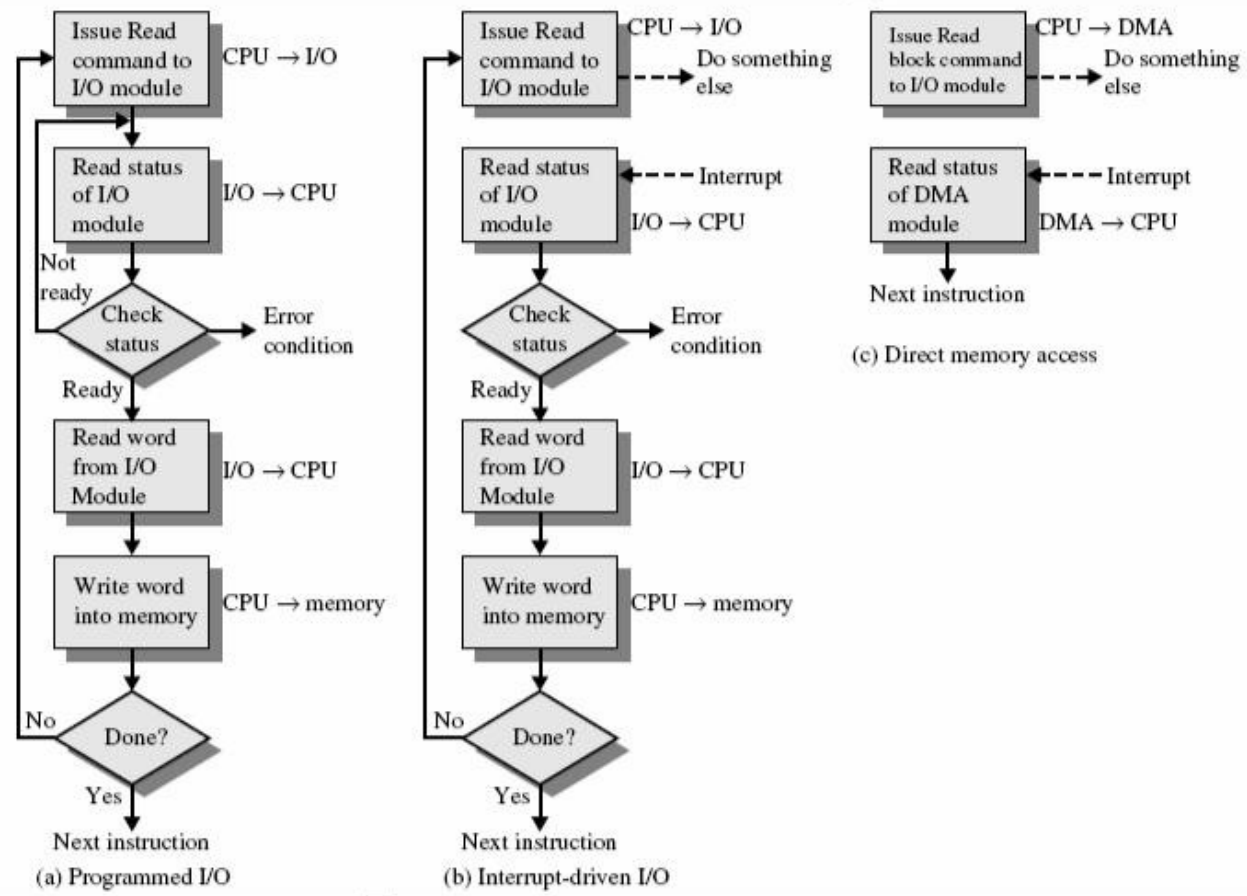
PROGRAMMED I/O

Overview of Programmed I/O

• Processor executes an I/O instruction by issuing command to appropriate I/O module

- I/O module performs the requested action and then sets the appropriate bits in the I/O status register – I/O module takes not further action to alert the processor – it does not interrupt the processor
- The processor periodically checks the status of the I/O module until it determines that the operation is complete I/O Commands The processor issues an address, specifying I/O module and device, and an I/O command. The commands are:

- **Control:** activate a peripheral and tell it what to do
- **Test:** test various status conditions associated with an I/O module and its peripherals
  - **Read:** causes the I/O module to obtain an item of data from the peripheral and place it into an internal register
- **Write:** causes the I/O module to take a unit of data from the data bus and



Three Techniques for Input of a Block of Data

I/O Instructions

Processor views I/O operations in a similar manner as memory operations Each device is given a unique identifier or address

Processor issues commands containing device address – I/O module must check address lines to see if the command is for itself.

I/O mapping

Memory-mapped I/O

¾ Single address space for both memory and I/O devices

- Disadvantage – uses up valuable memory address space

¾ I/O module registers treated as memory addresses
¾ Same machine instructions used to access both memory and I/O devices

- Advantage – allows for more efficient programming

¾ Single read line and single write lines needed
¾ Commonly used

- **Isolated I/O**

¾ Separate address space for both memory and I/O devices
¾ Separate memory and I/O select lines needed
¾ Small number of I/O instructions
¾ Commonly used

DMA AND INTERRUPTS

Interrupt-Driven I/O

- Overcomes the processor having to wait long periods of time for I/O modules
- The processor does not have to repeatedly check the I/O module status

I/O module view point

- I/O module receives a READ command form the processor
- I/O module reads data from desired peripheral into data register
- I/O module interrupts the processor
- I/O module waits until data is requested by the processor
- I/O module places data on the data bus when requested

Processor view point

- The processor issues a READ command
- The processor performs some other useful work
- The processor checks for interrupts at the end of the instruction cycle
- The processor saves the current context when interrupted by the I/O module
- The processor read the data from the I/O module and stores it in memory
- The processor the restores the saved context and resumes execution

Design Issues

How does the processor determine which device issued the interrupt How are multiple interrupts dealt with Device identification

Multiple interrupt lines – each line may have multiple I/O modules Software poll – poll each I/O module Separate command line – TESTI/O

Processor read status register of I/O module

Time consuming

Daisy chain

Hardware poll
Common interrupt request line
Processor sends interrupt acknowledge
Requesting I/O module places a word of data on the data lines – ―vector‖ that uniquely identifies the I/O module – vectored interrupt

- **Bus arbitration**
  - ¾ I/O module first gains control of the bus
  - ¾ I/O module sends interrupt request
  - ¾ The processor acknowledges the interrupt request
  - ¾ I/O module places its vector of the data lines
  - ¾

Multiple interrupts

- The techniques above not only identify the requesting I/O module but provide methods of assigning priorities

- Multiple lines – processor picks line with highest priority
- Software polling – polling order determines priority
- Daisy chain – daisy chain order of the modules determines priority
- Bus arbitration – arbitration scheme determines priority

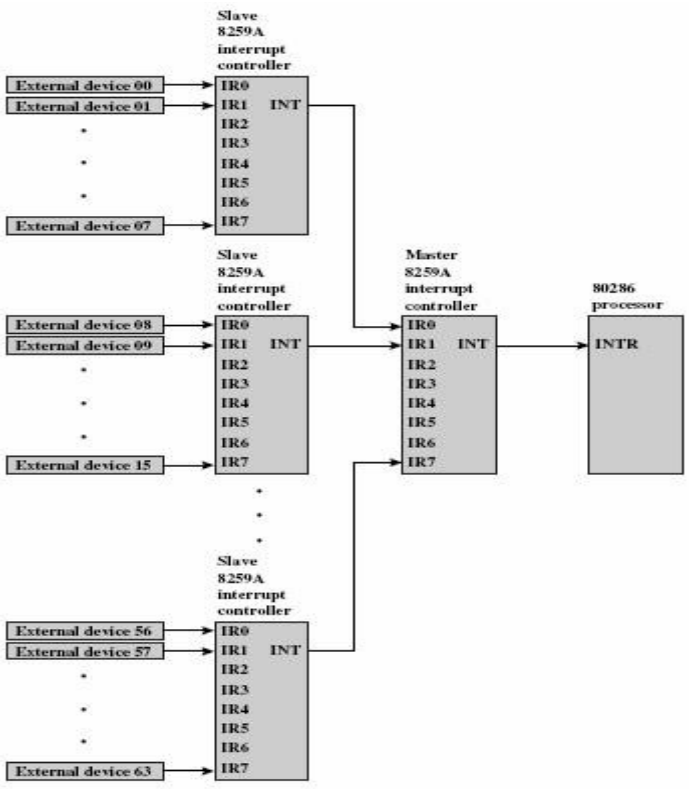
Intel 82C59A Interrupt Controller

Intel 80386 provides

- Single Interrupt Request line – INTR
- Single Interrupt Acknowledge line – INTA
  - Connects to an external interrupt arbiter, 82C59A, to handle multiple devices and priority structures
- 8 external devices can be connected to the 82C59A – can be cascaded to 64 82C59A operation – only manages interrupts
- Accepts interrupt requests
- Determines interrupt priority
- Signals the processor using INTR
- Processor acknowledges using INTA
- Places vector information of data bus
- Processor process interrupt and communicates directly with I/O module

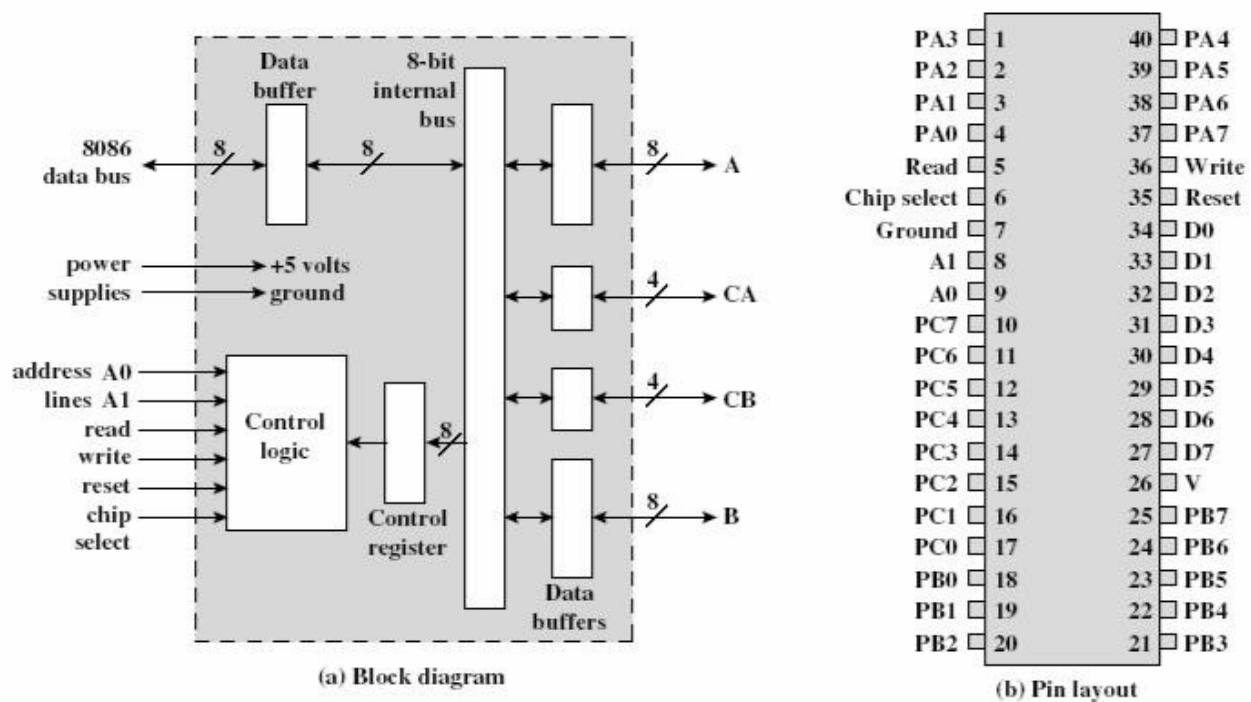
82C59A interrupt modes

Fully nested – priority form 0 (IR0) to 7 (IR7)
Rotating – several devices same priority - most recently device lowest priority
Special mask – processor can inhibit interrupts from selected devices.

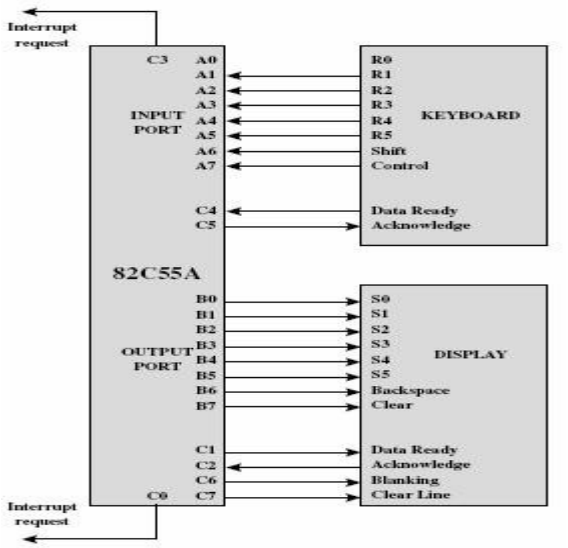


Intel 82C55A Programmable Peripheral Interface

- ¾ Single chip, general purpose I/O module
- ¾ Designed for use with the Intel 80386
- ¾ Can control a variety of simple peripheral devices



A, B, C function as 8 bit I/O ports (C can be divided into two 4 bit I/O ports) Left side of diagram show the interface to the 80386 bus.



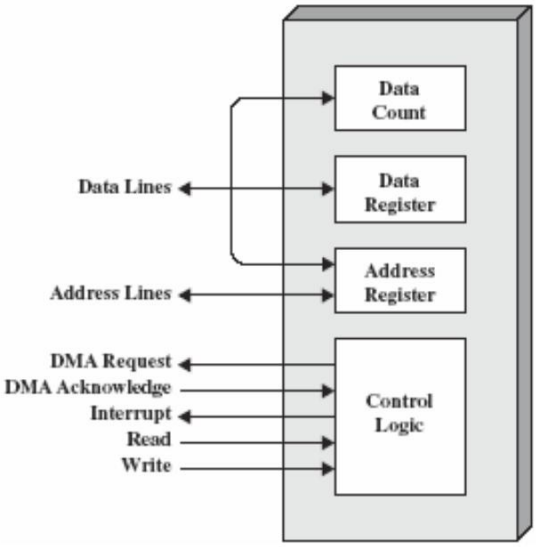
Direct Memory Access

Drawback of Programmed and Interrupt-Driven I/O

- I/O transfer rate limited to speed that processor can test and service devices
- Processor tied up managing I/O transfers

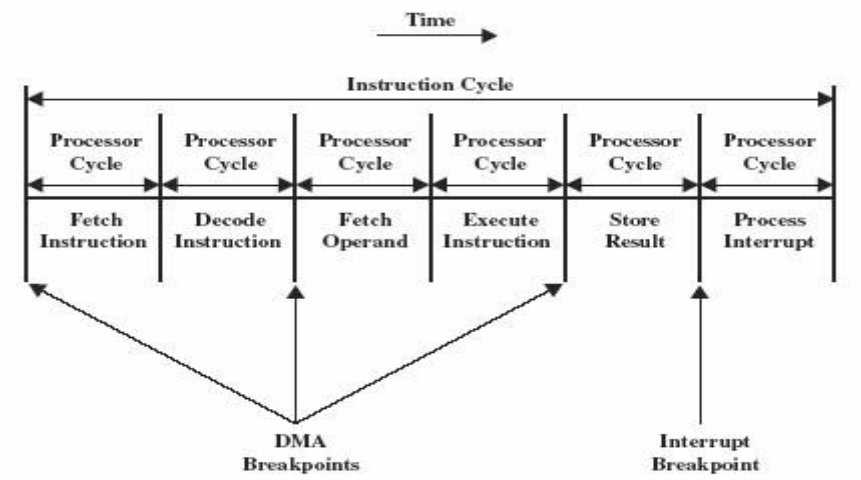
DMA Function

- DMA module on system bus used to mimic the processor.
- DMA module only uses system bus when processor does not need it.
- DMA module may temporarily force processor to suspend operations – cycle stealing.



DMA Operation

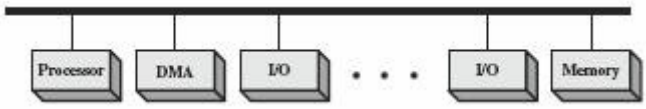
- The processor issues a command to DMA module
- Read or write
- I/O device address using data lines
- Starting memory address using data lines – stored in address register
- Number of words to be transferred using data lines – stored in data register
- The processor then continues with other work
- DMA module transfers the entire block of data – one word at a time – directly to or from memory without going through the processor DMA module sends an interrupt to the processor when complete



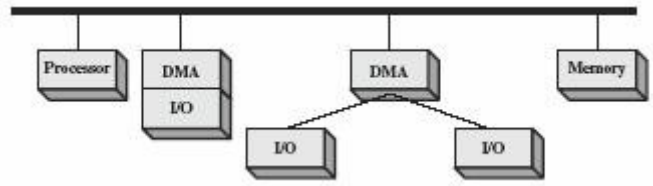
DMA and Interrupt Breakpoints during Instruction Cycle

- The processor is suspended just before it needs to use the bus.
- The DMA module transfers one word and returns control to the processor.
- Since this is not an interrupt the processor does not have to save context.
  - The processor executes more slowly, but this is still far more efficient that either programmed or interrupt-driven I/O.

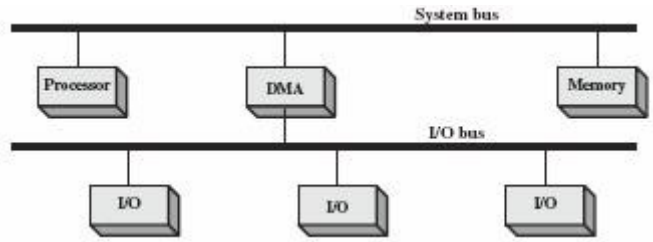
DMA Configurations



Single bus – detached DMA module
Each transfer uses bus twice – I/O to DMA, DMA to memory
Processor suspended twice.



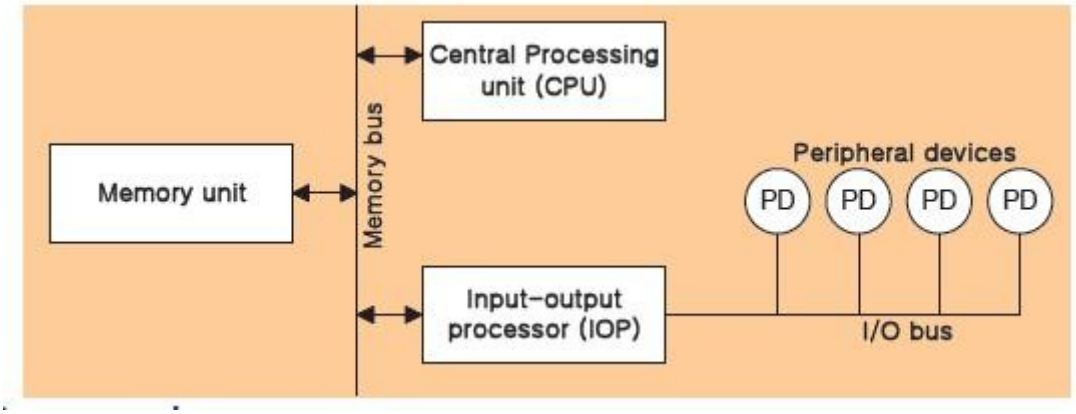
Single bus – integrated DMA module
Module may support more than one device
Each transfer uses bus once – DMA to memory
Processor suspended once.



Separate I/O bus
Bus supports all DMA enabled devices
Each transfer uses bus once – DMA to memory
Processor suspended once.

INPUT-OUTPUT PROCESSOR (IOP)

Communicate directly with all I/O devices
Fetch and execute its own instruction
IOP instructions are specifically designed to facilitate I/O transfer



Command

Instruction *that are read form memory by an IOP*

Distinguish from instructions that are read by the CPU

Commands are prepared by experienced programmers and are stored in memory

Command word = IOP program Memory

I/O Channels and Processors

The Evolution of the I/O Function

1. Processor directly controls peripheral device
2. Addition of a controller or I/O module – programmed I/O
3. Same as 2 – interrupts added
4. I/O module direct access to memory using DMA
5. I/O module enhanced to become processor like – I/O channel
6. I/O module has local memory of its own – computer like – I/O processor
   - More and more the I/O function is performed without processor involvement.
   - The processor is increasingly relieved of I/O related tasks – improved performance.

Characteristics of I/O Channels

- Extension of the DMA concept
- Ability to execute I/O instructions – special-purpose processor on I/O channel – complete control over I/O operations
- Processor does not execute I/O instructions itself – processor initiates I/O transfer by instructing the I/O channel to execute a program in memory
- Program specifies

  Device or devices

  Area or areas of memory

  Priority

  Error condition actions

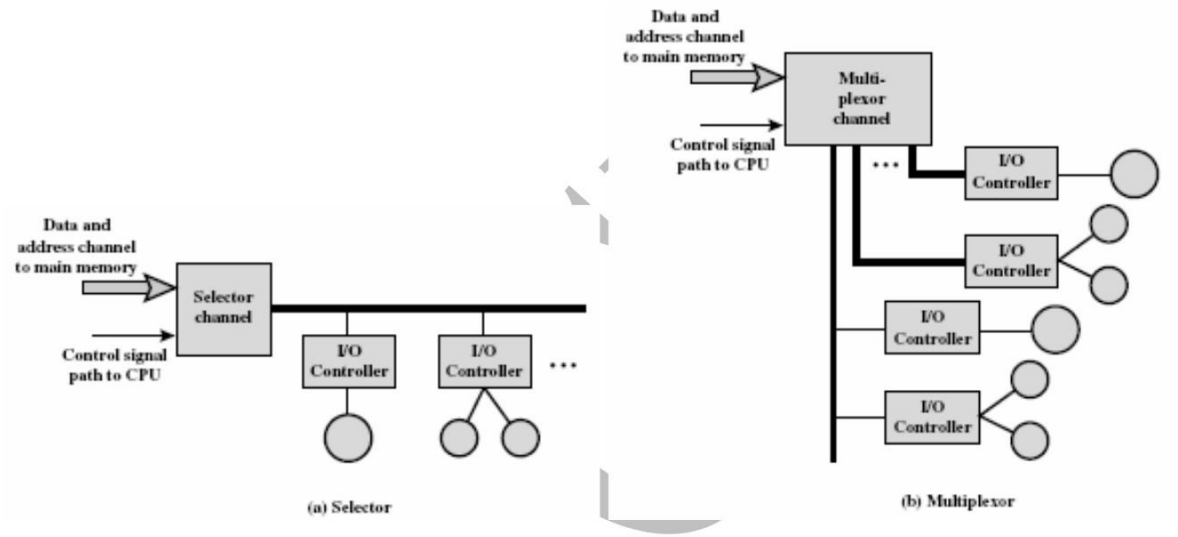
Two type of I/O channels

- Selector channel
  - ¾ Controls multiple high-speed devices
  - ¾ Dedicated to the transfer of data with one of the devices
  - ¾ Each device handled by a controller, or I/O module
  - ¾ I/O channel controls these I/O controllers
- Multiplexor channel

  Can handle multiple devices at the same time

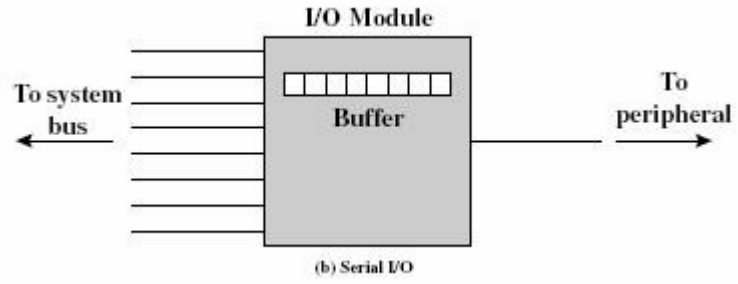
  Byte multiplexor – used for low-speed devices

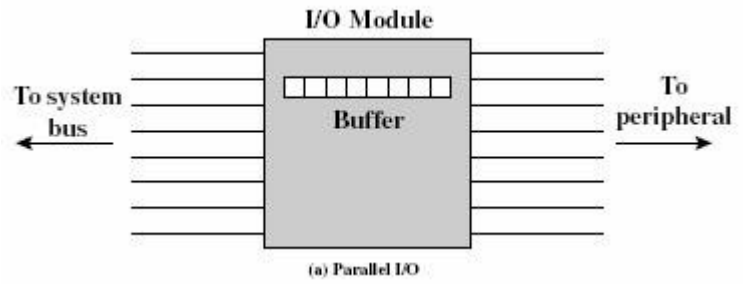
  Block multiplexor – interleaves blocks of data from several devices.



The External Interface: FireWire and Infiniband Type of Interfaces

- Parallel interface – multiple bits transferred simultaneously
- Serial interface – bits transferred one at a time





I/O module dialog for a write operation

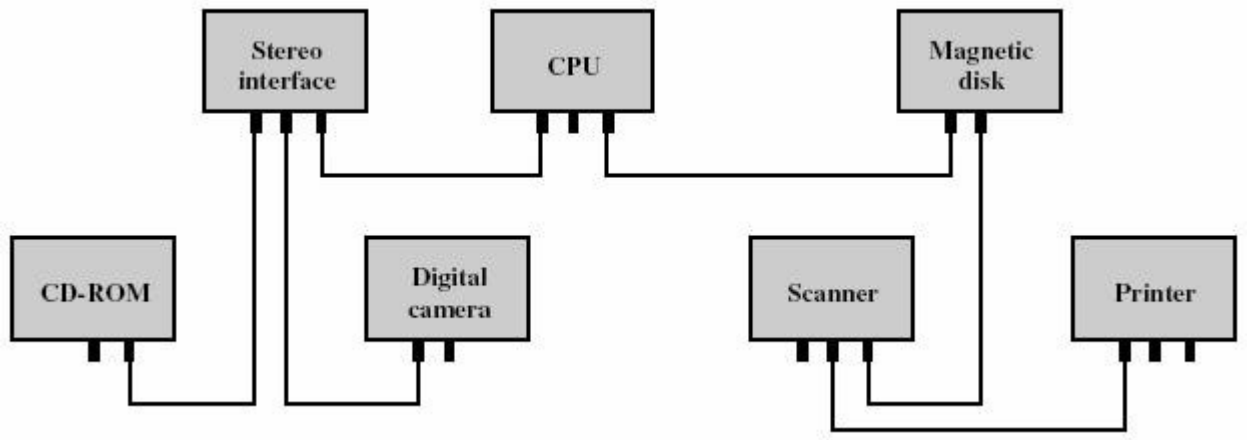
1. I/O module sends control signal – requesting permission to send data
2. Peripheral acknowledges the request
3. I/O module transfer data
4. Peripheral acknowledges receipt of data

FireWire Serial Bus – IEEE 1394

- Very high speed serial bus
- Low cost
- Easy to implement
- Used with digital cameras, VCRs, and televisions

FireWire Configurations

- Daisy chain
- 63 devices on a single port – 64 if you count the interface itself
- 1022 FireWire busses can be interconnected using bridges
- Hot plugging
- Automatic configuration
- No terminations
- Can be tree structured rather than strictly daisy chained



FireWire three layer stack:Physical layer

Defines the transmission media that are permissible and the electrical and signaling characteristics of each
25 to 400 Mbps

Converts binary data to electrical signals

Provides arbitration services

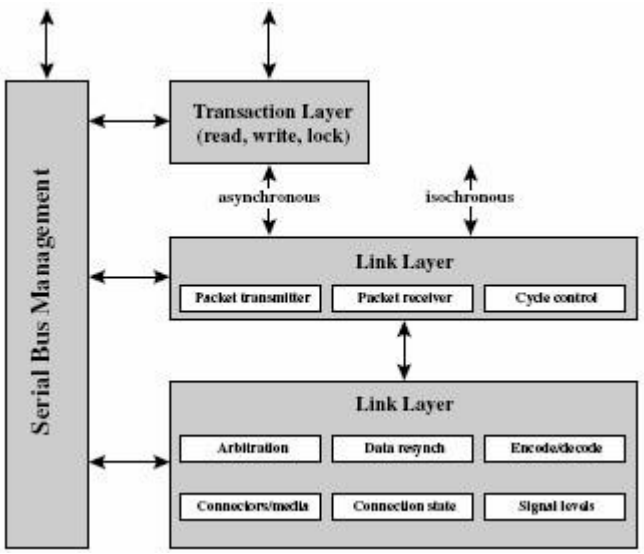
- ¾ Based on tree structure
- ¾ Root acts as arbiter
- ¾ First come first served
- ¾ Natural priority controls simultaneous requests – nearest root
- ¾ Fair arbitration
- ¾ Urgent arbitration

Link layer

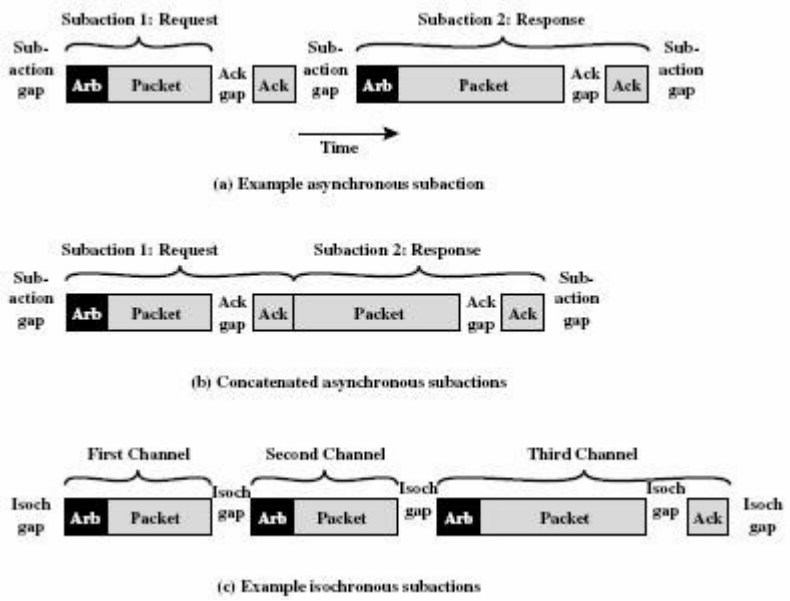
- Describes the transmission of data in the packets
- Asynchronous
  - o Variable amount of data and several bytes of transaction data transferred as a packet
  - o Uses an explicit address
  - o Acknowledgement returned
- Isochronous
  - o Variable amount of data in sequence of fixed sized packets at regular intervals
  - o Uses simplified addressing
  - o No acknowledgement

Transaction layer

- Defines a request-response protocol that hides the lower-layer detail of FireWire from applications.



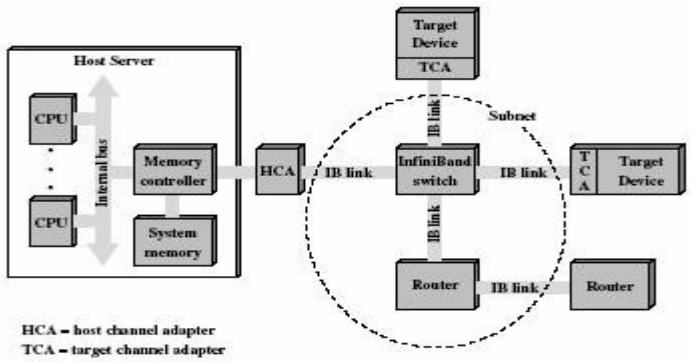
FireWire Protocol Stack



FireWire Subactions

InfiniBand

- Recent I/O specification aimed at high-end server market
- First version released early 2001
- Standard for data flow between processors and intelligent I/O devices
- Intended to replace PCI bus in servers
- Greater capacity, increased expandability, enhanced flexibility
- Connect servers, remote storage, network devices to central fabric of switches and links
- Greater server density
- Independent nodes added as required
- I/O distance from server up to
  o 17 meters using copper
  o 300 meters using multimode optical fiber
  o 10 kilometers using single-mode optical fiber
  • Transmission rates up to 30 Gbps



InfiniBand Operations

- 16 logical channels (virtual lanes) per physical link
- One lane for fabric management – all other lanes for data transport
- Data sent as a stream of packets
- Virtual lane temporarily dedicated to the transfer from one end node to another
- Switch maps traffic from incoming lane to outgoing lane