Computer Architecture معمارية الحاسبة

Computer Architecture: is the design of computers including their instruction sets, hardware components and system organization.

There are two essential parts of computer architecture:
1-Instruction Set Architecture (ISA)
2-Hardware System Architecture (HSA)

Instruction Set Architecture (ISA): Includes the specifications that determine how machine language programmers will interact with computer.

A computer is generally viewed in terms of its ISA, which determines the computational characteristics of the computer.

Hardware System Architecture (HSA): Deals with the computer's major hardware subsystems, including its central processing unit (CPU), its storage system and its input-output system (I/O) (which is the computer's interface to the world). The HSA includes both the logical design and the data flow organization of these subsystems, HSA determines how efficiently the machine will operate.

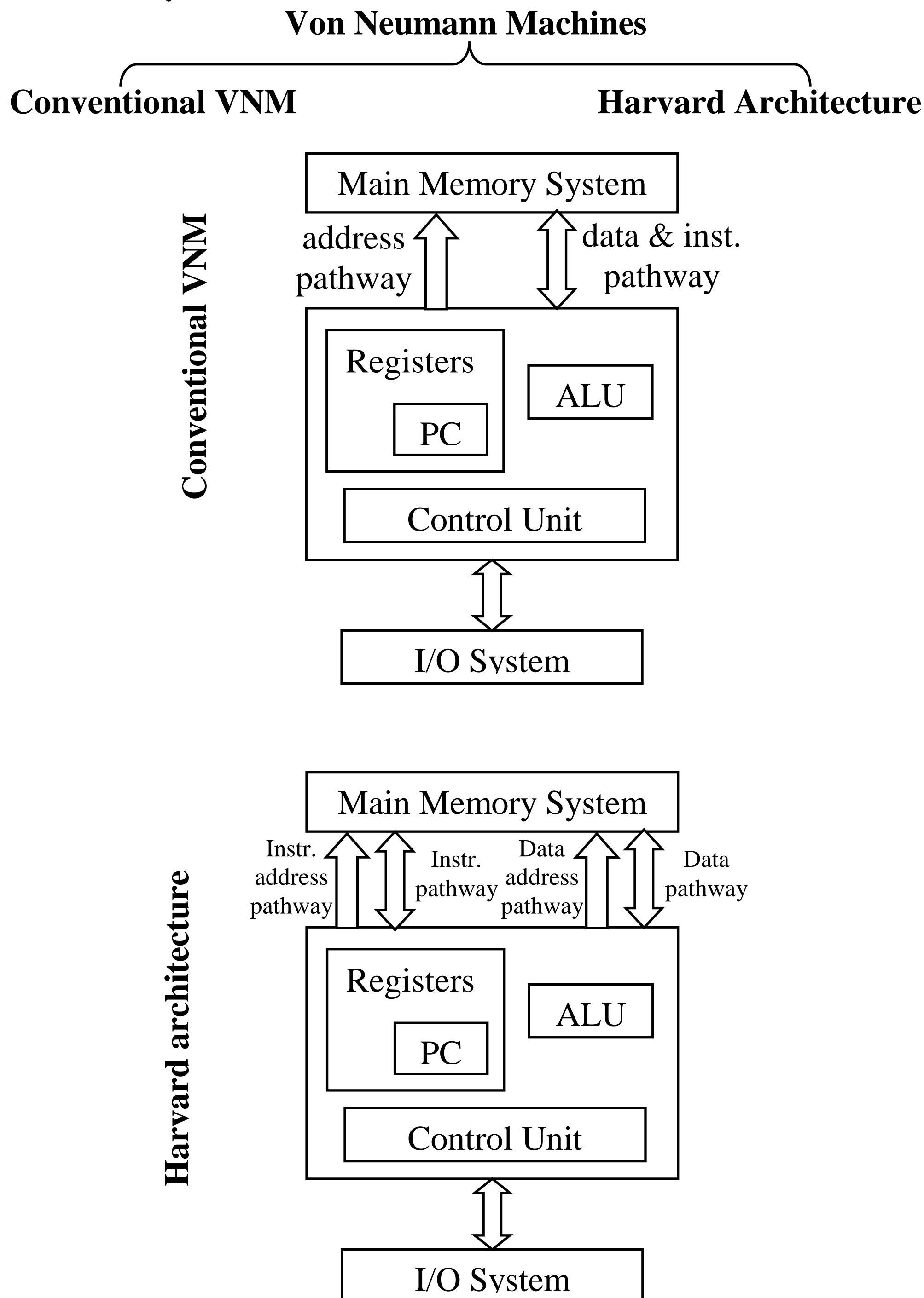
A Classification of Computer Architecture:

1) Von Neumann Machines:

Von Neumann Machines meet the following criteria:

- It has three basic hardware subsystems:
  - CPU
  - Main Memory System
  - I/O System
- It is a stored-program computer. The main memory system holds the program that controls the computer's operation and the computer can manipulate its own program more or less as it can any other data in memory.
- It carries out instruction sequentially. The CPU executes or at least appears to execute one program at a time.
- It has or at least appears to have a single path between the main memory system and the control unit of the CPU.

- Conventional Von Neumann Machines provide one pathway for addresses and a second pathway for data & instruction.
- Harvard Architecture: is a class of VNM similar to conventional computers except that they provide independent pathways for data addresses, data, instruction addresses and instructions. Harvard architectures allow the CPU to access instruction and data simultaneously.



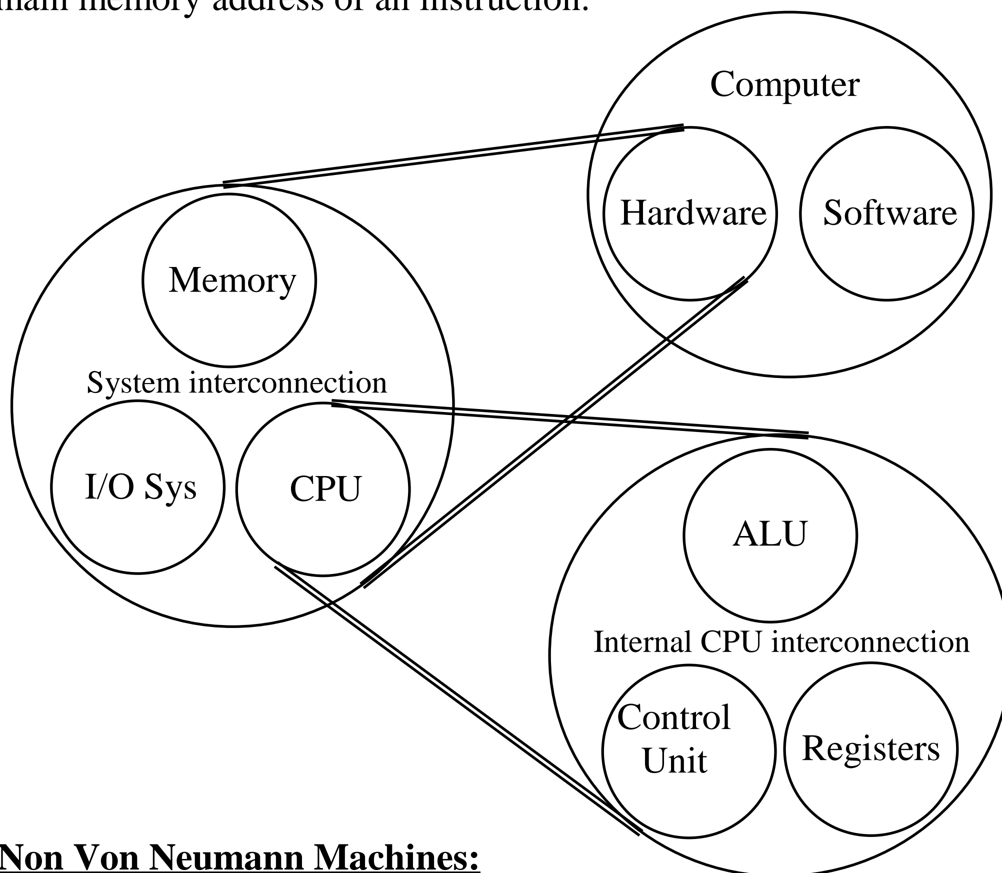
The main parts of the CPU:

1-Control Unit: which controls the operation of the computer.

2-Arithmetic & Logic Unit (ALU): which performs arithmetic, logical and shift operations to produce results.

3-Register Set: which holds various values during the computer's operations.

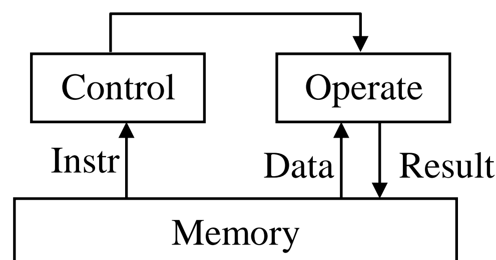
4-Program Counter (PC) (Instruction Pointer IP): which holds the main memory address of an instruction.



2) Non Von Neumann Machines:

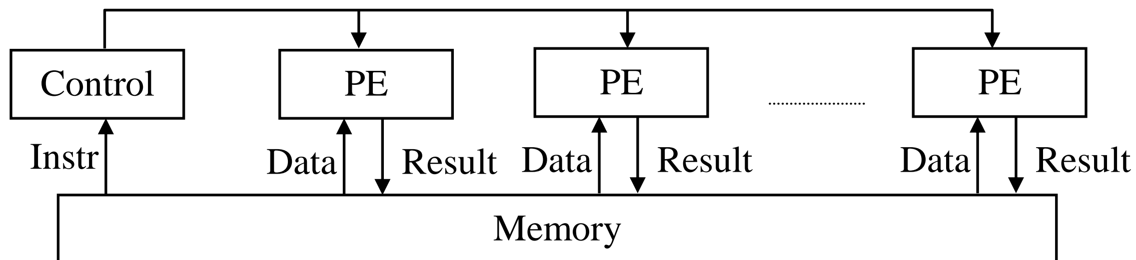
A) Single Instruction stream, Single Data stream (SISD)

The Von Neumann architecture belong to this classification. SISD computers have one CPU that execute one instruction at a time & fetch or stores one item of data at a time.



B) Single Instruction stream, Multiple Data stream (SIMD)

SIMD machine have a CU that operates like a VNM (i.e. it executes a single instruction stream). But have more than one PE (Processor Element). The CU generates the control signals for all of the PEs, which execute the same operation on different data items.

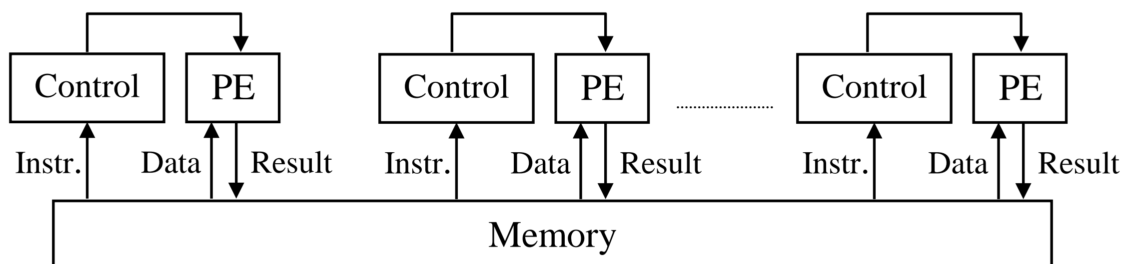


C) Multiple Instruction stream, Single Data stream (MISD)

Logically machines in this class would execute several different programs on the same data item. There are currently no such machines.

D) Multiple Instruction stream, Multiple Data stream (MIMD)

MIMD machine also called Multiprocessors. They are more than one independent processor, and each processor can execute a different program on its own data.



Memory System Architecture

The memory of a computer can be divided into three main groups:

1-Internal processor memory: This represent a small set of high-speed registers used as working memory for temporary storage of instructions and data.

2-Main memory (Also called Primary memory): This is a relatively large fast memory used for program and data storage during computer operation. It is characterized by the fact that locations in main memory can be accessed directly and rapidly by the CPU instruction set.

3-Secondary memory (Also called Auxiliary or Backing memory):
This is generally much large in capacity but also much slower than main memory. It is used for storing system programs and large data files. This type of memory has the following groups:

a- Magnetic tape b- Floppy disk
c- Hard disk d- CD-Rom (Compact Disk ROM)

Memory Device Characteristics:

Cost: Let C be the price in dollars of a complete memory system with S bits of storage capacity. We define the cost c of the memory as follows:

$$c = \frac{C}{S} \text{ dollars/ bit}$$

Access time: The performance of a memory device is primarily determined by the rate at which information can be read from or written into the memory. A convenient performance measure is the average time required to read a fixed amount of information, e.g. one word from the memory. This is termed the read access time, or more commonly, the access time of the memory and is denoted by t_A. (The write access time is defined similarity, it is typically, but not always, equal to the read access time). Access time depends on the physical characteristics of the storage medium, and also on the type of access mechanism used. Access time usually calculated from the time a read request is received by the memory unit to the time at which all the requested information has been made available at the memory output terminals.

Read Only Memory (ROM)

1-ROM
2-PROM (Programmable ROM)
3-EPROM (Erasable PROM)
4-EEPROM (Electrically EPROM)

1-ROM: Read Only Memory is a non volatile device that the CPU can read but cannot write. Computers use them for holding constants that specify the system's configuration. Many ROMs are factory programmed, and there is no way to alter their contents.

2-PROM: Field engineers can program this type of ROM memory by using special high-current device to destroy (burn) fuses that were manufactured into the devices. The result of burning a PROM is that certain bits are always 0s and the rest are always 1s. These values cannot be altered once written.

3-EPROM: This type of ROM can be erased by ultraviolet light and reprogrammed many times. The components in the memory matrix of the EPROM complex electronic devices, these devices act like diodes that can be turned on or off by the presence or absence of minute amounts of electrical charge.

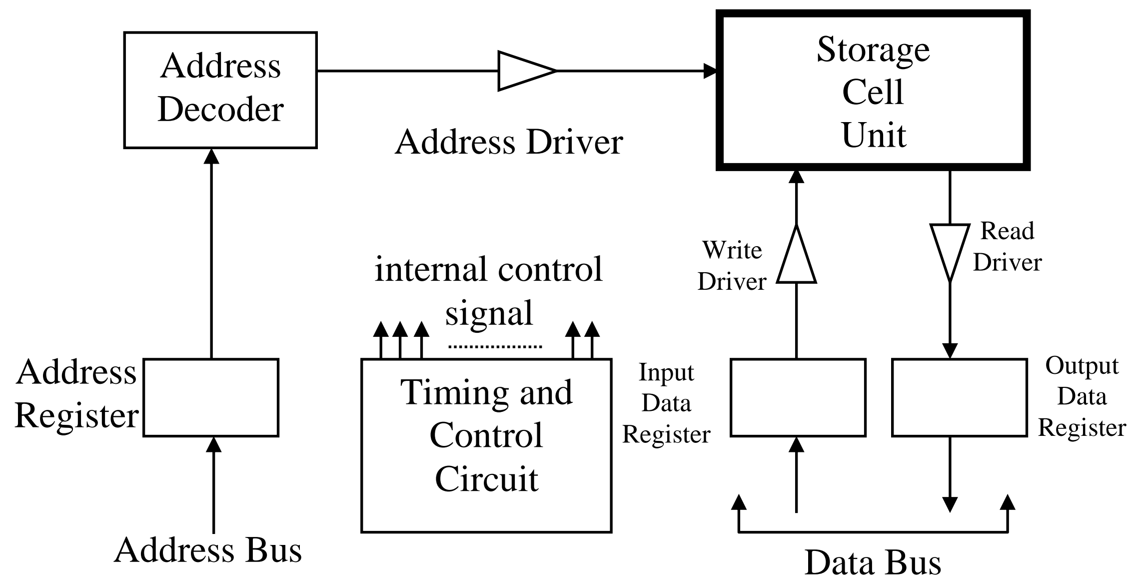
4-EEPROM: It uses components that are some what similar to those in the EPROM. However, the components in the EEPROM can be disconnected (thus erasing the memory) electrically rather than by exposure to ultraviolet light.

Random Access Memories (RAM)

RAMs are characterized by the fact that every location can be accessed independently. The access and cycle times for every location are constant and independent of its position. RAM is a memory device that the CPU can read and write. Both the reading and writing are accomplished through the use of electrical signals. RAM is a volatile which mean that it lose their information content whenever the power to the system is turned off. Thus RAM can be used only as temporary storage.

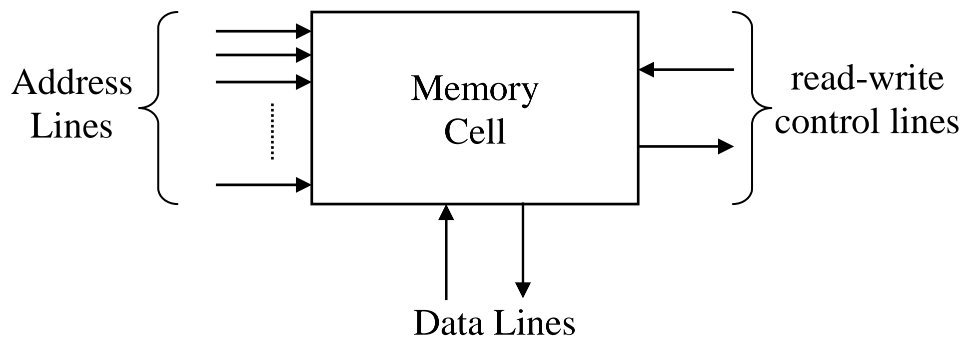
The figure below shows the main components of a RAM unit. The storage cell unit comprises N cells, each of which can store 1 bit of information.

The memory operates as follow: The address of the required location is transferred via the address bus to the memory address register, the address is then processed by the address decoder which select the required location in the storage cell unit. A read-write select control line specifies the type of access to be performed. If read is requested, the contents of the selected location is transferred to the output data register. If write is requested, the word to be written is first placed in the memory input data register and then transferred to the selected cell.



Random Access Memory (Main Components of a RAM Unit)

The various drivers, decoder and control circuit are collectively referred to as the access circuitry of the memory unit.



General Model of a RAM Cell

RAM Organization

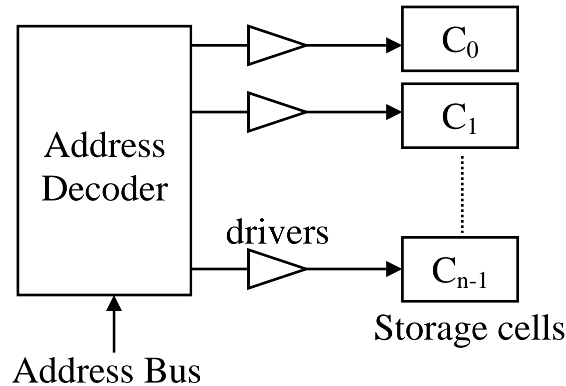
The access circuitry needed has a very significant effect on the total cost of any memory unit. RAM is called matrix or array organization. It has two essential features:

1-The storage cells are physically arranged as rectangular arrays of cells. This is primarily to facilitate layout of the connections between the cells and the access circuitry.

2-The memory address is partitioned into d components, so that the address A_i of cell C_i becomes a d-dimensional vector $(A_{i,1}, A_{i,2}, \ldots, A_{i,d}) = A_i$. Each of the d parts of an address word goes to a different address decoder and a different set of address drivers. A

particular cell is selected by simultaneously activating all d of its address lines. A memory unit with this kind of addressing is said to be a d-dimensional memory.

If d=1, called one-dimensional or 1-D memories. If the storage capacity of the unit is N bits, the access circuitry typically contains one-out-of N address decoder and N address drivers.



(One-dimensional addressing scheme)

e.g.: d=1, N=16, find number of drivers, address decoder, and number of bits of address bus?

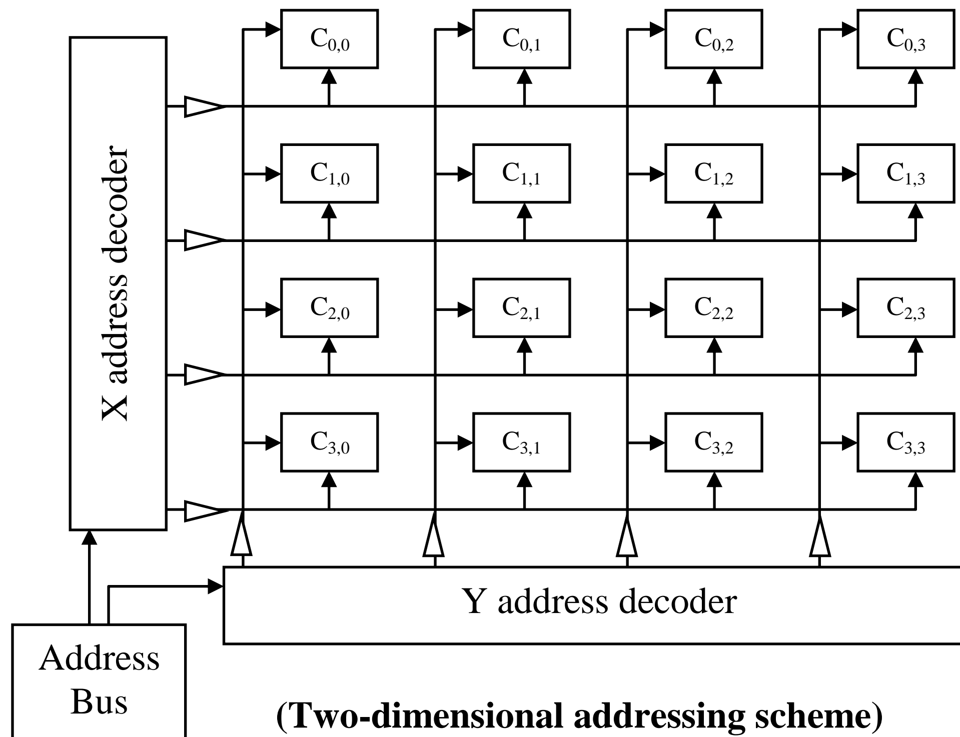
N = 16,
no of drivers = 16.
address decoder = 0ne-out-of 16.
$N = 16 = 2^4$,
no of bits in the address bus = 4.

If d=2, called two-dimensional (2-D) organization. The address field is divided into two components called X and Y, which consist of a_x and a_y bits respectively. The cells are arranged in a rectangular array of $N_x \le 2^{a_x}$ rows and $N_y \le 2^{a_y}$ columns, so that the total no of cells is $N = N_x \times N_y$. The 2-D organization requires less access circuitry than 1-D for a fixed amount of storage.

If $N_x = N_y = \sqrt{N}$ then,

The no of address drivers $= 2\sqrt{N}$.

and two one-out-of $\sqrt{N}$ address decoders.



(Two-dimensional addressing scheme)

e.g.: if d=2, the address bus consist of 4 bits, find the type and number of address decoders, no of rows, no of columns, no of drivers, no of total cells?

since d = 2, address bus = 4 bits

$\therefore$ X = 2 bits, Y = 2 bits

$\therefore$ we need 2 one-out-of 4 decoders (1 for X, 1 for Y)

$\therefore$ no of rows $= 2^{a_X} = 2^2 = 4$

$\therefore$ no of columns $= 2^{a_Y} = 2^2 = 4$

$\therefore$ $N = N_X * N_Y = 4*4 = 16$ cells.

Semiconductor RAMs

Semiconductor memories fall into two main categories, static and dynamic.

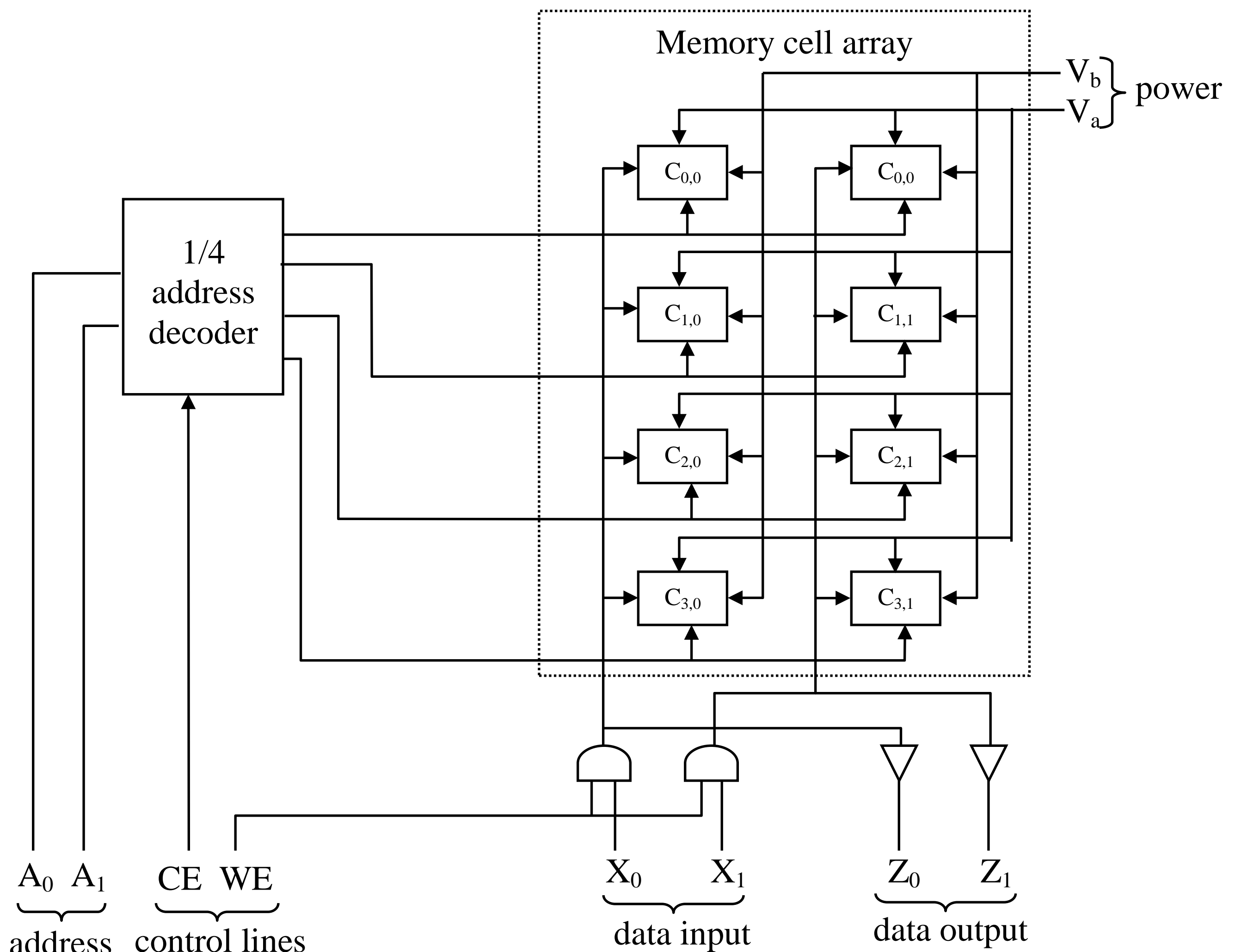
Static RAMs (SRAM):

These devices are composed of flip-flops that use a small current to maintain their logic level. The contents of SRAM memory remain unchanged for an indefinite period of time as long as the power is on. SRAMs are used mostly for the CPU registers and other high speed devices, although some computers use them for caches and main memory. SRAMs are currently the fastest and most expensive of the semiconductor memory circuits.

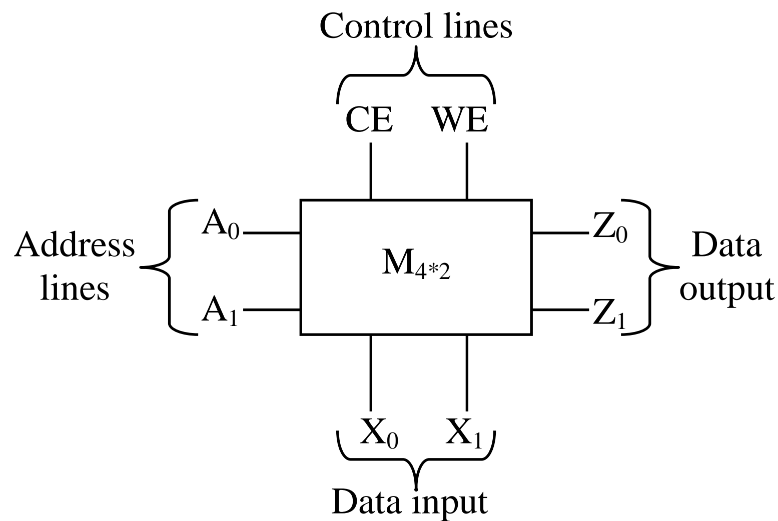
Dynamic RAMs (DRAM):

These devices are made with cells that store data as charge of capacitors (a device for holding an electrical charge) together with a single transistor. This pair of devices is smaller than the two or more gates required for each flip-flop in an SRAM. The presence of a positive charge on the capacitor (a positive voltage) can be interpreted as a one, and its absence (zero voltage) as a zero. Unfortunately, the capacitors slowly lose their charges due to leakage. So periodic charge refreshing is necessary to maintain data storage. The refresh circuit must refresh the charges about every 2ms.

A semiconductor RAM IC typically has a word organized array structure and contains all required access circuitry including address decoders, drivers and control circuits. The figure below shows a simple 4*2-bits RAM:



Structure of a 4*2-bit RAM



(Symbol for 4*2-bit RAM)

RAM design:

A memory design problem that the computer architect may encounter is the following: giving that certain m*n-bit RAM Ics denoted M_{m*n} are available, design an m'*n'-bit RAM, where $m' \geq m$ and/or $n' \geq n$. a general approach is to construct a p*q array of the M_{m*n} modules where p=m'/m and q=n'/n, when $m' \geq m$, additional external address decoding circuitry may be required.

Ex: Design a 16*4-bit memory using 4*2-bit Ics?

sol:

$M_{m*n} = M_{4*2}$

$p = m'/m = 16/4 = 4$ rows

$q = n'/n = 4/2 = 2$ columns

$m' = 16 = 2^4$, $\therefore$ no of address lines = 4, (A_0, A_1, A_2, A_3)

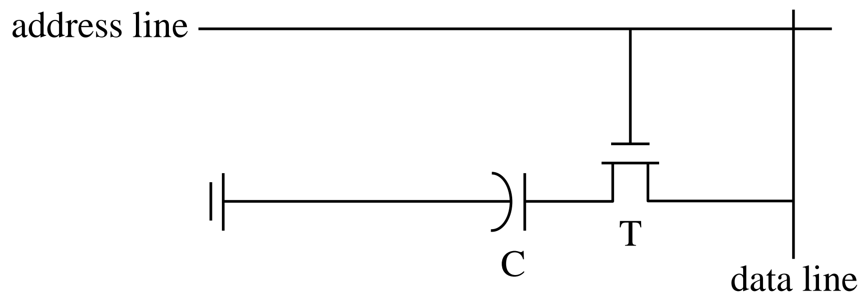
$m = 4 = 2^2$, (A_0, A_1)

$m' = 16 = 2^4$, ($\underline{A_0, A_1}$, A_2, A_3)

(A 16*4-bit RAM)

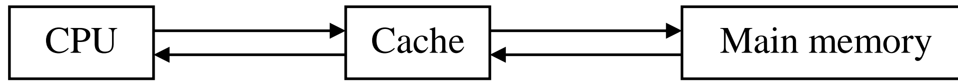
<u>Example</u>: The Intel 2186 64k-bit dynamic RAM

This commercial RAM chip, which was introduced in 1983, contains 64k one-transistor MOS (Metal Oxide Semiconductor) storage cells of the kind shown figure below:



Cache memory:

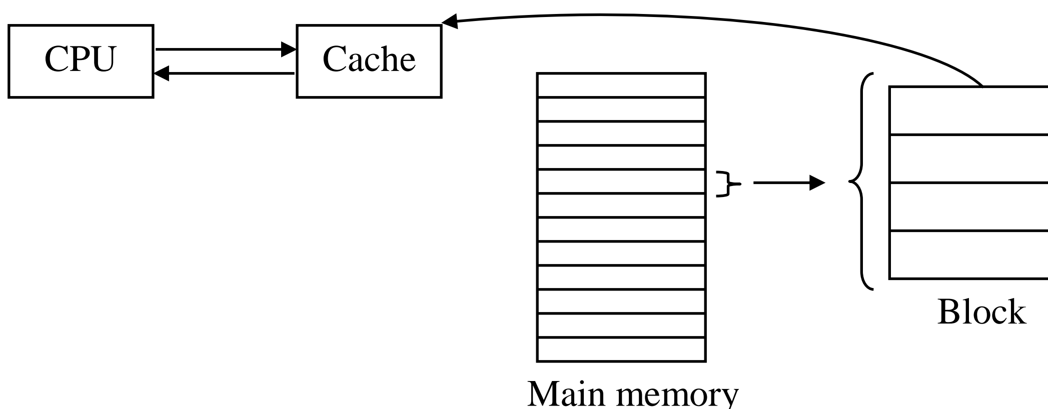
A cache is a small fast memory placed between a processor and main memory as illustrated in figure:



The cache is then the fastest component in the memory hierarchy. It can be viewed as a buffer memory for the main memory, so that the cache M_1 and main memory M_2 form a two-level hierarchy. Caches are used in various forms to reduce the effective time required by a processor to access address, instruction or data that are normally stored in main memory. The term cache is usually reserved for a general-purpose buffer memory designed to store instruction or data associated with the execution of all types of program.

Sometimes a cache is used to store instruction but not data, in which case the term instruction cache or instruction lookaside buffer are used. The advantage of restricting a cache to instruction is that, unlike data, instructions do not change, so the contents of an instruction cache need never be written back to main memory.

When a CPU demands a specific information (e.g. word), the CPU first checks the cache for the existence of this word. If not, this means the word exist in main memory, therefore, the main memory will transfer this word to cache with the block of information nearest to this word.



Cache design:

The performance goal of adding a cache memory to a computer is to make the average memory access time t_A seen by the processor as close as possible to that of the cache t_{A1}. to achieve this, a high percentage of all memory references should be satisfied by the cache, i.e.: the cache hit ratio should be close to 1. This is possible because of the locality-of-reference property of program.

Hit: means the information are in cache.

Miss: means the information are in main.

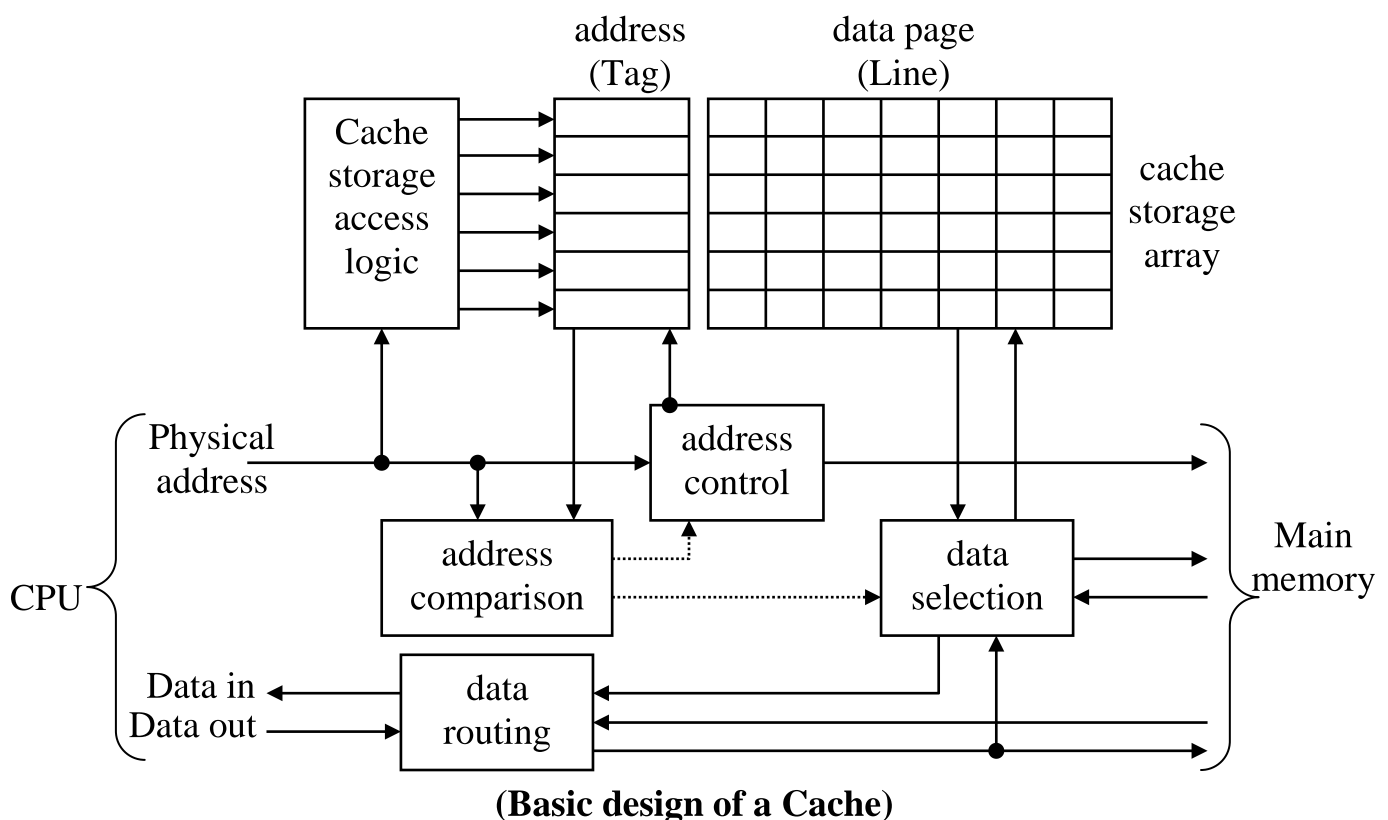
$$\text{Hit ratio} = \frac{\text{no. of Hits}}{\text{no. of Hits} + \text{no. of Miss}} \geq 0.9$$

Principle of locality-of-reference:

Over any short period of execution time, the addresses and data that the program need, are referenced in a specific area of main memory, but the other area are discarded not demand.

Loops, subprograms, subroutines and arrays are exact example of locality-of-references. This principle increases the hit ratio. But JMP and GOTO instructions decreases hit ratio because of the continuously block replacement.

The structure of cache memory:

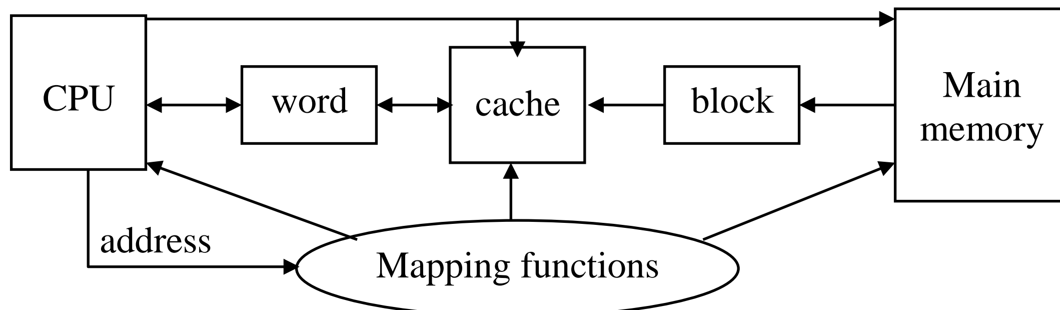


(Basic design of a Cache)

It stores a set of main memory addresses A_i and the corresponding data (words) $M(A_i)$. The data entries are grouped into blocks, cache pages, or called "lines". Each of which is a subblock of some main memory page, the corresponding stored address is therefore a block address. The contents of the cache array are thus copies of a set of small non-contiguous main memory blocks tagged with address.

The basic operation of cache:

A physical address A is sent to the cache from the CPU at the start of a read (load) or write (store) memory access cycle. The cache compares the relevant part of A (address tag) to all the addresses it currently stores. If there is a match i.e.: a cache hit, then the cache selects the desired word M(A) from the data entry corresponding to A. It completes the memory cycle by transferring data from the CPU to its copy of M(A) (write operation) or else retrieving its copy of M(A) and routing it to the CPU (read operation). If A fails to match any of the stored addresses, i.e.: a cache miss, then the cache usually initiates a sequence of one or more main memory read cycles to copy into the cache. The main memory block P(A) containing the desired item M(A).



Performance of cache memory:

1) $$C_s = \frac{C_c S_c + C_m S_m}{S_c + S_m}$$

C_s : average cost per byte of system (main + cache)
C_c : average cost per byte of cache.
C_m : average cost per byte of main.
S_c : size of cache.
S_m : size of main.

2) $$T_s = H * T_c + (1 - H)\,T_m$$

T_s : system access time.
T_c : cache access time.
T_m : main access time.
H : hit ratio.
1-H : miss ratio.

Ex: A computer with cache access time equal to 100ns, a main memory access time equal to 1000ns, and the hit ratio equal to 0.9. Find the average access time of memory?

Sol:

$$T_s = H * T_c + (1 - H)\, T_m$$
$$= 0.9 * 100 + 0.1 * 1000$$
$$= 90 + 100$$
$$= 190 \text{ ns.}$$

Mapping Functions:

قوانين مهمة (مثال يستخدم لشرح الـ mapping)

- $\text{no. of blocks in main} = \dfrac{\text{size of main}}{\text{size of block}}$
- $\text{no. of slots (lines) in cache} = \dfrac{\text{size of cache}}{\text{size of block}}$

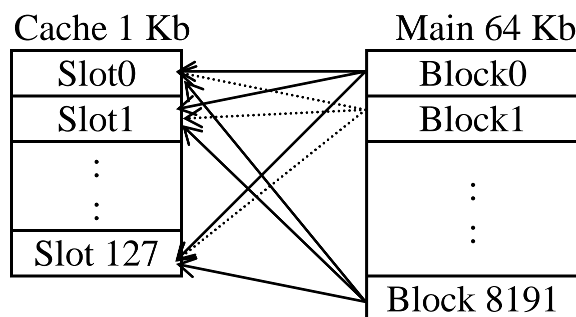
Ex: Suppose cache size = 1 Kbyte, data is to be transferred between main and cache in block of 8 bytes, main memory size = 64 Kbyte. Find no. of blocks in main, no. of slots in cache?

Sol:

- $\text{no. of blocks in main} = \dfrac{S_{main}}{S_{block}} = \dfrac{64\text{ kbyte}}{8\text{ byte}} = \dfrac{2^{10} \times 2^6}{2^3} = 2^{13} = 8\text{ Kblocks}$
- $\text{no. of slots in cache} = \dfrac{S_{cache}}{S_{block}} = \dfrac{1\text{ kbyte}}{8\text{ byte}} = \dfrac{2^{10}}{2^3} = 2^7 = 128\text{ slots}$

1 >> Associative Mapping:

A main memory block can be loaded (mapped) in any slot in the cache. Therefore, block0 can be mapped in slot0 or slot1 or any other slot in the cache. According to the example:



Main memory address = | 13 | 3 |,

tag (block addr. in main) word

The main memory address in this method will be in two parts, and according to the example, main memory address consists of 16 bits (64 Kb), and the block (8 bytes) can represent it by 3 bits, therefore the tag will be represented by 13 bits.

The CPU sends an address for a word to the cache, the tag part of main memory address will be compared to all tags in cache (no specified slot for specified block), if there is a match, then the word part will be picked up from the words part in block. If there is no match, then by the tag part the CPU will fetch the specified block from the main memory.

2 >> Direct Mapping:

Allows each block of main memory only one possible cache slot by using:

$$S = A \text{ modulo } C$$

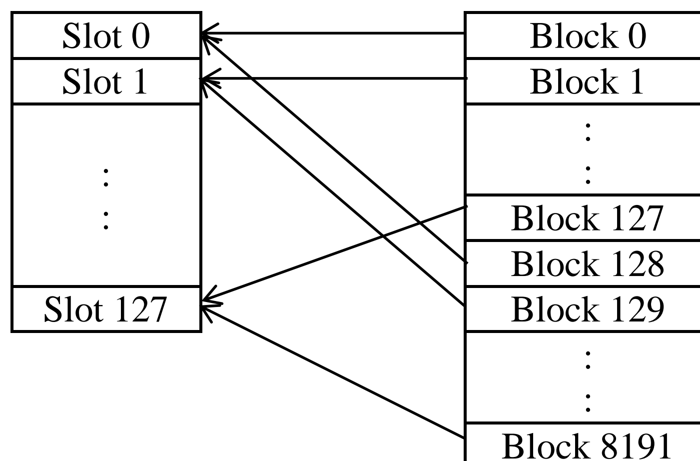
where : S = cache Slot no.
A = main memory Address (main memory no).
C = total no of slots in Cache.

According to the example:

Block 0 → S = 0 modulo 128 → S = 0
Block 1 → S = 1 modulo 128 → S = 1
Block 127 → S = 127 modulo 128 → S = 127
:
Block 128 → S = 128 modulo 128 → S = 0
Block 129 → S = 129 modulo 128 → S = 1

Therefore,

Blocks 0, 128, 256, 384, ……… mapping on slot 0
Blocks 1, 129, 257, 385, ……… mapping on slot 1
:
Blocks 127, 255, …, 8191 …… mapping on slot 127



Cache address = | tag | slot | word | ,

- The slot part will be index on cache to determine the desired slot,
  128 slots = $2^7 \rightarrow$ 7-bits address lines
- The word part determines which desired word in block,
  8 bytes = $2^3 \rightarrow$ 3-bits address lines
- The tag part preferences which block in the current slot, 7-bits for slots address lines, 3-bits for word address lines, then 6-bits for tag part, or by using this formula:

$$\frac{\text{no. of blocks that can mapped}}{\text{in each block cache}} = \frac{\text{no of block in main}}{\text{no of block in cache}} = \frac{8\text{ Kb}}{128} = \frac{2^{13}}{2^7} = 2^6$$

Cache address = | 6 | 7 | 3 | ,
tag slot word

The CPU sends 16-bits address to the cache, first compare the slot part to determine the block, second compare the tag part of the address with the tag part in cache to determine which block is in cache now, if match, this means the desired block exists (hit), then the desired word will be specified by the word part. If not match (miss), then the slot part and the tag part will merge to form the desired block address (13-bits) to get the block from main memory to cache.

3 >> Set Associative Mapping:

The cache is divided into I sets, each set consists of J slots, we have:

$C = I \times J$

$K = A \text{ modulo } I$

C : total no. of slots in cache.
I : no. of the sets in cache.
J : no. of slots in each set.
K : cache set no.
A : block address (block no. coming from main memory)

With this algorithm, the block containing address A can be mapped into any slot in set I. if I = C, J = 1, the set associative technique reduces the direct mapping, and for I =1, J = C, it reduces to associative mapping.

For example, if the no. of slots = 2 in each set, then the no. of set is: 128 / 2 = 64 sets in cache.

Any coming block from main memory will be mapped into any 1 of 2 slots in the specified set. The set will be specified by using:

$$\boxed{K = A \text{ modulo } I}$$

Block 0 $\rightarrow$ K = 0 modulo 64 $\rightarrow$ K = 0
Block 1 $\rightarrow$ K = 1 modulo 64 $\rightarrow$ K = 1
Block 63 $\rightarrow$ K = 63 modulo 64 $\rightarrow$ K = 127
Block 64 $\rightarrow$ K = 64 modulo 64 $\rightarrow$ K = 0

Therefore,
Blocks 0, 64, 128, 192, ……….. mapping on set 0
Blocks 1, 65, 129, 193, ……….. mapping on set 1
:
Blocks 63, 127, 191, …, 8191 … mapping on set 63

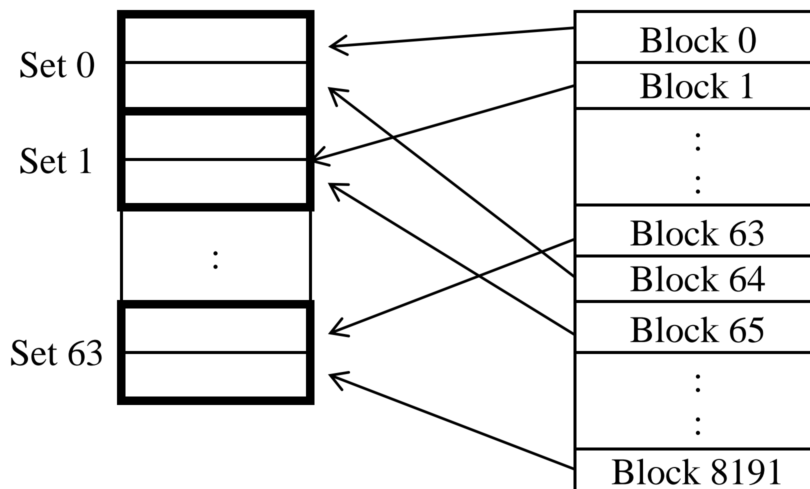
Cache address = | tag | set | word | ,

- The word part, as previous,
  8 bytes $\rightarrow$ 3-bits address lines.
- The set part calculated as:
  64 sets = 2^6 sets $\rightarrow$ 6-bits address lines.
- The tag part preferences which block in the current slot, 6-bits for the set address lines, 3-bits for word address lines, then 7-bits for tag part, or by using this formula:

$$\frac{\text{no. of blocks that can}}{\text{mapped in each set}} = \frac{\text{no of block in main}}{\text{no of set in cache}} = \frac{8\text{ Kb}}{64} = \frac{2^{13}}{2^6} = 2^7$$

Cache address = | 7 | 6 | 3 | ,
tag set word

Note: The hit ratio in the set associative mapping is larger than the direct mapping but less from the associative mapping.



Replacement Algorithm:

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

For direct mapping, there is one possible slot for any given block and no decision is needed.

For the associative & set associative mapping, a replacement algorithm is needed.

Replacement Types (Algorithms):

1) First – In – First – Out (FIFO)

   Replace that block in the set which has been in the cache longest.

2) Least Frequently Used (LFU)

   Replace that block in the set which has experienced the fewest references.

3) Least Recently Used (LRU)

   Replace that block in the set which has been in the cache longest with no reference to it.

Write Policy:

1) Write Through:

All writes operations are made to main memory as well as to the cache, ensuring that the main memory is always valid. The main disadvantage of this technique is that there is an unnecessarily high rate of memory writes.

2) Write Back (CopyBack)

Minimizes memory writes with write back, updates are made in the cache. When an update occurs, an update-bit associated with the slot is set (update-bit = 1). Then, when a block is replaced, it is written back to main memory if and only if the UPDATE-bit is set. It has the disadvantage that M1 and M2 can be inconsistent, i.e. have different data associated with the same physical address. This inconsistent found when more than one CPU work on the same main memory.

Virtual Memory:

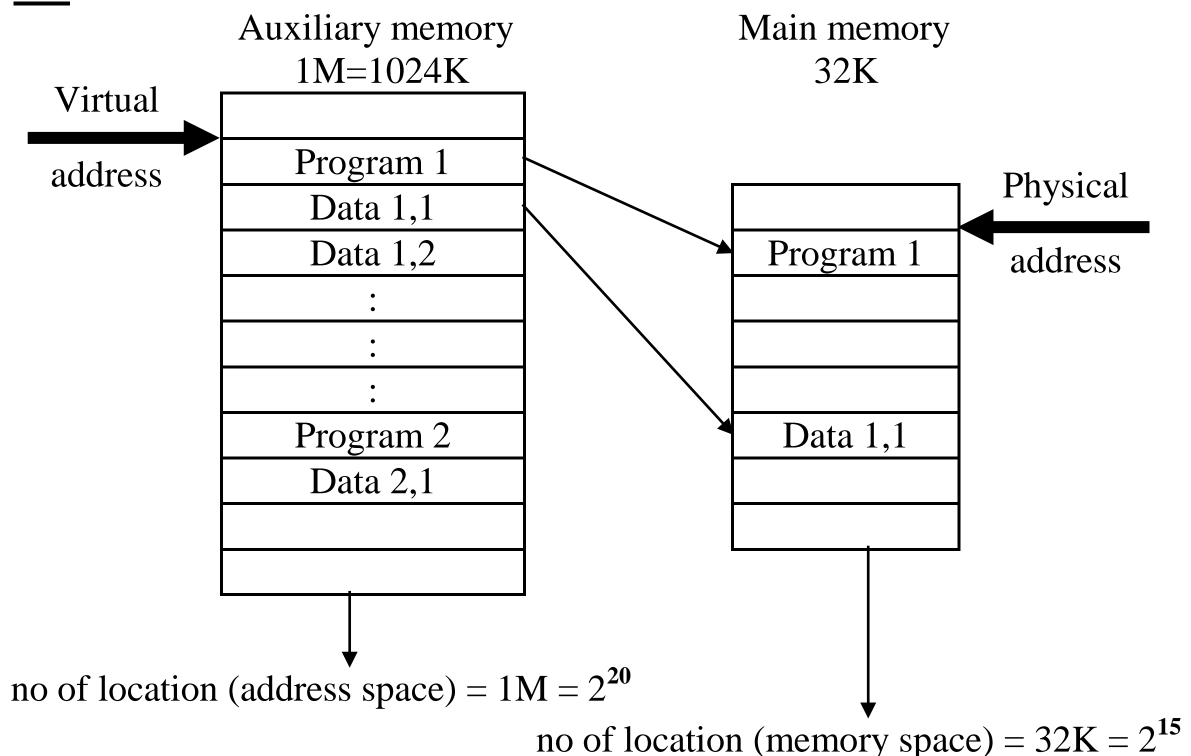
The operating system is used to produce the illusion of an extremely large memory. Since this large memory an illusion, it is called Virtual Memory. In virtual memory systems, the operating system loads only part of a program, the currently active part in main memory.

Virtual memory is used in large computer systems which allow the user to store large amount of data and programs in the secondary memory. Every address (virtual address) goes out of the CPU, passes through process steps to transform the virtual address to a physical address in the main memory.

If we have a big application program larger than the main memory size, then by using the virtual memory concept we can load this application to the main memory and then execute it. Where the active parts of the program will be loaded from the secondary memory to the main memory and will be executed by the CPU as dependent programs. This will achieved by the OS (operating system), where one of the OS responsibility is to manage the memory by controlling the memory and the data transmittance.

Virtual memory systems generally use one or both of two techniques for mapping effective addresses (CPU addresses) into physical addresses: paging and segmentation.

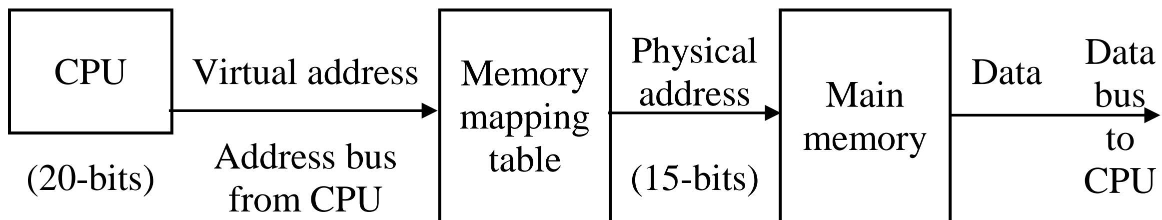
Ex:



From the figure above, we see that the secondary memory is larger 32 times than the main memory.

In this example, the CPU will issue the instruction & data addresses with length of 20-bits, but these instruction and data are currently exist in main memory (remember that the desired application and data to be executed had already transmitted from auxiliary memory to the main memory).

Therefore, we need such a table to convert the virtual addresses (with length 20-bits) to physical addresses (with length 15-bits). This process achieved dynamically, meaning that every address issued by the CPU will be converted directly and automatically to a physical address.

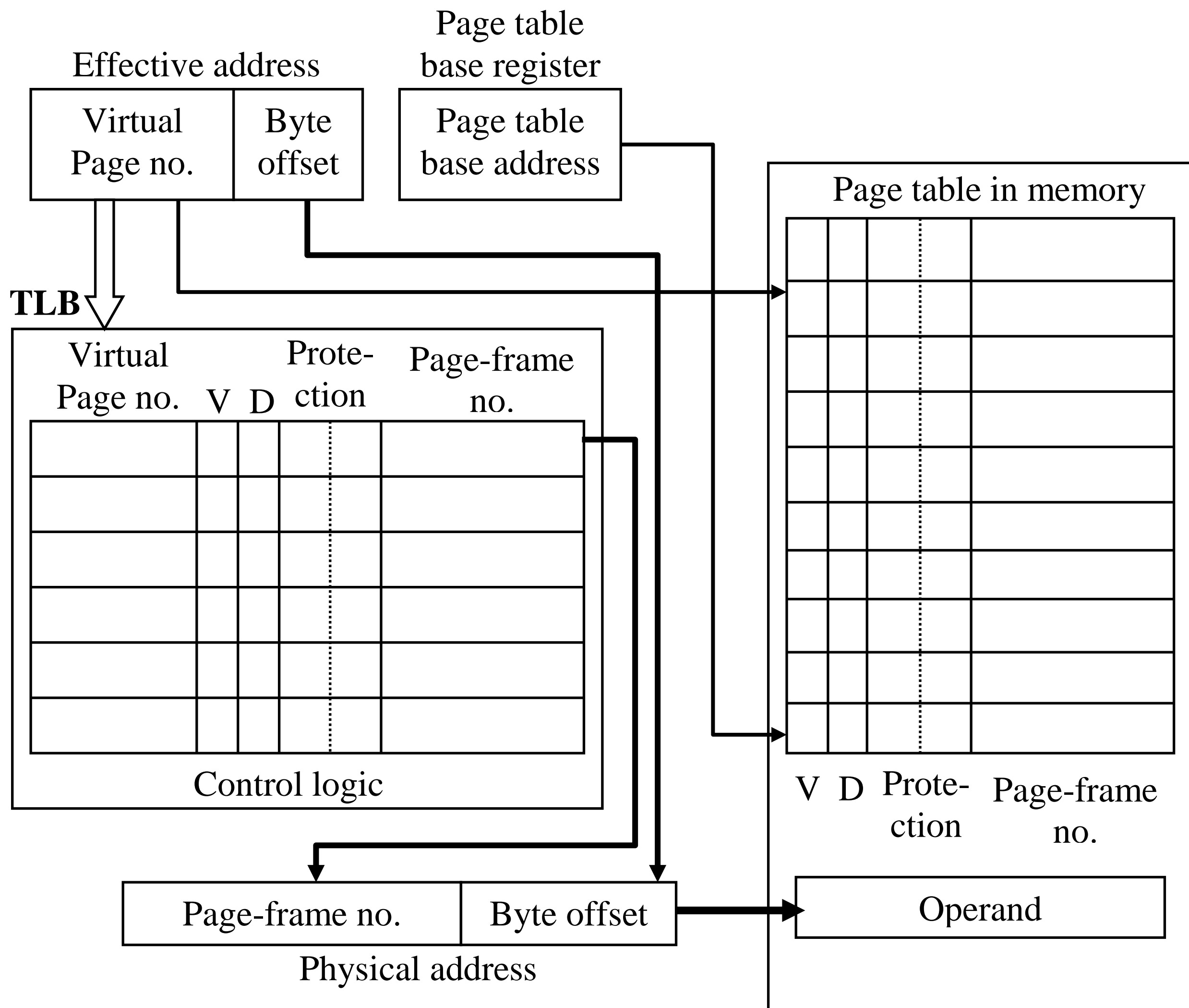


Paging:

Is a hardware-oriented technique for managing physical memory. Architects introduced paging so that large programs could run on computers with small physical memories. In essence, the computer loads into main memory only those parts of the program that it currently needs for execution. The remainder of the program resides in external storage until needed.

In paging system, the virtual memory hardware divides logical addresses into two parts: a page number and a word offset within the page. The hardware makes this division by partitioning the bits of the address to the following: the high order bits are the page number and the low order bits are the word offset.

The units of physical memory that hold pages are called page-frame (or sometimes called blocks).



(Components of a Paging System)

The page table in the main memory consist of many **entries** equal to the number of the existence pages. Each **entry** gives the specifications of a specific page, these specifications are:

1- **V**: (Valid) used to determine whether the desired page exist now in the main memory or not. If V=1 means that the page of this entry exist now in the main memory, but if V=0 means that this page is not in the main memory but it is now still exist in the second memory.

2- **D**: (Dirty) used to determine whether the page of this entry has been changed or not. If D=0 means the page has not been changed, but if D=1 means that the contents of this page has been changed, then the system will make a copy of this page to the secondary memory.

3- **Protection**: used for page protection, and consist of 2-bits, this protection include the protection for the instruction page from any change (write), and as follow:

Protection		Meaning
0	0	The page is for instruction only
0	1	The page is for read only
1	0	The page is for read and write
1	1	Nothing

4- **Page frame no.**: hold the number of the page determined by this entry in the main memory.

Note: from the figure, we see the hardware part, that is, the page table base register, which provides the beginning address of the page table in the main memory.

TLB (Translation Lookaside Buffer)

A small cache, some hardware systems maintain it as part of the page map (memory map). TLB holds essentially the same information as part of the page table. In general, a TLB holds entries only for the most recently accessed pages and only for valid pages, that is, pages that have an image in main memory (exact copies of the data).

For a paging system, whenever the CPU generates an effective address, the CPU sends it to the TLB, which produces the page-frame no. if it holds an entry for the page. If the TLB has no entry, the hardware consults the page table in main memory by using the page no. as an offset into the page table.

If the validity bit (V=1) indicates the page is in main memory, the hardware uses the page frame no. to access the memory and simultaneously copies the page table entry into the TLB. Otherwise, the hardware initiates a trap (interrupt) called a page fault, at which point the OS intervenes to load the demanded page in memory and updates the page table.

The number of entries in the page table is not fix, because it depends on the: 1-size of the applied program, 2-size of a single page. For example, if the size of the applied program is 20KB, and the size of the page is 4KB, then the number of pages used by this program is 5 pages. So, the number of the entries for this application in 5 entries, each of which contains the specifications of a single page.

The advantage of TLB is to speed up the system. If the TLB is not there, then the CPU reference to the main memory will be twice, once for searching the page entry inside the page table in the main memory, and second to access the data inside the page in the main memory after

creating the physical address. While by the existence of the TLB, the CPU will reference the main memory once to access the data there, when the demanded entry is exists in TLB.

Page Thrashing:

The state of excessively moving pages between memory and secondary storage. So the CPU spends most of it's time swapping pages rather than executing instructions.

For all types of paging systems, whenever a page fault occurs, the OS must decide in which page frame to put the demanded page. The OS will choose an empty page frame wherever possible. However, if all the page frames are occupied, the OS must delete an existing page to make room for the new page. OS use several different policies to do so.

Page Replacement Policies:

1- **FIFO**: The first page came to the main memory, the first page goes out of it (the oldest page has been loaded, the first page goes out).

2- **LRU**: The fewest reference page goes out.

Ex: Suppose the memory space has 3 pages, and the referenced page is: 4,2,3,4,5,1,4. Show the replacement policies using FIFO & LRU?

FIFO:

4	2	3	4	5	1	4
4*	4*	4*	4*	5	5	5*
	2	2	2	2*	1	1
		3	3	3	3*	4
			Hit			

LRU:

4	2	3	4	5	1	4
4*	4*	4*	4	4	4*	4
	2	2	2*	5	5	5*
		3	3	3*	1	1
			Hit			**Hit**

Ex: An address space is specified by 24 bits and the corresponding memory space by 16 bits. Find:

a) How many words are there in the address space?

b) How many words are there in the memory space?

c) If a page consists of 2K words, how many pages are there in the system?

Sol:

a) no. of words in address space = $2^{24} = 2^4 \times 2^{20}$ = 16 M words.

b) no. of words in memory space = $2^{16} = 2^6 \times 2^{10}$ = 64 K words.

c) no. of pages $= \dfrac{\text{no of words in address space}}{\text{no of words in each page}} =$

$$= \frac{2^4 \times 2^{20}}{2^1 \times 2^{10}} = 2^3 \times 2^{10} = 8\text{K} = 8192$$

Segmentation:

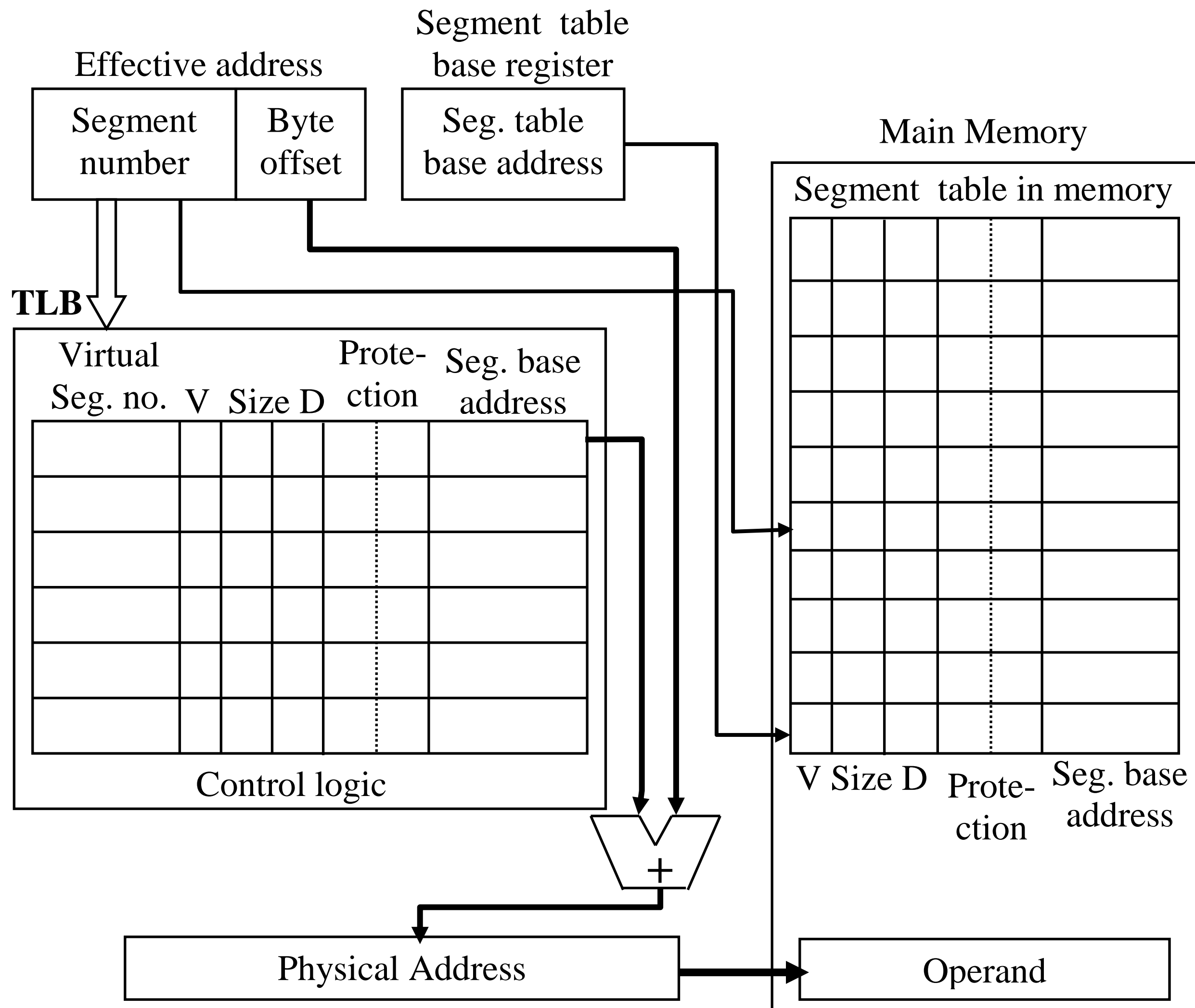
Like paging, segmentation is a virtual memory technique. Segmentation differs from paging in a number of ways: Instead of dividing logical addresses into pages of a fixed size and main memory into fixed size page frames. The hardware divides logical addresses into segments of arbitrary size, and the processor treats main memory as a single block.

Segments that contains only procedure code are called (Code Segment), and those with only data are called (Data Segment). Segment tend to be much larger than pages (frequently as large as 64KB).

Moreover, segments can usually range in size, where pages are always in one size. For a given system-segment sizes are chosen to reflect the sizes of the corresponding code or data they contain.

The size of the segments determined by the OS, or by the user if the user use the assembly language to expand the size of the segments.

The protection of data using the segmentation technique is better than the protection using paging technique.



(Components of Segmentation System)

Size: holds the size of the segment.

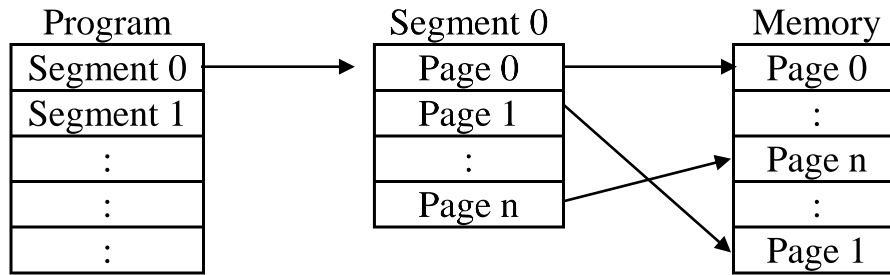
The physical address in paging technique found by binding the byte offset with page frame. But in segmentation technique, the physical address found by gathering the byte offset with segment address.

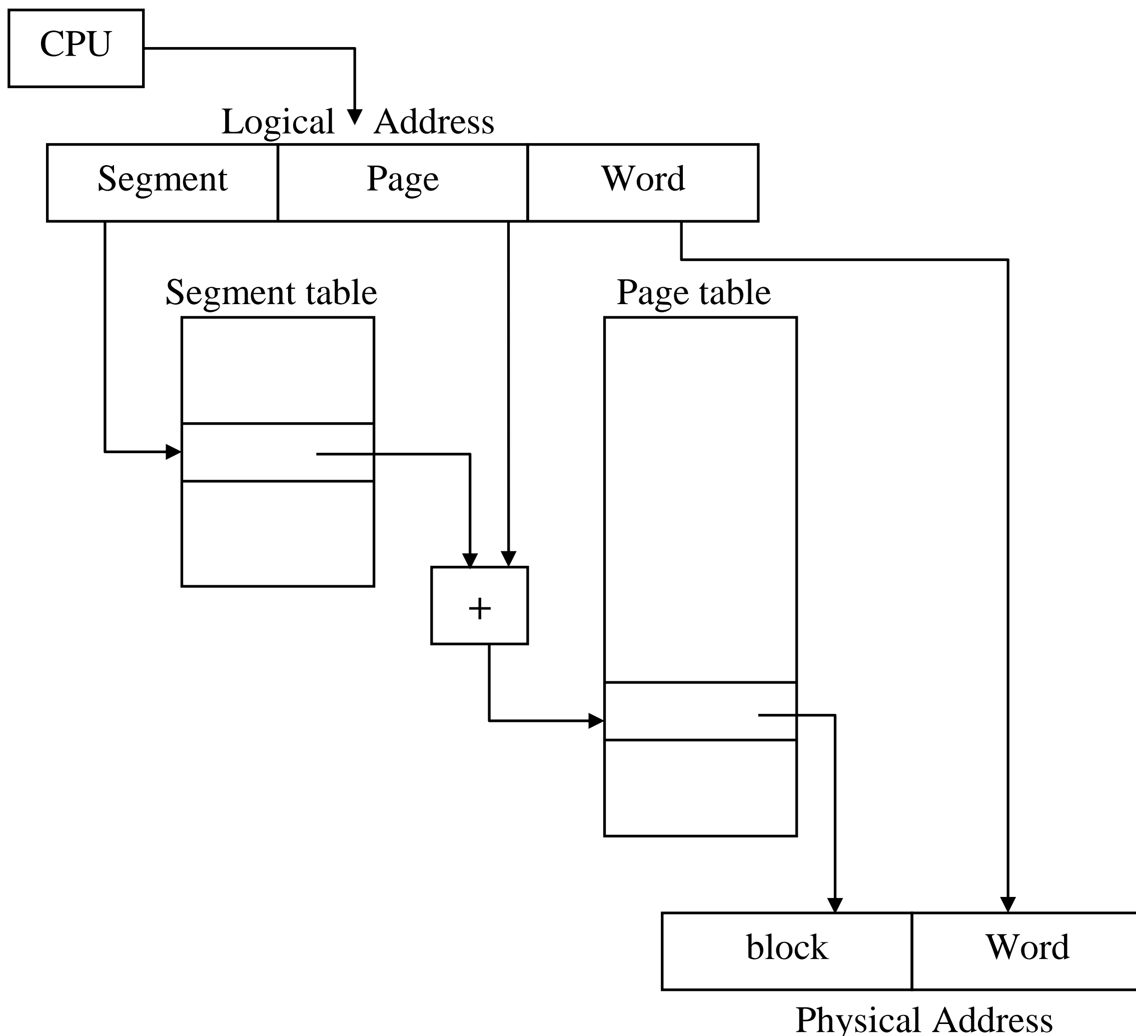
<u>Segmentation with Paging</u>:

Paging and segmentation can be combined in an attempt to give the advantages of both. This is done, by dividing each segment into pages. The memory map consists of segment table and set of page tables, one of each segment address is a pointer to the base of the corresponding page table. The page table is used in the usual way to determine the required physical address.

The greater advantage of breaking a segment into pages is that: elimination the need to store the segments in contiguous region of main memory.

In brief, the program is divided into segments, and the system will deal with each segment as a separate program, so, each segment need to be divided into many pages. Therefore, it will need a one single segment table with many page tables.





(Logical to Physical address mapping)

Ex: Suppose logical address length =20 bits, divided into the following: segment field =4 bits, page field =8 bits, word field =8 bits, find:

1) Total segments?
2) Total pages in each segment?
3) Total words in each page?
4) Number of words in the smallest segment size?
5) Number of words in the largest segment size?
6) Number of blocks in main memory?

Sol:

Segment	Page	Word
4	8	8

12 Block	8

1) Total segments = 2^4 16 segment.

$0 \leq$ segment number ≤ 15

2) Total pages in each segment = $2^8 = 256$ pages

$0 \leq$ page number ≤ 255

3) Total words in each page = $2^8 = 256$ words

$0 \leq$ word number ≤ 255

4) Smallest segment will have 1 page,

$\therefore$ one page consists of $2^8 = 256$ words.

5) Largest segment will have 2^8 (256) pages,

$\therefore 2^8$ pages $\times 2^8$ words $= 2^{16}$ words $= 2^6 \times 2^{10} = 64$ Kwords.

6) No. of blocks in main = 2^{12} = 4 K block in main memory.

Ex: The logical address space in a computer consists of 256 segments. Each segment can have up to 64 pages of 1K words.

a) Formulate the logical address format?

b) Give the binary representation of the logical address format for segment 20 and word number 16 in page 15?

Sol:

No. of segment = 256 = $2^8 \rightarrow$ 8 bits of segment field.

No. of pages = 64 = $2^6 \rightarrow$ 6 bits of page field.

No. of words = 1K = $2^{10} \rightarrow$ 10 bits of word field.

a)

	Segment	Page	Word
Logical Address	8	6	10

b)

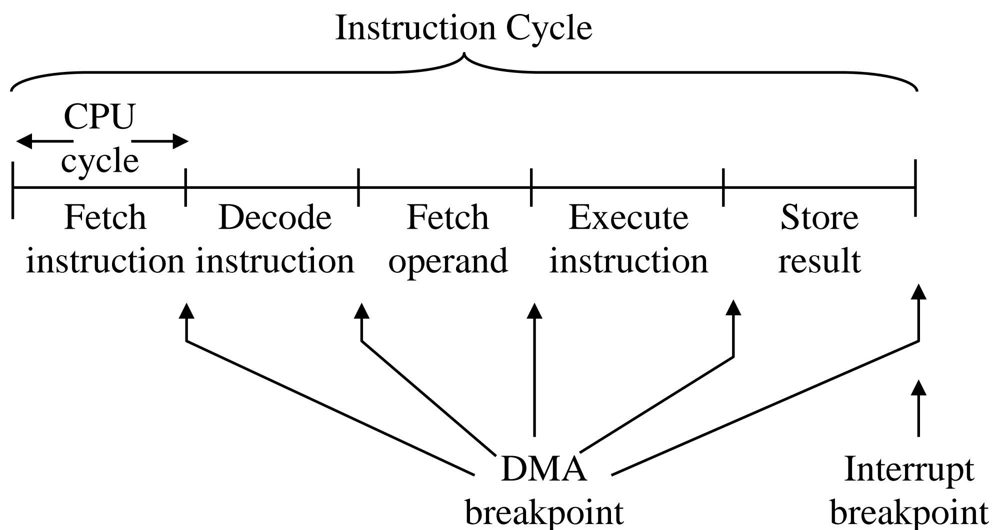
Segment	Page	Word
0 0 0 1 0 1 0 0	0 0 1 1 1 1	0 0 0 0 0 1 0 0 0 0

Direct Memory Access (DMA)

The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of the transfer. This transfer technique is called direct memory access (DMA).

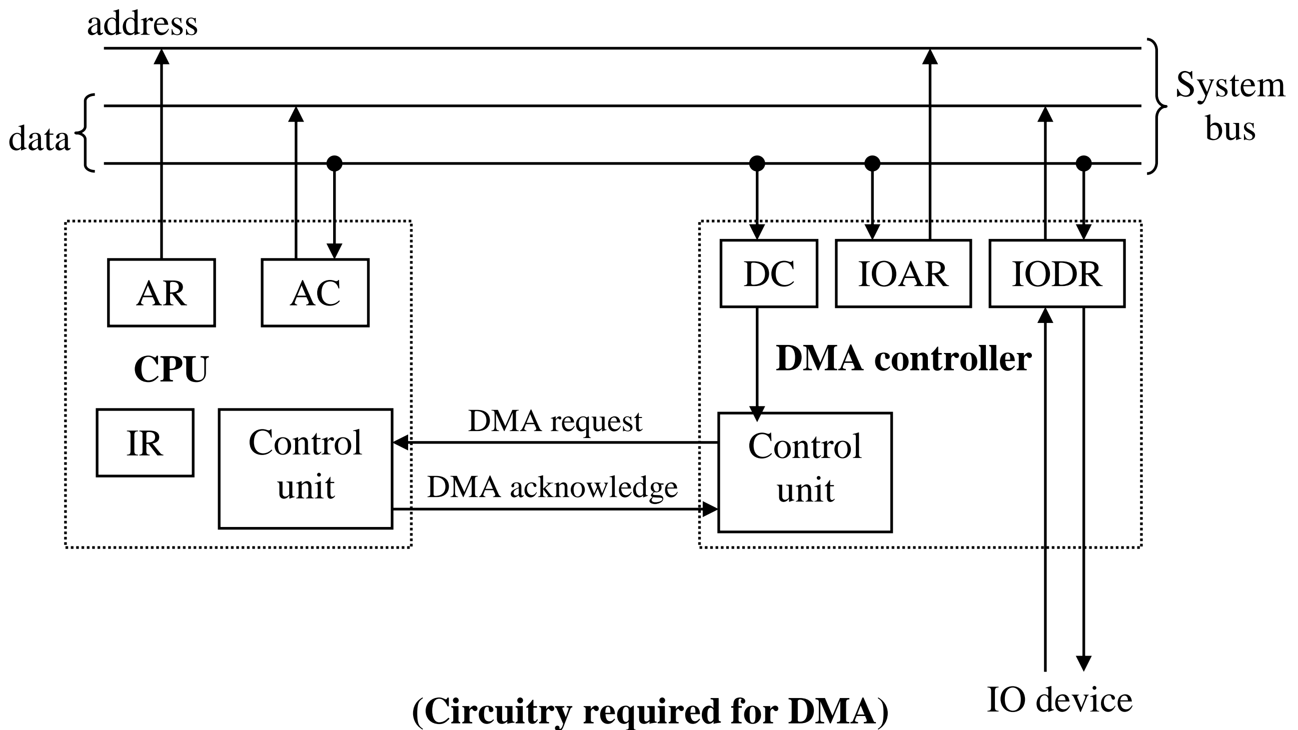
During DMA transfer, the CPU is idle and has no control of the memory buses. A DMA controller takes over the buses to manage the transfer directly between the I/O device and memory.

DMA circuits are used to increase the speed of I/O operations and eliminate most of the role played by the CPU in such operations. Special control line to which we assign the generic names DMA REQUEST go from the I/O devices to the CPU.



(DMA breakpoint during an instruction cycle)

This figure shows a typical sequence of CPU actions during an instruction cycle. Thus, during the instruction cycle, there are five points in time (breakpoints) when the CPU can respond to a DMA request. When the CPU receives such a request, it waits until the next breakpoint, releases the system bus, and signals the requesting I/O device by activating a DMA ACKNOWLEDGE control line.



(Circuitry required for DMA)

Essential Parts of DMA Controller

1) **IODR** (IO Data Register): contains the data that will be transferred from main memory to the IO device or from IO device to the main memory.

2) **IOAR** (IO Address Register): contains the address of the data to be transferred. This register is incremented after each word that is transferred to memory.

3) **DC** (Data Count) (Word Count Register): holds the number of words to be transferred. This register is decremented by one after each word transfer and internally tested for zero.

4) **Control Unit**: controls the DMA controller, and specifies the mode of transfer.

Types of DMA

When the DMA takes control of the bus system, it communicates directly with the memory. The transfer can be made in several ways:

1) (Burst Transfer) Block Transfer

In DMA burst transfer, a block sequence consisting of a number of memory words is transferred in a continuous burst while the DMA controller is master of the memory buses. This mode of transfer is needed

for fast devices where data transmission cannot be stopped or slowed down until an entire block is transferred.

2) Cycle Stealing

Allows the DMA controller to transfer one data word at a time, after which it must return control of the buses to the CPU. This means that long blocks of IO data are transferred by a sequence of DMA bus transactions interspersed (اجراء نثر) with CPU bus transactions. The CPU merely delays its operation for one memory cycle to allow the direct memory I/O transfer to "steal" one memory cycle.

3) Transparent DMA

Designing the DMA interface so that bus cycles are stolen only when the CPU is not actually using the system bus.

DMA Transfer

1) The CPU executes two IO instructions, which load the DMA register IOAR with the base address of the main memory region to be used in the data transfer and DC register with the number of words to be transferred.

2) When the DMA controller is ready to transmit or receive data, it activates the DMA REQUEST line to the CPU, the CPU wait for the next DMA breakpoint. It then relinquishes control of the data and address lines (system bus) and activates the DMA ACKNOWLEDGE.

3) The DMA controller now transfers data directly to or from main memory. After a word is transferred, IOAR incremented and DC decremented.

4) If DC is not decremented to zero but the IO device is not ready to send or receive the next batch of data, the DMA controller returns control to the CPU by releasing the system bus and deactivating the DMA REQUEST line. The CPU responds by deactivating DMA ACKNOWLEDGE and resuming normal operation.

5) If DC is decremented to zero, the DMA controller again relinquishes control of the system bus. It may also send an interrupt signal to the CPU. The CPU responds by halting the IO device by initiating a new DMA transfer.

CPU

The things that the CPU must do:

1- **Fetch Instruction**: The CPU must read instruction from memory.

2- **Interpret Instruction**: The instruction must be decoded to determine what action is required.

3- **Fetch Data**: The execution of an instruction may require reading data from memory or an I/O module.

4- **Process Data**: The execution of an instruction may require performing arithmetic or logical operation on data.

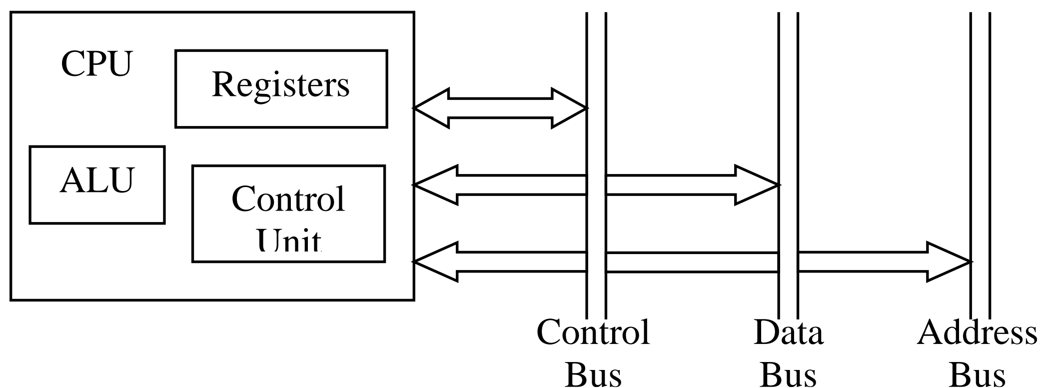
5- **Write Data**: The result of an execution may require writing data to memory or I/O module.

The major components of CPU are:

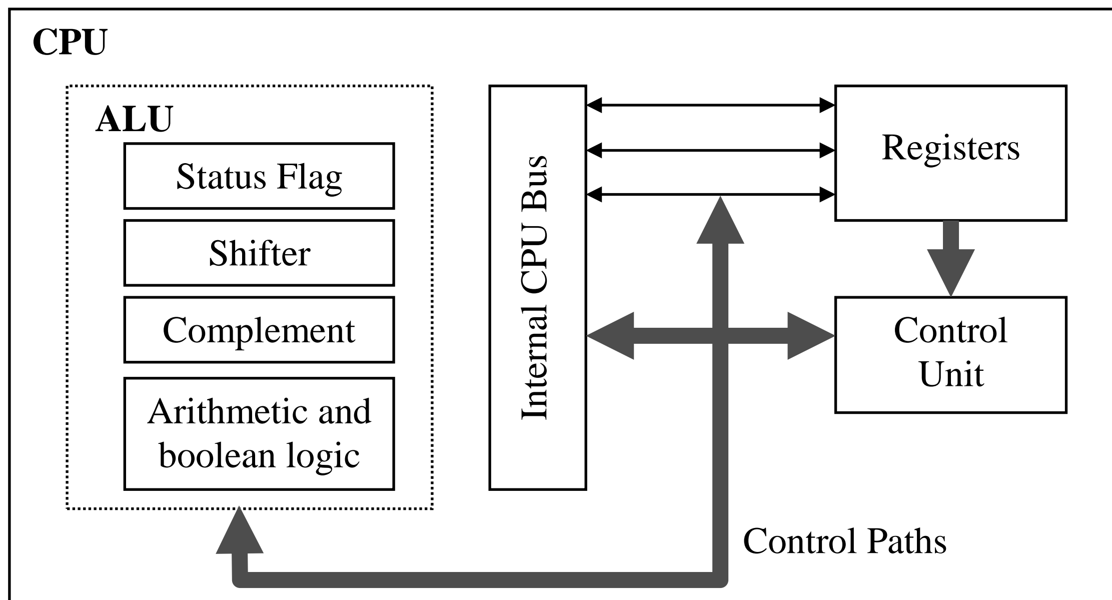
1- ALU.

2- Registers.

3- Control Unit.



(CPU with the system bus)



(Internal Structure of the CPU)

<u>Register Organization</u>

The registers in the CPU save two functions:

1- **<u>User Visible Register</u>**:

To enable the machine or assembly language programmer to minimize main memory references by optimizing use of registers. We can characterize these registers in the following categories:

- General purpose.
- Data.
- Address (i.e.: segment register)
- Condition codes (i.e.: flag register)

2- **<u>Control and Status Registers</u>**:

These are used by the control unit to control the operations of the CPU and the execution of program.

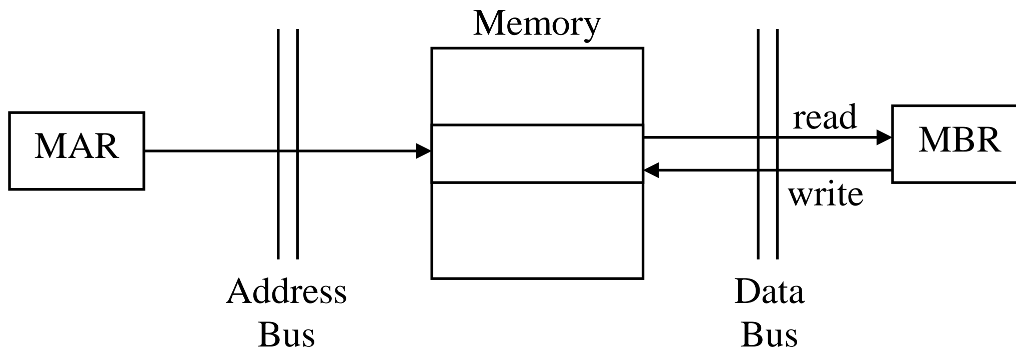
Four registers are essential to instruction execution:

- Program Counter (**PC**): Contains the address of an instruction to be fetched.

- Instruction Register (**IR**): Contains the instruction most recently fetched.

- Memory Address Register (**MAR**): Contains the address of a location in memory.

- Memory Buffer/Data Register (**MBR**) (**MDR**): Contains a word of data to be written to memory or the word most recently read.



Control Unit

The purpose of the control unit is to issue control signals or instruction to the data processing part. These control signals select the function to be performed at specific times and rout the data through the appropriate functional units.

Implementation Methods

1- Hardwired Control.
2- Microprogrammed Control.

1)) Hardwired Control

This implementation views the control unit as a sequential logic circuits to generate specific fixed sequence of control signals. Once constructed change in behaviour can be implemented only by redesigning, and physically rewiring the unit.

For designing such an implementation, the designers take the instruction set and designing the hardwired part for each instruction under consideration of all possibilities of each instruction, then they build the boolean algebra functions which represent the output of logical gates. These output represent the control signals. Meaning that each control signal generated by logical circuits for one specific instruction.

This type of implementation used in RISC computers (Reduce Instruction Set Computer). And because of constructing this unit from many logical gates, it guarantees speed for such unit, but also it increases the heat of this unit and consequently increases the heat of the CPU, therefore, a fan is needed for cooling the CPU.

2)) Microprogrammed Control

It is a method of control design, which the control signal selection and sequencing information is stored in control memory (CM), which stored in the control unit in the CPU.

The control signals to be activated at any time by a microinstruction. This microinstruction is fetched from the CM in much the same way an instruction is fetched from main memory.

The CM containing the microinstructions that are generates the control signals. For executing such a microinstruction, it must fetch this microinstruction from the CM, then decoding it, then executing it. Therefore, the microinstruction consists of many bits, each of represent a microoperation. Each bit controls one control line, if this bit is 0, it means this line is inactivated, if this bit is 1, it means this line is activated.

Any change in behaviour of an instruction, it needs to change the values of the bits inside the CM form 0 to 1 or from 1 to 0. Thus, this method is easy and quick and more flexible if need a change, but it is slower than the hardwired in execution.

Microprogram:

The sequence of microinstructions that represent a single machine instruction.

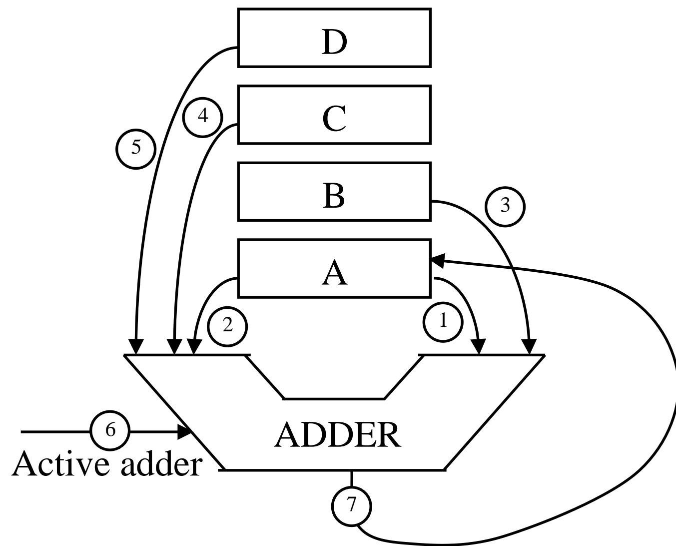
Microinstruction:

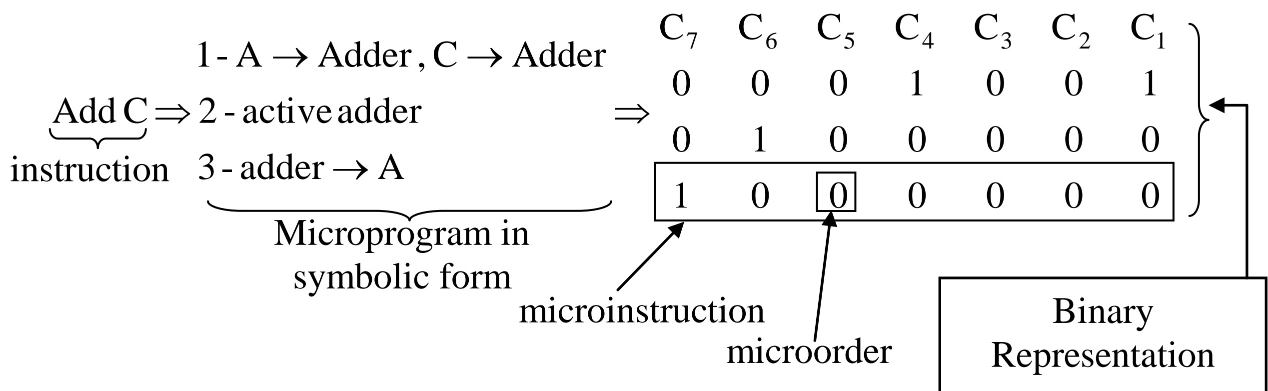
The set of microorders issued by the control unit at one a time.

Microorders (Microoperation):

Individual signals sent over dedicated lines to control individual components and devices.

<u>Ex</u>: Explain the microinstruction for the instruction (ADD C), where the number of the control lines of the system is equal to 7 and illustrated in this figure:





Let us assume that each instruction occupies one memory word. Therefore, execution of one instruction requires the following three steps to be performed by the CPU:

1- Fetch the contents of the memory location pointed at by the PC. The contents of this location are interpreted as an instruction to be executed. Hence, they are stored in the instruction register (IR). Symbolically, this can be written as:

$$IR \leftarrow [[PC]]$$

2- Increment the contents of the PC by 1.

$$PC \leftarrow [PC] + 1$$

3- Carry out the actions specified by the instruction stored in the IR.

Steps 1 and 2 can be repeated as many times as necessary to fetch the complete instruction. These two steps are usually referred to as the *fetch phase*, while step 3 constitutes the *execution phase*.

The figure below shows the arithmetic and logic unit (ALU) and all CPU registers are connected via a single common bus. This bus is internal to the CPU, and should not be confused with the external bus, or buses connecting the CPU to the memory and I/O devices. The external memory bus is connected to the CPU via the memory data and address registers MDR and MAR. the number and function of registers R0 to R(n-1) vary from one machine to another. They may be provided for general-purpose used by the programmer, some of them may be dedicated as special-purpose registers, such as index registers or stack pointers.

The registers Y and Z used only by the CPU for temporary storage during execution of some instructions. However, they are never used for storing data generated by one instruction.

Most of the operations in steps 1 to 3 can be carried out by performing one or more of the following functions in some pre-specified sequence:

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

Let us now consider in some detail the way in which each of the above functions is implemented in a typical computer.

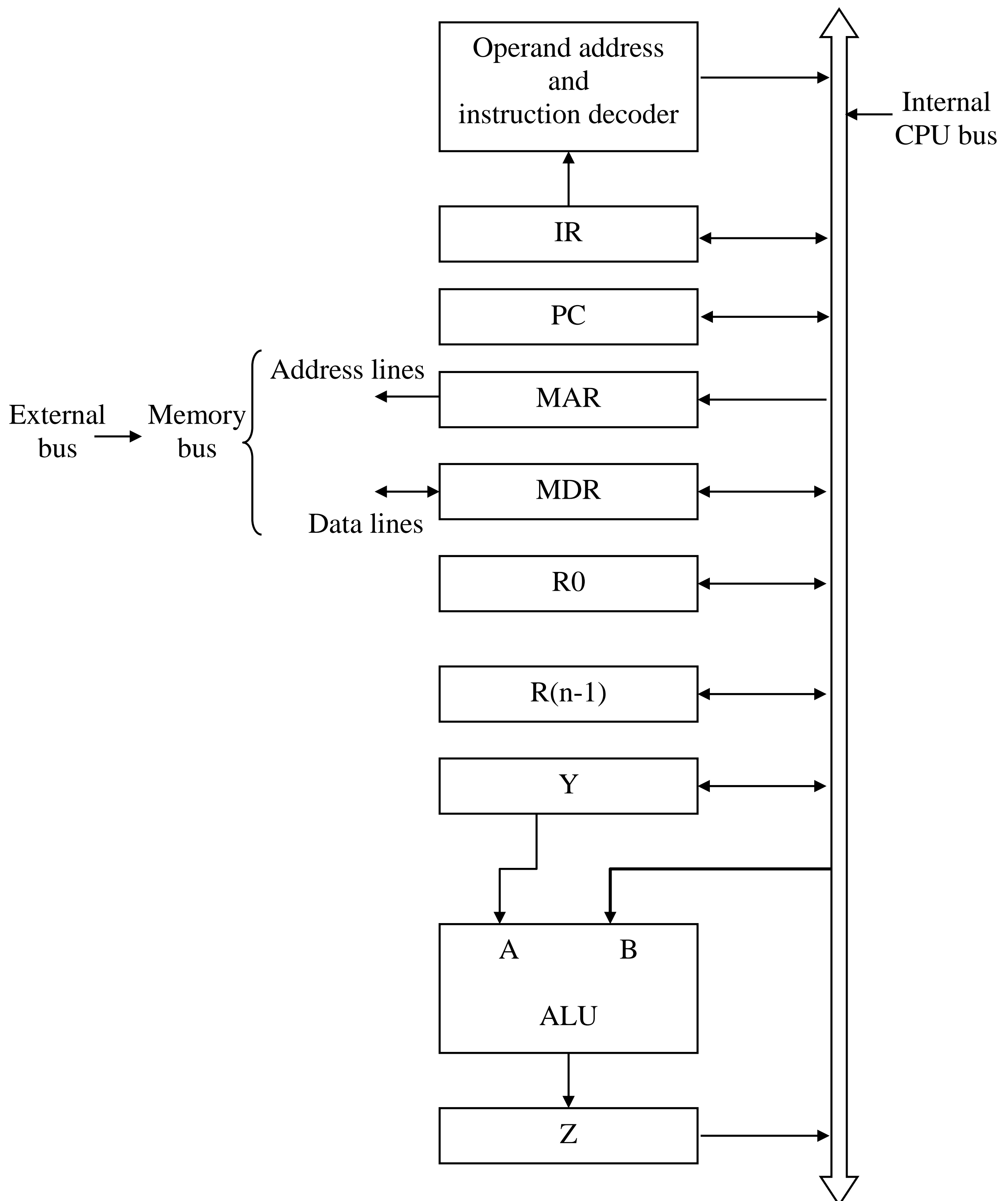


Figure 1: (Single-bus organization of the data paths inside the CPU)

1) Fetching a word from memory (Read Fetch):

To fetch a word of information from memory, the CPU has to specify the address of the memory location where this information is stored and request a READ operation. This applies whether the information to be fetched represents a new instruction in a program or a word of data (operand) specified by an instruction.

Thus, to perform a memory fetch, the CPU transfers the address of the required information word to the memory address register (MAR). Then this address is transferred to the main memory. Meanwhile, the CPU uses the control lines of the memory bus to indicate that a READ operation is required. Then, the CPU waits until it receives an answer from the memory, informing it that the requested function has been completed, this is accomplished through the use of another control signal on the memory bus, which will be referred to as memory-function-completed (MFC).

As soon as the MFC signal is set to 1, the information on the data lines is loaded into MDR and is thus available for use inside the CPU.

As an example, assume that the address of the memory location to be accessed is in register R1 and that the memory data is to be loaded into register R2. This is achieved by the following sequence of operation:

1. MAR ← [R1]
2. READ
3. Wait for the MFC signal
4. R2 ← [MDR]

The duration of step 3 depends upon the speed of the memory used. The functions that do not require the use of MDR or MAR can be carried out during the MFC. Such a situation arises during the fetch phase, that is the PC can be incremented while waiting for the Read operation to be completed.

The transfer mechanism where one device initiates the transfer (READ request) and wait until the other device responds (MFC signal) is referred to as an *asynchronous* transfer. It can be easily seen that this mechanism enable transfer of data between two independent devices that have different speeds of operation. An alternative scheme that can be found in some computers uses *synchronous* transfers.

2) Storing a Word into Memory (Write):

The data word to be written should be loaded into the MDR before the WRITE command is issued. Assume that the data word to be stored in the memory is in R2 and that the memory address is in R1, the WRITE operation requires the following sequence:

1. MAR ← [R1]
2. MDR ← [R2]
3. WRITE
4. Wait for MFC

Steps 1 and 2 are independent. Therefore, they can be carried out in any order. In fact, steps 1 and 2 can be carried out simultaneously, this would not be possible in the single-bus organization.

3) Register Transfers:

The input and output gates for register Ri are controlled by the signals Ri_{in} and Ri_{out} , respectively. Thus, when Ri_{in} is set to 1, the data available on the common bus is loaded into Ri. Similary, when Ri_{out} is set to 1, the contents of register Ri are placed on the bus. While Ri_{out} is equal to 0, the bus can be used for transferring data from other registers.

For example, to transfer the contents of register R1 to register R4, the following actions are needed:

- Enable the output gate of register R1 by setting $R1_{out}$ to 1. This places the contents of R1 on the CPU bus.
- Enable the input gate register R4 by setting $R4_{in}$ to 1. This loads data from the CPU bus into register R4.
  This data transfer can be represented symbolically as

$$R1_{out} , R4_{in}$$

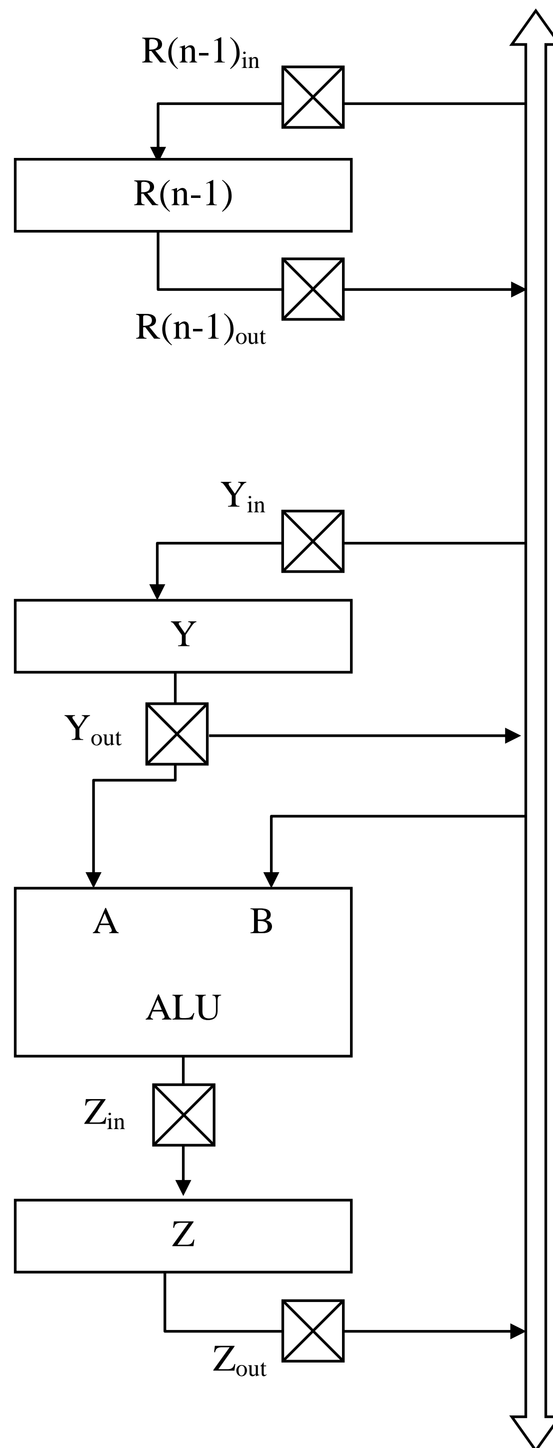


Figure 2: (input and output gating for the registers in Figure 1)

4) **Performing an Arithmetic or Logic Operation:**

The two numbers to be added should be made available at the two inputs of the ALU simultaneously. Register Y in figure 1, is provided for this purpose. It is used to hold one of the two numbers while the other number is gated to the bus. The result is stored temporarily in register Z. therefore, the sequence of operations to add the contents of register R1 to register R2 and store the result in register R3 should be as follows:

Step	Action
1.	$R1_{out}$, Y_{in}
2.	$R2_{out}$, Add , Z_{in}
3.	Z_{out} , $R3_{in}$

In step 2 of this sequence the contents of register R2 are gated to the bus, hence to input B of the ALU which is connected directly to the bus. The contents of register Y are always available at input A. the performed by the ALU depends upon the signal applied to the ALU control lines. In this case, the ADD line is set to 1, causing the output of the ALU to be the sum of the two numbers at A and B. this sum is loaded into register Z, since its input gate is enabled (Z_{in}). in step 3, the contents of register Z are transferred to the destination register R3.

Multibus Organization:

The single-bus organization of figure 1 represents only one of the possibilities for interconnecting different building blocks of the CPU. An alternative arrangement is the two-bus structure shown in figure 3. All register outputs are connected to bus A, and all register inputs are connected to bus B. The two buses are connected through the bus tie G, which, when enabled, transfers the data on bus A to bus B. When G is disabled, the two buses are electrically isolated. Note that the temporary storage register Z in figure 1 is not required in this organization because, with the bus tie disabled, the output of the ALU can be transferred directly to the destination register. For example, the addition operation ($R3 \leftarrow [R1] + [R2]$) can now be performed as follows:

Step	Action
1.	$R1_{out}$, G_{enable} , Y_{in}
2.	$R2_{out}$, Add , ALU_{out} , $R3_{in}$

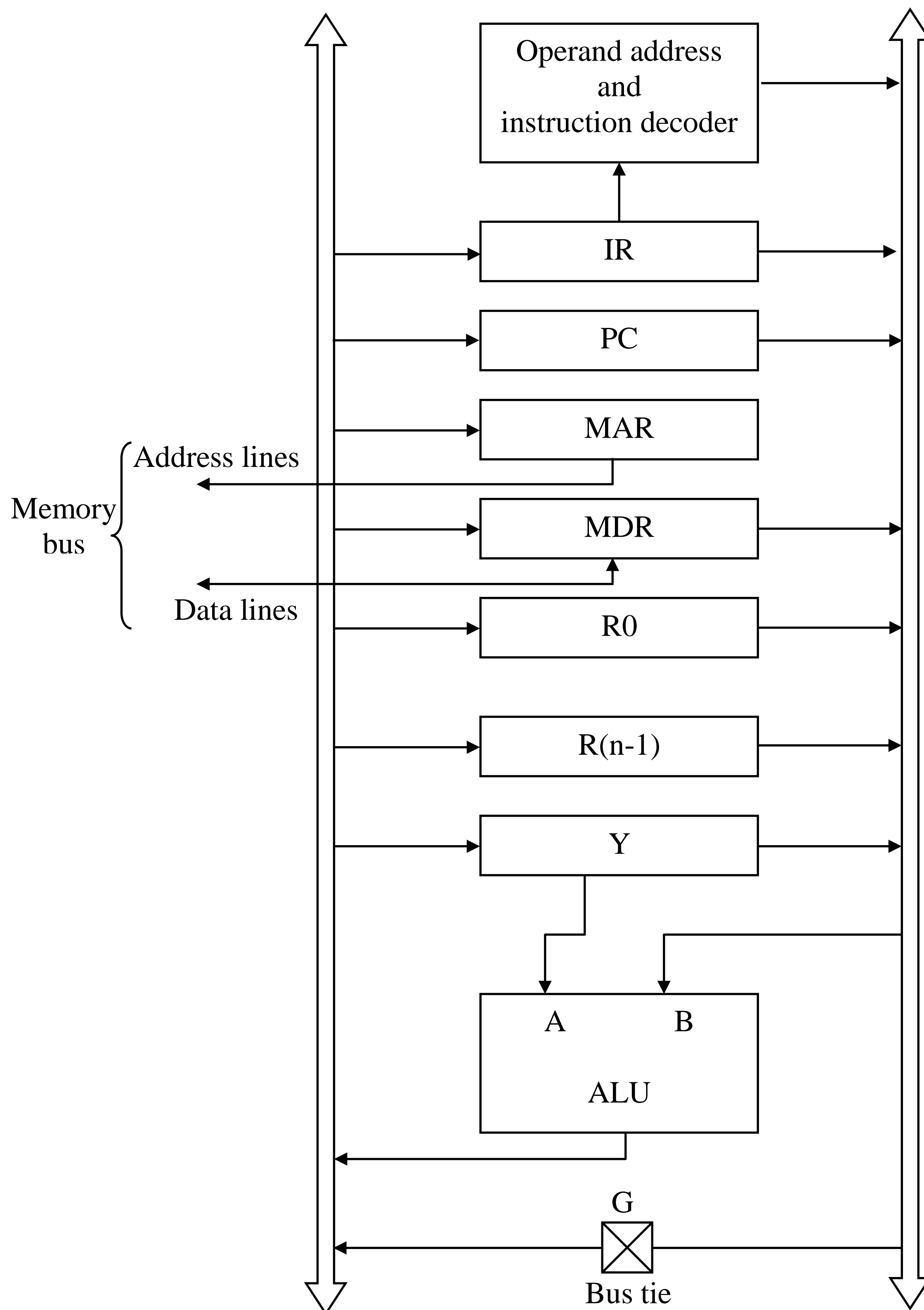


Figure 3: (Multi-bus organization of the data paths inside the CPU)

Execution Of A Complete Instruction

Let us now try to put together the sequence of elementary operations required to execute one instruction. Consider the instruction "Add contents of memory location NUM to register R1".

ADD R1 , [NUM]

Figure 4 gives the sequence of control steps required to implement the above operations for the single-bus architecture of figure 1. Thus, instruction execution proceeds as follows:

Step	Action
1.	PC_{out} , MAR_{in} , Read , Clear Y , Set carry-in to ALU , Add , Z_{in}
2.	Z_{out} , PC_{in} , Wait for MFC
3.	MDR_{out} , IR_{in}
4.	Address-field-of-IR_{out} , MAR_{in} , Read
5.	$R1_{out}$, Y_{in} , Wait for MFC
6.	MDR_{out} , Add , Z_{in}
7.	Z_{out} , $R1_{in}$
8.	End

Figure 4: "Add contents of memory location to register R1"

In step 1, the instruction fetch operation is initiated by loading the contents of the PC into the MAR and sending a Read request to the memory. At the same time the PC is incremented by 1 through the use of the ALU. This is accomplished by setting one of the inputs to the ALU (register Y) to 0 and by setting the other input (CPU bus) to the current value in PC. At the same time the carry-in to the ALU is set to 1 and an Add operation is specified. The updated value is moved from register Z back into the PC during step 2. Note that step 2 is started immediately after issuing the memory Read request without the need to wait for MFC. Step 3, however, has to be delayed until the MFC is received. In step 3, the word fetch from the memory is loaded into the IR. Step 1 through 3 constitutethe instruction fetch phase of the control sequence. Of course, this portion is the same for all instructions.

Steps 4 to 8, which can be referred to as the execution phase. In step 4, the address field of the IR, which contains the address NUM, is gated to the MAR, and a memory Read operation is initiated. Then the contents of R1 are transferred to register Y. when the Read operation is completed, the memory operand is available in register MDR. The addition operation is performed in step 6, and the result is transferred to R1 in step 7. The End signal, step 8, indicates completion of execution of the current instruction and usually causes a new fetch cycle to be started by going back to step 1.

EXAMPLES:

EXG R1 , R2

Step	Action
1.	PC_{out} , MAR_{in} , Read , Clear Y , Set carry-in to ALU , Add , Z_{in}
2.	Z_{out} , PC_{in} , Wait for MFC
3.	MDR_{out} , IR_{in}
4.	$R1_{out}$, Y_{in}
5.	$R2_{out}$, $R1_{in}$
6.	Y_{out} , $R2_{in}$
7.	End

CLEAR R1

Step	Action
1.	PC_{out} , MAR_{in} , Read , Clear Y , Set carry-in to ALU , Add , Z_{in}
2.	Z_{out} , PC_{in} , Wait for MFC
3.	MDR_{out} , IR_{in}
4.	Clear Y , Y_{out} , $R1_{in}$
5.	End

INC [NUM]

Step	Action
1.	PC_{out} , MAR_{in} , Read , Clear Y , Set carry-in to ALU , Add , Z_{in}
2.	Z_{out} , PC_{in} , Wait for MFC
3.	MDR_{out} , IR_{in}
4.	MDR_{out}, MAR_{in} , Read
5.	Wait for MFC
6.	MDR_{out} , Clear Y , Set Carry , Add , Z_{in}
7.	Z_{out} , MDR_{in} , Write
8.	Wait for MFC
9.	End

RETURN

Step	Action
1.	PC_{out} , MAR_{in} , Read , Clear Y , Set carry-in to ALU , Add , Z_{in}
2.	Z_{out} , PC_{in} , Wait for MFC
3.	MDR_{out} , IR_{in}
4.	SP_{out} , MAR_{in} , Read , Clear Y , Set Carry , Add , Z_{in}
5.	Z_{out} , SP_{in} , Wait for MFC
6.	MDR_{out} , PC_{in}
7.	End

Branching

Branching is accomplished by replacing the current contents of the PC by the branch address, that is, the address of the instruction to which branching is required. The branch address is usually obtained by adding an offset X, which is given in the address field of the branch instruction to the current value of the PC.

Figure 5 gives a control sequence that enables execution of an unconditional branch using the single-bus organization of figure 1. Execution starts as usual with the fetch phase until the instruction is loaded into the IR in step 3. To execute the branch instruction, the contents of the PC are transferred to register Y in step 4. Then, the offset X is gated to the bus, and the addition operation is performed. The result, which represents the branch address, is loaded into the PC in step 6.

Step	Action
1.	PC_{out} , MAR_{in} , Read , Clear Y , Set carry-in to ALU , Add , Z_{in}
2.	Z_{out} , PC_{in} , Wait for MFC
3.	MDR_{out} , IR_{in}
4.	PC_{out} , Y_{in}
5.	Address-field-of-IR_{out} , Add , Z_{in}
6.	Z_{out} , PC_{in}
7.	End

Figure 5: (Control sequence for an unconditional branch instruction)

The offset X should be the difference between the branch address and the address immediately following the branch instruction. For example, if the branch instruction is at location 1000, and it is required to branch to location 1050, the value of X should be set to 49.

Consider now the case of a conditional branch. The only difference between this case and that is the need to check the status of the condition codes between step 3 and 4. For example, if the instruction decoding circuitry interprets the contents of the IR as a Branch on Negative (BRN) instruction, the control unit proceeds as follows. First, the condition code register is checked. If bit N (negative) is equal to 1, the control unit proceeds with step 4 through 7 as in figure 5 (adding offset X). If N is equal to 0, an End signal is issued. This terminates execution of the branch instruction and causes the instruction immediately following in the program to be fetched when a new fetch operation is performed.

Therefore, the control sequence for the conditional branch instruction BRN can be obtained in figure 6.

Step	Action
1.	PC_{out} , MAR_{in} , Read , Clear Y , Set carry-in to ALU , Add , Z_{in}
2.	Z_{out} , PC_{in} , Wait for MFC
3.	MDR_{out} , IR_{in}
4.	if $\overline{N}$ then End
5.	else PC_{out} , Y_{in}
6.	Address-field-of-IR_{out} , Add , Z_{in}
7.	Z_{out} , PC_{in}
8.	End

Figure 6: (Control sequence for an conditional BRN instruction)

If the instruction decoding circuitry interprets the contents of the IR as a Branch on Zero (BRZ) instruction, the control unit proceeds as follows:

Step	Action
1.	PC_{out} , MAR_{in} , Read , Clear Y , Set carry-in to ALU , Add , Z_{in}
2.	Z_{out} , PC_{in} , Wait for MFC
3.	MDR_{out} , IR_{in}
4.	if $\overline{Z}$ then End
5.	if Z then
6.	PC_{out} , Y_{in}
7.	Address-field-of-IR_{out} , Add , Z_{in}
8.	Z_{out} , PC_{in}
9.	End

Parallelism in Microinstructions:

Microprogrammable processors are frequently characterized by the maximum number of microoperations that can be specified by a single Microinstruction. This number can vary from one to several hundred. Microinstructions that specify a single microoperation are quit similar to conventional machine instructions. They are relatively short, but due to the lack of parallelism, more microinstructions may be needed to perform a given operation.

Microinstructions are often designed to take advantage of the fact that at the microprogramming level, many operations can be performed in parallel.

Types of Microinstructions:

1) Horizontal Microinstructions.

2) Vertical Microinstructions.

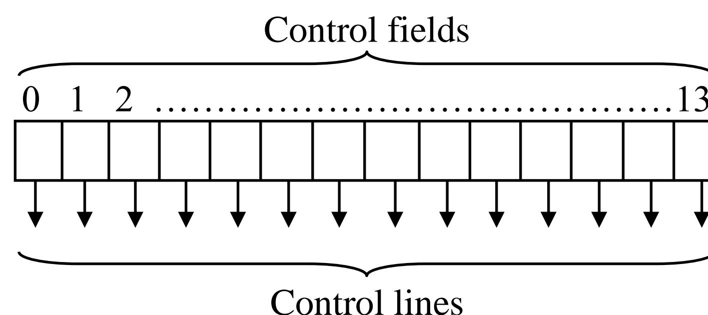
1) Horizontal Microinstructions:

In this form, for each microorder one bit is provided to achieve a corresponding microoperation. Each bit is a control field for a control line. Horizontal microinstructions have the following general attributes:

1- Long formats, thus the CM width depends on how long this format is?

2- Ability to express a high degree of parallelism.

3- Little encoding of the control information.



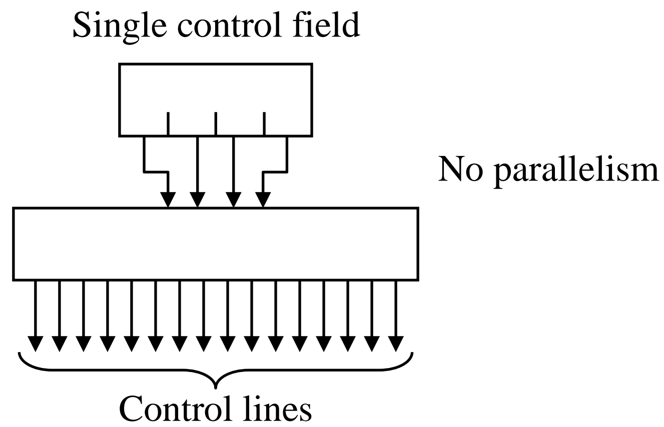
2) Vertical Microinstructions:

This type of microinstructions divided into control fields that each control field represented by many control bits. Therefore, each control field must be connected to a decoder from which the control signals are derived. This type has the following attributes:

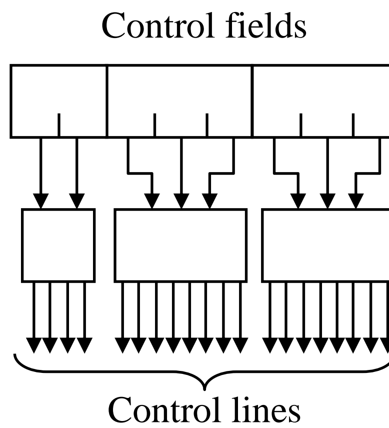
1- Short format.

2- Limited ability to express parallel microoperations.

3- Considerable encoding of the control information.



For a single control field microinstruction, there is no parallelism will be achieved, thus only a single one microoperation will take place in one cycle.



But for more control fields, each control field will achieve in a single microoperation, therefore, a few parallelisms will appear in one cycle.

Sometimes the two formats can be used together in a new format.