

Done by Mokhtar M. Hasan, Lecturer at the
Computer Science Department, College of
Science for Women, to fulfill the course
subject of Advanced Architecture material.

Advanced
Architecture,
Third Class,
Computer
Science
Department,
CSW, 2016-2017

Textbooks:

- 1- Computer Architecture, A Quantitative Approach, Fifth Edition, John L. Hennessy, David A. Patterson, Fifth Edition, 2012.
- 2- Computer Organization and Architecture, Designing for Performance, William Stallings, Eight Edition, 2010.
- 3- Computer Organization and Embedded Systems, C. Hamacher, Z. Vranesic, S. Zeki, N. Manjikian, 2012, Sixth Edition.

1. Introduction

Computer technology has made incredible progress in the roughly 65 years since the first general-purpose electronic computer was created. Today, less than \$500 will purchase a mobile computer that has more performance, more main memory, and more disk storage than a computer bought in 1985 for \$1 million. This rapid improvement has come both from advances in the technology used to build computers and from innovations in computer design.

Although technological improvements have been fairly steady, progress arising from better computer architectures has been much less consistent. During the first 25 years of electronic computers, both forces made a major contribution, delivering performance improvement of about 25% per year. The late 1970s saw the emergence of the microprocessor. The ability of the microprocessor to ride the improvements in integrated circuit technology led to a higher rate of performance improvement—roughly 35% growth per year.

This growth rate, combined with the cost advantages of a mass-produced microprocessor, led to an increasing fraction of the computer business being based on microprocessors. In addition, two significant changes in the computer marketplace made it easier than ever before to succeed commercially with a new architecture. First, the virtual elimination of assembly language programming reduced the need for object-code compatibility. Second, the creation of standardized, vendor-independent operating systems, such as UNIX and its clone, Linux, lowered the cost and risk of bringing out a new architecture.

These changes made it possible to develop successfully a new set of architectures with simpler instructions, called RISC (Reduced Instruction Set Computer) architectures, in the early 1980s. The RISC-based machines focused the attention of designers on two critical performance techniques, the exploitation of instruction level parallelism (initially through pipelining and later through multiple instruction issue) and the use of caches (initially in simple forms and later using more sophisticated organizations and optimizations).

Figure 1 shows that the combination of architectural and organizational enhancements led to 17 years of sustained growth in performance at an annual rate of over 50%—a rate that is unprecedented in the computer industry.

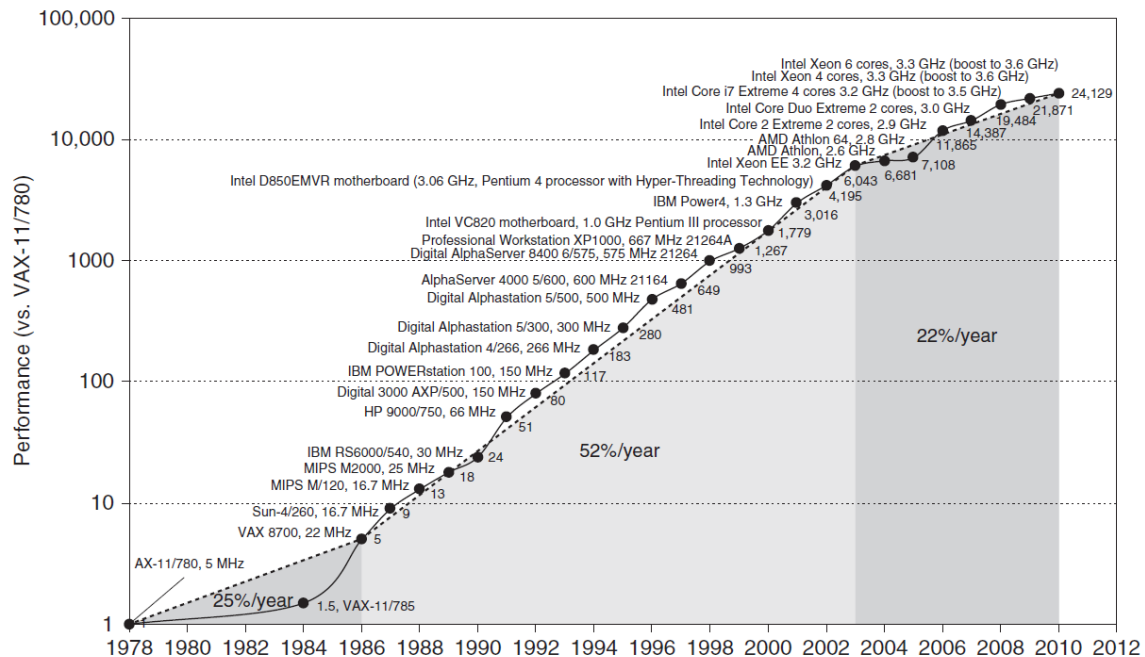


Figure 1: Growth in processor performance since the late 1970s.

The effect of this dramatic growth rate in the 20th century has been fourfold. First, it has significantly enhanced the capability available to computer users. For many applications, the highest-performance microprocessors of today outperform the supercomputer of less than 10 years ago. Second, this dramatic improvement in cost-performance leads to new classes of computers. Personal computers and workstations emerged in the 1980s with the availability of the microprocessor.

2. Classes of Computers

The historical evolution of computers can be divided roughly into the following classes:

- 1- Mainframes: In the 1960s, the dominant form of computing was on large mainframes, computers costing millions of dollars and stored in computer rooms with multiple operators overseeing their support. Typical applications included business data processing and large-scale scientific computing.
- 2- Minicomputer: The 1970s saw the birth of the minicomputer, a smaller-sized computer initially focused on applications in scientific laboratories, but rapidly branching out with the popularity of timesharing—multiple users sharing a computer interactively through independent terminals. That decade also saw the emergence of supercomputers, which were high-performance computers for scientific computing.

3- Desktop Computer: The 1980s saw the rise of the desktop computer based on microprocessors, in the form of both personal computers and workstations. The individually owned desktop computer replaced time-sharing and led to the rise of servers—computers that provided larger-scale services such as reliable, long-term file storage and access, larger memory, and more computing power.

4- PDA Electronics: The 1990s saw the emergence of the Internet and the World Wide Web, the first successful handheld computing devices (personal digital assistants or PDAs), and the emergence of high-performance digital consumer electronics, from video games to set-top boxes.

5- Embedded Computers: since 2000, where computers are lodged in other devices and their presence is not immediately obvious.

These changes in computer use have led to five different computing markets, each characterized by different applications, requirements, and computing technologies. Figure 2 summarizes these mainstream classes of computing environments and their important characteristics.

Feature	Personal mobile device (PMD)	Desktop	Server	Clusters/warehouse-scale computer	Embedded
Price of system	\$100–\$1000	\$300–\$2500	\$5000–\$10,000,000	\$100,000–\$200,000,000	\$10–\$100,000
Price of micro-processor	\$10–\$100	\$50–\$500	\$200–\$2000	\$50–\$250	\$0.01–\$100
Critical system design issues	Cost, energy, media performance, responsiveness	Price-performance, energy, graphics performance	Throughput, availability, scalability, energy	Price-performance, throughput, energy proportionality	Price, energy, application-specific performance

Figure 2: A summary of the five mainstream computing classes and their system characteristics.

Personal mobile device (PMD) is the term we apply to a collection of wireless devices with multimedia user interfaces such as cell phones, tablet computers, and so on. A **real-time performance** requirement means a segment of the application has an absolute maximum execution time. For example, in playing a video on a PMD, the time to process each video frame is limited, since the processor must accept and process the next frame shortly. Whereas **soft real-time**, arise when it is possible to occasionally miss the time constraint on an event, as long as not too many are missed. Other key characteristics in many PMD applications are the need to minimize

memory and the need to use energy efficiently. Energy efficiency is driven by both battery power and heat dissipation.

Desktop Computing The first, and probably still the largest market in dollar terms, is desktop computing. Since 2008, more than half of the desktop computers made each year have been battery operated laptop computers. Throughout this range in price and capability, the desktop market tends to be driven to optimize price-performance. This combination of performance (measured primarily in terms of compute performance and graphics performance) and price of a system is what matters most to customers in this market, and hence to computer designers. As a result, the newest, highest-performance microprocessors and cost-reduced microprocessors often appear first in desktop systems.

Servers As the shift to desktop computing occurred in the 1980s, the role of servers grew to provide larger-scale and more reliable file and computing services. Such servers have become the backbone of large-scale enterprise computing, replacing the traditional mainframe.

For servers, different characteristics are important. First, availability is critical. Consider the servers running ATM machines for banks or airline reservation systems. Failure of such server systems is far more catastrophic than failure of a single desktop, since these servers must operate seven days a week, 24 hours a day.

A second key feature of server systems is scalability. Server systems often grow in response to an increasing demand for the services they support or an increase in functional requirements. Thus, the ability to scale up the computing capacity, the memory, the storage, and the I/O bandwidth of a server is crucial.

Finally, servers are designed for efficient throughput. That is, the overall performance of the server—in terms of transactions per minute or Web pages served per second—is what is crucial.

Clusters/Warehouse-Scale Computers Clusters are collections of desktop computers or servers connected by local area networks to act as a single larger computer. Each node runs its own operating system, and nodes communicate using a networking protocol. The largest of the clusters are called warehouse-scale computers (WSCs), in that they are designed so that tens of thousands of servers can act as one. The growth of Software as a Service (SaaS) for applications like search, social networking, video sharing, multiplayer games, online shopping, and so on has led to the growth of a clusters.

The difference from servers is that WSCs **use redundant inexpensive components** as the building blocks, relying on a software layer to catch and isolate the many failures that will happen with computing at this scale.

Note that **scalability for a WSC is handled** by the local area network connecting the computers and not by integrated computer hardware, as in the case of servers.

Supercomputers are related to WSCs in that they are **equally expensive**, costing hundreds of millions of dollars, but supercomputers **differ** by emphasizing floating-point performance and by running large, communication-intensive batch programs that can run for weeks at a time. This tight coupling leads to use of much faster internal networks. In contrast, WSCs emphasize interactive applications, large-scale storage, dependability, and high Internet bandwidth.

Embedded Computers Embedded computers are found in everyday machines; microwaves, washing machines, most printers, most networking switches, and all cars contain simple embedded microprocessors. Embedded computers have the widest spread of processing power and cost. ¹They include 8-bit and ²16-bit processors that may cost less than a dime, ³32-bit microprocessors that execute 100 million instructions per second and cost under \$5, and ⁴high-end processors for network switches that cost \$100 and can execute billions of instructions per second large, price is a key factor in the design of computers for this space.

Two other key characteristics exist in many embedded applications: ¹the need to minimize memory and the ²need to minimize power. Larger memories also mean more power, and optimizing power is often critical in embedded applications. Although the emphasis on low power is frequently driven by the use of batteries.

3. Classes of Parallelism and Parallel Architectures

Parallelism at multiple levels is now the driving force of computer design across all four classes of computers, with energy and cost being the primary constraints.

There are basically two kinds of parallelism in applications:

1. Data-Level Parallelism (DLP) arises because there are many data items that can be operated on at the same time.
2. Task-Level Parallelism (TLP) arises because tasks of work are created that can operate independently and largely in parallel.

Computer hardware in turn can exploit these two kinds of application parallelism in four major ways:

1. Instruction-Level Parallelism exploits data-level parallelism at modest levels with compiler help using ideas like pipelining and at medium levels using ideas like speculative execution (it is an optimization technique where a computer system performs some task that may not be actually needed. The main idea is to do work before it is known whether that work will be needed at all, so as to prevent a

delay that would have to be incurred by doing the work after it is known whether it is needed. If it turns out the work was not needed after all, any changes made by the work are reverted and the results are ignored)^{wikipedia}.

2. Vector Architectures and Graphic Processor Units (GPUs) exploit data-level parallelism by applying a single instruction to a collection of data in parallel.
3. Thread-Level Parallelism exploits either data-level parallelism or task-level parallelism in a tightly coupled hardware model that allows for interaction among parallel threads.
4. Request-Level Parallelism exploits parallelism among largely decoupled tasks specified by the programmer or the operating system.

Flynn Classification (Taxonomy): Michael Flynn [1966] studied the parallel computing efforts in the 1960s, he found a simple classification whose abbreviations we still use today. He looked at the parallelism in the instruction and data streams called for by the instructions at the most constrained component of the multiprocessor, and placed all computers into one of four categories:

- 1- Single instruction stream, single data stream (SISD)— This category is the uniprocessor. The programmer thinks of it as the standard sequential computer, but it can exploit instruction-level parallelism.
- 2- Single instruction stream, multiple data streams (SIMD)—The same instruction is executed by multiple processors using different data streams. SIMD computers exploit data-level parallelism by applying the same operations to multiple items of data in parallel. Each processor has its own data memory (hence the MD of SIMD), but there is a single instruction memory and control processor, which fetches and dispatches instructions.
- 3- Multiple instruction streams, single data stream (MISD)—No commercial multiprocessor of this type has been built to date, but it rounds out this simple classification.
- 4- Multiple instruction streams, multiple data streams (MIMD)—Each processor fetches its own instructions and operates on its own data, and it targets task-level parallelism. In general, MIMD is more flexible than SIMD and thus more generally applicable, but it is inherently more expensive than SIMD. For example, MIMD computers can also exploit data-level parallelism, although the overhead is likely to be higher than would be seen in an SIMD computer. This overhead means that grain size must be sufficiently large to exploit the parallelism efficiently.

4. Defining Computer Architecture

The task the computer designer faces is a complex one: **Determine** what attributes are important for a new computer, then design a computer to ¹maximize performance and ²energy efficiency while ³staying within cost, ⁴power, and ⁵availability constraints. This task has many aspects, including ¹instruction set design, ²functional organization, ³logic design, and ⁴implementation (integrated circuit design, packaging, power, and cooling).

5. Instruction Set Architecture: The Myopic View of Computer Architecture

We use the term instruction set architecture (ISA) to refer to the actual programmer visible instruction set in this lectures. The ISA serves as the boundary between the software and hardware.

- 1- Class of ISA—Nearly all ISAs today are classified as general-purpose register architectures, where the operands are either registers or memory locations. The 80x86 has 16 general-purpose registers and 16 that can hold floating point data, while MIPS has 32 general-purpose and 32 floating-point registers. The two popular versions of this class are register-memory ISAs, such as the 80x86, which can access memory as part of many instructions, and load-store ISAs, such as ARM and MIPS, which can access memory only with load or store instructions. All recent ISAs are load-store.
- 2- Memory addressing—Virtually all desktop and server computers, including the 80x86, ARM, and MIPS, use byte addressing to access memory operands. Some architectures, like ARM and MIPS, require that objects must be aligned. An access to an object of size s bytes at byte address A is aligned if $A \bmod s = 0$. The 80x86 does not require alignment, but accesses are generally faster if operands are aligned.
- 3- Addressing modes—In addition to specifying registers and constant operands, addressing modes specify the address of a memory object. MIPS addressing modes are Register, Immediate (for constants), and Displacement, where a constant offset is added to a register to form the memory address. The 80x86 supports those three plus three variations of displacement: no register (absolute), two registers (based indexed with displacement), and two registers where one register is multiplied by the size of the operand in bytes (based with scaled index and displacement). It has more like the last three, minus the

displacement field, plus register indirect, indexed, and based with scaled index. ARM has the three MIPS addressing modes plus PC-relative addressing, the sum of two registers, and the sum of two registers where one register is multiplied by the size of the operand in bytes. It also has autoincrement and autodecrement addressing, where the calculated address replaces the contents of one of the registers used in forming the address.

- 4- Types and sizes of operands—Like most ISAs, 80x86, ARM, and MIPS support operand sizes of 8-bit (ASCII character), 16-bit (Unicode character or half word), 32-bit (integer or word), 64-bit (double word or long integer), and IEEE 754 floating point in 32-bit (single precision) and 64-bit (double precision). The 80x86 also supports 80-bit floating point (extended double precision).
- 5- Operations—The general categories of operations are data transfer, arithmetic logical, control (discussed next), and floating point. MIPS is a simple and easy-to-pipeline instruction set architecture, and it is representative of the RISC architectures being used in 2011.
- 6- Control flow instructions—Virtually all ISAs, including these three, support conditional branches, unconditional jumps, procedure calls, and returns. All three use PC-relative addressing, where the branch address is specified by an address field that is added to the PC. There are some small differences. MIPS conditional branches (BE, BNE, etc.) test the contents of registers, while the 80x86 and ARM branches test condition code bits set as side effects of arithmetic/logic operations. The ARM and MIPS procedure call places the return address in a register, while the 80x86 call (CALLF) places the return address on a stack in memory.
- 7- Encoding an ISA—There are two basic choices on encoding: fixed length and variable length. All ARM and MIPS instructions are 32 bits long, which simplifies instruction decoding. The 80x86 encoding is variable length, ranging from 1 to 18 bytes. Variable length instructions can take less space than fixed-length instructions, so a program compiled for the 80x86 is usually smaller than the same program compiled for MIPS.

6. Organization and Hardware

The implementation of a computer has two components: organization and hardware. The term organization includes the high-level aspects of a computer's design, such as the memory system, the memory interconnect, and the design of the internal processor or CPU (central processing unit—where arithmetic, logic, branching, and data transfer are implemented). The term microarchitecture is also used instead of organization. For example, two processors with the same instruction set architectures but different organizations are the AMD Opteron and the Intel Core i7. Both processors implement the x86 instruction set, but they have very different pipeline and cache organizations.

The switch to multiple processors per microprocessor led to the term core to also be used for processor. Instead of saying multiprocessor microprocessor, the term multicore has caught on.

Hardware refers to the specifics of a computer, including the detailed logic design and the packaging technology of the computer. Often a line of computers contains computers with identical instruction set architectures and nearly identical organizations, but they differ in the detailed hardware implementation. For example, the Intel Core i7 and the Intel Xeon 7560 are nearly identical but offer different clock rates and different memory systems, making the Xeon 7560 more effective for server computers.

7. Trends in Technology

If an instruction set architecture is to be successful, it must be designed to survive rapid changes in computer technology. After all, a successful new instruction set architecture may last decades—for example, the core of the IBM mainframe has been in use for nearly 50 years. An architect must plan for technology changes that can increase the lifetime of a successful computer.

Five implementation technologies, which change at a dramatic pace, are critical to modern implementations:

- Integrated circuit logic technology—Transistor density increases by about 35% per year.
- Semiconductor DRAM (dynamic random-access memory)—Now that most DRAM chips are primarily shipped in DIMM (Dual In-line Memory Modules) modules, it is harder to track chip capacity, as DRAM manufacturers typically offer several capacity products at the same time to match DIMM capacity. Capacity per DRAM chip has increased by about 25% to 40% per year recently, doubling roughly every two to three years.

- Semiconductor Flash (electrically erasable programmable read-only memory) This nonvolatile semiconductor memory is the standard storage device in PMDs, and its rapidly increasing popularity has fueled its rapid growth rate in capacity. Capacity per Flash chip has increased by about 50% to 60% per year recently, doubling roughly every two years. In 2011, Flash memory is 15 to 20 times cheaper per bit than DRAM.
- Magnetic disk technology— Disks are 15 to 25 times cheaper per bit than Flash. Given the slowed growth rate of DRAM, disks are now 300 to 500 times cheaper per bit than DRAM.
- Network technology—Network performance depends both on the performance of switches and on the performance of the transmission system.

8. Trends in Performance

Bandwidth or Throughput is the total amount of work done in a given time, such as megabytes per second for a disk transfer. In contrast, latency or response time is the time between the start and the completion of an event, such as milliseconds for a disk access. Performance is the primary differentiator for microprocessors and networks.

Capacity is generally more important than performance for memory and disks, so capacity has improved most.

9. Trends in Power and Energy in Integrated Circuits

Today, power is the biggest challenge facing the computer designer for nearly every class of computer. First, power must be brought in and distributed around the chip, and modern microprocessors use hundreds of pins and multiple interconnect layers just for power and ground. Second, power is dissipated as heat and must be removed.

10. Trends in Cost

Although costs tend to be less important in some computer designs—specifically supercomputers—cost-sensitive designs are of growing significance. Indeed, in the past 30 years, the use of technology improvements to lower cost, as well as increase performance, has been a major theme in the computer industry.

11. Principle of Locality

Important fundamental observations have come from properties of programs. The most important program property that we regularly exploit is the principle of locality: Programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code. An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past. The principle of locality also applies to data accesses, though not as strongly as to code accesses.

Two different types of locality have been observed. **Temporal locality** states that recently accessed items are likely to be accessed in the near future. **Spatial locality** says that items whose addresses are near one another tend to be referenced close together in time.

12. RISC and CISC

Reduced instruction set computing (RISC) is a CPU design strategy based on the insight that a simplified instruction set provides higher performance when combined with a microprocessor architecture capable of executing those instructions using fewer microprocessor cycles per instruction.

Complex instruction set computing (CISC) is a processor design where single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions.

CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program.

Examples of CISC processors are:

- IBM/360(excluding the 'scientific' Model 44)
- VAX
- PDP11
- Motorola 68000 family
- Intel x86 architecture based processors.

Examples of RISC processors are:

- Apple iPods (custom ARM7TDMI SoC)
- Apple iPhone (Samsung ARM1176JZF)
- Palm and PocketPC PDAs and smartphones (Intel XScale family, Samsung SC32442, ARM9)
- Nintendo Game Boy Advance (ARM7)
- Nintendo DS (ARM7, ARM9)
- Sony Network Walkman (Sony in- house ARM based chip)
- Some Nokia and Sony Ericsson mobile phones

Basic Processing Unit

1. Some Fundamental Concepts

To execute a program, the processor fetches one instruction at a time and performs the operation specified. Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered.

The processor keeps track of the address of the memory location containing the next instruction to be fetched using the program counter (PC). After fetching an instruction, the contents of the PC are updated to point to the next instruction in the sequence. Another key register in the processor is the instruction register, IR which contains the instruction that has to be executed.

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

1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are an instruction to be executed. Hence they are loaded into the IR.

This can be symbolically written as $IR \leftarrow [[PC]]$

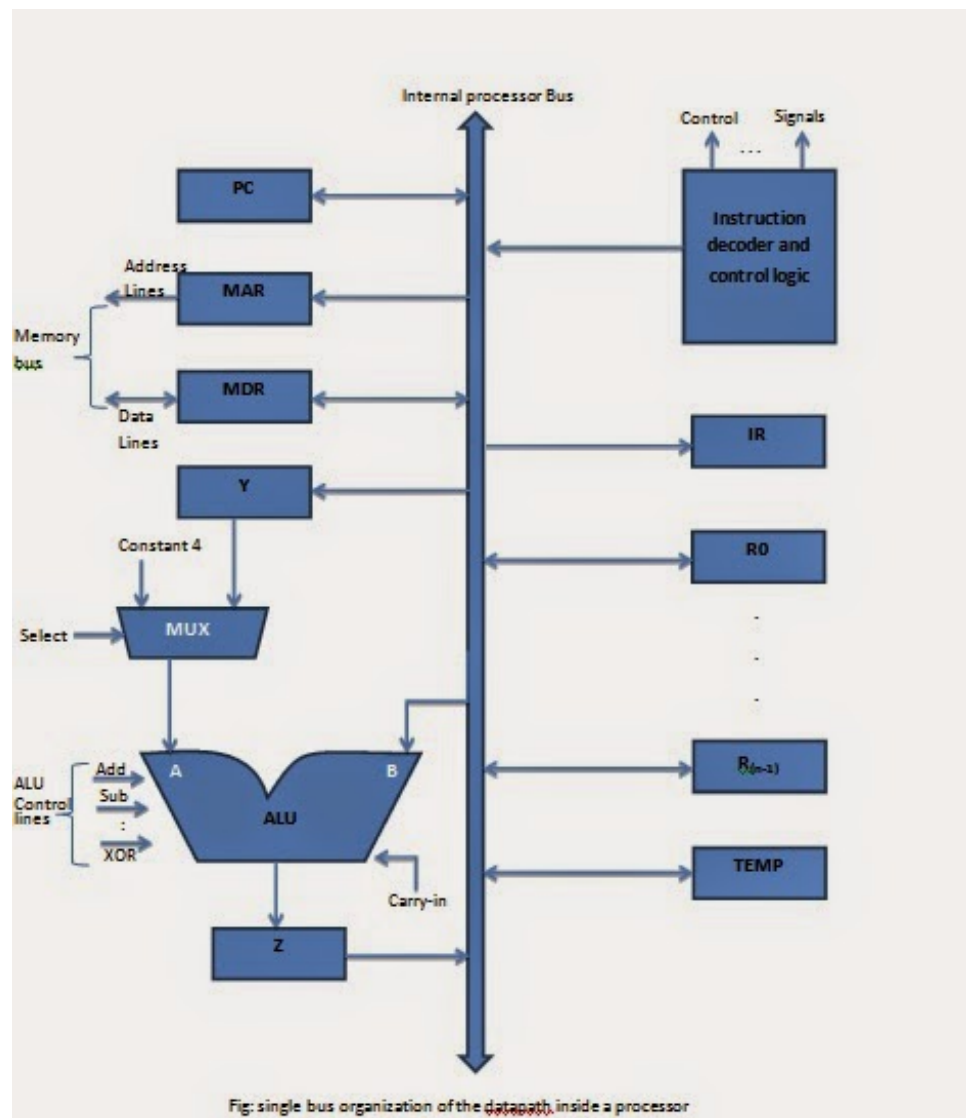
2. Assuming that the memory is byte addressable, increment the contents of the PC by 4, to point to the next instruction

$PC \leftarrow [PC] + 4$

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

In the instruction execution Step 1 and 2 are referred to as Instruction fetch phase and Step 3 as instruction execution phase.

Figure below shows single bus organization in which the arithmetic and logic unit (ALU) and all registers are interconnected (single bus organization).



The data and address lines of the external memory bus are connected to the internal processor bus via MDR and MAR.

Register MDR has two inputs and two outputs. Data may be loaded into MDR either from the memory bus or from the internal processor bus. The data stored in MDR may be placed on either bus.

The input of MAR is connected to the internal bus, and its output is connected to the external bus.

The control lines of the memory bus are connected to the instruction decoder and control logic block. This unit is responsible for issuing the signals that control the operation of all the units inside the processor and for interacting with the memory bus.

- The number and use of the processor registers R0 through R(n-1) vary from one processor to another.
- Registers y, z and TEMP are used by the processor for temporary storage during execution of some instructions.
- The multiplexer MUX selects either the output of register y or a constant value 4 that will be provided as input A of the ALU
- As instruction execution starts, data are transferred from one register to another, often passing through the ALU to perform some arithmetic or logic operation.

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

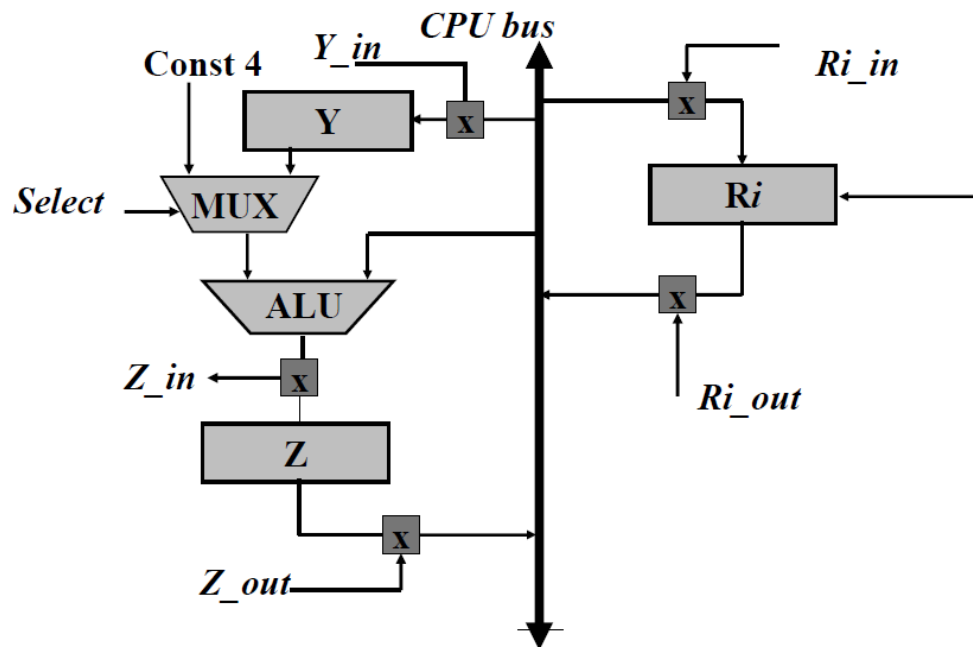
1. Transfer a word of data from one processor register to another or to the ALU
2. Perform arithmetic or a logic operation and store the result in a processor register.
3. Fetch the contents of a given memory location and load them into a processor register
4. Store a word of data from a processor register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

We will explain each of the above in details:

1. Suppose that we want to transfer the contents of register R1 to register R4. This done by the following steps: $R4 \leftarrow [R1]$
 - Enable the output of register R1 by setting R1out to 1. This places the contents of R1 on the processor bus.
 - Enable the input of register R4 by setting R4in to 1. This loads data from the processor bus into register R4.
 - This can be written as :

R1out, R4in

All operations and data transfers within the processor take place within time periods defined by the Processor Clock.



Register Gating showing the in and out gates for each

2. Performing an Arithmetic or Logic Operation

Arithmetic and logic unit has no internal storage. It performs arithmetic and logic operations on the two operands applied to its A and B inputs.

As shown in the fig above; one of the operands is the output of the multiplexer MUX and the other operand is obtained directly from the bus. The result produced by the ALU is stored temporarily in register Z.

Therefore, a sequence of operations to add the contents of register R1 to those of register R2 and store the result in register R3 is as follows:

$$R3 \leftarrow [R1] + [R2]$$

1. R1out, Yin
2. R2out, Select Y, ADD, Zin
3. Zout, R3in

Signals whose names are given in any step are activated for the duration of the clock cycle corresponding to that step. All other signals are inactive.

Multiplexer: is a device that selects one digital/analog input out of several and that is transferred over a single line.

3. Fetching a Word from Memory

To fetch a word of information from memory, the processor has to specify the address of the memory location where information is stored and request a Read operation. The information may be an instruction in a program or an operand specified by an instruction.

The processor transfers the required address to the MAR, whose output is connected to the address lines of the memory bus. At the same time, the processor uses the control lines of the memory bus to indicate Read operation. When requested data are received from the memory they are stored in register MDR. Then they are transferred to other registers in the processor.

Memory Function Completed (MFC) control signal is used to indicate the processor that the requested Read/Write operation has been completed.

Mov R2, [R1]

1. $MAR \leftarrow [R1]$
2. Start a Read operation on the memory bus
3. Wait for the MFC response from the memory
4. Load MDR from the memory bus
5. $R2 \leftarrow [MDR]$

The memory read operation requires three steps, which can be described by the activated signals as follows:

1. R1out, MARin, Read
2. MDRinE, WMFC
3. MDRout, R2in

WMFC is the control signal that causes the processor's control circuitry to wait for the arrival of the MFC signal.

4. Storing a Word in Memory

While writing a word into a memory location, the desired address is loaded into MAR. The data to be written are loaded into MDR, and a Write command is issued.

Executing the instruction `Mov [R1], R2` as follows:

- 1· R1out, MARin
- 2· R2out, MDRin, Write
- 3· WMFC

Write control signal causes the memory bus interface hardware to issue a write command on the memory bus. The processor remains in step 3 until the memory operation is completed and an MFC response is received.

2. Execution of a Complete Instruction

Consider the instruction **Add R1, [R3]** which adds the contents of a memory location pointed to by R3 to register R1. Execution of this instruction follows actions as below:

1. Fetch the instruction
2. Fetch the first operand (the contents of the memory location pointed to by R3)
3. Perform the addition
4. Load the result into R1

And can be written as:

Step	Action
1	<u>PC</u> _{out} , <u>MAR</u> _{in} , Read, Select4, Add, <u>Z</u> _{in}
2	<u>Z</u> _{out} , <u>PC</u> _{in} , <u>Y</u> _{in} , WMFC
3	<u>MDR</u> _{out} , <u>IR</u> _{in}
4	<u>R3</u> _{out} , <u>MAR</u> _{in} , Read
5	<u>R1</u> _{out} , <u>Y</u> _{in} , WMFC
6	<u>MDR</u> _{out} , SelectY, Add, <u>Z</u> _{in}
7	<u>Z</u> _{out} , <u>R1</u> _{in} , End

Illustration:

- 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.
- The Select Signal is set to Select4 which causes the multiplexer MUX to select constant 4. This value is added to the operand at input B, which is the contents of the PC and the result is stored in register Z. The updated value is moved from register Z back into the PC during Step 2 while waiting for the memory to respond.
- In Step3: the word fetched from the memory is loaded into the IR.
- Steps 1 to 3 are of the instruction fetch phase
- Steps 4 to 7 are of the instruction execution phase
- In step 4 the contents of register R3 are transferred to the MAR and a memory read operation is initiated.
- In step 5 the contents of R1 are transferred to register Y.
- When the Read operation is completed, the memory operand is available in Register MDR, and the addition operation is performed in step 6.
- The contents of MDR are to the input B of the ALU over bus, and register Y is selected as a input A of the ALU. After performing addition, Sum is stored in register Z.
- In step 7 contents of the register Z transferred to register R1.
- Note: updated contents of PC are stored in register Y, because the PC value is needed to compute the branch target address in the case of Branch Instructions.

Branch Instructions

A Branch instruction replaces the contents of the PC with the branch target address. This address is obtained by adding offset X, to the updated value of the PC.

Branch x

Step	Action
1	PC _{out} , MAR _{in} , Read, Select4, Add, Z _{in}
2	Z _{out} , PC _{in} , Y _{in} , WMFC
3	MDR _{out} , IR _{in}
4	Offset field of IR _{out} , Add, Z _{in}
5	Z _{out} , PC _{in} , End

Figure: Control Sequence for an unconditional Branch instruction

The figure above showing control sequence that implements an unconditional branch instruction.

- In Step1 PC contents are transferred to MAR, and Read operation is initiated. Addition is done by adding the contents of PC with constant 4 the SUM is stored in register Z.
- In Step 2 content of Z (updated PC value) is transferred to PC and Register Y.
In step 3 after the memory operation completion the MDR contents are transferred to IR.
- Steps 1 to 3 is of instruction fetch phase.
- In step 4 the offset X is moved onto the bus that is given as input B of ALU. The updated PC value is available in register Y that is given as input A of ALU, and then addition operation is performed. The result sends to register Z.
In step 5 the result which is the branch target address is loaded into PC.

For example, for a Branch-on-negative (Branch < 0) instruction, step 4 in figure is replaced with

4. If N=0 then End , if N=1 then offset-field-of-IRout, Add, Zin,.

3. Microprogrammed Control

In Microprogrammed control , control signals are generated by a program similar to machine language programs.

micro-instruction	PC_in	MAR_in	Addr_in	Z_in	...
1	0	1	00	1	...
2	1	0	00	0	...
3	0	0	01	0	...
..					
..					

Example shows the contents of microprogram memory

A Control Word(CW) is a word whose individual bits represents the various control signals.

Microroutine: a sequence of a machine instruction is called microroutine for that instruction.

The Individual control words in the microroutine are referred to as microinstruction.

The microroutines for all instructions in the instruction set of a computer are stored in a special memory called the control store.

The Control units can generate the control signals for any instructions by sequentially reading CWs of the corresponding microroutine from the control store.

To read the control words sequentially from the control store, a micro program counter(μ PC) is used.

Every time a new instruction is loaded into the IR, the output of the block labeled as “Starting address generator” is loaded into the μ PC.

The μ PC is then automatically incremented by the clock, causing successive microinstructions to be read from the control store. Hence the control signals are delivered to various parts of the processor in the correct sequence.

Conditional Branch Microinstructions in addition to the branch address specifies which of the external inputs, condition codes or bits of the instruction register should be checked as a condition for branching to takes place.

In this control unit the μ PC is incremented every time a new microinstruction is fetched from the microprogram memory except in the following situation:

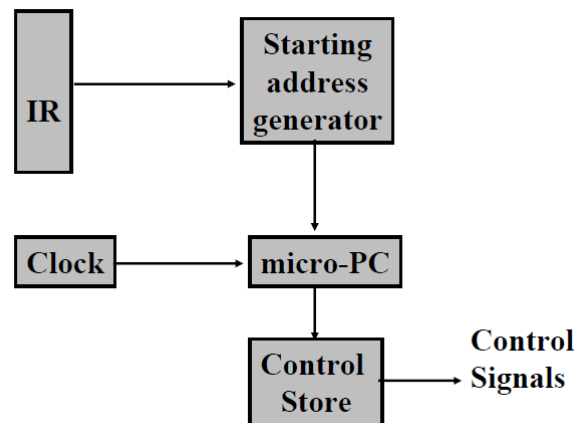
1. When a new instruction is loaded into the IR, the μ PC is loaded with the starting address of the microroutine for that instruction.
2. When a branch microinstruction is encountered and the branch condition is satisfied, the μ PC is loaded with the branch address.
3. When an End microinstruction is encountered, the μ PC is loaded with the address of the first CW in the microroutine for the instruction fetch cycle.

Sequencing of Control Signals

1. Microprogrammed Control Unit

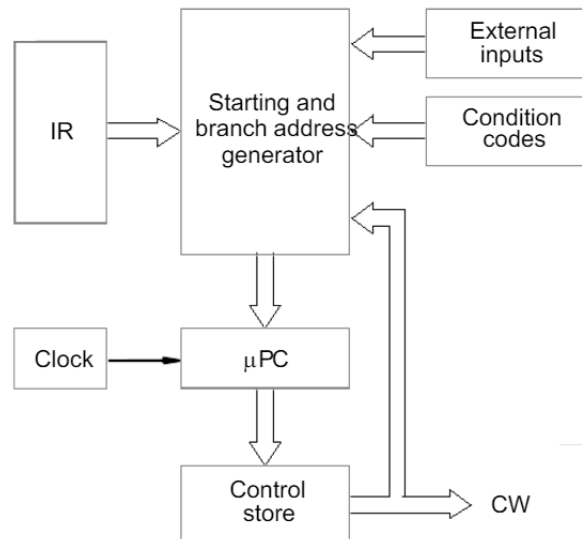
The microprogram requires sequential execution of microinstructions, except for the branch at the end of the fetch phase. If each machine instructions is implemented by a microroutine. In micro control structure μ PC governs the sequencing.

A microroutine is entered by decoding the machine instruction into a starting address that is loaded into the μ PC. Branching microinstructions specifies the branch address that transfers control to some other part.



Microprogramed control unit- basic organization

The above figure can be modified to enable conditional branch as follows:



Organization of the control unit to allow conditional branching in the microprogram

The **main advantages** of the microprogrammed control are the fact that once the hardware configuration is established; there should be no need for further hardware or wiring changes. If we want to establish a different control sequence for the system, all we need to do is specify different set microinstructions for control memory. The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory, but on the other hand, this technique is slower than Hardware Control Unit in since the execution of a single instruction requires many fetches from control store, however, this control unit is suitable for CISC design.

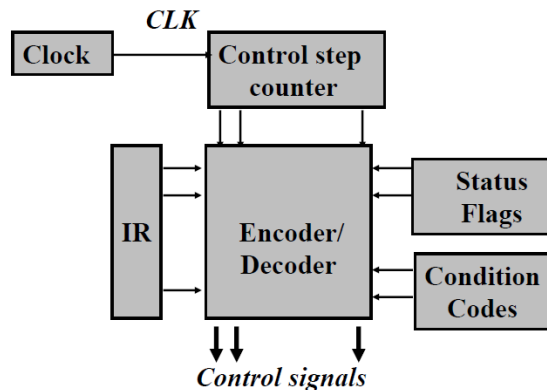
2. Hardwired Control Unit

In this case, the control signals are not stored anywhere in the control unit and these signals are generated instantly at the time of instruction fetch, this method depends mainly on hardware gates (oring, anding, not) to generate these control signals (consecutive control words).

The **advantage** of hardwired control is that is very fast. The **disadvantage** is that the instruction set and the control logic are directly tied together by special circuits that are complex and difficult to design or modify. If someone designs a hardwired computer and later decides to extend the instruction set, the physical components in the computer must be changed. This is prohibitively expensive, because not

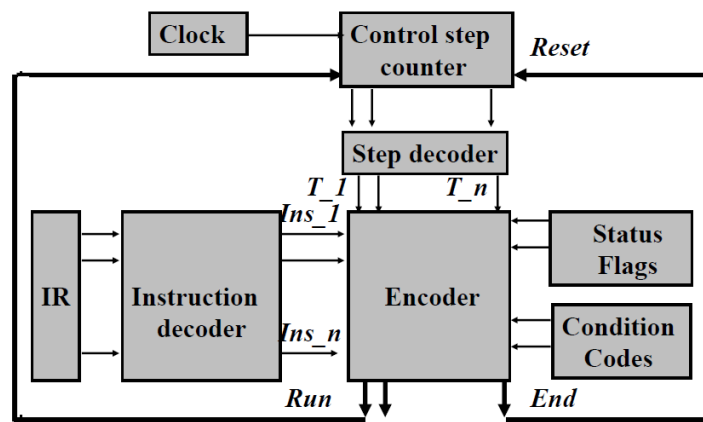
only must new chips be fabricated but also the old ones must be located and replaced, the following figures show the organization of this unit.

Control Unit Organization



Basic Organization of H/W control unit

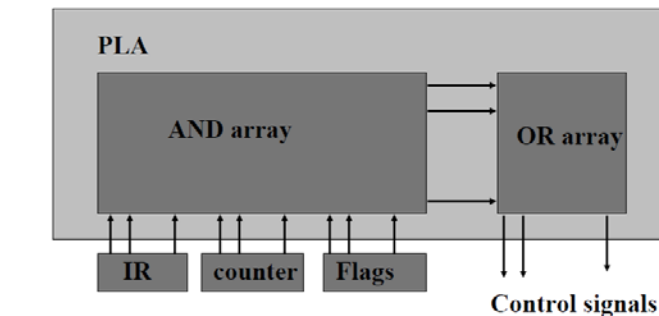
Separating decoding/encoding



Basic Organization of H/W control unit with a separate decoder/encoder

A programmable logic array (PLA) is a kind of programmable logic device used to implement combinational logic circuits. The PLA has a set of programmable AND gate planes, which link to a set of programmable OR gate planes, which can then be conditionally complemented to produce an output. The number of AND gates in the programmable AND array are usually much less and the number of inputs of each of the OR gates equal to the number of AND gates.

PLA's



PLA Schematic

3. Dynamic Microprogramming

A more advanced development known as dynamic microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. Control units that use dynamic microprogramming employ a writable control memory; this type of memory can be used for writing (to change the microprogram) but is used mostly for reading.

Memory Cycle Time (MCT) (time memory : t_m)

It is the time that is measured in nanoseconds, the time between one Ram access of time when the next Random Access Memory **RAM** access starts.

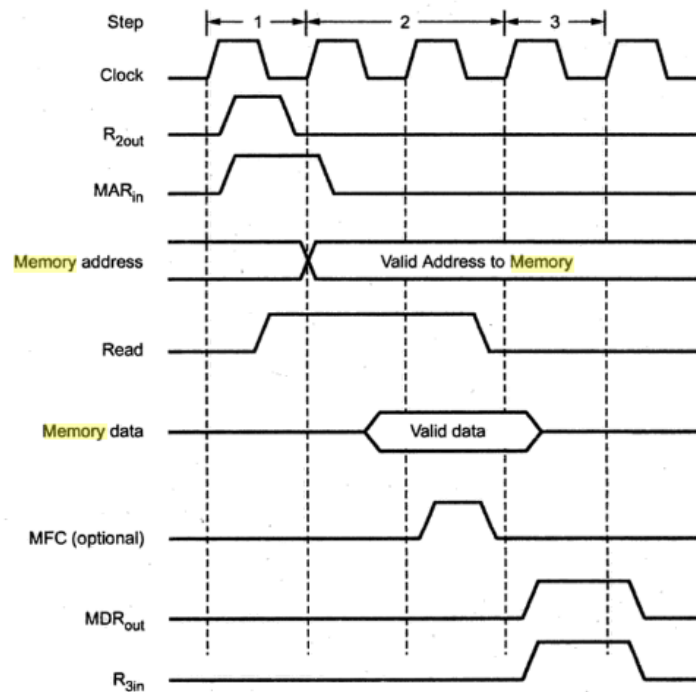
In other words, it is the minimum time elapsed between two successive read/write operations.

Memory access Time (MAT)

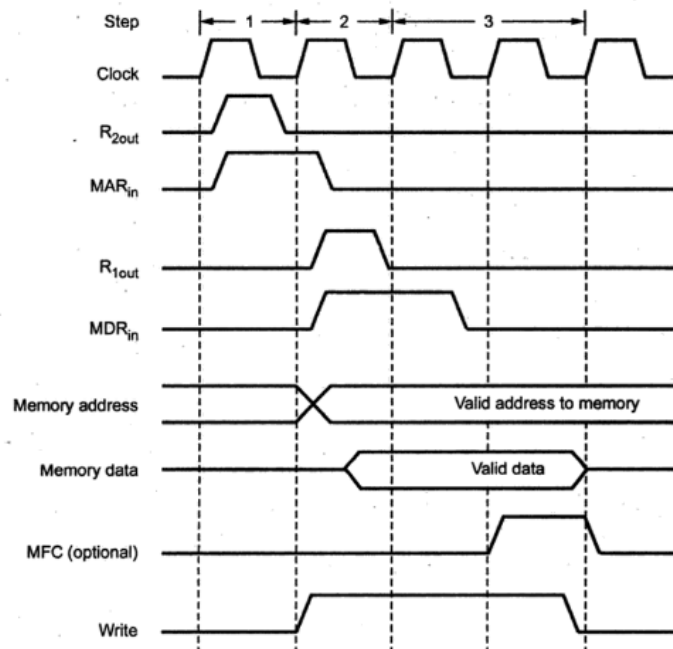
Access time is the amount of time it takes the processor to read data, instructions, and information from memory. A computer's access time directly affects how fast the computer processes data. Accessing data in memory can be more than 200,000 times faster than accessing data on a hard disk because of the mechanical motion of the hard disk.

In other word, it is the minimum time elapsed between the read/write signal and the MFC signal.

Drawing of the Read and Write Timing Signals



Timing diagram for **Mov R3, [R2]** instruction (memory read operation)

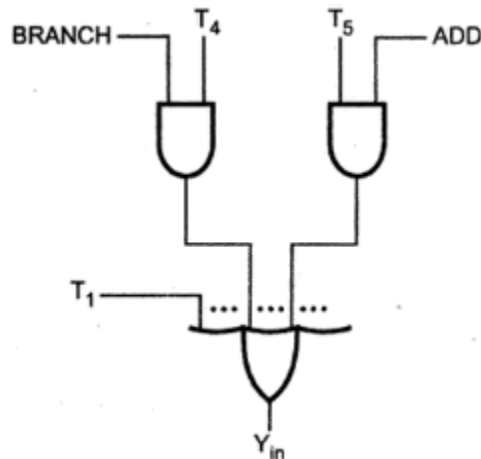


Timing diagram for **Mov [R2], R1** Instruction (memory write operation)

Hardware Control Unit- Signals Generating

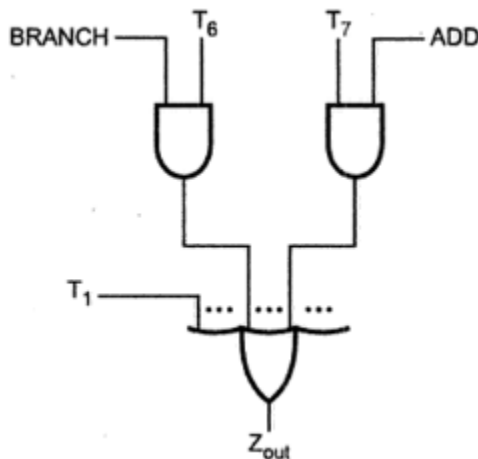
Let us see how the encoder generated signal for single bus processor organization shown in following figure, the encoder circuit implements the following logic equation to generate Y_{in} :

$$Y_{in} = T_1 + T_5 \cdot ADD + T_4 \cdot BRANCH + \dots$$



Generating of Yin control signal

$$Z_{out} = T_2 + T_7 \cdot ADD + T_6 \cdot BRANCH + \dots$$



Generating of Zout control signal

End signal

Other example:

$$\text{End} = T_7 \cdot \text{ADD} + T_6 \cdot \text{BR} + (T_6 \cdot N + T_4 \cdot \overline{N}) \cdot \text{BRN}$$

How can be sketched?

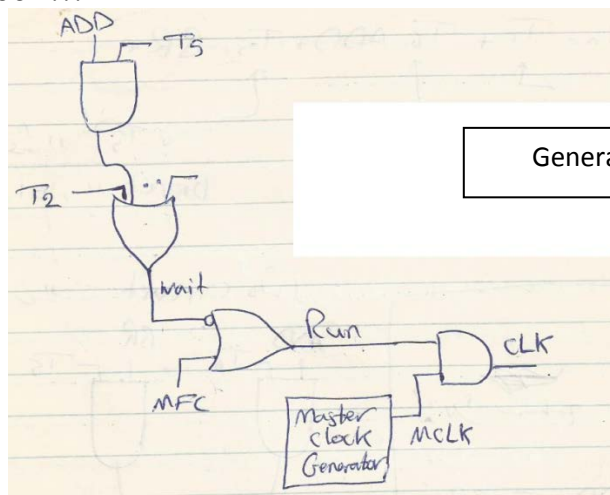
Performance

- Performance is dependent on:
 - Power of instructions
 - Cycle time
 - Number of cycles per instruction
- Performance improvement by:
 - Multiple datapaths
 - Instruction prefetching and pipelining
 - Caches

Run Signal

This signal is depends mainly on the WMFC signal, in which if there is no wait on memory, the run is 1 and the operation is normally going on, but, however, if there is any wait on memory, run became 0 which cause the CPU clocks became inactive for the entire time until the operation required form memory is completed.

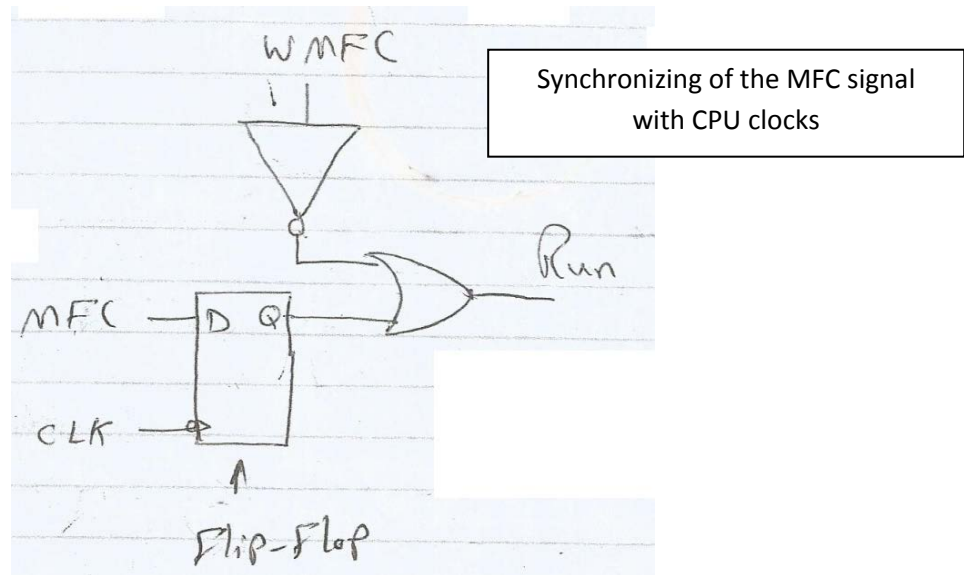
WMFC = $T_2 + T_5 \cdot \text{Add} + \dots$



Generating of the Run signal

The problem of the previous figure is that the MFC signal is independent of the CPU clocks and can arrive at any point during the CPU clock, the proper work is to start the operation at the positive edge of the clock since each operation required at least one clock to complete, if the operation started behind the positive edge (clock starting point), the remaining time of the clock will not be enough for any operation and error will happened.

To solve this problem, flip-flop is needed to store the MFC signal and trigger it when new CPU clock is emitted.



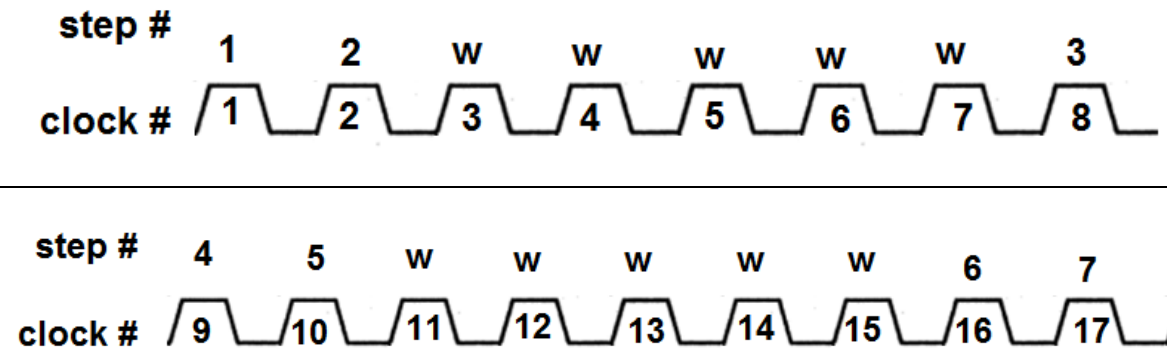
Microprogram Total Time and Clocks

The microprogram consumes some CPU time (clocks) depends on the length of that microprogram and the action performed.

Consider the CPU clocks (t_{cpu}) is 10 nsec, MAT=70 nsec. Find the total execution time for code in page 19, draw the clocks.

Steps	Time taken	Wait time
1, 2	70 nsec	50 nsec
3	10 nsec	0 nsec
4, 5	70 nsec	50 nsec
6	10 nsec	0 nsec
7	10 nsec	0 nsec
total	170 nsec	100 nsec

Number of clocks = $170/10=17$ clocks, 10 of them are **wait**.



DRAM and SRAM

DRAM (Dynamic RAM) requires the data to be refreshed periodically in order to retain the data.

SRAM (Static RAM) does not need to be refreshed as the transistors inside would continue to hold the data as long as the power supply is not cut off.

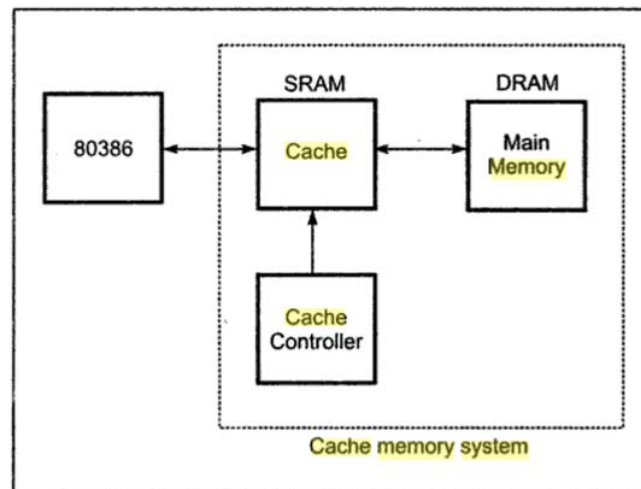
A DRAM module only needs a transistor and a capacitor for every bit of data where SRAM needs 6 transistors. Because the number of transistors in a memory module determine its capacity, a DRAM module can have almost 6 times more capacity with a similar transistor count to an SRAM module. This ultimately boils down to price, which is what most buyers are really concerned with.

Some Differences

1. SRAM is static while DRAM is dynamic.
2. SRAM is faster compared to DRAM.
3. SRAM consumes less power than DRAM.
4. SRAM uses more transistors per bit of memory compared to DRAM.
5. SRAM is more expensive than DRAM.
6. Cheaper DRAM is used in main memory while SRAM is commonly used in cache memory.

Cache Memory

A cache memory includes a small amount of fast memory (SRAM) and a large amount of slow memory (DRAM) as shown below.



Cache Memory System

In a computer system the program which is to be executed is loaded in the main memory (DRAM). Processor then fetches the code and data from the main memory to execute the program. The DRAMs which form the main memory are slower devices. So it is necessary to insert wait states in memory read/write cycles. This reduces the speed of execution. To speed up the process, high speed memories such as SRAMs must be used. But considering the cost and space required for SRAMs, it is not desirable to use SRAMs to form the main memory. The solution for this problem is come out with the fact that most of the microcomputer programs work with only small sections of code and data at a particular time. 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. This is accomplished by loading the active part of code and data from main memory to cache memory. The cache controller looks after this swapping between main memory and cache memory with the help of DMA controller. If processor finds that the addressed code or data is not available in cache, then processor accesses that code or data from the main memory (DRAM). The percentage of accesses where the processor finds the

code or data word it needs in the cache memory is called the hit rate. The hit rate is normally greater than 90 percent.

$$\text{Hit rate} = \frac{\text{Number of hits}}{\text{Number of read / write bus cycles}} \times 100\%$$

Ex. : The application program in a computer system with cache uses 1400 instruction acquisition bus cycle from cache memory and 100 from main memory. What is the hit rate? If the cache memory operates with zero wait state and the main memory bus cycles use three wait states, what is the average number of wait states experienced during the program execution?

Sol. :

$$\text{Hit rate} = \frac{1400}{1400 + 100} \times 100 = 93.3333\%$$

$$\text{Total wait states} = 1400 \times 0 + 100 \times 3 = 300$$

$$\text{Average wait states} = \frac{\text{Total wait states}}{\text{No. of memory bus cycles}} = \frac{300}{1500} = 0.2$$

While the miss rate is the opposite of the hit rate.

Program locality

The main principle of the cache memory is the prediction of the memory location for the next access, which is called program locality. Program locality enables cache controller to get block of memory instead of getting just single instruction.

This principal may not work properly when program executes jump and call instructions.

Locality of Reference

We know that program may contain a simple loop, nested loops, or a few procedures that repeatedly call each other. The point is that many instructions in localized area of the program are executed repeatedly during some time period, and the remainder of the program is accessed relatively infrequently. This is referred to as **locality of reference**. It manifests itself in two ways : temporal and spatial. The temporal means that a recently executed instruction is likely to be executed again very soon. The spatial means that instructions stored near by to the recently executed instruction are also likely to be executed soon.

The temporal aspect of the locality of reference suggests that whenever an instruction or data is first needed, it should be brought into the cache and it should remain there until it is needed again. The spatial aspect suggests that instead of bringing just one instruction or data from the main memory to the cache, it is wise to bring several instructions and data items that reside at adjacent address as well. We use the term block to refer to a set of contiguous addresses of some size.

Block Fetch

Block fetch technique is used to increase the hit rate of cache. A block fetch can retrieve the data located before the requested byte (look behind) or data located after the requested byte (look ahead), or both. When CPU needs to access any byte from the block, entire block that contains the needed byte is copied from main memory into cache.

The size of the block is one of the most important parameters in the design of a cache memory system. The following section describes the pros and cons of small block size and large block size.

PROS and CONS :

1. If the block size is too small, the look ahead and look-behind are reduced, and therefore the hit rate is reduced.
2. Larger blocks reduce the number of blocks that fit into a cache. As the number of blocks decrease, block rewrites from main memory becomes more likely.
3. Due to large size of block the ratio of required data and useless data is less.
4. Bus size between the cache and the main memory increases with block size to accommodate larger data transfers between main memory and the cache, which increases the cost of cache memory system.

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 alone and large enough so that the overall average access time is close to that of the cache alone.

Mapping function : The cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory. Thus we have to use mapping functions to relate the main memory blocks and cache blocks. There are two mapping functions commonly used : direct mapping and associative mapping.

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

There are four most common replacement algorithms :

- Least-Recently Used (LRU)
- First-in-First-out (FIFO)
- Least-Frequently-Used (LFU)
- Random

Cache design change according to the choice of replacement algorithm.

Write Policy : It is also known as cache updating policy. In cache system, two copies of the same data can exist at a time, one in cache and one in main memory. If one copy is altered and other is not, two different sets of data become associated with the same address. To prevent this the cache system has updating systems such as : write through system, buffered write through system and write-back system. The choice of cache write policy also change the design of cache.

Block Size : It should be optimum for cache memory system.

Number of Caches : When on-chip cache is insufficient the secondary cache is used. The cache design changes as number of caches used in the system changes.

Mapping Function

Since the cache memory has limited size as compared to main memory, some kind of mapping is required to decide which block of main memory can reside in which block of cache memory, this called Cache Mapping Function.

The cache mapping function is responsible for all cache operations:

- Placement strategy: where to place an incoming block in the cache
- Replacement strategy: which block to replace upon a miss
- Read and write policy: how to handle reads and writes upon cache misses

Three different types of mapping functions:

- Associative Mapping Function
- Direct mapped Mapping Function
- Block-set associative Mapping Function

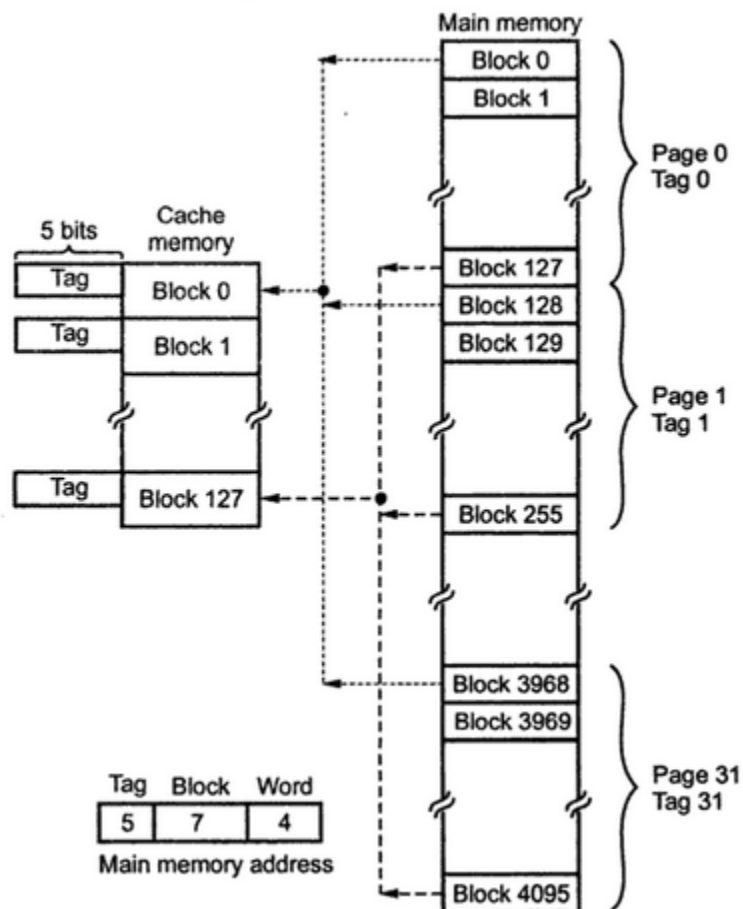
Cache Read and Write Policies

- Read and Write cache hit policies
 - Write through—updates both cache and MM upon each write, pros: (Important in multiprocessor systems), cons: (waste bus and memory bandwidth).
 - Write back—updates only cache. Updates MM only upon block removal.
Pros: (Reduces write traffic to memory), cons: (Takes longer to load new cache lines), (required dirty bit).
 - “Dirty bit” is set upon first write to indicate block must be written back.
- Read and Write cache miss policies

- Read miss—bring block in from MM, Either forward desired word as it is brought in, or, Wait until entire line is filled, then repeat the cache request.
- Write miss
- Write-allocate—bring block into cache, then update
- Write-no-allocate—write word to MM without bringing block into cache.

1. Direct Mapping Function

It is the simplest **mapping** technique. In this technique, each block from the main memory has only one possible location in the cache organisation. In this example, the block i of the main memory maps on to block i module 128 of the cache.



Direct Mapping Function

To implement such cache system, the address is divided into three fields, as shown **above**. The lower order 4-bits select one of the 16 words in a block. This field is known as **word field**. The second field known as **block field** is used to distinguish a block from other blocks. Its length is 7-bits since $2^7=128$. When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored. The third field is a **tag field**. It is used to store the

high-order 5-bits of memory address of the block. These 5-bit (tag bits) are used to identify which of the 32 blocks (pages) that are mapped into the cache.

Mapping process

- Use tag to see if a desired word is in cache
- If there is no match, the block containing the required word must first be read from the memory

Advantage

- simplest replacement algorithm

Disadvantage

- not flexible
- there is contention problem even when cache is not full

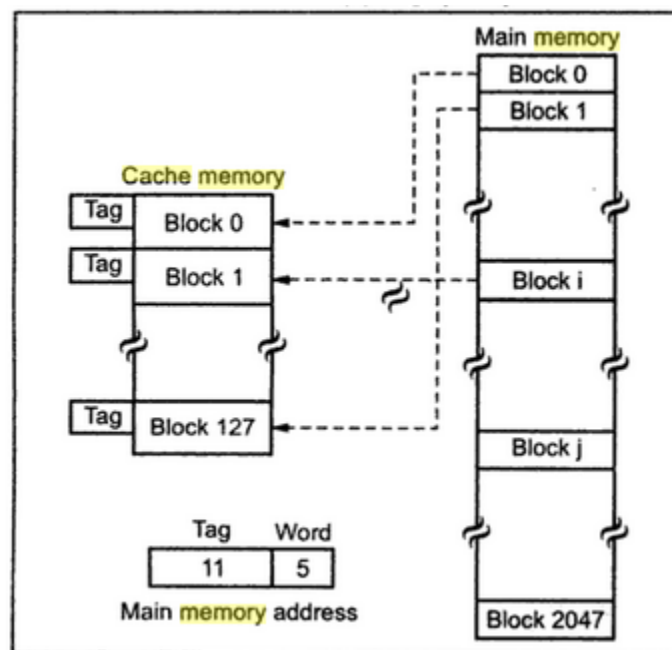
For example, block 0 and block 128 both take only block 0 of cache:

- $0 \text{ modulo } 128 = 0$
- $128 \text{ modulo } 128 = 0$
- *If both blocks 0 and 128 of the main memory are used a lot, it will be very slow*

2. Associative Mapping Function

In this technique, the main memory block can be placed into any cache block position, as there is no fix position.

The memory address has only two fields, word and tag, this technique is also referred to as fully associative.



Associative Mapping Function

For example, 4095 blocks $\rightarrow$ 4095 tag = $2^{12} \Rightarrow$ 12 bit tag

Advantage

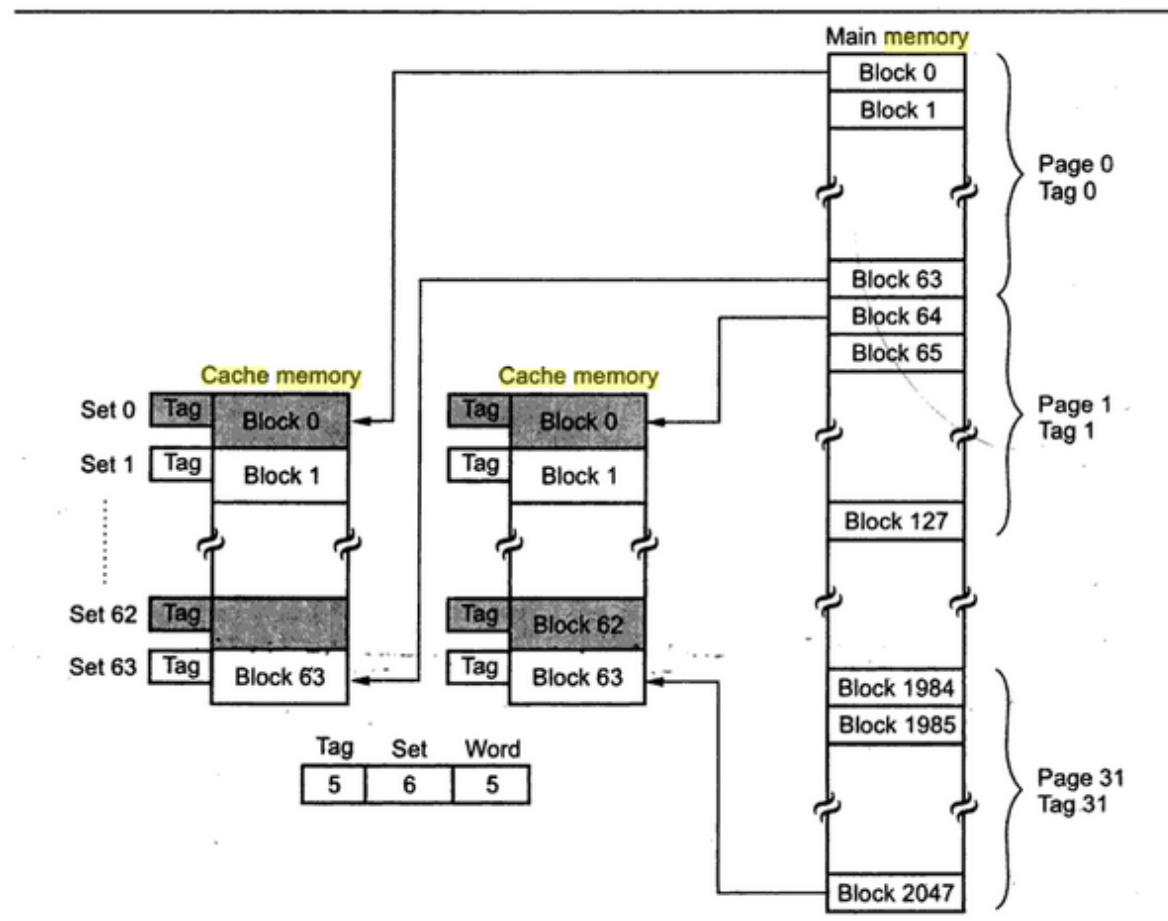
– Any empty block in cache can be used, flexible

Disadvantages

– Must check all tags to check for a hit, expensive

3. Set Associative Mapping Function

The set-associative mapping is a combination of both direct and associative mapping. It contains several groups of direct mapped blocks that operate as several direct mapped caches in parallel. A block of data from any page in the main memory can go into a particular block location of any direct-mapped cache. Hence the contention problem of the direct-mapped technique is eased by having a few choices for block placement. The required address comparisons depend on the number of direct mapped caches in the cache system. These comparisons are always less than the comparisons required in the fully-associative mapping.



Two-way set associative cache

To implement set-associative cache system, the address is divided into three fields, as shown . The 5-bit word field selects one of the 32 words in a block. The set field needs 6-bits to determine the desired block from 64 sets. However, there are now 31 pages. To identify which of the 32 blocks (pages) that are mapped into the particular set of cache, five tag bits are required.

No.	Direct Mapping	Associative Mapping	Set Associative Mapping
1.	Each block from the main memory has only one possible location in the cache.	A block of data from main memory can be placed into any cache block position.	A block of data from main memory can go into a particular block location of any direct-mapped cache.
2.	Needs only one comparison.	Needs comparison with all tag bits.	Needs number of comparisons equal to number of blocks per set.
3.	Cache hit ratio decreases if processor needs to access same memory location from two different pages of the main memory frequently.	Cache hit ratio has no effect if processor needs to access same memory location from two different pages of the main memory frequently.	The effect of reduction in cache hit ratio in case of frequent access to the two different pages of the main memory is reduced.
4.	Main memory address is divided into three fields : TAG, BLOCK and WORD.	Main memory address is divided into two fields : TAG and WORD.	Main memory address is divided into three fields : TAG, SET and WORD.
5.	Searching time is less.	Searching time is more.	Searching time increases with number of blocks per set.

Comparison between different placements techniques

Ex. 3.2 : Consider a cache consisting of 128 blocks of 16 words each, for a total of 2048 (2k) words, and assume that the main memory is addressable by a 16 bit address and it consists of 4k blocks. How many bits are there in each of the TAG, BLOCK/SET and word fields for different mapping techniques.

Sol. : We know that memory address is divided into three fields. We will now find the exact bits required for each field in different mapping techniques.

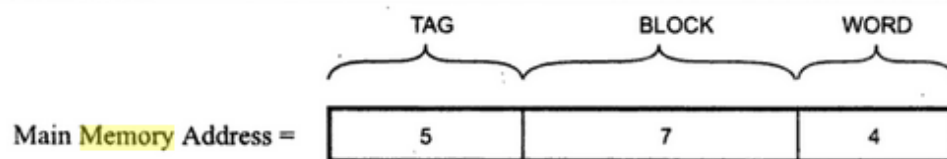
a) Direct mapping

Word Bits : We know that each block consists of 16 words. Therefore, to identify each word we must have ($2^4 = 16$) four bit reserved for it.

Block Bits : The cache memory consists of 128 blocks and using direct mapped technique block k of the main memory maps onto block k modulo 128 of the cache. It has one to one correspondence and requires unique address for each block. To address 128 block we require ($2^7 = 128$) seven bits.

Tag bits : The remaining 5 ($16 - 4 - 7$) address bits are tag bits which stores the higher address of the main memory.

The main memory address for direct mapping technique is divided as shown below :



b) Associative Mapping

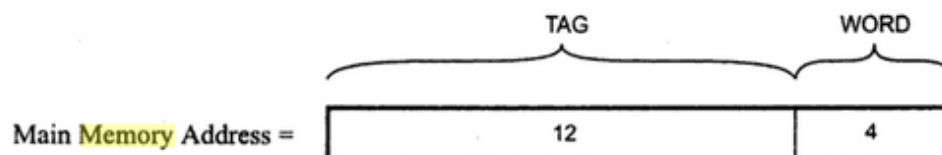
Word Bits : The word length will remain same i.e. 4 bits.

In the associative mapping technique each block in the main memory is identified by the tag bits and an address received from the CPU is compared with the tag bits of each block of the cache to see if the desired block is present. Therefore, this type of technique does not have block bits, but all remaining bits (except word bits) are reserved as tag bits.

Block Bits : 0

Tag Bits : To address each block in the main memory ($2^{12} = 4096$) 12 bits are required and therefore, there are 12 tag bits.

The main memory address for direct mapping technique is divided as shown below :



c) Set-associative mapping

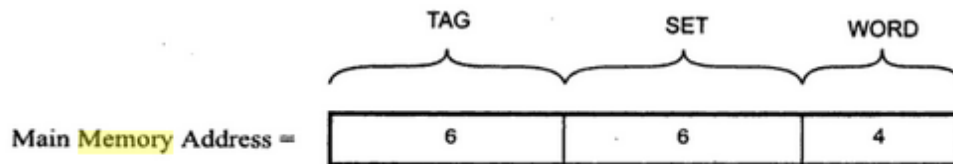
Let us assume that there is a 2-way set associative mapping. Here, **cache memory** is mapped with two blocks per set. **The set field of the address** determines which set of the **cache** might contain the desired block.

Word Bits : The word length will remain same i.e. 4 bits

Set bits : There are 64 sets (128/2). To identify each set ($2^6 = 64$) six bits are required.

Tag bits : The remaining 6 ($16 - 4 - 6$) address bits are the tag bits which stores higher address of the main memory.

The main memory address for 2-way set associative mapping technique is divided as shown below :



Ex. : A block set-associative **cache** consists of a total of 64 blocks divided into four-block sets. The main memory contain 4096 blocks, each consisting of 128 words.

- a) How many bits are there in a main memory address ?
- b) How many bits are there in each of the TAG, SET and WORD fields ?
- c) What is the size of cache memory ?

Sol. : a) The main memory contains 4096 blocks and each block consists of 128 words

$$\begin{aligned} \therefore \text{Total words in MM} &= 4096 \times 128 \\ &= 1048576 \end{aligned}$$

To address 1048576 words we require ($2^{20} = 1048576$) 20 bits.

b) Word bits : There are 128 words in each block. Therefore, to identify each word ($2^7 = 128$) 7 bits are required.

Set bits : There are 64 blocks and each set consists of 4 blocks. Therefore, there are 16 ($64/4$) sets. To identify each set ($2^4 = 16$) four bits are required.

Tag bits : We know that total address bits are 20 bits. Therefore tag bits are 9 ($20 - 7 - 4$).

The main memory address for 4-way set associative mapping is divided as shown below :

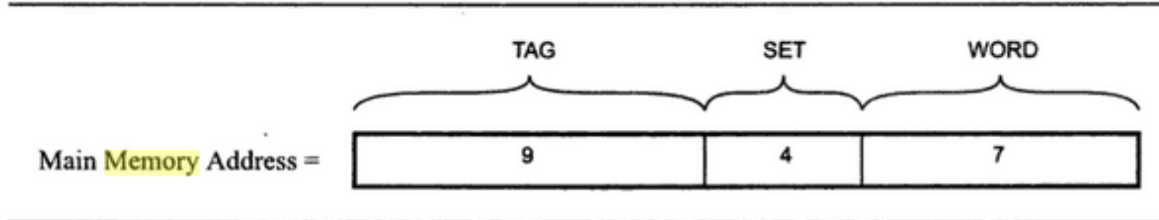


Fig. 3.27

c) The cache memory has 64 blocks of 128 words each. Therefore

$$\begin{aligned}\text{Total words in cache memory} &= 64 \times 128 \\ &= 8192\end{aligned}$$

Replacement Algorithms

When a new block is brought into the cache, one of the existing blocks must be replaced, by a new block. In case of direct mapping cache, we know that each block from the main memory has only one possible location in the cache, hence there is no choice. The previous data is replaced by the data from the same memory location from new page of the main memory. But for associative and set associative techniques, there is a choice of replacing existing block. The choice of replacement of the existing block should be such that the probability of accessing same block must be very less. The replacement algorithms do the task of selecting the existing block which must be replaced.

There are four most common replacement algorithms :

- Least-Recently Used (LRU)
- First-in-First-out (FIFO)
- Least-Frequently-Used (LFU)
- Random

Least-Recently-Used : In this technique, the block in the set which has been in the cache longest with no reference to it, is selected for the replacement. Since we assume that more-recently used memory locations are more likely to be referenced again. This technique can be easily implemented in the two-way set associative cache organisation.

First-In-First-Out : This technique uses same concept that stack implementation uses in the microprocessors. In this technique, the block which is first loaded in the cache amongst the present blocks in the cache is selected for the replacement.

Least Frequently Used : In this technique, the block in the set which has the fewest references is selected for the replacement.

Random : Here, there is no specific criteria for replacement of any block. The existing blocks are replaced randomly. Simulation studies have proved that random replacement algorithm provides only slightly inferior performance to algorithms just discussed.

Types of Cache Misses

- Three types
 - * Compulsory misses
 - Due to first-time access to a block
 - Also called cold-start misses or compulsory line (block) fills
 - Can be avoided by prefetch more
 - * Capacity misses
 - Induced due to cache capacity limitation
 - Can be avoided by increasing cache size
 - * Conflict misses
 - Due to conflicts caused by direct and set-associative mappings
 - Can be completely eliminated by fully associative mapping
 - Also called collision misses

Example:

Consider you have a cache memory of 4 blocks (**lines**) size, find the hit ratio for the following reference pattern = { 0, 4, 0, 8, 0, 8, 0, 4, 0, 4, 0, 4 }

For each of the following:

- 1- direct mapping
- 2- associative mapping
- 3- Set associative mapping, 2 blocks per set ?

Solution:

Direct-mapped cache state					
Block accessed	Hit or miss	Cache line 0	Cache line 1	Cache line 2	Cache line 3
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???
0	Miss	Block 0	???	???	???
8	Miss	Block 8	???	???	???
0	Miss	Block 0	???	???	???
8	Miss	Block 8	???	???	???
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???

Direct mapping

Reference pattern:
0, 4, 0, 8, 0, 8,
0, 4, 0, 4, 0, 4

Hit ratio = 0%

Fully associative cache state

	Block accessed	Hit or miss	Cache line 0	Cache line 1	Cache line 2	Cache line 3
Associative mapping	0	Miss	Block 0	???	???	???
	4	Miss	Block 0	Block 4	???	???
	0	Hit	Block 0	Block 4	???	???
	8	Miss	Block 0	Block 4	Block 8	???
Reference pattern: 0, 4, 0, 8, 0, 8, 0, 4, 0, 4, 0, 4	0	Hit	Block 0	Block 4	Block 8	???
	8	Hit	Block 0	Block 4	Block 8	???
	0	Hit	Block 0	Block 4	Block 8	???
	4	Hit	Block 0	Block 4	Block 8	???
Hit ratio = 75%	0	Hit	Block 0	Block 4	Block 8	???
	4	Hit	Block 0	Block 4	Block 8	???
	0	Hit	Block 0	Block 4	Block 8	???
	4	Hit	Block 0	Block 4	Block 8	???

Set-associative cache state

	Block accessed	Hit or miss	Set 0		Set 1	
			Cache line 0	Cache line 1	Cache line 0	Cache line 1
Set-associative mapping	0	Miss	Block 0	???	???	???
	4	Miss	Block 0	Block 4	???	???
	0	Hit	Block 0	Block 4	???	???
	8	Miss	Block 0	Block 8	???	???
Reference pattern: 0, 4, 0, 8, 0, 8, 0, 4, 0, 4, 0, 4	0	Hit	Block 0	Block 8	???	???
	8	Hit	Block 0	Block 8	???	???
	0	Hit	Block 0	Block 8	???	???
	4	Miss	Block 0	Block 4	???	???
Hit ratio = 67%	0	Hit	Block 0	Block 4	???	???
	4	Hit	Block 0	Block 4	???	???
	0	Hit	Block 0	Block 4	???	???
	4	Hit	Block 0	Block 4	???	???

Another example solved in direct mapping function:ref {0,7,9,10,0,7,9,10,0,7,9,10}

Direct-mapped cache state

	Block accessed	Hit or miss	Cache line 0	Cache line 1	Cache line 2	Cache line 3
Direct mapping	0	Miss	Block 0	???	???	???
	7	Miss	Block 0	???	???	Block 7
	9	Miss	Block 0	Block 9	???	Block 7
	10	Miss	Block 0	Block 9	Block 10	Block 7
Reference pattern: 0, 7, 9, 10, 0, 7, 9, 10, 0, 7, 9, 10	0	Hit	Block 0	Block 9	Block 10	Block 7
	7	Hit	Block 0	Block 9	Block 10	Block 7
	9	Hit	Block 0	Block 9	Block 10	Block 7
	10	Hit	Block 0	Block 9	Block 10	Block 7
Hit ratio = 67%	0	Hit	Block 0	Block 9	Block 10	Block 7
	7	Hit	Block 0	Block 9	Block 10	Block 7
	9	Hit	Block 0	Block 9	Block 10	Block 7
	10	Hit	Block 0	Block 9	Block 10	Block 7

Memory Interleaving

This is a design made to compensate for the relatively slow speed of dynamic random-access memory (DRAM) or core memory, by spreading memory addresses evenly across memory banks. That way, contiguous memory reads and writes are using each memory bank in turn, resulting in higher memory throughputs due to reduced waiting for memory banks to become ready for desired operations.

To speed up the memory operations (read and write), the main memory of $2^n = N$ words can be organized as a set of $2^{m=M}$ independent memory modules (where $m < n$ each containing 2^{n-m} words. If these M modules can work in parallel (or in a pipeline fashion), then ideally an M fold speed improvement can be expected.

In low-order interleaving, consecutive addresses in the memory will be found in different memory banks.

While in high-order interleaving, consecutive addresses in the main memory will be found in the same memory bank.

High-order arrangement

0	00	00	4	01	00	8	10	00	12	11	00
1	00	01	5	01	01	9	10	01	13	11	01
2	00	10	6	01	10	10	10	10	14	11	10
3	00	11	7	01	11	11	10	11	15	11	11
M0			M1			M2			M3		

Low-order arrangement (interleaving)

0	00	00	1	00	01	2	00	10	3	00	11
4	01	00	5	01	01	6	01	10	7	01	11
8	10	00	9	10	01	10	10	10	11	10	11
12	11	00	13	11	01	14	11	10	15	11	11
M0			M1			M2			M3		

High and low order interleaving