Computer Architecture 1

2024

ENG. Adeeb kenno

References: William Stallings-Computer Organization and Architecture: Designing for Performance-8th Edition-2011
John Catsoulis-An introduction to Computer Architecture: Designing Embedded Hardware- 2nd Edition- 2020

Introduction

- What is a **computer**? a computer is a machine designed to process, store, and retrieve data.
- What is a **computer architecture** refers to those attributes of a system visible to a programmer or, put another way, those attributes that have a direct impact on the logical execution of a program.
- What is a **Computer organization** refers to the operational units and their interconnections that realize the architectural specifications.

Introduction

- Examples of **architectural attributes** include the instruction set, the number of bits used to represent various data types (e.g., numbers, characters), I/O mechanisms, and techniques for addressing memory.
- Examples of **organizational attributes** include those hardware details transparent to the programmer, such as control signals; interfaces between the computer and peripherals; and the memory technology used.

Introduction

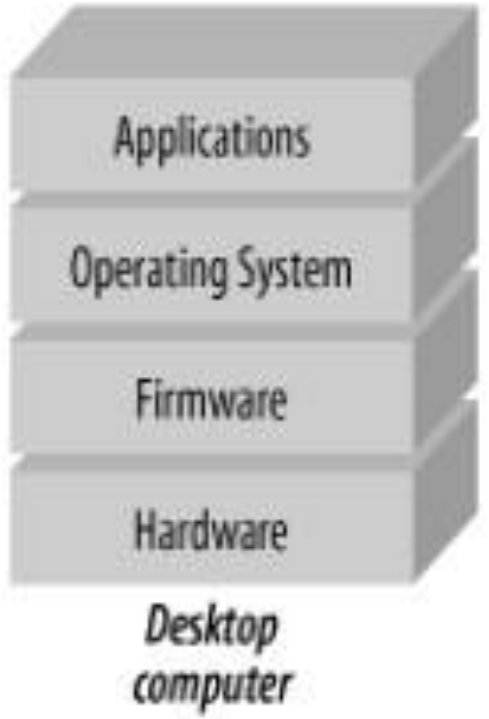
- Data that process by computer may be numbers in a spreadsheet, characters of text in a document, dots of color in an image, waveforms of sound, or the state of some system, such a CD player.
- All data is stored in the computer as numbers.
- The computer manipulates the data by performing operations on the numbers.

Computer System

- A computer system is composed of many parts, both hardware and software.
- At the heart of the computer is the processor, the hardware that executes the computer programs.
- The computer also has memory, often several different types in one system. The memory is used to store programs while the processor is running them, as well as store the data that the programs are manipulating.
- The computer also has devices for storing data, or exchanging data with the outside world.

Computer System

- The software controls the operation and functionality of the computer. There are many "layers" of software in the computer. Typically, a given layer will only interact with the layers immediately above or below it.



Software layers

Software layers

- At the lowest level ***'firmware'***, there are programs that are run by the processor when the computer first powers up. These programs initialize the other hardware subsystems to a known state and configure the computer for correct operation. This software, because it is permanently stored in the computer's memory, is known as *firmware* .
- The *bootloader* is located in the firmware. The bootloader is a special program run by the processor that reads the operating system from disk (or nonvolatile memory or network interface) and places it in memory so that the processor may then run it.

Software layers

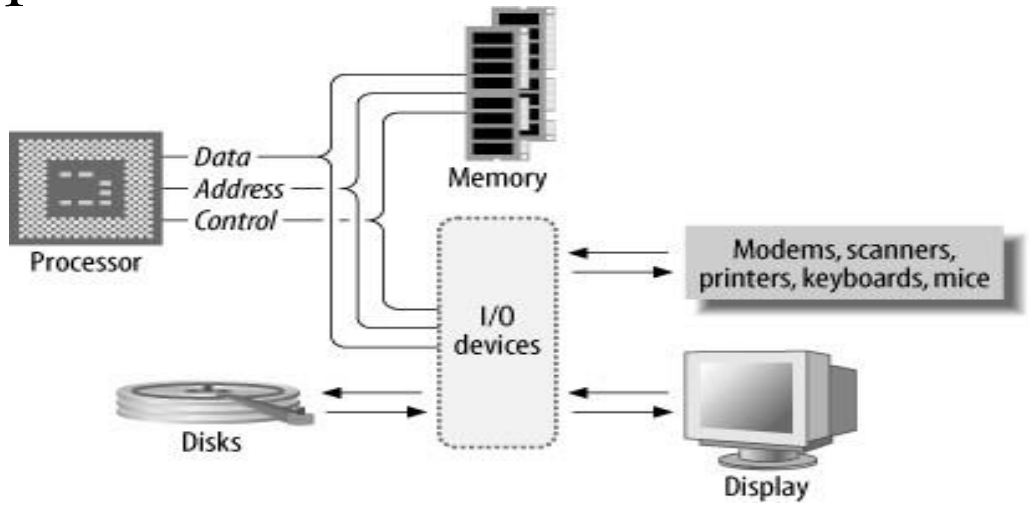
- Above the firmware, the '***operating system'*** controls the operation of the computer. It organizes the use of memory and controls devices such as the keyboard, mouse, screen, disk drives, and so on. It is also the software that often provides an interface to the user, enabling her to run application programs and access her files on disk.
- At the highest level, the '***application software'*** constitutes the programs that provide the functionality of the computer. Everything below the application is considered system software.

Processors

- The processor is the most important part of a computer, the component around which everything else is centered.
- The processor is the computing part of the computer.
- The processor is an electronic device capable of manipulating data (information) in a way specified by a sequence of instructions.
- The instructions are also known as opcodes or machine code.
- A sequence of instructions is what constitutes a program.
- Each type of processor has a different instruction set, meaning that the functionality of the instructions varies.
- Processor instructions are often quite simple, such as “add two numbers” or “call this function.” In some processors, however, they can be as complex and sophisticated as “if the result of the last operation was zero, then use this particular number to reference another number in memory, and then increment the first number once you’ve finished.”
- There are four main components of processor: **Control unit**: Controls the operation of the CPU and hence the computer, **Arithmetic and logic unit (ALU):** Performs the computer’s data processing functions, **Registers**: Provides storage internal to the CPU, **CPU interconnection**: Some mechanism that provides for communication among the control unit, ALU, and registers.

Basic System Architecture

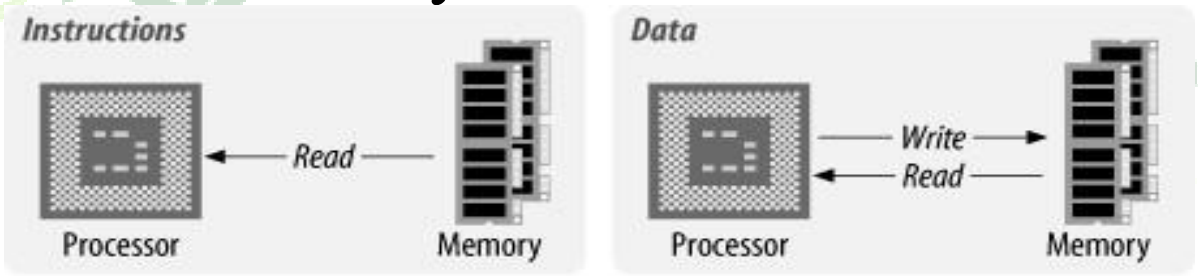
- The processor alone is incapable of successfully performing any tasks. It requires memory (for program and data storage), support logic, and at least one I/O device ("input/output device") used to transfer data between the computer and the outside world.



Basic computer system

Basic System Architecture

- The memory of the computer system contains both the **instructions** that the processor will execute and the **data** it will manipulate.
- The memory of a computer system is never empty. It always contains something, whether it be instructions, meaningful data, or just the random garbage that appeared in the memory when the system powered up. Instructions are read (fetched) from memory, while data is both read from and written to memory



Data flow

Basic System Architecture

- This form of computer architecture is known as a *Von Neumann machine*, named after John Von Neumann.
- Von Neumann computers are what can be termed control-flow computers. The steps taken by the computer are governed by the sequential control of a program. In other words, the computer follows a step-by-step program that governs its operation.

Basic System Architecture

A classical Von Neumann machine has several distinguishing characteristics:

- ***There is no real difference between data and instructions:*** A processor can be directed to begin execution at a given point in memory, and it has no way of knowing whether the sequence of numbers beginning at that point is data or instructions.
- ***Data has no inherent meaning:*** There is nothing to distinguish between a number that represents a dot of color in an image and a number that represents a character in a text document. Meaning comes from how these numbers are treated under the execution of a program.

Basic System Architecture

A classical Von Neumann machine has several distinguishing characteristics:

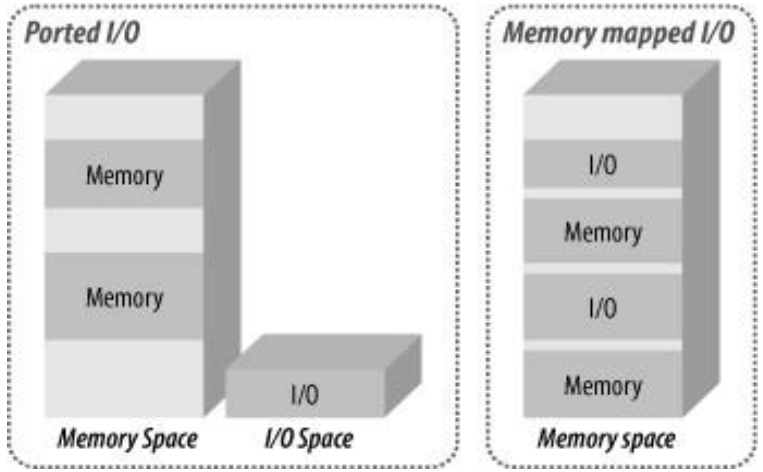
- ***Data and instructions share the same memory:*** This means that sequences of instructions in a program may be treated as data by another program. A compiler creates a program binary by generating a sequence of numbers (instructions) in memory.
- ***Memory is a linear (one-dimensional) array of storage locations:*** The processor's memory space may contain the operating system, various programs, and their associated data, all within the same linear space.

Basic System Architecture

- Each location in the memory space has a unique, sequential address. The address of a memory location is used to specify (and select) that location. The memory space is also known as the *address space* , and how that address space is partitioned between different memory and I/O devices is known as the *memory map* .
- The address space is the array of all addressable memory locations. In an 8-bit processor (such as the 68HC11) with a 16-bit address bus, this works out to be $2^{16} = 65,536 = 64K$ of memory. Hence, the processor is said to have a 64K address space. Processors with 32-bit address buses can access $2^{32} = 4,294,967,296 = 4G$ of memory.

Basic System Architecture

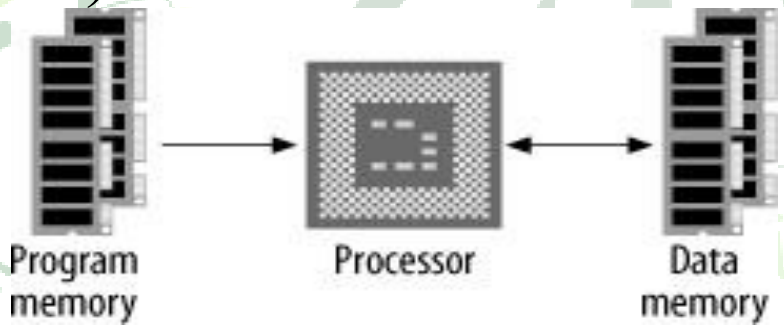
- Some processors, notably the Intel x86 family, have a separate address space for I/O devices with separate instructions for accessing this space. This is known as *ported I/O* . However, most processors make no distinction between memory devices and I/O devices within the address space. I/O devices exist within the same linear space as memory devices, and the same instructions are used to access each. This is known as *memory-mapped I/O*. Memory-mapped I/O is certainly the most common form. Ported I/O address spaces are becoming rare, and the use of the term even rarer.



Ported versus memory-mapped I/O spaces

Basic System Architecture

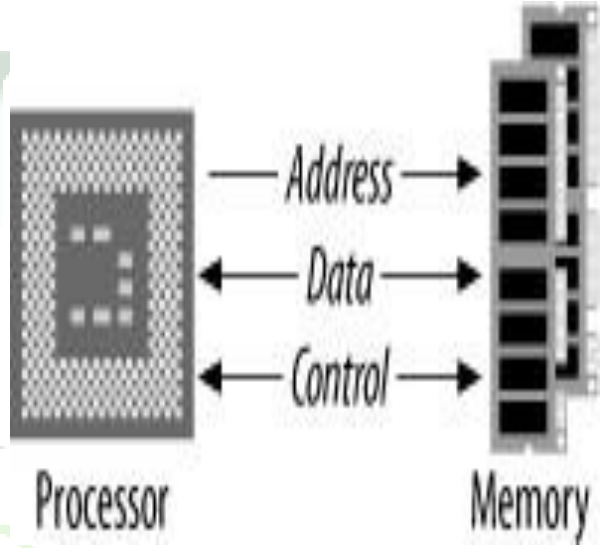
- The main deviation from this is the *Harvard architecture* , in which instructions and data have different memory spaces with separate address, data, and control buses for each memory space. This has a number of advantages in that instruction and data fetches can occur concurrently, and the size of an instruction is not set by the size of the standard data unit (word).



Harvard architecture

Buses

- A bus is a physical group of signal lines that have a related function. Buses allow for the transfer of electrical signals between different parts of the computer system and thereby transfer information from one device to another. For example, the data bus is the group of signal lines that carry data between the processor and the various subsystems that comprise the computer.
- The "width" of a bus is the number of signal lines dedicated to transferring information. For example, an 8-bit-wide bus transfers 8 bits of data in parallel.
- The majority of processors available today (with some exceptions) use the three-bus system architecture.
- The three buses are the ***address bus*** , the ***data bus***, and the ***control bus***.



Three-bus system

Buses

- The data bus is bidirectional, the direction of transfer being determined by the processor.
- The address bus carries the address, which points to the location in memory that the processor is attempting to access. It is the job of external circuitry to determine in which external device a given memory location exists and to activate that device. This is known as *address decoding* .
- The control bus carries information from the processor about the state of the current access, such as whether it is a write or a read operation. The control bus can also carry information back to the processor regarding the current access, such as an address error. Different processors have different control lines, but there are some control lines that are common among many processors. The control bus may consist of output signals such as read, write, valid address, etc. A processor usually has several input control lines too, such as reset, one or more interrupt lines, and a clock input.

Clock Speed and Instructions per Second

- Operations performed by a processor, such as fetching an instruction, decoding the instruction, performing an arithmetic operation, and so on, are governed by a system clock.
- Typically, all operations begin with the pulse of the clock. Thus, at the most fundamental level, the speed of a processor is dictated by the pulse frequency produced by the clock, measured in cycles per second, or Hertz (Hz).
- Typically, clock signals are generated by a quartz crystal, which generates a constant signal wave while power is applied. This wave is converted into a digital voltage pulse stream (1 or 0) that is provided in a constant flow to the processor circuitry.
- For example, a 1-GHz processor receives 1 billion pulses per second.
- The rate of pulses is known as the **clock rate**, or **clock speed**. One increment, or pulse, of the clock is referred to as a **clock cycle**, or a **clock tick**. The time between pulses is the **cycle time**.

Clock Speed and Instructions per Second

- The clock rate is not arbitrary, but must be appropriate for the physical layout of the processor.
- Actions in the processor require signals to be sent from one processor element to another.
- The execution of an instruction involves a number of discrete steps, such as **fetching the instruction from memory, decoding the various portions of the instruction, loading and storing data, and performing arithmetic and logical operations.**
- Thus, most instructions on most processors require multiple clock cycles to complete.
- Some instructions may take only a few cycles, while others require dozens.

Clock Speed and Instructions per Second

- A processor is driven by a clock with a constant frequency f or, equivalently, a constant cycle time T , where T=1Tf. Define the instruction count, Ic, for a program as the number of machine instructions executed for that program until it runs to completion or for some defined time interval. An important parameter is the average cycles per instruction CPI for a program. If all instructions required the same number of clock cycles, then CPI would be a constant value for a processor. However, on any give processor, the number of clock cycles required varies for different types of instructions, such as load, store, branch, and so on. Let CPIi be the number of cycles required for instruction type i. and Ii be the number of executed instructions of type I for a given program. Then we can calculate an overall CPI as follows:

$$CPI = \frac{\sigma_{i=1}^{n} CPI_i \times I_i}{Ic}$$

Clock Speed and Instructions per Second

- A common measure of performance for a processor is the rate at which instructions are executed, expressed as millions of instructions per second (MIPS), referred to as the **MIPS rate**. We can express the MIPS rate in terms of the clock rate and CPI as follows:

$$MIPS\ rate = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6}$$

Clock Speed and Instructions per Second

- For example, consider the execution of a program which results in the execution of 2 million instructions on a 400-MHz processor. The program consists of four major types of instructions. The instruction mix and the CPI for each instruction type are given below based on the result of a program trace experiment:

Instruction Type	CPI	Instruction Mix
Arithmetic and logic	1	60%
Load/store with cache hit	2	18%
Branch	4	12%
Memory reference with cache miss	8	10%

- The average CPI when the program is executed on a uniprocessor with the above trace results is CPI= 0.6+ (2× 0.18)+ (4 ×0.12) +(8× 0.1) =2.24. The corresponding MIPS rate is $(400 \times 10^6)/ (2.24\times 10^6) \approx 178$.

Clock Speed and Instructions per Second

- The CPU program execution time on the computer R1 for example:

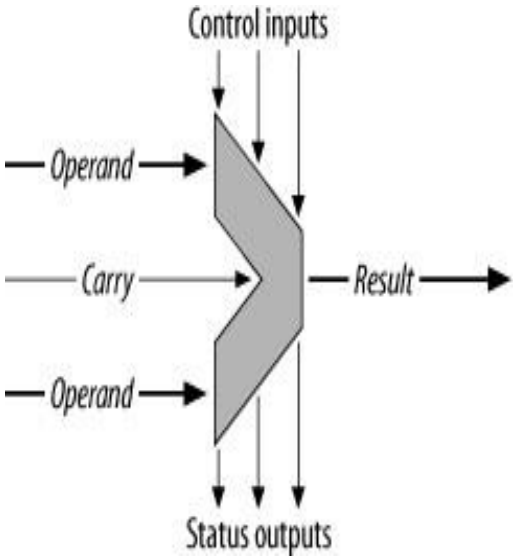
$$CPU_{time} = \frac{Number\_of\_instruction}{MIPS \times 10^6}$$

Processor operation

- There are **six basic types** of access that a processor can perform with external chips. The processor can:
  1. Write data to memory.
  2. Write data to an I/O device.
  3. Read data from memory.
  4. Read data from an I/O device.
  5. Read instructions from memory.
  6. Perform internal manipulation of data within the processor.
- In many systems, writing data to memory is functionally identical to writing data to an I/O device. Similarly, reading data from memory constitutes the same external operation as reading data from an I/O device, or reading an instruction from memory. In other words, the processor makes no distinction between memory and I/O.
- The internal data storage of the processor is known as its registers . The processor has a limited number of registers, and these are used to hold the current data/operands that the processor is manipulating.

Arithmetic Logic Unit (ALU)

- The Arithmetic Logic Unit (ALU) performs the internal arithmetic manipulation of data in the processor. The instructions that are read and executed by the processor control the data flow between the registers and the ALU. The instructions also control the arithmetic operations performed by the ALU via the ALU's control inputs.
- Whenever instructed by the processor, the ALU performs an operation (typically one of addition, subtraction, NOT, AND, OR, XOR, shift left/right, or rotate left/right) on one or more values. These values, called operands , are typically obtained from two registers, or from one register and a memory location. The result of the operation is then placed back into a given destination register or memory location. The status outputs indicate any special attributes about the operation, such as whether the result was zero, negative, or if an overflow or carry occurred. Some processors have separate units for multiplication and division, and for bit shifting, providing faster operation and increased throughput.



ALU block diagram

Arithmetic Logic Unit (ALU)

- In the binary number system, the numbers can be represented with just the digits zero and one.
- For purposes of computer storage and processing, however, we do not have the benefit of minus signs and periods.
- Only binary digits (0 and 1) may be used to represent numbers.
- In general, if an n-bit sequence of binary digits is interpreted as an unsigned integer A, its value is

$$A=\sum_{i=0}^{n-1} 2^i a_i$$

Arithmetic Logic Unit (ALU)

- **Sign-Magnitude Representation:**

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases}$$

+ 18	= 00010010	
− 18	= 10010010	(sign magnitude)

- **One's complement Representation:**

$A = \sigma_{ni=-01} 2_i a_{i-} (2^n - 1) a_{n-1}$

+18	= 00010010	
- 18	= 11101101	(One's complement)

- **Two's complement Representation:**

$A = \sigma_{ni=-01} 2_i a_{i-} (2^n) a_{n-1}$

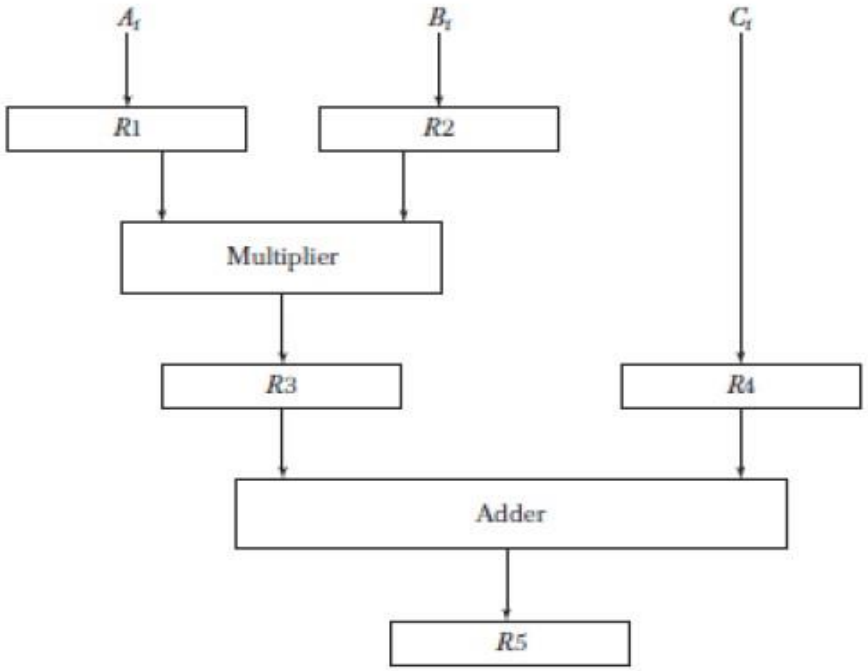
+18	= 00010010	
- 18	= 11101110	(Two's complement)

Pipeline processing

- Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of increasing the computational speed of a computer system. Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time.
- Pipelining is a technique of decomposing a sequential process into sub operations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments.
- Perhaps the simplest way of viewing the pipeline structure is to imagine that each segment consists of an input register followed by a combinational circuit. The register holds the data and the combinational circuit performs the sub operation in the particular segment. The output of the combinational circuit in a given segment is applied to the input register of the next segment.
- A clock is applied to all registers after enough time has elapsed to perform all segment activity. In this way the information flows through the pipeline one step at a time.

Pipeline processing

- Suppose that we want to perform the combined multiply and add operations with a stream of numbers. $A_i * B_i + C_i$ *for* $i=1,2,3,\ldots\ldots 7$



$R1 \leftarrow A_i, \quad R2 \leftarrow B_i$ — Input A_i and B_i

$R3 \leftarrow R1 * R2, \quad R4 \leftarrow C_i$ — Multiply and input C_i

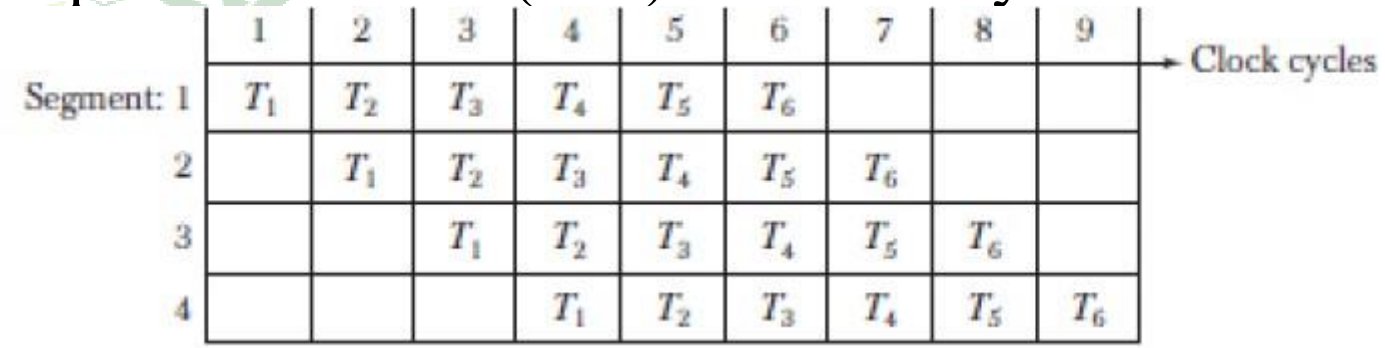
$R5 \leftarrow R3 + R4$ — Add C_i to product

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	–	–	–
2	A_2	B_2	$A_1 * B_1$	C_1	–
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	–	–	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	–	–	–	–	$A_7 * B_7 + C_7$

Pipeline processing

Pipeline processing

- Now consider the case where a *k*-segment pipeline with a clock cycle time *tp* is used to execute *n* tasks. The first task *T1* requires a time equal to *k tp* to complete its operation since there are *k*-segments in the pipe. The remaining *n*-1 tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to (*n*-1) *tp*. Therefore, to complete *n* tasks using a *k*-segment pipeline requires *k*+(*n*-1) clock cycles.
- The next Figure shows four segments and six tasks. The time required to complete all operations is 4 + (6 -1) = 9 clock cycles.



Pipeline processing

- Next consider a nonpipelined unit that performs the same operation and takes a time equal to *tn* to complete each task. The total time required for *n* tasks is *n*tn*. The speed up of a pipeline processing over an equivalent nonpipelined processing is defined by the ratio.

$$S = \frac{n\, t_n}{(k+n-1)t_p}$$

- As the number of tasks increases, n becomes much larger than *k - 1*, and *k + n — 1* approaches the value of n. Under this condition, the speedup becomes.

$$S = \frac{t_n}{t_p}$$

Pipeline processing

- To clarify the meaning of the speedup ratio, consider the following numerical example. Let the time it takes to process a sub operation in each segment be equal to tp=20ns.
- Assume that the pipeline has k=4 segments and executes n=100 tasks in sequence. The pipeline system will take $(k+n-1)tp$ = (4+99)*20=2060 ns to complete. Assuming that $tn=ktp$=4*20=80ns, a non pipelined system requires $nktp$=100*80=8000 ns to complete the 100tasks. The speed up ratio is equal to 8000/2060=3.88. As the number of tasks increases, the speed up will approach 4, which is equal to the number of segments in the pipeline. If we assume that tn=60ns, the speed up becomes 60/20=3.

Interrupts

- Interrupts (also known as *traps* or *exceptions* in some processors) are a technique of diverting the processor from the execution of the current program so that it may deal with some event that has occurred. Such an event may be an error from a peripheral, or simply that an I/O device has finished the last task it was given and is now ready for another. An interrupt is generated in your computer every time you type a key or move the mouse.
- Interrupts free the processor from having to continuously check the I/O devices to determine whether they require service. Instead, the processor may continue with other tasks. The I/O devices will notify it when they require attention by asserting one of the processor's interrupt inputs.
- Interrupts can be of varying priorities in some processors, thereby assigning differing importance to the events that can interrupt the processor.
- the processor is servicing a low-priority interrupt, it will pause it in order to service a higher-priority interrupt. However, if the processor is servicing an interrupt and a second, lower-priority interrupt occurs, the processor will ignore that interrupt until it has finished the higher-priority service.

Interrupts

When an interrupt occurs:

- The usual procedure is for the processor to save its state by pushing its registers and program counter onto the stack.
- The processor then loads an ***interrupt*** *vector* into the program counter.
- The interrupt vector is the address at which an *interrupt service routine* (*ISR*) lies.
- Loading the vector into the program counter causes the processor to begin execution of the ISR, performing whatever service the interrupting device required.
- The last instruction of an ISR is always a *return from interrupt* instruction. This causes the processor to reload its saved state (registers and program counter) from the stack and resume its original program.

Processors with **shadow registers** use these to save their current state, rather than pushing their register bank onto the stack. This saves considerable memory accesses (and therefore time) when processing an interrupt.

Types of interrupts (Hardware interrupts)

- There are two ways of telling when an I/O device (such as a serial controller or a disk controller) is ready for the next sequence of data to be transferred. The first is *busy waiting* or *polling* , where the processor continuously checks the device's status register until the device is ready. This wastes the processor's time but is the simplest to implement. For some time-critical applications, polling can reduce the time it takes for the processor to respond to a change of state in a peripheral.
- A better way is for the device to generate an interrupt to the processor when it is ready for a transfer to take place. Small, simple processors may only have one (or two) interrupt inputs, so several external devices may have to share the interrupt lines of the processor. When an interrupt occurs, the processor must check each device to determine which one generated the interrupt. (This can also be considered a form of polling.) The advantage of interrupt polling over ordinary polling is that the polling occurs only when there is a need to service a device. ***Polling interrupts is suitable only in systems that have a small number of devices***; otherwise, the processor will spend too long trying to determine the source of the interrupt.

Types of interrupts (Hardware interrupts)

- The other technique of servicing an interrupt is by using *vectored interrupts*, by which the interrupting device provides the interrupt vector that the processor is to take. Vectored interrupts reduce considerably the time it takes the processor to determine the source of the interrupt. If an interrupt request can be generated from more than one source, it is therefore necessary to assign priorities (levels) to the different interrupts.

Types of interrupts (Software interrupts)

- A software interrupt is generated by an instruction. It is the lowestpriority interrupt and is generally used by programs to request a service to be performed by the system software (operating system or firmware).

CISC and RISC

- There are two major approaches to processor architecture: *Complex Instruction Set Computer* (*CISC*, pronounced “Sisk”) processors and *Reduced Instruction Set Computer* (*RISC*) processors. Classic CISC processors are the Intel x86and Motorola 68xxx. *Common RISC architectures are the Freescale/IBM PowerPC, the MIPS architecture, the Atmel AVR, and the Microchip PIC.*

CISC

- CISC processors have a single processing unit, external memory, and a relatively small register set and many hundreds of different instructions.
- The tendency in processor design throughout the late 70s and early 80s was toward bigger and more complicated instruction sets. Need to input a string of characters from an I/O port?

Advantages:

- Making the job of the assembly-language programmer easier, since you had to write fewer lines of code to get the job done. As memory was slow and expensive in late 70s and early 80s, it also made sense to make each instruction do more. This reduced the number of instructions needed to perform a given function, and thereby reduced memory space and the number of memory accesses required to fetch instructions.

Disadvantages:

- As memory got cheaper and faster in early 90s, and compilers became more efficient, the relative advantages of the CISC approach began to diminish. One main disadvantage of CISC is that the processors themselves get increasingly complicated as a consequence of supporting such a large and diverse instruction set. The control and instruction decode units are complex and slow, the silicon is large and hard to produce, and they consume a lot of power and therefore generate a lot of heat. As processors became more advanced, the overheads that CISC imposed on the silicon became oppressive.

41

RISC

- **The realization of the advantages of CISC led to a rethink of processor design. The result was the RISC architecture, which has led to the development of very high-performance processors.**
- **The basic philosophy behind RISC is to move the complexity from the silicon to the language compiler. The hardware is kept as simple and fast as possible.**
- **A given complex instruction can be performed by a sequence of much simpler instructions. For example, many processors have an xor (exclusive OR) instruction for bit manipulation, and they also have a Clear instruction to set a given register to zero. However, a register can also be set to zero by xor-ing it with itself. Thus, the separate Clear instruction is no longer required. It can be replaced with the already present Xor. Further, many processors are able to clear a memory location directly by writing a zero to it. That same function can be implemented by clearing a register and then storing that register to the memory location.**
- **The instruction to load a register with a literal number can be replaced with the instruction for clearing a register, followed by an add instruction with the literal number as its operand. Thus, six instructions (xor, clear reg, clear memory, load, store, and add) can be replaced with just three (xor, store, and add).**

RISC

- CISC assembly pseudocode

```
clear 0x1000;  clear memory location 0x1000
load r1, #5;   load register 1 with the value 5
```

- RISC assembly pseudocode

```
xor r1, r1;            clear register 1
store r1, 0x1000; clear memory location 0x1000
add r1 , #5;           load register 1 with the value 5
```

- The resulting code size is bigger, but the reduced complexity of the instruction decode unit can result in faster overall operation. Dozens of such code optimizations exist to give RISC its simplicity.

RISC

RISC processors have a number of distinguishing characteristics:

- They have large register sets (in some architectures numbering over 1,000), thereby reducing the number of times the processor must access main memory. Often-used variables can be left inside the processor, reducing the number of accesses to (slow) external memory. Compilers of high-level languages (such as C) take advantage of this to optimize processor performance.
- By having smaller and simpler instruction decode units, RISC processors have fast instruction execution, and this also reduces the size and power consumption of the processing unit. Generally, RISC instructions will take only one or two cycles to execute.
- This is in contrast to instructions for a CISC processor, whose instructions may take many tens of cycles to execute.
- Instructions on a RISC processor have a simple format. All instructions are generally the same length (which makes instruction decode units simpler).

RISC

RISC processors have a number of distinguishing characteristics:

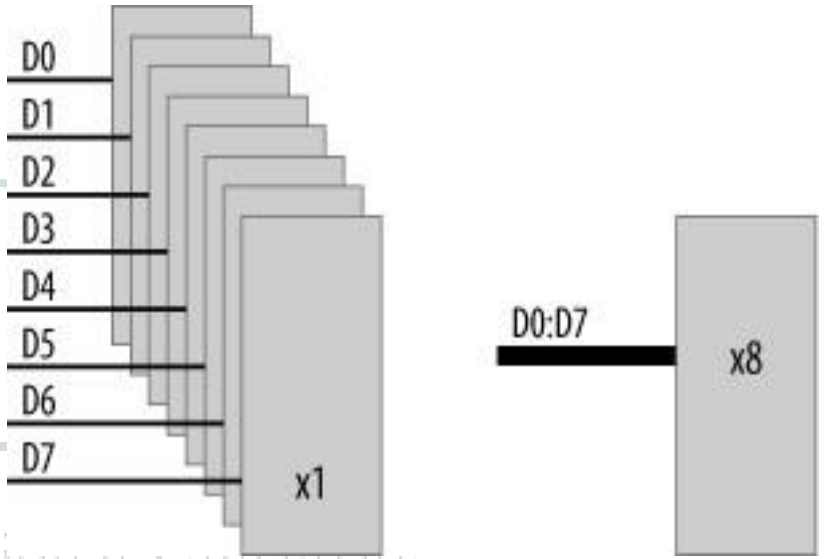
- RISC processors also often have pipelined instruction execution. This means that while one instruction is being executed, the next instruction in the sequence is being decoded, while the third one is being fetched. At any given moment, several instructions will be in the pipeline and in the process of being executed. Again, this provides improved processor performance.
- Due to their low power consumption and computing power, RISC processors are becoming widely used.

Memory

- Memory is used to hold data and software for the processor.
- There is a variety of memory types, and often a mix is used within a single system.
- Some memory will retain its contents while there is no power, yet will be slow to access.
- Other memory devices will be high-capacity, yet will require additional support circuitry and will be slower to access. Still other memory devices will trade capacity for speed, yielding relatively small devices, yet will be capable of keeping up with the fastest of processors.

Memory

- Memory chips can be organized in two ways, either in *word-organized* or *bitorganized* schemes.
- In theword-organized scheme, complete nybbles, bytes, or words are stored withina single component, whereas with bit-organized memory, each bit of a byte or word is allocated to a separate component



Eight bitword-organized -organized 8×18 1

Memory

Memory chips come in different sizes, with the width specified as part of the size description. For instance, a DRAM (dynamic RAM) chip might be described as being 4M×1 (bit-organized), whereas a SRAM (static RAM) may be 512K×8 (word-organized). In both cases, each chip has exactly the same storage capacity, but organized in different ways.

- The common widths for memory chips are x1, x4, and x8, although x16 devices are available. A 32-bit-wide bus can be implemented with thirty-two x1 devices, eight x4 devices, or four x8 devices.

Memory(RAM)

- RAM stands for *Random Access Memory*. This is a bit of a misnomer, since most (all) computer memory may be considered “random access.”
- RAM is the “working memory” in the computer system.
- It is where the processor may easily write data for temporary storage. RAM is generally *volatile*, losing its contents when the system loses power.
- Any information stored in RAM that must be retained must be written to some form of permanent storage before the system powers down.
- RAMs generally fall into two categories:
    1. Static RAM (also known as SRAM).
    2. Dynamic RAM (also known as DRAM).

Memory(RAM)

- SRAMs use pairs of logic gates to hold each bit of data.
- SRAMs are the fastest form of RAM available, require little external support circuitry, and have relatively low power consumption.
- Their drawbacks are that their capacity is considerably less than DRAM, while being much more expensive.
- Their relatively low capacity requires more chips to be used to implement the same amount of memory.

Memory(RAM)

- DRAM uses arrays of what are essentially capacitors to hold individual bits of data.
- The capacitor arrays will hold their charge only for a short period before it begins to diminish.
- Therefore, DRAMs need continuous refreshing, every few milliseconds or so. This perpetual need for refreshing requires additional support and can delay processor access to the memory.
- DRAMs are the highest-capacity memory devices available and come in a wide and diverse variety of subspecies.
- Many processors have instruction and/or data caches , which store recent memory accesses. These caches are (often, but not always) internal to the processors and are implemented with fast memory cells and high-speed data paths. Instruction execution normally runs out of the instruction cache, providing for fast execution. The processor is capable of rapidly reloading the caches from main memory should a cache miss occur. Some processors have logic that is able to anticipate a cache miss and begin the cache reload prior to the cache miss occurring. Caches are implemented using very fast SRAM and are most often used in large systems to compensate for the slowness of DRAM.

Memory(ROM)

- **ROM stands for *Read-Only Memory*. This is also a bit of a misnomer, since many (modern) ROMs can also be written to.**
- **ROMs are *nonvolatile memory*, requiring no power to retain their contents. They are generally slower than RAM, and considerably slower than fast static RAM.**
- **The primary purpose of ROM within a system is to hold the code (and sometimes data) that needs to be present at power-up. Such software is generally known as firmware and contains software to initialize the computer by placing I/O devices into a known state.**
- **It may contain either a bootloader program to load an operating system off disk.**
- **Standard ROM is fabricated (in a simplistic sense) from a large array of diodes. The unwritten bit state for a ROM is all 1s, each byte location reading as 0xFF. The process of loading software into a ROM is known as burning the ROM. This term comes from the fact that the programming process is performed by passing a sufficiently large current through the appropriate diodes to "blow them," or burn them, thereby creating a zero at that bit location. A device known as a ROM burner can accomplish this, or, if the system supports it, the ROM may be programmed in-circuit. This is known as In-System Programming (ISP) or, sometimes, In-Circuit Programming (ICP).**
- **One-Time Programmable (OTP) ROMs, as the name implies, can be burned once only. Computer manufacturers typically use them in systems where the firmware is stable and the product is shipping in bulk to customers.**

Memory(ROM)

- OTP ROMs are great for shipping in final products, but they are wasteful for debugging, since with each iteration of code change, a new chip must be burned and the old one thrown away. As such, OTPs make for a very expensive development option. No sane person uses OTPs for development work.
- A (slightly) better choice for system development and debugging is the *Erasable Programmable Read-Only Memory*, or *EPROM*. Shining ultraviolet light through a small window on the top of the chip can erase the EPROM, allowing it to be reprogrammed and reused.
- EPROMs and their equivalent OTP cousins range in capacity from a few kilobytes (exceedingly rare these days) to a megabyte or more.
- The drawback with EPROM technology is that the chip must be removed from the circuit to be erased, and the erasure can take many minutes to complete. The chip is then inserted into the burner, loaded with software, and then placed back in-circuit. This can lead to very slow debug cycles.

Memory(ROM)

- *EEROM* is *Electrically Erasable Read-Only Memory*, also known as *EEPROM* (*Electrically Erasable Programmable Read-Only Memory*). Very rarely, it is also called *Electrically Alterable ReadOnly Memory* (*EAROM*).
- EEROMs can be erased and reprogrammed in-circuit. Their capacity is significantly smaller than standard ROM (typically only a few kilobytes), and so they are not suited to holding firmware. Instead, they are typically used for holding system parameters and mode information to be retained during power-off.

Memory(ROM)

- *Flash* is the newest ROM technology and is now dominant. Flash memory has the re-programmability of EEROM and the large capacity of standard ROMs.
- Flash is normally organized as sectors and has the advantage that individual sectors may be erased and rewritten without affecting the contents of the rest of the device.

Input/Output

- The address space of the processor can contain devices other than memory.
- These are input/output devices (I/O devices, also known as *peripherals*) and are used by the processor to communicate with the external world.
- Some examples are serial controllers that communicate with keyboards, mice, modems, etc.; parallel I/O devices that control some external subsystem; or disk-drive controllers, video and audio controllers, or network interfaces.

Input/Output

There are three main ways in which data may be exchanged with the external world:

- ***Programmed I/O:*** The processor accepts or delivers data at times convenient to it (the processor).
- ***Interrupt-driven I/O:*** External events control the processor by requesting the current program be suspended and the external event be serviced. An external device will interrupt the processor, at which time the processor will suspend the current task (program) and begin executing an interrupt service routine. The service of an interrupt may involve transferring data from input to memory, or from memory to output.
- ***Direct Memory Access (DMA):*** DMA allows data to be transferred from I/O devices to memory directly without the continuous involvement of the processor. DMA is used in high-speed systems, where the rate of data transfer is important. Not all processors support DMA.

DMA

DMA is a way of streamlining transfers of large blocks of data between two sections of memory, or between memory and an I/O device. Let's say you want to read in 100M from disk and store it in memory. You have two options:

- One option is for the processor to read one byte at a time from the disk controller into a register and then store the contents of the register to the appropriate memory location. Then the process starts over again for the next byte.
- The second option in moving large amounts of data around the system is DMA. A special device, called a *DMA Controller* (*DMAC*), performs highspeed transfers between memory and I/O devices. Using DMA bypasses the processor by setting up a *channel* between the I/O device and the memory. Thus, data is read from the I/O device and written into memory without the need to execute code to perform the transfer on a byte-by-byte.

DMA

- In order for a DMA transfer to occur, the DMAC must have use of the address and data buses.
- There are several ways in which this could be implemented by the system designer.
- The most common approach (and probably the simplest) is to suspend the operation of the processor and for the processor to “release” its buses (the buses are tristate).
- This allows the DMAC to “take over” the buses for the short period required to perform the transfer.
- Processors that support DMA usually have a special control input that enables a DMAC (or some other processor) to request the buses.

DMA

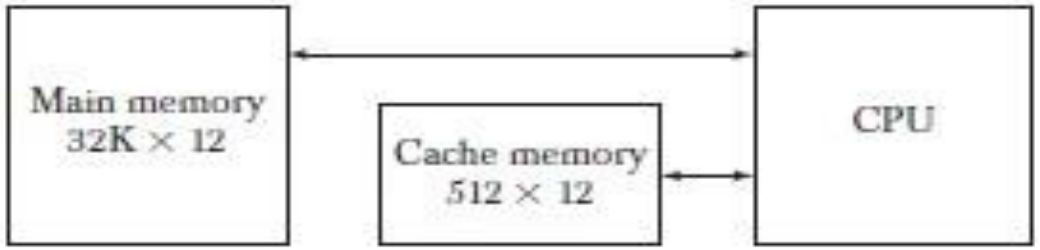
- There are four basic types of DMA:
- ***Standard block transfer:*** Accomplished by the DMA controller performing a sequence of memory transfers. The transfers involve a load operation from a source address followed by a store operation to a destination address. Standard block transfers are initiated under software control and are used for moving data structures from one region of memory to another.
- ***Demand-mode transfers:*** Similar to standard mode except that the transfer is controlled by an external device. Demand-mode transfers are used to move data between memory and I/O or vice versa. The I/O device requests and synchronizes the movement of data.

DMA

- ***Fly-by transfer:*** Provides high-speed data movement in the system. Instead of using multiple bus accesses as with conventional DMA transfers, fly-by transfers move data from source to destination in a single access. The data is not read into the DMAC before going to its destination. During a fly-by transfer, memory and I/O are given different bus control signals. For example, an I/O device is given a read request at the same time that memory is given a write request. Data moves from the I/O device straight into the memory device.
- ***Data-chaining transfers:*** Allow DMA transfers to be performed as specified by a linked-list in memory. Data chaining is started by specifying a pointer to a ***descriptor*** in memory. The descriptor is a table specifying byte count, source address, destination address, and a pointer to the next descriptor. The DMAC loads the relevant information about the transfer from this table and begins moving data. The transfer continues until the number of bytes transferred is equal to the entry in the byte-count field. On completion, the pointer to the next descriptor is loaded.
  This continues until a null pointer is found.

Cache Memory

- **Cache**: The small section of SRAM memory, added between main memory and processor (CPU) to speed up the process of execution, is known as cache memory. It is a high speed & expensive memory.
- Caches are divided into blocks, which may be of various sizes, The number of blocks in a cache is usually a power of 2.
- **Cache hit ratio**: It measures how effectively the cache fulfills the request for getting content. If data has been found in the cache, it is a cache hit else a cache miss. Cache hit ratio = No. of cache hits / (No. of cache hits + No. of cache Miss).



Cache Memory

Cache Memory

- Cache Mapping: The process /technique of bringing data of main memory blocks into the cache block is known as cache mapping. The mapping techniques can be classified as:
    1. Associative Mapping.
    2. Direct Mapping.
    3. Set-Associative Mapping.

Cache Memory(Associative Mapping)

Tag	Word

- Here the mapping of the main memory block can be done with any of the cache block. The memory address has only 2 fields here: tag & word. This technique is called as fully associative cache mapping.
- The associative memory stores both the address and content (data) of the memory word
- If the address is found, the corresponding data bits is read and sent to the CPU. If no match occurs, the main memory is accessed for the word.

Address	**Data**
013A	3871
0CB1	FF12
2239	FE45

Associative cache Mapping (all numbers in Hex)

Cache Memory(Associative Mapping)

Tag	Word

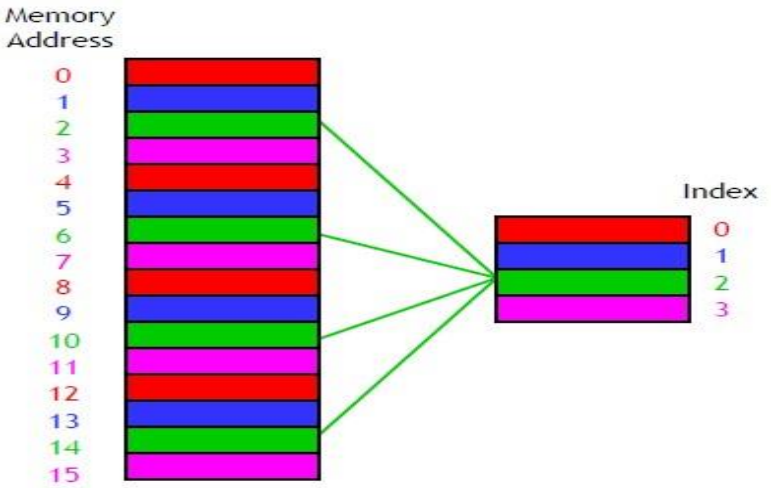
- ***Example:*** If we have a fully associative mapped cache of 8 KB size with block size = 128 bytes and say, the size of main memory is = 64 KB. Then:
- Number of bits for the physical address = 16 bits (as memory size = 64 KB = 26 × 210 = 216)
- Number of bits in **block** (word) offset = 7 bits (as block size = 128 bytes = 2^7)
- No of **tag** bits = Number of bits for the physical address – Number of bits in block offset = 16-7 = 9 bits
- No of cache Blocks = Cache size/block size = 8 KB / 128 Bytes = 8×1024 Bytes/128 Bytes = 2^6 blocks.

Cache Memory(Direct Mapping)

- Associative memories are expensive compared to random-access memories because of the added logic associated with each cell.
- **Direct Mapping**: Each block from main memory has only one possible place in the cache organization in this technique. For example: every block i of the main memory can be mapped to block j of the cache using the formula: **j = i modulo m**

Where: i = main memory block number. j = cache block number.

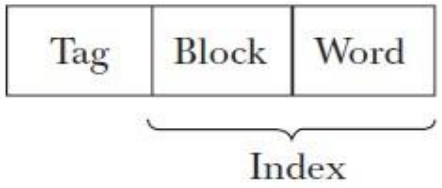
m = number of blocks in the cache.



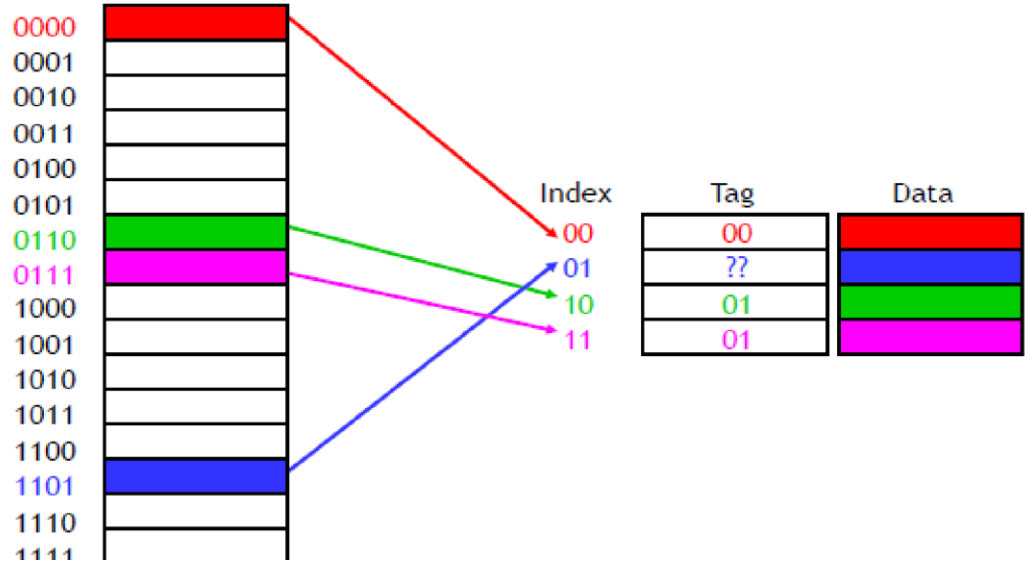
• One way to figure out which cache block a particular memory address should go to is to use the mod (remainder) operator.

Direct Mapping

Cache Memory(Direct Mapping)

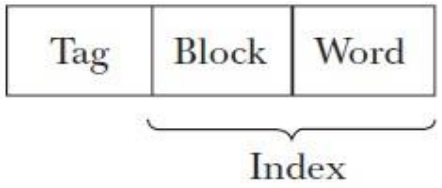


- We need to add tags to the cache, which supply the rest of the address bits to let us distinguish between different memory locations that map to the same cache block. The address here is divided into 3 fields: Tag, Block & Word.
- The number of bits in the Block & Word field is equal to the number of bits in the index fields.
- The number of bits in the index field is equal to the number of address bits required to access the cache memory.

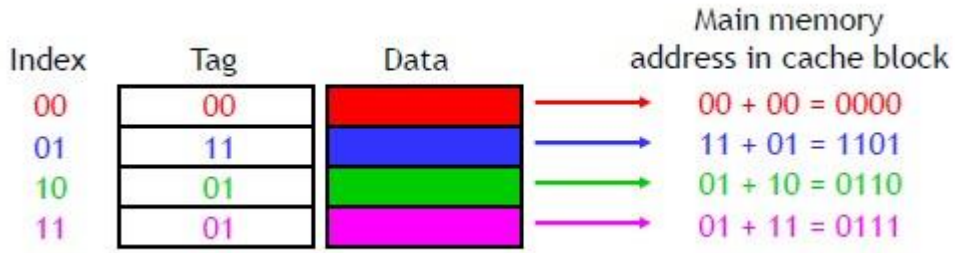


Direct Cache Mapping

Cache Memory(Direct Mapping)



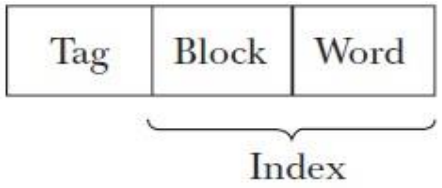
- Now we can tell exactly which addresses of main memory are stored in the cache, by concatenating the cache block tags with the block indices.



Direct Cache Mapping

- To map the memory address to cache: The BLOCK field of the address is used to access the cache's BLOCK. Then, the tag bits in the address is compared with the tag of the block. For a match, a cache hit occurs as the required word is found in the cache. Otherwise, a cache miss occurs and the required word has to be brought into the cache from the Main Memory. The word is now stored in the cache together with the new tag (old tag is replaced).

Cache Memory(Direct Mapping)



- ***Example:*** If we have a fully associative mapped cache of 8 KB size with block size = 128 bytes and say, the size of main memory is = 64 KB. (Assuming word size = 1 byte) Then :
- Number of bits for the physical address = 16 bits (as memory size = 64 KB = 26 × 210 = 216)

Number of bits for **Word** = 7 bits (as block size = 128 bytes = 2^7)

No of **Index** bits = 13 bits (as cache size = 8 KB = $2^3 \times 2^{10} = 2^{13}$)

No of **Block** bits = Number of Index bits- Number of bits for Word = 13 – 7 = 6bits

- OR

(No of cache Blocks = Cache size/block size = 8 KB / 128 Bytes = 8×1024 Bytes/128 Bytes = 2^6blocks → 6bits)

- No of **Tag** bits = Number of bits for the physical address — Number of bits in Index = 16-13 = 3 bits

Cache Memory(Set-Associative Mapping)

- The disadvantage of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time.
- It is the combination of advantages of bothdirect & associative mapping.

Here, the cache consists of a number sets, each of which consists of a number of blocks. The relationships are : **n = w * L**

i = j modulo w

where i : cache set number.

j : main memory block number.n : number of blocks in the cache.

w : number of sets.

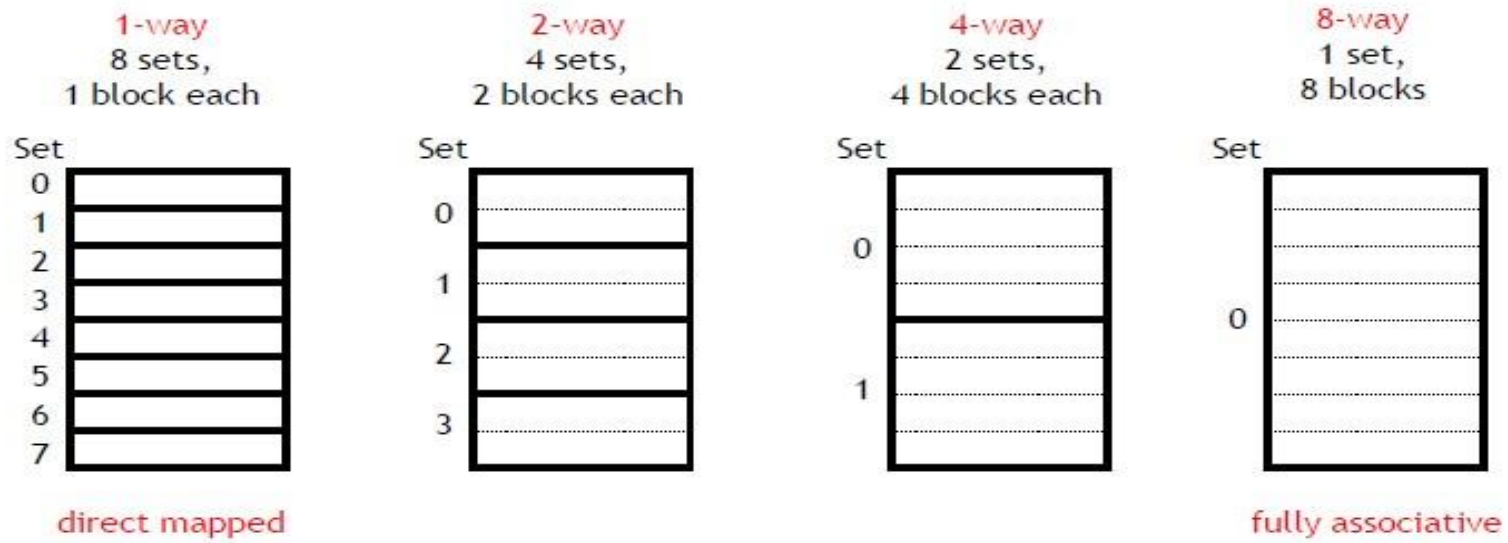
L : number of lines in each set.

The memory address has only 3 fields here: word & set & tag.

Cache Memory(Set-Associative Mapping)

Tag	Set	Word

- This is referred to as L-way set-associative mapping. Block Bj can be translated into any of the blocks in set j using this mapping.



Set-Associative Cache Mapping

Cache Memory(Set-Associative Mapping)

Tag	Set	Word

- **To map the memory address to cache:** Using set field in the memory address, we access the particular set of the cache. Then, the tag bits in the address are compared with the tag of all L blocks within that set. For a match, a cache hit occur as the required word is found in the cache. Otherwise, a cache miss occurs and the required word has to be brought in the cache from the Main Memory. According to the replacement policy used, a replacement is done if the cache is full.

Cache Memory(Set-Associative Mapping)

Tag	Set	Word

- ***Example1:*** If we have a fully associative mapped cache of 8 KB size with block size = 128 bytes and say, the size of main memory is = 64 KB, and we have *"2-way"* set-associative mapping (Assume each word has 8 bits). Then :
- Number of bits for the physical address = 16 bits (as memory size = 64

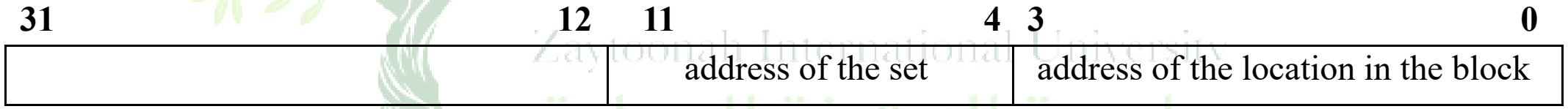
KB = 2_6 * 2_{10} = 2_{16})

- No of cache **Blocks** = Cache size/block size = 8 KB / 128 Bytes = 8×1024 Bytes/128 Bytes = 2^6 cache blocks.
- No of Main Memory Blocks = MM size/block size = 64 KB / 128 Bytes = 64×1024 Bytes/128 Bytes = 2^9 MM blocks.
  No of **sets** of size 2 = No of Cache Blocks/ L = $2^6/2 = 2^5$ cache sets. (L = 2 as it is 2-way set associative mapping)

Cache Memory(Set-Associative Mapping)

Tag	Set	Word

- ***Example2:*** On a computer with a 32-bit memory address and the length of the memory location of 1 byte is installed set-associative cache. Cache size is 16 KB, block (line) size is 16 Bytes, set associative cache is 4-way.
- How many sets are there in cache? M=16KB= 2^4 KB= 2^{14}B , block=B=16B= 2^4 B, E=4= 2^2 , S=M÷(E×B)=2^{14}÷(2^4×2^2)= 2^{14-6} = 2^8=256 sets.
- Which bits in the memory address determine the address of the set? Address bits of set: 4-11.



- Into which set is mapped the content of the memory address 10FFCFF$_{(HEX)}$?

00000001000011111111**11001111**1111, Address belongs to set 207.

Cache Memory

	Associative Mapping	Direct-mapping	Set-Associative Mapping
Advantages	• It is fast. • Easy to implement.	• Simplest type of mapping • Fast as only tag field matching is required while searching for a word. • It is comparatively less expensive than associative mapping.	• It gives better performance than the direct and associative mapping techniques.
Disadvantages	• Expensive because it needs to store address along with the data.	• It gives low performance because of the replacement for data-tag value.	• It is most expensive as with the increase in set size cost also increases.

Parallel and Distributed Computers

- The traditional architecture for computers follows the conventional, Von Neumann serial architecture.
- Computers based on this form usually have a single, sequential processor.
- The main limitation of this form of computing architecture is that the conventional processor is able to execute only one instruction at a time.
- Algorithms that run on these machines must therefore be expressed as a sequential problem.
- A given task must be broken down into a series of sequential steps, each to be executed in order, one at a time.

Parallel and Distributed Computers

- Many problems or large data set that are computationally intensive are also highly parallel.
- Thus, speed advantages may be gained from performing calculations in parallel for each element in the data set, rather than sequentially moving through the data set and computing each result in a serial manner.
- A coarsely grained machine has relatively few processors, whereas a finely grained machine may have tens of thousands of processing elements.
- There are several different forms of parallel machine. Each architecture has its own advantages and limitations.

Parallel and Distributed Computers

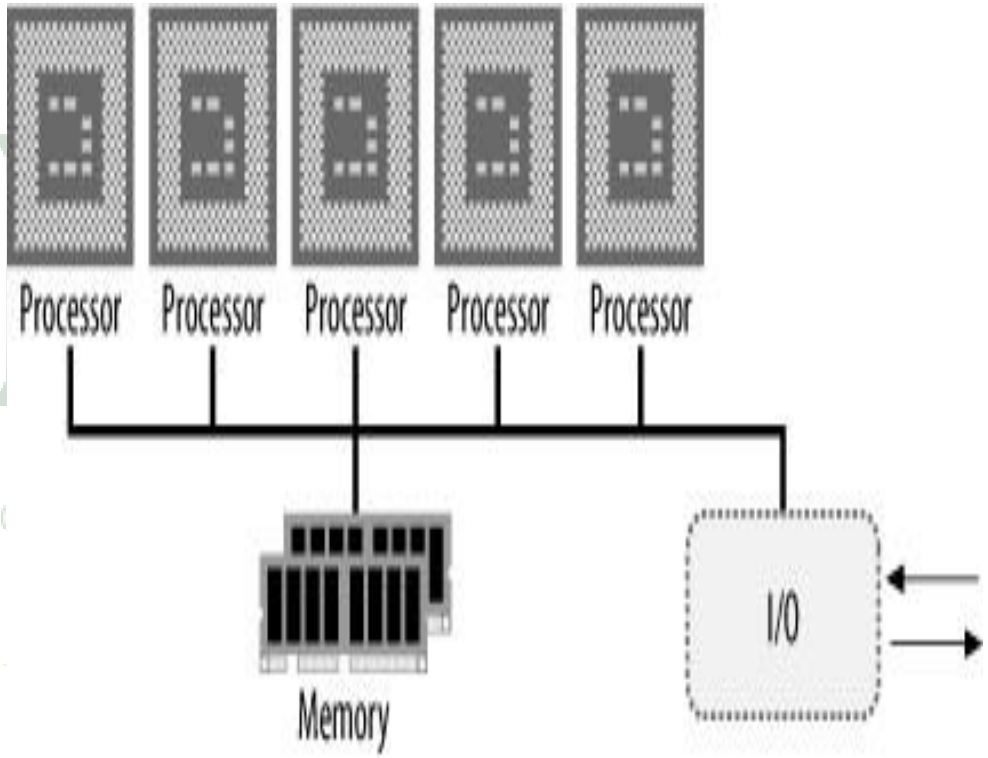
- **SIMD computers**: *Single-Instruction Multiple-Data* (*SIMD*) computers are highly parallel machines, employing large arrays of simple processing elements.
- In an SIMD machine, each processing element has a small amount of local memory.
- The instructions executed by the SIMD computer are broadcast from a central instruction server to every processing element within the machine.
- In this way, each processor executes the same instruction as all other processing elements within the machine.
- Since each processor executes the instruction on its local data, all elements within the data structure are worked upon simultaneously.
- The primary advantage of the SIMD machine is that simple and cheap processing elements are used to form the computer.
- In addition, since each processor is executing the same instructions and therefore sharing a common instruction fetch, the architecture of the machine is somewhat simpler. Only one instruction store is required for the entire computer.
- The use of multiple processing elements, each executing the same instructions in unison, is also the SIMD's main disadvantage. Many problems do not lend themselves to being broken down into a form suitable for executing on an SIMD computer.

Parallel and Distributed Computers

- **MIMD computers**: The other major form of parallel machine is the *Multiple-Instruction Multiple-Data* (***MIMD***) computer.
- These machines are typically coarsely grained collections of semiautonomous processors, each with their own local memory and local programs.
- An algorithm being executed on an MIMD computer is typically broken up into a series of smaller sub-problems, each executed on a processor of the MIMD machine.
- By giving each processing element in the MIMD machine identical programs to execute, the MIMD machine may be treated as an SIMD computer.
- MIMD computers tend to use a smaller number of very powerful processors, rather than a large number of less powerful ones.

Parallel and Distributed Computers

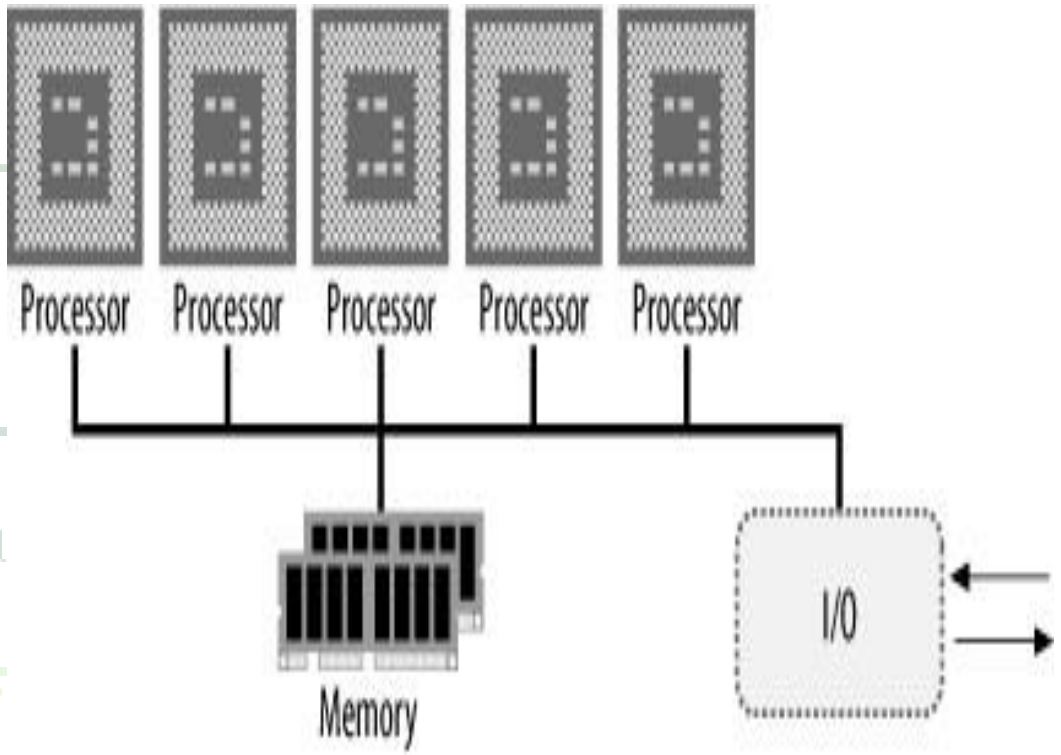
- MIMD computers can be of one of two types: *shared-memory MIMD* and *messagepassing MIMD* .
- **Shared-memory MIMD** systems have an array of high-speed processors, each with local memory or cache, and each with access to a large, global memory.
- The global memory contains the data and programs to be executed by the machine. Also in this memory is a table of processes (or subprograms) awaiting execution.
- Each processor will fetch a process and associated data into its local memory or cache and will run semi-autonomously of the other processors in the system. Process communication also takes place through the global memory.



Shared-memory MIMD

Parallel and Distributed Computers

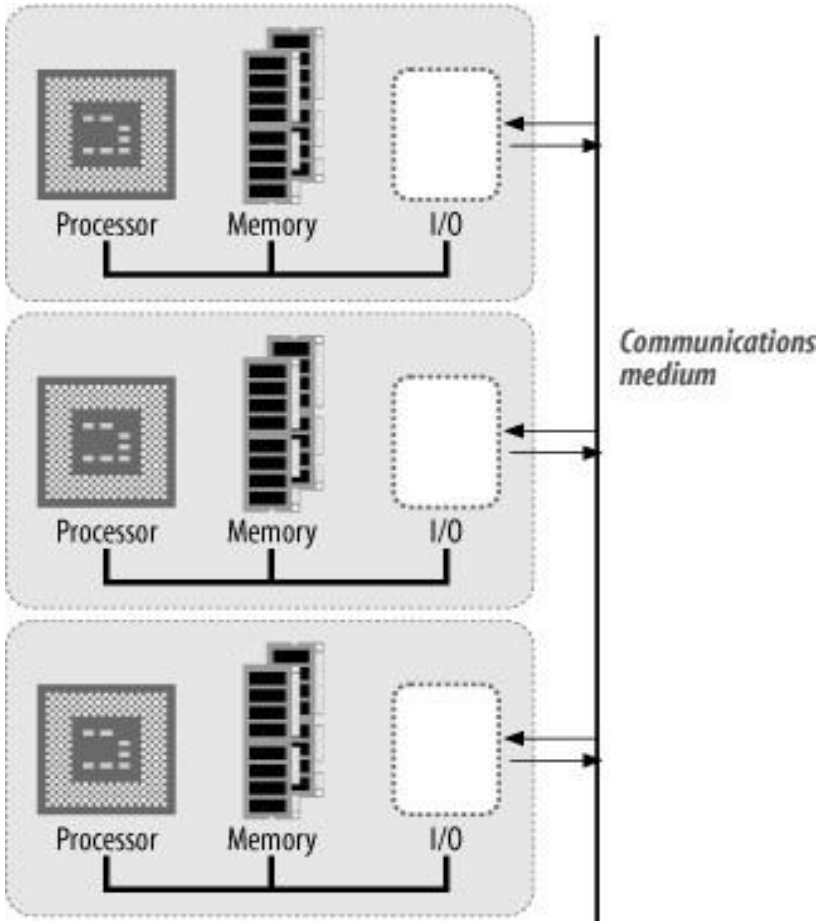
- A speed advantage is gained by sharing the program among several, powerful processors. However, logic within the system must arbitrate between processors for access to the shared memory and associated shared buses of the system.
- In addition, allowances must be made for a processor attempting to access data in global memory that is out of date. If processor A reads a processanddata structure into its local memory and subsequently modifies that data structure, processor B attempting to access the same data structure in main memory must be notified that a more recent version of the data structure exists.



Shared-memory MIMD

Parallel and Distributed Computers

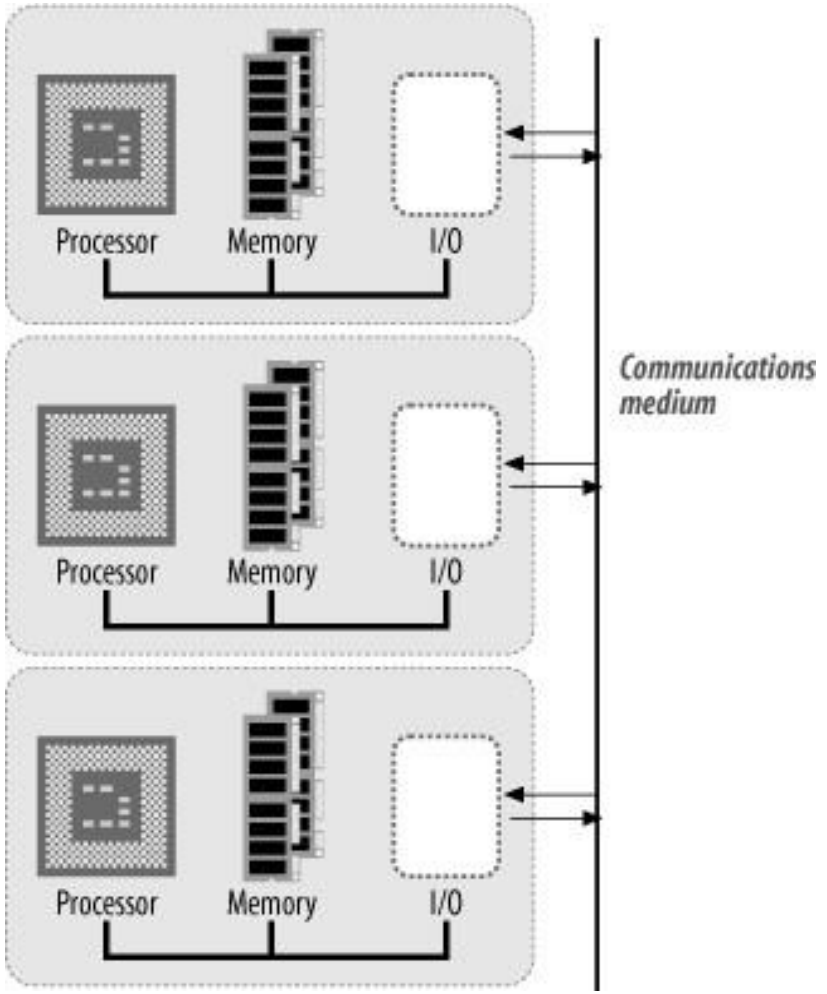
- An alternative MIMD architecture is that of the **message-passing MIMD** computer. In this system, each processor has its own local, main memory.
- No global memory exists for the machine.
- Each processing element (processor with local memory) either loads, or has loaded into it, the programs (and associated data) that it is to execute.
- Each process runs autonomously on its local processor, and interprocess communication is achieved by message-passing through a common medium. The processors may communicate through a single, shared bus.



Message-passing MIMD

Parallel and Distributed Computers

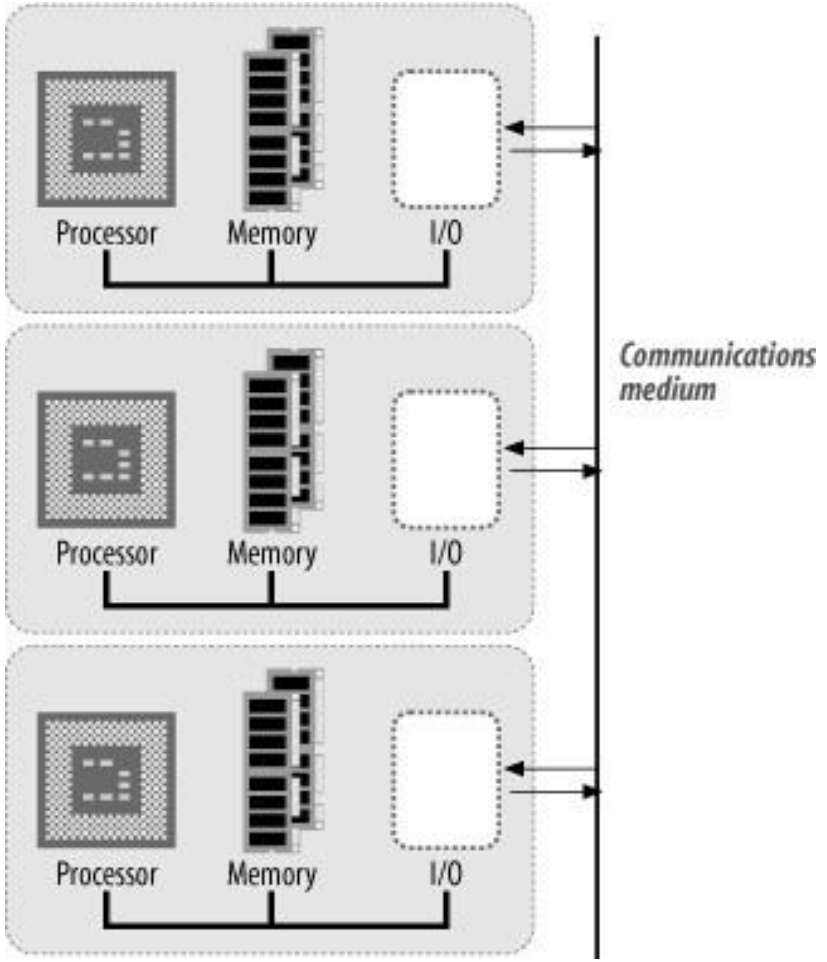
- Such machines do not suffer the bus-contention problems of shared-memory machines.
- However, the most effective and efficient means of interconnecting the processing nodes of a message-passing MIMD machine is still a major area of research.
- Problems that require only a limited amount of interprocess communication may work effectively on a machine without high interconnectivity, whereas other applications may weigh down the communications medium with their message passing. If a percentage of a processing node's time is spent in messagerouting for its neighbors, a machine with a high degree of interprocess communication but a low degree of interconnectivity may spend most of its time dealing in message passing, with little time spent on actual computation.



Message-passing MIMD

Parallel and Distributed Computers

- The ideal interconnection architecture is that of the fully interconnected system, where every processing node has a direct communications link with every other processing node.
  However, this is not always practical, due to the costs and logistics of such a high degree of interconnectivity. A solution to this problem is to provide each processing element in the machine with a limited number of connections, based on the assumption that a processing element will not need or be able to communicate with every other processing element in the machine simultaneously. These limited connections from each processing node may then be interconnected using a *crossbar switch* , thereby providing full interconnectivity for the machine through only a limited number of links per node.



Message-passing MIMD