ADVANCED COMPUTER ARCHITECTURE

Contents:

Module – I
Principles of Processor Performance,
RISC and CISC Architectures,
Pipelining fundamentals,
Pipeline Hazards,
Superscalar Architecture,
Super Pipelined Architecture,
VLIW Architecture.

Module – II
Basic Multiprocessor Architecture:
Flynn’s Classification,
UMA, NUMA,
Distributed Memory Architecture,
Array Processor,
Vector Processors,
Associative Processor,
Systolic architecture.
Interconnection Networks:
Static Networks,
Network Topologies,
Dynamic Networks.

Module –III
Hierarchical Memory Technology:
Data and Instruction caches,
Multi-level caches,
Cache memory mapping policies,
Cache Coherence,
Cache Performance,
Virtual memory,
Page replacement techniques,
Memory Inter leaving,
Memory Management hardware.

Module – IV
Data Flow Computer Architecture:
Static Data flow computer,
Dynamic Data flow computer,
Cluster computers,
Distributed computing,
Cloud computing.

Partha Sarathi

A **microprocessor** is a processing unit on a single chip. It is an integrated circuit which performs the core functions of a computer CPU. It is a multipurpose programmable silicon chip constructed using Metal Oxide Semiconductor (MOS) technology which is clock driven and register based. It accepts binary data as input and provides output after processing it as per the specification of instructions stored in the memory. These microprocessors are capable of processing 128 bits at a time at the speed of one billion instructions per second.

Characteristics of a micro processor:

- **Instruction Set –**
  Set of complete instructions that the microprocessor executes is termed as the instruction set.

- **Word Length –**
  The number of bits processed in a single instruction is called word length or word size. Greater the word size, larger the processing power of the CPU.

- **System Clock Speed –**
  Clock speed determines how fast a single instruction can be executed in a processor. The microprocessor's pace is controlled by the System Clock. Clock speeds are generally measured in million of cycles per second (MHz) and thousand million of cycles per second (GHz). Clock speed is considered to be a very important aspect of predicting the performance of a processor.

Classification of Microprocessors:
Central Processing Unit Architecture operates the capacity to work from "Instruction Set Architecture" to where it was designed. The architectural designs of CPU are RISC (Reduced instruction set computing) and CISC (Complex instruction set computing). CISC has the ability to execute addressing modes or multi-step operations within one instruction set. It is the design of the CPU where one instruction performs many low-level operations. For example, memory storage, an arithmetic operation and loading from memory. RISC is a CPU design strategy based on the insight that simplified instruction set gives higher performance when combined with a microprocessor architecture which has the ability to execute the instructions by using some microprocessor cycles per instruction.

- Hardware of the Intel is termed as Complex Instruction Set Computer (CISC)

- Apple hardware is Reduced Instruction Set Computer (RISC).

What is RISC and CISC Architecture
Hardware designers invent numerous technologies & tools to implement the desired architecture in order to fulfill these needs. Hardware architecture may be implemented to be either hardware specific or software specific, but according to the application both are used in the required quantity. As far as the processor hardware is concerned, there are 2 types of concepts to implement the processor hardware architecture. First one is RISC and other is CISC.

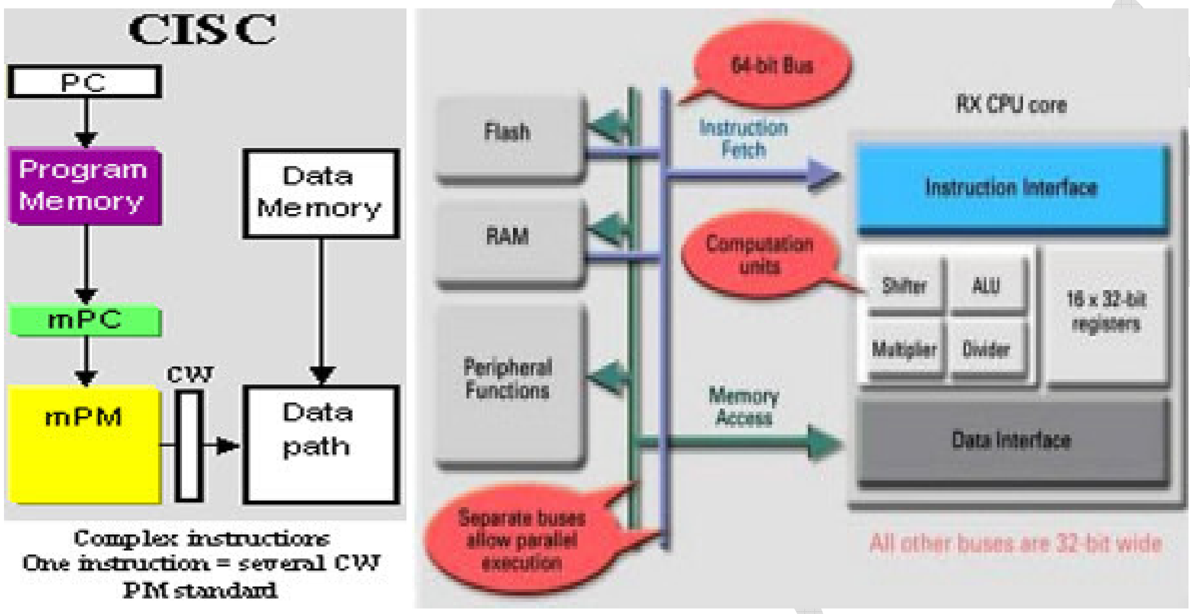
CISC Architecture
It stands for Complex Instruction Set Computer. These processors offer the users, hundreds of instructions of variable sizes. CISC architecture includes a complete set of special purpose circuits that carry out these instructions at a very high speed. These instructions interact with memory by using complex addressing modes. CISC processors reduce the program size and hence lesser number of memory cycles are required to execute the programs. This increases the overall speed of execution. The main idea is to make hardware complex as a single instruction will do all loading, evaluating and storing operations just like a multiplication command will do stuff like loading data, evaluating and storing it.

The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction. Computers based on the CISC architecture are designed to decrease the memory cost. Because, the large programs need more storage, thus increasing the memory cost and large memory becomes more expensive. To solve these problems, the number of instructions per program can be reduced by embedding the number of

Partha Sarathi

operations in a single instruction, thereby making the instructions more complex.
Examples: Intel architecture, AMD

- MUL loads two values from the memory into separate registers in CISC.
- CISC uses minimum possible instructions by implementing hardware and executes operations.
- Instruction Set Architecture is a medium to permit communication between the programmer and the hardware. Data execution part, copying of data, deleting or editing is the user commands used in the microprocessor and with this microprocessor the Instruction set architecture is operated.
- The main keywords used in the above Instruction Set Architecture are as below



Instruction Set: Group of instructions given to execute the program and they direct the computer by manipulating the data. Instructions are in the form – Opcode (operational code) and Operand. Where, opcode is the instruction applied to load and store data, etc. The operand is a memory register where instruction applied.
Addressing Modes: Addressing modes are the manner in the data is accessed. Depending upon the type of instruction applied, addressing modes are of various types such as direct mode where straight data is accessed or indirect mode where the location of the data is accessed. Processors having identical ISA may be very different in organization. Processors with identical ISA and nearly identical organization is still not nearly identical.
CPU performance is given by the fundamental law

$$CPU\ Time = \frac{Seconds}{Program} = \frac{Instructions}{Program} X \frac{Cycles}{Instructions} X \frac{Seconds}{Cycle}$$

Thus, CPU performance is dependent upon Instruction Count, CPI (Cycles per instruction) and Clock cycle time. And all three are affected by the instruction set architecture.

	Instruction Count	CPI	Clock
Program	X		
Compiler	X	X	
Instruction Set Architecture	X	X	X
Microarchitecture		X	X
Physical Design			X

This underlines the importance of the instruction set architecture. There are two prevalent instruction set architectures
Examples of CISC PROCESSORS

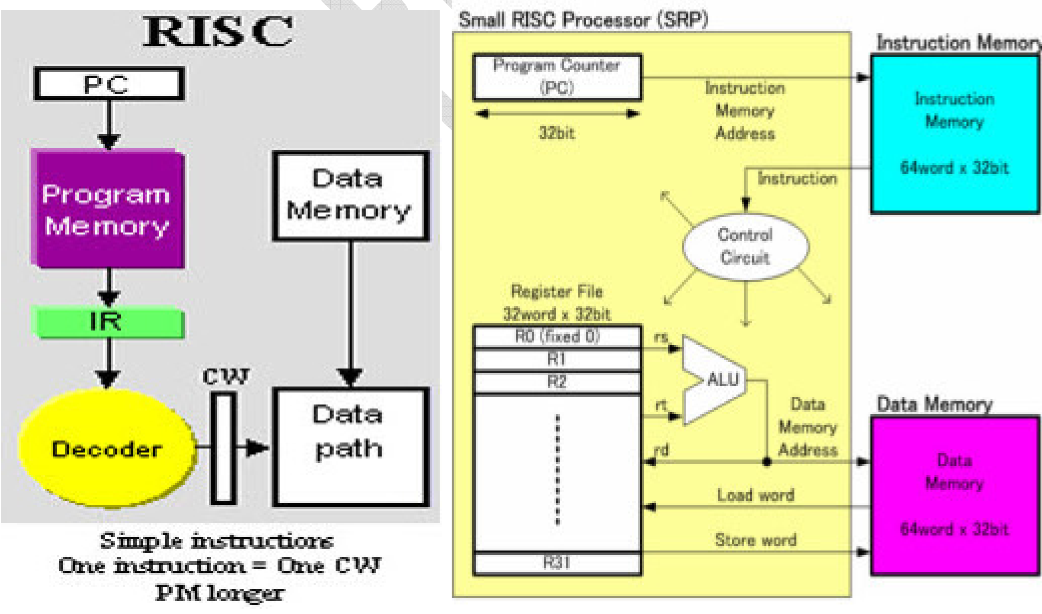
IBM 370/168 – It was introduced in the year 1970. CISC design is a 32 bit processor and four 64-bit floating point registers.
VAX 11/780 – CISC design is a 32-bit processor and it supports many numbers of addressing modes and machine instructions which is from Digital Equipment Corporation.
Intel 80486 – It was launched in the year 1989 and it is a CISC processor, which has instructions varying lengths from 1 to 11 and it will have 235 instructions.

CHARACTERISTICS OF CISC ARCHITECTURE

- Instruction-decoding logic will be Complex.
- Instruction are larger than one word size.
- Instruction may take more than single clock cycle to get executed.
- Complex Addressing Modes means One instruction is required to support multiple addressing modes.
- Less number of general purpose register as operation get performed in memory itself.
- Various CISC designs are set up two special registers for the stack pointer, handling interrupts, etc.
- MUL is referred to as a “complex instruction” and requires the programmer for storing functions.

RISC Architecture

It stands for Reduced Instruction Set Computer. It is a type of microprocessor architecture that uses a small set of instructions of uniform length. These are simple instructions which are generally executed in one clock cycle. RISC chips are relatively simple to design and inexpensive. The setback of this design is that the computer has to repeatedly perform simple operations to execute a larger program having a large number of processing operations. The main idea behind is to make hardware simpler by using an instruction set composed of a few basic steps for loading, evaluating and storing operations just like a load command will load data, store command will store the data.
RISC (Reduced Instruction Set Computer) is used in portable devices due to its power efficiency. For Example, Apple iPod and Nintendo DS. RISC is a type of microprocessor architecture that uses highly-optimized set of instructions. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program Pipelining is one of the unique feature of RISC. It is performed by overlapping the execution of several instructions in a pipeline fashion. It has a high performance advantage over CISC.
Examples: SPARC, POWER PC etc.



RISC ARCHITECTURE CHARACTERISTICS

- Simple Instructions are used in RISC architecture.
- RISC helps and supports few simple data types and synthesize complex data types.
- RISC utilizes simple addressing modes and fixed length instructions for pipelining.

- More number of general purpose register.
- Instruction take single clock cycle to get executed.
- The amount of work that a computer can perform is reduced by separating "LOAD" and "STORE" instructions.
- RISC contains Large Number of Registers in order to prevent various number of interactions with memory.
- In RISC, Pipelining is easy as the execution of all instructions will be done in a uniform interval of time i.e. one click.
- In RISC, more RAM is required to store assembly level instructions.
- Reduced instructions need a less number of transistors in RISC.
- RISC uses Harvard memory model means it is Harvard Architecture.
- A compiler is used to perform the conversion operation means to convert a high-level language statement into the code of its form.

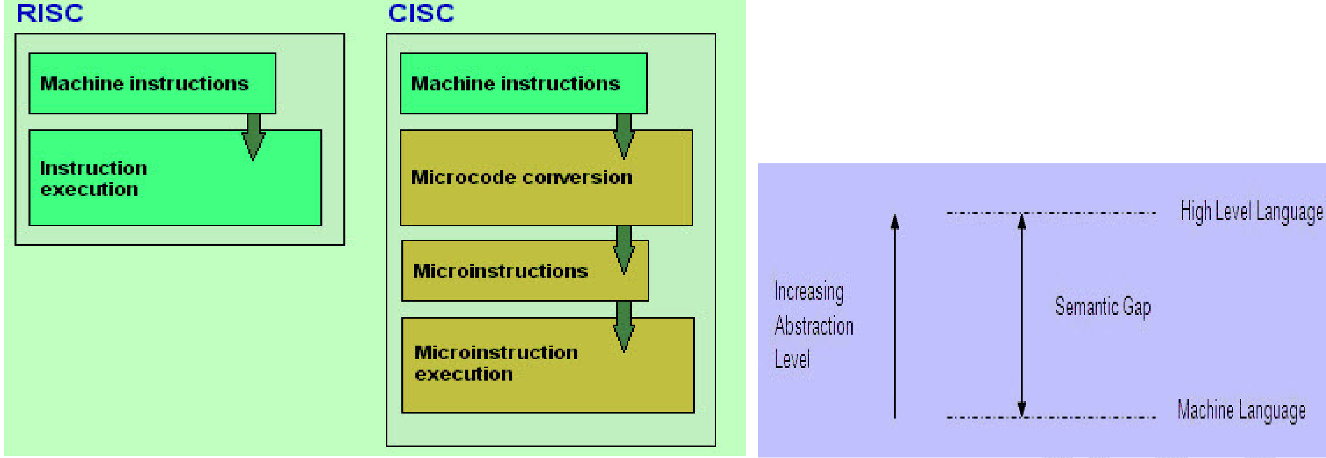
RISC & CISC Comparison

RISC	CISC
1. RISC stands for Reduced Instruction Set Computer.	1. CISC stands for Complex Instruction Set Computer.
2. RISC processors have simple instructions taking about one clock cycle. The average clock cycle per instruction (CPI) is 1.5	2. CSIC processor has complex instructions that take up multiple clocks for execution. The average clock cycle per instruction (CPI) is in the range of 2 and 15.
3. Performance is optimized with more focus on software	3. Performance is optimized with more focus on hardware.
4. It has no memory unit and uses a separate hardware to implement instructions..	4. It has a memory unit to implement complex instructions.
5. It has a hard-wired unit of programming.	5. It has a microprogramming unit.
6. The instruction set is reduced i.e. it has only a few instructions in the instruction set. Many of these instructions are very primitive.	6. The instruction set has a variety of different instructions that can be used for complex operations.
7. The instruction set has a variety of different instructions that can be used for complex operations.	7. CISC has many different addressing modes and can thus be used to represent higher-level programming language statements more efficiently.
8. Complex addressing modes are synthesized using the software.	8. CISC already supports complex addressing modes
9. Multiple register sets are present	9. Only has a single register set
10. RISC processors are highly pipelined	10. They are normally not pipelined or less pipelined
11. The complexity of RISC lies with the compiler that executes the program	11. The complexity lies in the microprogram
12. Execution time is very less	12. Execution time is very high
13. Code expansion can be a problem	13. Code expansion is not a problem
14. Decoding of instructions is simple.	14. Decoding of instructions is complex
15. It does not require external memory for calculations	15. It requires external memory for calculations
16. The most common RISC microprocessors are Alpha, ARC, ARM, AVR, MIPS, PA-RISC, PIC, Power Architecture, and SPARC.	16. Examples of CISC processors are the System/360, VAX, PDP-11, Motorola 68000 family, AMD and Intel x86 CPUs.
17. RISC architecture is used in high-end applications such as video processing, telecommunications and image processing.	17. CISC architecture is used in low-end applications such as security systems, home automation, etc.

MUL instruction is divided into three instructions
"LOAD" – moves data from the memory bank to a register
"PROD" – finds product of two operands located within the registers

"STORE" – moves data from a register to the memory banks

The main difference between RISC and CISC is the number of instructions and its complexity.

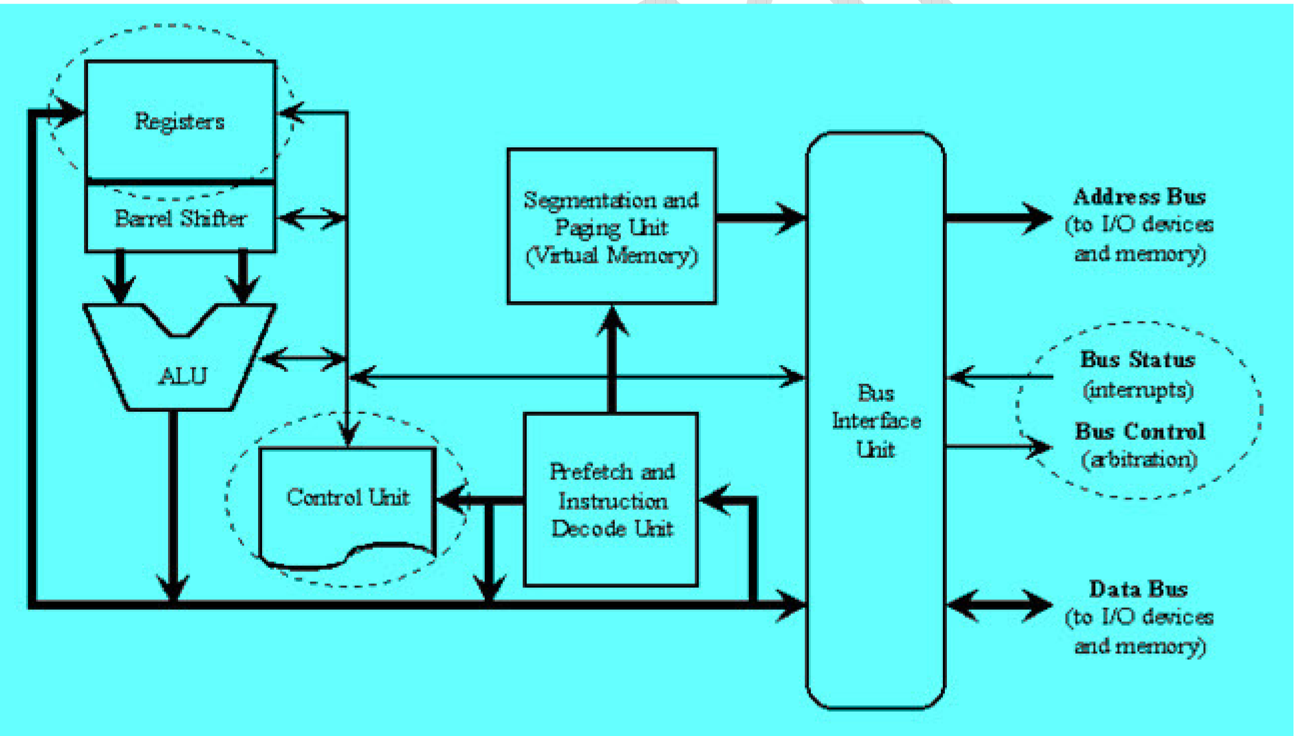


SEMANTIC GAP

Both RISC and CISC architectures have been developed as an attempt to cover the semantic gap.

With an objective of improving efficiency of software development, several powerful programming languages have come up, viz., Ada, C, C++, Java, etc. They provide a high level of abstraction, conciseness and power. By this evolution the semantic gap grows. To enable efficient compilation of high level language programs, CISC and RISC designs are the two options.

CISC designs involve very complex architectures, including a large number of instructions and addressing modes, whereas RISC designs involve simplified instruction set and adapt it to the real requirements of user programs.



Multiplication of two Numbers in Memory

If the main memory is divided into areas that are numbered from row1:column 1 to row 5 :column 4. The data is loaded into one of four registers (A, B, C, or D). To find multiplication of two numbers- One stored in location 1:3 and other stored in location 4:2 and store back result in 1:3.

The Advantages and Disadvantages of RISC and CISC

The Advantages of RISC architecture

- RISC(Reduced instruction set computing)architecture has a set of instructions, so high-level language compilers can produce more efficient code
- It allows freedom of using the space on microprocessors because of its simplicity.

- Many RISC processors use the registers for passing arguments and holding the local variables.
- RISC functions use only a few parameters, and the RISC processors cannot use the call instructions, and therefore, use a fixed length instruction which is easy to pipeline.
- The speed of the operation can be maximized and the execution time can be minimized. Very less number of instructional formats, a few numbers of instructions and a few addressing modes are needed.

The Disadvantages of RISC architecture

- Mostly, the performance of the RISC processors depends on the programmer or compiler as the knowledge of the compiler plays a vital role while changing the CISC code to a RISC code
- While rearranging the CISC code to a RISC code, termed as a code expansion, will increase the size. And, the quality of this code expansion will again depend on the compiler, and also on the machine's instruction set.
- The first level cache of the RISC processors is also a disadvantage of the RISC, in which these processors have large memory caches on the chip itself. For feeding the instructions, they require very fast memory systems.

Advantages of CISC architecture

- Microprogramming is easy assembly language to implement, and less expensive than hard wiring a control unit.
- The ease of microcoding new instructions allowed designers to make CISC machines upwardly compatible:
- As each instruction became more accomplished, fewer instructions could be used to implement a given task.

Disadvantages of CISC architecture

- The performance of the machine slows down due to the amount of clock time taken by different instructions will be dissimilar
- Only 20% of the existing instructions is used in a typical programming event, even though there are various specialized instructions in reality which are not even used frequently.
- The conditional codes are set by the CISC instructions as a side effect of each instruction which takes time for this setting – and, as the subsequent instruction changes the condition code bits – so, the compiler has to examine the condition code bits before this happens.

What is Pipelining?

To improve the performance of a CPU we have two options:
1) Improve the hardware by introducing faster circuits.
2) Arrange the hardware such that more than one operation can be performed at the same time.

Since, there is a limit on the speed of hardware and the cost of faster circuits is quite high, we have to adopt the 2nd option.

Pipelining : Pipelining is a process of arrangement of hardware elements of the CPU such that its overall performance is increased. Simultaneous execution of more than one instruction takes place in a pipelined processor. Let us see a real life example that works on the concept of pipelined operation. Consider a water bottle packaging plant. Let there be 3 stages that a bottle should pass through, Inserting the bottle(**I**), Filling water in the bottle(**F**), and Sealing the bottle(**S**). Let us consider these stages as stage 1, stage 2 and stage 3 respectively. Let each stage take 1 minute to complete its operation.
Now, in a non pipelined operation, a bottle is first inserted in the plant, after 1 minute it is moved to stage 2 where water is filled. Now, in stage 1 nothing is happening. Similarly, when the bottle moves to stage 3, both stage 1 and stage 2 are idle. But in pipelined operation, when the bottle is in stage 2, another bottle can be loaded at stage 1. Similarly, when the bottle is in stage 3, there can be one bottle each in stage 1 and stage 2. So, after each minute, we get a new bottle at the end of stage 3. Hence, the average time taken to manufacture 1 bottle is :

Without pipelining = 9/3 minutes = 3m

```
I F S | | | | | |

| | | I F S | | |

| | | | | | I F S (9 minutes)
```

With pipelining = 5/3 minutes = 1.67m

```
I F S | |
| I F S |
| | I F S (5 minutes)
```

Thus, pipelined operation increases the efficiency of a system.

Design of a basic pipeline

- In a pipelined processor, a pipeline has two ends, the input end and the output end. Between these ends, there are multiple stages/segments such that output of one stage is connected to input of next stage and each stage performs a specific operation.
- Interface registers are used to hold the intermediate output between two stages. These interface registers are also called latch or buffer.
- All the stages in the pipeline along with the interface registers are controlled by a common clock.

Execution in a pipelined processor

Execution sequence of instructions in a pipelined processor can be visualized using a space-time diagram. For example, consider a processor having 4 stages and let there be 2 instructions to be executed. We can visualize the execution sequence through the following space-time diagrams:

Non overlapped execution:

STAGE / CYCLE	1	2	3	4	5	6	7	8
S1	I_1				I_2			
S2		I_1				I_2		
S3			I_1				I_2	
S4				I_1				I_2

Total time = 8 Cycle

Overlapped execution:

STAGE / CYCLE	1	2	3	4	5
S1	I_1	I_2			
S2		I_1	I_2		
S3			I_1	I_2	
S4				I_1	I_2

Total time = 5 Cycle

Pipeline Stages

RISC processor has 5 stage instruction pipeline to execute all the instructions in the RISC instruction set. Following are the 5 stages of RISC pipeline with their respective operations:

- **Stage 1 (Instruction Fetch)**
  In this stage the CPU reads instructions from the address in the memory whose value is present in the program counter.
- **Stage 2 (Instruction Decode)**
  In this stage, instruction is decoded and the register file is accessed to get the values from the registers used in the instruction.
- **Stage 3 (Instruction Execute)**
  In this stage, ALU operations are performed.

- **Stage 4 (Memory Access)**
  In this stage, memory operands are read and written from/to the memory that is present in the instruction.
- **Stage 5 (Write Back)**
  In this stage, computed/fetched value is written back to the register present in the instructions.

Performance of a pipelined processor

Consider a 'k' segment pipeline with clock cycle time as 'Tp'. Let there be 'n' tasks to be completed in the pipelined processor. Now, the first instruction is going to take 'k' cycles to come out of the pipeline but the other 'n – 1' instructions will take only '1' cycle each, i.e, a total of 'n – 1' cycles. So, time taken to execute 'n' instructions in a pipelined processor:

ETpipeline = k + n – 1 cycles
= (k + n – 1) Tp

In the same case, for a non-pipelined processor, execution time of 'n' instructions will be:

ETnon-pipeline = n * k * Tp

So, speedup (S) of the pipelined processor over non-pipelined processor, when 'n' tasks are executed on the same processor is:

S = Performance of pipelined processor /

Performance of Non-pipelined processor

As the performance of a processor is inversely proportional to the execution time, we have,

S = ETnon-pipeline / ETpipeline
=> S = [n * k * Tp] / [(k + n – 1) * Tp]
S = [n * k] / [k + n – 1]

When the number of tasks 'n' are significantly larger than k, that is, n >> k

S = n * k / n

S = k

Where 'k' are the number of stages in the pipeline.

Also, **Efficiency** = Given speed up / Max speed up = S / Smax
We know that, Smax = k
So, **Efficiency** = S / k
Throughput = Number of instructions / Total time to complete the instructions
So, **Throughput** = n / (k + n – 1) * Tp
Note: The cycles per instruction (CPI) value of an ideal pipelined processor is 1

Dependencies in a pipelined processor

There are mainly three types of dependencies possible in a pipelined processor. These are :
1) Structural Dependency(Structural hazards)
2) Control Dependency (branch hazards)
3) Data Dependency(Data hazards)
These dependencies may introduce stalls in the pipeline.
Stall : A stall is a cycle in the pipeline without new input.

Structural dependency

This dependency arises due to the resource conflict in the pipeline. A resource conflict is a situation when more than one instruction tries to access the same resource in the same cycle. A resource can be a register, memory, or ALU.

Example:

INSTRUCTION / CYCLE	1	2	3	4	5
I_1	IF(Mem)	ID	EX	Mem	
I_2		IF(Mem)	ID	EX	
I_3			IF(Mem)	ID	EX
I_4				IF(Mem)	ID

In the above scenario, in cycle 4, instructions I_1 and I_4 are trying to access same resource (Memory) which introduces a resource conflict.

To avoid this problem, we have to keep the instruction on wait until the required resource (memory in our case) becomes available. This wait will introduce stalls in the pipeline as shown below:

CYCLE	1	2	3	4	5	6	7	8
I_1	IF(Mem)	ID	EX	Mem	WB			
I_2		IF(Mem)	ID	EX	Mem	WB		
I_3			IF(Mem)	ID	EX	Mem	WB	
I_4				–	–	–	IF(Mem)	

Solution for structural dependency

To minimize structural dependency stalls in the pipeline, we use a hardware mechanism called Renaming.

Renaming : According to renaming, we divide the memory into two independent modules used to store the instruction and data separately called Code memory(CM) and Data memory(DM) respectively. CM will contain all the instructions and DM will contain all the operands that are required for the instructions.

INSTRUCTION/ CYCLE	1	2	3	4	5	6	7
I_1	IF(CM)	ID	EX	DM	WB		
I_2		IF(CM)	ID	EX	DM	WB	
I_3			IF(CM)	ID	EX	DM	WB
I_4				IF(CM)	ID	EX	DM
I_5					IF(CM)	ID	EX
I_6						IF(CM)	ID
I_7							IF(CM)

Control Dependency (Branch Hazards)

This type of dependency occurs during the transfer of control instructions such as BRANCH, CALL, JMP, etc. On many instruction architectures, the processor will not know the target address of these instructions when it needs to insert the new instruction into the pipeline. Due to this, unwanted instructions are fed to the pipeline.

Consider the following sequence of instructions in the program:

100: I_1

101: I_2 (JMP 250)

102: I_3

.

.

250: BI_1

Expected output: I_1 -> I_2 -> BI_1

NOTE: Generally, the target address of the JMP instruction is known after ID stage only.

INSTRUCTION/ CYCLE	1	2	3	4	5	6
I_1	IF	ID	EX	MEM	WB	
I_2		IF	ID (PC:250)	EX	Mem	WB
I_3			IF	ID	EX	Mem
BI_1				IF	ID	EX

Output Sequence: I_1 -> I_2 -> I_3 -> BI_1

So, the output sequence is not equal to the expected output, that means the pipeline is not implemented correctly.

To correct the above problem we need to stop the Instruction fetch until we get target address of branch instruction. This can be implemented by introducing delay slot until we get the target address.

INSTRUCTION/ CYCLE	1	2	3	4	5	6
I_1	IF	ID	EX	MEM	WB	
I_2		IF	ID (PC:250)	EX	Mem	WB
Delay	–	–	–	–	–	–
BI_1				IF	ID	EX

Output Sequence: I_1 -> I_2 -> Delay (Stall) -> BI_1

As the delay slot performs no operation, this output sequence is equal to the expected output sequence. But this slot introduces stall in the pipeline.

Solution for Control dependency Branch Prediction is the method through which stalls due to control dependency can be eliminated. In this at 1st stage prediction is done about which branch will be taken.For branch prediction Branch penalty is zero.

Branch penalty : The number of stalls introduced during the branch operations in the pipelined processor is known as branch penalty.

NOTE : As we see that the target address is available after the ID stage, so the number of stalls introduced in the pipeline is 1. Suppose, the branch target address would have been present after the ALU stage, there would have been 2 stalls. Generally, if the target address is present after the kth stage, then there will be (k – 1) stalls in the pipeline.

Total number of stalls introduced in the pipeline due to branch instructions = **Branch frequency * Branch Penalty**

Data Dependency (Data Hazard)

Let us consider an ADD instruction S, such that

S : ADD R1, R2, R3

Addresses read by S = I(S) = {R2, R3}

Addresses written by S = O(S) = {R1}

Now, we say that instruction S2 depends in instruction S1, when

$$[I(S1) \cap O(S2)] \cup [O(S1) \cap I(S2)] \cup [O(S1) \cap O(S2)] \neq \phi$$

This condition is called Bernstein condition.

Three cases exist:

- **Flow (data) dependence**: O(S1) ∩ I (S2), S1 → S2 and S1 writes after something read by S2
- **Anti-dependence**: I(S1) ∩ O(S2), S1 → S2 and S1 reads something before S2 overwrites it
- **Output dependence**: O(S1) ∩ O(S2), S1 → S2 and both write the same memory location.

Flow dependency

A Flow dependency, also known as a data dependency or true dependency or read-after-write (RAW), occurs when an instruction depends on the result of a previous instruction:

```
1. A = 3
2. B = A
3. C = B
```

Instruction 3 is truly dependent on instruction 2, as the final value of C depends on the instruction updating B. Instruction 2 is truly dependent on instruction 1, as the final value of B depends on the instruction updating A. Since instruction 3 is truly dependent upon instruction 2 and instruction 2 is truly dependent on instruction 1, instruction 3 is also truly dependent on instruction 1. Instruction level parallelism is therefore not an option in this example. [1]

Anti-dependency

An anti-dependency, also known as write-after-read (WAR), occurs when an instruction requires a value that is later updated. In the following example, instruction 2 anti-depends on instruction 3 — the ordering of these instructions cannot be changed, nor can they be executed in parallel (possibly changing the instruction ordering), as this would affect the final value of A.

```
1. B = 3
2. A = B + 1
3. B = 7
```

An anti-dependency is an example of a name dependency. That is, renaming of variables could remove the dependency, as in the next example:

```
1. B = 3
N. B2 = B
2. A = B2 + 1
3. B = 7
```

A new variable, B2, has been declared as a copy of B in a new instruction, instruction N. The anti-dependency between 2 and 3 has been removed, meaning that these instructions may now be executed in parallel. However, the

modification has introduced a new dependency: instruction 2 is now truly dependent on instruction N, which is truly dependent upon instruction 1. As flow dependencies, these new dependencies are impossible to safely remove. [1]

Output dependency

An output dependency, also known as write-after-write (WAW), occurs when the ordering of instructions will affect the final output value of a variable. In the example below, there is an output dependency between instructions 3 and 1 — changing the ordering of instructions in this example will change the final value of A, thus these instructions cannot be executed in parallel.

```
1. B = 3
2. A = B + 1
3. B = 7
```

As with anti-dependencies, output dependencies are name dependencies. That is, they may be removed through renaming of variables, as in the below modification of the above example:

```
1. B2 = 3
2. A = B2 + 1
3. B = 7
```

A commonly used naming convention for data dependencies is the following: Read-after-Write or RAW (flow dependency), Write-After-Read or WAR (anti-dependency), or Write-after-Write or WAW (output dependency).

Example: Let there be two instructions I1 and I2 such that:
I1 : ADD R1, R2, R3
I2 : SUB R4, R1, R2
When the above instructions are executed in a pipelined processor, then data dependency condition will occur, which means that I_2 tries to read the data before I_1 writes it, therefore, I_2 incorrectly gets the old value from I_1.

INSTRUCTION / CYCLE	1	2	3	4
I_1	IF	ID	EX	DM
I_2		IF	ID(Old value)	EX

To minimize data dependency stalls in the pipeline, **operand forwarding** is used.
Operand Forwarding : In operand forwarding, we use the interface registers present between the stages to hold intermediate output so that dependent instruction can access new value from the interface register directly.
Considering the same example:
I1 : ADD R1, R2, R3
I2 : SUB R4, R1, R2

INSTRUCTION / CYCLE	1	2	3	4
I_1	IF	ID	EX	DM
I_2		IF	ID	EX

Control Dependency

An instruction B has a control dependency on a preceding instruction A if the outcome of A determines whether B should be executed or not. In the following example, the instruction **S2** has a control dependency on instruction **S1** . However, **S3** does not depend on **S1** because **S3** is always executed irrespective of the outcome of **S1**.

```
S1.    if (a == b)
S2.        a = a + b
S3.    b = a + b
```

Intuitively, there is control dependence between two statements A and B if

- B could be possibly executed after A
- The outcome of the execution of A will determine whether B will be executed or not.

A typical example is that there are control dependences between the condition part of an if statement and the statements in its true/false bodies.
A formal definition of control dependence can be presented as follows:
A statement **S2** is said to be control dependent on another statement **S1** iff

- there exists a path **P** from **S1** to **S2** such that every statement **Si** ≠ **S1** within **P** will be followed by **S2** in each possible path to the end of the program and
- **S1** will not necessarily be followed by **S2**, i.e. there is an execution path from **S1** to the end of the program that does not go through **S2**.

Expressed with the help of (post-)dominance the two conditions are equivalent to

- **S2** post-dominates all **Si**
- **S2** does not post-dominate **S1**

Construction of Control Dependences

Control dependences are essentially the dominance frontier in the reverse graph of the control flow graph (CFG).[2] Thus, one way of constructing them, would be to construct the post-dominance frontier of the CFG, and then reversing it to obtain a control dependence graph.
The following is a pseudo-code for constructing the post-dominance frontier:

```
for each X in a bottom-up traversal of the dominator tree do:
  PostDominanceFrontier(X) ← ∅
  for each Y ∈ Predecessors(X) do:
    if immediatePostDominator(Y) ≠ X:
      then PostDominanceFrontier(X) ← PostDominanceFrontier(X) ∪ {Y}
  done
  for each Z ∈ Children(X) do:
    for each Y ∈ PostDominanceFrontier(Z) do:
      if immediatePostDominator(Y) ≠ X:
        then PostDominanceFrontier(X) ← PostDominanceFrontier(X) ∪ {Y}
    done
  done
```

Here, Children(X) is the set of nodes in the CFG that are post-dominated by X, and Predecessors(X) are the set of nodes in the CFG that directly precede X in the CFG.
Once the post-dominance frontier map is computed, reversing it will result in a map from the nodes in the CFG to the nodes that have a control dependence on them.

Data Hazards

Data hazards occur when instructions that exhibit data dependence, modify data in different stages of a pipeline. Hazard cause delays in the pipeline. There are mainly three types of data hazards:

1) RAW (Read after Write) [Flow/True data dependency]
2) WAR (Write after Read) [Anti-Data dependency]
3) WAW (Write after Write) [Output data dependency]
Let there be two instructions I and J, such that J follow I. Then,

- RAW hazard occurs when instruction J tries to read data before instruction I writes it.
  Eg:
  I: R2 <- R1 + R3
  J: R4 <- R2 + R3
- WAR hazard occurs when instruction J tries to write data before instruction I reads it.
  Eg:
  I: R2 <- R1 + R3
  J: R3 <- R4 + R5
- WAW hazard occurs when instruction J tries to write output before instruction I writes it.
  Eg:
  I: R2 <- R1 + R3
  J: R2 <- R4 + R5

WAR and WAW hazards occur during the out-of-order execution of the instructions.

Types of pipeline

- Uniform delay pipeline
  In this type of pipeline, all the stages will take same time to complete an operation.
  In uniform delay pipeline, **Cycle Time (Tp) = Stage Delay**

If buffers are included between the stages then, **Cycle Time (Tp) = Stage Delay + Buffer Delay**

- Non-Uniform delay pipeline
  In this type of pipeline, different stages take different time to complete an operation.
  In this type of pipeline, Cycle Time (Tp) = Maximum(Stage Delay)

For example, if there are 4 stages with delays, 1 ns, 2 ns, 3 ns, and 4 ns, then
Tp = Maximum(1 ns, 2 ns, 3 ns, 4 ns) = 4 ns
If buffers are included between the stages,
Tp = Maximum(Stage delay + Buffer delay)

Example : Consider a 4 segment pipeline with stage delays (2 ns, 8 ns, 3 ns, 10 ns). Find the time taken to execute 100 tasks in the above pipeline.
Solution : As the above pipeline is a non-linear pipeline,
Tp = max(2, 8, 3, 10) = 10 ns
We know that $ET_{pipeline} = (k + n - 1)\ Tp = (4 + 100 - 1)\ 10\ ns = 1030\ ns$
NOTE: MIPS = Million instructions per second

Performance of pipeline with stalls

A stall causes the pipeline performance to degrade the ideal performance.

$$\textbf{Speedup from pipelining} = \frac{\textbf{Average instruction time unpipelined}}{\textbf{Average instruction time pipelined}}$$

$$= \frac{CPI_{unpipelined} * \text{Clock Cycle Time}_{unpipelined}}{CPI_{pipelined} * \text{Clock Cycle Time}_{pipelined}}$$

The ideal CPI on a pipelined machine is almost always 1. Hence, the pipelined CPI is

$$CPI_{pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}$$
$$= 1 + \text{Pipeline stall clock cycles per instruction}$$

Partha Sarathi

If we ignore the cycle time overhead of pipelining and assume the stages are all perfectly balanced, then the cycle time of the two machines are equal and

$$\textbf{Speedup} = \frac{\textbf{CPI}_{\text{unpipelined}}}{\textbf{1+ Pipeline stall cycles per instruction}}$$

If all instructions take the same number of cycles, which must also equal the number of pipeline stages (the depth of the pipeline) then unpipelined CPI is equal to the depth of the pipeline, leading to

$$\textbf{Speedup} = \frac{\textbf{Pipeline depth}}{\textbf{1 + Pipeline stall cycles per instruction}}$$

If there are no pipeline stalls, this leads to the intuitive result that pipelining can improve performance by the depth of pipeline.

Superscalar Processors

It was first invented in 1987. It is a machine which is designed to improve the performance of the scalar processor. In most applications, most of the operations are on scalar quantities. Superscalar approach produces the high performance general purpose processors.

The main principle of superscalar approach is that it executes instructions independently in different pipelines. As we already know, that Instruction pipelining leads to parallel processing thereby speeding up the processing of instructions. In Superscalar processor, multiple such pipelines are introduced for different operations, which further improves parallel processing.

There are multiple functional units each of which is implemented as a pipeline. Each pipeline consists of multiple stages to handle multiple instructions at a time which support parallel execution of instructions.

It increases the throughput because the CPU can execute multiple instructions per clock cycle. Thus, superscalar processors are much faster than scalar processors.

A **scalar processor** works on one or two data items, while the **vector processor** works with multiple data items.

A **superscalar processor** is a combination of both. Each instruction processes one data item, but there are multiple execution units within each CPU thus multiple instructions can be processing separate data items concurrently.

While a superscalar CPU is also pipelined, there are two different performance enhancement techniques. It is possible to have a non-pipelined superscalar CPU or pipelined non-superscalar CPU. The superscalar technique is associated with some characteristics, these are:

1. Instructions are issued from a sequential instruction stream.
2. CPU must dynamically check for data dependencies.
3. Should accept multiple instructions per clock cycle.

Superscalar Architecture

- Superscalar Architecture (SSA) describes a microprocessor design that execute more than one instruction at a time during a single clock cycle
- The design is sometimes called "Second Generation RISC". Another term used to describe superscalar processors is multiple instruction issue processors.
- In a SSA design, the processor or the instruction compiler is able to determine whether an instruction can be carried out independently of other sequential instructions, or whether it has a dependency on another instruction and must be executed sequentially.
- In a SSA, several scalar instructions can be initiated simultaneously and executed independently.
- A long series of innovations aimed at producing ever- faster microprocessors.
- Includes all features of pipelining but, in addition, there can be several instructions executing simultaneously in the same pipeline stage.

Partha Sarathi

- SSA introduces a new level of parallelism, called instruction-level parallelism.

Superscalar CPU Architecture

- In Superscalar CPU Architecture implementation of Instruction Level Parallelism (ILP) within a single processor allows faster CPU at a given clock rate.
- A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to functional units.
- Each functional unit is not a separate CPU core but an execution resource within a single CPU such as an arithmetic logic unit, a bit shifter, or a multiplier

SimpleScalar Architecture

- SimpleScalar is an open source computer architecture simulator which is written using 'C' programming language.
- A set of tools that model a virtual computer system with CPU, Cache and Memory Hierarchy.
- Using the tool, users can model applications that simulate programs running on a range of modern processors and systems.
- The tool set includes sample simulators ranging from a fast functional simulator to a detailed.

Scalar to Superscalar

- The simplest processors are scalar processors. Each instruction executed by a scalar processor typically manipulates one or two data items at a time.
- In a superscalar CPU the dispatcher reads instructions from memory and decides which one can be run in parallel.
- Therefore a superscalar processor can be proposed having multiple parallel pipelines, each of which is processing instructions simultaneously from a single instruction thread.

Pipelining in Superscalar Architecture

- In a superscalar processor, the simple operation latency should require only one cycle, as in the base scalar processor
- In order to fully utilise a superscalar processor of degree m with pipelining, m instructions must be executable parallely. This situation may not be true in all clock cycles.
- Pipelining is the process of breaking down task into substeps and executing them in different parts of processor.

Implement Superscalar

- A SSA processor fetches multiple instructions at a time, and attempts to find nearby instructions that are independent of each other and therefore can be executed in parallel.
- Based on the dependency analysis, the processor may issue and execute instructions in an order that differs from that of the original machine code.
- The processor may eliminate some unnecessary dependencies by the use of additional registers.

Partha Sarathi

Superscalar with Scalar Instructions Flow

Scalar Processor

Fetch	I1	I2	I3	I4			
Decode		I1	I2	I3	I4		
Execute			I1	I2	I3	I4	
Write-back				I1	I2	I3	I4

Super-Scalar Processor

Fetch	I1	I3	I5	I7			
	I2	I4	I6	I8			
Decode		I1	I3	I5	I7		
		I2	I4	I6	I8		
Execute			I1	I3	I5	I7	
			I2	I4	I6	I8	
Write-back				I1	I3	I5	I7
				I2	I4	I6	I8

I: Instructions from 1 to corresponding sequences

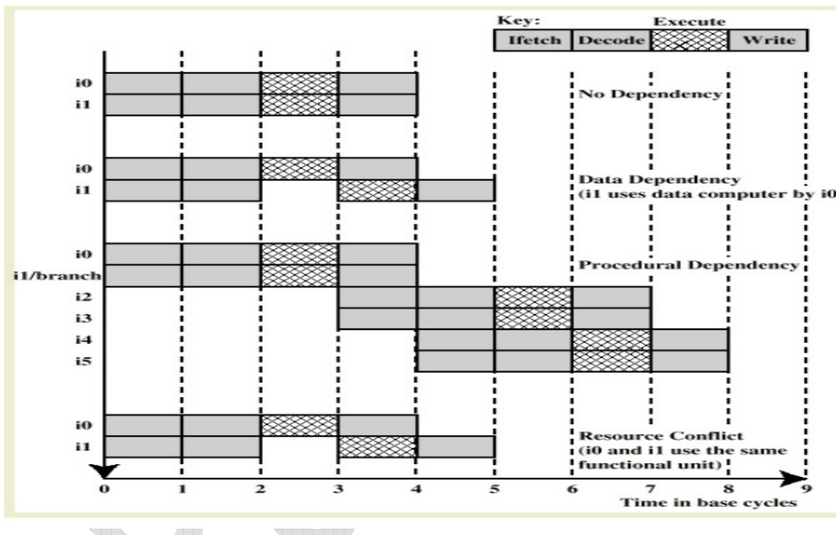
Fetch: fetches instructions from memory (ideally one per cycle for Scalar)

Decode: reveals instruction operations to be performed and identifies the resources needed

Execute: actual processing of operations as indicated by instruction

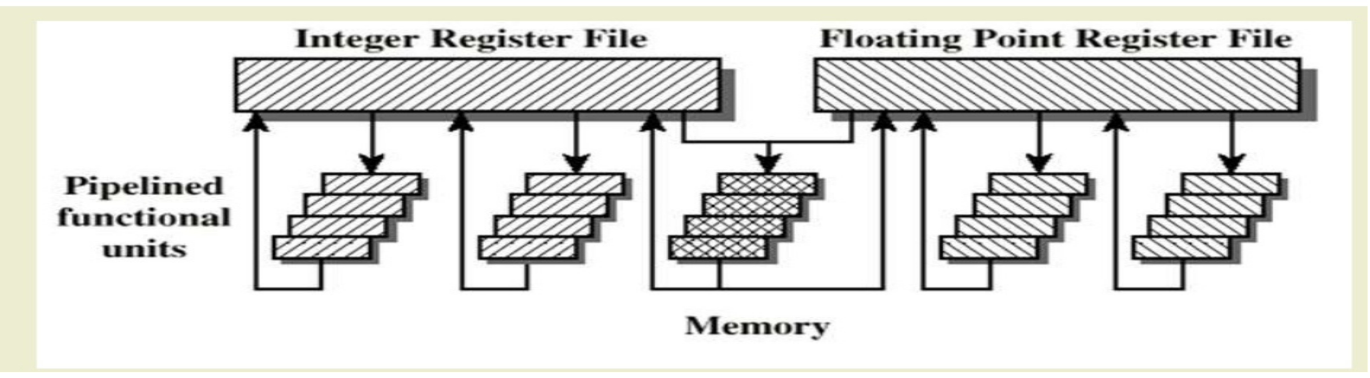
Store (Write Back): writing results into the registers.

Effect of Dependencies

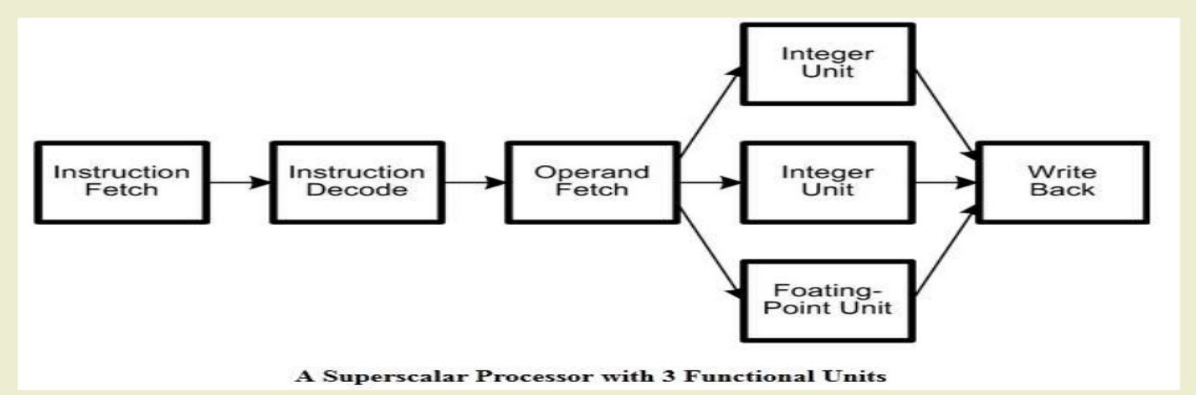


- ADD r1, r2 (r1 := r1+r2;)
- MOVE r3,r1 (r3 := r1;)
- Can fetch and decode second instruction in parallel with first
- Can NOT execute second instruction until first is finished

General Superscalar Organization

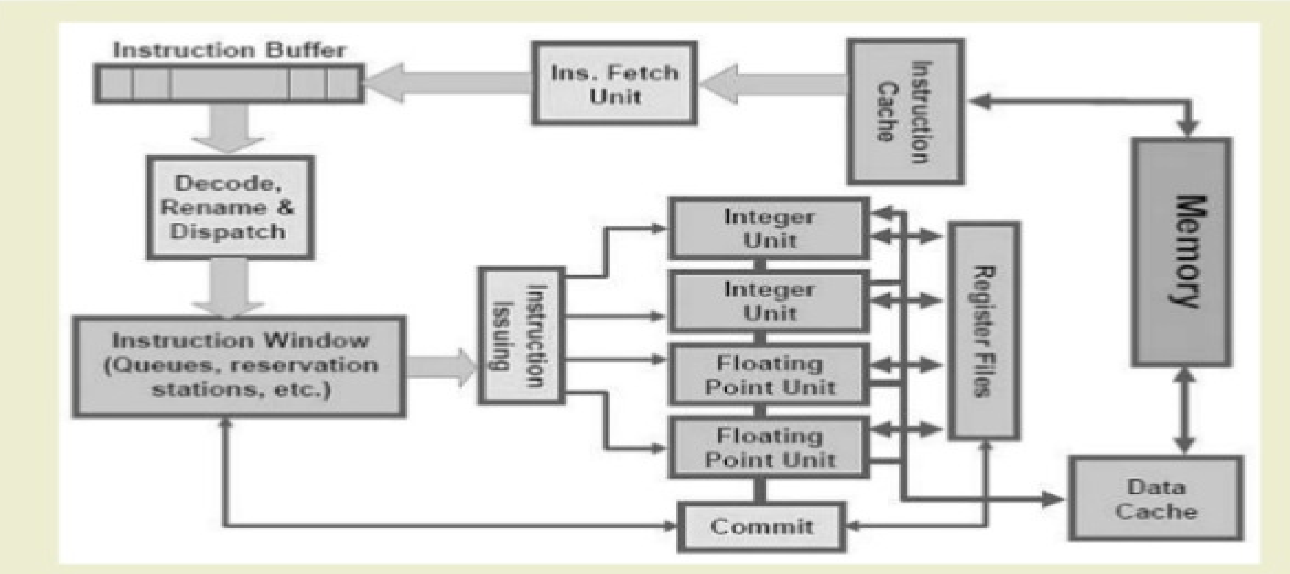


Superscalar Operational Block Diagram



A Superscalar Processor with 3 Functional Units

Instruction Flow in Superscalar Architecture



Superpipelining

- Superpipeling is based on dividing the stages into several sub- stages, and thus increasing the number of instructions which are handled by the pipeline at the same time.
- For example, by dividing each stage into two sub-stages, a pipeline can perform at twice the speed in the ideal situation:
    - ➢ No duplication of hardware is needed for these stages. 14 Superpipelining Figure: Duplication of hardware is for Superscalar
    - ➢ Tasks that require less than half a clock cycle.

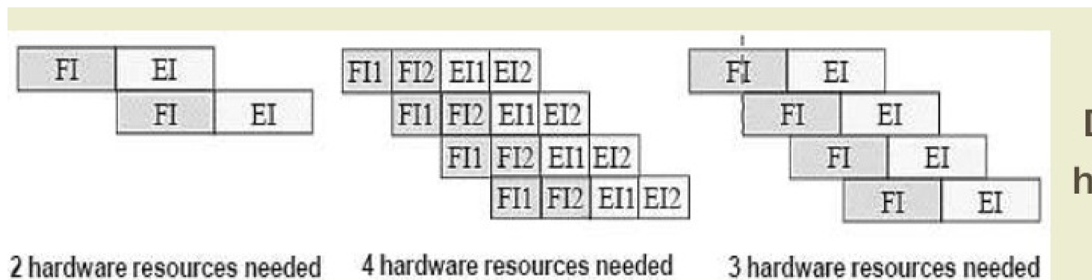
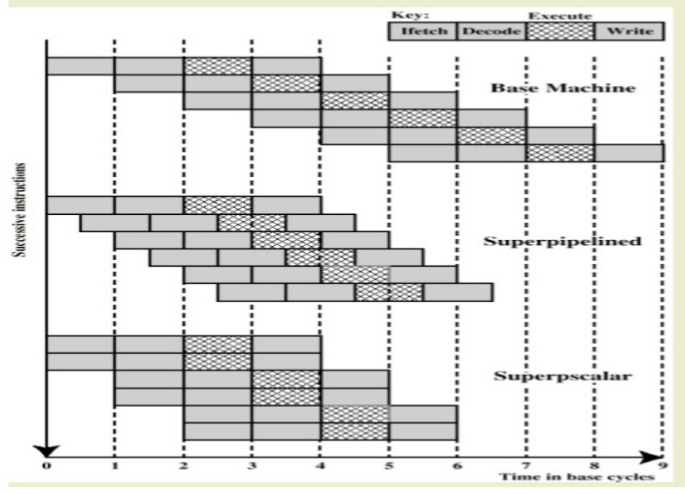


Figure: Duplication of hardware is for Superscalar

Superscalar vs. Superpipeline



Base machine: 4-stage pipeline

- Instruction fetch
- Operation decode
- Operation execution
- Result write back

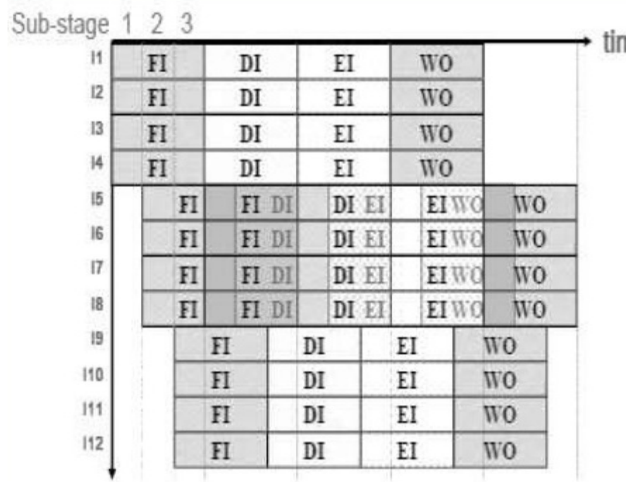
Superpipeline of degree 2

- A sub-stage often takes half a clock cycle to finish.

Superscalar of degree 2

- Two instructions are executed.
- Duplication of hardware is required by definition.

Superpipelined Superscalar



Superpipeline of degree 3 and superscalar of degree 4:

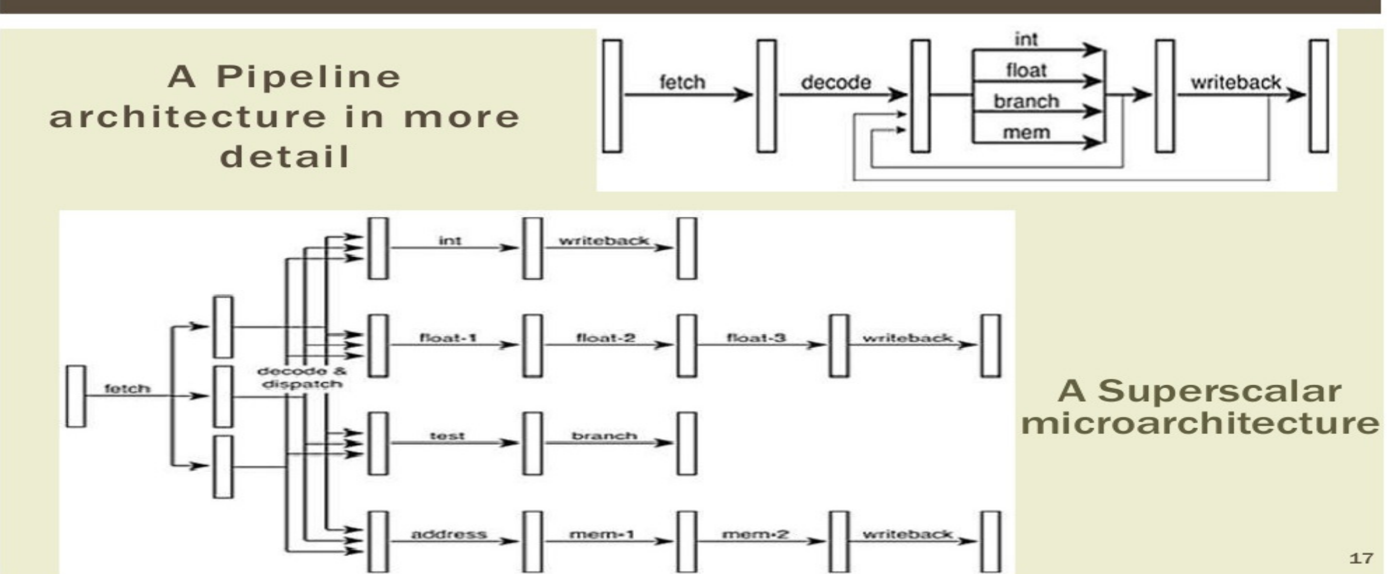
- 12 times speed-up over the base machine.
- 48 times speedup over sequential execution.

This is a new trend of architecture design:

- Pentium Pro(P6): 3-degree superscalar, 12-stage “superpipeline”.
- PowerPC 620: 4-degree superscalar, 4/6-stage pipeline.

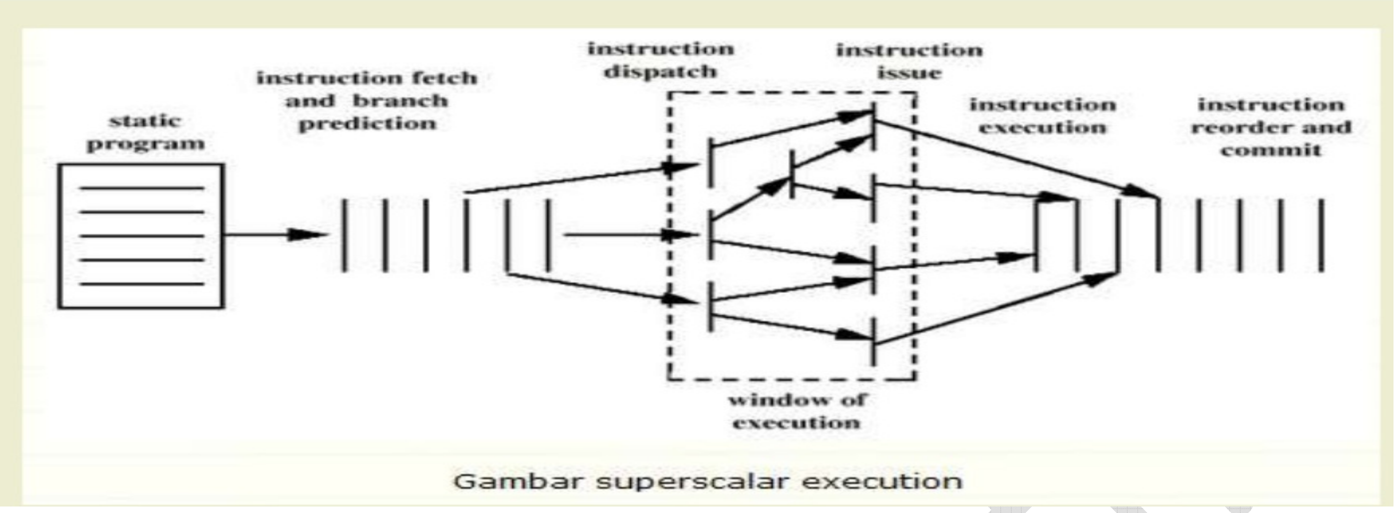
Instruction Flow

A Pipeline architecture in more detail



A Superscalar microarchitecture

Instruction Execution



Gambar superscalar execution

Superscalar Issues to Consider

- Tasks can be divided into the following
  - Parallel decoding
  - Superscalar instruction issue
  - Parallel instruction execution
- preserving sequential consistency of exception processing.
- preserving sequential consistency of execution.
- Parallel decoding – more complex task for scalar processors
- Parallel instruction execution task – While instructions are executed in parallel, instructions are usually. completed out of order in respect to a sequential operating procedure.
- Superscalar instruction issue – A higher issue rate gives rise to higher processor performance, but amplifies the restrictive effects of control and data dependencies on the processor performance.

Two way Superscalar Execution

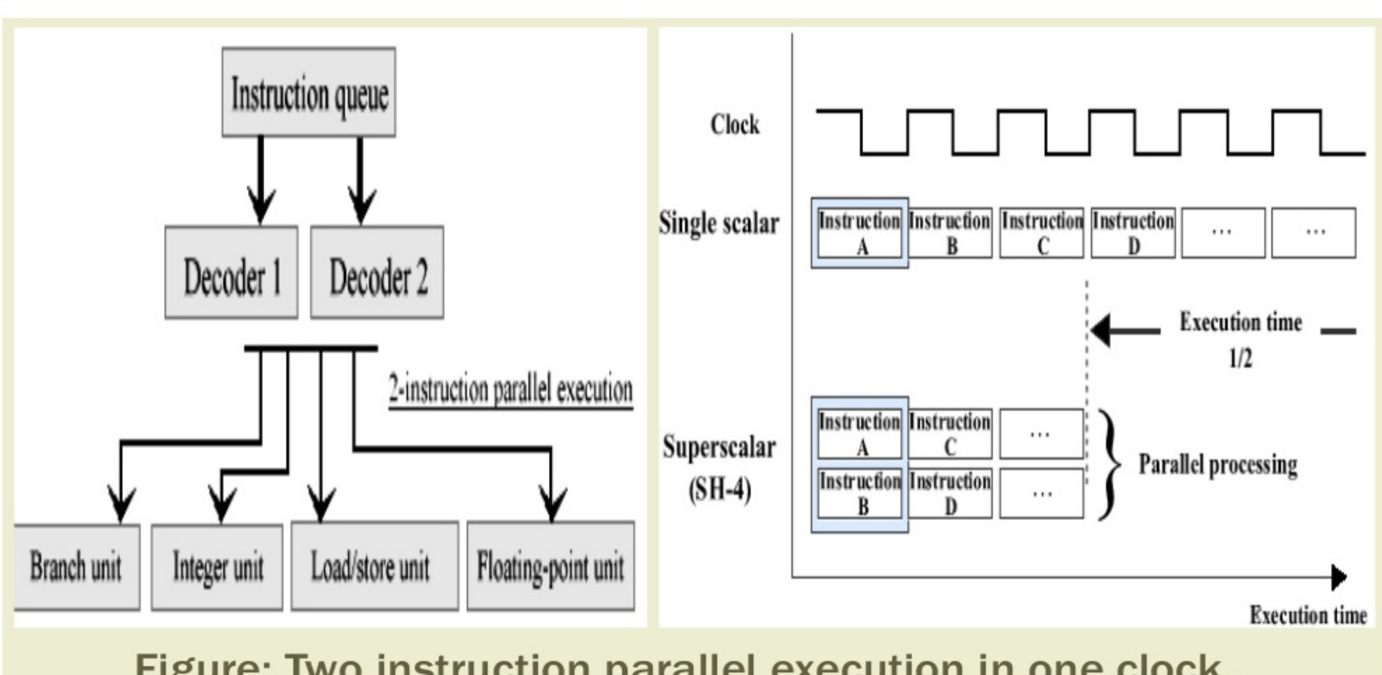


Figure: Two instruction parallel execution in one clock.

Limitations of Superscalar

- Instruction-fetch inefficiencies caused by both branch delays and instruction misalignment.
- not worthwhile to explore highly- concurrent execution hardware, rather, it is more appropriate to explore economical execution hardware.
- degree of intrinsic parallelism in the instruction stream (instructions requiring the same computational resources from the CPU).
- complexity and time cost of the dispatcher and associated dependency checking logic.
- branch instruction processing.

VLIW PROCESSORS

- Very long instruction word or VLIW refers to a processor architecture designed to take advantage of instruction level parallelism.
    - Instruction of a VLIW processor consists of multiple independent operations grouped together.
    - There are Multiple Independent Functional Units in VLIW processor architecture.
    - Each operation in the instruction is aligned to a functional unit.
    - All functional units share the use of a common large register file.
- This type of processor architecture is intended to allow higher performance without the inherent complexity of some other approaches.

Different Approaches

Other approaches to improving performance in processor architectures :

- Pipelining: Breaking up instructions into sub-steps so that instructions can be executed partially at the same time.
- Superscalar architectures: Dispatching individual instructions to be executed completely independently in different parts of the processor.
- Out-of-order execution: Executing instructions in an order different from the program

Instruction Level Parallelism (ILP)

- Instruction-level parallelism (ILP) is a measure of how many of the operations in a computer program can be performed simultaneously.
- The overlap among instructions is called instruction level parallelism.
- Ordinary programs are typically written under a sequential execution model where instructions execute one after the other and in the order specified by the programmer.
- Goal of compiler and processor designers implementing ILP is to identify and take advantage of as much ILP as possible.

Consider the following program:

```
op 1 e = a + b
op2 f = c + d
op3 m = e * f
```

Operation 3 depends on the results of operations 1 and 2, so it cannot be calculated until both of them are completed.However, operations 1 and 2 do not depend on any other operation, so they can be calculated simultaneously.If we assume that each operation can be completed in one unit of time then these three instructions can be completed in a total of two units of time. Giving an ILP of 3/2.

VLIW Compiler

- Compiler is responsible for static scheduling of instructions in VLIW processor.
- Compiler finds out which operations can be executed in parallel in the program.
- It groups together these operations in single instruction which is the very large instruction word.

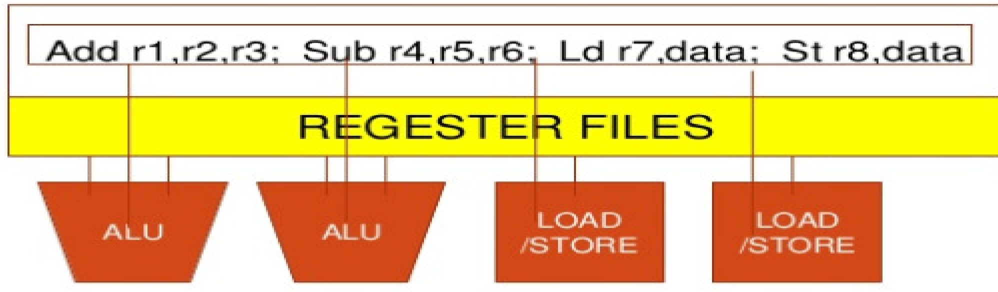
Partha Sarathi

- Compiler ensures that an operation is not issued before its operands are ready.

VLIW Instruction

- One VLIW instruction word encodes multiple operations which allows them to be initiated in a single clock cycle.
- The operands and the operation to be performed by the various functional units are specified in the instruction itself.
- One instruction encodes at least one operation for each execution unit of the device.
- So length of the instruction increases with the number of execution units
- To accommodate these operation fields, VLIW instructions are usually at least 64 bits wide, and on some architectures are much wider up to 1024 bits.

VLIW Instruction

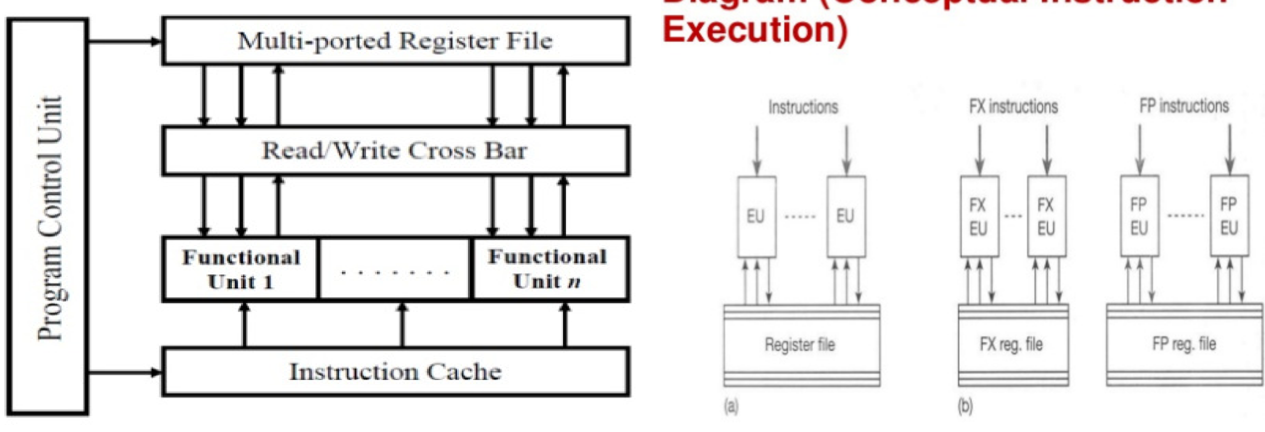


ILP in VLIW

Consider the computation of **y = a1x1 + a2x2 + a3x3**

On a sequential processor	On the VLIW processor with 2 load/store units, 1 multiply unit and 1 add unit
cycle 1: load a1 cycle 2: load x1 cycle 3: load a2 cycle 4: load x2 cycle 5: multiply z1 a1 x1 cycle 6: multiply z2 a2 x2 cycle 7: add y z1 z2 cycle 8: load a3 cycle 9: load x3 cycle 10: multiply z1 a3 x3 cycle 11: add y y z2	cycle 1: load a1 load x1 cycle 2: load a2 load x2 Multiply z1 a1 x1 cycle 3: load a3 load x3 Multiply z2 a2 x2 cycle 4: multiply z3 a3 x3 add y z1 z2 cycle 5: add y y z3
requires 11 cycles	requires 5 cycles

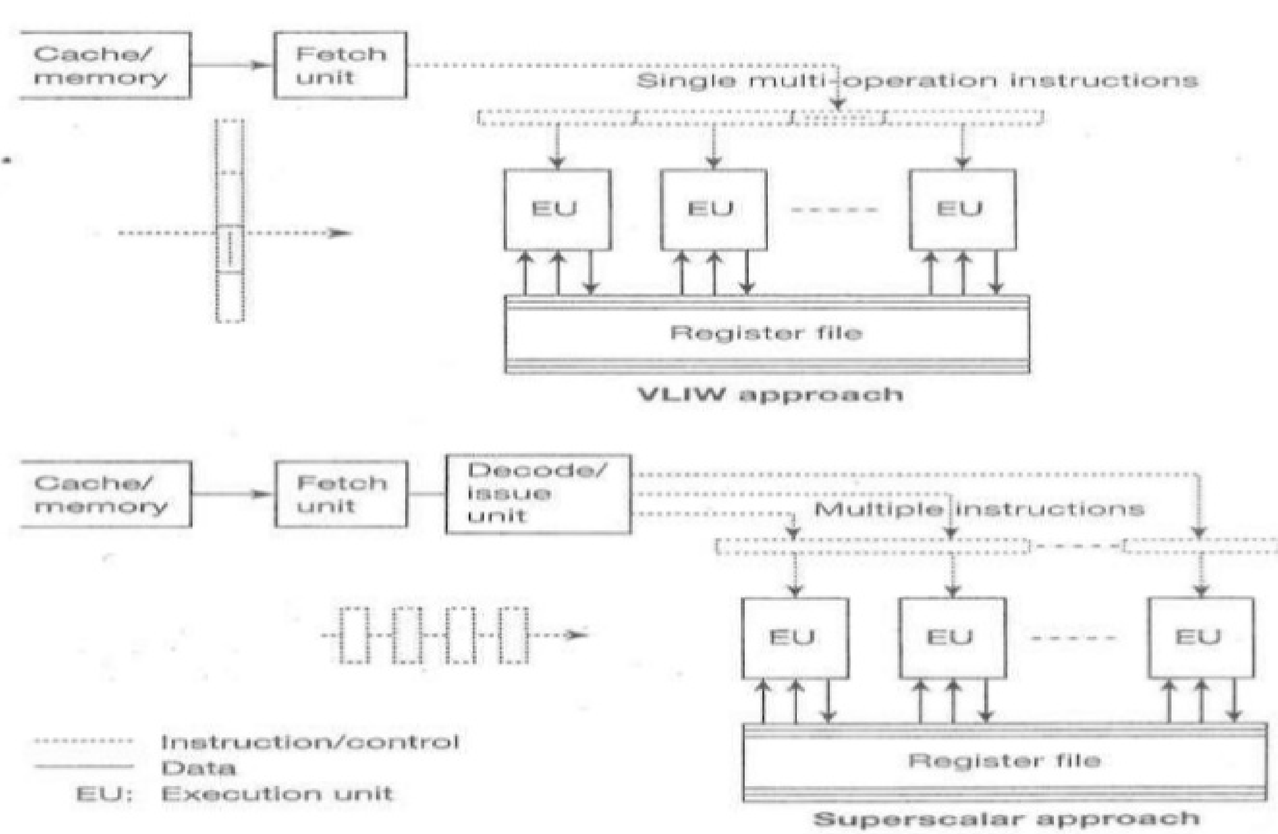
Block Diagram



Working

- Long instruction words are fetched from the memory.
- A common multi-ported register file for fetching the operands and storing the results.
- Parallel random access to the register file is possible through the read/write cross bar.
- Execution in the functional units is carried out concurrently with the load/store operation of data between RAM and the register file.
- One or multiple register files for FX and FP data.
- Rely on compiler to find parallelism and schedule dependency free program code.

Difference Between VLIW & Superscalar Architecture



VLIW vs. Superscalar Architecture

Instruction formulation

- **Superscalar**: Receive conventional instructions conceived for sequential processors.
- **VLIW:** Receive long instruction words, each comprising a field (or opcode) for each execution unit. Instruction word length depends number of execution units and code length to control each unit (such as opcode length, registers). Typical word length is 64 – 1024 bits, much longer than conventional machine word length

Instruction scheduling

Superscalar:

- Done dynamically at run-time by the hardware.
- Data dependency is checked and resolved in hardware.
- Need a look ahead hardware window for instruction fetch.

VLIW:

- Done statically at compile time by compiler.
- Data dependency is checked by compiler.
- In case of un-filled opcodes in a VLIW, memory space and instruction bandwidth are wasted.

ARCHITECTURE CHARACTERISTIC	CISC	RISC	VLIW
INSTRUCTION SIZE	Varies	One size, usually 32 bits	One size
INSTRUCTION FO RMAT	Field placement varies	Regular, consistent placement of fields	Regular, consistent placement of fields
INSTRUCTION SEMANTICS	Varies from simple to complex; possibly many dependent operations per instruction	Almost always one simple operation	Many simple, independent operations
REGISTERS	Few, sometimes special	Many, general-purpose	Many, general-purpose
MEMORY REFE RENCES	Bundled with operations in many different types of instructions	Not bundled with operations, i.e., load/store architecture	Not bundled with operations, i.e., load/store architecture
HARDWARE DESIGN FOCUS	Exploit microcoded implementations	Exploit implementations with one pipeline and & no microcode	Exploit implementations with multiple pipelines, no microcode & no complex dispatch logic
PICTURE OF FIVE TYPICAL INSTRU CTIONS ▭ = 1 BYTE			

Advantages of VLIW

- Dependencies are determined by compiler and used to schedule according to function unit latencies .
- Function units are assigned by compiler and correspond to the position within the instruction packet.
- Reduces hardware complexity.
  - ➢ Tasks such as decoding, data dependency detection, instruction issues etc. becoming simple.
  - ➢ Ensures potentially higher Clock Rate.
  - ➢ Ensures Low power consumption

Disadvantages of VLIW

- **Higher complexity of the compiler**
- **Compatibility across implementations** : Compiler optimization needs to consider technology dependent parameters such as latencies and load-use time of cache.
- **Unscheduled events (e.g. cache miss) stall** entire processor .
- **Code density**: In case of un-filled opcodes in a VLIW, memory space and instruction bandwidth are wasted i.e. low slot utilization.
- **Code expansion**: Causes high power consumption

Applications

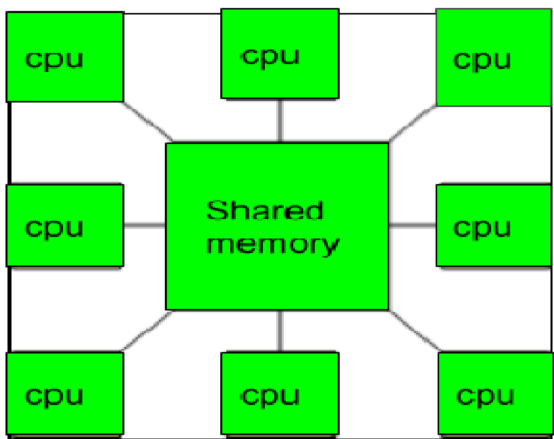
- VLIW architecture is suitable for Digital Signal Processing applications.
- Processing of media data like compression/decompression of Image and speech data.

Examples of VLIW processor

- **VLIW Mini supercomputers**: Multiflow TRACE 7/300, 14/300, 28/300 Multiflow TRACE /500 Cydrome Cydra 5 IBM Yorktown VLIW Computer
- **Single-Chip VLIW Processors**: Intel iWarp, Philip's LIFE Chips
- **Single-Chip VLIW Media (through-put) Processors**: Trimedia, Chromatic, Micro-Unity
- **DSP Processors** (TI TMS320C6x)

Mod-2 Multiprocessor:

A Multiprocessor is a computer system with two or more central processing units (CPUs) share full access to a common RAM. The main objective of using a multiprocessor is to boost the system's execution speed, with other objectives being fault tolerance and application matching.
There are two types of multiprocessors, one is called shared memory multiprocessor and another is distributed memory multiprocessor. In shared memory multiprocessors, all the CPUs shares the common memory but in a distributed memory multiprocessor, every CPU has its own private memory.



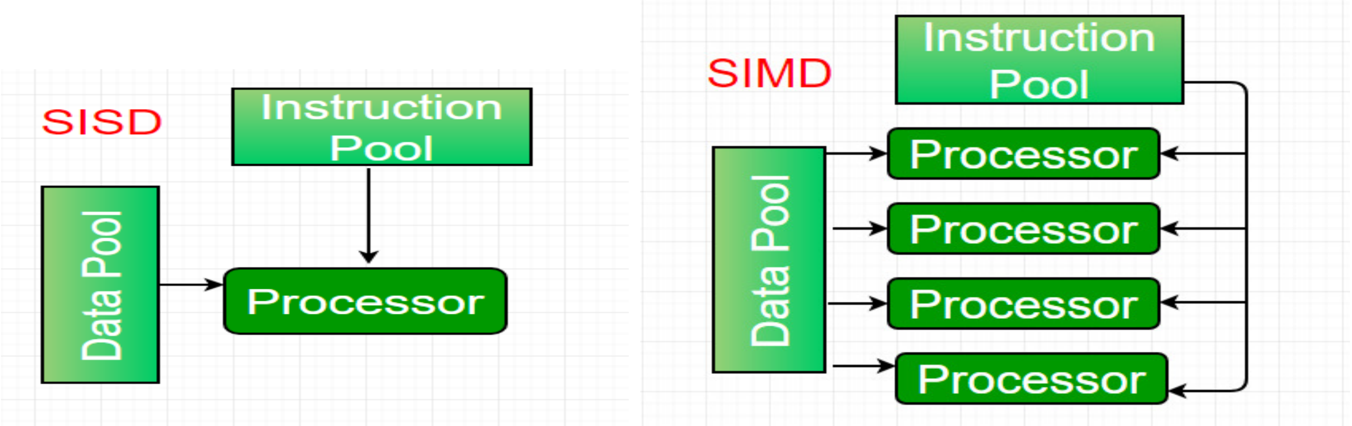
Parallel computing is a computing where the jobs are broken into discrete parts that can be executed concurrently. Each part is further broken down to a series of instructions. Instructions from each part execute simultaneously on different CPUs. Parallel systems deal with the simultaneous use of multiple computer resources that can include a single computer with multiple processors, a number of computers connected by a network to form a parallel processing cluster or a combination of both.
Parallel systems are more difficult to program than computers with a single processor because the architecture of parallel computers varies accordingly and the processes of multiple CPUs must be coordinated and synchronized.

Flynn's taxonomy is a classification of computer architectures, proposed by Michael J. Flynn in 1966. The classification system has stuck, and has been used as a tool in design of modern processors and their functionalities. Since the rise of multiprocessing central processing units (CPUs), a multiprogramming context has evolved as an extension of the classification system.The four classifications defined by Flynn are based upon the number of concurrent instruction (or control) streams and data streams available in the architecture.

SISD single instruction, single data; i.e., a conventional uni-processor
SIMD single instruction, multiple data; like MMX or SSE instructions in the x86 processor series, processor arrays and pipelined vector processors
MISD multiple instruction, single data; very rare but one example is the U.S. Space Shuttle flight controller
MIMD multiple instruction, multiple data; SMPs, clusters. This is the most common multiprocessor

Single-instruction, single-data (SISD) systems –
An SISD computing system is a uniprocessor machine which is capable of executing a single instruction, operating on a single data stream. In SISD, machine instructions are processed in a sequential manner and computers adopting this model are popularly called sequential computers. Most conventional computers have SISD architecture. All the instructions and data to be processed have to be stored in primary memory.
The speed of the processing element in the SISD model is limited(dependent) by the rate at which the computer can transfer information internally. Dominant representative SISD systems are IBM PC, workstations.

Partha Sarathi



Single-instruction, multiple-data (SIMD) systems –

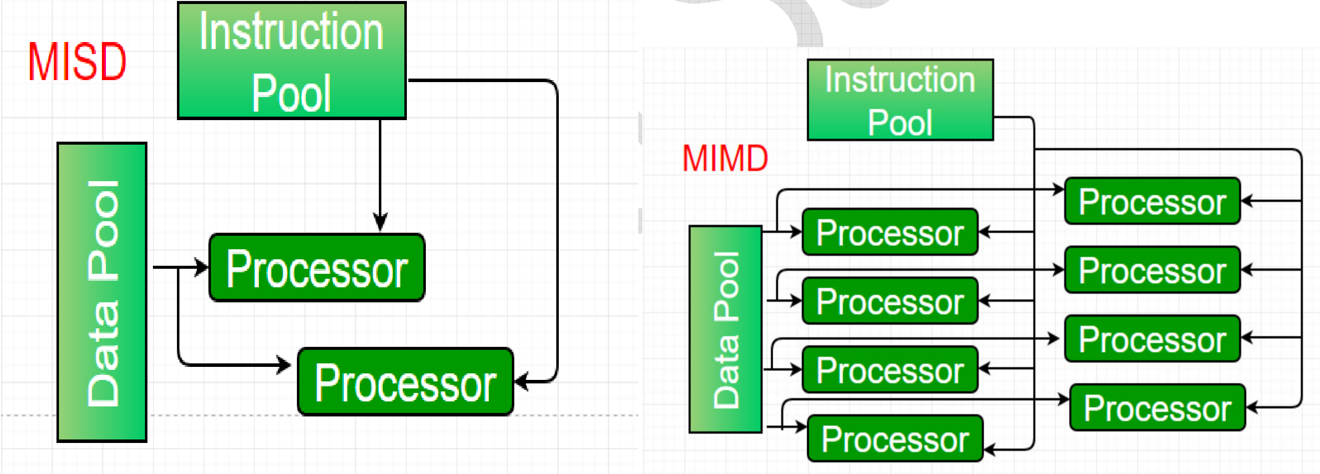
An SIMD system is a multiprocessor machine capable of executing the same instruction on all the CPUs but operating on different data streams. Machines based on an SIMD model are well suited to scientific computing since they involve lots of vector and matrix operations. So that the information can be passed to all the processing elements (PEs) organized data elements of vectors can be divided into multiple sets(N-sets for N PE systems) and each PE can process one data set. Dominant representative SIMD systems is Cray's vector processing machine.

Multiple-instruction, single-data (MISD) systems –

An MISD computing system is a multiprocessor machine capable of executing different instructions on different PEs but all of them operating on the same dataset .

Example Z = sin(x)+cos(x)+tan(x)

The system performs different operations on the same data set. Machines built using the MISD model are not useful in most of the application, a few machines are built, but none of them are available commercially.



Multiple-instruction, multiple-data (MIMD) systems –

An MIMD system is a multiprocessor machine which is capable of executing multiple instructions on multiple data sets. Each PE in the MIMD model has separate instruction and data streams; therefore machines built using this model are capable to any kind of application. Unlike SIMD and MISD machines, PEs in MIMD machines work asynchronously.

MIMD machines are broadly categorized into **shared-memory MIMD** and **distributed-memory MIMD** based on the way PEs are coupled to the main memory.

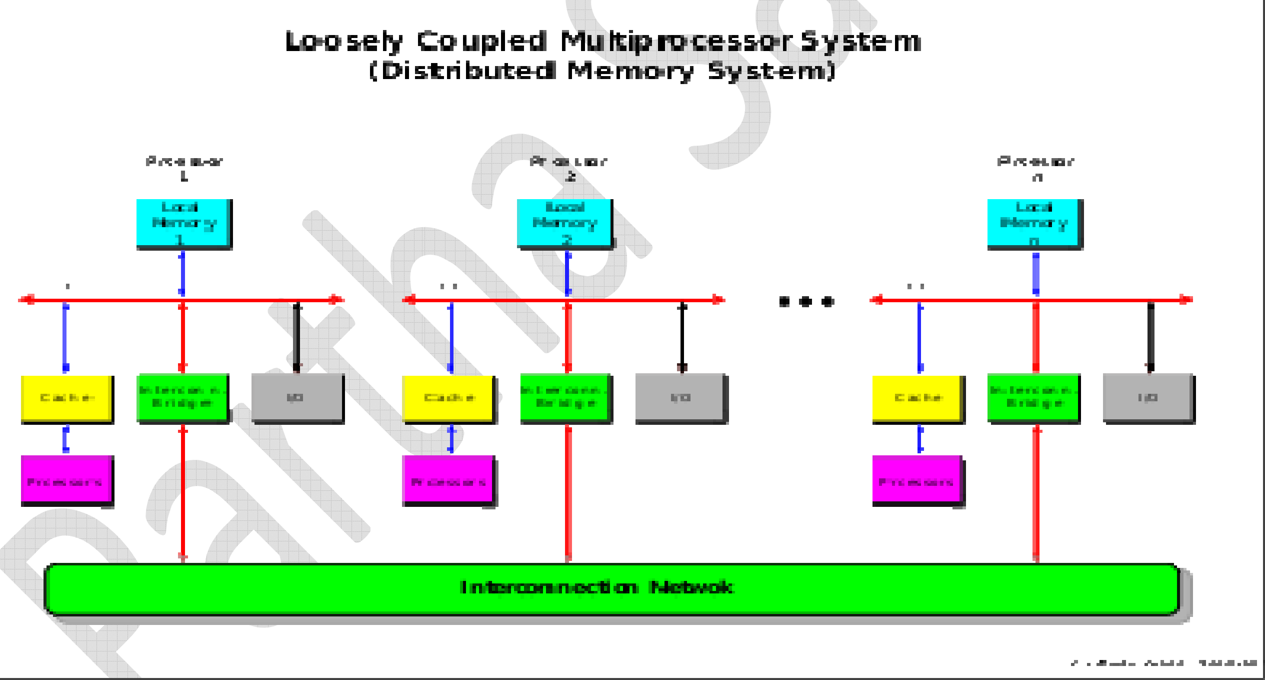
In the **shared memory MIMD** model (tightly coupled multiprocessor systems), all the PEs are connected to a single global memory and they all have access to it. The communication between PEs in this model takes place through the shared memory, modification of the data stored in the global memory by one PE is visible to all other PEs. Dominant representative shared memory MIMD systems are Silicon Graphics machines and Sun/IBM's SMP (Symmetric Multi-Processing).

In **Distributed memory MIMD** machines (loosely coupled multiprocessor systems) all PEs have a local memory. The communication between PEs in this model takes place through the interconnection network (the inter process communication channel, or IPC). The network connecting PEs can be configured to tree, mesh or in accordance with the requirement.

The shared-memory MIMD architecture is easier to program but is less tolerant to failures and harder to extend with respect to the distributed memory MIMD model. Failures in a shared-memory MIMD affect the entire system, whereas this is not the case of the distributed model, in which each of the PEs can be easily isolated. Moreover, shared memory MIMD architectures are less likely to scale because the addition of more PEs leads to memory contention. This is a situation that does not happen in the case of distributed memory, in which each PE has its own memory. As a result of practical outcomes and user's requirement , distributed memory MIMD architecture is superior to the other existing models.

Loosely-coupled (distributed memory) multiprocessor system
In loosely-coupled multiprocessor systems, each processor has its own local memory, input/output (I/O) channels, and operating system. Processors exchange data over a high-speed communication network by sending messages via a technique known as "message passing". Loosely-coupled multiprocessor systems are also known as distributed-memory systems, as the processors do not share physical memory and have individual I/O channels.
System characteristics

- These systems are able to perform multiple-instructions-on-multiple-data (MIMD) programming.
- This type of architecture allows parallel processing.
- The distributed memory is highly scalable.



Message passing is a technique for invoking behaviour (i.e., running a program) on a computer. The invoking program sends a message to a process (which may be an actor or object) and relies on the process and the supporting infrastructure to select and invoke the actual code to run. Message passing differs from conventional programming where a process, subroutine, or function is directly invoked by name. Message passing is key to some models of concurrency and object-oriented programming.

Message passing is used ubiquitously in modern computer software. It is used as a way for the objects that make up a program to work with each other and as a means for objects and systems running on different computers (e.g., the Internet) to interact. Message passing may be implemented by various mechanisms, including channels.

Partha Sarathi

Synchronous message passing
Synchronous message passing occurs between objects that are running at the same time. It is used by object-oriented programming languages such as Java and Smalltalk.
Synchronous messaging is analogous to a synchronous function call; just as the function caller waits until the function completes, the sending process waits until the receiving process completes. This can make synchronous communication unworkable for some applications. For example, large, distributed systems may not perform well enough to be usable. Such large, distributed systems may need to operate while some of their subsystems are down for maintenance, etc.
Imagine a busy business office having 100 desktop computers that send emails to each other using synchronous message passing exclusively. One worker turning off their computer can cause the other 99 computers to freeze until the worker turns their computer back on to process a single email.

Asynchronous message passing
With asynchronous message passing the receiving object can be down or busy when the requesting object sends the message. Continuing the function call analogy, it is like a function call that returns immediately, without waiting for the called function to complete. Messages are sent to a queue where they are stored until the receiving process requests them. The receiving process processes its messages and sends results to a queue for pickup by the original process (or some designated next process).
Asynchronous messaging requires additional capabilities for storing and retransmitting data for systems that may not run concurrently, and are generally handled by an intermediary level of software (often called middleware); a common type being Message-oriented middleware (MOM).
The buffer required in asynchronous communication can cause problems when it is full. A decision has to be made whether to block the sender or whether to discard future messages. A blocked sender may lead to deadlock. If messages are dropped, communication is no longer reliable.

Hybrids
Synchronous communication can be built on top of asynchronous communication by using a Synchronizer. For example, the α-Synchronizer works by ensuring that the sender always waits for an acknowledgement message from the receiver. The sender only sends the next message after the acknowledgement has been received. On the other hand, asynchronous communication can also be built on top of synchronous communication. For example, modern micro-kernels generally only provide a synchronous messaging primitive [citation needed] and asynchronous messaging can be implemented on top by using helper threads.

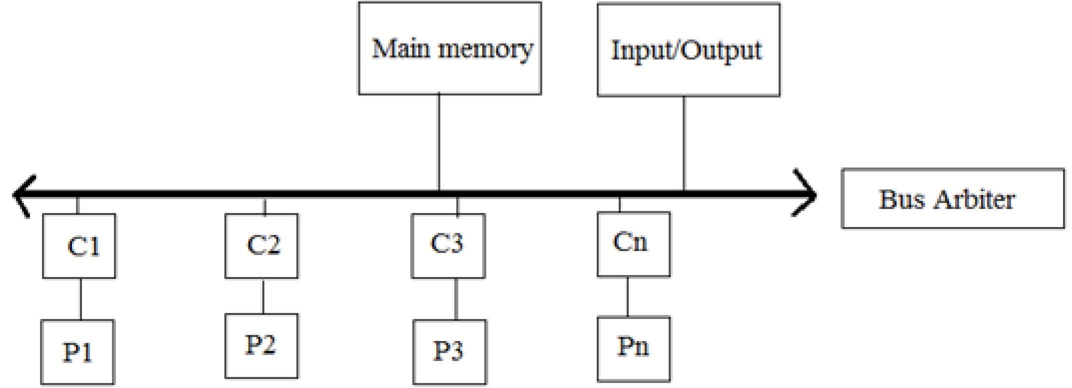
Tightly-coupled (shared memory) multiprocessor system
Multiprocessor system with a shared memory closely connected to the processors. A symmetric multiprocessing system is a system with centralized shared memory called main memory (MM) operating under a single operating system with two or more homogeneous processors
Multiprocessor system is divided into following basic architectures:

1. Symmetric Multiprocessor System (SMP)
2. UMA (Uniform Memory Access)
3. NUMA (Non-Uniform Memory Access)

1. Symmetric Multiprocessor System (SMP)
In this architecture, two or more processors are connected to the same memory. It has full access to input and output devices with the same rights. Only one instance of the operating system runs all processors equally. No one processor will treat as a special. Most of the multiprocessors use SMP architecture. SMP structure is given below.

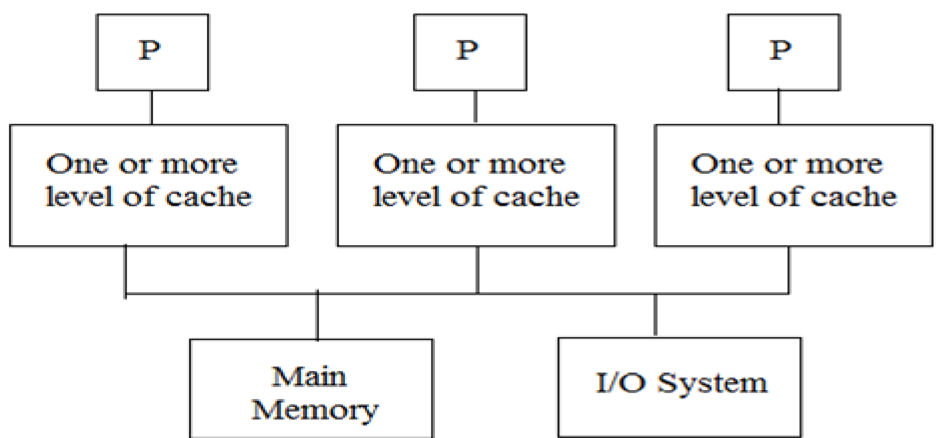
Partha Sarathi

SMP Architecture



SMP has a tightly coupled system because a number of homogeneous processors running independently of each other. That means each processor running different programs and uses different data sets. Above figure shows the pool of processors each one having own cache and sharing the common main memory as well as common i/o devices. When CPU wants to read the memory, it first checks the bus is idle or not. If the bus is idle, it puts the address of the bus it wants then it activates certain signals and waits for memory to put the required word on the bus. But if the bus is busy, the CPU has to wait.
To solve this problem, the cache is used with a processor. Due to this many reads can be possible. There is much less bus traffic and the system can support more CPUs.

2. **Uniform Memory Access (UMA)**
In this type of architecture, all processors share the common (Uniform) centralized primary memory. Each CPU has the same memory access time. This system also called as shared memory multiprocessor (SMM). In the figure below each processor has a cache at one or more level. And also shares a common memory as well as input output systems.
UMA architecture

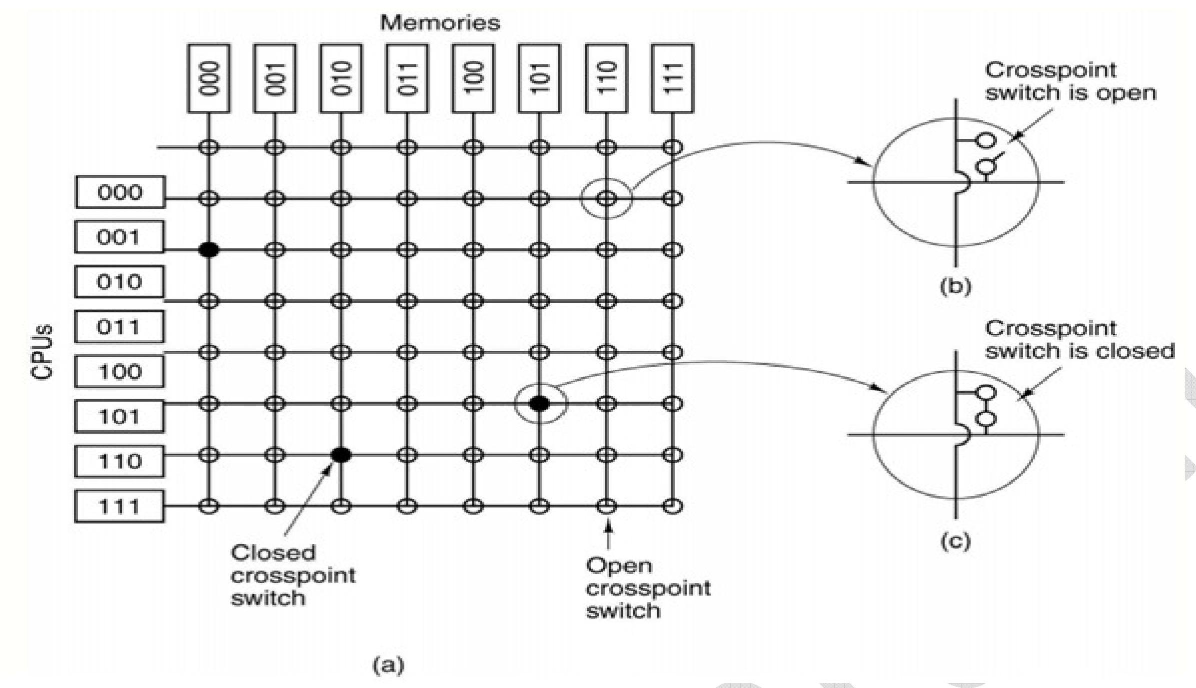


There are three types of UMA
I) Using a cross based switch
II) Using a multistage interconnection network
III) Using bus based symmetric multiprocessor

I) Using a cross based switch
In Normal structure, we can extend the size upto only 16 CPU limits. But sometimes we need to extend the limit. So, we required a different kind of interconnection networks. One of them is simple circuit crossbar which connects 'n' CPUs to 'k' memories. It is used mostly in telephone switches. Each intersection has a cross point which has a switch for closing and opening purpose. It is one of the non-blocking networks. E.g. Sun Enterprise 1000 uses this

technology. It consists of a single cabinet with up to 64 CPUs. The crossbar switch is packaged on a circuit board with eight plugs in slots on each side.

UMA cross based switch

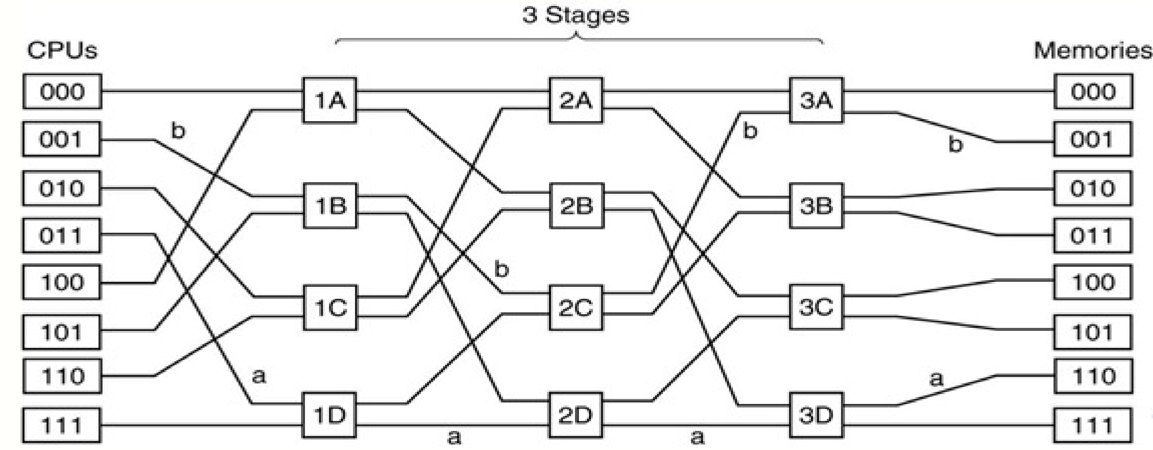


In the above figure, there are the cross-connection switching of multiple CPUs and memories. Each cross point has a switch which is either opened or closed (in (b) and (c)). The drawback of this architecture is 'n' number of CPUs and 'k' number of memory required n2 switches. i.e. 1000 CPU needs 1000000 switches. But the main advantage is any CPU can access any available memory with less blocking

II) Using multistage interconnection network

To go beyond the Sun Enterprise of limit 1000, we need powerful interconnection network. 2 x 2 switches can be used to build up the large network. Example of this technology is the Omega network. In this system, the wired pattern is shuffle perfectly. Each memory has given labels which used to find the route in the network. The Omega network is a blocking network. Requests come in a sequence but cannot be served concurrently. Conflicts can arise in using a connection or a switch, in accessing a memory block or in answering a CPU request. Many techniques are used to minimize conflicts.

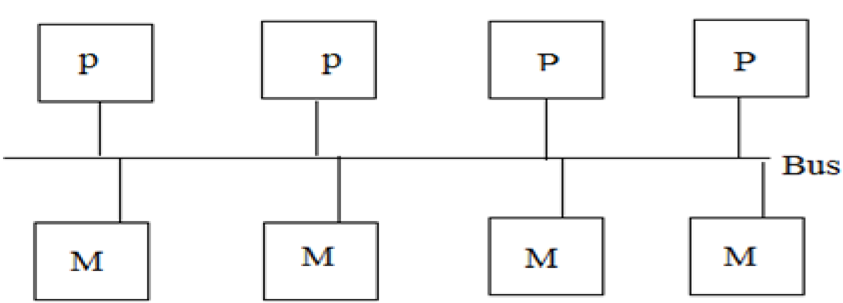
Multistage interconnection network



In the above figure, 8 CPUs are connected to 8 memories, using 12 switches laid out in three stages. In generally n CPUs and n memories require log2n stages and n/2 switch per stage, giving a total of (n/2) log2n.

III) Using bus based symmetric multiprocessor

Bus based symmetric multiprocessor



The simplest multiprocessors system consists of a single bus. Two or more CPUs and one or more memory modules all use the same bus for communication. If the bus is busy when a CPU wants to access memory, it must wait. Adding more CPUs results in more waiting. This can be mitigated to some degree by including processor cache support.

3. NUMA (Non-Uniform Memory Access)

If we want to improve the limit of the number of CPUs, UMA is not the best option. Because it accesses the memory uniformly. To solve this problem we have another technique called Non-Uniform Memory Access (NUMA). They share the single address space (local memory) through all the CPU for improving the result. But provides faster local access than remote access.

UMA programs can run without change in NUMA machines but performance may be slow. Memory access time depends on the memory location which is relative to a processor. The processor can access it's own local memory fastly than it's non-local memory. NUMA is used in a symmetric multiprocessing system.

SMP is obviously tightly coupled and shares everything in multiple processors by using a single operating system.

NUMA has three key properties.

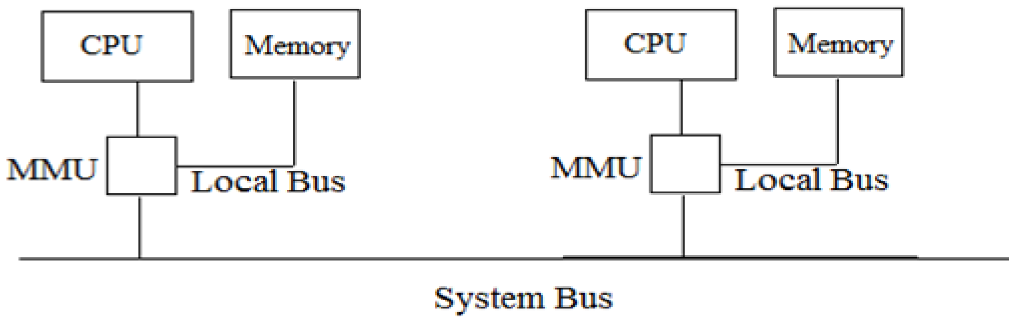
i. Access to remote memory is possible.

ii. Accessing remote memory is slower than local memory.

iii. Remote access time is not hidden by the caching.

The following figure shows the structure of NUMA. In this CPU and memory is connected to MMU (Memory management unit) via local bus and local memory connected to the system bus via local bus.

NUMA Structure



It is used in SMP. Multiprocessor system without NUMA create the problem of cache missed and only one processor can access the computer memory at a time. To solve this problem, separate memory is provided for each processor.

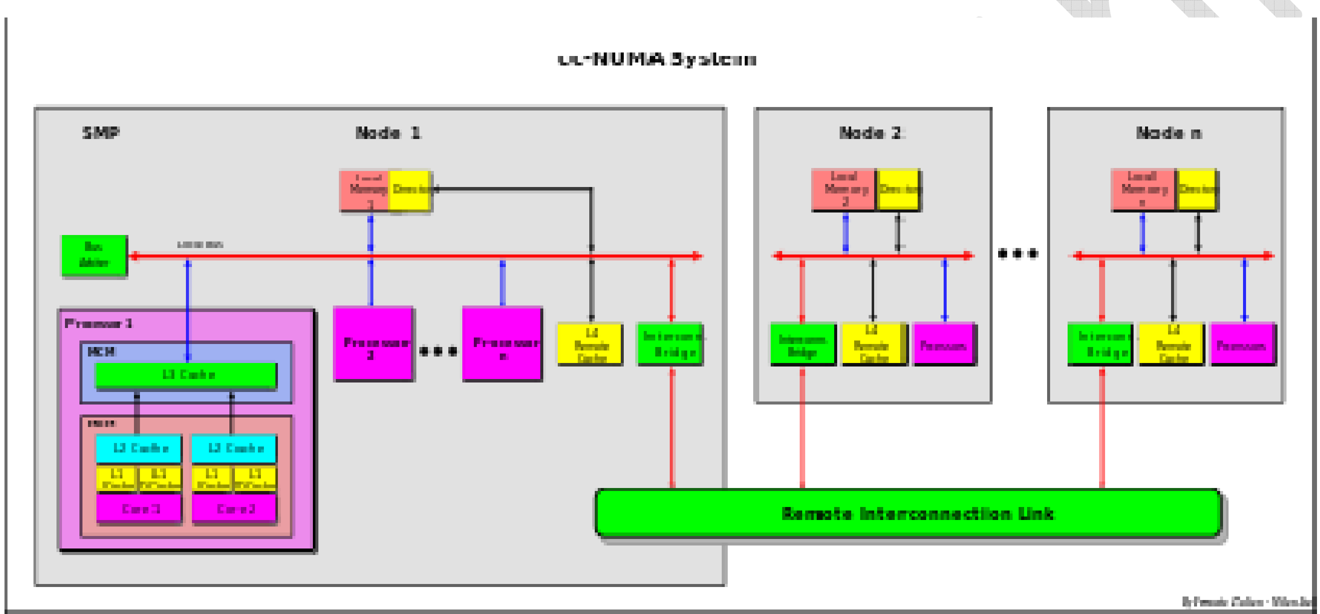
There are 2 types of NUMA

i. NC-NUMA:

When the time to accessing a remote is not hidden, the system is called as NC-NUMA (Non-Caching-Non-Uniform Memory Access). In this type of NUMA, processors have no local cache. Cache coherence problem is not present in NC-NUMA. Each memory item is in a single location. Remote memory access is difficult so this system helps to the software's those are relocated memory pages from one block to another just for improving the performance. A page scanner demon activates every few seconds, examines statistics on memory usage, and moves pages from one block to another. The Figure is the NC-NUMA type structure.

ii. CC- NUMA:

When a coherent cache is present, the system is called as CC-NUMA (Cache Coherent-Non Uniform Memory Access). CC-NUMA uses the directory based protocol rather than snooping. The basic idea is to manage each node in the system with a directory for its RAM blocks. A database tells that in which cache is located a RAM block, and what is its state.



Difference between UMA and NUMA:

S.NO	UMA	NUMA
1.	UMA stands for Uniform Memory Access.	NUMA stands for Non-uniform Memory Access.
2.	In Uniform Memory Access, Single memory controller is used.	In Non-uniform Memory Access, Different memory controller is used.
3.	Uniform Memory Access is slower than non-uniform Memory Access.	Non-uniform Memory Access is faster than uniform Memory Access.
4.	Uniform Memory Access has limited bandwidth.	Non-uniform Memory Access has more bandwidth than uniform Memory Access.
5.	Uniform Memory Access is applicable for general purpose applications and time-sharing applications.	Non-uniform Memory Access is applicable for real-time applications and time-critical applications.
6.	In uniform Memory Access, memory	In non-uniform Memory Access, memory

	access time is balanced or equal.	access time is not equal.
7.	There are 3 types of buses used in uniform Memory Access which are: Single, Multiple and Crossbar.	While in non-uniform Memory Access, There are 2 types of buses used which are: Tree and hierarchical.

Difference between loosely coupled and tightly coupled multiprocessor system:

S.NO	LOOSELY COUPLED	TIGHTLY COUPLED
1.	There is distributed memory in loosely coupled multiprocessor system.	There is shared memory, in tightly coupled multiprocessor system.
2.	Loosely Coupled Multiprocessor System has low data rate.	Tightly coupled multiprocessor system has high data rate.
3.	The cost of loosely coupled multiprocessor system is less.	Tightly coupled multiprocessor system is more costly.
4.	In loosely coupled multiprocessor system, modules are connected through Message transfer system network.	While there is PMIN, IOPIN and ISIN networks.
5.	In loosely coupled multiprocessor, Memory conflicts don't take place.	While tightly coupled multiprocessor system have memory conflicts.

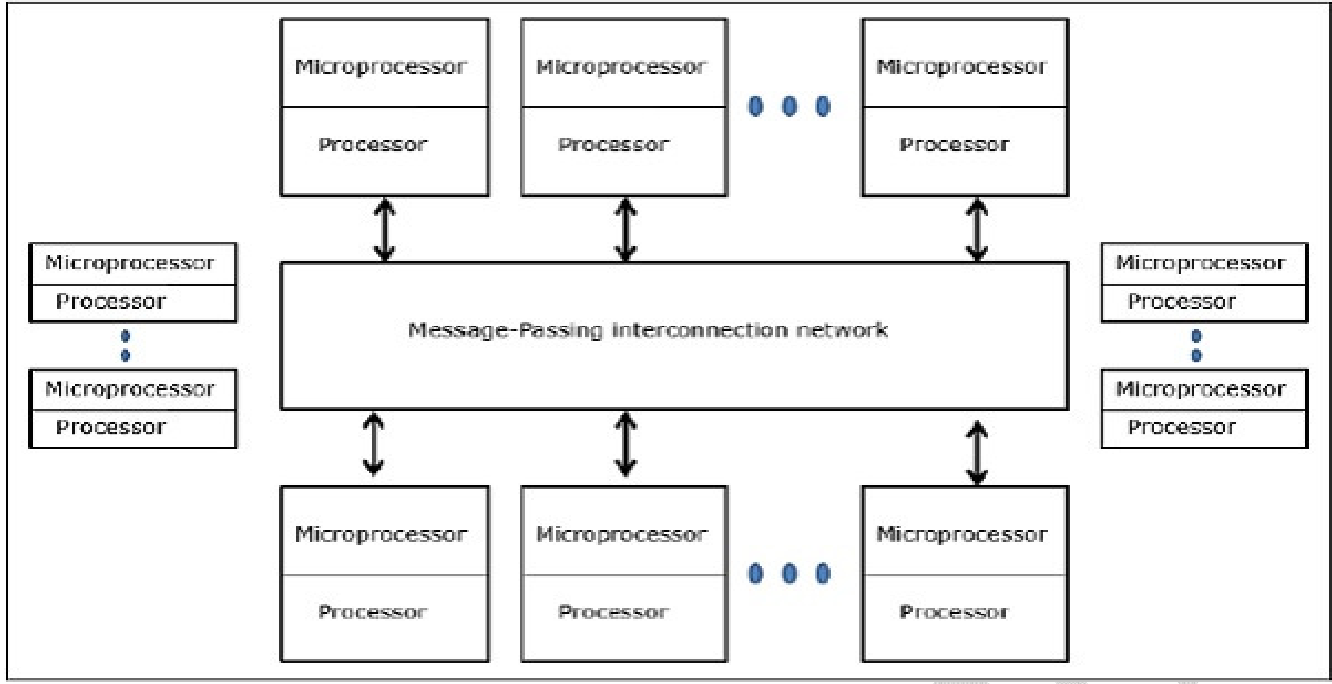
Distributed memory

Distributed memory refers to a multiprocessor computer system in which each processor has its own private memory. Computational tasks can only operate on local data, and if remote data is required, the computational task must communicate with one or more remote processors. In contrast, a shared memory multiprocessor offers a single memory space used by all processors. Processors do not have to be aware where data resides, except that there may be performance penalties, and that race conditions are to be avoided.

In a distributed memory system there is typically a processor, a memory, and some form of interconnection that allows programs on each processor to interact with each other. The interconnect can be organised with point to point links or separate hardware can provide a switching network. The network topology is a key factor in determining how the multiprocessor machine scales. The links between nodes can be implemented using some standard network protocol (for example Ethernet), using bespoke network links (used in for example the Transputer), or using dual-ported memories.

Distributed - Memory Multicomputers – A distributed memory multicomputer system consists of multiple computers, known as nodes, inter-connected by message passing network. Each node acts as an autonomous computer having a processor, a local memory and sometimes I/O devices. In this case, all local memories are private and are accessible only to the local processors. This is why, the traditional machines are called no-remote-memory-access (NORMA) machines.

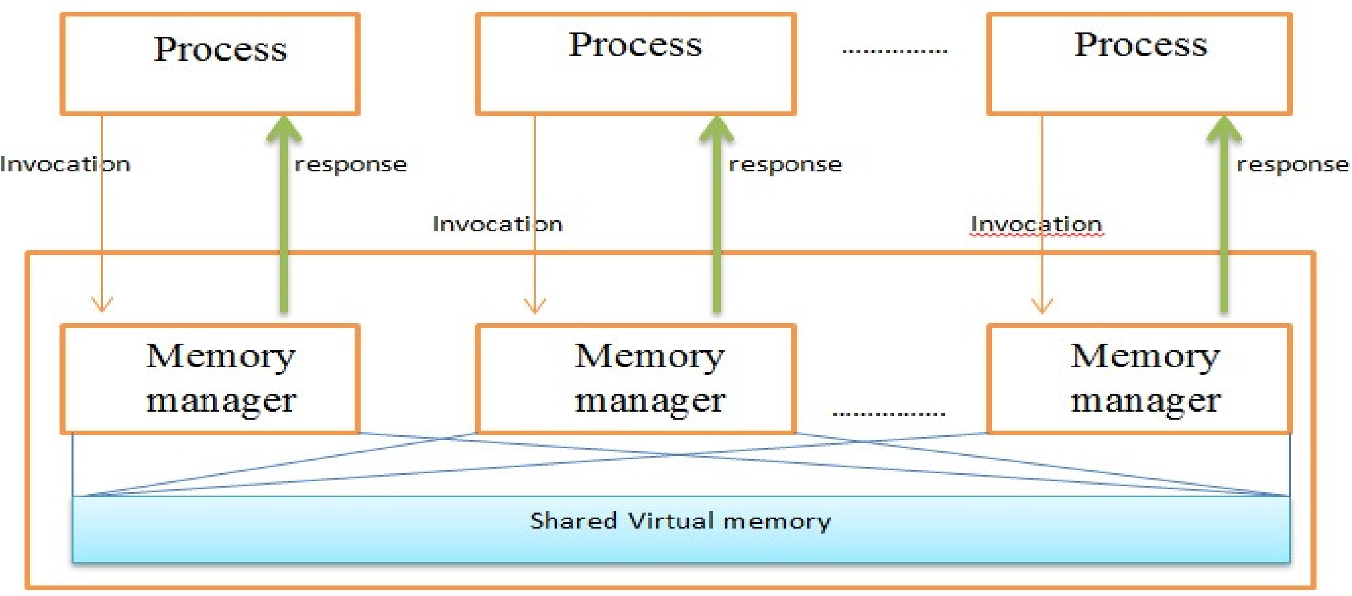
Partha Sarathi



Distributed shared memory

Distributed shared memory (**DSM**) is a form of memory architecture where physically separated memories can be addressed as one logically shared address space. Here, the term "shared" does not mean that there is a single centralized memory, but that the address space is "shared" (same physical address on two processors refers to the same location in memory). **Distributed global address space** (**DGAS**), is a similar term for a wide class of software and hardware implementations, in which each node of a cluster has access to shared memory in addition to each node's non-shared private memory.

A distributed-memory system, often called a multicomputer, consists of multiple independent processing nodes with local memory modules which is connected by a general interconnection network. Software DSM systems can be implemented in an operating system, or as a programming library and can be thought of as extensions of the underlying virtual memory architecture. When implemented in the operating system, such systems are transparent to the developer; which means that the underlying distributed memory is completely hidden from the users.



Distributed shared memory

Message Passing vs. DSM

Message passing	Distributed shared memory
Variables have to be marshalled	Variables are shared directly
Cost of communication is obvious	Cost of communication is invisible
Processes are protected by having private address space	Processes could cause error by altering data
Processes should execute at the same time	Executing the processes may happen with non-overlapping lifetimes

Advantages

- Scales well with a large number of nodes
- Message passing is hidden
- Can handle complex and large databases without replication or sending the data to processes
- Generally cheaper than using a multiprocessor system
- Provides large virtual memory space
- Programs are more portable due to common programming interfaces
- Shield programmers from sending or receiving primitives

Disadvantages

- Generally slower to access than non-distributed shared memory
- Must provide additional protection against simultaneous accesses to shared data
- May incur a performance penalty
- Little programmer control over actual messages being generated
- Programmers need to understand consistency models, to write correct programs
- DSM implementations use asynchronous message-passing, and hence cannot be more efficient than message-passing implementations

Parallel Processing Software

It is a programming challenge to develop parallel programs to solve problems efficiently.
Parallel programs are difficult to understand, even more difficult to create, more difficult to test, and even more difficult to debug. These are the principal reasons that they have not become widespread. Writing programs in which several simultaneous tasks have to synchronize and exchange data with each other is intellectually daunting.

Example
Suppose that you have ten thousand ballots, each with a ballot number, on separate pieces of paper. They are completely out of order and lying in a big box. You have one hundred volunteers who are going to place these ballots in sorted order, one behind the other, in a long, narrow container large enough to hold them all. The container does not have slots -- the ballots are placed one behind another in order, so they cannot just be put into specific positions. In other words, one cannot simply put ballot #32 into an entry labeled 32. It must be placed after ballot #31 and before ballot #33. Only one person at a time can put a ballot into the box, although the people can sort their piles outside of the box simultaneously.
Try to write up instructions for each volunteer to follow so that all volunteers can sort these ballots into the container simultaneously. How will each person know where to put the ballots? Should one person be in charge of putting ballots into the box, or can this effort be divided amongst the team? How will you prove that your algorithm is correct? Is it the best possible in terms of time? How can you prove it if it is?

This exercise, which in itself is challenging, is just the tip of a large iceberg of challenges. Even when parallel programs are written, they do not necessarily take full advantage of the parallelism inherent in the hardware. And if

Partha Sarathi

all of this isn't discouraging enough, as the number of processors increases, there is no guarantee that the performance gain will match the increase in the number of processors. This limitation is a consequence of Amdahl's Law.

Amdahl's Law

Definition. The speedup of a parallel algorithm running on p processors is the running time of the fastest known serial algorithm running on a single processor of a p-processor computer divided by the running time of the parallel algorithm running on the same p-processor computer using all p processors.

For example, if a particular parallel algorithm takes 10 seconds running on a computer using all of its 20 processors, and the fastest known serial algorithm that solves this same problem takes 150 seconds when run on one of these processors, the speedup of the algorithm is 150/10 = 15.

Definition. The efficiency of a parallel algorithm running on p processors is the speed-up of the algorithm divided by the number of processors, p. In the preceding example, the efficiency of the algorithm is 15/20 = 0.75, because it had a speedup of 15 and it used 20 processors to achieve it. Efficiency is always a number between 0 and 1.0.

Efficiency is a measure of how much a parallel algorithm takes advantage of the parallelism of the problem, as well as how well it utilizes the processors to decrease running time. For example, if a particular parallel algorithm has a speedup of 0.5p with p processors, then its efficiency is 0.5.
Gene Amdahl is a computer scientist who, among his many other accomplishments, formulated a law that puts limits on just how much speedup is possible for any given problem. It is called

Amdahl's Law:

Let **f** be the fraction of operations in a computation that must be performed sequentially, where **$0 \le f \le 1$**. The maximum speedup **Smax(p)** achievable by a parallel computer with p processors for this computation is **Smax(p) = 1/(f + (1-f)/p)** Notice that, as p approaches infinity, the maximum speedup approaches 1/f. Most problems have input and output, for example. These are usually sequential operations, as the media they access are linear in nature.

Example 1

Twenty percent of the instructions in a particular parallel algorithm are inherently sequential. An implementation of it is run on a parallel computer with 20 processors. What is the maximum possible speed-up? What is the maximum possible efficiency? Do the same with p = 100.

Solution.

In this case, f = 0.2 and p = 20, so the maximum speed-up is 1/(0.2 + 0.8/20) = 1/0.24 = 4.17. The maximum efficiency would be 4.17/20 = 0.2085. When p = 100, the maximum speedup is 1/(0.2 + 0.8/100) = 1/0.208 = 4.808. The maximum efficiency would be 4.808/100 = 0.048.

Array processor

Array processor A computer/processor that has an architecture especially designed for processing arrays (e.g. matrices) of numbers. The architecture includes a number of processors (say 64 by 64) working simultaneously, each handling one element of the array, so that a single operation can apply to all elements of the array in parallel. To obtain the same effect in a conventional processor, the operation must be applied to each element of the array sequentially, and so consequently much more slowly.

An array processor may be built as a self-contained unit attached to a main computer via an I/O port or internal bus; alternatively, it may be a *distributed array processor* where the processing elements are distributed throughout, and closely linked to, a section of the computer's memory.

Partha Sarathi

Array processors are very powerful tools for handling problems with a high degree of parallelism. They do however demand a modified approach to programming. The conversion of conventional (sequential) programs to serve array processors is not a trivial task, and it is sometimes necessary to select different (parallel) algorithms to suit the parallel approach.

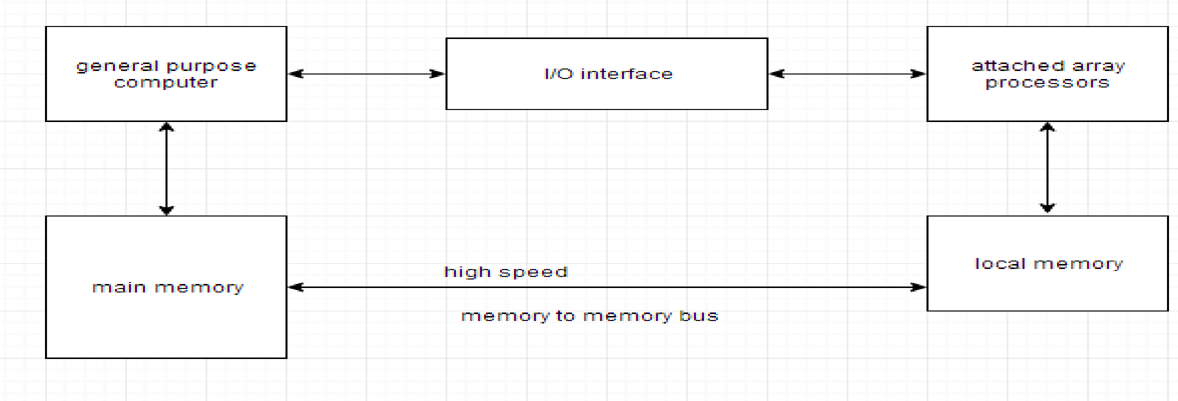
Types of Array Processors

There are basically two types of array processors:

1. Attached Array Processors
2. SIMD Array Processors

Attached Array Processors

An attached array processor is a processor which is attached to a general purpose computer and its purpose is to enhance and improve the performance of that computer in numerical computational tasks. It achieves high performance by means of parallel processing with multiple functional units.



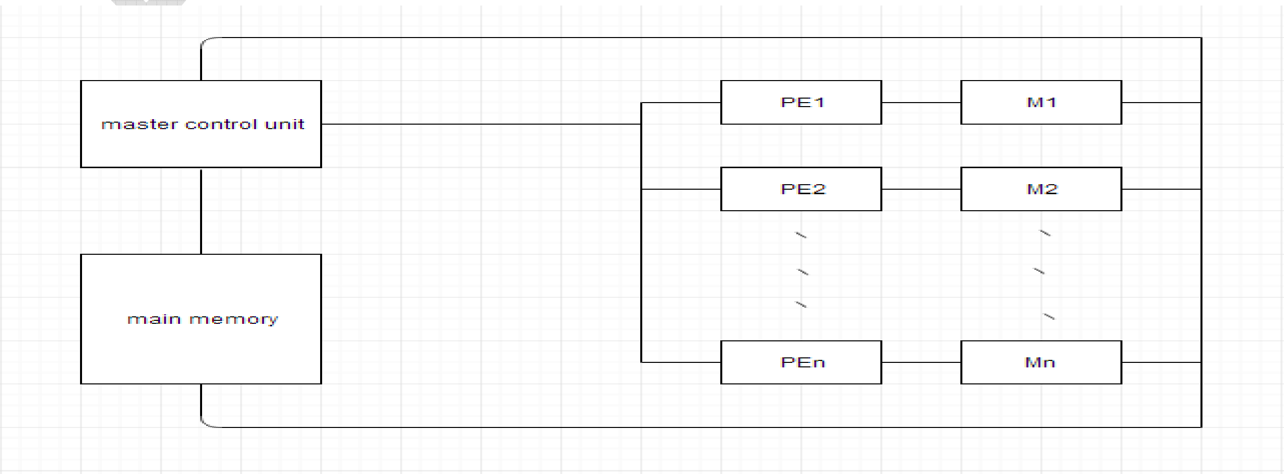
SIMD Array Processors

SIMD is the organization of a single computer containing multiple processors operating in parallel. The processing units are made to operate under the control of a common control unit, thus providing a single instruction stream and multiple data streams.

A general block diagram of an array processor is shown below. It contains a set of identical processing elements (PE's), each of which is having a local memory M. Each processor element includes an **ALU** and **registers**. The master control unit controls all the operations of the processor elements. It also decodes the instructions and determines how the instruction is to be executed.

The main memory is used for storing the program. The control unit is responsible for fetching the instructions. Vector instructions are send to all PE's simultaneously and results are returned to the memory.

The best known SIMD array processor is the **ILLIAC IV** computer developed by the **Burroughs corps**. SIMD processors are highly specialized computers. They are only suitable for numerical problems that can be expressed in vector or matrix form and they are not suitable for other types of computations.



To compute

$$Y = \sum_{i=1}^{N} A(i) * B(i)$$

Assuming:

- A dedicated memory organization.
- Elements of **A** and **B** are properly and perfectly distributed among processors (the compiler can help here).

We have:

- The product terms are generated in parallel.
- Additions can be performed in $\log_2 N$ iterations.
- Speed up factor (S) is:

$$S = \frac{2N-1}{1+\log_2 N}$$

Why use the Array Processor

- Array processors increases the overall instruction processing speed.
- As most of the Array processors operates asynchronously from the host CPU, hence it improves the overall capacity of the system.
- Array Processors has its own local memory, hence providing extra memory for systems with low memory.

When to use and not to use the array processor?

- The AP (array processor) is most efficient in doing repetitiveoperations such as doing FFT's and multiplying large vectors. Itsefficiency degrades for non repetitive operations, or operationsrequiring a great number of decisions based on the results ofcomputations.
- Since the AP's have their own program and data memory, the APinstruction and data must be transferred to , and the results transferredfrom the AP. These I/O operations may cost more CPU time than theamount saved by using the array processor.
- As a general rule , use of AP is most efficient than the CPU whenmultiple or complex (such as FFT) operations, which are highlyrepetitious, are going to be done on relatively large amount of data (thousands of words or more.). In other cases use of AP will not helpmuch and will keep other processes from using valuable resource.

Conclusion

- Though array processor can improve the performance but all problems can notbe attacked with this sort of solution. Instructions of array processor to processan array of data at a time necessarily adds complexity to the core CPU. Thatcomplexity typically makes other instructions run .The more complexinstructions also add to the complexity of the decoders, which might slowdown the decoding of the more common instructions such as normal adding.
- So the array processors work best only when there are large amounts of data tobe worked on. For this reason, these sorts of CPUs were found primarilyin supercomputers, as the supercomputers themselves were, in general, foundin places such as weather prediction centres and physics labs, where hugeamounts of data are "crunched".
- This architecture relies on the fact that the data sets are all acting on a singleinstruction. However if these data sets somewhat rely on each other then youcannot apply parallel processing. For example if data A has to be processedbefore data B then you cannot do both A and B simultaneously. Thisdependency is what makes parallel processing difficult to implement and it iswhy sequential machines are extremely common.

Partha Sarathi

How Array Processor can help?

- Consider the simple task of adding two groups of 10 numbers together. In a normal programming language you might have done something as
- **execute this loop 10 times**
    - **read the next instruction and decode it**
    - **fetch this number fetch that number**
    - **add them**
    - **put the result here**
- **End loop**
- But to an array processor this tasks looks as
- **read instruction and decode it**
- **fetch these 10 numbers**
- **fetch those 10 numbers**
- **add them**
- **put the results here**

Vector Processors

There are two essentially different models of parallel computers: vector processors and multiprocessors. A vector processor, is simply a machine that has an instruction that can operate on a vector. A pipelined vector processor is a vector processor that can issue a vector instruction that operates on all of the elements of the vector in parallel by sending those elements through a highly pipelined functional unit with a fast clock. A processor array is a vector processor that achieves the parallelism by having a collection of identical, synchronized processing elements (PE), each of which executes the same instruction on different data, which are controlled by a single control unit. Every PE has a unique identifier, its processor id, which can be used during the computation. The control unit, which might be a full-fledged CPU, broadcasts the instruction to be executed to the processing elements, which execute it on data from a memory that is usually local to each, and can store the result in their local memories, or can return global results back to the CPU. A global result line is usually a separate, parallel bus that allows each PE to transmit values back to the CPU to be combined by a parallel, global operation, such as a logical-and or a logical-or, depending upon the hardware support in the CPU.

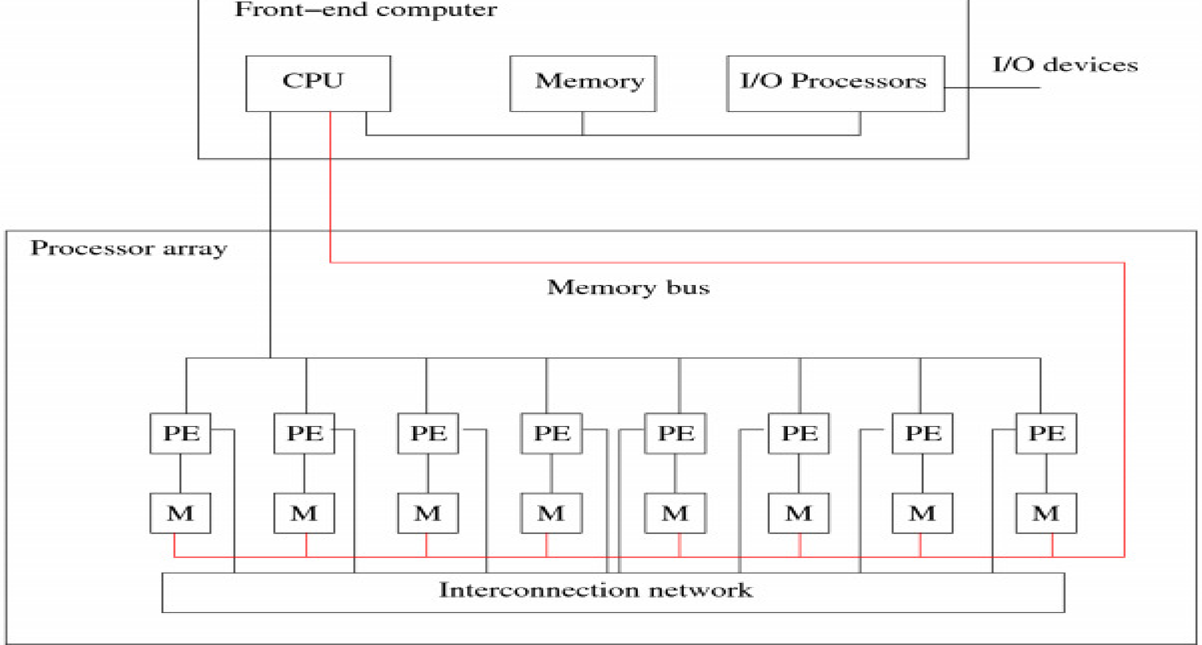


Figure 1: Processor array architecture.

Because all PEs execute the same instruction at the same time, this type of architecture is suited to problems with data parallelism. Data parallelism is a type of parallelism that is characterized by the ability to perform the same operation on different data simultaneously. For example, a loop of the form

```
for i = 0 to N-1
        do a[i] = a[i] + 1;
```

has data parallelism because the updates to the distinct array elements a[i] are independent of each other and may be performed in parallel, whereas the loop

```
for i = 1 to N-1
        do a[i] = a[i-1] + 1;
```

has no data parallelism because the update to a[i] cannot be performed until the update to a[i-1]. If the value of N is smaller than the number of processing elements, the entire loop takes the same amount of time as a single processor takes to perform the increment on a scalar variable. If the value of N is larger, then the work has to be distributed to the PEs so that they each update the values of several array elements. This may be handled by the hardware, by a runtime library, or by the programmer, depending on the particular architecture and software.

Data parallelism exists in many application areas, including all of the grand challenge problems mentioned earlier and most scientific problems in general. Commercial applications with a large degree of data parallelism also include image processing and high-end graphics.

Array processor hardware also has to handle what PEs do when they do not need to participate in a computation. For example, when N is smaller than the number of PEs in the above loop, then some PEs will have to be de-activated, or masked, during the computation. The PEs usually have the capability to mask themselves, which they would do conditionally depending on the value of their PE identifier.

The PEs are usually connected to each other through an interconnection network, which can take on many forms. Interconnection networks are covered later in this chapter.

Vector processor classification

According to from where the operands are retrieved in a vector processor, pipe lined vector computers are classified into two architectural configurations:

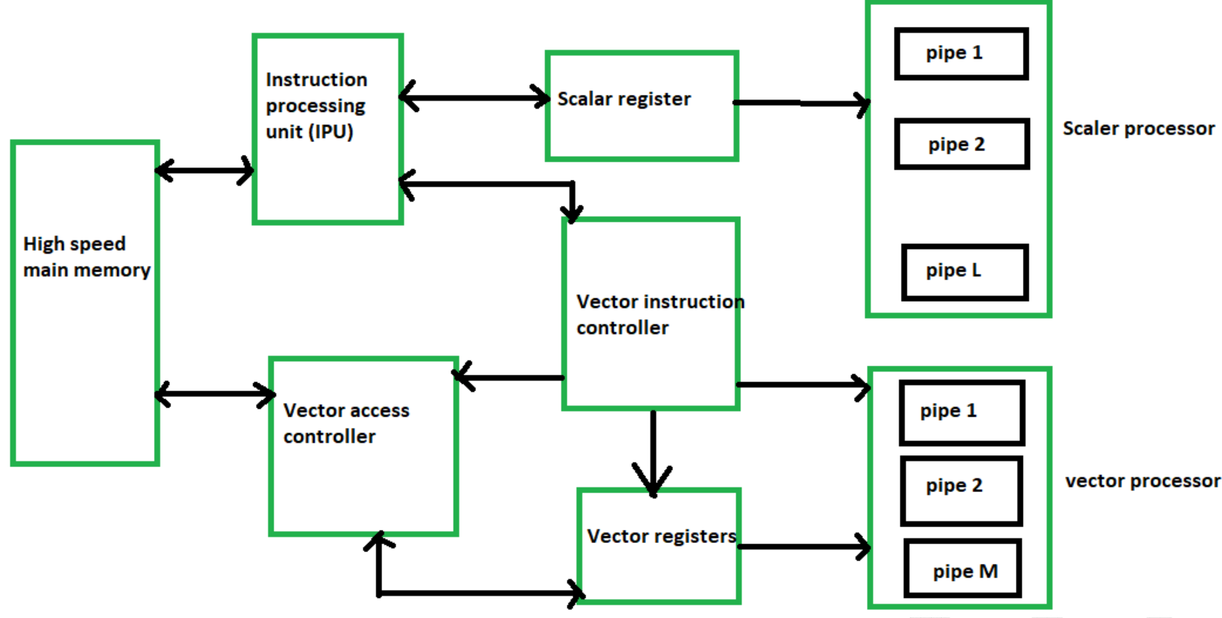
1. **Memory to memory architecture –**

   In memory to memory architecture, source operands, intermediate and final results are retrieved (read) directly from the main memory. For memory to memory vector instructions, the information of the base address, the offset, the increment, and the the vector length must be specified in order to enable streams of data transfers between the main memory and pipelines. The processors like *TI-ASC, CDC STAR-100, and Cyber-205* have vector instructions in memory to memory formats. The main points about memory to memory architecture are:
   - There is no limitation of size
   - Speed is comparatively slow in this architecture
2. **Register to register architecture –**

   In register to register architecture, operands and results are retrieved indirectly from the main memory through the use of large number of vector registers or scalar registers. The processors like *Cray-1 and the Fujitsu VP-200* use vector instructions in register to register formats. The main points about register to register architecture are:
   - Register to register architecture has limited size.
   - Speed is very high as compared to the memory to memory architecture.
   - The hardware cost is high in this architecture.

A block diagram of a modern multiple pipeline vector computer is shown below:

Partha Sarathi



Vector instruction types

A **Vector operand** contains an ordered set of n elements, where n is called the **length of the vector**. All elements in a vector are same type scalar quantities, which may be a floating point number, an integer, a logical value, or a character.

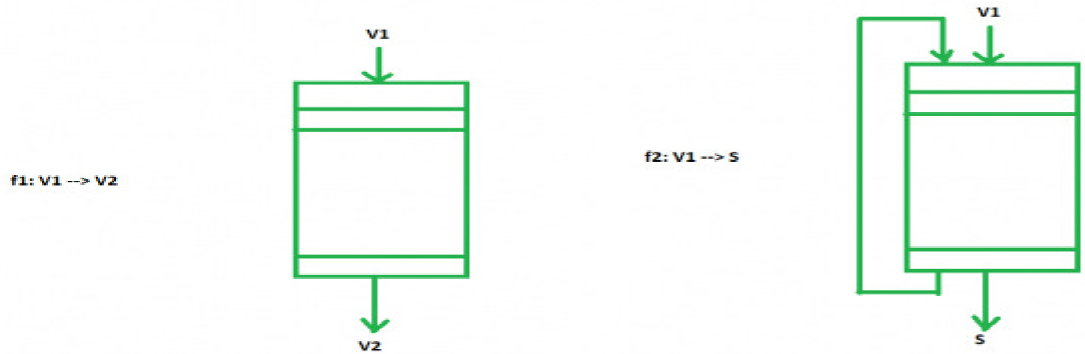
Four primitive types of vector instructions are:

```
f1 : V --> V
f2 : V --> S
f3 : V x V --> V
f4 : V x S --> V
```

Where V and S denotes a vector operand and a scalar operand, respectively.

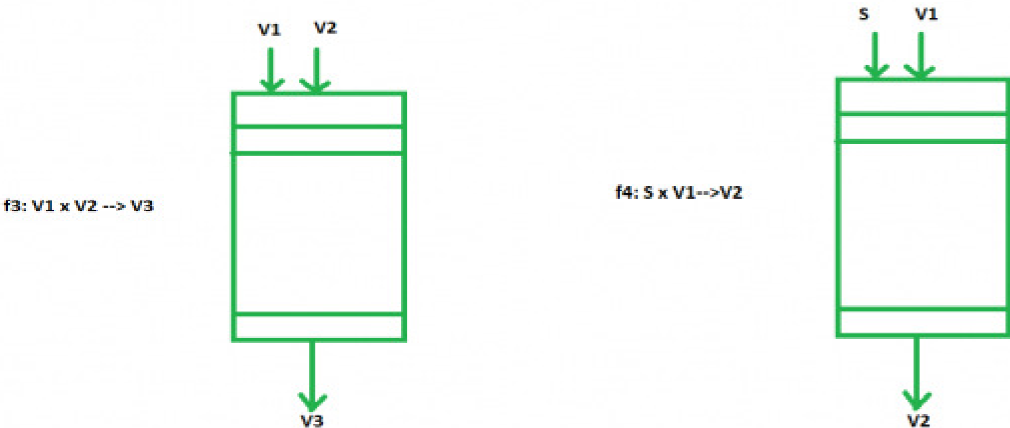
The instructions, f1 and f2 are unary operations and f3 and f4 are binary operations.

The **VCOM (vector complement)**, which complements each complement of the vector, is an f1 operation. The pipe lined implementation of f1 operation is shown in the figure:



The **VMAX (vector maximum)**, which finds the maximum scalar quantity from all the complements in the vector, is an f2 operation. The pipe lined implementation of f2 operation is shown in the above figure:

The **VMPL (vector multiply)**, which multiply the respective scalar components of two vector operands and produces another product vector, is an f3 operation. The pipe lined implementation of f3 operation is shown in the figure:



The **SVP (scalar vector product)**, which multiply one constant value to each component of the vector, is f4 operation. The pipe lined implementation of f4 operation is shown in the above figure:

Vector Instruction Format in Vector Processors

Different **Instruction formats** are used by different vector processors. Vector instructions are generally specified by some fields. The main fields that are used in **vector instruction set** are given below:

1. **Operations Code (Opcode) –**

   The operation code must be specified to select the functional unit or to reconfigure a multi-functional unit to perform the specified operation dictated by this field. Usually, microcode control is used to set up the required resources.

For example:

Opcode – 0001 mnemonic – ADD operation – add the content of memory to the content of accumulator

Opcode – 0010 mnemonic – SUB operation – subtract the content of memory to the content of accumulator

Opcode – 1111 mnemonic – HLT operation – stop processing

2. **Base addresses –**

   For a memory reference instruction, the base addresses are needed for both source operands and result vectors. The designated vector registers must be specified in the instruction, if the operands and results are located in the vector register file, i.e., collection of registers.

1. **For example**:

   ```
   ADD R1, R2
   ```

   Here, R1 and R2 are the addresses of the register.

2. **Offset (or Displacement)** –
   This field is required to get the effective memory address of operand vector. The address offset relative to the base address should be specified. Using the base address and the offset (positive or negative), the effective address is calculated.
3. **Address Increment** –
   The address increment between the scalar elements of vector operand must be specified. Some computers, i.e., the increment is always 1. Some other computers, like TI-ASC, can have a variable increment, which offers higher flexibility in application.
   For example:

   ```
   R1 <- 400
   ```

   Auto incr-R1 is incremented the value of R1 by 1.

   ```
   R1 = 399
   ```

4. **Vector length** – The vector length (positive integer) is needed to determine the termination of a vector instruction.

ADVANTAGE

- Each result is independent of previous results - allowing high clock rates.
- A single vector instruction performs a great deal of work - meaning less fetches and fewer branches (and in turn fewer mispredictions).
- Vector instructions access memory a block at a time which′ results in very low memory latency.
- Less memory access = faster processing time.
- Lower cost due to low number of operations compared to′ scalar counterparts.

DISADVANTAGES

- Works well only with data that can be executed in highly or completely parallel manner.
- Needs large blocks of data to operate on to be efficient′ because of the recent advances increasing speed of accessing memory.
- Severely lacking in performance compared to normal′ processors on scalar data.
- High price of individual chips due to limitations of on-chip′ memory.
- Increased code complexity needed to vectorize the data.
- High cost in design and low returns compared to′ superscalar microprocessors.

CONCLUSION

- The Vector machine is faster at performing mathematical′ operations on larger vectors.
- The Vector processing computer's vector register′ architecture makes it better able to compute vast amounts of data quickly.
- While Vector Processing is not widely popular today, it still′ represents a milestone in supercomputing achievement.
- It is still in use today in home PC's as SIMD units which′ augment the scalar CPU when necessary (usually GPUs).
- Since scalar processors designed can also be used for′ general applications their cost per unit is reduced drastically. Such is not the case for vector processors/supercomputers.
- Vector processors will continue to have a future in Large′ Scale computing and certain applications but can never reach the popularity of Scalar microprocessors

Difference between array processor and vector processor

In computing, a vector processor or array processor is a central processing unit (CPU) that implements an instruction set containing instructions that operate on one-dimensional arrays of data called vectors.

Vector and array processing are essentially the same with small differences.
An array is made up of indexed collections of information called indices. Though an array can, in rare cases, have only one index collection, a vector is technically indicative of an array with at least two indices. Vectors are sometimes referred to as "blocks" of computer data.
Vector and array processing technology are most often seen in high-traffic servers.

Associative Processor

Consider that a table or a list of record is stored in the memory and you want to find some information in that list. For example, the list consists of three fields as shown below:

Name ID	Number	Age
Sumit	234	23
Ramesh	136	26
Ravi	97	35

Partha Sarathi

Suppose now that we want to find the ID number and age of Ravi. If we use conventional RAM then it is necessary to give the exact physical address of entry related to Ravi in the instruction access the entry such as: **READ ROW 3**
Another alternative idea is that we search the whole list using the Name field as an address in the instruction such as**: READ NAME = RAVI**
Again with serial access memory this option can be implemented easily but it is a very slow process. An **associative memory** helps at this point and simultaneously examines all the entries in the list and returns the desired list very quickly. SIMD array computers have been developed with associative memory. An associative memory is content addressable memory, by which it is meant that multiple memory words are accessible in parallel. The parallel accessing feature also support parallel search and parallel compare. This capability can be used in many applications such as:

- Storage and retrieval of databases which are changing rapidly
- Radar signal tracking
- Image processing
- Artificial Intelligence

The inherent parallelism feature of this memory has great advantages and impact in parallel computer architecture. The associative memory is costly compared to RAM. The array processor built with associative memory is called Associative array processor. In this section, we describe some categories of associative array processor. Types of associative processors are based on the organisation of associative memory. Therefore, first we discuss about the associative memory organisation.

Associative Memory Organisations

The associative memory is organised in w words with b bits per word. In w x b array, each bit is called a cell. Each cell is made up of a flip-flop that contains some comparison logic gates for pattern match and read-write operations. Therefore, it is possible to read or write in parallel due to this logic structure. A group of bit cells of all the words at the same position in a vertical column is called bit slice as shown in Figure

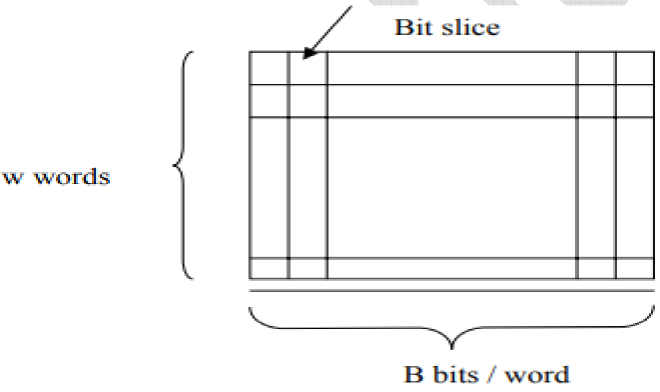


Figure 8: Associative memory

In the organisation of an associative memory, following registers are used:

- Comparand Register (C): This register is used to hold the operands, which are being searched for, or being compared with.
- Masking Register (M): It may be possible that all bit slices are not involved in parallel operations. Masking register is used to enable or disable the bit slices.
- Indicator (I) and Temporary (T) Registers: Indicator register is used to hold the current match patterns and temporary registers are used to hold the previous match patterns.

There are following two methods for organising the associative memory based on bit Architecture slices:

- Bit parallel organisation: In this organisation all bit slices which are not masked off, participate in the comparison process, i.e., all words are used in parallel.
- Bit Serial Organisation: In this organisation, only one bit slice participate in the operation across all the words. The bit slice is selected through an extra logic and control unit. This organisation is slower in speed but requires lesser hardware as compared to bit parallel which is faster.

Types of Associative Processor

Based on the associative memory organisations, we can classify the associative processors into the following categories:

1) **Fully Parallel Associative Processor**: This processor adopts the bit parallel memory organisation. There are two type of this associative processor:

- Word Organized associative processor: In this processor one comparison logic is used with each bit cell of every word and the logical decision is achieved at the output of every word.
- Distributed associative processor: In this processor comparison logic is provided with each character cell of a fixed number of bits or with a group of character cells. This is less complex and therefore less expensive compared to word organized associative processor.

2) Bit Serial Associative Processor: When the associative processor adopts bit serial memory organization then it is called bit serial associative processor. Since only one bit slice is involved in the parallel operations, logic is very much reduced and therefore this processor is much less expensive than the fully parallel associative processor.

PEPE is an example of distributed associative processor which was designed as a special purpose computer for performing real time radar tracking in a missile environment. STARAN is an example of a bit serial associative processor which was designed for digital image processing. There is a high cost performance ratio of associative processors. Due to this reason these have not been commercialised and are limited to military applications.

SYSTOLIC ARCHITECTURE

Parallel processing approach diverges from traditional Von Neumann architecture. One such approach is the concept of Systolic processing using systolic arrays.

A systolic array is a network of processors that rhythmically compute and pass data through the system. They derived their name from drawing an analogy to how blood rhythmically flows through a biological heart as the data flows from memory in a rhythmic fashion passing through many elements before it returns to memory.It is also an example of pipelining along with parallel computing.It was introduced in 1970s and was used by Intel to make CMU's iWarp processor in 1990.

In a systolic array there are a large number of identical simple processors or processing elements(PEs) that are arranged in a well organised structure such as linear or two dimensional array. Each processing element is connected with the other PEs and has a limited private storage.

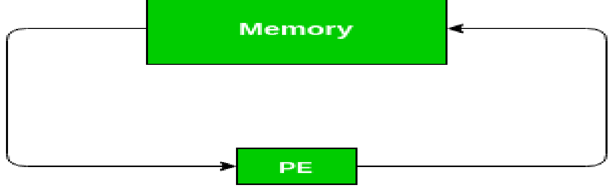


Figure - A Processing Element (PE)

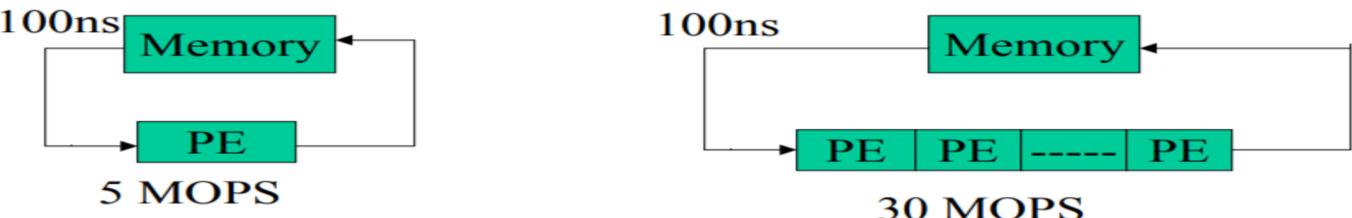
Partha Sarathi

A Host station is often used for communication with the outside world in the network.

Characteristics:

1. **Parallel Computing –**
   Many processes are carried out simultaneously. As the arrays have a non-centralized structure, parallel computing is implemented.
2. **Pipelinability –**
   It means that the array can achieve high speed. It shows a linear rate pipelinability.
3. **Synchronous evaluation –**
   Computation of data is timed by a global clock and then the data is passed through the network.The global clock synchronizes the array and has fixed length clock cycles.
4. **Repetability –**
   Most of the arrays have the repetition and interconnection of a single type of PE in the entire network.
5. **Spatial Locality –**
   The cells have a local communication interconnection.
6. **Temporal Locality –**
   One unit time delay is at least required for the transmission of signals from one cell to another.
7. **Modularity and regularity –**
   A systolic array consists of processing units that are modular and have homogeneous interconnection and the computer network can be extended indefinitely.

- Systolic computers are a new class of pipelined array architecture.
- The Basic Principle of a systolic system.

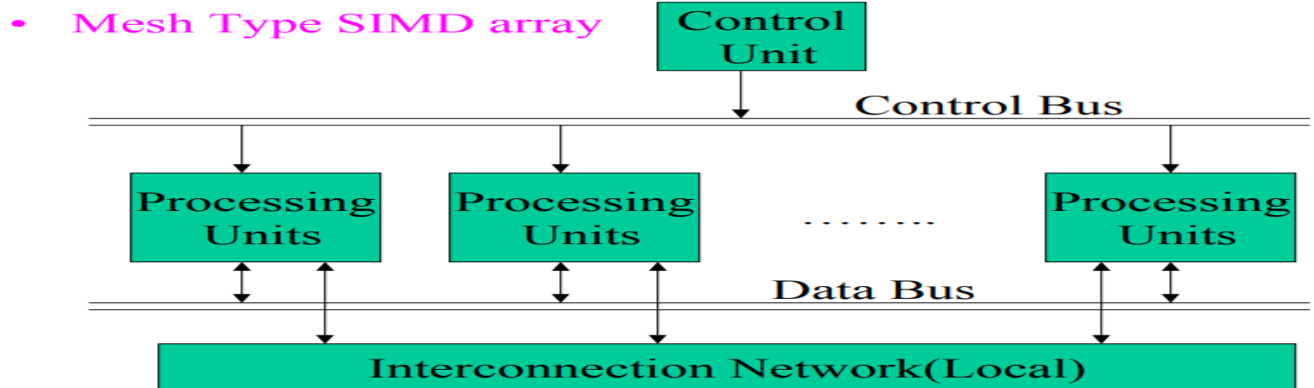


NOTE: MOPS⇒Millions of Operations Per Second

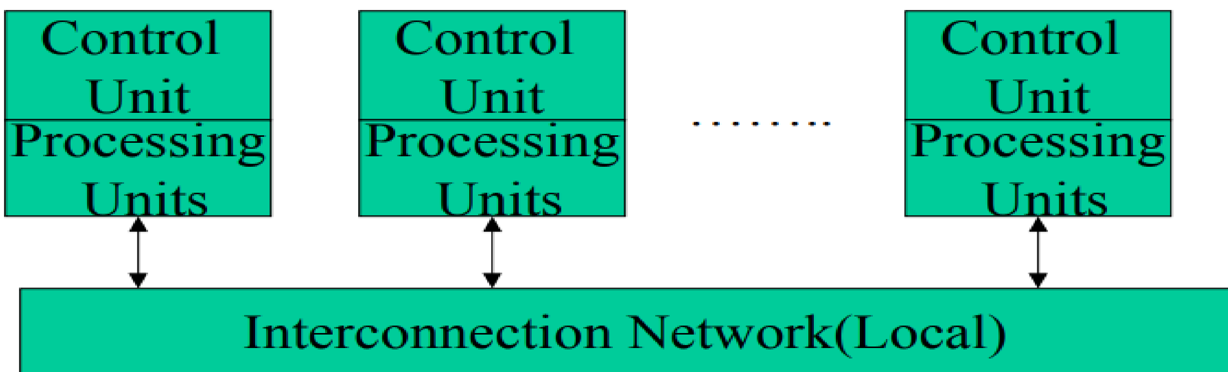
Systolic Systems

- Systolic systems consists of an array of PE(Processing Elements)
- Processors are called cells, each cell is connected to a small number of nearest neighbours in a mesh like topology. Each cell performs a sequence of operations on data that flows between them.
- Generally the operations will be the same in each cell, each cell performs an operation or small number of operations on a data item and then passes it to its neighbor.
- Systolic arrays compute in “lock-step” with each cell (processor) undertaking alternate compute/communicate phases.

Difference between SIMD array and Systolic array



- Systolic Array.



An SIMD array is a synchronous array of PEs under the supervision of one control unit and all PEs receive the same instruction broadcast from the control unit but operate on different data sets from distinct data streams. SIMD array usually loads data into its local memories before starting the computation. Systolic arrays usually pipe data from an outside host and also pipe the results back to the host.

Why Systolic Architecture?

It can be used for special purpose processing architecture because of

- Simple and Regular Design.
- Concurrency and communication.
- Balancing communication with I/O

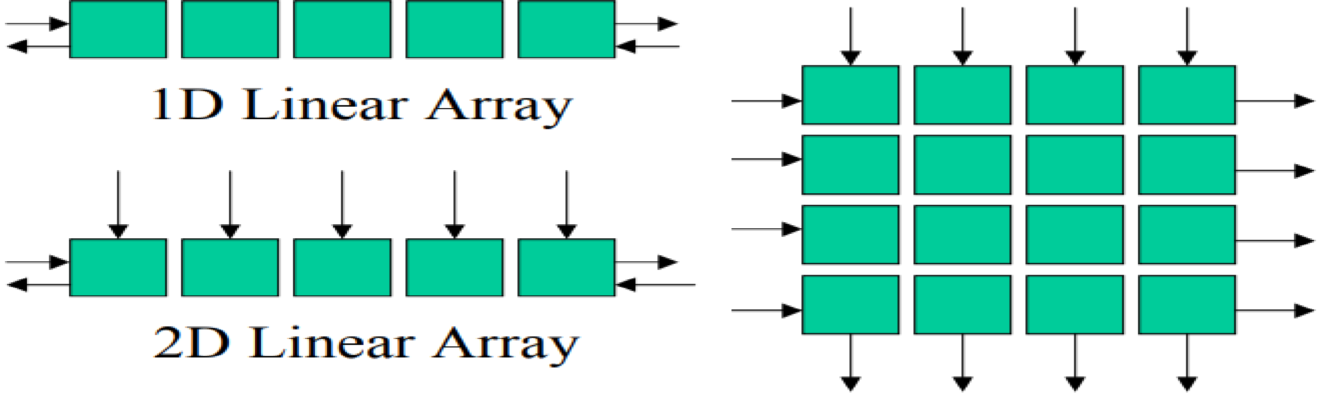
The systolic arrays has a regular and simple design (i.e) They are:

- cost effective,
- array is modular (i.e) adjustable to various performance goals ,
- large number of processors work together,
- local communication in systolic array is advantageous for communication to be faster.

A systolic array is used as attached array processor, it receives data and o/p the results through an attached host computer, therefore the performance goal of array processor system is a computation rate that balances I/o bandwidth with host. With relatively low bandwidth of current I/O devices, to achieve a faster computation rate it is necessary to perform multiple computations per I/O access. Systolic arrays does this efficiently.

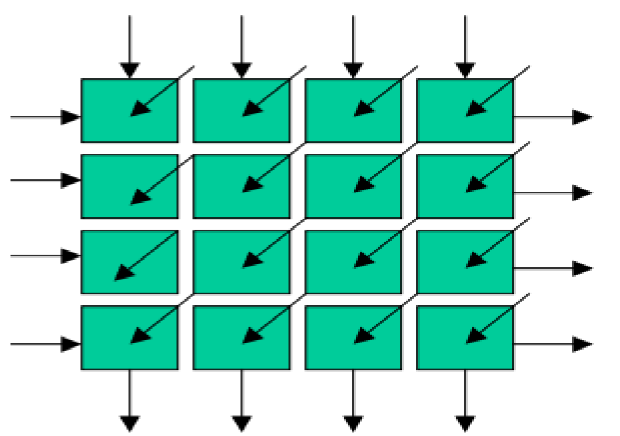
Types of systolic arrays

Early systolic arrays **are linear arrays and one dimensional(1D) or two dimensional I/O(2D)**. Most recently, systolic arrays are implemented as planar array with perimeter I/O to feed data through the boundary.



Partha Sarathi

Linear array with **1D I/O**. This configuration is suitable for single I/O. **Linear array with 2D I/O**. It allows more control over linear array**. Planar array with perimeter I/O**. This configuration allows I/O only through its boundary cells. **Focal Plane array** with 3D I/O. This configuration allows I/O to each systolic cell.



Systolic arrays can be built with variations in:

– 1. Connection Topology

- 2D Meshes
- hypercubes

– 2. Processor capability: ranging through:

- trivial- just an ALU
- ALU with several registers
- Simple CPU- registers, run own program
- Powerful CPU- local memory also

– 3 Re-configurable Field programmable Gate Arrays (FPGAs) offer the possibility that re-programmable, reconfigurable arrays can be constructed to efficiently compute certain problems.

In general, FPGA technology is excellent for building small systolic array-style processors. • Special purpose ALUs can be constructed and linked in a topology to, which the target application maps well

Example 1

Polynomial Evaluation is done by using a Linear array with 2D.

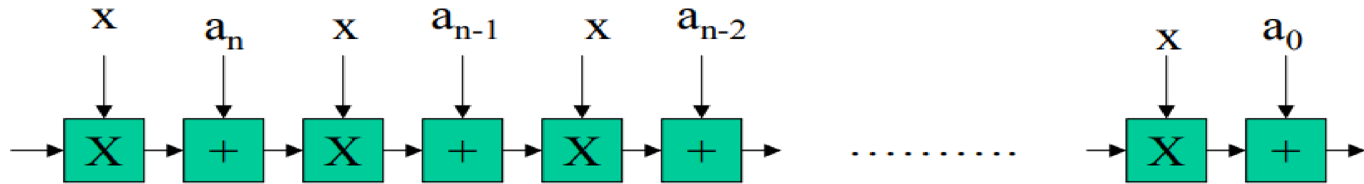
Expression:

$$Y = ((((a_n x + a_{n-1}) * x + a_{n-2}) * x + a_{n-3}) * x \ldots\ldots a_1) * x + a_0$$

Function of PEs in pairs

- Multiply input by x
- Pass result to right.
- Add aj to result from left.
- Pass result to right

Using systolic array for polynomial evaluation.



This array can produce a polynomial on every cycle - after 2n stages.

This is an example of a deeply pipelined computation- – The pipeline has 2n stages,

Advantages of Systolic arrays : Advantages of systolic arrays are: –

- Regularity and modular design(Perfect for VLSI implementation). –
- Local interconnections(Implements algorithms locality).
- High degree of pipelining.
- Highly synchronized multiprocessing.
- Simple I/O subsystem.
- Very efficient implementation for great variety of algorithms.
- High speed and Low cost.
- Elimination of global broadcasting and modular expansibility.

Disadvantages of systolic arrays

The main disadvantages of systolic arrays are:

- Global synchronization limits due to signal delays.
- High bandwidth requirements both for periphery(RAM) and between PEs.
- Poor run-time fault tolerance due to lack of interconnection protocol.

Applications Of Systolic Arrays

The various applications of systolic arrays are:

- Matrix Inversion and Decomposition.
- Polynomial Evaluation.
- Convolution.
- Systolic arrays for matrix multiplication.
- Image Processing.
- Systolic lattice filters used for speech and seismic signal processing.
- Artificial neural network.

Interconnection network

An **interconnection network** in a parallel machine transfers information from any source node to any desired destination node. This task should be completed with as small latency as possible. It should allow a large number of such transfers to take place concurrently. Moreover, it should be inexpensive as compared to the cost of the rest of the machine.

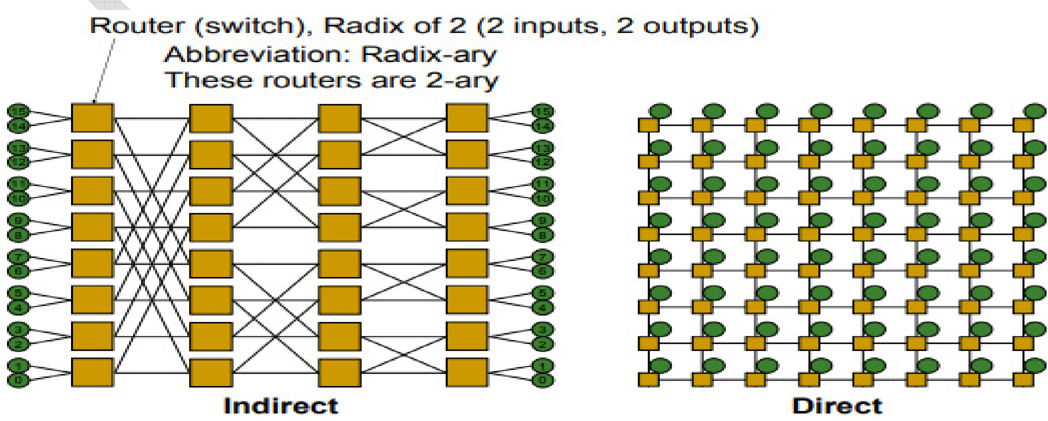
The network is composed of links and switches, which helps to send the information from the source node to the destination node. A network is specified by its topology, routing algorithm, switching strategy, and flow control mechanism.

Organizational Structure

Interconnection networks are composed of following three basic components −

- **Links** − A link is a cable of one or more optical fibers or electrical wires with a connector at each end attached to a switch or network interface port. Through this, an analog signal is transmitted from one end, received at the other to obtain the original digital information stream.
- **Switches** − A switch is composed of a set of input and output ports, an internal "cross-bar" connecting all input to all output, internal buffering, and control logic to effect the input-output connection at each point in time. Generally, the number of input ports is equal to the number of output ports.
- **Network Interfaces** − The network interface behaves quite differently than switch nodes and may be connected via special links. The network interface formats the packets and constructs the routing and control information. It may have input and output buffering, compared to a switch. It may perform end-to-end error checking and flow control. Hence, its cost is influenced by its processing complexity, storage capacity, and number of ports.

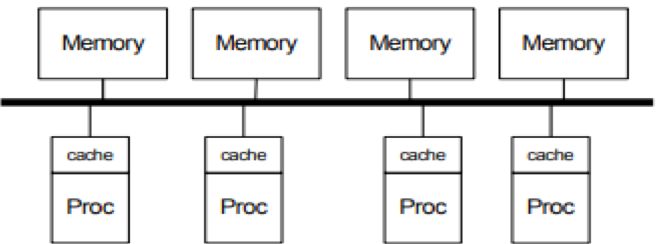
Interconnection Network



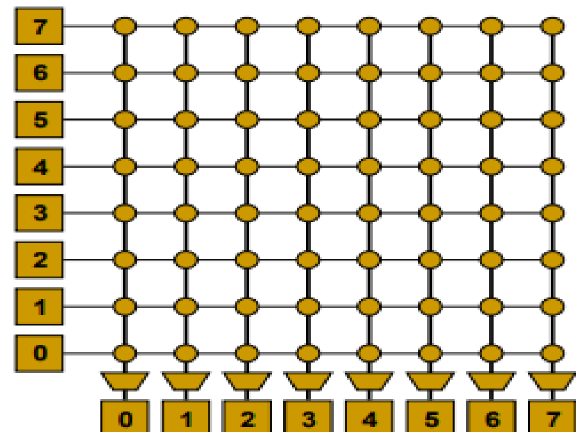
Partha Sarathi

Interconnection networks are composed of switching elements. Topology is the pattern to connect the individual switches to other elements, like processors, memories and other switches. A network allows exchange of data between processors in the parallel system.

- **Direct connection networks** – Direct networks have point-to-point connections between neighboring nodes. These networks are static, which means that the point-to-point connections are fixed. Some examples of direct networks are rings, meshes and cubes.
- **Indirect connection networks** – Indirect networks have no fixed neighbors. The communication topology can be changed dynamically based on the application demands. Indirect networks can be subdivided into three parts: bus networks, multistage networks and crossbar switches.
  - **Bus networks** – A bus network is composed of a number of bit lines onto which a number of resources are attached. When busses use the same physical lines for data and addresses, the data and the address lines are time multiplexed. When there are multiple bus-masters attached to the bus, an arbiter is required.



  - **Multistage networks** – A multistage network consists of multiple stages of switches. It is composed of 'axb' switches which are connected using a particular interstage connection pattern (ISC). Small 2x2 switch elements are a common choice for many multistage networks. The number of stages determine the delay of the network. By choosing different interstage connection patterns, various types of multistage network can be created.
  - **Crossbar switches** – A crossbar switch contains a matrix of simple switch elements that can switch on and off to create or break a connection. Turning on a switch element in the matrix, a connection between a processor and a memory can be made. Crossbar switches are non-blocking, that is all communication permutations can be performed without blocking.



Evaluating Design Trade-offs in Network Topology

If the main concern is the routing distance, then the dimension has to be maximized and a hypercube made. In store-and-forward routing, assuming that the degree of the switch and the number of links were not a significant cost factor, and the numbers of links or the switch degree are the main costs, the dimension has to be minimized and a mesh built.

In worst case traffic pattern for each network, it is preferred to have high dimensional networks where all the paths are short. In patterns where each node is communicating with only one or two nearby neighbors, it is preferred to have low dimensional networks, since only a few of the dimensions are actually used.

Routing

The routing algorithm of a network determines which of the possible paths from source to destination is used as routes and how the route followed by each particular packet is determined. Dimension order routing limits the set of legal paths so that there is exactly one route from each source to each destination. The one obtained by first traveling the correct distance in the high-order dimension, then the next dimension and so on.

Routing Mechanisms

Arithmetic, source-based port select, and table look-up are three mechanisms that high-speed switches use to determine the output channel from information in the packet header. All of these mechanisms are simpler than the kind of general routing computations implemented in traditional LAN and WAN routers. In parallel computer networks, the switch needs to make the routing decision for all its inputs in every cycle, so the mechanism needs to be simple and fast.

Deterministic Routing

A routing algorithm is deterministic if the route taken by a message is determined exclusively by its source and destination, and not by other traffic in the network. If a routing algorithm only selects shortest paths toward the destination, it is minimal, otherwise it is non-minimal.

Deadlock Freedom

Deadlock can occur in a various situations. When two nodes attempt to send data to each other and each begins sending before either receives, a 'head-on' deadlock may occur. Another case of deadlock occurs, when there are multiple messages competing for resources within the network.

The basic technique for proving a network is deadlock free, is to clear the dependencies that can occur between channels as a result of messages moving through the networks and to show that there are no cycles in the overall channel dependency graph; hence there is no traffic patterns that can lead to a deadlock. The common way of doing this is to number the channel resources such that all routes follow a particular increasing or decreasing sequences, so that no dependency cycles arise.

Switch Design

Design of a network depends on the design of the switch and how the switches are wired together. The degree of the switch, its internal routing mechanisms, and its internal buffering decides what topologies can be supported and what routing algorithms can be implemented. Like any other hardware component of a computer system, a network switch contains data path, control, and storage.

Ports

The total number of pins is actually the total number of input and output ports times the channel width. As the perimeter of the chip grows slowly compared to the area, switches tend to be pin limited.

Internal Datapath

The datapath is the connectivity between each of the set of input ports and every output port. It is generally referred to as the internal cross-bar. A non-blocking cross-bar is one where each input port can be connected to a distinct output in any permutation simultaneously.

Channel Buffers

The organization of the buffer storage within the switch has an important impact on the switch performance. Traditional routers and switches tend to have large SRAM or DRAM buffers external to the switch fabric, while in VLSI switches the buffering is internal to the switch and comes out of the same silicon budget as the datapath and the control section. As the chip size and density increases, more buffering is available and the network designer has more options, but still the buffer real-estate comes at a prime choice and its organization is important.

Flow Control

When multiple data flows in the network attempt to use the same shared network resources at the same time, some action must be taken to control these flows. If we don't want to lose any data, some of the flows must be blocked while others proceed.

Partha Sarathi

The problem of flow control arises in all networks and at many levels. But it is qualitatively different in parallel computer networks than in local and wide area networks. In parallel computers, the network traffic needs to be delivered about as accurately as traffic across a bus and there are a very large number of parallel flows on very small-time scale.

Multiprocessor Network Topologies

Networks are used to connect processors to processors, processors to memories and processormemory nodes to each other. The way that these entities are connected to each other has a significant effect on the cost, applicability, scalability, reliability, and performance of the parallel computer. In general, the things that are connected to each other will be called nodes, whether they are processors, memories, or processor-memory elements. The set of connections between nodes is called an **interconnection network**.

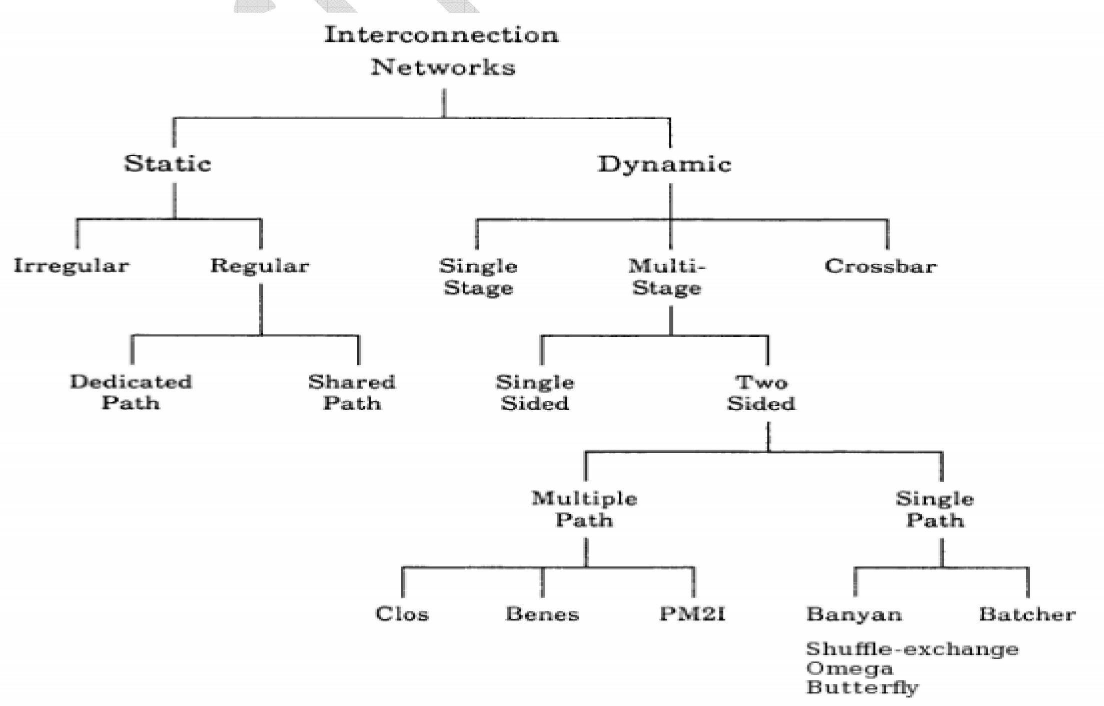
An interconnection network may be classified as shared or switched. A shared network can have at most one message on it at any time. A bus is a shared network, as is Ethernet. In contrast, a switched network allows point-to-point messages among pairs of nodes and therefore supports the transfer of multiple concurrent messages.

Interconnection networks may also be classified as **static or dynamic.** A network is static if communication links between pairs of processors are "permanent": they can only be changed by physically reconfiguring the network. Static networks are usually constructed by point-to-point connections between processors. A network is dynamic if links are established by dynamically configuring switches to establish paths between processors and other processors or memory modules. There are three types of dynamic networks:

- crossbar switching networks
- single-stage networks
- multi-stage networks.

Buses have been classified as both static and dynamic. They are static in the sense that the physical communication links are permanent and fixed, if one ignores the ability to dynamically add devices to them. They are dynamic if devices can attach and detach themselves from the bus dynamically. Some authors claim that they are dynamic because the logical links change dynamically on the bus, but this is not consistent with the definition of static given above.

Static networks are usually used in highly parallel multiprocessors to connect processors to each other, whereas dynamic networks are usually used in multiprocessors to connect processors to memory nodes or other processors. Below Figure is one way to classify the various types of interconnection networks.



In the figure, static networks are classified as either regular or irregular. This has to do with their topologies, which we will discuss below. Regular static networks can have shared or dedicated paths. Most of the static networks we examine have dedicated paths.
In mathematics, a topology is a set together with an adjacency relation on the set. Topologies do not have to be finite sets for mathematicians, but for computer scientists, the practical applications of topologies are finite sets. If you have already learned about graphs, then you know about topologies. Your mental picture of a topology should be a "tinker-toy" – a set of nodes connected by edges. In topologies, distance does not exist; when distance exists, the topology becomes a metric space. A graph is the same thing as a finite topology.

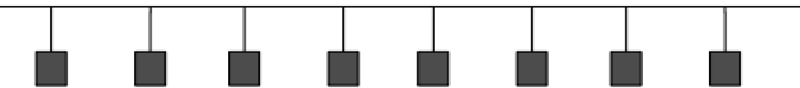


Figure 5: Bus topology

Networks can be characterized by their topologies. For example, a bus is a single line segment to which each node is attached, whereas a ring is like a bus in which the ends of the line segment are attached to each other, forming a cycle. Buses and rings have bidirectional links. Although buses and rings are similar from a topological point of view, they are very different in their performance. In a bus, there is only a single link, and only one message at a time can be in the network; it is a shared link. Each host checks to see if the message is addressed to it, and if not it ignores it. In contrast, in a ring, each host is attached to the network via a switch, and these switches act to separate the links. Thus, if there are n hosts, there are n links, and there can be n simultaneous messages on the network. Messages hop from one switch to another until they arrive at their destination address. These are just two very simple topologies. There are other, more complex topologies with interesting properties.

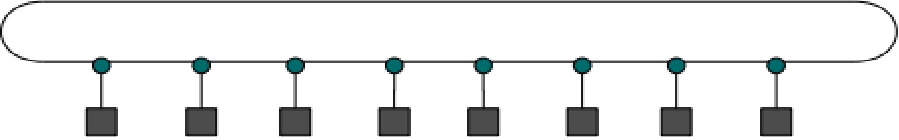


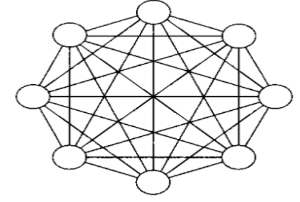
Figure 6: Ring topology

Static Topologies

There are many different network topologies. Some have been proposed but never realized. Others have found their way into commercial parallel computers. We will look at the most viable and the most interesting topologies.

Fully-Connected Network

In a fully-connected network, every node is connected to every other node, as in below Figure . If there are n nodes, there will be n(n-1)/2 links. Suppose n is even. Then there are n/2 even numbered nodes and n/2 odd numbered nodes. If we remove every edge that connect an even node to an odd node, then the even nodes will form a fully-connected network and so will the odd nodes, but the two sets will be disjoint. There are (n/2) edges from each even node to every odd node, so there are (n/2)2 edges that connect these two sets. Not removing any one of them fails to disconnect the two sets, so this is the minimum number. Therefore, the bisection width (and bandwidth) is (n/2)2 . The diameter is 1, since there is a direct link to any node from every node. The maximum edges per node is proportional to n, so this network does not scale well. Lastly, the maximum edge length will increase as the network grows, under the assumption that nodes take fixed amount of space. (Think of the nodes as lying on the surface of a sphere, and the edges as chords connecting them.)



Mesh networks

In a mesh network, nodes are arranged in a q-dimensional lattice. A 2-dimensional lattice with 36 nodes is illustrated in below Figure. In general, there are k2 nodes in a 2-dimensional mesh. A 3- dimensional lattice is the logical extension of a 2-dimensional one. It is not hard to imagine a 3- dimensional lattice. It consists of the lattice points in a 3-dimensional grid, with edges connecting adjacent points. A 3-dimensional mesh must have k3 nodes, for k = 1,2,3,...., like three-dimensional graph paper. While we cannot visually depict q-dimensional mesh networks when q > 3, we can describe their properties. A q-dimensional mesh network has kq nodes. k is the number of nodes in a single dimension of the mesh. Henceforth we let q denote the dimension of the mesh, and d, the number of nodes in a single dimension.

Communication is allowed only between neighboring nodes. The neighbors of a node are the nodes to which it is directly connected. The interior nodes of a mesh are connected to 2q other processors. (In the 2d case, they are connected to 4 processors; in the 3d case, to 6.) Noninterior nodes have fewer neighbors, depending upon their exact position. For example, "corner" nodes are connected to q processors.

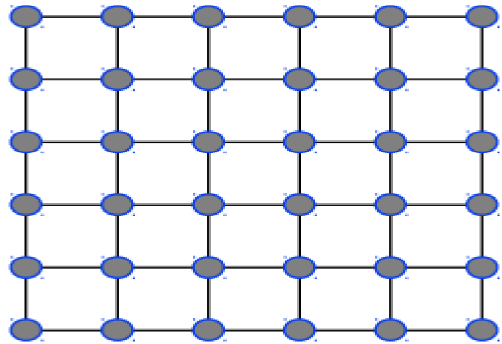
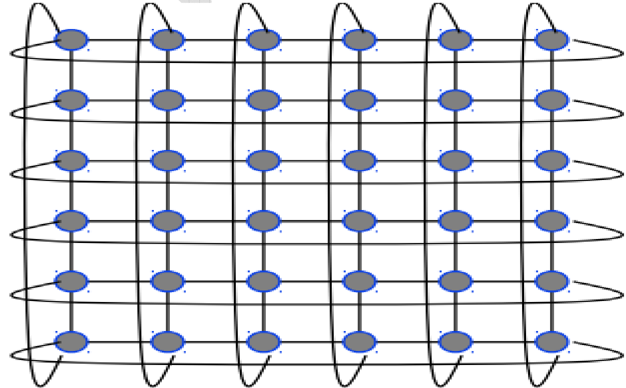


Figure 9 Two-dimensional mesh of size 36

We can evaluate the mesh network according to the criteria stated above. The diameter of a qdimensional mesh network with kq nodes is q(k - 1). The first step in understanding this is realizing that the farthest distance between nodes is from one corner to the diagonally opposite one. A recursive argument should convince you that the diameter is correct. In a 2-dimensional lattice with k2 nodes, you have to travel (k - 1) edges along the bottom row, then (k - 1) edges along the extreme column to get to the opposite corner. Thus you travel 2(k - 1) edges. Suppose we have a mesh of dimension q - 1. By assumption its diameter is (q-1)(k - 1). A mesh of one higher dimension has (k - 1) copies of the (q - 1)-dimensional mesh, side by side. To get from one corner to the opposite one, you have to travel to the corner of the (q - 1)-dimensional mesh first. That requires crossing (q - 1)(k - 1) edges, by hypothesis. Then we have to get to the kth copy of the mesh in the new dimension. We have to cross (k - 1) more edges. Thus we travel a total of (q - 1)(k - 1) + (k - 1) = q(k - 1) edges. The result is proved.

The diameter of a mesh increases as a linear function of the dimensionality of the mesh and the length of the side of the mesh; this can be a problem for many algorithms that require a lot of data movement, because data must be routed through many processors in the worst case.

Torus network



Partha Sarathi

An extension of a mesh is a torus. A torus, the 2-dimensional version of which is illustrated in the above Figure , is an extension of a mesh by the inclusion of edges between the exterior nodes in each row and those in each column. In higher dimensions, it includes edges between the exterior nodes in each dimension. It is called a torus because the surface that would be formed if it were wrapped around the nodes and edges with a thin film would be a mathematical torus, i.e., a doughnut

Binary Tree Networks

In a binary tree network the 2k -1 nodes are arranged into a complete binary tree of depth k-1. (See Figure 11) Recall that the depth of a binary tree is the maximum number of edges from the root to a leaf node. Each interior node is connected to two children, and each node other than the root is connected to its parent. Thus the maximum number of edges per node is three. The diameter of a binary tree network with 2k -1 nodes is only 2(k - 1), because the longest path in the tree is any path from a leaf node up to the root and then down to a different leaf node. If we let n = 2k -1 then 2(k-1) is approximately 2log2 n; i.e., the diameter of a binary tree network with n nodes is a logarithmic function of network size, which is very low.

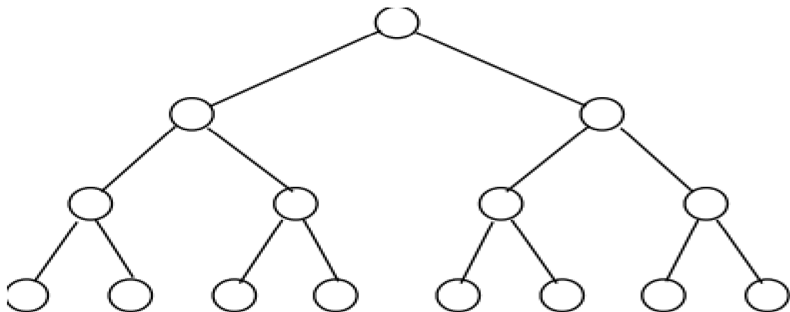


Figure 11 Binary tree network of depth 3 and size 15

Hypertree Networks (optional)

The problem with binary trees is their poor bisection width. Their advantage is their low diameter. A hypertree network is a modification of a binary tree with high bisection width but low diameter. The degree of a hypertree is the number of children per node. The degree of a hypertree must be a power of 2, i.e., 2, 4, 8, and so on, so we can have 2-ary, 4-ary, 8-ary 16-ary hypertrees etc. Unlike a binary tree, each node in a hypertree below the root level has two parents (just like people.) The best way to visualize a hypertree of degree k and depth d is to view it three dimensionally, using front and side views. In the frontal view, a hypertree of degree k and depth d looks like a k-ary tree of depth d, as shown in Figure 12. From the side, a hypertree looks like an upside down binary tree of depth d.

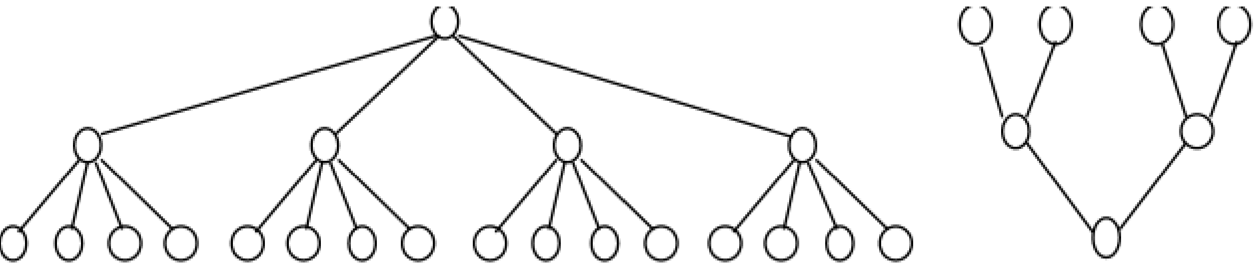


Figure 12 A 4-ary hypertree of depth 2 (front and side views)

Hypercube (Cube-Connected) Networks

A cube-connected network is a network with 2k nodes arranged as the vertices of a kdimensional cube, also called a hypercube. A square with edge length d is a 2D hypercube, consisting of the 4=22 vertices (0,0), (0,d), (d,d), and (d,0). A cube with edge length d is a 3D hypercube, consisting of the 8=23 vertices (0,0,0), (0,0,d), (0,d,d), (0,d,0), (d,0,0), (d,0,d), (d,d,d), and (d,d,0). This topology can be generalized to higher dimensions if we do not try to associate a picture with it. For simplicity assume that d=1, i.e., that the edge length is 1. A hypercube with unit length is called a unit hypercube, or simply a hypercube. The 4D hypercube consists of the 16 vertices

Node	Label
(0,0,0,0)	0
(0,0,0,1)	1
(0,0,1,1)	3
(0,0,1,0)	2
(0,1,1,0)	6
(0,1,1,1)	7
(0,1,0,1)	5
(0,1,0,0)	4
(1,1,0,0)	12
(1,1,0,1)	13
(1,1,1,1)	15
(1,1,1,0)	14
(1,0,1,0)	10
(1,0,1,1)	11
(1,0,0,1)	9
(1,0,0,0)	8

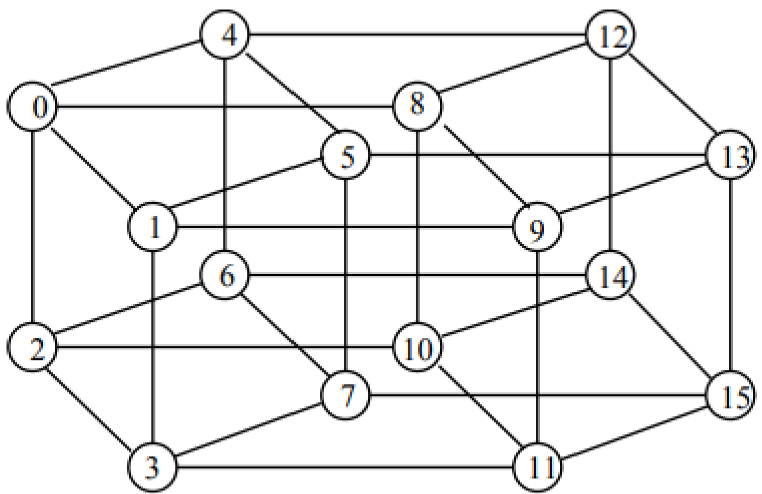


Figure 13: A four-dimensional hypercube.

If we could draw things in four dimensions, this would look like a cube of unit length in four dimensions. Notice that the coordinates of the points can be thought of as 4-bit binary representations of integers. The numbers adjacent to the vertices in the column to the right are the corresponding numbers. The integer representation of a node is called its label.

Crossbar Switching Networks

A crossbar matrix is an example of a dynamic network because the switches can be changed dynamically. Given p processors and m memory modules, a crossbar matrix consists of a rectangular grid of p*m switches, as illustrated in below Figure. In the figure, each Mi is a memory module and each Pj is a processor.

Switches are placed at the intersections of the horizontal and vertical lines in the matrix. Although the crossbar matrix in Figure 14 connects processors to memory modules, in general a crossbar connects input lines, which are the horizontal lines in the matrix, with output lines, which are the vertical lines; the inputs and outputs do not have to connect to processors and memories in general. Figure 15 shows how this same crossbar matrix could be used to connect the processors back to themselves.

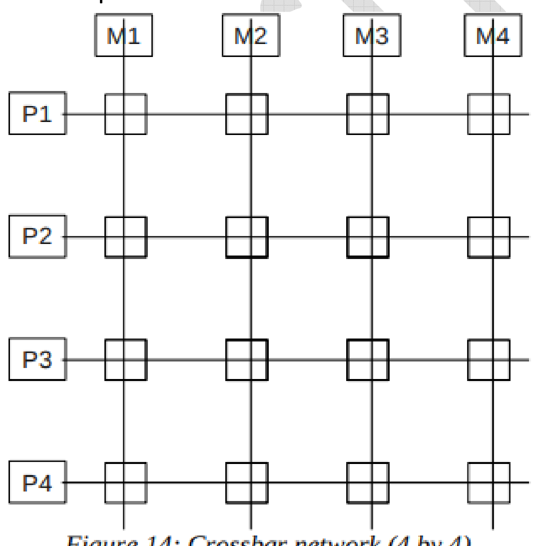


Figure 14: Crossbar network (4 by 4)

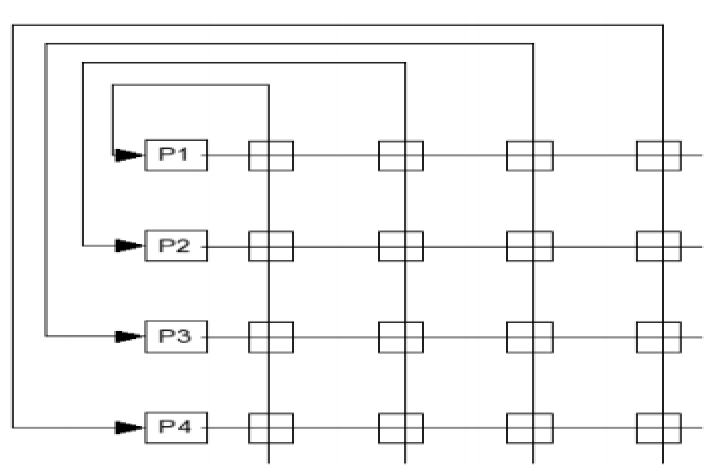
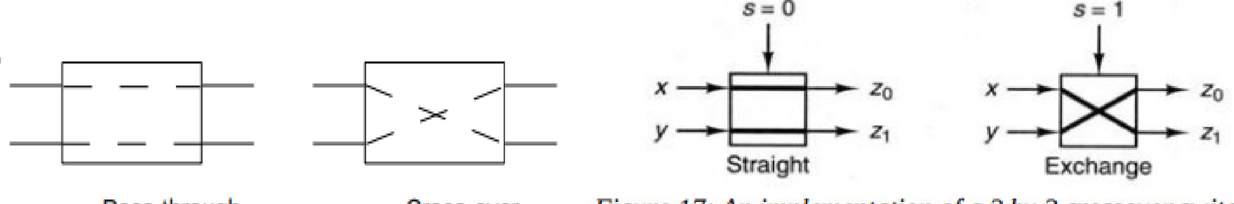


Figure 15: Crossbar switch used to connect processors to processors

Each switch in a crossbar matrix has 2 input lines and 2 output lines and can be in one of two states: pass-through or cross-over:



Pass-through Cross-over

Figure 17: An implementation of a 2 by 2 crossover switch

A crossbar network is a non-blocking network: access to one memory module by a processor does not prevent another processor from accessing a different memory module. More formally, **Definition**. A network is **non-blocking** if, for all sets of connections that can be established, all inactive input lines have access to all inactive output lines.

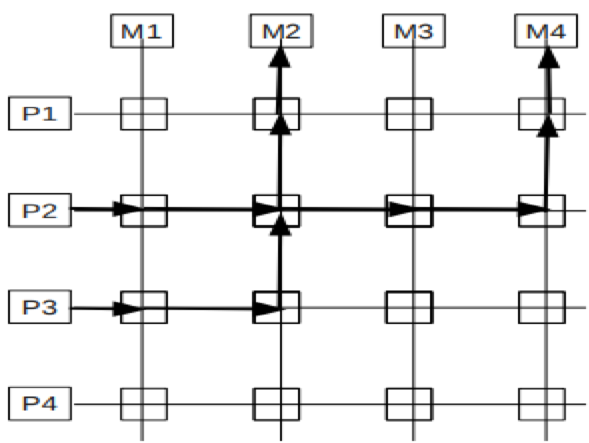


Figure 18: Crossbar matrix showing two simultaneous connections

Partha Sarathi

Multistage Networks

Certain network topologies are more suitable to be used in multistage networks. In a multistage network, instead of having a processor at each node, there is a switch at the node. The switches are less expensive and smaller and can be packed very densely. They are called multistage because a message travels from one switch to another in stages. Two common multistage networks are butterfly networks and omega networks.

Multistage networks are a compromise between buses and crossbars in terms of cost and performance -- they are less costly (Ω(p log p) switches versus Ω(pm) switches) than crossbars and slower than them, but faster than buses and more costly than them. A network with p processors will usually have log p stages. Messages travel from a processor to a memory module or other processor in successive stages across the network. These networks are generally blocking networks; two processors attempting to access different memory modules may not both be able to do so. This will be evident soon.

Butterfly Networks

A butterfly network represents a very different approach than the preceding ones. Butterfly networks are often used as multistage networks. A butterfly network consists of (k + 1)2k nodes arranged in k + 1 ranks, each containing n = 2k nodes. The ranks are labeled 0 through k. Sometimes ranks 0 and k are combined. Figure 19 depicts a butterfly network with 8 processor nodes. In the figure, the ranks are vertical. The rank 0 nodes would be connected to processors, and the rank 3 nodes could be connected either to processor nodes or to memory nodes, if the network were being used to implement a SMP machine. In a butterfly network, there is a path of length k from any node in rank 0 to any node in rank k+1. Since k is log n +1, where n is the number of inputs, this is a log n stage network.

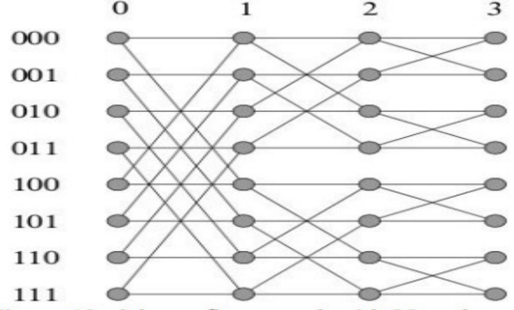


Figure 19: A butterfly network with 32 nodes.

Each switch in a butterfly network has two inputs and two outputs. The inputs come from two nodes in the preceding rank, and the outputs, to two nodes in the following rank. Which nodes depends upon the switch's position in the network.

Omega Network

The omega network is very similar to the butterfly6 . A three-stage omega network is illustrated in Figure 19. The only difference is in how the inputs and outputs of the switches are arranged. Like the butterfly and the crossbar, the switches are c22 switches. But in the omega network, the connections are based on a perfect shuffle network. A perfect shuffle is what happens to a deck of cards when it is split in half and shuffled perfectly. For example, if there are 8 cards numbered 0 through 7, then the shuffle puts them in the order 0 4 1 5 2 6 3 7. See Figure 21. The position that number k ends up in is its shuffle. For example, 1 ends up in position 2, 3 ends up in position 6. If you write the numbers in binary, observe that 001 goes to position 010 and 011 goes to position 110. In fact, take any number. Rotate it to the left, moving the most significant bit to the 0-bit position. Then the new value is the position of that number in the shuffle. Formally, if bn-1 bn-2 ...b0 is the binary representation of a node address, then the shuffle function can be described as

$$\text{shuffle}(b_{n-1}\, b_{n-2} \ldots b_0) = (b_{n-2}\, b_{n-3} \ldots b_0\, b_{n-1}).$$

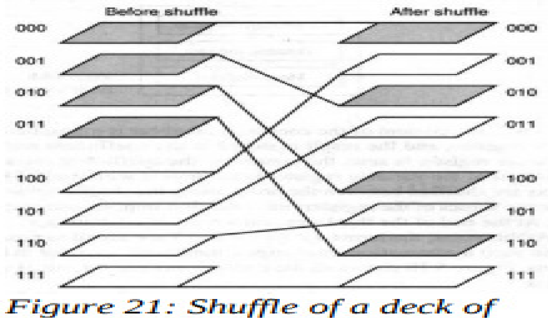


Figure 21: Shuffle of a deck of eight cards.

The shuffle operation determines where the output line k of switch floor(k/2) is connected. In Figure 22, the switch inputs and outputs are numbered from the top down as 000, 001, up to 111. Notice that in each stage, the output k is wired to the input whose value is shuffle(k).

A message is routed as follows: if s is a source and d is destination, the message is routed from s to d in log p stages as follows: at the first switch, if the leftmost bits of s and d are the same, pass-through state is used, otherwise crossover is used. At the next switch this is repeated: the second most significant bits are compared and used, and so on. This is also equivalent to the same routing algorithm used in the butterfly network -- if only the destination bit is used, then at switch k, if the kth most significant bit is 0, the upper output is used, otherwise the lower output is used.

Omega networks, are blocking networks. Suppose that processor Pk is connected to input line k in general and that output line m is connected to memory module Mm. If processor P2 is connected to memory M6 then the switch A3 is configured is cross-over, making it impossible for P6 to be connected to memory M4, as shown in Figure

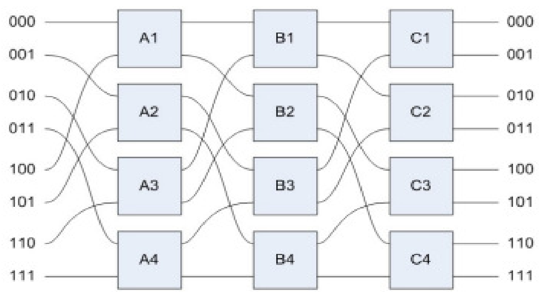


Figure 22: Three stage omega network (courtesy of Wikipedia)

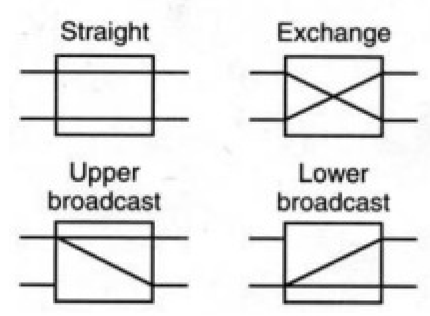


Figure 23: Four state C22 switch.

Omega networks are often designed to be broadcast networks. In a broadcast network, the switches can be in any of four different states.

Partha Sarathi

An omega network is used in the NYU Ultracomputer, a massively parallel machine with up to 4096 processors, to connect the processors to the memory banks. This omega network has the property that the switches have small buffers, enabling them to queue packets when a conflict occurs.

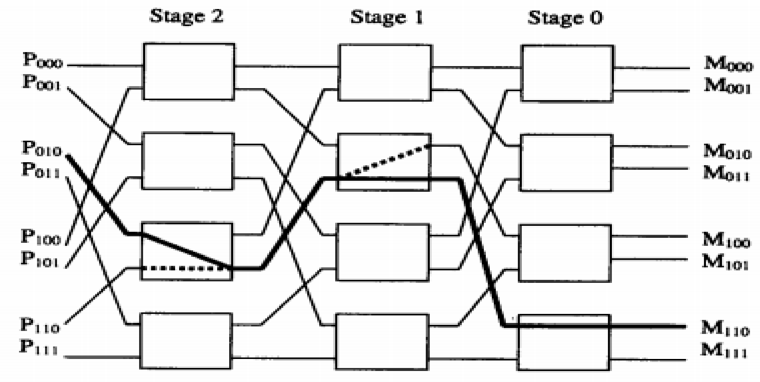


Figure 24: Omega network with P_2 connected to M_6

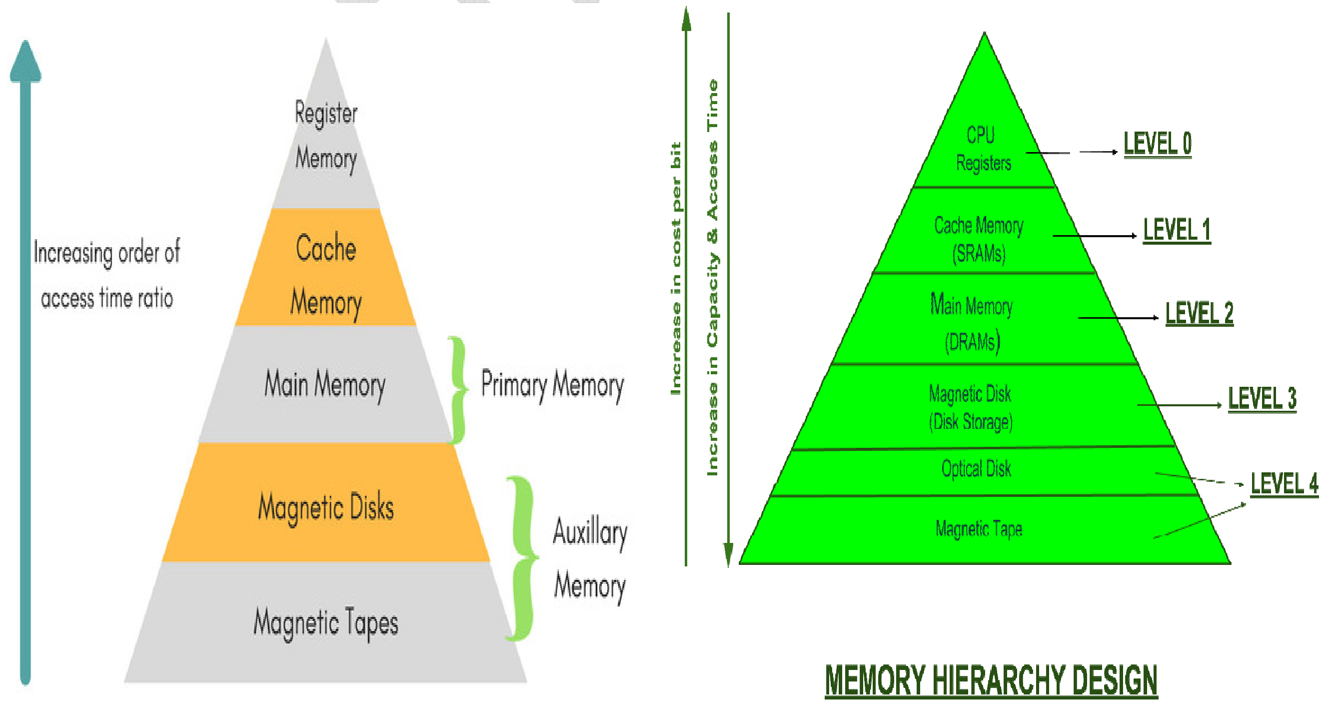
Mod-3

What is Memory Hierarchy?

The memory in a computer can be divided into five hierarchies based on the speed as well as use. The processor can move from one level to another based on its requirements. The five hierarchies in the memory are registers, cache, main memory, magnetic discs, and magnetic tapes. The first three hierarchies are volatile memories which mean when there is no power, and then automatically they lose their stored data. Whereas the last two hierarchies are not volatile which means they store the data permanently.
A memory unit is the collection of storage units or devices together. The memory unit stores the binary information in the form of bits. Generally, memory/storage is classified into 2 categories:

- Volatile Memory: This loses its data, when power is switched off.
- Non-Volatile Memory: This is a permanent storage and does not lose any data when power is switched off.

Memory Hierarchy in Computer Architecture



The **memory hierarchy design** in a computer system mainly includes different storage devices. Most of the computers were inbuilt with extra storage to run more powerfully beyond the main memory capacity. The following **memory hierarchy diagram** is a hierarchical pyramid for computer memory. The designing of the memory hierarchy is divided into two types such as primary (Internal) memory and secondary (External) memory.

Primary Memory

The primary memory is also known as internal memory, and this is accessible by the processor straightly. This memory includes main, cache, as well as CPU registers.

Secondary Memory

The secondary memory is also known as external memory, and this is accessible by the processor through an input/output module. This memory includes an optical disk, magnetic disk, and magnetic tape.

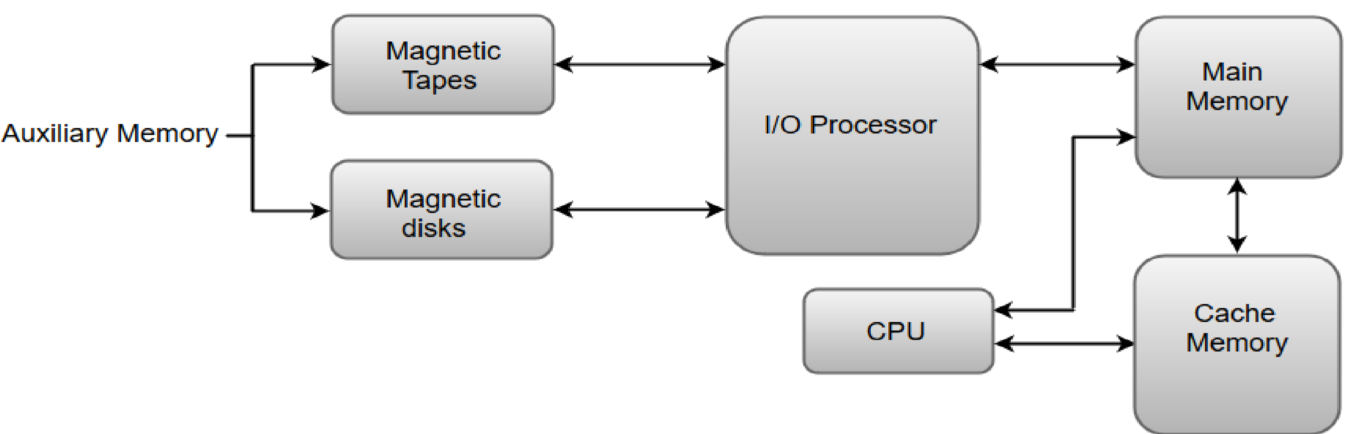
The total memory capacity of a computer can be visualized by hierarchy of components. The memory hierarchy system consists of all storage devices contained in a computer system from the slow Auxiliary Memory to fast Main Memory and to smaller Cache memory.

Auxillary memory access time is generally 1000 times that of the main memory, hence it is at the bottom of the hierarchy.The main memory occupies the central position because it is equipped to communicate directly with the CPU and with auxiliary memory devices through Input/output processor (I/O).

When the program not residing in main memory is needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space in main memory for other programs that are currently in use.

The cache memory is used to store program data which is currently being executed in the CPU. Approximate access time ratio between cache memory and main memory is about 1 to 7~10

Memory Hierarchy in a Computer System:



Memory Access Methods

Each memory type, is a collection of numerous memory locations. To access data from any memory, first it must be located and then the data is read from the memory location. Following are the methods to access information from memory locations:

1. Random Access: Main memories are random access memories, in which each memory location has a unique address. Using this unique address any memory location can be reached in the same amount of time in any order.
2. Sequential Access: This methods allows memory access in a sequence or in order.
3. Direct Access: In this mode, information is stored in tracks, with each track having a separate read/write head.

Main Memory

The memory unit that communicates directly within the CPU, Auxillary memory and Cache memory, is called main memory. It is the central storage unit of the computer system. It is a large and fast memory used to store data during computer operations. Main memory is made up of **RAM** and **ROM**, with RAM integrated circuit chips holing the major share.

- RAM: Random Access Memory
    - **DRAM**: Dynamic RAM, is made of capacitors and transistors, and must be refreshed every 10~100 ms. It is slower and cheaper than SRAM.
    - **SRAM**: Static RAM, has a six transistor circuit in each cell and retains data, until powered off.
    - **NVRAM**: Non-Volatile RAM, retains its data, even when turned off. Example: Flash memory.
- ROM: Read Only Memory, is non-volatile and is more like a permanent storage for information. It also stores the **bootstrap loader** program, to load and start the operating system when computer is turned on. **PROM**(Programmable ROM), **EPROM**(Erasable PROM) and **EEPROM**(Electrically Erasable PROM) are some commonly used ROMs.

Auxiliary Memory

Devices that provide backup storage are called auxiliary memory. **For example:** Magnetic disks and tapes are commonly used auxiliary devices. Other devices used as auxiliary memory are magnetic drums, magnetic bubble memory and optical disks.It is not directly accessible to the CPU, and is accessed using the Input/Output channels.

Cache Memory

The data or contents of the main memory that are used again and again by CPU, are stored in the cache memory so that we can easily access that data in shorter time.
Whenever the CPU needs to access memory, it first checks the cache memory. If the data is not found in cache memory then the CPU moves onto the main memory. It also transfers block of recent data into the cache and keeps on deleting the old data in cache to accomodate the new one.

Hit Ratio

The performance of cache memory is measured in terms of a quantity called **hit ratio**. When the CPU refers to memory and finds the word in cache it is said to produce a **hit**. If the word is not found in cache, it is in main memory then it counts as a **miss**.
The ratio of the number of hits to the total CPU references to memory is called hit ratio.
Hit Ratio = Hit/(Hit + Miss)

Associative Memory

It is also known as **content addressable memory (CAM)**. It is a memory chip in which each bit position can be compared. In this the content is compared in each bit cell which allows very fast table lookup. Since the entire chip can be compared, contents are randomly stored without considering addressing scheme. These chips have less storage capacity than regular memory chips.

Characteristics of Memory Hierarchy

We can infer the following characteristics of Memory Hierarchy Design from above figure:

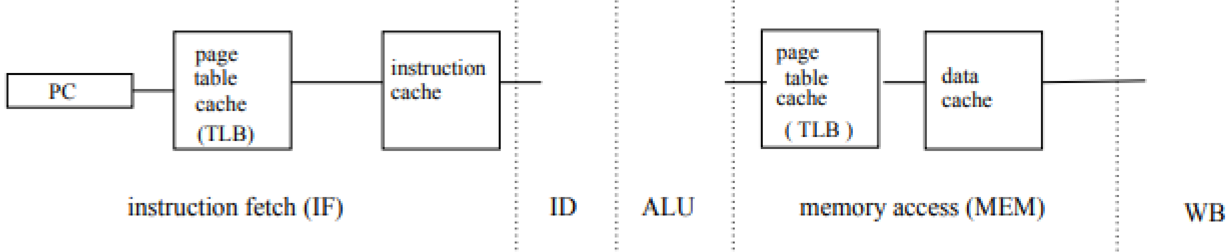
1. **Capacity:**
   It is the global volume of information the memory can store. As we move from top to bottom in the Hierarchy, the capacity increases.
2. **Access Time:**
   It is the time interval between the read/write request and the availability of the data. As we move from top to bottom in the Hierarchy, the access time increases.

Partha Sarathi

3. **Performance:**
   Earlier when the computer system was designed without Memory Hierarchy design, the speed gap increases between the CPU registers and Main Memory due to large difference in access time. This results in lower performance of the system and thus, enhancement was required. This enhancement was made in the form of Memory Hierarchy Design because of which the performance of the system increases. One of the most significant ways to increase system performance is minimizing how far down the memory hierarchy one has to go to manipulate data.
4. **Cost per bit:**
   As we move from bottom to top in the Hierarchy, the cost per bit increases i.e. Internal Memory is costlier than External Memory.

Data and instruction caches

The TLB provides a quick translation from a virtual address to a physical address in main memory. However, main memory is still a bit too slow to use every time we want an instruction or we want to access a data word. To speed up memory accesses, this is why we also use a cache for instructions and data.



case 1: each cache entry has one word (plus tag, etc)
Since MIPS instructions and data are often one word (4 bytes), it is natural to access 4-tuples of bytes at a time. Since the number of bytes in the instruction cache (or data cache) is 2^17 and we have one word per entry, the cache would have 2^15 entries holding one word (2^2 bytes) each.

Let's now run through the sequence of steps by which an instruction accesses a word in the cache. Suppose the processor is reading from the cache. In the case of an instruction cache, it is doing an instruction fetch. (In the case of a data cache, it is executing say a lw instruction.) The starting address of the word to be loaded is translated from virtual (32 bits) to physical (30 bits). We saw last class how the TLB does this. Then, the 30 bit physical address is used to try to find the word in the cache. How?

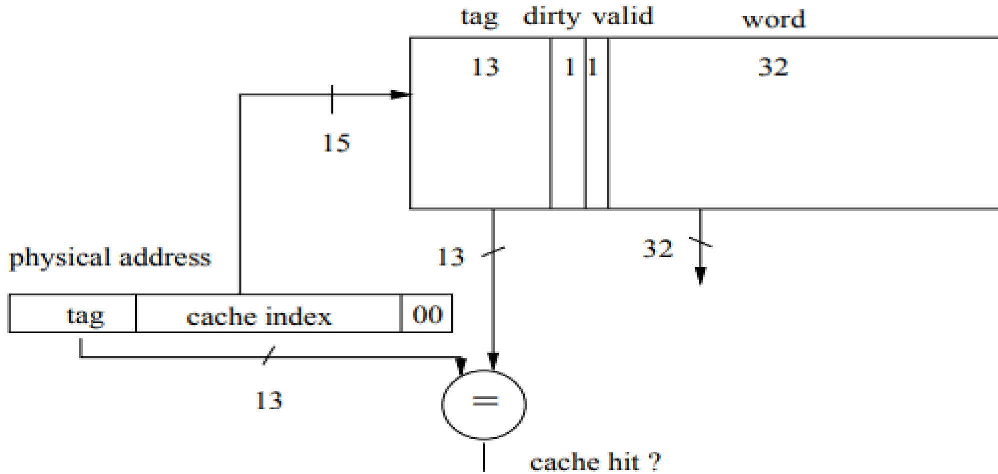
Assume the cache words are word aligned. That is, the physical addresses of the four bytes within each word in the cache have LSBs 11, 10, 01, 00. The lowest two bits of the address are called the byte offset since they specify one of the four bytes within a word. The next fifteen bits (2-16) are used to index into the 2^15 entries in the cache. We are assuming a read rather than write, so the entry is then read out of the cache. (Back in lecture 6, we looked at basic designs for retrieving bits from memory. You do not need to refresh all the details here, but it would be helpful to refresh the basic idea that you can feed 15 bits into a circuit and read out a row of data from a 2D array of flipflops.)

The upper 13 bits of the physical address (the tag) are compared to the 13 bit tag field at that cache entry. If the two are the same, and if the valid bit of that entry is on, then the word sitting at that entry of the cache is the correct one. If the upper 13 bits don't match the tag or if the valid bit is off, however, then the word cannot be loaded from the cache and that cache entry needs to be refilled from main memory. In this case, we say that a cache miss occurs. This is an exception and so a kernel program known as the cache miss handler takes over. (We will return to this later in the lecture.)

Using this approach, the cache memory would be organized as shown in the circuit below. For each of the 2^15 word entries in the cache, we store four fields:

- The word, namely a copy of the word that starts at the 30 bit physical (RAM) address represented by that cache entry.
- The upper 13 bits of that 30 bit physical address, called the tag. The idea of the tag is the same as we saw for the TLB. The tag is needed to distinguish all entries whose physical addresses have the same bits 2-16.
- A valid bit that specifies whether there is indeed something stored in the cache entry, or whether the tag and byte of data are junk bits, for example, leftover by a previous process that has terminated.
- A dirty bit that says whether the byte has been written to since it was brought into memory. We will discuss this bit later in the lecture.

The case of a write to the data cache e.g. sw uses a similar circuit. But there are some subtleties with writes. I will discuss these later.



case 2: each cache entry has a block of words (plus tag, etc)

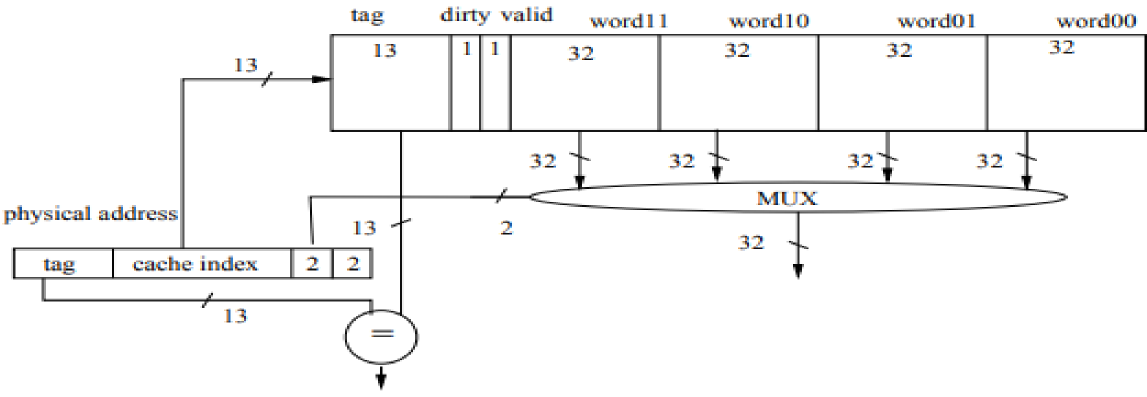
Case 1 above took advantage of a certain type of “spatial locality” of memory access, namely that bytes are usually accessed from memory in four-tuples (words). This is true both for instructions and data. Another type of spatial locality is that instructions are typically executed in sequence. (Branches and jumps occur only occasionally). At any time during the execution of a program we would like the instructions that follow the current one to be in the cache, since such instructions are likely to be all executed. Thus, whenever an instruction is copied into the cache, it would make sense to copy the neighboring instructions into the cache as well.

Spatial locality also arises for data word accesses. For example, arrays are stored in consecutive program addresses. Because pages are relatively large, neighboring program addresses tend to correspond to neighboring physical addresses. Similarly, words that are nearby on the stack tend to be accessed at similar times during the process execution.

A simple way to implement this idea of spatial locality is as follows: Rather than making the lines (rows) in the cache hold one word each, we let them hold (say) four words each. We take four words that are consecutive in memory (namely consecutive in both virtual memory and in physical memory). These consecutive words are called a block. If our cache holds 217 = 128 KB and each block holds 4 words or 4 × 4 = 24 bytes, then the cache can hold up to 213 blocks, so we would use 13 bits to index a line in the cache.

To address a word within a block, we need two bits, namely bits 2 and 3 of the physical address. (Bits 0 and 1 are the “byte offset” within a word.) Bits 2,3 are the block offset which specify one of four words within the block. Note that each block is 16 bytes (4 × 4) and the blocks are block aligned – they always start with the physical memory address that has 0000 in the least significant four bits. This might always work best, but it simplifies the circuitry.

As shown in the figure below, to access a word from the cache, we read an entire block out of the cache. Then, we select one of the four words in that block. One could draw a similar circuit for writing a word (or block) to the cache, although the multiplexor part would need to be changed.



Hits and misses

The above discussion was about indexing mechanisms, namely how we read from a cache assuming that the cache has the word we want (a hit). We next consider two other aspects of caches. The first is what happens when the address we are indexing is not in the cache: a cache miss. The second aspect is what happens when we write to a cache, either from a register to the cache (in the case of a store word), or when we copy a block from main memory to the cache in the case of a miss. Specifically, we are concerned here with consistency between the cache and main memory.

Instruction Cache

The instruction and data caches have a subtle difference: instructions are only fetched (read) from memory, but data can be read from or written to memory. For the instruction cache, blocks are copied from main memory to the cache. For the data cache, blocks can be copied either from main memory to the cache, or vice-versa. There can also be writes from registers to the data cache e.g. by sw (or swc1) instructions.

We begin with the instruction cache, which is simpler. If we are fetching and the instruction is in the cache, i.e. a hit, then we have the case discussed earlier in the lecture. If the instruction is not in the cache, however, we have a miss. This causes an exception – a branch to the exception handler which then branches to the cache miss handler. This kernel program arranges for the appropriate block in main memory (the one that contains the instruction) to be brought into the cache. The valid bit for that cache entry is then set, indicating that the block now in the cache indeed represents a valid block in main memory. The cache miss handler can then return control to the process and we try again to fetch the instruction. Of course, this time there is a hit.

Data Cache - write-through policy

The data cache is more complicated. Since we can write to the data cache (sw), it can easily happen that a cache line does not have the same data as the corresponding block in main memory. There are two policies for dealing with this issue: "write-through" and "write-back". The write-through policy ensures that the cache block is consistent with its corresponding main memory block. We describe it first.

Consider reading from the data cache (as in lw or lwc1). If the word is in the cache, then we have a hit and this case was covered earlier. If there is a miss, however, then an exception occurs and we replace that cache entry (namely, the entire block). The previous entry of the cache is erased in the process. This is no problem for the write-through policy since this policy ensures that the cache line (just erased) has the same data as its corresponding block in main memory, and so the erased data is not lost.

Consider happens when we write a word from a register to the cache (sw or swc1). First suppose that there is a hit: the cache has the correct entry. The word is copied from the register to the cache and also the word is copied back to the appropriate block in main memory, so that main memory and cache are consistent (i.e. write through).

If the desired block is not in the cache, then an exception occurs – a cache miss. The cache miss handler arranges that the appropriate block is transferred from main memory to the cache. The handler then returns control to the program which tries to write again (and succeeds this time – a cache hit). Here is a summary of the"write through (data cache)" policy:

Partha Sarathi

	hit	miss
read (lw)	copy word cache → reg	copy block main mem → cache (and set valid bit) copy word cache → register
write (sw)	copy word reg → cache copy word cache → main mem	copy block main memory → cache (and set valid bit) copy word register → cache copy word cache → main memory

Data cache: "write back" policy

The second policy avoids copying the updated cache block back to main memory unless it is absolutely necessary. Instead, by design, each entry in the cache holds the most recent version of a block in main memory. The processor can write to and read from a block in the cache as many times as it likes without updating main memory – as long as there are hits. The only special care that must be taken is when there is a cache miss. In this case, the entry in the cache must be replaced by a new block. But before this new block can be read into the cache, the old block must be written back into main memory so that inconsistencies between the block in the cache (more recent version) and the block in main memory (older version) are not lost. This is how the "write back" scheme delays the writing of the block to memory until it is really necessary.

To keep track of which lines in the cache are consistent with their corresponding blocks in main memory, , a dirty bit is used – one per cache line. When a block is first brought into the cache, the dirty bit is set to 0. When a word is written from a register to a cache block (e.g. sw), the dirty bit is set to 1 to indicate that at least one word in the block no longer corresponds to the word in main memory.

Later, when a cache miss occurs at the cache line, and when the dirty bit is 1, the data block at that cache line needs to be written back to main memory, before the new (desired) block can be brought into the cache. That is, we "write-back". This policy only writes a block back to main memory when it is really necessary, i.e. when the block needs to be replaced by another block. Note that there is just one dirty bit for the whole block. This bit doesn't indicate which word(s) is dirty. So the whole block is written back to main memory and a whole new block is brought in.

This "write-back" policy helps performance if there are several writes to individual words in a block in the cache before that block is replaced by another

The following table summarizes the steps of the data cache write-back policy.

	hit	miss
read	copy word: cache → reg	copy block: cache → main mem (only if valid and dirty bits are 1) copy block: main memory → cache and set dirty bit = 0, valid bit = 1 copy word: cache → register
write	copy word: reg → cache (and set dirty bit = 1)	copy block: cache → main mem (only if valid and dirty bits are 1) copy block: main mem → cache (and set valid bit = 1) copy word: reg → cache (and set dirty bit = 1)

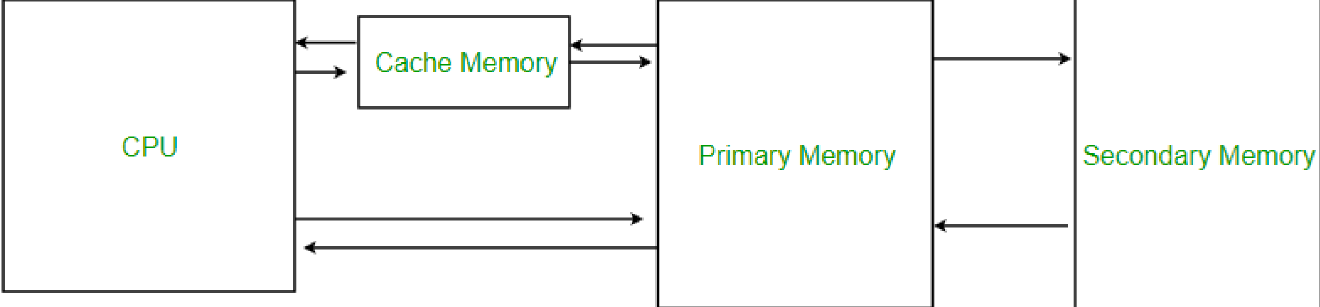
Note that the misses take more time than the hits. So this policy only makes sense when the hits are much more frequent than the misses.

Cache Memory in Computer Organization

Cache Memory is a special very high-speed memory. It is used to speed up and synchronizing with high-speed CPU. Cache memory is costlier than main memory or disk memory but economical than CPU registers. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.

Cache memory is used to reduce the average time to access data from the Main memory. The cache is a smaller and faster memory which stores copies of the data from frequently used main memory locations. There are various different independent caches in a CPU, which store instructions and data.

Partha Sarathi



Levels of memory:

- **Level 1 or Register –**
  It is a type of memory in which data is stored and accepted that are immediately stored in CPU. Most commonly used register is accumulator, Program counter, address register etc.
- **Level 2 or Cache memory –**
  It is the fastest memory which has faster access time where data is temporarily stored for faster access.
- **Level 3 or Main Memory –**
  It is memory on which computer works currently. It is small in size and once power is off data no longer stays in this memory.
- **Level 4 or Secondary Memory –**
  It is external memory which is not as fast as main memory but data stays permanently in this memory.

Cache Performance:

When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache.

- If the processor finds that the memory location is in the cache, a **cache hit** has occurred and data is read from cache
- If the processor **does not** find the memory location in the cache, a **cache miss** has occurred. For a cache miss, the cache allocates a new entry and copies in data from main memory, then the request is fulfilled from the contents of the cache.

The performance of cache memory is frequently measured in terms of a quantity called **Hit ratio.**

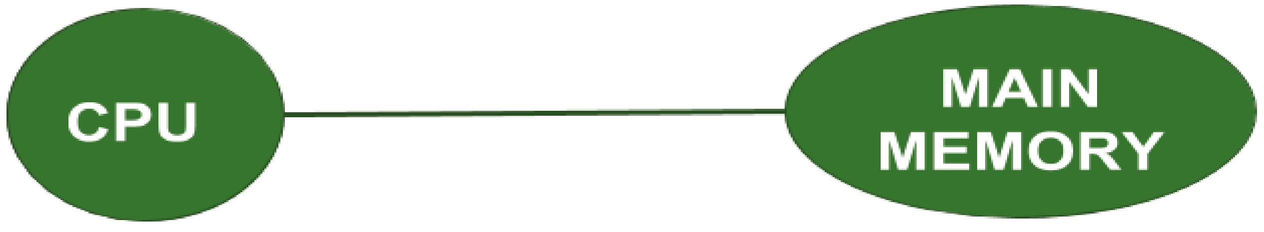
Hit ratio = hit / (hit + miss) = no. of hits/total accesses

We can improve Cache performance using higher cache block size, higher associativity, reduce miss rate, reduce miss penalty, and reduce Reduce the time to hit in the cache.

Multilevel Caches is one of the techniques to improve Cache Performance by reducing the *"MISS PENALTY"*. Miss Penalty refers to the extra time required to bring the data into cache from the Main memory whenever there is a ***"miss"*** in cache .

For clear understanding let us consider an example where CPU requires 10 Memory References for accessing the desired information and consider this scenario in the following 3 cases of System design :

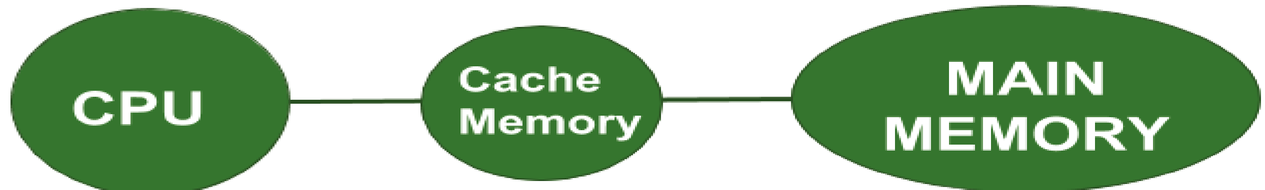
Case 1 : System Design without Cache Memory



Partha Sarathi

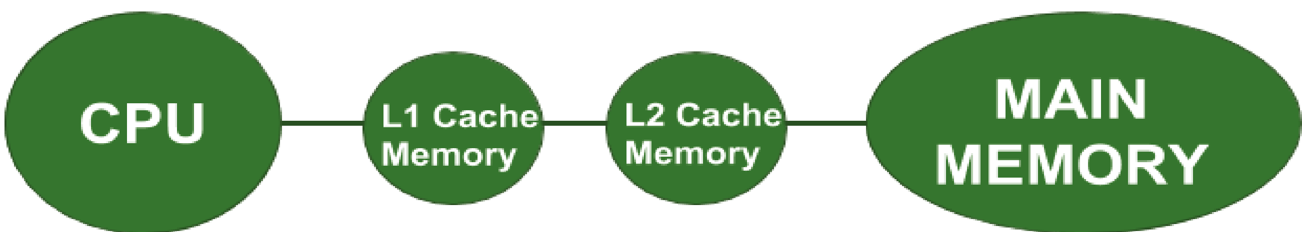
Here the CPU directly communicates with the main memory and no caches are involved.
In this case, the CPU needs to access the main memory 10 times to access the desired information.

Case 2 : System Design with Cache Memory



Here the CPU at first checks whether the desired data is present in the Cache Memory or not i.e. whether there is a ***"hit"*** in cache or ***"miss"*** in cache. Suppose there are *3 miss* in Cache Memory then the Main Memory will be accessed only 3 times. We can see that here the miss penalty is reduced because the Main Memory is accessed a lesser number of times than that in the previous case.

Case 3 : System Design with Multilevel Cache Memory



Here the Cache performance is optimized further by introducing multilevel Caches. As shown in the above figure, we are considering 2 level Cache Design. Suppose there are *3 miss* in the L1 Cache Memory and out of these 3 misses there are *2 miss* in the L2 Cache Memory then the Main Memory will be accessed only 2 times. It is clear that here the Miss Penalty is reduced considerably than that in the previous case thereby improving the Performance of Cache Memory.

NOTE :

We can observe from the above 3 cases that we are trying to decrease the number of Main Memory References and thus decreasing the Miss Penalty in order to improve the overall System Performance. Also, it is important to note that in the Multilevel Cache Design, L1 Cache is attached to the CPU and it is small in size but fast. Although, L2 Cache is attached to the Primary Cache i.e. L1 Cache and it is larger in size and slower but still faster than the Main Memory.

Effective Access Time = Hit rate * Cache access time + Miss rate * Lower level access time

Average access time (AAT)

Caches, being small in size, may result in frequent misses – when a search of the cache does not provide the sought-after information – resulting in a call to main memory to fetch data. Hence, the AAT is affected by the miss rate of each structure from which it searches for the data.

AAT for main memory is given by $\textbf{Hit time}_{\textbf{main memory}}$. AAT for caches can be given by

$$\textbf{Hit time}_{\textbf{cache}} + (\textbf{Miss rate}_{\textbf{cache}} \times \textbf{Miss Penalty}_{\textbf{time taken to go to main memory after missing cache}}).$$

The hit time for caches is less than the hit time for the main memory, so the AAT for data retrieval is significantly lower when accessing data through the cache rather than main memory.

Average access Time For Multilevel Cache: $T_{avg} = H_1 * C_1 + (1 - H_1) * (H_2 * C_2 + (1 - H_2) * M)$

Where H1 is the Hit rate in the L1 caches.

H2 is the Hit rate in the L2 cache.

C1 is the Time to access information in the L1 caches.

C2 is the Miss penalty to transfer information from the L2 cache to an L1 cache.

M is the Miss penalty to transfer information from the main memory to the L2 cache.

Example:

Find the Average memory access time for a processor with a 2 ns clock cycle time, a miss rate of 0.04 misses per instruction, a miss penalty of 25 clock cycles, and a cache access time (including hit detection) of 1 clock cycle. Also, assume that the read and write miss penalties are the same and ignore other write stalls.

Solution:

Average Memory access time(AMAT)= Hit Time + Miss Rate * Miss Penalty.

Hit Time = 1 clock cycle (Hit time = Hit rate * access time) but here Hit time is directly given so,

Miss rate = 0.04

Miss Penalty= 25 clock cycle (this is time taken by the above level of memory after the hit)

so, AMAT= 1 + 0.04 * 25

AMAT= 2 clock cycle

according to question 1 clock cycle = 2 ns

AMAT = 4ns

Trade-offs

While using the cache may improve memory latency, it may not always result in the required improvement for the time taken to fetch data due to the way caches are organized and traversed. For example, direct-mapped caches that are the same size usually have a higher miss rate than fully associative caches. This may also depend on the benchmark of the computer testing the processor and on the pattern of instructions. But using a fully associative cache may result in more power consumption, as it has to search the whole cache every time. Due to this, the trade-off between power consumption (and associated heat) and the size of the cache becomes critical in the cache design.

Evolution

In the case of a cache miss, the purpose of using such a structure will be rendered useless and the computer will have to go to the main memory to fetch the required data. However, with a multiple-level cache, if the computer misses the cache closest to the processor (level-one cache or L1) it will then search through the next-closest level(s) of cache and go to main memory only if these methods fail. The general trend is to keep the L1 cache small and at a distance of 1–2 CPU clock cycles from the processor, with the lower levels of caches increasing in size to store more data than L1, hence being more distant but with a lower miss rate. This results in a better AAT. The number of cache levels can be designed by architects according to their requirements after checking for trade-offs between cost, AATs, and size.

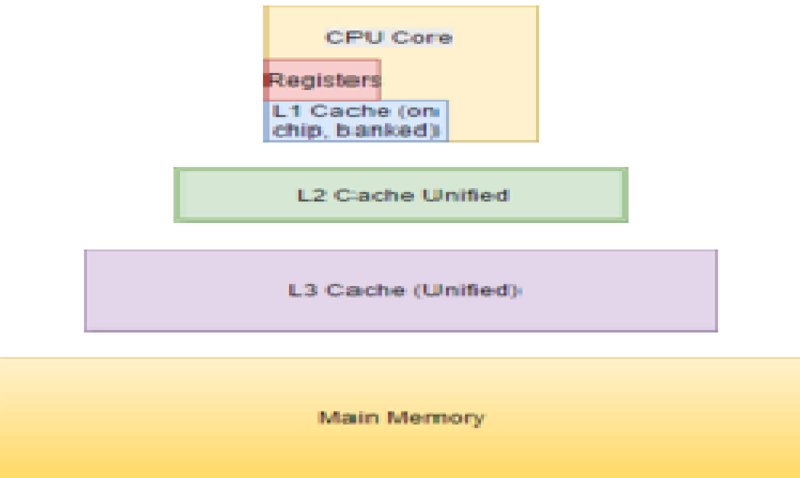
Performance gains

With the technology-scaling that allowed memory systems able to be accommodated on a single chip, most modern day processors have up to three or four cache levels.The reduction in the AAT can be understood by this example, where the computer checks AAT for different configurations up to L3 caches.

Example: main memory = 50 ns, L1 = 1 ns with 10% miss rate, L2 = 5 ns with1% miss rate), L3 = 10 ns with 0.2% miss rate.

- No cache, AAT = 50 ns
- L1 cache, AAT = 1 ns + (0.1 × 50 ns) = 6 ns
- L1–2 caches, AAT = 1 ns + (0.1 × [5 ns + (0.01 × 50 ns)]) = 1.55 ns
- L1–3 caches, AAT = 1 ns + (0.1 × [5 ns + (0.01 × [10 ns + (0.002 × 50 ns)])]) = 1.5001 ns

Partha Sarathi



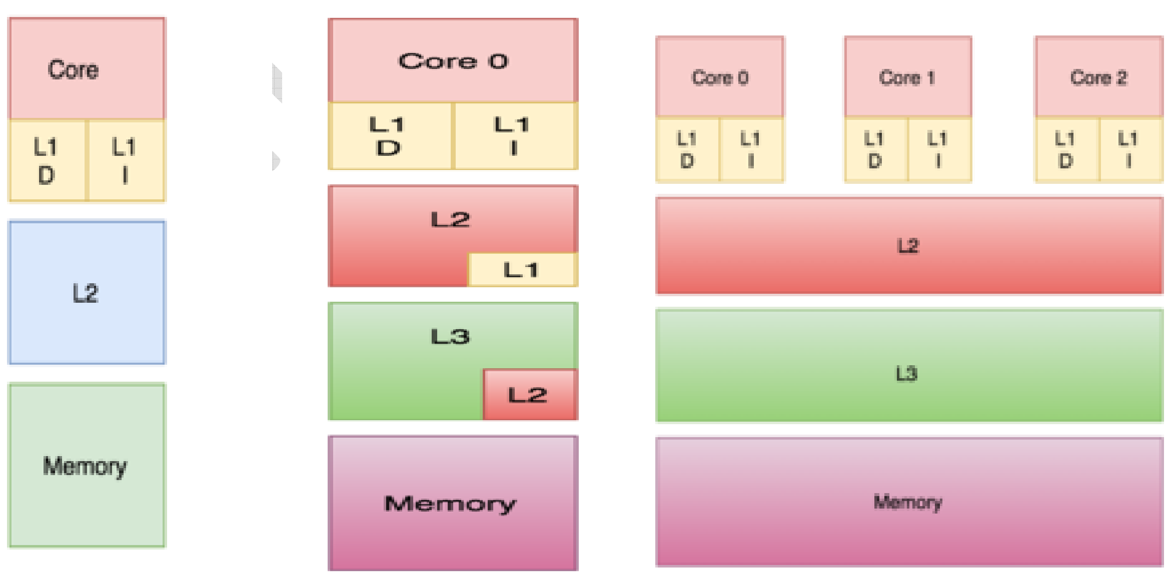
Disadvantages

- Cache memory comes at an increased marginal cost than main memory and thus can increase the cost of the overall system.
- Cached data is stored only so long as power is provided to the cache.
- Increased on-chip area required for memory system.
- Benefits may be minimized or eliminated in the case of a large programs with poor temporal locality, which frequently access the main memory.

Properties

Banked versus unified

In a banked cache, the cache is divided into a cache dedicated to instruction storage and a cache dedicated to data. In contrast, a unified cache contains both the instructions and data in the same cache. During a process, the L1 cache (or most upper-level cache in relation to its connection to the processor) is accessed by the processor to retrieve both instructions and data. Requiring both actions to be implemented at the same time requires multiple ports and more access time in a unified cache. Having multiple ports requires additional hardware and wiring, leading to a significant structure between the caches and processing units.To avoid this, the L1 cache is often organized as a banked cache which results in fewer ports, less hardware, and generally lower access times.
Modern processors have split caches, and in systems with multilevel caches higher level caches may be unified while lower levels split.



A) Cache organization with L1 as separate and L2 as unified
B) Inclusive cache organization
C) Cache organization with L1 private and L2 and L3 shared

Inclusion policies

Whether a block present in the upper cache layer can also be present in the lower cache level is governed by the memory system's inclusion policy, which may be inclusive, exclusive or non-inclusive non-exclusive (NINE).

With an inclusive policy, all the blocks present in the upper-level cache have to be present in the lower-level cache as well. Each upper-level cache component is a subset of the lower-level cache component. In this case, since there is a duplication of blocks, there is some wastage of memory. However, checking is faster.

Under an exclusive policy, all the cache hierarchy components are completely exclusive, so that any element in the upper-level cache will not be present in any of the lower cache components. This enables complete usage of the cache memory. However, there is a high memory-access latency.

The above policies require a set of rules to be followed in order to implement them. If none of these are forced, the resulting inclusion policy is called non-inclusive non-exclusive (NINE). This means that the upper-level cache may or may not be present in the lower-level cache.

Write policies

There are two policies which define the way in which a modified cache block will be updated in the main memory: write through and write back.

In the case of write through policy, whenever the value of the cache block changes, it is further modified in the lower-level memory hierarchy as well. This policy ensures that the data is stored safely as it is written throughout the hierarchy.

However, in the case of the write back policy, the changed cache block will be updated in the lower-level hierarchy only when the cache block is evicted. A "dirty bit" is attached to each cache block and set whenever the cache block is modified. During eviction, blocks with a set dirty bit will be written to the lower-level hierarchy. Under this policy, there is a risk for data-loss as the most recently changed copy of a datum is only stored in the cache and therefore some corrective techniques must be observed.

In case of a write where the byte is not present in the cache block, the byte may be brought to the cache as determined by a write allocate or write no-allocate policy. Write allocate policy states that in case of a write miss, the block is fetched from the main memory and placed in the cache before writing.In the write no-allocate policy, if the block is missed in the cache it will write in the lower-level memory hierarchy without fetching the block into the cache

Shared versus private

A private cache is assigned to one particular core in a processor, and cannot be accessed by any other cores. In some architectures, each core has its own private cache; this creates the risk of duplicate blocks in a system's cache architecture, which results in reduced capacity utilization. However, this type of design choice in a multi-layer cache architecture can also lend itself to a lower data-access latency.

A shared cache is a cache which can be accessed by multiple cores. Since it is shared, each block in the cache is unique and therefore has a larger hit rate as there will be no duplicate blocks. However, data-access latency can increase as multiple cores try to access the same cache.

In multi-core processors, the design choice to make a cache shared or private impacts the performance of the processor. In practice, the upper-level cache L1 (or sometimes L2) is implemented as private and lower-level caches are implemented as shared. This design provides high access rates for the high-level caches and low miss rates for the lower-level caches.

Cache Mapping:

When cache hit occurs:

- The required word is present in the cache memory.
- The required word is delivered to the CPU from the cache memory.

Partha Sarathi

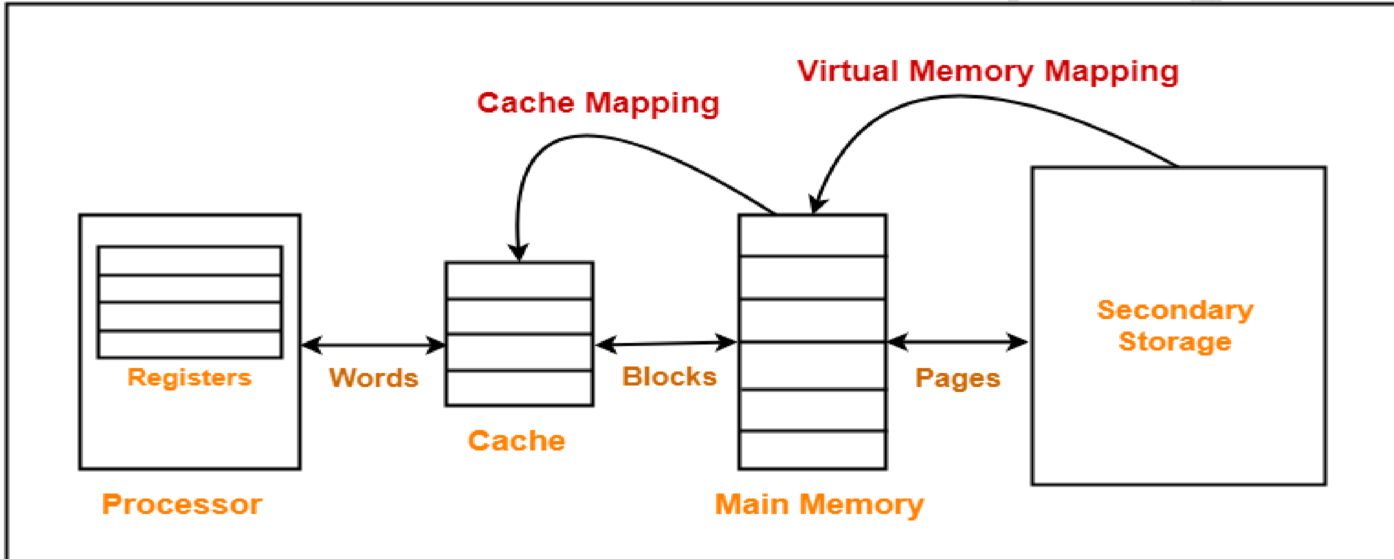
When cache miss occurs:

- The required word is not present in the cache memory.
- The page containing the required word has to be mapped from the main memory.
- This mapping is performed using cache mapping techniques.

Cache Mapping-

- Cache mapping defines how a block from the main memory is mapped to the cache memory in case of a cache miss.
  OR
- Cache mapping is a technique by which the contents of main memory are brought into the cache memory.

The following diagram illustrates the mapping process-



NOTES

- Main memory is divided into equal size partitions called as **blocks** or **frames**.
- Cache memory is divided into partitions having same size as that of blocks called as **lines**.
- During cache mapping, block of main memory is simply copied to the cache and the block is not actually brought from the main memory.

Cache Mapping Techniques-

Cache mapping is performed using following three different techniques-

1. Direct Mapping
2. Fully Associative Mapping
3. K-way Set Associative Mapping

1. Direct Mapping-

The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. Or In Direct mapping, assignee each memory block to a specific line in the cache. If a line is previously taken up by a memory block when a new block needs to be loaded, the old block is trashed. An address space is split into two parts index field and a tag field. The cache is used to store the tag field whereas the rest is stored in the main memory. Direct mapping`s performance is directly proportional to the Hit ratio.

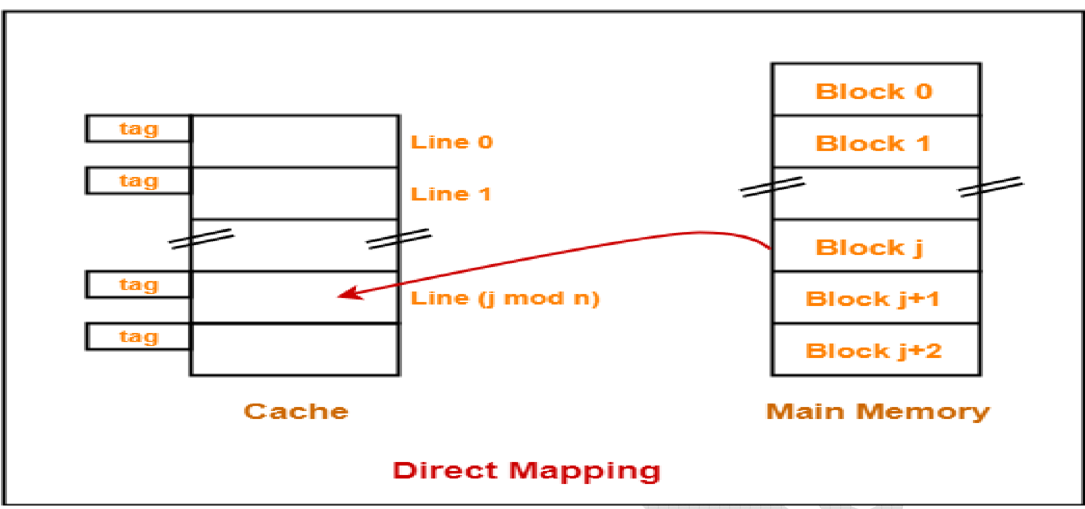
i = j modulo m
where
i=cache line number
j= main memory block number
m=number of lines in the cache

For purposes of cache access, each main memory address can be viewed as consisting of three fields. The least significant w bits identify a unique word or byte within a block of main memory. In most contemporary machines, the address is at the byte level. The remaining s bits specify one of the 2^s blocks of main memory. The cache logic interprets these s bits as a tag of s-r bits (most significant portion) and a line field of r bits. This latter field identifies one of the $m=2^r$ lines of the cache.

Cache line number = (Main Memory Block Address) Modulo (Number of lines in Cache)

Example-

- Consider cache memory is divided into 'n' number of lines.
- Then, block 'j' of main memory can map to line number (j mod n) only of the cache.



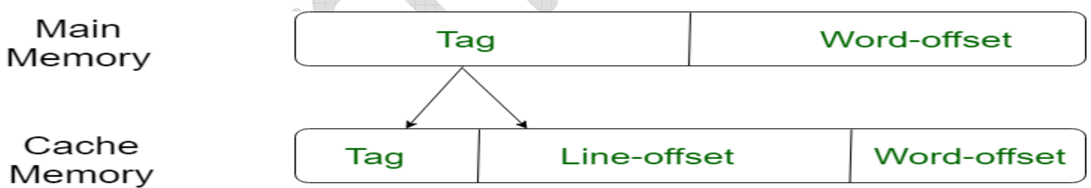
Need of Replacement Algorithm-

In direct mapping,

- There is no need of any replacement algorithm.
- This is because a main memory block can map only to a particular line of the cache.
- Thus, the new incoming block will always replace the existing block (if any) in that particular line.

Division of Physical Address-

In direct mapping, the physical address is divided as-



2. Fully Associative Mapping-

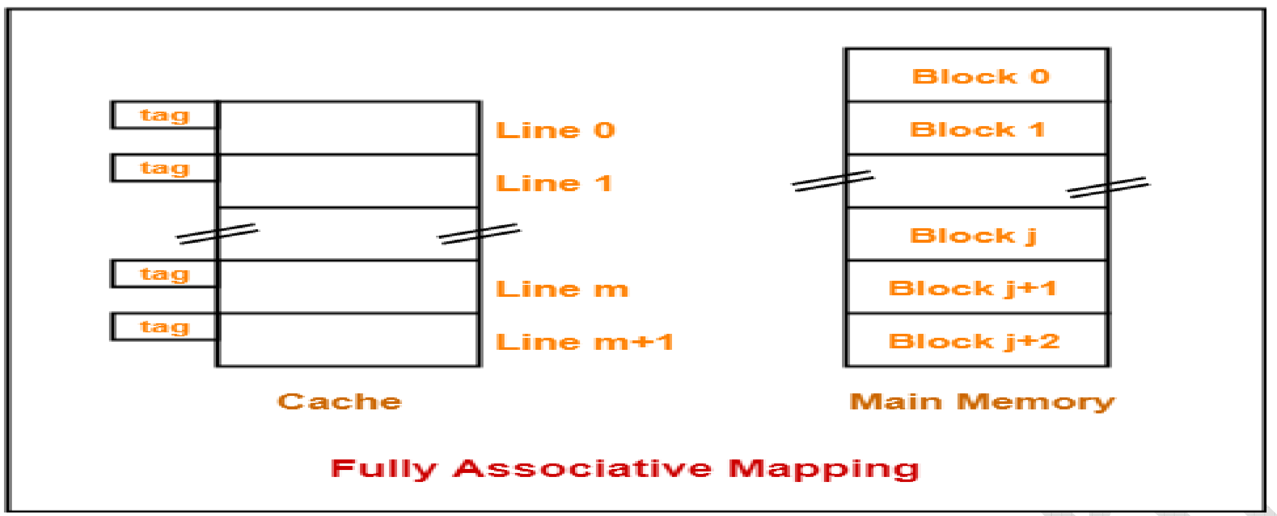
In this type of mapping, the associative memory is used to store content and addresses of the memory word. Any block can go into any line of the cache. This means that the word id bits are used to identify which word in the block is needed, but the tag becomes all of the remaining bits. This enables the placement of any word at any place in the cache memory. It is considered to be the fastest and the most flexible mapping form.

Example-

Consider the following scenario-

Here,

- All the lines of cache are freely available.
- Thus, any block of main memory can map to any line of the cache.
- Had all the cache lines been occupied, then one of the existing blocks will have to be replaced.



Need of Replacement Algorithm-

In fully associative mapping,

- A replacement algorithm is required.
- Replacement algorithm suggests the block to be replaced if all the cache lines are occupied.
- Thus, replacement algorithm like FCFS Algorithm, LRU Algorithm etc is employed.

Division of Physical Address-

In fully associative mapping, the physical address is divided as-

Block Number / Tag	Block / Line Offset

Division of Physical Address in Fully Associative Mapping

3. K-way Set Associative Mapping-

This form of mapping is an enhanced form of direct mapping where the drawbacks of direct mapping are removed. Set associative addresses the problem of possible thrashing in the direct mapping method. It does this by saying that instead of having exactly one line that a block can map to in the cache, we will group a few lines together creating a ***set***. Then a block in memory can map to any one of the lines of a specific set..Set-associative mapping allows that each word that is present in the cache can have two or more words in the main memory for the same index address. Set associative cache mapping combines the best of direct and associative cache mapping techniques.

In this case, the cache consists of a number of sets, each of which consists of a number of lines. The relationships are

m = v * k

i= j mod v

where

i=cache set number

j=main memory block number

v=number of sets

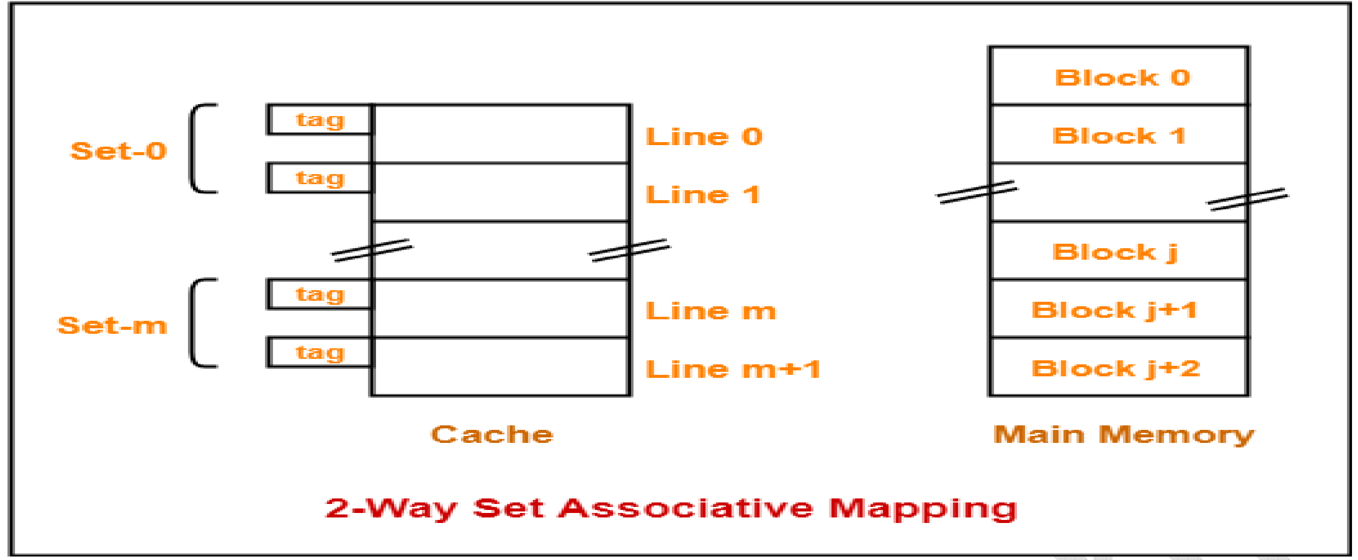
m=number of lines in the cache number of sets

k=number of lines in each set

Cache set number = (Main Memory Block Address) Modulo (Number of sets in Cache)

Example-

Consider the following example of 2-way set associative mapping-



Here,

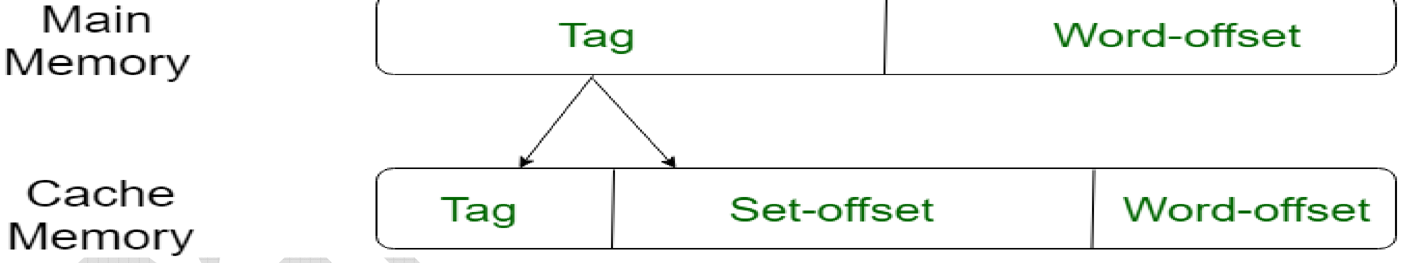
- k = 2 suggests that each set contains two cache lines.
- Since cache contains 6 lines, so number of sets in the cache = 6 / 2 = 3 sets.
- Block 'j' of main memory can map to set number (j mod 3) only of the cache.
- Within that set, block 'j' can map to any cache line that is freely available at that moment.
- If all the cache lines are occupied, then one of the existing blocks will have to be replaced.

Need of Replacement Algorithm-

- Set associative mapping is a combination of direct mapping and fully associative mapping.
- It uses fully associative mapping within each set.
- Thus, set associative mapping requires a replacement algorithm.

Division of Physical Address-

In set associative mapping, the physical address is divided as-



Special Cases-

- If k = 1, then k-way set associative mapping becomes direct mapping i.e.
  1-way Set Associative Mapping ≡ Direct Mapping
- If k = Total number of lines in the cache, then k-way set associative mapping becomes fully associative mapping.

Application of Cache Memory –

1. Usually, 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.
2. The correspondence between the main memory blocks and those in the cache is specified by a mapping function.

Types of Cache –

- **Primary Cache –**
  A primary cache is always located on the processor chip. This cache is small and its access time is comparable to that of processor registers.
- **Secondary Cache –**
  Secondary cache is placed between the primary cache and the rest of the memory. It is referred to as the level 2 (L2) cache. Often, the Level 2 cache is also housed on the processor chip.

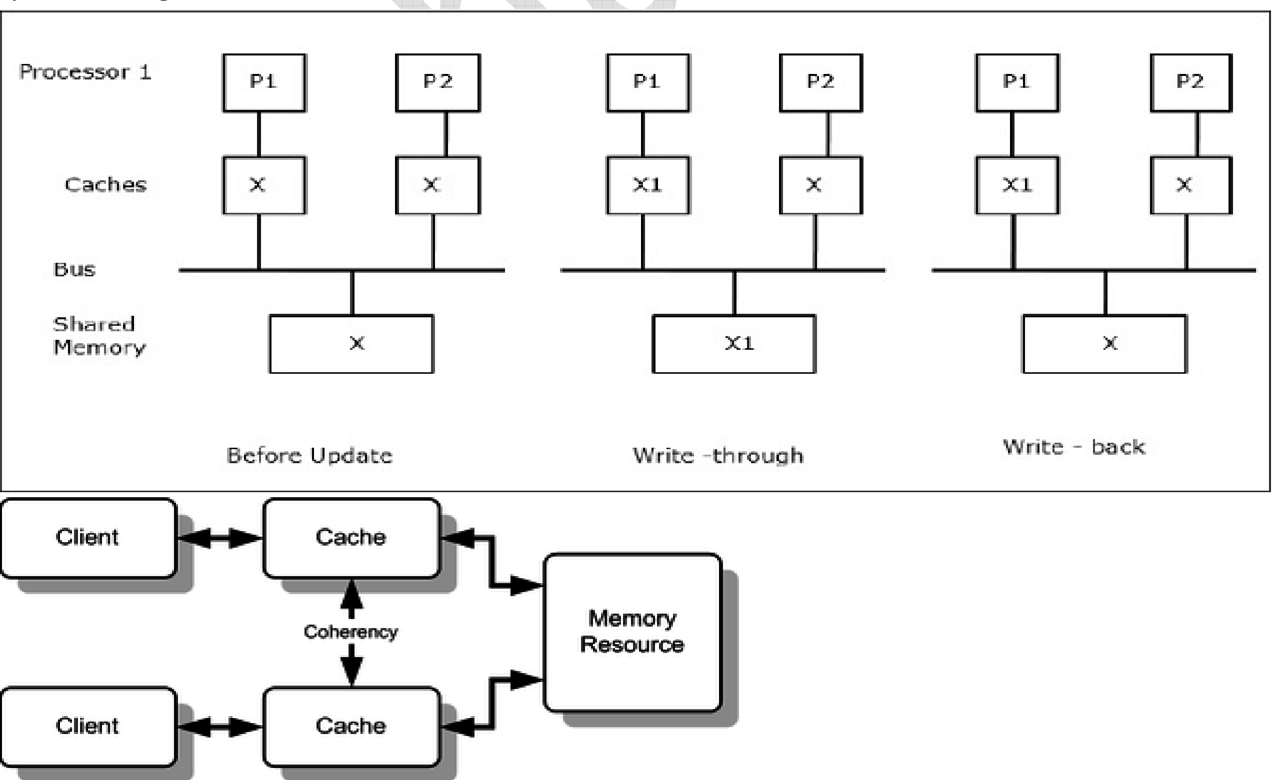
Locality of reference –
Since size of cache memory is less as compared to main memory. So to check which part of main memory should be given priority and loaded in cache is decided based on locality of reference.

Types of Locality of reference

1. **Spatial Locality of reference**
   This says that there is a chance that element will be present in the close proximity to the reference point and next time if again searched then more close proximity to the point of reference.
2. **Temporal Locality of reference**
   In this Least recently used algorithm will be used. Whenever there is page fault occurs within a word will not only load word in main memory but complete page fault will be loaded because spatial locality of reference rule says that if you are referring any word next word will be referred in its register that's why we load complete page table so the complete block will be loaded.

Cache Coherence

In a multiprocessor system, data inconsistency may occur among adjacent levels or within the same level of the memory hierarchy. For example, the cache and the main memory may have inconsistent copies of the same object. As multiple processors operate in parallel, and independently multiple caches may possess different copies of the same memory block, this creates **cache coherence problem**. **Cache coherence schemes** help to avoid this problem by maintaining a uniform state for each cached block of data.



An illustration showing multiple caches of some memory, which acts as a shared resource

Let X be an element of shared data which has been referenced by two processors, P1 and P2. In the beginning, three copies of X are consistent. If the processor P1 writes a new data X1 into the cache, by using **write-through policy**, the same copy will be written immediately into the shared memory. In this case, inconsistency occurs between cache memory and the main memory. When a **write-back policy** is used, the main memory will be updated when the modified data in the cache is replaced or invalidated.
In general, there are three sources of inconsistency problem −

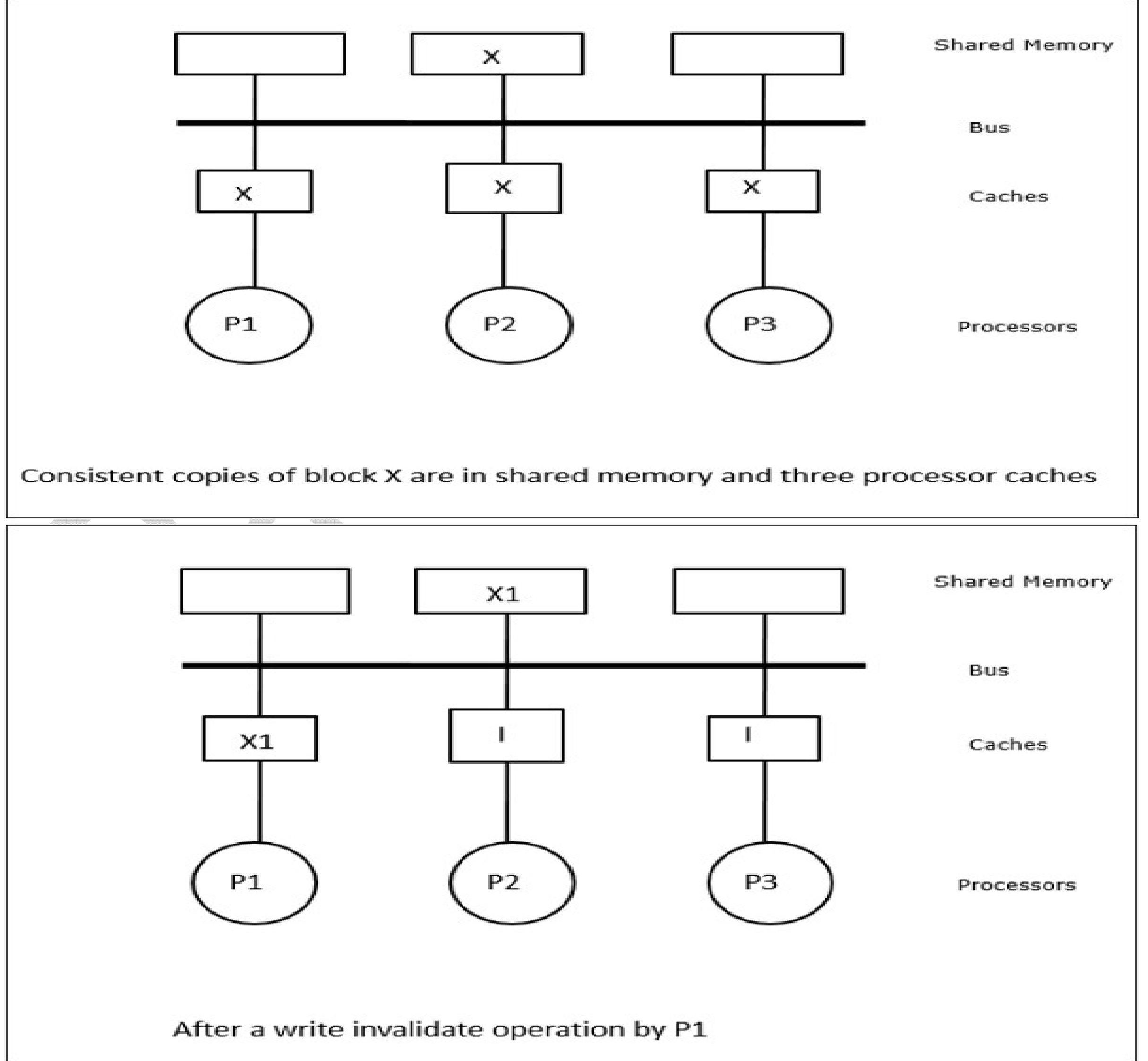
- Sharing of writable data
- Process migration
- I/O activity

Coherence mechanisms

The two most common mechanisms of ensuring coherency are ***snooping*** and ***directory-based***, each having their own benefits and drawbacks. Snooping based protocols tend to be faster, if enough bandwidth is available, since all transactions are a request/response seen by all processors. The drawback is that snooping isn't scalable. Every request must be broadcast to all nodes in a system, meaning that as the system gets larger, the size of the (logical or physical) bus and the bandwidth it provides must grow. Directories, on the other hand, tend to have longer latencies (with a 3 hop request/forward/respond) but use much less bandwidth since messages are point to point and not broadcast. For this reason, many of the larger systems (>64 processors) use this type of cache coherence.

Snoopy Bus Protocols

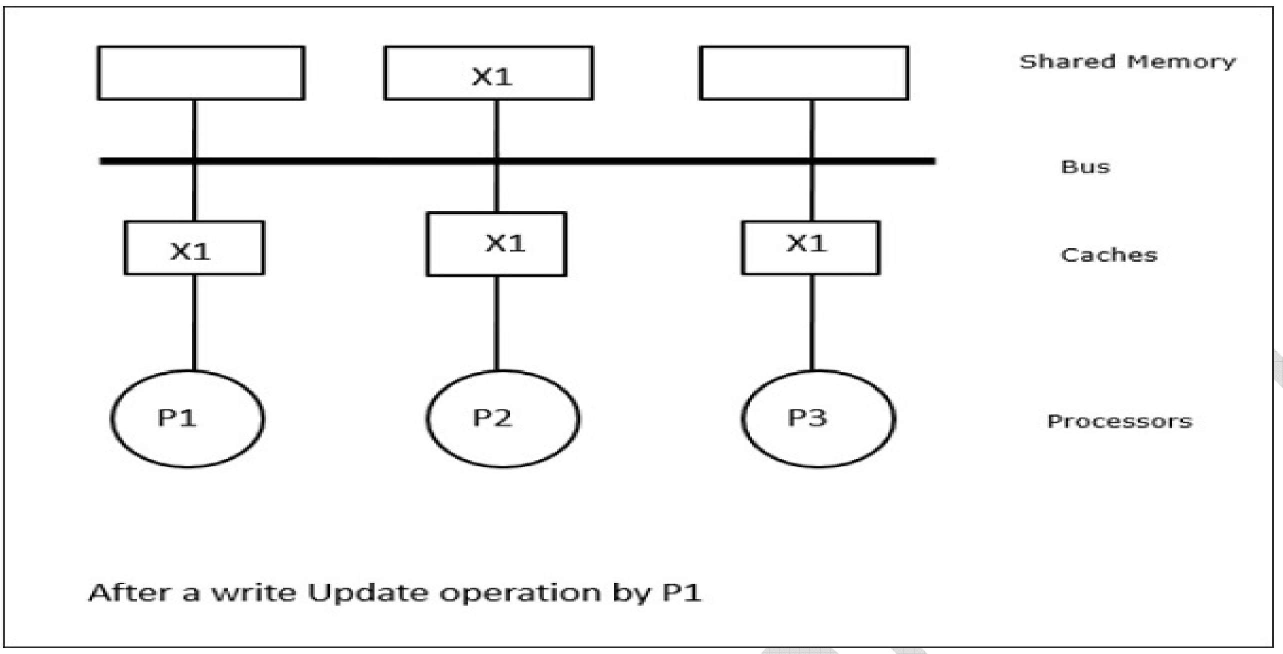
Snoopy protocols achieve data consistency between the cache memory and the shared memory through a bus-based memory system. **Write-invalidate** and **write-update** policies are used for maintaining cache consistency.



Consistent copies of block X are in shared memory and three processor caches

After a write invalidate operation by P1

In this case, we have three processors P1, P2, and P3 having a consistent copy of data element 'X' in their local cache memory and in the shared memory (Figure-a). Processor P1 writes X1 in its cache memory using **write-invalidate protocol**. So, all other copies are invalidated via the bus. It is denoted by 'I' (Figure-b). Invalidated blocks are also known as **dirty**, i.e. they should not be used. The **write-update protocol** updates all the cache copies via the bus. By using **write back cache**, the memory copy is also updated (Figure-c).



Cache Events and Actions

Following events and actions occur on the execution of memory-access and invalidation commands −

- **Read-miss** − When a processor wants to read a block and it is not in the cache, a read-miss occurs. This initiates a **bus-read** operation. If no dirty copy exists, then the main memory that has a consistent copy, supplies a copy to the requesting cache memory. If a dirty copy exists in a remote cache memory, that cache will restrain the main memory and send a copy to the requesting cache memory. In both the cases, the cache copy will enter the valid state after a read miss.
- **Write-hit** − If the copy is in dirty or **reserved** state, write is done locally and the new state is dirty. If the new state is valid, write-invalidate command is broadcasted to all the caches, invalidating their copies. When the shared memory is written through, the resulting state is reserved after this first write.
- **Write-miss** − If a processor fails to write in the local cache memory, the copy must come either from the main memory or from a remote cache memory with a dirty block. This is done by sending a **read-invalidate** command, which will invalidate all cache copies. Then the local copy is updated with dirty state.
- **Read-hit** − Read-hit is always performed in local cache memory without causing a transition of state or using the snoopy bus for invalidation.
- **Block replacement** − When a copy is dirty, it is to be written back to the main memory by block replacement method. However, when the copy is either in valid or reserved or invalid state, no replacement will take place.

Directory-Based Protocols

By using a multistage network for building a large multiprocessor with hundreds of processors, the snoopy cache protocols need to be modified to suit the network capabilities. Broadcasting being very expensive to perform in a multistage network, the consistency commands is sent only to those caches that keep a copy of the block. This is the reason for development of directory-based protocols for network-connected multiprocessors.

In a directory-based protocols system, data to be shared are placed in a common directory that maintains the coherence among the caches. Here, the directory acts as a filter where the processors ask permission to load an entry from the primary memory to its cache memory. If an entry is changed the directory either updates it or invalidates the other caches with that entry.

Cache Coherency in Shared Memory Machines

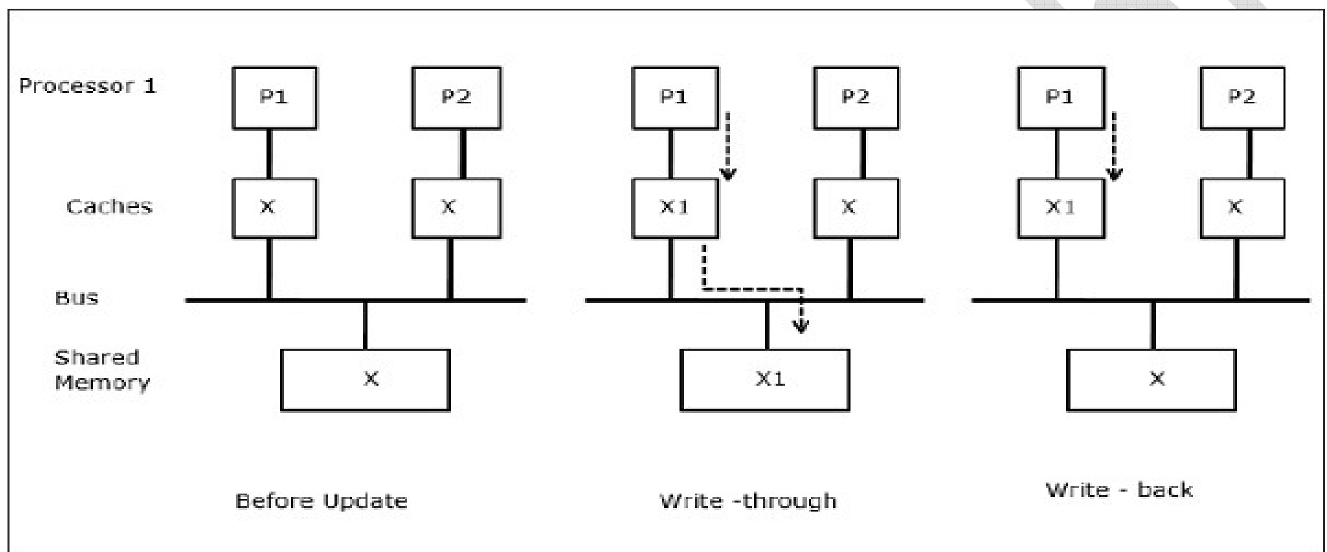
Maintaining cache coherency is a problem in multiprocessor system when the processors contain local cache memory. Data inconsistency between different caches easily occurs in this system.

The major concern areas are −

- Sharing of writable data
- Process migration
- I/O activity

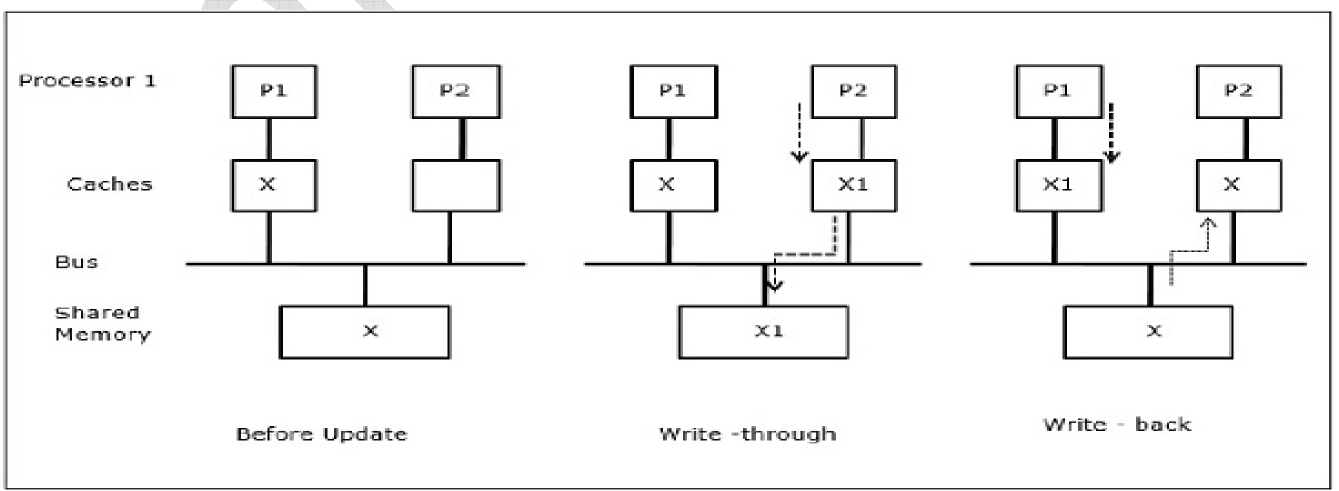
Sharing of writable data

When two processors (P1 and P2) have same data element (X) in their local caches and one process (P1) writes to the data element (X), as the caches are write-through local cache of P1, the main memory is also updated. Now when P2 tries to read data element (X), it does not find X because the data element in the cache of P2 has become outdated.



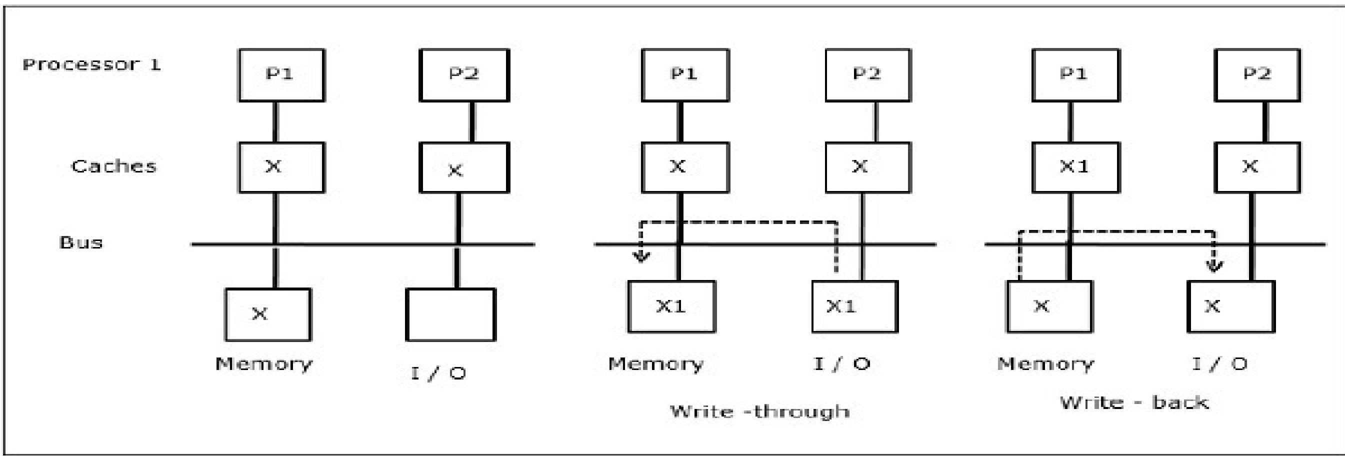
Process migration

In the first stage, cache of P1 has data element X, whereas P2 does not have anything. A process on P2 first writes on X and then migrates to P1. Now, the process starts reading data element X, but as the processor P1 has outdated data the process cannot read it. So, a process on P1 writes to the data element X and then migrates to P2. After migration, a process on P2 starts reading the data element X but it finds an outdated version of X in the main memory.



I/O activity

As illustrated in the figure, an I/O device is added to the bus in a two-processor multiprocessor architecture. In the beginning, both the caches contain the data element X. When the I/O device receives a new element X, it stores the new element directly in the main memory. Now, when either P1 or P2 (assume P1) tries to read element X it gets an outdated copy. So, P1 writes to element X. Now, if I/O device tries to transmit X it gets an outdated copy.



Cache Coherence Protocols in Multiprocessor System

In multiprocessor system where many processes needs a copy of same memory block, the maintenance of consistency among these copies raises a raises a problem referred to as **Cache Coherence Problem.**

This occurs mainly due to these causes:-

- Sharing of writable data.
- Process migration.
- Inconsistency due to I/O.

Cache Coherence Protocols:

These are explained as following below:

1. MSI Protocol:

This is a basic cache coherence protocol used in multiprocessor system. The letters of protocol name identify possible states in which a cache can be. So, for MSI each block can have one of the following possible states:

- **Modified –**
  The block has been modified n cache, i.e., the data in the cache is inconsistent with the backing store (memory). So, a cache with a block in “M” state has responsibility to write the block to backing store when it is evicted.
- **Shared –**
  This block is not modified and is present in atleast one cache. The cache can evict the data without writing it to backing store.
- **Invalid –**
  This block is invalid and must be fetched from memory or from another cache if it is to be stored in this cache.

2. MOSI Protocol:

This protocol is an extension of MSI protocol. It adds the following state in MSI protocol:

- **Owned –**
  It indicates that the present processor owns this block and will service requests from other processors for the block.

Partha Sarathi

3. MESI Protocol –

It is the most widely used cache coherence protocol. Every cache line is marked with one the following states:

- **Modified –**
  This indicates that the cache line is present in current cache only and is dirty i.e its value is different from the main memory. The cache is required to write the data back to main memory in future, before permitting any other read of invalid main memory state.
- **Exclusive –**
  This indicates that the cache line is present in current cache only and is clean i.e its value matches the main memory value.
- **Shared –**
  It indicates that this cache line may be stored in other caches of the machine.
- **Invalid –**
  It indicates that this cache line is invalid.

4. MOESI Protocol:

This is a full cache coherence protocol that encompasses all of the possible states commonly used in other protocols. Each cache line is in one of the following states:

- **Modified –**
  A cache line in this state holds the most recent, correct copy of the data while the copy in the main memory is incorrect and no other processor holds a copy.
- **Owned –**
  A cache line in this state holds the most recent, correct copy of the data. It is similar to shared state in that other processors can hold a copy of most recent, correct data and unlike shared state however, copy in main memory can be incorrect. Only one processor can hold the data in owned state while all other processors must hold the data in shared state.
- **Exclusive –**
  A cache line in this state holds the most recent, correct copy of the data. The main memory copy is also most recent, correct copy of data while no other holds a copy of data.
- **Shared –**
  A cache line in this state holds the most recent, correct copy of the data. Other processors in system may hold copies of data in shared state as well. The main memory copy is also the most recent, correct copy of the data, if no other processor holds it in owned state.
- **Invalid –**
  A cache line in this state does not hold a valid copy of data. Valid copies of data can be either in main memory or another processor cache.

Difference between CPU Cache and TLB

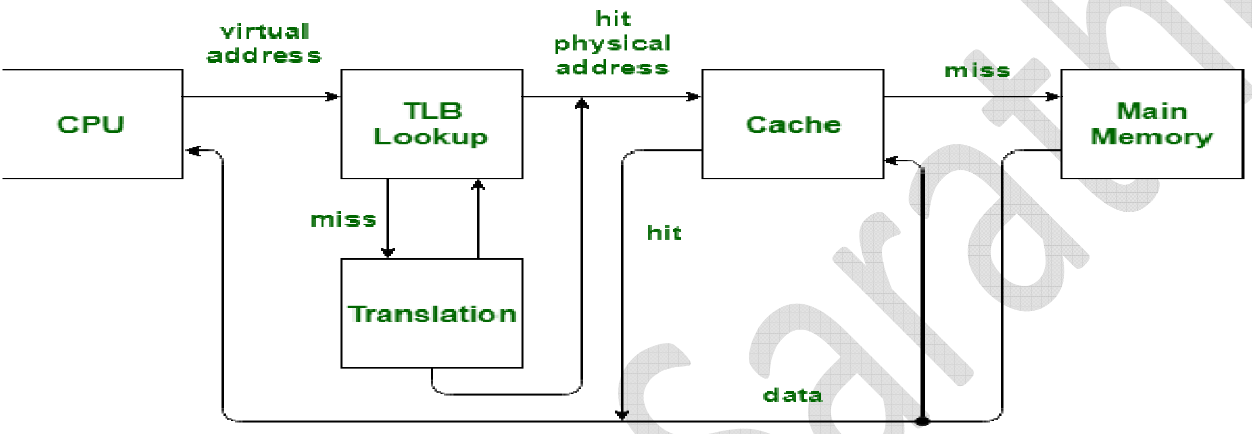
Both CPU Cache and TLB are hardware used in microprocessors but what's the difference, especially when someone says that TLB is also a type of Cache?

First thing first. **CPU Cache** is a fast memory which is used to improve latency of fetching information from Main memory (RAM) to CPU registers. So CPU Cache sits between Main memory and CPU. And this cache stores information temporarily so that the next access to the same information is faster. A CPU cache which used to store executable instructions, it's called Instruction Cache (I-Cache). A CPU cache which is used to store data, it's called Data Cache (D-Cache). So I-Cache and D-Cache speeds up fetching time for instructions and data respectively. A modern processor contains both I-Cache and D-Cache. For completeness, let us discuss about D-cache hierarchy as well. D-Cache is typically organized in a hierarchy i.e. Level 1 data cache, Level 2 data cache etc.. It should be noted that L1 D-Cache is faster/smaller/costlier as compared to L2 D-Cache. But the basic idea of '*CPU cache*' is to speed up instruction/data fetch time from Main memory to CPU.

Partha Sarathi

Translation Lookaside Buffer (i.e. TLB) is required only if Virtual Memory is used by a processor. In short, TLB speeds up translation of virtual address to physical address by storing page-table in a faster memory. In fact, TLB also sits between CPU and Main memory. Precisely speaking, TLB is used by MMU when physical address needs to be translated to virtual address. By keeping this mapping of virtual-physical addresses in a fast memory, access to page-table improves. It should be noted that page-table (which itself is stored in RAM) keeps track of where virtual pages are stored in the physical memory. In that sense, TLB also can be considered as a cache of the page-table.

But the scope of operation for *TLB* and *CPU Cach*e is different. TLB is about 'speeding up address translation for Virtual memory' so that page-table needn't to be accessed for every address. CPU Cache is about 'speeding up main memory access latency' so that RAM isn't accessed always by CPU. TLB operation comes at the time of address translation by MMU while CPU cache operation comes at the time of memory access by CPU. In fact, any modern processor deploys all I-Cache, L1 & L2 D-Cache and TLB.



Cache Performance:

When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache.

- If the processor finds that the memory location is in the cache, a **cache hit** has occurred and data is read from cache
- If the processor **does not** find the memory location in the cache, a **cache miss** has occurred. For a cache miss, the cache allocates a new entry and copies in data from main memory, then the request is fulfilled from the contents of the cache.
- *Average memory access time is a useful measure to evaluate the performance of a memory-hierarchy configuration.*

  Average memory access time (AMAT): AMAT = Hit time + (Miss Rate X Miss Penalty)

- It tells us how much penalty the memory system imposes on each access (on average).
- It can easily be converted into clock cycles for a particular CPU.
- But leaving the penalty in nanoseconds allows two systems with different clock cycles times to be compared to a single memory system.
- Example:
  - 1 cycle hit cost
  - 10 cycle miss penalty (11 cycles total for a miss)
  - Program has 10% miss rate
  - Average memory access time = 1.0 + 10% * 10 = 2.0

There may be different penalties for Instruction and Data accesses. In this case, you may have to compute them separately. This requires knowledge of the fraction of references that are instructions and the fraction that are data. The text gives 75% instruction references to 25% data references. We can also compute the write penalty separately from the read penalty. This may be necessary for two reasons:

Partha Sarathi

- Miss rates are different for each situation.
- Miss penalties are different for each situation.

- Treating them as a single quantity yields a useful CPU time formula:
  CPUtime = IC x (CPIexec + Mem. accesses per instruction x Miss rate x Miss penalty) x Tcycle

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory access}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}\right) \times \text{Clock Cycle Time}$$

Measuring miss rate as misses per instruction Misses per instruction = Memory accesses per instruction x Miss rate=>

CPUtime= IC x(CPIexec+ Misses per instruction x Miss penalty) x Tcycle

An Example

Compare the performance of a 64KB unified cache with a split cache with 32KB data and 16KB instruction .
The miss penalty for either cache is 100 ns, and the CPU clock runs at 200 MHz. Don't forget that the cache requires an extra cycle for load and store hits on a unified cache because of the structural conflict.
Calculate the effect on CPI rather than the average memory access time. Assume miss rates are as follows

- 64K Unified cache: 1.35%
- 16K instruction cache: 0.64%
- 32K data cache: 4.82%

Assume a data access occurs once for every 3 instructions, on average.

- The solution is to figure out the penalty to CPI separately for instructions and data.

- First, we figure out the miss penalty in terms of clock cycles: 100 ns/5 ns = 20 cycles.
    - For the unified cache, the per-instruction penalty is (0 + 1.35% x 20) = 0.27 cycles.
    - For data accesses, which occur on about 1/3 of all instructions, the penalty is (1 + 1.35% x 20) = 1.27 cycles per access, or 0.42 cycles per instruction.
    - The total penalty is 0.69 CPI .

- In the split cache, the per-instruction penalty is (0 + 0.64% x 20) = 0.13 CPI.
    - For data accesses, it is (0 + 4.82% x 20) x (1/3) = 0.32 CPI.
    - The total penalty is 0.45 CPI .

- In this case, the split cache performs better because of the lack of a stall on data accesses.

Effects of Cache Performance on CPU Performance

- Low CPI machines suffer more relative to some fixed CPI memory penalty.
    - A machine with a CPI of 5 suffers little from a 1 CPI penalty.
    - However, a processor with a CPI of 0.5 has its execution time tripled !

- Cache miss penalties are measured in cycles, not nanoseconds.
    - This means that a faster machine will stall more cycles on the same memory system.

- Amdahl's Law raises its ugly head again:
    - Fast machines with low CPI are affected significantly from memory access penalties.

Improving Cache Performance

- The increasing speed gap between CPU and main memory has made the performance of the memory system increasingly important.
- 15 distinct organizations characterize the effort of system architects in reducing average memory access time.
- These organizations can be distinguished by:
    - Reducing the miss rate.
    - Reducing the miss penalty.
    - Reducing the time to hit in a cache.

Partha Sarathi

Reducing Cache Misses

- Components of miss rate: All of these factors may be reduced using various methods we'll talk about.
- **Compulsory**
    - **Cold start misses** or **first reference misses** : The first access to a block can NOT be in the cache, so there must be a compulsory miss.
    - These are suffered regardless of cache size.
- **Capacity**
    - If the cache is too small to hold all of the blocks needed during execution of a program, misses occur on blocks that were discarded earlier.
    - In other words, this is the difference between the compulsory miss rate and the miss rate of a finite size fully associative cache.
- **Conflict**
    - If the cache has sufficient space for the data, but the block can NOT be kept because the set is full, a conflict miss will occur.
    - This is the difference between the miss rate of a non-fully associative cache and a fully-associative cache.
    - These misses are also called **collision** or **interference** misses.

Reducing Cache Miss Rate

- To reduce cache miss rate, we have to eliminate some of the misses due to the three C's.

- We cannot reduce capacity misses much except by making the cache larger.

- We can, however, reduce the conflict misses and compulsory misses in several ways:

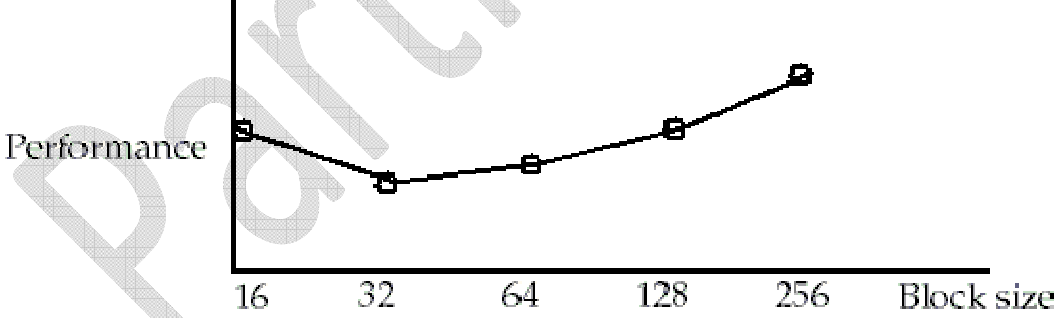
- Larger cache blocks
    - Larger blocks decrease the compulsory miss rate by taking advantage of spatial locality.

    - However, they may increase the miss penalty by requiring more data to be fetched per miss.

    - In addition, they will almost certainly increase conflict misses since fewer blocks can be stored in the cache.
        - And maybe even capacity misses in small caches.

Reducing Cache Miss Rate

- **Larger cache blocks**



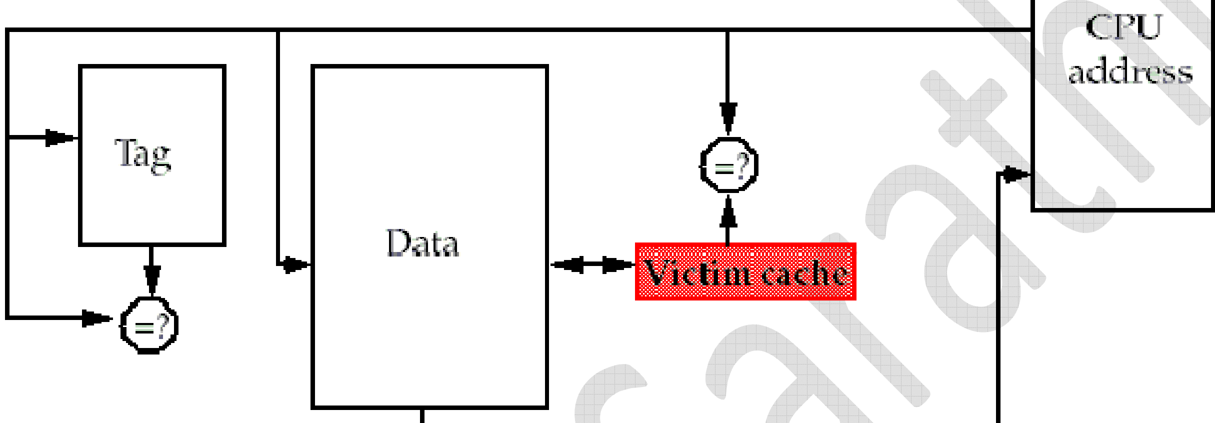
-
    - The performance curve is U-shaped because:
        - Small blocks have a higher miss rate and
        - Large blocks have a higher miss penalty (even if miss rate is not too high).

    - High latency , high bandwidth memory systems encourage large block sizes since the cache gets more bytes per miss for a small increase in miss penalty.
        - 32-byte blocks are typical for 1-KB, 4-KB and 16-KB caches while 64-byte blocks are typical for larger caches.

Partha Sarathi

- **Higher associativity**
  - Conflict misses can be a problem for caches with low associativity (especially direct-mapped).
  - 2:1 cache rule of thumb : a direct-mapped cache of size N has the same miss rate as a 2-way set-associative cache of size N/2.
  - However, there is a limit -- higher associativity means more hardware and usually longer cycle times (increased hit time).
    - In addition, it may cause more capacity misses.
  - Nobody uses more than 8-way set-associative caches today, and most systems use 4-way or less.
  - The problem is that the higher hit rate is offset by the slower clock cycle time.

- **Victim caches**
  - A victim cache is a small (usually, but not necessarily) fully-associative cache that holds a few of the most recently replaced blocks or victims from the main cache.



  - Can improve miss rates without affecting the processor clock rate.
  - This cache is checked on a miss before going to main memory.
    - If found, the victim block and the cache block are swapped.
  - It can reduce capacity misses but is best at reducing conflict misses.
  - It's particularly effective for small, direct-mapped data caches.
    - A 4 entry victim cache handled from 20% to 95% of the conflict misses from a 4KB direct-mapped data cache.

- **Pseudo-associative caches**
  - These caches use a technique similar to double hashing.

  - On a miss, the cache searches a different set for the desired block.
    - The second (pseudo) set to probe is usually found by inverting one or more bits in the original set **index** .

  - Note that two separate searches are conducted on a miss.
    - The first search proceeds as it would for direct-mapped cache.
    - Since there is no associative hardware, hit time is fast if it is found the first time.
  - While this second probe takes some time (usually an extra cycle or two), it is a lot faster than going to main memory.
    - The secondary block can be swapped with the primary block on a "slow hit".
  - This method reduces the effect of conflict misses.
  - Also improves miss rates without affecting the processor clock rate.

- **Hardware prefetch**
  - Prefetching is the act of getting data from memory before it is actually needed by the CPU.

  - Typically, the cache requests the next consecutive block to be fetched with a requested block.
    - It is hoped that this avoids a subsequent miss.
  - This reduces compulsory misses by retrieving the data before it is requested.

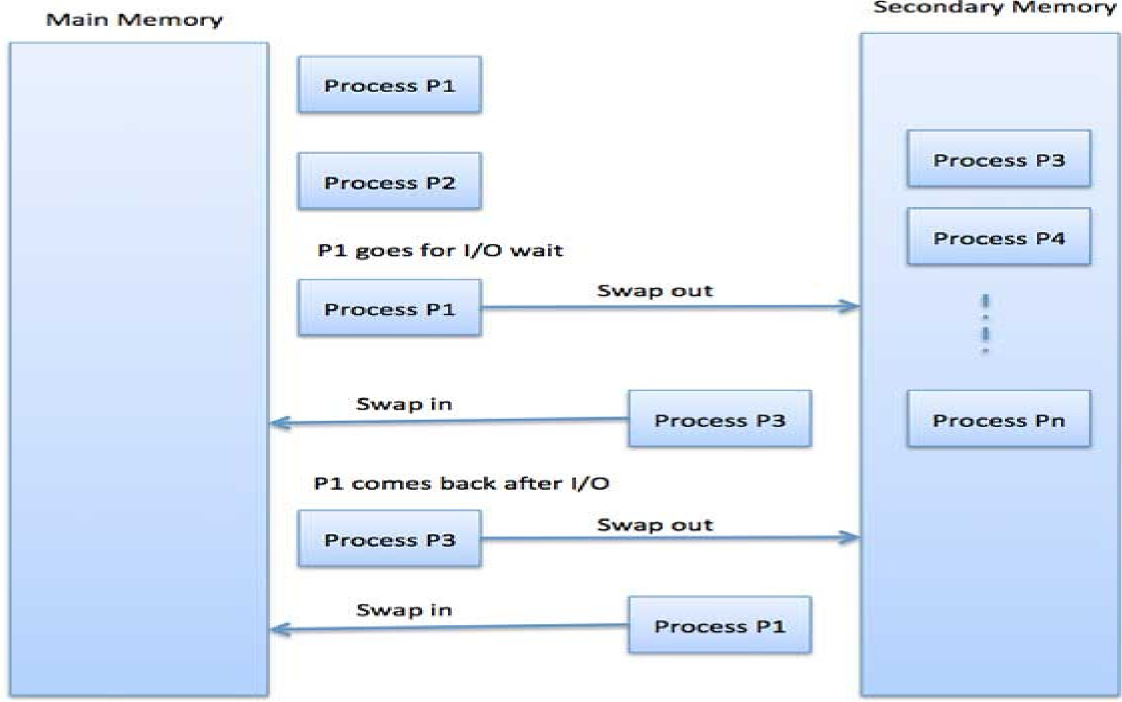
Partha Sarathi

- Of course, this may increase other misses by removing useful blocks from the cache.
    - Thus, many caches hold prefetched blocks in a **special buffer** until they are actually needed.
    - This buffer is faster than main memory but only has a limited capacity.
- Prefetching also uses main memory bandwidth.
    - It works well if the data is actually used.
    - However, it can adversely affect performance if the data is rarely used and the accesses interfere with `demand misses'.

Virtual Memory

Swapping

Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.
Though performance is usually affected by swapping process but it helps in running multiple and big processes in parallel and that's the reason Swapping is also known as a technique for memory compaction.



The total time taken by swapping process includes the time it takes to move the entire process to a secondary disk and then to copy the process back to memory, as well as the time the process takes to regain main memory.
Let us assume that the user process is of size 2048KB and on a standard hard disk where swapping will take place has a data transfer rate around 1 MB per second. The actual transfer of the 1000K process to or from memory will take

2048KB / 1024KB per second
= 2 seconds
= 2000 milliseconds
Now considering in and out time, it will take complete 4000 milliseconds plus other overhead where the process competes to regain main memory.

Memory Allocation
Main memory usually has two partitions −

- Low Memory − Operating system resides in this memory.
- High Memory − User processes are held in high memory.

Operating system uses the following memory allocation mechanism.

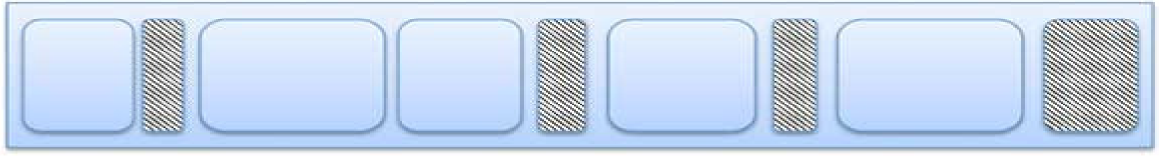
S.N.	Memory Allocation & Description
1	Single-partition allocation In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register.
2	Multiple-partition allocation In this type of allocation, main memory is divided into a number of fixed-sized partitions where each partition should contain only one process. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

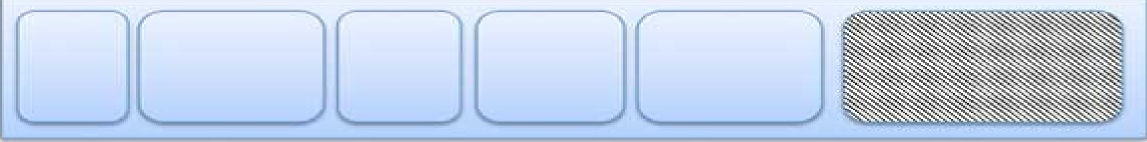
Fragmentation
As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types −

S.N.	Fragmentation & Description
1	External fragmentation Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.
2	Internal fragmentation Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory −

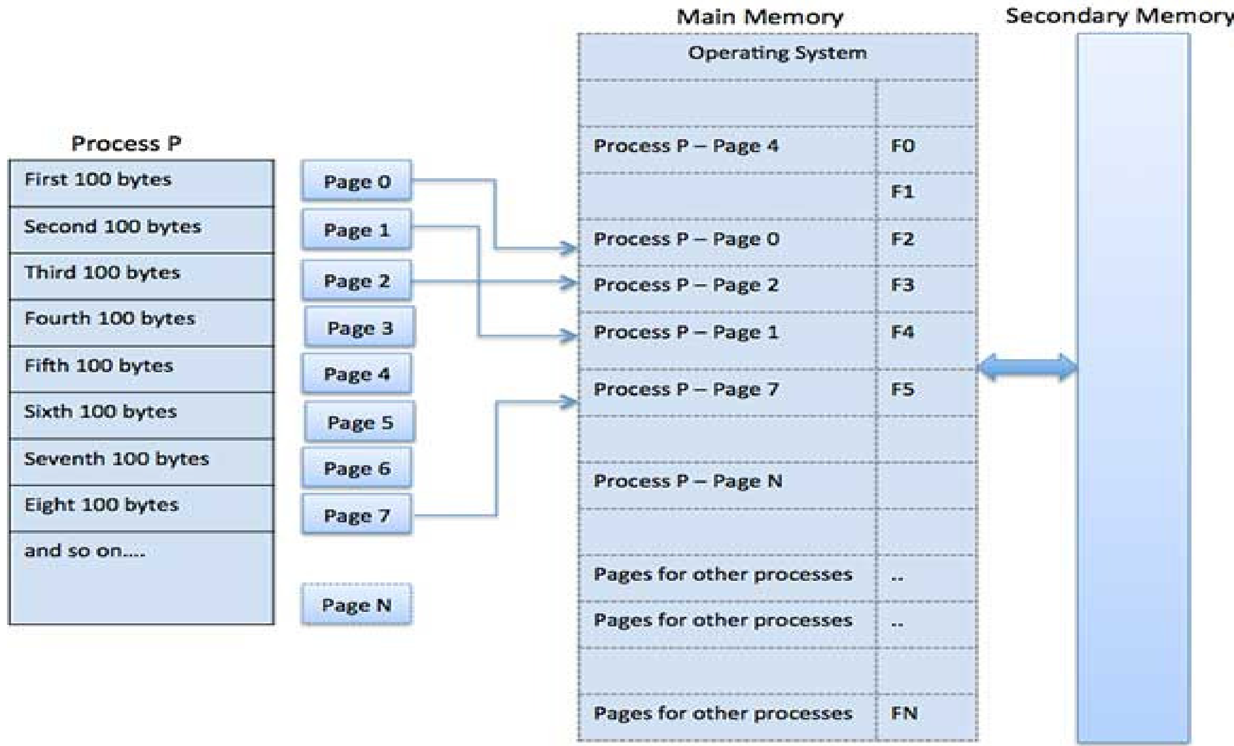




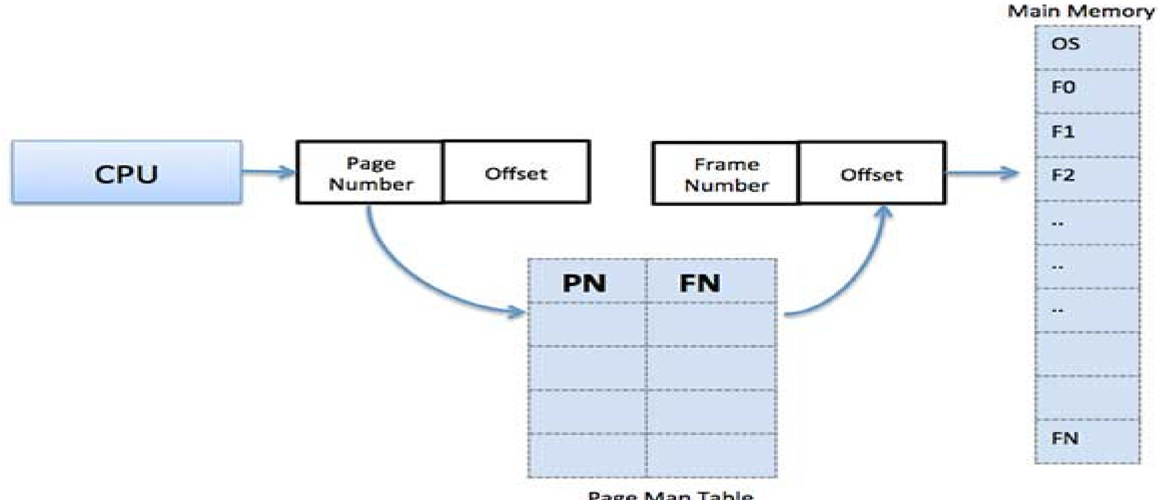
External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.
The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.

Paging
A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.

Paging is a memory management technique in which process address space is broken into blocks of the same size called pages (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.
Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called frames and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.



Address Translation
Page address is called **logical address** and represented by **page number** and the **offset**.
Logical Address = Page number + page offset
Frame address is called **physical address** and represented by a **frame number** and the **offset**.
Physical Address = Frame number + page offset
A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.

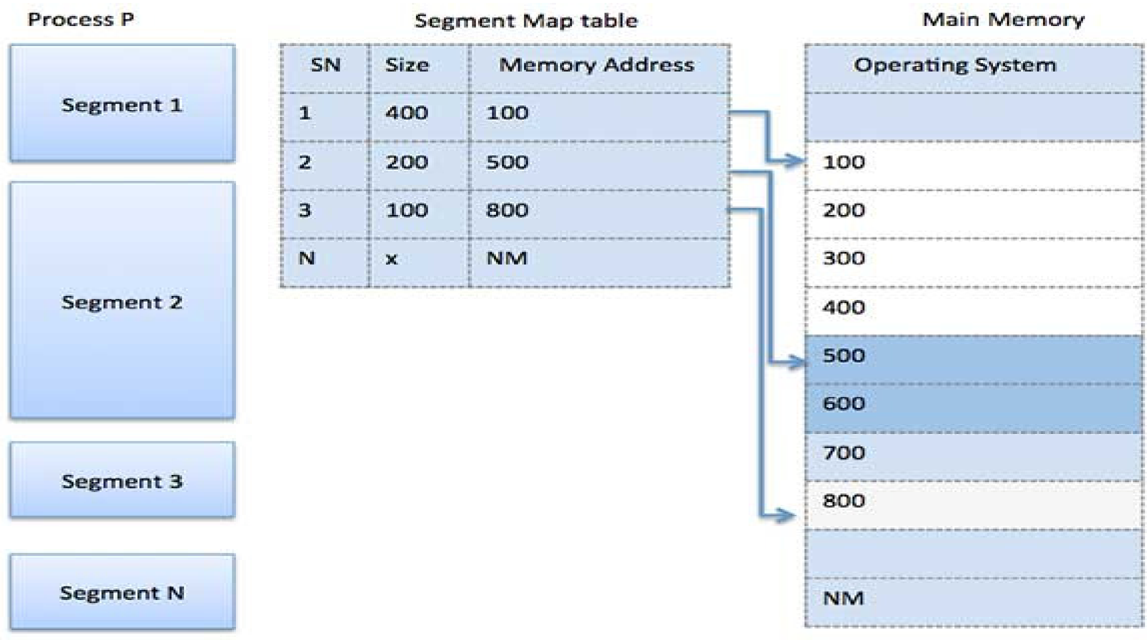


When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.
When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or

unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.
This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.
Advantages and Disadvantages of Paging
Here is a list of advantages and disadvantages of paging –

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.
- Page table requires extra memory space, so may not be good for a system having small RAM.

Segmentation
Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.
When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.
Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.
A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a **segment map table** for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.



Virtual Memory
Virtual Memory is a storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program generated addresses are translated automatically to the corresponding machine addresses.
The size of virtual storage is limited by the addressing scheme of the computer system and amount of secondary memory is available not by the actual number of the main storage locations.

It is a technique that is implemented using both hardware and software. It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory.

1. All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process can be swapped in and out of main memory such that it occupies different places in main memory at different times during the course of execution.

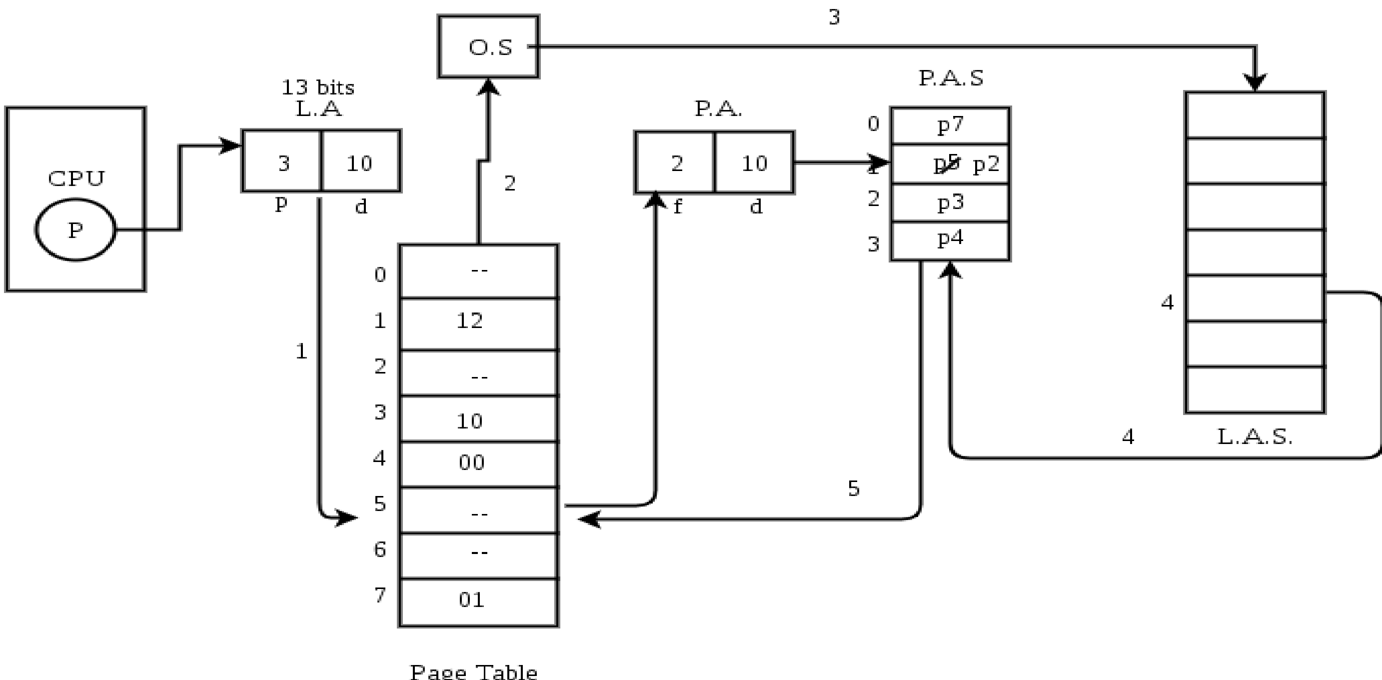
2. A process may be broken into number of pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time address translation and use of page or segment table permits this.

If these characteristics are present then, it is not necessary that all the pages or segments are present in the main memory during execution. This means that the required pages need to be loaded into memory whenever required. Virtual memory is implemented using Demand Paging or Demand Segmentation.

Demand Paging

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.

The process of loading the page into memory on demand (whenever page fault occurs) is known as demand paging. The process includes the following steps :



1. If CPU try to refer a page that is currently not available in the main memory, it generates an interrupt indicating memory access fault.
2. The OS puts the interrupted process in a blocking state. For the execution to proceed the OS must bring the required page into the memory.
3. The OS will search for the required page in the logical address space.
4. The required page will be brought from logical address space to physical address space. The page replacement algorithms are used for the decision making of replacing the page in physical address space.
5. The page table will updated accordingly.
6. The signal will be sent to the CPU to continue the program execution and it will place the process back into ready state.

Hence whenever a page fault occurs these steps are followed by the operating system and the required page is brought into memory.

Advantages :

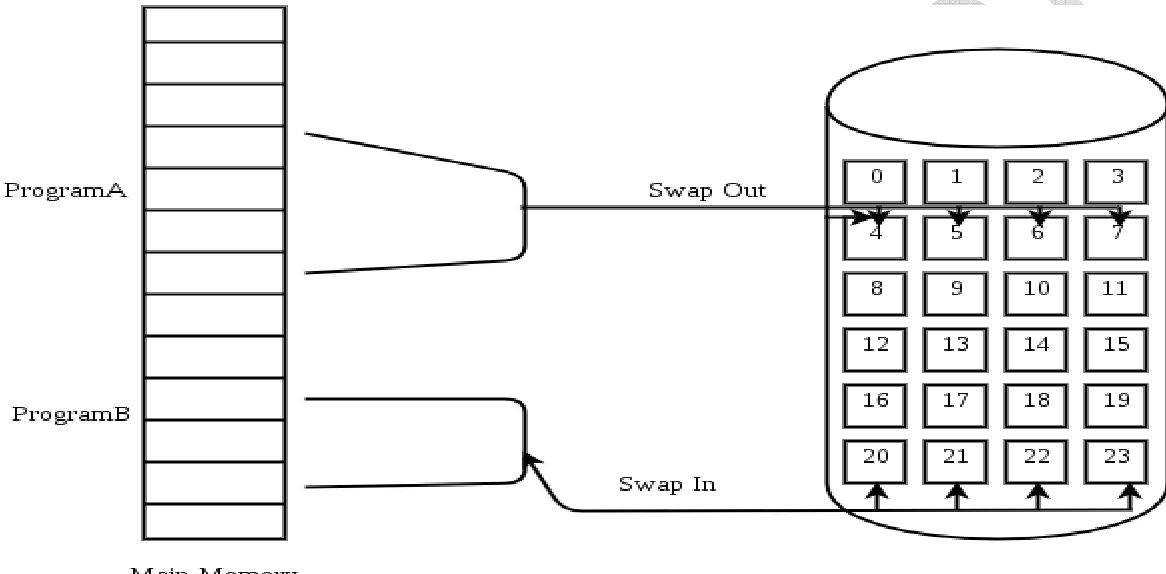
- More processes may be maintained in the main memory: Because we are going to load only some of the pages of any particular process, there is room for more processes. This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in the ready state at any particular time.

- A process may be larger than all of main memory: One of the most fundamental restrictions in programming is lifted. A process larger than the main memory can be executed because of demand paging. The OS itself loads pages of a process in main memory as required.
- It allows greater multiprogramming levels by using less of the available (primary) memory for each process.

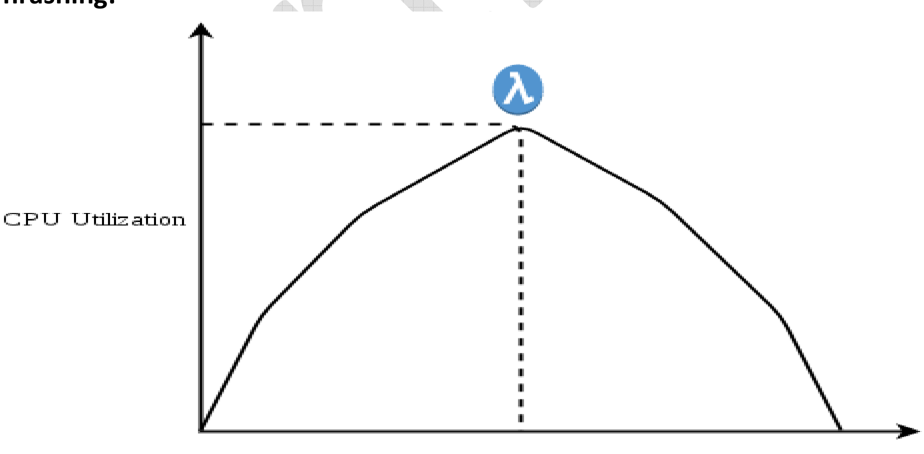
Page Fault Service Time :
The time taken to service the page fault is called as page fault service time. The page fault service time includes the time taken to perform all the above six steps.
Let Main memory access time is: m
Page fault service time is: s
Page fault rate is : p
Then, Effective memory access time = (p*s) + (1-p)*m

Swapping:

Swapping a process out means removing all of its pages from memory, or marking them so that they will be removed by the normal page replacement process. Suspending a process ensures that it is not runnable while it is swapped out. At some later time, the system swaps back the process from the secondary storage to main memory. When a process is busy swapping pages in and out then this situation is called thrashing.



Thrashing:



At any given time, only few pages of any process are in main memory and therefore more processes can be maintained in memory. Furthermore time is saved because unused pages are not swapped in and out of memory. However, the OS must be clever about how it manages this scheme. In the steady state practically, all of main memory will be occupied with process's pages, so that the processor and OS has direct access to as many processes as possible. Thus when the OS brings one page in, it must throw another out. If it throws out a page just before it is

used, then it will just have to get that page again almost immediately. Too much of this leads to a condition called Thrashing. The system spends most of its time swapping pages rather than executing instructions. So a good page replacement algorithm is required.

In the given diagram, initial degree of multi programming upto some extent of point(lamda), the CPU utilization is very high and the system resources are utilized 100%. But if we further increase the degree of multi programming the CPU utilization will drastically fall down and the system will spent more time only in the page replacement and the time taken to complete the execution of the process will increase. This situation in the system is called as thrashing.

Causes of Thrashing :

1. **High degree of multiprogramming** : If the number of processes keeps on increasing in the memory than number of frames allocated to each process will be decreased. So, less number of frames will be available to each process. Due to this, page fault will occur more frequently and more CPU time will be wasted in just swapping in and out of pages and the utilization will keep on decreasing.

For example:
Let free frames = 400
Case 1: Number of process = 100
Then, each process will get 4 frames.
Case 2: Number of process = 400
Each process will get 1 frame.
Case 2 is a condition of thrashing, as the number of processes are increased,frames per process are decreased. Hence CPU time will be consumed in just swapping pages.

2. **Lacks of Frames**:If a process has less number of frames then less pages of that process will be able to reside in memory and hence more frequent swapping in and out will be required. This may lead to thrashing. Hence sufficient amount of frames must be allocated to each process in order to prevent thrashing.

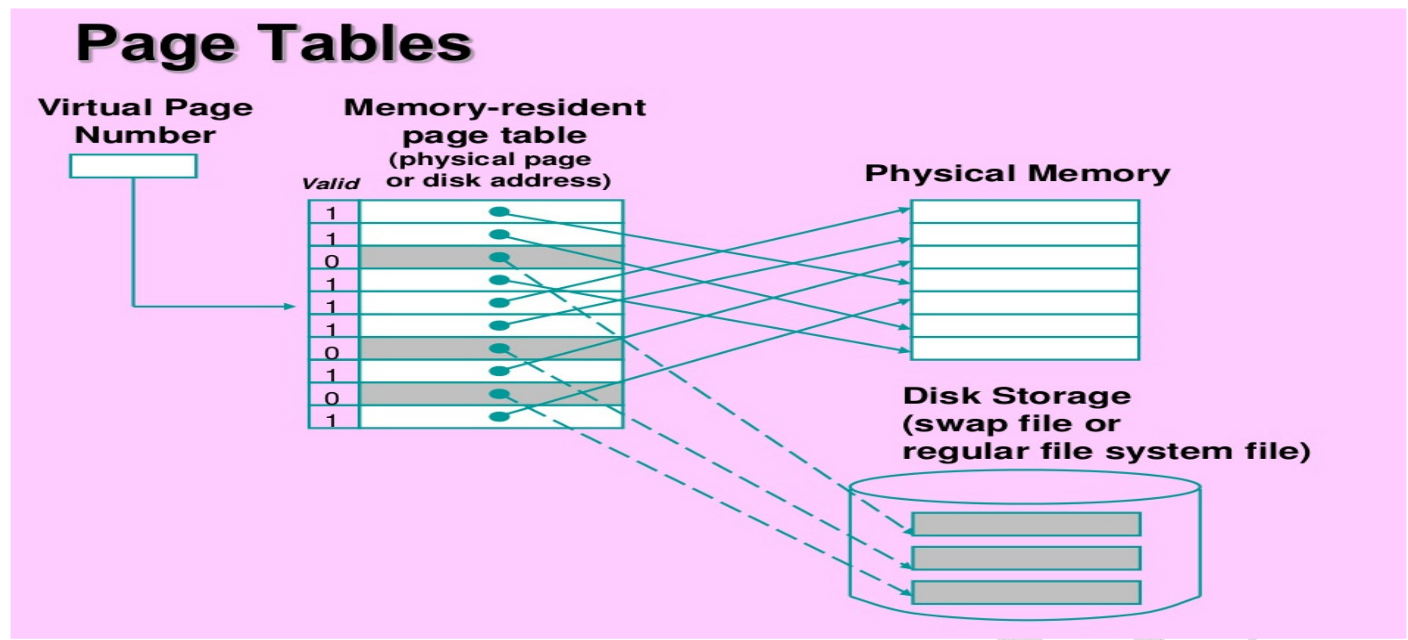
Recovery of Thrashing :

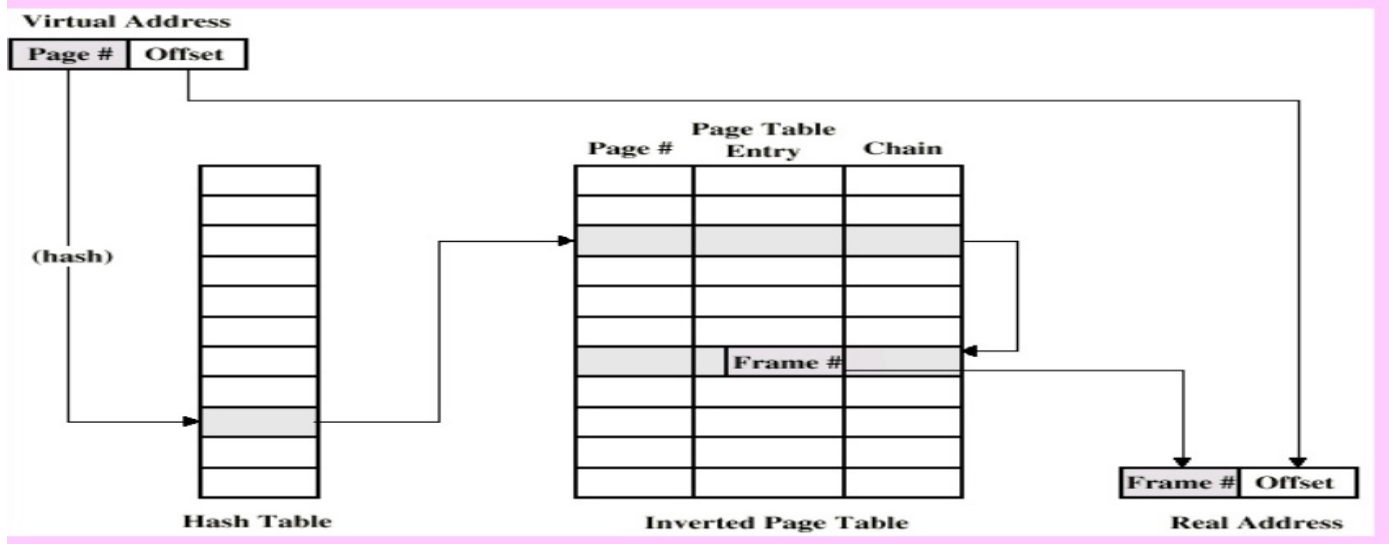
- Do not allow the system to go into thrashing by instructing the long term scheduler not to bring the processes into memory after the threshold.
- If the system is already in thrashing then instruct the mid term schedular to suspend some of the processes so that we can recover the system from thrashing.

Page Table Structure

- Each process has a list of frames that it is using, called a page table, stored in the PCB
- Page Table Structure is a Basic mechanism for reading a word from memory involves using a page table to translate :
  - a virtual address - page number and offset into
  - a physical address - frame number and offset
- Page tables may be very large
  - they cannot be stored in registers
  - they are often stored in virtual memory (so are subject to paging!)
  - sometimes a page directory is used to organize many pages of page tables ,Pentium uses such a two-level structure
  - sometimes an inverted page table structure is used to map a virtual address to a real address using a hash on the page number of the virtual address AS/400 and PowerPC use this idea.
- Page tables are variable in length (depends on process size)
- A single register holds the starting physical address of the page table of the currently running process then must be in main memory instead of registers.

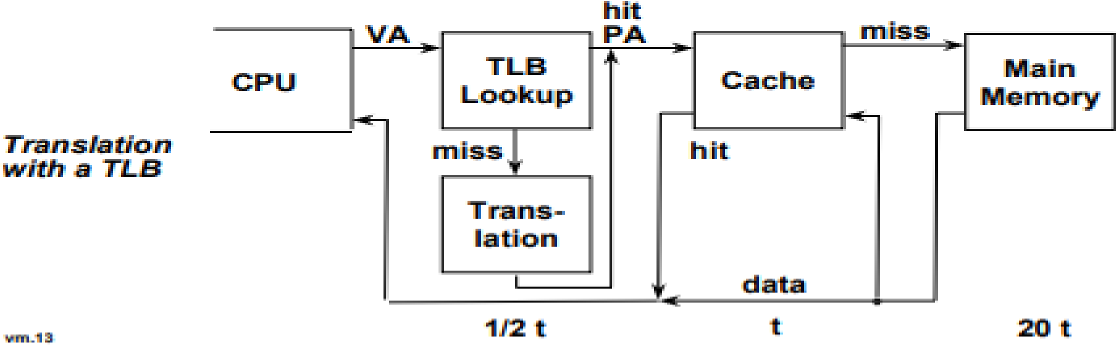
Partha Sarathi

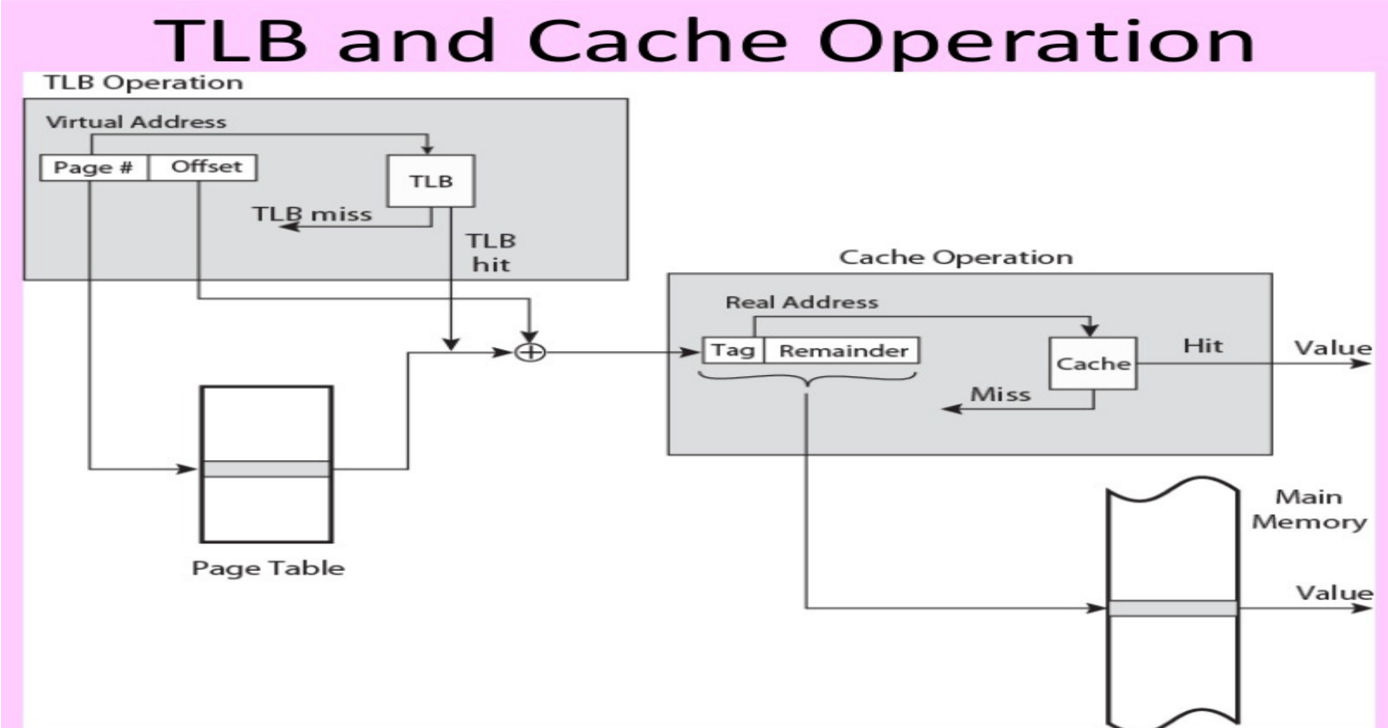




Translation Lookaside Buffer (TLB)

- Each virtual memory reference can causes two physical memory access
  - One to fetch the appropriate page table entry
  - One to fetch the desired data
- To overcome this problem a high-speed cache is set up for page table entries – Called a Translation Lookaside Buffer (TLB)
- TLB is a special cache, just for page table entries
- Contains page table entries that have been most recently used
- Given a virtual address, processor examines the TLB •
- If page table entry is present (TLB hit), the frame number is retrieved and the real address is formed
- If page table entry is not found in the TLB (TLB miss), the page number is used to index the process page table
- First checks if page is already in main memory – If not in main memory a page fault is issued
- The TLB is updated to include the new page entry





Reducing Translation Time

Machines with TLBs go one step further to reduce # cycles/cache access.
They overlap the cache access with the TLB access.
Works because high order bits of the VA are used to look in the TLB while low order bits are used as index into cache.

Advantages Of Virtual Memory

1. You can run more applications at once.
2. Allows you to fit many large programs into a relatively small RAM.
3. You don't have to buy more memory RAM
4. VM supports Swapping.
5. Common data or code may be shared to save memory.
6. Process need not be in memory as a whole , Only part of a program needs to be loaded into memory.
7. Process may even be larger than all of physical memory.
8. Data / code can be read from disk as needed.
9. Code can be placed anywhere in physical memory without relocation.
10. More processes can be maintained in Main Memory which increases effective use of CPU.
11. Don't need to break program into fragments to accommodate memory limitations

Partha

Difference between Virtual memory and Cache memory:

S.NO	VIRTUAL MEMORY	CACHE MEMORY
1.	Virtual memory increases the capacity of main memory.	While cache memory increase the accessing speed of CPU.
2.	Virtual memory is not a memory unit, its a technique.	Cache memory is exactly a memory unit.
3.	The size of virtual memory is greater than the cache memory.	While the size of cache memory is less than the virtual memory.
4.	Operating System manages the Virtual memory.	On the other hand hardware manages the cache memory.
5.	In virtual memory, The program with size larger than the main memory are executed.	While in cache memory, recently used data is copied into.

Page Replacement Algorithm

Page Fault –

A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory. Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page.

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated. Paging happens whenever a page fault occurs and a free page cannot be used for allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.

When the page that was selected for replacement and was paged out, is referenced again, it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm.

A page replacement algorithm looks at the limited information about accessing the pages provided by hardware, and tries to select which pages should be replaced to minimize the total number of page misses, while balancing it with the costs of primary storage and processor time of the algorithm itself. There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults,

Reference String

The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data, where we note two things.

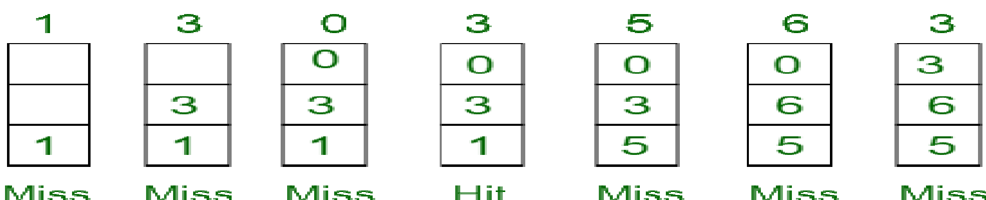
- For a given page size, we need to consider only the page number, not the entire address.
- If we have a reference to a page **p**, then any immediately following references to page **p** will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.
- For example, consider the following sequence of addresses − 123,215,600,1234,76,96
- If page size is 100, then the reference string is 1,2,6,12,0,0

Page Replacement Algorithms :

- **First In First Out (FIFO) –**
  This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.
  Example-1Consider page reference string 1, 3, 0, 3, 5, 6 with 3 page frames. Find number of page faults.

Partha Sarathi

1, 3, 0, 3, 5, 6, 3



Total Page Fault = 6

Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> 3 Page Faults.
when 3 comes, it is already in memory so —> 0 Page Faults.
Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>1 Page Fault.
6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>1 Page Fault.
Finally when 3 come it is not avilable so it replaces 0 1 page fault

Belady's anomaly – Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

- **Optimal Page replacement –**
  In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.Example-2:Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with 4 page frame. Find number of page fault.

Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2,3 No. of Page frame - 4

7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit	Hit	Hit	Hit

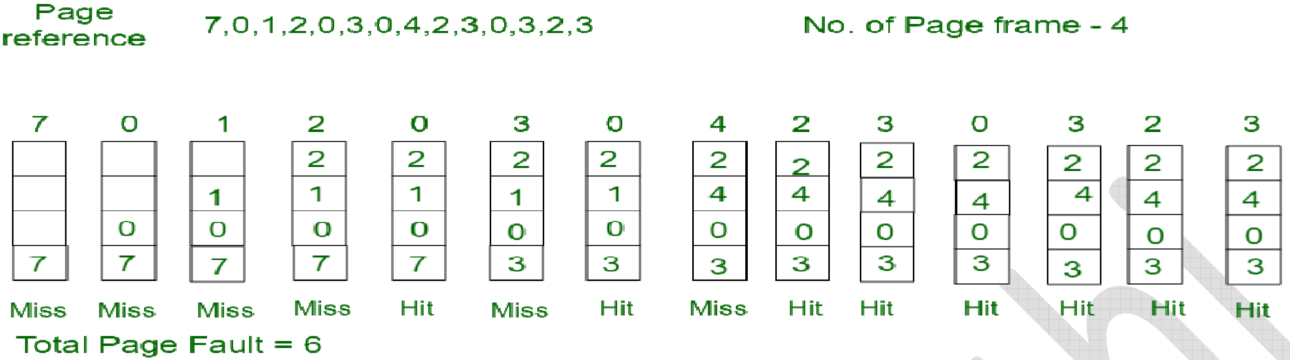
Total Page Fault = 6

Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> 4 Page faults
0 is already there so —> 0 Page fault.
when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future.—>1 Page fault.
0 is already there so —> 0 Page fault..
4 will takes place of 1 —> 1 Page Fault.

Now for the further page reference string —> 0 Page fault because they are already available in the memory.
Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

Partha Sarathi

- **Least Recently Used –**
  In this algorithm page will be replaced which is least recently used.
  Example-3Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 with 4 page frames.Find number of page faults.



  Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> 4 Page faults
  0 is already their so —> 0 Page fault.
  when 3 came it will take the place of 7 because it is least recently used —>1 Page fault
  0 is already in memory so —> 0 Page fault.
  4 will takes place of 1 —> 1 Page Fault
  Now for the further page reference string —> 0 Page fault because they are already available in the memory.

- **Page Buffering algorithm**
  - To get a process start quickly, keep a pool of free frames.
  - On page fault, select a page to be replaced.
  - Write the new page in the frame of free pool, mark the page table and restart the process.
  - Now write the dirty page out of disk and place the frame holding replaced page in free pool.

- **Least frequently Used(LFU) algorithm**
  - The page with the smallest count is the one which will be selected for replacement.
  - This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

- **Most frequently Used(MFU) algorithm**
  - This algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Memory Interleaving

Memory interleaving is the technique used to increase the throughput. The core idea is to split the memory system into independent banks, which can answer read or write requests independents in parallel.

There are two-address format for memory interleaving the address space.

Low order interleaving

Low order interleaving spreads contiguous memory location across the modules horizontally. This implies that the low order bits of the memory address are used to indentify the memory module. High order bits are the word addresses (displacement) within each module.

High order interleaving

High order interleaving uses the high order bits as the module address and the low order bits as the word address within each module.

Partha Sarathi

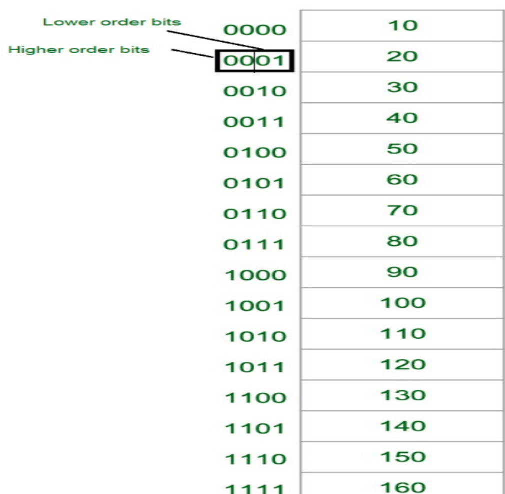
Memory Interleaving is less or More an Abstraction technique. Though its a bit different from Abstraction. It is a Technique which divides memory into a number of modules such that Successive words in the address space are placed in the Different module.

Consecutive Word in a Module:
Let us assume 16 Data's to be transferred to the Four Module. Where Module 00 be Module 1, Module 01 be Module 2, Module 10 be Module 3 & Module 11 be Module 4. Also 10, 20, 30....130 are the data to be transferred.

From the figure below in Module 1, 10 [Data] is transferred then 20, 30 & finally, 40 which are the Data. That means the data are added consecutively in the Module till its max capacity.

Most significant bit (MSB) provides the Address of the Module & least significant bit (LSB) provides the address of the data in the module.
For Example, to get 90 (Data) 1000 will be provided by the processor. In this 10 will indicate that the data is in module 10 (module 3) & 00 is the address of 90 in Module 10 (module 3). So,

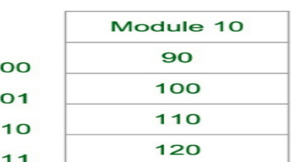
Module 1 Contains Data : 10, 20, 30, 40
Module 2 Contains Data : 50, 60, 70, 80
Module 3 Contains Data : 90, 100, 110, 120
Module 4 Contains Data : 130, 140, 150, 160



Address	Data
0000	10
0001	20
0010	30
0011	40
0100	50
0101	60
0110	70
0111	80
1000	90
1001	100
1010	110
1011	120
1100	130
1101	140
1110	150
1111	160

	Module 00
00	10
01	20
10	30
11	40

	Module 01
00	50
01	60
10	70
11	80



	Module 10
00	90
01	100
10	110
11	120

	Module 11
00	130
01	140
10	150
11	160

Consecutive Word in Consecutive Module:

Lower order bits
Higher order bits

Address	Data
0000	10
0001	20
0010	30
0011	40
0100	50
0101	60
0110	70
0111	80
1000	90
1001	100
1010	110
1011	120
1100	130
1101	140
1110	150
1111	160

	Module 00
00	10
01	50
10	90
11	130

	Module 01
00	20
01	60
10	100
11	140

	Module 10
00	30
01	70
10	110
11	150

	Module 11
00	40
01	80
10	120
11	160

Now again we assume 16 Data's to be transferred to the Four Module. But Now the consecutive Data are added in Consecutive Module. That is, 10 [Data] is added in Module 1, 20 [Data] in Module 2 and So on.

Least Significant Bit (LSB) provides the Address of the Module & Most significant bit (MSB) provides the address of the data in the module.

For Example, to get 90 (Data) 1000 will be provided by the processor. In this 00 will indicate that the data is in module 00 (module 1) & 10 is the address of 90 in Module 00 (module 1). That is,

Module 1 Contains Data : 10, 50, 90, 130
Module 2 Contains Data : 20, 60, 100, 140
Module 3 Contains Data : 30, 70, 110, 150
Module 4 Contains Data : 40, 80, 120, 160

Why we use Memory Interleaving? [Advantages]:
Whenever, Processor request Data from the main memory. A block (chunk) of Data is Transferred to the cache and then to Processor. So whenever a cache miss occurs the Data is to be fetched from main memory. But main memory is relatively slower than the cache. So to improve the access time of the main memory interleaving is used.

We can access all four Module at the same time thus achieving Parallelism. From Figure 2 the data can be acquired from the Module using the Higher bits. This method Uses memory effectively.

Mod-4

Data Flow Architecture

- Data Flow Architecture is transformed input data by a series of computational or manipulative components into output data.
- It is a computer architecture which do not have a program counter and therefore the execution is unpredictable which means behavior is indeterministic.
- Data flow architecture is a part of Von-neumann model of computation which consists of a single program counter, sequential execution and control flow which determines fetch, execution, commit order.
- This architecture has been successfully implemented.
- Data flow architecture reduces development time and can move easily between design and implementation.
- It has main objective is to achieve the qualities of reuse and modifiability.
- In data flow architecture, the data can be flow in the graph topology with cycles or in a linear structure without cycles.

There are three types of execution sequences between modules:

1. Batch Sequential
2. Pipe and Filter
3. Process Control

1. Batch Sequential

- Batch sequential compilation was regarded as a sequential process in 1970.
- In Batch sequential, separate programs are executed in order and the data is passed as an aggregate from one program to the next.
- It is a classical data processing model.

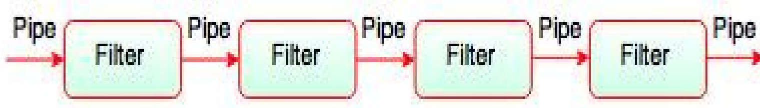


Fig. Batch Sequential

The above diagram shows the flow of batch sequential architecture. It provides simpler divisions on subsystems and each subsystem can be an independent program working on input data and produces output data.

- The main disadvantage of batch sequential architecture is that, it does not provide concurrency and interactive interface. It provides high latency and low throughput.

Partha Sarathi

2. Pipe and Filter

What is meant by Pipe?

- Pipe is a connector which passes the data from one filter to the next.
- Pipe is a directional stream of data implemented by a data buffer to store all data, until the next filter has time to process it.
- It transfers the data from one data source to one data sink.
- Pipes are the stateless data stream.

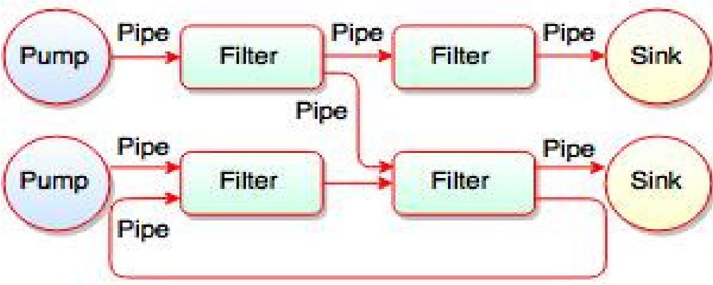


Fig. Pipes and Filters

The above figure shows the pipe-filter sequence. All filters are the processes that run at the same time, it means that they can run as different threads, coroutines or be located on different machines entirely.

Each pipe is connected to a filter and has its own role in the function of the filter. The filters are robust where pipes can be added and removed at runtime.

Filter reads the data from its input pipes and performs its function on this data and places the result on all output pipes. If there is insufficient data in the input pipes, the filter simply waits.

What are the Filters?

- Filter is a component.
- It has interfaces from which a set of inputs can flow in and a set of outputs can flow out.
- It transforms and refines input data.
- Filters are the independent entities.

There are two strategies to construct a filter:

1. **Active filter** derives the data flow on the pipes.
2. **Passive filter** is driven by the data flow on the pipes.

- Filter does not share state with other filters.
- They don't know the identity to upstream and downstream filters.
- Filters are implemented by separate threads. These may be either hardware or software threads or coroutines.

Advantages of Pipes and Filters

- Pipe-filter provides concurrency and high throughput for excessive data processing.
- It simplifies the system maintenance and provides reusability.
- It has low coupling between filters and flexibility by supporting both sequential and parallel execution.

Disadvantages of Pipe and Filter

- Pipe and Filter are not suitable for dynamic interactions.
- It needs low common denominator for transmission of data in ASCII format.
- It is difficult to configure Pipe-filter architecture dynamically.

3. Process Control

- Process Control Architecture is a type of Data Flow Architecture, where data is neither batch sequential nor pipe stream.
- In process control architecture, the flow of data comes from a set of variables which controls the execution of process.

Partha Sarathi

- This architecture decomposes the entire system into subsystems or modules and connects them.
- Process control architecture is suitable in the embedded system software design, where the system is manipulated by process control variable data and in the Real time system software, process control architecture is used to control automobile anti-lock brakes, nuclear power plants etc.
- This architecture is applicable for car-cruise control and building temperature control system.

Data flow computers are based on the concept of data-driven computation, which is drastically different from the operation of conventional von Neumann machine. The fundamental difference is that instruction execution in a conventional computer is under program-flow control, whereas that in a data flow computer is driven by the data (operand) availability.
The data-driven concept means asynchrony, which means that many instructions can be executed simultaneously and asynchronously. A higher degree of implicit parallelism is expected in dataflow computer. Because there is no use of shared memory cells, dataflow programs are free from side effects.
The Dataflow Principles section reviews the basic principles of the dataflow model. The Dataflow Graphs section gives the representations used in dataflow system. The Dataflow Architectures section provides a general description of the dataflow architecture. The discussion includes a comparison of the architectural characteristics and the evolutionary improvements in dataflow computing.

OBJECTIVE

Dataflow architecture is a computer architecture that directly opposite of the traditional von Neumann architecture or control flow architecture.
It has been successfully implemented in specialized hardware such as in digital signal processing, network routing, graphics processing, telemetry, and more recently in data warehousing.
The main objective and scope of our Project is about the discuss of principle and the uses of dataflow computer.

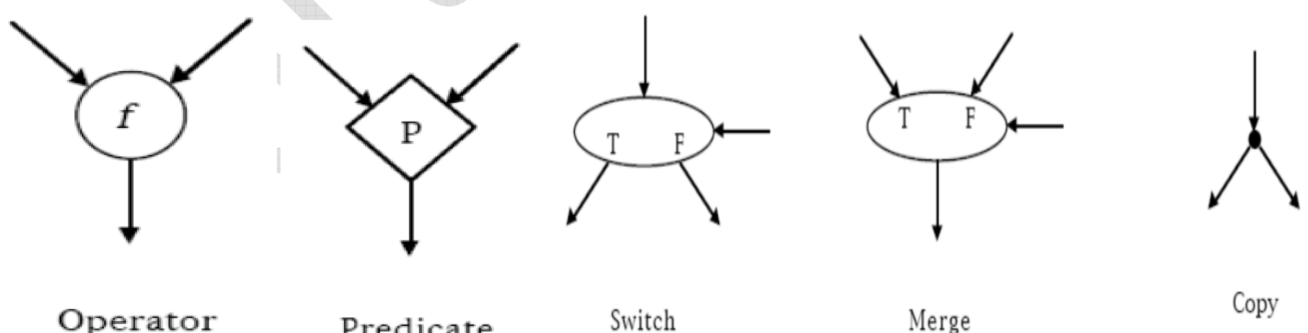
Features Of Dataflow Computers

- Intermediate or final results are passed directly as data token between instructions.
- There is no concept of shared data storage as embodied the traditional notation of a variable.
- Program sequencing is constrained only by data dependency among instructions.

DATA FLOW GRAPH

Dataflow graphs can be viewed as the machine language for dataflow computers. A data flow graph is a directed graph whose nodes correspond to operators and arcs are pointers for forwarding data tokens.

A producing node is connected to a consuming node by an arc, and the “point” where an arc enters a node is called an input port. The execution of an instruction is called the firing of a node. Data is sent along the arcs of the dataflow graph in the form of tokens, which are created by computational nodes and placed on output arcs.

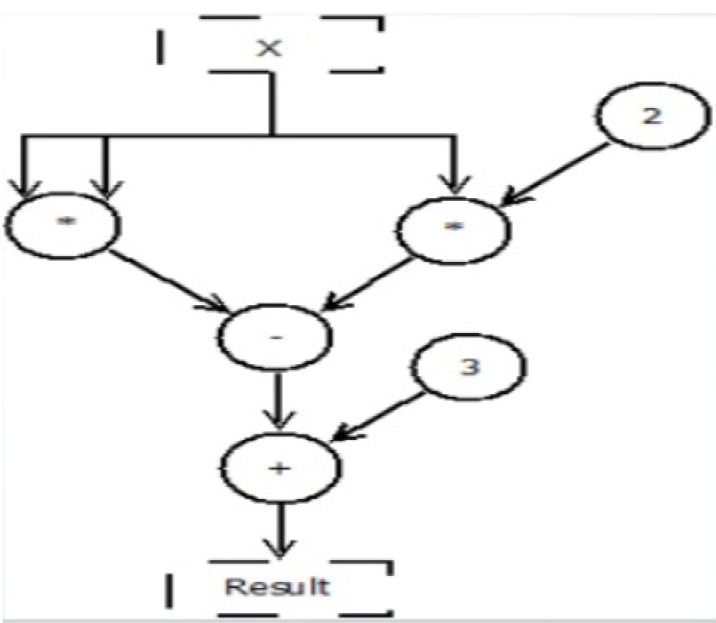


- The machine level language is represented by dataflow graphs.
- A data value is produced by an Operator as a result of some operation, f.
- A True or False control value is generated by a Decider (a predicate) depending on its input tokens.
- Data values are directed by means of either a Switch or a Merge actor.
- a Copy is an identity operator which duplicates input tokens.

Example With a expression

The below figure illustrates an example of dataflow graph for evaluation of expression **X^2– 2*X + 3.**

The following subtraction operation will not be carried out until these values are available. As soon as X2 and 2*X values are computed subtraction operation will be carried out which in turn will provide input to next addition operation.



MODELS OF DATAFLOWARCHITECTURE :

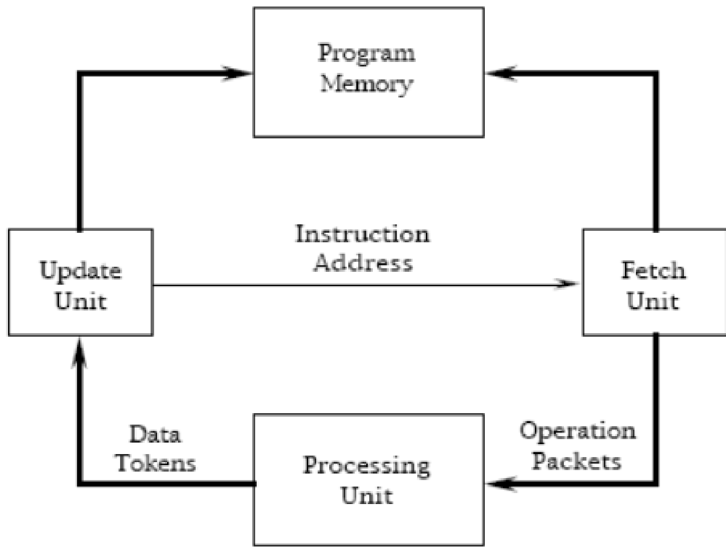
Depending on the way of handling data tokens, data flow computers are divided into :

I. static model
II. I dynamic model.

STATIC DATA FLOW MACHINES :

In static data flow machine, data tokens are assumed to move along the arcs of the data flow program graph to the operator nodes.

This architecture is considered static because tokens are not labeled and control tokens must be used to acknowledge the proper timing in transferring data tokens from node to node.



Opcode	
PB	Operand 1
PB	Operand 2
Destination 1	
Destination 2	

The Update Unit

- responsible for detecting the executability of instructions.
- sends the address of the enabled instruction to the Fetch Unit.

The Fetch Unit

– fetches and sends a complete operation packet containing the corresponding opcode, data, and destination list to the Processing Unit
– clears the presence bits.

The Processing Unit

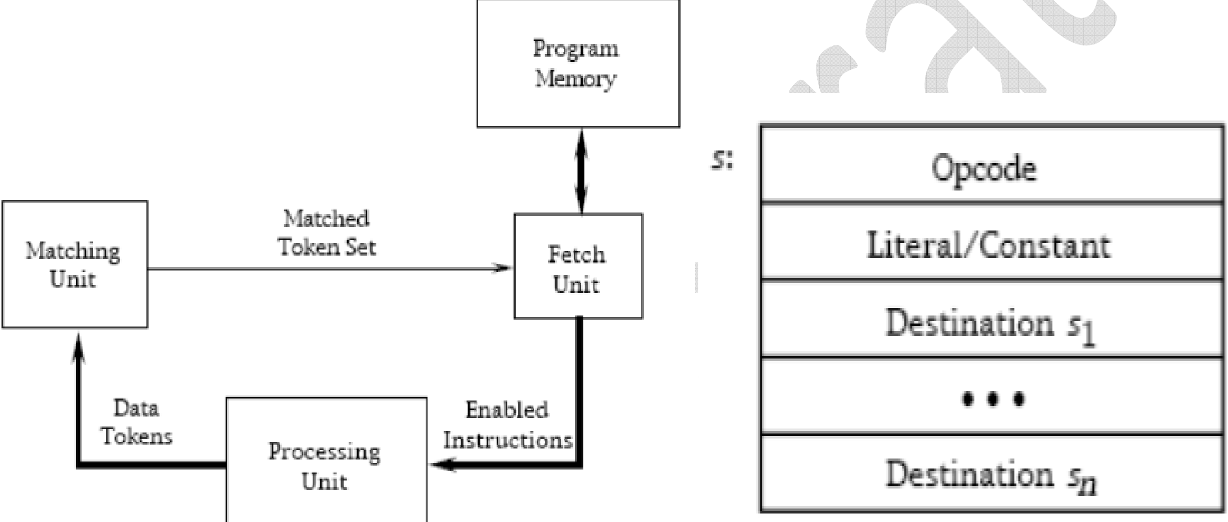
– performs the operation, forms a result packets, and sends them to the Update Unit.

The Update Unit

– stores each result in the appropriate operand slot and checks the presence bits to determine whether the activity is enabled.

DYNAMIC DATAFLOW MACHINE

- A dynamic data flow machine uses tagged tokens, so that more than one token can exist in an arc.The tagging is achieved by attaching a label with each token which uniquely identifies the context of that particular token.
- The dynamic dataflow allows greater exploitation of parallelism; however, this advantage comes at the expense of the overhead in terms of the generation of tags, larger data tokens, and complexity of the matching tokens.



The Matching Unit

– a memory containing a pool of waiting tokens
– Tokens are received by it.
– brings together tokens with identical tags.
– If a match exists, the matched token set is passed on to the Fetch Unit.
– If no match is found, the token is stored in the Matching Unit to await a partner.

In the Fetch Unit

– the tags of the token pair uniquely identify an instruction to be fetched from the Program Memory.
– The instruction together with the token pair forms the enabled instruction and is sent to the Processing Unit.

The Processing Unit

– executes the enabled instructions and produces result tokens to be sent to the Matching Unit.

Drawbacks in dataflow computing

• Too fine-grained parallelism
– Incurs more overhead in the execution of an instruction cycle
– Poor performance in applications with low degree of parallelism.

• Inefficiency in representing and handling data structures (e.g., arrays of data).
– Collections of many tokens

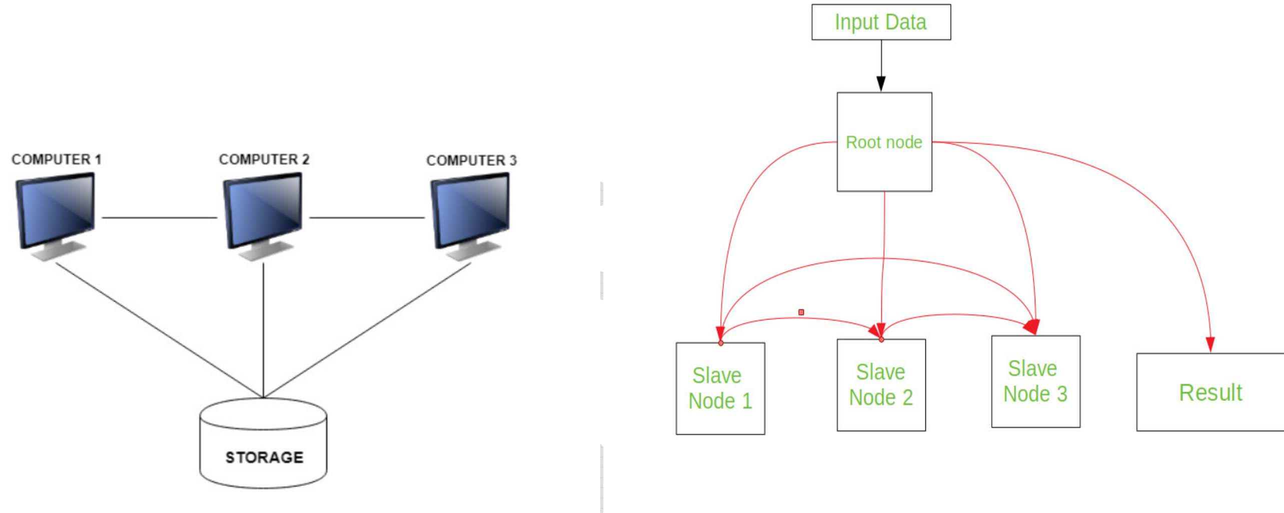
• Need for large token stores to handle all operations waiting for execution
• Means to schedule hundreds or thousands of operations that are ready to execute in a limited amount of available hardware.
• Difficulty of debugging
• The dataflow computing model has not been adopted in mainstream processor design.

Cluster computers

Cluster is a set of loosely or tightly connected computers working together as a unified computing resource that can create the illusion of being one machine. Computer clusters have each node set to perform the same task, controlled and produced by software.

The *components of a clusters* are usually connected to each other using fast area networks, with each node running its own instance of an operating system. In most circumstances, all the nodes uses same hardware and the same operating system, although in some setups different hardware or different operating system can be used.

Clustered systems are similar to parallel systems as they both have multiple CPUs. However a major difference is that clustered systems are created by two or more individual computer systems merged together. Basically, they have independent computer systems with a common storage and the systems work together.



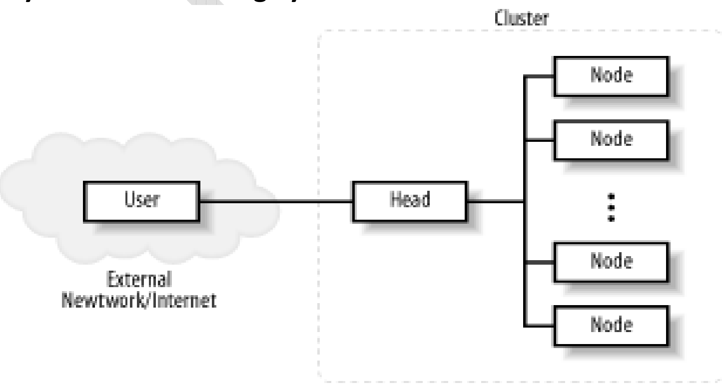
The clustered systems are a combination of hardware clusters and software clusters. The hardware clusters help in sharing of high performance disks between the systems. The software clusters makes all the systems work together .

Each node in the clustered systems contains the cluster software. This software monitors the cluster system and makes sure it is working as required. If any one of the nodes in the clustered system fail, then the rest of the nodes take control of its storage and resources and try to restart.

Types of Clustered Systems

There are primarily two types of clustered systems i.e. asymmetric clustering system and symmetric clustering system. Details about these are given as follows:

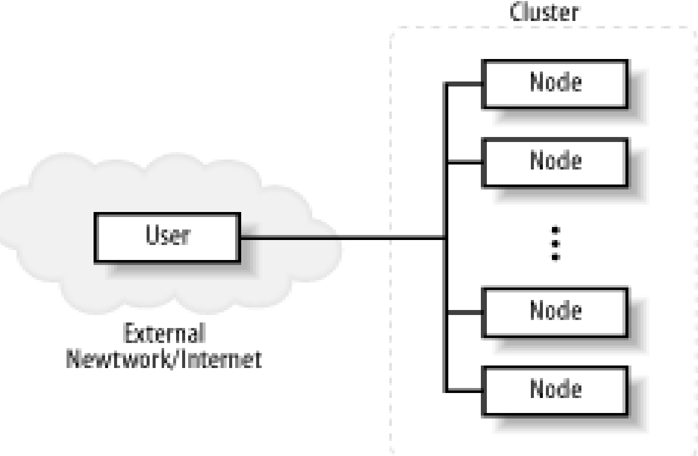
Asymmetric Clustering System



In this system, one of the nodes in the clustered system is in hot standby mode and all the others run the required applications. The hot standby mode is a failsafe in which a hot standby node is part of the system . The hot standby node continuously monitors the server and if it fails, the hot standby node takes its place.

Symmetric Clustering System

In symmetric clustering system two or more nodes all run applications as well as monitor each other. This is more efficient than asymmetric system as it uses all the hardware and doesn't keep a node merely as a hot standby.

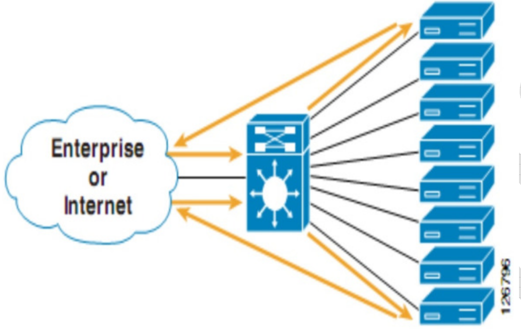


Attributes of Clustered Systems

There are many different purposes that a clustered system can be used for. Some of these can be scientific calculations, web support etc. The clustering systems that embody some major attributes are:

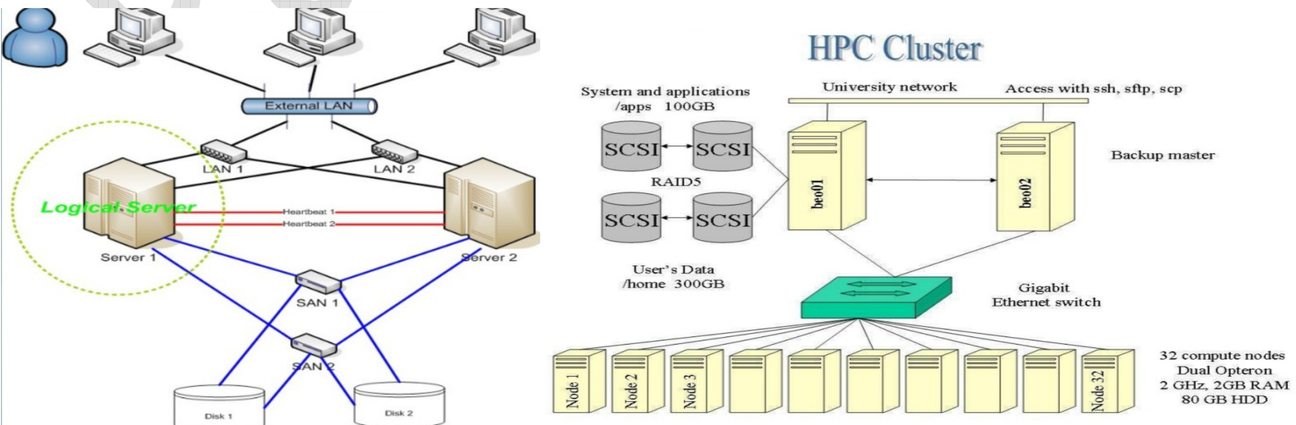
- **Load Balancing Clusters**

In this type of clusters, the nodes in the system share the workload to provide a better performance. For example: A web based cluster may assign different web queries to different nodes so that the system performance is optimized. Some clustered systems use a round robin mechanism to assign requests to different nodes in the system.



- **High Availability Clusters**

These clusters improve the availability of the clustered system. They have extra nodes which are only used if some of the system components fail. So, high availability clusters remove single points of failure i.e. nodes whose failure leads to the failure of the system. These types of clusters are also known as failover clusters or HA clusters.



- **Fail-Over Clusters**

The function of switching applications and data resources over from a failed system to an alternative system in the cluster is referred to as fail-over. These types are used to cluster database of critical mission, mail, file and application servers

Benefits of Clustered Systems

The difference benefits of clustered systems are as follows:

- **Performance**

Clustered systems result in high performance as they contain two or more individual computer systems merged together. These work as a parallel unit and result in much better performance for the system.

- **Fault Tolerance**

Clustered systems are quite fault tolerant and the loss of one node does not result in the loss of the system. They may even contain one or more nodes in hot standby mode which allows them to take the place of failed nodes.

- **Scalability**

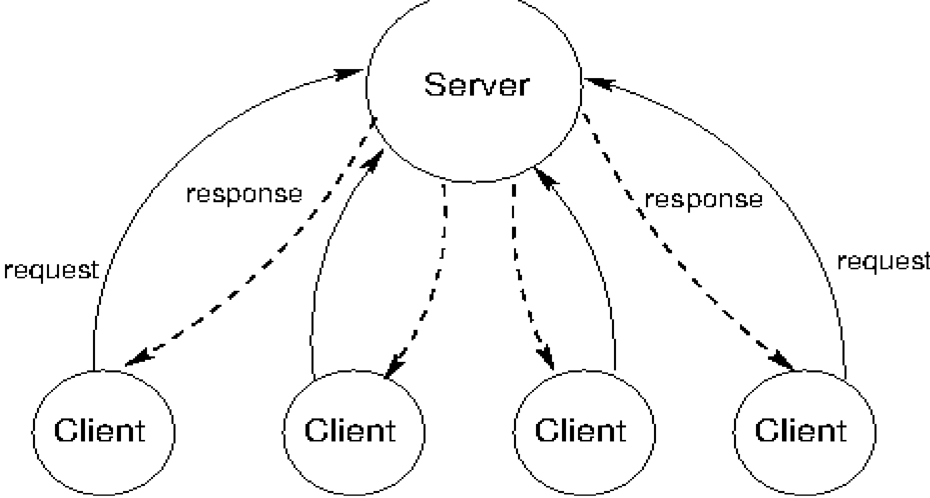
Clustered systems are quite scalable as it is easy to add a new node to the system. There is no need to take the entire cluster down to add a new node.

Distributed computing

- A distributed system consists of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility.
- A distributed system is a network of autonomous computers that communicate with each other in order to achieve a goal.
- The computers in a distributed system are independent and do not physically share memory or processors.
- They communicate with each other using *messages*, pieces of information transferred from one computer to another over a network.
- Messages can communicate many things: computers can tell other computers to execute a procedures with particular arguments, they can send and receive packets of data, or send signals that tell other computers to behave a certain way.
- Computers in a distributed system can have different roles. A computer's role depends on the goal of the system and the computer's own hardware and software properties. There are two predominant ways of organizing computers in a distributed system.
  - The first is the client-server architecture
  - the second is the peer-to-peer architecture.

Client/Server Systems

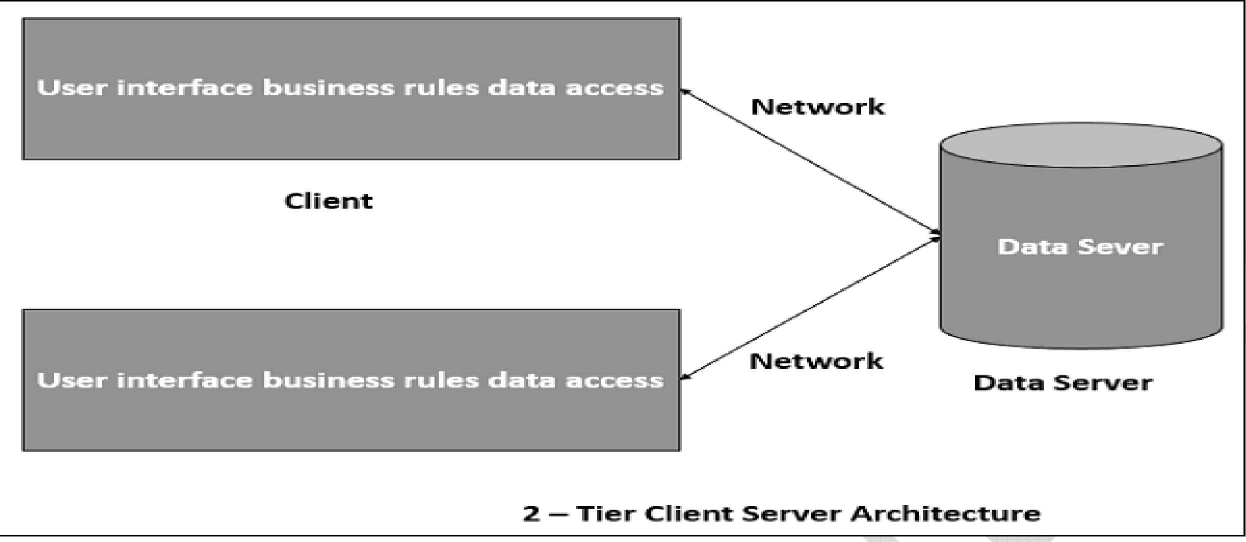
The client-server architecture is a way to dispense a service from a central source. There is a single *server* that provides a service, and multiple *clients* that communicate with the server to consume its products. In this architecture, clients and servers have different jobs. The server's job is to respond to service requests from clients, while a client's job is to use the data provided in response in order to perform some task.



The client-server architecture is the most common distributed system architecture which decomposes the system into two major subsystems or logical processes −

- **Client** − This is the first process that issues a request to the second process i.e. the server.
- **Server** − This is the second process that receives the request, carries it out, and sends a reply to the client.

In this architecture, the application is modelled as a set of services that are provided by servers and a set of clients that use these services. The servers need not know about clients, but the clients must know the identity of servers, and the mapping of processors to processes is not necessarily 1 : 1



Client-server Architecture can be classified into two models based on the functionality of the client −

Thin-client model

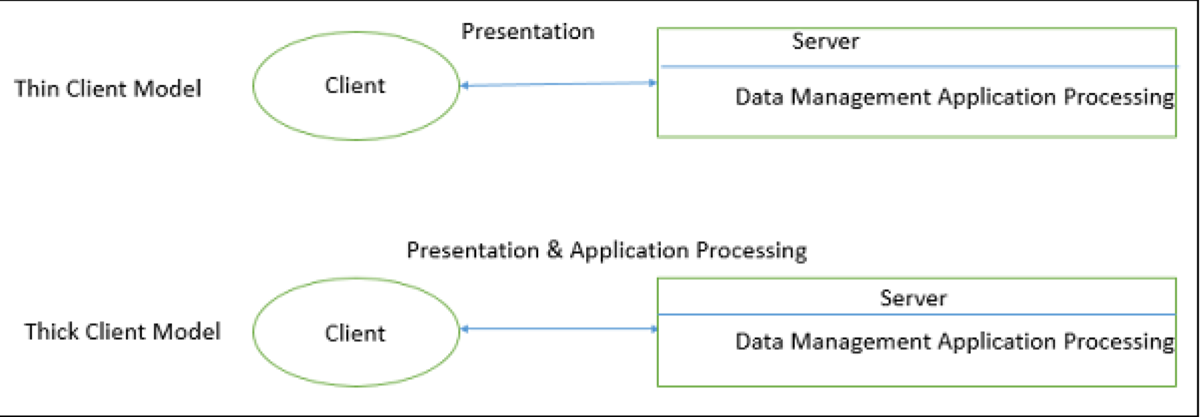
In thin-client model, all the application processing and data management is carried by the server. The client is simply responsible for running the presentation software.

- Used when legacy systems are migrated to client server architectures in which legacy system acts as a server in its own right with a graphical interface implemented on a client
- A major disadvantage is that it places a heavy processing load on both the server and the network.

Thick/Fat-client model

In thick-client model, the server is only in charge for data management. The software on the client implements the application logic and the interactions with the system user.

- Most appropriate for new C/S systems where the capabilities of the client system are known in advance
- More complex than a thin client model especially for management. New versions of the application have to be installed on all clients.



Advantages

- Separation of responsibilities such as user interface presentation and business logic processing.
- Reusability of server components and potential for concurrency
- Simplifies the design and the development of distributed applications

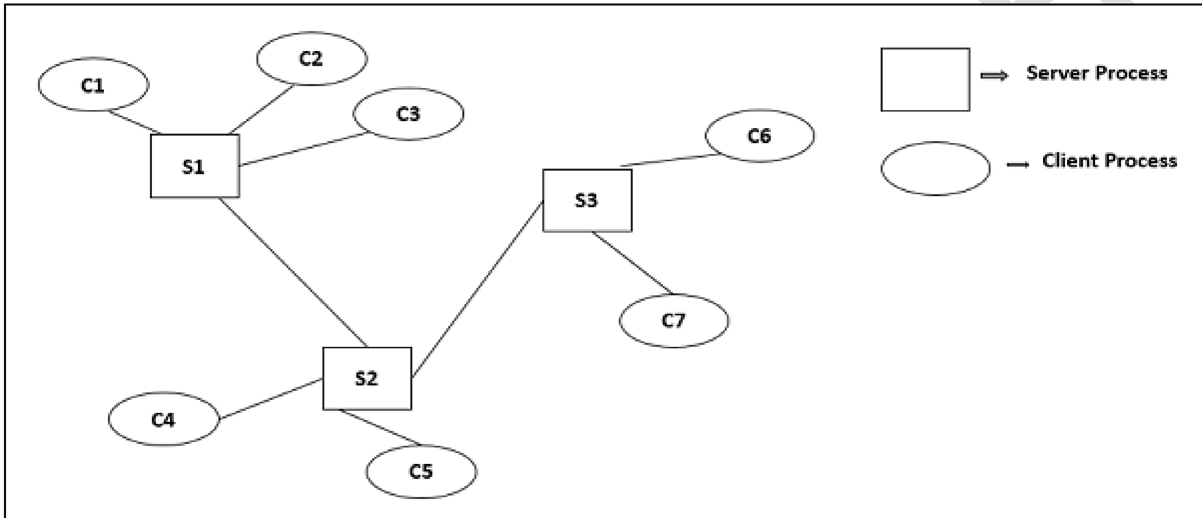
- It makes it easy to migrate or integrate existing applications into a distributed environment.
- It also makes effective use of resources when a large number of clients are accessing a high-performance server.

Disadvantages

- Lack of heterogeneous infrastructure to deal with the requirement changes.
- Security complications.
- Limited server availability and reliability.
- Limited testability and scalability.
- Fat clients with presentation and business logic together.

Multi-Tier Architecture (n-tier Architecture)

Multi-tier architecture is a client–server architecture in which the functions such as presentation, application processing, and data management are physically separated. By separating an application into tiers, developers obtain the option of changing or adding a specific layer, instead of reworking the entire application. It provides a model by which developers can create flexible and reusable applications.



The most general use of multi-tier architecture is the three-tier architecture. A three-tier architecture is typically composed of a presentation tier, an application tier, and a data storage tier and may execute on a separate processor.

Presentation Tier

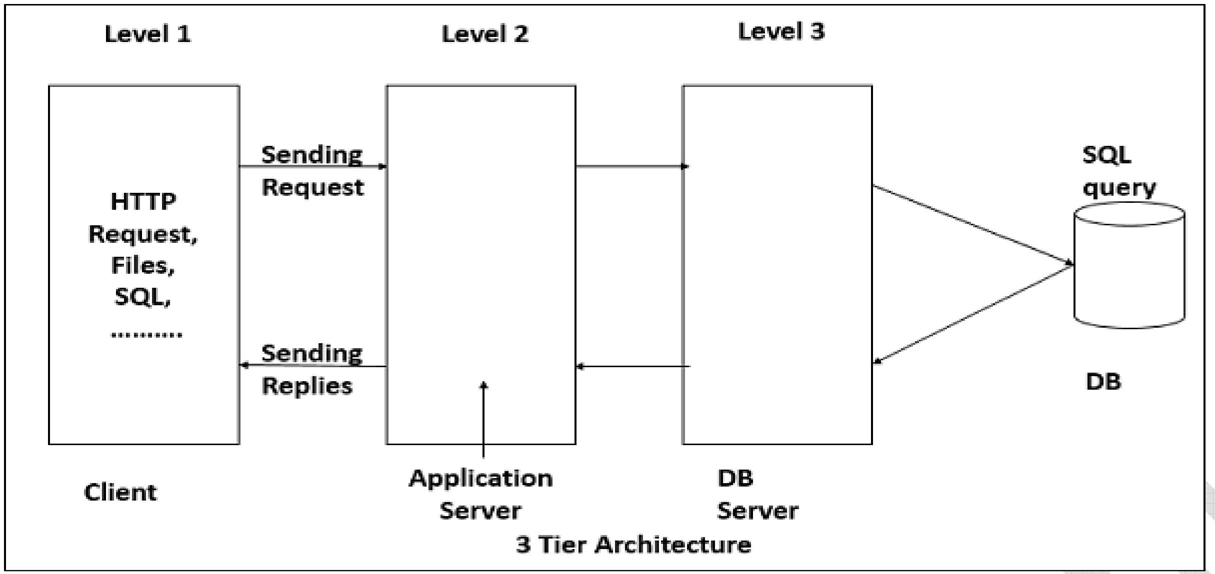
Presentation layer is the topmost level of the application by which users can access directly such as webpage or Operating System GUI (Graphical User interface). The primary function of this layer is to translate the tasks and results to something that user can understand. It communicates with other tiers so that it places the results to the browser/client tier and all other tiers in the network.

Application Tier (Business Logic, Logic Tier, or Middle Tier)

Application tier coordinates the application, processes the commands, makes logical decisions, evaluation, and performs calculations. It controls an application’s functionality by performing detailed processing. It also moves and processes data between the two surrounding layers.

Data Tier

In this layer, information is stored and retrieved from the database or file system. The information is then passed back for processing and then back to the user. It includes the data persistence mechanisms (database servers, file shares, etc.) and provides API (Application Programming Interface) to the application tier which provides methods of managing the stored data.



3 Tier Architecture

Advantages

- Better performance than a thin-client approach and is simpler to manage than a thick-client approach.
- Enhances the reusability and scalability − as demands increase, extra servers can be added.
- Provides multi-threading support and also reduces network traffic.
- Provides maintainability and flexibility

Disadvantages

- Unsatisfactory Testability due to lack of testing tools.
- More critical server reliability and availability.

Broker Architectural Style

Broker Architectural Style is a middleware architecture used in distributed computing to coordinate and enable the communication between registered servers and clients. Here, object communication takes place through a middleware system called an object request broker (software bus).

- Client and the server do not interact with each other directly. Client and server have a direct connection to its proxy which communicates with the mediator-broker.
- A server provides services by registering and publishing their interfaces with the broker and clients can request the services from the broker statically or dynamically by look-up.
- CORBA (Common Object Request Broker Architecture) is a good implementation example of the broker architecture.

Components of Broker Architectural Style

The components of broker architectural style are discussed through following heads −

Broker

Broker is responsible for coordinating communication, such as forwarding and dispatching the results and exceptions. It can be either an invocation-oriented service, a document or message - oriented broker to which clients send a message.

- It is responsible for brokering the service requests, locating a proper server, transmitting requests, and sending responses back to clients.
- It retains the servers' registration information including their functionality and services as well as location information.
- It provides APIs for clients to request, servers to respond, registering or unregistering server components, transferring messages, and locating servers.

Stub

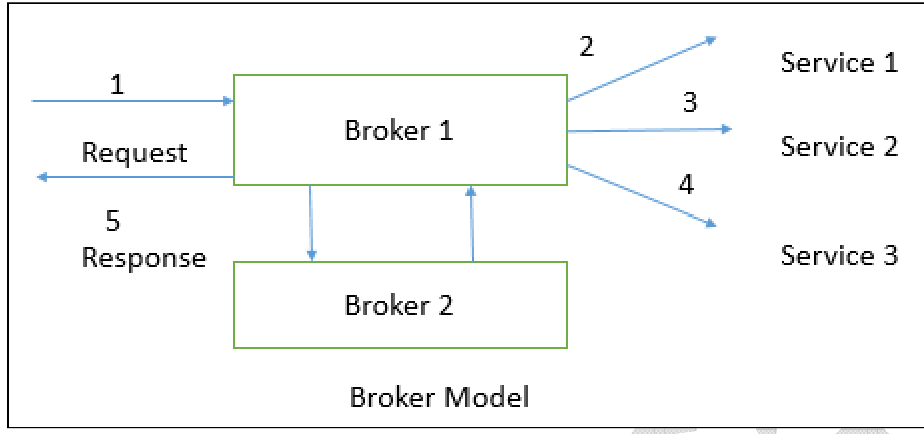
Stubs are generated at the static compilation time and then deployed to the client side which is used as a proxy for the client. Client-side proxy acts as a mediator between the client and the broker and provides additional transparency between them and the client; a remote object appears like a local one.

The proxy hides the IPC (inter-process communication) at protocol level and performs marshaling of parameter values and un-marshaling of results from the server.

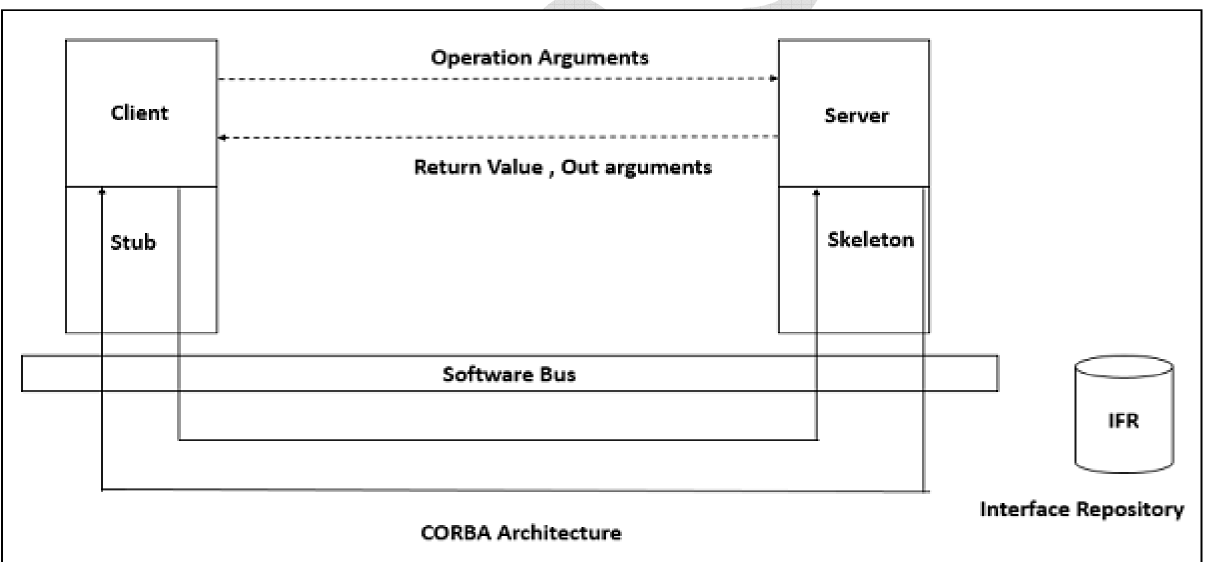
Partha Sarathi

Skeleton
Skeleton is generated by the service interface compilation and then deployed to the server side, which is used as a proxy for the server. Server-side proxy encapsulates low-level system-specific networking functions and provides high-level APIs to mediate between the server and the broker.
It receives the requests, unpacks the requests, unmarshals the method arguments, calls the suitable service, and also marshals the result before sending it back to the client.

Bridge
A bridge can connect two different networks based on different communication protocols. It mediates different brokers including DCOM, .NET remote, and Java CORBA brokers.
Bridges are optional component, which hides the implementation details when two brokers interoperate and take requests and parameters in one format and translate them to another format.

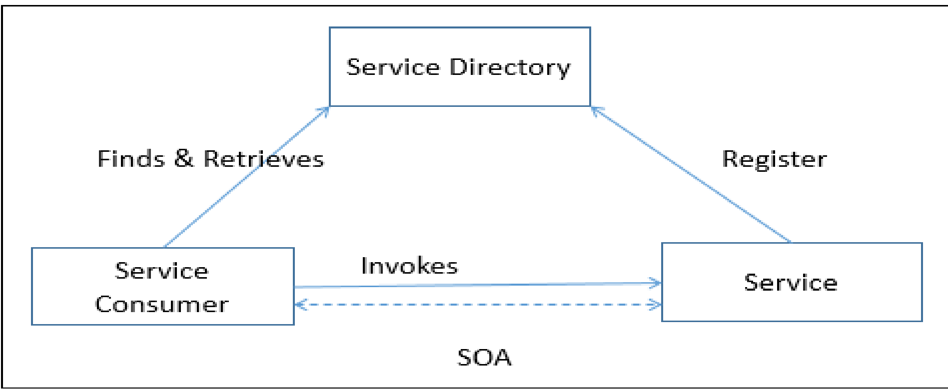


Broker implementation in CORBA
CORBA is an international standard for an Object Request Broker – a middleware to manage communications among distributed objects defined by OMG (object management group).



Service-Oriented Architecture (SOA)
A service is a component of business functionality that is well-defined, self-contained, independent, published, and available to be used via a standard programming interface. The connections between services are conducted by common and universal message-oriented protocols such as the SOAP Web service protocol, which can deliver requests and responses between services loosely.

Service-oriented architecture is a client/server design which support business-driven IT approach in which an application consists of software services and software service consumers (also known as clients or service requesters).



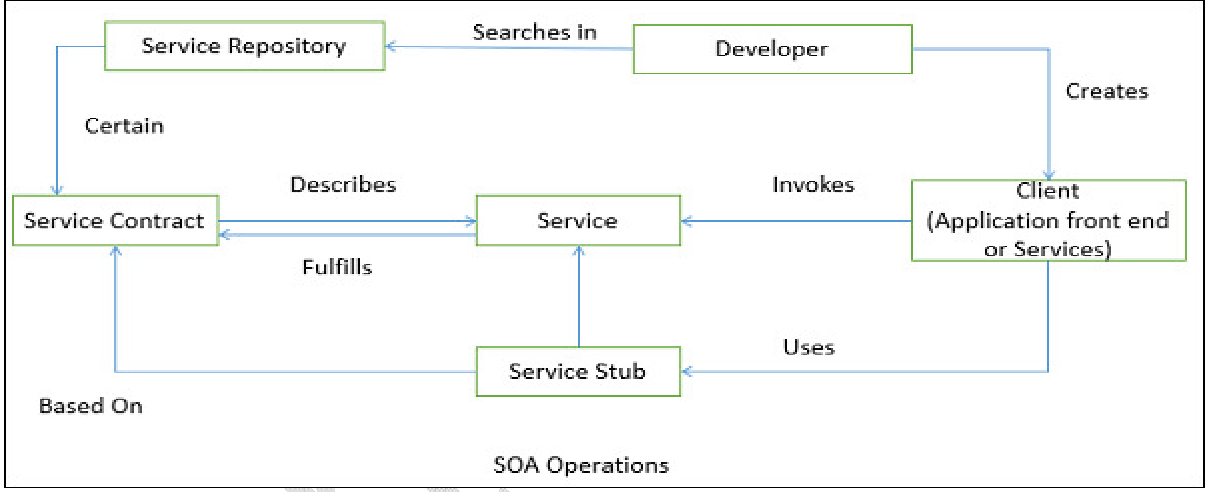
Features of SOA

A service-oriented architecture provides the following features –

- **Distributed Deployment** – Expose enterprise data and business logic as loosely, coupled, discoverable, structured, standard-based, coarse-grained, stateless units of functionality called services.
- **Composability** – Assemble new processes from existing services that are exposed at a desired granularity through well defined, published, and standard complaint interfaces.
- **Interoperability** – Share capabilities and reuse shared services across a network irrespective of underlying protocols or implementation technology.
- **Reusability** – Choose a service provider and access to existing resources exposed as services.

SOA Operation

The following figure illustrates how does SOA operate –



Advantages

- Loose coupling of service–orientation provides great flexibility for enterprises to make use of all available service recourses irrespective of platform and technology restrictions.
- Each service component is independent from other services due to the stateless service feature.
- The implementation of a service will not affect the application of the service as long as the exposed interface is not changed.
- A client or any service can access other services regardless of their platform, technology, vendors, or language implementations.
- Reusability of assets and services since clients of a service only need to know its public interfaces, service composition.
- SOA based business application development are much more efficient in terms of time and cost.
- Enhances the scalability and provide standard connection between systems.
- Efficient and effective usage of 'Business Services'.
- Integration becomes much easier and improved intrinsic interoperability.
- Abstract complexity for developers and energize business processes closer to end users.

Peer-to-peer Systems

The client-server model is appropriate for service-oriented situations. However, there are other computational goals for which a more equal division of labor is a better choice. The term *peer-to-peer* is used to describe distributed systems in which labor is divided among all the components of the system. All the computers send and receive data, and they all contribute some processing power and memory. As a distributed system increases in size, its capacity of computational resources increases. In a peer-to-peer system, all components of the system contribute some

processing power and memory to a distributed computation.

Division of labor among *all* participants is the identifying characteristic of a peer-to-peer system. This means that peers need to be able to communicate with each other reliably. In order to make sure that messages reach their intended destinations, peer-to-peer systems need to have an organized network structure. The components in these systems cooperate to maintain enough information about the locations of other components to send messages to

intended destinations.

In some peer-to-peer systems, the job of maintaining the health of the network is taken on by a set of specialized components. Such systems are not pure peer-to-peer systems, because they have different types of components that serve different functions. The components that support a peer-to-peer network act like scaffolding: they help the network stay connected, they maintain information about the locations of different computers, and they help newcomers take their place within their neighborhood.

The most common applications of peer-to-peer systems are data transfer and data storage. For data transfer, each computer in the system contributes to send data over the network. If the destination computer is in a particular computer's neighborhood, that computer helps send data along. For data storage, the data set may be too large to fit on any single computer, or too valuable to store on just a single computer. Each computer stores a small portion of the data, and there may be multiple copies of the same data spread over different computers. When a computer fails, the data that was on it can be restored from other copies and put back when a replacement arrives.

Skype, the voice- and video-chat service, is an example of a data transfer application with a peer-to-peer

architecture. When two people on different computers are having a Skype conversation, their communications are broken up into packets of 1s and 0s and transmitted through a peer-to-peer network. This network is composed of other people whose computers are signed into Skype. Each computer knows the location of a few other computers in its neighborhood. A computer helps send a packet to its destination by passing it on a neighbor, which passes it on to some other neighbor, and so on, until the packet reaches its intended destination. Skype is not a pure peer-to-peer system. A scaffolding network of *supernodes* is responsible for logging-in and logging-out users, maintaining information about the locations of their computers, and modifying the network structure to deal with users entering and leaving.

Modularity

The two architectures we have just considered -- peer-to-peer and client-server -- are designed to enforce *modularity*. Modularity is the idea that the components of a system should be black boxes with respect to each other. It should not matter how a component implements its behavior, as long as it upholds an *interface*: a specification for what outputs will result from inputs.

Modularity gives a system many advantages, and is a property of thoughtful system design. First, a modular system is easy to understand. This makes it easier to change and expand. Second, if something goes wrong with the system, only the defective components need to be replaced. Third, bugs or malfunctions are easy to localize. If the output of a component doesn't match the specifications of its interface, even though the inputs are correct, then that component is the source of the malfunction.

Message Passing

In distributed systems, components communicate with each other using message passing. A message has three essential parts: the **sender**, the **recipient**, and the **content**. The sender needs to be specified so that the recipient knows which component sent the message, and where to send replies. The recipient needs to be specified so that any computers who are helping send the message know where to direct it. The content of the message is the most variable. Depending on the function of the overall system, the content can be a piece of data, a signal, or instructions for the remote computer to evaluate a function with some arguments.

Partha Sarathi

At a high level, message contents can be complex data structures, but at a low level, messages are simply streams of 1s and 0s sent over a network. In order to be usable, all messages sent over a network must be formatted according to a consistent *message protocol*.

A **message protocol** is a set of rules for encoding and decoding messages. Many message protocols specify that a message conform to a particular format, in which certain bits have a consistent meaning. A fixed format implies fixed encoding and decoding rules to generate and read that format. All the components in the distributed system must understand the protocol in order to communicate with each other. That way, they know which part of the message corresponds to which information.

Message protocols are not particular programs or software libraries. Instead, they are rules that can be applied by a variety of programs, even written in different programming languages. As a result, computers with vastly different software systems can participate in the same distributed system, simply by conforming to the message protocols that govern the system.

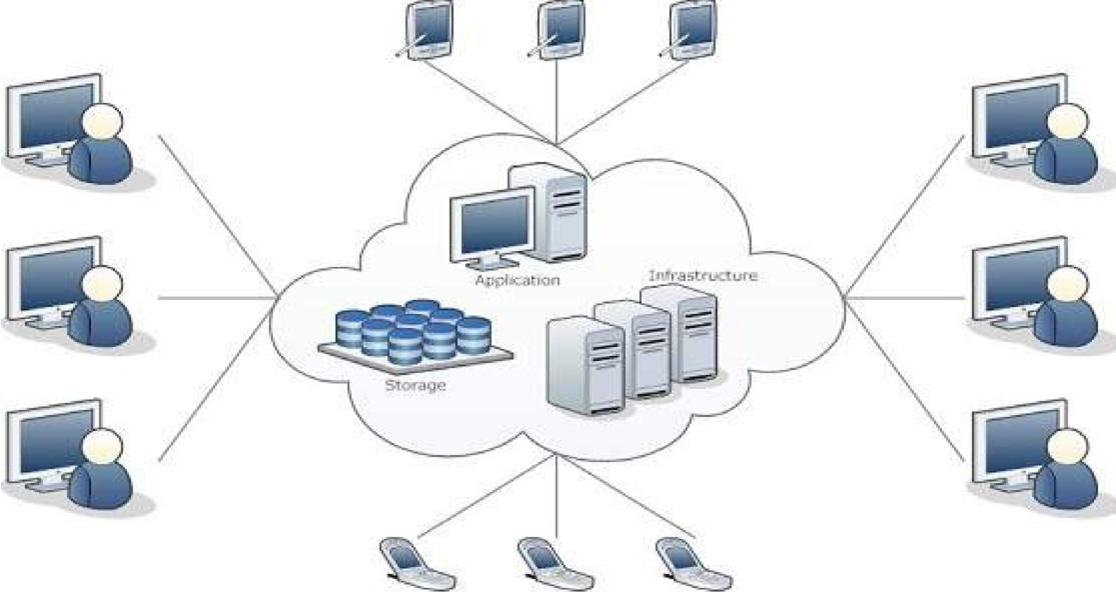
Cloud computing

What is Cloud?

The term Cloud refers to a Network or Internet. In other words, we can say that Cloud is something, which is present at remote location. Cloud can provide services over public and private networks, i.e., WAN, LAN or VPN. Applications such as e-mail, web conferencing, customer relationship management (CRM) execute on cloud.

What is Cloud Computing?

Cloud Computing refers to manipulating, configuring, and accessing the hardware and software resources remotely. It offers online data storage, infrastructure, and application.



Cloud computing offers platform independency, as the software is not required to be installed locally on the PC. Hence, the Cloud Computing is making our business applications mobile and collaborative.

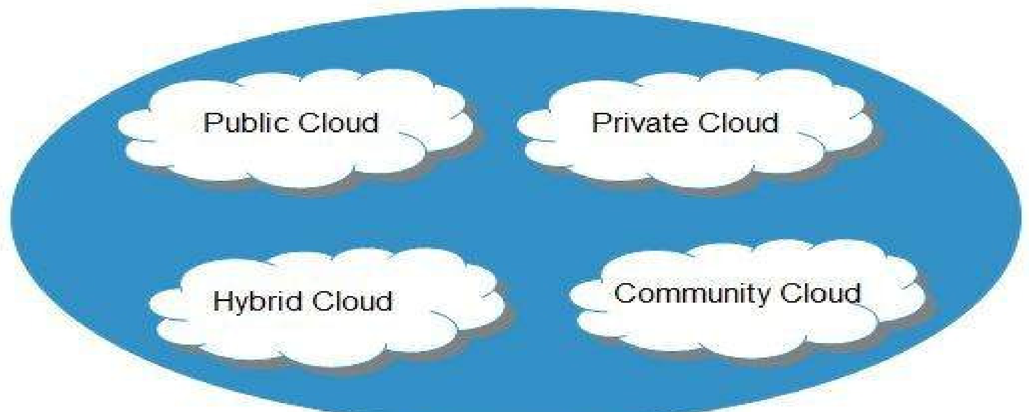
Basic Concepts

There are certain services and models working behind the scene making the cloud computing feasible and accessible to end users. Following are the working models for cloud computing:

- Deployment Models
- Service Models

Deployment Models

Deployment models define the type of access to the cloud, i.e., how the cloud is located? Cloud can have any of the four types of access: Public, Private, Hybrid, and Community.



Public Cloud
The public cloud allows systems and services to be easily accessible to the general public. Public cloud may be less secure because of its openness.

Private Cloud
The private cloud allows systems and services to be accessible within an organization. It is more secured because of its private nature.

Community Cloud
The community cloud allows systems and services to be accessible by a group of organizations.

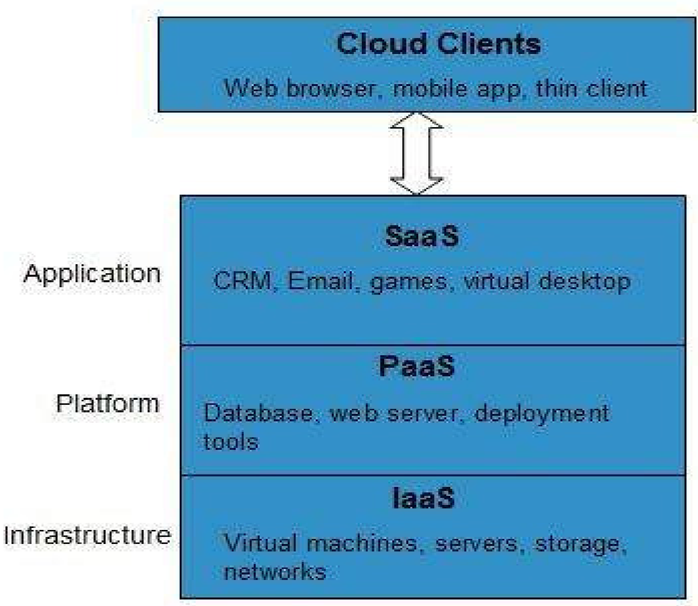
Hybrid Cloud
The hybrid cloud is a mixture of public and private cloud, in which the critical activities are performed using private cloud while the non-critical activities are performed using public cloud.

Service Models
Cloud computing is based on service models. These are categorized into three basic service models which are -

- Infrastructure-as–a-Service (IaaS)
- Platform-as-a-Service (PaaS)
- Software-as-a-Service (SaaS)

Anything-as-a-Service (XaaS) is yet another service model, which includes Network-as-a-Service, Business-as-a-Service, Identity-as-a-Service, Database-as-a-Service or Strategy-as-a-Service.

The **Infrastructure-as-a-Service (IaaS)** is the most basic level of service. Each of the service models inherit the security and management mechanism from the underlying model, as shown in the following diagram:



Artha Sarathi

Infrastructure-as-a-Service (IaaS)

IaaS provides access to fundamental resources such as physical machines, virtual machines, virtual storage, etc.

Apart from these resources, the IaaS also offers:

- Virtual machine disk storage
- Virtual local area network (VLANs)
- Load balancers
- IP addresses
- Software bundles

All of the above resources are made available to end user via **server virtualization.** Moreover, these resources are accessed by the customers as if they own them.

Benefits

IaaS allows the cloud provider to freely locate the infrastructure over the Internet in a cost-effective manner. Some of the key benefits of IaaS are listed below:

- Full control of the computing resources through administrative access to VMs.
- Flexible and efficient renting of computer hardware.
- Portability, interoperability with legacy applications.

Platform-as-a-Service (PaaS)

PaaS offers the runtime environment for applications. It also offers development and deployment tools required to develop applications. PaaS has a feature of **point-and-click** tools that enables non-developers to create web applications.

App Engine of Google and **Force.com** are examples of PaaS offering vendors. Developer may log on to these websites and use the **built-in API** to create web-based applications.

But the disadvantage of using PaaS is that, the developer **locks-in** with a particular vendor. For example, an application written in Python against API of Google, and using App Engine of Google is likely to work only in that environment.

Characteristics

Here are the characteristics of PaaS service model:

- PaaS offers **browser based development environment.** It allows the developer to create database and edit the application code either via Application Programming Interface or point-and-click tools.
- PaaS provides **built-in security, scalability,** and **web service interfaces.**
- PaaS provides built-in tools for defining **workflow, approval processes,** and business rules.
- It is easy to integrate PaaS with other applications on the same platform.
- PaaS also provides web services interfaces that allow us to connect the applications outside the platform.

Software-as-a-Service (SaaS)

SaaS model model allows to provide software application as a service to the end users. It refers to a software that is deployed on a host service and is accessible via Internet. There are several SaaS applications listed below:

- Billing and invoicing system
- Customer Relationship Management (CRM) applications
- Help desk applications
- Human Resource (HR) solutions

Some of the SaaS applications are not customizable such as **Microsoft Office Suite.** But SaaS provides us **Application Programming Interface (API),** which allows the developer to develop a customized application.

Characteristics

Here are the characteristics of SaaS service model:

- SaaS makes the software available over the Internet.
- The software applications are maintained by the vendor.
- The license to the software may be subscription based or usage based. And it is billed on recurring basis.
- SaaS applications are cost-effective since they do not require any maintenance at end user side.
- They are available on demand.
- They can be scaled up or down on demand.

Partha Sarathi

- They are automatically upgraded and updated.
- SaaS offers shared data model. Therefore, multiple users can share single instance of infrastructure. It is not required to hard code the functionality for individual users.
- All users run the same version of the software.

Benefits

Using SaaS has proved to be beneficial in terms of scalability, efficiency and performance. Some of the benefits are listed below:

- Modest software tools
- Efficient use of software licenses
- Centralized management and data
- Platform responsibilities managed by provider
- Multitenant solutions

Benefits

Cloud Computing has numerous advantages. Some of them are listed below -

- One can access applications as utilities, over the Internet.
- One can manipulate and configure the applications online at any time.
- It does not require to install a software to access or manipulate cloud application.
- Cloud Computing offers online development and deployment tools, programming runtime environment through PaaS model.
- Cloud resources are available over the network in a manner that provide platform independent access to any type of clients.
- Cloud Computing offers on-demand self-service. The resources can be used without interaction with cloud service provider.
- Cloud Computing is highly cost effective because it operates at high efficiency with optimum utilization. It just requires an Internet connection
- Cloud Computing offers load balancing that makes it more reliable.

Identity-as–a-Service (IDaaS).

IDaaS offers management of identity information as a digital entity. This identity can be used during electronic transactions.

Identity

Identity refers to set of attributes associated with something to make it recognizable. All objects may have same attributes, but their identities cannot be the same. A unique identity is assigned through unique identification attribute.

There are several **identity services** that are deployed to validate services such as validating web sites, transactions, transaction participants, client, etc. Identity-as-a-Service may include the following:

- Directory services
- Federated services
- Registration
- Authentication services
- Risk and event monitoring
- Single sign-on services
- Identity and profile management

Network-as-a-Service (NaaS)

NaaS allows us to access to network infrastructure directly and securely. NaaS makes it possible to deploy **custom routing protocols.**

NaaS uses **virtualized network infrastructure** to provide network services to the customer. It is the responsibility of NaaS provider to maintain and manage the network resources. Having a provider working for a customer decreases the workload of the customer. Moreover, NaaS offers **network as a utility.** NaaS is also based on **pay-per-use model.**

Partha Sarathi

Risks related to Cloud Computing
Although cloud Computing is a promising innovation with various benefits in the world of computing, it comes with risks. Some of them are discussed below:

Security and Privacy
It is the biggest concern about cloud computing. Since data management and infrastructure management in cloud is provided by third-party, it is always a risk to handover the sensitive information to cloud service providers. Although the cloud computing vendors ensure highly secured password protected accounts, any sign of security breach may result in loss of customers and businesses.

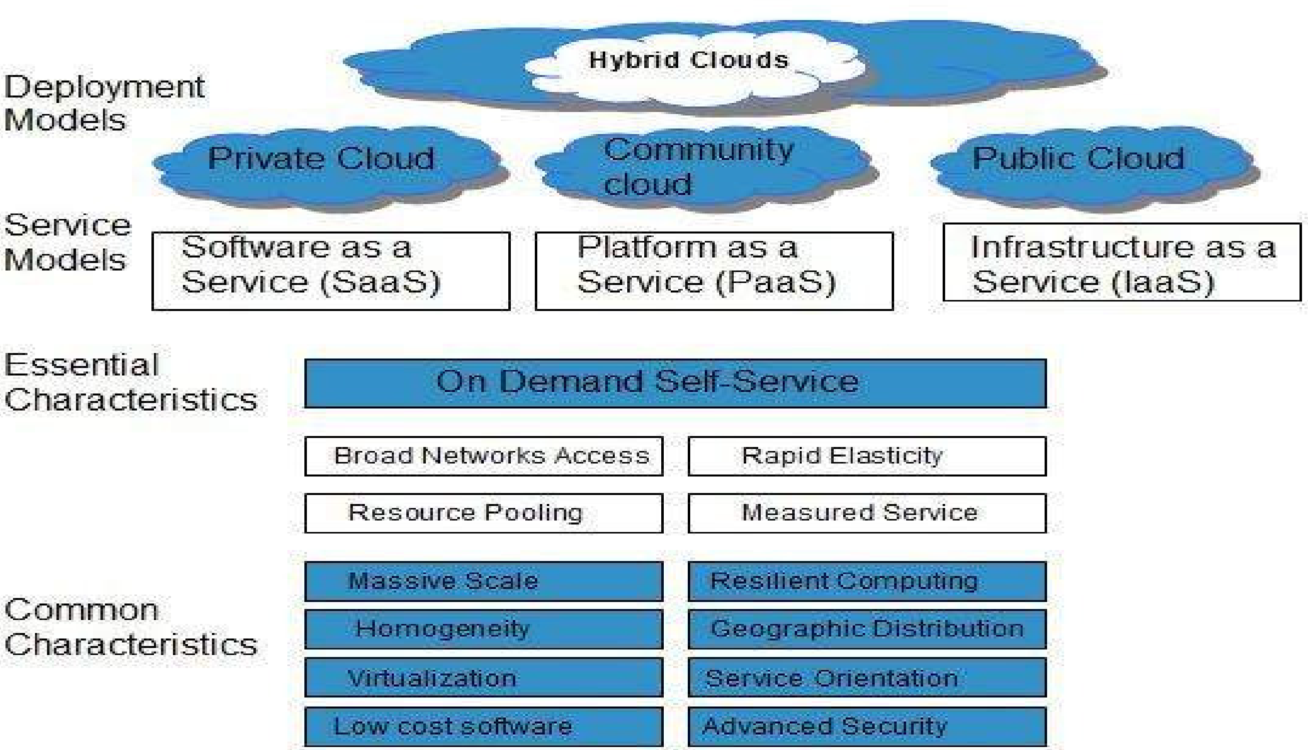
Lock In
It is very difficult for the customers to switch from one Cloud Service Provider (CSP) to another. It results in dependency on a particular CSP for service.

Isolation Failure
This risk involves the failure of isolation mechanism that separates storage, memory, and routing between the different tenants.

Management Interface Compromise
In case of public cloud provider, the customer management interfaces are accessible through the Internet.

Insecure or Incomplete Data Deletion
It is possible that the data requested for deletion may not get deleted. It happens because either of the following reasons
- Extra copies of data are stored but are not available at the time of deletion
- Disk that stores data of multiple tenants is destroyed.

Characteristics of Cloud Computing
There are four key characteristics of cloud computing. They are shown in the following diagram:



On Demand Self Service
Cloud Computing allows the users to use web services and resources on demand. One can logon to a website at any time and use them.

Broad Network Access
Since cloud computing is completely web based, it can be accessed from anywhere and at any time.

Resource Pooling
Cloud computing allows multiple tenants to share a pool of resources. One can share single physical instance of hardware, database and basic infrastructure.

Rapid Elasticity
It is very easy to scale the resources vertically or horizontally at any time. Scaling of resources means the ability of resources to deal with increasing or decreasing demand.
The resources being used by customers at any given point of time are automatically monitored.

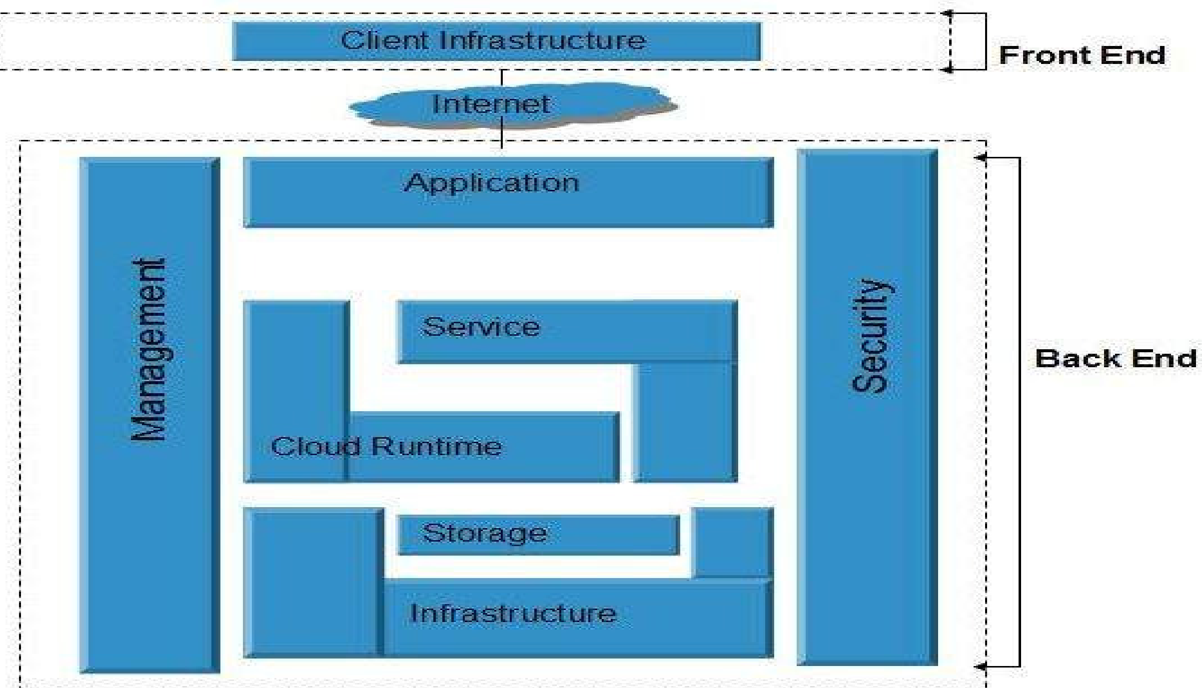
Measured Service
In this service cloud provider controls and monitors all the aspects of cloud service. Resource optimization, billing, and capacity planning etc. depend on it.

Cloud Computing Architecture:
Cloud computing architecture refers to the components and sub components required for cloud computing. These components typically refer to:

1. Front end(fat client, thin client)
2. Back end platforms(servers, storage)
3. Cloud based delivery and a network (Internet, Intranet, Inter cloud).

Front End
The **front end** refers to the client part of cloud computing system. It consists of interfaces and applications that are required to access the cloud computing platforms, Example - Web Browser.

Back End
The **back End** refers to the cloud itself. It consists of all the resources required to provide cloud computing services. It comprises of huge data storage, virtual machines, security mechanism, services, deployment models, servers, etc.



Note

- It is the responsibility of the back end to provide built-in security mechanism, traffic control and protocols.
- The server employs certain protocols known as middleware, which help the connected devices to communicate with each other.