Name of the Paper: Advanced Computer Architecture
Paper Code: CS802D
Contact (Periods/Week):3L/Week
Credit Point: 3
No. of Lectures: 35

Module – 1: Introduction to Advanced Computer Architectures [5L]

Different types of architectural classifications – instruction vs. data (SISD, SIMD, MISD, MIMD), serial vs. parallel, pipelining vs. parallelism; Pipelining: Definition, different types of pipelining, hazards in pipelining.
Concept of reservation tables, issue of multiple instructions with minimum average latency (MAL).

Module – 1: Introduction to Advanced Computer Architectures

LECTURE 1

Different types of architectural classifications:

- FLYNN'S TAXONOMY OF COMPUTER ARCHITECTURE: Flynn classification (1966) is based on multiplicity of instruction streams and the data streams in computer systems.
- FENG'S CLASSIFICATION : Feng's classification (1972) is based on serial versus parallel processing.
- Handler Classification : Handler's classification (1977) is determined by the degree of parallelism and pipelining in various subsystem levels.

Other types of architectural classification

- Classification based on coupling between processing elements
- Classification based on mode of accessing memory

Flynn's Classical Taxonomy:

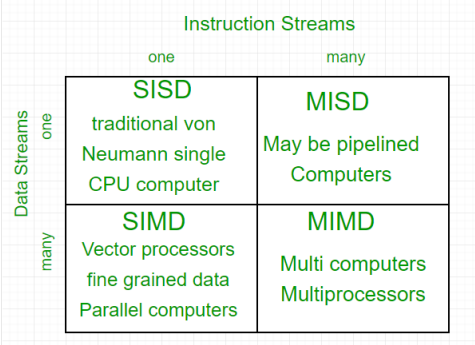


Figure 1.1

Among mentioned above the one widely used since 1966, is Flynn's Taxonomy. This taxonomy distinguishes multi-processor computer architectures according two independent dimensions of *Instruction stream* and *Data stream*. An instruction stream is sequence of instructions executed by machine. And a data stream is a sequence of data including input, partial or temporary results used by instruction stream. Each of these dimensions can have only one of two possible states: *Single* or *Multiple*. Flynn's classification depends on the distinction between the performance of control unit and the data processing unit rather than its operational and structural interconnections. Following are the four category of Flynn classification and characteristic feature of each of them.

1. Single instruction stream, single data stream (SISD)

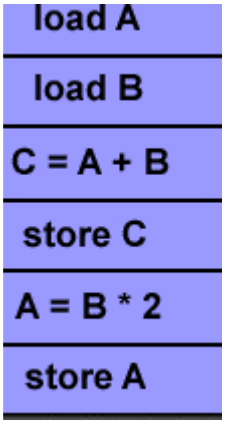


Figure 1.2 Execution of instruction in SISD processors

The figure 1.1 is represents an organization of simple SISD computer having one control unit, one processor unit and single memory unit.

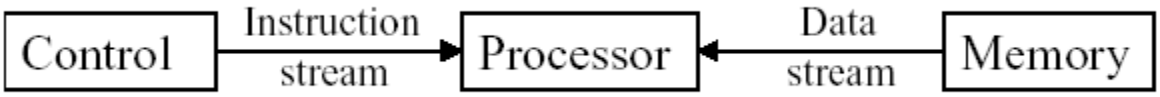


Figure 1.3 SISD processor organizations

- They are also called scalar processor i.e., one instruction at a time and each instruction have only one set of operands.
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle.
- Single data: only one data stream is being used as input during any one clock cycle.
- Deterministic execution.
- Instructions are executed sequentially.
- This is the oldest and until recently, the most prevalent form of computer.
- Examples: most PCs, single CPU workstations and mainframes.

b) Single instruction stream, multiple data stream (SIMD) processors

- A type of parallel computer.
- Single instruction: All processing units execute the same instruction issued by the control unit at any given clock cycle as shown in figure 5.4 where there are multiple processor executing instruction given by one control unit.

    - Multiple data: Each processing unit can operate on a different data element as shown if figure below the processor are connected to shared memory or interconnection network providing multiple data to processing unit.

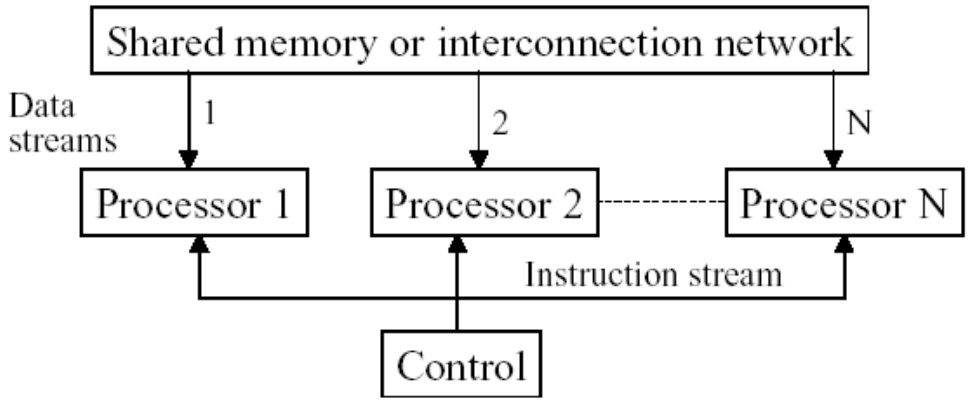


Figure 1.4 SIMD processor organizations

This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.

- Thus single instruction is executed by different processing unit on different set of data as shown in figure 1.4
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing and vector computation.
- Synchronous (lockstep) and deterministic execution.

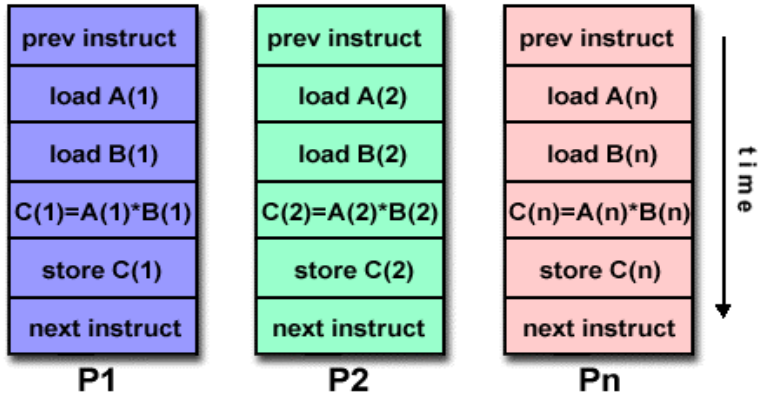


Figure 1.5 Execution of instructions in SIMD processors

c) Multiple instruction streams, single data stream (MISD)

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams as shown in figure 5.6 a single data stream is forwarded to different processing unit which are connected to different control unit and execute instruction given to it by control unit to which it is attached.

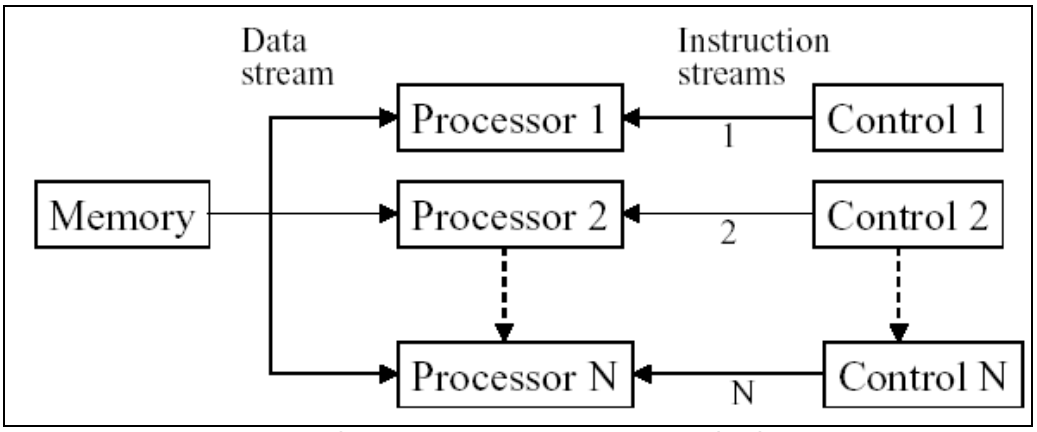


Figure 1.6 MISD processor organizations

□□Thus in these computers same data flow through a linear array of processors executing different instruction streams as shown in figure 1.6

- This architecture is also known as systolic arrays for pipelined execution of specific instructions.
- Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).
- Some conceivable uses might be:

1. Multiple frequency filters operating on a single signal stream
2. Multiple cryptography algorithms attempting to crack a single coded message.

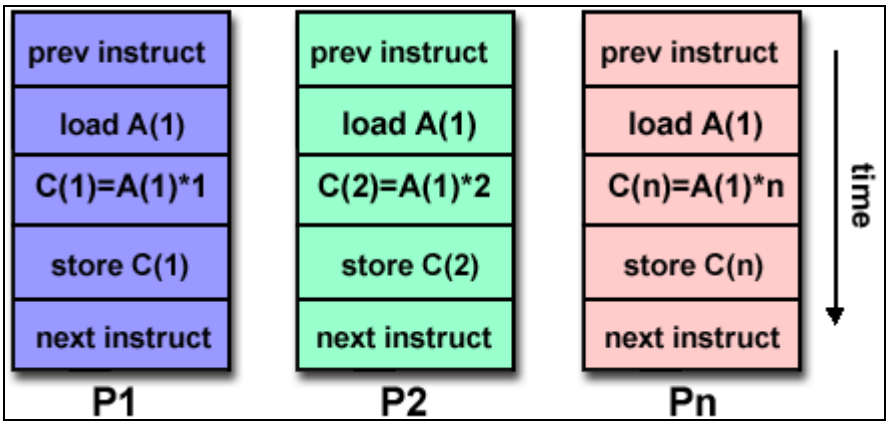


Figure 1.7 Execution of instructions in MISD processors

d) Multiple instruction stream, multiple data stream (MIMD)

- Multiple Instructions: every processor may be executing a different instruction stream
- Multiple Data: every processor may be working with a different data stream as shown in figure 5.8 multiple data stream is provided by shared memory.
- Can be categorized as loosely coupled or tightly coupled depending on sharing of data and control.
- Execution can be synchronous or asynchronous, deterministic or non-deterministic.

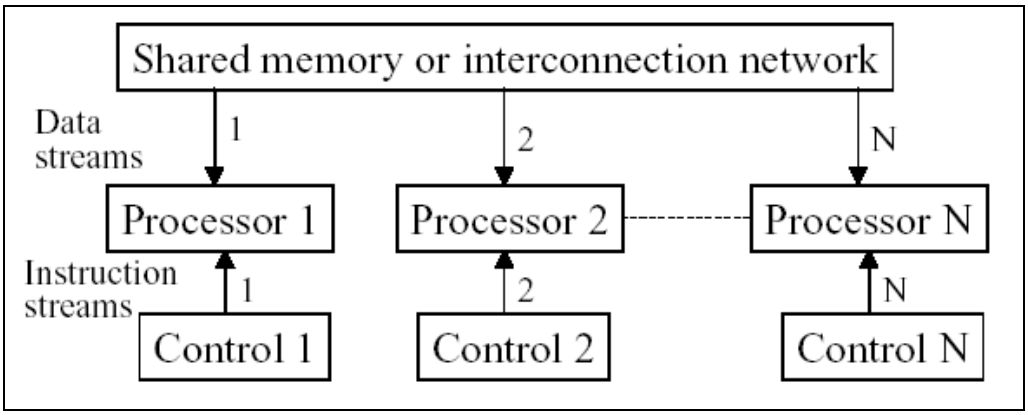


Figure 1.8 MIMD processor organizations

As shown in figure 5.8 there are different processor each processing different task.

- Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.

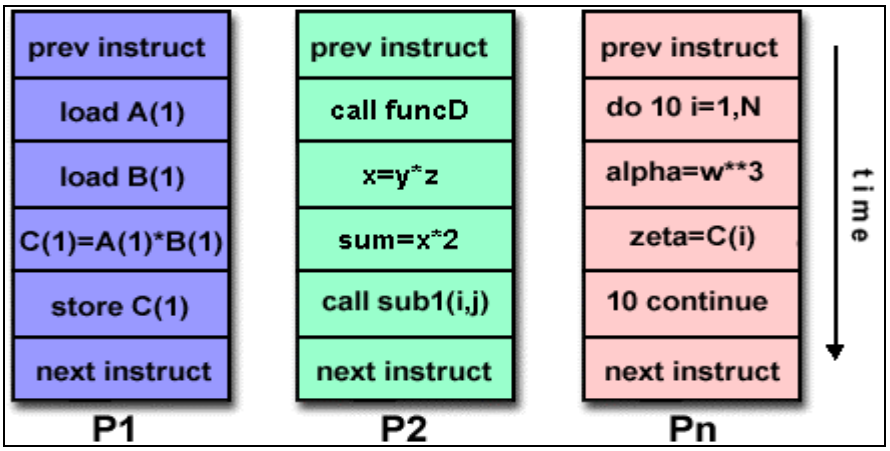


Figure 1.9 Execution of instructions MIMD processors

Here the some popular computer architecture and there types
SISD IBM 701, IBM 1620, IBM 7090, PDP VAX11/ 780
SISD (With multiple functional units) IBM360/91 (3); IBM 370/168 UP
SIMD (Word Slice Processing) Illiac – IV; PEPE
SIMD (Bit Slice processing) STARAN; MPP; DAP
MIMD (Loosely Coupled) IBM 370/168 MP; Univac 1100/80
MIMD (Tightly Coupled) Burroughs- D – 825

LECTURE 2

Introduction to Parallel Computing

Computer software were written conventionally for serial computing. This meant that to solve a problem, an algorithm divides the problem into smaller instructions. These discrete instructions are then executed on Central Processing Unit of a computer one by one. Only after one instruction is finished, next one starts.

Real life example of this would be people standing in a queue waiting for movie ticket and there is only cashier.Cashier is giving ticket one by one to the persons. Complexity of this situation increases when there are 2 queues and only one cashier.

So, in short Serial Computing is following:

1. In this, a problem statement is broken into discrete instructions.
2. Then the instructions are executed one by one.
3. Only one instruction is executed at any moment of time.

Look at point 3. This was causing a huge problem in computing industry as only one instruction was getting executed at any moment of time. This was a huge waste of hardware resources as only one part of the hardware will be running for a particular instruction and of time. As problem statements were getting heavier and bulkier, so does the amount of time in execution of those statements. Example of processors are Pentium 3 and Pentium 4.

Now let's come back to our real life problem. We could definitely say that complexity will decrease when there are 2 queues and 2 cashier giving tickets to 2 persons simultaneously. This is an example of Parallel Computing.

Parallel Computing –

It is the use of multiple processing elements simultaneously for solving any problem. Problems are broken down into instructions and are solved concurrently as each resource which has been applied to work is working at the same time.

Advantages of Parallel Computing over Serial Computing are as follows:

1. It saves time and money as many resources working together will reduce the time and cut potential costs.
2. It can be impractical to solve larger problems on Serial Computing.
3. It can take advantage of non-local resources when the local resources are finite.
4. Serial Computing 'wastes' the potential computing power, thus Parallel Computing makes better work of hardware.

Types of Parallelism:

1. **Bit-level parallelism:** It is the form of parallel computing which is based on the increasing processor's size. It reduces the number of instructions that the system must execute in order to perform a task on large-sized data.

   Example: Consider a scenario where an 8-bit processor must compute the sum of two 16-bit integers. It must first sum up the 8 lower-order bits, then add the 8 higher-order bits, thus requiring

two instructions to perform the operation. A 16-bit processor can perform the operation with just one instruction.

2. **Instruction-level parallelism:** A processor can only address less than one instruction for each clock cycle phase. These instructions can be re-ordered and grouped which are later on executed concurrently without affecting the result of the program. This is called instruction-level parallelism.
3. **Task Parallelism:** Task parallelism employs the decomposition of a task into subtasks and then allocating each of the subtasks for execution. The processors perform execution of sub tasks concurrently.

Why parallel computing?

- The whole real world runs in dynamic nature i.e. many things happen at a certain time but at different places concurrently. This data is extensively huge to manage.
- Real world data needs more dynamic simulation and modeling, and for achieving the same, parallel computing is the key.
- Parallel computing provides concurrency and saves time and money.
- Complex, large datasets, and their management can be organized only and only using parallel computing's approach.
- Ensures the effective utilization of the resources. The hardware is guaranteed to be used effectively whereas in serial computation only some part of hardware was used and the rest rendered idle.
- Also, it is impractical to implement real-time systems using serial computing.

Applications of Parallel Computing:

- Data bases and Data mining.
- Real time simulation of systems.
- Science and Engineering.
- Advanced graphics, augmented reality and virtual reality.

Limitations of Parallel Computing:

- It addresses such as communication and synchronization between multiple sub-tasks and processes which is difficult to achieve.
- The algorithms must be managed in such a way that they can be handled in the parallel mechanism.
- The algorithms or program must have low coupling and high cohesion. But it's difficult to create such programs.
- More technically skilled and expert programmers can code a parallelism based program well.

Future of Parallel Computing:

The computational graph has undergone a great transition from serial computing to parallel computing. Tech giant such as Intel has already taken a step towards parallel computing by employing multicore processors. Parallel computation will revolutionize the way computers work in the future, for the better good. With all the world connecting to each other even more than before, Parallel Computing does a better role in helping us stay that way? With faster networks, distributed systems, and multi-processor computers, it becomes even more necessary.

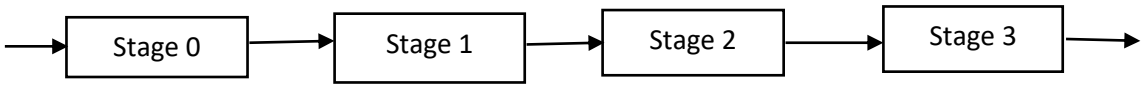
LECTURE 3

Pipelining

Pipelining is an implementation technique where multiple instructions are overlapped in execution. The computer pipeline is divided in stages. Each stage completes a part of an instruction in parallel. The stages are connected one to the next to form a pipe - instructions enter at one end, progress through the stages, and exit at the other end.

Pipelining does not decrease the time for individual instruction execution. Instead, it increases instruction throughput. The throughput of the instruction pipeline is determined by how often an instruction exits the pipeline.

Pipelining for instruction execution is similar to construction of factor assembly line for product manufacturing. The basic idea is to decompose the instruction execution process into a collection of smaller functions that can be independently performed by discrete subsystems in the processor implementation. An illustration of this decomposition into 4 parts is:



For pipelining, we will organized these discrete subsystems (which are called pipeline stages) implementing the instruction interpretation process into concurrently executing systems each operating on distinct instructions in the instruction stream (much like a factory assembly line).

Typical Non Pipelined Execution

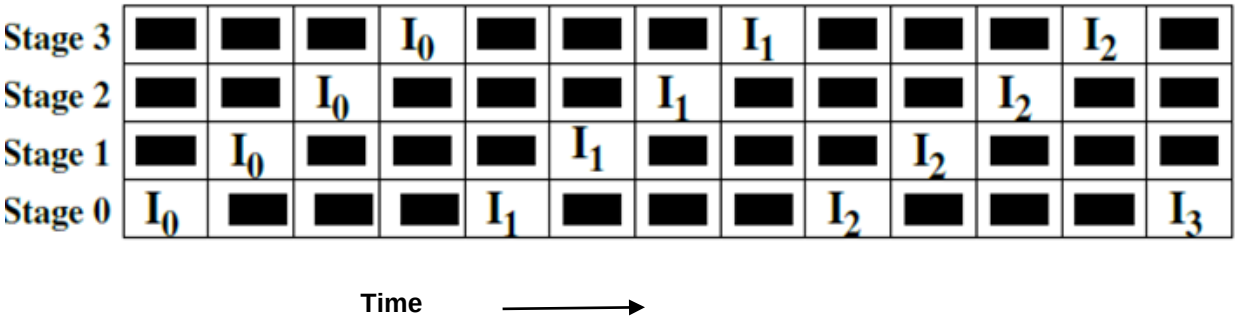


Figure 1.10Idealized Pipeline Executions

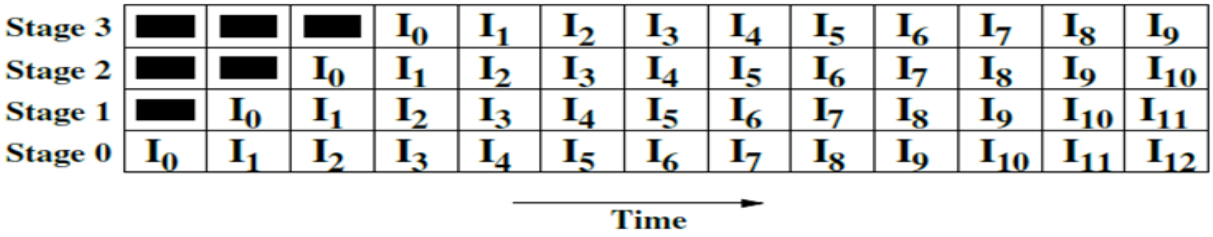


Figure 1.11 Actual Pipeline Executions

Time to execute n instructions: (3+n)t.

Steady state : $\frac{Clock\ cycles\ for\ unpipelined\ execution}{Pipeline\ depth}$

Speedup, Efficiency and Throughput

Ideally, a linear pipeline ok k stages can process n tasks in *k + (n+1)* clock cycles , where k cycles are needed to complete the execution of the very first task and the remaining n-1 tasks require n-1 cycles. Thus the total time required is:

$T_k = [k + (n-1)]\tau$

where τ is the clock period. Consider an equivalent function non-pipelined processor which has a *flow-through delay* of $k\tau$. The amount of time it takes to execute n tasks on this non pipelined processor is $T_1 = nk\tau$.

Speedup Factor

The speedup factor of a k-stage pipeline over an equivalent non-pipelined processor is defined as:

$$S_k = \frac{T_1}{T_K} = \frac{nk\tau}{k\tau + (n-1)\tau} = \frac{nk}{k + (n-1)}$$

Efficiency and Throughput

The efficiency E_k of a linear k-stage pipeline is defined as

$$E_k = \frac{Sk}{k} = \frac{n}{k + (n-1)}$$

Obviously, the efficiency approaches 1 when $n \rightarrow \infty$, and a lower bound on E_k is 1/k when n = 1. The pipeline throughput H_k is defined as the number of tasks (operations) per unit time :

$$H_k = \frac{n}{[k + (n-1)]\tau} = \frac{nf}{k + (n-1)}$$

The maximum throughput f occurs when $Ek \rightarrow 1$ and $n \rightarrow \infty$. This coincides with the speedup definition given in chapter 3. Note that $H_k = E_k \cdot f = E_k/\tau = S_k/k\tau$.

Consider the numerical example,
let the time it takes to process a sub-operation in each segment be equal to t_p = 20 ns. Assume that the pipeline has k = 4 segments and execute n = 100 tasks in sequence. The pipeline system will take $(k + n - 1)t_p = (4+ 99)x20 = 2060$ ns to complete.

Assuming that $t_n = kt_p = 4 \times 20 = 80$ ns,

a non-pipeline system requires $nkt_p = 100 \times 80 = 8000$ ns to complete the 100 tasks. The speedup ratio is equal to 8000/2060 = 3.88. As the number of tasks increases, the speedup will approach 4, which is equal to the number of segments in the pipeline. If we assume that t_n = 60 ns, the speedup becomes 60/20 = 3.

LECTURE 4

Linear vs Non-Linear,Static Vs Dynamic Vs Unifunction Vs Multifunction Pileline

A linear pipelining is a series of processing stages and memory access.

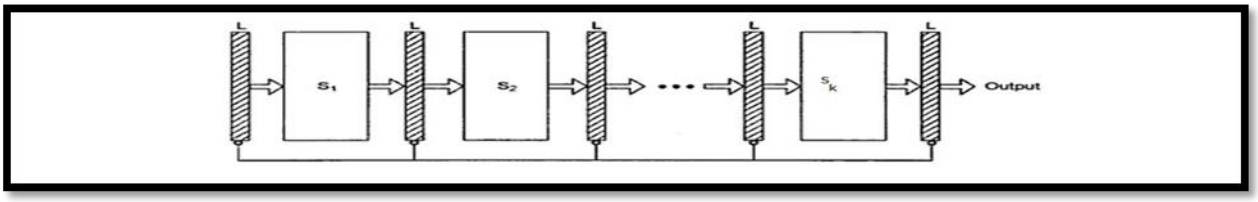


Figure 1.12 Linear Pipeline

A non linear pipelining can be configured to perform various functions at different times. In a dynamic pipeline there is also feed forward or feedback connection. Non-linear pipeline also allows very long instruction words.

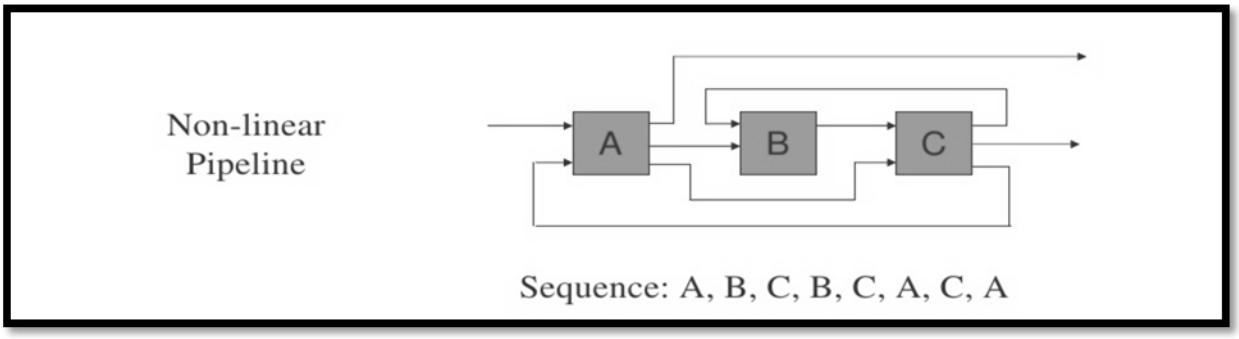


Figure 1.13 Linear Pipeline

Linear Pipeline	Non-Linear Pipeline
Linear pipeline are static pipeline because they are used to perform fixed functions.	Non-Linear pipeline are dynamic pipeline because they can be reconfigured to perform variable functions at different times.
Linear pipeline allows only streamline connections.	Non-Linear pipeline allows feed-forward and feedback connections in addition to the streamline connection.
It is relatively easy to partition a given function into a sequence of linearly ordered sub functions.	Function partitioning is relatively difficult because the pipeline stages are interconnected with loops in addition to streamline connections.
The Output of the pipeline is produced from the last stage.	The Output of the pipeline is not necessarily produced from the last stage.

The reservation table is trivial in the sense that data flows in linear streamline.	The reservation table is non-trivial in the sense that there is no linear streamline for data flows.
Static pipelining is specified by single Reservation table.	Dynamic pipelining is specified by more than one Reservation table.
All initiations to a static pipeline use the same reservation table.	A dynamic pipeline may allow different initiations to follow a mix of reservation tables.

There are two types of pipelines: Static and Dynamic. A static pipeline can perform only one function at a time whereas a dynamic pipeline can perform more than one function at a time.

Static pipelining - it is composition of stages one after another means that the output of one stage is become input to the next stage we also called it linear pipelining. it is further divided in two types synchronous and asynchronous.

Dynamic pipelining- in it stages are connected in a liner fashion but this kind of pipelining used feed forward and feed backward connections as a input to the stages. It performs variable function but static perform fixed functions. In dynamic pipelining we can take intermediate outputs.

Static	Dynamic
It may assume only one functional configuration at a time	It permits several functional configurations to exist simultaneously
It can be either unifunctional or multifunctional	A dynamic pipeline must be multi-functional
Static pipelines are preferred when instructions of same type are to be executed continuously	The dynamic configuration requires more elaborate control and sequencing mechanisms than static pipelining

A pipeline unit with a fixed and dedicated function is called **unifunctional.**

A **multifunction pipe** may perform different functions, either at different times or at the same time.

Unifunctional Pipelines	**Multifunctional Pipelines**
A pipeline unit with fixed and dedicated function is called unifunctional.	A multifunction pipe may perform different functions either at different times or same time, by interconnecting different subset of stages in pipeline.
It has 12 unifunctional pipelines described in four groups: – Address Functional Units: • Address Add Unit • Address Multiply Unit	It has – one instruction processing unit – four memory buffer units and – four arithmetic units.
Example: CRAY1 (Supercomputer - 1976)	Example 4X-TI-ASC (Supercomputer - 1973)

Instruction Pipeline

A stream of instructions can be executed by a pipeline in an overlapped manner.

The Instruction Cycle is given below:

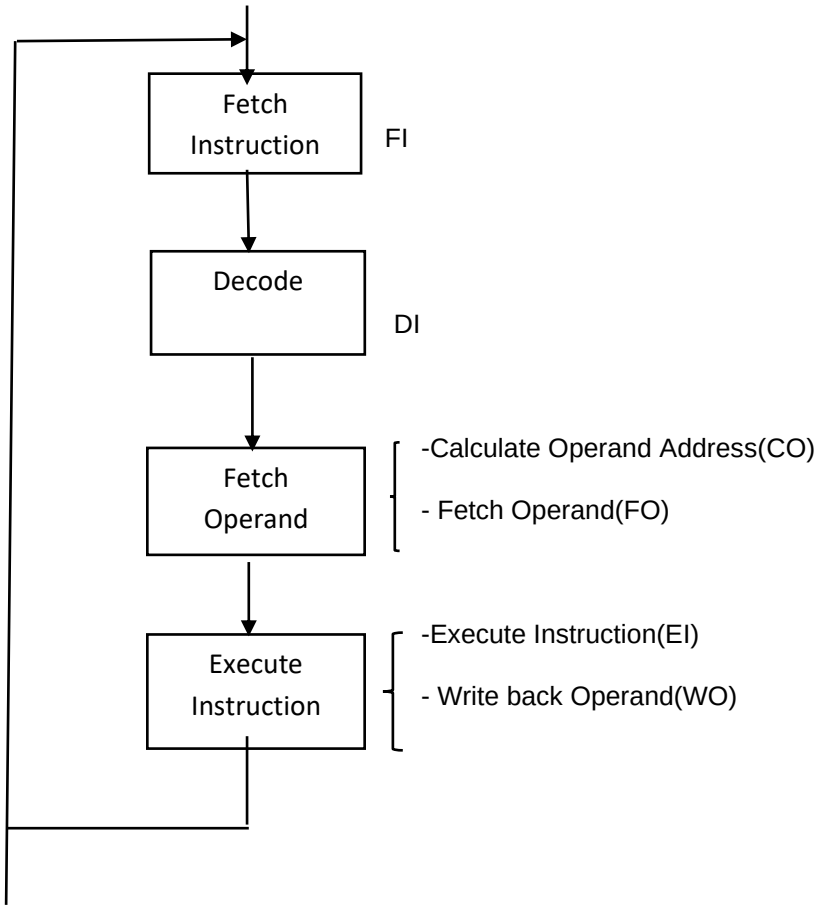


Figure 1.14Instruction Cycle

Instruction execution is extremely complex and involves several operations which are executed successively. This implies a large amount of hardware, but only one part of this hardware works at a given moment.

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. This is solved without additional hardware but only by letting different parts of the hardware work for different instructions at the same time.

The pipeline organization of a CPU is similar to an assembly line: the work to be done in an instruction is broken into smaller steps (pieces), each of which takes a fraction of the time needed to complete the entire instruction. Each of these steps is a pipe stage (or a pipe segment).
The time required to execute a stage and move to the next is called a *machine cycle* (this is one or several clock cycles). The execution of one instruction takes several machine cycles as it passes through the pipeline.

The Four Segment Pipelining:

Four segment pipeline:
FI: fetch instruction
DA: decode instruction
FO: fetch operand
EX: execute instruction

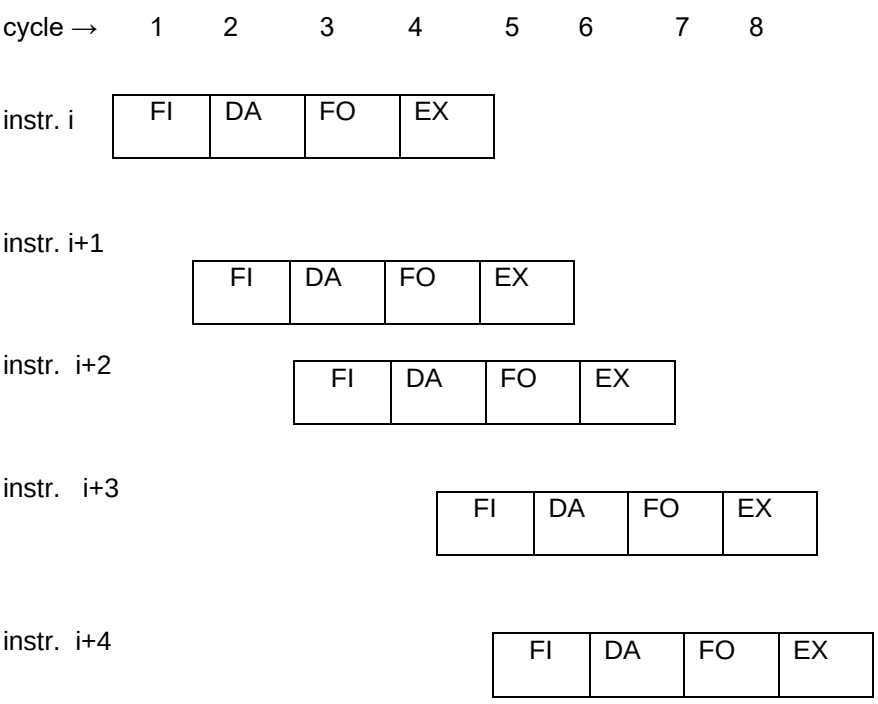


Figure 1.15 Pipelining by four Segments

Acceleration by Pipelining Six Segments:

Six stage pipeline:
FI: fetch instruction FO: fetch operand
DI: decode instruction EI: execute instruction
CO: calculate operand address WO:write operand

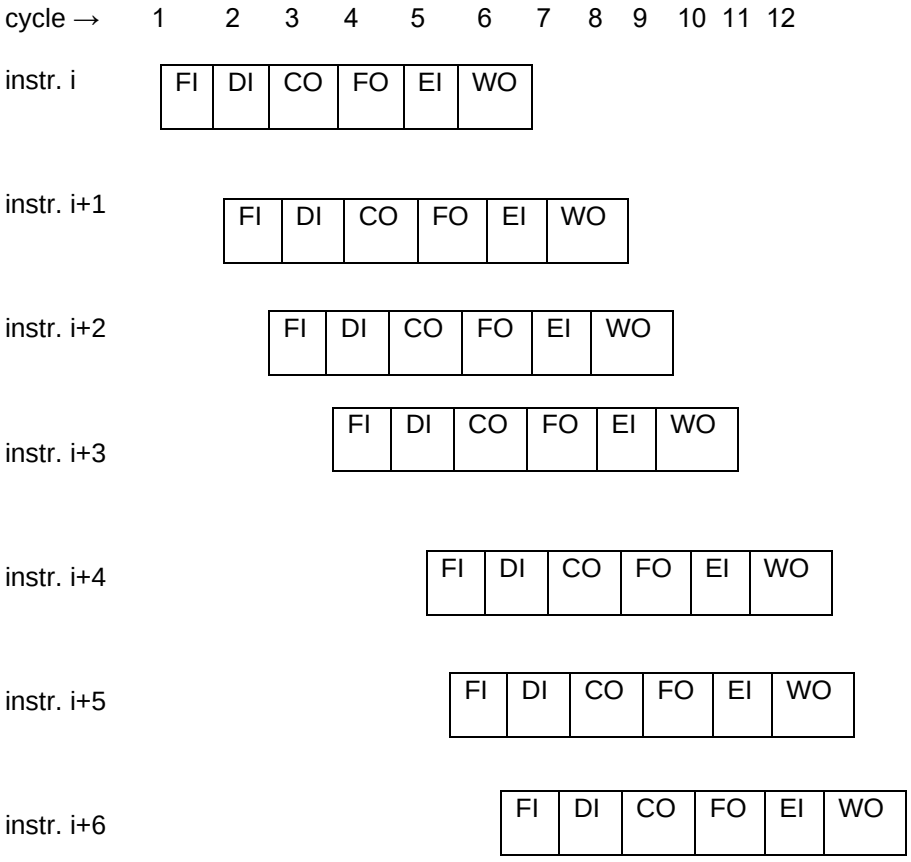


Figure 1.16 Pipelining by Six Segments

Execution time for the 7 instructions, with pipelining:

$$(T_{ex}/6)\times 12 = 2\times T_{ex}$$

• Acceleration: $7\times T_{ex} / 2\times T_{ex} = 7/2$

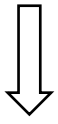
After a certain time (N-1 cycles) all the N stages of the pipeline are working: the pipeline is filled. Now, *theoretically*, the pipeline works providing maximal parallelism (N instructions are active simultaneously).

• τ: duration of one machine cycle
• *n*: number of instructions to execute
• *k*: number of pipeline stages
• $T_{k,n}$: total time to execute *n* instructions on a pipeline with *k* stages
• $S_{k,n}$: (theoretical) speedup produced by a pipeline with *k* stages when executing *n* instructions

$$T_{k,n} = [k + (n-1)] \times \tau$$

- The first instruction takes $k \times \tau$ to finish
- The following $n - 1$ instructions produce one result per cycle.

On a non-pipelined processor each instruction takes $k \times \tau$, and *n* instructions: $T_n = n \times k \times \tau$



$$S_{k,n} = \frac{Tn}{Tk,n} = \frac{nXkXt}{[k + (n-1)] \times t} = \frac{nXk}{k+(n-1)}$$

For large number of instructions (n → ∞) the speedup approaches *k* (number of stages).

• Apparently a greater number of stages always provides better performance. However:

- a greater number of stages increases the overhead in moving information between stages and synchronization between stages.
- with the number of stages the complexity of the CPU grows.
- it is difficult to keep a large pipeline at maximum rate because of *pipeline hazards*.

ARITHMETIC PIPELINE

Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating point operations. We will now discuss the pipeline unit for the floating point addition and subtraction.

The inputs to floating point adder pipeline are two normalized floating point numbers.

$$X = A \times 2^a$$
$$Y = B \times 2^b$$

A and B are mantissas and a and b are the exponents.

The floating point addition and subtraction can be performed in four segments.

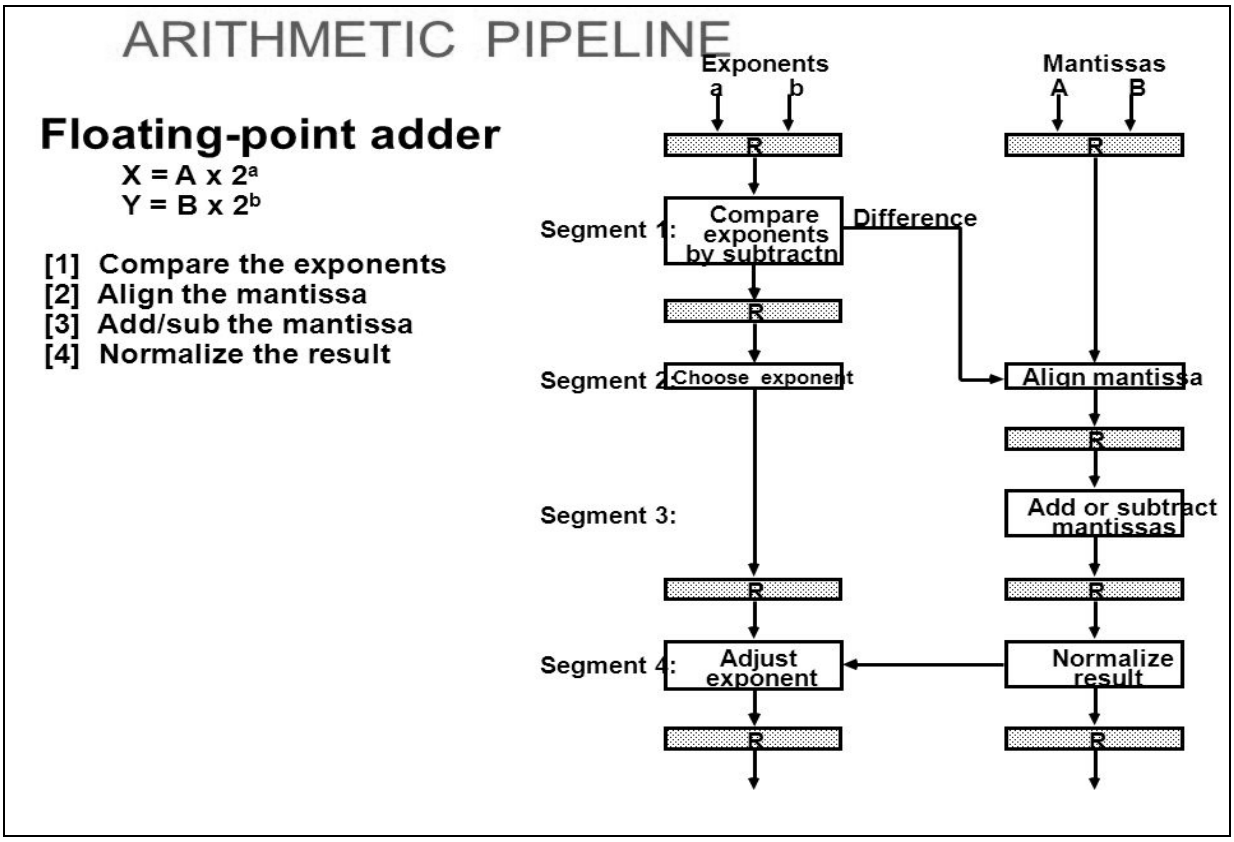


Figure 1.17 Arithmetic Pipeline

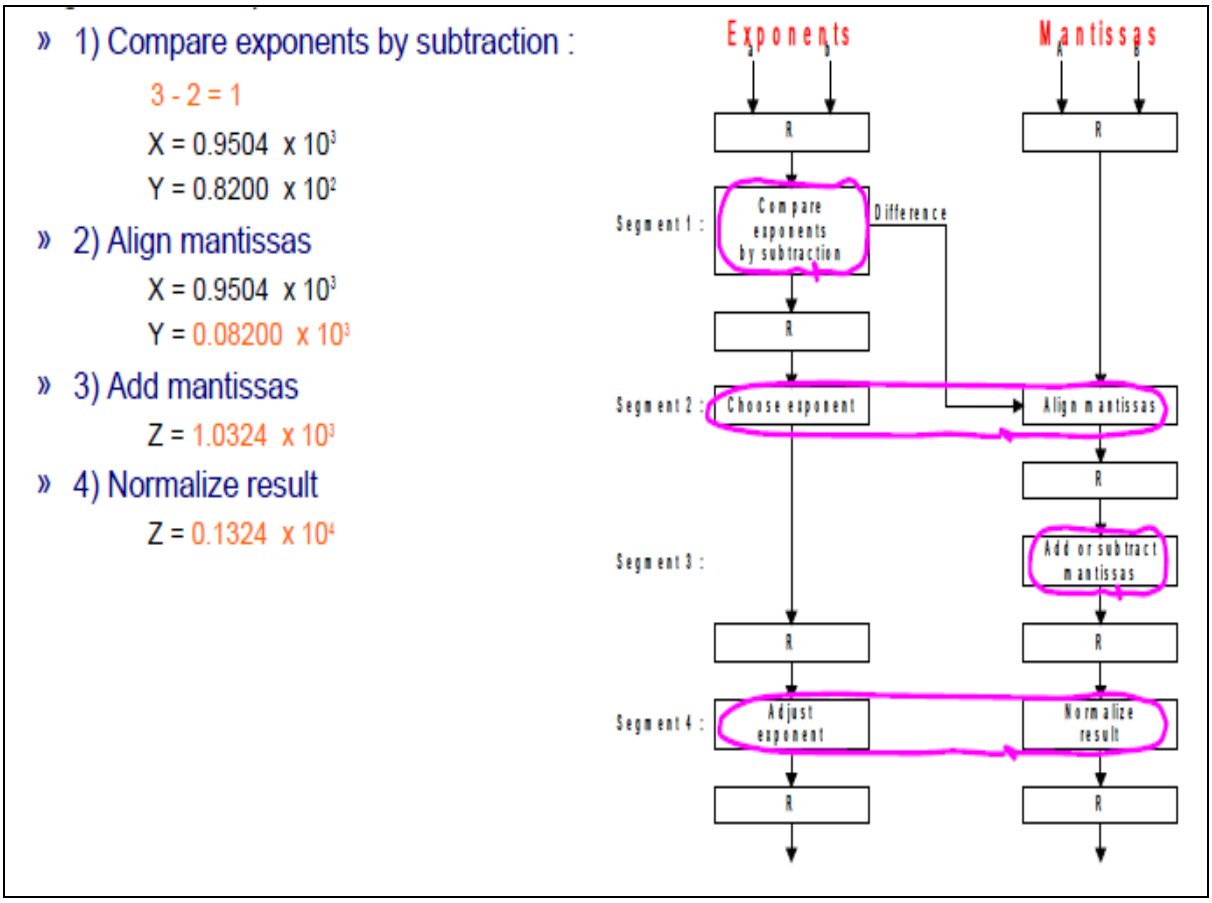


Figure 1.18 Operation on Pipeline segments

LECTURE 5

PIPELINE HAZARDS

There are situations, called **hazards**, that prevent the next instruction in the instruction stream from being executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining.

There are three classes of hazards:

1.Structural Hazards. They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
2.Data Hazards. They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
3.Control Hazards. They arise from the pipelining of branches and other instructions that change the PC.

Hazards in pipelines can make it necessary to stall the pipeline. The processor can stall on different events:

A ***cache miss***. A cache miss stalls all the instructions on pipeline both before and after the instruction causing the miss.
A ***hazard in pipeline.*** Eliminating a hazard often requires that some instructions in the pipeline to be allowed to proceed while others are delayed. When the instruction is stalled, all the instructions issued *later* than the stalled instruction are also stalled. Instructions issued *earlier* than the stalled instruction must continue, since otherwise the hazard will never clear.

HAZARDS

Data hazards

Data hazards occur when instructions that exhibit data dependence modify data in different stages of a pipeline. Ignoring potential data hazards can result in race conditions (also termed race hazards). There are three situations in which a data hazard can occur:

1. read after write (RAW), a *true dependency*
2. write after read (WAR), an *anti-dependency*
3. write after write (WAW), an *output dependency*

Consider two instructions i1 and i2, with i1 occurring before i2 in program order.

Read after write (RAW)

(i2 tries to read a source before i1 writes to it) A read after write (RAW) data hazard refers to a situation where an instruction refers to a result that has not yet been calculated or retrieved. This can occur because even though an instruction is executed after a prior instruction, the prior instruction has been processed only partly through the pipeline.

Example

For example:

i1. **R2**<- R1 + R3
i2. R4 <- **R2** + R3

The first instruction is calculating a value to be saved in register R2, and the second is going to use this value to compute a result for register R4. However, in a pipeline, when operands are fetched for the 2nd operation, the results from the first will not yet have been saved, and hence a data dependency occurs.

A data dependency occurs with instruction i2, as it is dependent on the completion of instruction i1.

Write after read (WAR)

(i2 tries to write a destination before it is read by i1) A write after read (WAR) data hazard represents a problem with concurrent execution.

Example

For example:

i1. R4 <- R1 + **R5**
i2. **R5**<- R1 + R2

In any situation with a chance that i2 may finish before i1 (i.e., with concurrent execution), it must be ensured that the result of register R5 is not stored before i1 has had a chance to fetch the operands.

Write after write (WAW)

(i2 tries to write an operand before it is written by i1) A write after write (WAW) data hazard may occur in a concurrent execution environment.

Example

For example:

i1. **R2**<- R4 + R7
i2. **R2**<- R1 + R3

The write back (WB) of i2 must be delayed until i1 finishes executing.

Structural hazards:

A structural hazard occurs when a part of the processor's hardware is needed by two or more instructions at the same time. A canonical example is a single memory unit that is accessed both in the fetch stage where an instruction is retrieved from memory, and the memory stage where data is written and/or read from memory. They can often be resolved by separating the component into orthogonal units (such as separate caches) or bubbling the pipeline.

A structural hazard would for example result from memory access of instruction fetch and memory access of data, were it not for separate data and instruction caches:

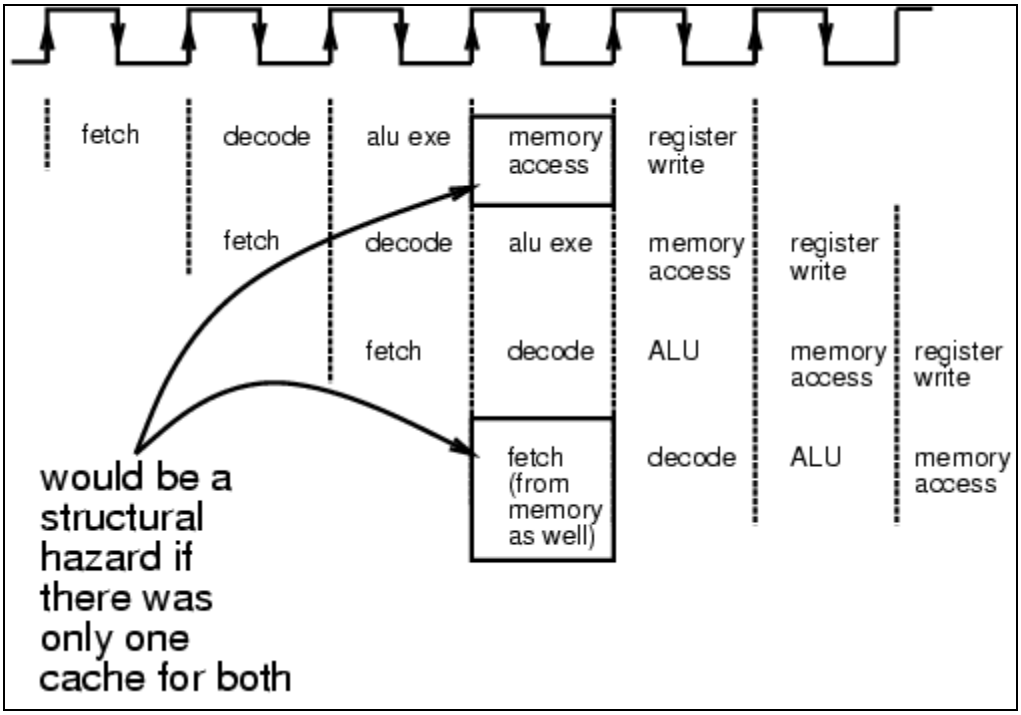


Figure 1.19 Structural hazards due to instruction fetch and memory access of data

Another example of a structural hazard is when decoding (setting up input registers) makes reference to same register as a register write:

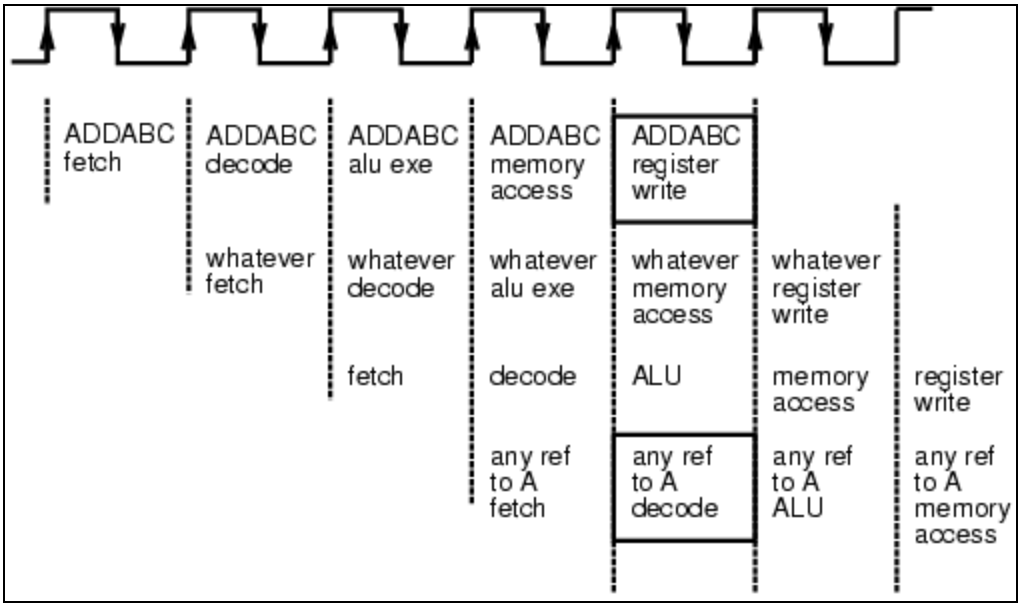


Figure 1.20 Structural hazards due to reference to same register as a register write

Control hazards (branch hazards):

Branching hazards (also termed control hazards) occur with branches. On many instruction pipeline micro architectures, the processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline (normally the *fetch* stage).

To avoid control hazards micro architectures can:

- insert a pipeline bubble (discussed above), guaranteed to increase latency, or
- use branch prediction and essentially make educated guesses about which instructions to insert, in which case a pipeline bubble will only be needed in the case of an incorrect prediction

Pipeline bubble or Pipeline Stall:

Bubbling the pipeline, also termed a pipeline break or pipeline stall, is a method to preclude data, structural, and branch hazards. As instructions are fetched, control logic determines whether a hazard could/will occur. If this is true, then the control logic inserts no operations (NOPs) into the pipeline. Thus, before the next instruction (which would cause the hazard) executes, the prior one will have had sufficient time to finish and prevent the hazard. If the number of NOPs equals the number of stages in the pipeline, the processor has been cleared of all instructions and can proceed free from hazards. All forms of stalling introduce a delay before the processor can resume execution.

Flushing the pipeline occurs when a branch instruction jumps to a new memory location, invalidating all prior stages in the pipeline. These prior stages are cleared, allowing the pipeline to continue at the new instruction indicated by the branch.

In computing, a **bubble** or **pipeline stall** is a delay in execution of an instruction in an instruction pipeline in order to resolve a hazard.

During the decoding stage, the control unit will determine if the decoded instruction reads from a register that the instruction currently in the execution stage writes to. If this condition holds, the control unit will stall the instruction by one clock cycle. It also stalls the instruction in the fetch stage, to prevent the instruction in that stage from being overwritten by the next instruction in the program.

To prevent new instructions from being fetched when an instruction in the decoding stage has been stalled, the value in the PC register and the instruction in the fetch stage are preserved to prevent changes. The values are preserved until the bubble has passed through the execution stage.

The execution stage of the pipeline must always be performing an action. A bubble is represented in the execution stage as a NOP instruction, which has no effect other than to stall the instructions being executed in the pipeline.

Timeline

The following is two executions of the same four instructions through a 4-stage pipeline but, for whatever reason, a delay in fetching of the purple instruction in cycle #2 leads to a bubble being created delaying all instructions after it as well.

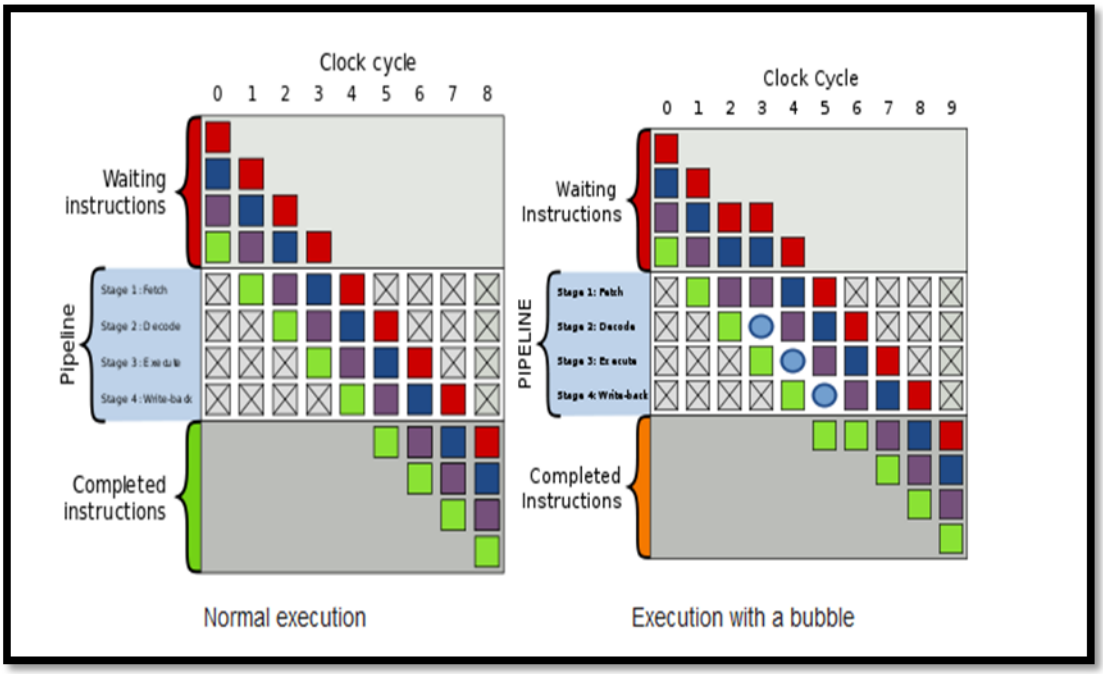


Figure 1.21 Bubbles in Pipeline

Branch Prediction:

In computer science, predication is an architectural feature that provides an alternative to conditional branch instructions. Predication works by executing instructions from both paths of the branch and only permitting those instructions from the taken path to modify architectural state. The instructions from the taken path are permitted to modify architectural state because they have been associated (predicated) with a predicate, a Boolean value used by the instruction to control whether the instruction is allowed to modify the architectural state or not.

Most computer programs contain conditional code, which will be executed only under specific conditions depending on factors that cannot be determined beforehand, for example depending on user input. As the majority of processors simply execute the next instruction in a sequence, the traditional solution is to insert branch instructions that allow a program to conditionally branch to a different section of code, thus changing the next step in the sequence. This was sufficient until designers began improving performance by implementing instruction pipelining, a method which is slowed down by branches. For a more thorough description of the problems which arose, and a popular solution, see branch predictor.

Luckily, one of the more common patterns of code that normally relies on branching has a more elegant solution. Consider the following pseudo code:

if condition

do this

else

do that

On a system that uses conditional branching, this might translate to machine instructions looking similar to:[1]

branch if condition to label 1

do that

branch to label 2

label 1:

do this

label 2:

...

With predication, all possible branch paths are coded inline, but some instructions execute while others do not. The basic idea is that each instruction is associated with a predicate (the word here used similarly to its usage in predicate logic) and that the instruction will only be executed if the predicate is true. The machine code for the above example using predication might look something like this:

(condition) do this

(not condition) do that
Note that beside eliminating branches, less code is needed in total, provided the architecture provides predicated instructions. While this does not guarantee faster execution in general, it will if the do this and do that blocks of code are short enough.

Predication's simplest form is partial predication, where the architecture has conditional move or conditional select instructions. Conditional move instructions write the contents of one register over another only if the predicate's value is true, whereas conditional select instructions choose which of two registers has its contents written to a third based on the predicate's value. A more generalized and capable form is full predication. Full predication has a set of predicate registers for storing predicates (which allows multiple nested or sequential branches to be simultaneously eliminated) and most instructions in the architecture have a register specifier field to specify which predicate register supplies the predicate.

Advantages:

The main purpose of predication is to avoid jumps over very small sections of program code, increasing the effectiveness of pipelined execution and avoiding problems with the cache. It also has a number of more subtle benefits:

- Functions that are traditionally computed using simple arithmetic and bitwise operations may be quicker to compute using predicated instructions.
- Predicated instructions with different predicates can be mixed with each other and with unconditional code, allowing better instruction scheduling and so even better performance.

- Elimination of unnecessary branch instructions can make the execution of necessary branches, such as those that make up loops, faster by lessening the load on branch prediction mechanisms.
- Elimination of the cost of a branch miss prediction which can be high on deeply pipelined architectures.

Disadvantages:

Predication's primary drawback is in increased encoding space. In typical implementations, every instruction reserves a bit field for the predicate specifying under what conditions that instruction should have an effect. When available memory is limited, as on embedded devices, this space cost can be prohibitive. However, some architecture such as Thumb-2 are able to avoid this issue (see below). Other detriments are the following:

- Predication complicates the hardware by adding levels of logic to critical paths and potentially degrades clock speed.
- A predicated block includes cycles for all operations, so shorter paths may take longer and be penalized.

Predication is most effective when paths are balanced or when the longest path is the most frequently executed, but determining such a path is very difficult at compile time, even in the presence of profiling information.

Pipeline Performance Analysis

1. CPI of a Pipeline Processor

Suppose an N-segment pipeline processes M instructions without stalls or penalties. We know that it takes N-1 cycles to load (setup) the pipeline, and M cycles to complete the instructions. Thus, the number of cycles is given by:

$N_{cyc} = N + M - 1$

The cycles per instruction are easily computed:

$CPI = N_{cyc}/M = 1 + (N - 1)/M$

2. Effect of Stalls

Now let us add some stalls to the pipeline processing scheme. Suppose that we have a N-segment pipeline processing M instructions, and we must insert K stalls to resolve data dependencies. This means that the pipeline now has a setup penalty of N-1 cycles, as before, a stall penalty of K cycles, and a processing cost (as before) of M cycles to process the M instructions. Thus, our governing equations become:

$N_{cyc} = N + M + K - 1$

and

$CPI = N_{cyc}/M = 1 + (N + K - 1)/M$

In practice, what does this tell us? Namely, that the stall penalty (and all the other penalties that we will examine) adversely impact CPI. Here is an example to show how we would analyze the problem of stalls in a pipelined program where the percentage of instructions that incur stalls versus non-stalls are specified.

3. Suppose that an N-segment pipeline executes M instructions, and that a fraction f_{stall} of the instructions require the insertion of K stalls per instruction to resolve data dependencies. The total number of stalls is given by $f_{stall} \cdot M \cdot K$ (fraction of instructions that are stalls, times the total number of instructions, times the average number of stalls per instruction). By substitution, our preceding equations for pipeline performance become:

$$N_{cyc} = N + M + (f_{stall} \cdot M \cdot K) - 1$$

and

$$CPI = N_{cyc}/M = 1 + (f_{stall} \cdot K) + (N - 1)/M$$

So, the CPI penalty due to the combined effects of setup cost and stalls now increases to fK + (N - 1)/M. If $f_{stall} = 0.1$, K = 3, N = 5, and M = 100, then CPI = 1 + 0.3 + 4/100 = 1.34, which is 34 percent larger than the fallacious assumption of CPI = 1.

3. Effect of Exceptions

For purposes of discussion, assume that we have M instructions executing on an N-segment pipeline with no stalls, but that a fraction f_{ex} of the instructions raise an exception in the EX stage. Further assume that each exception requires that (a) the pipeline segments before the EX stage be flushed, (b) that the exception be handled, requiring an average of H cycles per exception, then that (c) the instruction causing the exception and its following instructions be reloaded into the pipeline.

Thus, $f_{ex} \cdot M$ instructions will cause exceptions. In the MIPS pipeline, each of these instructions causes three instructions to be flushed out of the pipe (IF, ID, and EX stages), which incurs a penalty of four cycles (one cycle to flush, and three to reload) plus H cycles to handle the exception. Thus, the pipeline performance equations become:

$$N_{cyc} = N - 1 + (1 - f_{ex}) \cdot M + (f_{ex} \cdot M \cdot (H + 4))$$

which we can rewrite as

$$N_{cyc} = M + [N - 1 - M + (1 - f_{ex}) \cdot M + (f_{ex} \cdot M \cdot (H + 4))]$$

Rearranging terms, the equation for CPI can be expressed as

$$CPI = N_{cyc}/M = 1 + [1 - f_{ex} + (f_{ex} \cdot (H+4)) - 1 + (N - 1)/M]$$

After combining terms, this becomes:

$$CPI = N_{cyc}/M = 1 + [(f_{ex} \cdot (H+3)) + (N - 1)/M]$$

4. Effect of Branches

Branches present a more complex picture in pipeline performance analysis. Recall that there are three ways of dealing with a branch: (1) Assume the branch is not taken, and if the branch is taken, flush the

instructions in the pipe after the branch, then insert the instruction pointed to by the BTA; (2) the converse of 1); and (3) use a delayed branch with a branch delay slot and re-ordering of code (assuming that this can be done).

The first two cases are symmetric. Assume that an error in branch prediction (i.e., taking the branch when you expected not to, and conversely) requires L instruction to be flushed from the pipeline (one cycle for flushing plus L-1 "dead" cycles, since the branch target can be inserted in the IF stage). Thus, the cost of each branch prediction error is L cycles. Further assume that a fraction f_{br} of the instructions are branches and f_{be} of these instructions result in branch prediction errors.

The penalty in cycles for branch prediction errors is thus given by

branch_penalty = $f_{br} \cdot f_{be} \cdot$ M instructions $\cdot$ L cycles per instruction.

The pipeline performance equations then become:

$$N_{cyc} = N - 1 + (1 - f_{br} \cdot f_{be}) \cdot M + (f_{br} \cdot f_{be} \cdot M \cdot L)$$

which we can rewrite as

$$N_{cyc} = M + [N - 1 - M + (1 - f_{br} \cdot f_{be}) \cdot M + (f_{br} \cdot f_{be} \cdot M \cdot L)$$

Rearranging terms, the equation for CPI can be expressed as

$$CPI = N_{cyc}/M = 1 + [(1 - f_{br} \cdot f_{be}) + (f_{br} \cdot f_{be} \cdot L) - 1 + (N - 1)/M].$$

After combining terms, this becomes:

$$CPI = N_{cyc}/M = 1 + [(f_{br} \cdot f_{be} \cdot (L-1)) + (N - 1)/M]$$

In the case of the branch delay slot, we assume that the branch target address is computed and the branch condition is evaluated at the ID stage. Thus, if the branch prediction is correct, there is no penalty. Depending on the method by which the pipeline evaluates the branch and fetches (or pre-fetches) the branch target, a maximum of two cycles penalty (one cycle for flushing, one cycle for fetching and inserting the branch target) is incurred for insertion of a stall in the case of a branch prediction error. In this case, the pipeline performance equations become:

$$N_{cyc} = N - 1 + (1 - f_{br} \cdot f_{be}) \cdot M + (f_{br} \cdot f_{be} \cdot 2M)$$

This implies the following equation for CPI as a function of branches and branch prediction errors:

$$CPI = N_{cyc}/M = 1 + [f_{br} \cdot f_{be} + (N - 1)/M]$$

Since $f_{br} << 1$ is usual, and f_{be} is, on average, assumed to be no worse than 0.5, the product $f_{br} \cdot f_{be}$, which represents the additional branch penalty for CPI in the presence of delayed branch and BDS, is generally small.

Name of the Paper: Advanced Computer Architecture
Paper Code: CS802D
Contact (Periods/Week):3L/Week
Credit Point: 3
No. of Lectures: 35
Module –2: Parallel Processing & ILP[8L]

RISC architecture, characteristics of RISC instruction set & RISC pipeline, its comparisons with CISC,necessity of using optimizing compilers with RISC architecture, Review of instruction-level parallelism-Super pipelining, Superscalar architecture, Diversified pipelines and out of order execution, VLIW architecture, Dataflow and Control Flow Architectures, Loop Parallelization

Module –2: Parallel Processing & ILP [8L]

Lecture 1

Instruction set architecture (ISA)

Instruction set architecture(ISA) is the set of processor design techniques used to implement the instruction work flow on hardware. In more practical words, ISA tells you that how your processor going to process your program instructions.

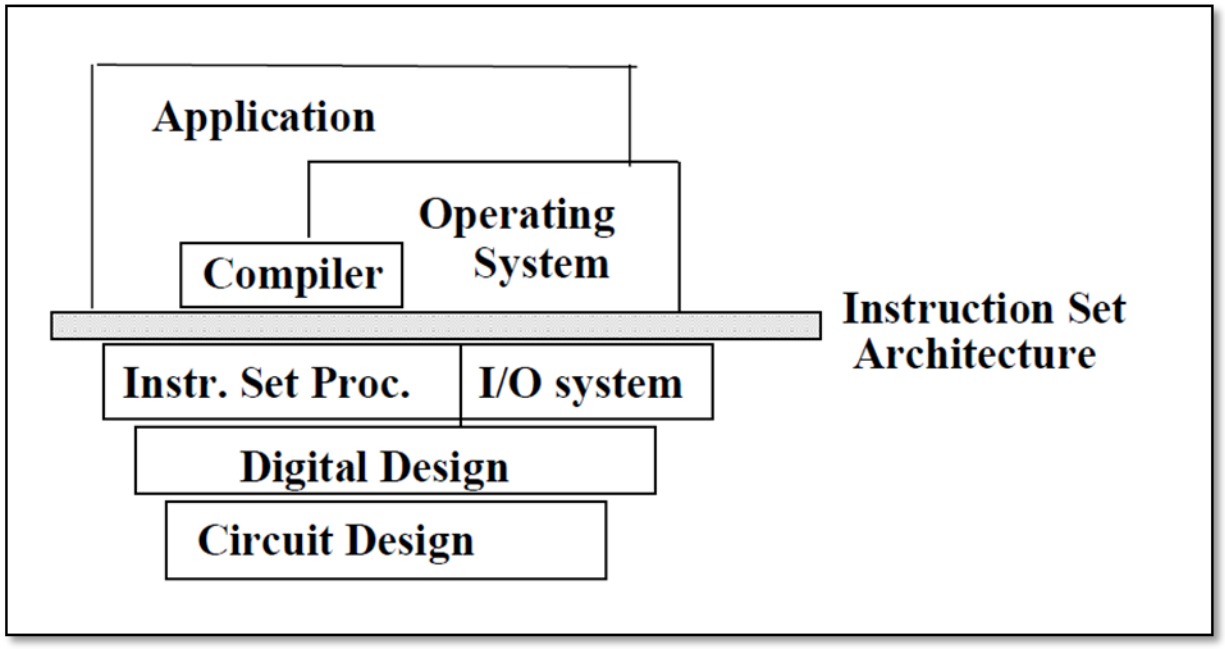


Fig: 2.1 Instruction set architecture (ISA)

There is no standard computer architecture accepting different types like CISC, RISC, etc.

Complex instruction set computer (CISC)

A complex instruction set computer (CISC /pronounce as ˈsisk'/) is a computer where single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions, as its name suggest "COMPLEX INSTRUCTION SET".

CISC Characteristics:

1. A large number of instructions-typically from 100 to 250 instructions.

2. Some instructions that perform specialized tasks and are used infrequently.

3. A large variety of addressing modes-typically from 5 to 20 different modes.

4. Variable-length instruction formats.

5. Instructions that manipulate operands in memory.

Reduced instruction set computer (RISC)

A reduced instruction set computer (RISC /pronounce as ˈrisk’/) is a computer which only use simple instructions that can be divide into multiple instructions which perform low-level operation within single clock cycle, as its name suggest “REDUCED INSTRUCTION SET”.

RISC Characteristics:

1. Relatively few instructions.

2. Relatively few addressing modes.

3. Memory access limited to load and store instructions.

4. All operations done within the registers of the CPU.

5. Fixed-length, easily decoded instruction format.

6. Single-cycle instruction execution.

7. Hardwired rather than micro programmed control.

RISC & CISC architecture with example

Let we take an example of multiplying two numbers
A = A * B; <<<======this is C statement

The CISC Approach: - The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding & executing a series of operations, this is where our CISC architecture introduced.

For this particular task, a CISC processor would come prepared with a specific instruction (we’ll call it “MULT”). When executed, this instructionLoads the two values into separate registers
Multiplies the operands in the execution unitAnd finally third, stores the product in the appropriate register.Thus, the entire task of multiplying two numbers can be completed with one instruction:

MULT A,B<<<======this is assembly statement

MULT is what is known as a "complex instruction." It operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions.

Advantages:-

- Compiler has to do very little work to translate a high-level language statement into assembly
- Length of the code is relatively short
- Very little RAM is required to store instructions
- The emphasis is put on building complex instructions directly into the hardware.

The RISC Approach: - RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MULT" command described above could be divided into three separate commands:

"LOAD" which moves data from the memory bank to a register

"PROD" which finds the product of two operands located within the registers

"STORE" which moves data from a register to the memory banks.

In order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly:

```
LOAD R1, A      <<<======this is assembly statement
LOAD R2,B<<<======this is assembly statement
PROD A, B       <<<======this is assembly statement
STORE R3, A     <<<======this is assembly statement
```

At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code, more RAM is needed to store the assembly level instructions. The compiler must also perform more work to convert a high-level language statement into code of this form.

Advantages:-

- Each instruction requires only one clock cycle to execute, the entire program will execute in approximately the same amount of time as the multi-cycle "MULT" command.
- These RISC "reduced instructions" require less transistors of hardware space than the complex instructions, leaving more room for general purpose registers. Because all of the instructions execute in a uniform amount of time (i.e. one clock)
- Pipelining is possible.
- LOAD/STORE mechanism:- Separating the "LOAD" and "STORE" instructions actually reduces the amount of work that the computer must perform. After a CISC-style "MULT" command is executed, the processor automatically erases the registers. If one of the operands needs to be used for another computation, the processor must re-load the data from the memory bank into a register. In RISC, the operand will remain in the register until another value is loaded in its place.

Example of RISC & CISC

Examples of CISC instruction set architectures are PDP-11, VAX, Motorola 68k, and your desktop PCs on Intel's x86 architecture based too.

Examples of RISC families include DEC Alpha, AMD 29k, ARC, Atmel AVR, Blackfin, Intel i860 and i960, MIPS, Motorola 88000, PA-RISC, Power (including PowerPC), SuperH, SPARC and ARM too.

Which one is better?

We cannot differentiate RISC and CISC technology because both are suitable at its specific application. What counts are how fast a chip can execute the instructions it is given and how well it runs existing software. Today, both RISC and CISC manufacturers are doing everything to get an edge on the competition.

What's new?

You might thinking that RISC is now-a-days used in microcontroller application widely, so it's better for that particular application and CISC at desktop application. But reality is both are at threat position cause of a new technology called EPIC.

EPIC (Explicitly Parallel Instruction Computing) :-EPIC is a invented by Intel and is in a way, a combination of both CISC and RISC. This will in theory allow the processing of Windows-based as well as UNIX-based applications by the same CPU.

Intel is working on it under code-name Merced. Microsoft is already developing their Win64 standard for it. Like the name says, Merced will be a 64-bit chip.

If Intel's EPIC architecture is successful, it might be the biggest thread for RISC. All of the big CPU manufactures but Sun and Motorola are now selling x86-based products, and some are just waiting for Merced to come out (HP, SGI). Because of the x86 market it is not likely that CISC will die soon, but RISC may.

So the future might bring EPIC processors and more CISC processors, while the RISC processors are becoming extinct.

CISC	RISC
Larger set of instructions. Easy to program	Smaller set of Instructions. Difficult to program.

Simpler design of compiler, considering larger set of instructions.	Complex design of compiler.
Many addressing modes causing complex instruction formats.	Few addressing modes, fix instruction format.
Instruction length is variable.	Instruction length varies.
Higher clock cycles per second.	Low clock cycle per second.
Emphasis is on hardware.	Emphasis is on software.
Control unit implements large instruction set using micro-program unit.	Each instruction is to be executed by hardware.
Slower execution, as instructions are to be read from memory and decoded by the decoder unit.	Faster execution, as each instruction is to be executed by hardware.
Pipelining is not possible.	Pipelining of instructions is possible, considering single clock cycle.

Lecture 2

Reduced Code Size in RISCs

As RISC computers started being used in embedded applications, the 32-bit fixed format became a liability since cost and hence smaller code are important. In response, several manufacturers offered a new hybrid version of their RISC instruction sets, with both 16-bit and 32-bit instructions. The narrow instructions support fewer operations, smaller address and immediate fields, fewer registers, and two-address format rather than the classic three-address format of RISC computers.

The Role of Compilers

Today almost all programming is done in high-level languages for desktop and server applications. This development means that since most instructions executed are the output of a compiler, an instruction set architecture is essentially a compiler target. In earlier times for these applications, architectural decisions were often made to ease assembly language programming or for a specific kernel. Because the compiler will significantly affect the performance of a computer, understanding compiler technology today is critical to designing and efficiently implementing an instruction set. Once it was popular to try to isolate the compiler technology and its effect on hardware performance from the architecture and its performance, just as it was popular to try to separate architecture from its implementation. This separation is essentially impossible with today's desktop compilers and computers. Architectural choices affect the quality of the code that can be generated for a computer and the complexity of building a good compiler for it, for better or for worse.

The Structure of Recent Compilers

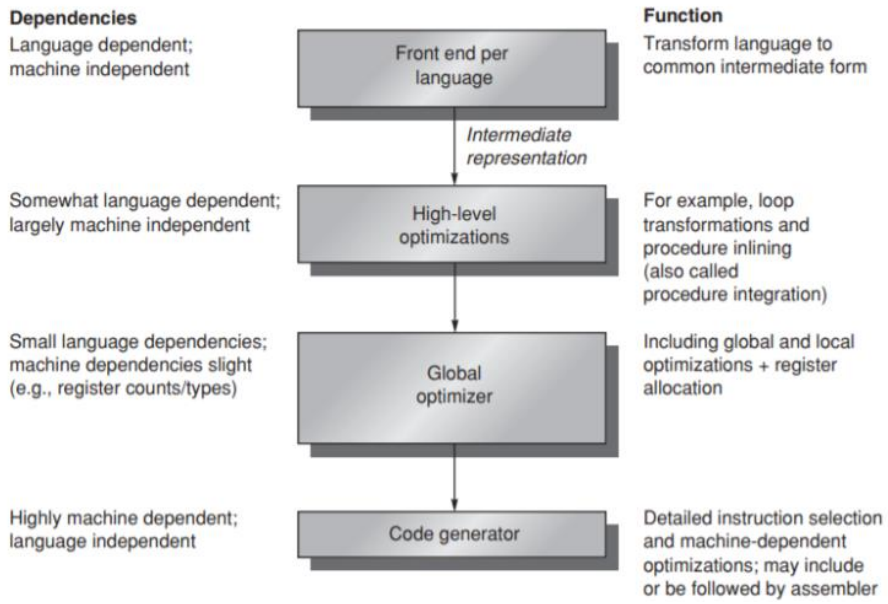


Figure: the structure of recent compilers

Compilers typically consist of two to four passes, with more highly optimizing compilers having more passes. This structure maximizes the probability that a program compiled at various levels of optimization will produce the same output when given the same input. The optimizing passes are designed to be optional and may be skipped when faster compilation is the goal and lower-quality code is acceptable. A pass is simply one phase in which the compiler reads and transforms the entire program. (The term phase is often used interchangeably with pass.) Because the optimizing passes are separated, multiple languages can use the same optimizing and code generation passes. Only a new front end is required for a new language.

A compiler writer's first goal is correctness—all valid programs must be compiled correctly. The second goal is usually speed of the compiled code. Typically, a whole set of other goals follows these two, including fast compilation, debugging support, and interoperability among languages. Normally, the passes in the compiler transform higher-level, more abstract representations into progressively lower-level representations. Eventually it reaches the instruction set. This structure helps manage the complexity of the transformations and makes writing a bug-free compiler easier. The complexity of writing a correct compiler is a major limitation on the amount of optimization that can be done. Although the multiple-pass structure helps reduce compiler complexity, it also means that the compiler must order and perform some transformations before others. In the diagram of the optimizing compiler, we can see that certain high-level optimizations are performed long before it is known what the resulting code will look like. Once such a transformation is made, the compiler can't afford to go back and revisit all steps, possibly undoing transformations. Such iteration would be prohibitive, both in compilation time and in complexity. Thus, compilers make assumptions about the ability of later steps to deal with certain problems. For example, compilers usually have to choose which procedure calls to expand inline before they know the exact size of the procedure being called. Compiler writers call this problem the phase-ordering problem.

How does this ordering of transformations interact with the instruction set architecture? A good example occurs with the optimization called global common sub expression elimination. This optimization finds two instances of an expression that compute the same value and saves the value of the first computation in a temporary. It then uses the temporary value, eliminating the second computation of the common expression.

For this optimization to be significant, the temporary must be allocated to a register. Otherwise, the cost of storing the temporary in memory and later reloading it may negate the savings gained by not re-computing the expression. There are, in fact, cases where this optimization actually slows down code when the temporary is not register allocated. Phase ordering complicates this problem because register allocation is typically done near the end of the global optimization pass, just before code generation. Thus, an optimizer that performs this optimization must assume that the register allocator will allocate the temporary to a register.

Optimizations performed by modern compilers can be classified by the style of the transformation, as follows:

■ High-level optimizations are often done on the source with output fed to later optimization passes.

■ Local optimizations optimize code only within a straight-line code fragment (called a basic block by compiler people).

■ Global optimizations extend the local optimizations across branches and introduce a set of transformations aimed at optimizing loops.

■ Register allocation associates registers with operands.

■ Processor-dependent optimizations attempt to take advantage of specific architectural knowledge.

Lecture 3

PARALLELISM

With the era of increasing processor speeds slowly coming to an end, computer architects are exploring new ways of increasing throughput. One of the most promising is to look for and exploit different types of parallelism in code.

Levels of parallelism are described below:

1. **Instruction Level:** At instruction level, a grain is consist of less than 20 instruction called fine grain. Fine grain parallelism at this level may range from two thousands depending an individual program single instruction stream parallelism is greater than two but the average parallelism at instruction level is around fine rarely exceeding seven in ordinary program. For scientific applications average parallel is in the range of 500 to 300 fortran statements executing concurrently in an idealized environment.

2. **Loop Level:** It embrace iterative loop operations. A loop may contain less than 500 instructions. Some loop independent operation can be vectorized for pipelined execution or for look step execution of SIMD machines. Loop level parallelism is the most optimized program construct to execute on a parallel or vector computer. But recursive loops are different to parallelize. Vector processing is mostly exploited at the loop level by vectorizing compiler.

3. **Procedural Level:** It communicate to medium grain size at the task, procedure, subroutine levels. Grain at this level has less than 2000 instructions. Detection of parallelism at this level is much more difficult than a finer grain level. Communication obligation is much less as compared with 16 that MIMD execution mode. But here major efforts are requisite by the programmer to reorganize a program at this level.

4. **Subprogram Level:** Subprogram level communicate to job steps and related subprograms. Grain size here have less than 1000 instructions. Job steps can overlap across diverse jobs. Multiprogramming an uniprocessor or multiprocessor is conducted at this level.

5. **Job Level:** It corresponds to parallel executions of independent tasks on parallel computer. Grain size here can be tens of thousands of instructions. It is handled by program loader and by operating system. Time sharing & space sharing multiprocessors explores this level of parallelism.

Instruction Level Parallelism

Instruction-level parallelism (ILP) is a measure of how many of the instructions in a computer program can be executed simultaneously.

Pipelining can overlap the execution of instructions when they are independent of one another. This potential overlap among instructions is called instruction-level parallelism (ILP) since the instructions can be evaluated in parallel.

(ILP) is a measure of how many of the operations in a computer program can be performed simultaneously. Consider the following program:

1. e = a + b

2. f = c + d

3. g = 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 theycan be calculated simultaneously. If we assume that each operation can be completed in one unitof time then these three instructions can be completed in a total of two units of time, giving anILP of 3/2.

A goal of compiler and processor designers is to identify and take advantage of as much ILP as possible. ILP allowsthe compiler and the processor to overlap the execution of multiple instructions or even tochange the order in which instructions are executed.

How much ILP exists in programs is very application specific. In certain fields, such as graphicsand scientific computing the amount can be very large. However, workloads such ascryptography exhibit much less parallelism.

The simplest and most common way to increase the amount of parallelism available among instructions is to exploit parallelism among iterations of a loop. This type of parallelism is often called loop-level parallelism.

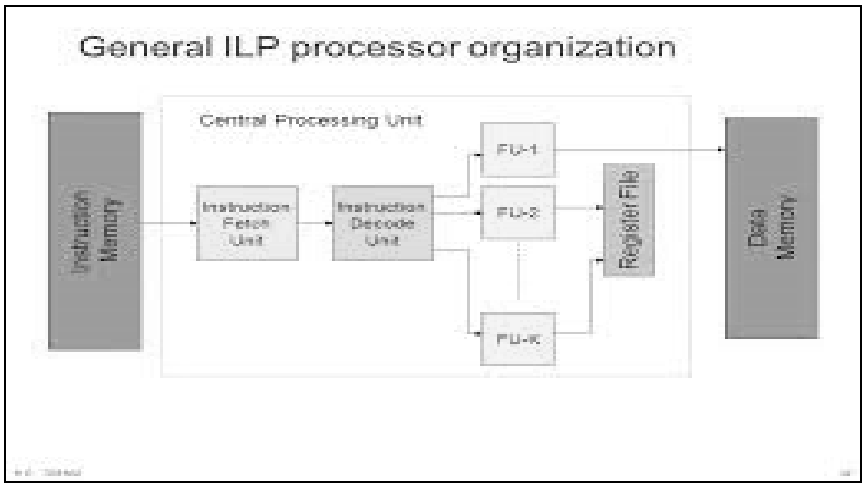


Fig: 2.1 ILP Processor

LECTURE 4

Superscalar processor

A superscalar processor is a CPU that implements a form of parallelism called instruction-level parallelism within a single processor**.**

In contrast to a scalar processor that can execute at most one single instruction per clock cycle, a superscalar processor can execute more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to different execution units on the processor. It therefore allows for more throughput (the number of instructions that can be executed in a unit of time) than would otherwise be possible at a given clock rate. Each execution unit is not a separate processor (or a core if the processor is a multi-core processor), but an execution resource within a single CPU such as an arithmetic logic unit.

The superscalar technique is traditionally associated with several identifying characteristics (within a given CPU):

1.Instructions are issued from a sequential instruction stream

2.The CPU dynamically checks for data dependencies between instructions at run time (versus software checking at compile time)

3.The CPU can execute multiple instructions per clock cycle.

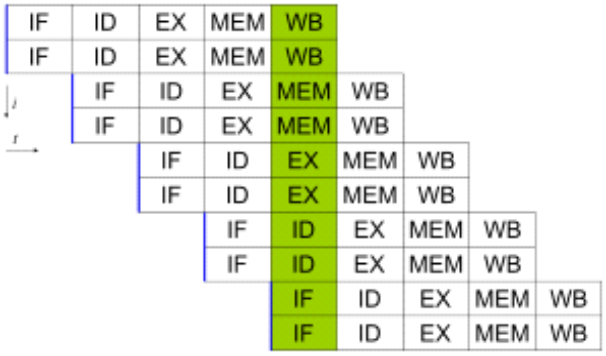


fig:- Simple superscalar pipeline. By fetching and dispatching two instructions at a time, a maximum of two instructions per cycle can be completed. (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back, i = Instruction number, t = Clock cycle [i.e., time])

Fig:2.2 Superscalar Execution

Super pipelined Processors

Traditional pipelined architectures have a single pipeline stage for each of instruction cycle stage: instruction fetch, instruction decode, memory read, ALU operation and memory write.

A super pipelined processor has a pipeline where each of these logical steps may be sub divided into multiple pipeline stages.

In contrast to a superscalar processor, a super pipelined one has split the main computational pipeline into more stages. Each stage is simpler (does less work) and thus the clock speed can be

increased. However the *latency*, measured in clock cycles, for any instruction to complete has increased from 4 cycles in early RISC processors to 8 or more.

Benefit

The major benefit of super pipelining is the increase in the number of instructions which can be in the pipeline at one time and hence the level of parallelism.

Drawbacks

The larger number of instructions "in flight" (*ie* in some part of the pipeline) at any time, increases the potential for data dependencies to introduce stalls. Simulation studies have suggested that a pipeline depth of more than 8 stages tends to be counter-productive.

Superscalar vs. Superpipelined

- Superscalar machines can issue several instructions per cycle. Superpipelined machines can issue only one instruction per cycle, but they have cycle times shorter than the time required for any operation.

  Both of these techniques exploit instruction-level parallelism, which is often limited in many applications. Superpipelined machines are shown to have better performance and less cost than superscalar machines.

LECTURE 5

Overcoming Data Hazards with Dynamic Scheduling

A major limitation of simple pipelining techniques is that they use in-order instruction issue and execution: Instructions are issued in program order, and if an instruction is stalled in the pipeline, no later instructions can proceed. Thus, if there is a dependence between two closely spaced instructions in the pipeline, this will lead to a hazard and a stall will result. If there are multiple functional units, these units could lie idle. If instruction j depends on a long-running instruction i, currently in execution in the pipeline, then all instructions after j must be stalled until i is finished and j can execute. For example, consider this code:

DIV.D F0, F2, F4
ADD.D F10, F0, F8
SUB.D F12, F8, F14

The SUB.D instruction cannot execute because the dependence of ADD.D on DIV.D causes the pipeline to stall; yet SUB.D is not data dependent on anything in the pipeline. This hazard creates a performance limitation that can be eliminated by not requiring instructions to execute in program order.

In the classic five-stage pipeline, both structural and data hazards could be checked during instruction decode (ID): When an instruction could execute without hazards, it was issued from ID knowing that all data hazards had been resolved.

To allow us to begin executing the SUB.D in the above example, we must separate the issue process into two parts:

- checking for any structural hazards and waiting for the absence of a data hazard. Thus, we still use in-order instruction issue (i.e., instructions issued in program order),
- but we want an instruction to begin execution as soon as its data operands are available. Such a pipeline does out-of-order execution, which implies out-of-order completion.

Out-of-order execution

Out-of-order execution introduces the possibility of WAR and WAW hazards, which do not exist in the five-stage integer pipeline and its logical extension to an in-order floating-point pipeline. Consider the following MIPS floating-point code sequence:

DIV.D F0, F2, F4

ADD.D F6, F0, F8

SUB.D F8, F10, F14

MUL.D F6, F10, F8

There is an antidependence between the ADD.D and the SUB.D, and if the pipeline executes the SUB.D before the ADD.D (which is waiting for the DIV.D), it will violate the antidependence, yielding a WAR hazard. Likewise, to avoid violating output dependences, such as the write of F6 by MUL.D, WAW hazards must be handled. As we will see, both these hazards are avoided by the use of register renaming. Out-of-order completion also creates major complications in handling exceptions. Dynamic scheduling with out-of-order completion must preserve exception behavior in the sense that exactly those exceptions that would arise if the program were executed in strict program order actually do arise. Dynamically scheduled processors preserve exception behavior by ensuring that no instruction can generate an exception until the processor knows that the instruction raising the exception will be executed; we will see shortly how this property can be guaranteed.

Although exception behavior must be preserved, dynamically scheduled processors may generate imprecise exceptions. An exception is imprecise if the processor state when an exception is raised does not look exactly as if the instructions were executed sequentially in strict program order. Imprecise exceptions can occur because of two possibilities:

1. The pipeline may have already completed instructions that are later in program order than the instruction causing the exception.

2. The pipeline may have not yet completed some instructions that are earlier in program order than the instruction causing the exception.

LECTURE 6

VLIW (Very Long Instruction Word)

Very long instruction word (VLIW) describes a computer processing architecture in which a language compiler or pre-processor breaks program instruction down into basic operations that can be performed by the processor in parallel (that is, at the same time). These operations are put into a very long instruction word which the processor can then take apart without further analysis, handing each operation to an appropriate functional unit.

VLIW is sometimes viewed as the next step beyond the reduced instruction set computing (RISC) architecture, which also works with a limited set of relatively basic instructions and can usually execute more than one instruction at a time (a characteristic referred to as *superscalar*).

The VLIW architecture is generalized from two well-established concepts: horizontal microcoding and superscalar processing. A typical VLIW (very long instruction word) machine has instruction words hundreds of bits in length. As illustrated in fig. 4.14a, multiple functional units are used concurrently in a VLIW processor. All functional units share the use of a common large register file. The operations to be simultaneously executed by the functional units are synchronised in a VLIW instruction, say , 256 or 1024 bits per instruction word, as implemented in the Multi-flow computer models .

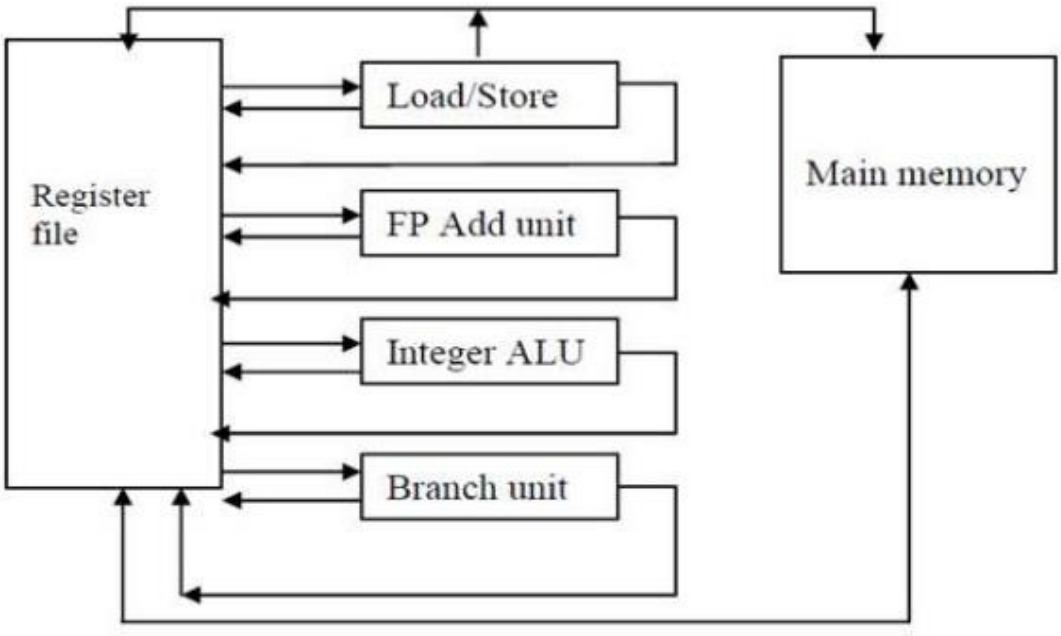


Fig: 2.3VLIW Architecture

The VLIW concept is borrowed from horizontal micro coding. Different fields of the long instruction word carry the opcodes to be dispatched to different functional units. Programs written in conventional short instruction words(say 32 bits) must be compacted together to form

the VLIW instructions. This code compaction must be done by a compiler which can predict branch outcomes using elaborate heuristics or run-time statistics.

The main advantage of VLIW processors is that complexity is moved from the hardware to the software, which means that the hardware can be smaller, cheaper, and require less power to operate. The challenge is to design a compiler or pre-processor that is intelligent enough to decide how to build the very long instruction words. If dynamic pre-processing is done as the program is run, performance may be a concern.

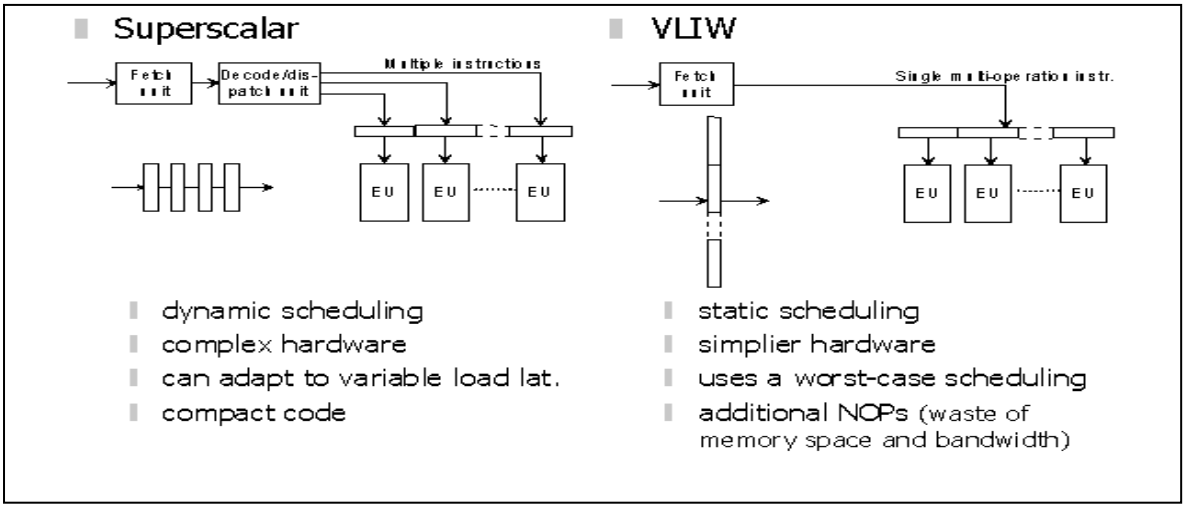


Fig: 2.4 Superscalar Vs VLIW Architecture

Super Pipelined:

In contrast to a superscalar processor, a superpipelined one has split the main computational pipeline into more stages. Each stage is simpler (does less work) and thus the clock speed can be increased. However the latency, measured in clock cycles, for any instruction to complete has increased from 4 cycles in early RISC processors to 8 or more.

The major benefit of superpipelining is the increase in the number of instructions which can be in the pipeline at one time and hence the level of parallelism.

.

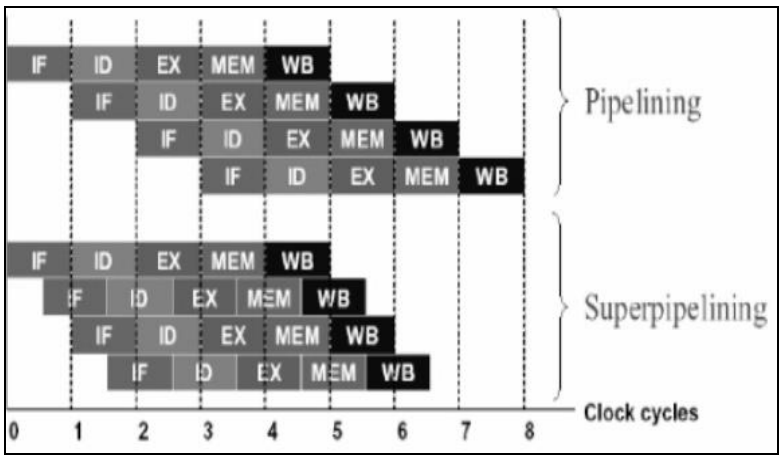


Fig: 2.5 Pipelining Vs Super pipelining Architecture

The larger number of instructions "in flight" (ie in some part of the pipeline) at any time, increases the potential for data dependencies to introduce stalls. Simulation studies have suggested that a pipeline depth of more than 8 stages tends to be counter-productive

Super pipelining is based on dividing the stages of a pipeline into sub-stages and thus increasing thenumber of instructions which are supported by the pipeline at a given moment. By dividing each stage into two, the clock cycle period t will be reduced to the half, t/2; hence, at the maximum capacity, the pipeline produces a result every t/2 s. For a given architecture and the correspondingInstruction set there is an optimal number of pipeline stages; increasing the number of stagesover this limit reduces the overall performance. A solution to further improve speed is theSuperscalar architecture.

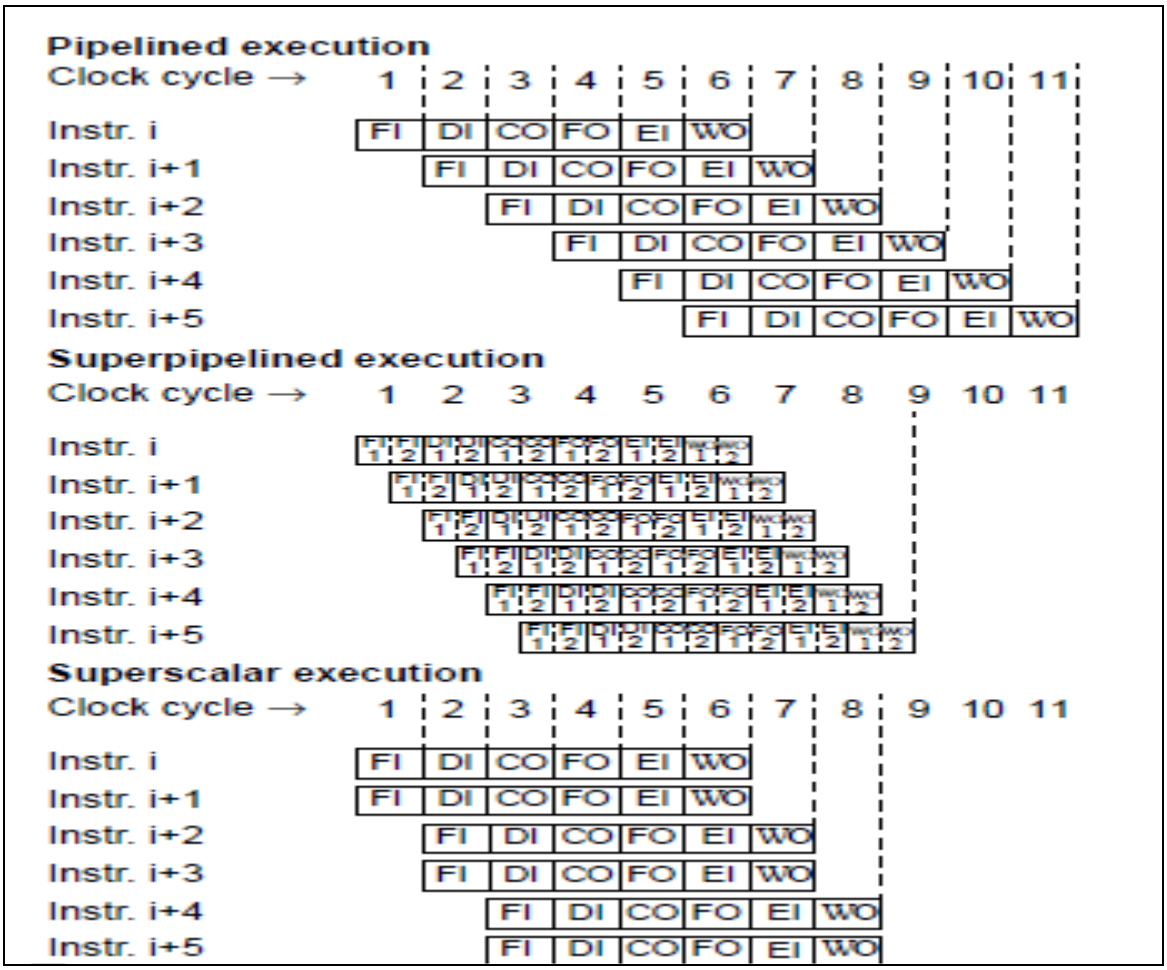


Fig: 2.6 Pipeline Vs Super Pipeline Vs Superscalar

Lecture 7

Program flow mechanisms

Traditional computers are founded on control flow mechanism by which the order of program execution is explicitly stated in the user program. Data flow computers have high degree of parallelism at the fine grain instruction level reduction computers are based on demand driven method which commence operation based on the demand for its result by other computations.

Data flow & control flow computers : There are mainly two sort of computers. Data flow computers are connectional computer based on Von Neumann machine. It carry out instructions under program flow control whereas control flow computer, executes instructions under availability of data.

Control flow Computers : Control Flow computers employ shared memory to hold program instructions and data objects. Variables in shared memory are updated by many instructions. The execution of one instruction may produce side effects on other instructions since memory is shared. In many cases, the side effects prevent parallel processing from taking place. In fact, a uniprocessor computer is inherently sequential due to use of control driven mechanism.

Data Flow Computers : In data flow computer, the running of an instruction is determined by data availability instead of being guided by program counter. In theory any instruction should be ready for execution whenever operands become available. The instructions in data driven program are not ordered in any way. Instead of being stored in shared memory, data are directly held inside instructions. Computational results are passed directly between instructions. The data generated by instruction will be duplicated into many copies and forwarded directly to all needy instructions.

This data driven scheme requires no shared memory, no program counter and no control sequencer. However it requires special method to detect data availability, to match data tokens with needy instructions and to enable the chain reaction of asynchronous instructions execution.

Name of the Paper: Advanced Computer Architecture
Paper Code: CS802D
Contact (Periods/Week):3L/Week
Credit Point: 3
No. of Lectures: 35

Module – 3: Interconnection Networks[13L]
Desirable properties of interconnection networks, static interconnection networks – path, cycle, double-loop, star, wheel, 2D mesh and its variants, multi-mesh, tree, shuffle-exchange, cube, cube-connected cycles

Dynamic interconnection networks: concepts of blocking, rearrangeable and blocking but rearrangeable networks, various types of multistage interconnection networks (MIN)- crossbar, clos, baseline, omega, Benes.

Module – 3: Interconnection Networks

LECTURE 1

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.

Desirable properties of interconnection networks

The topology of an interconnection network can be either static or dynamic. Static networks are created point-to-point direct connections which will not alter during execution. Dynamic networks are applied with switched channels, which are dynamically configured to match the communication demand in user programs. Static networks are used for fixed connections amid sub systems of a centralized system or multiple computing nodes of a distributed system. Dynamic networks consist of buses, crossbar switches, multistage networks, which are often used in shared memory multi processors. Both types of network have also been employ for inter PE data routing in SIMD computers. In general, a network is characterized by graph of finite number of nodes linked by directed or undirected edges. The number of nodes in the graph is called the network size.

Node Degree and Network Diameter : The number of edges incident on a node is called the node degree d. In the case of unidirectional channels, the number of channels into a node is the _in; degree and that out of a node is the _out‘ degree. Then the node degree is the total of the two. The node degree reveals the number of I/O ports required per node and their cost of a node. Hence, the node degree should be kept a constant, as small as possible in order to reduce cost. A constant node degree is very much preferred to get modularity in building blocks for scalable systems.

The diameter D of a network is the maximum shortest path amid any two nodes. The path length is measured by the number of links visited. The network diameter show the maximum number of distinct hops amid any two nodes, thus giving a figure of communication pros for the network. thus, the network diameter should be as small as doable from communication point of view.

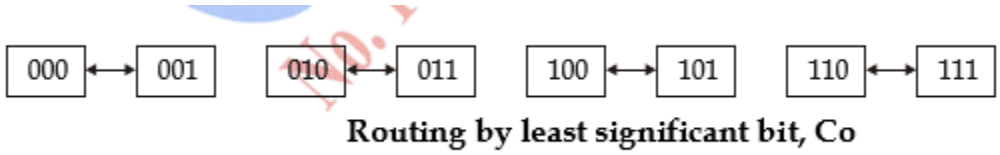
Bisection Width: When a specified network is cut into two identical halves, the minimum number of edges along the cut is termed as channel bisection width b. In the case of communication network, each edge match up to a channel with w bit wires. Then the wire bisection width is B = bw. This parameter B reflects the wiring density of a network. When B is fixed, the channel width w = B/b. Thus the bisection width offer a fine estimate of maximum

communication band width along the bisect ion of a network. Rest cross sections should be bounded by bisection width.

Data routing functions: Data routing networks is used for inter PE data exchange. Data routing network can be static or dynamic. In multicomputer network data routing is achieved by message among multiple computer nodes. Routing network reduces the time required for data exchange and thus system performance is enhanced. Commonly used data routing functions are shifting, rotation, permutations, broadcast, multicast, personalized communication, shuffle etc. Some Data routing functions are described below:

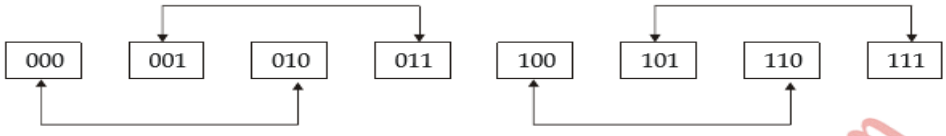
(a) Permutations: Let there are n objects, and then there are nf permutations by which n objects can be recorded. Set of all permutations form a permutation group with respect to composition operation. Generally cycle notation is used to specify permutation function. Cross can be used to implement the permutation. Multi stage network can implement some of the permutations in one or multiple passes through the network. Shifting and broadcast operation are also used to implement permutation operation. Permutation capability of a network is used to indicate the data routing capacity. Permutation speed dominates the performance of data routing network, when n is large.

(b) Hypercube routing function: Three dimensional cube is shown below:
Routing functions are defined by three bits in the node address. Bit order is C2C1Co. Data can be exchanged among adjacent nodes which differs in the least significant bit Co as shown below.

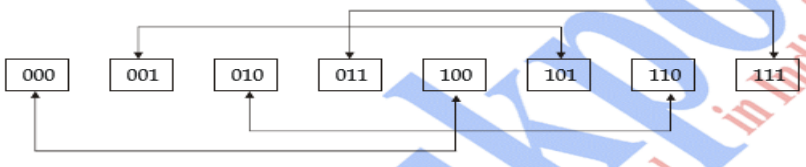


Routing by least significant bit, Co

Similarly routing pattern by using bit C_1 & C_2 is shown below:



Routing by middle bit, C_1



Routing by most significant bit, C_2

common pattern informs that n-dimensional, cube has n-routing functions, which are defined by each bit of the n-bit address. These data routing task are used in routing messages in a hypercube multi workstation.

(c) Broadcast & Multicast: Broad cast is one to all mapping. This is achieved by SIMD computers using a broadcast bus extending from array controller to all PEs. A mechanism is used to broadcast a message in message passing multi computer. Multicast means mapping from one subset to another. There is a variation of broadcast called personalized broadcast. Personalized broadcast sends messages to only selected receivers. Broadcast is a global operation in multi computer. Personalized broadcast may have to be implemented with matching of destination codes in the network.

LECTURE 2

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.

Classification of Interconnection network:

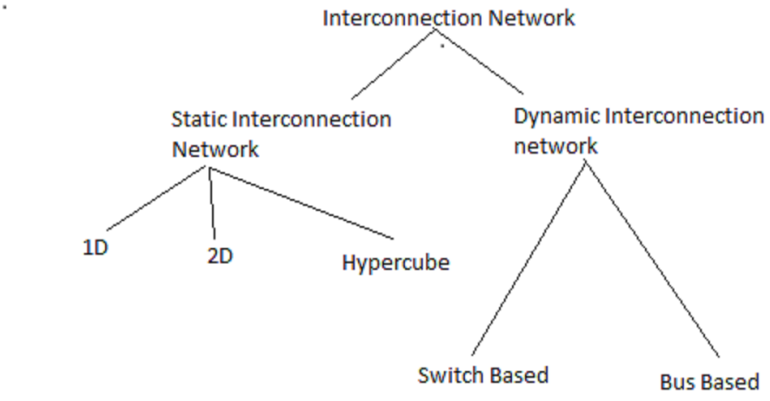
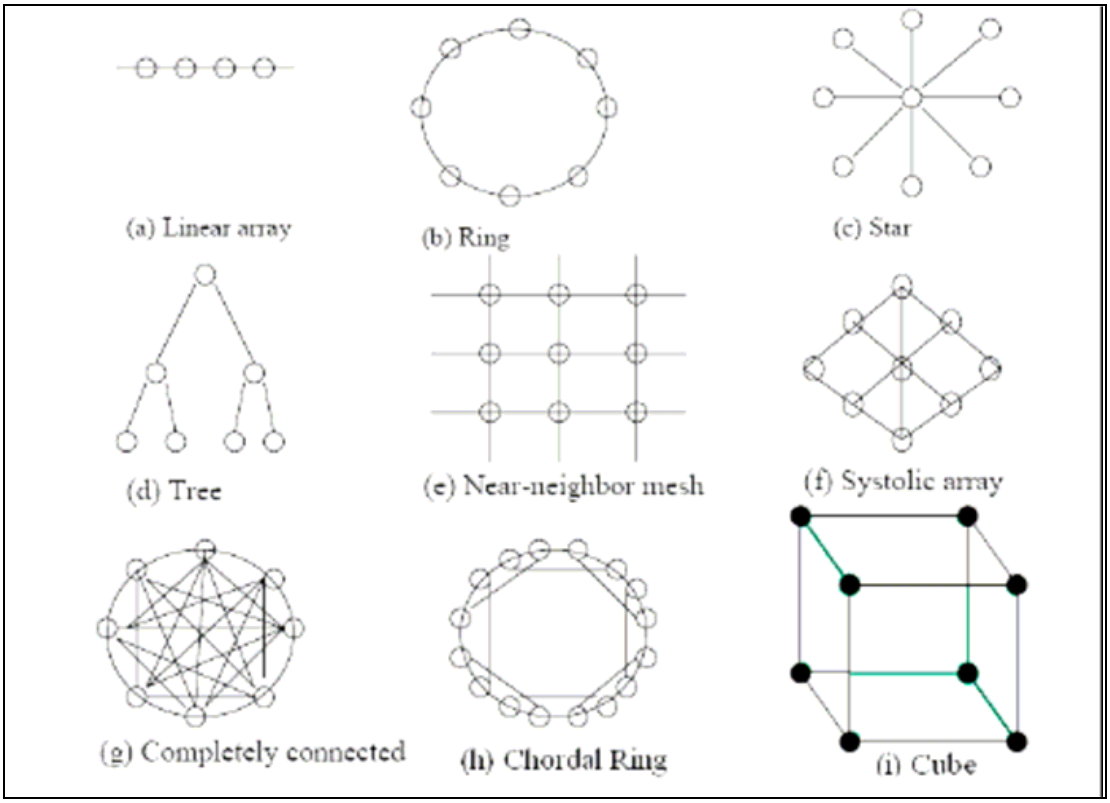
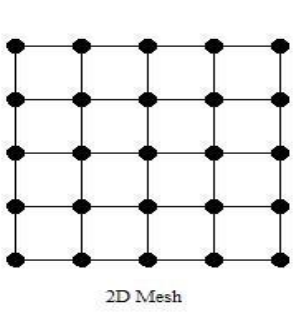


Figure: 3.1 Classification of Interconnection network

Static Interconnection Networks:

Static interconnection networks for elements of parallel systems (ex. processors, memories) are based on fixed connections that can't be modified without a physical re-designing of a system. Static interconnection networks can have many structures such as a linear structure (pipeline), a matrix, a ring, a torus, a **complete connection** structure, a tree, a star, a **hyper-cube**.





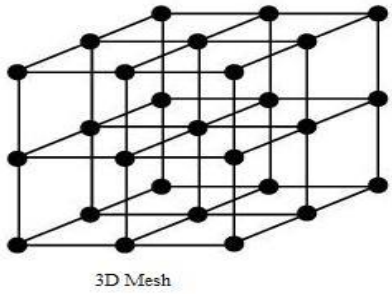


Figure: 3.2 Static interconnection network topologies

LECTURE 3

Hypercube Interconnection Network:

In a hypercube structure, processors are interconnected in a network, in which connections between processors correspond to edges of a n-dimensional cube. The hypercube structure is very advantageous since it provides a low **network diameter** equal to the degree of the cube. The network diameter is the number of edges between the most distant nodes. . The network diameter determines the number in intermediate transfers that have to be done to send data between the most distant nodes of a network. In this respect the hyper cubes have very good properties, especially for a very large number of constituent nodes. Due to this hyper cubes are popular networks in existing parallel systems.

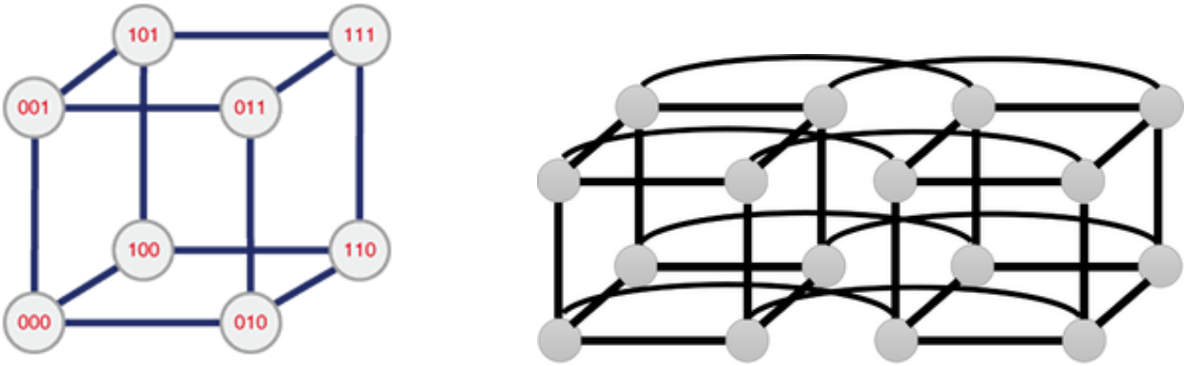


Figure: 3.3 Hypercube Network.

Dynamic Interconnection Networks:

Dynamic interconnection networks between processors enable changing (reconfiguring) of the connection structure in a system. It can be done before or during parallel program execution.

LECTURE 4

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 inter stage connection pattern (ISC). Small 2x2 switch elements are a common choice for many multistage networks. The number of stages determines the delay of the network. By choosing different inter stage 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.

LECTURE 5

Bus Networks:

- A bus is the simplest type dynamic interconnection networks. It constitutes a common data transfer path for many devices. Depending on the type of implemented transmissions we have **serial busses** and **parallel busses.** The devices connected to a bus can be processors, memories, I/O units, as shown in the figure below.

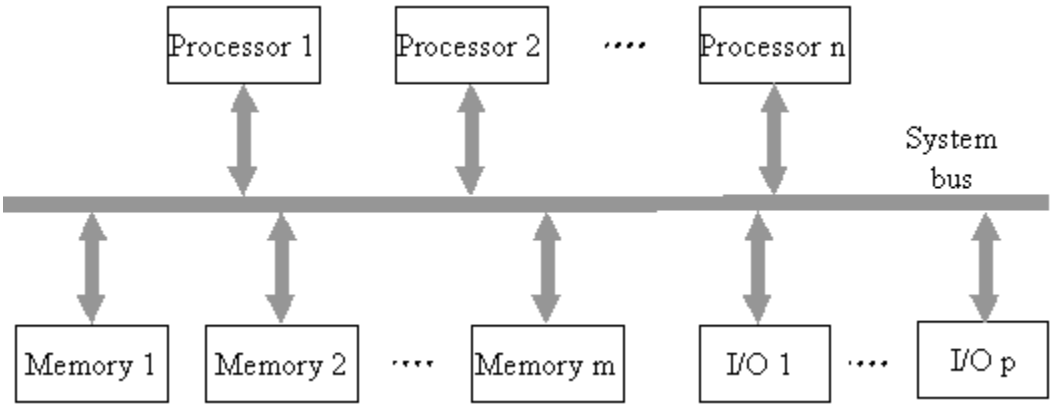


Figure: 3.4 a diagram of a system based on a single bus

Only one device connected to a bus can transmit data. Many devices can receive data. In the last case we speak about a **multicast transmission**. If data are meant for all devices connected to a bus we speak about a **broadcast transmission**. Accessing the bus must be synchronized. It is done with the use of two methods: a **token method** and a **bus arbiter method.** With the token method, a token (a special control message or signal) is circulating between the devices connected to a bus and it gives the right to transmit to the bus to a single device at a time. The bus arbiter receives data transmission requests from the devices connected to a bus. It selects one device according to a selected strategy (ex. using a system of assigned priorities) and sends an acknowledge message (signal) to one of the requesting devices that grants it the transmitting right. After the selected device completes the transmission, it informs the arbiter that can select another request. The receiver (s) address is usually given in the header of the message. Special header values are used for the broadcast and multicasts. All receivers read and decode headers. These devices that are specified in the header, read-in the data transmitted over the bus.

The throughput of the network based on a bus can be increased by the use of a **multi-bus network** shown in the figure below. In this network, processors connected to the busses can transmit data in parallel (one for each bus) and many processors can read data from many busses at a time.

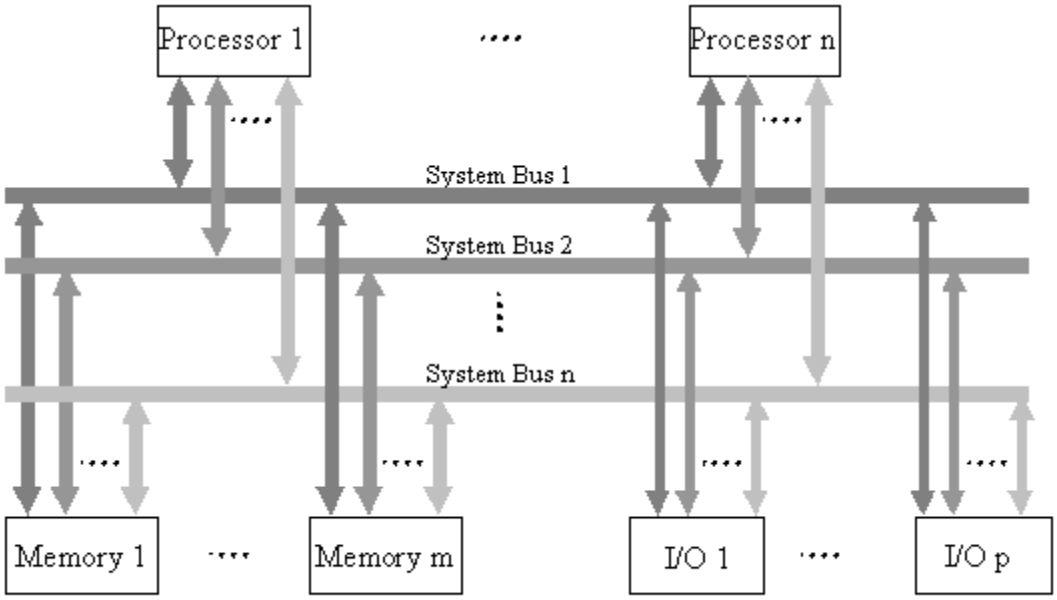


Figure: 3.5 a diagram of a system based on a multi- bus.

LECTURE: 6

Crossbar switches:

A crossbar switch is a circuit that enables many interconnections between elements of a parallel system at a time. A crossbar switch has a number of input and output data pins and a number of control pins. In response to control instructions set to its control input, the crossbar switch implements a stable connection of a determined input with a determined output. The diagrams of a typical crossbar switch are shown in the figure below.

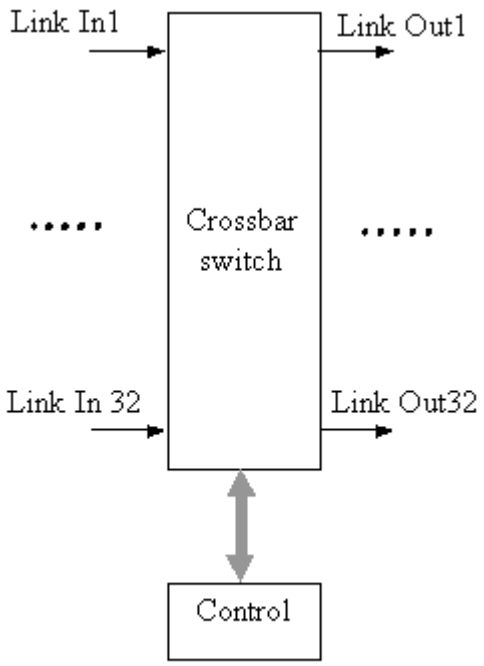


Figure: 3.6 Crossbar Switch, general scheme.

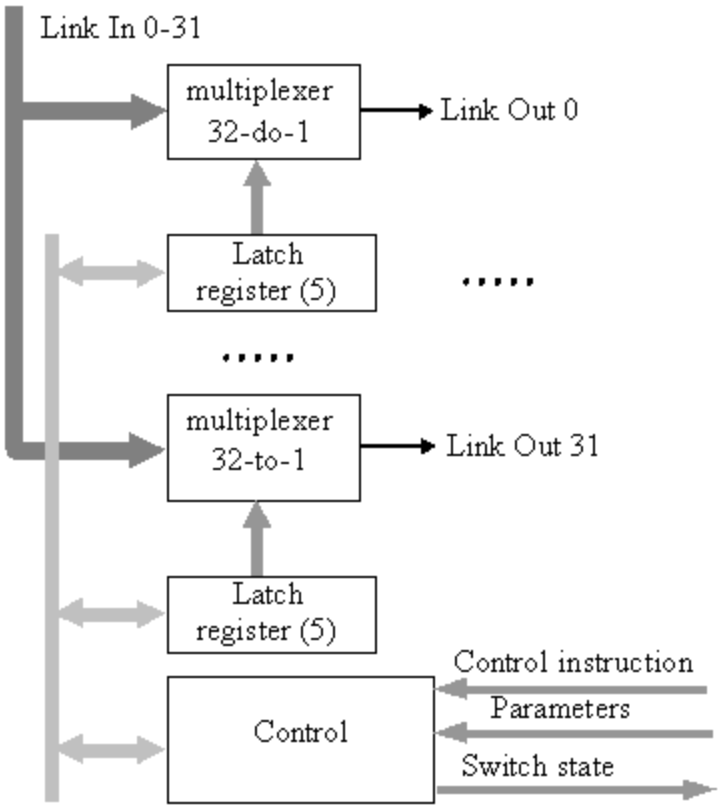


Figure: 3.7 Crossbar switch, internal structure

Control instructions can request reading the state of specified input and output pins i.e. their current connections in a crossbar switch. Crossbar switches are built with the use of multiplexer circuits, controlled by latch registers, which are set by control instructions. Crossbar switches implement direct, single **non-blocking connections**, but on the condition that the necessary input and output pins of the switch are free. The connections between free pins can always be implemented independently on the status of other connections. New connections can be set during data transmissions through other connections. The non-blocking connections are a big advantage of crossbar switches. Some crossbar switches enable broadcast transmissions but in a blocking manner for all other connections. The disadvantage of crossbar switches is that extending their size, in the sense of the number of input/output pins, is costly in terms of hardware. Because of that, crossbar switches are built up to the size of 100 input/output pins. The crossbar switches that contain hundreds of pins are implemented using the technique of multistage interconnection networks that is discussed in the next section of the lecture.

LECTURE: 7

Multistage Interconnection (Omega) Networks:

Multistage connection networks are designed with the use of small elementary crossbar switches (usually they have two inputs) connected in multiple layers. The elementary crossbar switches can implement 4 types of connections: straight, crossed upper broadcast and lower broadcast. All elementary switches are controlled simultaneously. The network like this is an alternative for crossbar switches if we have to switch a large number of connections, over 100. The extension cost for such a network is relatively low.

In such networks, there is no full freedom in implementing arbitrary connections when some connections have already been set in the switch. Because of this property, these networks belong to the category of so called **blocking networks.**

However, if we increase the number of levels of elementary crossbar switches above the number necessary to implement connections for all pairs of inputs and outputs, it is possible to implement all requested connections at the same time but statically, before any communication is started in the switch. It can be achieved at the cost of additional redundant hardware included into the switch. The block diagram of such a network, called the Benes network, is shown in the figure below.

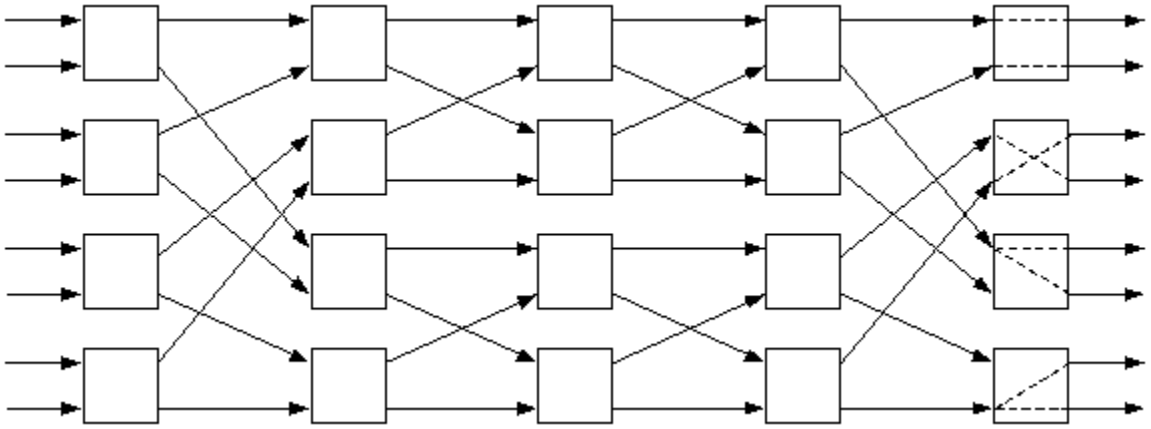


Figure: 3.8 Multistage Connection Network For Parallel Systems.

To obtain non blocking properties of the multistage connection network, the redundancy level in the circuit should be much increased. To build a non blocking multistage network ***n*** *x* ***n***, the elementary two-input switches have to be replaced by 3 layers of switches ***n*** *x* ***m***, ***r*** *x* ***r*** and ***m*** *x* ***n***, where ***m , 2n - 1*** and ***r*** is the number of elementary switches in the layer 1 and 3. Such a switch

was designed by a French mathematician Clos and it is called the **Clos network.** This switch is commonly used to build large integrated crossbar switches. The block diagram of the Clos network is shown in the figure below.

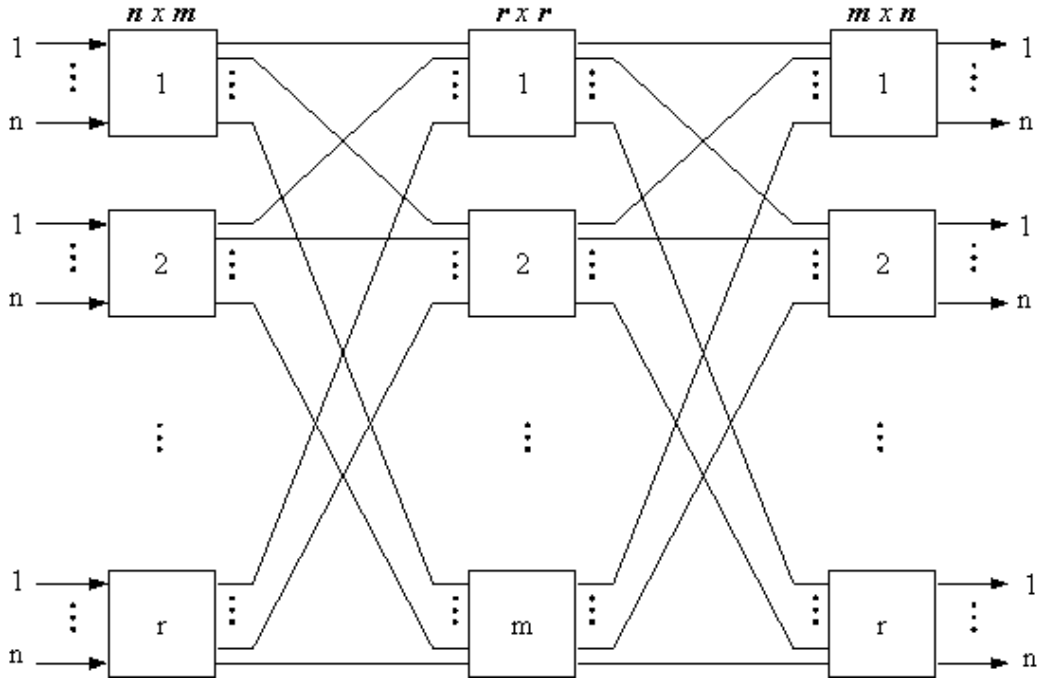


Figure: 3.9 A non-blocking Clos interconnection network

LECTURE: 8

Baseline Network:

Baseline network is one of the important interconnection networks employed in parallel computing systems. Baseline network is a type of permutation network, which connects an equal number of inputs and outputs and realizes a set of permutations. In the Baseline network, the maximum number of allowable permutations is 2n *N/2, where n is the number of switching stages (n = log2N) and each switch has two inputs and two outputs. Fig. 5.30 depict Baseline networks.

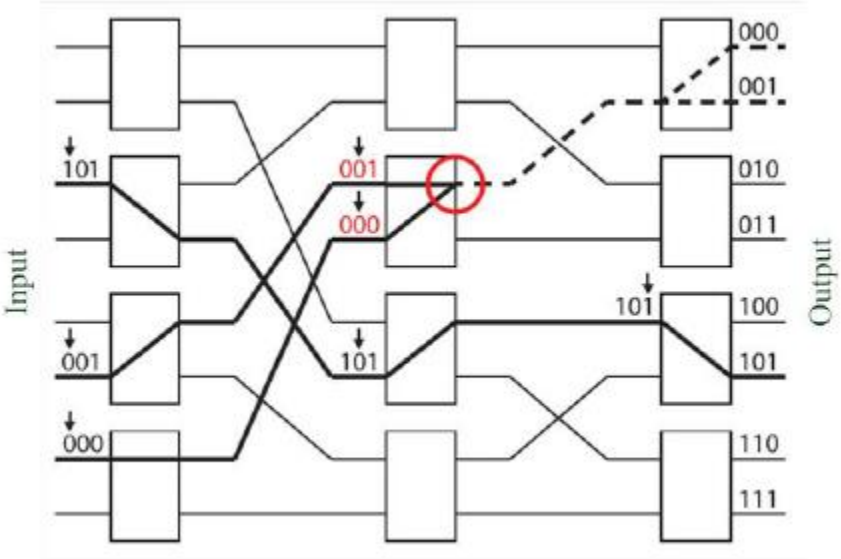


Figure: 3.9 8x8 Baseline network

Butterfly Network:

A **butterfly network** is a computer science technique to link multiple computers into a high-speed computing network. This form of multistage interconnection network topology can be used to connect different nodes in a multiprocessor system. The interconnect network for a shared memory multiprocessor system must have low latency and high bandwidth compared to other network systems, like local area networks (LANs) or internet. Multiprocessor systems must have low latency and high bandwidth for three reasons: (1) Messages are relatively short as most messages consist of coherence protocol requests and responses without data. (2) Messages are generated frequently because each read or write miss generates messages to every node in the system to ensure coherence. Read or write misses occur when the requested data is not in the

processor's cache and must be fetched from either memory or another processor's cache. (3) Messages are generated frequently, therefore rendering it difficult for processors to hide the communication delay.

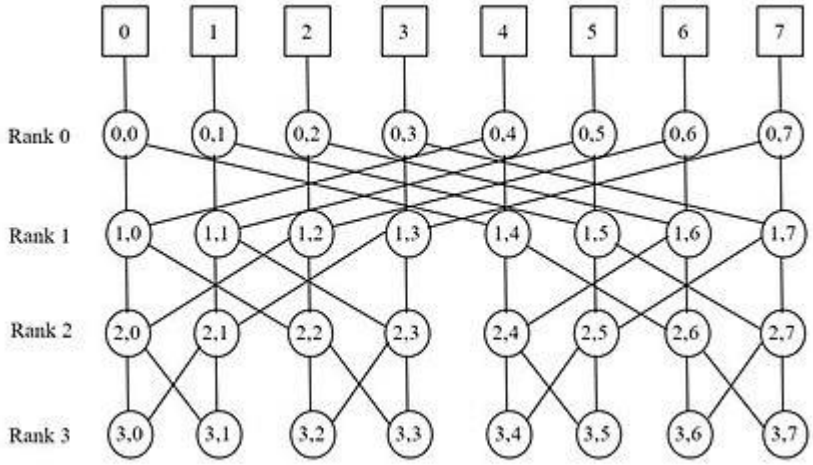


Figure: 3.10: Butterfly Network for 8 processors.

The major components of an interconnect network are:

- Processor Nodes which consist of one or more processors along with their caches, memories and communication assist.
- Switching Nodes (Router) which connect communication assist of different processor nodes in a system. In multistage topologies, higher level switching nodes connect to lower level switching nodes as shown in figure 1, where switching nodes in rank 0 connect to processor nodes directly while switching nodes in rank 1 connect to switching nodes in rank 0.
- Links which are physical wires between two switching nodes (routers). They can be uni-directional or bi-directional.

These multistage networks have lower cost than a cross bar but still obtain lower contention than a bus. The ratio of switching nodes to processor nodes is greater than one in a butterfly network. Such topology where the ratio of switching nodes to processor nodes is greater than one is called an indirect topology.

The network derived its name from connections between nodes in two adjacent ranks (as shown in figure 5.31), which resembles a butterfly. When top and bottom ranks are merged into a single rank, it is called a *Wrapped Butterfly Network*. In figure 5.31, if rank 3 nodes are connected back to respective rank 0 nodes, then it becomes a wrapped butterfly network.

BBN Butterfly, a massive parallel computer built by Bolt, Beranek and Newman in the 1980s, used a butterfly interconnect network. Later in 1990, Cray Research's machine Cray C90, used a butterfly network to communicate between its 16 processors and 1024 memory banks.

LECTURE 9

Butterfly network building:

For a butterfly network with 'p' processor nodes, there needs to be p ($\log_2$ p + 1) switching nodes. Figure 5.31 shows a network with 8 processor nodes, which means there are 32 switching nodes. It also represents each node as N (rank, column number). For example, node at column 6 in rank 1 is represented as (1, 6) and node at column 2 in rank 0 is represented as (0, 2).

For any 'i' greater than zero, a switching node N (i,j) gets connected to N(i-1, j) and N(i-1, m), where 'm' is obtained by flipping the i^{th} most significant bit of j. For example, consider the node N (1,6): i equals 1 and j equals 6, therefore m is obtained by flipping the first most significant bit of 6.

Variable	Binary representation	Decimal Representation
j	110	6
m	010	2

Table 3.1

As a result, the nodes connected to N (1,6) are :-

N(i,j)	N(i-1,j)	N(i-1,m)
(1,6)	(0,6)	(0,2)

Table 3.2

Thus, N (0,6), N(1,6), N(0,2), N(1,2) form a butterfly pattern. Several butterfly patterns exist in the figure and therefore, this network is called a Butterfly Network.

CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.

- Programmer responsibility for synchronization constructs that insure "correct" access of global memory.

- Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

Name of the Paper: Advanced Computer Architecture
Paper Code: CS802D
Contact (Periods/Week):3L/Week
Credit Point: 3
No. of Lectures: 35

Module -4: Shared Memory Architecture [4L]

Fundamentals of UMA, NUMA, NORMA,COMA architectures, Performance measurement for parallel architectures –Amadahl's law, Gustafson's law

Module -4: Shared Memory Architecture [4L]

LECTURE 1

Multiprocessor:
A multiprocessor system is a computer system comprising of two or more processor. An interconnection network links this processor. The primary objective of multiprocessor system is to enhance the performance by means of parallel processing. It falls under MIMD architecture.

Besides providing high performance, the multiprocessor also offers the following benefits:

1. Fault tolerance and graceful degradation.
2. Scalability and modular growth.

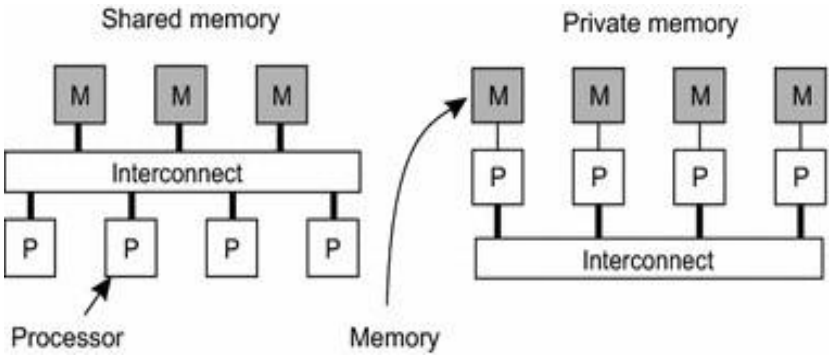


Figure 4.1 Multiprocessor systems.

Classification:

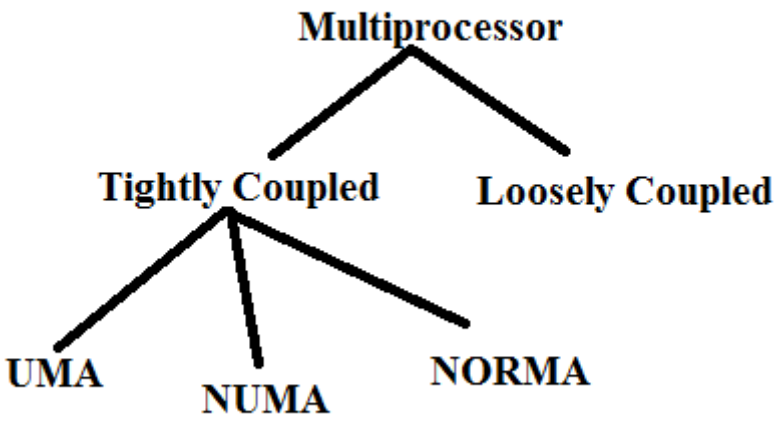


Figure 4.2 Classification of Multiprocessor

Tightly Coupled Multiprocessor System: In tightly coupled multiprocessor; the multiple processor share information by a common memory (Global Memory).Hence, this type is also known as shared memory multiprocessor system. Beside sharing the global memory dedicated to its which cannot be accessed by other processors in the system.

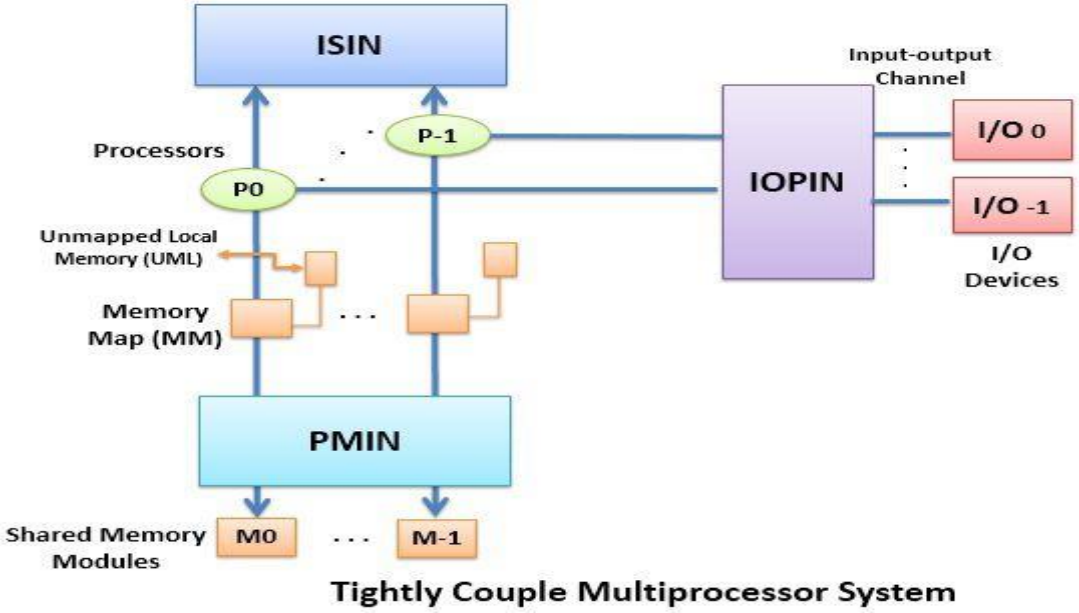


Figure: 4.3

LECTURE 2

Loosely Coupled Multiprocessor System: In loosely coupled multiprocessor system memory is not shared and each processor has its own memory. This type of a system is known as distributed memory multiprocessor system. The information is exchanged network by a common message passing protocol.

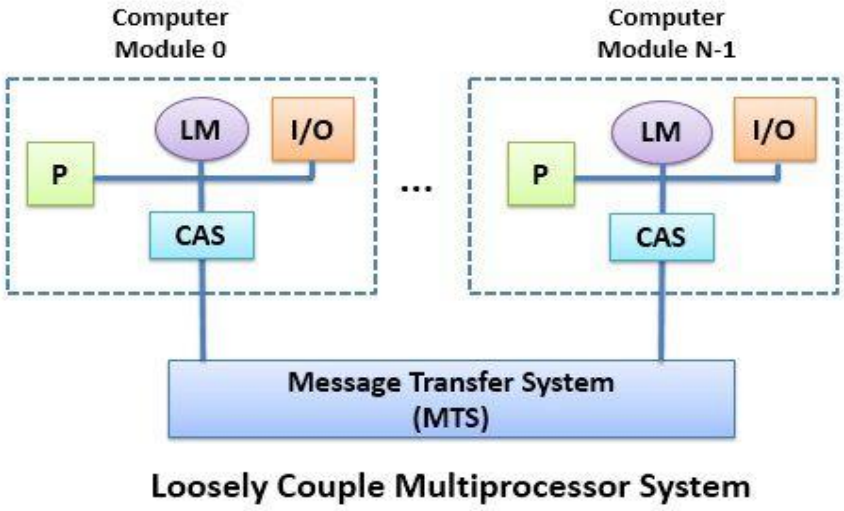


Figure: 4.4 Loosely Coupled Multiprocessor System

Uniform Memory Access:

Uniform memory access (UMA) is a shared memory architecture used in parallel computers. All the processors in the UMA model share the physical memory uniformly. In UMA architecture, access time to a memory location is independent of which processor makes the request or which memory chip contains the transferred data. Uniform memory access computer architectures are often contrasted with non-uniform memory access (NUMA) architectures. In the UMA architecture, each processor may use a private cache. Peripherals are also shared in some fashion. The UMA model is suitable for general purpose and time sharing applications by multiple users. It can be used to speed up the execution of a single large program in time critical applications.

In a uniform memory access system the access time of memory is equal for all processor. A symmetric multiprocessor is UMA multiprocessor system with identical processors, equally capable of performing similar function in a identical manner. All the processors have equal access time for the memory and I/O resources.

Types of UMA architectures:

1. UMA using bus-based symmetric multiprocessing (SMP) architectures.
2. UMA using crossbar switches.

3. UMA using multistage interconnection networks.

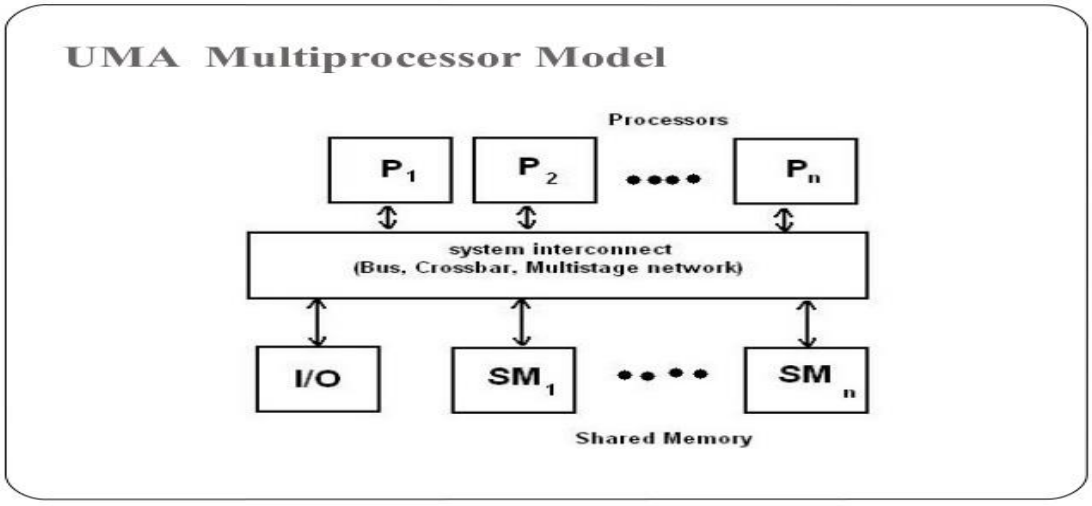


Figure: 4.5 UMA architectures

LECTURE: 3

Non-Uniform Memory Access:

Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors). The benefits of NUMA are limited to particular workloads, notably on servers where the data are often associated strongly with certain tasks or users.

NUMA architectures logically follow in scaling from symmetric multiprocessing (SMP) architectures. They were developed commercially during the 1990s by Burroughs (later Unisys), Convex Computer (later Hewlett-Packard), Honeywell Information Systems Italy (HISI) (later Groupe Bull), Silicon Graphics (later Silicon Graphics International), Sequent Computer Systems (later IBM), Data General (later EMC), and Digital (later Compaq, now HP). Techniques developed by these companies later featured in a variety of Unix-like operating systems, and to an extent in Windows NT. The first commercial implementation of a NUMA-based UNIX system was the Symmetrical Multi Processing XPS-100 family of servers, designed by Dan Gielan of VAST Corporation for Honeywell Information Systems Italy.

Modern CPUs operate considerably faster than the main memory they use. In the early days of computing and data processing, the CPU generally ran slower than its own memory. The performance lines of processors and memory crossed in the 1960s with the advent of the first supercomputers. Since then, CPUs increasingly have found themselves "starved for data" and having to stall while waiting for data to arrive from memory. Many supercomputer designs of the 1980s and 1990s focused on providing high-speed memory access as opposed to faster processors, allowing the computers to work on large data sets at speeds other systems could not approach.

Limiting the number of memory accesses provided the key to extracting high performance from a modern computer. For commodity processors, this meant installing an ever-increasing amount of high-speed cache memory and using increasingly sophisticated algorithms to avoid cache misses. But the dramatic increase in size of the operating systems and of the applications run on them has generally overwhelmed these cache-processing improvements. Multi-processor systems without NUMA make the problem considerably worse. Now a system can starve several processors at the same time, notably because only one processor can access the computer's memory at a time. NUMA attempts to address this problem by providing separate memory for each processor, avoiding the performance hit when several processors attempt to address the same memory. For problems involving spread data(common for servers and similar applications), NUMA can improve the performance over a single shared memory by a factor of roughly the number of processors (or separate memory banks). Another approach to addressing this problem, utilized mainly by non-NUMA systems, is the multi-channel memory architecture; multiple memory channels are increasing the number of simultaneous memory accesses.

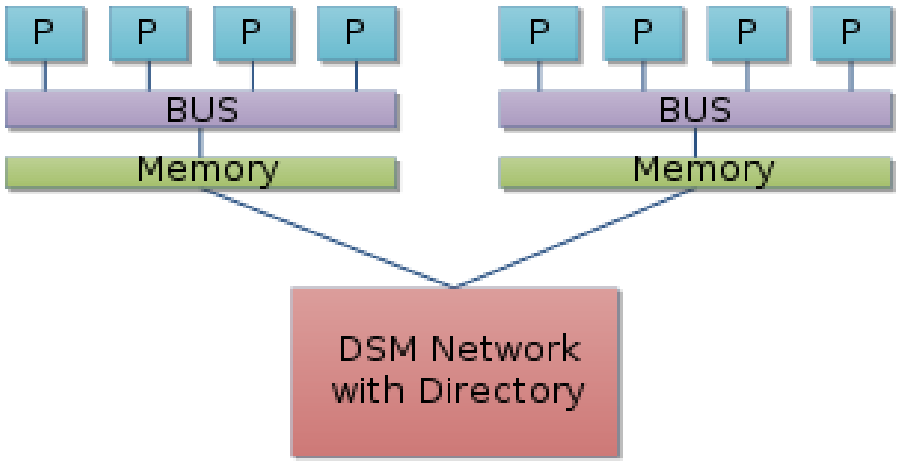


Figure: 4.6 Architecture of a NUMA system.

No-Remote Memory Access

No Remote Memory Access (NORMA) is a computer memory architecture for multiprocessor system.

In NORMA architecture, the address space globally is not unique and the memory is not globally accessible by the processor.

Accesses to remote memory modules are only indirectly possible by message through the interconnection network to other processors, which in turn possibly deliver the desired data in a reply message.

Two categories of parallel computers are discussed below namely shared common memory or unshared distributed memory.

Shared memory multiprocessors

Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.

- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: ***UMA, NUMA and COMA.***

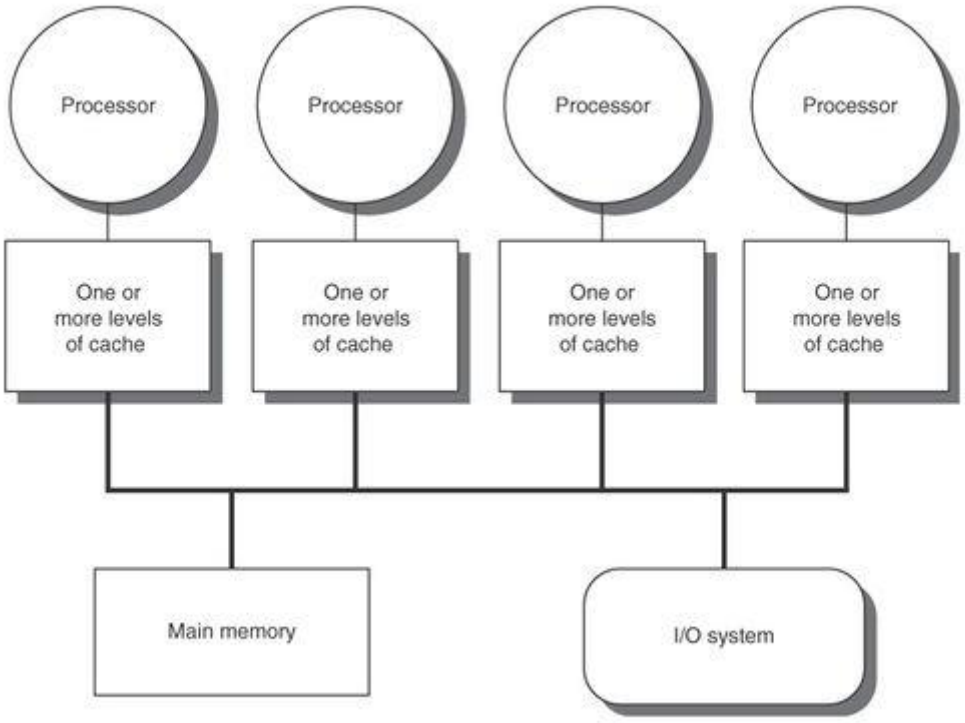


Figure: 4.7 Shared memory multiprocessors

Advantages:

- Global address space provides a user-friendly programming perspective to memory.
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs.

Disadvantages:

• Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory.

LECTURE 4

AMDAHL'S LAW

The theory of doing computational work in parallel has some fundamental laws that place limits on the benefits one can derive from parallelizing a computation (or really, any kind of work). To understand these laws, we have to first define the objective. In general, the goal in large scale computation is to get as much work done as possible in the shortest possible time within our budget. We ``win'' when we can do a big job in less time or a bigger job in the same time and not go broke doing so. The ``power'' of a computational system might thus be usefully defined to be the amount of computational work that can be done divided by the time it takes to do it, and we generally wish to optimize power per unit cost, or cost-benefit.

Physics and economics conspire to limit the raw power of individual single processor systems available to do any particular piece of work even when the dollar budget is effectively unlimited. The cost-benefit scaling of increasingly powerful single processor systems is generally nonlinear and very poor - one that is twice as fast might cost four times as much, yielding only half the cost-benefit, per dollar, of a cheaper but slower system. One way to increase the power of a computational system (for problems of the appropriate sort) past the economically feasible single processor limit is to apply more than one computational engine to the problem.

This is the motivation for Beowulf design and construction; in many cases a Beowulf may provide access to computational power that is available in a alternative single or multiple processor designs, but only at a far greater cost.

In a perfect world, a computational job that is split up among N processors would complete in 1/N time, leading to an N -fold increase in power. However, any given piece of parallelized work to be done will contain parts of the work that *must* be done serially, one task after another, by a single processor. This part does *not* run any faster on a parallel collection of processors (and might even run more slowly). Only the part that can be parallelized runs as much as N-fold faster.

The ``speedup'' of a parallel program is defined to be the ratio of the rate at which work is done (the power) when a job is run on N processors to the rate at which it is done by just one. To simplify the discussion, we will now consider the ``computational work'' to be accomplished to be an arbitrary task (generally speaking, the particular problem of greatest interest to the reader). We can then define the speedup (increase in power as a function of N) in terms of the time required to complete this particular fixed piece of work on 1 to N processors.

Let T(N)be the time required to complete the task on N processors. The speedup S(N) is the ratio

In many cases the time T(1) has, as noted above, both a serial portion T_s and a parallelizable portion T_p. The serial time does not diminish when the parallel part is split up. If one is

"optimally" fortunate, the parallel time is decreased by a factor of 1/N. The speedup one can expect is thus

$$S(N) = \frac{T(1)}{T(N)} = \frac{T_s + T_p}{T_s + T_p/N}. \quad (2)$$

This elegant expression is known as *Amdahl's Law* and is usually expressed as an inequality. This is in almost all cases the *best* speedup one can achieve by doing work in parallel, so the real speed up $S(N)$ is less than or equal to this quantity.

Amdahl's Law immediately eliminates many, many tasks from consideration for parallelization. If the serial fraction of the code is not much smaller than the part that could be parallelized (if we rewrote it and were fortunate in being able to split it up among nodes to complete in less time than it otherwise would), we simply won't see much speedup no matter how many nodes or how fast our communications. Even so, Amdahl's law is still far too optimistic. It ignores the overhead incurred due to parallelizing the code. We must generalize it.

A fairer (and more detailed) description of parallel speedup includes at least two more times of interest:

$\mathbf{T_s}$ The original single-processor serial time.

$\mathbf{T_{is}}$ The (average) additional *serial* time spent doing things like interprocessor communications (IPCs), setup, and so forth in all parallelized tasks. This time can depend on N in a variety of ways, but the simplest assumption is that each system has to expend this much time, one after the other, so that the total additional serial time is for example $N * T_{is}$.

$\mathbf{T_p}$ The original single-processor parallelizeable time.

$\mathbf{T_{ip}}$ The (average) *additional* time spent by each processor doing just the setup and work that it does in parallel. This may well include idle time, which is often important enough to be accounted for separately.

It is worth remarking that generally, the most important element that contributes to T_{is} is the time required for communication between the parallel subtasks. This communication time is always there - even in the simplest parallel models where identical jobs are farmed out and run in parallel on a cluster of networked computers, the remote jobs must be begun and controlled with messages passed over the network. In more complex jobs, partial results developed on each CPU may have to be sent to all other CPUs in the beowulf for the calculation to proceed, which can

be *very* costly in scaled time. As we'll see below, T_{is} in particular plays an extremely important role in determining the speedup scaling of a given calculation. For this (excellent!) reason many beowulf designers and programmers are obsessed with communications hardware and algorithms.

It is common to combine T_{ip}, N and T_{is} into a single expression $T_o(N)$ (the ``overhead time") which includes any complicated $N$-scaling of the IPC, setup, idle, and other times associated with the overhead of running the calculation in parallel, as well as the scaling of these quantities with respect to the ``size" of the task being accomplished. The description above (which we retain as it illustrates the generic form of the relevant scalings) is still a *simplified* description of the times - real life parallel tasks can be much more complicated, although in many cases the description above is adequate.

Using these definitions and doing a bit of algebra, it is easy to show that an improved (but still simple) estimate for the parallel speedup resulting from splitting a particular job up between N nodes (assuming one processor per node) is:

$$S(N) = \frac{T_s + T_p}{T_s + N * T_{is} + T_p/N + T_{ip}}. \tag{3}$$

This expression will suffice to get at least a general feel for the scaling properties of a task that might be parallelized on a typical beowulf.

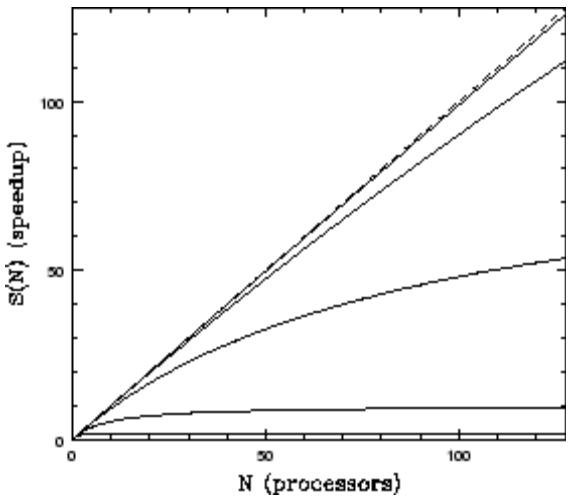


Figure 4.8: $T_{is} = 0$ and $T_p =$ 10, 100, 1000, 10000, 100000 (in increasing order).

It is useful to plot the dimensionless ``real-world speedup" (3) for various *relative* values of the times. In all the figures below, T_s = 10 (which sets our basic scale, if you like) and T_p = 10,

100, 1000, 10000, 100000 (to show the systematic effects of parallelizing more and more work compared to T_s).

Gustafson's law

In 1988, Gustafson and Barsis at Sandia Laboratories studied the paradox created by Amdahl's law and the fact that parallel architectures comprised of hundreds of processors were built with substantial improvement in performance. In introducing their law, Gustafson recognized that the fraction of indivisible tasks in a given algorithm might not be known a priori. They argued that in practice, the problem size scales with the number of processors, n. This contradicts the basis of Amdahl's law. Recall that Amdahl's law assumes that the amount of time spent on the parts of the program that can be done in parallel, (1 2 f), is independent of the number of processors, n. Gustafson and Brasis postulated that when using a more powerful processor, the problem tends to make use of the increased resources. They found that to a first approximation the parallel part of the program, not the serial part, scales up with the problem size. They postulated that if s and p represent respectively the serial and the parallel time spent on a parallel system, then sþpn represents the time needed by a serial processor to perform the computation. They therefore, introduced a new factor, called the scaled speedup factor, SS(n), which can be computed as:

$$SS(n) = \frac{s + p \times n}{s + p} = s + p \times n = s + (1 - s) \times n = n + (1 - n) \times s$$

This equation shows that the resulting function is a straight line with a slope¼(1 2 n). This shows clearly that it is possible, even easier, to achieve efficient parallel performance than is implied by Amdahl's speedup formula. Speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size. Having considered computational models and rebutted some of the criticism set forth by a number of computer architects in the face of using parallel architectures, we now move to consider some performance issues in dynamic and static interconnection networks. The emphasis will be on the performance of the interconnection networks rather than the computational aspects of the processors.

Module – 5: Embedded System Architecture [5L]