

# Instruction Pipeline Design - Instruction Execution Phases

A G Sreejith

November 23, 2018

# Instruction Execution Phases

- ① Instruction Fetch
- ② Decode instruction
- ③ Operand Fetch
- ④ Execute
- ⑤ write-back

# Instruction Fetch

## Instruction Fetch

fetches instructions from a cache memory, presumably one per cycle

# Decode instruction

## Decode instruction

reveals the instruction function to be performed and identifies the resources needed. Resources include general-purpose registers, buses, and functional units

# Operand Fetch

## Operand Fetch

Fetch operands from memory if necessary: If any operands are memory addresses, initiate memory read cycles to read them into CPU registers.

## Execute

### Execute

Perform the function of the instruction. If arithmetic or logic instruction, utilize the ALU circuits to carry out the operation on data in registers.

# write-back

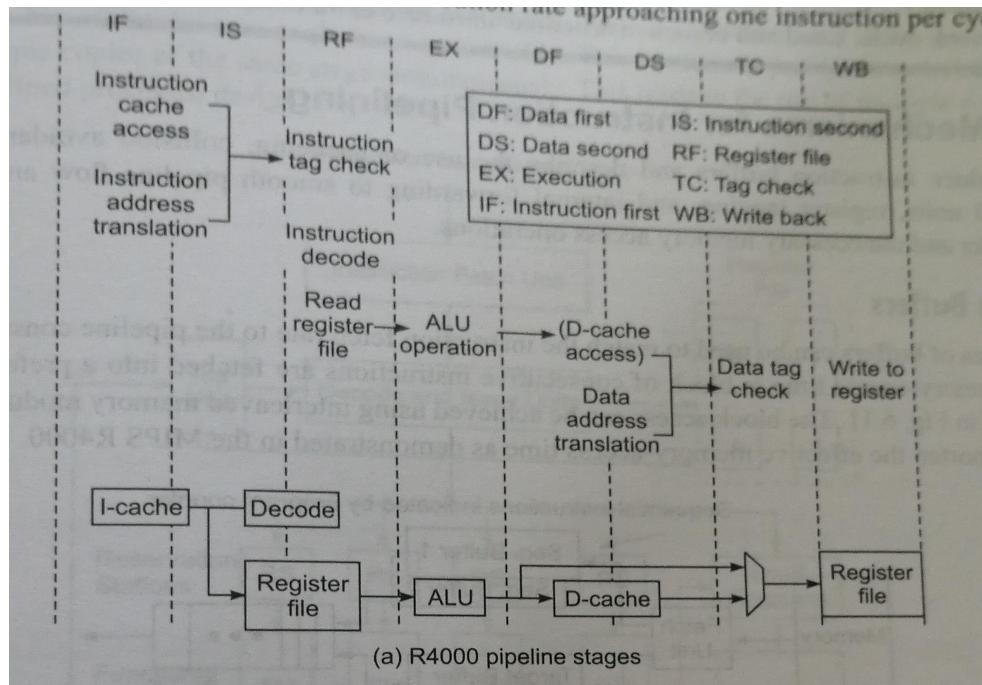
## write-back

write-back stage is used to write results into the registers. Memory load or store operations are treated as part of execution.

# MIPS R4000 Instruction Pipeline

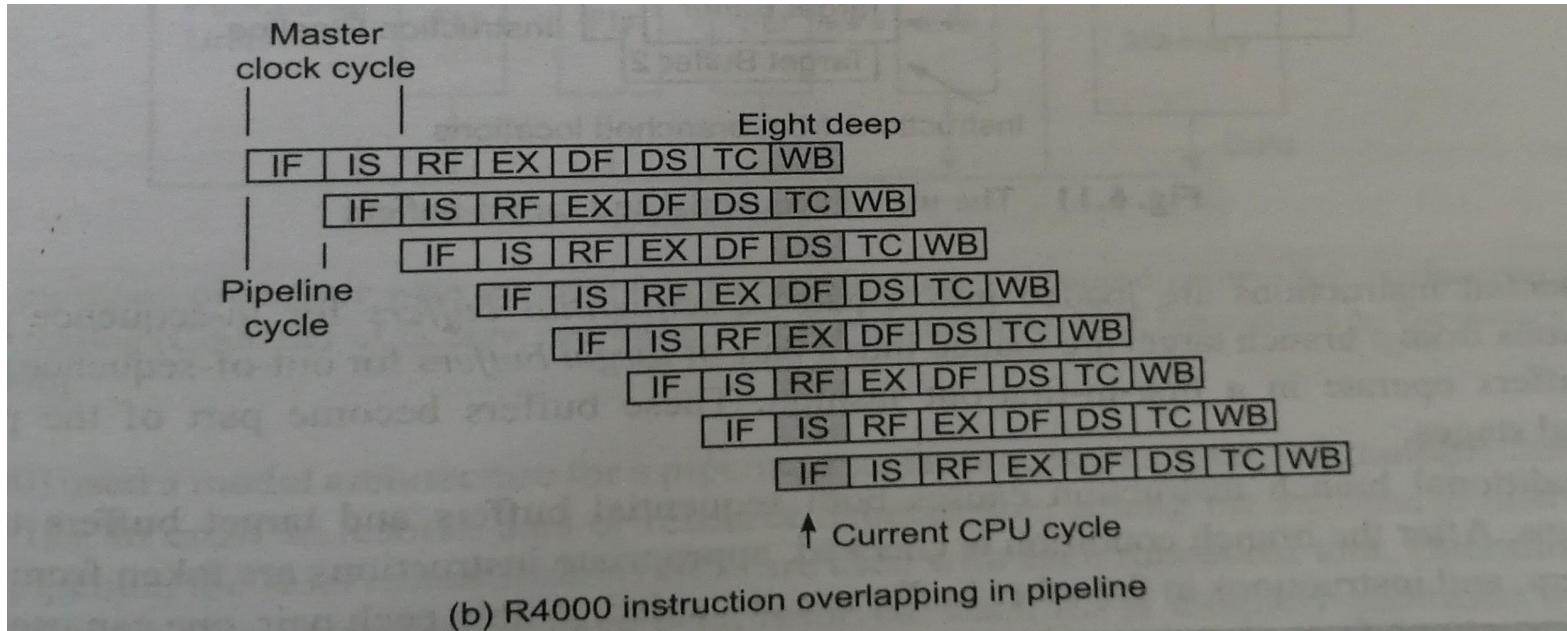
- MIPS R4000 was a pipelined 64-bit processor using separate instruction and data caches
- Eight-Stage pipeline for executing register-based instructions
- The processor pipeline design was targeted to achieve an execution rate approaching one instruction per cycle

# R4000 Pipeline Stages



- The execution of each R4000 instruction consisted of eight major steps
- Each of these steps required approximately one clock cycle
- The instruction and data memory references are split across two stages.
- The single-cycle ALU stage took slightly more time than each of the cache access stages.

# R4000 Instruction Overlapping in Pipeline



- This pipeline operated efficiently because different CPU resources,such as address and bus access,ALU operations,register access and so on,were utilized simultaneously on a non interfering basis
- The internal pipeline clock rate(100 Mhz) of the R4000 was twice the external input or master clock frequency
- Load and branch instructions introduce extra delays

# PREFETCH BUFFERS

MECHANISM FOR INSTRUCTION PIPELINING

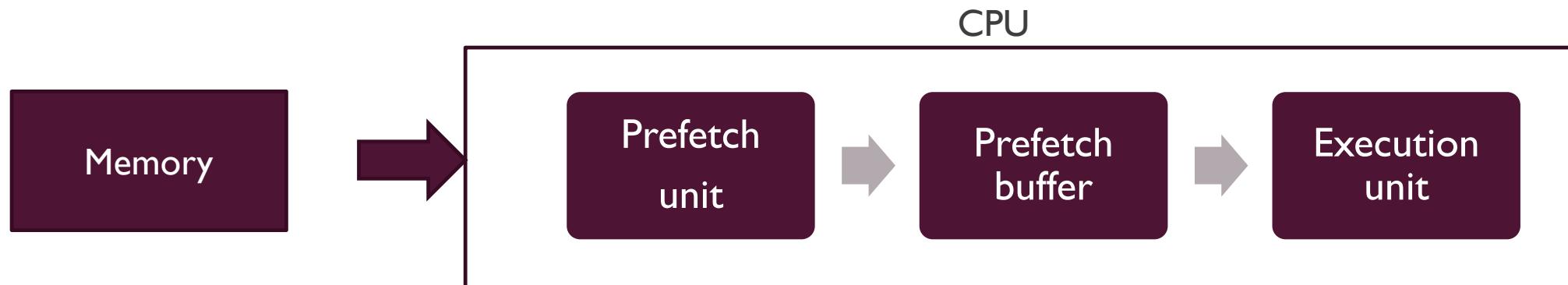
Abhirami P H  
CSE S7  
MUT15CS003

# INSTRUCTION PREFETCH

- Technique which attempts to minimize the time a processor spends waiting for machine instruction to be fetched from memory.
- Instructions following the one being currently executed are loaded into a prefetch queue when the processor's external bus is otherwise idle.
- Instruction prefetch is combined with pipelining in an attempt to keep the pipeline busy.
- By 1995, most processors used prefetching. E.g. Motorola 680x0, intel 80x86.

# INSTRUCTION LEVEL PARALLELISM-INSTRUCTION PREFETCH

- Break up the fetch-execute cycle and do the two in parallel.
- This dates to the IBM stretch (1959).



## CONTD..

- The prefetch buffer is implemented in the CPU with on-chip registers.
- The prefetch buffer is implemented as a single register or a queue.
- Think of the prefetch buffer as containing the IR; When the execution of one instruction completes, the next one is already in the buffer and does not need to be fetched.
- Naturally, a program branch(loop structure, conditional branch etc.) invalidates the contents of the prefetch buffer, which needs to be reloaded.



# **Mechanisms for Instruction Pipelining - Multiple Functional Units**

**AKHIL GOKULDAS  
ROLL NO :- 4  
CSE - S7**



# **What is instruction pipelining?**

**Instruction pipelining** is a technique used in the design of modern microprocessors, microcontrollers and CPUs to increase their instruction throughput (the number of instructions that can be executed in a unit of time).



# Concept & Working

1. The main idea is to divide the processing of a CPU instruction, as defined by the instruction **microcode**, into a series of independent steps of micro-operations (also called "**microinstructions**", "**micro-op**" or " **$\mu$ op**"), with storage at the end of each step.
2. This allows the CPUs control logic to handle instructions at the processing rate of the slowest step, which is much faster than the time needed to process the instruction as a single step.



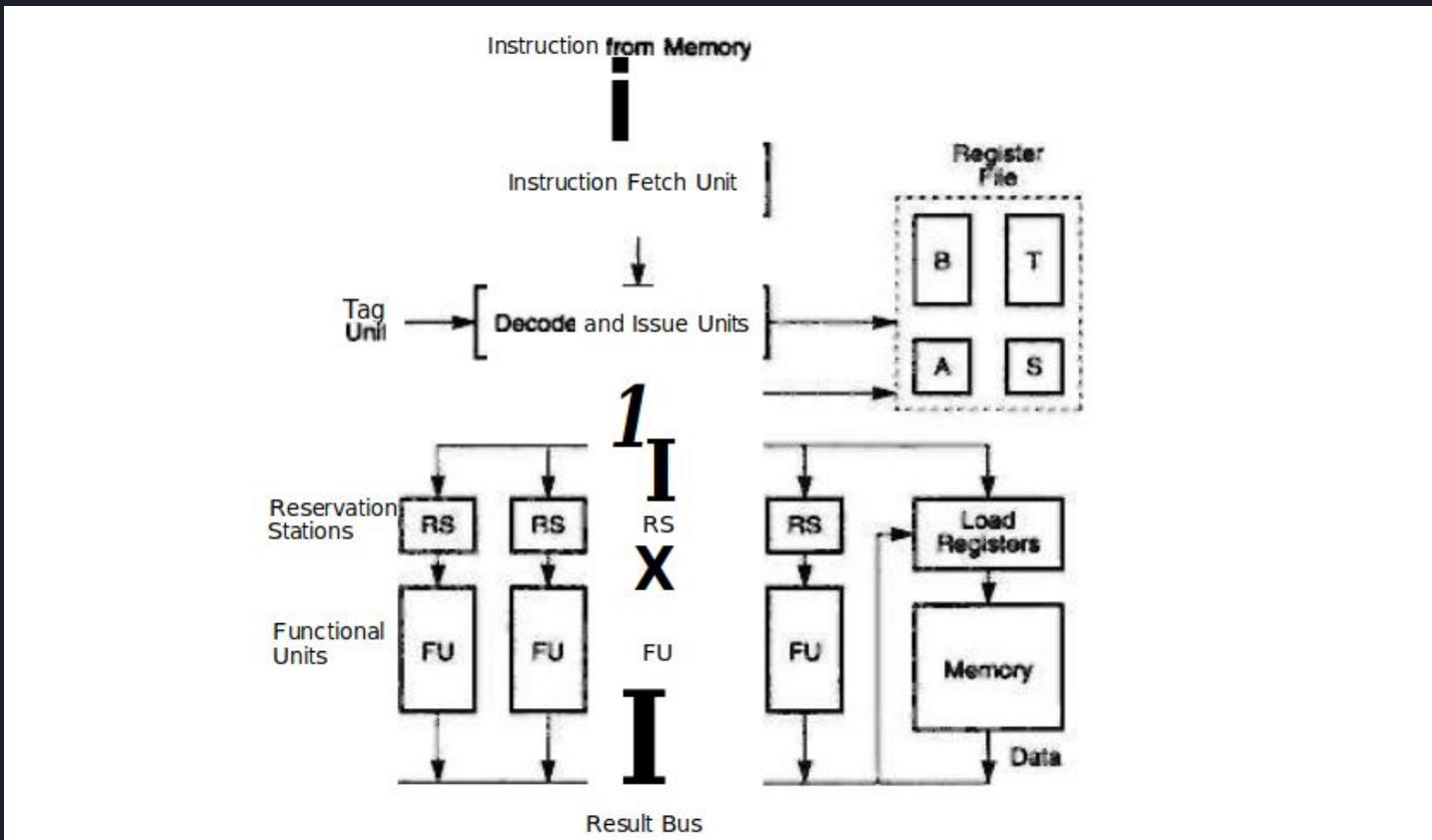
# Background

Most modern CPUs are driven by a clock. The CPU consists internally of logic and memory (flip flops). When the clock signal arrives, the flip flops store their new value then the logic requires a period of time to decode the flip flops new values. Then the next clock pulse arrives and the flip flops store another values, and so on. By breaking the logic into smaller pieces and inserting flip flops between pieces of logic, the time required by the logic (to decode values till generating valid outputs depending on these values) is reduced.



# Multiple Functional Units

- sometimes a certain pipeline stage becomes the bottleneck.
- This stage corresponds to the row with the maximum number of checkmarks in the reservation table.
- This bottleneck problem can be alleviated by using multiple copies of the same stage simultaneously.
- This leads to the use of multiple execution units in a pipelined processor design .



A pipelined processor with multiple functional units and distributed reservation stations supported by tagging.



# Continuation

a model architecture for a pipelined scalar processor containing multiple functional units (Fig. 6.12). In order to resolve data or resources dependences among the successive instructions entering the pipeline, the reservation stations (RS) are used with each functional unit. Operands can wait in the RS until its data dependence have been resolved. Each RS is uniquely identified by a tag, which is monitored by a tag unit.



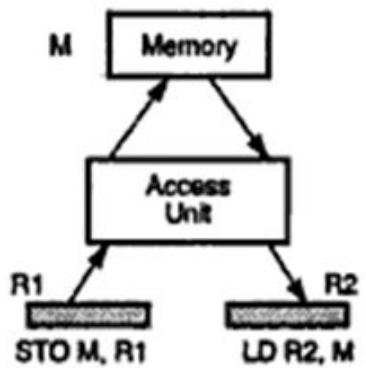
**THANKYOU**

# Internal Data Forwarding

CSA Assignment

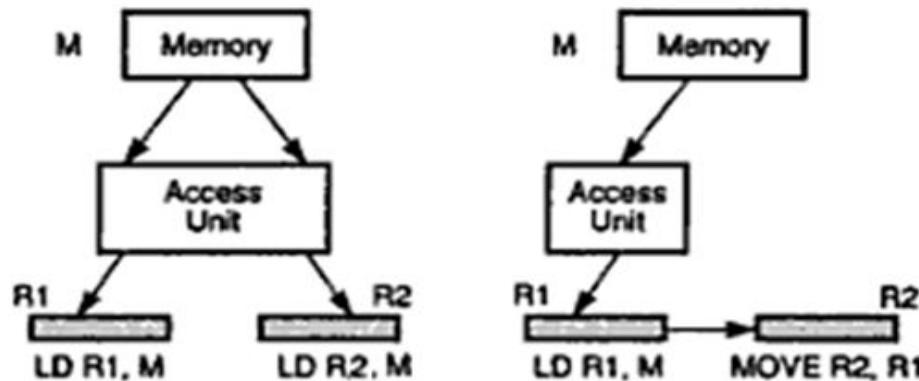
- An optimization technique to speed up the performance and overcome data hazards.
- Replacing unnecessary memory accesses by register-to-register transfers.
- After each stage in pipeline there are buffers to store the intermediate results.
- Take the result from the earliest point that it exists in any of the pipeline state registers and forward it to the functional units (e.g., the ALU) that need it that cycle
- Concept is explained in 3 directions: store-load, load-load & store-store forwarding.

# Store – Load Forwarding

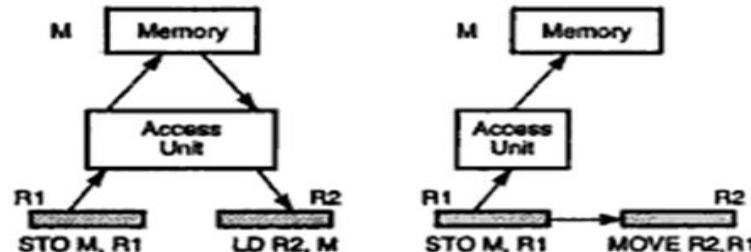


(a) Store-load forwarding

# Load – Load Forwarding



# Store – Load Forwarding



(a) Store-load forwarding

- Forwarding can achieve a CPI of 1 even in the presence of data dependencies

# Mechanisms for Instruction

## Pipelining

## Hazard Avoidance

# Generic

## Pipeline bubbling

- *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.

# Data hazards

There are several main solutions and algorithms used to resolve data hazards:

- insert a *pipeline bubble* whenever a read after write (RAW) dependency is encountered, guaranteed to increase latency, or
- use out-of-order execution to potentially prevent the need for pipeline bubbles
- use *operand forwarding* to use data from later stages in the pipeline

In the case of out-of-order execution, the algorithm used can be:

- scoreboard, in which case a *pipeline bubble* is needed only when there is no functional unit available
- the Tomasulo algorithm, which uses register renaming, allowing continual issuing of instructions.

The task of removing data dependencies can be delegated to the compiler, which can fill in an appropriate number of NOP instructions between dependent instructions to ensure correct operation, or re-order instructions where possible.

# Control hazards (branch hazards)

To avoid control hazards microarchitectures 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

In the event that a branch causes a pipeline bubble after incorrect instructions have entered the pipeline, care must be taken to prevent any of the wrongly-loaded instructions from having any effect on the processor state, excluding energy wasted processing them before they were discovered to be loaded incorrectly.

# Other techniques

- Memory latency is another factor that designers must attend to, because the delay could reduce performance.
- Different types of memory have different accessing time to the memory.
- Thus, by choosing a suitable type of memory, designers can improve the performance of the pipelined data path.

# Tomasulo algorithm

# Introduction

- Tomasulo's algorithm is a computer architecture hardware algorithm for dynamic scheduling of instructions that allows out-of-order execution and enables more efficient use of multiple execution units.
- It was developed by Robert Tomasulo at IBM in 1967 and was first implemented in the IBM System/360 Model 91's floating point unit.

# Implementation concepts

- Common data bus
- Instruction order
- Register renaming
- Exceptions

# Instruction lifecycle

- Stage 1: issue

In the issue stage, instructions are issued for execution if all operands and reservation stations are ready or else they are stalled. Registers are renamed in this step, eliminating WAR and WAW hazards.

## Stage 2: execute

In the execute stage, the instruction operations are carried out. Instructions are delayed in this step until all of their operands are available, eliminating RAW hazards. Program correctness is maintained through effective address calculation to prevent hazards through memory.

## Stage 3: write result

In the write Result stage, ALU operations results are written back to registers and store operations are written back to memory

# Tomasulo algorithm

# Introduction

- Tomasulo's algorithm is a computer architecture hardware algorithm for dynamic scheduling of instructions that allows out-of-order execution and enables more efficient use of multiple execution units.
- It was developed by Robert Tomasulo at IBM in 1967 and was first implemented in the IBM System/360 Model 91's floating point unit.

# Implementation concepts

- Common data bus
- Instruction order
- Register renaming
- Exceptions

# Instruction lifecycle

- Stage 1: issue

In the issue stage, instructions are issued for execution if all operands and reservation stations are ready or else they are stalled. Registers are renamed in this step, eliminating WAR and WAW hazards.

## Stage 2: execute

In the execute stage, the instruction operations are carried out. Instructions are delayed in this step until all of their operands are available, eliminating RAW hazards. Program correctness is maintained through effective address calculation to prevent hazards through memory.

## Stage 3: write result

In the write Result stage, ALU operations results are written back to registers and store operations are written back to memory

# Branching Techniques

## And

## It's Effects

Amal Binoy  
S7 CSE  
10

# Branching Techniques

- Branching is a basic concept in computer science. It means an instruction that tells a computer to begin executing a different part of a program rather than executing statements one-by-one.
- Branching is implemented as a series of control flow statements in high-level programming languages. These can include:
- If statements
- For loops
- While loops
- Goto statements .

Branching instructions are also implemented at the CPU level, though they are much less sophisticated than the kinds of instructions found in high-level languages. These instructions are accessed through assembly programming and are also referred to as "jump" instructions.

# **Effect Of Branching**

One of the major problems in designing an instruction pipe line is assuming a steady flow of instructions to the initial stages of the pipeline.

The primary problem is the conditional branches instruction until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not.

A variety of approaches have been taken for dealing with conditional branches:

- Multiple streams
- Prefetch branch target
- Loop buffer
- Branch prediction
- Delayed branch

- **Multiple streams**

A single pipeline suffers a penalty for a branch instruction because it must choose one of two instructions to fetch next and sometimes it may make the wrong choice. A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams.

- **Prefetch Branch target**

When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This target is then saved until the branch instruction is executed. If the branch is taken, the target has already been prefetched.

- **Loop Buffer:**

A top buffer is a small, very high speed memory maintained by the instruction fetch stage of the pipeline and containing the most recently fetched instructions, in sequence. The loop buffer is similar in principle to a cache dedicated to instructions. The differences are that the loop buffer only retains instructions in sequence and is much smaller in size and hence lower in cost.

- **Branch Prediction:**

Various techniques can be used to predict whether a branch will be taken or not. The most common techniques are:

- Predict never taken
- Predict always taken
- Predict by opcode
- Taken/not taken switch
- Branch history table.

- **Delayed branch**

A conditional branch instruction found in some RISC architectures that include pipelining. The effect is to execute one or more instructions following the conditional branch before the branch is taken. This avoids stalling the pipeline while the branch condition is evaluated, thus keeping the pipeline full and minimizing the effect of conditional branches on processor performance.

***THANK YOU***

# Branch Handling Techniques - Branch Prediction

ANAKHA SADANANDAN  
S7 CSE  
ROLL NO: 11

# Branch Handling Techniques - Branch Prediction

- Branch prediction is one of the ancient performance improving techniques which still finds relevance into modern architectures.
- While the simple prediction techniques provide fast lookup and power efficiency they suffer from high misprediction rate.
- On the other hand, complex branch predictions – either neural based or variants of two level branch prediction – provide better prediction accuracy but consume more power and complexity increases exponentially.
- In addition to this, in complex prediction techniques the time taken to predict the branches is itself very high – ranging from 2 to 5 cycles – which is comparable to the execution time of actual branches.
- Branch prediction is essentially an optimization (minimization) problem where the emphasis is on to achieve lowest possible miss rate, low power consumption and low complexity with minimum resources.

# Different types of branches

- **Forward conditional branches** - based on a run-time condition, the PC (Program Counter) is changed to point to an address forward in the instruction stream.
- **Backward conditional branches** - the PC is changed to point backward in the instruction stream. The branch is based on some condition, such as branching backwards to the beginning of a program loop when a test at the end of the loop states the loop should be executed again.
- **Unconditional branches** - this includes jumps, procedure calls and returns that have no specific condition.

# Static Branch Prediction

Static Branch Prediction predicts always the same direction for the same branch during the whole program execution. It comprises hardware-fixed prediction and compiler-directed prediction. Simple hardware-fixed direction mechanisms can be:

- Predict always not taken
- Predict always taken
- Backward branch predict taken, forward branch predict not taken.

Sometimes a bit in the branch opcode allows the compiler to decide the prediction direction.

# Dynamic Branch Prediction

- Dynamic Branch Prediction: the hardware influences the prediction while execution proceeds.
- Prediction is decided on the computation history of the program.
- During the start-up phase of the program execution, where a static branch prediction might be effective, the history information is gathered and dynamic branch prediction gets effective.
- In general, dynamic branch prediction gives better results than static branch prediction, but at the cost of increased hardware complexity.



# Branch Handling Techniques - Delayed Branches

Ananth Krishna KS  
S7 CSE  
12

# What is branch ?

- A **branch** is an instruction in a computer program that can cause a computer to begin executing a different instruction sequence and thus deviate from its default behavior of executing instructions in order.
- **Branch** may also refer to the act of switching execution to a different instruction sequence as a result of executing a branch instruction. Branch instructions are used to implement control flow in program loops and conditionals.

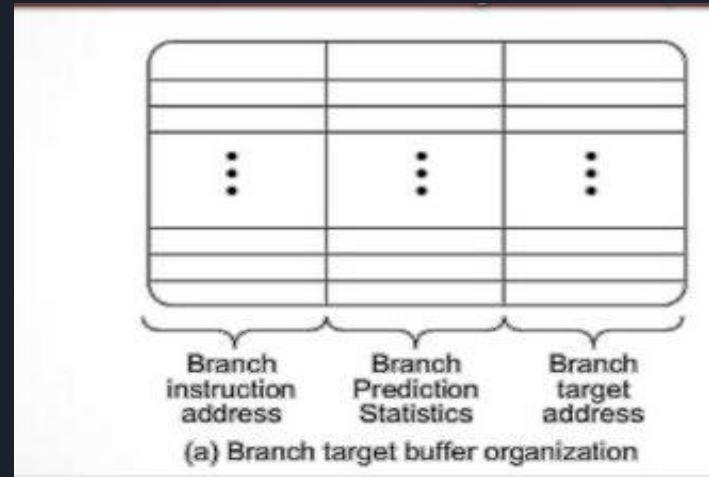


# Branch Handling Techniques

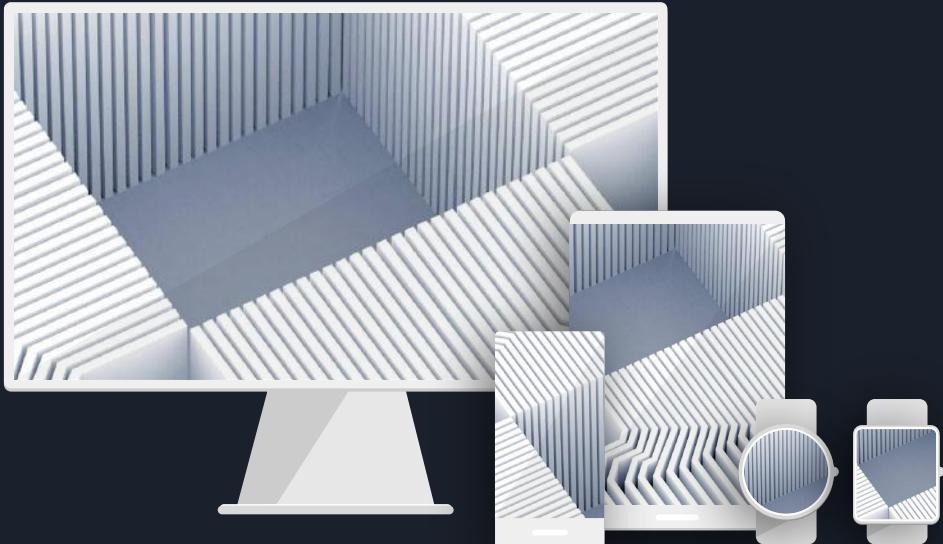
- Branch taken, Branch target, Delayed slot.
- Effect of branching:
  - Parameters:
    - k :Number of stages in pipeline.
    - n :Total no of instructions
    - p :Percentage of branch instructions over n.
    - q :Percentage of successful branch instructions over p.
    - b :Delay slot.
    - t :Pipeline cycle.
  - Branch penalty : $q \text{ of } (p \text{ of } n) * bT$
  - Execution time: $[k + (n - 1) + pqnb] * T$

# Delayed Branches

- A delayed branch of  $d$  cycles allow at most  $d-1$  useful instruction to be executed following the broken branch.
- Execution of these instructions should be independent of branch instruction to achieve a zero branch penalty.



Thank you!



# Fixed Point Operations

o To convert a number from a fixed point type with scaling factor  $R$  to another type with scaling factor  $S$ , the underlying integer must be multiplied by  $R$  and divided by  $S$ ; that is, multiplied by the ratio  $R/S$ . Thus, for example, to convert the value  $1.23 = 123/100$  from a type with scaling factor  $R=1/100$  to one with scaling factor  $S=1/1000$ , the underlying integer 123 must be multiplied by  $(1/100)/(1/1000) = 10$ , yielding the representation  $1230/1000$ .

- To add or subtract two values of the same fixed-point type, it is sufficient to add or subtract the underlying integers, and keep their common scaling factor. The result can be exactly represented in the same type, as long as no overflow occurs (i.e. provided that the sum of the two integers fits in the underlying integer type). If the numbers have different fixed-point types, with different scaling factors, then one of them must be converted to the other before the sum.

- o multiply two fixed-point numbers, it suffices to multiply the two underlying integers, and assume that the scaling factor of the result is the product of their scaling factors. This operation involves no rounding. For example, multiplying the numbers 123 scaled by 1/1000 (0.123) and 25 scaled by 1/10 (2.5) yields the integer  $123 \times 25 = 3075$  scaled by  $(1/1000) \times (1/10) = 1/10000$ , that is  $3075/10000 = 0.3075$ . If the two operands belong to the same fixed-point type, and the result is also to be represented in that type, then the product of the two integers must be explicitly multiplied by the common scaling factor; in this case the result may have to be rounded, and overflow may occur. For example, if the common scaling factor is 1/100, multiplying 1.23 by 0.25 entails multiplying 123 by 25 to yield 3075 with an intermediate scaling factor of 1/10000. This then must be multiplied by 1/100 to yield either 31 (0.31) or 30 (0.30), depending on the rounding method used, to result in a final scale factor of 1/100.

- divide two fixed-point numbers, one takes the integer quotient of their underlying integers, and assumes that the scaling factor is the quotient of their scaling factors. The first division involves rounding in general. For example, division of 3456 scaled by 1/100 (34.56) and 1234 scaled by 1/1000 (1.234) yields the integer  $3456 \div 1234 = 3$  (rounded) with scale factor  $(1/100)/(1/1000) = 10$ , that is, 30. One can obtain a more accurate result by first converting the dividend to a more precise type: in the same example, converting 3456 scaled by 1/100 (34.56) to 3,456,000 scaled by 1/100000, before dividing by 1234 scaled by 1/1000 (1.234), would yield  $3456000 \div 1234 = 2801$  (rounded) with scaling factor  $(1/100000)/(1/1000) = 1/100$ , that is 28.01 (instead of 30). If both operands and the desired result are represented in the same fixed-point type, then the quotient of the two integers must be explicitly divided by the common scaling factor.

# **Floating Point Operation and Floating Point Number**

# Floating Point Number

- A floating point number  $X$  is represented by a pair  $(m, e)$ 
  - $m$  - mantissa
  - $e$  - exponent with an implied base
- The algebraic value is represented as  $X = m \times (r^e)$ 
  - The sign of  $X$  can be embedded in mantissa.

# Floating-point operations

Four primitive operations are defined below for a pair of floating point number represented by  $X = (m_x, e_x)$  and  $Y = (m_y, e_y)$ . For clarity, we assume  $e_x \leq e_y$  and base r=2.

$$X + Y = (m_x \times 2^{e_x - e_y} + m_y) \times x^{e_y}$$

$$X - Y = (m_x \times 2^{e_x - e_y} - m_y) \times x^{e_y}$$

$$X \times Y = (m_x \times m_y) \times 2^{e_x + e_y}$$

$$X \div Y = (m_x \div m_y) \times 2^{e_x - e_y}$$

- The above equations clearly identifies the number of arithmetic operations involved in each floating point function.
- These operations can be divided into 2.
  - For exponent operations such as comparing their relative magnitude or adding or subtracting them.
  - For mantissa operations, including 4 type of fixed operations.
- Floating-point units are ideal for pipelined implementation.

- Two halves of operation demand almost twice as much hardware as that required in fixed point unit.
- Arithmetic shifting operations are needed for equalizing the 2 exponent before their mantissa can be added or subtracted.
- In addition, normalization of a floating point number also require left shift to be performed.

---

# **Static Arithmetic Pipelines - Arithmetic Pipeline Stages**

Anu S Alunkal  
15-CSE

---

# Introduction

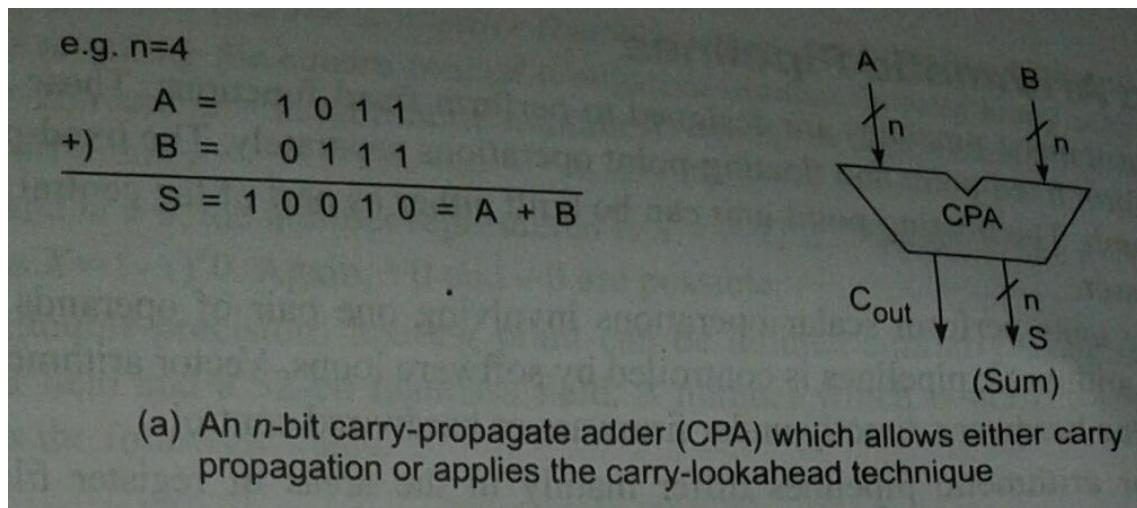
- Arithmetic pipelines are designed to perform fixed functions.
- These arithmetic/logic units performs fixed and floating points operations separately.
- These arithmetic unit performs scalar operations involving one pair of operands at a time.
- The pipelining in scalar arithmetic pipelines are controlled by software loops.
- The pipelining in vector arithmetic pipelines are hardware controlled.
- Both scalar and vector pipelines differ in register files and control mechanism.

---

## Arithmetic pipeline stages

- Depending on the functions to be implemented, different pipeline stages in an arithmetic unit require different hardware logic.
- Since all arithmetic operations can be implemented with the basic add and shifting operations, the core arithmetic stages require some form of hardware to add and to shift.
- For eg: a typical 3 stage floating point adder includes
  - 1.Exponent comparison and equalization
  - 2.Fraction Addition
  - 3.Fraction normalization and exponent readjustment

- (1) can be easily implemented with shift register.
- High speed addition requirement into use carry propagation adder which adds 2 numbers and produces an arithmetic sum (in figure.)
- (3) is implemented using another shift register and addition logic.



Thank  
you!



# Static Arithmetic Pipelines

ARAVIND M MENON  
R.NO 16  
CSE S7



# INTRODUCTION

Pipelining is one way of improving the overall processing performance of a processor. This architectural approach allows the simultaneous execution of several instructions. Pipelining is transparent to the programmer; it exploits parallelism at the instruction level by overlapping the execution process of instructions. It is analogous to an assembly line where workers perform a specific task and pass the partially completed product to the next worker.



# ARITHMATIC PIPELINING

Some functions of the arithmetic logic unit of a processor can be pipelined to maximize performance. An arithmetic pipeline is used for implementing complex arithmetic functions like floating-point addition, multiplication, and division. These functions can be decomposed into consecutive subfunctions. For example Figure 3.13 presents a pipeline architecture for floating-point addition of two numbers. (A nonpipelined architecture of such an adder is described in Chapter 2.) The floating-point addition can be divided into three stages: mantissas alignment, mantissas addition, and result normalization [MAN 82,HAY 78].

# FIXED POINT MULTIPLICATION PIPELINE

- A pipelined multiplier based on the digit products can be designed using digit product generation logic and the digit adders.

Example:

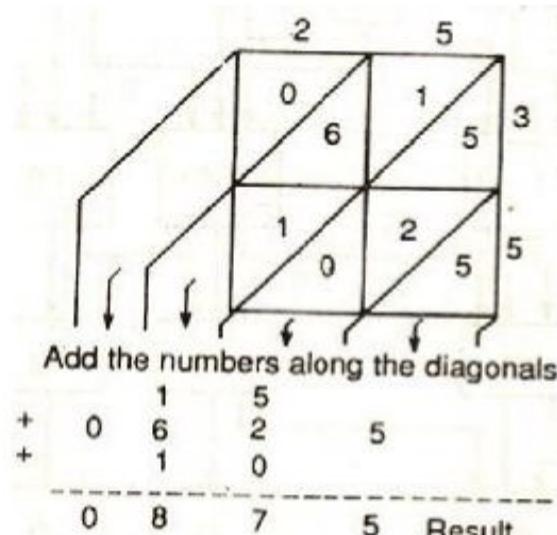
$$25 * 35 = 875$$

Now for binary multiplication:

$$A = \begin{matrix} a_1 & a_0 \end{matrix}$$

$$B = \begin{matrix} b_1 & b_0 \end{matrix}$$

$$\begin{array}{r} & a_1 & a_0 \\ & b_1 & b_0 \\ \hline . & a_1b_0 & a_0b_0 \\ a_1b_1 & a_0b_1 \\ \hline a_1b_1 & a_1b_0 + a_0b_1 & a_0b_0 \end{array}$$





# STATIC ARITHMETIC PIPELINES

CONVERGENCE DIVISION

Archana Venugopal  
CSE S7 17

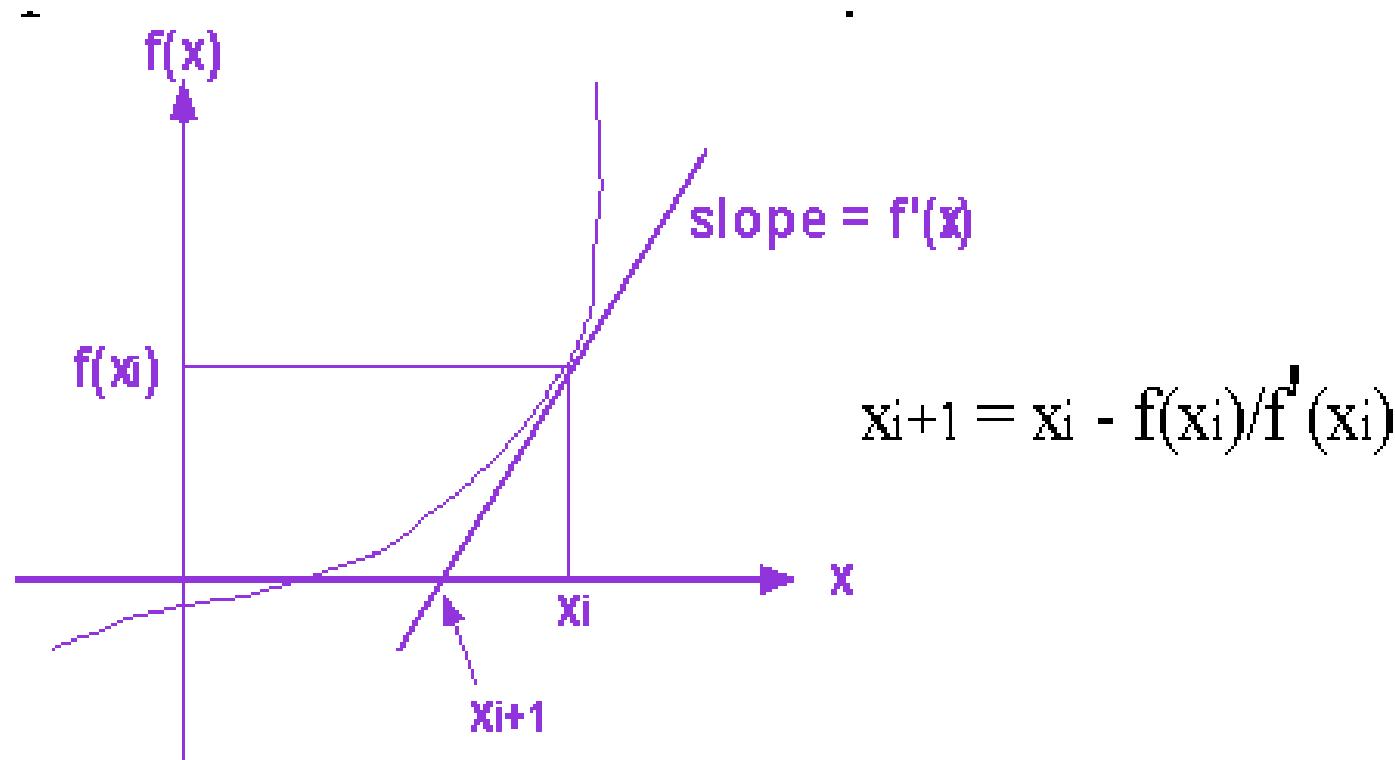
# STATIC PIPELINES

- Pipelines are usually divided into two classes: instruction pipelines and arithmetic pipelines.
- There are two types of pipelines: static and dynamic.
- A static pipeline can perform only one function (such as addition or multiplication) at a time.
- Static pipelines are linear - A linear pipeline processor is a series of processing stages and memory access.
- The operation of a static pipeline can only be changed after the pipeline has been drained. (A pipeline is said to be drained when the last input data leave the pipeline.)

- For example,

Consider a static pipeline that is able to perform addition and multiplication. Each time that the pipeline switches from a multiplication operation to an addition operation, it must be drained and set for the new operation. The performance of static pipelines is severely degraded when the operations change often, since this requires the pipeline to be drained and refilled each time.

# CONVERGENCE DIVISION



# CONVERGENCE DIVISION

- Generate the reciprocal of the divisor by an iterative process and then use:

$$A/B = A^* (1/B)$$

- Use (great, great, great grand-uncle) Newton Raphson method to solve for  $(1/B)$ .

$$X_{i+1} = X_i - f(X_i)/f'(X_i)$$

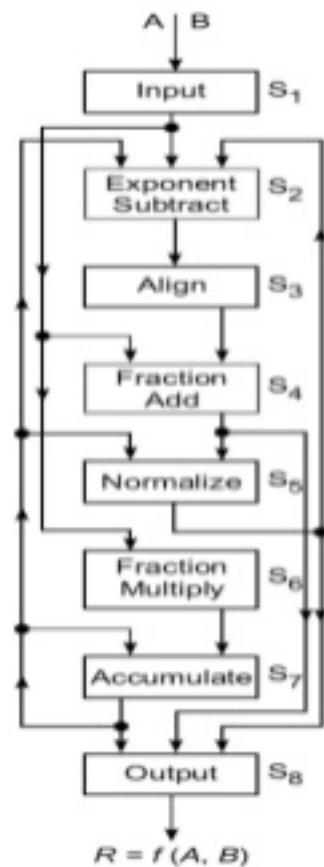
# MULTIFUNCTIONAL ARITHMETIC PIPELINES

Basil Reji  
CSE S7 18

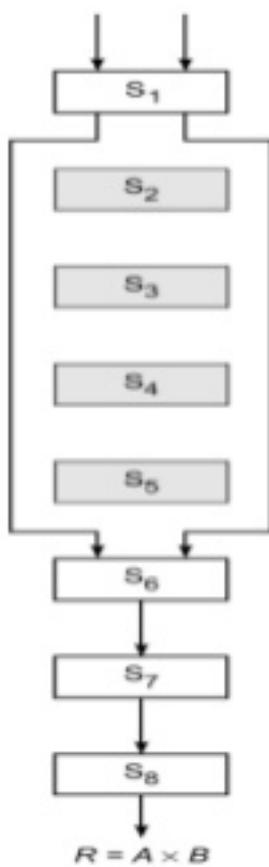
# DEFINITION

- When different functions at different times are performed through the pipeline, this is called Multifunctional Pipelines.
- Multifunctional pipelines reconfigurable at different times according to the operations being performed.
- It is interconnecting different subsets of stages in the pipeline.

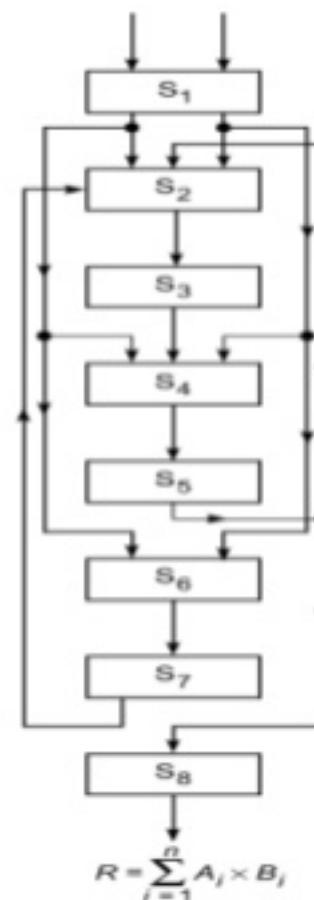
- It has
  - One instruction processing unit.
  - Four memory buffer units.
  - Four arithmetic units.
- It provides four parallel execution pipelines below the IPU.
- Any mixture of scalar and vector instructions can be executed simultaneously in four pipes.



(a) Pipeline stages and interconnections



(b) Fixed-point multiplication



(c) Floating-point dot product

**Fig. 6.27** The multiplication arithmetic pipeline of the TI Advanced Scientific Computer and the interstage connections of two representative functions (Shaded stages are unutilized)

# TYPES

- Static
  - Initially configured for one functional evaluation.
  - For another function,it need to be drained and reconfigured.
  - You cannot have two inputs of different function at the same time.

- Dynamic
  - Can do different functional evaluation at a time.
  - It is difficult to control as we need to be sure that there is no conflict in usage of stages.

# Superscalar Pipeline Design - Pipeline Design Parameters

Basil K Y

November 21, 2018

# Pipeline design parameters

- Instruction issue rate : The number of instructions issued per cycle.
- Instruction issue latency : Time required between issuing of two adjacent instructions.
- Simple operation latency : Latency of simple operations like integer adds, loads, stores, moves etc.
- Instruction level parallelism(ILP) is the maximum number of instructions that can be simultaneously executed in the pipeline.
- Pipeline cycle for the scalar base processor is assumed to be 1 time unit, called the base cycle.

# Design parameters for pipeline design

Machine Type	Scalar base machine of k pipeline stages	Superscalar machine of degree m
Machine pipeline cycle	1 (base cycle)	1
Instruction issue rate	1	m
Instruction issue latency	1	1
Simple operation latency	1	1
ILP to fully utilize the pipeline	1	m

# SuperScalar Pipeline Scheduling

Elizabeth Saba  
S7,CSE(23)

- Instruction issue and completion policies are critical to superscalar processor performance.
- Based on the way a dispatch unit of superscalar processor issue the instructions, superscalar architectures can be classified into
  - In-Order issue processor
  - Out-Of -Order issue processor

# In-Order Issue processor

- Issue instructions in the order in which they appear in the program .
- Consider the below figure in which superscalar processor of degree 2 is shown

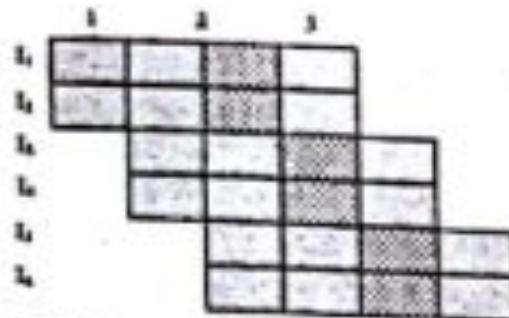
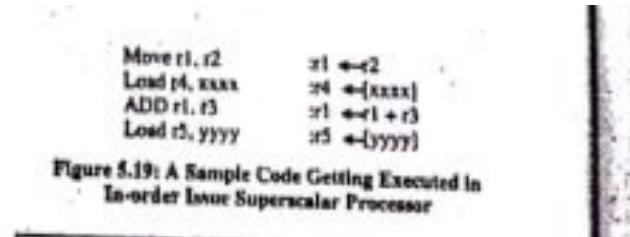


Figure 5.18: Superscalar Processor of Degree 2 with 4-Stage Pipeline

- The pattern in which piece of code get executed is



- The first two instructions are fetched simultaneously and decoded and scheduled for execution in parallel.
- Since add instruction has a raw dependency with the move instruction, it is stalled for one cycle.

# Out of order issue processor

- In out-of-order issue processor it would have not stalled the last load instruction.
- It provides much better performance than in order issue processor.
- It requires much more complex hardware to implement.

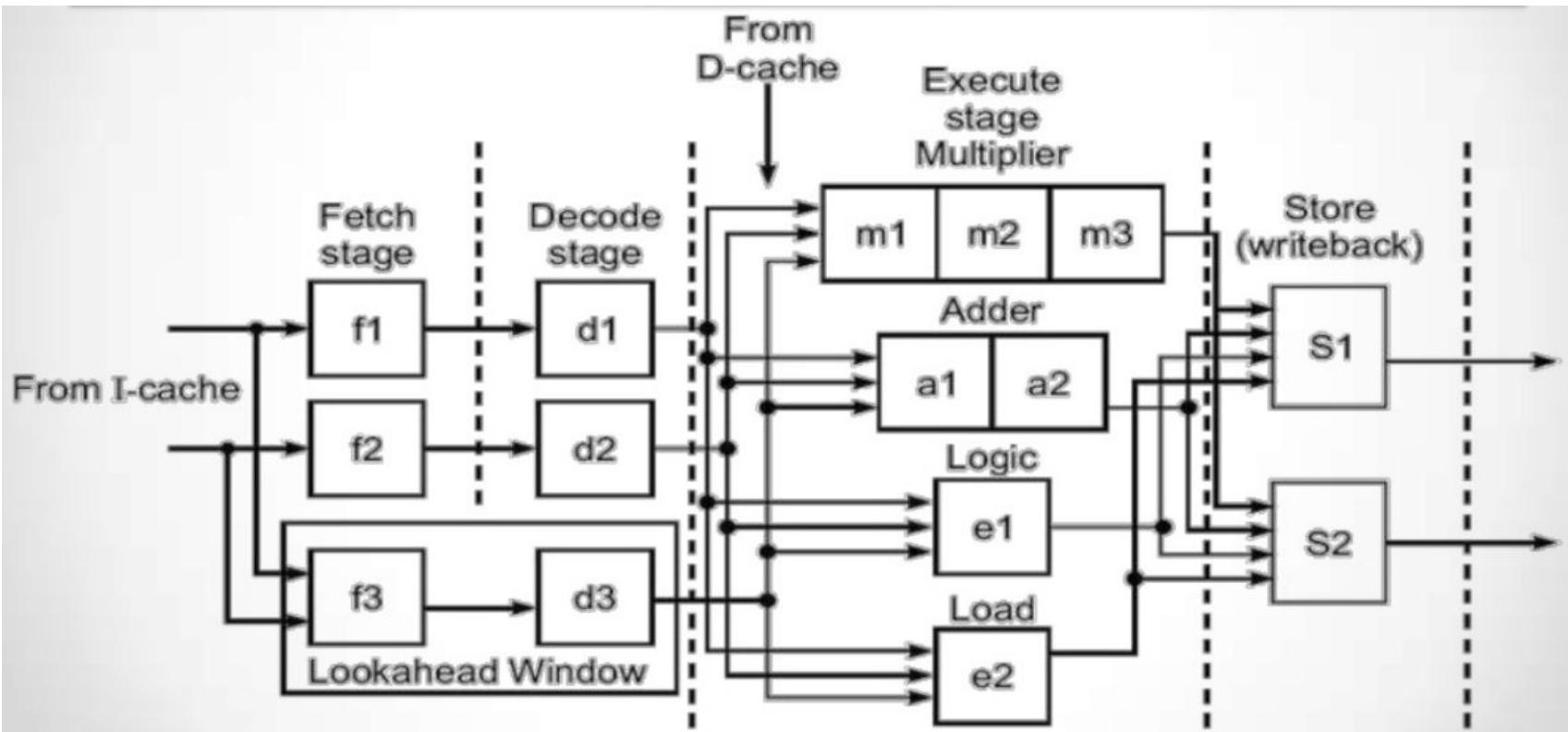
# Superscalar Pipeline Structure

DANY JOY  
21  
S7 CSE

# Superscalar Pipeline Structure

- In an m-issue superscalar processor, the instruction decoding and execution resources are increased to form effectively m pipelines operating concurrently. At some pipeline stages, the functional units may be shared by multiple pipelines. This resource shared multiple pipeline structure is illustrated in next page.
- The process can issue two instructions per cycle if there is no resource conflict and no data dependence problem. There are essentially two pipelines in the design. Both having four processing stages labelled fetch, decode, execute and store respectively.

# A dual-pipeline Superscalar Processor



- Each pipeline essentially has its own fetch unit, and store unit. The two instruction streams flowing the two pipelines are retrieved from a single source stream (the I cache).
- The fan out from a single instruction stream is subject to resource constraints and a data dependence relationship among the successive instructions.
- For simplicity, we assume that each pipeline stage requires one cycle, except the execute stage which may require a variable number of cycles.
- Four functional units, multiplier, adder, logic unit, and load unit , are available for use in the execute stage.

- The multiplier itself has 3 pipeline stages, the adder has 2 stages, and the others each have only one stage.
- The 2 store units ( $s_1, s_2$ ) can be dynamically used by the 2 pipelines, depending on availability at a particular cycle.
- There is a lookahead window with its own fetch and decoding logic. This window is used for instruction lookahead in case out of order instruction issue is desired to achieve better pipeline throughput.
- It requires complex logic to schedule multiple pipelines simultaneously, especially when the instructions are retrieved from same source. The aim is to avoid pipeline stalling and minimize pipeline idle time.

# Superscalar Pipeline Stalling

BY,

ELDHO SHAJU

S7 CSE 22

# ABOUT

- A superscalar processor is a CPU that implements a form of parallelism called instruction-level parallelism within a single processor. While a superscalar CPU is typically also pipelined, superscalar and pipelining execution are considered different performance enhancement techniques.

# Superscalar Processor

- Superscalar processors are designed to exploit more instruction-level parallelism in user programs.
- Only independent instructions can be executed in parallel without causing a wait state.
- The amount of instruction-level parallelism varies widely depending on the type of code being executed.

# Pipelining in Superscalar Processor

- In order to fully utilise a superscalar processor of degree  $m$ ,  $m$  instructions must be executable in parallel. This situation may not be true in all clock cycles. In that case, some of the pipelines may be stalling in a wait state
- In a superscalar processor, the simple operation latency should require only one cycle, as in the base scalar processor.

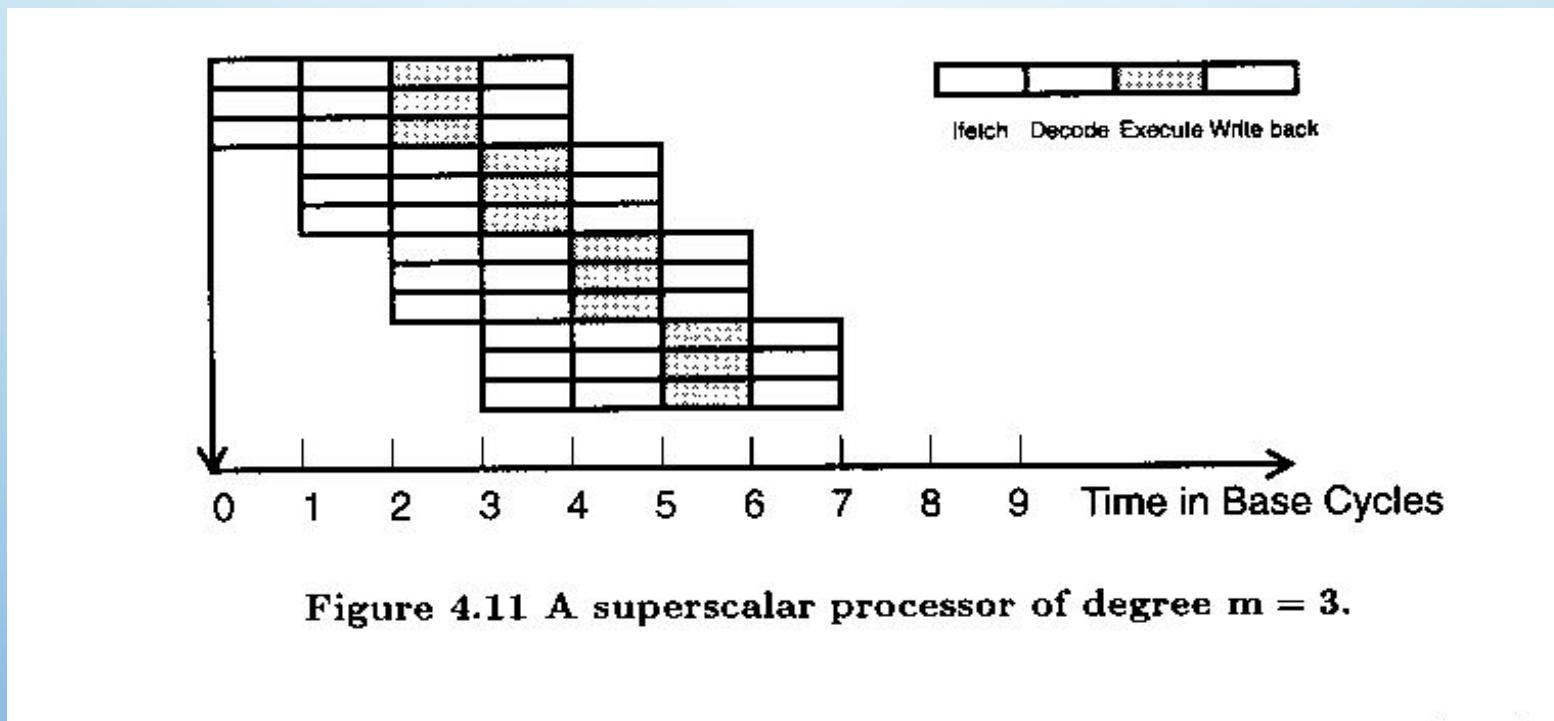


Figure 4.11 A superscalar processor of degree  $m = 3$ .

**THANK  
YOU!**

# SuperScalar Pipeline Scheduling

Elizabeth Saba  
S7,CSE(23)

- Instruction issue and completion policies are critical to superscalar processor performance.
- Based on the way a dispatch unit of superscalar processor issue the instructions, superscalar architectures can be classified into
  - In-Order issue processor
  - Out-Of -Order issue processor

# In-Order Issue processor

- Issue instructions in the order in which they appear in the program .
- Consider the below figure in which superscalar processor of degree 2 is shown

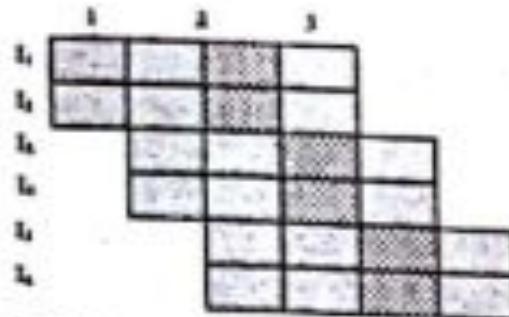
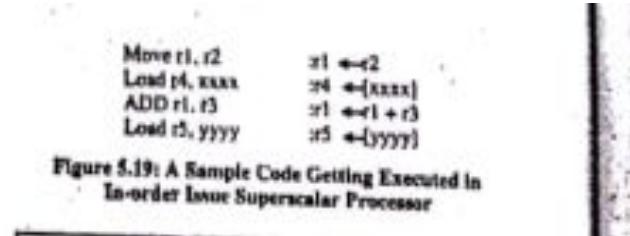


Figure 5.18: Superscalar Processor of Degree 2 with 4-Stage Pipeline

- The pattern in which piece of code get executed is



- The first two instructions are fetched simultaneously and decoded and scheduled for execution in parallel.
- Since add instruction has a raw dependency with the move instruction, it is stalled for one cycle.

# Out of order issue processor

- In out-of-order issue processor it would have not stalled the last load instruction.
- It provides much better performance than in order issue processor.
- It requires much more complex hardware to implement.

# Superscalar Pipeline Design

## In-order and Out-of-order Issue

Submitted by,

Elizabeth Eldho  
S7 CSE, 24

# 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 than would otherwise be possible at a given clock rate. Each execution unit is not a separate processor , but an execution resource within a single CPU such as an arithmetic logic unit.

# In-order instruction execution

- instructions are fetched, executed & committed in compiler-generated order .If one instruction stalls, all instructions behind it stall
- instructions are statically scheduled by the hardware
- scheduled in compiler-generated order
- how many of the next n instructions can be issued, where n is the superscalar issue width
- superscalars can have structural & data hazards within the n instructions
- advantage of in-order instruction scheduling: simple implementation
  - 1.faster clock cycle
  - 2.fewer transistors
  - 3.faster design/development/debug time

# Out-order instruction execution

- instructions are fetched in compiler-generated order
- instruction completion may be in-order (today) or out-of-order (older computers)
- in between they may be executed in some other order
- instructions are dynamically scheduled by the hardware
- superscalars can have structural & data hazards within the n instructions
- advantages: higher performance
  - better at hiding latencies, less processor stalling
  - higher utilization of functional units

# In-order instruction issue: Alpha 21164

2 styles of static instruction scheduling

- dispatch buffer & instruction slotting (Alpha 21164)
- shift register model (UltraSPARC-1)

Instruction slotting:

- can issue up to 4 instructions
  - completely empty the instruction buffer before filling it again
  - compiler can pad with nop s so a conflicting instruction is issued with the following instructions, not alone

# 21164 Instruction Unit Pipeline

## Fetch & Issue:

- S0: instruction fetch, branch prediction bits read
- S1: opcode decode, target address calculation, if predict taken, redirect the fetch
- S2: instruction slotting: decide which of the next 4 instructions can be, issued intra-cycle structural hazard check, intra-cycle data hazard check
- S3: instruction dispatch inter-cycle load-use hazard check, register read

## UltraSparc 1:Shift register model

- can issue up to 4 instructions per cycle
- shift in new instructions after every group of instructions is issued

# Superscalars

Hardware impact:

- more & pipelined functional units
- multi-ported registers for multiple register access
- more buses from the register file to the additional functional units
- multiple decoders
- more hazard detection logic
- more bypass logic
- wider instruction fetch
- multi-banked L1 data cache

Else the processor has structural hazards (due to an unbalanced design) and stalling. There are restrictions on instruction types that can be issued together to reduce the amount of hardware. Static (compiler) scheduling helps.

# Superscalar Performance

BY:

ELIZEBETH SHIJU



## Definition

- Superscalar processing is the ability to initiate multiple instructions during the same clock cycle.
- A typical Superscalar processor fetches and decodes the incoming instruction stream several instructions at a time.
- Superscalar architecture exploit the potential of ILP(Instruction Level Parallelism).



To compare the relative performance of a superscalar processor with that of a scalar base machine

- Estimate the ideal execution of N independent instructions through pipeline.
- The time required by the scalar base machine is,

$$T(1,1) = k + N - 1 \text{ (base cycles)}$$

- The ideal execution time required by an m-issue superscalar machine is

$$T(m,1)=k + (N-m)/m \text{ (base cycles)}$$

Where k is the time required to execute the first m instructions through m pipelines

The second term is the time required to execute the remaining N-m instructions,m per cycle,through m pipelines.



- The ideal speedup of the superscalar machine over the base machine is

$$S(m,1) = T(1,1)/T(m,1)$$

$$= (N+k-1)/(N/m+k-1)$$

$$= (m(N+k-1))/(N+m(k-1))$$

As  $N \rightarrow \infty$ , the speedup limit  $S(m,1) \rightarrow m$ , as expected.

# Latency-hiding techniques

Febi Justin-Roll No:26

# Introduction

- In distributed shared memory machines, access to remote memory is likely to be slow compared to the ever-increasing speeds of processors.
- Thus, any scalable architecture must rely on techniques to reduce/hide/tolerate remote-memory-access latencies.
- There are four methods,
  1. Use of prefetching techniques
  2. Use of coherent cacheing techniques
  3. Relaxing the memory consistency requirements
  4. Using multiple-contexts to hide latency

# Prefetching Techniques

- Prefetching is either software-controlled or hardware-controlled.
- In software-controlled prefetching explicit "prefetch" instructions are issued for data that is "known" to be remote.
- Hardware-controlled prefetching is done through the use of long cache lines to capitalize on spatial locality or through the use of instruction lookahead.
- Long cache lines introduce the problem of "false sharing", whereas instruction lookahead is limited by branches and/or branch-prediction accuracy.

# Coherent Caches

- There are examples of maintaining coherent caches using snoopy caches for bus-based systems and using directory-based caches for general distributed memory machines.
- A major problem with the design of coherent caches is scalability.

# Relaxed Memory Consistency Models

- Under the sequential consistency model, there is a partial ordering of all reads and writes by all processors, which obeys the program ordering.
- Thus, the result of any execution appears as some interleaving of the operations of the individual processors (as if executed on some multithreaded sequential machine).
- Most of the references to sequential consistency often attribute stronger conditions in that they require a total ordering of all reads and writes (a property that is better named "dynamic atomicity").

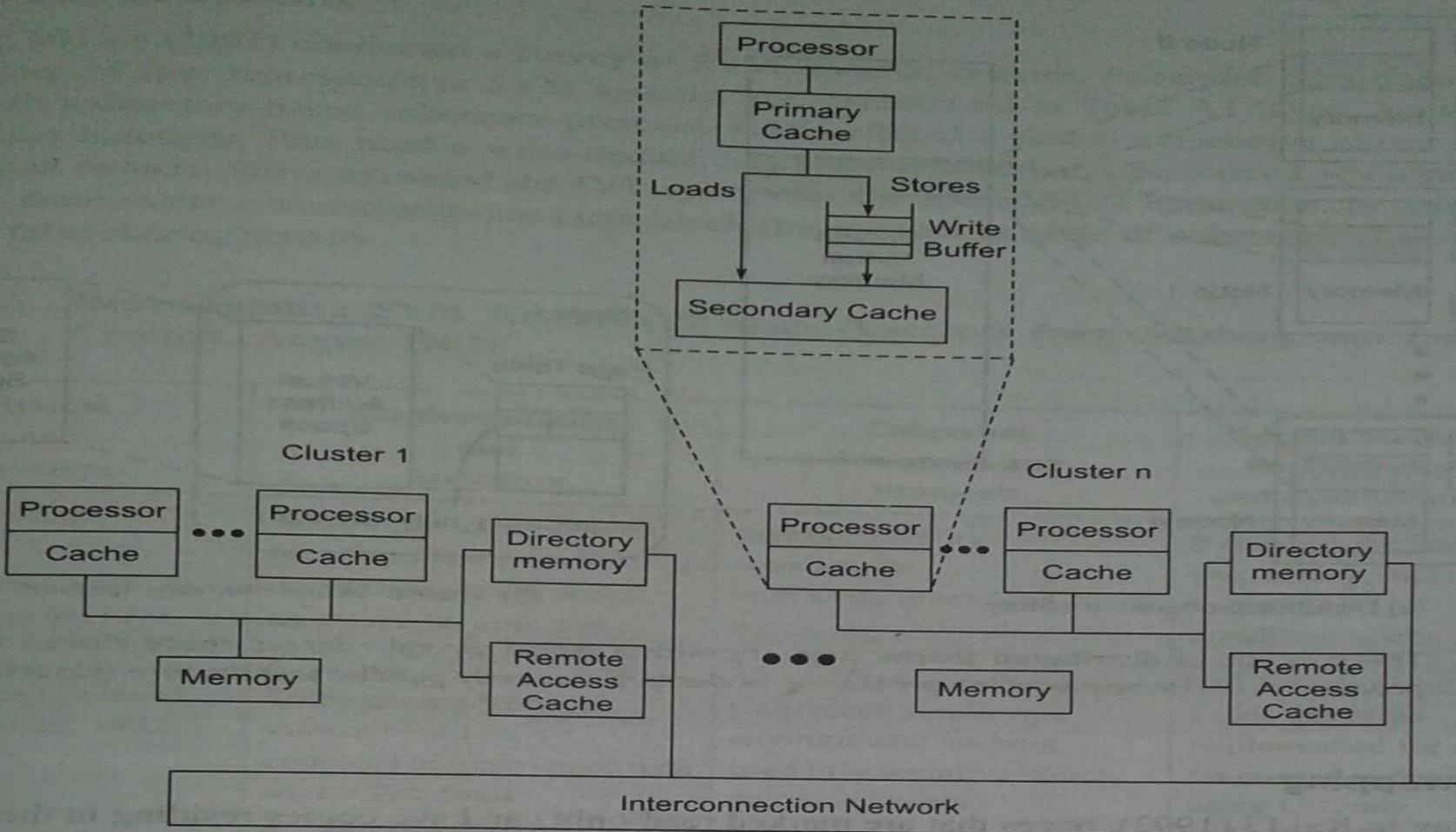
- Under the Weak consistency model, synchronization operators are not allowed to "perform" until all preceding load/store operations have completed.
- Similarly, no load/store operations are allowed to "perform" until all preceding synchronization operations have completed.
- In addition, (only) synchronization operators are sequentially consistent.

- Under processor consistency, writes issued by the same processor are never seen out of order (by any other processor in the system), but writes by different processors may be observed in different orders by different processors.
- Read operations following a write operation may bypass it (i.e. be observed by other processors as happening before the write).

# **Shared Virtual Memory : The Architecture Environment**

# The Architecture Environment

- Single address space multi-processors must use shared virtual memory.
- **Stanford Dash Experiment** provides the model for such an architectural environment



**Fig. 9.1** A scalable coherent cache multiprocessor with distributed shared memory modeled after the Stanford Dash (Courtesy of Anoop Gupta et al, Proc. 1991 Ann. Int. Symp. Computer Arch.)

- The Dash architecture (in previous slide) has a large scale, cache-coherent, NUMA multiprocessor system.
- It consists of multiprocessor clusters connected through a scalable, low latency interconnection network.
- Physical m/y is distributed among the processing nodes in various clusters.
- Distributed m/y forms a global address space.
- Cache-coherence is maintained using an invalidating, distributed directory based protocol.
- For each m/y block, directory keeps tracks of remote node caching it.
- When write occurs : Point-to-point message is sent to invalidate remote copies.
- Ack messages were used to inform the originating nodes about completion of invalidation.

- Two levels of local cache were used.
- Loads and writes were separated using write buffers.
- Main m/y share all processing nodes in same cluster.
- For prefetching,directory based coherence protocol, directory m/y and remote access are used for each cluster.
- Remote access cache are shared by all processors in same cluster.

---

# Shared Virtual Memory

---

George Scaria  
S7, CSE

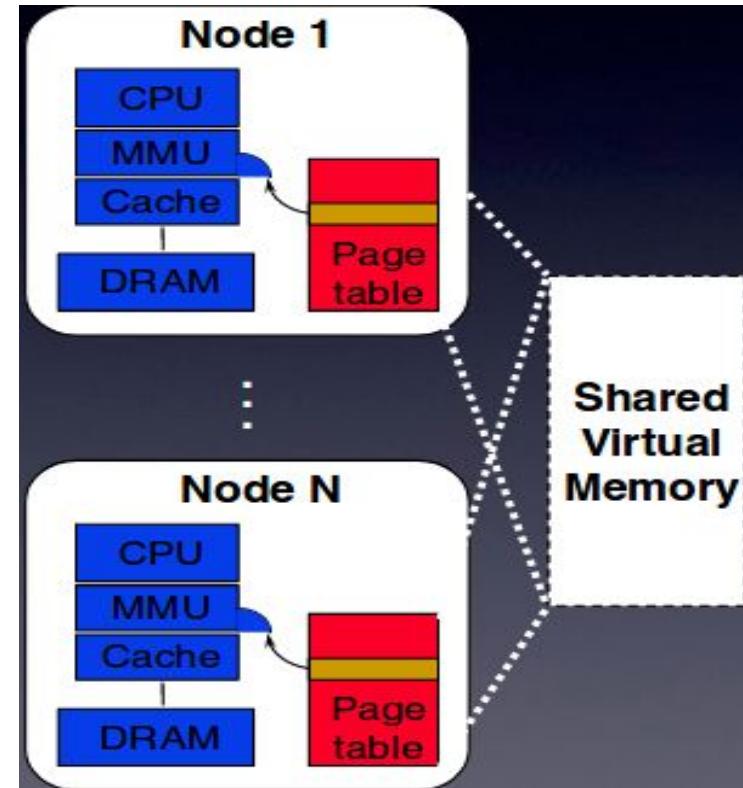
# Shared Virtual memory

- Pool of “shared pages”: if not local, page is not mapped
- Page table entry access bits

Virtual page #	Physical page #	Valid	Access
----------------	-----------------	-------	--------

- H/w detects read access to invalid page
  - read faults
- H/w detects writes to mapped memory with no write access
  - write faults

# Shared Virtual memory



# Shared Virtual memory

- Virtual memory makes it easy for processes to share memory as all memory accesses are decoded using page tables.
- For processes to share the same virtual memory, the same physical pages are referenced by many processes. The page tables for each process contain the Page Table Entries that have the same physical PFN.

# Shared Virtual memory

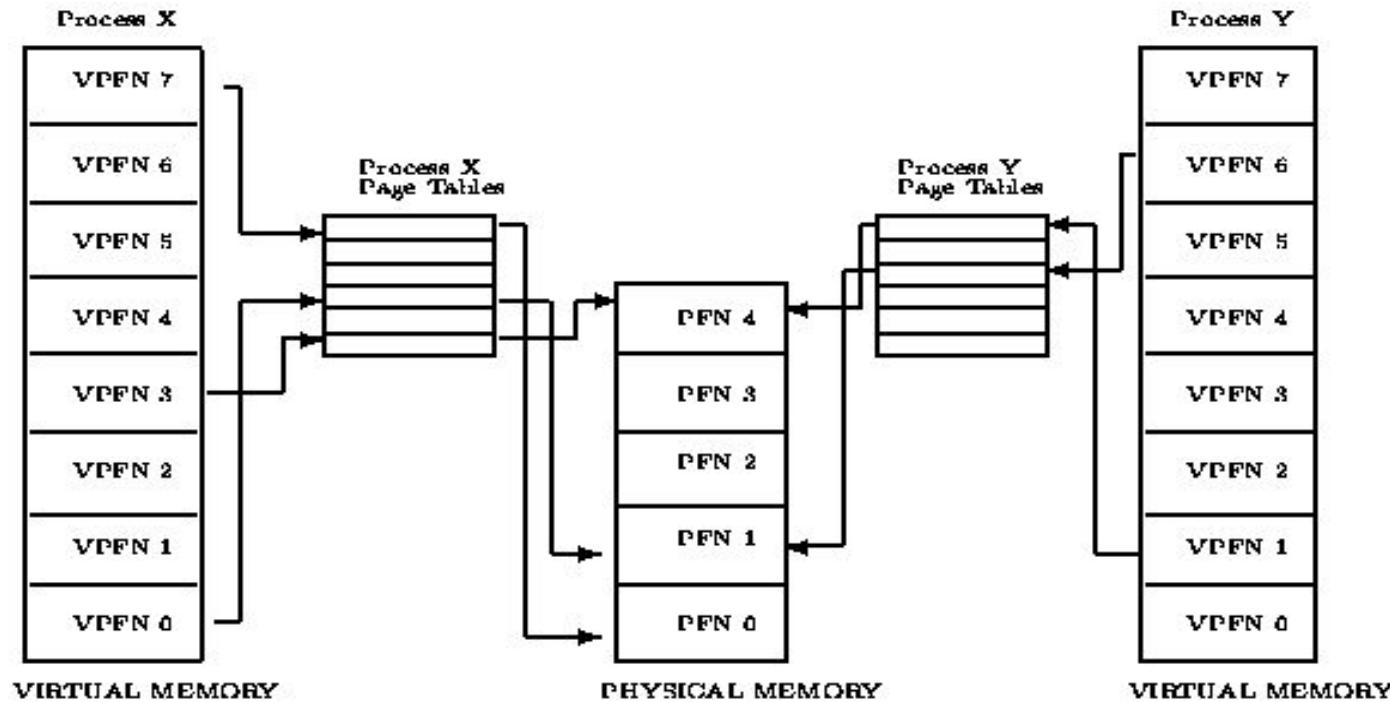
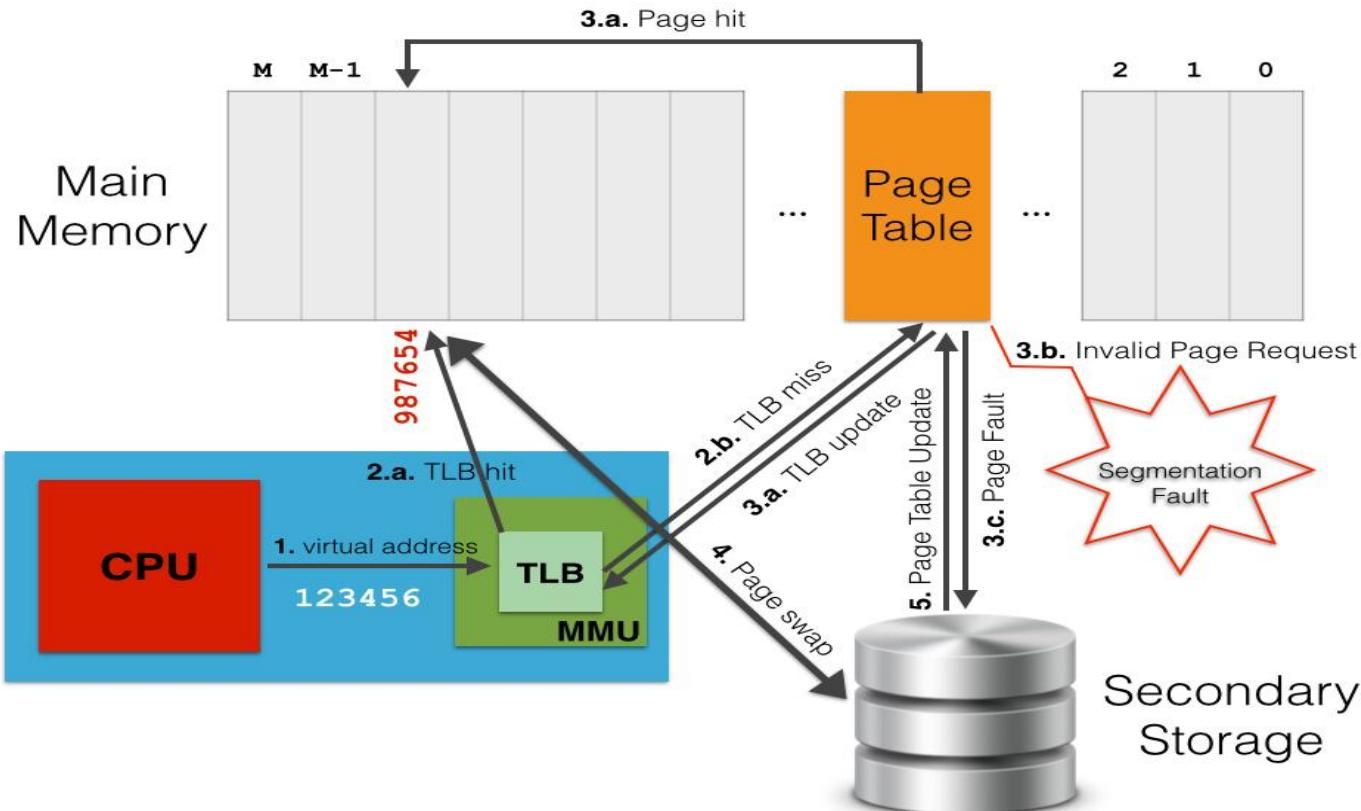


Fig : Sharing physical memory between processes

# Shared Virtual Memory - Page Swapping

Submitted by  
Gopika Chandrakumar  
29 s7,CSE

- ❑ If a process needs to bring a virtual page into physical memory and there are no free physical pages available, the operating system must make room for this page by discarding another page from physical memory.
- ❑ If the page to be discarded from physical memory came from an image or data file and has not been written to then the page does not need to be saved. Instead it can be discarded and if the process needs that page again it can be brought back into memory from the image or data file.



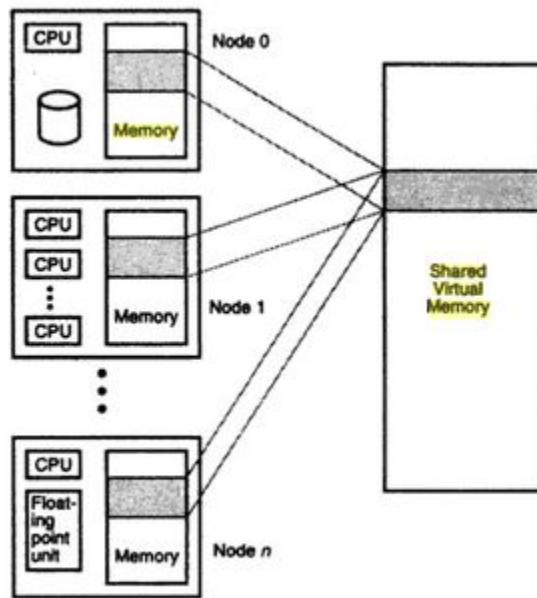
- ❑ However, if the page has been modified, the operating system must preserve the contents of that page so that it can be accessed at a later time. This type of page is known as a *dirty* page and when it is removed from memory it is saved in a special sort of file called the swap file. Accesses to the swap file are very long relative to the speed of the processor and physical memory and the operating system must juggle the need to write pages to disk with the need to retain them in memory to be used again.
- ❑ If the algorithm used to decide which pages to discard or swap (the *swap algorithm*) is not efficient then a condition known as *thrashing* occurs. In this case, pages are constantly being written to disk and then being read back and the operating system is too busy to allow much real work to be performed. If, for example, physical page frame number 1 in Figure [3.1](#) is being regularly accessed then it is not a good candidate for swapping to hard disk. The set of pages that a process is currently using is called the *working set*. An efficient swap scheme would make sure that all processes have their working set in physical memory.

- ❑ Linux uses a Least Recently Used (LRU) page aging technique to fairly choose pages which might be removed from the system. This scheme involves every page in the system having an age which changes as the page is accessed. The more that a page is accessed, the younger it is; the less that it is accessed the older and more stale it becomes. Old pages are good candidates for swapping.
- ❑

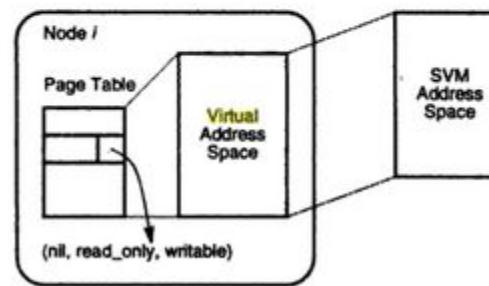
# Shared Virtual Memory

SUBMITTED BY,  
GOPIKA G NAIR  
S7 CSE,NO:30

- Shared Virtual Memory was first developed in a Ph.D thesis by Li at Yale University.
- The idea is to implement coherent shared memory on a network of processors without physically shared memory.
- The system uses virtual addresses instead of physical address for memory references.
- Each virtual address space can be large as a single node can provide and is shared by all the nodes in the system.



(a) Distributed shared memory



(b) Shared virtual memory mapping

- The SVM address space is organized in pages which can be accessed by any node in the system.
- A memory-mapping manager on each node views its local memory as a large cache of pages for its associated processor.
- SVM system uses page replacement policies to find an available page frame, swapping its content to the sending node.
- The memory coherence problem is solved in SVM through fault handlers and their servers.
- To client programs, this mechanism is completely transparent.

# EXAMPLES OF SVM SYSTEMS

System and Developer	Implementation and Structure	Coherence Semantics and Protocols	Special mechanics for Performance and Synchronization
Stanford Dash (Lenoski, Laudon, Gharachorloo, Gupta, and Hennessy, 1988–).	Mesh-connected network of Silicon Graphics 4D/340 workstations with added hardware for coherent caches and prefetching.	Release memory consistency with write-invalidate protocol.	Relaxed coherence, prefetching, and use queued locks for synchronization.
Yale Linda (Carriero and Gelernter, 1982–).	Software-implemented system based on the concepts of tuple space with access functions to achieve coherence via virtual memory management.	Coherence varies with environment; hashing is used in associative search; no mutable data.	Linda can be implemented for many languages and machines using C-Linda or Fortran-Linda interfaces.
CMU Plus (Bisiani and Ravishankar, 1988–).	A hardware implementation using MC 88000, Caltech mesh, and Plus kernel.	Uses processor consistency, nondemand write-update coherence, delayed operations.	Pages for sharing, words for coherence, complex synchronization instructions.

# PREFETCHING TECHNIQUES

Submitted by,  
Greeshma M Benny  
S7, CSE,31

# Prefetching

- In **computer architecture**, instruction **prefetch** is a technique used in central processor units to speed up the execution of a program by reducing wait states.
- **Prefetching** occurs when a processor requests an instruction or data block from main memory before it is actually needed.

# Prefetching Approaches

- Software-based
  - Explicit “fetch” instructions Additional instructions executed
  - Additional instructions executed
- Hardware-based
  - Special hardware
  - Unnecessary prefetchings (w/o compile time information)

# Software Data Prefetching

- fetch instruction
  - Non-blocking memory operation
  - Cannot cause exceptions (e.g. page faults)
- Modest hardware complexity
- Challenge -- prefetch scheduling
  - Placement of fetch inst relative to the matching load or store inst
  - Hand-coded by programmer or automated by compiler

# Loop-based Prefetching

- Loops of large array calculations
  - Common in scientific codes
  - Poor cache utilization
  - Predictable array referencing patterns
- fetch instructions can be placed inside loop bodies s.t. current iteration prefetches data for a future iteration

# Limitation of Software-based Prefetching

- Normally restricted to loops with array accesses
- Hard for general applications with irregular access patterns
- Processor execution overhead
- Significant code expansion
- Performed statically

# Sequential Prefetching

- Take advantage of spatial locality
- One block lookahead (OBL) approach
  - Initiate a prefetch for block  $b+1$  when block  $b$  is accessed
  - Prefetch-on-miss
    - Whenever an access for block  $b$  results in a cache miss
  - Tagged prefetch
    - Associates a tag bit with every memory block
    - When a block is demand-fetched or a prefetched block is referenced for the first time.

# Software vs. Hardware Prefetching

- Software
  - Compile-time analysis, schedule fetch instructions within user program
- Hardware
  - Run-time analysis w/o any compiler or user support
- Integration
  - e.g. compiler calculates degree of prefetching ( K ) for a particular reference stream and pass it on to the prefetch hardware.

# Benefits of Prefetching

Irene Abraham  
32  
s7,cse

# PREFETCHING

Prefetching uses knowledge about the expected misses in a program to move the corresponding data close to the processor before it is actually needed. prefetching can be classified based on whether it is binding or nonbinding, and whether it is controlled by hardware or software.

Prefetching is the loading of a resource before it is required to decrease the time waiting for that resource. Examples include instruction prefetching where a CPU caches data and instruction blocks before they are executed, or a web browser requesting copies of commonly accessed web pages. Prefetching functions often make use of a cache.

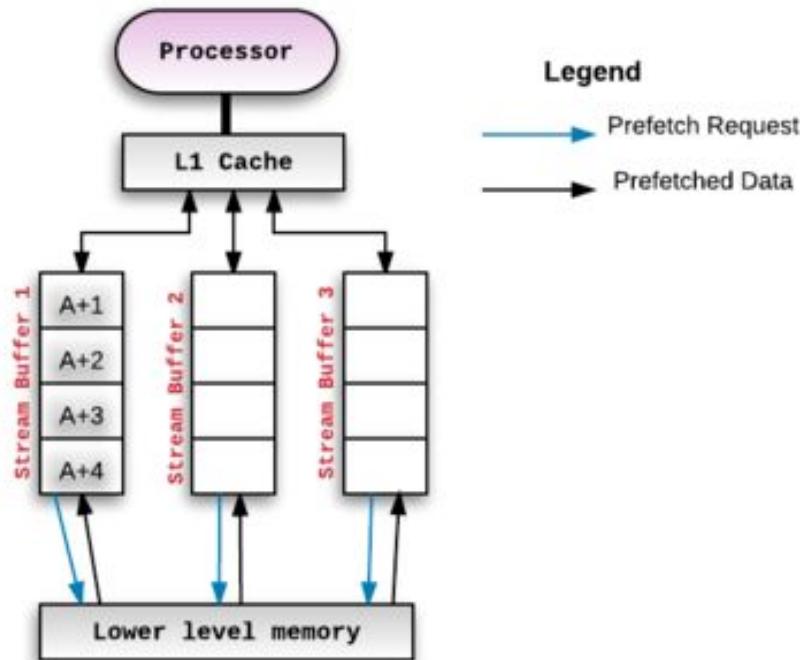
# Benefits of Prefetching

The benefits of prefetching come from different sources.

- Most obvious benefit occur when a prefetch id issued early enough in the code so that the line is already in the cache by the time it is referenced
- Prefetching can even improve performance even when the above one is not possible. Example, when the address of a data structure cannot be determined until immediately before it is referenced.
- If multiple prefetches are issued back to back to fetch the data structure, the latency of all but the first prefetched reference can be hidden due to the pipelining of the memory access.

- Prefetching offers another benefits in multiprocessors that use an ownership based cache coherence protocol
- If a cache block line is to be modified,prefetching it directly with ownership can significantly reduce the write latencies and the ensuing network traffic for obtaining ownership.
- Network traffic is reduced in read-modify-write instructions, since prefetching with ownership avoids first fetching a read shared copy.
- Prefetching allows applications and hardware to maximize performance and minimize wait times by preloading resources that users will need before they request them.

# Prefetching in cache



# Prefetching

Benchmark Results

# Prefetching

---

**Prefetching** in computer science is a technique for speeding up fetch operations by beginning a fetch operation whose result is expected to be needed soon. Usually this is before it is known to be needed, so there is a risk of wasting time by prefetching data that will not be used. The technique can be applied in several circumstances:

- Cache prefetching, a speedup technique used by computer processors where instructions or data are fetched before they are needed
- Prefetch input queue (PIQ), in computer architecture, pre-loading machine code from memory
- Link prefetching, a web mechanism for prefetching links
- The Prefetcher technology in modern releases of Microsoft Windows
- Prefetch buffer, a feature of DDR SDRAM memory
- Swap prefetch, in computer operating systems, anticipatory paging

# Benchmarking

---

- **Benchmarking** is comparing ones business processes and performance metrics to industry bests and best practices from other companies.
- In project management benchmarking can also support the selection, planning and delivery of projects.
- Dimensions typically measured are quality, time and cost.
- In the process of best practice benchmarking, management identifies the best firms in their industry, or in another industry where similar processes exist, and compares the results and processes of those studied to one's own results and processes.
- Benchmarking is used to measure performance using a specific indicator (cost per unit of measure, productivity per unit of measure, cycle time of x per unit of measure or defects per unit of measure) resulting in a metric of performance that is then compared to others.

# Benchmark Results

---

- For each benchmark, the two bars correspond to the cases with no prefetching (**N**) and with selective prefetching (**S**).
- In each bar, the bottom section is the amount of time spent executing instructions (including instruction overhead of prefetching), and the section above that is the memory stall time.
- For the prefetching cases, there is also a third component-stall time due to memory overheads caused by prefetching.
- Specifically, the stall time corresponds to two situations:
  - when the processor attempts to issue a prefetch but the prefetch issue buffer is already full,
  - when the processor attempts to execute a load or store when the cache tags are already busy with a prefetch fill.

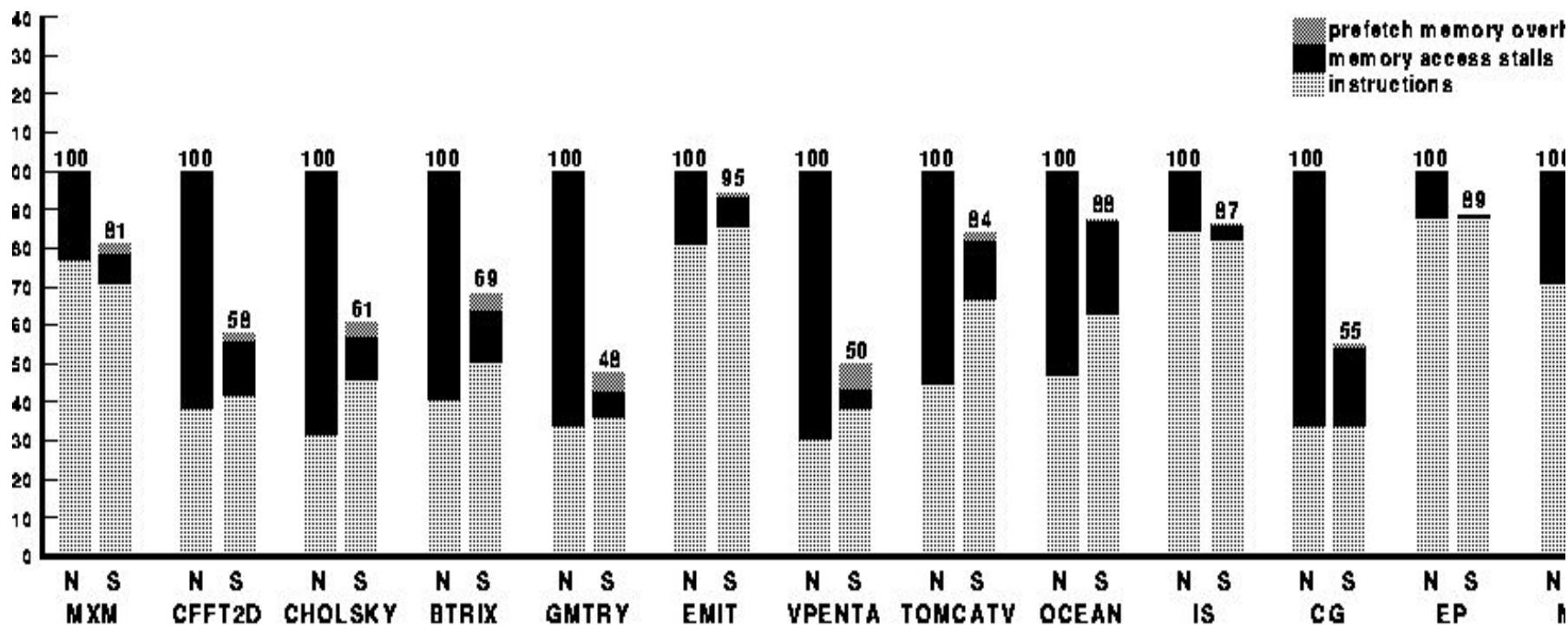


Figure 3: Overall performance of the selective prefetching algorithm (N = no prefetching, and S = selective prefetching).

- 
- Having established the benefits of prefetching, we now focus on the costs. Figure shows that the instruction overhead of prefetching causes less than a 15% increase in instruction count in over half of the benchmarks. In fact, in two of those cases (MXM and IS) the number of instructions actually decreased due to savings through loop unrolling.
  - In other cases (CHOLSKY, BTRIX, VPENTA, TOMCATV, OCEAN), the number of instructions increased by 25%to 50%.
  - Finally, the stalls due to prefetching memory overhead are typically small-never more than 15%of original execution time. In each case, we observe that the overheads of prefetching are low enough compared to the gains that the net improvement remains large.

- 
- As shown in Figure, the speedup in overall performance ranges from 5%to 100%, with 6 of the 13 benchmarks improving by over 45%. The memory stall time is significantly reduced in all the cases.
  - The miss penalty is reduced because even if a prefetched line is replaced from the primary cache before it can be referenced, it is still likely to be present in the secondary cache.
  - Also, the miss latency may be partially hidden if the miss occurs while the prefetch access is still in progress. Overall, 50%to 90%of the original memory stall cycles are eliminated.

# Distributed Coherent Caches - Dash Experience and Benefits of Caching

By  
Jishnu Prakasan

Distributed shared memory is an architectural approach that allows multiprocessors to support a single shared address space that is implemented with physically distributed memories. Hardware-supported distributed shared memory is becoming the dominant approach for building multiprocessors with moderate to large numbers of processors. Cache coherence allows such architectures to use caching to take advantage of locality in applications without changing the programmer's model of memory.

DASH is a scalable shared-memory multiprocessor whose architecture consists of powerful processing nodes, each with a portion of the shared-memory, connected to a scalable interconnection network. A key feature of DASH is its distributed direction-based cache coherence protocol. Unlike traditional snoopy coherence protocols, the DASH protocol does not rely on broadcast; instead it uses point-to-point messages sent between the processors and memories to keep caches consistent. Furthermore, the DASH system does not contain any single serialization or control point. While these features provide the basis for scalability, they also force a reevaluation of many fundamental issues involved in the design of a protocol. These include the issues of correctness, performance, and protocol complexity.

## Caching

Caching has long been recognized as a powerful performance enhancement technique in many areas of computer design. Most modern computer systems include a hardware cache between the processor and main memory, and many operating systems include a software cache between the file system routines and the disk hardware. In a distributed file system, where the file systems of several client machines are separated from the server backing store by a communications network, it is desirable to have a cache of recently used file blocks at the client, to avoid some of the communications overhead. In this configuration, special care must be taken to maintain consistency between the client caches, as some disk blocks may be in use by more than one client. For this reason, most current distributed file systems do not provide a cache at the client machine. Those systems that do place restrictions on the types of file blocks that may be shared, or require extra communication to confirm that a cached block is still valid each time the block is to be used. The Caching Ring is a combination of an intelligent network interface and an efficient network protocol that allows caching of all types of file blocks at the client machines. Blocks held in a client cache are guaranteed to be valid copies. We measure the style of use and performance improvement of caching in an existing file system, and develop the protocol and interface architecture of the Caching Ring. Using simulation, we study the performance of the Caching Ring and compare it to similar schemes using conventional network hardware.

# DASH

HTTP Adaptive Streaming (HAS) is gradually being adopted by Over The Top (OTT) content providers. In HAS, a wide range of video bitrates of the same video content are made available over the internet so that clients' players pick the video bitrate that best fit their bandwidth. Yet, this affects the performance of some major components of the video delivery chain, namely CDNs or transparent caches since several versions of the same content compete to be cached. In this context we investigate the benefits of a Cache Friendly HAS system (CF-DASH), which aims to improve the caching efficiency in mobile networks and to sustain the quality of experience of mobile clients. Firstly, we motivate our work by presenting a set of observations we made on large number of clients requesting HAS contents. Secondly we introduce the CF-Dash system and our testbed implementation. Finally, we evaluate CF-dash based on trace-driven simulations and testbed experiments. Our validation results are promising. Simulations on real HAS traffic show that we achieve a significant gain in hit-ratio that ranges from 15% up to 50%

# Benefits of Caching

The advantages of cache memory are as follows –

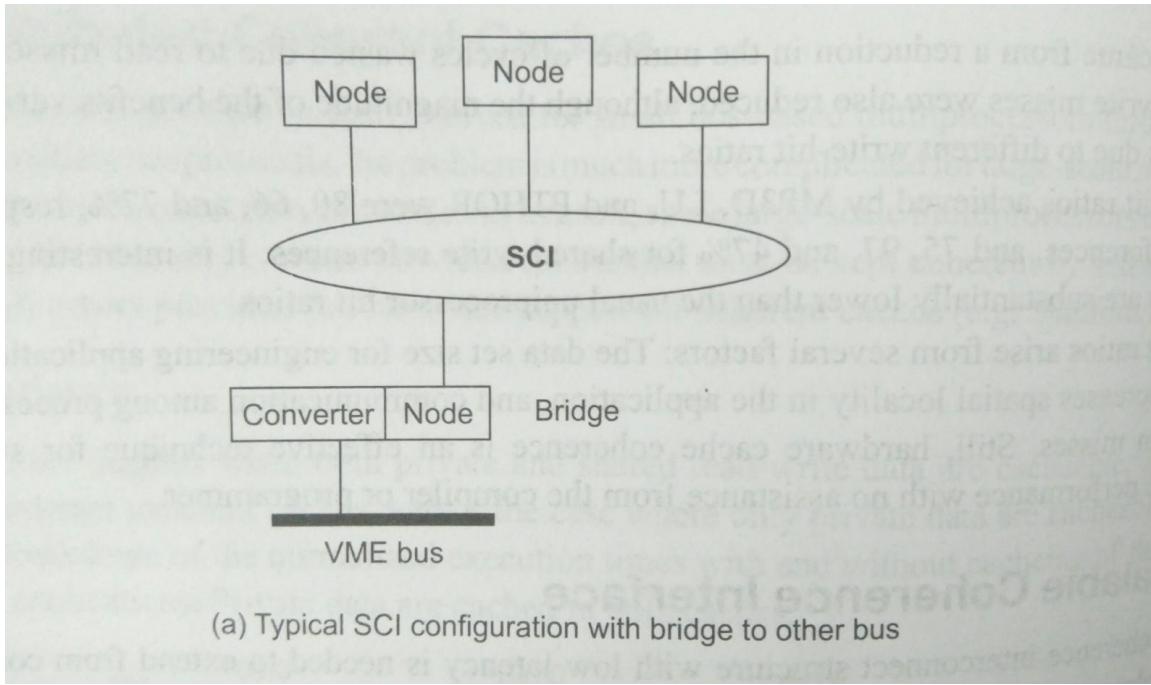
- Cache memory is faster than main memory.
- It consumes less access time as compared to main memory.
- It stores the program that can be executed within a short period of time.
- It stores data for temporary use.

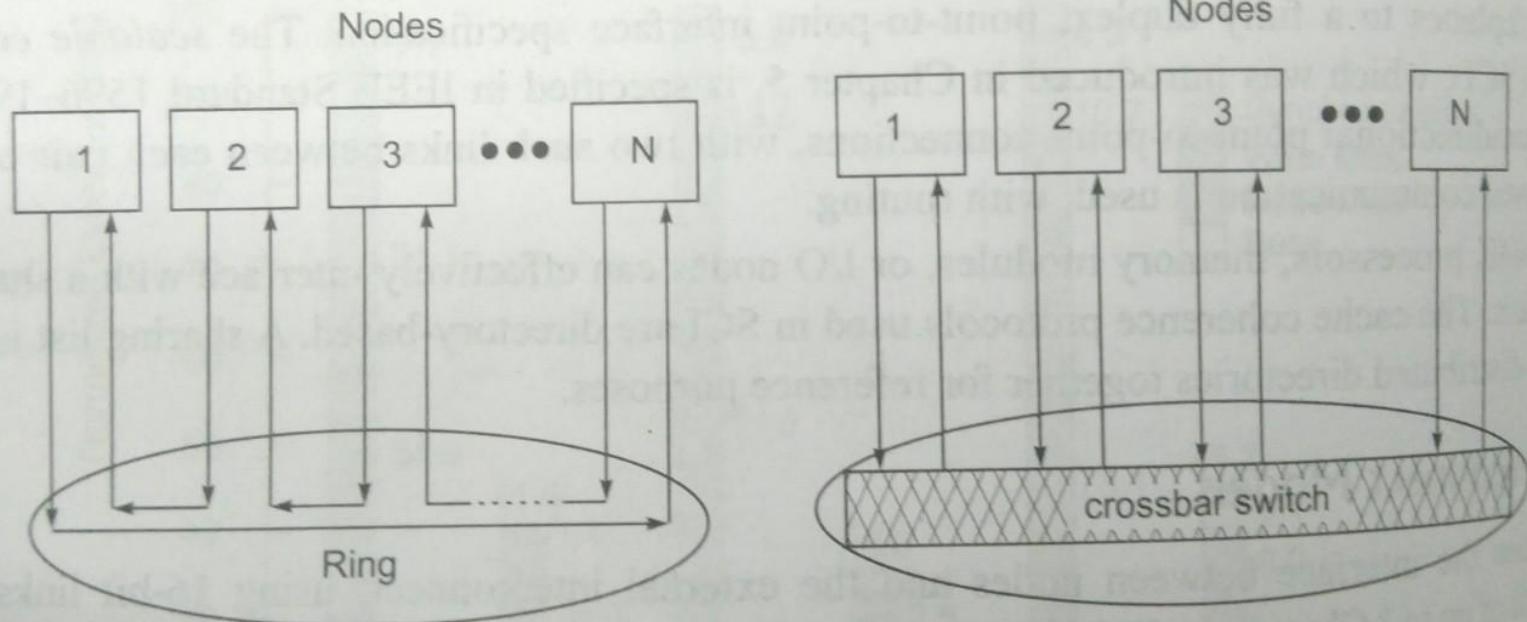
# SCI Interconnect Models

---

Jose D Maliakkal  
S7 cse  
36

- SCI defines the interface between nodes and the external interconnect, using a 16 bit links with a bandwidth of upto 1 gigabyte per link.
- As a result backplane buses have been replaced by unidirectional point to point links.
- Each SCI node can be a processor with attached memory and I/O devices.
- The SCI interconnector can be a ring structure or a crossbar switch.
- Each node has an input link and an output link which are connected from or to the SCI ring or crossbar.





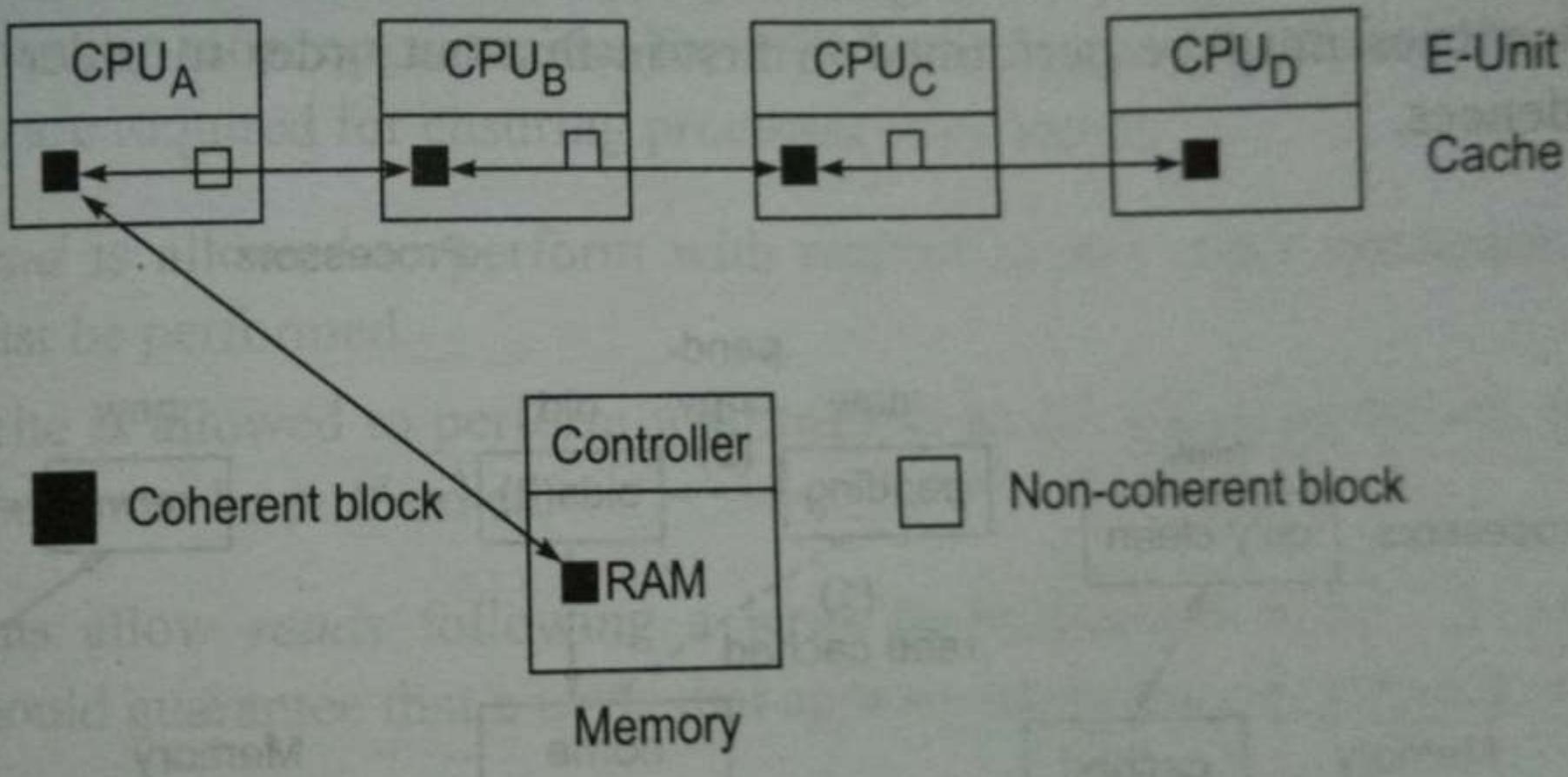
(b) A ring for point-to-point transactions

(b) A crossbar multiprocessor

# SHARING LIST STRUCTURES

- Sharing lists are used in SCI to build chain directories for cache coherence use. The length of the sharing lists is efficiently unbounded.
- Sharing list are dynamically created, pruned and destroyed.
- Each coherently cached block is entered onto a list of processors sharing the block.

## Processors



SCI cache coherence protocol with distributed directory

- Processors have the option of bypassing the coherence protocol for locally cashed data. Cache block of 64 bytes are assumed.
- By distributing the directories among the sharing processors, SCI avoids scaling limitations imposed by using a central directory.
- Communication among sharing processors are supported by heavily shared memory controllers.
- Other blocks may be locally cached and are not invisible to the coherence protocol.

- For every block address, the memory and cache entries have additional tag bits which are used to identify the first processor(head) in the sharing list and to link the previous and following nodes.
- Doubled linked list are maintained between processors in the sharing list, with forward and backward pointer(double arrow in each link in the figure).
- Non coherent copies may also be made coherent by page-level control.
- However, such higher level software coherent protocol are beyond the scope of the SCI standard.

# Effect Of Release Consistency

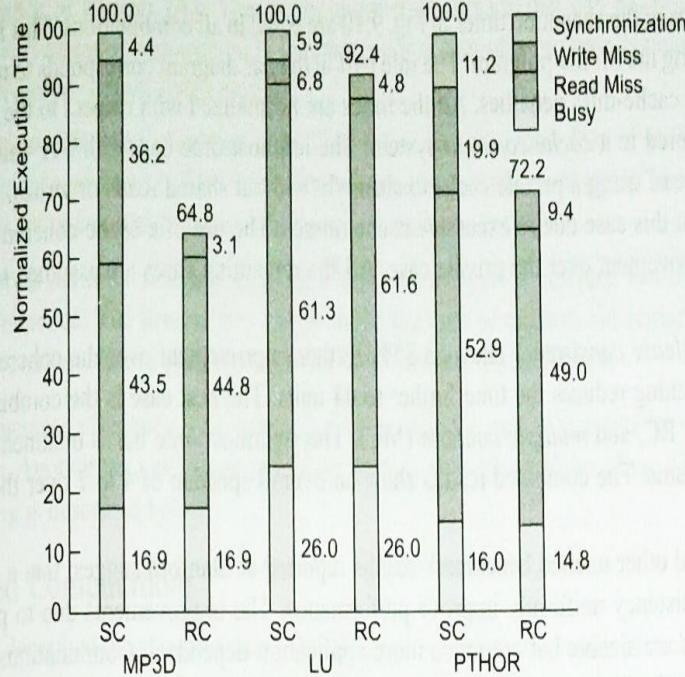


Fig. 9.9 Effect of relaxing the shared-memory model from sequential consistency (SC) to release consistency (RC) (Courtesy of Gupta et al, Proc. Int. Symp. Comput. Archit., Toronto, Canada, May 1991)

- RC removes all idle time due to write-miss latency (from fig).
- The gains are large in MP3D and PTHOR since the write-miss time constitutes a large portions of the executions time under SC(35 and 20% respectively), while the gain is small in LU due to the relatively small write-miss time under SC(7%).

# Effect Of Combining Mechanisms - 1

- Busy parts of the execution times are equal in all combinations.
- The idle part in the bar diagram corresponds to memory latency and includes all cache-miss penalties.
- All the times are normalized with respect to the execution time(100 units) required in cache-coherent system.
- Left most time bar – the worst case of using a private cache exclusively without shared reads or writes.

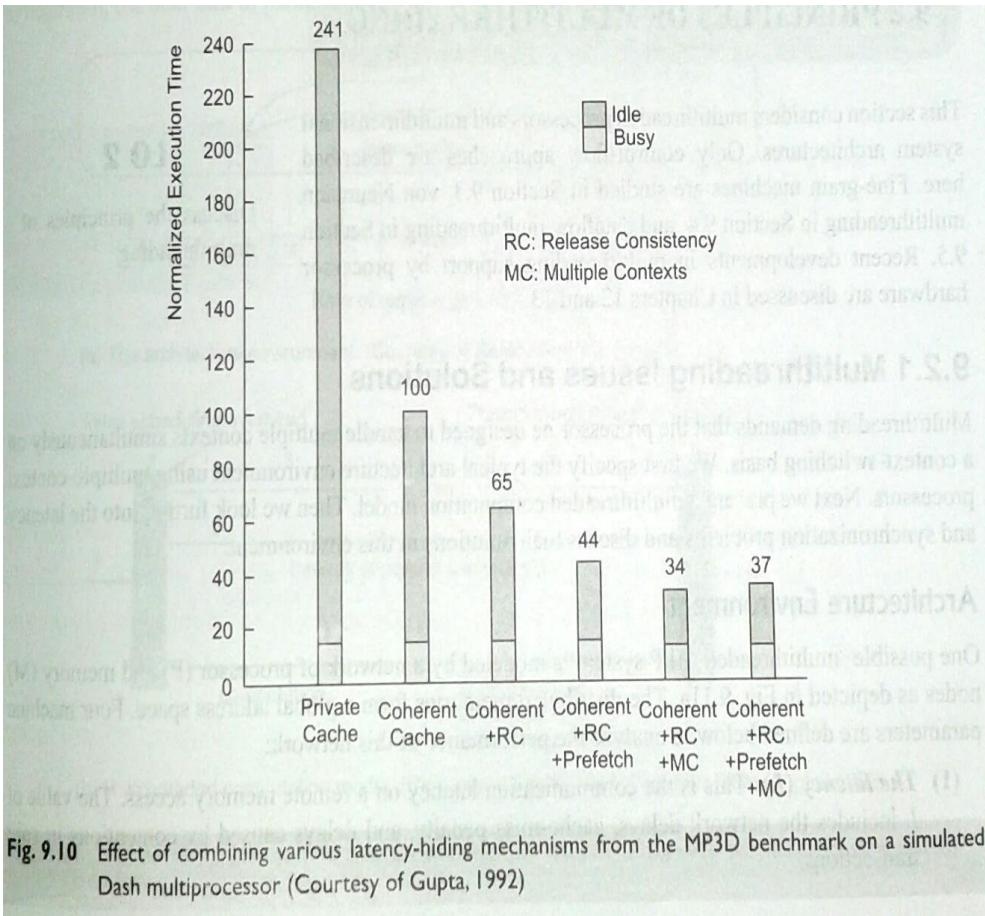


Fig. 9.10 Effect of combining various latency-hiding mechanisms from the MP3D benchmark on a simulated Dash multiprocessor (Courtesy of Gupta, 1992)

# **Effect Of Combining Mechanisms - 2**

- Long overhead is experienced in this case due to excessive cache misses.
- All the remaining cases are assumed to use hardware coherent caches.
- The use of release consistency shows 35% further improvement over the coherent systems.
- The adding of prefetching reduces the time further to 44 units.
- The best case is the combination of using coherent caches, RC, and multiple contexts(MC).

# **Effect Of Combining Mechanisms - 3**

- The rightmost time bar is obtained from applying all 4 mechanisms.
- The combined results show an overall speedup of 4 to 7 over the case of using private caches.
- Coherent cache and relaxed consistency uniformly improve performance.
- The improvements due to prefetching and multiple contexts are sizable but are much more applicant-dependent.
- Combinations of the various latency-hiding mechanisms generally attain a better performance than each one on its own.

# Problems of Asynchrony

Submitted by

Maria Baby

S7,CSE

Roll No: 42

# Asynchronous

- In computer programs, asynchronous operation means that a process operates independently of other processes, whereas synchronous operation means that the process runs only as a result of some other process being completed or handing off operation.
- A typical activity that might use a synchronous protocol would be a transmission of files from one point to another.

- As each transmission is received, a response is returned indicating success or the need to resend. Each successive transmission of data requires a response to the previous transmission before a new one can be initiated.
- In telecommunication signaling within a network or between networks, an asynchronous signal is one that is transmitted at a different clock rate than another signal.

- Asynchronous data transfer: sender provides a synchronization signal to the receiver before starting the transfer of each message
- does not need clock signal between the sender and the receiver
- slower data transfer rate

# Advantages

- The character is self contained & Transmitter and receiver need not be synchronized
- Transmitting and receiving clocks are independent of each other

# Disadvantages

- Overhead of start and stop bits
- False recognition of these bits due to noise on the channel

# Applications

- If channel is reliable, then suitable for high speed else low speed transmission
- Most common use is in the ASCII terminals

# Multithreading Solutions and Distributed Cacheing

Submitted by  
Marvel Monson  
S7,CSE  
Roll No : 43

# Multithreading

- Multithreading is a type of execution model that allows multiple threads to exist within the context of a process such that they execute independently but share their process resources.
- A thread maintains a list of information relevant to its execution including the priority schedule, exception handlers, a set of CPU registers, and stack state in the address space of its hosting process.
- Multithreading is also known as threading.

Operating systems use threading in two ways:

- Pre-emptive multithreading, in which the context switch is controlled by the operating system. Context switching might be performed at an inappropriate time, Hence, a high priority thread could be indirectly pre-empted by a low priority thread.
- Cooperative multithreading, in which context switching is controlled by the thread. This could lead to problems, such as deadlocks, if a thread is blocked waiting for a resource to become free.

# Distributed Caching

- Caching has become the de facto technology to boost application performance as well as reduce costs.
- By caching frequently used data in memory – rather than making database round trips – application response times can be dramatically improved.

- Distributed caching is simply an extension of this concept, but the cache is configured to span multiple servers.
- It's commonly used in cloud computing and virtualised environments, where different servers give a portion of their cache memory into a pool which can then be accessed by virtual machines. This also means it's a much more scalable option.

# What makes distributed caching effective?

- Performance
- Scalability
- Availability
- Manageability
- Simplicity
- Affordability

# **CONTEXT SWITCHING IN MULTITHREADING**

By,  
**MERIN JOSEPH**  
**S7 CSE**  
**45**

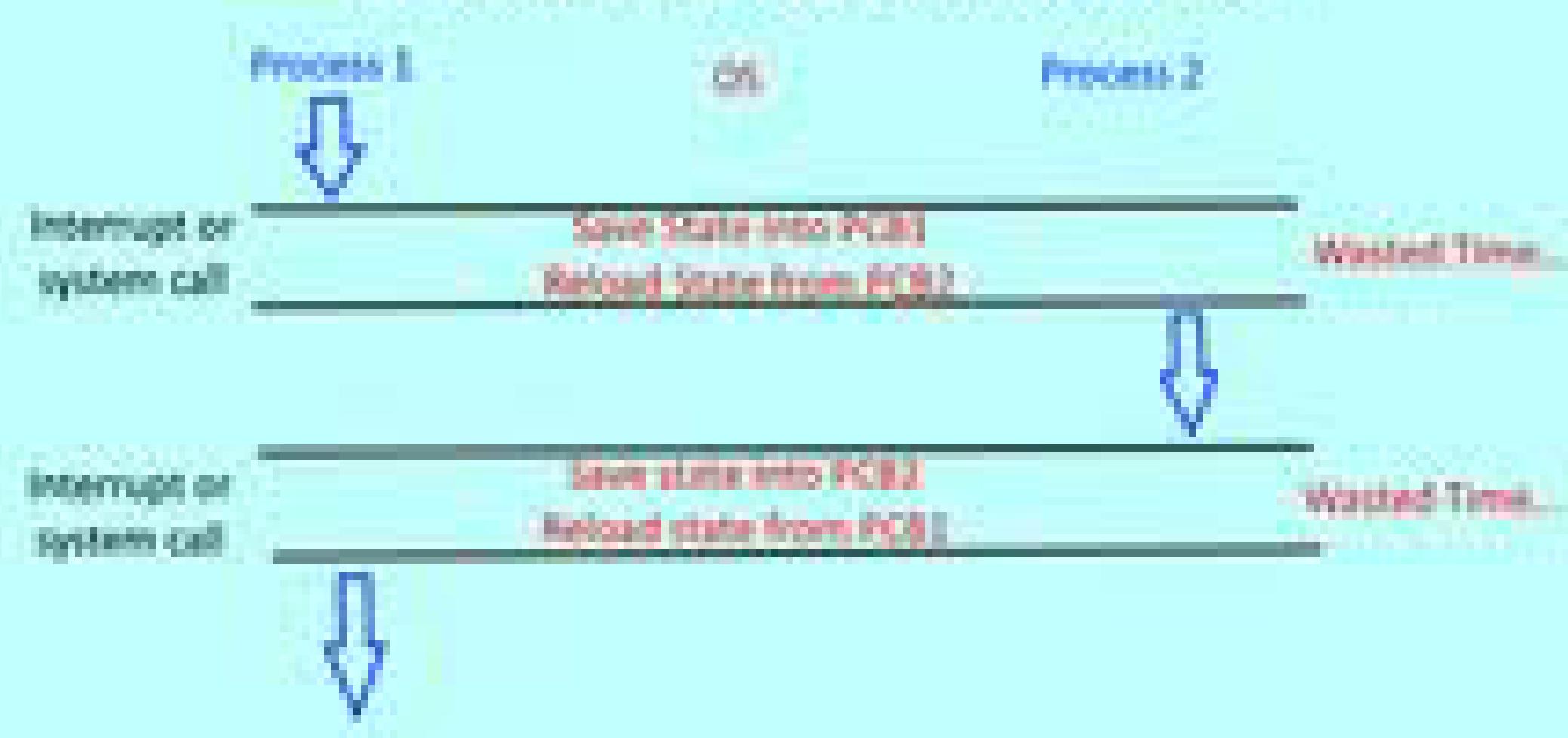
# Context switching

- Is an essential feature of multithreading.
- A context switch is the process of storing the state of a process or of a thread, so that it can be restored and execution resumed from the same point later.
- This allows multiple processes to share a single CPU.
- It is the switching of CPU from one process or thread to another .

- Context switching in detail can be explained as the kernel performing the following activities with regard to processes on the CPU:
- (1) suspending the progression of one process and storing the CPU's state for that process somewhere in memory
- (2) retrieving the context of the next process from memory and restoring it in the CPU's register
- (3) returning to the location indicated by the program counter (i.e., returning to the line of code at which the process was interrupted) in order to resume the process.

# Context Switch

Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process.



# Multithreading -Processor Efficiency

ASSIGNMENT-02

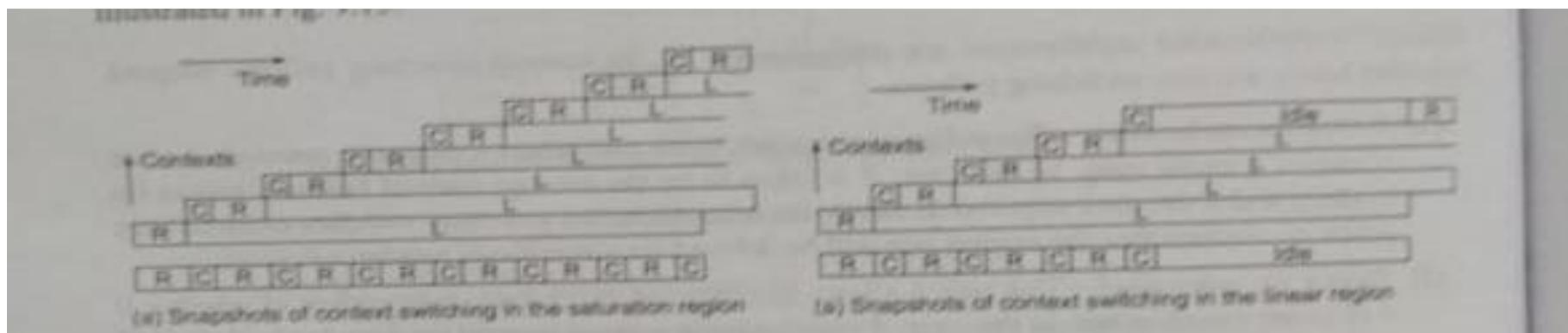
Devika Satheesh

Rollno:46

- No context switch and no switch overhead
- Efficiency of Single threaded machine is

$$E=R/R+L$$

- This shows performance degradation



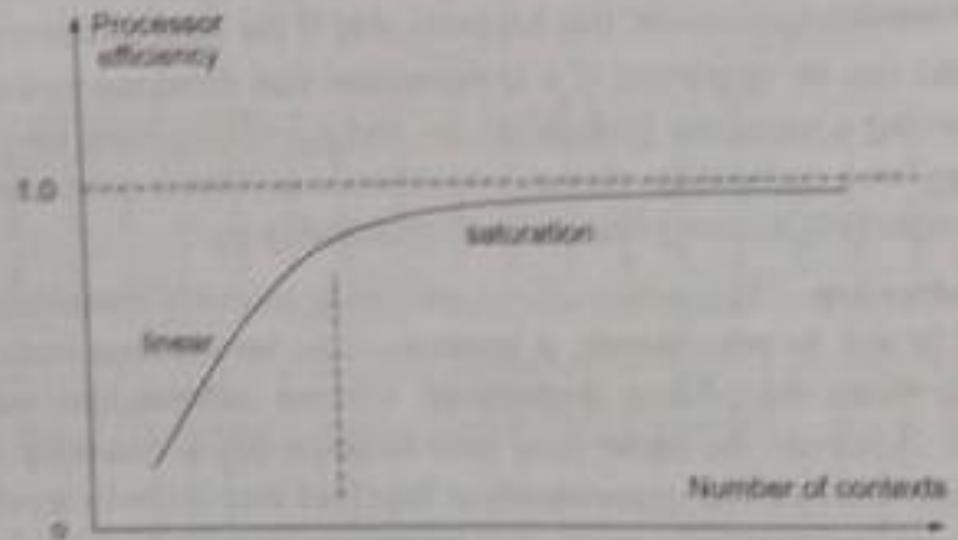
# Saturation Region

- Processor operates with maximum utilization.
- Cycle of renewal process in this case is  $R+C$  and efficiency is:

$$E=R/R+C$$

- Observe that efficiency in saturation is independent of latency
- Saturation is achieved when  $(N-1)(R+C) > L$ . This gives saturation point under constant.

$$N=(L/R+C)+1$$

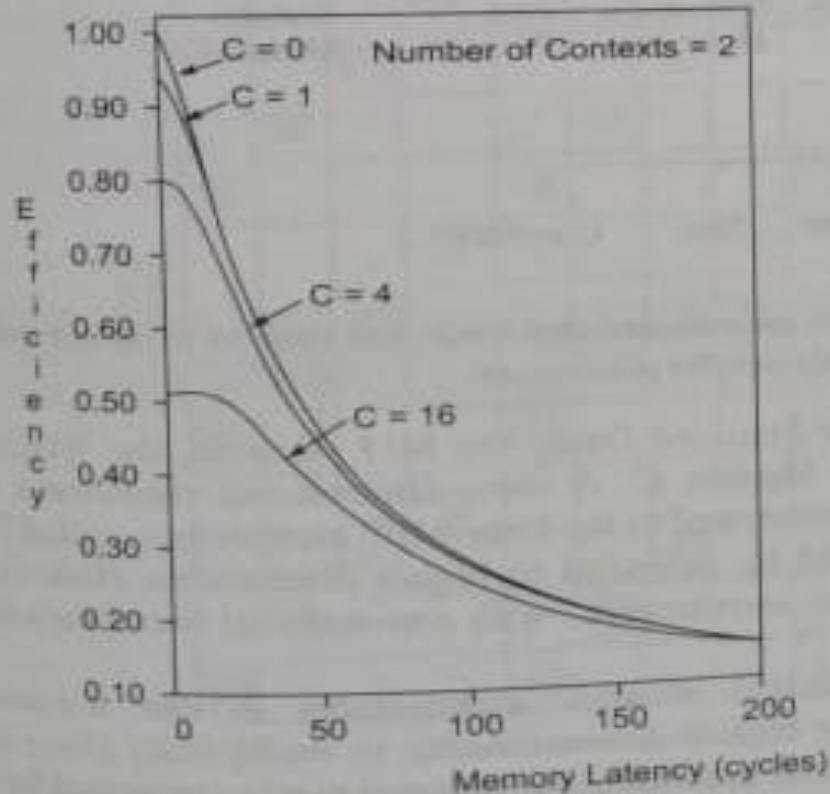


(c) Efficiency curve

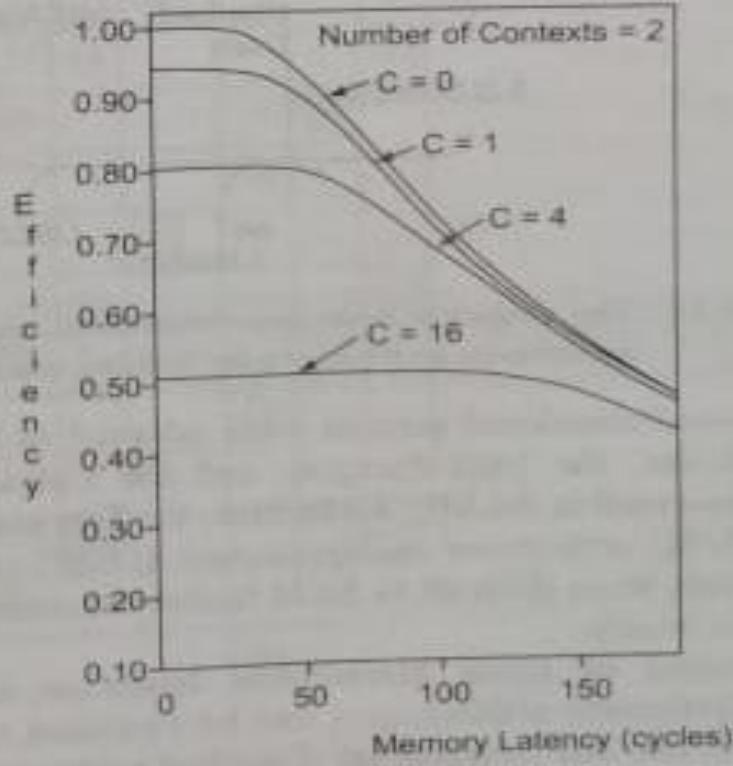
# Linear Region

- When number of contexts is below saturation point, there may be no ready contexts which, so the processor will experience idle cycles.
- The time required to switch to ready context, execute it until a remote reference is issued, and process the reference is equal to  $R+C+L$ .
- Assuming  $N$  is below saturation point, during this time all other contexts have a turn in processor

$$E = NR/R + C + L$$



(a) Two contexts per processor



(b) Six contexts per processor

Fig. 9.16 Processor efficiency of a multithreaded architecture (Courtesy of R. Saavedra, D. E. Culler, and T. von Eicken, 1992)

# MULTIDIMENSIONAL DATA ARCHITECTURE

POORNASREE R MOHAN

S7, CSE

ROLL NO:47

# CONTENT:-

- Introduction of Multidimensional architecture
- Component of Multidimensional architectures
- Types of architectural models
  - [A]. Data Cube Model.
  - [B]. Star Schema Model.
  - [C]. Snow Flake Schema Model.
  - [D]. Fact Constellations.

# INTRODUCTION

- The Multi- Dimensional Model was developed for implementing data warehouse and data marts
- It provide both a mechanism to store data and a way for computer data analysis.

# COMPONENT OF MULTIDIMENSIONAL ARCHITECTURE

The two primary component of dimensional model are Dimensions and Facts.

- Dimensions:- Texture Attributes to analyses data.
- Facts:- Numeric volume to analyze business.

# TYPES OF ARCHITECTURES

- [A]. Data Cube Model.
- [B]. Star Schema Model.
- [C]. Snow Flake Schema Model.
- [D]. Fact Constellations .

# DATA CUBE MULTI-DIMENSIONAL ARCHITECTURE

- When data is grouped or combined together in multidimensional matrices called Data Cubes.
- In Two Dimension :- row & Column or Products &fiscal quarters.
- In Three Dimension:- one regions, products and fiscal quarters

# STAR SCHEMA MODEL

- It is also known as Star Join Schema.
- It is the simplest style of data warehouse schema.
- It is called a Star Schema because the entity relationship diagram of this Schema resembles a star, with points radiating from central table.
- A star query is a join between a fact table and a no. of dimension table.
- Each dimension table is joined to the fact table using primary key to foreign key join but dimension table are not joined to each other.
- A typical fact table contain key and measure.

# SNOW FLAKE SCHEMA

- It is slightly different from a star schema in which the dimensional tables from a star schema are organized into a hierarchy by normalizing them.
- The Snow Flake Schema is represented by centralized fact table which are connected to multiple dimensions.
- The Snow Flaking effecting only affecting the dimension tables and not the fact tables.

# DIFFERENCE B/W STAR SCHEMA AND SNOW FLAKE:-

## SNOW FLAKE SCHEMA

### STAR SCHEMA

Star Schema dimension are de-normalized with each dimension being represented in single table.

Snowflake Schema dimension are normalized into multiple related tables.

# FACT CONSTELLATIONS

- It is set of fact tables that share some dimensions tables.
- It limits the possible queries for the data warehouse.

# Wisconsin Multicube

- A large-scale, shared-memory multiprocessor architecture that uses a snooping cache protocol over a grid of buses.
- Each processor has a conventional (SRAM) cache optimized to minimize memory latency and a large (DRAM) snooping cache optimised to reduce bus traffic and to maintain consistency.
- Large snooping cache guarantee traffic on busses generated by I/O and access of shared data.

# Programmer's view

- Multi-A set of processors having access to a common shared memory with no notion of geographical locality.
- Thus writing software, including the operating system, should be a straightforward extension of those techniques being developed for multis.

- The interconnection topology allows for a cache-coherent protocol for which most bus requests can be satisfied with no more than twice the number of bus operations required of a single-bus multi.
- The total symmetry guarantees that there are no topology-induced bottlenecks.
- The total bus bandwidth grows in proportion to the product of the number of processors and the average path length.

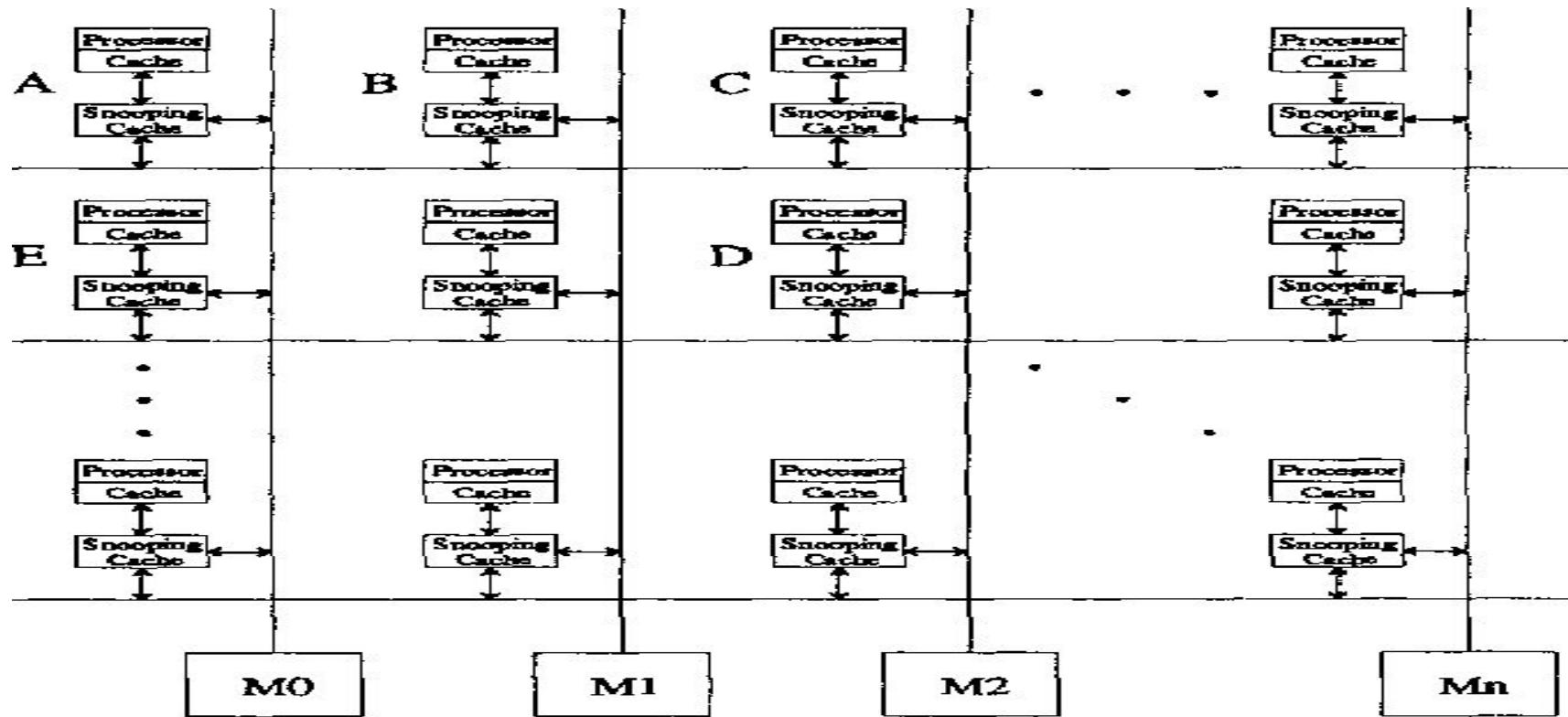


Figure 1: The Wisconsin Multicube

<sup>1</sup>This cache is external to the processor, which likely will include an on-chip cache as well.

<sup>2</sup>Of course, the enormous total memory and computing power of this machine will permit

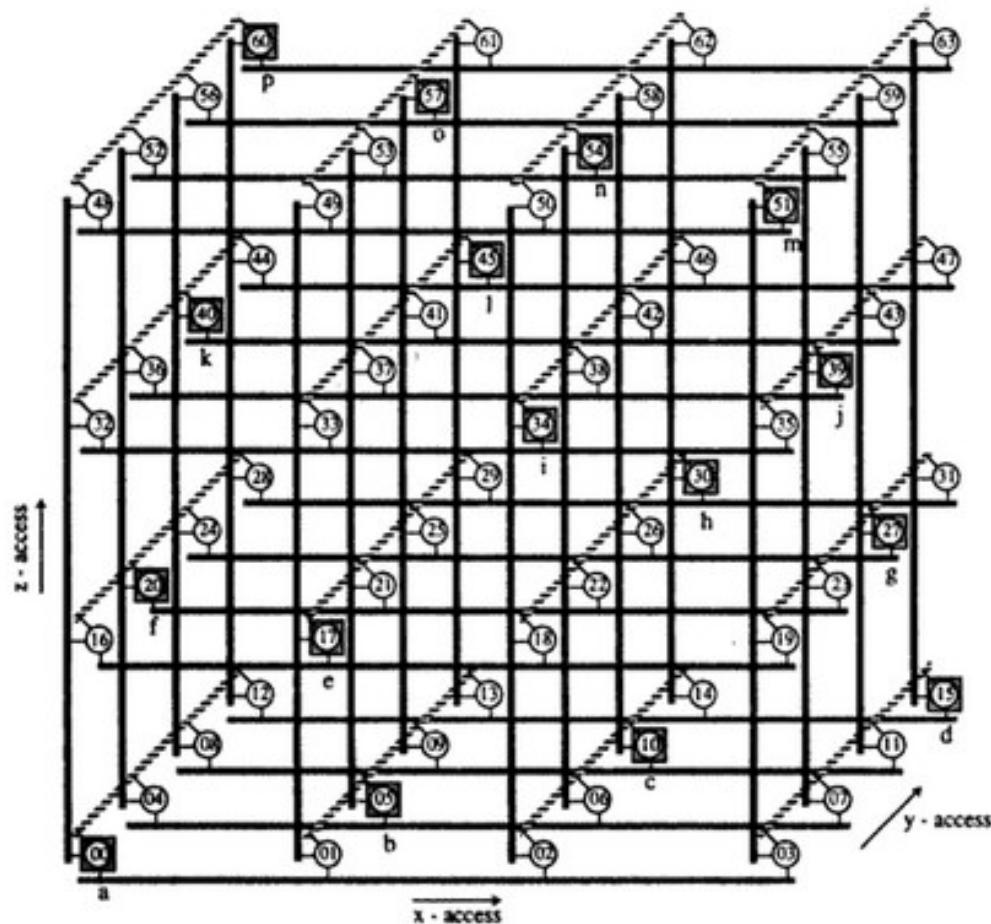
# ORTHOGONAL MULTIPROCESSOR

Riya Binny  
CSE S7 49

# ORTHOGONAL MULTIPROCESSOR

- A generalised orthogonal multiprocessor is denoted as an OMP( $n, k$ ), where
  - n = Dimension
  - k = Mutliplicity
- There are  $p=k^{(n-1)}$  and  $m=k^n$  memory modules in the system where  $p \gg n$  and  $p \gg k$ .
- The system uses p memory buses, each spanning into n dimensions. But only one dimension is used in a given memory cycle.
- There are k memory modules attached to each spanning bus.

- The OMP architecture supports special purpose computations in which data sets can be regularly arranged as matrices.
- OMP is well suited for SPMD(single program, multiple data) operations in which n processors are synchronised at the memory access level when data sets are vectorized in matrix format.



(c) The 3-D OMP(3,4) architecture. (Processors are labeled a, b, ..., p; memory modules are labeled 00, 01, ..., 63)

$\text{OMP}(n, k)$	$p = k^{n-1}$	$m = k^n$
OMP(2, 8)	8	64
OMP(2, 16)	16	256
OMP(3, 8)	64	512
OMP(3, 16)	256	4096
OMP(4, 8)	512	4096
OMP(4, 16)	4096	65,536
OMP(5, 16)	65,536	1,048,576

Note:  $p$  = number of processors;  $m$  = number of memory modules.

- Each module is connected to n out of p buses through an n-way switch.
- Dimension n corresponds to the number of accessible ports that each memory module has.
- Each memory module is shared by n out of  $p=k^{n-1}$  processors.

# Multidimensional extensions

ROBIN JOSEPH

S7 CSE

ROLL NO : 50

# Multidimensional extensions

Used to :

- Identify dimensions by which objects are perceived or evaluated.
- Position the objects with respect to those dimensions.
- Make positioning decisions for new and old products.

# Basic concepts of Multidimensional scaling (MDS)

- MDS uses proximities among different objects as input.
- Proximities data is used to produce a geometric configuration of points in a two-dimensional space as output.
- The fit between the derived distances and the two proximities in each dimension is evaluated through a measure called stress.
- The approximate number of dimensions required to locate objects can be obtained by plotting stress values against the number of dimensions.

# Attribute based MDS

## Advantages:

- Attributes can have diagnostic and operational values.
- Attribute data is easier for the respondents to use.

## Disadvantages:

- If the list of attributes is not accurate and complete, the study will suffer.
- The respondents may not perceive or evaluate objects in terms of underlying attributes.

# Application of MDS with non-attribute data

- Reflect the perceived similarity of two objects from the respondents perspectives.
- Perceptual map is obtained from the average similarity ratings.
- Able to find the smallest number of dimensions for which there is a reasonably good fit between the input similarity rankings and the rankings of the distance between objects in the resulting space.

# MIT J-Machine - Architecture and MDP Design

Roshin Jojo  
S7 CSE 52

# Introduction

- » The J-Machine (Jellybean-Machine) was a parallel computer designed by the MIT Concurrent VLSI Architecture group in conjunction with the Intel Corporation. The machine used "jellybean" parts—cheap and multitudinous commodity parts, each with a processor, memory, and a fast communication interface—and a novel network interface to implement fine grained parallel programs

# Architecture

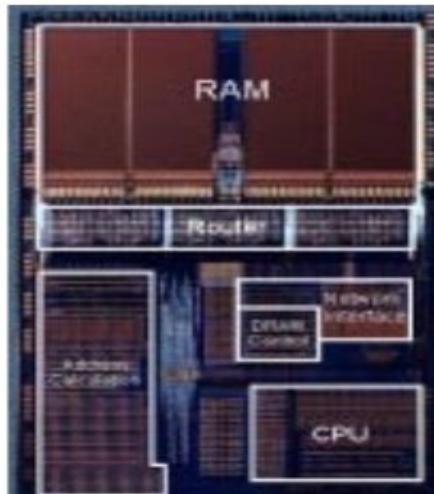
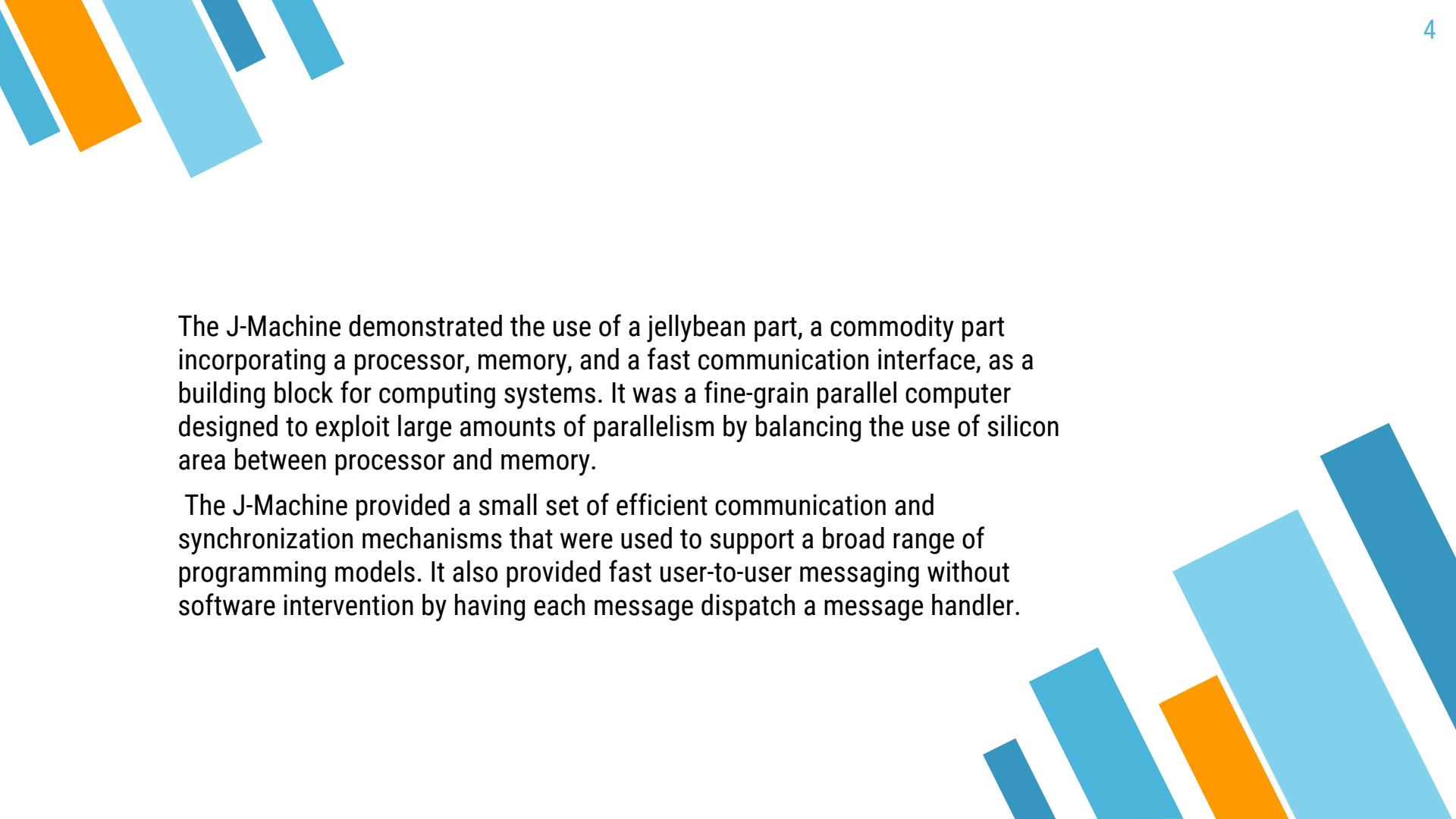


Figure 1: MDP Die Photo



Figure 2: J-Machine

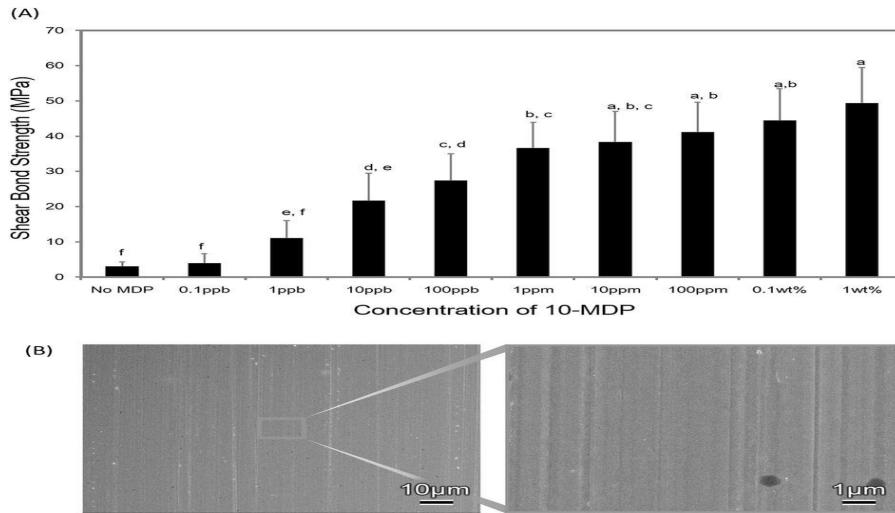


The J-Machine demonstrated the use of a jellybean part, a commodity part incorporating a processor, memory, and a fast communication interface, as a building block for computing systems. It was a fine-grain parallel computer designed to exploit large amounts of parallelism by balancing the use of silicon area between processor and memory.

The J-Machine provided a small set of efficient communication and synchronization mechanisms that were used to support a broad range of programming models. It also provided fast user-to-user messaging without software intervention by having each message dispatch a message handler.

# MDP Design

Each processing node of the J-Machine is composed of a single Message-Driven Processor (MDP) chip and three 1M 4 DRAM chips. The resulting node is 2" by 3". Each J-Machine processor board contains 64 MDP chips along with their external memories. The board measures 26" by 20.5". Each pair of nodes shares a set of elastomeric conductors to communicate with the corresponding nodes on the boards above and below in the board stack. A total of 48 elastomeric connectors held in four connector holders provide 1420 electrical connections between adjacent boards. Of these connections, 960 are used for signalling and the remaining 460 are ground returns. The use of elastomeric conductors enables the J-Machine to achieve a very high network bisection bandwidth (a peak rate of 14.4 Gbits/sec) in a very small cross sectional area (2 ft<sup>2</sup>) and at the same time keep the length of the longest channel wire in a 1024-node machine under four inches. A stack of sixteen processor boards contains 1024 processing nodes (12,500 MIPS peak) and 1Gbyte of DRAM memory in a volume of 4 ft<sup>3</sup>, and dissipates 1500W (1.5W per node).



# THANKS!

---

# MIT J-Machine - Instruction Set Architecture and Communication Support

---

Sara Jacob  
S7 CSE 53

# Instruction-Set Architecture

- The MDP extended a conventional microprocessor instruction-set architecture with instructions to support parallel processing.
- The instruction set contained fixed-format, three address instructions.
- Two 17-bit instructions fit into each 36-bit word with 2 bits reserved for type checking.

- Separate register sets were provided to support rapid switching among 3 execution levels
  - 1) background
  - 2) priority 0 (P0)
  - 3) priority 1 (P1)
- MDP executed at background level while no message created a task, and initiated execution upon message arrival at P0 & P1 level depending on message priority.
- P1 level had higher priority than P0 level.

# Communication Support

- MDP transmitted a message using a series of SEND instruction.
- For sending a four-word message using 3 variants of SEND instruction

SEND R0,0 ; send net address

SEND2 R1,R2,0 ; header and receiver

SEND2E R3,[3,A3],0 ; selector and communication end message

- First SEND instruction reads the absolute address of the destination node in  $\langle X, Y, Z \rangle$  format from R0 and forwards it to the network hardware.
- SEND2 reads first 2 words of the message out of registers R1 and R2 and enqueues them for transmission.
- Final instruction enqueues 2 additional words of data, one from R3 and one from memory.

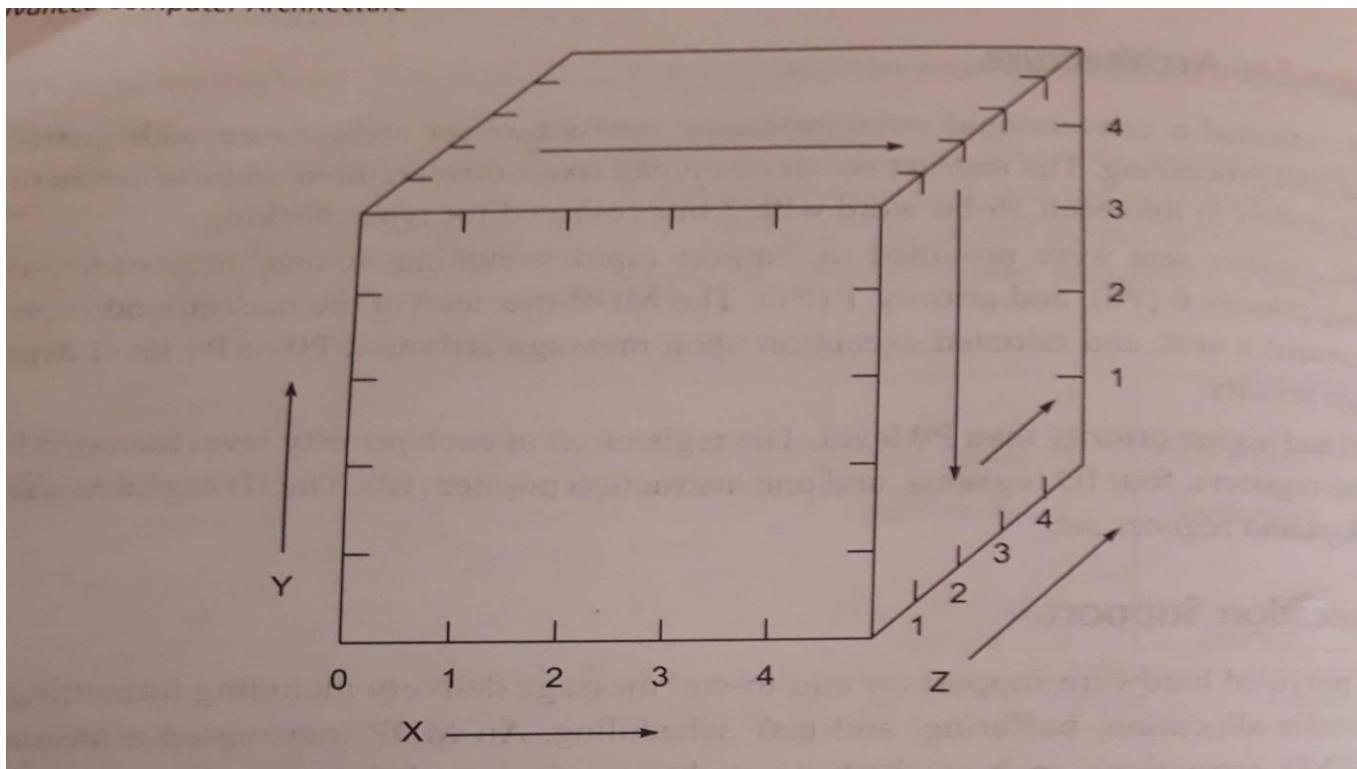
# MIT J-machine Message Format and routing

Sherin George

Roll no : 54

- The J-machine used deterministic dimension-order **E-cube routing**.
- Here all messages are routed first in the x-dimension, then in the y-dimension, and then in the z-dimension.
- Since messages routed in dimension order and messages running in opposite directions along the same direction cannot block, resource cycles are avoided, making the network **deadlock-free**

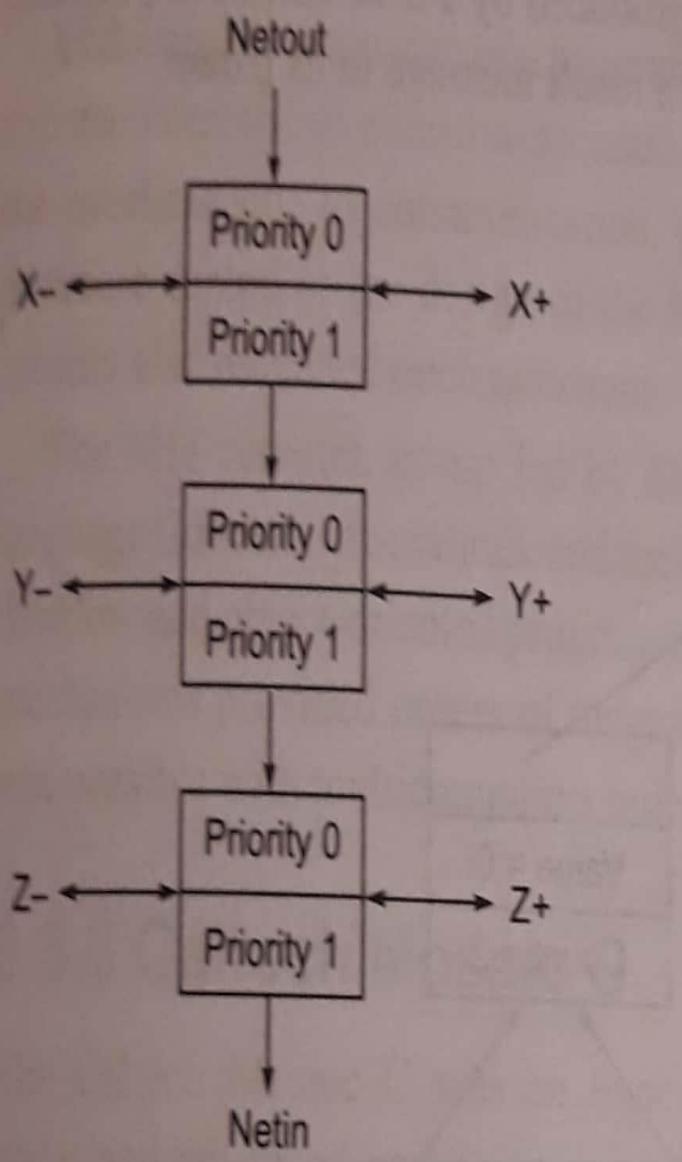
- The below figure shows E-cube routing from node(1,5,2) to node(5,1,4) on a 6-ary 3-cube.



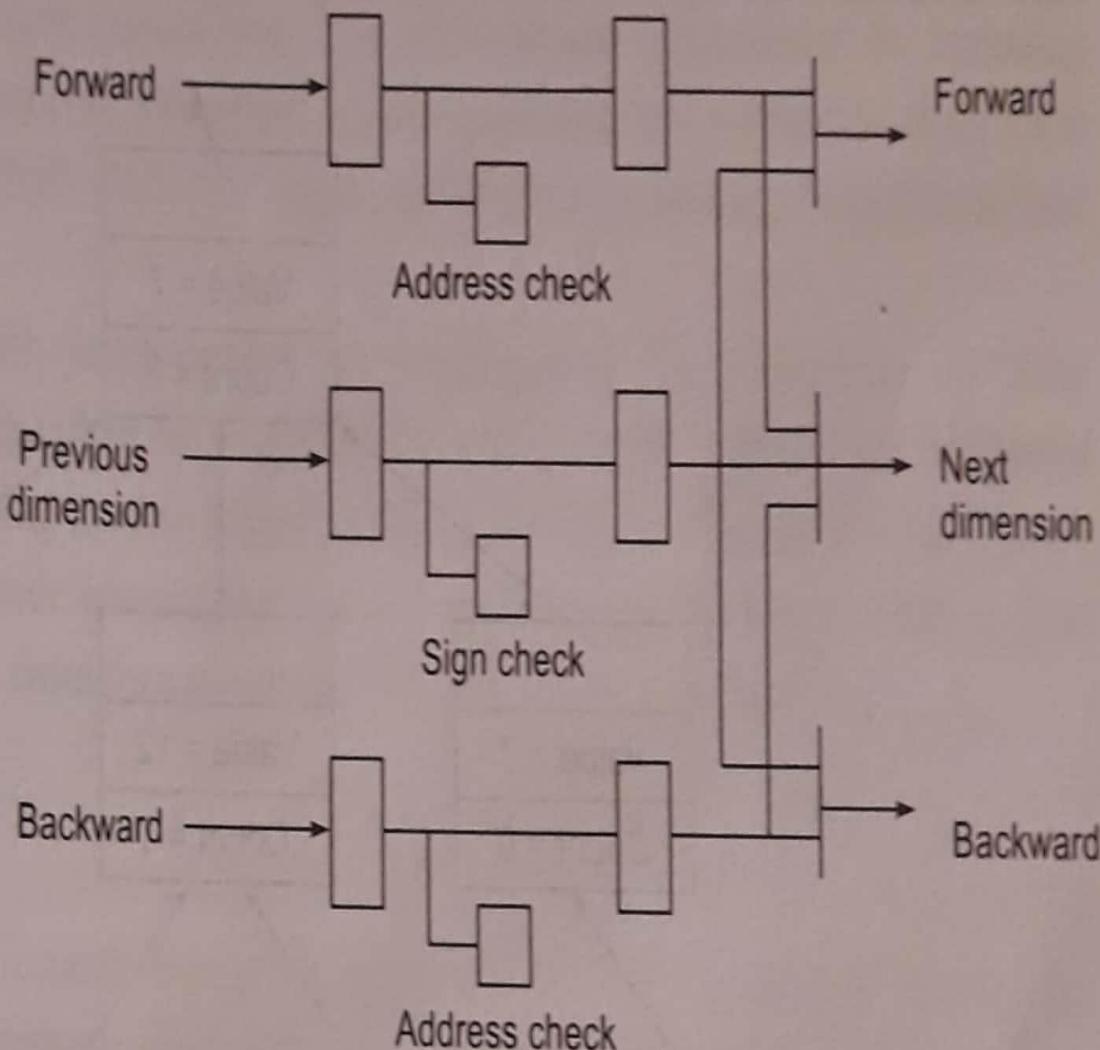
# Router Design

- The routers formed the switches in a MIT J-machine network and delivered messages to their destinations.
- It contain three independent routers, one for each bidirectional dimension of the network.
- Each router contains two separate virtual networks with different priorities that shared the same physical channels.

- The priority-1 network could preempt the wires even if the priority-0 network was congested or jammed.
- Each of the 18 router paths contained buffers, comparators, and output arbitration.
- A message entering the dimension competed with messages continuing in the dimension at a two-to-one switch.



(a) Dual-priority levels per dimension  
in the router



(a) Each priority with forward, reverse, and previous  
data paths to the next dimension.

- Two priorities of messages shared the physical wires but used completely separate buffers and routing logic.
- This allowed priority-1 messages to proceed through blockages at priority 0.

# MIT J-machine - Router Design, Synchronization and Research Issues



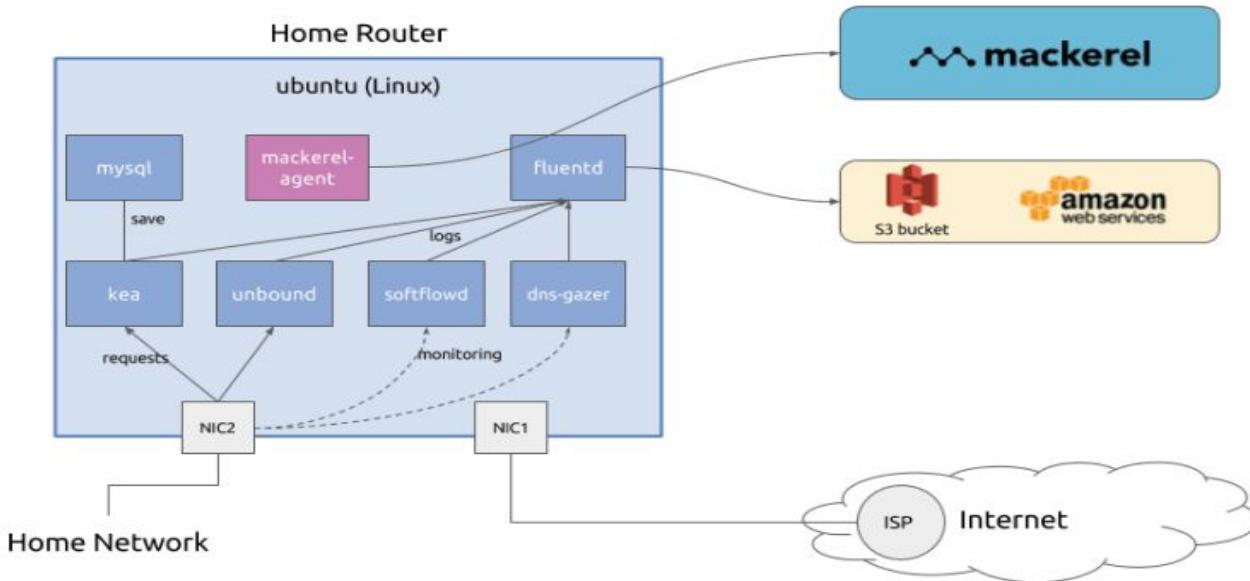
Sreehari Rajeev  
CSE,S7,55



# INTRODUCTION

- The J-machine project was started at MIT in about 1988 as an experiment in message-passing computing based on work that Bill Dally did at Caltech for his doctoral dissertation.
- The work was driven by the VLSI philosophy "processors are cheap" and "memory is expensive". This philosophy is based on a idealistic view of VLSI economics, in which the cost of a function is based on the VLSI area dedicated to it.
- The design of the J-machine incorporated several novel technologies.

# ROUTER DESIGN



# SYNCHRONIZATION

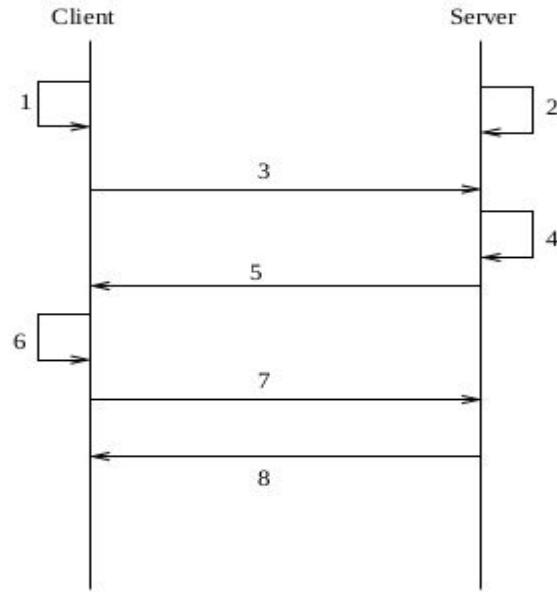


Figure 2: Simplified diagram of the synchronization



# RESEARCH ISSUES

- J machine is an exploratory research project.
- Rather than being specialized in a single model of computation, the MDP incorporates primitive mechanisms for effective communications, synchronization and naming.
- The machine is used as a platform for software experiments in fine-grain parallel programming.



**THANK YOU...**

# Caltech Mosaic C

SREELAKSHMI S.K

S7 , CSE

ROLL NO : 56

# INTRODUCTION

- The Caltech Mosaic C is an experimental, fine-grain multicomputer that employs single-chip nodes and advanced packaging technology to demonstrate the performance/cost advantages of the fine-grain-multicomputer architecture.

# ARCHITECTURE

- Each Mosaic node includes 64KB of single-clock-cycle dynamic RAM
- 2KB of self-test and bootstrap ROM
- An 11MIPS processor
- A packet interface
- A 60MB/s, two-dimensional, self-timed router

# OPERATIONS

- The operating node is a single, 9.25mm\Theta10.00mm, 1.2 m-feature-size, CMOS chip
- CMOS chip operate at V<sub>dd</sub> = 5V
- operates at 30MHz and dissipates 0.5W
- These chips are packaged by tape-automated bonding in 8\Theta8 arrays on circuit boards that can, in turn, be composed in two dimensions to construct arbitrarily large arrays of nodes

# PROGRAMMING FEATURES

- The Mosaic programming system consists of a compiler for a source programming notation called C + -, and a distributed runtime system.
- C + - is an extension of C ++ that supports concurrent processes in much the same way that C ++ supports program objects
- The runtime system, which is written in C + -, provides copy less message handling, highly distributed resource management, automatic process placement, and scheduling.

# APPLICATIONS

- The Mosaic is intended both for large-scale-computing and for embedded-systems applications.
- A 16K-node Mosaic system is being constructed at Caltech to explore the programmability and application span of the architecture for large-scale computations.
- The ATOMIC local-area network is an early example of an embedded-system application of Mosaic components.

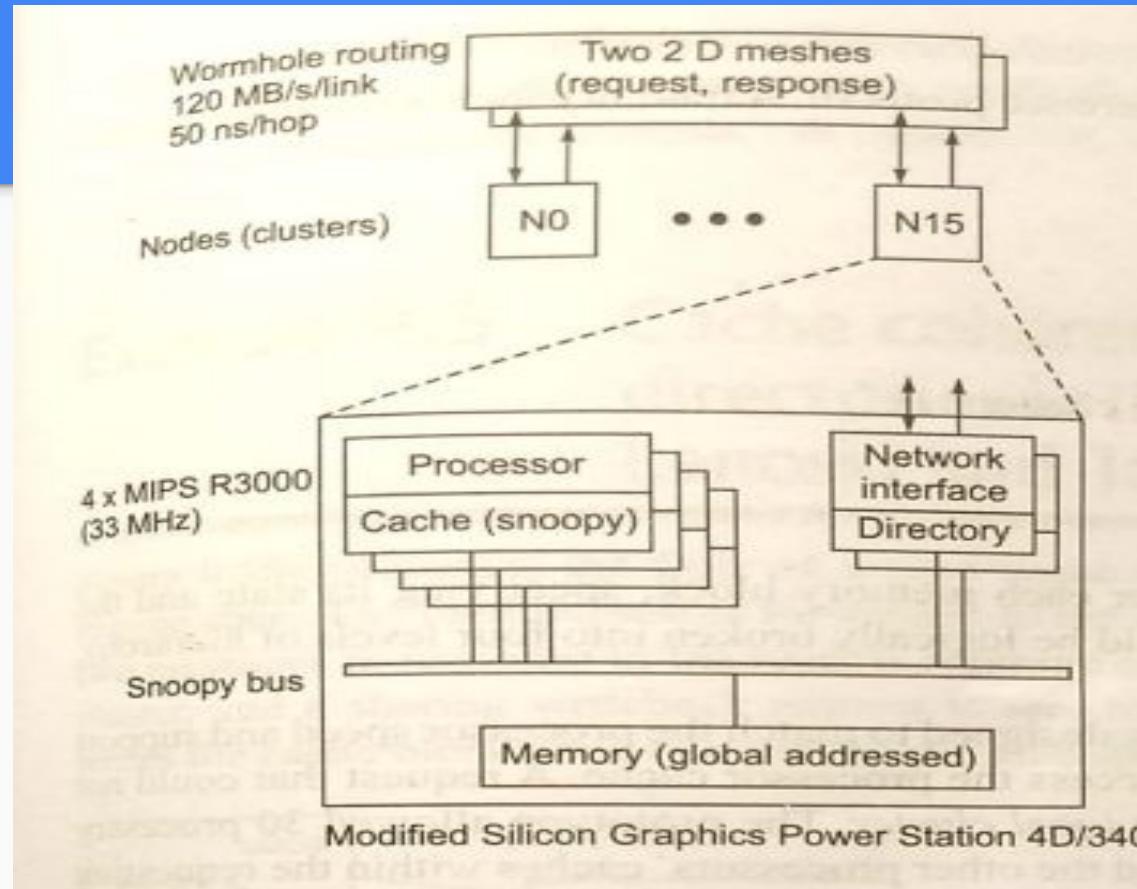
# Stanford Dash Multiprocessor Prototype Architecture

Submitted by  
V J HARAN

## Main Features

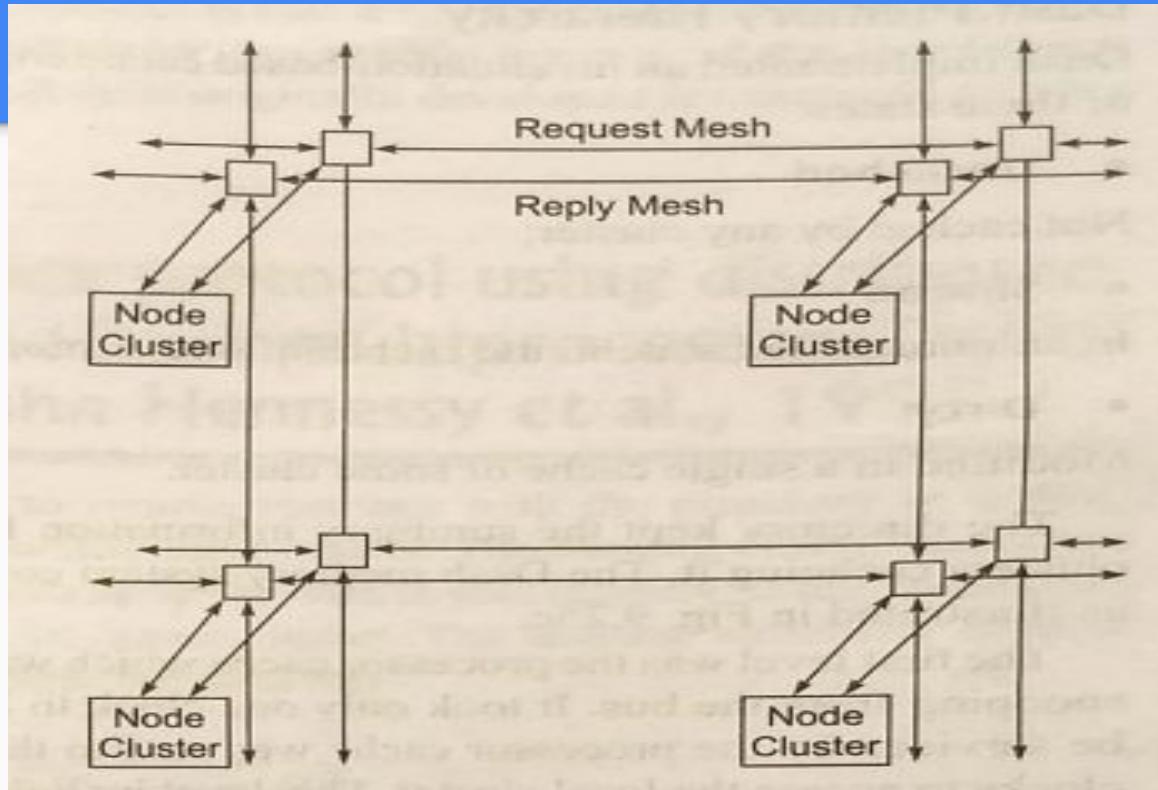
- Developed by John Hennessey and co-workers at Stanford University, 1988
- DASH(Directory Architecture for Shared Memory)
- A scalable high-performance machine
- Single address space, coherent caches & distributed memories
- Incorporates upto 64 MIPS R3000/R3010 microprocessors with 16 clusters of 4 PEs each
- Cluster hardware : Silicon Graphics 4D/340 nodes
- Pair of Wormhole routed mesh networks are used for interconnection among 16 multiprocessor clusters
- Channel width : 16 bits
- System is scalable to support 100's of 1000's of processors

## Prototype node implementation

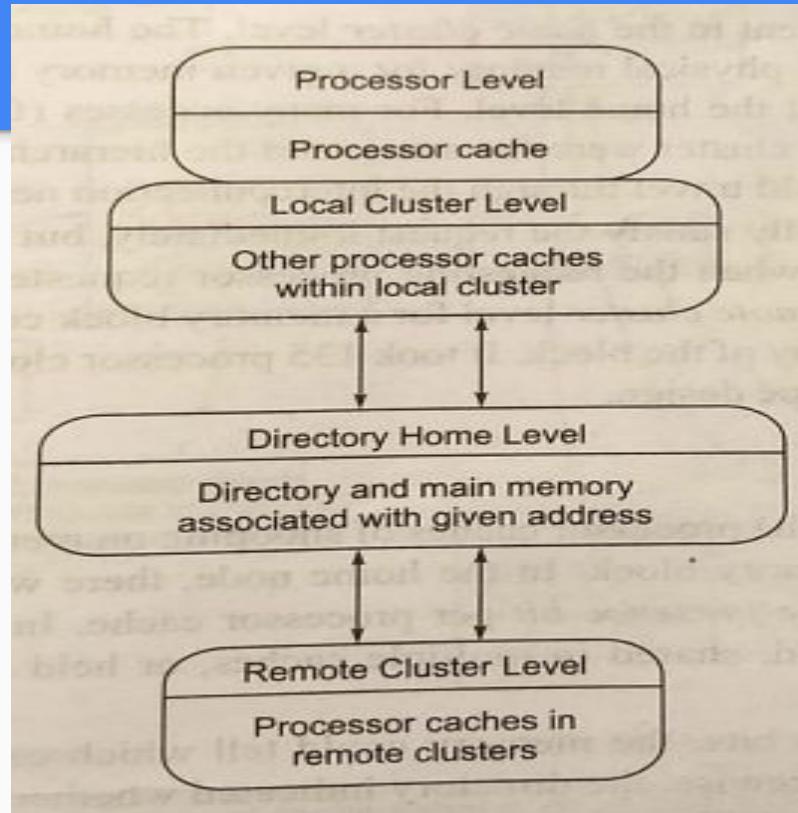


- To use the 4D/340 in the Dash ---
  1. Designed new board to support intercluster interface and directory memory
  2. Modified central bus arbiter to accept mask from directory
- Mask held off a processor's retry until a remote request was serviced
- New directory control board contains ---
  - Directory memory
  - Intercluster coherence state machines and buffers
  - Local section of global interconnection network
- Permitted incremental porting
- Factors for improved performance includes mechanisms for reducing and tolerating latency, well-designed I/O capabilities

## Block diagram of 2x2 mesh interconnect



# Logic Memory Hierarchy



# THE DATA FLOW MODEL (OF A COMPUTER)

- Von Neumann model: An instruction is fetched and executed in **control flow order**
  - As specified by the **instruction pointer**
  - Sequential unless explicit control flow instruction
- Dataflow model: An instruction is fetched and executed in **data flow order**
  - i.e., when its operands are ready
  - i.e., there is **no instruction pointer**
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies “who” should receive the result
    - An instruction can “fire” whenever all operands are received
  - Potentially many instructions can execute at the same time
    - Inherently more parallel

# CSA ASSIGNMENT 2

## Evolution of Dataflow Computers, Dataflow Graphs and Static vs Dynamic Dataflow

Vysakh Prasannan

S7,CSE

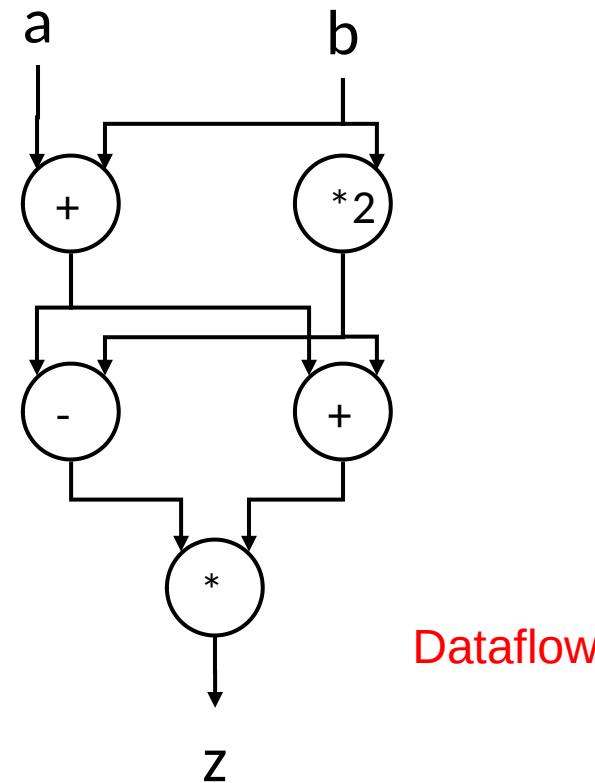
Roll No:59

# VON NEUMANN VS DATAFLOW

- Consider a Von Neumann program
  - What is the significance of the program order?
  - What is the significance of the storage locations?

```
v <= a + b;  
w <= b * 2;  
x <= v - w  
y <= v + w  
z <= x * y
```

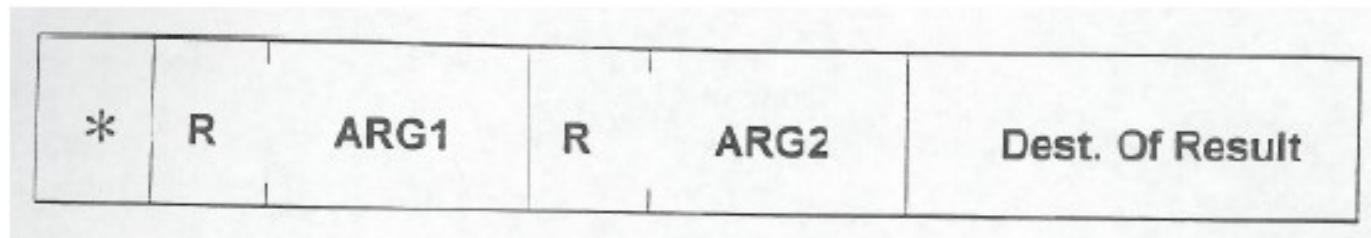
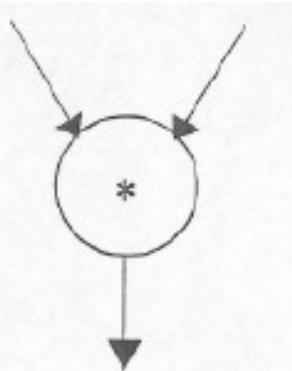
Sequential



- Which model is more natural to you as a programmer?

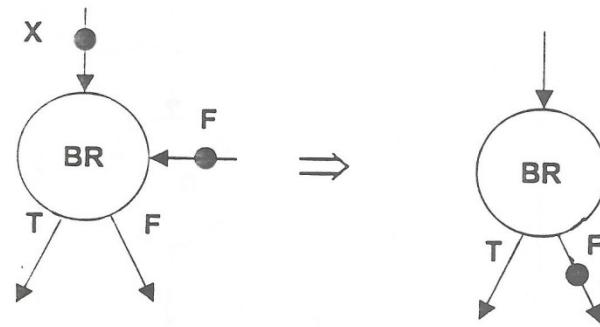
# MORE ON DATA FLOW

- In a data flow machine, a program consists of data flow nodes
  - A data flow node fires (fetched and executed) when all its inputs are ready
    - i.e. when all inputs have tokens
- Data flow node and its ISA representation

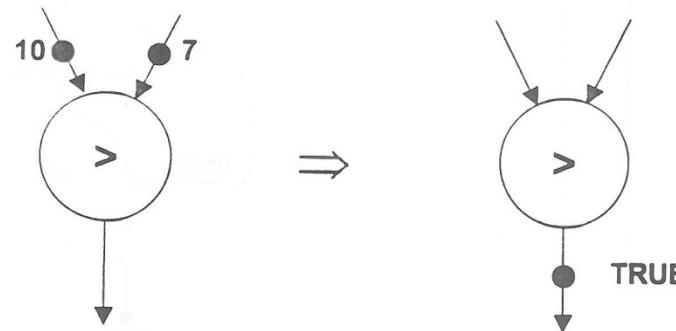


# DATA FLOW NODES

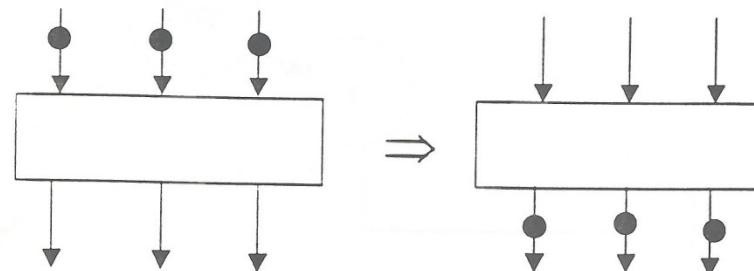
\**Conditional*



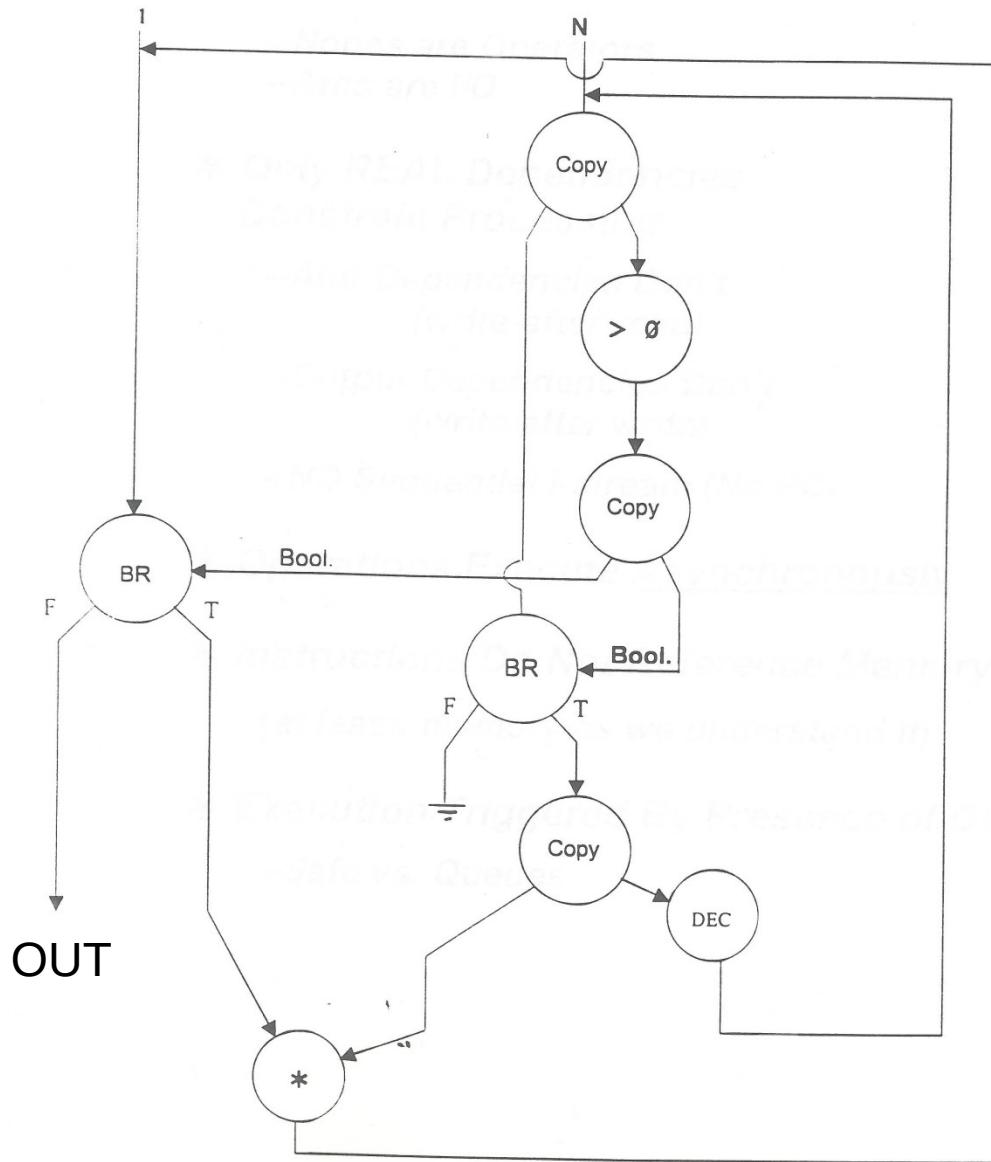
\**Relational*



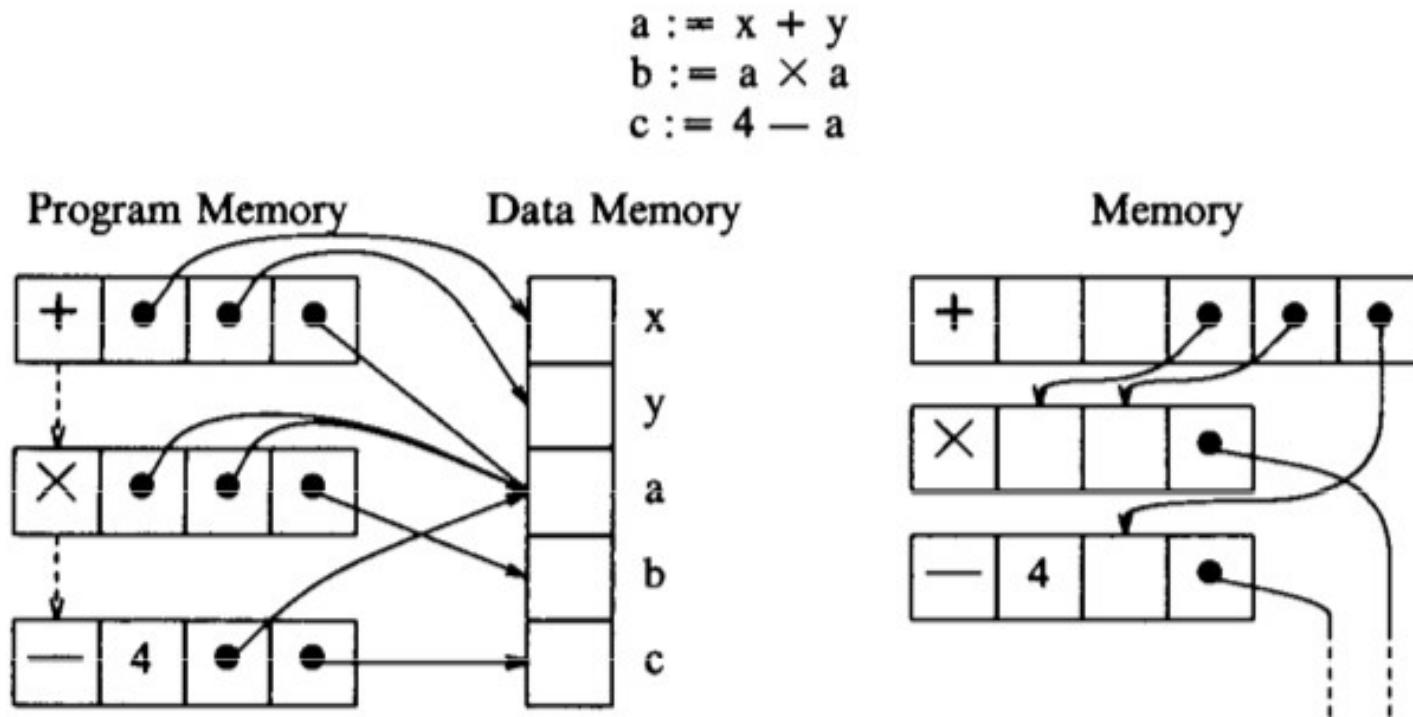
\**Barrier Synch*



# AN EXAMPLE DATA FLOW PROGRAM

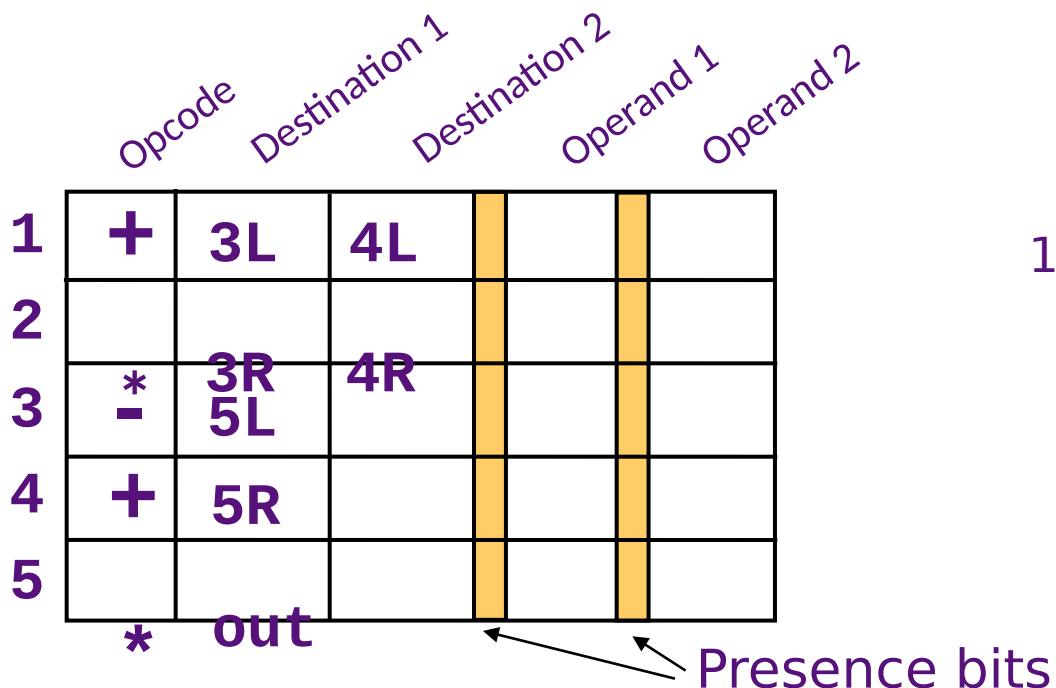


# CONTROL FLOW VS. DATA FLOW

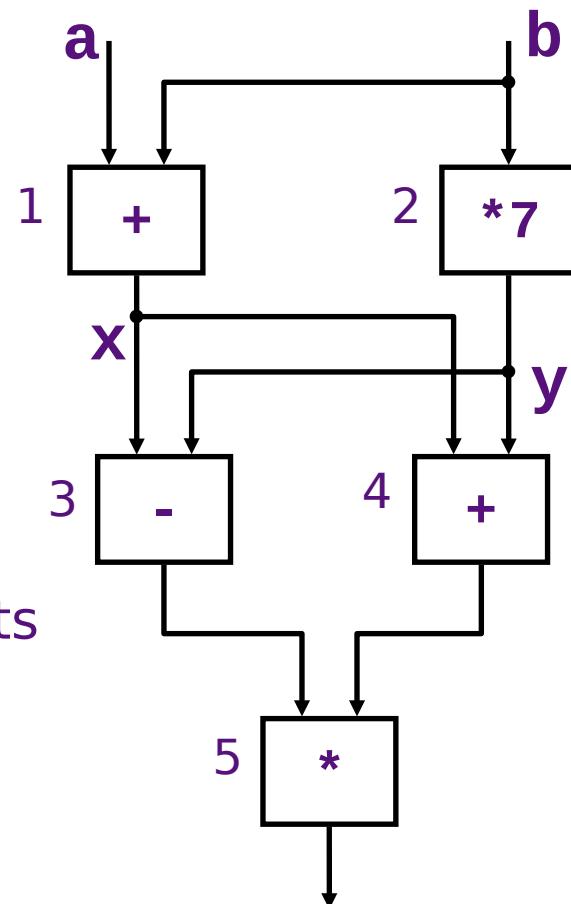


**Figure 2.** A comparison of control flow and dataflow programs. On the left a control flow program for a computer with memory-to-memory instructions. The arcs point to the locations of data that are to be used or created. Control flow arcs are indicated with dashed arrows; usually most of them are implicit. In the equivalent dataflow program on the right only one memory is involved. Each instruction contains pointers to all instructions that consume its results.

# Dataflow Machine: Instruction Templates



Each arc in the graph has an operand slot in the program



Dennis and Misunas, "A Preliminary Architecture for a Basic Data Flow Processor," ISCA 1974.

# DATA FLOW SUMMARY

- Availability of data determines order of execution
- A data flow node fires when its sources are ready
- Programs represented as data flow graphs (of nodes)
- Data Flow at the ISA level has not been (as) successful
- Data Flow implementations under the hood (while preserving sequential ISA semantics) have been successful
  - Out of order execution
  - Hwu and Patt, “**HPSm, a high performance restricted data flow architecture having minimal functionality**,” ISCA 1986.

# DATA FLOW CHARACTERISTICS

- Data-driven execution of instruction-level graphical code
  - Nodes are operators
  - Arcs are data (I/O)
  - As opposed to control-driven execution
- Only real dependencies constrain processing
- No sequential I-stream
  - No program counter
- Operations execute asynchronously
- Execution triggered by the presence of data

# DATA FLOW ADVANTAGES/DISADVANTAGES

- Advantages
  - Very good at exploiting **irregular parallelism**
  - Only real dependencies constrain processing
- Disadvantages
  - Debugging difficult (no precise state)
    - Interrupt/exception handling is difficult (what is precise state semantics?)
  - Implementing dynamic data structures difficult in pure data flow models
  - Too much parallelism? (Parallelism control needed)
  - High bookkeeping overhead (tag matching, data storage)
  - Instruction cycle is inefficient (delay between dependent instructions), memory locality is not exploited

# ANOTHER WAY OF EXPLOITING PARALLELISM

- SIMD:
  - Concurrency arises from performing the **same operations on different pieces of data**
- MIMD:
  - Concurrency arises from performing **different operations on different pieces of data**
    - Control/thread parallelism: execute different threads of control in parallel  multithreading, multiprocessing
  - Idea: Use multiple processors to solve a problem

# FLYNN'S TAXONOMY OF COMPUTERS

- Mike Flynn, “**Very High-Speed Computing Systems**,” Proc. of IEEE, 1966
- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- **MISD**: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor

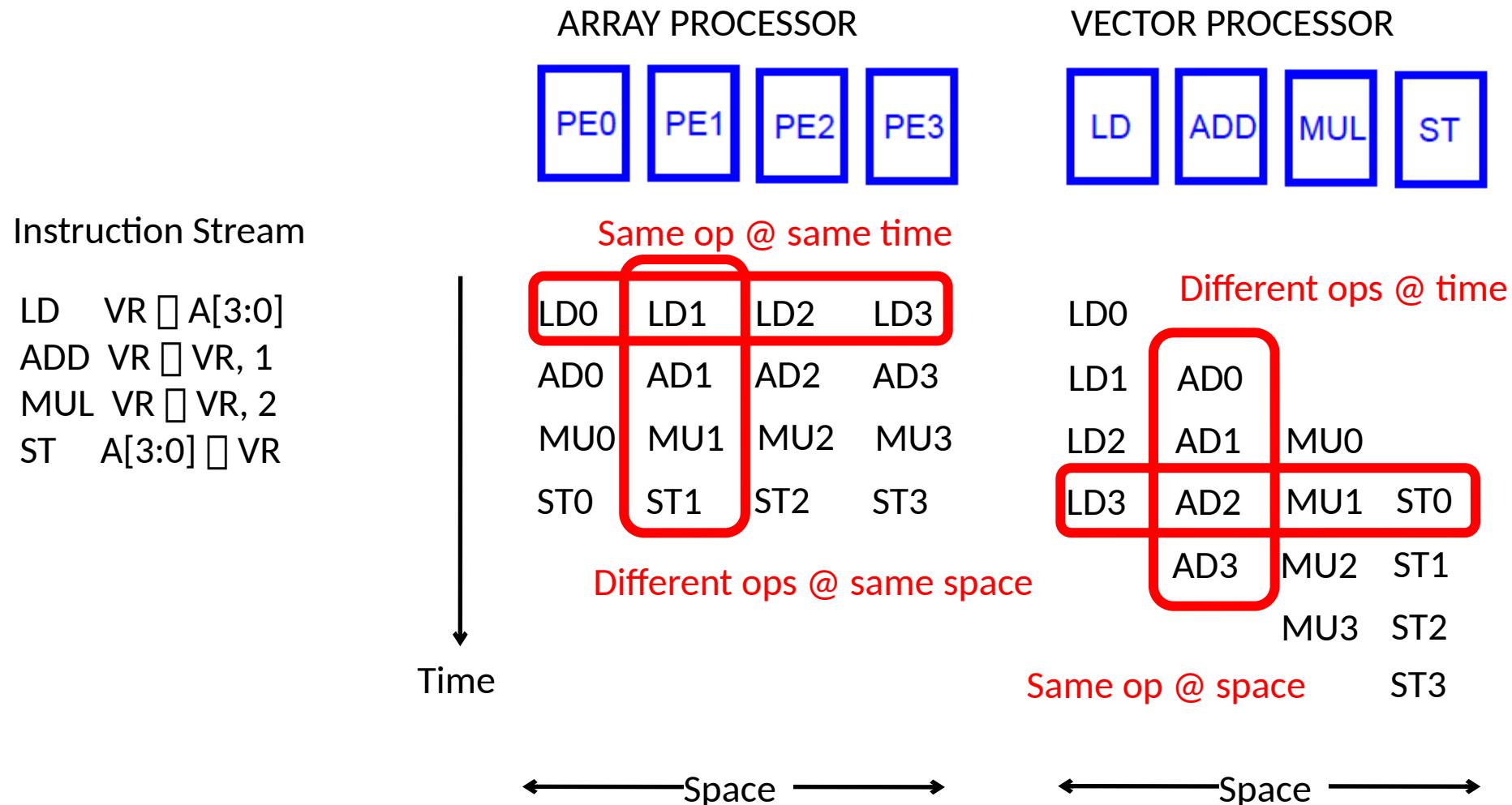
# SIMD PROCESSING

- Concurrency arises from performing the **same operations on different pieces of data**
  - Single instruction multiple data (SIMD)
  - E.g., dot product of two vectors
- Contrast with thread (“control”) parallelism
  - Concurrency arises from executing different threads of control in parallel
- Contrast with data flow
  - Concurrency arises from executing different operations in parallel (in a data driven manner)
- SIMD exploits instruction-level parallelism
  - Multiple instructions concurrent: instructions happen to be the same

# SIMD PROCESSING

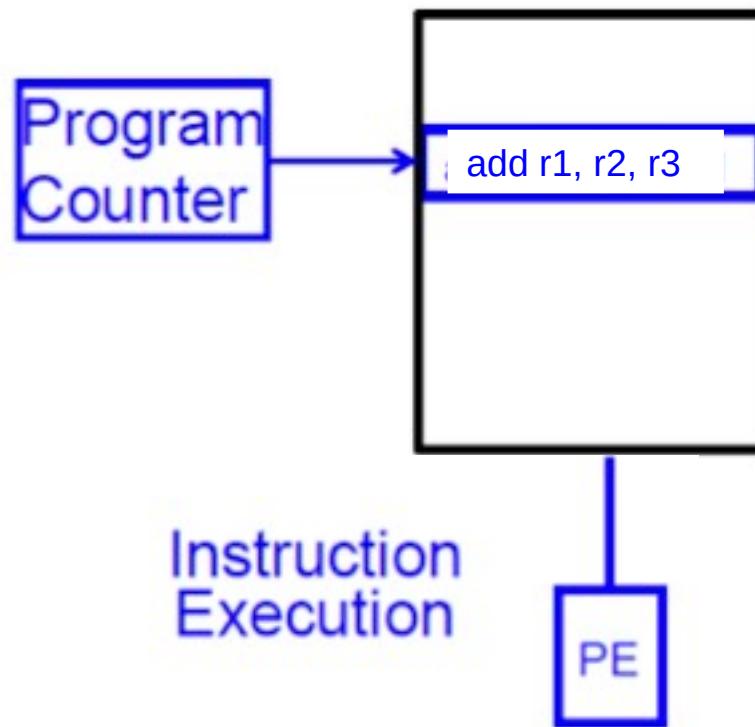
- Single instruction operates on multiple data elements
  - In time or in space
- Multiple processing elements
- Time-space duality
  - **Array processor**: Instruction operates on multiple data elements at the same time
  - **Vector processor**: Instruction operates on multiple data elements in consecutive time steps

# ARRAY VS. VECTOR PROCESSORS



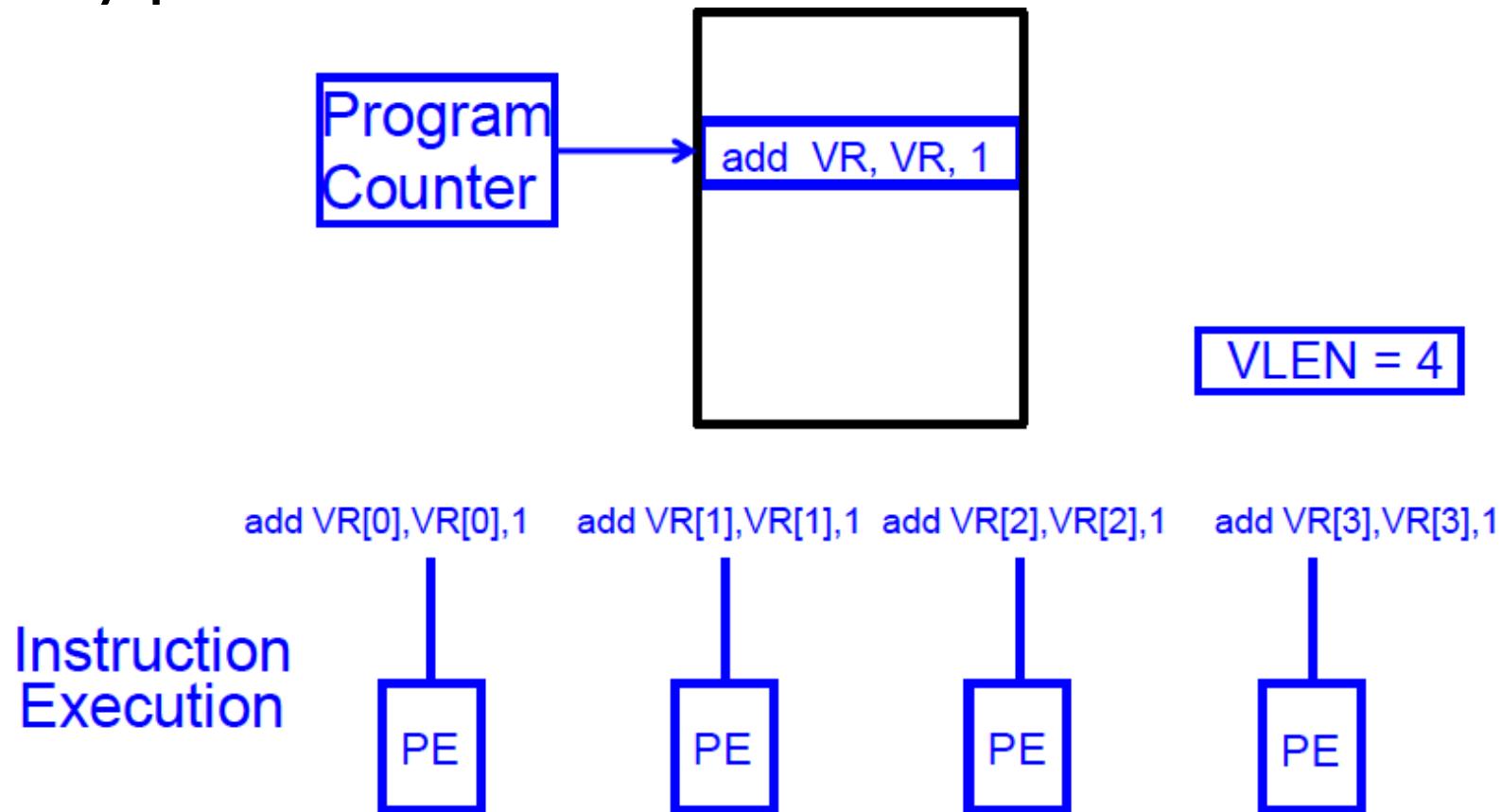
# SCALAR PROCESSING

- Conventional form of processing (von Neumann model)



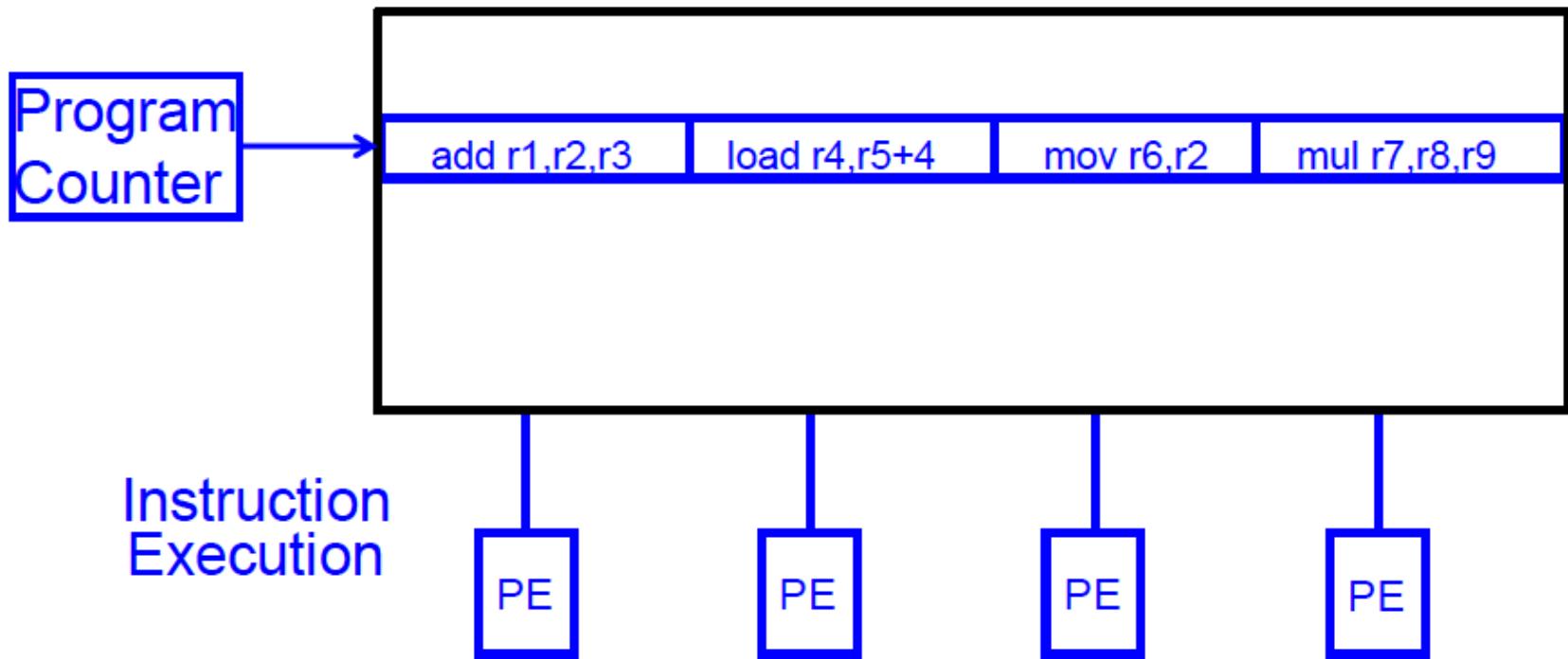
# SIMD ARRAY PROCESSING

- Array processor



# VLIW PROCESSING

- Very Long Instruction Word
  - We will get back to this later



# VECTOR PROCESSORS

- A vector is a one-dimensional array of numbers
- Many scientific/commercial programs use vectors

```
for (i = 0; i<=49; i++)
```

$$C[i] = (A[i] + B[i]) / 2$$

- A vector processor is one whose instructions operate on vectors rather than scalar (single data) values
- Basic requirements
  - Need to load/store vectors  **vector registers** (contain vectors)
  - Need to operate on vectors of different lengths  **vector length register (VLEN)**
  - Elements of a vector might be stored apart from each other in memory  **vector stride register (VSTR)**
    - Stride: distance between two elements of a vector

# VECTOR PROCESSOR ADVANTAGES

## + No dependencies within a vector

- Pipelining, parallelization work well
- Can have very deep pipelines, no dependencies!

## + Each instruction generates a lot of work

- Reduces instruction fetch bandwidth

## + Highly regular memory access pattern

- Interleaving multiple banks for higher memory bandwidth
- Prefetching

## + No need to explicitly code loops

- Fewer branches in the instruction sequence

# Static DataFlow

- Allows only one instance of a node to be enabled for firing
- A dataflow node is fired only when all of the tokens are available on its input arcs and no tokens exist on any of its output arcs

# Dynamic Dataflow

- Allocate instruction templates, i.e., a frame, dynamically to support each loop iteration and procedure call
  - termination detection needed to deallocate frames
- The code can be shared if we separate the code and the operand storage

# Static versus Dynamic Dataflow Machines

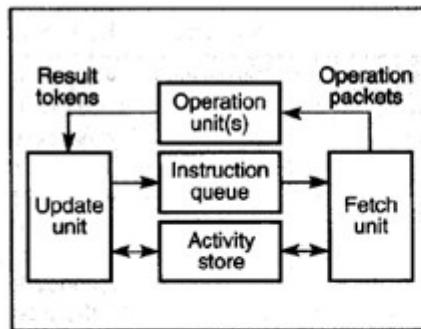


Figure 1. The basic organization of the static dataflow model.

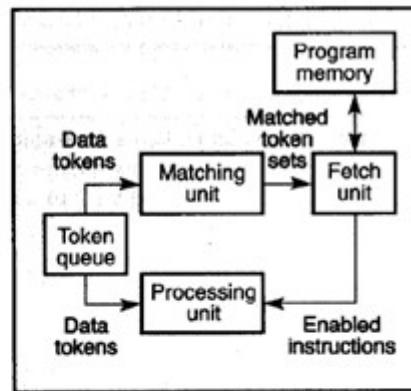


Figure 3. The general organization of the dynamic dataflow model.

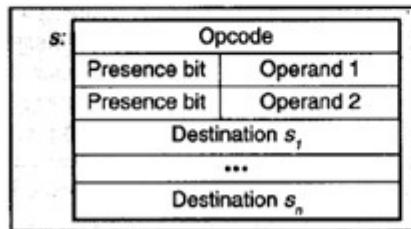


Figure 2. An instruction template for the static dataflow model.

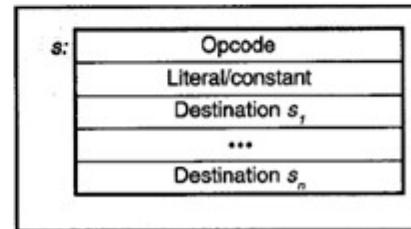


Figure 4. An instruction format for the dynamic dataflow model.

# Pure Dataflow Machines, Explicit Token Store Machines and hybrid and Unified Architectures

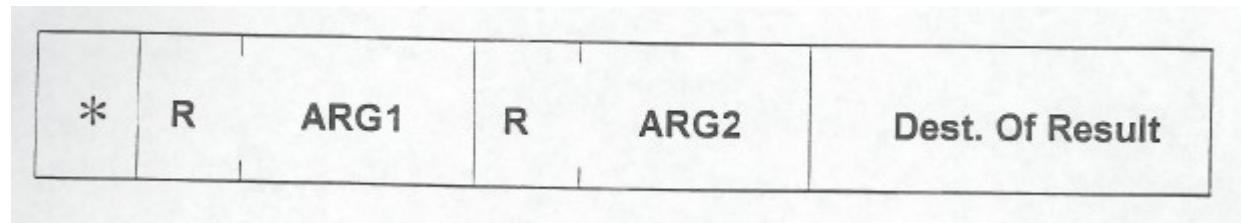
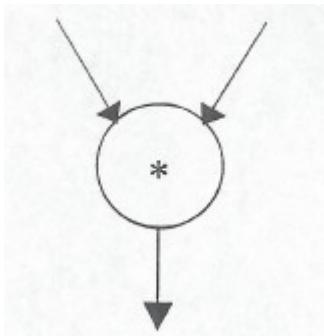
YADHU KRISHNAN P D

S7 CSE

ROLL NO : 60

# Pure Dataflow Machines

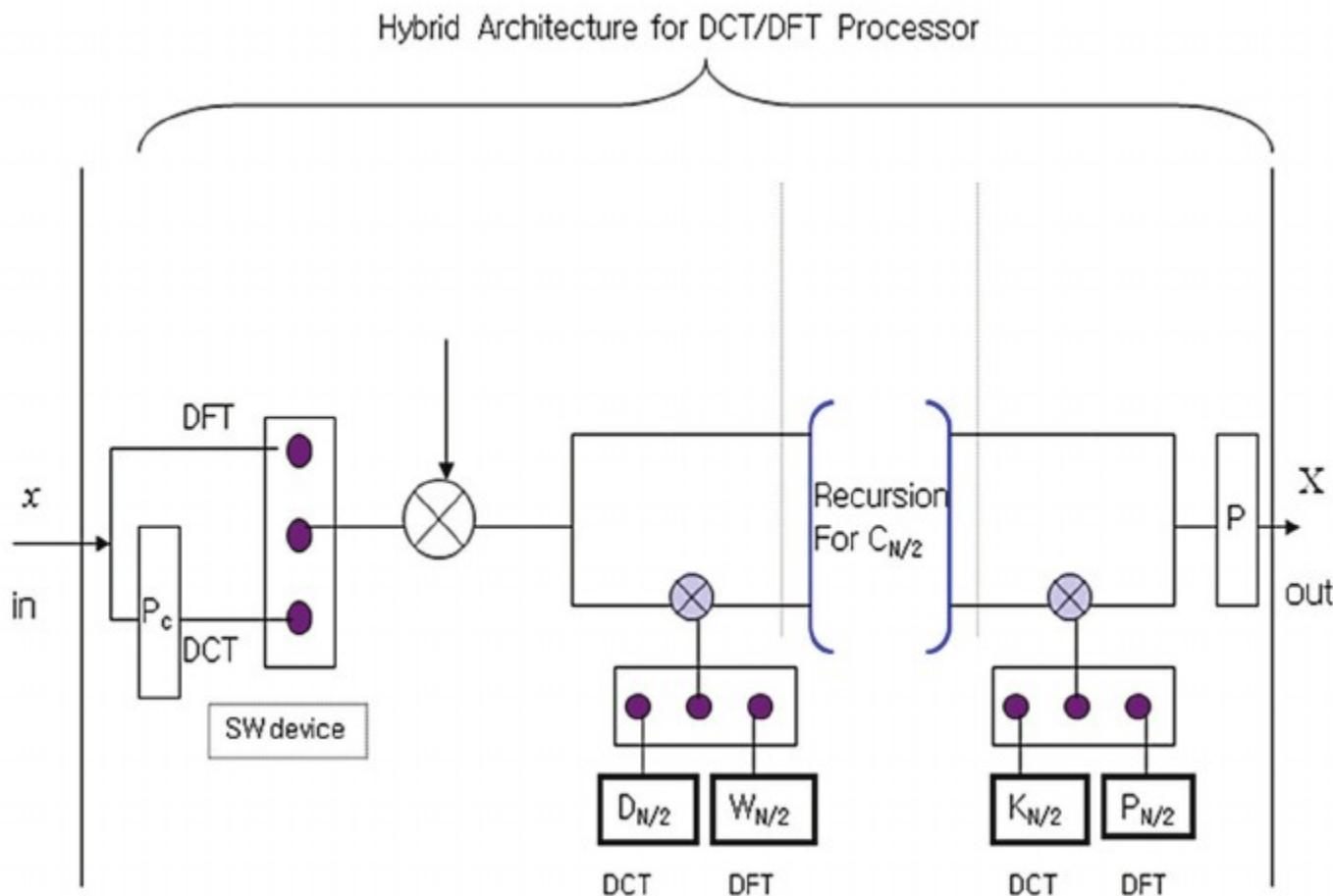
- In a data flow machine, a program consists of data flow nodes
- A data flow node fires (fetched and executed) when all its inputs are ready,i.e. when all inputs have tokens.
- Data flow node and its ISA representation:



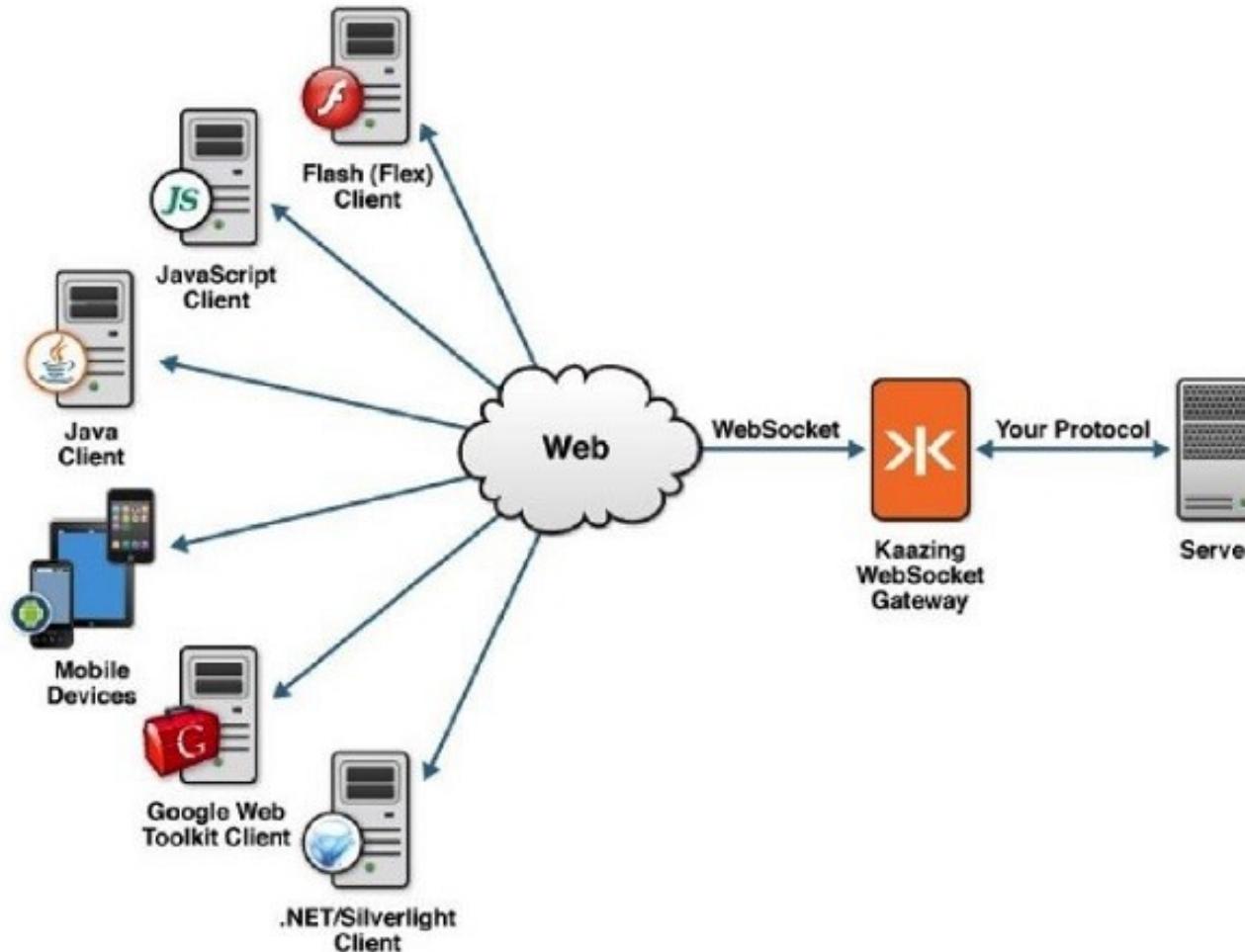
# Explicit Token Store Machines

- A greatly simplified approach to data-flow execution, called the explicit token store (ETS) architecture, and its current realization in Monsoon are presented.
- The essence of dynamic data-flow execution is captured by a simple transition on state bits associated with storage local to a processor .
- The Explicit Token Store (ETS) architecture is an unusually simple model of dynamic dataflow execution that is realized in Monsoon, a large-scale dataflow multiprocessor.
- A Monsoon processor prototype is operational at the MIT Laboratory for Computer Science, running large programs compiled from the dataflow language Id.

# Hybrid Architecture



# Unified Architecture



# Multithreading - Architecture Environment and Computations

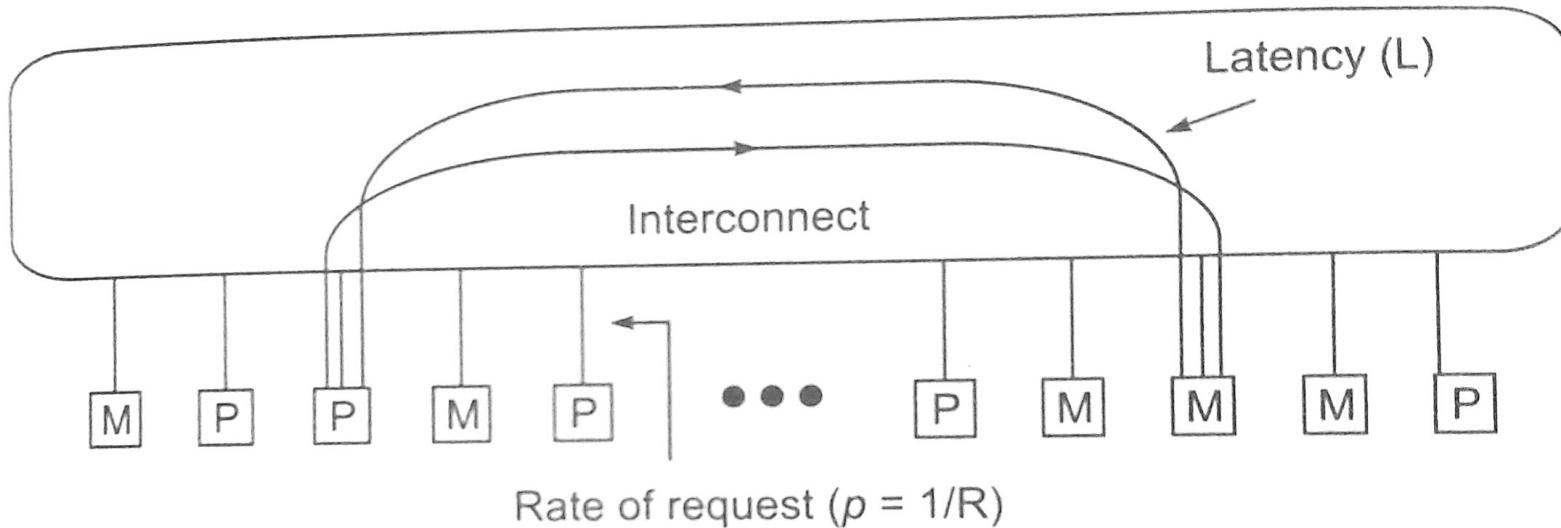
Submitted By

Manu John  
CSE S7, 41

# Architecture Environment

One of the possible multithreaded MPP system is modeled by network of processor(p) and memory(p). The distributed memories form a global address space. Four machine parameter are defined below

- The Latency (L): Communication latency on a remote memory access. The value of L includes the network delay, cache-miss penalty, and delays caused by contentions in split transaction
- The number of threads (N): Number of process that can be interleaved in each processor
- The context-switching overhead (C ): The cycles lost performing context switching in a processor
- The interval between switches (R ): The cycle between switches triggered by remote reference. The inverse  $p=1/R$  the rate of requests



(a) The architecture environment. (Courtesy of Rafael Saavedra, 1992)

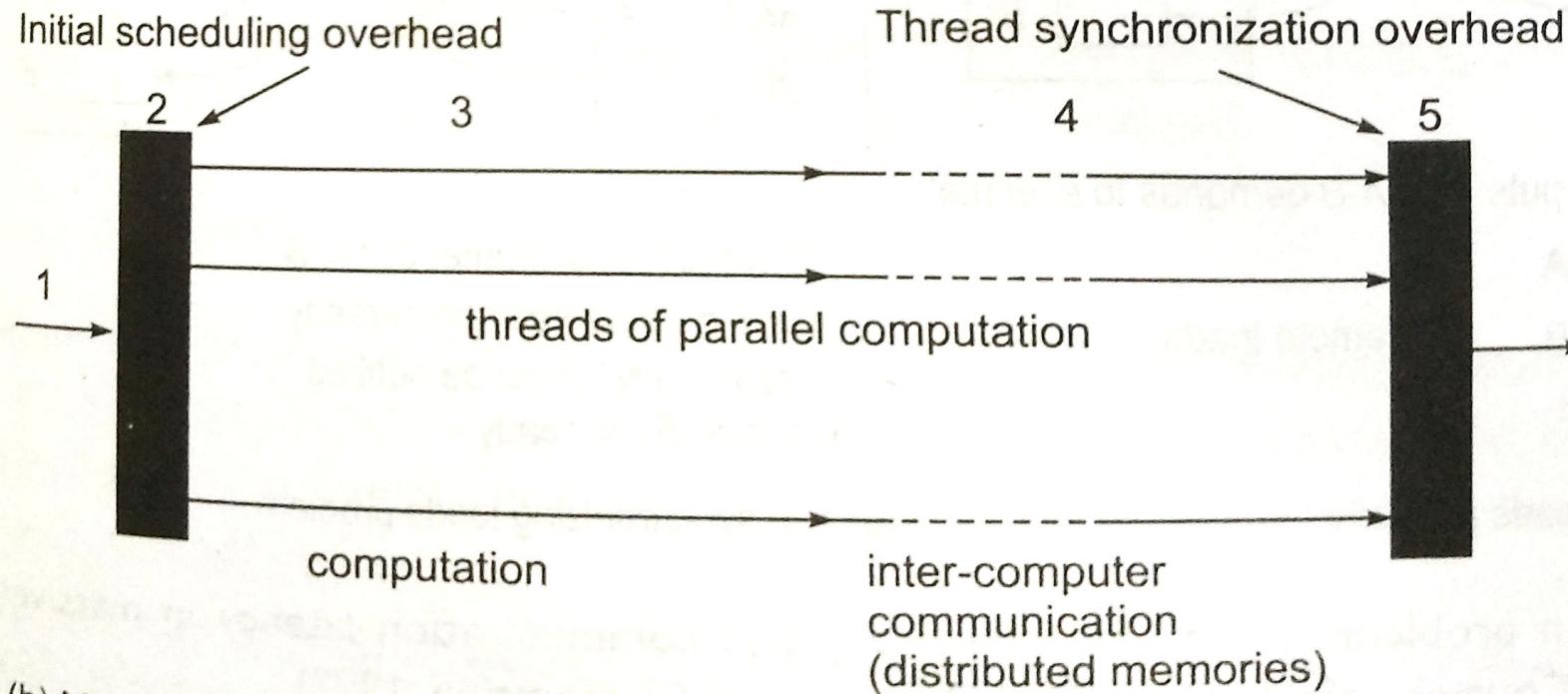
In order to increase efficiency,

- One approach is to reduce the rate of requests by using distributed coherent caches
- Eliminate processor waiting through multithreading

# Multithreaded Computations

- The computation model starts with a sequential thread
- Followed by supervisory scheduling
- Where the processors begin threads of computation
- Intercomputer messages that update variable among the nodes when the computer has a distributed memory
- Finally by synchronization prior to beginning the next unit of parallel work

# Multithreaded Computations



---

# **Relaxed Memory Consistency - Processor and —Release Consistency—**

---

Liya Yohannan,S7 CS

# Relaxed Memory Models

- Relaxed memory models do not make programming any more complex(many would disagree however).However, you need to follow additional rules about how to synchronize threads.C11, Pthreads and Java are specified for relaxed memory models.
- In relaxed memory models, both the program order of memory references and the write atomicity requirements are removed and are replaced with different rules.
- For compiler writers and computer architects this means that more optimizations are permitted.
- For programmers it means two things:
  - you must protect data with special system recognized synchronization primitives, e.g. locks, instead of normal variables used as flags.
  - your code will almost certainly become faster, perhaps by between 10 and 20 percent, due to eliminating the write access penalty.

# Relaxed Memory Models

- Relaxed memory models relax the two constraints of memory accesses: program order and write atomicity.
- We have the following possibilities of relaxing SC.
- Relaxing A to B program order: we permit execution of B before A.
  - write to read program order to different addresses
  - write to write program order to different addresses
  - read to write program order to different addresses
  - read to read program order to different addresses
  - read other CPU's write early
  - read own write early
- Different relaxed memory models permit different subsets of these.

# Assumptions

- All writes will eventually be visible to all CPUs.
- All writes are serialized (recall: this can be done at the memory by letting one write be handled at a time — other pending writes must be retried).
- Uniprocessor data and control dependences are enforced.

# Relaxing the Write to Read Program Order Constraint

- A read may be executed before a preceding write has completed.

## Additionally Relaxing the Write to Write Program Order Constraint

- Consider the program below

```
int a, f;

// called by T1          // called by T2
void v(void)              void w(void)
{
    a = u();            while (!f)
    f = 1;             ;
}                           printf("a = %d\n", a);
                           }
```

- By relaxing the write to write program order constraint, the write to f may be executed by T 1 even before the function call to u, resulting in somewhat unexpected output.

- Machines with relaxed memory models have special machine instructions for synchronization.
- Consider a machine with a sync instruction with the following semantics:
- When executed, all memory access instructions issued before the sync must complete before the sync may complete.
- All memory access instructions issued after the sync must wait (i.e. not execute) until the sync has completed.
- The memory consistency model introduced with the sync instruction is called **Weak Ordering**

# Release Consistency

- Release Consistency, RC, is an extension to WO, and was invented for the Stanford DASH research project.
- Two different synchronization operations are identified.
  - An acquire at a lock.
  - A release at an unlock.
- An acquire orders all subsequent memory accesses, i.e. no read or write is allowed to execute before the acquire. Neither the compiler nor the hardware may move the access to before the acquire, and all acknowledged invalidations that have arrived before the acquire must be applied to the cache before the acquire can complete (i.e. leave the pipeline).
- A release orders all previous memory accesses. The processor must wait for all reads to have been performed (i.e. the write which produced the value read must have been acknowledged by all CPUs) and all writes made by itself must be acknowledged before the release can complete.

# WO vs RC

- Recall: by  $A \rightarrow B$  we mean B may execute before A
- WO relaxation: data  $\rightarrow$  data
- RC relaxation:
  - data  $\rightarrow$  data
  - data  $\rightarrow$  acquire
  - release  $\rightarrow$  data
  - release  $\rightarrow$  acquire
  - acquire  $\rightarrow$  data: forbidden
  - data  $\rightarrow$  release: forbidden

---

# **Stanford Dash Multiprocessor - Data Memory Hierarchy and Directory Protocol**

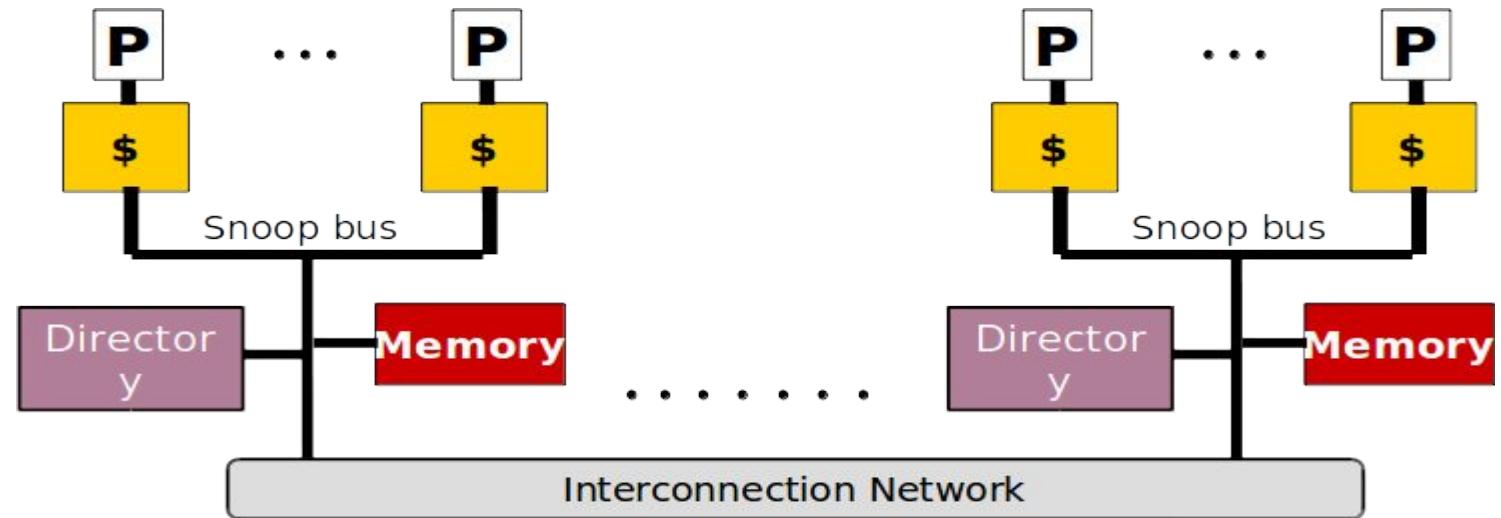
Submitted by ,

Vimal P Viswan  
S7 CSE

---

## Stanford DASH multiprocessor

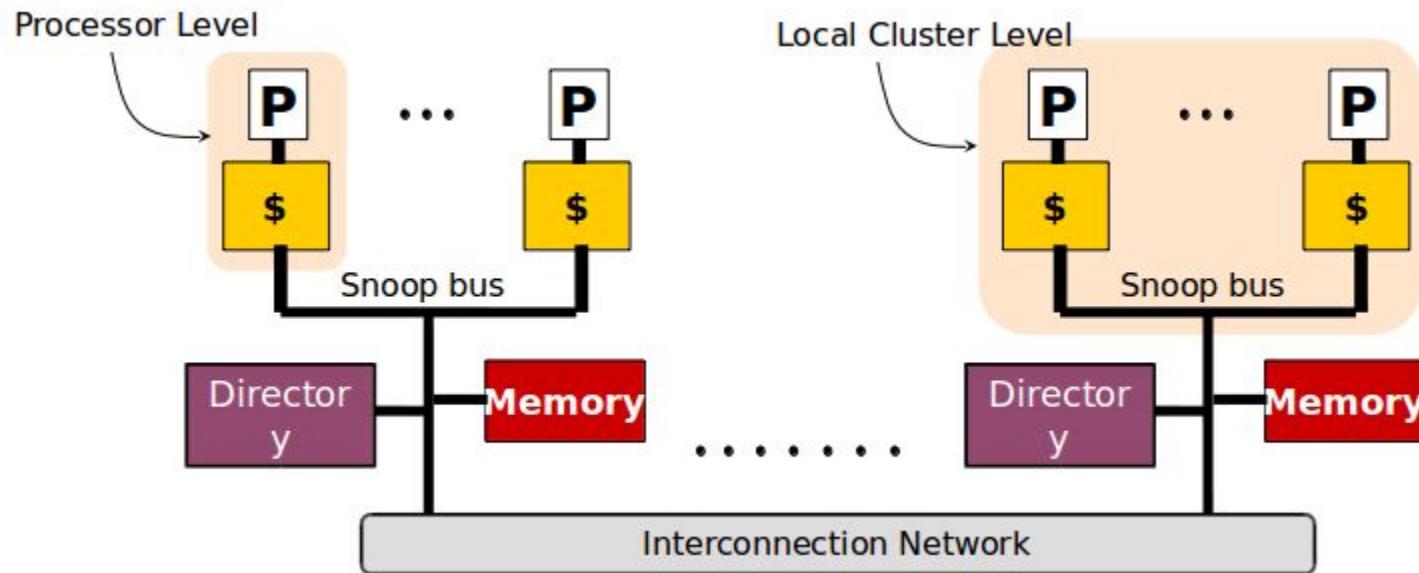
- The Computer Systems Laboratory at Stanford University is developing a shared-memory multiprocessor called Dash(an abbreviation for Directory Architecture for Shared Memory)
- .The fundamental premise behind the architecture is that it is possible to build a scalable high-performance machine with a single address space and coherent caches.



## • Stanford DASH

- Stanford DASH: 4 CPUs in each cluster, total 16 clusters (1992)
  - Invalidation-based cache coherence
  - Directory keeps one of the 3 status of a cache block at its home node
    - Uncached
    - Shared (unmodified state)
    - Dirty

# Data Memory Hierarchy



---

# Hierarchy

- Processor Level
- Local Cluster Level
- Home Cluster Level (address is at home)
  - If dirty, needs to get it from remote node which owns it
- Remote Cluster Level

# Directory Coherence Protocol

