1) Illustrate snoopy protocols with its approaches.

Snoopy Protocols and Their Approaches

In multiprocessor systems, maintaining cache coherence is crucial to ensure that multiple caches hold consistent data. One common approach to maintaining cache coherence is using **Snoopy Protocols**. These protocols rely on the ability of each cache to "snoop" on the communication between the other caches and the main memory. Here's a detailed illustration of Snoopy Protocols and their approaches:

Basic Concept

- **Snoopy Protocols**: These protocols ensure that all caches that have a copy of a memory block maintain a coherent view of the memory. Each cache controller monitors (or "snoops" on) the bus to determine if it has a copy of a block that is requested or modified by another processor.

Types of Snoopy Protocols

1. **Write-Invalidate Protocol**
2. **Write-Update Protocol**

Write-Invalidate Protocol

- **Mechanism**: When a processor writes to a block, it invalidates all other copies of that block in other caches.
- **Process**:
  - When a processor wants to write to a cache block, it sends an invalidation signal to other caches.
  - Other caches snoop this signal and invalidate their copies of the block.
  - The writing processor can then update its local copy.
- **Example**:
  - Assume Processor P1 writes to a memory location X.
  - P1 sends an invalidation signal for X.
  - If Processor P2 has X in its cache, it invalidates X.
  - P1 updates X in its cache.
  - Any subsequent read or write by other processors to X will cause them to fetch the updated value from P1 or main memory.
- **Advantage**: Ensures that only one copy of the modified data exists, simplifying consistency.

- **Disadvantage**: Can lead to high invalidation traffic, especially if data is frequently shared and written by multiple processors.

Write-Update Protocol

- **Mechanism**: When a processor writes to a block, it updates all other copies of that block in other caches.
- **Process**:
  - When a processor wants to write to a cache block, it sends an update signal along with the new data to other caches.
  - Other caches snoop this signal and update their copies of the block with the new data.
- **Example**:
  - Assume Processor P1 writes to a memory location X.
  - P1 sends an update signal for X along with the new value.
  - If Processor P2 has X in its cache, it updates X with the new value.
  - All caches now have the updated value of X.
- **Advantage**: Reduces invalidation traffic and can be more efficient if data is frequently read after being written.
- **Disadvantage**: Can lead to high update traffic, which may be inefficient if updates are frequent and the data is not often read by other processors.

2) With neat diagram explain the bus system at board level ,backplane and I/O level

1. Board Level Bus System

At the board level, the bus system is integral to the operation of the motherboard, connecting the CPU, memory, and various peripherals.

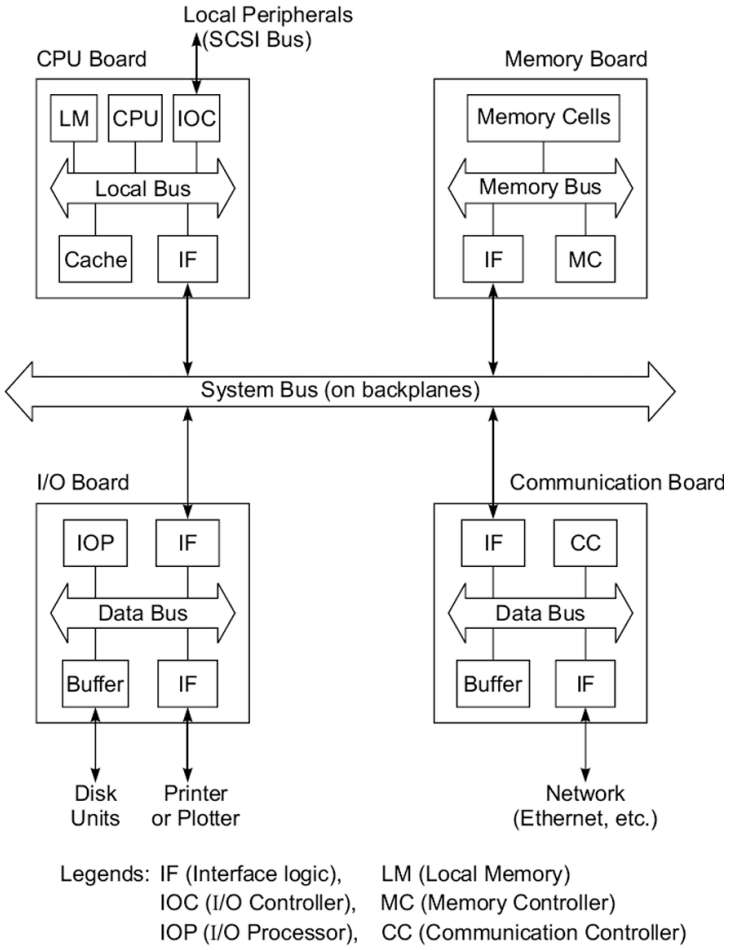
- **Address Bus**: Carries the addresses of data (but not the data itself) between the CPU and memory.
- **Data Bus**: Transports actual data between the CPU, memory, and peripherals.
- **Control Bus**: Carries control signals from the CPU to other components to coordinate various operations.

Components Connected by the Bus System:

- **CPU (Central Processing Unit)**: The main processor that performs all computations and processing tasks.

- **Memory (RAM, ROM, Cache)**: Temporary storage for data and instructions that the CPU needs to execute tasks.
- **Peripherals (e.g., USB ports, SATA interfaces)**: External devices connected to the motherboard for additional functionality.

The bus system at this level ensures that the CPU can fetch instructions and data from memory and send/receive data to and from peripherals efficiently.



2. Backplane Bus System

The backplane bus system is used in systems that require multiple circuit boards to be connected. This is common in servers and high-end workstations where modularity and expandability are critical.

- **Backplane**: A large board that contains multiple slots for connecting various boards (like CPU boards, memory boards, I/O boards). It acts as the backbone of the system.
- **Connections**: The backplane provides the physical and electrical connections between the boards, allowing them to communicate and share power.

Functionality:

- **Modularity**: Allows different boards to be easily added or replaced, facilitating upgrades and maintenance.
- **Scalability**: Supports the addition of multiple CPUs, memory modules, and I/O interfaces, making it suitable for high-performance applications.

The backplane bus system is essential for creating flexible and expandable computer systems, particularly in environments that require robust and scalable computing power.

3. I/O Level Bus System

At the I/O level, the bus system manages communication between the central processing unit (CPU) and various input/output devices.

- **I/O Bus**: The pathways through which data is transferred between the CPU and peripheral devices. Common I/O buses include PCI (Peripheral Component Interconnect) and USB (Universal Serial Bus).
- **I/O Controllers**: Dedicated hardware that manages data transfer between the CPU and peripheral devices. Examples include disk controllers, network interface controllers, and graphics controllers.

Peripheral Devices:

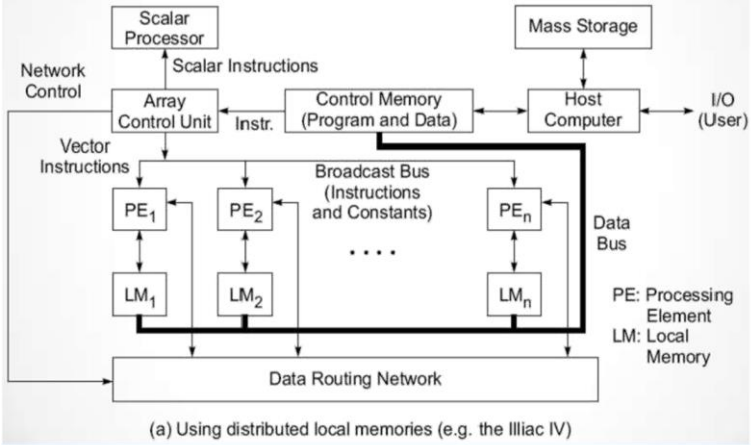
- **Storage Devices (e.g., HDDs, SSDs)**: Store data persistently and communicate with the CPU via I/O buses.
- **Input Devices (e.g., Keyboard, Mouse)**: Send user inputs to the CPU.
- **Output Devices (e.g., Monitors, Printers)**: Receive data from the CPU to provide output to the user.
- **Network Interfaces (e.g., Ethernet, Wi-Fi Adapters)**: Enable communication over a network.

3) With a neat diagram, describe the implementation models of SIMD.

SIMD (Single Instruction, Multiple Data) models can be distinguished based on their memory distribution and addressing schemes. Two primary models for constructing SIMD supercomputers are the Distributed Memory Model and the Shared Memory Model. Here is a detailed explanation of these models:

1. Distributed Memory Model

In the Distributed Memory Model, each Processing Element (PE) has its own local memory. This model is known for its spatial parallelism, where each PE operates on its own data independently.



(a) Using distributed local memories (e.g. the Illiac IV)

Key Features:

- **Spatial Parallelism**: Exploits parallelism by distributing tasks across multiple PEs, each with its own memory.
- **Array of PEs**: The system consists of an array of PEs, each supplied with local memory, all controlled by an array control unit.
- **Control Unit**: The control unit manages the overall operation. It loads programs and data into the control memory through the host computer and distributes them to the PEs' local memories.
- **Instruction Decoding**:
  - **Scalar Operations**: If the instruction is scalar or involves program control, it is executed by a scalar processor attached to the control unit.
  - **Vector Operations**: If the instruction is a vector operation, it is broadcast to all PEs for parallel execution.
- **Data Distribution**: Partitioned datasets are distributed to the PEs' local memories via a vector data bus.
- **Interconnection Network**: PEs are interconnected by a data routing network, which performs inter-PE data communications like shifting, permutation, and other routing operations.

Advantages:

- Scalable, as each PE operates independently with its local memory.
- Reduces memory access contention since each PE has its dedicated memory.

Disadvantages:

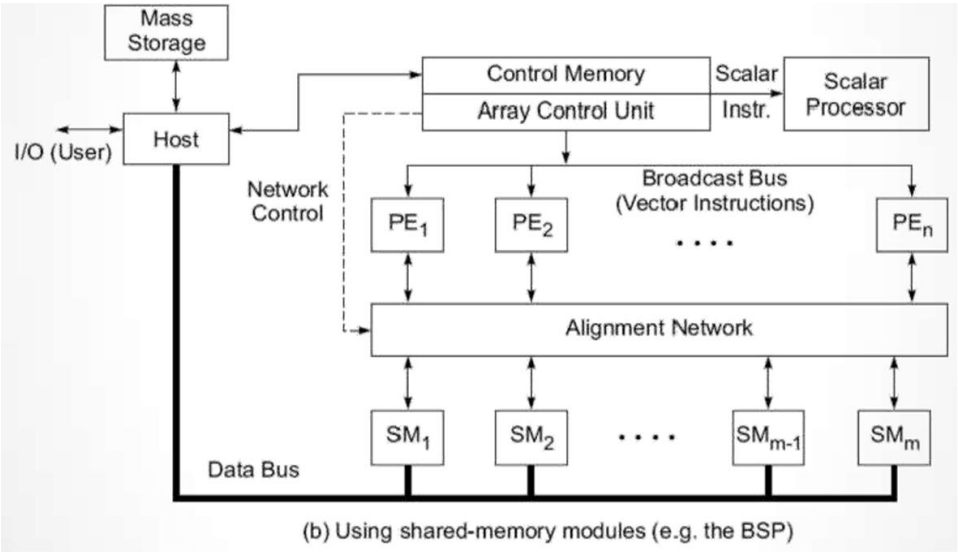
- Communication overhead can be high due to the need for explicit data transfer between PEs.
- Complexity in programming and managing data distribution.

Example Systems:

- **MasPar MP-1 and MP-2**: Massively parallel SIMD supercomputers with distributed memory architecture.

2. Shared Memory Model

In the Shared Memory Model, all PEs share a common memory space, which allows for more straightforward data access but requires careful management to avoid conflicts.



(b) Using shared-memory modules (e.g. the BSP)

Key Features:

- **Shared Memory**: All PEs access a common shared memory space, controlled by a single control unit.
- **Alignment Network**: An inter-PE memory communication network, called the alignment network, is used to manage memory access. This network ensures that data is correctly aligned and prevents access conflicts.
- **Control Unit**: The control unit coordinates the alignment network and manages the distribution of instructions and data to the PEs.
- **Data Access**: Proper setting of the alignment network is crucial to avoid access conflicts and ensure efficient data transfer among PEs.

Advantages:

- Simplifies the programming model, as all PEs can directly access shared data.
- Eliminates the need for explicit data distribution among PEs.

Disadvantages:

- Memory access conflicts can occur, requiring sophisticated management techniques.

4) Discuss different vector access memory schemes.

Or Explain C-access and S-access organization for m way interleaved memory

Vector processing involves accessing multiple data elements simultaneously, and different memory schemes are used to optimize this access. Here are three primary vector access memory organization schemes: C-Access, S-Access, and C/S-Access.

1. C-Access Memory Organization

The C-Access memory organization involves interleaving memory to allow concurrent and overlapping access to multiple words. This scheme is designed to maximize memory throughput by staggering access cycles.

Key Features:

- **Low-Order Memory Structure**: Allows mmm words to be accessed concurrently and overlapped.
- **Memory Modules**: The low-order bits of the address select the memory modules, and the high-order bits select the word within each module.
- **Addressing**: m=2am = 2^am=2a, and a+b=na + b = na+b=n (total address length).

Operation:

- **Stride of 1**: Successive addresses are latched in the address buffer at the rate of one per cycle. It takes mmm minor cycles to fetch mmm words, equating to one major memory cycle.
- **Stride of 2**: Successive accesses must be separated by two minor cycles to avoid conflicts, reducing memory throughput by half.
- **Stride of 3**: No module conflict, yielding maximum throughput.
- **General Case**: Maximum throughput (mmm words per cycle) is achieved if the stride is relatively prime to mmm.

Advantages:

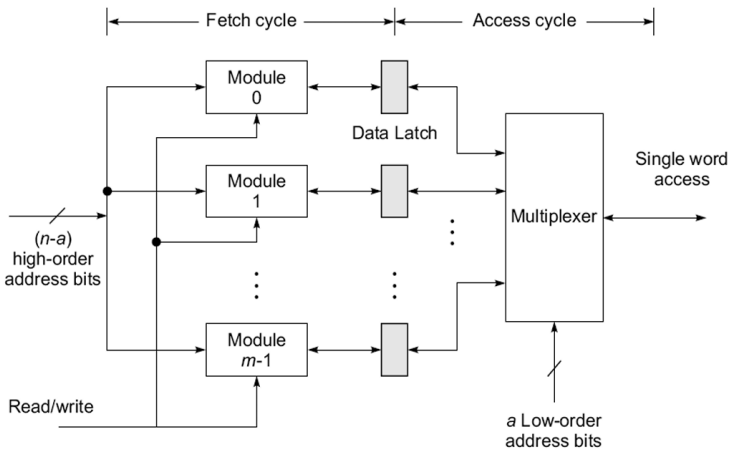
- **High Throughput**: Optimized for strides that are relatively prime to the number of memory modules.
- **Concurrent Access**: Enables concurrent access to multiple words.

Disadvantages:

- **Throughput Variability**: Throughput decreases significantly for non-optimal strides.

2. S-Access Memory Organization

The S-Access memory organization synchronizes the access to all memory modules, allowing simultaneous access to words across modules.



(a) S-access organization for an *m*-way interleaved memory

Key Features:

- **Synchronized Access**: All memory modules are accessed simultaneously.
- **High-Order Address Bits**: The high-order bits select the same offset word from each module.

Operation:

- **Memory Cycle**: At the end of each memory cycle, m=2am = 2^am=2a consecutive words are latched.
- **Stride Greater Than 1**: Throughput decreases roughly in proportion to the stride.

Advantages:

- **Simplicity**: Easy to implement and manage.
- **Concurrent Access**: All modules are accessed at the same time, allowing simultaneous data fetching.

Disadvantages:

- **Throughput Reduction**: Throughput decreases with increased stride, which can limit performance.

3. C/S-Access Memory Organization

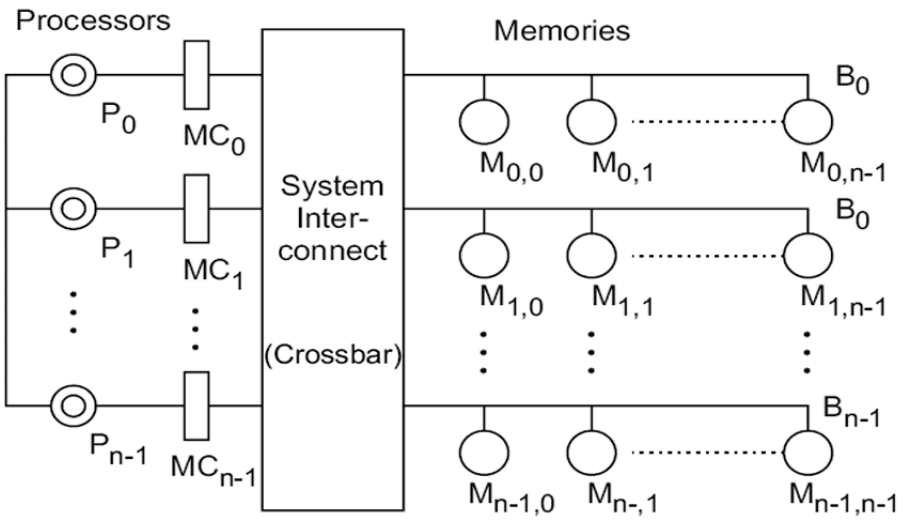
The C/S-Access memory organization combines the features of C-Access and S-Access schemes, providing high bandwidth and parallel pipelined access.

Key Features:

- **Combination of Schemes**: Utilizes both C-Access and S-Access methods.
- **Access Buses**: nnn access buses are used with mmm interleaved memory modules attached to each bus.
- **Interleaving**: The mmm modules on each bus are interleaved to allow C-Access.

Operation:

- **Pipelined Memory Accesses**: In each memory cycle, up to m×nm \times nm×n words can be fetched if the buses are fully used.
- **Parallel Access**: Suitable for vector multiprocessor configurations, providing parallel pipelined access.



Advantages:

- **High Bandwidth**: Provides high bandwidth access to vector data sets.
- **Parallel Pipelined Access**: Suitable for high-performance vector processing.

Disadvantages:

- **Complexity**: More complex to implement due to the combination of access methods and multiple buses.

5) Illustrate the processor consistency models.

The processor consistency models are covered under the topic of relaxed memory consistency models. These models define how memory operations (loads and stores) performed by one processor appear to other processors. The consistency models vary in their strictness of how operations are ordered and made visible across processors. Here are the primary models discussed in the provided material:

1. **Sequential Consistency (SC)**:
    - This is the strictest model where the results of execution are as if all operations were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program.
2. **Relaxed Consistency Models**:
    - These models relax one or more requirements of sequential consistency to allow more flexibility in execution and better performance. They do not provide memory consistency at the hardware level; instead, the programmers must ensure consistency using synchronization techniques. The main types of relaxed consistency models include:
    - **Processor Consistency (PC)**:
        - Ensures that writes issued by a single processor are observed by all other processors in the order they were issued, but does not guarantee the ordering of writes from different processors.
    - **Weak Consistency (WC)**:
        - Allows both reads and writes to be reordered, but requires that all memory operations must be completed before any synchronization operations (e.g., locks) can be performed.
    - **Release Consistency (RC)**:
        - Further refines weak consistency by distinguishing between acquire and release operations. An acquire operation (e.g., acquiring a lock) ensures that all previous writes are visible before it is performed, while a release operation (e.g., releasing a lock) ensures that all previous writes are visible to other processors after it is performed.
    - **Eventual Consistency**:
        - This model ensures that if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value. It is often used in distributed systems and web services where high availability is preferred over immediate consistency.

6) Describe Coherence problems in data sharing and process migration.

Coherence Problems in Data Sharing

Data Sharing:

- **Writable Data Sharing:** When multiple processors share writable data, coherence problems can occur. Each processor might cache a copy of the shared data, leading to inconsistencies if one processor updates its copy without informing the others.
- **Cache Coherence:** In a multiprocessor system, each processor may have its private cache. If one processor updates a shared data item in its cache, other processors may still have the old value in their caches. This leads to a situation where different caches have different values for the same data, causing incoherence.
- **Write Through vs. Write Back Policies:**
    - **Write Through:** When a processor writes data to its cache, it also writes the same data to the main memory. This approach can still cause inconsistencies if the other processors' caches are not updated simultaneously.
    - **Write Back:** The processor updates only its cache, and the main memory is updated later. This approach can lead to inconsistencies if the data in the cache is not written back to the main memory before another processor reads it.

Example Scenario:

- Consider a multiprocessor system with two processors, P1 and P2, both sharing the main memory. Let X be a shared data element.
    - **Initial State:** Both processors have consistent copies of X.
    - **Write Through Policy:** If P1 writes a new value X' to its cache, the main memory is immediately updated, but P2's cache may still hold the old value X.
    - **Write Back Policy:** If P1 writes X' to its cache, the main memory and P2's cache remain with the old value X until P1's cache line is written back to the main memory.

Coherence Problems in Process Migration

Process Migration:

- When a process containing a shared variable migrates from one processor to another, the coherence of the variable's data can be affected.

- **Write Back Cache:** If a process migrates from processor 1 to processor 2, the data modified by the process may reside in the cache of processor 1 and not be updated in the main memory. Thus, processor 2 may read stale data from the main memory.
- **Write Through Cache:** Even with write-through caches, inconsistencies can occur if the cache is not properly invalidated or updated during the migration process.

Example Scenario:

- A process with a shared variable X migrates from processor 1 to processor 2.
    - **Write Back Cache:** Processor 1 has updated X to X' in its cache. If the process migrates to processor 2, it may read the old value X from the main memory since X' has not been written back yet.
    - **Write Through Cache:** If the process migrates, the caches need to be carefully managed to ensure consistency, as direct updates to the main memory may still leave other caches with stale data.

7) What is Cache Coherence Problem? What are the different causes of cache inconsistencies?

Cache Coherence Problem refers to the issue in multiprocessor systems where multiple processors cache copies of the same memory location. When one processor updates its cache, the other processors may still have stale data, leading to inconsistencies across the system.

Causes of Cache Inconsistencies

1. **Simultaneous Writes:**
    - When two or more processors attempt to write different values to the same memory location simultaneously, it leads to inconsistencies as each processor's cache may reflect a different value.
2. **Write-Back Caches:**
    - In a write-back cache system, the processor updates its cache and postpones writing the updated value to the main memory. Other processors may read stale data from the main memory if they do not have the latest cached value.
3. **Write-Through Caches:**
    - Even with write-through caches, where updates are immediately written to the main memory, inconsistencies can occur if other processors' caches are not updated simultaneously. The updated value in the main memory might not be propagated to all caches promptly.
4. **Cache Invalidation Delays:**

- When a processor updates a cached value, it should ideally invalidate or update the corresponding entries in other caches. Delays or failures in this invalidation process can result in other processors using outdated data.

5. **Non-Synchronized Access:**
   - If multiple processors access and modify shared data without proper synchronization mechanisms (like locks or semaphores), it can lead to race conditions, where the final value of the shared data depends on the sequence of operations, causing inconsistencies.

6. **Memory Reordering:**
   - Modern processors often reorder memory operations to optimize performance. This can lead to situations where the order of reads and writes seen by different processors is inconsistent, resulting in stale or inconsistent data being accessed.

8) Explain Context Switching Policies

Context switching is essential for multitasking in operating systems, enabling a single processor to manage multiple processes by switching between them. Different multithreaded architectures employ various context-switching policies, which are crucial for maximizing processor efficiency and minimizing idle time. Here are four primary context-switching policies:

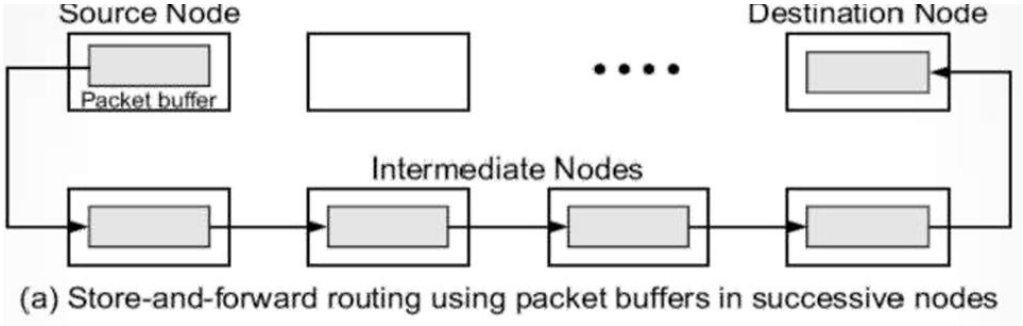
1. **Switch on Cache Miss:**
   - **Policy:** The context is preempted when it causes a cache miss.
   - **Details:** The processor switches contexts only when it is certain that the current one will be delayed for a significant number of cycles due to a cache miss. The average interval between misses (R) and the time required to satisfy the miss (L) are key factors.
2. **Switch on Every Load:**
   - **Policy:** The context switches on every load, regardless of whether it causes a miss.
   - **Details:** This policy allows switching on every load, independent of a cache miss. Here, the average interval between loads (R) is considered. Blocking occurs only if the load results in a cache miss.
3. **Switch on Every Instruction:**
   - **Policy:** The context switches on every instruction, regardless of its type.

- **Details:** This policy makes successive instructions independent, which benefits pipelined execution. It ensures that the processor frequently switches contexts, aiming to utilize pipeline stages efficiently.

4. **Switch on Block of Instructions:**
   - **Policy:** Blocks of instructions from different threads are interleaved.
   - **Details:** Interleaving blocks of instructions improves the cache-hit ratio due to locality and benefits single-context performance by making better use of cache memory n

9) Explain Store and forward routing wormhole routing related to message routing

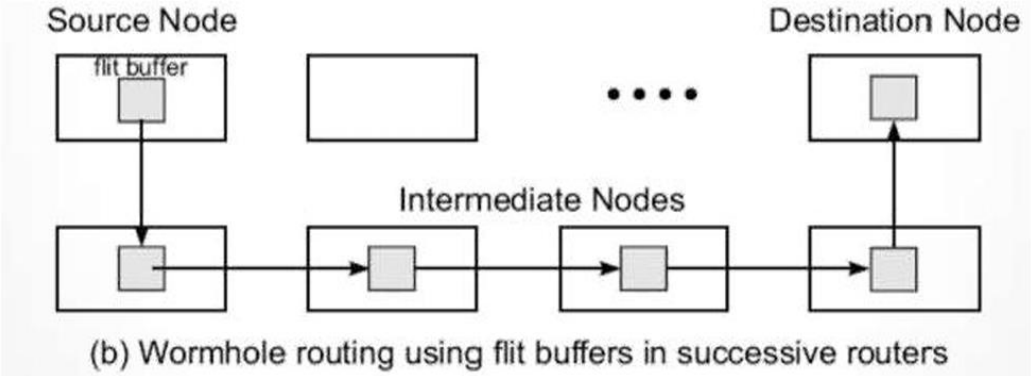
Store and Forward Routing:

- **Mechanism**:
  - In store and forward routing, packets are the basic unit of information flow.
  - Each node in the network uses a packet buffer.
  - A packet is transmitted from a source node to a destination node through a sequence of intermediate nodes.
  - When a packet reaches an intermediate node, it is stored in the buffer first.
  - The packet is forwarded to the next node only if the desired output channel and a packet buffer in the receiving node are both available.
  - 

(a) Store-and-forward routing using packet buffers in successive nodes
- **Advantages**:
  - Reliable as it ensures that packets are not lost during transmission.
  - Simplifies the routing process since each node handles entire packets.
- **Disadvantages**:
  - High latency due to the need to store and forward packets at each intermediate node.
  - Requires large buffer sizes to store packets at each node.

Wormhole Routing:

- **Mechanism**:
  - Packets are subdivided into smaller units called flits (flow control digits).
  - Flit buffers are used in the hardware routers attached to nodes.
  - Transmission from the source node to the destination node is done through a sequence of routers.
  - Only the header flit knows the destination, and all data flits must follow the header flit.
  - Flits from different packets cannot be mixed; otherwise, they may be routed to the wrong destination.
  - Pipelined transmission: flits move asynchronously using a handshaking protocol.



(b) Wormhole routing using flit buffers in successive routers

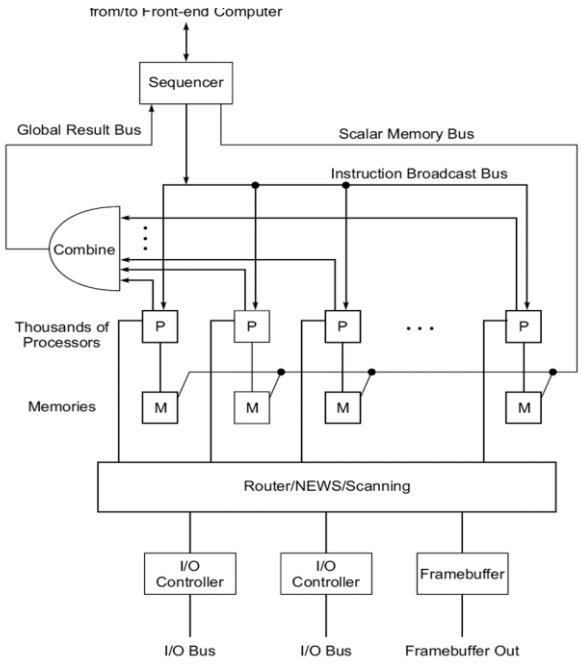
- **Advantages**:
  - Efficient as it reduces the need for large buffers.
  - Faster transmission due to reduced latency.
  - Suitable for high-performance computing environments.
- **Disadvantages**:
  - Complex hardware requirements for managing flit buffers.
  - Potential for deadlock if virtual channels are not managed properly.

Comparison:

- **Latency**:
  - Store and Forward Routing has a latency directly proportional to the distance between the source and the destination.
  - Wormhole Routing has a latency almost independent of the distance, primarily dependent on the packet length and the flit length.
- **Buffer Requirements**:
  - Store and Forward Routing requires larger buffers at each node to store entire packets.
  - Wormhole Routing requires smaller flit buffers but needs sophisticated management to handle flit movement.
- **Performance**:

- Wormhole Routing generally offers better performance due to reduced latency and more efficient use of bandwidth .

10) Explain with diagram CM-2 Architecture



Program Execution Paradigm

- **Execution Start**: All programs begin execution on a front-end.
- **Microinstructions**: The front-end issues microinstructions to the back-end processing array for data-parallel operations.
- **Sequencer**: These microinstructions are broken down by a sequencer and broadcast to all data processors in the array(Module-4 ACA).

Data Exchange

Data sets and results can be exchanged between the front-end and the processing array in three ways:

1. **Broadcasting**: Carried out through a broadcast bus to all data processors at once.
2. **Global Combining**: Allows the front-end to obtain the sum, largest value, logical OR, etc., from values from each processor.
3. **Scalar Memory Bus**: Allows the front-end to read or write one 32-bit value at a time from or to the memories attached to the data processors(Module-4 ACA).

Processing Array

- **Configuration**: The processing array contains from 4K to 64K bit-slice data processors (PEs).
- **Control**: All PEs are controlled by a sequencer(Module-4 ACA).

Processing Nodes

- **Components**: Each data processing node contains 32 bit-slice data processors, an optional floating-point accelerator, and interfaces for inter-processor communication(Module-4 ACA).

Hypercube Routers

- **Structure**: The router nodes on all processor chips are wired together to form a Boolean n-cube.
- **Configuration**: A full configuration of CM-2 has 4096 router nodes on processor chips interconnected as a 12-dimensional hypercube(Module-4 ACA).

Major Applications

The CM-2 has been applied in various MPP and grand challenge applications, including:

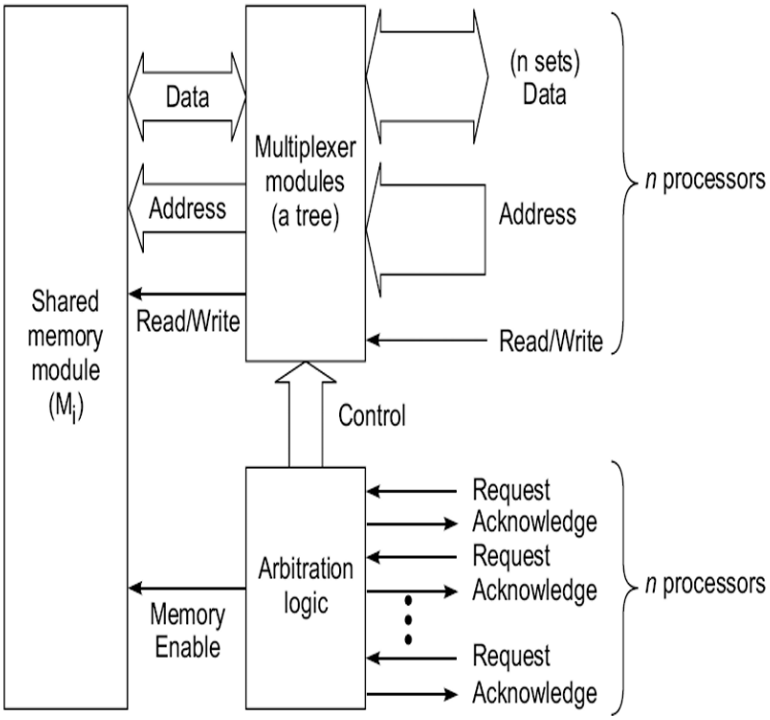
- Document retrieval using relevance feedback
- Memory-based reasoning (e.g., medical diagnostic system called QUACK)
- Bulk processing of natural languages
- SPICE-like VLSI circuit analysis and layout
- Computational fluid dynamics

11) Explain Crossbar network with diagram?

Crossbar Switch:

1. **Definition:**
    - A crossbar switch is a type of single-stage network.
    - It is designed to connect every input to every output through a crosspoint switch.
2. **Structure:**
    - The crossbar switch uses a grid (or mesh) of switches where each switch connects a processor to a memory module.
    - Each column in the n * m crossbar mesh contains n crosspoint switches.
3. **Functionality:**
    - Only one switch in each column can be connected at a time.
    - The design must handle potential contention for each memory module.
    - It avoids competition for bandwidth by using $O(N^2)$ switches to connect N inputs to N outputs.
4. **Scalability:**
    - Crossbar switches are highly non-scalable due to their complexity and the number of switches required.
    - They are typically used to connect a small number of workstations, generally 20 or fewer.
5. **Connection Process:**

- Each processor provides a request line, a read/write line, a set of address lines, and a set of data lines to the crosspoint switch.
- The crosspoint switch responds with an acknowledgment once the access is completed.



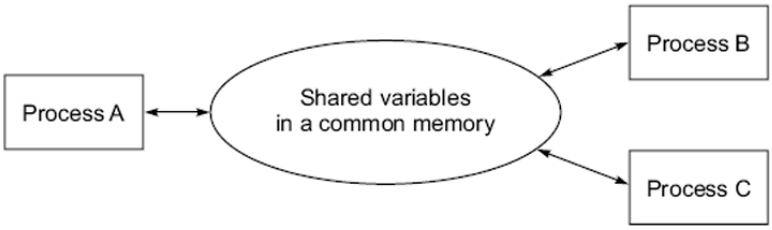
1) Define parallel programming models. Discuss any two models.

Parallel programming models provide a simplified and transparent view of the computer hardware and software system, designed specifically for multiprocessors, multicomputers, or vector/SIMD computers. These models abstract the complexity of the underlying architecture, making it easier to develop parallel programs.

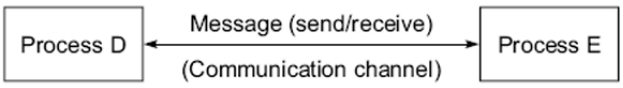
Shared-Variable Model

In the shared-variable model, multiple processes share a common address space and communicate by reading and writing shared variables. This model is commonly used in multiprocessor programming.

- **Characteristics**:
  - **Active and Passive Resources**: Processors are active resources, while memory and I/O devices are passive resources.
  - **Inter-Process Communication (IPC)**: Parallelism depends on how IPC is implemented. The process address space is shared among processes.
  - **Critical Section**: A critical section (CS) is a code segment that accesses shared variables and must be executed by only one process at a time.
    - **Mutual Exclusion**: Ensures that at most one process is executing the CS at a time.
    - **No Deadlock**: Ensures that there is no circular wait among processes.
    - **No Preemption**: Ensures that a process in CS is not interrupted until it finishes.
    - **Eventual Entry**: Ensures that each process gets a chance to enter the CS eventually.
  - **Protected Access**: The granularity of CS affects performance. Too large a CS limits parallelism, while too small a CS adds unnecessary complexity.



(a) IPC using shared variable



(b) IPC using message passing

- **Operational Modes**:
  - **Multiprogramming**: Multiple programs are executed concurrently.

- **Multiprocessing**: Multiple processors execute multiple programs simultaneously.
- **Multitasking**: Multiple tasks (processes) are executed concurrently.
- **Multithreading**: Multiple threads within a process are executed concurrently.

Message-Passing Model

The message-passing model is used primarily in multicomputer systems where each processor has its own local memory. Processes communicate by passing messages.

- **Characteristics**:
  - **Communication**: Processes residing on different processor nodes communicate by passing messages through a direct network. Messages can be instructions, data, synchronization signals, or interrupt signals.
  - **Synchronous Message Passing**:
    - No shared memory or mutual exclusion.
    - Sender and receiver processes must synchronize, similar to a telephone call.
    - No buffer is used; one process must block if the other is not ready.
  - **Asynchronous Message Passing**:
    - No synchronization in time and space is required.
    - Arbitrary communication delays may occur.
    - Similar to a postal service using mailboxes; no synchronization between senders and receivers.

Data-Parallel Model

The data-parallel model is used in SIMD computers where parallelism is achieved through hardware synchronization and flow control. This model is particularly useful for applications with regular data structures, like arrays.

- **Characteristics**:
  - **Data Parallelism**: This technique is used in array processors (SIMD).
  - **Array Language Extensions**: Various data-parallel languages are used to represent high-level data types.
  - **Example Languages**: CFD for Illiac 4, DAP Fortran for Distributed Array Processor, C* for Connection Machine.
  - **Challenge**: Matching the problem size with the machine size.

Object-Oriented Model

The object-oriented model allows objects to be dynamically created and manipulated, with processing performed by sending and receiving messages among objects. It leverages the concepts of abstraction and reusability.

- **Concurrent Object-Oriented Programming (COOP)**:
  - **Concurrent Manipulation**: Objects encapsulate data and operations and are manipulated concurrently.
  - **Actor Model**: A framework for COOP where actors (independent components) communicate via asynchronous message passing.
    - **Primitives**: Create, send-to, and become.
  - **Patterns for Parallelism**: Pipeline concurrency, divide and conquer, cooperative problem solving.

Functional and Logic Model

This model includes functional programming languages and logic programming languages, which are used for different types of parallelism.

- **Functional Programming**:
  - Examples: Lisp, Sisal, Strand 88.
  - Characteristics: No side effects, no storage, assignment, or branching; single assignment and data flow language.
- **Logic Programming**:
  - Examples: Concurrent Prolog, Parlog.
  - Characteristics: Implicit search strategy, support for parallel execution, and parallel reduction techniques; used in artificial intelligence for knowledge processing from large databases.

2) With a neat diagram, illustrate different phases of parallelizing compiler.

A parallelizing compiler transforms sequential code into parallel code, optimizing it for execution on parallel architectures. The main phases of a parallelizing compiler can be illustrated as follows:

1. **Source Code**: The starting point, consisting of the original sequential code written by the programmer.
2. **Flow Analysis**: This phase involves analyzing the data and control dependencies within the code.
   - **Data Dependence**: Determines how data dependencies between instructions affect parallelization.

- **Control Dependence**: Analyzes the order of execution of instructions.
- **Reuse Analysis**: Identifies opportunities to reuse data to improve performance.

3. **Program Optimizations**: Optimizes the code for better parallel execution.
   - **Vectorization**: Converts scalar operations to vector operations to exploit data-level parallelism.
   - **Parallelizations**: Transforms code to run concurrently on multiple processors.
   - **Locality**: Optimizes data access patterns to enhance cache performance.
   - **Pipelining**: Arranges computations to overlap execution phases, improving throughput.
4. **Parallel Code Generation**: Generates parallel code tailored to the target architecture.
   - **Granularity**: Determines the size of tasks to balance load and minimize overhead.
   - **Degree of Parallelism**: Establishes how many tasks can be executed in parallel.
   - **Code Scheduling**: Arranges the execution order of tasks to optimize performance.
5. **Target Architecture Specific Optimization**:
   - **Superscalar Processor**: Involves scheduling, register allocation, and context switching.
   - **Shared-Memory Multiprocessor**: Focuses on partitioning, synchronization, and load balancing.
   - **Distributed-Memory Multicomputer**: Handles distributed data and computations, along with message-passing for communication.

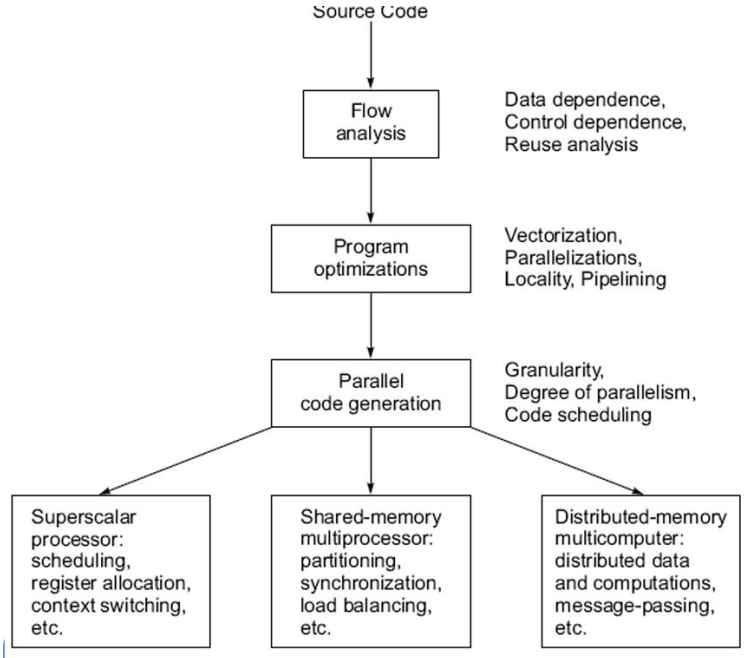


Fig. 10.4 Compilation phases in parallel code generation

3) Describe testing algorithm for dependence testing.

Dependence testing is crucial in parallel computing to identify dependencies among instructions or data elements that could potentially hinder parallel execution. The following steps outline a typical algorithm for dependence testing:

1. **Identify Dependencies:**
   - **Data Dependence (RAW, WAW, WAR):** Determine Read-After-Write (RAW), Write-After-Write (WAW), and Write-After-Read (WAR) dependencies among instructions.
   - **Control Dependence:** Check if the execution of instructions depends on control structures like branches.
   - **Resource Dependence:** Ensure the availability of resources needed for instruction execution.
2. **Flow Analysis:**
   - **Instruction Level:** Analyze the flow of instructions to identify dependencies at the VLSI or superscalar processor level.
   - **Loop Level:** Analyze dependencies within loops, especially for SIMD and systolic arrays.
   - **Task Level:** Evaluate dependencies among tasks in multiprocessor or multicomputer environments.
3. **Optimization Phases:**
   - **Local Optimization:** Optimize code within a basic block.
   - **Global Optimization:** Optimize across basic blocks considering the entire program flow.
4. **Register Renaming:**
   - **Avoiding WAW and WAR:** Use register renaming to avoid write-after-write and write-after-read dependencies. For example, if two instructions write to the same register, assign a new register to one of the instructions to prevent the WAW dependency.
5. **Operand Forwarding:**
   - **Speeding Up Execution:** Implement operand forwarding to reduce the impact of RAW dependencies. This technique allows the result of an operation to be used by subsequent instructions without writing it back to a register first.
6. **Control Dependencies Handling:**
   - **Branch Prediction:** Use accurate branch prediction techniques to minimize the performance impact of control dependencies. Mis-predicted branches should trigger a pipeline flush and reorder buffer clear.
7. **Dependence Graphs:**
   - **Graph Construction:** Construct dependence graphs to visualize and analyze dependencies among instructions. This aids in identifying and resolving conflicts.

8. **Parallel Code Generation:**
    - **Compiler Directives:** Utilize compiler directives to generate parallel code and optimize performance. Tools like Parafase and PFC can transform sequential programs into parallel ones, perform inter-procedure flow analysis, and generate vector code.

4) Illustrate the dynamic scheduling of a pipeline using Tomasulo's algorithm.

Tomasulo's algorithm is a technique used to dynamically schedule instructions to improve the utilization of the pipeline and minimize delays caused by various hazards (data hazards, structural hazards, etc.). It helps in achieving out-of-order execution while maintaining correct program execution.

Key Components

1. **Reservation Stations (RS):**
    - Temporary storage locations associated with each functional unit (like ALUs, multipliers, etc.).
    - Hold the instruction until it is ready to execute.
2. **Register Renaming:**
    - Uses a set of temporary registers to avoid conflicts between instructions that might use the same registers (eliminates WAR and WAW hazards).
3. **Common Data Bus (CDB):**
    - A bus that broadcasts the results from the functional units to the reservation stations and registers.
    - Ensures that once an operand is ready, it is available to all the instructions waiting for it.

Steps in Tomasulo's Algorithm

1. **Instruction Issue:**
    - Instructions are issued to reservation stations if there is a free station.
    - If the operands are available, they are fetched and stored in the reservation station.
    - If operands are not available, the reservation station records which functional unit will produce the operand (using tags).
2. **Execution:**
    - Once all operands for an instruction in a reservation station are available, the instruction is sent to the functional unit for execution.

- The reservation station keeps track of which operands are ready and which are not.

3. **Write-back:**
   - Once the functional unit completes the execution, it writes the result on the Common Data Bus (CDB).
   - All reservation stations and registers waiting for this result capture it from the CDB.
   - This allows the waiting instructions to proceed with execution.

Example

Consider the following instructions:

1. `I1: ADD R1, R2 -> R3`
2. `I2: MUL R3, R4 -> R5`
3. `I3: SUB R6, R3 -> R7`

Here, `I2` and `I3` depend on the result of `I1` because they need `R3`.

Execution Flow:

1. **Issue Stage:**
   - `I1` is issued to an ADD reservation station.
   - `I2` is issued to a MUL reservation station but is marked as waiting for `R3` (tagged as `I1`).
   - `I3` is issued to a SUB reservation station but is also marked as waiting for `R3` (tagged as `I1`).
2. **Operand Forwarding:**
   - When `I1` completes, it broadcasts the result of `R3` on the CDB.
   - Reservation stations holding `I2` and `I3` capture the value of `R3` from the CDB and update their status to "ready."
3. **Execution:**
   - `I2` now has all operands and can execute.
   - `I3` can execute after `I2` or in parallel if there are no resource conflicts.
4. **Write-back:**
   - Results of `I2` and `I3` are written back to their respective destination registers.

5) Explain Object Oriented Programming Model and its characteristics

Object Oriented Programming (OOP) Model is a programming paradigm that uses "objects" to design applications and computer programs. Objects are instances of classes, which can hold both data and methods. The characteristics of the OOP model include:

1. **Encapsulation**: This is the mechanism of hiding the internal state of an object and requiring all interaction to be performed through an object's methods. It helps in protecting the integrity of the data within the object.
2. **Abstraction**: Abstraction means dealing with the level of complexity by hiding the unnecessary details from the user. It allows focusing on the interactions at a higher level without knowing the internal complexities.
3. **Inheritance**: This feature allows a new class to inherit properties and behavior (methods) from an existing class. It provides a way to reuse code, create a hierarchical relationship between classes, and establish a polymorphic relationship.
4. **Polymorphism**: It refers to the ability of different objects to respond, each in its own way, to identical messages. It allows methods to do different things based on the object it is acting upon, promoting flexibility and integration.

Concurrent OOP: This involves the concurrent manipulation of objects, which are program entities encapsulating data and operations in a single unit. Concurrent OOP emphasizes abstraction and reusability.

Actor Model: It is a framework for concurrent OOP. In this model, "actors" are independent components that communicate via asynchronous message passing. The three primitives involved are:

- **Create**: To create new actors.
- **Send-to**: To send messages to actors.
- **Become**: To change the actor's behavior in response to a message.

Parallelism in COOP:

- **Pipeline Concurrency**: Divides a task into stages, each handled by a different process.
- **Divide and Conquer**: Breaks a problem into sub-problems, solves them independently, and combines the results.
- **Cooperative Problem Solving**: Multiple processes collaborate to solve a problem, sharing their intermediate results.

6) Explain Functional and logical models

Functional Programming Model

- **Characteristics**:
  - **No Side Effects**: Functions in functional programming should not produce side effects, meaning they do not alter any state or interact with outside systems (like modifying a global variable or performing I/O operations).
  - **No Storage, Assignment, or Branching**: There are no concepts of storage locations (variables that can be modified), assignment operations, or branching (conditional statements) as found in imperative programming languages.
  - **Single Assignment and Data Flow**: Variables are assigned exactly once. This aligns with the principles of data flow programming where the output of one operation is directly used as the input to another, promoting immutability and predictability.
- **Examples**: Languages like Lisp, Sisal, and Strand 88 are mentioned as examples of functional programming languages.

Logic Programming Model

- **Characteristics**:
  - **Knowledge Processing**: Logic programming is primarily used for knowledge representation and processing, often in the context of large databases or artificial intelligence.
  - **Implicit Search Strategy**: It supports implicit search strategies where the logic interpreter automatically searches through possible solutions to find one that satisfies the given constraints or rules.
  - **Parallel Execution**: It supports parallel execution through techniques like AND-parallelism (parallel execution of conjunctive goals) and OR-parallelism (parallel execution of alternative solutions).

7) Describe in brief Tomasula algorithm

Tomasulo's algorithm is a hardware algorithm used for dynamic scheduling of instructions to overcome some of the limitations posed by static scheduling. Here is a brief description of the algorithm based on the provided document:

1. **Register Renaming**:
   - Register renaming is an implicit part of Tomasulo's algorithm.
   - It requires a set of program-invisible registers to which programmable registers are re-mapped.
   - These invisible registers are provided with reservation stations of functional units.
2. **Functional Units and Reservation Stations**:

- Functional units are assumed to be internally pipelined and can complete one operation per clock cycle.
- Each functional unit can initiate one operation per clock cycle if the reservation station is ready with the required input operands.
- Reservation stations hold the operation code and operand values (if available) or operand tags (if not available).

3. **Operation Execution**:
   - When the operands become available, the functional unit initiates the required operation.
   - Operand values are forwarded over the common data bus (CDB) along with source tags.
   - These values are copied into all reservation station operand slots with matching tags.
4. **RAW Dependence Handling**:
   - Assume instruction I1 writes its result into R4, and subsequent instructions I2 and I3 read the result.
   - If the result from I1 is not available when I2 and I3 are issued, they are parked in the reservation stations with source tags corresponding to I1's output.
   - When I1's result is available, it is sent over the CDB with the source tag, and all destinations with matching tags copy the data simultaneously.
5. **Dynamic Scheduling**:
   - Instructions are scheduled dynamically, considering inter-instruction dependences and the state of functional units.
   - This maximizes instruction-level parallelism.

Tomasulo's algorithm effectively manages data hazards and enhances instruction-level parallelism by dynamically scheduling instructions and using register renaming to resolve WAR and WAW dependences

8) Explain synchronous message passing and asynchronous passing related to message passing model

Synchronous Message Passing

1. Definition: Synchronous message passing requires that both the sender and receiver processes are synchronized in their communication. This means both processes must be ready for the communication to occur.

2. No Shared Memory: Unlike shared-memory models, synchronous message passing does not rely on shared memory. Instead, all data transfer occurs through message exchanges.

3. No Mutual Exclusion: There is no need for mechanisms to ensure mutual exclusion since there is no shared data that might lead to concurrent access issues.

4. Synchronization: Synchronization is critical in synchronous message passing. The sender process must wait until the receiver process is ready to accept the message, and vice versa. This ensures that messages are only passed when both processes are prepared for communication.

5. Blocking Behavior: A notable characteristic of synchronous message passing is its blocking behavior. If the sender process is ready to send a message but the receiver process is not ready to receive it, the sender process will be blocked (i.e., it will wait) until the receiver is ready. Similarly, the receiver process will wait if it is ready to receive a message but no message has been sent yet.

6. Example: A practical example of synchronous message passing is a telephone call where both parties must be available to communicate in real time. If one party calls and the other is not available, the caller must wait until the other party is ready to talk.

Asynchronous Message Passing

1. Definition: Asynchronous message passing allows the sender and receiver processes to operate independently without needing to synchronize their communication times. Messages can be sent at any time, and the receiver can process them later.

2. No Synchronization Required: There is no need for the sender and receiver to be synchronized in time and space. The sender can send a message without knowing if the receiver is ready, and the receiver can process the message at a later time.

3. Communication Delay: Asynchronous message passing may involve arbitrary communication delays. The sender does not necessarily know when the receiver will process the message, leading to potential delays in acknowledgment.

4. Buffering: Messages are typically stored in a buffer (or mailbox) until the receiver is ready to process them. This allows the sender to continue its execution without waiting for the receiver, enhancing concurrency and reducing blocking.

5. Non-blocking Behavior: In asynchronous message passing, the sender process does not wait for the receiver to be ready. Once the message is sent, the sender can continue its execution. The receiver retrieves the message from the buffer when it is ready to process it.

6. Example: A common example of asynchronous message passing is email communication. The sender can send an email at any time, and the receiver can read and respond to it later. There is no need for both parties to be available simultaneously.

9) Explain With neat diagram Recoder buffer

Structure of a Reorder Buffer

A reorder buffer is organized as an array of entries, each containing the following fields:

1. Instruction Identifier: Identifies the instruction.
2. Value Computed: The result of the instruction.
3. Program-Specified Destination: Where the computed value should be stored.
4. Completion Flag: Indicates whether the instruction has completed execution and its result is available (Module-5 ACA).

Role of the Reorder Buffer

- Maintaining Program Order: While instructions may complete out of order, the reorder buffer ensures they are committed in the correct program order. This means the program and processor states reflect only the results of instructions that have been committed.
- Handling Data Dependences: The reorder buffer helps in resolving data dependences. For example, Read After Write (RAW) dependences can hold up the execution of instructions waiting for their operands. The reorder buffer ensures these dependencies are managed and resolved efficiently.
- Managing Control Dependences: If a branch prediction is incorrect, instructions in the reorder buffer that are part of the mis-predicted branch are flushed to maintain correct program flow.
- Resource Dependences: The buffer allows instructions to wait for the required functional unit to become available, thus enabling out-of-order execution while ensuring resource availability

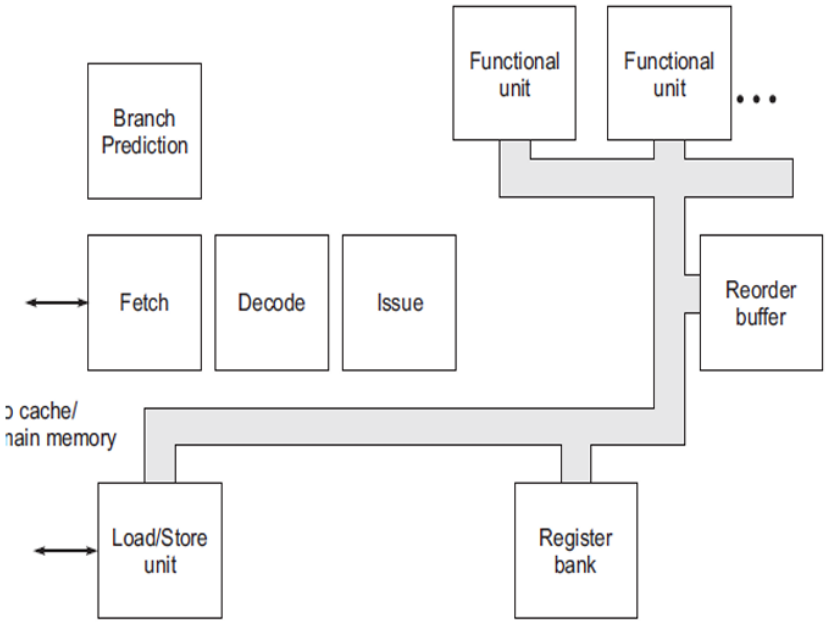


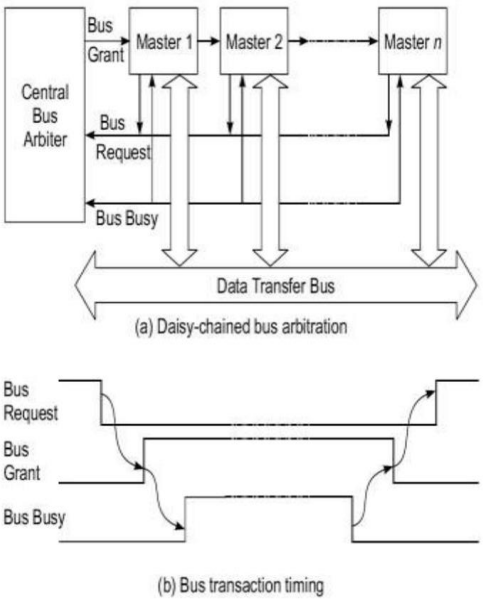
Fig. 12.4 Processor design with reorder buffer

1. Discuss bus arbitration and its types in multiprocessor systems.

Bus arbitration in multiprocessor systems is essential for managing access to the shared communication bus, ensuring orderly communication, and preventing data collisions. The arbitration process selects the next bus master and restricts the tenure of the bus to one master at a time. Competing requests are arbitrated based on fairness or priority, and arbitration competition and bus transactions occur concurrently on a parallel bus over separate lines.

Types of Bus Arbitration

1. **Central Arbitration:** Central arbitration uses a central arbiter and a daisy-chained cascade of potential masters. A special signal line propagates the bus grant from the first master to the last master. All requests share the same bus-request line, and the bus-request signal raises the bus-grant level, which in turn raises the bus-busy level.
    - **Advantages:**
        - Simple scheme
        - Easy to add devices
    - **Disadvantages:**
        - Fixed-priority sequence, which is not fair
        - Slow propagation of the bus-grant signal
        - Not fault-tolerant



g. 5.4 Central bus arbitration using shared requests and daisy-chained bus grants with a fixed priorit

2. **Distributed Arbitration:** In distributed arbitration, each master has its own arbiter and a unique arbitration number. This number is used to resolve arbitration competition. When multiple devices compete for the bus, the winner is the one with the largest arbitration number, determined by

Parallel Contention Arbitration. All potential masters send their arbitration number to shared-bus request/grant (SBRG) lines and compare their own number with the SBRG number. If the SBRG number is greater, the requester is dismissed. The winner's arbitration number remains on the arbitration bus, and after the current bus transaction is completed, the winner seizes control of the bus.

- **Advantages:**
  - Priority-based scheme
- **Disadvantages:**
  - Complex to implement

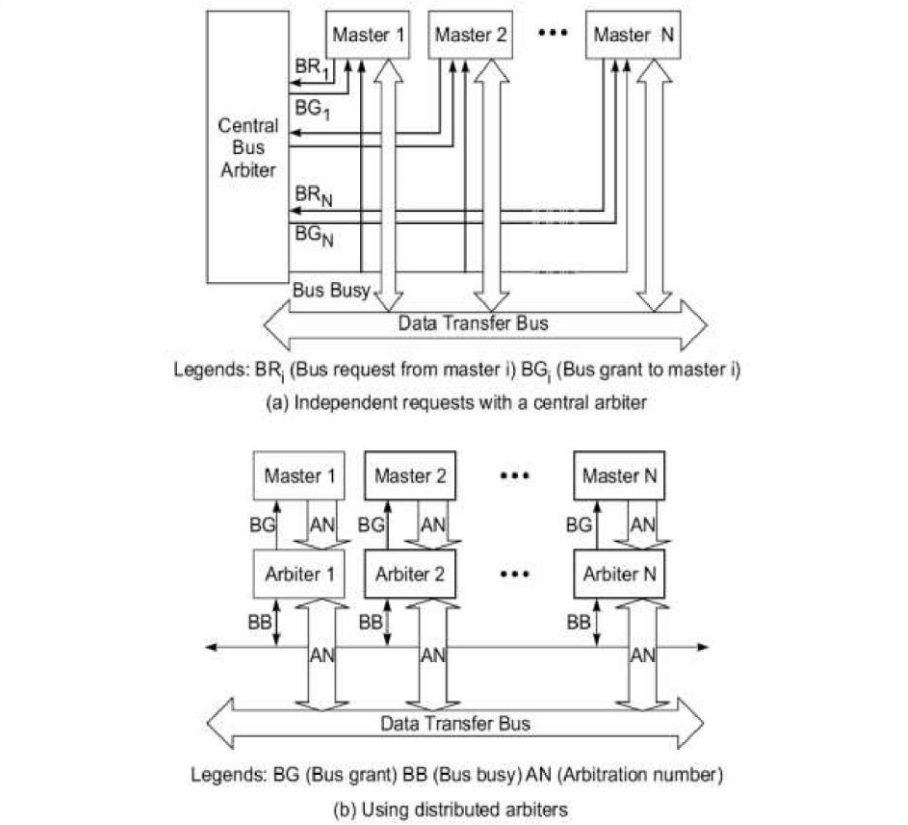


Fig. 5.5 Two bus arbitration schemes using independent requests and distributed arbiters, respectively

Independent Requests and Grants: This scheme provides independent bus-request and grant signals for each master, eliminating the need for daisy chaining. It still requires a central arbiter but can use a priority or fairness-based policy.

- **Advantages:**
  - More flexible and faster than daisy-chained policies
- **Disadvantages:**
  - Requires a larger number of lines, making it costly

2. Illustrate sequential and weak consistency models.

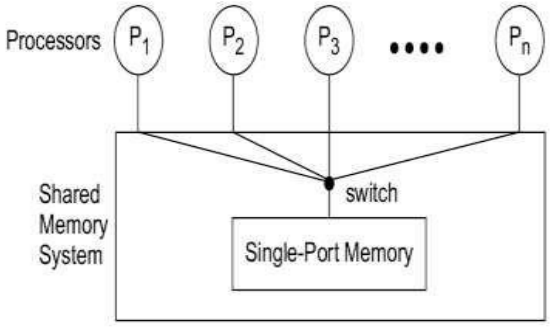


Fig. 5.20 Sequential consistency memory model (Courtesy of Sindhu, Frailong, and Cekleov; reprinted with permission from *Scalable Shared-Memory Multiprocessors*, Kluwer Academic Publishers, 1992)

In multiprocessor systems, consistency models define how memory operations (loads and stores) appear to execute in the system, affecting the behavior of shared memory. Two primary models are Sequential Consistency and Weak Consistency.

Sequential Consistency Model

Definition: Sequential consistency ensures that the operations of all processors are executed in some order that is consistent with the order specified by each individual processor.

Simple Explanation: Imagine you have two friends, Alice and Bob, both writing notes on a shared notepad. Sequential consistency means that if you read the notepad, it will look like Alice and Bob took turns writing, one after the other, in the exact order they each wrote their notes.

Example:

- **Processor P1:**
  - Write(A, 1)
  - Read(B)
- **Processor P2:**
  - Write(B, 1)
  - Read(A)

Imagine P1 and P2 are writing to memory locations A and B. For sequential consistency, if you see P1 write to A first, then you might see:

1. **P1** writes 1 to A.
2. **P2** writes 1 to B.
3. **P1** reads B and sees 1.
4. **P2** reads A and sees 1.

This means operations are interleaved, but the order respects each processor's sequence of operations.

Weak Consistency Model

Definition: Weak consistency allows memory operations to be executed out of order, as long as they become consistent at certain synchronization points.

Simple Explanation: Imagine the same shared notepad, but Alice and Bob can write notes out of order. However, at specific times (synchronization points), they must both pause and ensure their notes are up to date and visible to each other.

Example:

- **Processor P1:**
  - Write(A, 1)
  - Sync
  - Read(B)
- **Processor P2:**
  - Write(B, 1)
  - Sync
  - Read(A)

Here, `Sync` is a point where both processors ensure all previous operations are visible to each other. Between sync points, operations can be out of order.

A possible sequence:

1. **P1** writes 1 to A.
2. **P2** writes 1 to B.
3. **P1** and **P2** reach the sync point, ensuring all writes are visible.
4. **P1** reads B and sees 1.
5. **P2** reads A and sees 1.

Key Differences:

- **Sequential Consistency:** All operations are strictly interleaved in a global order that respects each processor's order.
- **Weak Consistency:** Operations can be out of order except at synchronization points where order is enforced.

Dual Sequential Buffer (DSB) Model

Simple Explanation:

- **Writes are delayed:** When a processor writes to memory, the write operation is buffered and might not be immediately visible to other processors.
- **Ordered per processor:** Each processor's memory operations happen in the order it issued them, but they may be delayed in being seen by others.

Total Store Order (TSO) Model

Simple Explanation:

- **Writes seen in order:** Each processor's writes are seen by other processors in the order they were issued.
- **Reads can be out of order:** A processor might read values that don't immediately reflect the most recent writes by other processors.
- **Immediate self-visibility:** A processor can immediately see its own writes.

3. Discuss any two mapping techniques.

1. Direct Mapped Cache

Explanation: Direct mapped cache is the simplest cache mapping technique where each memory block maps to exactly one cache line.

Key Points:

- **One-to-One Mapping:** Each block in main memory maps to only one cache line.
- **Cache Line:** The position in the cache where a specific memory block can be stored.
- **Address Division:** The memory address is divided into three parts: tag, index, and block offset.
    - **Tag:** Identifies if a block is in the cache.
    - **Index:** Determines the specific cache line.
    - **Block Offset:** Identifies the exact location within the block.

Example: Consider a cache with 8 lines (index range 0 to 7):

- Memory blocks 0, 8, 16, etc., all map to cache line 0.
- Memory blocks 1, 9, 17, etc., all map to cache line 1.
- This continues until block 7, which maps to cache line 7.

When accessing memory address 12:

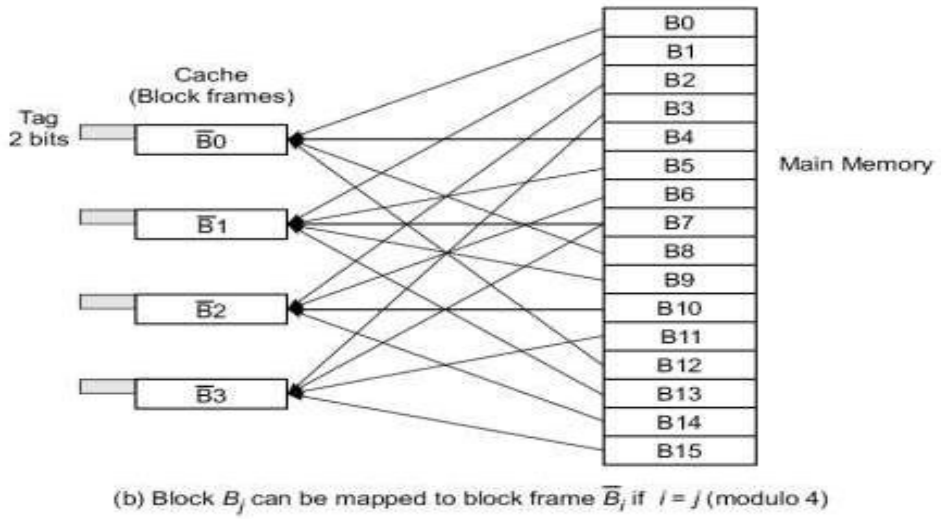
1. Calculate the index: 12 modulo 8 (number of cache lines) = 4.
2. Memory block 12 maps to cache line 4.
3. Check the tag to verify if block 12 is in cache line 4.

Pros:

- Simple and easy to implement.
- Fast access time due to direct mapping.

Cons:

- Cache conflicts: Multiple blocks mapping to the same line cause frequent replacements (thrashing).



(b) Block B_j can be mapped to block frame $\overline{B}_i$ if $i = j$ (modulo 4)

Fig. 5.9 Direct-mapping cache organization and a mapping example

Associative Cache

Explanation: Fully associative cache allows any memory block to be stored in any cache line, offering maximum flexibility.

Key Points:

- **Any-to-Any Mapping:** Any memory block can be placed in any cache line.
- **Address Division:** The memory address is divided into tag and block offset.
    - **Tag:** Used to identify blocks in cache.

    - **Block Offset:** Identifies the exact location within the block.
- **Search Mechanism:** All cache lines are searched in parallel to find a block.

Example: Consider a cache with 8 lines:

- Any memory block can be stored in any of the 8 cache lines.
- When accessing memory address 12, search all cache lines to find the block with the corresponding tag.

Pros:

- Eliminates cache conflicts: Any block can go anywhere in the cache.
- Improved cache utilization.

Cons:

- Higher complexity and cost: Needs hardware to search all cache lines in parallel.
- Slower access time compared to direct mapped cache due to parallel search.

Sector Mapping Cache

Definition: Sector mapping is a cache mapping technique that divides the cache into sectors. Each sector contains multiple cache lines, and each sector is associated with a single tag. This approach aims to improve the utilization of cache space and reduce tag storage overhead.

Key Points:

1. **Sector Division:** The cache is divided into multiple sectors, each containing several cache lines.
2. **Single Tag per Sector:** Each sector has one tag that applies to all the cache lines within that sector.
3. **Lines within Sector:** Each cache line within a sector can store a separate block of data, but they all share the same tag.

Example:

Consider a cache with 2 sectors, each containing 4 cache lines (for simplicity):

- **Sector 0:**
    - Tag: 0x1
    - Lines: 0, 1, 2, 3
- **Sector 1:**
    - Tag: 0x2

- Lines: 4, 5, 6, 7

Memory blocks are mapped to cache lines within a sector, and the sector tag is checked to verify if the data is present.

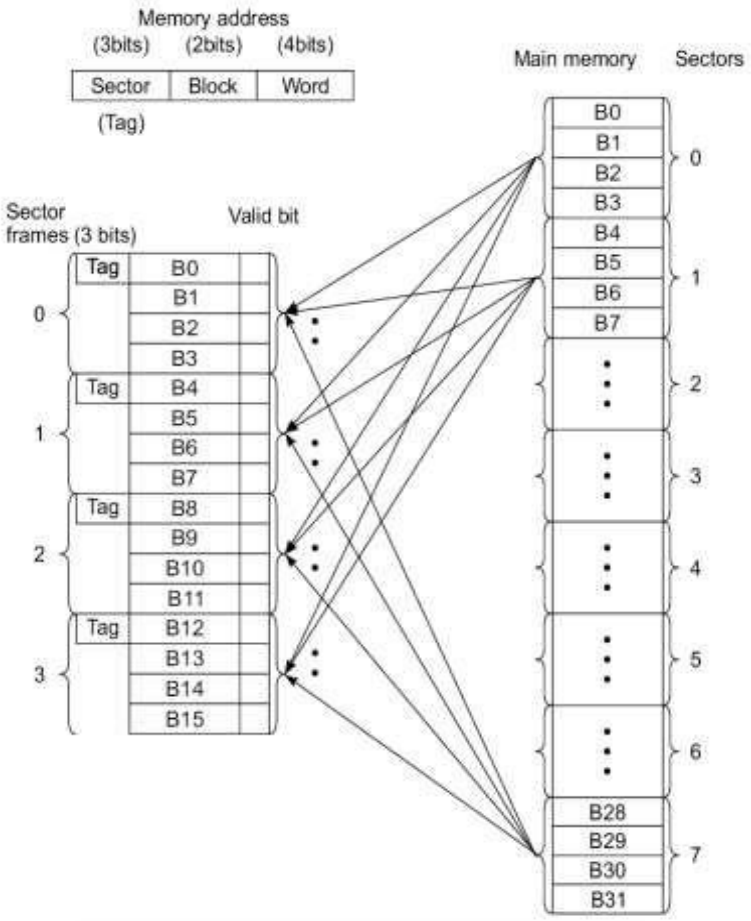


Fig. 5.12 A four-way sector mapping cache organization

4. Explain with diagram Backplane bus Specification

A backplane bus is essential in tightly coupled hardware systems, connecting processors, data storage, and peripherals. It enables seamless communication between devices without interrupting their internal operations. The specification includes various components and protocols to ensure efficient data transfer and arbitration.

Overview

- **Purpose:** Connects processors, data storage, and peripherals, enabling communication without disrupting internal activities.
- **Functionality:** Utilizes timing protocols and operational rules for orderly data transfers and arbitration among multiple requests.

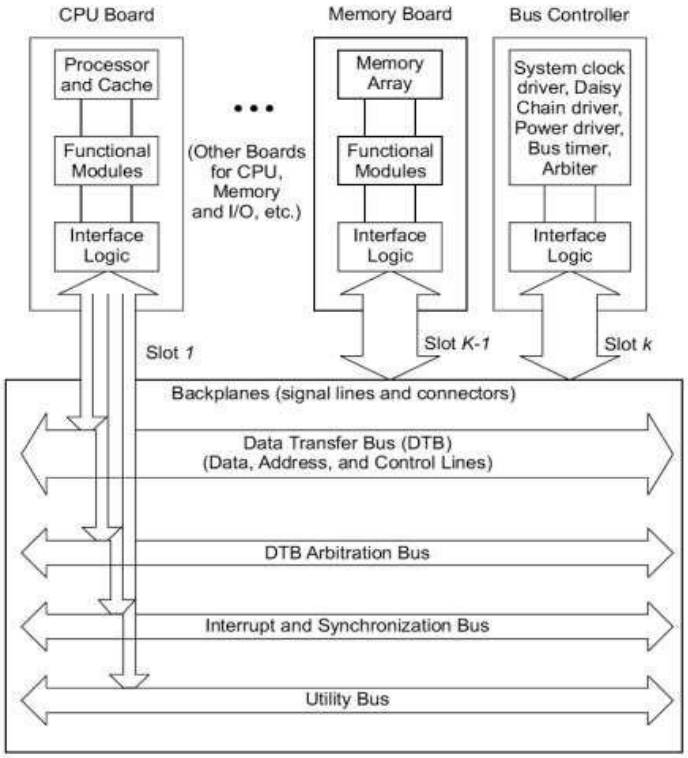


Fig. 5.1 Backplane buses, system interfaces, and slot connections to various functional boards in a multiprocessor system

Components and Mechanisms

Data Transfer Bus (DTB)

The Data Transfer Bus (DTB) consists of three main types of lines:

1. **Data Lines:**
   - Proportional to the memory word length.
   - Carry the actual data being transferred between components.
2. **Address Lines:**
   - Proportional to the log of the address space size.
   - Broadcast data and device addresses.
3. **Control Lines:**
   - Manage read/write operations, timing, and bus error conditions.
   - Ensure proper sequencing and execution of data transfers.

Bus Arbitration and Control

Bus arbitration and control mechanisms ensure that only one device, the master, controls the DTB at a time, while the receiver is referred to as the slave.

1. **Arbitration:**
    - Assigns DTB control to a requester (master).
    - Ensures orderly access to the bus.
2. **Interrupt Lines:**
    - Handle prioritized interrupts.
    - Allow high-priority tasks to gain immediate bus access.
3. **Utility Lines:**
    - Manage periodic timing, system clocking, and power sequences.
    - Include signal lines and connectors for proper communication and control logic.

Functional Modules

Several functional modules support the operation of the backplane bus:

1. **Arbiter:**
    - Grants DTB control to one requester at a time.
    - Ensures fair and efficient bus access.
2. **Bus Timer:**
    - Measures and terminates DTB cycle times.
    - Ensures timely data transfers and avoids bus contention.
3. **Interrupter Module:**
    - Manages interrupt requests.
    - Provides status and ID information for efficient interrupt handling.
4. **Location Monitor:**
    - Monitors DTB data transfers.
    - Ensures accurate and reliable data communication.
5. **System Clock Driver:**
    - Provides clock timing signals.
    - Synchronizes the operations of all connected devices.

5. Explain the following terms associated with cache and memory architecture

I. Low order memory interleaving

II. Atomic v/s non atomic memory

III. Physical address cache vs virtual address cache

IV. Memory bandwidth and fault tolerance

I. Low Order Memory Interleaving

Definition: Low order memory interleaving is a technique used to improve memory access speed by distributing consecutive memory addresses across multiple memory modules or banks.

Key Points:

- **Distribution:** Consecutive memory addresses are spread across different memory banks.
- **Parallel Access:** Multiple memory banks can be accessed simultaneously, improving access speed.
- **Low Order Bits:** The lower order bits of the address determine which memory bank an address maps to.

Example: Consider a system with 4 memory banks (Bank 0, Bank 1, Bank 2, Bank 3). With low order memory interleaving:

- Address 0 maps to Bank 0
- Address 1 maps to Bank 1
- Address 2 maps to Bank 2
- Address 3 maps to Bank 3
- Address 4 maps to Bank 0
- And so on...

This allows for simultaneous access to different banks, speeding up memory operations.

II. Atomic vs Non-Atomic Memory

Atomic Memory Operations:

- **Definition:** Operations that are completed in a single step without any possibility of interruption.
- **Key Points:** Ensures that once an operation starts, it runs to completion without any other operations intervening.
- **Usage:** Critical for synchronization in multi-threaded and multi-processor systems.
- **Example:** An atomic increment operation ensures that reading, incrementing, and writing back a value happens without interruption.

Non-Atomic Memory Operations:

- **Definition:** Operations that can be interrupted, and other operations can occur between their start and completion.
- **Key Points:** More susceptible to race conditions and inconsistencies in multi-threaded environments.
- **Usage:** Generally simpler but need careful handling in concurrent scenarios.

- **Example:** A non-atomic read-modify-write operation can be interrupted, leading to potential inconsistencies if other operations occur in between.

III. Physical Address Cache vs Virtual Address Cache

Physical Address Cache:

- **Definition:** A cache that uses physical addresses for tagging and indexing.
- **Key Points:**
  - Avoids issues related to address translation.
  - Simpler coherence protocols.
- **Performance:** Can be slower due to the need to translate virtual addresses to physical addresses before accessing the cache.

Virtual Address Cache:

- **Definition:** A cache that uses virtual addresses for tagging and indexing.
- **Key Points:**
  - Faster access since it avoids initial address translation.
  - Can suffer from aliasing issues (different virtual addresses mapping to the same physical address).
- **Performance:** Generally faster but requires mechanisms to handle address translation consistency and aliasing.

IV. Memory Bandwidth and Fault Tolerance

Memory Bandwidth:

- **Definition:** The rate at which data can be read from or written to memory by the processor.
- **Key Points:**
  - Measured in bytes per second (B/s).
  - Critical for performance in data-intensive applications.
- **Improvement:** Can be enhanced through techniques like memory interleaving, higher bus speeds, and wider data paths.

Fault Tolerance:

- **Definition:** The ability of a system to continue functioning correctly even when some of its components fail.
- **Key Points:**

- **Error Detection and Correction:** Techniques like parity checks, ECC (Error-Correcting Code) memory to detect and correct errors.
- **Redundancy:** Use of redundant components to take over in case of a failure .
- **Graceful Degradation:** Ability of the system to reduce functionality in a controlled way rather than failing completely.

6. Explain Synchronous and Asynchronous Timing Protocols

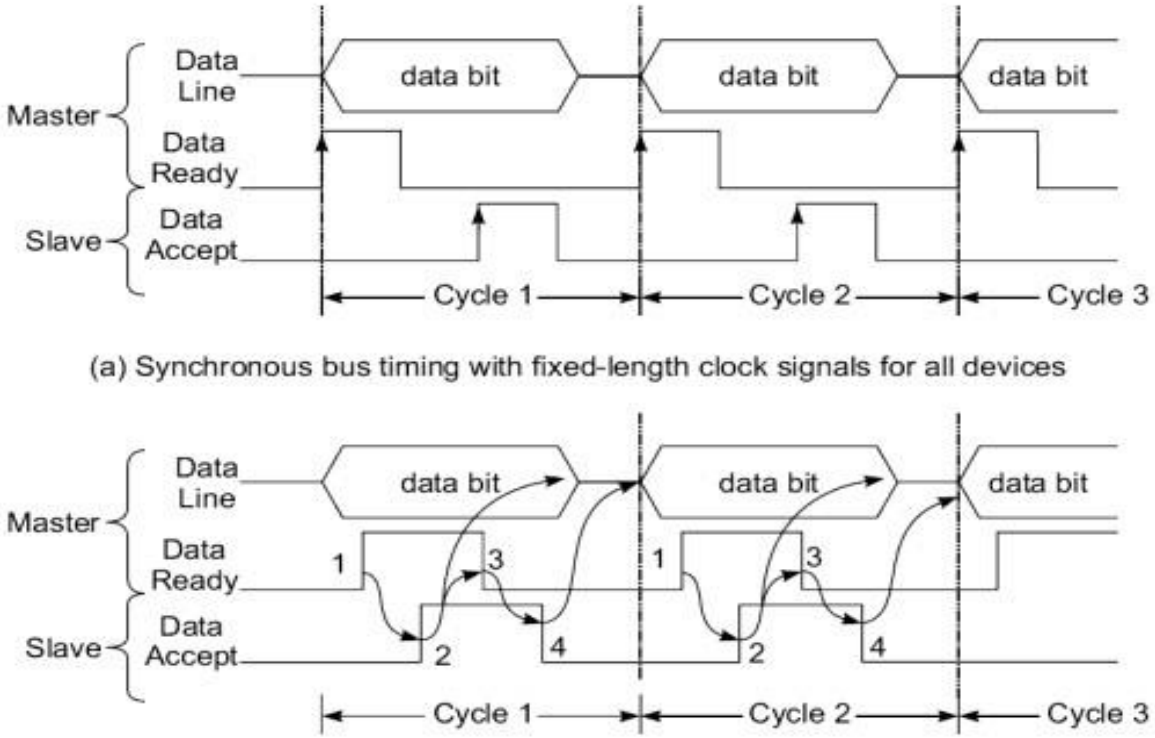


Fig. 5.3 Synchronous versus asynchronous bus timing protocols

Synchronous Bus Timing (Figure 5.3a)

Description:

- **Data Line:** Transmits the actual data bits.
- **Data Ready:** Indicates when the data on the data line is valid and ready to be read.
- **Data Accept:** Indicates when the data has been successfully read and accepted.

Key Features:

1. **Fixed-Length Clock Signals:** The timing of data transmission is controlled by fixed-length clock cycles.

2. **Synchronization:** Both the master and slave devices are synchronized to the same clock signal, ensuring that data bits are transmitted and received at precisely defined intervals.
3. **Cycle Timing:** Each cycle is consistent in length, as shown by the uniform width of each cycle (Cycle 1, Cycle 2, Cycle 3).

Asynchronous Bus Timing (Figure 5.3b)

Description:

- **Data Line:** Transmits the actual data bits.
- **Data Ready:** Indicates when the data on the data line is valid and ready to be read.
- **Data Accept:** Indicates when the data has been successfully read and accepted.

Key Features:

1. **Variable Length Signals:** Data transmission is controlled using a four-edge handshaking mechanism, which allows for variable-length signals.
2. **Handshaking:** The process involves multiple steps:
    - **Step 1:** The master sets the Data Ready signal to indicate that data is available.
    - **Step 2:** The slave sets the Data Accept signal to indicate it is ready to read the data.
    - **Step 3:** The master clears the Data Ready signal once the data has been read.
    - **Step 4:** The slave clears the Data Accept signal, indicating it has accepted the data.
3. **Interlocking Mechanism:** This handshaking mechanism ensures that both devices are synchronized in terms of data transfer without relying on a fixed clock signal.
4. **Flexibility:** Different speed devices can communicate effectively because the protocol does not depend on a fixed clock cycle.

7. With Neat diagram C access Interleaved Memory organization

8. With neat diagram explain High order interleaving

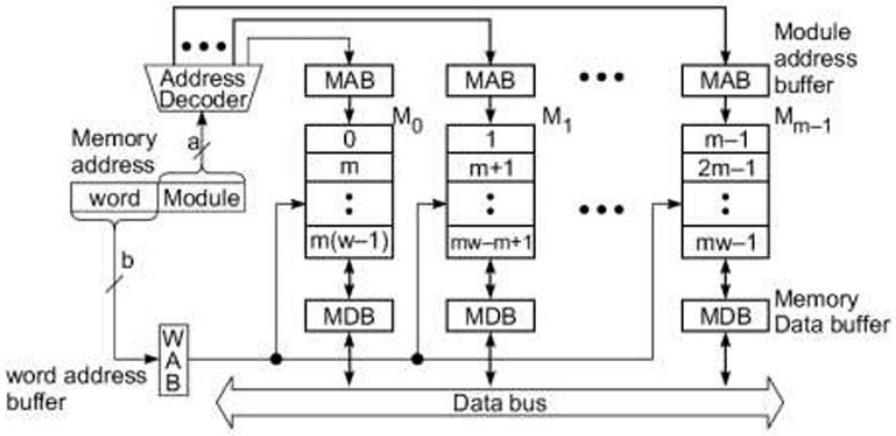
Low-Order m-Way Interleaving (C-Access Memory Scheme)

In low-order interleaving, memory addresses are distributed across multiple memory modules such that consecutive addresses are placed in different modules. This allows for parallel access to different memory locations, improving overall memory throughput.

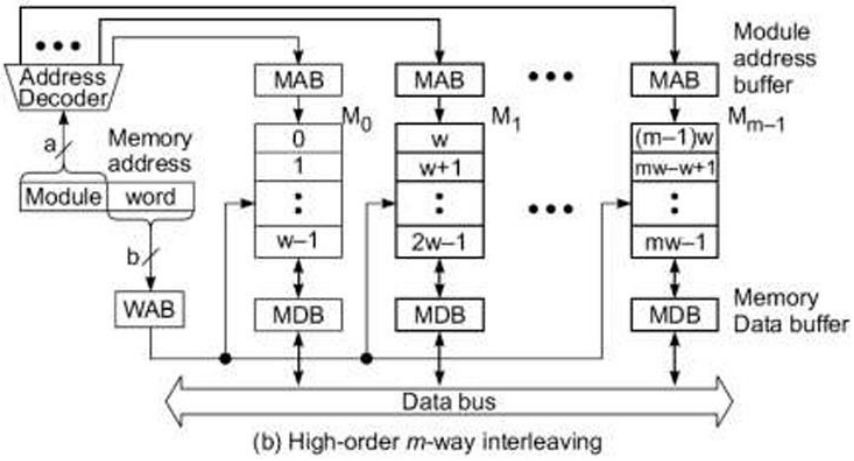
Diagram Explanation:

- **Address Decoder**: The address decoder splits the memory address into two parts: a (which identifies the module) and b (which identifies the word within the module).

- **Memory Modules (M0, M1, ..., Mm-1)**: Each module contains a subset of the memory addresses. For instance, M0 contains addresses 0, m, 2m, ..., (w-1)m, while M1 contains addresses 1, m+1, 2m+1, ..., (w-1)m+1, and so on.
- **Module Address Buffer (MAB)**: Each module has its address buffer to hold the memory addresses for that module.
- **Memory Data Buffer (MDB)**: Each module has its data buffer to hold the data being transferred.
- **Data Bus**: The data bus is used for data transfer between the memory modules and the CPU.



(a) Low-order *m*-way interleaving (the C-access memory scheme)



(b) High-order *m*-way interleaving

Two interleaved memory organizations with $m = 2^a$ modules and $w = 2^b$ words per module (word addresses shown in boxes)

High-Order m-Way Interleaving

In high-order interleaving, memory addresses are distributed across multiple memory modules in such a way that blocks of consecutive addresses are placed in different modules.

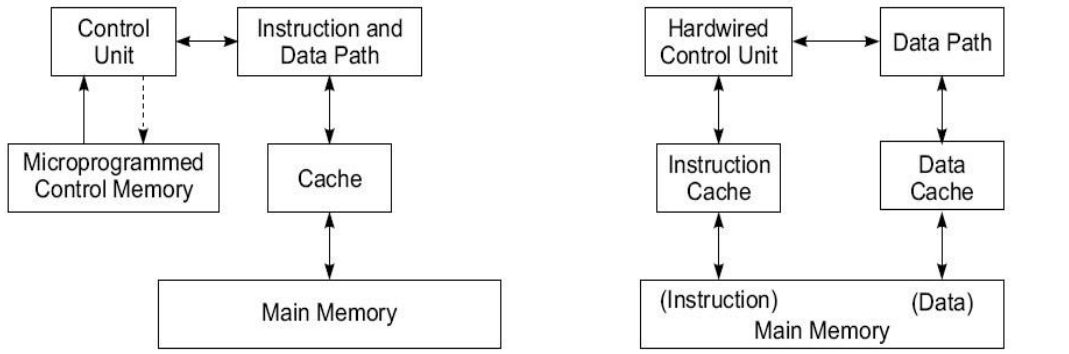
Diagram Explanation:

- **Address Decoder**: Similar to low-order interleaving, the address decoder splits the memory address into parts: a (module identifier) and b (word within the module).

- **Memory Modules (M0, M1, ..., Mm-1)**: Each module contains a block of consecutive addresses. For example, M0 contains addresses 0, 1, 2, ..., w-1, M1 contains addresses w, w+1, w+2, ..., 2w-1, and so on.
- **Module Address Buffer (MAB)**: Each module has its address buffer to hold the memory addresses for that module.
- **Memory Data Buffer (MDB)**: Each module has its data buffer to hold the data being transferred.
- **Data Bus**: The data bus is used for data transfer between the memory modules and the CPU.

1) Explain with diagram RISC & CISC OR Explain The difference?

RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer) are two fundamental approaches to processor design. They differ primarily in their philosophy towards instruction set architecture and execution.



(a) The CISC architecture with microprogrammed control and unified cache

(b) The RISC architecture with hardwired control and split instruction cache and data cache

Fig. 4.4 Distinctions between typical RISC and typical CISC processor architectures (Courtesy of Gordon Bell, 1989)

CISC (Complex Instruction Set Computer)

Characteristics:

- **Many different instructions:** Numerous and varied instructions.
- **Many operand data types:** Supports various types of data operands.
- **Many operand addressing formats:** Multiple ways to address operands.
- **Few general-purpose registers:** Limited number of general-purpose registers.
- **High-level language alignment:** Instructions closely match high-level language constructs.

Architectural Distinctions:

- **Unified cache:** One cache for both instructions and data.
- **Control units:** Earlier models used microprogrammed control units with ROM; some newer systems use hard-wired control units.

Advantages:

- **Smaller program size:** Requires fewer instructions to perform tasks.
- **Simpler control unit design:** Less complex control units.
- **Simpler compiler design:** Easier to design compilers.

RISC (Reduced Instruction Set Computer)

Characteristics:

- **Fewer instructions:** Simpler and fewer instructions compared to CISC.
- **Fixed instruction format:** Standardized instruction format, typically 32 bits.
- **Simple operand addressing:** Easier and more straightforward addressing.
- **Many registers:** A large number of registers available for use.
- **Small CPI and high clock rates:** Cycles per instruction (CPI) close to 1, resulting in high clock rates.

Architectural Distinctions:

- **Separate caches:** Different caches for instructions and data.
- **Hard-wired control units:** Uses hard-wired control units for faster operation.

Advantages:

- **Potential for higher speed:** Can be faster due to simpler instructions.
- **Many registers:** More registers improve performance and reduce memory access.

Feature	**CISC (Complex Instruction Set Computer)**	**RISC (Reduced Instruction Set Computer)**
Instruction Set	Many different instructions	Fewer, simpler instructions
Operand Data Types	Many different operand data types	Simple operand data types
Operand Addressing	Many operand addressing formats	Fixed instruction format and simple operand addressing
Registers	Few general-purpose registers	Many general-purpose registers
Instruction Format	Variable length	Fixed length (e.g., 32 bits)
Cache	Unified cache for instructions and data	Separate caches for instructions and data
Control Units	Microprogrammed control units (older), some hard-wired (newer)	Hard-wired control units
Program Size	Smaller (fewer instructions)	Larger (more instructions)

Compiler Design	Simpler compiler design	More complex compiler design
CPI (Cycles per Instruction)	Higher CPI	Lower CPI (close to 1)
Clock Rates	Typically lower	Higher clock rates
Performance	May be slower due to complex instructions	Potentially faster due to simpler instructions

2) Explain VLIW architecture With Its Instruction Pipelining?

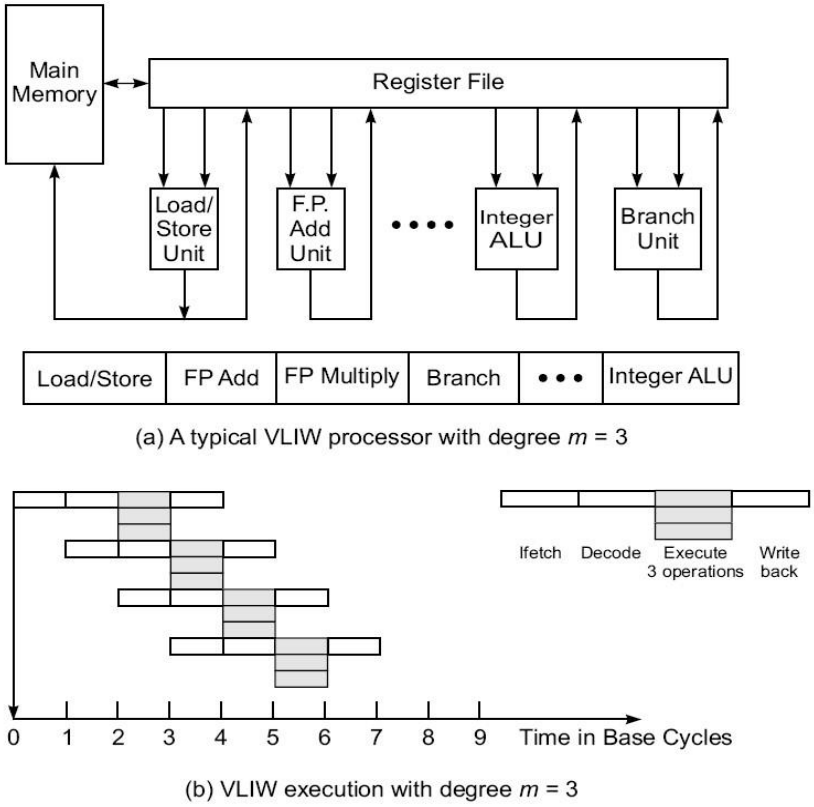


Fig. 4.14 The architecture of a very long instruction word (VLIW) processor and its pipeline operations (Courtesy of Multiflow Computer, Inc., 1987)

VLIW (Very Long Instruction Word) Architecture:

Concept:

- VLIW stands for Very Long Instruction Word.
- It is designed to exploit instruction-level parallelism (ILP) by packing multiple operations into a single long instruction word.

- Each long instruction word (called a "bundle") contains multiple independent instructions that can be executed in parallel.

Key Features:

- **Parallelism:** VLIW processors execute multiple instructions simultaneously by including several operations in a single instruction word.
- **Simple Hardware:** The control logic in VLIW processors is simpler compared to superscalar processors because the compiler handles the instruction scheduling.
- **Compiler Dependency:** VLIW relies heavily on the compiler to identify parallelism and schedule instructions effectively.

Example: A VLIW instruction word might include:

- An arithmetic operation (e.g., add)
- A memory operation (e.g., load)
- A branch operation (e.g., jump)
- A floating-point operation (e.g., multiply)

Instruction Pipelining in VLIW

Concept:

- Instruction pipelining divides the execution process into several stages, allowing multiple instructions to be processed simultaneously.

Pipelining Stages:

1. **Fetch:** Get the instruction from memory.
2. **Decode:** Identify the operations in the instruction.
3. **Issue:** Assign operations to the appropriate functional units.
4. **Execute:** Perform the operations.
5. **Write-back:** Save the results back to registers or memory.

Pipelining in VLIW:

- Each pipeline stage works on different instruction words at the same time.
- Operations within each instruction word are issued to different functional units in parallel.

3) Explain What Is Virtual Memory technology and It's different Virtual Memory models?

Virtual Memory Technology

Concept:

- Virtual memory is a memory management technique that provides an "illusion" of a large, continuous memory space, even if the physical memory (RAM) is limited.
- It allows a computer to use more memory than physically available by using disk space to extend RAM.

How It Works:

- **Paging:** Divides memory into fixed-sized blocks called pages. The operating system (OS) maintains a page table to keep track of where pages are located.
- **Swapping:** Moves data between RAM and disk storage (swap space) as needed. Inactive pages in RAM are moved to disk, freeing up space for active pages.
- **Address Translation:** Maps virtual addresses to physical addresses using a memory management unit (MMU).

Advantages:

- **Increased Memory:** Allows running large applications that require more memory than physically available.

Disadvantages:

- **Performance Overhead:** Frequent swapping between RAM and disk can slow down the system (known as thrashing).
- **Complexity:** Increases the complexity of the OS and hardware.

Private Virtual Memory

Description:

- **Private Virtual Memory** refers to a memory model where each process has its own private virtual address space.
- This means that the memory allocated to one process is not accessible to any other process, ensuring isolation and security.

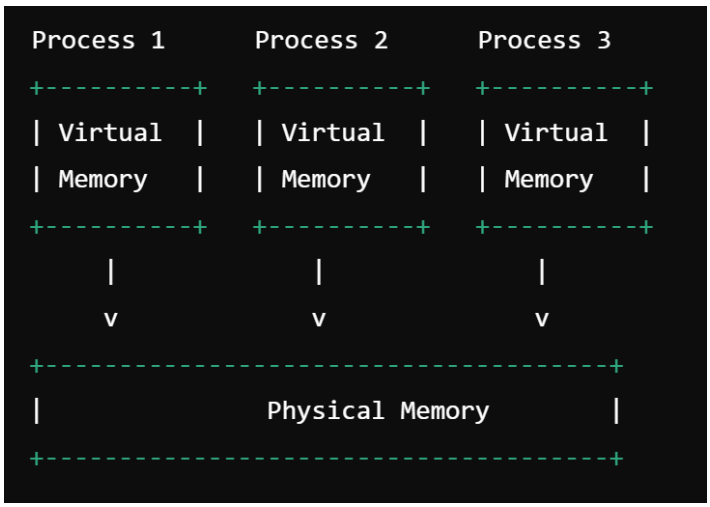
Characteristics:

1. **Isolation:** Each process operates in its own virtual address space, which prevents processes from interfering with each other's memory.
2. **Security:** Because memory spaces are isolated, a faulty or malicious process cannot corrupt the memory of another process.
3. **Independence:** Memory management for each process is handled independently, which simplifies memory allocation and deallocation.

Example:

- Modern operating systems like Windows, Linux, and macOS use private virtual memory to ensure that user applications run in isolation from each other.

Diagram:



Shared Virtual Memory

Description:

- **Shared Virtual Memory** allows multiple processes to share a common virtual address space.
- This model enables processes to access shared data without needing explicit IPC mechanisms.

Characteristics:

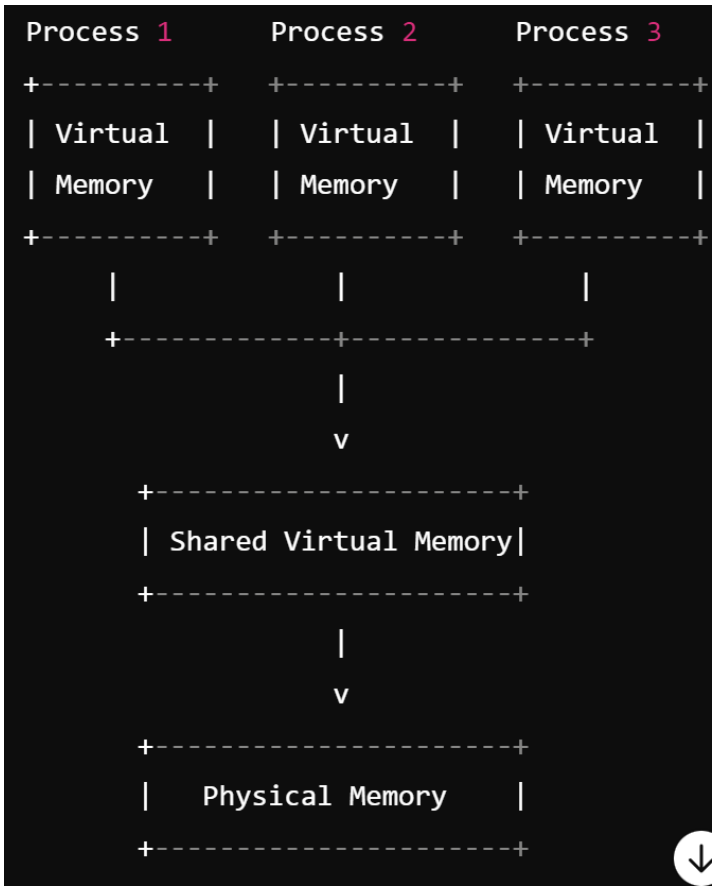
1. **Shared Resources:** A region of memory can be marked as shared, allowing multiple processes to read from and write to it.
2. **Synchronization:** Processes must synchronize access to shared memory regions to avoid data corruption.

3. **Efficiency:** Reduces the need for data duplication and facilitates efficient data sharing.

Advantages:

- **Efficient Communication:** Simplifies data sharing between processes, improving communication efficiency.
- **Resource Saving:** Reduces memory usage by avoiding data duplication.
- **Collaborative Work:** Facilitates collaborative work between processes on shared data.

Diagram



4) Explain Memory Hierarchy Technology?

1. Registers

Description:

- The smallest and fastest memory component.
- Located inside the CPU.

- **Speed:** Extremely fast access (few nanoseconds).
- **Capacity:** Very limited (usually 32 to 128 registers).
- **Cost:** Very high per bit.

Example: Program Counter (PC), Accumulator, Base register.

2. Cache Memory

Description:

- A small, fast memory located between the CPU and main memory.
- Used to temporarily hold frequently accessed data and instructions.

- **Speed:** Faster than main memory but slower than registers (few nanoseconds to tens of nanoseconds).
- **Capacity:** Larger than registers but smaller than main memory (typically kilobytes to a few megabytes).
- **Cost:** Less expensive than registers but more expensive than main memory.

Use:

- Reduces the average time to access data from the main memory.
- Stores copies of data from frequently used main memory locations

3. Main Memory (RAM)

Description:

- The primary storage area for data and programs currently in use.

- **Speed:** Slower than cache memory (tens of nanoseconds).
- **Capacity:** Larger than cache memory (typically gigabytes).
- **Cost:** Less expensive than cache memory.

Use:

- Stores the operating system, application programs, and data currently being used.
- Volatile memory: loses its content when the power is turned off.

4. Secondary Storage

Description:

- Non-volatile memory used for long-term storage of data.
- Examples include Hard Disk Drives (HDDs) and Solid State Drives (SSDs).

- **Speed:** Slower than main memory (milliseconds for HDD, microseconds for SSD).
- **Capacity:** Much larger than main memory (typically terabytes).
- **Cost:** Less expensive per bit compared to main memory.

Use:

- Stores data and programs not currently in use.
- Persistent storage: retains data even when the power is off.

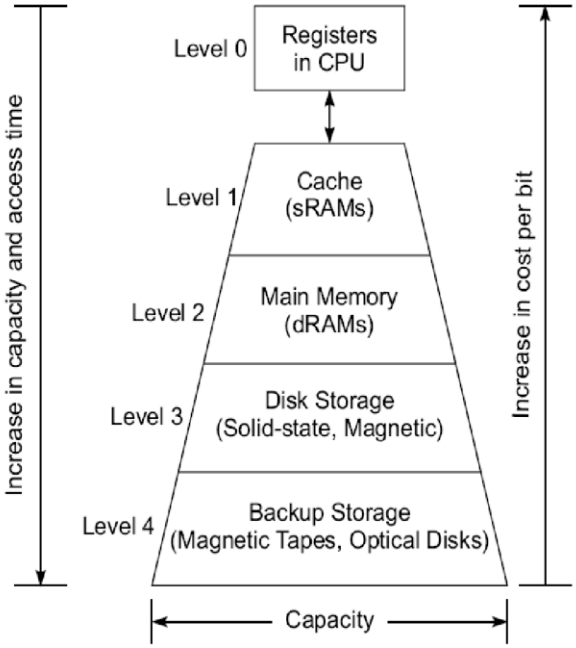


Fig. 4.17 A four-level memory hierarchy with increasing capacity and decreasing speed and cost from low to high levels

5) Explain Inclusion, Coherence, and Locality?

1. Inclusion Property

- **Definition**: Memory levels (like cache and main memory) are organized such that each higher level includes all items of information from the lower level.
- **Implication**: If a data item is in a higher memory level (like main memory), it should also be present in the lower memory levels (like cache).

- **Inverse**: The reverse is not necessarily true; a data item present in a higher level might not be in the lower levels, resulting in a "miss" when accessed from the lower level.

2. Coherence Property

- **Definition**: Copies of data items across different memory levels (like cache and main memory) must be consistent.
- **Strategies**:
    - **Write-through**: Immediately update all corresponding copies in higher levels when a data item in a lower level is modified. This ensures all copies are always consistent but is more resource-intensive.
    - **Write-back**: Delay the update of higher-level copies until the lower-level item is replaced or evicted. This is more efficient but can lead to temporary inconsistencies if not managed carefully, especially in systems with multiple processors sharing memory.

3. Locality of References

- **Definition**: Memory references (for instructions or data) tend to cluster in certain patterns over time.
- **Dimensions of Locality**:
    - **Temporal Locality**: If a memory location is accessed now, it's likely to be accessed again in the near future.
    - **Spatial Locality**: If a memory location is accessed now, nearby locations (in terms of memory addresses) are likely to be accessed soon after.
    - **Sequential Locality**: If a memory location is accessed now, subsequent locations (like the next elements in an array) are likely to be accessed in order.
- **Implication**: Leveraging these patterns allows for optimizing memory access and cache performance, as systems can prefetch or cache data more effectively based on expected access patterns.

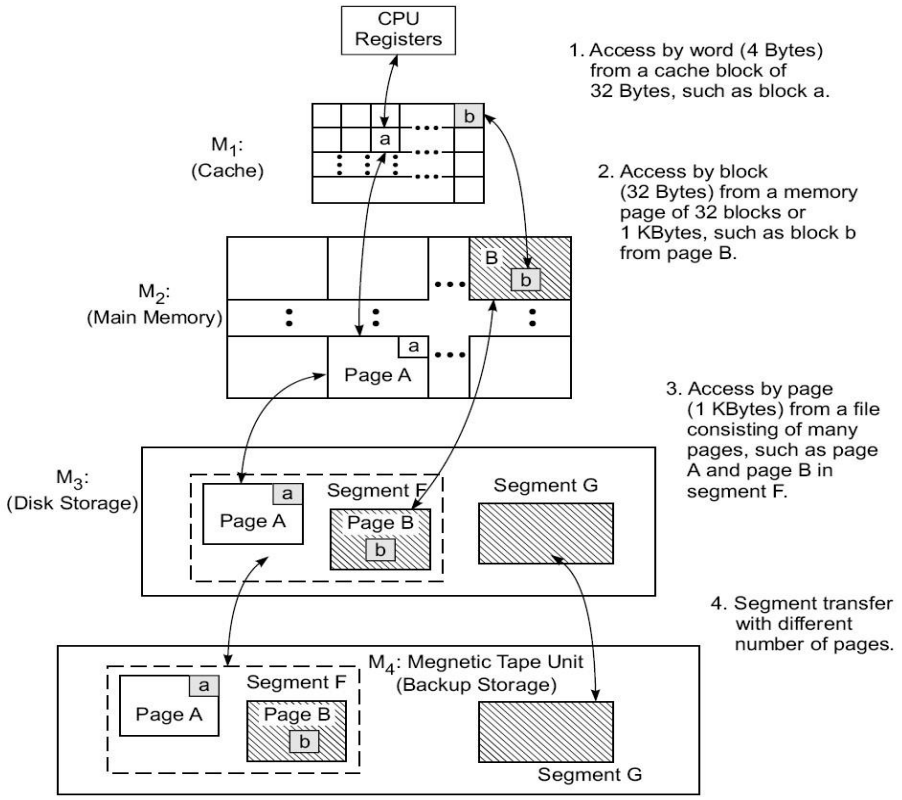


Fig. 4.18 The inclusion property and data transfers between adjacent levels of a memory hierarchy

6) Explain Page replacement policies with help of Example.

Page replacement policies are algorithms in operating systems that decide which memory pages to swap out, write to disk, or replace when a new page needs to be loaded into memory. These policies are crucial for managing memory efficiently and ensuring good system performance.

Common Page Replacement Policies:

1. **First-In-First-Out (FIFO)**
2. **Least Recently Used (LRU)**
3. **Optimal Page Replacement**
4. **Least Frequently Used (LFU)**
5. **Circular FIFO**

1. First-In-First-Out (FIFO)

Description:

- The oldest page in memory is replaced first.
- Easy to implement using a queue.

Example:

Consider a page reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3 Assume 3 frames available.

Steps:

1. Load pages 7, 0, 1 into frames.
2. Replace 7 with 2 (FIFO).
3. Replace 0 with 3.
4. Replace 1 with 0.
5. Continue in FIFO order.

Page Faults:

Page Reference	Memory Frames	Page Fault
7	7 - -	Yes
0	7 0 -	Yes
1	7 0 1	Yes
2	2 0 1	Yes
0	2 0 1	No
3	2 3 1	Yes
0	2 3 0	Yes
4	4 3 0	Yes
2	4 2 0	Yes
3	4 2 3	Yes

Total Page Faults = 9

2. Least Recently Used (LRU)

Description:

- Replaces the page that has not been used for the longest time.
- Can be implemented using a stack or a counter.

Example:

Consider the same page reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3 Assume 3 frames available.

Steps:

1. Load pages 7, 0, 1 into frames.
2. Replace the least recently used page when needed.

Page Faults:

Page Reference	Memory Frames	Page Fault
7	7 - -	Yes
0	7 0 -	Yes
1	7 0 1	Yes
2	2 0 1	Yes
0	2 0 1	No
3	2 0 3	Yes
0	2 0 3	No
4	4 0 3	Yes
2	4 2 3	Yes
3	4 2 3	No

Total Page Faults = 7

3. Optimal Page Replacement

Description:

- Replaces the page that will not be used for the longest time in the future.
- Provides the lowest possible page fault rate but is impractical because it requires future knowledge.

Example:

Consider the same page reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3 Assume 3 frames available.

Steps:

1. Load pages 7, 0, 1 into frames.
2. Replace the page that will not be used for the longest time.

Page Faults:

Page Reference	Memory Frames	Page Fault
7	7 - -	Yes
0	7 0 -	Yes
1	7 0 1	Yes
2	2 0 1	Yes
0	2 0 1	No
3	2 0 3	Yes
0	2 0 3	No
4	2 4 3	Yes
2	2 4 3	No
3	2 4 3	No

Total Page Faults = 6

4. Least Frequently Used (LFU)

Description:

- Replaces the page with the lowest count of accesses.
- Keeps track of how often each page is used.

Example:

Consider the same page reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3 Assume 3 frames available.

Steps:

1. Load pages 7, 0, 1 into frames and initialize their counts.
2. Replace the least frequently used page when needed.

Page Faults:

Page Reference	Memory Frames	Page Fault
7	7 - - (1)	Yes
0	7 0 - (1)	Yes
1	7 0 1 (1)	Yes
2	2 0 1 (1)	Yes
0	2 0 1 (2)	No
3	2 0 3 (1)	Yes
0	2 0 3 (3)	No
4	4 0 3 (1)	Yes
2	4 2 3 (1)	Yes
3	4 2 3 (2)	No

Total Page Faults = 8

7) Explain Characteristics of Symbolic Processor

1. **Representation of Data**:
   - Symbolic processors represent data in the form of symbols and expressions, rather than just numbers.
   - They work with lists, trees, and graphs to represent complex data structures.
2. **Complex Data Types**:
   - Support for complex data types such as strings, lists, sets, and other abstract data types.
   - These data types are crucial for symbolic computations and manipulations.
3. **Specialized Instructions**:
   - Provide specialized instructions for manipulating symbols, lists, and other abstract data types.
   - Operations include pattern matching, unification, symbolic differentiation, and integration.
4. **Dynamic Memory Management**:
   - Efficient garbage collection to handle dynamic memory allocation and deallocation.
   - This is important for managing the memory used by complex data structures and ensuring efficient memory usage.
5. **Parallelism**:
   - Ability to exploit parallelism inherent in symbolic computations.
   - Some symbolic processors support parallel execution of independent tasks to improve performance.

6. **Language Support**:
    - Often designed to work with specific high-level programming languages such as LISP, Prolog, or ML.
    - These languages are tailored for symbolic computations and provide powerful constructs for manipulating symbols and expressions.
7. **Rule-Based Processing**:
    - Support for rule-based processing and inference mechanisms.
    - Useful for expert systems, logic programming, and other AI applications where rules are used to derive conclusions from known facts.
8. **Pattern Matching**:
    - Efficient pattern matching capabilities for recognizing and manipulating patterns in data.
    - Essential for applications in NLP and AI where patterns are matched and processed.
9. **Tree and Graph Processing**:
    - Optimized for processing tree and graph structures.
    - Many symbolic computations involve tree-like or graph-like data representations, such as parse trees in compilers and knowledge representation in AI.
10. **Non-Deterministic Computation**:
    - Support for non-deterministic computation models.
    - Allows for exploring multiple possibilities simultaneously, which is useful in AI search algorithms and problem-solving.

Example Applications:

1. **Artificial Intelligence**:
    - Symbolic processors are used in AI for tasks such as natural language understanding, automated reasoning, and expert systems.
2. **Mathematical Computation**:
    - Used in computer algebra systems (CAS) for symbolic mathematics, such as symbolic differentiation, integration, and solving algebraic equations.
3. **Natural Language Processing**:
    - Essential for parsing and understanding human language, involving complex pattern matching and manipulation of symbolic data.
4. **Knowledge Representation and Reasoning**:
    - Used in representing knowledge bases and performing logical inference in knowledge-based systems.

8) Explain Typical superscalar RISC processor consisting of unit integer with diagram

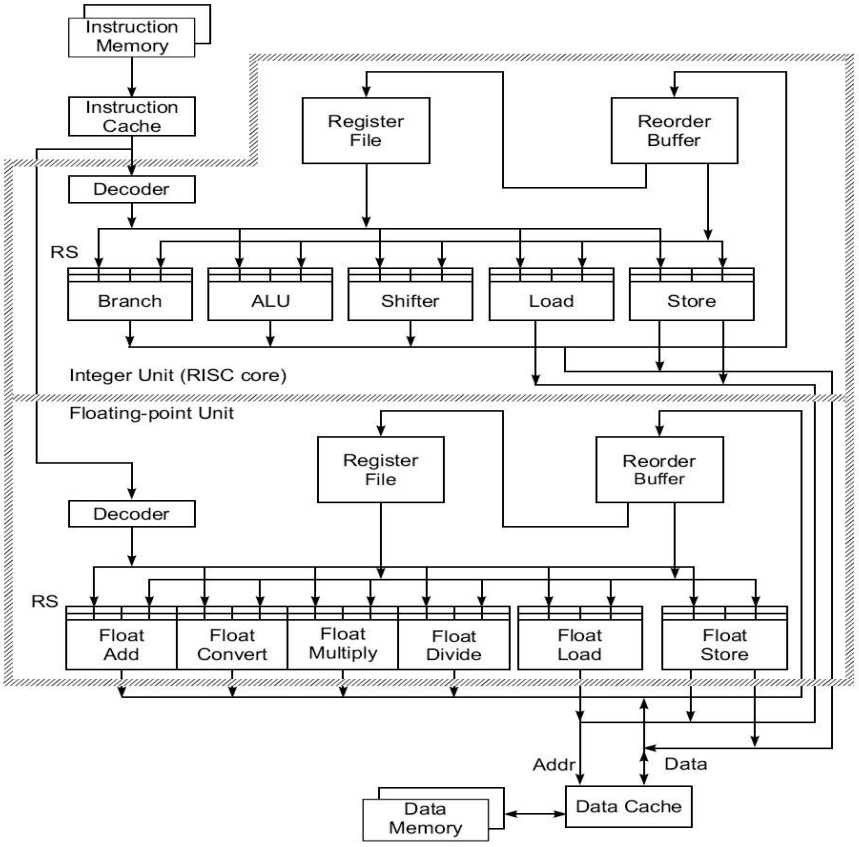


Fig. 4.12 A typical superscalar RISC processor architecture consisting of an integer unit and a floating-point unit (Courtesy of M. Johnson, 1991; reprinted with permission from Prentice-Hall, Inc.)

A superscalar RISC processor can execute multiple instructions simultaneously by dispatching them to appropriate execution units. The architecture typically includes multiple functional units for parallel instruction processing. Here, we focus on a superscalar RISC processor with an integer unit.

Key Characteristics:

1. **Superscalar Architecture**:
   - **Multiple Instruction Issue**: Capable of issuing more than one instruction per clock cycle.
   - **Parallel Execution Units**: Several execution units work in parallel to handle different types of operations (e.g., integer, floating-point, branch).
2. **RISC Principles**:
   - **Reduced Instruction Set**: Simplified instruction set for faster execution.
   - **Fixed-Length Instructions**: Typically 32-bit fixed-length instructions for consistency and speed.
   - **Load/Store Architecture**: Operations are performed between registers, with separate instructions for loading/storing data to/from memory.

Components of a Superscalar RISC Processor:

1. **Instruction Fetch Unit (IFU)**:
    - Fetches multiple instructions from memory.
    - Utilizes branch prediction to improve flow control and reduce stalls.
2. **Instruction Decode Unit (IDU)**:
    - Decodes fetched instructions.
    - Determines instruction types and operand locations.
3. **Register Renaming Unit (RRU)**:
    - Renames registers to avoid data hazards.
    - Helps in out-of-order execution by allowing instructions to use any available physical registers.
4. **Instruction Dispatch Unit (IDU)**:
    - Dispatches decoded instructions to appropriate execution units.
5. **Execution Units**:
    - **Integer Execution Unit (IEU)**: Executes integer arithmetic and logic operations.
    - **Floating Point Unit (FPU)**: Handles floating-point operations (if present).
    - **Branch Execution Unit (BEU)**: Manages branch instructions.
6. **Load/Store Unit (LSU)**:
    - Handles memory operations.
    - Executes load and store instructions, managing data transfer between registers and memory.
7. **Commit Unit (CU)**:
    - Commits completed instructions to ensure program correctness.
    - Handles precise exceptions and maintains instruction order.

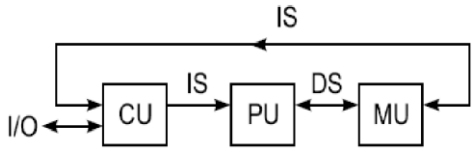
1) Describe Flynn's classification of computer architecture.

1. Single Instruction, Single Data (SISD)

- **Description**: This is the traditional sequential computer architecture where a single instruction stream is processed by a single data stream. One instruction operates on one data element at a time.
- **Example**: Classic single-core processors, such as early Intel processors.

Basic Concept: One instruction operates on one piece of data at a time.
Execution: Instructions are executed one after another, in sequence.
Common Use: Most traditional computers, like PCs and single CPU workstations.

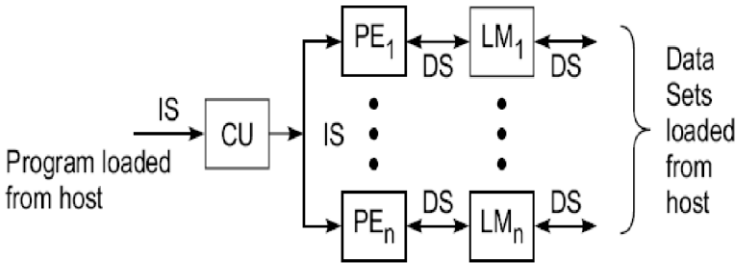
Examples: Early Intel processors, classic single-core CPUs.



(a) SISD uniprocessor architecture

2. Single Instruction, Multiple Data (SIMD)

- **Description**: A single instruction is applied to multiple data elements simultaneously. This is useful for tasks that involve performing the same operation on a large dataset.
- **Example**: Vector processors, GPUs (Graphics Processing Units).
  - **Basic Concept**: One instruction is applied to multiple pieces of data at the same time.
  - **Execution**: All processing units perform the same operation on different data elements simultaneously.
  - **Common Use**: Tasks with repetitive operations on large datasets, such as image processing and video games.
  - **Examples**: Graphics cards (GPUs), vector processors like Cray supercomputers.

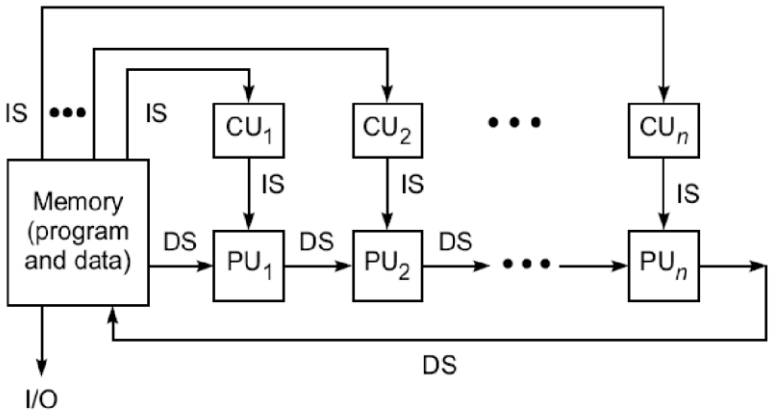


(b) SIMD architecture (with distributed memory)

1. Multiple Instruction, Single Data (MISD)

Description: MISD systems are quite rare and involve multiple processors executing different instructions on the same data stream.

- **Basic Concept**: Different instructions operate on the same data.
- **Execution**: Each processor performs different operations on the same data.
- **Common Use**: Specialized tasks, such as fault-tolerant systems.
- **Examples**: Some fault-tolerant computers, specific types of signal processing systems.



(d) MISD architecture (the systolic array)

Fig. 1.3 Flynn's classification of computer architectures (Derived from Michael Flynn,

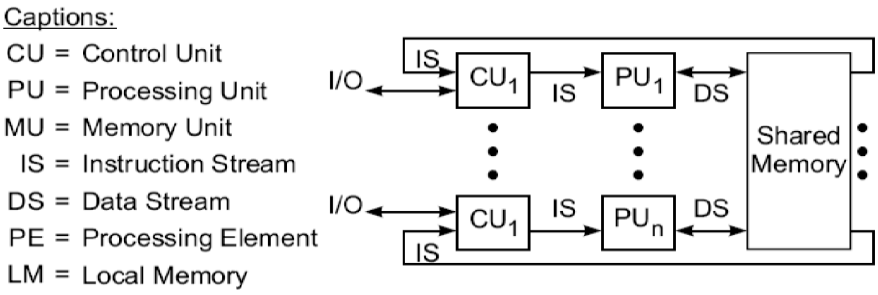
.

2. Multiple Instruction, Multiple Data (MIMD)

 Description: MIMD systems consist of multiple processors that execute different instructions on different data streams independently.

- **Basic Concept**: Different instructions operate on different data simultaneously.

- **Execution**: Independent execution of instructions by each processor.
- **Common Use**: General-purpose parallel computing, complex simulations.
- **Examples**: IBM Blue Gene, multi-core processors in modern computers, networked parallel computer grids.



(c) MIMD architecture (with shared memory)

2) **With a neat diagram, describe the three shared memory multi processor models**

Shared Memory Multiprocessor Models

1. Uniform Memory Access (UMA) Model

- **Concept**:
  - All processors share a single, centralized memory.
  - Each processor has equal access time to all memory locations.
- **Architecture**:
  - Processors connected to a common bus or switch.
  - Can have private caches for faster access to frequently used data.
- **Use Cases**:
  - Suitable for general-purpose computing and time-sharing applications.
  - Effective for speeding up single large programs.

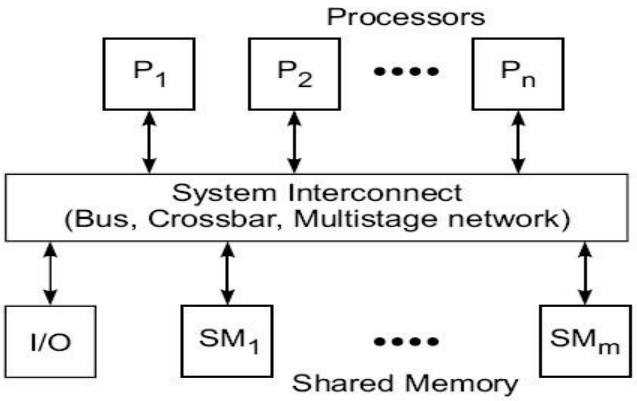


Fig. 1.6 The UMA multiprocessor model

2. Non-Uniform Memory Access (NUMA) Model

- **Concept**:
  - Memory is physically distributed among processors.
  - Access time varies based on whether data is local or remote.
- **Architecture**:
  - Each processor has its own local memory.
  - Interconnected via a network that adds delay to remote memory accesses.
- **Use Cases**:
  - Suitable for larger systems needing scalability.
  - Common in high-performance computing and servers.

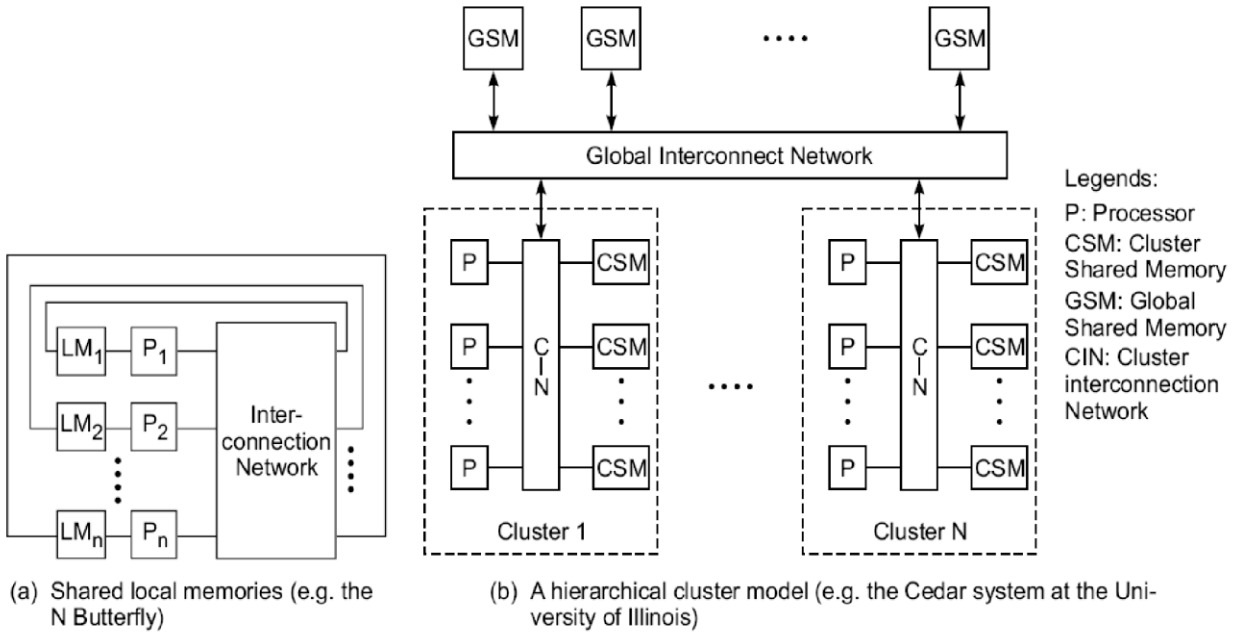


Fig. 1.7 Two NUMA models for multiprocessor systems

3. Cache-Only Memory Architecture (COMA) Model

- **Concept**:
  - Main memory is treated as a large shared cache.
  - Each processor accesses data through a cache-coherence protocol.
- **Architecture**:
  - No distinct hierarchy between cache and main memory.
  - Distributed cache directories assist in locating data blocks.
- **Use Cases**:
  - Useful for systems with unpredictable memory access patterns.
  - Found in some advanced research and supercomputing systems.

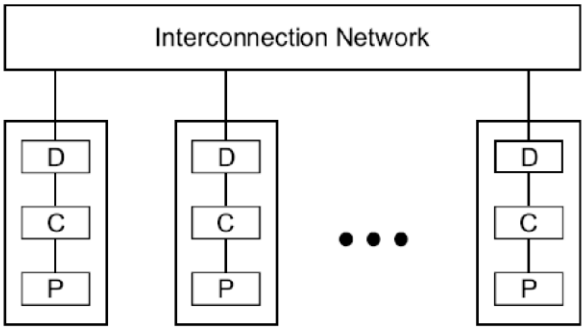


Fig. 1.8 The COMA model of a multiprocessor (P: Processor, C: Cache, D: Directory; e.g. the KSR-1)

- **UMA**: Simple, equal access time, centralized memory, suitable for general-purpose applications.
- **NUMA**: Variable access time, distributed memory, scalable for larger systems.
- **COMA**: Dynamic caching, no memory hierarchy, suitable for complex and unpredictable data access patterns.

3. With respect to dependencies between the instructions, discuss the following with example: (i) Data dependency (ii) Control dependency (iii) Resource dependency Or Explain Condition for Parallelism

1. Data Dependency

Definition: Data dependency occurs when the execution of an instruction depends on the result of a previous instruction. There are two main types of data dependencies:

- **Flow Dependency (RAW)**: Occurs when an instruction S2 depends on the output of instruction S1.

Example:

S1: Load R1, A ; R1 ← Memory(A)

S2: Add R2, R1 ; R2 ← R1 + R2

S2 has a RAW dependency on S1 because it uses the result (R1) produced by S1.

 Anti-dependency (WAR): Occurs when an instruction S2 writes to a register or memory location that is later read by instruction S1.

 Output Dependency (WAW): Occurs when multiple instructions write to the same register or memory location.

Example

S1: Load R1, A

S2: Add R2, R1

S3: Move R1, R3

S4: Store B, R1

Flow dependency exists from S1 to S2 (RAW), anti-dependency from S2 to S3 (WAR), and output dependency from S1 to S3 (WAW).

2. Control Dependency

Definition: Control dependency occurs due to conditional branches in code. It refers to the situation where the execution of an instruction depends on the outcome of a previous branch or jump instruction.

```
if (condition) {
    x = x + 1;   // Instruction A
} else {
    x = x - 1;   // Instruction B
}
y = x * 2;       // Instruction C
```

 Instructions A and B are control dependent because the execution of instruction C depends on which branch (if or else) was taken.

 The dependency exists because the value of x (and thus the result of instruction C) depends on the condition evaluated in the preceding branch.

3. Resource Dependency

Definition: Resource dependency occurs when two instructions compete for the same hardware resource, such as a functional unit, memory access, or a specific register.

S1: Load R1, A

S2: Add R2, R3

S3: Store B, R1

S3 depends on S1 for register R1, illustrating a resource dependency.

4. Illustrate the architecture of vector super computer with neat diagram

Vector Supercomputer Architecture

Vector supercomputers are designed to handle large-scale computations efficiently by exploiting parallelism inherent in vector operations. They typically consist of the following components:

1. Vector Processing Units

- **Vector Registers**: Specialized registers capable of holding arrays (vectors) of data elements.
- **Vector Functional Units**: Dedicated hardware units designed to perform operations on vector data.
- **Vector Pipelines**: Parallel execution pipelines optimized for vector instructions.

2. Memory Subsystem

- **Vector Memory**: Specialized memory subsystem optimized for high-bandwidth data access.
- **Vector Cache**: Cache memory designed to handle large blocks of data aligned with vector operations.

3. Interconnection Network

- **High-Speed Interconnect**: Efficient network connecting vector units, memory, and I/O devices.
- **Crossbar Switch or Multistage Network**: Used to ensure high-bandwidth data transfer among components.

4. Control Unit

- **Vector Control Processor**: Manages vector operations, scheduling, and synchronization.
- **Instruction Fetch and Decode Unit**: Fetches vector instructions and decodes them for execution.

5. I/O Subsystem

- **I/O Controllers**: Manage input and output operations to external devices.
- **High-Speed Interfaces**: Connect supercomputer to external networks and storage.

 Vector Control Processor: Manages the overall operation of the supercomputer, including instruction scheduling and synchronization of vector units.

 Vector Registers: Hold vector data elements used by the functional units for processing.

 Vector Functional Units: Execute vector instructions such as vector addition, multiplication, and other operations in parallel.

 Vector Memory Subsystem: Optimized for high-speed access to large data sets aligned with vector operations.

 Vector Cache: Stores frequently accessed vector data to reduce memory latency.

 Interconnection Network: Facilitates high-bandwidth communication between components to support parallel processing.

 I/O Subsystem: Handles input and output operations to connect the supercomputer with external devices and networks.

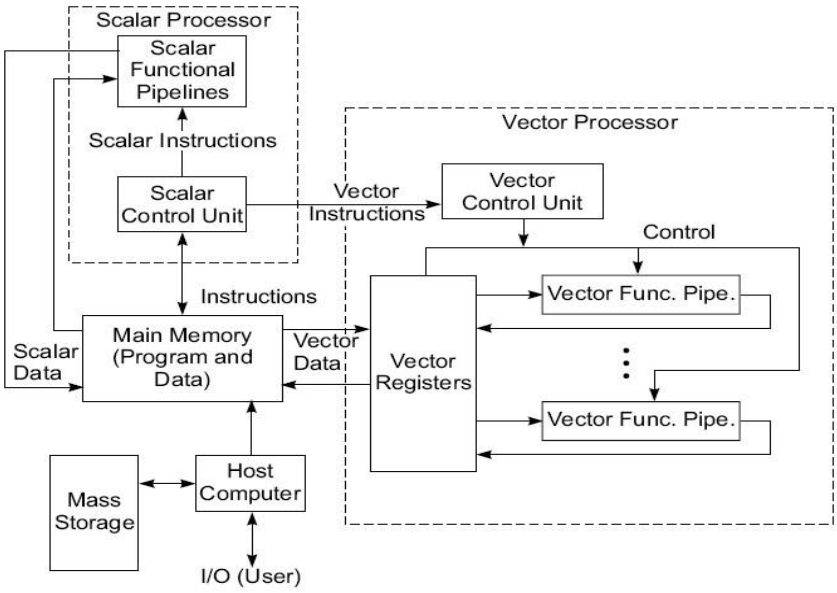


Fig. 1.11 The architecture of a vector supercomputer

5. Explain With neat diagram Elements of Modern Computers.

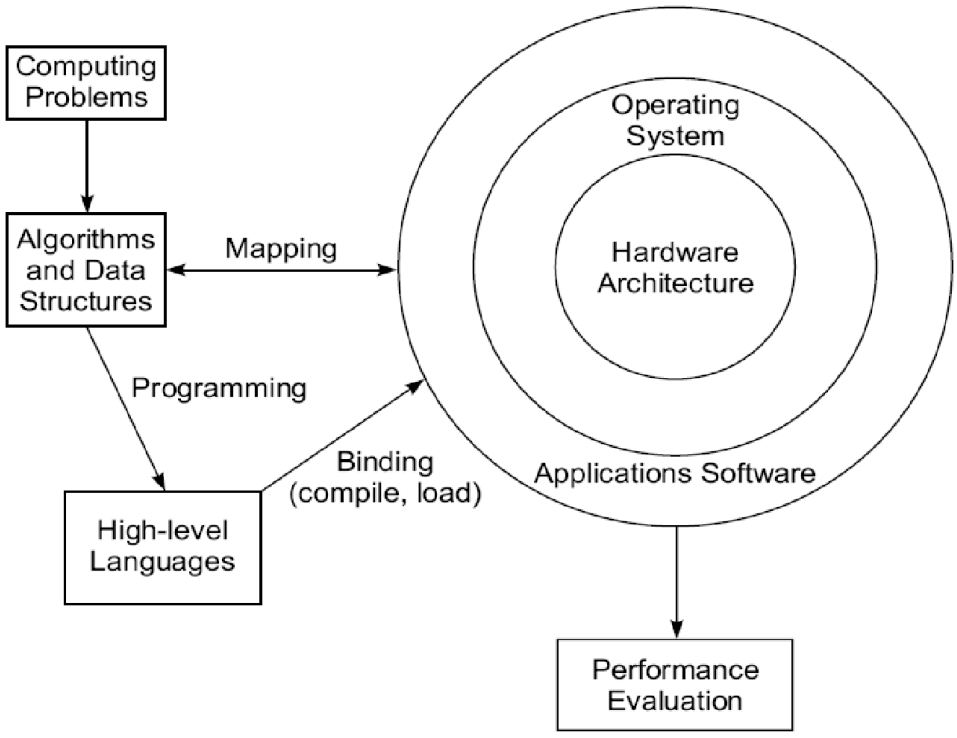


Fig. 1.1 Elements of a modern computer system

Computing Problems

Computing problems can broadly be categorized into numerical computing, transaction processing, and logical reasoning, each requiring different computing resources:

- **Numerical Computing**: Involves complex mathematical calculations and intensive integer or floating-point computations. Examples include scientific simulations and engineering analyses.
- **Transaction Processing**: Focuses on accurate and efficient handling of transactions, large database management, and information retrieval operations. Common in business and government sectors.
- **Logical Reasoning (AI Problems)**: Involves logic inferences, symbolic manipulations, and decision-making processes. Used in artificial intelligence applications for tasks like natural language processing and image recognition.

Hardware Resources

Modern computer systems are composed of:

- **Processors**: Execute instructions and perform calculations.
- **Memory**: Stores data and instructions temporarily.
- **Peripheral Devices**: Include input/output devices like terminals, scanners, printers, and networking equipment.
- **Interfaces**: Specialized hardware interfaces facilitate communication between peripherals and the main system.

Operating System

An operating system (OS) manages hardware resources and supports application software:

- **Resource Management**: Allocates and deallocates resources (CPU time, memory, etc.) for user programs.
- **Application Support**: Facilitates the development and execution of application software.
- **Performance Evaluation**: Benchmark programs are used to evaluate the performance of the system and applications.

System Software Support

System software supports efficient program development and execution:

- **High-Level Languages (HLL)**: Programs are written in languages like C++, Java, or Python.
- **Compiler**: Translates high-level code into machine-readable object code, optimizing it for efficient execution.
- **Assembler**: Converts object code into machine code understandable by the hardware.
- **Loader**: Initiates program execution through the OS kernel.

Compiler Support

Compiler technologies enable efficient program execution:

- **Preprocessor**: Uses a sequential compiler with a library to implement high-level parallel constructs.
- **Precompiler**: Analyzes program flow, checks dependencies, and performs limited optimizations for parallelism detection.
- **Parallelizing Compiler**: Automatically detects parallelism in source code and transforms sequential code into parallel constructs, enhancing performance on multi-core and vectorized architectures.

6) Explain Bernstain's Conditions for parallesim with example:

Bernstein's conditions are a set of criteria used to determine whether two processes (or instructions) can execute in parallel without interfering with each other. These conditions are based on the dependencies of data used and produced by the processes. The conditions are described as follows:

Notations:

- I_i: Input set of process P_i (variables read by P_i)
- O_i: Output set of process P_i (variables written by P_i)

For two processes P_1 and P_2 to execute in parallel, the following conditions must hold:

1. Flow Independence (Read-After-Write, RAW)
   - $O_1 \cap I_2 = \emptyset$
   - The output of P_1 does not overlap with the input of P_2. This ensures that P_2 does not read any value that P_1 writes, preventing P_2 from using stale data.

2. **Anti-Independence (Write-After-Read, WAR)**
   - $I_1 \cap O_2 = \emptyset$
   - The input of P_1 does not overlap with the output of P_2. This ensures that P_1 does not read any value that P_2 writes, preventing P_1 from using stale data.
3. **Output Independence (Write-After-Write, WAW)**
   - $O_1 \cap O_2 = \emptyset$
   - The outputs of P_1 and P_2 do not overlap. This ensures that P_1 and P_2 do not overwrite each other's results.

Illustration with Example

Consider the following simple example with two processes, P_1 and P_2:

- $P_1 : \mathrm{a} = \mathrm{b} + \mathrm{c};$
- $P_2 : \mathrm{d} = \mathrm{e} + \mathrm{f};$

Let's identify the input and output sets:

- For P_1: $I_1 = \{b, c\}$, $O_1 = \{a\}$
- For P_2: $I_2 = \{e, f\}$, $O_2 = \{d\}$

Checking Bernstein's conditions:

1. **Flow Independence**:
   - $O_1 \cap I_2 = \{a\} \cap \{e, f\} = \emptyset$ (Condition satisfied)
2. **Anti-Independence**:
   - $I_1 \cap O_2 = \{b, c\} \cap \{d\} = \emptyset$ (Condition satisfied)
3. **Output Independence**:
   - $O_1 \cap O_2 = \{a\} \cap \{d\} = \emptyset$ (Condition satisfied)

Since all three conditions are satisfied, P_1 and P_2 can be executed in parallel without any risk of data dependency issues.

7) Explain Evolution of Computer

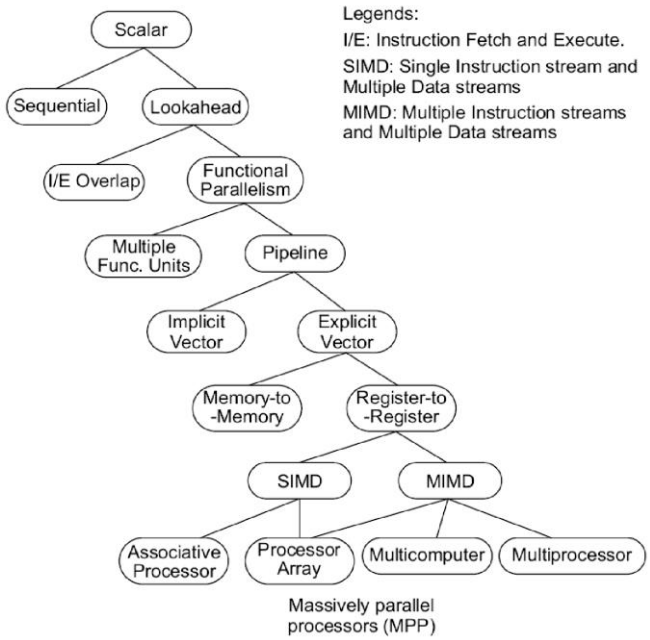


Fig. 1.2 Tree showing architectural evolution from sequential scalar computers to vector processors and parallel computers

- The study of computer architecture involves both hardware organization and programming/software requirements.
- As seen by an assembly language programmer, computer architecture is abstracted by its instruction set, which includes opcode (operation codes), addressing modes, registers, virtual memory, etc.
- From the hardware implementation point of view, the abstract machine is organized with CPUs, caches, buses, microcode, pipelines, physical memory, etc.
- Therefore, the study of architecture covers both instruction-set architectures and machine implementation organizations.

Lookahead, Parallelism, and Pipelining

Lookahead techniques were introduced to prefetch instructions in order to overlap I/E (instruction fetch/decode and execution) operations and to enable functional parallelism. Functional parallelism was supported by two approaches:

1. using multiple functional units simultaneously,
2. to practice pipelining at various processing levels.

8) **Explain with diagram operational model of SIMD super computer**

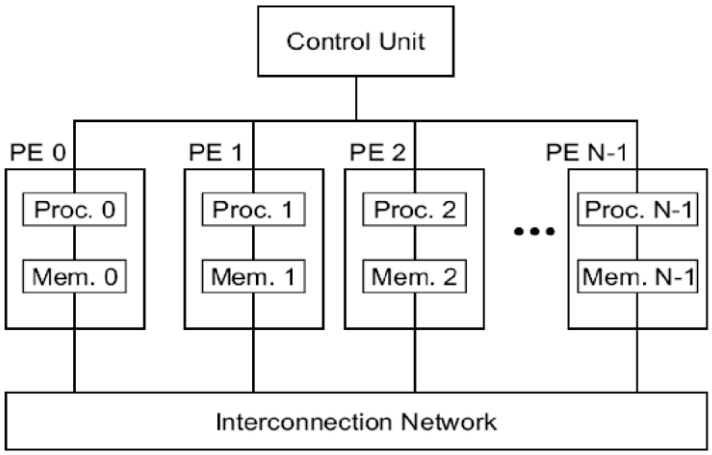


Fig. 1.12 Operational model of SIMD computers

SIMD computers have a single instruction stream over multiple data streams.

An operational model of an SIMD computer is specified by a 5-tuple:

M = (N,C, I,M, R) where

1. ***N*** is the number of *processing elements* (PEs) in the machine. For example, the Illiac IV had 64 PEs and the Connection Machine CM-2 had 65,536 PEs.
2. ***C*** is the set of instructions directly executed by the *control unit* (CU), including scalar and program flow control instructions.
3. **I** s the set of instructions broadcast by the CU to all PEs for parallel execution. These include arithmetic, logic, data routing, masking, and other local operations executed by each active PE over data within that PE.
4. **M** is the set of masking schemes, where each mask partitions the set of PEs into enabled and disabled subsets.

5. **R** is the set of data-routing functions, specifying various patterns to be set up in the interconnection network for inter-PE communications.

9)Explain with diagram operational tagged token data flow computer

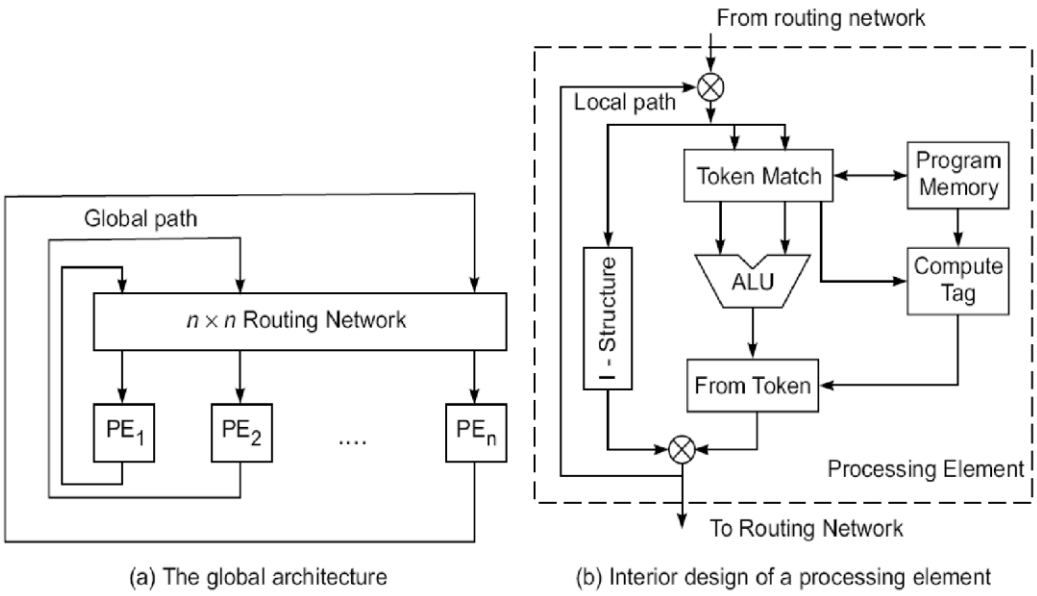


Fig. 2.12 The MIT tagged-token dataflow computer (adapted from Arvind and Iannucci, 1986 with permission)

Tagged Token Data Flow Computers represent a class of parallel computers that execute instructions based on the availability of data rather than a predetermined sequence. This architecture contrasts with traditional von Neumann architectures, which execute instructions in a fixed order. The data flow model is especially suitable for applications with irregular data dependencies and inherent parallelism.

- **Data Flow Graph (DFG):**
  - Represents computations as nodes (instructions) and edges (data dependencies).
  - Nodes are executed when all required data tokens are available on their input edges.
- **Tokens:**
  - Carry data values and tags.
  - Tags indicate the context or position within the data flow graph.
- **Matching Unit:**
  - Matches tokens with the same tag to enable the execution of the corresponding node.
- **Processing Elements (PEs):**
  - Execute instructions once the required tokens are matched.

- Each PE can handle multiple instructions in parallel.

Token Creation:

- Data tokens are generated when an instruction produces an output.
- Tokens include data values and tags indicating their position in the data flow graph.

Token Matching:

- Tokens are sent to the matching unit.
- The matching unit checks for tokens with matching tags (representing that all required inputs for a node are available).

Instruction Execution:

- Matched tokens are sent to an available processing element.
- The processing element executes the corresponding instruction.

Result Generation:

- The processing element generates new tokens based on the instruction's output.
- These tokens are sent back to the matching unit or token queue, continuing the cycle.