Computer Architecture Cheatsheet

Module 16 – System

System Overview

Components Beyond CPU and Memory:

- **Input/Output (I/O) Devices:** Interfaces for data exchange between the computer and the external environment.
- **Secondary/Tertiary Storage:** Flash drives, HDDs, SSDs, tapes.
- **Network Interfaces:** Ethernet, WiFi, Bluetooth, LTE.
- **Human-Machine Interfaces (HMIs):** Keyboards, mice, touchscreens, displays, audio systems.
- **Printers and Sensors:** Various types including 3D printers, GPS sensors, heart rate monitors.
- **Actuators:** Valves, robotic arms, automotive brakes.

I/O Communication Methods

Direct Channels/Register-Based I/O:

- Use specific registers or channels for communication.
- Common in microcontrollers and simple embedded systems.

Memory-Mapped I/O (MMIO):

- Reserve a portion of the address space for I/O devices.
- I/O operations are performed using standard memory instructions (load/store).
- Typically "uncached" memory addresses (using page tables).
- Virtual Address considerations.

Communication Technologies and Protocols

- **Time-Multiplexed, Shared, and Slow:** PCI, IDE.
- **Shared but Faster:** Ethernet, SATA.
- **Dedicated Channels:** PCIe.
- **Handshaking Protocols:** Asynchronous communication using arbiter, initiator, and target.
- **Fixed Timing Protocols:** Synchronous communication with predefined timing.

Direct Memory Access (DMA)

Overview:

- DMA allows peripherals to read/write memory without CPU intervention.
- Offloads data transfer tasks from the CPU, improving performance.
- Typically managed by a DMA controller or engine.

DMA Operation:

1. CPU initializes DMA transfer by setting up DMA registers.
2. DMA controller handles the data transfer between I/O device and memory.
3. Upon completion, DMA controller sends an interrupt to notify the CPU.

Benefits and Drawbacks: Benefits:

- Reduces CPU overhead for data transfers.
- Enables simultaneous data transfers and CPU processing.

Drawbacks:

- Complexity in managing multiple DMA channels.
- Potential for bus contention and performance bottlenecks.

Network-on-Chip (NoC)

Overview:

- NoC is an on-chip communication subsystem that connects multiple cores, memory controllers, GPUs, and I/O devices.
- Facilitates efficient data exchange and scalability in multicore processors.

Design Challenges:

- **Performance Optimization:** Ensuring low latency and high throughput.
- **Scalability:** Supporting increasing numbers of cores and devices.
- **Energy Efficiency:** Minimizing power consumption.
- **Security:** Protecting against data breaches and unauthorized access.
- **Integration with Emerging Paradigms:** Adapting to new computing models and technologies.

Design Ingredients:

- **Topology:** Network structure (mesh, torus, star, etc.).
- **Routing Logic:** Algorithms for data packet traversal.
- **Router Design:** Handling data packets, buffering, and flow control.
- **Bandwidth and Latency:** Ensuring sufficient data transfer rates and minimal delays.

Interrupts, Exceptions, and Traps

Definitions:

- **Exception:** An unusual condition occurring at runtime associated with an instruction (e.g., division by zero).
- **Trap:** A synchronous transfer of control to a handler due to an exceptional condition within a thread.
- **Interrupt:** An external event that occurs asynchronously to the current thread (e.g., I/O events).

Handling Mechanisms: Interrupt Handling:

- **Interrupt Exception:** Occurs to handle the interrupt.
- **Trap Execution:** Transfers control to the interrupt handler.
- **Priority Management:** Determines when to trap based on interrupt priority.

Exception Handling:

- **Precise Exceptions:** Ensure all states are maintained for accurate recovery.
- **Examples:** Branch mispredictions, unknown instructions, hardware failures.

Trap Handling:

- **System Calls:** Transition to OS handlers with specific service parameters.
- **Handler Tables:** Use tables to jump to appropriate subroutines based on trap parameters.

Interrupt vs. Polling vs. Hybrid

Interrupt:

- **Pros:** Low CPU overhead until event occurs.
- **Cons:** Can occur unpredictably, causing context-switch overhead.

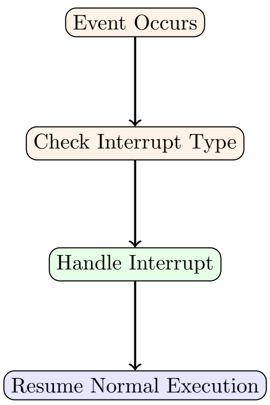
Polling:

- **Pros:** Can be fully scheduled, predictable.
- **Cons:** High CPU overhead, difficult to implement efficiently.

Hybrid Approach:

- Use interrupts initially, switch to polling after the first event, and revert to interrupts after a certain period.

Interrupt Handling Flowchart



Module 15 – Memory Consistency

Consistency Problem Example

Initially: $M[Z] = M[Y] = 0,\ x3 = 100,\ x4 = 200$

Core1: lw x1, Z(x0)
...
sw Y, x3(x0)

Core2: lw x2, Y(x0)
...
sw Z, x4(x0)

Potential Values for $x1$ and $x2$: Depending on the order of memory operations, $x1$ and $x2$ can take values 0 or 200 and 0 or 100, respectively.

Memory Consistency vs. Coherency

- **Consistency:** Ordering of parallel accesses between different addresses.
- **Coherency:** Ordering of parallel accesses to the same address.

Memory Model

Definition: Interactions between threads and the shared memory. In other words, what are the possible values for a load?

Sequential Consistency (SC)

1. Processors see their own loads and stores in program order.
2. Processors see others' loads and stores in program order.
3. All processors see the same global load/store ordering.

Characteristics:

- Simplest memory model.
- Makes multicore systems indistinguishable from multi-programmed in-order uniprocessors.
- Too slow due to strict ordering constraints.

Memory Operation Ordering

Four Types of Memory Operation Orderings:

- **W→R:** Write must complete before subsequent read.
- **R→R:** Read must complete before subsequent read.
- **R→W:** Read must complete before subsequent write.
- **W→W:** Write must complete before subsequent write.

Sequential Consistency: Maintains all four orderings.
Relaxed Memory Consistency Models: Allow certain orderings to be violated for performance gains.

Total Store Order (TSO) Model

Definition: A relaxed memory consistency model where processors can reorder certain memory operations to improve performance.

Key Points:

- Processors can move their own reads in front of their own writes.
- Read by other processors cannot return the new value of a write until the write is visible to all processors.

Example Scenario:

$a = 0,\ flag = 0$

Thread 1:

store $1 \to a$

store $1 \to flag$

Thread 2:

loop: if (flag == 0) goto loop

load a

Question: Can the load in Thread 2 read 0?

TSO vs. SC:

Initially: $M[Z] = M[Y] = 0$

Core1: lw x1, Z

Core2: lw x2, Y

...

Core1: sw Y, 100

Core2: sw Z, 200

Potential Values for $x1$ and $x2$: Depending on the visibility and ordering of writes, $x1$ and $x2$ can take 0 or the updated values.

Weak Consistency

- Relaxed memory ordering allows certain memory operation orderings to be violated.
- Performance vs. complexity trade-off.
- Different architectures use different weak consistency models.

RISC-V Weak Memory Ordering (RVWMO)

- Loads are optimistically fired to memory on arrival to the Load-Store Unit (LSU) for performance.
- Load instructions compare their address with all dependent store addresses:
  - If a match is found, the memory request is either:
    - **Killed:** If the store data is not present, the load goes to sleep and retries later.
    - **Forwarded:** If the store data is present, forward the data to the load and mark it as succeeded.
- **Behavior:**
  - **Write → Read Constraint:** Relaxes the ordering, allowing newer loads to execute before older stores.
  - **Read → Read Constraint:** Maintains ordering; loads to the same address appear in order.
  - **Own Writes Early:** A thread can read its own writes early.

Writing Correct Programs

Approaches:

- **Race-Free Programming:** Ensure that there are no data races in the program.
- **Synchronization Primitives:** Use fences, mutexes, semaphores, etc., to enforce ordering and visibility.

Synchronization Mechanisms

Fences:

- **Definition:** An instruction that enforces all memory operations before it to complete before the fence is executed. All memory operations after the fence must wait until the fence is completed.
- **Usage in RISC-V:**

```
fence rwio
fence r, r
```

Semaphores and Barriers: Semaphores:

- Used to synchronize access to shared resources.
- Implemented using flags or counters.
- Typically implemented in software.

Barriers:

- Synchronize all processors at a certain point in the program.
- Implemented using hardware flags or software synchronization.

Mutexes and Locks:

- **Mutual Exclusiveness:** Ensures only one thread accesses a shared resource at a time.
- **Implementation:**

```
acquire(lock) {
    while (lock != 0) { /* busy
       wait */ }
    lock = 1; // Acquire the lock
}

release(lock) {
    lock = 0; // Release the lock
}
```

Atomic Instructions:

- Hardware primitives that guarantee instructions are executed in-order and without interruption.
- **Examples:**
  - **Test-and-Set:**

```
TS(int x) {
    oldval = SWAP(x, 1);
       // Atomic swap
    return oldval;
}
```

  - **Compare-and-Swap (CAS)**
  - **Fetch-and-Add**

Load-Reserved and Store-Conditional (LR/SC):

- **Load-Reserved:** Processor remembers the address.
- **Store-Conditional:** Attempts to store only if no write has occurred to the address since the last load-reserved.
- **Example in RISC-V:**

```
    loop:
        lr.w x2, 0(x1)
        addi x2, x0, 1
        sc.w x2, 0(x1)
        bnez x2, loop // Retry if
            store-conditional failed
```

- **MESI Protocol Integration:**
  - **Load-Reserved:** Ensures the cache line is in Exclusive or Modified state.
  - **Store-Conditional:** Succeeds only if the cache line remains in Exclusive or Modified state.

Synchronization Example: Producer-Consumer Model

Example Code:

```
thread1:
    sw Y, 100
    fence
    sw X, 1

thread2:
    lw X, 0
    if (X == 1) {
        lw Y, 0
    }
```

Explanation:

- **Thread1:** Stores to Y and then to X with a memory fence to ensure ordering.
- **Thread2:** Loads X first and, if X is 1, then loads Y, ensuring Y is updated before X.

Optimizations in Synchronization

Test-Test-and-Set (TTAS):

```
TTS(int x) {
    oldval = x; // Read the lock
    if (oldval == 0) {
        oldval = SWAP(x, 1); // Attempt to
            acquire lock
    }
    return oldval;
}

// Usage
while (TTS(lock) == 1) { /* busy wait */ }
```

Benefits:

- Reduces the number of atomic operations by first testing the lock before attempting to set it.

Other Optimizations:

- **Queue-Based Locks:** Ensures fairness by queuing lock acquisition requests.
- **Multiple/Parallel Locks:** Allows multiple locks to be acquired concurrently for different resources.

Final Recap and Bottom-Line

Memory Consistency Models:

- Range from Sequential Consistency (strong) to Weak Consistency (relaxed).
- Balance between performance and correctness.

Synchronization:

- Essential for ensuring correctness in parallel programs.
- Achieved using fences, mutexes, semaphores, and atomic operations.

Cache Coherency:

- Critical for maintaining consistency across multicore processors.
- Implemented using protocols like MESI, MOESI, and MOESIF.

TLB and Address Translation:

- TLBs speed up virtual to physical address translation.
- Critical for performance in systems with virtual memory.

I/O and DMA:

- Efficient I/O handling is crucial for overall system performance.
- DMA offloads data transfer tasks from the CPU, enhancing efficiency.

Network-on-Chip (NoC):

- Facilitates efficient communication in multicore systems.
- Essential for scalability and performance in modern processors.

Module 14 – Cache Coherency

Cache Coherency Basics

Definition: Uniformity of shared resource data that ends up stored in multiple local caches.

Coherency vs. Consistency

- **Coherency:** Ensures that multiple caches reflect the same value for a shared data item.
- **Consistency:** Refers to the order in which memory operations appear to execute across multiple processors.

Cache Coherence Problem

Example Scenario:

$$M[Z] = 0 \quad \text{(Z is an address, e.g., } 0x200\text{)}, \; x5 = 10$$

Core1: lw x1, 0(Z)

...

Core2: lw x3, 0(Z)

...

Core1: sw x5, 0(Z)

Core2: lw x6, 0(Z)

Problem: Ensuring that Core2's load after Core1's store gets the updated value.

Coherence Protocols

Snooping Protocols:

- **Definition:** Caches monitor (snoop) the shared bus for memory operations to maintain coherence.
- **Write Miss:** The address is invalidated in all other caches before the write is performed.
- **Read Miss:** If a dirty copy is found in another cache, a write-back is performed before the memory is read. Otherwise, read from memory.

Coherence States

Basic States (MI Protocol):

- **Modified (M):** The cache line is dirty and has the only valid copy.
- **Invalid (I):** The cache line is invalid.

Enhanced States:

- **Shared (S):** The cache line is clean and can be shared among multiple caches.
- **Exclusive (E):** The cache line is clean and is only present in one cache.
- **Owned (O):** The cache line is dirty but shared; one cache owns the dirty copy.
- **Forwarding (F):** The cache line is shared and one cache acts as a forwarder.

MSI Coherence Protocol

States:

- **Modified (M)**
- **Shared (S)**
- **Invalid (I)**

State Transitions:

- **Read Miss:** Transition from I to S or E.
- **Write Miss:** Transition from I to M.
- **Write to S:** Transition from S to M (with invalidation).

MESI Coherence Protocol

States:

- **Modified (M)**
- **Exclusive (E)**
- **Shared (S)**
- **Invalid (I)**

Enhancements:

- **Exclusive (E):** Allows silent upgrades to M without broadcasting.
- **Transition Rules:**
  - **I → E:** On read miss if no other cache has the line.
  - **E → M:** On write, silently upgrade to M.
  - **E → S:** If another cache requests the line, transition to S.

MOSI Coherence Protocol

States:

- **Modified (M)**
- **Owned (O)**
- **Shared (S)**
- **Invalid (I)**

Enhancements:

- **Owned (O):** Allows one cache to own the dirty copy while others have shared clean copies.
- **State Transitions:**
  - **M → O:** On read request from another cache.
  - **O → I:** On write request from another cache.

MOESI Coherence Protocol

States:

- **Modified (M)**
- **Owned (O)**
- **Exclusive (E)**
- **Shared (S)**
- **Invalid (I)**

Enhancements:

- **Exclusive (E) and Owned (O) States:** Combine benefits of E and O.
- **State Transitions:**
  - **I → E:** On read miss with no other copies.
  - **E → M:** On write.
  - **E → S:** On read by another cache.
  - **M → O:** On read by another cache.
  - **O → I:** On write by another cache.

MOESIF Coherence Protocol

States:

- **Modified (M)**
- **Owned (O)**
- **Exclusive (E)**
- **Shared (S)**
- **Invalid (I)**
- **Forwarder (F)**

Enhancements:

- **Forwarder (F):** Acts as a source for data to other caches, reducing the need for multiple cache-to-cache transfers.
- **State Transitions:**
  - **S → F:** When a cache acts as a forwarder.
  - **F → I:** When another cache modifies the data.

False Sharing

Definition: Occurs when multiple processors cache the same cache line containing different variables, leading to unnecessary invalidations.

Impact: Causes coherence misses and reduces cache efficiency.

Directory-Based Coherence Protocol

- **Definition:** Uses a centralized directory to keep track of which caches have copies of each cache line.
- **Advantages:** Scales better than snooping protocols, reduces bus traffic.
- **Operation:**
  - On a cache miss, the directory is consulted to determine which caches have the line.
  - Coherence actions are directed only to those caches, avoiding unnecessary broadcasts.

Coherence Example

$M[Z] = 0$ (Z is an address, e.g., $0x200$), $x5 = 10$

Core1: lw x1, 0(Z)

...

Core2: lw x3, 0(Z)

...

Core1: sw x5, 0(Z)

Core2: lw x6, 0(Z)

Goal: Ensure Core2's load after Core1's store gets the updated value.

Module 15 – Memory Consistency (Continued)

Consistency Problems

- **Reordering Issues:** SW and LW (to different addresses) within one core can be reordered, disrupting assumptions in other cores.
- **Timing Issues:** The timing between cores might be off, even without reordering.

Solutions to Consistency Problems

Synchronization Strategies:

- **Fences:** Enforce ordering constraints.
- **Mutexes/Locks:** Ensure mutual exclusiveness for shared resources.
- **Semaphores and Barriers:** Coordinate actions among multiple threads or cores.

Synchronization Using Semaphores

Example:

Core1: lw x1, Z

...

Core1: sw Y, 100

Core2: lw x3, Y

...

Core2: sw Z, 200

Implementation in Hardware:

- Use one-hot encoding to set a flag when a core reaches the barrier.
- Wait until all flags are set.

Mutual Exclusiveness

Definition: Ensures only one process accesses a shared region at any given time.

Implementation Using Mutexes/Locks:

```
acquire(lock) {
    while (lock != 0) { /* busy wait */ }
    lock = 1; // Acquire the lock
}

release(lock) {
    lock = 0; // Release the lock
}
```

Assembly Version:

```
loop:
    lw ra, 0(x1)
    bnez ra, loop
    addi ra, x0, 1
    sw ra, 0(x1)

release:
    sw x0, 0(x1)
```

Atomic Instructions:

- **Test-and-Set (TS):**

```
TS(int x) {
    oldval = SWAP(x, 1); // Atomic
        swap
    return oldval;
}
```

- **Load-Reserved and Store-Conditional (LR/SC):**

```
loop:
    lr.w x2, 0(x1)
    addi x2, x0, 1
    sc.w x2, 0(x1)
    bnez x2, loop // Retry if
        store-conditional failed
```

Release Consistency

- Guarantees that all reads and writes within acquire/release blocks are completed and visible to all cores upon releasing the lock.
- Allows for reordering of memory operations outside synchronization blocks.

Recap

- **Locks:** Test-and-set, Atomic operations, LR/SC.
- **Optimizations:** Test-Test-and-Set (TTAS), Queue-based locks, Multiple/Parallel Locks.
- **Synchronization:** Essential for maintaining consistency and correctness in multicore systems.

Module 14 – Cache Coherency (Continued)

False Sharing

- **Definition:** Occurs when multiple processors cache the same cache line containing different variables, leading to unnecessary invalidations.

- **Impact:** Causes coherence misses and reduces cache efficiency.

Optimizing MSI Coherence Protocol

- **Issue:** Unnecessary broadcasts and write-backs when a core with an exclusive cache line wants to write.
- **Solution:** Introduce the **Exclusive (E)** state to allow silent upgrades from S to M without broadcasting.

Coherence States and Transitions

MSI Protocol:

- **States:** Modified (M), Shared (S), Invalid (I).
- **Transitions:**
  - **Read Miss:** I → S or E.
  - **Write Miss:** I → M.
  - **Write to S:** S → M (with invalidation).

MESI Protocol:

- **States:** Modified (M), Exclusive (E), Shared (S), Invalid (I).
- **Enhancements:** Allows silent upgrades from E to M, transitions between E and S.

MOSI Protocol:

- **States:** Modified (M), Owned (O), Shared (S), Invalid (I).
- **Enhancements:** Owned state allows a cache to supply data to others without write-back.

MOESI Protocol:

- **States:** Modified (M), Owned (O), Exclusive (E), Shared (S), Invalid (I).
- **Enhancements:** Combines Exclusive and Owned states for better efficiency.

MOESIF Protocol:

- **States:** Modified (M), Owned (O), Exclusive (E), Shared (S), Invalid (I), Forwarder (F).
- **Enhancements:** Forwarder state allows one cache to supply data to multiple caches, reducing cache-to-cache transfers.

Optimizing MSI and MESI

- **Silently Upgrade to M from E:** Avoid unnecessary broadcasts.
- **Forwarding (F) State in MOESIF:** Reduces cache-to-cache traffic by designating a forwarder.
- **Directory-Based Coherence:** Scales better by using a centralized directory to track cache line states.

Coherence Example with MOESI

$$\begin{aligned}
&\text{Mem}[Z] = 0,\ Z = \text{address } 0x200,\ x5 = 10\\
&\text{Core1: lw x1, 0(Z)}\\
&\ldots\\
&\text{Core2: lw x3, 0(Z)}\\
&\ldots\\
&\text{Core1: sw x5, 0(Z)}\\
&\text{Core2: lw x6, 0(Z)}
\end{aligned}$$

Goal: Ensure Core2's load after Core1's store gets the updated value using the MOESI protocol.

Directory-Based Coherence Protocol

- **Central Directory:** Keeps track of which caches have copies of each cache line.
- **Operation:**
  - On a cache miss, consult the directory to determine which caches have the line.
  - Send coherence messages only to those caches, avoiding unnecessary broadcasts.
- **Advantages:** Scales better than snooping protocols, reduces bus traffic.

Module 14 – Cache Coherency (Continued)

Cache Coherency States and Transitions

States:

- **Shared (S):** Multiple caches have the line in a clean state.
- **Modified (M):** Only one cache has the line in a dirty state.
- **Exclusive (E):** Only one cache has the line, and it is clean.
- **Owned (O):** One cache has the dirty line, and others may have shared clean copies.
- **Invalid (I):** The cache line is invalid.
- **Forwarder (F):** A cache acts as the forwarder for the line.

State Transitions:

- **Read Miss:** I → S or E.
- **Write Miss:** I → M.
- **Read Hit:** S, E, O → Remain or transition based on access.
- **Write Hit:** S, E, O → M.

False Sharing

Definition: Occurs when multiple processors cache the same cache line containing different variables, leading to unnecessary invalidations.

Impact: Causes coherence misses and reduces cache efficiency.

Directory-Based Coherence Protocol

- **Central Directory:** Maintains a record of which caches have copies of each cache line.
- **Operation:**
  - On a cache miss, the requesting cache queries the directory.
  - The directory responds with the list of caches that have the line.
  - Coherence actions (e.g., invalidations, data forwarding) are directed only to those caches.
- **Advantages:**
  - Scales better than snooping protocols.
  - Reduces unnecessary bus traffic.

Coherence Protocol Example: MOESIF

States:

- **Modified (M)**
- **Owned (O)**
- **Exclusive (E)**
- **Shared (S)**
- **Invalid (I)**
- **Forwarder (F)**

State Transitions:

- **I → E:** On a read miss if no other cache has the line.
- **E → M:** On a write, silently upgrade to M.
- **E → S:** On a read by another cache, transition to S.
- **M → O:** On a read by another cache, transition to O.
- **O → I:** On a write by another cache, transition to I after writing back.
- **S → F:** When acting as a forwarder.

Benefits:

- Reduces unnecessary data traffic.
- Improves cache line utilization.
- Enhances performance by minimizing cache-to-cache transfers.

Cache Coherency Example with MOESIF

$$\begin{aligned}
&M[Z] = 0,\ Z = \text{address } 0x200,\ x5 = 10\\
&\text{Core1: lw x1, 0(Z)} \quad \text{(S)}\\
&\dots\\
&\text{Core2: lw x3, 0(Z)} \quad \text{(S)}\\
&\dots\\
&\text{Core1: sw x5, 0(Z)} \quad \text{(M)}\\
&\text{Core2: lw x6, 0(Z)} \quad \text{(F)}
\end{aligned}$$

Goal: Ensure Core2's load after Core1's store gets the updated value efficiently using MOESIF protocol.

Key Metrics and Formulas

Cache Metrics

Average Memory Access Time (AMAT):

$$\text{AMAT} = \text{Hit Time} + (\text{Miss Rate} \times \text{Miss Penalty})$$

Cache Access Time:

$$\text{Total Access Time} = \text{L1 Access Time} + \text{L2 Access Time} + \text{L3 Acc}$$

Performance Metrics

Clock Cycles per Instruction (CPI):

$$\text{CPI} = \frac{\text{Total Clock Cycles}}{\text{Total Instructions}}$$

Execution Time:

$$\text{Execution Time} = \text{CPI} \times \text{Clock Period} \times \text{Number of Instructions}$$

Address Translation Metrics

Page Offset Bits:

$$\text{Page Offset Bits} = \log_2(\text{Page Size})$$

VPN Bits:

$$\text{VPN Bits} = \text{Virtual Address Size} - \text{Page Offset Bits}$$

PPN Calculation:

$$\text{Physical Page Number (PPN)} = \left\lfloor \frac{\text{Physical Address}}{\text{Page Size}} \right\rfloor$$

Tables and Figures

Cache Performance Metrics

Cache Level	Hit Time (ns)	Miss Penalty (ns)
L1	1	10
L2	5	50
L3	10	100
Main Memory	100	-

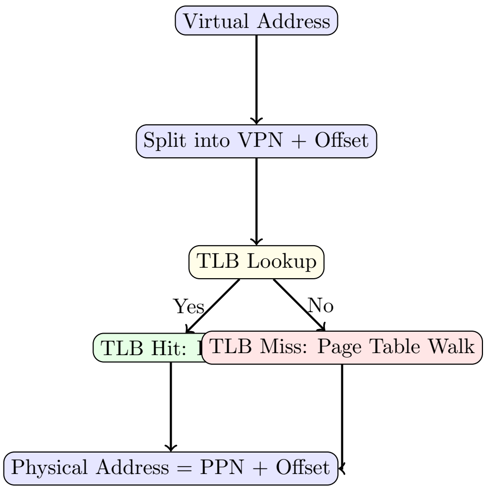
Table 1: Cache Performance Metrics

Cache Mapping Schemes Comparison

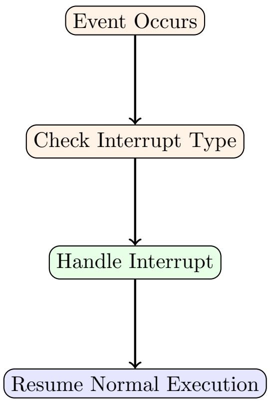
Mapping Scheme	Flexibility	Complexity	Speed
Direct-Mapped	Low	Low	Fast
Set-Associative	Medium	Medium	Moderate
Fully Associative	High	High	Slow

Table 2: Comparison of Cache Mapping Schemes

TLB Workflow Diagram



Interrupt Handling Flowchart



Multi-Core Systems

Introduction to Multi-Core Systems

- Leverage multiple cores to improve performance.
- Design choices include how to share memory, cache hierarchies, and interconnects.
- Challenges include cache coherency, memory consistency, and efficient inter-core communication.

SMT vs. Multitasking

Simultaneous Multithreading (SMT):

- Execute multiple threads on the same pipeline.
- Utilizes pipeline resources more efficiently.
- Each thread requires its own:
  - Program Counter (PC)
  - Register file
  - Logic for virtual address translation
  - Exception handling mechanism

Multitasking:

- Running different processes/tasks on the same core.
- Managed by the operating system through thread scheduling and context switching.
- Creates the illusion of concurrent execution on single-core systems.

Multicore Design Choices

Shared vs. Private Caches:

- **Shared Caches:** Multiple cores share a common cache level (e.g., L3).
- **Private Caches:** Each core has its own dedicated cache levels (e.g., L1, L2).
- **Hybrid Approach:** Combines shared and private caches to balance performance and scalability.

Interconnects:

- Network-on-Chip (NoC) used to connect multiple cores and shared resources.
- Ensures efficient data transfer and communication between cores.

Cache Coherency in Multicore Systems

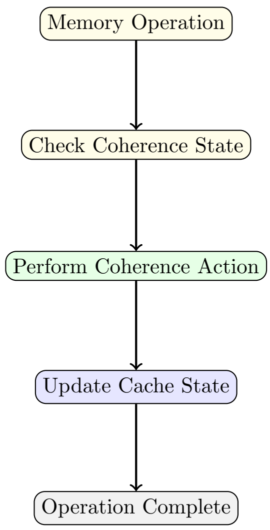
- Ensures that all caches reflect the most recent write to any shared data.
- Implemented using coherence protocols like MESI, MOESI, and MOESIF.
- Can use snooping or directory-based approaches.

Cache Coherency Example

$$
\begin{aligned}
&\text{Mem}[Z] = 0,\ Z = \text{address } 0x200,\ x5 = 10 \\
&\text{Core1: lw x1, 0(Z)} \quad \text{(S)} \\
&\dots \\
&\text{Core2: lw x3, 0(Z)} \quad \text{(S)} \\
&\dots \\
&\text{Core1: sw x5, 0(Z)} \quad \text{(M)} \\
&\text{Core2: lw x6, 0(Z)} \quad \text{(F)}
\end{aligned}
$$

Goal: Ensure Core2's load after Core1's store gets the updated value efficiently using the MOESIF protocol.

Cache Coherence Flowchart



Additional Tables

Coherence Protocols Overview

Protocol	States	Key Feature	Benefit
MSI	M, S, I	Simple state model	Low complexity
MESI	M, E, S, I	Adds Exclusive state	Reduces invalidations
MOSI	M, O, S, I	Adds Owned state	Efficient cache-to-cache transfers
MOESI	M, O, E, S, I	Combines Exclusive and Owned	Enhanced performance
MOESIF	M, O, E, S, I, F	Adds Forwarder state	Reduces cache-to-cache traffic

Table 3: Overview of Coherence Protocols