ECE 251: Computer Architecture

Week 13: Memory Hierarchies (Part 1)

Prof Rob Marano

Spring 2026

Chapter 4 Retrospective: Architecture Evolution

- **Single-Cycle Datapath:** One instruction per clock cycle. Cycle time is dictated by the longest path (Load Word). Simple control, but poor clock frequency.
- **Multi-Cycle Datapath:** Broke instructions into 5 steps (Fetch, Decode, Execute, Memory, Write-Back). Faster clock, but complex FSM control.
- **Pipelined Processor:** Overlapped the 5 execution stages to achieve an ideal CPI of 1.0. Massive throughput gains but introduced structural, data, and control hazards.
- **Hazard Resolution:** Implemented Forwarding Units, Hazard Detection Units (stalls/bubbles), and Branch Prediction (flushing).
- **Exceptions & Interrupts:** Added EPC and Cause registers, expanding the datapath to handle unpredictable external/internal events.

Agenda

1. 1. Introduction to Memory Hierarchy
2. 2. Memory Technologies
3. 3. The Basics of Caches
4. 4. SystemVerilog Hardware Implementations
5. 5. Worked Textbook Examples

1. The Journey: From Processor to Memory

- **Where we've been:** We built the logic gates, the ALU, the single-cycle datapath, and finally the pipelined processor with hazard detection.
- **The Final Frontier:** A fast CPU is useless if it has no data. We must now design the Memory System.
- **The Historical Problem:** Early computers had simple, flat memory. Over time, processors became exponentially faster, but memory speeds lagged. CPUs were starving for data!
- **The Solution:** Architects invented the **Memory Hierarchy** (Caches, Main Memory, Virtual Memory) to bridge this chasm.

1. The Processor/Memory Paradox

- Computer architects face a fundamental physical paradox:
- **Processors are incredibly fast** (Clock cycles in nanoseconds or picoseconds).
- **Memory is relatively slow** (Access times in tens of nanoseconds or milliseconds).
- If the CPU has to wait for Main Memory on every instruction fetch and data load, the massive speed gains of pipelining ($CPI \approx 1.0$) are entirely lost!

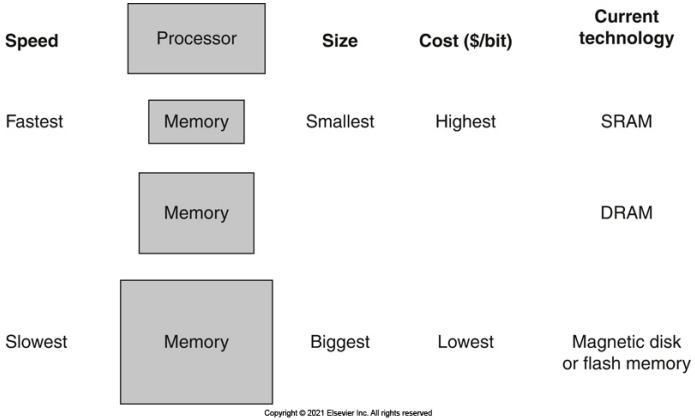
1. The Principle of Locality

- To keep the processor fed with data without stalling, we rely on the **Principle of Locality**:

- **Temporal Locality (Locality in Time):**
  - If an item is referenced, it will tend to be referenced again soon.
  - *Example:* Loops iterating over the same variables.

- **Spatial Locality (Locality in Space):**
  - If an item is referenced, items whose addresses are close by will tend to be referenced soon.
  - *Example:* Sequentially accessing elements in an array.

1. The Memory Hierarchy Pyramid



- We exploit locality by implementing multiple tiers of memory.
- **Closer to CPU**: Faster, smaller, more expensive per bit.
- **Further from CPU**: Slower, larger, cheaper per bit.

1. Key Terminology

- **Block (or Line):** The minimum unit of information that can be either present or not present in a two-level hierarchy.
- **Hit:** Data requested by the processor appears in some block in the upper level.
  - **Hit Rate:** The fraction of memory accesses found in the upper level.
  - **Hit Time:** The time required to access data in the upper level.
- **Miss:** Data is not found in the upper level.
  - **Miss Penalty:** The time required to fetch the block from the lower level into the upper level, plus the time to deliver it to the processor.

1. Harris Perspective: The “Desk” Analogy

- *Harris & Harris* provide a highly physical analogy for the memory hierarchy:
- **The CPU**: You, sitting at a desk, actively working on a problem.
- **The Cache**: The physical surface of your desk. Small, but you can access any paper on it instantly (SRAM).
- **Main Memory**: A filing cabinet across the room. It holds much more data, but retrieving a folder requires you to stand up and walk over (DRAM Miss Penalty).
- **Magnetic Disk**: The warehouse down the street. It holds everything, but takes a massive amount of time to fetch.

1. Average Memory Access Time (AMAT)

- How do we mathematically measure the performance of this hierarchy?
- **AMAT Formula:** `Hit Time + (Miss Rate` $\times$ `Miss Penalty)`
- *Example*: If cache hit time is 1 cycle, miss penalty is 100 cycles, and miss rate is 5%:
- `AMAT = 1 + (0.05` $\times$ `100) = 1 + 5 = 6 cycles`

2. Memory Technologies: SRAM vs DRAM

- **SRAM (Static RAM):**
  - Used for caches (L1, L2, L3).
  - Fast (0.5ns - 2.5ns access time).
  - Requires no refreshing to keep data (hence "static").
  - Low density and very expensive (6 transistors per bit).
- **DRAM (Dynamic RAM):**
  - Used for Main Memory.
  - Slower (50ns - 70ns access time).
  - Data stored as a charge on a capacitor; must be periodically "refreshed" because charge leaks.
  - High density and cheaper (1 transistor per bit).

2. Deep-Dive: Inside a DRAM Cell

- **1T1C Architecture:** 1 Transistor + 1 Capacitor per bit. Massive density.
- **Destructive Reads:** Reading the capacitor drains its tiny electrical charge. The sense amplifiers must immediately rewrite the data back into the cell.
- **Refresh Cycles:** Capacitors leak! The memory controller must halt the CPU and *refresh* (read and rewrite) every row every ≈ 64ms to prevent data corruption.
- **Multiplexing:** Row Address Strobe (RAS) and Column Address Strobe (CAS) send the row and column halves of the address over the exact same pins to save space on the physical chip.

2. Non-Volatile Memory

- **Flash Memory:**
  - Non-volatile semiconductor memory (EEPROM).
  - Survives power-off.
  - Reads are fast, but writes are significantly slower and physically wear out the memory cells over time (requires wear leveling).
- **Magnetic Disk:**
  - Used for massive secondary storage.
  - Relies on moving mechanical parts (platters and read/write heads).
  - Measured in milliseconds (millions of times slower than SRAM).

2. Hamacher Perspective: Write Policies

- When the CPU executes a `sw` (Store Word), data is written to the Cache. But when does it update Main Memory?
- **Write-Through:** Data is written to *both* the cache block and the lower-level memory simultaneously. Perfectly safe, but extremely slow.
- **Write-Back:** Data is written *only* to the cache. The block is marked with a **Dirty Bit**. It is only written to Main Memory if that block is evicted later. Incredibly fast, but vulnerable to power loss.

2. Hamacher Perspective: Memory Interleaving

- To reduce the **Miss Penalty** when fetching a block from slow DRAM, *Hamacher* introduces **Interleaving**.
- Instead of storing sequential blocks on one memory chip, the memory controller distributes consecutive blocks across multiple discrete memory banks.
- The CPU can send fetch requests to all banks simultaneously, receiving the data in parallel rather than sequentially!

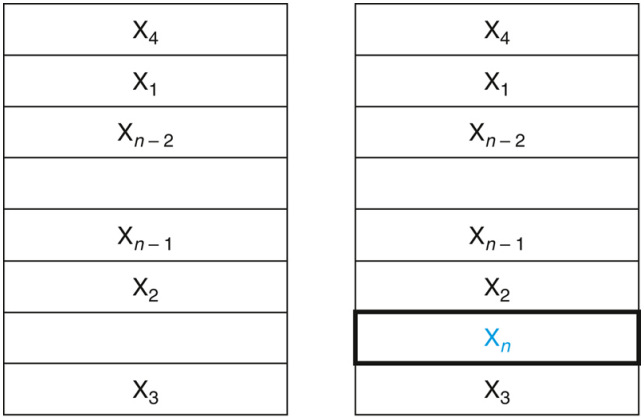
3. Mapping Memory to Cache

- The **Cache** is the level of the memory hierarchy closest to the CPU.
- How do we map memory blocks from the massive Main Memory into the tiny Cache?
- There are three fundamental topologies, trading off hardware complexity against miss rates:
  1. **Direct-Mapped Caches**
  2. **Fully Associative Caches**
  3. **Set-Associative Caches (N-Way)**

3. Direct Mapped Caches

- In a direct-mapped cache, each memory block maps to exactly *one* specific block in the cache.
- **Mapping Formula:** `(Block address) modulo (Number of blocks in the cache)`
- **Address Bit Breakdown:**
  - **Offset:** The lowest bits. Identifies specific byte. Width = $\log_2$(Bytes per Block).
  - **Index:** The middle bits. Identifies exact cache row. Width = $\log_2$(Number of Cache Blocks).
  - **Tag:** Leftover upper bits stored alongside data to verify a match.
- **Pros/Cons:** Extremely fast hit time (only one tag to check). Very cheap. Prone to high *Conflict Misses*.

3. Direct Mapped Cache Architecture



a. Before the reference to X_n b. After the reference to X_n

3. Fully Associative Caches

- To solve conflict misses, a **Fully Associative** cache allows any memory block to be placed in *any* available cache block.
- **Mapping:** There is no Index field! The physical address only contains the **Tag** and **Offset**.
- **Hardware Cost:** The cache must compare the requested Tag against *every single Tag in the cache simultaneously*. Requires massive, expensive parallel hardware comparators.
- **Pros/Cons:** Lowest possible miss rate. Incredibly expensive and consumes significant power. Restricted to very small memory structures (like TLBs).

3. Set-Associative Caches (N-Way)

- The "Goldilocks" compromise. The cache is divided into "Sets", and each set contains N cache blocks (e.g., 2-way, 4-way, 8-way).
- **Mapping:** A memory block maps to exactly *one* Set (using the Index), but can be placed into *any* of the N blocks within that set.
- **Address Bit Breakdown:**
  - **Offset:** $\log_2$(Bytes per Block).
  - **Index:** $\log_2$(Number of Sets).
  - **Tag:** Leftover bits.
- **Pros/Cons:** Greatly reduces conflict misses compared to Direct-Mapped, without the insane hardware costs of Fully Associative. Uses LRU replacement.

4. Level 1: Basic Synchronous Memory

- How do we build Main Memory in RTL? As a massive array of registers.

```
module main_memory (
    input  logic        clk,
    input  logic        we,       // Write Enable
    input  logic [31:0] addr,
    input  logic [31:0] wd,       // Write Data
    output logic [31:0] rd        // Read Data
);
    logic [31:0] mem [0:1023];    // 4KB Memory Array

    assign rd = mem[addr[31:2]];  // Asynchronous Read (Word aligned)

    always_ff @(posedge clk) begin
        if (we) mem[addr[31:2]] <= wd; // Synchronous Write
    end
```

4. Level 2: Direct-Mapped Cache Arrays

- A Direct-Mapped Cache requires three distinct physical arrays:

```
// Parameters for a small 4KB Direct-Mapped Cache
// Block Size = 1 Word (4 Bytes)
// Number of Blocks = 1024

logic [31:0] cache_data  [0:1023]; // The actual data blocks
logic [19:0] cache_tags  [0:1023]; // The upper 20 bits of the address
logic        cache_valid [0:1023]; // 1-bit valid flags
```

4. Level 3: Combinational Hit/Miss Logic

- The Cache Controller must slice the address and determine a Hit.

```
// Step 1: Slice the 32-bit CPU physical address
assign index = cpu_address[11:2];  // 10 bits for 1024 blocks
assign tag   = cpu_address[31:12]; // Remaining 20 upper bits

// Step 2: Extract the stored Tag and Valid bit using the Index
assign stored_tag = cache_tags[index];
assign is_valid   = cache_valid[index];

// Step 3: Hit Logic (Combinational)
assign cache_hit = is_valid & (stored_tag == tag);

// If cache_hit == 0, trigger the FSM to stall and fetch from DRAM!
```

5. Example 1: Address Fields (Harris & Harris)

Problem: 32-bit address, Direct-Mapped Cache, 32 KB capacity, 64-Byte blocks. Calculate bits for Offset, Index, and Tag.

- **1. Offset:** Driven by Block Size.

$$\text{Offset Bits} = \log_2(64) = 6 \text{ bits}$$

- **2. Index:** Driven by Number of Blocks.

$$\text{Blocks} = \frac{32,768 \text{ Bytes}}{64 \text{ Bytes}} = 512 \text{ Blocks}$$

$$\text{Index Bits} = \log_2(512) = 9 \text{ bits}$$

- **3. Tag:** Driven by the remaining bits.

$$\text{Tag Bits} = 32 - 9 - 6 = 17 \text{ bits}$$

5. Example 2: Access Trace (Patterson & Hennessy)

Problem: Direct-Mapped Cache with 8 blocks. Access sequence: 22, 26, 22, 26, 16, 3, 16, 18.

Access	Block Address	Cache Index (Block % 8)	Hit/Miss
1	22	$22 \pmod 8 = 6$	Miss
2	26	$26 \pmod 8 = 2$	Miss
3	22	$22 \pmod 8 = 6$	Hit
4	26	$26 \pmod 8 = 2$	Hit
5	16	$16 \pmod 8 = 0$	Miss
6	3	$3 \pmod 8 = 3$	Miss
7	16	$16 \pmod 8 = 0$	Hit
8	18	$18 \pmod 8 = 2$	Miss (Evicts 26)

Result: 5 Misses, 3 Hits.

5. Example 3: Performance Impact (Hamacher)

Problem: Base CPI = 1.0, Hit Time = 1 cycle, Miss Penalty = 100 cycles, Miss Rate = 5%.

- **1. Calculate Average Memory Access Time (AMAT):**

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

$$\text{AMAT} = 1 + (0.05 \times 100) = 1 + 5 = 6 \text{ cycles}$$

- **2. Calculate Effective CPI:**

$$\text{Memory Stall Cycles} = \text{Miss Rate} \times \text{Miss Penalty} = 5$$

$$\text{Effective CPI} = \text{Base CPI} + \text{Memory Stall Cycles} = 1.0 + 5.0 = 6.0$$

Conclusion: A 5% miss rate makes the CPU 6 times slower!

Synthesis: Quantifying Performance (Ch 1-4)

1. The Baseline: Iron Law (Ch 1)

$$\text{CPU Time} = \text{IC} \times \text{CPI} \times \text{Clock Cycle Time}$$

- **Single-Cycle:** Ideal CPI $= 1.0$, but Clock Cycle Time is massive.

2. The Pipelining Upgrade (Ch 4)

$$\text{CPI}_{\text{pipeline}} = \text{Ideal CPI } (1.0) + \text{Pipeline Stall Cycles / Instr}$$

- Clock cycle is drastically reduced (split into 5 stages).
- CPI increases slightly due to structural, data, and control hazards.
- **Proof:** The massive clock speed increase vastly outweighs the minor stall penalty.

Synthesis: The Memory Wall (Ch 5)

3. The Memory Wall & AMAT Main memory is slow. If we access it every cycle, the pipeline breaks.

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

4. Fully Integrated System Performance We calculate the exact time penalty of cache misses on our pipeline:

$$\text{Mem Stalls} = \text{IC} \times \frac{\text{Mem Accesses}}{\text{Instr}} \times \text{Miss Rate} \times \text{Miss Penalty}$$

$$\text{CPI}_{\text{actual}} = \text{CPI}_{\text{pipeline}} + \frac{\text{Mem Stall Cycles}}{\text{Instr}}$$

Conclusion: Caches successfully exploit locality to keep Miss Rates $< 5\%$, driving AMAT close to 1 cycle and preserving our pipelined clock rate!

5. Hardware Simulation: The Silicon Proof

Test Setup: ‘loop_test.asm‘

- Executed a 5-iteration loop summing an array of 4 integers.
- Explicitly tests for **temporal locality**.
- Simulated with a hardcoded 10-cycle main memory penalty in SystemVerilog.

The Uncached Execution (Direct to Main Memory)

```
PERFORMANCE METRICS (UNCACHED):
Total Clock Cycles: 347
Instructions Executed: ~80
Effective CPI: 4.34
```

Result: The CPU spent the vast majority of its execution time frozen, waiting for the Memory Wall.

5. Hardware Simulation: The Silicon Proof (Cont.)

The Cached Execution (L1 Enabled)

- The first iteration incurred the 10-cycle penalty (4 Misses).
- The remaining 4 iterations hit the cache instantly (24 Hits).

```
PERFORMANCE METRICS (CACHED):
Total Clock Cycles: 151
Instructions Executed: ~80
Effective CPI: 1.89
Cache Hits: 24
Cache Misses: 4
```

Conclusion: By uniting the Iron Law of Performance, Pipelined Datapaths, and the Cache Memory Hierarchy, we successfully dropped the execution time from 347 cycles to **151 cycles!**

Questions?