# Computer Architecture Project - Multi Core Processor Simulator

Sagi Ber: 208276188
Igor Bezuyevskiy: 342388717
Yaron Silberstein: 318321759

Lecturer: Doctor Gadi Oxman
TA: Mr Eyal Naor

15/2/2025

## 1 General Overview

In this project, we designed and implemented a simulation of a pipelined, multi-core processor architecture using the MESI protocol for cache coherence. The CPU simulator processes a hexadecimal-decoded set of instructions and initial memory states for each core. The output includes trace files for instructions, memory, cache status, core registers, and main memory data, along with performance statistics.

The goal is to analyze multi-core processing performance, handle memory accesses, and ensure consistency across caches using the MESI protocol. The project was developed collaboratively using a Git repository, enabling parallel deployment.

## 2 Pipeline Implementation in a Single Core

The pipeline implementation in a single core follows a structured process where each instruction moves through multiple stages: Fetch, Decode, Execute, Memory, and Write-Back. The pipeline registers (FD, DE, EM, MWB) store intermediate results between these stages, ensuring smooth execution flow while minimizing hazards. The following sections describe each stage in detail:

### 2.1 Pipeline Stages

**Fetch (FD)**:

- The instruction is fetched from the instruction memory (IMEM) using the program counter (PC).
- The fetched instruction is stored in the Fetch-Decode (FD) register along with the current PC.
- If a branch is taken, the PC is updated accordingly using `pipe_decode()`.

- If a stall is detected, the fetch stage holds its operation to maintain consistency.

**Decode (DE)**:

- The instruction is decoded to extract the opcode, source registers (R_RS, R_RT), destination register (RD), and immediate value (IMM).
- Stalls are introduced if a RAW hazard is detected, preventing incorrect data usage.
- Updates Decode-Execute (DE) registers with values needed for execution.
- Initiates Branch Resolution:
  - If a branch instruction is detected, the pipeline must determine whether the branch will be taken.
  - Since no branch prediction is implemented, branches are resolved in the decode stage.
  - If the branch is taken, the program counter is updated, and next instruction will be executed due to delay slot functionality.

**Execute (EM)**:

- Arithmetic and logic operations are performed using the ALU.
- The result is stored in the Execute-Memory (EM) register.
- For memory operations, the effective memory address is computed.
- Handles conditional branches by updating the PC if required.

**Memory (MWB)**:

- Load/store operations are processed.
- If a cache hit occurs, data is retrieved from the cache.
- If a cache miss occurs, a bus request is generated using `pipe_memory_hit_or_bus_request()`.
- Manages cache coherence using the MESI protocol.

**Write-Back (WB)**:

- The result is written back to registers if necessary.
- The correct result is selected using the write-back multiplexer (select between ALU or MEM output ).
- Register updates are handled by core_regs_update().

## 2.2 Stall Handling

- If a memory stall occurs, the simulator prevents updates to certain pipeline registers to maintain data integrity.
- If a RAW hazard is detected, the decode stage stalls until the dependency is resolved.

By structuring pipeline execution at both core and simulator levels, the system maintains efficiency and prevents bottlenecks in a multi-core environment.

# 3 Bus Request Handling and MESI Stage Update

The bus request handling mechanism ensures that memory access and cache coherence operations are performed efficiently across cores. Each core may issue bus requests when a memory access is needed and the data is not available in its cache. The bus request process follows these steps:

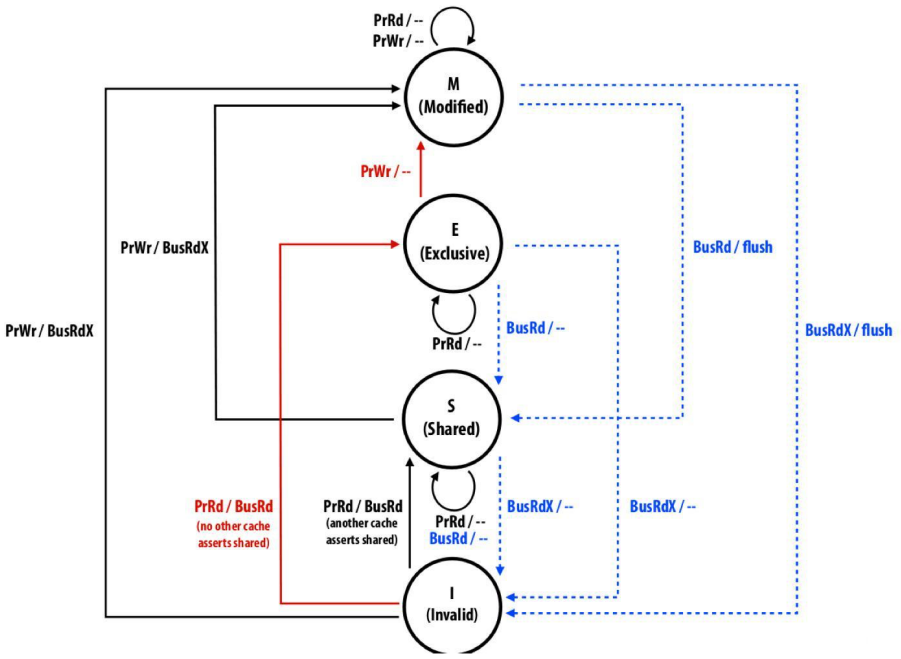


Figure 1: MESI protocol - taken from Multy-Core Coherency recitation

## 3.1 Bus Request Process

- A core issues a request to the bus when a memory operation requires data not available in the cache.
- The request is queued and handled based on a round-robin arbitration mechanism implemented in `round_robin_arbitration()`.
- If a bus request is granted, `execute_transaction()` processes the request and updates the relevant structures.
- Following Figure 1 - if Modified data is found in another core's cache, a cache-to-cache transfer will be initiated. For all other cases Main memory access will be triggered. Any send operation, implemented using `send_transaction()`.

## 3.2 MESI Protocol Update

To maintain cache coherence, each cache line follows the MESI protocol (Modified, Exclusive, Shared, Invalid). The protocol is updated based on bus signals in the initial state of the transaction (snooping):

- If a core reads data and no other core has it in a Modified state, the line is marked as Shared or Exclusive.
- If a core writes to a line, a BUSRDX command is issued, and other cores invalidate their copies (transition to Invalid state).
- If a core holds a line in Modified state and another core requests it with BUSRDX, the line is flushed to memory before being transitioned to Shared.
- The `pipe_memory_bus_snooping()` function continuously monitors bus transactions and updates cache states accordingly.
- By Figure 1 - for any transaction of BUSRDX the data will be arrive from Main memory (exclude the case of Modified block in another core)

### 3.3 Memory Stall Handling

Memory stalls occur when a core attempts to access data that is not available in its cache, requiring a bus transaction to retrieve the data from another core or main memory. During a stall, the pipeline halts further instruction execution to prevent incorrect data dependencies. The function `pipe_memory_hit_or_bus_request()` determines whether the requested data is in the cache; if not, a bus request is initiated, and the core enters a stalled state. While stalled, no new instructions proceed beyond the decode stage, and pipeline registers hold their values until the requested data arrives. The function `pipe_memory_bus_snooping()` ensures that cores listening to the bus update their cache states accordingly. Once the data is retrieved, the stall condition is lifted, pipeline execution resumes, and the fetched data is used in subsequent pipeline stages. These mechanisms ensure proper synchronization and data consistency between cores while optimizing performance by reducing unnecessary memory accesses.

## 4 Testing Programs

To validate the functionality and correctness of our simulator, we implemented several test programs. These programs were designed to evaluate different aspects of our system, including pipeline execution, memory access, cache coherence, and bus arbitration.

### 4.1 Test 1: Sequential Counter

- Objective: This test increments a counter across all four cores in a sequential manner.
- Expected Outcome: The final result should reflect the correct count stored in the main memory after all cores have executed their portion of the task.
- Purpose: This test ensures proper synchronization of shared memory updates across multiple cores.
- Implementation: This assembly code iterates over a shared counter, checking if its core ID matches the counter's modulo 4 value, and if so, it increments the counter in memory. The process continues until the internal counter reaches zero, ensuring each core takes turns modifying the counter based on their assigned ID.

- Assembly code of core 2 for example:

```
    add $r6, $imm, $zero, 127   # PC= 0, Setting the counter value
    add $r2, $zero, $imm, 2     # PC= 1, Setting core id
    add $r5, $zero, $imm, 4     # PC= 2, For mod 4
loop 1:
    lw $r3, $zero, $imm, 0      # PC= 3, Reading counter val
    bge $imm, $r3, $r5, 8       # PC= 4, Branch to loop 2
    add $zero, $zero, $zero, 0  # PC= 5, NO-OP
    beq $imm, $zero, $zero, 11  # PC= 6, Branch to loop 3
    add $zero, $zero, $zero, 0  # PC= 7, NO-OP
loop 2:
    sub $r3, $r3, $r5, 0        # PC= 8, Mod 4 calculation
    beq $imm, $zero, $zero, 4   # PC= 9,
    add $zero, $zero, $zero, 0  # PC= 10, NO-OP
loop 3:
    beq $imm, $r3, $r2, 15      # PC= 11, Increase if we are the core
    add $zero, $zero, $zero, 0  # PC= 12, NO-OP
    beq $imm, $zero, $zero, 3   # PC= 13, Otherwise go back
    add $zero, $zero, $zero, 0  # PC= 14, Stalling
    lw $r3, $zero, $imm, 0      # PC= 15, Reading counter from memory
    add $r3, $r3, $imm, 1       # PC= 16, Increasing counter
    sw $r3, $imm, $zero, 0      # PC= 17, Writing back to memory
    bne $imm, $zero, $r6, 3     # PC= 18, Iterating while starting value not 0
    sub $r6, $r6, $imm, 1       # PC= 19, Reducing internal counter
    halt $zero, $zero, $zero, 0 # PC= 20
    halt $zero, $zero, $zero, 0 # PC= 21
    halt $zero, $zero, $zero, 0 # PC= 22
    halt $zero, $zero, $zero, 0 # PC= 23
    halt $zero, $zero, $zero, 0 # PC= 24, Finish.
```

## 4.2 Test 2: Vector Addition (Serial)

- Objective: Compute the sum of two vectors using a single core.
- Expected Outcome: The resulting vector should match the expected sum stored in the main memory.
- Purpose: Validates correct execution of arithmetic operations within the pipeline and proper handling of memory accesses.
- Implementation: This assembly code reads values from two memory offsets, starting at 0 and 4096, increasing consecutively and adds their values. It stores the result in a designated location, from 8192 until 12288 addresses in main memory. The core iterates 4096 times, one for each addition. It is done on one core only, and when done we read 256 addressesin order to write back the data from cache to main memory, then halts execution.
- Assembly code of core 0:

```
    add $r2, $zero, $imm, 0         # PC=0, Load READ LOW OFFSET
    add $r3, $zero, $imm, 1         # PC=1, Load READ HIGH OFFSET
    sll $r3, $r3, $imm, 12          # PC=2, COMPLETE TO 4096
    add $r4, $r3, $r3, 0            # PC=3, Load WRITE OFFSET
loop1:
    lw $r6, $r2, $r9, 0             # PC=4, Load from low offset
    lw $r7, $r3, $r9, 0             # PC=5, Load from high offset
    add $r8, $r6, $r7, 0            # PC=6, Add loaded values
    sw $r8, $r4, $r9, 0             # PC=7, Store result
    blt $imm, $r9, $r3, loop1       # PC=8, Loop until $r9 >= $r3
    add $r9, $r9, $imm, 1           # PC=9, Increment counter
    beq $imm, $zero, $zero, 17      # PC=10, Jump to PC=17
    add $zero, $zero, $zero, 0      # PC=11, NO-OP
exit:
    halt $zero, $zero, $zero, 0     # PC=12, Halt execution
    halt $zero, $zero, $zero, 0     # PC=13
    halt $zero, $zero, $zero, 0     # PC=14
    halt $zero, $zero, $zero, 0     # PC=15
    halt $zero, $zero, $zero, 0     # PC=16, Finish.
    add $r4, $r4, $r9, 0            # PC=17
    sub $r4, $r4, $imm, 2           # PC=18
    sub $r3, $r4, $imm, 248         # PC=19
loop2:
    lw $r6, $r3, $imm, 256          # PC=20, Load from offset 256
    blt $imm, $r3, $r4, loop2       # PC=21, Loop until $r3 >= $r4
    add $r3, $r3, $imm, 4           # PC=22, Increment address by 4
    beq $imm, $zero, $zero, exit    # PC=23, Jump to exit
    add $zero, $zero, $zero, 0      # PC=24, NO-OP
```

### 4.3 Test 3: Vector Addition (Parallel)

- Objective: Compute the sum of two vectors using all four cores in parallel.
- Expected Outcome: The resulting vector should be identical to the serial test but computed faster.
- Purpose: Tests the efficiency of multi-core execution and the correctness of cache coherence through the MESI protocol.
- Implementation: This assembly code reads values from two memory offsets, starting at 0 and 4096, increasing consecutively and adds their values. It stores the result in a designated location, from 8192 until 12288 addresses in main memory. The code is executed in parallel on 4 core to reduce runtime. Each core receives a quarter of the load (1024 additions), and iterates 256 times. In each iteration an entire block is handled, i.e. 4 results are calculated and written to main memory. When done halts execution.
- Assembly code of core 0 for example:

```
    add $r2, $imm, $zero ,0     # PC=0, Base address for the first memory
                                 block (offset 0)
    add $r3, $imm, $zero, 0     # PC=1, Initialize registers for loading
                                 values from memory
    add $r4, $imm, $zero, 0     # PC=2
    add $r5, $imm, $zero, 0     # PC=3
    add $r6, $imm, $zero, 0     # PC=4, End of initialization
    add $r7, $imm, $zero ,0     # PC=5, Registers for second memory block
                                 values
    add $r8, $imm, $zero ,0     # PC=6
    add $r9, $imm, $zero ,0     # PC=7
    add $r10, $imm, $zero ,0    # PC=8
    add $r11, $imm, $zero ,255  # PC=9, Loop counter (255 iterations)
    add $r12, $imm, $zero, 1    # PC=10, Load 1 into $r12
    sll $r12, $r12, $imm, 12    # PC=11, Shift left by 12 (4096: base of
                                 second memory block)
    add $r13, $r12, $r12, 0     # PC=12, Copy $r12 to $r13 (base of
                                 result storage = 8192)
    add $r14, $imm, $zero, 0    # PC=13, Initialize loop counter
loop1:
    lw $r3, $r2 ,$imm, 0        # PC=14, Load first memory block values
    lw $r4, $r2 ,$imm, 1        # PC=15
    lw $r5, $r2 ,$imm, 2        # PC=16
    lw $r6, $r2 ,$imm, 3        # PC=17
    lw $r7, $r12 ,$imm, 0       # PC=18, Load second memory block values
    lw $r8, $r12 ,$imm, 1       # PC=19
    lw $r9, $r12 ,$imm, 2       # PC=20
    lw $r10, $r12 ,$imm, 3      # PC=21
    add $r3, $r3 ,$r7 , 0       # PC=22, Add corresponding values
    add $r4, $r4 ,$r8 , 0       # PC=23
    add $r5, $r5 ,$r9 , 0       # PC=24
    add $r6, $r6 ,$r10 , 0      # PC=25
    sw $r3, $r13, $imm, 0       # PC=26, Store results in memory
    sw $r4, $r13, $imm, 1       # PC=27
    sw $r5, $r13, $imm, 2       # PC=28
    sw $r6, $r13, $imm, 3       # PC=29
    lw $r3, $r12 ,$imm, 0       # PC=30, Load next block from second memory
    add $r2, $r2 ,$imm, 4       # PC=31, Increment memory addresses for next
                                 iteration
    add $r12, $r12 ,$imm, 4     # PC=32
    add $r13, $r13 ,$imm, 4     # PC=33
    blt $imm, $r14, $r11, loop1 # PC=34, Loop until counter reaches 255
    add $r14, $r14 ,$imm, 1     # PC=35, Increment loop counter
    halt $zero, $zero, $zero, 0 # PC=36 Halt execution
    halt $zero, $zero, $zero, 0 # PC=37
    halt $zero, $zero, $zero, 0 # PC=38
    halt $zero, $zero, $zero, 0 # PC=39
    halt $zero, $zero, $zero, 0 # PC=40, Finish.
```

# 5 Summary

This project provides a detailed simulation of a pipe-lined multi-core processor and mainly implementing the MESI protocol for cache coherence. It enables performance analysis and debugging in multi-core architectures using the log files, trace files and statistics. In addition we can simulate the run of actual programs and test the correctness of our simulator. The project gave us a better understating of how multi-core processors work and about the coherence protocol we learned in class.

# 6 Data Structures and Constants

## 6.1 Defined Constants

| **Constant** | **Description** |
|---|---|
| NUMBER_OF_INPUT_ARGS (28) | Number of command-line arguments |
| MAX_LINE_LENGTH (500) | Maximum length of a line in input files |
| NUMBER_OF_CORE_REGS (16) | Number of general-purpose registers per core |
| INST_MEMORY_DEPTH (1024) | Instruction memory size per core |
| CACHE_DSRAM_DEPTH (256) | Data cache size (rows) |
| CACHE_TSRAM_DEPTH (64) | Tag SRAM depth for MESI metadata |
| CACHE_BLOCK_SIZE (4) | Number of words per cache block |
| CACHE_TAG_WIDTH (12) | Bit width of cache tags |
| MAIN_MEMORY_DEPTH (1048576) | Main memory depth (1MB) |
| MAIN_MEMORY_DELAY (16) | Delay for main memory accesses |
| PIPE_REGS_WIDTH (6) | Number of pipeline registers per stage |
| NUM_CORES (4) | Number of processor cores |

## 6.2 Simulator Struct

| **Field** | **Description** |
|---|---|
| PC_0, PC_1, PC_2, PC_3 | Program counters for each core |
| main_memory[MAIN_MEM_SIZE] | Shared main memory (1MB) |
| max_main_memory_index | Maximum accessed index in main memory |
| run_en | Simulation enable flag |
| clk | Global simulation clock |
| transaction_timer | Timer for bus transactions |
| bus_is_busy | Indicates if the bus is occupied |
| sender_id | ID of the core that should return the data for requesting core(0-3 cores,4 main memory) |

| | |
|---|---|
| dirty_block | Tracks dirty cache blocks (was M and updated to different state) |
| busrdx_en | Current bus command is BUSRDX |
| arb_is_done | Indicates if arbitration is complete |
| bus_is_updated | Flag indicating bus was updated due to requesting core |
| last_transaction | Indicates completion of last bus transaction |
| bus_d, bus_q | Structures for bus communication |

## 6.3 Core Struct

| **Field** | **Description** |
|---|---|
| PC, next_PC | Program counter and next instruction PC |
| regs[NUMBER_OF_CORE_REGS] | General-purpose registers (16 per core) |
| imem[INST_MEMORY_DEPTH] | Instruction memory (1KB per core) |
| dsram[CACHE_DSRAM_DEPTH] | Data SRAM for cache (256 rows) |
| tsram[CACHE_TSRAM_DEPTH] | Tag SRAM for cache (64 rows) |
| pipe_regs | Pipeline registers |
| stats | Performance statistics |
| halt_en | Core halt flag |
| branch_en | Branch enable flag |
| priority | Bus arbitration priority |
| request_bus | Bus request indicator |
| wb_sel | Write-back selection signal |
| stall_en | Core stall flag |
| core_done | Indicates completion of execution |
| bus_cmd, bus_addr, bus_data | Bus command, address, and data |
| is_hit | Cache hit indicator |
| memory_stall | Memory stall flag |
| transaction_number | Tracks transactions for debugging |

## 6.4 Pipeline Registers Struct

| **Field** | **Description** |
|---|---|
| FD_reg | Fetch-Decode pipeline registers |
| DE_regs | Decode-Execute pipeline registers |
| EM_regs | Execute-Memory pipeline registers |
| MWB_regs | Memory-Write-Back pipeline registers |

## 6.5 Bus Struct

| **Field** | **Description** |
|---|---|
| origid | Origin ID (core issuing the command) |
| cmd | Command type (e.g., BUSRD, BUSRDX) |

| | |
|---|---|
| addr | Address being accessed |
| data | Data being transmitted |
| shared | Indicates if the data is shared across caches |

### 6.6 Flip-Flop (ff) Struct

| **Field** | **Description** |
|---|---|
| d | Data input |
| q | Data output |

### 6.7 Pipeline Registers Struct

| **Field** | **Description** |
|---|---|
| FD_reg | Fetch-Decode pipeline registers |
| DE_regs | Decode-Execute pipeline registers |
| EM_regs | Execute-Memory pipeline registers |
| MWB_regs | Memory-Write-Back pipeline registers |

## 7 Main Helper Functions

The project includes various helper functions that manage simulation setup, memory access, pipeline execution, and bus operations:

### 7.1 File Handling Functions

- `open_file_to_read()` - Opens a file for reading.
- `open_file_to_write()` - Opens a file for writing.
- `open_file_to_append()` - Opens a file for appending.

### 7.2 Simulator Initialization Functions

- `init_simulator()` - Initializes the simulator by setting up program counters, memory structures, and bus communication.
- `load_data_from_input_file_to_main_mem()` - Loads memory initialization data from an input file into the main memory of the simulator.
- `initialize_bus()` - Sets up bus structures, ensuring proper handling of memory transactions and inter-core communication.

## 7.3 Core Initialization Functions

- `init_core()` - Initializes a core by setting its program counter, pipeline registers, cache structures, and execution state.
- `load_data_from_input_file_to_core()` - Loads instruction memory data into the core's instruction memory.
- `initialize_dsram()` - Allocates and clears the data SRAM (cache memory) of the core.
- `initialize_tsram()` - Initializes the tag SRAM, which tracks cache states and implements MESI coherence.

## 7.4 Memory and Resource Cleanup Functions

- `free_core()` - Releases all dynamically allocated memory associated with a core, including pipeline registers, cache structures, and statistics tracking.
- `free_simulator()` - Frees memory allocated for the simulator, including main memory and bus structures.
- `shutdown_simulation()` - Ensures that all memory is properly deallocated before terminating the program.