Multicore Processor Simulator

Project Documentation

Course: Computer Architecture
Project: Multicore Processor with Cache Coherence
Date: December 2025

Table of Contents

1. Introduction

This project implements a cycle-accurate simulator for a quad-core processor with cache coherence. The simulator models:

- **4 independent cores** running in parallel
- **5-stage pipeline** per core with delay slots
- **Direct-mapped caches** with MESI coherence protocol
- **Shared main memory** with bus arbitration
- **Round-robin bus arbitration** for memory access

The simulator is written in C and produces detailed trace files for debugging and verification.

2. System Architecture

2.1 High-Level Block Diagram

```
┌─────────────────────────────────────────────────────────────────────┐
│                          SYSTEM BUS                                   │
│                    (Round-Robin Arbitration)                         │
└───────┬───────────────┬───────────────┬───────────────┬─────────────┘
        │               │               │               │
        ▼               ▼               ▼               ▼
┌──────────────┐  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐
│   CORE 0     │  │   CORE 1     │  │   CORE 2     │  │   CORE 3     │
├──────────────┤  ├──────────────┤  ├──────────────┤  ├──────────────┤
│  Pipeline    │  │  Pipeline    │  │  Pipeline    │  │  Pipeline    │
│ (5 stages)   │  │ (5 stages)   │  │ (5 stages)   │  │ (5 stages)   │
├──────────────┤  ├──────────────┤  ├──────────────┤  ├──────────────┤
│   Cache      │  │   Cache      │  │   Cache      │  │   Cache      │
│ (512 words)  │  │ (512 words)  │  │ (512 words)  │  │ (512 words)  │
│  DSRAM+TSRAM │  │  DSRAM+TSRAM │  │  DSRAM+TSRAM │  │  DSRAM+TSRAM │
└──────────────┘  └──────────────┘  └──────────────┘  └──────────────┘
        │               │               │               │
        └───────────────┴───────┬───────┴───────────────┘
                                │
                                ▼
                    ┌───────────────────────┐
                    │     MAIN MEMORY       │
                    │    (2^21 words)       │
                    │    16-cycle latency   │
                    └───────────────────────┘
```

2.2 System Parameters

Component	Specification
Number of Cores	4
Address Space	21 bits (2,097,152 words)
Word Size	32 bits
Instruction Memory	1024 words per core
Data Cache	512 words per core
Cache Block Size	8 words
Memory Latency	16 cycles + 8 words transfer

---

3. Core Design

3.1 Register File

Each core has 16 registers (32-bit each):

Register	Name	Description
R0	$zero	Constant zero (read-only)
R1	$imm	Sign-extended immediate from instruction

Register	Name	Description
R2-R15	General	General-purpose registers

3.2 Program Counter

- **Width**: 10 bits
- **Range**: 0-1023 (instruction memory size)
- **Initial Value**: 0

3.3 Core State Machine

```
┌──────────────────┐
│     RUNNING      │
│                  │
│  Execute cycles  │
└────────┬─────────┘
         │ HALT instruction
         ▼
┌──────────────────┐
│     HALTING      │
│                  │
│  Drain pipeline  │
└────────┬─────────┘
         │ Pipeline empty
         ▼
┌──────────────────┐
│     HALTED       │
│                  │
│  Core stopped    │
└──────────────────┘
```

---

4. Pipeline Implementation

4.1 Pipeline Stages

The processor uses a classic 5-stage RISC pipeline:

```
┌─────────┐   ┌─────────┐   ┌─────────┐   ┌─────────┐   ┌─────────┐
│  FETCH  │──▶│ DECODE  │──▶│ EXECUTE │──▶│   MEM   │──▶│   WB    │
│         │   │         │   │         │   │         │   │         │
│ PC→IMEM │   │ RegRead │   │   ALU   │   │ Cache   │   │RegWrite │
│ IR←Inst │   │ Hazard  │   │ Branch  │   │ Access  │   │         │
└─────────┘   └─────────┘   └─────────┘   └─────────┘   └─────────┘
```

4.2 Stage Details

Fetch Stage

- Reads instruction from instruction memory at PC
- Increments PC (unless stalled)
- Passes instruction to Decode stage

Decode Stage

- Decodes instruction fields (opcode, rd, rs, rt, immediate)
- Reads register values
- **Resolves branches** (branch decision made here)
- **Detects data hazards** and stalls if needed
- No forwarding - all hazards resolved by stalling

Execute Stage

- Performs ALU operations
- Calculates memory addresses for LW/SW
- Computes branch targets

Memory Stage

- Accesses data cache for LW/SW instructions
- **Stalls on cache miss** until data available
- Handles MESI protocol transactions

Writeback Stage

- Writes results back to register file
- Updates R2-R15 (R0 and R1 are read-only)

4.3 Delay Slot

The processor implements a **single delay slot** for branch instructions:

- The instruction immediately after a branch **always executes**
- This is true regardless of whether the branch is taken
- Compilers/assemblers must fill the delay slot appropriately

4.4 Data Hazard Detection

Hazards are detected in the Decode stage by checking dependencies:

```
// For ALU instructions: check rs and rt as sources
if (decode.rs == execute.rd || decode.rs == mem.rd || decode.rs == wb.rd ||
    decode.rt == execute.rd || decode.rt == mem.rd || decode.rt == wb.rd)
    → STALL

// For SW instruction: check rd, rs, and rt as sources (rd is the data to store)
```

```
  if (decode.rd == execute.rd || decode.rd == mem.rd || decode.rd == wb.rd ||
      decode.rs == execute.rd || decode.rs == mem.rd || decode.rs == wb.rd ||
      decode.rt == execute.rd || decode.rt == mem.rd || decode.rt == wb.rd)
      → STALL
```

4.5 Pipeline Stalls

Two types of stalls:

Stall Type	Cause	Resolution
Decode Stall	Data hazard (RAW)	Wait for producer to reach WB
Memory Stall	Cache miss	Wait for bus transaction

5. Cache System

5.1 Cache Organization

```
┌─────────────────────────────────────────────────────────────────┐
│                           CACHE (512 words)                      │
├─────────────────────────────────────────────────────────────────┤
│  64 cache lines × 8 words per line = 512 words                  │
├─────────────────────────────────────────────────────────────────┤
│                                                                  │
│  ┌───────────────────────────────────────────────────────────┐  │
│  │ TSRAM (Tag + State RAM) - 64 entries                      │  │
│  │ ┌─────────┬───────────────────────────────────────────┐   │  │
│  │ │ MESI(2) │                TAG (12 bits)              │   │  │
│  │ └─────────┴───────────────────────────────────────────┘   │  │
│  └───────────────────────────────────────────────────────────┘  │
│                                                                  │
│  ┌───────────────────────────────────────────────────────────┐  │
│  │ DSRAM (Data RAM) - 512 entries (32-bit words)             │  │
│  │ ┌──────┬──────┬──────┬──────┬──────┬──────┬──────┬──────┐ │  │
│  │ │Word 0│Word 1│Word 2│Word 3│Word 4│Word 5│Word 6│Word 7│ │  │
│  │ └──────┴──────┴──────┴──────┴──────┴──────┴──────┴──────┘ │  │
│  └───────────────────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────────────────┘
```

5.2 Address Breakdown (21 bits)

```
┌────────────────┬─────────────────┬────────────────┐
│    TAG (12)    │   INDEX (6)     │   OFFSET (3)   │
├────────────────┼─────────────────┼────────────────┤
│  bits 20:9     │   bits 8:3      │   bits 2:0     │
└────────────────┴─────────────────┴────────────────┘
```

```
• TAG: Identifies which memory block is cached
• INDEX: Selects one of 64 cache lines
• OFFSET: Selects one of 8 words within the block
```

5.3 Cache Parameters

Parameter	Value
Total Size	512 words (2KB)
Block Size	8 words (32 bytes)
Number of Lines	64
Associativity	Direct-mapped
Write Policy	Write-back, Write-allocate
Tag Bits	12
Index Bits	6
Offset Bits	3

5.4 Write Policy

- **Write-Back**: Modified data is written to memory only on eviction
- **Write-Allocate**: On write miss, fetch block then write to cache

---

6. MESI Protocol

6.1 State Encoding

Code	State	Description
0	Invalid (I)	Line not valid, must fetch from memory
1	Shared (S)	Clean copy, may exist in other caches
2	Exclusive (E)	Clean copy, no other cache has it
3	Modified (M)	Dirty copy, must write back before eviction

6.2 State Transition Diagram

```
                      ┌─────────────────────────────────────┐
                      │                                     │
                      │                                     │
                      ▼                                     │
                                                            │
                ┌───────────┐                               │
           ┌────│  INVALID  │◄───────────────────────┐      │
           │    └───────────┘                        │      │
           │                                         │      │
           │          │                              │      │
```

```
      │          │ PrRd/PrWr                  │     │
      │          │ (miss)                     │     │
      │          ▼                            │     │
      │   ┌───────────┐  BusRd (snoop)  ┌───────────┐│
      │   │ EXCLUSIVE │────────────────►│  SHARED   ││
      │   └───────────┘                 └───────────┘│
      │          │                            │     │
      │          │ PrWr                       │ PrWr │
      │          ▼                            ▼     │
      │   ┌───────────┐                 ┌───────────┐│
      └───│ MODIFIED  │◄────────────────│  SHARED   ││
          └───────────┘      PrWr       └───────────┘│
                 │                            │     │
                 │ BusRdX (snoop)             │     │
                 │ Flush data                 │     │
                 └────────────────────────────┴─────┘
```

6.3 Bus Commands

Command	Code	Description
No-Op	0	No bus activity
BusRd	1	Read request (for read miss)
BusRdX	2	Read-exclusive request (for write miss)
Flush	3	Write modified block to memory

6.4 Snoop Logic

When a core observes a bus transaction:

Current State	Bus Command	Action	New State
Modified	BusRd	Flush data, set shared	Shared
Modified	BusRdX	Flush data, invalidate	Invalid
Exclusive	BusRd	Set shared signal	Shared
Exclusive	BusRdX	Invalidate	Invalid
Shared	BusRd	Set shared signal	Shared
Shared	BusRdX	Invalidate	Invalid
Invalid	Any	No action	Invalid

---

7. Bus and Arbitration

7.1 Bus Signals

```
┌─────────────────────────────────────────────────────────────────────┐
│                           SYSTEM BUS                                 │
├─────────────────────────────────────────────────────────────────────┤
│  bus_origid  [2 bits]  - Requesting core ID (0-3)                   │
│  bus_cmd     [2 bits]  - Command (NoOp/BusRd/BusRdX/Flush)          │
│  bus_addr    [21 bits] - Memory address                             │
│  bus_data    [32 bits] - Data word (for Flush)                      │
│  bus_shared  [1 bit]   - Shared signal from snoopers                │
└─────────────────────────────────────────────────────────────────────┘
```

7.2 Round-Robin Arbitration

The bus uses round-robin arbitration to ensure fairness:

```
Priority rotates: Core 0 → Core 1 → Core 2 → Core 3 → Core 0 → ...

When multiple cores request the bus:
1. Check from current priority core
2. Grant to first requesting core in order
3. After transaction, priority moves to next core
```

7.3 Bus Transaction Sequence

Read Miss (BusRd)

```
Cycle 1:    Core issues BusRd
Cycles 2-17: Wait for memory (16 cycles)
Cycles 18-25: Receive 8 words from memory
Cycle 26:   Transaction complete, core resumes
```

Write Miss (BusRdX)

```
Cycle 1:    Core issues BusRdX
            Other caches invalidate their copies
Cycles 2-17: Wait for memory (16 cycles)
Cycles 18-25: Receive 8 words from memory
Cycle 26:   Transaction complete, core writes
```

Flush (Modified → Memory)

```
Cycles 1-8: Write 8 words to memory
Cycle 9:    Transaction complete
```

8. Main Memory

8.1 Specifications

Parameter	Value
Size	2^21 words (8 MB)
Address Width	21 bits
Word Width	32 bits
Read Latency	16 cycles
Transfer Rate	1 word per cycle
Block Transfer	8 words (8 cycles)

8.2 Memory Timing

```
Memory Read Transaction:
┌───────────────────────────────────────────────────────────────────┐
│ Request │◄────── 16 cycles latency ──────►│ W0│W1│W2│W3│W4│W5│W6│W7│
└───────────────────────────────────────────────────────────────────┘
          └─────────────────────────────────┘ └───────────────────┘
                  Wait for memory                 8 words arrive
                                                  (1 per cycle)
```

9. Instruction Set Architecture

9.1 Instruction Format (32 bits)

```
┌────────┬────────┬────────┬────────┬──────────────────────────────┐
│ Opcode │   Rd   │   Rs   │   Rt   │          Immediate           │
│  (8)   │  (4)   │  (4)   │  (4)   │            (12)              │
├────────┼────────┼────────┼────────┼──────────────────────────────┤
│ 31:24  │ 23:20  │ 19:16  │ 15:12  │            11:0              │
└────────┴────────┴────────┴────────┴──────────────────────────────┘
```

9.2 Instruction Set

Arithmetic/Logic Instructions

Opcode	Mnemonic	Operation
0	ADD	R[rd] = R[rs] + R[rt]

Opcode	Mnemonic	Operation
1	SUB	R[rd] = R[rs] - R[rt]
2	AND	R[rd] = R[rs] & R[rt]
3	OR	R[rd] = R[rs] | R[rt]
4	XOR	R[rd] = R[rs] ^ R[rt]
5	MUL	R[rd] = R[rs] × R[rt]
6	SLL	R[rd] = R[rs] << R[rt]
7	SRA	R[rd] = R[rs] >> R[rt] (arithmetic)
8	SRL	R[rd] = R[rs] >> R[rt] (logical)

Branch Instructions

Opcode	Mnemonic	Condition
9	BEQ	Branch if R[rs] == R[rt]
10	BNE	Branch if R[rs] != R[rt]
11	BLT	Branch if R[rs] < R[rt]
12	BGT	Branch if R[rs] > R[rt]
13	BLE	Branch if R[rs] <= R[rt]
14	BGE	Branch if R[rs] >= R[rt]
15	JAL	Jump and link (R[15] = PC+1)

Note: Branch target is R[rd][9:0]. Delay slot always executes.

Memory Instructions

Opcode	Mnemonic	Operation
16	LW	R[rd] = MEM[R[rs] + R[rt]]
17	SW	MEM[R[rs] + R[rt]] = R[rd]

Control Instructions

Opcode	Mnemonic	Operation
20	HALT	Stop this core

---

10. File Formats

10.1 Input Files

File	Description	Format
imem0-3.txt	Instruction memory	8 hex digits per line
memin.txt	Initial main memory	8 hex digits per line

10.2 Output Files

File	Description	Format
memout.txt	Final main memory	8 hex digits per line
regout0-3.txt	Final registers (R2-R15)	8 hex digits per line
core0-3trace.txt	Pipeline trace	See below
bustrace.txt	Bus transactions	See below
dsram0-3.txt	Cache data	8 hex digits per line
tsram0-3.txt	Cache tags	8 hex digits per line
stats0-3.txt	Statistics	Text format

10.3 Trace File Formats

Core Trace Format

```
CYCLE FETCH DECODE EXEC MEM WB R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15
```

- CYCLE: Decimal cycle number
- FETCH-WB: 3 hex digits for PC, or "---" if inactive
- R2-R15: 8 hex digits each

Bus Trace Format

```
CYCLE bus_origid bus_cmd bus_addr bus_data bus_shared
```

- CYCLE: Decimal
- bus_origid, bus_cmd, bus_shared: 1 hex digit
- bus_addr: 6 hex digits
- bus_data: 8 hex digits

10.4 Statistics File Format

```
cycles X
instructions X
read hit X
```

```
write hit X
read miss X
write miss X
decode stall X
mem stall X
```

---

11. Implementation Details

11.1 Source Code Organization

```
src/
├── MulticoreProcessor.c      # Main simulator entry point
├── Core/
│   ├── include/
│   │   ├── Core.h           # Core state structures
│   │   ├── Pipeline.h       # Pipeline stage definitions
│   │   ├── Cache.h          # Cache structures
│   │   └── Opcodes.h        # Instruction opcodes
│   └── src/
│       ├── Core.c           # Core initialization/execution
│       ├── Pipeline.c       # Pipeline stage logic
│       ├── Cache.c          # Cache operations
│       └── Opcodes.c        # Instruction execution
└── Interface/
    ├── include/
    │   ├── Bus.h            # Bus interface
    │   ├── MainMemory.h     # Memory interface
    │   ├── Files.h          # File I/O
    │   └── Helpers.h        # Utility functions
    └── src/
        ├── Bus.c            # Bus arbitration logic
        ├── MainMemory.c     # Memory operations
        └── Files.c          # File handling
```

11.2 Key Data Structures

Core Structure

```c
typedef struct Core {
    int id;                    // Core ID (0-3)
    uint32_t registers[16];    // Register file
    uint32_t pc;               // Program counter
    uint32_t imem[1024];       // Instruction memory
    PipelineState pipeline;    // Pipeline stages
    Cache cache;               // Data cache
    CoreState state;           // Running/Halting/Halted
    Statistics stats;          // Performance counters
} Core;
```

Cache Structure

```c
typedef struct Cache {
    uint32_t dsram[512];        // Data SRAM
    uint32_t tsram[64];         // Tag + State SRAM
    // TSRAM format: [31:20] = tag, [1:0] = MESI state
} Cache;
```

11.3 Simulation Loop

```
while (!all_cores_halted()) {
    global_cycle++;

    // Phase 1: Bus arbitration and transactions
    handle_bus_transactions();

    // Phase 2: Snoop all caches
    for each core:
        snoop_cache(core);

    // Phase 3: Advance pipelines
    for each core:
        if (core.state != HALTED)
            execute_pipeline_cycle(core);

    // Phase 4: Write trace files
    write_traces();
}
```

---

12. Test Programs

12.1 Counter Test (counter/)

Description: Four cores increment a shared counter at memory address 0.

Algorithm:

- Each core increments the counter 128 times
- Cores take turns in round-robin order (0→1→2→3→0→...)
- Synchronization uses memory-based flags

Expected Result:

- memout.txt[0] = 0x00000200 (512)

12.2 Serial Matrix Multiply (mulserial/)

Description: Single-core 16×16 matrix multiplication.

Memory Layout:

Address Range	Content
0x000-0x0FF	Matrix A (16×16)
0x100-0x1FF	Matrix B (16×16)
0x200-0x2FF	Result C (16×16)

Algorithm:

```
for i = 0 to 15:
    for j = 0 to 15:
        C[i][j] = 0
        for k = 0 to 15:
            C[i][j] += A[i][k] * B[k][j]
```

12.3 Parallel Matrix Multiply (mulparallel/)

Description: Four cores collaborate on 16×16 matrix multiplication.

Work Distribution:

- Core 0: Rows 0-3
- Core 1: Rows 4-7
- Core 2: Rows 8-11
- Core 3: Rows 12-15

Expected Result: Same as serial multiply, faster execution.

---

13. Building and Running

13.1 Build Requirements

- Visual Studio Community Edition (Windows)
- C compiler with C99 support

13.2 Build Instructions

1. Open `MulticoreProcessor.sln` in Visual Studio
2. Select Release configuration
3. Build → Build Solution (Ctrl+Shift+B)
4. Executable: `sim.exe`

13.3 Running the Simulator

With command-line arguments:

```
sim.exe imem0.txt imem1.txt imem2.txt imem3.txt memin.txt memout.txt
        regout0.txt regout1.txt regout2.txt regout3.txt
        core0trace.txt core1trace.txt core2trace.txt core3trace.txt
        bustrace.txt
        dsram0.txt dsram1.txt dsram2.txt dsram3.txt
        tsram0.txt tsram1.txt tsram2.txt tsram3.txt
        stats0.txt stats1.txt stats2.txt stats3.txt
```

With default filenames (files in same directory):

```
sim.exe
```

13.4 Assembler

The project includes an assembler for converting assembly to machine code:

```
asm.exe program.asm imem.txt
```

Appendix A: MESI State Truth Table

Current	Event	Next	Bus Action	Notes
I	PrRd	E/S	BusRd	E if no sharers, S if shared
I	PrWr	M	BusRdX	Invalidates other copies
S	PrRd	S	-	Hit, no bus activity
S	PrWr	M	BusRdX	Upgrade, invalidates others
S	BusRd	S	-	Assert shared signal
S	BusRdX	I	-	Invalidate
E	PrRd	E	-	Hit, no bus activity
E	PrWr	M	-	Silent upgrade
E	BusRd	S	-	Assert shared signal
E	BusRdX	I	-	Invalidate
M	PrRd	M	-	Hit
M	PrWr	M	-	Hit
M	BusRd	S	Flush	Write-back, share

Current	Event	Next	Bus Action	Notes
M	BusRdX	I	Flush	Write-back, invalidate

Appendix B: Performance Metrics

Statistics collected per core:

Metric	Description
cycles	Total clock cycles until halt
instructions	Instructions executed (completed WB)
read hit	Cache hits on LW instructions
write hit	Cache hits on SW instructions
read miss	Cache misses on LW instructions
write miss	Cache misses on SW instructions
decode stall	Cycles stalled due to data hazards
mem stall	Cycles stalled due to cache misses

Appendix C: Design Decisions

1. **No Forwarding**: Simplifies hardware, all hazards resolved by stalling
2. **Branch in Decode**: Reduces branch penalty to 1 cycle (delay slot)
3. **Write-Back Cache**: Reduces memory traffic for write-heavy workloads
4. **Round-Robin Arbitration**: Ensures fairness among cores
5. **Direct-Mapped Cache**: Simple design, predictable behavior

---

End of Documentation