Multi-Core Processor Project Documentation

Introduction

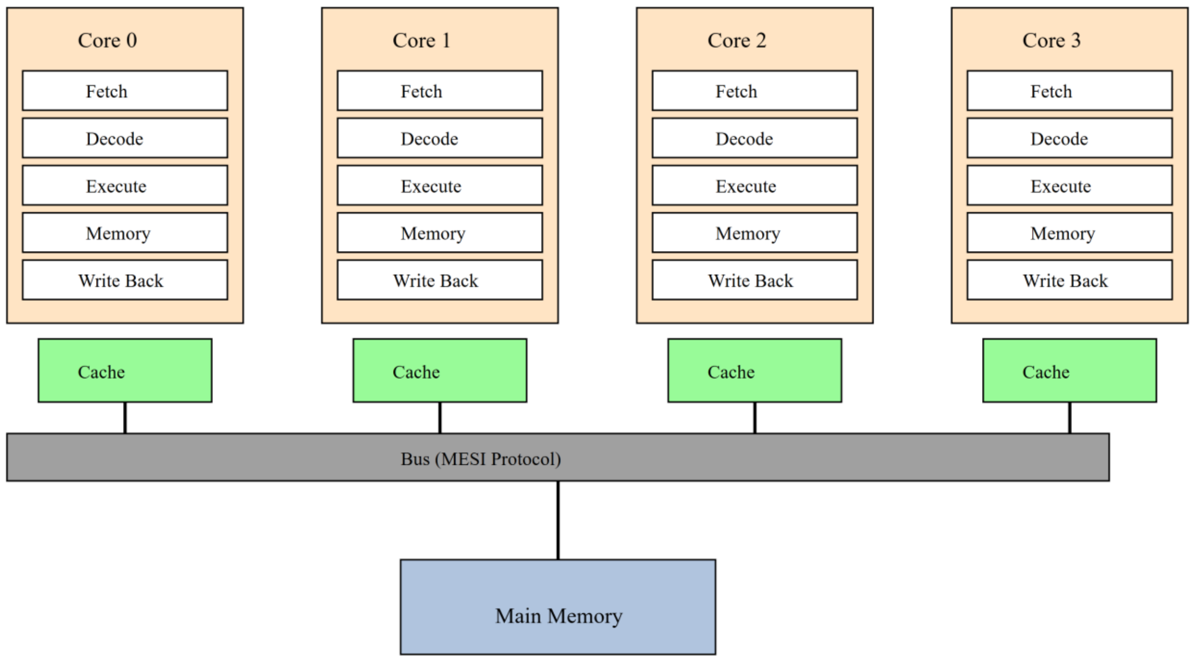
Project Overview

The Multi-Core Processor project is designed to simulate a processor with four parallel-running pipelined cores. Each core has its own private instruction memory (IMEM) and data cache, and the cores are connected via a BUS using the MESI cache coherence protocol to a shared main memory.

Objectives

- Implement a sim for a multi-core processor.
- Ensure data coherence between cores using the MESI protocol.
- Provide detailed output files for memory contents, register states, pipeline states, and BUS transactions.

System Architecture



Core Design

- **Registers**: Each core has 16 registers (32-bit wide). Register 1 is special and always contains the immediate field after sign extension. Register 0 is always zero.
- **Instruction Memory**: Each core has a private 32-bit wide, 1024-line deep SRAM instruction memory.

- **Pipeline**: Each core uses a 5-stage pipeline (Fetch, Decode, Execute, Mem, Write Back) with delay slots and no bypassing.

Memory and Cache Design

- **Data Cache**: Each core has a direct-mapped data cache with 256 words, a block size of 4 words, and a write-back, write-allocate policy.
- **Main Memory**: 20-bit address space, 2^20 words, accessed via a BUS with specific commands (BusRd, BusRdX, Flush).

Bus and MESI Protocol

- **Bus**: The BUS connects the cores to the shared main memory and handles data transfers using specific commands.
- **MESI Protocol**: The MESI (Modified, Exclusive, Shared, Invalid) protocol is used to maintain cache coherence between cores.

Detailed Component Descriptions

bus.c

Handles bus transactions, including writing to the bus trace file, initiating transactions, executing transactions, and snooping.

- **Functions**:
    - bus_trace_write: Writes the current state of the bus transaction to the bus trace file.
    - Bus_transaction: Handles the execution of bus transactions based on the current bus command. It manages different types of transactions such as BusRd, BusRdX, and Flush.
    - read_miss: Handles read misses by checking if any core has the requested data in the modified state. If so, it initiates a bus transaction to transfer the data.
    - write_miss: Similar to read_miss, this function handles write misses by checking for modified data in other cores' caches and initiating a bus transaction to transfer the data.
    - bus_call: Manages bus transaction requests from the main function. It checks if the bus is needed and if it is free or occupied.

core.c

Simulates the operation of a core, including pipeline stages (Fetch, Decode, Execute, Mem, Write Back) and handling stalls and hazards.

- **Functions**:
  - core: Simulates the entire operation of a core, including pipeline stages and handling stalls and hazards. It processes instructions through the pipeline stages, interacts with the data cache, and initiates bus transactions if needed.

coreFile.c

Handles writing the core's register file, data SRAM, tag SRAM, and statistics to their respective output files.

- **Functions**:
  - coreFiles: Writes the core's register file, data SRAM, tag SRAM, and statistics to their respective output files. It ensures that all necessary data is recorded for analysis and debugging.

header.h

Defines data structures and function prototypes used throughout the project.

- **Structures**:
  - ifidReg, idieReg, ieimReg, imwbReg: Pipeline registers for different stages.
  - transaction: Structure for bus transactions.
  - coreArgs: Structure for core-specific arguments and states.
  - bus_struct: Structure for the current state of the bus.
- **Function Prototypes**:
  - Prototypes for all functions used in the project.

initializator.c

Initializes the core arguments structure and sets up the necessary files for the core's operation.

- **Functions**:
  - initializator: Initializes the core arguments structure and sets up the necessary files for the core's operation. It prepares the core for simulation by setting initial values and creating output files.

mem_file_handler.c

Manages memory operations, including reading the initial memory contents, updating the memory during simulation, and writing the final memory contents to the output file.

- **Functions**:
    - sign_extension: Performs sign extension on a register value and formats it as a hexadecimal string of a specified length.
    - dmemin_read: Reads the initial contents of the main memory from the memin file and stores them in the main memory array.
    - max_mem_update: Updates the maximum written address in the main memory.
    - dmemout_write: Writes the final contents of the main memory to the memout file.

mesi.c

Implements the MESI protocol to maintain cache coherence between cores.

- **Functions**:
    - whoIsInM: Checks if any other cache has the requested block in the Modified state.
    - snooping: Performs snooping to check if the requested block exists in the cache and updates the bus shared status if it does.
    - mesi_update: Updates the MESI state of the caches based on the current bus transaction.

round_robin.c

Manages bus access using the Round Robin policy.

- **Functions**:
    - Round_Robin_update: Updates the bus arbitrator queue using the Round Robin policy.
    - Round_Robin_remove: Removes a core from the bus arbitrator queue after it has finished its bus transaction.
    - Bus_History_Queue_Update: Updates the bus history queue to reflect the most recent bus access.

sim.c

The main function that coordinates the overall operation of the sim, managing the main loop, core execution, bus transactions, and file I/O.

- **Functions**:
  - main: Initializes the cores, reads the initial memory contents, manages the main simulation loop, and writes the final memory contents to the output file.

Interactions and Data Flow

How Components Interact

- **Cores and Bus**: Cores interact with the bus to access the shared main memory. The bus handles data transfers and ensures coherence using the MESI protocol.
- **Pipeline Stages**: Each core processes instructions through its pipeline stages, interacting with its private instruction memory and data cache.
- **Memory Operations**: Memory operations are managed by the mem_file_handler.c file, which reads initial contents, updates memory during simulation, and writes final contents.

Data Flow Between Components

- **Instruction Fetch**: Instructions are fetched from the private instruction memory of each core.
- **Pipeline Execution**: Instructions are processed through the pipeline stages, with data being read from and written to the registers and data cache.
- **Bus Transactions**: Data transfers between cores and the main memory are managed by the bus, with coherence ensured by the MESI protocol.

Detailed Interaction Flow

1. **Initialization**:
   - The sim.c file's main function initializes the cores using the initializator.c file.
   - Each core's instruction memory is loaded from the respective imem files.
   - The main memory is initialized from the memin file using the mem_file_handler.c file.

2. **Core Execution**:
    - Each core executes instructions through its pipeline stages (Fetch, Decode, Execute, Mem, Write Back) as defined in core.c.
    - During the Fetch stage, instructions are fetched from the core's private instruction memory.
    - During the Decode stage, instructions are decoded, and hazards are checked.
    - During the Execute stage, arithmetic and logical operations are performed.
    - During the Mem stage, memory operations are performed, interacting with the data cache and potentially initiating bus transactions.
    - During the Write Back stage, results are written back to the registers.
3. **Bus Transactions**:
    - Bus transactions are managed by bus 1.c, which handles read and write misses, snooping, and updating the MESI state.
    - The round_robin.c file manages bus access using a Round Robin policy, ensuring fair access to the bus for all cores.
4. **Memory Operations**:
    - Memory operations are handled by mem_file_handler.c, which reads initial memory contents, updates memory during simulation, and writes final memory contents to the memout file.
5. **Output Generation**:
    - The final states of the registers, data SRAM, tag SRAM, and statistics for each core are written to their respective output files by coreFile.c.
    - The bus transactions are logged in the bustrace.txt file by bus 1.c.

Testing and Validation

Test Programs

- **Counter**: Increment a counter in main memory 128 times in a round-robin fashion across all cores.
- **AddSerial**: Sum two vectors of size 4096 on a single core.
- **AddParallel**: Sum two vectors of size 4096 across all four cores in parallel.

Expected Outputs

- **Memory Contents**: Final state of the main memory.
- **Register States**: Final state of the registers for each core.
- **Pipeline States**: State of the pipeline stages for each core at each clock cycle.
- **BUS Transactions**: Detailed log of bus transactions.

File Structure and Organization

Directory Structure

- **Input Files**: imem0.txt, imem1.txt, imem2.txt, imem3.txt, memin.txt
- **Output Files**: 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

Purpose of Each File

- **Input Files**:
  - imem0.txt, imem1.txt, imem2.txt, imem3.txt: These files contain the initial contents of the instruction memory for each core.
  - memin.txt: This file contains the initial contents of the main memory.
- **Output Files**:
  - memout.txt: This file contains the final contents of the main memory after the simulation.
  - regout0.txt, regout1.txt, regout2.txt, regout3.txt: These files contain the final state of the registers for each core.
  - core0trace.txt, core1trace.txt, core2trace.txt, core3trace.txt: These files contain the state of the pipeline stages for each core at each clock cycle.

  - bustrace.txt: This file contains a detailed log of bus transactions.
  - dsram0.txt, dsram1.txt, dsram2.txt, dsram3.txt: These files contain the final contents of the data SRAM for each core.
  - tsram0.txt, tsram1.txt, tsram2.txt, tsram3.txt: These files contain the final contents of the tag SRAM for each core.
  - stats0.txt, stats1.txt, stats2.txt, stats3.txt: These files contain statistics for each core, such as the number of cycles, instructions executed, cache hits and misses, and pipeline stalls.

Dependencies and Requirements

External Libraries and Tools

- **C Standard Library**: Used for standard input/output and memory operations.
- **Visual Studio**: Development environment for compiling and running the project.

Installation and Configuration

- **Setup**: Ensure all input files are in the same directory as the executable.
- **Compilation**: Use Visual Studio to compile the project and generate the executable.