Computer Architectures

Ron Mondshein -Yaniv Muallem -Eden Daya

Table of Contents

Introduction

This assignment involves designing a simulation of a multi-core processor architecture, integrating key components like cores, caches, bus system, and memory. It focuses on implementing critical concepts such as instruction pipelining, memory coherence, caching, and bus arbitration to emulate real-world system interactions.

The project highlights:

- **Instruction Execution:** Simulating instruction fetch, decode, execute, and memory stages.
- **Cache Management:** Designing a cache system with coherence protocols and efficient data access.
- **Bus Coordination:** Managing processor-memory communication via a bus controller.

We divided our project into distinct modules, each handling a specific system function, such as processing, memory management, caching, and bus communication. This modular approach ensures clarity and efficiency. In the following sections, we elaborate on each module, detailing its purpose, key structures, and functionality.
At the end of this file, we will provide detailed explanations of the assembly test files we wrote.

CacheController

General Description

The CacheController class manages the operations and behaviors of the cache memory system in the project. It ensures data consistency and optimizes data retrieval by coordinating between the cores and main memory. It simulate a module that is critical for performance, as it reduces the latency associated with accessing frequently used data.
The implementation of this class is divided into two files:

- **C File:** Contains the logic and function implementations.
- **Header File:** Provides the function prototypes and defines the structures and constants used in the class.

Main Variables and structions

CacheAddressFields

- **Description:** Defines the breakdown of a memory address for cache indexing.
- **Contents:**
    - Uint32_t **Tag:** Identifies the block in memory, 2 bits.
    - Unit32_t **Index:** Points to a specific cache line, 6 bits.
    - Unit32_t **Offset:** Refers to the exact byte within a block, 12 bits.
- **Purpose:** Facilitates address decoding for efficient cache lookup.

CacheAddressInfo

- **Description:** Contains metadata derived from a memory address.
- **Contents:**
    - Uint32_t **Address:** Raw 32-bit address
    - cacheAddressFields **Fields:** Decomposed fields
- **Purpose:** Helps to manage the mapping of memory addresses to cache lines.

MESIState

- **Description:** Represents the states in the MESI protocol (Modified, Exclusive, Shared, Invalid).
- **Contents:** Enum values corresponding to each MESI state.
- **Purpose**: Maintains coherence in multi-processor systems by tracking cache line states.

Cache_Id_enum

- **Description:** Identifies different caches, one for each core.
- **Contents:**
    - Enum values from CACHE_ID_CORE0 to CACHE_ID_CORE3.
- **Purpose:** Distinguishes between diffrenet caches of diffrenet cores.

TSRAMLine

- **Description:** Represents a single line in the Tag Store RAM (TSRAM).
- **Contents:**
    - Unit16_t **Tag**: Identifies memory block, 12 bits.
    - Unit8_t **Flags**: MESI state, 2 bits.
- **Purpose:** Stores metadata for cache line lookup.

DRAMLine

- **Description:** Represents a line in the DRAM cache.
- **Contents:**
    - Uint32_t **data**.
- **Purpose:** saves the data located inside the DRAM cache.

tracking_info

- **Description:** Contains runtime metadata for debugging or profiling.
- **Contents:**
    - uint32_t **read_hits**
    - uint32_t **read_misses**
    - uint32_t **write_hits**
    - uint32_t **write_misses**
- **Purpose:** Tracks performance metrics for analysis.

Cache_Data

- **Description:** Core structure holding cache content and metadata.
- **Contents:**
    - Cache_Id_enum **id:** Cache ID, between 0 to 3, 4 caches for 4 cores, one for each.
    - TSRAMLine **tsram[NUM_BLOCKS]:** Array of Tag and state SRAM
    - DRAMLine **dram[CACHE_SIZE]:** Array of Data SRAM
    - tracking_info **tracking_info:** Cache performance tracking
    - bool **isStalled:** Flag to indicate if the cache is stalled
- **Purpose:** Implements the main storage and information of the cache system.

Main Functions

Static Functions

static bool **is_cache_busy**(Cache_Data* cache_data)

- **Purpose:** Checks whether the cache is currently busy handling another operation.
- **Input:**
    - cache_data: Pointer to the cache data structure.
- **Output:**
    - true if the cache is busy, otherwise false.

static bool **shared_or_modified_handler**(void *cache, bus_transaction* transaction, bool* is_modified)

- **Purpose:** Handles cache states when a shared or modified transaction occurs.
- **Input:**
    - cache: Pointer to the cache object.
    - transaction: Pointer to the bus transaction structure.
    - is_modified: Boolean flag indicating if the cache line is modified.
- **Output:**
    - true if the operation succeeds, otherwise false.

static bool **snooping_handler**(void* cache, bus_transaction* transaction, uint8_t address_offset)
Purpose: Handles snooping of cache lines to maintain coherence.

- **Input:**
    - cache: Pointer to the cache object.
    - transaction: Pointer to the bus transaction.
    - address_offset: Offset of the address being snooped.
- **Output:**
    - true if snooping is successful, otherwise false.

static bool **snooped_transaction**(Cache_Data* cache_data, TSRAMLine* tsram_line, bus_transaction* transaction, CacheAddressInfo c_addr, uint8_t address_offset)

- **Purpose:** Processes a snooped transaction for a specific cache line. Update the mesi next state.
- **Input:**
    - cache_data: Pointer to the cache data structure.
    - tsram_line: Pointer to the tag store line.
    - transaction: Pointer to the bus transaction.
    - c_addr: Cache address information.
    - address_offset: Offset of the address being snooped.
- **Output:**
    - true if the snooped transaction is handled, otherwise false.

static bool **cache_response_handle**(void* data, bus_transaction* transaction, uint8_t* address_offset)

- **Purpose:** Manages cache responses for bus transactions.
  Handle both single and multiful words (when a block need to be sent).
  Set the MESI state to the relvent state, when necessary.
- **Input:**
    - data: Pointer to the cache data or context.
    - transaction: Pointer to the bus transaction.
    - address_offset: Pointer to the address offset.
- **Output:**
    - true if the response is successfully handled and completed, otherwise, like when more data need to be sent, false.

static void **flush_data**(Cache_Data* cache_data, bus_transaction* transaction)

- **Purpose:** make a transaction for a flush case when a cache line's data need to be flushed back to memory.
- **Input:**
    - cache_data: Pointer to the cache data structure.
    - transaction: Pointer to the bus transaction.
- **Output:** None.

static **Cache_Id_enum MSEI_invalid** (Cache_Data* cache_data, bus_transaction* transaction);

- **Purpose:** Handles transitions for the Invalid state in the MESI protocol.
- **Input:**
    - cache_data: Pointer to the cache data structure.
    - transaction: Pointer to the bus transaction.
- **Output:** Returns MESI_STATE_INVALID

static **Cache_Id_enum MSEI_shared** (Cache_Data* cache_data, bus_transaction* transaction);

- **Purpose:** Handles transitions for the Shared state in the MESI protocol.
- **Input:**
    - cache_data: Pointer to the cache data structure.
    - transaction: Pointer to the bus transaction.
- **Output:** returns MESI_STATE_INVALID or MESI_STATE_SHARED according to the bus_cmd.

static **Cache_Id_enum MSEI_exclusive** (Cache_Data* cache_data, bus_transaction* transaction);

- **Purpose:** Handles transitions for the Exclusive state in the MESI protocol.
- **Input:**
    - cache_data: Pointer to the cache data structure.
    - transaction: Pointer to the bus transaction.
- **Output:** returns MESI_STATE_INVALID or MESI_STATE_SHARED or MESI_STATE_EXCLUSIVE according to the bus_cmd.

static **Cache_Id_enum MSEI_modified** (Cache_Data* cache_data, bus_transaction* transaction);

- **Purpose:** Handles transitions for the Modified state in the MESI protocol.
- **Input:**
    - cache_data: Pointer to the cache data structure.
    - transaction: Pointer to the bus transaction.
- **Output:** returns MESI_STATE_INVALID or MESI_STATE_SHARED or MESI_STATE_MODIFIED according to the bus_cmd and make a flush transaction if necessary.

static states_machine **state_handler**

- **Purpose:** A function pointer to handle state transitions in the cache coherence protocol.
- **Input:**
    - Varies based on the specific state of the state machine.
- **Output:** The updated state of the cache line after processing, according to the 4 functions above.

static bool **readHit**(Cache_Data* cache_data, CacheAddressInfo addr, uint32_t* data, bool miss_occurred_read)

- **Purpose:** Retrieves data from the cache for a read hit, it updates the read hit statistics accordingly.
- **Input:**
    - cache_data: Pointer to the cache data structure.
    - addr: Address information containing index, tag, and offset fields.
    - data: Pointer to the location where the retrieved data will be stored.
    - miss_occurred_read: Flag indicating if a read miss has occurred.
- **Output:**
    - Returns true if a read miss occurred, otherwise false.

static void **handle_dirty_block**(Cache_Data* cache_data, TSRAMLine* tsram_line, CacheAddressInfo addr)

- **Purpose:** Handles eviction of a dirty cache block.
- **Description:** This function checks if the block being evicted is in a MODIFIED state. If so, it constructs a transaction to flush the block's data to memory and adds it to the bus for processing.
- **Input:**
    - cache_data: Pointer to the cache data structure.
    - tsram_line: Pointer to the cache's TSRAM line corresponding to the address.
    - addr: Address information of the block to be evicted.
- **Output:** None.

static void **handle_transaction**(Cache_Data* cache_data, CacheAddressInfo addr, cmd_on_the_bus b_cmd)

- **Purpose:** Initiates a bus transaction for memory operations.
- **Description:** This function creates and queues a bus transaction for reading or writing data from/to memory. It sets the transaction type, address, and associated core ID.
- **Input:**
    - cache_data: Pointer to the cache data structure.
    - addr: Address information for the transaction.
    - b_cmd: Command type (e.g., busRd or busRdX) for the transaction.
- **Output:** None.

Public Functions

void **CacheController_Init**(Cache_Data *cache_data, Cache_Id_enum id)

- **Purpose:** Initializes the cache with a given ID and prepares it for operation.
- **Description:** This function allocates memory for the cache, sets default values for its parameters and initialize it connection to the bus.
- **Input:**
    - cache_data: Pointer to the cache data structure to initialize.
    - id: Cache ID from 0 to 3.
- **Output:** None.

void **Cache_InitializeBusCallbacks**(void);

- **Purpose:** Sets up the necessary bus callbacks for communication between the cache and the bus.
- **Input:** None.
- **Output:** None.

void **print_Cache_Data**(Cache_Data* cache_data, FILE* file_dram, FILE* file_tsram);

- **Purpose:** Outputs the current state of the cache data, including DRAM and TSRAM contents.
- **Description:** Logs the contents of the cache, helping in debugging and analysis. The data in DRAM and TSRAM is printed to the specified file streams for detailed examination.
- **Input:**
  - cache_data: Pointer to the cache data structure to print.
  - file_dram: File pointer for logging DRAM data.
  - file_tsram: File pointer for logging TSRAM data.
- **Output:** None.

bool **Write_Data_to_Cache**(Cache_Data* cache_data, uint32_t address, uint32_t data);

- **Purpose:** Writes a data block to the cache at a specified memory address.
  Description: This function first checks if the cache is busy, after it, it checks if the memory address exists in the cache. If it is a cache hit, the function updates the data. If it is a cache miss, it loads the data into the cache line and writes the new value. It also handles dirty block if necessary.
- **Input:**
  - cache_data: Pointer to the cache data structure.
  - address: Memory address where the data will be written.
  - data: Data to write.
- **Output:** true if the operation succeeds, otherwise false.

bool **Read_Data_from_Cache**(Cache_Data* cache_data, uint32_t address, uint32_t* data);

- **Purpose:** Reads a data block from the cache at a specified memory address.
  Description: This function first checks if the cache is busy. Then, retrieves data from the cache if it exists (cache hit) or fetches it from main memory if it does not (cache miss). It also handles dirty block if necessary.
- **Input:**
  - cache_data: Pointer to the cache data structure.
  - address: Memory address to read from.
  - data: Pointer to a variable to store the retrieved data.
- **Output:** true if the data is successfully read, otherwise false.

ProcessorCore

General Description

The ProcessorCore class represents the central processing unit of the simulated system, responsible for executing instructions and managing core-level operations. It handles the stages of instruction execution, such as fetch, decode, execute, and write-back, while interacting with other components like the pipeline, cache, and memory. This class is pivotal in ensuring efficient instruction processing and coordination within the system.
The implementation of this class is divided into two files:

- **C File:** Contains the logic and function implementations.
- **Header File:** Provides the function prototypes and defines the structures and constants used in the class.

Main Variables and structions

tracking_info_core

- **Description:** Tracks the performance and execution statistics of the processor core.
- **Contents:**
  - uint32_t **cycles**: Counts the total number of cycles the core has executed.
  - uint32_t **instructions**: Tracks the total number of instructions executed by the core.
- **Purpose:** Provides essential performance metrics for analyzing the efficiency and behavior of the processor core during simulation.

ProcessorCore

- **Description:** Represents a single core in the simulated processor, handling instruction execution and communication with other system components.
- **Contents:**
  - uint32_t **coreId**: Identifies the core, allowing for differentiation in multi-core setups, between o to 3.
  - uint32_t **pc**: Tracks the program counter, pointing to the current instruction, takes 10 bits as the address space is 1K words long.
  - uint32_t **registers[REGISTERCOUNT]:** Stores the 16 general-purpose registers, each 32 bits.
  - uint32_t **instruction_memory[INSTRUCTIONMEMORYSIZE]:** Holds the instruction memory (1K words).
  - CoreFileHandles **fileHandles**: Manages file handles for trace and memory outputs.
  - Pipe_fig **pipelineController**: Coordinates the pipeline stages for instruction execution.
  - bool **isHalted**: Indicates whether the core is halted.
  - tracking_info_core **tracking_info_core**: Embedded structure to track cycles and instructions.
- **Purpose:** Simulates the behavior of a single processor core, including execution, memory interaction, and performance tracking.

Main Functions

Static Functions

static void **Print_tracking_info**(ProcessorCore* core)

- **Purpose:** Logs the performance statistics of the processor core to an output file.
- **Description:** This function writes metrics such as cycles, instructions executed, cache hits/misses, and stalls into a designated statistics file. These metrics provide insights into the core's performance and behavior during execution.
- **Input:**
    - core: Pointer to the ProcessorCore structure containing the tracking information.
- **Output:** None.

static void **Print_registers**(ProcessorCore* core)

- **Purpose:** Outputs the current values of the processor's registers to a file.
- **Description:** This function iterates through the core's registers and writes their values into a file. It is primarily used to capture the state of the registers for debugging and analysis.
- **Input:**
    - core: Pointer to the ProcessorCore structure containing the registers.
- **Output:** None.

static int **InstMem_init**(ProcessorCore* core)

- **Purpose:** Initializes the instruction memory of the processor core.
- **Description:** This function loads instructions from an input file into the core's instruction memory. It continues loading until the memory is full or the end of the file is reached.
- **Input:**
    - core: Pointer to the ProcessorCore structure whose instruction memory is being initialized.
- **Output:** The number of instructions successfully loaded into the instruction memory.

static void **write_trace**(ProcessorCore* core, uint32_t* reg)

- **Purpose:** Writes the trace of the processor's execution to a file.
- **Description:** This function logs the execution trace for debugging, including the current cycle, pipeline states, and register values. It is used to track the core's activity across cycles.
- **Input:**
    - core: Pointer to the ProcessorCore structure.
    - reg: Pointer to the previous state of the core's registers.
- **Output:** None.

static void **write_trace_reg**(ProcessorCore* core, uint32_t* reg)

- **Purpose:** Logs the register values during the execution trace.
- **Description:** This function writes the values of all mutable registers into the execution trace file, providing a snapshot of the core's register state.
- **Input:**
    - core: Pointer to the ProcessorCore structure.
    - reg: Pointer to the registers to be logged. If NULL, the current core register values are used.
- **Output:** None.

static void **update_tracking_info**(ProcessorCore* core)

- **Purpose:** Updates the core's performance statistics.
- **Description:** This function increments the cycle count and, if no stalls occur, the instruction count. It tracks the execution progress and provides data for performance analysis.
- **Input:**
  - core: Pointer to the ProcessorCore structure containing the tracking information.
- **Output:** None.

Public Functions

void **ProcessorCore_Init**(ProcessorCore* core, uint32_t coreId)

- **Purpose:** Initializes the processor core with its default state and configurations.
- **Description:** This function sets up the program counter, initializes the core registers and instruction memory, and prepares the pipeline and cache. It also assigns a unique ID to the core and loads instructions from a file if available.
- **Input:**
  - core: Pointer to the ProcessorCore structure to initialize.
  - coreId: Unique identifier for the processor core.
- **Output:** None.

void **core_run_single_cycle**(ProcessorCore* c)

- **Purpose:** Simulates a single cycle of operation for the processor core.
- **Description:** This function executes one cycle of the processor, updating pipeline stages, processing instructions, and writing execution traces. It handles pipeline flushing and halts the core if necessary.
- **Input:**
  - c: Pointer to the ProcessorCore structure.
- **Output:** None.

void **Core_Shutdown**(ProcessorCore* core)

- **Purpose:** Gracefully shuts down the processor core.
- **Description:** This function outputs the final state of the registers, cache data, and performance statistics to the appropriate files. It ensures all data is saved and prepares the core for termination.
- **Input:**
  - core: Pointer to the ProcessorCore structure to shut down.
- **Output:** None.

bool **core_is_halted**(ProcessorCore* core)

- **Purpose:** Checks if the processor core is halted.
- **Input:**
  - core: Pointer to the ProcessorCore structure.
- **Output:** Returns true if the core is halted, otherwise false.

MainMemory

General Description

The MainMemory module represents the primary memory of the multicore processor simulator. It is responsible for storing and managing data used during program execution. The memory is implemented as a flat address space of size 2^{20} bytes, supporting read and write operations through transactions initiated by the bus controller. Additionally, it tracks memory performance and provides mechanisms for initializing, printing, and processing memory operations.
This module is implemented across two files:

- **C File (MainMemory.c)**: Contains the implementation of the main memory functions, such as initialization, transaction handling, and memory printing.
- **Header File (MainMemory.h)**: Provides the definitions, constants, and function prototypes needed to interact with the main memory.

Main Variables and structions

AddressFields

- **Description:** Represents the breakdown of a memory address into block and offset components.
- **Contents:**
  - uint8_t **offset**: 2 bits used to identify the word within a block (4 words per block).
  - uint32_t **block**: 18 bits used to identify the block in the memory.
- **Purpose:** Enables efficient decoding and management of memory addresses by separating the offset and block fields.

MemoryAddress

- **Description:** Represents a memory address as either a raw value or decomposed into its constituent fields.
- **Contents:**
  - uint32_t **address**: Raw 20-bit memory address.
  - AddressFields **fields**: Decomposed representation of the address.
- **Purpose:** Provides flexibility in managing memory addresses, allowing for both raw and field-based access.

mainMemory[MAIN_MEMORY_SIZE]

- **Description:** The primary memory storage for the system, implemented as a flat array.
- **Contents:** Array of uint32_t values with a size of 2^{20} bytes.
- **Purpose:** Serves as the main storage for data, accessible through memory transactions.

gIsMemoryBusy

- **Description:** A flag indicating whether the memory is currently busy handling a transaction.
- **Contents:** Boolean value initialized to false.
- **Purpose:** Prevents simultaneous memory operations and manages transaction delays.

Main Functions

Static Functions

static size_t countMemoryLines(void)

- **Purpose:** Counts the number of non-empty lines in the main memory.
- **Description:** This function traverses the memory array in reverse order to identify the highest non-zero entry, effectively counting the used lines.
- **Input:** None.
- **Output:** The total number of used memory lines (size_t).

static bool initialize_memory_transaction(bool direct_transaction)

- **Purpose:** Initializes the state for a memory transaction.
- **Description:** This function sets up the transaction delay and marks the memory as busy. It applies different delays for direct or indirect transactions.
- **Input:**
    - direct_transaction: Boolean indicating if the transaction is direct.
- **Output:**
    - true if the memory is busy, otherwise false.

static bool process_memory_command(bus_transaction* transaction)

- **Purpose:** Processes a bus transaction command for the memory, depends on the bus_cmd value.
- **Input:**
    - transaction: Pointer to the bus transaction structure.
- **Output:**
    - true if the command is successfully processed, otherwise false.

static void valuesToChange(bus_transaction* transaction)

- **Purpose:** Modifies the bus transaction fields for memory operations.
- **Description:** This function updates the transaction's origin ID, command type, and data fields to align with memory operations.
- **Input:**
    - transaction: Pointer to the bus transaction structure.
- **Output:** None.

static bool bus_transaction_handler(bus_transaction* packet, bool direct_transaction)

- **Purpose:** Handles bus transactions directed to the main memory.
- **Description:** This function processes memory-related bus commands, applying transaction delays and performing read or write operations. It ensures proper handling of direct and indirect transactions.
- **Input:**
    - packet: Pointer to the bus transaction structure.
    - direct_transaction: Boolean indicating if the transaction is direct.
- **Output:**
    - true if the transaction is completed or in progress, otherwise false.

Public Functions

void MainMemoryInit()

- **Purpose:** Initializes the main memory for the simulator.
- **Description:** This function sets all memory locations to zero and loads initial memory values from an input file (MemIn). It also registers the memory callback function with the bus controller to enable memory transactions.
- **Input:** None.
- **Output:** None.

void MainMemoryPrint(FILE* file)

- **Purpose:** Prints the contents of the main memory to an output file.
- **Description:** This function iterates through the memory and writes its contents in hexadecimal format to the specified file. It only prints used lines of the memory for efficient output.
- **Input:**
  - file: Pointer to the file where the memory contents will be printed.
- **Output:** None.

FilesManager

General Description

The FilesManager module is responsible for managing all file-related operations within the simulation. This includes opening, validating, and closing input and output files required for memory initialization, trace logs, and statistical outputs. By centralizing file management, this module ensures that all files are properly handled and accessible throughout the simulation.
The FilesManager module is implemented in two files:

- **C File (FilesManager.c)**: Contains the implementation of functions for file handling, including opening and closing files and validating file operations.
- **Header File (FilesManager.h)**: Provides the structure definitions, global variables, and function prototypes for file management operations.

Main Variables and structions

CoreFileHandles

- **Description:** Represents the file handles for input and output files associated with each processor core.
- **Contents:**
  - FILE* **instructionMemoryFile**: File for instruction memory input.
  - FILE* **registerOutputFile**: File for register values output.
  - FILE* **executionTraceFile**: File for execution trace logs.
  - FILE* **dataCacheFile**: File for data cache content (DSRAM).
  - FILE* **tagCacheFile**: File for tag cache content (TSRAM).
  - FILE* **coreStatsFile**: File for core statistics output.
- **Purpose:** Provides a centralized structure to manage file operations for each core, ensuring efficient handling of file-based data.

extern CoreFileHandles **coreFileHandlesArray[NUM_OF_CORES]**

- **Description:** An array of CoreFileHandles structures, one for each processor core.
- **Contents:**
  - Each element in the array is a CoreFileHandles instance representing file handles for a specific core.
- **Purpose:** Enables separate file management for each core in the simulation.

FILE* **MemIn**

- **Description:** File pointer for the memory initialization file.
- **Purpose:** Used to load initial data into the main memory during simulation initialization.

FILE* **MemOut**

- **Description:** File pointer for the memory output file.
- **Purpose:** Stores the memory's state at the end of the simulation for verification and debugging.

FILE* **BusTrace**

- **Description:** File pointer for the bus trace log.
- **Purpose:** Tracks bus activity, including memory transactions and data transfers, for debugging and analysis.

Main Functions

Static Functions

static FILE* **openFile**(bool useRelativePath, const char* relativePath, const char* argvPath, const char* mode)

- **Purpose:** Opens a file with the specified path and mode.
- **Description:** This function checks if a relative path or a command-line argument is to be used for the file path and opens the file in the requested mode (read, write, etc.). It ensures that the correct path is used for file operations.
- **Input:**
  - useRelativePath: Boolean indicating whether to use a relative path.
  - relativePath: Path to the file if a relative path is used.
  - argvPath: Path provided through command-line arguments.
  - mode: Mode in which the file is to be opened (e.g., "r", "w").
- **Output:**
  - Returns a pointer to the opened file or NULL if the file could not be opened.

static bool **fileCoreFailedToOpen**(CoreFileHandles* coreFileHandles, int core)

- **Purpose:** Checks if any core-specific files failed to open.
- **Description:** This function verifies that all file handles associated with a specific core are valid. If any file fails to open, it prints an error message indicating the problematic file and core number.
- **Input:**
  - coreFileHandles: Pointer to the CoreFileHandles structure for the specified core.
  - core: Integer representing the core number.
- **Output:**
  - Returns true if any file failed to open, otherwise false.

static bool **fileFailedToOpen**()

- **Purpose:** Checks if any global or core-specific files failed to open.
- **Description:** This function validates the file pointers for both global and core-specific files. It identifies and logs any errors, ensuring that all required files are available before simulation begins.
- **Input:** None.
- **Output:**
  - Returns true if any file failed to open, otherwise false.

Public Functions

int **OpenRequiredFiles**(char* argv[], int argc)

- **Purpose:** Opens all required files for the simulation.
- **Description:** This function processes command-line arguments or uses default relative paths to open global and core-specific files. It validates that all necessary files are successfully opened and logs errors for any that fail to open. If all files are successfully opened, the simulation is ready to proceed.
- **Input:**
  - argv[]: Array of command-line arguments specifying file paths.
  - argc: Number of command-line arguments provided.

- **Output:**
    - Returns 0 if all files are successfully opened.
    - Returns 1 if one or more files fail to open.

void **closeFiles**()

- **Purpose:** Closes all opened files.
- **Description:** This function iterates through all global and core-specific file pointers, closing them to ensure no file remains open after the simulation completes. It finalizes file operations for memory, trace logs, and statistics.
- **Input:** None.
- **Output:** None.

BusController

General Description

The **BusController** module manages communication between processor cores, cache controllers, and main memory in the simulated multicore system. It implements a centralized bus arbitration mechanism to ensure orderly transaction processing and memory coherence. This module is vital for coordinating data transfers and handling shared resources across cores.
This module is implemented across two files:

- **C File (BusController.c):** Contains the implementation of bus-related functionality, including transaction queuing, arbitration, and interaction with caches and memory.
- **Header File (BusController.h):** Declares function prototypes, structures, and constants for bus operations.

Main Variables and structions

Bus_transaction_caller

- **Description:** Enumerates the entities that can initiate a bus transaction.
- **Contents:**
  - *core0*: Core 0 initiator.
  - *core1*: Core 1 initiator.
  - *core2*: Core 2 initiator.
  - *core3*: Core 3 initiator.
  - *main_memory*: Main memory initiator.
  - *invalid_caller = 0xFFFF*: Indicates an invalid caller.

cmd_on_the_bus

- **Description:** Enumerates the commands that can be executed on the bus.
- **Contents:**
  - *no_cmd*: No command.
  - *busRd*: Bus read command.
  - *busRdX*: Bus read-exclusive command.
  - *flush*: Flush command.

bus_transaction

- **Description:** Represents a transaction on the bus.
- **Contents:**
  - **Bus_transaction_caller** *original_caller*: The initial originator of the transaction.
  - **Bus_transaction_caller** *origid*: The current originator of the transaction.
  - **cmd_on_the_bus** *bus_cmd*: The command being executed.
  - **uint32_t** *bus_addr*: The address involved in the transaction.
  - **uint32_t** *bus_data*: Data associated with the transaction.
  - **bool** *bus_shared*: Indicates if the data is shared across cores.

Bus_core_cache

- **Description:** Represents the cache interface of a core.
- **Contents:**
  - **int** *core_id*: Identifier for the core.
  - **void*** *bus_cache_data*: Pointer to the core's cache data.

AddressFields

- **Description:** Represents the breakdown of a memory address into block and offset components. Facilitates memory address decoding for transaction processing
- **Contents:**
    - **uint8_t** *offset*: 2 bits identifying the word within a block (4 words per block).
    - **uint32_t** *block*: 18 bits identifying the memory block.

MemoryAddress

- **Description:** Represents a memory address as a union. Enables flexibility in handling memory addresses during transactions.
- **Contents:**
    - **uint32_t** *address*: Raw 20-bit memory address.
    - **AddressFields** *fields*: Decomposed address components.

state_of_transaction

- **Description:** Enumerates the states of a bus transaction.
- **Contents:**
    - *idle*: The transaction is idle.
    - *wait_cmd*: The transaction is waiting for a command.
    - *operation*: The transaction is in progress.
    - *finally*: The transaction is in its final stage.

queue_for_bus

- **Description:** Represents a node in the FIFO queue for bus transactions.
- **Contents:**
    - **bus_transaction** *item*: The transaction stored in the queue node.
    - **struct queue_for_bus*** *prev*: Pointer to the previous queue node.
    - **struct queue_for_bus*** *next*: Pointer to the next queue node.

gSharedData_Callback

- **Description:** Callback function for detecting shared data.
- **Type: static SharedData_Callback** *gSharedData_Callback*

bool gIsBusTransactionActive

- **Description:** Indicates if a bus transaction is currently active.

state_of_transaction gTransactionStatePerCore[NUM_OF_CORES]

- **Description:** Array tracking the transaction state for each core.

queue_for_bus* head_of_queue

- **Description:** Pointer to the head of the transaction queue.

queue_for_bus* tail_of_queue

- **Description:** Pointer to the tail of the transaction queue.

Main Functions

Static Functions

static void print_to_bustrace(bus_transaction TransactionPacket)

- **Purpose:** Logs bus transaction details for analysis.
- **Input: TransactionPacket:** The transaction to log.

static bool is_any_cache_snoop(bus_transaction* TransactionPacket)

- **Purpose:** Checks if any cache responds to the current transaction.
- **Input: TransactionPacket**: Pointer to the transaction being processed.
- **Output:** true if a response is detected, otherwise false.

static bool is_shared_line(bus_transaction* TransactionPacket, bool* is_data_modified)

- **Purpose:** Determines if the transaction involves shared data across cores.
- **Input:**
    - **TransactionPacket**: Pointer to the transaction being processed.
    - **is_data_modified**: Pointer to a flag indicating data modification.
- **Output:** true if shared data is detected, otherwise false.

Public Functions

void Bus_InitializeCache(Bus_core_cache cache_interface)

- **Purpose:** Registers a cache configuration with the bus controller.
- **Input: cache_interface:** Cache configuration for a specific core.

void ConfigureCacheCallbacks_for_bus(SharedData_Callback signal_callback,
SnoopingCache_Callback snooping_callback,
GetCacheResponse_Callback response_callback)

- **Purpose:** Configures cache callback functions for communication.
- **Inputs:**
    - **signal_callback:** Callback for shared data detection.
    - **snooping_callback:** Callback for cache snooping.
    - **response_callback:** Callback for cache response handling.

void ConfigureMemoryCallback_for_bus(Mem_Callback callback)

- **Purpose**: Registers the memory callback function with the bus.
- **Input**: **callback**: The memory callback function.

void AddTransaction_to_bus(bus_transaction packet)

- **Purpose:** Adds a transaction to the bus queue and updates its state.
- **Input: transaction**: The transaction to add.

bool IsBusInTransaction(Bus_transaction_caller originator)

- **Purpose:** Checks if the bus is actively handling a transaction.
- **Input: initiator**: The ID of the transaction originator.
- **Output:** true if the bus is active, otherwise false.

bool IsBusWaitForTransaction(Bus_transaction_caller originator)

- **Purpose:** Checks if the bus is waiting for a transaction.
- **Input: initiator**: The ID of the transaction originator.
- **Output:** true if the bus is waiting, otherwise false.

void Run_Bus_Iteration(void)

- **Purpose:** Processes the next iteration of the bus transaction cycle.
- **Description:** Handles transaction queuing, cache snooping, and memory interaction. Updates the state of the ongoing transaction and logs activity.

callback Functions

SharedData_Callback

- **Type:** *typedef bool (*SharedData_Callback)(void* bus_cache_data, bus_transaction* packet, bool* is_modified)*
- **Purpose:** Detects if a transaction involves shared data.
- **Inputs:**
  - **void*** *bus_cache_data*: Pointer to the cache data.
  - **bus_transaction*** *packet*: Pointer to the transaction being processed.
  - **bool*** *is_modified*: Pointer to a flag indicating whether the data is modified.
- **Output:** Returns true if the data is shared, otherwise false.

SnoopingCache_Callback

- **Type:** *typedef bool (*SnoopingCache_Callback)(void* bus_cache_data, bus_transaction* packet, uint8_t address_offset)*
- **Purpose:** Handles cache snooping for a transaction.
- **Inputs:**
  - **void*** *bus_cache_data*: Pointer to the cache data.
  - **bus_transaction*** *packet***:** Pointer to the transaction being snooped.
  - **uint8_t** *address_offset:* The offset within the address being snooped.
- **Output**: Returns true if the snooping is successful, otherwise false.

GetCacheResponse_Callback

- **Type:** *typedef bool (*GetCacheResponse_Callback)(void* bus_cache_data, bus_transaction* packet, uint8_t* address_offset)*
- **Purpose:** Processes a cache response for the bus transaction.
- **Inputs:**
  - **void*** *bus_cache_data*: Pointer to the cache data.
  - **bus_transaction*** *packet*: Pointer to the transaction.
  - **uint8_t*** address_offset: Pointer to the address offset for block-wise processing.
- **Output**: Returns true if the cache response is successfully handled, otherwise false.

Mem_Callback

- **Type:** *typedef bool (*Mem_Callback)(bus_transaction* packet, bool direct_transaction)*
- **Purpose:** Handles memory transactions based on bus commands.
- **Inputs:**
    - **bus_transaction*** *packet*: Pointer to the transaction being processed.
    - **bool direct_transaction**: Indicates whether the transaction is direct.
- **Output**: Returns true if the memory transaction is successfully handled, otherwise false.

PipelineController

General Description

The PipelineController module is responsible for managing the stages of instruction execution within the processor pipeline. It handles the sequential flow of instructions through the pipeline stages: fetch, decode, execute, memory access, and write-back. The module ensures correct execution while handling hazards, stalls, and pipeline flushing.
The PipelineController module is implemented across two files:

- **C File (PipelineController.c)**: Contains the implementation of pipeline functionality, including stage-specific logic, hazard detection, and performance tracking.
- **Header File (PipelineController.h)**: Declares the data structures, function prototypes, and constants required for pipeline management.

Main Variables and structions

Pipe_figstate

- **Description:** Represents the different stages of the pipeline.
- **Contents:**
  - FETCH: Instruction fetch stage.
  - DECODE: Instruction decode stage.
  - EXECUTE: Instruction execution stage.
  - MEM: Memory access stage.
  - WRITE_BACK: Write-back stage.
  - PIPE_SIZE: Total number of stages in the pipeline.
- **Purpose:** Provides an enumeration of pipeline stages to manage the instruction flow systematically.

Pipe_instruction_stage

- **Description:** Represents a single stage in the pipeline.
- **Contents:**
  - Pipe_figstate **state**: Current state of the stage.
  - uint16_t **pc**: Program counter associated with the instruction in this stage.
  - Format_of_instruction **instruction**: The instruction being processed in this stage.
  - uint32_t **result_of_execution**: Result produced by the execution of the instruction.
  - void **(*operation)(OpcodeParams* params):** Pointer to the operation associated with the instruction.
- **Purpose:** Contains the details and execution context of an individual pipeline stage.

Pipe_Stats

- **Description:** Tracks pipeline performance statistics related to stalls.
- **Contents:**
  - uint32_t **stalls_in_decode**: Number of decode stage stalls encountered.
  - uint32_t **stalls_in_mem**: Number of memory stage stalls encountered.
- **Purpose:** Provides performance metrics for analyzing the efficiency and behavior of the pipeline during simulation.

Pipe_fig

- **Description:** Represents the entire pipeline, including its stages and operational state.
- **Contents:**
    - bool **is_halted**: Indicates whether the pipeline is halted.
    - bool **data_stall**: Flag for data hazard-induced stalls.
    - bool **mem_stall**: Flag for memory access-induced stalls.
    - uint32_t* **insturcionts_pnt**: Pointer to the instruction memory.
    - uint32_t* **regs_pnt**: Pointer to the core's register file.
    - Cache_Data **data_in_cache**: Cache data used by the pipeline.
    - Pipe_instruction_stage **stages_in_pipe[PIPE_SIZE]:** Array representing the stages of the pipeline.
    - OpcodeParams **params_of_op**: Parameters required for executing an operation.
    - Pipe_Stats **stats**: Tracks pipeline performance metrics.
- **Purpose:** Serves as a comprehensive structure for managing and monitoring the pipeline's execution, enabling efficient instruction flow and handling hazards or stalls.

Main Functions

Static Functions

static void fetch(Pipe_fig* pipeline)

- **Purpose:** Handles the fetch stage of the pipeline.
- **Description:** Fetches the instruction from the instruction memory using the program counter (PC) and increments the PC if no data stall exists.
- **Input:**
    - pipeline: Pointer to the Pipe_fig structure representing the pipeline.
- **Output:** None.

static void decode(Pipe_fig* pipeline)

- **Purpose:** Decodes the fetched instruction in the decode stage.
- **Description:** Extracts the opcode from the fetched instruction, identifies the operation, and prepares the parameters for execution. Handles branch instructions by prepare their registers parameters in this stage.
- **Input:**
    - pipeline: Pointer to the Pipe_fig structure representing the pipeline.
- **Output:** None.

static void execute(Pipe_fig* pipeline)

- **Purpose:** Executes the instruction in the execution stage.
- **Description:** Performs arithmetic, logical, and other operations as defined by the instruction. Prepares the parameters and executes the operation using the opcode handler.
- **Input:**
    - pipeline: Pointer to the Pipe_fig structure representing the pipeline.
- **Output:** None.

static void mem(Pipe_fig* pipeline)

- **Purpose:** Handles memory operations in the memory stage.
- **Description:** Performs memory access (read/write) based on the opcode (e.g., LW or SW) and manages stalls if the cache access is not successful.
- **Input:**
    - pipeline: Pointer to the Pipe_fig structure representing the pipeline.
- **Output:** None.

static void writeback(Pipe_fig* pipeline)

- **Purpose:** Writes the result of the executed instruction back to the register file in the write-back stage.
- **Description:** Updates the destination register with the result produced in the execution stage or handles special cases like JAL where the program counter is updated.
- **Input:**
    - pipeline: Pointer to the Pipe_fig structure representing the pipeline.
- **Output:** None.

static void execute_pipe_stages(Pipe_fig* pipeline)

- **Purpose:** Executes all valid stages of the pipeline.
- **Description:** Controls the sequential execution of pipeline stages while accounting for stalls and halts. Invokes the appropriate stage function for each active stage.
- **Input:**
    - pipeline: Pointer to the Pipe_fig structure representing the pipeline.
- **Output:** None.

static void enter_params_to_regs(Pipe_fig* pipeline, Pipe_figstate stage)

- **Purpose:** Prepares and transfers parameters to registers for the current stage.
- **Description:** Extracts register values from the instruction in the specified pipeline stage and prepares them for execution.
- **Input:**
    - pipeline: Pointer to the Pipe_fig structure representing the pipeline.
    - stage: The pipeline stage from which parameters are to be extracted.
- **Output:** None.

static bool checkfor_data_hazards(Pipe_fig* pipeline)

- **Purpose:** Detects data hazards in the pipeline.
- **Description:** Analyzes dependencies between pipeline stages to determine if data hazards exist that could stall execution.
- **Input:**
    - pipeline: Pointer to the Pipe_fig structure representing the pipeline.
- **Output:** true if a data hazard is detected, false otherwise.

static bool check_hazrads_by_comparing_regs(Pipe_fig* pipeline, Pipe_figstate stage)

- **Purpose:** Identifies potential data hazards by comparing register dependencies.
- **Description:** Checks the destination register of instructions in different stages for conflicts with source registers, considering immediate and zero registers.
- **Input:**
    - pipeline: Pointer to the Pipe_fig structure representing the pipeline.
    - stage: The pipeline stage being checked for hazards.
- **Output:** true if a hazard is detected, false otherwise.

static void stats_update(Pipe_fig* pipeline)

- **Purpose:** Updates pipeline performance statistics.
- **Description:** Increments counters for decode and memory stalls based on the pipeline's current state.
- **Input:**
    - pipeline: Pointer to the Pipe_fig structure representing the pipeline.
- **Output:** None.

static void (*stage_to_exe[PIPE_SIZE])(Pipe_fig* pipeline)

- **Purpose:** Provides a function pointer array for executing pipeline stages.
- **Description:** Maps each pipeline stage to its corresponding function (e.g., fetch, decode, etc.), enabling sequential execution.

Public Functions

void Pipe_Init(Pipe_fig *pipeline)

- **Purpose:** Initializes the pipeline with default values.
- **Description:** This function sets up the pipeline by clearing its stages, resetting statistics, and configuring default values for the program counter and halt flags. It ensures the pipeline is ready to process instructions.
- **Input:**
    - pipeline: Pointer to the Pipe_fig structure to be initialized.
- **Output:** None.

void Pipe_iteration_exe(Pipe_fig* pipeline)

- **Purpose:** Executes an iteration of the pipeline stages.
- **Description:** This function performs a single cycle of the pipeline, handling data and memory hazards, executing pipeline stages, and updating statistics.
- **Input:**
    - pipeline: Pointer to the Pipe_fig structure representing the pipeline.
- **Output:** None.

void Pipe_ToTrace(Pipe_fig* pipeline, FILE *trace_file)

- **Purpose:** Logs the state of the pipeline to a trace file.
- **Description:** This function outputs the program counter values of each pipeline stage to the trace file or marks inactive stages with "---" for debugging and analysis.
- **Input:**
    - pipeline: Pointer to the Pipe_fig structure representing the pipeline.
    - trace_file: Pointer to the file where the pipeline trace will be written.
- **Output:** None.

void Pipe_Bubbles(Pipe_fig* pipeline)

- **Purpose:** Inserts bubbles into the pipeline as needed.
- **Description:** This function handles stalls and propagates bubbles through the pipeline stages to manage hazards or delays.
- **Input:**
    - pipeline: Pointer to the Pipe_fig structure representing the pipeline.
- **Output:** None.

void BubbleStage(Pipe_instruction_stage* dest, const Pipe_instruction_stage* src)

- **Purpose:** Transfers a bubbled instruction from one stage to the next.
- **Description:** Copies instruction data, program counter, and execution results from the source stage to the destination stage, simulating a bubble's progression through the pipeline.
- **Input:**
    - dest: Pointer to the destination stage.
    - src: Pointer to the source stage.
- **Output:** None.

bool Pipe_Flush(Pipe_fig* pipeline)

- **Purpose:** Flushes all stages of the pipeline.
- **Description:** This function clears all pipeline stages by resetting their program counters to an invalid state (UINT16_MAX) and ensures the pipeline is halted before flushing.
- **Input:**
    - pipeline: Pointer to the Pipe_fig structure representing the pipeline.
- **Output:**
    - Returns true if the pipeline is successfully flushed.
    - Returns false otherwise.

OpcodeHandlers

General Description

The OpcodeHandlers module is responsible for implementing all the operations associated with opcodes in the simulator. It provides functionality for arithmetic, logical, branching, and memory operations, allowing the system to interpret and execute instructions effectively. Each opcode is mapped to a corresponding function, which performs the required computation or action.
This module is implemented across two files:

- **C File (OpcodeHandlers.c)**: Contains the implementation of all opcode-related functions, including arithmetic operations (e.g., add, sub), logical operations (e.g., and, or), branching conditions, and utility functions.
- **Header File (OpcodeHandlers.h)**: Declares the opcode functions, defines constants and types, and provides a mapping between opcodes and their corresponding functions.

Main Variables and structions

OpcodeFunctions

- **Description:** Enumeration of all supported opcode operations in the simulator.
- **Contents:**
  - **Arithmetic Operations:**
    - ADD: Adds two values.
    - SUB: Subtracts one value from another.
    - MUL: Multiplies two values.
  - **Logical Operations:**
    - AND: Performs a bitwise AND.
    - OR: Performs a bitwise OR.
    - XOR: Performs a bitwise XOR.
  - **Shift Operations:**
    - SLL: Logical shift left.
    - SRL: Logical shift right.
    - SRA: Arithmetic shift right.
  - **Branch Operations:**
    - BEQ: Branch if equal.
    - BNE: Branch if not equal.
    - BLT: Branch if less than.
    - BGT: Branch if greater than.
    - BLE: Branch if less than or equal.
    - BGE: Branch if greater than or equal.
    - JAL: Jump and link.
  - **Memory Operations:**
    - LW: Load word.
    - SW: Store word.
  - **Other:**
    - HALT: Halts execution.

OpcodeParams

- **Description:** Structure containing all parameters needed to execute an opcode.
- **Contents:**
    - uint32_t *rd: Pointer to the destination register.
    - uint32_t rs: Value of the source register.
    - uint32_t rt: Value of the target register.
    - uint32_t *memory_p: Pointer to the memory for load/store operations.
    - uint16_t *pc: Pointer to the program counter.
    - bool *halt: Pointer to the halt flag.

Main Functions

Public Functions

Arithmetic Operations
These functions perform basic arithmetic computations on the provided operands.

- **void add(OpcodeParams* params)**: Adds the values of rs and rt and stores the result in rd.
- **void sub(OpcodeParams* params)**: Subtracts the value of rt from rs and stores the result in rd.
- **void mul(OpcodeParams* params)**: Multiplies the values of rs and rt and stores the result in rd.

These functions are used for arithmetic operations commonly encountered in computational tasks.

Logical Operations
These functions perform bitwise logical computations.

- **void and(OpcodeParams* params)**: Computes the bitwise AND of rs and rt and stores the result in rd.
- **void or(OpcodeParams* params)**: Computes the bitwise OR of rs and rt and stores the result in rd.
- **void xor(OpcodeParams* params)**: Computes the bitwise XOR of rs and rt and stores the result in rd.

These functions are essential for logical comparisons and conditional evaluations in the program.

Shift Operations
These functions handle bitwise shift operations.

- **void logicShiftLeft(OpcodeParams* params)**: Shifts the value of rs left by the number of bits specified in rt and stores the result in rd.
- **void arithmeticShiftRight(OpcodeParams* params)**: Performs an arithmetic right shift on rs by the number of bits specified in rt and stores the result in rd.
- **void logicShiftRight(OpcodeParams* params)**: Performs a logical right shift on rs by the number of bits specified in rt and stores the result in rd.

These functions are used to manipulate data by shifting its bits, useful in low-level programming tasks.

Branch Operations
These functions handle conditional branching based on operand values.

- **void branchEqual(OpcodeParams* params)**: Sets the program counter (pc) to the value in rd if rs equals rt.
- **void branchNotEqual(OpcodeParams* params)**: Sets the program counter (pc) to the value in rd if rs does not equal rt.
- **void branchLessThan(OpcodeParams* params)**: Sets the program counter (pc) to the value in rd if rs is less than rt.
- **void branchGreaterThan(OpcodeParams* params)**: Sets the program counter (pc) to the value in rd if rs is greater than rt.
- **void branchLessEqual(OpcodeParams* params)**: Sets the program counter (pc) to the value in rd if rs is less than or equal to rt.
- **void branchGreaterEqual(OpcodeParams* params)**: Sets the program counter (pc) to the value in rd if rs is greater than or equal to rt.

These functions control the flow of the program by altering the pc based on specific conditions.

Jump Operations
These functions handle direct changes to the program counter.

- **void jump(OpcodeParams* params)**: Sets the program counter (pc) to the value in rd.

Jump operations are used to unconditionally redirect the flow of execution.

Utility Functions
These functions provide metadata about the opcodes.

- **bool IsOpcodeBranch(uint16_t opcode)**: Returns true if the opcode corresponds to a branch operation, otherwise returns false.
- **bool IsOpcodeMemory(uint16_t opcode)**: Returns true if the opcode corresponds to a memory operation (e.g., LW, SW), otherwise returns false.

These utility functions help in identifying the type of operation an opcode represents.

Opcode Function Table

- **static void (*OpcodeFunctionTable[NUMBER_OPCODES])(OpcodeParams* params)**: A lookup table mapping opcodes to their corresponding function implementations.

Simulation

General Description

The MultiCoreSim.c file serves as the main entry point for the simulation, orchestrating the execution of the entire multicore processor system. It coordinates the initialization, execution, and finalization of all system components, including processor cores, main memory, caches, and the bus controller.
This file handles the following high-level responsibilities:

- **File Management:** Ensures that all required input and output files are properly opened, accessed, and closed during the simulation.
- **Component Initialization:** Sets up the cores, memory, caches, and bus controller to prepare the system for execution.
- **Simulation Loop:** Executes the simulation by running each core and coordinating bus transactions until all cores are halted.
- **Result Finalization:** Finalizes the simulation by shutting down cores, printing memory contents, and generating output files.

In addition there is a header file, sim.h, contains some essential definitions, constants, and data structure required for the multicore processor simulator.

Main Variables and structions

ProcessorCore **cores[NUM_OF_CORES]**

- **Description:** Represents an array of processor cores in the simulation.
- **Contents:**
    - Each element is a ProcessorCore instance that contains the state and operations of a single core.
    - NUM_OF_CORES: A constant defining the total number of cores in the system.
    - **:** Enables parallel simulation of multiple processor cores, enable the execution of instructions and coordination with shared resources such as memory and the bus. Each core operates independently while contributing to the overall simulation.

Format_of_instruction

- **Description:** Represents the format of a single instruction in the simulator.
- **Contents:**
    - uint16_t imm: 12-bit immediate value [0:11].
    - uint16_t rt: 4-bit source register 1 [12:15].
    - uint16_t rs: 4-bit source register 0 [16:19].
    - uint16_t rd: 4-bit destination register [20:23].
    - uint16_t opcode: 8-bit operation code [24:31].
- **Purpose:** Allowing for decoding and execution in a structured manner.

#define **NUM_OF_CORES** 4

- **Description:** Defines the total number of processor cores in the simulator.

#define **NUM_OF_REGS** 16

- **Description:** Specifies the total number of registers available to each core.

#define **IMM_REG** 1

- **Description:** Index of the register reserved for immediate values.

#define **ZERO_REG** 0

- **Description:** Index of the zero register.

#define **START_MUTABLE_REG** 2

- **Description:** Index of the first general-purpose mutable register.
- **Purpose:** Defines the starting point for general-purpose register allocation.

#define **PC_REG** 15

- **Description:** Index of the program counter register.

Main Functions

Static Functions

static void **initCores**()

- **Purpose:** Initializes all processor cores in the system.
- **Description:** This function sets up the ProcessorCore instances for the simulation. It assigns file handles to each core and initializes them with unique IDs.
- **Input:** None.
- **Output:** None.

static bool **isProcessorHalted**()

- **Purpose:** Checks whether all processor cores are halted.
- **Description:** This function iterates through all cores and verifies their halted state using the core_is_halted function. It returns true if all cores are halted, otherwise false.
- **Input:** None.
- **Output:**
    - true if all cores are halted.
    - false if at least one core is still active.

Assembly Test Files

Counter

General Overview
The counter system is implemented across 4 cores, with each core executing its unique assembly code to participate in a synchronized counting operation. This system ensures coordinated updates to a global counter while maintaining local counters for each core. Synchronization is achieved using memory-mapped counters and a modulo operation to determine core execution turns.

Each core has:

- A **global counter** stored in memory address 0, shared among all cores.
- A **local counter** initialized to 128, decremented during the core's operation.
- A **core number** (r2), unique to each core, used for synchronization.
- Modulo logic (mod 4) to determine which core is allowed to execute.

Registers Used

- **r2**: Core number (unique to each core).
- **r3**: Global counter (shared among all cores).
- **r4**: Local counter (specific to the core).
- **r5**: Modulo result to determine core execution turn.
- **r6**: (Core 3 only) Additional register used to force conflict for testing.

Core 0 Assembly Code (counter_core0.asm)

- **Description:** Core 0 initializes the global counter, updates it, and decrements its local counter while synchronizing with other cores via modulo logic.
- **Behavior**:
    - Loads the global counter from memory and increments it.
    - Uses a modulo operation to check if it's this core's turn to execute.
    - Decrements its local counter during each execution cycle.
    - Polls (waits) if it's not its turn, ensuring synchronization with the other cores.

Core 1 Assembly Code (counter_core1.asm)

- **Description:** Core 1 operates similarly to Core 0 but is assigned core number 1. It uses the same logic for synchronization and counter updates.
- **Behavior**:
    - Loads the global counter and uses modulo logic to determine its turn.
    - If it's not its turn, it waits (polls) for the global counter to update.
    - Decrements its local counter when active and increments the global counter to signal the next core.

Core 2 Assembly Code (counter_core2.asm)

- **Description:** Core 2 is assigned core number 2. It follows the same synchronization and counter update logic as the previous cores.
- **Behavior**:
    - Follows the same synchronization pattern: checking the modulo result, waiting for its turn, and updating counters.
    - Ensures its local counter reaches zero before halting.

Core 3 Assembly Code (counter_core3.asm)

- **Description:** Core 3 is assigned core number 3. In addition to the standard logic, it introduces a forced conflict to test cache behavior.
- **Behavior**:
    - Performs the same synchronization and counting operations as other cores.
    - Introduces a deliberate memory access (to a specific address) to simulate a cache conflict or memory contention scenario.
    - Ensures its local counter and global counter are properly updated while testing the system's robustness under stress.

Each core operates independently but synchronizes through the shared global counter and the modulo logic. This ensures fair execution and coordinated updates across all cores. Core 3 adds a layer of complexity by testing how the system handles contention, making it a valuable addition to the process.

AddSerial

General Overview: This assembly code implements a vector addition operation on two vectors, where each vector contains 1024 elements. The program performs vector addition on a single core, utilizing loop unrolling to optimize the process. In this optimization, 4 elements are processed during each iteration. Each block in memory contains 4 numbers and the block of each vector we need to sum is mapped to the same lines in cache since the cache is direct map. Therefore, we choose to process 4 elements in each iteration to improve performance.

Registers Used:

- **$r2:** Base address of Vector 1.
- **$r3:** Base address of Vector 2.
- **$r4:** Base address of the Sum Vector.
- **$r5:** Loop counter, decrements by 4 for each iteration of the unrolled loop.
- **$r6-$r9:** Temporary registers for holding individual elements of Vector 1.
- **$r10-$r13:** Temporary registers for holding individual elements of Vector 2.
- **$r14-$r15:** Temporary registers for storing sums of vector elements.

Initialization:
The program initializes the base addresses for three vectors:

- Vector 1 at address 0
- Vector 2 at address 4096
- Sum Vector at address 8192.
  A loop counter is set to 1024, meaning the program will process 1024 elements, with 4 elements processed per iteration.

Unrolled Loop:
The loop is unrolled to process 4 elements at a time, reducing the iterations to 256. Each iteration involves:

1. Loading 4 elements from each vector.
2. Adding the corresponding elements and storing the result in the Sum Vector.
3. Updating the memory addresses for the next iteration.
4. Decrementing the loop counter to track processed elements.

Cache Conflict Misses:
To ensure that the cache is written to memory, the program intentionally creates memory access conflicts:

1. A counter is set to control the memory accesses.
2. The program sequentially loads data from different memory addresses, forcing cache evictions and writing the cache contents to memory.
3. The process continues until the counter reaches zero.

Program Termination:
After completing the vector addition and cache conflict simulation, the program terminates.

AddParallel

General Overview
The parallel addition is implemented across 4 cores, with each core executing an assembly program to contribute to the parallel addition of two vectors of size 4096 each. The code of each core is very similar to addserial code, but here, each core handles a quarter of the data. The cores can compute in parallel since each core is responsible for different 1024 numbers, and they store it to different addresses in memory.

Registers Used

- **$r2:** Base address of Vector 1.
- **$r3:** Base address of Vector 2.
- **$r4:** Base address of the Sum Vector.
- **$r5:** Loop counter, decrements by 4 for each iteration of the unrolled loop.
- **$r6-$r9:** Temporary registers for holding individual elements of Vector 1.
- **$r10-$r13:** Temporary registers for holding individual elements of Vector 2.
- **$r14-$r15:** Temporary registers for storing sums of vector elements.

Core 0 Assembly Code (addparallel_core0.asm)

- **Description**: Core 0 computes the sum of a portion of the vector (elements 0 to 1023 and their corresponding elements in the second vector) and updates its assigned part of the Sum vector.
- **Behavior**:
    - Initializes the base addresses for Vector 1, Vector 2, and the Sum vector.
    - The loop counter is set to 256, meaning Core 0 will process 256 iterations (each handling 4 elements).
    - In each iteration, Core 0 loads 4 elements from Vector 1 and Vector 2, storing them in temporary registers ($r6 to $r9 for Vector 1, $r10 to $r13 for Vector 2).
    - Core 0 computes the sum of each corresponding element from the two vectors (e.g., V1[0] + V2[0]) and stores the result in the corresponding positions in the Sum vector.
    - The base addresses for Vector 1, Vector 2, and Sum are incremented by 4 to point to the next 4 elements.
    - After each iteration, the loop counter is decremented by 1, ensuring the loop runs for 256 iterations in total.
    - Finally, to ensure that the cache is written to memory, the program intentionally creates memory access conflicts.

Core 1 Assembly Code (addparallel_core1.asm)

- **Description**: Core 1 computes the sum of a portion of the vector (elements 1024 to 2047 and their corresponding elements in the second vector) and updates its assigned part of the Sum vector.
- **Behavior**:
  - Initializes the base addresses for Vector 1, Vector 2, and the Sum vector.
  - The loop counter is set to 256, meaning Core 1 will process 256 iterations (each handling 4 elements).
  - In each iteration, Core 1 loads 4 elements from Vector 1 and Vector 2, storing them in temporary registers ($r6 to $r9 for Vector 1, $r10 to $r13 for Vector 2).
  - Core 1 computes the sum of each corresponding element from the two vectors (e.g., V1[0] + V2[0]) and stores the result in the corresponding positions in the Sum vector.
  - The base addresses for Vector 1, Vector 2, and Sum are incremented by 4 to point to the next 4 elements.
  - After each iteration, the loop counter is decremented by 1, ensuring the loop runs for 256 iterations in total.
  - Finally, to ensure that the cache is written to memory, the program intentionally creates memory access conflicts.

Core 2 Assembly Code (addparallel_core2.asm)

- **Description**: Core 2 computes the sum of a portion of the vector (elements 2048 to 3071 and their corresponding elements in the second vector) and updates its assigned part of the Sum vector.
- **Behavior**:
  - Initializes the base addresses for Vector 1, Vector 2, and the Sum vector.
  - The loop counter is set to 256, meaning Core 2 will process 256 iterations (each handling 4 elements).
  - In each iteration, Core 2 loads 4 elements from Vector 1 and Vector 2, storing them in temporary registers ($r6 to $r9 for Vector 1, $r10 to $r13 for Vector 2).
  - Core 2 computes the sum of each corresponding element from the two vectors (e.g., V1[0] + V2[0]) and stores the result in the corresponding positions in the Sum vector.
  - The base addresses for Vector 1, Vector 2, and Sum are incremented by 4 to point to the next 4 elements.
  - After each iteration, the loop counter is decremented by 1, ensuring the loop runs for 256 iterations in total.
  - Finally, to ensure that the cache is written to memory, the program intentionally creates memory access conflicts.

Core 3 Assembly Code (addparallel_core3.asm)

- **Description**: Core 3 computes the sum of a portion of the vector (elements 3072 to 4095 and their corresponding elements in the second vector) and updates its assigned part of the Sum vector.
- **Behavior**:
  - Initializes the base addresses for Vector 1, Vector 2, and the Sum vector.
  - The loop counter is set to 256, meaning Core 3 will process 256 iterations (each handling 4 elements).

- In each iteration, Core 3 loads 4 elements from Vector 1 and Vector 2, storing them in temporary registers ($r6 to $r9 for Vector 1, $r10 to $r13 for Vector 2).
- Core 3 computes the sum of each corresponding element from the two vectors (e.g., V1[0] + V2[0]) and stores the result in the corresponding positions in the Sum vector.
- The base addresses for Vector 1, Vector 2, and Sum are incremented by 4 to point to the next 4 elements.
- After each iteration, the loop counter is decremented by 1, ensuring the loop runs for 256 iterations in total.
- Finally, to ensure that the cache is written to memory, the program intentionally creates memory access conflicts.