Design Document: Functional Simulator for Subset of RISC-V instruction set

The document describes the design aspect of myRISCVsim, a functional simulator for a subset of the RISC- V instruction set.

**Input/Output**

**Input**

Input to the simulator is a MC file that contains the encoded instruction and the corresponding address at which instruction is supposed to be stored, separated by space. For example:

0x0 0xFFC10113

0x4 0x00D00193

0x8 0x00312023

0xC 0x00012203

0x10 0x00410113

0x14 0x0000007F

**Functional Behaviour and output**

**Proposed Solution:**

We propose to implement a Cache class that models the cache memory. The Cache class will have the following attributes and methods:

**Attributes:**

- cacheSize: An integer representing the size of the cache memory in bytes.

- blockSize: An integer representing the size of each cache block in bytes.

- numBlocks: An integer representing the total number of blocks in the cache.

- numSets: An integer representing the number of sets in the cache for set-associative caches.

- numWays: An integer representing the number of ways for set-associative caches.

- cacheType: A string representing the type of the cache (direct\_mapped, fully\_associative, or set\_associative).

- replacementPolicy: A string representing the replacement policy used by the cache (LRU or LFU).

- hitTime: An integer representing the cache hit time in clock cycles.

- missPenalty: An integer representing the cache miss penalty in clock cycles.

- accesses: An integer representing the total number of memory accesses.

- hits: An integer representing the total number of cache hits.

- misses: An integer representing the total number of cache misses.

- coldMisses: An integer representing the number of cold misses (i.e., a block is not present In the cache).

- conflictMisses: An integer representing the number of conflict misses (i.e., a block is present in the cache but not in the correct set).

- capacityMisses: An integer representing the number of capacity misses (i.e., the cache is full and a block needs to be replaced).

- memoryStalls: An integer representing the number of clock cycles spent waiting for memory.

**Methods:**

- Constructor: Initializes the Cache object with the input parameters. Calculates the number of sets and ways based on the cache type and associativity.

- Destructor: Deallocates the memory used by the Cache object.

- lookup: Looks up the cache for the given memory address. If the block is present in the cache, returns the data from the cache. If the block is not present, loads it from the memory and updates the cache. Uses the specified replacement policy to replace the block if needed.

- updateLRU: Updates the cache using the LRU replacement policy. Returns the data from the cache.

- retrieveLRU: Retrieves the data from the cache using the LRU replacement policy. Updates the cache.

- updateLFU: Updates the cache using the LFU replacement policy. Returns the data from the cache.

- retrieveLFU: Retrieves the data from the cache using the LFU replacement policy. Updates the cache.

- getData: Returns the data from memory for the given memory address.

**Components:**

Predictor.h: header file for branch predictor

Pipeline.h: header file for pipeline stages

Common.h: header file for common data structures

Fetch.h: header file for the fetch stage

Decode.h: header file for the decode stage

Execute.h: header file for the execute stage

MemAccess.h: header file for the memory access stage

Writeback.h: header file for the writeback stage

Main.cpp: main file that instantiates pipeline stages and executes instructions

**Data hazards:**

checkHazardRS1(): checks for hazards between stages based on opcode and register usage

checkHazardRS2(): checks for hazards between stages based on opcode and register usage

forward(): forwards data between pipeline stages to avoid hazards

**Instruction types:**

R-type: arithmetic and logic operations

I-type: immediate arithmetic and logic operations, loads

S-type: stores

B-type: branches

U-type: upper immediate loads

J-type: jumps

**Pipeline stages:**

Fetch: fetches instructions from memory

Decode: decodes instructions and reads registers

Execute: performs arithmetic and logic operations

Memory access: accesses memory for loads and stores

Writeback: writes results to registers

**Limitations:**

The implementation does not support exceptions, interrupts, or privileged instructions.

**Design of Simulator**

**Data structure**

Registers, memories, intermediate output for each stage of instruction execution are declared as global variables. Being static, the variables are not visible outside the file, thus making the data encapsulated in the myRISCVSim.c.

**Simulator flow:**

There are two steps:

* First memory is loaded with an input memory file.
* Simulator executes instructions depending on the knob values.

There is an infinite loop, which simulates all the instructions till the instruction sequence reads “0x7f”.

Next we describe the implementation of fetch, decode, execute, memory, and write-back function.

**Test plan**

We test the simulator with following assembly programs:

* Fibonacci Program
* Bubble Sort
* Sum of the array of N elements. Initialize an array in the first loop with each element equal to its index. In the second loop find the sum of this array, and store the result at Arr[N].