RISC-V ISA Simulator:

Design Documentation

Team members:

Bellam Jayanth - 2019CSB1080

Sagina Jaya Govinda Venkata Sai Teja – 2019CSB1114

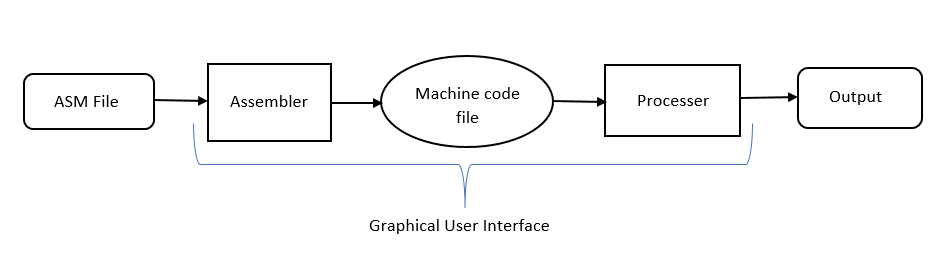
Rohit Dagar – 2019EEB1186

Satia – 2019CSB1118

Keshav Goyal – 2019CSB1165

**Features:**

* **Simulator backend:** Python
* **ISA:** RISC-V RV32IM
* **Size of instruction:** 32 bits
* **Number of registers:** 32
* **Size of registers:** 32 bits

**Overview:**

**Phase 1: Five step instruction execution**

## Input:

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

Functional behaviour:

All the instructions given in the input .mc file is executed as per the functional behaviour of the instructions. Each Instruction will go through the following steps:

* Step 1: **Fetch:** It gets the instruction and increment the PC, and prints that the instruction is fetched and PC is incremented.
* Step 2: **Decode:** Identify the instruction and registers (source and destination), and prints the operation, rs1, rs2, rd.
* Step 3: **Execute:** ALU operations is done or effective address calculation for load, store instructions. The control circuitry select the ALU operation and either Read data from the registers or a sign-extended immediate value as inputs to the ALU, then prints the instruction statement.
* Step 4: **Memory Access:** Read/write data from/to the memory. The control lines set in this stage are Branch, Memory Read, and Memory Write. Then print the retrieved memory at the memory address.
* Step 5: **Register Update or Writeback:** update destination register, and print the result updated in the destination register.

# Test plan:

We test the simulator with following assembly programs:

* Fibonacci Program
* Factorial Program

**Phase 2: Pipeline implementation with and without data forwarding**

**Input:**

Input is the same that is a machine code file.

**Pipeline Vs Non-Pipeline:**

|  |  |
| --- | --- |
| Pipeline | non pipeline |
| In pipelining system, multiple instructions are overlapped during execution. | In a Non-Pipelining system, processes like decoding, fetching, execution and writing memory are merged into a single unit or a single step. |
| Many instructions are executed at the same time. | Only one instruction is executed at one time. |
| CPI (ideal) is 1 | CPI is 5 |

**Pipeline Hazards:**

**Structural Hazard**

When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

**Data Hazards**

Also called a **pipeline data hazard**. When a planned instruction cannot execute in the proper clock cycle because data that are needed to execute the instruction are not yet available.

**Solution:**

**Forwarding**

Also called **bypassing**. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory.

* We have included forwarding paths from:
* M to M
* M to E
* E to E
* M to D
* E to D

**Stalling**

Also called **bubble**. A stall initiated in order to resolve a hazard.

* If data forwarding is stopped then stalling will be implemented.

**Control hazard**

Also called **branch hazard**. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

**Solution:** (We used static branch prediction)

**Branch prediction**

A method of resolving a branch hazard that assumes a given outcome for the conditional branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

**Branch Target Buffer (BTB):**

* Stores the address of control instructions.
* Stores the target address (if Taken) for each of them.
* Indexed using PC!
* Note: Auipc instruction is assumed to be ALU type instruction.

The simulator prints messages at the end of simulation:

* Stat1: Total number of cycles
* Stat2: Total instructions executed
* Stat3: CPI
* Stat4: Number of Data-transfer (load and store) instructions executed
* Stat5: Number of ALU instructions executed
* Stat6: Number of Control instructions executed
* Stat7: Number of stalls/bubbles in the pipeline
* Stat8: Number of data hazards
* Stat9: Number of control hazards
* Stat10: Number of branch mispredictions
* Stat11: Number of stalls due to data hazards
* Stat12: Number of stalls due to control hazards
* Values in the register file at the end of each cycle.
* Information in the pipeline registers at the end of each cycle (similar to tracing), along with cycle number.
* pipeline registers information for a particular instruction of our interest.

**Phase 3: Appending a cache like memory module to Phase 2**

**Input parameters for Instruction cache and Data cache:**

Cache size, Cache block size, Number of ways for Set Associative (SA).

**Cache**

Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.

Cache memory is used to reduce the average time to access data from the Main memory. A safe place for hiding or storing things.

**Cache Working**

Upon receiving a address from the processor, the control circuitry in the cache (namely, cache controller) will Compare this address with those of Tag array.

If (there is a match)

{ then its called Cache Hit.

Cache controller will access Data array and send a copy of the requested word to the processor.

}

Else

{ Cache Miss.

Cache controller will forward this address to the next level in the hierarchy.

}

* The Divide() function in Cache class of Simulator.py counts number of bits for block offsets, index, tag.
* In Search() function in Cache class of Simulator.py for instruction cache the value of offset=4 and for Memory cache the value of offset=1. This is for uniformity.
* The search() function in Cache class of Simulator.py calculates index, block offset, tag.

**4 levels of cache working:**

**Block Placement**

Placement of a block in the cache.

**Direct-mapped cache**

A cache structure in which each memory location is mapped to exactly one location in the cache.

**Fully associative cache**

A cache structure in which a block can be placed in any location in the cache.

**Set-associative cache**

(We have done working of Set associative cache in our project)

A cache that has a fixed number of locations (at least two) where each block can be placed.

**Block Identification**

By looking address of the block.

The tag is matched with tag array to check hit/miss , if it is a hit then its output will be given to Data MUX, it will help Data MUX to select data from Data array, then the requested word will be forwarded to the processor.

**Block Replacement**

When it’s a miss the address is forwarded to the next level and the next level replies with the requested block of data.

In the meantime, the cache controller needs to find space in that Set. If there is a vacant space in the Set, then the incoming block will be stored in it. If not, one of the existing blocks in the Set needs to be evicted. This block is called Victim, and it is selected by replacement policy.

Here, for Set Associative we have used Least Recent Used (LRU) Policy.

**Working of LRU policy**

The most recently accessed block is placed at first (head) and least used at last.

In search() function in Simulator.py, We identify set by index then we check for hit and if it’s a hit then we implement LRU, else we return false.

**Write Strategy**

* In write() function, we find block address and then we insert block in the memory.
* We implement write allocate.

The simulator prints messages at the end of simulation:

Two sets of stats will be printed - one for Instruction cache and another for Data cache.

Stats are as following:

* Number of accesses
* Number of hits
* Number of misses
* Number of cold, conflict, capacity misses.