**FINAL DESIGN DOCUMENT: RISC-V SIMULATOR**

The document describes the design aspect of our RISC-V simulator made using Python as a programming language.

# **Input/Output**

## **Input**

The input to the program is a .mc file that contains the encoded instructions and the corresponding address at which instruction is supposed to be stored, separated by space. For example:

0x0 0xE3A0200A

0x4 0xE3A03002

0x8 0xE0821003

## **Functional Behavior and output**

The simulator reads the instruction from instruction memory, decodes the instruction, reads the register, executes the operation, and writes back to the register file. The instruction set supported is the same as given in the lecture notes of CS-204.

The execution of instruction and fetching continues till it reaches instruction “text\_end” in the input file. In other words as soon as the instruction reads “text\_end”, the simulator stops and writes the updated memory contents to a memory dictionary.

The simulator also prints messages for each stage, for example:

We have added **GUI** for this simulator :

1. A window will appear

2. If you want to run the code step by step:-

>>>>Click on Step button on the gui

-There is a Registers Button and a Memory Button.

-The Registers Button shows all the registers and the values in them

-The Memory Button shows all the data stored at different addresses in the memory

-There are three more buttons which show the Heap and Stack Memory.

3. If you want to run the whole code together:- Click on the Run Button

A separate box is shown where the clock cycle number is displayed. Steps followed in each instruction are also shown as follows:

- Fetch prints:

o “FETCH:

PC\_temp -> PC+4

Fetched instruction - instruction”

- Decode

o “DECODE:

Instruction Type - instr\_type

Operation - operation +

Register values are read.”

- Execute

o “EXECUTE:

PC -> hex(PC)

Returned value - calculated value”

- Memory

o “MEMORY ACCESS:

Memory at 'address' is updated.”

- Writeback

o “REGISTER UPDATE:

Register xi is updated.”

# **Design of Simulator**

## **Data structure**

Registers, memories, Instruction register , clock used for each stage of instruction execution are declared as global variables.

For the implementation of registers and memories two separate dictionaries are used, while for storing the instructions ‘instructions’ and for storing data in memory ‘memory’’ is used.

## **Simulator flow:**

There are two steps:

1. First memory is loaded with an input memory dictionary and each instruction of the input file is stored in instructions and memory data in memory.

2. Simulator executes instructions one by one.

For the second step, there is an infinite loop, which simulates all the instructions till the instruction sequence reads “text\_end”.

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

### **FETCH:**

In this step after the input file is passed to temp3.py, current instruction is returned by this function which is stored in ‘instructions’ .

### **DECODE:**

As for each instruction stored in ‘instructions’ dictionary, each instruction is decoded according to the opcode it has. Further according to the type instruction necessary details are stored and returned in reg\_list which is further used in the Execution step. All sign extensions required are taken care of for the execution step in this stage.

### **EXECUTE:**

The execute step uses the already calculated values of the register addresses and the instruction to be performed to calculate a value (var) which is then used to write to registers or memory depending upon the type of instruction.The value of PC is also updated in this step, as PC\_temp is calculated.

### **MEMORY:**

In this step the memory function is called with the address of memory to be updated or accessed as a parameter and according to the operation which requires the use of memory such as lb,lh,ld,jal,jalr the values in the ‘memory’ dictionary are read or updated. When memory is accessed, the read value is returned.

### **WRITEBACK:**

If any operation requires updating the values of registers, they are updated by calling the registerUpdate function.

# **Test plan**

We test the simulator with following assembly programs:

1. input\_fib11.mc
2. input\_fact10.mc
3. input\_bubble.mc
4. Inp.mc

**PHASE 2:**

**PIPELINED IMPLEMENTATION OF RISC-V INSTRUCTION EXECUTION**

The instructions in this phase are implemented in a pipelined fashion. Pipelining of Instruction Execution is an implementation technique in which multiple instructions are

overlapped in execution. Pipelining improves performance by increasing instruction throughput. The number of simultaneously executable instructions also increases with pipelining. The overall ALU also becomes faster.

After the execution, an output file is created, named “ksh3.out” in which all the required statistics and messages are printed.

About the GUI interface:

A window opens, the number of knobs are:

* Enable/disable forwarding (Enables stalling if not forwarding)
* Enable/disable pipelining
* Cycle by cycle instruction execution
* Run

We handled two types of hazards: Data hazards and control hazards

**DATA STRUCTURES**

In addition to the ones implemented in phase 1, 4 queues were implemented as buffers namely “fetch\_buffer”, “decode\_buffer”, “execute\_buffer”, “memory\_buffer”. To figure which function (fetch, decode, execute, etc) is to be implemented, 5 counters were used which carried the PC of the instruction to be implemented.

The values from the buffers were used while implementing data forwarding.

**DATA FORWARDING**

Buffers are introduced after each stage and instead of passing values directly to the next stage, the buffer provides the values to the next one. While handling hazards using data forwarding, the required values are directly given from the buffers instead of registers or memory. This saves a lot of time and cycles and makes the pipelining more efficient.

**STALLING**

If a hazard is identified in the decode stage and stalling is enabled, the pipeline is stalled till the previous instruction has completed its execute or memory access stage (according to the situation). The current instruction is stopped or stalled till then. After it is stalled, it resumes and the pipelining continues.

**HAZARDS**

The decode stage identifies any control hazards that might be present in the input code. A dictionary, registers\_bool carries all the registers and their states. If free, the state is 0. If it is in the decode stage, its status is 3, 2 in execute and 1 in memory access. If the state of the register in use is not zero, a hazard has occurred and it accesses the buffers to get the required value and adds that value further to its buffer.

# **Test plan**

We test the simulator with following assembly programs:

- Fibonacci Program

- Factorial Program

- Bubble sort Program

**STATS TO BE PRINTED**

• 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

**PHASE 3: IMPLEMENTING CACHES**

On the hardware level, accessing cache is a much faster solution as compared to referring to the main memory every time. In phase 3, we are implementing cache which takes block size, cache size and set associativity as input. Block size should be entered in bytes and is expected to be a multiple of 4.

We have treated data and instruction cache separately and have made functions for reading and writing to cache, wherever needed. The instruction cache handles all requests from fetch instruction and the rest(load and store) are handled by data cache. The **readDataCache** function takes address as an input and checks if the corresponding block of data is already present in the data cache. If yes,then it's a hit and it simply reads the data from the cache and makes other necessary changes else, it's a miss and it loads the data from the main memory. Similar functioning is performed by **readInstrCache.** In case of **writeDataCache**, input parameters are the data to be stored and the address. The corresponding write function is performed in this case. If the address is already present in the cache, it's a hit and corresponding write operation is done (to the cache as well as main memory). If the address is not present in the cache, it's a miss and the address is accessed and updated from the main memory.

As mentioned in the project description, no pipeline stalls are introduced on a cash miss.

**LRU implementation:** For identifying the least recently used block, we maintain a tag which indicates the recency of each block. This recency value is updated after every read or write cache operation. Consequently, when we need to add a new entry in place of another one, the least recently used block is freed and used further.

In the GUI, as a part of stats, we have shown the set that is accessed in load, store and fetch instruction.

We have also shown the victim block in case of a miss. And have printed the number of accesses, hits and misses for both instruction and data cache.