Design Document: RISC-V SIMULATOR

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

# Phase 1

# Input/Output

## Input

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

For example:

0x0 0x07B00293

0x4 0x01300213

0x8 0x0480006F

0xc 0xFF010113

0x10 0x00112023

## Functional Behavior and output

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

The execution of instruction and fetching continues till it reaches instruction “0xffffc” in the input file. In other words as soon as instruction reads “0xffffc”, simulator stops and writes the updated memory contents on to a memory text file.

The simulator also prints messages for each stage, for example for the third instruction above following messages are printed.

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

A window will appear

| 2. Firstly click on Assemble

| If you want to run the code step by step:-

| >>>>Click on run button on the gui

| -In output log, will get the output after each step

| -Like the value stored in the register, opcode, immediate, Instruction Type etc.

| If you want to run the whole code together:-

|

|

| >>>>Click on run button

| -And the whole code will get executed

We have mentioned further details in the readme file.

* Fetch prints:
  + “FETCH: Fetch instruction 0xE3A0200A from address 0x0”
* Decode
  + “DECODE: Operation is ADD, first operand R2, Second operand R3, destination register R1”
  + “DECODE: Read registers R2 = 10, R3 = 2”
* Execute
  + “EXECUTE: ADD 10 and 2”
* Memory
  + “MEMORY: No memory operation”
* Writeback
  + “WRITEBACK: write 12 to R1”

# Design of Simulator

## Data structure

Registers, memories, intermediate, PC, RZ , Instruction register , clock, muxy, 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 instruction\_dict and for storing data in memory data\_dict is used. Our memory is Byte Addressable.

## Simulator flow:

There are two steps:

1. First memory is loaded with input memory dictionary and each instruction of the input file is stored in instruction\_dict and memory data in data\_dict.
2. Simulator executes instruction one by one.

For the second step, there is infinite loop, which simulates all the instruction till the instruction sequence reads “0xffffc”.

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

### FETCH:

In this step the input file data.mc is passed to the function fetch\_file when called and two dictionaries are returned by this function which is stored in instruction\_dict and data\_dict for further stages.

### DECODE:

As for each instruction stored in instruction register each instruction is decoded according to the opcode it has. Further according to each of the R,I,S,SB,U,UJ type instruction the rs1, rs2, immediate value , the type of operation , and other necessary details are stored and returned in the decoded\_info dictionary 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 decoded\_info and the pc\_temp (temporary program counter are passed to the execute function to execute the necessary operations according to the field. In this step according to the type of instruction information is stored in the decode\_info the function to execute that appropriate function is called off and the executed operation value is returned along with a pointer to the execution which we call as PC immediate which stores value of PC if any change occurs in it.These returned values are further stored in RZ and pc\_final variable for the next step.

### MEMORY:

In this step the memory function is called off 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 / updated and the updated memory dictionary along with other necessary parameters are returned along with updated rz if necessary. The value the memory step returns in this step is stored in muxy.

### WRITEBACK:

If any operation requires the updating of value given in the destination register(rd) is updated by calling the write back function with the passing of necessary parameters muxy and register\_dictionary.

At last pc is updated to pc\_final value and this process repeats until 0xffffc is encountered.

# Phase 3: Appending RISC-V pipeline with Caches

In our RISC-V Simulator, we have two types of memory. The primary memory and the cache memory are the two types of memory.

We're using a Python dictionary to implement the main memory. Data is stored in this dictionary as key-value pairs. The memory address is the key in this case, and the data stored there is the value. While taking input or during store instructions, these key-value pairs are modified as required.

The cache memory is initialized by the function “cacheinitialization” which takes an list input which has the cache size , block size and the value of k for set associativity.

We created a "memory" module that mimics the operation of cache memory. We have two functions that **doing\_load\_cache** which takes the inputs memory address given , memory dictionary, cache dictionary, number of clock cycles and variables hit or miss to detect weather the given address was found or not. Similarly **doing\_store\_cache** with all same parameters just one additional parameter byte\_val which needs to stored.

**LRU Policy** - In cache implementation, we are using LRU policy (Least Recently Used). We are keeping track of the least recently used block by maintaining a recency list. This list already being updated when the above two function **doing\_load\_cache** and **doing\_store\_cache** are called by the main file.

As mentioned in the project Description there are no pipeline stalls upon a cache miss. We have instantiated two caches one will work as **Instruction cache (I$)** and another will work as **Data cache (D$)**.So, all the requests from Fetch stage of your pipeline will be handled by I$. Like wise, requests from Memory stage will be handled by D$.

# Phase 2: Pipelined Implementation of RISC-V Instruction Execution

Pipelining of Instruction Execution is an implementation technique in which multiple instructions are

overlapped in execution. The following are the advantages of Pipelining:

1. Pipelining improves performance by increasing instruction throughput,
2. The number of simultaneously executable instructions also increases with pipelining,
3. The ALU becomes faster.

In the GUI Interface the user selects the option whether to introduce pipelining or continue with the Non Pipelined way as per the Project Phase1. In the case of the Pipelined approach, the user is prompted to select between data forwarding or stalling options to cater to data hazards, through a knob in the GUI.

Two types of hazards can be encountered:

1. DATA HAZARDS –

These can be resolved by Stalling or Data Forwarding approaches as chosen by the user.

1. CONTROL HAZARDS

# Design of Simulator

## Data structures

In addition to the data structures used in Phase 1, Queues have been introduced to keep track of the PC instructions in the various stages of pipelining in fetch\_pc, decode\_pc, execute\_pc, mem\_pc and write\_pc queues.

A new dictionary is formed to keep track of the Buffer values during simulation. These buffer values come to aid during data forwarding. Another dictionary of 32 registers depicts the state of rd registers which is explained below in Stalling Part.

## Data Forwarding:

Buffers are introduced in between the stages so that we may not have to wait for the writeback stage to obtain the value of register after execution stage. In the code the buffers are stored in the form of a dictionary and the values are used by the next instruction in case of data dependency. This helps avoid the delay and conflict caused and we are able to get good optimization.

## Stalling:

A dictionary buffer\_val\_for\_rd represents the 32 registers has all values initially set to -1. This default value indicates that the corresponding rd is not under processing. After the decode stage we upgrade the concerned destination register value in the dictionary by a positive value, indicating that the value of register in under processing.

Now, a variable named dummy\_val uses the dictionary buffer\_val\_for\_rd, and with some computation stores the difference between the decode and the write back stage. It helps repeat the process of decode (not advance to next step of execute) until the dummy\_val becomes 0 while iterating in a loop. This indeed provides a setup for stalling and solves data hazard issue.

## Control Hazards:

The code for the decode stage identifies if the instruction is a control instruction like jal, beq etc or not. Doing this in the decode stage helps save one cycle.

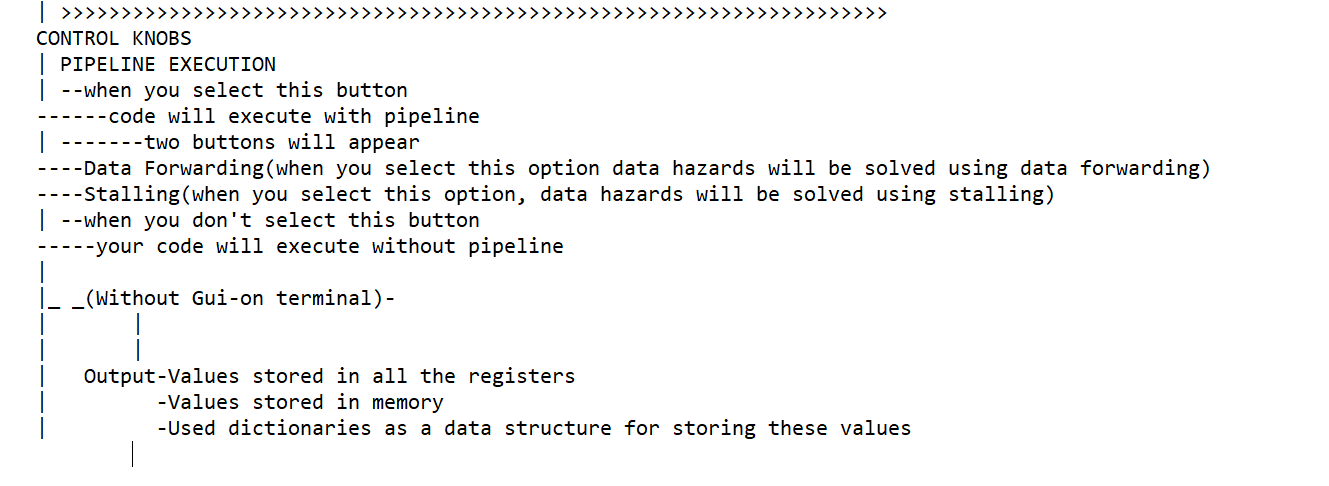
Next the btb dictionary stores the PC for the instruction given by the control instruction. The Execute stage gives the actual PC which is matched with that of btb dictionary. If the two are same, then it is indicated that the prediction is correct. In case of mismatch, the wrong PC is flushed out of the decode queue ‘decode\_pc’ and the correct one is pushed in the queue.

## Stats to be Printed in the Output File:

* Stat1: The Total Number of Cycles have been counted in a clock variable in the implementation
* Stat2: The count of instructions executed is taken care of in the execute function
* Stat3: The CPI is equal to the Total Number of Cycles divided by the Count of instructions and hence can be calculated using Stat 1 and Stat 2.
* Stat4, 5 and 6: The different types of instructions are identified in the decode stage and the count of Number of Data Transfer (load and Store instructions), number of ALU instructions and number of Control Instructions executed, are maintained.
* Stat8: Number of Data hazards are obtained from the forwarding/stalling occurrences.
* Stat9: Number of Control hazards are obtained from the number of mismatches in the Control part of the code.
* Stats 10, 11,12: The number of stalls/bubbles are calculated with the help of manipulation of dummy\_val variable in stalling part of data hazards and number of mispredictions in control part.

## GUI:

The control of Pipelining will appear on the GUI as shown below:



On the Output Front:

The Assemble button, Step and Run button, Register View button and Memory view button will work as per convention (similar to Venus simulator convention)

Additional Introduced Features:

The Print Pipeline Register will show all the pipeline registers

The Hit Branch Predictor will indicate a Hit or a Miss of the prediction

The Block Diagram of Instructions will be a rectangle where

* The 1st row means that the first row will get executed.
* The 2nd row means that 2nd instruction will get executed in the code
* 1st column will show which part of instruction get executed in the first cycle
* 2nd column will show which part of which execution will get executed in the 2nd cycle.

TODO

# Test plan

We test the simulator with following assembly programs:

* Fibonacci Program
* Factorial Program
* Bubble sort Program