Design Document: Functional Simulator for RISC-V 32 bit instruction set

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

Instructions that Simulator supports are-

R format - add, and, or, sll, slt, sra, srl, sub, xor, mul, div, rem

I format - addi, andi, ori, lb, ld, lh, lw, jalr

S format - sb, sw, sd, sh

SB format - beq, bne, bge, blt

U format - auipc, lui UJ format - jal

# **for non-pipelined implementation:-**

## **Input**

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

0x0 0xE3A0200A

0x4 0xE3A03002

0x8 0x003202B3

## **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 Project\_Phase1.pdf .

The execution of instruction continues till it reaches at the end of instruction.mc file. After every instruction simulator update destination register and writes the updated memory contents on to a memory text file(if instruction is of store type).

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

* Fetch prints:
  + INSTRUCTION : 0x003202B3
* Decode prints:
  + decode:
  + instruction is of r format(load)
  + add rs1: 4 rs2: 3 rd: 5
  + values [rs1]: 0 [rs2]:0 “(values that you stores initially in registers 4 and 3 here I am assuming as 0 and 0) “
* Execute prints:
  + ALU
  + OPERATION Performing : add
  + RZ = sum : 0
* Memory:
* Writeback:
  + writes 0 in register 5

# **Design of Simulator**

## **Data structure**

Registers, memories, intermediate output for each stage of instruction execution are declared as global variables .

## **Simulator flow:**

There are two steps:

1. First memory is loaded with an input memory file.
2. Simulator executes instructions one by one.

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

**Fetch()**

This function read instruction from instruction.mc file, we search instruction by PC and store in IR global variable. Its work as input for decode stage. Increment to PC.

**decode()**

This function takes instruction as an input from IR(updated by fetch stage) and extracts its opcode,fun3,fun7 based on this it’s decode the type and format of instruction and return instruction’s mnemonic value

And update the global variable RA,RB,rd,imme as input(value) of source register 1,value of source register 2 ,destination register number and immediate value(sign extended) based on type of instruction.

Also prints the values in specific format shown above in decode prints stage

**alu()**

This function takes the variable ‘operation’ (output of decode) and global variables RA, RB, imme as its input.

The function computes either the arithmetic operation value or the target address according to the input operation and updates the global variable RZ as output.

The function also prints the name of operation it is performing, type of value in RZ and the value of the RZ

**access\_memory()**

**WriteBack()**

This function does not take any parameter but it uses a control variable which is a global variable which comes from the decode stage.

This control variable is to detect whether the instruction is R-type, I-type, U-type, UJ-type or not. Because only in this type of instruction write back stage happens.

It also uses the RZ value which is also a global variable and it is output of ALU operation.

# **Test plan**

We test the simulator with following assembly programs:

* Fibonacci Program

For pipelined implementation:-

Input

In this implementation there are two files data.mc (data memory) and instruction.mc(instruction memory).

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

0x0 0xE3A0200A

0x4 0xE3A03002

0x8 0x003202B3

And data memory(data.mc file) contains data and the corresponding address at which is supposed to be stored, separated by space for example:

0x10000000 0x00101526

0x10000004 0x00101525

……………………………………………………..

## **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 in a pipelined behavior . The instruction set supported is the same as given in the Project\_Phase2.pdf .

The execution of instruction continues till it reaches at the end of instruction.mc file. After every instruction simulator update destination register and writes the updated memory contents on to a memory file(data.mc)(if instruction is of store type).

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

* Fetch prints:
  + INSTRUCTION : 0x003202B3
* Decode prints:
  + decode:
  + instruction is of r format(load)
  + add rs1: 4 rs2: 3 rd: 5
  + values in pipeline buffers decodeRA: 0 decodeRB: 0
* Execute prints:
  + ALU
  + OPERATION Performing : add
  + RZ = sum : 0
* Memory:
* Writeback:
  + writes 0 in register 5

# **Design of Simulator**

## **Data structure**

Registers, memories, intermediate output and intermediate buffers(pipeline registers) for each stage of instruction execution are declared as global variables .

## **Simulator flow:**

There are two steps:

1. First memory is loaded with an input memory file.
2. Simulator executes instructions in a pipelined format.

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

**Main()**

This fuction is the initializer of the program. The function first initializes the memory hash tables and asks for user inuts for enabling/disabling various knobs.

If pipeline is turned on, the main function calls the pipeline() function to start the pipelined simulation.

If pipeline is turned off, it calls the nonpipeline function to execute the non pipelined simulation.

The function outputs the various simulation parameters at the end including instruction count, total clock, CPI, number of control instructions etc.

**Fetch()**

This function read instruction from instruction.mc file, we search instruction by PC and store in IR global variable. Its work as input for decode stage. Increment to PC.

**pipelinedecode()**

This function takes instruction as an input from IR(updated by fetch stage) and extracts its opcode,fun3,fun7 based on this it’s decode the type and format of instruction and return instruction’s mnemonic value

And update the global variable decodeRA,decodeRB,decodeRD (interstage buffer of decode),rd,imme as input(value) initial value of source register 1,initial value of source register 2 ,destination register number and immediate value(sign extended) based on type of instruction.

It’s also stores the value of destination register of previous 2 instruction and current instruction in datadependence(global list that’s contain three elements) for checking of data hazards in execute stage for updating RA,and RB. Also prints the values in specific format shown above in decode prints stage. For more information check decode.py file

**dependenceChecker()**

This function takes the variable ‘operation’ (output of decode) and global list datadependence to check whether the input registers to alu have data dependence or not.

The function checks for data dependence on last two functions and modifies a global list if the alu input registers are dependent on the last two instructions.

The function also identifies the type of data dependence and accordingly increases the stall count if data forwarding is switched off.

**InitMemory()**

This function opens the given .mc file and distributes the content on it to the Hash tables TextMemory and DataMemory. Access of data in memory access step occurs through DataMemory.

**pipelinealu()**

This function takes the variable ‘operation’ (output of decode) and global variables RA, RB, imme as its input.

The function checks the data dependence on its inputs by calling dependenceChecker and accordingly receives the input data from either decode buffers or from the data forwarding paths

The function computes either the arithmetic operation value or the target address according to the input operation and updates the global variable RZ and alu buffer aluRZ as output.

The function also prints the name of operation it is performing, type of value in RZ and the value of the RZ

**access\_memory()**

**WriteBack()**

This function does not take any parameter but it uses a control variable which is a global variable which comes from the decode stage.

This control variable is to detect whether the instruction is R-type, I-type, U-type, UJ-type or not. Because only in this type of instruction write back stage happens.

It also uses the RZ value which is also a global variable and it is output of ALU operation

Phase 3: Implementing a data cache and a instruction cache to phase 2

Input:-

For input we are taking cache size, cache block size and number of ways for set associative cache from user.

Output:-

In output we are printing number of accesses, number of hits and number of misses for both data cache and instruction cache separately.

## **Functional Behavior and output :-**

We are using LRU policy for replacement in both caches with simple modification for replacement we are using the same index that found as a miss in previous iteration.

All simulation flow and data flow is followed of phase 2 with a simple addition in it we also using address from BTB as an input to set associative cache for counting number of accesses, hits and misses in fetch stage.

In memory access stage all simulation is followed of phase 2 in addition for load store instruction we are using address an an input to set associative data cache to count number of accesses ,hits and misses.

All other part is same as phase 2.

The simulator also prints messages for each caches at the end of simulation:-

To show for example lets take number of misses hits and accesses as 5 , 4 and 9 in both caches.

Instruction cache:

Number of access: 9

Number of hits: 5

Number of misses:4

Data cache:

Number of access: 9

Number of hits: 5

Number of misses:4

Different Functions used in simulation:-

There are only two additional functions one for instruction cache and another for data cache and some supporting functions for extracting bloc offset , index and tag array from address other than that all functions are same as phase 2 .

Global variable : DataNumHits, DataNumMisses, InsNumHits and InsNumMisses for counting number of misses and hits in data cache and instruction cache

There are few global variables named bo\_bits, index\_bits, tag\_bits, cachesize, blocksize, n(number of ways of SA)

Data\_cache() (function) that counts these following values-

blocks= cachesize/blocksize, numsets=blocks/n

bo\_bits=log(blocksize) here log is to the base 2

index\_bits=log(numsets),tag\_bits=total bits(32)-bo\_bits-index\_bits

total bits of address =32 (according to risc-v 32 bit isa)

Supporting functions:-

indexAndTag():- This function takes address (from BTB ) as input and returns index and tag array from address with the help of index\_bits and tag\_bits.

BlockOffSet():- This function takes address as an input and returns block offset value with the help of bo\_bits

SAInst():-

First this function extracts the tag array and data array values according to index ,then compre that’s tag value to tag of address(that’s we got from indexAndTag function)

If it is matching then it’s a hit otherwise it’s a miss. On a miss this function also writes its tag and data values in tag array and data array

And at last on hit it is return a instruction that we are passing into next decode phase.

SAData():-

This function counts Hits and misses same as SAInst() , update cache on misses and return data on load instruction and write on store instructions.