CS204

Computer Architecture

Phase 3

DESIGN DOCUMENT

Group Members:

1. Aayush Sabharwal 2019CSB1060
2. Aman Palariya 2019CSB1068
3. Sagalpreet Singh 2019CSB1113
4. Uday Gupta 2019CSB1127
5. Amritanshu Rai 2019EEB1140

Aim

To make a functional RISC-V Simulator, for a subset of RISC-V ISA.

Input/ Output

Languages Used: Python3 was used for designing the simulator. Tkinter was used to design GUI.

The input of the simulator is a .mc file, containing the machine code and their corresponding address, along with the data segment, which contains data stored in the memory at the start of the code.

For example:

```
0x0  0x10000517
0x4  0x00050513
0x8  0x10000597
...
0x50  0xFFFFFFFF

0x10000000 0x34
0x10000001 0x23
```

As seen above the input file contains the text segment, followed by the data segment.

The last instruction of the text segment must be 0xFFFFFFFF. This instruction marks the end of the program.

The data segment should be mentioned byte-wise.

The memory is divided for different functions:

1. The text segment is stored at the address 0x00000000
2. The data segment is stored at the address 0x10000000
3. The stack segment is stored at the address 0x7FFFFFFC
4. The heap segment is stored at the address 0x10007FE8

Implementation Description

The simulator takes input from a .mc file. In the .mc file, the data segment and text segment are written in different sections.

The simulator supports the following instructions from the RISC-V ISA

1. **R-Type:** add, and, div, mul, or, rem, sll, slt, sra, srl, sub, xor
2. **I-Type:** addi, andi, ori, lb, ld, lh, lw, jalr
3. **S Format:** sb, sw, sd, sh
4. **SB Format:** beq, bne, bge, blt
5. **U Format:** auipc, lui
6. **UJ Format:** jal

The simulator in phase-2 will be able to execute the program in 2 ways, either in a pipelined manner or in a non-pipelined manner.

Steps that are followed by the simulator are:

1. **Fetch**

   During this process, the simulator receives the instruction based on the value of the PC. here the instruction register is updated with the new value. If the value of the instruction register is 0xFFFFFFFF, then the simulator exits the program. The value of the PC is also stored in the PC-Temp.

   ```
   FETCH
           Reading word from memory location 0x00000000
           IR: 0x10000517
   ```

   Fig: Message Printed after execution of Fetch Stage

2. **Decode**

   In this step, the value of opcode, rs1, rs2, rd, immediate, func3, funct7 and control lines are calculated based on the value of the instruction register.

```
DECODE
        Opcode: 0b0010111
        rd: 0b01010
        funct3: 0b000
        rs1: 0b00000
        rs2: 0b00000
        funct7: 0b0001000
        U format detected
        immediate: 10000000
        alu.muxA: 0b1
        alu.muxB: 0b1
        alu.aluOp: 0b0
        alu.muxY: 0b0
        branch: 0b0
        jump: 0b0
        reg_write: True
```

Fig: Message Printed after execution of Decode Stage

3. **Execute**

   The ALU performs the operation on the register value rs1 and rs2 based on the value of opcode and control lines. Here we receive the value of rd.

```
EXECUTE
        Reading registers
        A: 0x00000000
        B: 0x00000000
        alu.operand1: 0x00000000
        alu.toperand2: 0x10000000
        alu.RM: 0x00000000
        alu.RZ: 0x10000000
        alu.zero: False
```

Fig: Message Printed after execution of Execute Stage

4. **Memory Access**

   For load/store, the memory is accessed to read/write the value according to the value of control lines.

```
MEMORY ACCESS
        alu.RY: 0x10000000
        Updating PC
        Moved to next sequential instruction
        PC: 4
```

Fig: Message Printed after execution of Memory Access Stage

5. **Register Update**

   In this step, the value of the rd register is updated, and the cycle counter is incremented by 1.

```
REGISTER UPDATE
        Writing value 0x10000000 to register x10
```

Fig: Message Printed after execution of Register Update

Modules Designed

IAG

This module keeps track of the PC. Depending on the type of instruction, the value of PC is incremented by 4 or value of immediate.

Control

This module controls the flow of execution. This module initiates the 5 substeps (fetch, decode, execute, memory access and register update), according to the method selected(pipelined, or non-pipelined).
The value of control lines are also set by this module.

ALU

The module executes all arithmetic operations such as add, mul, sub, etc. based on the value of control lines, funct3 and funct7, it decides the operation to take place and then operates on rs1, rs2 and immediate to give output.

Buffer

This module uses a dictionary to create buffers between various steps. Using this module we can delete or add items in the dictionary.
A total of 4 buffers are there

- Fetch_Decode Buffer
- Decode_Execute Buffer
- Execute_Memory Buffer
- Memory_Register Buffer

Memory

Used to handle processes in the memory, such as loading the instructions, or retrieving values from the memory.
The data width can be changed from byte to doubleword.

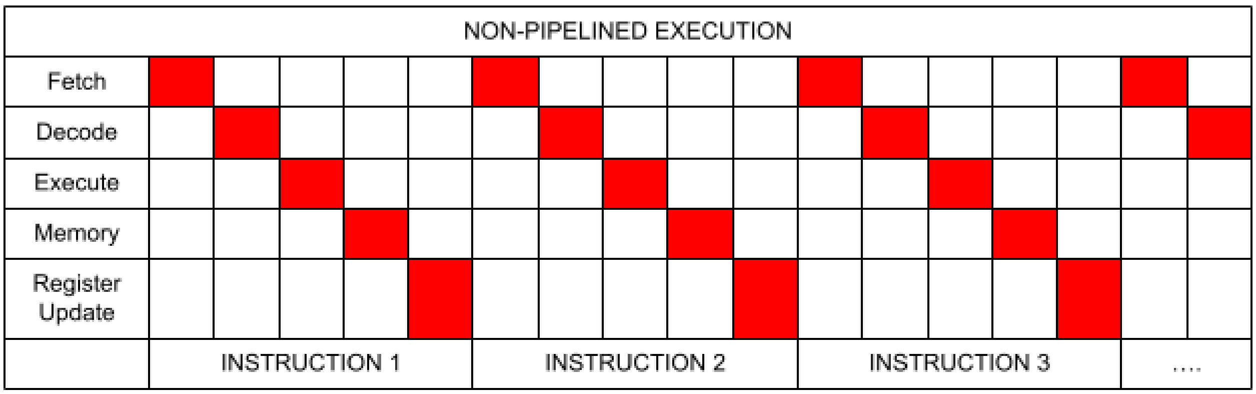
Register

This module maintains the value in registers x0-x31. The module is accessed in the decode and register update stage.

Working

Non-Pipelined Execution

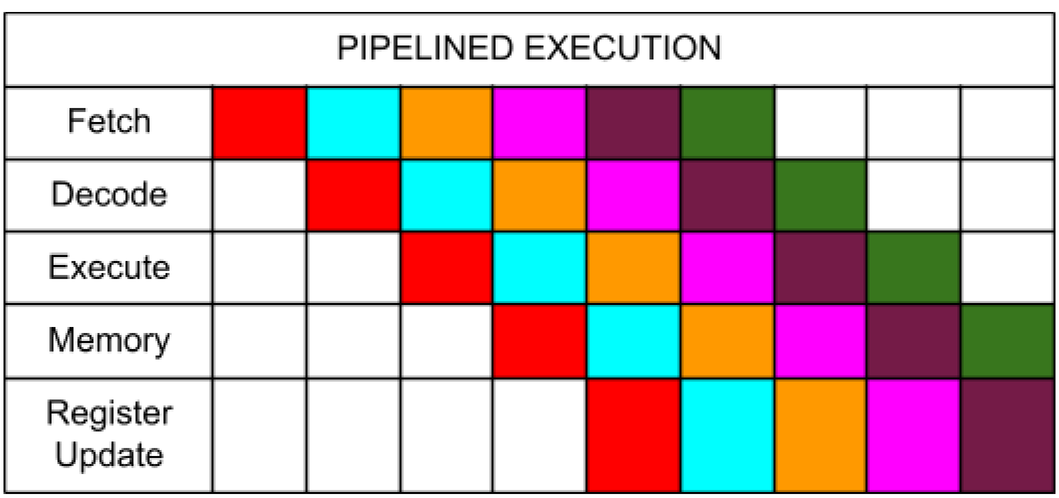
In this method of execution, the simulator executes all 5 steps- fetch, decode, execute, memory access and register update, then moves to the next instruction.



From the image above, we can see that the next instruction is fetched only when the previous instruction has completed its execution, till the register update.
This type of execution takes more time than pipelined execution.

Pipelined Execution

This type of execution increases the efficiency of the hardware and also decreases the execution time. The simulator is executing all 5 steps(fetch, decode, execute, memory access and register update) all at same time, for different instructions.



In the above diagram, similar colors show the same instruction execution.
So we can see that as soon as the instruction 1 reaches the decode stage, then the instruction 2 reaches the fetch stage.
Here 6 instructions are executing faster than 3 instructions in non-pipelined execution.
Pipelined execution demands, faster hardware, since both read and write in the memory is happening at the same time. So the memory module has to be 2 times faster than other stages.
The simulator in pipelined execution executes register update first, then moves to memory access, then to the execute, decode and then fetch.
The simulator will be able to switch between pipelined and non-pipelined execution by just a button.

Measures for Structural Hazards

In case of pipelined execution, the set of instructions in the buffer are executed on the basis of the sub-step each instruction has reached.
For example, if any buffer has any instruction whose only register update step is left, then the simulator executes this instruction first in a cycle.

This solves the problem of faster register module required and double memory module required for instruction and data.

Measures for Data Hazards

The process of checking for data hazard is handled by the instruction detect_data_hazards in the control module. The process is invoked in the Decode step.

A. Stalling

The simulator checks for dependencies between instructions based on the type of instruction and their occurrence, to decide upon stalling the execution.

For example:

```
add x1,x2,x3
sub x6,x1,x7
```

In the above case, the simulator identifies the data hazard due to x1 register in the **Decode Stage**, and is able to direct the simulator to add a stall. Because of this the 2nd instruction stalls for 1 cycle, and the same process for it begins again next cycle.

B. Data Forwarding

Using the same function, used for stalling, the simulator also enables data forwarding. The data forwarding paths supported by the simulator are

1. E to D
2. E to E
3. M to D
4. M to E
5. M to M

In case of data hazard, the data is forwarded to reduce the amount of time lost, and makes the program faster.

The data forwarded has been stored in the buffer, present between sub-steps. For example: in case of E to E data forwarding, the buffer present between execute and memory access provides the data required for the execute stage of the next instruction.

Measure for Control Hazard

For removing control hazards, the simulator has a static branch predictor. The branch predictor always guesses NOT TAKEN for any branch instruction. In case the prediction is a MISS, then the simulator flushes the wrong implemented instructions.

Cache Management- Phase 3

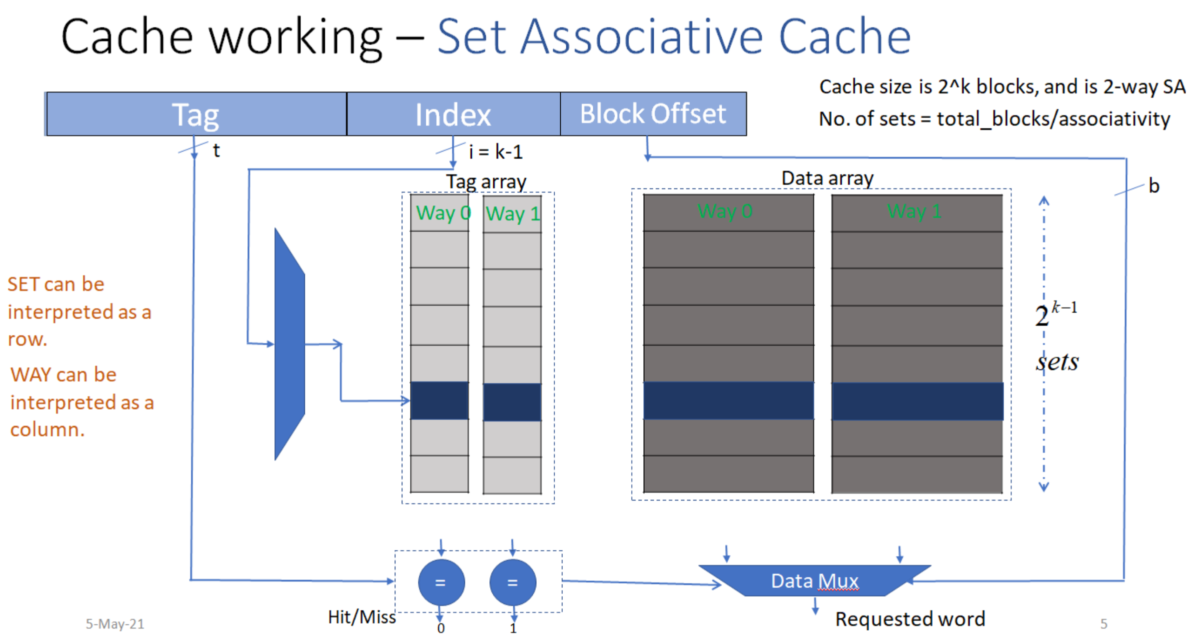
During the phase 3 of the project, the main aim was to introduce cache in the simulator.

Cache is based on SRAM technology. Static because it does not need to refresh to retain its cell state. Cache is used to support the fast processor, by keeping some data near the processor.

Working of the cache

When data is demanded by the processor, then the cache control circuit takes up the memory address and matches it with the tag address. In case of a match, the data from the cache is forwarded to the processor and the case is registered as a hit. In case of a miss, the memory address is forwarded to the next level, and checked for hit.

n-way associative cache



A n way set associative cache means that the cache has n sets, each having m lines. The data will be stored in a specific set, and the cache line will be chosen randomly.

The cache implemented in the simulator uses the write through method. In this method, whenever write is called by the processor, then the data to be written is sent to all levels of the cache. Thus data coherency is maintained.

In the cache implemented in the simulator, the cache circuitry follows LRU. This means that the block used least recently will be eliminated in case the cache is full. The eliminated block is called the victim block, and the preference is measured by a number assigned to it, which determines, how near in the past, the block was used.

Statistics Printed

1. Total number of cycles
2. Total Instruction executed
3. CPI
4. Number of Data-Transfer Instruction Executed
5. Number of ALU Instructions
6. Number of control instruction
7. Number of stalls
8. Number of Data Hazard
9. Number of Control Hazard
10. Number of Branch Misprediction
11. Number of stall due to data Hazard
12. Number of stall due to Control Hazard
13. Number of Cache Accesses, Hits, Misses (for both I$ and D$)
14. Victim block (if any) (for both I$ and D$)

Test Cases

1. Sum till n: sum of first n Natural Numbers
2. Fibonacci: stores first n numbers in a fibonacci series at location 0x10000000.
3. Bubble Sort: sort the elements in the memory at a specified location, using bubble sort.
4. Factorial: Calculates factorial of a number by calling the function recursively. **fact(n)=n*fact(n-1)**

Contribution

Although we worked as a team and helped each other with the work assigned, following can give the idea of work distribution:

1. Aayush Sabharwal- ALU, Control, Pipeline architecture, Updated ALU, Hazard Detection, Data Forwarding, Cache, Documentation
2. Aman Palariya- Memory, Pipeline architecture, Hazard Detection, Updated GUI, Stall Handling, Cache, Documentation
3. Sagalpreet Singh- IAG, GUI, Control (Minor), Pipeline architecture, Hazard Detection, Stall Handling, Cache, Documentation
4. Uday Gupta- Register, Pipeline architecture, Updated Decode Stage, Updated IAG, Counting the statistics, Data Forwarding, Cache, GUI, Documentation
5. Amritanshu Rai- Updated Decode Stage, Updated IAG, Test Cases, Design Document, Data Forwarding, Cache, Documentation