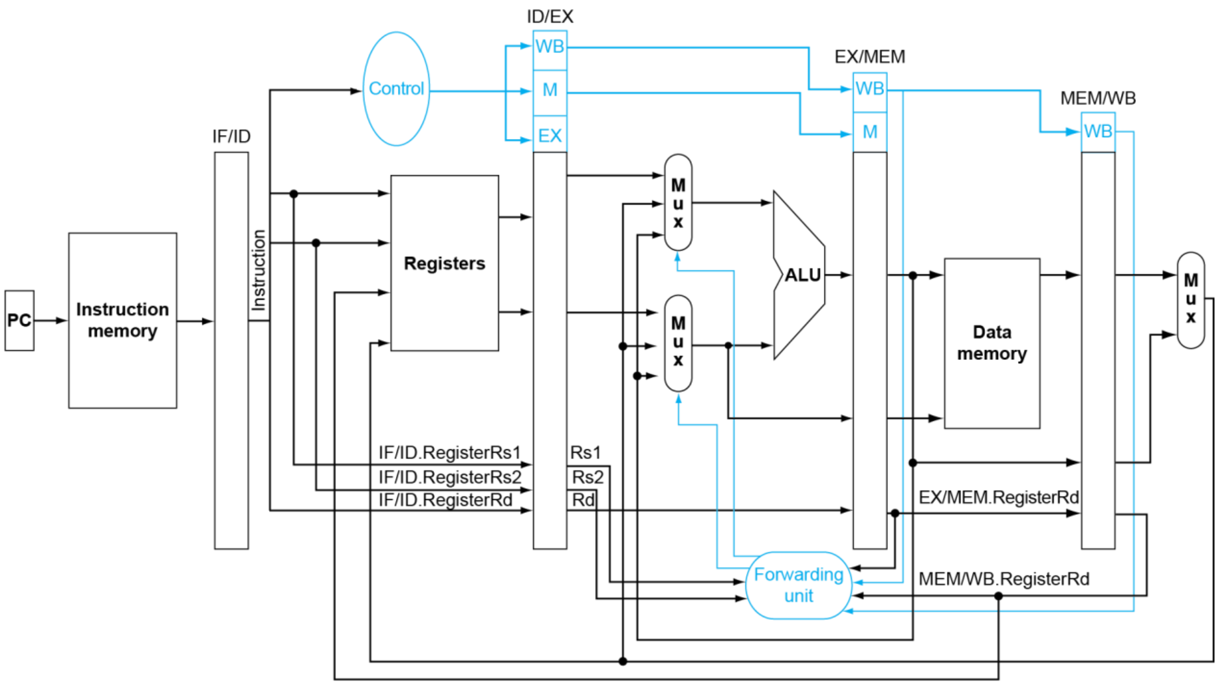
CMPE 344

Computer Organization RISC-V Instruction Set Simulator Project

Muhammed Göktepe – Erencan Uysal

2017400162-2017400069

Introduction:



Main aim of this project is designing a simulator for the datapath and control path that you can see above. The simulator gets a sequence of RISC-V instructions as a text file and then perform a cycle accurate simulation. Our simulator gives output to an output file which you can give its path in terminal while calling simulator.

In the output of the simulator, you can see values in each register, Total clock cycle, total instructions, CPI, number of stalls and the instructions that cause stalls.

We designed our simulator compact and feasible. There is one cpp file in our simulator.

Implementation:

We have 18 functions in total. There are 25 instructions that you can use in our simulator. We assumed all operations are using execution unit for 1 clock cycle. It can be change with ALU. Our ALU can handle all operations in one clock cycle. We also assume 1 nop for load use hazard and 2 flush for branch use hazards. Because our control path has forwarding unit. We used branch always not taken approach for branch prediction. If branch should be taken, we are flushing instructions which in if and id. We are executing instructions in ex phase. We are fetching all instructions from parsedInst array and wait their execution until ex stage.

int ifetch = -1;   // stores instruction fetch state value

int id = -1;   // stores instruction decoding value

int ex = -1;   // stores execution state value

int mem=-1;   // stores memory state value

int wb=-1;   // stores write back state value

We used 5 different integers for 5 state of our datapath. They are initially initialized to -1. It means they are empty. If we flush an instruction from one of them, we change it to -1. They are pointing to indexes of parsedInst array.

double registers[32] = {0};

We used a double array for holding registers values. We can easily change their values with it.

vector<string> instructions;

We used string vector for initially read input file. We are holding each line of input file in this vector. Vector size may vary, so we can find out how much line that we have in input file.

map<string, int> labels;

We used a map to assign instruction index to labels. For example if we use branching, we can find where to go with this map. It holds integer that represents our targer in process counter.

string parsedInst[instructions.size()][4];

We mostly used this 2D array in our code. We initialize this array in main function because instructions size may vary. We are using this array as our process counter. It has 4 fields for each instruction. We can have 4 field at most in RISC-V architecture. For example if our first instruction is add x1, x2, x3, we will have "add" in parsedInst[0][1], "x1" in parsedInst[0][2], "x2" in parsedInst[0][2] and "x3" in parsedInst[0][3].

double memory[65535];

We have a double array to represent memory. We used double data type for the sake of simplicity. If our memory instructions are ld or sd , we can easily take value from memory in one index. If we have lw or sw as memory instruction, the code is making some calculations and find a value. For example if we have lw x1, 0(x2), it takes lower half of the index 0(x2). If we have lw x1, 4(x2), it takes upper half of 0(x2). If we have ld x1, 0(x2), it takes directly 0(x2). You can also change memory size with this array. Size of this array represents the size of memory as byte.

Instructions:

We implemented all instructions we can see in slides. We implemented **add, addi, sub, subi, mul, div, ld, sd, lw, sw, sll, srl, and, andi, or, ori, xor, xori, beq, bne, blt, bge, jal, jalr.**

Main Function:

Most operations are done in main function. Firstly we read input file path and output file path with the arguments of main function. Then we open a stream to input file and read values from it in a for loop. In this part, if we find a colon, it means we find a label. Then we add this line to labels map. Once all instructions read, we are starting to parse them.

In this part, we used another for loop. We initialized parsedInst array and filled it. First we trimmed spaces of lines, then we add lines to this array. If our instruction is I-format, we have 2 registers and an immediate value, so 4 of our indexes are filled. If our instruction is R-format, we have 3 registers, so 4 of our indexes are filled. However if our instruction is S-format, we have 2 registers and one immediate as offset value. So 3 of our indexes are filled.

We initialized a hazard counter here. **hazards[instructions.size()].** It has an index for each instructions. We are incrementing an index, if there is an hazard. We can find which instructions cause stalls this way.

We have a long for loop in our code. It starts with fetching an instruction from instruction memory which is parsedInst array. The instruction to be fetched is decided from w. If there is no branch instruction, we increment w by one. If there is a branch instruction, the result of it determines which instruction will be fetched.

This for loop also provides transitions between pipeline phases. Initially we transport instructions from memory to writeback, from execution to memory, from instruction decoding to execution and from instruction fetch to instruction decoding.

We execute instructions which are in execution phase. Instructions wait for their turn until execution phase.

If our instruction is arithmetic operation, we can forward data and next instruction to be executed can use this data.

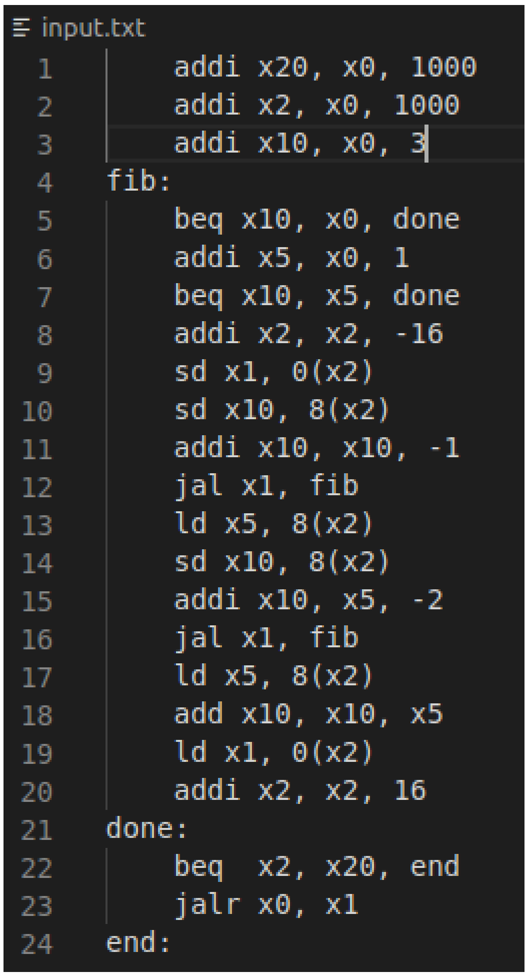
If executed instruction is load, we are looking for next instruction. If destination register of load is source register of next instruction, there is a data hazard. We are handling this hazard and increment stall counter by one.

If executed instruction is branch, we are looking for result of branch comparison. We are using branch always not taken prediction model. If branch should be taken, we are flushing 2 instructions from instruction decoding and instruction fetch states. If branch should not be taken, there will be no stalls and flushing. If there is a stall, we are incrementing stall counter by two.

Finally after this for loop, we are writing outputs to output file which is provided with command line argument.

We are assuming 2 stalls for branch hazard because our datapath calculates result of comparison in execution state, so newly fetched 2 instructions should be flushed from ID and IF. We are assuming 1 stall for load use hazard because our datapath has forwarding unit. Result of load can be pipelined before writeback state.

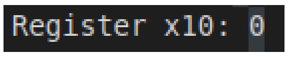
Code simulation:



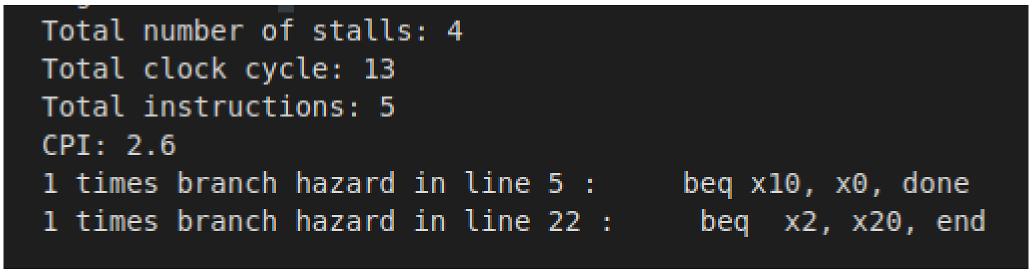
```
input.txt
 1      addi x20, x0, 1000
 2      addi x2, x0, 1000
 3      addi x10, x0, 3
 4  fib:
 5      beq x10, x0, done
 6      addi x5, x0, 1
 7      beq x10, x5, done
 8      addi x2, x2, -16
 9      sd x1, 0(x2)
10      sd x10, 8(x2)
11      addi x10, x10, -1
12      jal x1, fib
13      ld x5, 8(x2)
14      sd x10, 8(x2)
15      addi x10, x5, -2
16      jal x1, fib
17      ld x5, 8(x2)
18      add x10, x10, x5
19      ld x1, 0(x2)
20      addi x2, x2, 16
21  done:
22      beq  x2, x20, end
23      jalr x0, x1
24  end:
```

As you can see, we passed the n variable with x10 register in our code. Firstly we initialize the stack pointer x2. We used 1000 but you can use anything up to memory size. Then we initialize x10 with add statement. We can add “n” to 0 and hold in x10. The code in the book was not operable, so we added “beq x2, x20, end” instruction to done label. With the help of this instruction, we can know when our code is done.

For f(0), there will be 3 instruction before branch. There are two branch instruction after that. So instruction count will be 5, and there will be 2 branch hazard :

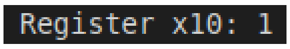


```
Register x10: 0
```

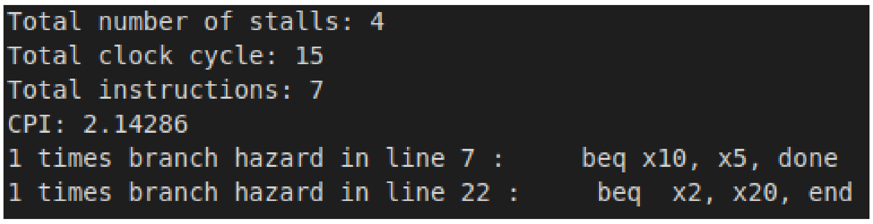


```
Total number of stalls: 4
Total clock cycle: 13
Total instructions: 5
CPI: 2.6
1 times branch hazard in line 5 :     beq x10, x0, done
1 times branch hazard in line 22 :     beq  x2, x20, end
```

For f(1), first branch will not taken but second will taken, so there will be 7 instructions, and 2 branch hazards.

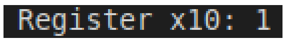


```
Register x10: 1
```

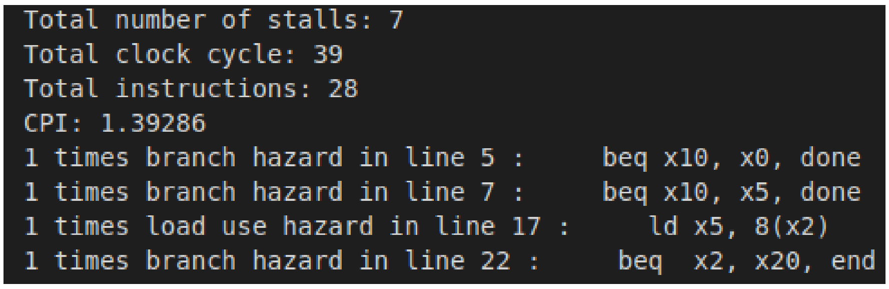


```
Total number of stalls: 4
Total clock cycle: 15
Total instructions: 7
CPI: 2.14286
1 times branch hazard in line 7 :     beq x10, x5, done
1 times branch hazard in line 22 :     beq  x2, x20, end
```

For f(2) we can see that both f(1) and f(0) will be calculated. After that, we will use all instructions to sum them. Then there will be 7+5+16=28 instructions, 1 load use hazard and 3 branch hazards.

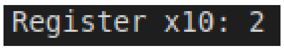


```
Register x10: 1
```

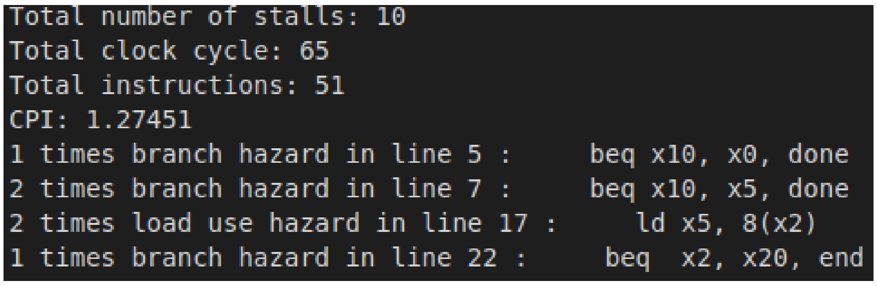


```
Total number of stalls: 7
Total clock cycle: 39
Total instructions: 28
CPI: 1.39286
1 times branch hazard in line 5 :     beq x10, x0, done
1 times branch hazard in line 7 :     beq x10, x5, done
1 times load use hazard in line 17 :     ld x5, 8(x2)
1 times branch hazard in line 22 :     beq  x2, x20, end
```

For f(3), firstly f(2) and f(1) will be calculated. After that we will use all instructions to sum them. Then there will be 28+7+16=51 instructions, 2 times load use hazards and 4 times branch hazards.

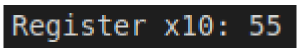


```
Register x10: 2
```

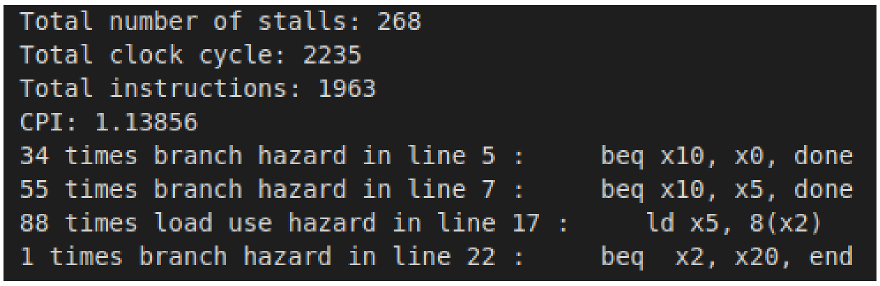


```
Total number of stalls: 10
Total clock cycle: 65
Total instructions: 51
CPI: 1.27451
1 times branch hazard in line 5 :     beq x10, x0, done
2 times branch hazard in line 7 :     beq x10, x5, done
2 times load use hazard in line 17 :     ld x5, 8(x2)
1 times branch hazard in line 22 :     beq  x2, x20, end
```

For f(10):



```
Register x10: 55
```



```
Total number of stalls: 268
Total clock cycle: 2235
Total instructions: 1963
CPI: 1.13856
34 times branch hazard in line 5 :     beq x10, x0, done
55 times branch hazard in line 7 :     beq x10, x5, done
88 times load use hazard in line 17 :     ld x5, 8(x2)
1 times branch hazard in line 22 :     beq  x2, x20, end
```

You can reach our code with the link below. It is well commented.

To compile:

```
g++ project.cpp -o main
```

To run:

```
./main ./input.txt ./output.txt
```

https://drive.google.com/file/d/1PlrduWH4ABMoeKsXkBllu-mJ3wpn_5OQ/view?usp=sharing