Ece Dilara Aslan
Salih Bedirhan Eker

CMPE 344 COMPUTER ORGANIZATION
Project Documentation

1- Introduction:

In this project, we make a simple datapath and control path for add, sub, or, and, ld, sd and beq instructions simulator for the RISC-V architecture by using a high level language (C++). Simulation supports 5 stage pipeline modules with branch prediction (predict as the branch is not taken).

2- Detailed Description of Program Architecture and Modules:

A. General Concept:

Firstly, the program takes a RISC-V source code in a text file as an argument. In the main, instructions and labels are read and written as global data. Then, program initialize pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB) with default values -mostly zero or false- in the main (structure of the pipeline registers explained in later sections). Then, instructions begin to be processed one by one in a while loop. The loop contains the main work, all datapath operations. After the instructions are over, loop breaks and necessary output is written.

B. Global Data and Resources:

string instruction_memory[1024][6]: This is used as storing the instructions. Each line of the matrix represents an instruction.
instruction_memory[i][0] => instruction type,
instruction_memory[i][1] => source register 1,
instruction_memory[i][2] => source register 2,
instruction_memory[i][3] => destination register,
instruction_memory[i][4] => offset,
instruction_memory[i][5] => label name.

For example if the instruction is add x1, x2, x3, ==>>
instruction_memory[i][0] = “add”,
instruction_memory[i][1] = “2”,
instruction_memory[i][2] = “3”,
instruction_memory[i][3] = “1”,
instruction_memory[i][4] = ”” (not used, empty string as default),
instruction_memory[i][5] = “” (not used, empty string as default).

vector<string> instructions: This is used for accessing the instructions that causing stalls. It stores the instructions line by line as strings in the right order.

map<string, long int> labels: Holds the label names and address of subsequent instruction of it. When a branch is taken we use this map to find the value of the PC.

long int PC: Used as program counter.

long int registers[32] = {0}: Used as registers x0-x31. Initially all of them are zero.

long int data_memory[4096]: Data memory of the program and used to store the data, initially all entries are zero to ease of the program. This memory is designed as doubleword addressable memory. Each cell represents 64-bits of data.

bool IFID_write = true: This flag is used only in stalls, it is true as default, however when the program needs a load-use stall, it is assigned as false for one cycle temporarily.

int cycle: Keeps the number of the cycles.

int stall: Keeps the number of stalls.

vector<string> stall_inst: Keeps the instructions that are caused the stall and after the program is done, we print these instructions.

C. Pipeline Registers:

Pipeline registers are the fundamental parts of the program. Each of the pipeline register is implemented as a struct data type and we have 2 of all, new and old. Each struct has fields that are necessary for calculations and control flags. Also they have a function named ***copy()***. The use of ***copy()*** function is that when we are in a new cycle we need to carry the data forward and in order not to lose the previous data, we create a copy of the old value of a pipeline register. (Note that each ***copy()*** function is seperate, because every pipeline register has separate fields and control flags)

D. Helper Functions:

void line_handler(vector<string> tokens, int count): This helper function is used in reading the input. It takes tokens of an input line and puts it in ***string instruction_memory[i][6]*** in a proper form. You can find more details about it in the comment section, we are not going into details of it because it does not provide anything special about pipelined datapath.

void control(string inst_name, struct ID_EX * new_IDEX): In IF/ID register, program gets an instruction and when it needs to create the control lines, we use this function. This function gets the name of the instruction and sets all the control flags according to the name of the instruction. This operation is done in the instruction decode stage -ID-.

E. Code of Data and Control Paths:

Main code of the program is the ***while(PC != inst_count+4)*** in the main. All operations are done here. Each iteration of the while loop represents a cycle. In each iteration we have several parts:

1- Copying new values of the previous cycle to old values.
2- Write the results to register operations, **WB** stage.
3- Instruction Fetch, **IF** stage.
4- Instruction Decode, **ID** stage.
5- Calculating the address or the arithmetic/logic operations, **EX** stage.
6- Memory access, **MEM** stage.
7- ***Hazard detection unit*** of load-use hazards.
Also we have a ***forwarding unit*** in **EX** stage and ***branch prediction unit*** in **MEM** stage.

1- Copying new values to old values:

At the beginning of an iteration we copy the values in order not to lose them. Previous new_values are current old_values now so we copy them (old ⇐new). Moreover, if we are dealing with the load-use stalls, the **IF/ID** register must not be changed so use ***IFID_write*** to control this condition.

2- WB stage:

In order to read the right value of source registers, **WB** stage must be implemented before the **ID** stage. Register access is very fast and in a cycle ve can both write the data (in the first half of the cycle) and then we can read data (in the second half of the cycle) from the registers. Also, if the destination register is x0, the write back operation is not done to keep its value 0.

3- IF stage:

We already created the instruction memory and know the value of the PC. Therefore just read the information in the instruction memory and write necessary values to the **IF/ID** register.

4- ID stage:

We already know the name of the current instruction. In this part first write the necessary values to **ID/EX** register. However if we try to write or alter the wrong values it may cause an error, so control the values that we are trying to convert to integers that exist in that place. For example, if the current instruction is add, use ***source1***, ***source2*** and ***destination***. If we try to convert to integer the ***immediate*** value we get an error, because its default value is “” (empty string). After we put the resources into the **ID/EX** register now we need to set the control lines. We use the ***control()*** helper function in order to set the control flags.

5- Ex stage:

At the beginning of the **EX** stage, we need to carry the necessary data forward from **ID/EX** register to **EX/MEM** register. Then, we have a ***forwarding unit***. We must check whether any source register values need to be forwarded or not. If so, register value from **MEM** or **WB** stage is copied to **EX** stage before it is used for calculation. After the forwarding process, the program makes the address calculation or necessary arithmetic/logic operations according to flags. And at last, the program carries the control flags that need to be used in later stages.

6- MEM stage:

At first, the program carries the data that is used in later stages. Later, the program makes reading and writing operations in data memory. Also, we have a branch prediction unit in this stage. **Program predicts branch as not taken**. If so, the program does not change anything. However if branch is taken, then 3 instructions after beq instruction must be flushed from the pipeline. It is done by setting control values of **EX/MEM** and **ID/EX** registers to 0, also setting the name of the instruction in **IF** stage to “” (empty string) which will cause this instruction to have control values as 0 in ***control()*** function. In order to take the branch, PC is set to branch target minus 1 because it will be increased by 1 at the end of the cycle. Also, ***stall*** value is increased by 3 due to 3 stalls and this instruction with the corresponding number of stalls is added to the ***stall_inst*** variable. At last, control flags of the **MEM** stage are carried to the **MEM/WB** register.

7- Hazard detection unit of load-use hazards:

In this unit, first we need to check whether we have any branch hazard and *load/use* hazard at the same time. If so, *load/use* hazard is ignored due to the flushing of *load/use*. Otherwise, the program checks if any *load/use* hazard exists and if so the program adds one stall. In order to do this, we set all flags to zero in the **ID/EX** register, decrease **PC** by 1 because it will be increased by 1 at the end of the cycle and set ***IFID_write*** to *false* to keep the same instruction in the **ID** stage. Also, ***stall*** value is increased by 1 due to 1 stall and we add those instructions with the corresponding number of stalls to the ***stall_inst*** vector. After the datapath operations end, we print these instructions that caused stalls.

F. Writing Outputs

CPI is calculated by dividing the *number of cycles* by *number of instructions*. Then, CPI, number of cycles, number of stalls and instructions that caused stalls with the corresponding number of stalls are written.

3- Input Format and Compile/Run the Code:

Program takes a .txt file as an argument. This .txt file consists of RISC-V instructions (add, sub, and, or, ld, sd, beq) and must not have any empty lines.

Compiler version:
g++ (apple clang 11.0.3)

Compiling the code:

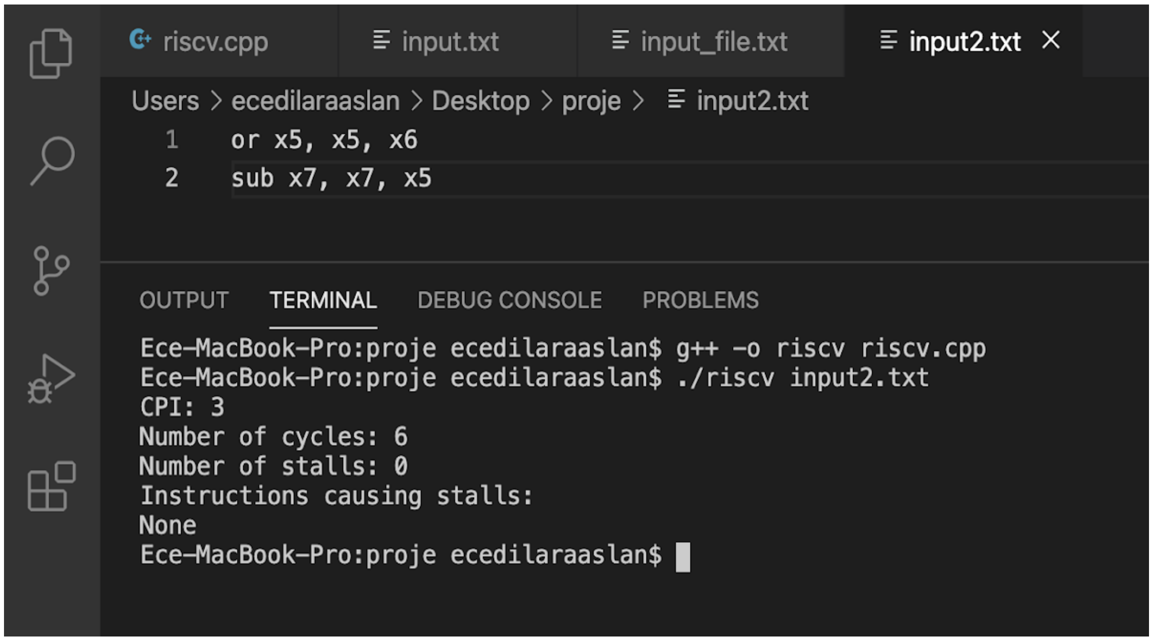
g++ riscv.cpp -o riscv

Running the code:

./riscv input_file.txt

4- Example Input and Output:

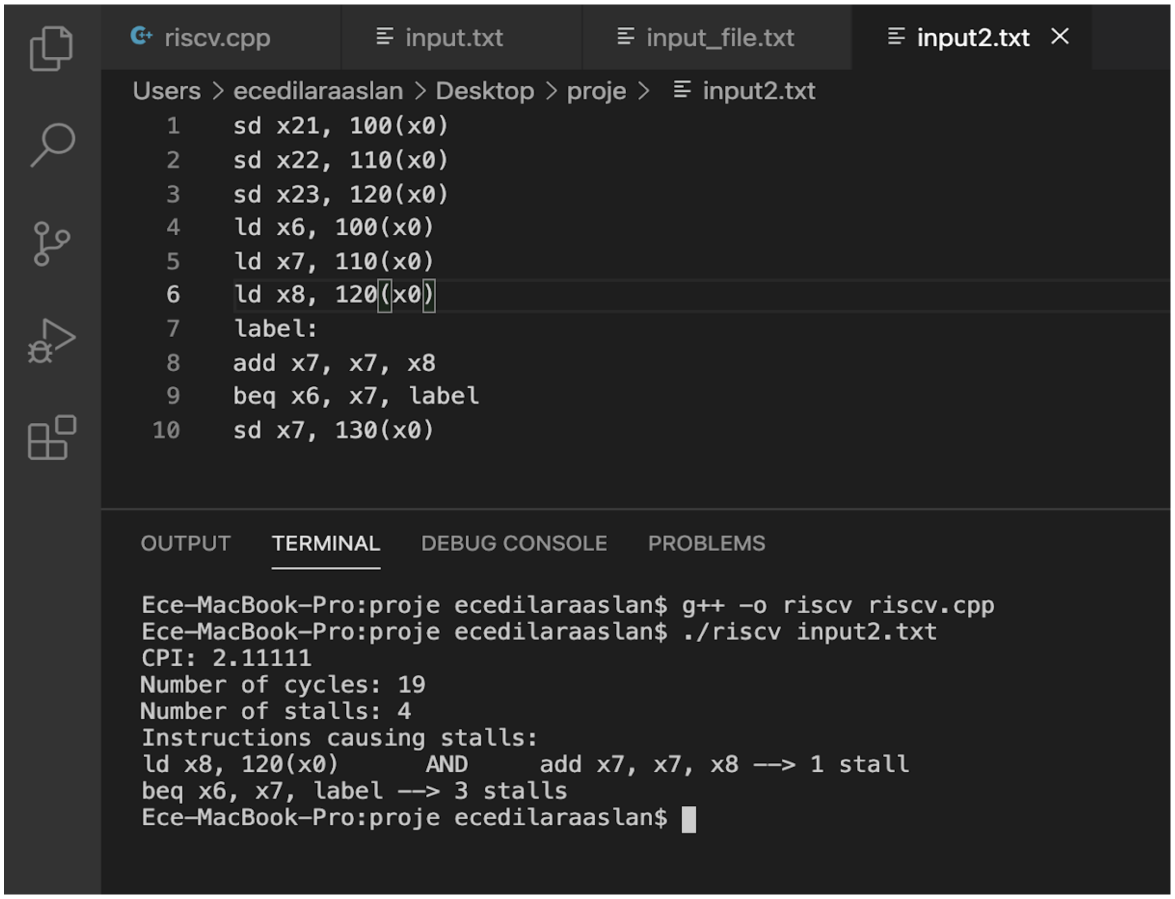
- with initial values x5 = 10, x6 = 20, x7 = 35



```
or x5, x5, x6
sub x7, x7, x5
```

```
Ece-MacBook-Pro:proje ecedilaraaslan$ g++ -o riscv riscv.cpp
Ece-MacBook-Pro:proje ecedilaraaslan$ ./riscv input2.txt
CPI: 3
Number of cycles: 6
Number of stalls: 0
Instructions causing stalls:
None
Ece-MacBook-Pro:proje ecedilaraaslan$
```

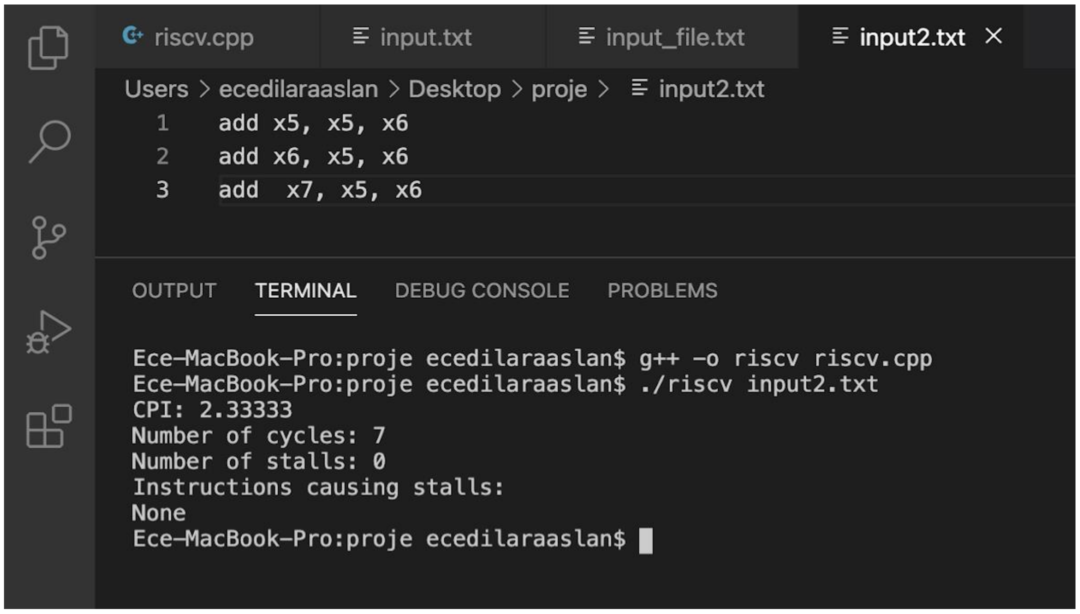
- with initial values x21 = 50, x22 = 40, x23 = 10



```
sd x21, 100(x0)
sd x22, 110(x0)
sd x23, 120(x0)
ld x6, 100(x0)
ld x7, 110(x0)
ld x8, 120(x0)
label:
add x7, x7, x8
beq x6, x7, label
sd x7, 130(x0)
```

```
Ece-MacBook-Pro:proje ecedilaraaslan$ g++ -o riscv riscv.cpp
Ece-MacBook-Pro:proje ecedilaraaslan$ ./riscv input2.txt
CPI: 2.11111
Number of cycles: 19
Number of stalls: 4
Instructions causing stalls:
ld x8, 120(x0)      AND     add x7, x7, x8 --> 1 stall
beq x6, x7, label --> 3 stalls
Ece-MacBook-Pro:proje ecedilaraaslan$
```

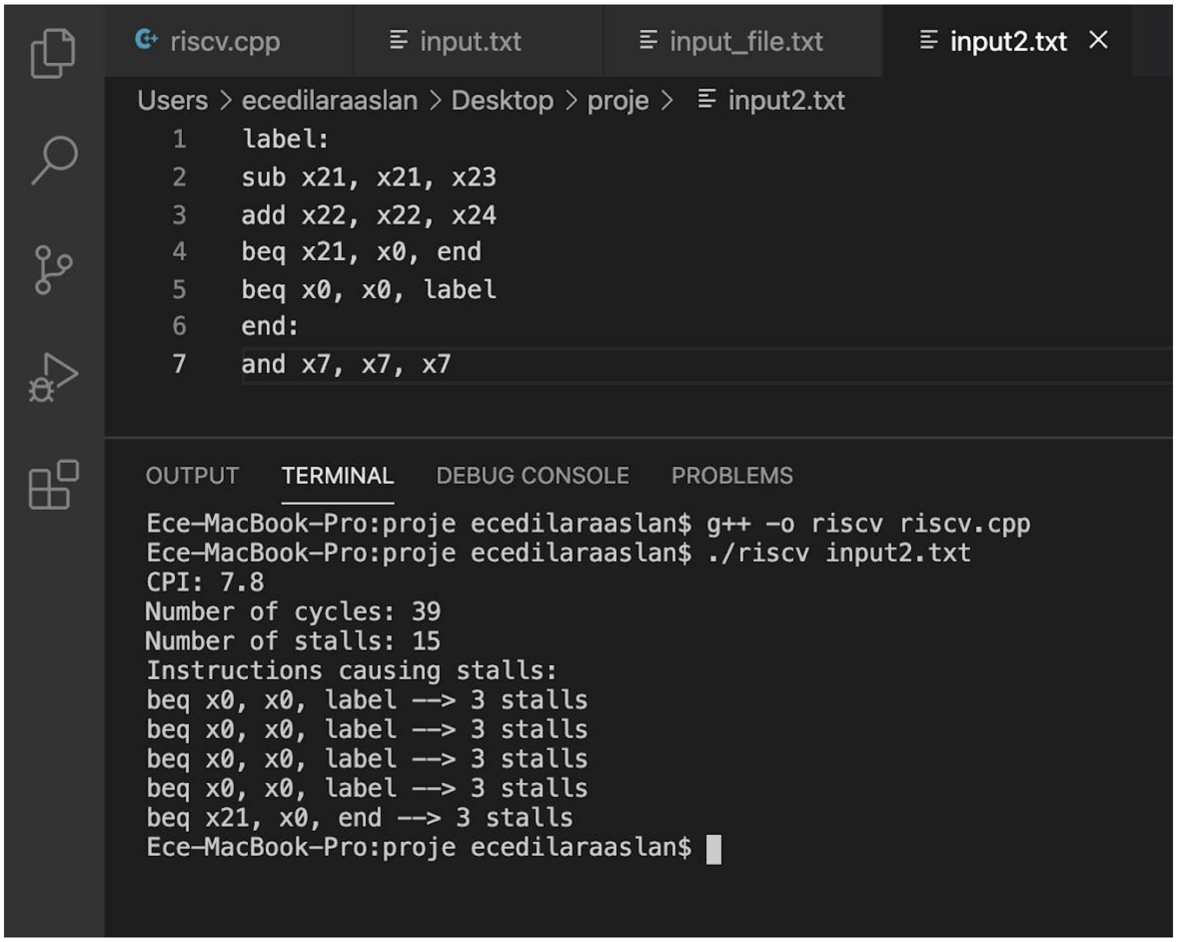
- with initial values x5 =10, x6 = 20, x7 = 30



```
add x5, x5, x6
add x6, x5, x6
add  x7, x5, x6
```

```
Ece-MacBook-Pro:proje ecedilaraaslan$ g++ -o riscv riscv.cpp
Ece-MacBook-Pro:proje ecedilaraaslan$ ./riscv input2.txt
CPI: 2.33333
Number of cycles: 7
Number of stalls: 0
Instructions causing stalls:
None
Ece-MacBook-Pro:proje ecedilaraaslan$
```

- with initial values x21 = 5, x22 = 11, x23 = 1, x24 =11



```
label:
sub x21, x21, x23
add x22, x22, x24
beq x21, x0, end
beq x0, x0, label
end:
and x7, x7, x7
```

```
Ece-MacBook-Pro:proje ecedilaraaslan$ g++ -o riscv riscv.cpp
Ece-MacBook-Pro:proje ecedilaraaslan$ ./riscv input2.txt
CPI: 7.8
Number of cycles: 39
Number of stalls: 15
Instructions causing stalls:
beq x0, x0, label --> 3 stalls
beq x0, x0, label --> 3 stalls
beq x0, x0, label --> 3 stalls
beq x0, x0, label --> 3 stalls
beq x21, x0, end --> 3 stalls
Ece-MacBook-Pro:proje ecedilaraaslan$
```