The American University in Cairo

Computer Architecture

Project 2

Tomasulo Algorithm Simulation

Ahmed Osama Hassan

900171850

Mohamed Ashraf Hemdan

900171923

Wednesday , July 22, 2020

Submitted to:

Dr. Cherif Salama

In this project we try to simulate Tomasulo's algorithm for single issue applications. The algorithm is implemented for a RISC processor that uses of a 16-bit addressable space. Our goal is to simulate different kind of reservation stations of different capacities that performs different kind of operations based on the instructions of interest. The coding language of choice to solve this simulation problem was c++ since it enables the use of pointers with ease and many pointers were to be deployed to achieve the design we had in mind. Thus to implement this simulation in code 4 different approaches to problem solving were considered:

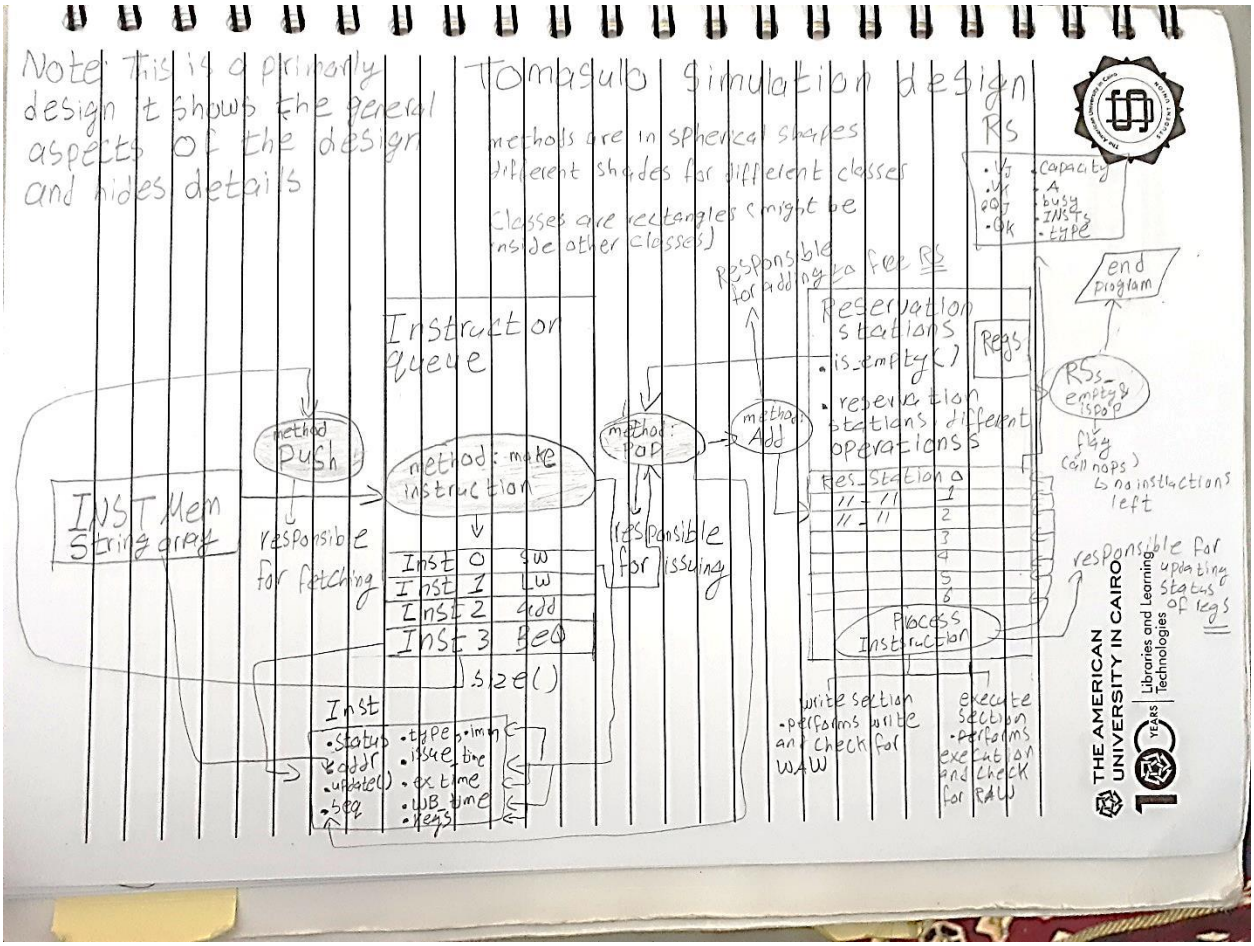
1) Imperative:
   We figured this might be a little lousy since we are supposed to simulate real hardware components so it would be better to not just have a bunch of functions that don't represent this hardware as objects and consequently doesn't emphasize the communication between them
2) OOP:
   This Approach was found more suitable to our task at hand since it allowed us to think of each hardware component independently and figure out the attributes and methods it will need as well as allowing us to use things like inheritance which would come out handy later in implementation. The different components were later passed to each other to allow communication between them.
3) Functional:
   For this approach it was hard to design a solution since it isn't very famous for using loops to reach solutions and many loops were needed to finish the design of our simulator. Also order of execution as well as state changes are important for the reservation stations as well as the instruction queue which again is not the functional approach strongest point.
4) Logic:
   It needed a bit of mathematical background which would have needed some more time and added more difficulty to come up with a solution, although it might have worked but the idea was ignored.

So the programming paradigm used to implement our simulation in C++ was OOP since it met most of the requirements needed and also the familiarity with it was superior to that of the Logic approach.

The design also took into consideration the simulation of a memory that asks the user of his preferred method to read the instruction whether from a file or from keyboard, the OOP implementation of this component was ignored since it wouldn't have too many methods nor does it communicate with that many other components so it was implemented in an imperative manner and only its array output was used in the OOP implementation of the rest of Tomasulo simulation design.

Solution design:

This design implements the OOP to achieve the Tomasulo of the risc v processor under the single-issue policy:



Design Challenges and solutions

1- The presence of Data Hazards
   Since the Tomasulo Algorithm performs out-of-order execution other types of data dependencies, other than RAW, can result in hazards this certain restrictions need to be placed on each stage: issue, execute and Write. The other two dependencies are the WAR and WAW.

2- The presence of Control Hazards
   The project implements different kind of instructions included in these is the beq instruction which relies on an offset prediction that may or may not be true. In case of it not being true then the flushing of instructions inside the instruction queue need to be handled accordingly as to have correct results.

3- The variant operations depending on particular type of RS
   The many different types of reservation stations has to have different methods since they have different operations because each handle a different instruction out of the 11 instructions that we support in this project.

Design Solutions:

1- Data Hazard Handling methods
   The OOP is made use of in abstracting the different hardware components that are present in the design, for example the instruction queue eliminates the WAR hazards by implementing restrictions on poping the instructions to reservation stations in particular, you can't issue unless you do so in order, the abstraction seamlessly achieved this function while the next class, reservation stations, had not to worry about this issue. In the same manner the RS class contains the function process_instruction which handles the RAW so that clocking can happen reducing the execution time of all instructions, given they don't depend, without the need to specify certain instructions for clocking; the same process_instruction also handles WAW instruction by checking on instructions so that they cannot write if their rd is the same as a prior instruction making use again of the abstraction in the instruction class which saves the SEQ , the order according to issuing, to decide whether the instruction with the same rd would or wouldn't cause stall.
2- Control Hazard Handling methods.
   The RS also make use of the abstraction of instructions to get the immediate from branch instructions to decide based on it the prediction of the jump if feasible and later to execute the jump given the right methods. The RS can also communicate with the queue to make it flush instructions if it turns out afterwards that the prediction was wrong, hence appears the importance of communication between objects of different classes that the OOP was chosen for.
3- Inheritance and polymorphism
   To solve the problem of having the same attributes but different methods we can apply one of OOP most useful features namely inheritance to inherit all the attributes and other functions and use polymorphism to override the methods of the children to make it specific to the chosen operation depending on the type of the reservation station. Finally we made use of the fact that we can use a parent pointer to create an array of child objects which gave us the opportunity to have the different types of reservations stations in one array inside the RS class; this is another plus for using c++ as the language of implementation.

Testing Procedures:

We used several test programs to test our Tomasulo simulator. All of them try to pay attention to the Data and control Hazards that may arise from the out-of-order execution.

Tests:

1) First test: checks the different Arithmatic instructions.

First Test: Simple test over arithmetic operations whether R-format or immediate arithmatic

Addr	Instruction	Fetch	Issue	Execute	Write
1	ADDI R1, R0, 15	0	1	2-3	4
2	ADDI R2, R0, 18	0	2	3-4	5
3	ADD R3, R1, R2	0	3	5-6	7
4	SUB R3, R2, R1	0	5	6-7	8
5	MULT R4, R3, R2	1	6	8-15	16
6	NAND R5, R1, R4	2	7	16	17

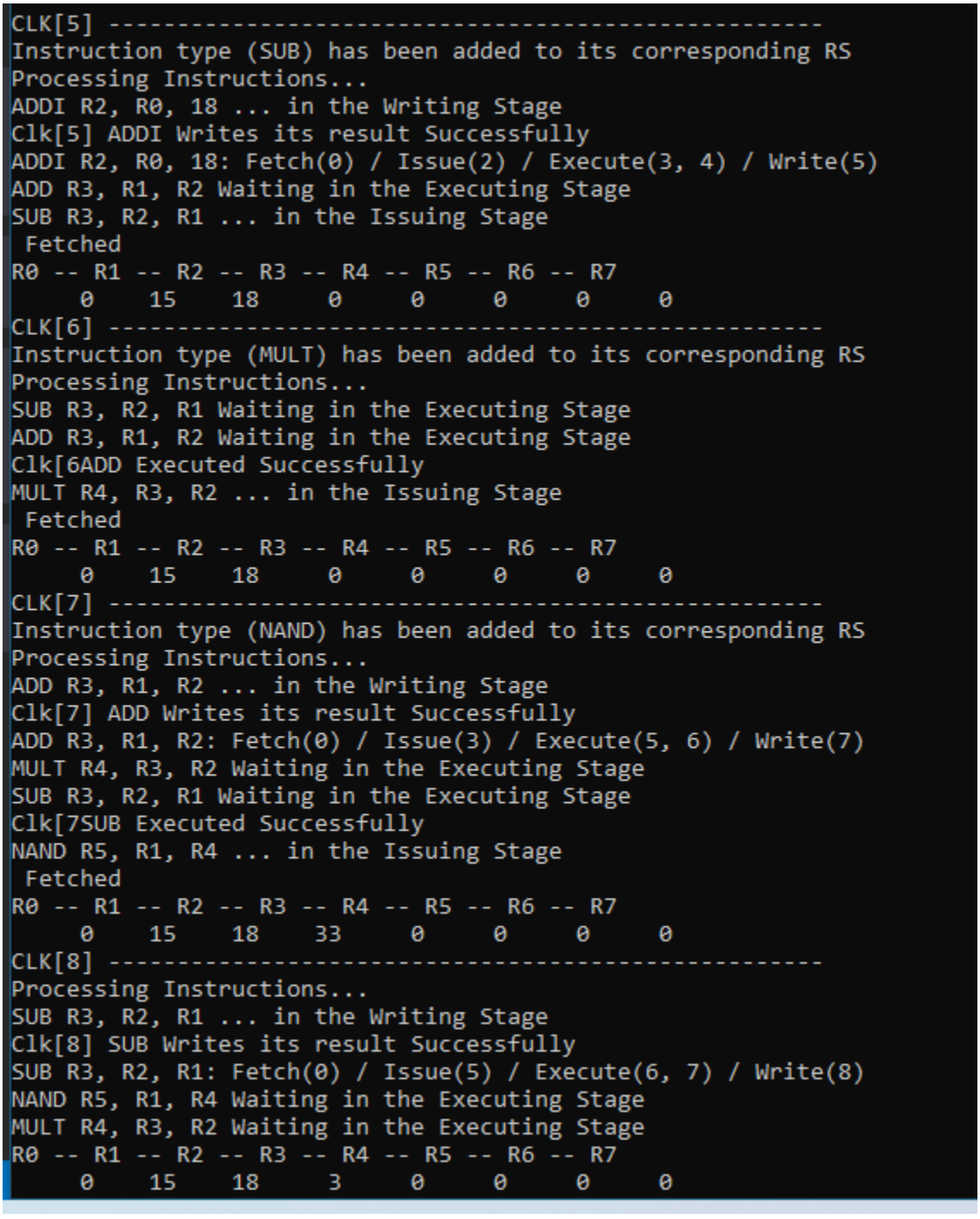
This is a test that doesn't use the memory at data at all hence all data memories are initialized to zeros

Here are some screenshots from the simulation of the first test:

```
If you want a detailed instruction processing, enter (y):n
ADDI R1, R0, 15: Fetch(0) / Issue(1) / Execute(2, 3) / Write(4)
ADDI R2, R0, 18: Fetch(0) / Issue(2) / Execute(3, 4) / Write(5)
ADD R3, R1, R2: Fetch(0) / Issue(3) / Execute(5, 6) / Write(7)
SUB R3, R2, R1: Fetch(0) / Issue(5) / Execute(6, 7) / Write(8)
MULT R4, R3, R2: Fetch(1) / Issue(6) / Execute(8, 15) / Write(16)
NAND R5, R1, R4: Fetch(2) / Issue(7) / Execute(16, 16) / Write(17)
IPC -> 0.315789
```

As you can see the clocks where each instruction was fetched, issued, executed and written matches the hand-solved program in the table so we can say from this that at least our Data-hazard handling and other OOP classes and their communication is functioning correctly.

Also our simulation gives you the option to choose to see the short and final version of instruction execution like this or to inspect the detailed instructions processing, here is a screenshot for the same program if you choose to answer the previous question with “yes”:



This shows detailed information about the instructions fetched, issued, exected, written as well as the values of the register file.

2) Second test:

The following test was mainly designed to test the beq instruction and make sure our program can handle the resulting control hazards and deal with the aftermath of wrong predictions by applying flushing according to the sequential number as well as flushing instructions in instructions queue:

Addr	Instruction	Fetch	Issue	Execute	Write
1	ADDI R1, R0, 18	0	1	2-3	4
2	MULT R3, R1, R0	0	2	4-11	12
3	BEQ R0, R0, 3	0	3	4	5
4	ADDI R2, R0, 3	0	4	-	-
5	SUB R3, R1, R2	1	-	-	-
6	ADD R4, R3, R2	2	-	-	-
7	ADD R7, R3, R1	4	5	12-13	14
8	SUB R4, R7, R3	4	7	14-15	16

This is a test that doesn't use the memory at data at all hence all data memories are initialized to zeros

Here are some screenshots from the simulation of the second test:

```
[31]
If you want a detailed instruction processing, enter (y):n
ADDI R1, R0, 18: Fetch(0) / Issue(1) / Execute(2, 3) / Write(4)
Processor flushed
BEQ R0, R0, 3: Fetch(0) / Issue(3) / Execute(4, 4) / Write(5)
MULT R3, R1, R0: Fetch(0) / Issue(2) / Execute(4, 11) / Write(12)
ADD R7, R3, R1: Fetch(4) / Issue(5) / Execute(12, 13) / Write(14)
SUB R4, R7, R3: Fetch(4) / Issue(6) / Execute(14, 15) / Write(16)
IPC -> 0.277778
```

Here don't mind this small problem in printing however the sequence written is correct, since the first ADDI and MULT were issued first then the processor flushed finally only the instructions after the jump were executed and notice how both the last instructions were fetched in cycle 4 (after the beq executed) finally as you can see both the instructions fetched at the same time since we fetch as many instructions as the free spaces in the queue have (like at the start of any program for instructions are fetched at once, then a single instruction per cycle) however here since we flushed all the instructions in

the queue we could fetch up to 4 instructions however these two instructions were the last in the program so we made do with these.

For better inspection of the instruction processing here is a screenshot for the same program with more detailed information:

```
CLK[3] ----------------------------------------------------
Instruction type (BEQ) has been added to its corresponding RS
Processing Instructions...
MULT R3, R1, R0 Waiting in the Executing Stage
ADDI R1, R0, 18 Waiting in the Executing Stage
Clk[3ADDI Executed Successfully
BEQ R0, R0, 3 ... in the Issuing Stage
ADD R7, R3, R1 Fetched
R0 -- R1 -- R2 -- R3 -- R4 -- R5 -- R6 -- R7
     0     0     0     0     0     0     0     0
CLK[4] ----------------------------------------------------
Instruction type (ADDI) has been added to its corresponding RS
Processing Instructions...
ADDI R1, R0, 18 ... in the Writing Stage
Clk[4] ADDI Writes its result Successfully
ADDI R1, R0, 18: Fetch(0) / Issue(1) / Execute(2, 3) / Write(4)
MULT R3, R1, R0 Waiting in the Executing Stage
BEQ R0, R0, 3 Waiting in the Executing Stage
Processor flushed
Clk[4BEQ Executed Successfully
ADD R7, R3, R1 Fetched
SUB R4, R7, R3 Fetched
 Fetched
 Fetched
R0 -- R1 -- R2 -- R3 -- R4 -- R5 -- R6 -- R7
     0    18     0     0     0     0     0     0
CLK[5] ----------------------------------------------------
Instruction type (ADD) has been added to its corresponding RS
Processing Instructions...
BEQ R0, R0, 3 ... in the Writing Stage
Clk[5] BEQ Writes its result Successfully
BEQ R0, R0, 3: Fetch(0) / Issue(3) / Execute(4, 4) / Write(5)
MULT R3, R1, R0 Waiting in the Executing Stage
ADD R7, R3, R1 ... in the Issuing Stage
 Fetched
R0 -- R1 -- R2 -- R3 -- R4 -- R5 -- R6 -- R7
     0    18     0     0     0     0     0     0
CLK[6] ----------------------------------------------------
```

Here you can see that the ADD instruction was fetched again after the beq executed along with the following instruction since the queue is now flushed and cleared as discussed before.

There should have been a third test, for the immediate arithmetic instructions in specially however some of these instructions were tested here successfully also immediate arithmetic instructions were test during the previous milestone so since some of them are still working the rest is assumed to be working.

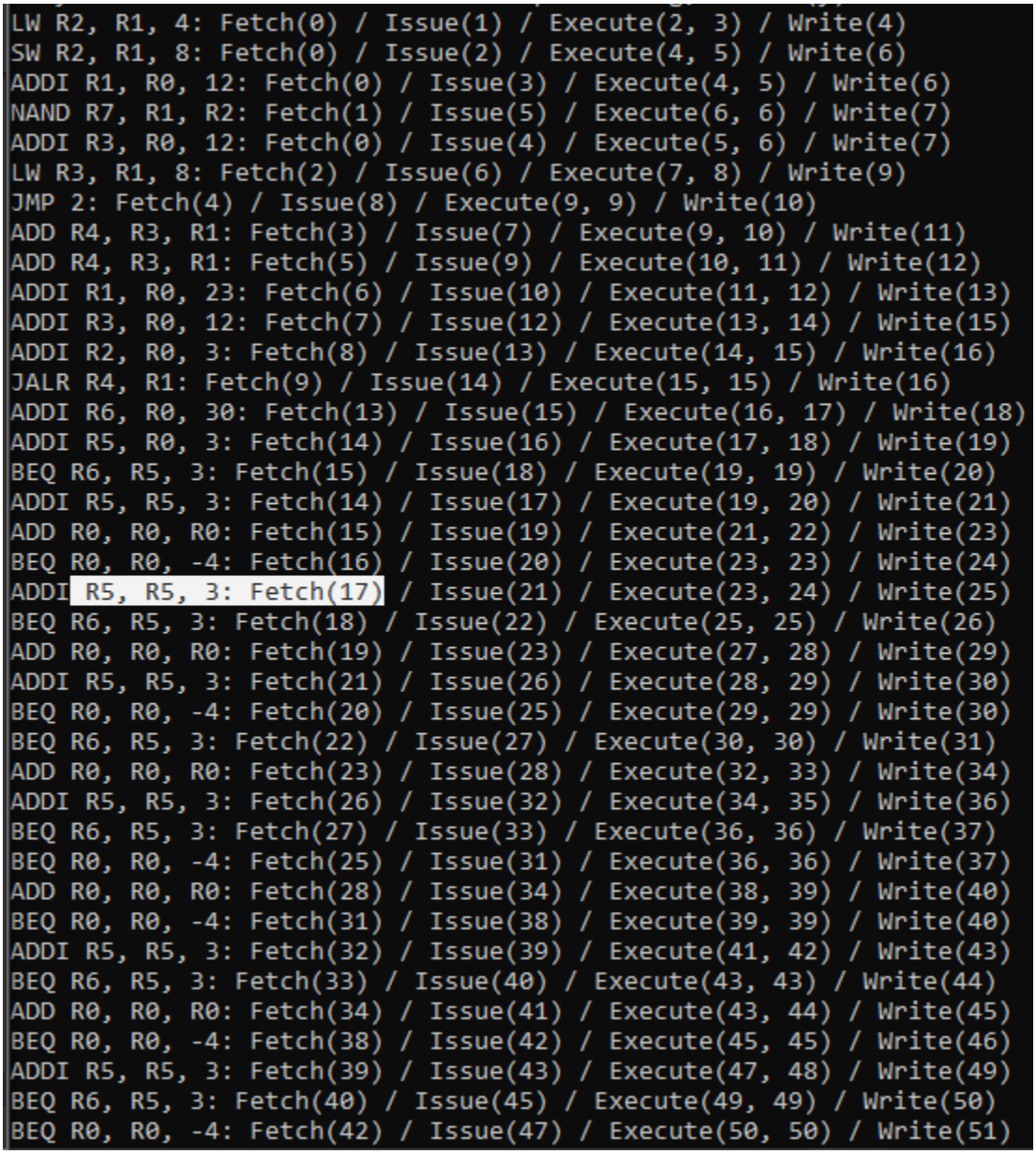
3) Third test:

This test was made to test branching, jalr, ret, jmp instructions and load/store ones.

Below is the test in assemply language along with the predicted values for issue, fetch, execute, and write

Addr	Instruction	Fetch	Issue	Execute	Write
0	LW R2, R1, 4	0	1	2-3	4
1	SW R2, R1, 8	0	2	4-5	6
2	ADDI R1, R0, 12	0	3	4-5	6
3	ADDI R3, R0, 12	0	4	5-6	7
4	NAND R7, R0, 13	1	5	6	7
5	LW R3, R1, 8	2	6	7-8	9
6	ADD R4, R3, R1	3	7	9-10	11
7	JMP 2	4	8	9	10
8	ADDI R1, R0, 6	-	-	-	-
9	NAND R2, R0, R0	-	-	-	-
10	ADD R4, R3, R1	5	9	10-11	12
11	ADDI R1, R0, 23	6	10	11-12	13
12	ADDI R3, R0, 12	7	12	13-14	14
13	ADDI R2, R0, 3	8	13	14-15	16
14	JALR R4, R1	9	14	15	16
15	ADDI R3, R1, 8	-	-	-	-
16	ADD R5, R3, R1	-	-		-
17	ADDI R1, R0, 6	-	-		
18					
19					
20					
21					
22					
23					
24	ADDI R6, R0, 30	13	15	16-17	18
25	ADDI R5, R0, 3	14	16	17-18	19
26	ADDI R5, R5, 3	14	17	19-20	21
27	BEQ R6, R5, 3	15	18	21	22
28	ADD R0, R0, R0	15	19	21-22	23
29	BEQ R0, R0, -4	16	20	23	24
30	ADD R0, R0, R0	---			
31	RET R4				

Here is a screenshot from the program while running



```
ADD R0, R0, R0: Fetch(41) / Issue(46) / Execute(48, 49) / Write(52)
ADDI R5, R5, 3: Fetch(43) / Issue(48) / Execute(52, 53) / Write(54)
BEQ R0, R0, -4: Fetch(47) / Issue(53) / Execute(56, 56) / Write(57)
BEQ R6, R5, 3: Fetch(45) / Issue(51) / Execute(56, 56) / Write(57)
ADD R0, R0, R0: Fetch(46) / Issue(52) / Execute(54, 55) / Write(59)
ADDI R5, R5, 3: Fetch(48) / Issue(54) / Execute(58, 59) / Write(60)
BEQ R6, R5, 3: Fetch(51) / Issue(58) / Execute(59, 59) / Write(60)
ADD R0, R0, R0: Fetch(52) / Issue(59) / Execute(61, 62) / Write(63)
BEQ R0, R0, -4: Fetch(53) / Issue(60) / Execute(63, 63) / Write(64)
ADDI R5, R5, 3: Fetch(54) / Issue(61) / Execute(63, 64) / Write(65)
BEQ R6, R5, 3: Fetch(58) / Issue(62) / Execute(65, 65) / Write(66)
ADD R0, R0, R0: Fetch(59) / Issue(63) / Execute(67, 68) / Write(69)
ADDI R5, R5, 3: Fetch(61) / Issue(66) / Execute(68, 69) / Write(70)
BEQ R0, R0, -4: Fetch(60) / Issue(65) / Execute(69, 69) / Write(70)
Processor flushed
BEQ R6, R5, 3: Fetch(62) / Issue(67) / Execute(70, 70) / Write(71)
RET R4: Fetch(70) / Issue(71) / Execute(76, 76) / Write(77)
IPC -> 0.683544
Branch Misprediction percentage -> 0.0526316
```

We adjusted the fetching to work based on dependencies. Therefore, prediction is conditioned so that the instruction writing to the source register of the branch instruction should execute first before predicting the branch. This is illustrated in the below picture

```
LW R3, R1, 8 ... in the Writing Stage
Clk[9] LW Writes its result Successfully
LW R3, R1, 8: Fetch(2) / Issue(6) / Execute(7, 8) / Write(9)
ADD R4, R3, R1 Waiting in the Executing Stage
JMP 2 Waiting in the Executing Stage
Clk[9JMP Executed Successfully
ADD R4, R3, R1 ... in the Issuing Stage
JALR R4, R1 Fetched
R0 -- R1 -- R2 -- R3 -- R4 -- R5 -- R6 -- R7
     0    12    20-13108     0     0     0     0
CLK[10] ----------------------------------------------------
Instruction type (ADDI) has been added to its corresponding RS
Processing Instructions...
JMP 2 ... in the Writing Stage
Clk[10] JMP Writes its result Successfully
JMP 2: Fetch(4) / Issue(8) / Execute(9, 9) / Write(10)
ADD R4, R3, R1 Waiting in the Executing Stage
ADD R4, R3, R1 Waiting in the Executing Stage
Clk[10ADD Executed Successfully
ADDI R1, R0, 23 ... in the Issuing Stage
R0 -- R1 -- R2 -- R3 -- R4 -- R5 -- R6 -- R7
     0    12    20-13108     0     0     0     0
CLK[11] ----------------------------------------------------
Processing Instructions...
ADD R4, R3, R1 ... in the Writing Stage
Clk[11] ADD Writes its result Successfully
ADD R4, R3, R1: Fetch(3) / Issue(7) / Execute(9, 10) / Write(11)
ADD R4, R3, R1 Waiting in the Executing Stage
Clk[11ADD Executed Successfully
ADDI R1, R0, 23 Waiting in the Executing Stage
R0 -- R1 -- R2 -- R3 -- R4 -- R5 -- R6 -- R7
     0    12    20-13108-13096     0     0     0
CLK[12] ----------------------------------------------------
Instruction type (ADDI) has been added to its corresponding RS
Processing Instructions...
ADD R4, R3, R1 ... in the Writing Stage
Clk[12] ADD Writes its result Successfully
ADD R4, R3, R1: Fetch(5) / Issue(9) / Execute(10, 11) / Write(12)
ADDI R1, R0, 23 Waiting in the Executing Stage
Clk[12ADDI Executed Successfully
ADDI R3, R0, 12 ... in the Issuing Stage
R0 -- R1 -- R2 -- R3 -- R4 -- R5 -- R6 -- R7
     0    12    20-13108-13096     0     0     0
```

```
CLK[13] ----------------------------------------------------
Instruction type (ADDI) has been added to its corresponding RS
Processing Instructions...
ADDI R1, R0, 23 ... in the Writing Stage
Clk[13] ADDI Writes its result Successfully
ADDI R1, R0, 23: Fetch(6) / Issue(10) / Execute(11, 12) / Write(13)
ADDI R3, R0, 12 Waiting in the Executing Stage
ADDI R2, R0, 3 ... in the Issuing Stage
ADDI R6, R0, 30 Fetched
R0 -- R1 -- R2 -- R3 -- R4 -- R5 -- R6 -- R7
     0    23    20-13108-13096     0     0     0
CLK[14] ----------------------------------------------------
ADDI R5, R0, 3 Fetched
Instruction type (JALR R1 ) has been added to its corresponding RS
Processing Instructions...
ADDI R2, R0, 3 Waiting in the Executing Stage
ADDI R3, R0, 12 Waiting in the Executing Stage
Clk[14ADDI Executed Successfully
JALR R4, R1 ... in the Issuing Stage
ADDI R5, R5, 3 Fetched
R0 -- R1 -- R2 -- R3 -- R4 -- R5 -- R6 -- R7
     0    23    20-13108-13096     0     0     0
CLK[15] ----------------------------------------------------
BEQ R6, R5, 3 Fetched
Instruction type (ADDI) has been added to its corresponding RS
Processing Instructions...
ADDI R3, R0, 12 ... in the Writing Stage
Clk[15] ADDI Writes its result Successfully
ADDI R3, R0, 12: Fetch(7) / Issue(12) / Execute(13, 14) / Write(15)
ADDI R2, R0, 3 Waiting in the Executing Stage
Clk[15ADDI Executed Successfully
JALR R4, R1 Waiting in the Executing Stage
Clk[15JALR Executed Successfully
ADDI R6, R0, 30 ... in the Issuing Stage
ADD R0, R0, R0 Fetched
R0 -- R1 -- R2 -- R3 -- R4 -- R5 -- R6 -- R7
     0    23    20    12-13096     0     0     0
```

Nothing was fetched after the JALR except when the instruction using R1 was finished. It predicted all instruction based on the criteria mentioned in the project manual. In case any misprediction is found, the processor is flushed completely except for previous instructions. We also handled the case of a BEQ instruction or a JMP instruction before the BEQ so that no branch instruction can predict and fetch if there is a previous branch instruction in the instruction queue.