Project 2: Tomasulo

Nour Abdalla 900213024
Sarah Elsamanody 900212915

Computer Architecture
Dr. Cherif Salama

Table of Contents:

I.Introduction

The objective of this project is to construct one of the main concepts taken in our course, Instruction-Level-Parallelism, also known as the ability to execute multiple instructions simultaneously and in our case on a single-issue processor through Tomosulo's Algorithm. Therefore, this project implements a simulator using this algorithm which allows dynamic scheduling through out-of-order execution whilst taking into account any potential data dependencies and hazards. In the given program, it only supports 9 basic instructions of risc-v, which are load,store, call,return,add,addi,mul, and nand using python code. We implemented this program by adding different functions for each stage issue, execution, and write ; in addition, we added extra functions for other requirements which are needed in these stages, for the validation, and for combining everything together. The bonus chosen was to take advantage of the tables outputted by giving the user the option to track the changes step by step, however we did not implement a GUI and instead we made the output in the console in a user-friendly way as it is not only useful for anyone running the program,but also for us when debugging and making sure that everything's in place.

II.How to use the code

Here is how we take our inputs for the memory and instructions:

```
samanodyjr@Sarahs-MacBook-Pro-3 Assembly_project2 % /Users/samanodyjr/.pyenv/versions/3.8.9/b
in/python /Users/samanodyjr/Documents/badshit/Assembly_project2/tomasulo.py
--------------------Tomasulo's Algorithm ------------------
Please enter your starting adress: 0
Please enter a memory address you want to initialize or exit to cancel:
address: 0
data: 1
address: 1
data: 2
address: 2
data: 4
address: 3
data: 6
address: 4
data: 1
address: exit
```

```
address: exit
Please enter the instructions of your program and write exit when you are done:
insturction: load r5, 0(r0)
insturction: load r2, 1(r0)
insturction: load r4, 2(r0)
insturction: load r6, 3(r0)
insturction: load r1, 4(r0)
insturction: mul r0, r2, r4
insturction: mul r4, r2, r6
insturction: add r6, r4, r4
insturction: beq r2, r2, 2
insturction: addi r2, r2, 1
insturction: ret
insturction:  nand r6, r5, r1
insturction:  beq r6, r5, 3
insturction: store r1,2(r2)
insturction: beq r1, r2, 0
insturction: call 9
insturction: add r0 , r0 , r0
insturction: exit
```

Then after the exit we showcase everything step by step and you jump step by step by writing ‘y’ to preview it and lastly you write exit when you reach write of the last instruction to end the program and see your analysis. The following picture shows the results of our main test case and we explained how it works through the test case.

This is how everything gets displayed:

This shows how since we have only 2 reservation station for load they wait for one another

```
--------------------Tomasulo's Algorithm ------------------
3 PC
current clk cycle:  9

Tracing Table:
```

Instructions	Issue	Execution Start	Execution End	Write
load r5, 0 (r0)	1	2	7	8
load r2, 1 (r0)	2	3	8	9
load r4, 2 (r0)	8	9		
load r6, 3 (r0)	9			

Reservation Stations:

Name	Busy	Op	Vj	Vk	Qj	Qk	A
LOAD1	Y	load	0				2
LOAD2	Y	load	0				3
STORE	N						
BEQ	N						
CALL/RET	N						
ADD/ADDI 1	N						
ADD/ADDI 2	N						
ADD/ADDI 3	N						
ADD/ADDI 4	N						
NAND1	N						
NAND2	N						
MUL	N						

Reagister Status:

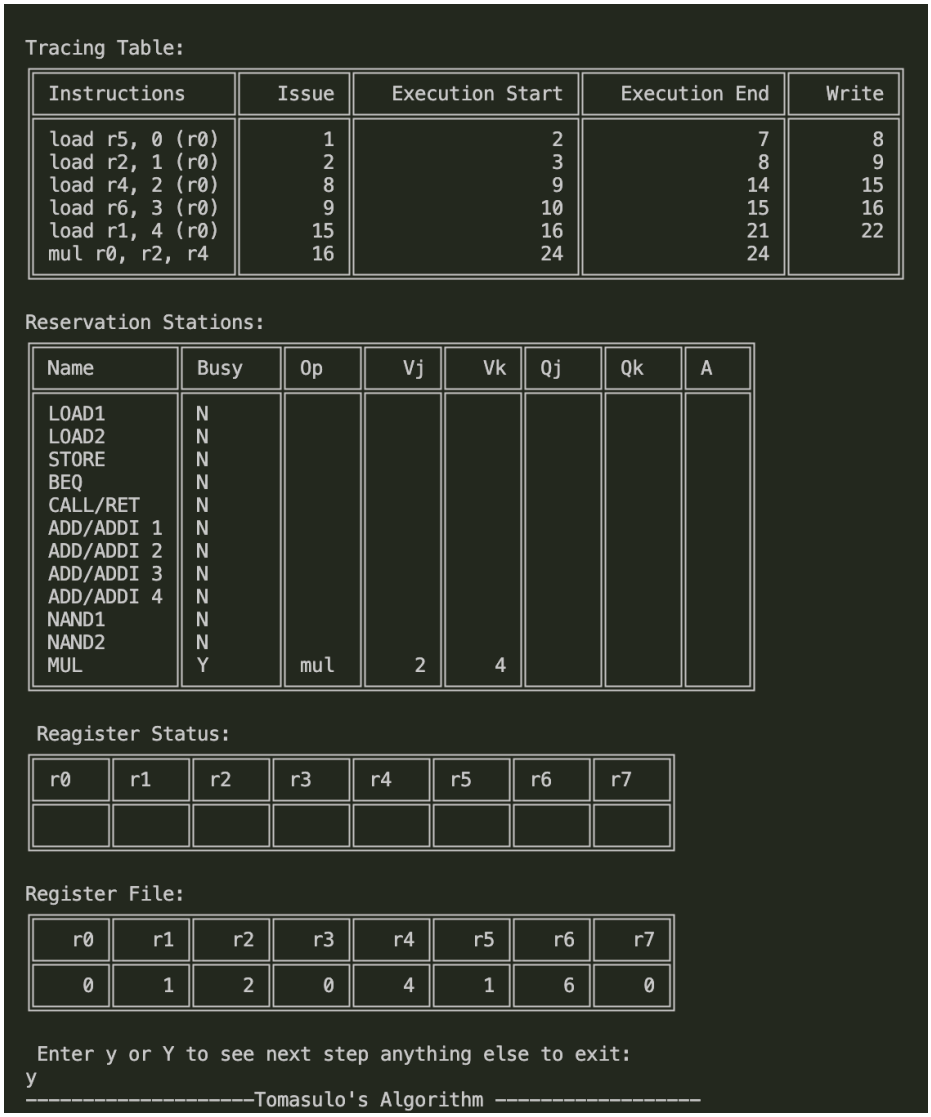
r0	r1	r2	r3	r4	r5	r6	r7
				LOAD1		LOAD2	

Register File:

r0	r1	r2	r3	r4	r5	r6	r7
0	0	2	0	0	1	0	0

```
Enter y or Y to see next step anything else to exit:
```

This also emphasizes on the idea that multiplication has only one reservation station which makes us not issue the next instruction since it is also a multiplication until it is done as issuing in tomasulo happens in order



Tracing Table:

Instructions	Issue	Execution Start	Execution End	Write
load r5, 0 (r0)	1	2	7	8
load r2, 1 (r0)	2	3	8	9
load r4, 2 (r0)	8	9	14	15
load r6, 3 (r0)	9	10	15	16
load r1, 4 (r0)	15	16	21	22
mul r0, r2, r4	16	24	24	

Reservation Stations:

Name	Busy	Op	Vj	Vk	Qj	Qk	A
LOAD1	N						
LOAD2	N						
STORE	N						
BEQ	N						
CALL/RET	N						
ADD/ADDI 1	N						
ADD/ADDI 2	N						
ADD/ADDI 3	N						
ADD/ADDI 4	N						
NAND1	N						
NAND2	N						
MUL	Y	mul	2	4			

Reagister Status:

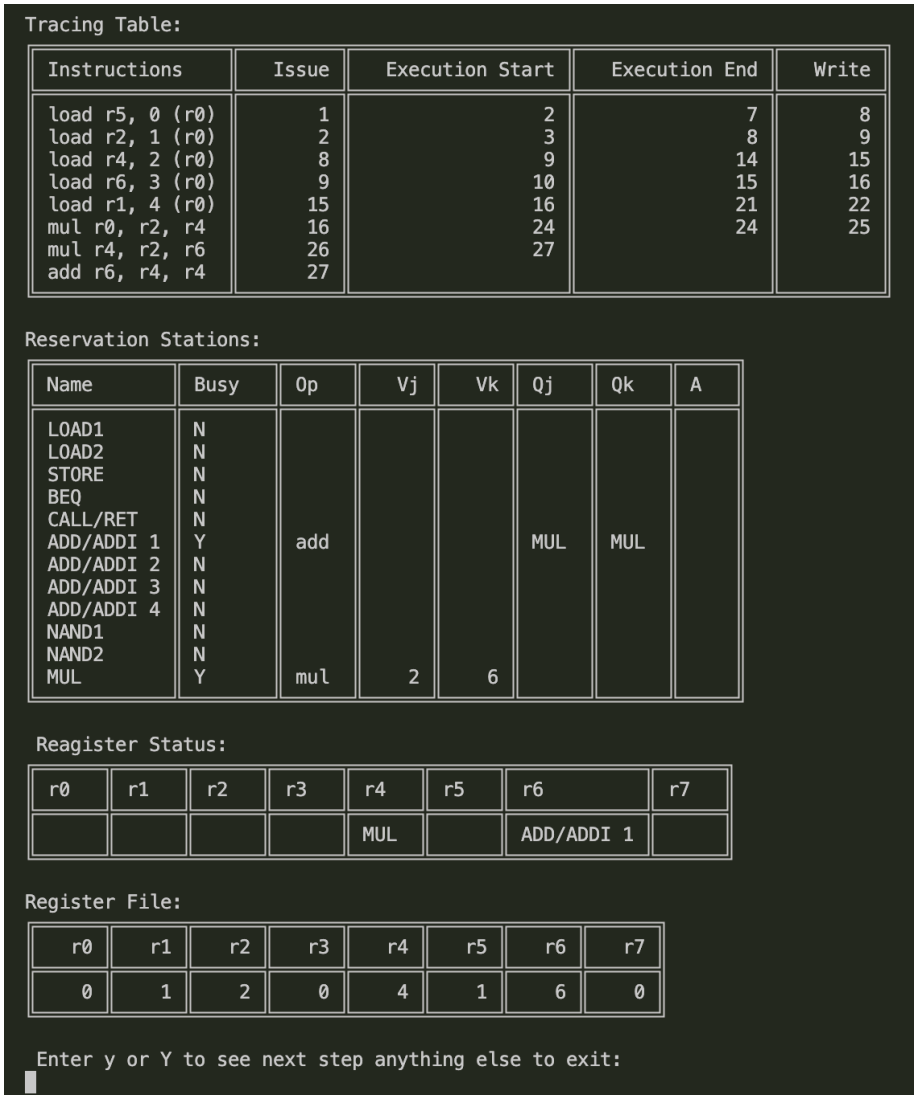
r0	r1	r2	r3	r4	r5	r6	r7

Register File:

r0	r1	r2	r3	r4	r5	r6	r7
0	1	2	0	4	1	6	0

Enter y or Y to see next step anything else to exit:
y
----------Tomasulo's Algorithm ----------

Here is the issuing of the multiplication afterwards and it execution lead to the issuing of add afterwards normally since there are free RS to allow add issuing, We can also see that there is a RAW dependency that is being handled by using Qj and Qk as an indication that add instruction awaits the values of the register in use. Also observe the changes in the register status:



Tracing Table:

Instructions	Issue	Execution Start	Execution End	Write
load r5, 0 (r0)	1	2	7	8
load r2, 1 (r0)	2	3	8	9
load r4, 2 (r0)	8	9	14	15
load r6, 3 (r0)	9	10	15	16
load r1, 4 (r0)	15	16	21	22
mul r0, r2, r4	16	24	24	25
mul r4, r2, r6	26	27		
add r6, r4, r4	27			

Reservation Stations:

Name	Busy	Op	Vj	Vk	Qj	Qk	A
LOAD1	N						
LOAD2	N						
STORE	N						
BEQ	N						
CALL/RET	N						
ADD/ADDI 1	Y	add			MUL	MUL	
ADD/ADDI 2	N						
ADD/ADDI 3	N						
ADD/ADDI 4	N						
NAND1	N						
NAND2	N						
MUL	Y	mul	2	6			

Reagister Status:

r0	r1	r2	r3	r4	r5	r6	r7
				MUL		ADD/ADDI 1	

Register File:

r0	r1	r2	r3	r4	r5	r6	r7
0	1	2	0	4	1	6	0

Enter y or Y to see next step anything else to exit:

Here we can notice our first branch instruction, it basically allows issuing to occur since we are basing everything on branch not taken prediction and in our code if the branch doesn't take 1 cycle to execute and we made it take 2 for instance we will notice that it does not allow execution unless we write the branch instruction as by then we will know if we need to flush the issued instructions or not.

Tracing Table:

Instructions	Issue	Execution Start	Execution End	Write
load r5, 0 (r0)	1	2	7	8
load r2, 1 (r0)	2	3	8	9
load r4, 2 (r0)	8	9	14	15
load r6, 3 (r0)	9	10	15	16
load r1, 4 (r0)	15	16	21	22
mul r0, r2, r4	16	24	24	25
mul r4, r2, r6	26	27		
add r6, r4, r4	27			
beq r2, r2, 2	28	29	29	
addi r2, r2, 1	29			

Reservation Stations:

Name	Busy	Op	Vj	Vk	Qj	Qk	A
LOAD1	N						
LOAD2	N						
STORE	N						
BEQ	Y	beq	2	2			11
CALL/RET	N						
ADD/ADDI 1	Y	add			MUL	MUL	
ADD/ADDI 2	Y	addi	2				
ADD/ADDI 3	N						
ADD/ADDI 4	N						
NAND1	N						
NAND2	N						
MUL	Y	mul	2	6			

Reagister Status:

r0	r1	r2	r3	r4	r5	r6	r7
		ADD/ADDI 2		MUL		ADD/ADDI 1	

Register File:

r0	r1	r2	r3	r4	r5	r6	r7
0	1	2	0	4	1	6	0

We can notice that the instruction got flushed during the write indicating that branch was taken:

Instructions	Issue	Execution Start	Execution End	Write
load r5, 0 (r0)	1	2	7	8
load r2, 1 (r0)	2	3	8	9
load r4, 2 (r0)	8	9	14	15
load r6, 3 (r0)	9	10	15	16
load r1, 4 (r0)	15	16	21	22
mul r0, r2, r4	16	24	24	25
mul r4, r2, r6	26	27		
add r6, r4, r4	27			
beq r2, r2, 2	28	29	29	30

Reservation Stations:

Name	Busy	Op	Vj	Vk	Qj	Qk	A
LOAD1	N						
LOAD2	N						
STORE	N						
BEQ	N						
CALL/RET	N						
ADD/ADDI 1	Y	add			MUL	MUL	
ADD/ADDI 2	N						
ADD/ADDI 3	N						
ADD/ADDI 4	N						
NAND1	N						
NAND2	N						
MUL	Y	mul	2	6			

Reagister Status:

r0	r1	r2	r3	r4	r5	r6	r7
				MUL		ADD/ADDI 1	

Register File:

r0	r1	r2	r3	r4	r5	r6	r7
0	1	2	0	4	1	6	0

Enter y or Y to see next step anything else to exit:

We can here see how it jumped to the target PC of the beq instruction and no executing NAND and we are still waiting got MUL to be finished since it takes 8 cycles to finish:

Instructions	Issue	Execution Start	Execution End	Write
load r5, 0 (r0)	1	2	7	8
load r2, 1 (r0)	2	3	8	9
load r4, 2 (r0)	8	9	14	15
load r6, 3 (r0)	9	10	15	16
load r1, 4 (r0)	15	16	21	22
mul r0, r2, r4	16	24	24	25
mul r4, r2, r6	26	27		
add r6, r4, r4	27			
beq r2, r2, 2	28	29	29	30
nand r6, r5, r1	31			

Reservation Stations:

Name	Busy	Op	Vj	Vk	Qj	Qk	A
LOAD1	N						
LOAD2	N						
STORE	N						
BEQ	N						
CALL/RET	N						
ADD/ADDI 1	Y	add			MUL	MUL	
ADD/ADDI 2	N						
ADD/ADDI 3	N						
ADD/ADDI 4	N						
NAND1	Y	nand	1	1			
NAND2	N						
MUL	Y	mul	2	6			

Reagister Status:

r0	r1	r2	r3	r4	r5	r6	r7
				MUL		NAND1	

Register File:

r0	r1	r2	r3	r4	r5	r6	r7
0	1	2	0	4	1	6	0

Enter y or Y to see next step anything else to exit:

The instruction after the NAND is a BEQ instruction that is waiting for NAND to return a value:

Tracing Table:

Instructions	Issue	Execution Start	Execution End	Write
load r5, 0 (r0)	1	2	7	8
load r2, 1 (r0)	2	3	8	9
load r4, 2 (r0)	8	9	14	15
load r6, 3 (r0)	9	10	15	16
load r1, 4 (r0)	15	16	21	22
mul r0, r2, r4	16	24	24	25
mul r4, r2, r6	26	27		
add r6, r4, r4	27			
beq r2, r2, 2	28	29	29	30
nand r6, r5, r1	31	32	32	
beq r6, r5, 3	32			

Reservation Stations:

Name	Busy	Op	Vj	Vk	Qj	Qk	A
LOAD1	N						
LOAD2	N						
STORE	N						
BEQ	Y	beq		1	NAND1		3
CALL/RET	N						
ADD/ADDI 1	Y	add			MUL	MUL	
ADD/ADDI 2	N						
ADD/ADDI 3	N						
ADD/ADDI 4	N						
NAND1	Y	nand	1	1			
NAND2	N						
MUL	Y	mul	2	6			

Reagister Status:

r0	r1	r2	r3	r4	r5	r6	r7
				MUL		NAND1	

Register File:

r0	r1	r2	r3	r4	r5	r6	r7
0	1	2	0	4	1	6	0

Here we can see that MUL instruction is finally done with execution and so as the NAND as it only takes one cycle which then we can see how values got immediately reflected in reservation station. The reason behind why the issuing stopped is that the instruction following this is another BEQ that can't be issued as not RS is free to handle it.

Instructions	Issue	Execution Start	Execution End	Write
load r5, 0 (r0)	1	2	7	8
load r2, 1 (r0)	2	3	8	9
load r4, 2 (r0)	8	9	14	15
load r6, 3 (r0)	9	10	15	16
load r1, 4 (r0)	15	16	21	22
mul r0, r2, r4	16	24	24	25
mul r4, r2, r6	26	34	34	35
add r6, r4, r4	27			
beq r2, r2, 2	28	29	29	30
nand r6, r5, r1	31	32	32	33
beq r6, r5, 3	32	34	34	
store r1, 2, r2	33			

Reservation Stations:

Name	Busy	Op	Vj	Vk	Qj	Qk	A
LOAD1	N						
LOAD2	N						
STORE	Y	store	1	2			2
BEQ	Y	beq	-2	1			3
CALL/RET	N						
ADD/ADDI 1	Y	add	12	12			
ADD/ADDI 2	N						
ADD/ADDI 3	N						
ADD/ADDI 4	N						
NAND1	N						
NAND2	N						
MUL	N						

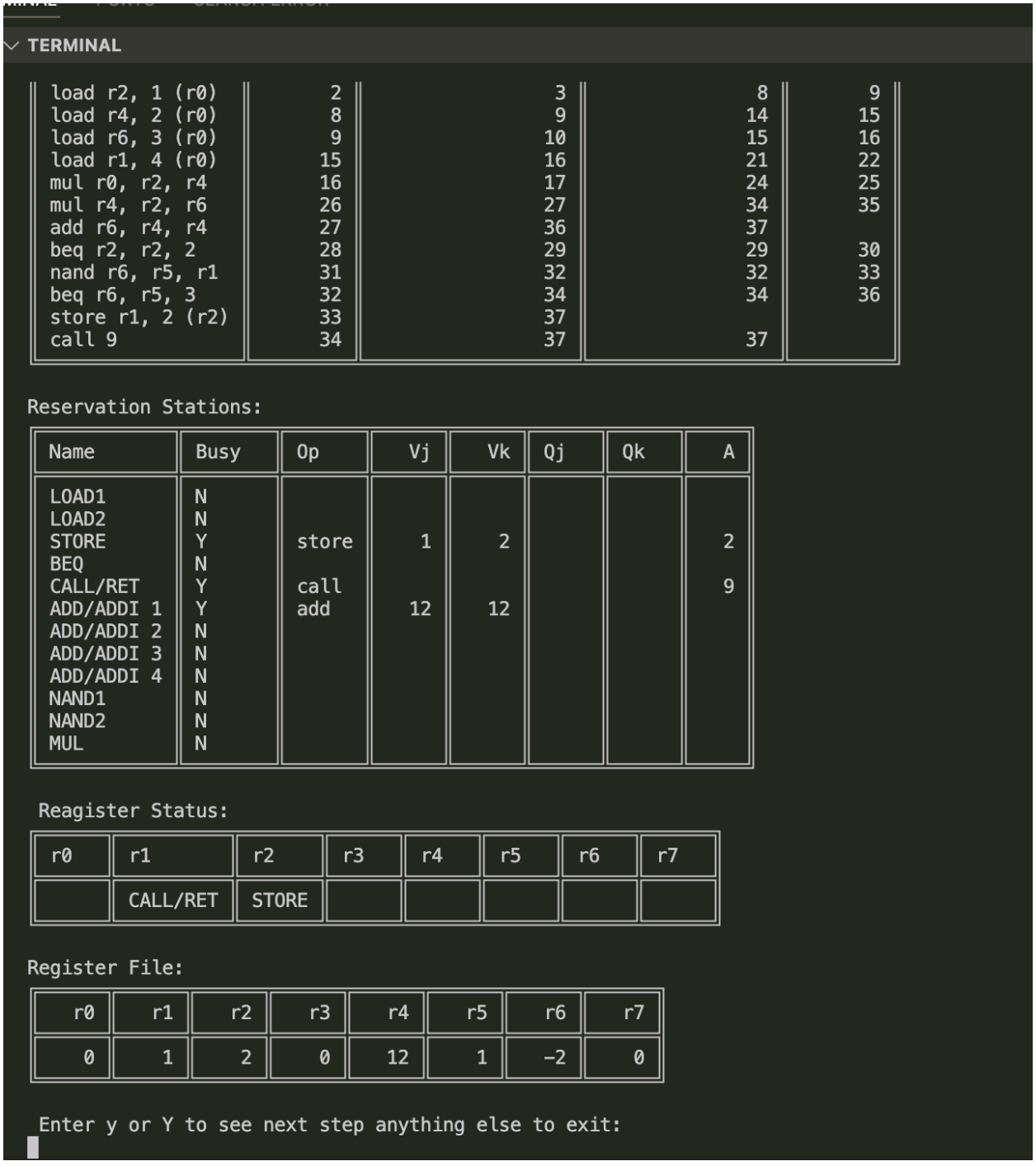
Reagister Status:

r0	r1	r2	r3	r4	r5	r6	r7

Register File:

r0	r1	r2	r3	r4	r5	r6	r7
0	1	2	0	12	1	-2	0

Here is the following instruction after BEQ, STORE to see a CALL instruction and we can see that we stalled issuing for multiple cycles til the CALL is written:



TERMINAL

load r2, 1 (r0)	2	3	8	9
load r4, 2 (r0)	8	9	14	15
load r6, 3 (r0)	9	10	15	16
load r1, 4 (r0)	15	16	21	22
mul r0, r2, r4	16	17	24	25
mul r4, r2, r6	26	27	34	35
add r6, r4, r4	27	36	37	
beq r2, r2, 2	28	29	29	30
nand r6, r5, r1	31	32	32	33
beq r6, r5, 3	32	34	34	36
store r1, 2 (r2)	33	37		
call 9	34	37	37	

Reservation Stations:

Name	Busy	Op	Vj	Vk	Qj	Qk	A
LOAD1	N						
LOAD2	N						
STORE	Y	store	1	2			2
BEQ	N						
CALL/RET	Y	call					9
ADD/ADDI 1	Y	add	12	12			
ADD/ADDI 2	N						
ADD/ADDI 3	N						
ADD/ADDI 4	N						
NAND1	N						
NAND2	N						
MUL	N						

Reagister Status:

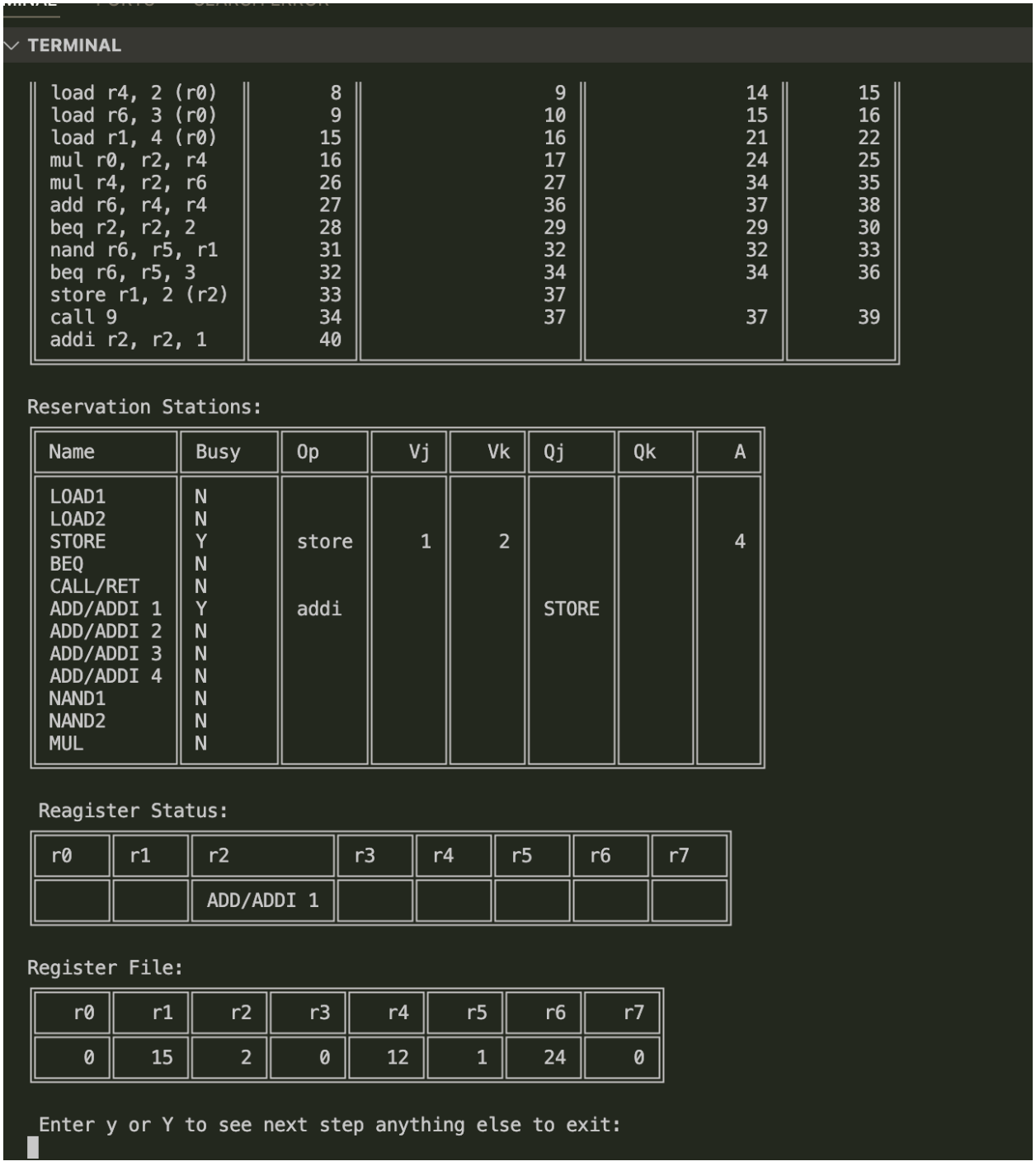
r0	r1	r2	r3	r4	r5	r6	r7
	CALL/RET	STORE					

Register File:

r0	r1	r2	r3	r4	r5	r6	r7
0	1	2	0	12	1	-2	0

Enter y or Y to see next step anything else to exit:

Here we can see it goes back to ADDI in pc 9 after writing the call where it can go back to issuing:



Here we can see that ret function got issued and stalled issuing until after it got written to avoid hazards :

Tracing Table:

Instructions	Issue	Execution Start	Execution End	Write
load r5, 0 (r0)	1	2	7	8
load r2, 1 (r0)	2	3	8	9
load r4, 2 (r0)	8	9	14	15
load r6, 3 (r0)	9	10	15	16
load r1, 4 (r0)	15	16	21	22
mul r0, r2, r4	16	17	24	25
mul r4, r2, r6	26	27	34	35
add r6, r4, r4	27	36	37	38
beq r2, r2, 2	28	29	29	30
nand r6, r5, r1	31	32	32	33
beq r6, r5, 3	32	34	34	36
store r1, 2 (r2)	33	37	42	43
call 9	34	37	37	39
addi r2, r2, 1	40	44	45	
ret	41	42	42	44
add r0, r0, r0	45			

Reservation Stations:

Name	Busy	Op	Vj	Vk	Qj	Qk	A
LOAD1	N						
LOAD2	N						
STORE	N						
BEQ	N						
CALL/RET	N						
ADD/ADDI 1	Y	addi	24				
ADD/ADDI 2	Y	add	0	0			
ADD/ADDI 3	N						
ADD/ADDI 4	N						
NAND1	N						
NAND2	N						
MUL	N						

Reagister Status:

r0	r1	r2	r3	r4	r5	r6	r7

Lastly, the final instruction run and we have to write exit in order to quit the program as it has to be done manually and then we get results and analysis accordingly:

store r1, 2 (r2)	33	37	42	43
call 9	34	37	37	39
addi r2, r2, 1	40	44	45	46
ret	41	42	42	44
add r0, r0, r0	45	46	47	48

Reservation Stations:

Name	Busy	Op	Vj	Vk	Qj	Qk	A
LOAD1	N						
LOAD2	N						
STORE	N						
BEQ	N						
CALL/RET	N						
ADD/ADDI 1	N						
ADD/ADDI 2	N						
ADD/ADDI 3	N						
ADD/ADDI 4	N						
NAND1	N						
NAND2	N						
MUL	N						

Reagister Status:

r0	r1	r2	r3	r4	r5	r6	r7

Register File:

r0	r1	r2	r3	r4	r5	r6	r7
0	15	25	0	12	1	24	0

Enter y or Y to see next step anything else to exit:
exit
----------Additional Information----------

Branch Misprediction: 50.0 %

total execution time: 48

instruction per cycles: 0.3333333333333333

III. Test Cases

1.*test case 1 code*:

```
load r5, 0(r0)
load r2, 1(r0)
load r4, 2(r0)
load r6, 3(r0)
load r1, 4(r0)
mul r0, r2, r4
mul r4, r2, r6
add r6, r4, r4
beq r2, r2, 2
addi r2, r2, 1
ret
nand r6, r5, r1
beq r6, r5, 3
store r1,2(r2)
mul r2, r4, r7
call 29
add r0 , r0 , r0
```

2.*test case 2 code*:

```
addi r5,r0,5
addi r2,r0,2
store r2, 0(r0)
load r3, 0(r0)
mul r5,r2,r3
beq r5, r0, 3
addi r5,r5,-2
nand r6,r5,r5
beq r0,r0,-4
add r7,r5,r0
mul r5,r2,r3
addi r1,r0,0
```

3. ***test case 3 code***:

Initialize memory address 0 with 4 and memory address 1 with 6

```
addi r6,r0,-1
addi r7,r0,1 1
load r6, 0(r0)
load r7, 1(r0)
store r3, 0(r0)
store r2, 1(r0)
call 28
mul r4, r0,r2
add r2,r3,r3
mul r5,r2,r3
```

IV. Commentary on the results

Test case 1:

Test case 1 was used to explain how our program works and we explained it thoroughly up above. It is the main test case that runs all instructions our code provides and showcases the most common and majority of dependencies encountered.

Test case 2:

This test case's main purpose is to provide us with a loop to test how the code properly works. We enter the loop to decrement a register value until it is zero to exit and we calculate nand of this register with itself within the body of a loop. This program was made to test how efficiently the BEQ instructions work and goes back and forth within our logic. It also showcases the store and how it actually works as we load after storing to see that it is really reflected.

```
ck_cycles, PC):
```

PORTS SEARCH ERROR

TERMINAL

current clk cycle: 55

Tracing Table:

Instructions	Issue	Execution Start	Execution End	Write
addi r5, r0, 5	1	2	3	4
addi r2, r0, 2	2	3	4	5
store r2, 0 (r0)	3	6	11	12
load r3, 0 (r0)	4	13	18	19
mul r5, r2, r3	5	20	27	28
beq r5, r0, 3	6	29	29	30
addi r5, r5, -2	7	31	32	33
nand r6, r5, r5	8	34	34	35
beq r0, r0, -4	31	32	32	34
beq r5, r0, 3	35	36	36	37
addi r5, r5, -2	36	38	39	40
nand r6, r5, r5	37	41	41	42
beq r0, r0, -4	38	39	39	41
beq r5, r0, 3	42	43	43	44
add r7, r5, r0	45	46	47	48
mul r5, r2, r3	46	47	54	55
addi r1, r0, 0	47	48	49	50

Reservation Stations:

Name	Busy	Op	Vj	Vk	Qj	Qk	A
LOAD1	N						
LOAD2	N						
STORE	N						
BEQ	N						
CALL/RET	N						
ADD/ADDI 1	N						
ADD/ADDI 2	N						
ADD/ADDI 3	N						
ADD/ADDI 4	N						
NAND1	N						
NAND2	N						
MUL	N						

Reagister Status:

r0	r1	r2	r3	r4	r5	r6	r7

Register File:

r0	r1	r2	r3	r4	r5	r6	r7
0	0	2	2	0	4	-1	0

Enter y or Y to see next step anything else to exit:
exit
----------Additional Information----------

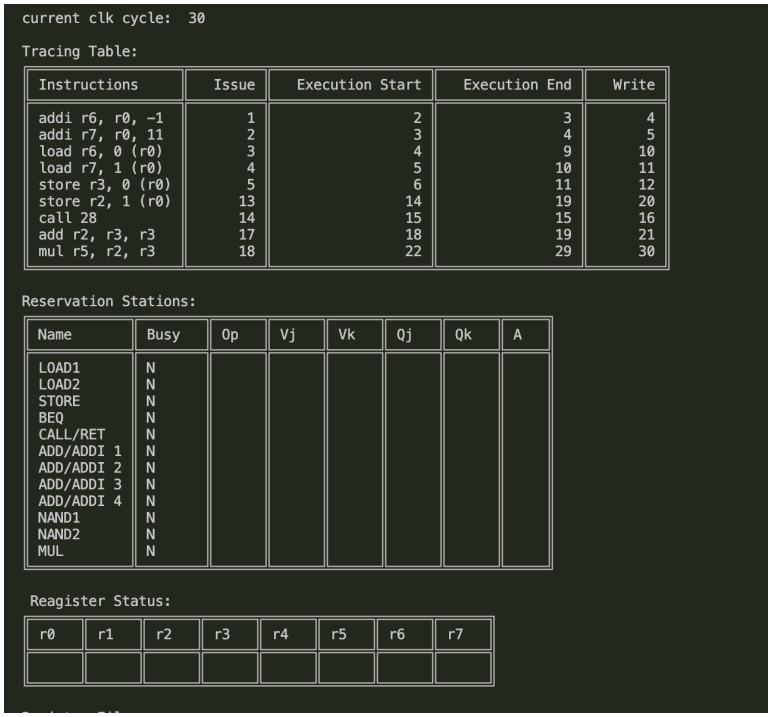
Branch Misprediction: 60.0 %

total execution time: 55

instruction per cycles: 0.3090909090909091

Test case 3:

This test case is to add extra testing to our program and we here tested the call as well as loading and storing .



current clk cycle: 30

Tracing Table:

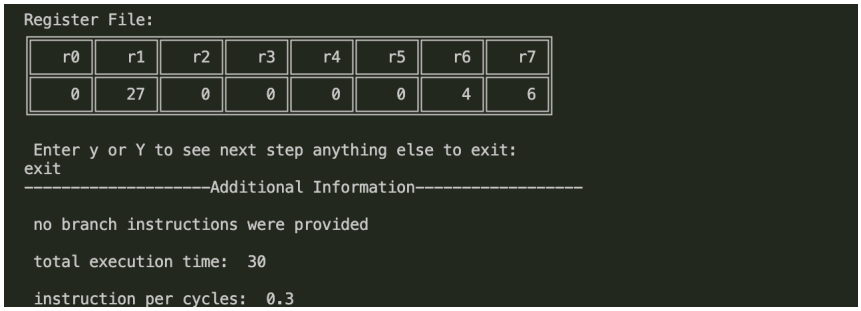
Instructions	Issue	Execution Start	Execution End	Write
addi r6, r0, -1	1	2	3	4
addi r7, r0, 11	2	3	4	5
load r6, 0 (r0)	3	4	9	10
load r7, 1 (r0)	4	5	10	11
store r3, 0 (r0)	5	6	11	12
store r2, 1 (r0)	13	14	19	20
call 28	14	15	15	16
add r2, r3, r3	17	18	19	21
mul r5, r2, r3	18	22	29	30

Reservation Stations:

Name	Busy	Op	Vj	Vk	Qj	Qk	A
LOAD1	N						
LOAD2	N						
STORE	N						
BEQ	N						
CALL/RET	N						
ADD/ADDI 1	N						
ADD/ADDI 2	N						
ADD/ADDI 3	N						
ADD/ADDI 4	N						
NAND1	N						
NAND2	N						
MUL	N						

Reagister Status:

r0	r1	r2	r3	r4	r5	r6	r7



Register File:

r0	r1	r2	r3	r4	r5	r6	r7
0	27	0	0	0	0	4	6

Enter y or Y to see next step anything else to exit:
exit
----------Additional Information----------

no branch instructions were provided

total execution time: 30

instruction per cycles: 0.3

IV. How did we tackle this problem

When implementing this program, we put many things into consideration as well as trying as much as possible to always divide our problem into subproblems when writing our code, making it easier to construct. Firstly, we wanted to make our code similar to how we normally traced it in the lecture, cycle by cycle. This helped us significantly as we understood what gets issued/executed/written step by step which led us to identifying the data dependencies and hazards which may occur and therefore brainstorming together how to handle them accordingly. As a result, we were very cautious in how we order the execution and write in our functions and where we call each at the main function(simulator) along with the right conditions to be checked in order for the program to run effectively and correctly. Another problem that we took into account is the common bus, in which we cannot write at the same time and so we solved this problem by adding a write queue and taking them one by one and then pop according to the order. For the display, we wanted to follow the same format we took in the course as it is very comprehensive and shows everything needed and output from the program and the register file ; however we don't show how the memory changes, but the second test case shown above is highlighting and validating that the memory is changing correctly. On the other hand, we faced some problems when implementing the call and return in which we had to stall for the issuing as well as the branch where we stalled in the execution. Overall whenever we got stuck we always referred back to tracing the problem in the code as we did by hand and in the lecture and even doing the bonus helped us more in debugging as we displayed in each cycle after each instruction what gets changed and what doesn't.