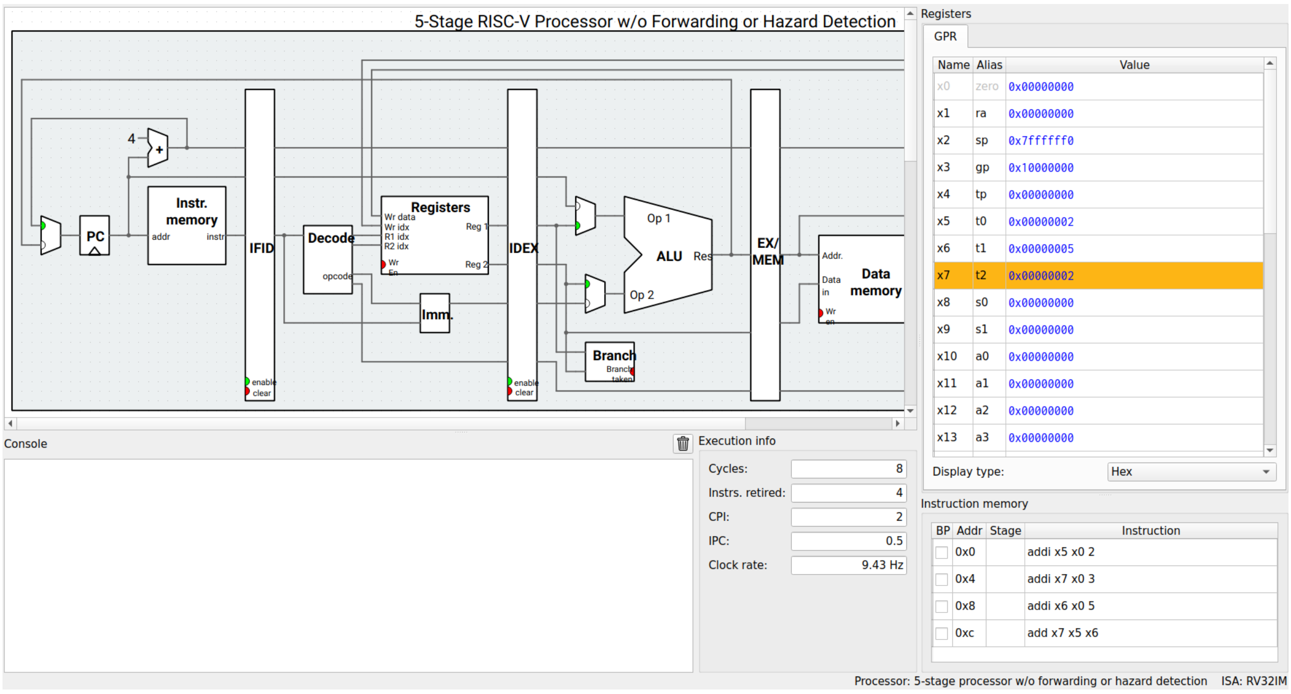
CS232 - Lab 4 Report

Name - Atishay Jain
Roll Number - 210050026

10th March 2023

1 Question 1

1.1 Part (a) Read After Write (RAW) hazard



Registers — GPR

Name	Alias	Value
x0	zero	0x00000000
x1	ra	0x00000000
x2	sp	0x7ffffff0
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000002
x6	t1	0x00000005
x7	t2	0x00000002
x8	s0	0x00000000
x9	s1	0x00000000
x10	a0	0x00000000
x11	a1	0x00000000
x12	a2	0x00000000
x13	a3	0x00000000

Display type: Hex

Execution info

Cycles:	8
Instrs. retired:	4
CPI:	2
IPC:	0.5
Clock rate:	9.43 Hz

Instruction memory

BP	Addr	Stage	Instruction
☐	0x0		addi x5 x0 2
☐	0x4		addi x7 x0 3
☐	0x8		addi x6 x0 5
☐	0xc		add x7 x5 x6

Processor: 5-stage processor w/o forwarding or hazard detection ISA: RV32IM

Figure 1: Screenshot after running my code demonstrating RAW hazard

The code which was used here is :

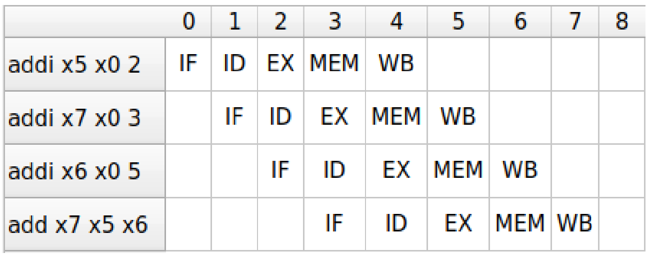
```
li t0, 2            # load 2 into t0
li t2, 3            # load 3 into t2
li t1, 5            # load 5 into t1
add t2, t0, t1      # add t0 and t1 and store result in t2
```

The expected output from this code was meant to be

$$\texttt{t0} = 2, \texttt{t1} = 5, \texttt{t2} = 7$$

However, due to RAW hazard, the output at register `t2` was affected. As seen from the above image of the register values after running the code, the output I got is `t0` = 2, `t1` = 5, `t2` = **2**

The processor used here is a 5 stage processor without forwarding or hazard detection, and each instruction takes 5 stages. We know that for a load instruction, the value is written to memory at `WB` stage, which is the last stage of an instruction. The stage at which values are read from memory is the `ID` stage when decoding the value stored from the memory occurs.



	0	1	2	3	4	5	6	7	8
addi x5 x0 2	IF	ID	EX	MEM	WB				
addi x7 x0 3		IF	ID	EX	MEM	WB			
addi x6 x0 5			IF	ID	EX	MEM	WB		
add x7 x5 x6				IF	ID	EX	MEM	WB	

Figure 2: Pipeline diagram of q1-a

So, as seen from the above pipeline diagram of the code -

- The first 3 lines are writing the values into some registers. This writing operation gets completed at the last stage of the 5 stages of an instruction.
- At cycle number 4, where the add instruction ($t2 = t0 + t1$) is at its decode stage (`ID`), the reading of data from the registers would be happening.
- The first instruction's write back (`WB`) is also at cycle 4, and infact would be done in first half of cycle. As the reading part of 4th instruction would be in later half of this cycle, it correctly gets the value stored in `t0` as 2
- The second instruction writes value of 3 into `t2` by cycle 5. So after cycle 5, the value in `t2` is 3. But, this value of t2 does not affect the final outcome of `t2` as it is overwritten by the last instruction's outcome which writes back its value to `t2` at cycle 7. Also, the `ID` stage of last instruction is not affected by 2nd instruction as it does not need to read value of `t2`. It just needs to read `t0`, `t1` and write its output to `t2`
- The third instruction's `WB` happens at cycle 6. Therefore, it writes the value of 5 into `t1` by cycle 6. But, the reading from the `t1` for last add instruction has already been done at cycle 4. This is were the RAW hazard happens. We expected the 5 to be the value present in `t1` during the last add instruction, but due to RAW hazard, the value stored in `t1` during the `ID` stage of last instruction was 0 and it reads it instead of 5.
- Therefore, when the addition executes, it calculates its output by taking inputs as `t0` = 2 and `t1` = 0, although we wanted them to be `t0` = 2 and `t1` = 5. And the result as $2 + 0 = 2$ gets stored in `t2` at cycle 7. This is why we got the value 2 (i.e. $2 + 0$) at the end of program instead of 7 (i.e. $2 + 5$) due to RAW hazard

1.2 Part (b) Write After Read (WAR) hazard

This hazard is **not possible** in the 5 stage processor without forwarding or hazard detection. I am submitting empty file for it. This is because of the structure of pipeline as we read at decode stage (`ID`) and write at write back stage (`WB`). The `WB` stage is far after the `ID` stage. Therefore, even if next instruction writes to a register which it read in previous instruction, it

will write correctly and the subsequent instructions will also use the correct value.
For example, consider

```
add t1 t0 t2
li t0 5
```

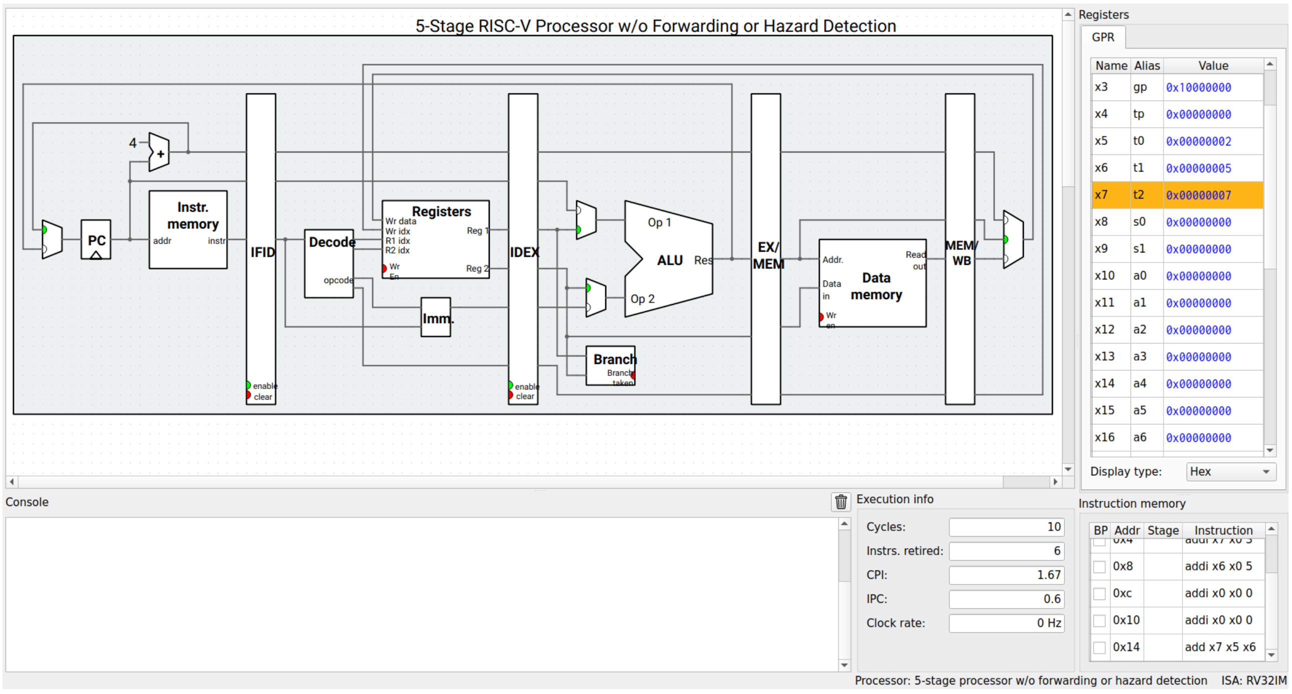
Here, in pipeline diagram, the read of 1st instruction would be completed before the write of 2nd one, because read (2ns stage) occurs before the write back stage (5th stage) of a instruction. That is the read of 1st would be done with fetch stage of 2nd and therefore will not cause WAR hazard.

2 Question 2

2.1 Modified Q1(a) with `nop` instructions to eliminate all the hazards

```
li t0, 2
li t2, 3
li t1, 5
nop
nop
add t2, t0, t1
```

I have used 2 `nop` instructions here for removing RAW hazard. As seen from q1-a pipeline in Figure 2, the hazard was happening because `WB` of 2nd last instruction was after `ID` of last instruction. A `nop` instruction is a instruction which does not affects the current program in any way, or in other words, it does nothing.



Registers

GPR

Name	Alias	Value
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000002
x6	t1	0x00000005
x7	t2	0x00000007
x8	s0	0x00000000
x9	s1	0x00000000
x10	a0	0x00000000
x11	a1	0x00000000
x12	a2	0x00000000
x13	a3	0x00000000
x14	a4	0x00000000
x15	a5	0x00000000
x16	a6	0x00000000

Display type: Hex

Execution info

Cycles:	10
Instrs. retired:	6
CPI:	1.67
IPC:	0.6
Clock rate:	0 Hz

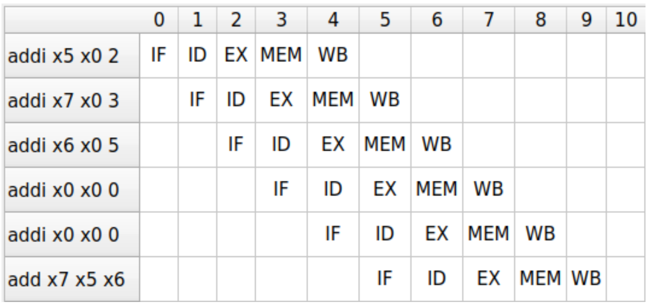
Instruction memory

BP	Addr	Stage	Instruction
	0x4		addi x7 x0 3
	0x8		addi x6 x0 5
	0xc		addi x0 x0 0
	0x10		addi x0 x0 0
	0x14		add x7 x5 x6

Processor: 5-stage processor w/o forwarding or hazard detection ISA: RV32IM

Figure 3: Modified Q1-a part with nop instructions

There can be many examples of nop, the one used here when writing `nop` was adding 0 to some other register, which had no effect on the current program. Adding nop at 4th and 5th line adds 2 extra instructions. Due to them, the pipeline diagram changed as - So, now the `ID` of



	0	1	2	3	4	5	6	7	8	9	10
addi x5 x0 2	IF	ID	EX	MEM	WB						
addi x7 x0 3		IF	ID	EX	MEM	WB					
addi x6 x0 5			IF	ID	EX	MEM	WB				
addi x0 x0 0				IF	ID	EX	MEM	WB			
addi x0 x0 0					IF	ID	EX	MEM	WB		
add x7 x5 x6						IF	ID	EX	MEM	WB	

Figure 4: Modified pipeline diagram after adding `nop`

last add instruction is at the same cycle (cycle number 6) as `WB` of the 3rd instruction in which the value is written in `t1`. And as I have told that if they are in same cycle, the read gets the correct value after being written as it is in 2nd half, the hazard does not occur. So, last instruction gets correct values of t1 (= 5) and t0 (= 2) and stores the correct expected output (t2 = 7) as shown in Figure 3.

2.2 Modified Q1(b) with `nop` instructions to eliminate all the hazards

As the WAR hazard itself was not possible in the 5 staged pipeline without forwarding or hazard detection, the q1-b part was empty and so is this. Therefore, I am submitting an empty file for q2-b

3 Question 3

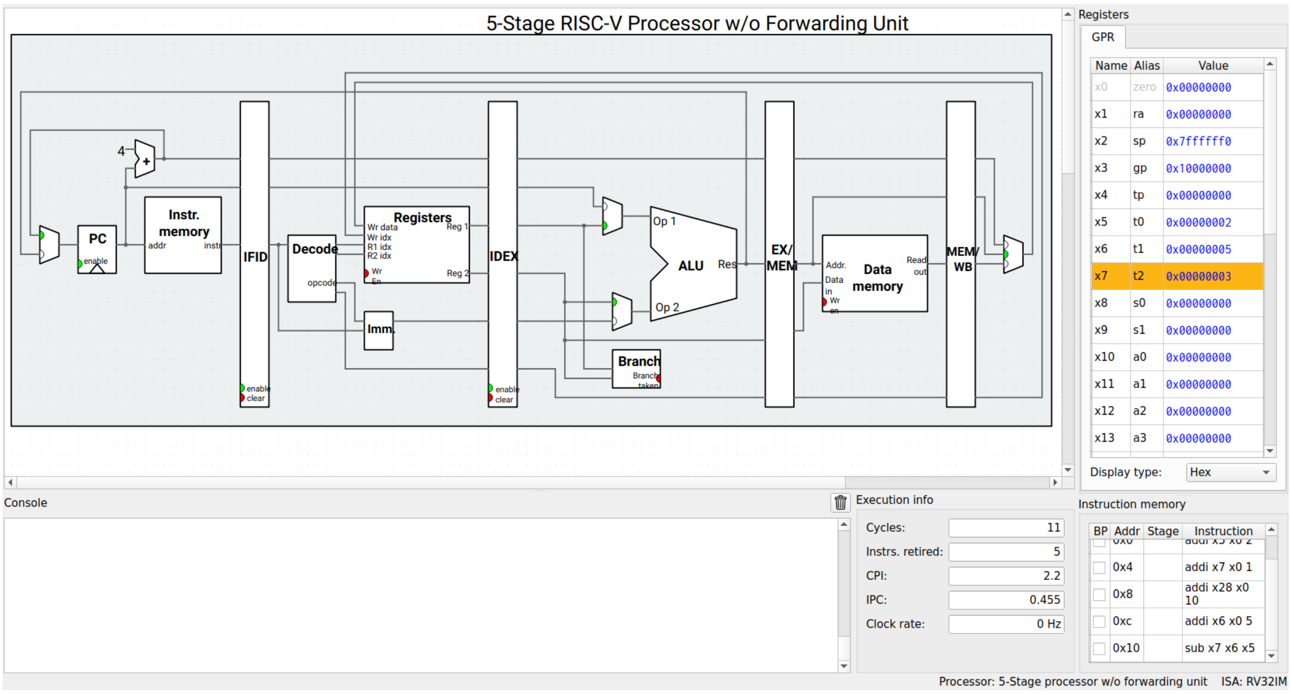
3.1 Part (a) Cycles in 5 stage processor vs without forwarding

Below is an instruction set which, when run on a 5 stage processor without forwarding, uses higher number of cycles than when run on a 5 stage processor. This code just loads some values into some registers and then performs a subtraction instruction.

```
li t0 2
li t2 1
li t3 10
li t1 5
sub t2 t1 t0
```

3.1.1 Run on a 5 stage processor without forwarding

The number of cycles used is this case are **11**. This is because due to no forwarding, 2 stalls are inserted at the last instruction to prevent hazard. The code I have written is such that there is a possibility of RAW hazard at last instruction and as there is hazard detection in the processor and no option of forwarding, it inserted two empty stalls at cycle number 6 and 7 of at last instruction. Therefore 11 cycles are used here.



Name	Alias	Value
x0	zero	0x00000000
x1	ra	0x00000000
x2	sp	0x7ffffff0
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000002
x6	t1	0x00000005
x7	t2	0x00000003
x8	s0	0x00000000
x9	s1	0x00000000
x10	a0	0x00000000
x11	a1	0x00000000
x12	a2	0x00000000
x13	a3	0x00000000

Execution info

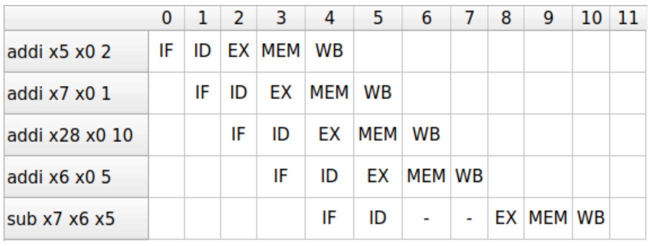
Cycles:	11
Instrs. retired:	5
CPI:	2.2
IPC:	0.455
Clock rate:	0 Hz

Instruction memory

BP	Addr	Stage	Instruction
	0x0		addi x5 x0 2
	0x4		addi x7 x0 1
	0x8		addi x28 x0 10
	0xc		addi x6 x0 5
	0x10		sub x7 x6 x5

Processor: 5-Stage processor w/o forwarding unit ISA: RV32IM

Figure 5: Screenshot after running on 5 stage processor without forwarding

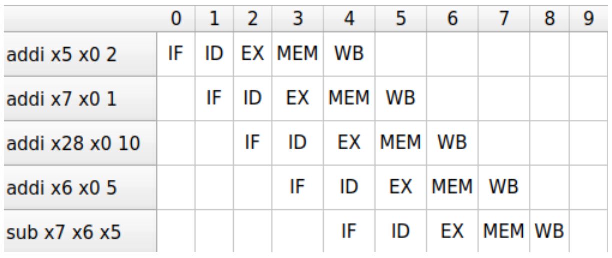


	0	1	2	3	4	5	6	7	8	9	10	11
addi x5 x0 2	IF	ID	EX	MEM	WB							
addi x7 x0 1		IF	ID	EX	MEM	WB						
addi x28 x0 10			IF	ID	EX	MEM	WB					
addi x6 x0 5				IF	ID	EX	MEM	WB				
sub x7 x6 x5					IF	ID	-	-	EX	MEM	WB	

Figure 6: Stalls in the pipeline diagram in without forwarding case

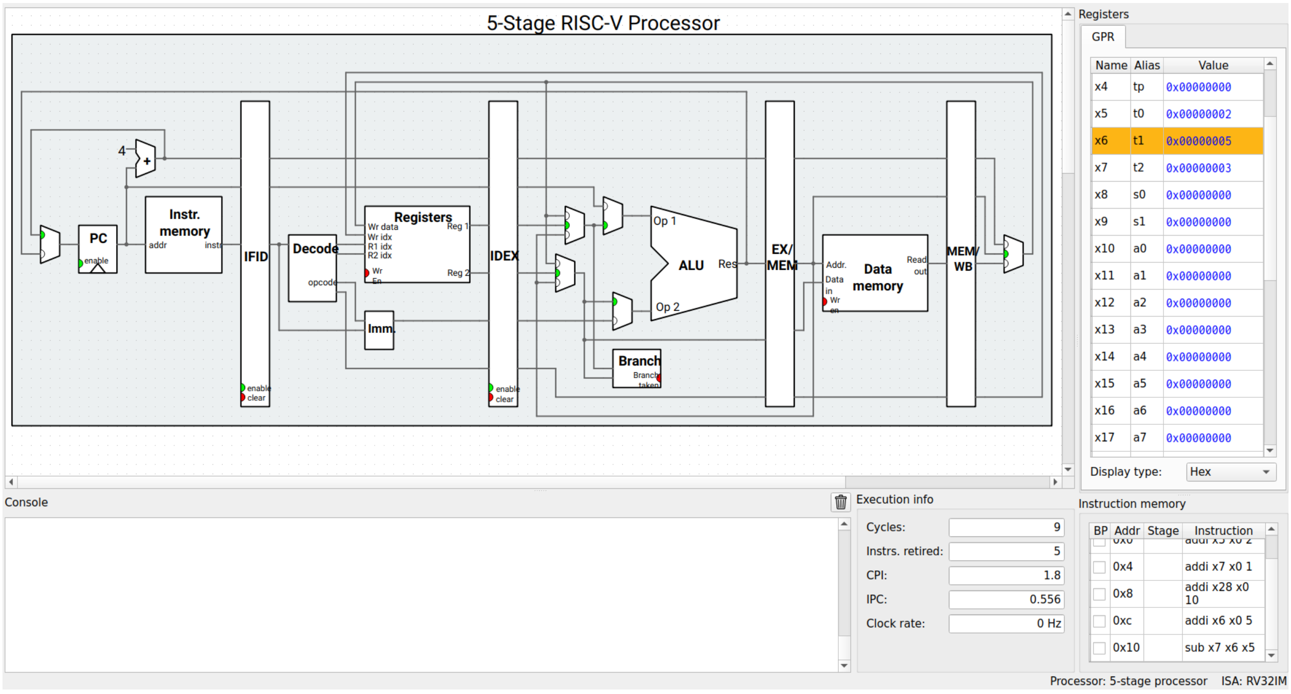
3.1.2 Run on a 5 stage processor

The number of cycles used in this case are **9**. The difference in number of cycles arises due to forwarding mechanism. In the first case, when there is no forwarding, 2 stalls were inserted at last instruction at cycle 6 and 7 in order to prevent hazard. However, here forwarding is available so the value is directly read as the result already present in ALU is forwarded to subsequent instructions, which prevents hazard without inserting stalls, and therefore does not require 2 extra cycles. So, the cycles used here are 9.



	0	1	2	3	4	5	6	7	8	9
addi x5 x0 2	IF	ID	EX	MEM	WB					
addi x7 x0 1		IF	ID	EX	MEM	WB				
addi x28 x0 10			IF	ID	EX	MEM	WB			
addi x6 x0 5				IF	ID	EX	MEM	WB		
sub x7 x6 x5					IF	ID	EX	MEM	WB	

Figure 7: There were no stalls present in this case



Name	Alias	Value
x4	tp	0x00000000
x5	t0	0x00000002
x6	t1	0x00000005
x7	t2	0x00000003
x8	s0	0x00000000
x9	s1	0x00000000
x10	a0	0x00000000
x11	a1	0x00000000
x12	a2	0x00000000
x13	a3	0x00000000
x14	a4	0x00000000
x15	a5	0x00000000
x16	a6	0x00000000
x17	a7	0x00000000

Execution info

Cycles:	9
Instrs. retired:	5
CPI:	1.8
IPC:	0.556
Clock rate:	0 Hz

BP	Addr	Stage	Instruction
	0x4		addi x7 x0 1
	0x8		addi x28 x0 10
	0xc		addi x6 x0 5
	0x10		sub x7 x6 x5

Processor: 5-stage processor ISA: RV32IM

Figure 8: Screenshot after running on 5 stage processor

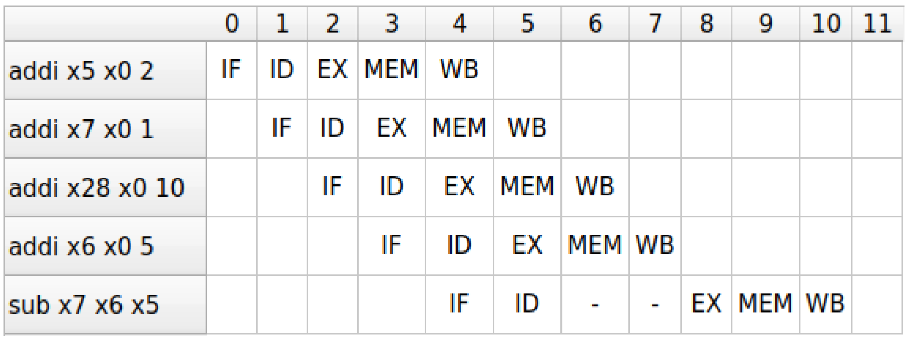
3.2 Part (b) Different number of cycles in different order

3.2.1 Original

The original code I used for this part uses the same instructions as part(a). The code is -

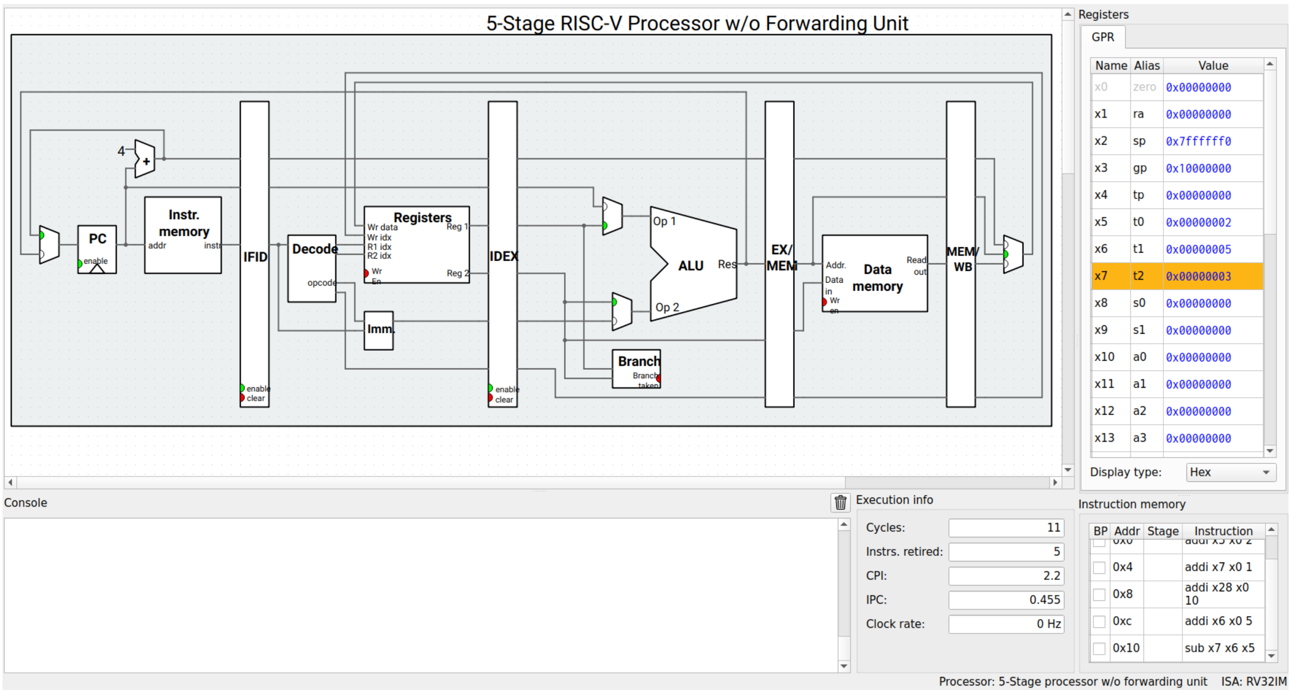
```
li t0 2
li t2 1
li t3 10
li t1 5
sub t2 t1 t0
```

Number of cycles used in original code = 11. The processor used here s 5 staged without forwarding and it has hazard detection. Since, there is no forwarding, it inserts stalls appropriate positions in order to prevent hazards.



	0	1	2	3	4	5	6	7	8	9	10	11
addi x5 x0 2	IF	ID	EX	MEM	WB							
addi x7 x0 1		IF	ID	EX	MEM	WB						
addi x28 x0 10			IF	ID	EX	MEM	WB					
addi x6 x0 5				IF	ID	EX	MEM	WB				
sub x7 x6 x5					IF	ID	-	-	EX	MEM	WB	

Figure 9: Stalls present in original code



Registers

GPR

Name	Alias	Value
x0	zero	0x00000000
x1	ra	0x00000000
x2	sp	0x7ffffff0
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000002
x6	t1	0x00000005
x7	t2	0x00000003
x8	s0	0x00000000
x9	s1	0x00000000
x10	a0	0x00000000
x11	a1	0x00000000
x12	a2	0x00000000
x13	a3	0x00000000

Display type: Hex

Cycles:	11
Instrs. retired:	5
CPI:	2.2
IPC:	0.455
Clock rate:	0 Hz

BP	Addr	Stage	Instruction
	0x0		addi x5 x0 2
	0x4		addi x7 x0 1
	0x8		addi x28 x0 10
	0xc		addi x6 x0 5
	0x10		sub x7 x6 x5

Processor: 5-Stage processor w/o forwarding unit ISA: RV32IM

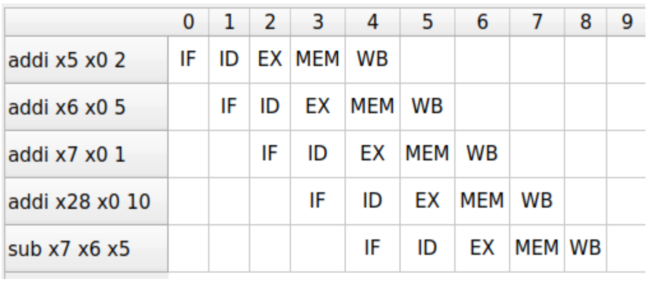
Figure 10: Original code with 11 cycles

3.2.2 Modified

In modified code, I changed the order of statement as -

```
li t0 2
li t1 5
li t2 1
li t3 10
sub t2 t1 t0
```

That is, I have swapped 2nd and 4th instruction which changed the order of instructions. Number of cycles in this modified code = 9. The reason is due to no stalls being inserted in



	0	1	2	3	4	5	6	7	8	9
addi x5 x0 2	IF	ID	EX	MEM	WB					
addi x6 x0 5		IF	ID	EX	MEM	WB				
addi x7 x0 1			IF	ID	EX	MEM	WB			
addi x28 x0 10				IF	ID	EX	MEM	WB		
sub x7 x6 x5					IF	ID	EX	MEM	WB	

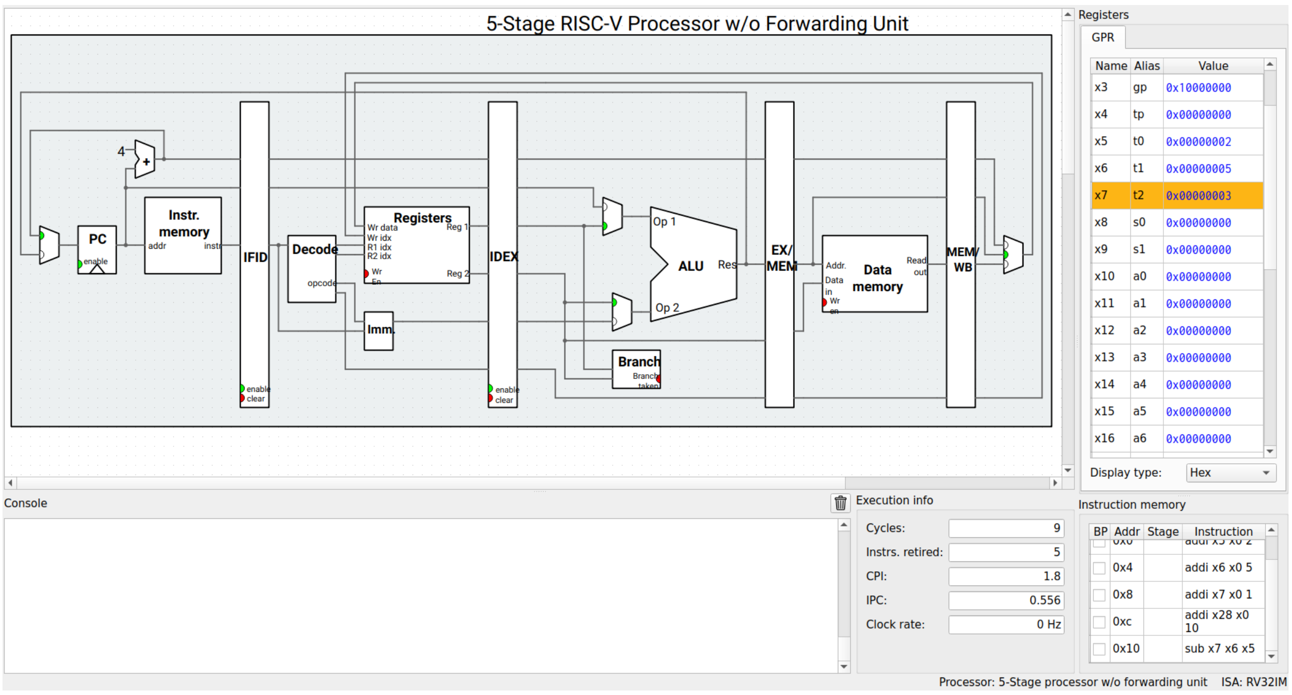
Figure 11: No stalls in my modifed code

modified version and which is because I removed the possibility of hazard by changing order. In original, the value of 5 into `t1` was being written just before last instruction, where it was being read. Since it may cause RAW hazard, the hazard detection inserted 2 stalls in original code to prevent this. Stalls were inserted just to stop pipeline and wait so that value of at `t1` is correctly written before being used up in next instruction, leading to 11 cycles used.

Since in modified code, value of 5 is written in `t1` in 2nd statement, which gets completed by cycle 5. Also, the other statements after 2nd do not affect data being read in last statement, there is no possibility of hazard in this modified version. And that's why stalls are not inserted here which leads to the number of cycles in modified code to be 9, which is different from the cycles in original version and this difference in number of cycles occurs by same set of instructions, when executed in a different order in 5 stage processor without forwarding.

Also, note that both the versions are performing the same task and the execution result after both is same -

$$\texttt{t0} = 2,\ \texttt{t1} = 5,\ \texttt{t2} = \texttt{t1 - t0} = 3$$



Name	Alias	Value
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000002
x6	t1	0x00000005
x7	t2	0x00000003
x8	s0	0x00000000
x9	s1	0x00000000
x10	a0	0x00000000
x11	a1	0x00000000
x12	a2	0x00000000
x13	a3	0x00000000
x14	a4	0x00000000
x15	a5	0x00000000
x16	a6	0x00000000

Display type: Hex

Cycles:	9
Instrs. retired:	5
CPI:	1.8
IPC:	0.556
Clock rate:	0 Hz

BP	Addr	Stage	Instruction
	0x0		addi x5 x0 2
	0x4		addi x6 x0 5
	0x8		addi x7 x0 1
	0xc		addi x28 x0 10
	0x10		sub x7 x6 x5

Processor: 5-Stage processor w/o forwarding unit ISA: RV32IM

Figure 12: Modified code with 9 cycles

4 Question 4

```
li t0 1
li t1 2
li t2 3
li t3 4
add t0 t0 t1
li t1 5
add t2 t2 t3
li t3 6
add t0 t0 t1
li t1 7
add t2 t2 t3
li t3 8
add t0 t0 t1
li t1 9
add t2 t2 t3
li t3 10
add t0 t0 t1
add t2 t2 t3
add t0 t0 t2

# this code loads 10 integers (1 to 10)
# into different registers and adds them
# final addition result (55) is stored in t0
```

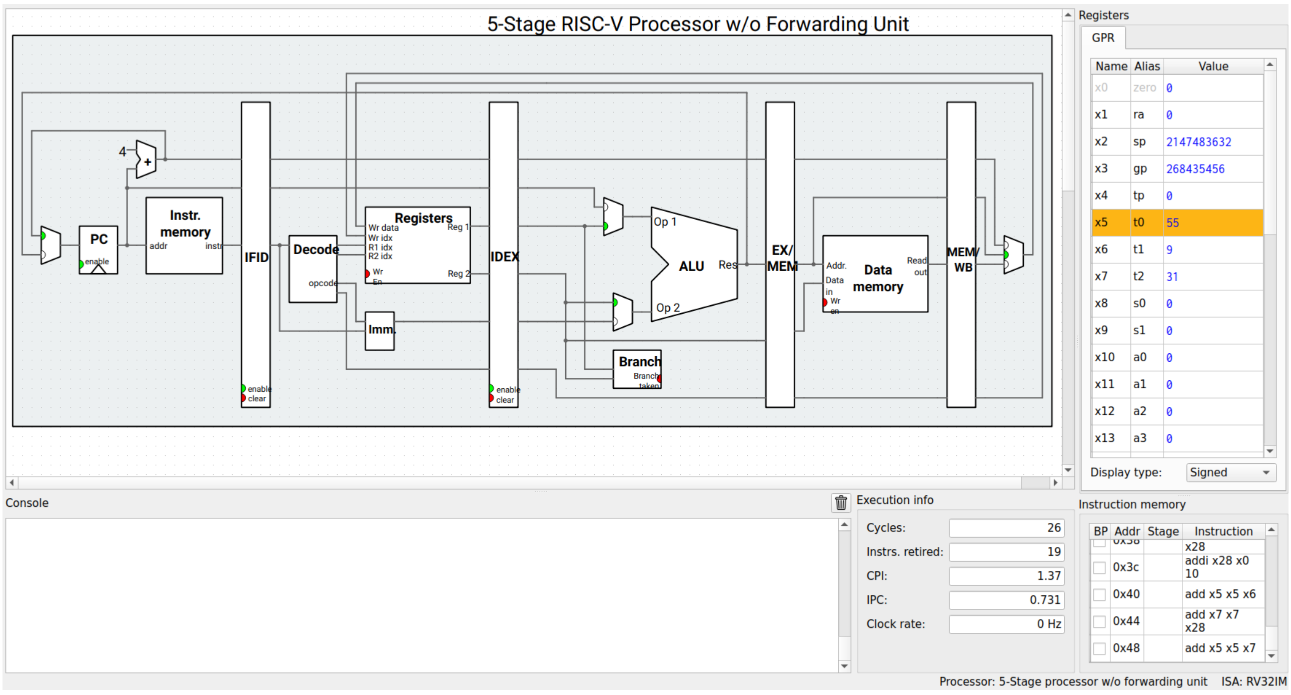
Here, I am using 4 registers (t0, t1, t2, t3) and loading 10 integers into them in some particular order so that I can minimize cycles used. Just for example, I am using the first 10 natural number in my code. After adding all these number, the final result -

$$\sum_{i=1}^{10} n = 55$$

This sum value of 55 is stored in `t0` at last.

The minimum number of cycles that I got = 26
And the minimum number of registers I got = 4

For convincing me that this would be minimum, I thought that for a 5 staged processor, the minimum number of cycles needed were 23. Now, when I use a 5 staged processor without forwarding, I need to maintain some gap between the registers loading and adding as well as consecutive adding into same register so as to use minimum stalls leading to minimum cycles. For this, I used 4 registers and grouped them into 3 for avoiding extra stalls. So, at last 3 extra cycles would be needed for adding these 3, leading to 23 + 3 = 26 minimum cycles with 4 minimum registers used.



Name	Alias	Value
x0	zero	0
x1	ra	0
x2	sp	2147483632
x3	gp	268435456
x4	tp	0
x5	t0	55
x6	t1	9
x7	t2	31
x8	s0	0
x9	s1	0
x10	a0	0
x11	a1	0
x12	a2	0
x13	a3	0

Cycles:	26
Instrs. retired:	19
CPI:	1.37
IPC:	0.731
Clock rate:	0 Hz

BP	Addr	Stage	Instruction
	0x38		x28
	0x3c		addi x28 x0 10
	0x40		add x5 x5 x6
	0x44		add x7 x7 x28
	0x48		add x5 x5 x7

Processor: 5-Stage processor w/o forwarding unit ISA: RV32IM

Figure 13: Loading and adding 10 integers with 26 cycles and 4 registers