Lab4: Visualizing instruction Pipeline

Ravi Shankar Meena (210050135)
March 8, 2023

1 Q1

1.1 Q1-A

Code:-

```
.text
        main:
        li s0 , 1
        li s1 , 2
        add t0, s0, s1
        add a0, t0 ,x0
```

Explanations:-
The above code encounters a RAW (Read-After-Write) hazard because the instruction "add t0, s0, s1" writes the result of the addition operation to register t0, and the following instruction "add a0, t0, x0" reads from register t0.

Since the two instructions are executed one after the other without any intervening instruction that modifies t0, there is a risk that the value of t0 used in the second instruction is not the correct value due to the delay between the write operation in the first instruction and the read operation in the second instruction.

1.2 Q1-B

In a vanilla 5-stage processor, there is no possibility of a WAR hazard. This is because the write-back stage, where the result of an instruction is written to a register or memory, is always executed after the execute stage, where the instruction is performed on the data. Therefore, a write instruction will always complete before a subsequent read instruction tries to use the value.

In other words, the vanilla 5-stage processor's pipeline design guarantees that write operations always complete before read operations can be executed. Therefore, there is no possibility of a WAR hazard occurring in a vanilla 5-stage processor.

2 Q2

2.1 Q2-A

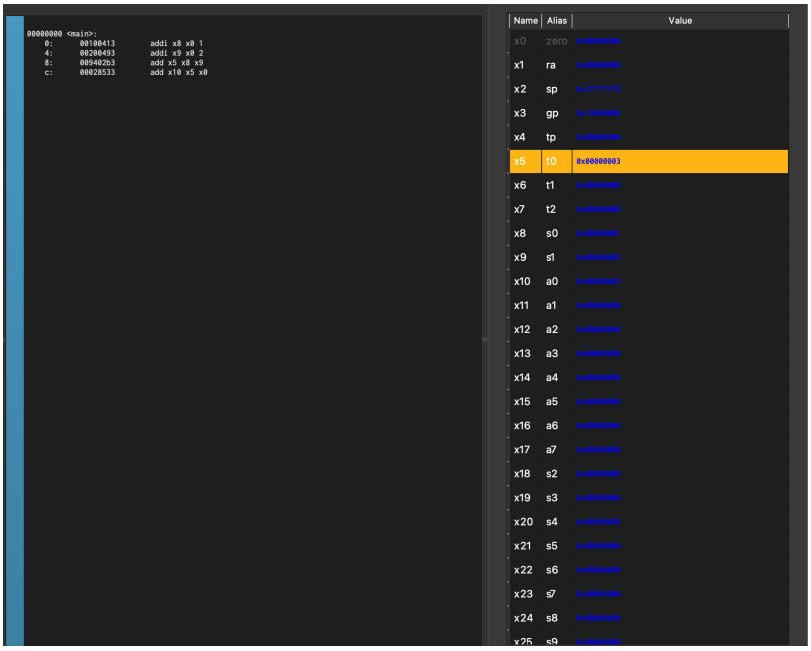
Code:-

```
.text
main:
   li s0 , 1
   li s1 , 2
   add t0, s0, s1
   nop
   nop
   nop
   add a0 , t0 , x0
```

Explanations:-

In this code, the add instruction that stores the result in register t0 modifies the register s0, which is then read by the following add instruction that uses the value of s0 in its calculation. As a result, there is a RAW hazard between the two add instructions.

To resolve this hazard, the processor needs to stall the execution of the second add instruction until the first add instruction completes and writes its result to the register file. In a 5-stage processor, this can be done by inserting a NOP instruction or a bubble in the execute stage of the second add instruction. In this code, there are two NOP instructions after the first add instruction, which can be used to eliminate the RAW hazard by stalling the second add instruction until the result of the first add instruction is written to the register file. Therefore, the code is safe from RAW hazards while using a 5-stage processor.



2.2 Q2-B

In a vanilla 5-stage processor, there is no possibility of a WAR hazard. This is because the write-back stage, where the result of an instruction is written to a register or memory, is always executed after the execute stage, where the instruction is performed on the data. Therefore, a write instruction will always complete before a subsequent read instruction tries to use the value.

In other words, the vanilla 5-stage processor's pipeline design guarantees that write operations always complete before read operations can be executed. Therefore, there is no possibility of a WAR hazard occurring in a vanilla 5-stage processor.

3 Q3

3.1 Q3-A

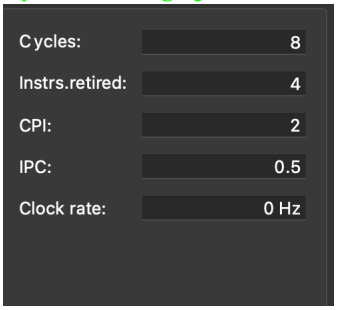
Code:-

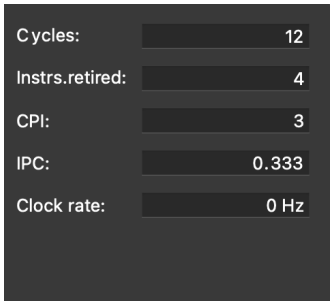
```
.text
 main:
    li s0 , 1
    li s1 , 2
    add t0, s0, s1
    add t3, t0 , x0
```

Explanations:-

Cycle in 5 stage processor without forwarding:- 12

Cycle in 5 stage processor:- 8





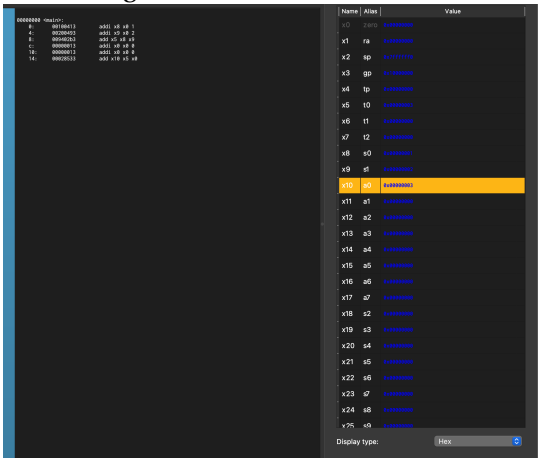
This code is a simple MIPS assembly code that initializes two registers s0 and s1 with the values 1 and 2, respectively. Then it adds the contents of s0 and s1 and stores the result in register t0. After that, it adds the contents of register t0 with 0 and stores the result in register t3.

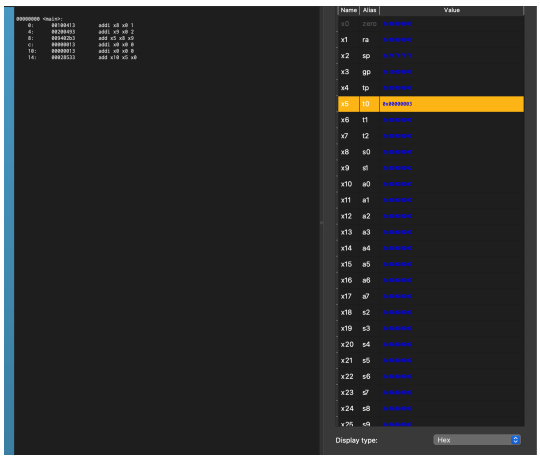
When this code is run on a 5-stage processor without forwarding, it will take a higher number of cycles to execute than when run on a 5-stage processor with forwarding. This is because without forwarding, the processor must wait until the contents of t0 are written back to the register file before it can be used as an input for the second add instruction.

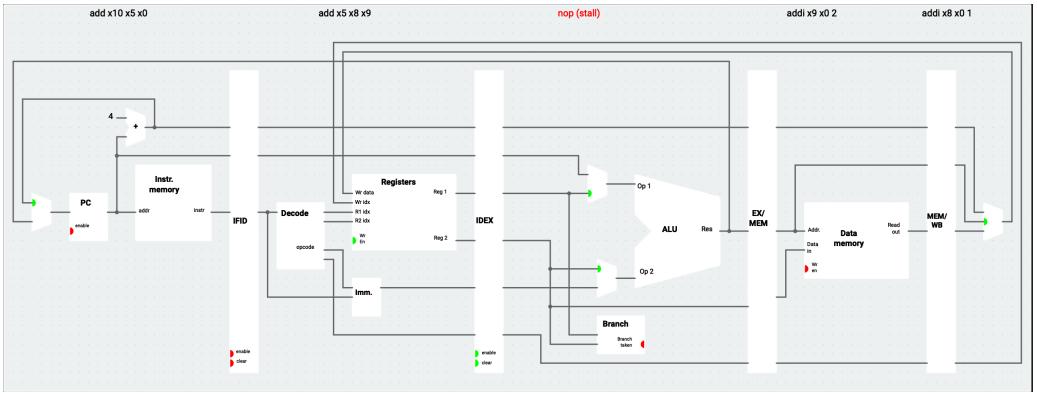
In a 5-stage pipeline without forwarding, the second add instruction will have to wait until the write-back stage of the first add instruction is complete before it can access the value of t0 in the register file. This creates a pipeline stall, which increases the number of cycles needed to execute the code.

On the other hand, a 5-stage pipeline with forwarding can eliminate the pipeline stall by forwarding the result of the first add instruction directly to the input of the second add instruction. This allows the second add instruction to proceed without waiting for the contents of t0 to be written back to the register file, resulting in a faster execution time.

Therefore, in this code, using a 5-stage processor without forwarding will result in a higher number of cycles needed to execute the code compared to using a 5-stage processor with forwarding.







When running above code in Ripes, the processor detects this RAW and inserts a stall (NOP instruction) between the two instructions to allow the first instruction to complete writing to t0 before the second instruction reads from t0. This stall ensures that the second instruction reads the correct value from t0.

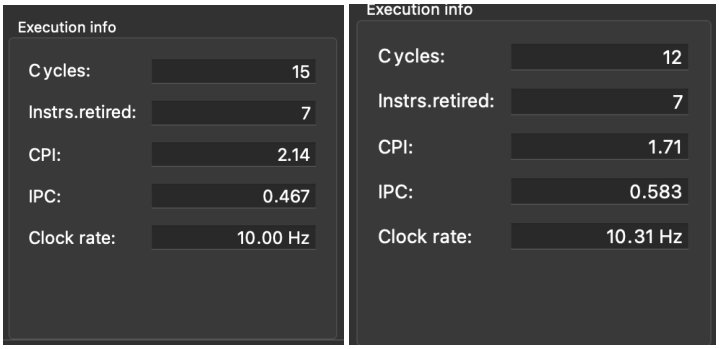
3.2 Q3-B

Code:-

```
original
.text
li s0,0
li s1,1
li s2,2
li s3,3
main:
add t0, s0, s1
add t1, t0, s2
add t2, t1, s3
```

```
modified
.text
 li s0,0
 li s1,1
 li s2,2
 li s3,3
 main:
 add t0, s0, s1
 add t1, s3, s2
 add t2, t0, s1
```

Explanations:-
Cycle in Original code:- 15
Cycle in Modified code:- 12

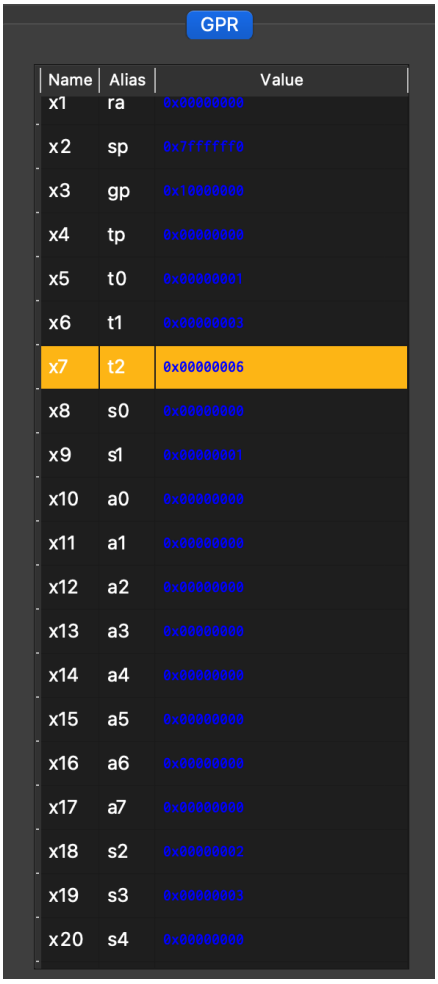


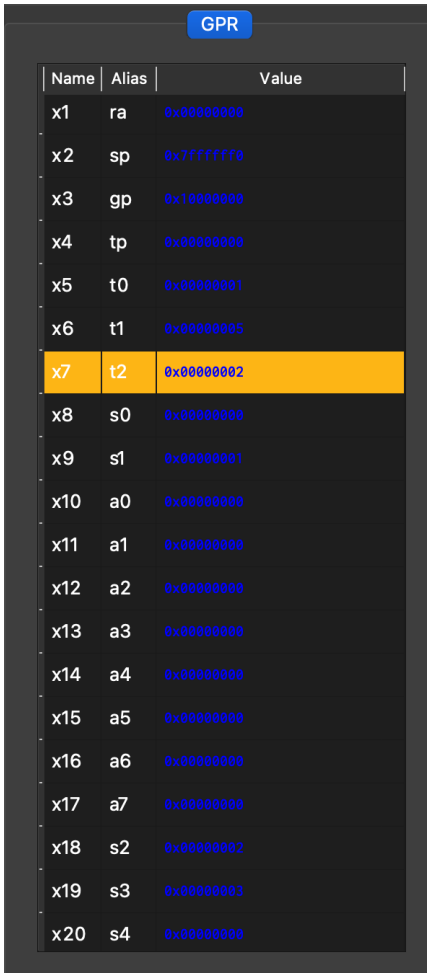
In both cases, there is a Read-After-Write (RAW) hazard between instruction 1 and instruction 2. In the first code, instruction 2 cannot execute until instruction 1 has completed, and in the second code, instruction 2 does not depend on the result of instruction 1, so it can execute immediately.

However, in the first code, instruction 3 depends on the result of instruction 2, so it cannot execute until instruction 2 has completed. This results in a total of 4 cycles being required to execute all 3 instructions.

In the second code, instruction 3 depends on the result of instruction 1, so it cannot execute until instruction 1 has completed. This results in a total of 5 cycles being required to execute all 3 instructions.

Therefore, the same set of instructions, when executed in a different order, can take a different number of cycles to execute on a 5-stage processor without forwarding.


GPR
Name
Alias
Value
x1 ra 0x00000000
x2 sp 0x7ffffff0
x3 gp 0x10000000
x4 tp 0x00000000
x5 t0 0x00000001
x6 t1 0x00000003
x7 t2 0x00000006
x8 s0 0x00000000
x9 s1 0x00000001
x10 a0 0x00000000
x11 a1 0x00000000
x12 a2 0x00000000
x13 a3 0x00000000
x14 a4 0x00000000
x15 a5 0x00000000
x16 a6 0x00000000
x17 a7 0x00000000
x18 s2 0x00000002
x19 s3 0x00000003
x20 s4 0x00000000


GPR
Name
Alias
Value
x1 ra 0x00000000
x2 sp 0x7ffffff0
x3 gp 0x10000000
x4 tp 0x00000000
x5 t0 0x00000001
x6 t1 0x00000005
x7 t2 0x00000002
x8 s0 0x00000000
x9 s1 0x00000001
x10 a0 0x00000000
x11 a1 0x00000000
x12 a2 0x00000000
x13 a3 0x00000000
x14 a4 0x00000000
x15 a5 0x00000000
x16 a6 0x00000000
x17 a7 0x00000000
x18 s2 0x00000002
x19 s3 0x00000003
x20 s4 0x00000000

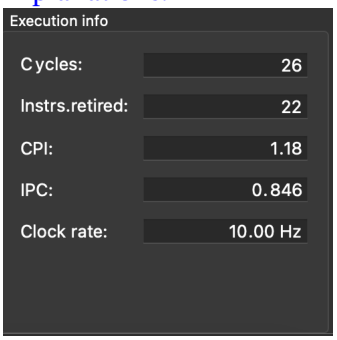
4 Q4

Code:-

```
.data
.text
.globl main
main:

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
li  t1 , 0
add t2 , t2, t3
li t3 , 0
li t1 ,0
add t0 ,t0 ,t2
```

Explanations:-



The program first loads the values 1, 2, 3, and 4 into registers t0, t1, t2, and t3, respectively, using the "li" (load immediate) instruction. It then performs a series of addition operations on t0 and t2, which involve adding t1 and t3 to t0 and t2, respectively. After each addition, the program loads a new value into t1 or t3 using the "li" instruction.Finally, the program adds the contents of t0 and t2 together and stores the result in t0. The program ends by setting the contents of t1, t2, and t3 to 0 using the "li" instruction.