CO_Lab3_111550076

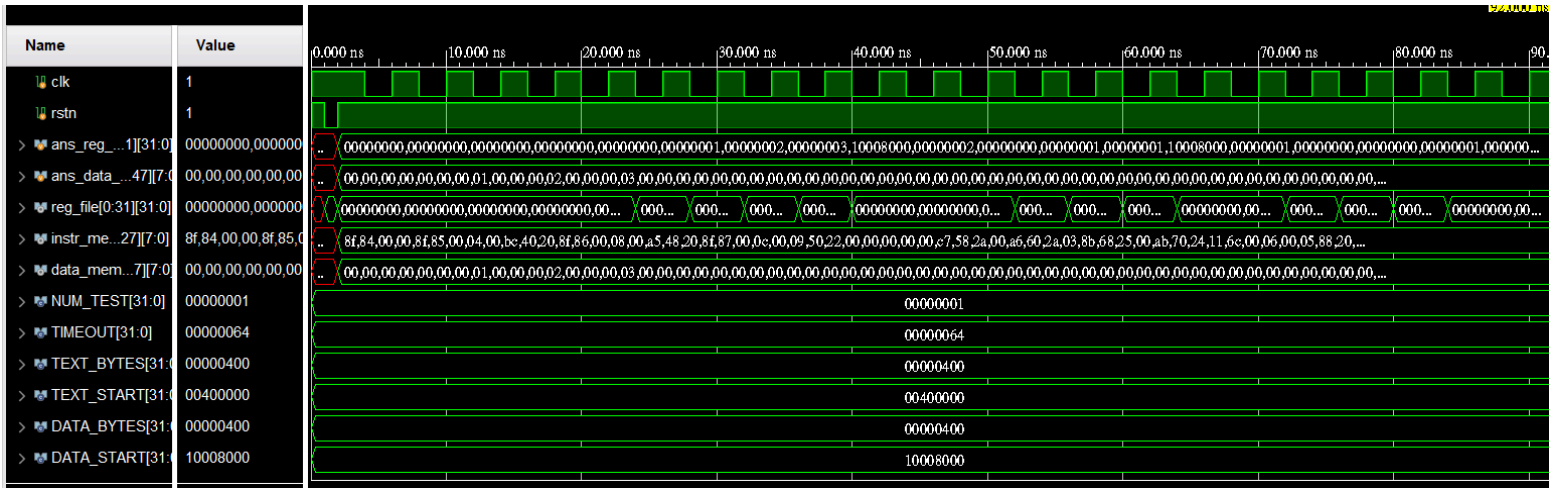
notion link:

[CO_Lab3_111550076]

---

1. Experimental Result

1. Show the waveform screen shot of the test we provided.



2. What other testcase you've tested? Why you choose them?

I reordered the given assembly code to eliminate hazard issues and randomly inserted three `addi` instructions to verify their functionality. The fact that the reordered and newly added code executed correctly indicates the program is functioning as intended. Notably, I retained the test results where three instructions after `beq` execute before jumping to `end` due to the absence of hazard handling and forwarding. This ensures that my implementation of the pipeline CPU reflects one without hazard handling or forwarding mechanisms.

```
main:   lw      $a1, 4($gp)
        lw      $a0, 0($gp)
        lw      $a2, 8($gp)
        add     $t1, $a1, $a1
        add     $t0, $a1, $gp
        lw      $a3, 12($gp)
        sub     $t2, $0, $t1
        addi    $t7, $a1, 200 #addi
```

```
        nop
        slt     $t3, $a2, $a3
        slt     $t4, $a1, $a2
        addi    $t1, $t1, 10  #addi
        addi    $t2, $t2, 20  #addi
        and     $t6, $a1, $t3
        or      $t5, $gp, $t3
        beq     $t3, $t4, end   # taken
        add     $s1, $0, $a1    #being executed
        add     $s2, $0, $a2    #being executed
        add     $s3, $0, $a3    #being executed
        add     $s4, $a2, $a2
        add     $s5, $a2, $a3
        add     $s6, $a3, $a3
end:    add     $s7, $s4, $a2
```

```
R0 (r0)  = 00000000
R1 (at)  = 00000000
R2 (v0)  = 00000000
R3 (v1)  = 00000000
R4 (a0)  = 00000000
R5 (a1)  = 00000001
R6 (a2)  = 00000002
R7 (a3)  = 00000003
R8 (t0)  = 10008001
R9 (t1)  = 0000000c #addi result
R10 (t2) = 00000012 #addi result
R11 (t3) = 00000001
R12 (t4) = 00000001
R13 (t5) = 10008001
R14 (t6) = 00000001
R15 (t7) = 000000c9 #addi result
```

```
R16 (s0) = 00000000
R17 (s1) = 00000001 #add occur
R18 (s2) = 00000002 #add occur
R19 (s3) = 00000003 #add occur
R20 (s4) = 00000000
R21 (s5) = 00000000
R22 (s6) = 00000000
R23 (s7) = 00000002
R24 (t8) = 00000000
R25 (t9) = 00000000
R26 (k0) = 00000000
R27 (k1) = 00000000
R28 (gp) = 10008000
R29 (sp) = 7fffff74
R30 (s8) = 00000000
R31 (ra) = 00000000
```

2. Answer the following Questions

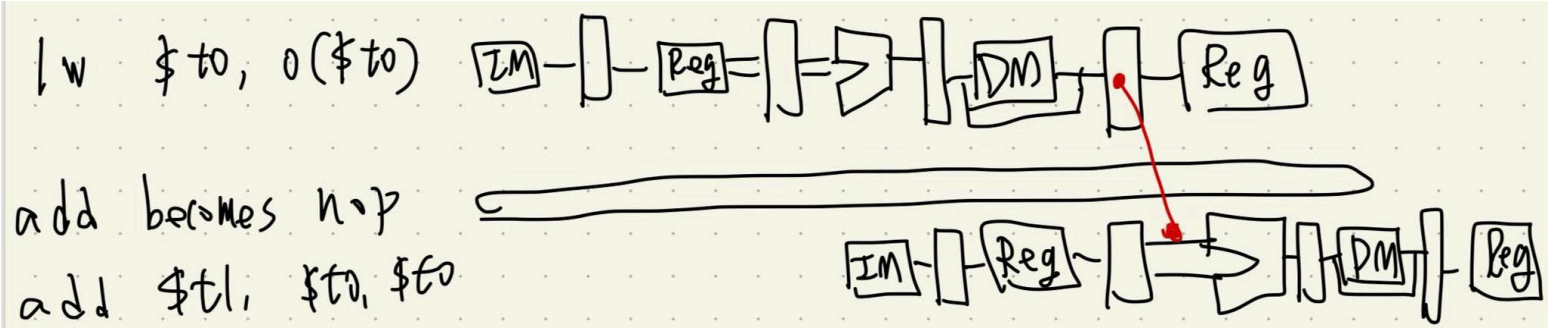
1. For each code sequence below, state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding.

(1). Sequence 1

```
lw $t0, 0($t0)
add $t1, $t0, $t0
```

Answer: **Must stall (1 clock with forwarding)**

Explain: The `rs` and `rd` of the `add` operation depend on the result of the `lw` instruction. Therefore, we must retrieve the value of `$t0` before reaching the `EX` stage of the `add` operation. The value of `$t0` first becomes available at the MEM-WB register. By utilizing forwarding, we can pass the value from the MEM-WB register of the `lw` instruction directly to the `EX` stage of the `add` operation, requiring only a single clock cycle of stalling.

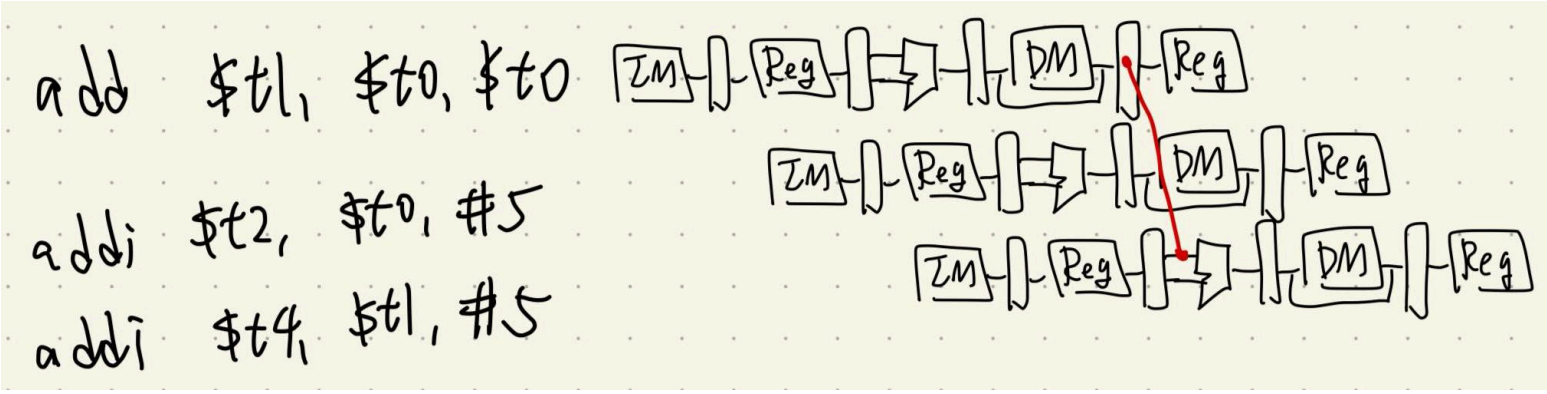


(2). Sequence 2

```
add $t1, $t0, $t0
addi $t2, $t0, #5
addi $t4, $t1, #5
```

Answer: **can avoid stalls using only forwarding.**

Explain: The third operation relies on `$t1`, which would normally have to wait for the `add` instruction to complete its write-back before obtaining the value, which resulting in several clock stalls. With forwarding, however, the value of `$t1` is passed directly from the `add` instruction's MEM-WB register to the execution stage of the subsequent `addi`, thereby preventing any stall.



(3). Sequence 3

```
addi $t1, $t0, #1
addi $t2, $t0, #2
addi $t3 $t0, #2
addi $t3, $t0, #4
addi $t5, $t0, #5
```

Answer: **can execute without stalling or forwarding.**

Each instruction uses `$t0` as its source operand, and since the value of `$t0` isn't modified throughout the sequence, each operation will execute without any stalls or data hazards. The result of each `addi` instruction will be correctly computed without interference from subsequent instructions, and they will execute independently.

2. Explain the difference between throughput and latency.

`Throughput` and `latency` are two key performance metrics, particularly in computing and networking:

1. **Throughput:**
   - Measures how much work or data a system can handle over time.
   - Expressed as a rate, like "requests per second."
2. **Latency:**
   - Measures the time it takes to complete a specific task or send data.
   - Expressed in units like milliseconds.

Difference:

- `Throughput` is about the volume of work done, while `latency` is about the speed at which a task is finished.

3. A group of students were debating the efficiency of the five-stage pipeline when one student pointed out that not all instructions are active in every stage of the pipeline. After deciding to ignore the effects of hazards, they made the following five statements. Which ones are correct? Explain why or why not.

(1). Allowing jumps, branches, and ALU instructions to take fewer stages than the five required by the load instruction will increase pipeline performance under all circumstances.

Answer: **False.**

Explain: Reducing stages can lead to increasing complexity in handling dependencies and hazards, potentially increasing the occurrence of stalls. Furthermore, performance improvements depend on the mix of instruction types and their specific patterns in the executed programs.

(2). Trying to allow some instructions to take fewer cycles does not help, since the throughput is determined by the clock cycle; the number of pipe stages per instruction affects latency, not throughput.

Answer: **True**

Explain: The throughput of a pipelined processor is primarily determined by the clock cycle, which is set according to the longest pipeline stage. While reducing the number of cycles needed for some instructions could decrease their latency, it does not improve throughput if the clock speed remains the same. The throughput of the pipeline is generally influenced by how quickly instructions can be fetched, decoded, and completed at each clock cycle, while the overall latency depends on the total number of stages.

(3). You cannot make ALU instructions take fewer cycles because of the writeback of the result, but branches and jumps can take fewer cycles, so there is some opportunity for improvement.

Answer: **True**

Explain: ALU instructions operate according to a predetermined number of cycles defined by the pipeline structure. Branches and jumps can potentially reduce cycles through prediction technologies, yet they are also influenced by the pipeline architecture's limitations and possibilities, similar to ALU instructions.

(4). Instead of trying to make instructions take fewer cycles, we should explore making the pipeline longer, so that instructions take more cycles, but

the cycles are shorter. This could improve performance.

Answer: **True**

Explain: In a deeply pipelined architecture, instructions take more cycles to complete since each pipeline stage is shorter. However, because each cycle duration is reduced, the overall clock speed can be increased, potentially improving the throughput of the pipeline. This design is particularly advantageous if the additional stages do not lead to an increase in stalls or hazards.