Organization of Digital Computers

EECS 112L

Lab #4: Pipeline Processor

Miles Jennings

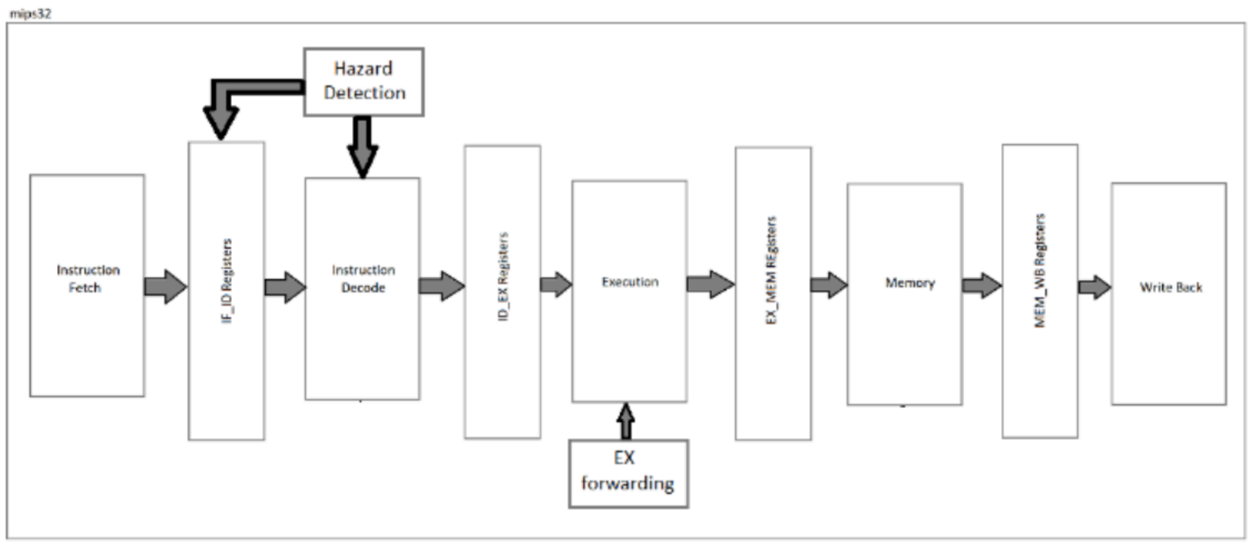
66356019

2/24/2025

1. Objective

The objective of this lab is to create a pipeline processor in Verilog. The general format of the processor is as follows.

Figure 1: Pipeline processor

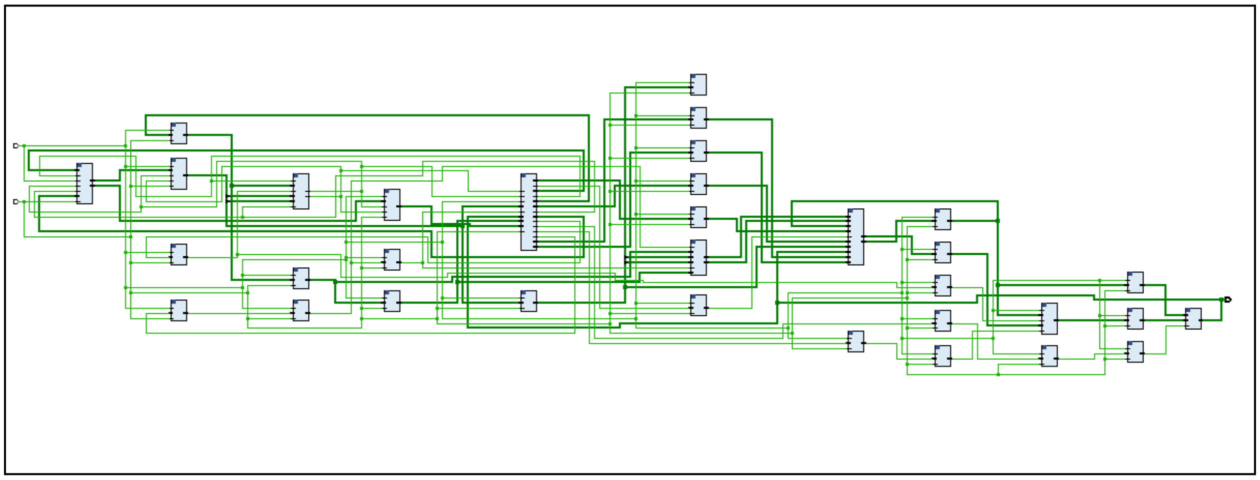


I am going to learn how to build/create components by diving into the actual inputs and outputs of each stage. I will then combine them into the same file so that they create a pipeline processor.

2. Procedure

I first looked at the circuit of each stage. Using the given code and my knowledge of building components, I was able to replicate them in vivado using instantiation of the components.
I was able to debug by using the TCL console and looking at the object values in different scopes. For example, when I was getting failed test cases, I went through each object starting from the IF pipe stage and making sure that there were no Z or X values where there was supposed to be a value. Additionally, using the debugging insights in the lab manual were useful as well.

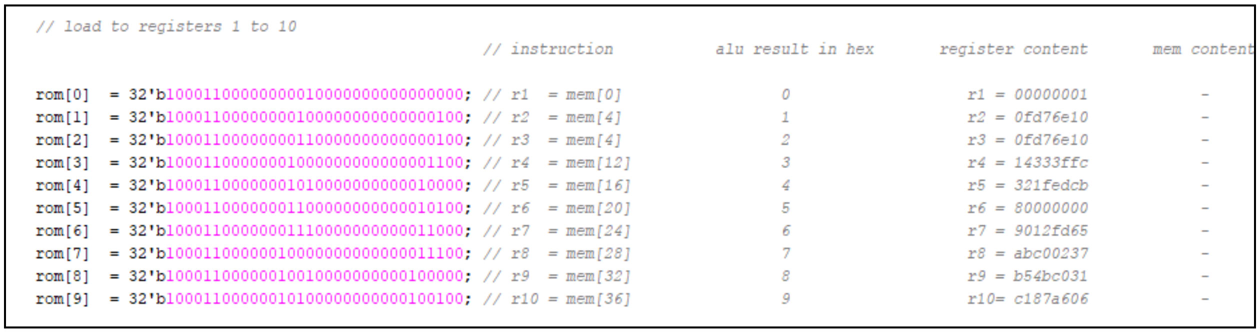
After checking that all of my test cases were successful from the testbench given, I was able to start viewing my simulation results and breaking them down to make sure they made sense. Utilizing lab time and bouncing ideas off of fellow peers, I was able to create a working pipeline processor that handled forwarding, stalling, and data hazards to name a few. If I were to do one thing differently, I wish I would have started sooner because breaking down and understanding the test cases took a good amount of time.



Elaborated Design

3. Simulation Results

- Test Case 1:

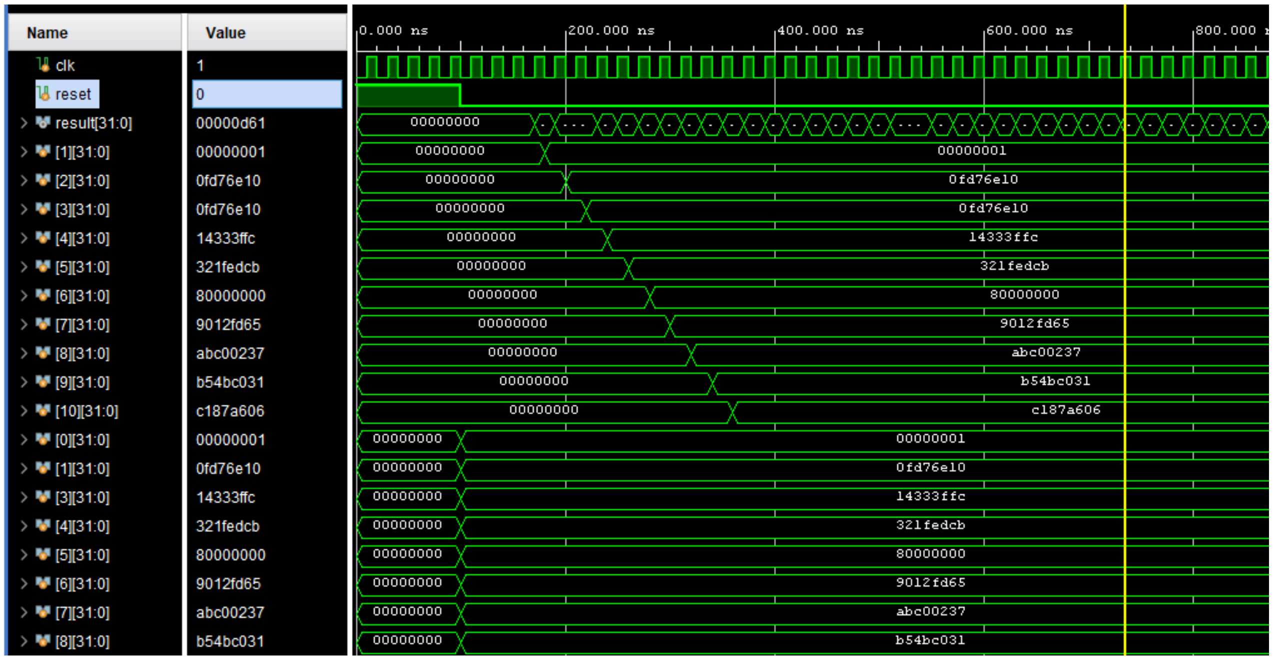


```
// load to registers 1 to 10
                                                           // instruction           alu result in hex      register content      mem content

rom[0]  = 32'b10001100000000010000000000000000; // r1  = mem[0]            0               r1 = 00000001           -
rom[1]  = 32'b10001100000000100000000000000100; // r2  = mem[4]            1               r2 = 0fd76e10           -
rom[2]  = 32'b10001100000000110000000000000100; // r3  = mem[4]            2               r3 = 0fd76e10           -
rom[3]  = 32'b10001100000001000000000000001100; // r4  = mem[12]           3               r4 = 14333ffc           -
rom[4]  = 32'b10001100000001010000000000010000; // r5  = mem[16]           4               r5 = 321fedcb           -
rom[5]  = 32'b10001100000001100000000000010100; // r6  = mem[20]           5               r6 = 80000000           -
rom[6]  = 32'b10001100000001110000000000011000; // r7  = mem[24]           6               r7 = 9012fd65           -
rom[7]  = 32'b10001100000010000000000000011100; // r8  = mem[28]           7               r8 = abc00237           -
rom[8]  = 32'b10001100000010010000000000100000; // r9  = mem[32]           8               r9 = b54bc031           -
rom[9]  = 32'b10001100000010100000000000100100; // r10 = mem[36]           9               r10= c187a606           -
```

This test is used to prove that the different stages of my pipeline processor work. They are just load word instructions used to load the value from mem[X] into register rY. This proves that my processor correctly fetches the instruction (in this case lw), decodes it to know what memory address to get the value from and what register to store it into, executes by calculating the memory address, reads data from the memory, and writes the loaded data to the designated registers.

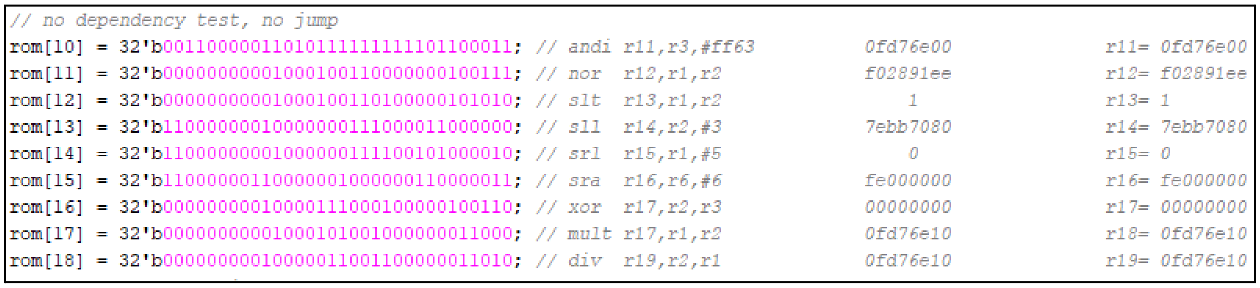
Simulation results:



Name	Value
clk	1
reset	0
result[31:0]	00000d61
[1][31:0]	00000001
[2][31:0]	0fd76e10
[3][31:0]	0fd76e10
[4][31:0]	14333ffc
[5][31:0]	321fedcb
[6][31:0]	80000000
[7][31:0]	9012fd65
[8][31:0]	abc00237
[9][31:0]	b54bc031
[10][31:0]	c187a606
[0][31:0]	00000001
[1][31:0]	0fd76e10
[3][31:0]	14333ffc
[4][31:0]	321fedcb
[5][31:0]	80000000
[6][31:0]	9012fd65
[7][31:0]	abc00237
[8][31:0]	b54bc031

As one can see, this waveform shows the values from each memory address being loaded into the registers. Since the memory is byte addressed but the instructions are word addressed, the load word uses an offset of the actual memory location. This makes us divide by 4 to reverse the offset and find the true memory location. This is why underneath the registers (1-10), I have the values at memory addresses (1-8) to show that they line up with the values being stored into the registers. For example, mem[4] has the true memory location of 1, so the value at memory location 1 = the value being stored into register 2 and 3.

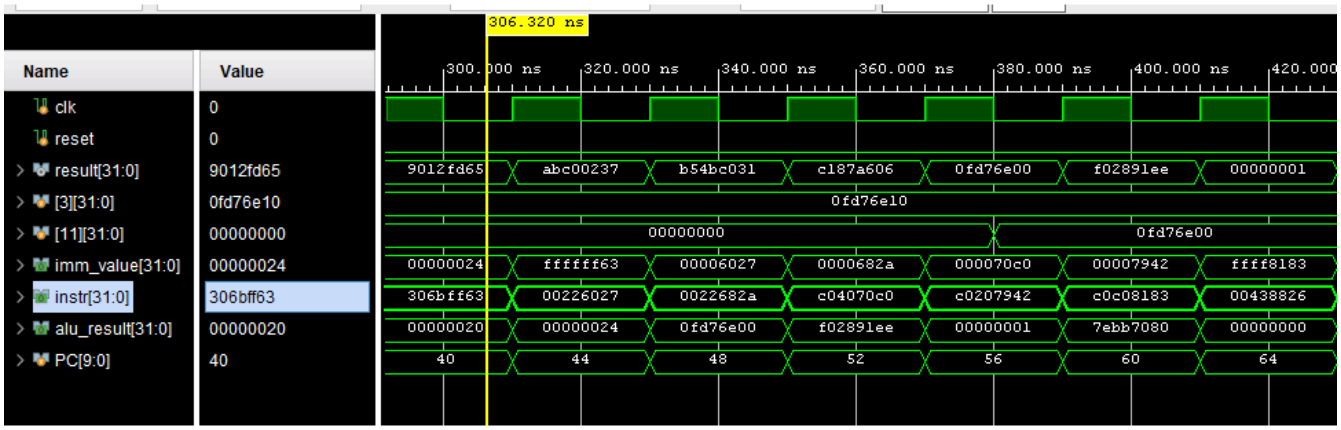
- <u>Test Case 2:</u>



```
// no dependency test, no jump
rom[10] = 32'b00110000011010111111111101100011; // andi r11,r3,#ff63          0fd76e00             r11= 0fd76e00
rom[11] = 32'b00000000001000100110000000100111; // nor  r12,r1,r2             f02891ee             r12= f02891ee
rom[12] = 32'b00000000001000100110100000101010; // slt  r13,r1,r2                 1                r13= 1
rom[13] = 32'b11000000001000000011100001100000 0; // sll  r14,r2,#3           7ebb7080             r14= 7ebb7080
rom[14] = 32'b11000000001000000111100101000010; // srl  r15,r1,#5                 0                r15= 0
rom[15] = 32'b11000000110000001000000110000011; // sra  r16,r6,#6             fe000000             r16= fe000000
rom[16] = 32'b00000000001000011100010000100110; // xor  r17,r2,r3             00000000             r17= 00000000
rom[17] = 32'b00000000001000101001000000011000; // mult r17,r1,r2             0fd76e10             r18= 0fd76e10
rom[18] = 32'b00000000001000001100110000011010; // div  r19,r2,r1             0fd76e10             r19= 0fd76e10
```

This test case is used to test independent instructions and to see if they get the correct result. Hence the name "no dependency test," the instructions do not depend on previous results. Additionally, there are no control hazards since there are no branches or jumps. This test helps prove that my ALU operations are working and that I am able to use values from two different registers and get correct values after an instruction is ran. This test also proves that there is no forwarding when there shouldn't be.

Simulation results:



Name	Value
clk	0
reset	0
result[31:0]	9012fd65
[3][31:0]	0fd76e10
[11][31:0]	00000000
imm_value[31:0]	00000024
instr[31:0]	306bff63
alu_result[31:0]	00000020
PC[9:0]	40

306.320 ns
300.000 ns
320.000 ns
340.000 ns
360.000 ns
380.000 ns
400.000 ns
420.000
9012fd65 abc00237 b54bc031 c187a606 0fd76e00 f02891ee 00000001
0fd76e10
00000000 0fd76e00
00000024 ffffff63 00006027 0000682a 000070c0 00007942 ffff8183
306bff63 00226027 0022682a c04070c0 c0207942 c0c08183 00438826
00000020 00000024 0fd76e00 f02891ee 00000001 7ebb7080 00000000
40 44 48 52 56 60 64

I chose a random test case to showcase the 'no-dependency' of my code. Being represented by this simulation is rom[10] or the andi instruction. As one can see from the test instructions, the expected value is 0x0fd76e00. This is because

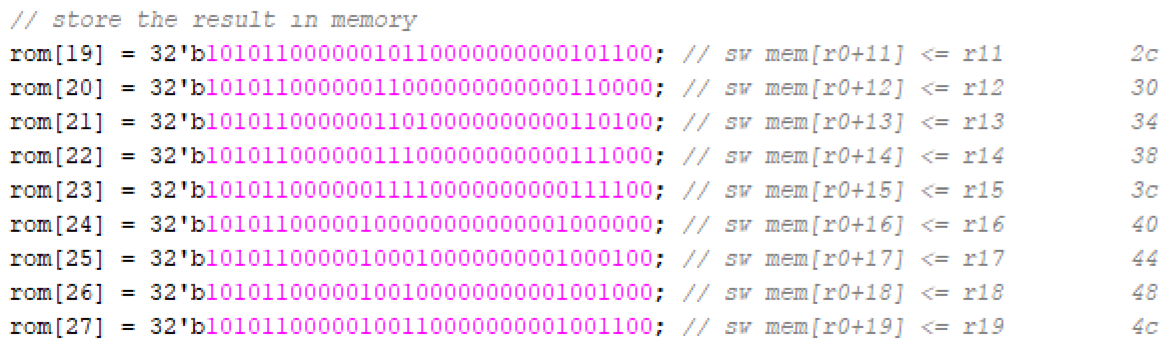
```
  1111 1111 1111 1111 1111 1111 0110 0011   (0xffffff63)
& 0000 1111 1101 0111 0110 1110 0001 0000   (0x0fd76e10)
--------------------------------------------------------------------------------------
  0000 1111 1101 0111 0110 1110 0000 0000   (Result) = (0x0fd76e00).
```

The waveform shows this by first fetching the ANDI instruction, located at rom[10] (equal to 0x306bff63 in hex). This occurs at PC = 40 because the program counter follows the logic: PC = instruction_index * 4.

Next, the instruction is decoded, which is evident as the immediate value (0xffffff63) appears in the imm_value signal. Following this, the ALU performs the AND operation, and the result (0x0fd76e00) appears in the ALU_Result register in the subsequent cycle.

Although the result is calculated at this stage, the value is not immediately visible in r11. This delay occurs because the value is stored into the memory in the next cycle and after that cycle, the register write-back happens on the falling edge of the clock(resulting in 2 cycles passing before the value is in the register). This timing behavior aligns with typical pipeline behavior, where the write-back stage is completed on the next clock cycle, and the updated register value is only observable when the waveform samples data on the falling edge.

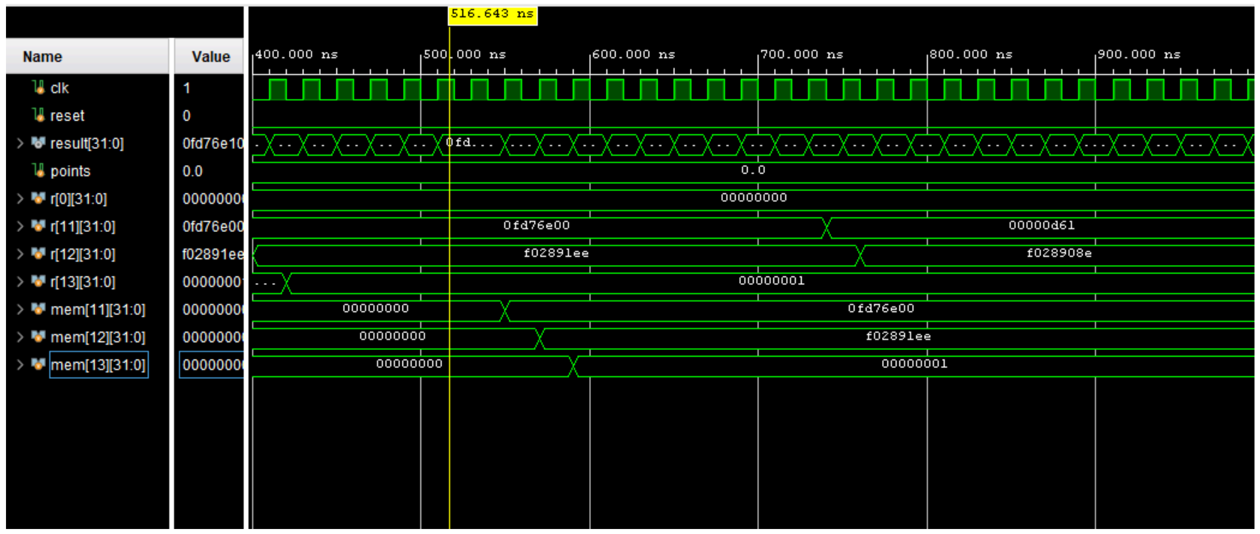
- <u>Test Case 3:</u>



```
// store the result in memory
rom[19] = 32'b10101100000010110000000000101100; // sw mem[r0+11] <= r11        2c
rom[20] = 32'b10101100000011000000000000110000; // sw mem[r0+12] <= r12        30
rom[21] = 32'b10101100000011010000000000110100; // sw mem[r0+13] <= r13        34
rom[22] = 32'b10101100000011100000000000111000; // sw mem[r0+14] <= r14        38
rom[23] = 32'b10101100000011110000000000111100; // sw mem[r0+15] <= r15        3c
rom[24] = 32'b10101100000100000000000001000000; // sw mem[r0+16] <= r16        40
rom[25] = 32'b10101100000100010000000001000100; // sw mem[r0+17] <= r17        44
rom[26] = 32'b10101100000100100000000001001000; // sw mem[r0+18] <= r18        48
rom[27] = 32'b10101100000100110000000001001100; // sw mem[r0+19] <= r19        4c
```

Since the first test case was successful, this test case should follow similar results as to what it shows happens in my pipeline processor. For instance, the instruction is fetched from ROM → it is then decoded to determine what kind of instruction it is → executed by computing the effective memory address by adding r0 to the immediate offset → goes into memory by storing the value from the specified register. However, since it is a store instruction, it does not write back to a register. It instead, updates memory.

Simulation result:



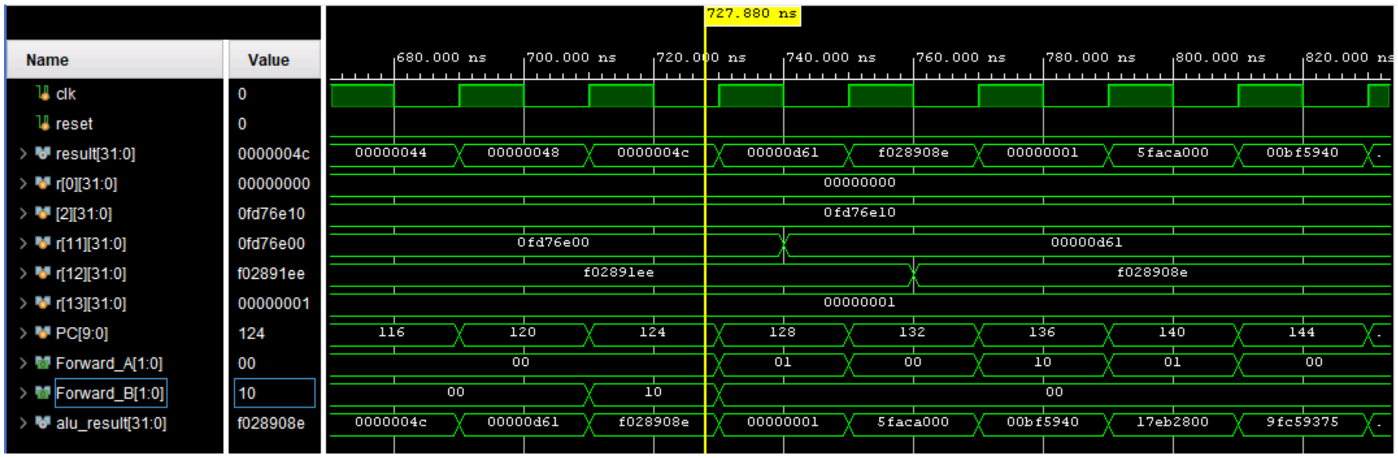
These results are expected as one can see, since r0 = 0, whatever immediate value is added, that register's value will be stored at the same memory location. Therefore, as since r11 = 0x0fd76e00, the memory address of 11 (mem[11]) should reflect the same value. This holds true since in the simulation, mem[11] = 0x0fd76e00.

- Test Case 4:

```
// forwarding test
rom[28] = 32'b00110000111010110000111101100011; // andi r11,r7,#f63          00000d61            r11= 00000d61
rom[29] = 32'b00000000010010110110000000100111; // nor  r12,r2,r11           f028908e            r12= f028908e
rom[30] = 32'b00000001011000100110100000101010; // slt  r13,r11,r2              1                r13=     1
rom[31] = 32'b11000000111000000111001101000000; // sll  r14,r2,#13           5faca000            r14= 5faca000
rom[32] = 32'b11000001110000000111100111000010; // srl  r15,r14,#7           00bf5940â€¬            r15= 00bf5940
rom[33] = 32'b11000001110000000100000010000011; // sra  r16,r14,#2           17eb2800â€¬            r16= 17eb2800â€
rom[34] = 32'b00000000010001111000100000100110; // xor  r17,r2,r7            9fc59375            r17= 9fc59375
rom[35] = 32'b00000000010001111001000000011000; // mult r18,r2,r7            e4e43c50            r18= e4e43c50
rom[36] = 32'b00000000111100011001100000011010; // div  r19,r7,r17              0                r19=     0
```

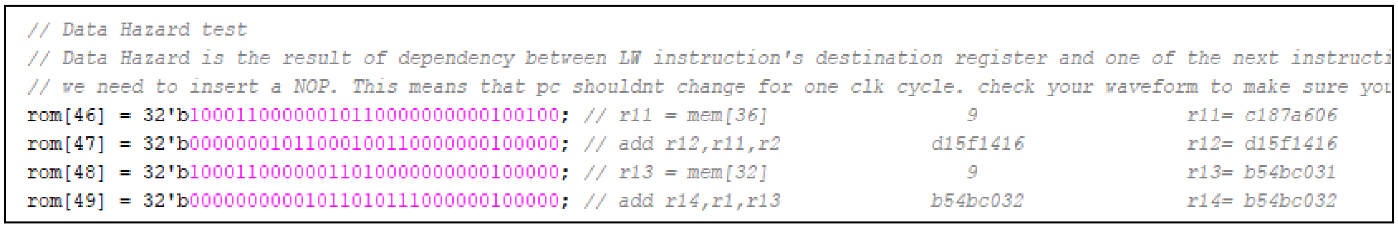
This test case will test if my processor works as expected when instructions depend on values that were just computed in the previous instruction. If my processor does not work, the simulation will show stalls because register values will not be updated by the time the register value is needed.

Simulation Result:



These results prove that my pipeline processor handles forwarding successfully. I chose to use register values 2, 11, 12, and 13 because they are used multiple times in the first few instructions from the given test cases. Additionally, one can see that forwarding is required to get the correct result because the registers from the previous instruction are used in the instruction right after it. From my simulation results, I see that the forwarding case for Forward_B is triggered because it has the value of 2’b10 when the value of r11 is needed in the nor instruction but it is has not been written back to r11 yet. When Forward_B is 2’b10, this means that the value that is going to be used in the next instruction requires the value from the execution stage in the previous instruction and the pipeline needs to forward that value so that the nor can get the correct output. In more detail, between the execution stage and memory stage (in FD(EM)W), the next instruction is fetched and requires the new value of r11, so it is then forwarded from between the execution stage and memory stage to the execution stage of the nor instruction. Therefore, when the nor instruction reaches the execution stage, it will have the inputs of the updated r11 as well as r12 to get the expected output of 0xf028908e.

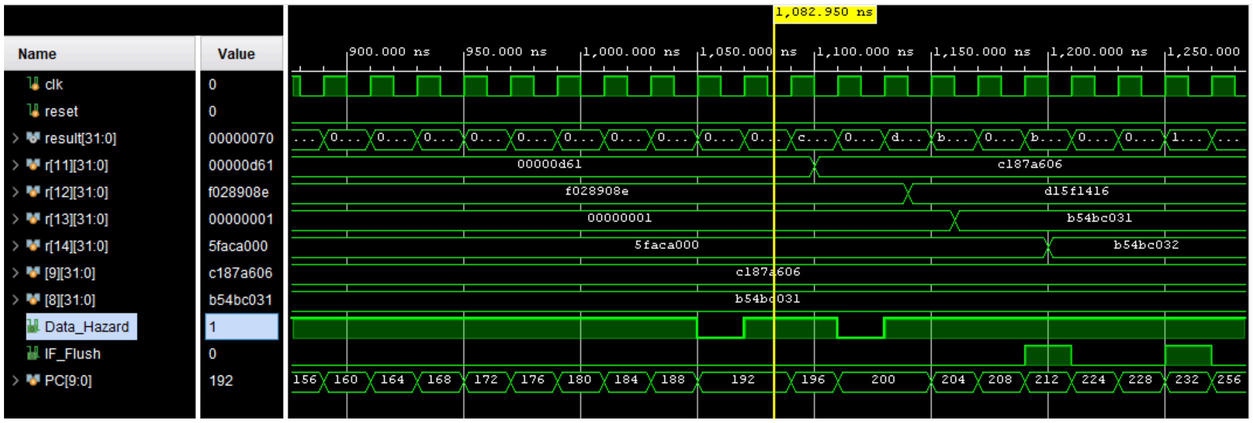
- Test Case 5:



```
// Data Hazard test
// Data Hazard is the result of dependency between LW instruction's destination register and one of the next instructi
// we need to insert a NOP. This means that pc shouldnt change for one clk cycle. check your waveform to make sure you
rom[46] = 32'b10001100000001011000000000100100; // r11 = mem[36]          9           r11= c187a606
rom[47] = 32'b00000001011000100110000000100000; // add r12,r11,r2         d15f1416    r12= d15f1416
rom[48] = 32'b10001100000001101000000000100000; // r13 = mem[32]          9           r13= b54bc031
rom[49] = 32'b00000000001011010111000000100000; // add r14,r1,r13         b54bc032    r14= b54bc032
```

The data hazard test will test if my processor behaves as expected when the pipeline needs to be stalled. There will need to be stalls when a branch instruction is in the decode stage, the next instruction will be in the fetch stage, and if the branch is taken → the program will need to go to the new address and get rid of the instruction that is already in the IF stage. A similar thing will occur during jump instructions since the instruction after jump is already in the IF stage.

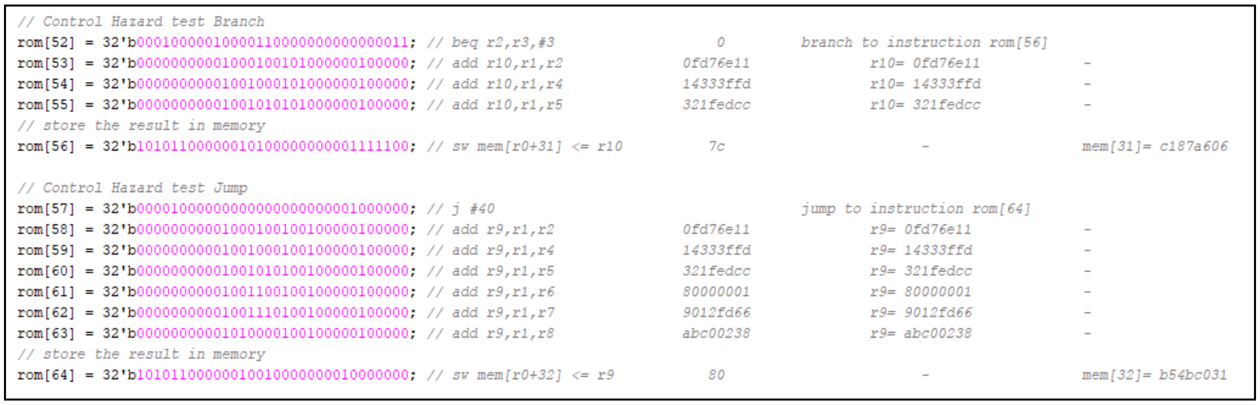
Simulation Results:



In this case, there is going to be stalls because as from the data hazard test instructions, the first one (rom[46]) is a load word instruction. The memory value(mem[36] = actual mem location of 9) is loaded into the register (r11) and the value is not going to be loaded into the register until the write back stage. However, the instruction after (rom[47]) is trying to use r11 as an input in it's addition operation, but since the value is not yet in r11, there is a stall after the decode stage (of rom[47]) until the writeback stage of the previous instruction (rom[46]) so that the correct inputs can be used.

From the simulation, it is clear that after the value is written back to r11 (at around 1100 ns), the value of r12 is not updated until 2 clock cycles after because it is stalled and the execution stage does not commence until after the value in r11 can be used. Additionally, the simulation aligns with the manual, Data_Hazard output is active low and makes the pipeline stall for 1 clock cycle.

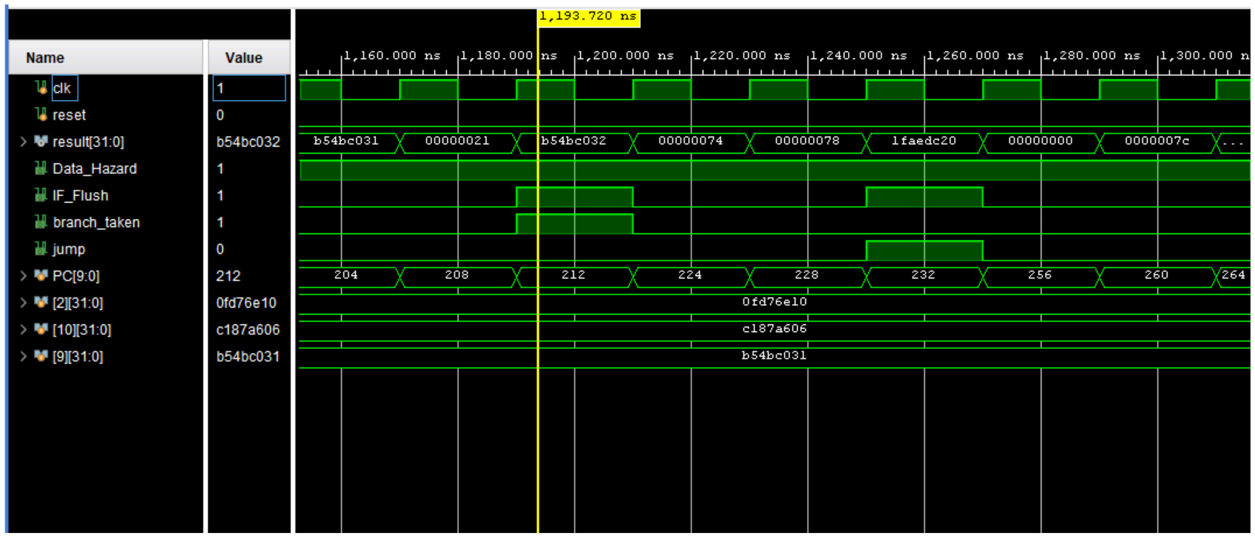
- Test Case 5:


```
// Control Hazard test Branch
rom[52] = 32'b00010000010000110000000000000011; // beq r2,r3,#3                 0         branch to instruction rom[56]
rom[53] = 32'b00000000001000100101000000100000; // add r10,r1,r2         0fd76e11                r10= 0fd76e11             -
rom[54] = 32'b00000000001001000101000000100000; // add r10,r1,r4         14333ffd                r10= 14333ffd             -
rom[55] = 32'b00000000001001010101000000100000; // add r10,r1,r5         321fedcc                r10= 321fedcc             -
// store the result in memory
rom[56] = 32'b10101100000010100000000001111100; // sw mem[r0+31] <= r10        7c                     -                 mem[31]= c187a606

// Control Hazard test Jump
rom[57] = 32'b00001000000000000000000001000000; // j #40                                  jump to instruction rom[64]
rom[58] = 32'b00000000001000100100100000100000; // add r9,r1,r2          0fd76e11                r9= 0fd76e11              -
rom[59] = 32'b00000000001001000100100000100000; // add r9,r1,r4          14333ffd                r9= 14333ffd              -
rom[60] = 32'b00000000001001010100100000100000; // add r9,r1,r5          321fedcc                r9= 321fedcc              -
rom[61] = 32'b00000000001001100100100000100000; // add r9,r1,r6          80000001                r9= 80000001              -
rom[62] = 32'b00000000001001110100100000100000; // add r9,r1,r7          9012fd66                r9= 9012fd66              -
rom[63] = 32'b00000000001010000100100000100000; // add r9,r1,r8          abc00238                r9= abc00238              -
// store the result in memory
rom[64] = 32'b10101100000010010000000010000000; // sw mem[r0+32] <= r9         80                     -                 mem[32]= b54bc031
```

This test case will test if control hazard occurs in my pipeline processor when branch or jump is activated. When this happens, the IF_Flush will make the IF/ID registers zero(via lab manual). More specifically, our flush gets rid of any old instructions so that the branch or jump instructions can change the PC to the correct address.

<u>Simulation results:</u>

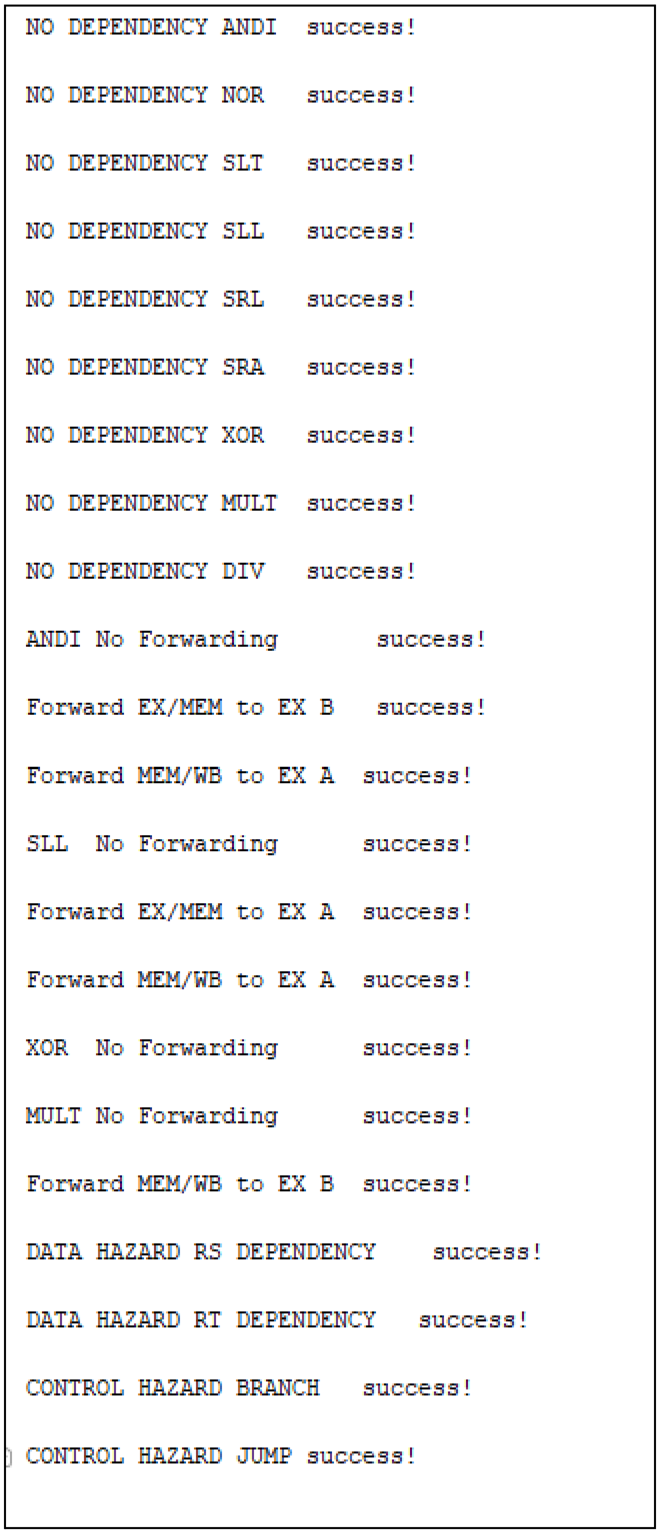


These simulation results are expected. When the flush is triggered, either jump or branch is taken. Additionally, the PC also aligns with the test case because after either the jump or branch, the PC is now at the respective instruction.

If the branch is taken, then program flushes and it goes to PC = 56(from rom[56]) * 4 → 224 which is expected. On the other hand, after the jump instruction is done, the program flushes and the PC is now 256 which is expected since the instruction jumps to rom[64] and PC = 64 * 4 → 256.

Conclusion:

My Pipeline Processor is working as expected and passes all test cases given in the testbench.



```
NO DEPENDENCY ANDI  success!

NO DEPENDENCY NOR   success!

NO DEPENDENCY SLT   success!

NO DEPENDENCY SLL   success!

NO DEPENDENCY SRL   success!

NO DEPENDENCY SRA   success!

NO DEPENDENCY XOR   success!

NO DEPENDENCY MULT  success!

NO DEPENDENCY DIV   success!

ANDI No Forwarding       success!

Forward EX/MEM to EX B   success!

Forward MEM/WB to EX A  success!

SLL  No Forwarding      success!

Forward EX/MEM to EX A  success!

Forward MEM/WB to EX A  success!

XOR  No Forwarding      success!

MULT No Forwarding      success!

Forward MEM/WB to EX B  success!

DATA HAZARD RS DEPENDENCY    success!

DATA HAZARD RT DEPENDENCY   success!

CONTROL HAZARD BRANCH   success!

CONTROL HAZARD JUMP success!
```

Questions:
N/A