CSCE 3301 – Computer Architecture

Project 1: PIPELINED CPU

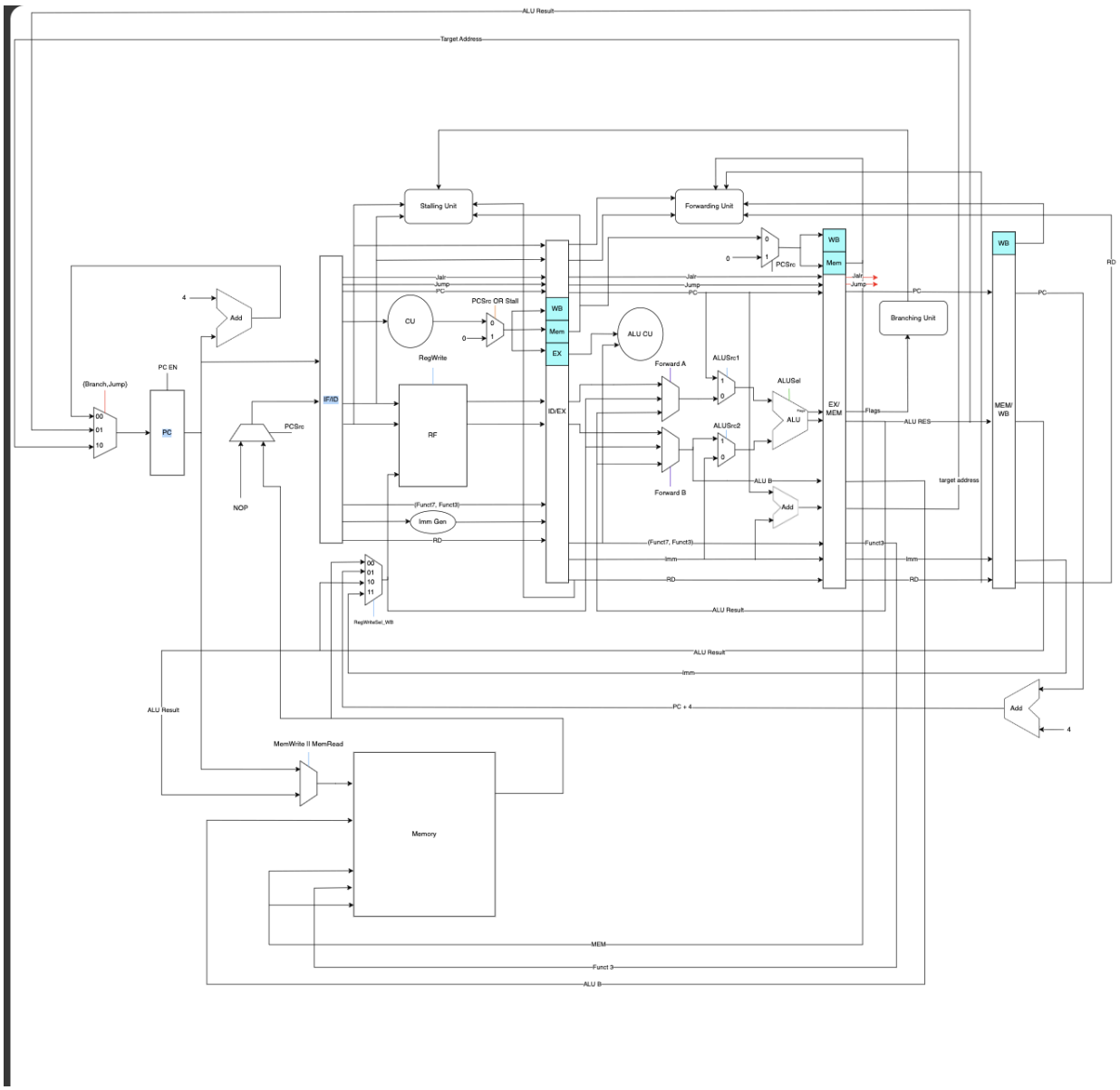
Hussein Heggi - 900213220

Youssef Elmahdy - 900212370

1. **Objective**:

The objective of this project is to develop an RV32-IC pipelined processor that can handle all hazards and have a single memory for data and instructions.

2. **Design**:



3. **Implementation**:

● **Program Counter (PC)**: Implemented as a register that increments by 4 if the PC EN signal is active. It also checks for the BranchAnd signal and the Jump signal to determine whether to add the immediate value from ALU_out to the PC instead of the default increment of 4, facilitating branching or jumping.

● **Single Memory**: This module consists of a straightforward register file with a size of 256. It assumes that instructions precede data in memory. To manage the structural hazard arising from having a single memory unit, a signal named Using_Mem is employed. When Using_Mem is active (indicating the need to read from or write to memory), the necessary signals are selected, and the pipeline is stalled for one cycle. Otherwise, the signals required for reading an instruction are sent, and there's no stall. Additionally, an offset is applied to account for data not starting at address 0.

● **Register File**: This component accepts read and write sources extracted from the instructions obtained from the Memory module. All registers are reset to 0 if the reset signal is active. Otherwise, based on the provided inputs, the module reads the two sources and writes to the destination register (RD) only if the WriteReg signal is active.

● **Arithmetic Logic Unit (ALU)**: The ALU receives ALU_A and ALU_B as inputs, along with ALUSel determining the operation to be performed (arithmetic or logical). It conducts the specified operation on the inputs and sets flags, which are subsequently used by the Branch Control Unit. All flags are generated by subtracting B from A.

● **Control:**

○ Control Unit: Assigns control signals for each instruction based on the opcode's last bits.

○ ALU Control Unit: Determines ALU operation based on func3 of each instruction and ALUOp.

○ Branching Unit: Determines branch conditions based on func3 and flags, assigning BranchAnd accordingly.

○ Forwarding Unit: Determines if forwarding is necessary by comparing source registers from the execute stage with the return register from the write-back stage, as well as write signals. Forwards signals A and B to the ALU based on these comparisons.

○ Stalling Unit: Determines if stalling is necessary by comparing source registers from the decode stage with the return register from the execute stage, along with the memory read signal. Sets the stall signal accordingly, controlling the loading of the IF/ID register, PC increment, and control signals entering the ID/EX register.

○ Flushing Muxes: Handles branch and jump instructions by inserting a NOP instruction into the IF/ID register and zeroing the control signals entering both the ID/EX and EX/MEM registers when necessary.

4. **Difficulties:**

As expected, our primary challenge revolved around implementing the single memory unit. We encountered issues where certain signals weren't working properly, leading to a point where our store instructions stopped to function properly. However, through extensive testing, we opted to address the structural hazard by introducing pipeline stalls whenever memory access was necessary.

Initially, we faced difficulties in tracking our pipeline during testing. However, with time and experience, we gradually became more adept at identifying the origin of each signal.

The complexity of the datapath posed another significant hurdle, necessitating the omission of numerous signals due to space constraints.

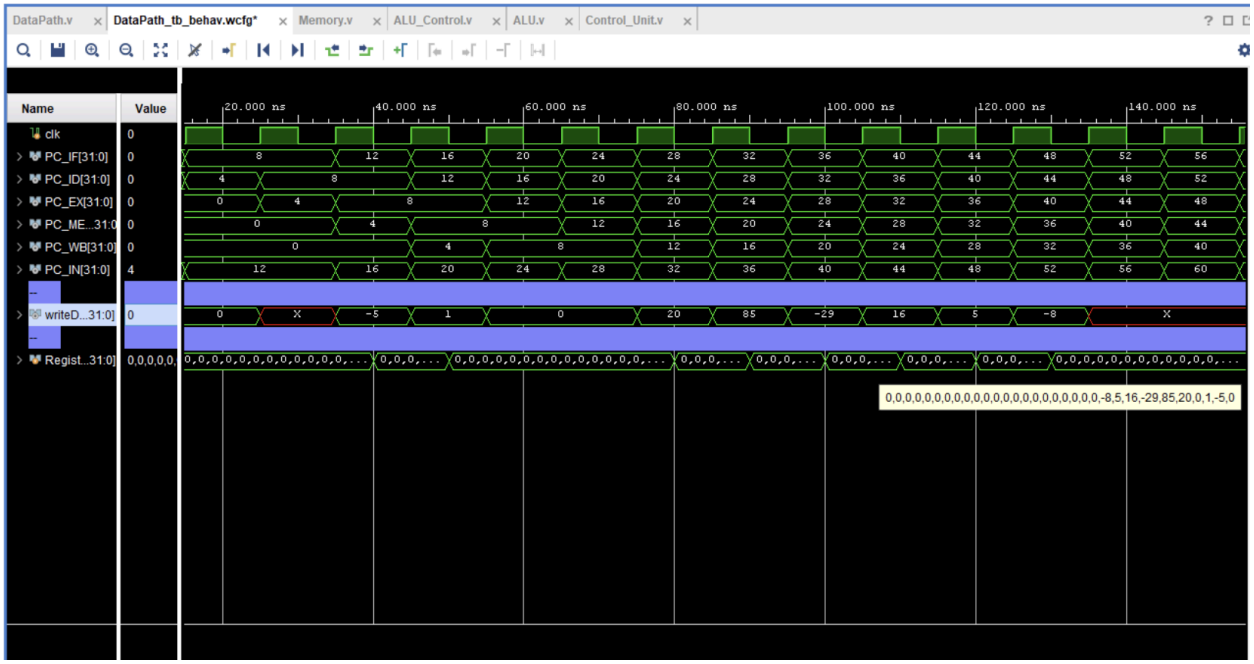
Time management emerged as a major obstacle for us, as debugging consistently consumed more time than anticipated.

5. **Testing:**

We split the testing of the 40 instructions on a couple of test programs, each test program testing a few different instructions. We did 6 test programs in total three of which were programs that we wrote simply to test whether the instructions work or not rather than being an actual function which does a certain operation and one test program was a function that writes an array in memory and swaps it.

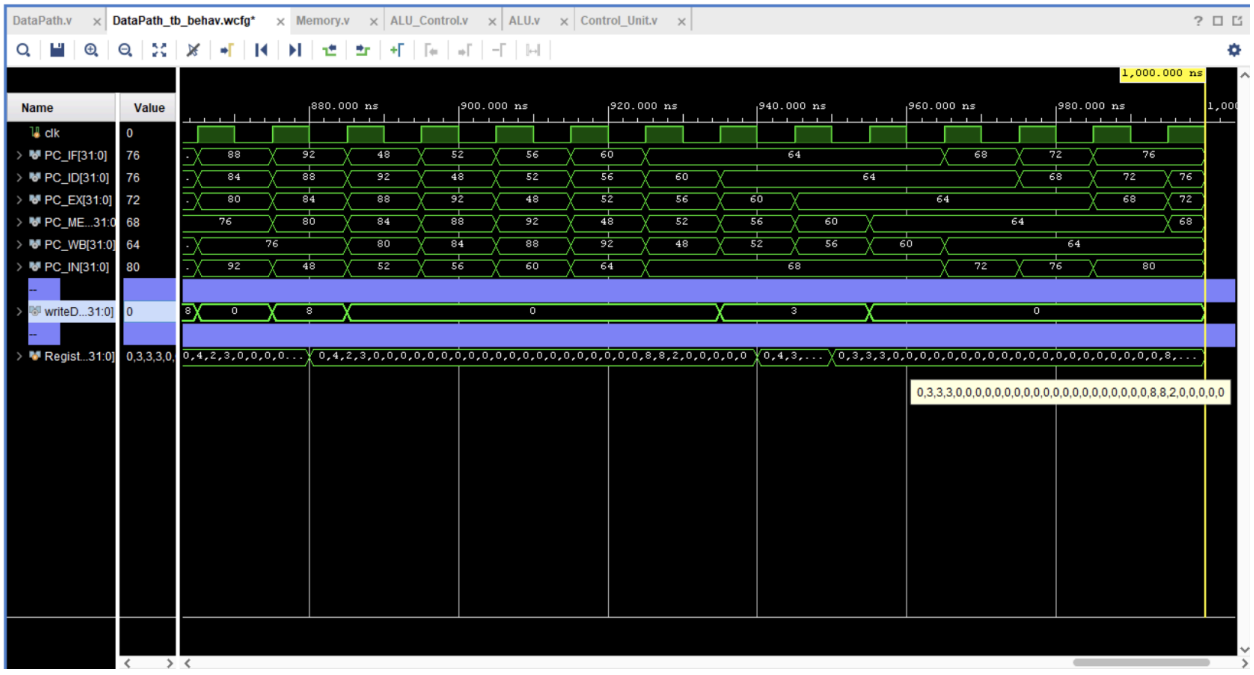
Test1:

```
//    test case 1
        {mem[3], mem[2], mem[1], mem[0]} = 32'b11111111101100000000000010010011; //addi x1, x0, -5
        {mem[7], mem[6], mem[5], mem[4]} = 32'b00000000000110000101000100010011; //slti x2, x1, 3
        {mem[11], mem[10], mem[9], mem[8]} = 32'b00000000000110000101100011010011; //sltiu x3, x1, 3
        {mem[15], mem[14], mem[13], mem[12]} = 32'b00000001010000011100001000010011; //xori x4, x3, 20
        {mem[19], mem[18], mem[17], mem[16]} = 32'b00000101010100100110001010010011; //ori x5, x4, 85
        {mem[23], mem[22], mem[21], mem[20]} = 32'b11111110011100001111001100010011; //andi x6, x1, -25
        {mem[27], mem[26], mem[25], mem[24]} = 32'b00000000010000010001001110010011; //slli x7, x2, 4
        {mem[31], mem[30], mem[29], mem[28]} = 32'b00000000000100010010101000010011; //srli x8, x4, 2
        {mem[35], mem[34], mem[33], mem[32]} = 32'b01000000000100011010101010010011; //srai x9, x6, 2
```



Test2:

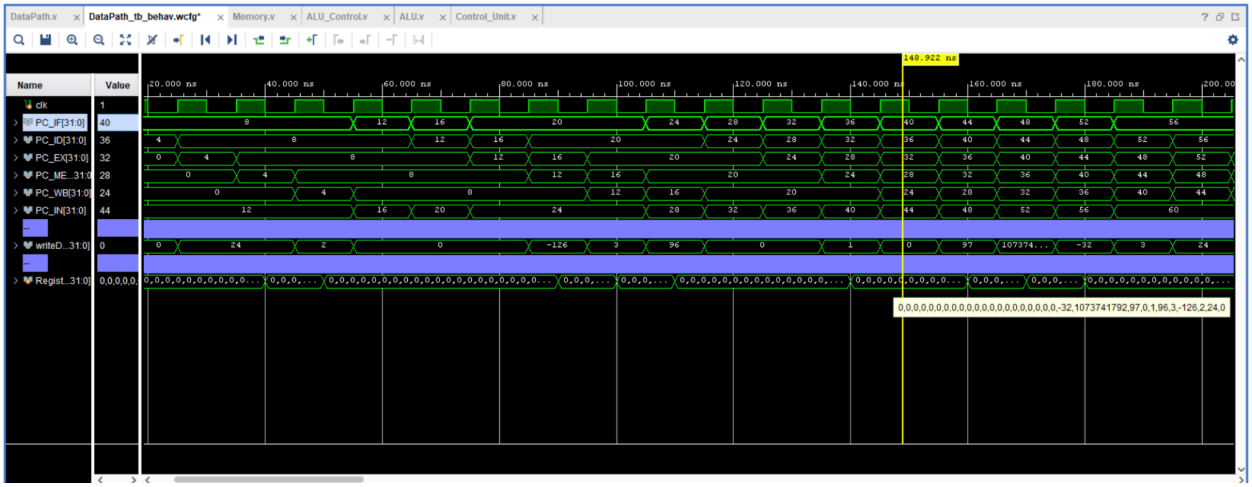
```
//test case 2
{mem[3], mem[2], mem[1], mem[0]} = 32'b00000000000010000000000001010010011; //addi x5, x0, 1 # x = 1
{mem[7], mem[6], mem[5], mem[4]} = 32'b00000000000000000000000001100010011; //addi x6, x0, 0 # i = 0
{mem[11], mem[10], mem[9], mem[8]} = 32'b00000000011000000000000001110010011; //addi x7, x0, 6 # y = 6
{mem[15], mem[14], mem[13], mem[12]} = 32'b00000000011100101000101001100011; //beq x5, x7, endStore # i == y?

{mem[19], mem[18], mem[17], mem[16]} = 32'b00000000010100011001000000100011; //sw x5, 0(x6)
{mem[23], mem[22], mem[21], mem[20]} = 32'b00000000010000110000001100010011; //addi x6, x6, 4
{mem[27], mem[26], mem[25], mem[24]} = 32'b00000000000010010100000101010011; //addi x5, x5, 1
{mem[31], mem[30], mem[29], mem[28]} = 32'b11111110000000000000100011100011; //beq x0, x0, store # loop back

{mem[35], mem[34], mem[33], mem[32]} = 32'b00000000000000000000000001010010011; //addi x5, x0, 0 # i = 0
{mem[39], mem[38], mem[37], mem[36]} = 32'b00000000000000000000000001100010011; //addi x6, x0, 0 # *i

{mem[43], mem[42], mem[41], mem[40]} = 32'b00000001000000000000000001110010011; //addi x7, x0, 16 # *(4-i)
{mem[47], mem[46], mem[45], mem[44]} = 32'b00000000000110000000011100010011; //addi x8, x0, 3 # x = 3
{mem[51], mem[50], mem[49], mem[48]} = 32'b00000011110000101000001001100011; //beq x5, x8, endSwap # i == x?
{mem[55], mem[54], mem[53], mem[52]} = 32'b00000000000000011001011101000011; //lw x9, 0(x6) # temp1 = mem[i]
{mem[59], mem[58], mem[57], mem[56]} = 32'b00000000000000011101011110000011; //lw x10, 0(x7) # temp2 = mem[4-i]
{mem[63], mem[62], mem[61], mem[60]} = 32'b00000001110100111010000000100011; //sw x9, 0(x7) # mem[4-i] = temp1
{mem[67], mem[66], mem[65], mem[64]} = 32'b00000001111000110010000000100011; //sw x10, 0(x6) # mem[i] = temp2

{mem[71], mem[70], mem[69], mem[68]} = 32'b00000000000010010100000101010011; //addi x5, x5, 1 # i++
{mem[75], mem[74], mem[73], mem[72]} = 32'b00000000001000011000000110010011; //addi x6, x6, 4 # *(i+1)
{mem[79], mem[78], mem[77], mem[76]} = 32'b11111111111000011100000111010011; //addi x7, x7, -4 # *(4-i-1)
{mem[83], mem[82], mem[81], mem[80]} = 32'b11111110000000000000000011100011; //beq x0, x0, swap # loop back
```



Test3:

```
//Test case 3
    {mem[3], mem[2], mem[1], mem[0]} = 32'b00000000000000000100000010000011; // lbu x1, 0(x0) #x1 = 24
    {mem[7], mem[6], mem[5], mem[4]} = 32'b00000000010000000001000100000011; // lh x2, 4(x0) #x2 = 2
    {mem[11], mem[10], mem[9], mem[8]} = 32'b00000000100000000000000110000011; // lb x3, 8(x0) #x3 = -126
    {mem[15], mem[14], mem[13], mem[12]} = 32'b00000000110000000101001000000011; // lhu x4, 12(x0) #x4 = 3
    {mem[19], mem[18], mem[17], mem[16]} =32'b00000000000100000100100101010110011; // sll x5, x1, x2 #x5 = 96
    {mem[23], mem[22], mem[21], mem[20]} = 32'b00000000000010001101000110010011; // slt x6, x3, x1 #x6 = 1
    {mem[27], mem[26], mem[25], mem[24]} = 32'b00000000000010001101100111010011; // sltu x7, x3, x1 #x7 = 0
    {mem[31], mem[30], mem[29], mem[28]} = 32'b00000000010100110100010000110011; // xor x8, x6, x5 #x8 = 97
    {mem[35], mem[34], mem[33], mem[32]} = 32'b00000000000100001110101001010011; // srl x9, x3, x2 #x9 = 1073741792
    {mem[39], mem[38], mem[37], mem[36]} = 32'b01000000001000011101010100110011; // sra x10,x3, x2 #x10 = -32
```

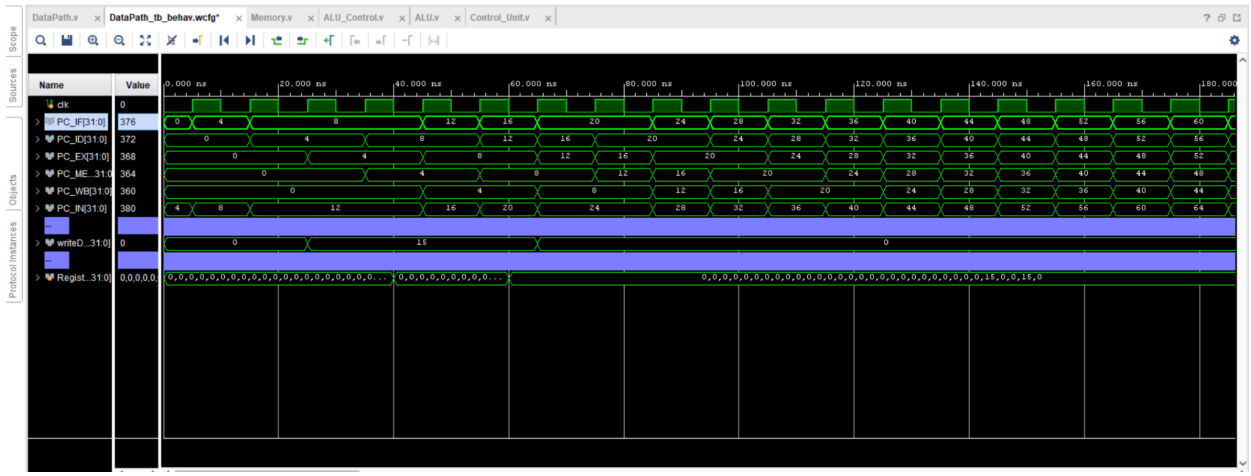
```
  {mem[43], mem[42], mem[41], mem [40]} = 32'b00000000000000000000000000011000; // 24
  {mem[47], mem[46], mem[45], mem[44]} = 32'b00000000000000000000000000000010; // 2
  {mem[51], mem[50], mem[49], mem[48]} = 32'b00000000000000000000000010000010; // 66
  {mem[55], mem[54], mem[53], mem[52]} = 32'b00000000000000000000000000000011; // 3
```



Test4:

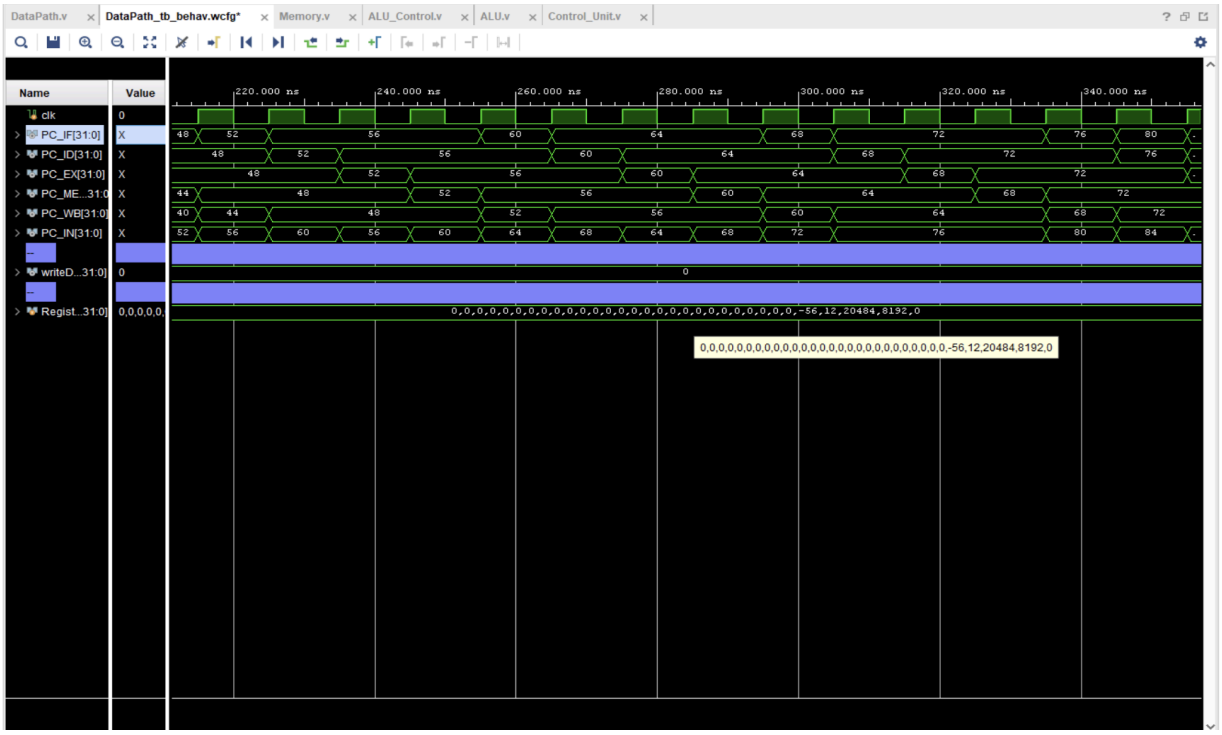
```
//Test case 4
    {mem[3], mem[2], mem[1], mem[0]} = 32'b000000000000_00000_010_00001_0000011; // lw x1, 0(x0)
    {mem[7], mem[6], mem[5], mem[4]} = 32'b0000000_00000_00001_110_00100_0110011; // or x4, x1, x0
    {mem[11], mem[10], mem[9], mem[8]} = 32'b0000000_00100_00000_010_01100_0100011; // sw x4, 12(x0)
```

```
  {mem[15], mem[14], mem[13], mem[12]} = 32'd15;
```



Test5:

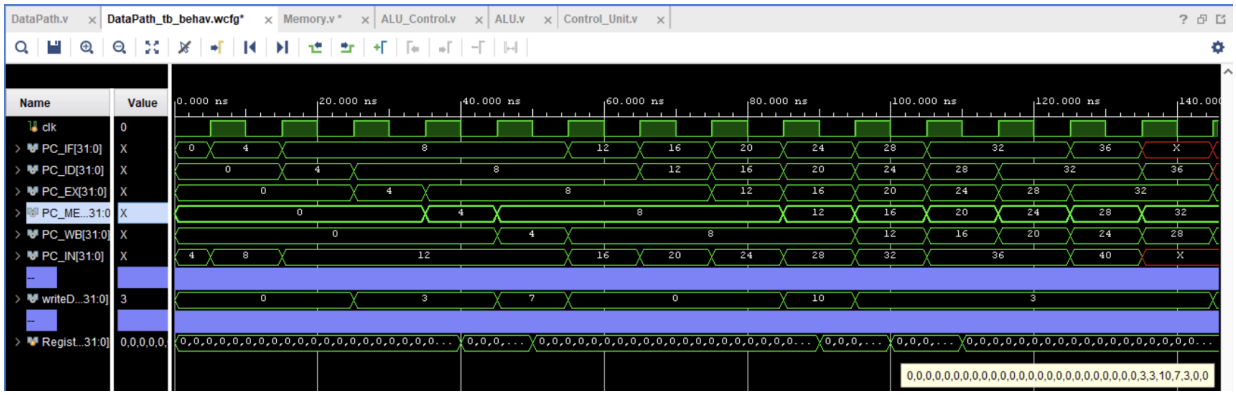
```
//Test case 5
    {mem[3], mem[2], mem[1], mem[0]} = 32'b00000000000000000010000010110111; // lui x1, 2
    {mem[7], mem[6], mem[5], mem[4]} = 32'b00000000000000000101000100010111; // auipc x2, 5
    {mem[11], mem[10], mem[9], mem[8]} = 32'b00000000100000000000000111101111; // jal x3, 8

    {mem[15], mem[14], mem[13], mem[12]} = 32'b00000000000100000000000001110011; // ebreak
    {mem[19], mem[18], mem[17], mem[16]} = 32'b00000000001000001001010001100011; // bne x1, x2, 8

    {mem[23], mem[22], mem[21], mem[20]} = 32'b00000000000000000000000001110011; // ecall
    {mem[27], mem[26], mem[25], mem[24]} = 32'b11111100100000000000001000010011; // addi x4, x0, -56
    {mem[31], mem[30], mem[29], mem[28]} = 32'b00000000001000001100010001100011; // blt x1, x2, 8

    {mem[35], mem[34], mem[33], mem[32]} = 32'b00000000000000000000000001110011; // ecall
    {mem[39], mem[38], mem[37], mem[36]} = 32'b00000000001000000000000000100011; // sb x2, 0(x0)
    {mem[43], mem[42], mem[41], mem[40]} = 32'b00000000000010001010101000001100011; // bge x2, x1, 8

    {mem[47], mem[46], mem[45], mem[44]} = 32'b00000000000000000000000001110011; // ecall
    {mem[51], mem[50], mem[49], mem[48]} = 32'b00000000001000001110010001100011; // bltu x1, x4, 8

    {mem[55], mem[54], mem[53], mem[52]} = 32'b00000000000000000000000001110011; // ecall
    {mem[59], mem[58], mem[57], mem[56]} = 32'b00000000001000100111010001100011; // bgeu x4, x2, 8

    {mem[63], mem[62], mem[61], mem[60]} = 32'b00000000000000000000000001110011; // ecall
    {mem[67], mem[66], mem[65], mem[64]} = 32'b00000000001000000001000010100011; // sh x4, 1(x0)
    {mem[71], mem[70], mem[69], mem[68]} = 32'b00000000000000000000000001110011; // ecall
    {mem[75], mem[74], mem[73], mem[72]} = 32'b00001111111110000000000000001111; // fence
    {mem[79], mem[78], mem[77], mem[76]} = 32'b00000000000000001100000001100111; //  jalr x0, x3, 0
```



Test6:

```
//Test case 6
    {mem[3], mem[2], mem[1], mem[0]} = 32'b00000000000000000010000100000011; // lw x2, 0(x0)
    {mem[7], mem[6], mem[5], mem[4]} = 32'b00000000001000000001000110000011; // lw x3, 4(x0)
    {mem[11], mem[10], mem[9], mem[8]} = 32'b00000000001100010000001000110011; // add x4, x2, x3
    {mem[15], mem[14], mem[13], mem[12]} = 32'b01000000001100010000000101010110011; // sub x5, x4, x3
    {mem[19], mem[18], mem[17], mem[16]} = 32'b00000000001100010111001100110011; // and x6, x2, x3
```

```
{mem[23], mem[22], mem[21], mem[20]} = 32'b00000000000000000000000000000011; // 3
{mem[27], mem[26], mem[25], mem[24]} = 32'b00000000000000000000000000000111; // 7
```



7. Bonus:

- Suggest and implement a different solution to handle the structural hazard introduced by the single memory requirement => we stall once we detect an active MemRead or MemWrite signal.
- Provide a test program generator => we wrote a python program that generates the assembly code along with its machine code.