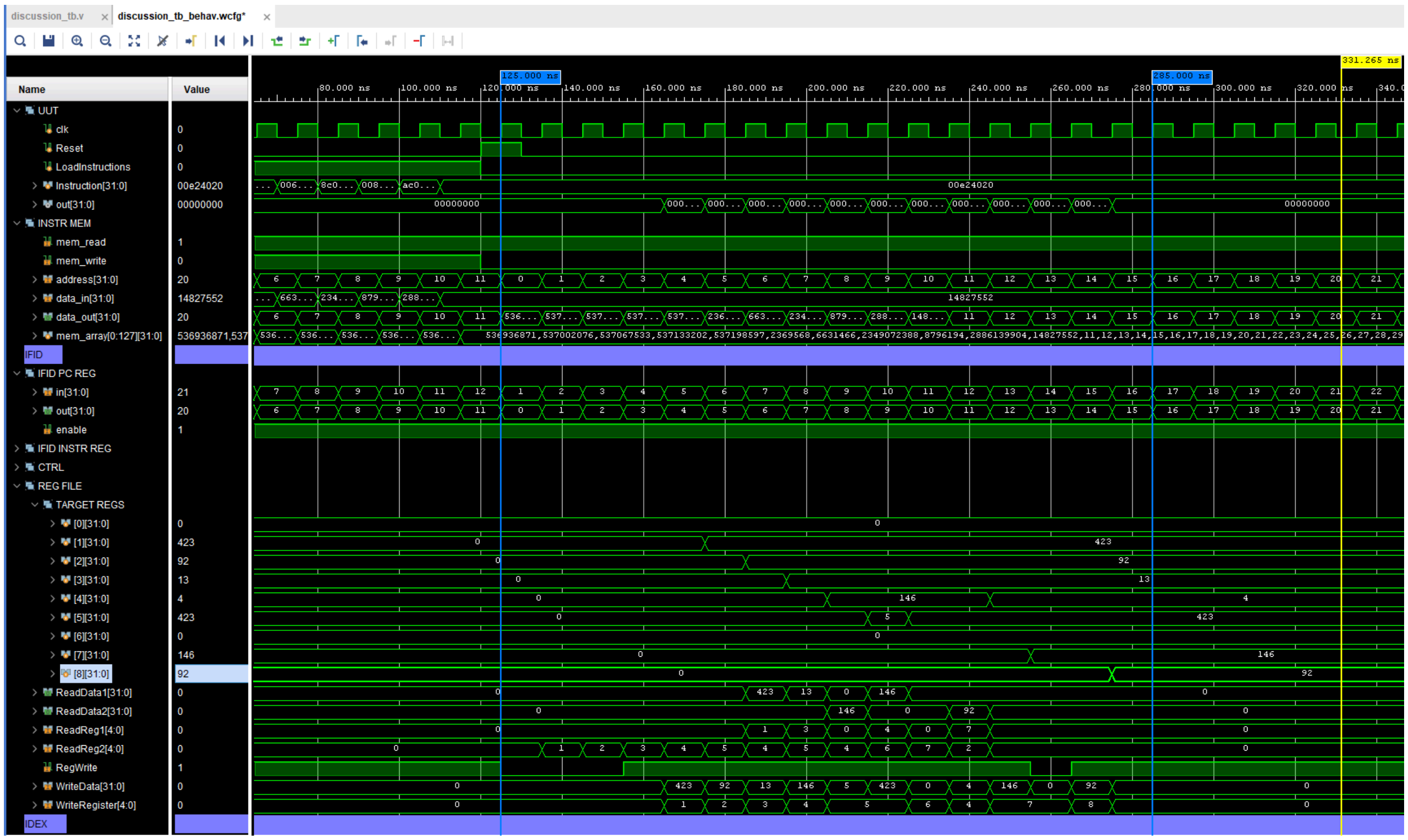
Jilin Zheng | EC413 Lab 7 Report

Task 1/Prelab: Synthesize given files, generate outputs for given testbench



The region in between the blue markers in the simulation waveform is the duration of the instructions being executed in the provided testbench. As can be seen, incorrect data populates the WriteData register and the register file itself, due to data not being properly forwarded. Also notice that since this is a five-stage pipeline, it takes five clock cycles for the first result to show up in the WriteData register.

Tasks 2-6:

I combined my implementation and completion of tasks 2-6, so I will only include the code, simulation waveform, and modified hardware diagram once to avoid repetition, but I will detail each task separately. All line numbers in the task descriptions are references to my Forwarding Module, on the page after the next page.

- **Task 2: Add "1 ahead" forwarding:**
    - In my Forwarding Unit module, the first conditional for both ForwardA and ForwardB signals implement 1-ahead forwarding. Specifically, this would be line 16 for ForwardA and line 35 for ForwardB.
- **Task 3: Add "2 ahead" forwarding:**
    - In my Forwarding Unit module, the first conditional for both ForwardA and ForwardB signals implement 1-ahead forwarding. Specifically, this would be line 22 for ForwardA and line 41 for ForwardB.
- **Task 4: Add arbitration logic for deciding between "`1 ahead" or "2 ahead":**
    - The arbitration logic to decide between 1-ahead and 2-ahead is mixed into the conditionals I've already mentioned for the 1-ahead and 2-ahead forwarding. By checking for the 1-ahead case first, I am able to discard the 2-ahead case, if the 1-ahead conditions are met, which is desired since the 1-ahead case has the most recent data. Then, if the 1-ahead conditions are not met, but

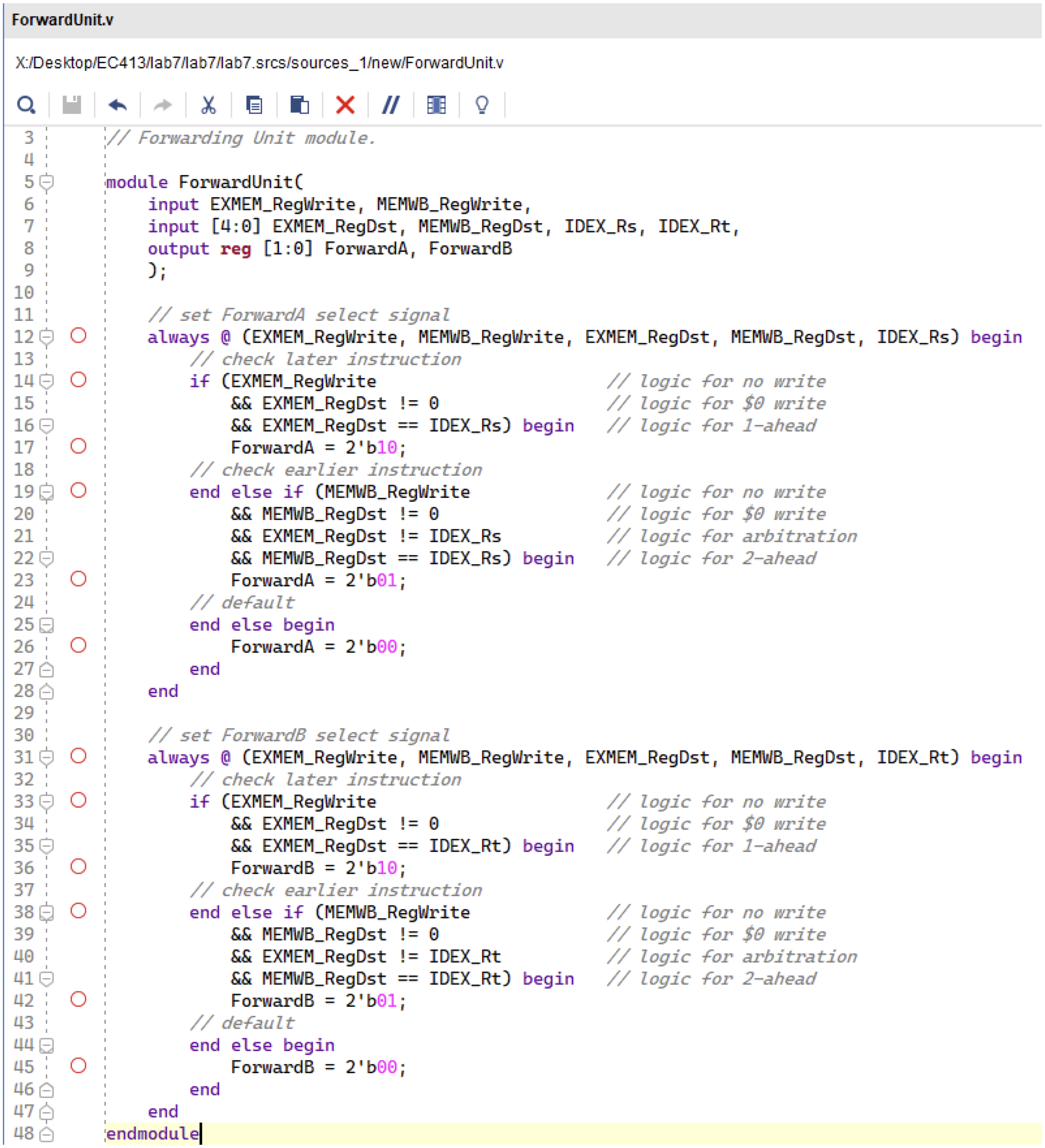
the 2-ahead conditions are met, the 2-ahead data can be forwarded appropriately. The relevant lines of code would be lines 14-22 for the ForwardA signal and lines 33-41 for the ForwardB signal.

- **Task 5: Add logic for $0 write:**
    - Logic for a $0 write is simple: since register $0 is always zero, nothing will be forwarded for the zero.
- **Task 6: Add logic for No Write:**
    - No Write is also simple: by looking at the RegWrite control signal, I can simply choose to forward only if the RegWrite control signal is on/high/1. Consequently, if RegWrite is off/low/0, no data will be forwarded.

As for modifications in the actual pipelined CPU, I added two wires, one between the forwarding mux and the first ALU input, and the other between the register/immediate mux and the forwarding mux for the second ALU input (reminder that the second input can either be a register or immediate for R- and I-type instructions, respectively, and in the case of the I-type instruction, no data is forwarded for the second input). I also added two 3-1 forwarding muxes, as you may have guessed, for the two ALU inputs. And of course, I also added the actual forwarding unit. The relevant lines of code would be lines 158-173 and lines 190-200 on the page after the next page, where I showcase my modified CPU EX stage.

On the page after that, I have included my simulation waveform using the given ‘discussion_tb.v’ testbench (also included on subsequent page). And finally on the page after the simulation waveform is my modified hardware diagram, where I have drawn the hardware I added and described above.

Forwarding Unit Module:

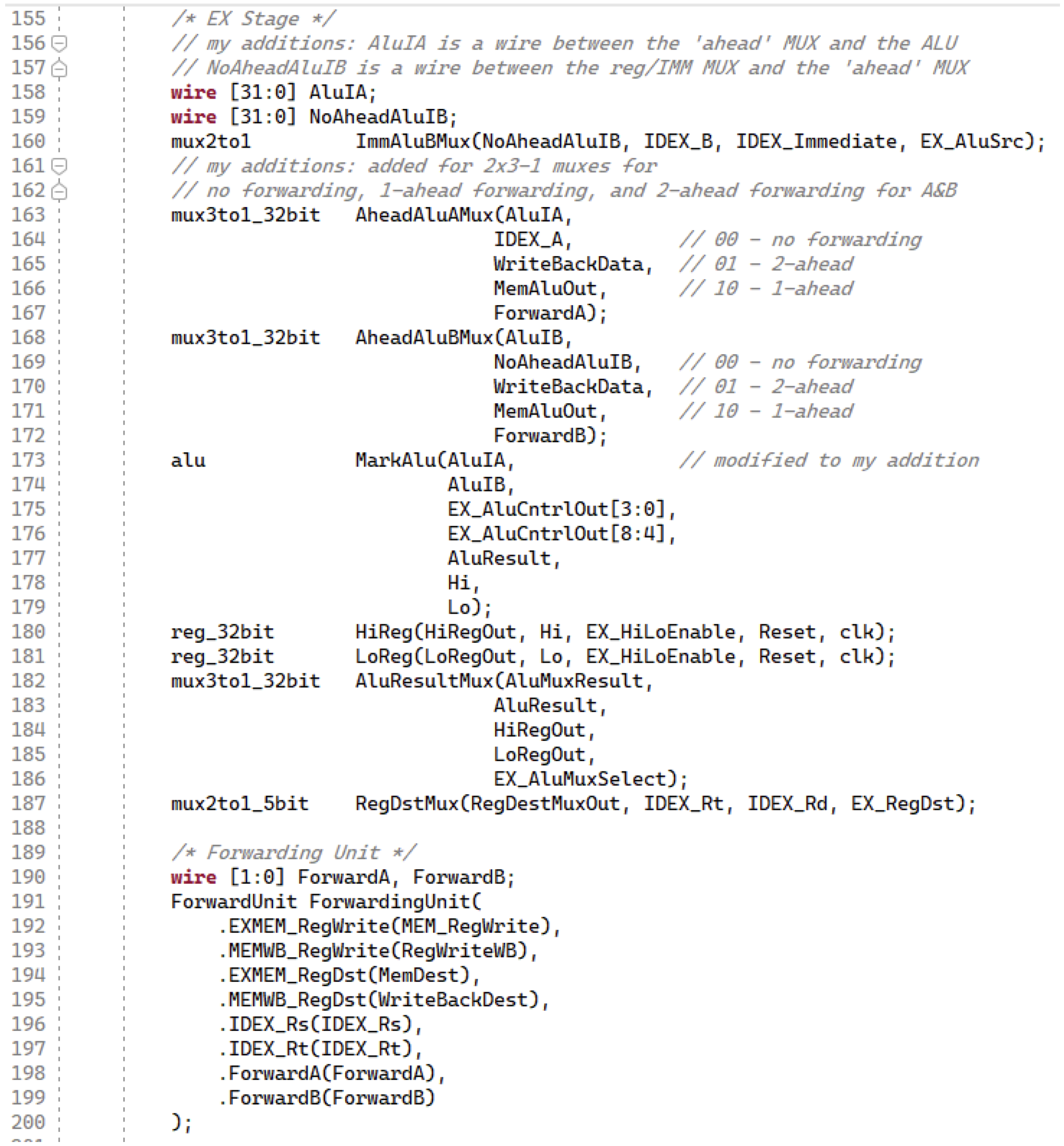


ForwardUnit.v

X:/Desktop/EC413/lab7/lab7/lab7.srcs/sources_1/new/ForwardUnit.v

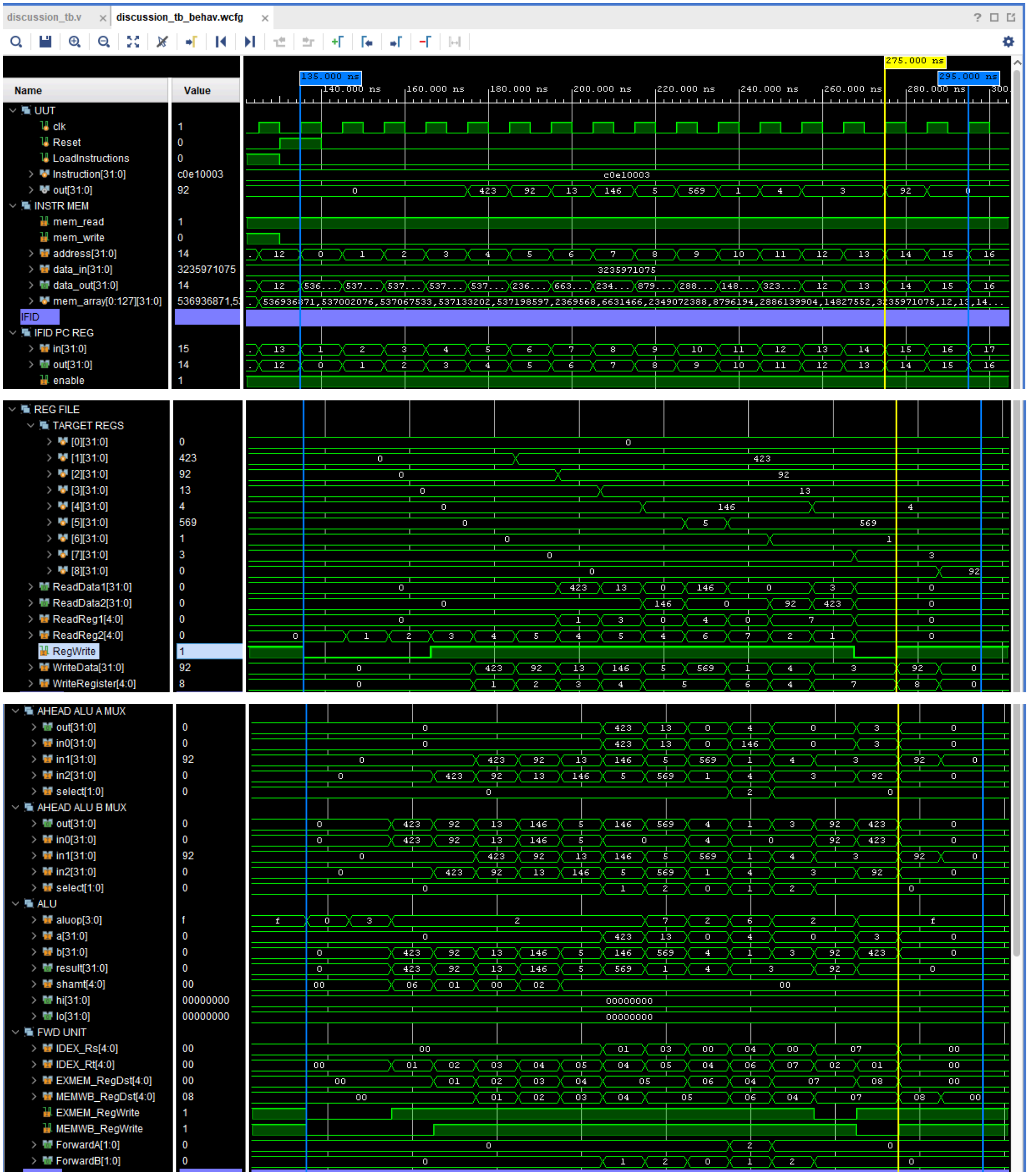
```verilog
// Forwarding Unit module.

module ForwardUnit(
    input EXMEM_RegWrite, MEMWB_RegWrite,
    input [4:0] EXMEM_RegDst, MEMWB_RegDst, IDEX_Rs, IDEX_Rt,
    output reg [1:0] ForwardA, ForwardB
    );

    // set ForwardA select signal
    always @ (EXMEM_RegWrite, MEMWB_RegWrite, EXMEM_RegDst, MEMWB_RegDst, IDEX_Rs) begin
        // check later instruction
        if (EXMEM_RegWrite                      // logic for no write
            && EXMEM_RegDst != 0                // logic for $0 write
            && EXMEM_RegDst == IDEX_Rs) begin   // logic for 1-ahead
            ForwardA = 2'b10;
        // check earlier instruction
        end else if (MEMWB_RegWrite             // logic for no write
            && MEMWB_RegDst != 0                // logic for $0 write
            && EXMEM_RegDst != IDEX_Rs          // logic for arbitration
            && MEMWB_RegDst == IDEX_Rs) begin   // logic for 2-ahead
            ForwardA = 2'b01;
        // default
        end else begin
            ForwardA = 2'b00;
        end
    end

    // set ForwardB select signal
    always @ (EXMEM_RegWrite, MEMWB_RegWrite, EXMEM_RegDst, MEMWB_RegDst, IDEX_Rt) begin
        // check later instruction
        if (EXMEM_RegWrite                      // logic for no write
            && EXMEM_RegDst != 0                // logic for $0 write
            && EXMEM_RegDst == IDEX_Rt) begin   // logic for 1-ahead
            ForwardB = 2'b10;
        // check earlier instruction
        end else if (MEMWB_RegWrite             // logic for no write
            && MEMWB_RegDst != 0                // logic for $0 write
            && EXMEM_RegDst != IDEX_Rt          // logic for arbitration
            && MEMWB_RegDst == IDEX_Rt) begin   // logic for 2-ahead
            ForwardB = 2'b01;
        // default
        end else begin
            ForwardB = 2'b00;
        end
    end
endmodule
```

Modified CPU EX stage, including new Forwarding Unit:



```verilog
/* EX Stage */
// my additions: AluIA is a wire between the 'ahead' MUX and the ALU
// NoAheadAluIB is a wire between the reg/IMM MUX and the 'ahead' MUX
wire [31:0] AluIA;
wire [31:0] NoAheadAluIB;
mux2to1         ImmAluBMux(NoAheadAluIB, IDEX_B, IDEX_Immediate, EX_AluSrc);
// my additions: added for 2x3-1 muxes for
// no forwarding, 1-ahead forwarding, and 2-ahead forwarding for A&B
mux3to1_32bit   AheadAluAMux(AluIA,
                            IDEX_A,         // 00 - no forwarding
                            WriteBackData,  // 01 - 2-ahead
                            MemAluOut,      // 10 - 1-ahead
                            ForwardA);
mux3to1_32bit   AheadAluBMux(AluIB,
                            NoAheadAluIB,   // 00 - no forwarding
                            WriteBackData,  // 01 - 2-ahead
                            MemAluOut,      // 10 - 1-ahead
                            ForwardB);
alu             MarkAlu(AluIA,              // modified to my addition
                        AluIB,
                        EX_AluCntrlOut[3:0],
                        EX_AluCntrlOut[8:4],
                        AluResult,
                        Hi,
                        Lo);
reg_32bit       HiReg(HiRegOut, Hi, EX_HiLoEnable, Reset, clk);
reg_32bit       LoReg(LoRegOut, Lo, EX_HiLoEnable, Reset, clk);
mux3to1_32bit   AluResultMux(AluMuxResult,
                            AluResult,
                            HiRegOut,
                            LoRegOut,
                            EX_AluMuxSelect);
mux2to1_5bit    RegDstMux(RegDestMuxOut, IDEX_Rt, IDEX_Rd, EX_RegDst);

/* Forwarding Unit */
wire [1:0] ForwardA, ForwardB;
ForwardUnit ForwardingUnit(
    .EXMEM_RegWrite(MEM_RegWrite),
    .MEMWB_RegWrite(RegWriteWB),
    .EXMEM_RegDst(MemDest),
    .MEMWB_RegDst(WriteBackDest),
    .IDEX_Rs(IDEX_Rs),
    .IDEX_Rt(IDEX_Rt),
    .ForwardA(ForwardA),
    .ForwardB(ForwardB)
);
```

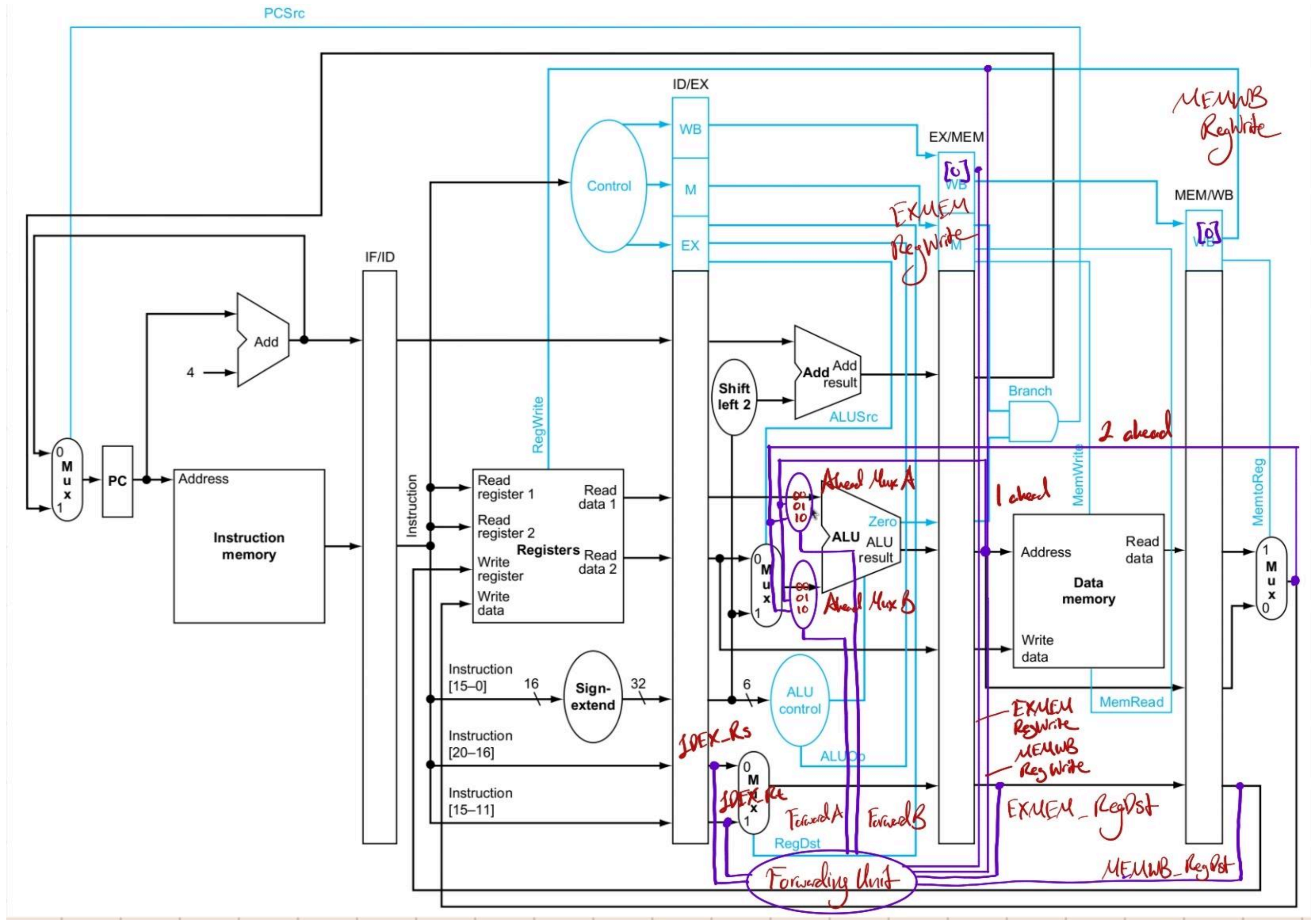
Simulation waveform after modifying pipelined CPU (tasks 2-6):



Testbench used to produce previous simulation waveform (provided with assignment, with my slightly modified comments):

```
Instruction = 32'b001000_00000_00001_0000000110100111;      // addi $R1, $0, 423
#10; // 1                                                   // -> 423
Instruction = 32'b001000_00000_00010_0000000001011100;      // addi $R2, $0, 92
#10; // 2                                                   // -> 92
Instruction = 32'b001000_00000_00011_0000000000001101;      // addi $R3, $0, 13
#10; // 3                                                   // -> 13
Instruction = 32'b001000_00000_00100_0000000010010010;      // addi $R4, $0, 146
#10; // 4                                                   // -> 146
Instruction = 32'b001000_00000_00101_0000000000000101;      // addi $R5, $0, 5
#10; // 5                                                   // -> 5
Instruction = 32'b000000_00001_00100_00101_00000_100000;    // add $R5, $R1, $R4
#10; // 6                                                   // -> 569 (423 if wrong, 2-ahead)
Instruction = 32'b000000_00011_00101_00110_00000_101010;    // slt $R6, $R3, $R5
#10; // 7                                                   // -> 1 (0 if wrong, 1-ahead)
Instruction = 32'b100011_00000_00100_0000000000000100;      // lw $R4, 4(R0)
#10; // 8                                                   // -> 4
Instruction = 32'b000000_00100_00110_00111_00000_100010;    // sub $R7, $R4, $R6
#10; // 9                                                   // -> 3 (146 or 145 if wrong, 1-ahead & 2-ahead)
```

Modified hardware diagram:

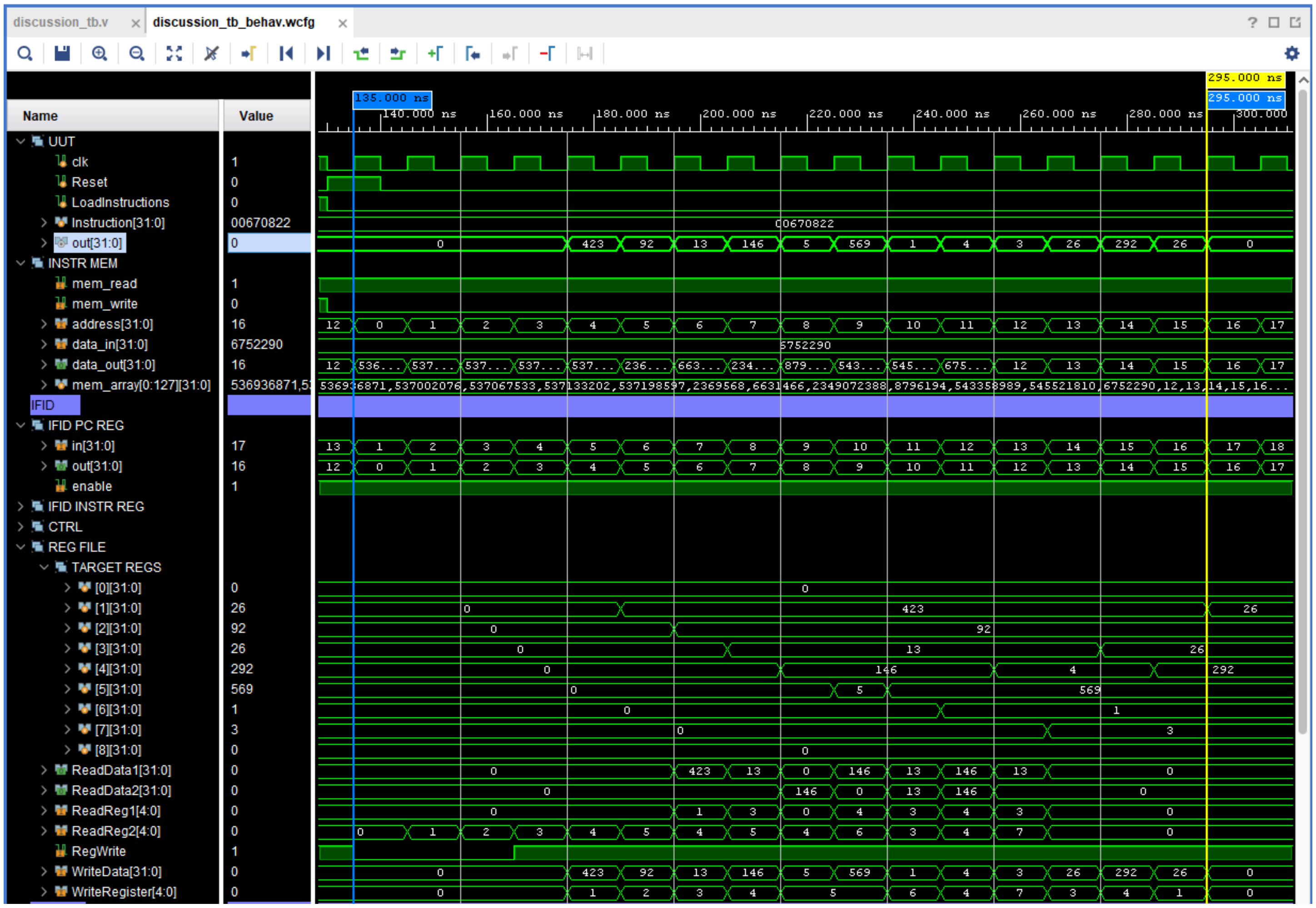


Task 7: Check register bypass works

To check the register bypass, I added another test case that would be 3-ahead (as opposed to 1- or 2-ahead), and commented/uncommented the provided register bypass logic as I ran the testbench/simulations. The modified testbench is as follows (note the bottom three instructions after the first sub instruction).

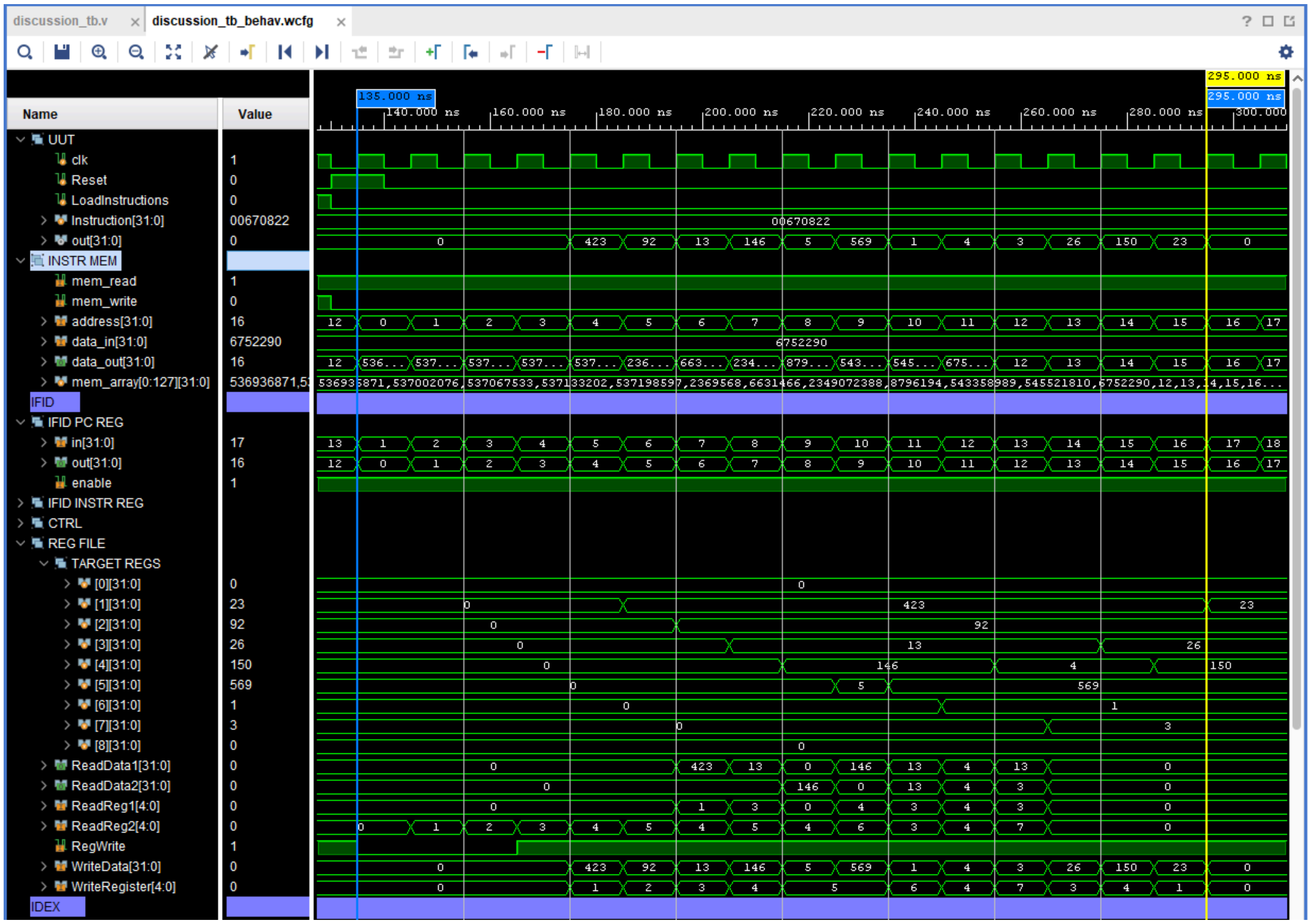
```
Instruction = 32'b001000_00000_00001_0000000110100111;      // addi $R1, $0, 423
#10; // 1                                                    // -> 423
Instruction = 32'b001000_00000_00010_0000000001011100;      // addi $R2, $0, 92
#10; // 2                                                    // -> 92
Instruction = 32'b001000_00000_00011_0000000000001101;      // addi $R3, $0, 13
#10; // 3                                                    // -> 13
Instruction = 32'b001000_00000_00100_0000000010010010;      // addi $R4, $0, 146
#10; // 4                                                    // -> 146
Instruction = 32'b001000_00000_00101_0000000000000101;      // addi $R5, $0, 5
#10; // 5                                                    // -> 5
Instruction = 32'b000000_00001_00100_00101_00000_100000;    // add $R5, $R1, $R4
#10; // 6                                                    // -> 569 (423 if wrong, 2-ahead)
Instruction = 32'b000000_00011_00101_00110_00000_101010;    // slt $R6, $R3, $R5
#10; // 7                                                    // -> 1 (0 if wrong, 1-ahead)
Instruction = 32'b100011_00000_00100_0000000000000100;      // lw $R4, 4(R0)
#10; // 8                                                    // -> 4
Instruction = 32'b000000_00100_00110_00111_00000_100010;    // sub $R7, $R4, $R6
#10; // 9                                                    // -> 3 (146 or 145 if wrong, 1-ahead & 2-ahead)

Instruction = 32'b001000_00011_00011_0000000000001101;      // addi $R3, $R3, 13
#10; // 10                                                   // -> 26
Instruction = 32'b001000_00100_00100_0000000010010010;      // addi $R4, $R4, 146
#10; // 11                                                   // -> 150 (292 if wrong, testing register bypass/3-ahead)
Instruction = 32'b000000_00011_00111_00001_00000_100010;    // sub $R1, $R3, $R7
#10;                                                         // -> 23 (26 if wrong, testing register bypass/3-ahead)
```

Without register bypass, the last two values are 292 and 26; however, this is wrong, since the last operation, sub $R1, $R3, $R7, should have stored 26 - 3 = 23 into $R1, and the second-to-last operation, addi $R4, $R4, 146, should have stored 4 + 146 = 150 into $R4. Because register bypass was not activated, the newly written value for $R7 was not properly loaded.



Name	Value
UUT	
clk	1
Reset	0
LoadInstructions	0
Instruction[31:0]	00670822
out[31:0]	0
INSTR MEM	
mem_read	1
mem_write	0
address[31:0]	16
data_in[31:0]	6752290
data_out[31:0]	16
mem_array[0:127][31:0]	536936871,5
IFID	
IFID PC REG	
in[31:0]	17
out[31:0]	16
enable	1
IFID INSTR REG	
CTRL	
REG FILE	
TARGET REGS	
[0][31:0]	0
[1][31:0]	26
[2][31:0]	92
[3][31:0]	26
[4][31:0]	292
[5][31:0]	569
[6][31:0]	1
[7][31:0]	3
[8][31:0]	0
ReadData1[31:0]	0
ReadData2[31:0]	0
ReadReg1[4:0]	0
ReadReg2[4:0]	0
RegWrite	1
WriteData[31:0]	0
WriteRegister[4:0]	0

In contrast, the following waveform is with register bypass enabled. As can be seen, the last two values are 150 and 23, which are expected and correct. Thus the register bypass provided is correctly functioning.



Name	Value
UUT	
clk	1
Reset	0
LoadInstructions	0
Instruction[31:0]	00670822
out[31:0]	0
INSTR MEM	
mem_read	1
mem_write	0
address[31:0]	16
data_in[31:0]	6752290
data_out[31:0]	16
mem_array[0:127][31:0]	536936871,5
IFID	
IFID PC REG	
in[31:0]	17
out[31:0]	16
enable	1
IFID INSTR REG	
CTRL	
REG FILE	
TARGET REGS	
[0][31:0]	0
[1][31:0]	23
[2][31:0]	92
[3][31:0]	26
[4][31:0]	150
[5][31:0]	569
[6][31:0]	1
[7][31:0]	3
[8][31:0]	0
ReadData1[31:0]	0
ReadData2[31:0]	0
ReadReg1[4:0]	0
ReadReg2[4:0]	0
RegWrite	1
WriteData[31:0]	0
WriteRegister[4:0]	0
IDEX	