Gene Mary Cheruvathur and Ashley Guillard
EE 469
May 3, 2024

Lab 3 Report

Procedure

For Lab 2, we tackled the challenge of crafting a single cycle processor. Taking it further, we expanded on our work in this lab to develop a multicycle processor. We broke down the task into smaller parts, implementing techniques like pipelining, forwarding, stalling, and branching. These enhancements allowed us to execute multiple instructions simultaneously, effectively completing more tasks within a given timeframe.

Task #1 - Pipelining the Processor

For this part of the lab, we're transitioning from a single-cycle processor to a multi-cycle one. With the single-cycle setup, each instruction completed fully before the next began. By introducing pipelining to the processor, we aim to enhance efficiency by executing multiple instructions simultaneously, thereby increasing the throughput of instructions within a given timeframe. You can observe the comparison in Figure 1, and we've provided an example of how a set of instructions runs through the pipelined processor in Figure 2.

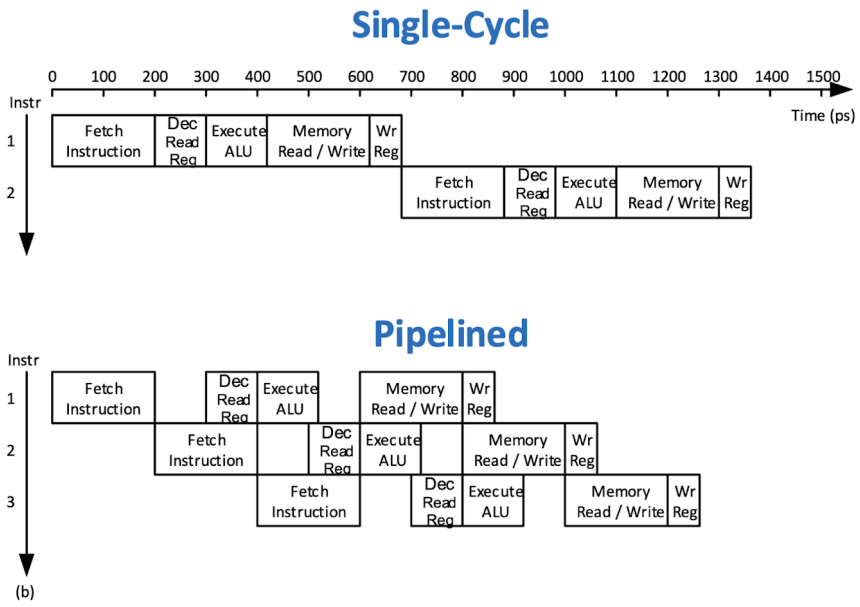

Figure 1: Comparison of Single Cycle Processor and a Pipelined Processor

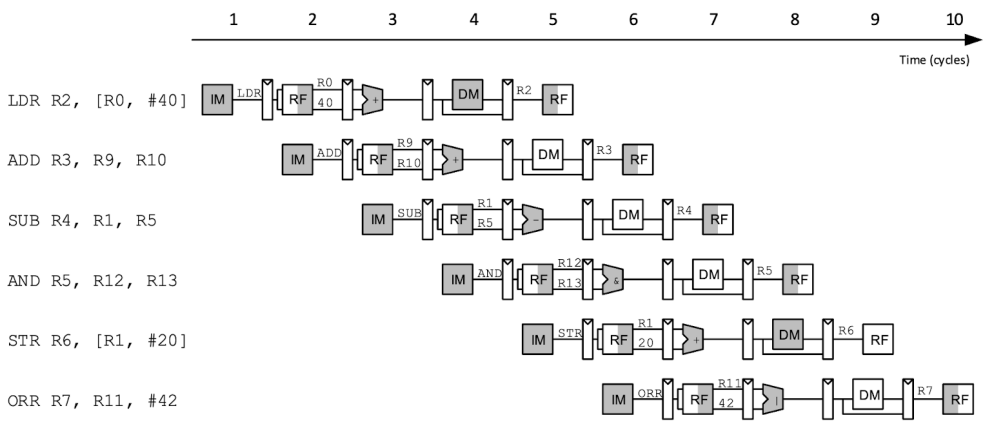

Figure 2: Comparison of Single Cycle Processor and a Pipelined Processor

When implementing pipelining, we segmented the processor into five stages: fetch, decode, execute, memory, and writeback. Each stage handles different operations as the instructions progress.

We integrated pipeline registers into our Lab 2 ARM module, aligning with the connections illustrated in Figure 3. This addition required more wiring and some adjustments to the logic. For instance, in the single processor setup, we needed two adders for PCPlus8. However, with pipelining, PCPlus8 no longer necessitates an extra adder; instead, by the next clock cycle, it automatically adds the additional 4 to itself. Additionally, some signal assertions for the multiplexers changed due to the inclusion of extra logic gates in the execute stage.

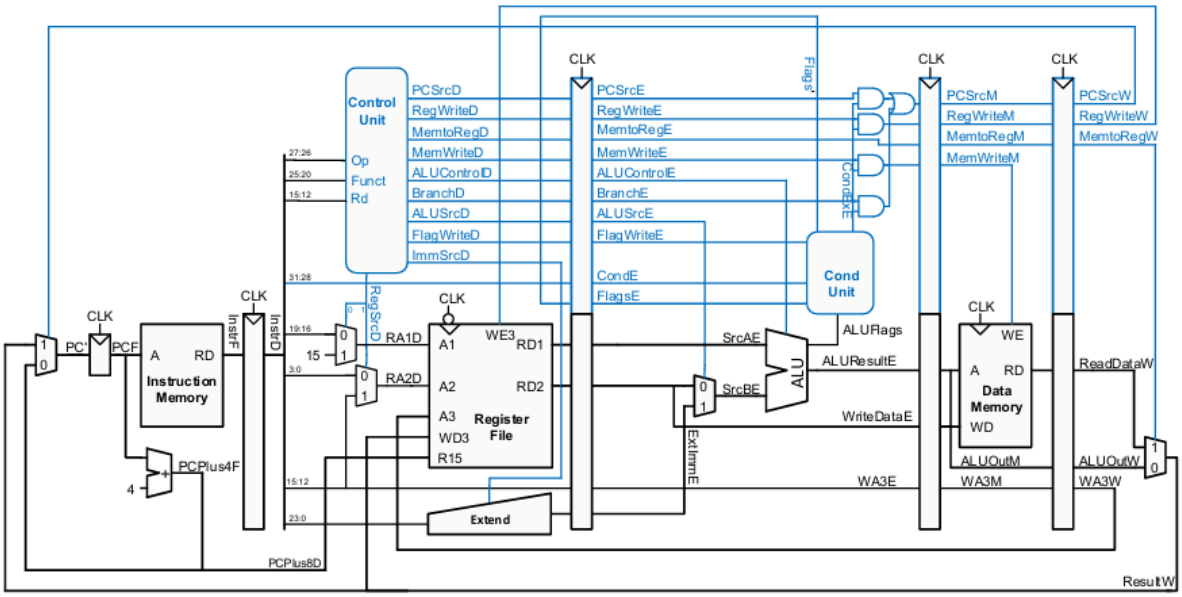

Figure 3: ARM processor with pipeline registers and the connections between each stage

After configuring the pipelined processor, we conducted tests to ensure that instructions were indeed being executed within a shorter time frame. To achieve this, we devised a test vector, depicted in Figure 4. It's essential to emphasize that this test vector was designed to avoid hazards; it's a hazard-free test intended to produce valid outputs..

```
// FIRST TEST FOR PIPELINE REGISTERS !

// Only adds values that do not cause hazards

/*
11100000100011110000000000001111 // ADD R0, R15, R15   PC=0   R0=16
11100010100011110001000000001010 // ADD R1, R15, #10   PC=4   R1=22
11100000100011110010000000001111 // ADD R2, R15, R15   PC=8   R2=32
11100010100011110011000000001100 // ADD R3, R15, #12   PC=12  R3=32
11100000100000000100000000000001 // ADD R4, R0,  R1    PC=16  R4=38
11100010100000000101000000000001 // ADD R5, R0,  #1    PC=20  R5=17
11100000100011110000000000001111 // ADD R0, R15, R15   PC=24  R0=64
11100000100011110000000000001111 // ADD R0, R15, R15   PC=28  R0=72
11100000100011110000000000001111 // ADD R0, R15, R15   PC=32  R0=80
11100000100011110000000000001111 // ADD R0, R15, R15   PC=36  R0=88
11100000100011110000000000001111 // ADD R0, R15, R15   PC=40  R0=96
11100000100011110000000000001111 // ADD R0, R15, R15   PC=44  R0=104
*/
```

Figure 4: Test Vector created for testing Pipelined Processor

Task #2 - Hazard Controls

After testing that the pipelined processor was able to pass along the correct information between the pipeline registers, we began working on eliminating the hazards so that we could test our processor with the provided memfile3.dat file. There were three main hazards that needed to be fixed in order to finish our pipelined processor, data forwarding, stalling, and flushing instructions when branching. By implementing an additional controller unit to our arm.sv file, we were able to handle each of these problems one at a time. Our processor would now consist of a datapath, controller, and hazard control.

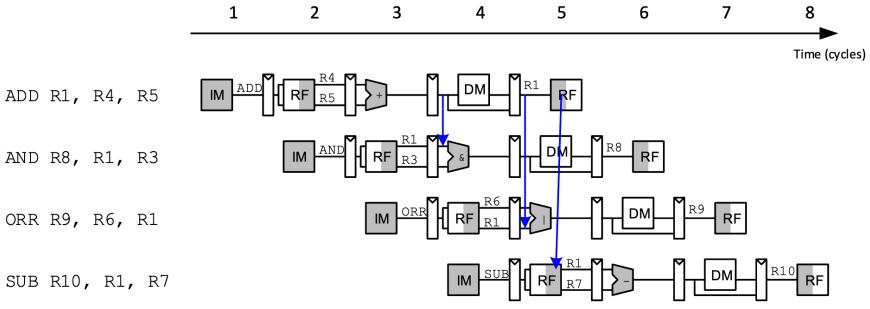

Figure 5: Data Forwarding Hazard Example

The first hazard, data forwarding, happened when the next instruction used data from the previous instruction that had not been stored within the register in the writeback stage. Figure 5

shows a couple examples where the new data for R1, calculated in the first instruction, needs to be used before the register's new data has been written in the reg_file. As can be seen, R1's new data is determined when entering the memory stage of our processor. This means that we can use the data in either the memory stage or the writeback stage to solve this hazard. Since it would be disadvantageous to stall the system, we can try to input the data the next instruction needs into the execute stage.

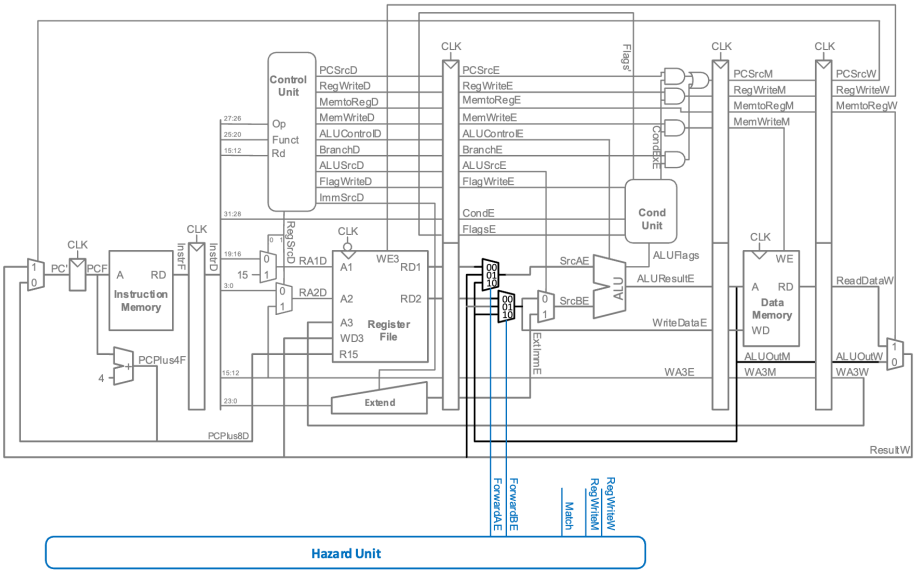

Figure 6: Hazard Unit for Forwarding

To do this, we created two variables, ForwardAE and ForwardBE, which would be 2-bits and based on when the read address for ALU sources A and B, in the execute stage, as well as the write addresses of the previous instruction are the same to determine when to forward data. When ForwardAE is 00, no data is forwarded, when ForwardAE is 01, the data within the memory stage will be forwarded to the next instruction's execute stage. Figure 6 shows the multiplexers used to determine data forwarding. With these new variables, the arm processor was testing, see Result section.

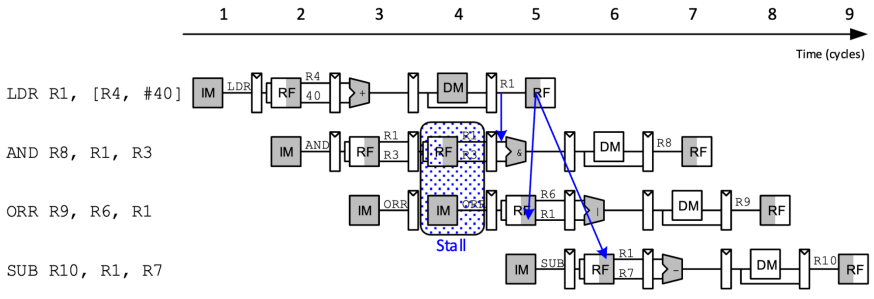

Figure 7: Stalling Hazard Example

The second hazard, stalling, happened when we loaded a register with data from the dmem.sv file. When loading a register, we get the data from the memory stage. This means that we cannot use only data forwarding to put this data into the register, we have to stall the processor for one clock cycle giving the system enough time to read the dmem file and write back into the register file before it is read in the next decode stage. This process can be seen in Figure 7.

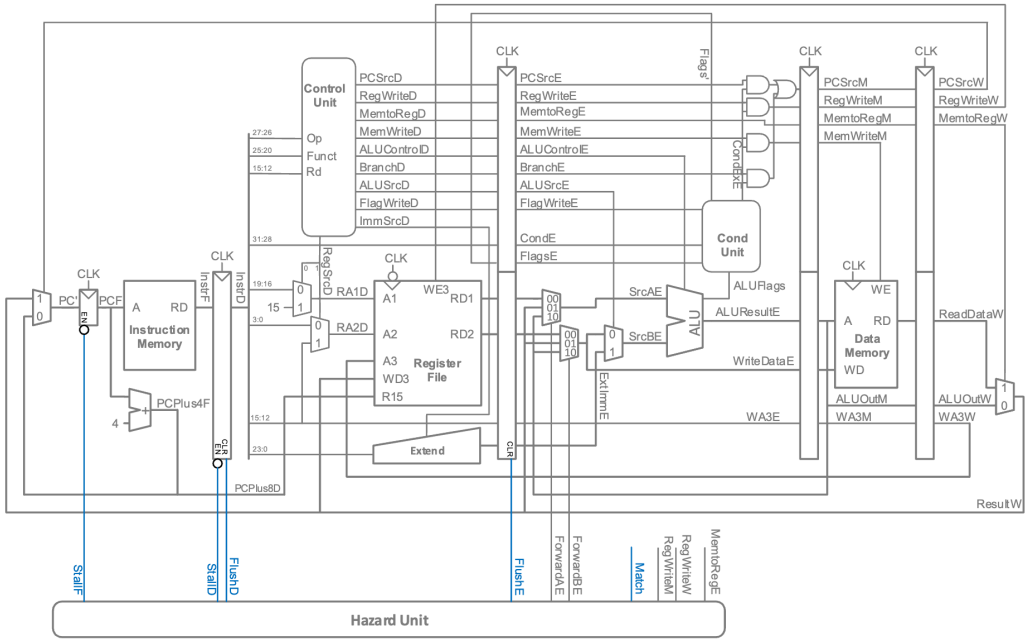

Figure 8: Hazard Unit for Stalling

To determine when we need to stall the processor, we need to know when the current execute and writeback addresses match and we are writing to the reg_file. Using variables StallD and StallF, we can stall the processor when the instructions are still gathering the date for loading, see Figure 8. Since we have now stalled the system, we need to flush the previous values within the decode and execute stages. Flushing these values ensures that the system does not start these instructions too soon causing inaccurate readings. Once this section of the hazard unit was completed, this hazard was tested, see Results section.

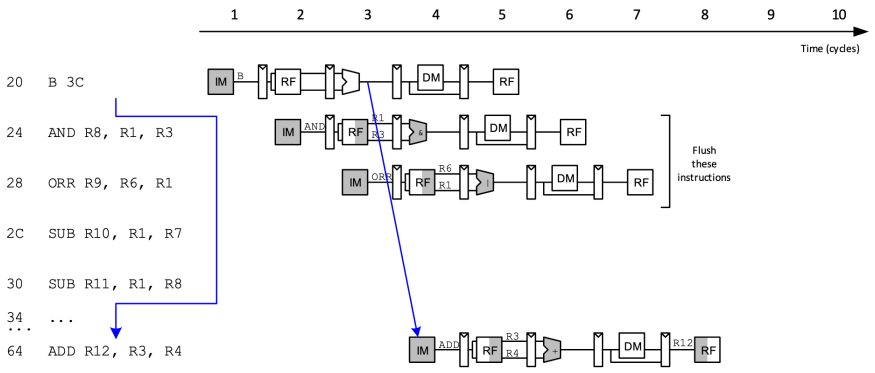

Figure 9: Branching Hazard Example

The third and final hazard happens when branching. When we fetch the branching instruction, it takes at least 2 more cycles to determine if we need to branch, based on condition, and where we are branching to, based on ALUResult. Due to this, we need to stall the processor for two cycles using our already created variables, StallF and StallD. In addition, similar to the previous hazard, we have to flush the instructions that were fetched before we knew if we were branching or not. Figure 9 shows an example where we would need to stall and flush instructions in order to properly branch.

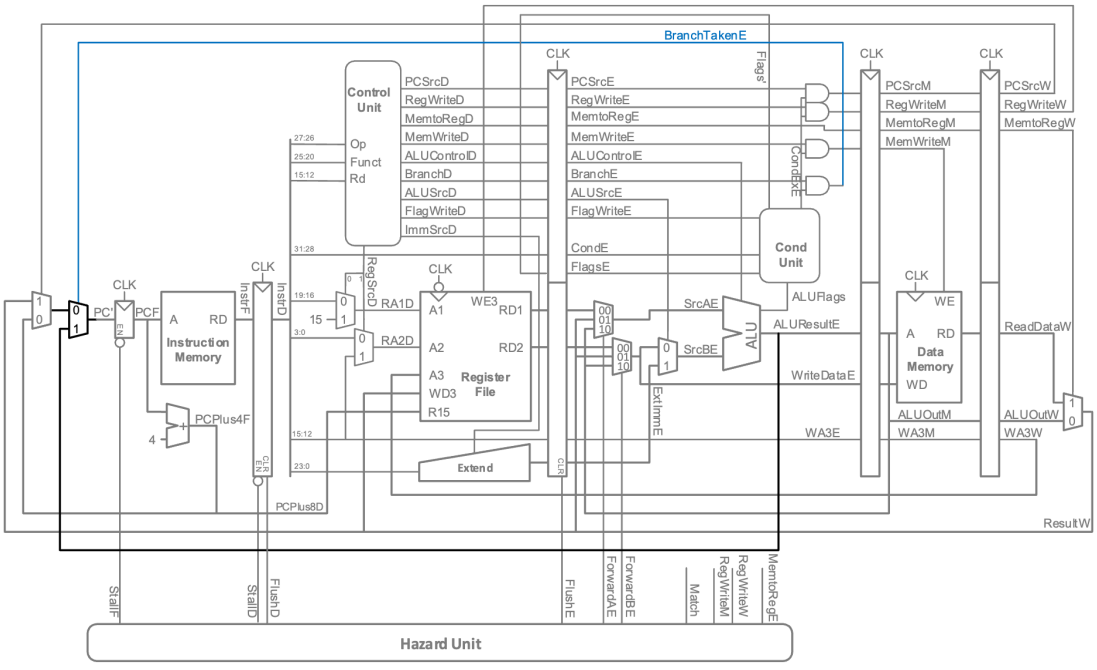

Figure 10: Finished ARM Processor

The earliest stage we can determine if and where to branch is the execute stage. Figure 10 shows the variables BranchTakenE which is only on when BranchD was on and the condition is met

based on previous SUBS command, which stores the ALU flags after a subtraction. As described in the previous lab, these flags determine if the instructions condition has been met. With this final hazard/control variable, each of the possible hazards with our current arm processor are addressed and handled. The final product was tested using the provided memfile3 with extra instructions, see Results or Appendix D.

Results

Task #1 - Pipelining the Processor

After running the test vector outlined in the procedure, we examined the resulting waveforms, which are depicted below. We meticulously assessed each stage of the pipelining process to ensure that the output of each pair of logics was synchronized and accurate.

The first waveform corresponds to the pipeline register between the fetch and decode stages (Figure 11). Here, we observed that the waveform for InstrD appeared one cycle after Instr, affirming the correctness of our connection.

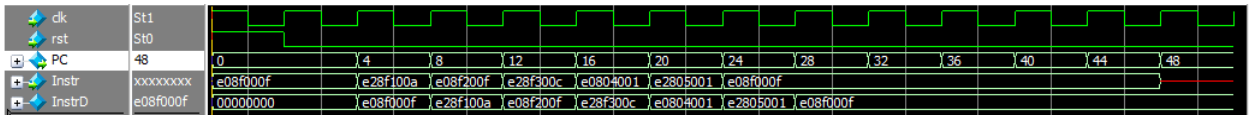

Figure 11: Pipeline Register for the Fetch to Decoder stage transition

Moving on to the second test, which evaluates the logics between the decode and execute stages. While not all logics are displayed in the waveform (Figure 12), we identified consistent patterns in the connections that remained unchanged. Additionally, we specifically examined CondE and Instr[31:28] to confirm that CondE accurately captured Instr[31:28] one cycle later. We also scrutinized the register file inputs and outputs to verify the correct addresses were being inputted to SrcAE and SrcBE. It's worth noting that adjustments were made to the logic assigning values to SrcAE and SrcBE due to the introduction of the pipeline register, but overall, the waveform aligned with our expectations.

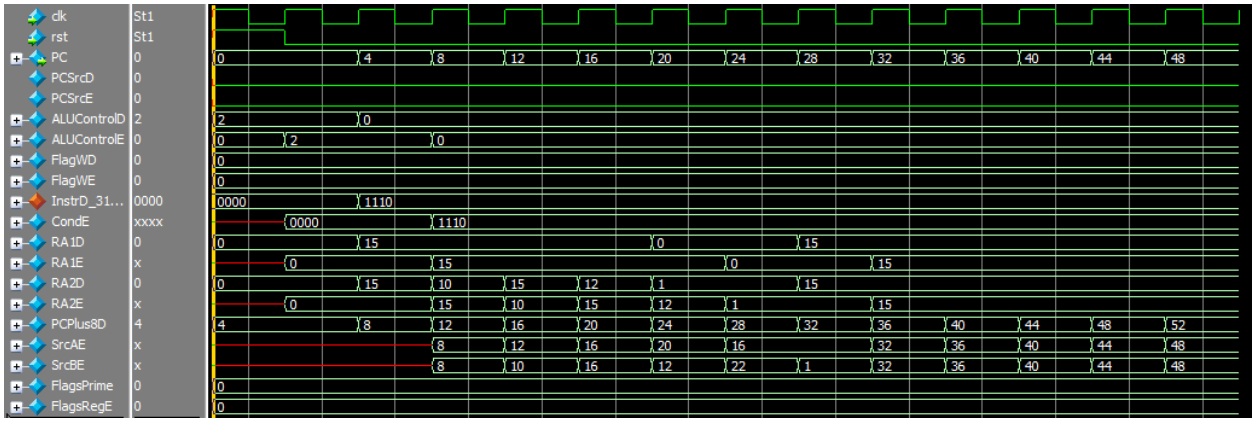

Figure 12: Pipeline Register for the Decoder to Execute stage transition

Moving forward, the third waveform (Figure 13) scrutinizes the transition from the execute to memory stages, with every line and number appearing as expected. Similarly, the waveform representing the pipeline register between the memory and writeback stages (Figure 14) exhibits accurate transitions.

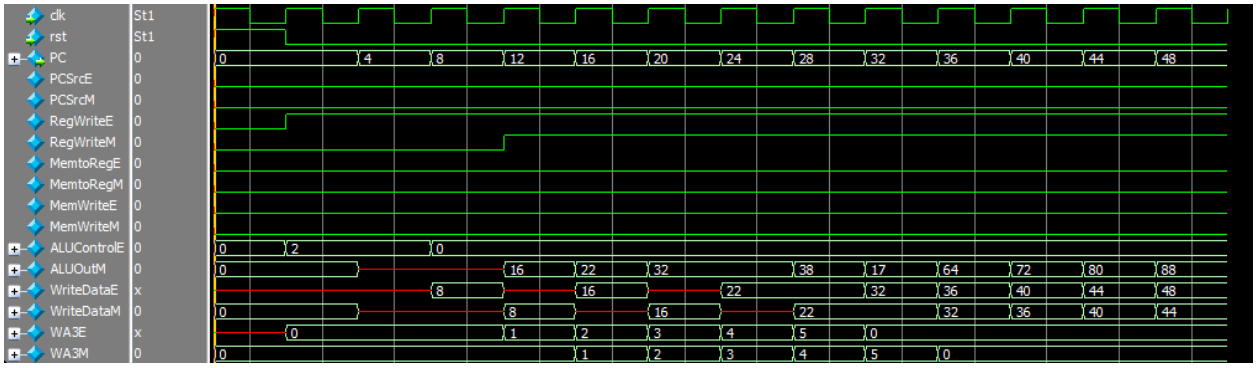

Figure 13: Pipeline Register for the Execute to Memory stage transition

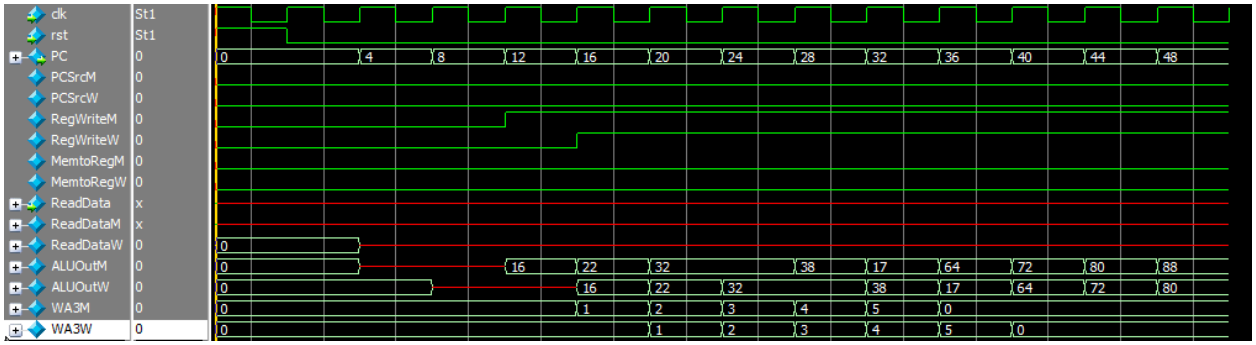

Figure 14: Pipeline Register for the Memory to Writeback stage transition

Finally, we proceeded to validate the accuracy of the executed instructions' outputs. Table 1 summarizes all executed instructions and their expected results. These results are also illustrated

in Figure 15, with the output beginning to print out five cycles later. This outcome confirms the successful completion of the pipeline connections for the multi-cycle processor, as we anticipated. Since we have five stages, the completion of five cycles aligns with the pipeline structure.

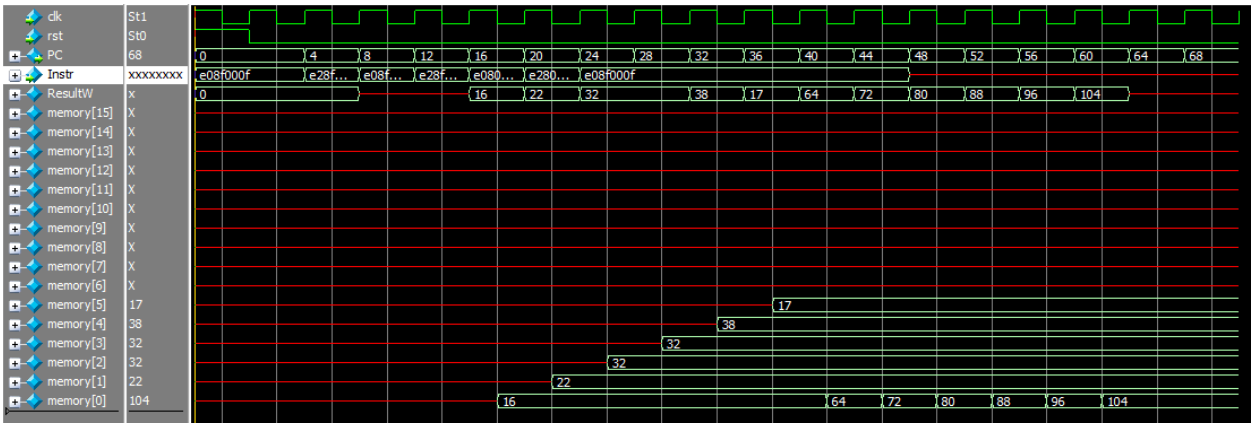

Figure 15: Results and Register Data after all instructions have been executed

Cycle	PC	Action	Result
1	0	ADD R0, R15, R15	R0 ← 16
2	4	ADD R1, R15, #10	R1 ← 22
3	8	ADD R2, R15, R15	R2 ← 32
4	12	ADD R3, R15, #12	R3 ← 32
5	16	ADD R4, R0, R1	R4 ← 38
6	20	ADD R5, R0, #1	R5 ← 17
7	24	ADD R0, R15, R15	R0 ← 64
8	28	ADD R0, R15, R15	R0 ← 72
9	32	ADD R0, R15, R15	R0 ← 80
10	36	ADD R0, R15, R15	R0 ← 88
11	40	ADD R0, R15, R15	R0 ← 96
12	44	ADD R0, R15, R15	R0 ← 104

Table 1: First Test for Pipeline Registers (Pipelining)

Task #2 - Hazard Controls

As mentioned in the procedure section of this report, after ensuring that the correct values were passed in the pipelined processor, the hazards were found and solutions were made so that the system could perform each instruction correctly no matter what order they come in. The second test written in the TestingFile covered cases where data forwarding would be necessary. This test can be found under the label "SECOND TEST FOR PIPELINE REGISTERS !" within the TestingFile. For ease, the below table shows each instruction used in order and provides relevant information of when and how forwarding was tested.

Cycle	PC	Action	Forwarding?	Result
1	0	ADD R0, R15, R15	No	R0 ← 16
2	4	ADD R1, R0, #10	Forward R0 from memory to execute stage.	R1 ← 26
3	8	ADD R2, R0, R1	Forward R0 from writeback to execute stage. Forward R1 from memory to execute stage.	R2 ← 42
4	12	ADD R0, R15, #0	No	R0 ← 20
5	16	ADD R0, R15, #0	No	R0 ← 24
6	20	ADD R0, R15, #0	No	R0 ← 28
7	24	ADD R0, R15, #0	No	R0 ← 32
8	28	ADD R0, R15, #0	No	R0 ← 36

Table 2: Second Test for Pipeline Registers (Forwarding)

The below ModelSim quickly shows that each result, from Table 2, was accurately calculated in the processor. The registers 0 through 2 were written with the correct information during this test proving that the system was able to complete the above forwarding trial. During the execute stage of the second instruction, R1 was able to grab the information from the memory stage of the first instruction without error. This will be shown further below.

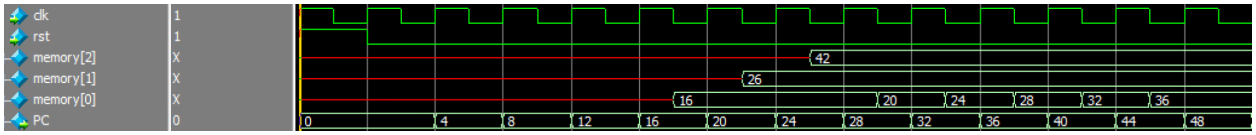

Figure 16: Forwarding ModelSim with PC and Memory

The next ModelSim displays forwarding variables, ForwardAE and ForwardBE in the bottom two lines. The four match waveforms above are used to determine when the memory/writeback

addresses and next execute read addresses are the same. In the fifth clock cycle, execute stage of the second instruction, the read address for SrcAE, and the write memory address, WA3M are equal and we are performing a register write instruction. This turns ForwardAE to -2 (10 in binary) meaning SrcAE will not be filled from the RD1E but instead from ALUOutM which was 16, the data that will be written in R0. The same logic can be applied to the other sections.

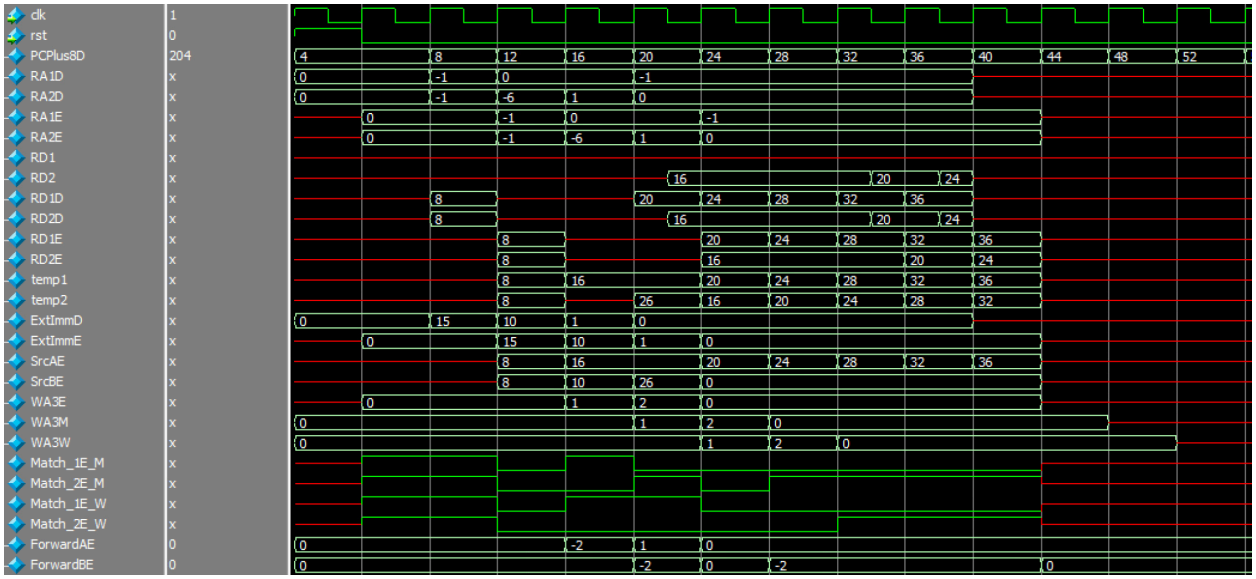

Figure 17: Forwarding ModelSim with Read/Write Addresses

The final forwarding ModelSim shows the values that SrcAE and SrcBE take from when forwarding, ALUOutM and ALUOutW when forwarding. In the fifth cycle, that was discussed above, we can see that the ALUOutM was assigned to SrcAE correctly when needed resulting in the correct value stored in R1. With the assurance that our processor could handle data forwarding, we were able to move on to testing the stalling hazard that could be caused by loading registers. Table 3 shows the third test within the TestingFile, see Appendix D.

Cycle	PC	Action	Stall?	Result
1	0	ADD R0, R15, #0	No	R0 ← 8
2	4	STR R0, [R0, #0]	No	Store R0 in dmem address 8.
3	8	ADD R1, R15, #0	No	R1 ← 16
4	12	LDR R1, [R0, #0]	No	R1 ← 8
5	16	ADD R1, R1, #1	Stall until R1 is loaded	R1 ← 9
6	20	LDR R2, [R0, #0]	No	R2 ← 8
7	24	ADD R1, R2, #2	Stall until R2 is loaded	R1 ← 10

8	28	LDR R3, [R0, #0]	No	R3 ← 8
9	32	ADD R1, R3, #3	Stall until R3 is loaded	R1 ← 11
10	36	LDR R4, [R0, #0]	No	R4 ← 8
11	40	ADD R1, R4, #4	Stall until R4 is loaded	R1 ← 12

Table 3: Third Test for Pipeline Registers (Stalling)

This table requires the system to stall after each load, but requesting the data that would be stored into the loaded register. The below ModelSim shows the dmem, register file memory, and PC to show that the resulting stored values are accurate to the result column of Table 3. The below section will dive into the stalling and flushing variables as well as their effect on the system.

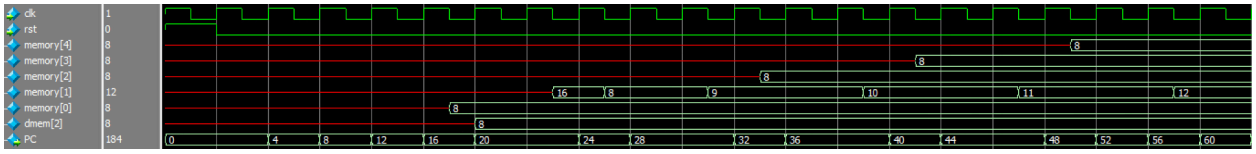

Figure 18: Stalling ModelSim with PC and Memory

The next ModelSim shows when Idrstall, StallF, StallD, and FlushE turn on. All of these variables are connected and turn on at the same time. Idrstall is based on when we are loading and the next read address matches the one being loaded. In cycle 7, Match_12D_E is on and MemtoReg is on, see Figure 19. As can be seen in cycle 7, the execute values are flushed and become 0. This is important as it turns the instructions into an no-op, no operation, instruction.

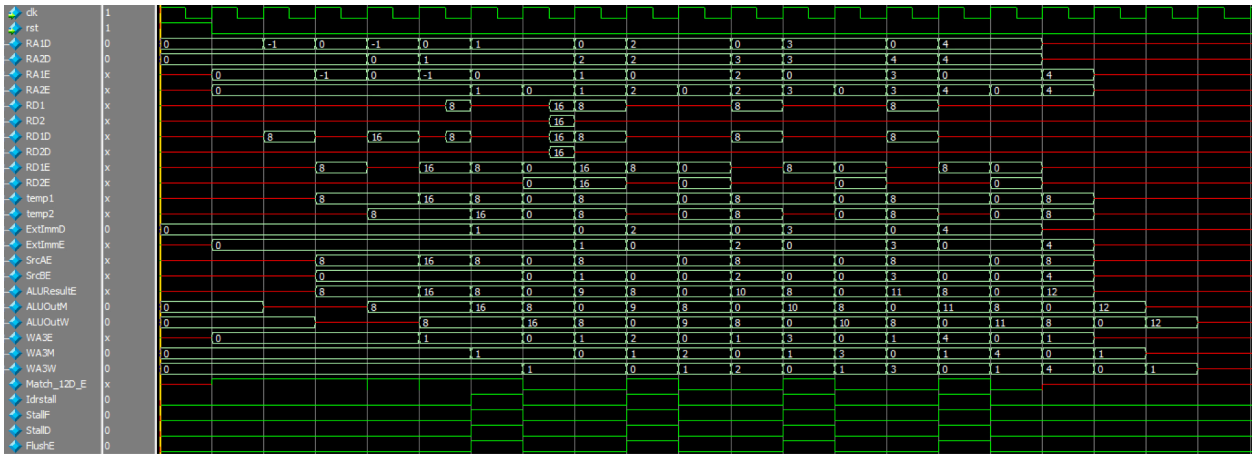

Figure 19: Stalling ModelSim with IdrStall

The final stalling ModelSim shows that Idrstall does not turn on unless MemtoReg is on and the match conditions from the previous figure are met. When stalled, the PC remains the same for a

clock cycle before being able to move forward. This is the correct response since we need to stall the entire system to properly load the register before using.

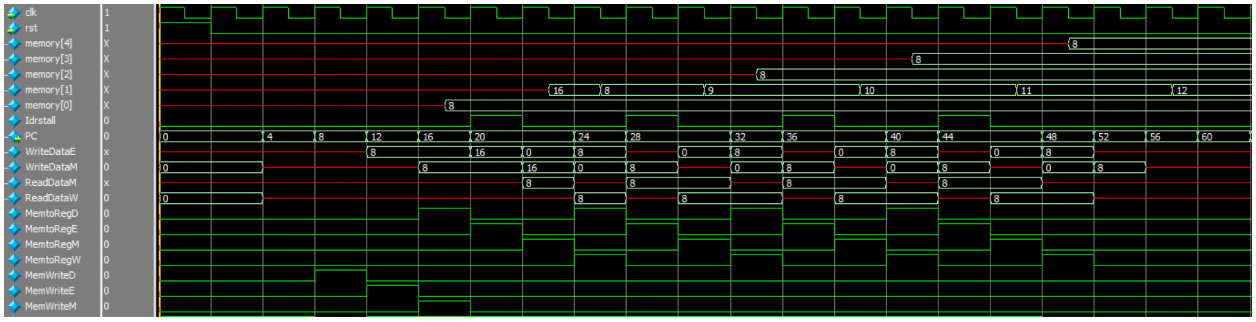

Figure 20: Stalling ModelSim with MemtoReg and PC

So far, the data forwarding and stalling have accurately happened during the tests making sure that hazards and warnings do not produce unwanted results. With the first two hazards out of the way, we were able to focus on the branching hazard. As mentioned in the Procedure section, when we branch, we need to flush two entire instructions whether or not the branching condition is met. A new test was created as shown in Table 4 and TestingFile, fourth test.

Cycle	PC	Action	Branching?	Result
1	0	ADD R0, R15, #0	No	R0 ← 8
2	4	SUBS R1, R0, #8	No	R1 ← 0 R0 and 8 are equal
3	8	BNE FORWARD	No Condition not met.	Do not branch PC=12
4	12	ADD R2, R0, R1	No	R2 ← 40
5	16	ADD R0, R15, #0	No	R0 ← 24
6	20	ADD R0, R15, #0	No	R0 ← 28
7	24	ADD R0, R15, #0	No	R0 ← 32
8	28	ADD R0, R15, #0	No	R0 ← 36
9	32	ADD R0, R15, #0	No	R0 ← 40
10	36	B LOOP	Yes Unconditional	Branch to PC=36
11	36	B LOOP	Yes Unconditional	Branch to PC=36

Table 4: Fourth Test for Pipeline Registers (Branching)

This test focused on both conditional and unconditional branching. Figure 21 shows the resulting register memory in lines 3-5 with accurate results. Whether the branch was allowed in the execute stage, like in cycle 13, or not allowed in the execute stage, like in cycle 6, the system only stalls for two clock cycles at maximum, when either BranchD or BranchTakenE is on. Additionally, this waveform shows the flags register to confirm that the branch should not happen in cycle 6.

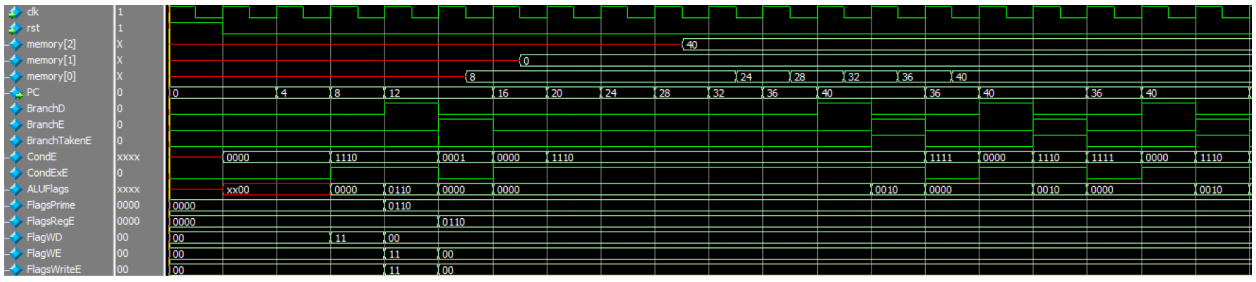

Figure 21: Branching ModelSim with Flags and PC

The final branching ModelSim shown below, Figure 22, shows the effect of StallF, StallD, and FlushF, FlushD, and FlushE on the stored values. An addition to the branching mechanism, is that the incoming instructions are flushed by the variable FlushF when branching. This seemed to provide a smoother transition between the branches and allowed for easier debugging. Flushing in the fetch stage meant that the chance of the 2 flushed instructions passing through the processor was essentially zero. These instructions were flushed at all possible stages and would not cause a problem while branching.

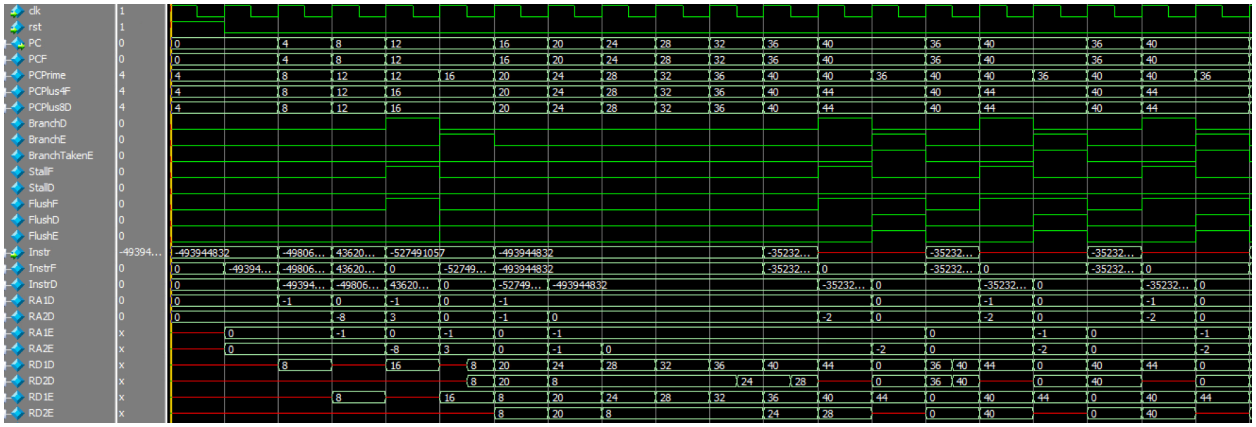

Figure 22: Branching ModelSim Flush and Stall

After testing each of the individual hazards separately, the memfile3.dat provided in the lab specs was tested. Two branching lines were added to test both branching allowed in execute and branching not allowed in the execute stage. The final test can be found in the TestingFile under the previous ones and in Table 5 below.

Cycle	PC	Action	Forwarding?	Stalling?	Branching?	Result
1	0	SUB R0 R15 R15	No	No	No	R0 ← 0
2	4	ADD R1 R0 #1	Yes	No	No	R1 ← 1
3	8	ORR R2 R0 R1	Yes	No	No	R2 ← 1
4	12	ADD R2 R0 #2	Yes	No	No	R2 ← 2
5	16	SUBS R0 R2 #0	Yes	No	No	R0 ← 2
6	20	BEQ TAG1	No	Yes	No Condition not met	PC' ← 24
7	24	AND R2 R2 R0	No	No	No	R2 ← 2
8	28	AND R1 R2 R0	Yes	No	No	R1 ← 2
9	32	ADD R9 R1 R0	Yes	No	No	R9 ← 4
10	36	STR R9 [R0, #11]	No	No	No	Store R9 in dmem[11]
11	40	LDR R3 [R0, #9]	No	No	No	R3 ← 4
12	44	AND R2 R3 R2	Yes	Yes	No	R2 ← 0
13	48	SUBS R2 R2 R0	Yes	No	No	R2 ← -2
14	52	BLT REDO	No	Yes	Yes	PC' ← 44
15	44	AND R2 R3 R2	No	No	No	R2 ← 4
16	48	SUBS R2 R2 R0	Yes	No	No	R2 ← 2
17	52	BLT REDO	No	Yes	No Condition not met	PC' ← 60
18	56	B LOOP	No	Yes	Yes Unconditional	PC' ← 60
19	56	B LOOP	No	Yes	Yes Unconditional	PC' ← 60

Table 5: Fifth Test for Pipeline Registers (All functions)

This table also has columns showing when each type of hazard was tested during the cycles. Figure 23 shows that the register files memory which matches the results shown in Table 5. Based on the fact that our files passed each of the previous tests and has the correct resulting values within the register files, we can conclude that the system works as expected.

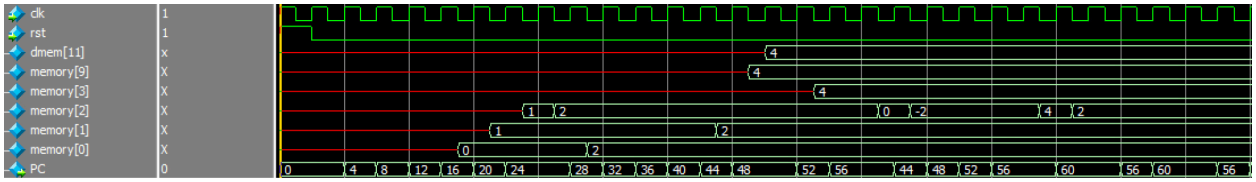

Figure 23: ModelSim with Memory and PC

The final ModelSim shows the PC and hazard controls that were toggled throughout the final test. These were used to ensure that each hazard control turned on at the right time and was checked at the very end of this lab.

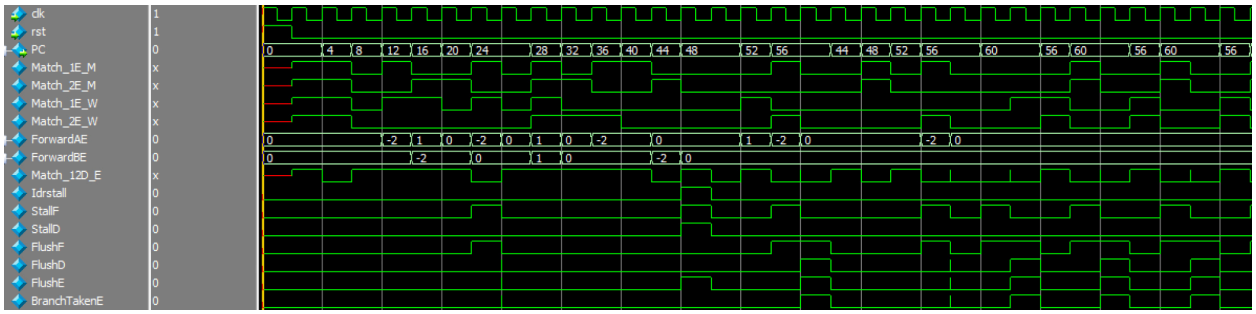

Figure 24: ModelSim with Hazard Controls

Final Product

This lab introduced an interesting concept of pipelining CPUs. As shown in the lab, this can actually save time when processing instructions. Task 1's Procedure compared the previous single cycle processor we have made to a pipelined processor. For this lab, we were instructed to edit our Lab 2 ARM processors to change it from a single cycle to a multi-cycle processor. To achieve this we pipelined, forwarded, stalled, and branched the processor. When we completed the revisions, we tested each of the four changes and the overall change with 5 test vectors. After simulating the arm processor, we had observed the results to be accurate with our expectations.

The most challenging part to this lab was the confusion between lectures and the lab specs. The lecture preceding this lab mentioned the PCSrc variables and a new variable labeled PCWrPendingF. This seemed to confuse many people working on this lab including our group and we spent a lot of time trying to debug the PCSrc issues. However, the lab specifications did not mention needing to implement this into lab 3. After confirming with various TAs, we found that we did not need to implement this and that during our tests the PCSrcs would not be used. This is why they did not show up within the second task's result section.

Appendix A - Processor

1) arm.sv

```
// I will input mine unless you finish yours
/* arm is the spotlight of the show and contains the bulk of the datapath and
control logic. This module is split into two parts, the datapath and control.
*/

// clk - system clock
// rst - system reset
// Instr - incoming 32 bit instruction from imem, contains opcode, condition,
addresses and or immediates
// ReadData - data read out of the dmem
// WriteData - data to be written to the dmem
// MemWrite - write enable to allowed WriteData to overwrite an existing dmem
word
// PC - the current program count value, goes to imem to fetch instruciton
// ALUResult - result of the ALU operation, sent as address to the dmem

module arm (
    input  logic        clk, rst,
    input  logic [31:0] Instr,
    input  logic [31:0] ReadData,
    output logic [31:0] WriteData,
    output logic [31:0] PC, ALUResult,
    output logic        MemWrite
);

    // datapath buses and signals
    logic [31:0] PCF, PCPrime, PCPlus4F, PCPlus8D; // pc signals
       logic [31:0] InstrF, InstrD;                // instructions
       logic [ 3:0] CondE;                         // condition from
instructions
    logic [ 3:0] RA1D, RA2D, RA1E, RA2E;           // regfile input addresses
    logic [31:0] RD1, RD2, RD1D, RD2D, RD1E, RD2E; // raw regfile outputs
       logic [31:0] temp1, temp2;                  // intermediate between
regfile and alu inputs
       logic [31:0] ExtImmD, ExtImmE;              // immediate values
       logic [31:0] SrcAE, SrcBE;                  // alu inputs
    logic [ 3:0] ALUFlags;                         // alu combinational flag
outputs
    logic [31:0] ALUResultE, ALUOutM, ALUOutW;     // alu result registers
       logic [ 3:0] WA3E, WA3M, WA3W;              // regfile write
addresses
       logic [31:0] WriteDataE, WriteDataM;        // data memory write
data
       logic [31:0] ReadDataM, ReadDataW;          // data memory read data
    logic [31:0] ResultW;                          // computed or fetched
value to be written into regfile or pc
```

```
        logic [ 3:0] FlagsPrime, FlagsRegE;            // stores the flags from
the most recent SUBS command

    // control signals
    logic       PCSrcD, PCSrcE, PCSrcM, PCSrcW;             // pc source
control
        logic       MemtoRegD, MemtoRegE, MemtoRegM, MemtoRegW; // memory to
register control
        logic       ALUSrcD, ALUSrcE;                           // alu source
control
        logic       RegWriteD, RegWriteE, RegWriteM, RegWriteW; // register
write control
        logic       CondExE;                                    // condition
control
        logic       MemWriteD, MemWriteE, MemWriteM;            // memory write
control
        logic [1:0] ALUControlD, ALUControlE;                   // alu control
        logic       BranchD, BranchE;                           // branch
control
        logic [1:0] FlagWD, FlagWE;                             // writing
flags control
        logic [1:0] FlagsWriteE;                                // writing
flags control (after conditioning)
        logic [1:0] RegSrcD;                                    // register
source control
        logic [1:0] ImmSrcD;                                    // immediate
source control

        // hazard controls
        logic       Match_1E_M, Match_2E_M;   // regfile read and next write
addresses match
        logic       Match_1E_W, Match_2E_W;   // regfile read and write
addresses match
        logic [1:0] ForwardAE, ForwardBE;     // where to forward data from
        logic       Match_12D_E;              // regfile read and data memory
write addresses match
        logic       Idrstall, StallF, StallD; // Stalls incoming instructions
        logic       FlushF, FlushD, FlushE;   // flush instructions
        logic       BranchTakenE;             // ON when a branch is taken


        ////////////////////
        // Module Outputs  //
        ////////////////////

        assign PC        = PCF;         // output pc for instructions memory
        assign ALUResult = ALUOutM;     // output alu result for data memory
address
        assign WriteData = WriteDataM; // output write data for data memory
```

```
       assign MemWrite  = MemWriteM;  // output enable for data memory write


    /* The datapath consists of a PC as well as a series of muxes to make
decisions about which data words to pass forward and operate on. It is
    ** noticeably missing the register file and alu, which you will fill in
using the modules made in lab 1. To correctly match up signals to the
    ** ports of the register file and alu take some time to study and
understand the logic and flow of the datapath.
    */

//------------------------------------------------------------------------
---
    //                                      DATAPATH

//------------------------------------------------------------------------
---

       ////////////////////
       //      Fetch     //
       ////////////////////

       // priority mux, used to determine if a branch was taken, if not either
default or newly computed value
       always_comb begin
            if (BranchTakenE)  PCPrime = ALUResultE;
            else if (PCSrcW) PCPrime = ResultW;
            else               PCPrime = PCPlus4F;
       end
    assign PCPlus4F = PCF + 'd4;                    // default value to
access next instruction
    assign PCPlus8D = PCPlus4F;                     // value read when
reading from reg[15]

    // update the PC (unless stalled), at rst initialize to 0
    always_ff @(posedge clk) begin
        if (rst) PCF <= '0;
        else if (~StallF) PCF <= PCPrime;
    end

       // fetch instructions unless reset
       assign InstrF = (rst | FlushF) ? '0 : Instr;


       ////////////////////
       //     Decode     //
       ////////////////////

       // update instructions unless stalled/flushed
```

```
        always_ff @(posedge clk) begin
             if (~StallD) InstrD <= InstrF; // Stall
             else if (FlushD) InstrD <= '0; // Flush
        end

        // determine the register addresses based on control signals
    // RegSrc[0] is set if doing a branch instruction
    // RefSrc[1] is set when doing memory instructions
    assign RA1D = RegSrcD[0] ? 4'd15        : InstrD[19:16];
    assign RA2D = RegSrcD[1] ? InstrD[15:12] : InstrD[ 3: 0];

        // Register File (16x32)
    // Stores the data for registers 0 through 15.
        // 2 asynchronous read ports and 1 synchronous write port.
    reg_file u_reg_file (
        .clk       (~clk),
        .wr_en     (RegWriteW),
        .write_data(ResultW),
        .write_addr(WA3W),
        .read_addr1(RA1D),
        .read_addr2(RA2D),
        .read_data1(RD1),
        .read_data2(RD2)
    );

        // two muxes, put together into an always_comb for clarity
    // determines which set of instruction bits are used for the immediate
    always_comb begin
              if (FlushD) ExtImmD = '0;
         else if      (ImmSrcD == 'b00) ExtImmD =
{{24{InstrD[7]}},InstrD[7:0]};          // 8 bit immediate - reg operations
         else if (ImmSrcD == 'b01) ExtImmD = {20'b0, InstrD[11:0]};
// 12 bit immediate - mem operations
         else                      ExtImmD = {{6{InstrD[23]}}, InstrD[23:0],
2'b00}; // 24 bit immediate - branch operation
    end

        // mux to determine if reading from r15 or regfile
        always_comb begin
                  RD1D = (FlushD) ? '0 : ((RA1D == 'd15) ? PCPlus8D : RD1);
                  RD2D = (FlushD) ? '0 : ((RA2D == 'd15) ? PCPlus8D : RD2);
        end


        ////////////////////
        //     Execute    //
        ////////////////////

        // update execute instructions unless flushed
```

```
    always_ff @(posedge clk) begin
        if(FlushE) begin // Flush
            RD1E <= '0;
            RD2E <= '0;
            WA3E <= '0;
            ExtImmE <= '0;
            RA1E <= '0;
            RA2E <= '0;
        end
        else begin // Update
            RD1E <= RD1D;
            RD2E <= RD2D;
            WA3E <= InstrD[15:12];
            ExtImmE <= ExtImmD;
            RA1E <= RA1D;
            RA2E <= RA2D;
        end
    end

    // alu source values and write data
    assign SrcAE = temp1;
    assign SrcBE = (ALUSrcE) ? ExtImmE : temp2;
    assign WriteDataE = temp2;

    // muxes to determine forwarding data
    always_comb begin
            case(ForwardAE)
                2'b00: temp1 = RD1E;
                2'b01: temp1 = ResultW;
                2'b10: temp1 = ALUOutM;
                default: temp1 = 'X;
            endcase
            case(ForwardBE)
                2'b00: temp2 = RD2E;
                2'b01: temp2 = ResultW;
                2'b10: temp2 = ALUOutM;
                default: temp2 = 'X;
            endcase
    end

    // ALU (Arethmatic Logic Unit)
// Preforms addition, subtraction, ANDing, as well as ORing.
    // Outputs result of the operation and flags (NZCV).
alu u_alu (
    .a          (SrcAE),
    .b          (SrcBE),
    .ALUControl (ALUControlE),
    .Result     (ALUResultE),
    .ALUFlags   (ALUFlags)
```

```
    );

        // DFF for storing flags from SUBS command
        assign FlagsWriteE = {2{CondExE}} & FlagWE;
        assign FlagsPrime[3:2] = FlagsWriteE[1] ? ALUFlags[3:2] :
FlagsRegE[3:2];
        assign FlagsPrime[1:0] = FlagsWriteE[0] ? ALUFlags[1:0] :
FlagsRegE[1:0];
        always_ff @(posedge clk) begin
             if (rst) begin
                   FlagsRegE <= '0;
             end
             else begin
                   FlagsRegE <= FlagsPrime;
             end
        end


        ////////////////////
        //     Memory     //
        ////////////////////

        // update memory instructions
        always_ff @(posedge clk) begin
             if (rst) begin
                   ALUOutM    <= '0;
                   WriteDataM <= '0;
                   WA3M       <= '0;
             end
             else begin
                   ALUOutM    <= ALUResultE;
                   WriteDataM <= WriteDataE;
                   WA3M       <= WA3E;
             end
        end

        // read data from dmem file
        assign ReadDataM = ReadData;


        ////////////////////
        //    WriteBack   //
        ////////////////////

        // update writeback instructions
        always_ff @(posedge clk) begin
             if (rst) begin
                   ReadDataW <= '0;
                   ALUOutW   <= '0;
```

```
                WA3W      <= '0;
            end
            else begin
                ReadDataW <= ReadDataM;
                ALUOutW   <= ALUOutM;
                WA3W      <= WA3M;
            end
        end

        // determine the result to run back to PC or the register file based on
whether we used a memory instruction
    assign ResultW = MemtoRegW ? ReadDataW : ALUOutW;    // determine whether
final writeback result is from dmemory or alu



    /* The control conists of a large decoder, which evaluates the top bits of
the instruction and produces the control bits
    ** which become the select bits and write enables of the system. The write
enables (RegWrite, MemWrite and PCSrc) are
    ** especially important because they are representative of your processors
current state.
    */

//------------------------------------------------------------------------------
---
    //                                        CONTROL

//------------------------------------------------------------------------------
---

        ////////////////////
        //     Decode     //
        ////////////////////

        // decode instructions
    always_comb begin
            if (FlushD) begin // Flush
                PCSrcD    = 0;
          MemtoRegD = 0; // doesn't matter
          MemWriteD = 0;
          ALUSrcD   = 0;
          RegWriteD = 0;
                BranchD = 0;
                FlagWD = 2'b00;
          RegSrcD   = 'b00;
          ImmSrcD   = 'b00;
          ALUControlD = 'b00;  // do an add
```

```
            end
            else begin
        casez (InstrD[27:20])

            // ADD (Imm or Reg)
            8'b00?_0100_? : begin   // note that we use wildcard "?" in bit
25. That bit decides whether we use immediate or reg, but regardless we add
                PCSrcD    = 0;
                MemtoRegD = 0;
                MemWriteD = 0;
                ALUSrcD   = InstrD[25]; // may use immediate
                RegWriteD = 1;
                              BranchD = 0;
                              FlagWD = InstrD[20] ? 2'b11 : 2'b00;
                RegSrcD   = 'b00;
                ImmSrcD   = 'b00;
                ALUControlD = 'b00;
            end

            // SUB (Imm or Reg) OR CMP
            8'b00?_0010_? : begin   // note that we use wildcard "?" in bit
25. That bit decides whether we use immediate or reg, but regardless we sub
                PCSrcD    = 0;
                MemtoRegD = 0;
                MemWriteD = 0;
                ALUSrcD   = InstrD[25]; // may use immediate
                RegWriteD = 1;
                              BranchD = 0;
                              FlagWD = InstrD[20] ? 2'b11 : 2'b00; // keep
flags (CMP), don't keep flags (SUB)
                RegSrcD   = 'b00;
                ImmSrcD   = 'b00;
                ALUControlD = 'b01;
            end

            // AND
            8'b000_0000_? : begin
                PCSrcD    = 0;
                MemtoRegD = 0;
                MemWriteD = 0;
                ALUSrcD   = 0;
                RegWriteD = 1;
                              BranchD = 0;
                              FlagWD = InstrD[20] ? 2'b10 : 2'b00;
                RegSrcD   = 'b00;
                ImmSrcD   = 'b00;    // doesn't matter
                ALUControlD = 'b10;
            end
```

```
// ORR
8'b000_1100_? : begin
    PCSrcD    = 0;
    MemtoRegD = 0;
    MemWriteD = 0;
    ALUSrcD   = 0;
    RegWriteD = 1;
                BranchD = 0;
                FlagWD = InstrD[20] ? 2'b10 : 2'b00;
    RegSrcD   = 'b00;
    ImmSrcD   = 'b00;    // doesn't matter
    ALUControlD = 'b11;
end

// LDR
8'b010_1100_1 : begin
    PCSrcD    = 0;
    MemtoRegD = 1;
    MemWriteD = 0;
    ALUSrcD   = 1;
    RegWriteD = 1;
                BranchD = 0;
                FlagWD = 2'b00;
    RegSrcD   = 'b10;    // msb doesn't matter
    ImmSrcD   = 'b01;
    ALUControlD = 'b00;  // do an add
end

// STR
8'b010_1100_0 : begin
    PCSrcD    = 0;
    MemtoRegD = 0; // doesn't matter
    MemWriteD = 1;
    ALUSrcD   = 1;
    RegWriteD = 0;
                BranchD = 0;
                FlagWD = 2'b00;
    RegSrcD   = 'b10;    // msb doesn't matter
    ImmSrcD   = 'b01;
    ALUControlD = 'b00;  // do an add
end

// B
8'b1010_???? : begin
        PCSrcD    = 0; // Used BranchD instead
        MemtoRegD = 0;
        MemWriteD = 0;
        ALUSrcD   = 1;
        RegWriteD = 0;
```

```
                                    BranchD = 1;
                                    FlagWD = 2'b00;
                  RegSrcD   = 'b01;
                  ImmSrcD   = 'b10;
                  ALUControlD = 'b00;  // do an add
            end

                  default: begin
                              PCSrcD    = 0;
               MemtoRegD = 0; // doesn't matter
               MemWriteD = 0;
               ALUSrcD   = 0;
               RegWriteD = 0;
                              BranchD = 0;
                              FlagWD = 2'b00;
               RegSrcD   = 'b00;
               ImmSrcD   = 'b00;
               ALUControlD = 'b00;  // do an add
                  end
       endcase
            end
   end


      ////////////////////
      //     Execute    //
      ////////////////////

      // update execute controls unless flushed
      always_ff @(posedge clk) begin
            if(FlushE) begin // Flush
                  PCSrcE <= '0;
                  RegWriteE <= '0;
                  MemtoRegE <= '0;
                  MemWriteE <= '0;
                  ALUControlE <= '0;
                  BranchE <= '0;
                  ALUSrcE <= '0;
                  FlagWE <= '0;
                  CondE <= '1; // Default: OFF
            end
            else begin // Update
                  PCSrcE <= PCSrcD;
                  RegWriteE <= RegWriteD;
                  MemtoRegE <= MemtoRegD;
                  MemWriteE <= MemWriteD;
                  ALUControlE <= ALUControlD;
                  BranchE <= BranchD;
                  ALUSrcE <= ALUSrcD;
```

```
                FlagWE <= FlagWD;
                CondE <= InstrD[31:28];
            end
        end

        // Condition Check
        always_comb begin

                case (CondE)
                        4'b0000: CondExE = FlagsRegE[2];
// EQ (Equal)
                        4'b0001: CondExE = ~FlagsRegE[2];
// NE (Not equal)
                        4'b0010: CondExE = FlagsRegE[1];
// CS / HS (Carry set / Unsigned higher or same)
                        4'b0011: CondExE = ~FlagsRegE[1];
// CC / LO (Carry clear / Unsigned lower)
                        4'b0100: CondExE = FlagsRegE[3];
// MI (Minus / Negative)
                        4'b0101: CondExE = ~FlagsRegE[3];
// PL (Plus / Positive pf Zero)
                        4'b0110: CondExE = FlagsRegE[0];
// VS (Overflow / Overflow set)
                        4'b0111: CondExE = ~FlagsRegE[0];
// VC (No overflow / Overflow clear)
                        4'b1000: CondExE = ~FlagsRegE[2] & FlagsRegE[1];
// HI (Unsigned higher)
                        4'b1001: CondExE = FlagsRegE[2] | ~FlagsRegE[1];
// LS (Unsigned lower or same)
                        4'b1010: CondExE = ~(FlagsRegE[3] ^ FlagsRegE[0]);
// GE (Signed greater than or equal)
                        4'b1011: CondExE = FlagsRegE[3] ^ FlagsRegE[0];
// LT (Signed less than)
                        4'b1100: CondExE = ~FlagsRegE[2] & ~(FlagsRegE[3] ^
FlagsRegE[0]); // GT (Signed greater than)
                        4'b1101: CondExE = FlagsRegE[2] | (FlagsRegE[3] ^
FlagsRegE[0]);   // LE (Signed less than or equal)
                        4'b1110: CondExE = 1;
// AL / none (Always / unconditional)
                        default: CondExE = 0;
                endcase
        end

        assign BranchTakenE = (CondExE) ? BranchE : '0;


        ////////////////////
        //     Memory     //
        ////////////////////
```

```
        // update memroy controls
        always_ff @(posedge clk) begin
             if (rst) begin
                   PCSrcM    <= '0;
                   RegWriteM <= '0;
                   MemtoRegM <= '0;
                   MemWriteM <= '0;
             end
             else begin
                   PCSrcM    <= (PCSrcE & CondExE);// | (BranchE & CondExE);
                   RegWriteM <= RegWriteE & CondExE;
                   MemtoRegM <= MemtoRegE;
                   MemWriteM <= MemWriteE & CondExE;
             end
        end


        ////////////////////
        //    WriteBack    //
        ////////////////////


        // update writeback controls
        always_ff @(posedge clk) begin
             if (rst) begin
                   PCSrcW    <= '0;
                   RegWriteW <= '0;
                   MemtoRegW <= '0;
             end
             else begin
                   PCSrcW    <= PCSrcM;
                   RegWriteW <= RegWriteM;
                   MemtoRegW <= MemtoRegM;
             end
        end


//-------------------------------------------------------------------------------
---
    //                                  HAZARD CONTROL

//-------------------------------------------------------------------------------
---

        ////////////////////
        //   Forwarding   //
        ////////////////////
```

```
        // do the current execute and memory/writeback addresses match
        assign Match_1E_M = (RA1E == WA3M);
        assign Match_2E_M = (RA2E == WA3M);
        assign Match_1E_W = (RA1E == WA3W);
        assign Match_2E_W = (RA2E == WA3W);

        // muxes to determine forwarding procedure
        always_comb begin
                    if (Match_1E_M * RegWriteM) ForwardAE = 2'b10;
                    else if (Match_1E_W * RegWriteW) ForwardAE = 2'b01;
                    else ForwardAE = 2'b00;

                    if (Match_2E_M * RegWriteM) ForwardBE = 2'b10;
                    else if (Match_2E_W * RegWriteW) ForwardBE = 2'b01;
                    else ForwardBE = 2'b00;
        end


        //////////////////////
        //    Branching     //
        //////////////////////

        assign FlushF = (rst) ? '0 : (BranchD | BranchTakenE);  // Flush
Fetched instrucitons
        assign FlushD = (rst) ? '0 : BranchTakenE;              // Flush decode
stage
        assign FlushE = (rst) ? '0 : (Idrstall | BranchTakenE); // Flush
execute stage


        //////////////////////
        //    Stalling      //
        //////////////////////

        // do either decode and execute addresses match
        assign Match_12D_E = (RA1D == WA3E) | (RA2D == WA3E);

        // stall when calling a register that has not been loaded yet
        assign Idrstall = (MemtoRegE) ? Match_12D_E : '0;
        assign StallF = (rst) ? '0 : (Idrstall | BranchD);
        assign StallD = (rst) ? '0 : (Idrstall);

endmodule
```

Appendix B - Register File

1) reg_file.sv

```
//This module is created for Lab 1 Task 2, where we design a 16x32
//register file with 2 asynchronous read ports and 1 synchronous
//write port. The inputs are the write and two read addresses,
//the write data and if write is enabled. The outputs are the data
//at the specified read addresses.
//
//Inputs: clk, wr_en, write_data, write_addr, read_addr1, read_addr2
//Outputs: read_data1, read_data2
module reg_file(clk, wr_en, write_data, write_addr, read_addr1,
                                          read_addr2, read_data1, read_data2);
      input logic clk, wr_en;
      input logic [31:0] write_data;
      input logic [3:0] write_addr;
      input logic [3:0] read_addr1, read_addr2;
      output logic [31:0] read_data1, read_data2;

      //16x32 registers holding data as memory
      logic [15:0][31:0] memory;

      //writes data to specified address on memory at positive edge of
      //clock cycles (synchronous)
      always_ff @(posedge clk) begin
            //only write if wr_en is true
            if (wr_en) begin
                  memory[write_addr] <= write_data;
            end
      end

      //gives the data immediately when both read data is specified (asynch)
      assign read_data1 = memory[read_addr1];
      assign read_data2 = memory[read_addr2];
endmodule
```

Appendix C - ALU

1) FullAdder.sv

```
// Adds three 1-bit inputs, A, B, and cin. Produces the sum and carry out.
//
// Inputs: A, B, cin (carry in)
// Outputs: sum, cout (carry out)
module FullAdder (A, B, cin, sum, cout);

	input logic A, B, cin;
	output logic sum, cout;

	assign sum = A ^ B ^ cin;
	assign cout = A&B | cin & (A^B);

endmodule
```

2) adder.sv

```
// Adds two N-bit numbers and produces the output and carry out.
// Generates an instantiations of the FullAdder module for each
// of the N bits. The initial carry in is used for the first
// Full Adder.
//
// Inputs: a, b, cin (initial carry in)
// Outputs: result, cout0 (final carry out)
module adder #(parameter N = 3) (
	input logic [N-1:0] a, b,
	input logic cin,
	output logic [N-1:0] result,
	output logic cout0
	);

	// cin and cout
	wire [N-1:0]cout;
	assign cout0 = cout[N-1];

	genvar i;
	generate
		for(i=0; i<N; i++) begin : Adders
			if (i==0) FullAdder FA0 (.A(a[i]), .B(b[i]), .cin(cin),
.sum(result[i]), .cout(cout[i]));
			else FullAdder FA (.A(a[i]), .B(b[i]), .cin(cout[i-1]),
.sum(result[i]), .cout(cout[i]));
		end
	endgenerate
```

```
endmodule
```

3) alu.sv

```
// This module holds the ALU for lab 1 Task 3 which operates on two N-bit
inputs.
// This ALU is able to change operations based on the 2-bit "ALUControl"
input. The
// possible operations are: Addition (00), Subtraction (01), AND (10), as well
as
// OR (11). Based on the given input, this module will output the correct
result of
// the selected operation. This module additionally outputs flags (within the
"ALUFlags"
// variable) when the result is negative (ALUFlags[3]) and result is zero
([2]). When
// adding or subtracting, the ALUFlags[1] and ALUFlags[0] will show whether
the adder
// produced a carry out or experienced an overflow, respectively.
//
// Inputs: a, b, ALUControl
// Outputs: Result, ALUFlags
module alu #(parameter N = 32)(

      // Inputs a and b (N-bit)
      input logic [N-1:0] a, b,

      // Controls the operation
      input logic [1:0] ALUControl,

      // Output result from operation
      output logic [N-1:0] Result,

      // ALU flags:
      // [0] = Adder results in overflow
      // [1] = Adder produced a carry out
      // [2] = Result is 0
      // [3] = Result is negative
      output logic [3:0] ALUFlags
      );

      // Results from each operation
      logic [N-1:0] sum, AND, OR, b_new;

      // Cout for adder
      logic cout;
```

```
	// add/sub b value MUX
	always_comb begin
		if (ALUControl[0]) b_new = ~b; // If subtracting, use ~b
		else b_new = b;                // If adding, use b
	end

	// Instantiate add module
	// Uses Full Adders to add each individual bits together. Produces a
Cout value.
	adder #(.N(N)) adding (.a, .b(b_new), .cin(ALUControl[0]), .result(sum),
.cout0(cout));

	// Store result of ANDing
	assign AND = a & b;

	// Store result from ORing
	assign OR = a | b;

	// Ending result MUX
	always_comb begin
		case(ALUControl)
			2'b10: Result = AND;
			2'b11: Result = OR;
			default: Result = sum;
		endcase
	end

	// ALU Flags output circuit
	assign ALUFlags[0] = ~( (ALUControl[0]) ^ (a[N-1]) ^ (b[N-1]) ) & (
(a[N-1]) ^ (sum[N-1]) ) & ~( ALUControl[1] );
	assign ALUFlags[1] = ~ALUControl[1] & cout;
	assign ALUFlags[2] = (Result == 0);
	assign ALUFlags[3] = Result[N-1];

endmodule
```

Appendix D - TestingFile

1) TestingFile.txt

```
// ADD R - 1110000001000AAAADDDD00000000BBBB
// ADD I - 1110001010000AAAADDDD0000IIIIIIII
// SUB R - 1110000000100AAAADDDD00000000BBBB
// SUB I - 1110001001000AAAADDDD0000IIIIIIII
// CMP R - 1110000001010AAAADDDD00000000BBBB
// CMP I - 1110001001010AAAADDDD0000IIIIIIII
// LDR   - 1110010110010AAAADDDDIIIIIIIIIIII
// STR   - 1110010110000AAAADDDDIIIIIIIIIIII
// BXX   - COND1010IIIIIIIIIIIIIIIIIIIIIIII

// Equal            - COND = 0000
// Not Equal        - COND = 0001
// Greater or Equal - COND = 1010
// Greater          - COND = 1100
// Less or Equal    - COND = 1101
// Less             - COND = 1011


// FIRST TEST FOR PIPELINE REGISTERS !


// Only adds values that do not cause hazards

/*
11100000100011110000000000001111 // ADD R0, R15, R15   PC=0   R0=16
11100010100011110001000000001010 // ADD R1, R15, #10   PC=4   R1=22
11100000100011110010000000001111 // ADD R2, R15, R15   PC=8   R2=32
11100010100011110011000000001100 // ADD R3, R15, #12   PC=12  R3=32
11100000100000000100000000000001 // ADD R4, R0,  R1    PC=16  R4=38
11100010100000000101000000000001 // ADD R5, R0,  #1    PC=20  R5=17
11100000100011110000000000001111 // ADD R0, R15, R15   PC=24  R0=64
11100000100011110000000000001111 // ADD R0, R15, R15   PC=28  R0=72
11100000100011110000000000001111 // ADD R0, R15, R15   PC=32  R0=80
11100000100011110000000000001111 // ADD R0, R15, R15   PC=36  R0=88
11100000100011110000000000001111 // ADD R0, R15, R15   PC=40  R0=96
11100000100011110000000000001111 // ADD R0, R15, R15   PC=44  R0=104
*/


// SECOND TEST FOR PIPELINE REGISTERS !


// Tests Forwarding
// Only adds two values together that would cause a data hazard
```

```
/*
11100000100011110000000000001111 // ADD R0, R15, R15   PC=0   R0=16
11100010100000000001000000001010 // ADD R1, R0,  #10   PC=4   R1=26
11100000100000000010000000000001 // ADD R2, R0,  R1    PC=8   R2=40
11100010100011110000000000000000 // ADD R0, R15, #0    PC=12  R0=20
11100010100011110000000000000000 // ADD R0, R15, #0    PC=16  R0=24
11100010100011110000000000000000 // ADD R0, R15, #0    PC=20  R0=28
11100010100011110000000000000000 // ADD R0, R15, #0    PC=24  R0=32
11100010100011110000000000000000 // ADD R0, R15, #0    PC=28  R0=36
*/


// THIRD TEST FOR PIPELINE REGISTERS !


// Tests stalling
// Uses multiple operations to properly test

/*
11100010100011110000000000000000 // ADD R0, R15, #0     PC=0   R0=8
11100101100000000000000000000000 // STR R0, [R0, #0]    PC=4   STR
11100010100011110001000000000000 // ADD R1, R15, #0     PC=8   R1=16
11100101100100000001000000000000 // LDR R1, [R0, #0]    PC=12  R1=8
11100010100000010001000000000001 // ADD R1, R1,  #1     PC=16  R1=9
11100101100100000010000000000000 // LDR R2, [R0, #0]    PC=20  R2=8
11100010100000010001000000000010 // ADD R1, R2,  #2     PC=24  R1=10
11100101100100000011000000000000 // LDR R3, [R0, #0]    PC=28  R3=8
11100010100000011001000000000011 // ADD R1, R3,  #3     PC=32  R1=11
11100101100100000100000000000000 // LDR R4, [R0, #0]    PC=36  R4=8
11100010100001000001000000000100 // ADD R1, R4,  #4     PC=40  R1=12
*/


// FOURTH TEST FOR PIPELINE REGISTERS !


// Tests branching and flag conditions
// Uses SUBS (CMP) to store flags and BXX to branch

/*
11100010100011110000000000000000 // ADD R0, R15, #0   PC=0   R0=8  BACK
11100010010100000001000000001000 // CMP R1, R0,  #8   PC=4   R1=0
00011010000000000000000000000011 // BNE FORWARD       PC=8   PC=28
11100000100011110010000000001111 // ADD R2, R15, R15  PC=12  R2=40
11100010100011110000000000000000 // ADD R0, R15, #0   PC=16
11100010100011110000000000000000 // ADD R0, R15, #0   PC=20
11100010100011110000000000000000 // ADD R0, R15, #0   PC=24
11100010100011110000000000000000 // ADD R0, R15, #0   PC=28         FORWARD
```

```
11100001010001111000000000000000 // ADD R0, R15, #0   PC=32
11101010111111111111111111111110 // B LOOP            PC=36         LOOP
*/
```

// FIFTH TEST FOR PIPELINE REGISTERS !

```
// Tests all functions in one test
// Adds more instructions to the memfile3 provided

/*
11100000010011110000000000001111    //  MAIN    SUB  R0 R15 R15    PC=0   R0=0
11100010100000000001000000000001    //          ADD  R1 R0 #1      PC=4   R1=1
11100001100000000010000000000001    //          ORR  R2 R0 R1      PC=8   R2=1
11100010100000000010000000000010    //          ADD  R2 R0 #2      PC=12  R2=2
11100010010100100000000000000000    //       SUBS R0 R2 #0      PC=16  R0=2
00001010000000000000000000000001 //       BEQ  TAG1          PC=20  PC'=24
11100000000000100010000000000000    //          AND  R2 R2 R0      PC=24  R2=2
11100000000000100001000000000000    //          AND  R1 R2 R0      PC=28  R1=2
11100000100000011001000000000000    //  TAG1    ADD  R9 R1 R0      PC=32  R9=4
11100101100000001001000000001001    //          STR  R9 [R0, #9]   PC=36  STR
11100101100100000011000000001001    //          LDR  R3 [R0, #9]   PC=40  R3=4
11100000000000110010000000000010    //  REDO    AND  R2 R3 R2      PC=44  R2=0
R2=4
11100000010100100010000000000000    //          SUBS R2 R2 R0      PC=48
R2=-2   R2=2
10111010111111111111111111111100 //             BLT REDO           PC=52
PC'=44  PC'=56
11101010111111111111111111111110 //  LOOP B LOOP            PC=56
PC'=56
*/
```