Justin Sim and Mina Gao
EE 469
April 22, 2024
Lab 3 Report

Procedure

In this lab, we are tasked to extend the simple single-cycle processor developed in lab 2 into a 5 stage pipelined processor that manages multiple instructions at different stages of execution concurrently. Which involves adding pipeline to enhance performance while addressing the complexities introduced by this pipelining, such as managing datapath and control hazards.

We began by implementing the five-stage pipeline: Fetch, Decode, Execute, Memory, and Writeback by following the 5 Stage Pipelined Processor diagram shown below. Our focus was on ensuring that the register file could handle data correctly between stages and that branching decisions were handled in the execute stage. Each stage of the pipeline operates in parallel with the others, with each stage completing part of an instruction in one clock cycle.

To manage this parallel operation, pipeline registers are placed between each stage. These flip-flops store the output of one stage at the end of the current clock cycle, which then becomes the input for the next stage in the following clock cycle. This setup ensures that each stage can operate independently and concurrently on different parts of different instructions. We find forwarding crucial for managing dependencies between instructions at different stages. We identified the scenarios where data could be forwarded effectively from the memory and write-back stages to earlier stages to resolve data dependencies and maintain correct execution.

Finally, we tested our implementation on ModelSim using the provided test program by varying the instructions in memfile.dat, memfile2.dat, and memfile3.dat. This allowed us to verify the correctness of our pipelined processor design under different instruction sequences.

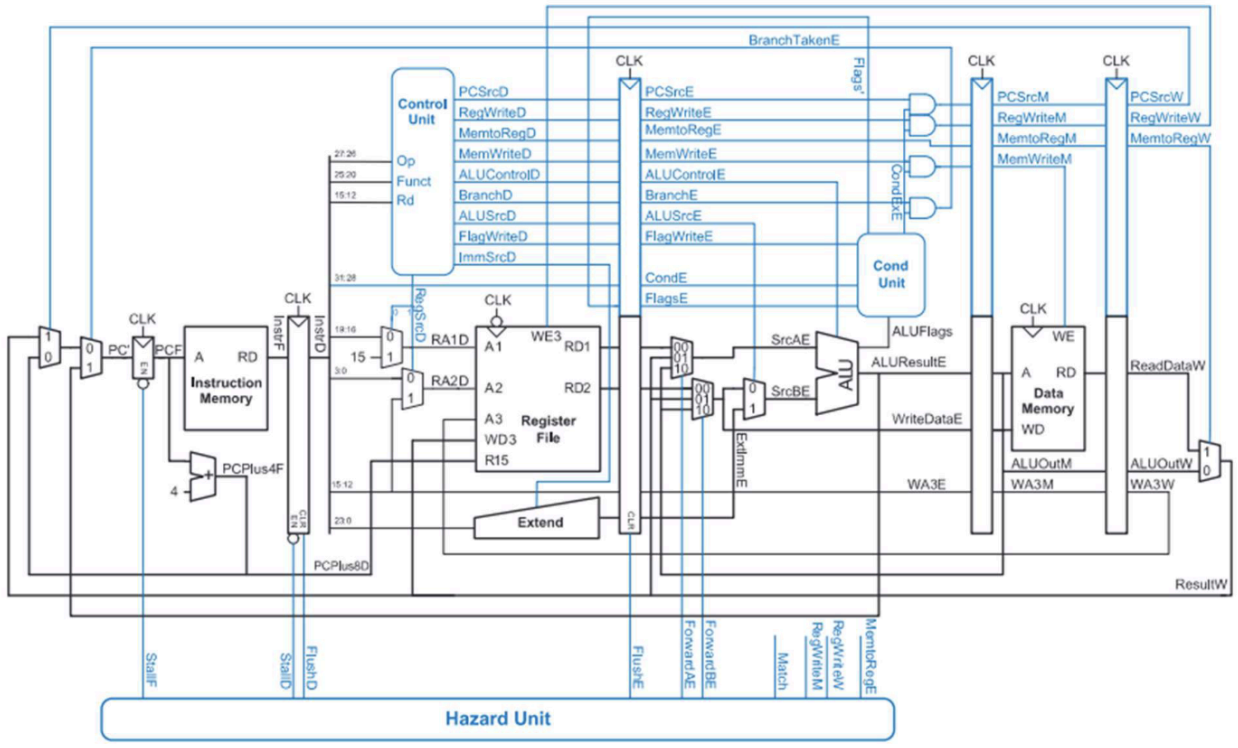


Figure_1: ARM 5 Stage Pipelined Processor

Results

For most robust testbench, the simulation waveforms embedded into the results sections show these cases

1. **An example of forwarding from the memory stage to the execute stage.**
2. **An example of forwarding from the writeback stage to the execute stage.**
3. **An example of stalling for a memory instruction.**
4. **An example of flushing for a branch instruction**

We have included the following additional programs used to test the processor:

5. **Memfile2.dat test cases from Lab2**
6. **ModelSim Verification of Pipeline Registers**

With these 6 test cases, we have verified that the pipelined processor can accomplish the same results as a single-cycle CPU, but with higher throughput and reduced overhead than the single-cycle processor.

1 and 2. Forwarding:

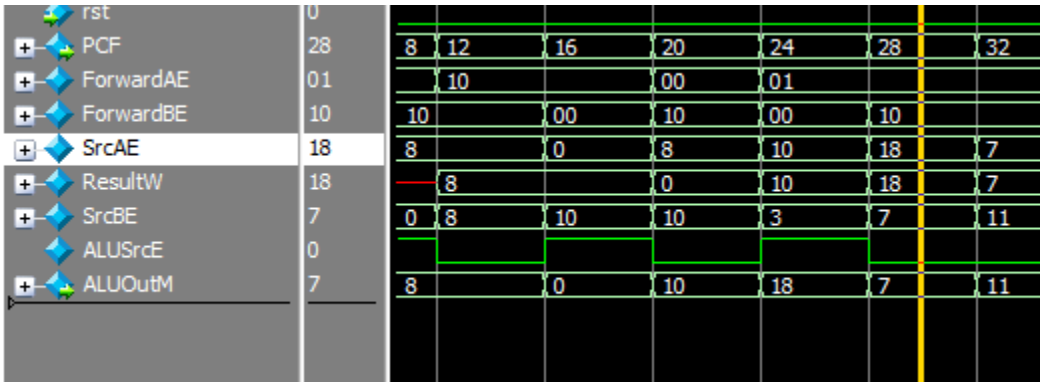


Figure_7: ModelSim arm.sv - Forwarding (BOTH memory to execute and writeback to execute cases)

The figure above shows an example of forwarding from memory to execute. ForwardAE is 01 meaning that SrcAE (the input for ALU in execute stage) should be getting it's value from ResultW (the result of the previous instruction in writeback stage) this occurs when an instruction uses a previously targeted register value, such that the CPU must point to a later stage in the cycle for its current execution input.

The figure also shows an example of forwarding memory from writeback to execute. ForwardBE is 10, meaning that SrCBE should get its value from ALUOutM (given ALUSrcE is 0), which it shows that it does correctly. The reason why we need this is similar to the point stated above, where values must be forwarded when pointing to a previously manipulated register is necessary.

This example conveniently demonstrates both scenarios, and verifies completely the forwarding capabilities of our pipelined CPU.

3. Stalling for Memory

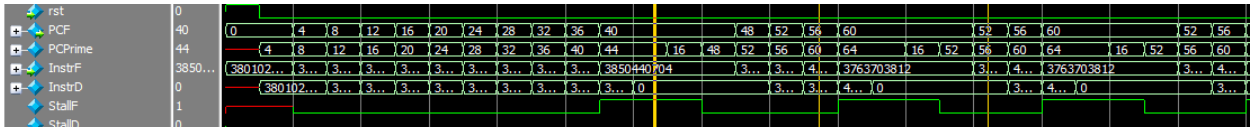


Figure_7: ModelSim arm.sv - Memfile2 StallF and PCF Testbench Results

The figure above shows an example of stalling for a memory instruction. We can see that StallF becomes High and the PCF is equal to itself, demonstrating that the CPU is waiting for the memory instruction to update. The reason this occurs is because sometimes the CPU must read from a register in which the previously stored value is at that register. Also, it is useful in branching when needing to skip PC.

3. Flush for Branch

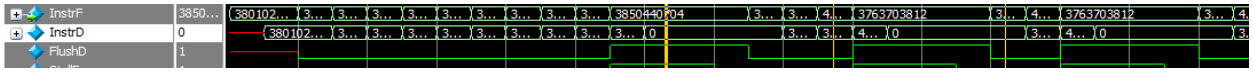


Figure_7: ModelSim arm.sv - Memfile2 FlushD and InstrD Testbench Results

The figure above shows InstrD set equal to 0 when FlushD is high, this is important because it helps maintain the accuracy of the processor's operation. Flushing the pipeline clears any instructions that may conflict with exception processing, allowing the exception handling routine to start fresh. It prevents any potentially incorrect/half-decoded instruction from moving to the next stage of the pipeline.

5. Memfile2 Verification

```
  else                                          $fatal( Task 1 Failed at: %t , $time);
assert(cpu.processor.u_reg_file.mem[8] == 32'hb) $display("Task 1 Passed R8");
else                                           $display("Task 1 Failed R8");

assert(cpu.processor.u_reg_file.mem[7] == 32'hb) $display("Task 1 Passed R7");
else                                           $display("Task 1 Failed R7");

assert(cpu.processor.u_reg_file.mem[6] == 32'hf) $display("Task 1 Passed R6");
else                                           $display("Task 1 Failed R6");

assert(cpu.processor.u_reg_file.mem[5] == 32'hb) $display("Task 1 Passed R5");
else                                           $display("Task 1 Failed R5");

assert(cpu.processor.u_reg_file.mem[4] == 32'h7) $display("Task 1 Passed R4");
else                                           $display("Task 1 Failed R4");

assert(cpu.processor.u_reg_file.mem[3] == 32'h12)$display("Task 1 Passed R3");
else                                           $display("Task 1 Failed R3");

assert(cpu.processor.u_reg_file.mem[2] == 32'ha) $display("Task 1 Passed R2");
else                                           $display("Task 1 Failed R2");

assert(cpu.processor.u_reg_file.mem[1] == 32'h0) $display("Task 1 Passed R1");
else                                           $display("Task 1 Failed R1");

assert(cpu.processor.u_reg_file.mem[0] == 32'h8) $display("Task 1 Passed R0");
else                                           $display("Task 1 Failed R0");
```

Figure_7: Figure_7: arm.sv memfile2.dat Testbench Verification

```
# Task 1 Passed R8
# Task 1 Passed R7
# Task 1 Passed R6
# Task 1 Passed R5
# Task 1 Passed R4
# Task 1 Passed R3
# Task 1 Passed R2
# Task 1 Passed R1
# Task 1 Passed R0
# ** Note: $stop    : ./testbench.sv(76)
#    Time: 5200 fs  Iteration: 1  Instance: /testbench
```

Figure_7: ModelSim arm.sv - memfile2.dat Terminal Verification

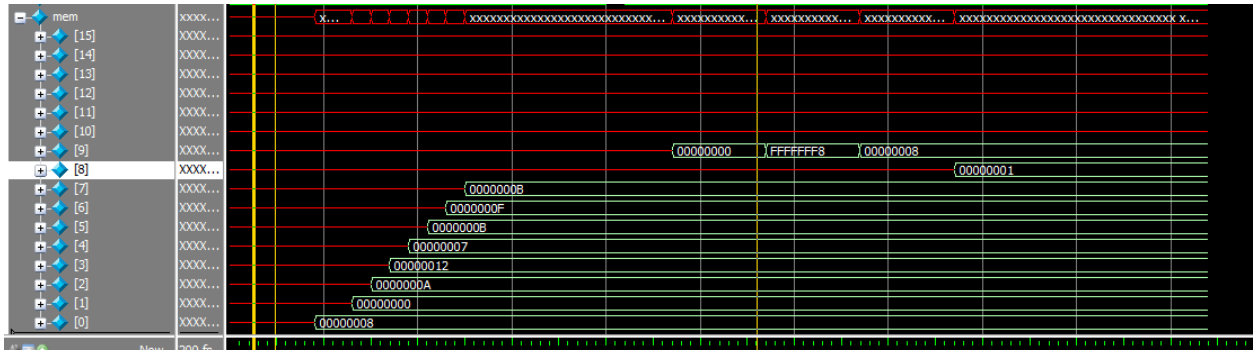


Figure_7: ModelSim arm.sv - memfile2.dat Waveform Verification

The 3 figures above show the ModelSim waveform output of memfile2.dat with the Pipelined CPU. We can see that Registers 0-7 are updated to the correct values as expected.

6. Pipeline Register Datapath Logic Verification

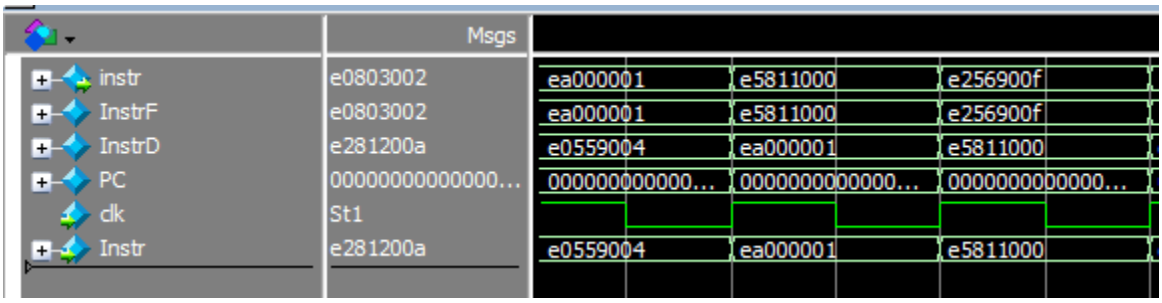


Figure_7: Fetch to Decoder Pipeline Datapath Logic ModelSim

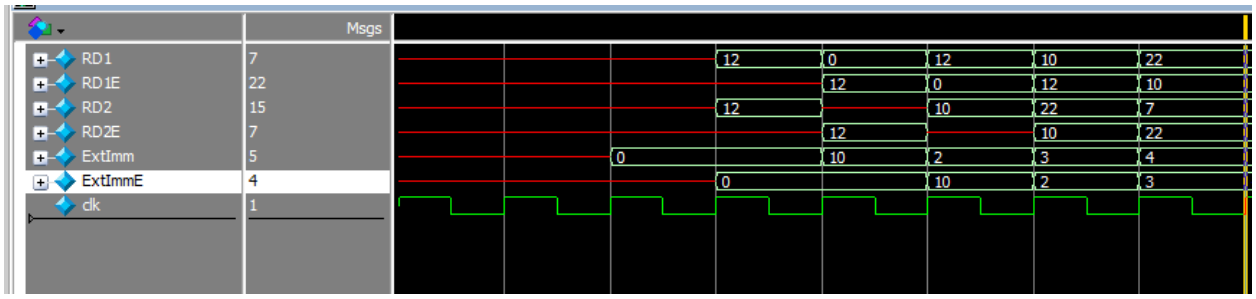


Figure_7: Decoder to Execute Pipeline Datapath Logic ModelSim

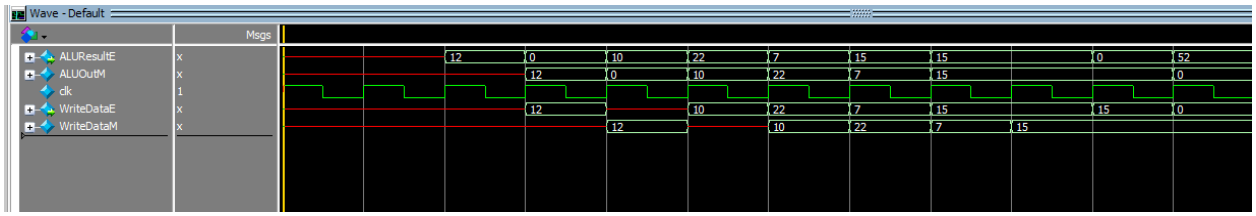


Figure_7: Execute to Memory Datapath Logic ModelSim

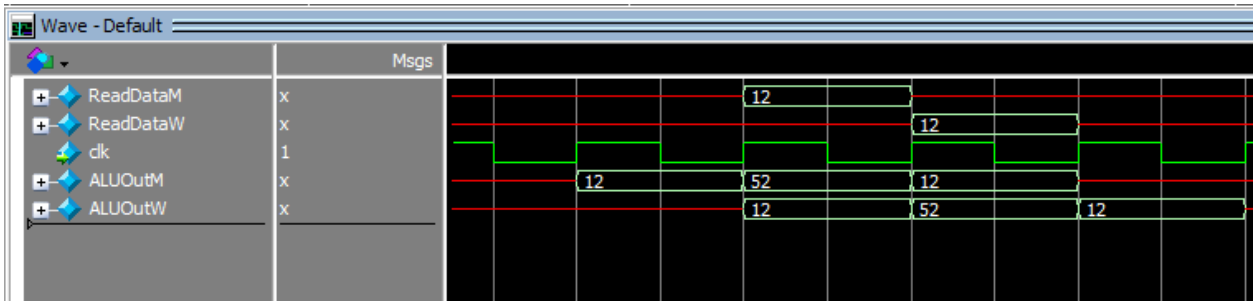


Figure_7: Memory to Writeback Pipeline Datapath Logic ModelSim

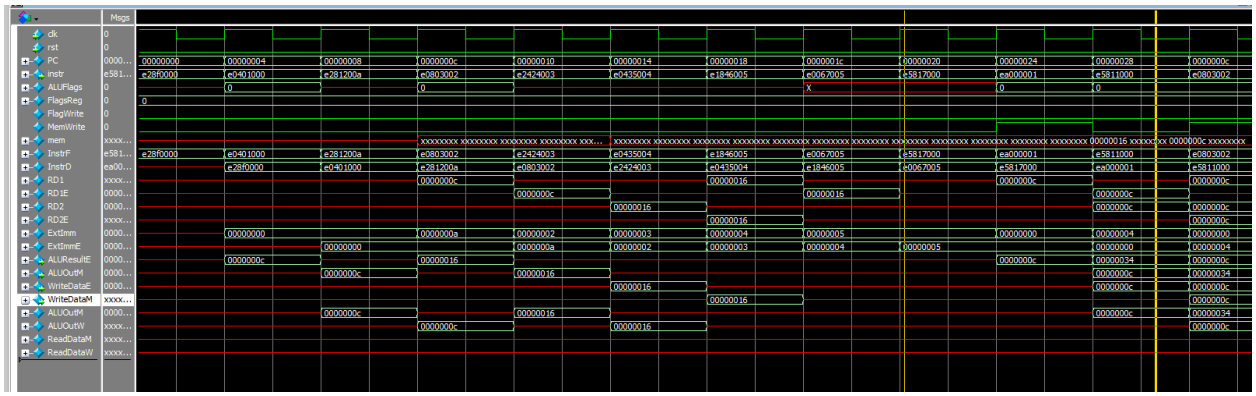
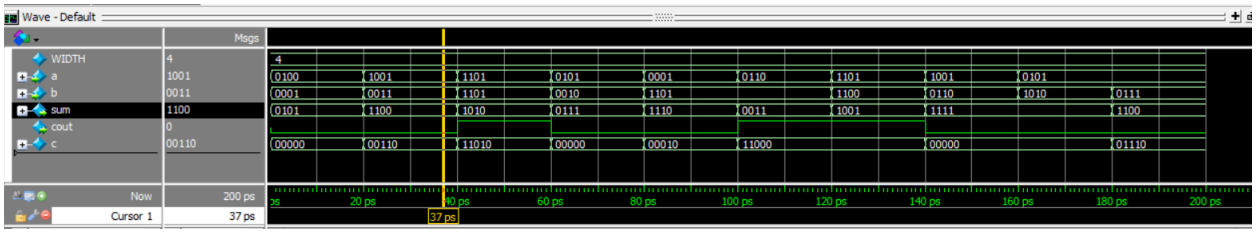


Figure_7: Full Pipeline Datapath Logic ModelSim

The 4 figures above demonstrate the pipeline register functionality for the datapath logic. We can see the flow of logic through each of the registers at one clock cycle most clearly in the last figure. The registers are important for holding datapath logic in place to modularize the CPU instructions and allow for variables to be used intersectional within the CPU stages.

For reference, all testbench code is provided in the **Appendix** section.

CarryRippleAdder Testbench



```
# .main_pane.objects.interior.cs.body.tree
# SUCCESSFUL ADDITION
#  a:  4,b:  1,sum:  5,cout: 0
# SUCCESSFUL ADDITION
#  a:  9,b:  3,sum: 12,cout: 0
# SUCCESSFUL ADDITION
#  a: 13,b: 13,sum: 10,cout: 1
# SUCCESSFUL ADDITION
#  a:  5,b:  2,sum:  7,cout: 0
# SUCCESSFUL ADDITION
#  a:  1,b: 13,sum: 14,cout: 0
# SUCCESSFUL ADDITION
#  a:  6,b: 13,sum:  3,cout: 1
# SUCCESSFUL ADDITION
#  a: 13,b: 12,sum:  9,cout: 1
# SUCCESSFUL ADDITION
#  a:  9,b:  6,sum: 15,cout: 0
# SUCCESSFUL ADDITION
#  a:  5,b: 10,sum: 15,cout: 0
# SUCCESSFUL ADDITION
#  a:  5,b:  7,sum: 12,cout: 0
# ** Note: $stop    : ./CarryRippleAdder.sv(51)
#    Time: 200 ps  Iteration: 0  Instance: /CarryRippleAdder_tb
```

Figure_12: ModelSim CarryRippleAdder Testbench (WIDTH = 4)

The Figure above shows the successful verification of CarryRippleAdder with 4 bit inputs. The testbench gives random input values to ensure expected and unexpected cases. A verification code was written and can be referred to in **Appendix** Section, it ensures that the desired output is given at the correct time. As evident in the Terminal, we can see that the relevant test case ran successfully.

Arithmetic Logic Unit (ALU) Testbench

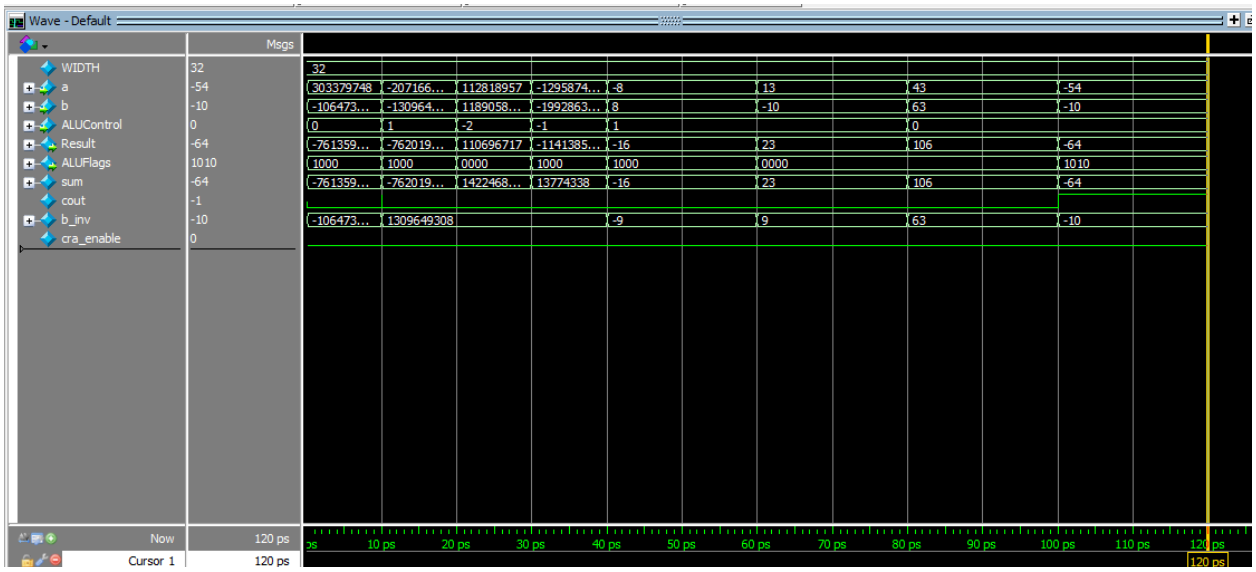


Figure_13: ModelSim ALU Testbench (Constrained Random Functional Test)

The Figure above shows the successful verification of each ALU function - add, subtract, and, or. Additionally, the figure above verifies the overflow conditions for both add and subtract. A verification code was written and can be referred to in **Appendix** Section, it ensures that the desired output is given at the correct time. As evident in the Terminal, we can see that the relevant test case ran successfully.

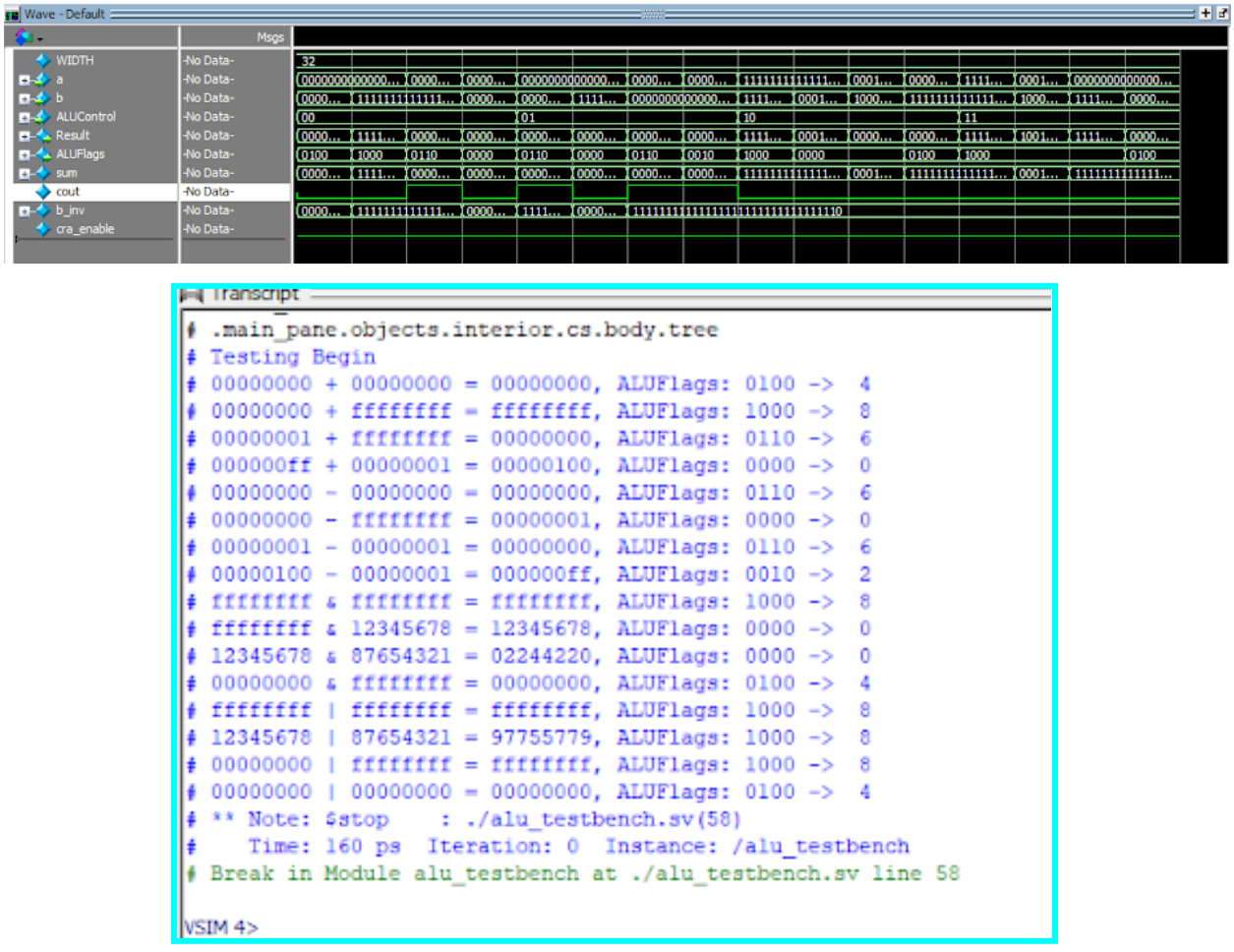


Figure_14: ModelSim ALU Testbench (Test Vector File)

The Figure above shows the result of the test vector file given in spec. It shows a successful verification, since it matches the manual calculations. A verification code was written and can be referred to in **Appendix** Section, it ensures that the desired output is given at the correct time. As evident in the Terminal, we can see that the relevant test case ran successfully, and we can further verify the output by comparing them to the given outputs in spec and when calculated by hand. Figure_11 also provides the fully filled resulting table of values for the empty test vector table given in the spec.

Appendix

Files changed from Lab2: arm.sv, top.sv,
Including: memfile3.dat, reg_file.sv

```
1   /*
2   Mina Gao and Justin Sim
3   4/18/2024
4   EE 469 Hussein
5   Lab 1,2,3
6
7   Reg File:
8   This module represents a register file, a storage component in a digital system composed of
    multiple registers.
9   It serves as a bank of storage locations, each typically capable of storing a single binary
    value.
10  The register file allows reading from and writing to individual registers based on
    specified control signals.
11  It facilitates data movement and storage within the system.
12
13  */
14  module reg_file(
15     input logic          clk, wr_en,
16     input logic [31:0]   write_data,
17     input logic [3:0]    write_addr, read_addr1, read_addr2,
18     output logic [31:0]  read_data1, read_data2
19  );
20
21     logic [15:0][31:0] mem;
22
23     // write operation
24     always_ff@(posedge clk) begin
25        if (wr_en) begin
26           mem[write_addr] <= write_data;
27        end
28     end
29
30     // read operations
31     assign read_data1 = mem[read_addr1];
32     assign read_data2 = mem[read_addr2];
33  endmodule
34
35  //-------------------------------------------------------------------------------
36  // Register File Testbench
37  // Covers 3 test scenarios given in spec to verify functionality
38  module reg_file_tb;
39
40     logic          clk, wr_en;
41     logic [31:0]   write_data;
42     logic [3:0]    write_addr, read_addr1, read_addr2;
43     logic [31:0]  read_data1, read_data2;
44
45    // Instantiate reg_file module
46    reg_file reg_file_inst(.*);
47
48    // Clock generation
49    parameter clock_period = 100;
50    initial begin clk <= 0;
51      forever #(clock_period /2) clk <= ~clk;
52    end
53
54    // Run Test Scenarios (uncomment to run each individually)
55    initial begin
56  //     write_test;
57         read_test;
58  //     write_read_test;
59         $stop;
60     end
61
62
    //-------------------------------------------------------------------------------------------
    -
63    // Test scenario 1: Write data is written the cycle after wr_en is asserted
64
65    // "Write data is written into the register file the clock cycle after wr_en is asserted"
66
    //-------------------------------------------------------------------------------------------
    -
67    task write_test; begin
68
69      // Test write operation
```

```
70        wr_en = 1;
71        write_addr = 2;
72        write_data = 32'hABCDEFFF;
73        @(posedge clk);
74
75        // Test read operation in the next cycle
76        wr_en       = 0;
77        write_addr  = 0;
78        write_data  = 0;
79        read_addr1  = 2;
80        @(posedge clk);
81
82        // Check if the written data is available in the next cycle
83        if(read_data1 != 32'hABCDEFFF) $fatal("Test Scenario 1 FAIL at" , $time);
84        else $display("Test Scenario 1 GOOD at" , $time);
85
86      end endtask : write_test
87
      //-------------------------------------------------------------------------------------------
      -
88      // Test scenario 2: Read data is updated the same cycle the address was provided
89
90      // "Read data is updated to the register data at an address the same cycle the address was
91      // provided. Do this for both read addresses and data outputs"
92
      //-------------------------------------------------------------------------------------------
      -
93      task read_test; begin
94
95        wr_en = 1;
96        write_addr = 2; write_data = 32'hAAAAAAAA;
97        @(posedge clk);
98
99        read_addr1 = 2; if (read_data1 != 32'hAAAAAAAA) $fatal("read_addr1: Test Scenario 2
      FAIL at", $time);
100                       else $display("read_addr1: Test Scenario 2 GOOD at" , $time); #5;
101       read_addr2 = 2; if (read_data2 != 32'hBBBBBBBB) $fatal("read_addr2: Test Scenario 2
      FAIL at", $time);
102                       else $display("read_addr2: Test Scenario 2 GOOD at" , $time); #5;
103
104     end endtask
105
      //-------------------------------------------------------------------------------------------
      -
106     // Test scenario 3: Read data is updated to write data the cycle after the address was
      provided
107
108     // "Read data is updated to write data at an address the cycle after the address was
      provided
109     // if the write address is the same and wr_en was asserted. Do this for both read addresses
110     // and data outputs"
111
      //-------------------------------------------------------------------------------------------
      -
112     task write_read_test; begin
113       // Write operation
114       wr_en = 1;
115       write_addr = 2;
116       write_data = 32'h12345678;
117       @(posedge clk);
118       // provide a read address to same write_addr
119       read_addr1 = 2;
120       @(posedge clk);
121       // change write data at same address
122       wr_en = 1;
123       write_addr = 2;
124       write_data = 32'hCCCCCCCC;
125       @(posedge clk);
126       @(posedge clk);
127       // Check if the read data 1 is updated to write data in the next cycle
128       if (read_data1 != 32'hCCCCCCCC) $fatal("write_read_test FAIL: read_data: %h, \n
      expected read_data: %h", read_data1, write_data);
129       else $display("read_addr1: Test Scenario 3 GOOD" );
130       //------------------------------
131       //--- Repeat for read_addr2--//
132       //------------------------------
```

```
133         // Write operation
134         wr_en = 1;
135         write_addr = 2;
136         write_data = 32'h12345678;
137         @(posedge clk);
138         // provide a read address to same write_addr
139         read_addr2 = 2;
140         @(posedge clk);
141         // change write data at same address
142         wr_en = 1;
143         write_addr = 2;
144         write_data = 32'hCCCCCCCC;
145         @(posedge clk);
146         @(posedge clk);
147         // Check if the read data 1 is updated to write data in the next cycle
148         if (read_data2 != 32'hCCCCCCCC) $fatal("write_read_test FAIL: read_data: %h, \n
        expected read_data: %h", read_data2, write_data);
149         else $display("read_addr1: Test Scenario 3 GOOD" );
150       end endtask
151
152     endmodule
```

```
1   /*
2   Justin Sim and Mina Gao
3   4/28/2024
4   EE 469 Hussein
5   Lab 3
6   */
7   /* top is a structurally made toplevel module. It consists of 3 instantiations, as well as
    the signals that link them.
8   ** It is almost totally self-contained, with no outputs and two system inputs: clk and rst.
    clk represents the clock
9   ** the system runs on, with one instruction being read and executed every cycle. rst is the
    system reset and should
10  ** be run for at least a cycle when simulating the system.
11  */
12
13  // clk - system clock
14  // rst - system reset. Technically unnecessary
15  module top(
16      input logic clk, rst
17  );
18
19      // processor io signals
20      logic [31:0] InstrF, InstrD;
21      logic [31:0] ReadDataM;
22      logic [31:0] WriteDataE;
23      logic [31:0] PC, ALUResultE;
24      logic [31:0] ALUOutM, WriteDataM;  // From Execute Pipeline
25      logic [31:0] ALUOutW, ReadDataW;   // For Writebak Pipeline
26      logic        MemWrite;
27
28      // Pipeline Registers for Writeback
29        always_ff @(posedge clk) begin
30          if (!rst) begin
31             ReadDataW <= ReadDataM;
32             ALUOutW <= ALUOutM;
33          end
34        end
35
36      // our single cycle arm processor
37      arm processor (
38          .clk        (clk        ),
39          .rst        (rst        ),
40          .InstrF      (InstrF     ),
41          .ReadDataW  (ReadDataW  ),
42          .WriteDataE (WriteDataE ),
43          .PCF         (PC         ),    // <- change
44          .ALUResultE (ALUResultE ),
45          .MemWrite   (MemWrite   ),
46          .ALUOutM    (ALUOutM    ),
47          .WriteDataM (WriteDataM )
48      );
49
50      // instruction memory
51      // contained machine code instructions which instruct processor on which operations to
    make
52      // effectively a rom because our processor cannot write to it
53      imem imemory (
54          .addr   (PC     ),
55          .instr  (InstrF )
56      );
57
58      // data memory
59      // containes data accessible by the processor through ldr and str commands
60      dmem dmemory (
61          .clk     (clk       ),
62          .wr_en   (MemWrite  ),
63          .addr    (ALUOutM   ),
64          .wr_data (WriteDataM),
65          .rd_data (ReadDataM )
66      );
67
68
69  endmodule : top
```

```
1   /*
2   Justin Sim and Mina Gao
3   4/28/2024
4   EE 469 Hussein
5   Lab 3
6
7   arm 5 stage pipelined processor
8   arm is the spotlight of the show and contains the bulk of the datapath and control logic.
9   This module is split into two parts, the datapath and control.
10  It simulates a simplified ARM processor with a 5-stage pipeline with
11  instruction fetch, decode, execute, memory access, and write back stages.
12  */
13
14  // clk - system clock
15  // rst - system reset
16  // InstrF - fetched 32 bit instruction from instruction memory
17  // ReadDataW - data read out of the register pipelining memory and writeback
18  // WriteDataE - data to be written to the dmem
19  // MemWrite - write enable to allowed WriteData to overwrite an existing dmem word
20  // PC - the current program count value, goes to imem to fetch instruciton
21  // ALUResultE - result of the ALU operation during EXECUTION, sent as address to the dmem
22
23  module arm (
24      input  logic        clk, rst,
25      input  logic [31:0] InstrF,
26      input  logic [31:0] ReadDataW,
27
28      output logic [31:0] WriteDataE,
29      output logic [31:0] PCF, ALUResultE,
30      output logic        MemWrite,
31      output logic [31:0] ALUOutM, WriteDataM  // From Execute Pipeline
32  );
33
34      // datapath buses and signals
35      logic [31:0] PCPrime, PCPlus4, PCPlus8, PCPlus8_temp, PC;  // pc signals, adding PCF
36
37  //  logic [ 3:0] RA1, RA2;                      // regfile input addresses
38  //    logic [31:0] RD1, RD2;                       // raw regfile outputs
39      logic [ 3:0] ALUFlags;                      // alu combinational flag outputs
40      logic [31:0] ExtImm, SrcA, SrcB;            // immediate and alu inputs
41      logic [31:0] ResultW;                       // computed or fetched value to be written
    into regfile or pc
42      logic [ 3:0] FlagsReg;             // register to store the flags from the most recent
    CMP command
43      logic [31:0] RA1E, RA2E;              // regfile inputs addresses
44      logic [31:0] ALUOutW;
45
46      // control signals
47      // adding FlagWrite to control if loading the FlagsReg, control signal for the decoder
48      logic PCSrc, MemtoReg, ALUSrc, RegWrite, FlagWrite;
49      logic [1:0] RegSrc, ImmSrc, ALUControl;
50      logic [5:0] Funct;
51      logic NoWrite = 0;
52
53      // fetch stage
54      logic StallF;
55
56      // decode stage
57      logic PCSrcD, RegWriteD, MemtoRegD, MemWriteD, BranchD=0, ALUSrcD, FlagWriteD, ImmSrcD,
    RegSrcD, StallD, FlushD;
58      logic [1:0] ALUControlD;
59      logic [3:0] RA1D, RA2D;
60      logic [31:0] InstrD;
61      logic [31:0] RD1D, RD2D;
62
63      // execute stage
64      logic PCSrcE, RegWriteE, MemtoRegE, MemWriteE, BranchE, ALUSrcE, FlushE;
65      logic [3:0] CondE, FlagsE, Flags, WA3E;
66      logic [1:0] ALUControlE, FlagWriteE;
67      logic [1:0] ForwardAE, ForwardBE;
68      logic CondExE, BranchTakenE;
69      logic [31:0] RD1E, RD2E, ExtImmE;                  // decoder pipeline outputs
70
71
72      // memory stage
73      logic PCSrcM, RegWriteM, MemtoRegM, MemWriteM;
```

```
74        logic [3:0] WA3M;
75
76        // writeback (back to register file WD3)
77        logic PCSrcW, RegWriteW, MemtoRegW;
78        logic [3:0] WA3W;
79
80        // Additional Floating Logic
81        logic [3:0] FlagsPrime;
82        logic PCS;
83
84        // PCSrc Control Unit Logic
85        assign PCS = ((InstrD[15:12] == 'd15) & RegWriteD) | BranchD;
86        assign PCSrcD = CondExE ? PCS : 0;
87
88
89
90        /* 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
91        ** 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
92        ** ports of the register file and alu take some time to study and understand the logic
      and flow of the datapath.
93        */
94        //------------------------------------------------------------------------------
95        //                                      DATAPATH
96        //------------------------------------------------------------------------------
97    ////////////////////////////////////////////////////////////////////////////////////////////////
      ////////////////////////
98                        // FETCH STAGE
99    ////////////////////////////////////////////////////////////////////////////////////////////////
      ////////////////////////
100
101       // update PCPrime according to branch condition
102      assign PC = PCSrcW ? ResultW : PCPlus4;
103      assign PCPrime = BranchTakenE ? ALUResultE : PC;
104      assign PCPlus4 = PCF + 'd4;                    // default value to access next instruction
105       // PCPlus8
106      always_ff @(posedge clk) begin
107         PCPlus8_temp <= PCPlus4 + 'd4; end
108            always_ff @(posedge clk) begin
109         PCPlus8 <= PCPlus8_temp; end
110
111       // update PCF based on the stall and reset
112       always_ff @(posedge clk) begin
113           if (rst)     PCF <= '0;
114      else if (StallF)  PCF <= PCF;
115           else         PCF <= PCPrime;
116       end
117
118       // FETCH registers
119       always_ff @(posedge clk) begin
120         if (FlushD)    InstrD <= '0;
121         else if (StallD)  InstrD <= InstrD;  // holds the old value
122         else        InstrD <= InstrF;
123      end
124
125
126   ////////////////////////////////////////////////////////////////////////////////////////////////
      ////////////////////////
127                           // DECODE STAGE
128   ////////////////////////////////////////////////////////////////////////////////////////////////
      ////////////////////////
129
130
131
132       // determine the register addresses based on control signals
133       // RegSrc[0] is set if doing a branch instruction
134       // RefSrc[1] is set when doing memory instructions
135       assign RA1D = RegSrc[0] ? 4'd15         : InstrD[19:16];
136       assign RA2D = RegSrc[1] ? InstrD[15:12] : InstrD[ 3: 0];
137
138       // -------------------------------------------------------
139       // Instantiate register
140       // Inputs: clk, RegWriteW, ResultW, WA3W, RA1, RA2
141       // Outputs: , RD2D
142       // -------------------------------------------------------
```

```
143         reg_file u_reg_file (
144             .clk        (!clk),
145             .wr_en      (RegWriteW),
146             .write_data(ResultW),
147             .write_addr(WA3W),
148             .read_addr1(RA1D),
149             .read_addr2(RA2D),
150             .read_data1(RD1D),
151             .read_data2(RD2D)
152         );
153 
154         // two muxes, put together into an always_comb for clarity
155         // determines which set of instruction bits are used for the immediate
156         always_comb begin
157             if      (ImmSrc == 'b00) ExtImm = {{24{InstrD[7]}},InstrD[7:0]};          // 8 bit
    immediate - reg operations
158             else if (ImmSrc == 'b01) ExtImm = {20'b0, InstrD[11:0]};                  // 12 bit
    immediate - mem operations
159             else                     ExtImm = {{6{InstrD[23]}}, InstrD[23:0], 2'b00}; // 24 bit
    immediate - branch operation
160         end
161 
162         //------------------------------
163         // DECODER PIPE
164         //------------------------------
165 
166         always_ff @(posedge clk) begin
167        if (~FlushE) begin
168           PCSrcE <= PCSrcD;
169              RegWriteE <= RegWriteD;
170           MemtoRegE <= MemtoRegD;
171              MemWriteE <= MemWriteD;
172           ALUControlE <= ALUControlD;
173           RD1E <= RD1D;
174              RD2E <= RD2D;
175              RA1E <= RA1D;
176              RA2E <= RA2D;
177           WA3E <= InstrD[15:12];
178           BranchE <= BranchD;
179              ALUSrcE <= ALUSrcD;
180           FlagWriteE <= FlagWriteD;
181              CondE <= InstrD[31:28];
182              FlagsE <= FlagsPrime;
183           ExtImmE <= ExtImm;
184        end else begin
185           PCSrcE <= '0;
186              RegWriteE <= '0;
187           MemtoRegE <= '0;
188              MemWriteE <= '0;
189           ALUControlE <= '0;
190           RD1E <= '0;
191              RD2E <= '0;
192              RA1E <= '0;
193              RA2E <= '0;
194           WA3E <= '0;
195           BranchE <= '0;
196              ALUSrcE <= '0;
197           FlagWriteE <= '0;
198              CondE <= '0;
199              FlagsE <= '0;
200           ExtImmE <= '0;
201    //        $display("FlushE is HIGH at time %d", $time);
202        end
203         end
204 
205 
206         // Additional Control logic assignment
207         assign Funct = InstrD[25:20];
208 
209 
210    ////////////////////////////////////////////////////////////////////////////////////////////
       ////////////////////////
211                         // EXECUTE STAGE
212    ////////////////////////////////////////////////////////////////////////////////////////////
       ////////////////////////
213 
```

```
214       // ------------------------------------------------------------------
215       // Data forwarding logic
216       // check if execute stage register matches Memory stage register
217       // check if execute stage register matches Writeback stage register
218       // ------------------------------------------------------------------
219       logic Match_1E_M, Match_2E_M, Match_1E_W, Match_2E_W;
220       assign Match_1E_M = (RA1E == WA3M);
221       assign Match_2E_M = (RA2E == WA3M);
222
223       assign Match_1E_W = (RA1E == WA3W);
224       assign Match_2E_W = (RA2E == WA3W);
225
226
227       always_comb begin
228         // ForwardAE
229         if (Match_1E_M & RegWriteM)      ForwardAE = 2'b10;
230         else if (Match_1E_W & RegWriteW) ForwardAE = 2'b01;
231         else                             ForwardAE = 2'b00;
232
233         // ForwardBE
234         if (Match_2E_M & RegWriteM)      ForwardBE = 2'b10;
235         else if (Match_2E_W & RegWriteW) ForwardBE = 2'b01;
236         else                             ForwardBE = 2'b00;
237       end
238
239       // ------------------------------------------------------------------
240       //Stalling and flushing logic
241       // Match_12D_E: check if either source register in the Decode stage the same
242       //          as the one being written in the Execute stage
243       // ldrStall: if a LDR in the Execute stage AND Match_12D_E
244       // Control Stalling Logic
245       // PCWrPendingF: if write to PC in Decode, Execute or Memory
246       // StallF: if PCWrPendingF
247       // FlushD: if PCWrPendingF OR PC is written in Writeback OR branch is taken
248       // FlushE: if branch is taken
249       // StallD: if ldrStall
250       // ------------------------------------------------------------------
251       logic Match_12D_E, PCWrPendingF, ldrStall;
252       assign Match_12D_E = ((RA1D == WA3E) || (RA2D == WA3E)) ? 1 : 0;
253       assign ldrStall = Match_12D_E & MemtoRegE;
254       assign PCWrPendingF = PCSrcD | PCSrcE | PCSrcM;
255       assign StallF = ldrStall | PCWrPendingF;
256       assign FlushD = PCWrPendingF | PCSrcW | BranchTakenE;
257       assign FlushE = ldrStall | BranchTakenE;
258       assign StallD = ldrStall;
259
260       // WriteDataE and SrcAE are outputs of the register file, wheras SrcEB is chosen
      between reg file output and the immediate
261       // Forwarding mux for SrcAE, SrcBE, and WriteDataE
262       logic [31:0] SrcAE, SrcBE;
263       always_comb begin
264      case (ForwardAE)
265         2'b00:      SrcAE = (RA1E == 'd15) ? PCPlus8 : RD1E; // substitute the 15th regfile
      register for PC
266         2'b01:      SrcAE = ResultW;
267         2'b10:      SrcAE = ALUOutM; // previous instruction
268         default: SrcAE = 0;
269      endcase
270      case (ForwardBE)
271         2'b00:      WriteDataE = (RA2E == 'd15) ? PCPlus8 : RD2E; // substitute the 15th
      regfile register for PC
272         2'b01:      WriteDataE = ResultW;
273         2'b10:      WriteDataE = ALUOutM; // previous instructionS
274         default: WriteDataE = 0;
275      endcase
276       end
277       assign SrcBE = ALUSrcE ? ExtImmE : WriteDataE; // determine alu operand to be either
      from reg file or from immediate
278
279       // ---------------------------------------
280       // Instantiate ALU
281       // Inputs: a, b, ALUControl
282       // Outputs: ResultW, ALUFlags
283       // ---------------------------------------
284       alu u_alu (
285           .a          (SrcAE),
```

```
286             .b          (SrcBE),
287             .ALUControl (ALUControlE),
288             .Result     (ALUResultE),
289             .ALUFlags   (ALUFlags)
290         );
291
292
293         always_ff@(posedge clk) begin
294           if (FlagWriteE[0]) FlagsPrime[1:0] <= ALUFlags[1:0];
295           if (FlagWriteE[1]) FlagsPrime[3:2] <= ALUFlags[3:2];
296         end
297
298         //--------------------------------
299         // Cond Unit
300         //--------------------------------
301
302         // Cond Logic
303         logic N;
304         logic Z;
305         logic C;
306         logic V;
307         assign N = ALUFlags[3];
308         assign Z = ALUFlags[2];
309         assign C = ALUFlags[1];
310         assign V = ALUFlags[0];
311         // Cond Assignment
312         // CondExE = instruction executed
313         always_comb begin
314        case(CondE)
315           4'b0000: CondExE = Z;
316           4'b0001: CondExE = !Z;
317           4'b0010: CondExE = C;
318           4'b0011: CondExE = !C;
319           4'b0100: CondExE = N;
320           4'b0101: CondExE = !N;
321           4'b0110: CondExE = V;
322           4'b0111: CondExE = !V;
323           4'b1000: CondExE = !Z & C;
324           4'b1001: CondExE = Z | !C;
325           4'b1010: CondExE = !(N ^ V);
326           4'b1011: CondExE = N ^ V;
327           4'b1100: CondExE = !Z & !(N ^ V);
328           4'b1101: CondExE = Z | (N ^ V);
329           4'b1110: CondExE = 1;
330           default:    CondExE = 0;
331        endcase
332         end
333
334
335         // Execute -> Memory Pipeline
336         always_ff @(posedge clk) begin
337
338           MemtoRegM <= MemtoRegE;
339           WA3M <= WA3E;
340           ALUOutM <= ALUResultE;
341           WriteDataM <= WriteDataE;
342
343           // AND CondExE
344           BranchTakenE <= BranchE & CondExE;
345           RegWriteM <= RegWriteE & CondExE & ~NoWrite;
346           MemWriteM <= MemWriteE & CondExE;
347           PCSrcM <= PCSrcE & CondExE;
348
349         end
350
351     ////////////////////////////////////////////////////////////////////////////////////////////////
        ////////////////////////
352                         // WRITEBACK
353     ////////////////////////////////////////////////////////////////////////////////////////////////
        ////////////////////////
354
355         // memory -> Writeback Pipelin
356         always_ff @(posedge clk) begin
357           PCSrcW <= PCSrcM;
358           RegWriteW <= RegWriteM;
359                MemtoRegW <= MemtoRegM;
```

```
360         WA3W <= WA3M;
361         ALUOutW <= ALUOutM;
362       end
363
364
365       // determine the result to run back to PC or the register file based on whether we used
      a memory instruction
366       assign ResultW = MemtoRegW ? ReadDataW : ALUOutW;     // determine whether final
      writeback result is from dmemory or alu
367
368
369   //    initial begin
370   //    if (StallF == StallD == FlushE == ldrStall) $display("Stall Control Logic is GOOD!");
371   //    else $display("Stall ERROR! at time %d", $time);
372   //
373   //    if (ldrStall == (Match_12D_E & MemtoRegE)) $display("ldrstall logic is GOOD!");
374   //       else $display("ldrstall ERROR! at time %d", $time);
375   //  end
376   //
377
378       /* The control conists of a large decoder, which evaluates the top bits of the
      instruction and produces the control bits
379       ** which become the select bits and write enables of the system. The write enables
      (RegWrite, MemWrite and PCSrc) are
380       ** especially important because they are represefntative of your processors current
      state.
381       */
382       //-------------------------------------------------------------------------------
383       //                                       CONTROL
384       //-------------------------------------------------------------------------------
385
386       // Control Unit
387       always_comb begin
388      casez (InstrD[31:20])
389
390           //------------------
391           // ADD (Imm or Reg)
392           //------------------
393           12'b111000?01000 : begin
394
395                    // Main Decoder
396                    MemtoRegD = 0;
397                    MemWriteD = 0;
398                    ALUSrcD   = InstrD[25];
399                    ImmSrc    = Funct[5] ? 'b00 : 'bXX;
400                    RegWriteD = 1;
401                    RegSrc    = Funct[5] ? 'bX0 : 'b00;
402
403                    // ALU Decoder
404                    ALUControlD = 'b00;
405                    FlagWriteD = Funct[0] ? 2'b11 : 2'b00;
406
407                    // $display("ADD at time %d", $time); // uncomment for verification
408           end // ADD
409
410           //------------------
411           // SUB (Imm or Reg)
412           //------------------
413           12'b111000?00100 : begin
414
415                    // Main Decoder
416                    MemtoRegD = 0;
417                    MemWriteD = 0;
418                    ALUSrcD   = InstrD[25];
419                    ImmSrc    = Funct[5] ? 'b00 : 'bXX;
420                    RegWriteD = 1;
421                    RegSrc    = Funct[5] ? 'bX0 : 'b00;
422
423                    // ALU Decoder
424                    ALUControlD = 'b01;
425                    FlagWriteD = Funct[0] ? 2'b11 : 2'b00;
426
427                    // $display("SUB at time %d", $time); // uncomment for verification
428           end // SUB
429
430           //------------------
```

```
431           // AND
432           //------------------
433           12'b111000000000 : begin
434
435                   // Main Decoder
436                   MemtoRegD = 0;
437                   MemWriteD = 0;
438                   ALUSrcD   = InstrD[25];
439                   ImmSrc    = Funct[5] ? 'b00 : 'bXX;
440                   RegWriteD = 1;
441                   RegSrc    = Funct[5] ? 'bX0 : 'b00;
442
443                   // ALU Decoder
444                   ALUControlD = 'b10;
445                   FlagWriteD = Funct[0] ? 2'b10 : 2'b00;
446
447   //              $display("AND at time %d", $time); // uncomment for veri..
448           end // AND
449
450           //------------------
451           // ORR
452           //------------------
453           12'b111000011000 : begin
454
455                   // Main Decoder
456                   MemtoRegD = 0;
457                   MemWriteD = 0;
458                   ALUSrcD   = InstrD[25];
459                   ImmSrc    = Funct[5] ? 'b00 : 'bXX;
460                   RegWriteD = 1;
461                   RegSrc    = Funct[5] ? 'bX0 : 'b00;
462
463                   // ALU Decoder
464                   ALUControlD = 'b11;
465                   FlagWriteD = Funct[0] ? 2'b10 : 2'b00;
466
467   //              $display("OR at time %d", $time);
468           end // ORR
469
470           //------------------
471           // LDR
472           //------------------
473           12'b111001011001 : begin
474
475                   // Main Decoder
476                   MemtoRegD = 1;
477                   MemWriteD = 0;
478                   ALUSrcD   = 1;
479                   RegWriteD = 1;
480                   RegSrc    = 'bX0;
481                   ImmSrc    = 'b01;
482
483                   // ALU Decoder
484                   ALUControlD = 'b00;
485                   FlagWriteD = 0;
486
487   //              $display("LDR at time %d", $time);
488           end
489
490           //------------------
491           // STR
492           //------------------
493           12'b111001011000 : begin
494
495                   // Main Decoder
496                   MemtoRegD = 'bX; // doesn't matter
497                   MemWriteD = 1;
498                   ALUSrcD   = 1;
499                   RegWriteD = 0;
500                   RegSrc    = 'b10;    // msb doesn't matter
501                   ImmSrc    = 'b01;
502
503                   // ALU Decoder
504                   ALUControlD = 'b00;  // do an add
505                   FlagWriteD = 0;
506
```

```
507   //              $display("STR at time %d", $time);
508         end
509
510         //------------------
511         // B
512         //------------------
513         12'b????1010???? : begin
514           case (InstrD[31:28])
515              // B, BEQ, BNE, BGE, BGT, BLE, BLT
516              4'b1110, 4'b0000, 4'b0001, 4'b1010, 4'b1100, 4'b1101, 4'b1011: begin
517
518                    // Main Decoder
519                    BranchD = 1;
520                    MemtoRegD = 0;
521                    MemWriteD = 0;
522                    ALUSrcD   = 1;
523                    RegWriteD = 0;
524                    RegSrc    = 'bX1;
525                    ImmSrc    = 'b10;
526
527                    // ALU Decoder
528                    ALUControlD = 'b00;
529                    FlagWriteD = 0;
530
531   //                 $display("B at time %d", $time);
532                    end
533           endcase
534         end // B
535
536         //------------------
537         // CMP
538         //------------------
539         12'b111000?00101 : begin
540
541                  // Main Decoder
542                  MemtoRegD = 0;
543                  MemWriteD = 0;
544                  ALUSrcD   = InstrD[25];
545                  RegWriteD = 0;
546                  RegSrc    = 'b00;
547                  ImmSrc    = 'b00;
548
549                  // ALU Decoder
550                  ALUControlD = 'b01;
551                  FlagWriteD = 2'b11;
552
553                  // NoWrite
554                  NoWrite = 1;
555
556   //             $display("CMP at time %d", $time);
557               end
558
559         //------------------
560         // Default Skin
561         //------------------
562         default: begin
563                  MemtoRegD = 0;
564                  MemWriteD = 0;
565                  ALUSrcD   = InstrD[25];
566                 RegWriteD = 1;
567                 RegSrc    = 'b00;
568                 ImmSrc    = 'b00;
569                 ALUControlD = 'b00;  // do an add
570                 FlagWriteD = 0;
571                 BranchD = 0;
572         end
573       endcase
574     end // always_comb
575
576   endmodule : arm
```

```
1   // ADD R - 1110_000_0100_0_AAAA_DDDD_0000_0000_BBBB
2   // ADD I - 1110_001_0100_0_AAAA_DDDD_0000_IIII_IIII
3   // SUB R - 1110_000_0010_0_AAAA_DDDD_0000_0000_BBBB
4   // SUB I - 1110_001_0010_0_AAAA_DDDD_0000_IIII_IIII
5   // CMP R - 1110_000_0010_1_AAAA_DDDD_0000_0000_BBBB
6   // CMP I - 1110_001_0010_1_AAAA_DDDD_0000_IIII_IIII
7   // AND   - 1110_000_0000_0_AAAA_DDDD_0000_0000_BBBB
8   // ORR   - 1110_000_1100_0_AAAA_DDDD_0000_0000_BBBB
9   // LDR   - 1110_010_1100_1_AAAA_DDDD_IIII_IIII_IIII
10  // STR   - 1110_010_1100_0_AAAA_DDDD_IIII_IIII_IIII
11  // COND_1010_IIII_IIII_IIII_IIII_IIII_IIII
12
13  // Equal            - COND = 0000
14  // Not Equal        - COND = 0001
15  // Greater or Equal - COND = 1010
16  // Greater          - COND = 1100
17  // Less or Equal    - COND = 1101
18  // Less             - COND = 1011
19
20  1110_000_0010_0_1111_0000_0000_0000_1111   //  MAIN SUB  R0 R15 R15
21  1110_001_0100_0_0000_0001_0000_0000_0001   //       ADD  R1 R0 #1
22  1110_000_1100_0_0000_0010_0000_0000_0001   //       ORR  R2 R0 R1
23  1110_001_0100_0_0000_0010_0000_0000_0010   //       ADD  R2 R0 #2
24  1110_001_0010_1_0010_0000_0000_0000_0000   //       SUBS R0 R2 #0
25  0000_1010_0000_0000_0000_0000_0000_0001       //       BEQ  TAG1
26  1110_000_0000_0_0010_0010_0000_0000_0000   //       AND  R2 R2 R0
27  1110_000_0000_0_0010_0001_0000_0000_0000   //       AND  R1 R2 R0
28  1110_000_0100_0_0001_1001_0000_0000_0000   //  TAG1 ADD  R9 R1 R0
29  1110_010_1100_0_0000_1001_0000_0000_1001   //       STR  R9 [R0, #9]
30  1110_010_1100_1_0000_0011_0000_0000_1001   //       LDR  R3 [R0, #9]
31  1110_000_0000_0_0011_0010_0000_0000_0010   //       AND  R2 R3 R2
```