Justin Sim and Mina Gao
EE 469
April 9, 2024
Lab 2 Report

Procedure

In this lab, we are tasked with building and simulating a simplified ARM single-cycle processor. It involves two main tasks: integrating the given modules to complete the processor and then extending its functionality with new instructions.

Task #1

In Task 1, we integrated our ALU and Register File from Lab 1 with other provided processor components. First, we inserted the reg_file and alu modules into the given arm processor's datapath as shown in Figure 1 & 2.

```
//-------------------------------------------
// Instantiate ALU
// inputs: a, b, ALUControl
// outputs: Result, ALUFlags
//-------------------------------------------
alu u_alu (
    .a          (SrcA),
    .b          (SrcB),
    .ALUControl (ALUControl),
    .Result     (ALUResult),
    .ALUFlags   (ALUFlags)
);
```

Figure_1: alu module integration in arm's datapath

```
//-----------------------------------------------------
// Instantiate reg_file
// inputs: clk, RegWrite, Result, Instr[15:12], RA1, RA2
// outputs: RD1. RD2
//-----------------------------------------------------
reg_file u_reg_file (
    .clk        (clk),
    .wr_en      (RegWrite),
    .write_data(Result),
    .write_addr(Instr[15:12]),
    .read_addr1(RA1),
    .read_addr2(RA2),
    .read_data1(RD1),
    .read_data2(RD2)
);
```

Figure_2: reg_file module integration in arm's datapath

We then tested it on ModelSim using the provided test program that includes a specific set of instructions like ADD, SUB, AND, ORR, LDR, STR, and B. Ensured that the simulation waveform includes specific signals (clk, reset, PC, Instr, ALUResult, WriteData, MemWrite, ReadData) that align with the provided ARM schematic shown below.

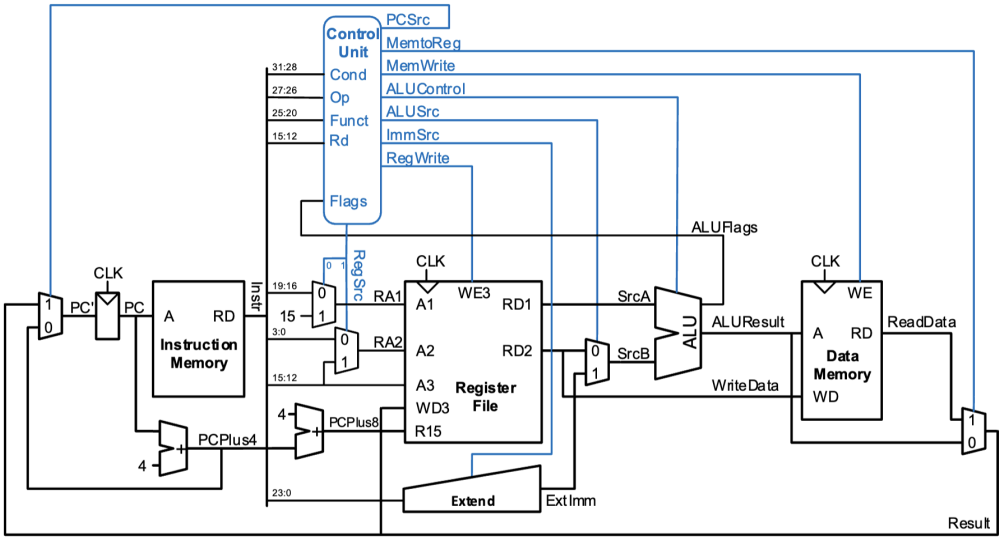


Figure_3: Single-cycle ARM processor

Then we have verified the register file contents and debugged the program with expected values we filled in Table 3 (as shown in Figure 4).

Cycle	PC	Instr	SrcA	SrcB	ALUResult	WriteData	ReadData	MemWrite	RegWrite	Result
1	00	ADD R0, R15, #0	8	0	8	Don't Care	X	0	1	8
2	04	SUB R1, R0, R0	8	8	0	8	X	0	1	0
3	08	ADD R2, R1, #10	0	A	A	Don't Care	X	0	1	A
4	12	ADD R3, R0, R2	8	A	12	A	X	Ø	1	12
5	16	SUB R4, R2, #3	A	3	7	Don't care	X	Ø	1	7
6	20	SUB R5, R3, R4	12	7	5	7	X	Ø	1	1011_2 = b
7	24	ORR R6, R4, R5	7	b	f	b	X	Ø	1	f
8	28	AND R7, R6, R5	f	b	b	b	X	Ø	1	b
9	32	STR R7, [R1, #0]	Ø	Ø	Ø	b	X	1	Ø	Ø
10	36	B SKIP	2C	4	30	0	X	Ø	Ø	30
11	48	LDR R8, [R1, #0]	Ø	Ø	Ø	X	b	Ø	1	b
12	52	B LOOP	3C	FFFFFFF8	34	X	X	Ø	Ø	34
13	52	"	"	"	"	"	"	"	"	"
14	"	"	"	"	"	"	"	"	"	"
15	"	"	"	"	"	"	"	"	"	"
16	"	"	"	"	"	"	"	"	"	"
17	"	"	"	"	"	"	"	"	"	"
18	"	"	"	"	"	"	"	"	"	"
19	"	"	"	"	"	"	"	"	"	"

Table 3. First nineteen cycles of executing memfile.dat

Figure_4: A complete version of Table 3

Task #2

For Task 2, we are expanding the processor's capabilities by adding new instructions to the existing ARM module. In order to do so, as shown in Figure 5 we added a new FlagsReg register to store flags(N, Z, C, V flags) from the most recent CMP command and updated the control logic to support the CMP instruction and the conditional branching instructions (BXX). Uses the flags to evaluate conditions for conditional execution (EQ, NE, GE, GT, LE, LT) and modify the branch instruction accordingly by using the condition mnemonics table provided in the lecture slide (Figure 6). To verify the module is working as expected, we used the given test program, memfile2.dat.

```
// FlagsReg will store the flags from the most recent CMP command
always_ff @(posedge clk)
  FlagsReg <= ALUFlags;
```

Figure_5: Sequential logic to ensure that the flag registers are updated with the most recent CMP command

$Cond_{3:0}$	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\bar{Z}$
0010	CS / HS	Carry set / Unsigned higher or same	C
0011	CC / LO	Carry clear / Unsigned lower	$\bar{C}$
0100	MI	Minus / Negative	N
0101	PL	Plus / Positive of zero	$\bar{N}$
0110	VS	Overflow / Overflow set	V
0111	VC	No overflow / Overflow clear	$\bar{V}$
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	$Z\ OR\ \bar{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\bar{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	$Z\ OR\ (N \oplus V)$
1110	AL (or none)	Always / unconditional	ignored

Figure_6: Condition Mnemonics table

Results

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

Task 1 Testbench:

In task 1, we facilitated assembly through the memfile.dat, meticulously laying out the instructions. Subsequently, we meticulously scrutinized the functional behavior, validating our predictions across 9 designated memory slots through assertive measures. Providing initial instructions with memfile1.dat and using **Figure 4** as a point of reference

```
  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");
```

```
# 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
```

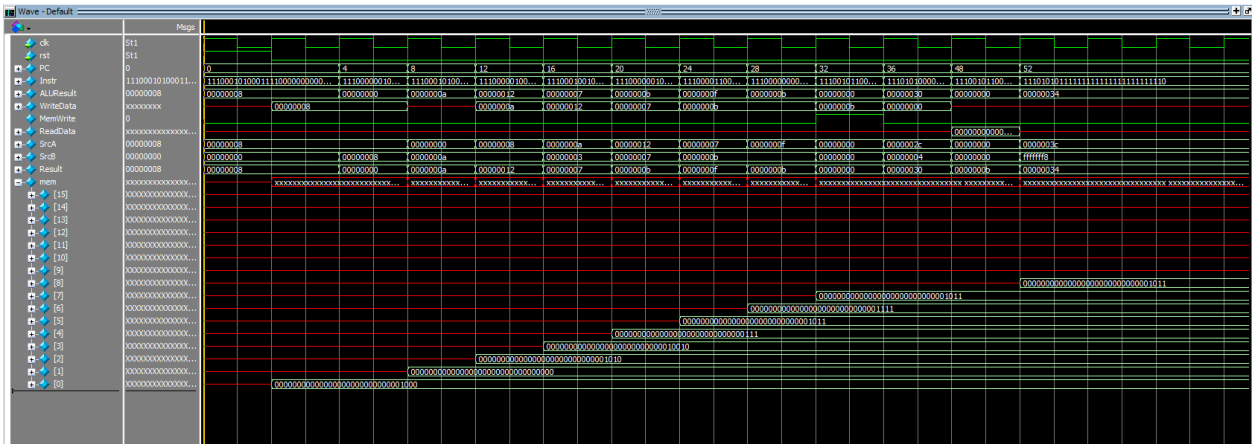


Figure_7: ModelSim arm.sv Task 1 Testbench Results

The meticulous verification tests yielded success, evident both in the terminal logs and the comprehensive waveforms. Notably, the simulation adhered precisely to our expectations, as evidenced by the iteration over PC = 52, aligning seamlessly with our anticipated results.

Task 2 Testbench:

In task 2, we facilitated assembly through the memfile.dat, meticulously laying out the instructions. Subsequently, we meticulously scrutinized the functional behavior, validating our predictions across 9 designated memory slots through assertive measures, providing the initial memory instructions with memfile2.dat.

Additionally, we've implemented a verification code to ensure that the program counter (PC) cycles at the appropriate intervals and completes the loop at the final PC address as intended.

```
        // task 2:
        assert(cpu.processor.u_reg_file.mem[8] == 32'd1)  $display("Task 2 Passed");
        else                                                $display("Task 2 Failed");
        // verify skips
        assert(cpu.processor.PC == 32'd40 || cpu.processor.PC == 32'd40) $display("Task 2 Failed SKIP at PC 36");
        else                                                             $display("Task 2 Passed SKIP at PC 36");
        assert(cpu.processor.PC == 32'd68)                               $display("Task 2 Failed SKIP at PC 64");
        else                                                             $display("Task 2 Passed SKIP at PC 64");
        assert(cpu.processor.PC == 32'd88)                               $display("Task 2 Failed SKIP at PC 84");
        else                                                             $display("Task 2 Passed SKIP at PC 84");
        assert(cpu.processor.PC == 32'd108)                              $display("Task 2 Failed SKIP at PC 104");
        else                                                             $display("Task 2 Passed SKIP at PC 104");

        // verify end loops
        repeat(5) assert(cpu.processor.PC == 32'd116)                    $display("Task 2 Passed LOOP at PC 116 !");
        else                                                             $display("Task 2 Failed LOOP at PC 116");
        $stop;
    end

endmodule
```

```
Transcript
#    Time: 0 fs  Iteration: 0  Instance: /testbench/cpu/processor/u_alu File: ./alu.sv
# .main_pane.objects.interior.cs.body.tree
# .main_pane.wave.interior.cs.body.pw.wf
# Task 2 Passed
# Task 2 Passed SKIP at PC 36
# Task 2 Passed SKIP at PC 64
# Task 2 Passed SKIP at PC 84
# Task 2 Passed SKIP at PC 104
# Task 2 Passed LOOP at PC 116 !
# Task 2 Passed LOOP at PC 116 !
# Task 2 Passed LOOP at PC 116 !
# Task 2 Passed LOOP at PC 116 !
# Task 2 Passed LOOP at PC 116 !
# ** Note: $stop    : ./testbench.sv(66)
#    Time: 5200 fs  Iteration: 1  Instance: /testbench
# Break in Module testbench at ./testbench.sv line 66

VSIM(paused)>
```

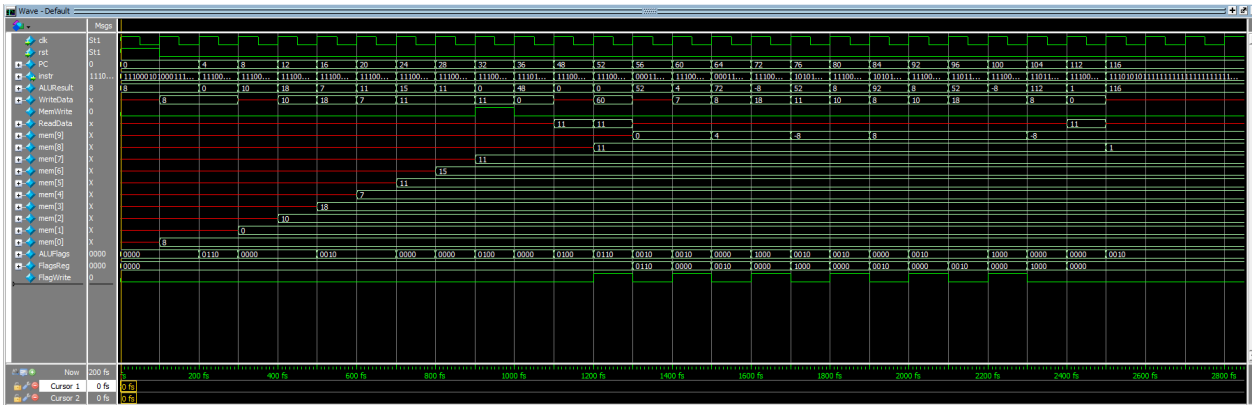


Figure_8: ModelSim arm.sv Task 2 Testbench Results

We can see from the figure above that the waveform and terminal output the expected cases, and skips occur where we expect them to. Additionally, the Controls are in line with our expectations for the instructions

Register Testbench

To ensure full coverage of the register's capability's, the following 3 scenarios were simulated:

1. Write data is written into the register file the clock cycle **after** wr_en is asserted.
2. Read data is updated to the register data at an address the **same** cycle the address was provided. Do this for both read addresses and data outputs.
3. Read data is updated to write data at an address the cycle **after** the address was provided if the write address is the same and wr_en was asserted. Do this for both read addresses and data outputs.

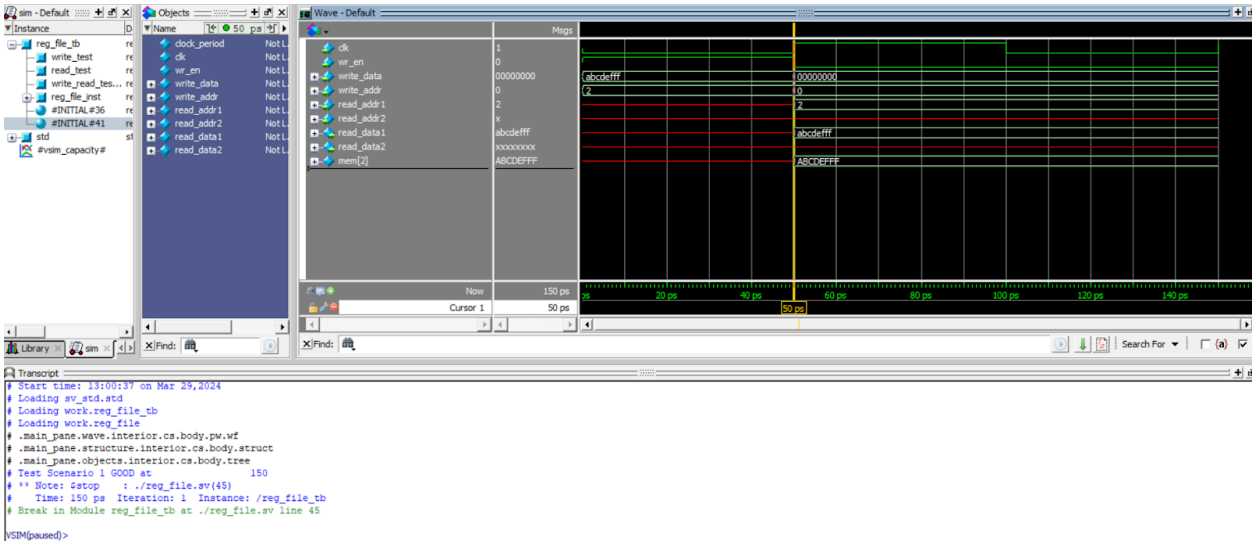


Figure_9: ModelSim reg_file Testbench (Scenario 1)

The Figure above shows the following scenario given in spec: Write data is written into the register file the clock cycle after wr_en is asserted. 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.

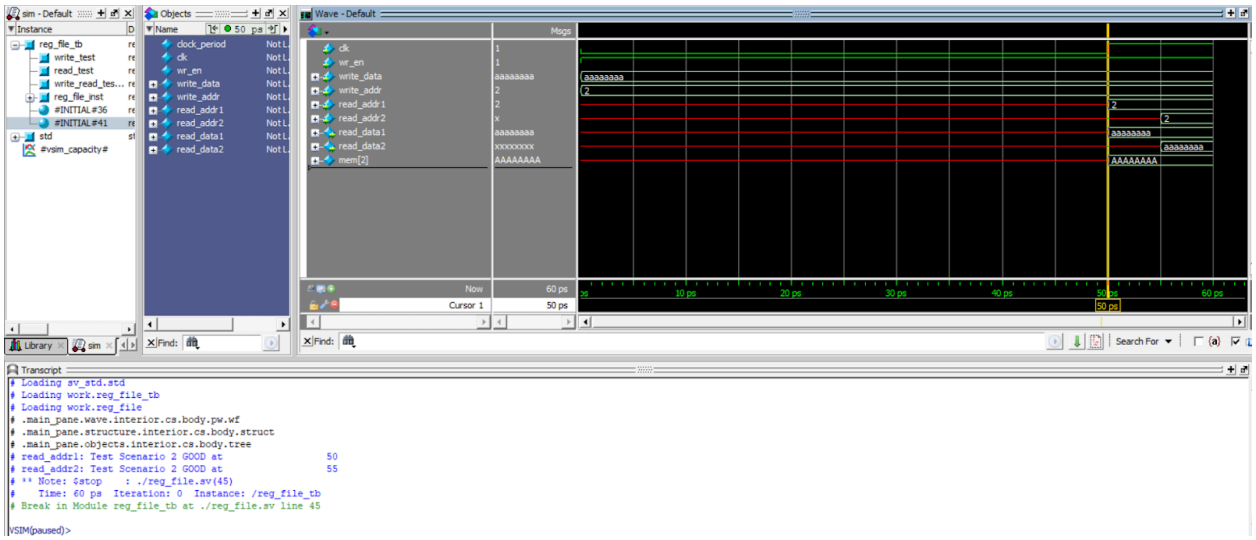


Figure_10: ModelSim reg_file Testbench (Scenario 2)

The Figure above shows the following scenario given in spec: Read data is updated the same cycle the address was provided. 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.

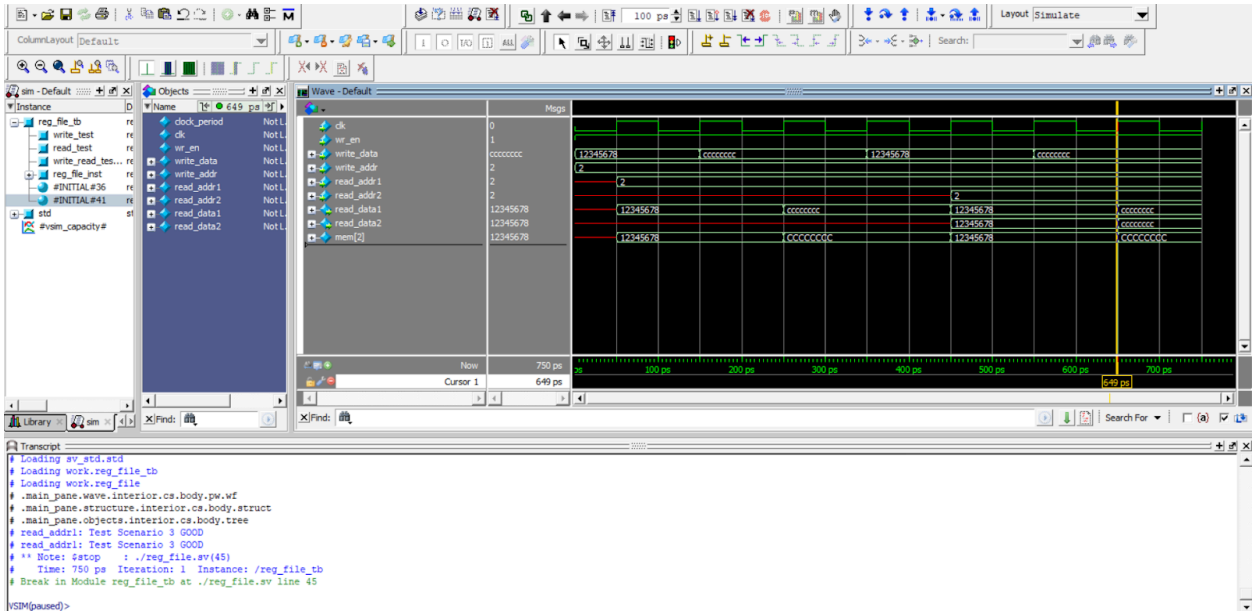
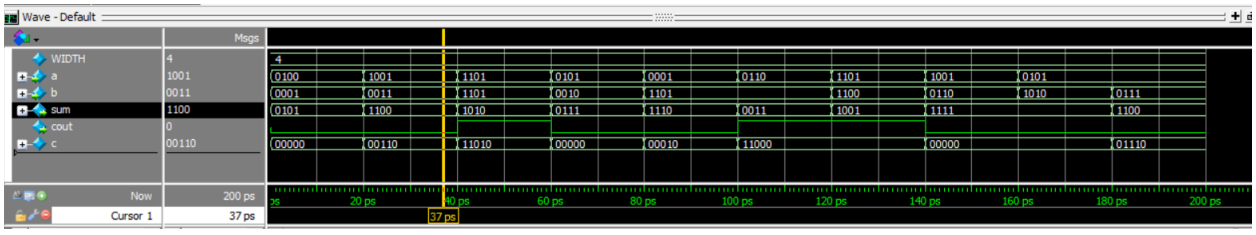


Figure_11: ModelSim reg_file Testbench (Scenario 3)

The Figure above shows the following scenario given in spec: “Read data is updated to write data at an address the cycle after the address was provided if the write address is the same and wr_en was asserted. Do this for both read addresses and data outputs"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.

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

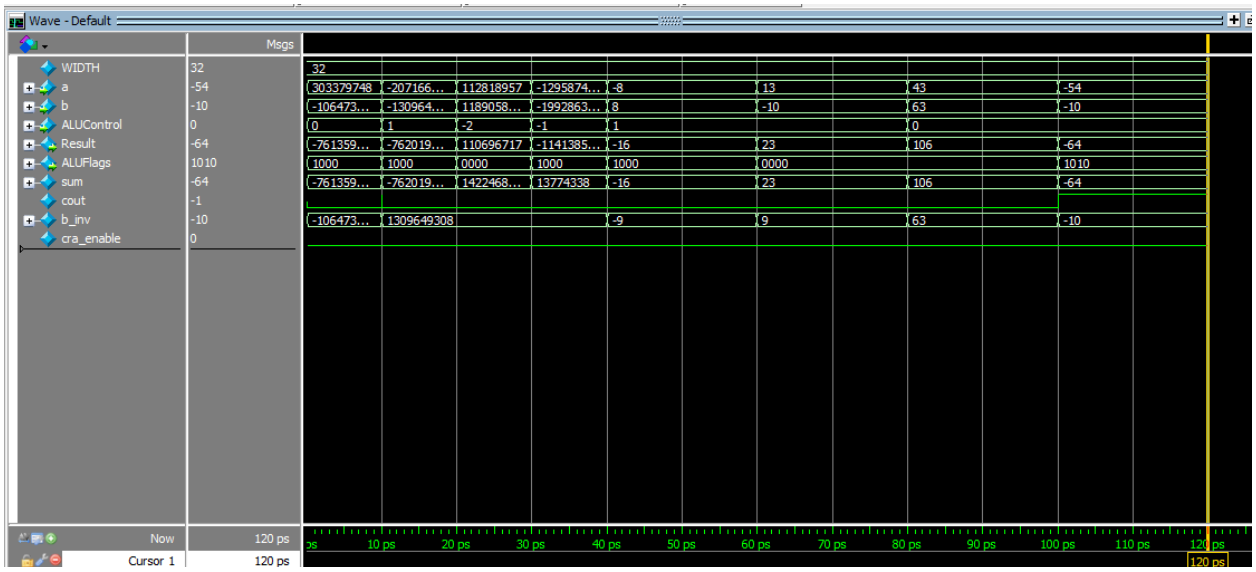


Figure_13: ModelSim ALU Testbench (Constrained Random Functional Test)

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.

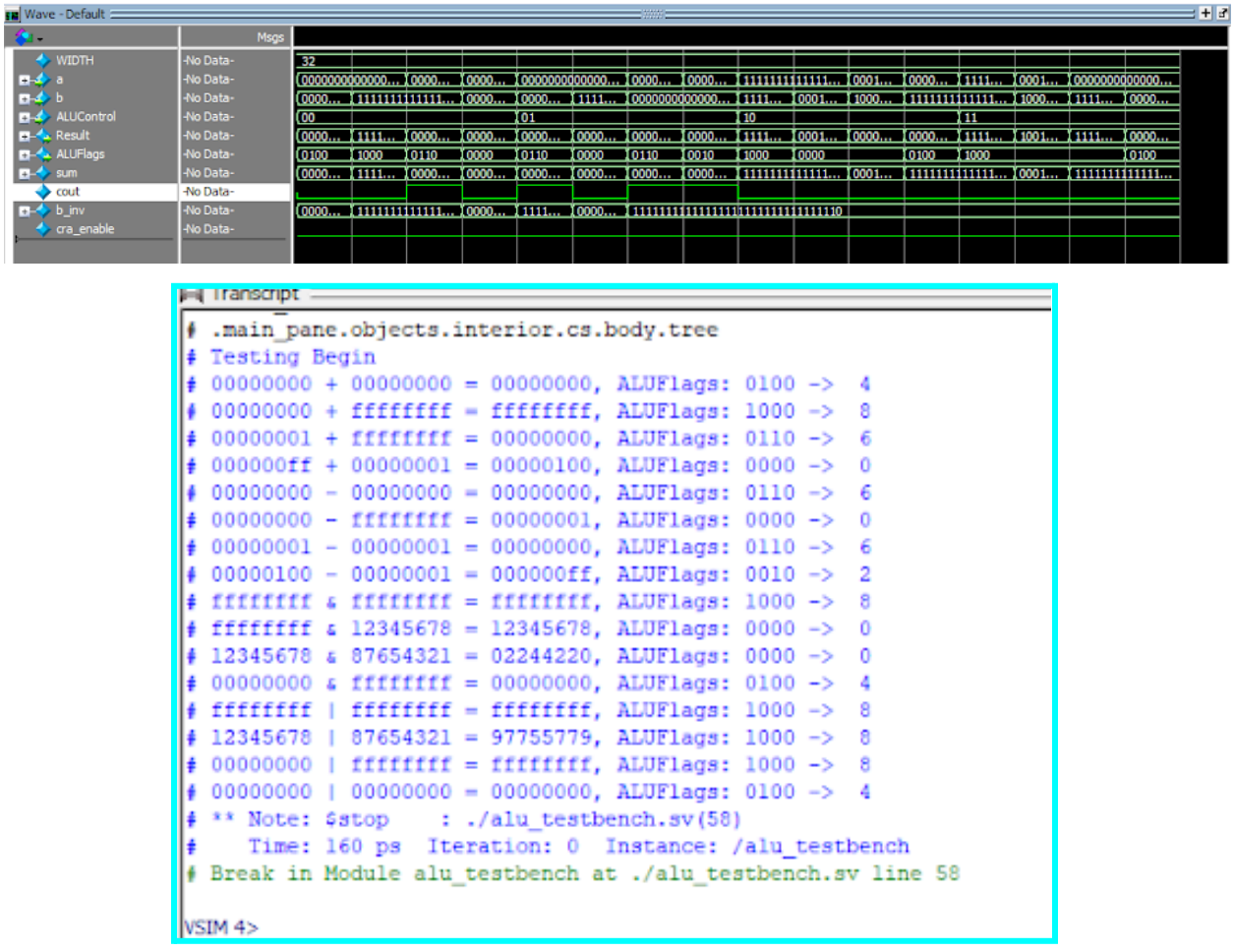


Figure_14: ModelSim ALU Testbench (Test Vector File)

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

```
1   /*
2   Justin Sim and Mina Gao
3   4/18/2024
4   EE 469 Hussein
5   Lab 2
6   */
7   /* dmem is a more traditional, albeit very uninteresting, random access 64 word x 32 bit per word memory.
8   ** This module is also written in RTL, and likely strongly resembles your own register file except for a
9   ** few minor differences. The first is that there is only a single read port, compared to the register
10  ** file's two read ports. The other difference is that the dmem is also byte aligned, and therefore
11  ** discards the bottom two bits of the address when doing a read or write.
12  */
13
14  // clk - system clock, same as the processor
15  // wr_en - write enable, allows the wr_data to overwrite the 32 bit word stored in memory[addr]
16  // addr - the location to which you intend to read or write from
17  // wr_data - the 32 bit data word which you intend to write into memory
18  // rd_data - the data currently stored at memory[addr]
19  module dmem (
20      input  logic        clk, wr_en,
21      input  logic [31:0] addr,
22      input  logic [31:0] wr_data,
23      output logic [31:0] rd_data
24  );
25
26      logic [31:0] memory [63:0];
27
28      // asyncrhnous read
29      assign rd_data = memory[addr[31:2]]; // word aligned, drop bottom 2 bits
30
31      // syncrhonous gated write
32      always_ff @(posedge clk) begin
33          if (wr_en) memory[addr[31:2]] <= wr_data; // word aligned, drop bottom 2 bits
34      end
35
36  endmodule : dmem
```

```
1   /*
2   Mina Gao and Justin Sim
3   4/18/2024
4   EE 469 Hussein
5   Lab 1 & 2
6
7      Full Adder: A digital circuit that performs addition of three bits - two inputs and a
    carry-in bit.
8      It produces a sum and a carry-out bit.
9      The sum is the XOR of the inputs and the carry-in.
10     The carry-out is generated when two or more of the inputs are high.
11
12  */
13  module fullAdder (
14      input logic a,
15      input logic b,
16      input logic cin,
17      output logic sum,
18      output logic cout
19  );
20      assign sum = a ^ b ^ cin;
21      assign cout = (a & b) | (cin & (a ^ b));
22  endmodule
23  //------------------------------------------------------------------------------
24  // Full Adder Testbench
25  // It tests each case of a 1 bit adder
26  // It provides a full functional coverage for the fullAdder
27
28  module fullAdder_tb();
29     logic a,b,cin,sum,cout;
30
31     fullAdder dut(.*);
32
33     integer i;
34     initial begin
35        for(i=0;i<2**3;i++) begin
36           {a,b,cin} = i; #10;
37        end
38     end
39  endmodule
40
```

```
1   /*
2   Justin Sim and Mina Gao
3   4/18/2024
4   EE 469 Hussein
5   Lab 2
6   */
7   /* imem is the read only, 64 word x 32 bit per word instruction memory for our processor.
8   ** Its module is written in RTL, and it strongly resembles a ROM (read only memory) or LUT
9   ** (look up table). This memory has no clock, and cannot be written to, but rather it
10  ** asynchronously reads out the word stored in its memory as soon as an address is given.
11  ** The address and memory are byte aligned, meaning that the bottom two bits are discarded
12  ** when looking for the word. One important line to note is the
13  **      Initial $readmemb("memfile.dat", memory);
14  ** which determines the contents of the memory when the system is initialized. You will
    alter
15  ** this line to use programs given to you as a part of this lab.
16  */
17
18  // addr - 32 bit address to determine the instruction to return. Note not all 32 bits are
    used since this
19  //        memory only has 64 words
20  // instr - 32 bit instruction to be sent to the processor
21  module imem(
22      input  logic [31:0] addr,
23      output logic [31:0] instr
24  );
25      logic [31:0] memory [63:0];
26
27      // modify the name and potentially directory prefix of the file within to load the
    correct program and preprocessing
28      initial $readmemb("memfile2.dat", memory);
29
30
31      assign instr = memory[addr[31:2]]; // word aligned, drops bottom 2 bits
32
33  endmodule : imem
```

```
// ADD R - 1110_000_0100_0_AAAA_DDDD_0000_0000_BBBB
// ADD I - 1110_001_0100_0_AAAA_DDDD_0000_IIII_IIII
// SUB R - 1110_000_0010_0_AAAA_DDDD_0000_0000_BBBB
// SUB I - 1110_001_0010_0_AAAA_DDDD_0000_IIII_IIII
// CMP R - 1110_000_0010_1_AAAA_DDDD_0000_0000_BBBB
// CMP I - 1110_001_0010_1_AAAA_DDDD_0000_IIII_IIII
// AND   - 1110_000_0000_0_AAAA_DDDD_0000_0000_BBBB
// ORR   - 1110_000_1100_0_AAAA_DDDD_0000_0000_BBBB
// LDR   - 1110_010_1100_1_AAAA_DDDD_IIII_IIII_IIII
// STR   - 1110_010_1100_0_AAAA_DDDD_IIII_IIII_IIII
//COND_1010_IIII_IIII_IIII_IIII_IIII_IIII

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


1110_001_0100_0_1111_0000_0000_0000_0000 // MAIN     ADD R0, R15, #0          0
1110_000_0010_0_0000_0001_0000_0000_0000 //        SUB R1, R0, R0           4
1110_001_0100_0_0001_0010_0000_0000_1010 //        ADD R2, R1, #10          8
1110_000_0100_0_0000_0011_0000_0000_0010 //        ADD R3, R0, R2           12
1110_001_0010_0_0010_0100_0000_0000_0011 //        SUB R4, R2, #3           16
1110_000_0010_0_0011_0101_0000_0000_0100 //        SUB R5, R3, R4           20
1110_000_1100_0_0100_0110_0000_0000_0101 //        ORR R6, R4, R5           24
1110_000_0000_0_0110_0111_0000_0000_0101 //        AND R7, R6, R5           28
1110_010_1100_0_0001_0111_0000_0000_0000 //        STR R7, [R1, #0]     32
1110_1010_0000_0000_0000_0000_0000_0001  //        B SKIP                   36
1110_010_1100_0_0001_0001_0000_0000_0000 //        STR R1, [R1, #0]         40
1110_1010_0000_0000_0000_0000_0000_0000  //        B LOOP                   44
1110_010_1100_1_0001_1000_0000_0000_0000 // SKIP     LDR R8, [R1, #0]         48
1110_001_0010_1_0110_1001_0000_0000_1111 // B_START     CMP R9, R6, #15           52
0001_1010_1111_1111_1111_1111_1111_1101  //             BNE B_START               56
1110_000_0010_1_0101_1001_0000_0000_0100 //             CMP R9, R5, R4            60
0001_1010_0000_0000_0000_0000_0000_0000  //             BNE BNE_TESTED            64
1110_1010_1111_1111_1111_1111_1111_1010  //             B B_START                 68
1110_000_0010_1_0010_1001_0000_0000_0011 // BNE_TESTED  CMP R9, R2, R3            72
1010_1010_1111_1111_1111_1111_1111_1000  //             BGE B_START               76
1110_000_0010_1_0011_1001_0000_0000_0010 //             CMP R9, R3, R2            80
1010_1010_0000_0000_0000_0000_0000_0000  //             BGE BGE_TESTED            84
1110_1010_1111_1111_1111_1111_1111_0101  //             B B_START                 88
1110_000_0010_1_0011_1001_0000_0000_0010 // BGE_TESTED  CMP R9, R3, R2            92
1101_1010_1111_1111_1111_1111_1111_0011  //             BLE B_START               96
1110_000_0010_1_0010_1001_0000_0000_0011 //             CMP R9, R2, R3            100
1101_1010_0000_0000_0000_0000_0000_0000  //             BLE BLE_TESTED            104
1110_1010_1111_1111_1111_1111_1111_0000  //             B B_START                 108
1110_001_0100_0_0001_1000_0000_0000_0001 // BLE_TESTED  ADD R8, R1, #1            112
1110_1010_1111_1111_1111_1111_1111_1110  // LOOP        B LOOP                    116
```

```
1   /*
2   Justin Sim and Mina Gao
3   4/18/2024
4   EE 469 Hussein
5   Lab 2
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] Instr;
21      logic [31:0] ReadData;
22      logic [31:0] WriteData;
23      logic [31:0] PC, ALUResult;
24      logic        MemWrite;
25
26      // our single cycle arm processor
27      arm processor (
28          .clk        (clk        ),
29          .rst        (rst        ),
30          .Instr      (Instr      ),
31          .ReadData   (ReadData   ),
32          .WriteData  (WriteData  ),
33          .PC         (PC         ),
34          .ALUResult  (ALUResult  ),
35          .MemWrite   (MemWrite   )
36      );
37
38      // instruction memory
39      // contained machine code instructions which instruct processor on which operations to
    make
40      // effectively a rom because our processor cannot write to it
41      imem imemory (
42          .addr   (PC     ),
43          .instr  (Instr  )
44      );
45
46      // data memory
47      // containes data accessible by the processor through ldr and str commands
48      dmem dmemory (
49          .clk     (clk       ),
50          .wr_en   (MemWrite  ),
51          .addr    (ALUResult ),
52          .wr_data (WriteData ),
53          .rd_data (ReadData  )
54      );;
55
56
57  endmodule : top
```

```
1   /*
2   Justin Sim and Mina Gao
3   4/18/2024
4   EE 469 Hussein
5   Lab 2
6   */
7   /* testbench is a simulation module which simply instantiates the processor system and runs
    50 cycles
8   ** of instructions before terminating. At termination, specific register file values are
    checked to
9   ** verify the processors' ability to execute the implemented instructions.
10  */
11  module testbench();
12
13      // system signals
14      logic clk, rst;
15
16      // generate clock with 100ps clk period
17      initial begin
18          clk = 1'1;
19          forever #50 clk = ~clk;
20      end
21
22      // processor instantion. Within is the processor as well as imem and dmem
23      top cpu (.clk(clk), .rst(rst));
24
25      initial begin
26          // start with a basic reset
27          rst = 1; @(posedge clk);
28          rst <= 0; @(posedge clk);
29
30          // repeat for 50 cycles. Not all 50 are necessary, however a loop at the end of the
    program will keep anything weird from happening
31          repeat(50) @(posedge clk);
32
33          // basic checking to ensure the right final answer is achieved. These DO NOT prove
    your system works. A more careful look at your
34          // simulation and code will be made.
35
36          // task 1:
37  //        assert(cpu.processor.u_reg_file.mem[8] == 32'hb) $display("Task 1 Passed R8");
38  //          else                                           $display("Task 1 Failed R8");
39  //      assert(cpu.processor.u_reg_file.mem[7] == 32'hb) $display("Task 1 Passed R7");
40  //          else                                           $display("Task 1 Failed R7");
41  //      assert(cpu.processor.u_reg_file.mem[6] == 32'hf) $display("Task 1 Passed R6");
42  //          else                                           $display("Task 1 Failed R6");
43  //      assert(cpu.processor.u_reg_file.mem[5] == 32'hb) $display("Task 1 Passed R5");
44  //          else                                           $display("Task 1 Failed R5");
45  //      assert(cpu.processor.u_reg_file.mem[4] == 32'h7) $display("Task 1 Passed R4");
46  //          else                                           $display("Task 1 Failed R4");
47  //      assert(cpu.processor.u_reg_file.mem[3] == 32'h12) $display("Task 1 Passed R3");
48  //          else                                           $display("Task 1 Failed R3");
49  //      assert(cpu.processor.u_reg_file.mem[2] == 32'ha) $display("Task 1 Passed R2");
50  //          else                                           $display("Task 1 Failed R2");
51  //      assert(cpu.processor.u_reg_file.mem[1] == 32'h0) $display("Task 1 Passed R1");
52  //          else                                           $display("Task 1 Failed R1");
53  //      assert(cpu.processor.u_reg_file.mem[0] == 32'h8) $display("Task 1 Passed R0");
54  //          else                                           $display("Task 1 Failed R0");
55
56          // task 2:
57          assert(cpu.processor.u_reg_file.mem[8] == 32'd1)  $display("Task 2 Passed");
58          else                                                $display("Task 2 Failed");
59          // verify skips
60          assert(cpu.processor.PC == 32'd40 || cpu.processor.PC == 32'd40) $display("Task 2
    Failed SKIP at PC 36");
61          else                                                             $display("Task 2
    Passed SKIP at PC 36");
62          assert(cpu.processor.PC == 32'd68)                               $display("Task 2
    Failed SKIP at PC 64");
63          else                                                             $display("Task 2
    Passed SKIP at PC 64");
64          assert(cpu.processor.PC == 32'd88)                               $display("Task 2
    Failed SKIP at PC 84");
```

```
65          else                                                    $display("Task 2
    Passed SKIP at PC 84');
66          assert(cpu.processor.PC == 32'd108)                     $display("Task 2
    Failed SKIP at PC 104');
67          else                                                    $display("Task 2
    Passed SKIP at PC 104');
68
69          // verify end loops
70          repeat(5) assert(cpu.processor.PC == 32'd116)           $display("Task 2
    Passed LOOP at PC 116 !');
71          else                                                    $display("Task 2
    Failed LOOP at PC 116');
72          $stop;
73      end
74
75
76  endmodule
```

```
1   /*
2   Mina Gao and Justin Sim
3   4/18/2024
4   EE 469 Hussein
5   Lab 1 & 2
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 addresse
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   Mina Gao and Justin Sim
3   4/18/2024
4   EE 469 Hussein
5   Lab 1 & 2
6
7      Carry Ripple Adder Module:
8      This module implements a carry ripple adder, supporting binary addition operations.
9      It consists of multiple full adders connected in a ripple-carry configuration.
10     The module accepts two input vectors for addition/subtraction and produces a sum vector.
11     Additionally, it generates a carry-out signal to indicate overflow during addition.
12     The carry ripple adder module accommodates addition and subtraction of binary numbers
13
14  */
15  module CarryRippleAdder #(
16     parameter WIDTH = 32
17     )(
18      input logic   [WIDTH-1:0] a,b,
19      input logic   cin,
20      output logic  [WIDTH-1:0] sum,
21      output logic  cout
22  );
23     // hold value of cin for each iteration form 0 to WIDTH
24     logic [WIDTH:0] c;
25     assign c[0] = cin;
26
27     // iterate fullAdder through each bit of a and b
28     generate
29        genvar i;
30              for (i=0; i < WIDTH; i++) begin
31                 fullAdder fa (.a(a[i]),
32                               .b(b[i]),
33                               .cin(c[i]),
34                               .sum(sum[i]),
35                               .cout(c[i+1])
36                               );
37              end
38        endgenerate
39
40     assign cout = c[WIDTH];
41
42  endmodule : CarryRippleAdder
43  //------------------------------------------------------------------------------
44  // CarryRippleAdder Testbench
45  // It has 3 different tasks to verify functionality of CarryRippleAdder
46  // It includes a constrained Random Verification task called rand_vect_10
47  // It has an adjustable WIDTH to verify successful implementation of varying
48  // input sizes
49
50  module CarryRippleAdder_tb ();
51  parameter WIDTH=4;
52     //---------------------------------
53     // Define IN OUT for FUT
54     //---------------------------------
55     logic [WIDTH-1:0] a,b,sum;
56     logic cin, cout;
57
58     //---------------------------------
59     // Instantiate FullAdder
60     //---------------------------------
61     CarryRippleAdder #(WIDTH) dut(.*);
62
63
64     //---------------------------------
65     // Run Testing Procedures
66     //---------------------------------
67     initial begin
68        rand_vect_10; //SUCCESS
69        $stop;
70     end
71
72
73     //------------------------------------------------------------------------
```

```
74      // Define Tasks for testing Functionality
75      //---------------------------------------------------------------------
76      task test; begin
77         a = 4'd5;
78         b = 4'd5;
79         cin = 0;
80         #10;
81         if (sum == a + b)
82               $display("SUCCESSFUL ADDITION \n a: %b,b: %b,sum: %b,cout: %b", a,b,sum,cout);
83            else
84               $fatal("ERROR in ADD\n a: %d, b: %d, sum: %d,cout: %b", a,b,sum,cout);
85            #10;
86      end endtask
87
88      // test_32: Test all cases
89      task test_all; begin
90         integer i;
91         // Test Addition
92         for(i=0;i<2**2*WIDTH;i++) begin
93            cin = 0;
94            {a,b} = i; #10;
95
96            if (sum == a + b)
97               $display("SUCCESSFUL ADDITION \n a: %d,b: %d,sum: %d,cout: %b", a,b,sum,cout);
98            else
99               $fatal("ERROR in ADD\n a: %d, b: %d, sum: %d,cout: %b", a,b,sum,cout);
100           #10;
101        end
102     end endtask
103
104     // rand_vect_10: create 10 random test vectors
105     task rand_vect_10; begin
106        // Test Addition
107        repeat (10) begin
108           a = $random;
109           b = $random;
110           cin = 0;
111           #10;
112           if (sum == a + b)
113              $display("SUCCESSFUL ADDITION \n a: %d,b: %d,sum: %d,cout: %b", a,b,sum,cout);
114           else
115              $fatal("ERROR in ADD\n a: %d, b: %d, sum: %d,cout: %b", a,b,sum,cout);
116           #10;
117        end
118     end endtask
119
120  endmodule : CarryRippleAdder_tb
121
```

```
1   /*
2   Justin Sim and Mina Gao
3   4/18/2024
4   EE 469 Hussein
5   Lab 2
6
7   arm is the spotlight of the show and contains the bulk of the datapath and control logic.
8   This module is split into two parts, the datapath and control.
9   */
10
11  // clk - system clock
12  // rst - system reset
13  // Instr - incoming 32 bit instruction from imem, contains opcode, condition, addresses and
    or immediates
14  // ReadData - data read out of the dmem
15  // WriteData - data to be written to the dmem
16  // MemWrite - write enable to allowed WriteData to overwrite an existing dmem word
17  // PC - the current program count value, goes to imem to fetch instruciton
18  // ALUResult - result of the ALU operation, sent as address to the dmem
19
20  module arm (
21      input  logic        clk, rst,
22      input  logic [31:0] Instr,
23      input  logic [31:0] ReadData,
24      output logic [31:0] WriteData,
25      output logic [31:0] PC, ALUResult,
26      output logic        MemWrite
27  );
28
29      // datapath buses and signals
30      logic [31:0] PCPrime, PCPlus4, PCPlus8; // pc signals
31      logic [ 3:0] RA1, RA2;                  // regfile input addresses
32      logic [31:0] RD1, RD2;                  // raw regfile outputs
33      logic [ 3:0] ALUFlags;                  // alu combinational flag outputs
34      logic [31:0] ExtImm, SrcA, SrcB;        // immediate and alu inputs
35      logic [31:0] Result;                    // computed or fetched value to be written into
    regfile or pc
36      logic [ 3:0] FlagsReg;                  // register to store the flags from the most
    recent CMP command
37
38      // control signals
39      // adding FlagWrite to control if loading the FlagsReg, control signal for the decoder
40      logic PCSrc, MemtoReg, ALUSrc, RegWrite, FlagWrite;
41      logic [1:0] RegSrc, ImmSrc, ALUControl;
42
43
44      /* 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
45      ** 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
46      ** ports of the register file and alu take some time to study and understand the logic
    and flow of the datapath.
47      */
48      //-------------------------------------------------------------------------------
49      //                                      DATAPATH
50      //-------------------------------------------------------------------------------
51
52
53      assign PCPrime = PCSrc ? Result : PCPlus4;  // mux, use either default or newly
    computed value
54      assign PCPlus4 = PC + 'd4;                  // default value to access next instruction
55      assign PCPlus8 = PCPlus4 + 'd4;             // value read when reading from reg[15]
56
57      // update the PC, at rst initialize to 0
58      always_ff @(posedge clk) begin
59          if (rst) PC <= '0;
60          else     PC <= PCPrime;
61      end
62
63      // determine the register addresses based on control signals
64      // RegSrc[0] is set if doing a branch instruction
65      // RefSrc[1] is set when doing memory instructions
66      assign RA1 = RegSrc[0] ? 4'd15        : Instr[19:16];
```

```
67        assign RA2 = RegSrc[1] ? Instr[15:12] : Instr[ 3: 0];
68
69        // -------------------------------------------------------
70        // Instantiate register
71        // Inputs: clk, RegWrite, Result, Instr[15:12], RA1, RA2
72        // Outputs: RD1, RD2
73        // -------------------------------------------------------
74        reg_file u_reg_file (
75            .clk       (clk),
76            .wr_en     (RegWrite),
77            .write_data(Result),
78            .write_addr(Instr[15:12]),
79            .read_addr1(RA1),
80            .read_addr2(RA2),
81            .read_data1(RD1),
82            .read_data2(RD2)
83        );
84
85        // two muxes, put together into an always_comb for clarity
86        // determines which set of instruction bits are used for the immediate
87        always_comb begin
88            if      (ImmSrc == 'b00) ExtImm = {{24{Instr[7]}},Instr[7:0]};          // 8 bit
    immediate - reg operations
89            else if (ImmSrc == 'b01) ExtImm = {20'b0, Instr[11:0]};                 // 12 bit
    immediate - mem operations
90            else                     ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00}; // 24 bit
    immediate - branch operation
91        end
92
93        // WriteData and SrcA are direct outputs of the register file, wheras SrcB is chosen
    between reg file output and the immediate
94        assign WriteData = (RA2 == 'd15) ? PCPlus8 : RD2;           // substitute the 15th
    regfile register for PC
95        assign SrcA      = (RA1 == 'd15) ? PCPlus8 : RD1;           // substitute the 15th
    regfile register for PC
96        assign SrcB      = ALUSrc        ? ExtImm  : WriteData;     // determine alu operand to
    be either from reg file or from immediate
97
98        // ---------------------------------------
99        // Instantiate ALU
100       // Inputs: a, b, ALUControl
101       // Outputs: Result, ALUFlags
102       // ---------------------------------------
103       alu u_alu (
104           .a          (SrcA),
105           .b          (SrcB),
106           .ALUControl (ALUControl),
107           .Result     (ALUResult),
108           .ALUFlags   (ALUFlags)
109       );
110
111
112       // update the FlagsReg if FlagWrite is enable
113       always_ff @(posedge clk) begin
114           if (FlagWrite)  FlagsReg <= ALUFlags;
115           else      FlagsReg <= '0;
116       end
117
118       // determine the result to run back to PC or the register file based on whether we used
    a memory instruction
119       assign Result = MemtoReg ? ReadData : ALUResult;    // determine whether final
    writeback result is from dmemory or alu
120
121
122       /* The control conists of a large decoder, which evaluates the top bits of the
    instruction and produces the control bits
123       ** which become the select bits and write enables of the system. The write enables
    (RegWrite, MemWrite and PCSrc) are
124       ** especially important because they are representative of your processors current
    state.
125       */
126       //-------------------------------------------------------------------------------
127       //                                      CONTROL
```

```
128       //------------------------------------------------------------------------------
129
130       always_comb begin
131      // check the instruction condition (Instr[31:28]) is valid before deciding to execute
     the branch
132      casez (Instr[31:20])
133
134              // ADD (Imm or Reg)
135         // note that we use wildcard "?" in bit 25. That bit decides whether we use
     immediate or reg, but regardless we add
136         12'b111000?01000 : begin
137         PCSrc    = 0;
138                  MemtoReg = 0;
139                  MemWrite = 0;
140                  ALUSrc   = Instr[25]; // may use immediate
141                  RegWrite = 1;
142                  RegSrc   = 'b00;
143                  ImmSrc   = 'b00;
144                  ALUControl = 'b00;
145              FlagWrite = 0;
146               end
147
148              // SUB (Imm or Reg)
149          12'b111000?00100 : begin
150         PCSrc    = 0;
151                  MemtoReg = 0;
152                  MemWrite = 0;
153                  ALUSrc   = Instr[25]; // may use immediate
154                  RegWrite = 1;
155                  RegSrc   = 'b00;
156                  ImmSrc   = 'b00;
157                  ALUControl = 'b01;
158              FlagWrite = 0;
159               end
160
161              // AND
162          12'b111000000000 : begin
163         PCSrc    = 0;
164                  MemtoReg = 0;
165                  MemWrite = 0;
166                  ALUSrc   = 0;
167                  RegWrite = 1;
168                  RegSrc   = 'b00;
169                  ImmSrc   = 'b00;    // doesn't matter
170                  ALUControl = 'b10;
171         FlagWrite = 0;
172          end
173
174              // ORR
175          12'b111000011000 : begin
176         PCSrc    = 0;
177                  MemtoReg = 0;
178                  MemWrite = 0;
179                  ALUSrc   = 0;
180                  RegWrite = 1;
181                  RegSrc   = 'b00;
182                  ImmSrc   = 'b00;    // doesn't matter
183                  ALUControl = 'b11;
184         FlagWrite = 0;
185          end
186
187              // LDR
188          12'b111001011001 : begin
189         PCSrc    = 0;
190                  MemtoReg = 1;
191                  MemWrite = 0;
192                  ALUSrc   = 1;
193                  RegWrite = 1;
194                  RegSrc   = 'b10;    // msb doesn't matter
195                  ImmSrc   = 'b01;
196                  ALUControl = 'b00;  // do an add
197         FlagWrite = 0;
198          end
```

```
199
200             // STR
201          12'b111001011000 : begin
202         PCSrc    = 0;
203                    MemtoReg = 0; // doesn't matter
204                    MemWrite = 1;
205                    ALUSrc   = 1;
206                    RegWrite = 0;
207                    RegSrc   = 'b10;    // msb doesn't matter
208                    ImmSrc   = 'b01;
209                    ALUControl = 'b00;  // do an add
210         FlagWrite = 0;
211          end
212
213             // B
214          12'b11101010???? : begin
215         PCSrc    = 1;
216                    MemtoReg = 0;
217                    MemWrite = 0;
218                    ALUSrc   = 1;
219                    RegWrite = 0;
220                    RegSrc   = 'b01;
221                    ImmSrc   = 'b10;
222                    ALUControl = 'b00;  // do an add
223         FlagWrite = 0;
224               end
225
226          // BEQ
227          12'b00001010???? : begin
228         PCSrc    = FlagsReg[2];
229                    MemtoReg = 0;
230                    MemWrite = 0;
231                    ALUSrc   = 1;
232                    RegWrite = 0;
233                    RegSrc   = 'b01;
234                    ImmSrc   = 'b10;
235                    ALUControl = 'b00;  // do an add
236         FlagWrite = 0;
237               end
238
239          // BNE
240          12'b00011010???? : begin
241         PCSrc    = ~FlagsReg[2];
242                    MemtoReg = 0;
243                    MemWrite = 0;
244                    ALUSrc   = 1;
245                    RegWrite = 0;
246                    RegSrc   = 'b01;
247                    ImmSrc   = 'b10;
248                    ALUControl = 'b00;  // do an add
249         FlagWrite = 0;
250               end
251
252          // BGE
253          12'b10101010???? : begin
254         PCSrc    = ~(FlagsReg[3]^FlagsReg[0]);
255                    MemtoReg = 0;
256                    MemWrite = 0;
257                    ALUSrc   = 1;
258                    RegWrite = 0;
259                    RegSrc   = 'b01;
260                    ImmSrc   = 'b10;
261                    ALUControl = 'b00;  // do an add
262         FlagWrite = 0;
263               end
264
265          // BGT
266          12'b11001010???? : begin
267         PCSrc    = ~FlagsReg[2] & ~(FlagsReg[3]^FlagsReg[0]);
268                    MemtoReg = 0;
269                    MemWrite = 0;
270                    ALUSrc   = 1;
271                    RegWrite = 0;
```

```
272                     RegSrc   = 'b01;
273                     ImmSrc   = 'b10;
274                     ALUControl = 'b00;  // do an add
275             FlagWrite = 0;
276                   end
277
278                   // BLE
279              12'b11011010???? : begin
280             PCSrc    = FlagsReg[2] | (FlagsReg[3]^FlagsReg[0]);
281                     MemtoReg = 0;
282                     MemWrite = 0;
283                     ALUSrc   = 1;
284                     RegWrite = 0;
285                     RegSrc   = 'b01;
286                     ImmSrc   = 'b10;
287                     ALUControl = 'b00;  // do an add
288             FlagWrite = 0;
289                   end
290
291              // BLT
292              12'b10111010???? : begin
293             PCSrc    = FlagsReg[3]^FlagsReg[0];
294                     MemtoReg = 0;
295                     MemWrite = 0;
296                     ALUSrc   = 1;
297                     RegWrite = 0;
298                     RegSrc   = 'b01;
299                     ImmSrc   = 'b10;
300                     ALUControl = 'b00;  // do an add
301             FlagWrite = 0;
302                   end
303
304              // CMP
305              12'b111000?00101 : begin
306             PCSrc    = 0;
307                     MemtoReg = 0;
308                     MemWrite = 0;
309                     ALUSrc   = Instr[25]; // may use immediate
310                     RegWrite = 1;
311                     RegSrc   = 'b00;
312                     ImmSrc   = 'b00;
313                     ALUControl = 'b01;
314             FlagWrite = 1;
315                   end
316
317              default: begin
318             PCSrc    = 0;
319                     MemtoReg = 0; // doesn't matter
320                     MemWrite = 0;
321                     ALUSrc   = 0;
322                     RegWrite = 0;
323                     RegSrc   = 'b00;
324                     ImmSrc   = 'b00;
325                     ALUControl = 'b00;  // do an add
326             FlagWrite = 0;
327              end
328               endcase
329           end
330
331
332   endmodule : arm
```

```
1   /*
2   Mina Gao and Justin Sim
3   4/18/2024
4   EE 469 Hussein
5   Lab 1 & 2
6
7      ALU (Arithmetic Logic Unit): A digital circuit that performs arithmetic and logical
    operations.
8      It supports addition, subtraction, bitwise AND, and bitwise OR operations.
9       For addition, it takes two input operands and produces a sum.
10      For subtraction, it takes two input operands and produces a difference.
11      For bitwise AND, it takes two input operands and produces a bitwise AND result.
12      For bitwise OR, it takes two input operands and produces a bitwise OR result.
13      The ALU operation is determined by a control signal.
14
15  */
16  module alu#(
17     parameter WIDTH = 32
18     )(
19     input logic [WIDTH-1:0] a,b,
20     input logic [1:0] ALUControl,
21     output logic [WIDTH-1:0] Result,
22     output logic [3:0] ALUFlags
23  );
24
25     //---------------------------------------
26     // Define Intermediate logic
27     //---------------------------------------
28     logic [WIDTH-1:0] sum;
29     logic cout;
30     logic [WIDTH-1:0] b_inv;
31     bit cra_enable;   // enables carryrippleadder
32
33     //-----------------------------------------
34     // Execute operations based on CTRL signal
35     //-----------------------------------------
36     always_comb begin
37        case(ALUControl[1])
38           0: begin //Addition or Subtraction
39           // Selective Inverter :
40           //    'flips' b if ALUControl == 2'b01
41              b_inv = b ^ {WIDTH{ALUControl[0]}};
42              Result = sum;
43  //          $display("Adding or Subtracting");
44           end
45
46           1: begin // Bitwise AND
47              if (ALUControl[0] == 0) begin
48                 Result = a & b;
49              end else begin // Bitwise OR
50                 Result = a | b;
51              end
52
53              cout = 0;
54           end
55        endcase
56     end
57     //-----------------------------------------
58     // Instantiate CarryRippleAdder if Selected
59     //-----------------------------------------
60        CarryRippleAdder #(WIDTH) cra(.a(a),
61                                      .b(b_inv),
62                                      .sum(sum),
63                                      .cin(ALUControl[0]),
64                                      .cout(cout));
65
66     //---------------------------------------
67     // Assignments for ALUFlags
68     //---------------------------------------
69     assign ALUFlags[0] =(( (ALUControl == 2'b00) && ((a>0 && b>0 && sum<0) || (a<0 && b<0 &&
    sum>0)) )
70                         || ( (ALUControl == 2'b01) && ((a<0 && b>0 && sum>0) || (a>0 && b<0 &&
    sum<0)) ))
```

```
71                          ? 1 : 0;
72      assign ALUFlags[1] = (cout == 1'b1) ? 1: 0;
73      assign ALUFlags[2] = (Result == 0) ? 1: 0;
74      assign ALUFlags[3] = Result[WIDTH-1] ? 1: 0;
75
76  endmodule : alu
77  //------------------------------------------------------------------------------------------
78  // ALU Testbench
79  // It does 1 random test for each ALU function
80  // It provides a full functional coverage for the ALU Testbench
81  // It provides verification for 2 Overflow cases
82
83  module alu_tb;
84  parameter WIDTH = 32;
85      // Inputs
86      logic [WIDTH-1:0] a;
87      logic [WIDTH-1:0] b;
88      logic [1:0] ALUControl;
89
90      // Outputs
91      logic [WIDTH-1:0] Result;
92      logic [3:0] ALUFlags;
93
94      // Instantiate the ALU module
95      alu #(WIDTH) dut(.*);
96
97      // Test cases
98      initial begin
99          // Test Addition at
100         a = $random;
101         b = $random;
102         ALUControl = 2'b00;
103         #10;
104         if (Result == a + b)
105             $display("SUCCESSFUL ADDITION a: %d, b: %d,result: %d at: %t", a,b,Result, $time
    );
106          else
107             $fatal("ERROR in ADD\n a: %d, b: %d,result: %d", a,b,Result);
108
109         // Test Subtraction
110         a = $random;
111         b = $random;
112         ALUControl = 2'b01;
113         #10;
114         if (Result == a - b)
115             $display("SUCCESSFUL SUBTRACTION a: %d, b: %d,result: %d at: %t", a,b,Result,
    $time);
116          else
117             $fatal("ERROR in SUSBTRACT\n a: %d, b: %d,result: %d", a,b,Result);
118
119         // Test OR
120         a = $random;
121         b = $random;
122         ALUControl = 2'b10;
123         #10;
124         if (Result == (a & b))
125             $display("SUCCESSFUL AND a: %d, b: %d,result: %d at: %t", a,b,Result, $time);
126          else
127             $fatal("ERROR in AND a: %b, b: %b,result: %b", a,b,Result);
128
129         // Test OR
130         a = $random;
131         b = $random;
132         ALUControl = 2'b11;
133         #10;
134         if (Result == (a | b))
135             $display("SUCCESSFUL OR a: %d b: %d,result: %d at: %t", a,b,Result, $time);
136          else
137             $fatal("ERROR in OR a: %b, b: %b,result: %b", a,b,Result);
138
139         // Test Subtraction OVerflow
140          ALUControl = 2'b01;
141 //       constraint a_c {a<0;};
```

```
142   //        constraint b_c {b>0;};
143   //        a.randomize() with {a_c;};
144   //        b.randomize() with {b_c;};
145             a = -8;
146             b = 8;
147             #10;
148           if ((Result > 0) && (ALUControl[0] != 1)) begin
149               $fatal("FAIL TO SEE OVERFLOW: %d-%d!=%d", a,b,Result);
150             end
151           #10;
152   //        constraint a_c {a>0;};
153   //        constraint b_c {b<0;};
154             a = $urandom_range(1, 100);
155             b = $urandom_range(-100, -1);
156             #10;
157           if ((Result < 0) && (ALUControl[0] != 1)) begin
158               $fatal("FAIL TO SEE OVERFLOW: %d-%d!=%d", a,b,Result);
159             end
160           #10;
161           $display("SUB OVERFLOW DETECTION SUCCESSFUL at: %t", $time);
162
163
164             // Test Addition OVerflow
165             ALUControl = 2'b00;
166
167             a = $urandom_range(1, 100);;
168             b = $urandom_range(1, 100);
169             #10;
170           if ((Result < 0) && (ALUControl[0] != 1)) begin
171               $fatal("FAIL TO SEE OVERFLOW: %d+%d!=%d", a,b,Result);
172             end
173           #10;
174
175             a = $urandom_range(-100, -1);
176             b = $urandom_range(-100, -1);
177             #10;
178           if ((Result[WIDTH-1] == 0) && (ALUControl[0] != 1)) begin
179               $fatal("FAIL TO SEE OVERFLOW at %t",$time);
180             end
181           #10;
182           $display("ADD OVERFLOW DETECTION SUCCESSFUL at: %t", $time);
183           $stop;
184
185       end // initial begin
186
187
188
189   endmodule : alu_tb
190
```