

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.

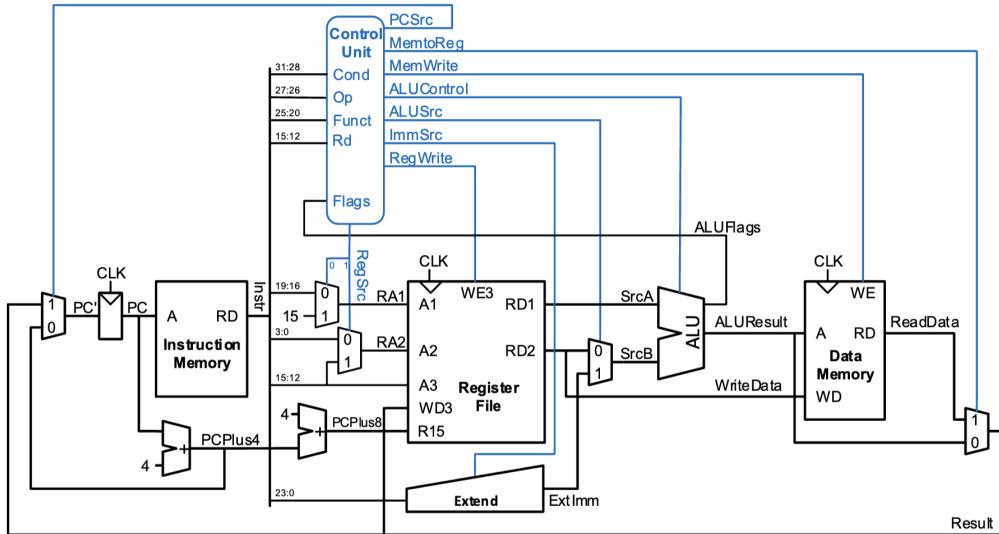
```
// Instantiate ALU
// inputs: a, b, ALUControl
// outputs: Result, ALUFlags
//-----
alu u_alu (
    .a          (SrcA),
    .b          (SrcB),
    .ALUControl (ALUControl),
    .Result     (ALUResult),
    .ALUFlags   (ALUFlags)
);
```

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).

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	$\bar{N} \oplus V$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\bar{Z}(N \oplus V)$
1101	LE	Signed less than or equal	Z OR (N $\oplus V$)
1110	AL (or none)	Always / unconditional	ignored

Figure_6: Condition Mnemonics table

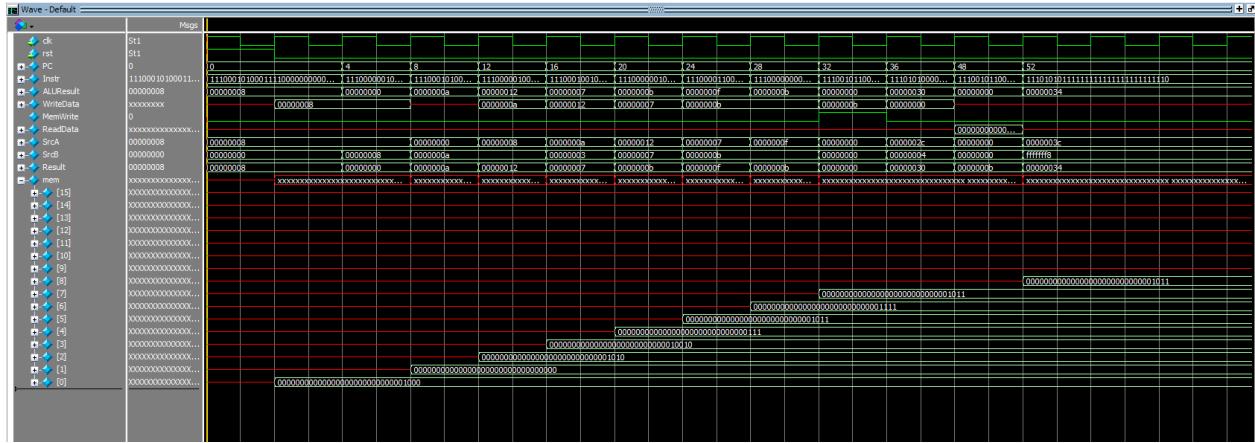
Results

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 $display("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

```

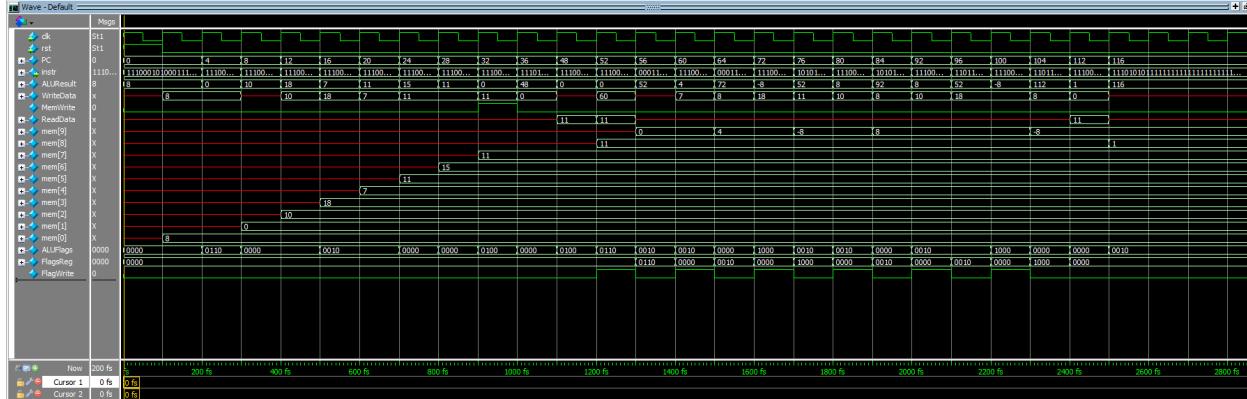
Transcript

```

# Warning: (vsim 666) ./arm.sv(66) @ 5200 fs Iteration: 1 Instance: /testbench/cpu/processor/u_alu File: ./alu.sv
# 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)>

```



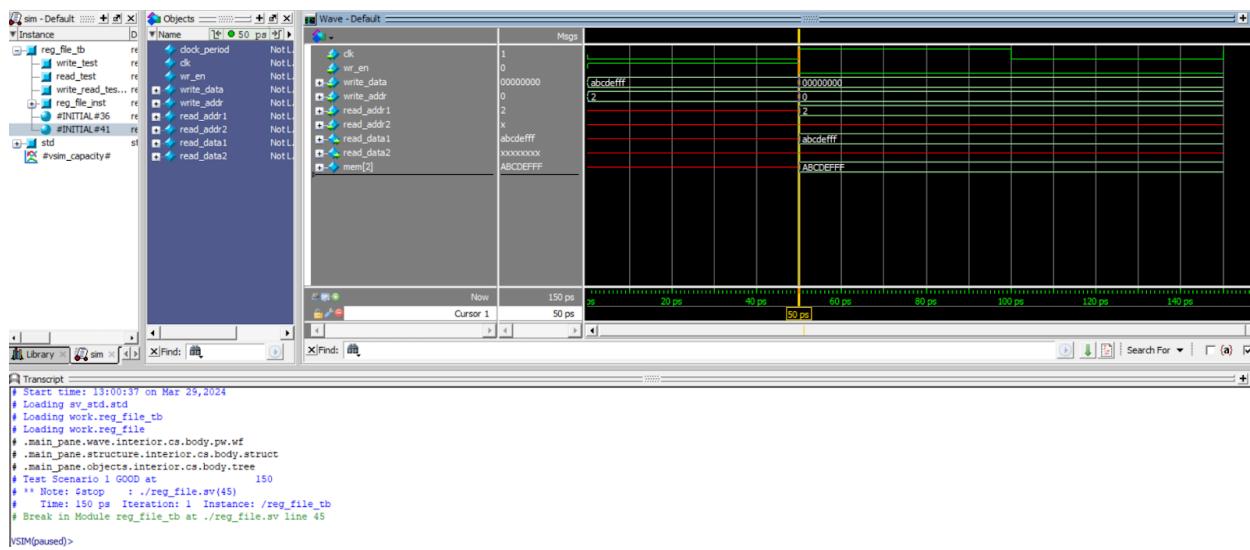
Figure_8: ModelSim arm.sv Task 2 Testbench Results

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

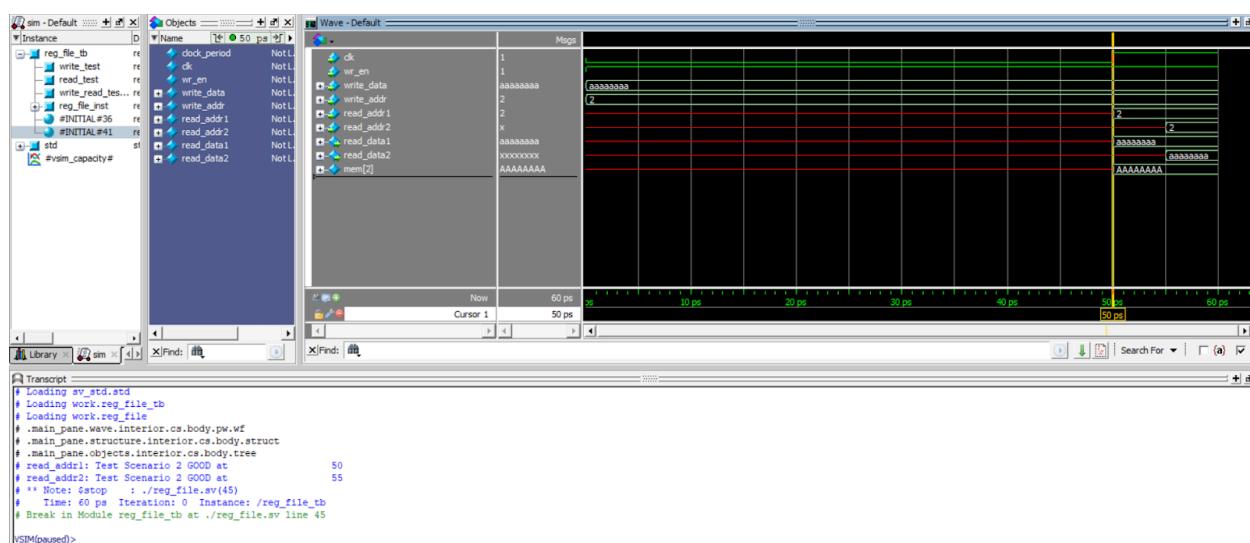
To ensure full coverage of the register's capability's, the following 3 scenarios were simulated:

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.



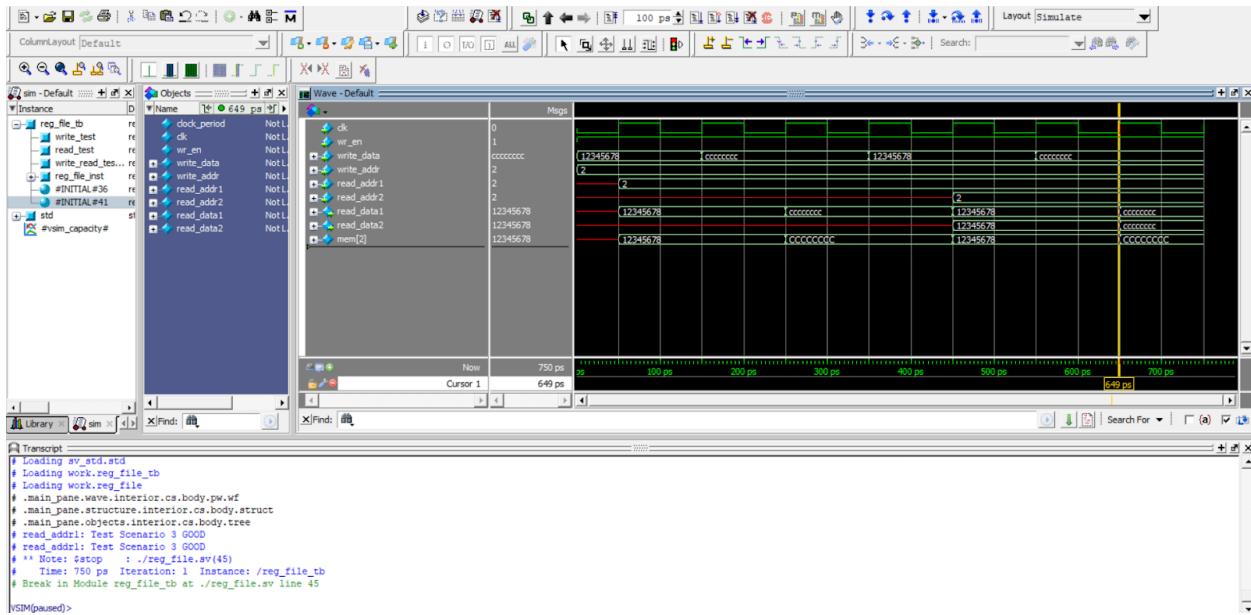
Figure_9: ModelSim reg_file Testbench (Scenario 1)

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)

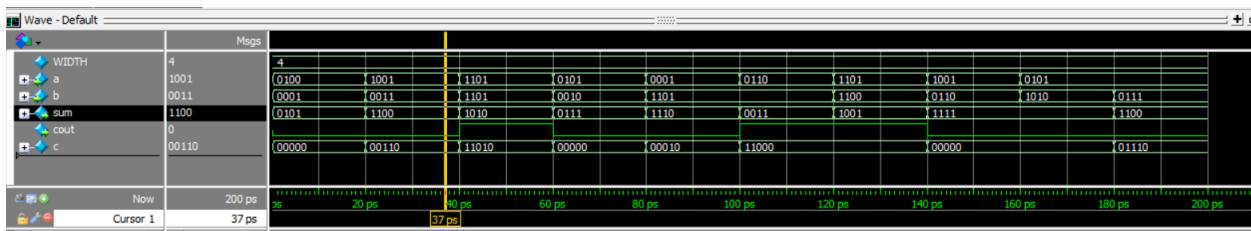
The Figure above shows the following scenario given in spec: Read data is updated the same cycle the address was provided. A verification code was written and can be referred to in **Appendix** Section, it ensures that the desired output is given at the correct time. As evident in the Terminal, we can 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



```

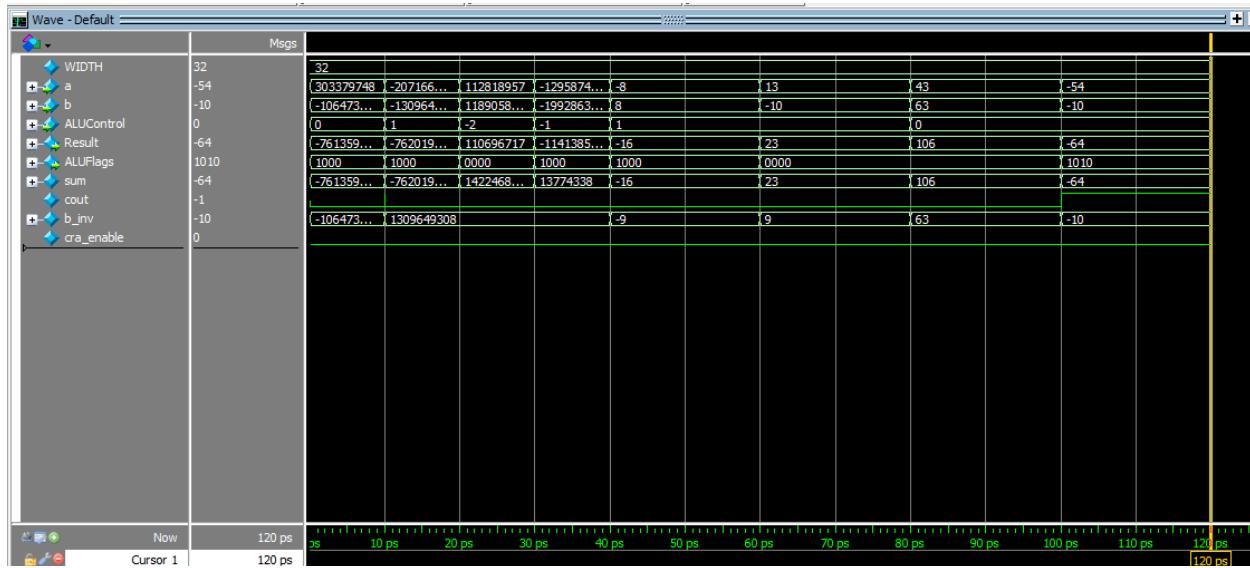
# .main_pane.objects.interior.cs.body.tree
# SUCCESSFUL ADDITION
# a: 4,b: 1,sum: 5,cout: 0
# SUCCESSFUL ADDITION
# a: 9,b: 3,sum: 12,cout: 0
# SUCCESSFUL ADDITION
# a: 13,b: 13,sum: 10,cout: 1
# SUCCESSFUL ADDITION
# a: 5,b: 2,sum: 7,cout: 0
# SUCCESSFUL ADDITION
# a: 1,b: 13,sum: 14,cout: 0
# SUCCESSFUL 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



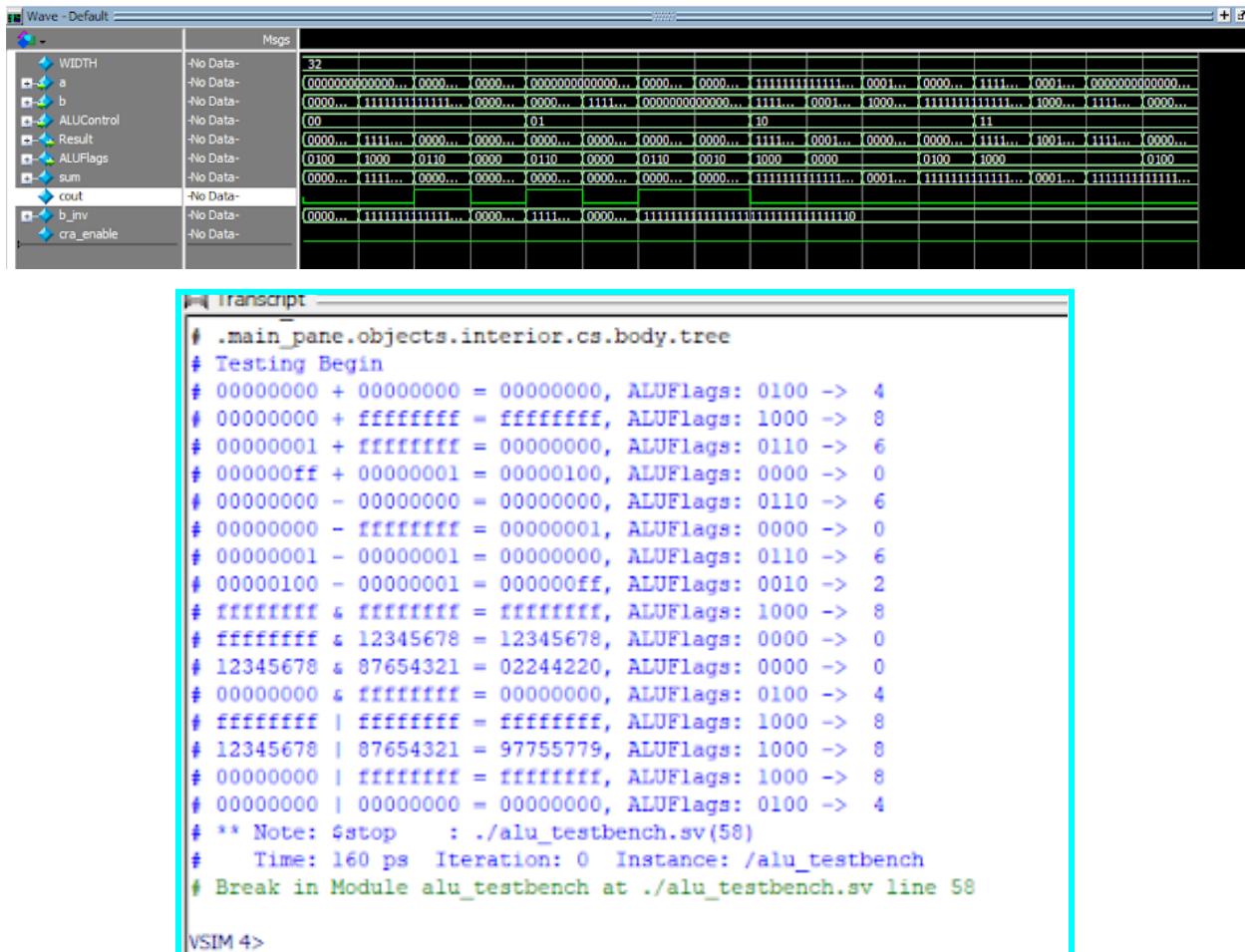
```

[1] Transcript
# Loading work.CarryRippleAdder
# Loading work.fullAdder
# ** Warning: (vsim-3839) ./alu.sv(37): Variable '/alu_tb/dut/cout', driven via a port connection, is multiply driven. See ./alu.sv(50).
#   Time: 0 ps Iteration: 0 Instance: ./alu_tb
# ** Warning: (vsim-WLF-5000) WLF file currently in use: vsim.wlf
#   File in use by: justi Hostname: ECCY ProcessID: 24532
#   Attempting to use alternate WLF file "./wlftecgwsg".
# ** Warning: (vsim-WLF-5001) Could not open WLF file: vsim.wlf
#   Using alternate file: ./wlftecgwsg
# .main_pane.wave.interior.cs.body.pw.wf
# .main_pane.structure.interior.cs.body.struct
# .main_pane.objects.interior.cs.body.tree
# SUCCESSFUL ADDITION a: 303379748, b: 3230228097,result: 3533607845 at: 10
# SUCCESSFUL SUBTRACTION a: 222328057, b: 2985317987,result: 3532947366 at: 20
# SUCCESSFUL AND a: 112818957, b: 1189058957,result: 110696717 at: 30
# SUCCESSFUL OR a: 2999092325 b: 2302104082,result: 3153581687 at: 40
# SUD OVERFLOW DETECTION SUCCESSFUL at: 80
# ADD OVERFLOW DETECTION SUCCESSFUL at: 120
# ** Note: $stop : ./alu.sv(190)
#   Time: 120 ps Iteration: 0 Instance: ./alu_tb
# Break in Module alu_tb at ./alu.sv line 190

```

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     // asynchronous read
29     assign rd_data = memory[addr[31:2]]; // word aligned, drop bottom 2 bits
30
31     // synchronous 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
8  carry-in bit.
9  It produces a sum and a carry-out bit.
10 The sum is the XOR of the inputs and the carry-in.
11 The carry-out is generated when two or more of the inputs are high.
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
15 ** alter
16 ** this line to use programs given to you as a part of this lab.
17 */
18 // addr - 32 bit address to determine the instruction to return. Note not all 32 bits are
19 // used since this
20 //           memory only has 64 words
21 // instr - 32 bit instruction to be sent to the processor
22 module imem(
23   input logic [31:0] addr,
24   output logic [31:0] instr
25 );
26   logic [31:0] memory [63:0];
27   // modify the name and potentially directory prefix of the file within to load the
28   // correct program and preprocessing
29   initial $readmemb("memfile2.dat", memory);
30
31   assign instr = memory[addr[31:2]]; // word aligned, drops bottom 2 bits
32
33 endmodule : imem
```

```

1 // ADD R - 1110_000_0100_0_AAAA_DDDD_0000_0000_BBBB
2 // ADD I - 1110_001_0100_0_AAAA_DDDD_0000_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_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_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
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
21 1110_001_0100_0_1111_0000_0000_0000_0000 // MAIN      ADD R0, R15, #0          0
22 1110_000_0010_0_0000_0001_0000_0000_0000 //           SUB R1, R0, R0          4
23 1110_001_0100_0_0001_0010_0000_0000_1010 //           ADD R2, R1, #10         8
24 1110_000_0100_0_0000_0011_0000_0000_0010 //           ADD R3, R0, R2         12
25 1110_001_0010_0_0010_0100_0000_0000_0011 //           SUB R4, R2, #3         16
26 1110_000_0010_0_0011_0101_0000_0000_0100 //           SUB R5, R3, R4         20
27 1110_000_1100_0_0100_0110_0000_0000_0101 //           ORR R6, R4, R5         24
28 1110_000_0000_0_0110_0111_0000_0000_0101 //           AND R7, R6, R5         28
29 1110_010_1100_0_0001_0111_0000_0000_0000 //           STR R7, [R1, #0]       32
30 1110_1010_0000_0000_0000_0000_0000_0001 //           B SKIP                  36
31 1110_010_1100_0_0001_0001_0000_0000_0000 //           STR R1, [R1, #0]       40
32 1110_1010_0000_0000_0000_0000_0000_0000 //           B LOOP                  44
33 1110_010_1100_1_0001_1000_0000_0000_0000 //           SKIP                   LDR R8, [R1, #0]        48
34 1110_001_0010_1_0110_1001_0000_0000_1111 //           B_START                CMP R9, R6, #15          52
35 0001_1010_1111_1111_1111_1111_1101 //           BNE B_START             56
36 1110_000_0010_1_0101_1001_0000_0000_0100 //           CMP R9, R5, R4          60
37 0001_1010_0000_0000_0000_0000_0000_0000 //           BNE BNE_TESTED          64
38 1110_1010_1111_1111_1111_1111_1010 //           B B_START               68
39 1110_000_0010_1_0010_1001_0000_0000_0011 //           BNE_TESTED              72
40 1010_1010_1111_1111_1111_1111_1000 //           CMP R9, R2, R3          76
41 1110_000_0010_1_0011_1001_0000_0000_0010 //           BGE B_START              80
42 1010_1010_0000_0000_0000_0000_0000_0000 //           CMP R9, R3, R2          84
43 1110_1010_1111_1111_1111_1111_0101 //           BGE BGE_TESTED            88
44 1110_000_0010_1_0011_1001_0000_0000_0010 //           BGE_TESTED               92
45 1101_1010_1111_1111_1111_1111_0011 //           BLE B_START               96
46 1110_000_0010_1_0010_1001_0000_0000_0011 //           CMP R9, R2, R3          100
47 1101_1010_0000_0000_0000_0000_0000_0000 //           BLE BLE_TESTED            104
48 1110_1010_1111_1111_1111_1111_0000 //           B B_START                 108
49 1110_001_0100_0_0001_1000_0000_0000_0001 //           BLE_TESTED                ADD R8, R1, #1          112
50 1110_1010_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     // contains 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
8  50 cycles
9  ** of instructions before terminating. At termination, specific register file values are
10 checked to
11 ** verify the processors' ability to execute the implemented instructions.
12 */
13 module testbench();
14
15     // system signals
16     logic clk, rst;
17
18     // generate clock with 100ps clk period
19     initial begin
20         clk = '1;
21         forever #50 clk = ~clk;
22     end
23
24     // processor instantiation. Within is the processor as well as imem and dmem
25     top cpu (.clk(clk), .rst(rst));
26
27     initial begin
28         // start with a basic reset
29         rst = 1; @(posedge clk);
30         rst <= 0; @(posedge clk);
31
32         // repeat for 50 cycles. Not all 50 are necessary, however a loop at the end of the
33         // program will keep anything weird from happening
34         repeat(50) @(posedge clk);
35
36         // basic checking to ensure the right final answer is achieved. These DO NOT prove
37         // your system works. A more careful look at your
38         // simulation and code will be made.
39
40         // task 1:
41         assert(cpu.processor.u_reg_file.mem[8] == 32'hb) $display("Task 1 Passed R8");
42         // else
43         assert(cpu.processor.u_reg_file.mem[7] == 32'hb) $display("Task 1 Failed R8");
44         // else
45         assert(cpu.processor.u_reg_file.mem[6] == 32'hf) $display("Task 1 Passed R7");
46         // else
47         assert(cpu.processor.u_reg_file.mem[5] == 32'hb) $display("Task 1 Failed R7");
48         // else
49         assert(cpu.processor.u_reg_file.mem[4] == 32'h7) $display("Task 1 Passed R6");
50         // else
51         assert(cpu.processor.u_reg_file.mem[3] == 32'h12) $display("Task 1 Failed R6");
52         // else
53         assert(cpu.processor.u_reg_file.mem[2] == 32'ha) $display("Task 1 Passed R5");
54         // else
55         assert(cpu.processor.u_reg_file.mem[1] == 32'h0) $display("Task 1 Failed R5");
56         // else
57         assert(cpu.processor.u_reg_file.mem[0] == 32'h8) $display("Task 1 Passed R4");
58         // else
59         // task 2:
60         assert(cpu.processor.u_reg_file.mem[8] == 32'd1) $display("Task 2 Passed");
61         // else
62         assert(cpu.processor.u_reg_file.mem[7] == 32'd0) $display("Task 2 Failed");
63         // verify skips
64         assert(cpu.processor.PC == 32'd40 || cpu.processor.PC == 32'd40) $display("Task 2
Failed SKIP at PC 36");
65         // else
66         assert(cpu.processor.PC == 32'd68) $display("Task 2
Passed SKIP at PC 36");
67         assert(cpu.processor.PC == 32'd68) $display("Task 2
Failed SKIP at PC 64");
68         // else
69         assert(cpu.processor.PC == 32'd88) $display("Task 2
Passed SKIP at PC 64");
70         assert(cpu.processor.PC == 32'd88) $display("Task 2
Failed SKIP at PC 84");

```

Date: April 18, 2024

testbench.sv

Project: DE1_SoC_Lab2

```
65      else
66          Passed SKIP at PC 84';
67          assert(cpu.processor.PC == 32'd108)
68      Failed SKIP at PC 104';
69      else
70          Passed SKIP at PC 104';
71          // verify end loops
72          repeat(5) assert(cpu.processor.PC == 32'd116)
73      Passed LOOP at PC 116 !';
74      else
75          Failed LOOP at PC 116';
76          $stop;
77      end
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
748
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
788
789
790
791
792
793
794
795
796
797
797
798
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
818
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
848
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
888
889
889
890
891
892
893
894
895
896
897
897
898
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
918
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
948
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
968
969
970
971
972
973
974
975
976
977
978
978
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1096
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1196
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1296
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1317
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1396
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1417
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1496
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1517
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1596
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1617
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1696
1697
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1796
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1896
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295

```

```

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

```

```

67      task write_test; begin
68          // Test write operation
69          wr_en = 1;
70          write_addr = 2;
71          write_data = 32'hABCDEFFF;
72          @(posedge clk);
73
74          // Test read operation in the next cycle
75          wr_en = 0;
76          write_addr = 0;
77          write_data = 0;
78          read_addr1 = 2;
79          @(posedge clk);
80
81          // Check if the written data is available in the next cycle
82          if(read_data1 != 32'hABCDEFFF) $fatal("Test Scenario 1 FAIL at", $time);
83          else $display("Test Scenario 1 GOOD at", $time);
84
85      end endtask : write_test
86
87 //-----
88 // Test scenario 2: Read data is updated the same cycle the address was provided
89 // "Read data is updated to the register data at an address the same cycle the address was
90 // provided. Do this for both read addresses and data outputs"
91
92 //-----
93 task read_test; begin
94
95     wr_en = 1;
96     write_addr = 2; write_data = 32'hAAAAAAA;
97     @(posedge clk);
98
99     read_addr1 = 2; if (read_data1 != 32'hAAAAAAA) $fatal("read_addr1: Test Scenario 2
100 FAIL at", $time); else $display("read_addr1: Test Scenario 2 GOOD at", $time); #5;
101     read_addr2 = 2; if (read_data2 != 32'hBBBBBBB) $fatal("read_addr2: Test Scenario 2
102 FAIL at", $time); else $display("read_addr2: Test Scenario 2 GOOD at", $time); #5;
103
104 end endtask
105
106 //-----
107 // Test scenario 3: Read data is updated to write data the cycle after the address was
108 // provided
109 // "Read data is updated to write data at an address the cycle after the address was
110 // provided
111 // if the write address is the same and wr_en was asserted. Do this for both read addresses
112 // and data outputs"
113
114 //-----
115 task write_read_test; begin
116     // Write operation
117     wr_en = 1;
118     write_addr = 2;
119     write_data = 32'h12345678;
120     @(posedge clk);
121     // provide a read address to same write_addr
122     read_addr1 = 2;
123     @(posedge clk);
124     // change write data at same address
125     wr_en = 1;
126     write_addr = 2;
127     write_data = 32'hcccccccc;
128     @(posedge clk);
129     @(posedge clk);
130
131     // 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
129      expected read_data: %h", read_data1, write_data);
130      else $display("read_addr1: Test Scenario 3 GOOD");
130
131      //-----
131      //--- Repeat for read_addr2--//
132      //-----
133
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
149      expected read_data: %h", read_data2, write_data);
150      else $display("read_addr1: Test Scenario 3 GOOD");
151
152 endtask
152
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 from 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 //-----//
45 // CarryRippleAdder Testbench
46 // It has 3 different tasks to verify functionality of CarryRippleAdder
47 // It includes a constrained Random Verification task called rand_vect_10
48 // It has an adjustable WIDTH to verify successful implementation of varying
49 // input sizes
50
51 module CarryRippleAdder_tb ();
52   parameter WIDTH=4;
53   //-----
54   // Define IN OUT for FUT
55   //-----
56   logic [WIDTH-1:0] a,b,sum;
57   logic cin, cout;
58
59   //-----
60   // Instantiate FullAdder
61   //-----
62   CarryRippleAdder #(WIDTH) dut(.*);
63
64
65   //-----
66   // Run Testing Procedures
67   //-----
68   initial begin
69     rand_vect_10; //SUCCESS
70     $stop;
71   end
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
14 // or immediates
15 // ReadData - data read out of the dmem
16 // WriteData - data to be written to the dmem
17 // MemWrite - write enable to allow WriteData to overwrite an existing dmem word
18 // PC - the current program count value, goes to imem to fetch instruction
19 // ALUResult - result of the ALU operation, sent as address to the dmem
20
21 module arm (
22     input logic      clk, rst,
23     input logic [31:0] Instr,
24     input logic [31:0] ReadData,
25     output logic [31:0] WriteData,
26     output logic [31:0] PC, ALUResult,
27     output logic      Memwrite
28 );
29
30     // datapath buses and signals
31     logic [31:0] PCPrime, PCPlus4, PCPlus8; // pc signals
32     logic [3:0] RA1, RA2; // regfile input addresses
33     logic [31:0] RD1, RD2; // raw regfile outputs
34     logic [3:0] ALUFlags; // alu combinational flag outputs
35     logic [31:0] ExtImm, SrcA, SrcB; // immediate and alu inputs
36     logic [31:0] Result; // computed or fetched value to be written into
37     regfile_or_pc
38     logic [3:0] FlagsReg; // register to store the flags from the most
39     recent CMP command
40
41     // control signals
42     // adding Flagwrite to control if loading the FlagsReg, control signal for the decoder
43     logic PCSrc, MemtoReg, ALUSrc, RegWrite, Flagwrite;
44     logic [1:0] RegSrc, ImmSrc, ALUControl;
45
46
47     /* The datapath consists of a PC as well as a series of muxes to make decisions about
48     which data words to pass forward and operate on. It is
49     ** noticeably missing the register file and alu, which you will fill in using the
50     ** modules made in lab 1. To correctly match up signals to the
51     ** ports of the register file and alu take some time to study and understand the logic
52     and flow of the datapath.
53 */
54
55 //-----
56 //----- DATAPATH -----
57
58
59 assign PCPrime = PCSrc ? Result : PCPlus4; // mux, use either default or newly
60 computed value
61 assign PCplus4 = PC + 'd4; // default value to access next instruction
62 assign PCplus8 = PCplus4 + 'd4; // value read when reading from reg[15]
63
64 // update the PC, at rst initialize to 0
65 always_ff @(posedge clk) begin
66     if (rst) PC <= '0;
67     else PC <= PCPrime;
68 end
69
70 // determine the register addresses based on control signals
71 // RegSrc[0] is set if doing a branch instruction
72 // RegSrc[1] is set when doing memory instructions
73 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
89          else if (ImmSrc == 'b01) ExtImm = {20'b0, Instr[11:0]};                // 12 bit
90          else                      ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00}; // 24 bit
91          immediate - branch operation
92      end
93
94      // WriteData and SrcA are direct outputs of the register file, wheras SrcB is chosen
95      // between reg file output and the immediate
96      assign WriteData = (RA2 == 'd15) ? PCPlus8 : RD2;                         // substitute the 15th
97      regfile register for PC
98      assign SrcA     = (RA1 == 'd15) ? PCPlus8 : RD1;                         // substitute the 15th
99      regfile register for PC
100     assign SrcB     = ALUSrc      ? ExtImm : WriteData;                      // determine alu operand to
101     be either from reg file or from immediate
102
103     // -----
104     // Instantiate ALU
105     // Inputs: a, b, ALUControl
106     // Outputs: Result, ALUFlags
107     // -----
108     alu u_alu (
109         .a           (SrcA),
110         .b           (SrcB),
111         .ALUControl (ALUControl),
112         .Result      (ALUResult),
113         .ALUFlags    (ALUFlags)
114     );
115
116
117     // update the FlagsReg if Flagwrite is enable
118     always_ff @(posedge clk) begin
119         if (Flagwrite) FlagsReg <= ALUFlags;
120         else          FlagsReg <= '0;
121     end
122
123     // determine the result to run back to PC or the register file based on whether we used
124     // a memory instruction
125     assign Result = MemtoReg ? ReadData : ALUResult; // determine whether final
126     writeback result is from dmemory or alu
127
128
129     /* The control consists of a large decoder, which evaluates the top bits of the
130      instruction and produces the control bits
131      ** which become the select bits and write enables of the system. The write enables
132      (Regwrite, Memwrite and PCSrc) are
133      ** especially important because they are representative of your processors current
134      state.
135      */
136      //-----
137      //----- CONTROL

```

```

128      //-----
129
130      always_comb begin
131          // check the instruction condition (Instr[31:28]) is valid before deciding to execute
132          // the branch
133          casez (Instr[31:20])
134
135              // ADD (Imm or Reg)
136              // note that we use wildcard "?" in bit 25. That bit decides whether we use
137              // immediate or reg, but regardless we add
138              12'b111000?01000 : begin
139                  PCSrc = 0;
140                  MemtoReg = 0;
141                  MemWrite = 0;
142                  ALUSrc = Instr[25]; // may use immediate
143                  RegWrite = 1;
144                  RegSrc = 'b00;
145                  ImmSrc = 'b00;
146                  ALUControl = 'b00;
147                  FlagWrite = 0;
148
149              // SUB (Imm or Reg)
150              12'b111000?00100 : begin
151                  PCSrc = 0;
152                  MemtoReg = 0;
153                  MemWrite = 0;
154                  ALUSrc = Instr[25]; // may use immediate
155                  RegWrite = 1;
156                  RegSrc = 'b00;
157                  ImmSrc = 'b00;
158                  ALUControl = 'b01;
159                  FlagWrite = 0;
160
161              // AND
162              12'b1110000000000 : 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
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
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

```

```

199
200      // STR
201      12'b1110001011000 : 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'b000001010???? : 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'b000011010???? : 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
8   * operations.
9   * It supports addition, subtraction, bitwise AND, and bitwise OR operations.
10  * For addition, it takes two input operands and produces a sum.
11  * For subtraction, it takes two input operands and produces a difference.
12  * For bitwise AND, it takes two input operands and produces a bitwise AND result.
13  * For bitwise OR, it takes two input operands and produces a bitwise OR result.
14  * The ALU operation is determined by a control signal.
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      assign ALUFlags[1] = (cout == 1'b1) ? 1: 0;
72      assign ALUFlags[2] = (Result == 0) ? 1: 0;
73      assign ALUFlags[3] = Result[WIDTH-1] ? 1: 0;
74
75  endmodule : alu
76
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 // Test Addition Overflow
164 ALUControl = 2'b00;
165
166 a = $urandom_range(1, 100);;
167 b = $urandom_range(1, 100);
168 #10;
169 if ((Result < 0) && (ALUControl[0] != 1)) begin
170   $fatal("FAIL TO SEE OVERFLOW: %d+%d!=%d", a,b,Result);
171 end
172 #10;
173
174 a = $urandom_range(-100, -1);
175 b = $urandom_range(-100, -1);
176 #10;
177 if ((Result[WIDTH-1] == 0) && (ALUControl[0] != 1)) begin
178   $fatal("FAIL TO SEE OVERFLOW at %t", $time);
179 end
180 #10;
181 $display("ADD OVERFLOW DETECTION SUCCESSFUL at: %t", $time);
182 $stop;
183
184 end // initial begin
185
186
187
188
189 endmodule : alu_tb
190

```