

Procedure:

This lab was comprised of two tasks:

1. Implementing the given arm CPU module with the ALU and reg_file that was created in Lab 1.
2. Add extra control path logic to implement CMP, B EQ, B NE, B GE, B GT, B LE, B LT instructions

Once the designs for both tasks were implemented, they were thoroughly tested in ModelSim using the given testbenches and the DAT files, to demonstrate their functionality.

Task #1:

The first task was to implement the given ARM CPU with the ALU and reg_file that were created in Lab 1. This was done by looking through the mapped-out diagram for the single-cycle CPU in the Lab 2 document and then aligning the proper signals that were already implemented in the ARM module with the inputs and outputs for the reg_file. The outputs from the reg_file were then properly aligned with the inputs of the ALU and then processed by the ALU and outputted for the memory or writeback sections. The schematic I implemented is shown in the diagram below.

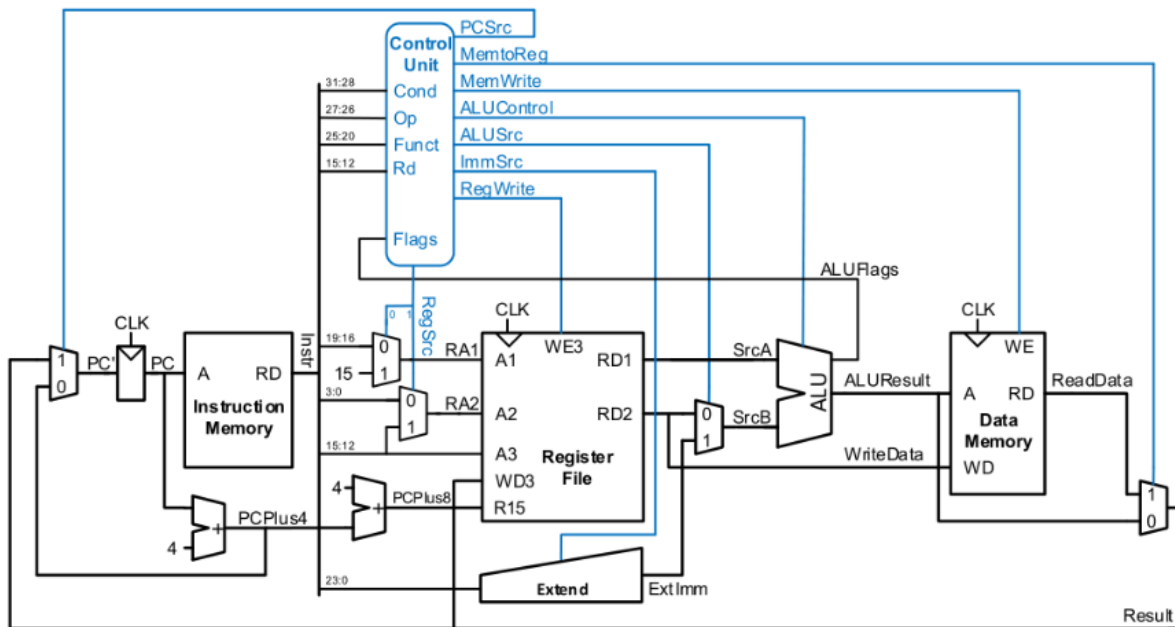


Figure 3. Single-cycle ARM processor

Figure 1: Schematic for a single-cycle ARM CPU

This schematic was then written and compiled in Quartus and then tested in ModelSim by varying the Instruction input based on the memfile.dat file.

The values of instructions and the resulting signals and outputs were tabulated to determine the proper expected values from the CPU as it goes through memfile.dat

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	5	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	0	1	12
5	16	SUB R4, R2, #3	A	3	7	12	X	0	1	7
6	20	SUB R5, R3, R4	12	7	B	7	X	0	1	B
7	24	ORR R6, R4, R5	7	B	F	B	X	0	1	F
8	28	AND R7, R6, R5	F	B	B	B	X	0	1	B
9	32	STR R7, [R1, #0]	0	0	0	B	X	1	0	0
10	36	B SKIP	2C	4	30	0	X	0	0	30
11	48	LDR R8, [R1, #0]	0	0	0	X	B	0	1	B
12	52	B LOOP	0	0	0	X	X	0	0	34
13	52	B LOOP	0	0	0	X	X	0	0	34
14	52	B LOOP	0	0	0	X	X	0	0	34
15	52	B LOOP	0	0	0	X	X	0	0	34
16	52	B LOOP	0	0	0	X	X	0	0	34
17	52	B LOOP	0	0	0	X	X	0	0	34
18	52	B LOOP	0	0	0	X	X	0	0	34
19	52	B LOOP	0	0	0	X	X	0	0	34

Table 3. First nineteen cycles of executing memfile.dat

Figure 2: Signals and output when running memfile.dat

Task #2:

The second task was to implement CMP, B EQ, B NE, B GE, B GT, B LE, and B LT instructions. This required modifying the control logic and data path to first store the flags from CMP instructions, then to modify the control logic sections to account for the conditional bits within the instruction to implement the new instructions.

Following the implementation of the new instructions, the modules were compiled in Quartus and then tested in ModelSim by varying the Instruction input based on the memfile2.dat file.

Before testing, the PC sequence was written out to determine what set of instructions should have been taken based on the conditional branching.



Figure 3: This is the PC sequence for memfile2.dat

The PC sequence displayed above in Figure 3 was compared with the simulation results to determine if the instructions were implemented correctly.

Results

Task #1:

After implementing the ARM module, I ran Modelsim to test it.

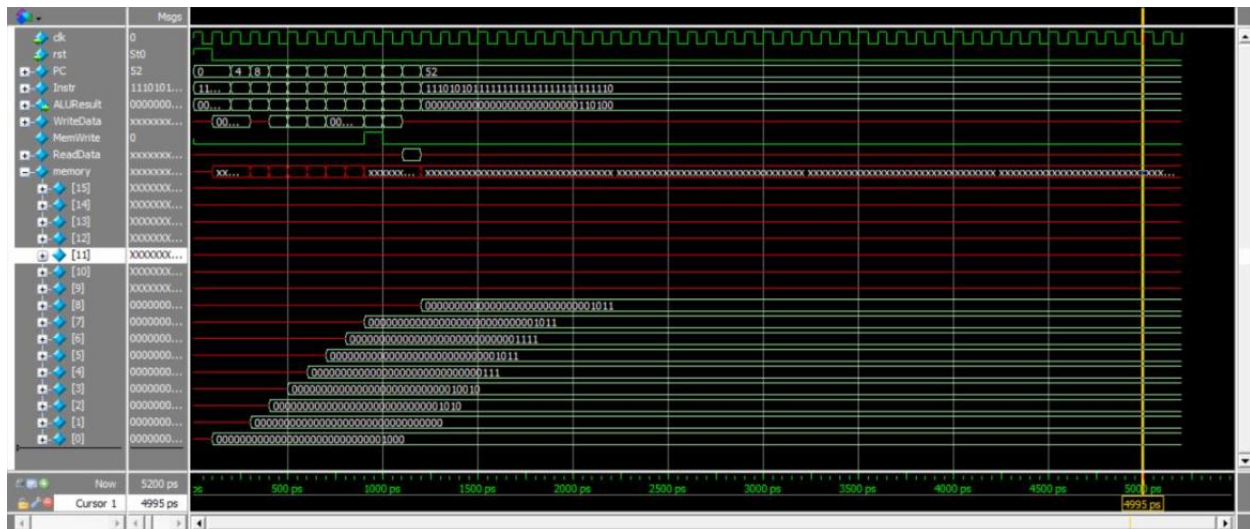


Figure 4: The waveform generated for the task 1 ARM module testbench (memfile.dat)

As seen in the waveforms above, the ARM CPU varies the values of the regfile based on the instructions executed, as a CPU should.

Task #2:

After implementing the new instructions in Quartus, I ran Modelsim to test it.

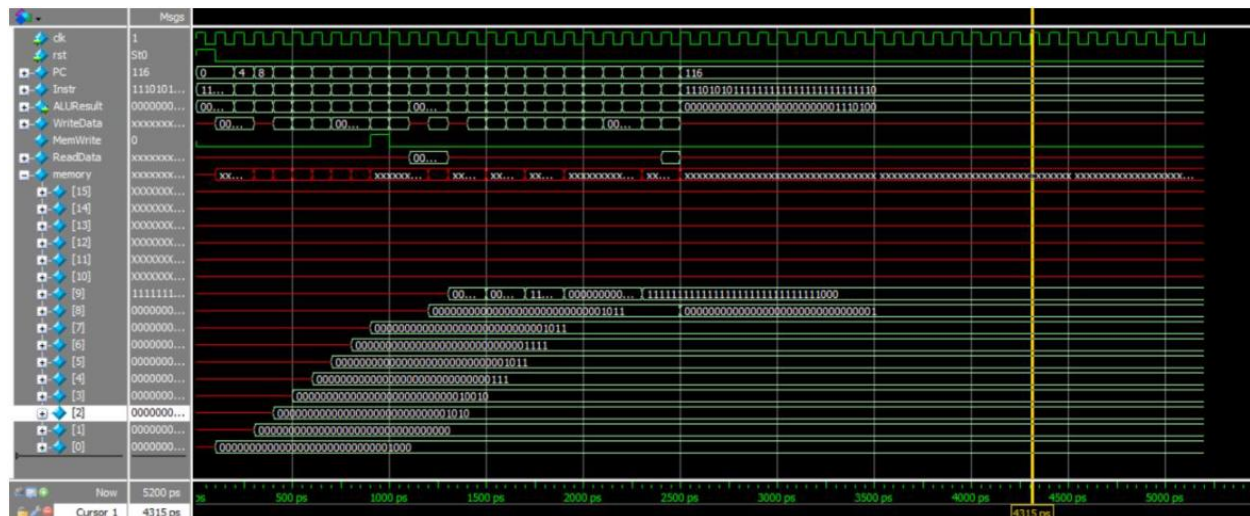


Figure 5: Modelsim waves for testing CMP, B EQ, B NE, B GE, B GT, B LE, B LT when implemented (memfile2.dat)

As seen in the waveforms above, the ARM CPU varies the values of the regfile based and PC based on the branches and instructions executed, as a CPU should.

Appendix

See the following list for the order:

top.sv

testbench.sv

dmem.sv

imem.sv

alu.sv

alu_testbench.sv

arm.sv

fullAdder.sv

reg_file.sv

reg_file_testbench.sv

singleALU.sv

See the attached documents for the code:

```

1  // Eugene Ngo
2  // 4/20/2023
3  // CSE 469
4  // Lab 2, Task 1 and 2
5
6  /* top is a structurally made toplevel module. It consists of 3 instantiations, as well as
   the signals that link them.
7  ** It is almost totally self-contained, with no outputs and two system inputs: clk and rst.
   clk represents the clock
8  ** the system runs on, with one instruction being read and executed every cycle. rst is the
   system reset and should
9  ** be run for at least a cycle when simulating the system.
10 */
11
12 // clk - system clock
13 // rst - system reset. Technically unnecessary
14 module top(
15     input logic clk, rst
16 );
17
18     // processor io signals
19     logic [31:0] Instr;
20     logic [31:0] ReadData;
21     logic [31:0] WriteData;
22     logic [31:0] PC, ALUResult;
23     logic
24         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
40     // effectively a rom because our processor cannot write to it
41     make 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 endmodule
57
58
59 // testbench tests the behaviors of the ALU by running through an instance of the
60 // top module. The top module instantiates the imeme module which calls on
61 // the memfile.dat and memfile2.dat files that send in the instruction inputs
62 // for the testbench module to use. The results are compared to actual expected
63 // values to determine if the functionality of the overall CPU is correct.
64 module testbench();
65
66     // system signals
67     logic clk, rst;
68
69     // generate clock with 100ps clk period

```

```
70     initial begin
71         clk = '1;
72         forever #50 clk = ~clk;
73     end
74
75     // processor instantiation. within is the processor as well as imem and dmem
76     top cpu (.clk(clk), .rst(rst));
77
78     initial begin
79         // start with a basic reset
80         rst = 1; @(posedge clk);
81         rst <= 0; @(posedge clk);
82
83         // repeat for 50 cycles. Not all 50 are necessary, however a loop at the end of the
84         program will keep anything weird from happening
85         repeat(50) @(posedge clk);
86
87         // basic checking to ensure the right final answer is achieved. These DO NOT prove
88         your system works. A more careful look at your
89         // simulation and code will be made.
90
91         // task 1:
92         // assert(cpu.processor.u_reg_file.memory[8] == 32'd11) $display("Task 1 Passed");
93         // else $display("Task 1 Failed");
94
95         // task 2:
96         assert(cpu.processor.u_reg_file.memory[8] == 32'd1) $display("Task 2 Passed");
97         else $display("Task 2 Failed");
98
99         $stop;
100     end
101 endmodule
```

```
1 // Eugene Ngo
2 // 4/20/2023
3 // CSE 469
4 // Lab 2, Task 1 and 2
5
6 /* testbench is a simulation module which simply instantiates the processor system and runs
7 50 cycles
8 ** of instructions before terminating. At termination, specific register file values are
9 checked to
10 ** verify the processors' ability to execute the implemented instructions.
11 */
12 module testbench();
13     // system signals
14     logic clk, rst;
15
16     // generate clock with 100ps clk period
17     initial begin
18         clk = '1;
19         forever #50 clk = ~clk;
20     end
21
22     // processor instantiation. 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
31         program will keep anything weird from happening
32         repeat(50) @(posedge clk);
33
34         // basic checking to ensure the right final answer is achieved. These DO NOT prove
35         your system works. A more careful look at your
36         // simulation and code will be made.
37
38         // task 1:
39         assert(cpu.processor.u_reg_file.memory[8] == 32'd11) $display("Task 1 Passed");
40         else $display("Task 1 Failed");
41
42         // task 2:
43         //assert(cpu.processor.u_reg_file.memory[8] == 32'd1) $display("Task 2 Passed");
44         //else $display("Task 2 Failed");
45
46         $stop;
47     end
48 endmodule
```



```
1  // Eugene Ngo
2  // 4/20/2023
3  // CSE 469
4  // Lab 2, Task 1 and 2
5
6  /* dmem is a more traditional, albeit very uninteresting, random access 64 word x 32 bit
   per word memory.
7  ** This module is also written in RTL, and likely strongly resembles your own register file
   except for a
8  ** few minor differences. The first is that there is only a single read port, compared to
   the register
9  ** file's two read ports. The other difference is that the dmem is also byte aligned, and
   therefore
10 ** discards the bottom two bits of the address when doing a read or write.
11 */
12
13 // clk - system clock, same as the processor
14 // wr_en - write enable, allows the wr_data to overwrite the 32 bit word stored in
   memory[addr]
15 // addr - the location to which you intend to read or write from
16 // wr_data - the 32 bit data word which you intend to write into memory
17 // rd_data - the data currently stored at memory[addr]
18 module dmem (
19     input logic          clk, wr_en,
20     input logic [31:0]   addr,
21     input logic [31:0]   wr_data,
22     output logic [31:0]  rd_data
23 );
24
25     logic [31:0] memory [63:0];
26
27     // asynchrnous read
28     assign rd_data = memory[addr[31:2]]; // word aligned, drop bottom 2 bits
29
30     // synchrononous gated write
31     always_ff @(posedge clk) begin
32         if (wr_en) memory[addr[31:2]] <= wr_data; // word aligned, drop bottom 2 bits
33     end
34
35 endmodule
```

```
1  // Eugene Ngo
2  // 4/20/2023
3  // CSE 469
4  // Lab 2, Task 1 and 2
5
6  /* imem is the read only, 64 word x 32 bit per word instruction memory for our processor.
7  ** Its module is written in RTL, and it strongly resembles a ROM (read only memory) or LUT
8  ** (look up table). This memory has no clock, and cannot be written to, but rather it
9  ** asynchronously reads out the word stored in its memory as soon as an address is given.
10 ** The address and memory are byte aligned, meaning that the bottom two bits are discarded
11 ** when looking for the word. One important line to note is the
12 **     initial $readmemb("memfile.dat", memory);
13 ** which determines the contents of the memory when the system is initialized. You will
14 ** this line to use programs given to you as a part of this lab.
15 */
16
17 // addr - 32 bit address to determine the instruction to return. Note not all 32 bits are
18 // 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
28     // correct program and preprocessing
29     initial $readmemb(
30         "C:\\Users\\egeen\\Desktop\\School\\EE 469\\Lab\\Lab 2\\memfile2.dat,"
31         memory);
32
33     assign instr = memory[addr[31:2]]; // word aligned, drops bottom 2 bits
34 endmodule
```

```

1  // Eugene Ngo
2  // 4/20/2023
3  // CSE 469
4  // Lab 1, Task 3
5
6  // A module to implement a 32 bit ARM-based ALU.
7  // This contains the central logic for calling on submodules
8  // that make up the ALU.
9
10 module alu(input logic [31:0] a, b,
11            input logic [1:0] ALUControl,
12            output logic [31:0] Result,
13            output logic [3:0] ALUFlags);
14
15    // 00 = add
16    // 01 = subtract
17    // 10 = AND
18    // 11 = OR
19
20    logic [31:0] carries;
21
22    // This initial call to the singleALU is to set the carry input for the
23    // genvar statements.
24    singleALU setCarries (.a(a[0]), .b(b[0]), .carryIn(ALUControl[0]),
25                        .ALUControl(ALUControl), .Result(Result[0]),
26                        .carryOut(carries[0]));
27
28    // The genvar statements break down the 32 bit inputs and sends them to
29    // 1 bit ALUs that add and subtract them based on the input control signal.
30    // The carry outs are processed and sent to the next bit that is covered,
31    // thus linking them and creating the actual 32 bit output value.
32    genvar i;
33    generate
34        for (i = 1; i < 32; i++) begin: ALUPipeline
35            singleALU results (.a(a[i]), .b(b[i]), .carryIn(carries[i - 1]),
36                            .ALUControl(ALUControl), .Result(Result[i]),
37                            .carryOut(carries[i]));
38        end // end loop
39    endgenerate // end generate
40
41    // Setting flags:
42    // Overflow is xor of carry[30], carry[31]
43    xor overFlowCheck (ALUFlags[0], carries[31], carries[30]);
44    // Carryout is the last carry.
45    assign ALUFlags[1] = carries[31];
46
47    // Inefficient and bad style. RTL would be better.
48    // Checks if any one of the bits within the 32-bit outputted value
49    // contains any 1s. If there is a single 1, the output zero flag is not
50    // raised.
51    nor zeroChecker
52        (ALUFlags[2], Result[31], Result[30], Result[29], Result[28],
53         Result[27], Result[26], Result[25], Result[24], Result[23],
54         Result[22], Result[21], Result[20], Result[19], Result[18],
55         Result[17], Result[16], Result[15], Result[14], Result[13],
56         Result[12], Result[11], Result[10], Result[9], Result[8],
57         Result[7], Result[6], Result[5], Result[4], Result[3],
58         Result[2], Result[1], Result[0]);
59
60    assign ALUFlags[3] = Result[31];
61
62 endmodule
63
64 // alu_testbench tests the behaviors of the ALU by running a .tv file
65 // through it. The results are compared to actual expected values to
66 // determine if the functionality of the ALU is correct.
67
68 module alu_testbench();
69     logic [31:0] a,b;
70     logic [1:0] ALUControl;
71     logic [31:0] Result;
72     logic [3:0] ALUFlags;
73     logic clk;
74     logic [103:0] testvectors [1000:0];

```

```
74
75 // Calls on the ALU module to test it.
76 alu dut (.a(a), .b(b), .ALUControl(ALUControl), .Result(Result),
77         .ALUFlags(ALUFlags));
78
79 parameter CLOCK_PERIOD = 100;
80
81 initial clk = 1;
82
83 // Generates a clock signal
84 always begin
85     #(CLOCK_PERIOD/2);
86     clk = ~clk;
87 end
88
89 // Reads through the .tv file and individually and test if the
90 // vector file values are the same as the values generated
91 // by the alu files.
92 initial begin
93     $readmemh("alu.tv", testvectors);
94
95     for (int i = 0; i < 20; i = i + 1) begin
96         {ALUControl, a, b, Result, ALUFlags} = testvectors[i];
97         @(posedge clk);
98     end // end loop
99 end // end initial
100 endmodule
101
```

```
1  // Eugene Ngo
2  // 4/7/2023
3  // CSE 469
4  // Lab 1 Task 3
5
6  // alu_testbench tests all expected, unexpected, and edgecase behaviors
7  module alu_testbench();
8      logic [31:0] a,b;
9      logic [1:0] ALUControl;
10     logic [31:0] Result;
11     logic [3:0] ALUFlags;
12     logic clk;
13     logic [103:0] testvectors [1000:0];
14
15     alu dut (.a(a), .b(b), .ALUControl(ALUControl), .Result(Result),
16             .ALUFlags(ALUFlags));
17
18     parameter CLOCK_PERIOD = 100;
19
20     initial clk = 1;
21
22     always begin
23         #(CLOCK_PERIOD/2);
24         clk = ~clk;
25     end
26
27     initial begin
28         $readmemh("alu.tv", testvectors);
29
30         for (int i = 0; i < 20; i = i + 1) begin
31             {ALUControl, a, b, Result, ALUFlags} = testvectors[i];
32             @(posedge clk);
33         end // end loop
34     end // end initial
35 endmodule
```

```

1  // Eugene Ngo
2  // 4/20/2023
3  // CSE 469
4  // Lab 2, Task 1 and 2
5
6  /* arm is the spotlight of the show and contains the bulk of the datapath and control
   logic. This module is split into two parts, the datapath and control.
7  */
8
9  // clk - system clock
10 // rst - system reset
11 // Instr - incoming 32 bit instruction from imem, contains opcode, condition, addresses and
   or immediates
12 // ReadData - data read out of the dmem
13 // WriteData - data to be written to the dmem
14 // MemWrite - write enable to allowed WriteData to overwrite an existing dmem word
15 // PC - the current program count value, goes to imem to fetch instruciton
16 // ALUResult - result of the ALU operation, sent as address to the dmem
17
18 module arm (
19     input logic clk, rst,
20     input logic [31:0] Instr,
21     input logic [31:0] ReadData,
22     output logic [31:0] WriteData,
23     output logic [31:0] PC, ALUResult,
24     output logic MemWrite
25 );
26
27 // datapath buses and signals
28 logic [31:0] PCPrime, PCPlus4, PCPlus8; // pc signals
29 logic [3:0] RA1, RA2; // regfile input addresses
30 logic [31:0] RD1, RD2; // raw regfile outputs
31 logic [3:0] ALUFlags; // alu combinational flag outputs
32 logic [3:0] FlagsReg; // register for storing flag outputs
33 logic [31:0] ExtImm, SrcA, SrcB; // immediate and alu inputs
34 logic [31:0] Result; // computed or fetched value to be written into
   regfile or pc
35
36 // control signals
37 logic PCSrc, MemtoReg, ALUSrc, RegWrite, FlagWrite;
38 logic [1:0] RegSrc, ImmSrc, ALUControl;
39
40
41 /* 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
42 ** 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
43 ** ports of the register file and alu take some time to study and understand the logic
   and flow of the datapath.
44 */
45 //-----
46 //                                     DATAPATH
47 //-----
48
49     assign PCPrime = PCSrc ? Result : PCPlus4; // mux, use either default or newly
   computed value
50
51     assign PCPlus4 = PC + 'd4; // default value to access next instruction
52     assign PCPlus8 = PCPlus4 + 'd4; // value read when reading from reg[15]
53
54 // update the PC, at rst initialize to 0
55 always_ff @(posedge clk) begin
56     if (rst) PC <= '0;
57     else PC <= PCPrime;
58 end
59
60 // determine the register addresses based on control signals
61 // RegSrc[0] is set if doing a branch instruction
62 // RefSrc[1] is set when doing memory instructions
63 assign RA1 = RegSrc[0] ? 4'd15 : Instr[19:16];
64 assign RA2 = RegSrc[1] ? Instr[15:12] : Instr[3:0];
65
66 // Takes the control signals from the control module, and combines them with the

```

```

67 // different input signals RA1, RA2, Instr[15:12], Result. It outputs the read data
68 // signals: RD1, RD2 and writes in and saves data into its own memory if the
69 // wr_en signal is enabled.
70 reg_file u_reg_file (
71     .clk          (clk),
72     .wr_en        (RegWrite),
73     .write_data    (Result),
74     .write_addr    (Instr[15:12]),
75     .read_addr1    (RA1),
76     .read_addr2    (RA2),
77     .read_data1    (RD1),
78     .read_data2    (RD2)
79 );
80
81 // two muxes, put together into an always_comb for clarity
82 // determines which set of instruction bits are used for the immediate
83 always_comb begin
84     if (ImmSrc == 'b00) ExtImm = {{24{Instr[7]}}, Instr[7:0]}; // 8 bit
immediate - reg operations
85     else if (ImmSrc == 'b01) ExtImm = {20'b0, Instr[11:0]}; // 12 bit
immediate - mem operations
86     else ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00}; // 24 bit
immediate - branch operation
87 end
88
89 // WriteData and SrcA are direct outputs of the register file, whereas SrcB is chosen
between reg file output and the immediate
90 assign WriteData = (RA2 == 'd15) ? PCPlus8 : RD2; // substitute the 15th
regfile register for PC
91 assign SrcA = (RA1 == 'd15) ? PCPlus8 : RD1; // substitute the 15th
regfile register for PC
92 assign SrcB = ALUSrc ? ExtImm : WriteData; // determine alu operand to
be either from reg file or from immediate
93
94 // Forwards the outputs from the register files and the selection logic above,
95 // to the ALU. The ALU computes a mathematical operation on the incoming values
96 // based on the inputs and outputs the result and the Flags that are triggered.
97 alu u_alu (
98     .a          (SrcA),
99     .b          (SrcB),
100     .ALUControl (ALUControl),
101     .Result      (ALUResult),
102     .ALUFlags    (ALUFlags)
103 );
104
105 // FlagsReg setting logic
106 // sets the flags if the Flagwrite control signal is enabled
107 always_ff @ (posedge clk) begin
108     if (FlagWrite) begin
109         FlagsReg <= ALUFlags;
110     end
111 end
112
113 // determine the result to run back to PC or the register file based on whether we used
a memory instruction
114 assign Result = MemtoReg ? ReadData : ALUResult; // determine whether final
writeback result is from dmemory or alu
115
116
117 /* The control consists of a large decoder, which evaluates the top bits of the
instruction and produces the control bits
** which become the select bits and write enables of the system. The write enables
(RegWrite, MemWrite and PCSrc) are
** especially important because they are representative of your processors current
state.
120 */
121 //-----
122 // CONTROL
123 //-----
124
125 always_comb begin
126     casez (Instr[31:20])
127

```

```

128 // ADD (Imm or Reg)
129 12'b111000?01000 : begin // note that we use wildcard "?" in bit 25. That bit
decides whether we use immediate or reg, but regardless we add
130     PCSrc = 0;
131     MemtoReg = 0;
132     MemWrite = 0;
133     ALUSrc = Instr[25]; // may use immediate
134     RegWrite = 1;
135     RegSrc = 'b00;
136     ImmSrc = 'b00;
137     ALUControl = 'b00;
138     FlagWrite = 0;
139 end
140
141 // SUB (Imm or Reg)
142 12'b111000?00100 : begin // note that we use wildcard "?" in bit 25. That bit
decides whether we use immediate or reg, but regardless we sub
143     PCSrc = 0;
144     MemtoReg = 0;
145     MemWrite = 0;
146     ALUSrc = Instr[25]; // may use immediate
147     RegWrite = 1;
148     RegSrc = 'b00;
149     ImmSrc = 'b00;
150     ALUControl = 'b01;
151     FlagWrite = 0;
152 end
153
154 // AND
155 12'b1110000000000 : begin
156     PCSrc = 0;
157     MemtoReg = 0;
158     MemWrite = 0;
159     ALUSrc = 0;
160     RegWrite = 1;
161     RegSrc = 'b00;
162     ImmSrc = 'b00; // doesn't matter
163     ALUControl = 'b10;
164     FlagWrite = 0;
165 end
166
167 // ORR
168 12'b111000011000 : begin
169     PCSrc = 0;
170     MemtoReg = 0;
171     MemWrite = 0;
172     ALUSrc = 0;
173     RegWrite = 1;
174     RegSrc = 'b00;
175     ImmSrc = 'b00; // doesn't matter
176     ALUControl = 'b11;
177     FlagWrite = 0;
178 end
179
180 // LDR
181 12'b111001011001 : begin
182     PCSrc = 0;
183     MemtoReg = 1;
184     MemWrite = 0;
185     ALUSrc = 1;
186     RegWrite = 1;
187     RegSrc = 'b10; // msb doesn't matter
188     ImmSrc = 'b01;
189     ALUControl = 'b00; // do an add
190     FlagWrite = 0;
191 end
192
193 // STR
194 12'b111001011000 : begin
195     PCSrc = 0;
196     MemtoReg = 0; // doesn't matter
197     MemWrite = 1;
198     ALUSrc = 1;

```



```

199         RegWrite = 0;
200         RegSrc   = 'b10;    // msb doesn't matter
201         ImmSrc   = 'b01;
202         ALUControl = 'b00;  // do an add
203         FlagWrite = 0;
204     end
205
206     // B
207     12'b11101010???? : begin
208         PCSrc = 1;
209         MemtoReg = 0;
210         MemWrite = 0;
211         ALUSrc = 1;
212         RegWrite = 0;
213         RegSrc = 'b01;
214         ImmSrc = 'b10;
215         ALUControl = 'b00;  // do an add
216         FlagWrite = 0;
217     end
218
219     // CMP
220     12'b111000?00101 : begin
221         PCSrc = 0;
222         MemtoReg = 0;
223         MemWrite = 0;
224         ALUSrc = Instr[25]; // may use immediate
225         RegWrite = 1;
226         RegSrc = 'b00;
227         ImmSrc = 'b00;
228         ALUControl = 'b01;
229         FlagWrite = 1;
230     end
231
232     // B EQ
233     12'b00001010???? : begin
234         PCSrc = FlagsReg[2];
235         MemtoReg = 0;
236         MemWrite = 0;
237         ALUSrc = 1;
238         RegWrite = 0;
239         RegSrc = 'b01;
240         ImmSrc = 'b10;
241         ALUControl = 'b00;  // do an add
242         FlagWrite = 0;
243     end
244
245     // B NE
246     12'b00011010???? : begin
247         PCSrc = ~ FlagsReg[2];
248         MemtoReg = 0;
249         MemWrite = 0;
250         ALUSrc = 1;
251         RegWrite = 0;
252         RegSrc = 'b01;
253         ImmSrc = 'b10;
254         ALUControl = 'b00;  // do an add
255         FlagWrite = 0;
256     end
257
258     // B GE
259     12'b10101010???? : begin
260         PCSrc = ~ FlagsReg[3] | FlagsReg[2];
261         MemtoReg = 0;
262         MemWrite = 0;
263         ALUSrc = 1;
264         RegWrite = 0;
265         RegSrc = 'b01;
266         ImmSrc = 'b10;
267         ALUControl = 'b00;  // do an add
268         FlagWrite = 0;
269     end
270
271     // B GT

```

```

272 12'b11001010???? : begin
273     PCSrc      = ~ FlagsReg[3];
274     MemtoReg    = 0;
275     MemWrite    = 0;
276     ALUSrc      = 1;
277     RegWrite    = 0;
278     RegSrc      = 'b01;
279     ImmSrc      = 'b10;
280     ALUControl  = 'b00; // do an add
281     FlagWrite   = 0;
282 end
283
284 // B LE
285 12'b11011010???? : begin
286     PCSrc      = FlagsReg[3] | FlagsReg[2];
287     MemtoReg    = 0;
288     MemWrite    = 0;
289     ALUSrc      = 1;
290     RegWrite    = 0;
291     RegSrc      = 'b01;
292     ImmSrc      = 'b10;
293     ALUControl  = 'b00; // do an add
294     FlagWrite   = 0;
295 end
296
297 // B LT
298 12'b10111010???? : begin
299     PCSrc      = FlagsReg[3];
300     MemtoReg    = 0;
301     MemWrite    = 0;
302     ALUSrc      = 1;
303     RegWrite    = 0;
304     RegSrc      = 'b01;
305     ImmSrc      = 'b10;
306     ALUControl  = 'b00; // do an add
307     FlagWrite   = 0;
308 end
309
310 default: begin
311     PCSrc      = 0;
312     MemtoReg    = 0; // doesn't matter
313     MemWrite    = 0;
314     ALUSrc      = 0;
315     RegWrite    = 0;
316     RegSrc      = 'b00;
317     ImmSrc      = 'b00;
318     ALUControl  = 'b00; // do an add
319     FlagWrite   = 0;
320 end
321 endcase
322 end
323 endmodule

```

```
1  // Eugene Ngo
2  // 4/20/2023
3  // CSE 469
4  // Lab 1 Task 3
5
6  // A module to implement a fullAdder.
7  // This file was reused from a given file in EE 371
8  module fullAdder (a, b, carryIn, Result, carryOut);
9
10     input logic a, b, carryIn;
11     output logic Result, carryOut;
12
13     assign Result = a ^ b ^ carryIn;
14     assign carryOut = (a & b) | (carryIn & (a ^ b));
15
16 endmodule
```

```

1  // Eugene Ngo
2  // 4/20/2023
3  // CSE 469
4  // Lab 1, Task 2
5
6  // A module to implement a 32 bit, 16 register, reg_file.
7  // This module takes in data and stores it in flip flops.
8  // The data can then be read via two read inputs and outputs.
9  // The data can be written in via 1 write input and 1 write enable.
10
11 module reg_file (input logic clk, wr_en, input logic [31:0] write_data,
12                 input logic [3:0] write_addr,
13                 input logic [3:0] read_addr1, read_addr2,
14                 output logic [31:0] read_data1, read_data2);
15
16     // Stores all the values.
17     logic [15:0][31:0] memory;
18
19     // Main logic unit. Clocked flip flops.
20     always_ff @ (posedge clk) begin
21         // if write enabled, write data
22         if (wr_en) begin
23             memory[write_addr] <= write_data;
24         end
25     end
26
27     // Read out data the instant the read_data signals are updated.
28     always_comb begin
29         read_data1 = memory[read_addr1];
30         read_data2 = memory[read_addr2];
31     end
32
33 endmodule
34
35 // reg_file_testbench tests the behaviors of the reg_file by inputting
36 // expected testing values. This tests for cycle delays in writes,
37 // same-cycle reads, and 1 cycle delays for reads occurring after writes.
38 // The results are compared to expected values to determine if the
39 // functionality of the reg_file is correct.
40
41 module reg_file_testbench();
42     logic clk, wr_en;
43     logic [31:0] write_data, read_data1, read_data2;
44     logic [3:0] write_addr, read_addr1, read_addr2;
45
46     // Calls on the reg_file module to test it.
47     reg_file dut (.clk, .wr_en, .write_data, .write_addr, .read_addr1, .read_addr2, .
48 read_data1, .read_data2);
49
50     parameter clock_period = 10000;
51
52     integer i;
53
54     initial begin // Set up the clock
55         clk <= 0;
56         for (i=0; i<1000; i++) begin: clockCount
57             forever #(clock_period/2) clk <= ~clk;
58         end
59     end
60
61     initial begin
62         $display("%t Behavior check", $time);
63
64         // Testing functionality of
65         // 1 cycle delay of writes.
66         wr_en = 1'b1; @ (posedge clk);
67         write_data = 32'b0; @ (posedge clk);
68         write_addr = 4'b0010; @ (posedge clk);
69         @ (posedge clk);
70
71         write_data = 32'b1; @ (posedge clk);
72         write_addr = 4'b0011; @ (posedge clk);

```

```
73                                     @(posedge clk);
74         // Testing functionality of
75         // same cycle reads.
76         read_addr1 = 4'b0010;         @(posedge clk);
77         read_addr2 = 4'b0011;         @(posedge clk);
78
79         // Testing the functionality of
80         // 1 cycle delayed reads
81         // after an updated write
82         write_addr = 4'b0010;         @(posedge clk);
83         read_addr1 = 4'b0010;         @(posedge clk);
84                                     @(posedge clk);
85         // read_data1 should update to 1 now.
86
87         write_data = 32'b0;           @(posedge clk);
88         write_addr = 4'b0011;         @(posedge clk);
89
90         read_addr2 = 4'b0011;         @(posedge clk);
91                                     @(posedge clk);
92         // read_data2 should update to 0 now.
93
94     end
95
96 endmodule
```

```
1 // Eugene Ngo
2 // 4/7/2023
3 // CSE 469
4 // Lab 1 Task 2
5
6 // reg_file_testbench tests all expected, unexpected, and edgecase behaviors
7 module reg_file_testbench();
8     logic clk, wr_en;
9     logic [31:0] write_data, read_data1, read_data2;
10    logic [3:0] write_addr, read_addr1, read_addr2;
11
12    reg_file dut (.clk, .wr_en, .write_data, .write_addr, .read_addr1, .read_addr2, .
13    read_data1, .read_data2);
14
15    parameter clock_period = 10000;
16
17    integer i;
18
19    initial begin // Set up the clock
20        clk <= 0;
21        for (i=0; i<1000; i++) begin: clockCount
22            forever #(clock_period/2) clk <= ~clk;
23        end
24    end
25
26    initial begin
27        $display("%t Behavior check", $time);
28
29        // Testing functionality of
30        // 1 cycle delay of writes.
31        wr_en = 1'b1; @ (posedge clk);
32        write_data = 32'b0; @ (posedge clk);
33        write_addr = 4'b0010; @ (posedge clk);
34        @ (posedge clk);
35
36        write_data = 32'b1; @ (posedge clk);
37        write_addr = 4'b0011; @ (posedge clk);
38        @ (posedge clk);
39
40        // Testing functionality of
41        // same cycle reads.
42        read_addr1 = 4'b0010; @ (posedge clk);
43        read_addr2 = 4'b0011; @ (posedge clk);
44
45        // Testing the functionality of
46        // 1 cycle delayed reads
47        // after an updated write
48        write_addr = 4'b0010; @ (posedge clk);
49        read_addr1 = 4'b0010; @ (posedge clk);
50        @ (posedge clk);
51        // read_data1 should update to 1 now.
52
53        write_data = 32'b0; @ (posedge clk);
54        write_addr = 4'b0011; @ (posedge clk);
55
56        read_addr2 = 4'b0011; @ (posedge clk);
57        @ (posedge clk);
58        // read_data2 should update to 0 now.
59
60
61    end
62
63
64
65
66
67 endmodule
68
```

```
1 // Eugene Ngo
2 // 4/20/2023
3 // CSE 469
4 // Lab 2, Task 1 and 2
5
6 // A module to implement ALU logic for single bits.
7 // It takes in single bit values and adds, subtracts, performs OR or AND
8 // based on the ALUControl signals. The functions are all performed at once
9 // and then the outputted value is selected.
10 // Combine single bit ALUs to make up large ALUs.
11 module singleALU (a, b, carryIn, ALUControl, Result, carryOut);
12     input logic a, b, carryIn;
13     input logic [1:0] ALUControl;
14     output logic Result, carryOut;
15
16
17     logic [2:0] outputs;
18
19     and andValue (outputs[1], b, a);
20     or orValue (outputs[2], b, a);
21
22     // Selecting B (Addition = A+B+0, Subtraction = A+(~B)+1)
23     // logic subtractSelector = ALUControl[0];
24
25     // MUX to select B (ALUControl[0] == 1 = select ~B)
26     wire middleValues[1:0];
27     and selectB (middleValues[0], ~ALUControl[0], b);
28     and selectNotB (middleValues[1], ALUControl[0], ~b);
29
30     logic selectedB;
31     or selectedBValue (selectedB, middleValues[0], middleValues[1]);
32
33     fullAdder fullAddedValue (.a(a), .b(selectedB), .carryIn(carryIn),
34                             .Result(outputs[0]), .carryOut(carryOut));
35
36     // MUX to select between computed values (add, sub, and, or)
37
38     always_comb begin
39         case (ALUControl)
40             2'b00: Result = outputs[0];
41             2'b01: Result = outputs[0];
42             2'b10: Result = outputs[1];
43             2'b11: Result = outputs[2];
44             default: Result = 1'bx;
45         endcase // end case statements
46     end // end comb block
47
48 endmodule // End of module
```