Stephen Macris
Aaron Hong
EE469 Lab 2
4/21/2023

Procedure

Task 1

In task 1 we were given multiple files that were all part of a single-cycle processor. The files given were *top.sv, imem.sv, dmem.sv, arm.sv,* and *testbench.sv.* We were tasked with implementing our register file and ALU that were made in 1 into the data path of the arm processor. To do this it was very important to study and understand how the datapath worked, studying the schematic of the processor was of great assistance for this part, and is pictured below. After the register file and ALU were implemented, we then had to test to verify it was working correctly.

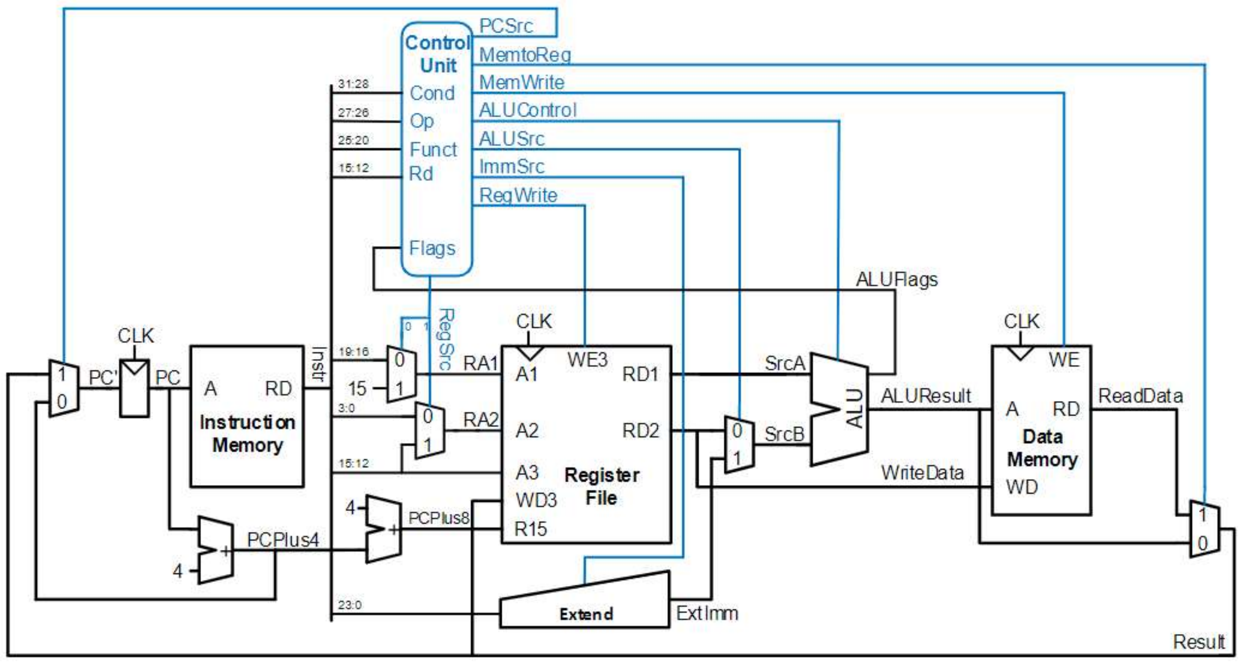

Figure 1: processer schematic

Stephen Macris
Aaron Hong
EE469 Lab 2
4/21/2023

Cycle	PC	Instr	SrcA	SrcB	ALUResult	WriteData	ReadData	MemWrite	RegWrite	Result
1	**0**	**ADD R0, R15, #0**	**8**	**0**	**8**	**8**	**X**	**0**	**1**	**8**
2	**4**	**SUB R1, R0, R0**	**8**	**8**	**0**	**8**	**X**	**0**	**1**	**0**
3	**8**	**ADD R2, R1, #10**	**0**	**A**	**A**	**X**	**X**	**0**	**1**	**A**
4	**12**	**ADD R3, R0, R2**	**8**	**A**	**12**	**X**	**X**	**0**	**1**	**12**
5	**16**	**SUB R4, R2, #3**	**A**	**3**	**7**	**X**	**X**	**0**	**1**	**7**
6	**20**	**SUB R5, R3, R4**	**12**	**7**	**B**	**X**	**X**	**0**	**1**	**B**
7	**24**	**ORR R6, R4, R5**	**7**	**B**	**F**	**X**	**X**	**0**	**1**	**F**
8	**28**	**AND R7, R6, R5**	**F**	**B**	**B**	**X**	**X**	**0**	**1**	**B**
9	**32**	**STR R7, [R1, #0]**	**X**	**X**	**X**	**X**	**X**	**0**	**0**	**X**
10	**36**	**B SKIP**	**X**	**X**	**X**	**X**	**X**	**0**	**0**	**X**
11	**48**	**LDR R8, [R1, #0]**	**X**	**X**	**X**	**X**	**B**	**1**	**1**	**X**
12	**52**	**B LOOP**	**X**	**X**	**X**	**X**	**X**	**0**	**0**	**X**
13	**52**	**B LOOP**	**X**	**X**	**X**	**X**	**X**	**0**	**0**	**X**
14	**52**	**B LOOP**	**X**	**X**	**X**	**X**	**X**	**0**	**0**	**X**
15	**52**	**B LOOP**	**X**	**X**	**X**	**X**	**X**	**0**	**0**	**X**
16	**52**	**B LOOP**	**X**	**X**	**X**	**X**	**X**	**0**	**0**	**X**
17	**52**	**B LOOP**	**X**	**X**	**X**	**X**	**X**	**0**	**0**	**X**
18	**52**	**B LOOP**	**X**	**X**	**X**	**X**	**X**	**0**	**0**	**X**
19	**52**	**B LOOP**	**X**	**X**	**X**	**X**	**X**	**0**	**0**	**X**

Table 1: First nineteen cycles of memfile.dat

Stephen Macris
Aaron Hong
EE469 Lab 2
4/21/2023

Task 2

In this task we had to implement new instructions to the processor. We implemented the CMP instruction which gave purpose to ALUFLags. Comparator instructions such as EQ, NE, GE, GT, LE, and LT were also implemented. Once these new instructions were implemented, a more complicated test was given to ensure that the instructions were implemented correctly, and the processor is functioning as intended.

The PC cycle of memfile2.dat is:

0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 48, 52, 56, 60, 64, 72, 76, 80, 84, 92, 96, 100, 104, 112, 116

Results

Task 1

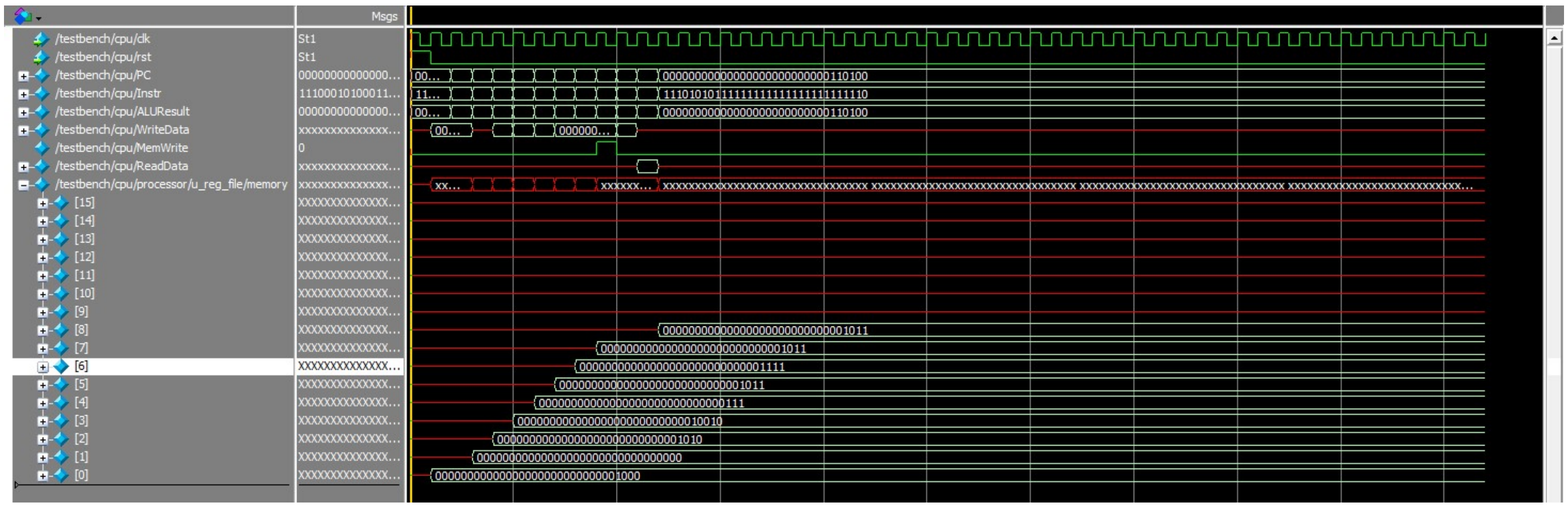

Figure 2: Task 1 simulation

The screenshot above shows the simulation results from ModelSim after the ALU and register file were implemented. In this test, memfile.dat was used. The purpose of the simulation was to test to see if the ALU and register were implemented correctly, and if the processor is functioning properly. The results from the screenshot demonstrate that the implementation is correct.

Stephen Macris
Aaron Hong
EE469 Lab 2
4/21/2023
Task 2

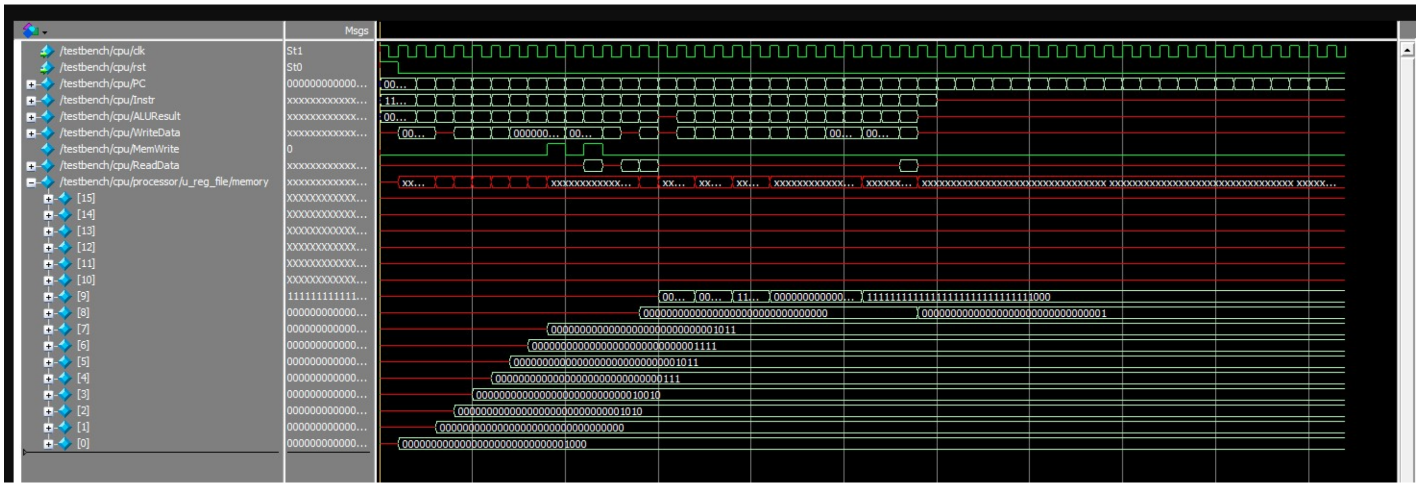

Figure 3: Task 2 simulation

The screenshot above shows the waveforms from the simulated processor after the additional instructions were added. In this simulation, memfile2 was used to test if the instructions were implemented correctly. The results from the screenshot show that the new instructions were implemented correctly, and the processor is operating as expected.

Appendix

```systemverilog
/* arm is the spotlight of the show and contains the bulk of the datapath and control logic. This module is split into two parts, the datapath and control.
*/

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

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

    // datapath buses and signals
    logic [31:0] PCPrime, PCPlus4, PCPlus8; // pc signals
    logic [ 3:0] RA1, RA2;                  // regfile input addresses
    logic [31:0] RD1, RD2;                  // raw regfile outputs
    logic [ 3:0] ALUFlags;                  // alu combinational flag outputs
    logic [31:0] ExtImm, SrcA, SrcB;        // immediate and alu inputs
    logic [31:0] Result;                    // computed or fetched value to be written into regfile or pc

    // control signals
    logic PCSrc, MemtoReg, ALUSrc, RegWrite, FlagWrite;
    logic [1:0] RegSrc, ImmSrc, ALUControl;
    logic [3:0] FlagsReg, cond;


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


    assign PCPrime = PCSrc ? Result : PCPlus4;  // mux, use either default or newly computed value
    assign PCPlus4 = PC + 'd4;                  // default value to access next instruction
    assign PCPlus8 = PCPlus4 + 'd4;             // value read when reading from reg[15]

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

    // determine the register addresses based on control signals
    // RegSrc[0] is set if doing a branch instruction
    // RefSrc[1] is set when doing memory instructions
    assign RA1 = RegSrc[0] ? 4'd15        : Instr[19:16];
    assign RA2 = RegSrc[1] ? Instr[15:12] : Instr[ 3: 0];

    // Instantiates a register file to hold values.
    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)
    );

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

    // WriteData and SrcA are direct outputs of the register file, wheras SrcB is chosen between reg file output and the immediate
    assign WriteData = (RA2 == 'd15) ? PCPlus8 : RD2;           // substitute the 15th regfile register for PC
    assign SrcA      = (RA1 == 'd15) ? PCPlus8 : RD1;           // substitute the 15th regfile register for PC
    assign SrcB      = ALUSrc        ? ExtImm  : WriteData;     // determine alu operand to be either from reg file or from immediate


    // Instantiates an alu module to do arithmetic operations.
    alu u_alu (
        .a          (SrcA),
        .b          (SrcB),
        .ALUControl (ALUControl),
        .Result     (ALUResult),
        .ALUFlags   (ALUFlags)
    );

    // determine the result to run back to PC or the register file based on whether we used a memory instruction
    assign Result = MemtoReg ? ReadData : ALUResult;    // determine whether final writeback result is from dmemory or alu

    always_ff @(posedge clk) begin
         if (FlagWrite) FlagsReg = ALUFlags;
    end

    /* The control conists of a large decoder, which evaluates the top bits of the instruction and produces the control bits
    ** which become the select bits and write enables of the system. The write enables (RegWrite, MemWrite and PCSrc) are
    ** especially important because they are representative of your processors current state.
    */
    //------------------------------------------------------------------------------
    //                                      CONTROL
    //------------------------------------------------------------------------------
    assign cond = Instr[31:28];

    always_comb begin
        casez (Instr[27:20])

            // ADD (Imm or Reg)
            8'b00?_0100_0 : begin   // note that we use wildcard "?" in bit 25. That bit decides whether we use immediate or reg, but regardless we add
                PCSrc    = 0;
                MemtoReg = 0;
                MemWrite = 0;
                ALUSrc   = Instr[25]; // may use immediate
                RegWrite = 1;
                RegSrc   = 'b00;
                ImmSrc   = 'b00;
                ALUControl = 'b00;
                FlagWrite = 0;
            end

            // SUB (Imm or Reg)
            8'b00?_0010_0 : begin   // note that we use wildcard "?" in bit 25. That bit decides whether we use immediate or reg, but regardless we sub
                PCSrc    = 0;
                MemtoReg = 0;
                MemWrite = 0;
                ALUSrc   = Instr[25]; // may use immediate
                RegWrite = 1;
                RegSrc   = 'b00;
                ImmSrc   = 'b00;
                ALUControl = 'b01;
                FlagWrite = 0;
            end

            // AND
            8'b000_0000_0  : begin
                PCSrc    = 0;
                MemtoReg = 0;
                MemWrite = 0;
                ALUSrc   = 0;
                RegWrite = 1;
                RegSrc   = 'b00;
                ImmSrc   = 'b00;    // doesn't matter
                ALUControl = 'b10;
               FlagWrite = 0;
            end

            // ORR
            8'b000_1100_0  : begin
                PCSrc    = 0;
                MemtoReg = 0;
                MemWrite = 0;
                ALUSrc   = 0;
                RegWrite = 1;
                RegSrc   = 'b00;
                ImmSrc   = 'b00;    // doesn't matter
                ALUControl = 'b11;
                FlagWrite = 0;
            end

            // LDR
            8'b010_1100_1  : begin
                PCSrc    = 0;
                MemtoReg = 1;
                MemWrite = 0;
                ALUSrc   = 1;
                RegWrite = 1;
                RegSrc   = 'b10;    // msb doesn't matter
                ImmSrc   = 'b01;
                ALUControl = 'b00;  // do an add
                FlagWrite = 0;
            end

            // STR
            8'b010_1100_0  : begin
                PCSrc    = 0;
                MemtoReg = 0; // doesn't matter
                MemWrite = 1;
                ALUSrc   = 1;
                RegWrite = 0;
                RegSrc   = 'b10;    // msb doesn't matter
                ImmSrc   = 'b01;
                ALUControl = 'b00;  // do an add
                FlagWrite = 0;
            end

            // B
            8'b1010_???? : begin
               if((cond == 1110) ||
                  (cond == 0000 && FlagsReg[2]) ||
                  (cond == 0001 && !FlagsReg[2]) ||
                  (cond == 1010 && !FlagsReg[3]) ||
                  (cond == 1100 && !FlagsReg[3] && !FlagsReg[2]) ||
                  (cond == 1101 && (FlagsReg[3] || FlagsReg[2])) ||
                  (cond == 1011 && FlagsReg[3])
                  ) begin
                   PCSrc    = 1;
                   MemtoReg = 0;
                   MemWrite = 0;
                   ALUSrc   = 1;
                   RegWrite = 0;
```

```
                        RegSrc    = 'b01;
                        ImmSrc    = 'b10;
                        ALUControl  = 'b00;  // do an add
                        FlagWrite = 0;
                     end else begin
                        PCSrc     = 0;
                        MemtoReg = 0;
                        MemWrite = 0;
                        ALUSrc    = 0;
                        RegWrite = 0;
                        RegSrc    = 'b00;
                        ImmSrc    = 'b00;    // doesn't matter
                        ALUControl  = 'b00;
                        FlagWrite = 0;
                     end
//                   PCSrc    = 1;
//                      MemtoReg = 0;
//                      MemWrite = 0;
//                      ALUSrc   = 1;
//                      RegWrite = 0;
//                      RegSrc   = 'b01;
//                      ImmSrc   = 'b10;
//                      ALUControl = 'b00;
//                      FlagWrite = 0;
                 end

                 // CMP
                 8'b00?00101 : begin
                      PCSrc     = 0;
                      MemtoReg = 0;
                      MemWrite = 0;
                      ALUSrc    = Instr[25];
                      RegWrite = 1;
                      RegSrc    = 'b00;
                      ImmSrc    = 'b00;
                      ALUControl  = 'b01;  // subtract
                      FlagWrite = 1;
                 end

              default: begin
                      PCSrc     = 0;
                      MemtoReg = 0; // doesn't matter
                      MemWrite = 0;
                      ALUSrc    = 0;
                      RegWrite = 0;
                      RegSrc    = 'b00;
                      ImmSrc    = 'b00;
                      ALUControl  = 'b00;  // do an add
                      FlagWrite = 0;
              end
             endcase
        end


endmodule
```

```
//Aaron Hong (ahong02)
//Stephen Macris (smacris)
//3/29/23
//EE469 Lab1

//This module creates an asynchronous, two read port, one write port, 16x32 register file.

//Inputs: Two 1-bits clk (clock signal), wr_en (write enable), three 4-bits write_addr, read_addr1,
//read_addr2 (write and read addresses), one 32-bit write_data (data to be written at given write address).
//Outputs: Two 32-bits read_data1, read_data2 (data read from given read addresses).
module reg_file (clk, wr_en, write_data, write_addr, read_addr1, read_addr2, read_data1, read_data2);

   input  logic  clk, wr_en;
   input  logic [3:0] write_addr, read_addr1, read_addr2;
   input  logic [31:0] write_data;
   output logic [31:0] read_data1, read_data2;

   logic [15:0][31:0] memory;

   always_ff @(posedge clk) begin
      if(wr_en) begin
         memory[write_addr] <= write_data;
      end
   end

   always_comb begin
      read_data1 <= memory[read_addr1];
      read_data2 <= memory[read_addr2];
   end

endmodule


```

```
//Aaron Hong (ahong02)
//Stephen Macris (smacris)
//3/29/23
//EE469 Lab1

//This module creates a basic arithmetic logic unit capable of addition, subtraction,
//ANDing, and ORing.

//Inputs: Two 32-bits a, b (inputs to be processed), one 2-bit ALUControl (controls which
operation).
//Outputs: One 32-bit Result (result of chosen operation), one 4-bit ALUFlags (flags thrown
depending on the result).
module alu (a, b, ALUControl, Result, ALUFlags);

   input  logic  [31:0] a, b;
   input  logic [1:0] ALUControl;
   output logic [31:0] Result;
   output logic [3:0] ALUFlags;
   logic cout, x, diff_sign;
   logic [31:0] sum, b_mod;

   assign x = ~ALUControl[0] & ~ALUControl[1] & (a[31] == b[31]);
   assign diff_sign = a[31] ^ sum[31];

   //Instantiates a 32-bit full adder to do addition and subtraction operations.
   fulladder32 FA(.A(a), .B(b_mod), .cin(ALUControl[0]), .sum(sum), .cout(cout));

   //Determines what operation to perform depending on ALUControl.
   always_comb begin
      case (ALUControl[0])
         1'b0: b_mod = b;
         1'b1: b_mod = ~b;
      endcase;

      case (ALUControl)
         2'b00: Result = sum;
         2'b01: Result = sum;
         2'b10: Result = a & b;
         2'b11: Result = a | b;
      endcase

      ALUFlags[3] = Result[31];
      ALUFlags[2] = (Result == 32'b0);
      ALUFlags[1] = ~ALUControl[1] & cout;
      ALUFlags[0] = x & diff_sign & ~ALUControl[1];
   end

endmodule
```

```
//Aaron Hong (ahong02)
//Stephen Macris (smacris)
//EE469 Lab1

//This module adds two 32-bit numbers and a given carry-in value.

//Inputs: Two 32-bits A, B (inputs to be added), one 1-bit cin (carry-in value).
//Outputs: One 32-bit sum (sum), one 1-bit cout (carry-out value).
module fulladder32 (A, B, cin, sum, cout);

   input logic [31:0] A;
   input logic [31:0] B;
   input logic cin;

   output logic [31:0] sum;
   output logic cout;

   logic c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15, c16, c17, c18
, c19, c20, c21, c22, c23, c24, c25, c26, c27, c28, c29, c30;

   //Chains together 32 instantiations of a 1-bit adder.
   fulladder FA0(.A(A[0]), .B(B[0]), .cin(cin), .sum(sum[0]), .cout(c0));
   fulladder FA1(.A(A[1]), .B(B[1]), .cin(c0), .sum(sum[1]), .cout(c1));
   fulladder FA2(.A(A[2]), .B(B[2]), .cin(c1), .sum(sum[2]), .cout(c2));
   fulladder FA3(.A(A[3]), .B(B[3]), .cin(c2), .sum(sum[3]), .cout(c3));
   fulladder FA4(.A(A[4]), .B(B[4]), .cin(c3), .sum(sum[4]), .cout(c4));
   fulladder FA5(.A(A[5]), .B(B[5]), .cin(c4), .sum(sum[5]), .cout(c5));
   fulladder FA6(.A(A[6]), .B(B[6]), .cin(c5), .sum(sum[6]), .cout(c6));
   fulladder FA7(.A(A[7]), .B(B[7]), .cin(c6), .sum(sum[7]), .cout(c7));
   fulladder FA8(.A(A[8]), .B(B[8]), .cin(c7), .sum(sum[8]), .cout(c8));
   fulladder FA9(.A(A[9]), .B(B[9]), .cin(c8), .sum(sum[9]), .cout(c9));
   fulladder FA10(.A(A[10]), .B(B[10]), .cin(c9), .sum(sum[10]), .cout(c10));
   fulladder FA11(.A(A[11]), .B(B[11]), .cin(c10), .sum(sum[11]), .cout(c11));
   fulladder FA12(.A(A[12]), .B(B[12]), .cin(c11), .sum(sum[12]), .cout(c12));
   fulladder FA13(.A(A[13]), .B(B[13]), .cin(c12), .sum(sum[13]), .cout(c13));
   fulladder FA14(.A(A[14]), .B(B[14]), .cin(c13), .sum(sum[14]), .cout(c14));
   fulladder FA15(.A(A[15]), .B(B[15]), .cin(c14), .sum(sum[15]), .cout(c15));
   fulladder FA16(.A(A[16]), .B(B[16]), .cin(c15), .sum(sum[16]), .cout(c16));
   fulladder FA17(.A(A[17]), .B(B[17]), .cin(c16), .sum(sum[17]), .cout(c17));
   fulladder FA18(.A(A[18]), .B(B[18]), .cin(c17), .sum(sum[18]), .cout(c18));
   fulladder FA19(.A(A[19]), .B(B[19]), .cin(c18), .sum(sum[19]), .cout(c19));
   fulladder FA20(.A(A[20]), .B(B[20]), .cin(c19), .sum(sum[20]), .cout(c20));
   fulladder FA21(.A(A[21]), .B(B[21]), .cin(c20), .sum(sum[21]), .cout(c21));
   fulladder FA22(.A(A[22]), .B(B[22]), .cin(c21), .sum(sum[22]), .cout(c22));
   fulladder FA23(.A(A[23]), .B(B[23]), .cin(c22), .sum(sum[23]), .cout(c23));
   fulladder FA24(.A(A[24]), .B(B[24]), .cin(c23), .sum(sum[24]), .cout(c24));
   fulladder FA25(.A(A[25]), .B(B[25]), .cin(c24), .sum(sum[25]), .cout(c25));
   fulladder FA26(.A(A[26]), .B(B[26]), .cin(c25), .sum(sum[26]), .cout(c26));
   fulladder FA27(.A(A[27]), .B(B[27]), .cin(c26), .sum(sum[27]), .cout(c27));
   fulladder FA28(.A(A[28]), .B(B[28]), .cin(c27), .sum(sum[28]), .cout(c28));
   fulladder FA29(.A(A[29]), .B(B[29]), .cin(c28), .sum(sum[29]), .cout(c29));
   fulladder FA30(.A(A[30]), .B(B[30]), .cin(c29), .sum(sum[30]), .cout(c30));
   fulladder FA31(.A(A[31]), .B(B[31]), .cin(c30), .sum(sum[31]), .cout(cout));

endmodule

//Tests fulladder32.
module fulladder32_testbench();

   logic [31:0] A, B, sum;
   logic cin, cout;

   fulladder32 dut (A, B, cin, sum, cout);


   integer i;
   initial begin


       A = 32'd15; B = 32'd16; cin = 1'b0; #10;
       A = 32'd15; B = 32'd16; cin = 1'b1; #10;
       A = 32'd15; B = 32'd163; cin = 1'b0; #10;
       A = 32'd15; B = 32'd216; cin = 1'b0; #10;
       A = 32'd15; B = 32'd146; cin = 1'b0; #10;

   end //initial
```

```

endmodule
```

```
//Aaron Hong (ahong02)
//Stephen Macris (smacris)
//EE469 Lab1

//Adds two 1-bit values and a carry-in value.

//Inputs: Three 1-bits A, B (inputs to be added), cin (carry-in value).
//Outputs: Two 1-bits sum (sum), cout (carry-out value).
module fulladder (A, B, cin, sum, cout);

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

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

endmodule

//Tests module fulladder by simulating all 2^3 input bit combinations
module fulladder_testbench();

   logic A, B, cin, sum, cout;

   fulladder dut (A, B, cin, sum, cout);


   integer i;
   initial begin

      for(i=0; i<2**3; i++) begin
       {A, B, cin} = i; #10;
      end //for loop

   end //initial

endmodule

```