```
/*ALU*/
module arm_alu(output reg [31:0] result, output reg N, Z, C, V, input [3:0] opcode, input [31:0]
A, B, input carry);

        always @ (A, B, carry, opcode)
                begin
                        case(opcode)
                                4'b0000: // AND - Logical AND
                                        begin
                                                result = A & B;
                                        end
                                4'b0001: // EOR - Logical EOR
                                        begin
                                                result = A ^ B;
                                                C = carry;
                                        end
                                4'b0010: // SUB - Substract
                                        begin
                                                {C,result} = A - B;
                                                C = ~C;
                                                update_SUB_V_Flag();
                                        end
                                4'b0011: //RSB - Reverse Subtract
                                        begin
                                                {C,result} = B - A;
                                                C = ~C;
                                                update_RSB_V_Flag();
                                        end
                                4'b0100: // ADD - Addition
                                        begin
                                                {C,result} = A + B;
                                                update_ADD_V_Flag();
                                        end
                                4'b0101: // ADC - Addition with Carry
                                        begin
                                                {C,result} = A + B + C;
                                                update_ADD_V_Flag();
                                        end
                                4'b0110: // SBC - Subtract with Carry
                                        begin
                                                C = ~C;
                                                {C,result} = A - B - C;
                                                C = ~C;
                                                update_SUB_V_Flag();
```

```
                    end
                4'b0111: //RSC - Reverse Subtract with Carry
                    begin
                        C = ~C;
                        {C,result} = B - A - C;
                        C = ~C;
                        update_RSB_V_Flag();
                    end
                4'b1000: //TST - Test (Logical AND, with no results written)
                    begin
                        result = A & B;
                        C = carry;
                    end
                4'b1001: //TEQ - Test Equivalence (Logical EOR, with no results
written)
                    begin
                        result = A ^ B;
                        C = carry;
                    end
                4'b1010: //CMP - Compare (SUB, with no results written)
                    begin
                        {C,result} = A - B;
                        C = ~C;
                        update_SUB_V_Flag();
                    end
                4'b1011: //CMN - Compare Negated (NOT ADD, with no results
written)
                    begin
                        {C,result} = A + B;
                        update_ADD_V_Flag();
                    end
                4'b1100: // ORR - Logical OR
                    begin
                        result = A | B;
                        C = carry;
                    end
                4'b1101: //MOV - Move
                    begin
                        result = B;
                        C = carry;
                    end
                4'b1110: //BIC - Bit Clear
                    begin
                        result = A & ~B;
                        C = carry;
                    end
                4'b1111: //MVN - Move Not
                    begin
                        result = ~B;
                        C = carry;
                    end
            endcase

            N = result[31]; // Set Negative (N) Flag to the result's most significant bit
            if(result == 0) Z = 1; // Set Zero (Z) Flag
            else Z = 0;
        end

        task update_ADD_V_Flag; // Check for Overflow when adding
        begin
            if((A[31] == 0 && B[31] == 0 && result[31] == 1) || (A[31] == 1 && B[31]
== 1 && result[31] == 0))
                V=1;
            else
                V=0;
        end
        endtask

        task update_SUB_V_Flag; // Check for overflow when substracting
        begin
            if((A[31] == 0 && B[31] == 1 && result[31] == 1) || (A[31] == 1 && B[31]
== 0 && result[31] == 0))
                V=1;
            else
                V=0;
        end
        endtask

        task update_RSB_V_Flag; // Check for overflow when reverse substracting
        begin
            if((B[31] == 0 && A[31] == 1 && result[31] == 1) || (B[31] == 1 && A[31]
== 0 && result[31] == 0))
                V=1;
            else
                V=0;
        end
        endtask
```

```
endmodule//Branch Extender

module branch_extender(output reg [31:0] final_offset, input [23:0] offset);

reg [25:0] offset1;

always @ (*)

begin

  offset1=offset;

  offset1=offset1<<2;

  final_offset= $signed(offset1);

end

endmodule
//Control Unit

module control_unit(output reg [44:0] cu_out, input clk, reset, mfc, input [31:0] cu_in, input
[3:0] flags);

//Wires connected to other components

wire im_out, Mh_out, enc_en, le, inv_out;
wire [1:0] next_state_out;
wire [5:0] enc_out, inc_out, Mj_out, Mk_out, incR_out;
wire [54:0] rom_out;
reg [31:0] tempIR;
parameter ONE = 6'b000001;
parameter ZERO = 6'b000000;


wire [54:0] tempPipe;
reg [5:0] nxSt;

//Components Created

PipelineRegister pipe_reg (tempPipe, rom_out, clk);

Input_Manager input_man (im_out, flags, cu_in [31:28]);
```

```
mux_2to1_1bit MuxH (Mh_out, tempPipe[15], im_out, mfc);

mux_4to1_6bits MuxJ (Mj_out, next_state_out, enc_out, ONE, tempPipe[8:3], incR_out);

mux_2to1_6bits MuxK (Mk_out, reset, ZERO, Mj_out);

encoder enc (enc_out, tempIR, clk);

Microstore_ROM rom (rom_out, Mk_out);

incrementer inc (inc_out, Mk_out);

IncrementerRegister incReg (incR_out, inc_out, clk);

Nxt_St_Sel next_state (next_state_out [0], next_state_out [1], inv_out, tempPipe[2:0]);

Inverter inverter (inv_out, Mh_out, tempPipe[54]);



always @ (*)
begin
        cu_out <= tempPipe[53:9];
        tempIR <= cu_in;
end

endmodule
//Datapath

module DataPath(output reg mfc, output reg [31:0] IR_Out, StaReg_Out, input [3:0] cu_opcode,
write_cu, readA_cu, readB_cu, input [1:0] Ma, dataType, imme_SEL, Mg,
                                        input RFen, Mb, Mc, Md, Me, Mf, Mo, Ms, Ml, MAREn, MDREn,
IREn, SHFen, SignExtEn, SignExtEn2, r_w, MemEN, dwp, mfa, clk, clr, StatusRegEn_cu);

//Wires connected to other components

wire [31:0] RFOut_A, RFOut_B, BranchExt_Out, SignExt_Out, Ma_Out, Mb_Out, Mc_Out,
Md_Out, Me_Out,
        SignExt_Out2, ALU_Out, Shf_Out, MDR_Out, MAR_Out, MemOut, Flags;

wire [3:0] write, readA, readB, opcode_in, Ml_out;
reg carry;
wire StatusRegEn;
```

```
//assign write = wireIR[15:12];
assign write = Ml_out;

//Temp variables

wire [31:0] temp4, tempStaReg;
wire [31:0] wireIR;
wire tempMfc;


assign temp4 = 32'h00000004;

always @ (*)
begin
	IR_Out = wireIR;
	StaReg_Out = tempStaReg;
	mfc = tempMfc;
	carry = StaReg.Y[2];
end

//Components created

register_file regf(RFOut_A, RFOut_B, ALU_Out, readA, readB, write, RFen, clr, clk);

mux_4to1 MuxA (Ma_Out, Ma, RFOut_B, MDR_Out, Mb_Out, temp4);

mux_2to1 MuxB (Mb_Out, Mb, BranchExt_Out, SignExt_Out);

mux_2to1 MuxC (Mc_Out, Mc, ALU_Out, MemOut);

mux_2to1 MuxD (Md_Out, Md, Me_Out, RFOut_A);

mux_2to1 MuxE (Me_Out, Me, wireIR, wireIR);

mux_2to1_4bits MuxF (readA, Mf, readA_cu, wireIR[19:16]);

mux_4to1_4bits MuxG (readB, Mg, readB_cu, wireIR[3:0], wireIR[15:12], wireIR[19:16]);

mux_2to1_4bits MuxL (Ml_out, Ml, write_cu, wireIR[15:12]);

mux_2to1_4bits MuxO (opcode_in, Mo, cu_opcode, wireIR[24:21]);

mux_2to1_1bit MuxS (StatusRegEn, Ms, StatusRegEn_cu, ~wireIR[20]);
```

```
arm_alu alu (ALU_Out, Flags [0], Flags [1], Flags[2], Flags [3], opcode_in, RFOut_A, Shf_Out,
carry);

branch_extender branchExt (BranchExt_Out, wireIR[23:0]);

Imme_Sign_Extension signExt (SignExt_Out, wireIR, SignExtEn, imme_SEL);

Sign_Extension2 signExt2 (SignExt_Out2, Mc_Out, dataType, SignExtEn2);

Shifter shift (Shf_Out, Flags [2], Ma_Out, wireIR[6:5], Md_Out, SHFen);

register_32_bits InsReg (wireIR, MemOut, IREn, clr, clk);

register_32_bits StaReg (tempStaReg, Flags, StatusRegEn, clr, clk);

register_32_bits Mar (MAR_Out, ALU_Out, MAREn, clr, clk);

register_32_bits Mdr (MDR_Out, SignExt_Out2, MDREn, clr, clk);

Ram ram (MDR_Out, MemEN, r_w, mfa, dataType, dwp, MAR_Out [7:0], tempMfc, MemOut);

endmodule
//Decoder 16 to 1

module decoder (output reg [15:0] out, input [3:0] in, input enable);

always @ (in, enable)

begin
case (in)
4'b0000: out = 16'b0000000000000001;
4'b0001: out = 16'b0000000000000010;
4'b0010: out = 16'b0000000000000100;
4'b0011: out = 16'b0000000000001000;
4'b0100: out = 16'b0000000000010000;
4'b0101: out = 16'b0000000000100000;
4'b0110: out = 16'b0000000001000000;
4'b0111: out = 16'b0000000010000000;
4'b1000: out = 16'b0000000100000000;
4'b1001: out = 16'b0000001000000000;
4'b1010: out = 16'b0000010000000000;
4'b1011: out = 16'b0000100000000000;
4'b1100: out = 16'b0001000000000000;
4'b1101: out = 16'b0010000000000000;
```

```
4'b1110: out = 16'b0100000000000000;
4'b1111: out = 16'b1000000000000000;
endcase
   if(!enable)
    begin
        out = 16'h0000;
    end
end

endmodule
//Encoder

module encoder (output reg [5:0] out, input [31:0] in, input clk);

reg [31:0] temp;

initial begin
    assign temp = in;
end

always @ (in)
begin
    case (temp[27:25])

        3'b000:

            begin

                if (temp[4] == 1'b0)

                    begin

                        if (temp[11:7] == 5'b00000)

                            out = 6'b000101; //5 = Register Shifter Operand

                        else

                        out = 6'b000111; //7 = Shift by Immediate Shifter Operand

                    end
                if (temp[4] == 1'b1)

                    begin

                        if (temp[24] == 1'b0)

                            begin

                                if(temp[20] == 1'b1)

                                    begin

                                        out = 6'b101001; //41
= Immediate Post-Indexed Load MISCELLANEOUS

                                    end

                                else

                                        out = 6'b100101; //37
= Immediate Post-Indexed Store MISCELLANEOUS
                            end

                        else

                            begin

                                case (temp[21])

                                    1'b0:

                                        begin

                                            if(temp[20] ==
1'b1)

                                                begin


    out = 6'b100111; //39 = Immediate Offset Load MISCELLANEOUS

                                                end

                                            else
    out = 6'b100011; //35 = Immediate Offset Store MISCELLANEOUS
                                        end
                                    1'b1:
                                        begin
                                            if(temp[20] ==
1'b1)
                                                begin
    out = 6'b101000; //40 = Immediate Pre-Indexed Load MISCELLANEOUS
                                                end
                                            else
    out = 6'b100100; //36 = Immediate Pre-Indexed Store MISCELLANEOUS
                                        end
                                endcase
                            end
                    end
            end
        3'b001:
            out = 6'b000110; //6 = 32-bit Immediate Shifter Operand
        3'b010:
            begin
                if (temp[24] == 1'b0)
```

```
                    begin
                        if(temp[20] == 1'b1)
                            begin
                                out = 6'b011000; //24 = Immediate Post-Indexed Load
                            end
                        else
                                out = 6'b010000; //16 = Immediate Post-Indexed Store
                    end
                else
                    begin
                        case (temp[21])
                            1'b0:
                                begin
                                    if(temp[20] == 1'b1)
                                        begin
                                            out = 6'b010100; //20 = Immediate Offset Load
                                        end
                                    else
                                            out = 6'b001100; //12 = Immediate Offset Store
                                end
                            1'b1:
                                begin
                                    if(temp[20] == 1'b1)
                                        begin
                                            out =
6'b010110; //22 = Immediate Pre-Indexed Load
                                        end
                                    else
                                            out =
6'b001110; //14 = Immediate Pre-Indexed Store
                                end
                        endcase
                    end
            end
        3'b011:
            begin
                if (temp[24] == 1'b0)
                    begin
                        if(temp[20] == 1'b1)
                            begin
                                out = 6'b011010; //26 = Register
Post-Indexed Load
                            end
                        else
```

```
                                out = 6'b010010; //18 = Register
Post-Indexed Store
                    end
                else
                    begin
                        case (temp[21])
                            1'b0:
                                begin
                                    if(temp[20] == 1'b1)
                                        begin
                                            out =
6'b010101; //21 = Register Offset Load
                                        end
                                    else
                                            out =
6'b001101; //13 = Register Offset Store
                                end
                            1'b1:
                                begin
                                    if(temp[20] == 1'b1)
                                        begin
                                            out =
6'b010111; //23 = Register Pre-Indexed Load
                                        end
```

```
                                                                                    else

                                                                                            out =
6'b001111; //15 = Register Pre-Indexed Store
                                                                                end

                                                                endcase

                                                        end

                                        end

                                3'b101:

                                        begin

                                                if (temp[24] == 1'b0)

                                                        begin

                                                                out = 6'b101011; //43 = Branch

                                                        end

                                                else

                                                        begin

                                                                out     = 6'b101100; //44 = Branch with Link

                                                        end

                                        end

                        endcase

                        begin
                        if (temp == 31'h00000000)
                                out=6'b000000;
                        end
end

endmodule
// Sign Extension
```

```
module Imme_Sign_Extension (output reg [31:0] Y, input [31:0] in, input enable, input [1:0]
immeSEL);

always @ (*)

begin
        if (!enable)
        begin
                case (immeSEL)
                        2'b00:
                        begin
                                Y[7:0] = in[7:0];
                                Y[31:8] = {24{in[7]}};
                        end
                        2'b01:
                        begin
                                Y[11:0] = in[11:0];
                                Y[31:12] = {20{in[7]}};
                        end
                        2'b10:
                        begin
                                Y[23:0] = in[23:0];
                                Y[31:24] = {8{in[7]}};
                        end
                        2'b11:
                        begin
                                Y[7:0] = {in[11:8], in[3:0]};
                                Y[31:8] = {24{in[7]}};
                        end
                endcase
        end

        else
        begin
                case (immeSEL)
                        2'b00:
                        begin
                                Y[7:0] = in[7:0];
                                Y[31:8] = 24'b000000000000000000000000;
                        end
                        2'b01:
                        begin
                                Y[11:0] = in[11:0];
                Y[31:12] = 20'b00000000000000000000;
            end
            2'b10:
            begin
                Y[23:0] = in[23:0];
                Y[31:24] = 8'b00000000;
            end
            2'b11:
            begin
                Y[7:0] = {in[11:8], in[3:0]};
                Y[31:8] = 24'b000000000000000000000000;
            end
        endcase
    end
end

endmodule// Incrementer
// Provides a serial number that auto increments on each time tick.
module incrementer (output reg [5:0] inc_out, input [5:0] inc_in);

// Set count to 0 by default
initial inc_out = 6'b000000;

always @ (*)
    inc_out = inc_in + 1'b1; // bit increment
endmodule
// Incrementer Register
// Takes care of saving temporarily the status of the incrementer.
module IncrementerRegister (output reg [5:0] out, input [5:0] in, input clk);

always @(posedge clk)
   out <= in;
endmodule
// Input Signal Manager

// Flags[0] = Negative Flag
// Flags[1] = Zero Flag
// Flags[2] = Carry Flag
// Flags[3] = Overflow Flag

module Input_Manager (output reg out, input [3:0] Flags, cond_Code);


always @ (*)
```

```
begin

    case (cond_Code)

    4'b0000: // EQ
    begin
        out = Flags[1];
    end

    4'b0001: // NE
    begin
        out = ~Flags[1];
    end

    4'b0010: // CS/HS
    begin
        out = Flags[2];
    end

    4'b0011: // CC/LO
    begin
        out = ~Flags[2];
    end

    4'b0100: // MI
    begin
        out = Flags[0];
    end

    4'b0101: // PL
    begin
        out = ~Flags[0];
    end

    4'b0110: // VS
    begin
        out = Flags[3];
    end

    4'b0111: // VC
    begin
        out = ~Flags[3];
    end
```

```
        4'b1000: // HI
        begin
                out = Flags[2] & ~Flags[1];
        end

        4'b1001: // LS
        begin
                out = ~Flags[2] | Flags[1];
        end

        4'b1010: // GE
        begin
                out = ~(Flags[0] ^ Flags[3]);
        end

        4'b1011: // LT
        begin
                out = Flags[0] ^ Flags[3];
        end

        4'b1100: // GT
        begin
                out = ~(Flags[0] ^ Flags[3]) & (~Flags[1]);
        end

        4'b1101: // LE
        begin
                out = (Flags[0] ^ Flags[3]) | (Flags[1]);
        end

        4'b1110: // AL
        begin
                out = 1'b1;
        end

        // Spare
        4'b1111:
        begin
        end

        endcase
end

endmodule// Inverter
```

```
// Takes input and negates it, depending on control signal INV
module Inverter (output reg out, input in, INV);

always @(*)
begin
        if(INV == 1'b0)
                out = in;
        else
                out = ~in;
end

endmodule//Multiplexer 1 bit 2 to 1

module mux_2to1_1bit (output reg Y, input S,
                 input D0, input D1);

always @ (S,D0,D1)

case (S)

1'b0:   Y = D0;
1'b1:   Y = D1;

endcase

endmodule
//Multiplexer 32 bits 16 to 1

module mux_16to1 (output reg [31:0] Y, input [3:0] S, input [31:0] D0, D1, D2, D3, D4, D5, D6,
D7, D8, D9, D10, D11, D12, D13, D14, D15);

always @ (S,D0,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15)

case (S)

4'b0000: Y = D0;
4'b0001: Y = D1;
4'b0010: Y = D2;
4'b0011: Y = D3;
4'b0100: Y = D4;
4'b0101: Y = D5;
4'b0110: Y = D6;
4'b0111: Y = D7;
4'b1000: Y = D8;
```

```
4'b1001: Y = D9;
4'b1010: Y = D10;
4'b1011: Y = D11;
4'b1100: Y = D12;
4'b1101: Y = D13;
4'b1110: Y = D14;
4'b1111: Y = D15;

endcase

endmodule
//Multiplexer 32 bits 4 to 1

module mux_4to1 (output reg [31:0] Y, input [1:0] S,
                 input [31:0] D0, input [31:0] D1, input [31:0] D2, input [31:0] D3);

always @ (S,D0,D1,D2,D3)

case (S)

2'b00:   Y = D0;
2'b01:   Y = D1;
2'b10:   Y = D2;
2'b11:   Y = D3;

endcase

endmodule

//Multiplexer 32 bits 2 to 1

module mux_2to1 (output reg [31:0] Y, input S,
                 input [31:0] D0, input [31:0] D1);

always @ (S,D0,D1)

case (S)

1'b0:   Y = D0;
1'b1:   Y = D1;

endcase

endmodule
```

```
//Multiplexer 4 bits 2 to 1
module mux_2to1_4bits (output reg [3:0] Y, input S,
                    input [3:0] D0, input [3:0] D1);

          always @ (S,D0,D1)

                    case (S)
                    1'b0:   Y = D0;
                    1'b1:   Y = D1;
                    endcase

endmodule


//Multiplexer 4 bits 4 to 1
module mux_4to1_4bits (output reg [3:0] Y, input [1:0] S,
                    input [3:0] D0, input [3:0] D1, input [3:0] D2, input [3:0] D3);

          always @ (S,D0,D1)

                    case (S)
                    2'b00:   Y = D0;
                    2'b01:   Y = D1;
                    2'b10:   Y = D2;
                    2'b11:   Y = D3;
                    endcase

endmodule//Multiplexer 5 bits 4 to 1

module mux_4to1_5bits (output reg [4:0] Y, input [1:0] S,
                    input [4:0] D0, input [4:0] D1, input [4:0] D2, input [4:0] D3);

always @ (S,D0,D1,D2,D3)

case (S)

2'b00:   Y = D0;
2'b01:   Y = D1;
2'b10:   Y = D2;
2'b11:   Y = D3;

endcase

endmodule
```

```
//Multiplexer 5 bits 2 to 1

module mux_2to1_5bits (output reg [4:0] Y, input S,
                  input [4:0] D0, input [4:0] D1);

always @ (S,D0,D1)

case (S)

1'b0:   Y = D0;
1'b1:   Y = D1;

endcase

endmodule
//Multiplexer 6 bits 4 to 1

module mux_4to1_6bits (output reg [5:0] Y, input [1:0] S,
                  input [5:0] D0, input [5:0] D1, input [5:0] D2, input [5:0] D3);

always @ (S,D0,D1,D2,D3)

case (S)

2'b00:   Y = D0;
2'b01:   Y = D1;
2'b10:   Y = D2;
2'b11:   Y = D3;

endcase

endmodule

//Multiplexer 6 bits 2 to 1

module mux_2to1_6bits (output reg [5:0] Y, input S,
                  input [5:0] D0, input [5:0] D1);

always @ (S,D0,D1)

case (S)

1'b0:   Y = D0;
```

```
1'b1:   Y = D1;

endcase

endmodule
//MICROPROCESDOR ARM

module microprocessor();

wire mfc;
wire [31:0] IR_Out, StaReg_Out;
wire [44:0] cu_out;
reg clk, clr;

//Module Instantiation
DataPath datapath (.mfc(mfc), .IR_Out(IR_Out), .StaReg_Out(StaReg_Out),
.cu_opcode(cu_out[30:27]), .write_cu(cu_out[42:39]), .readA_cu(cu_out[38:35]),
.readB_cu(cu_out[34:31]),
            .Ma(cu_out[26:25]), .dataType(cu_out[9:8]), .imme_SEL(cu_out[1:0]),
.RFen(cu_out[44]), .Mb(cu_out[20]), .Mc(cu_out[19]), .Md(cu_out[18]), .Me(cu_out[17]),
.Mf(cu_out[16]),
            .Mg(cu_out[15:14]), .Mo(cu_out[12]), .Ms(cu_out[22]), .Ml(cu_out[13]),
.MAREn(cu_out[4]), .MDREn(cu_out[3]), .IREn(cu_out[5]), .SHFen(cu_out[24]),
.SignExtEn(cu_out[21]),
            .SignExtEn2(cu_out[2]), .r_w(cu_out[11]), .MemEN(cu_out[10]), .dwp(cu_out[7]),
.mfa(cu_out[6]), .clk(clk), .clr(clr), .StatusRegEn_cu(cu_out[23]));

control_unit cu (.cu_out(cu_out), .clk(clk), .reset(clr), .mfc(mfc), .cu_in(IR_Out),
.flags(StaReg_Out [3:0]));

//Memory Variables
reg [31:0] dat[0:155];
parameter WORD = 2'b10;
parameter WRITE = 1'b1;
parameter ENABLE = 1'b0;
reg [31:0] temp_data_in;
reg [7:0] temp_addr;

initial #5000 $finish;

initial $readmemb("testcode_arm1.txt", dat);

reg [8:0] i; // loop index
```

```
initial begin

//--------------------------------------------------------------------
//                                                  FILL RAM MEMORY
//--------------------------------------------------------------------
        for(i=9'h000;i<9'h030;i=i+9'h004)
        begin
                temp_data_in[31:24] <= dat[i[7:0]];
                temp_data_in[23:16] <= dat[i[7:0]+1];
                temp_data_in[15:8] <= dat[i[7:0]+2];
                temp_data_in[7:0] <= dat[i[7:0]+3]; //setting up data
                #5;
                datapath.ram.ram[i[7:0]] <= temp_data_in[31:24];
                datapath.ram.ram[i[7:0]+1]<= temp_data_in[23:16];
                datapath.ram.ram[i[7:0]+2] <= temp_data_in[15:8];
                datapath.ram.ram[i[7:0]+3] <= temp_data_in[7:0];
                #5;
         end

        for(i=9'h000;i<9'h030;i=i+9'h004)
        begin
                $write ("WORD DATA at %d: %h", i, datapath.ram.ram[i[7:0]]);
                $write ("%h", datapath.ram.ram[i[7:0]+1]);
                $write ("%h", datapath.ram.ram[i[7:0]+2] );
                $display ("%h", datapath.ram.ram[i[7:0]+3]);
        end

        clr = 1'b0;
        #5 clk = 1'b0;
        #5 clk = 1'b1;
        #5 clk = 1'b0;
        #5 clk = 1'b1;
        clr = 1'b1;
        repeat (500) #5 clk = ~clk;

        #10 $display("\nTesting content of Registers:\nR0 = %h\nR1 = %h\nR2 = %h\nR3 =
%h\nR4 = %h\nR5 = %h\nR6 = %h\nR7 = %h\nR8 = %h\nR9 = %h\nR10 = %h\nR11 = %h\nR12 =
%h\nR13 = %h\nR14 = %h\nR15 = %h\n", datapath.regf.register0.Y, datapath.regf.register1.Y,
datapath.regf.register2.Y, datapath.regf.register3.Y, datapath.regf.register4.Y,
datapath.regf.register5.Y, datapath.regf.register6.Y, datapath.regf.register7.Y,
datapath.regf.register8.Y, datapath.regf.register9.Y, datapath.regf.register10.Y,
datapath.regf.register11.Y, datapath.regf.register12.Y, datapath.regf.register13.Y,
datapath.regf.register14.Y, datapath.regf.register15.Y);
```

```
	for(i=9'h000;i<9'h030;i=i+9'h004)
	begin
		$write ("WORD DATA at %d: %h", i, datapath.ram.ram[i[7:0]]);
		$write ("%h", datapath.ram.ram[i[7:0]+1]);
		$write ("%h", datapath.ram.ram[i[7:0]+2] );
		$display ("%h", datapath.ram.ram[i[7:0]+3]);
	end

end

initial begin

	/*$monitor("R10: %h\nR5: %h\nR3: %h\nR2: %h\nR1: %h\nR0: %h\nR15: %h\nRead A:
%b\nRead B: %b\nR15 Enable: %d\nRF Enable: %d\nIR: %h\nIR Enable: %d\nMDR: %h\nMAR:
%h\nMar enable: %b\nRAM out: %h\nMFC: %b\nPipeToNext: %b\nCond S: %b\nInv Out:
%b\nFlags: %b\nCond Code: %b\nIM Out: %b\nM0: %b\nM1: %b\nEncode OUT: %d\nData A:
%h\nData B: %h\nALU out: %h\nOpcode: %b\nMJ Out: %d\nMK Sel: %b\nMK Out: %d\nBranch
In: %h\nBranch Out: %h\nMuxA Sel: %b\nMuxA Out: %h\nMuxB Out: %h\nMuxB Sel:
%b\nTime: %d\nNext State: %d\n--------------------------------------", datapath.regf.register10.Y,
datapath.regf.register5.Y, datapath.regf.register3.Y, datapath.regf.register2.Y,
datapath.regf.register1.Y, datapath.regf.register0.Y, datapath.regf.register15.Y,
datapath.regf.readA, datapath.regf.readB, datapath.regf.register15.LE, datapath.regf.enable,
datapath.InsReg.Y, datapath.InsReg.LE, datapath.Mdr.Y, datapath.Mar.Y, datapath.Mar.LE,
datapath.ram.data_out, datapath.ram.mfc, cu.next_state.pipeline, cu.next_state.cond_S,
cu.inverter.out, cu.input_man.Flags, cu.input_man.cond_Code, cu.input_man.out,
cu.next_state.M0, cu.next_state.M1, cu.enc.enc_out, datapath.regf.outA, datapath.alu.dataB,
datapath.alu.result, datapath.alu.op_code, cu.MuxJ.Y, cu.MuxK.S, cu.MuxK.Y,
datapath.branchExt.offset, datapath.branchExt.final_offset, datapath.MuxA.S,
datapath.MuxA.Y, datapath.MuxB.Y, datapath.MuxB.S, $time, cu.rom.index);*/

	$monitor("MAR: %0d\n\n--------------\n\nState: %d", datapath.Mar.Y, cu.rom.index);
	//$monitor("MAR: %0d\n Flags (ALU): %b %b %b %b\n Flags (SR): %b\n Flags (CU):
%b\n--------------\n", datapath.Mar.Y, datapath.alu.N_flag, datapath.alu.Z_flag,
datapath.alu.C_flag, datapath.alu.V_flag, datapath.StaReg.Y[3:0], cu.flags);

end

endmodule
// MICROSTORE (ROM)
module Microstore_ROM (output reg[54:0] out, input [5:0] index);

always @(index)
begin
	case(index)
```

```
                    6'b000000:  out =
55'b0100000000000000110100110101000001001100011110000001010;
                    6'b000001:  out =
55'b0110000111111111101001101010010000011000101100000010011;
                    6'b000010:  out =
55'b0011111111111110100111101010000000001000111100000010011;
                    6'b000011:  out =
55'b1111111111111111101111101010000000001001010100000011101;
                    6'b000100:  out =
55'b0111111111111111101111101010000000001000111100000001000;
                    6'b000101:  out =
55'b0010000000000000000000101101001011101100011110000001001;
                    6'b000110:  out =
55'b0010000000000000000001010111100101110110001111000001001;
                    6'b000111:  out =
55'b0010000000000000000001010111001101110110001111000001001;
                    6'b001000:  out =
55'b1110000000000000110101110101001011000000111010100100 0101;
                    6'b001001:  out =
55'b0010000000000000110101110100001011001100011110100000100 1;
                    6'b001010:  out =
55'b0110000000000000110100110100001101001100011010100101101 1;
                    6'b001011:  out =
55'b1110000000000000110100110100001101010001111110100101111 1;
                    6'b001100:  out =
55'b0110000000000000010010110010001011001100010110100101001 0;
                    6'b001101:  out =
55'b0110000000000000010000110100001011001100010110100101001 0;
                    6'b001110:  out =
55'b0010000000000000010010110100001011001100010110100101001 0;
                    6'b001111:  out =
55'b0010000000000000010000110100001011001100010110100101001 0;
                    6'b010000:  out =
55'b0110000000000000110100110100001011001100010110101000101 1;
                    6'b010001:  out =
55'b0010000000000000010010110100001011001100011110100101001 0;
                    6'b010010:  out =
55'b0110000000000000110100110100001011001100010110101001101 1;
                    6'b010011:  out =
55'b0010000000000000010000110100001011001100011110100101001 0;
                    6'b010100:  out =
55'b0110000000000000010010110010001011001100010110100100001 0;
                    6'b010101:  out =
55'b0110000000000000010000110100001011001100010110100100001 0;
```

```
                    6'b010110:  out =
55'b0010000000000000100101101000010110011000101101001000010;
                    6'b010111:  out =
55'b0010000000000000100001101000010110011000101101001000010;
                    6'b011000:  out =
55'b0110000000000001101001101000010110011000101101011001011;
                    6'b011001:  out =
55'b0010000000000000100101101000010110011000111101001000010;
                    6'b011010:  out =
55'b0110000000000001101001101000010110011000101101011011011;
                    6'b011011:  out =
55'b0010000000000000100001101000010110011000111101001000010;
                    6'b011100:  out =
55'b1110000000000001101001101010010110000001110011011100101;
                    6'b011101:  out =
55'b0010000000000001101011101000010110011000111111000001001;
                    6'b011110:  out =
55'b0110000000000001101001101000011010011000110011011111011;
                    6'b011111:  out =
55'b1110000000000001101001101000011010100000111111011111111;
                    6'b100000:  out =
55'b0011111111111110100101101000000000011000111110000001001;
                    6'b100001:  out =
55'b0011110111111111101001101000000000011000111110100010011;
                    6'b100010:  out =
55'b0011111111111110100101101100000000011000111110000001001;
                    6'b100011:  out =
55'b0110000000000000100101100000010110011000101111011110010;
                    6'b100100:  out =
55'b0010000000000000100101101000010110011000101111011110010;
                    6'b100101:  out =
55'b0110000000000001101001101000010110011000101111100110011;
                    6'b100110:  out =
55'b0010000000000000100101101000010110011000111111011110010;
                    6'b100111:  out =
55'b0110000000000000100101100000010110011000101111011100010;
                    6'b101000:  out =
55'b0010000000000000100101101000010110011000101111011100010;
                    6'b101001:  out =
55'b0110000000000001101001101000010110011000101111101001011;
                    6'b101010:  out =
55'b0010000000000000100101101000010110011000111111011100010;
                    6'b101011:  out =
55'b0110000000000001101001101000010110011000111111100000111;
```

```
                    6'b101100:  out = 
55'b0110000000000001101001101000010110011000111111100001111;
          endcase
end

endmodule
// NEXT STATE ADDRESS SELECTOR
module Nxt_St_Sel (output reg m0, output reg m1, input condS, input [2:0] pipeline);

initial begin
          m0 = 1'b1;
          m1 = 1'b0;
end

always @ (*)
case (pipeline)

          3'b000: // Encoder
          begin
                    m0 = 1'b0;
                    m1 = 1'b0;
          end

          3'b001: // Fetch
          begin
                    m0 = 1'b1;
                    m1 = 1'b0;
          end

          3'b010: // Pipeline
          begin
                    m0 = 1'b0;
                    m1 = 1'b1;
          end

          3'b011: // Incrementer
          begin
                    m0 = 1'b1;
                    m1 = 1'b1;
          end

          3'b100:
          begin
                    if(condS == 1'b1)
```

```
    begin
        m0 = 1'b0;
        m1 = 1'b1;
    end
    else
    begin
        m0 = 1'b0;
        m1 = 1'b0;
    end
end

3'b101:
begin
    if(condS == 1'b1)
    begin
        m0 = 1'b0;
        m1 = 1'b1;
    end
    else
    begin
        m0 = 1'b1;
        m1 = 1'b1;
    end
end

3'b110:
begin
    if(condS == 1'b1)
    begin
        m0 = 1'b0;
        m1 = 1'b0;
    end
    else
    begin
        m0 = 1'b1;
        m1 = 1'b1;
    end
end

3'b111:
begin
    if(condS == 1'b1)
    begin
        m0 = 1'b0;
```

```
                        m1 = 1'b1;
                end
                else
                begin
                        m0 = 1'b1;
                        m1 = 1'b0;
                end
        end
endcase

endmodule// PIPELINE REGISTER
module PipelineRegister (output reg [54:0] out, input [54:0] pipeline_in, input clk);

always @(posedge clk)
   out <= pipeline_in;
endmodule
module Ram (input [31:0] data_in, input enable, input r_w, input Ready, input [1:0] dtype,
input _dwp1, input [7:0] addr,
                        output reg Clear, output reg [31:0] data_out);

reg [7:0] ram[0:255]; // RAM 256 bytes
reg [7:0] t_addr; // Temporary Address

// Constants
  parameter WRITE = 1'b1; // One for Write operation
  parameter READ = 1'b0; // Zero for Read operation
  parameter ENABLE = 1'b0; // Enable is equal to one, assuming Acctive low
// Costant Data Types
  parameter BYTE = 2'b00; //8 bit
  parameter HALF = 2'b01; // 16 bit
  parameter WORD = 2'b10; // 32 bit
  parameter DWORD = 2'b11;          // 64 bit

always@(Ready) //Data ready signal
begin
        if(enable == 0) begin
        case(dtype) //Data type
                BYTE:begin
                        if(r_w==WRITE)begin
                                ram[addr] <= data_in[7:0]; //Writing to ram memory first 8 bits in
data in.
                        end
                data_out <= {{24{1'b0}}, ram[addr]}; //Reading first 8 bits in ram
memory to data out and setting the rest to zero
        end
        HALF:begin
            t_addr[7:1] <= addr[7:1]; // Halfword is 16 bits so it neads 2 columns, set
temporary address begingin at 1
            t_addr[0] <= 1'b0; // setting first bit in address to zero so that it is never
odd
            #5;
            if(r_w==WRITE)begin
                ram[t_addr] <= data_in[15:8]; //Most significant bits in smallest
address, Big Endian.
                ram[t_addr+1] <= data_in[7:0];
                #3;
            end

                data_out <= {{16{1'b0}}, ram[t_addr], ram[t_addr+1] }; // Reading
biggest address first then smallest followed by zeros
            #3;
        end
        WORD:begin
            t_addr[7:2] <= addr[7:2];
            t_addr[1:0] <= 2'b00;
            #1;

            if(r_w==WRITE)begin
                ram[t_addr] <= data_in[31:24];
                ram[t_addr+1] <= data_in[23:16];
                ram[t_addr+2] <= data_in[15:8];
                ram[t_addr+3] <= data_in[7:0];
                #1;
            end

            data_out[31:24] <= ram[t_addr];
            data_out[23:16] <= ram[t_addr+1];
            data_out[15:8] <= ram[t_addr+2];
            data_out[7:0] <= ram[t_addr+3];
            #1;
        end
        DWORD:begin
            t_addr[7:3] <= addr[7:3];
            t_addr[2:0] <= 3'b000;
            #5;
            if(r_w==WRITE)
                begin
                    if(_dwp1)
                        begin
                            ram[t_addr] <= data_in[31:24];
                            ram[t_addr+1] <= data_in[23:16];
                            ram[t_addr+2] <= data_in[15:8];
                            ram[t_addr+3] <= data_in[7:0];
                        end
                    else
                        begin
                            ram[t_addr+4] <= data_in[31:24];
                            ram[t_addr+5] <= data_in[23:16];
                            ram[t_addr+6] <= data_in[15:8];
                            ram[t_addr+7] <= data_in[7:0];
                        end
                #3;
            end

            if(_dwp1)
                begin
                    data_out[31:24] <= ram[t_addr];
                    data_out[23:16] <= ram[t_addr+1];
                    data_out[15:8] <= ram[t_addr+2];
                    data_out[7:0] <= ram[t_addr+3];
                end
            else
                begin
                    data_out[31:24] <= ram[t_addr+4];
                    data_out[23:16] <= ram[t_addr+5];
                    data_out[15:8] <= ram[t_addr+6];
                    data_out[7:0] <= ram[t_addr+7];
                end
            #3;
        end
    endcase

    #5 Clear <= 1'b1;
    end
end

always@(Ready)
begin
    if (Ready == 1'b1)
```

```
                Clear <= 1'b0;
end

endmodule
// Register File
// Controls 16 highly available 32-bit registers
module register_file (output reg [31:0] PA, PB, input [31:0] in, input [3:0] SA, SB, C, input
enable, clr, clk);

reg decoder_enable;

wire [31:0] temp;
wire [31:0] temp1;
wire [15:0] decoder_out;
wire [31:0] register_out0, register_out1, register_out2, register_out3, register_out4,
register_out5, register_out6, register_out7,
        register_out8, register_out9, register_out10, register_out11, register_out12,
register_out13, register_out14, register_out15;

always @ (enable)
begin

   if(!enable)

     begin

        decoder_enable = 1'b1;

     end

   else

      begin

        decoder_enable = 1'b0;

      end

end

always @ (*)
begin
        PA = temp;
        PB = temp1;
```

```
end


decoder dec (decoder_out, C, decoder_enable);

register_32_bits register0 (register_out0, in, ~decoder_out[0], clr, clk);
register_32_bits register1 (register_out1, in, ~decoder_out[1], clr, clk);
register_32_bits register2 (register_out2, in, ~decoder_out[2], clr, clk);
register_32_bits register3 (register_out3, in, ~decoder_out[3], clr, clk);
register_32_bits register4 (register_out4, in, ~decoder_out[4], clr, clk);
register_32_bits register5 (register_out5, in, ~decoder_out[5], clr, clk);
register_32_bits register6 (register_out6, in, ~decoder_out[6], clr, clk);
register_32_bits register7 (register_out7, in, ~decoder_out[7], clr, clk);
register_32_bits register8 (register_out8, in, ~decoder_out[8], clr, clk);
register_32_bits register9 (register_out9, in, ~decoder_out[9], clr, clk);
register_32_bits register10 (register_out10, in, ~decoder_out[10], clr, clk);
register_32_bits register11 (register_out11, in, ~decoder_out[11], clr, clk);
register_32_bits register12 (register_out12, in, ~decoder_out[12], clr, clk);
register_32_bits register13 (register_out13, in, ~decoder_out[13], clr, clk);
register_32_bits register14 (register_out14, in, ~decoder_out[14], clr, clk);
register_32_bits register15 (register_out15, in, ~decoder_out[15], clr, clk);




mux_16to1 muxA (temp, SA, register_out0, register_out1, register_out2, register_out3,
register_out4, register_out5, register_out6, register_out7,
        register_out8, register_out9, register_out10, register_out11, register_out12,
register_out13, register_out14, register_out15);

mux_16to1 muxB (temp1, SB, register_out0, register_out1, register_out2, register_out3,
register_out4, register_out5, register_out6, register_out7,
        register_out8, register_out9, register_out10, register_out11, register_out12,
register_out13, register_out14, register_out15);

endmodule
//Register of 32 bits

module register_32_bits (output reg [31:0] Y, input [31:0] I,
                        input LE, CLR, CLK);

always @ (posedge CLK, negedge CLR)
```

```
if (!CLR)

   Y <= 32'h00000000; //Clear es active low, 0 = clear y 1 = no esta clear

else if ((LE == 0) && (CLK == 1))

   Y = I;

endmodule
//Shifter Module
module Shifter (output reg [31:0] result, output reg C_flag, input [31:0] data, input [1:0] shift,
input [31:0] shiftNum , input enable);

reg [31:0] tempNum;
reg [31:0] tempData;
reg [31:0] temp;
integer i;

always @ (data, posedge enable, shift, C_flag)

//Active-low
if (enable==1'b0)
        begin

                tempNum = shiftNum;
                tempData = data;
                temp = data;

                        case(shift)

                                2'b00: // LSL
                                begin
                                        for(i = 0; i < tempNum; i = i+1)
                                        begin
                                                C_flag = tempData[31:31];
                                                tempData= tempData<<1;
                                        end
                                end

                                2'b01: // LSR
                                begin
                                        for(i = 0; i < tempNum; i = i+1)
                                        begin
                                                C_flag = tempData[0];
                        tempData= tempData>>1;
                    end
                end

                2'b10: // ASR
                begin
                    for(i = 0; i < tempNum; i = i+1)
                        begin
                        tempData= tempData>>1;
                        tempData = {data[31:31], tempData[30:0]};
                        end
                end

                2'b11: // ROR
                begin
                    for(i = 0; i < tempNum; i = i+1)
                        begin
                        temp = tempData;
                        tempData= tempData>>1;
                        tempData = {temp[0:0], tempData[30:0]};
                        end
                end

            endcase
    result = tempData;
    end
else begin
    result = data;
end

endmodule
// Sign Extension for Memory Data

module Sign_Extension2 (output reg [31:0] Y, input [31:0] in, input [1:0] dataType, input
enable);

always @ (in, dataType)

begin
  if (!enable)
    case(dataType)
         2'b00: // Byte
             begin
                 assign Y[7:0] = in[7:0];
```

```
                                     assign Y[31:8] = {24{in[7]}};
                             end
                  2'b01: // Halfword
                           begin
                                    assign Y[15:0] = in[15:0];
                                    assign Y[31:16] = {16{in[15]}};
                           end
                  2'b10: // Word
                           begin
                                    assign Y = in;
                           end
                  2'b11: // Doubleword
                           begin
                                    assign Y = in;
                           end
    endcase
  else
    assign Y = in;
end

endmodule
```