ENG 463-L1

Lab #2 ARM Pipeline Implementation

Brian Worts and Chris Jenson

03/16/21

Table of Contents

Introduction:

For this lab, a pipelined ARM architecture processor will be implemented and verified in Verilog. Pipelining is the process of fetching the next instruction while the current instruction is being executed. This is supported by the processor to increase the speed of program execution. ARM processors are based on the RISC (reduced instruction set computer) architecture to perform a smaller set of instructions at high speeds. They provide high performance while requiring considerably less power than devices using CISC (complex instruction set computing). By having less instructions, they require fewer transistors allowing for small sizes of the integrated circuitry. The ARM processor's smaller size, low complexity, and lower power usage make them ideal for small devices.

Problem Statement:

Implement the pipelined ARM architecture using an as high as possible level of abstract design methodology. The implementation will only have to support the following instructions (simplified ISA): LDUR, STUR, ADD, SUB, AND, ORR, and CBZ. The architecture will be designed based on *Figure 1*. The memories, as well as the register file, are of the asynchronous read, synchronous write type. The control is based on *Figures 2 & 3*. The design must be verified using a testbench.

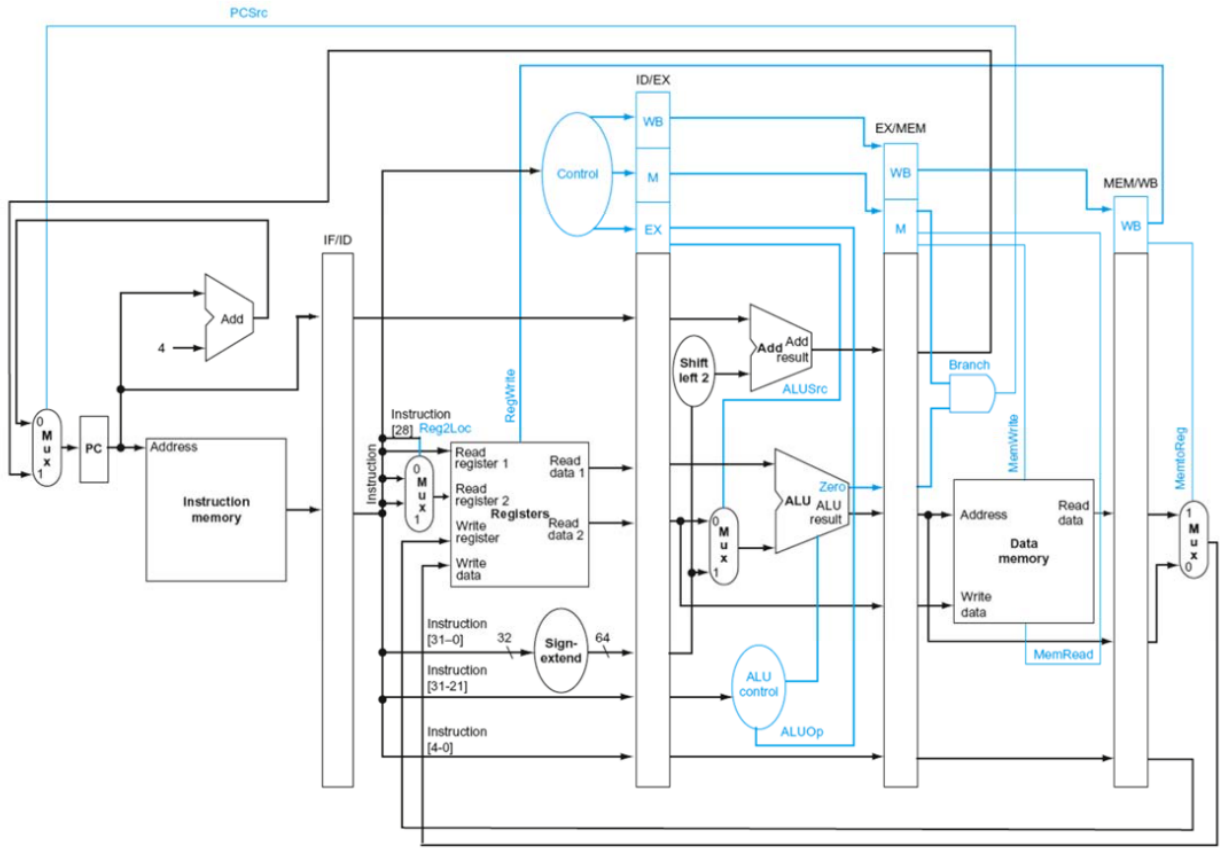

Figure 1: Overall Architecture

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
LDUR	00	load register	XXXXXXXXXXX	add	0010
STUR	00	store register	XXXXXXXXXXX	add	0010
CBZ	01	compare and branch on zero	XXXXXXXXXXX	pass input b	0111
R-type	10	add	10001011000	add	0010
		subtract	11001011000	subtract	0110
		AND	10001010000	AND	0000
		ORR	10101010000	OR	0001

Figure 2: ALU Control Truth Table

Instruction	Instruction decode stage control lines	Execution/address calculation stage control lines			Memory access stage control lines			Write-back stage control lines	
	Reg2Loc	ALUOp1	ALUOp0	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemtoReg
R-format	0	1	0	0	0	0	0	1	0
LDUR	X	0	0	1	0	1	0	1	1
STUR	1	0	0	1	0	0	1	0	X
CBZ	1	0	1	0	1	0	0	0	X

Figure 3: Main Control Truth Table

Procedure/Results:

The group started by looking at Lab 5 from ELC 363 where a simplified version of the processor was made. The IM, DM, and register blocks were still applicable to this design. The IM, DM, and RM (register map) files followed the same principle but IM had to be changed to avoid data hazards. The instructions were chosen to create a comprehensive test of the system and were originally written in Arm Assembly Language:

Add X1, X2, X3
And X4, X5, X6
Sub X7, X8, X9
Orr X10, X11, X12
Ldur X14, [X13, #0]
Stur X16, [X15, #0]
Add X17, X18, X19
Cbz X0, #-16

The instructions were then broken down into their components based on format in binary and then converted into hexadecimal. This is shown in **Table 1**.

Table 1*:* This table shows the deconstruction of the instructions

Assembly	Instruct	Opcode	Rm	Shamt	Add	Op	Rn	Rd/Rt	Hex	Binary
Add X1, X2, X3	ADD	10001011000	00010	000000			00011	00001	8B020061	10001011000000100000000001100001

And X4, X5, X6	AND	10001010000	00101	000 000			00110	00100	8A0500 C4	1000101000000101 0000000011000100
Sub X7, X8, X9	SUB	11001011000	01000	000 000			01001	00111	CB080 127	1100101100001000 0000000100100111
Orr X10, X11, X12	Or	10101010000	01011	000 000			01100	01010	AA0B0 18A	1010101000001011 0000000110001010
Ldur X14, [X13, #0]	LDUR	11111000010			00000 0000	00	01101	01110	F84001 AE	1111100001000000 0000000110101110
Stur X16, [X15, #0]	STUR	11111000000			00000 0000	00	10000	01111	F80002 0F	1111100000000000 0000001000001111
Add X17, X18, X19	ADD	10001011000	10010	000 000			10011	10001	8B1202 71	1000101100010010 0000001001110001
Cbz X0, #-16	CBX	10110100			11111 11111 11111 1000			00000	B4FFF F80	1011010011111111 1111111100010000

With the inputs, outputs, and instructions prepared, the group moved on to construct the various basic modules the design would need. These modules include the MUX's, add blocks, shift block, and the sign extend. The pipes between modules were ready to be connected. Starting with the ALU. Here is where the operations are performed based on the opcode and the registers defined in the instruction.

A major block to be programmed was the control block. This block determined the control lines based on the instruction. Control lines go to the various MUX's and influence what the processor is executing by determining what is passed on to the blocks. With all of these components finished, simulation was used to check if it was performing as desired. The results are shown below in Figures 4-9.

The simulation was run with a 100MHz clock signal and a 1ns propagation delay for all assignments. In Figures 4-9, the blue dividers represent the different sections of the pipelined processor: IF, ID, EX, MEM, and WB as labeled in the figures. The figures show red boxes around each clock cycle to show the progression of commands through the processor. The commands are written in the pink boxes inside of the pipeline dividers. Lastly, the values stored in each register and each memory location are equal to the index of each register. For example, X4 holds value 4 unless changed via an instruction.

Figure 4: First 3 clock cycles of pipeline, showing instructions moving through the pipeline and modules working as expected

Figure 5: Clock cycles 4-6 of pipeline, showing instructions moving through the pipeline and modules working as expected. ADD and AND successfully complete in this figure. Figure 8 shows the registers changing appropriately.

Figure 6: Clock cycles 7-9 of pipeline, showing instructions moving through the pipeline and modules working as expected. SUB, ORR, and LDUR successfully complete in this figure. Figure 8 shows the registers changing appropriately.

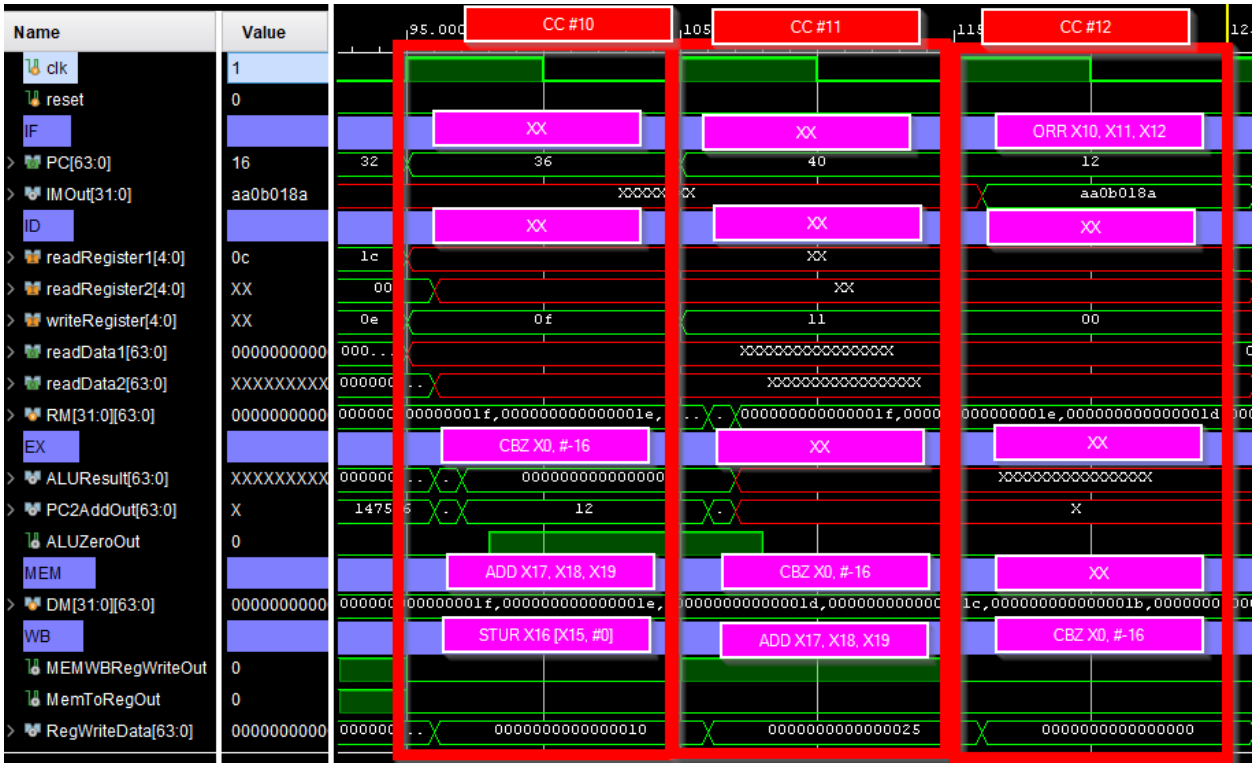

Figure 7: Clock cycles 10-12 of pipeline, showing instructions moving through the pipeline and modules working as expected. STUR, ADD, and CBZ successfully complete in this figure. Figure 8 shows the registers changing appropriately and Figure 9 shows the DM changing appropriately for STUR. PC can be seen in this image to be 16 less than it is in the MEM section of the pipeline (28 - 16 = 12) because of the CBZ instruction.

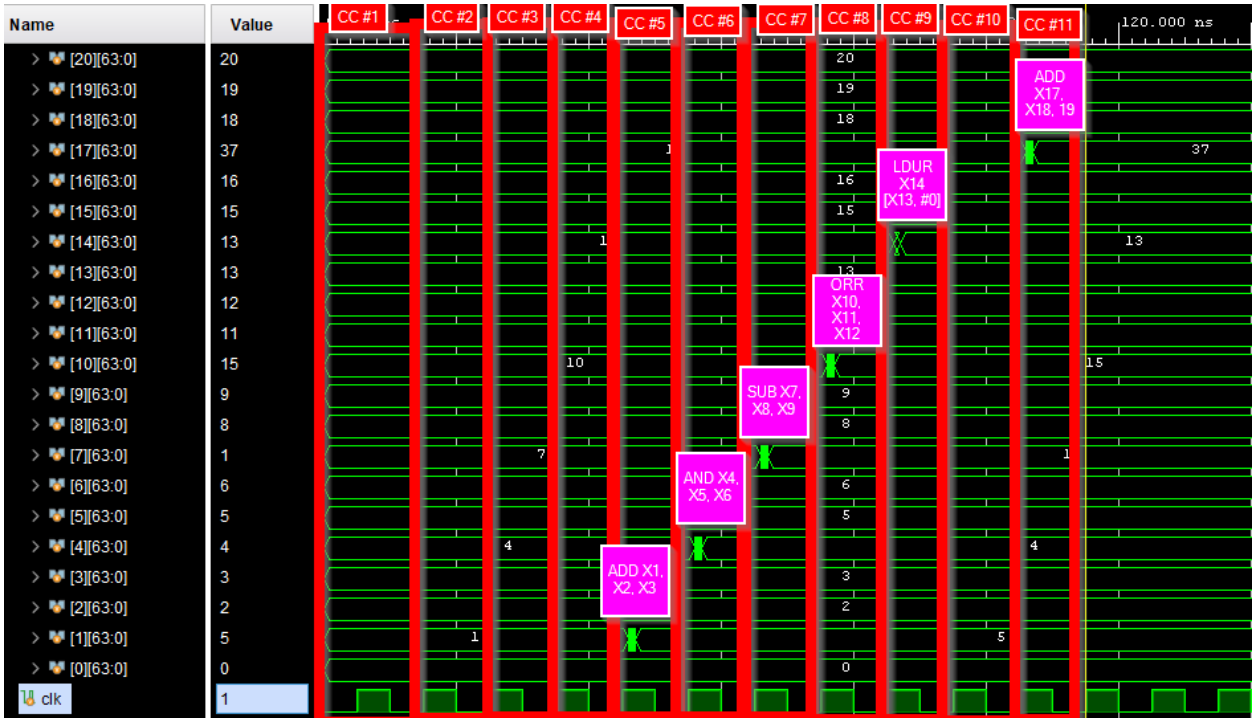

Figure 8: Registers changing with pipeline clock cycle shown in red at the top and the instructions in the pink boxes. The pink boxes are located directly above the register they are changing.

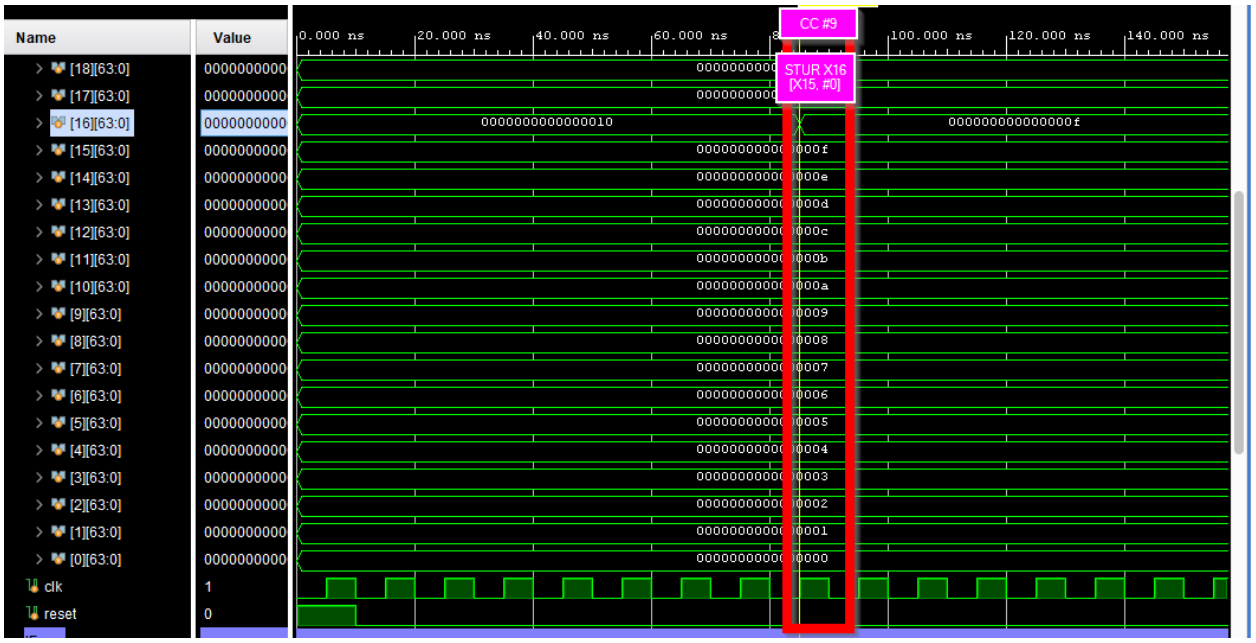

Figure 9: Data Memory with the STUR instruction highlighted in red, where it alters DM in the pipeline

Discussion:

One of the issues the group encountered was that there are an exorbitant amount of wires across the design. Naming conventions had to be strictly followed to avoid bugs and confusion throughout the entire process. A second issue the group had to account for was the possibility of data hazards. This wasn't a potential issue in ELC 363 because the processor ran in a single clock cycle. Pipelined processors run over multiple cycles which is where data hazards occur. An instruction may intend to alter the value in a register which a later process accesses. In this case, the later process might not get the correct value because the first process did not finish executing.

Conclusion:

The use of a testbench with simulation showed that the design in *Figure 1* was successfully implemented. Memory was able to be read in and written to. The waveforms in the Results section showed that the state of the CPU (PC and pertinent registers), and the pertinent memory contents after each instruction was executed matched what was expected. Implementing this design gave the group a deeper understanding of the ARM architecture as how it may be implemented in a real world application. It also showed the value of ARM implementations over potential other architectures due to its simplified nature.

Appendix A (ARM Assemble Test Program):

```
Add X1, X2, X3
And X4, X5, X6
Sub X7, X8, X9
Orr X10, X11, X12
Ldur X14, [X13, #0]
Stur X16, [X15, #0]
Add X17, X18, X19
Cbz X0, #-16
```

Appendix B (Initial Program Memory Load File):

```
@0    8B020061
@4    8A0500C4
@8    CB080127
@c    AA0B018A
@10   F84001AE
@14   F800020F
@18   8B120271
@1c   B4FFFF80
```

Appendix C (Verilog Files):

```
`timescale 1ns / 1ps

module testbench();
    ////////////////////////////////////////
    //CLK
    reg clk;

    //Generate Clock
    initial
    begin
        clk = 0;
        while(1)
        begin
            #5 clk = ~clk;
        end
    end
    ////////////////////////////////////////////
    //Reset
    reg reset; //Active Low
    initial
    //Initize reset
    begin
        reset = 1;
        #10 reset = 0;
    end
    ////////////////////////////////////////////


    always @(posedge clk)
    //FRAME
    begin
        if (reset)
        begin

        end
        else
        begin

        end
    end

    wire [63:0] PC;

    ////////////////////////////////////////////
```

```
    //IM
    wire [31:0] IMOut;

    IM u_IM (
        .PC(PC),
        .IMOut(IMOut)
    );
    ////////////////////////////////////////////
    //DM
    wire DMMemWrite;
    wire DMMemRead;

    wire [63:0] DMAddress;
    wire [63:0] DMWriteData;
    wire [63:0] DMReadData;

    DM u_DM(
        .DMMemWrite(DMMemWrite),
        .DMMemRead(DMMemRead),

        .DMAddress(DMAddress),
        .DMWriteData(DMWriteData),
        .DMReadData(DMReadData)
    );
    ////////////////////////////////////////////

    Datapath u_Datapath (
        .clk(clk),
        .reset(reset),

        //PC
        .PC(PC),

        //IM
        .IFIMOut(IMOut),

        //DM
        .DMMemWrite(DMMemWrite),
        .DMMemRead(DMMemRead),

        .DMAddress(DMAddress),
        .DMWriteData(DMWriteData),
        .DMReadData(DMReadData)
    );

endmodule
```

```
`timescale 1ns / 1ps

module IM(
      input [63:0] PC,
      output [31:0] IMOut
   );

   reg [31:0] IM[31:0];

   initial
   begin
   //Initialize IM
      $readmemh("D:/CELab2/ARM-Pipeline-Processor/IM.dat", IM);
   end

   assign #1 IMOut = IM[PC];

Endmodule
```

```
`timescale 1ns / 1ps

module DM(
        input DMMemWrite,
        input DMMemRead,

        input [63:0] DMAddress,
        input [63:0] DMWriteData,
        output [63:0] DMReadData
    );

    reg [63:0] DM[31:0];

    initial begin
    //Initialize IM
        $readmemh("D:/CELab2/ARM-Pipeline-Processor/DM.dat", DM);
    end

    assign DMReadData = DM[DMAddress];

    always @(*)
    begin
        if (DMMemWrite)
        begin
            DM[DMAddress] = DMWriteData;
        end
    end
endmodule
```

```
`timescale 1ns / 1ps

    //Async read, sync write
        //if write and read at same time, should read old value

module Datapath(
    input clk,
    input reset,
    //PC
    output [63:0] PC,
    //IM
    input [31:0] IFIMOut,
    //DM
    output DMMemWrite,
    output DMMemRead,

    output [63:0] DMAddress,
    output [63:0] DMWriteData,
    input [63:0] DMReadData
    );

    ////////////////////////////////
    //IF
    wire PCSrc;

    wire [63:0] PCMuxOut;

    wire [63:0] PCAddOut;
    wire [63:0] EXMEMPCAdd2Out;

    Mux u_PCMux(
        .inputA(PCAddOut),
        .inputB(EXMEMPCAdd2Out),
        .selector(PCSrc),
        .outputData(PCMuxOut)
    );

    Add u_PCAdd(
        .inputA(PC),
        .inputB(64'd4), //Always add 4
        .outputData(PCAddOut)
    );

    PC u_PC(
        .clk(clk),
        .reset(reset),
```

```
    .PC(PC),
    .PCMuxOut(PCMuxOut)
);
////////////////////////////////
//IF/ID
wire [63:0] IFIDPCOut;
wire [31:0] IFIDIMOut;

IF_ID u_IF_ID(
    .clk(clk),
    .reset(reset),

    .IFIDPCIn(PC),
    .IFIDPCOut(IFIDPCOut),
    .IFIDIMIn(IFIMOut),
    .IFIDIMOut(IFIDIMOut)
);
////////////////////////////////
//ID
wire [4:0] ReadRegister2In;
wire [4:0] Reg2MuxAIn;
wire [4:0] Reg2MuxBIn;
wire Reg2Loc;
assign Reg2Loc = IFIDIMOut[28];
assign Reg2MuxAIn = IFIDIMOut[20:16];
assign Reg2MuxBIn = IFIDIMOut[4:0];
Mux u_RegsMux(
    .inputA(Reg2MuxAIn),
    .inputB(Reg2MuxBIn),
    .selector(Reg2Loc),
    .outputData(ReadRegister2In)
);

wire [4:0] ReadRegister1In;
wire [4:0] WriteRegister;
wire [63:0] RegWriteData;
wire [63:0] ReadData1;
wire [63:0] ReadData2;
wire [63:0] InsSignExtendOut;

wire MEMWBRegWriteOut;


Registers u_Registers(
    .readRegister1(IFIDIMOut[9:5]),
    .readRegister2(ReadRegister2In),
```

```
    .writeRegister(WriteRegister),
    .writeData(RegWriteData),

    .regWrite(MEMWBRegWriteOut),

    .readData1(ReadData1),
    .readData2(ReadData2)
);

assign #1 InsSignExtendOut = { {32{IFIDIMOut[31] }}, IFIDIMOut[31:0] };

wire [1:0] WBControlOut;
wire [2:0] MControlOut;
wire [2:0] EXControlOut;


Control u_Control(
    .Instruction(IFIDIMOut),

    .WB(WBControlOut),
    .M(MControlOut),
    .EX(EXControlOut)
);
////////////////////////////////
//ID/EX
wire [1:0] IDEXWBControlOut;
wire [2:0] IDEXMOut;
wire [1:0] IDEXALUOpOut;
wire IDEXALUSrcOut;
wire [63:0] IDEXPCOut;
wire [63:0] IDEXReadData1Out;
wire [63:0] IDEXReadData2Out;
wire [63:0] IDEXSignExtendOutOut;
wire [10:0] IDEXOpcodeOut;
wire [4:0] IDEXWriteRegisterOut;

ID_EX u_ID_EX(
    .clk(clk),

    .IDEXWBIn(WBControlOut),
    .IDEXWBOut(IDEXWBControlOut),

    .IDEXMIn(MControlOut),
    .IDEXMOut(IDEXMOut),

    .IDEXEXIn(EXControlOut),
```

```
    .IDEXALUOpOut(IDEXALUOpOut),
    .IDEXALUSrcOut(IDEXALUSrcOut),

    .IDEXPCIn(IFIDPCOut),
    .IDEXPCOut(IDEXPCOut),

    .IDEXReadData1In(ReadData1),
    .IDEXReadData1Out(IDEXReadData1Out),

    .IDEXReadData2In(ReadData2),
    .IDEXReadData2Out(IDEXReadData2Out),

    .IDEXSignExtendOutIn(InsSignExtendOut),
    .IDEXSignExtendOutOut(IDEXSignExtendOutOut),

    .IDEXOpcodeIn(IFIDIMOut[31:21]),
    .IDEXOpcodeOut(IDEXOpcodeOut),

    .IDEXWriteRegisterIn(IFIDIMOut[4:0]),
    .IDEXWriteRegisterOut(IDEXWriteRegisterOut)
);

////////////////////////////////
//EX
wire [63:0] PC2AddOut;
reg [20:0] IDEXSignExtendOutOutShifted;
always @(*)
begin
        IDEXSignExtendOutOutShifted = #1 (IDEXSignExtendOutOut[23:5] << 2);
end

BranchAdd u_BranchAdd(
    .inputA(IDEXPCOut),
    .inputB(IDEXSignExtendOutOutShifted), //Shift Left 2

    .outputData(PC2AddOut)
);

wire [63:0] ALUMuxOut;
Mux u_ALUMux(
    .inputA(IDEXReadData2Out),
    .inputB(IDEXSignExtendOutOut),//DEBUG BITS????????????????????????????
    .selector(IDEXALUSrcOut),
    .outputData(ALUMuxOut)
);
```

```
wire [63:0] ALUResult;
wire ALUZeroOut;
wire [3:0] ALUControlOut;

ALUControl u_ALUControl(
    .OpcodeIn(IDEXOpcodeOut),
    .ALUOp(IDEXALUOpOut),

    .ALUControlOut(ALUControlOut)
);

ALU u_ALU(
    //Input Data
    .inputA(IDEXReadData1Out),
    .inputB(ALUMuxOut),
    //ALUOp
    .ALUOp(IDEXALUOpOut),
    //Control
    .ALUControl(ALUControlOut),
    //Output Data
    .ALUOutput(ALUResult),
    .Zero(ALUZeroOut)
);
////////////////////////////////
//EX/MEM
wire [1:0] EXMEMWBOut;
wire EXMEMBranchOut;
wire EXMEMMemWriteOut;
wire EXMEMMemReadOut;

wire EXMEMALUZeroOut;
wire [63:0] EXMEMALUResultOut;
wire [63:0] EXMEMReadData2Out;
wire [4:0] EXMEMWriteRegisterOut;

EX_MEM u_EX_MEM(
    .clk(clk),

    .EXMEMWBIn(IDEXWBControlOut),
    .EXMEMWBOut(EXMEMWBOut),

    .EXMEMMIn(IDEXMOut),
    .EXMEMBranchOut(EXMEMBranchOut),
    .EXMEMMemWriteOut(EXMEMMemWriteOut),
    .EXMEMMemReadOut(EXMEMMemReadOut),
```

```
    .EXMEMPC2AddIn(PC2AddOut),
    .EXMEMPC2AddOut(EXMEMPCAdd2Out),

    .EXMEMALUZeroIn(ALUZeroOut),
    .EXMEMALUZeroOut(EXMEMALUZeroOut),
    .EXMEMALUResultIn(ALUResult),
    .EXMEMALUResultOut(EXMEMALUResultOut),

    .EXMEMReadData2In(IDEXReadData2Out),
    .EXMEMReadData2Out(EXMEMReadData2Out),
    .EXMEMWriteRegisterIn(IDEXWriteRegisterOut),
    .EXMEMWriteRegisterOut(EXMEMWriteRegisterOut)
);
////////////////////////////////
//MEM
assign PCSrc = EXMEMBranchOut & EXMEMALUZeroOut;
assign DMAddress = EXMEMALUResultOut;
assign DMWriteData = EXMEMReadData2Out;
assign DMMemWrite = EXMEMMemWriteOut;
assign DMMemRead = EXMEMMemReadOut;
////////////////////////////////
//MEM/WB
wire MemToRegOut;
wire [63:0] MEMWBDMReadDataOut;
wire [63:0] MEMWBALUResultOut;
wire [4:0] MEMWBWriteRegisterOut;


MEM_WB u_MEM_WB(
    .clk(clk),

    .MEMWBWBIn(EXMEMWBOut),
    .MEMWBRegWriteOut(MEMWBRegWriteOut),
    .MEMWBMemToRegOut(MemToRegOut),

    .MEMWBDMReadDataIn(DMReadData),
    .MEMWBDMReadDataOut(MEMWBDMReadDataOut),

    .MEMWBALUResultIn(EXMEMALUResultOut),
    .MEMWBALUResultOut(MEMWBALUResultOut),

    .MEMWBWriteRegisterIn(EXMEMWriteRegisterOut),
    .MEMWBWriteRegisterOut(MEMWBWriteRegisterOut)
);
////////////////////////////////
//WB
```

```
    Mux u_WBMux(
        .inputA(MEMWBALUResultOut),
        .inputB(MEMWBDMReadDataOut),
        .selector(MemToRegOut),
        .outputData(RegWriteData)
    );
    assign WriteRegister = MEMWBWriteRegisterOut;

    ////////////////////////////////
endmodule
```

```
`timescale 1ns / 1ps

module Mux(
    input clk,
    input reset,

    input [63:0] inputA,
    input [63:0] inputB,
    input selector,
    output reg [63:0] outputData
    );

    always @(*)
    //Select data in Mux
    //Output
        //outputData
    begin
        if(reset)
        begin
            //#1 outputData = 0;
        end
        else
        begin
            if (selector)
            begin
                #1 outputData = inputB;
            end
            else
            begin
                #1 outputData = inputA;
            end
        end
    end


    /*
    always @(posedge clk)
    //Select data in Mux
    //Output
        //outputData
    begin
        if(reset)
        begin
            outputData <= 0;
        end
        else
```

```
        begin
            if (selector)
            begin
                outputData <= inputB;
            end
            else
            begin
                outputData <= inputA;
            end
        end
    end*/

endmodule
```

```
`timescale 1ns / 1ps

module Add(
   input clk,
   input reset,

   input [63:0] inputA,
   input [63:0] inputB,

   output reg [63:0] outputData
   );

   always @(*)
   begin
      if (reset)
      begin
        // #1 outputData = 0;
      end
      else
      begin
         #1 outputData = inputA + inputB;
      end
   end

   /*
   always @(posedge clk)
   begin
      if (reset)
      begin
         outputData <= 0;
      end
      else
      begin
         outputData <= inputA + inputB;
      end
   end

   */
Endmodule
```

```
`timescale 1ns / 1ps

module PC(
    input clk,
    input reset,

    input [63:0] PCMuxOut,
    output reg [63:0] PC
    );

    always @(posedge clk)
    begin
        if (reset)
        begin
            PC <= 0;
        end
        else
        begin
            PC <= PCMuxOut;
        end
    end

Endmodule
```

```
`timescale 1ns / 1ps

module IF_ID(
    input clk,
    input reset,

    input [63:0] IFIDPCIn,
    output reg[63:0] IFIDPCOut,
    input [31:0] IFIDIMIn,
    output reg [31:0] IFIDIMOut
    );

    always @(posedge clk)
    begin
        if (reset)
        begin

        end
        else
        begin
            IFIDPCOut <= IFIDPCIn;
            IFIDIMOut <= IFIDIMIn;
        end
    end

endmodule
```

```
`timescale 1ns / 1ps

module Registers(
    input clk,
    input reset,
    //Control value
    input regWrite,
    //Reg locations (64)
    input [4:0] readRegister1,
    input [4:0] readRegister2,
    input [4:0] writeRegister,
    //Data (64)
    input [63:0] writeData,
    //Data (64)
    output [63:0] readData1,
    output [63:0] readData2
    );

    reg [63:0] RM[31:0];

    initial
    begin
        $readmemh("D:/CELab2/ARM-Pipeline-Processor/RM.dat", RM);
    end

    assign readData1 = RM[readRegister1];
    assign readData2 = RM[readRegister2];

    always @(*)
    begin
    //Drive RM
    //Output
        //RM
        if (reset)
        begin
            $readmemh("D:/CELab2/ARM-Pipeline-Processor/RM.dat", RM);
        end
        else
        begin
            if (regWrite)
            begin
                #1 RM[writeRegister] = writeData;
            end
        end
    end
endmodule
```

```
`timescale 1ns / 1ps

module Control(
    input [31:0] Instruction,

    output reg [1:0] WB,
    output reg [2:0] M,
    output reg [2:0] EX
    );



    always @(*)
    begin
        if ((Instruction[31:21] == 11'b10001011000) || (Instruction[31:21] == 11'b10001010000) ||
(Instruction[31:21] == 11'b10101010000) || (Instruction[31:21] == 11'b10001010000))
                                //add                           //sub                           //or
//and
        begin
            EX = 3'b100;
            M = 3'b000;
            WB = 2'b10;
        end
        if ((Instruction[31:21] == 11'b11111000010)) //LDUR
        begin
            EX = 3'b001;
            M = 3'b010;
            WB = 2'b11;
        end
        if ((Instruction[31:21] == 11'b11111000000)) //STUR
        begin
            EX = 3'b001;
            M = 3'b001;
            WB = 2'b00;
        end
        if ((Instruction[31:24] == 11'b10110100)) //CBZ
        begin
            EX = 3'b010;
            M = 3'b100;
            WB = 2'b00;
        end
    end

Endmodule
```

```
`timescale 1ns / 1ps

module ID_EX(
    input clk,

    input [1:0] IDEXWBIn,
    output reg [1:0] IDEXWBOut,

    input [2:0] IDEXMIn,
    output reg [2:0] IDEXMOut,

    input [2:0] IDEXEXIn,
    output reg [1:0] IDEXALUOpOut,
    output reg IDEXALUSrcOut,

    input [63:0] IDEXPCIn,
    output reg [63:0] IDEXPCOut,

    input [63:0] IDEXReadData1In,
    output reg [63:0] IDEXReadData1Out,

    input [63:0] IDEXReadData2In,
    output reg [63:0] IDEXReadData2Out,

    input [63:0] IDEXSignExtendOutIn,
    output reg [63:0] IDEXSignExtendOutOut,

    input [10:0] IDEXOpcodeIn,
    output reg [10:0] IDEXOpcodeOut,

    input [4:0]IDEXWriteRegisterIn,
    output reg [4:0] IDEXWriteRegisterOut
    );

    always @(posedge clk)
    begin
        IDEXWBOut <= IDEXWBIn;
        IDEXMOut <= IDEXMIn;
        IDEXALUOpOut <= IDEXEXIn[2:1];
        IDEXALUSrcOut <= IDEXEXIn[0];

        IDEXPCOut <= IDEXPCIn;
        IDEXReadData1Out <= IDEXReadData1In;
        IDEXReadData2Out <= IDEXReadData2In;
        IDEXSignExtendOutOut <= IDEXSignExtendOutIn;
        IDEXOpcodeOut <= IDEXOpcodeIn;
```

```
        IDEXWriteRegisterOut <= IDEXWriteRegisterIn;
    end
Endmodule
```

```
`timescale 1ns / 1ps

module BranchAdd(
    input clk,
    input reset,

    input signed [63:0] inputA,
    input signed [18:0] inputB,

    output reg signed [63:0] outputData
    );

    always @(*)
    begin
        if (reset)
        begin
            // #1 outputData = 0;
        end
        else
        begin
            #1 outputData = inputA + inputB;
        end
    end

endmodule
```

```
`timescale 1ns / 1ps

module ALUControl(
        input [10:0] OpcodeIn,
        input [1:0] ALUOp,

        output reg [3:0] ALUControlOut
    );


    always @(*)
    begin
        if (ALUOp == 2'b00)
        begin
            ALUControlOut = 4'b0010;
        end
        else if (ALUOp == 2'b01)
        begin
            ALUControlOut = 4'b0111;
        end
        else if (ALUOp == 2'b10)
        begin
            case(OpcodeIn)
                //case 0000: inputA AND(&) inputB
                11'b10001010000: begin
                    ALUControlOut = 4'b0000;
                end

                //case 0001: inputA OR(|) inputB
                11'b10101010000: begin
                    ALUControlOut = 4'b0001;
                end

                //case 0010: inputA add(+) inputB
                11'b10001011000: begin
                    ALUControlOut = 4'b0010;
                end

                //case 0110: inputA subtract(-) inputB
                11'b11001011000: begin
                    ALUControlOut = 4'b0110;
                end
            endcase
        end
    end
endmodule
```

```
`timescale 1ns / 1ps

module ALU(
    //Input Data
    input [63:0] inputA,
    input [63:0] inputB,
    //ALUOP(debug)
    input [1:0] ALUOp,
    //Control
    input [3:0] ALUControl,
    //Output Data
    output reg [63:0] ALUOutput,
    output reg Zero
    );

    reg [63:0] addInputB;

    wire [9:0] DebugInputB;
    assign DebugInputB = addInputB;

    always @(*)
    begin
        case(ALUOp)
            2'b00: begin
                addInputB = inputB[20:12];
            end

            default: begin
                addInputB = inputB;
            end
        endcase
    end

    always@(*) begin    //aluc
        //switch statement
        case(ALUControl)
            //case 0000: inputA AND(&) inputB
            4'b0000: begin
                ALUOutput = #1 inputA & inputB;
            end

            //case 0001: inputA OR(|) inputB
            4'b0001: begin
                ALUOutput = #1 inputA | inputB;
            end
```

```
        //case 0010: inputA add(+) inputB
        4'b0010: begin
            ALUOutput = #1 inputA + addInputB;
        end

        //case 0110: inputA subtract(-) inputB
        4'b0110: begin
            ALUOutput = #1 inputA - inputB;
        end

        //case 0111: Pass inputB, result = inputB
        4'b0111: begin
            ALUOutput = #1 inputB; //For CBZ
        end

    endcase
  end

  always@(*) begin //posedge CKLK
    if(ALUOutput == 0)
        Zero = #1 1;
    else
        Zero = #1 0;
  end
endmodule
```

```
`timescale 1ns / 1ps

module EX_MEM(
    input clk,

    input [1:0] EXMEMWBIn,
    output reg [1:0] EXMEMWBOut,
    input [2:0] EXMEMMIn,
    output reg EXMEMBranchOut,
    output reg EXMEMMemWriteOut,
    output reg EXMEMMemReadOut,

    input [63:0] EXMEMPC2AddIn,
    output reg [63:0] EXMEMPC2AddOut,

    input EXMEMALUZeroIn,
    output reg EXMEMALUZeroOut,
    input [63:0] EXMEMALUResultIn,
    output reg [63:0] EXMEMALUResultOut,

    input [63:0] EXMEMReadData2In,
    output reg [63:0] EXMEMReadData2Out,
    input [4:0] EXMEMWriteRegisterIn,
    output reg [4:0] EXMEMWriteRegisterOut
    );

    always @(posedge clk)
    begin
        EXMEMWBOut <= EXMEMWBIn;

        EXMEMBranchOut <= EXMEMMIn[2];
        EXMEMMemWriteOut <= EXMEMMIn[0];
        EXMEMMemReadOut <= EXMEMMIn[1];

        EXMEMPC2AddOut <= EXMEMPC2AddIn;
        EXMEMALUZeroOut <= EXMEMALUZeroIn;
        EXMEMALUResultOut <= EXMEMALUResultIn;
        EXMEMReadData2Out <= EXMEMReadData2In;
        EXMEMWriteRegisterOut <= EXMEMWriteRegisterIn;
    end
endmodule
```

```
`timescale 1ns / 1ps


module MEM_WB(
    input clk,

    input [1:0] MEMWBWBIn,
    output reg MEMWBRegWriteOut,
    output reg MEMWBMemToRegOut,

    input [63:0] MEMWBDMReadDataIn,
    output reg [63:0] MEMWBDMReadDataOut,

    input [63:0] MEMWBALUResultIn,
    output reg [63:0] MEMWBALUResultOut,

    input [4:0] MEMWBWriteRegisterIn,
    output reg [4:0] MEMWBWriteRegisterOut
    );

    always @ (posedge clk)
    begin
        MEMWBRegWriteOut <= MEMWBWBIn[1];
        MEMWBMemToRegOut <= MEMWBWBIn[0];

        MEMWBDMReadDataOut <= MEMWBDMReadDataIn;
        MEMWBALUResultOut <= MEMWBALUResultIn;
        MEMWBWriteRegisterOut <= MEMWBWriteRegisterIn;
    end

endmodule
```