ECSE 4770 LAB 5 REPORT

Dec. 2021

Yuchen Wang, Rin: 661960546, section 2

Partner: Yilu Zhou, Rin: 661931826, section 2

Abstract

- The purpose of this lab is to implement an Arm v4 CPU by Verilog and run it on the FPGA board. The basic framework of the Arm v4 CPU is provided on LMS, student need to implement serval instructions to fulfill the requirement of the lab, and run the provided instruction set of this lab. Here the missing functions in the CPU are EOR, BIC, MVN, TST, TEQ, CMP, CMN, and BL.
- After implementing these instructions, student needs to first simulate the CPU in Quartus and check the simulation result. Then run it on the FPGA board. The hex display on the board is used to verify if the instruction is correctly operated.

Overall design description

- Most code of the lab is provided. It is an Arm v4 CPU contains 16 registers, processing the instruction data and operate the instruction like common CPUs. There is no big change in the overall design.
- Students here only need to implement the ALU and data path in Verilog file. The following sections will include students implementation of each part.
- To make the CPU work, The provided file will read the instructions from a txt file provided on LMS. Then the CPU process the instructions. The result will printed on the hex display of the FPGA board. The left 2 bit is for memory address, right 2 bit for data written to memory. Which can be used to verify STR operations. The mid 2 bit is for program counter to verify the machine is running the instruction is correct order.

Schematic and State Diagram

- Student implementation is this lab focused on the data path part and the ALU part. In this section of the report, only the schematic of the two parts will be included.

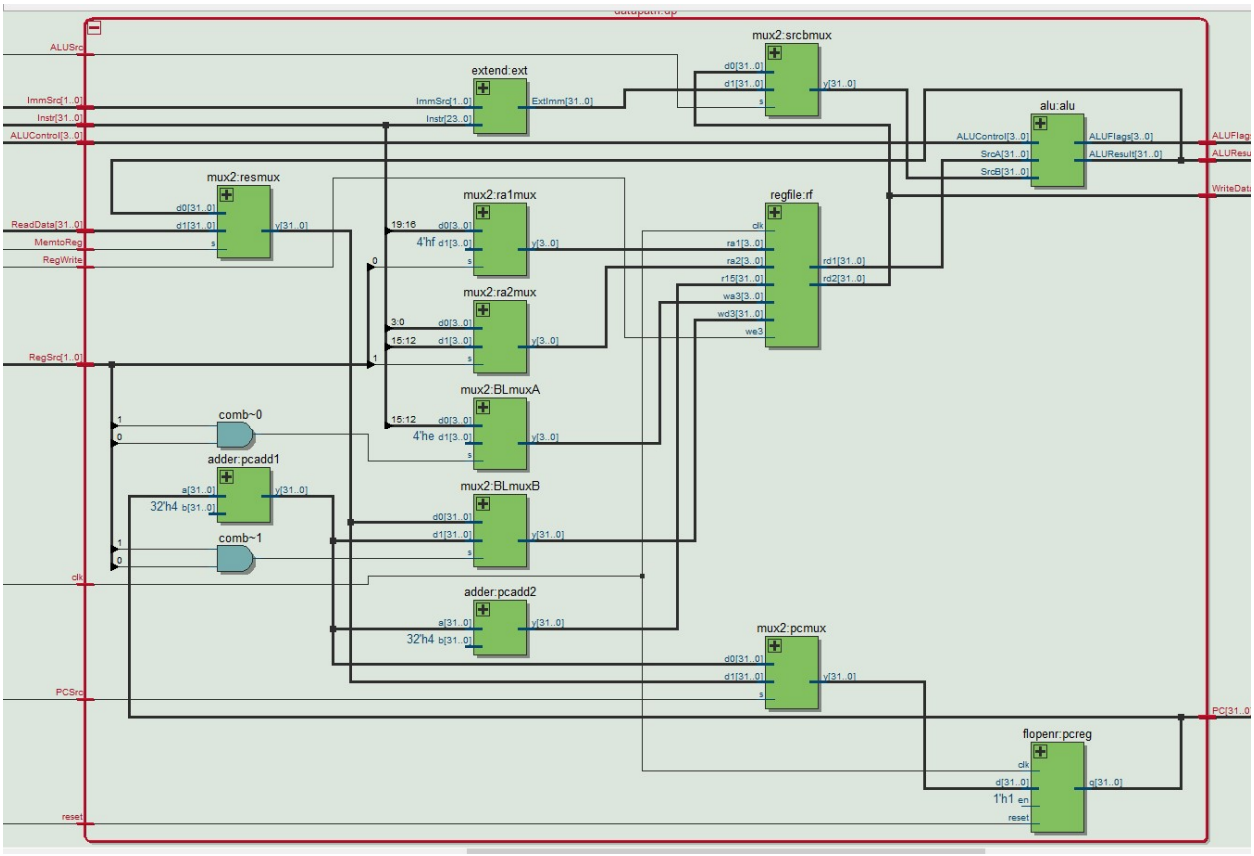


Figure: Schematic of data path

- This is the schematic for data path. To implement BL instruction, here 2 multiplexer is added to the data path file. BLmuxA and BLmuxB are the two MUXs.
- BLmuxA will select the address of the R14 when executing BL instruction. Other wife, it will select Instr[15:12] like the original version. Regfile, wa will receive the data from this mux.

- BLmuxB will select PCPlus4 when BL is called. This mux can save the PCPlus4 to R14. The result of this MUX is connected to wd of the regfile. When both MUXs are selected to d1, the PCPlus4 can be saved to R14 and the program can return to the stop point after finishing executing a branch.

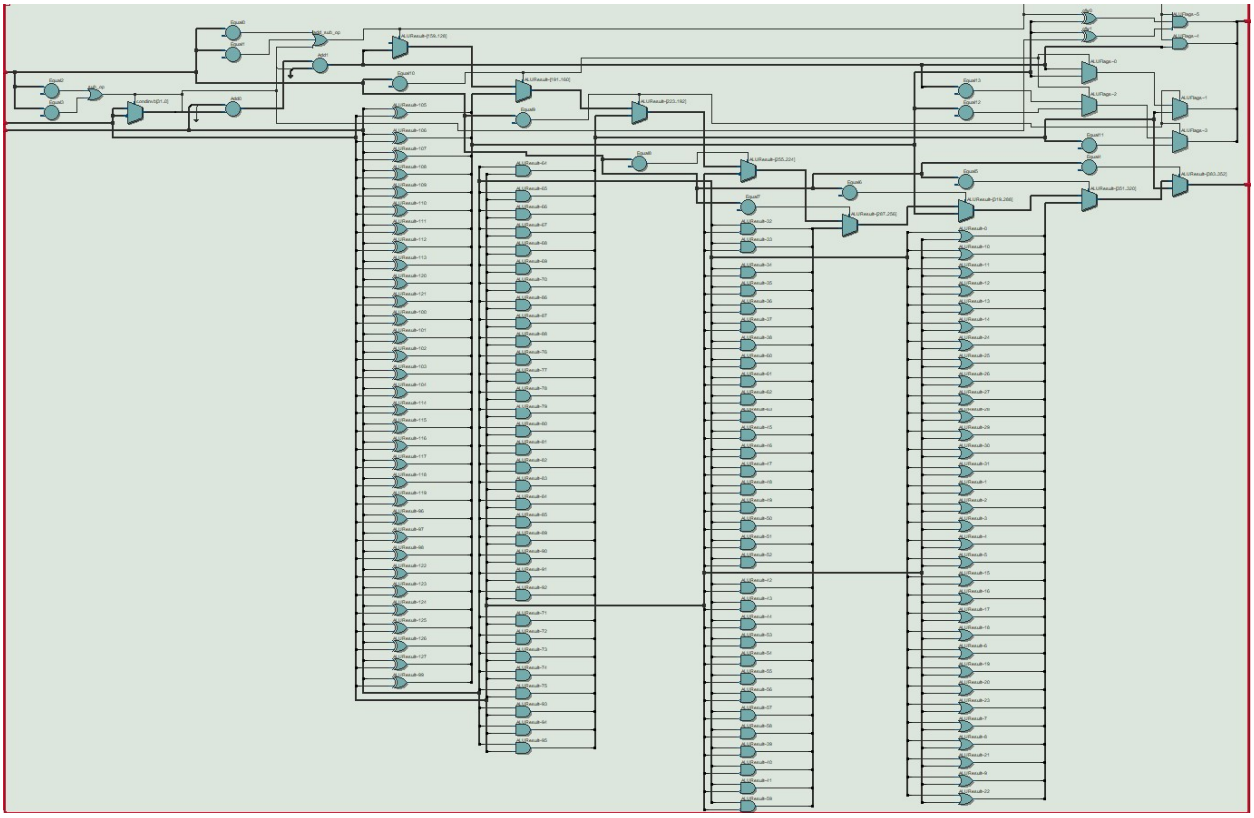


Figure: ALU schematic

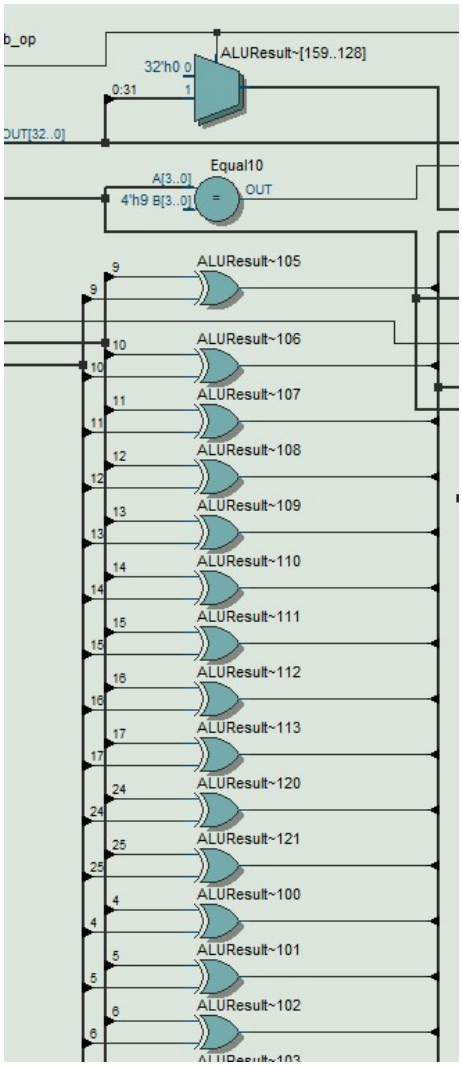


Figure: EOR of ALU

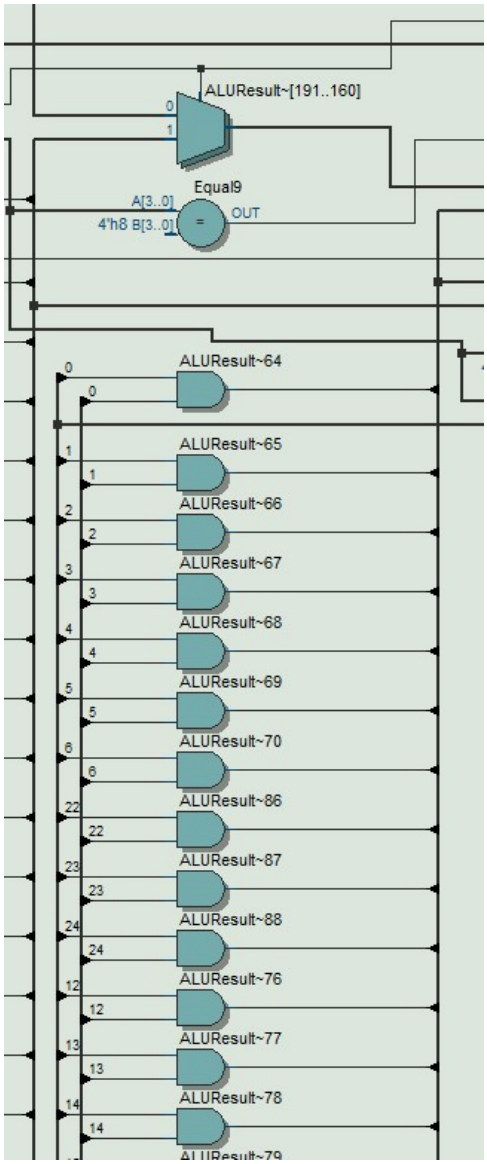


Figure: BIC of ALU

- EOR instruction uses 32 XOR gates which compare bit between SrcA and SrcB.
- BIC instruction uses 32 AND gates with one inverted input. The inverted input are the bits from SrcB. Since ALUResult to be SrcA & ~SrcB. The bits from SrcA and inverted SrcB should go through the AND gates.

Code

armv4.v

```
//__MAIN MODULES__

//full CPU
module armv4(input clk,reset,
    input [31:0] ReadData,
    input [31:0] Instr,
    output [31:0] PC,
    output MemWrite,
    output [31:0] ALUResult,WriteData);

    wire [3:0] ALUFlags, ALUControl;
    wire RegWrite, ALUSrc, MemtoReg, PCSrc;
    wire [1:0] RegSrc,ImmSrc;

    controller
c( .clk(clk), .reset(reset), .Instr(Instr[31:12]), .ALUFlags(ALUFlags), .RegSrc(R
egSrc), .RegWrite(RegWrite), .ImmSrc(ImmSrc), .ALUSrc(ALUSrc), .ALUControl(ALUCon
trol), .MemWrite(MemWrite), .MemtoReg(MemtoReg), .PCSrc(PCSrc) );
    datapath
dp(.clk(clk), .reset(reset), .RegSrc(RegSrc), .RegWrite(RegWrite), .ImmSrc(ImmSrc
), .ALUSrc(ALUSrc), .ALUControl(ALUControl), .MemtoReg(MemtoReg), .PCSrc(PCSrc),
.ALUFlags(ALUFlags), .PC(PC), .Instr(Instr), .ALUResult(ALUResult), .WriteData(Wr
iteData), .ReadData(ReadData) );

endmodule

//datapath
module datapath(input clk,reset,
    input [1:0] RegSrc,
    input RegWrite,
    input [1:0] ImmSrc,
    input ALUSrc,
    input [3:0] ALUControl,
    input MemtoReg,
    input PCSrc,
    input [31:0] Instr,
    input [31:0] ReadData,
    output [3:0] ALUFlags,
    output [31:0] PC,
```

```
    output [31:0] ALUResult, WriteData);

    wire [31:0] PCNext,PCPlus4,PCPlus8;
    wire [31:0] ExtImm, SrcA, SrcB, Result;
    wire [3:0] RA1, RA2;
    wire [3:0] temp1;
    wire [31:0] temp2;
    //next PC logic
    mux2 #(32) pcmux(.d0(PCPlus4), .d1(Result), .s(PCSrc), .y(PCNext) );
    flopenr #(32)
pcreg(.clk(clk), .reset(reset), .d(PCNext), .q(PC), .en(1'b1) );
    adder #(32) pcadd1(.a(PC), .b(32'b100), .y(PCPlus4) );
    adder #(32) pcadd2(.a(PCPlus4), .b(32'b100), .y(PCPlus8) );

    //register file logic
    mux2 #(4) ra1mux(.d0(Instr[19:16]), .d1(4'b1111), .s(RegSrc[0]), .y(RA1));
    mux2 #(4) ra2mux(.d0(Instr[3:0]), .d1(Instr[15:12]), .s(RegSrc[1]), .y(RA2));
    // regfile rf
(.clk(clk), .we3(RegWrite), .ra1(RA1), .ra2(RA2), .wa3(Instr[15:12]), .wd3(Result
), .r15(PCPlus8), .rd1(SrcA), .rd2(WriteData) );
    mux2 #(4) BLmuxA(.d0(Instr[15:12]), .d1(4'b1110), .s(RegSrc[0] &
RegSrc[1]), .y(temp1)); // Additional mux2 to set BL
    mux2 #(32) BLmuxB (.d0(Result), .d1(PCPlus4), .s(RegSrc[0] &
RegSrc[1]), .y(temp2));
    regfile rf
(.clk(clk), .we3(RegWrite), .ra1(RA1), .ra2(RA2), .wa3(temp1), .wd3(temp2), .r15(
PCPlus8), .rd1(SrcA), .rd2(WriteData) );
    mux2 #(32) resmux(.d0(ALUResult), .d1(ReadData), .s(MemtoReg), .y(Result));
    extend ext( .Instr(Instr[23:0]), .ImmSrc(ImmSrc), .ExtImm(ExtImm) );


    //ALU logic
    mux2 #(32) srcbmux(.d0(WriteData), .d1(ExtImm), .s(ALUSrc), .y(SrcB));
    alu alu
(.SrcA(SrcA), .SrcB(SrcB), .ALUControl(ALUControl), .ALUResult(ALUResult), .ALUFl
ags(ALUFlags) );

endmodule

//ALU
module alu(input [31:0] SrcA, SrcB,
    input [3:0] ALUControl,
    output [31:0] ALUResult,
    output [3:0] ALUFlags);
```

```
    wire [32:0] condinvb;
    wire [32:0] sum;
    wire ofw1;
    wire ofw0;
    wire Cout;
    wire add_op;        // if true, this is an add operation
    wire sub_op;        // if true, this is a sub operation
    wire add_sub_op;    // if true, this is an add or sub operation

    // set internal signals
    assign add_op =
        (ALUControl == 4'b0100) || (ALUControl == 4'b1011);
    assign sub_op =
        (ALUControl == 4'b0010) || (ALUControl == 4'b1010);
    assign add_sub_op = add_op | sub_op;
    assign condinvb =
        (sub_op == 1'b1) ? ~{1'b0,SrcB} : {1'b0,SrcB};
    assign sum = {1'b0,SrcA} + condinvb + sub_op;
    assign Cout = sum[32];
    assign ofw1 = ~(SrcA[31] ^ SrcB[31] ^ sub_op);
    assign ofw0 = SrcA[31] ^ SrcB[31] ^ add_sub_op;

    assign ALUResult =
        (ALUControl == 4'b0000) ? SrcA & SrcB : // AND
        (ALUControl == 4'b1100) ? SrcA | SrcB : // OR
        (ALUControl == 4'b0001) ? SrcA ^ SrcB : // EOR
        (ALUControl == 4'b1110) ? SrcA & ~SrcB :// BIC
        (ALUControl == 4'b1111) ? ~SrcB :       // MVN
        (ALUControl == 4'b1000) ? SrcA & SrcB : // TST
        (ALUControl == 4'b1001) ? SrcA ^ SrcB : // TEQ//(add_sub_op == 1'b1 &&
ALUControl != 4'b1010 && ALUControl != 4'b1011) ? sum[31:0] :
        (add_sub_op == 1'b1) ? sum[31:0] :
          // sub is 2's comp. same adder is used for add and sub
        32'b0; // unrecognized ALU op code

    //Set Flags
    assign ALUFlags[3] =
        (ALUControl == 4'b1000) ? ((SrcA & SrcB) >> 31) : // TST
        (ALUControl == 4'b1001) ? ((SrcA ^ SrcB) >> 31) : // TEQ
        sum[31]; //N negative

    assign ALUFlags[2] =
        (ALUControl == 4'b1000) ? ((SrcA & SrcB) == 32'b0) :
        // TST
        (ALUControl == 4'b1001) ? ((SrcA ^ SrcB) == 32'b0) :
```

```
        // TEQ
        (sum[31:0] == 32'b0); //Z zero

    assign ALUFlags[1] = Cout & add_sub_op; // C carry-out

    assign ALUFlags[0]= ofw1 & ofw0 & add_sub_op; //V overflow

    // set final result


endmodule

//controller
module controller(input clk,reset,
    input [31:12] Instr,
    input [3:0] ALUFlags,
    output [1:0] RegSrc,
    output RegWrite,
    output [1:0] ImmSrc,
    output ALUSrc,
    output [3:0] ALUControl,
    output MemWrite, MemtoReg,
    output PCSrc);

    wire [1:0] FlagW;
    wire PCS, RegW, MemW;

    decoder
dec( .Op(Instr[27:26]), .Funct(Instr[25:20]), .Rd(Instr[15:12]), .FlagW(FlagW), .
PCS(PCS), .RegW(RegW), .MemW(MemW), .MemtoReg(MemtoReg), .ALUSrc(ALUSrc), .ImmSrc
(ImmSrc), .RegSrc(RegSrc), .ALUControl(ALUControl) );
    condlogic
cl( .clk(clk), .reset(reset), .Cond(Instr[31:28]), .ALUFlags(ALUFlags), .FlagW(Fl
agW), .PCS(PCS), .RegW(RegW), .MemW(MemW), .PCSrc(PCSrc), .RegWrite(RegWrite), .M
emWrite(MemWrite) );

endmodule

//condlogic
module condlogic(input clk, reset,
    input  [3:0] Cond,
    input  [3:0] ALUFlags,
    input  [1:0] FlagW,
    input  PCS, RegW, MemW,
    output  PCSrc, RegWrite, MemWrite); // will be wires in tb
```

```
    wire [1:0] FlagWrite;
    wire [3:0] Flags;
    //reg [3:0] Flags_r;
    //assign Flags = Flags_r;
    wire CondEx;

    flopenr #(2)
flagreg1( .clk(clk), .reset(reset), .en(FlagWrite[1]),.d(ALUFlags[3:2]), .q(Flags
[3:2]) );
    flopenr #(2)
flagreg0( .clk(clk), .reset(reset), .en(FlagWrite[0]),.d(ALUFlags[1:0]), .q(Flags
[1:0]) );

    // write controls are conditional
    condcheck cc(.Cond(Cond), .Flags(Flags), .CondEx(CondEx) );

    assign FlagWrite = FlagW & {2{CondEx}};
    assign RegWrite = RegW & CondEx;
    assign MemWrite = MemW & CondEx;
    assign PCSrc = PCS & CondEx;

endmodule

//condcheck
module condcheck(input  [3:0] Cond,
    input [3:0] Flags,
    output reg CondEx);

    wire neg, zero, carry, overflow, ge;
    assign {neg, zero, carry, overflow} = Flags;
    assign ge = (neg == overflow);

    always @(*)  begin
        case(Cond)
            4'b0000: CondEx = zero;             // EQ
            4'b0001: CondEx = ~zero;            // NE
            4'b0010: CondEx = carry;            // CS
            4'b0011: CondEx = ~carry;           // CC
            4'b0100: CondEx = neg;              // MI
            4'b0101: CondEx = ~neg;             // PL
            4'b0110: CondEx = overflow;         // VS
            4'b0111: CondEx = ~overflow;        // VC
            4'b1000: CondEx = carry & ~zero;    // HI
            4'b1001: CondEx = ~(carry & ~zero); // LS
```

```
            4'b1010: CondEx = ge;               // GE
            4'b1011: CondEx = ~ge;              // LT
            4'b1100: CondEx = ~zero & ge;       // GT
            4'b1101: CondEx = ~(~zero & ge);    // LE
            4'b1110: CondEx = 1'b1;             // Always
            default: CondEx = 1'bx;             // undefined
        endcase
    end
endmodule

//decoder
module decoder(input  [1:0] Op,
    input  [5:0] Funct,
    input  [3:0] Rd,
    output [1:0] FlagW,
    output PCS, RegW, MemW,
    output MemtoReg, ALUSrc,
    output [1:0] ImmSrc, RegSrc,
    output [3:0] ALUControl);

    wire [9:0] controls;
    wire Branch, ALUOp;
    wire [1:0] FlagW_;
    assign FlagW = FlagW_;
    wire temp;

    // Main Decoder
    assign controls =
        (Op == 2'b00) ? ((Funct[5]) ? 10'b0000101001 : 10'b0000001001) :
        // Data-processing immediate operand : Data-processing register operand
        (Op == 2'b01) ? ((Funct[0]) ? 10'b0001111000 : 10'b1001110100) :
        // LDR : STR
        //(Op == 2'b10) ? 10'b0110100010 : 10'b0;// B : undefined
        (Op == 2'b10) ? ((Funct[0]) ? 10'b1110101010 : 10'b0110100010 ) : // BL :
B
        10'b0;
    assign {RegSrc, ImmSrc, ALUSrc, MemtoReg, temp, MemW, Branch, ALUOp} =
controls;
    assign RegW = (Funct[4:1] == 4'b1000 || Funct[4:1] == 4'b1001 || Funct[4:1]
== 4'b1010 || Funct[4:1] == 4'b1011 ) ? 0 : temp;

    // ALU Decoder
    wire add_sub_op;
    assign add_sub_op =
        (Funct[4:1] == 4'b0100 ||
```

```
        Funct[4:1] == 4'b0010 ||
        Funct[4:1] == 4'b1011 ||
        Funct[4:1] == 4'b1010);
    // this is a summation instruction
    assign ALUControl = (ALUOp == 1'b1) ? Funct[4:1] : 4'b0100;
    // if ALU instruction, set ALU control signals.
    // Otherwise perform addition for Branch and memory target address
calculation
    assign FlagW_[1] = (ALUOp & Funct[0]);
    // update ALU flags on ALU instruction completion if ALU op and S bit is set
    assign FlagW_[0] = FlagW_[1] & add_sub_op;
    // ALU C (carry) and V (overflow) flags are only updated if summation is
being performed

    // PC Logic
    assign PCS = ((Rd == 4'b1111) & RegW) | Branch;

endmodule

//__GENERIC BUILDING BLOCKS__

//mux2
module mux2 #(parameter WIDTH = 8)
    (input [WIDTH-1:0] d0,d1,
    input s,
    output [WIDTH-1:0] y);

    assign y= s ? d1 : d0;
endmodule

//flopenr
module flopenr #(parameter WIDTH = 8)
    (input clk,reset,en,
    input [WIDTH-1:0] d,
    output reg [WIDTH -1:0] q);

    always @(posedge clk, posedge reset)begin
        if (reset) q<=0;
        else if (en) q<=d;
    end
endmodule

//extend
module extend(input [23:0] Instr,
    input [1:0] ImmSrc,
```

```
    output reg [31:0] ExtImm);

    always @(*) begin
        case(ImmSrc)
            2'b00: ExtImm={24'b0,Instr[7:0]};
            //8bit unsigned imm
            2'b01: ExtImm={20'b0,Instr[11:0]};
            //12 bit unsignd imm
            2'b10: ExtImm={ {6{Instr[23]}}, Instr[23:0], 2'b00 };
            //24 bit 2s comp shifted branch
            default: ExtImm=32'bx;
        endcase
    end
endmodule

//adder
module adder #(parameter WIDTH=8)
    (input [WIDTH-1:0] a,b,
    output [WIDTH-1:0] y);
    assign y=a+b;
endmodule

// regfile
module regfile( input clk,
    input we3,
    input  [3:0] ra1, ra2, wa3,
    input  [31:0] wd3, r15,
    output [31:0] rd1, rd2);

    reg [31:0] rf[14:0];

    //3 ported register file
    //read 2 ports combinationally
    //write 3rd port on rising clock edge
    //register 15 reads PC+8 instead

    always @(posedge clk) begin
        if (we3) rf[wa3]<=wd3;
    end

    assign rd1=(ra1==4'b1111) ? r15 : rf[ra1];
    assign rd2=(ra2==4'b1111) ? r15 : rf[ra2];

endmodule
```

ARM_system.v

```
//Top module. Includes setting of clock
module ARM_system(input clk, reset,
    output [31:0] PC,Instr,
    output [31:0] WriteData, DataAdr,
    output MemWrite);

    reg [30:0]counter;
    reg CLK1;

    // clock divider
    always @(posedge clk) begin
        counter=counter+1;
        if (counter==25000000) begin        // for FPGA
        //if (counter==2) begin                 // for SIM
            counter=0;
            CLK1=!CLK1;
        end
    end

    wire [31:0]  ReadData;

    // instantiate processor and memories
    armv4
cpu( .clk(CLK1), .reset(reset), .PC(PC), .Instr(Instr), .MemWrite(MemWrite), .ALU
Result(DataAdr),.WriteData(WriteData), .ReadData(ReadData) );
    imem imem( .clk(CLK1), .a(PC), .rd(Instr) );
    dmem
dmem( .clk(CLK1), .we(MemWrite), .a(DataAdr), .wd(WriteData), .rd(ReadData) );

endmodule

//Dmem module
module dmem(input clk, we,
    input [31:0] a, wd,
    output [31:0] rd);

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

    assign rd = RAM[a[31:2]]; // word aligned

    always @(posedge clk) begin
      if (we) RAM[a[31:2]] <= wd;
```

```
   end
endmodule

//Imem module
module imem(input clk,
    input [31:0] a,
    output reg [31:0] rd);

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

    initial
        //$readmemh("memfile.txt",RAM);
        $readmemh("All_Inst (copy).txt", RAM);
    always @(negedge clk) begin
        rd = RAM[a[31:2]]; // word aligned
   end
endmodule
```

All_Inst(copy).txt, ARM_system.v, and 7seg.v are not modified. These files are provided.

Additional Function

- **EOR**

For EOR instruction, students used SrcA ^ SrcB to get the result. The result of this operation need to be saved. So students set the ALUResult and the function will take care of flags.

```
(ALUControl == 4'b0001) ? SrcA ^ SrcB : // EOR
```

Whent the ALUcontrol equals to 0001, the EOR will be operated.

- **BIC**

According to the provided material, BIC instruction makes the SrcA & ~SrcB. This instruction is similar to the EOR, the result need to be kept. In case of ALUControl = 1110, the instruction will operate.

```
(ALUControl == 4'b1110) ? SrcA & ~SrcB :// BIC
```

- **MVN**

This instruction will invert the number of SrcB. Very similar to the previous 2 instruction, when ALUControl == 1111, SrcB will be inverted.

```
(ALUControl == 4'b1111) ? ~SrcB :       // MVN
```

- **Decoder Change**

For the instructions like TST, TEQ, CMP, CMN, they need to use the memory of the CPU. The result of these instructions will be saved to the register file. To present the written data , we need to set RegW in the decoder to 0.

Code modification:

```
assign {RegSrc, ImmSrc, ALUSrc, MemtoReg, temp, MemW, Branch, ALUOp} = controls;
assign RegW = (Funct[4:1] == 4'b1000 || Funct[4:1] == 4'b1001 || Funct[4:1] ==
4'b1010 || Funct[4:1] == 4'b1011 ) ? 0 : temp;
```

- **TST**

TST instruction is a test instruction. This means that the result will not be saved. However, flags need to be saved which means the result of SrcA & SrcB will determine the flag change.

Since it is a bitwise operation, only N and Z flags need to be updated. For the sign of the result, it needs to shift the 31 bit of SrcA & SrcB right, because it is a 2's complement numbers. This value is the MSB, the machine should determine which flag to update. MSB is 1, N flag to 1. MSB is 0, N flag to 0.

Then it needs to update the Z flag. When the result is 0, Z flag will be updated.

Lastly, RegW will be set to 0 to prevent writing to register file.

- **TEQ**

TEQ check the equivalence of SrcA and SrcB. It is the same as TST that no need to save result to register file. The only thing need to change is N and Z flags. RegW should be 0 to prevent writing to register file

The following code is for TEQ and TST:

```
assign ALUResult =
        (ALUControl == 4'b0000) ? SrcA & SrcB : // AND
        (ALUControl == 4'b1100) ? SrcA | SrcB : // OR
        (ALUControl == 4'b0001) ? SrcA ^ SrcB : // EOR
        (ALUControl == 4'b1110) ? SrcA & ~SrcB :// BIC
        (ALUControl == 4'b1111) ? ~SrcB :       // MVN
        (ALUControl == 4'b1000) ? SrcA & SrcB : // TST
        (ALUControl == 4'b1001) ? SrcA ^ SrcB : // TEQ//(add_sub_op == 1'b1 &&
ALUControl != 4'b1010 && ALUControl != 4'b1011) ? sum[31:0] :
        (add_sub_op == 1'b1) ? sum[31:0] :
          // sub is 2's comp. same adder is used for add and sub
        32'b0; // unrecognized ALU op code
```

```
//Set Flags
assign ALUFlags[3] =
    (ALUControl == 4'b1000) ? ((SrcA & SrcB) >> 31) : // TST
    (ALUControl == 4'b1001) ? ((SrcA ^ SrcB) >> 31) : // TEQ
    sum[31]; //N negative

assign ALUFlags[2] =
    (ALUControl == 4'b1000) ? ((SrcA & SrcB) == 32'b0) :
    // TST
    (ALUControl == 4'b1001) ? ((SrcA ^ SrcB) == 32'b0) :
    // TEQ
    (sum[31:0] == 32'b0); //Z zero
```

- **CMP and CMN**

These 2 test instructions involved plus and minus operation. This means CMP and CMN need to update the flags and check the result. Overflows and carriers need to be considered. To make the ALU know a add or subtract is operated. Student set add_sub_op in the decoder. After computation. The result is stored in ALUResult. After updating the flags, RegW will be set to 0 to prevent writing in the register.

Here is the Code for implement these instructions:

```
wire add_sub_op;
assign add_sub_op =
    (Funct[4:1] == 4'b0100 ||
     Funct[4:1] == 4'b0010 ||
     Funct[4:1] == 4'b1011 ||
     Funct[4:1] == 4'b1010);
```

- **CMP and CMN**

Bl instructions needs to write PCPLus4 to the R14 so that the program can go back to where the branch happens. Student add 2 MUXs to implement this function.

BLmuxA, can select the address of the R14, which is 1110, when RegSrc[0] and RegSrc[1] are high. This happens when the BL instruction is called. The second MUX, BLmuxB, can select the PCPlus4 when both variables are high. These 2 MUX will not interrupt other instructions to be operated.

Here it needs to write to the register file, so that the RegW should be high.

The following part is the change in code:

MUX added for BL:

```
 mux2 #(4) BLmuxA(.d0(Instr[15:12]), .d1(4'b1110), .s(RegSrc[0] &
RegSrc[1]), .y(temp1)); // Additional mux2 to set BL
    mux2 #(32) BLmuxB (.d0(Result), .d1(PCPlus4), .s(RegSrc[0] &
RegSrc[1]), .y(temp2));
```

BL change in Decoder:

```
 // Main Decoder
    assign controls =
        (Op == 2'b00) ? ((Funct[5]) ? 10'b0000101001 : 10'b0000001001) :
        // Data-processing immediate operand : Data-processing register operand
        (Op == 2'b01) ? ((Funct[0]) ? 10'b0001111000 : 10'b1001110100) :
        // LDR : STR
        //(Op == 2'b10) ? 10'b0110100010 : 10'b0;// B : undefined
        (Op == 2'b10) ? ((Funct[0]) ? 10'b1110101010 : 10'b0110100010 ) : // BL :
B
        10'b0;
    assign {RegSrc, ImmSrc, ALUSrc, MemtoReg, temp, MemW, Branch, ALUOp} =
controls;
    assign RegW = (Funct[4:1] == 4'b1000 || Funct[4:1] == 4'b1001 || Funct[4:1]
== 4'b1010 || Funct[4:1] == 4'b1011 ) ? 0 : temp;
```

Simulation Result

In this part, student will show the simulation result from the Quaturs simulation waveform.

- **EOR**

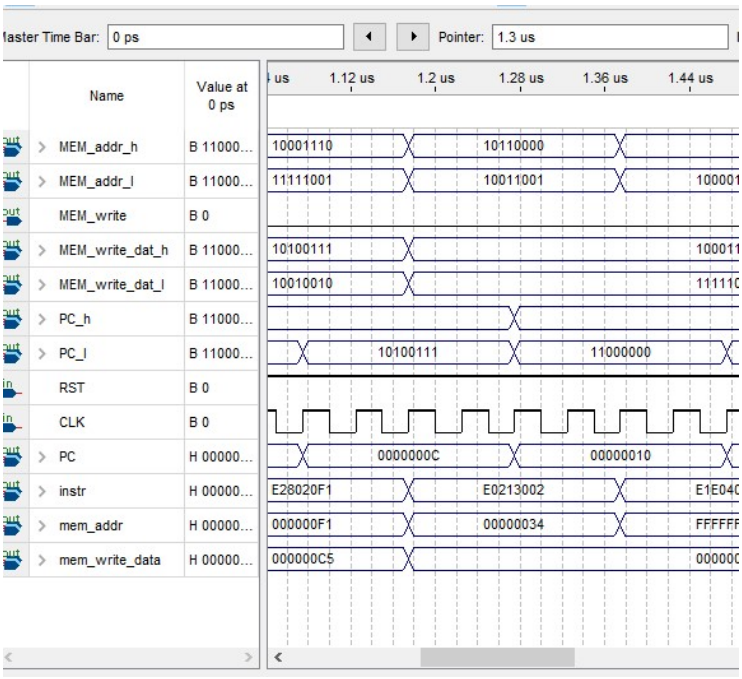


Figure: EOR simulation result

The PC for EOR is 0x0C, returned value is 0x34. The CPU worked as expectation.

- **BIC**

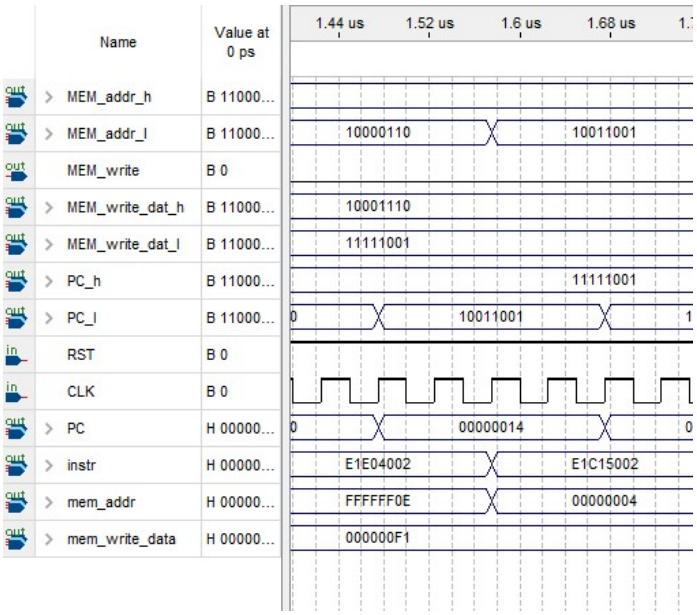


Figure: BIC simulation result

From the Provided file, we know the PC of BIC is 0x14, here memory change to 0x04, use the register R5. The result of this instruction is 0x04. It works as expected.

- **MVN**

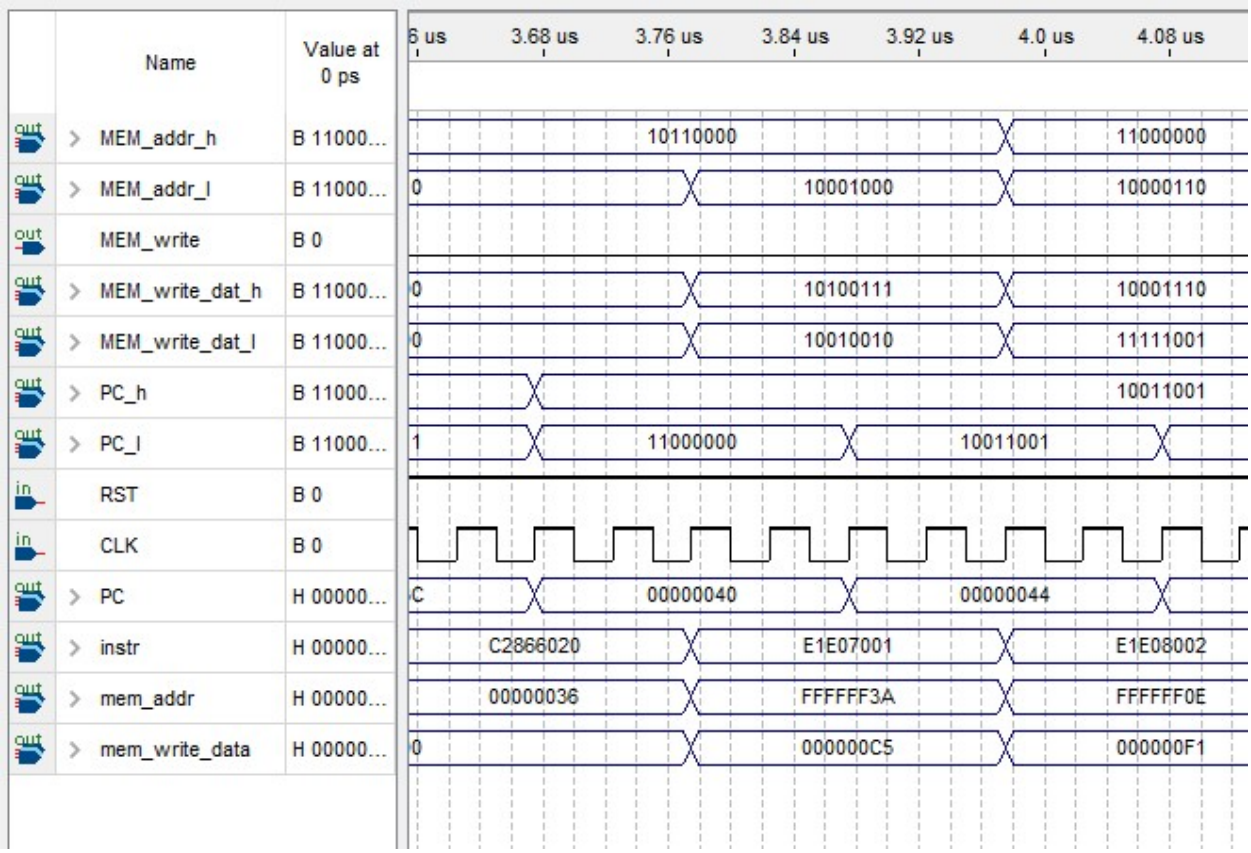


Figure: MVN simulation result

For MVN instruction, PC is 0x40, the return value should be 0xFFFFFF3A. In the diagram, after 0x40, memory address changed to 0xFFFFFF3A, MVN works as expected, the result is saved in R7.

It is the same as PC = 0x44, the mem_addr = 0xFFFFFF0E, R8.

- **TST**

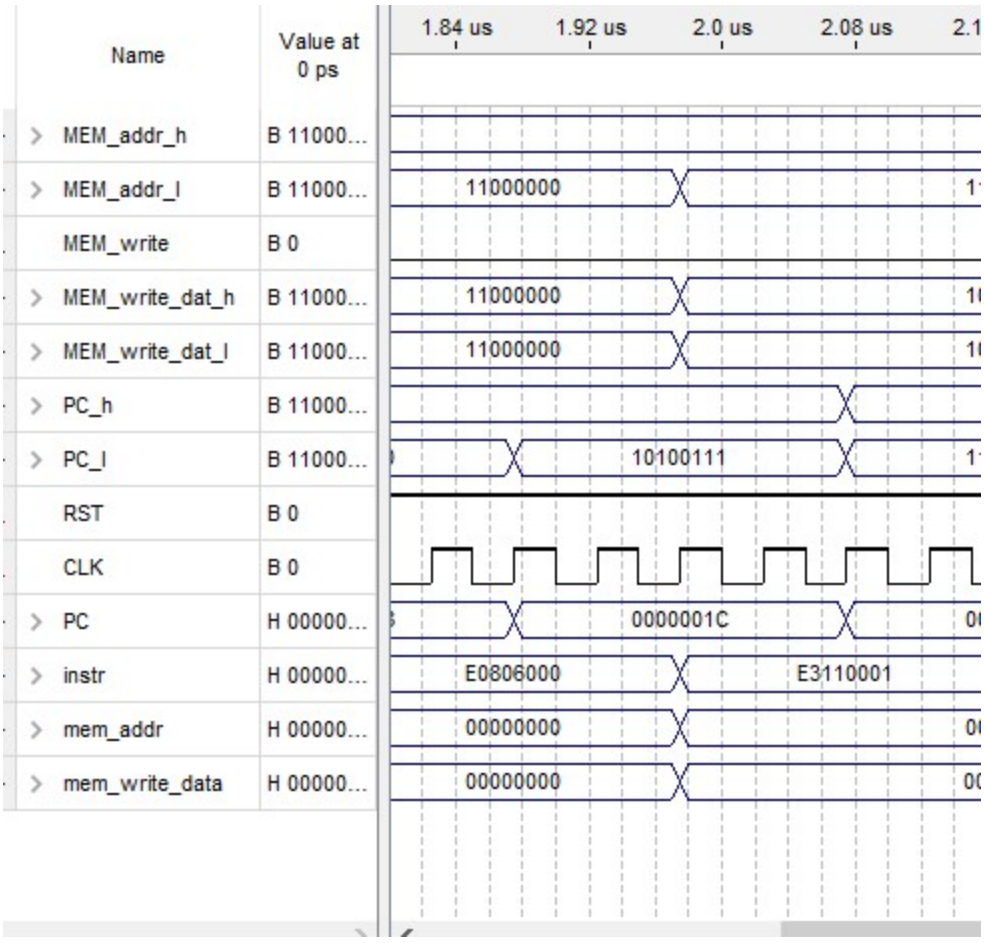


Figure: TST simulation result

When PC = 0x1C, mem_addr is 1, which means TST get a correct result from ALU.

- **TEQ**

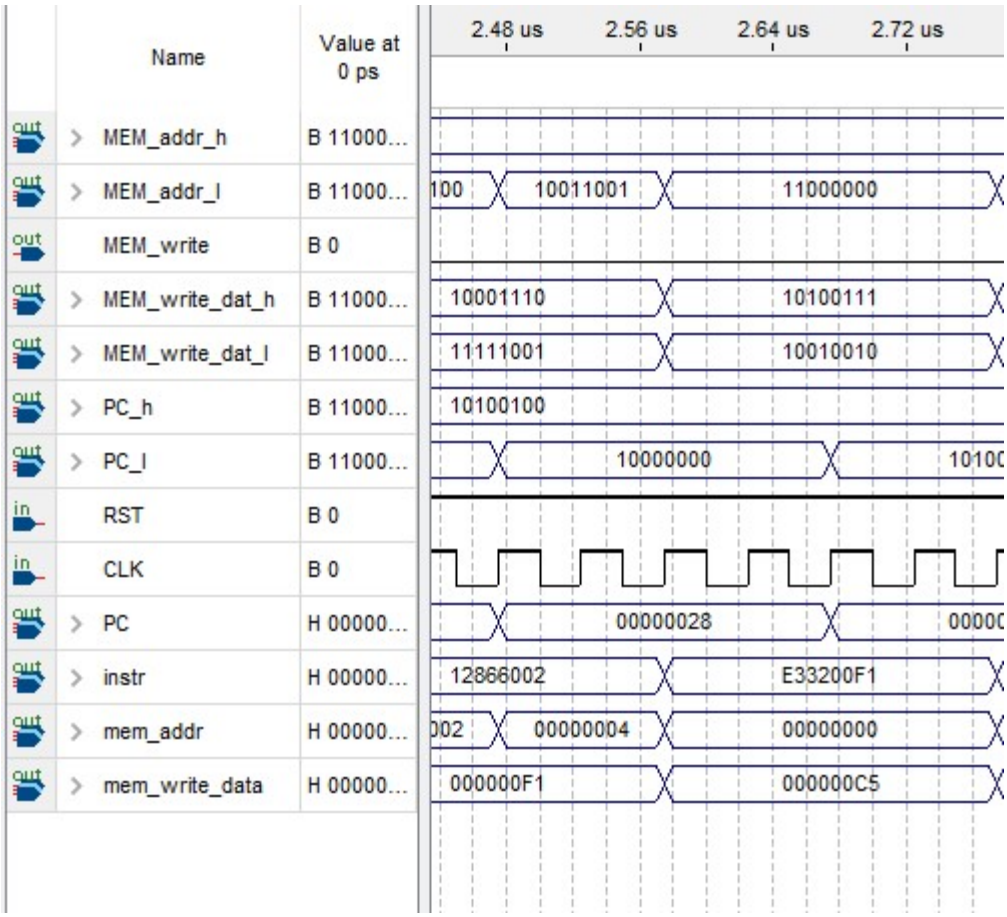


Figure: TEQ simulation result

TEQ is runned when PC = 0x28. The displayed value should be 0. The simulation result is the same as expected.

- **CMP**

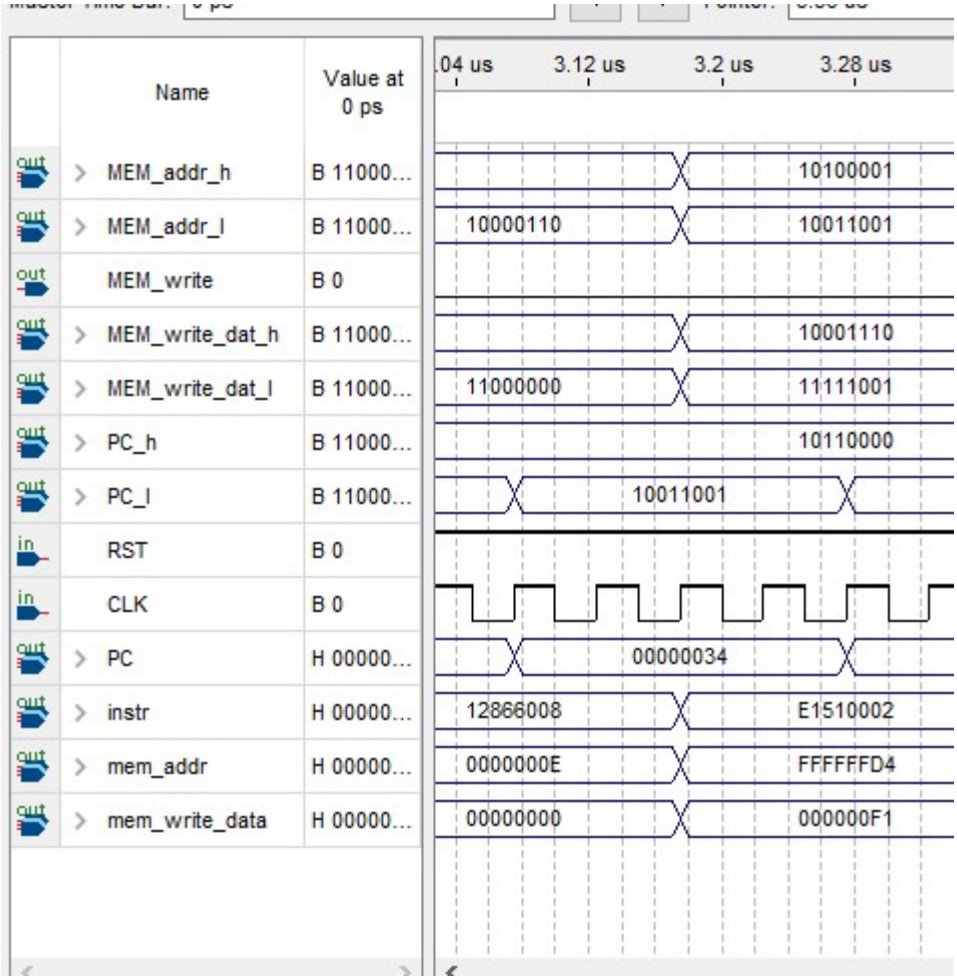


Figure CMP simulation result

When PC = 0x34, CMP is operated. The displayed value should be D4 according to the provided running result. Here the mem_addr= 0xFFFFFFD4, which means the address is D4 and instruction worked as expected.

- **CMN**

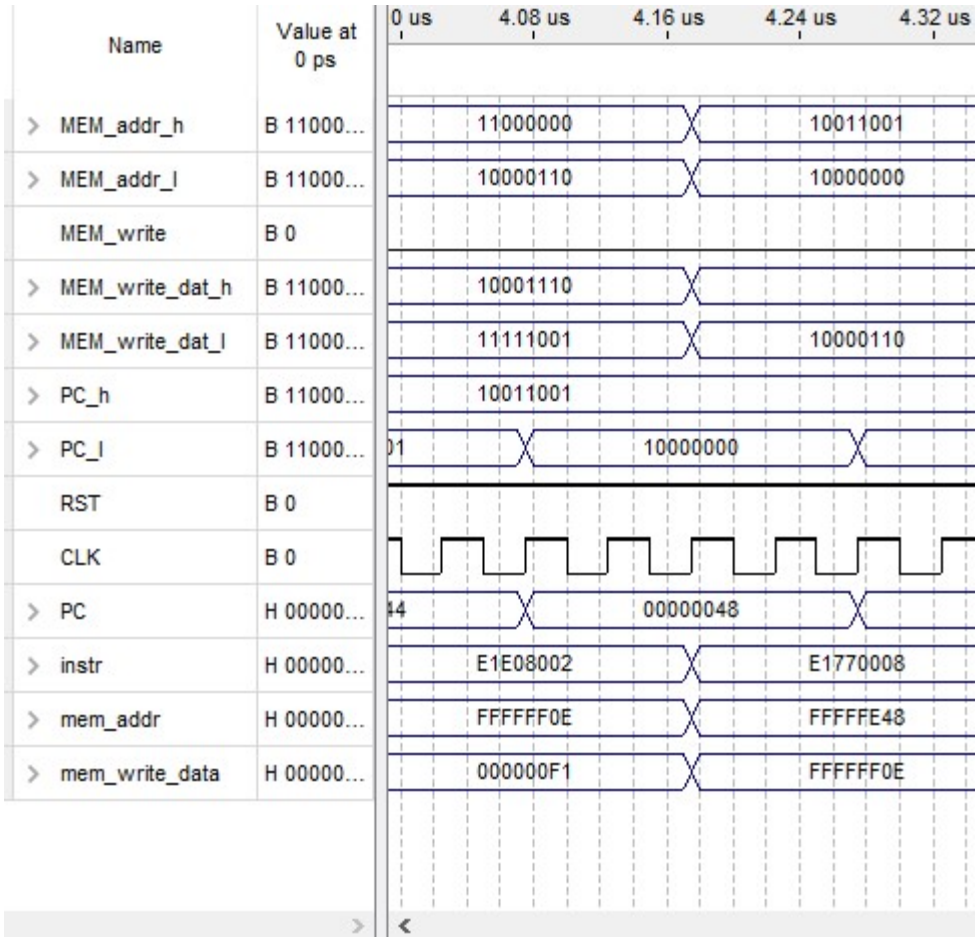


Figure Simulation of CMN

When PC = 0x48, CMN is operated. The displayed value is 48. Here the mem_add = 0xFFFFFE48. The result is the same as expected.

- **BL**

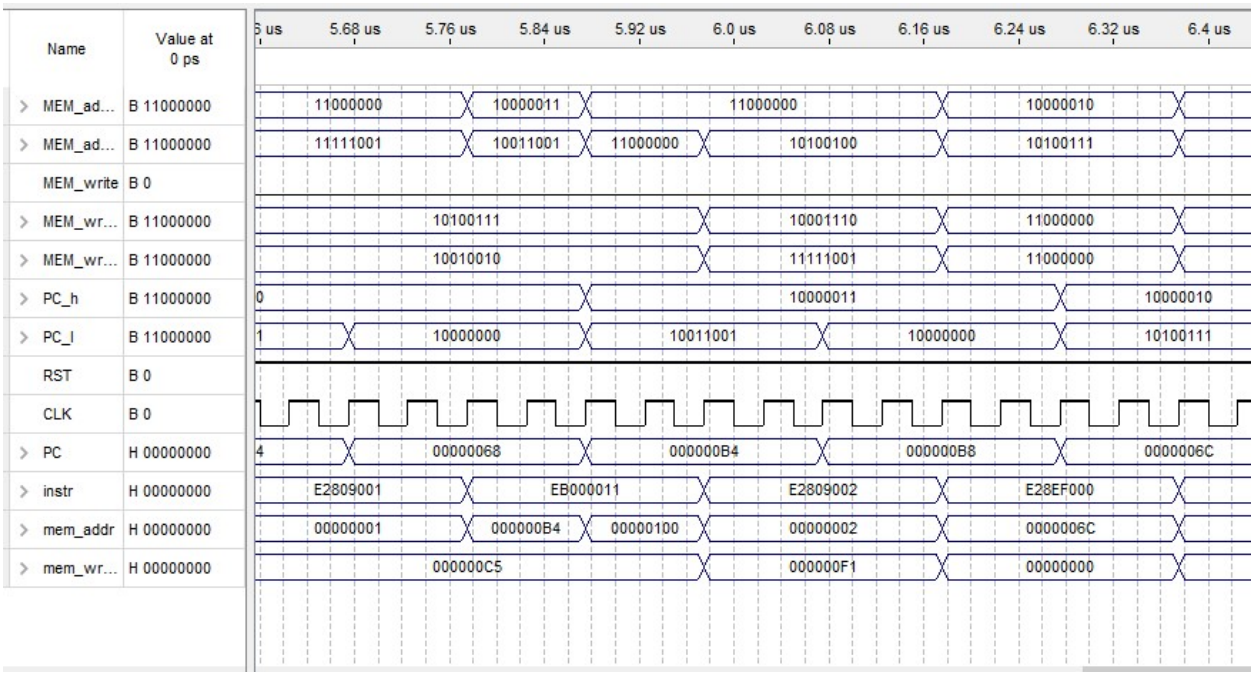


Figure: BL simulation result

After the BL executed, the program will return R14, then jump to command PC = 0xB4. Here the 0x6C will be saed to R14 so that it can be read by 0xB8. Then in instruction 0xB8, 0x6C will be loaded and jump to PC = 0x6C.

- **PC in the green box**

The following diagram includes the running result in the green box of the correct running result. From PC = 0x7C to 0xA8. CPU should generate correct mem_addr. For other instructions, the CPU should generate correct mem_write_data.

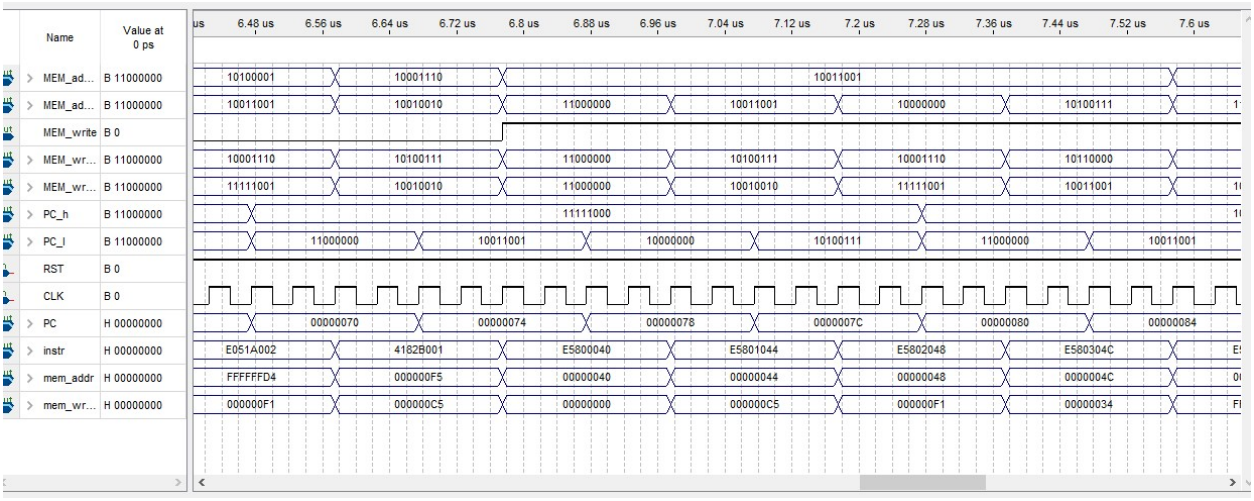


Figure: PC from 0x70 to 0x84

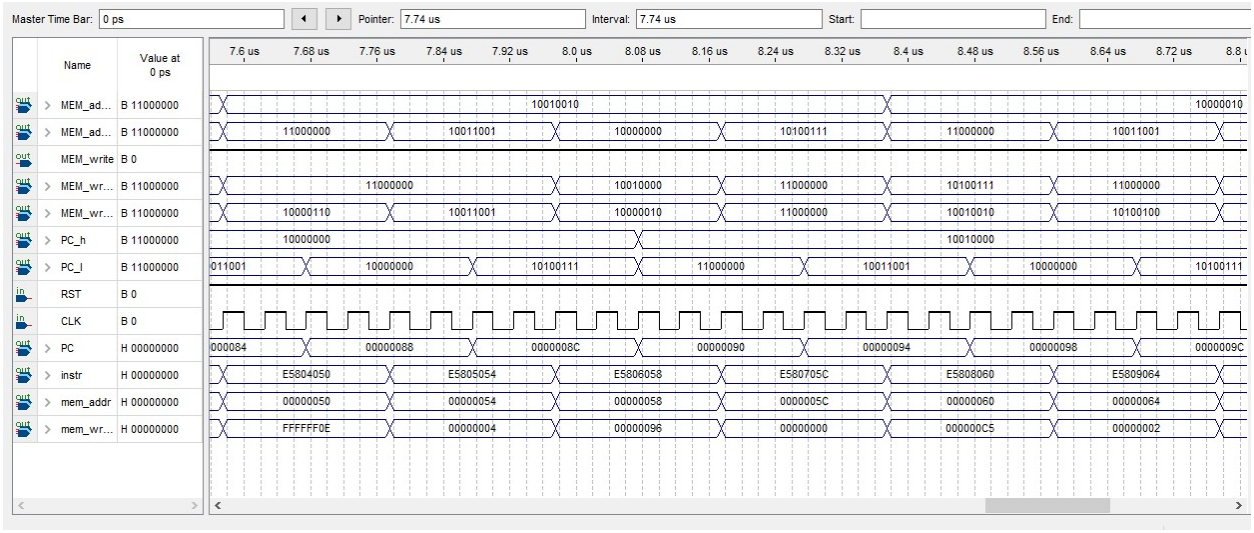


Figure: PC from 0x84 to 0x98

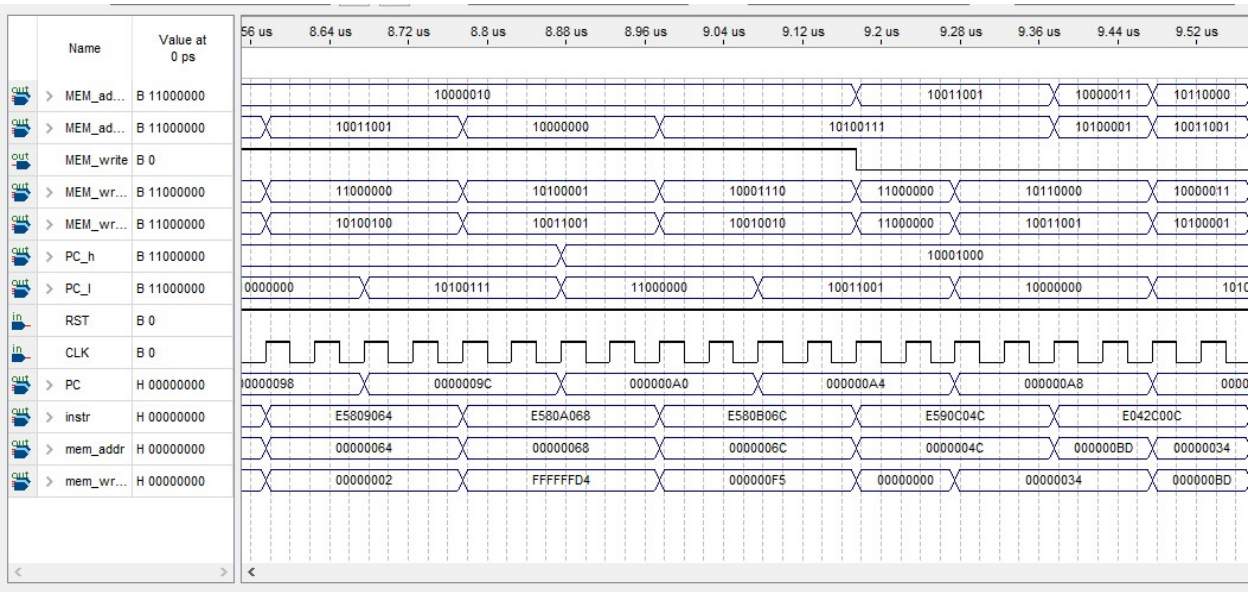


Figure: PC from 0x98 to 0xA8

FPGA Functionality

- Work on FPGA

To make the design worked on the FPGA, the clock source of the CPU need to be modified. The main idea is using the clock source on the FPGA board. The CLK will read the PIN_N5. This makes the CPU read the 10M Hz clock provided by the board.

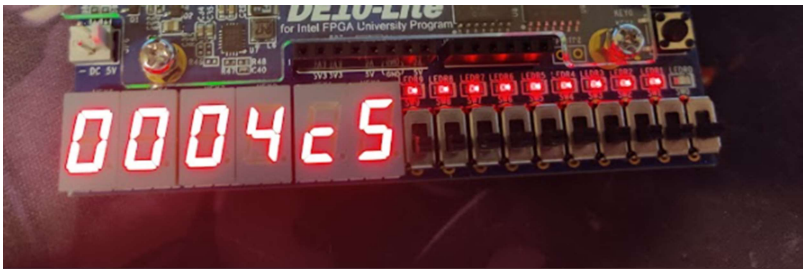
Besides, to make the result can be checked by human eyes, the overflow of the clock should be changed. The change was made in the ARM_system.v. The overflow is changed to 50000000 when operating on the FPGA board. So that the instruction will change every 1 second.

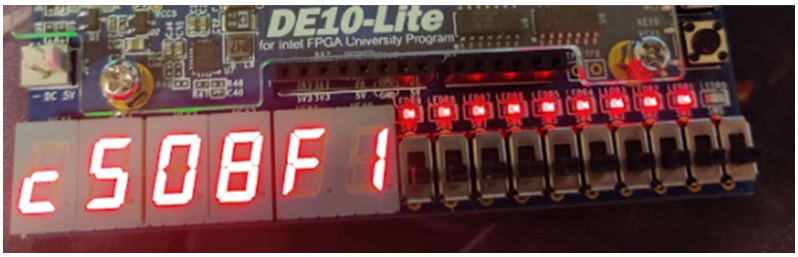
Change in ARM_system.v:

```
// clock divider
always @(posedge clk) begin
    counter=counter+1;
    //if (counter==25000000) begin      // for FPGA
    if (counter==2) begin                   // for SIM
        counter=0;
        CLK1=!CLK1;
    end
end
```

- Simulation results

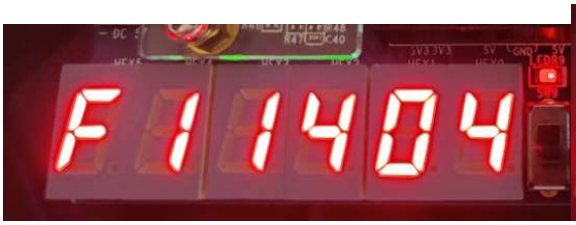
Here student will only include the Hex display of the board




DE10-Lite
for Intel FPGA University Program
c508F1


F10c34


F1100E


F11404


001800


005096


005400


0060c5


c56401


c570F5


F17c48


34804c


0EB450


048854


968c58


00905c


c59460


029864


d49c68


F5A06c


34A8bd

Verification

The verification of the design is not difficult in this lab, since there is a correct running result provided in the lab, so student can check the Quaturs simulation with the correct result. And each parameter is included in the simulation waveform.

All the Verilog can be downloaded to the FPGA board, the board can worked as expected. The running result is the same as the provided result. So the design is complete.

Conclusion

In this lab student implement a ARM v4 CPU. FPGA can verify the design and check if each instruction can work as expected. It is a very complex design of a ARM CPU. However, the FPGA can still simulate it. The cost of verification of the design is small.