Single Cycle Processor

冯硕 11911736

This is the lab report of Single Cycle Processor. In the previous labs, I have built up some modules for this CPU core and the corresponding test benches. Below will shows my ARM CPU core overview, instruction and data flows, simulation results and the source codes.

Analysis Report

The components and temporary signals for connection are shown below. The over all structure of Single-Cycle ARM Processor is shown in Fig. 1.

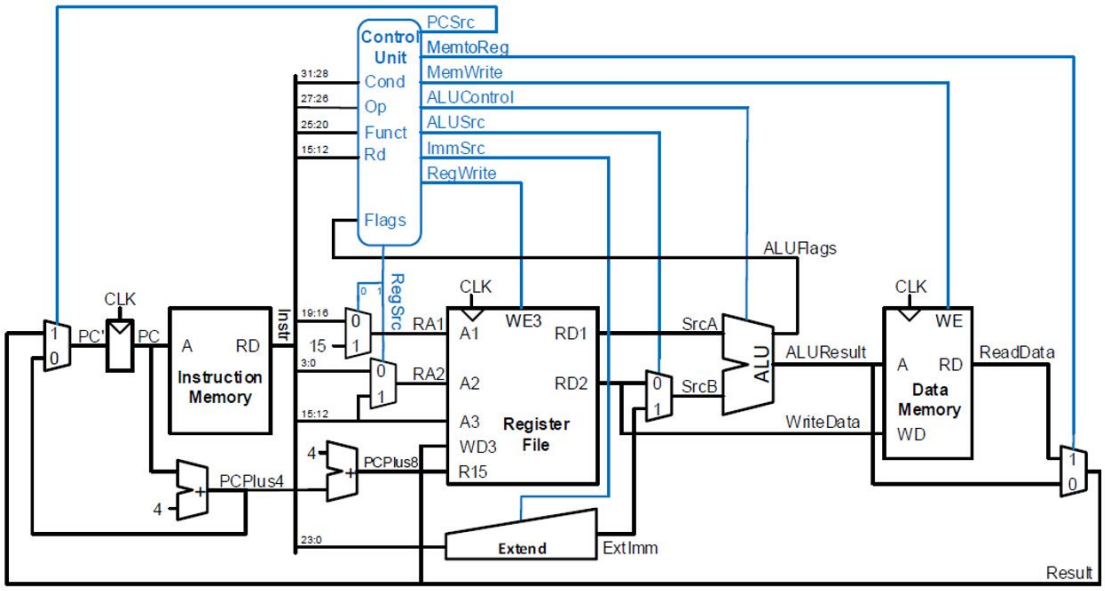

Fig. 1: Single-Cycle ARM Processor with control

Modules in CPU (ARMcore_top.v)

Module	Instance	Description
ProgrammCounter.v	PC0	gives the instruction address
instr_mem.v	InstructionMem0	only need 7 bit to specify the address, capacity is 128 words
ControlUnit.v	ControlUnit0	gives the control signals: **PCsrc**, **MemtoReg**, **MemWrite**, **ALUControl**, **ALUSrc**, **ImmSrc**, **RegWrite**.
RegisterFile.v	RegisterFile0	2^4 registers, each register stores 32-bit data
Extend.v	Extend0	extends the immediate number to 32-bit **ExtImm**
ALU.v	ALU0	gives the arithmetic logic results **ALUResult**
data_mem.v	DataMem0	gives **ReadData** and can write data in data memory, 128 words capacity, 32-bit for each data

Signals for Connections

In my ARM core CPU, I have set these temporary signals for the connection between the modules listed above. Also, these signals also plays a role in generation of some combinatorial logic or sequential logic connections, one can check in Fig. 1.

Signal	Digit	Description
current_PC	32-bit	from Program Counter, address
PC_Plus_4	32-bit	from Program Counter, address
Instr	32-bit	from Instruction Memory, instruction
PCSrc	1-bit	from Control Unit, decides PC for current instruction
MemtoReg	1-bit	from Control Unit, decides which data comes to **Result** signal
MemWrite	1-bit	from Control Unit, memory write enable
ALUControl	2-bit	from Control Unit, Control what operation is executed by ALU
ALUSrc	1-bit	from Control Unit, decide from Immediate or RF
ImmSrc	2-bit	from Control Unit, extension mode
RegWrite	1-bit	from Control Unit, register file write enable
RegSrc	2-bit	from Control Unit, decide **RA1** and **RA2**
RD1	32-bit	from Register File
RD2	32-bit	from Register File
ExtImm	32-bit	from Extend
ALUFlags	4-bit	from ALU, for Control Unit, used for condition judgment
ALUResult	32-bit	from ALU
ReadData	32-bit	from Data Memory
RA1	4-bit	for RF
RA2	4-bit	for RF
PC_Plus_8	32-bit	**PC** plus 8 byte, address
SrcB	32-bit	for ALU
Result	32-bit	from **ReadData** or **ALUResult**

Elaborate Design of Test

Design of Instructions

The carefully designed instruction flow to test your CPU core (including all functions, all possible cases) is shown below in assembly language. The loop will repeat 10 times based on my instruction initiation. Some details of function is also shown in comments below. This is part of the Instruction memory.

Note: These Instruction Memory contains all the functions that the ALU can execute (AND, ORR, ADD, SUB), as well as all the conditions: EQ, NE, CS/HS, CC/LO, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE.

```
//-----------------------------------------------------------------
// Instruction Memory
```

```
//------------------------------------------------------------------
integer i;
initial begin
            INSTR_MEM[0] = 32'hE2100000; //ANDS R0, R0, #0
            INSTR_MEM[1] = 32'h15901001; //not execute, LDRNE R1, [R0, #1]
            INSTR_MEM[2] = 32'h05901001; //LDREQ R1, [R0, #1]
            INSTR_MEM[3] = 32'h25902002; //not execute, LDRCS R2, [R0, #2]
            INSTR_MEM[4] = 32'h35902002; //LDRCC R2, [R0, #2]
            INSTR_MEM[5] = 32'h45903003; //not execute, LDRMI R3, [R0, #3]
            INSTR_MEM[6] = 32'h55903003; //LDRPL R3, [R0, #3]
            INSTR_MEM[7] = 32'h65904004; //not execute, LDRVS R4, [R0, #4]
            INSTR_MEM[8] = 32'h75904004; //LDRVC R4, [R0, #4]
            INSTR_MEM[9] = 32'h82805009; //not execute, ADDHI R5, R0, #9
            INSTR_MEM[10] = 32'h92805009;//ADDLS R5, R0, #9
        //Loop
            INSTR_MEM[11] = 32'hA0816002; //ADDGE R6, R1, R2
            INSTR_MEM[12] = 32'hB0816002; //not execute, ADDLT R6, R1, R2
            INSTR_MEM[13] = 32'hC0437004; //if (Z==0&N==V), SUBGT R7, R3, R4
            INSTR_MEM[14] = 32'hD0437004; //if (z==1|N!=V), SUBLE R7, R3, R4
            INSTR_MEM[15] = 32'hE0437004; //SUB R7, R3, R4
            INSTR_MEM[16] = 32'hE1861007; //ORR R1, R6, R7
            INSTR_MEM[17] = 32'hE0063007; //AND R3, R6, R7
            INSTR_MEM[18] = 32'hE0468007; //SUB R8, R6, R7
            INSTR_MEM[19] = 32'hE2800001; //ADD R0, R0, #1
            INSTR_MEM[20] = 32'hE5808000; //STR R8, [R0]
            INSTR_MEM[21] = 32'hE2555001; //SUBS R5, R5, #1
            INSTR_MEM[22] = 32'h1AFFFFF3; //BNE LOOP
        //Out of Loop
            INSTR_MEM[23] = 32'hE5909000; //LDR R9, [R0]
            for(i = 24; i < 128; i = i+1) begin
                INSTR_MEM[i] = 32'h0;
            end
end
```

The condition for the loop is according to the value stored in R_5. Each loop R_5 will decrease by 1, and set ALU flags. The BNE instruction will exam whether the value of Z equals 0. In this instruction flow, R_5 is initialized as 9, which means the instructions in the loop will repeat 10 times, let's check whether this is correct in simulation results!

Simulations

The overall simulation results are shown in Fig. 2, which also clarify the signals of Register File and Data Memory. The first line are instruction signals that is corresponding to the assembly language above; next 16 lines combine as Register File; The last several lines refer to part of Data Memory. To check whether it is correct, more analysis will be given in the following pictures.

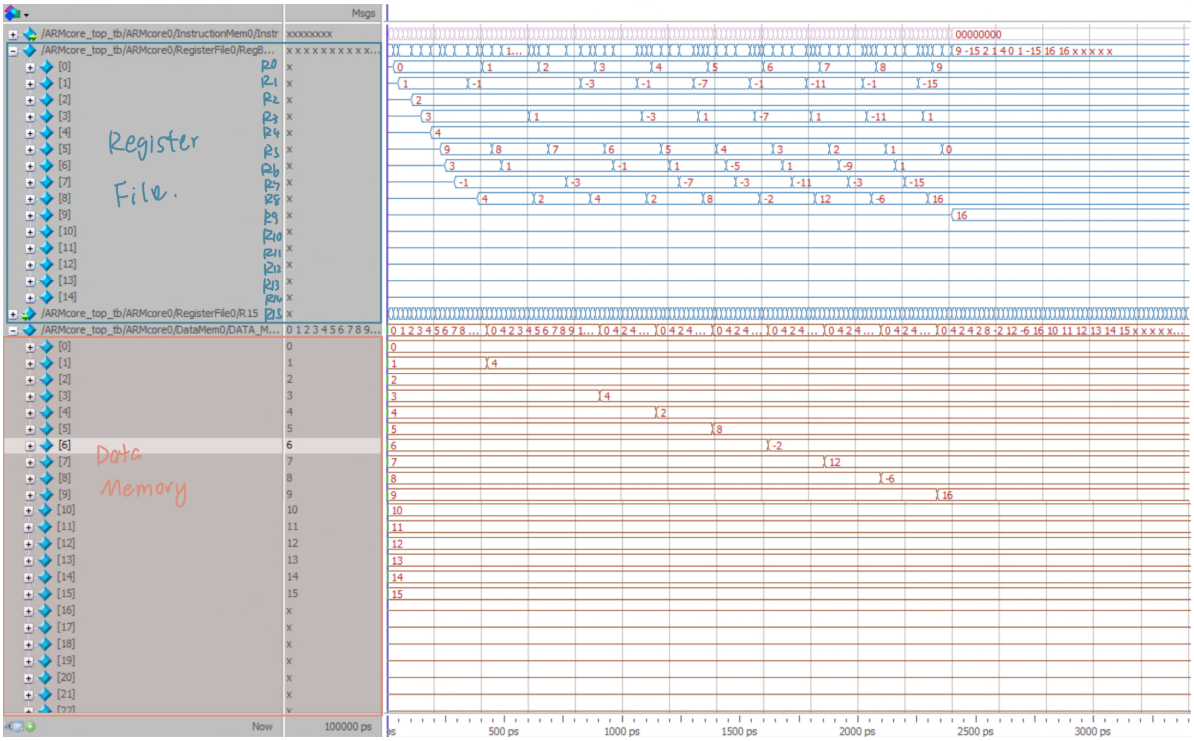

*Fig. 2: Overall simulation results for the designed instruction flow, which is shown in **Design of Instructions** part.*

This result is clear to have an overview of the program, the loop is right 10 times, which satisfy the expectation. More detailed analysis:

Before Loop

Before loop, the simulation results are shown below. Before Loop, the instructions and conditions for execution are:

```
            ; NZCV = 0100
            INSTR_MEM[0] = 32'hE2100000; //ANDS R0, R0, #0, set NZ = 01
            INSTR_MEM[1] = 32'h15901001; //~Z = 0, not execute, LDRNE R1, [R0,
#1]
            INSTR_MEM[2] = 32'h05901001; //Z = 1, LDREQ R1, [R0, #1]
            INSTR_MEM[3] = 32'h25902002; //C = 0, not execute, LDRCS R2, [R0,
#2]
            INSTR_MEM[4] = 32'h35902002; //~C = 1, LDRCC R2, [R0, #2]
            INSTR_MEM[5] = 32'h45903003; //N = 0, not execute, LDRMI R3, [R0,
#3]
            INSTR_MEM[6] = 32'h55903003; //~N = 1, LDRPL R3, [R0, #3]
            INSTR_MEM[7] = 32'h65904004; //V = 0, not execute, LDRVS R4, [R0,
#4]
            INSTR_MEM[8] = 32'h75904004; //~V = 1, LDRVC R4, [R0, #4]
            INSTR_MEM[9] = 32'h82805009; //C&(~Z) = 0, not execute, ADDHI R5,
R0, #9
            INSTR_MEM[10] = 32'h92805009;//(~C)|Z = 1, ADDLS R5, R0, #9
```

We can check the Register File and Data Memory:

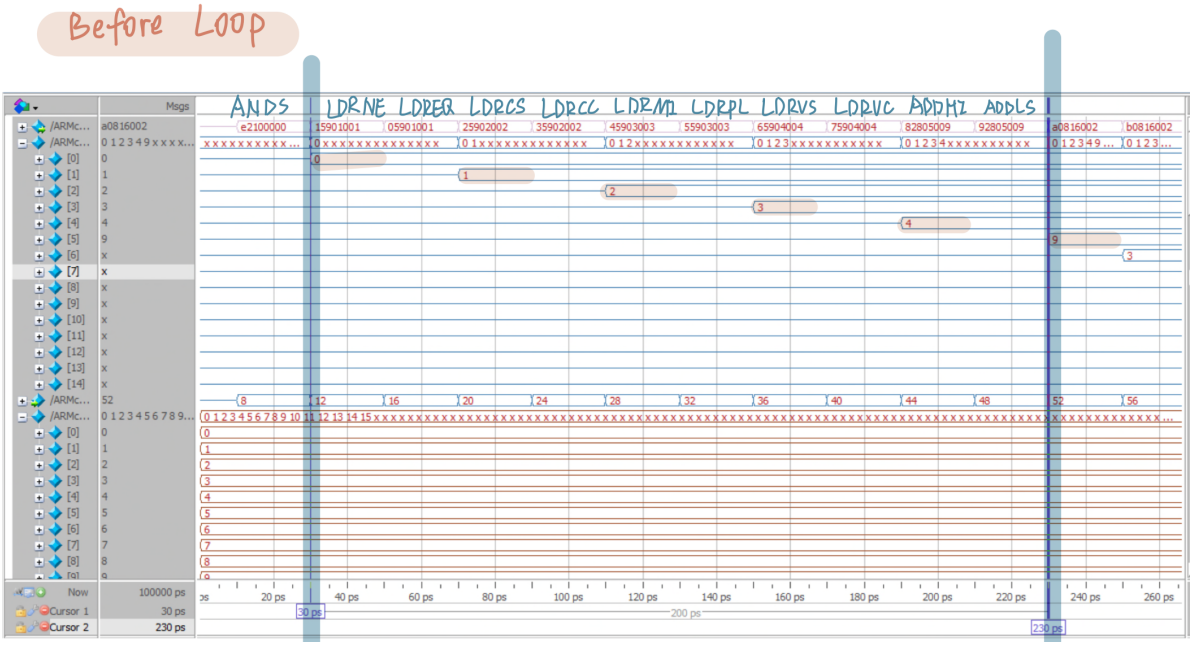

Fig. 3: Instruction flow and the data flow before loop

In Loop

The instructions in Loop are shown below. Whether NZCV satisfied the condition is also given in comments.

```
    ;//Loop
    ;in this loop, NZCV = 0000
            INSTR_MEM[11] = 32'hA0816002; //~N^V = 1, ADDGE R6, R1, R2
            INSTR_MEM[12] = 32'hB0816002; //N^V = 0, not execute, ADDLT R6, R1,
R2
            INSTR_MEM[13] = 32'hC0437004; //~Z&(~N^V) = 1, SUBGT R7, R3, R4
            INSTR_MEM[14] = 32'hD0437004; //Z|(N^V) = 0, not execute, SUBLE R7,
R3, R4
            INSTR_MEM[15] = 32'hE0437004; //SUB R7, R3, R4
            INSTR_MEM[16] = 32'hE1861007; //ORR R1, R6, R7
            INSTR_MEM[17] = 32'hE0063007; //AND R3, R6, R7
            INSTR_MEM[18] = 32'hE0468007; //SUB R8, R6, R7
            INSTR_MEM[19] = 32'hE2800001; //ADD R0, R0, #1
            INSTR_MEM[20] = 32'hE5808000; //STR R8, [R0]
            INSTR_MEM[21] = 32'hE2555001; //SUBS R5, R5, #1
            INSTR_MEM[22] = 32'h1AFFFFF3; //Z = 0, not execute, BNE LOOP
```

The Simulation also results corresponds to the Instruction executions:

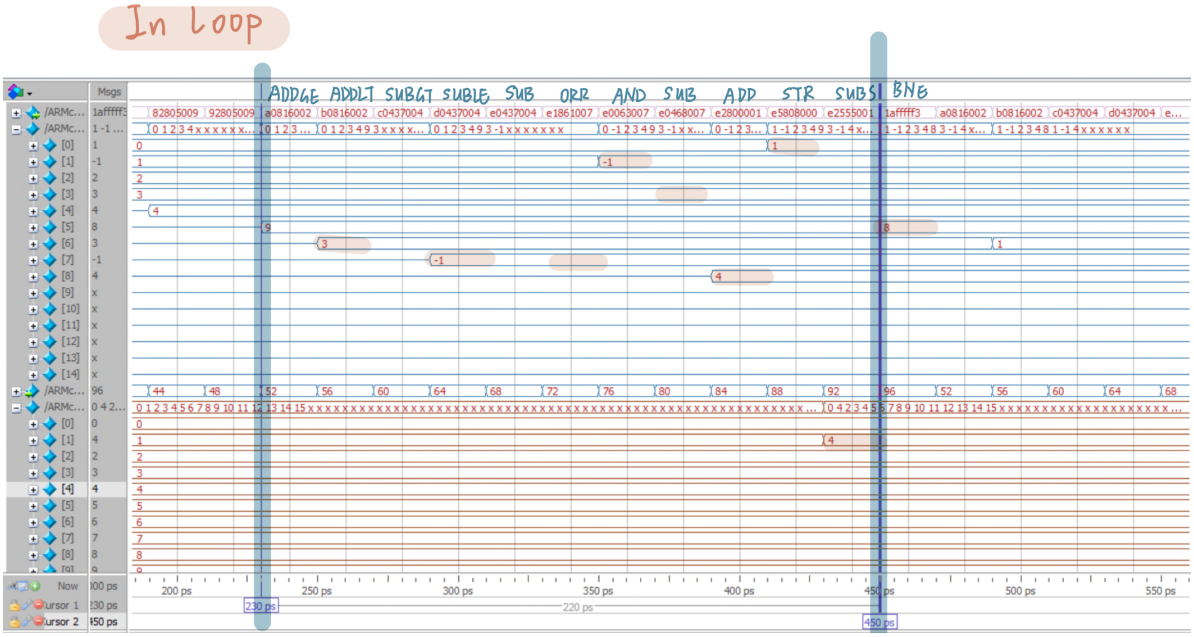

Fig. 4: Instruction flow and the data flow in the first loop

Out of Loop and after Loop

```
        ;//Out of Loop
            INSTR_MEM[23] = 32'hE5909000; //LDR R9, [R0]
```

Conditions for "out of loop" is generated in SUBS instruction and they are shown in simulation results in Fig. 5 (instructions are similar to **In loop** part given above). The number of Instructions of "after Loop" is only 1. This instruction is to store Data_Mem [R0] in R9. Now Lets check data in R9 !

This is correct because R9 now has a value of 16.

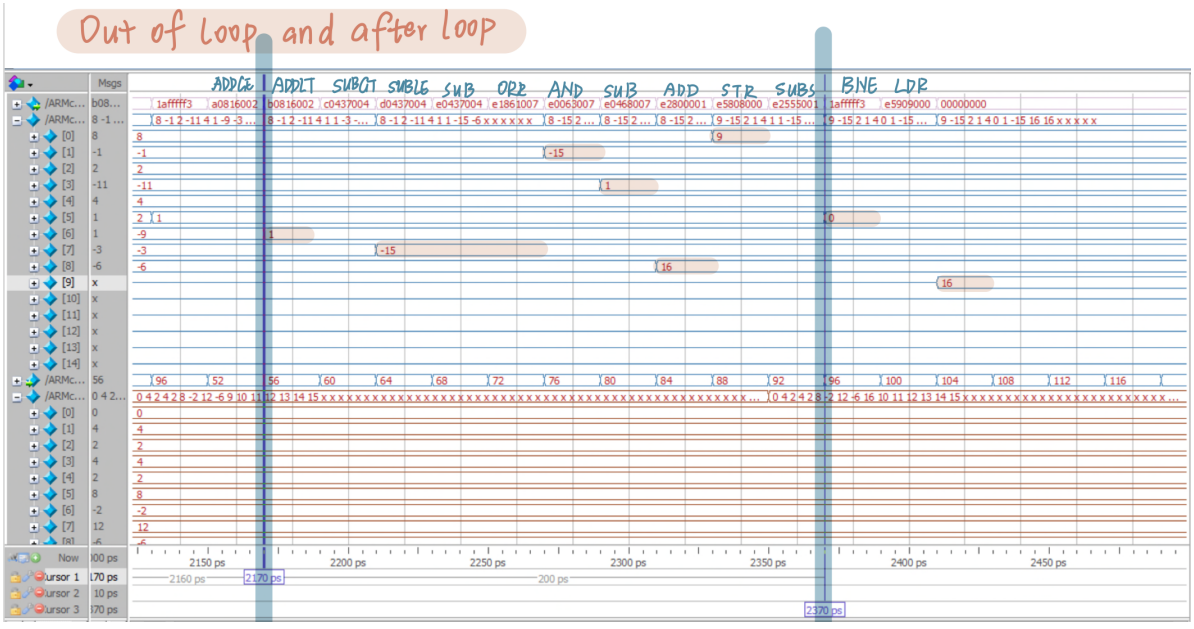

Fig. 5: Instruction flow and the data flow in the last loop and after loop

Source Codes

Top Module

```
/************************************************************
*This is top module of the ARM core.
*There are 7 modules and some multipliers for connections.
*File        : ARMcore_top.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.11.16(V1.0)
************************************************************/

module ARMcore_top(
    input wire CLK,
    input wire Reset    );


    wire [31:0] current_PC;//from ProgramCounter
    wire [31:0] PC_Plus_4; //from ProgramCounter


    wire [31:0] Instr; //from instr_mem


    wire            PCSrc;     //from Control Unit
    wire            MemtoReg;  //from Control Unit
    wire            MemWrite;  //from Control Unit
    wire [1:0]  ALUControl;//from Control Unit
    wire            ALUSrc;    //from Control Unit
    wire [1:0]  ImmSrc;    //from Control Unit
    wire            RegWrite;  //from Control Unit
    wire [1:0]  RegSrc;    //from Control Unit


    wire [31:0] RD1;     //from RegisterFile RD1
    wire [31:0] RD2;   //from RegisterFile RD2


    wire [31:0] ExtImm;//from Extend


    wire [3:0]  ALUFlags;  //from ALU
    wire [31:0] ALUResult;//from ALU


    wire [31:0] ReadData;//from Data Memory



//generate signals RA1, RA2 and PC_Plus_8 for Register File
    wire [3:0] RA1;//for RF
    wire [3:0] RA2;//for RF
    wire [31:0] PC_Plus_8; //for RF
    assign RA1 = (RegSrc[0])?4'b1111:Instr[19:16];
    assign RA2 = (RegSrc[1])?Instr[15:12]:Instr[3:0];
    assign PC_Plus_8 = PC_Plus_4 + 32'd4;
```

```
//generate signal SrcB for ALU
    wire [31:0] SrcB;//equals RD2 or ExtImm, controled by ALUSrc
    assign SrcB = (ALUSrc)?ExtImm:RD2;


//generate signal Result for PC and RF
    wire [31:0] Result; //equals ReadData or ALUResult, controled by MemtoReg
    assign Result = (MemtoReg)?ReadData:ALUResult;



/*************************************************
*Program Counter: gives the instruction address
*************************************************/
    ProgramCounter PC0(
        //input
        .CLK(CLK),
        .Reset(Reset),
        .PCSrc(PCSrc),  // from Control Unit
        .Result(Result),//ALUResult from ALU or ReadData from data_mem

        //output
        .current_PC(current_PC),//output reg [31:0]
        .PC_Plus_4(PC_Plus_4)//output [31:0]
);




/*************************************************
*Instruction Memory: gives 32-bit Instruction to Control Unit
*************************************************/
    instr_mem InstructionMem0(
        //input
        .PC(current_PC),//input wire [31:0],from ProgramCounter

        //output
        .Instr(Instr)//output wire [31:0]
);



/*************************************************
*Control Unit: gives the control signals: PCsrc, MemtoReg, MemWrite,
*ALUControl, ALUSrc, ImmSrc, RegWrite.
*************************************************/
    ControlUnit ControlUnit0(
        //input
        .CLK(CLK),
        .Instr(Instr),  //input [31:0], from instr_mem
        .ALUFlags(ALUFlags),    //input [3:0], from ALU

        //output
        .MemtoReg(MemtoReg), //from Decoder
        .MemWrite(MemWrite), //from CondLogic
        .ALUSrc(ALUSrc),     //from Decoder
        .ImmSrc(ImmSrc),       //from Decoder
```

```
        .RegWrite(RegWrite), //from CondLogic
        .RegSrc(RegSrc),     //from Decoder
        .ALUControl(ALUControl),//from Decoder
        .PCSrc(PCSrc)        //from CondLogic
    );



/*************************************************
*Register File: 2^4 registers, each register stores 32-bit data
*************************************************/
    RegisterFile RegisterFile0 (
        //input
        .CLK(CLK),
        .WE3(RegWrite),//high active, from Control Unit
        .A1(RA1),//input [3:0], Read index1
        .A2(RA2),//input [3:0], Read index2
        .A3(Instr[15:12]),   //input [3:0], from Instr Write index
        .WD3(Result),//input [31:0], Write data
        .R15(PC_Plus_8),//[31:0], R15 Data in

        //output
        .RD1(RD1),//Read data1,output reg [31:0]
        .RD2(RD2) //Read data2,output reg [31:0]
    );



/*************************************************
*Extend: gives 32-bit ExtImm
*************************************************/
    Extend Extend0(
        //input
        .ImmSrc(ImmSrc),//input [1:0], from Control Unit
        .InstrImm(Instr[23:0]),//input [23:0], from Instr

        //output
        .ExtImm(ExtImm)// [31:0]
    );



/*************************************************
*ALU: gives ALUresult
*************************************************/
    ALU ALU0(
        //input
        .A(RD1),//[31:0], from RF
        .B(SrcB),//[31:0], from mux
        .ALUControl(ALUControl),//[1:0], from Control Unit

        //output
        .ALUResult(ALUResult),//output reg [31:0]
        .ALUFlags(ALUFlags)//output [3:0], N, Z, C, V,
    );
```

```
/***********************************************
*Data Memory: gives ReadData and can write data.
***********************************************/
    data_mem DataMem0(
        //input
        .CLK(CLK),
        .Address(ALUResult),//input wire    [31:0], from ALUResult
        //input write port
        .WE(MemWrite),      //Write Enable, from Control Unit
        .WD(RD2),  //Write Data, [31:0], from RF

        //output read port
        .ReadData(ReadData)//output wire [31:0]
    );



endmodule
```

Top Module Test Bench

```
/***************************************************************
*This is the test bench of the top module of the ARM core.
*File        : ARMcore_top_tb.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.11.16(V1.0)
***************************************************************/

module ARMcore_top_tb();

    reg CLK;
    reg Reset;

    initial begin
        CLK = 1'b0;
        forever #10 CLK = ~CLK;//unit is ps
    end

    initial begin
        #0  Reset = 1'b1;
        #25 Reset = 1'b0;
    end

    ARMcore_top ARMcore0(
        .CLK(CLK),
        .Reset(Reset)
    );

    endmodule
```

Modules in CPU

ProgramCounter.v

```
/*************************************************************
*This is a design of program counter
*File     : ProgramCounter.v
*Author   : Shuo Feng
*Class    : SME 309
*DateTime : 2021.9.28 (V1.0)
*************************************************************/

module ProgramCounter(
    input CLK,
    input Reset,
    input PCSrc,  //control signal
    input [31:0] Result,

    output reg [31:0] current_PC,
    output [31:0] PC_Plus_4
);

assign PC_Plus_4=current_PC + 32'd4;

    always@(posedge CLK)
        begin

        if(Reset)
        current_PC<=32'b0;
        else if(PCSrc)
        current_PC<=Result;
        else
        current_PC<=PC_Plus_4;
        end


endmodule
```

instr_mem.v

```
/*************************************************************
*This is the Instruction Memory block, which is a part of the processor
*File        : instr_mem.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.11.16(V1.0)
*************************************************************/

module instr_mem(
    input wire  [31:0]  PC,
    output wire [31:0]  Instr
);

    reg [31:0]  INSTR_MEM[0:127];//only need 7 bit to specify the address, so pc
```

```
//  initial $readmemh (
"C:/intelFPGA/18.1/project/microcontroller/Files/instr_mem(example).txt",
INSTR_MEM ); //you should write your own path here

//---------------------------------------------------------------------
// Instruction Memory (given example in lab tutorial)
//---------------------------------------------------------------------
//integer i;
//initial begin
//          INSTR_MEM[0] = 32'hE2000000;
//          INSTR_MEM[1] = 32'hE5901001;
//          INSTR_MEM[2] = 32'hE5902002;
//          INSTR_MEM[3] = 32'hE5903003;
//          INSTR_MEM[4] = 32'hE5904004;
//          INSTR_MEM[5] = 32'hE28F5000;
//          //LOOP
//          INSTR_MEM[6] = 32'hE0816002; //
//          INSTR_MEM[7] = 32'hE0437004;
//          INSTR_MEM[8] = 32'hE1861007;
//          INSTR_MEM[9] = 32'hE0063007;
//          INSTR_MEM[10] = 32'hE0468007;
//          INSTR_MEM[11] = 32'hE2800001;
//          INSTR_MEM[12] = 32'hE5808000;
//          INSTR_MEM[13] = 32'hE2555001;
//          INSTR_MEM[14] = 32'h1AFFFFF6; //
//          INSTR_MEM[15] = 32'hE5909000; //
//          for(i = 16; i < 128; i = i+1) begin
//              INSTR_MEM[i] = 32'h0;
//          end
//end



// Instruction Memory
//---------------------------------------------------------------------
//---------------------------------------------------------------------
// Instruction Memory
//---------------------------------------------------------------------
integer i;
initial begin
            INSTR_MEM[0] = 32'hE2100000; //ANDS R0, R0, #0
            INSTR_MEM[1] = 32'h15901001; //not execute, LDRNE R1, [R0, #1]
            INSTR_MEM[2] = 32'h05901001; //LDREQ R1, [R0, #1]
            INSTR_MEM[3] = 32'h25902002; //not execute, LDRCS R2, [R0, #2]
            INSTR_MEM[4] = 32'h35902002; //LDRCC R2, [R0, #2]
            INSTR_MEM[5] = 32'h45903003; //not execute, LDRMI R3, [R0, #3]
            INSTR_MEM[6] = 32'h55903003; //LDRPL R3, [R0, #3]
            INSTR_MEM[7] = 32'h65904004; //not execute, LDRVS R4, [R0, #4]
            INSTR_MEM[8] = 32'h75904004; //LDRVC R4, [R0, #4]
            INSTR_MEM[9] = 32'h82805009; //not execute, ADDHI R5, R0, #9
            INSTR_MEM[10] = 32'h92805009;//ADDLS R5, R0, #9
        //Loop
            INSTR_MEM[11] = 32'hA0816002; //ADDGE R6, R1, R2
            INSTR_MEM[12] = 32'hB0816002; //not execute, ADDLT R6, R1, R2
            INSTR_MEM[13] = 32'hC0437004; //SUBGT R7, R3, R4
            INSTR_MEM[14] = 32'hD0437004; //SUBLE R7, R3, R4
            INSTR_MEM[15] = 32'hE0437004; //SUB R7, R3, R4
            INSTR_MEM[16] = 32'hE1861007; //ORR R1, R6, R7
```

```
            INSTR_MEM[17] = 32'hE0063007; //AND R3, R6, R7
            INSTR_MEM[18] = 32'hE0468007; //SUB R8, R6, R7
            INSTR_MEM[19] = 32'hE2800001; //ADD R0, R0, #1
            INSTR_MEM[20] = 32'hE5808000; //STR R8, [R0]
            INSTR_MEM[21] = 32'hE2555001; //SUBS R5, R5, #1
            INSTR_MEM[22] = 32'h1AFFFFF3; //BNE LOOP
        //Out of Loop
            INSTR_MEM[23] = 32'hE5909000; //LDR R9, [R0]
            for(i = 24; i < 128; i = i+1) begin
                INSTR_MEM[i] = 32'h0;
            end
end



    assign Instr = ( (PC >= 32'h00000000) & (PC <= 32'h000001FC) ) ? // To check
if PC is in the valid range, assuming 128 word memory.
                INSTR_MEM[PC[8:2]] : 32'h00000000 ; //7 bit, no problem

endmodule
```

ControlUnit.v

```
/*************************************************************
*This is Control Unit block, which contains Decoder and Conditional Logic.
*This instantiates one Decoder (Decoder1) and one CondLogic (CondLogic1)
*File        : ControlUnit.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.10.26(V1.0)
*************************************************************/

module ControlUnit(
    input [31:0] Instr,  //for Decoder, CondLogic
    input [3:0] ALUFlags,//for CondLogic
    input CLK,           //for CondLogic

    output  MemtoReg, //from Decoder
    output  MemWrite, //from CondLogic
   output  ALUSrc,   //from Decoder
   output  [1:0] ImmSrc,//from Decoder
   output  RegWrite, //from CondLogic
   output  [1:0] RegSrc,//from Decoder
   output  [1:0] ALUControl,//from Decoder
   output  PCSrc     //from CondLogic
    );

    wire [4:0] Cond;//input of Condtional Logic
    wire PCS, RegW, MemW;//output of Decoder
    wire [1:0] FlagW;////output of Decoder

    assign Cond = Instr[31:28];

    Decoder Decoder1(
    //input
     .Instr(Instr),
```

```
     //output
     .MemtoReg(MemtoReg),
    .MemW(MemW),
    .ALUSrc(ALUSrc),
    .ImmSrc(ImmSrc),
    .RegW(RegW),
    .RegSrc(RegSrc),
    .ALUControl(ALUControl),
    .FlagW(FlagW),
    .PCS(PCS)
    );

    CondLogic CondLogic1(
     //input
     .CLK(CLK),
     .PCS(PCS),
     .RegW(RegW),
     .MemW(MemW),
     .FlagW(FlagW),
     .Cond(Cond),
     .ALUFlags(ALUFlags),

     //output
     .PCSrc(PCSrc),
     .RegWrite(RegWrite),
     .MemWrite(MemWrite)
    );

endmodule



/*****************************************************************
*This is a conditional logic block, which is a part of Control Unit
*File        : CondLogic.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.10.26(V1.0)
*****************************************************************/

`define EQ 4'b0000
`define NE 4'b0001
`define CS_HS 4'b0010
`define CC_LO 4'b0011
`define MI 4'b0100
`define PL 4'b0101
`define VS 4'b0110
`define VC 4'b0111
`define HI 4'b1000
`define LS 4'b1001
`define GE 4'b1010
`define LT 4'b1011
`define GT 4'b1100
`define LE 4'b1101
`define AL 4'b1110
```

```
module CondLogic(
    input CLK,
    input PCS,
    input RegW,
    input MemW,
    input [1:0] FlagW,
    input [3:0] Cond,
    input [3:0] ALUFlags,

    output PCSrc,
    output RegWrite,
    output MemWrite
    );

    reg CondEx;  //whether to exacute
    reg N = 0, Z = 0, C = 0, V = 0;
    wire [1:0] FlagWrite;

    //1 define the output of CondLogic
    assign PCSrc = PCS & CondEx;
    assign RegWrite = RegW & CondEx;
    assign MemWrite = MemW & CondEx;
    assign FlagWrite[1] = FlagW[1] & CondEx;
    assign FlagWrite[0] = FlagW[0] & CondEx;


    //2 flags register update
    always @(posedge CLK) begin
        if(FlagWrite[1])
            {N,Z} <= ALUFlags[3:2];
        if(FlagWrite[0])
            {C,V} <= ALUFlags[1:0];
    end

    //CondEx generate, case statement
    always @(*) begin
        case(Cond)
            `EQ   : CondEx = Z;
            `NE   : CondEx = ~Z;
            `CS_HS: CondEx = C;
            `CC_LO: CondEx = ~C;
            `MI   : CondEx = N;
            `PL   : CondEx = ~N;
            `VS   : CondEx = V;
            `VC   : CondEx = ~V;
            `HI   : CondEx = (~Z) & C;
            `LS   : CondEx = Z | (~C);
            `GE   : CondEx = ~(N^V);
            `LT   : CondEx = N ^ V;
            `GT   : CondEx = (~Z)&(~(N^V));
            `LE   : CondEx = Z|(N^V);
            `AL   : CondEx = 1'b1;
          default: CondEx = 1'bx;
          endcase
    end

endmodule
```

```
/*****************************************************************
*This is a Decoder
*File         : Decoder.v
*Author       : Shuo Feng
*Class        : SME 309
*DateTime     : 2021.10.19(V1.0)
*****************************************************************/

//Cases of MainD
`define DP_reg1 4'b0001
`define DP_reg2 4'b0000
`define DP_imm1 4'b0010
`define DP_imm2 4'b0011
`define STR1 4'b0100
`define STR2 4'b0110
`define LDR1 4'b0101
`define LDR2 4'b0111
`define B1 4'b1000
`define B2 4'b1001
`define B3 4'b1010
`define B4 4'b1011

//Cases of ALUD
`define NotDP1 5'b00000
`define NotDP2 5'b00001
`define NotDP3 5'b00010
`define NotDP4 5'b00011
`define NotDP5 5'b00100
`define NotDP6 5'b00101
`define NotDP7 5'b00110
`define NotDP8 5'b00111
`define NotDP9 5'b01000
`define NotDP10 5'b01001
`define NotDP11 5'b01010
`define NotDP12 5'b01011
`define NotDP13 5'b01100
`define NotDP14 5'b01101
`define NotDP15 5'b01110
`define NotDP16 5'b01111
`define ADD 5'b10100
`define SUB 5'b10010
`define AND 5'b10000
`define ORR 5'b11100


module Decoder(
    input [31:0] Instr,

    output reg MemtoReg,
    output reg MemW,
    output reg ALUSrc,
    output reg [1:0] ImmSrc,
    output reg RegW,
    output reg [1:0] RegSrc,
    output reg [1:0] ALUControl,
```

```
    output reg [1:0] FlagW,
    output reg PCS
    );

   reg ALUOp ;
   reg Branch ;

    wire [3:0] MainD;
    wire S;//Funct0
    wire Rd = Instr[15:12];
    reg [4:0] ALUD;
    assign MainD = {Instr[27:25],Instr[20]};
    assign S=Instr[20];


    //Main Decoder
    always @(*)begin
        case(MainD)
            `DP_reg1: begin Branch = 1'b0; MemtoReg = 1'b0; MemW = 1'b0; ALUSrc
= 1'b0;
                      ImmSrc = 2'bxx; RegW = 1'b1; RegSrc = 2'b00; ALUOp =
1'b1;
                      end
            `DP_reg2: begin Branch = 1'b0; MemtoReg = 1'b0; MemW = 1'b0; ALUSrc
= 1'b0;
                      ImmSrc = 2'bxx; RegW = 1'b1; RegSrc = 2'b00; ALUOp =
1'b1;
                      end

            `DP_imm1: begin
                      Branch = 0; MemtoReg = 0; MemW = 0; ALUSrc = 1;
                      ImmSrc = 2'b00; RegW = 1; RegSrc = 2'bx0; ALUOp = 1'b1;
                      end
            `DP_imm2: begin
                      Branch = 0; MemtoReg = 0; MemW = 0; ALUSrc = 1;
                      ImmSrc = 2'b00; RegW = 1; RegSrc = 2'bx0; ALUOp = 1'b1;
                      end

            `STR1   : begin
                      Branch = 0; MemtoReg = 1'bx; MemW = 1; ALUSrc = 1;
                      ImmSrc = 2'b01; RegW = 0; RegSrc = 2'b10; ALUOp = 1'b0;
                      end
            `STR2   : begin
                      Branch = 0; MemtoReg = 1'bx; MemW = 1; ALUSrc = 1;
                      ImmSrc = 2'b01; RegW = 0; RegSrc = 2'b10; ALUOp = 1'b0;
                      end

            `LDR1   : begin
                      Branch = 0; MemtoReg = 1; MemW = 0; ALUSrc = 1;
                      ImmSrc = 2'b01; RegW = 1; RegSrc = 2'bx0; ALUOp = 1'b0;
                      end
            `LDR2   : begin
                      Branch = 0; MemtoReg = 1; MemW = 0; ALUSrc = 1;
                      ImmSrc = 2'b01; RegW = 1; RegSrc = 2'bx0; ALUOp = 1'b0;
                      end

            `B1     : begin
                      Branch = 1; MemtoReg = 0; MemW = 0; ALUSrc = 1;
```

```
                        ImmSrc = 2'b10; RegW = 0; RegSrc = 2'bx1; ALUOp = 1'b0;
                        end
            `B2     : begin
                        Branch = 1; MemtoReg = 0; MemW = 0; ALUSrc = 1;
                        ImmSrc = 2'b10; RegW = 0; RegSrc = 2'bx1; ALUOp = 1'b0;
                        end
            `B3     : begin
                        Branch = 1; MemtoReg = 0; MemW = 0; ALUSrc = 1;
                        ImmSrc = 2'b10; RegW = 0; RegSrc = 2'bx1; ALUOp = 1'b0;
                        end
            `B4     : begin
                        Branch = 1; MemtoReg = 0; MemW = 0; ALUSrc = 1;
                        ImmSrc = 2'b10; RegW = 0; RegSrc = 2'bx1; ALUOp = 1'b0;
                        end

            default:begin
                        Branch = 1'bx; MemtoReg = 1'bx; MemW = 1'bx; ALUSrc =
1'bx;
                        ImmSrc = 2'bxx; RegW = 1'bx; RegSrc = 2'bxx; ALUOp =
1'bx;
                    end
    endcase
    ALUD = {ALUOp, Instr[24:21]};

    //ALU Decoder
    case(ALUD)
            `NotDP1: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP2: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP3: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP4: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP5: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP6: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP7: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP8: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP9: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP10: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP11: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP12: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP13: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP14: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP15: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
            `NotDP16: begin
                ALUControl = 2'b00; FlagW = 2'b00; end
```

```
            `ADD  : begin
                ALUControl = 2'b00;
                if(S) FlagW = 2'b11; else FlagW = 2'b00; end
            `SUB  : begin
                ALUControl = 2'b01;
                if(S) FlagW = 2'b11; else FlagW = 2'b00; end
            `AND  : begin
                ALUControl = 2'b10;
                if(S) FlagW = 2'b10; else FlagW = 2'b00; end
            `ORR  : begin
                ALUControl = 2'b11;
                if(S) FlagW = 2'b10; else FlagW = 2'b00; end
            default: begin
                ALUControl = 2'bxx; FlagW = 2'bxx; end
        endcase
    end


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


endmodule
```

RegisterFile.v

```
/*****************************************************************
*This is a design of register file
*File     : RegisterFile_tb.v
*Author   : Shuo Feng
*Class    : SME 309
*DateTime : 2021.9.28 (V1.0)
*****************************************************************/

module RegisterFile (
    input CLK,
    input WE3,//high active
    input [3:0] A1,//Read index1
    input [3:0] A2,//Read index2
    input [3:0] A3,//Write index
    input [31:0] WD3,//Write data
    input [31:0] R15,//R15 Data in
    output reg [31:0] RD1,//Read data1
    output reg [31:0] RD2//Read data2
    );

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

    //read
    always@(*) begin
    if(A1 == 4'd15)
    RD1 = R15;
    else
    RD1 = RegBankCore[A1];
    if(A2 == 4'd15)
    RD2 = R15;
```

```
    else
    RD2 = RegBankCore[A2];
    end

    //write
    always@(posedge CLK)
    begin
    if(WE3)
    RegBankCore[A3] <= WD3;
    end

endmodule
```

Extend.v

```
/*****************************************************************
*This is an extend block, which is a part of the processor
*File        : Extend.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.11.16(V1.0)
*****************************************************************/

module Extend (
    input [1:0] ImmSrc,
    input [23:0] InstrImm,

    output reg [31:0] ExtImm
    );

    always@(*) begin
        case(ImmSrc)
        2'b00  : ExtImm = {24'b0,InstrImm[7:0]};//DP
        2'b01  : ExtImm = {20'b0,InstrImm[11:0]};//LDR/STR
        2'b10  : ExtImm = {{6{InstrImm[23]}},InstrImm[23:0],{2{1'b0}}};//B
        default: ExtImm = 32'hxxxxxxxx;
        endcase
    end

endmodule
```

ALU.v

```
/*****************************************************************
*This is an ALU (arithmetic logic unit).
*File        : ALU.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.11.2(V1.0)
*****************************************************************/
`define ADD 2'b00
`define SUB 2'b01
`define AND 2'b10
`define ORR 2'b11
module ALU(
    input [31:0] A,
    input [31:0] B,
```

```
    input [1:0] ALUControl,

    output reg [31:0] ALUResult,
    output [3:0] ALUFlags//N, Z, C, V, connect to the Conditional Logic Unit
    );

    reg Cin;
    wire Cout;
    wire [31:0] Sum;
    wire [31:0] And;
    wire [31:0] Orr;

    reg [31:0] Btem;

    //Arithmetic Operation
    //get 32-bit Sum
    Adder32bit Adder0(.a(A), .b(Btem), .Cin(Cin), .Cout(Cout), .Sum(Sum));
    //get 32-bit And
    assign And = A & B;
    //get 32-bit Orr
    assign Orr = A | B;

    //ALUFlags Generation
    always@(*) begin
    if(ALUControl[0]) begin Btem=~B; Cin = 1'b1; end
    else begin Btem=B; Cin = 1'b0; end
    end

    always@(*) begin
    case(ALUControl)
        `ADD : ALUResult = Sum;
        `SUB : ALUResult = Sum;
        `AND : ALUResult = And;
        `ORR : ALUResult = Orr;
       default: ALUResult = 32'hxxxxxxxx;
    endcase
    end

    //Z
    assign ALUFlags[2] = &(~ALUResult[31:0]);
    //N
    assign ALUFlags[3] = ALUResult[31];
    //C
    assign ALUFlags[1] = (~ALUControl[1]) & Cout;
    //V
    assign ALUFlags[0] = ((~(ALUControl[0]^A[31]^B[31]))&(A[31]^Sum[31])&
(~ALUControl[1]));

endmodule


/*************************************************************
*This is a 32-bit Ripple Carry Adder, whole upper module is ALU.v
*THere are 32 inctances of FullAdder.v
*File        : Adder32bit.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.11.2(V2.0)
```

```
*V1.0 is duplicated 32 full adders made up 32-bit adder.
**************************************************************/
module Adder32bit(
    input [31:0] a,
    input [31:0] b,
    input Cin,

    output Cout,
    output [31:0] Sum
    );

    wire [31:0] c;
    assign Cout=c[31];

    fa fa0(.a(a[0]),.b(b[0]),.cin(Cin),.cout(c[0]),.sum(Sum[0]));

    genvar i;
    generate
        for (i=1; i<32; i=i+1) begin: generate_block_identifier // <-- example
block name

        fa whatever_fa(.a(a[i]),.b(b[i]),.cin(c[i-1]),.cout(c[i]),.sum(Sum[i]));

        end
    endgenerate

//This is version 1, duplicated 32 full adders made up 32-bit adder

//  fa fa0(.a(a[0]),.b(b[0]),.cin(cin),.cout(c[0]),.s(s[0]));
//  fa fa1(.a(a[1]),.b(b[1]),.cin(c[0]),.cout(c[1]),.s(s[1]));
//  fa fa2(.a(a[2]),.b(b[2]),.cin(c[1]),.cout(c[2]),.s(s[2]));
//  fa fa3(.a(a[3]),.b(b[3]),.cin(c[2]),.cout(c[3]),.s(s[3]));
//  fa4 ~ fa30 .........
//  fa fa31(.a(a[31]),.b(b[31]),.cin(c[30]),.cout(c[31]),.s(s[31]));

endmodule




/**************************************************************
*This is a 1-bit full adder, whole upper module is Adder32bit.v
*File        : fa.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.11.2(V1.0)
**************************************************************/
module fa(
    input a,
    input b,
    input cin,

    output cout,
    output sum
    );

    assign sum = a ^ b ^ cin;
    assign cout = (a & b) | cin & (a^b);
```

```
endmodule
```

data_mem.v

```
/*************************************************************
*This is the Data Memory block, which is a part of the processor
*File        : data_mem.v
*Author      : Shuo Feng
*Class       : SME 309
*DateTime    : 2021.11.16(V1.0)
*************************************************************/

module data_mem(
        input wire         CLK,
        input wire  [31:0] Address,

        // write port
        input wire         WE,     //Write Enable
        input wire  [31:0] WD,     //Write Date

        // read port
        output wire [31:0] ReadData
    );



    reg [31:0] DATA_MEM [0:127];

    initial $readmemh (
"C:/intelFPGA/18.1/project/microcontroller/Files/data_mem(example).txt",
DATA_MEM ); //you should write your own path here



//mem write
    always @(posedge CLK) begin
        if (WE)
            DATA_MEM[Address] <= WD;
        else
            DATA_MEM[Address] <=DATA_MEM[Address];
    end

//mem read

    assign ReadData=DATA_MEM[Address][31:0];

endmodule
```

Additional Test Result

The test results of original given example in lab tutorial is shown in Fig. 6. The analysis is similar to the process in **Elaborate Design of Test** part.

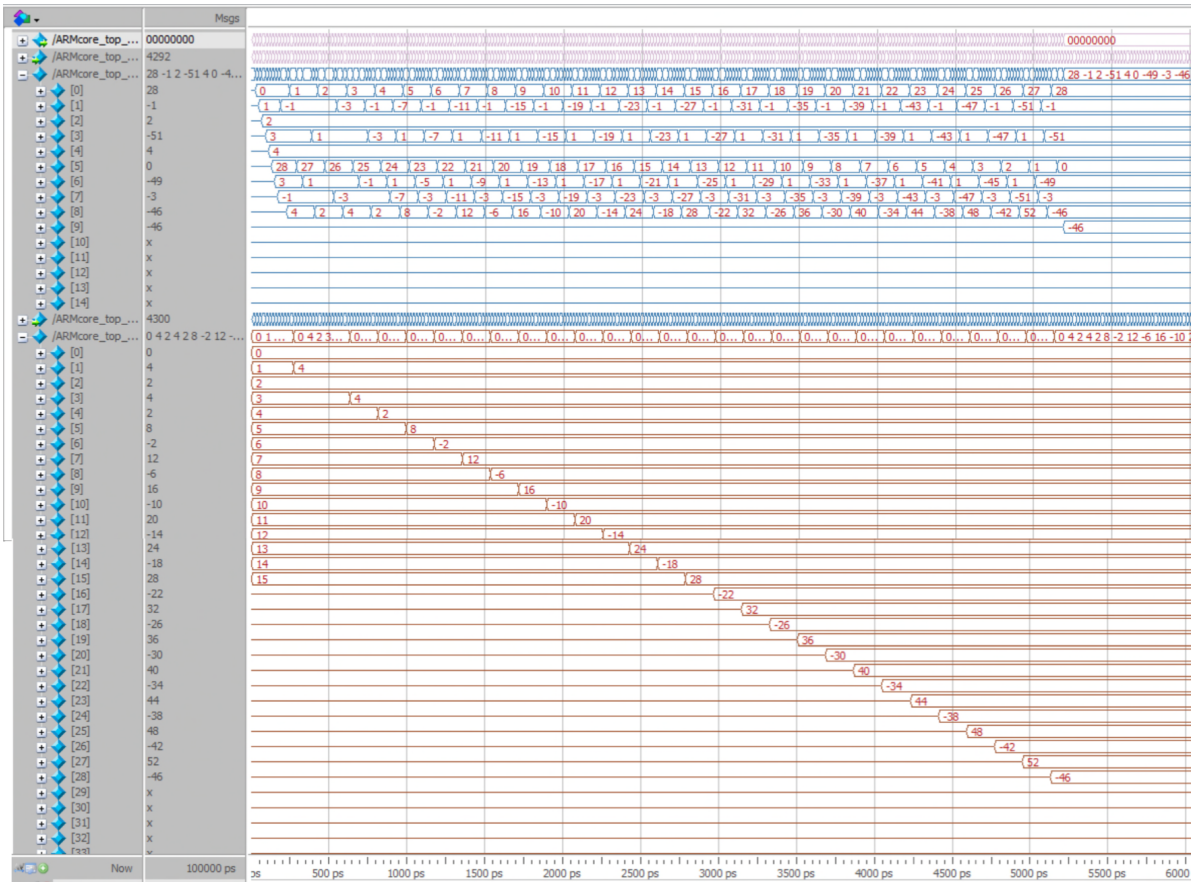

Fig 6: The test results of given example in lab tutorial