SME309 Lab Report - Single Cycle Processor

By HUANG Guanchao, SID 11912309, from the School of Microelectronics. All of the materials, including source code, diagrams and the report itself can be retrieved at **my GitHub repo**.

- Code edited in **VSCode**, with support from extensions **TerosHDL** and **FPGA Develop Support**.
- Compiled and simulated with Vivado 2021.1, ModelSim 2020.1
- Testbench assembled using Keil uVision 5

Instruction Set Architecture

The single cycle processor is implemented based on reduced 32bit ARMv3 instruction set architecture, which provides support for the following instructions:

- Datapath instructions
  - `ADD`
  - `SUB`
  - `AND`
  - `ORR`

  Immediate number with rotation, register with shift, and register-shifted register are not supported for `Src2`.
- Branching instruction `B`
- Memory instructions
  - `LDR`
  - `STR`

  Register bias, as well as preindex and postindex for accessing memory are not supported.

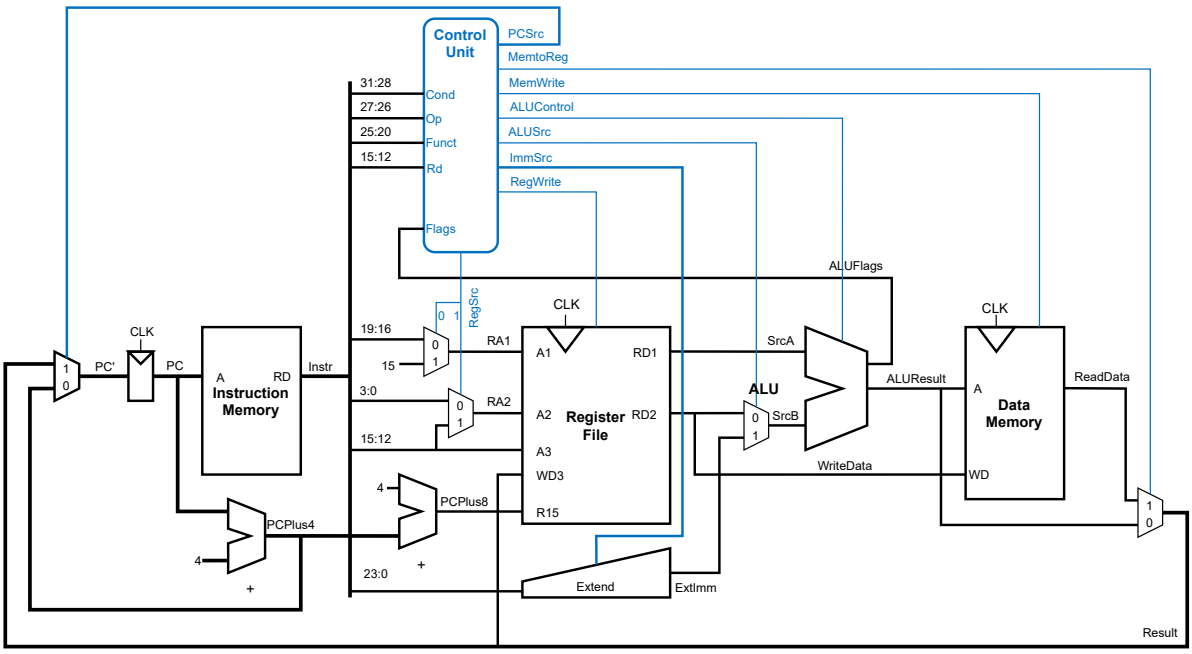
Instructions with conditions and flags setting are supported.

The register file is of 16 registers, with `R15` dedicated for program counter.

Processor Implementation

The processor diagram is shown below.

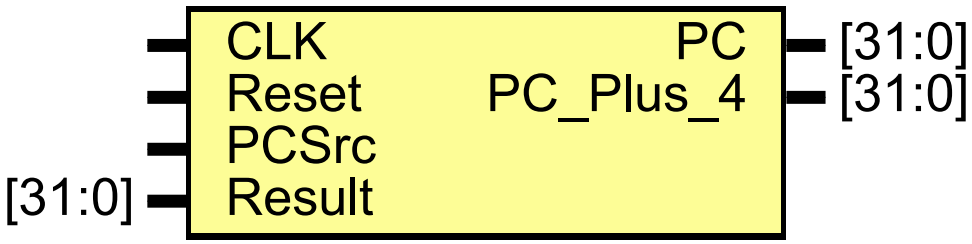
> This diagram is redrawn with **draw.io**



> Some of the following elaborations for entities are generated with TerosHDL documentation tool.

Program Counter

File: `ProgramCounter.v`



- Ports

Port name	Direction	Type
`CLK`	input	
`Reset`	input	
`PCSrc`	input	
`Result`	input	[31:0]
`PC`	output	[31:0]
`PC_Plus_4`	output	[31:0]

- Signals

Name	Type	Description
`next_PC`	reg [31:0]	Intermediate wire `next_PC`
`PC_tmp`	reg [31:0]	Sequential, synchronous reset, and updates `current_PC`

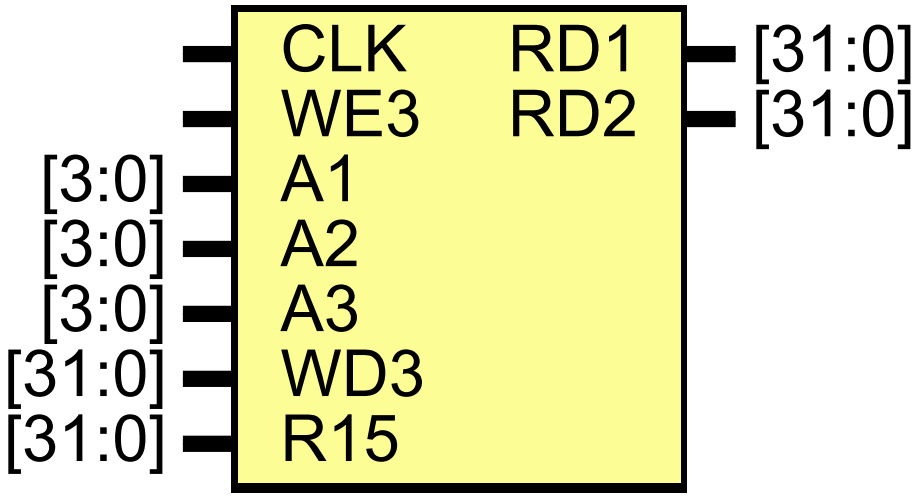
- Verilog Implementation

```
module ProgramCounter (
    input CLK,
    input Reset, // reset is high-active
    input PCSrc, // PC source
    input [31:0] Result, // From ALU

    output [31:0] PC,
    output [31:0] PC_Plus_4
  );


  //! Intermediate wire `next_PC`
  reg [31:0] next_PC;
  //! Combinational, defines `PC_Plus_4`
  assign PC_Plus_4 = PC + 4;


  //! Sequential, synchronous reset, and updates `current_PC`
  reg [31:0] PC_tmp;

  always @(posedge CLK)
    if (Reset == 1'b1)
      PC_tmp <= 0;
    else
      PC_tmp <= next_PC;

  assign PC = PC_tmp;

  //! Combinational, defines `next_PC`
  always @(*)
    if (PCSrc == 1'b0)
      next_PC = PC_Plus_4;
    else
```

```
34          next_PC = Result;
35
36
37      endmodule
```

Register File

File: `RegisterFile.v`



- Ports

Port name	Direction	Type
`CLK`	input	
`WE3`	input	
`A1`	input	[3:0]
`A2`	input	[3:0]
`A3`	input	[3:0]
`WD3`	input	[31:0]
`R15`	input	[31:0]
`RD1`	output	[31:0]
`RD2`	output	[31:0]

- Signals

Name	Type
`RegBankCore`	reg [31:0]

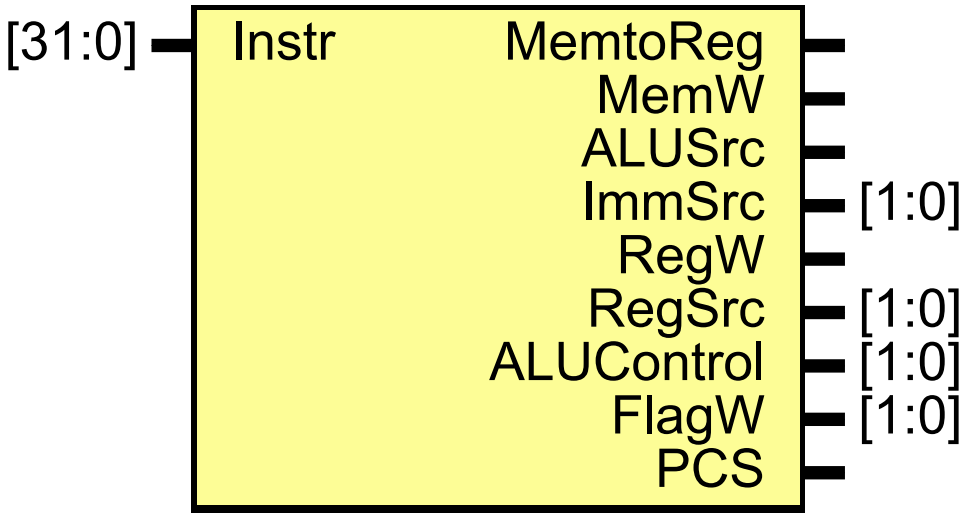
- Verilog Implementation

```
1   // RegisterFile.v
2   module RegisterFile(
```

```
3       input CLK,
4       input WE3, // high active
5       input [3:0] A1, // Read index1
6       input [3:0] A2, // Read index2
7       input [3:0] A3, // Write index
8       input [31:0] WD3,// Write data
9       input [31:0] R15, // R15 Data in
10
11      output reg [31:0] RD1, // Read data1
12      output reg [31:0] RD2 // Read data2
13    );
14
15
16    reg [31:0] RegBankCore[0:14];
17
18
19    //! Sequential, writes `WD3` into `RegBankCore` at the rising edge of clk
20    always @(posedge CLK)
21      if (WE3)
22        RegBankCore[A3] <= WD3;
23
24
25    //! Combinational, defines `RD1`
26    always @(*)
27      if (A1 == 4'b1111)
28        RD1 = R15;
29      else
30        RD1 = RegBankCore[A1];
31
32
33    //! Combinational, defines `RD2`
34    always @(*)
35      if (A2 == 4'b1111)
36        RD2 = R15;
37      else
38        RD2 = RegBankCore[A2];
39
40
41  endmodule
```

Decoder

File: `Decoder.v`



- Ports

Port name	Direction	Type
`Instr`	input	[31:0]
`MemtoReg`	output	
`MemW`	output	
`ALUSrc`	output	
`ImmSrc`	output	[1:0]
`RegW`	output	
`RegSrc`	output	[1:0]
`ALUControl`	output	[1:0]
`FlagW`	output	[1:0]
`PCS`	output	

- Signals

Name	Type
`ALUOp`	reg
`Branch`	reg
`Main`	reg [9:0]
`ALU`	reg [3:0]

- Verilog Implementation

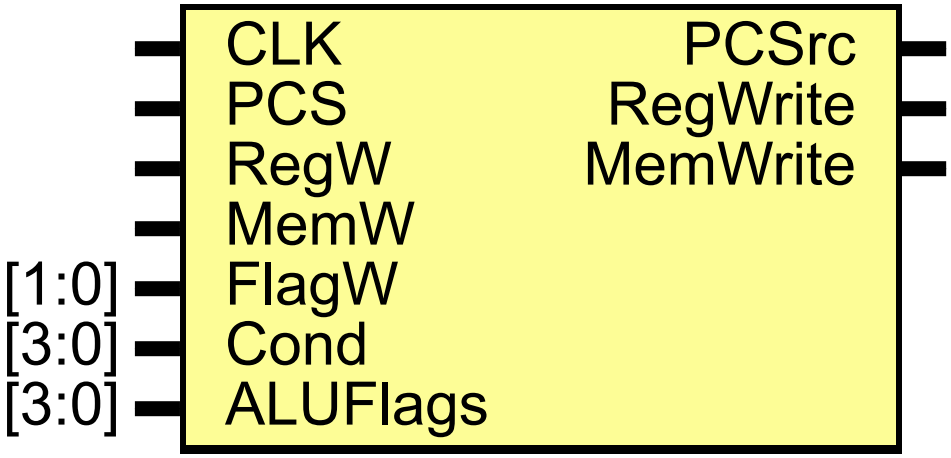
```
1   module Decoder(
2       input [31:0] Instr,
3
4       output reg MemtoReg,
```

```
5       output reg MemW,
6       output reg ALUSrc,
7       output reg [1:0] ImmSrc,
8       output reg RegW,
9       output reg [1:0] RegSrc,
10      output reg [1:0] ALUControl,
11      output reg [1:0] FlagW,
12      output reg PCS
13    );
14
15    reg ALUOp;
16    reg Branch;
17
18    reg [9:0] Main;
19    //! Main Decoder
20    always @(*)
21      begin
22        casex ({Instr[27:25], Instr[20]}) // Op, Funct5, Funct0
23          4'b000x :
24            Main = 10'b0000xx1001; // DP reg
25          4'b001x :
26            Main = 10'b0001001x01; // DP imm
27          4'b01x0 :
28            Main = 10'b0x11010100; // STR
29          4'b01x1 :
30            Main = 10'b0101011x00; // LDR
31          default:
32            Main = 10'b1001100x10; // B
33        endcase
34        {Branch, MemtoReg, MemW, ALUSrc, ImmSrc, RegW, RegSrc, ALUOp} = Main;
35      end
36
37    reg [3:0] ALU;
38    //! ALU Decoder
39    always @(*)
40      begin
41        case ({ALUOp, Instr[24:20]}) // ALUOp, Funct4:1, Funct0
42          6'b101000 :
43            ALU = 4'b0000; // ADD
44          6'b101001 :
45            ALU = 4'b0011; // ADDS
46          6'b100100 :
47            ALU = 4'b0100; // SUB
48          6'b100101 :
49            ALU = 4'b0111; // SUBS
50          6'b100000 :
51            ALU = 4'b1000; // AND
52          6'b100001 :
53            ALU = 4'b1010; // ANDS
54          6'b111000 :
55            ALU = 4'b1100; // ORR
56          6'b111001 :
57            ALU = 4'b1110; // ORRS
58          default:
59            ALU = 4'b0000; // Not DP
60        endcase
61        {ALUControl, FlagW} = ALU;
62      end
```

```
63
64      //! PC Logic
65      always @(*)
66        PCS = Branch;
67
68    endmodule
```

Conditional Logic

File: `CondLogic.v`



- Ports

Port name	Direction	Type
`CLK`	input	
`PCS`	input	
`RegW`	input	
`MemW`	input	
`FlagW`	input	[1:0]
`Cond`	input	[3:0]
`ALUFlags`	input	[3:0]
`PCSrc`	output	
`RegWrite`	output	
`MemWrite`	output	

- Signals

Name	Type
`CondEx`	reg
`N`	reg
`Z`	reg
`C`	reg
`V`	reg

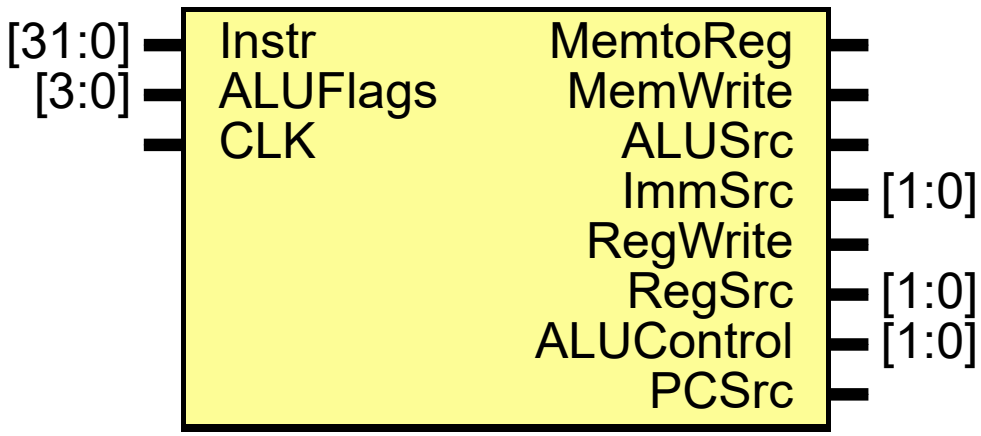
- Verilog Implementation

```
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;
  reg N = 0, Z = 0, C = 0, V = 0;
  wire [1:0] FlagWrite = FlagW[1:0] & {2{CondEx}};

  //! Output stage
  assign {PCSrc, RegWrite, MemWrite} = {PCS, RegW, MemW} & {3{CondEx}};


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


  //! Condition Check
  always @(*)
    case (Cond)
      4'b0000:
        CondEx = Z; // EQ -Equal
      4'b0001 :
        CondEx = !Z; // NE - Not equal
      4'b0010 :
        CondEx = C; // CS / HS- Carry set / Unsigned higher or same
      4'b0011 :
        CondEx = !C; // CC / LO - Carry clear / Unsigned lower
      4'b0100 :
```

```
44            CondEx = N; // MI - Minus / Negative
45          4'b0101 :
46            CondEx = !N; // PL - Plus / Positive of zero
47          4'b0110 :
48            CondEx = V; // VS - Overflow / Overflow set
49          4'b0111 :
50            CondEx = !V; // VC - No overflow / Overflow clear
51          4'b1000 :
52            CondEx = !V & C; // HI - Unsigned lower or same
53          4'b1001 :
54            CondEx = Z | !C; // LS - Unsigned lower or same
55          4'b1010 :
56            CondEx = !(N ^ V); // GE -  Signed greater than or equal
57          4'b1011 :
58            CondEx = N ^ V; // LT - Signed less than
59          4'b1100 :
60            CondEx = !Z & !(N ^ V); // GT - Signed greater than
61          4'b1101 :
62            CondEx = Z | (N ^ V); // LE - Signed less than or equal
63          default:
64            CondEx = 1'b1; // AL - Always / unconditional
65        endcase
66
67    endmodule
```

Control Unit

File: `ControlUnit.v`



- Ports

Port name	Direction	Type
`Instr`	input	[31:0]
`ALUFlags`	input	[3:0]
`CLK`	input	
`MemtoReg`	output	
`MemWrite`	output	
`ALUSrc`	output	
`ImmSrc`	output	[1:0]
`RegWrite`	output	
`RegSrc`	output	[1:0]
`ALUControl`	output	[1:0]
`PCSrc`	output	

- Signals

Name	Type
`Cond`	wire [3:0]
`PCS`	wire
`RegW`	wire
`MemW`	wire
`FlagW`	wire [1:0]

- Instantiations
- `CondLogic1` : `CondLogic`
- `Decoder1` : `Decoder`
- Verilog Implementation

```
`include "Decoder.v"
`include "CondLogic.v"

module ControlUnit(
    input [31:0] Instr,
    input [3:0] ALUFlags,
    input CLK,

    output MemtoReg,
    output MemWrite,
    output ALUSrc,
    output [1:0] ImmSrc,
    output RegWrite,
    output [1:0] RegSrc,
```

```
15        output [1:0] ALUControl,
16        output PCSrc);
17
18      wire [3:0] Cond = Instr[31:28];
19      wire PCS, RegW, MemW;
20      wire [1:0] FlagW;
21
22
23      CondLogic CondLogic1(
24                  CLK,
25                  PCS,
26                  RegW,
27                  MemW,
28                  FlagW,
29                  Cond,
30                  ALUFlags,
31
32                  PCSrc,
33                  RegWrite,
34                  MemWrite);
35
36      Decoder Decoder1(
37                Instr,
38
39                MemtoReg,
40                MemW,
41                ALUSrc,
42                ImmSrc,
43                RegW,
44                RegSrc,
45                ALUControl,
46                FlagW,
47                PCS);
48
49    endmodule
```

ALU

File: `ALU.v`

- Ports

Port name	Direction	Type
`A`	input	[31:0]
`B`	input	[31:0]
`ALUControl`	input	[1:0]
`Result`	output	[31:0]
`ALUFlags`	output	[3:0]

- Signals

Name	Type	Description
`AddIn`	reg [31:0]	FullAdder32
`Sum`	wire [31:0]	
`Cout`	wire	
`Cin`	wire	
`N`	reg	
`Z`	reg	
`C`	reg	
`V`	reg	

- Verilog Implementation

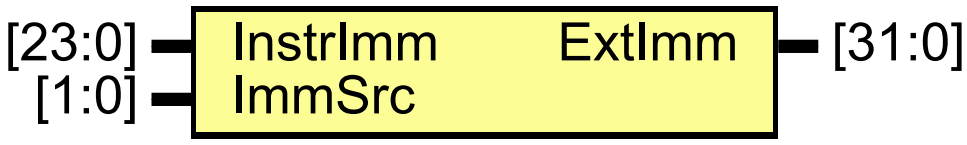
```
module ALU(
    input [31:0] A,
    input [31:0] B,
    input [1:0] ALUControl,

    output reg [31:0] Result,
    output [3:0] ALUFlags
  );


  //! FullAdder32
  reg [31:0] AddIn;
  wire [31:0] Sum;
  wire Cout;
  wire Cin = ALUControl[0];
  FullAdder32 fulladder(A, AddIn, Cin, Sum, Cout);

  //! Result
  always  @(*)
    begin: Result_Define
      if (Cin)
        AddIn = ~B;
      else
        AddIn = B;
```

```
25
26        case (ALUControl)
27          2'b11:
28            Result = A | B;
29          2'b10:
30            Result = A & B;
31          default:
32            Result = Sum;
33        endcase
34      end
35
36    reg N, Z, C, V;
37    //! Flags
38    always @(*)
39      begin: Flags_Set
40        Z = ~(|Result);
41        N = Result[31];
42        C = ~ALUControl[1] & Cout;
43        V = ~ALUControl[1] & (A[31] ^ Sum[31]) & ~(A[31] ^ B[31] ^ Cin);
44      end
45
46    assign ALUFlags = {N, Z, C, V};
47
48  endmodule
49
50  module FullAdder1(
51      input A,
52      input B,
53      input Cin,
54      output Sum,
55      output Cout);
56
57    assign Sum = A ^ B ^ Cin;
58    assign Cout = (A & B) | (A ^ B) & Cin;
59  endmodule
60
61  module FullAdder32 (
62      input [31:0] A,
63      input [31:0] B,
64      input Cin,
65      output [31:0] Sum,
66      output Cout);
67
68    wire [31:0] Cout_tmp;
69
70    FullAdder1 fulladder0(
71                A[0],
72                B[0],
73                Cin,
74                Sum[0],
75                Cout_tmp[0]);
76
77    genvar i;
78    generate
79      for(i = 1; i <= 31; i = i + 1)
80        begin: adder_gen
81          FullAdder1  fulladder(
82                        .A(A[i]),
```

```
                        .B(B[i]),
                        .Cin(Cout_tmp[i - 1]),
                        .Sum(Sum[i]),
                        .Cout(Cout_tmp[i]));
        end
    endgenerate

    assign Cout = Cout_tmp[31];
endmodule
```

Extend

File: Extend.v



- Ports

Port name	Direction	Type
`InstrImm`	input	[23:0]
`ImmSrc`	input	[1:0]
`ExtImm`	output	[31:0]

Single Cycle Processor

The previously implemented modules are connected to form the single cycle processor.

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


    //! ALU instance
    reg [31:0] SrcA, SrcB;
    wire [1:0] ALUControl;

    wire [31:0] ALUResult;
    wire [3:0] ALUFlags;

    ALU alu(
          .A(SrcA),
          .B(SrcB),
          .ALUControl(ALUControl),

          .Result(ALUResult),
          .ALUFlags(ALUFlags)
        );


```

```
24      //! ControlUnit instance
25      wire [31:0] Instr;
26
27      wire MemtoReg, MemWrite, ALUSrc;
28      wire [1:0] ImmSrc;
29      wire RegWrite;
30      wire [1:0] RegSrc;
31      wire PCSrc;
32
33      ControlUnit controlUnit(
34                  .Instr(Instr),
35                  .ALUFlags(ALUFlags),
36                  .CLK(CLK),
37
38                  .MemtoReg(MemtoReg),
39                  .MemWrite(MemWrite),
40                  .ALUSrc(ALUSrc),
41                  .ImmSrc(ImmSrc),
42                  .RegWrite(RegWrite),
43                  .RegSrc(RegSrc),
44                  .ALUControl(ALUControl),
45                  .PCSrc(PCSrc)
46                );
47
48
49      //! Extend instance
50      wire [23:0] InstrImm = Instr[23:0];
51      wire [31:0] ExtImm;
52
53      Extend extend(
54               .InstrImm(InstrImm),
55               .ImmSrc(ImmSrc),
56
57               .ExtImm(ExtImm)
58             );
59
60
61      //! ProgramCounter instance
62      wire [31:0] PC;
63      wire [31:0] PCPlus4;
64      reg [31:0] Result;
65
66      ProgramCounter programCounter(
67                      .CLK(CLK),
68                      .Reset(Reset),
69                      .PCSrc(PCSrc),
70                      .Result(Result),
71
72                      .PC(PC),
73                      .PC_Plus_4(PCPlus4)
74                    );
75
76
77      //! RegisterFile instance
78      reg [3:0] A1, A2;
79      wire [3:0] A3 = Instr[15:12];
80      wire [31:0] PCPlus8 = PCPlus4 + 4;
81      wire [31:0] RD1, RD2;
```

```
82
83      RegisterFile registerFile(
84                      .CLK(CLK),
85                      .WE3(RegWrite),
86                      .A1(A1),
87                      .A2(A2),
88                      .A3(A3),
89                      .WD3(Result),
90                      .R15(PCPlus8),
91
92                      .RD1(RD1),
93                      .RD2(RD2)
94                    );
95
96      always @(*)
97        begin
98          if (RegSrc[0])
99            A1 = 4'b1111;
100         else
101           A1 = Instr[19:16];
102
103         if (RegSrc[1])
104           A2 = Instr[15:12];
105         else
106           A2 = Instr[3:0];
107
108         if (ALUSrc)
109           SrcB = ExtImm;
110         else
111           SrcB = RD2;
112
113         SrcA = RD1;
114       end
115
116
117     //! InstrMem instance
118     InstrMem instrMem(
119                 .PC(PC),
120                 .Instr(Instr)
121               );
122
123
124     //! DataMem instance
125     wire [31:0] RD;
126     DataMem dataMem(
127               .CLK(CLK),
128               .Address(ALUResult),
129               .WE(MemWrite),
130               .WD(RD2),
131
132               .ReadData(RD)
133             );
134
135     always @(*)
136       if (MemtoReg)
137         Result = RD;
138       else
139         Result = ALUResult;
```

```
140
141     endmodule
```

Testbench

The basic idea for the testbench is to make each of the instruction depend on the previous result, such that, if the final result is correct, it is relatively confident to conclude that the processor performs as expected for each of the instructions.

After the testbench is composed, it is than compiled using `Keil uVision5`. The corresponding instructions and data are initialized in `DataMem.v` and `InstrMem.v`. Simulation is then performed with Vivado and ModelSim.

Testbench for Memory Operations

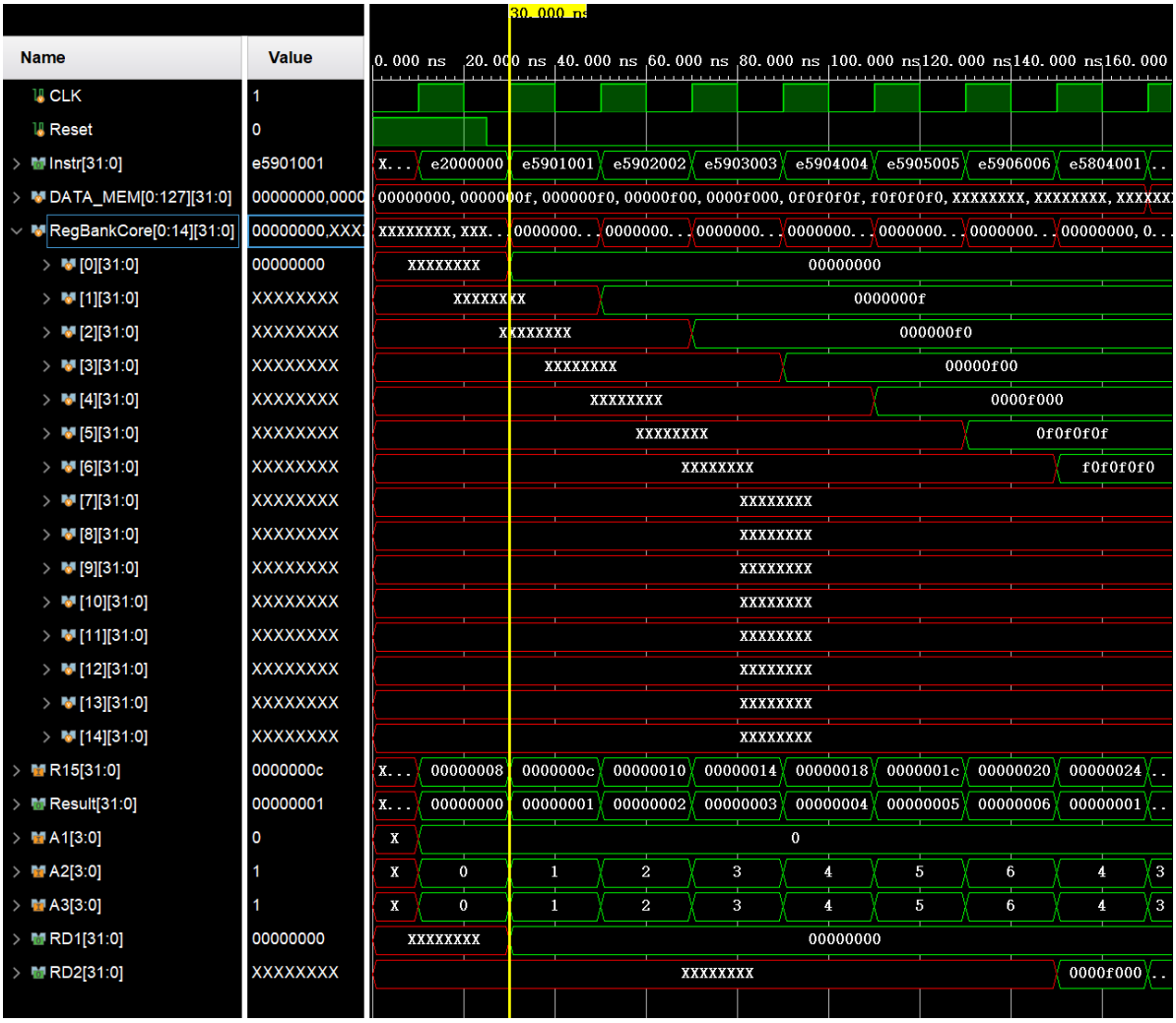
We first test for the memory operations, namely `LDR` and `STR` instructions. Since `MOV` instruction is not available in our implementation, we use `AND` operation to initialize `R0` with `0`, which is then used as the base address for loading data from `DATA_MEM`. The loaded values in the registers can be utilized in subsequent tests.

```
1   ; Test for LDR
2   AND R0, R0, #0      ; initialize R0 as 0, R0 = 0x00000000
3   LDR R1, [R0, #1]    ; R1 = DATA_MEM[1], 0x0000_000F
4   LDR R2, [R0, #2]    ; R2 = DATA_MEM[2], 0x0000_00F0
5   LDR R3, [R0, #3]    ; R3 = DATA_MEM[3], 0x0000_0F00
6   LDR R4, [R0, #4]    ; R4 = DATA_MEM[4], 0x0000_F000
7   LDR R5, [R0, #5]    ; R5 = DATA_MEM[5], 0x0F0F_0F0F
8   LDR R6, [R0, #6]    ; R6 = DATA_MEM[6], 0xF0F0_F0F0
```

The corresponding HEX value is

```
1   0x00000000  E2000000  AND       R0,R0,#0x00000000
2   0x00000004  E5901001  LDR       R1,[R0,#0x0001]
3   0x00000008  E5902002  LDR       R2,[R0,#0x0002]
4   0x0000000C  E5903003  LDR       R3,[R0,#0x0003]
5   0x00000010  E5904004  LDR       R4,[R0,#0x0004]
6   0x00000014  E5905005  LDR       R5,[R0,#0x0005]
7   0x00000018  E5906006  LDR       R6,[R0,#0x0006]
```

The simulation waveform is shown below.



According to the waveform form from 30ns to 160ns, signal `RegBankCore` clearly shows that the expected data are initialized and loaded into `R0` through `R6`.

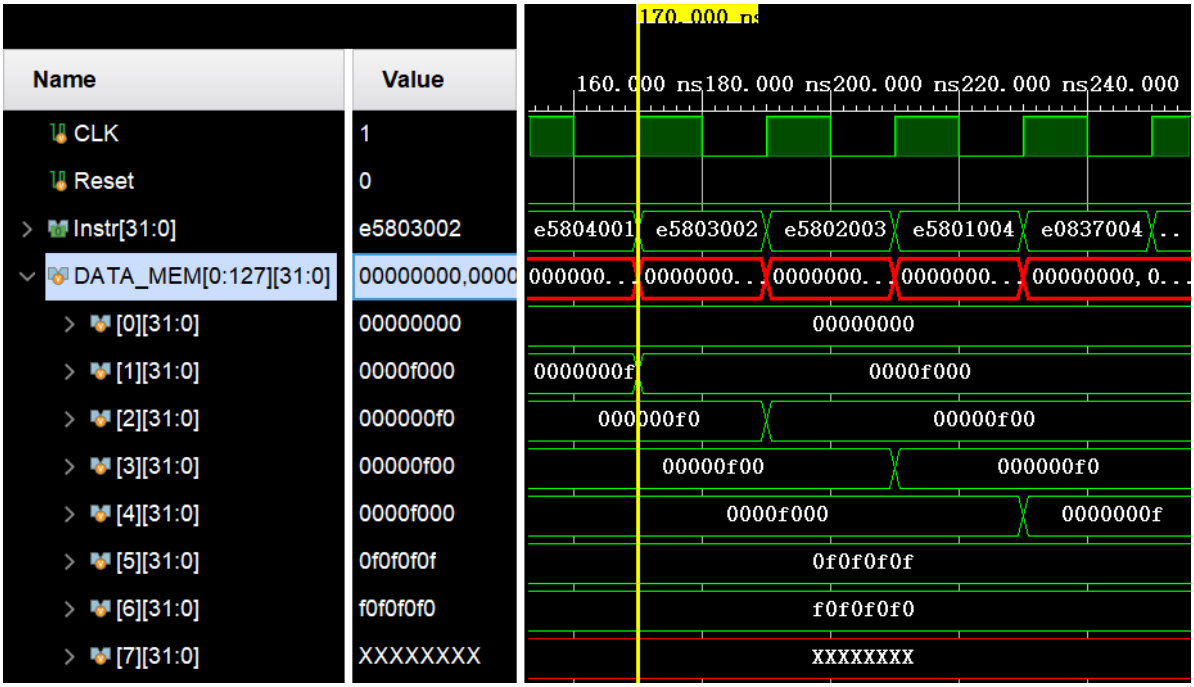
Afterwards, we write the data from the register file back into the data memory in reverse order.

```
; Test for STR
STR R4, [R0, #1]    ; DATA_MEM[1] = R4, 0x0000_F000
STR R3, [R0, #2]    ; DATA_MEM[2] = R3, 0x0000_0F00
STR R2, [R0, #3]    ; DATA_MEM[3] = R2, 0x0000_00F0
STR R1, [R0, #4]    ; DATA_MEM[4] = R1, 0x0000_000F
```

The corresponding HEX value is

```
0x0000001C  E5804001  STR       R4,[R0,#0x0001]
0x00000020  E5803002  STR       R3,[R0,#0x0002]
0x00000024  E5802003  STR       R2,[R0,#0x0003]
0x00000028  E5801004  STR       R1,[R0,#0x0004]
```

The simulation waveform is shown below.



According to the waveform form of signal `DATA_MEM` from 170ns to 240ns, the data in the data memory are modified as expected.

Thus, we may conclude that the CPU performs memory operations normally.

Testbench for Data-Path Operations

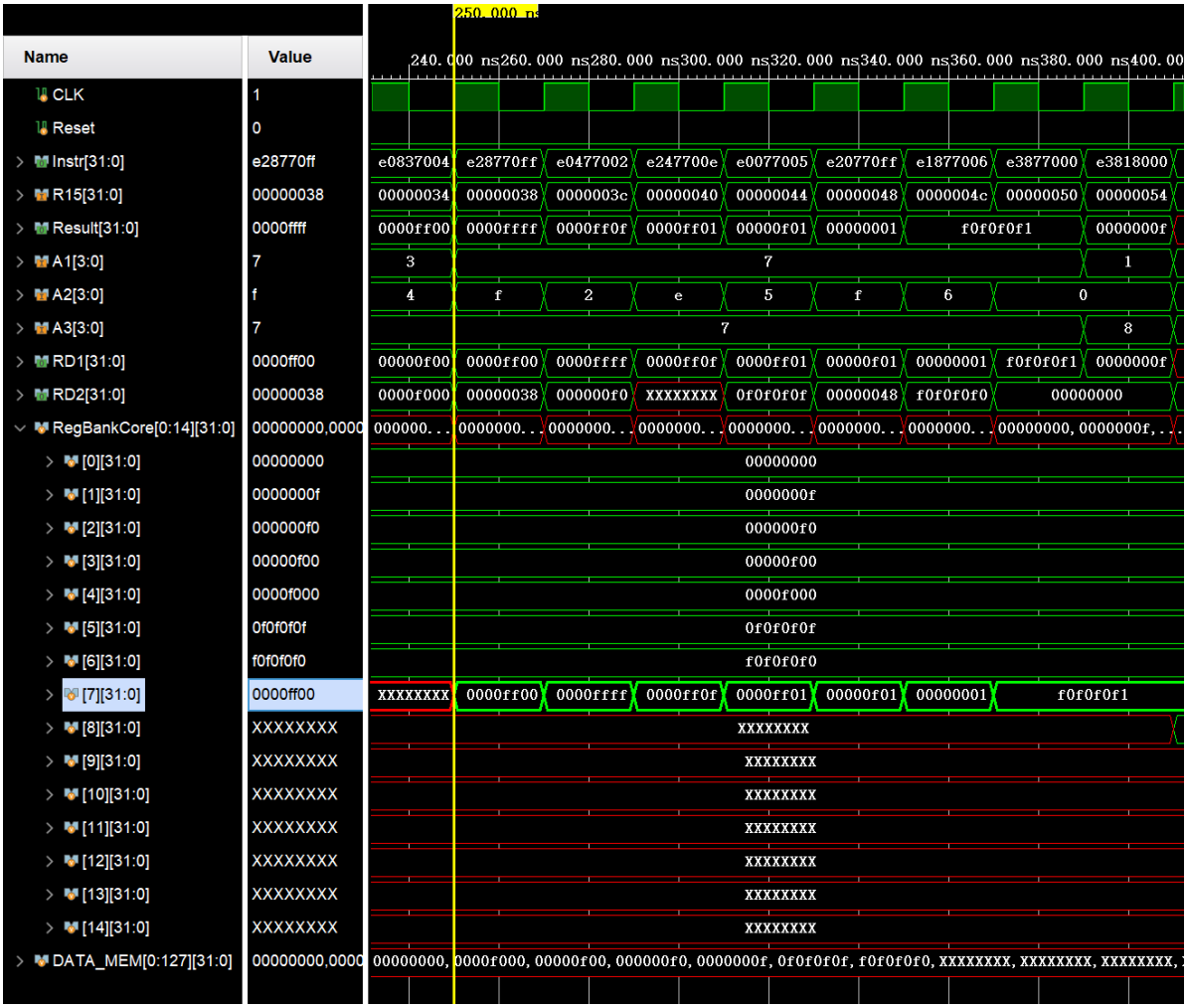
The testbench for Data-Path operations, namely the `ADD`, `SUB`, `AND` and `ORR` instructions, consists of a sequence of instructions, each depends on the result from its preceding instruction. Both immediate number and register as `Src2` are included.

```
; Test for DP instructions
ADD R7, R3, R4      ; R7 = 0x0000_FF00
ADD R7, R7, #0xFF   ; R7 = 0x0000_FFFF
SUB R7, R7, R2      ; R7 = 0x0000_FF0F
SUB R7, R7, #0xE    ; R7 = 0x0000_FF01
AND R7, R7, R5      ; R7 = 0x0000_0F01
AND R7, R7, #0xFF   ; R7 = 0x0000_0001
ORR R7, R7, R6      ; R7 = 0xF0F0_F0F1
ORR R7, R7, #0      ; R7 = 0xF0F0_F0F1
```

The corresponding HEX value is

```
0x0000002C  E0837004  ADD       R7,R3,R4
0x00000030  E28770FF  ADD       R7,R7,#0x000000FF
0x00000034  E0477002  SUB       R7,R7,R2
0x00000038  E247700E  SUB       R7,R7,#0x0000000E
0x0000003C  E0077005  AND       R7,R7,R5
0x00000040  E20770FF  AND       R7,R7,#0x000000FF
0x00000044  E1877006  ORR       R7,R7,R6
0x00000048  E3877000  ORR       R7,R7,#0x00000000
0x0000004C  E3818000  ORR       R8,R1,#0x00000000
```

The simulation waveform is shown below.



According to the waveform form of signal `RegBankCore` and `Result[31:0]` from 240ns to 420ns, the final result, as well as each intermediate result of the DP instructions are expected.

Testbench for Branch Instruction

The testbench for branch instruction consists of a loop for increment of one register and decrement of another.

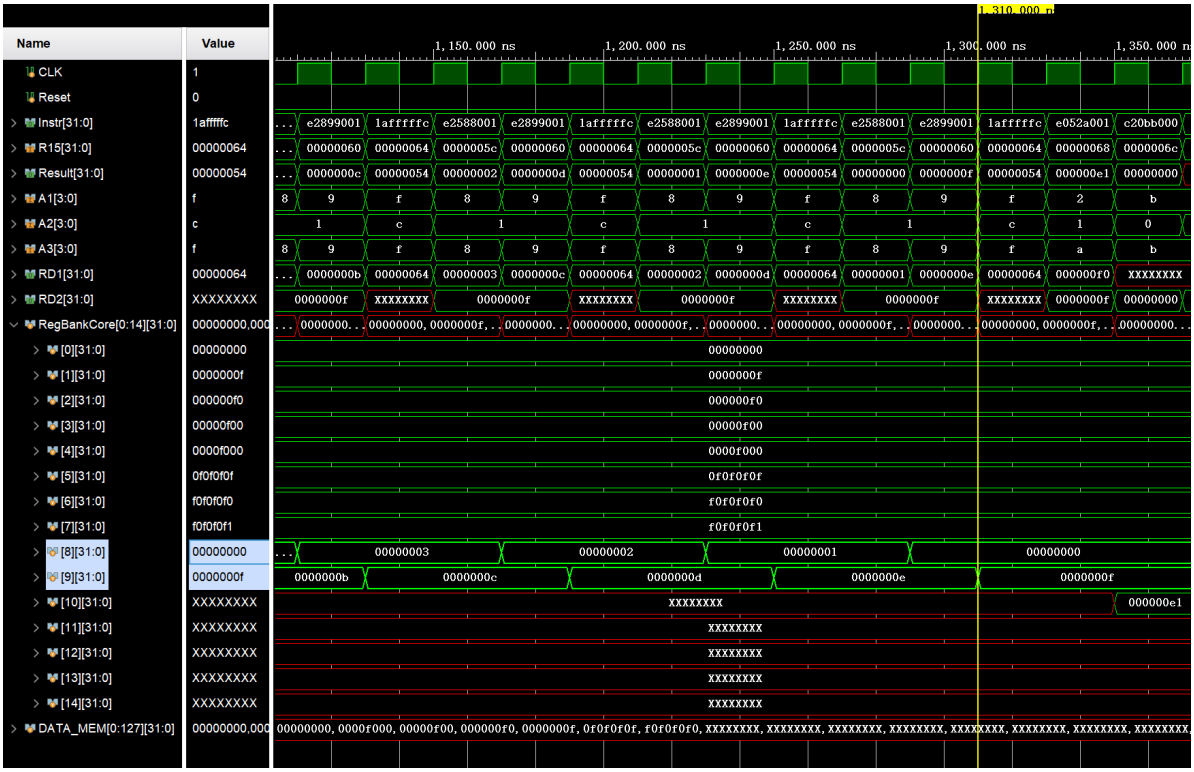
```
    ; Test for Branch instruction
    ORR R8, R1, #0    ; R8 = R1 = 0xF
    AND R9, R9, #0    ; R9 = 0x0
LOOP
    SUBS R8, R8, #1   ; Update R8 and set flags, R8--
    ADD R9, R9, #1    ; Update R9, R9++
    BNE LOOP          ; LOOP until R8 = 0
```

After the loop exits, `R8` should decrease to `0`, while `R9` takes the original value of `R8` before the loop.

The corresponding HEX value is

```
0x00000058  E2899000  AND       R9,R9,#0x00000000
0x00000054  E2588001  SUBS      R8,R8,#0x00000001
0x00000058  E2899001  ADD       R9,R9,#0x00000001
0x0000005C  1AFFFFFC  BNE       0x00000054
```

The simulation waveform is shown below.



According to the waveform of signal `RegBankCore`, the loop is executed and exited correctly.

Testbench for Flags and Condition Logic

Since the amount of possible conditions for instructions is too large, we here only consider the `HI`, `LS`, `GT`, and `LE` conditions and thus have covered all 4 of the `N`, `Z`, `C`, and `V` ALU flags.

cond	Mnemonic	Name	CondEx
1000	HI	Unsigned higher	!Z & C
1001	LS	Unsigned lower or same	Z | !C
1100	GT	Signed greater than	!Z & !(N ^ V)
1101	LE	Signed less than or equal	Z | (N ^ V)

The assembly instructions is shown in the code blocks below.

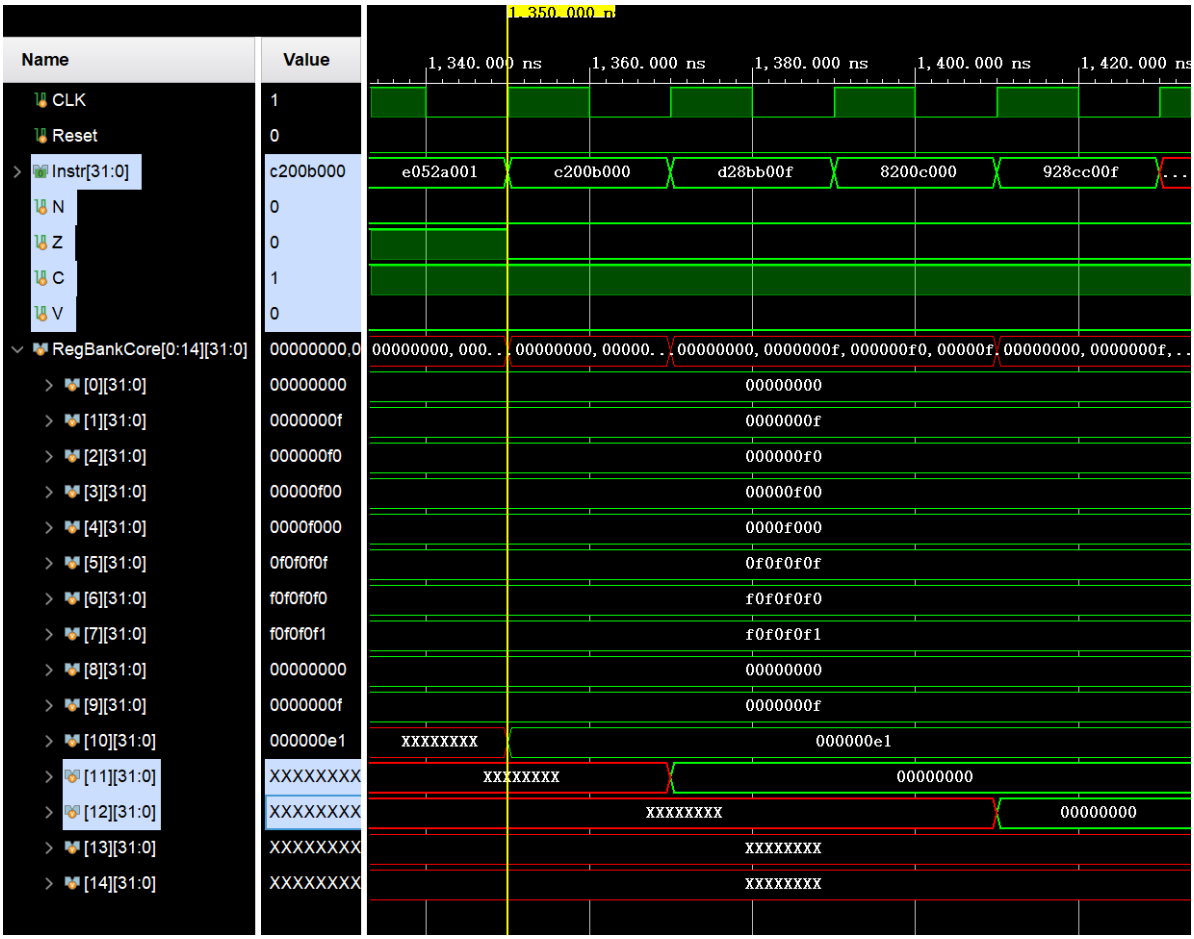
```
1   ; Test for flags and condition
2   SUBS R10, R2, R1
3   ANDGT R11, R0, #0       ; GT should be TRUE, R11 = 0
4   ADDLE R11, R11, #0xF    ; LE should be FALSE, R11 = 0
5   ANDHI R12, R0, #0       ; HI should be TRUE, R12 = 0
6   ADDLS R12, R12, #0xF    ; LS should be FALSE, R12 = 0
```

Ideally, since `R2` is greater than `R1` in both signed and unsigned arithmetics, the `SUBS` instruction would set the ALU flags as `NZCV = 0000`, and thus `GT` and `HI` hold true, `LE` and `LS` should be false.

The corresponding HEX value is

```
1    0x00000060  E052A001  SUBS      R10,R2,R1
2    0x00000064  C200B000  ANDGT     R11,R0,#0x00000000
3    0x00000068  D28BB00F  ADDLE     R11,R11,#0x0000000F
4    0x0000006C  8200C000  ANDHI     R12,R0,#0x00000000
5    0x00000070  928CC00F  ADDLS     R12,R12,#0x0000000F
```

The simulation waveform is shown below.



According to the waveform, the values stored in the Flag registers are correct, and the conditioned instructions are executed as expected.

Conclusion

Since the testbench has covered a large percentage of the categories of instructions, we can conclude that this implementation of single-cycle processor functions normally, though some minor bugs might take place in certain extreme cases.