Date:7/3/2022

Fall Semester 2021-2022

ECE5014 – ASIC Design

M.Tech VLSI Design

School of Electronics Engineering

Vellore Institute of Technology

Name: Shreyas S Bagi

Register Number: 21MVD0086

Slot: L3+L4

Date:7/3/2022

Lab Task 01

Design Specification and Architecture Design of Simple Processor

Section 1 Block Diagram

Aim: Write the Architecture Design of Simple Processor.

Specification:

A digital system that contains a number of 9-bit registers, a multiplexer, an adder/subtractor unit, and a control unit (finite state machine). Data is input to this system via the 9-bit DIN input. This data can be loaded through the 9-bit wide multiplexer into the various registers, such as R0,….R7 and A. The multiplexer also allows data to be transferred from one register to another. The multiplexer's output wires are called a bus in the figure because this term is often used for wiring that allows data to be transferred from one location in a system to another. Addition or subtraction is performed by using the multiplexer to first place one 9-bit number onto the bus wires and loading this number into register A. Once this is done, a second 9-bit number is placed onto the bus, the adder/subtractor unit performs the required operation, and the result is loaded into register G. The data in G can then be transferred to one of the other registers as required.

The system can perform different operations in each clock cycle, as governed by the control unit. This unit determines when particular data is placed onto the bus wires and it controls which of the registers is to be loaded with this data? For example, if the control unit asserts the signals R0out and Ain, then the multiplexer will place the contents of register R0 onto the bus and this data will be loaded by the next active clock edge into register A. A system like this is often called a processor. It executes operations specified in the form of instructions.

Table below lists the instructions that the processor has to support for this exercise. The left column shows the name of an instruction and its operand. The meaning of the syntax Rx [Ry] is that the contents of register Ry are loaded into register Rx. The mv (move) instruction allows data to be copied from one register to another. For the mvi (move immediate) instruction the expression Rx D indicates that the 9-bit constant D is loaded into register Rx.

Operation	Function performed
mv Rx,Ry	$Rx \leftarrow [Ry]$
mvi Rx,#D	$Rx \leftarrow D$
add Rx, Ry	$Rx \leftarrow [Rx] + [Ry]$
sub Rx, Ry	$Rx \leftarrow [Rx] - [Ry]$

Table 1: Instructions performed in the processor.

Each instruction can be encoded and stored in the IR register using the 9-bit format IIIXXXYYY, where III represents the instruction, XXX gives the Rx register, and YYY gives the Ry register. Although only two bits are needed to encode our four instructions, we are using three bits because other instructions will be added to the processor in later parts of this exercise. Instructions are loaded from an external source; hence IR has to be connected to the nine bits of the DIN input, as indicated in Figure 1. For the mvi instruction the YYY field has no meaning, and the immediate data #D has to be supplied on the 9-bit DIN input after the mvi instruction word is stored into IR. Some instructions, such as an addition or subtraction, take more than one clock cycle to complete, because multiple transfers have to be performed across the bus. The finite state machine in the control unit "steps through" such instructions, asserting the control signals needed in successive clock cycles until the instruction has completed. The processor starts executing the instruction on the DIN input when the Run signal is asserted and the processor asserts the Done output when the instruction is finished.

Table 2 indicates the control signals that can be asserted in each time

	T_1	T_2	T_3
(mv): I_0	RY_{out}, RX_{in}, $Done$		
(mvi): I_1	DIN_{out}, RX_{in}, $Done$		
(add): I_2	RX_{out}, A_{in}	RY_{out}, G_{in}	G_{out}, RX_{in}, $Done$
(sub): I_3	RX_{out}, A_{in}	RY_{out}, G_{in}, $AddSub$	G_{out}, RX_{in}, $Done$

Table 2: Control signals asserted in each instruction/time step.

Date:7/3/2022

Architecture Design of Simple Processor:

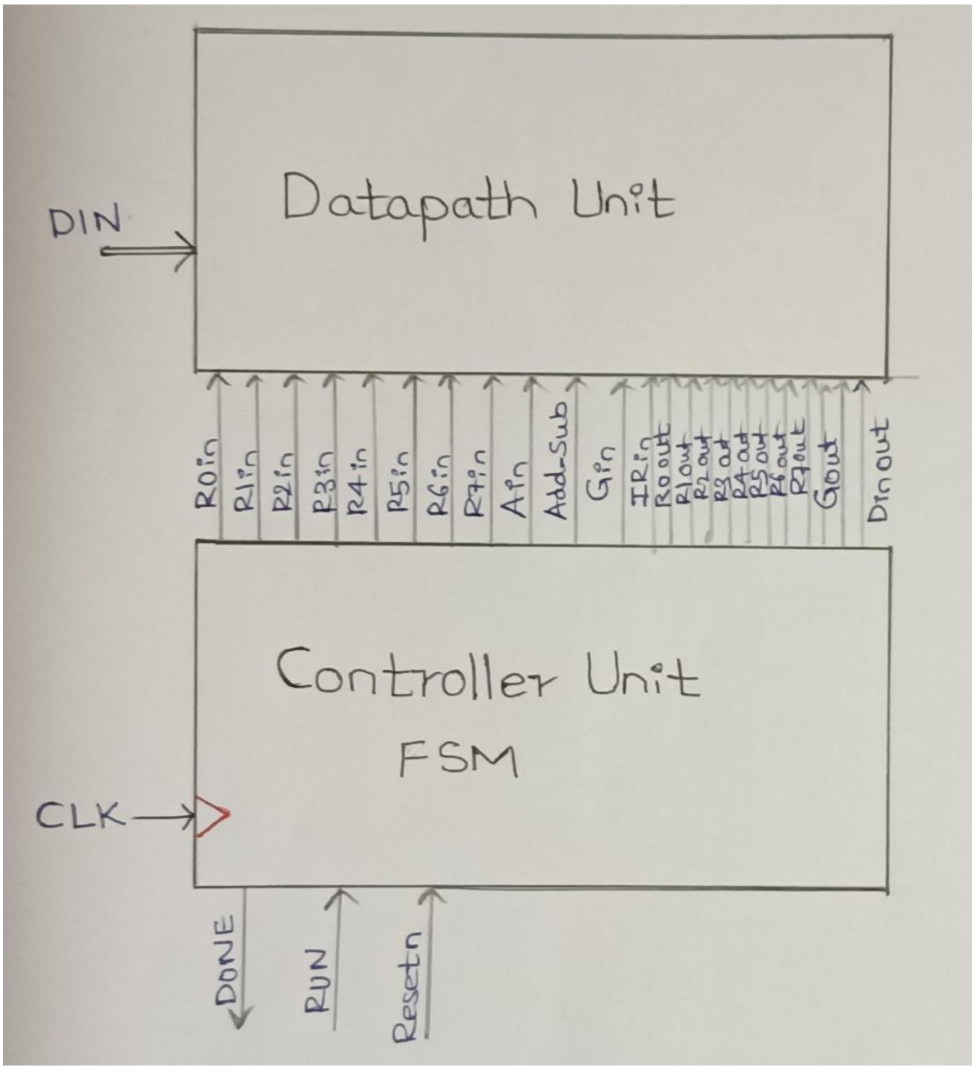


Figure 1.1 The Block diagram of Simple processor showing Datapath and Control path block.

Date:7/3/2022

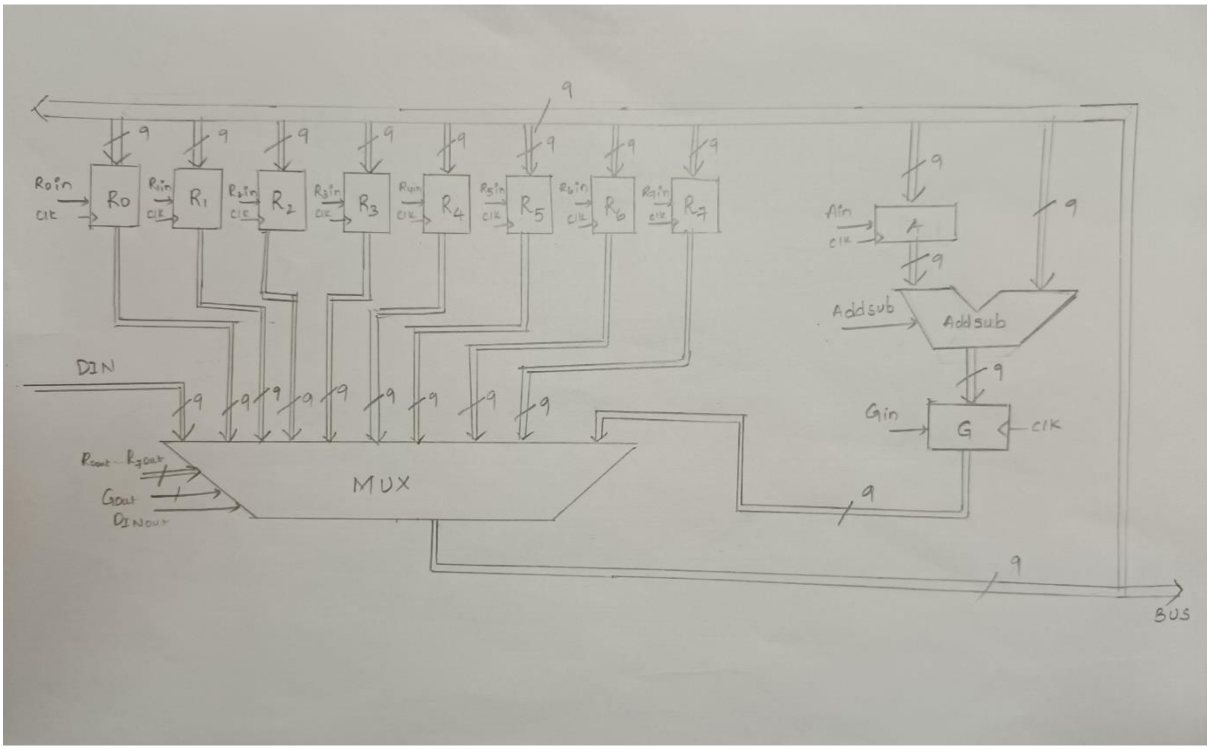
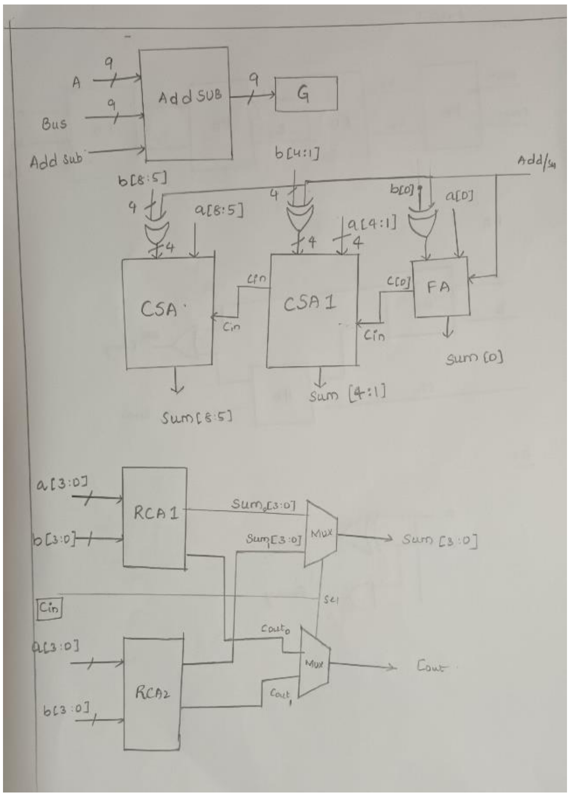


Figure 1.2 The Block diagram of Simple processor showing Datapath block.



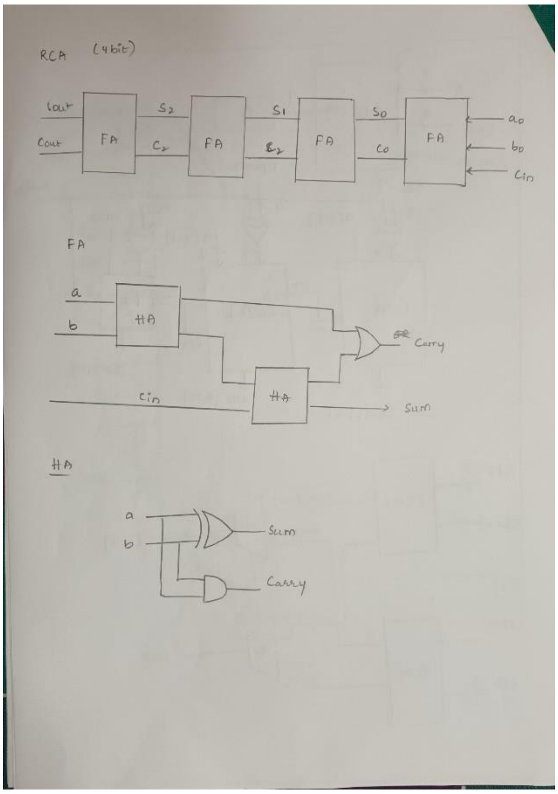


Figure 1.3 The Block diagram of Adder and Subtractor block.

Date:7/3/2022

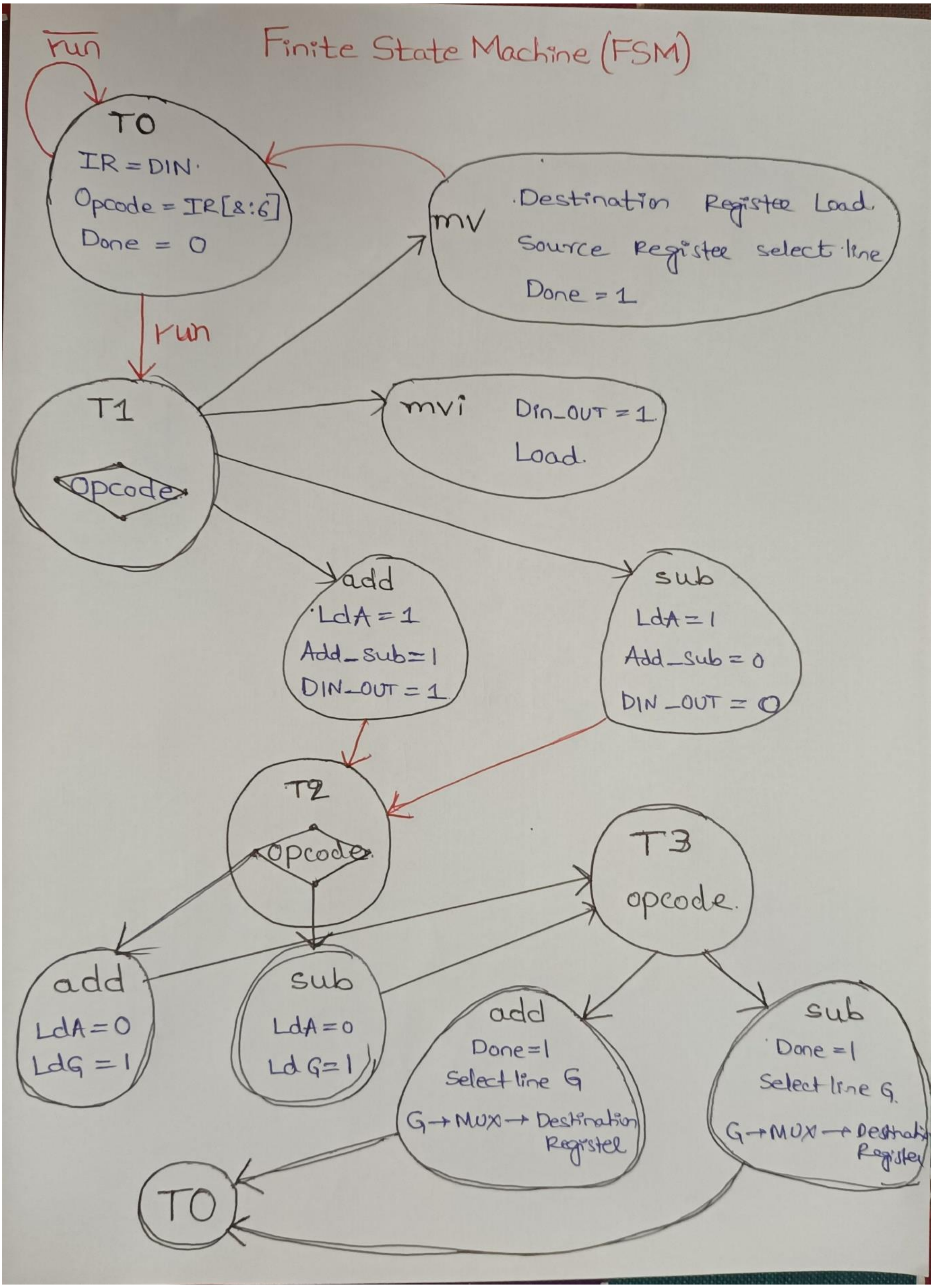


Figure 1.4 The Finite State Machine for Controller Block.

Date:7/3/2022

RTL Coding:

1. **Control path Verilog Code**

```
module controller_new (DIN, Run, Resetn, Clock, Done, R0out, R1out,
R2out, R3out, R4out, R5out,
R6out, R7out, Gout, DINout, LdR0, LdR1, LdR2, LdR3, LdR4, LdR5, LdR6,
LdR7, LdA, LdG, Add_sub);
input Run, Resetn, Clock;
input [8:0] DIN;
output reg LdR0, LdR1, LdR2, LdR3, LdR4, LdR5, LdR6, LdR7, LdA, LdG,
Add_sub;
output reg R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, Gout,
Done, DINout;
wire [9:0] Xreg, Yreg;
reg [9:0] Sel; //output DINout
reg [3:0] PS,NS;
reg [8:0] IR; // Instruction Register
// One - hot Encoding
localparam T0 = 4'b0001, T1 = 4'b0010, T2 = 4'b0100, T3 = 4'b1000; //One
hot Encoding
localparam mv = 3'b000, add = 3'b001, sub = 3'b010, mvi = 3'b011;
reg IRin;
// T0 is for loading state
// If mv instruction then go T1 and RYout and LdRx enabled Control signals
go high
// [8:0] IR
// [8,7,6] IR ------> Opcode
// [5,4,3] IR ------> RX
// [2,1,0] IR ------> RY

wire [2:0] opcode_decoded;
reg [2:0] opcode;


// add R1,R2
//001 001 010 ----> Din at Time t=0
// State is in T0
//opcode = Data_reg [8:6];

dec3to8 Z1 (IR[5:3],Xreg); // RX register -----> Xreg is Destination Register
dec3to8 Z2 (IR[2:0],Yreg); // RY register -----> Yreg is Source Register
dec3to8_3bit Z3 (IR[8:6], opcode_decoded); // Opcode decoding


// Current State Logic Generation
always @(posedge Clock)
```

```
begin
if(Resetn)
PS <= NS;
else
PS <= T0;
end


always @(posedge Clock)
begin
if(IRin)
opcode <= opcode_decoded;
else
opcode <= opcode;
end

// Next State Generation Logic
always @(PS,Run,Done) // removed Run or Done signal in senstivity list.
begin
case(PS) // Use only Blocking statements.
T0: NS = Run? T1:T0;

T1: begin   // Destination Register Load Signal.
            case(opcode)
                        mv   : NS =T0; // Change go back to stage T0 not to be in T1.
                        add  : NS =T2;
                        sub  : NS =T2;
                        mvi  : NS = T2;
                        default :NS = T0;
             endcase
       end

T2:begin  // Source Register Load Signal.
            case(opcode)
                        add  :  NS=T3;
                        sub  :  NS=T3;
                        mvi  :  NS = T0;
                        default : NS = T0;
            endcase
       end

 T3:begin  // Source Register Load Signal.
            case(opcode)
                        add  : NS=T0;
                        sub  :  NS=T0;
                        default : NS = T0;
```

```
            endcase
        end
default : NS = T0;
endcase
end


// Output Generation always block that is Control Signal Generation Logic
always @(PS)
begin
case(PS) // USe only Blocking statements.
T0: begin
        // Initial Reset all the Load Signals
        LdR0 = 0; LdR1 = 0; LdR2 = 0; LdR3 = 0; LdR4 =0;  LdR5 = 0;  LdR6
= 0;  LdR7 = 0;LdA = 0; LdG =0;
        IRin = 1;
        if(!Run)
                begin
                  Done = 0;
                end
            else
                    begin
                    IR   = DIN;
                    Done = 0;
                    end
      end
T1: begin   // Destination Register Load Signal.
            IRin = 0;
      if (opcode == mv)
      begin
      // Ryout corresponding Reg should be high
      // Rx in corresponding Reg should be high
      // Done 1
      // mv R1,R2   R1<----- R2
      Sel = Yreg;
     {R0out,  R1out,  R2out,  R3out,  R4out,  R5out,  R6out,  R7out,  Gout,
DINout} = Sel;
            {LdR0,LdR1,LdR2,LdR3,LdR4,LdR5,LdR6,LdR7} = Xreg[9:2];
            // changed to reg from sel
       Done = 1;
      end

     else if(opcode == add)
      begin
      // Ryout corresponding Reg should be high
      // Rx in corresponding Reg should be high
```

```
       // Done 1
       // add R1,R2   R1<----R1+R2
       Sel = Xreg;
       {R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, Gout,
DINout} = Sel;
       LdA = 1;
       LdG = 0;
      // A = BusWires;
       Add_sub = 1; // 1 for add
       end

      else if(opcode == sub)
      begin
       Sel = Xreg;
       {R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, Gout,
DINout} = Sel;
       LdA = 1;
       LdG = 0;
       //A <= BusWires;
       Add_sub = 0; // 1 for sub
      end

      else if(opcode == mvi)
      begin
      Sel = 10'b0000000001; // Sel = Din out
     {R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, Gout, DINout}
= Sel;
            {LdR0,LdR1,LdR2,LdR3,LdR4,LdR5,LdR6,LdR7} = Xreg[9:2];

     end
     end

T2:begin  // Source Register Load Signal.
            //IRin = 0;
              if(opcode == add)
       begin
       Sel = Yreg;
       {R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, Gout,
DINout} = Sel;
       LdA = 0;
       LdG = 1;
       //G = Z; // Z is result of addition from the adder block;
       end

       else if(opcode == mvi)
       begin
```

```
      Sel = 10'b0000000001; // Sel = Din out
      {R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, Gout,
DINout} = Sel;
              {LdR0,LdR1,LdR2,LdR3,LdR4,LdR5,LdR6,LdR7} = Xreg[9:2];
      Done = 1;
      end

     else if(opcode == sub)
     begin
      Sel = Yreg;
      {R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, Gout,
DINout} = Sel;
      LdA = 0;
      LdG = 1;
     end
             end

T3: begin
            //IRin = 0;
      if(opcode == add)
      begin
      Done = 1;
      Sel=10'b0000000010; // Data from G Register to Destination Register.
Gout=1
      {R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, Gout,
DINout} = Sel;
              {LdR0,LdR1,LdR2,LdR3,LdR4,LdR5,LdR6,LdR7} = Xreg[9:2];

      end

     else if(opcode == sub)
     begin
     Done = 1;
     Sel=10'b0000000010; // Data from G Register to Destination Register.
     {R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, Gout,
DINout} = Sel;
             {LdR0,LdR1,LdR2,LdR3,LdR4,LdR5,LdR6,LdR7} = Xreg[9:2];

     end
     end
default : begin //IRin = 0;
              end
endcase
end

endmodule
```

```
/*
module dec3to8 (W, Y);
input [2:0] W;
output reg [9:0] Y;

always @(W)
begin
case (W)
3'b000: Y = 10'b1000000000;
3'b001: Y = 10'b0100000000;
3'b010: Y = 10'b0010000000;
3'b011: Y = 10'b0001000000;
3'b100: Y = 10'b0000100000;
3'b101: Y = 10'b0000010000;
3'b110: Y = 10'b0000001000;
3'b111: Y = 10'b0000000100;
//3'b1000: Y = 10'b0000000010;
endcase
end
endmodule


module dec3to8_3bit (W, Y);
input [2:0] W;
output reg [2:0] Y;

always @(W)
begin
case (W)
3'b000: Y = 3'b000;
3'b001: Y = 3'b001;
3'b010: Y = 3'b010;
3'b011: Y = 3'b011;
3'b100: Y = 3'b100;
3'b101: Y = 3'b101;
3'b110: Y = 3'b110;
3'b111: Y = 3'b111;
default: Y = 3'b000;
endcase
end
endmodule
```

2. Datapath Verilog Code

```
/*

File_Name : Data_path_work

revision: 2

Last_ Modified : 12/03/2021

*/

//`timescale 1ns/1ns

module datapath_register_array (R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, Gout, DINout,

Clock,rst,R0in, R1in,R2in,R3in,R4in,R5in,R6in,R7in,Ain,Bus,DIN,AddSub,Gin);


input R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, Gout, DINout, Clock,R0in, R1in,R2in,R3in,R4in,R5in,R6in,R7in,Ain,rst,AddSub,Gin;

input [8:0] DIN;

output [8:0] Bus;

wire [8:0] R0_data_out, R1_data_out, R2_data_out, R3_data_out, R4_data_out;

wire [8:0] R5_data_out, R6_data_out, R7_data_out, A_data_out, Sum,G;


mux_10to1 m1 (.R0out(R0out), .R1out(R1out), .R2out(R2out), .R3out(R3out), .R4out(R4out), .R5out(R5out), .R6out(R6out), .R7out(R7out), .Gout(Gout), .DINout(DINout),.DIN(DIN), .R0(R0_data_out),.R1(R1_data_out), .R2(R2_data_out), .R3(R3_data_out), .R4(R4_data_out), .R5(R5_data_out),.R6(R6_data_out), .R7(R7_data_out), .G(G), .Bus(Bus)); //Priority Multiplexer


Add_Sub add_top (.A(A_data_out), .Bus(Bus), .AddSub(AddSub),.ALU_out(Sum)); //Add_Sub (Aout, Bus, AddSub,ALU_out);

reg_G g2 (.Sum(Sum),.Gin(Gin), .Clock(Clock), .rst(rst), .Z(G));


Register Reg0 (Clock, R0_data_out, Bus, R0in,rst);

Register Reg1 (Clock, R1_data_out, Bus, R1in,rst);

Register Reg2 (Clock, R2_data_out, Bus, R2in,rst);

Register Reg3 (Clock, R3_data_out, Bus, R3in,rst);
```

```
Register Reg4 (Clock, R4_data_out, Bus, R4in,rst);
Register Reg5 (Clock, R5_data_out, Bus, R5in,rst);
Register Reg6 (Clock, R6_data_out, Bus, R6in,rst);
Register Reg7 (Clock, R7_data_out, Bus, R7in,rst);
Register A5   (Clock, A_data_out,   Bus, Ain,rst);
endmodule



module mux_10to1 (R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, Gout, DINout,
DIN, R0,R1, R2, R3, R4, R5,R6, R7, G, Bus);

input R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, Gout, DINout;
input [8:0] DIN, R0,R1, R2, R3, R4, R5,R6, R7, G;
output reg [8:0] Bus;
wire [9:0] sel; //selection line of mux_10to1

assign sel = {R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, Gout, DINout};
always @ (*)
 begin
   if (sel == 10'b0000000001)
         Bus = DIN;
    else if (sel == 10'b0000000010)
         Bus = G;
    else if (sel == 10'b0000000100)
         Bus = R7;
            else if (sel == 10'b0000001000)
         Bus = R6;
            else if (sel == 10'b0000010000)
         Bus = R5;
```

```
        else if (sel == 10'b0000100000)
      Bus = R4;
        else if (sel == 10'b0001000000)
      Bus = R3;
        else if (sel == 10'b0010000000)
      Bus = R2;
        else if (sel == 10'b0100000000)
      Bus = R1;
        else if (sel == 10'b1000000000)
      Bus = R0;
   else
      Bus = Bus;
 end
endmodule

//For register
module Register (Clock,dout,din,EN,rst);
input Clock;
input EN,rst;
input [8:0] din;
output reg [8:0] dout;

always @(posedge Clock)
begin
if(EN)
        dout <= din;
else if(~rst)
        dout <= 9'b0;
end
endmodule
```

```
//ADD_SUB
module Add_Sub (A, Bus, AddSub,ALU_out);
input [8:0] A, Bus;
input AddSub;
output [8:0] ALU_out;
wire [8:0] xrout;
wire Cout;

xor (xrout[0],Bus[0],AddSub), (xrout[1],Bus[1],AddSub),
(xrout[2],Bus[2],AddSub), (xrout[3],Bus[3],AddSub),
    (xrout[4],Bus[4],AddSub), (xrout[5],Bus[5],AddSub),
(xrout[6],Bus[6],AddSub), (xrout[7],Bus[7],AddSub),
    (xrout[8],Bus[8],AddSub);
csa_9bit add1 (.a(A), .b(xrout),.cin(AddSub) , .sum(ALU_out), .cout(Cout));

endmodule

module reg_G(Sum,Gin, Clock,rst, Z);
input Gin;
input [8:0] Sum;
input Clock, rst;
output reg [8:0] Z;
always @ (posedge Clock)
begin
if (~rst)
begin
Z <= 9'b0;
end

else
```

```
begin
if(Gin == 1)
Z <= Sum;
else
Z <= Z;
end
end
endmodule



module csa_9bit(a, b,cin, sum, cout);
//In main module a= A, b= Bus , c= AddSub
//module carry_select_adder_16bit(a, b, cin, sum, cout);
input [8:0] a,b;
input cin;
output [8:0] sum;
output cout;

wire [1:0] c;
// assign {cout,sum} = a+b+cin;
//full adder
full_adder fa0(.a(a[0]),.b(b[0]),.cin(cin),.sum(sum[0]),.cout(c[0]));
carry_select_adder_4bit_slice
csa_slice1(.a(a[4:1]),.b(b[4:1]),.cin(c[0]),.sum(sum[4:1]),.cout(c[1]));
carry_select_adder_4bit_slice
csa_slice2(.a(a[8:5]),.b(b[8:5]),.cin(c[1]),.sum(sum[8:5]), .cout(cout));

endmodule

////////////////////////////////////
//4-bit Carry Select Adder Slice
```

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

module carry_select_adder_4bit_slice(a, b, cin, sum, cout);
input [3:0] a,b;
input cin;
output [3:0] sum;
output cout;

wire [3:0] s0,s1;
wire c0,c1;

ripple_carry_4_bit rca1(.a(a[3:0]),.b(b[3:0]),.cin(1'b0),.sum(s0[3:0]),.cout(c0));
ripple_carry_4_bit rca2(.a(a[3:0]),.b(b[3:0]),.cin(1'b1),.sum(s1[3:0]),.cout(c1));

mux2X1 ms0(.in0(s0[3:0]),.in1(s1[3:0]),.sel(cin),.out(sum[3:0]));
mux2X1_1 mc0(.in0(c0),.in1(c1),.sel(cin),.out(cout));
endmodule

////////////////////
//2X1 Mux
////////////////////

module mux2X1( in0,in1,sel,out);
//parameter width=16;
input [3:0] in0,in1;
input sel;
output [3:0] out;
assign out=(sel)?in1:in0;
endmodule
```

```
 module mux2X1_1( in0,in1,sel,out);
//parameter width=16;
input  in0,in1;
input sel;
output  out;
assign out=(sel)?in1:in0;
endmodule
////////////////////////////////
//4-bit Ripple Carry Adder
////////////////////////////////
module ripple_carry_4_bit(a, b, cin, sum, cout);
input [3:0] a,b;
input cin;
output [3:0] sum;
output cout;
 wire c1,c2,c3;
 full_adder fa0(.a(a[0]),.b(b[0]),.cin(cin),.sum(sum[0]),.cout(c1));
full_adder fa1(.a(a[1]),.b(b[1]),.cin(c1),.sum(sum[1]),.cout(c2));
full_adder fa2(.a(a[2]),.b(b[2]),.cin(c2),.sum(sum[2]),.cout(c3));
full_adder fa3(.a(a[3]),.b(b[3]),.cin(c3),.sum(sum[3]),.cout(cout));
endmodule

////////////////////
//1bit Full Adder
////////////////////

module full_adder(a,b,cin,sum,cout);
input a,b,cin;
output sum, cout;
wire x,y,z;
```

```
half_adder h1(.a(a), .b(b), .sum(x), .cout(y));
half_adder h2(.a(x), .b(cin), .sum(sum), .cout(z));
or or_1(cout,z,y);
endmodule

//////////////////////
// 1 bit Half Adder
//////////////////////`timescale 1ns / 1ns
module half_adder( a,b, sum, cout );
input a,b;
output sum, cout;
xor xor_1 (sum,a,b);
and and_1 (cout,a,b);
endmodule
```

3. Top Module Code

```
module simple_processor_Top (Run, Resetn, Clock, DIN, Bus, Done);
input Run, Resetn, Clock;
input [8:0] DIN;
output [8:0] Bus;
output Done;

wire R0out, R1out, R2out, R3out, R4out, R5out, R6out, R7out, Gout,
DINout, LdR0, LdR1, LdR2, LdR3, LdR4, LdR5, LdR6, LdR7, LdA, LdG,
Add_sub;

controller_new G1_Controller (DIN, Run, Resetn, Clock, Done, R0out,
R1out, R2out, R3out, R4out, R5out, R6out, R7out, Gout, DINout, LdR0,
LdR1, LdR2, LdR3, LdR4, LdR5, LdR6, LdR7, LdA, LdG, Add_sub);

datapath_register_array G2_Datapath (R0out, R1out, R2out, R3out, R4out,
R5out, R6out, R7out, Gout, DINout, Clock, Resetn, LdR0, LdR1, LdR2,
LdR3, LdR4, LdR5, LdR6, LdR7, LdA, Bus, DIN, Add_sub, LdG);

endmodule
```

Testbench Codes based on Operations

1. Move Immediate Operation

```
`timescale 1ns/1ns
module simple_processor_testbench ();
reg Run, Resetn, Clock;
reg [8:0] DIN;
wire [8:0] Bus;
wire Done;

controller_new G1 (DIN, Run, Resetn, Clock, Done, R0out, R1out,
R2out, R3out, R4out, R5out,
R6out, R7out, Gout, DINout, LdR0, LdR1, LdR2, LdR3, LdR4, LdR5,
LdR6, LdR7, LdA, LdG, Add_sub);

datapath_register_array G2 (R0out, R1out, R2out, R3out, R4out, R5out,
R6out, R7out, Gout, DINout, Clock, Resetn, LdR0,
LdR1, LdR2, LdR3, LdR4, LdR5, LdR6, LdR7, LdA, Bus, DIN,
Add_sub, LdG);

initial
Clock = 1'b1;

always #5 Clock = ~Clock;
```

```
initial
begin

$set_toggle_region(simple_processor_testbench.G1,
simple_processor_testbench.G2);
$toggle_start();
// ...

Resetn = 1'b0;
#10 Resetn = 1'b1;
Run = 1'b1;

DIN = 9'b011000001; // mvi operation
#20 DIN = 9'b111001111; // immediate data
//R0 is loaded with 111001111;
#5;

#20 DIN = 9'b011010001; // mvi operation
#20 DIN = 9'b111111111; // immediate data after 50ns from 1st instuction
@t=10
// R2 is loaded with 111111111;

#30 DIN = 9'b011001001; // mvi operation
#20 DIN = 9'b101010101; // immediate data
//R1 is loaded with 101010101;
#5;

#30 DIN = 9'b011011001; // mvi operation
#20 DIN = 9'b101010111; // immediate data
//R3 is loaded with 101010101;
#5;

#30 DIN = 9'b011100001; // mvi operation
#20 DIN = 9'b111101111; // immediate data
//R4 is loaded with 101010101;
#5;

#30 DIN = 9'b011101001; // mvi operation
#20 DIN = 9'b110110110; // immediate data
//R5 is loaded with 101010101;
#5;

#30 DIN = 9'b011110001; // mvi operation
#20 DIN = 9'b111011011; // immediate data
//R6 is loaded with 101010101;
#5;
```

```
#30 DIN = 9'b011111001; // mvi operation
#20 DIN = 9'b111111111; // immediate data
//R7 is loaded with 101010101;

// ...
$toggle_stop();
$toggle_report("Simple_Processor_SAIF_Move_Immediate_Operation.
saif", 1.0e-12, "simple_processor_testbench");
#5; $stop;

Run = 1'b0;
end

initial
 begin

/*
// mvi R1,R5
#2 DIN = 9'b011001010; // mvi operation
#30 DIN = 9'b111001111; // immediate data
Run = 0;
#10 Run = 1;
DIN = 9'b000001010;
#1000 $stop;
*/

end
endmodule
```

Date:7/3/2022

Waveforms:

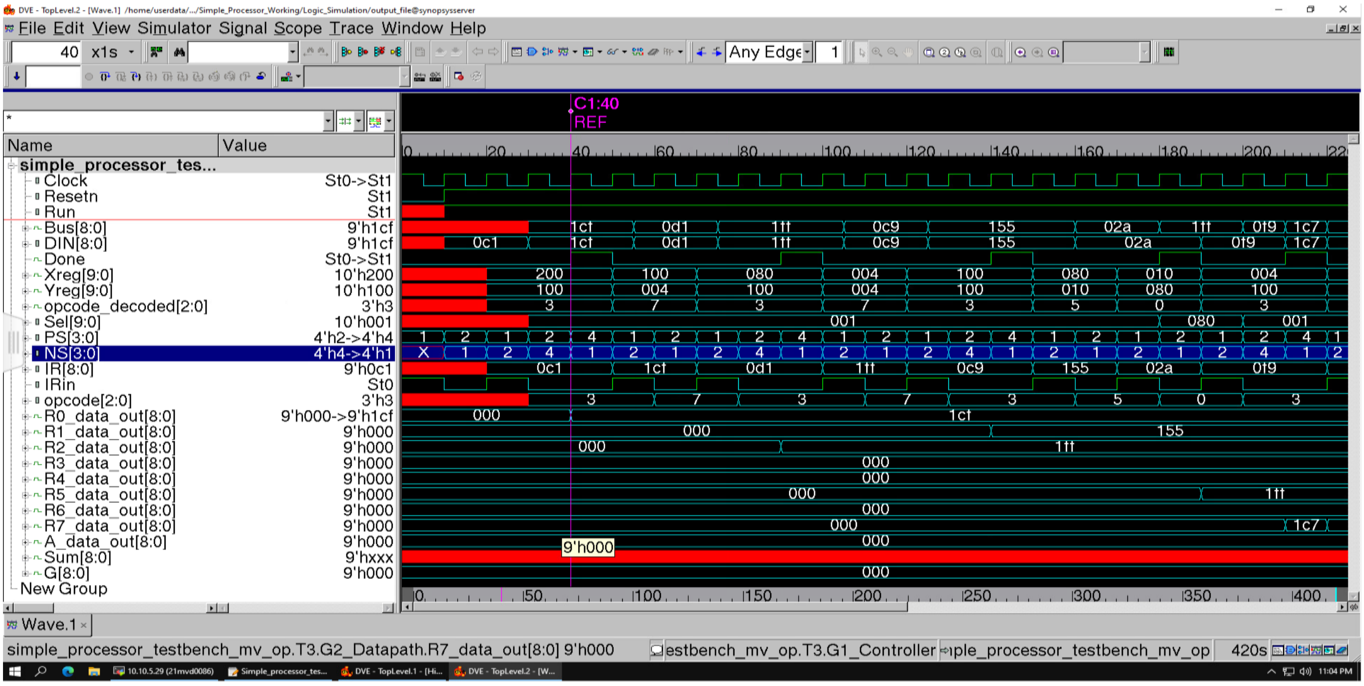


Figure 1.1: The data stored in Register and Move Immediate Operation.

In the Figure 1.1 we are doing Move immediate operation. At time instant from t=0ns to 40ns we observe that Register R0 is stored with 0 value and after 40ns we store the data 9'h1CF data. The flow of data is like this at 10ns the (2^{nd} Clock pulse), Resetn = 1 and Run = 1. These conditions are met then only start the operation of processor. The Move immediate operation has three states T0 to T1 to T2. In these states the operation occurs. At time t=40ns we observe R0 changes data. Similarly In Figure 1.2 we observe the move operation taking place for R2 register and R1 register.

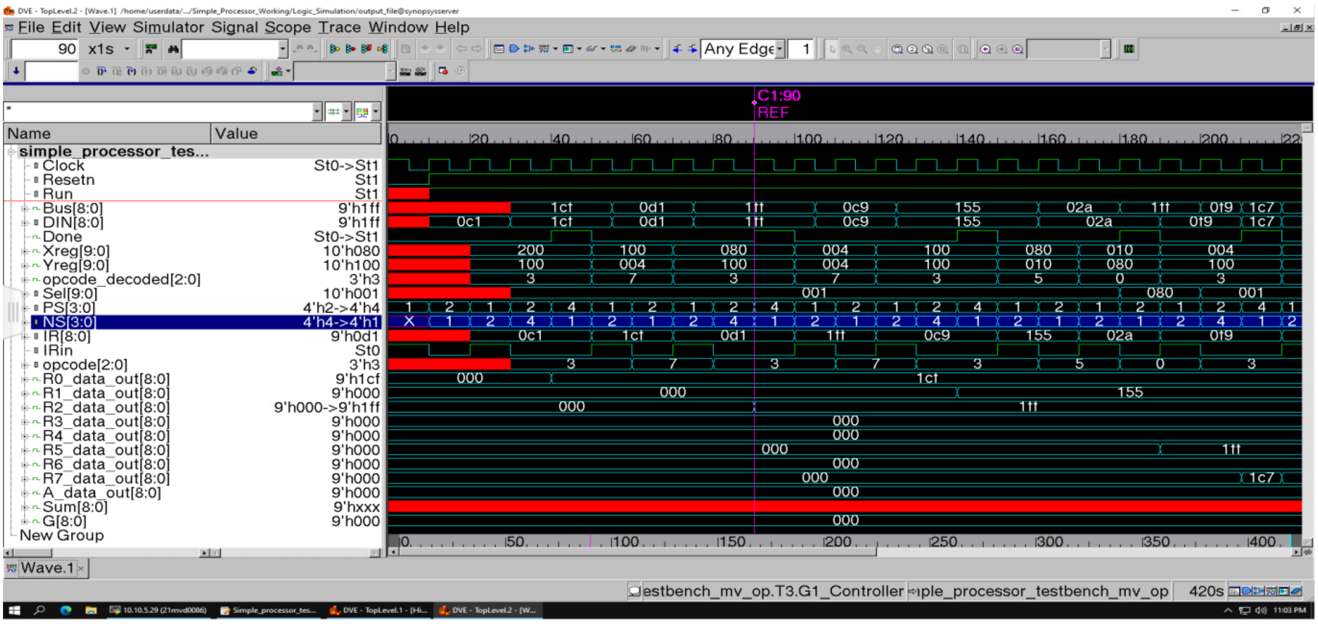


Figure 1.2: The data stored in Register and Move Immediate Operation.

2. Move Operation

```
`timescale 1ns/1ns
module simple_processor_testbench_mv_op ();
reg Run, Resetn, Clock;
reg [8:0] DIN;
wire [8:0] Bus;
wire Done;

controller_new U1 (DIN, Run, Resetn, Clock, Done, R0out, R1out,
R2out, R3out, R4out, R5out,
R6out, R7out, Gout, DINout, LdR0, LdR1, LdR2, LdR3, LdR4, LdR5,
LdR6, LdR7, LdA, LdG, Add_sub);

datapath_register_array U2 (R0out, R1out, R2out, R3out, R4out, R5out,
R6out, R7out, Gout, DINout,
Clock,rst,R0in,
R1in,R2in,R3in,R4in,R5in,R6in,R7in,Ain,Bus,DIN,AddSub,Gin);


initial
Clock = 1'b1;

always #5 Clock = ~Clock;

initial
begin
$set_toggle_region(simple_processor_testbench_mv_op.U1,
simple_processor_testbench_mv_op.U2);
$toggle_start();
// ...

Resetn = 1'b0;
#10 Resetn = 1'b1;
Run = 1'b1;

DIN = 9'b011_000_001; // mvi operation
#20 DIN = 9'b111_001_111; // immediate data
//R0 is loaded with 111001111;
#5;

#20 DIN = 9'b011_010_001; // mvi operation
#20 DIN = 9'b111_111_111; // immediate data after 50ns from 1st
instuction @t=10
// R2 is loaded with 111111111;
```

```
#30 DIN = 9'b011_001_001; // mvi operation
#20 DIN = 9'b101_010_101; // immediate data
//R1 is loaded with 101010101;
#5;

#30 DIN = 9'b000_101_010;

#30 DIN = 9'b011_111_001; // mvi operation
#20 DIN = 9'b111_000_111; // immediate data
//R1 is loaded with 101010101;
#10 DIN = 9'b000_001_111;

// ...
$toggle_stop();
$toggle_report("Simple_Processor_SAIF_Move_Operation.saif",    1.0e-
12, "simple_processor_testbench_mv_op");

#200 $stop;
end

endmodule
```

Date:7/3/2022

Waveforms:

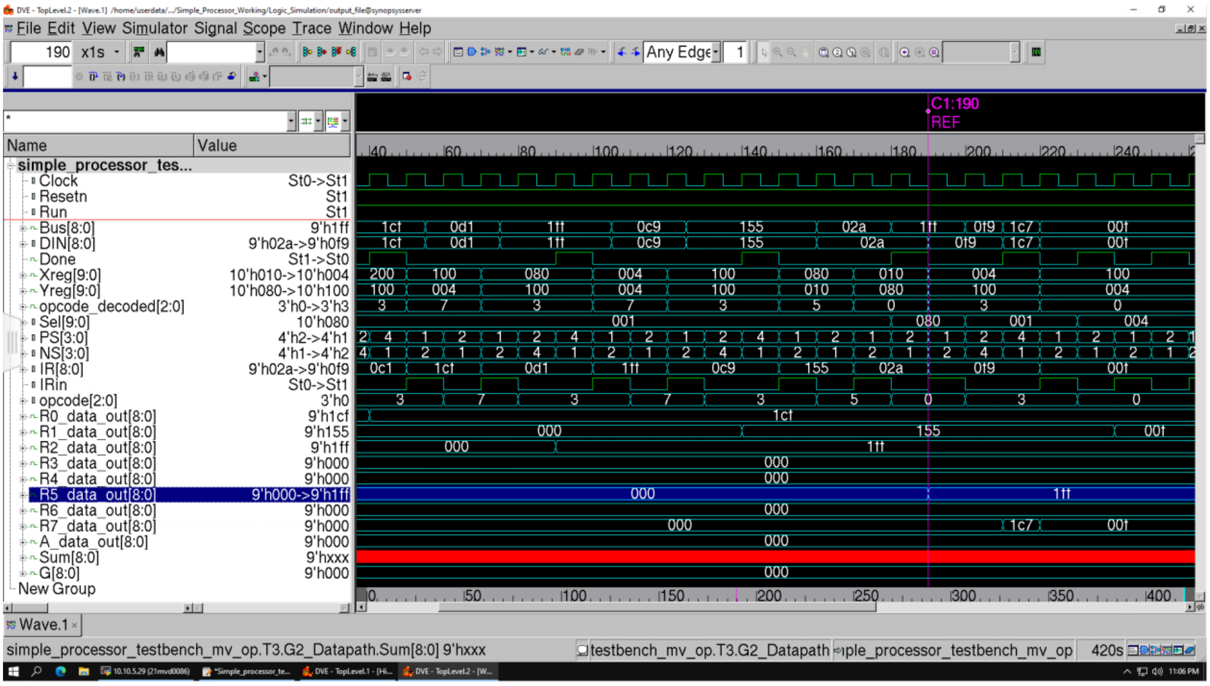


Figure 1.3: The Move operation taking place from R2 to R5.

In the Figure 1.3 we are doing Move operation. We have done the Move immediate operation for Register R0, R1 and R2 already. At time instant from t=170ns to 190ns we observe that. The Move operation takes two states (two clock pulses) to complete the operation. We observe the data stored in Register R2 is copied to Register R5

3. Addition Operation

```
`timescale 1ns/1ns
module simple_processor_testbench_add_op();
reg Run, Resetn, Clock;
reg [8:0] DIN;
wire [8:0] Bus;
wire Done;

controller_new G1 (DIN, Run, Resetn, Clock, Done, R0out, R1out,
R2out, R3out, R4out, R5out,
R6out, R7out, Gout, DINout, LdR0, LdR1, LdR2, LdR3, LdR4, LdR5,
LdR6, LdR7, LdA, LdG, Add_sub);

datapath_register_array G2 (R0out, R1out, R2out, R3out, R4out, R5out,
R6out, R7out, Gout, DINout, Clock, Resetn, LdR0,
LdR1, LdR2, LdR3, LdR4, LdR5, LdR6, LdR7, LdA, Bus, DIN,
Add_sub, LdG);

initial
Clock = 1'b1;

always #5 Clock = ~Clock;

initial
begin
$set_toggle_region(simple_processor_testbench_add_op.G1,
simple_processor_testbench_add_op.G2);
$toggle_start();
// ...

Resetn = 1'b0;
#10 Resetn = 1'b1;
Run = 1'b1;

DIN = 9'b011_000_001; // mvi operation
#20 DIN = 9'b111_001_111; // immediate data
//R0 is loaded with 111001111;
#5;

/*
#20 DIN = 9'b011_010_001; // mvi operation
#20 DIN = 9'b111_111_111; // immediate data after 50ns from 1st
instuction @t=10
// R2 is loaded with 111111111;
*/
```

```
/*
#30 DIN = 9'b011_001_001; // mvi operation
#20 DIN = 9'b101_010_101; // immediate data
//R1 is loaded with 101010101;
*/
#40 DIN = 9'b001_000_010;
#20 Run =0;

// ...
$toggle_stop();
$toggle_report("Simple_Processor_SAIF_ADD_Operation.saif",1.0e-
12, "simple_processor_testbench_add_op");

#10 $stop;
end
endmodule
```

Date:7/3/2022

Waveforms:

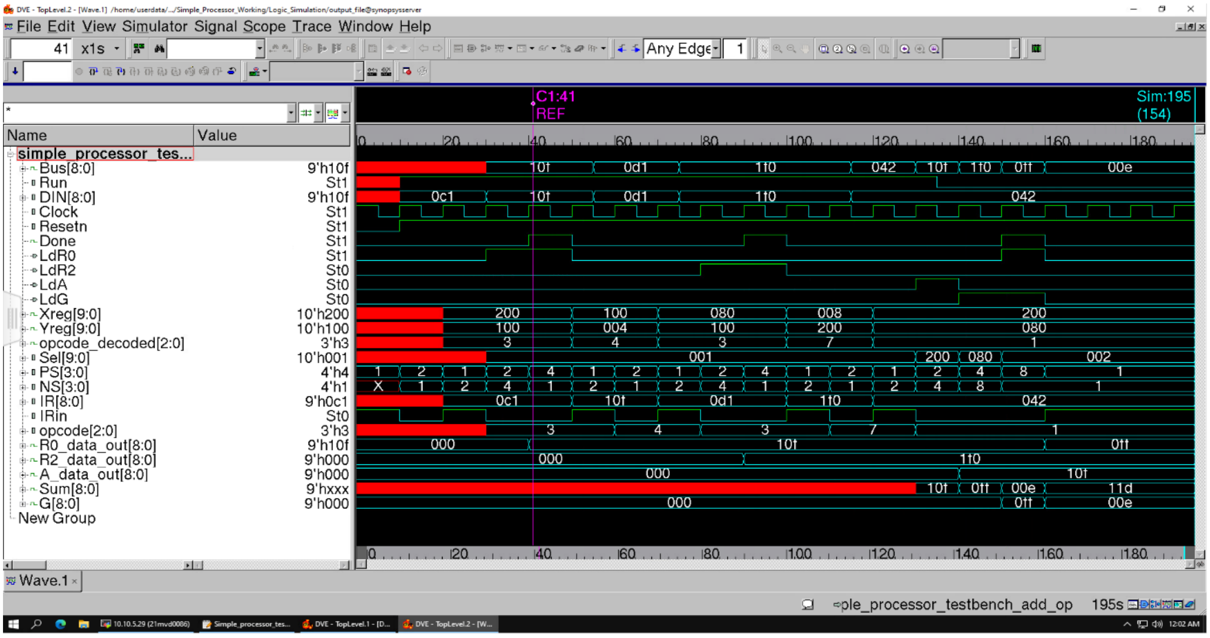


Figure 1.4: The Load operation for R0.

In the figure we observe that the data 9'h10F loaded to register R0. Observe from time instant from 0 to 40ns no data loaded. At t=40ns we observe that R0 is loaded with the data 9'h10F. We are doing Move immediate operation for Register R0.

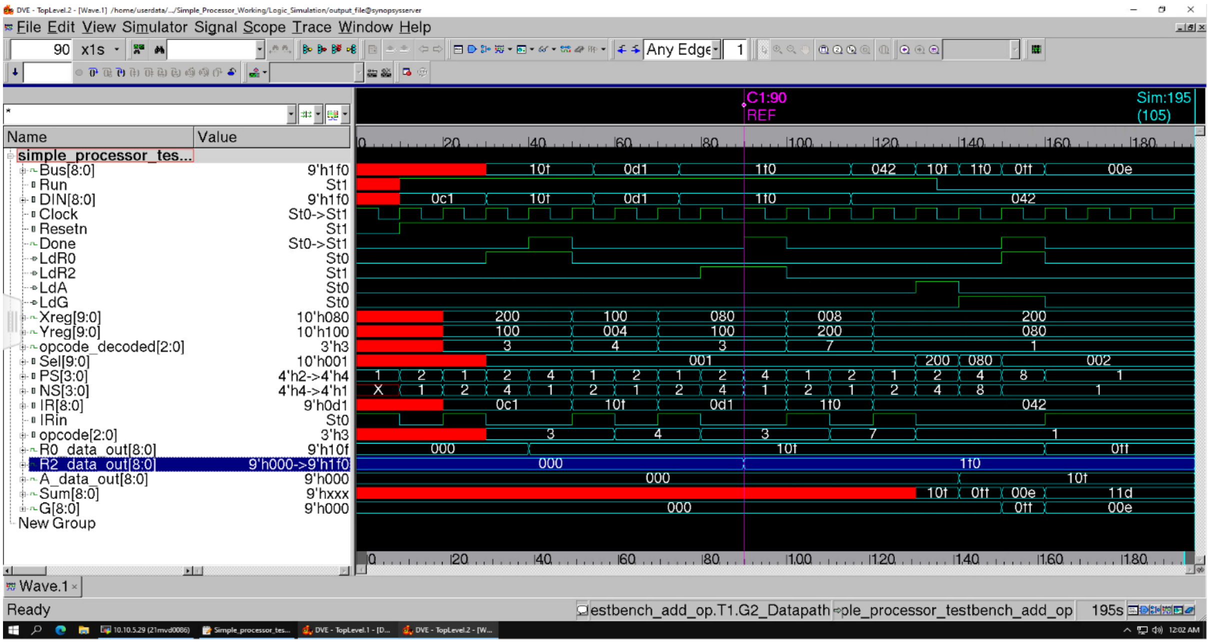


Figure 1.5: The Load operation for R2.

In the figure we observe that the data 9'h1F0 loaded to register R2. Observe from time instant from 0 to 90ns no data loaded. At t=90ns we observe that R2 is loaded with the data 9'h1F0. We are doing Move immediate operation for Register R2.

Date:7/3/2022

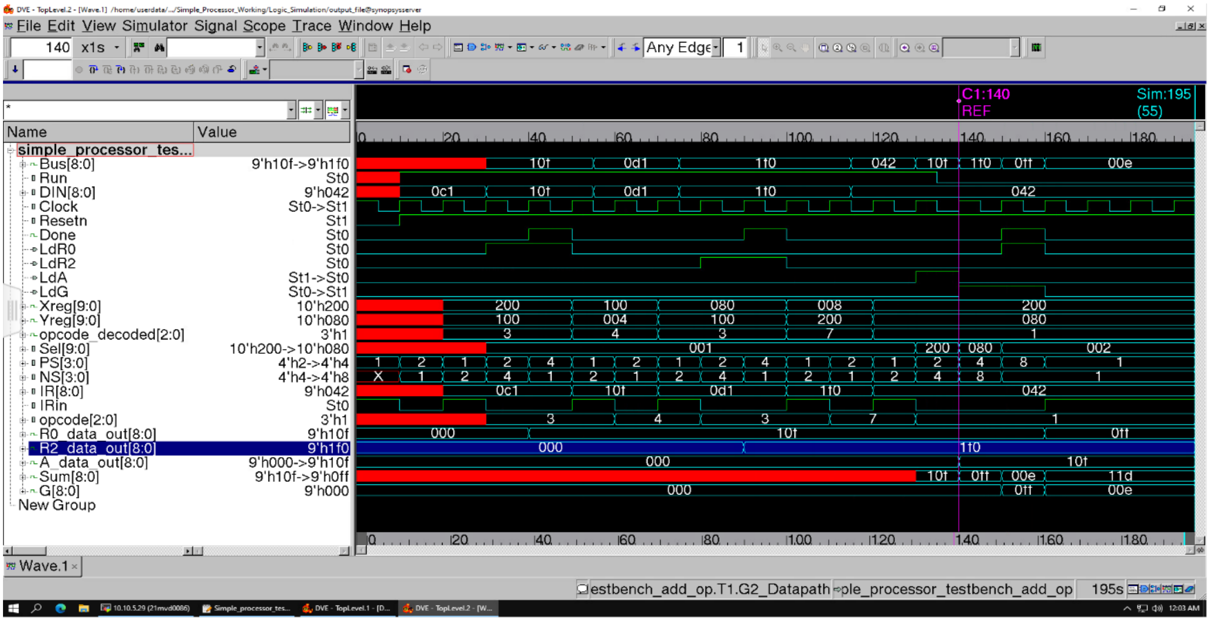


Figure 1.6: The Addition operation opcode decoding.

In the Figure 1.6 we observe that the R0 data value is copied to Register A. The control signal LdA is generated high and goes low after that, the addition takes place with bus loaded with R2 data. In next clock pulse we observe that the LdG signal became high and addition result is stored to Register G.

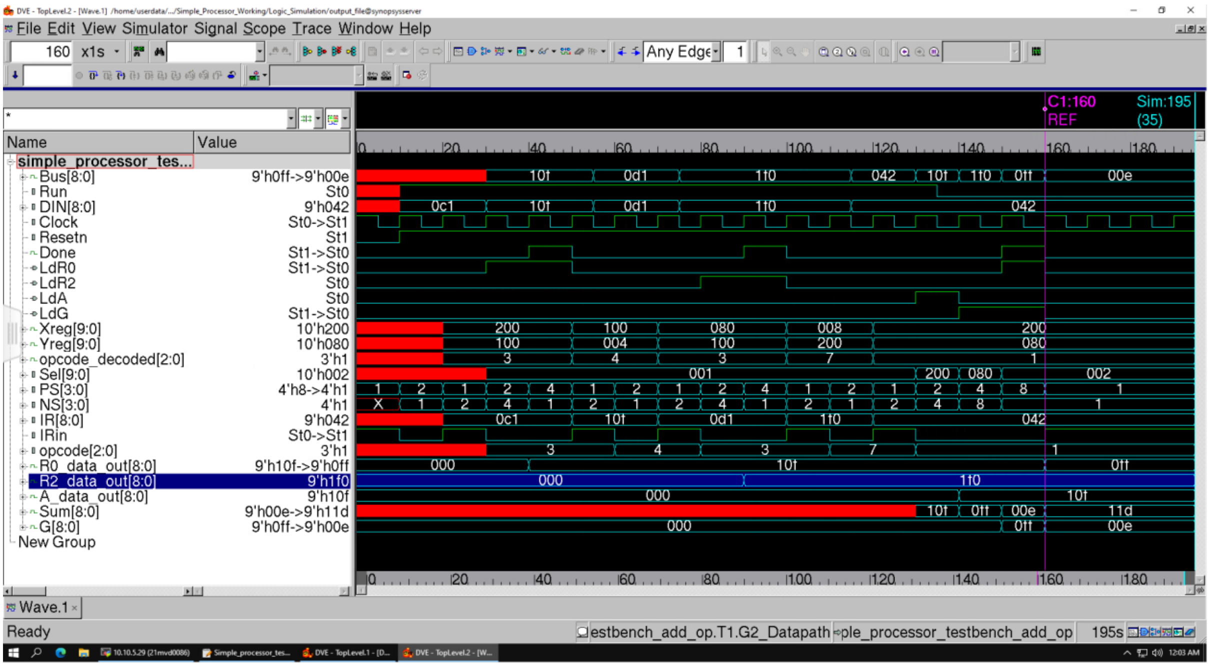


Figure 1.7: The Addition operation opcode decoding.

In the Figure 1.7 we observe that sum result stored Register G is stored back to register R0.

Date:7/3/2022

4. Subtraction Operation

```
`timescale 1ns/1ns
module simple_processor_testbench_add_op();
reg Run, Resetn, Clock;
reg [8:0] DIN;
wire [8:0] Bus;
wire Done;

simple_processor_Top T1 (Run,Resetn,Clock,DIN,Bus,Done);

initial
Clock = 1'b1;

always #5 Clock = ~Clock;

initial
begin
$set_toggle_region(simple_processor_testbench_add_op.T1);
$toggle_start();
// ...

Resetn = 1'b0;
#10 Resetn = 1'b1;
Run = 1'b1;

DIN = 9'b011_000_001; // mvi operation
#20 DIN = 9'b111_110_000; // immediate data
//R0 is loaded with 111001111;
#5;


#20 DIN = 9'b011_010_001; // mvi operation
#20 DIN = 9'b100_001_111; // immediate data after 50ns from 1st instuction @t=10
// R2 is loaded with 111111111;


/*
#30 DIN = 9'b011_001_001; // mvi operation
#20 DIN = 9'b101_010_101; // immediate data
//R1 is loaded with 101010101;
*/

#40 DIN = 9'b010_000_010; // sub R0=R0-R2
#60 Run = 0;
```

```
// ...
$toggle_stop();
$toggle_report("Simple_Processor_SAIF_ADD_Operation.saif",    1.0e-
12, "simple_processor_testbench_add_op");

#0 $stop;
end
endmodule
```

Date:7/3/2022

Waveforms:

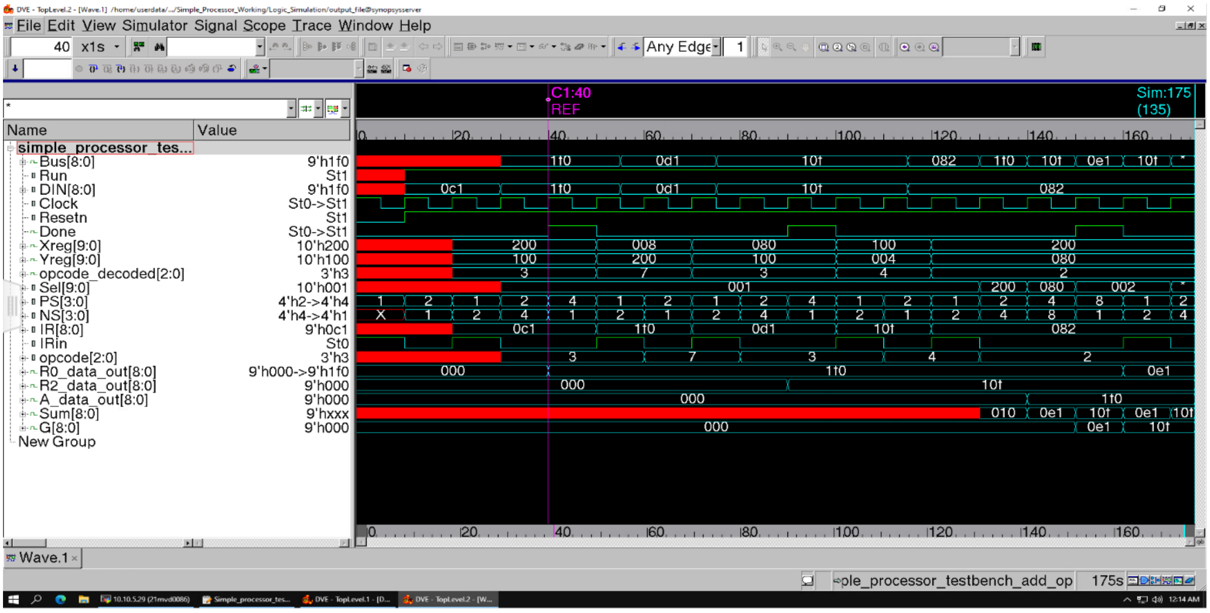


Figure 1.8: The Load operation for R0.

In the Figure 1.8 we observe that the data 9’h1F0 loaded to register R0. Observe from time instant from 0 to 40ns no data loaded. At t=40ns we observe that R0 is loaded with the data 9’h1F0. We are doing Move immediate operation for Register R0.

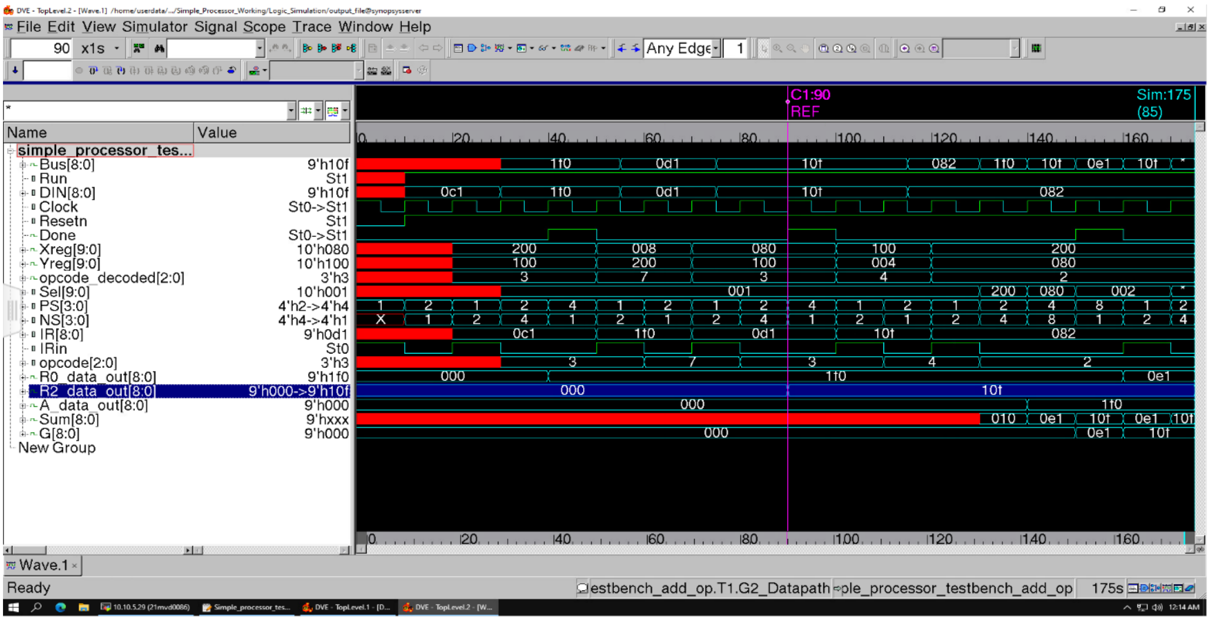


Figure 1.9: The Load operation for R2.

In the Figure 1.9 we observe that the data 9’h10F loaded to register R2. Observe from time instant from 0 to 90ns no data loaded. At t=90ns we observe that R2 is loaded with the data 9’h10F. We are doing Move immediate operation for Register R2.

Date:7/3/2022

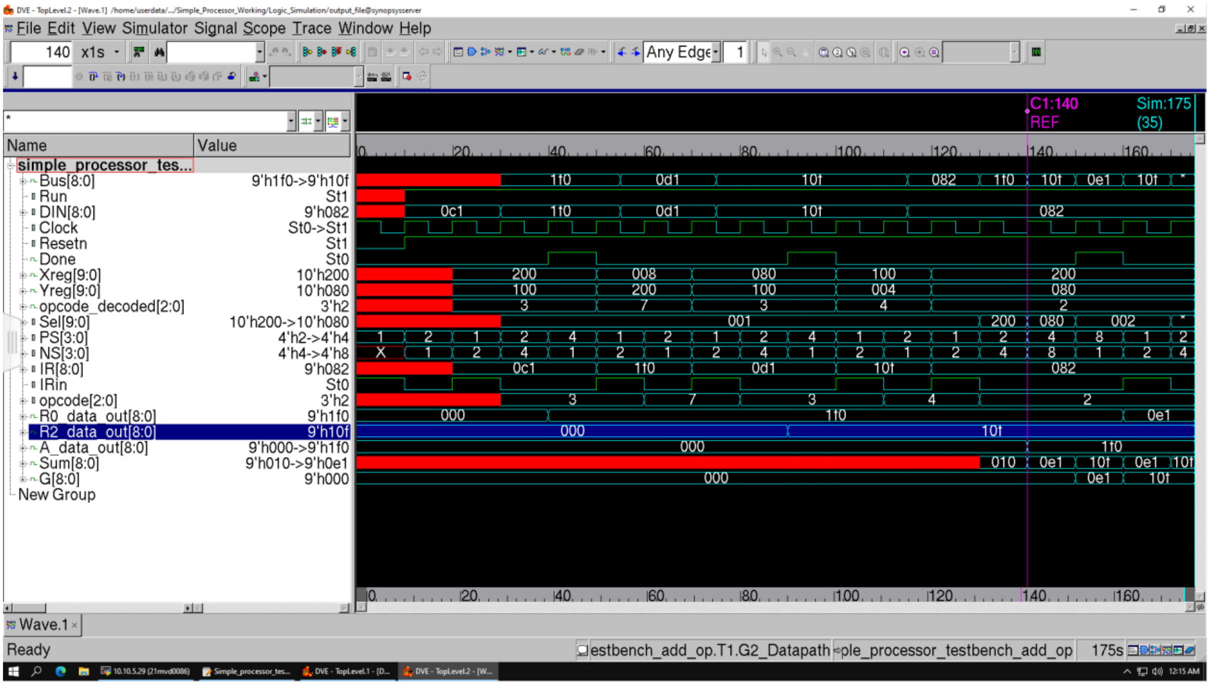


Figure 1.10: The Subtraction operation opcode decoding.

In the Figure 1.10 we observe that the R0 data value is copied to Register A. The control signal LdA is generated high and goes low after that, the subtraction takes place with bus loaded with R2 data. In next clock pulse we observe that the LdG signal became high and subtraction result is stored to Register G.

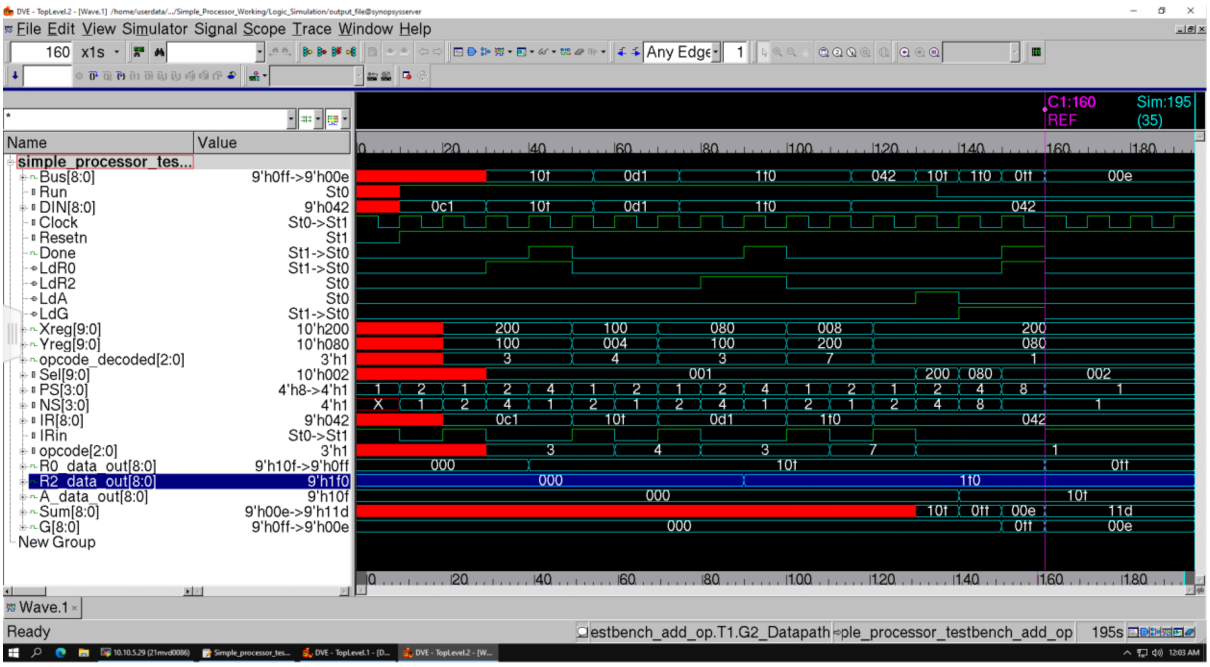


Figure 1.11: The Addition operation opcode decoding.

In the Figure 1.11 we observe that difference result stored Register G is stored back to register R0.

Date:7/3/2022

Inference:

1. The Simple Processor 9 bit was designed using Verilog code. The simulation was done using Synopsys VCS tool.
2. The opcodes used were Move, Move immediate, Addition and Subtraction operation was done and verified.