ELEC 374 Lab

Phase one

Group 25
Andy Zheng Li: 20124884
William Du: 20100436

Section 1: VHDL code and Schematics (Datapath and its component)

datapath.vhd

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_unsigned.all;
4   use work.components_all.all;
5
6   entity datapath is port (
7   clk: in std_logic;
8   encoderin: in std_logic_vector(31 downto 0);
9   register_enable: in std_logic_vector(31 downto 0);
10  Mdatain: in std_logic_vector(31 downto 0);
11  MDR_Read: in std_logic;
12  ALU_sel: in std_logic_vector(4 downto 0);
13  PC_plus: in std_logic;
14
15  BusMuxOut: out std_logic_vector(31 downto 0);
16  R2out    : out std_logic_vector(31 downto 0);
17  R4out    : out std_logic_vector(31 downto 0);
18  R5out    : out std_logic_vector(31 downto 0);
19  HIout    : out std_logic_vector(31 downto 0);
20  LOout    : out std_logic_vector(31 downto 0);
21  IRout    : out std_logic_vector(31 downto 0);
22  Zout: out std_logic_vector(63 downto 0)
23  );
24  end entity;
25
26  architecture behav of datapath is
27
28  signal in_port, out_port: std_logic_vector(31 downto 0);
29  signal BusMuxIn_R0: std_logic_vector(31 downto 0);
30  signal BusMuxIn_R1: std_logic_vector(31 downto 0);
31  signal BusMuxIn_R2: std_logic_vector(31 downto 0);
32  signal BusMuxIn_R3: std_logic_vector(31 downto 0);
33  signal BusMuxIn_R4: std_logic_vector(31 downto 0);
34  signal BusMuxIn_R5: std_logic_vector(31 downto 0);
35  signal BusMuxIn_R6: std_logic_vector(31 downto 0);
36  signal BusMuxIn_R7: std_logic_vector(31 downto 0);
37  signal BusMuxIn_R8: std_logic_vector(31 downto 0);
38  signal BusMuxIn_R9: std_logic_vector(31 downto 0);
39  signal BusMuxIn_R10: std_logic_vector(31 downto 0);
40  signal BusMuxIn_R11: std_logic_vector(31 downto 0);
41  signal BusMuxIn_R12: std_logic_vector(31 downto 0);
42  signal BusMuxIn_R13: std_logic_vector(31 downto 0);
43  signal BusMuxIn_R14: std_logic_vector(31 downto 0);
44  signal BusMuxIn_R15: std_logic_vector(31 downto 0);
45  signal BusMuxIn_HI: std_logic_vector(31 downto 0);
46  signal BusMuxIn_LO: std_logic_vector(31 downto 0);
47  signal BusMuxIn_Zhigh: std_logic_vector(31 downto 0);
48  signal BusMuxIn_Zlow: std_logic_vector(31 downto 0);
49  signal BusMuxIn_PC: std_logic_vector(31 downto 0);
50  signal BusMuxIn_MDR: std_logic_vector(31 downto 0);
51  signal BusMuxIn_Inport: std_logic_vector(31 downto 0);
52  signal C_sign_extended: std_logic_vector(31 downto 0);
53  signal internalBusMuxOut: std_logic_vector(31 downto 0);
54  signal clr: std_logic;
55  signal default_zeros: std_logic_vector(31 downto 0);
56  signal overflow: std_logic;
57  signal MARout: std_logic_vector(31 downto 0);
58  signal IIRout: std_logic_vector(31 downto 0);
59
60
61  begin
62  default_zeros <= (others =>'0');
63  clr<='1';
64  R0  : register32bit port map (internalBusMuxOut, register_enable(0),clr, clk,
    BusMuxIn_R0);
65  R1  : register32bit port map (internalBusMuxOut, register_enable(1),clr, clk,
    BusMuxIn_R1);
66  R2 : register32bit port map (internalBusMuxOut, register_enable(2),clr, clk,
    BusMuxIn_R2);
```

Figure 1:first page of Datapath code

```
67   R3 : register32bit port map (internalBusMuxOut, register_enable(3),clr, clk,
     BusMuxIn_R3);
68   R4 : register32bit port map (internalBusMuxOut, register_enable(4),clr, clk,
     BusMuxIn_R4);
69   R5 : register32bit port map (internalBusMuxOut, register_enable(5),clr, clk,
     BusMuxIn_R5);
70   R6 : register32bit port map (internalBusMuxOut, register_enable(6),clr, clk,
     BusMuxIn_R6);
71   R7 : register32bit port map (internalBusMuxOut, register_enable(7),clr, clk,
     BusMuxIn_R7);
72   R8 : register32bit port map (internalBusMuxOut, register_enable(8),clr, clk,
     BusMuxIn_R8);
73   R9 : register32bit port map (internalBusMuxOut, register_enable(9),clr, clk,
     BusMuxIn_R9);
74   R10: register32bit port map (internalBusMuxOut, register_enable(10),clr,
     clk,BusMuxIn_R10);
75   R11: register32bit port map (internalBusMuxOut, register_enable(11),clr,
     clk,BusMuxIn_R11);
76   R12: register32bit port map (internalBusMuxOut, register_enable(12),clr,
     clk,BusMuxIn_R12);
77   R13: register32bit port map (internalBusMuxOut, register_enable(13),clr,
     clk,BusMuxIn_R13);
78   R14: register32bit port map (internalBusMuxOut, register_enable(14),clr,
     clk,BusMuxIn_R14);
79   R15: register32bit port map (internalBusMuxOut, register_enable(15),clr,
     clk,BusMuxIn_R15);
80   HI : register32bit port map (internalBusMuxOut, register_enable(16),clr,
     clk,BusMuxIn_HI);
81   LO : register32bit port map (internalBusMuxOut, register_enable(17),clr,
     clk,BusMuxIn_LO);
82   PC : register32bit port map (internalBusMuxOut, register_enable(18),clr,
     clk,BusMuxIn_PC);
83   IR  : register32bit port map (internalBusMuxOut, register_enable(19),clr, clk, IIRout);
84   MD_R: MDR port map(MDR_Read, register_enable(20), clr, clk, internalBusMuxOut, Mdatain,
     BusMuxIn_MDR);
85   MAR: register32bit port map (internalBusMuxOut, register_enable(21), clr, clk, MARout);
86   alu_datapath : ALU_path port map (clk, clr, register_enable(22),
     register_enable(23),internalBusMuxOut, internalBusMuxOut, PC_plus, ALU_sel, overflow,
     BusMuxIn_Zlow, BusMuxIn_Zhigh);
87   inport_register: register32bit port map ( in_port, register_enable(24),clr,clk,
     BusMuxIn_Inport);
88   outport_register: register32bit port map (internalBusMuxOut,register_enable(25), clr,
     clk, out_port);
89
90   datapathBus : the_bus port map (BusMuxIn_R0,BusMuxIn_R1, BusMuxIn_R2, BusMuxIn_R3,
91   BusMuxIn_R4, BusMuxIn_R5, BusMuxIn_R6, BusMuxIn_R7, BusMuxIn_R8, BusMuxIn_R9,
     BusMuxIn_R10, BusMuxIn_R11,BusMuxIn_R12, BusMuxIn_R13,
92   BusMuxIn_R14, BusMuxIn_R15, BusMuxIn_HI, BusMuxIn_LO, BusMuxIn_Zhigh,BusMuxIn_Zlow,
     BusMuxIn_PC, BusMuxIn_MDR, BusMuxIn_Inport,C_sign_extended,
93   default_zeros, default_zeros, default_zeros, default_zeros, default_zeros,
     default_zeros, default_zeros, default_zeros, encoderin, internalBusMuxOut);
94
95   BusMuxOut <= internalBusMuxOut;
96   R2out <= BusMuxIn_R2;
97   R4out <= BusMuxIn_R4;
98   R5out <= BusMuxIn_R5;
99   HIout <= BusMuxIn_HI;
100  LOout <= BusMuxIn_LO;
101  Zout(63 downto 32) <= BusMuxIn_Zhigh;
102  Zout(31 downto 0) <= BusMuxIn_Zlow;
103  IRout <= IIRout;
104  end architecture;
```

Figure 2: second page of Datapath code

ALU_path.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use work.components_all.all;

entity ALU_path is port(
clk:in std_logic;
clr:in std_logic;
Y_enable:in std_logic;
Z_enable: in std_logic;
In1: in std_logic_vector(31 downto 0);
In2: in std_logic_vector(31 downto 0);
PC_plus: in std_logic;
ALU_sel: in std_logic_vector(4 downto 0);
overflow: out std_logic;
ToLow:out std_logic_vector(31 downto 0);
ToHi: out std_logic_vector(31 downto 0)
);
end entity ALU_path;

architecture structure of ALU_path is
signal Yout: std_logic_vector(31 downto 0);
signal Zin: std_logic_vector(63 downto 0);
signal ALU_out: std_logic_vector(63 downto 0);
signal PCinc: std_logic_vector(63 downto 0);

begin
register_Y: register32bit port map (In1, Y_enable, clr, clk, Yout);
the_alu: alu port map(Yout, In2, AlU_sel, overflow, ALU_out);
register_Z: register64bit port map(Zin, Z_enable, clr, clk, ToLow, ToHi);

PCinc(31 downto 0)<= std_logic_vector(unsigned(In1)+1); --increment the content of PC
by 1
PCinc(63 downto 32)<= (others=>'0');
Zin<= PCinc when PC_plus ='1' else
        ALU_out; -- the mux between alu and Z

end architecture structure;
```

Figure 3: ALU path code

register32bit.vhd

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_arith.ALL;
Use ieee.std_logic_unsigned.ALL;

entity register32bit is Port(
D : in std_logic_vector (31 downto 0);
ld : in std_logic;
clr : in std_logic;
clk : in std_logic;
Q: out std_logic_vector (31 downto 0));

end entity register32bit;


architecture behav of register32bit is
begin
    process(clk, clr)
        begin
            if clr = '0' then
                Q<= (others=>'0');
            elsif (clk'event and clk='1') then
            if ld = '1' then
                Q <= D;
            end if;
        end if;
    end process;
end behav;
```

Figure 4: 32bit register code

register64bit.vhd

```
1   Library ieee;
2   Use ieee.std_logic_1164.ALL;
3   Use ieee.std_logic_arith.ALL;
4   Use ieee.std_logic_unsigned.ALL;
5
6   entity register64bit is Port(
7   D : in std_logic_vector (63 downto 0);
8   ld : in std_logic;
9   clr : in std_logic;
10  clk : in std_logic;
11  Q0: out std_logic_vector (31 downto 0);
12  Q1: out std_logic_vector (31 downto 0)
13  );
14
15  end entity register64bit; --for register Z
16
17
18  architecture behav of register64bit is
19  begin
20      process(clk, clr)
21          begin
22              if clr = '0' then --active low
23                  Q0<= (others=>'0');
24                  Q1<= (others=>'0');
25              elsif (clk'event and clk ='1') then
26              if ld = '1' then
27                  Q0 <= D(31 downto 0);
28                       Q1 <= D(63 downto 32);
29              end if;
30          end if;
31      end process;
32  end behav;
```

Figure 5: 64bit register code

alu.vhd

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_unsigned.all;
4   --use ieee.numeric_std.all;
5   use work.components_all.all;
6
7   entity alu is port(
8   A : in std_logic_vector(31 downto 0);
9   B : in std_logic_vector(31 downto 0);
10  ALU_sel : in std_logic_vector(4 downto 0);
11  overflow : out std_logic;
12  C : out std_logic_vector(63 downto 0)
13  );
14  end entity alu;
15
16  architecture behav of alu is
17  signal Mulout: std_logic_vector(63 downto 0);
18  signal Divout: std_logic_vector(63 downto 0);
19  signal addorsub: std_logic;
20  signal addsuboverflow: std_logic;
21  signal addsubout: std_logic_vector(31 downto 0);
22  signal rorout:std_logic_vector(31 downto 0);
23  signal rolout:std_logic_vector(31 downto 0);
24  signal shrout: std_logic_vector(31 downto 0);
25  signal shlout: std_logic_vector(31 downto 0);
26  signal negout: std_logic_vector(31 downto 0);
27
28  begin
29  aluMul: booth_logic port map (A, B, Mulout);
30  aluDiv: lpm_divida port map(B, A, Divout(31 downto 0), Divout(63 downto 32));
31  aluAddSub: lpm_add_sua port map (addorsub, A, B, addsuboverflow, addsubout);
32  aluRor: ror32 port map (A, B(4 downto 0), rorout);
33  aluRol: rol32 port map (A, B(4 downto 0), rolout);
34  aluShl: shl32 port map (A, B, shlout);
35  aluShr: shr32 port map (A, B, shrout);
36  aluNeg: negate32 port map(B, negout);
37
38  process(A,B,ALU_sel,Mulout,Divout,addorsub,addsuboverflow,addsubout,negout, shlout,
    shrout, rorout, rolout) is
39
40  begin
41  addorsub <= '0';
42  overflow <= '0';
43
44  if(ALU_sel = "01001")then
45      C(31 downto 0)<= A and B;
46      C(63 downto 32)<=(others=>'0');
47
48  elsif (ALU_sel = "01010") then
49      C(31 downto 0)<= A or B;
50      C(63 downto 32)<=(others=>'0');
51
52  elsif (ALU_sel = "01010") then
53      C(31 downto 0)<= A or B;
54      C(63 downto 32)<=(others=>'0');
55
56  elsif (ALU_sel = "10001") then
57      C(31 downto 0)<= not B;
58      C(63 downto 32)<=(others=>'0');
59
60  elsif (ALU_sel = "10000") then
61      C(31 downto 0)<= negout;
62      C(63 downto 32)<=(others=>'0');
63
64  elsif (ALU_sel = "00101") then
65      C(31 downto 0)<= shrout;
66      C(63 downto 32)<=(others=>'0');
67
68  elsif (ALU_sel = "00110") then
```

Figure 6: first page of ALU code

```
69        C(31 downto 0)<= shlout;
70        C(63 downto 32)<=(others=>'0');
71
72    elsif (ALU_sel = "00111") then
73        C(31 downto 0)<= rorout;
74        C(63 downto 32)<=(others=>'0');
75
76    elsif (ALU_sel = "01000") then
77        C(31 downto 0)<= rolout;
78        C(63 downto 32)<=(others=>'0');
79
80    elsif (ALU_sel = "01110") then
81        C <= Mulout;
82
83    elsif (ALU_sel = "01111") then
84        C <= Divout;
85
86    elsif (ALU_sel ="00011") then
87        addorsub <= '1';
88        C(31 downto 0) <= addsubout;
89        C(63 downto 32) <= (others => '0');
90        overflow <= addsuboverflow;
91
92    elsif (ALU_sel ="00100") then
93        addorsub <= '0';
94        C(31 downto 0) <= addsubout;
95        C(63 downto 32) <= (others => '0');
96        overflow <= addsuboverflow;
97    else
98        C <= (others=>'0');
99
100   end if ;
101   end process;
102   end architecture behav;
```

Figure 7: second page of ALU code

lpm_add_sub (built-in function)

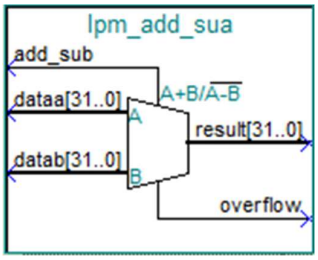

Figure 8: schematic of built-in add/subtract function

lpm_divide (built-in function)

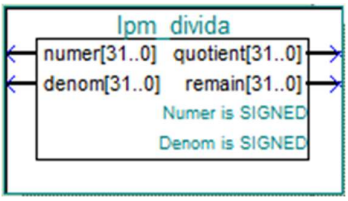

Figure 9: schematic of built-in divide function

booth_logic.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity booth_logic is port(
MUcand : in std_logic_vector(31 downto 0);
MUer : in std_logic_vector(31 downto 0);
product : out std_logic_vector(63 downto 0)
);
end entity booth_logic;

architecture logic of booth_logic is
begin
process(MUcand, MUer)
variable sum : signed(63 downto 0) := (others => '0');
variable M : signed(63 downto 0);
variable q : signed(32 downto 0);
    begin
    M := resize(signed(MUcand),  M'length);
    q(0) := '0';
    q(32 downto 1) := signed(MUer);
    for i in 0 to 31 loop
        if(q(0) = '0' and q(1) = '1') then
            sum := sum - (M sll i);
        elsif(q(0) = '1' and q(1) = '0') then
            sum := sum + (M sll i);
        end if;
        q := q srl 1;
    end loop;
    product <= std_logic_vector(sum);
    sum := (others => '0');
    end process;
end architecture logic;
```

Figure 10: booth multiplication code

negate32.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity negate32 is
    port(
        input: in std_logic_vector(31 downto 0);
        output : out std_logic_vector(31 downto 0)
    );
end entity negate32;

architecture logic of negate32 is
begin
output <= std_logic_vector(signed(not input) + 1);
end architecture logic;
```

Figure 11: two's compliment code

rol32.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY lpm;
USE lpm.all;

ENTITY rol32 IS
    PORT
    (
        data        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        distance        : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        result      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END rol32;


ARCHITECTURE SYN OF rol32 IS

    SIGNAL sub_wire0    : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL sub_wire1    : STD_LOGIC ;



    COMPONENT lpm_clshift
    GENERIC (
        lpm_shifttype       : STRING;
        lpm_type        : STRING;
        lpm_width       : NATURAL;
        lpm_widthdist       : NATURAL
    );
    PORT (
            data    : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            direction   : IN STD_LOGIC ;
            distance    : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
            result  : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
    END COMPONENT;

BEGIN
    sub_wire1    <= '0';
    result    <= sub_wire0(31 DOWNTO 0);

    LPM_CLSHIFT_component : LPM_CLSHIFT
    GENERIC MAP (
        lpm_shifttype => "ROTATE",
        lpm_type => "LPM_CLSHIFT",
        lpm_width => 32,
        lpm_widthdist => 5
    )
    PORT MAP (
        data => data,
        direction => sub_wire1,
        distance => distance,
        result => sub_wire0
    );


END SYN;
```

Figure 12: rotate left code

ror32.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY lpm;
USE lpm.all;

ENTITY ror32 IS
    PORT
    (
        data        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        distance        : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        result      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END ror32;


ARCHITECTURE SYN OF ror32 IS

    SIGNAL sub_wire0    : STD_LOGIC_VECTOR (31 DOWNTO 0);
    SIGNAL sub_wire1    : STD_LOGIC ;



    COMPONENT lpm_clshift
    GENERIC (
        lpm_shifttype       : STRING;
        lpm_type        : STRING;
        lpm_width       : NATURAL;
        lpm_widthdist       : NATURAL
    );
    PORT (
            data    : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            direction   : IN STD_LOGIC ;
            distance    : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
            result  : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
    END COMPONENT;

BEGIN
    sub_wire1    <= '1';
    result    <= sub_wire0(31 DOWNTO 0);

    LPM_CLSHIFT_component : LPM_CLSHIFT
    GENERIC MAP (
        lpm_shifttype => "ROTATE",
        lpm_type => "LPM_CLSHIFT",
        lpm_width => 32,
        lpm_widthdist => 5
    )
    PORT MAP (
        data => data,
        direction => sub_wire1,
        distance => distance,
        result => sub_wire0
    );


END SYN;
```

Figure 13: rotate right code

shl32.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity shl32 is port(
    input0 : in std_logic_vector(31 downto 0);
    input1 : in std_logic_vector(31 downto 0);
    output : out std_logic_vector(31 downto 0)
);
end entity shl32;

architecture logic of shl32 is
begin
output <= std_logic_vector(shift_left(unsigned(input0), to_integer(signed(input1))));
end architecture logic;
```

Figure 14: shift left code

shr32.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity shr32 is port(
    input0 : in std_logic_vector(31 downto 0);
    input1 : in std_logic_vector(31 downto 0);
    output : out std_logic_vector(31 downto 0)
);
end entity shr32;

architecture logic of shr32 is
begin
output <= std_logic_vector(shift_right(unsigned(input0), to_integer(signed(input1))));
end architecture logic;
```

Figure 15: shift right code

the_bus.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.components_all.all;

entity the_bus is port (
        muxIn0      : in std_logic_vector (31 downto 0);
        muxIn1      : in std_logic_vector (31 downto 0);
        muxIn2      : in std_logic_vector (31 downto 0);
        muxIn3      : in std_logic_vector (31 downto 0);
        muxIn4      : in std_logic_vector (31 downto 0);
        muxIn5      : in std_logic_vector (31 downto 0);
        muxIn6      : in std_logic_vector (31 downto 0);
        muxIn7      : in std_logic_vector (31 downto 0);
        muxIn8      : in std_logic_vector (31 downto 0);
        muxIn9      : in std_logic_vector (31 downto 0);
        muxIn10     : in std_logic_vector (31 downto 0);
        muxIn11     : in std_logic_vector (31 downto 0);
        muxIn12     : in std_logic_vector (31 downto 0);
        muxIn13     : in std_logic_vector (31 downto 0);
        muxIn14     : in std_logic_vector (31 downto 0);
        muxIn15     : in std_logic_vector (31 downto 0);
        muxIn16     : in std_logic_vector (31 downto 0);
        muxIn17     : in std_logic_vector (31 downto 0);
        muxIn18     : in std_logic_vector (31 downto 0);
        muxIn19     : in std_logic_vector (31 downto 0);
        muxIn20     : in std_logic_vector (31 downto 0);
        muxIn21     : in std_logic_vector (31 downto 0);
        muxIn22     : in std_logic_vector (31 downto 0);
        muxIn23     : in std_logic_vector (31 downto 0);
        muxIn24     : in std_logic_vector (31 downto 0);
        muxIn25     : in std_logic_vector (31 downto 0);
        muxIn26     : in std_logic_vector (31 downto 0);
        muxIn27     : in std_logic_vector (31 downto 0);
        muxIn28     : in std_logic_vector (31 downto 0);
        muxIn29     : in std_logic_vector (31 downto 0);
        muxIn30     : in std_logic_vector (31 downto 0);
        muxIn31     : in std_logic_vector (31 downto 0);
        encoder32In : in std_logic_vector(31 downto 0);
        BusMuxOut   : out std_logic_vector(31 downto 0)
);
end entity the_bus;

architecture structure of the_bus is
signal Mux_sel: std_logic_vector(4 downto 0);

begin
Bus_encoder: encoder32to5 port map(encoder32In, Mux_sel);
Bus_Mux: lpm_mua port map (muxIn0, muxIn1, muxIn2, muxIn3, muxIn4, muxIn5,
muxIn6, muxIn7, muxIn8, muxIn9, muxIn10, muxIn11, muxIn12, muxIn13,
muxIn14, muxIn15, muxIn16, muxIn17, muxIn18, muxIn19, muxIn20, muxIn21,
muxIn22, muxIn23, muxIn24, muxIn25, muxIn26, muxIn27, muxIn28, muxIn29,
muxIn30, muxIn31, Mux_sel, BusMuxOut);

end architecture structure;
```

Figure 16: bus code

encoder32to5.vhd

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_unsigned.all;
4
5   entity encoder32to5 is port(
6           input : in std_logic_vector(31 downto 0);
7           output: out std_logic_vector(4 downto 0)
8   );
9   end entity encoder32to5;
10
11  architecture behav of encoder32to5 is
12  begin
13  output <= "00000" when (input = "00000000000000000000000000000001")
14                else "00001" when (input = "00000000000000000000000000000010")
15                else "00010" when (input = "00000000000000000000000000000100")
16                else "00011" when (input = "00000000000000000000000000001000")
17                else "00100" when (input = "00000000000000000000000000010000")
18                else "00101" when (input = "00000000000000000000000000100000")
19                else "00110" when (input = "00000000000000000000000001000000")
20                else "00111" when (input = "00000000000000000000000010000000")
21                else "01000" when (input = "00000000000000000000000100000000")
22                else "01001" when (input = "00000000000000000000001000000000")
23                else "01010" when (input = "00000000000000000000010000000000")
24                else "01011" when (input = "00000000000000000000100000000000")
25                else "01100" when (input = "00000000000000000001000000000000")
26                else "01101" when (input = "00000000000000000010000000000000")
27                else "01110" when (input = "00000000000000000100000000000000")
28                else "01111" when (input = "00000000000000001000000000000000")
29                else "10000" when (input = "00000000000000010000000000000000")
30                else "10001" when (input = "00000000000000100000000000000000")
31                else "10010" when (input = "00000000000001000000000000000000")
32                else "10011" when (input = "00000000000010000000000000000000")
33                else "10100" when (input = "00000000000100000000000000000000")
34                else "10101" when (input = "00000000001000000000000000000000")
35                else "10110" when (input = "00000000010000000000000000000000")
36                else "10111" when (input = "00000000100000000000000000000000")
37                else "11000" when (input = "00000001000000000000000000000000")
38                else "11001" when (input = "00000010000000000000000000000000")
39                else "11010" when (input = "00000100000000000000000000000000")
40                else "11011" when (input = "00001000000000000000000000000000")
41                else "11100" when (input = "00010000000000000000000000000000")
42                else "11101" when (input = "00100000000000000000000000000000")
43                else "11110" when (input = "01000000000000000000000000000000")
44                else "11111";
45  end architecture behav;
```

Figure 17: 32-to-5 encoder

lpm_mua (built-in function)

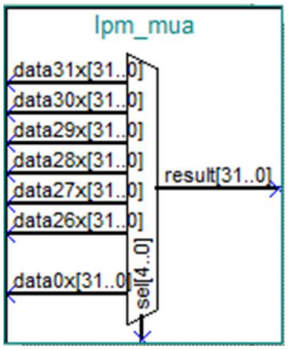

Figure 18: schematic of built-in MUA function

MDR.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use work.components_all.all;
5
6  entity MDR is port(
7  MDMux_read: in std_logic;
8  MDR_enable: in std_logic;
9  clr: in std_logic;
10 clk: in std_logic;
11 MDR_in1:in std_logic_vector(31 downto 0);
12 MDR_in2: in std_logic_vector(31 downto 0); --1 is from bus, 2 is from memory
13 MDR_out: out std_logic_vector(31 downto 0)
14 );
15 end entity MDR;
16
17 architecture behav of MDR is
18 signal MDMux_out: std_logic_vector(31 downto 0);
19 begin
20 MDMux_out <= MDR_in1 when MDMux_read = '0' else
21                MDR_in2;
22 The_MDR: register32bit port map(MDMux_out, MDR_enable, clr, clk, MDR_out);
23
24 end architecture behav;
```

Figure 19: MDR code

Section 2: Testbench

Testbench file for “and” instruction:

```
library ieee;
use ieee.std_logic_1164.all;

entity datapath_and_tb is
end;

architecture logic of datapath_and_tb is

signal clk_tb : std_logic;
signal clr_tb : std_logic;
signal IncPC_tb : std_logic;
signal MDRRead_tb   : std_logic;
signal aluOp        : std_logic_vector(4 downto 0);
signal encoderIn_tb : std_logic_vector(31 downto 0);
signal RegEnable_tb : std_logic_vector(31 downto 0);
signal Mdatain_tb   :   std_logic_vector(31 downto 0);

signal BusMuxOut_tb,IRout_tb:   std_logic_vector(31 downto 0);
signal Zout_tb:std_logic_vector(63 downto 0);
signal R2out_tb: std_logic_vector(31 downto 0);
signal R4out_tb: std_logic_vector(31 downto 0);
signal R5out_tb: std_logic_vector(31 downto 0);
signal HIout_tb, LOout_tb: std_logic_vector(31 downto 0);
type    state is(default, Reg_load1a, Reg_load1b, Reg_load2a, Reg_load2b, Reg_load3a,
Reg_load3b, T0, T1, T2, T3, T4, T5, T6);
signal  present_state: State := default;

component datapath
    port (
        clk: in std_logic;
        encoderin: in std_logic_vector(31 downto 0);
        register_enable : in std_logic_vector(31 downto 0);
        Mdatain : in std_logic_vector(31 downto 0);
        MDR_Read    : in std_logic;
        ALU_sel : in std_logic_vector(4 downto 0);
        PC_plus : in std_logic;

        BusMuxOut: out std_logic_vector(31 downto 0);
        R2out   : out std_logic_vector(31 downto 0);
        R4out   : out std_logic_vector(31 downto 0);
        R5out   : out std_logic_vector(31 downto 0);
        HIout   : out std_logic_vector(31 downto 0);
        LOout   : out std_logic_vector(31 downto 0);
        IRout   : out std_logic_vector(31 downto 0);
        Zout    : out std_logic_vector(63 downto 0)
    );
end component;

begin
Test : datapath port map (clk_tb, encoderIn_tb, RegEnable_tb, Mdatain_tb, MDRRead_tb,
aluOp, IncPC_tb, BusMuxOut_tb,
    R2out_tb, R4out_tb, R5out_tb, HIout_tb, LOout_tb, IRout_tb, Zout_tb);

clk_process: process
begin
    clk_tb <= '1', '0' after 10 ns;
    wait for 20 ns;
end process clk_process;

process(clk_tb)
begin
    if(clk_tb'event and clk_tb = '1') then
        case present_state is
            when default =>
                present_state <= Reg_load1a;
            when Reg_load1a =>
                present_state <= Reg_load1b;
            when Reg_load1b =>
                present_state <= Reg_load2a;
```

Figure 20: first page of testbench code for “and” instruction

```
            when Reg_load2a =>
                present_state <= Reg_load2b;
            when Reg_load2b =>
                present_state <= Reg_load3a;
            when Reg_load3a =>
                present_state <= Reg_load3b;
            when Reg_load3b =>
                present_state <= T0;
            when T0 =>
                present_state <= T1;
            when T1 =>
                present_state <= T2;
            when T2 =>
                present_state <= T3;
            when T3 =>
                present_state <= T4;
            when T4 =>
                present_state <= T5;
            when T5 =>
                present_state <= T6;
            when others =>
        end case;
    end if;
end process;

process (present_state)
begin
    case present_state is
        when default =>
            IncPC_tb <= '0';
            MDRRead_tb <= '0';
            aluOp <= (others => '0');
            Mdatain_tb <= (others => '0');
            encoderIn_tb <= (others => '0');
            RegEnable_tb <= (others => '0');
        when Reg_load1a =>
            Mdatain_tb <= x"00000022";
            MDRRead_tb <= '0', '1' after 10 ns, '0' after 25 ns;
            RegEnable_tb <= (others=>'0'), (20=>'1', others=>'0') after 10 ns,
            (others=>'0') after 25 ns;   -- MDRin enable (the 20th bit of regenable);

        when Reg_load1b =>
            encoderIn_tb <= (others=>'0'), (21 => '1', others => '0') after 10 ns,
            (others=>'0') after 25 ns;
            RegEnable_tb <= (others=>'0'), (2 => '1', others => '0') after 10 ns,
            (others=>'0') after 25 ns;

        when Reg_load2a =>
            Mdatain_tb <= x"00000024";
            MDRRead_tb <= '0', '1' after 10 ns, '0' after 25 ns;

            RegEnable_tb <= (others=>'0'), (20=>'1', others=>'0') after 10 ns,
            (others=>'0') after 25 ns;   -- MDRin
        when Reg_load2b =>
            encoderIn_tb <= (others=>'0'), (21 => '1', others => '0')after 10 ns,
            (others=>'0') after 25 ns;
            RegEnable_tb <= (others=>'0'), (4 => '1', others => '0')after 10 ns,
            (others=>'0') after 25 ns; -- R4 load_en

        when Reg_load3a =>
            Mdatain_tb <= x"00000026";
            MDRRead_tb <= '0', '1' after 10 ns, '0' after 25 ns;
            RegEnable_tb <= (others=>'0'), (20=>'1', others=>'0') after 10 ns,
            (others=>'0') after 25 ns;

        when Reg_load3b =>
            encoderIn_tb <= (others=>'0'), (21 => '1', others => '0')after 10 ns,
            (others=>'0') after 25 ns;
            RegEnable_tb <= (others=>'0'), (5 => '1', others => '0')after 10 ns,
```

Figure 21: second page of testbench code for “and” instruction

```
            (others=>'0') after 25 ns; -- R5 load_en
129
130         when T0 =>
131             encoderIn_tb <= (20 => '1', others => '0'); -- pc encoder
132             RegEnable_tb <= (21 => '1', 23 => '1', others => '0');  -- MAR, Zin
133             IncPC_tb <= '1';
134
135         when T1 =>
136             encoderIn_tb <= (19 => '1', others => '0'); -- zlow encoder
137             RegEnable_tb <= (18 => '1', 20 => '1', others => '0'); -- pc load_en, MDR
                load_en
138             IncPC_tb <= '0';
139             MDRRead_tb <= '1';
140             Mdatain_tb <= x"4A920000";  -- opcode for "and R5, R2, R4"
141
142         when T2 =>
143             MDRRead_tb <= '0';
144             Mdatain_tb <= (others => '0');
145             encoderIn_tb <= (21 => '1', others => '0'); -- MDR encoder input
146             RegEnable_tb <= (19 => '1', others => '0'); -- IR load_en
147
148         when T3 =>
149             encoderIn_tb <= (2 => '1', others => '0');  -- R2 encoder input
150             RegEnable_tb <= (22 => '1', others => '0'); -- RY load_en
151
152         when T4 =>
153             encoderIn_tb <= (4 => '1', others => '0');  -- R4 encoder input
154             aluOp <= "01001";
155             RegEnable_tb <= (23 => '1', others => '0'); -- RZ load_en
156
157         when T5 =>
158             encoderIn_tb <= (19 => '1', others => '0'); -- Zlow encoder
159             RegEnable_tb <= (5 => '1', 17=>'1', others => '0'); -- R5, LO load_en
160
161         when T6 =>
162             encoderIn_tb <= (18 => '1', others => '0'); -- Zhigh encoder
163             RegEnable_tb <= (16 => '1', others=>'0');   -- HI load_en
164         when others =>
165     end case;
166 end process;
167 end architecture;
```

Figure 22: third page of testbench code for "and" instruction

The above printout of the VHDL code for the "and" instruction testbench is similar to most other testbenches other than the "negative" and "not" instruction testbenches. The ones that are similar to the "and" instruction testbench differ in only three places. The value of the instruction opcode and the value of the ALU opcode. The value in registers is occasionally different in order to properly show the correct functionality of the instruction. The "negative" and "not" instruction test benches are similar to each other and, on top of the differences mentioned previously, differ from the other testbenches by having one less state due to needing less registers.

Section 3: Functional Simulation and Demonstration

The screenshots of the testbench files for all the functional simulation result shown below have been provided in Section 2: Testbench above.

3.a and R5, R2, R4

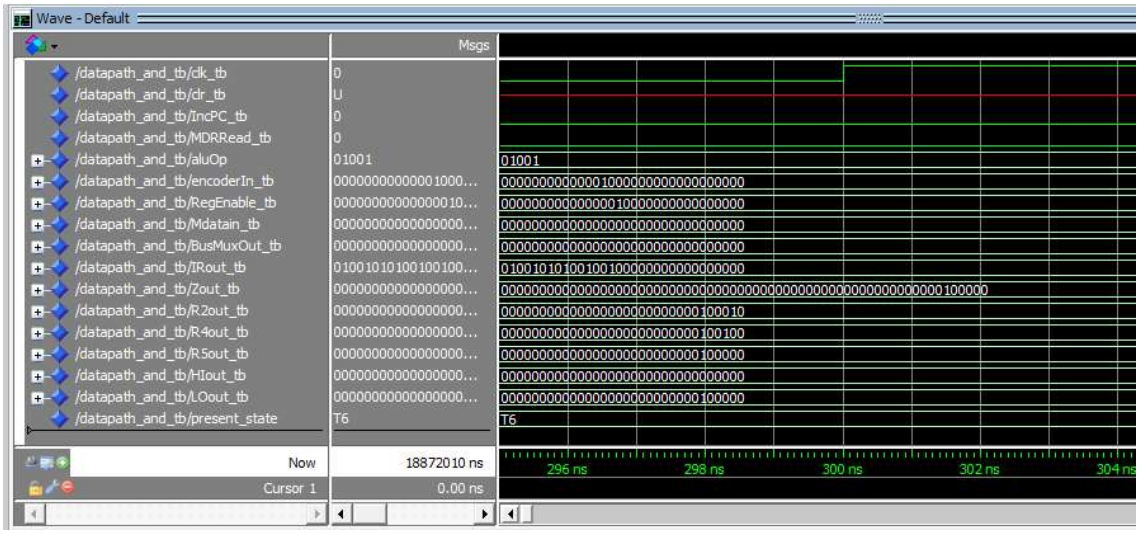

Figure 23:Functional simulation for AND operation

(1) R2 is initialized with value 34 in decimal (or hex “0000_0022”/ “100010” in binary)
(2) R4 is initialized with value 36 in decimal (or hex “0000_0024”/ “100100” in binary)
(3) R5 is initialized with value 38 in decimal (or hex “0000_0026”/ “100110” in binary)

T0: PCout signal is for encoder to select PC as input to the bus, incPC turns to ‘1’ (to increment the PC), MARin and Zin enables the registers MAR and Z, respectively.

T1: Zlowout signal for encoder selects the lower half of register as input to the bus, PCin enables the PC, MDRin enables the MDR to accept data, read signal select Mdata as input to the MDR, the Mdatain is loaded with the instruction x"4A920000" for the AND operation.

T2: MDRout signal causes the bus to select the output of MDR as input, IRin enables the IR.

T3: R2out is the input to the bus encoder for selection of R2 as the input to the bus, Yin enables the Y register.

T4: R4out is the input to the bus encoder for selection of R4 as the input to the bus, AND is the opcode signal for the ALU to performance AND operation, Zin enables register Z to accept data.

T5: Zlowout signal ensure the bus takes the lower 32-bit of the Z register (the result of the AND operation) as the input, R5in enables R5 to accept data coming from the bus.

(4) As we can see in Figure 23 above, in the end of state T6 (State T6 is an extra step for the upper 32-bit of register Z), the result of the AND operation is equal to 100000 which makes logical sense (100010 and 100100 = 100000). The IRout also matches the instruction opcode for this operation, which is **x"4A920000"**. The ALU signal is 01001 which is the opcode for the AND operation.

3.b or R5, R2, R4

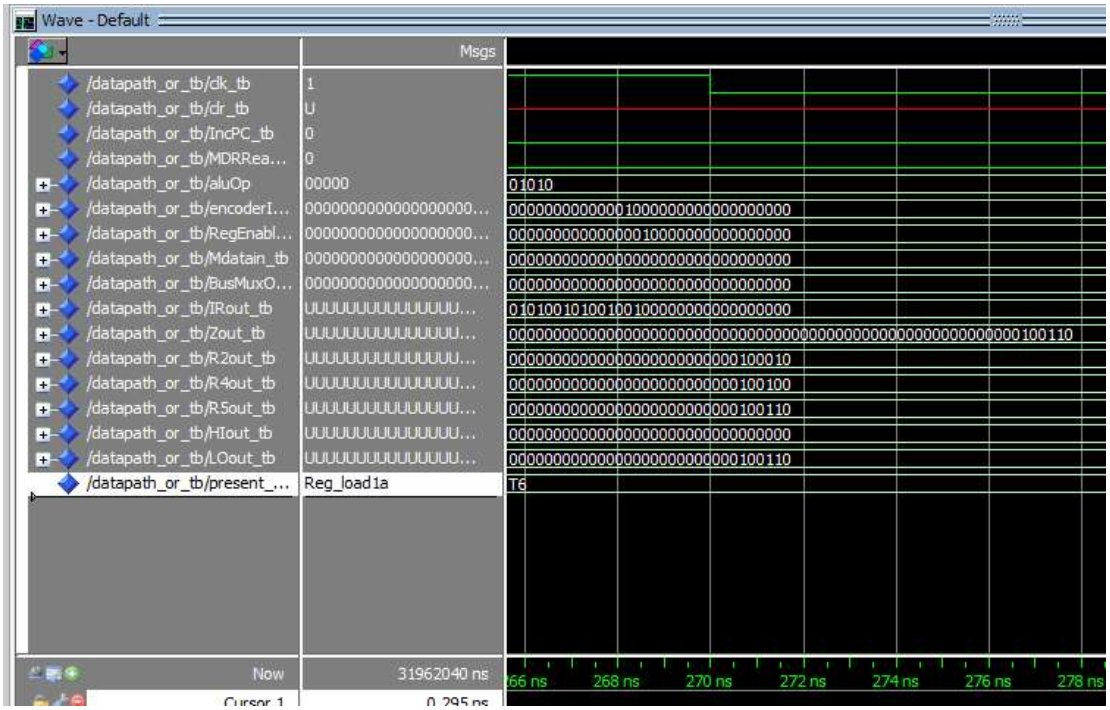

Figure 24:Functional simulation for OR operation

(1) R2 is initialized with value 34 in decimal (or hex “0000_0022”/ “100010” in binary)
(2) R4 is initialized with value 36 in decimal (or hex “0000_0024”/ “100100” in binary)
(3) R5 is initialized with value 38 in decimal (or hex “0000_0026”/ “100110” in binary)

The control sequences are mostly the same as the AND operation in 3.a and R5, R2, R4, except the OR operation has a different instruction and ALU opcode which is equal to **x"52920000"** and **01010**, respectively.

(4) As we can see in Figure 24 above, in the end of state T6 (State T6 is an extra step for the upper 32-bit of register Z), the result of the OR operation is equal to 100110 which makes logical sense (100010 or 100100 = 100110).

3.c add R5, R2, R4

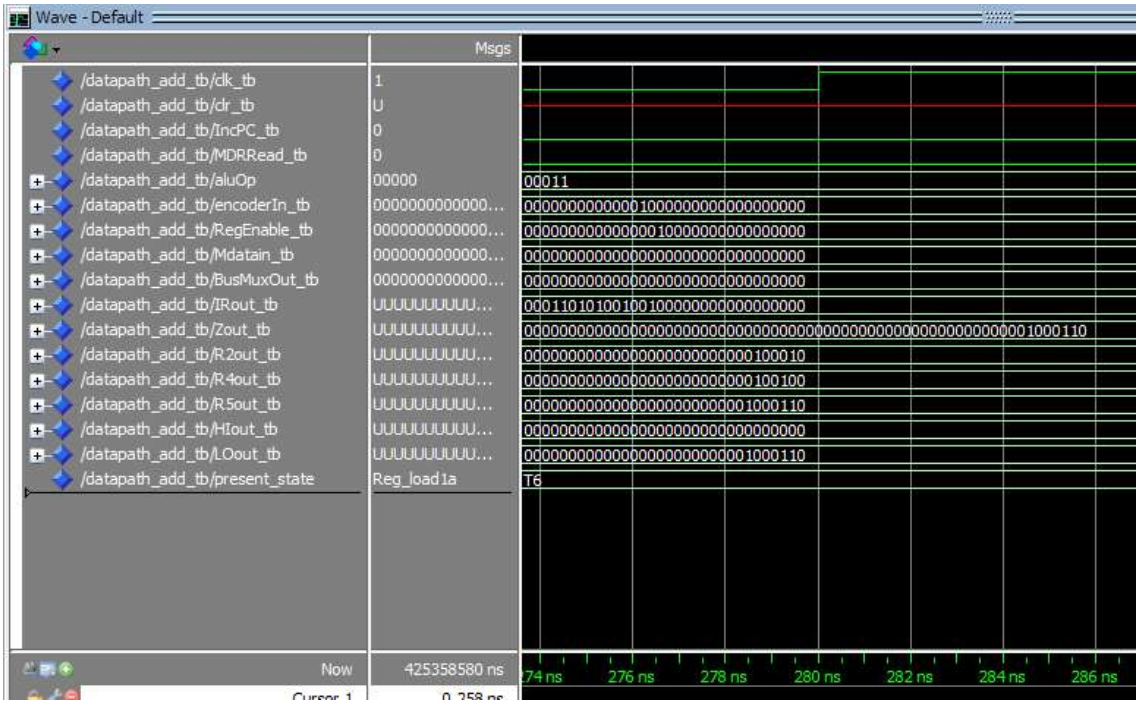

Figure 25:Functional Simulation for ADD operation

(1) R2 is initialized with value 34 in decimal (or hex “0000_0022”/ “100010” in binary)
(2) R4 is initialized with value 36 in decimal (or hex “0000_0024”/ “100100” in binary)
(3) R5 is initialized with value 38 in decimal (or hex “0000_0026”/ “100110” in binary)

The control sequences are mostly the same as the AND operation in 3.a and R5, R2, R4, except the ADD operation has a different instruction and ALU opcode which is equal to **x"1A920000"** and **00011**, respectively.

(4) As we can see in Figure 25 above, in the end of state T6 (State T6 is an extra step for the upper 32-bit of register Z), the result of the ADD operation is equal to 1000110 (34+36 = 70 (in decimal)) which makes logical sense.

3.d sub R5, R2, R4

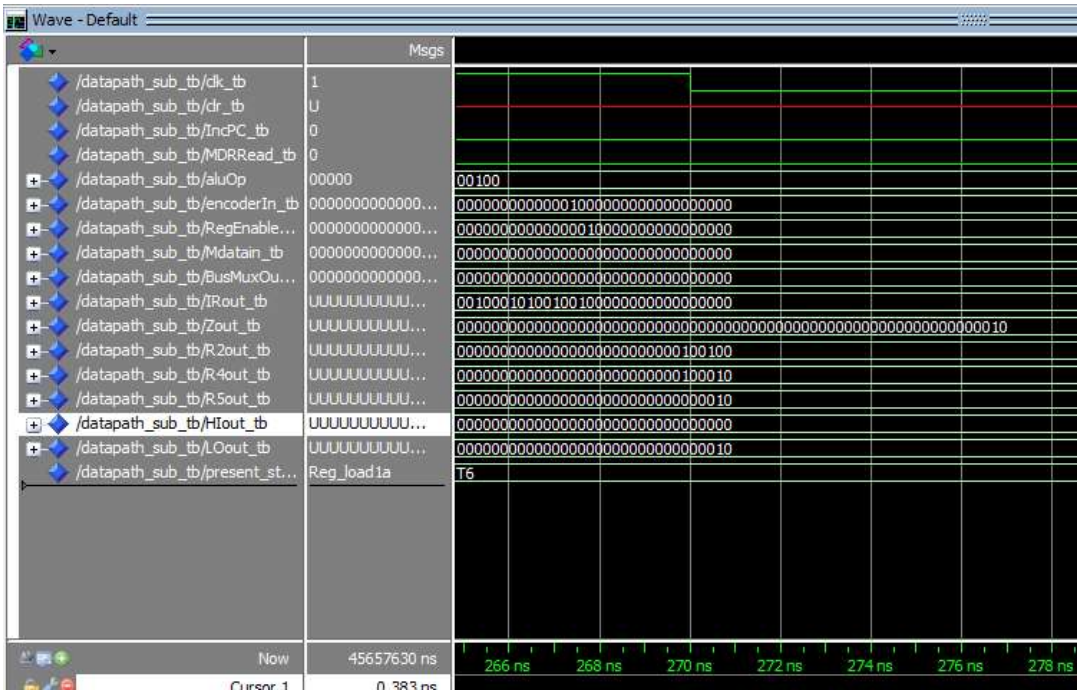

Figure 26:Functional Simulation of sub operation

(1) R4 is initialized with value 34 in decimal (or hex “0000_0022”/ “100010” in binary)
(2) R2 is initialized with value 36 in decimal (or hex “0000_0024”/ “100100” in binary)
(3) R5 is initialized with value 38 in decimal (or hex “0000_0026”/ “100110” in binary)

The control sequences are mostly the same as the ADD operation in 3.c add R5, R2, R4, except the sub operation has a different instruction and ALU opcode which is equal to **x"22920000"**and **00100**, respectively.

(4) As we can see in Figure 26 above, in the end of state T6 (State T6 is an extra step for the upper 32-bit of register Z), the result (which is inside both R5 and the LO register) of the sub operation is equal to 00010 (36 – 34 = 2 (in decimal)) which makes logical sense.

3.e mul R2, R4

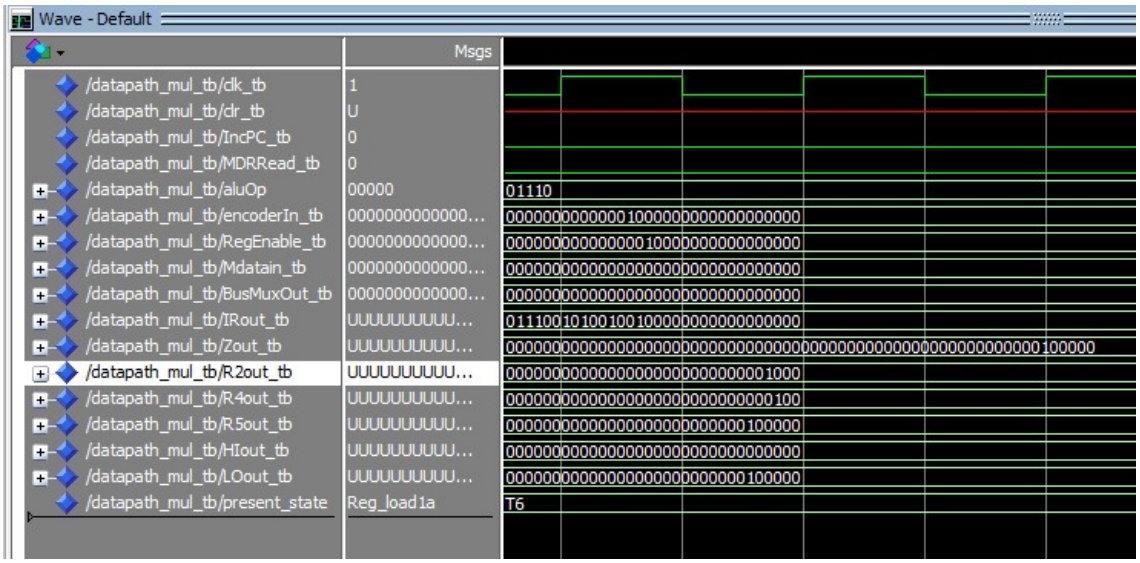

Figure 27:Functional Simulation of mul operation

(1) R2 is initialized with value 8 in decimal (or hex “0000_0008”/ “1000” in binary)
(2) R4 is initialized with value 4 in decimal (or hex “0000_0004”/ “0010” in binary)

The control sequences are mostly the same as the ADD operation in 3.c add R5, R2, R4, except the MUL operation has a different instruction and ALU opcode which is equal to **x"72920000"**and **01110**, respectively. In addition, the higher 32-bit of register Z is outputted to the HI register at state T6.

(4) As we can see in Figure 27 above, in the end of state T6, the result of the MUL operation is equal to 100000 (8 * 4 = 32 (in decimal)) which makes logical sense. The result of the operation is stored in the LO register as expected. The R5out in the above diagram is included only for convenience (not making change to the testbench code), the actual operation does not involve R5.

3.f div R2, R4

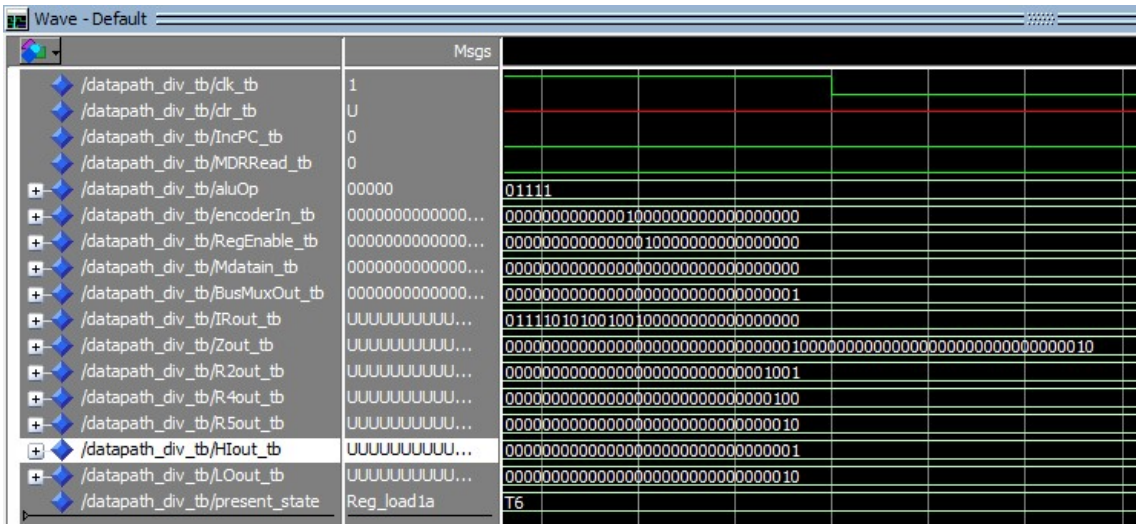

Figure 28:Functional Simulation for div operation

(1) R2 is initialized with value 9 in decimal (or hex “0000_0009”/ “1001” in binary)
(2) R4 is initialized with value 4 in decimal (or hex “0000_0004”/ “0010” in binary)

The control sequences are mostly the same as the MUL operation in 3.e mul R2, R4, except the div operation has a different instruction and ALU opcode which is equal to **x"7A920000"** and **01111**, respectively. In addition, the higher 32-bit of register Z is outputted to the HI register at state T6.

(4) As we can see in Figure 28 above, in the end of state T6, the result of the div operation is equal to 2R1 (9/4=2R1(in decimal)) which makes logical sense. The quotient (10 = 2 (in decimal)) of the operation is stored in the LO register, and the remainder (01 = 1(in decimal)) is stored in the HI register as expected. The R5out in the above diagram is included only for convenience (not making change to the testbench code), the actual operation does not involve R5.

3.g shr R5, R2, R4

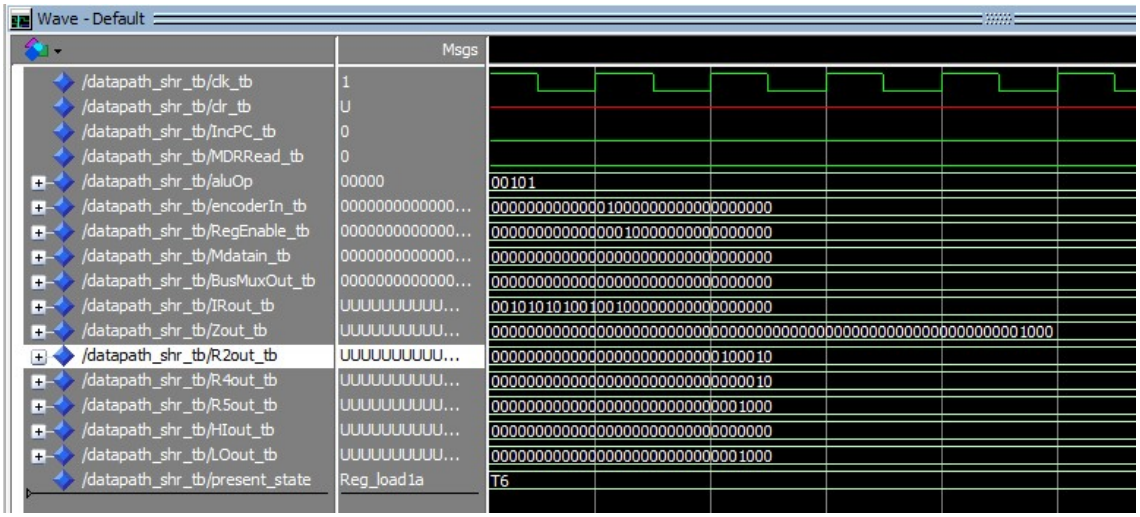

Figure 29:Functional Simulation of shr operation

(1) R2 is initialized with value 34 in decimal (or hex “0000_0022”/ “100010” in binary)
(2) R4 is initialized with value 2 in decimal (or hex “0000_0002”/ “000010” in binary)
(3) R5 is initialized with value 38 in decimal (or hex “0000_0026”/ “100110” in binary)

The control sequences are mostly the same as the SUB operation in 3.d sub R5, R2, R4, except the SHR operation has a different instruction and ALU opcode which is equal to **x"2A920000"**and **00101**, respectively.

(4) As we can see in Figure 29 above, in the end of state T6, the result of the SHR operation is equal to 1000 which makes logical sense. The content in R2 (100010) is shifted to the right by 2 bits (content in R4), which results in 10000 and it is stored in R5. The initial value stored in R5 is overwritten.

3.h shl R5, R2, R4

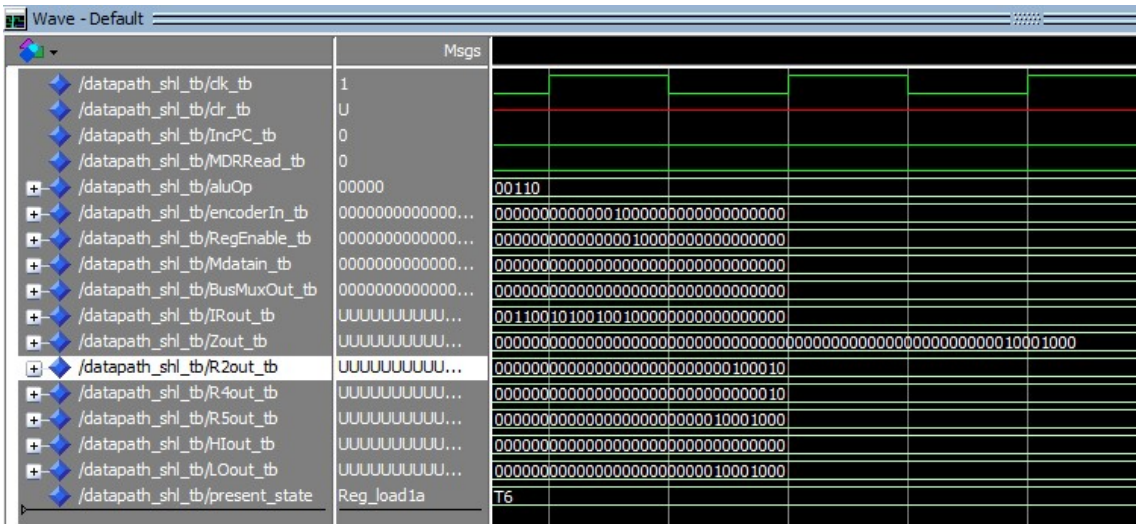

Figure 30:Functional Simulation of shl operation

(1) R2 is initialized with value 34 in decimal (or hex “0000_0022”/ “100010” in binary)
(2) R4 is initialized with value 2 in decimal (or hex “0000_0002”/ “000010” in binary)
(3) R5 is initialized with value 38 in decimal (or hex “0000_0026”/ “100110” in binary)

The control sequences are mostly the same as the SHR operation in 3.g shr R5, R2, R4, except the SHL operation has a different instruction and ALU opcode which is equal to **x"32920000"** and **00110**, respectively.

(4) As we can see in Figure 30 above, in the end of state T6, the result of the SHL operation is equal to 10001000 which makes logical sense. The content in R2 (100010) is shifted to the left by 2 bits (content in R4), which results in 10001000 and it is stored in R5. The initial value stored in R5 is overwritten.

3.i ror R5, R2, R4

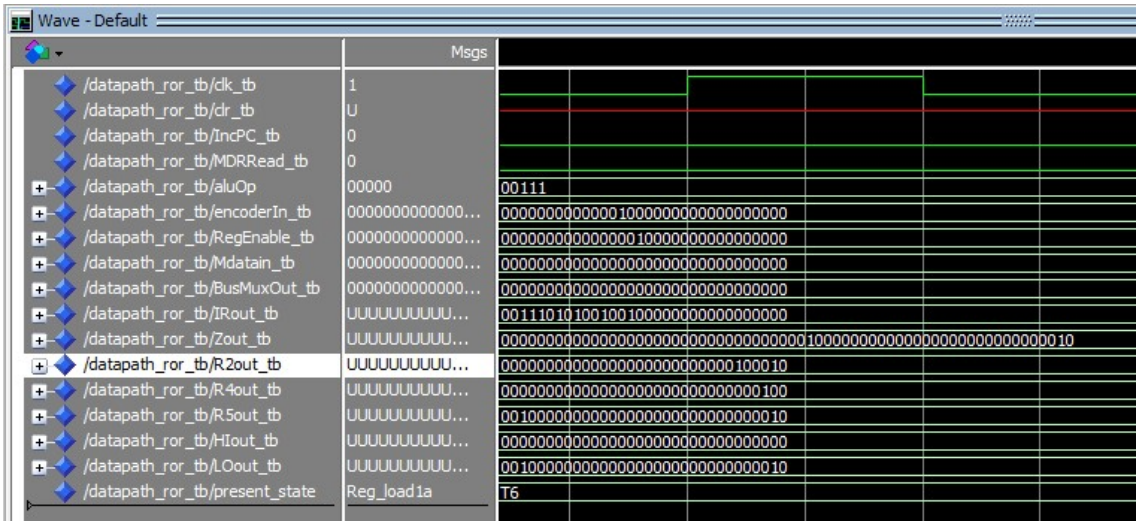

Figure 31:Functional Simulation of ror operation

(1) R2 is initialized with value 34 in decimal (or hex “0000_0022”/ “100010” in binary)
(2) R4 is initialized with value 4 in decimal (or hex “0000_0004”/ “000010” in binary)
(3) R5 is initialized with value 38 in decimal (or hex “0000_0026”/ “100110” in binary)

The control sequences are mostly the same as the SHR operation in 3.g shr R5, R2, R4, except the ROR operation has a different instruction and ALU opcode which is equal to **x"3A920000"** and **00111**, respectively.

(4) As we can see in Figure 31 above, in the end of state T6, the result of the ROR operation is equal to **0010**00000000000000000000000000**10** which makes logical sense. The content in R2 (100010) is rotated to the right by 4 bits (content in R4), which means the four most significant bits (0010) will become the four least significant bit of the 32-bit number since it got rotated to the right.

3.j rol R5, R2, R4

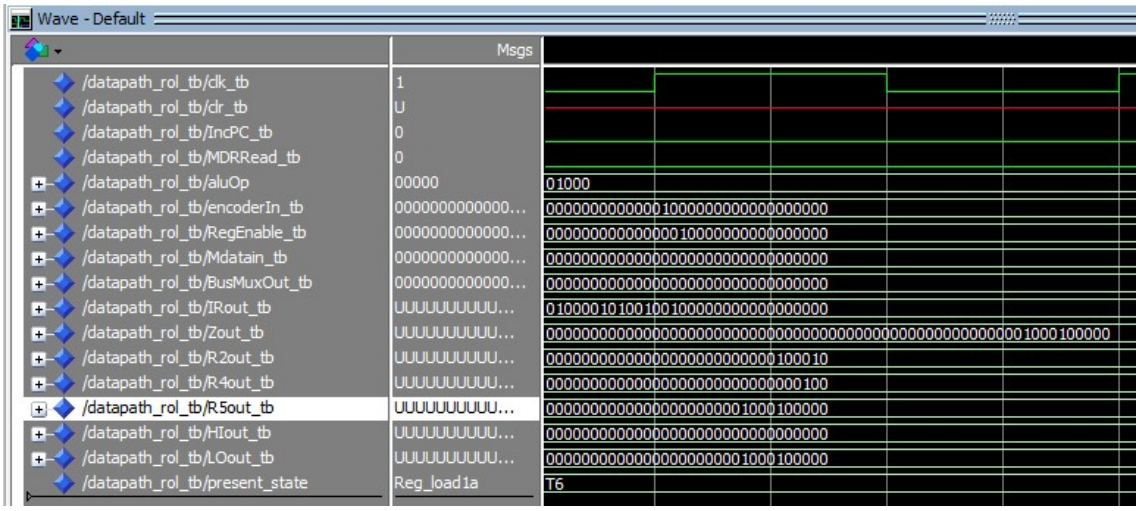

Figure 32:Functional Simulation of rol operation

(1) R2 is initialized with value 34 in decimal (or hex “0000_0022”/ “100010” in binary)
(2) R4 is initialized with value 4 in decimal (or hex “0000_0004”/ “000010” in binary)
(3) R5 is initialized with value 38 in decimal (or hex “0000_0026”/ “100110” in binary)

The control sequences are mostly the same as the ROR operation in 3.i ror R5, R2, R4, except the ROL operation has a different instruction and ALU opcode which is equal to **x"42920000"**and **01000**, respectively.

(4) As we can see in Figure 32 above, in the end of state T6, the result of the ROL operation is equal to **100010**0000 which makes logical sense. The content in R2 (100010) is rotated to the left by 4 bits (content in R4), which means the four least significant bits (0000) will become the four most significant bit of the 32-bit number since it got rotated to the left.

3.k neg R5, R2

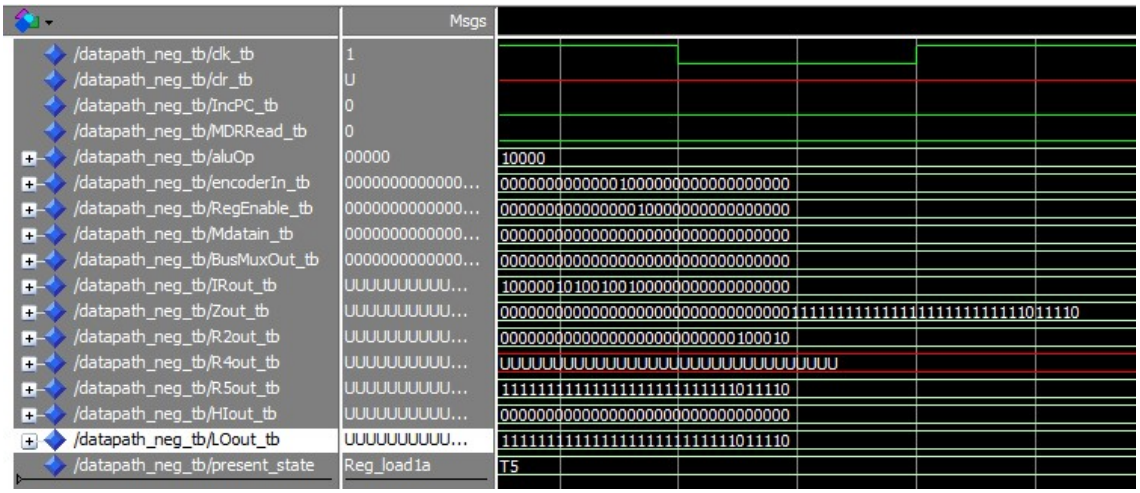

Figure 33:Functional Simulation of neg operation

(1) R2 is initialized with value 34 in decimal (or hex “0000_0022”/ “100010” in binary)
(3) R5 is initialized with value 38 in decimal (or hex “0000_0026”/ “100110” in binary)

The control sequences are mostly the same as the ROR operation in 3.i ror R5, R2, R4, except the NEG operation has a different instruction and ALU opcode which is equal to **x"82920000"**and **10000**, respectively. In addition, neg operation only has 5 states (T0 to T4) since the instruction does not involve register R4, the T5 shown in the diagram above is only for demonstration of the HI register, which is filled with zero.

(4) As we can see in Figure 33 above, in the end of state T5, the result of the NEG operation is equal to 11111111111111111111111111**011110** which makes logical sense. The content in R2 (0…000000100010) is converted to its two’s complement, which basically means flipping all the zeros with ones, and ones with zero and then add 1 to the result. The result will be stored in R5, and the process does not involve R4. It can be clearly seen that the right most 6 digits (100010) become 011101 + 1 = 011110, which confirms the result.

3.l not R5, R2

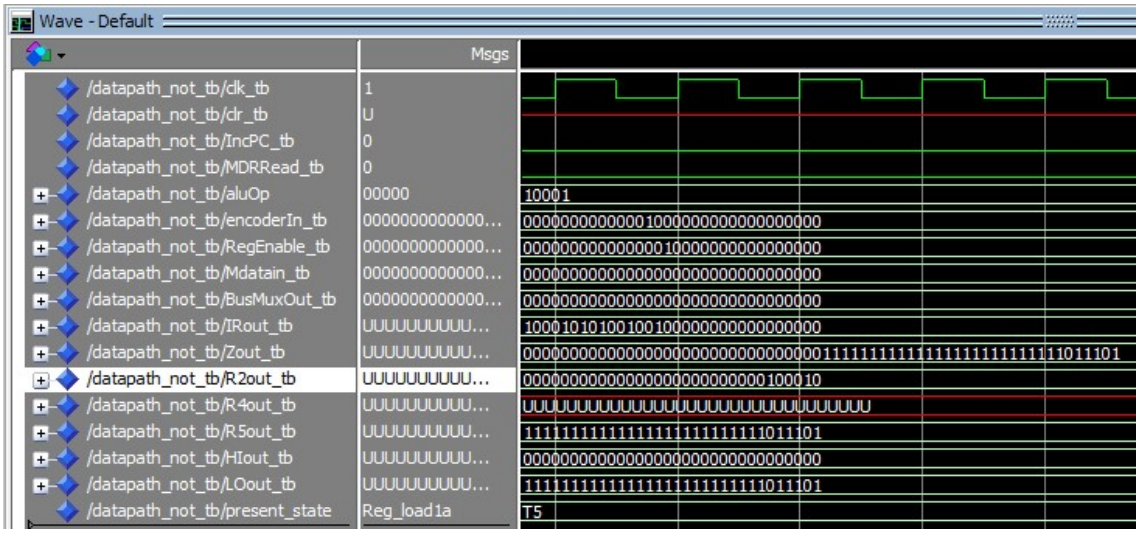

Figure 34:Functional Simulation of not operation

(1) R2 is initialized with value 34 in decimal (or hex “0000_0022”/ “100010” in binary)
(3) R5 is initialized with value 38 in decimal (or hex “0000_0026”/ “100110” in binary)

The control sequences are mostly the same as the NEG operation in 3.k neg R5, R2, except the NOT operation has a different instruction and ALU opcode which is equal to **x"8A920000"** and **10001**, respectively. In addition, the NEG operation only has 5 states (T0 to T4) since the instruction does not involve register R4, the T5 shown in the diagram above is only for demonstration of the HI register, which is filled with zero.

(4) As we can see in Figure 34 above, in the end of state T5, the result of the NOT operation is equal to 11111111111111111111111111**011101** which makes logical sense. All the zeros and ones will be replaced by their complement in R2 (0…000000100010), which simply means flipping all the zeros with

ones, and ones with zeros. The result will be stored in R5, and the process does not involve R4. It can be clearly seen that the right most 6 digits (100010) become 011101, which confirms the result.