Christopher Lall

CSc 342/343 Comp Org – Professor Gertner

ALU Lab Experiment

Due 4/10/22

Table of Contents

Objective:

The goal of this lab is to use an Instruction Register (IR), a 3-Port RAM (which accesses registers Rs, Rt, and Rd as needed), an Immediate Register for I-Format Instructions, a Data Memory (1-Port RAM), an ADD/SUB unit with flags overflow, negative, and zero, and finally a Bitwise Operation Unit integrated within the former unit to implement 16 MIPS Instructions in VHDL. Students must evaluate the validity of their design using ModelSim waveform simulation and compare the VHDL executed version of the MIPS instruction to its MARS equivalent after creating these instructions.

Code:

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity Lall_RD is
5      generic (Lall_datawidth: integer := 32);
6      port (
7         Lall_clock, Lall_write, Lall_read, Lall_enable: in std_logic;
8         Lall_content: in std_logic_vector(Lall_datawidth-1 downto 0);
9         Lall_result: out std_logic_vector(Lall_datawidth-1 downto 0));
10  end Lall_RD;
11
12  architecture arch of Lall_RD is
13
14     signal Lall_memory: std_logic_vector(Lall_datawidth-1 downto 0);
15
16     begin
17        P1: process(Lall_clock, Lall_write)
18        begin
19           if(rising_edge(Lall_clock) and Lall_write = '1')
20              then Lall_memory <= Lall_content;
21           end if;
22        end process P1;
23
24        P2: process(Lall_read, Lall_enable, Lall_memory)
25        begin
26           if(Lall_read = '1' and Lall_enable = '1')
27              then Lall_result <= Lall_memory;
28           elsif(Lall_enable = '0')
29              then Lall_result <= (others => '0');
30           end if;
31        end process P2;
32  end arch;
```

Figure 1. RD register VHDL code

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity Lall_RS is
5      generic (Lall_datawidth: integer := 32);
6      port (
7         Lall_clock, Lall_write, Lall_read, Lall_enable: in std_logic;
8         Lall_content: in std_logic_vector(Lall_datawidth-1 downto 0);
9         Lall_result: out std_logic_vector(Lall_datawidth-1 downto 0));
10  end Lall_RS;
11
12  architecture arch of Lall_RS is
13
14     signal Lall_memory: std_logic_vector(Lall_datawidth-1 downto 0);
15
16     begin
17        P1: process(Lall_clock, Lall_write)
18        begin
19           if(rising_edge(Lall_clock) and Lall_write = '1')
20              then Lall_memory <= Lall_content;
21           end if;
22        end process P1;
23
24        P2: process(Lall_read, Lall_enable, Lall_memory)
25        begin
26           if(Lall_read = '1' and Lall_enable = '1')
27              then Lall_result <= Lall_memory;
28           elsif(Lall_enable = '0')
29              then Lall_result <= (others => '0');
30           end if;
31        end process P2;
32  end arch;
```

Figure 2. RS register VHDL code

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity Lall_RT is
5     generic (Lall_datawidth: integer := 32);
6     port (
7        Lall_clock, Lall_write, Lall_read, Lall_enable: in std_logic;
8        Lall_content: in std_logic_vector(Lall_datawidth-1 downto 0);
9        Lall_result: out std_logic_vector(Lall_datawidth-1 downto 0));
10 end Lall_RT;
11
12 architecture arch of Lall_RT is
13
14    signal Lall_memory: std_logic_vector(Lall_datawidth-1 downto 0);
15
16    begin
17       P1: process(Lall_clock, Lall_write)
18       begin
19          if(rising_edge(Lall_clock) and Lall_write = '1')
20             then Lall_memory <= Lall_content;
21          end if;
22       end process P1;
23
24       P2: process(Lall_read, Lall_enable, Lall_memory)
25       begin
26          if(Lall_read = '1' and Lall_enable = '1')
27             then Lall_result <= Lall_memory;
28          elsif(Lall_enable = '0')
29             then Lall_result <= (others => '0');
30          end if;
31       end process P2;
32 end arch;
```

Figure 3. RT register VHDL code

Those are the three register files we had to create that IR register accesses based on the instructions. They are all 32-bit registers where RS and RT are used to store the two data values and RD writes the output signal once the arithmetic logic is complete.

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.STD_LOGIC_SIGNED.ALL;
4   use IEEE.numeric_std.ALL;
5
6   entity Lall_NBit_AddSub is
7       generic(n: integer := 32);
8       port( Lall_a, Lall_b: in std_logic_vector(n-1 downto 0);
9             Lall_sum: out std_logic_vector(n-1 downto 0);
10            Lall_cin: in std_logic;
11            Lall_op: in std_logic;
12            Lall_cout, Lall_O, Lall_N, Lall_Z: out std_logic);
13  end Lall_NBit_AddSub;
14
15  architecture nbitaddsub_behav of Lall_NBit_AddSub is
16      signal result: std_logic_vector(n-1 downto 0);
17      begin
18          result <= (Lall_a + Lall_b) WHEN (Lall_op = '0') ELSE (Lall_a - Lall_b);
19          Lall_cout <= '1' WHEN (Lall_op = '0' AND Lall_a(n-1) = Lall_b(n-1) AND result(n-1) /= Lall_a(n-1)) OR
20                                (Lall_op = '1' AND Lall_a(n-1) /= Lall_b(n-1) AND result(n-1) /= Lall_a(n-1)) ELSE '0';
21          Lall_O <= '1' WHEN (Lall_op = '0' AND Lall_a(n-1) = Lall_b(n-1) AND result(n-1) /= Lall_a(n-1)) OR
22                                (Lall_op = '1' AND Lall_a(n-1) /= Lall_b(n-1) AND result(n-1) /= Lall_a(n-1)) ELSE '0';
23          Lall_sum <= result;
24          Lall_N <= result(n-1);
25
26          process(result)
27              begin
28                  if unsigned(result) = "0" then
29                      Lall_Z <= '1';
30              else
31                      Lall_Z <= '0';
32                  end if;
33          end process;
34
35  end nbitaddsub_behav;
```

Figure 4. ADD/SUB VHDL code

When the opcode is 0 or 1 representing add and sub, we use the N-bit add/sub-component that we constructed in the previous experiments to perform adder/subtraction with overflow and negative flags.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.std_logic_signed.all;
4  use IEEE.numeric_std.all;
5
6  entity Lall_ALU is
7      generic(N: integer := 32);
8      port(
9          Lall_clk: in std_logic;
10         Lall_RS_i, Lall_RT_i, Lall_MDR_i: in std_logic_vector (N-1 downto 0); --32 bit register inputs
11         Lall_imm: in std_logic_vector (15 downto 0); --16 bit immediate
12         Lall_op: in std_logic_vector(3 downto 0); --Operation Code
13         Lall_RD_i: out std_logic_vector (N-1 downto 0); --32 bit register output
14         Lall_RD_out: out std_logic_vector (N-1 downto 0); --32 bit register output --TODO: HAVE RD OUTPUT
15         Lall_Z: out std_logic := '0'; --Zero flag
16         Lall_N: out std_logic := '0'; --Negative flag
17         Lall_O: out std_logic := '0'); --Overflow flag
18 end Lall_ALU;
19
20 architecture arch of Lall_ALU is
21     signal cout, z_temp0, n_temp0, o_temp0, z_temp1, n_temp1, o_temp1, z_temp2, n_temp2, o_temp2, z_temp3, n_temp3, o_temp3,
22             z_temp4, n_temp4, o_temp4, z_temp5, n_temp5, o_temp5, z_temp6, n_temp6, o_temp6, mar_wren, mdr_wren : std_logic := '0';
23     signal temp0, temp1, temp2, temp3, temp4, temp5, temp6: std_logic_vector (N-1 downto 0);
24     signal RS_o, RT_o, RT_out, RD_i, ext_o, MAR_o, MDR_o: std_logic_vector (N-1 downto 0);
25     signal imm_o: std_logic_vector (15 downto 0);
26
27     component Lall_RD is
28         generic (Lall_datawidth: integer := 32);
29         port (
30             Lall_clock, Lall_write, Lall_read, Lall_enable: in std_logic;
31             Lall_content: in std_logic_vector(Lall_datawidth-1 downto 0);
32             Lall_result: out std_logic_vector(Lall_datawidth-1 downto 0));
33     end component;
34
35     component Lall_RT is
36         generic (Lall_datawidth: integer := 32);
37         port (
38             Lall_clock, Lall_write, Lall_read, Lall_enable: in std_logic;
39             Lall_content: in std_logic_vector(Lall_datawidth-1 downto 0);
40             Lall_result: out std_logic_vector(Lall_datawidth-1 downto 0));
41     end component;
```

Figure 5. ALU VHDL code part 1

```
42
43     component Lall_RS is
44         generic (Lall_datawidth: integer := 32);
45         port (
46             Lall_clock, Lall_write, Lall_read, Lall_enable: in std_logic;
47             Lall_content: in std_logic_vector(Lall_datawidth-1 downto 0);
48             Lall_result: out std_logic_vector(Lall_datawidth-1 downto 0));
49     end component;
50
51     component Lall_IMM16 is
52         generic (Lall_imm_size: integer := 16);
53         port (
54             Lall_clock, Lall_write, Lall_read, Lall_enable: in std_logic;
55             Lall_in: in std_logic_vector(Lall_imm_size-1 downto 0);
56             Lall_out: out std_logic_vector(Lall_imm_size-1 downto 0));
57     end component;
58
59     component Lall_Sign_Extend is
60     port( Lall_I: in std_logic_vector(15 downto 0);
61             Lall_Q: out std_logic_vector(31 downto 0));
62     end component;
63
64     component Lall_MAR is
65         generic (Lall_datawidth: integer := 32);
66         port (
67             Lall_clock, Lall_write, Lall_read, Lall_enable: in std_logic;
68             Lall_content: in std_logic_vector(Lall_datawidth-1 downto 0);
69             Lall_ext: in std_logic_vector(Lall_datawidth-1 downto 0);
70             Lall_result: out std_logic_vector(Lall_datawidth-1 downto 0));
71     end component;
72
73     component Lall_MDR is
74         generic (Lall_datawidth: integer := 32);
75         port (
76             Lall_clock, Lall_write, Lall_read, Lall_enable: in std_logic;
77             Lall_content: in std_logic_vector(Lall_datawidth-1 downto 0);
78             Lall_result: out std_logic_vector(Lall_datawidth-1 downto 0));
79     end component;
```

Figure 6. ALU VHDL code part 2

```
80
81    component Lall_NBit_AddSub is
82       generic(n: integer := 32);
83       port( Lall_a, Lall_b: in std_logic_vector(n-1 downto 0);
84             Lall_sum: out std_logic_vector(n-1 downto 0);
85             Lall_cin: in std_logic;
86             Lall_op: in std_logic;
87             Lall_cout, Lall_O, Lall_N, Lall_Z: out std_logic);
88    end component;
89
90    component Lall_bitwise_op is
91       generic(N: integer := 32);
92       port(
93          Lall_I0, Lall_I1, Lall_ext: in std_logic_vector (N-1 downto 0);
94          Lall_imm: in std_logic_vector (15 downto 0);
95          Lall_op: in std_logic_vector(3 downto 0);
96          Lall_Q: out std_logic_vector (N-1 downto 0);
97          Lall_Z: out std_logic := '0';
98          Lall_N: out std_logic := '0';
99          Lall_O: out std_logic := '0');
100   end component;
101
102   begin
103      -- Registers
104      RD: Lall_RD generic map (Lall_datawidth => 32) port map (Lall_clk, '1', '1', '1', RD_i, Lall_RD_i);
105      RS: Lall_RS generic map (Lall_datawidth => 32) port map (Lall_clk, '1', '1', '1', Lall_RS_i, RS_o);
106      RT: Lall_RT generic map (Lall_datawidth => 32) port map (Lall_clk, '1', '1', '1', Lall_RT_i, RT_o);
107      imm: Lall_IMM16 generic map (Lall_imm_size => 16) port map (Lall_clk, '1', '1', '1', Lall_imm, imm_o);
108      ext: Lall_Sign_Extend port map (Lall_imm, ext_o);
109      MAR: Lall_MAR generic map (Lall_datawidth => 32) port map (Lall_clk, '1', '1', '1', RS_o, ext_o, MAR_o);
110      MDR: Lall_MDR generic map (Lall_datawidth => 32) port map (Lall_clk, '1', '1', '1', Lall_MDR_i, MDR_o);
111
112      -- Operations
113      add: Lall_NBit_AddSub generic map (n => 32) port map (RS_o, RT_o, temp0, '0', '0', cout, o_temp0, n_temp0, z_temp0);
114      addi: Lall_NBit_AddSub generic map (n => 32) port map (RS_o, ext_o, temp1, '0', '0', cout, o_temp1, n_temp1, z_temp1);
115      addiu: Lall_NBit_AddSub generic map (n => 32) port map (RS_o, ext_o, temp2, '0', '0', cout, o_temp2, n_temp2, z_temp2);
116      addu: Lall_NBit_AddSub generic map (n => 32) port map (RS_o, RT_o, temp3, '0', '0', cout, o_temp3, n_temp3, z_temp3);
117      sub: Lall_NBit_AddSub generic map (n => 32) port map (RS_o, RT_o, temp4, '0', '1', cout, o_temp4, n_temp4, z_temp4);
118      subu: Lall_NBit_AddSub generic map (n => 32) port map (RS_o, RT_o, temp5, '0', '1', cout, o_temp5, n_temp5, z_temp5);
119      bit_op: Lall_bitwise_op generic map (N => 32) port map (RS_o, RT_o, ext_o, imm_o, Lall_op, temp6, z_temp6, n_temp6,  o_temp6);
120
```

Figure 7. ALU VHDL code part 3

```
120
121      -- Assign results
122      P1: process(temp0, temp1, temp2, temp3, temp4, temp5, temp6)
123         begin
124            case Lall_op is
125               when "0000"=> Lall_RD_out <= temp0; Lall_O <= o_temp0; Lall_N <= n_temp0; Lall_Z <= z_temp0;
126               when "0001"=> RT_out <= temp1; Lall_O <= o_temp1; Lall_N <= n_temp1; Lall_Z <= z_temp1;
127               when "0010"=> RT_out <= temp2; Lall_O <= '0'; Lall_N <= n_temp2; Lall_Z <= z_temp2;
128               when "0011"=> Lall_RD_out <= temp3; Lall_O <= '0'; Lall_N <= n_temp3; Lall_Z <= z_temp3;
129               when "0100"=> Lall_RD_out <= temp4; Lall_O <= o_temp4; Lall_N <= n_temp4; Lall_Z <= z_temp4;
130               when "0101"=> Lall_RD_out <= temp5; Lall_O <= '0'; Lall_N <= n_temp5; Lall_Z <= z_temp5;
131               when "0110" | "1000" | "1001"| "1011" | "1100" | "1101" => Lall_RD_out <= temp6; Lall_O <= o_temp6; Lall_N <= n_temp6
132               when "0111" | "1010" => RT_out <= temp6; Lall_O <= o_temp6; Lall_N <= n_temp6; Lall_Z <= z_temp6;
133               when "1110" => RT_out <= MAR_o;
134               when "1111" => RT_out <= MDR_o;
135               when others => Lall_RD_out <= x"00000000";
136            end case;
137         end process;
138
139   end arch;
```

Figure 8. ALU VHDL code part 4

The ALU file operates as a control unit, redirecting to other components based on the opcode to accomplish the logic operation. It redirects to the n-bit adder sub if the opcode is 0 or 1.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
use IEEE.numeric_std.all;

entity Lall_bitwise_op is
    generic(N: integer := 32);
    port(
        Lall_I0, Lall_I1, Lall_ext: in std_logic_vector (N-1 downto 0); --32 bit register inputs
        Lall_imm: in std_logic_vector (15 downto 0); --16 bit immediate
        Lall_op: in std_logic_vector(3 downto 0); --Operation Code
        Lall_Q: out std_logic_vector (N-1 downto 0); --32 bit register output
        Lall_Z: out std_logic := '0'; --Zero flag (0 because bitwise op)
        Lall_N: out std_logic := '0'; --Negative flag (0 because bitwise op)
        Lall_O: out std_logic := '0'); --Overflow flag (0 because bitwise op)
end Lall_bitwise_op;

architecture arch of Lall_bitwise_op is
    signal result: std_logic_vector (N-1 downto 0);

    begin
        P1: process(Lall_I0, Lall_I1, Lall_op, result)
            begin
                case Lall_op is
                    when "0110"=> result <= Lall_I0 AND Lall_I1; -- and
                    when "0111"=> result <= Lall_I0 AND Lall_ext; -- andi
                    when "1000"=> result <= Lall_I0 NOR Lall_I1; -- nor
                    when "1001"=> result <= Lall_I0 OR Lall_I1; -- or
                    when "1010"=> result <= Lall_I0 OR Lall_ext; --ori
                    when "1011"=> result <= std_logic_vector(shift_left(unsigned(Lall_I1), to_integer(unsigned(Lall_imm)))); -- sll
                    when "1100"=> result <= std_logic_vector(shift_right(unsigned(Lall_I1), to_integer(unsigned(Lall_imm)))); -- srl
                    when "1101"=> result <= std_logic_vector(shift_right(signed(Lall_I1), to_integer(unsigned(Lall_imm)))); -- sra
                    when others=> result <= x"00000000";
                end case;

        Lall_Q <= result;
        end process;
end arch;
```

Figure 9. Bitwise op VHDL code

This component is called when OPCODE points to a bitwise operation, and it, along with the sign extender, conducts bitwise operations for two 32-bit values.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Lall_Sign_Extend is
    port( Lall_I: in std_logic_vector(15 downto 0);
          Lall_Q: out std_logic_vector(31 downto 0));
end Lall_Sign_Extend;

architecture arch of Lall_Sign_Extend is
    signal extended: std_logic_vector(31 downto 0);

    begin
        Lall_Q <= std_logic_vector(resize(signed(Lall_I), extended'length));
end arch;
```

Figure 10. Sign Extender VHDL code

When we have a 32-bit value and a 16-bit IMM value, we use a sign extender to make the 16-bit value 32 bits by adding 0s to the tail of the 16-bit value. This allows us to complete the logic with two 32-bit values rather than one 32-bit and one 16-bit integer.

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3
4   entity Lall_IMM16 is
5      generic (Lall_imm_size: integer := 16);
6      port (
7         Lall_clock, Lall_write, Lall_read, Lall_enable: in std_logic;
8         Lall_in: in std_logic_vector(Lall_imm_size-1 downto 0);
9         Lall_out: out std_logic_vector(Lall_imm_size-1 downto 0));
10  end Lall_IMM16;
11
12  architecture arch of Lall_IMM16 is
13
14     signal Lall_memory: std_logic_vector(Lall_imm_size-1 downto 0);
15
16     begin
17        P1: process(Lall_clock, Lall_write)
18        begin
19           if(rising_edge(Lall_clock) and Lall_write = '1')
20              then Lall_memory <= Lall_in;
21           end if;
22        end process P1;
23
24        P2: process(Lall_read, Lall_enable, Lall_memory)
25        begin
26           if(Lall_read = '1' and Lall_enable = '1')
27              then Lall_out <= Lall_memory;
28           elsif(Lall_enable = '0')
29              then Lall_out <= (others => '0');
30           end if;
31        end process P2;
32  end arch;
```

Figure 11. IMM16 VHDL code

This 16-bit immediate register that helps us perform the immediate operations with the help of sign extenders which turns the 16-bit to 32- bits.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.STD_LOGIC_SIGNED.ALL;
use IEEE.numeric_std.ALL;

entity Lall_MAR is
    generic (Lall_datawidth: integer := 32);
    port (
        Lall_clock, Lall_write, Lall_read, Lall_enable: in std_logic;
        Lall_content: in std_logic_vector(Lall_datawidth-1 downto 0);
        Lall_ext: in std_logic_vector(Lall_datawidth-1 downto 0);
        Lall_result: out std_logic_vector(Lall_datawidth-1 downto 0));
end Lall_MAR;

architecture arch of Lall_MAR is

    signal Lall_memory: std_logic_vector(Lall_datawidth-1 downto 0);

    begin
        P1: process(Lall_clock, Lall_write)
        begin
            if(rising_edge(Lall_clock) and Lall_write = '1')
                then Lall_memory <= Lall_content + Lall_ext;
            end if;
        end process P1;

        P2: process(Lall_read, Lall_enable, Lall_memory)
        begin
            if(Lall_read = '1' and Lall_enable = '1')
                then Lall_result <= Lall_memory;
            elsif(Lall_enable = '0')
                then Lall_result <= (others => '0');
            end if;
        end process P2;
end arch;
```

Figure 12. MAR VHDL code

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.STD_LOGIC_SIGNED.ALL;
4  use IEEE.numeric_std.ALL;
5
6  entity Lall_MDR is
7     generic (Lall_datawidth: integer := 32);
8     port (
9        Lall_clock, Lall_write, Lall_read, Lall_enable: in std_logic;
10       Lall_content: in std_logic_vector(Lall_datawidth-1 downto 0);
11       Lall_result: out std_logic_vector(Lall_datawidth-1 downto 0));
12 end Lall_MDR;
13
14 architecture arch of Lall_MDR is
15
16    signal Lall_memory: std_logic_vector(Lall_datawidth-1 downto 0);
17
18    begin
19       P1: process(Lall_clock, Lall_write)
20       begin
21          if(rising_edge(Lall_clock) and Lall_write = '1')
22             then Lall_memory <= Lall_content;
23          end if;
24       end process P1;
25
26       P2: process(Lall_read, Lall_enable, Lall_memory)
27       begin
28          if(Lall_read = '1' and Lall_enable = '1')
29             then Lall_result <= Lall_memory;
30          elsif(Lall_enable = '0')
31             then Lall_result <= (others => '0');
32          end if;
33       end process P2;
34 end arch;
```

Figure 13. MDR VHDL code

Simulation/ModelSim:

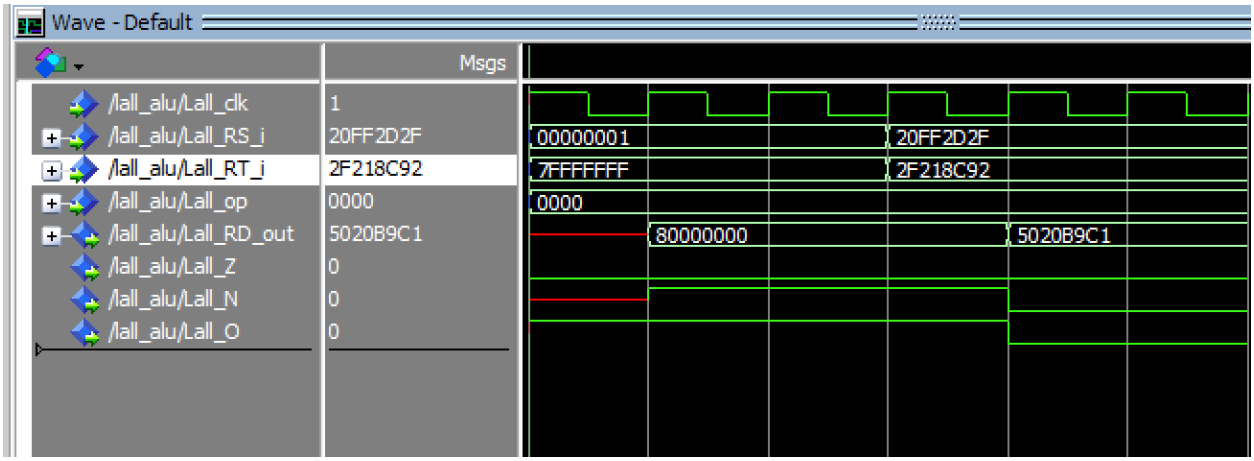

Figure 14. ADD waveform neg and overflow modelsim.

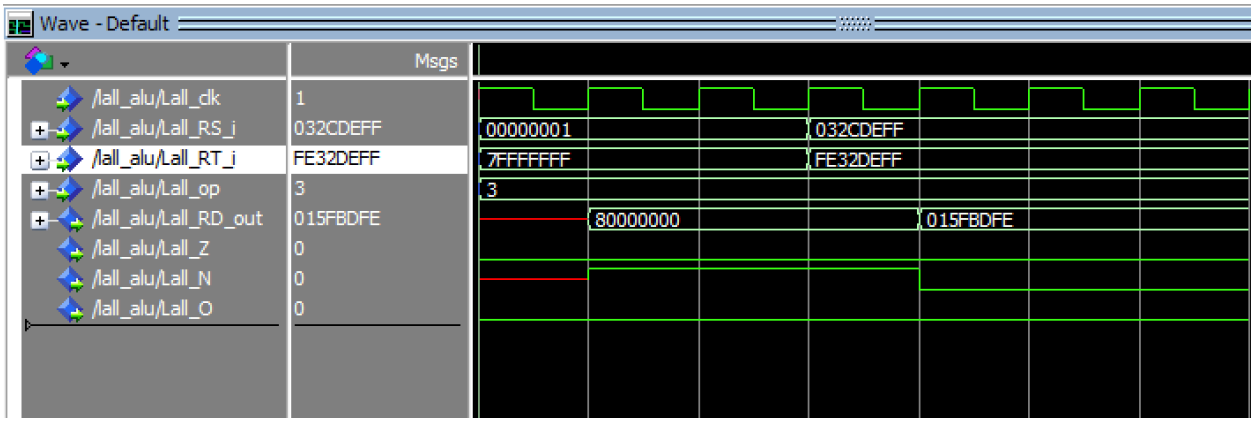

Figure 15. Add underflow waveform modelsim

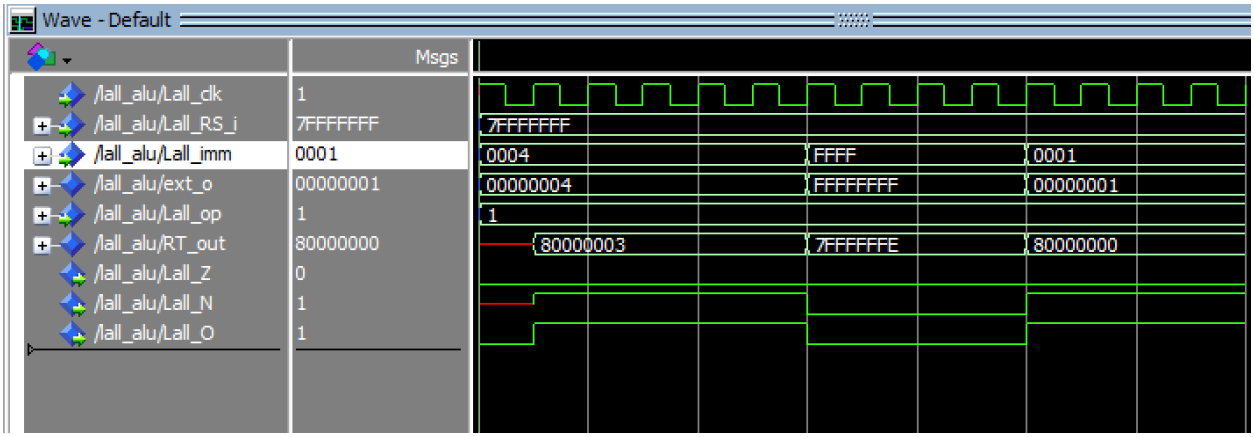

Figure 16. ADD with IMM ModelSim waveform

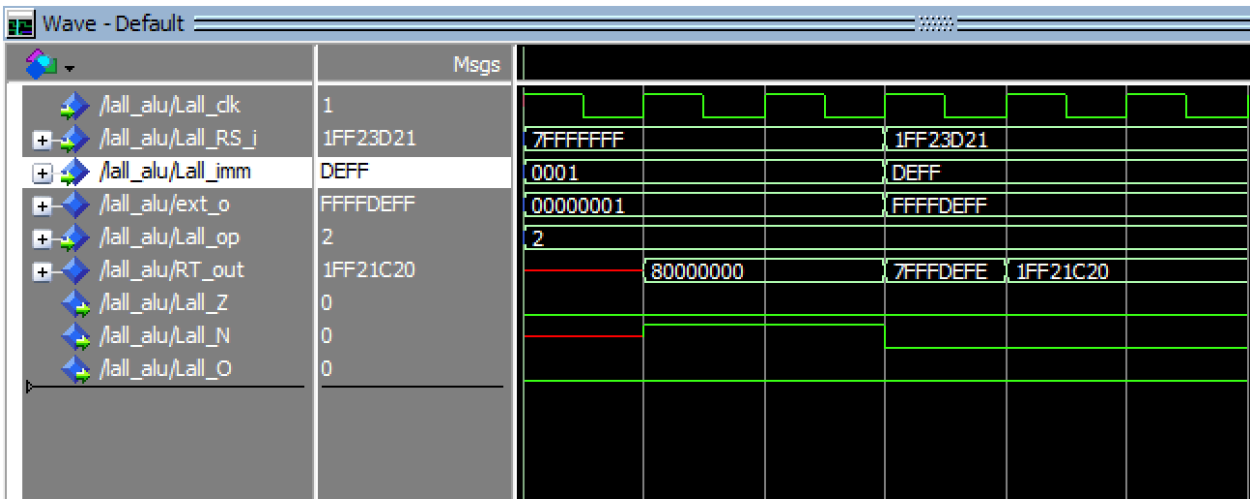

Figure 17. ADDIU IMM modelsim waveform

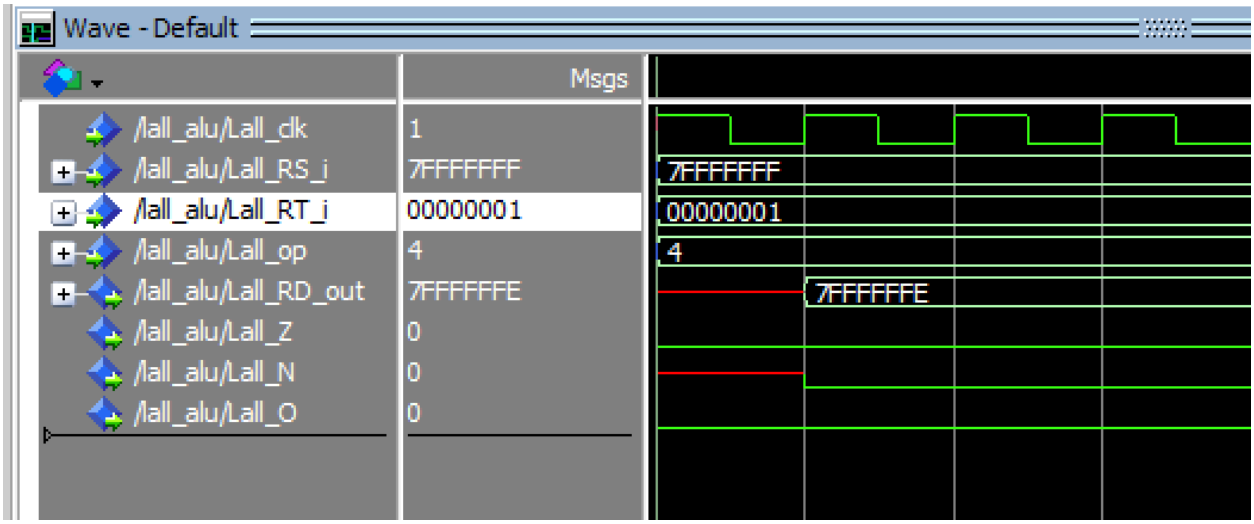

Figure 18. Subtraction of ALU modelsim waveform

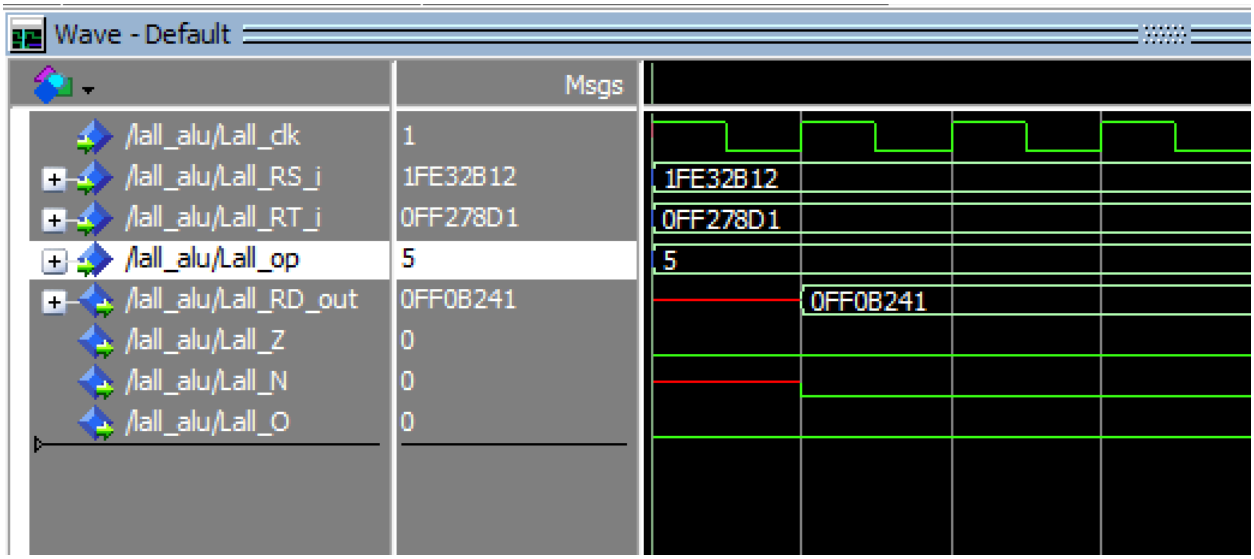

Figure 19. SUBU modelsim waveform

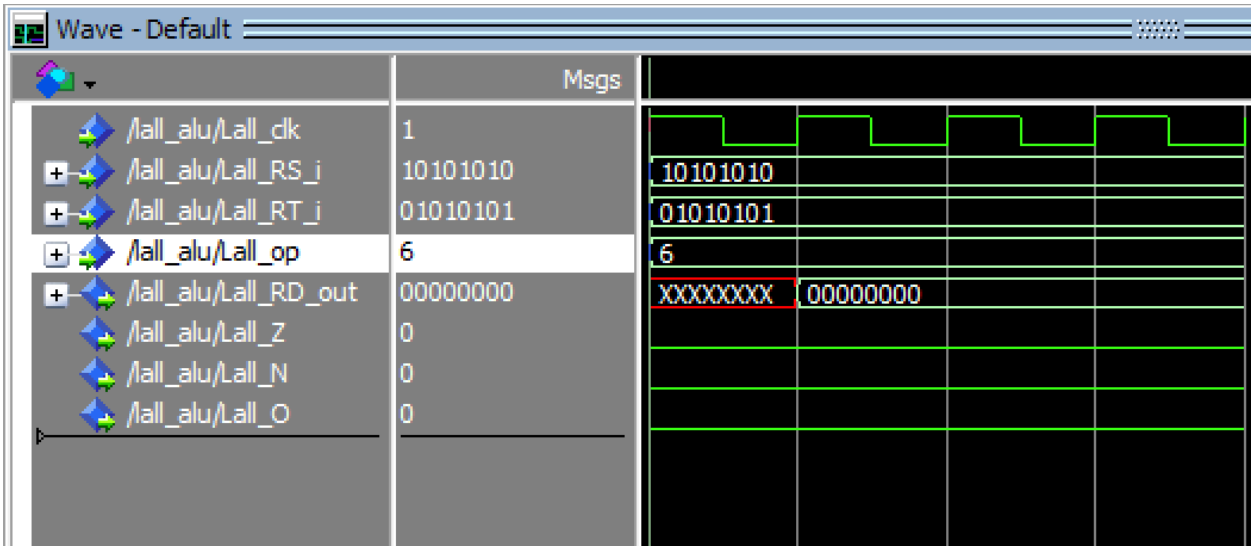

Figure 20. AND modelsim waveform

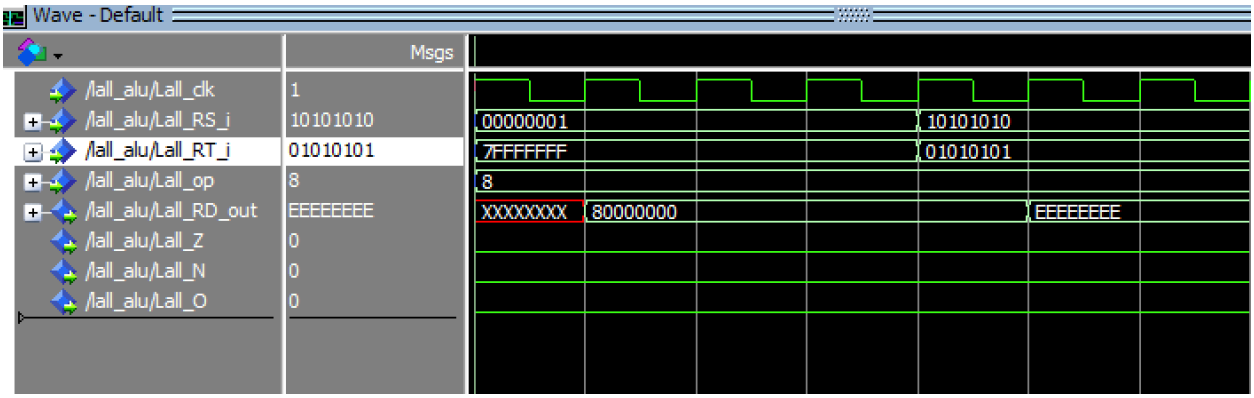

Figure 21. NOR modelsim waveform

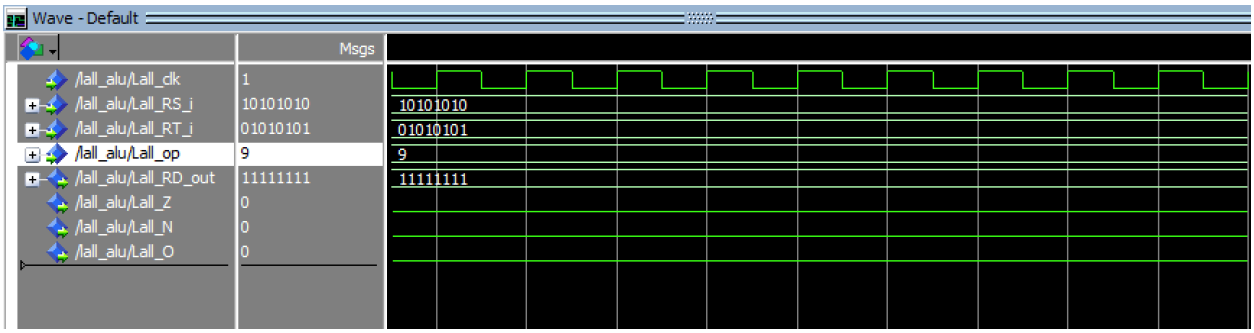

Figure 22. OR modelsim waveform

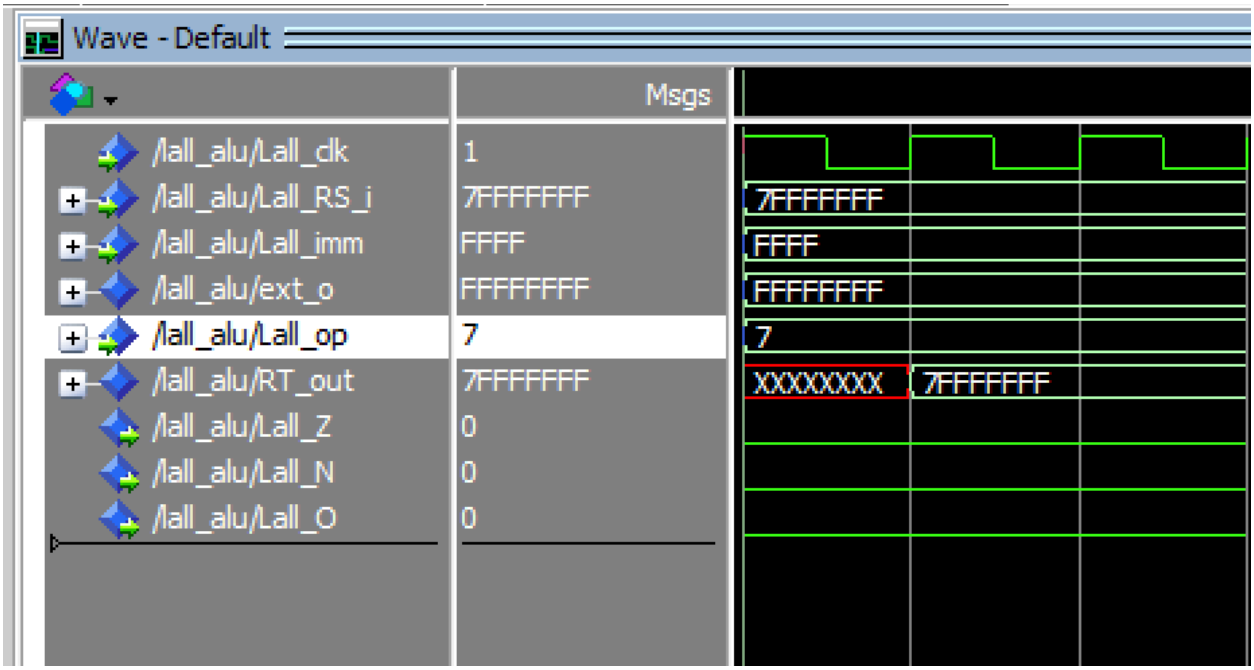

Figure 23. ADDI modelsim waveform

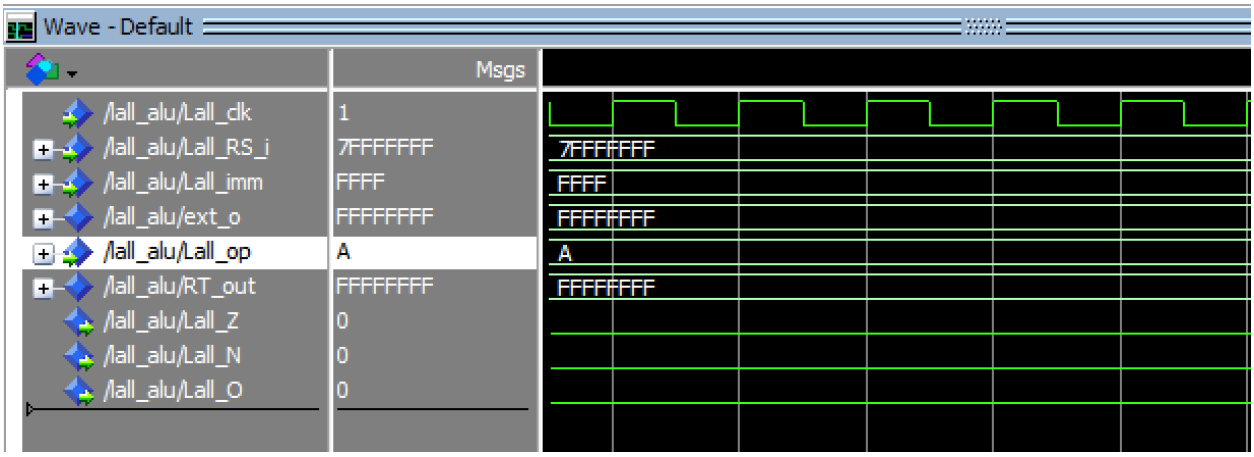

Figure 24. ORI modelsim waveform

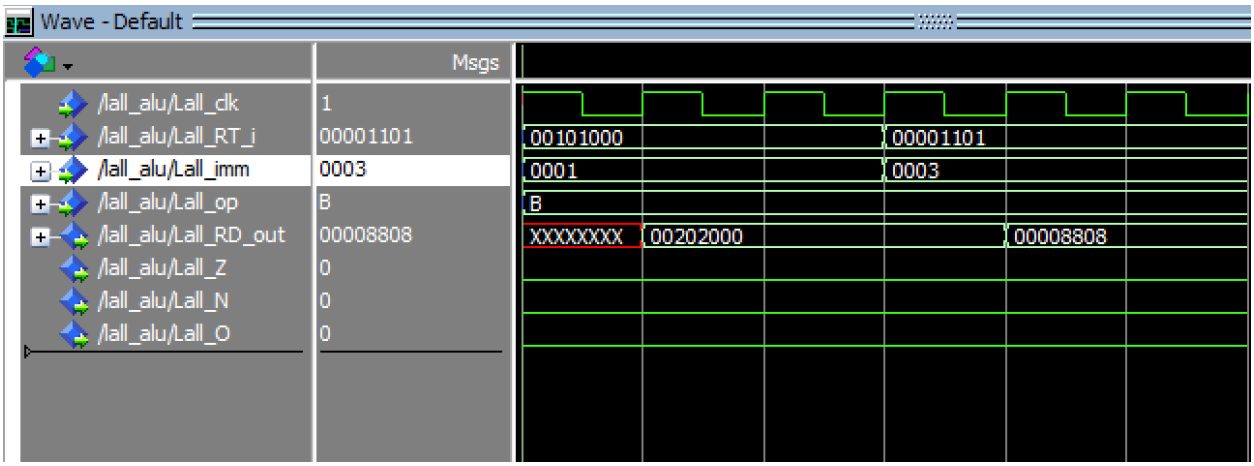

Figure 25. SLL modelsim waveform

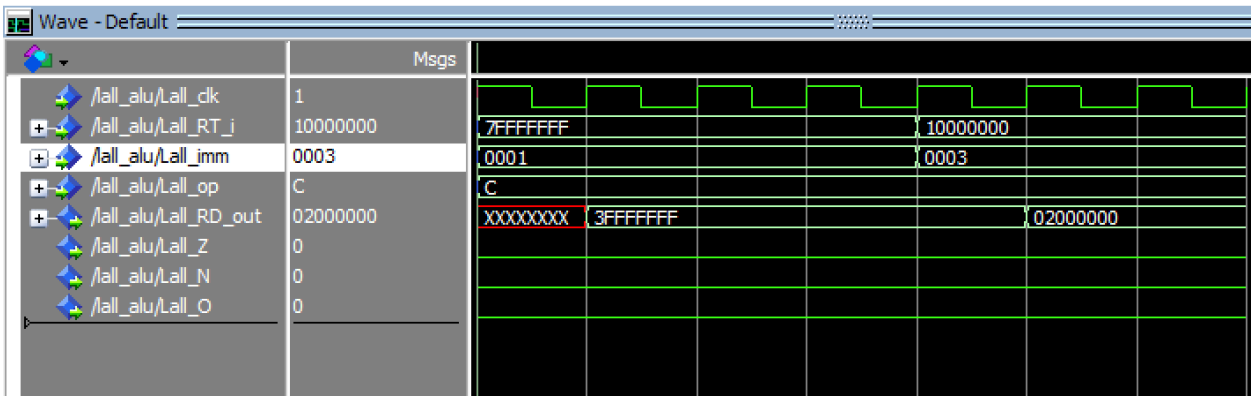

Figure 26. SRL modelsim waveform

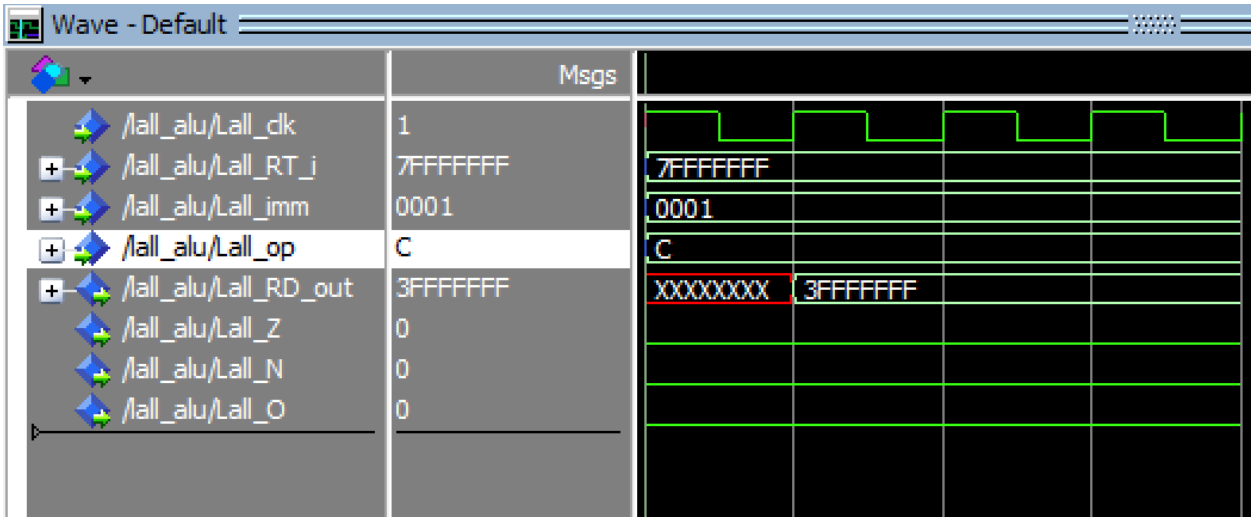

Figure 27. SRA modelsim waveform

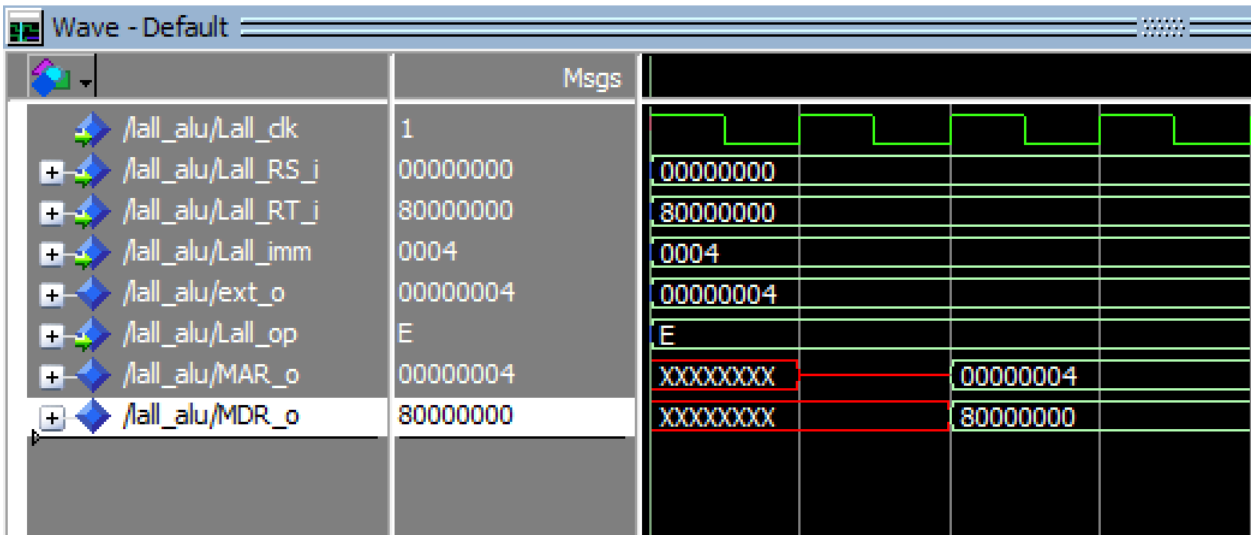

Figure 28. SW modelsim waveform

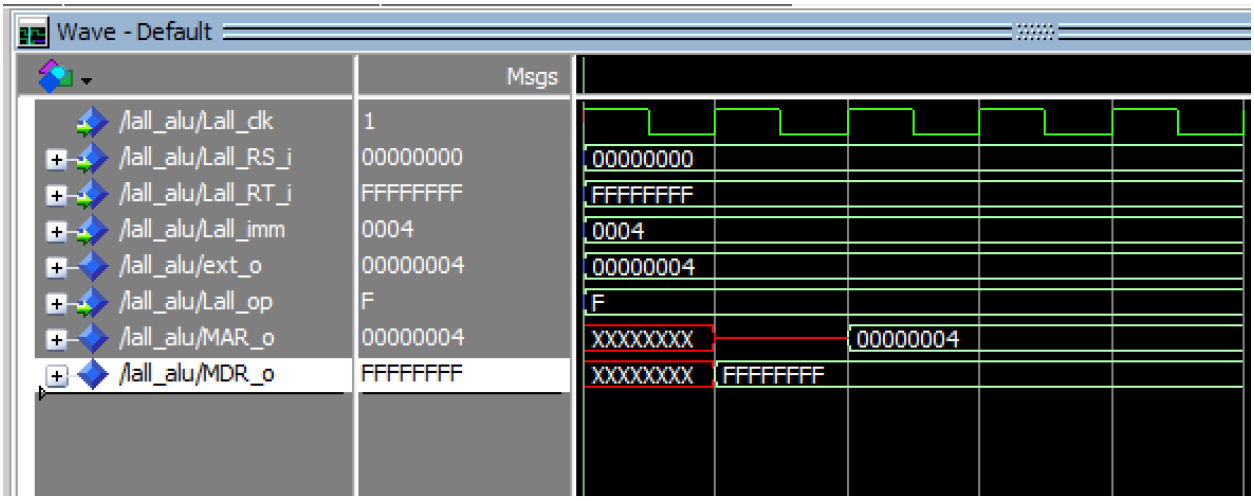

Figure 29. LW modelsim waveform

Conclusion:

We learned how to design an ALU unit capable of simulating the behavior of 16 MIPS instructions in both R-Format and I-Format formats. We learned how to create a data memory unit and how to utilize it to do operations like store and load words. We learnt how to configure and use access registers RS, RT, and RD for these instructions. Finally, we learned how arithmetic/logic operations are conducted among register values and how these values compare to the MIPS operations when these components are combined.