Christopher Lall

CSc 342/343 – Professor Gertner

Single Cycle CPU lab

Due 5/22/2022

Table of Contents

Objective:

The goal of this lab is to create a Single Cycle CPU. It will get the sum of five int's using MIPS instructions. This is a MIPS processor with lower capabilities. All of the essential components for a MIPS CPU are coded using the VHDL language and Quartus prime. W used VHDL to program various MIPS instructions in MIPS ARMS which allowed us to manipulate instructions. It allows us to demonstrate how and what we have worked with in MIPS. In the upcoming sections, you will see vhdl code of our programs that help us as they are components we need to create the CPU. Through simulation and modules and components, Quartus Prime will help us create a design that allows us to check the input and output the sum.

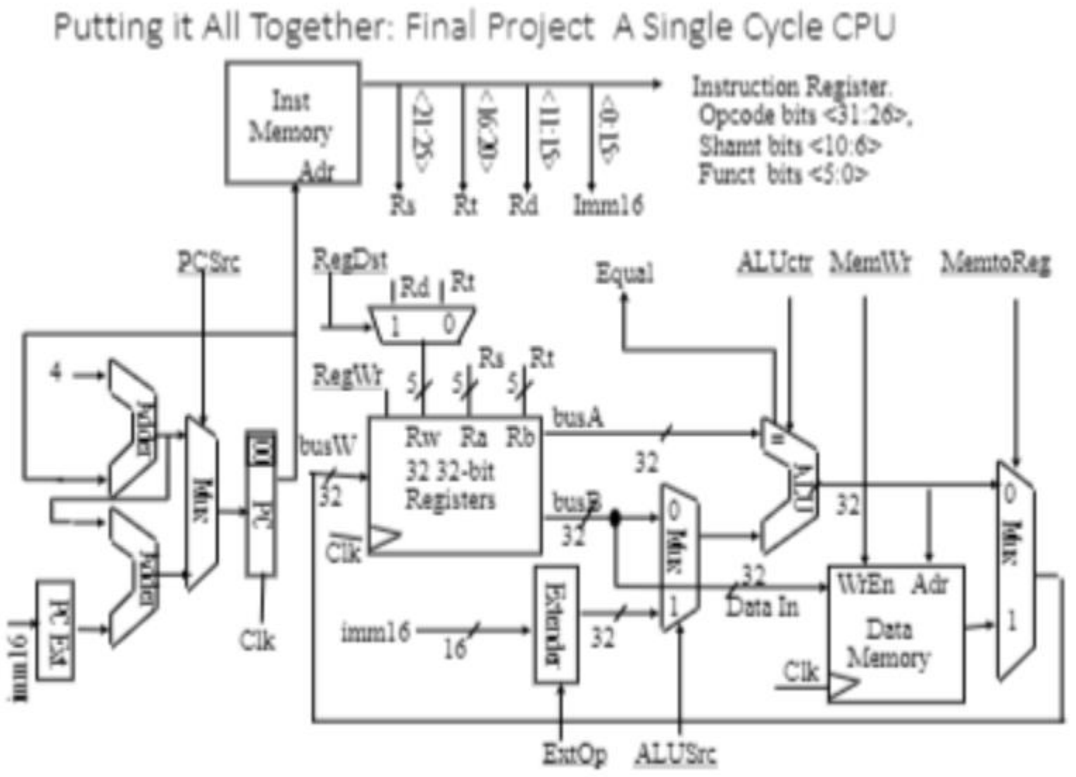
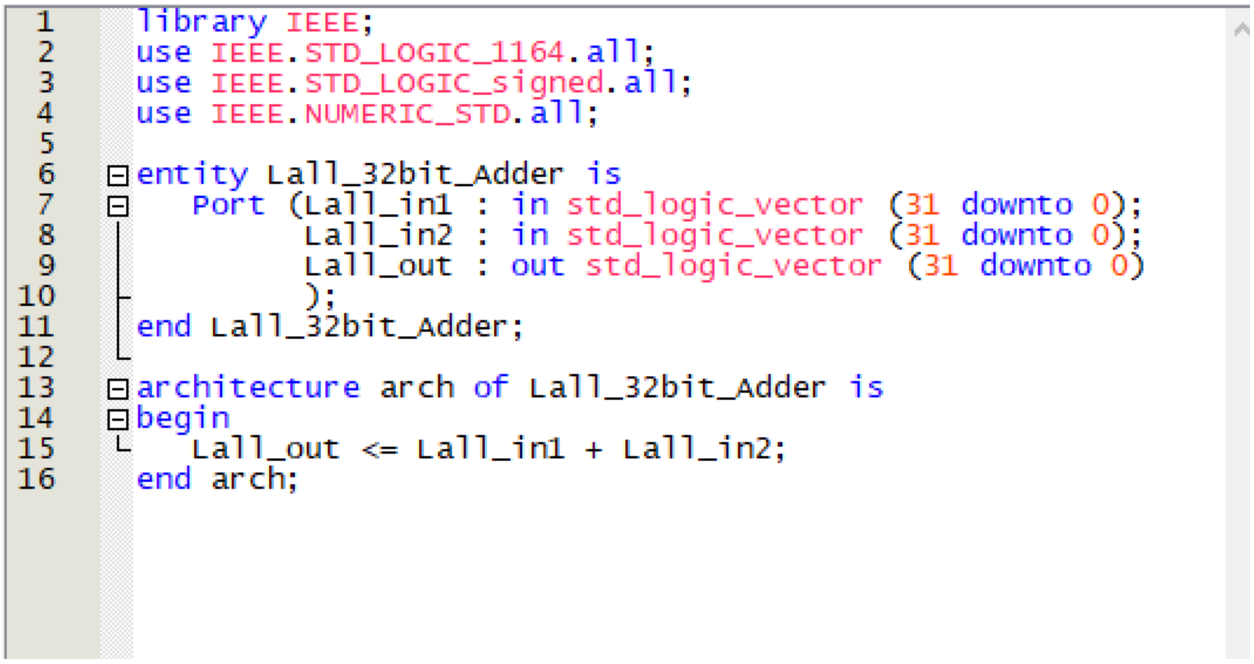


Figure 1. Single Cycle CPU schematic

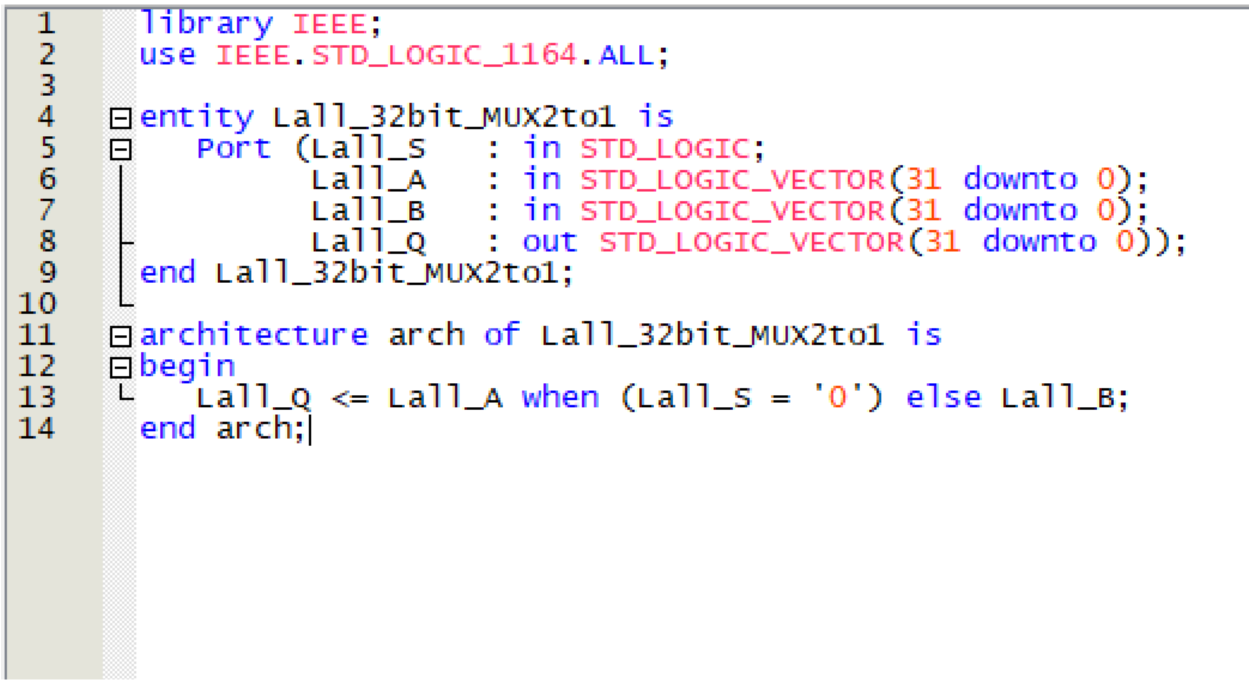
Code:

In this section I will only post the VHDL code. I will create another section to demonstrate simulation waveforms from MODELSIM. This code is in no particular order.



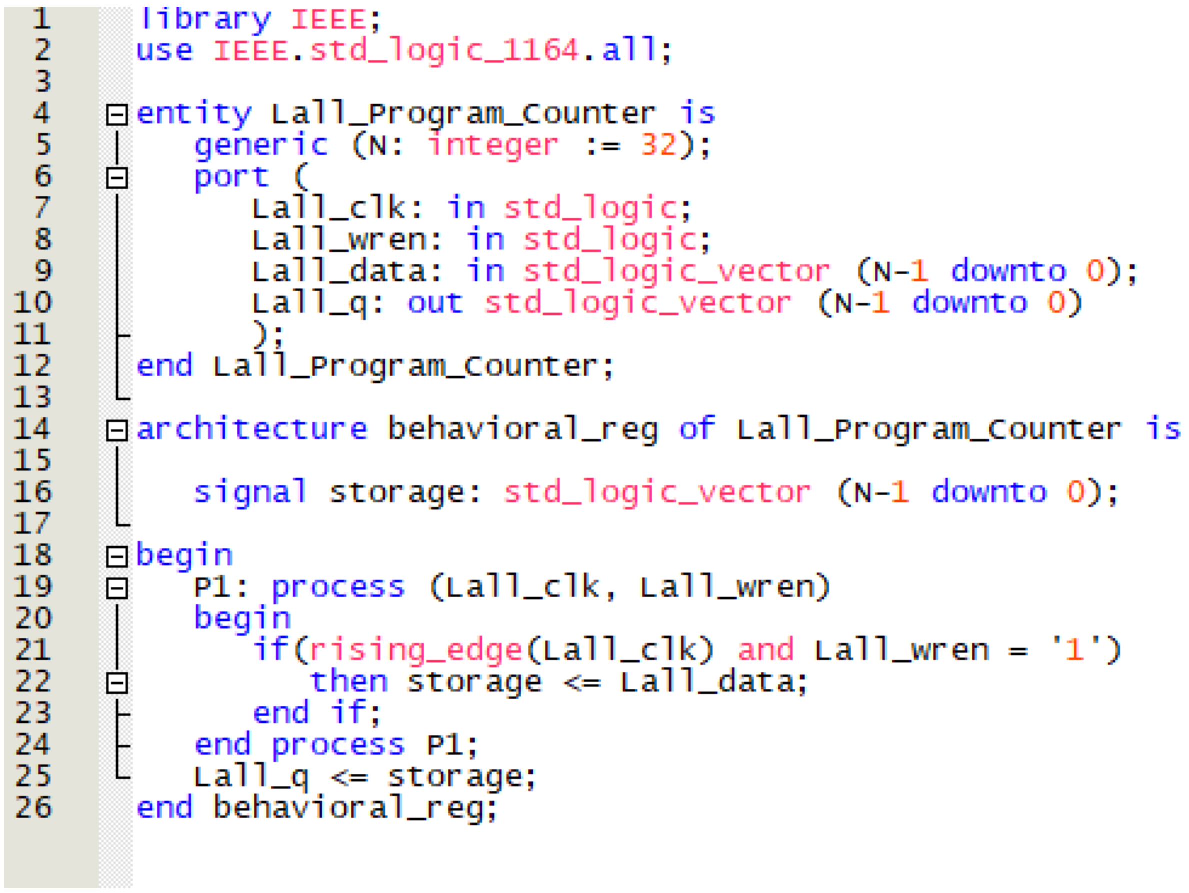
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_signed.all;
use IEEE.NUMERIC_STD.all;

entity Lall_32bit_Adder is
    Port (Lall_in1 : in std_logic_vector (31 downto 0);
          Lall_in2 : in std_logic_vector (31 downto 0);
          Lall_out : out std_logic_vector (31 downto 0)
          );
end Lall_32bit_Adder;

architecture arch of Lall_32bit_Adder is
begin
    Lall_out <= Lall_in1 + Lall_in2;
end arch;
```

Figure 2. VHDL code for adder



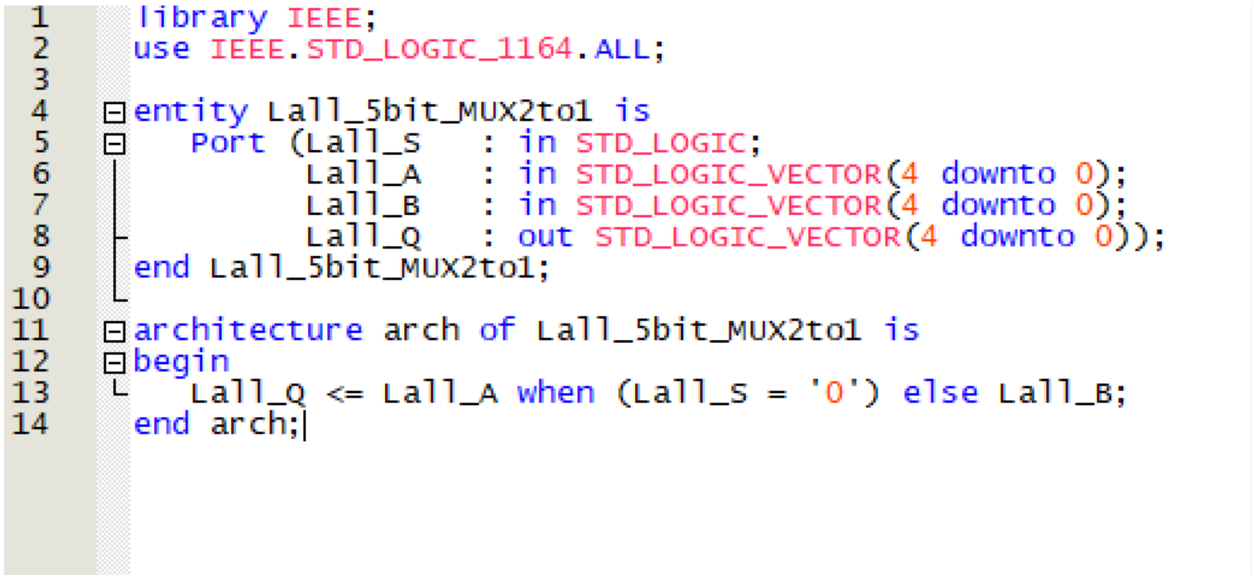
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Lall_32bit_MUX2to1 is
    Port (Lall_S   : in STD_LOGIC;
          Lall_A   : in STD_LOGIC_VECTOR(31 downto 0);
          Lall_B   : in STD_LOGIC_VECTOR(31 downto 0);
          Lall_Q   : out STD_LOGIC_VECTOR(31 downto 0));
end Lall_32bit_MUX2to1;

architecture arch of Lall_32bit_MUX2to1 is
begin
    Lall_Q <= Lall_A when (Lall_S = '0') else Lall_B;
end arch;
```

Figure 3. VHDL code for 32bit 2:1 MUX



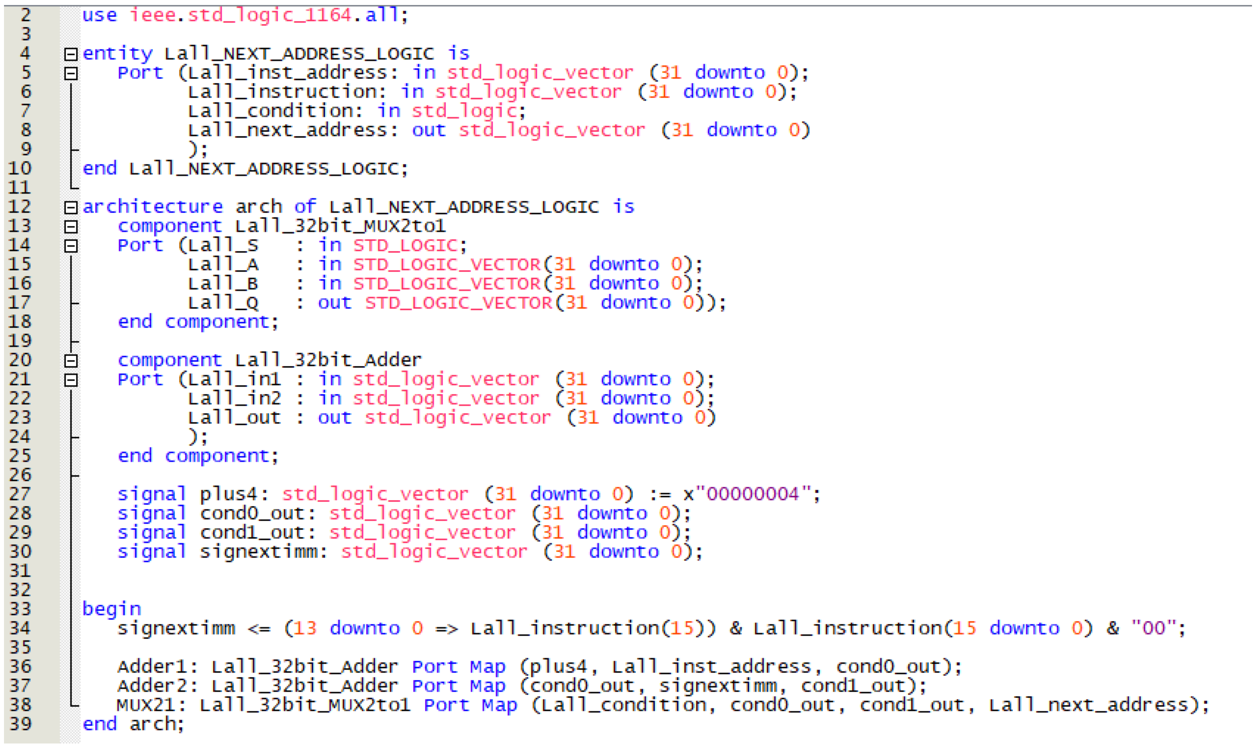
```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity Lall_Program_Counter is
   generic (N: integer := 32);
   port (
      Lall_clk: in std_logic;
      Lall_wren: in std_logic;
      Lall_data: in std_logic_vector (N-1 downto 0);
      Lall_q: out std_logic_vector (N-1 downto 0)
      );
end Lall_Program_Counter;

architecture behavioral_reg of Lall_Program_Counter is

   signal storage: std_logic_vector (N-1 downto 0);

begin
   P1: process (Lall_clk, Lall_wren)
   begin
      if(rising_edge(Lall_clk) and Lall_wren = '1')
         then storage <= Lall_data;
      end if;
   end process P1;
   Lall_q <= storage;
end behavioral_reg;
```

Figure 4. VHDL code program counter.



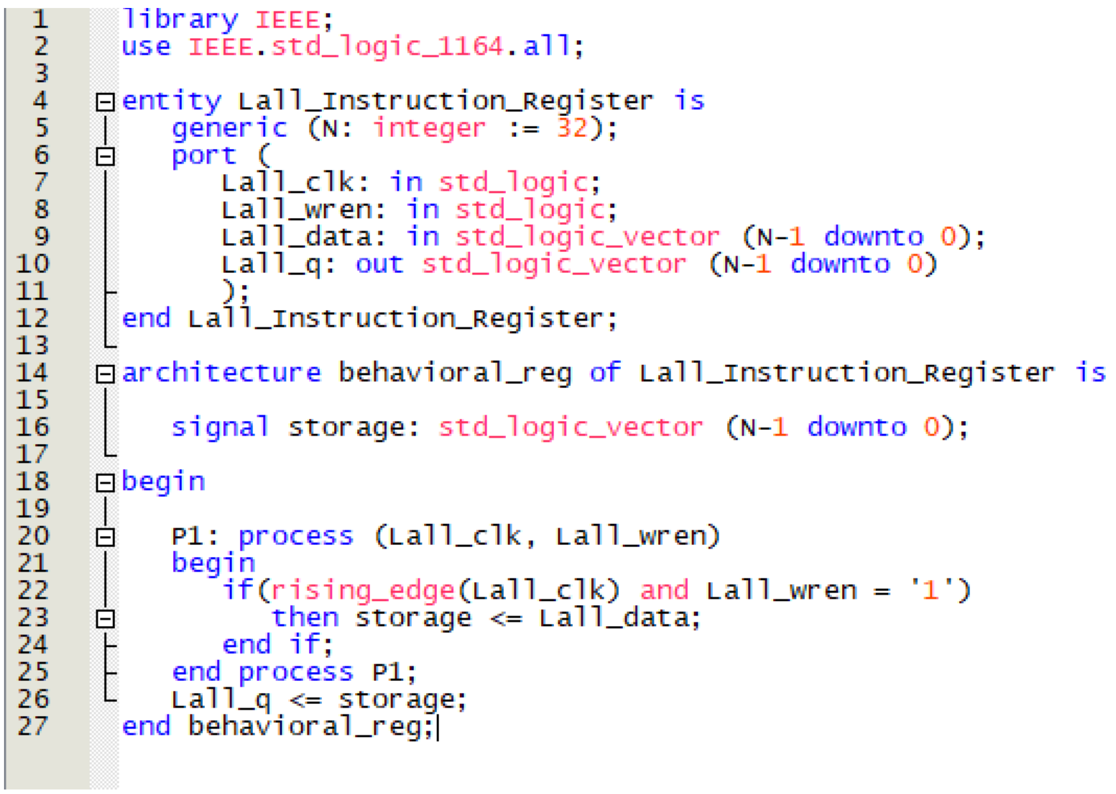
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Lall_5bit_MUX2to1 is
   Port (Lall_S    : in STD_LOGIC;
         Lall_A    : in STD_LOGIC_VECTOR(4 downto 0);
         Lall_B    : in STD_LOGIC_VECTOR(4 downto 0);
         Lall_Q    : out STD_LOGIC_VECTOR(4 downto 0));
end Lall_5bit_MUX2to1;

architecture arch of Lall_5bit_MUX2to1 is
begin
   Lall_Q <= Lall_A when (Lall_S = '0') else Lall_B;
end arch;
```

Figure 5. VHDL code 5bit 2:1 Mux



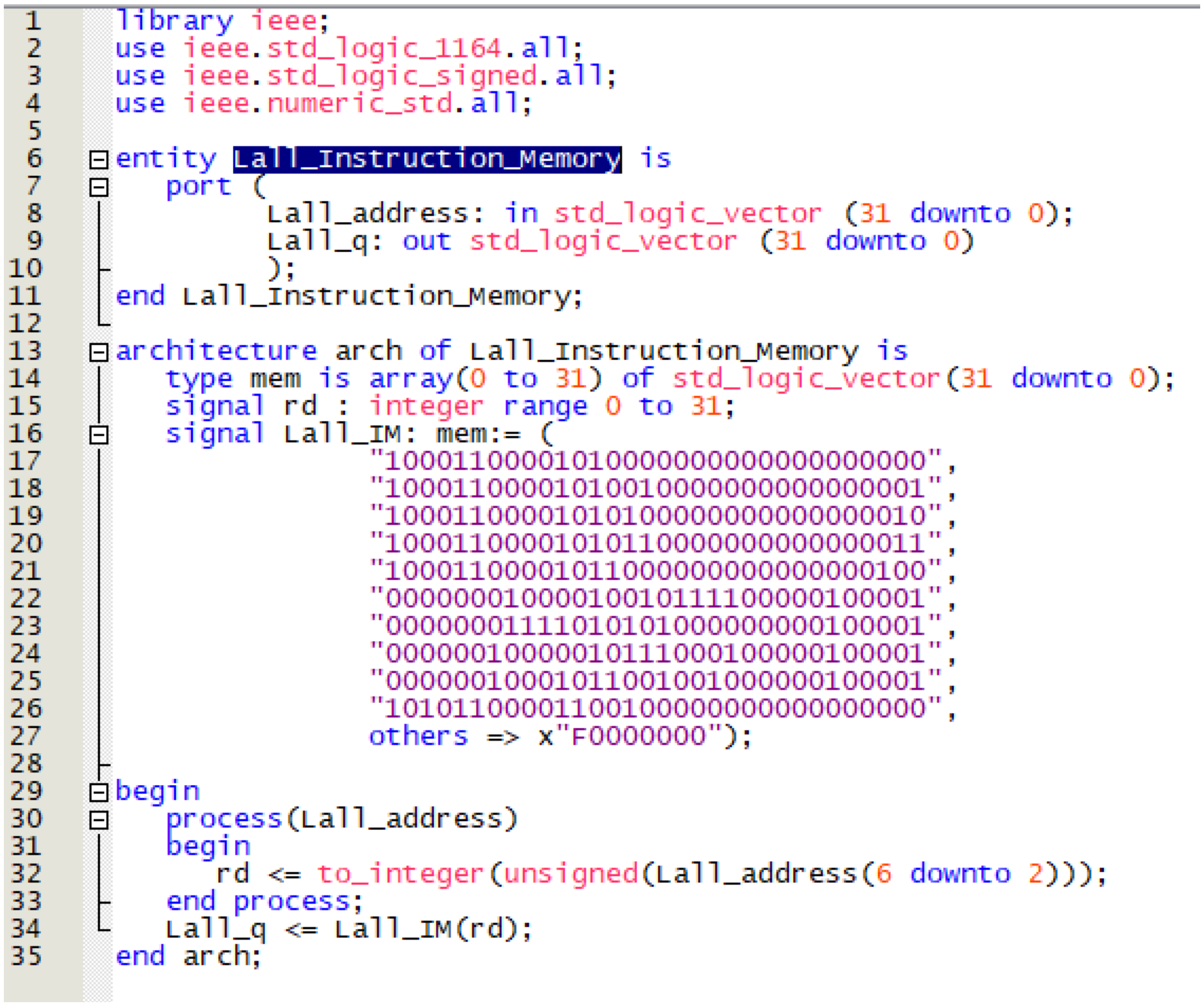
```vhdl
use ieee.std_logic_1164.all;

entity Lall_NEXT_ADDRESS_LOGIC is
   Port (Lall_inst_address: in std_logic_vector (31 downto 0);
         Lall_instruction: in std_logic_vector (31 downto 0);
         Lall_condition: in std_logic;
         Lall_next_address: out std_logic_vector (31 downto 0)
         );
end Lall_NEXT_ADDRESS_LOGIC;

architecture arch of Lall_NEXT_ADDRESS_LOGIC is
   component Lall_32bit_MUX2to1
   Port (Lall_S    : in STD_LOGIC;
         Lall_A    : in STD_LOGIC_VECTOR(31 downto 0);
         Lall_B    : in STD_LOGIC_VECTOR(31 downto 0);
         Lall_Q    : out STD_LOGIC_VECTOR(31 downto 0));
   end component;

   component Lall_32bit_Adder
   Port (Lall_in1 : in std_logic_vector (31 downto 0);
         Lall_in2 : in std_logic_vector (31 downto 0);
         Lall_out : out std_logic_vector (31 downto 0)
         );
   end component;

   signal plus4: std_logic_vector (31 downto 0) := x"00000004";
   signal cond0_out: std_logic_vector (31 downto 0);
   signal cond1_out: std_logic_vector (31 downto 0);
   signal signextimm: std_logic_vector (31 downto 0);


begin
   signextimm <= (13 downto 0 => Lall_instruction(15)) & Lall_instruction(15 downto 0) & "00";

   Adder1: Lall_32bit_Adder Port Map (plus4, Lall_inst_address, cond0_out);
   Adder2: Lall_32bit_Adder Port Map (cond0_out, signextimm, cond1_out);
   MUX21: Lall_32bit_MUX2to1 Port Map (Lall_condition, cond0_out, cond1_out, Lall_next_address);
end arch;
```

Figure 6. VHDL code Next Address Logic



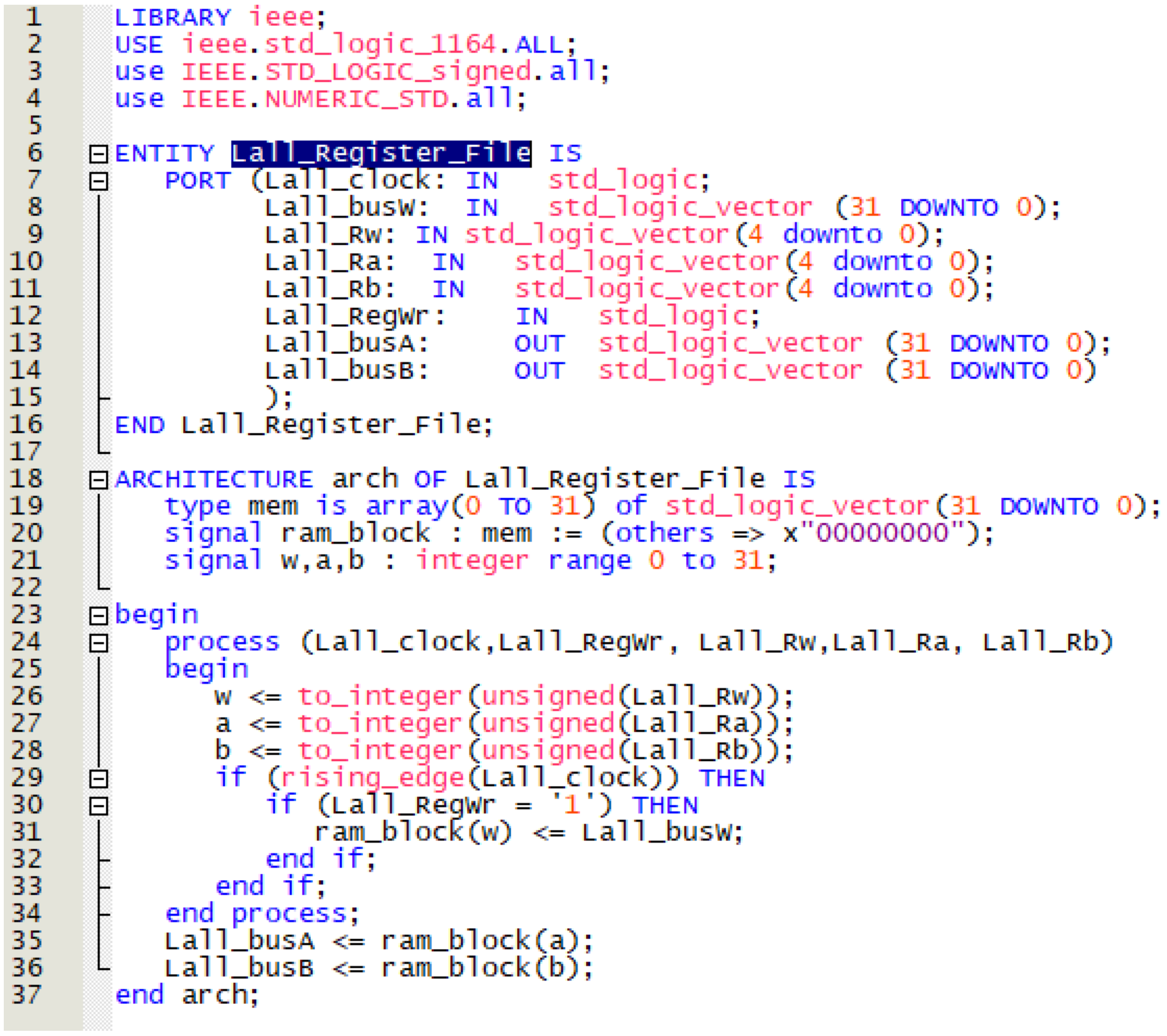
```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity Lall_Instruction_Register is
   generic (N: integer := 32);
   port (
      Lall_clk: in std_logic;
      Lall_wren: in std_logic;
      Lall_data: in std_logic_vector (N-1 downto 0);
      Lall_q: out std_logic_vector (N-1 downto 0)
      );
end Lall_Instruction_Register;

architecture behavioral_reg of Lall_Instruction_Register is

   signal storage: std_logic_vector (N-1 downto 0);

begin

   P1: process (Lall_clk, Lall_wren)
   begin
      if(rising_edge(Lall_clk) and Lall_wren = '1')
         then storage <= Lall_data;
      end if;
   end process P1;
   Lall_q <= storage;
end behavioral_reg;
```

Figure 7. VHDL code instruction register.



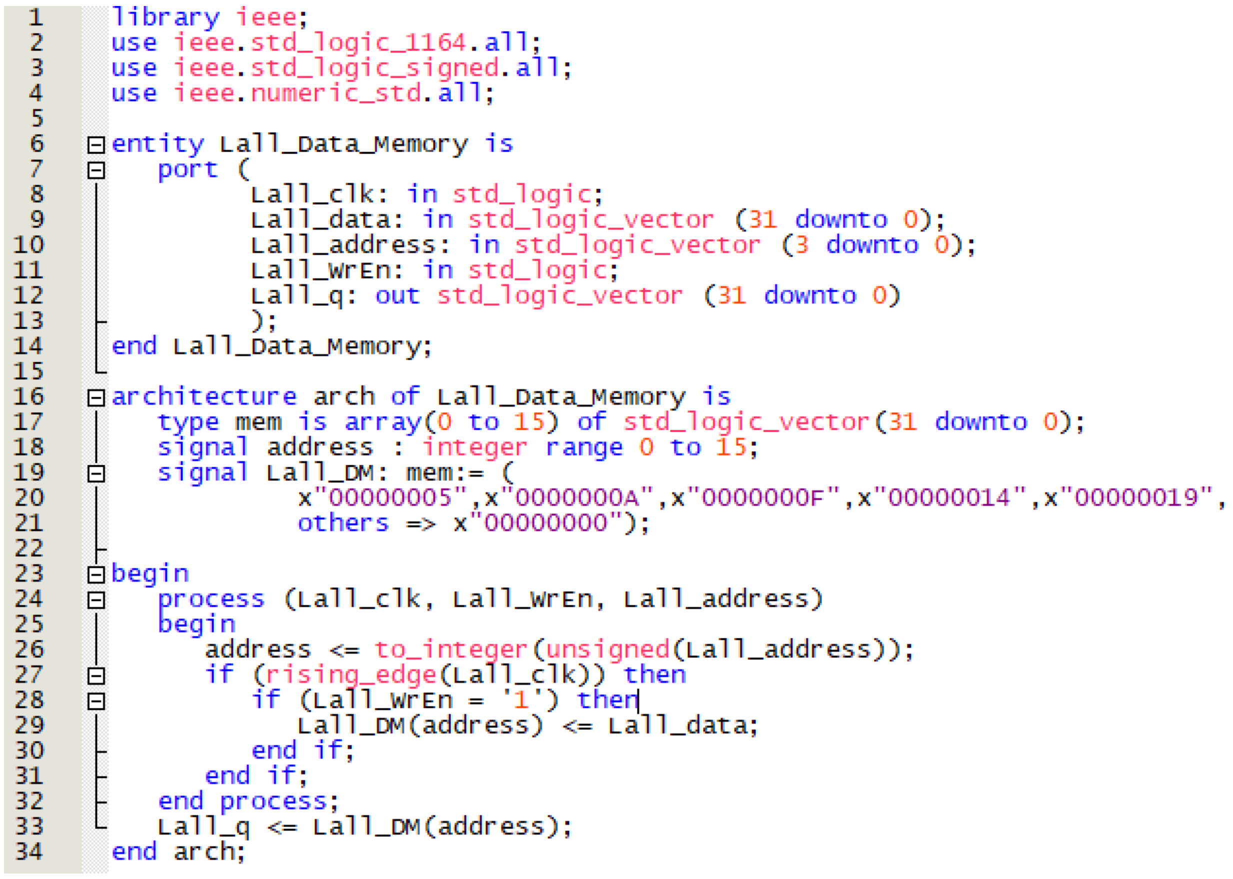
```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;

entity Lall_Instruction_Memory is
    port (
            Lall_address: in std_logic_vector (31 downto 0);
            Lall_q: out std_logic_vector (31 downto 0)
            );
end Lall_Instruction_Memory;

architecture arch of Lall_Instruction_Memory is
    type mem is array(0 to 31) of std_logic_vector(31 downto 0);
    signal rd : integer range 0 to 31;
    signal Lall_IM: mem:= (
                        "10001100001010000000000000000000",
                        "10001100001010010000000000000001",
                        "10001100001010100000000000000010",
                        "10001100001010110000000000000011",
                        "10001100001011000000000000000100",
                        "00000001000010010111100000100001",
                        "00000001111010101000000000100001",
                        "00000010000010111000100000100001",
                        "00000010001011001001000000100001",
                        "10101100001100100000000000000000",
                        others => x"F0000000");

begin
    process(Lall_address)
    begin
        rd <= to_integer(unsigned(Lall_address(6 downto 2)));
    end process;
    Lall_q <= Lall_IM(rd);
end arch;
```

Figure 8. VHDL code Instruction memory



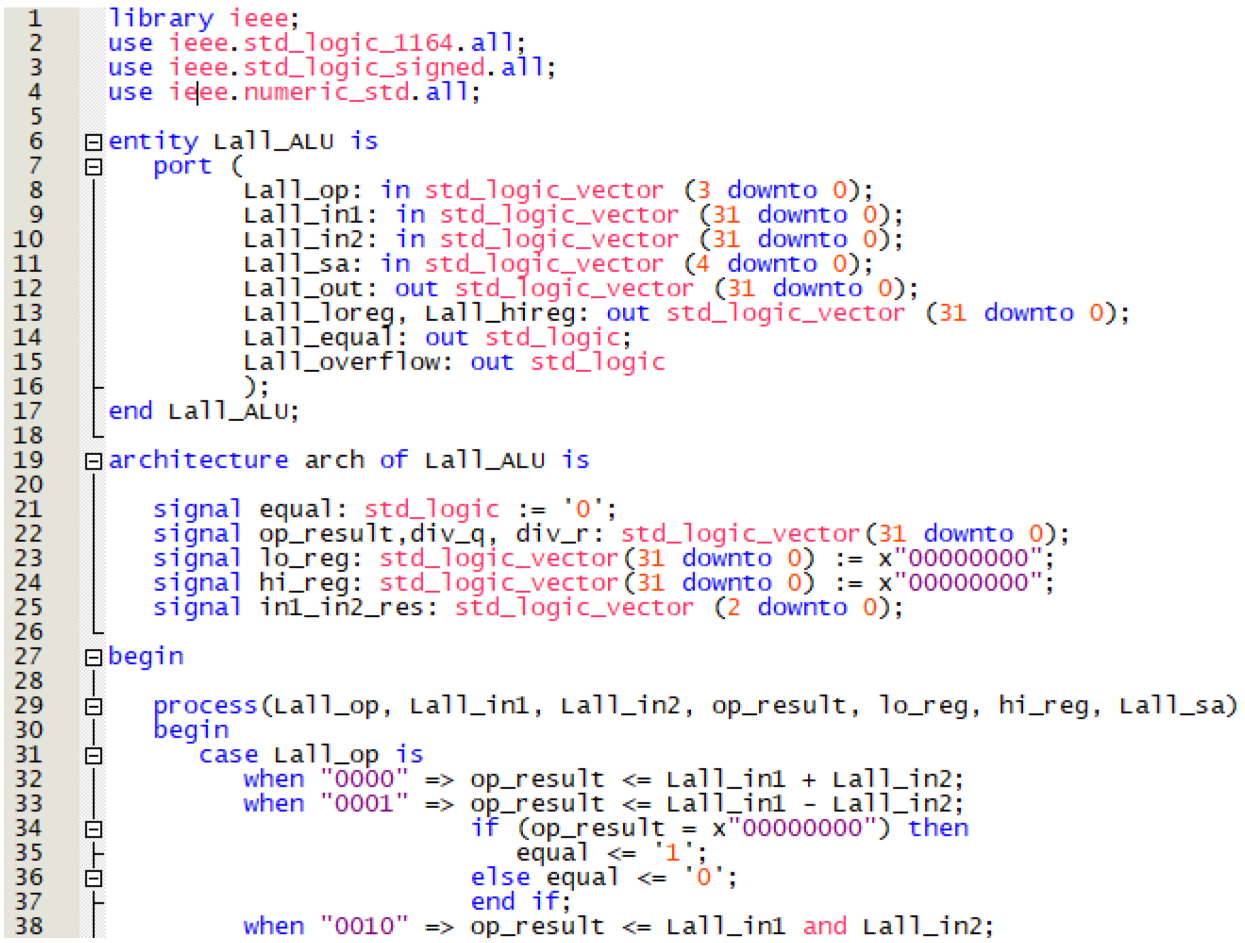
```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use IEEE.STD_LOGIC_signed.all;
use IEEE.NUMERIC_STD.all;

ENTITY Lall_Register_File IS
   PORT (Lall_clock: IN   std_logic;
         Lall_busW:  IN   std_logic_vector (31 DOWNTO 0);
         Lall_Rw: IN std_logic_vector(4 downto 0);
         Lall_Ra:  IN   std_logic_vector(4 downto 0);
         Lall_Rb:  IN   std_logic_vector(4 downto 0);
         Lall_RegWr:    IN   std_logic;
         Lall_busA:     OUT  std_logic_vector (31 DOWNTO 0);
         Lall_busB:     OUT  std_logic_vector (31 DOWNTO 0)
         );
END Lall_Register_File;

ARCHITECTURE arch OF Lall_Register_File IS
   type mem is array(0 TO 31) of std_logic_vector(31 DOWNTO 0);
   signal ram_block : mem := (others => x"00000000");
   signal w,a,b : integer range 0 to 31;

begin
   process (Lall_clock,Lall_RegWr, Lall_Rw,Lall_Ra, Lall_Rb)
   begin
      w <= to_integer(unsigned(Lall_Rw));
      a <= to_integer(unsigned(Lall_Ra));
      b <= to_integer(unsigned(Lall_Rb));
      if (rising_edge(Lall_clock)) THEN
         if (Lall_RegWr = '1') THEN
            ram_block(w) <= Lall_busW;
         end if;
      end if;
   end process;
   Lall_busA <= ram_block(a);
   Lall_busB <= ram_block(b);
end arch;
```

Figure 9. VHDL code register file.

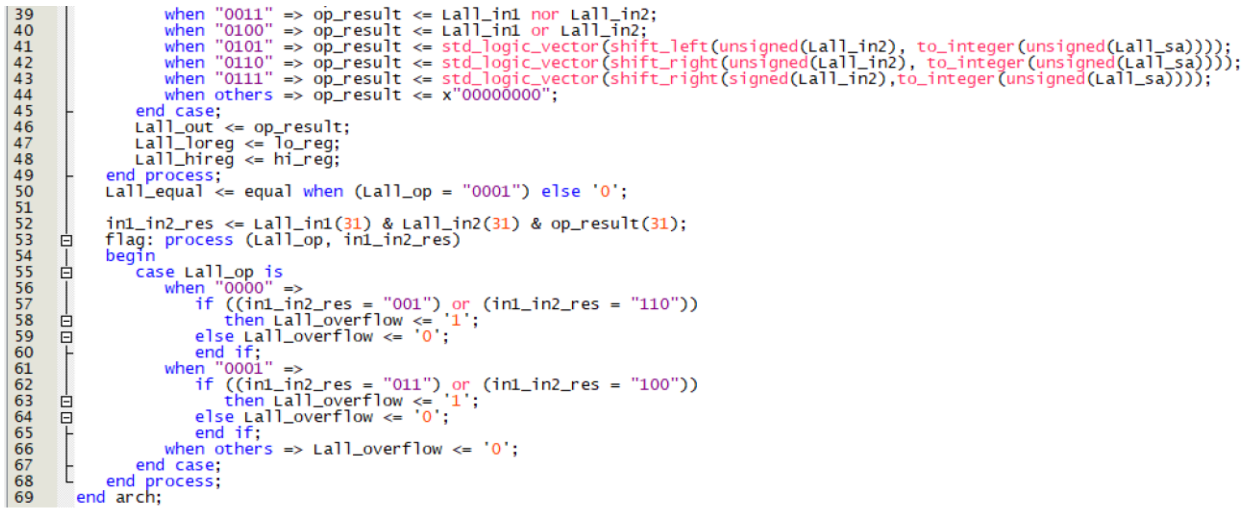


```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;

entity Lall_Data_Memory is
    port (
            Lall_clk: in std_logic;
            Lall_data: in std_logic_vector (31 downto 0);
            Lall_address: in std_logic_vector (3 downto 0);
            Lall_WrEn: in std_logic;
            Lall_q: out std_logic_vector (31 downto 0)
            );
end Lall_Data_Memory;

architecture arch of Lall_Data_Memory is
    type mem is array(0 to 15) of std_logic_vector(31 downto 0);
    signal address : integer range 0 to 15;
    signal Lall_DM: mem:= (
                x"00000005",x"0000000A",x"0000000F",x"00000014",x"00000019",
                others => x"00000000");

begin
    process (Lall_clk, Lall_WrEn, Lall_address)
    begin
        address <= to_integer(unsigned(Lall_address));
        if (rising_edge(Lall_clk)) then
            if (Lall_WrEn = '1') then
                Lall_DM(address) <= Lall_data;
            end if;
        end if;
    end process;
    Lall_q <= Lall_DM(address);
end arch;
```

Figure 10. VHDL code data memory.

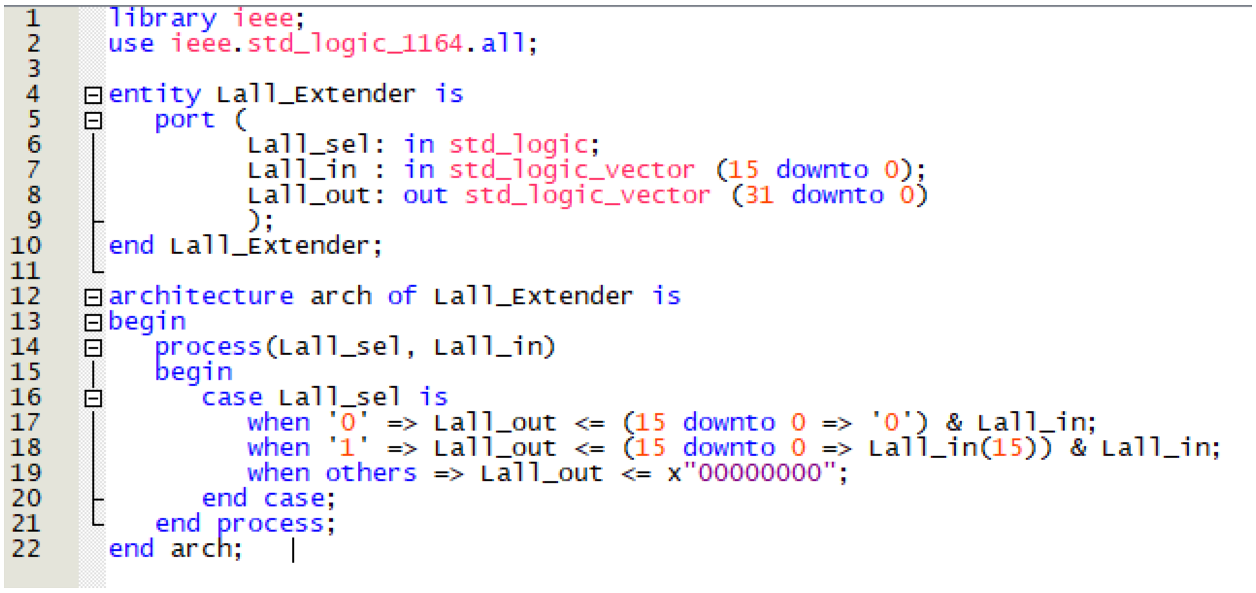


```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;

entity Lall_ALU is
   port (
         Lall_op: in std_logic_vector (3 downto 0);
         Lall_in1: in std_logic_vector (31 downto 0);
         Lall_in2: in std_logic_vector (31 downto 0);
         Lall_sa: in std_logic_vector (4 downto 0);
         Lall_out: out std_logic_vector (31 downto 0);
         Lall_loreg, Lall_hireg: out std_logic_vector (31 downto 0);
         Lall_equal: out std_logic;
         Lall_overflow: out std_logic
         );
end Lall_ALU;

architecture arch of Lall_ALU is

   signal equal: std_logic := '0';
   signal op_result,div_q, div_r: std_logic_vector(31 downto 0);
   signal lo_reg: std_logic_vector(31 downto 0) := x"00000000";
   signal hi_reg: std_logic_vector(31 downto 0) := x"00000000";
   signal in1_in2_res: std_logic_vector (2 downto 0);

begin

   process(Lall_op, Lall_in1, Lall_in2, op_result, lo_reg, hi_reg, Lall_sa)
   begin
      case Lall_op is
         when "0000" => op_result <= Lall_in1 + Lall_in2;
         when "0001" => op_result <= Lall_in1 - Lall_in2;
                        if (op_result = x"00000000") then
                           equal <= '1';
                        else equal <= '0';
                        end if;
         when "0010" => op_result <= Lall_in1 and Lall_in2;
```



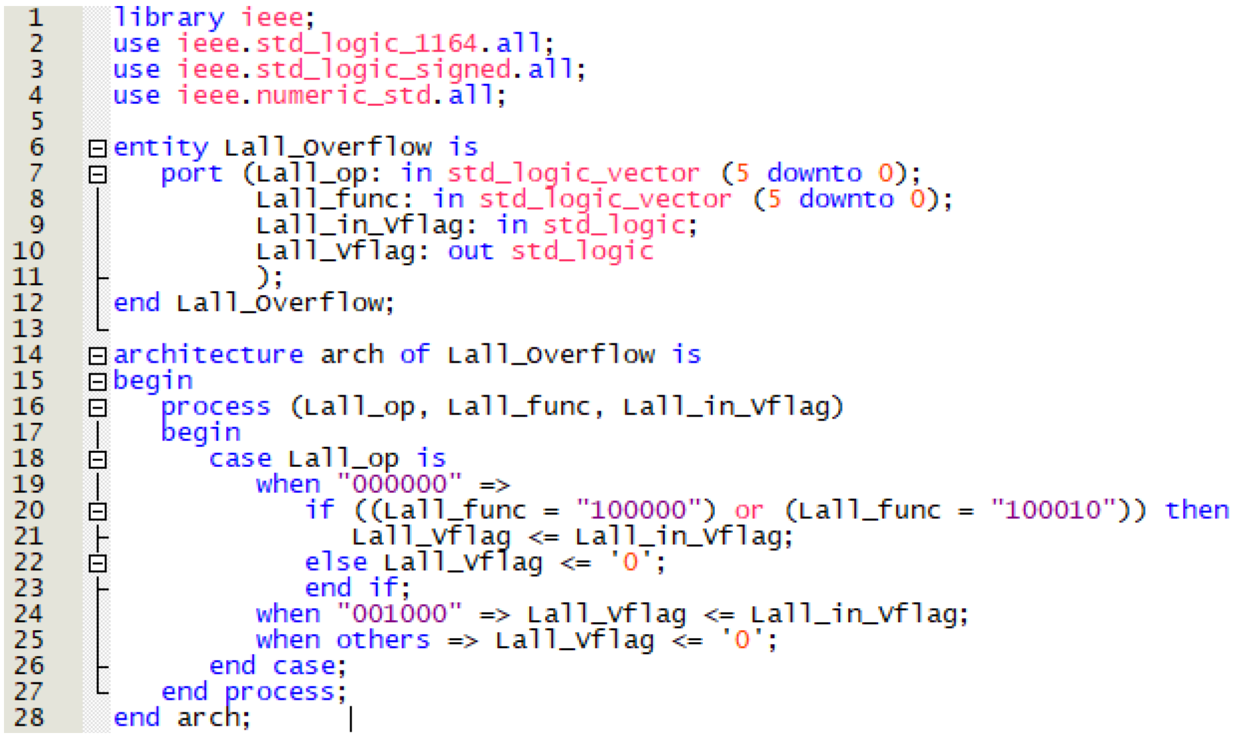
```vhdl
         when "0011" => op_result <= Lall_in1 nor Lall_in2;
         when "0100" => op_result <= Lall_in1 or Lall_in2;
         when "0101" => op_result <= std_logic_vector(shift_left(unsigned(Lall_in2), to_integer(unsigned(Lall_sa))));
         when "0110" => op_result <= std_logic_vector(shift_right(unsigned(Lall_in2), to_integer(unsigned(Lall_sa))));
         when "0111" => op_result <= std_logic_vector(shift_right(signed(Lall_in2),to_integer(unsigned(Lall_sa))));
         when others => op_result <= x"00000000";
      end case;
      Lall_out <= op_result;
      Lall_loreg <= lo_reg;
      Lall_hireg <= hi_reg;
   end process;
   Lall_equal <= equal when (Lall_op = "0001") else '0';

   in1_in2_res <= Lall_in1(31) & Lall_in2(31) & op_result(31);
   flag: process (Lall_op, in1_in2_res)
   begin
      case Lall_op is
         when "0000" =>
            if ((in1_in2_res = "001") or (in1_in2_res = "110"))
               then Lall_overflow <= '1';
            else Lall_overflow <= '0';
            end if;
         when "0001" =>
            if ((in1_in2_res = "011") or (in1_in2_res = "100"))
               then Lall_overflow <= '1';
            else Lall_overflow <= '0';
            end if;
         when others => Lall_overflow <= '0';
      end case;
   end process;
end arch;
```

Figure 11. VHDL code ALU



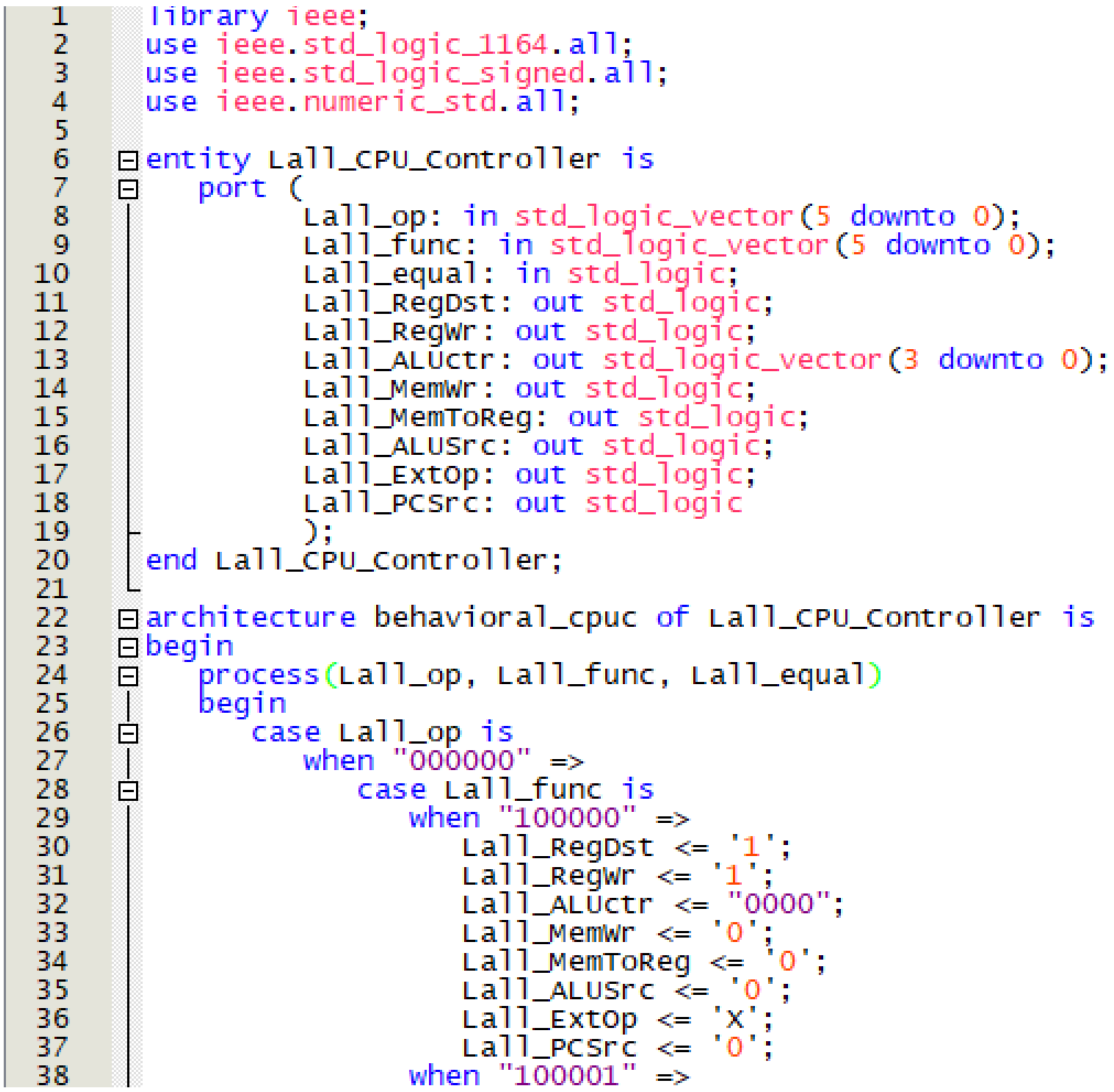
```
library ieee;
use ieee.std_logic_1164.all;

entity Lall_Extender is
    port (
            Lall_sel: in std_logic;
            Lall_in : in std_logic_vector (15 downto 0);
            Lall_out: out std_logic_vector (31 downto 0)
            );
end Lall_Extender;

architecture arch of Lall_Extender is
begin
    process(Lall_sel, Lall_in)
    begin
        case Lall_sel is
            when '0' => Lall_out <= (15 downto 0 => '0') & Lall_in;
            when '1' => Lall_out <= (15 downto 0 => Lall_in(15)) & Lall_in;
            when others => Lall_out <= x"00000000";
        end case;
    end process;
end arch;
```

Figure 12. VHDL code extender.



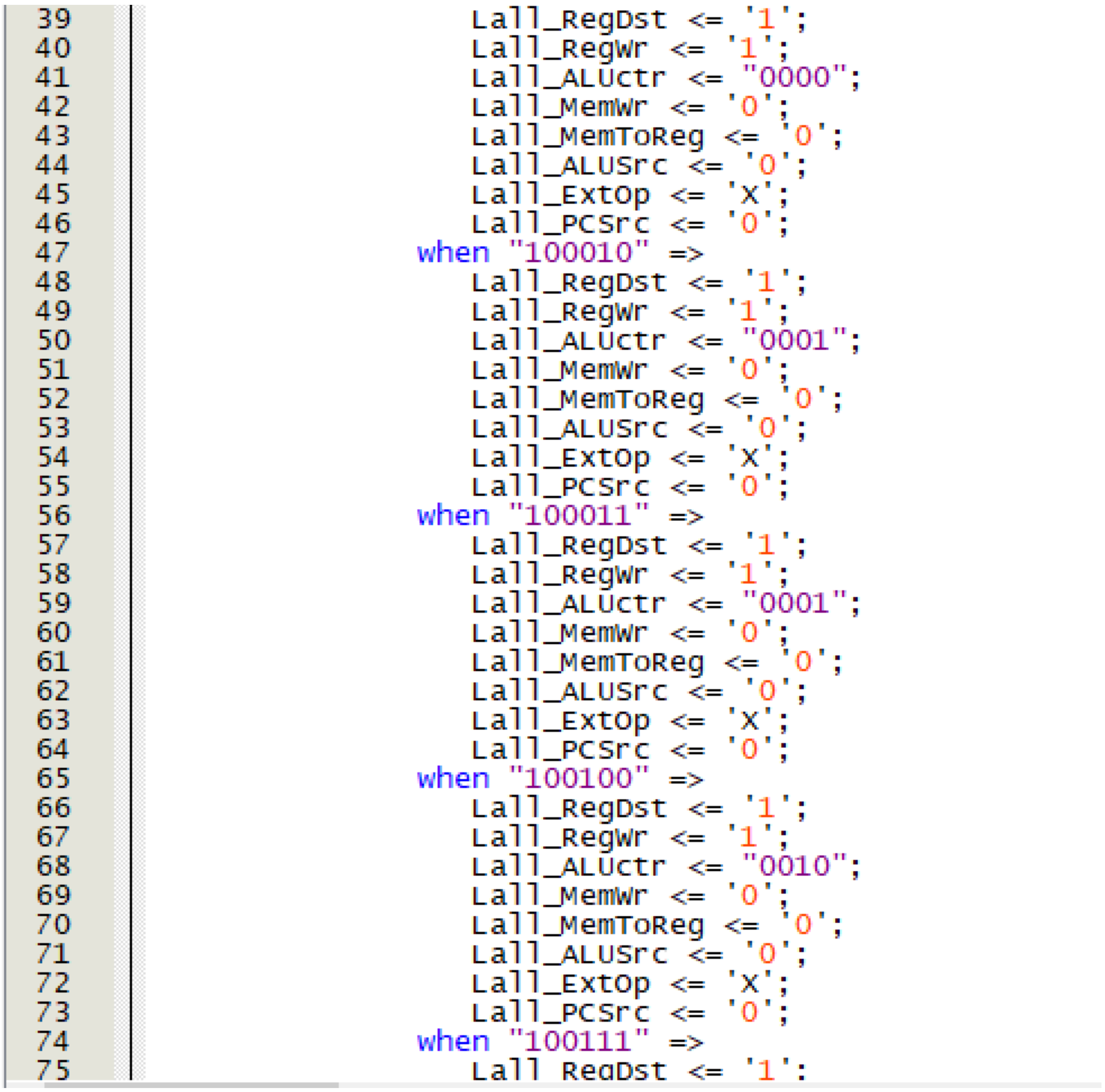
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;

entity Lall_Overflow is
    port (Lall_op: in std_logic_vector (5 downto 0);
            Lall_func: in std_logic_vector (5 downto 0);
            Lall_in_Vflag: in std_logic;
            Lall_Vflag: out std_logic
            );
end Lall_Overflow;

architecture arch of Lall_Overflow is
begin
    process (Lall_op, Lall_func, Lall_in_Vflag)
    begin
        case Lall_op is
            when "000000" =>
                if ((Lall_func = "100000") or (Lall_func = "100010")) then
                    Lall_Vflag <= Lall_in_Vflag;
                else Lall_Vflag <= '0';
                end if;
            when "001000" => Lall_Vflag <= Lall_in_Vflag;
            when others => Lall_Vflag <= '0';
        end case;
    end process;
end arch;
```

Figure 13. VHDL code overflow



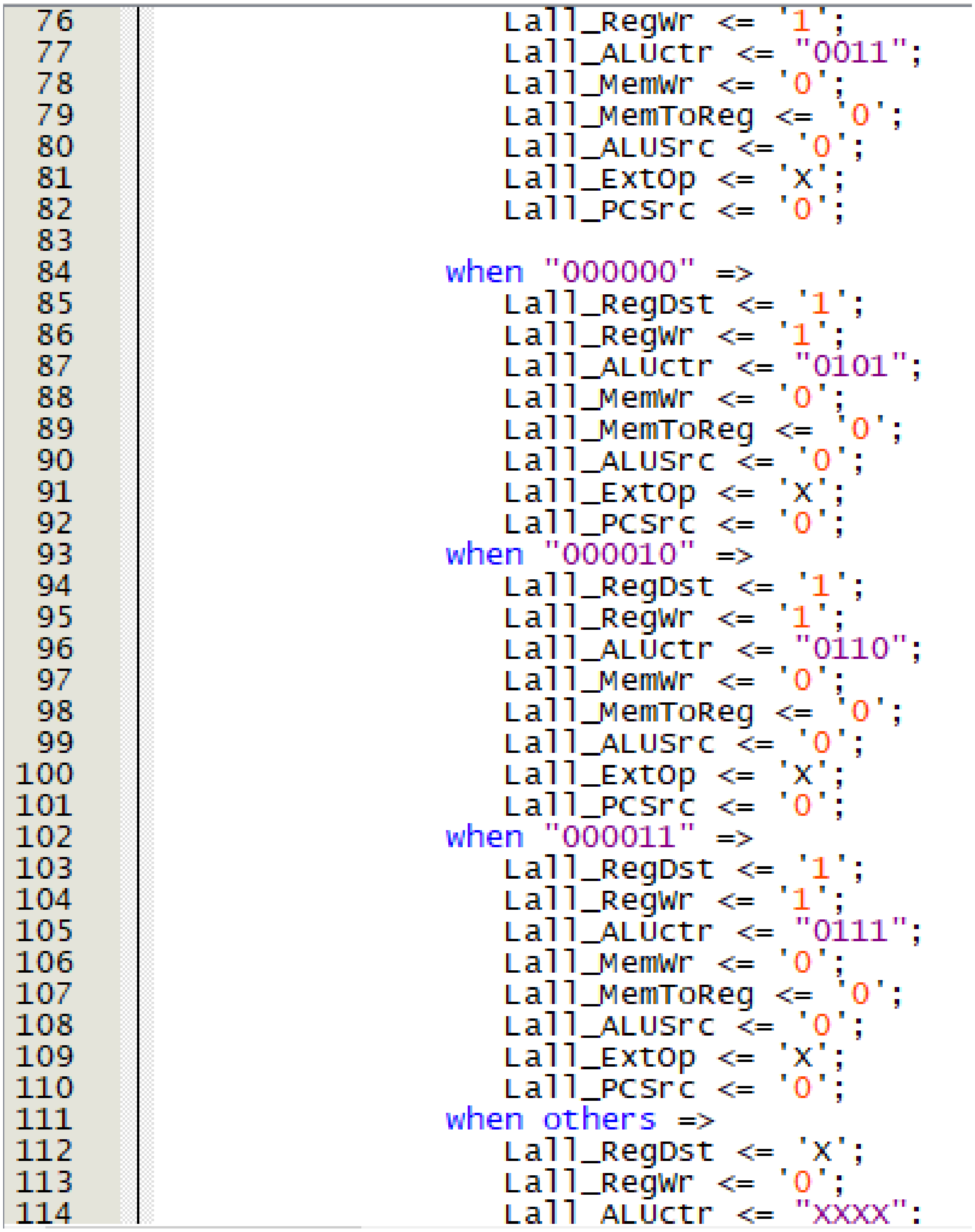
```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;

entity Lall_CPU_Controller is
    port (
            Lall_op: in std_logic_vector(5 downto 0);
            Lall_func: in std_logic_vector(5 downto 0);
            Lall_equal: in std_logic;
            Lall_RegDst: out std_logic;
            Lall_RegWr: out std_logic;
            Lall_ALUctr: out std_logic_vector(3 downto 0);
            Lall_MemWr: out std_logic;
            Lall_MemToReg: out std_logic;
            Lall_ALUSrc: out std_logic;
            Lall_ExtOp: out std_logic;
            Lall_PCSrc: out std_logic
            );
end Lall_CPU_Controller;

architecture behavioral_cpuc of Lall_CPU_Controller is
begin
    process(Lall_op, Lall_func, Lall_equal)
    begin
        case Lall_op is
            when "000000" =>
                case Lall_func is
                    when "100000" =>
                        Lall_RegDst <= '1';
                        Lall_RegWr <= '1';
                        Lall_ALUctr <= "0000";
                        Lall_MemWr <= '0';
                        Lall_MemToReg <= '0';
                        Lall_ALUSrc <= '0';
                        Lall_ExtOp <= 'X';
                        Lall_PCSrc <= '0';
                    when "100001" =>
```

Figure 14. VHDL code CPU controller pt 1



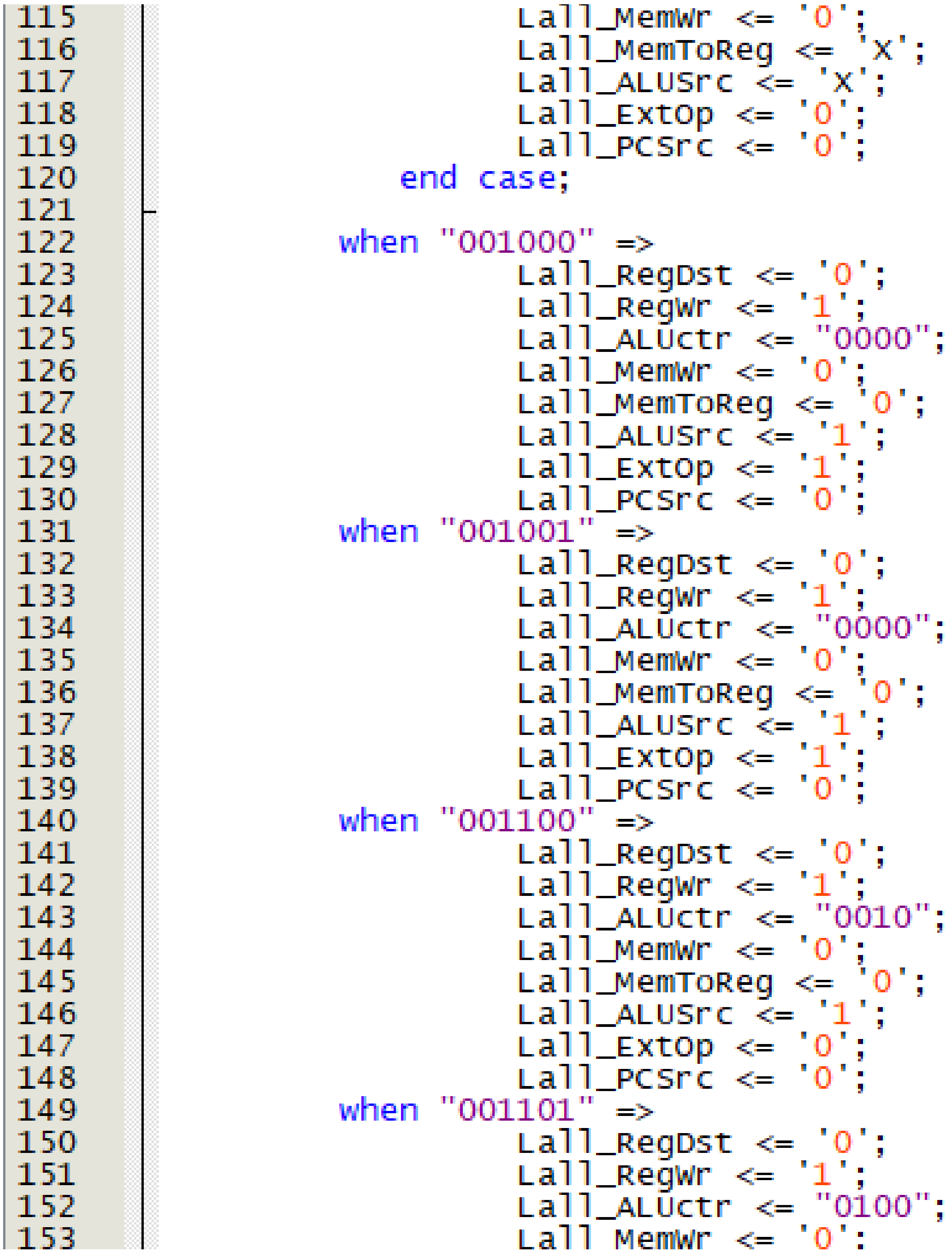
```
                Lall_RegDst <= '1';
                Lall_RegWr <= '1';
                Lall_ALUctr <= "0000";
                Lall_MemWr <= '0';
                Lall_MemToReg <= '0';
                Lall_ALUSrc <= '0';
                Lall_ExtOp <= 'X';
                Lall_PCSrc <= '0';
            when "100010" =>
                Lall_RegDst <= '1';
                Lall_RegWr <= '1';
                Lall_ALUctr <= "0001";
                Lall_MemWr <= '0';
                Lall_MemToReg <= '0';
                Lall_ALUSrc <= '0';
                Lall_ExtOp <= 'X';
                Lall_PCSrc <= '0';
            when "100011" =>
                Lall_RegDst <= '1';
                Lall_RegWr <= '1';
                Lall_ALUctr <= "0001";
                Lall_MemWr <= '0';
                Lall_MemToReg <= '0';
                Lall_ALUSrc <= '0';
                Lall_ExtOp <= 'X';
                Lall_PCSrc <= '0';
            when "100100" =>
                Lall_RegDst <= '1';
                Lall_RegWr <= '1';
                Lall_ALUctr <= "0010";
                Lall_MemWr <= '0';
                Lall_MemToReg <= '0';
                Lall_ALUSrc <= '0';
                Lall_ExtOp <= 'X';
                Lall_PCSrc <= '0';
            when "100111" =>
                Lall_RegDst <= '1';
```

Figure 15. VHDL code CPU controller pt 2



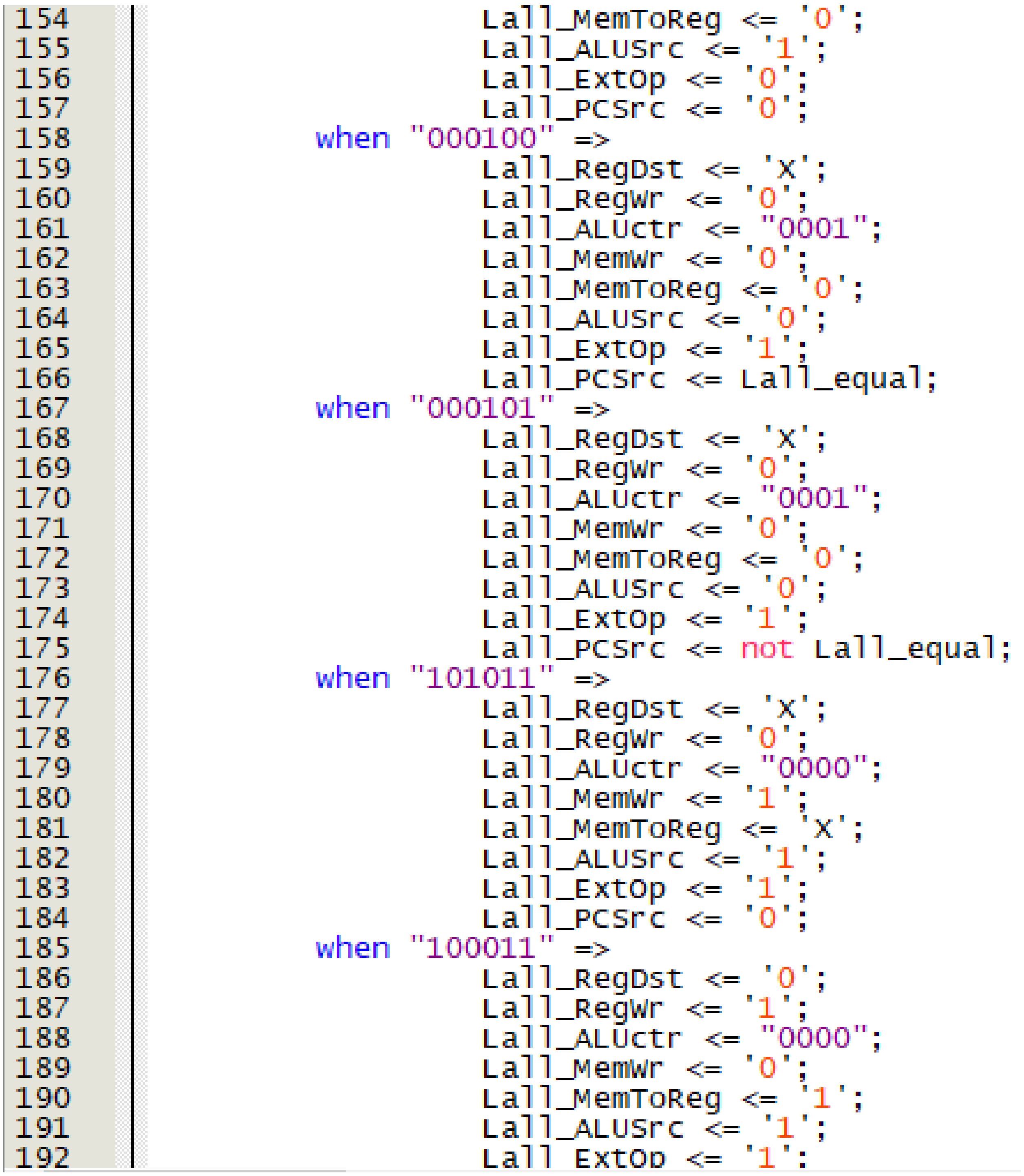
```
                        Lall_RegWr <= '1';
                        Lall_ALUctr <= "0011";
                        Lall_MemWr <= '0';
                        Lall_MemToReg <= '0';
                        Lall_ALUSrc <= '0';
                        Lall_ExtOp <= 'X';
                        Lall_PCSrc <= '0';

                    when "000000" =>
                        Lall_RegDst <= '1';
                        Lall_RegWr <= '1';
                        Lall_ALUctr <= "0101";
                        Lall_MemWr <= '0';
                        Lall_MemToReg <= '0';
                        Lall_ALUSrc <= '0';
                        Lall_ExtOp <= 'X';
                        Lall_PCSrc <= '0';
                    when "000010" =>
                        Lall_RegDst <= '1';
                        Lall_RegWr <= '1';
                        Lall_ALUctr <= "0110";
                        Lall_MemWr <= '0';
                        Lall_MemToReg <= '0';
                        Lall_ALUSrc <= '0';
                        Lall_ExtOp <= 'X';
                        Lall_PCSrc <= '0';
                    when "000011" =>
                        Lall_RegDst <= '1';
                        Lall_RegWr <= '1';
                        Lall_ALUctr <= "0111";
                        Lall_MemWr <= '0';
                        Lall_MemToReg <= '0';
                        Lall_ALUSrc <= '0';
                        Lall_ExtOp <= 'X';
                        Lall_PCSrc <= '0';
                    when others =>
                        Lall_RegDst <= 'X';
                        Lall_RegWr <= '0';
                        Lall_ALUctr <= "XXXX";
```

Figure 16. VHDL code CPU controller pt 3



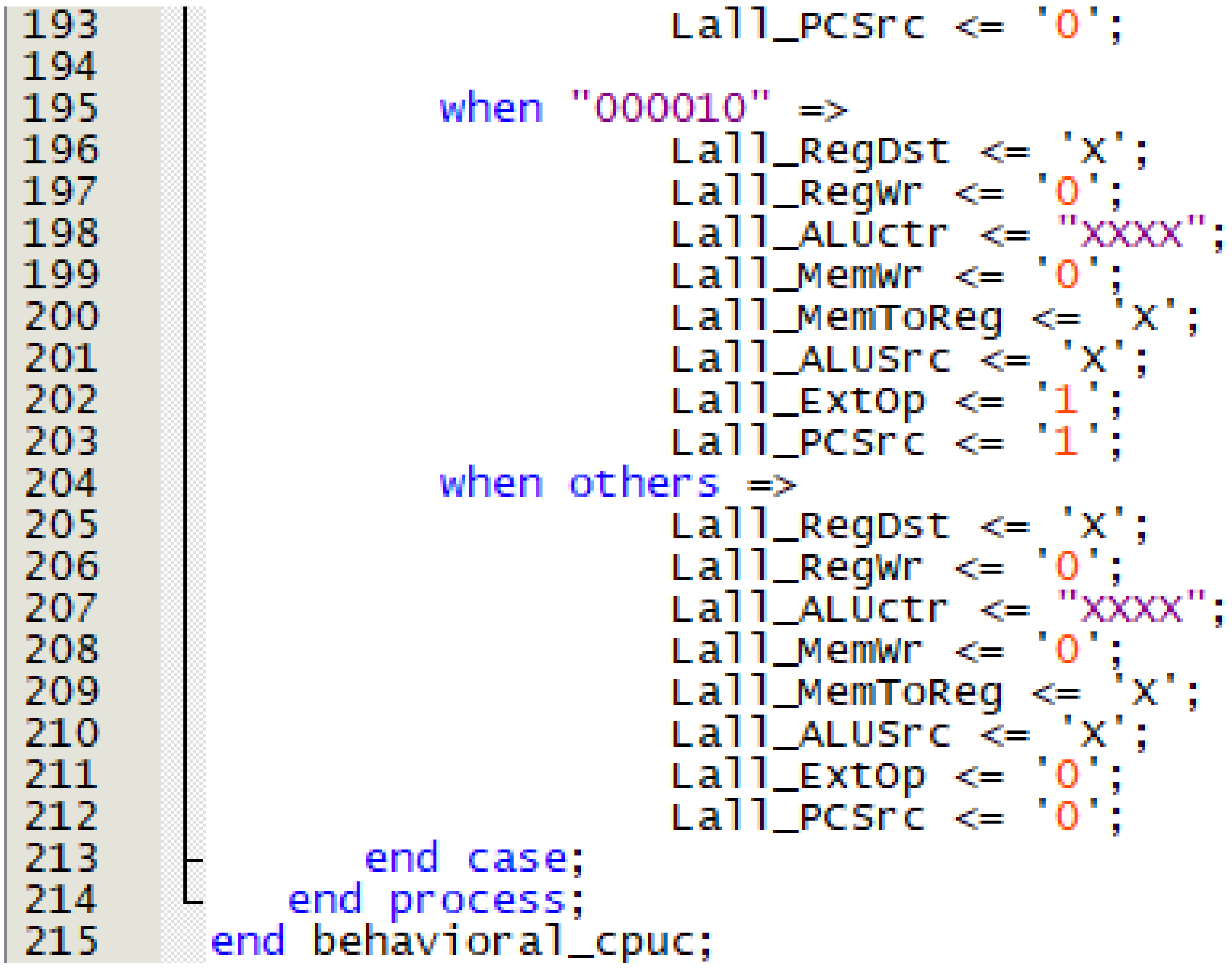
```
                Lall_MemWr <= '0';
                Lall_MemToReg <= 'X';
                Lall_ALUSrc <= 'X';
                Lall_ExtOp <= '0';
                Lall_PCSrc <= '0';
        end case;

    when "001000" =>
                Lall_RegDst <= '0';
                Lall_RegWr <= '1';
                Lall_ALUctr <= "0000";
                Lall_MemWr <= '0';
                Lall_MemToReg <= '0';
                Lall_ALUSrc <= '1';
                Lall_ExtOp <= '1';
                Lall_PCSrc <= '0';
    when "001001" =>
                Lall_RegDst <= '0';
                Lall_RegWr <= '1';
                Lall_ALUctr <= "0000";
                Lall_MemWr <= '0';
                Lall_MemToReg <= '0';
                Lall_ALUSrc <= '1';
                Lall_ExtOp <= '1';
                Lall_PCSrc <= '0';
    when "001100" =>
                Lall_RegDst <= '0';
                Lall_RegWr <= '1';
                Lall_ALUctr <= "0010";
                Lall_MemWr <= '0';
                Lall_MemToReg <= '0';
                Lall_ALUSrc <= '1';
                Lall_ExtOp <= '0';
                Lall_PCSrc <= '0';
    when "001101" =>
                Lall_RegDst <= '0';
                Lall_RegWr <= '1';
                Lall_ALUctr <= "0100";
                Lall_MemWr <= '0';
```

Figure 17. VHDL code CPU controller pt 4



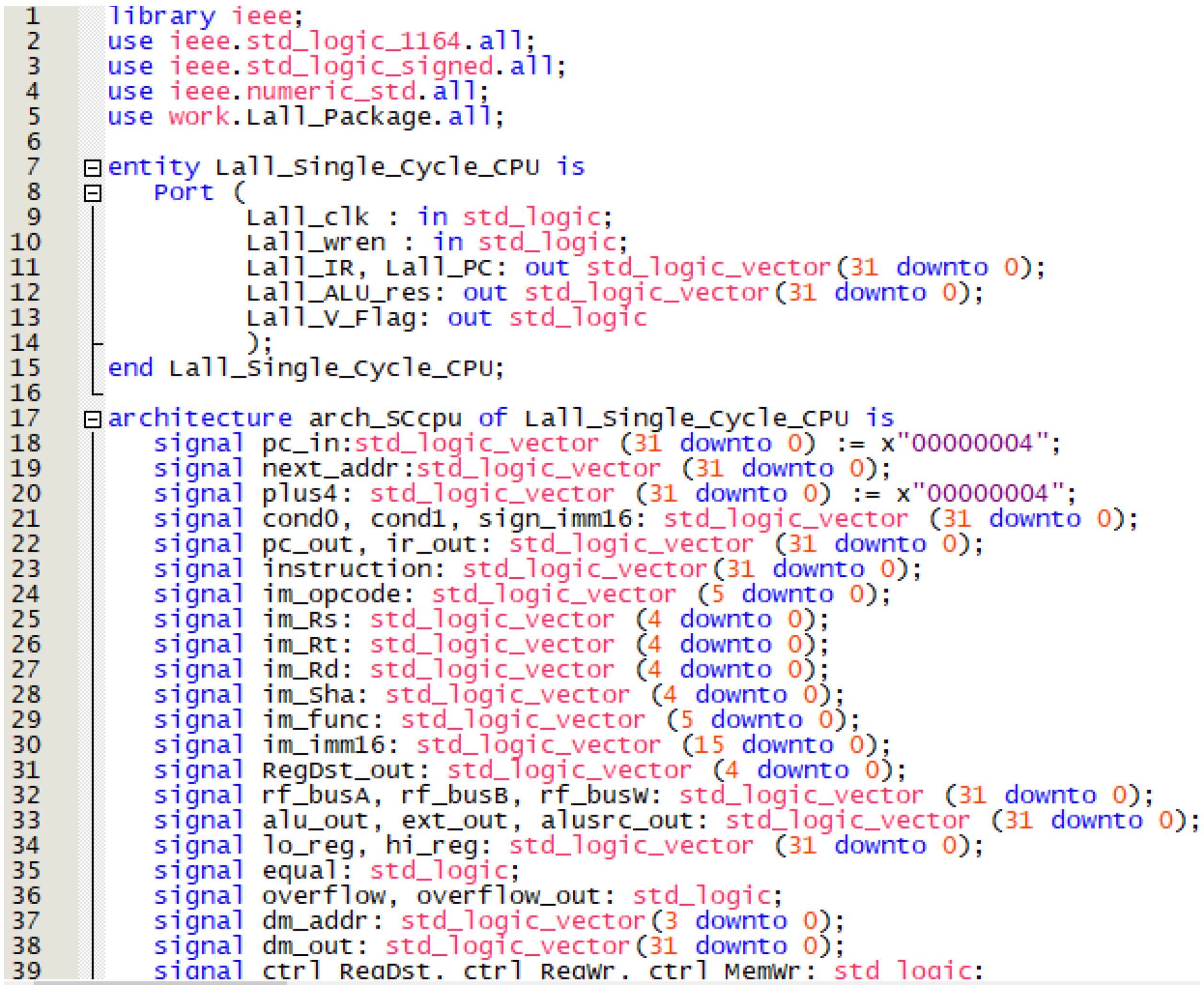
```
                        Lall_MemToReg <= '0';
                        Lall_ALUSrc <= '1';
                        Lall_ExtOp <= '0';
                        Lall_PCSrc <= '0';
                when "000100" =>
                        Lall_RegDst <= 'X';
                        Lall_RegWr <= '0';
                        Lall_ALUctr <= "0001";
                        Lall_MemWr <= '0';
                        Lall_MemToReg <= '0';
                        Lall_ALUSrc <= '0';
                        Lall_ExtOp <= '1';
                        Lall_PCSrc <= Lall_equal;
                when "000101" =>
                        Lall_RegDst <= 'X';
                        Lall_RegWr <= '0';
                        Lall_ALUctr <= "0001";
                        Lall_MemWr <= '0';
                        Lall_MemToReg <= '0';
                        Lall_ALUSrc <= '0';
                        Lall_ExtOp <= '1';
                        Lall_PCSrc <= not Lall_equal;
                when "101011" =>
                        Lall_RegDst <= 'X';
                        Lall_RegWr <= '0';
                        Lall_ALUctr <= "0000";
                        Lall_MemWr <= '1';
                        Lall_MemToReg <= 'X';
                        Lall_ALUSrc <= '1';
                        Lall_ExtOp <= '1';
                        Lall_PCSrc <= '0';
                when "100011" =>
                        Lall_RegDst <= '0';
                        Lall_RegWr <= '1';
                        Lall_ALUctr <= "0000";
                        Lall_MemWr <= '0';
                        Lall_MemToReg <= '1';
                        Lall_ALUSrc <= '1';
                        Lall_ExtOp <= '1';
```

Figure 18. VHDL code CPU controller pt 5



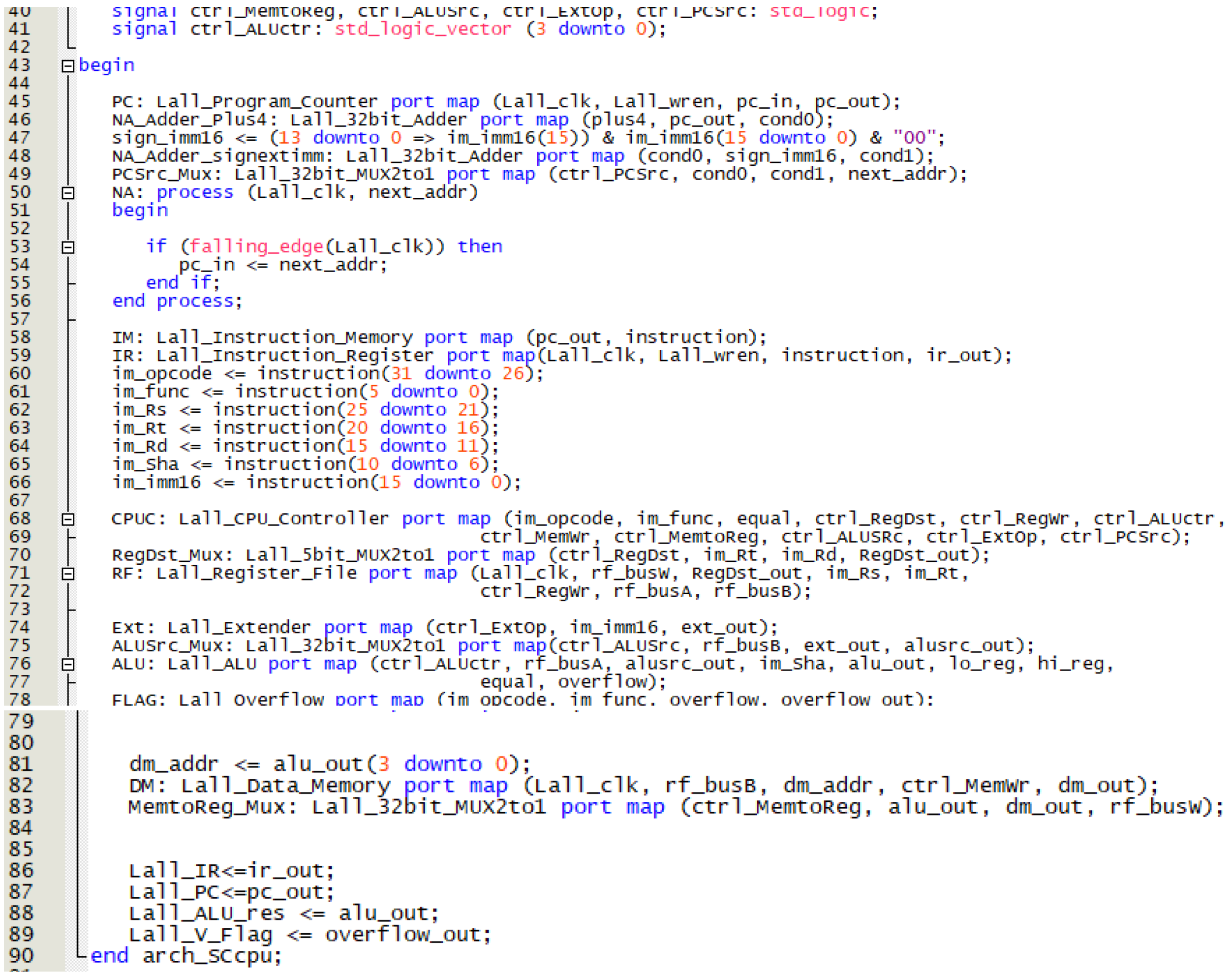
```
                        Lall_PCSrc <= '0';

                when "000010" =>
                        Lall_RegDst <= 'X';
                        Lall_RegWr <= '0';
                        Lall_ALUctr <= "XXXX";
                        Lall_MemWr <= '0';
                        Lall_MemToReg <= 'X';
                        Lall_ALUSrc <= 'X';
                        Lall_Extop <= '1';
                        Lall_PCSrc <= '1';
                when others =>
                        Lall_RegDst <= 'X';
                        Lall_RegWr <= '0';
                        Lall_ALUctr <= "XXXX";
                        Lall_MemWr <= '0';
                        Lall_MemToReg <= 'X';
                        Lall_ALUSrc <= 'X';
                        Lall_Extop <= '0';
                        Lall_PCSrc <= '0';
            end case;
        end process;
end behavioral_cpuc;
```

Figure 19. VHDL code CPU controller pt 6.



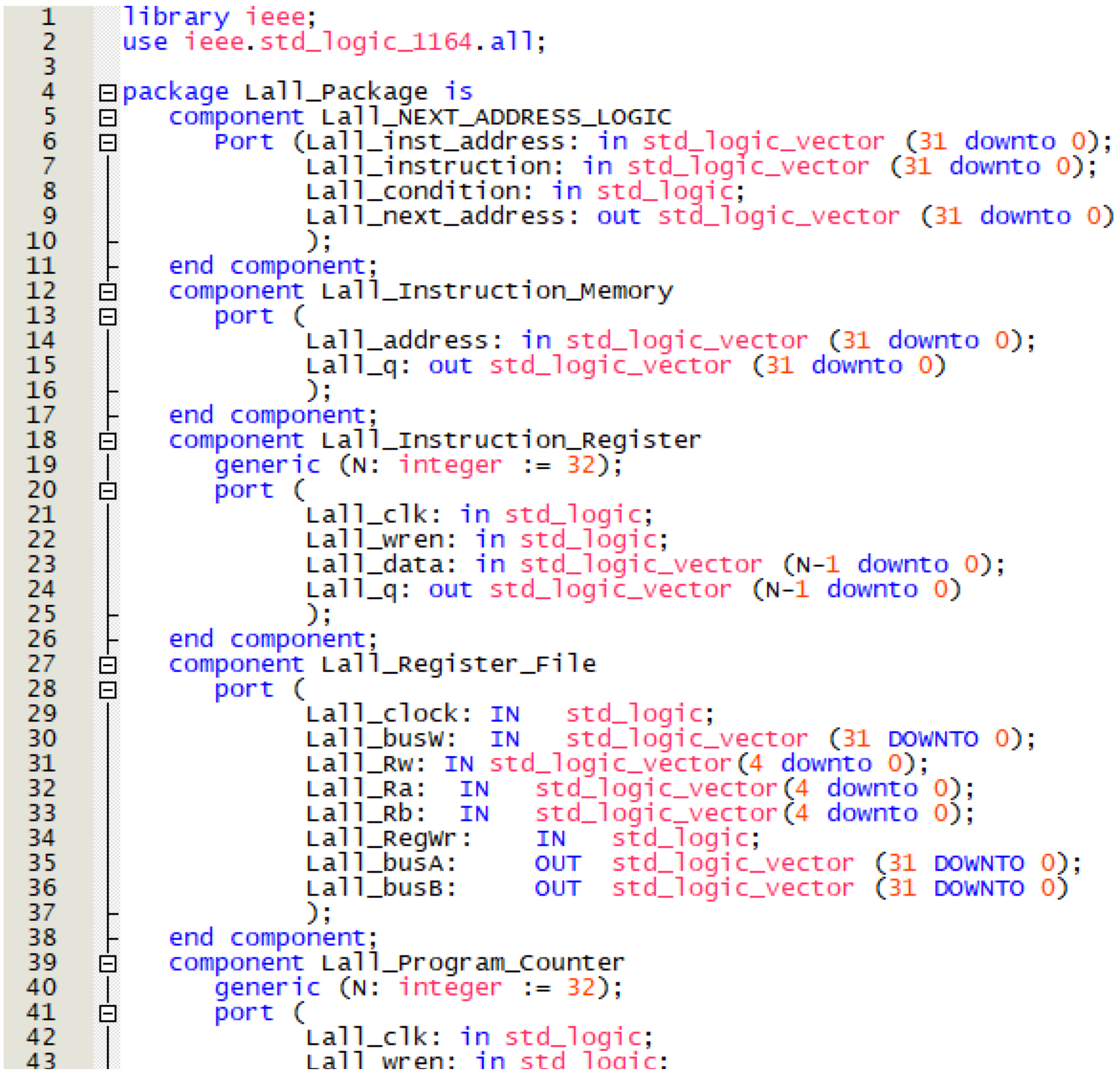
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;
use work.Lall_Package.all;

entity Lall_Single_Cycle_CPU is
   Port (
         Lall_clk : in std_logic;
         Lall_wren : in std_logic;
         Lall_IR, Lall_PC: out std_logic_vector(31 downto 0);
         Lall_ALU_res: out std_logic_vector(31 downto 0);
         Lall_V_Flag: out std_logic
         );
end Lall_Single_Cycle_CPU;

architecture arch_SCcpu of Lall_Single_Cycle_CPU is
   signal pc_in:std_logic_vector (31 downto 0) := x"00000004";
   signal next_addr:std_logic_vector (31 downto 0);
   signal plus4: std_logic_vector (31 downto 0) := x"00000004";
   signal cond0, cond1, sign_imm16: std_logic_vector (31 downto 0);
   signal pc_out, ir_out: std_logic_vector (31 downto 0);
   signal instruction: std_logic_vector(31 downto 0);
   signal im_opcode: std_logic_vector (5 downto 0);
   signal im_Rs: std_logic_vector (4 downto 0);
   signal im_Rt: std_logic_vector (4 downto 0);
   signal im_Rd: std_logic_vector (4 downto 0);
   signal im_Sha: std_logic_vector (4 downto 0);
   signal im_func: std_logic_vector (5 downto 0);
   signal im_imm16: std_logic_vector (15 downto 0);
   signal RegDst_out: std_logic_vector (4 downto 0);
   signal rf_busA, rf_busB, rf_busW: std_logic_vector (31 downto 0);
   signal alu_out, ext_out, alusrc_out: std_logic_vector (31 downto 0);
   signal lo_reg, hi_reg: std_logic_vector (31 downto 0);
   signal equal: std_logic;
   signal overflow, overflow_out: std_logic;
   signal dm_addr: std_logic_vector(3 downto 0);
   signal dm_out: std_logic_vector(31 downto 0);
   signal ctrl_RegDst, ctrl_RegWr, ctrl_MemWr: std_logic;
```

Figure 20. VHDL Code Single Cycle CPU pt1.



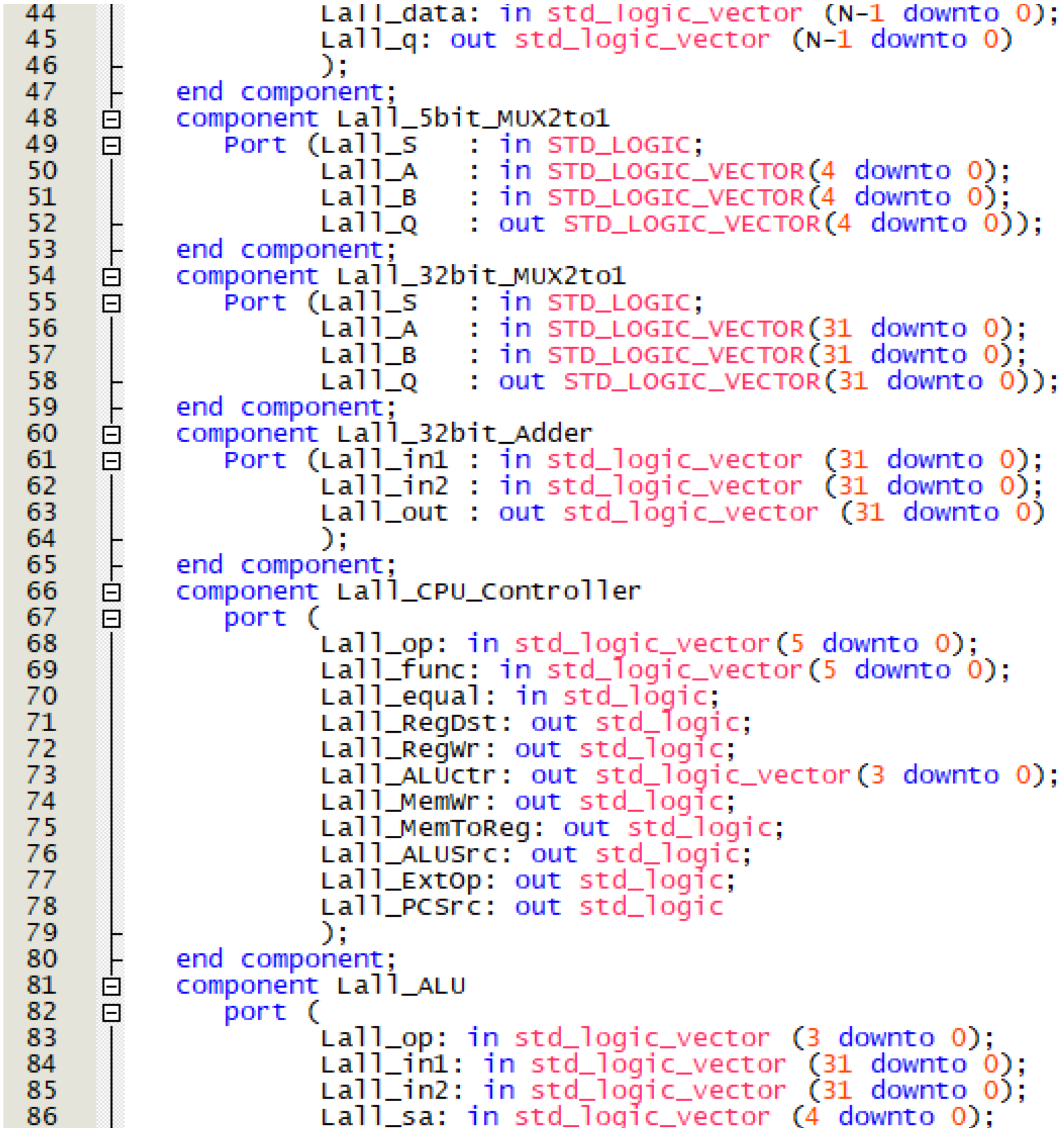
```vhdl
   signal ctrl_MemtoReg, ctrl_ALUSrc, ctrl_ExtOp, ctrl_PCSrc: std_logic;
   signal ctrl_ALUctr: std_logic_vector (3 downto 0);

begin

   PC: Lall_Program_Counter port map (Lall_clk, Lall_wren, pc_in, pc_out);
   NA_Adder_Plus4: Lall_32bit_Adder port map (plus4, pc_out, cond0);
   sign_imm16 <= (13 downto 0 => im_imm16(15)) & im_imm16(15 downto 0) & "00";
   NA_Adder_signextimm: Lall_32bit_Adder port map (cond0, sign_imm16, cond1);
   PCSrc_Mux: Lall_32bit_MUX2to1 port map (ctrl_PCSrc, cond0, cond1, next_addr);
   NA: process (Lall_clk, next_addr)
   begin

      if (falling_edge(Lall_clk)) then
         pc_in <= next_addr;
      end if;
   end process;

   IM: Lall_Instruction_Memory port map (pc_out, instruction);
   IR: Lall_Instruction_Register port map(Lall_clk, Lall_wren, instruction, ir_out);
   im_opcode <= instruction(31 downto 26);
   im_func <= instruction(5 downto 0);
   im_Rs <= instruction(25 downto 21);
   im_Rt <= instruction(20 downto 16);
   im_Rd <= instruction(15 downto 11);
   im_Sha <= instruction(10 downto 6);
   im_imm16 <= instruction(15 downto 0);

   CPUC: Lall_CPU_Controller port map (im_opcode, im_func, equal, ctrl_RegDst, ctrl_RegWr, ctrl_ALUctr,
                                    ctrl_MemWr, ctrl_MemtoReg, ctrl_ALUSrc, ctrl_ExtOp, ctrl_PCSrc);
   RegDst_Mux: Lall_5bit_MUX2to1 port map (ctrl_RegDst, im_Rt, im_Rd, RegDst_out);
   RF: Lall_Register_File port map (Lall_clk, rf_busW, RegDst_out, im_Rs, im_Rt,
                                    ctrl_RegWr, rf_busA, rf_busB);

   Ext: Lall_Extender port map (ctrl_ExtOp, im_imm16, ext_out);
   ALUSrc_Mux: Lall_32bit_MUX2to1 port map(ctrl_ALUSrc, rf_busB, ext_out, alusrc_out);
   ALU: Lall_ALU port map (ctrl_ALUctr, rf_busA, alusrc_out, im_Sha, alu_out, lo_reg, hi_reg,
                                    equal, overflow);
   FLAG: Lall_Overflow port map (im_opcode, im_func, overflow, overflow_out);


   dm_addr <= alu_out(3 downto 0);
   DM: Lall_Data_Memory port map (Lall_clk, rf_busB, dm_addr, ctrl_MemWr, dm_out);
   MemtoReg_Mux: Lall_32bit_MUX2to1 port map (ctrl_MemtoReg, alu_out, dm_out, rf_busW);


   Lall_IR<=ir_out;
   Lall_PC<=pc_out;
   Lall_ALU_res <= alu_out;
   Lall_V_Flag <= overflow_out;
end arch_SCcpu;
```

Figure 21. VHDL Code Single Cycle CPU pt2.



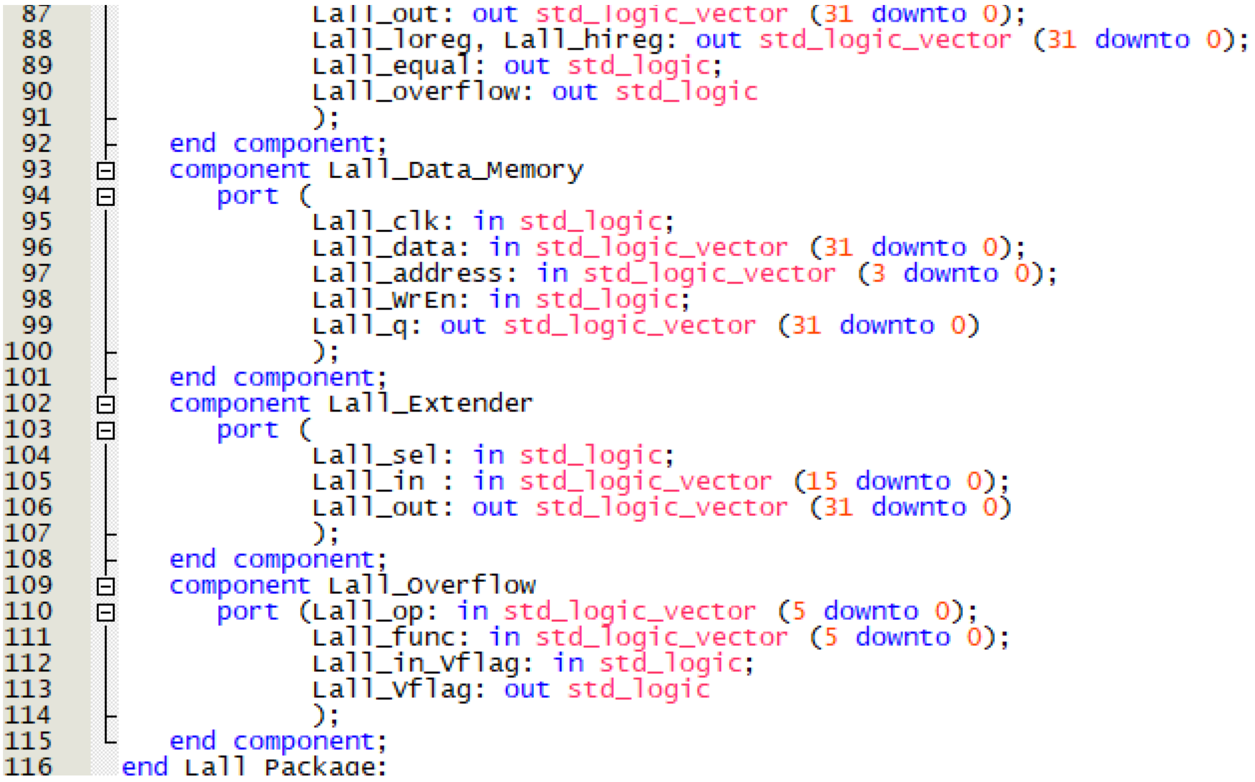
```
library ieee;
use ieee.std_logic_1164.all;

package Lall_Package is
   component Lall_NEXT_ADDRESS_LOGIC
      Port (Lall_inst_address: in std_logic_vector (31 downto 0);
            Lall_instruction: in std_logic_vector (31 downto 0);
            Lall_condition: in std_logic;
            Lall_next_address: out std_logic_vector (31 downto 0)
            );
   end component;
   component Lall_Instruction_Memory
      port (
            Lall_address: in std_logic_vector (31 downto 0);
            Lall_q: out std_logic_vector (31 downto 0)
            );
   end component;
   component Lall_Instruction_Register
      generic (N: integer := 32);
      port (
            Lall_clk: in std_logic;
            Lall_wren: in std_logic;
            Lall_data: in std_logic_vector (N-1 downto 0);
            Lall_q: out std_logic_vector (N-1 downto 0)
            );
   end component;
   component Lall_Register_File
      port (
            Lall_clock: IN   std_logic;
            Lall_busW:  IN   std_logic_vector (31 DOWNTO 0);
            Lall_Rw: IN std_logic_vector(4 downto 0);
            Lall_Ra:  IN   std_logic_vector(4 downto 0);
            Lall_Rb:  IN   std_logic_vector(4 downto 0);
            Lall_RegWr:    IN   std_logic;
            Lall_busA:     OUT  std_logic_vector (31 DOWNTO 0);
            Lall_busB:     OUT  std_logic_vector (31 DOWNTO 0)
            );
   end component;
   component Lall_Program_Counter
      generic (N: integer := 32);
      port (
            Lall_clk: in std_logic;
            Lall_wren: in std_logic;
```

Figure 22. VHDL code package pt1



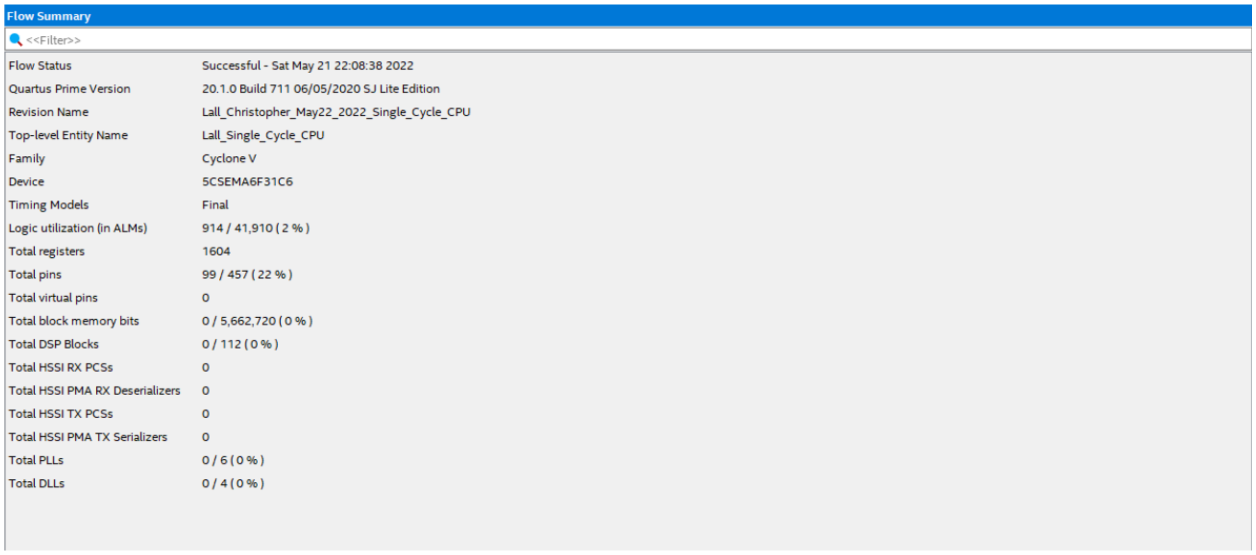
```
            Lall_data: in std_logic_vector (N-1 downto 0);
            Lall_q: out std_logic_vector (N-1 downto 0)
            );
    end component;
    component Lall_5bit_MUX2to1
        Port (Lall_S   : in STD_LOGIC;
              Lall_A   : in STD_LOGIC_VECTOR(4 downto 0);
              Lall_B   : in STD_LOGIC_VECTOR(4 downto 0);
              Lall_Q   : out STD_LOGIC_VECTOR(4 downto 0));
    end component;
    component Lall_32bit_MUX2to1
        Port (Lall_S   : in STD_LOGIC;
              Lall_A   : in STD_LOGIC_VECTOR(31 downto 0);
              Lall_B   : in STD_LOGIC_VECTOR(31 downto 0);
              Lall_Q   : out STD_LOGIC_VECTOR(31 downto 0));
    end component;
    component Lall_32bit_Adder
        Port (Lall_in1 : in std_logic_vector (31 downto 0);
              Lall_in2 : in std_logic_vector (31 downto 0);
              Lall_out : out std_logic_vector (31 downto 0)
              );
    end component;
    component Lall_CPU_Controller
        port (
              Lall_op: in std_logic_vector(5 downto 0);
              Lall_func: in std_logic_vector(5 downto 0);
              Lall_equal: in std_logic;
              Lall_RegDst: out std_logic;
              Lall_RegWr: out std_logic;
              Lall_ALUctr: out std_logic_vector(3 downto 0);
              Lall_MemWr: out std_logic;
              Lall_MemToReg: out std_logic;
              Lall_ALUSrc: out std_logic;
              Lall_ExtOp: out std_logic;
              Lall_PCSrc: out std_logic
              );
    end component;
    component Lall_ALU
        port (
              Lall_op: in std_logic_vector (3 downto 0);
              Lall_in1: in std_logic_vector (31 downto 0);
              Lall_in2: in std_logic_vector (31 downto 0);
              Lall_sa: in std_logic_vector (4 downto 0);
```

Figure 23. VHDL code package pt2



```
            Lall_out: out std_logic_vector (31 downto 0);
            Lall_loreg, Lall_hireg: out std_logic_vector (31 downto 0);
            Lall_equal: out std_logic;
            Lall_overflow: out std_logic
            );
    end component;
    component Lall_Data_Memory
       port (
            Lall_clk: in std_logic;
            Lall_data: in std_logic_vector (31 downto 0);
            Lall_address: in std_logic_vector (3 downto 0);
            Lall_WrEn: in std_logic;
            Lall_q: out std_logic_vector (31 downto 0)
            );
    end component;
    component Lall_Extender
       port (
            Lall_sel: in std_logic;
            Lall_in : in std_logic_vector (15 downto 0);
            Lall_out: out std_logic_vector (31 downto 0)
            );
    end component;
    component Lall_Overflow
       port (Lall_op: in std_logic_vector (5 downto 0);
            Lall_func: in std_logic_vector (5 downto 0);
            Lall_in_Vflag: in std_logic;
            Lall_Vflag: out std_logic
            );
    end component;
end Lall_Package;
```

Figure 24. VHDL code package pt3



Flow Summary

Flow Status	Successful - Sat May 21 22:08:38 2022
Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Lite Edition
Revision Name	Lall_Christopher_May22_2022_Single_Cycle_CPU
Top-level Entity Name	Lall_Single_Cycle_CPU
Family	Cyclone V
Device	5CSEMA6F31C6
Timing Models	Final
Logic utilization (in ALMs)	914 / 41,910 (2 %)
Total registers	1604
Total pins	99 / 457 (22 %)
Total virtual pins	0
Total block memory bits	0 / 5,662,720 (0 %)
Total DSP Blocks	0 / 112 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 25. Successful compilation report.

Block Diagram & Simulation:

In this section I will show all simulation results.

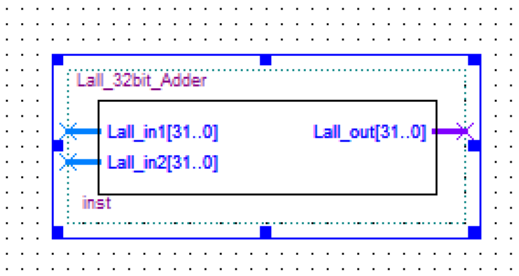
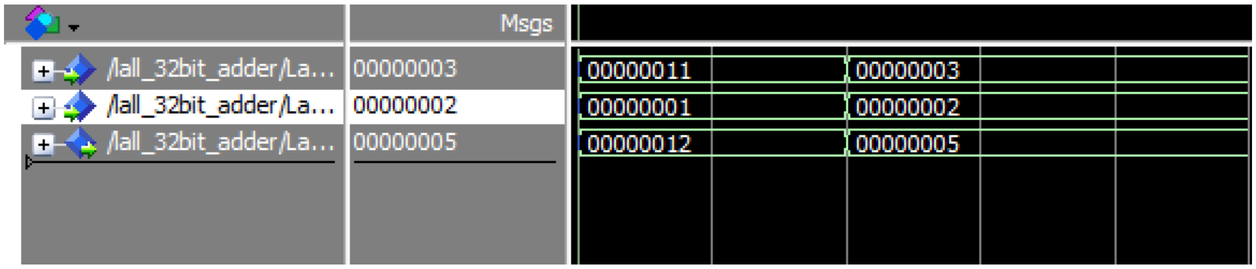


Figure 26. Block diagram for adder

Adder:



```
/lall_32bit_adder/La...   00000003   00000011   00000003
/lall_32bit_adder/La...   00000002   00000001   00000002
/lall_32bit_adder/La...   00000005   00000012   00000005
```

Figure 27. Adder simulation.

This is the adder simulation. As you can see in the simulation, adding 11 and 1 give us 12. In terms of two 32 bit inputs, we get 0x00000011 + 0x00000001 = 0x00000012. Our second example is 0x00000003 + 0x00000002 = 0x00000005. Address Logic and Program counter uses adder as a component.

2:1 Mux:

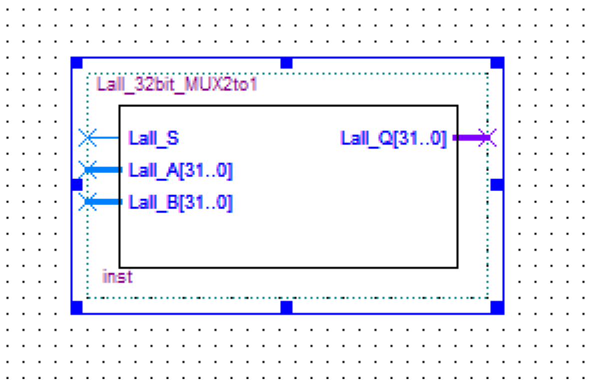
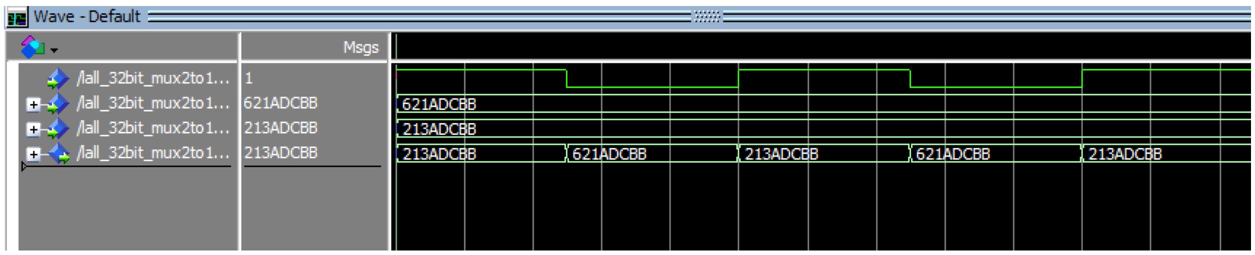


Figure 28. 32 bit 2:1 mux bsf.



```
/lall_32bit_mux2to1...   1
/lall_32bit_mux2to1...   621ADCBB   621ADCBB
/lall_32bit_mux2to1...   213ADCBB   213ADCBB
/lall_32bit_mux2to1...   213ADCBB   213ADCBB   621ADCBB   213ADCBB   621ADCBB   213ADCBB
```

Figure 29. 32bit mux in modelsim simulation.

Above is bsf and simulation for 2:1 mux. This will help the data being selected during instruction execution time. The mux will help guarantee data chose is appropriate for the instructions execution. The value switches when the selector is changing from 1 to 0 or vise versa.

Program counter:

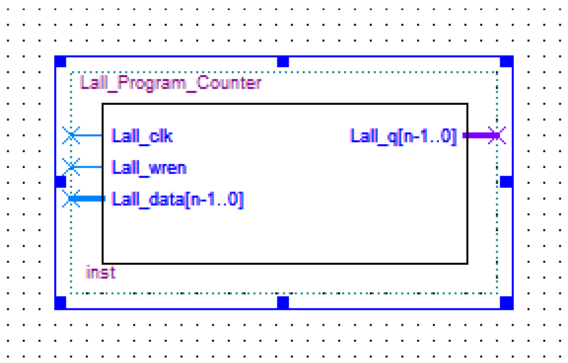
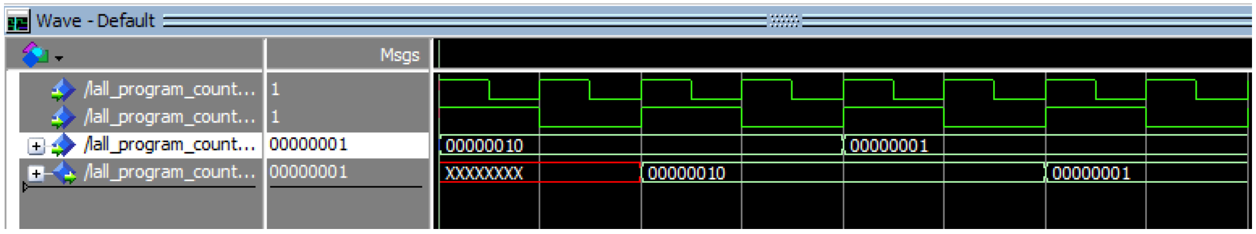


Figure 30. program counter bsf



```
/lall_program_count...   1
/lall_program_count...   1
/lall_program_count...   00000001    00000010                00000001
/lall_program_count...   00000001    XXXXXXXX    00000010                00000001
```

Figure 31. program counter simulation.

This will update the value of program counter which leads to the next instruction. Program counter value will output during rising edge of clock. Output is then sent to instruction memory and adder before returning to program counter as an input.

5bit Mux:

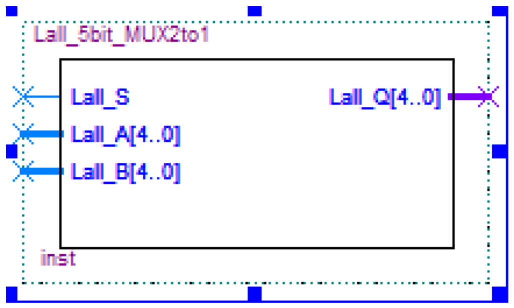


Figure 32. Block diagram for 5bit Mux

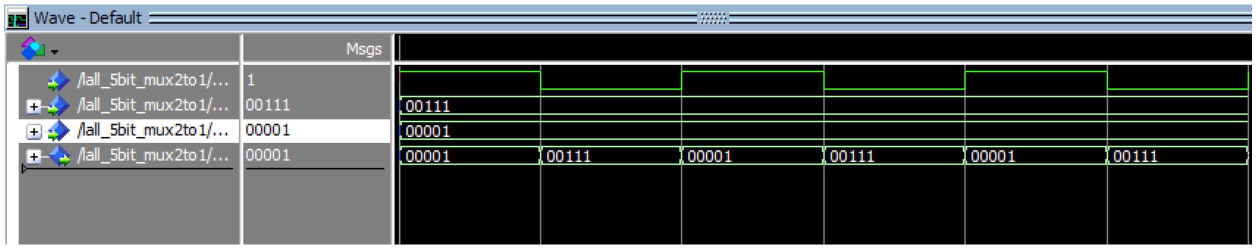

Wave - Default
Msgs
/lall_5bit_mux2to1/... 1
/lall_5bit_mux2to1/... 00111
/lall_5bit_mux2to1/... 00001
/lall_5bit_mux2to1/... 00001
00111
00001
00001
00111
00001
00111
00001
00111

Figure 33. 5bit Mux simulation.

In these two above figures, we see the block diagram and the simulation. Our input in this case is 5-bit. I put selector set to clock so that we can see the output change. If Selector is 1, signal A is displayed and if selector is 0, signal b is displayed. Register destination inside of CPU control is handled by this component.

Next Address Logic:

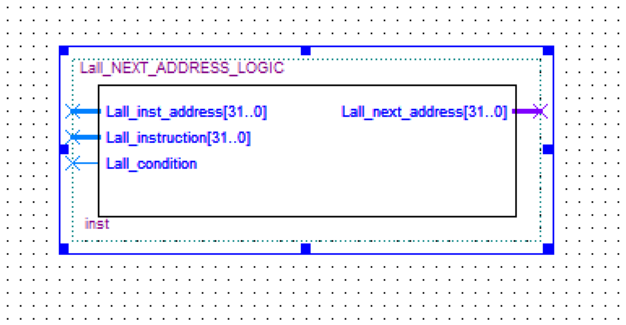


Figure 34. next address bsf.

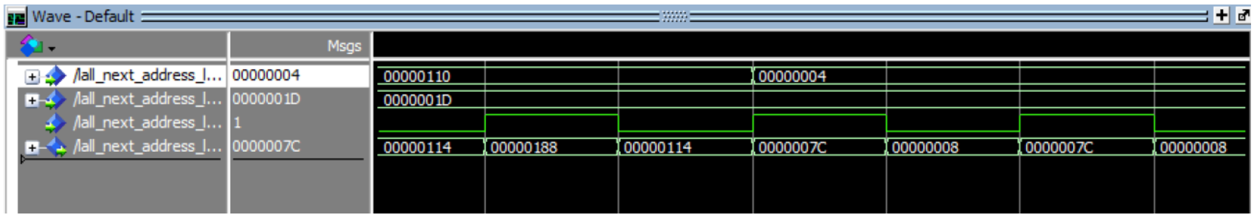

Wave - Default
Msgs
/lall_next_address_l... 00000004
/lall_next_address_l... 0000001D
/lall_next_address_l... 1
/lall_next_address_l... 0000007C
00000110
00000004
0000001D
00000114
00000188
00000114
0000007C
00000008
0000007C
00000008

Figure 35. Next address simulation

If condition is 0, Program counter + 4. If condition is 1, PC +4 + Extended Immediate "00"

Instruction Register:

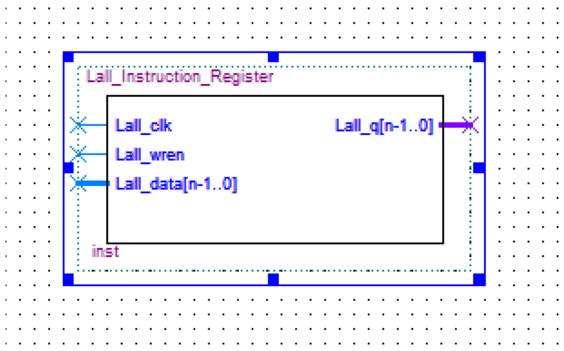
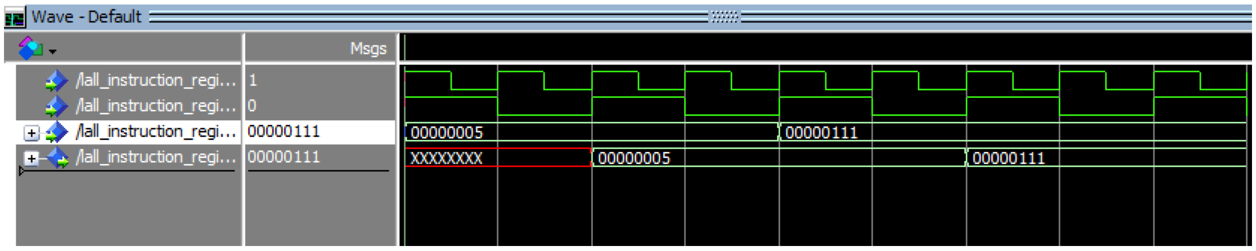


Figure 36. Instruction register bsf.



	Msgs	
/lall_instruction_regi...	1	
/lall_instruction_regi...	0	
/lall_instruction_regi...	00000111	00000005, 00000111
/lall_instruction_regi...	00000111	XXXXXXXX, 00000005, 00000111

Figure 37. instruction register simulation.

This is our instruction register. This will take instructions and decode it for ALU and assign RS,RT, and RD.

Instruction Memory:

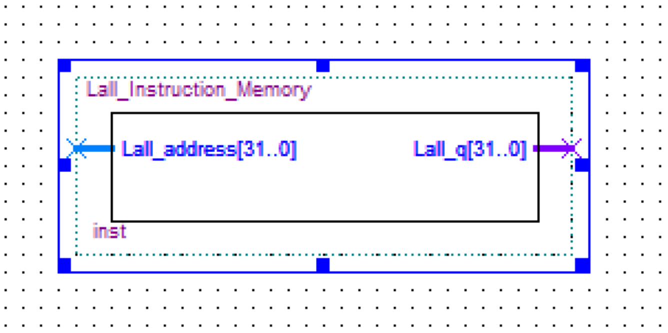
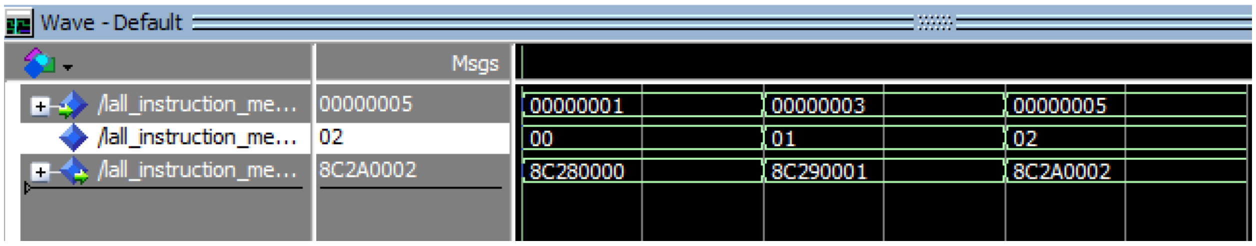


Figure 38. instruction memory bsf.



	Msgs			
/lall_instruction_me...	00000005	00000001	00000003	00000005
/lall_instruction_me...	02	00	01	02
/lall_instruction_me...	8C2A0002	8C280000	8C290001	8C2A0002

Figure 39. instruction memory simulation.

This is the Instruction Memory's file. This is where we put the MARS simulator code for the dot product. As a result, the values displayed here correspond to the MIPS assembly code for computing the dot product, which is included at the end of this file. Because the Instruction Memory can't go higher than 32 bits, I'll comment this part out until I've added more operations for testing. Our typical ADD, SUB, and MUL operations are commented out in these parts.

Register File:

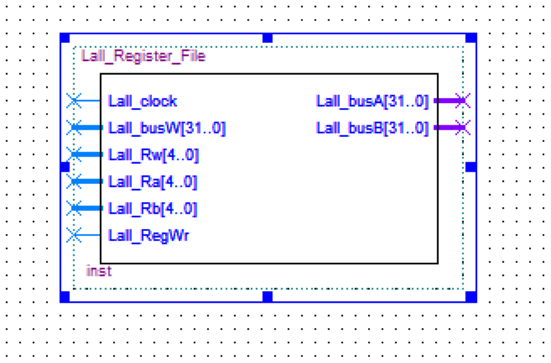


Figure 40. register file bsf.

The register file is a 3-Port RAM file. With a total capacity of 32 32-bit data, it has one input port and two output ports. The 5-bit addresses Ra,Rb are used to read the Register values using busA and busB, whereas Rw is used to write into the Register using busW. This type of RAM can read and write two data sets at the same time. The ALU and memory to register writing will be done on the buses. In this situation, all registers are set to 0.

Data Memory:

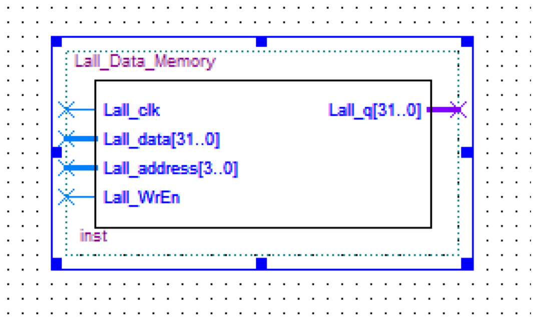


Figure 41. Data memory bsf

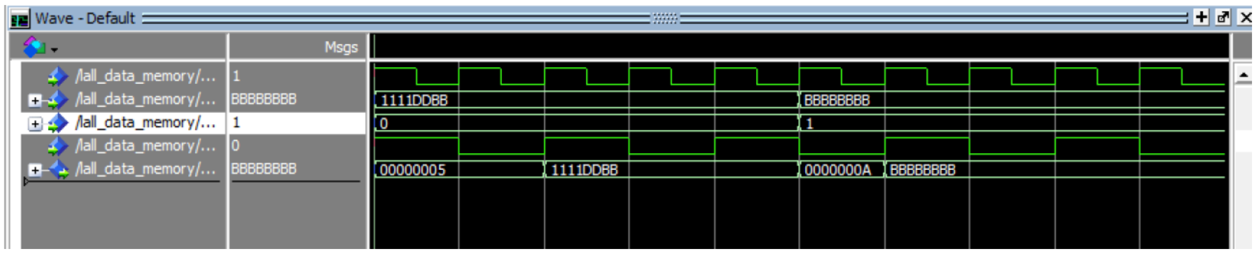


Figure 42. Data memory simulation

As demonstrated in the waveform above, the address is the same for both read and write access. We assert the write enable when we want to write to a specific address in Data Memory. The memory reads and prints the value based on the 4-bit address specified. The array is used to construct the data memory instead of the LPM Module.

ALU:

The ALU unit has 10 operations. The table below will show the opcode for the operations that will be used by the CPU Controller.

ALU OPERATIONS	
OPERATION	CODE
ADD	0000
SUB	0001
AND	0010
NOR	0011
OR	0100
SLL	0101
SRL	0110
SRA	0111

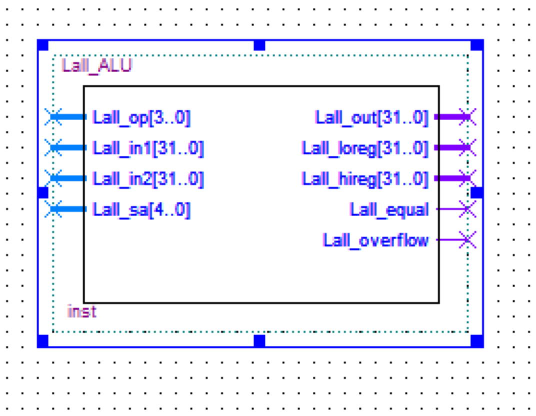
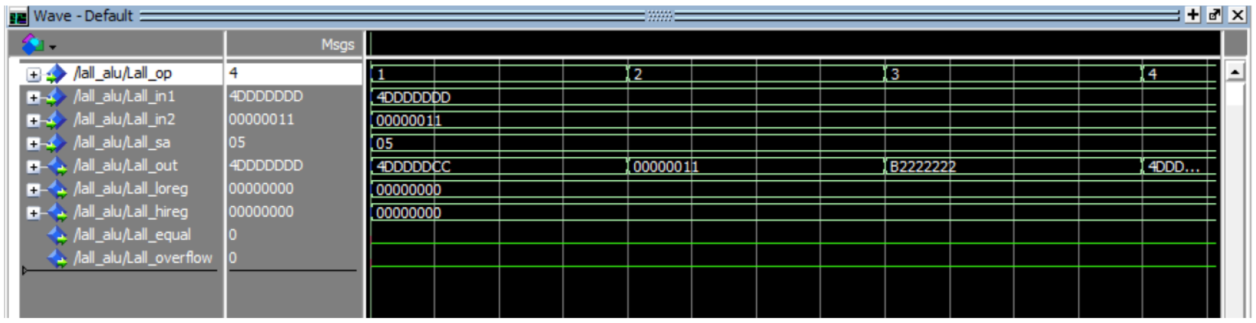


Figure 43. ALU bsf.



```
/lall_alu/Lall_op        4          1         2          3          4
/lall_alu/Lall_in1       4DDDDDDD   4DDDDDDD
/lall_alu/Lall_in2       00000011   00000011
/lall_alu/Lall_sa        05         05
/lall_alu/Lall_out       4DDDDDDD   4DDDDDCC  00000011   B2222222   4DDD...
/lall_alu/Lall_loreg     00000000   00000000
/lall_alu/Lall_hireg     00000000   00000000
/lall_alu/Lall_equal     0
/lall_alu/Lall_overflow  0
```

Figure 44. alu simulation.

The figures above show the alu unit operations. We don't obtain overflow because I did not use values that give us an overflow. My ALU unit calculates the OVERFLOW for ADD and SUB instructions, including the unsigned operation. To distinguish between unsigned and signed, an Overflow Unit is built to output the resulting overflow, which is reliant on the opcode and function from the instruction.

Sign Extender:

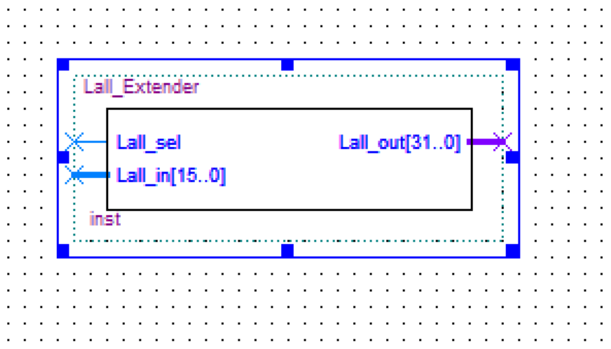
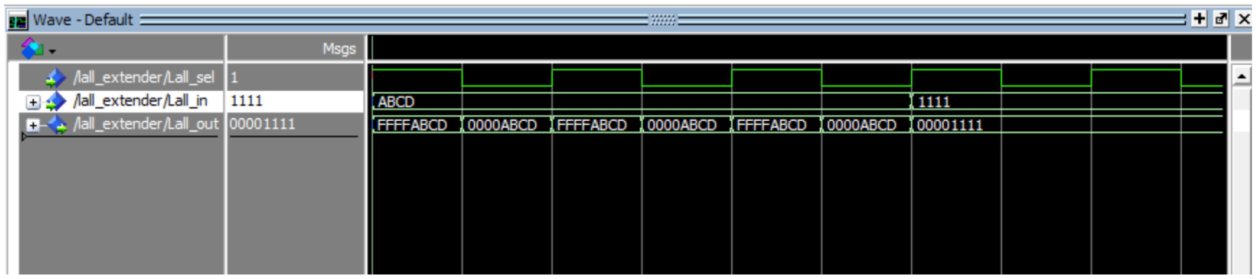


Figure 45. sign extender bsf



```
/lall_extender/Lall_sel   1
/lall_extender/Lall_in    1111       ABCD                                                                   1111
/lall_extender/Lall_out   00001111   FFFFABCD 0000ABCD FFFFABCD 0000ABCD FFFFABCD 0000ABCD 00001111
```

Figure 46. Sign extender simulation.

Sign extender allows us to stretch a 16bit input to a 32-bit output. The selector determines how this works. The CPU controller will decide if the number must be extended to zero or extended in sign direction.

Overflow:

There is no overflow bsf or simulation since it must be used injunction with other components. However, this file will check for overflow when doing addition.

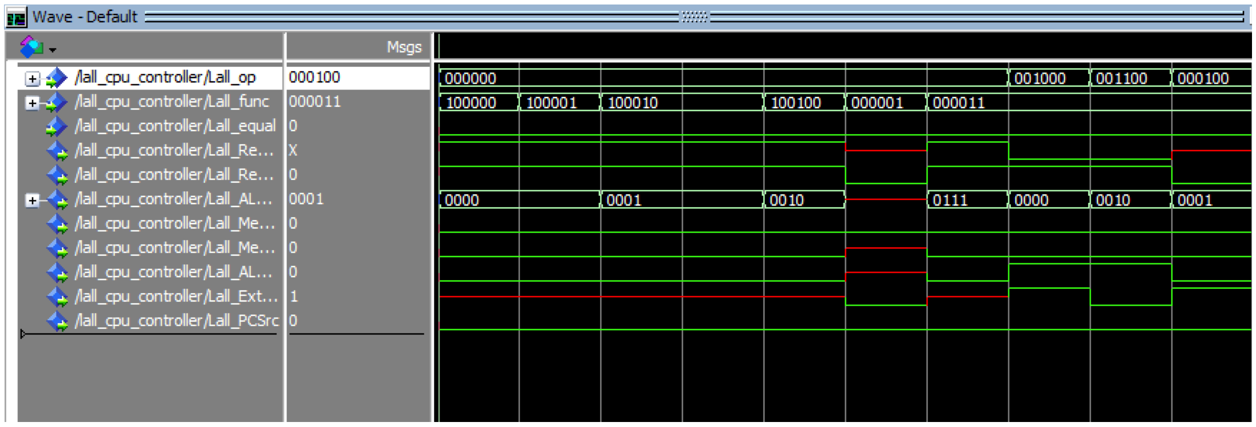
CPU Controller:

The CPU Controller is the brain of a single-cycle CPU. It determined the proper data path for the instructions being run. Multiplexers, Extenders, ALU Controller, Memory, and Register writes are all overseen by this.

	CPU CONTROLLER TABLE							
	RegDst	RegWr	ALUctr	MemWr	MemToReg	ALUSrc	ExtOp	PCSrc
ADD	1	1	0000	0	0	0	X	0
ADDI	0	1	0000	0	0	1	1	0
ADDIU	0	1	0000	0	0	1	1	0
ADDU	1	1	0000	0	0	0	X	0
SUB	1	1	0001	0	0	0	X	0
SUBU	1	1	0001	0	0	0	X	0
AND	1	1	0010	0	0	0	X	0
ANDI	0	1	0010	0	0	1	0	0
NOR	1	1	0011	0	0	0	X	0
ORI	0	1	0100	0	0	1	0	0
SLL	1	1	0101	0	0	0	X	0
SRL	1	1	0110	0	0	0	X	0
SRA	1	1	0111	0	0	0	X	0
SW	X	0	0000	1	X	1	1	0
LW	0	1	0000	0	1	1	1	0
BEQ	X	0	0001	0	X	0	1	EQUAL
BNE	X	0	0001	0	X	0	1	~EQUAL
J	X	0	XXXX	0	X	X	1	1

In the table above, the values required for the Register Write, Memory Write, PCSrc, ALUSrc, Memory To Register, Register Destination, and Extension operations of the 18 MIPS instruction are listed. The table was used to design the CPU Controller unit.

Single Cycle CPU:



	Msgs
/lall_cpu_controller/Lall_op	000100
/lall_cpu_controller/Lall_func	000011
/lall_cpu_controller/Lall_equal	0
/lall_cpu_controller/Lall_Re…	X
/lall_cpu_controller/Lall_Re…	0
/lall_cpu_controller/Lall_AL…	0001
/lall_cpu_controller/Lall_Me…	0
/lall_cpu_controller/Lall_Me…	0
/lall_cpu_controller/Lall_AL…	0
/lall_cpu_controller/Lall_Ext…	1
/lall_cpu_controller/Lall_PCSrc	0

Figure 47. Single Cycle CPU waveform.

This simulation above shows given opcodes and functions and the output that is given. For opcode 000000, it is an R-type instruction so it relies on the function. However, if we test opcodes 00100, 001100, and 000100, they do not rely on the function since they are I-type instruction based. One opcode that I did not show was the j-type instruction, but the opcode is 00010.

Component Package:

Included in the component package are the following components:

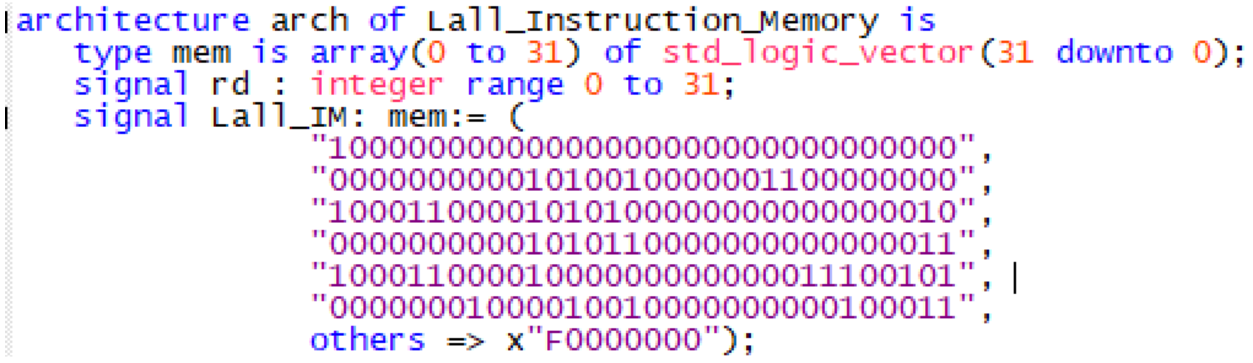
- 32-bit adder
- 32bit Mux
- Program Counter
- 5bit Mux

- Next Address Logic (NAL)
- Instruction register
- Instruction memory
- Register File
- Data memory
- ALU
- Sign Extender
- Overflow
- CPU Controller

Single Cycle CPU:

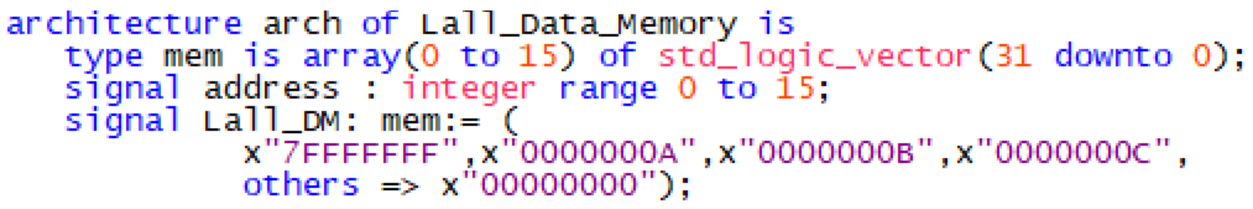
In this section, I will demonstrate the CPU and its required instructions.

ADDI, ORI, ADDIU, NOR, LW Instructions:



```
architecture arch of Lall_Instruction_Memory is
    type mem is array(0 to 31) of std_logic_vector(31 downto 0);
    signal rd : integer range 0 to 31;
    signal Lall_IM: mem:= (
                    "10000000000000000000000000000000",
                    "00000000001010010000001100000000",
                    "10001100001010100000000000000010",
                    "00000000001010110000000000000011",
                    "10001100001000000000000011100101",
                    "00000001000010010000000000100011",
                    others => x"F0000000");
```

Figure 48. Updated code for instruction memory for instructions.



```
architecture arch of Lall_Data_Memory is
   type mem is array(0 to 15) of std_logic_vector(31 downto 0);
   signal address : integer range 0 to 15;
   signal Lall_DM: mem:= (
            x"7FFFFFFF",x"0000000A",x"0000000B",x"0000000C",
            others => x"00000000");
```

Figure 49. Updated data memory for instructions.

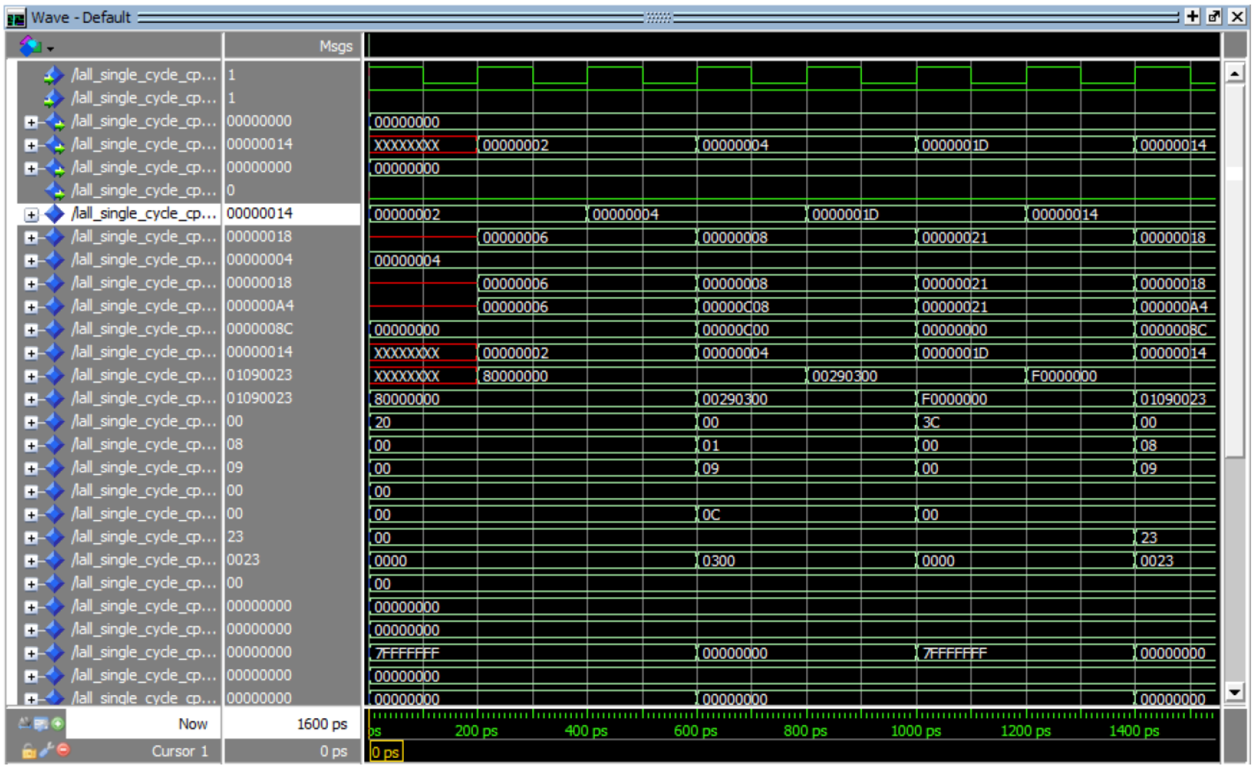
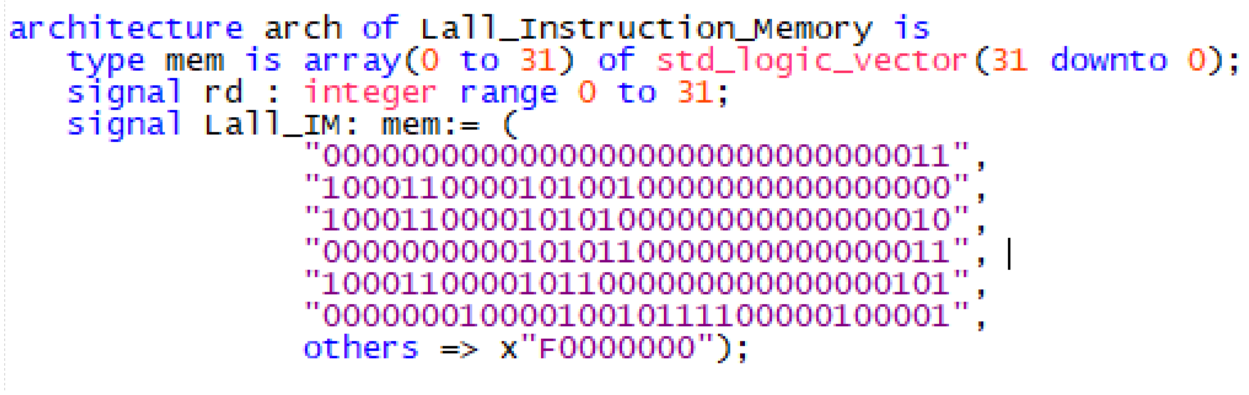


Figure 50. Modelsim waveform for ADDI, ORI, ADDIU, NOR, LW Instruction

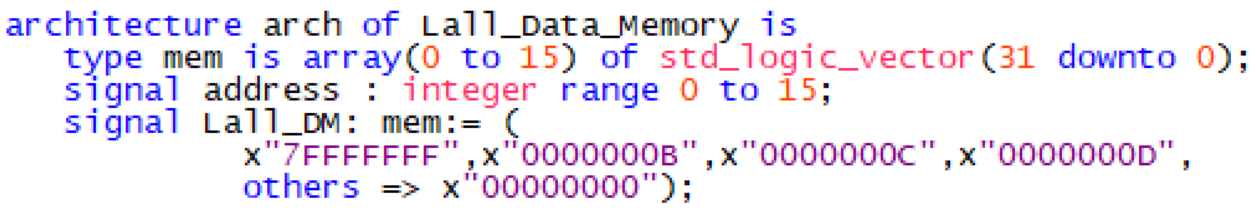
In this simulation we see the PC pointing to the next address and hold desired instruction. You can see the ADDI instruction with the use of the IMM16. We can also see the ORI with the IMM16 . We load values from our data memory, into the register. ADDIU is also shown as it does addition and won't trigger overflow flag.

LW, ADD, ADU, SW Instructions:



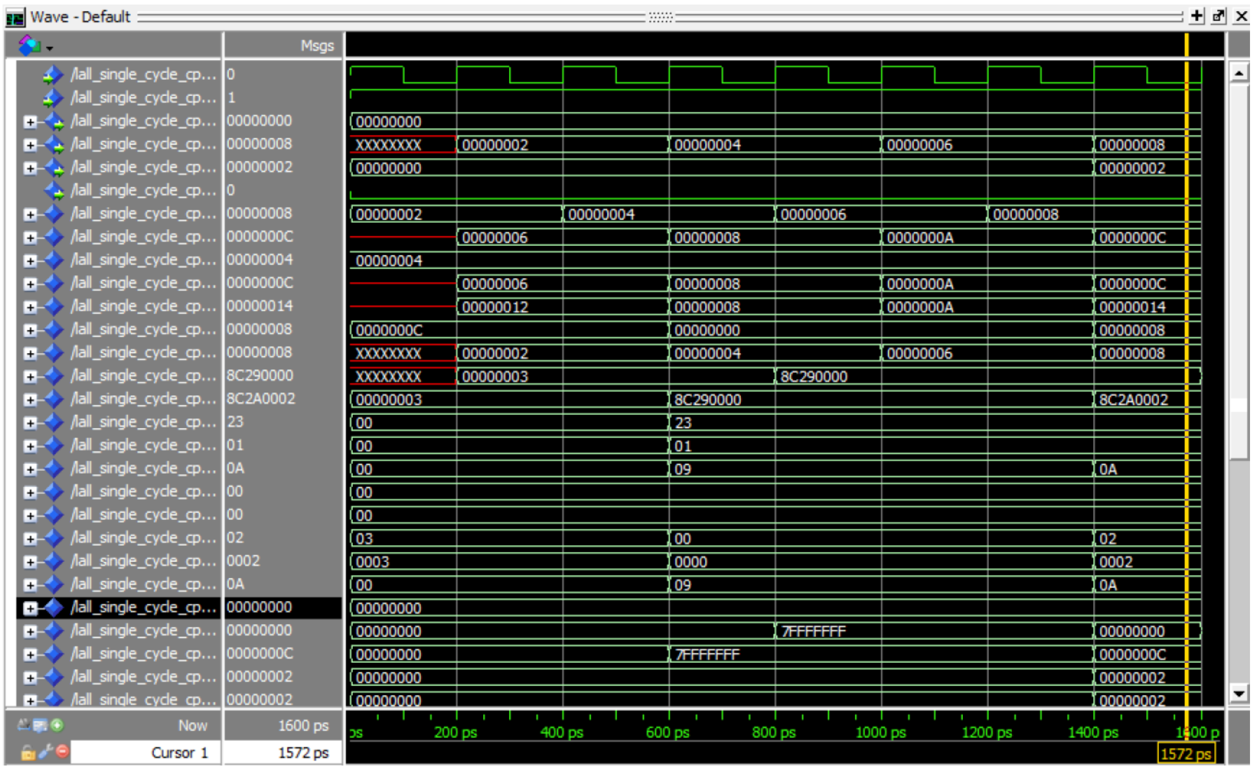
```
architecture arch of Lall_Instruction_Memory is
    type mem is array(0 to 31) of std_logic_vector(31 downto 0);
    signal rd : integer range 0 to 31;
    signal Lall_IM: mem:= (
                "00000000000000000000000000000011",
                "10001100001010010000000000000000",
                "10001100001010100000000000000010",
                "00000000001010110000000000000011",
                "10001100001011000000000000000101",
                "00000001000010010111100000100001",
                others => x"F0000000");
```

Figure 51. Updated instruction memory code for instructions.



```
architecture arch of Lall_Data_Memory is
   type mem is array(0 to 15) of std_logic_vector(31 downto 0);
   signal address : integer range 0 to 15;
   signal Lall_DM: mem:= (
            x"7FFFFFFF",x"0000000B",x"0000000C",x"0000000D",
            others => x"00000000");
```

Figure 52. Updated data memory code for instructions.



```
Wave - Default
                                Msgs
/lall_single_cycle_cp...   0
/lall_single_cycle_cp...   1
/lall_single_cycle_cp...   00000000
/lall_single_cycle_cp...   00000008
/lall_single_cycle_cp...   00000002
/lall_single_cycle_cp...   0
/lall_single_cycle_cp...   00000008
/lall_single_cycle_cp...   0000000C
/lall_single_cycle_cp...   00000004
/lall_single_cycle_cp...   0000000C
/lall_single_cycle_cp...   00000014
/lall_single_cycle_cp...   00000008
/lall_single_cycle_cp...   00000008
/lall_single_cycle_cp...   8C290000
/lall_single_cycle_cp...   8C2A0002
/lall_single_cycle_cp...   23
/lall_single_cycle_cp...   01
/lall_single_cycle_cp...   0A
/lall_single_cycle_cp...   00
/lall_single_cycle_cp...   00
/lall_single_cycle_cp...   02
/lall_single_cycle_cp...   0002
/lall_single_cycle_cp...   0A
/lall_single_cycle_cp...   00000000
/lall_single_cycle_cp...   00000000
/lall_single_cycle_cp...   0000000C
/lall_single_cycle_cp...   00000002
/lall_single_cycle_cp...   00000002
                Now        1600 ps
            Cursor 1       1572 ps
```

Figure 53. Modelsim simulation for LW, ADD, ADDU, SW Instruction.

The waveform above shows that for our first clock cycle, nothing is changing since we can not perform anything. The PC will hold the next address due to our components. The second instruction s then given when the second clock cycle begins. For me that is address 0x8C290000. This loads the value of data memory which if you recall from the edited VHDL code above, it is 0x7FFFFFF since it was initialized data. The PC now moves to the following address 0x00000004. Here, $9 is loaded onto the register in the next clock cycle and points to the next address. This process continue to repeat. The ADDU operation was used for the next instruction to obtain the same result as ADD except for triggering the overflow flag.

BEQ, BNE, J, SUB:

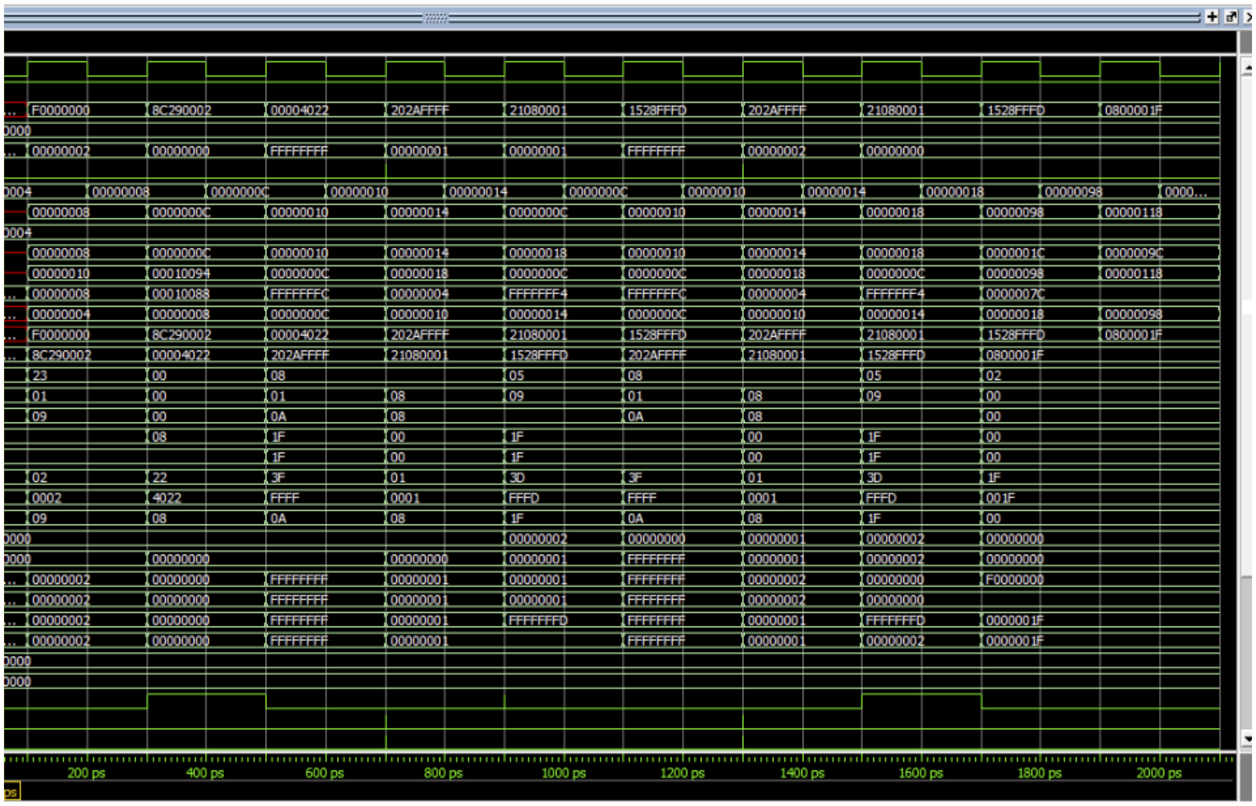
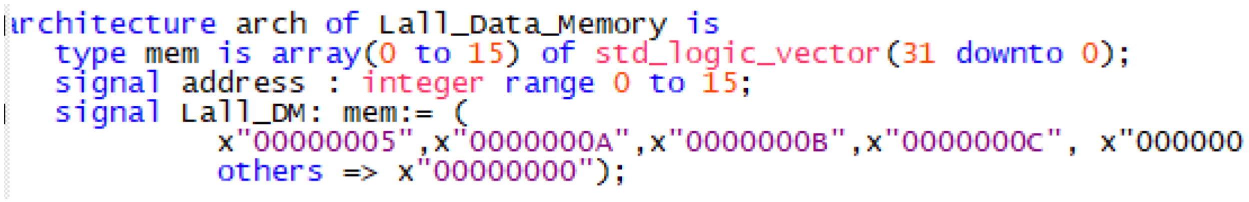


Figure 54. Modelsim simulation BEQ, BNE, J, SUB instructions

Ultimate Test:

```
architecture arch of Lall_Instruction_Memory is
    type mem is array(0 to 31) of std_logic_vector(31 downto 0);
    signal rd : integer range 0 to 31;
    signal Lall_IM: mem:= (
                        "10001100000000000001000000000000",
                        "10001100001010010000001100000000",
                        "10000000001010100000000000000010",
                        "00000000001010110000000000000011",
                        "10001100001000000000000011100101",
                        "00000001000010010000000000100000",
                        "00000001000010011110000000100000",
                        "00000001000010010000000000100000",
                        "00001101000010010000000000100000",
                        "11000001000010010000000000100000",
                        others => x"F0000000");
```

Figure 55.instruction memory updated code



```
rchitecture arch of Lall_Data_Memory is
  type mem is array(0 to 15) of std_logic_vector(31 downto 0);
  signal address : integer range 0 to 15;
  signal Lall_DM: mem:= (
            x"00000005",x"0000000A",x"0000000B",x"0000000C", x"000000
            others => x"00000000");
```

Figure 56. data memory updated code

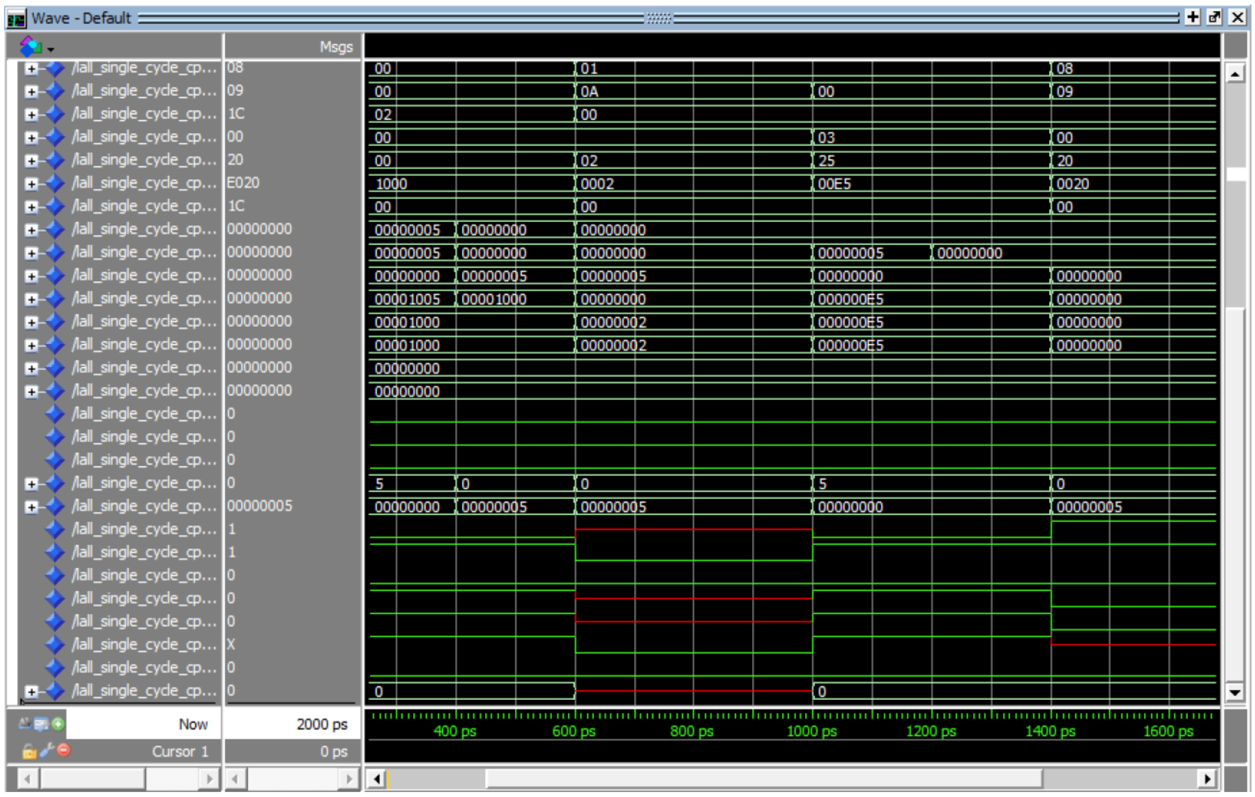
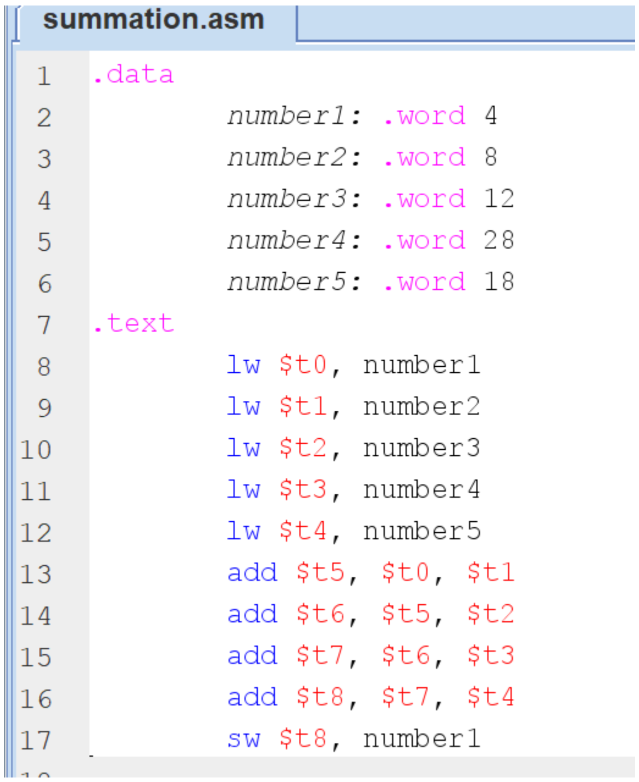


Figure 57. Modelsim simulation example

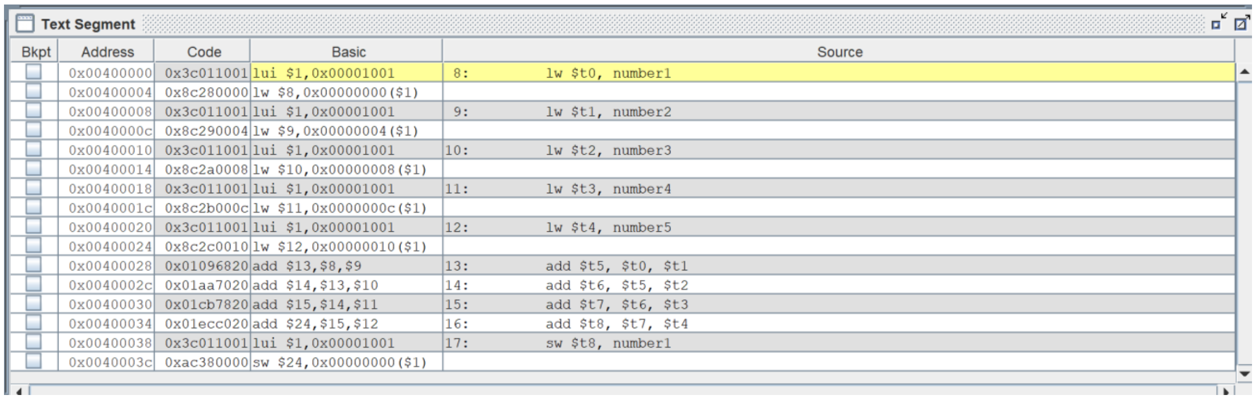
Mips Summation:



summation.asm

```
.data
          number1: .word 4
          number2: .word 8
          number3: .word 12
          number4: .word 28
          number5: .word 18
.text
          lw $t0, number1
          lw $t1, number2
          lw $t2, number3
          lw $t3, number4
          lw $t4, number5
          add $t5, $t0, $t1
          add $t6, $t5, $t2
          add $t7, $t6, $t3
          add $t8, $t7, $t4
          sw $t8, number1
```

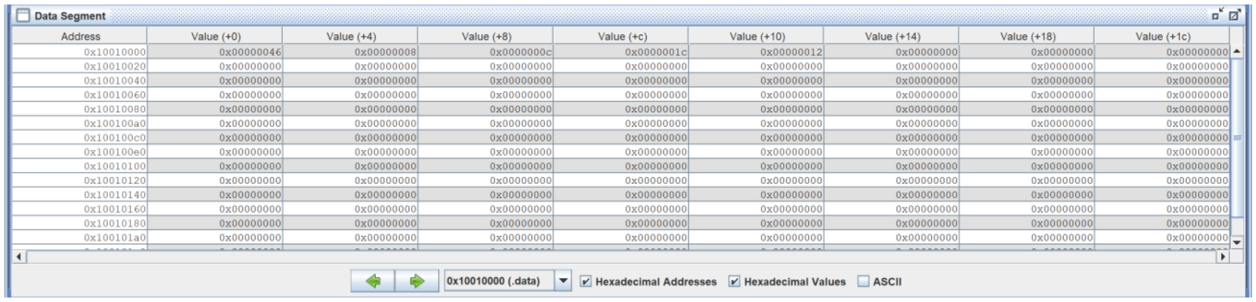
Figure 58. Summation code .asm



Text Segment

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c011001	lui $1,0x00001001	8: lw $t0, number1
☐	0x00400004	0x8c280000	lw $8,0x00000000($1)	
☐	0x00400008	0x3c011001	lui $1,0x00001001	9: lw $t1, number2
☐	0x0040000c	0x8c290004	lw $9,0x00000004($1)	
☐	0x00400010	0x3c011001	lui $1,0x00001001	10: lw $t2, number3
☐	0x00400014	0x8c2a0008	lw $10,0x00000008($1)	
☐	0x00400018	0x3c011001	lui $1,0x00001001	11: lw $t3, number4
☐	0x0040001c	0x8c2b000c	lw $11,0x0000000c($1)	
☐	0x00400020	0x3c011001	lui $1,0x00001001	12: lw $t4, number5
☐	0x00400024	0x8c2c0010	lw $12,0x00000010($1)	
☐	0x00400028	0x01096820	add $13,$8,$9	13: add $t5, $t0, $t1
☐	0x0040002c	0x01aa7020	add $14,$13,$10	14: add $t6, $t5, $t2
☐	0x00400030	0x01cb7820	add $15,$14,$11	15: add $t7, $t6, $t3
☐	0x00400034	0x01ecc020	add $24,$15,$12	16: add $t8, $t7, $t4
☐	0x00400038	0x3c011001	lui $1,0x00001001	17: sw $t8, number1
☐	0x0040003c	0xac380000	sw $24,0x00000000($1)	

Figure 59. text segment



Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000046	0x00000008	0x0000000c	0x0000001c	0x00000012	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ✓ Hexadecimal Addresses ✓ Hexadecimal Values ☐ ASCII

Figure 60. data segment

Conclusion:

I learned a lot from this project. Using all components that we have previous used has allowed me to create this Single Cycle CPU and it taught me to continue to reuse code experience that we had used. I got a better understanding on how to use advanced ideas in Quartus and I was able to complete the given task in the final lab.