Final Project: Designing a Processor to Compute Dot Product of Two Vectors in VHDL

Zi Xuan Li

The Grove School of Engineering, The City College of New York

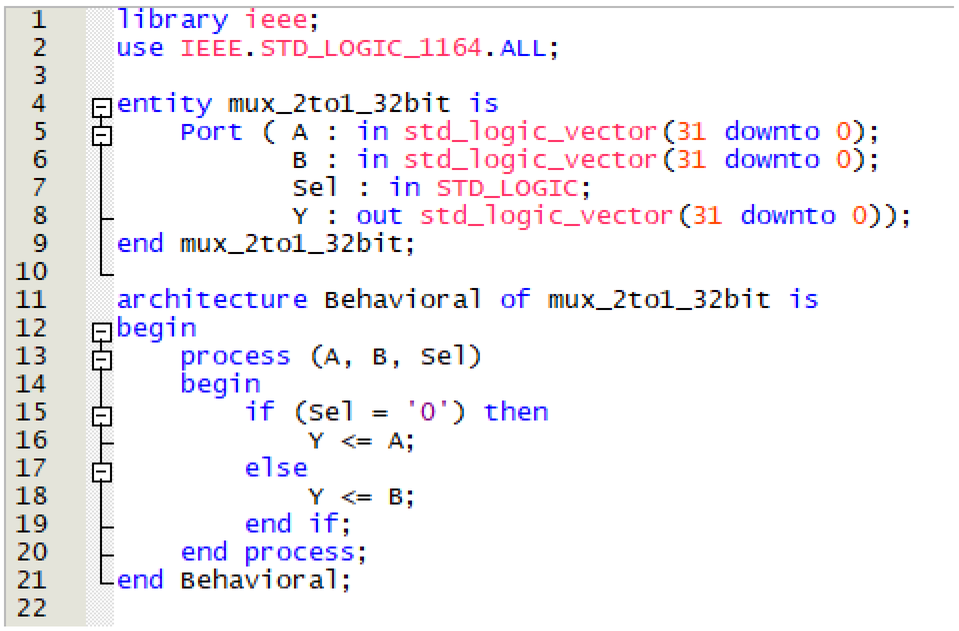
CSC 34300 5DE[43223]: Computer Systems Design Laboratory

Professor Izidor Gertner, TA: Albi Arapi

May 13th, 2024

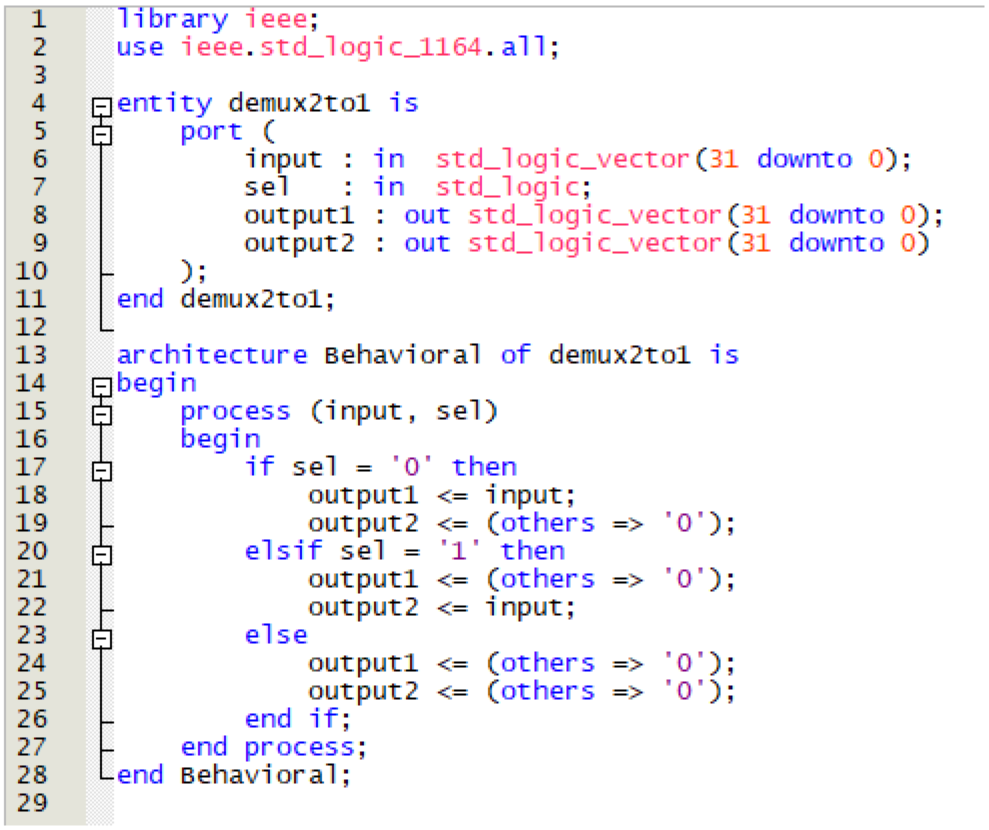
Objective:

To implement and validate the functionality of a digital signal processing system by initializing SRAM with two 32-bit integer vectors, feeding these vectors into a multiplier and add/sub unit designed in VHDL, and verifying the accuracy of the resulting dot product through simulation.



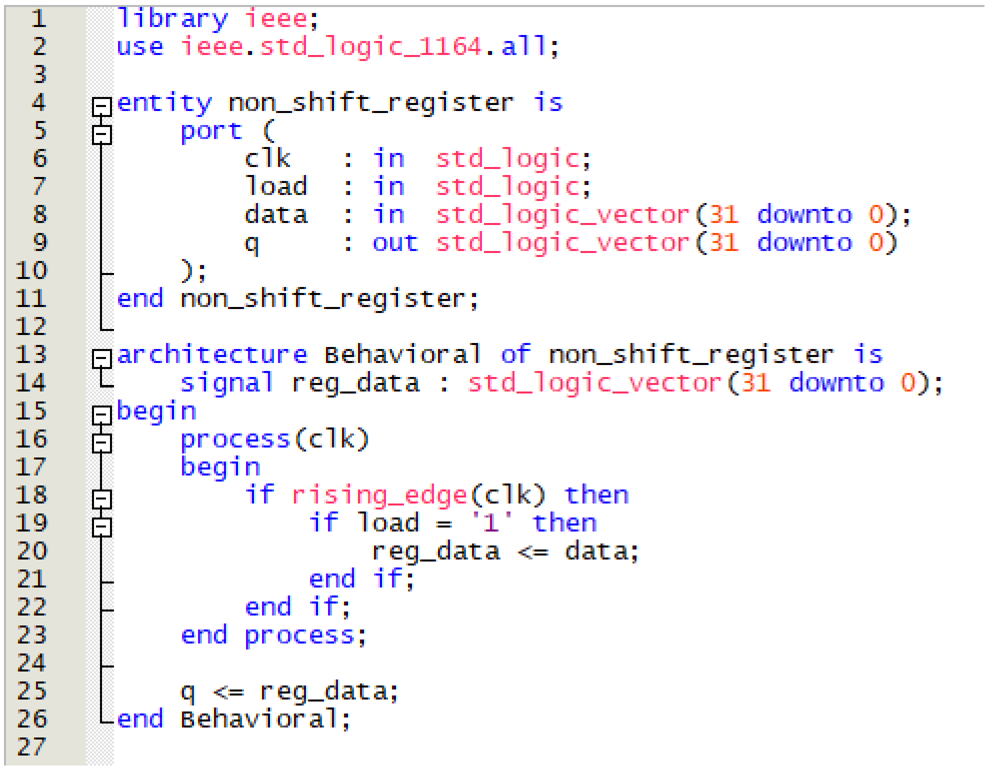
```vhdl
library ieee;
use IEEE.STD_LOGIC_1164.ALL;

entity mux_2to1_32bit is
    Port ( A : in std_logic_vector(31 downto 0);
           B : in std_logic_vector(31 downto 0);
           Sel : in STD_LOGIC;
           Y : out std_logic_vector(31 downto 0));
end mux_2to1_32bit;

architecture Behavioral of mux_2to1_32bit is
begin
    process (A, B, Sel)
    begin
        if (Sel = '0') then
            Y <= A;
        else
            Y <= B;
        end if;
    end process;
end Behavioral;
```

Figure 1: Image of the 32-bit 2 to 1 multiplexer VHDL code.



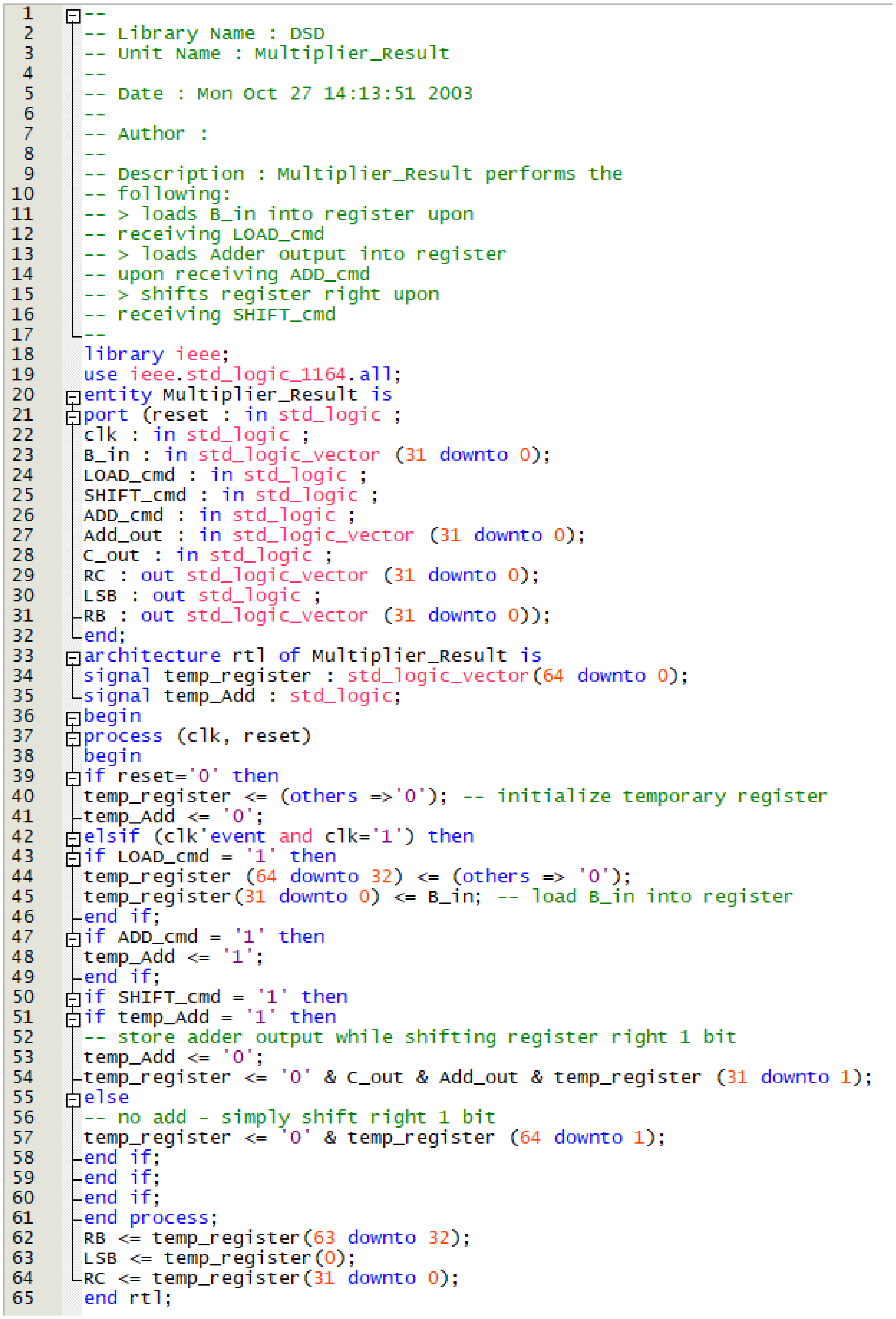
```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity demux2to1 is
    port (
        input : in  std_logic_vector(31 downto 0);
        sel   : in  std_logic;
        output1 : out std_logic_vector(31 downto 0);
        output2 : out std_logic_vector(31 downto 0)
    );
end demux2to1;

architecture Behavioral of demux2to1 is
begin
    process (input, sel)
    begin
        if sel = '0' then
            output1 <= input;
            output2 <= (others => '0');
        elsif sel = '1' then
            output1 <= (others => '0');
            output2 <= input;
        else
            output1 <= (others => '0');
            output2 <= (others => '0');
        end if;
    end process;
end Behavioral;
```

Figure 2: Image of the 32-bit 2 to 1 demultiplexer VHDL code.



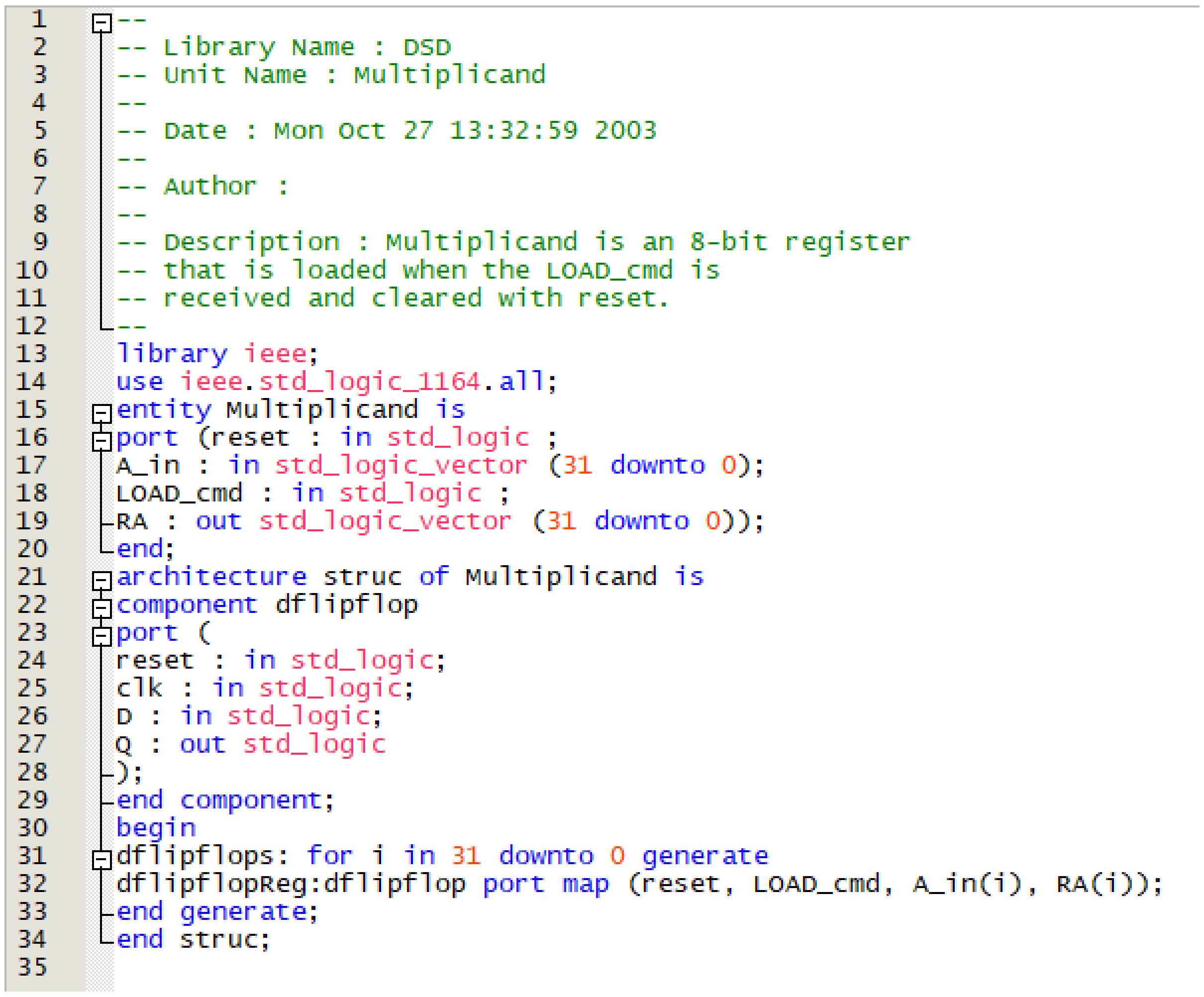
```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity non_shift_register is
    port (
        clk   : in  std_logic;
        load  : in  std_logic;
        data  : in  std_logic_vector(31 downto 0);
        q     : out std_logic_vector(31 downto 0)
    );
end non_shift_register;

architecture Behavioral of non_shift_register is
    signal reg_data : std_logic_vector(31 downto 0);
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if load = '1' then
                reg_data <= data;
            end if;
        end if;
    end process;

    q <= reg_data;
end Behavioral;
```

Figure 3: Image of the 32-bit non-shift register VHDL code.



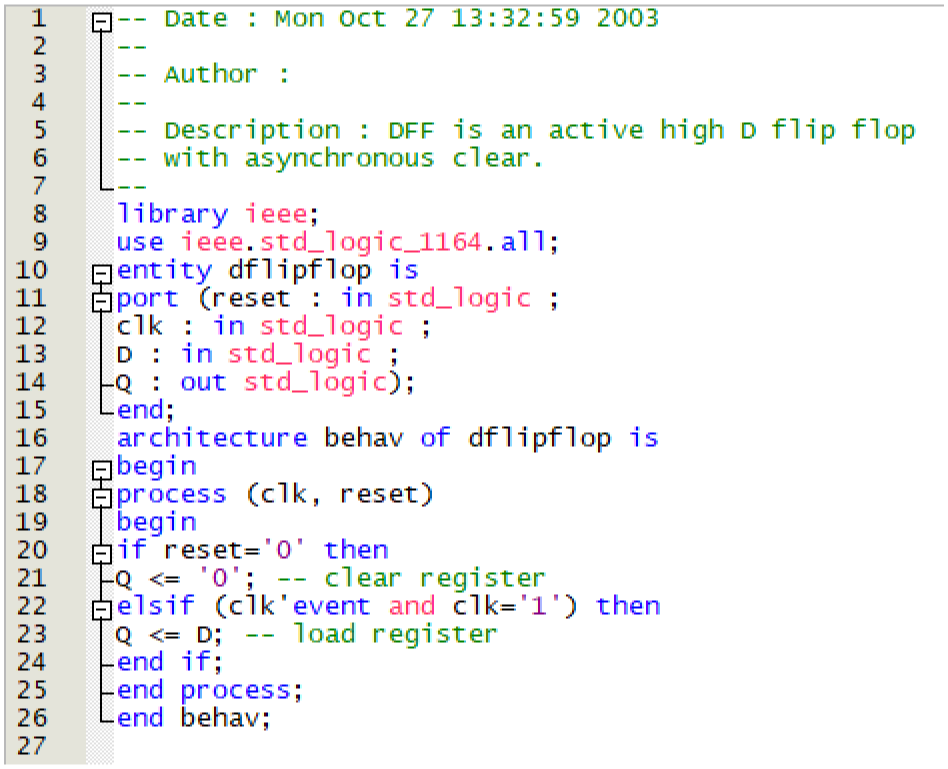
```
--
-- Library Name : DSD
-- Unit Name : Multiplier_Result
--
-- Date : Mon Oct 27 14:13:51 2003
--
-- Author :
--
-- Description : Multiplier_Result performs the
-- following:
-- > loads B_in into register upon
-- receiving LOAD_cmd
-- > loads Adder output into register
-- upon receiving ADD_cmd
-- > shifts register right upon
-- receiving SHIFT_cmd
--
library ieee;
use ieee.std_logic_1164.all;
entity Multiplier_Result is
port (reset : in std_logic ;
clk : in std_logic ;
B_in : in std_logic_vector (31 downto 0);
LOAD_cmd : in std_logic ;
SHIFT_cmd : in std_logic ;
ADD_cmd : in std_logic ;
Add_out : in std_logic_vector (31 downto 0);
C_out : in std_logic ;
RC : out std_logic_vector (31 downto 0);
LSB : out std_logic ;
RB : out std_logic_vector (31 downto 0));
end;
architecture rtl of Multiplier_Result is
signal temp_register : std_logic_vector(64 downto 0);
signal temp_Add : std_logic;
begin
process (clk, reset)
begin
if reset='0' then
temp_register <= (others =>'0'); -- initialize temporary register
temp_Add <= '0';
elsif (clk'event and clk='1') then
if LOAD_cmd = '1' then
temp_register (64 downto 32) <= (others => '0');
temp_register(31 downto 0) <= B_in; -- load B_in into register
end if;
if ADD_cmd = '1' then
temp_Add <= '1';
end if;
if SHIFT_cmd = '1' then
if temp_Add = '1' then
-- store adder output while shifting register right 1 bit
temp_Add <= '0';
temp_register <= '0' & C_out & Add_out & temp_register (31 downto 1);
else
-- no add - simply shift right 1 bit
temp_register <= '0' & temp_register (64 downto 1);
end if;
end if;
end if;
end process;
RB <= temp_register(63 downto 32);
LSB <= temp_register(0);
RC <= temp_register(31 downto 0);
end rtl;
```

Figure 4: Image of the Multiplier Result VHDL code used in the Multiplier unit.



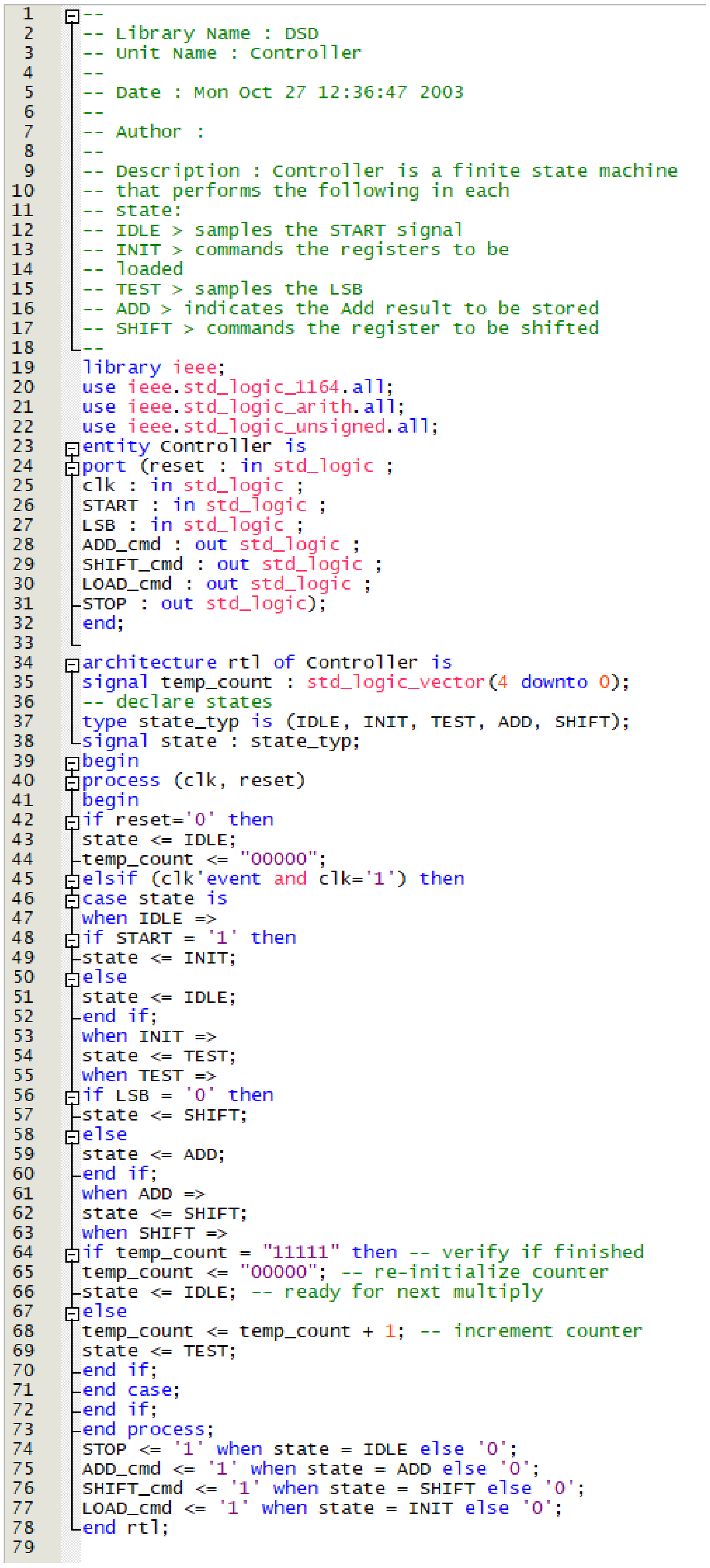
```
--
-- Library Name : DSD
-- Unit Name : Multiplicand
--
-- Date : Mon Oct 27 13:32:59 2003
--
-- Author :
--
-- Description : Multiplicand is an 8-bit register
-- that is loaded when the LOAD_cmd is
-- received and cleared with reset.
--
library ieee;
use ieee.std_logic_1164.all;
entity Multiplicand is
port (reset : in std_logic ;
A_in : in std_logic_vector (31 downto 0);
LOAD_cmd : in std_logic ;
RA : out std_logic_vector (31 downto 0));
end;
architecture struc of Multiplicand is
component dflipflop
port (
reset : in std_logic;
clk : in std_logic;
D : in std_logic;
Q : out std_logic
);
end component;
begin
dflipflops: for i in 31 downto 0 generate
dflipflopReg:dflipflop port map (reset, LOAD_cmd, A_in(i), RA(i));
end generate;
end struc;
```

Figure 5: Image of the Multiplicand VHDL code used in the Multiplier unit.



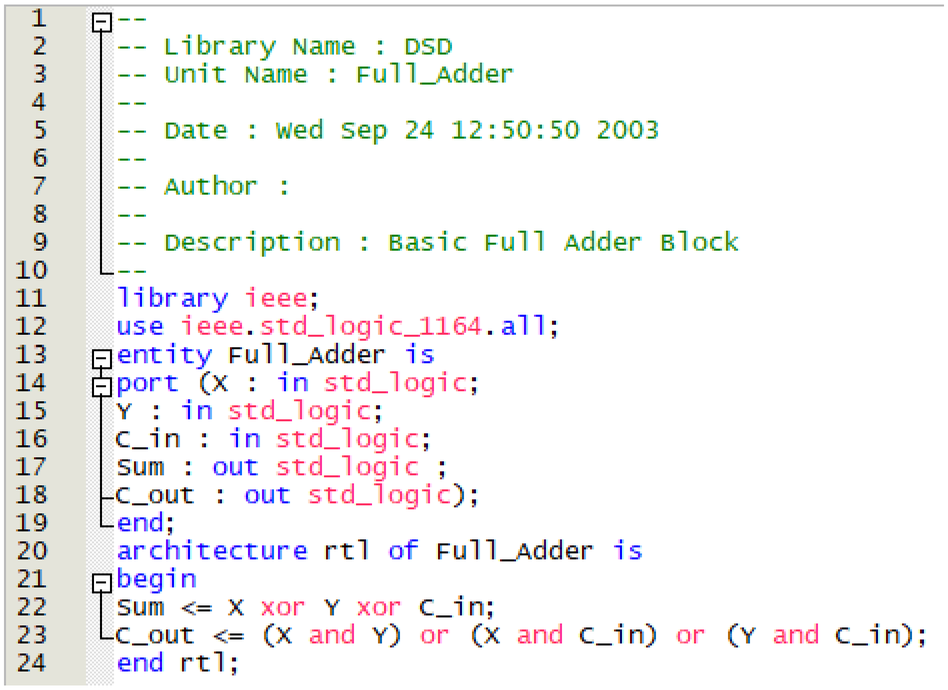
```vhdl
-- Date : Mon Oct 27 13:32:59 2003
--
-- Author :
--
-- Description : DFF is an active high D flip flop
-- with asynchronous clear.
--
library ieee;
use ieee.std_logic_1164.all;
entity dflipflop is
port (reset : in std_logic ;
clk : in std_logic ;
D : in std_logic ;
Q : out std_logic);
end;
architecture behav of dflipflop is
begin
process (clk, reset)
begin
if reset='0' then
Q <= '0'; -- clear register
elsif (clk'event and clk='1') then
Q <= D; -- load register
end if;
end process;
end behav;
```

Figure 6: Image of the D-flip flop VHDL code used in the Multiplicand unit.



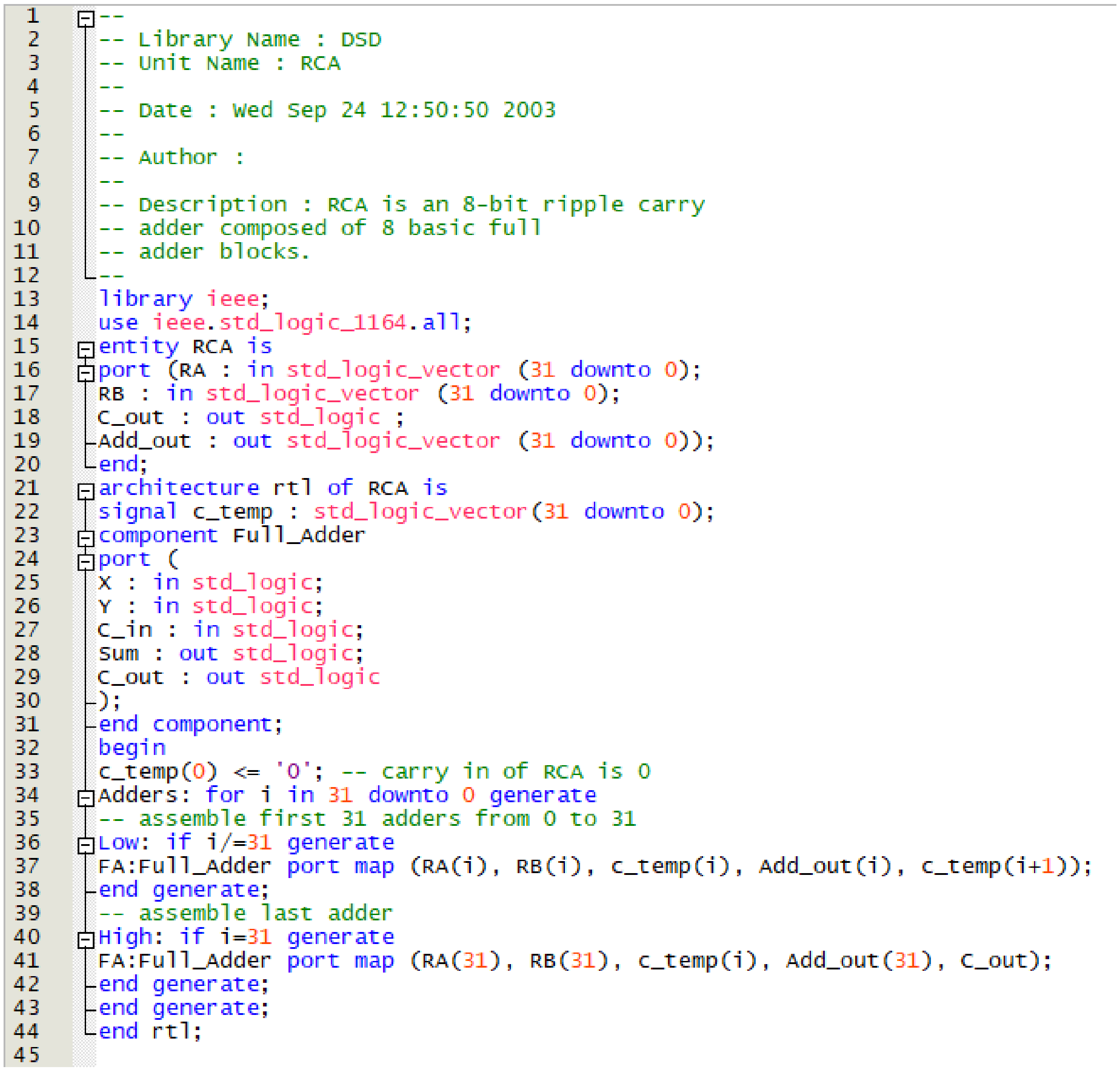
```vhdl
--
-- Library Name : DSD
-- Unit Name : Controller
--
-- Date : Mon Oct 27 12:36:47 2003
--
-- Author :
--
-- Description : Controller is a finite state machine
-- that performs the following in each
-- state:
-- IDLE > samples the START signal
-- INIT > commands the registers to be
-- loaded
-- TEST > samples the LSB
-- ADD > indicates the Add result to be stored
-- SHIFT > commands the register to be shifted
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity Controller is
port (reset : in std_logic ;
clk : in std_logic ;
START : in std_logic ;
LSB : in std_logic ;
ADD_cmd : out std_logic ;
SHIFT_cmd : out std_logic ;
LOAD_cmd : out std_logic ;
STOP : out std_logic);
end;

architecture rtl of Controller is
signal temp_count : std_logic_vector(4 downto 0);
-- declare states
type state_typ is (IDLE, INIT, TEST, ADD, SHIFT);
signal state : state_typ;
begin
process (clk, reset)
begin
if reset='0' then
state <= IDLE;
temp_count <= "00000";
elsif (clk'event and clk='1') then
case state is
when IDLE =>
if START = '1' then
state <= INIT;
else
state <= IDLE;
end if;
when INIT =>
state <= TEST;
when TEST =>
if LSB = '0' then
state <= SHIFT;
else
state <= ADD;
end if;
when ADD =>
state <= SHIFT;
when SHIFT =>
if temp_count = "11111" then -- verify if finished
temp_count <= "00000"; -- re-initialize counter
state <= IDLE; -- ready for next multiply
else
temp_count <= temp_count + 1; -- increment counter
state <= TEST;
end if;
end case;
end if;
end process;
STOP <= '1' when state = IDLE else '0';
ADD_cmd <= '1' when state = ADD else '0';
SHIFT_cmd <= '1' when state = SHIFT else '0';
LOAD_cmd <= '1' when state = INIT else '0';
end rtl;

```

Figure 7: Image of the Controller VHDL code used in the Multiplier unit.



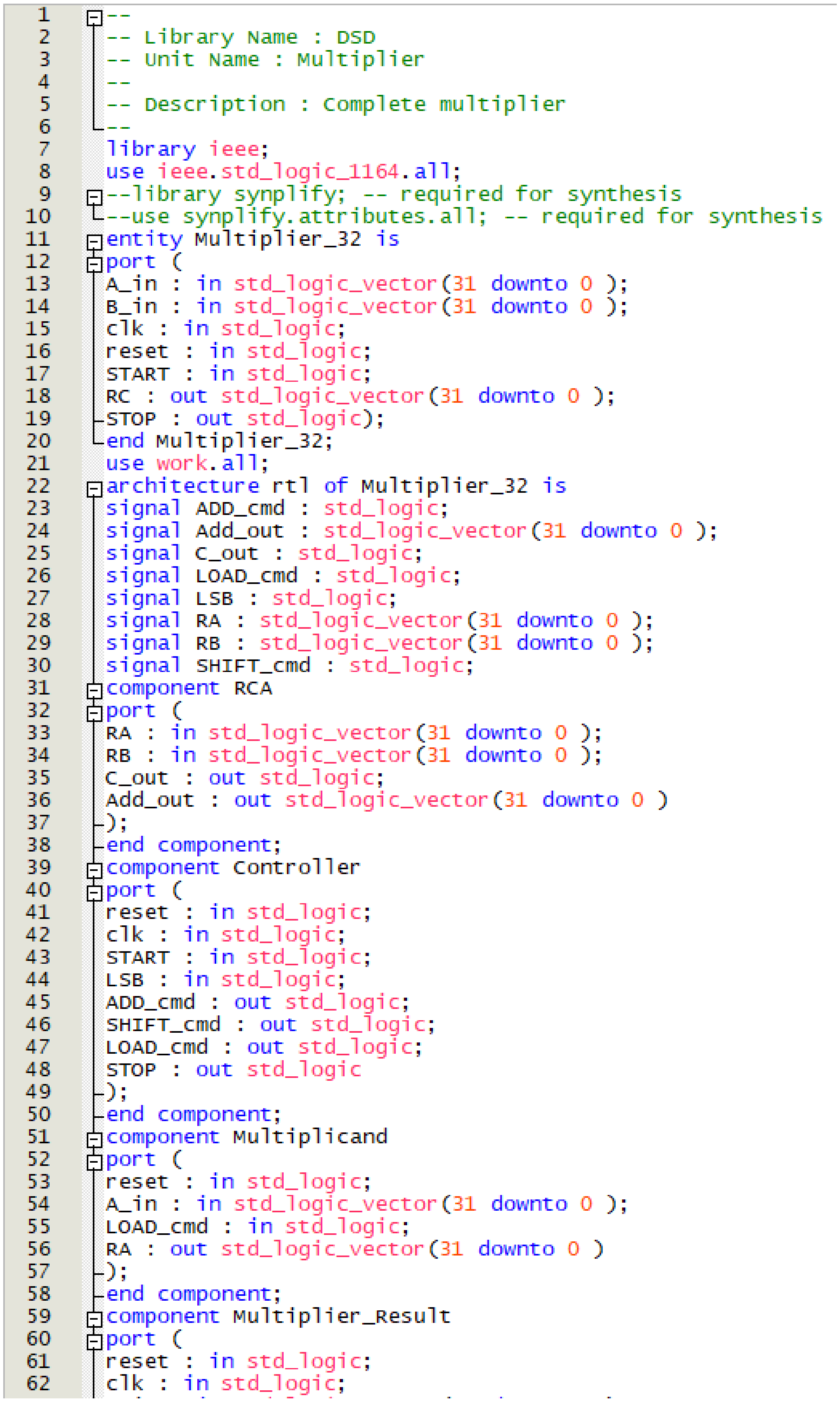
```
--
-- Library Name : DSD
-- Unit Name : Full_Adder
--
-- Date : Wed Sep 24 12:50:50 2003
--
-- Author :
--
-- Description : Basic Full Adder Block
--
library ieee;
use ieee.std_logic_1164.all;
entity Full_Adder is
port (X : in std_logic;
Y : in std_logic;
C_in : in std_logic;
Sum : out std_logic ;
C_out : out std_logic);
end;
architecture rtl of Full_Adder is
begin
Sum <= X xor Y xor C_in;
C_out <= (X and Y) or (X and C_in) or (Y and C_in);
end rtl;
```

Figure 8: Image of the Full Adder VHDL code used in the Multiplier unit.

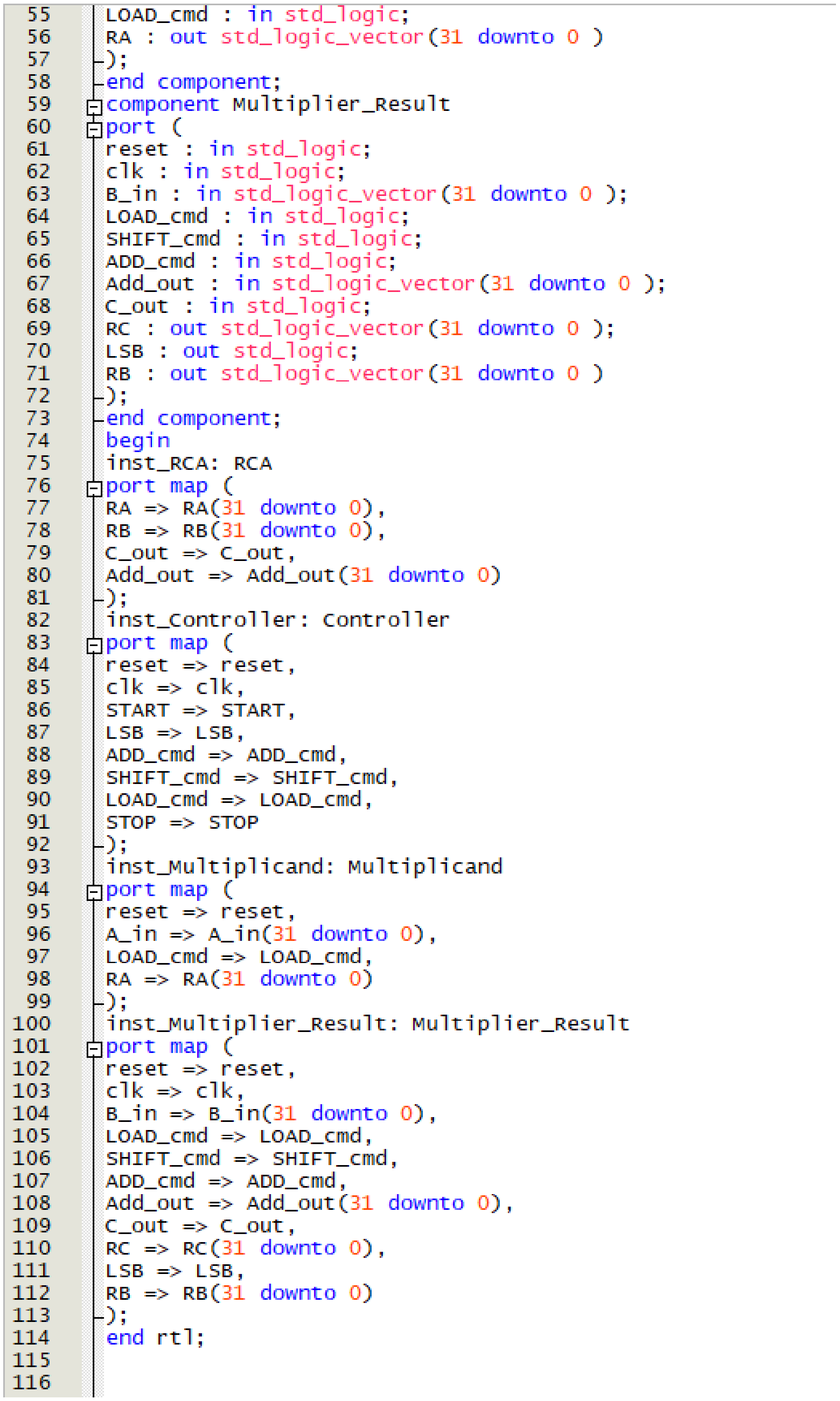


```
--
-- Library Name : DSD
-- Unit Name : RCA
--
-- Date : Wed Sep 24 12:50:50 2003
--
-- Author :
--
-- Description : RCA is an 8-bit ripple carry
-- adder composed of 8 basic full
-- adder blocks.
--
library ieee;
use ieee.std_logic_1164.all;
entity RCA is
port (RA : in std_logic_vector (31 downto 0);
RB : in std_logic_vector (31 downto 0);
C_out : out std_logic ;
Add_out : out std_logic_vector (31 downto 0));
end;
architecture rtl of RCA is
signal c_temp : std_logic_vector(31 downto 0);
component Full_Adder
port (
X : in std_logic;
Y : in std_logic;
C_in : in std_logic;
Sum : out std_logic;
C_out : out std_logic
);
end component;
begin
c_temp(0) <= '0'; -- carry in of RCA is 0
Adders: for i in 31 downto 0 generate
-- assemble first 31 adders from 0 to 31
Low: if i/=31 generate
FA:Full_Adder port map (RA(i), RB(i), c_temp(i), Add_out(i), c_temp(i+1));
end generate;
-- assemble last adder
High: if i=31 generate
FA:Full_Adder port map (RA(31), RB(31), c_temp(i), Add_out(31), C_out);
end generate;
end generate;
end rtl;
```

Figure 9: Image of the RCA VHDL code used in the Multiplier unit.

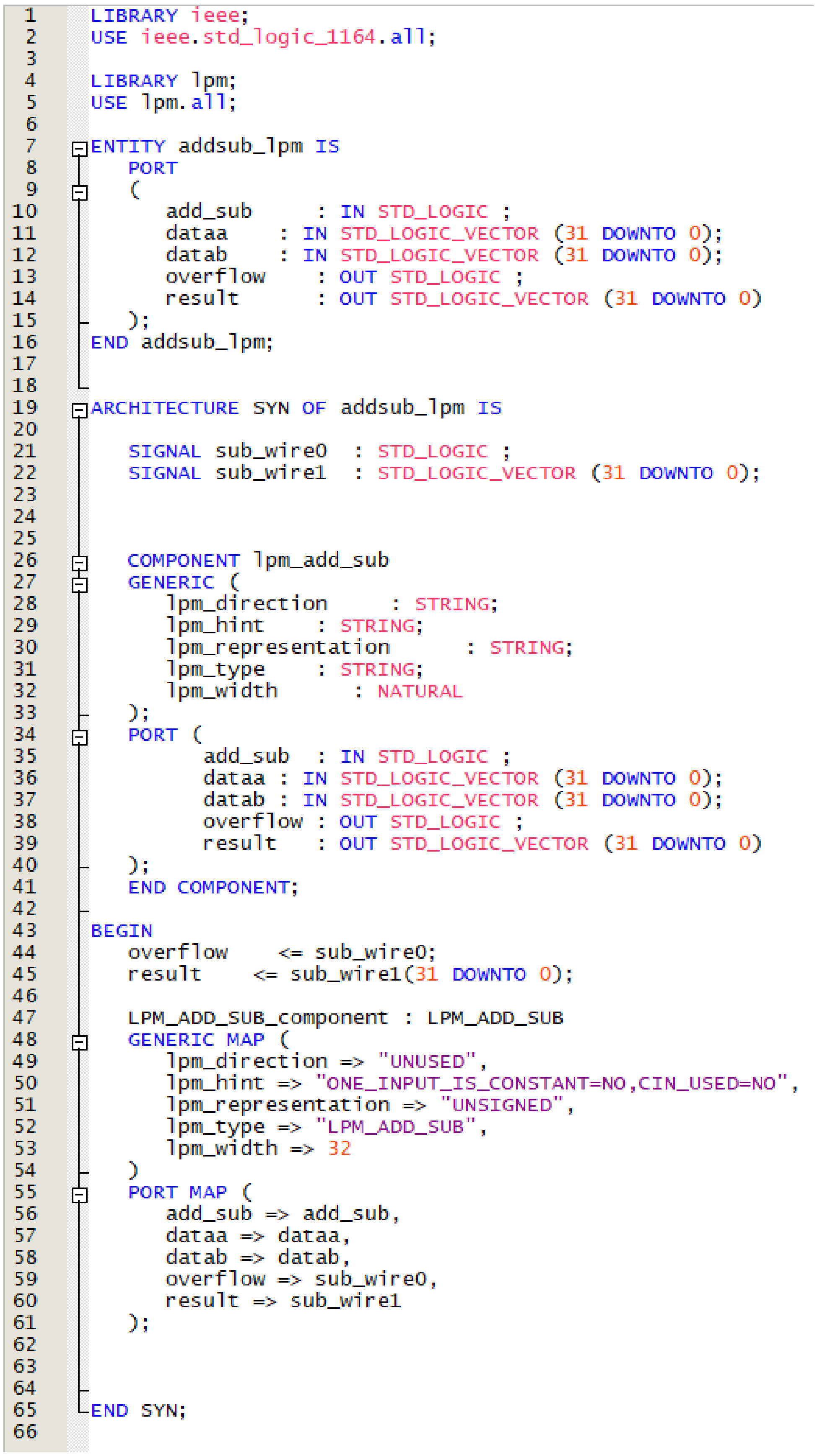


```vhdl
--
-- Library Name : DSD
-- Unit Name : Multiplier
--
-- Description : Complete multiplier
--
library ieee;
use ieee.std_logic_1164.all;
--library synplify; -- required for synthesis
--use synplify.attributes.all; -- required for synthesis
entity Multiplier_32 is
port (
A_in : in std_logic_vector(31 downto 0 );
B_in : in std_logic_vector(31 downto 0 );
clk : in std_logic;
reset : in std_logic;
START : in std_logic;
RC : out std_logic_vector(31 downto 0 );
STOP : out std_logic);
end Multiplier_32;
use work.all;
architecture rtl of Multiplier_32 is
signal ADD_cmd : std_logic;
signal Add_out : std_logic_vector(31 downto 0 );
signal C_out : std_logic;
signal LOAD_cmd : std_logic;
signal LSB : std_logic;
signal RA : std_logic_vector(31 downto 0 );
signal RB : std_logic_vector(31 downto 0 );
signal SHIFT_cmd : std_logic;
component RCA
port (
RA : in std_logic_vector(31 downto 0 );
RB : in std_logic_vector(31 downto 0 );
C_out : out std_logic;
Add_out : out std_logic_vector(31 downto 0 )
);
end component;
component Controller
port (
reset : in std_logic;
clk : in std_logic;
START : in std_logic;
LSB : in std_logic;
ADD_cmd : out std_logic;
SHIFT_cmd : out std_logic;
LOAD_cmd : out std_logic;
STOP : out std_logic
);
end component;
component Multiplicand
port (
reset : in std_logic;
A_in : in std_logic_vector(31 downto 0 );
LOAD_cmd : in std_logic;
RA : out std_logic_vector(31 downto 0 )
);
end component;
component Multiplier_Result
port (
reset : in std_logic;
clk : in std_logic;
```



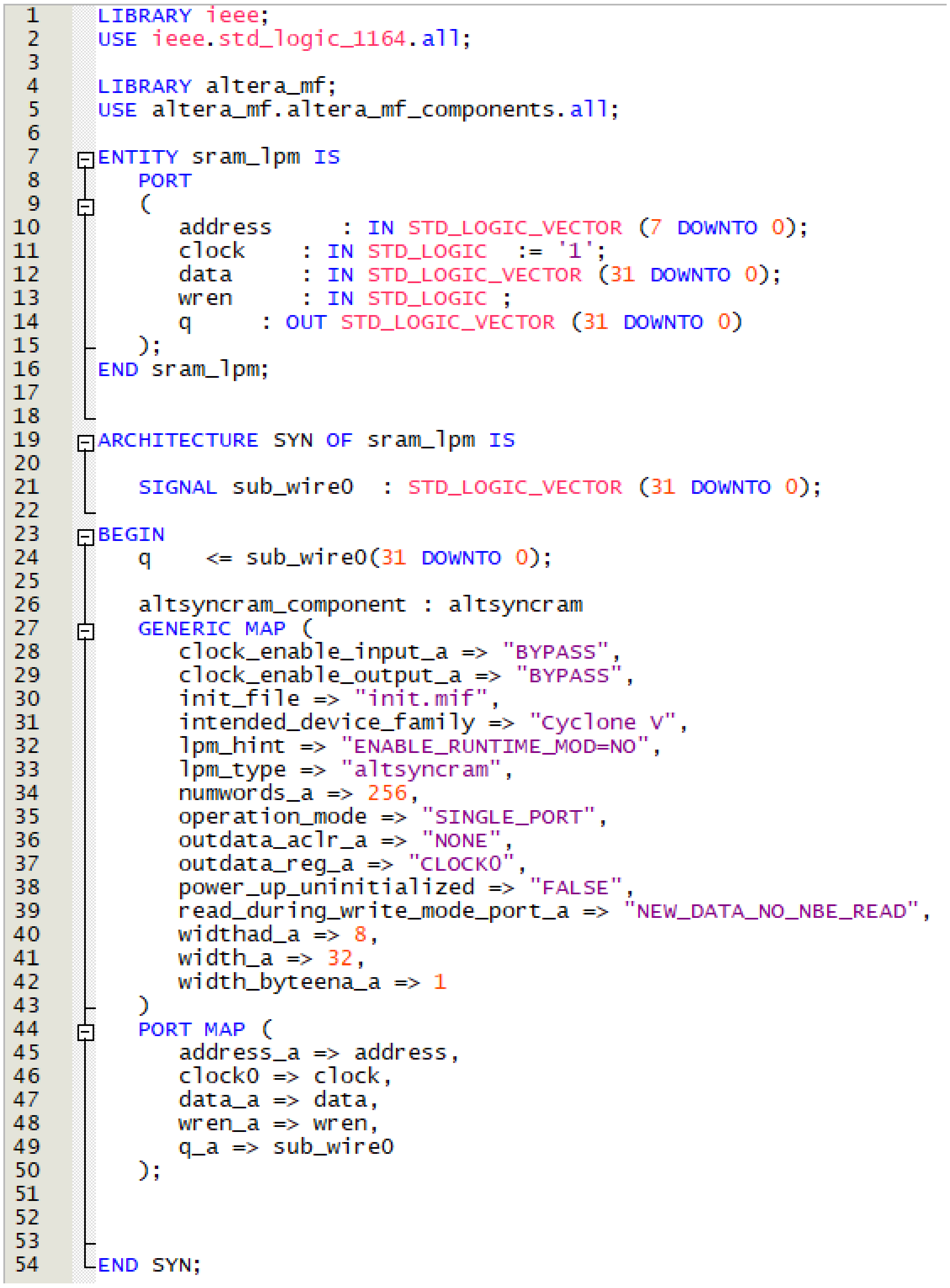
```
LOAD_cmd : in std_logic;
RA : out std_logic_vector(31 downto 0 )
);
end component;
component Multiplier_Result
port (
reset : in std_logic;
clk : in std_logic;
B_in : in std_logic_vector(31 downto 0 );
LOAD_cmd : in std_logic;
SHIFT_cmd : in std_logic;
ADD_cmd : in std_logic;
Add_out : in std_logic_vector(31 downto 0 );
C_out : in std_logic;
RC : out std_logic_vector(31 downto 0 );
LSB : out std_logic;
RB : out std_logic_vector(31 downto 0 )
);
end component;
begin
inst_RCA: RCA
port map (
RA => RA(31 downto 0),
RB => RB(31 downto 0),
C_out => C_out,
Add_out => Add_out(31 downto 0)
);
inst_Controller: Controller
port map (
reset => reset,
clk => clk,
START => START,
LSB => LSB,
ADD_cmd => ADD_cmd,
SHIFT_cmd => SHIFT_cmd,
LOAD_cmd => LOAD_cmd,
STOP => STOP
);
inst_Multiplicand: Multiplicand
port map (
reset => reset,
A_in => A_in(31 downto 0),
LOAD_cmd => LOAD_cmd,
RA => RA(31 downto 0)
);
inst_Multiplier_Result: Multiplier_Result
port map (
reset => reset,
clk => clk,
B_in => B_in(31 downto 0),
LOAD_cmd => LOAD_cmd,
SHIFT_cmd => SHIFT_cmd,
ADD_cmd => ADD_cmd,
Add_out => Add_out(31 downto 0),
C_out => C_out,
RC => RC(31 downto 0),
LSB => LSB,
RB => RB(31 downto 0)
);
end rtl;
```

Figure 10: Image of the 32-bit Multiplier VHDL code.



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

LIBRARY lpm;
USE lpm.all;

ENTITY addsub_lpm IS
    PORT
    (
        add_sub     : IN STD_LOGIC ;
        dataa     : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        datab     : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        overflow    : OUT STD_LOGIC ;
        result      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END addsub_lpm;


ARCHITECTURE SYN OF addsub_lpm IS

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



    COMPONENT lpm_add_sub
    GENERIC (
        lpm_direction     : STRING;
        lpm_hint    : STRING;
        lpm_representation      : STRING;
        lpm_type    : STRING;
        lpm_width       : NATURAL
    );
    PORT (
            add_sub  : IN STD_LOGIC ;
            dataa : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            datab : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
            overflow : OUT STD_LOGIC ;
            result   : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
    END COMPONENT;

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

    LPM_ADD_SUB_component : LPM_ADD_SUB
    GENERIC MAP (
        lpm_direction => "UNUSED",
        lpm_hint => "ONE_INPUT_IS_CONSTANT=NO,CIN_USED=NO",
        lpm_representation => "UNSIGNED",
        lpm_type => "LPM_ADD_SUB",
        lpm_width => 32
    )
    PORT MAP (
        add_sub => add_sub,
        dataa => dataa,
        datab => datab,
        overflow => sub_wire0,
        result => sub_wire1
    );



END SYN;

```

Figure 11: Image of the ADDSUB LPM VHDL code.



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

LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

ENTITY sram_lpm IS
    PORT
    (
        address     : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        clock     : IN STD_LOGIC   := '1';
        data      : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        wren      : IN STD_LOGIC ;
        q       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END sram_lpm;


ARCHITECTURE SYN OF sram_lpm IS

    SIGNAL sub_wire0  : STD_LOGIC_VECTOR (31 DOWNTO 0);

BEGIN
    q    <= sub_wire0(31 DOWNTO 0);

    altsyncram_component : altsyncram
    GENERIC MAP (
        clock_enable_input_a => "BYPASS",
        clock_enable_output_a => "BYPASS",
        init_file => "init.mif",
        intended_device_family => "Cyclone V",
        lpm_hint => "ENABLE_RUNTIME_MOD=NO",
        lpm_type => "altsyncram",
        numwords_a => 256,
        operation_mode => "SINGLE_PORT",
        outdata_aclr_a => "NONE",
        outdata_reg_a => "CLOCK0",
        power_up_uninitialized => "FALSE",
        read_during_write_mode_port_a => "NEW_DATA_NO_NBE_READ",
        widthad_a => 8,
        width_a => 32,
        width_byteena_a => 1
    )
    PORT MAP (
        address_a => address,
        clock0 => clock,
        data_a => data,
        wren_a => wren,
        q_a => sub_wire0
    );



END SYN;
```

Figure 12: Image of the SRAM LPM VHDL code.

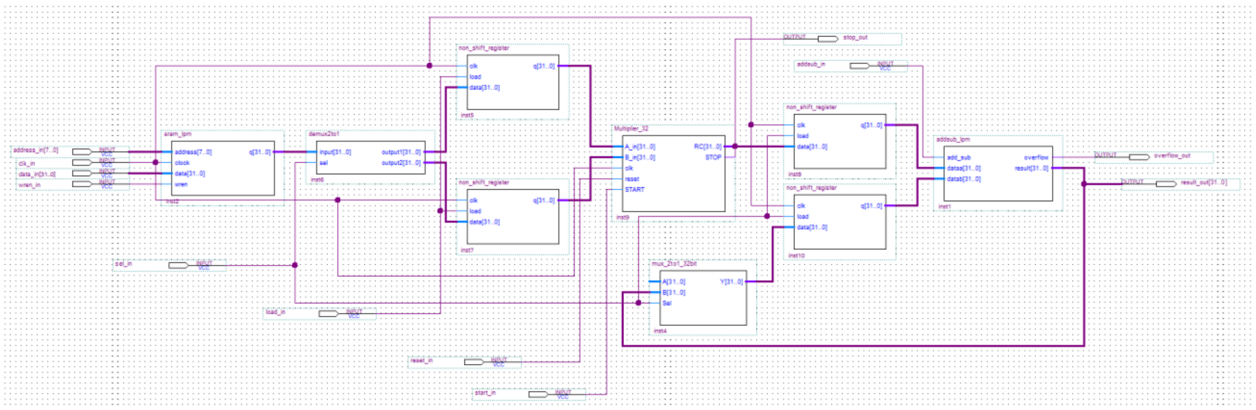


Figure 13: Image of the BDF file showing the digital circuit connections between the SRAM, registers, ADDSUB unit, multiplier unit, multiplexer, and demultiplexer.

The objective of the circuit is to compute the dot product of a vector which is a way to combine two vectors into a single number. This number is calculated by multiplying each pair of corresponding elements from the two vectors and summing up the products. This is why it can be read as the summation of xk, yk from starting from k = 1 to k = 3. To do this in VHDL, we would initialize a SRAM with 6 addresses each with their own 32-bit integer to mimic the 6 different values we would be using in the dot product. These numbers are written into the multiplier unit to find the product which is then transported to a register that's connected to a add/sub unit. We then take the sum of the product with the existing number in the add/sub unit (initialized to 0) to accumulate the products of each element of the vector. If we do this 3 times with new addresses it's the same as computing the summation which allows us to find the dot product.

With this being said, I understood how the circuit should work but could not write a working testbench. Which is why there is no simulation in the report.

Conclusion:

Through the process of designing a processor to compute the dot product of two vectors using VHDL, I gained significant insights into digital circuit design principles, VHDL programming, system integration, problem-solving, and simulation techniques. Despite encountering challenges that led to an incomplete project, such as time constraints or technical hurdles, the experience was invaluable. Integrating components like SRAM, multiplexers, add/sub units, and multipliers highlighted the importance of meticulous design and thorough testing. While the project's completion remained elusive, the knowledge and skills acquired are applicable across various digital system design contexts and FPGA-based development endeavors.