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.

```
library ieee;
 2
      use IEEE.STD_LOGIC_1164.ALL;
    曰entity mux_2to1_32bit is
曰 Port ( A · in fin
 3
 4
 5
          6
 7
                 Sel : in STD_LOGIC;
                 Y : out std_logic_vector(31 downto 0));
 8
 9
      end mux_2to1_32bit;
10
      architecture Behavioral of mux_2to1_32bit is
11
    口 begin pr
12
13
          process (A, B, Sel)
14
          begin
15
    中山
              if (Sel = '0') then
16
                  Y <= A;
17
18
                  Y <= B;
19
              end if;
20
          end process;
     Lend Behavioral;
21
22
```

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

```
library ieee;
 2
      use ieee.std_logic_1164.all;
 3
    pentity demux2to1 is
 4
 5
          port (
 6
               input : in std_logic_vector(31 downto 0);
 7
                     : in std_logic;
 8
               output1 : out std_logic_vector(31 downto 0);
 9
               output2 : out std_logic_vector(31 downto 0)
10
          );
11
      end demux2to1;
12
      architecture Behavioral of demux2to1 is
13
     □begin
14
15
           process (input, sel)
16
          begin
               if sel = '0' then
17
18
                   output1 <= input;
19
                   output2 <= (others => '0');
     占
               elsif sel = '1' then
20
21
                   output1 <= (others => '0');
22
                   output2 <= input;
     占
23
               else
24
                   output1 <= (others => '0');
                   output2 <= (others => '0');
25
26
               end if:
27
          end process;
28
     Lend Behavioral;
29
```

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

```
library ieee;
 1
 2
       use ieee.std_logic_1164.all;
 3
     pentity non_shift_register is
 4
 5
            port (
                       : in std_logic;
: in std_logic;
: in std_logic_vector(31 downto 0);
                 c1k
 6
 7
                 load
 8
 9
                         : out std_logic_vector(31 downto 0)
10
       end non_shift_register;
11
12

    Tarchitecture Behavioral of non_shift_register is
    signal reg data : ctd last.
13
            signal reg_data : std_logic_vector(31 downto 0);
14
     □begin
15
16
            process(clk)
     中
17
            begin
     日日
18
                 if rising_edge(clk) then
                      if load = '1' then
19
20
                           reg_data <= data;
21
                      end if;
22
                 end if;
23
            end process;
24
25
            q <= reg_data;
26
      Lend Behavioral;
27
```

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

```
2
      -- Library Name : DSD
 3
      -- Unit Name : Multiplier_Result
 4
 5
      -- Date: Mon Oct 27 14:13:51 2003
 6
 7
      -- Author :
 8
 9
      -- Description : Multiplier_Result performs the
10
      -- following:
11
      -- > loads B_in into register upon
12
      -- receiving LOAD_cmd
      -- > loads Adder output into register
13
14
      -- upon receiving ADD_cmd
15
      -- > shifts register right upon
16
      -- receiving SHIFT_cmd
17
18
      library ieee;
19
      use ieee.std_logic_1164.all;
20
     entity Multiplier_Result is
     ipport (reset : in std_logic ; clk : in std_logic ;
21
22
23
      B_in : in std_logic_vector (31 downto 0);
      LOAD_cmd : in std_logic ;
24
25
      SHIFT_cmd : in std_logic ;
26
      ADD_cmd : in std_logic ;
27
      Add_out : in std_logic_vector (31 downto 0);
28
      C_out : in std_logic ;
29
      RC : out std_logic_vector (31 downto 0);
30
      LSB : out std_logic ;
     LRB : out std_logic_vector (31 downto 0));
end;
31
32
     parchitecture rtl of Multiplier_Result is
33
34
      signal temp_register : std_logic_vector(64 downto 0);
     Lsignal temp_Add : std_logic;
35
36
     ⊟begin
     process (clk, reset)
37
38
     begin
     pif reset='0' then
39
      temp_register <= (others =>'0'); -- initialize temporary register
40
41
     _temp_Add <= '0';
42
     elsif (clk'event and clk='1') then
     if LOAD_cmd = '1' then
43
44
      temp_register (64 downto 32) <= (others => '0');
      temp_register(31 downto 0) <= B_in; -- load B_in into register
45
     end if;
46
    if ADD_cmd = '1' then temp_Add <= '1';
47
48
49
      end if:
50
     if SHIFT_cmd = '1' then
     古if temp_Add = '1' then
51
      -- store adder output while shifting register right 1 bit
52
53
      temp_Add <= '0';
     temp_register <= '0' & C_out & Add_out & temp_register (31 downto 1);
54
    pelse
55
56
       -- no add - simply shift right 1 bit
57
      temp_register <= '0' & temp_register (64 downto 1);
58
      end if;
59
     end if:
60
     end if;
61
      end process;
62
      RB <= temp_register(63 downto 32);</pre>
63
      LSB <= temp_register(0);
     LRC <= temp_register(31 downto 0);
64
65
      end rtl;
```

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

```
1
     □--
 2
       -- Library Name : DSD
 3
       -- Unit Name : Multiplicand
 4
 5
       -- Date: Mon Oct 27 13:32:59 2003
 6
       -- Author :
 8
       -- Description : Multiplicand is an 8-bit register
 9
       -- that is loaded when the LOAD_cmd is
10
11
       -- received and cleared with reset.
12
       library ieee;
use ieee.std_logic_1164.all;
13
14
     pentity Multiplicand is
15
     Eport (reset : in std_logic ;
   A_in : in std_logic_vector (31 downto 0);
   LOAD_cmd : in std_logic ;
16
17
18
      LRA : out std_logic_vector (31 downto 0));
end;
19
20
     parchitecture struc of Multiplicand is
21
     ☆component dflipflop
22
23
     port (
24
       reset : in std_logic;
25
       clk : in std_logic;
       D : in std_logic;
26
27
       Q : out std_logic
28
      ⊦);
29
      -end component;
30
      begin
     dflipflops: for i in 31 downto 0 generate
31
32
       dflipflopReg:dflipflop port map (reset, LOAD_cmd, A_in(i), RA(i));
33
      end generate;
end struc;
34
35
```

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

```
☐-- Date : Mon Oct 27 13:32:59 2003
 2
 3
      -- Author :
 4
 5
      -- Description : DFF is an active high D flip flop
 6
      -- with asynchronous clear.
 7
 8
      library ieee;
      use ieee.std_logic_1164.all;
 9
    entity dflipflop is
10
    bport (reset : in std_logic ;
11
      clk : in std_logic ;
12
13
      D : in std_logic ;
     -Q : out std_logic);
end;
14
15
      architecture behav of dflipflop is
16
    □begin
17
    白process (clk, reset)
18
19
     begin
    自if reset='0' then
20
     Q <= '0'; -- clear register
21
    elsif (clk'event and clk='1') then
22
23
      Q <= D; -- load register
24
     -end if;
25
     end process;
     Lend behav;
26
27
```

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

```
-- Library Name : DSD
         -- Unit Náme : Controller
 4
 5
         -- Date : Mon Oct 27 12:36:47 2003
 6
7
         -- Author :
 8
         -- Description : Controller is a finite state machine
 Q.
         -- that performs the following in each
10
11
         -- state:
12
         -- IDLE > samples the START signal
13
         -- INIT > commands the registers to be
         -- loaded
         -- TEST > samples the LSB
-- ADD > indicates the Add result to be stored
15
         -- SHIFT > commands the register to be shifted
17
18
        library ieee;
use ieee.std_logic_1164.all;
19
20
         use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
21
22
23
       pentity Controller is
       clk: in std_logic;

start: in std_logic;

start: in std_logic;

LSB: in std_logic;
24
26
27
         ADD_cmd : out std_logic ;
SHIFT_cmd : out std_logic ;
28
29
30
         LOAD_cmd : out std_logic ;
        -STOP : out std_logic);
end;
31
32
33
       parchitecture rtl of Controller is
34
        signal temp_count : std_logic_vector(4 downto 0);
-- declare states
35
36
       type state_typ is (IDLE, INIT, TEST, ADD, SHIFT);
signal state : state_typ;
37
38
       口begin
白process (clk, reset)
39
40
41
        begin
       ☆if reset='0' then
42
      ☐ IT 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:
43
44
45
47
48
49
50
51
        state <= IDLE;
52
        end if;
53
         when INIT =>
      when TEST =>
Dif LSB = '0' then
-state <= SHIFT;
Delse
54
55
57
58
59
        state <= ADD:
60
         end if:
61
         when ADD =>
62
         state <= SHIFT;
63
         when SHIFT =>
       if temp_count = "11111" then -- verify if finished temp_count <= "00000"; -- re-initialize counter
64
65
        -state <= IDLE; -- ready for next multiply
66
       pelse
67
68
         temp_count <= temp_count + 1; -- increment counter
69
         state <= TEST:
70
71
        -end if;
-end case;
-end if;
72
73
74
         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';
75
76
78
        Lend rt1;
79
```

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

```
1
 2
         -- Library Name : DSD
 3
         -- Unit Name : Full_Adder
 4
 5
         -- Date : Wed Sep 24 12:50:50 2003
 6
         -- Author :
 8
 9
         -- Description : Basic Full Adder Block
10
         library ieee;
use ieee.std_logic_1164.all;
11
12
      pentity Full_Adder is
port (X : in std_logic;
Y : in std_logic;
C_in : in std_logic;
Sum : out std_logic;
13
14
15
16
17
       -C_out : out std_logic);
Lend;
18
19
20
         architecture rtl of Full_Adder is
      □begin
21
22
         Sum <= X xor Y xor C_in;
       L_{C_out} \leftarrow (X \text{ and } Y) \text{ or } (X \text{ and } C_in) \text{ or } (Y \text{ and } C_in);
23
         end rtl;
24
```

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

```
-- Library Name : DSD
 3
      -- Unit Name : RCA
 4
 5
      -- Date : Wed Sep 24 12:50:50 2003
 6
 7
      -- Author :
 8
 9
      -- Description : RCA is an 8-bit ripple carry
10
      -- adder composed of 8 basic full
      -- adder blocks.
11
12
      library ieee;
use ieee.std_logic_1164.all;
13
14
     □entity RCA is
□port (RA : in std_logic_vector (31 downto 0);
15
16
17
      RB : in std_logic_vector (31 downto 0);
      C_out : out std_logic ;
18
19
      -Add_out : out std_logic_vector (31 downto 0));
     Lend;
20
21
     □architecture rtl of RCA is
22
     signal c_temp : std_logic_vector(31 downto 0);
     pcomponent Full_Adder
23
24
     古port (
|X : in std_logic;
25
      Y : in std_logic;
26
27
      C_in : in std_logic;
28
      Sum : out std_logic;
29
      C_out : out std_logic
30
     -);
31
      -end component;
32
      begin
      c_{temp(0)} \leftarrow 0'; -- carry in of RCA is 0
33
     Adders: for i in 31 downto 0 generate
34
35
      -- assemble first 31 adders from 0 to 31
36

<u>⊢</u>Low: if i/=31 generate

37
      FA:Full_Adder port map (RA(i), RB(i), c_temp(i), Add_out(i), c_temp(i+1));
38
     end generate;
39
      -- assemble last adder
     ⊟High: if i=31 generate
40
      FA:Full_Adder port map (RA(31), RB(31), c_temp(i), Add_out(31), C_out);
41
     -end generate;
-end generate;
-end rtl;
42
43
44
45
```

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

```
2
      -- Library Name : DSD
 3
      -- Unit Name : Multiplier
 4
 5
      -- Description : Complete multiplier
 6
 7
       library ieee;
 8
       use ieee.std_logic_1164.all;
     B--library symplify; -- required for synthesis
L--use symplify.attributes.all; -- required for synthesis
 9
10
11
     ⊟entity Multiplier_32 is
12
     ⊟port (
13
       A_in : in std_logic_vector(31 downto 0 );
14
       B_in : in std_logic_vector(31 downto 0 );
15
       clk : in std_logic;
16
       reset : in std_logic;
17
       START : in std_logic;
18
       RC : out std_logic_vector(31 downto 0 );
19
      STOP : out std_logic);
20
     Lend Multiplier_32;
21
       use work.all;
22
     □architecture rtl of Multiplier_32 is
23
       signal ADD_cmd : std_logic;
       signal Add_out : std_logic_vector(31 downto 0 );
24
25
       signal C_out : std_logic;
      signal LOAD_cmd : std_logic;
signal LSB : std_logic;
26
27
      signal RA : std_logic_vector(31 downto 0 );
signal RB : std_logic_vector(31 downto 0 );
28
29
30
       signal SHIFT_cmd : std_logic;
31
     ☆component RCA
32
     port (
33
       RA : in std_logic_vector(31 downto 0 );
34
       RB : in std_logic_vector(31 downto 0 );
       C_out : out std_logic;
35
36
      Add_out : out std_logic_vector(31 downto 0 )
      -);
-end component;
37
38
     component Controller
39
40
     ⊟port (
41
      reset : in std_logic;
42
       clk : in std_logic;
43
       START : in std_logic;
       LSB : in std_logic;
44
45
      ADD_cmd : out std_logic;
46
       SHIFT_cmd : out std_logic:
       LOAD_cmd : out std_logic;
47
48
       STOP : out std_logic
49
      ⊢);
50
      -end component;
     ☆component Multiplicand
51
52
     白port (
53
      reset : in std_logic;
54
       A_in : in std_logic_vector(31 downto 0 );
55
       LOAD_cmd : in std_logic;
56
       RA : out std_logic_vector(31 downto 0 )
     -);
57
58
      -end component;
     ☆component Multiplier_Result
59
60
     ☆port (
      reset : in std_logic;
61
62
      clk : in std_logic;
```

```
55
        LOAD_cmd : in std_logic;
 56
        RA : out std_logic_vector(31 downto 0 )
 57
 58
       end component:
 59
      内component Multiplier_Result
      ⊟port (
 60
 61
        reset : in std_logic;
 62
        clk : in std_logic;
 63
        B_in : in std_logic_vector(31 downto 0 );
 64
        LOAD_cmd : in std_logic;
 65
        SHIFT_cmd : in std_logic;
 66
        ADD_cmd : in std_logic;
 67
        Add_out : in std_logic_vector(31 downto 0 );
 68
        C_out : in std_logic;
 69
        RC : out std_logic_vector(31 downto 0 );
 70
        LSB : out std_logic;
 71
        RB : out std_logic_vector(31 downto 0 )
 72
       -);
 73
       -end component;
 74
        begin
 75
        inst_RCA: RCA
 76
      ☆port map (
 77
        RA \Rightarrow RA(31 \text{ downto } 0),
 78
        RB => RB(31 downto 0),
        C_out => C_out,
 79
 80
        Add_out => Add_out(31 downto 0)
 81
 82
        inst_Controller: Controller
      ⊟port map (
 83
 84
        reset => reset,
 85
        clk => clk,
 86
        START => START,
 87
        LSB => LSB,
 88
        ADD_cmd => ADD_cmd,
 89
        SHIFT_cmd => SHIFT_cmd,
 90
        LOAD_cmd => LOAD_cmd,
 91
        STOP => STOP
 92
 93
        inst_Multiplicand: Multiplicand
 94
      port map (
 95
        reset => reset,
 96
        A_in => A_in(31 downto 0),
 97
        LOAD_cmd => LOAD_cmd,
 98
        RA \Rightarrow RA(31 \text{ downto } 0)
 99
       -):
        inst_Multiplier_Result: Multiplier_Result
100
      ⊟port map (
101
102
        reset => reset,
103
        clk => clk,
104
        B_{in} => B_{in}(31 \text{ downto } 0),
105
        LOAD_cmd => LOAD_cmd,
106
        SHIFT_cmd => SHIFT_cmd,
107
        ADD_cmd => ADD_cmd,
108
        Add_out => Add_out(31 downto 0),
109
        C_out => C_out,
110
        RC \Rightarrow RC(31 \text{ downto } 0),
111
        LSB => LSB,
112
        RB \Rightarrow RB(31 \text{ downto } 0)
113
       -);
        end rtl;
114
115
116
```

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

```
LIBRARY ieee;
 2
       USE ieee.std_logic_1164.all;
 3
 4
       LIBRARY 1pm;
 5
       USE lpm.all;
 6
     □ENTITY addsub_lpm IS
 7
 8
          PORT
 9
          (
     白
10
             add_sub
                         : IN STD_LOGIC ;
11
                       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
12
                       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
             datab
13
             overflow : OUT STD_LOGIC ;
                         : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
14
             result
15
          );
16
       END addsub_lpm;
17
18
19
     ARCHITECTURE SYN OF addsub_lpm IS
20
21
          SIGNAL sub_wire0 : STD_LOGIC :
22
          SIGNAL sub_wire1 : STD_LOGIC_VECTOR (31 DOWNTO 0);
23
24
25
26
          COMPONENT 1pm_add_sub
27
          GENERIC (
             1pm_direction
28
                               : STRING;
29
             lpm_hint : STRING;
30
             lpm_representation
                                        : STRING:
31
             lpm_type : STRING;
32
             lpm_width
                              : NATURAL
33
          );
     中
34
          PORT (
35
                 add_sub : IN STD_LOGIC ;
36
                 dataa : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
37
                 datab : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
                overflow : OUT STD_LOGIC ;
result : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
38
39
40
          END COMPONENT;
41
42
43
       BEGIN
          overflow
44
                     <= sub_wire0;
45
                    <= sub_wire1(31 DOWNTO 0);
          result
46
47
          LPM_ADD_SUB_component : LPM_ADD_SUB
48
          GENERIC MAP (
             lpm_direction => "UNUSED",
lpm_hint => "ONE_INPUT_IS_CONSTANT=NO,CIN_USED=NO",
lpm_representation => "UNSIGNED",
49
50
51
             lpm_type => "LPM_ADD_SUB",
52
             1pm_width => 32
53
54
55
     白
          PORT MAP (
             add_sub => add_sub,
56
57
             dataa => dataa,
58
             datab => datab,
             overflow => sub_wire0,
59
60
             result => sub_wire1
          ):
61
62
63
64
65
     LEND SYN;
66
```

Figure 11: Image of the ADDSUB LPM VHDL code.

```
LIBRARY ieee:
       USE ieee std_logic_1164.all;
 2
 3
 4
       LIBRARY altera_mf;
 5
       USE altera_mf.altera_mf_components.all;
 6
     □ENTITY sram_lpm IS
 7
 8
          PORT
 9
          (
     ▤
10
             address
                          : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                       : IN STD_LOGIC := '1';
: IN STD_LOGIC_VECTOR (31 DOWNTO 0);
11
12
                        : IN STD_LOGIC ;
13
             wren
14
                    : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
15
16
      END sram_lpm;
17
18
     □ARCHITECTURE SYN OF sram_lpm IS
19
20
21
          SIGNAL sub_wire0 : STD_LOGIC_VECTOR (31 DOWNTO 0);
22
     □BEGIN
23
24
                <= sub_wire0(31 DOWNTO 0);
25
26
          altsyncram_component : altsyncram
27
          GENERIC MAP (
28
             clock_enable_input_a => "BYPASS"
             clock_enable_output_a => "BYPASS"
29
30
             init_file => "init.mif",
             intended_device_family => "Cyclone v",
31
             lpm_hint => "ENABLE_RUNTIME_MOD=NO",
lpm_type => "altsyncram",
32
33
             numwords_a => 256,
operation_mode => "SINGLE_PORT",
34
35
             outdata_aclr_a => "NONE"
36
             outdata_reg_a => "CLOCKO"
37
             power_up_uninitialized => "FALSE"
38
             read_during_write_mode_port_a => "NEW_DATA_NO_NBE_READ",
39
40
             widthad_a => 8,
41
             width_a \Rightarrow 32,
42
             width_byteena_a => 1
43
44
          PORT MAP (
45
             address_a => address,
46
             clock0 => clock,
             data_a => data,
47
48
             wren_a => wren,
49
             q_a => sub_wire0
50
          ):
51
52
53
54
     LEND SYN:
```

Figure 12: Image of the SRAM LPM VHDL code.

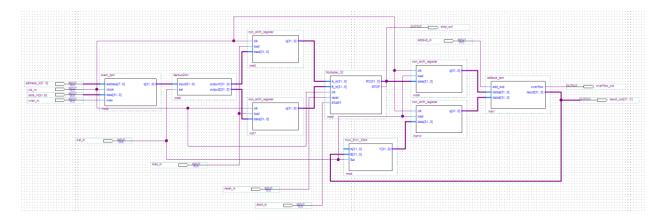


Figure 13: Image of the BDF file showing the digital circuit connections between the SRAM, registers, ADDSUB unit, multiplier unit, multiplexer, and demultiplexer.

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.