LAB Assignment 3

Multiplicative Divider

Dr. Ahmed Elhossini

Introduction

The third lab assignment will deal with multiplicative divider we discussed in lecture 7. To build such a divider circuit we first need to build an efficient multiplier then use it to build the divider. We also need to build a timing controller in VHDL which is usually implemented as a state machine.

Pre-requisites

1. Basic Knowledge of VHDL.

Objectives:

1. Implement 16 x 16-bits radix-4 booth multiplier using VHDL.
2. Simulate the multiplier using Xilinx ISIM.
3. Use the radix-4 multiplier to build a 16 / 16 bits multiplicative divider based on **Goldschmidt Algorithm.**
4. Simulate the divider using Xilinx ISIM.

Equipment and Tools

1. Xilinx ISE tool chain.

Lab Time:

3 Weeks from December 21th 2016 to January 25nd 2017.

Goldschmidt Algorithm

The **Goldschmidt Algorithm** is a division multiplicative algorithm based on the approximation of the reciprocal of the divisor. The algorithm requires the multiplication of both the dividend and the divisor with some value that tend to make the divisor approaches 1, as shown in *Figure 1*. For this reason it is important to build an efficient multiplier in order to perform an efficient division. In this lab assignment you are asked to build a multiplier based on the Radix-4 booth multiplication algorithm. This multiplier should later be used to build the **Goldschmidt Algorithm.**

Algorithm 1 Goldschmidt-Divide (A,B) – Goldschmidt's iterative algorithm for computing A/B

Require: $|e_0| < 1$

1: Initialize $N_{-1} \leftarrow A, D_{-1} \leftarrow B, F_{-1} \leftarrow (1 - e_0)/B$

2: **for** $i \leftarrow 0$ to k do

3: $N_i \leftarrow N_{i-1} * F_{i-1}$

4: $D_i \leftarrow D_{i-1} * F_{i-1}$

5: $F_i \leftarrow 2 - D_i$

6: **end for**

7: Return (N)

Figure 1 Goldschmidt Algorithm

Part 1: Radix-4 Booth multiplier

As discussed in the lecture 6, booth multiplication algorithm is used to process the multiplier bit in groups and encode thin into a non-redundant signed digit form that has less number of digits. This reduces the number of partial products generated, and reduces the circuit size of the multi-operand adder network required to generate the final product.

1. Partial Product Generation

For radix-4 booth multiplication, multiplier bits are processed in groups of 3 bits with one bit overlapping. The selection table to generate the signed digits, and the partial products, is shown in Table 1. According to that table, the multiplier bits are scanned 3 bits at a time, and according to the bit combination, the multiplicand is multiplied with the signed digit selected to generate the partial product. All the partial products can be generated using simple transfer and shifting operations. A possible partial product generation circuit is shown in Figure 2. The circuit should be repeated for each group of bits in the multiplier. Each partial product should be sign extended because it is a signed number, and two's complement bits should also be added to the partial products. This is shown in Figure 3 for 5 bits operands. For 16 bits operands, the number of partial products will be 8, as shown in *Figure 5*.

Model the partial product generation circuit using VHDL. The input to the partial product generator circuit is 3 bits forming the multiplier bits group, and the output is a 17-bit partial product.

Bits Group			Signed Digit
a_{i+1}	a_i	a_{i-1}	D_i
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

Table 1 Radix 4 booth multiplier selection table.

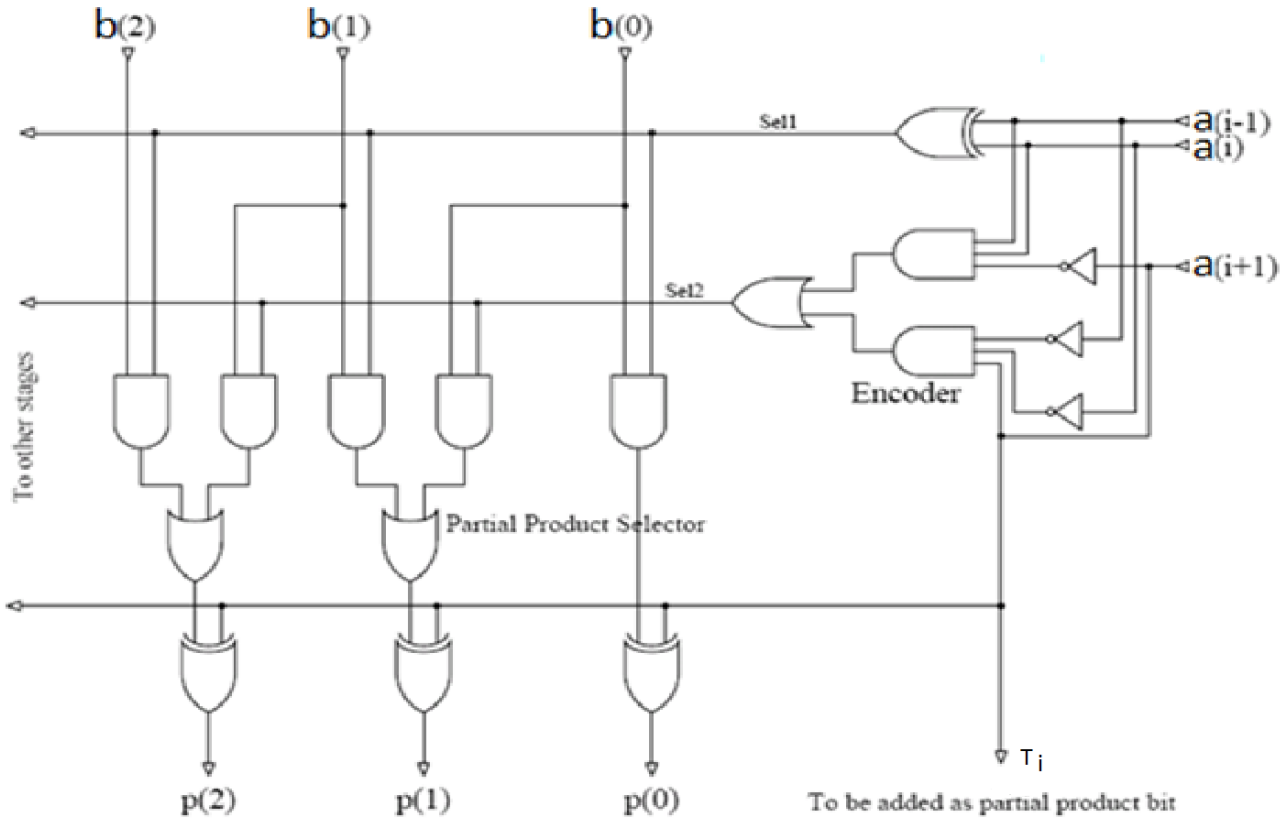

Figure 2 Radix-4 Partial Product Generator

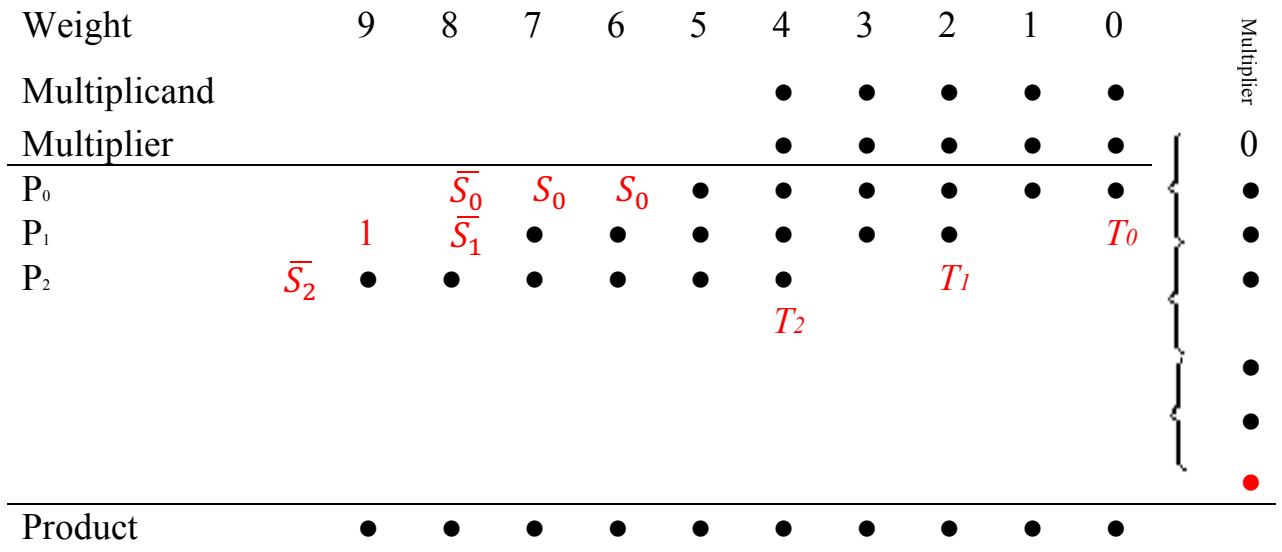

Figure 3 Dot diagram of 5 x 5 radix-4 booth multiplier with sign extension

2. Partial Products Reduction (Addition)

Carry-save addition can used to add partial products. This is shown in *Figure 5* using 4:2 counters (and mix of 3:2, 2:2 counters). As discussed in the lecture, carry-save addition can be performed in several ways. The approach presented in *Figure 5* can be used but you are free to modify it if required. All types of counters used in this scheme are already implemented in the previous Lab Assignment; reuse these components when required.

Implement the carry save adder network using VHDL.

3. Final Adder Circuit

Use any type of adder circuit to add the final two rows. The '+' operator in VHDL can be used as well. The synthesizer will generate an adder circuit depending on the number of bits in the operands. An example of 8-bits adder is shown in *Figure 4*.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_signed.ALL;
entity Add8bits is
   port ( a, b : in  STD_LOGIC_VECTOR (7 downto 0);
          ci : in  STD_LOGIC;
          s : out  STD_LOGIC_VECTOR(7 downto 0);
          co : out  STD_LOGIC);
end Add8bits;
architecture Behavioral of Add8bits is
    signal  s_t : STD_LOGIC_VECTOR(8 downto 0);
begin
      s_t <= (a(7)&a) + (b(7)&b) + ci;
     s <= s_t(7 downto 0);     co <= s_t(8);
end Behavioral;
```

Figure 4 VHDL code for 8-bits adder using the + sign operator

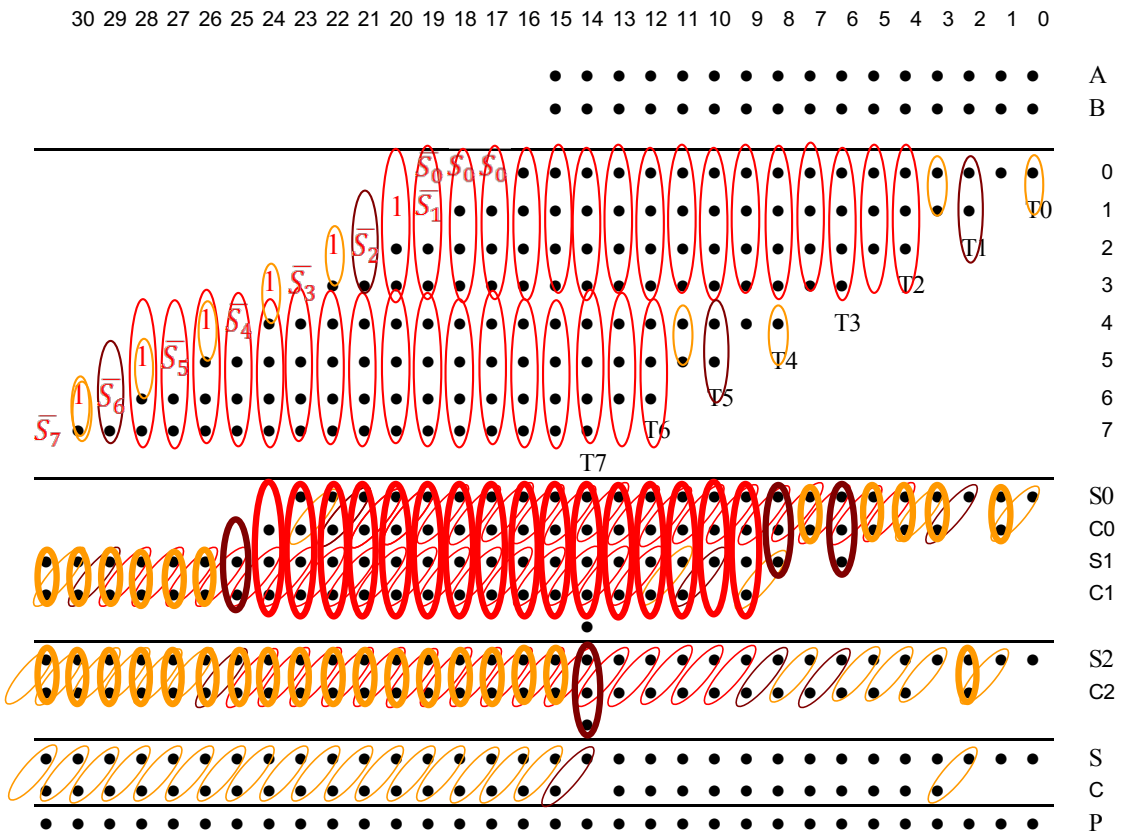

Figure 5 Dot Diagram of Radix 4-booth multiplier. 4-2 counters are used in the first two stages.

Part 2: Goldschmidt Multiplicative Divider

A block diagram of the Goldschmidt Divider is shown in Figure 6. The multiplier implemented in Part 1, can be used to implement the divider. There are three other modules need to be implemented in VHDL to complete the divider.

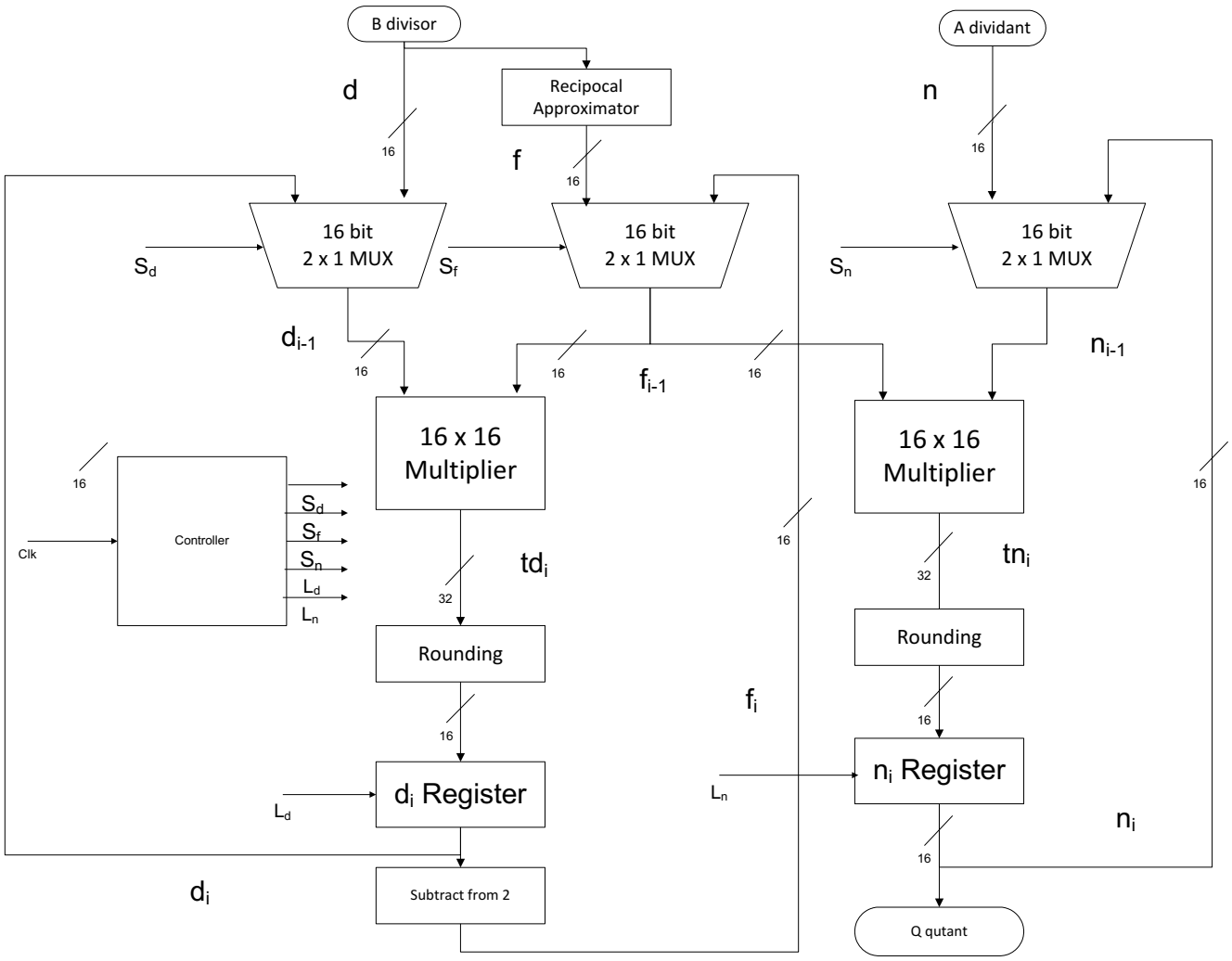

Figure 6 Goldschmidt Algorithm Implementation

1. **Reciprocal Approximate**

   The purpose of the reciprocal approximation is to generate an initial value of *F* in the algorithm shown in *Figure 1*. As discussed in the lecture, several approaches can be used to generate that initial value. Swapping the divisor bits from the most significant bit to the least significant bit is one of these approaches discussed in lecture 7 and may be used in this stage.

2. Rounding

As shown in *Figure 6*, the output of the multipliers is 32-bits. This output should be fed back to the input of the multiplier, which is only 16-bits. In the rounding stage, some of the bits of the multiplier output should be selected for the input. Depending on the fixed point fraction format, these bits may be different. If the input of the multiplier has 8-bits integer part (with sign), and 8-bits fraction, the middle 16 bits of the multiplier output can be selected in this stage.

3. Building a state-machine using VHDL

State machines are used to perform sequential operations in a digital system. You need to build a controller to manage the steps required to execute the sequence of operations required by the algorithm. *Figure 7* shows a sample state machine. This is a sample of Moore type of state machines in which the output only depends on the current state. The output is shown inside the vertex and the input is shown on the edges. VHDL process statement is used to build state-machines as shown in the code of *Figure 8*.

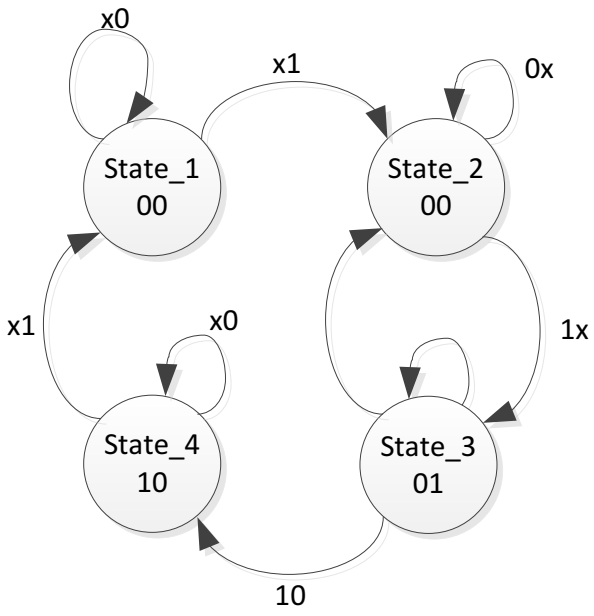

Figure 7 Sample State Machine

The code is composed of three different processes. The first process performs the transition from the current state to the next state. The second state decodes the current state to generate the output. The last state is used to decode the current state, and the current input to generate the next state. Modify this code to build the required controller.

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

entity divider_control is
    Port ( clk : in  STD_LOGIC;
           input : in  STD_LOGIC_VECTOR (1 downto 0);
           output : out  STD_LOGIC_VECTOR (1 downto 0);
           rst : in  STD_LOGIC);
end divider_control;

architecture Behavioral of divider_control is
   type state_type is (state_1, state_2, state_3, state_4);
   signal state, next_state : state_type;
   --Declare internal signals for all outputs of the state-machine
   signal output_i : STD_LOGIC_VECTOR (1 downto 0);  -- example output signal
begin
   SYNC_PROC: process (clk)
   begin
      if (clk'event and clk = '1') then
         if (rst = '1') then
            state <= state_1;
            output <= (others=>'0');
         else
            state <= next_state;
            output <= output_i;
         end if;
      end if;
   end process;

   --MOORE State-Machine - Outputs based on state only
   OUTPUT_DECODE: process (state)
   begin
      --insert statements to decode internal output signals
      case (state) is
            when state_1 => output_i <= "00";
            when state_2 => output_i <= "00";
            when state_3 => output_i <= "01";
            when state_4 => output_i <= "10";
      end case;
   end process;

   NEXT_STATE_DECODE: process (state, input)
   begin
      --declare default state for next_state to avoid latches
      next_state <= state;  --default is to stay in current state
      --insert statements to decode next_state
      case (state) is
         when state_1 =>
            if input(0) = '1' then
               next_state <= state_2;
            end if;
         when state_2 =>
            if input(1)= '1' then
                     next_state <= state_3;
                else
                     next_state <= state_1;
            end if;
         when state_3 =>
            if input(0)='0' and input(1)='1' then
                next_state <= state_4;
            end if;
         when state_4 =>
            if input(0)='1' then
                next_state <= state_1;
            end if;
      end case;
   end process;
end Behavioral;
```

Figure 8 Sample code for the state machine

The controller should control the input of the multiplexers at each step of the algorithm, and the output register to generate the final output of the divider.

Deliverables:

1. **VHDL code for all modules implemented.**
2. **Simulation results for each part (behavior and timing).**
    a. **Multiplier output for some sample input.**
    b. **Divider output for some sample input.**
3. **Frequency analysis: What is the maximum frequency of operation?.**