Lab 6: Arithmetic - Radix-4 Multiplier

Written By: Joel Bailey

Date: November 10th, 2020

ECE 524/Lab

Lab 6: Arithmetic - Radix-4 Multiplier

Table of Contents

I. Introduction

In this lab, a radix-4 multiplier is designed in VHDL. A radix-4 multiplier performs modular multiplication on a multiplicand by adding the results of each operation to its predecessor with a 2-bit shift. An example in dot notation can be found in the following diagram:

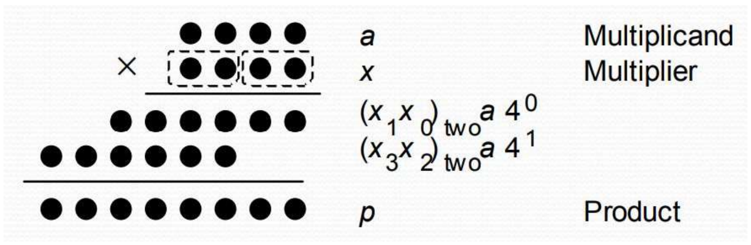

Fig. 1. Radix-4 multiplication in dot notation

This design has a fixed bit width for the multiplicand and multiplier at 10 and 6-bits, respectively. The multiplication algorithm benefits from allowing high order multiplication in a small area at the cost of time. This is converse to array multiplication wherein the area is increased but the operation is quicker.

II. Procedure

A high level overview of the design was given in the specifications of the lab, and the solution for this lab only modified it slightly. Below is a modified schematic of the radix-4 multiplier:

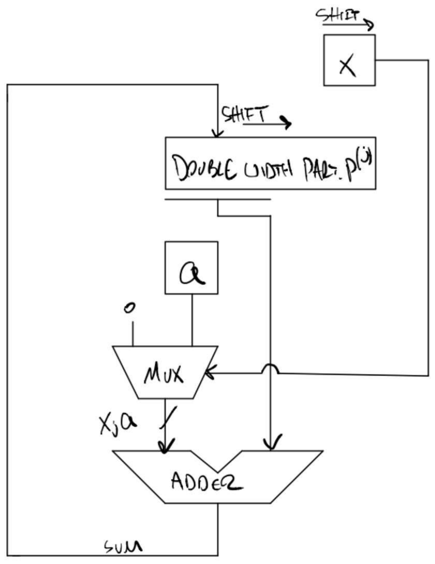

Fig. 2. Radix-4 multiplier schematic

From the schematic, the design was broken into the following four functional components:

- Multiplier, X. Clocked 6-bit shift register.
- Multiplicand, A. 10-bit Multiplexer
- Adder. 12-bit adder
- Partial product. Clocked 16-bit shift register

The 6-bit multiplier shifts the 2 least significant bits out from the multiplier and they are used as the select line for the multiplexer. The value on the select line is the multiplication coefficient that is applied to the multiplicand, A, and selects (0 * A) to (3 * A). This selected value is added to the 12 most significant bits from the partial product register and is then returned to the 12 MSB. This is then shifted 2-bits to the right and the new 12 most significant bits are read for the next iteration. A simple example of 5 times 5 is given below to show the function of the algorithm.

Example:

Decimal:	$5 * 5 = 25$
Binary:	$101 * 101 = 011001$

A = 00 0101
X = 00 0000 0101

CP = 0
Select = $A_1 A_0$ = 01
Mux_Out = X * 1 = X = 00 0000 0101
Adder = Mux_out + Part_Prod(15:4) = 00 0000 0101 + 00 0000 0000 = 00 0000 0101

CP = 2
Select = $A_3 A_2$ = 01
Mux_Out = X * 1 = X = 00 0000 0101
Adder = Mux_out + Part_Prod(15:4) = 00 0000 0101 + 00 0000 0001 = 00 0000 0110

CP = 4
Select = $A_5 A_4$ = 00
Mux_Out = X * 0 = X = 00 0000 0000
Adder = Mux_out + Part_Prod(15:4) = 00 0000 0000 + 00 0000 0001 = 00 0000 0001

Partial Product Register

CP	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0
3	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0
4	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1
5	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1

Product = 0000 0000 0001 1001 = 25

In the final design, the partial product register has 2 shifting mechanisms, a sequential shift that places the accumulated sum into position, and a combinational 2-bit shift for the operation.

```
--async reset
process(CLK, RST) begin
    if RST = '1' then
        REG <= (others=>'0');
    elsif rising_edge(CLK) then
        --Shift out the 2 LSB of REG
        REG <= DIN & REG(5 downto 2);
    end if;
end process;

--Accumulation out is shifted 2 bits to the right for the operation
ACCOUT <= std_logic_vector(unsigned(REG(15 downto 4)) srl 2);

--When output enable is high, the product is ready
with OE select
    PRODUCT <= REG when '1',
               (others=>'Z') when others;
```

This, coupled with placing the smaller bit number as the multiplier, allows the result to be available at the 3rd clock cycle after values are loaded into the circuit.

III. Testing Strategy

The design was developed for unsigned numbers, therefore, the test vectors were chosen to test the limits of the register widths, along with the multiplication operation, itself. The following are the vectors tested:

1. -- A=60 X=60, 60 * 60 = 3600

2. -- A=1023 X=63, 1023 * 63 = 64449

3. -- A=100 X=0, 100 * 0 = 0

4. -- A=500 X=1, 500 * 1 = 500

5. -- A=0 X=60, 0 * 60 = 0

IV. Results (Data)

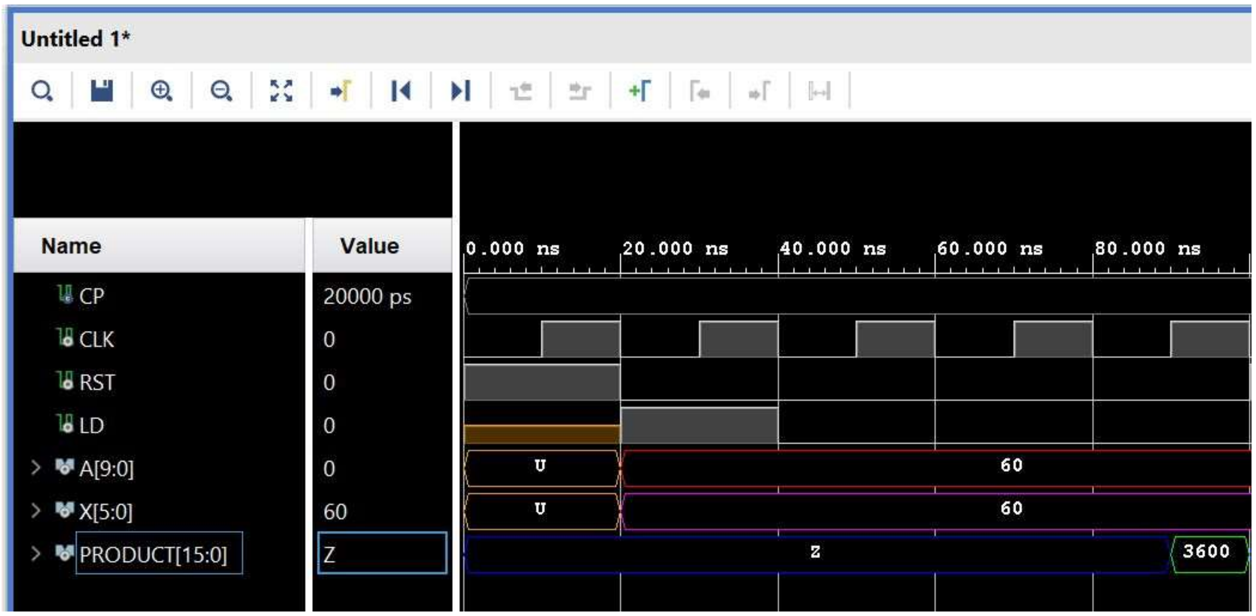

Fig. 3. Vector 1: 60 * 60 = 3600

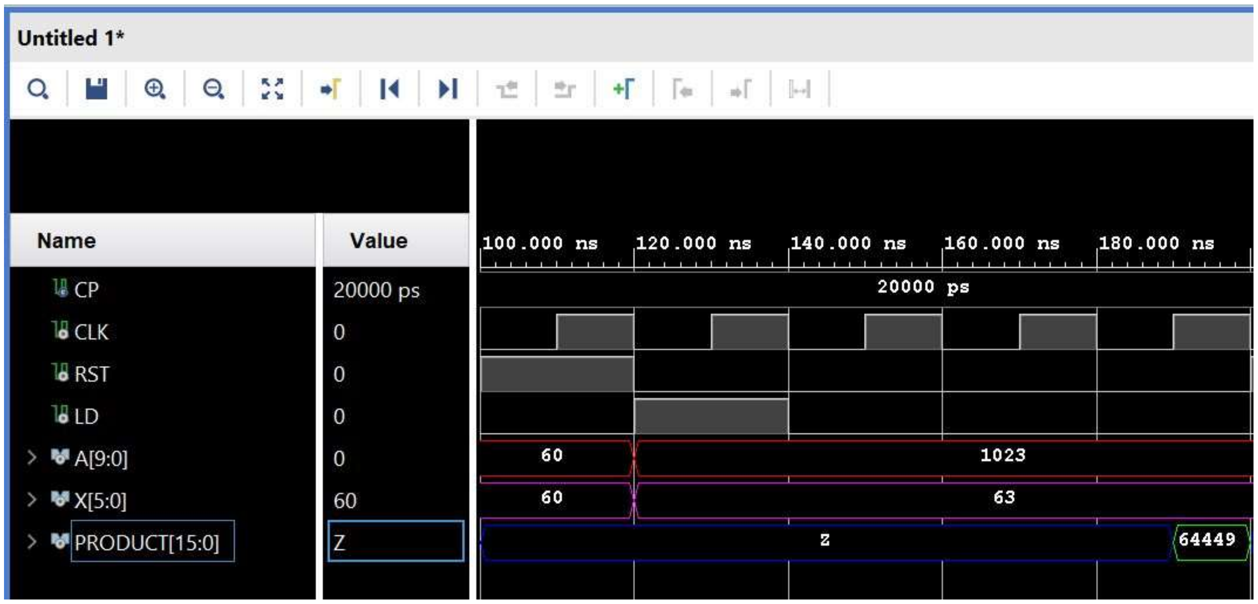

Fig. 4. Vector 2: 1023* 63 = 64449

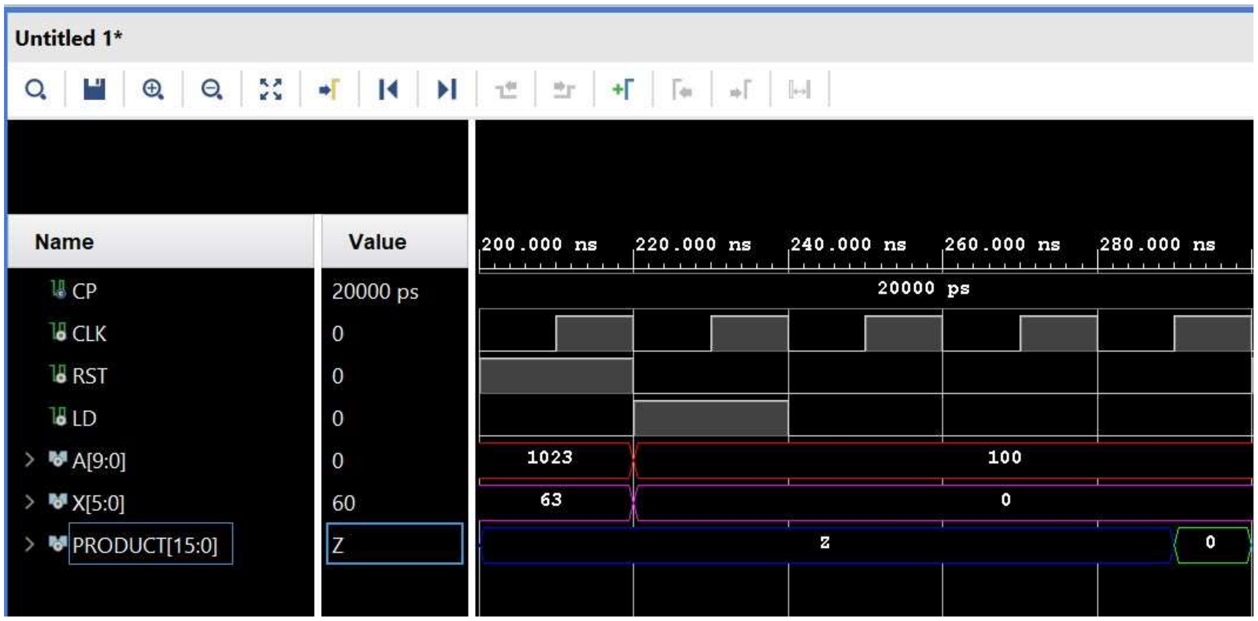

Fig. 5. Vector 3: 100 * 0 = 0

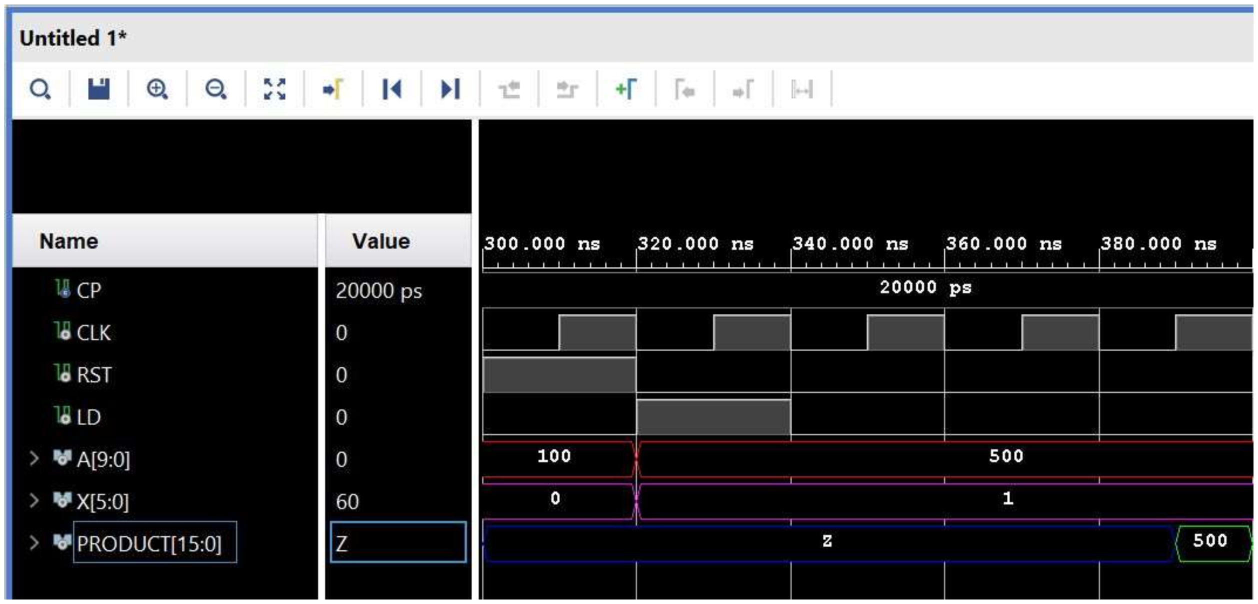

Fig. 6. Vector 4: 500 * 1 = 500

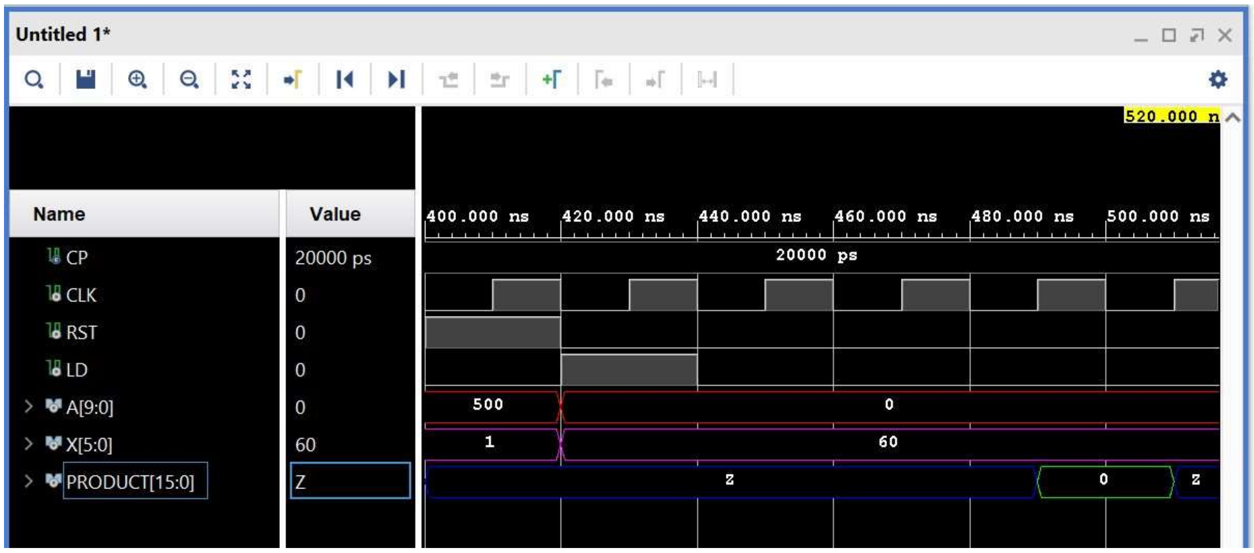

Fig. 7. Vector 5: 0 * 60 = 0

V. Analysis / Conclusion

The design was able to produce the product of a 6-bit and 10-bit number in 3 clock cycles. A value of high impedance was used at the output as an indicator that the product was not ready yet, in an actual implementation, this could be changed according to specs. I also added a control counter into the top level to enable the output and this was done post design as a tool for validation. In an actual implementation, the value would be retained for as long as the specifications required.