Experiment 2: Multiplier

Rathour Param Jitendrakumar Roll Number: 190070049
EE-214, WEL, IIT Bombay
March 10, 2021

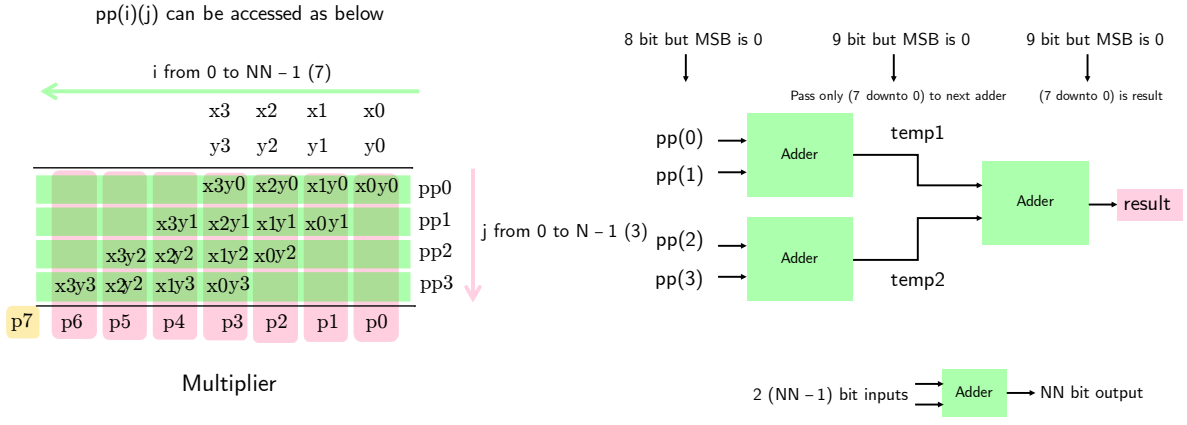
Overview of the experiment:

The purpose of this experiment is to design a combinational circuit that multiplies two 4-bit numbers. Describe it in VHDL using Behavioral modelling, simulate with the generic test-bench using Quartus and verify all test cases using scan chain.

I used the provided skeleton code, designed the circuit accordingly and verified my implementation using post-synthesis gate-level and scan chain by generating the trace file for all 256 combinations.
Subparts: generic Adder (already implemented), storing partial products in a 1D × 1D array, calculating result

This report contains my approach to the experiment, the circuit's design with the relevant code followed by the DUT input-output format, RTL netlist view, and design confirmation by output waveforms of RTL, Gate-Level Simulation and scan chain.

Approach to the experiment:



N is number of bits in input (= 4), NN is number of bits in output (= 8)

First, we define a type called pp_type to store elementwise product terms ($x_i y_i$).
This is a 1D × 1D array, basically an array of std logic vectors.
A variable pp of type pp_type will store N × NN elements. Among them, at most N × N are non-zero (figure).
Now, pp(i)(j) = x(j – i) AND y(i) (for i ∈ {0 to N – 1}, for j ∈ {i to i + N – 1})
Result = pp(0) + pp(1) + pp(2) + pp(3) = (pp(0) + pp(1)) + (pp(2) + pp(3)) = temp1 + temp2
The given adder function adds two (NN – 1) bit (= 7) inputs and outputs NN bit answer.
So, care is taken while passing arguments. It does not matter as the MSB of temp1, temp2, and result is 0.

Design document and VHDL code if relevant:

This whole design is part of TopLevel with 8 input bits & 8 output bits to confirm our design.
See DUT Input/Output Format
Code can be broken down into 2 parts

- storing partial products (used nested for loops)
- calculating result (used previously implemented adder as a function)

Entity declaration

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

entity multiplier is
    generic(
        N : integer:=4; -- operand width
        NN : integer:=8 -- result width
        );
    port (
        A: in std_logic_vector(N-1 downto 0);
        B: in std_logic_vector(N-1 downto 0);
        M: out std_logic_vector((NN)-1 downto 0)
        ) ;
end multiplier;
```

Architechture of multiplier
pp_type

```
  -- Unbounded 1D X 1D array declaration
  type pp_type is array (natural range<>) of std_logic_vector(NN-1 downto 0);
```

adder function

```
-- Adder function adds two 8-bit number.
-- Usage: var := adder(X,Y) where var is a variable and X,Y are two 8-bit inputs to be added
function adder(A: in std_logic_vector; B: in std_logic_vector)
    return std_logic_vector is
                                                                -- Variable declaration
    variable sum : std_logic_vector(NN downto 0):= (others=>'0');
    variable carry : std_logic_vector(NN downto 0):= (others=>'0');
    begin
                                                                -- Describing behaviour of adder
    for i in 0 to NN-1 loop
        sum(i) := A(i) xor B(i) xor carry(i);
        carry(i+1) := (A(i) and B(i)) or (carry(i) and (A(i) xor B(i)));
    end loop;
    sum(NN):=carry(NN);
    return sum;
end adder;
```

Multiplier process

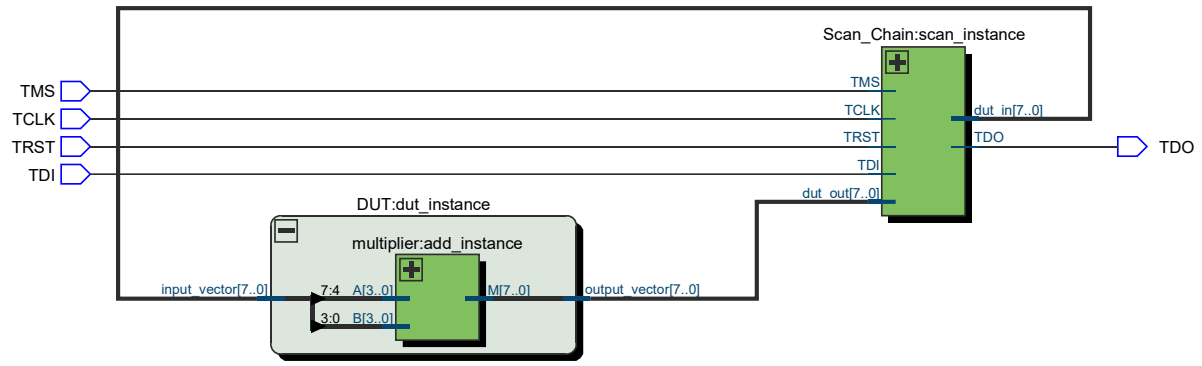
```
    begin
        multiplier : process(A, B)
        -- Declaration of 1D X 1D array to store partial products
        variable pp : pp_type(0 to 3) := (others => (others =>'0'));

        -- Declaration of summation of partial product will give multiplication result which is stored in variable, result.
        variable result : std_logic_vector(NN downto 0):= (others=>'0'); -- Stores the final result (temp1 + temp2)

        -- Temporary array is required to avoid combinaitonal loop while adding partial product
        variable temp1 : std_logic_vector(NN downto 0):= (others=>'0'); -- Stores the result of pp0 + pp1
        variable temp2 : std_logic_vector(NN downto 0):= (others=>'0'); -- Stores the result of pp2 + pp3

        begin

        for i in 0 to N-1 loop                                  -- Calculation of partial product
            for j in i to i+N-1 loop
                pp(i)(j) := A(j-i) and B(i);                    -- pp(i)(j) is A(j-i)B(i)
            end loop;
        end loop;

        temp1 := adder(pp(0),pp(1));                            -- Summation of partial product
        temp2 := adder(pp(2),pp(3));                            -- temp1 = pp0 + pp1
                                                                -- temp2 = pp2 + pp3
                                                                -- temp1,2 are 9 bit, only 8 LSB's are required (MSB is 0 (carry))
        result := adder(temp1(NN-1 downto 0),temp2(NN-1 downto 0)); -- result = temp1 + temp2
        M <= result(NN-1 downto 0);                             -- Assignment of final result (again only 8 LSB's used)
    end process ;                                               -- Multiplier
```

Python Code to generate test cases for Multiplier

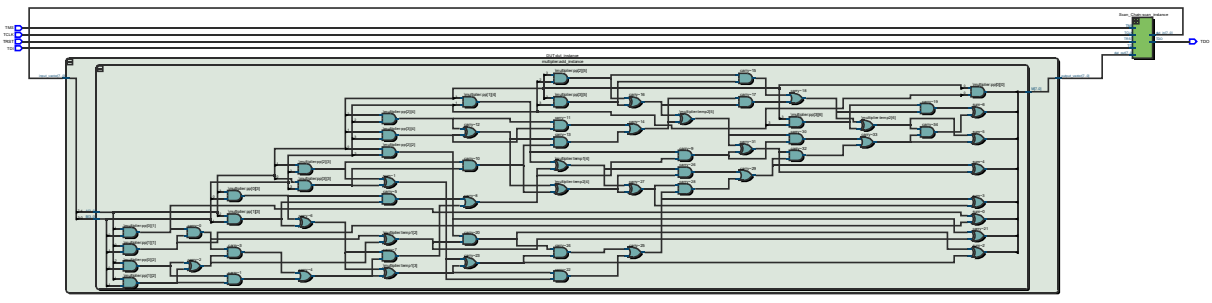
```python
import numpy as np
import sympy as sp

n = 4
Max = 2**n

def generateBinaryStrings(n, Max):
    return [bin(i)[2:].zfill(n) for i in range(Max)]

BinaryStrings = generateBinaryStrings(n, Max)

def multiplyIntegers(a,b,n, Max):
    return a * b;

numberOfOutputBits = 8

with open("Tracefile.txt", "w") as file:
    for i in range(Max):
        for j in range(Max):
            I1 = BinaryStrings[i]
            I2 = BinaryStrings[j]
            output = multiplyIntegers(i,j,n, Max)
            mask = '1'*numberOfOutputBits
            o = bin(output)[2:].zfill(numberOfOutputBits)
            str = "{}{} {} {}\n".format(I1, I2, o, mask)
            file.write(str)
```

RTL View:

Schematic



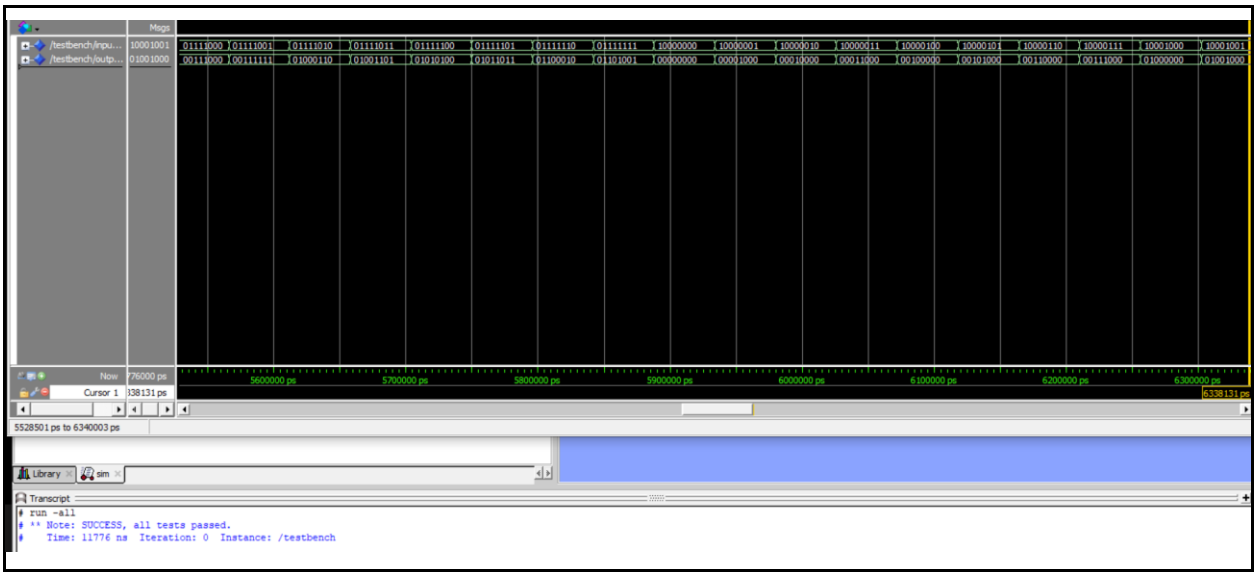
Multiplier



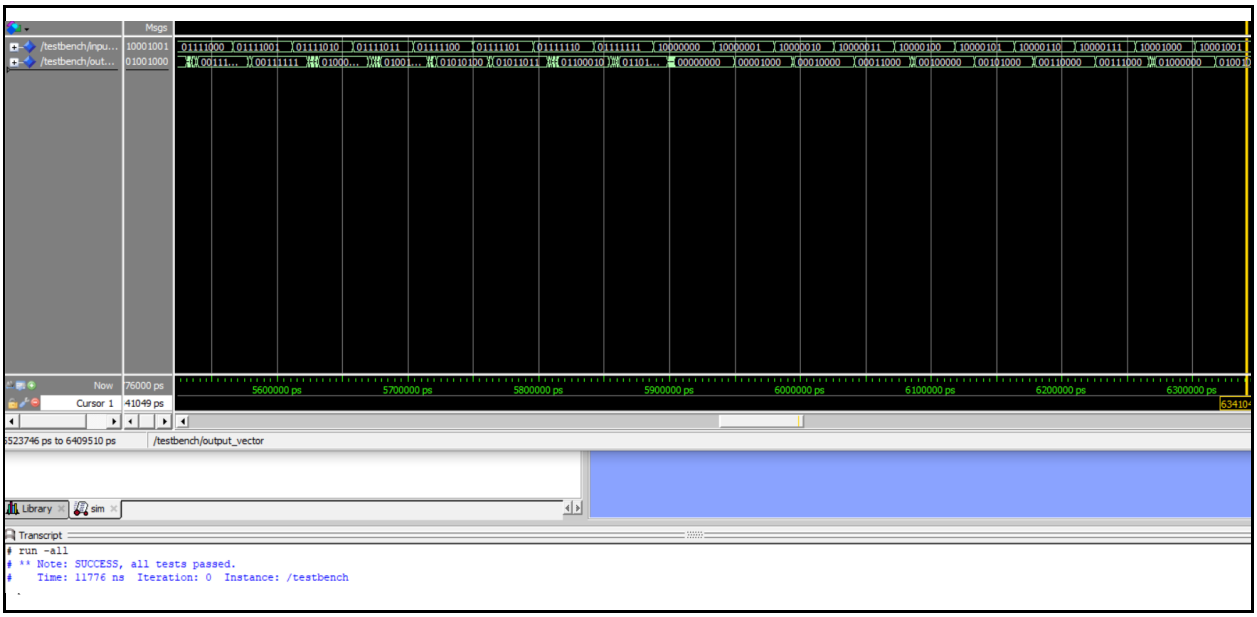
DUT Input/Output Format:

Input Format: $A_3A_2A_1A_0B_3B_2B_1B_0$ (2 4-bit numbers A (A_3 is MSB A_0 is LSB) and B (B_3 is MSB B_0 is LSB))
Output Format: $M_7M_6M_5M_4M_3M_2M_1M_0$ (8-bit number M (M_7 is MSB M_0 is LSB))
00000000 00000000 11111111 00011111 00001111 11111111 00111000 00011000 11111111 01100001 00000110 11111111 01111000 00111000 11111111 10010111 00111111 11111111 10101001 01011010 11111111 10111000 01011000 11111111 11010010 00011010 11111111 11111111 11100001 11111111

RTL Simulation:



Gate-level Simulation:



Krypton board:

Krypton part was scan chain testing only, with inputs and outputs given using Tiva-C microcontroller.
Pin assignment

Node Name	Direction	Location	I/O Bank	Fitter Location
TCLK	Input	PIN_23	1	PIN_23
TDI	Input	PIN_5	1	PIN_5
TDO	Output	PIN_3	1	PIN_3
TMS	Input	PIN_7	1	PIN_7
TRST	Input	PIN_21	1	PIN_21

Observations:

All test cases passed in RTL, Gate-Level as well as in scan chain.

Example: For input 01001101 (A = 4, B = 13), output is 00110100 (M = 4 × 13 = 52)

```
#------ Command -  508 : RUNTEST 1 MSEC ------#


#------ Command -  509 : SDR 8 TDI(FE) 8 TDO(D2) MASK(FF) ------#

Successfully entered the input..
Sampling out data..
FF
Output Comparison :  Success

#------ Command -  510 : RUNTEST 1 MSEC ------#


#------ Command -  511 : SDR 8 TDI(FF) 8 TDO(E1) MASK(FF) ------#

Successfully entered the input..
Sampling out data..
FF
Output Comparison :  Success

#------ Command -  512 : RUNTEST 1 MSEC ------#

Sampling out data..
FF
Output Comparison :  Success
OK. All Test Cases Passed.
Transaction Complete.

E:\Softwares\Simulation and Drawing\Quartus\Quartus\Week 2\Multiplier>
```

References:

Tertulien Ndjountche *Digital Electronics 2 Sequential and Arithmetic Logic Circuits* Wiley