Experiment 2: Sequential Circuits

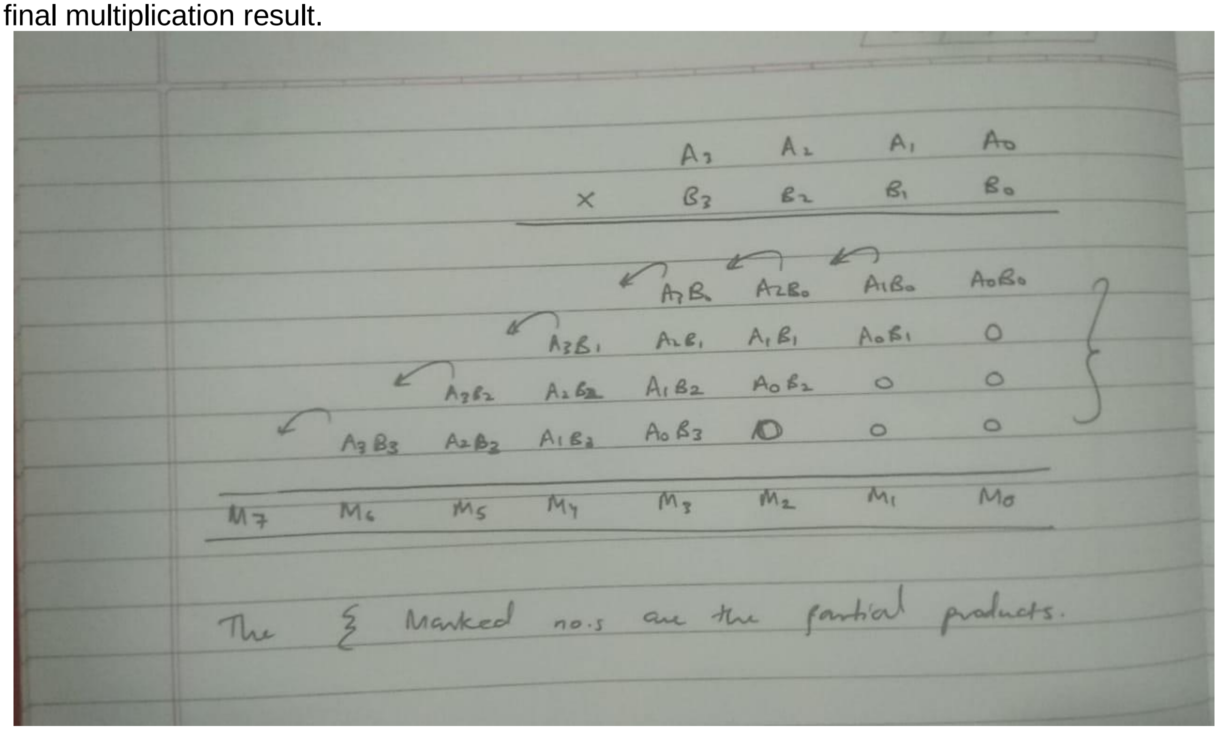
Prasann Viswanathan Roll Number 190070047
EE-214, WEL, IIT Bombay
March 9, 2021

Overview of the experiment:

The purpose of the experiment was to make a 4-bit multiplier, using behavioral style of description with the help of loops, and sequential statements.
I defined our own type of variable called pp_type and used it to store the partial products generated during multiplication. Then the given function that adds two 8 bit numbers to add all the partial products was used by me to implement addition of all the partial products, which gives the final result.
Then I finished DUT for the given experiment and added some changes to Testbench as well. I made a simple python code called test_gen.py to generate all testcases with their masks in a TRACEFILE.txt. I ran both the RTL as well as Gate level simulations to confirm whether my implementation was correct before checking on Krypton board using ScanChain. Finally, verified the correctness of my design using ScanChain. Once everything was correctly implemented I presented my work to the TA.

Approach to the experiment:

The partial products generated in 8-bit form (in variable of pp_type), are added to arrive at the final multiplication result.



Design document and VHDL code if relevant:

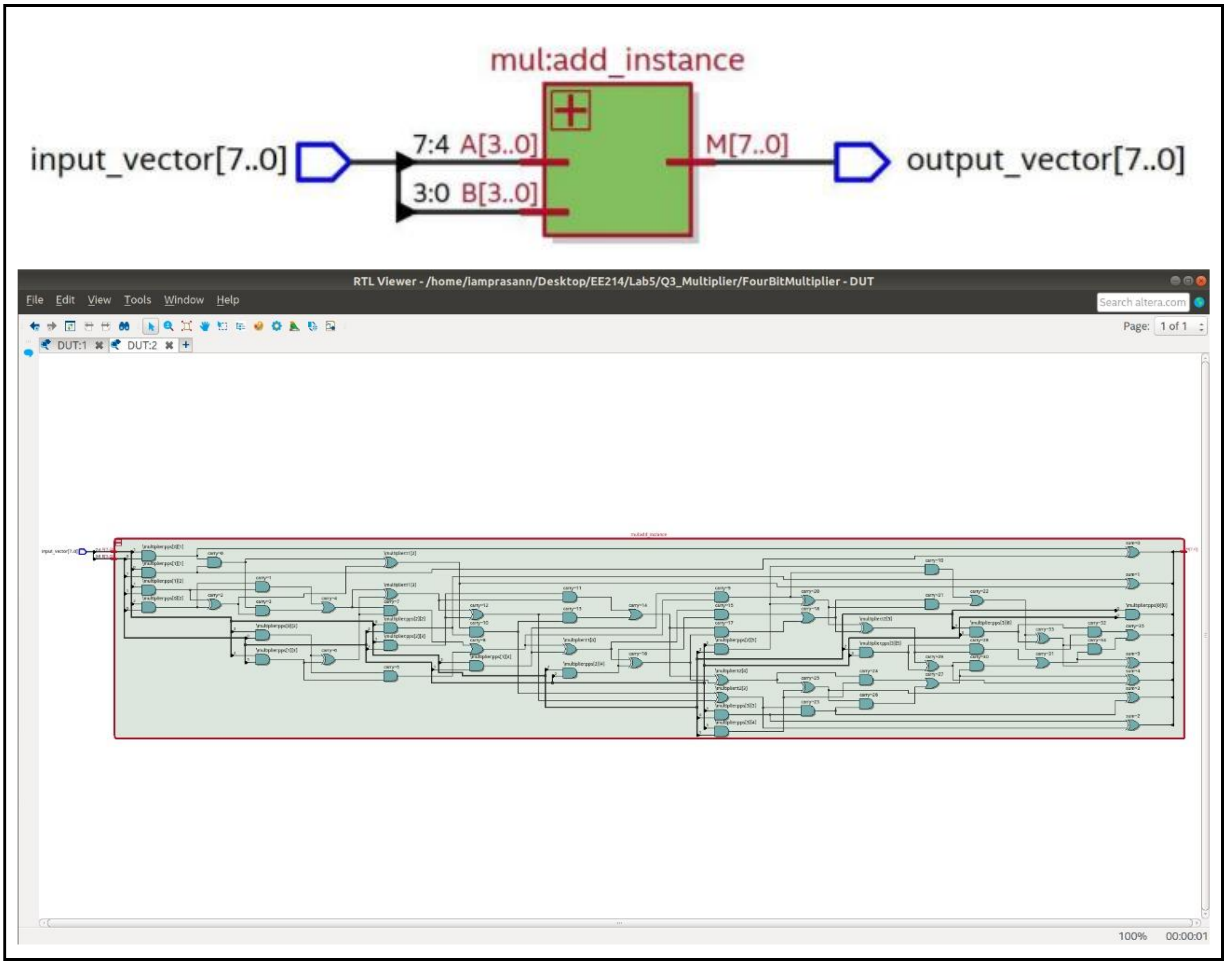
DUT: Wraps the Multiplier entity and converts the inputs and outputs to be std logic vectors.
Testbench: Compares outputs of TRACEFILE.txt and output of implementation.
Multiplier: Main logic, was a code template provided by TAs. The emboldened parts are the changes and code additions done by me.

Below is the architecture of Multiplier; The main logic:

```
architecture beh of mul is

-- unbounded 1D X 1D array declaration

type pp_type is array (N-1 downto 0) of std_logic_vector(NN-1 downto 0);

-- 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;

begin
multiplier : process(A, B)
-- declaration of 1D X 1D array to store partial products
variable pps : pp_type := (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-1 downto 0):= (others=>'0');

variable t1 : std_logic_vector(NN-1 downto 0):= (others=>'0');
variable t2 : std_logic_vector(NN-1 downto 0):= (others=>'0');

begin
-- Calculation of partial product

for i in 0 to N-1 loop
        for j in 0 to N-1 loop
                pps(i)(i+j):=A(j) and B(i);
        end loop;
end loop;

-- summation of partial product
```

```
t1 := adder(pps(0), pps(1))(NN-1 downto 0);
t2 := adder(t1, pps(2))(NN-1 downto 0);
result := adder(t2, pps(3))(NN-1 downto 0);

M <= result; -- assignment of final result
end process ; -- multiplier
end beh ; -- beh
```

RTL View:



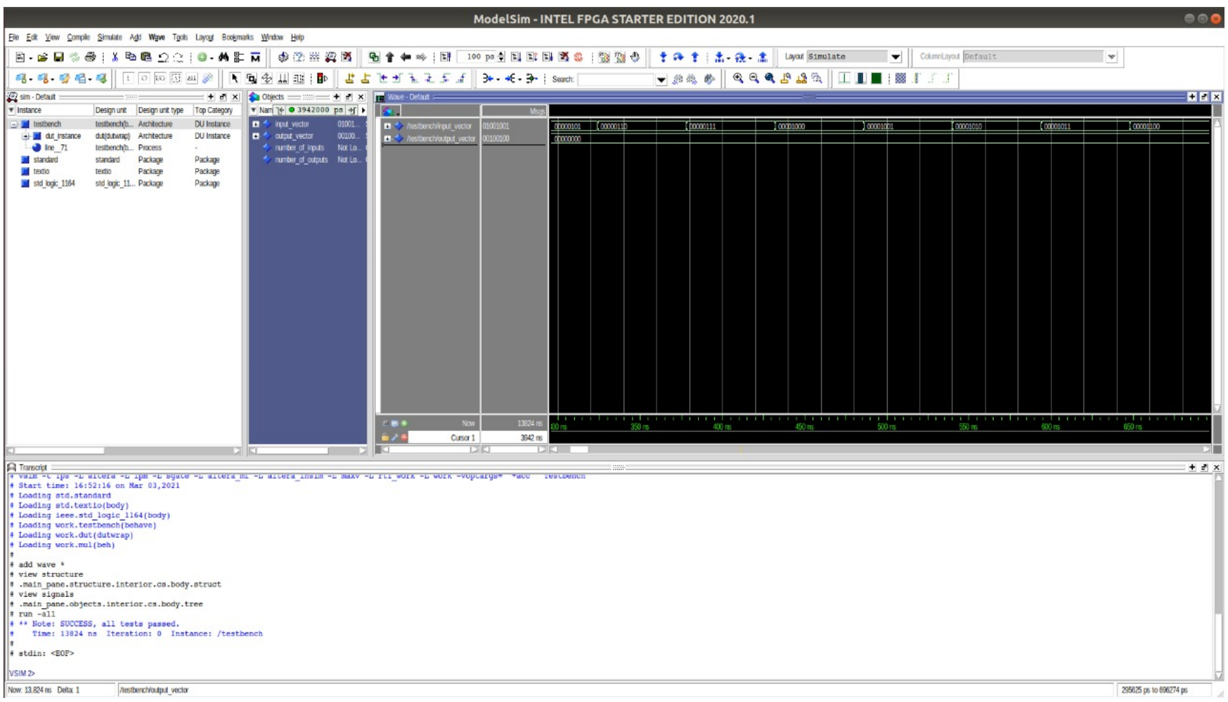
DUT Input/Output Format:

Input is a std logic vector of size 8. The lower nibble of the input represents B3B2B1B0 and the upper nibble of the input is A3A2A1A0 where A and B are the input 4 bit numbers to be multiplied.
Output is a std logic vector of size 8. It represents M7,…,M0, the 8 bit result of multiplication of A and B.
Some examples of testcases taken from TRACEFILE.txt are:

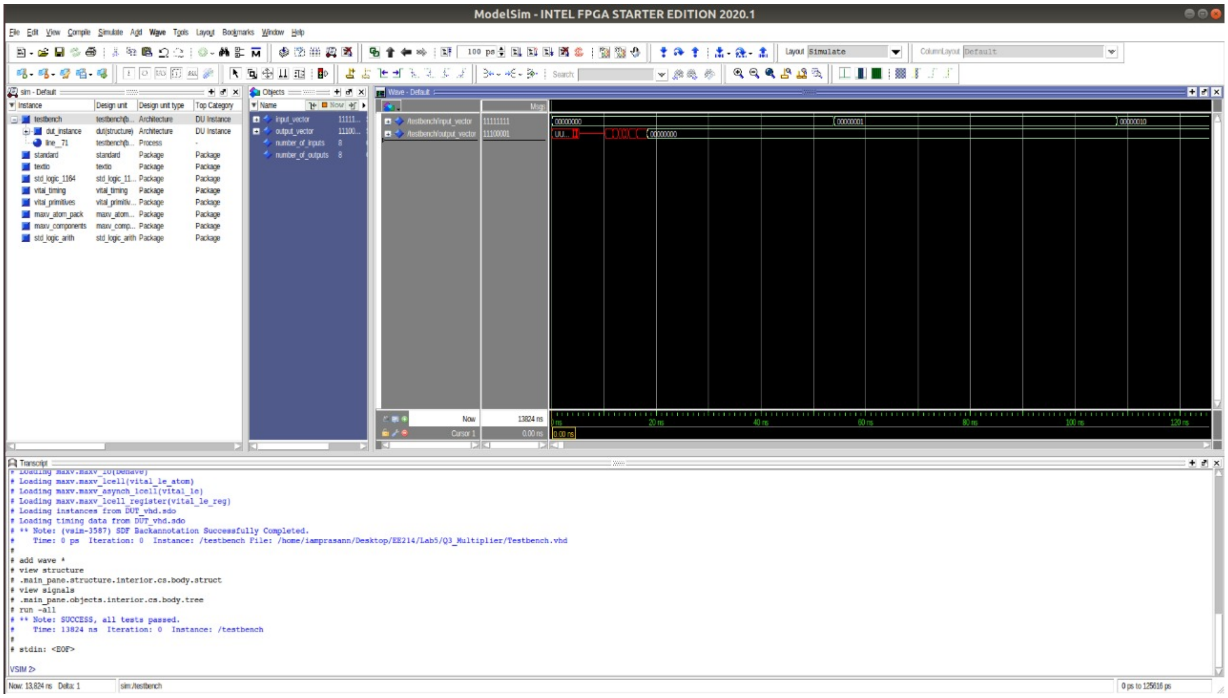
10110001 00001011 11111111
10110010 00010110 11111111
10110011 00100001 11111111
10110100 00101100 11111111
10110101 00110111 11111111
10110110 01000010 11111111
10110111 01001101 11111111

There were generated by a simple python script.

RTL Simulation:

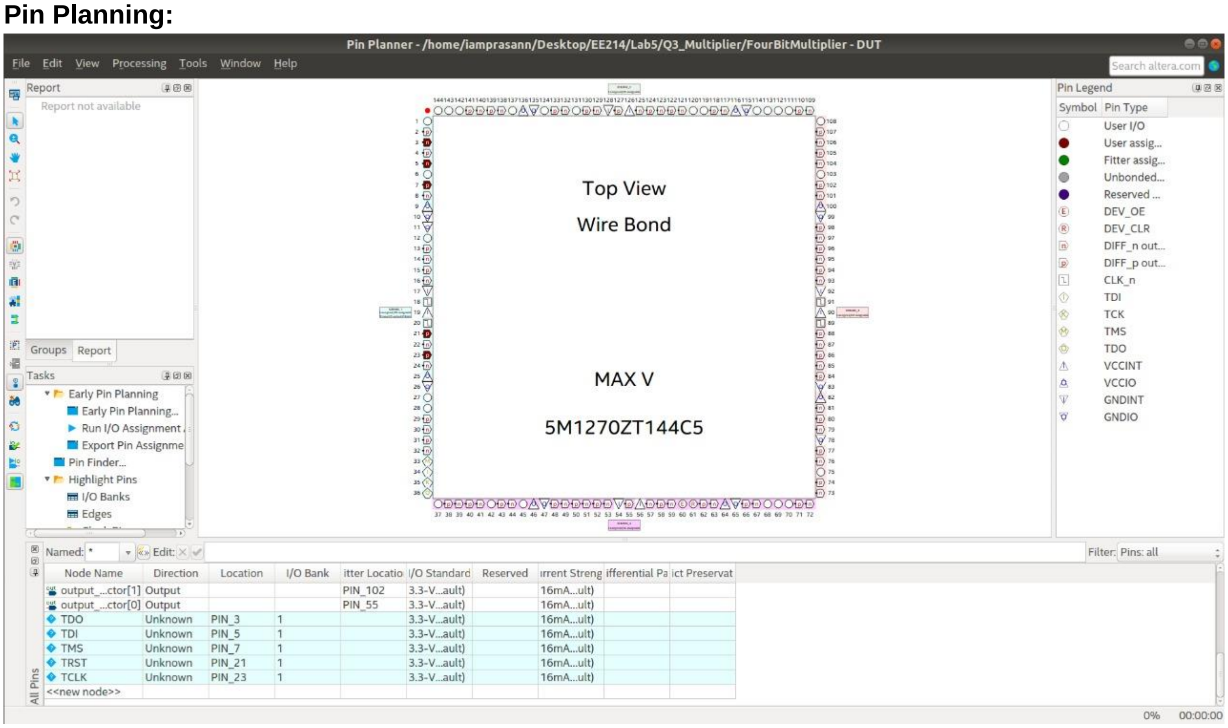


Gate-level Simulation:

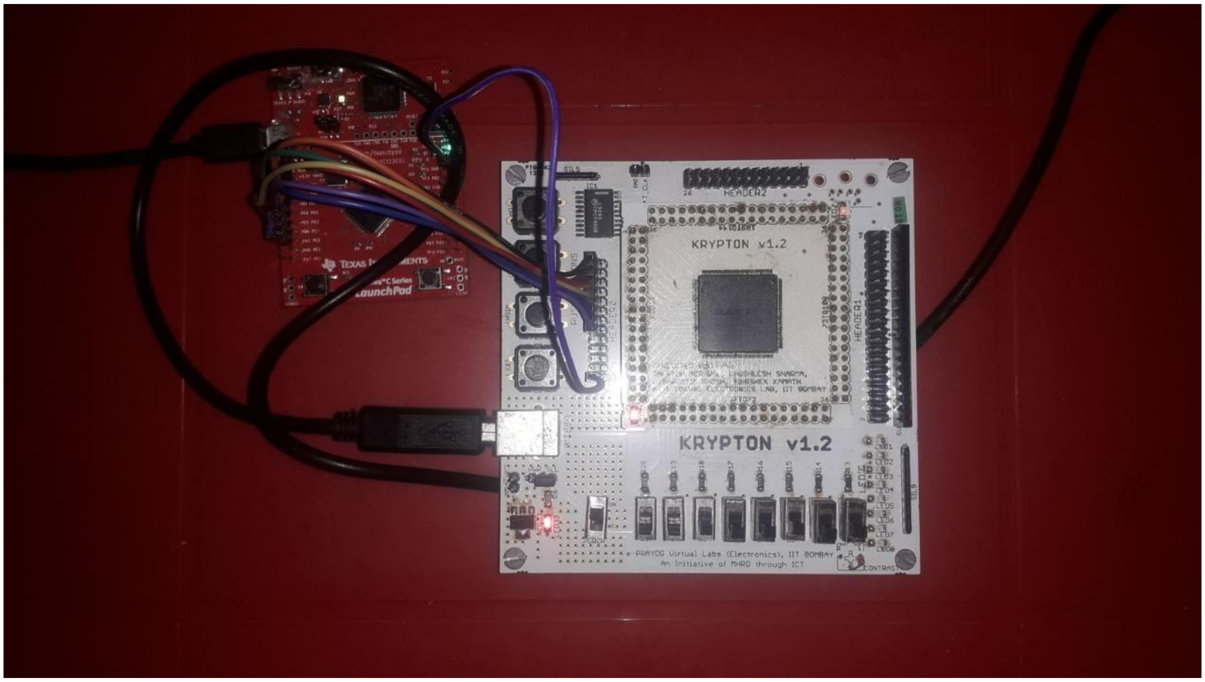


Krypton board:

Pin Planning:



Connections on Krypton and Tiva:



Outputs Passed:

out.txt [Read-Only]
~/Desktop/EE214/Lab5/Q3_Multiplier

Expected Output	Received Output	Remarks
00	00	Success
00	00	Success
00	00	Success
00	00	Success
00	00	Success
00	00	Success
00	00	Success
00	00	Success
00	00	Success
00	00	Success
00	00	Success
00	00	Success
00	00	Success
00	00	Success
00	00	Success
00	00	Success
00	00	Success
01	01	Success
02	02	Success
03	03	Success
04	04	Success
05	05	Success
06	06	Success
07	07	Success
08	08	Success
09	09	Success
0A	0A	Success
0B	0B	Success
0C	0C	Success
0D	0D	Success
0E	0E	Success
0F	0F	Success
00	00	Success
02	02	Success
04	04	Success
06	06	Success
08	08	Success
0A	0A	Success
0C	0C	Success
0E	0E	Success
10	10	Success
12	12	Success

Plain Text ▾ Tab Width: 8 ▾ Ln 1, Col 1 ▾ INS

Observations:

The TIVA Scan Chain routine acted to automatically provide 8 bit inputs. Then all these testcases were run in the Krypton board and it passed all test cases as informed by out.txt

References:

Outline code provided by EE214 TA team.