Experiment 1: Prime Checker and Adder

Rathour Param Jitendrakumar Roll Number: 190070049
EE-214, WEL, IIT Bombay
February 20, 2021

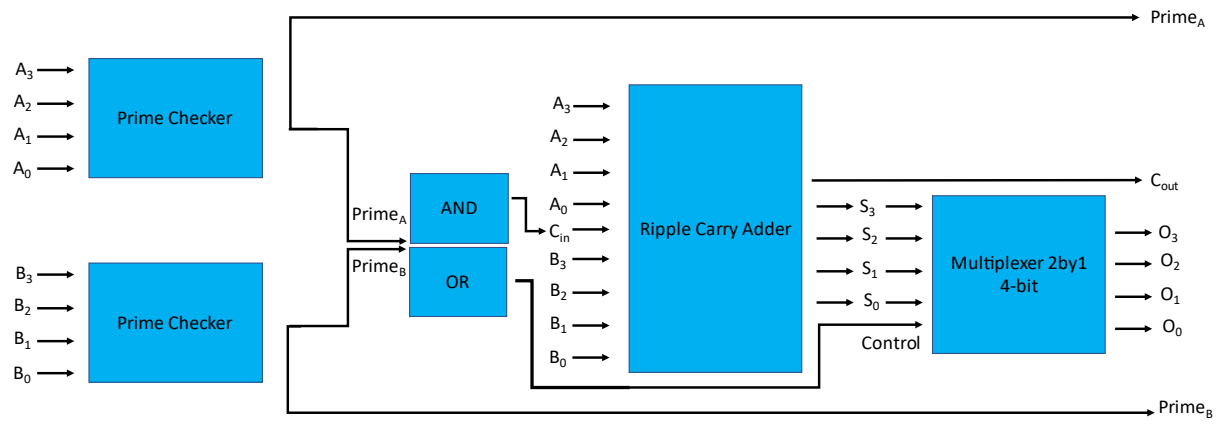
Overview of the experiment:

This experiment aims to design a logic circuit that determines if 2 input numbers are prime and outputs their addition/subtraction accordingly. Describe it using VHDL, simulate with the generic test-bench using Quartus and map it to the Krypton board and verify all test cases using scan chain

I designed the circuit and verified my implementation using post-synthesis gate-level by generating the trace file for all 256 combinations. Then programmed it to the Krypton board.
Subparts: 4-bit Prime Checker with AND, OR, NOT gates and with only Multiplexers.

This report contains my approach to the experiment, the design of the circuit 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 mapping it to Krypton board.

Approach to the experiment:

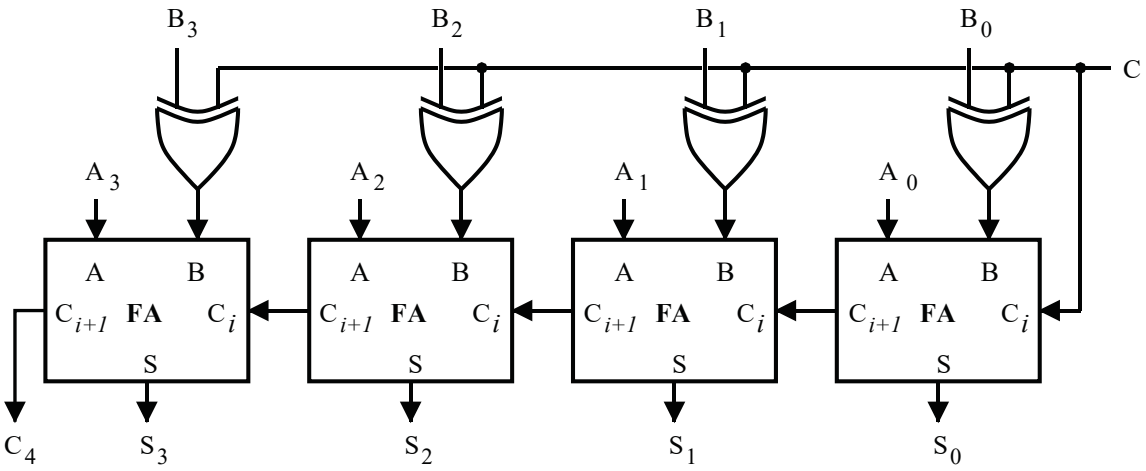


Full Schematic

My approach was to first design a circuit for Prime Checking using K-map minimisation.
So, the function is F = $A_2(\sim A_1)A_0 + (\sim A_3)A_2A_0 + (\sim A_2)A_1A_0 + (\sim A_3)\,(\sim A_2)A_1$ (~ means NOT)

I used a generic 4-bit ripple carry adder, which adds if carry is 0 and subtracts when the carry is 1 and output carry is XORed with the input carry as numbers are in 2's complement form.
This input carry is determined by taking AND of outputs of Prime Checker.
This method also solves problem of addition when only 1 input is prime & subtraction if both are prime.

For the case when both inputs are not prime all output must be zero. This is done by taking OR of outputs of Prime Checker and using that signal as control bit for 4 2by1 Multiplexers (one for each bit)



A - B

Design document and VHDL code if relevant:

This whole design is part of Device Under Test with 8 input bits & 7 output bits which confirm our design.
See DUT Input/Output Format:
I constructed an entity named:
PrimeChecker which checks if a 4-bit number is prime

PrimeCheckerAdder,which uses PrimeChecker's output, adds/subtracts given 4-bit numbers accordingly

Generic RippleCarryAdder is used to perform addition and subtraction and

Mux2by1 handles case when both numbers are not prime (all outputs zero)

PrimeCheckerAdder

```
entity PrimeCheckerAdder is
    port(A: in  std_logic_vector(3 downto 0);
         B: in  std_logic_vector(3 downto 0);
         O: out std_logic_vector(3 downto 0);
         Cout, PrimeA, PrimeB: buffer std_logic);
end entity PrimeCheckerAdder;

architecture Struct of PrimeCheckerAdder is
    signal Cin, Co, En: std_logic;
    signal AopB: std_logic_vector(3 downto 0);
begin

    CheckA: PrimeChecker port map (A => A, Y => PrimeA); -- NotA(0) = NOT A(0)
    CheckB: PrimeChecker port map (A => B, Y => PrimeB); -- NotA(0) = NOT A(0)

    Carry: AND_2 port map (A => PrimeA, B => PrimeB, Y => Cin);

    -- This adder does both addition and subtraction based on carry
    Addition: RippleCarryAdder port map (A => A, B=> B, Cin=> Cin, S=> AopB, Cout=> Co);

    -- Below part is if both numbers are not prime then make all outputs 0
    Enable: OR_2 port map (A => PrimeA, B => PrimeB, Y => En);
    FinalOutput: MUX_2by1_4bit port map (En => '1', S => En, Cin => Co, I => "00000", J => AopB, Cout => Cout, Y => O);

end Struct;
```

Prime Checker

```
entity PrimeChecker is
    port(A: in  std_logic_vector(3 downto 0);
         Y: out std_logic);
end entity PrimeChecker;

architecture Struct of PrimeChecker is
    signal C: std_logic_vector(4 downto 1);
    signal NotA: std_logic_vector(3 downto 0);
begin

    Not1: INVERTER port map (A => A(0), Y => NotA(0)); -- NotA(0) = NOT A(0)
    Not2: INVERTER port map (A => A(1), Y => NotA(1)); -- NotA(1) = NOT A(1)
    Not3: INVERTER port map (A => A(2), Y => NotA(2)); -- NotA(2) = NOT A(2)
    Not4: INVERTER port map (A => A(3), Y => NotA(3)); -- NotA(3) = NOT A(3)

    -- Output bit function is = A(2)· NotA(2)· A(0) + NotA(3)· A(2) · A(0) + NotA(2) · A(1)· A(0) + NotA(3) · NotA(2) · A(1)
    A1: AND_3 port map (A => A(2), B => NotA(1), C => A(0), Y => C(1)); -- C(1) = A(2)· NotA(1)· A(0)
    A2: AND_3 port map (A => NotA(3), B => A(2),    C => A(0), Y => C(2)); -- C(2) = NotA(3)· A(2)   · A(0)
    A3: AND_3 port map (A => NotA(2),    B => A(1), C => A(0), Y => C(3)); -- C(3) = NotA(2)   · A(1)· A(0)
    A4: AND_3 port map (A => NotA(3),    B => NotA(2),    C => A(1), Y => C(4)); -- C(4) = NotA(3)   · NotA(2)   · A(1)

    O4: OR_4 port map (A => C(1), B => C(2), C => C(3), D => C(4), Y => Y); -- Y = C(1) + C(2) + C(3) + C(4)

end Struct;
```

Prime Checker with only Multiplexers

```
entity PrimeCheckerWithMux is
    port(A: in  std_logic_vector(3 downto 0);
         Y: out std_logic);
end entity PrimeCheckerWithMux;

architecture Struct of PrimeCheckerWithMux is
    signal I: std_logic_vector(7 downto 0);
begin
    I <= '0' & A(0) & A(0) & '0' & A(0) & A(0) & '1' & '0';
    MuX8by1_instance: MuX_8by1 port map (En => '1', I => I, S => A(3 downto 1), Y => Y);

end Struct;
```

RippleCarryAdder

```
entity RippleCarryAdder  is
  port(A, B: in  std_logic_vector(3 downto 0);
       Cin: in std_logic;
       S:  out std_logic_vector(3 downto 0);
       Cout: out std_logic);
end entity RippleCarryAdder;

architecture Struct of RippleCarryAdder is
  signal BMod: std_logic_vector(3 downto 0);
  signal C: std_logic_vector(2 downto 0);
  signal Co: std_logic;
begin

  -- XOR to invert B
  XOR1: XOR_2 port map (A => B(0), B => Cin, Y => Bmod(0));

  XOR2: XOR_2 port map (A => B(1), B => Cin, Y => Bmod(1));

  XOR3: XOR_2 port map (A => B(2), B => Cin, Y => Bmod(2));

  XOR4: XOR_2 port map (A => B(3), B => Cin, Y => Bmod(3));

  FA1: FULL_ADDER port map (A => A(0), B => BMod(0), Cin => Cin,  S => S(0), Cout => C(0));

  FA2: FULL_ADDER port map (A => A(1), B => BMod(1), Cin => C(0), S => S(1), Cout => C(1));

  FA3: FULL_ADDER port map (A => A(2), B => BMod(2), Cin => C(1), S => S(2), Cout => C(2));

  FA4: FULL_ADDER port map (A => A(3), B => BMod(3), Cin => C(2), S => S(3), Cout => Co);

  FinalCarry: XOR_2 port map (A => Co, B => Cin, Y => Cout);

end Struct;
```

Multiplexer 8by1

```
entity MUX_8by1  is
    port(En:in  std_logic;
              I: in  std_logic_vector(7 downto 0);
              S: in  std_logic_vector(2 downto 0);
              Y: out std_logic);
end entity MUX_8by1;

architecture Struct of MUX_8by1 is
    signal C: std_logic_vector(2 downto 1);
    signal InputToMuxc: std_logic_vector(3 downto 0);
    signal ControlToMuxc: std_logic_vector(1 downto 0);
begin

    MuX4by1a: MuX_4by1 port map (En => En, I => I(3 downto 0), S => S(1 downto 0), Y=> C(1));

    MuX4by1b: MuX_4by1 port map (En => En, I => I(7 downto 4), S => S(1 downto 0), Y=> C(2));

    InputToMuxc <= C(2) & "00" & C(1);
    ControlToMuxc <= S(2) & S(2);
    MuX4by1c: MuX_4by1 port map (En => En, I => InputToMuxc, S => ControlToMuxc, Y=> Y);

end Struct;
```

Multiplexer 4by1

```
entity MUX_4by1  is
    port(En:in  std_logic;
              I: in  std_logic_vector(3 downto 0);
              S: in  std_logic_vector(1 downto 0);
              Y: out std_logic);
end entity MUX_4by1;

architecture Struct of MUX_4by1 is
    signal C: std_logic_vector(4 downto 1);
    signal NotS: std_logic_vector(1 downto 0);
begin

    Not1: INVERTER port map (A => S(0), Y => NotS(0)); -- NotS(0) = NOT S(0)
    Not2: INVERTER port map (A => S(1), Y => NotS(1)); -- NotS(1) = NOT S(1)

    A1: AND_4 port map (A => NotS(1), B => NotS(0), C => I(0), D => En, Y => C(1)); -- C(1) = NotS(1)· NotS(0)· I(0)· En
    A2: AND_4 port map (A => NotS(1), B => S(0),    C => I(1), D => En, Y => C(2)); -- C(2) = NotS(1)· S(0)   · I(1)· En
    A3: AND_4 port map (A => S(1),    B => NotS(0), C => I(2), D => En, Y => C(3)); -- C(3) = S(1)   · NotS(0)· I(2)· En
    A4: AND_4 port map (A => S(1),    B => S(0),    C => I(3), D => En, Y => C(4)); -- C(4) = S(1)   · S(0)   · I(3)· En

    O4: OR_4 port map (A => C(1), B => C(2), C => C(3), D => C(4), Y => Y); -- Y = C(1) + C(2) + C(3) + C(4)

end Struct;
```

Multiplexer 2by1

```
 entity MUX_2by1  is
     port(En:in  std_logic;
          I1,I2: in  std_logic;
          S: in  std_logic;
          Y: out std_logic);
 end entity MUX_2by1;

 architecture Struct of MUX_2by1 is
     signal C: std_logic_vector(2 downto 1);
     signal NotS: std_logic;
 begin

     Not1: INVERTER port map (A => S, Y => NotS); -- NotS = NOT S

     A1: AND_3 port map (A => NotS, B => I1, C => En, Y => C(1)); -- C(1) = NotS· I1· En
     A2: AND_3 port map (A => S, B => I2, C => En, Y => C(2)); -- C(2) = S· I2· En

     O4: OR_2 port map (A => C(1), B => C(2), Y => Y); -- Y = C(1) + C(2)

 end Struct;
```

Python Code to generate test cases for PrimeCheckerAdder

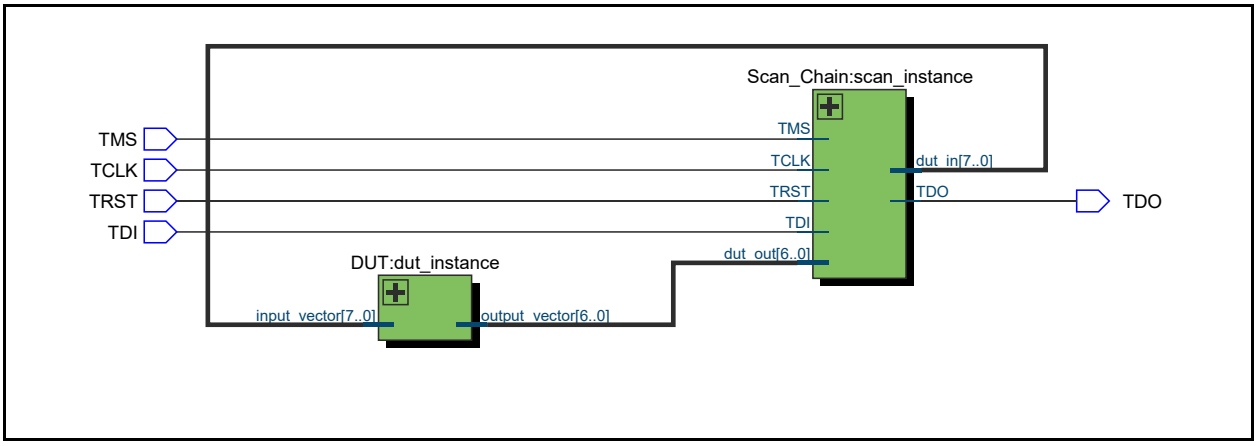
```
import numpy as np
import sympy as sp

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 addBinaryIntegers(a,b,n, Max):
    c = a + b
    carry = 0
    if c > Max - 1:
        c = c - Max
        carry = 1
    return c, carry

def subBinaryIntegers(a,b,n, Max):
    b = 2**n - b
    c = a + b
    carry = 0
    if c > Max - 1:
        c = c - Max
        carry = 1
    carry = 1 - carry
    return c, carry

subBinaryIntegers(2,14,n,Max)

def PrimeCheck(string):
    if (sp.isprime(int(string,2))):
        return 1
    else:
        return 0

numberOfOutputBits = 7
```

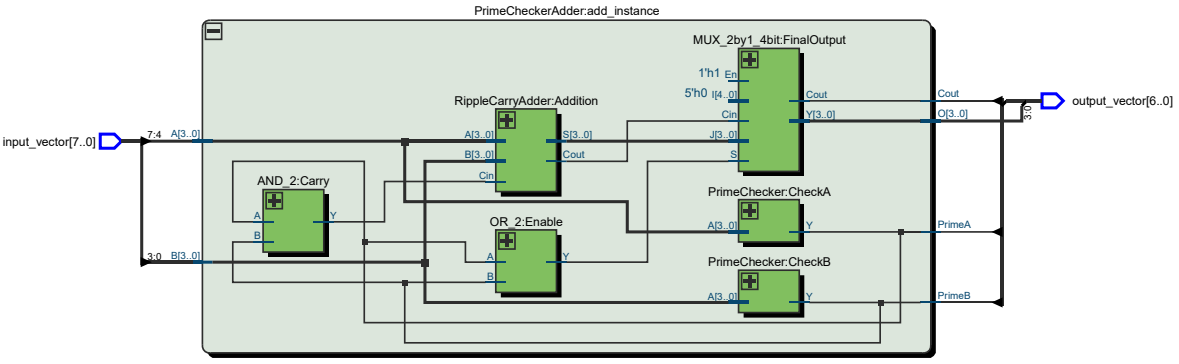
```
with open("Tracefile.txt", "w") as file:
    for i in range(Max):
        for j in range(Max):
            I1 = BinaryStrings[i]
            I2 = BinaryStrings[j]
            opa = PrimeCheck(I1)
            opb = PrimeCheck(I2)
            if opa == 1 and opb == 1:
                output, carry = subBinaryIntegers(i,j,n, Max)
            elif opa == 1 or opb == 1:
                output, carry = addBinaryIntegers(i,j,n, Max)
            else:
                output, carry = 0, 0
            mask = '1'*numberOfOutputBits
            o = bin(output)[2:].zfill(n)
            str = "{}{} {}{}{}{} {}\n".format(I1, I2, opa, opb, carry, o, mask)
            file.write(str)
```

Python Code to generate test cases for PrimeChecker

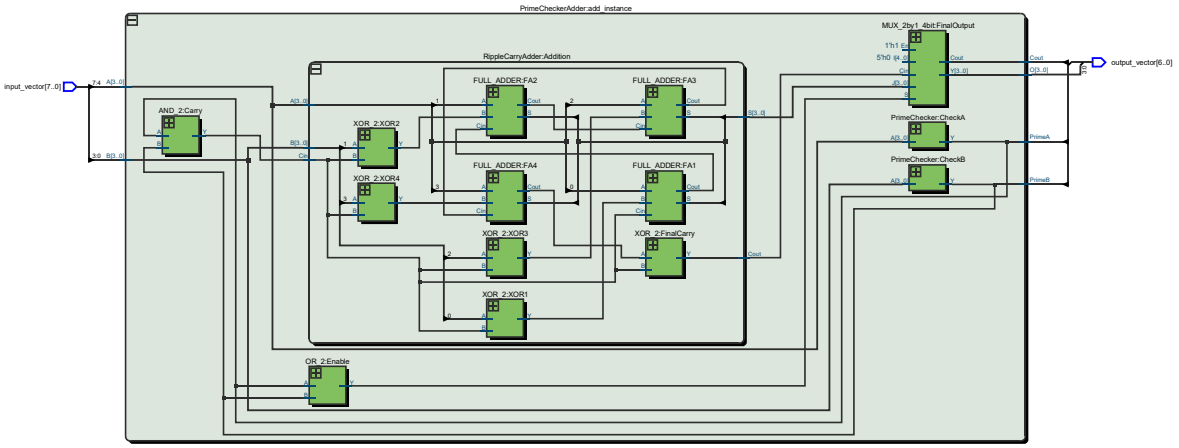
```python
import numpy as np
import sympy as sp

n = 4
Max = 2**n

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

BinaryStrings = generateBinaryStrings(n, Max)

s = "1011"

int(s,2)

def PrimeCheck(string):
    if (sp.isprime(int(string,2))):
        return 1
    else:
        return 0

numberOfOutputBits = 1

with open("TRACEFILE.txt", "w") as file:
    for i in range(Max):
        I = BinaryStrings[i]
        output = PrimeCheck(I)
        mask = '1'*numberOfOutputBits
        O = bin(output)[2:].zfill(numberOfOutputBits)
        str = "{} {} {}\n".format(I, O, mask)
        file.write(str)
```

RTL View:

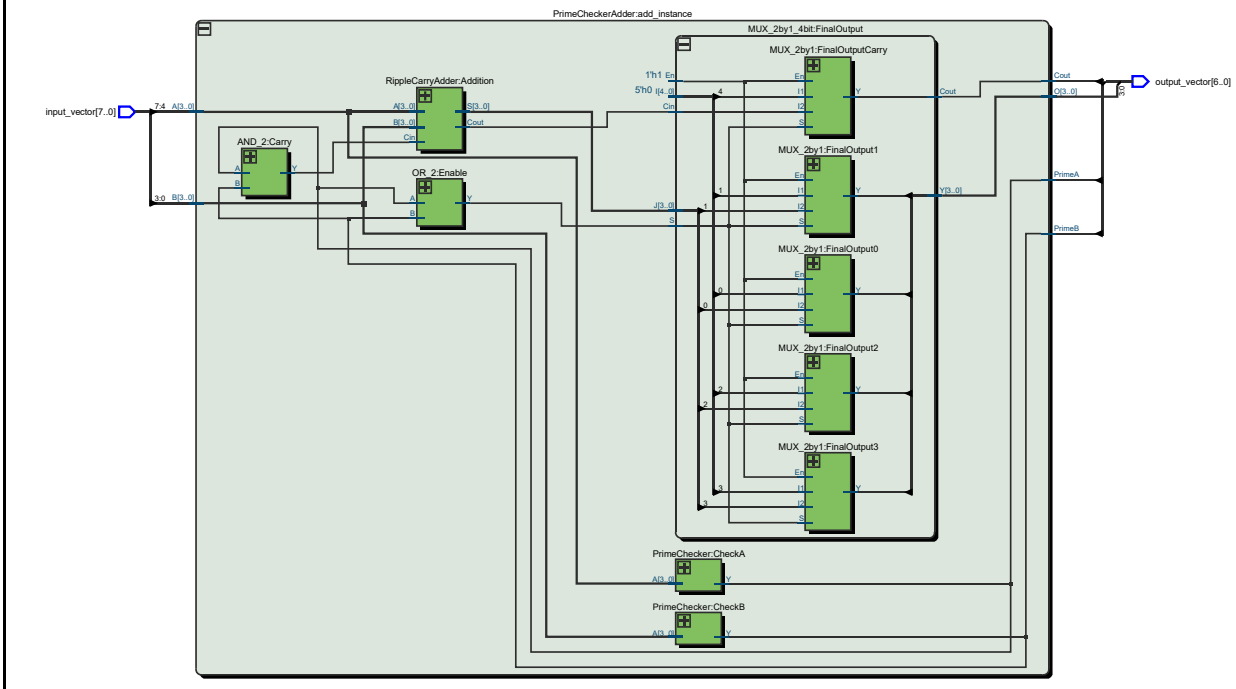




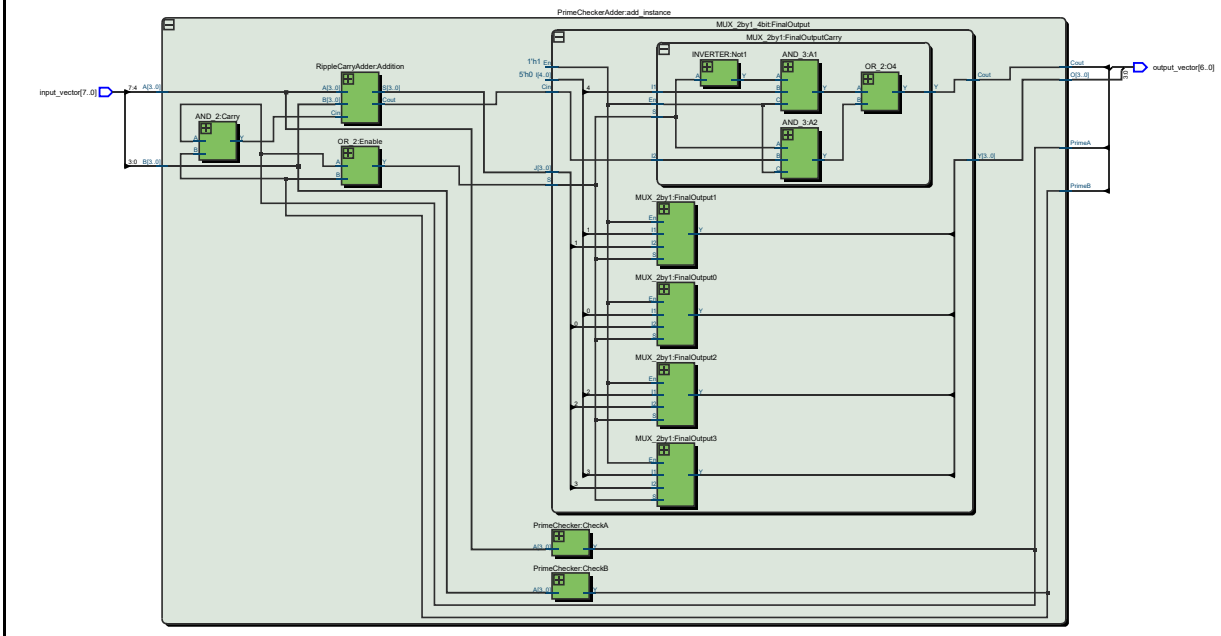
Prime Checker and Adder



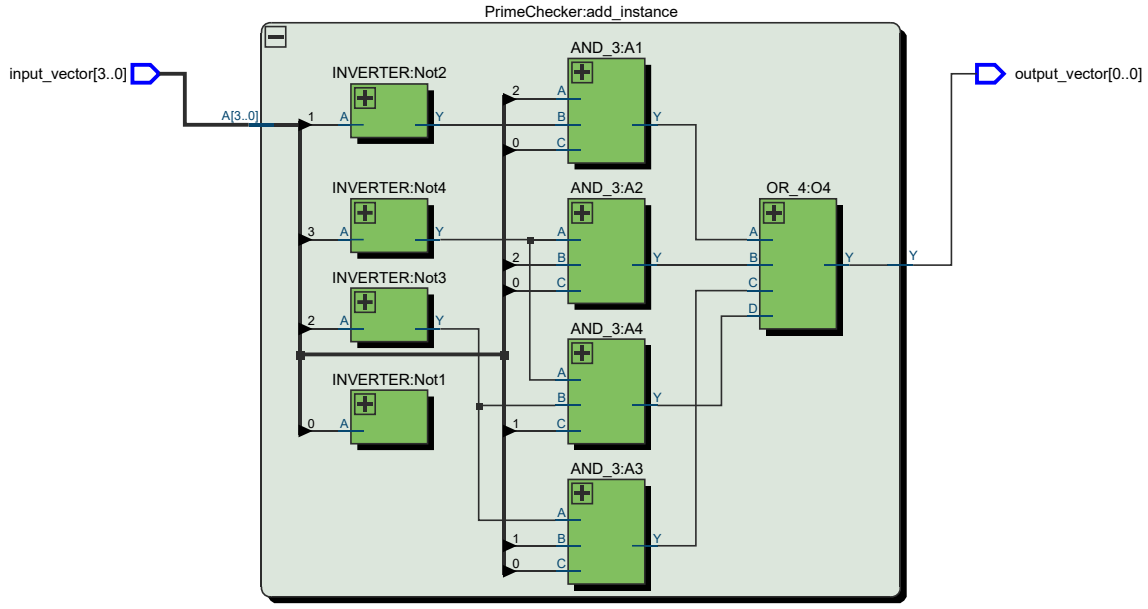
Ripple Carry Adder



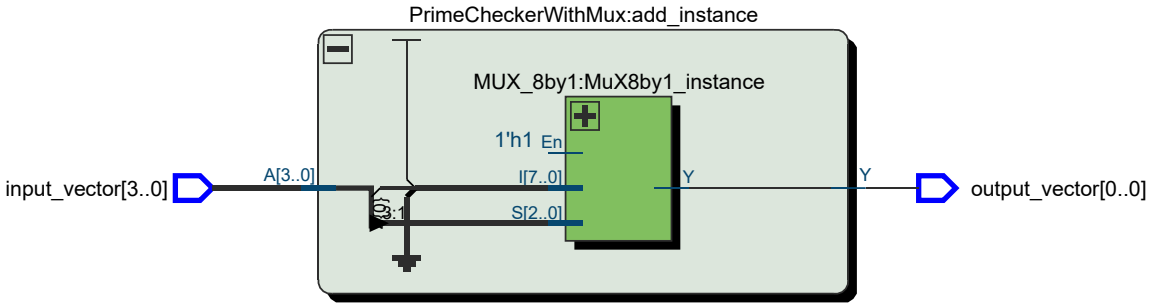
Mux 2by1 4-bit

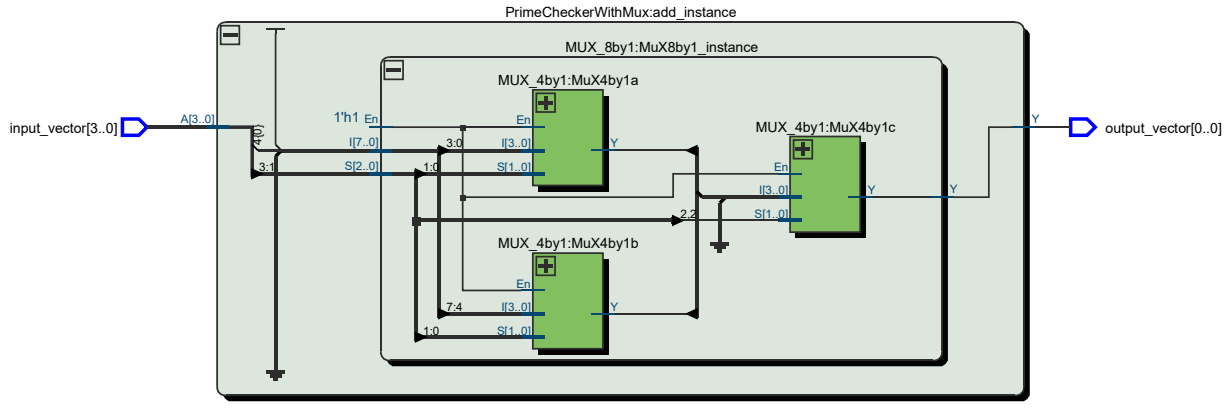


Prime Checker



Prime Checker With Mux





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: $P_AP_BC_{out}O_2O_1O_0$ (P_AP_B checks primality of A,B, Carry (1-bit) and Sum (4-bit S_3 is MSB S_0 is LSB)
00000000 0000000 1111111 00001011 0101011 1111111 00101100 1001110 1111111 01110001 1001000 1111111 10111011 1100000 1111111 11011100 1011001 1111111 11110100 0000000 1111111 11111111 0000000 1111111

RTL Simulation:

Gate-level Simulation:

Krypton board:

For Part 1) & 3)
Inputs: S4-S1
Outputs: LED_8

input_vector[3]	Input	PIN_43	4
input_vector[2]	Input	PIN_44	4
input_vector[1]	Input	PIN_45	4
input_vector[0]	Input	PIN_48	4
output_vector[0]	Output	PIN_49	4

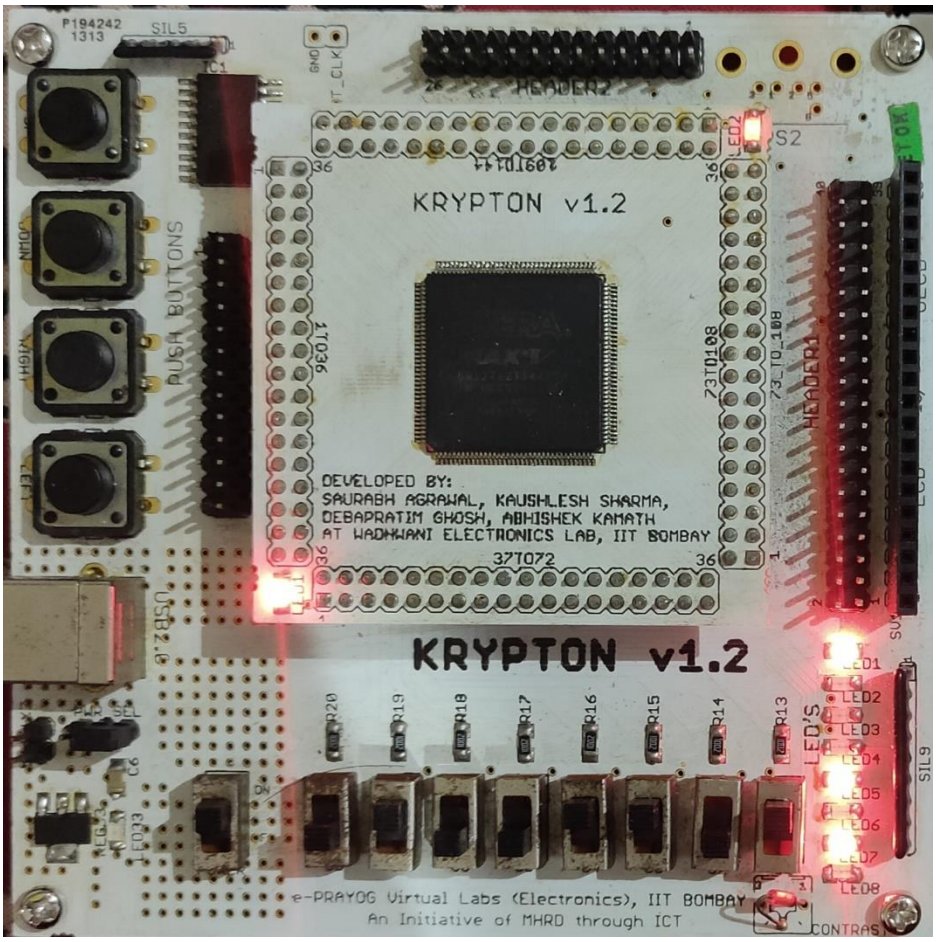
For Part 2)
Inputs: S8-S5 and S4-S1
Outputs: $LED_8LED_7LED_5LED_4LED_3LED_2LED_1$
For Scan Chain

in TCLK	Input	PIN_23	1	PIN_23
in TDI	Input	PIN_5	1	PIN_5
out TDO	Output	PIN_3	1	PIN_3
in TMS	Input	PIN_7	1	PIN_7
in TRST	Input	PIN_21	1	PIN_21

For DUT

input_vector[7]	Unknown	PIN_39
input_vector[6]	Unknown	PIN_40
input_vector[5]	Unknown	PIN_41
input_vector[4]	Unknown	PIN_42
input_vector[3]	Unknown	PIN_43
input_vector[2]	Unknown	PIN_44
input_vector[1]	Unknown	PIN_45
input_vector[0]	Unknown	PIN_48

output_vector[6]	Unknown	PIN_49
output_vector[5]	Unknown	PIN_50
output_vector[4]	Unknown	PIN_52
output_vector[3]	Unknown	PIN_53
output_vector[2]	Unknown	PIN_55
output_vector[1]	Unknown	PIN_57
output_vector[0]	Unknown	PIN_58



Observations:

All test cases passed in RTL, Gate-Level as well as Scan-Chain
Example (see figure): For input 01001101 output is 0110001 ($LED_{7,5,1}$ are on)
4 is not a prime, 13 is a prime, 4 + 13 = 1 and carry = 1

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


#------ Command -  509 : SDR 8 TDI(FE) 7 TDO(00) MASK(FF) ------#

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

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


#------ Command -  511 : SDR 8 TDI(FF) 7 TDO(00) 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.
```

References:

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