1- Introduction

2- Basic Structures of VHDL

3- Combinational Circuits

4- Sequential Circuits

5- Memory

6- Writing Testbenches

7- Synthesis Issues

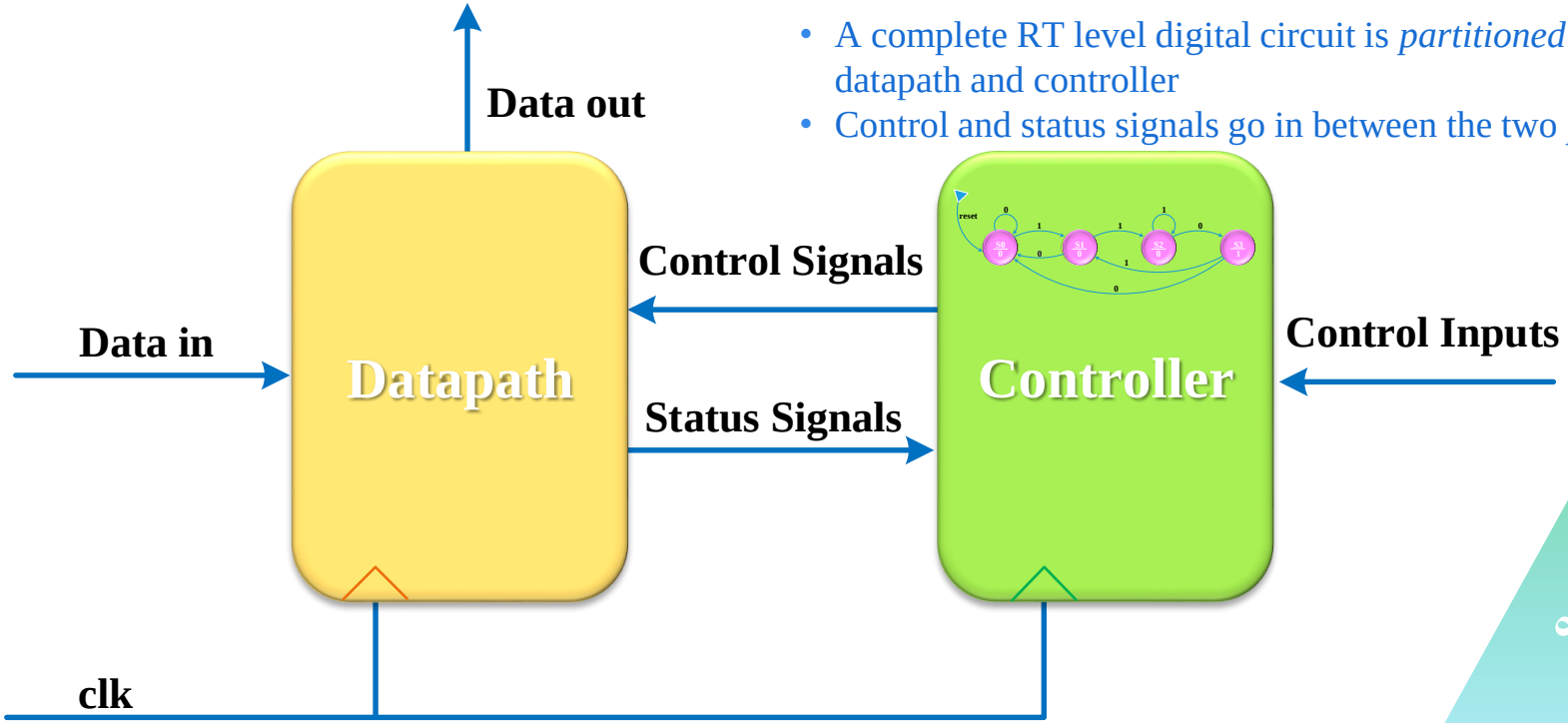
8- RTL Cores

- **Datapath and Controller**
- **RTL Timing**
- **Quad Serial Adder Circuit**
- **Exponential Circuit**

8- RTL Cores

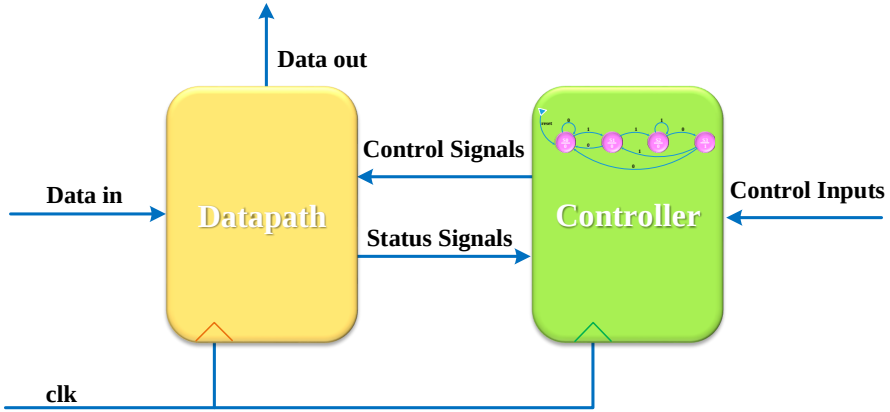
DataPath and Controller

- A complete RT level digital circuit is *partitioned* into datapath and controller
- Control and status signals go in between the two *partitions*



DataPath and Controller

- **Datapath**
  - Combinational circuits
  - Sequential circuits
- **Controller**
  - State machines



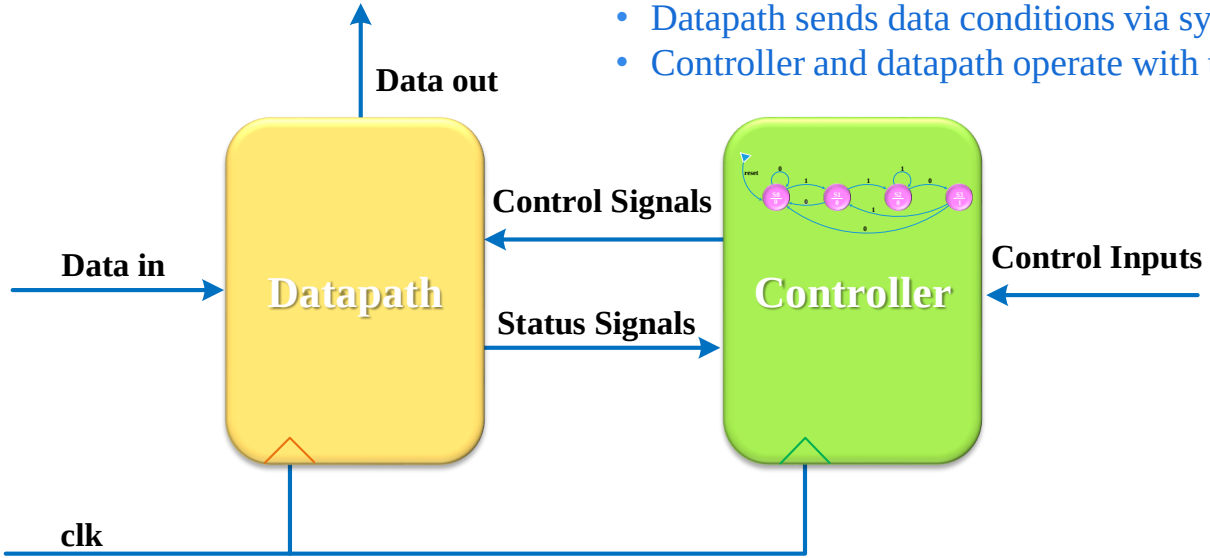
- Datapath contains datapath components and their connections through busses
- With its components, datapath performs operations
- Control signals tell datapath when to perform operations
- Datapath: **What to do**
- Controller: **When to do**
- Controller properly issues control signals telling datapath when to perform an operation

DataPath

- A datapath is a collection of functional units such as arithmetic logic units or multipliers that perform data processing operations, registers, and buses.
- The datapath and the control unit compose the operation of the RTL circuit.
- For a larger design, individual RTL circuits can be interfaced.

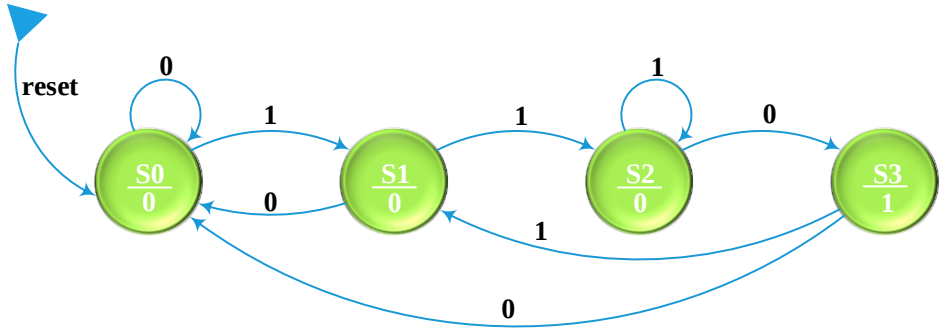
RTL Timing

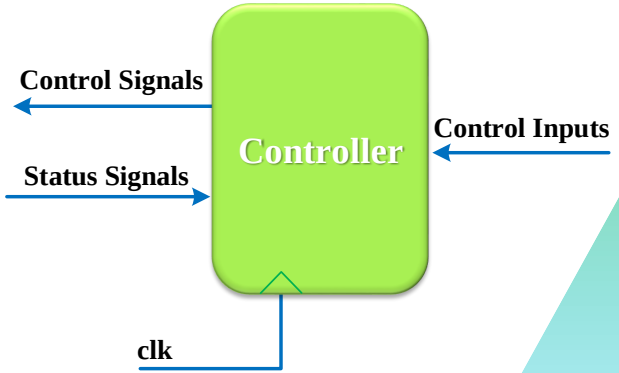
- Before we start on datapath and controller and design of complete system, timing and synchronization of datapath and controller deserve attention
- Controller issues control signals to the datapath
- Datapath sends data conditions via synchronous signals to the controller
- Controller and datapath operate with the same exact clock



RTL Timing

- State Machine
  - Let's look at controller timing
  - Consider controller as a state machine
  - State machine inputs are signals from datapath or external signals
  - State machine outputs are signals issued to datapath and external control signals

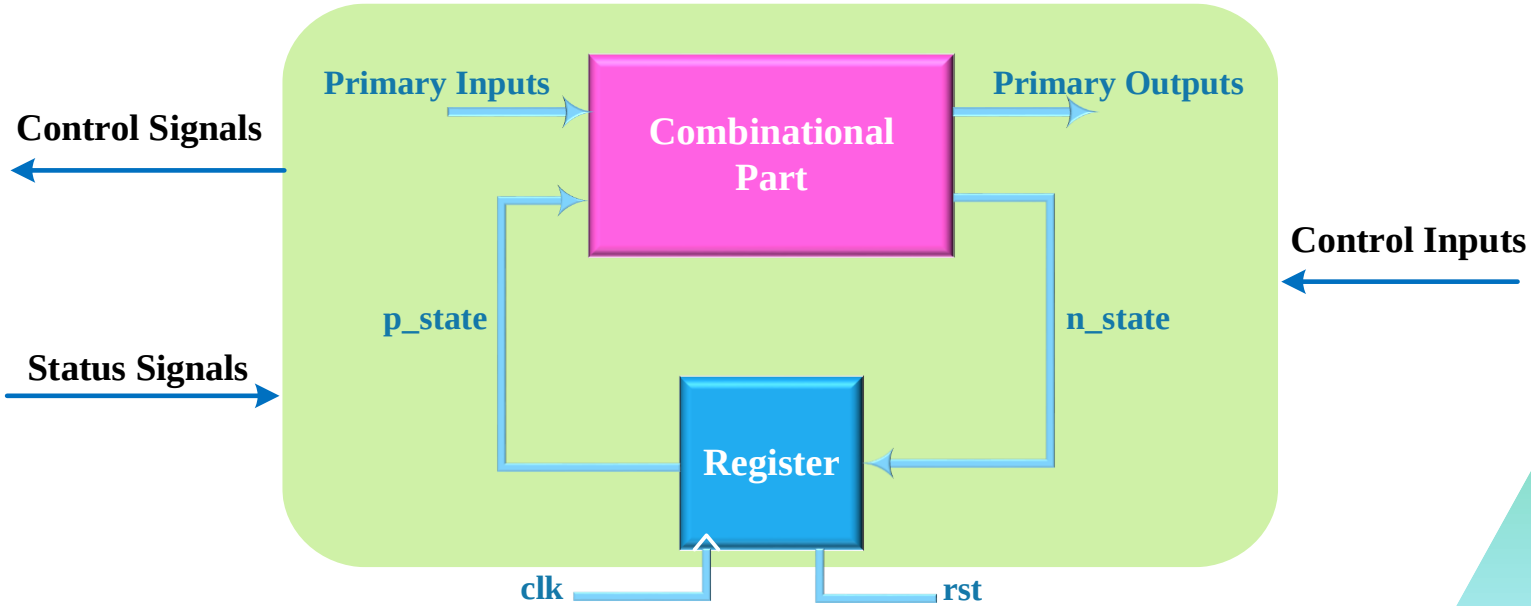




8- RTL Cores

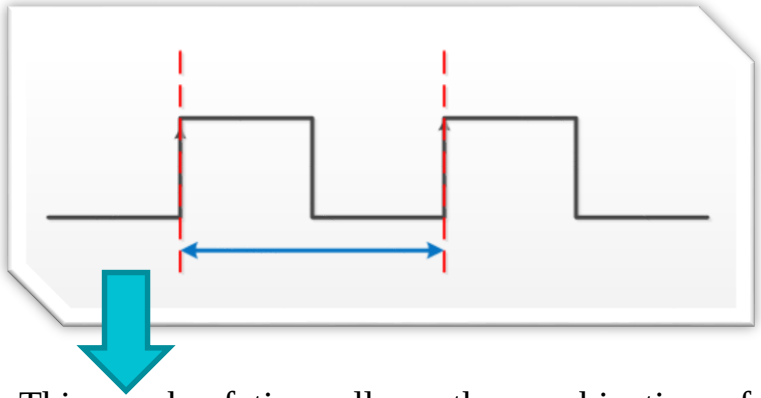
RTL Timing

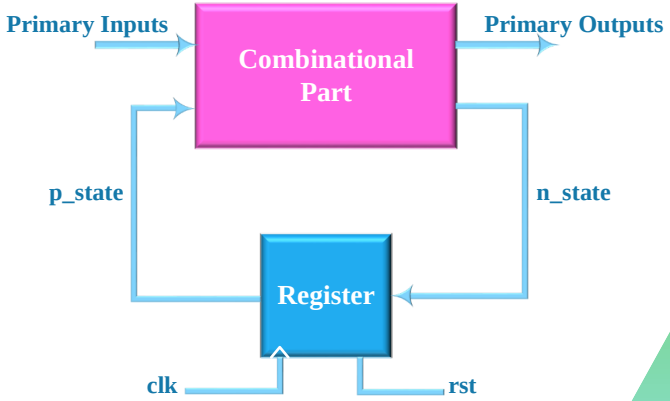
- Controller FSM can be categorized as Huffman model



RTL Timing

- Controller timing



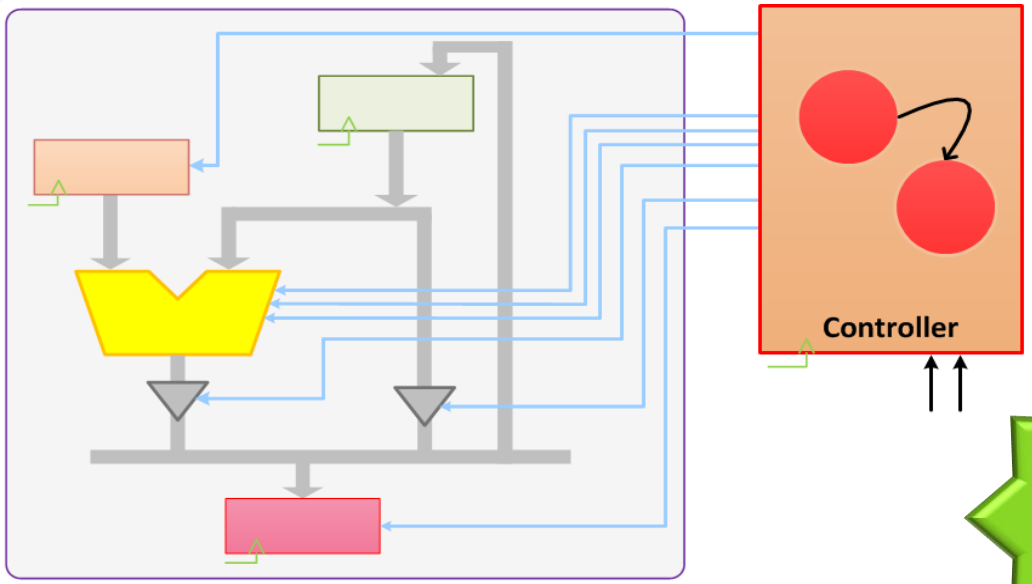


This much of time allows the combination of PS and PI to propagate and reach NS and PO output.

Primary outputs produce their values right after the clock ticks and remain that way for some time after the next clock.

RTL Timing

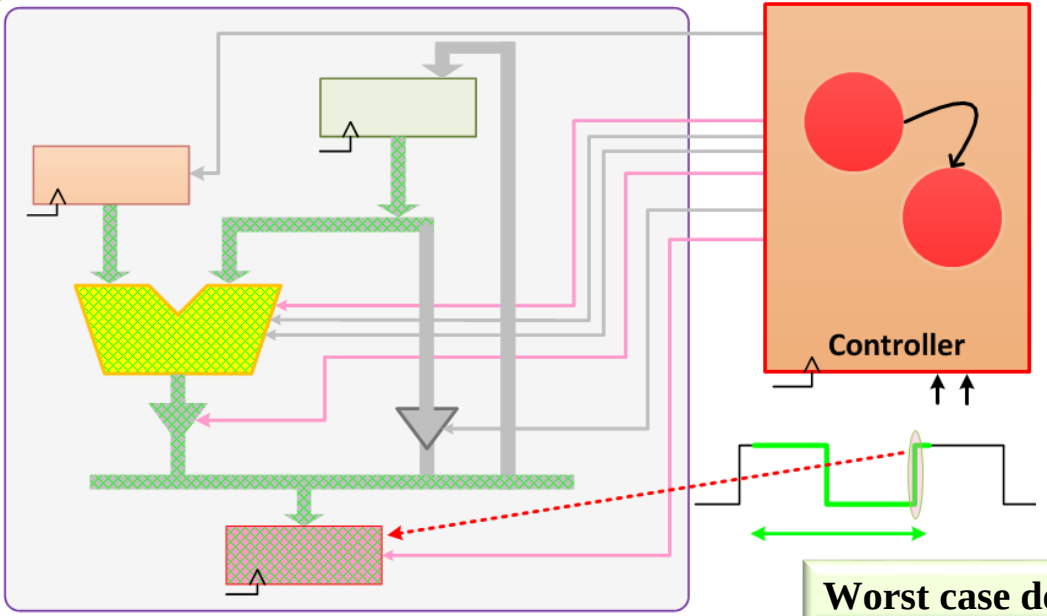
- Datapath / Controller timing



- Control signals issued to the datapath follow the same exact timing as those internal to the controller
- Control signals control flow of data in datapath
- Generally, propagation in datapath is slower than that in controller

RTL Timing

- Datapath / Controller timing



- Bus logic selection and register clocking signals all go from one edge to the next edge
- Clock must be slow enough to enable longest register to register path to propagate in datapath

Worst case delay => Clocking speed

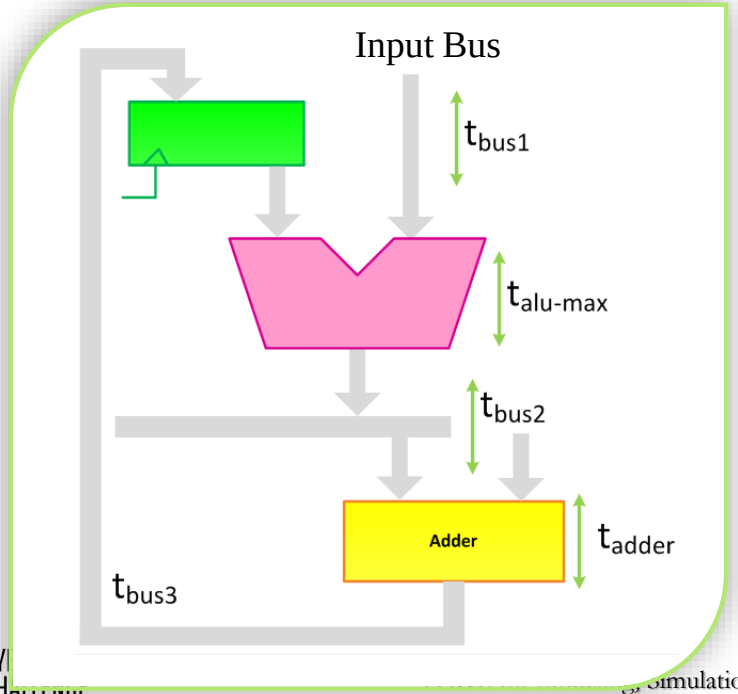
RTL Timing

Datapath timing

- For finding the clock frequency we should consider the worst case delay (bottleneck) of the datapath
- If we have an ALU in the datapath, we should take the delay of the slowest ALU operation as the ALU delay
- Selection of each bus has also its own delay
- Each register has setup time delay

RTL Timing

- Datapath timing

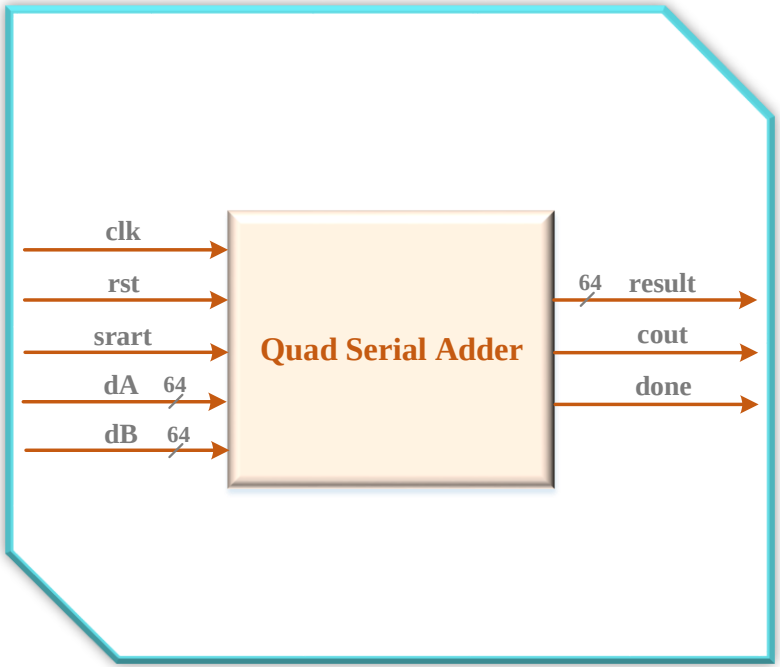


- Clock speed calculation
- This is the longest path in this example from the input bus to the register

$$t_{total} = t_{bus1} + t_{alu\text{-}max} + t_{bus2} + t_{adder} + t_{bus3} + t_{setup}$$

8- RTL Cores

Quad Serial Adder Circuit



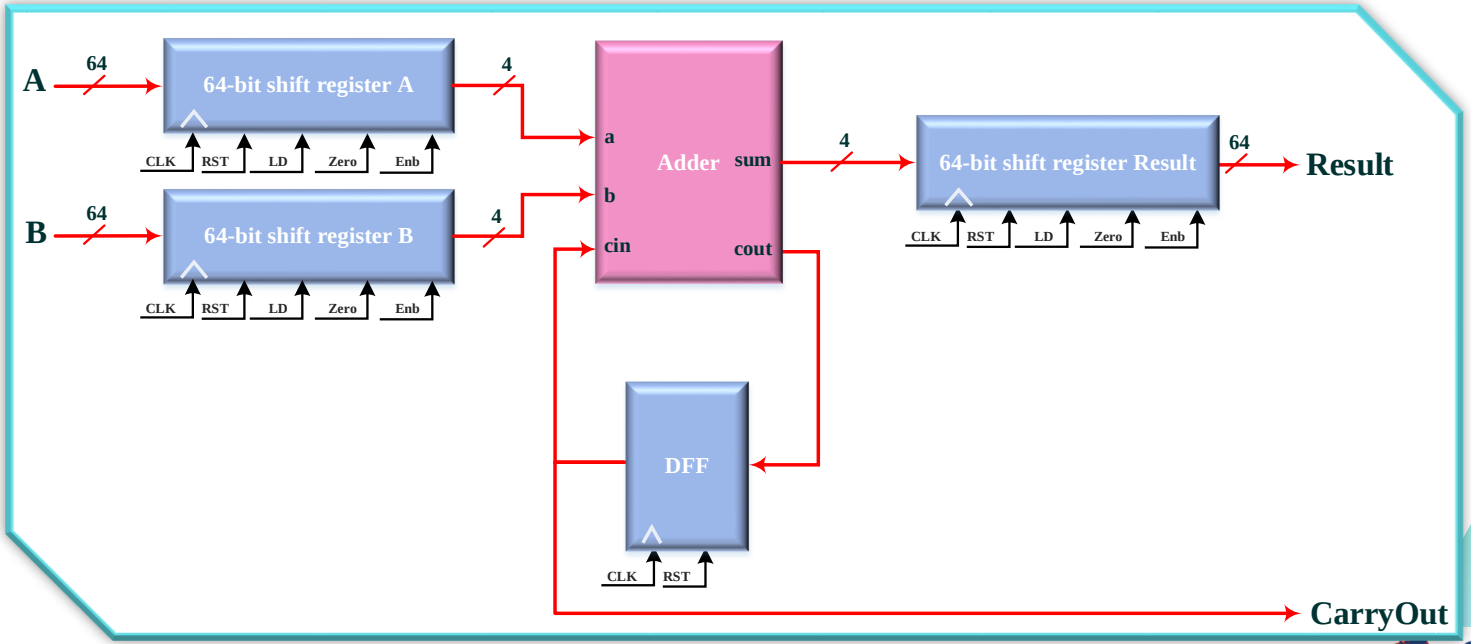
- Let's design a quad serial adder
- Our first RTL example, 64-bit unsigned adder using an expensive fast carry lookahead adder
- Serialized data into 8 4-bit chunks to be used with the adder

8- RTL Cores

Quad Serial Adder Circuit

- Datapath

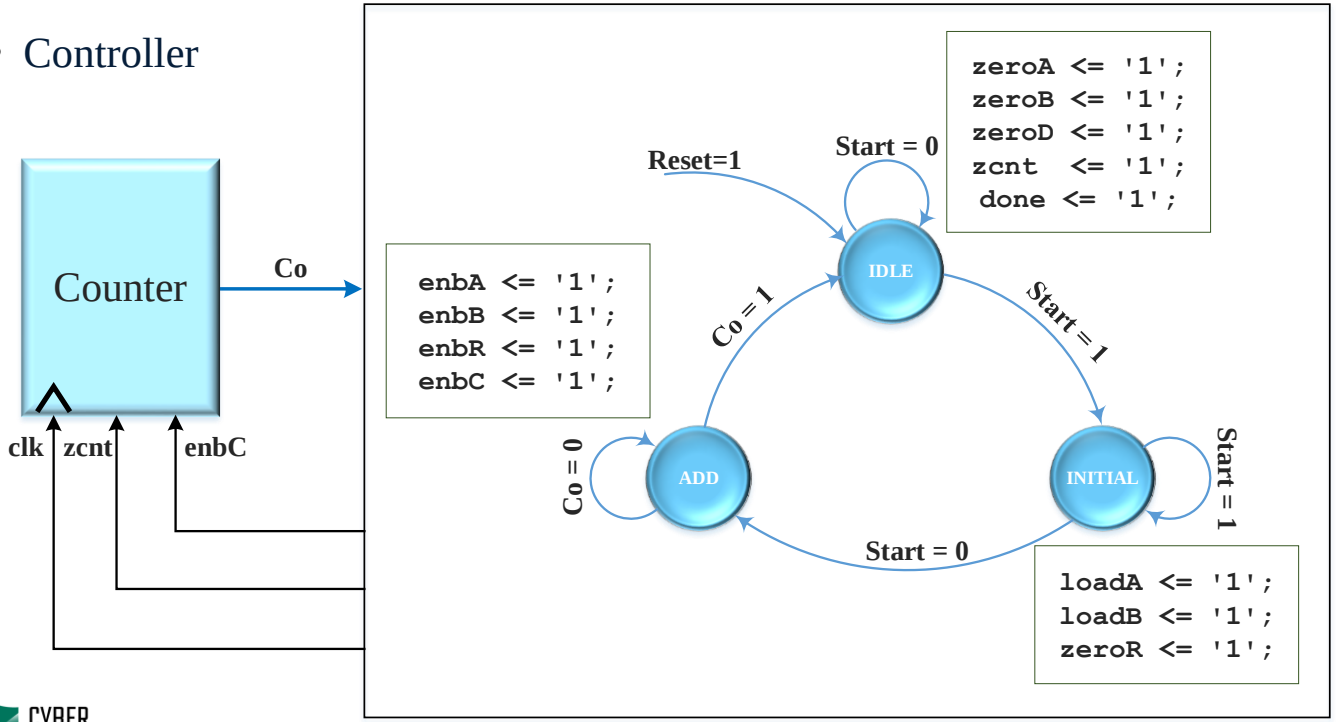
- The datapath has 64-bit shift registers for input and output, a 4-bit adder and a DFF for saving carry



8- RTL Cores

Quad Serial Adder Circuit

- Controller



- The controller has an FSM for start and stop and
- FSM for counting 4-bit chunks (nibble)

8- RTL Cores

Quad Serial Adder Circuit

- Adder

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3  USE IEEE.std_logic_unsigned.ALL;
4
5  ENTITY adder IS
6      PORT(
7          cin: IN  std_logic;
8          a  : IN  std_logic_vector(3 DOWNTO 0);
9          b  : IN  std_logic_vector(3 DOWNTO 0);
10         sum: OUT std_logic_vector(3 DOWNTO 0);
11         cout : OUT std_logic);
12 END adder;
13
14 ARCHITECTURE behavioral OF adder IS
15     SIGNAL tmp : std_logic_vector(4 DOWNTO 0);
16 BEGIN
17
18     tmp <= ('0' & a) + ('0' & b) + cin ;
19     sum <= tmp(3 DOWNTO 0);
20     cout  <= tmp(4);
21
22 END behavioral;
23
```

- Let's do the VHDL description
- Use concurrent signal assignments for describing the adder

8- RTL Cores

Quad Serial Adder Circuit

- DFF

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3
4  ENTITY DFF IS
5      PORT(
6          clk, rst, zero, d: IN  std_logic;
7          q : OUT std_logic);
8  END DFF;
9
10 ARCHITECTURE behavioral OF DFF IS
11 BEGIN
12
13 PROCESS (clk, rst)
14     BEGIN
15         IF (rst = '1') THEN
16             q <= '0';
17         ELSIF (clk = '1' AND clk'EVENT) THEN
18             IF (zero = '1') THEN
19                 q <= '0';
20             ELSE
21                 q <= d;
22             END IF;
23         END IF;
24     END PROCESS;
25 END behavioral;
```

- The DFF has an asynchronous reset
- This follows the register coding style presented before

8- RTL Cores

Quad Serial Adder Circuit

- 64-bit Shift Register

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY ShiftReg IS
    PORT (
        clk, rst, enb, load, zero : IN std_logic;
        filler : IN std_logic_vector (3 DOWNTO 0);
        d : IN std_logic_vector (63 DOWNTO 0);
        qout : OUT std_logic_vector (63 DOWNTO 0));
END ShiftReg;

ARCHITECTURE behavioral OF ShiftReg IS
    SIGNAL q : std_logic_vector (63 DOWNTO 0);
BEGIN
    PROCESS (clk, rst)
    BEGIN
        IF (rst = '1') THEN
            q <= (OTHERS => '0');
        ELSIF (clk = '1' AND clk'EVENT) THEN
            IF (zero = '1') THEN
                q <= (OTHERS => '0');
            ELSIF (load = '1') THEN
                q <= d;
            ElSIF (enb = '1') THEN
                q <= filler & q(63 DOWNTO 4);
            END IF;
        END IF;
    END PROCESS;
    qout <= q;
END behavioral;
```

- Input/output shift register description
- We use a single description for all shift registers
- Shifting is done in 4-bit chunks
- For the input shift register, we use “0000” as left hand filler
- The description follows the coding style presented before

8- RTL Cores

Quad Serial Adder Circuit

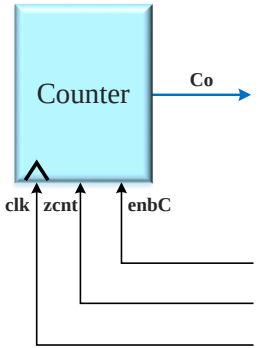
- Only use right most 4-bit slices of *qA* and *qB*
- Connect those to adder inputs
- Output register does not require a parallel input

- Datapath

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3
4  ENTITY Datapath IS
5      PORT(
6          clk, rst, enbA, enbB, enbR, loadA, loadB, loadR, zeroA, zeroB, zeroR, zeroD : IN std_logic;
7          dA, dB : IN  std_logic_vector(63 DOWNTO 0);
8          result : OUT std_logic_vector(63 DOWNTO 0);
9          carryOut : OUT std_logic);
10 END Datapath;
11
12 ARCHITECTURE behavioral OF Datapath IS
13     SIGNAL cin, cout : std_logic;
14     SIGNAL dR: std_logic_vector(3 DOWNTO 0);
15     SIGNAL qA, qB: std_logic_vector(63 DOWNTO 0);
16 BEGIN
17
18     ShRegA : ENTITY WORK.shiftReg PORT MAP (clk, rst, enbA, loadA, zeroA, "0000", dA, qA);
19     ShRegB : ENTITY WORK.shiftReg PORT MAP (clk, rst, enbB, loadB, zeroB, "0000", dB, qB);
20     ShRegR : ENTITY WORK.shiftReg PORT MAP (clk, rst, enbR, loadR, zeroR, dR, (OTHERS => '0'), result);
21     add    : ENTITY WORK.adder    PORT MAP (cin, qA(3 DOWNTO 0), qB(3 DOWNTO 0), dR, cout);
22     dff    : ENTITY WORK.DFF      PORT MAP (clk, rst, zeroD, cout, cin);
23     carryOut <= cin;
24 END behavioral;
```

Quad Serial Adder Circuit

- Counter



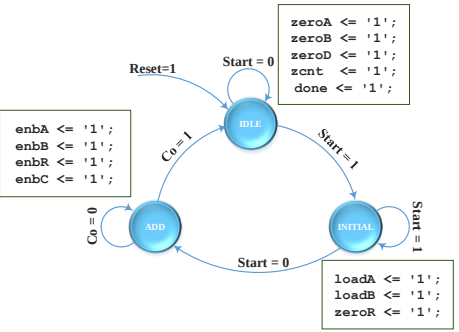

```vhdl
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY counter IS
    PORT (
        clk, rst, zero, load, enb : IN std_logic;
        d  : IN  std_logic_vector (3 DOWNTO 0);
        q  : OUT std_logic_vector (3 DOWNTO 0);
        co : OUT std_logic);
END counter;

ARCHITECTURE behavioral OF counter IS
    SIGNAL tmp : std_logic_vector (3 DOWNTO 0);
BEGIN
    PROCESS (clk, rst)
    BEGIN
        IF (rst= '1') THEN
            tmp <= (OTHERS => '0');
        ELSIF (clk = '1' AND clk'EVENT) THEN
            IF (zero= '1') THEN
                tmp <= (OTHERS => '0');
            ELSIF (load = '1') THEN
                tmp <= d;
            ELSIF (enb = '1') THEN
                tmp <= tmp + "0001";
            END IF;
        END IF;
    END PROCESS;
    q <= tmp;
    co <= '1' WHEN (tmp = "1111") ELSE '0' ;
END behavioral;
```

- This is the counter part of the controller
- A simple counter with a carry out that becomes one when count is “1111”

8- RTL Cores

Quad Serial Adder Circuit

- Controller

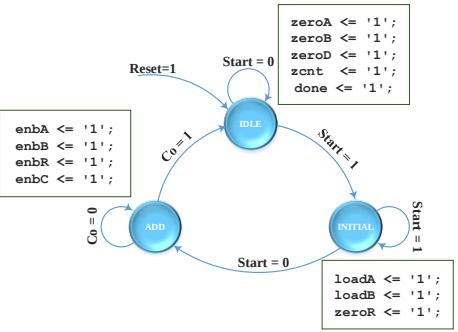



```vhdl
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY Controller IS
    PORT(
    clk, rst, start : IN std_logic;
    done, enbA, enbB, enbR, loadA, loadB, loadR, zeroA, zeroB, zeroR, zeroD : OUT std_logic);
END Controller;

ARCHITECTURE behavioral OF Controller IS
    TYPE state IS (Idle, Initialization, Add);
    SIGNAL p_state, n_state : state;

    SIGNAL count: std_logic_vector(3 DOWNTO 0);
    SIGNAL co, zcnt, loadC, enbC : std_logic;
BEGIN
    PROCESS (p_state, co, start) BEGIN
        CASE p_state IS
            WHEN Idle =>
                IF start='1' THEN n_state <= Initialization;
                ELSE n_state <= Idle; END IF;
            WHEN Initialization =>
                IF start='1' THEN n_state <= Initialization;
                ELSE n_state <= Add; END IF;
            WHEN Add =>
                IF co='1' THEN n_state <= Idle;
                ELSE n_state <= Add; END IF;
            WHEN OTHERS => n_state <= Idle;
        END CASE;
    END PROCESS;
```

- Putting the two parts of the controller together

Quad Serial Adder Circuit

- Controller

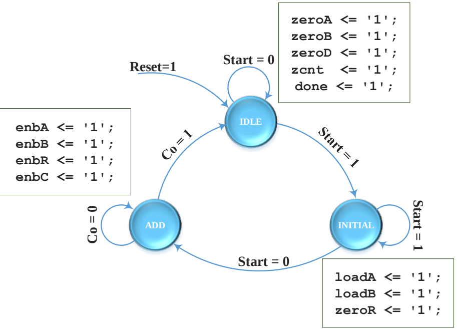


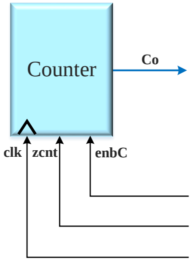

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

    PROCESS (p_state, co, start) BEGIN
        done <= '0'; enbA <= '0'; enbB <= '0'; enbR <= '0'; loadA <= '0';
        loadB <= '0'; loadR <= '0'; zeroA <= '0'; zeroB <= '0'; zeroR <= '0';
        zeroD <= '0'; zcnt <= '0'; loadC<= '0'; enbC<= '0';
        CASE p_state IS
            WHEN Idle => zeroA <= '1';
                         zeroB <= '1';
                         zeroD <= '1';
                         zcnt  <= '1';
                         done <= '1';


                                  loadB <= '1';
                                  zeroR <= '1';

            WHEN Add => enbA <= '1';
                        enbB <= '1';
                        enbR <= '1';
                        enbC <= '1';

            WHEN OTHERS =>
            done <= '0'; enbA <= '0'; enbB <= '0'; enbR <= '0'; loadA <= '0';
            loadB <= '0'; loadR <= '0'; zeroA <= '0'; zeroB <= '0'; zeroR <= '0';
            zeroD <= '0'; zcnt <= '0'; loadC<= '0'; enbC<= '0';
        END CASE;
    END PROCESS;
```

- Putting the two parts of the controller together

Quad Serial Adder Circuit

- Controller






```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
```

```
    PROCESS (p_state, co, start) BEGIN
        done <= '0'; enbA <= '0'; enbB <= '0'; enbR <= '0'; loadA <= '0';
        loadB <= '0'; loadR <= '0'; zeroA <= '0'; zeroB <= '0'; zeroR <= '0';
        zeroD <= '0'; zcnt <= '0'; loadC<= '0'; enbC<= '0';
        CASE p_state IS
            WHEN Idle => zeroA <= '1';
                         zeroB <= '1';
                         zeroD <= '1';
                         zcnt  <= '1';
                         done <= '1';


                                   loadB <= '1';
                                   zeroR <= '1';
```

```
    sequential: PROCESS (clk, rst) BEGIN
        IF rst = '1' THEN
            p_state <= Idle;
        ELSIF (clk = '1' AND clk'EVENT) THEN
            p_state <= n_state;
        END IF;
    END PROCESS sequential;

    cnt : ENTITY WORK.counter PORT MAP (clk, rst, zcnt, loadC, enbC, "0000", count, co);

END ARCHITECTURE;
```

- Putting the two parts of the controller together

Quad Serial Adder Circuit

- Quad Serial Adder

- Instantiation of datapath and controller to form the complete quad serial adder
- Control signals going between

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3
4  ENTITY QuadSerialAdder IS
5      PORT(
6          clk, rst, start : IN  std_logic;
7          dA, dB : IN  std_logic_vector(63 DOWNTO 0);
8          done   : OUT std_logic;
9          result : OUT std_logic_vector(63 DOWNTO 0);
10         carryOut : OUT std_logic);
11 END QuadSerialAdder;
12
13 ARCHITECTURE behavioral OF QuadSerialAdder IS
14     SIGNAL enbA, enbB, enbR, loadA, loadB, loadR, zeroA, zeroB, zeroR, zeroD: std_logic;
15 BEGIN
16
17     Datapath   : ENTITY WORK.Datapath   PORT MAP (clk, rst, enbA, enbB, enbR, loadA, loadB, loadR,
18                                                   zeroA, zeroB, zeroR, zeroD, dA, dB, result, carryOut);
19
20     Controller : ENTITY WORK.Controller PORT MAP (clk, rst, start, done, enbA, enbB, enbR,
21                                                   loadA, loadB, loadR, zeroA, zeroB, zeroR, zeroD);
22
23 END behavioral;
```

8- RTL Cores

Quad Serial Adder Circuit

- Testbench

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY QuadSerialAdder_TB IS
END QuadSerialAdder_TB;

ARCHITECTURE behavior OF QuadSerialAdder_TB IS

    --Inputs
    SIGNAL clk : std_logic := '0';
    SIGNAL rst : std_logic := '0';
    SIGNAL start : std_logic := '0';
    SIGNAL dA : std_logic_vector(63 DOWNTO 0) := (OTHERS => '0');
    SIGNAL dB : std_logic_vector(63 DOWNTO 0) := (OTHERS => '0');

    --Outputs
    SIGNAL done : std_logic;
    SIGNAL result : std_logic_vector(63 DOWNTO 0);
    SIGNAL carryOut : std_logic;

    -- Clock period definitions
    CONSTANT clk_period : time := 10 ns;

BEGIN
```

- Testbench has local signals to connect CUT inputs and outputs
- Testbench instantiates CUT
- Several processes in the testbench provide test data

8- RTL Cores

Quad Serial Adder Circuit

- Testbench

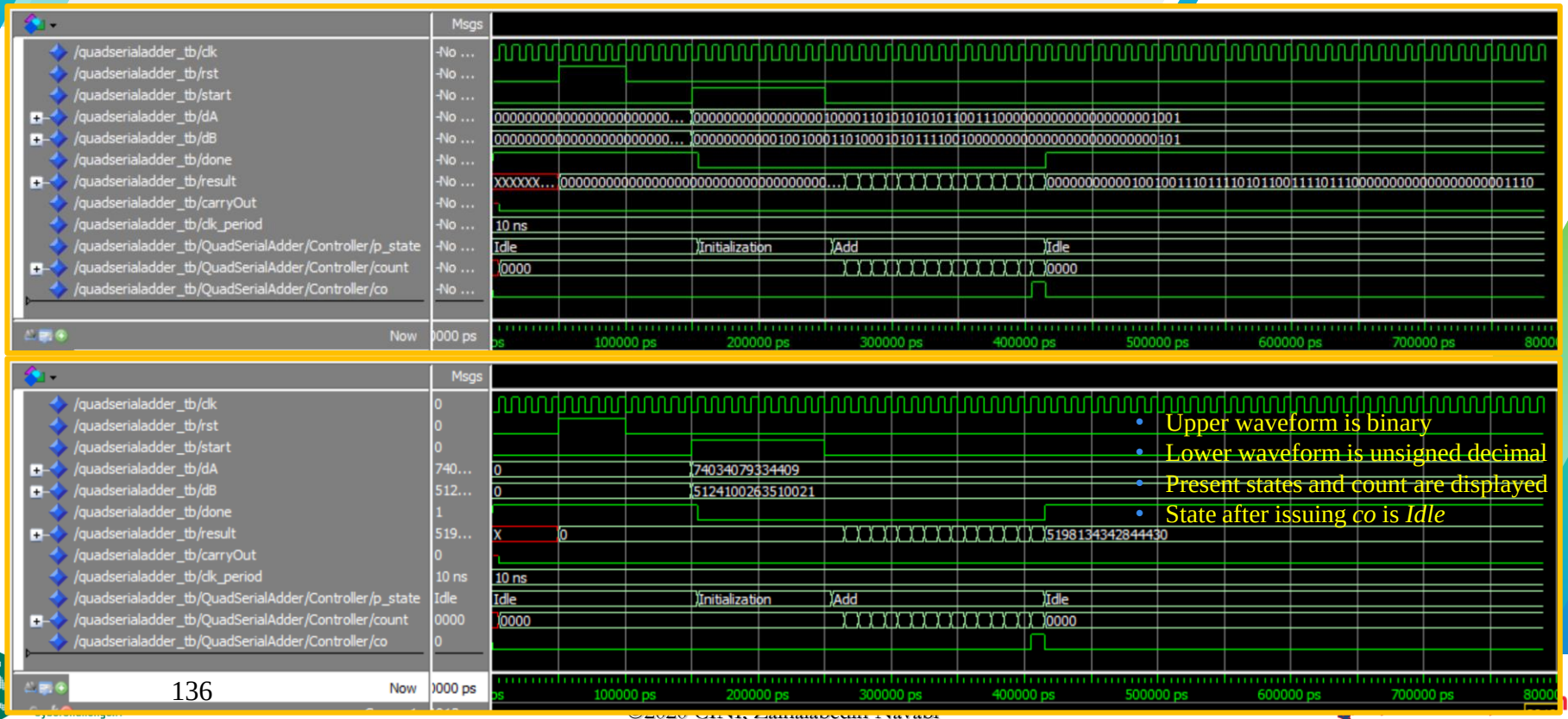
```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

BEGIN
    -- Instantiate the Unit Under Test (UUT)
    QuadSerialAdder : ENTITY WORK.QuadSerialAdder  PORT MAP (clk, rst, start, dA, dB, done, result, carryOut);

    -- Clock definitions
    clk <= NOT(clk) AFTER clk_period/2 WHEN NOW < 5000 ns;

    -- Stimulus process
    stim_proc: PROCESS
    BEGIN
        start <= '0';
        rst <= '0';
        WAIT FOR 50 ns;
        rst <= '1';
        WAIT FOR 50 ns;
        rst <= '0';
        WAIT FOR 50 ns;
        start <= '1';
        dA <= x"0000435567000009";
        dB <= x"0012345790000005";      -- HEX format

        WAIT FOR clk_period*10;
        start <= '0';

        WAIT;
    END PROCESS;
END;
```

- Testbench has local signals to connect CUT inputs and outputs
- Testbench instantiates CUT

Several processes in the testbench provide test data

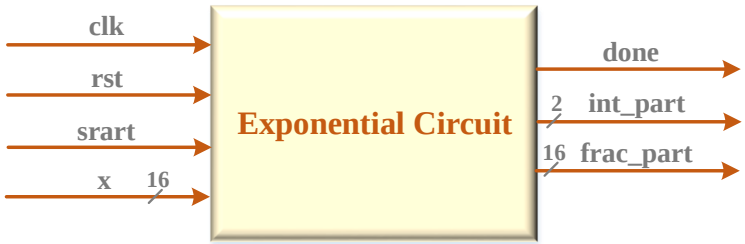
8- RTL Cores

Quad Serial Adder Waveforms



Exponential Circuit

- The circuit calculates e^x using Taylor expansion.
- The input is a 16-bit fixed-point number.
- The output is a 18-bit fixed-point number including 2 integer bits and 16 fractional bits.
- The circuit receives x as the input with the pulse on the start signal.
- The calculation continues for 8 iterations.
- When the result becomes ready, done signal will be issued.



Exponential Circuit

Input-output range

- With 16 bit input size, x is between 0.0000000000000000 for the smallest and 0.1111111111111111 for the largest value.
- Smallest output = 1 when e^0
- Largest output = 10.1000010110000010 (2.68357) for e^1

Exponential Circuit

- Taylor series

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

```
r = 1; // 1 for result
t = 1; // 1 for term
for( k = 1; k < n; k++ )
{
    t = t × x × ( 1 / i );
    r = r + t;
}
```

Sharing a multiplier

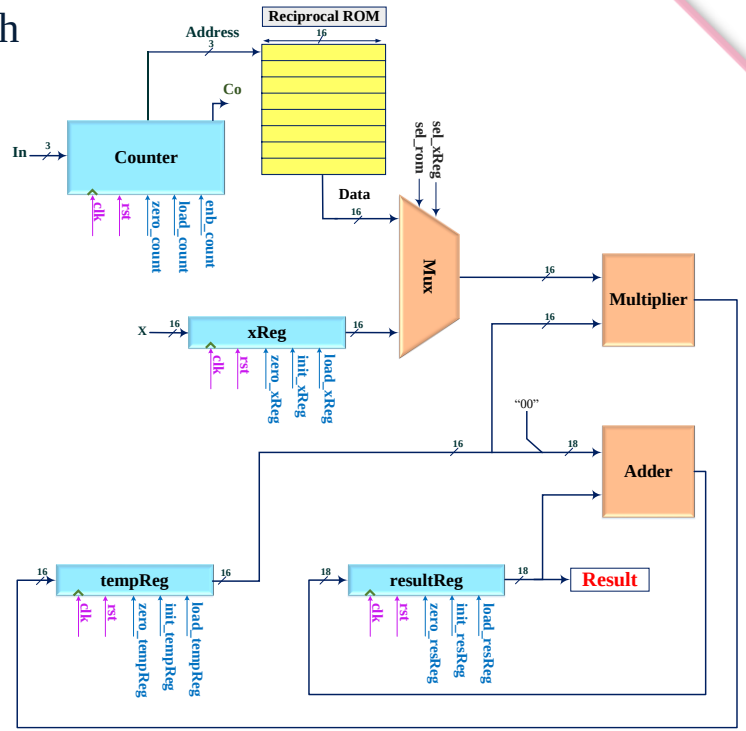
```
r = 1; // 1 for result
t = 1; // 1 for term
for( k = 1; k < n; k++ )
{
    t = t × x
    t = t × ( 1 / i );
    r = r + t;
}
```

- Implementation algorithm
- Optimize for a single multiplier

8- RTL Cores

Exponential Circuit

- DataPath

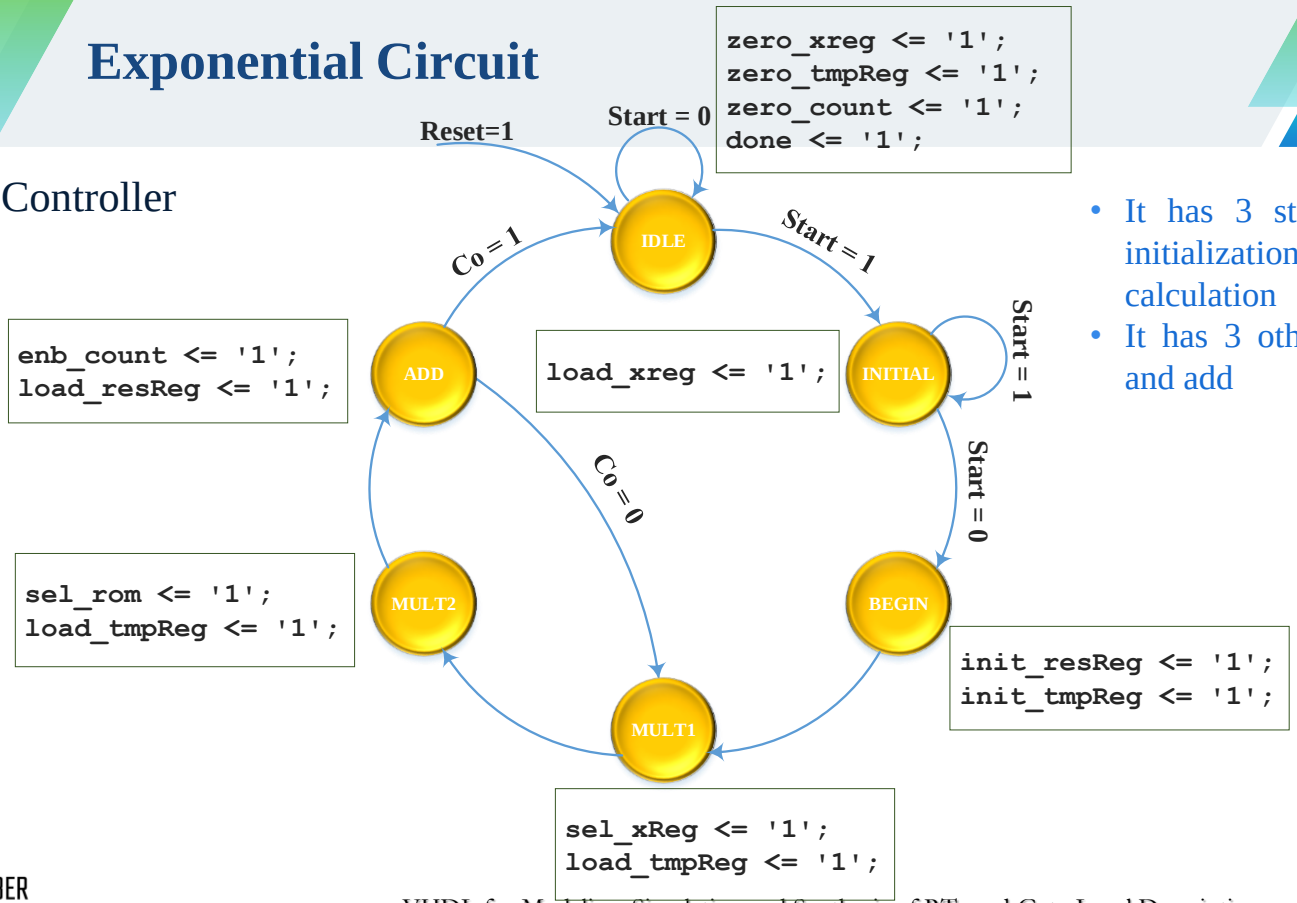


- *xReg* to hold the x input
- Reciprocal ROM to hold fractional $\frac{1}{i}$, i from 1 to 8
- Counter to index Reciprocal ROM
- A multiplier to do both multiplications
- A multiplexer selects *xReg* or Reciprocal ROM
- An adder adds terms of the series
- The result register will hold Taylor series result

8- RTL Cores

Exponential Circuit

- Controller



- It has 3 states for waiting for start, initialization and beginning series calculation
- It has 3 other states for multiplication and add

8- RTL Cores

Exponential Circuit

- Datapath

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE std.textio.ALL;
USE IEEE.std_logic_textio.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY Datapath IS
    PORT(
        clk, rst, zero_xReg, init_xReg, load_xReg, zero_tmpReg, init_tmpReg,
        load_tmpReg, zero_resReg, init_resReg, load_resReg, zero_count,
        enb_count, sel_xReg, sel_rom : IN std_logic;
        x : IN  std_logic_vector(15 DOWNTO 0);
        co : OUT std_logic;
        result : OUT std_logic_vector(17 DOWNTO 0));
END Datapath;

ARCHITECTURE behavioral OF Datapath IS

    SIGNAL mult : std_logic_vector(31 DOWNTO 0);
    SIGNAL result_in, tmp_result, tmp_add : std_logic_vector(17 DOWNTO 0);
    SIGNAL temp_out, rom_out, temp_in, xReg_out, mux_out : std_logic_vector(15 DOWNTO 0);
    SIGNAL tmp_count : std_logic_vector(2 DOWNTO 0);
    SIGNAL address : std_logic_vector(2 DOWNTO 0);

    TYPE memory IS ARRAY (0 TO 7) OF std_logic_vector(15 DOWNTO 0);
    SIGNAL data : memory;

BEGIN
```

- The datapath description contains all the datapath components
- Each component is described by one or two processes

8- RTL Cores

Exponential Circuit

- Datapath => 16-bit x-Regisrer

```
28  BEGIN
29
30      --xReg
31      PROCESS (clk, rst)
32      BEGIN
33          IF (rst= '1') THEN
34              xReg_out <= (OTHERS => '0');
35          ELSIF (clk = '1' AND clk'EVENT) THEN
36              IF (zero_xReg= '1') THEN
37                  xReg_out <= (OTHERS => '0');
38              ELSIF (init_xReg = '1') THEN
39                  xReg_out <= (OTHERS => '1');
40              ELSIF (load_xReg = '1') THEN
41                  xReg_out <= x;
42              END IF;
43          END IF;
44      END PROCESS;
```

- *xReg* is a simple register with asynchronous reset
- If needed, it will be initialized to all (1)
- This feature is not used

8- RTL Cores

Exponential Circuit

- Datapath => 16-bit temp-Regisrer

```
45
46      --tempReg
47      PROCESS (clk, rst)
48      BEGIN
49          IF (rst= '1') THEN
50              temp_out <= (OTHERS => '0');
51          ELSIF (clk = '1' AND clk'EVENT) THEN
52              IF (zero_tmpReg= '1') THEN
53                  temp_out <= (OTHERS => '0');
54              ELSIF (init_tmpReg = '1') THEN
55                  temp_out <= (OTHERS => '1');
56              ELSIF (load_tmpReg = '1') THEN
57                  temp_out <= temp_in;
58              END IF;
59          END IF;
60      END PROCESS;
```

- *tempReg* initializes to all (1), this function is needed for the first term of the series

8- RTL Cores

Exponential Circuit

- Datapath => 18-bit result-Register

```
62      --resultReg
63      PROCESS (clk, rst)
64      BEGIN
65          IF (rst= '1') THEN
66              tmp_result <= (OTHERS => '0');
67          ELSIF (clk = '1' AND clk'EVENT) THEN
68              IF (zero_resReg = '1') THEN
69                  tmp_result <= (OTHERS => '0');
70              ELSIF (init_resReg = '1') THEN
71                  tmp_result <= "0011111111111111";
72              ELSIF (load_resReg = '1') THEN
73                  tmp_result <= result_in;
74              END IF;
75          END IF;
76      END PROCESS;
```

- *resultReg* is a simple register with asynchronous reset
- This is fractional number closest possible to 1
- This register requires initialization to all 1's

8- RTL Cores

Exponential Circuit

- Datapath => Counter

```
78      --Counter
79      PROCESS (clk, rst)
80      BEGIN
81          IF (rst= '1') THEN
82              tmp_count <= (OTHERS => '0');
83          ELSIF (clk = '1' AND clk'EVENT) THEN
84              IF (zero_count= '1') THEN
85                  tmp_count <= (OTHERS => '0');
86              ELSIF (enb_count = '1') THEN
87                  tmp_count <= tmp_count + "001";
88              END IF;
89          END IF;
90      END PROCESS;
91      address  <= tmp_count;
92      co <= '1' WHEN (tmp_count >= "111") ELSE '0' ;
93
```

- This is a 3-bit counter
- A carry-out, *co*, becomes 1 when count is 111

8- RTL Cores

Exponential Circuit

- Datapath => Reciprocal ROM

```
94      --ROM
95      InitiateROM_proc: PROCESS
96          FILE fptr              : text;
97          VARIABLE fstatus       : file_open_status;
98          VARIABLE file_line     : line;
99          VARIABLE var_location  : integer;
100         VARIABLE var_space     : character;
101         VARIABLE var_data      : std_logic_vector(15 DOWNTO 0);
102     BEGIN
103         FILE_OPEN(fstatus, fptr, "myfile.txt", read_mode);
104         FOR i IN 0 TO 7 LOOP
105             data(i) <= (OTHERS => '0');
106         END LOOP;
107
108         WHILE NOT ENDFILE(fptr) LOOP
109             READLINE(fptr, file_line);
110             READ(file_line, var_location);
111             READ(file_line, var_space);
112             READ(file_line, var_data);
113             data(var_location) <= var_data;
114         END LOOP;
115         FILE_CLOSE(fptr);
116
117         WAIT;
118     END PROCESS;
119
120     rom_out <= data(conv_integer(address));
```

- Describing the Reciprocal ROM with integer indexing
- It is initialized by content of *myfile.txt*
- *rom_out* uses address from counter to reach ROM contents

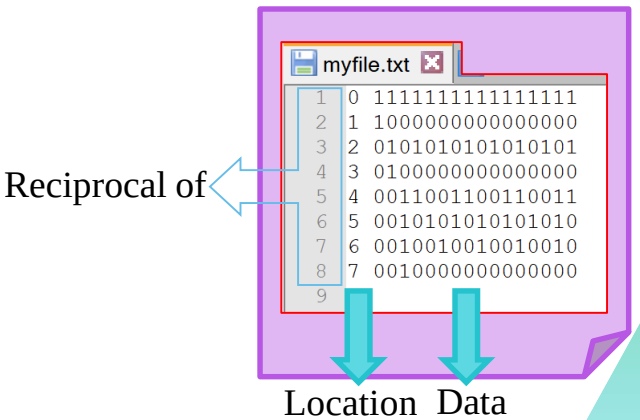
8- RTL Cores

Exponential Circuit

- Datapath => Reciprocal ROM

```
94      --ROM
95      InitiateROM_proc: PROCESS
96          FILE fptr              : text;
97          VARIABLE fstatus       : file_open_status;
98          VARIABLE file_line     : line;
99          VARIABLE var_location  : integer;
100         VARIABLE var_space     : character;
101         VARIABLE var_data      : std_logic_vector(15 DOWNTO 0);
102     BEGIN
103         FILE_OPEN(fstatus, fptr, "myfile.txt", read_mode);
104         FOR i IN 0 TO 7 LOOP
105             data(i) <= (OTHERS => '0');
106         END LOOP;
107
108         WHILE NOT ENDFILE(fptr) LOOP
109             READLINE(fptr, file_line);
110             READ(file_line, var_location);
111             READ(file_line, var_space);
112             READ(file_line, var_data);
113             data(var_location) <= var_data;
114         END LOOP;
115         FILE_CLOSE(fptr);
116
117         WAIT;
118     END PROCESS;
119
120     rom_out <= data(conv_integer(address));
```

- Describing the Reciprocal ROM with integer indexing
- It is initialized by content of *myfile.txt*
- *rom_out* uses address from counter to reach ROM contents



8- RTL Cores

Exponential Circuit

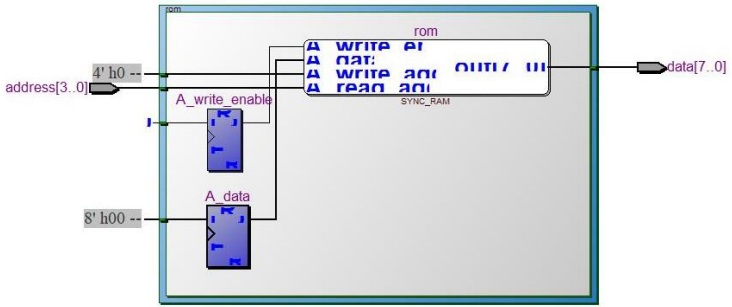
- ROM

- Use this coding style for synthesis using memory block

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY ROMsig IS
    PORT (address: IN integer RANGE 0 TO 7;
          data: OUT std_logic_vector(15 DOWNTO 0));
END ENTITY;

ARCHITECTURE sig OF ROMsig IS
    TYPE rom_array IS ARRAY (0 TO 7) OF std_logic_vector (15 DOWNTO 0);
    SIGNAL rom: rom_array := ( "1111111111111111", "1000000000000000", "0101010101010101", "0100000000000000",
                               "0011001100110011", "0010101010101010", "0010010010010010", "0010000000000000");
BEGIN
    data <= rom(address);
END ARCHITECTURE sig;
```

Exponential Circuit

- ROM=> Synthesis Output



8- RTL Cores

Exponential Circuit

- Datapath =>

- Multiplexer
- Multiplier
- Adder

```
122     --Multiplexer
123     mux_out <= xReg_out WHEN (sel_xReg = '1') ELSE
124                rom_out  WHEN (sel_rom  = '1') ELSE (OTHERS => 'X');
125
126     --Multiplier
127     mult <= mux_out * temp_out;
128     temp_in <= mult(31 DOWNTO 16) ;
129
130     --Adder
131     tmp_add <= "00" & temp_out;
132     result_in <= tmp_result + tmp_add;
133
134     --
135     result <= tmp_result;
136 END behavioral;
```

- Describing the combinational parts; Multiplexer, Multiplier and Adder
- Line 135 describes bussing

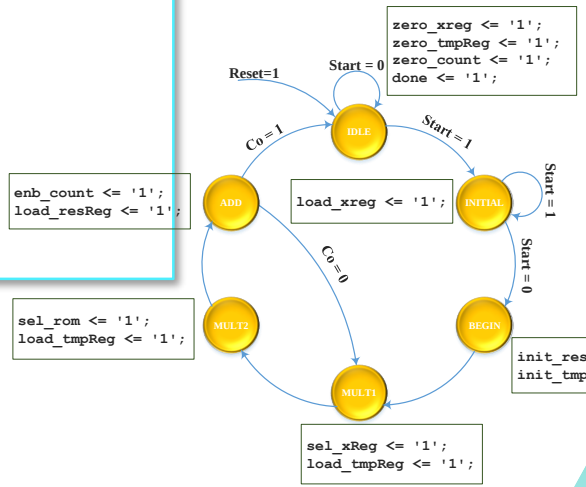
8- RTL Cores

Exponential Circuit

- Controller

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3
4  ENTITY Controller IS
5      PORT(
6          clk, rst, start, co: IN  std_logic;
7          done, zero_xReg, init_xReg, load_xReg, zero_tmpReg, init_tmpReg,
8          load_tmpReg, zero_resReg, init_resReg, load_resReg, zero_count,
9          enb_count, sel_xReg, sel_rom : OUT  std_logic
10       );
11 END Controller;
12
13 ARCHITECTURE behavioral OF Controller IS
14     TYPE state IS (Idle, Initialization, Beg, Mult1, Mult2, Add);
15     SIGNAL p_state, n_state : state;
16 BEGIN
```

- Use Huffman model, follow synthesis rules for the combinational and sequential parts
- The controller has asynchronous reset



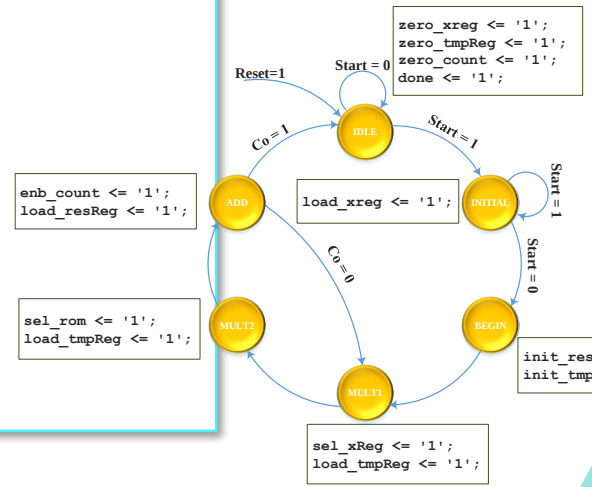
8- RTL Cores

Exponential Circuit

- Controller

```
LIBRARY IEEE;
BEGIN
    PROCESS (p_state, co, start) BEGIN
        CASE p_state IS
            WHEN Idle =>
                IF start='1' THEN n_state <= Initialization;
                ELSE n_state <= Idle; END IF;
            WHEN Initialization =>
                IF start='1' THEN n_state <= Initialization;
                ELSE n_state <= Beg; END IF;
            WHEN Beg =>
                n_state <= Mult1;
            WHEN Mult1 =>
                n_state <= Mult2;
            WHEN Mult2 =>
                n_state <= Add;
            WHEN Add =>
                IF co='1' THEN n_state <= Idle;
                ELSE n_state <= Mult1; END IF;
            WHEN OTHERS => n_state <= Idle;
        END CASE;
    END PROCESS;
```

- Use Huffman model, follow synthesis rules for the combinational and sequential parts
- The controller has asynchronous reset



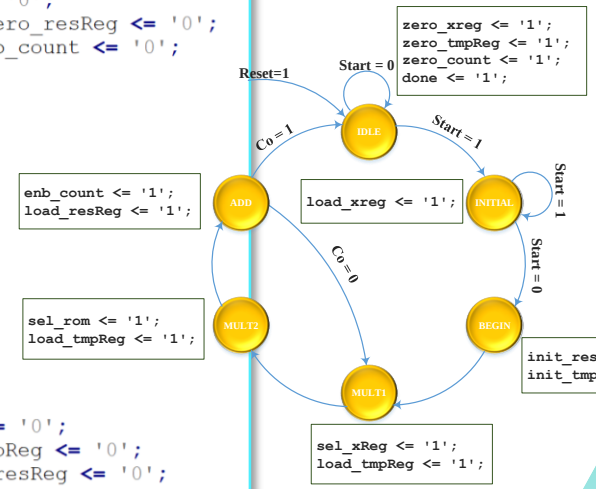
8- RTL Cores

Exponential Circuit

- Controller

```
LIBRARY IEEE;
BEGIN
    PROCESS (p_state, co, start) BEGIN
        done <= '0'; zero_xreg <= '0'; init_xreg <= '0'; load_xreg <= '0';
        zero_tmpReg <= '0'; init_tmpReg <= '0'; load_tmpReg <= '0'; zero_resReg <= '0';
        init_resReg <= '0'; load_resReg <= '0'; zero_count <= '0'; enb_count <= '0';
        sel_xReg <= '0'; sel_rom <= '0';
        CASE p_state IS
            WHEN Idle => zero_xreg <= '1';
                         zero_tmpReg <= '1';
                         zero_count <= '1';
                         done <= '1';
            WHEN Initialization => load_xreg <= '1';
            WHEN Beg => init_resReg <= '1';
                        init_tmpReg <= '1';
            WHEN Mult1 => sel_xReg <= '1';
                          load_tmpReg <= '1';
            WHEN Mult2 => sel_rom <= '1';
                          load_tmpReg <= '1';
            WHEN Add => enb_count <= '1';
                        load_resReg <= '1';
            WHEN OTHERS => done <= '0'; zero_xreg <= '0'; init_xreg <= '0';
                           load_xreg <= '0'; zero_tmpReg <= '0'; init_tmpReg <= '0';
                           load_tmpReg <= '0'; zero_resReg <= '0'; init_resReg <= '0';
                           load_resReg <= '0'; zero_count <= '0'; enb_count <= '0';
                           sel_xReg <= '0'; sel_rom <= '0';
        END CASE;
    END PROCESS;
```

- Use Huffman model, follow synthesis rules for the combinational and sequential parts
- The controller has asynchronous reset



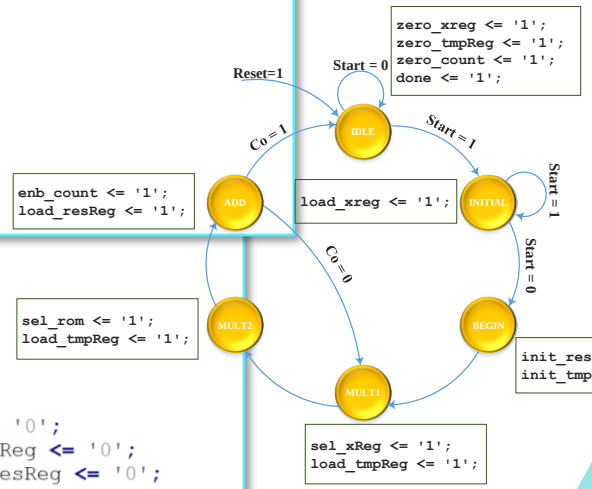
8- RTL Cores

Exponential Circuit

- Controller

```
LIBRARY IEEE;
BEGIN
    PROCESS (p_state, co, start) BEGIN
        sequential: PROCESS (clk, rst) BEGIN
            IF rst = '1' THEN
                p_state <= Idle;
            ELSIF (clk = '1' AND clk'EVENT) THEN
                p_state <= n_state;
            END IF;
        END PROCESS sequential;

END ARCHITECTURE;
                    init_tmpReg <= '1';
            WHEN Mult1 => sel_xReg <= '1';
                    load_tmpReg <= '1';
            WHEN Mult2 => sel_rom <= '1';
                    load_tmpReg <= '1';
            WHEN Add => enb_count <= '1';
                    load_resReg <= '1';
            WHEN OTHERS => done <= '0'; zero_xreg <= '0'; init_xreg <= '0';
                    load_xreg <= '0'; zero_tmpReg <= '0'; init_tmpReg <= '0';
                    load_tmpReg <= '0'; zero_resReg <= '0'; init_resReg <= '0';
                    load_resReg <= '0'; zero_count <= '0'; enb_count <= '0';
                    sel_xReg <= '0'; sel_rom <= '0';
        END CASE;
    END PROCESS;
```

- Use Huffman model, follow synthesis rules for the combinational and sequential parts
- The controller has asynchronous reset



8- RTL Cores

Exponential Circuit

- Exponential

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY Exponential IS
    PORT(
        clk, rst, start : IN  std_logic;
        x : IN std_logic_vector(15 DOWNTO 0);
        done : OUT std_logic;
        int_part  : OUT std_logic_vector(1 DOWNTO 0);
        frac_part : OUT std_logic_vector(15 DOWNTO 0));
END Exponential;

ARCHITECTURE behavioral OF Exponential IS
    SIGNAL  co, zero_xReg, init_xReg, load_xReg, zero_tmpReg, init_tmpReg, load_tmpReg,
            zero_resReg, init_resReg, load_resReg, zero_count, enb_count, sel_xReg,
            sel_rom : std_logic;
    SIGNAL  int_frac : std_logic_vector(17 DOWNTO 0);
BEGIN
    Datapath   : ENTITY WORK.Datapath   PORT MAP (clk, rst, zero_xReg, init_xReg, load_xReg,
                                                  zero_tmpReg, init_tmpReg, load_tmpReg, zero_resReg,
                                                  init_resReg, load_resReg, zero_count, enb_count,
                                                  sel_xReg, sel_rom, x, co, int_frac);
    Controller : ENTITY WORK.Controller PORT MAP (clk, rst, start, co, done, zero_xReg, init_xReg,
                                                  load_xReg, zero_tmpReg, init_tmpReg, load_tmpReg,
                                                  zero_resReg, init_resReg, load_resReg, zero_count,
                                                  enb_count, sel_xReg, sel_rom);
    int_part  <= int_frac(17 DOWNTO 16);
    frac_part <= int_frac(15 DOWNTO 0);
END behavioral;
```

- Instantiate datapath and controller
- Control signals connect between the two
- That completes exponential circuit description

8- RTL Cores

Exponential Circuit

- Testbench

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY Exponential_TB IS
END Exponential_TB;

ARCHITECTURE behavior OF Exponential_TB IS

    --Inputs
    SIGNAL clk : std_logic := '0';
    SIGNAL rst : std_logic := '0';
    SIGNAL start : std_logic := '0';
    SIGNAL x : std_logic_vector(15 DOWNTO 0) := (OTHERS => '0');

    --Outputs
    SIGNAL done : std_logic;
    SIGNAL intpart : std_logic_vector(1 DOWNTO 0);
    SIGNAL fracpart : std_logic_vector(15 DOWNTO 0);

    -- Clock period definitions
    CONSTANT clk_period : time := 10 ns;

BEGIN
```

- Testing the exponential design

8- RTL Cores

Exponential Circuit

- Testbench

```
LIBRARY IEEE;
USE IEEE.std logic 1164.ALL;
EN
EN
AR
BE
```

```
BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: ENTITY WORK.Exponential PORT MAP(clk, rst, start, x, done, intpart, fracpart);

    -- Clock definitions
    clk <= NOT(clk) AFTER clk_period/2 WHEN NOW < 5000 ns;

    -- Stimulus process
    stim_proc: PROCESS
    BEGIN
        rst <= '0';
        WAIT FOR 20 ns;
        rst <= '1';
        WAIT FOR 50 ns;
        rst <= '0';
        WAIT FOR 15 ns;
        start <= '1';
        WAIT FOR clk_period*10;
        start <= '0';
        x <= "1000000000000000";

        WAIT;
    END PROCESS;
END;
```

- Testing the exponential design

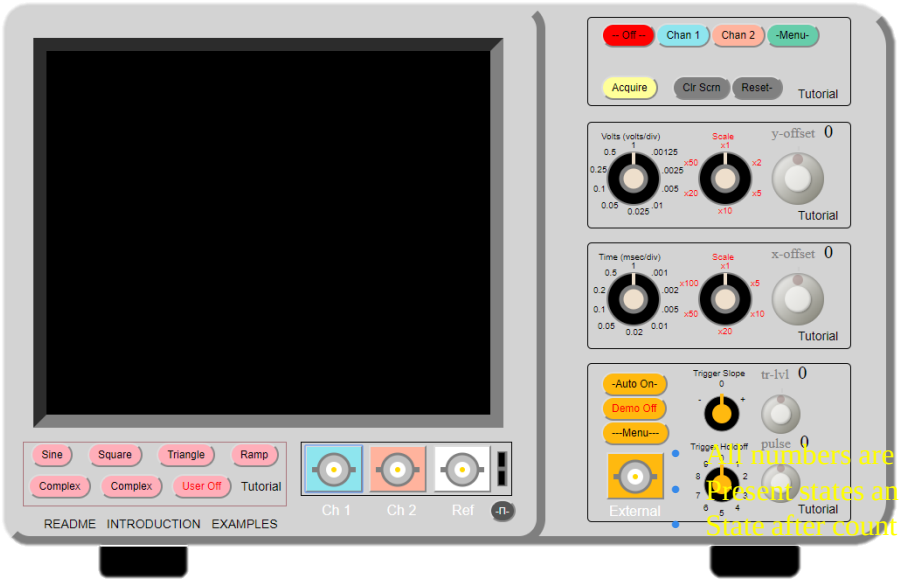
8- RTL Cores

Simulation Waveform

- Simulation results confirm the correct operation of exponential circuit
- The circuit is completely synthesizable

Calculation : $e^{0.5} = 1.648721271$

Simulation : $e^{0.5} = 1.6486053467$



All numbers are in fixed point binary
Present states and count are displayed
State after count of 111 is *Idle*

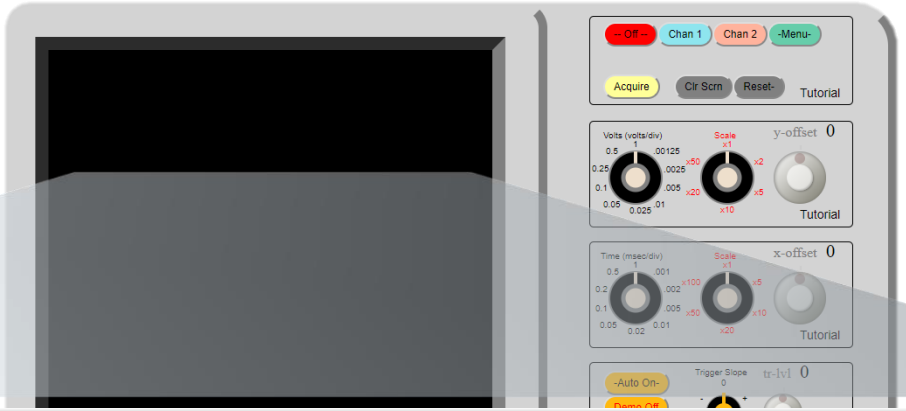
8- RTL Cores

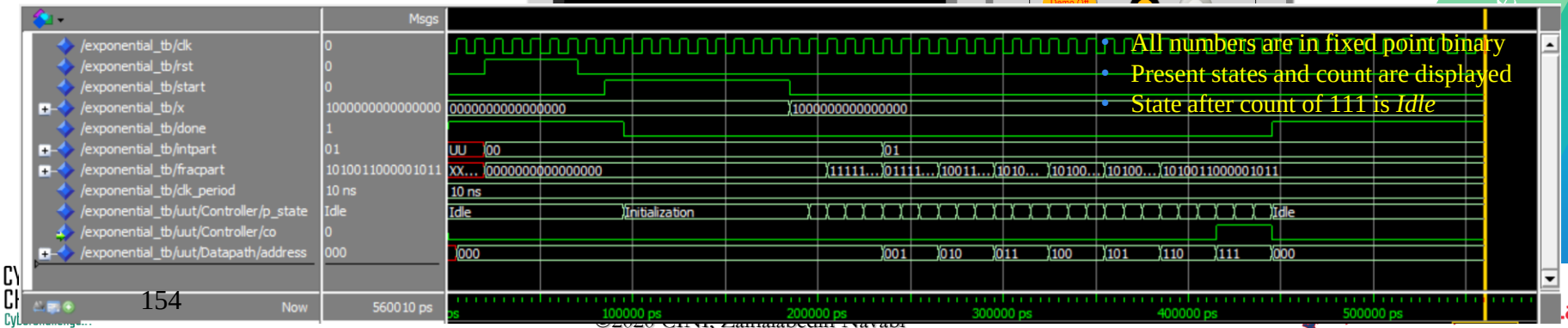
Simulation Waveform

- Simulation results confirm the correct operation of exponential circuit
- The circuit is completely synthesizable

Calculation : $e^{0.5} = 1.648721271$

Simulation : $e^{0.5} = 1.6486053467$

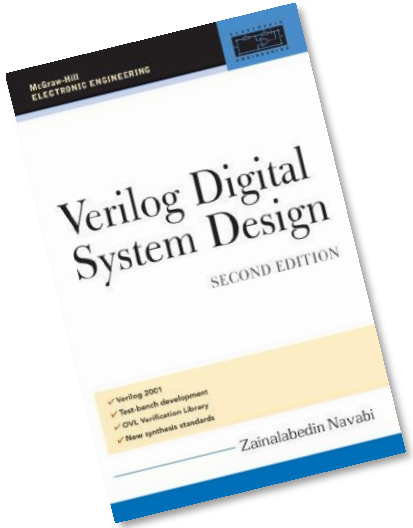


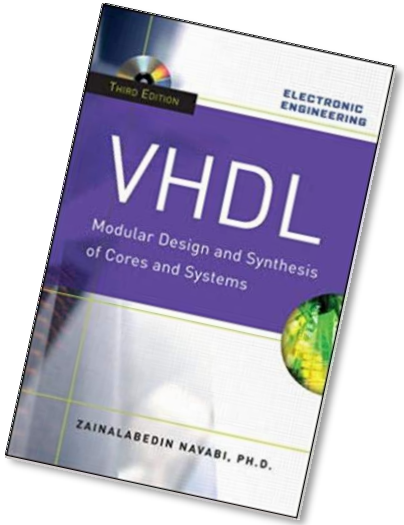


- All numbers are in fixed point binary
- Present states and count are displayed
- State after count of 111 is *Idle*

References

- Navabi, Zainalabedin. “VHDL: Modular Design and Synthesis of Cores and Systems.” McGraw Hill , 2007.
- Navabi, Zainalabedin. “Verilog digital system design.” McGraw Hill, 2005.





- **All content is available in the book and you can refer to the book for more information.**

Acknowledgment

- Slides by Maryam Rajabalipanah

Thanks!

SARV-E ABARKUH:
IRAN'S 4500-YEAR-OLD CYPRESS

Any questions?

You can find me at:

navabi@wpi.edu

navabi@ut.ac.ir