Single Clock Domain Designs

Reset and Initialization

Modelling FSMs in VHDL

LECTURE 4

IOULIIA SKLIAROVA

Universidade de Aveiro

Single Clock Domain Designs

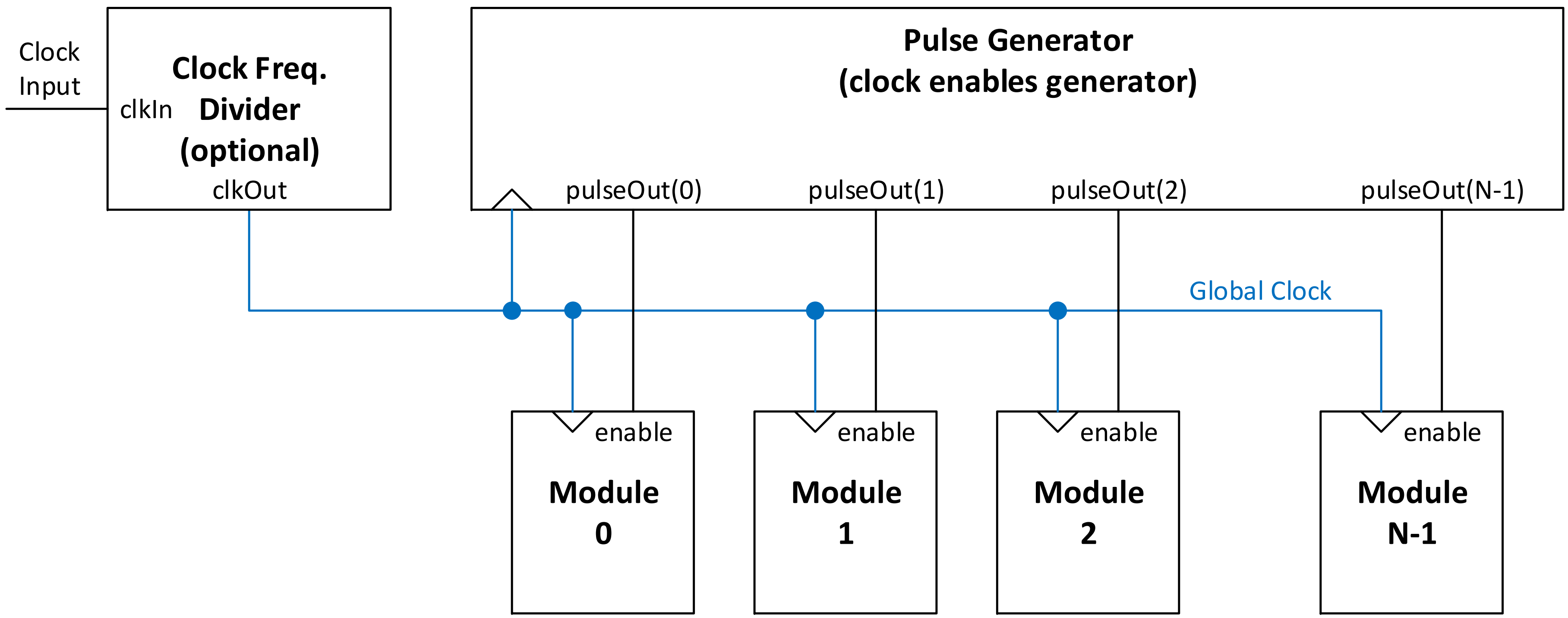
A **clock domain** is the subset of the system components that are synchronized by a single clock signal.

Utilization of two or more clock domains in a system is frequently required but can lead to complex timing issues.

Recommendation: in all your projects you should:

- Use only the “clk” clock signal, or other clock derived from it (using a clock frequency divider or a clock IP).
- Use a single clock signal in conjunction with enable pulses to synchronize/sequence slower operations.
- All the components are synchronized by the same clock signal and each one has its own enable(s).

Single Clock Domain with Enables



Example of a Pulse Generator

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity pulse_gen is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           pulse : out STD_LOGIC);
end pulse_gen;

architecture Behavioral of pulse_gen is
    constant MAX : natural := 100_000_000;
    signal s_cnt : natural range 0 to MAX-1;
begin

process(clk)
begin
    if (rising_edge(clk)) then
        pulse <= '0';
        if (reset = '1') then
            s_cnt <= 0;
        else
            s_cnt <= s_cnt + 1;
            if (s_cnt = MAX-1) then
                s_cnt <= 0;
                pulse <= '1';
            end if;
        end if;
    end if;
end process;

end Behavioral;
```

What is the active duration the output `pulse`?

What is the frequency of `pulse` output?

Example of a Pulse Generator

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity generator is
    generic(NUMBER_STEPS : positive := 50_000_000);
    Port ( clk   : in STD_LOGIC;
           reset : in STD_LOGIC;
           blink: out STD_LOGIC);
end generator;

architecture Behavioral of generator is
    signal s_counter : natural range 0 to NUMBER_STEPS-1;
begin

count_proc: process(clk)
begin
    if rising_edge(clk) then
        if (reset = '1') or (s_counter >= NUMBER_STEPS-1) then
            s_counter <= 0;
        else
            s_counter <= s_counter + 1;
        end if;
        blink <= '1' when s_counter >= (NUMBER_STEPS/2) else '0'; -- VHDL-2008 !
    end if;
end process;

end Behavioral;
```

What is the duty-cycle of the output `blink`?

What is the frequency of `blink` output?

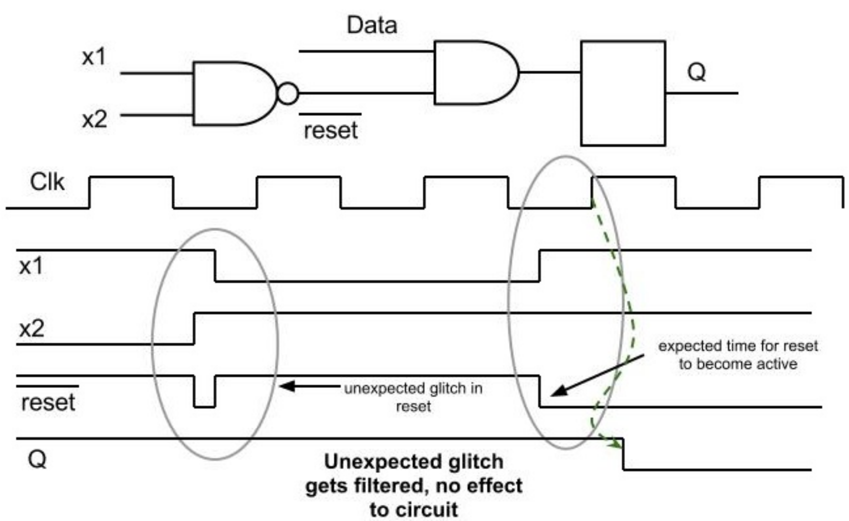
Initialization and Reset

Most sequential circuits require the initialization of their memory elements (e.g. FSM state register, counters, accumulators, etc.)

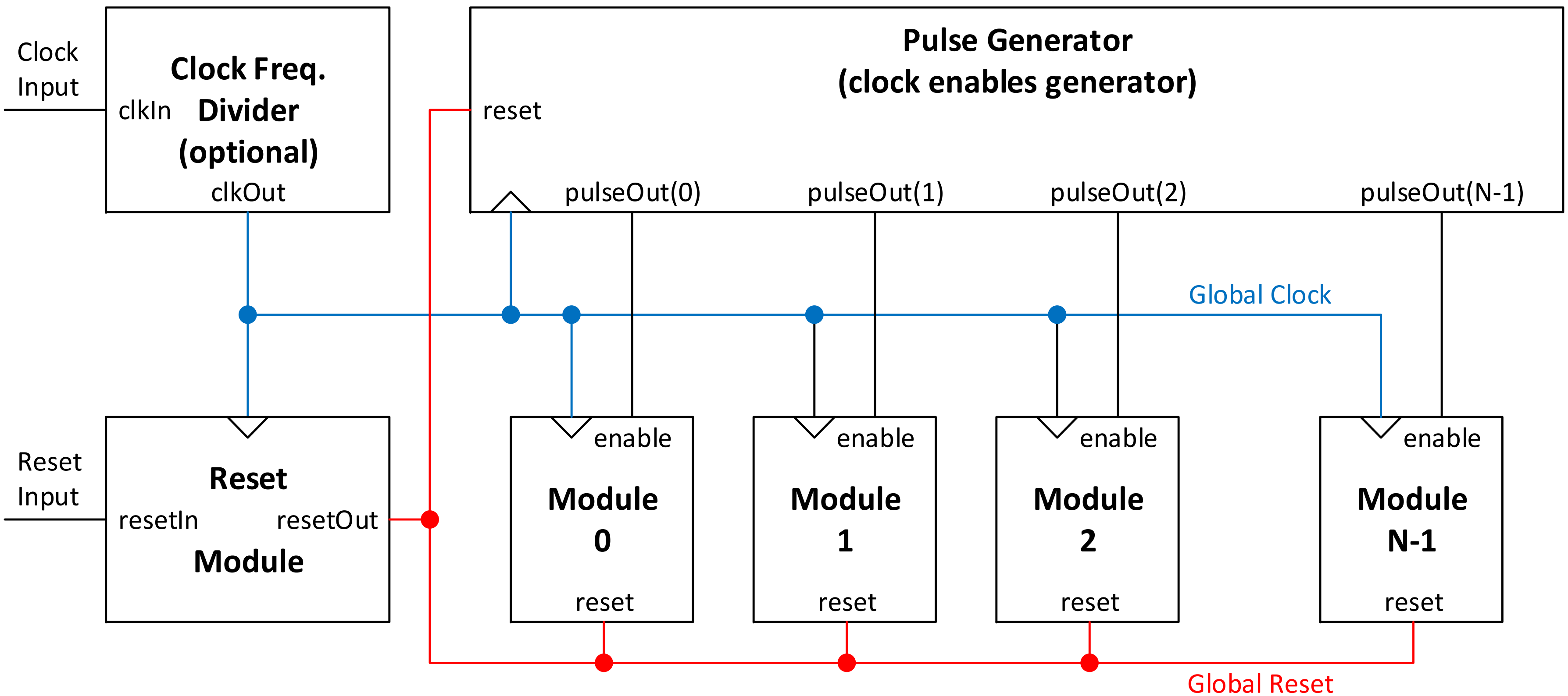
The initialization must be performed

- at system boot / after FPGA programming
- whenever needed, through the activation of global or local reset signals

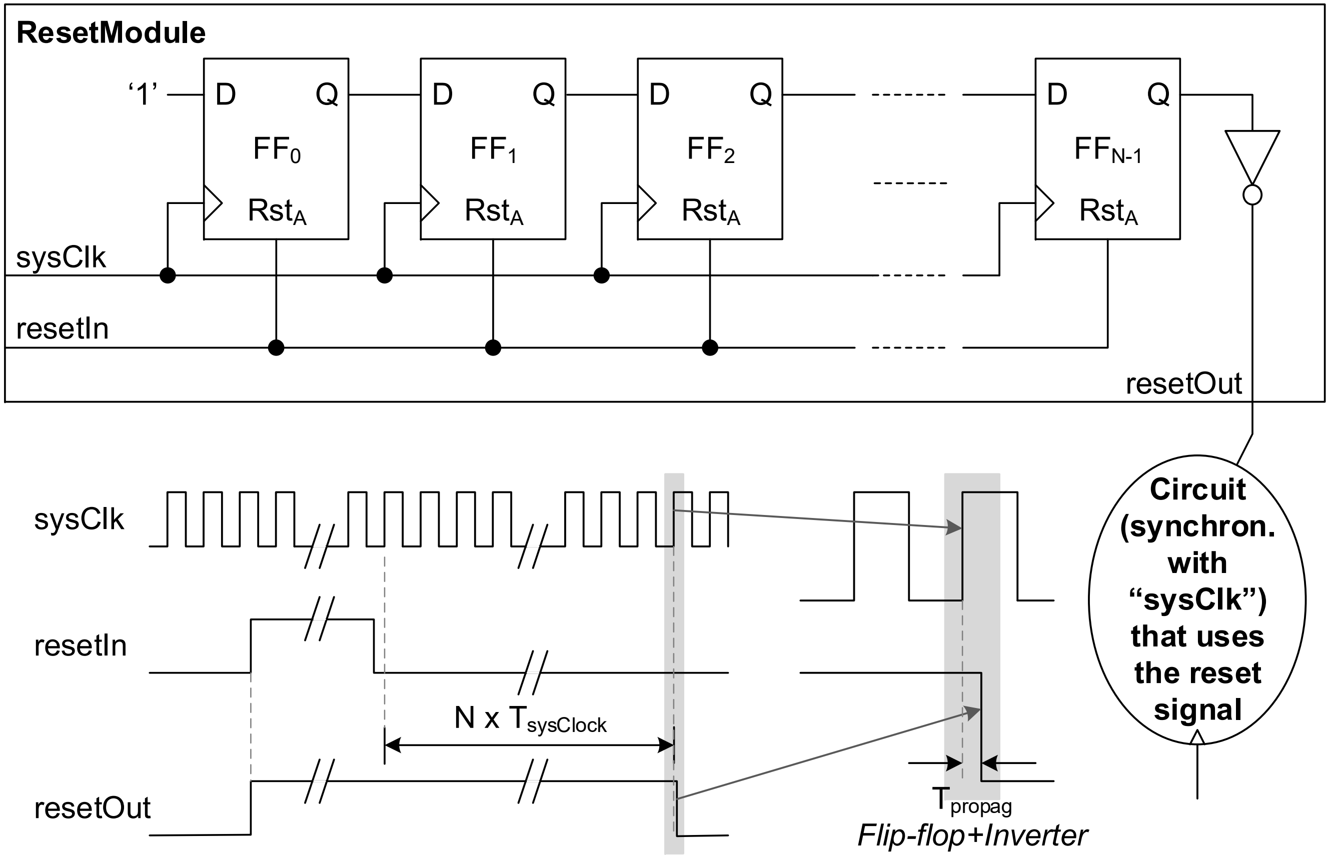
Use of asynchronous reset can easily create circuits that glitch => synchronous reset components must be preferred



Single Clock Domain with Enables and Reset



Example of the Reset Module



After FPGA programming, all the FFs are loaded with 0's and the module activates the reset output

All the system components must use preferably synchronous resets

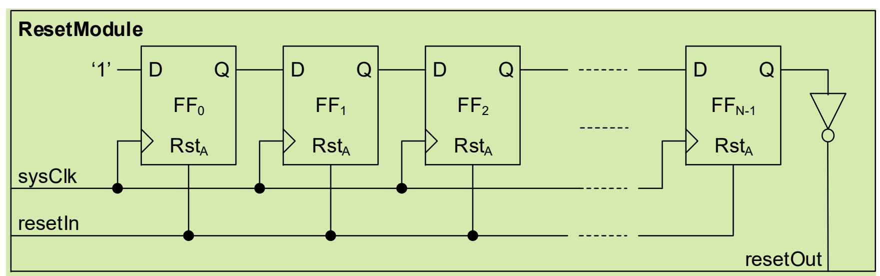
The clock period and the number of flips-flops ensure a minimum reset activation time

Example of the Reset Module

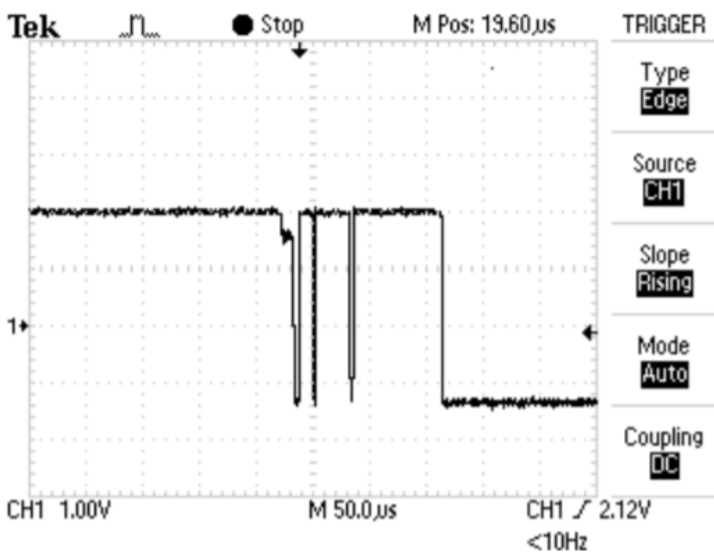
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity ResetModule is
    generic(N       : positive := 4);
    port(sysClk     : in  std_logic;
         resetIn    : in  std_logic;
         resetOut   : out std_logic);
end ResetModule;

architecture Behavioral of ResetModule is

    signal s_shiftReg : std_logic_vector((N - 1) downto 0) := (others => '0');

begin
    assert(N >= 2);

    shift_proc : process(resetIn, sysClk)
    begin
        if (resetIn = '1') then
            s_shiftReg <= (others => '0');
        elsif (rising_edge(sysClk)) then
            s_shiftReg((N - 1) downto 1) <= s_shiftReg((N - 2) downto 0);
            s_shiftReg(0) <= '1';
        end if;
    end process;

    resetOut <= not s_shiftReg(N - 1);
end Behavioral;
```

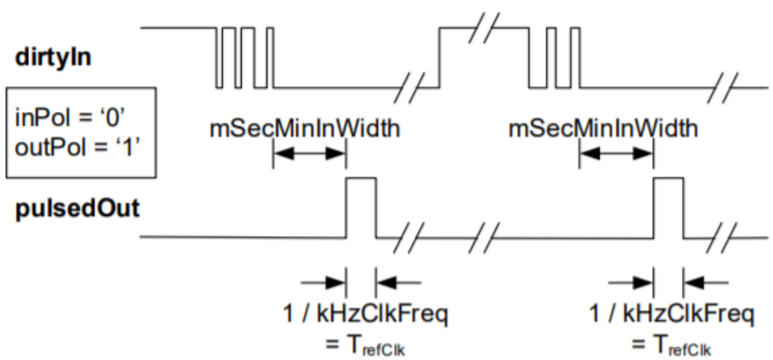
Generates a reset pulse, with the duration ~N x **`sysClk`** periods

Initialization of the **`s_shiftReg`** signal during FPGA programming



Input Debouncing





E-learning:

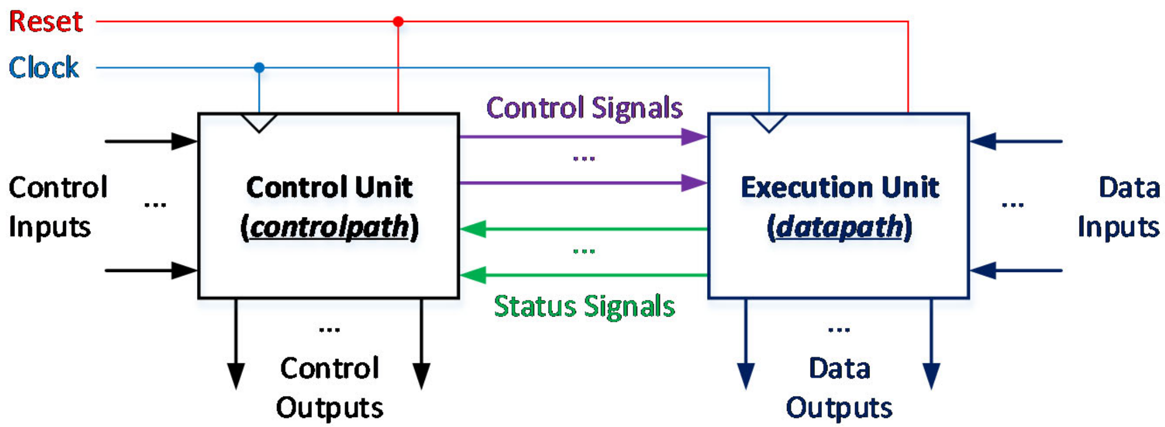
```
entity DebounceUnit is
    generic(kHzClkFreq      : positive := 50000;
            mSecMinInWidth  : positive := 100;
            inPolarity      : std_logic := '0';
            outPolarity     : std_logic := '1');
    port(refClk    : in  std_logic;
         dirtyIn   : in  std_logic;
         pulsedOut : out std_logic);
end DebounceUnit;
```

```
debounce_BTNC:
   entity work.DebounceUnit(Behavioral)
     generic map (kHzClkFreq => 100_000,
                  mSecMinInWidth => 100,
                  inPolarity => '1',
                  outPolarity => '1')
     port map(refClk => clk,
         dirtyIn => btnC,
         pulsedOut => s_startStop);
```

Computational System

Datapath (execution unit)

- Components
  - Functional
  - Routing
  - Storage



Controlpath

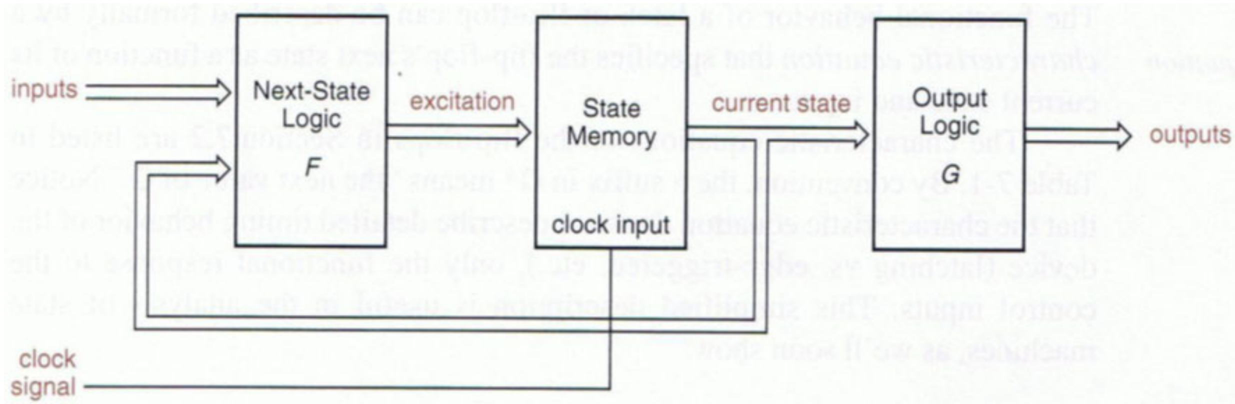
- Control unit
  - FSM(s)

Controlpath - datapath interconnection

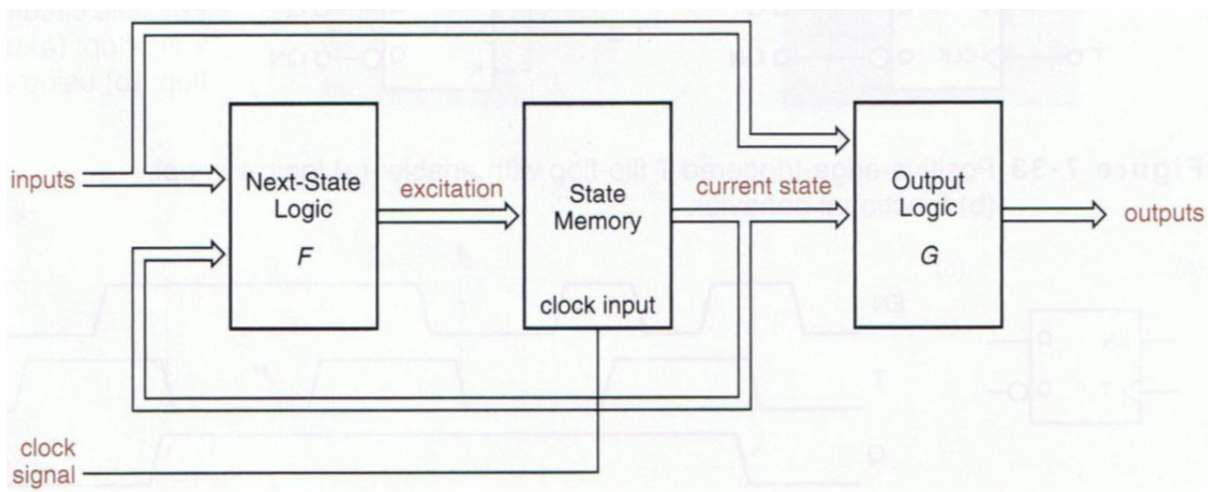
- Control signals (controlpath → datapath)
- Status signals (controlpath ← datapath

FSM Structure

Moore:



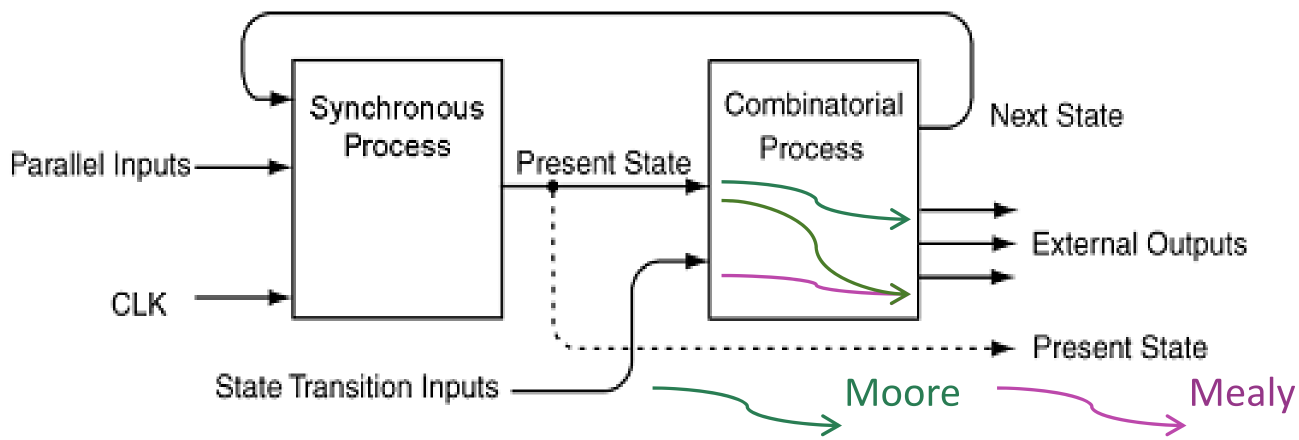
Mealy:



Modelling FSMs in VHDL

Two VHDL processes:

- State memory (synchronous process)
- Combinational circuit (next state logic + output logic)
  - Depending on the way outputs are assigned
    - *Moore* (outputs depend only on the current state)
    - *Mealy* (outputs depend on both the current state and FSM inputs)
  - Ensure a value is always assigned to next state and outputs
    - Must be a combinational circuit – no latches!!!



VHDL Coding

VHDL coding:

- There exist many different styles.
- The style explained here considers two processes: one for the state transitions, and another for the outputs.

Required steps:

- Have your state-transition diagram ready.
- The coding then just simply follows the state diagram.
- We need to define a custom user data type (e.g., “state”) to represent the states:

```
type state is (STOPPED, STARTED, BUSY);

signal y: state;
```

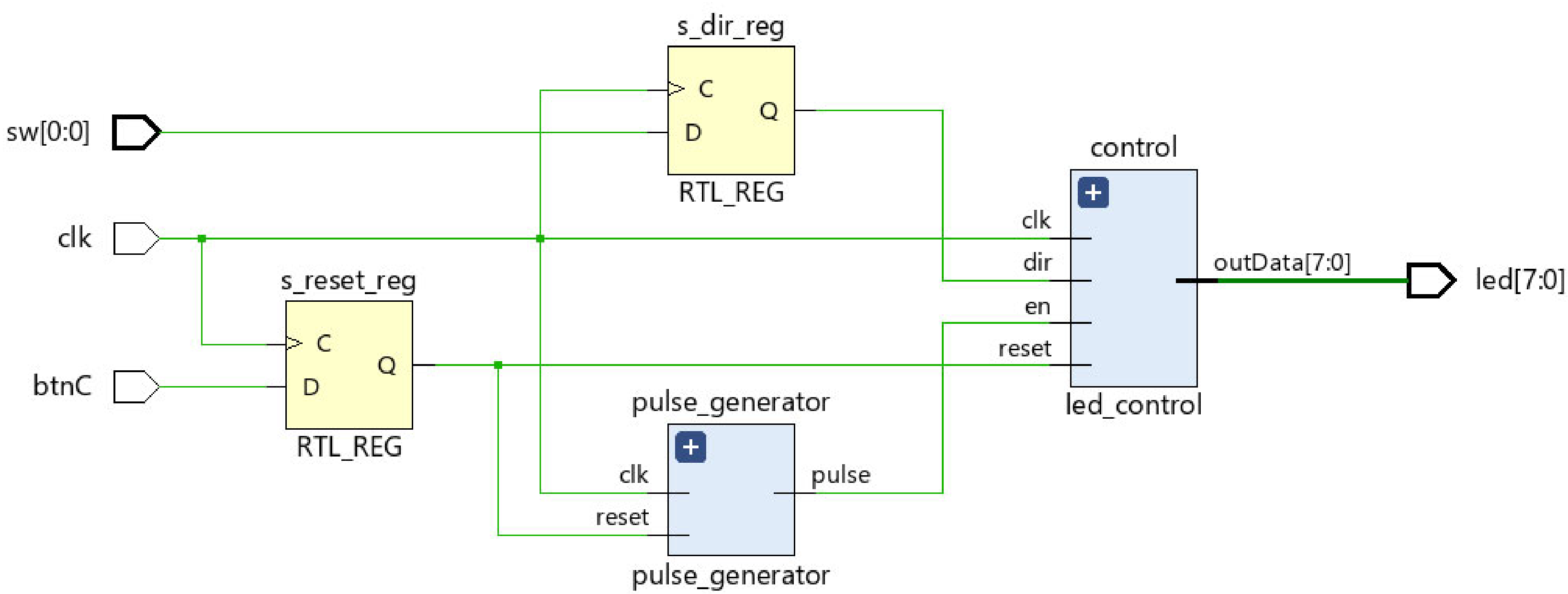
Two processes must be constructed:

- Sync_proc: it is where the state transitions (that occur on the clock edge) are described.
- Comb_proc: this is a combinational circuit where next state and outputs are defined based on the current state (and input signals).

Example: LED Sequence Controller

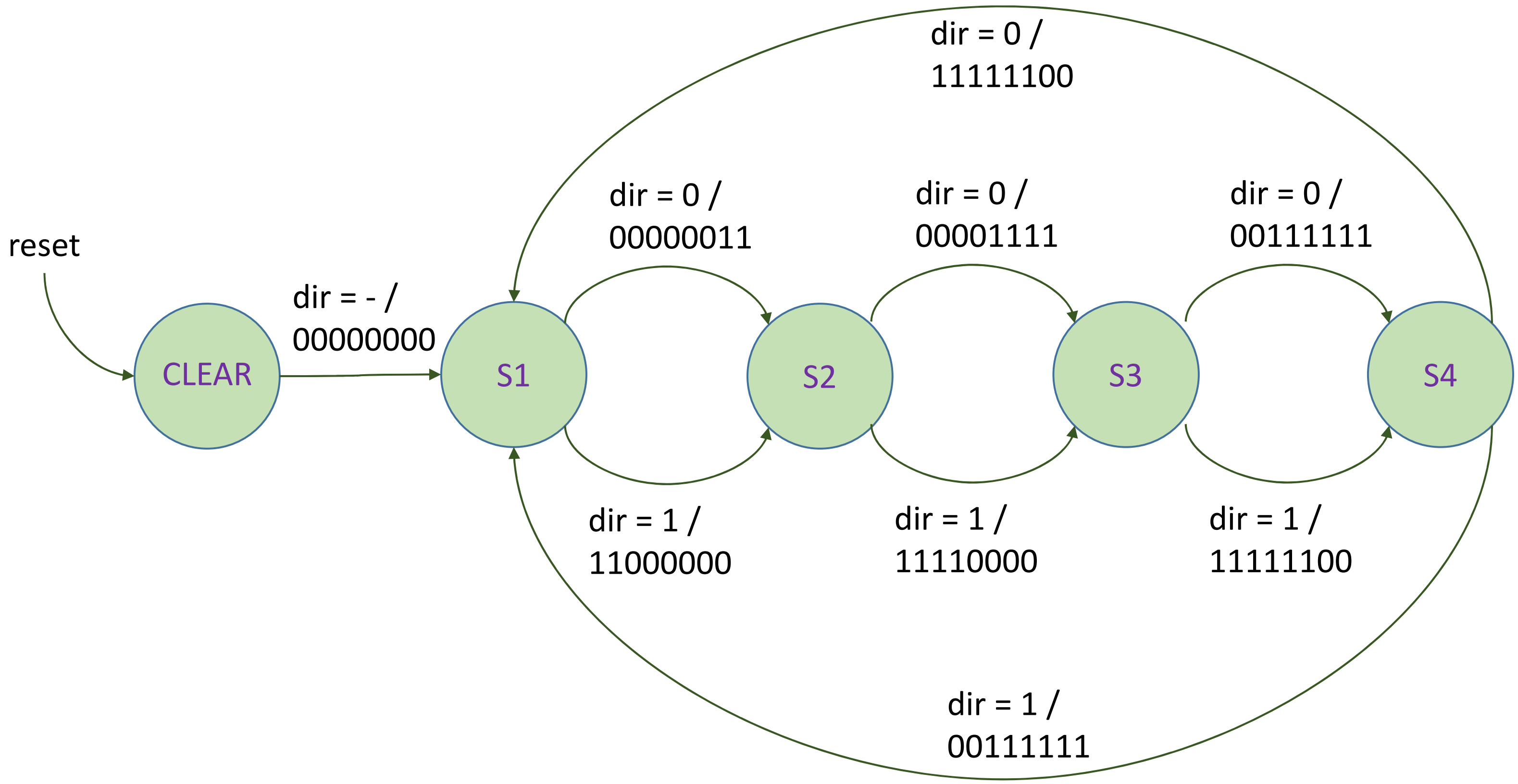
Sequence: 00000011, 00001111, 00111111, 11111100 when sw(0) = ‘0’, or 11000000, 11110000, 11111100, 00111111 when sw(0) = ’1’

The FSM includes an enable that allows for state transitions with frequency of 2Hz.



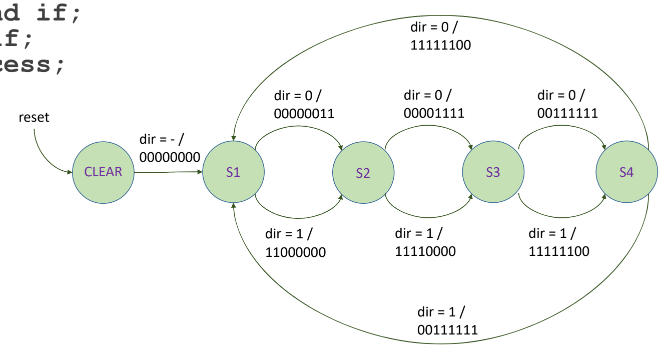
State Diagram

Sequence: 00000011, 00001111, 00111111, 11111100 when sw(0) = ‘0’, or 11000000, 11110000, 11111100, 00111111 when sw(0) = ’1’



Specification in VHDL

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity led_control is
   port(clk          : in  std_logic;
        en           : in std_logic;
        reset        : in  std_logic;
        dir          : in  std_logic;
        outData      : out std_logic_vector(7 downto 0));
end led_control;

architecture Behavioral of led_control is
   type TState is (CLEAR, S1, S2, S3, S4);
   signal pState, nState: TState;

   signal s_data : std_logic_vector(outData'range) :=
                              (others => '0');

begin

sync_proc : process(clk)
   begin
      if (rising_edge(clk)) then
         if (reset = '1') then
            pState <= CLEAR;
         elsif en = '1' then
            pState <= nState;
         end if;
      end if;
   end process;
```




```vhdl
comb_proc : process(pState, dir)
begin
   case pState is
      when CLEAR =>
         s_data <= (others => '0');
         nState <= S1;
      when S1 =>
        if dir = '0' then --left
           s_data <= "00000011";
        else
           s_data <= "11000000";
        end if;
        nState <= S2;
      when S2 =>
        if dir = '0' then
           s_data <= "00001111";
        else
           s_data <= "11110000";
        end if;
        nState <= S3;
      when S3 =>
        if dir = '0' then
           s_data <= "00111111";
        else
           s_data <= "11111100";
        end if;
        nState <= S4;
      when S4 =>
        if dir = '0' then
           s_data <= "11111100";
        else
           s_data <= "00111111";
        end if;
        nState <= S1;
      when others => -- "Catch all" condition
         nState    <= CLEAR;
         s_data <= (others => '0');
      end case;
   end process;

   outData <= s_data;

end Behavioral;
```

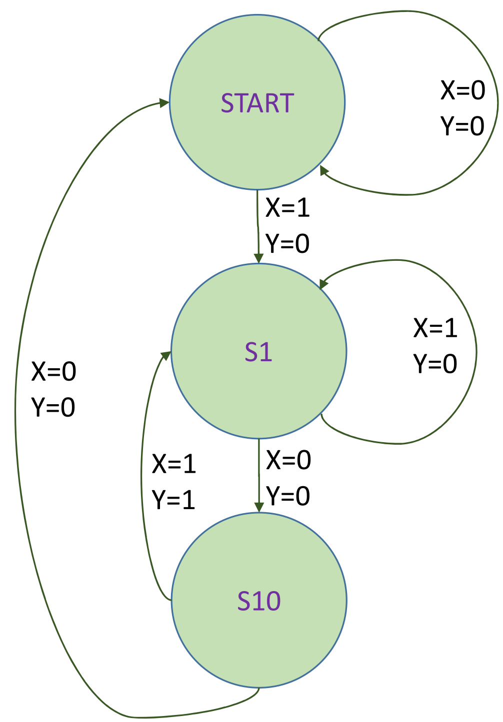
Example: Sequence Detector

Design a sequence detector according to the Mealy model.

The output **Y** must be asserted whenever the sequence 101 is detected at **X** input. Overlapping sequences are allowed.

Example:

- **`X`** `01110101101010001101`
- **`Y`** `00000101001010000001`



Tasks

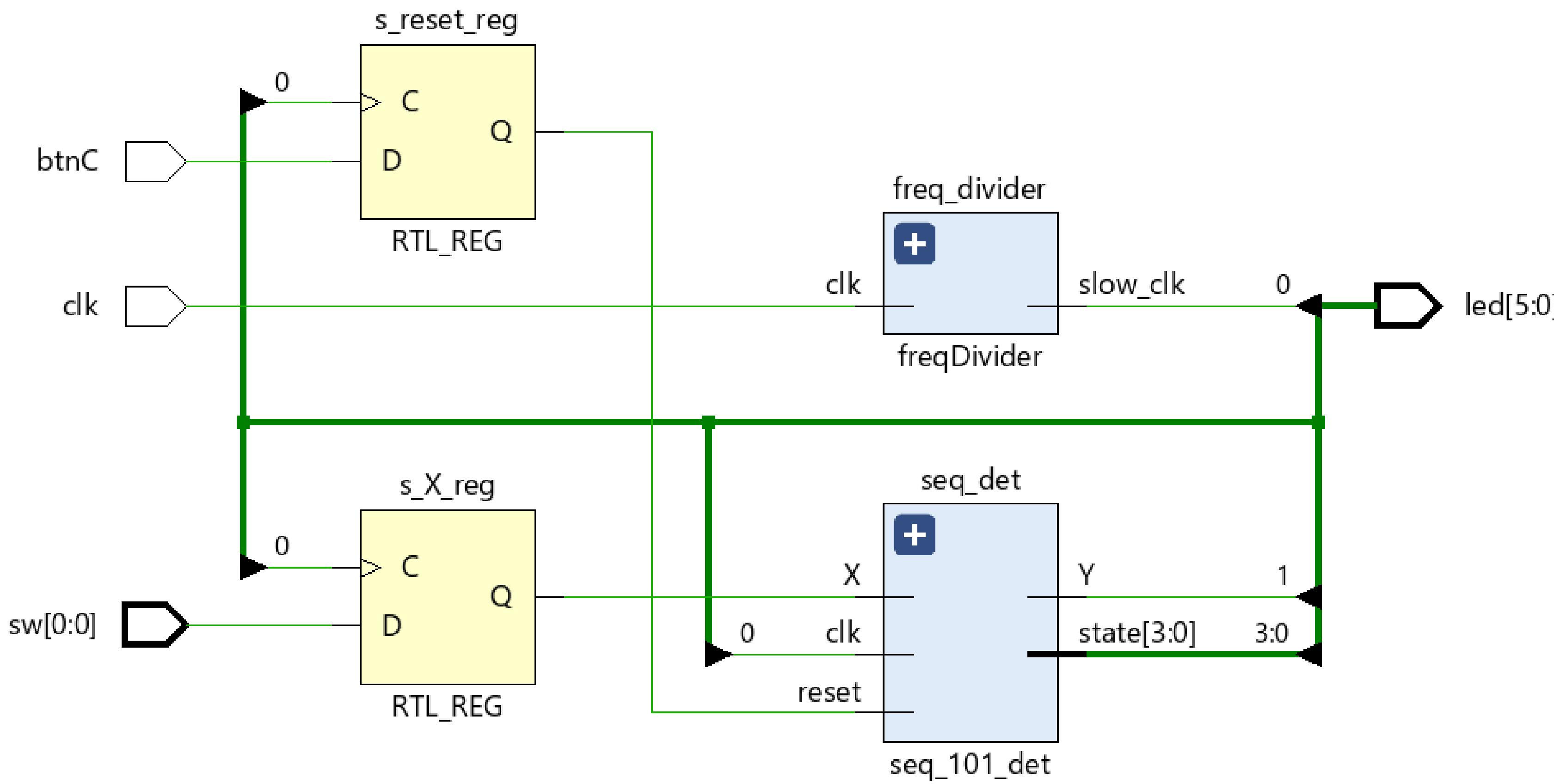
- Design the state diagram
- Model in VHDL
- Simulate with an adequate *testbench*
- Test in the kit (with a very low clock frequency!)

Specification in VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity seq_101_det is
   port(clk      : in  std_logic;
        reset    : in  std_logic;
        X        : in  std_logic;
        Y        : out std_logic;
        state    : out std_logic_vector(3 downto 0));
end seq_101_det;

architecture Behavioral of seq_101_det is
   type TState is (START, S1, S10);
   signal pState, nState: TState;
begin

sync_proc : process(clk)
   begin
      if (rising_edge(clk)) then
         if (reset = '1') then
            pState <= START;
         else
            pState <= nState;
         end if;
      end if;
   end process;
```

```
comb_proc : process(pState, X)
begin
    Y       <= '0';
    nState  <= pState;   -- preserve the state

   case pState is
      when START =>
         if X = '1' then
            nState <= S1;
         end if;
      when S1 =>
        if X = '0' then
            nState <= S10;
        end if;
      when S10 =>
        if X = '0' then
            nState <= START;
        else
            nState <= S1;
            Y <= '1';
        end if;
      when others => -- "Catch all" condition
        nState    <= START;
      end case;
  end process;

  with pState select state <=
    "0001" when START,
    "0010" when S1,
    "0100" when S10,
    "1111" when others;

end Behavioral;
```

Wrapper



s_reset_reg
0
C
Q
D
RTL_REG
btnC
clk
freq_divider
clk
slow_clk
0
freqDivider
led[5:0]
s_X_reg
0
C
Q
D
RTL_REG
sw[0:0]
seq_det
X
Y
1
0
clk
state[3:0]
3:0
reset
seq_101_det

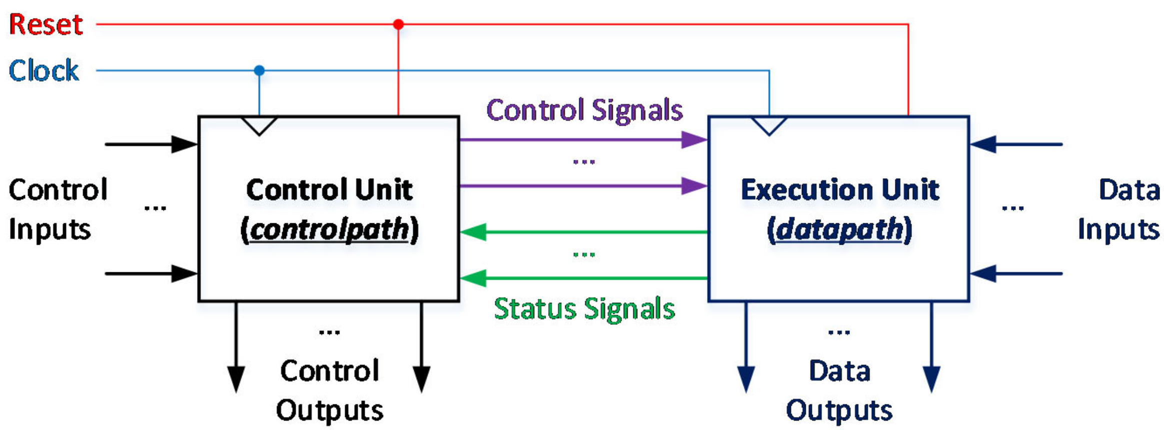
Example: Sequential Multiplier

Unsigned multiplier

- Combinational/parallel implementation (in VHDL):

```
multResult   <= operand0  *  operand1;
```

- Iterative implementation
  - Several clock cycles are required to calculate the result
- Combinational versus sequential implementation
  - Compromise between performance / frequency / resources



Multiplication Algorithm

Operands: **N** bits

Result: **2N** bits

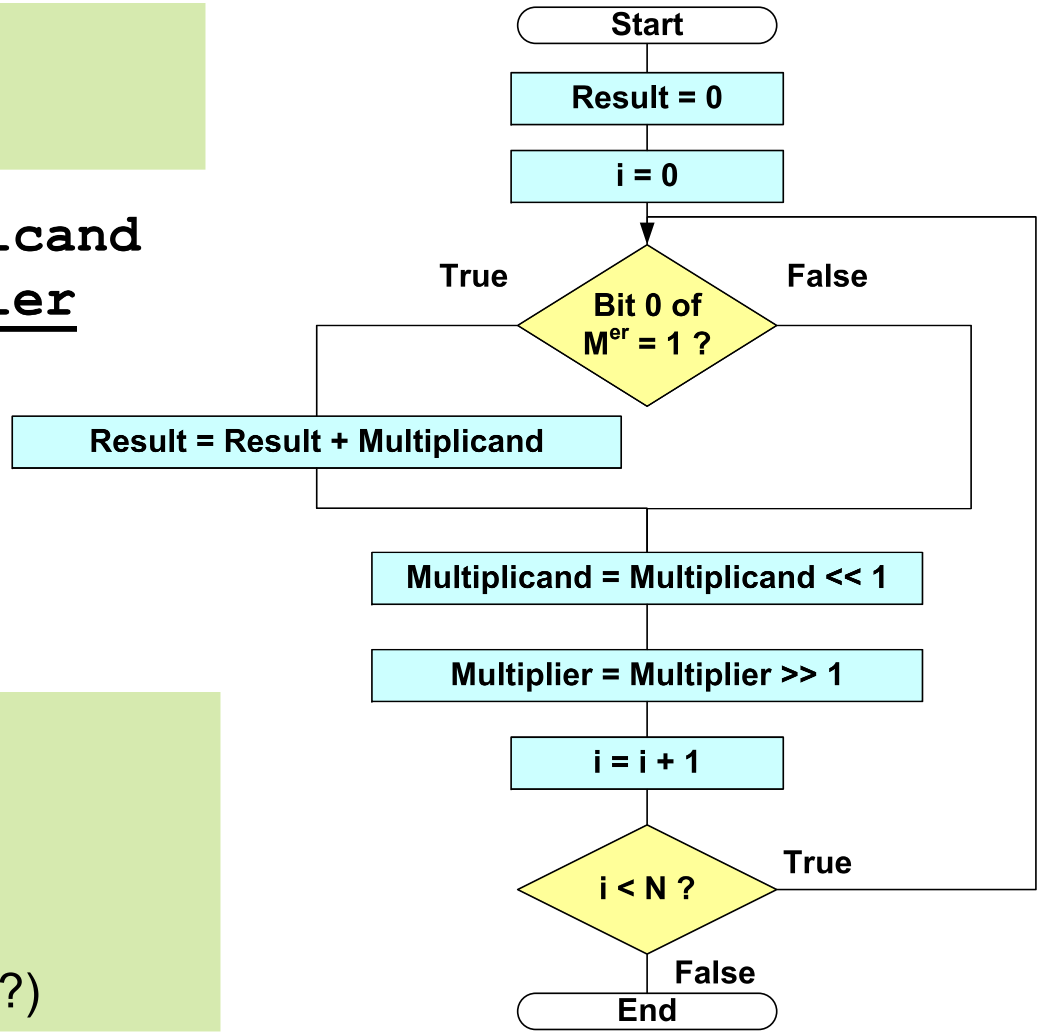
```
   0101 multiplicand
  x0110 multiplier
   0000
  01010
 010100
+0000000
 0011110
```

Required registers:

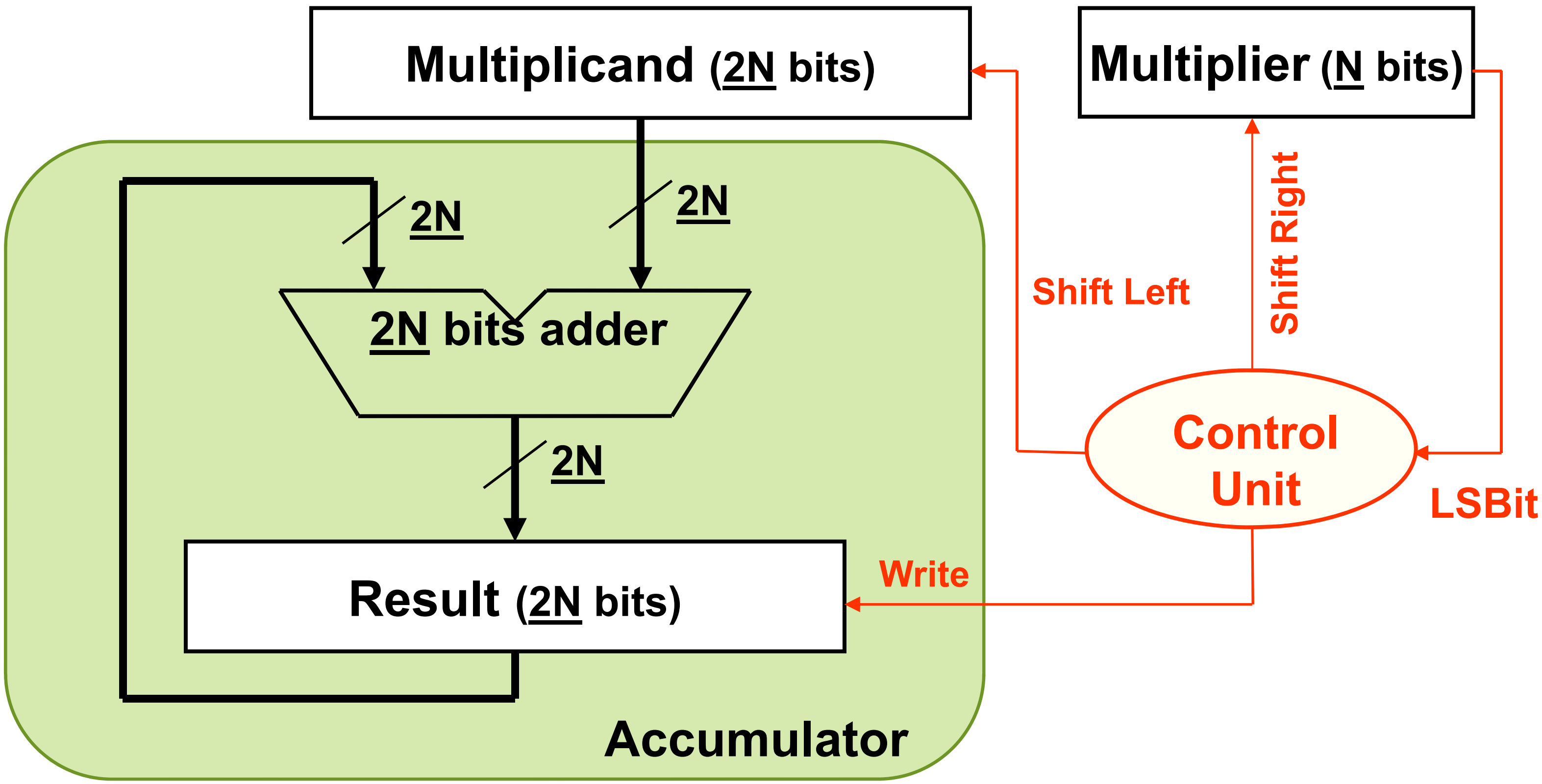
Result: 2N bits

Multiplier : N bits

Multiplicand: 2N bits (why?)

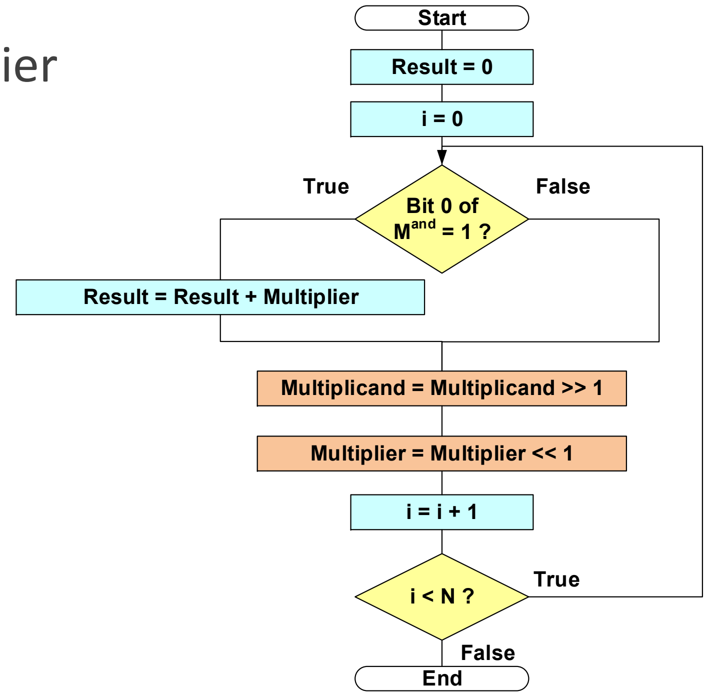


Block Diagram



Datapath Components

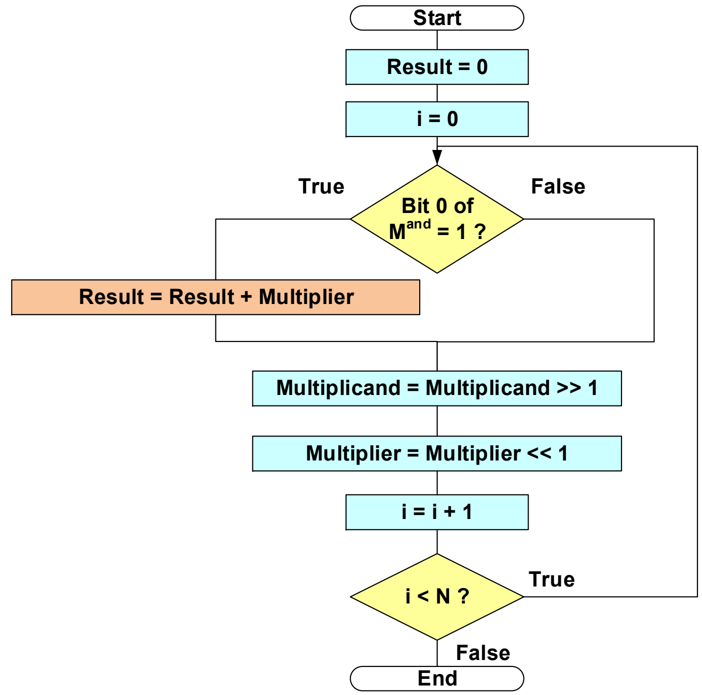
Shift registers for multiplicand and multiplier




```
entity ShiftRegN is
   generic(X           : positive := 8);
   port(reset          : in  std_logic;
        clk            : in  std_logic;
        enable         : in  std_logic;
        load           : in  std_logic;
        shiftLeft      : in  std_logic;
        shiftRight     : in  std_logic;
        shLeftSerIn    : in  std_logic;
        shRightSerIn   : in  std_logic;
        parDataIn      : in  std_logic_vector((X - 1) downto 0);
        parDataOut     : out std_logic_vector((X - 1) downto 0));
end ShiftRegN;
```

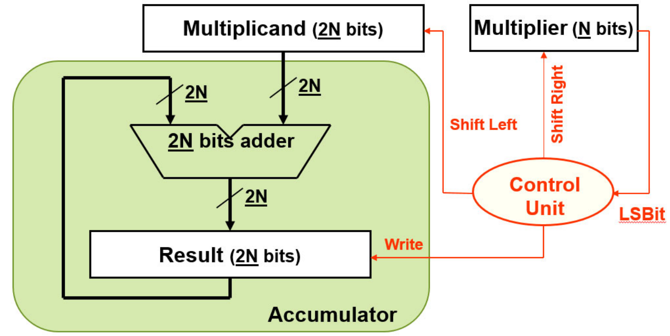
Datapath Components

Accumulator for the result

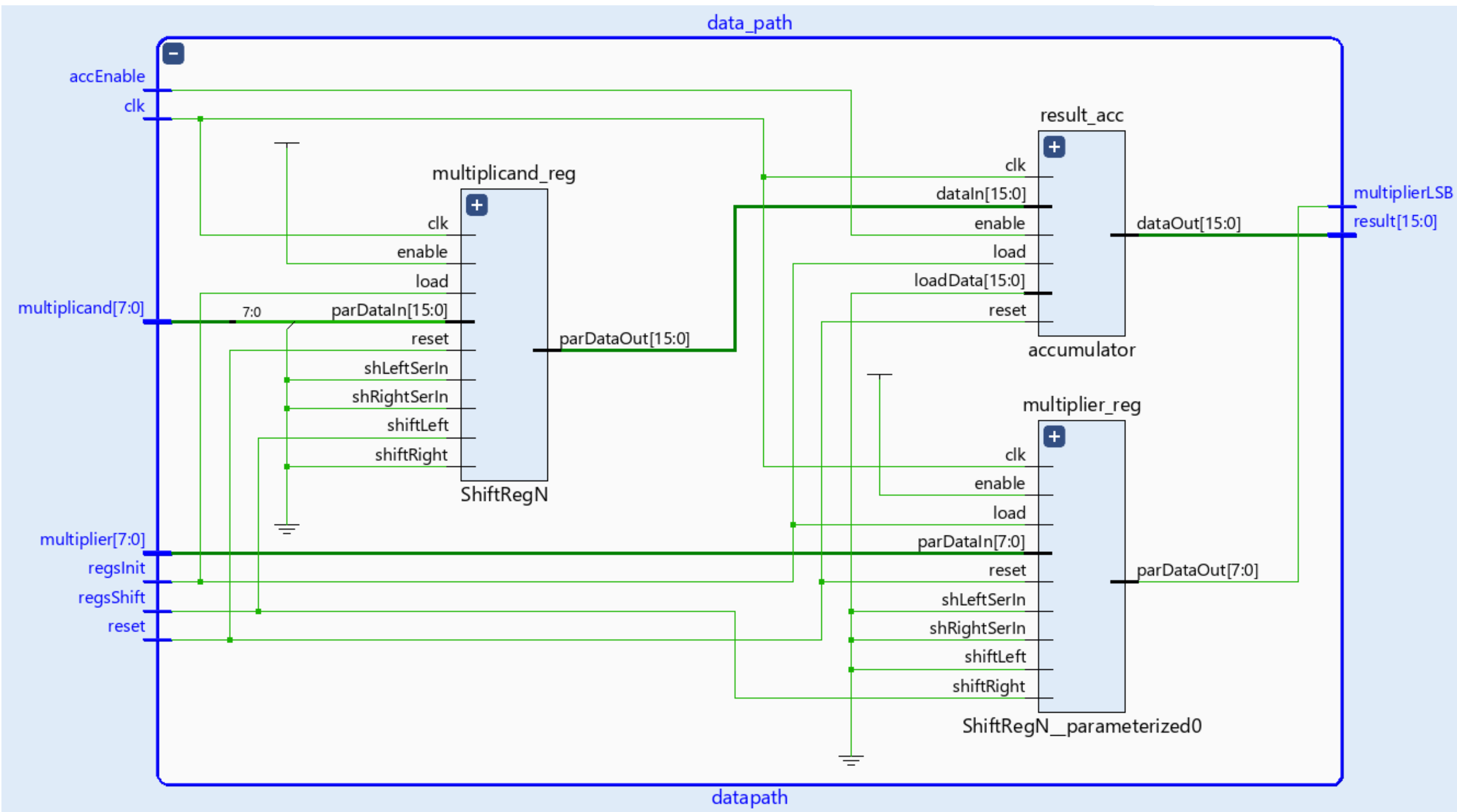



```
entity Accumulator is
   generic(X          : positive := 8);
   port(reset         : in   std_logic;
        clk           : in   std_logic;
        load          : in   std_logic;
        loadData      : in   std_logic_vector((X - 1) downto 0);
        enable        : in   std_logic;
        dataIn        : in   std_logic_vector((X - 1) downto 0);
        dataOut       : out  std_logic_vector((X - 1) downto 0));
end Accumulator;
```

Datapath



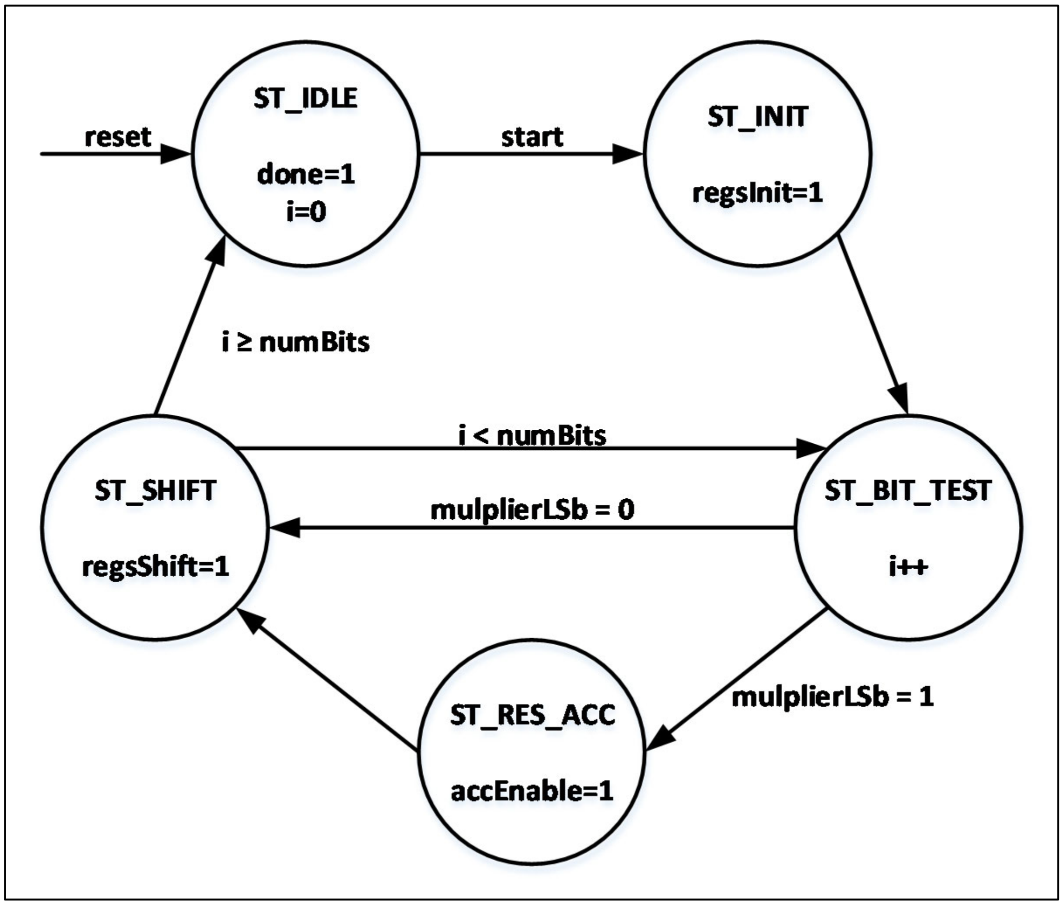
Multiplicand (2N bits)
Multiplier (N bits)
2N
2N
2N bits adder
2N
Result (2N bits)
Accumulator
Shift Left
Shift Right
Control Unit
LSBit
Write

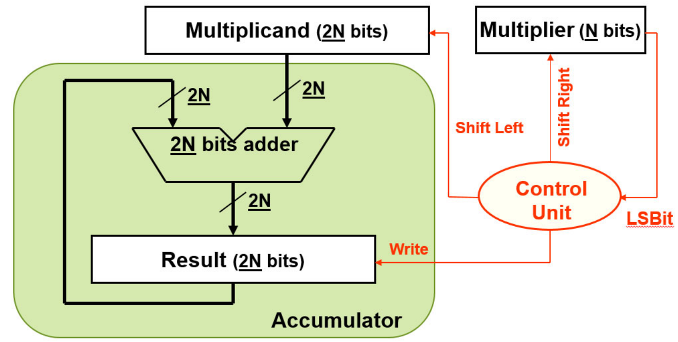


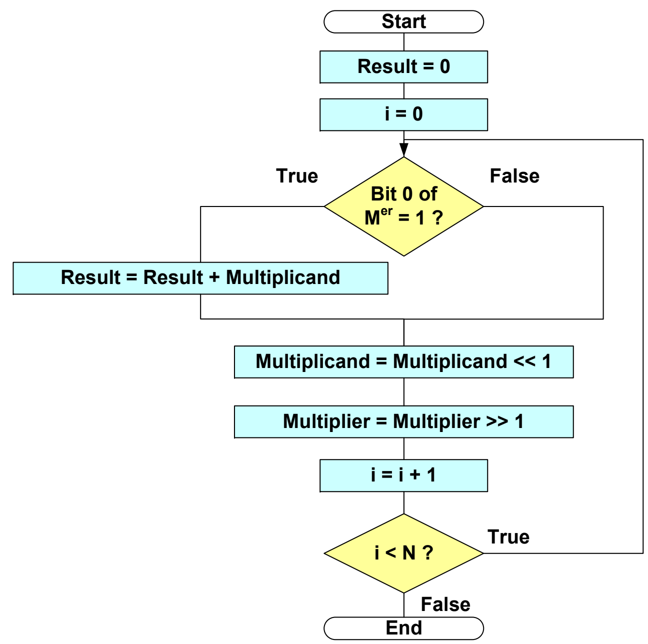
data_path
accEnable
clk
multiplicand[7:0]
multiplier[7:0]
regsInit
regsShift
reset
multiplicand_reg
clk
enable
load
parDataIn[15:0]
reset
shLeftSerIn
shRightSerIn
shiftLeft
shiftRight
parDataOut[15:0]
7:0
ShiftRegN
result_acc
clk
dataIn[15:0]
enable
load
loadData[15:0]
reset
dataOut[15:0]
accumulator
multiplier_reg
clk
enable
load
parDataIn[7:0]
reset
shLeftSerIn
shRightSerIn
shiftLeft
shiftRight
parDataOut[7:0]
ShiftRegN__parameterized0
multiplierLSB
result[15:0]
datapath

Control Unit

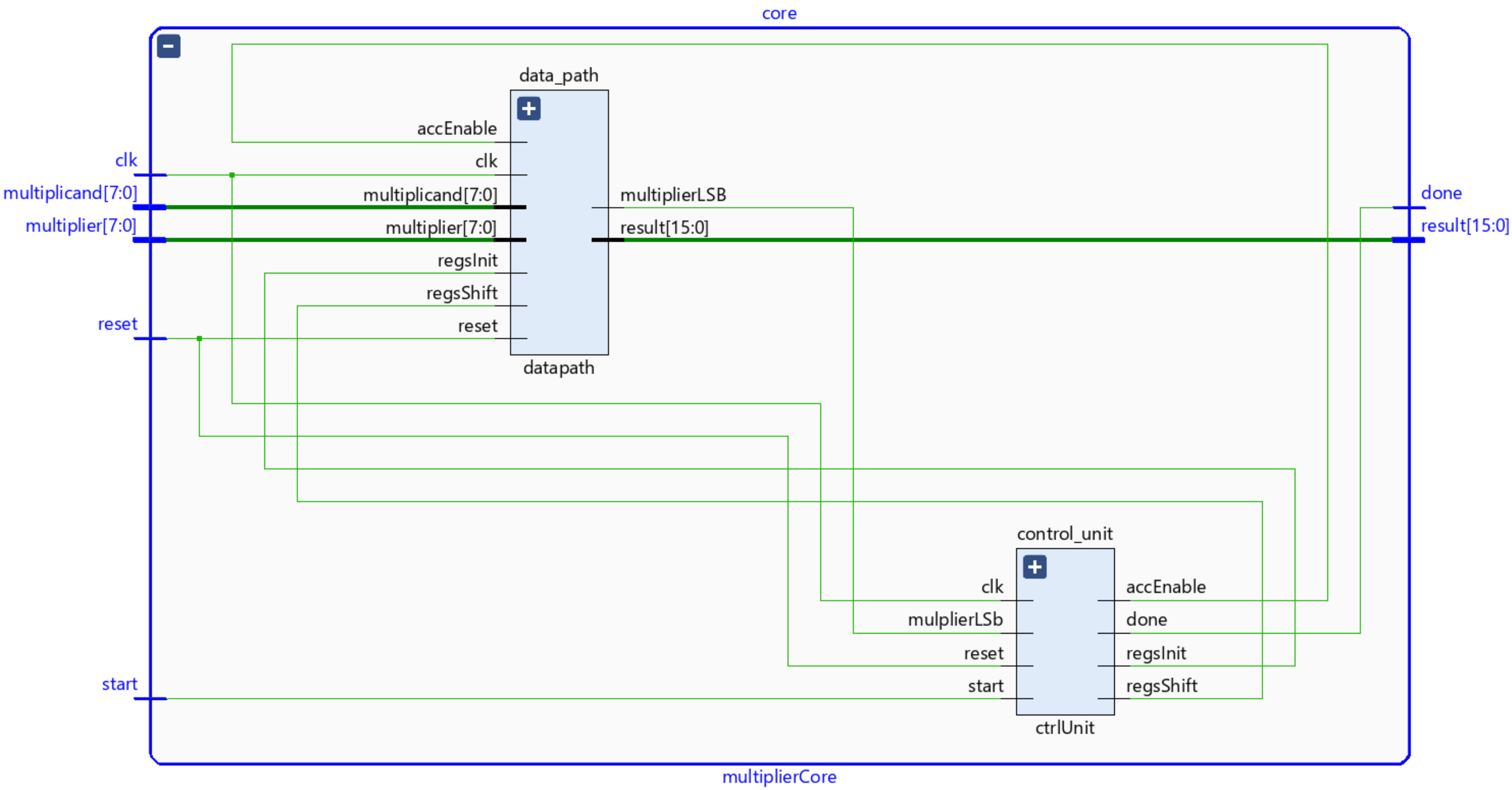
Moore FSM (with datapath – FSMD)



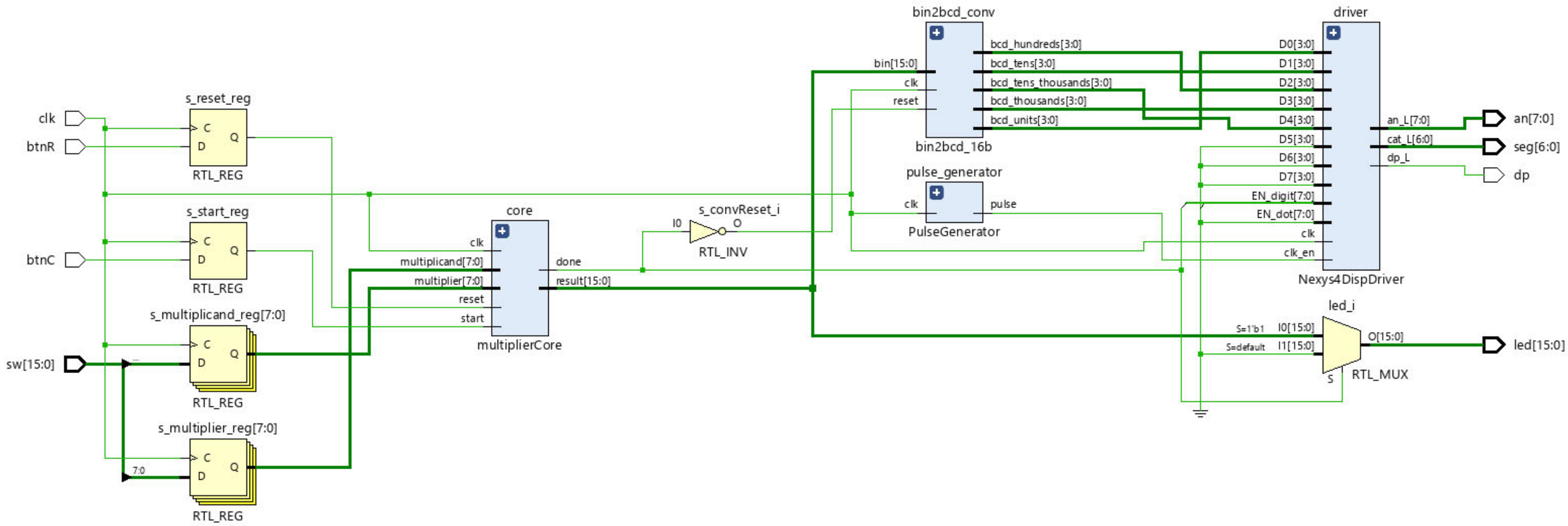




Multiplier Core



Wrapper



clk
btnR
btnC
sw[15:0]
s_reset_reg
s_start_reg
s_multiplicand_reg[7:0]
s_multiplier_reg[7:0]
RTL_REG
core
clk
multiplicand[7:0]
multiplier[7:0]
reset
start
done
result[15:0]
multiplierCore
s_convReset_i
I0
O
RTL_INV
bin2bcd_conv
bin[15:0]
clk
reset
bcd_hundreds[3:0]
bcd_tens[3:0]
bcd_tens_thousands[3:0]
bcd_thousands[3:0]
bcd_units[3:0]
bin2bcd_16b
pulse_generator
clk
pulse
PulseGenerator
driver
D0[3:0]
D1[3:0]
D2[3:0]
D3[3:0]
D4[3:0]
D5[3:0]
D6[3:0]
D7[3:0]
EN_digit[7:0]
EN_dot[7:0]
clk
clk_en
an_L[7:0]
cat_L[6:0]
dp_L
Nexys4DispDriver
led_i
I0[15:0]
I1[15:0]
O[15:0]
RTL_MUX
an[7:0]
seg[6:0]
dp
led[15:0]

Final Remarks

At the end of this lecture you should be able to:

- Use a single clock signal for all the project's components
- Ensure proper system initialization (reset)
- Prefer synchronous over asynchronous reset
- Debounce inputs if required
- Describe FSMs in VHDL
- Describe FSMs with datapath in VHDL
- Decompose complex systems in datapath and controlpath

To do:

- Test the given projects on Nexys-4 kit