DIGITAL SYSTEM DESIGN
VHDL Coding for FPGAs
Unit 7

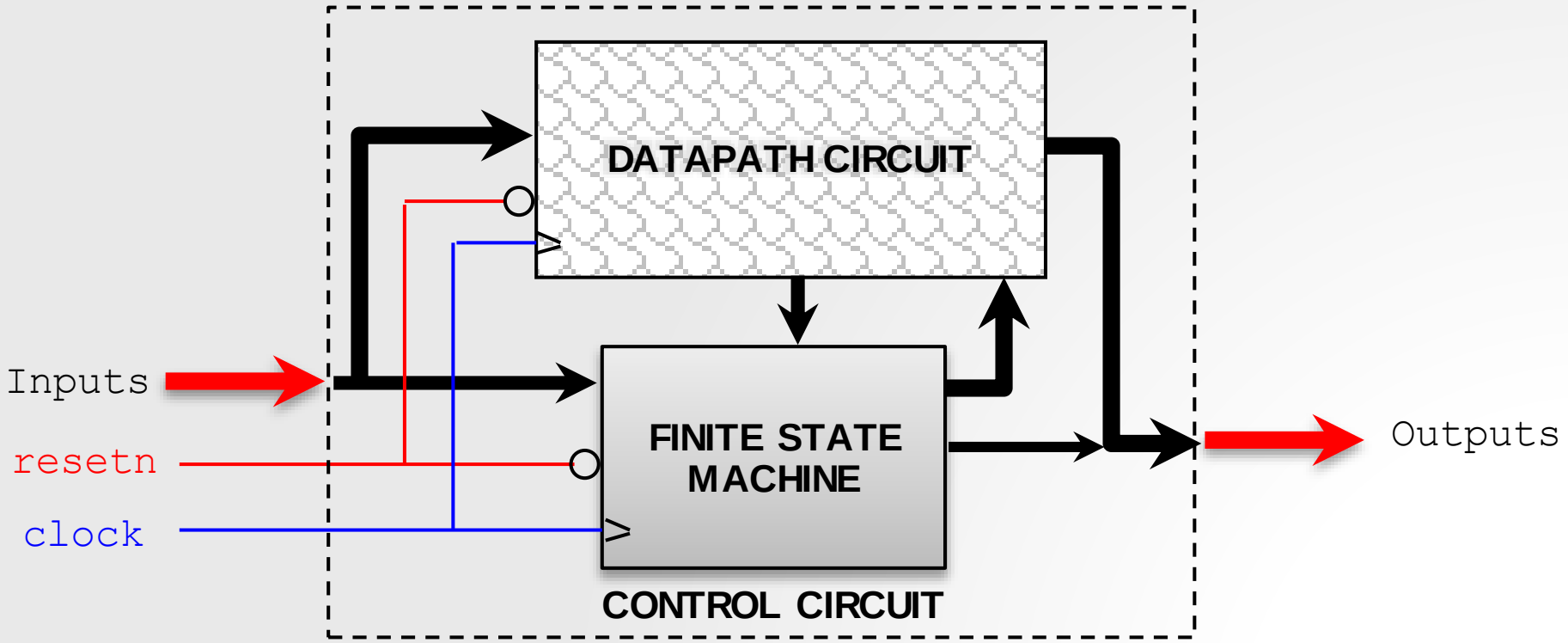
✓INTRODUCTION TO DIGITAL SYSTEM DESIGN:

- Digital System Components
- Use of generic map to map parameters.
- Example: Digital Stopwatch
- Example: Lights Pattern
- Embedding counters and registers in ASM diagrams.

Daniel Llamocca

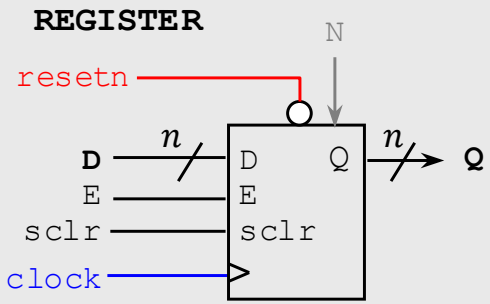
✓ DIGITAL SYSTEM DESIGN

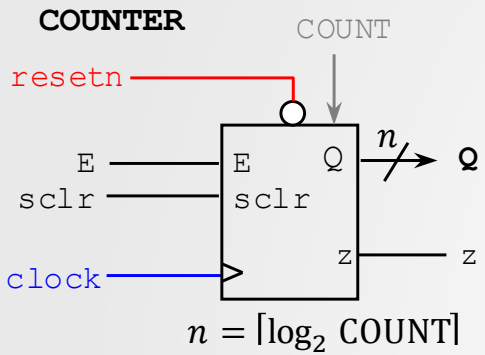
- **Digital System Components:**
  - Finite State Machine, Datapath circuit
- **Design Steps**: Circuit Design, VHDL coding, Synthesis, Simulation, Place and Route (also pin assignment), and FPGA Programming and Testing.

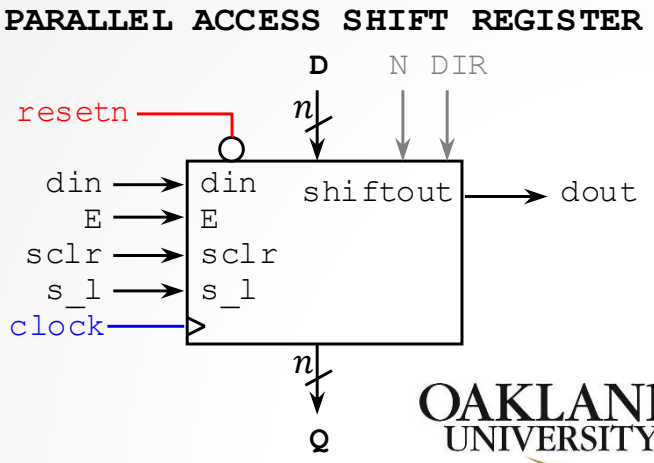


✓ Parameterized components:

- Digital systems include a large number of components. The most common ones are registers, shift registers, and counters. To accelerate development time and to avoid issues with lower-level components, it is strongly recommended to use fully-tested parameterized components (VHDL for FPGAs Tutorial – Unit 5). To indicate parameters, see 'use of generic map' in the following slides.
  - n-bit register with enable and synchronous clear: my_rege
  - Counter modulo-N (COUNT=N) with enable and synchronous clear: my_genpulse_sclr
  - n-bit parallel access (right/left) register with enable and synchronous clear: my_pashiftreg_sclr

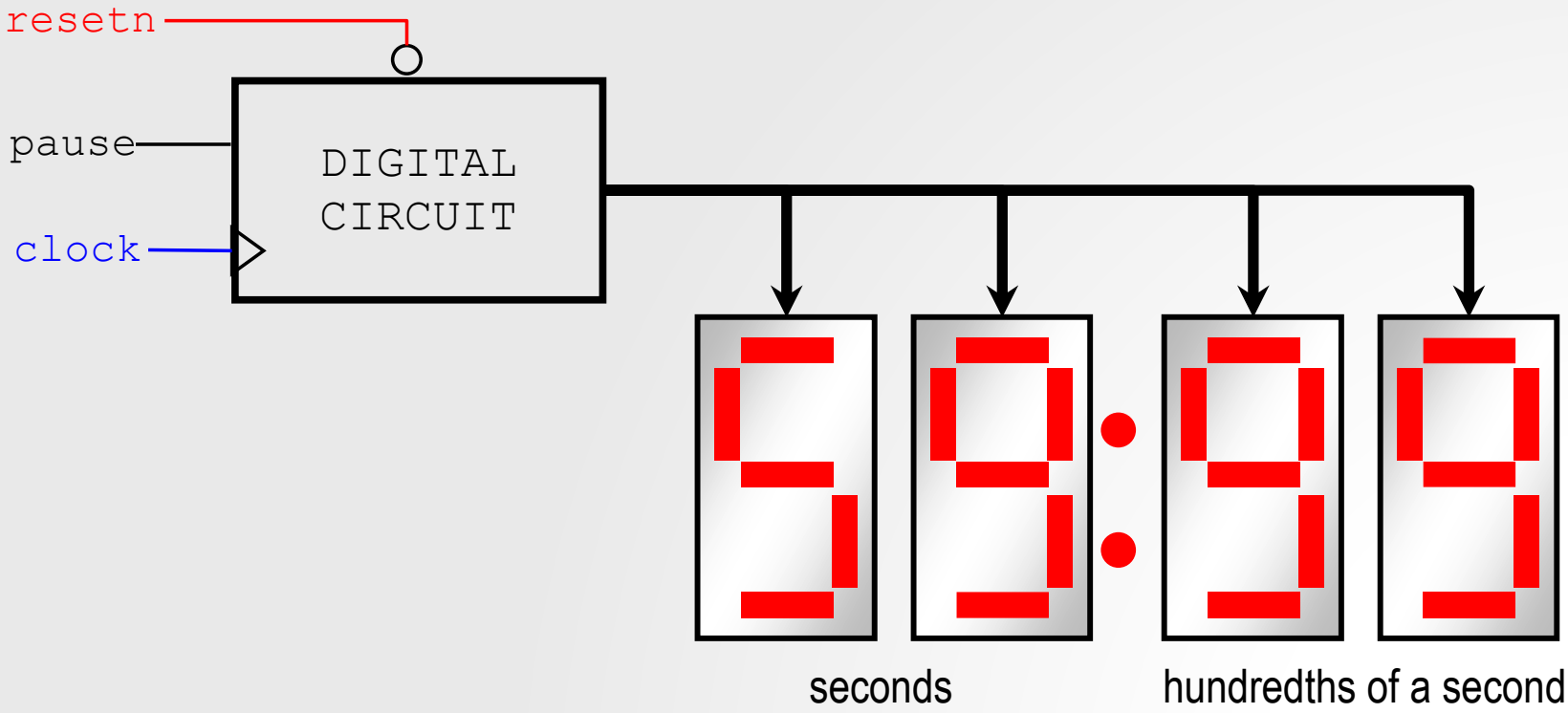






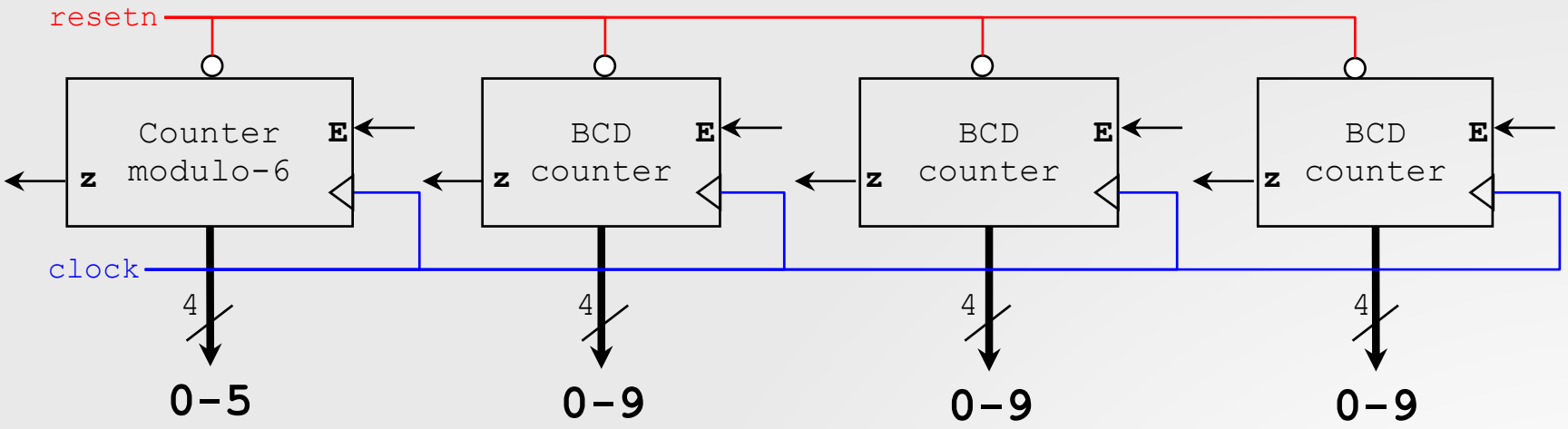
✓EXAMPLE: STOPWATCH

- The Stopwatch that counts in increments of 1/100th of a second. Circuit design and VHDL implementation.
  - **Inputs**: Pause, resetn, clock
  - **Outputs**: Count on four 7-segment displays
  - **Target Board**: DIGILENT NEXYS-4 Board

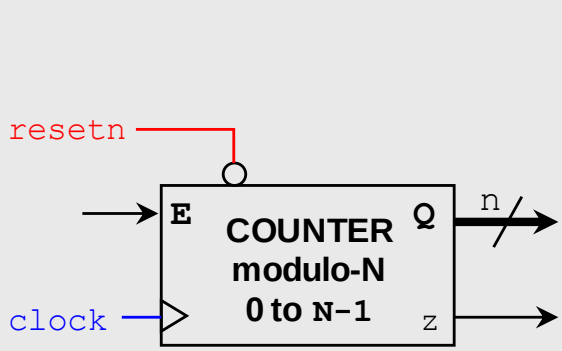


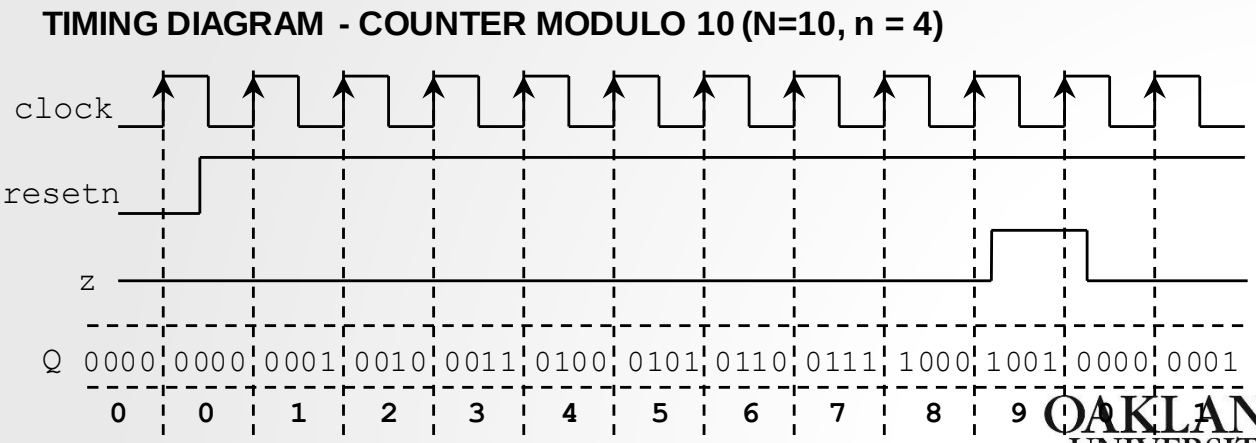
✓EXAMPLE: STOPWATCH

- **Datapath design**: We need four counters. Tree counters modulo-10 and one counter module-6.



- The figure depicts a generic modulo-N counter, where $n = \lceil \log_2 N \rceil$





✓ DIGITAL COUNTER DESIGN

- Counters are usually designed as State Machines. However, for different counts, we need a different state machine. Moreover, if the count is large, the FSM gets intractable.
- More efficient manner: Think of them as accumulators. This way, the VHDL code is easier to read and modify (if we require a different count).
- Example: BCD counter

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

entity my_bcdcount is
 port ( clock, resetn, E: in std_logic;
        Q: out std_logic_vector(3 downto 0);
        z: out std_logic);
end my_bcdcount;

architecture bhv of my_bcdcount is
  signal Qt: std_logic_vector(3 downto 0);
begin
  process (resetn,clock,E)
  begin
    if resetn = '0' then Qt <= "0000";
    elsif (clock'event and clock='1') then
       if E = '1' then
           if Qt = "1001"' then
               Qt <= "0000";
           else
               Qt <= Qt + "0001";
           end if;
       end if;
    end if;
  end process;
  z <= '1' when Qt = "1001" else '0';
  Q <= Qt;
end bhv;
```

✓ COUNTER DESIGN: PARAMETRIC CODE

- The previous VHDL code allows for easy parameterization of counters with arbitrary counts.
- Parametric VHDL code: The following VHDL code has a parameter COUNT. This is the '`my_genpulse.vhd`' code (Unit 5).

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use ieee.math_real.log2.all;
use ieee.math_real_ceil.all;

entity my_genpulse is
 generic (COUNT: INTEGER:= 10);
 port ( clock, resetn, E: in std_logic;
   Q: out std_logic_vector(integer(ceil(log2(real(COUNT))))-1 downto 0);
   z: out std_logic);
end my_genpulse;

architecture bhv of my_genpulse is
  constant nbits:= INTEGER:= integer(ceil(log2(real(COUNT))));
  signal Qt: std_logic_vector(nbits - 1 downto 0);
...
```

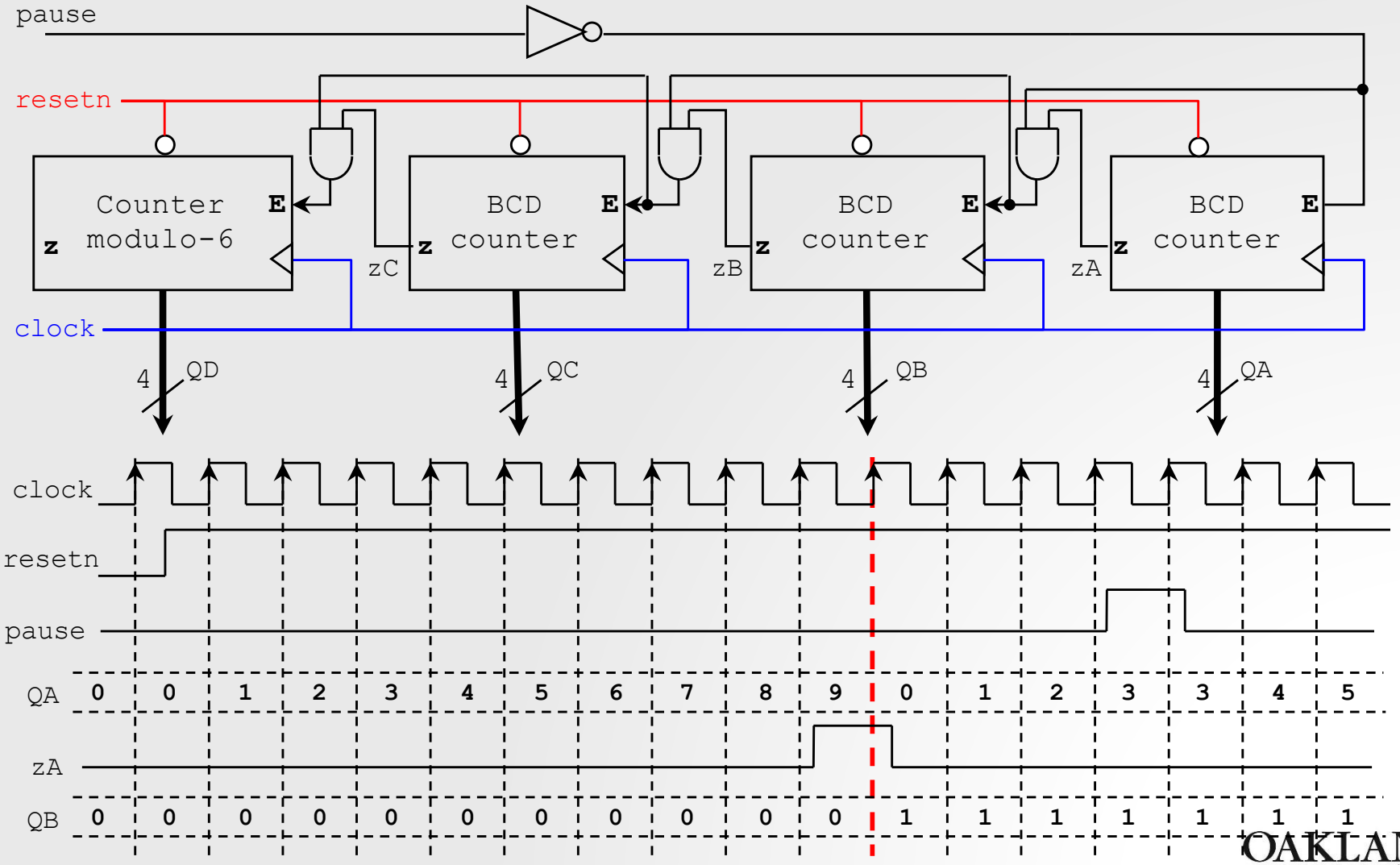
✓ COUNTER DESIGN: PARAMETRIC CODE

- This parametric counter is not only useful to generate pulses, but also to generate counters with any arbitrary counts.

```
...
begin
  process (resetn,clock,E)
  begin
    if resetn = '0' then Qt <= (others => '0');
    elsif (clock'event and clock='1') then
       if E = '1' then
           if Qt = conv_std_logic_vector(COUNT-1,nbits) then
               Qt <= (others => '0');
           else
               Qt <= Qt + conv_std_logic_vector(1,nbits);
           end if;
       end if;
    end if;
  end process;
  z <= '1' when Qt = conv_std_logic_vector (COUNT-1, nbits) else '0';
  Q <= Qt;
end bhv;
```

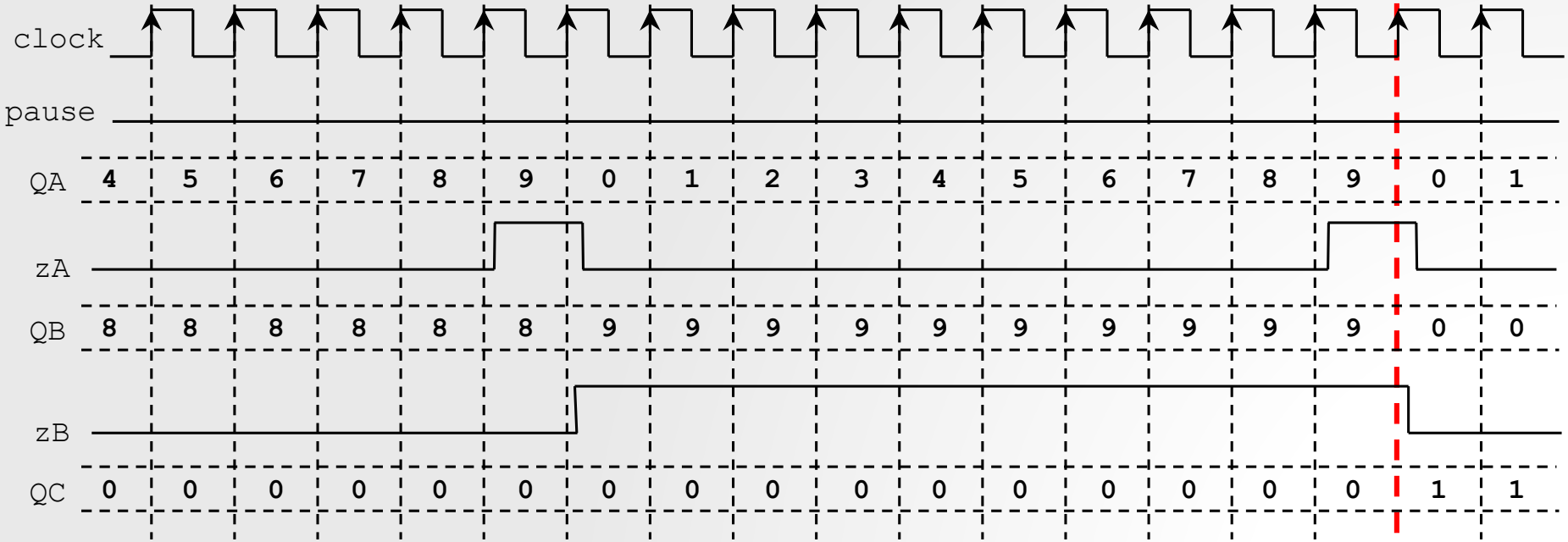
✓EXAMPLE: STOPWATCH

- **Datapath design**: A cascade interconnection allows the counters to behave as desired.



✓ EXAMPLE: STOPWATCH

- '*pause*' input: If the user asserts this input, the count must freeze. This is achieved by using *not(pause)* to enable all the counters.
- Note that it is possible to come up with this circuit by designing an FSM that controls the four enable inputs of the counters.
- Timing diagram: note what needs to happen so that the third counter (QC) increments its count.

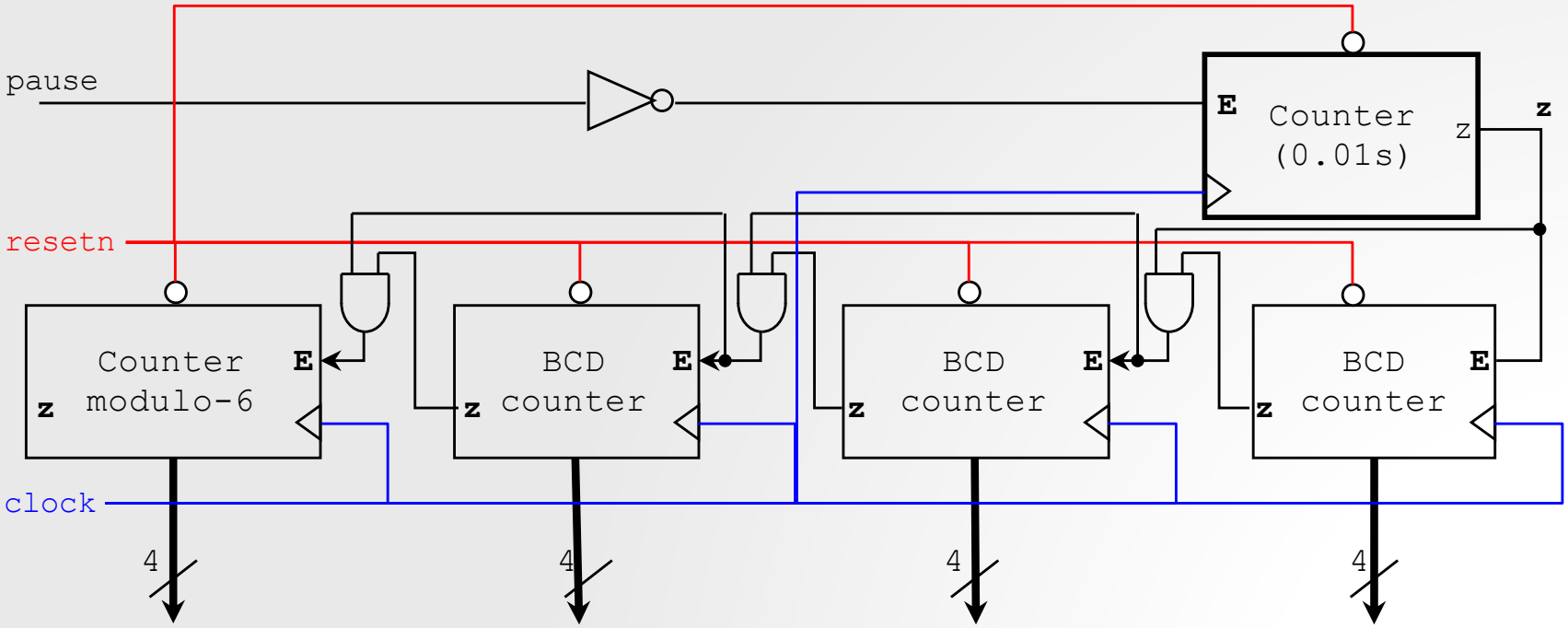


✓STOPWATCH DESIGN

- NEXYS-4 Board: 100 MHz clock. Thus, the counter QA will increment its count every 10 ns.
- We want QA to increment its count every 0.01s (10 ms).
- **Straightforward solution**: Change the input clock to 100 Hz (period 10 ms). This can be a hard problem if precise input clock is required. If we modify the frequency using counters, we might have FPGA clock skew since the 100 Hz clock will not go in the clock tree (among other problems).
- **Efficient solution**: We use the counter that generates a pulse every 10 ms. The output 'z' is then connected to every enable input of the counters. This way, we get the same effect as modifying the clock frequency to 100 Hz.
- The pulse is of duration of the input clock period (10 ns). To generate a pulse every 10 ms, we need a counter that counts up to (10ms/10ns) -1 = 10^6 -1. Note that our generic counter with COUNT=10^6 allows for a quick implementation of this circuit.

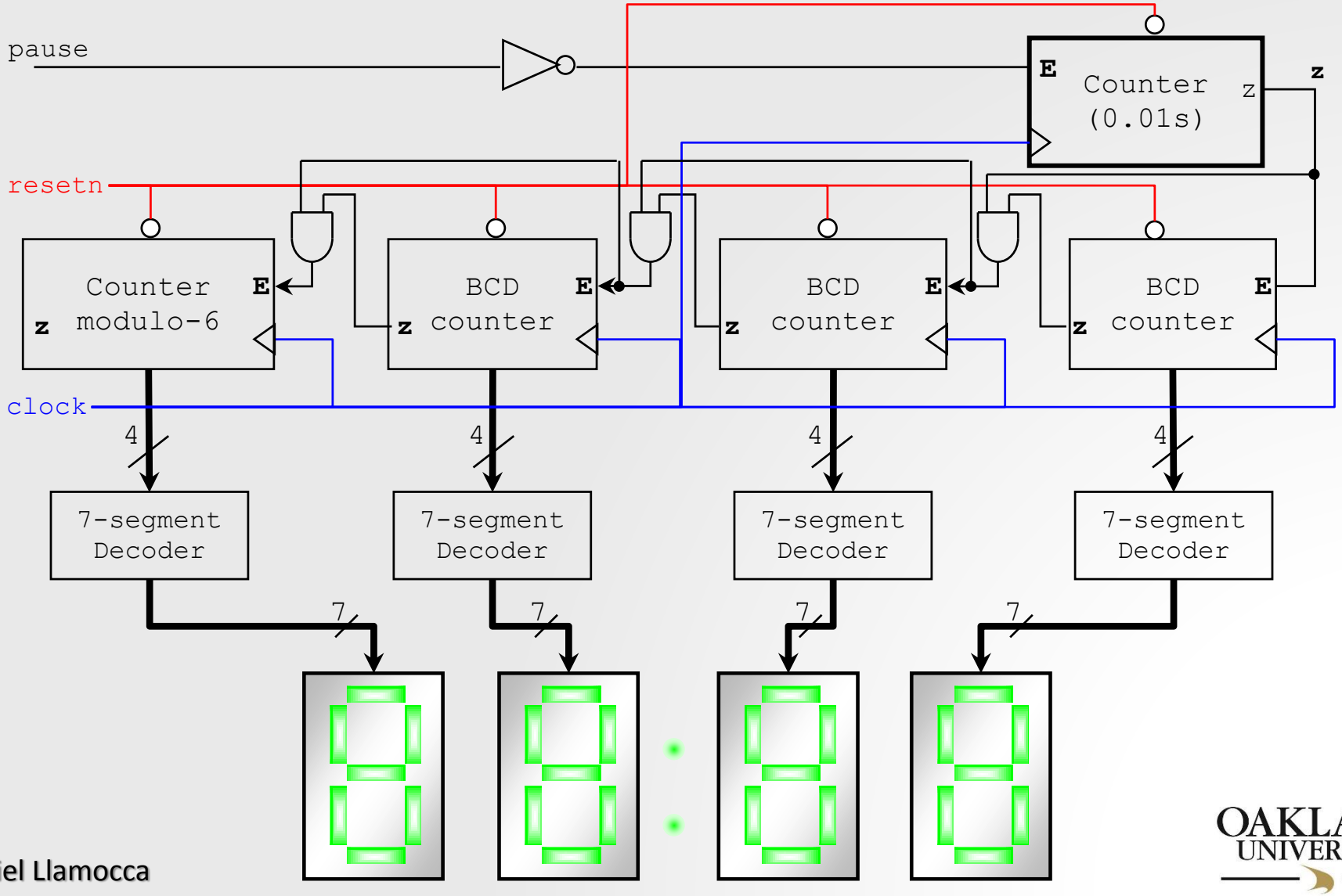
✓STOPWATCH DESIGN

- **Datapath circuit:** The figure shows the stopwatch with the counter up to 10^6 (named 0.01s counter).
- Note how the output 'z' of the 0.01s counter is now connected to all the enables in the counters. This way we make sure that every transition in the counter only occurs every 0.01 s (10 ms)



✓STOPWATCH DESIGN

- **Datapath circuit:** Final system with four 7-segment decoders so that the counter outputs can be seen in the displays.

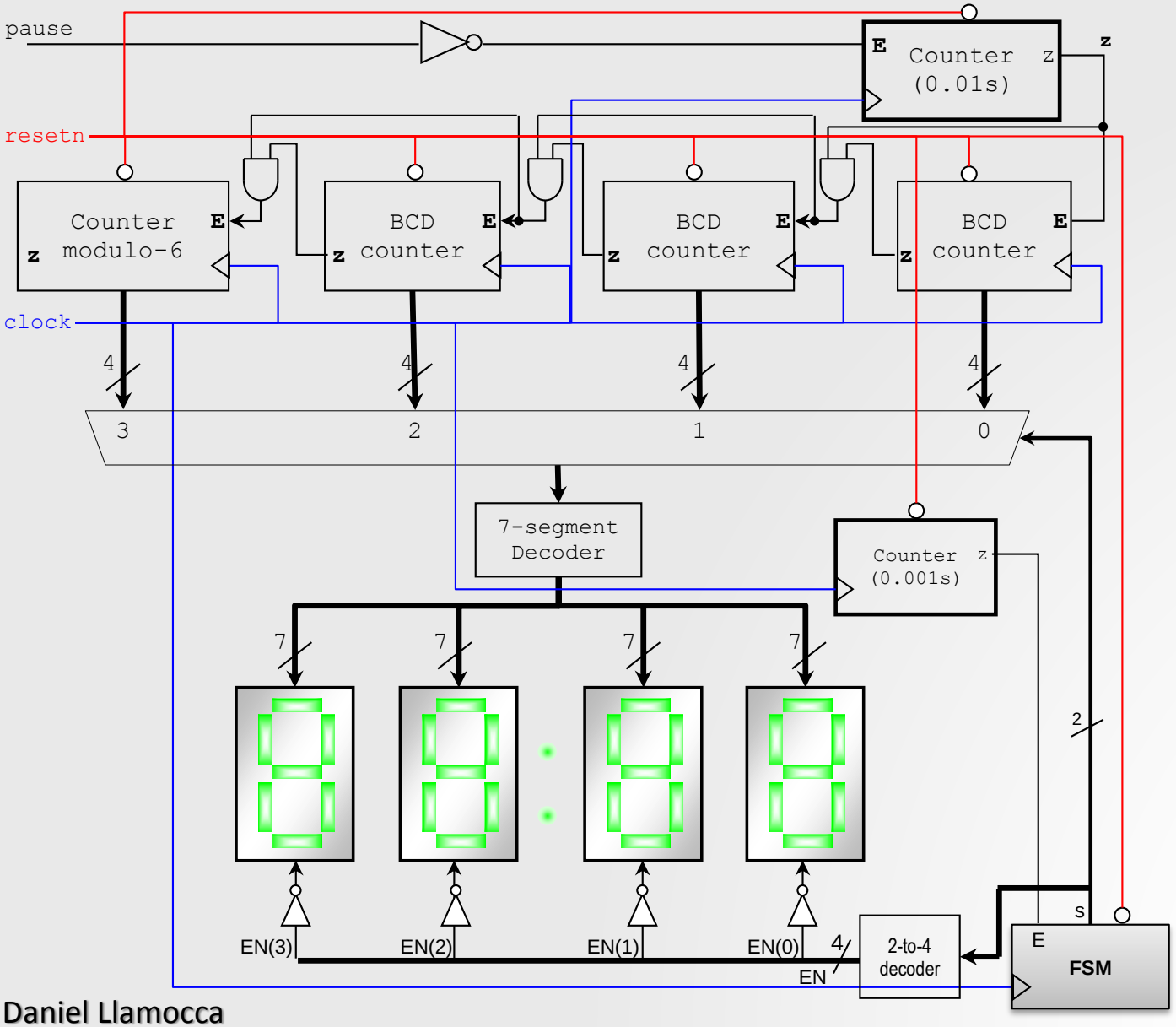


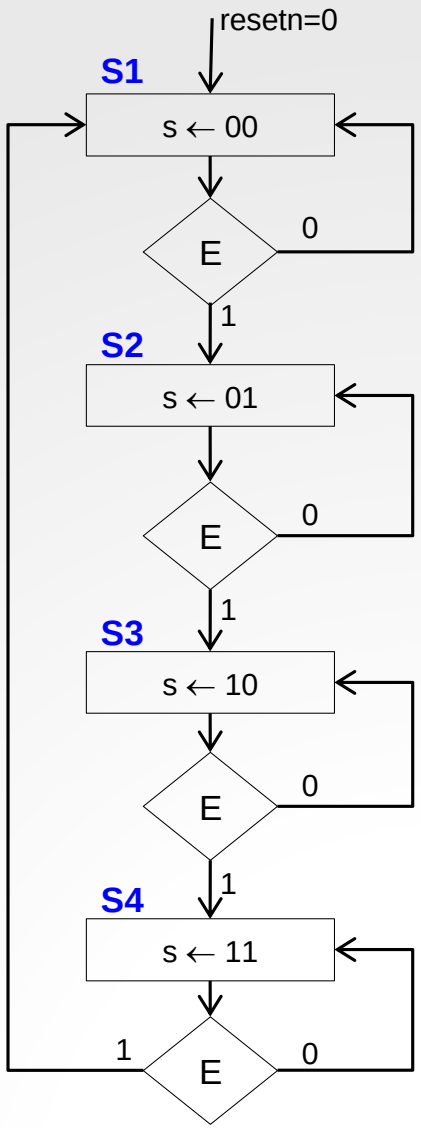
✓STOPWATCH DESIGN

- NEXYS-4 Board: It has eight 7-segment displays, but all the displays share the same inputs. We can also enable (negative logic) a particular 7-segment display via the common anode (see Seven Display section in NEXYS4 datasheet).
- Why do we have then eight 7-seg displays, if apparently all of them will display the same pattern?
- OUTPUT SERIALIZATION: With an enable for each 7-segment display, we can use one 7-segment display at a time. In order for each digit to appear bright and continuously illuminated, we illuminate each digit for only 1ms every 4 ms.
- We need only one 7-segment decoder, a Multiplexor, and an FSM that control the selector of the multiplexor.
- If we want the multiplexor selector to transition only 1 ms every 4 ms, we connect the output 'z' of a new counter (to 0.001s, COUNT = 10^5) to the enable input of the FSM.
- In our design, we only use four 7-segment displays.

✓STOPWATCH DESIGN

- Final Design: Datapath Circuit + FSM

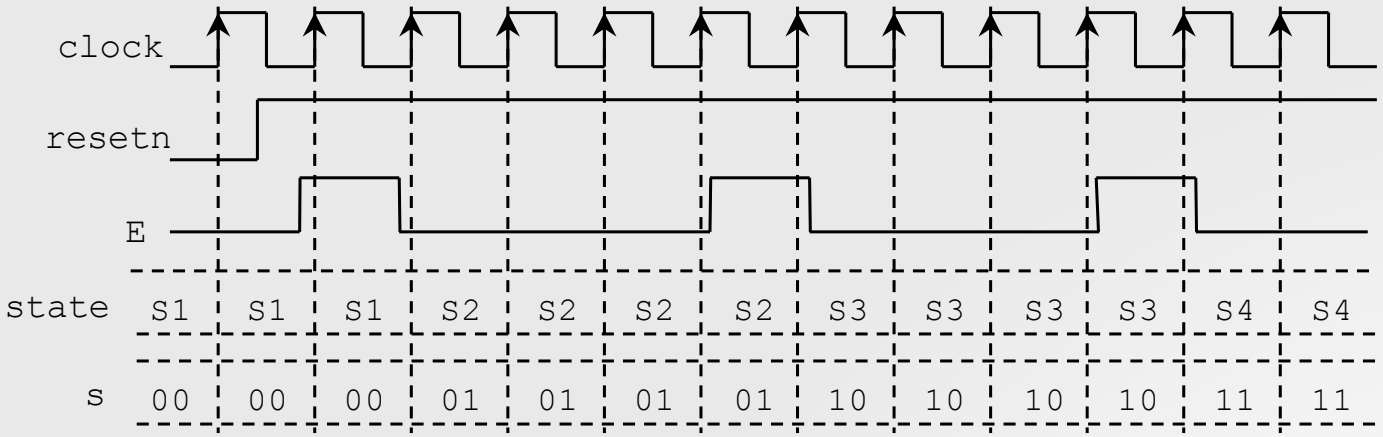




Daniel Llamocca

✓STOPWATCH DESIGN

- FSM Timing Diagram:



- **Xilinx ISE Project:**
  - VHDL code: Instantiation (port map, generic map). For VHDL styling, *see other units in VHDL for FPGAs Tutorial.*
  - Simulation: Testbench. If 0.01s counter is omitted and z = E, we can easily simulate the circuit.
  - Place-and-Route (pin assignment)
  - FPGA programming and testing

➢ **`dig_stopwatch.zip`:**
`dig_stopwatch.vhd`,
`my_genpulse.vhd`,
`sevenseg.vhd`,
`tb_dig_stopwatch.vhd`,
`dig_stopwatch.ucf`

✓USE OF generic map

- Digital system design: many VHDL components available, some as parameterized VHDL code (for re-usability). So, when instantiating these components into a top-level file, we both map the signals (port map) and the parameters (generic map).

- **StopWatch** design: We need to instantiate six counters. Parametric VHDL counter: `my_genpulse.vhd`. We have 3 counters modulo-10, 1 counter modulo-6, 1 counter modulo-10^6, and 1 counter modulo-10^5.
- Here, we must use generic map (see `dig_stopwatch.vhd`. We first declare `my_genpulse.vhd` in the top file:

```
...
architecture struct of dig_stopwatch is
...
 component my_genpulse
  generic (COUNT: INTEGER:= (10**8)/2);
  port (clock, resetn, E: in std_logic;
    Q: out std_logic_vector(integer(ceil(log2(real(COUNT))))-1 downto 0);
    z: out std_logic);
 end component;
...
begin
...
```

We copy what is in the entity of `my_genpulse.vhd`

✓USE OF generic map

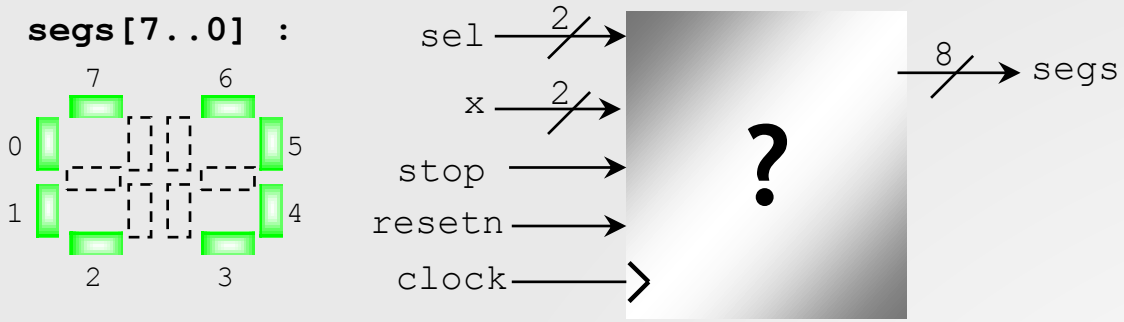
- The port map statement takes care of interconnecting the signals.
- The generic map statement maps the parameters. More than one parameter can be mapped. `my_genpulse` only has one parameter:

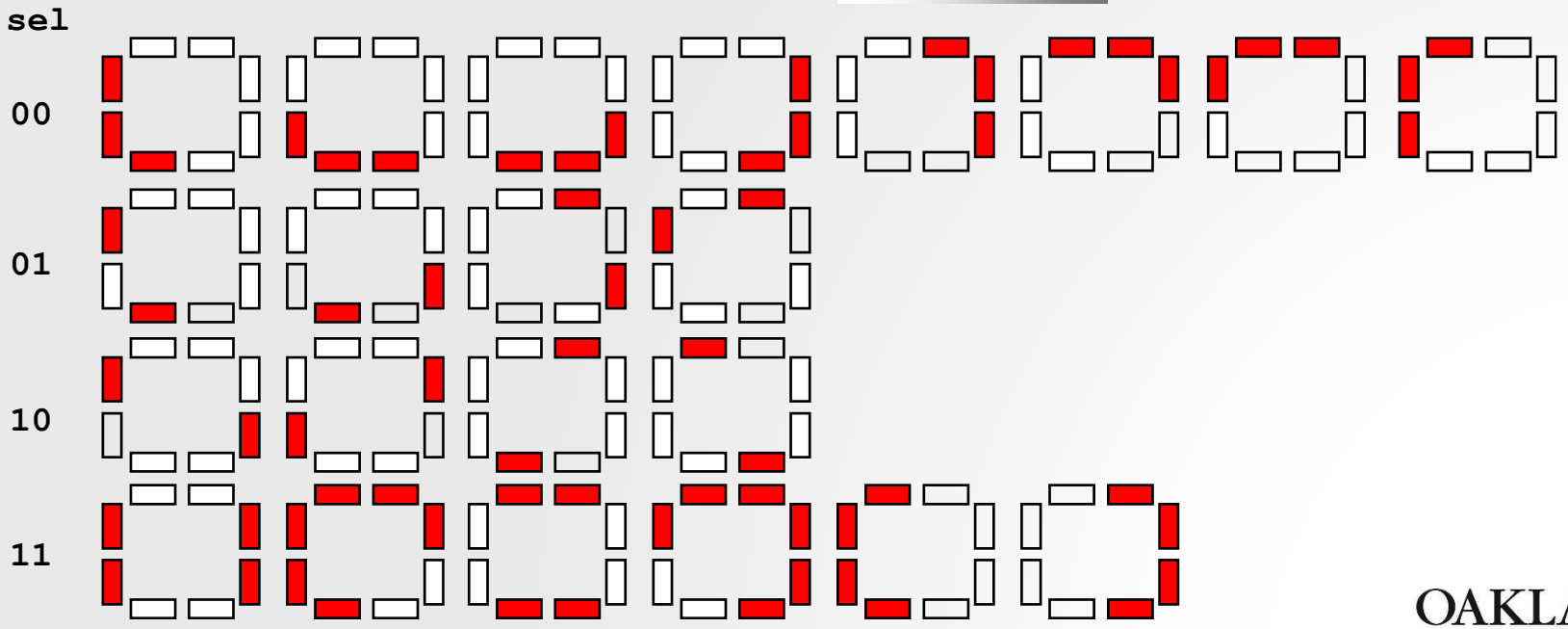
  generic map (*parameter name in* `my_genpulse` => *parameter value in top file*).
- Here, we map 5 counters, each with a different count.
- If parameters are not mapped, the parameter values in the component declaration inside the architecture are assigned. This is a bad coding style: we might need to use a component more than once in the design, each time with different parameters.

```
...
  gz: my_genpulse generic map(COUNT => 10**6) -- modulo-10^6 counter
      port map(clock => clock, resetn =>resetn, E => npause, z => z);
  g0: my_genpulse generic map(COUNT => 10) -- BCD counter
      port map(clock => clock, resetn =>resetn, E => z, Q=>Q0,z=>z_0);
  g1: my_genpulse generic map(COUNT => 10) -- BCD counter
      port map(clock => clock, resetn =>resetn, E =>E_1,Q=>Q1,z=>z_1);
  g2: my_genpulse generic map(COUNT => 10) -- BCD counter
      port map(clock => clock, resetn =>resetn, E =>E_2,Q=>Q2,z=>z_2);
  g3: my_genpulse generic map(COUNT => 6) -- modulo-6 counter
      port map(clock => clock, resetn =>resetn, E =>E_3,Q=>Q3,z=>z_3);
...
end struct;
```

✓EXAMPLE: LIGTHS PATTERN

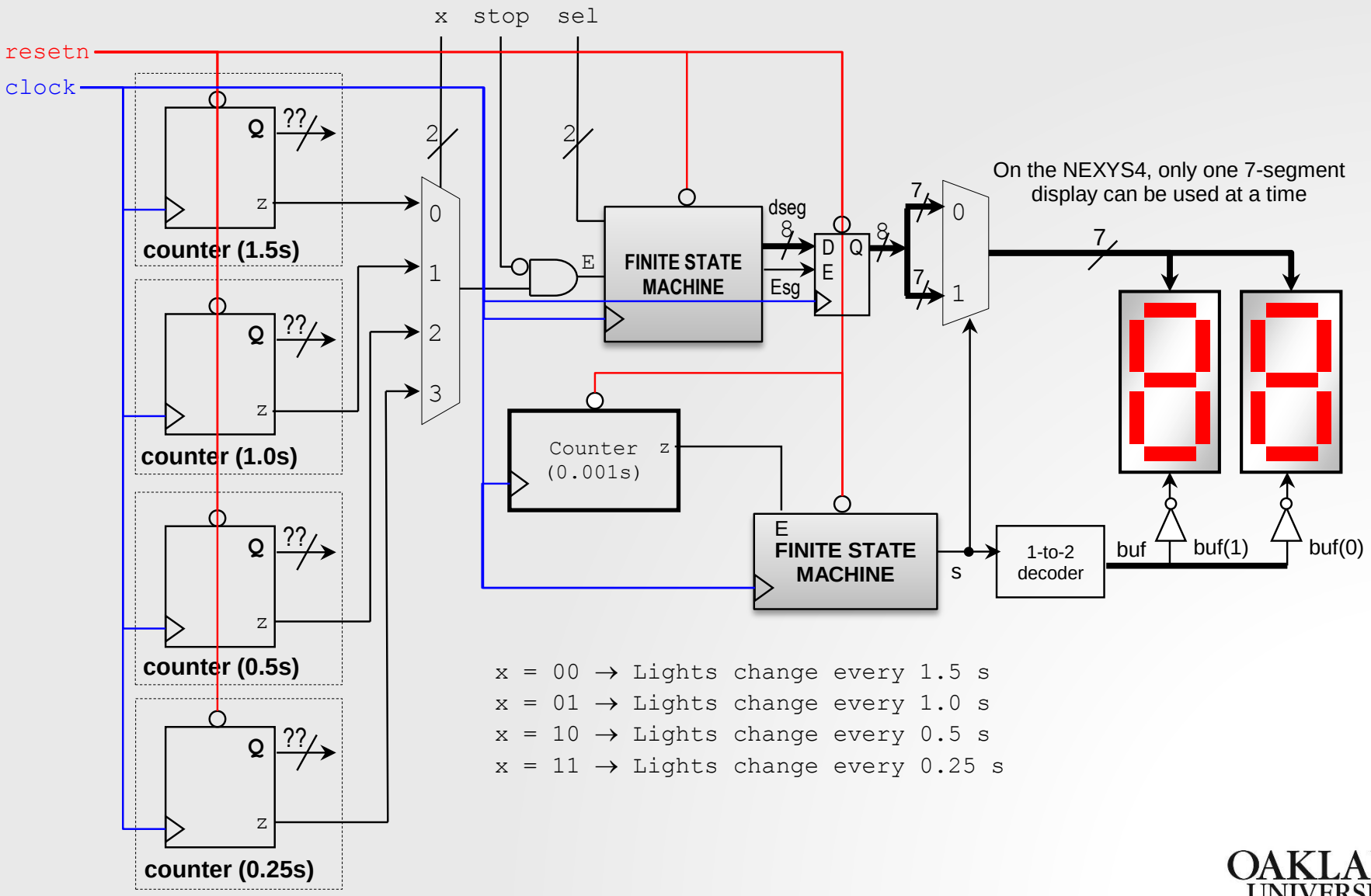
- Configurable lights' pattern generator: sel: selects pattern, stop: freezes the pattern. X: selects the rate at which lights' pattern change (every 1.5, 1.0, 0.5, or 0.25 s)





✓EXAMPLE: LIGTHS PATTERN

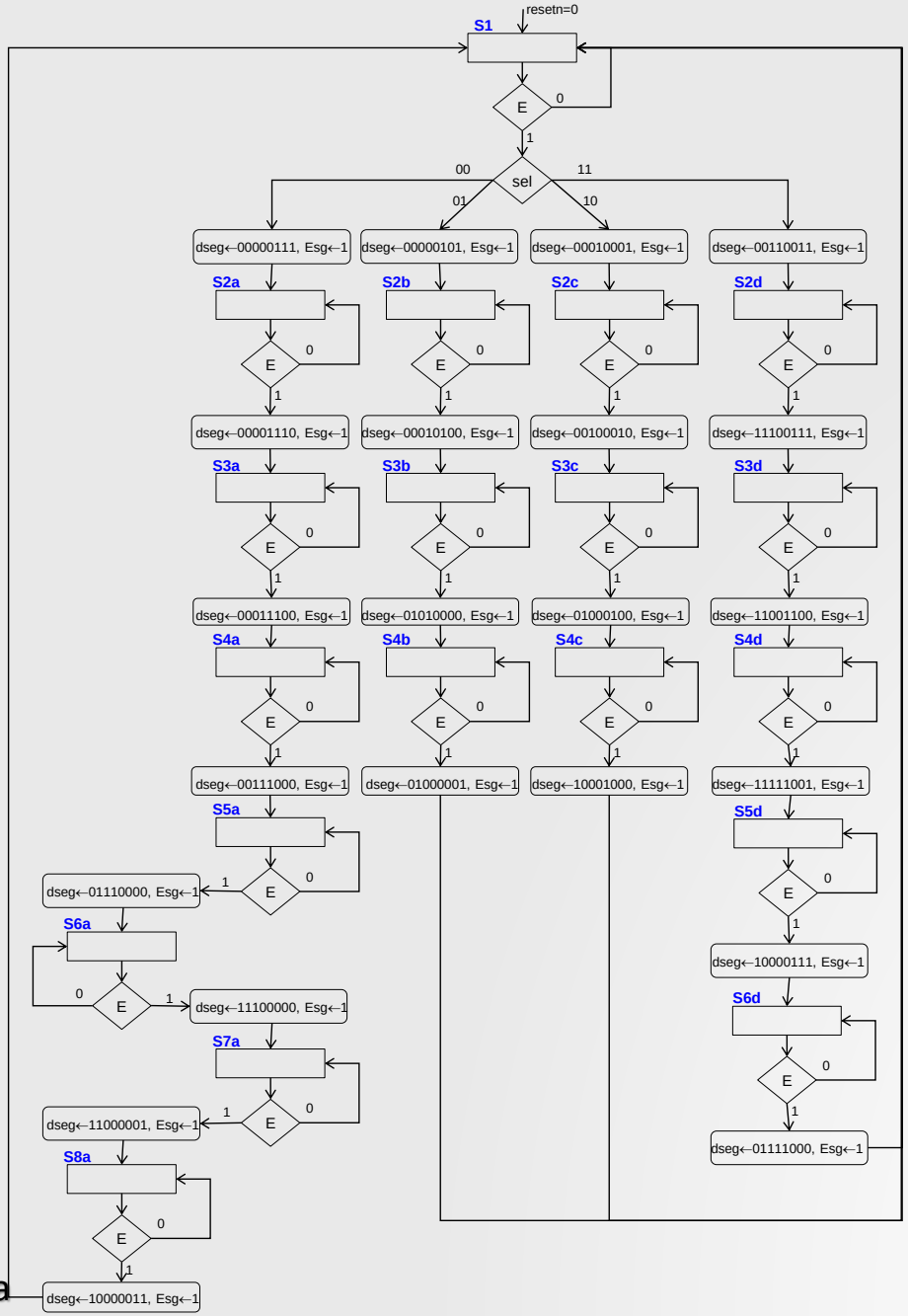
- Entire System:

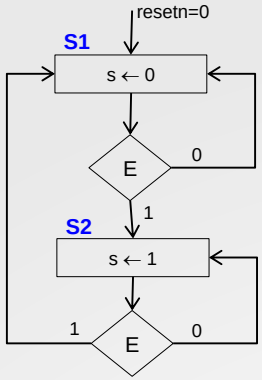



```
x = 00 → Lights change every 1.5 s
x = 01 → Lights change every 1.0 s
x = 10 → Lights change every 0.5 s
x = 11 → Lights change every 0.25 s
```

✓EXAMPLE: LIGTHS PATTERN

- FSMs:

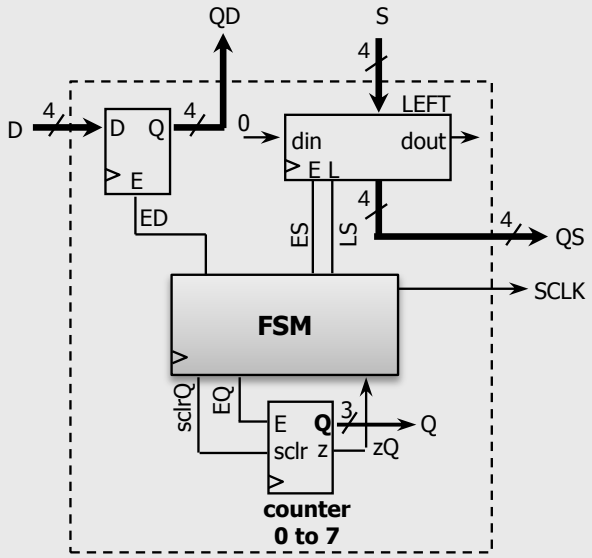


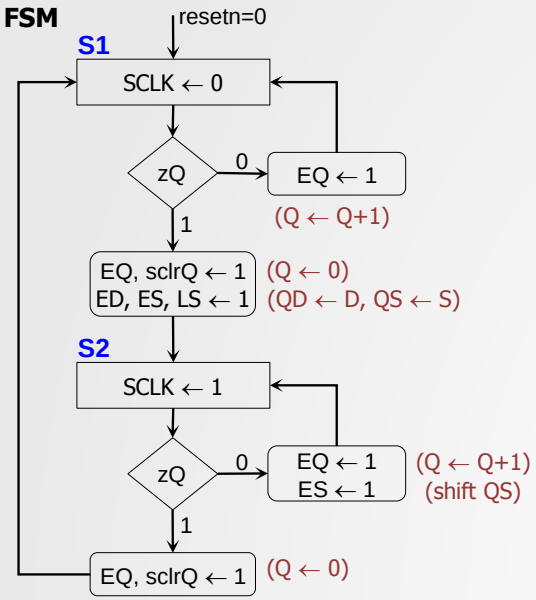


- **lights_pattern.zip**:
  lights_pattern.vhd,
  my_genpulse.vhd,
  my_rege.vhd,
  tb_lights_pattern.vhd,
  lights_pattern.ucf

✓EMBEDDING COUNTERS AND REGISTER IN ASM diagrams

- FSMs usually require counters and registers for proper control. In VHDL code, this requires integrating the fsm description with port map/generic map to instantiate the counters and registers.
- **Example**: `digsys_ex`: An FSM, a counter, a register, and a shift register.
- Asserting EQ or sclrQ can update the counter output Q, which can only be updated (Q←0, Q←Q+1) at the clock edge. For example: after asserting EQ←1, we must wait until the clock edge for Q←Q+1. For the register, asserting ED ← 1 means that QD ← D on the next clock cycle.





Shift Register:

ES ←1: After this signal is asserted, the shift register shifts data on the next clock cycle.

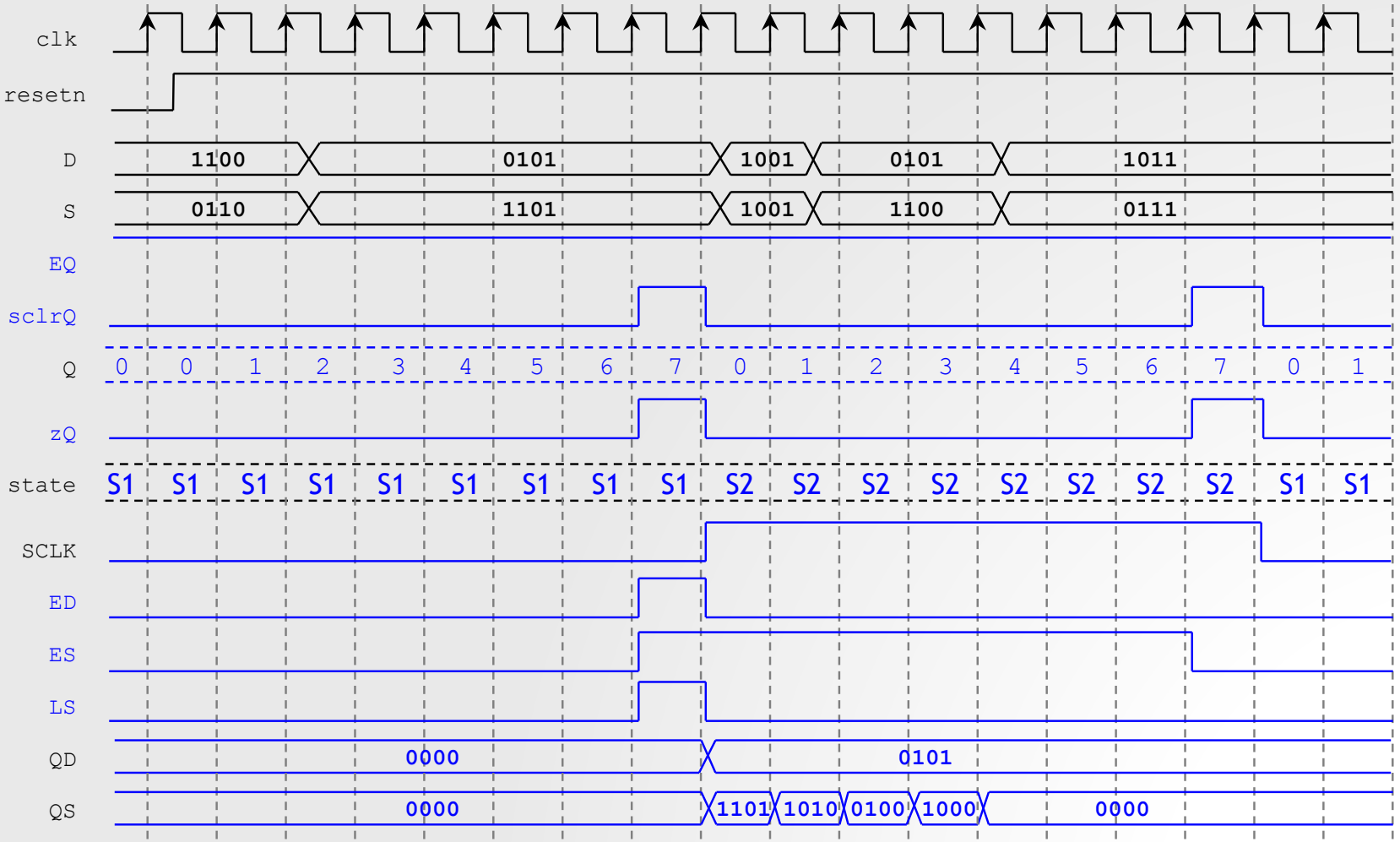
ES ←1, LS ← 1: After these 2 signals are asserted, the shift register will load parallel data on the next clock cycle.

Daniel Llamocca

✓EMBEDDING COUNTERS AND REGISTER IN ASMs

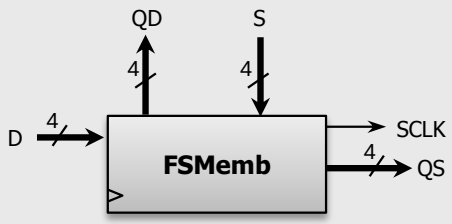
- Timing diagram: `digsys_ex`.

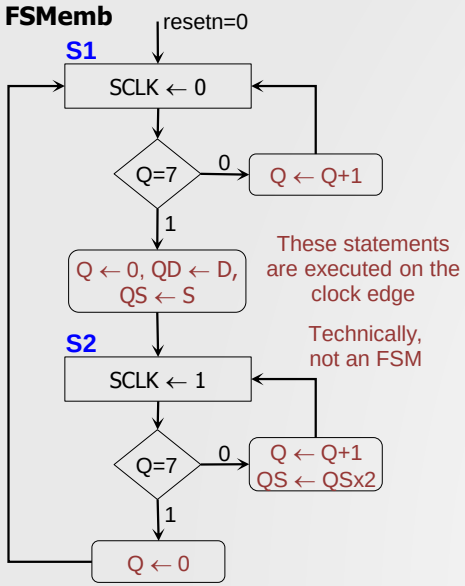
➢ **`dig_sys_ex.zip`**: `my_digsys_ex.vhd, my_genpulse_sclr.vhd, my_rege.vhd, my_pashiftreg.vhd, tb_digsys_ex.vhd.`



✓EMBEDDING COUNTERS AND REGISTER IN ASM diagrams

- **Alternative coding style:** we can embed counters and registers inside the ASM description in VHDL (no need of port map). Note that the resulting circuit is not technically an FSM, since now some of the outputs are registered. For `digsys_ex`, only SCLK is now a combinational output.
- Procedure: Include the statements to infer counters and registers in the State Transitions process. We need to clear their outputs when `resetn=0`.
- New circuit: It is functionally the same as the previous circuit, but the representation is different.





Updated ASM Diagram (**FSMemb**): It is a bit misleading. The statements indicating updates to the counter, register, and shift register outputs do not take effect immediately, but rather on the immediate clock edge.

So, even though this representation can be helpful, it can also be confusing.

✓EMBEDDING COUNTERS AND REGISTER IN ASM diagrams

- VHDL code: Use this coding style sparingly, as it is difficult to debug when there are too many inferred counters and registers.
- The output signals QD, QS, sclk, behave exactly the same as in the original `digsys_ex` circuit (the one with FSM and port map/generic map).

Note that Q is defined as an integer to simplify coding.

QSt: we need this auxiliary signal as we cannot feedback the output QS back to the circuit (we need this for shifting).

- **dig_sys_exp.zip**: my_digsys_ex.vhd, tb_digsys_ex.vhd.

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

entity digsys_ex is
 port ( clock, resetn: in std_logic;
   D,S: in std_logic_vector(3 downto 0);
   QD,QS: out std_logic_vector(3 downto 0);
   sclk: out std_logic);
end digsys_ex;

architecture bhv of digsys_ex is
  type state is (S1, S2);
  signal y: state;
  signal Q: integer range 0 to 7;
  signal QSt: std_logic_vector(3 downto 0);

begin
...
```

✓EMBEDDING COUNTERS AND REGISTER IN ASM diagrams

Note how easy it is to update a counter, a register, or shift register.

Do not forget to assign the values on reset.

This avoids having to use port map instructions.

```
...
Transitions: process (resetn, clock)
  begin
    if resetn = '0' then -- asynchronous signal
       y <= S1; QD<="0000"; Q <= 0; QSt<="0000" -- values on reset
    elsif (clock'event and clock='1') then
       case y is
          when S1 =>
             if Q=7 then y<=S2; Q<=0; QD <= D; QSt <= S;
             else y<=S1; Q <= Q+1; end if;
          when S2 =>
             if Q=7 then y<=S1; Q<=0;
             else
                y<=S2; Q <= Q+1; QSt(0) <= '0';
                QSt(3 downto 1) <= QSt(2 downto 0);
             end if;
       end case;
    end if;
  end process;
  Q <= QSt;
   Outputs: process (y,Q)
   begin
      sclk <= '0';
      case y is
         when S1 =>
         when S2 => sclk <= '1';
       end case;
    end process;
end bhv;
```