FPGA tutorial

Lecture 11

20.01.2021

Jochen Steinmann

III. Physikalisches Institut B

RWTH AACHEN UNIVERSITY

Organisatorical

Solution

ToDo from last time

Lecture 10

A. Implement a synchronous FiFo
   - Status outputs:
     - Full
     - Empty
   - Control inputs:
     - Reset
     - Read enable
     - Write enable

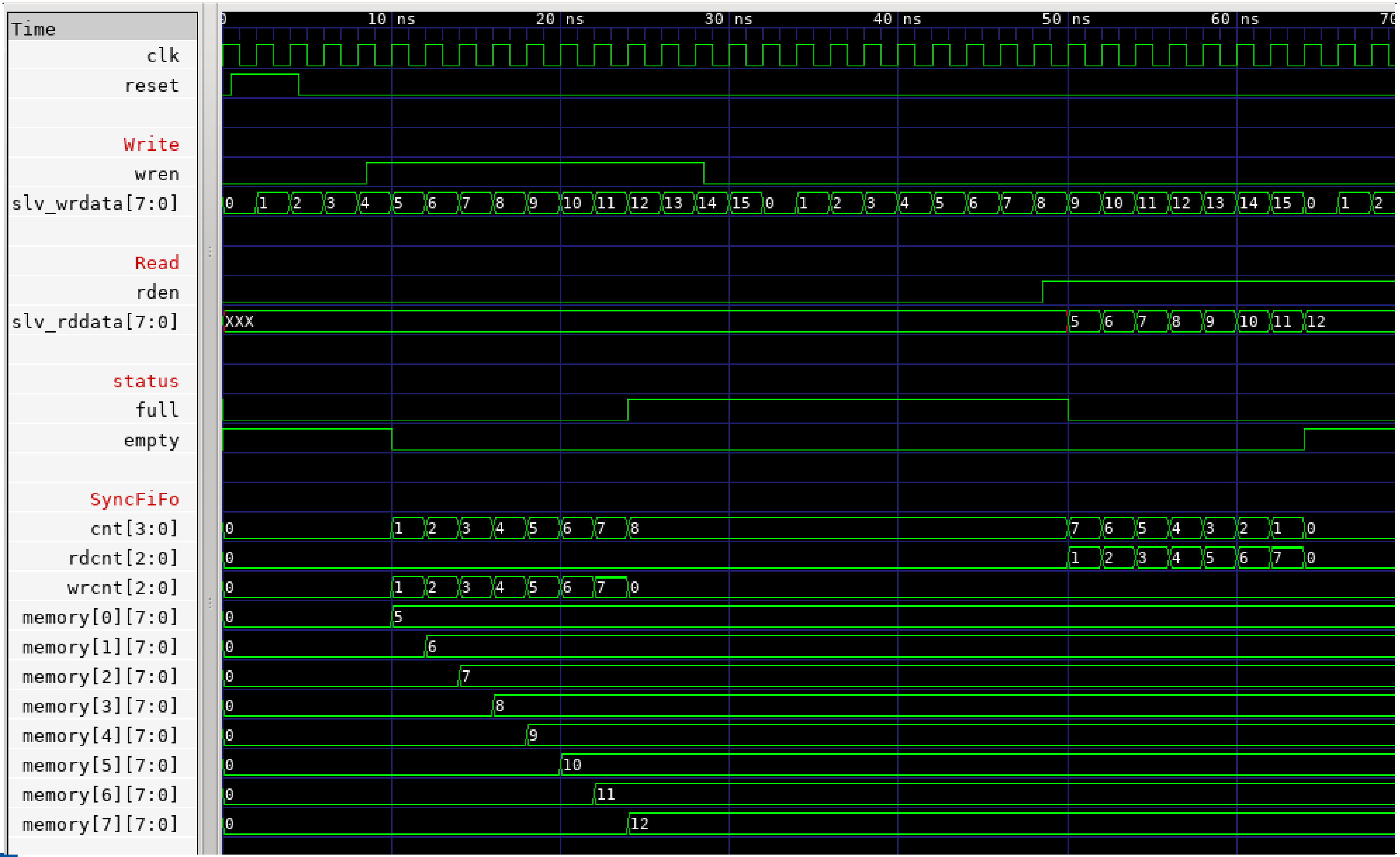
B. Implement a asynchronous FiFo
   - Same inputs and outputs as above
   - BUT: a additional read / write clock
   - **Note:** The full and empty flags are not that trivial
   - Think about why this is not trivial, and about a possible way to solve this.

Synchronous FiFo (Simulation)

```vhdl
architecture behavior of SyncFiFo is
    signal memory : slv8_array_t(7 downto 0) := (others=>(others=>'0'));    -- storage

    -- read and write count 3 bit wide
    signal wrcnt  : unsigned(2 downto 0) := (others => '0');    -- write pointer
    signal rdcnt  : unsigned(2 downto 0) := (others => '0');    -- read pointer

    signal cnt    : unsigned(3 downto 0) := (others => '0');    -- how many elements are in the FiFo?


    signal full   : std_logic := '0';
    signal empty  : std_logic := '0';
begin
```

```vhdl
    process(i_sl_CLK)   -- read and write
    begin
        if rising_edge(i_sl_CLK) then
            if ( i_sl_reset = '1' ) then
                -- reset read and write counter
                rdcnt <= (others => '0');
                wrcnt <= (others => '0');

                cnt   <= (others => '0');    -- reset counter
            else
                if (i_sl_wrEN = '1' and full = '0') then    -- write only, when not full
                    memory( to_integer(wrcnt) ) <= i_slv_wrdata;
                    wrcnt <= wrcnt + '1';
                end if;

                if (i_sl_rdEN = '1' and empty = '0') then   -- read only, when not empty
                    o_slv_rddata <= memory( to_integer(rdcnt)) ;
                    rdcnt <= rdcnt + '1';
                end if;

                -- bookkeeping
                if    (i_sl_wrEN = '1' and i_sl_rdEN = '0' and full = '0') then    -- we are adding elements
                    cnt <= cnt + '1';
                elsif (i_sl_wrEN = '0' and i_sl_rdEN = '1' and empty = '0') then    -- we are removing elements
                    cnt <= cnt - '1';
                end if;
            end if;
        end if;
    end process;

    empty <= '1' when ( cnt = x"0" ) else '0';
    full  <= '1' when ( cnt = x"8" ) else '0';
```

Synchronous FiFo (Simulation)



III. Physikalisches Institut B

RWTH AACHEN UNIVERSITY

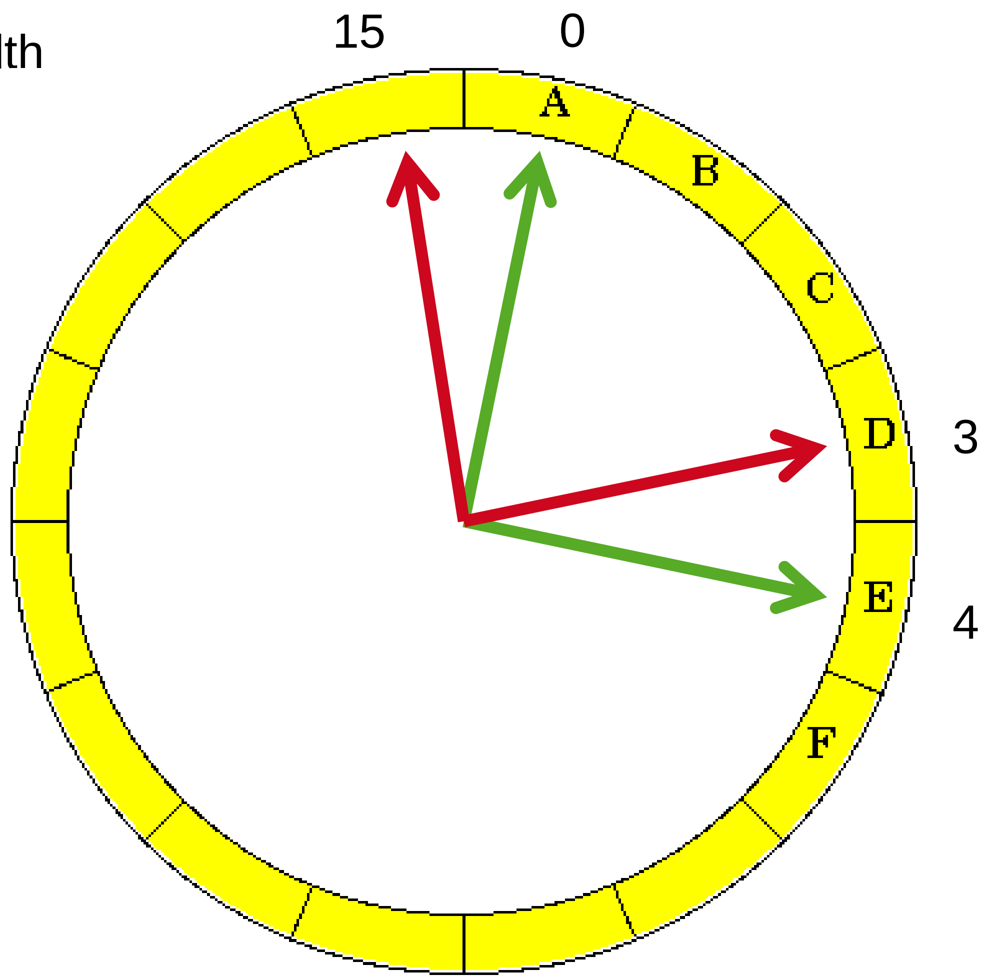
Asynchronous FiFo

```
architecture behavior of SyncFiFo is
    signal memory : slv8_array_t(7 downto 0) := (others=>(others=>'0'));    -- storage

    -- read and write count 3 bit wide
    signal wrcnt  : unsigned(2 downto 0) := (others => '0');    -- write pointer
    signal rdcnt  : unsigned(2 downto 0) := (others => '0');    -- read pointer

    signal cnt    : unsigned(3 downto 0) := (others => '0');    -- how many elements are in the FiFo?

    signal full   : std_logic := '0';
    signal empty  : std_logic := '0';
begin
```

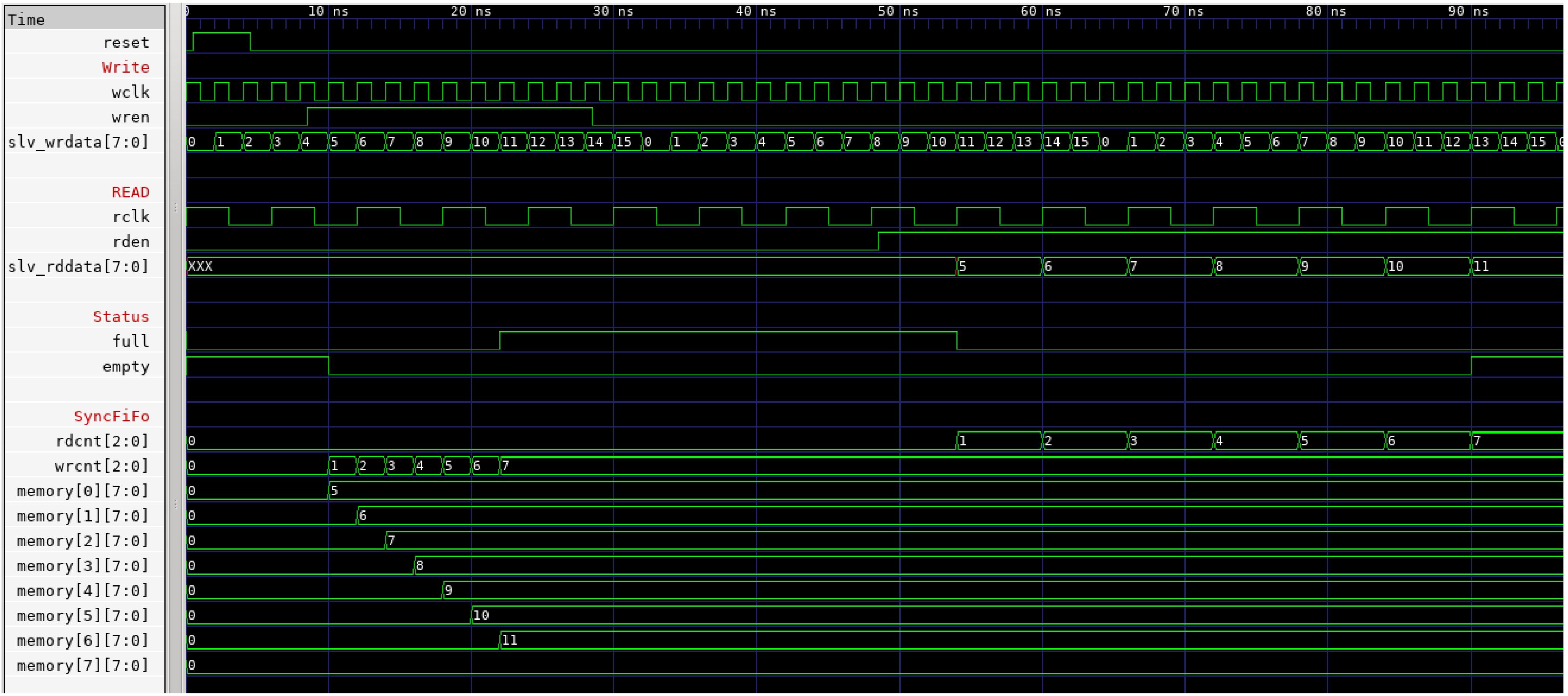
```
    empty <= '1' when ( wrcnt = rdcnt ) else '0';
    full  <= '1' when ( rdcnt = wrcnt + '1' ) else '0';

    o_sl_empty <= empty;
    o_sl_full  <= full;

    process(i_sl_wCLK)   -- write
    begin
        if rising_edge(i_sl_wCLK) then
            if ( i_sl_reset = '1' ) then
                wrcnt <= (others => '0');
            else
                if (i_sl_wrEN = '1' and full = '0') then    -- write only, when not full
                    memory( to_integer(wrcnt) ) <= i_slv_wrdata;
                    wrcnt <= wrcnt + '1';
                end if;
            end if;
        end if;
    end process;

    process(i_sl_rCLK)   -- write
    begin
        if rising_edge(i_sl_rCLK) then
            if ( i_sl_reset = '1' ) then
                rdcnt <= (others => '0');
            else
                if (i_sl_rdEN = '1' and empty = '0') then   -- read only, when not empty
                    o_slv_rddata <= memory( to_integer(rdcnt)) ;
                    rdcnt <= rdcnt + '1';
                end if;
            end if;
        end if;
    end process;


end behavior;
```

Full / Empty

- Using the features of the memory
  - Storing 8 elements => 3bit address width
- Keep in mind:
  - 15 + 1 = 0 (for a 4 bit unsigned)
  - 7 + 1 = 0 (for a 3 bit unsigned)



Asynchronous FiFo (Simulation)



Drawbacks of this method?

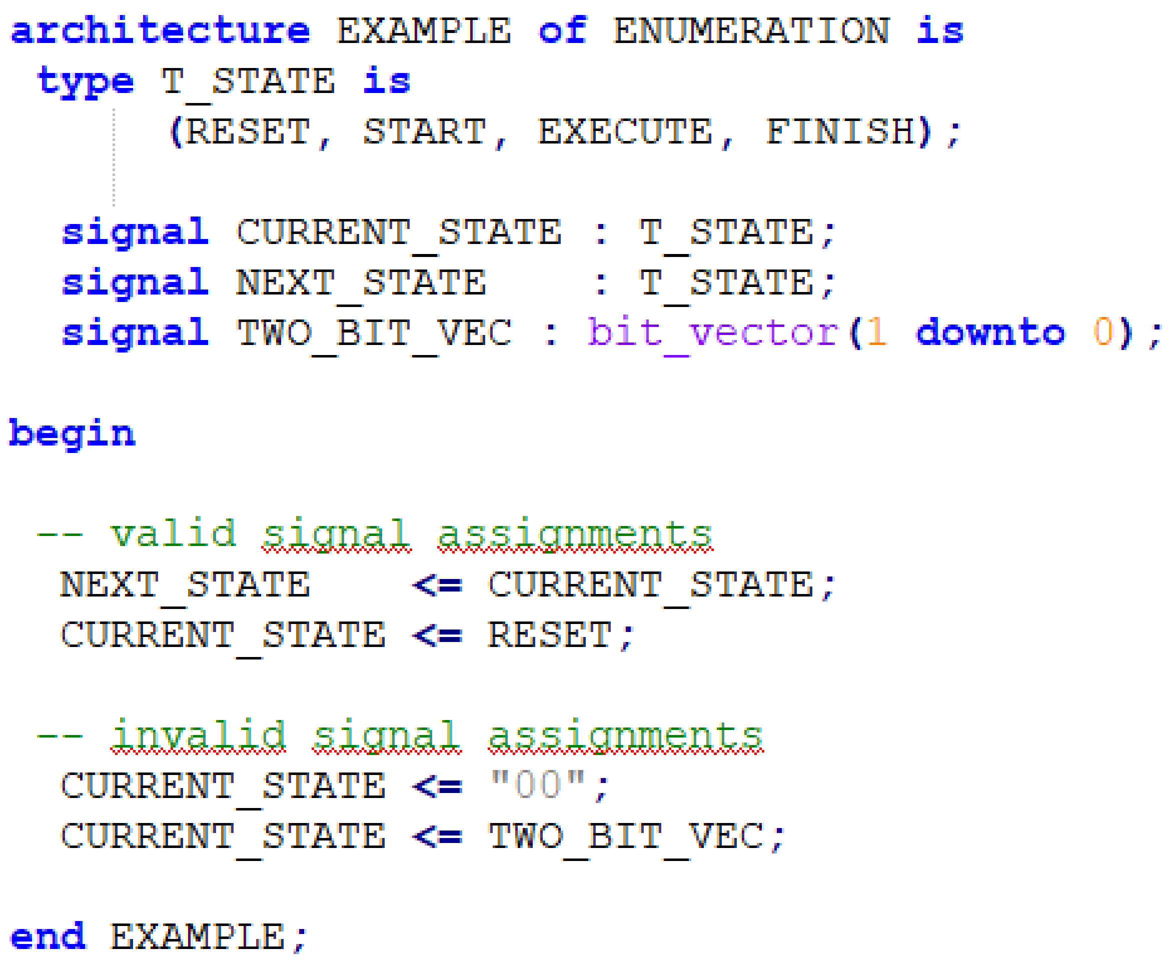
What do you think?

NEW

VHDL

Type definition enum

- Synthesis tools map enumerations onto a suitable bit pattern automatically



```
architecture EXAMPLE of ENUMERATION is
 type T_STATE is
      (RESET, START, EXECUTE, FINISH);

  signal CURRENT_STATE : T_STATE;
  signal NEXT_STATE    : T_STATE;
  signal TWO_BIT_VEC : bit_vector(1 downto 0);

begin

 -- valid signal assignments
  NEXT_STATE    <= CURRENT_STATE;
  CURRENT_STATE <= RESET;

 -- invalid signal assignments
  CURRENT_STATE <= "00";
  CURRENT_STATE <= TWO_BIT_VEC;

end EXAMPLE;
```

Finite State Machines

Doing different tasks

Finite State Machines

- Until now only „programs“ with always the same behaviour
- Sometimes, it is neccessary to react on different situations
  - If triggered – read out the detector
  - If done – wait for next trigger and do nothing
- Useful:
  - Finite State Machines
- In this course: No complete introduction on FSM

FSM

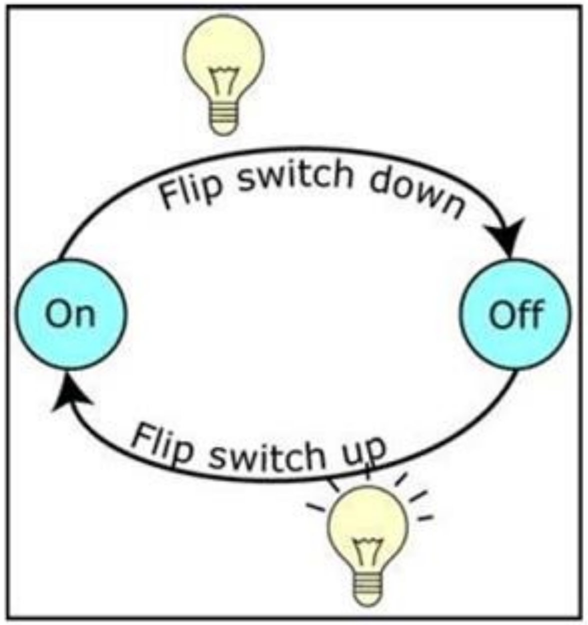
In General

- FSM always needs an event to change state!
- **very simple model**
  - Two States (ON / OFF)
  - Two Events (DOWN / UP)

- Each state consists of:
  - Input action
    - Action done, when entering the state

  - State action
    - Action done, when beeing inside the state

  - Output action
    - Action done, when leaving the state



ON

IA:	**switch light on**
SA:	**keep light on**
OA:	**nothing**

OFF

IA:	**switch light off**
SA:	**keep light off**
OA:	**nothing**

Definition of an Event

- Event causes FSM to change the state
  - Can be clock, buttons or other trigger information
  - But also more complex structures

Definition of a state

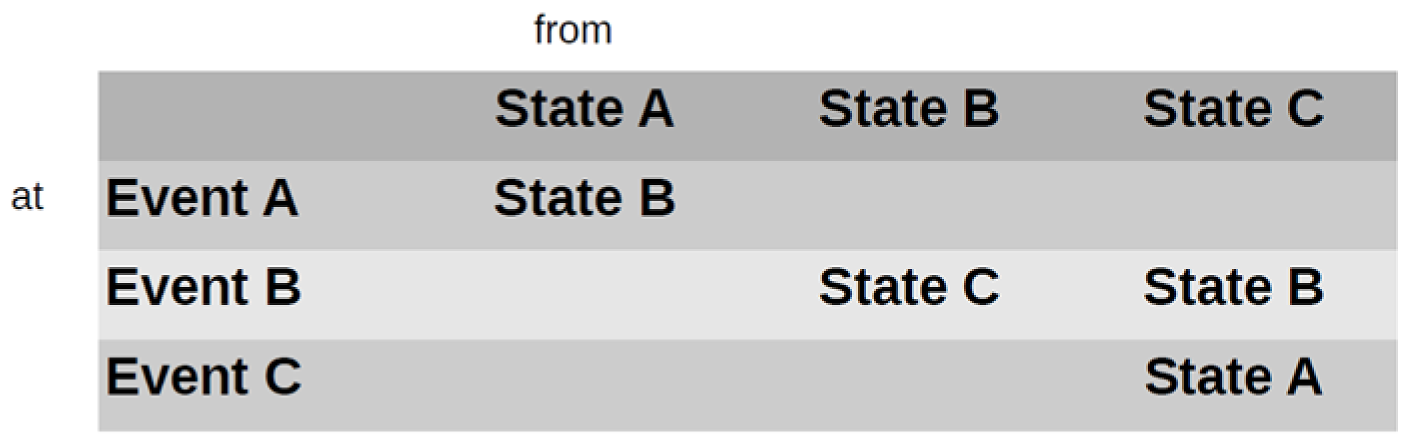
- Input action
    - Output, when entering the state
- Output action
    - Output when leaving the state
- Action during the state

Advantage of FSMs

- Why FSMs are nice:
  - For given events, they always behave the same
  - If their state is known, you know what they are doing
- Usage of FSMs
  - Decoding of protocols
    - start – data – stop
  - Different operation modes
    - Initialisation
    - Run
    - Shutdown
  - User interactions
    - Menu structure

Planning of a FSM

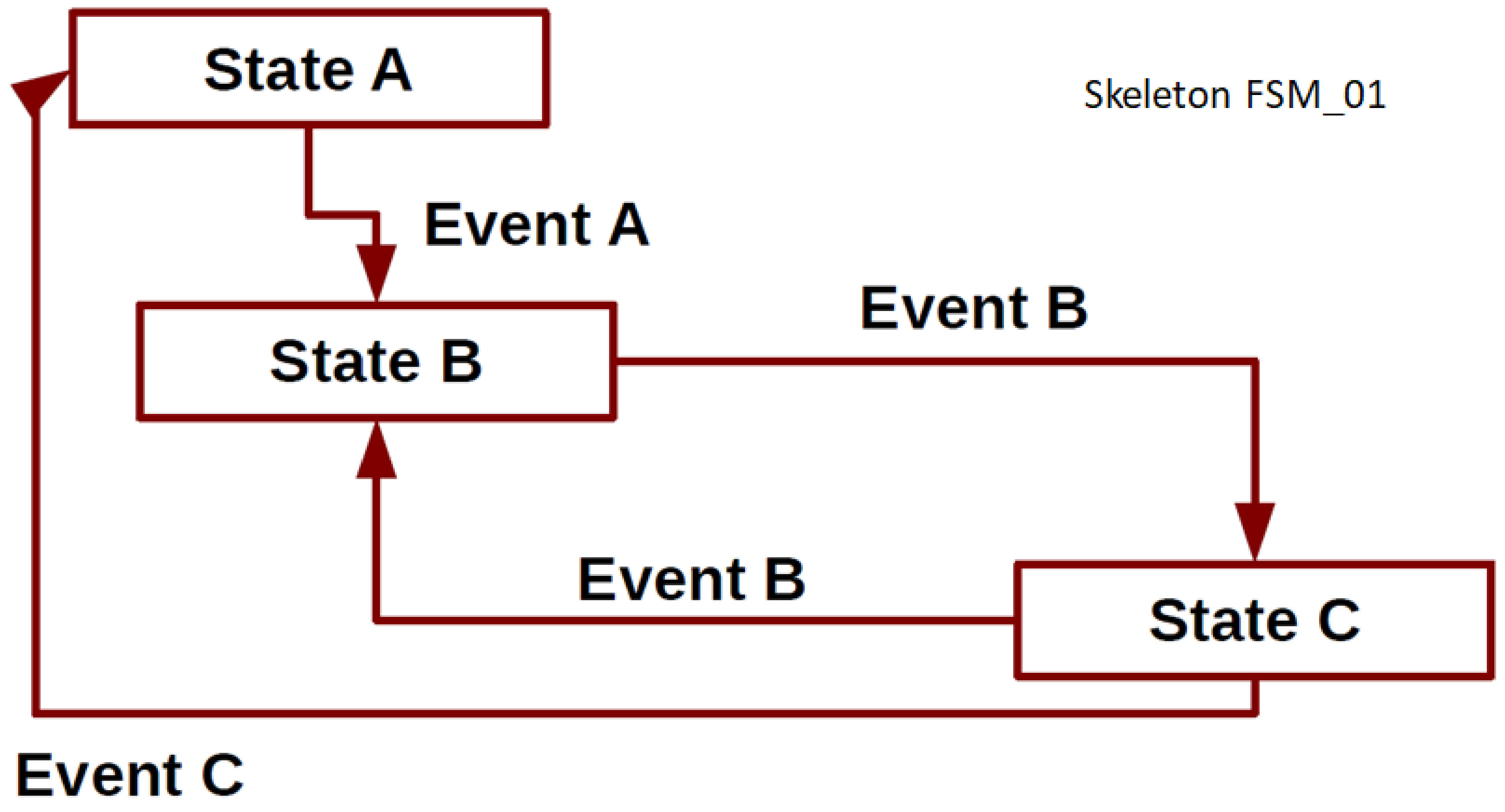
- State and transition table is very helpful
- States and events and their actions are listed in the table



from

		State A	State B	State C
at	Event A	State B		
	Event B		State C	State B
	Event C			State A

FSM Overview

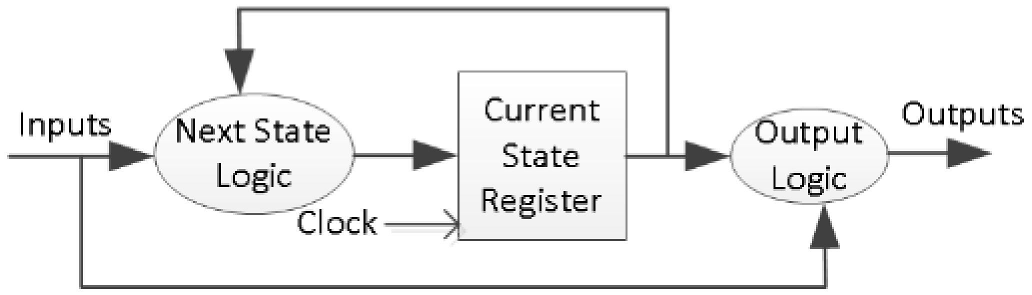


Skeleton FSM_01

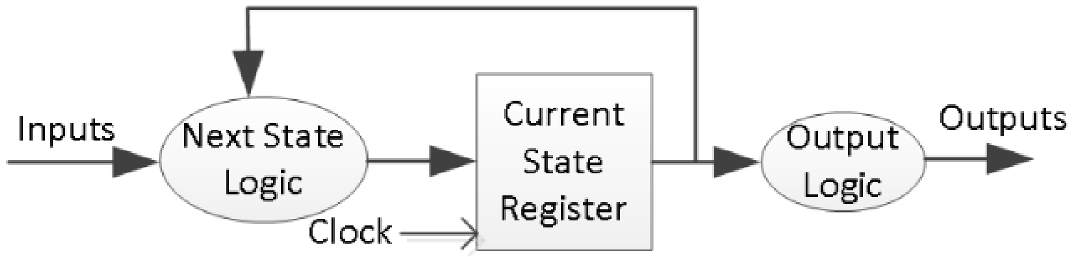
Three blocks State machines

Some special kind of FSM implementation

- Three processes handling different things



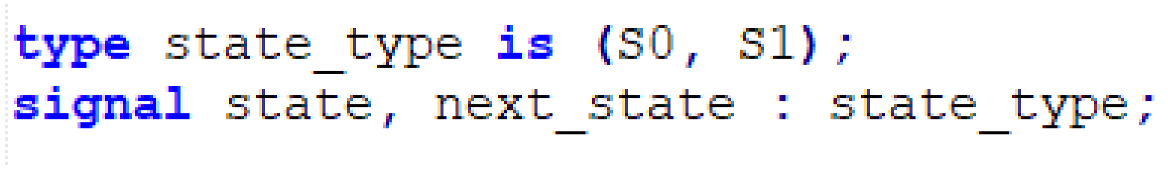
Marley FSM:
Outputs depend on states + inputs



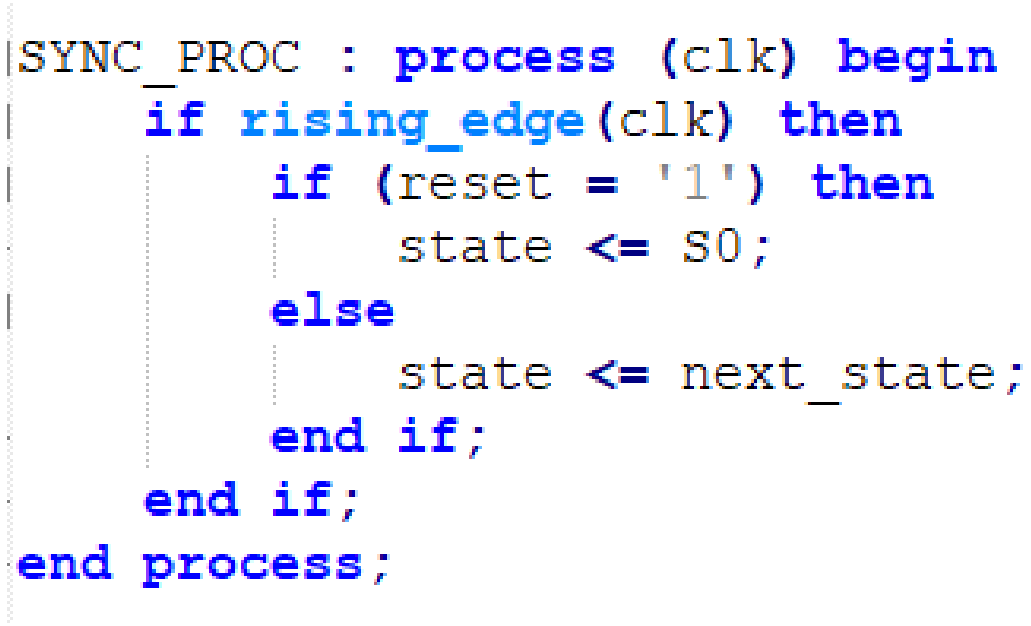
Moore FSM:
Outputs depend on states

Process 1

Handling the next state



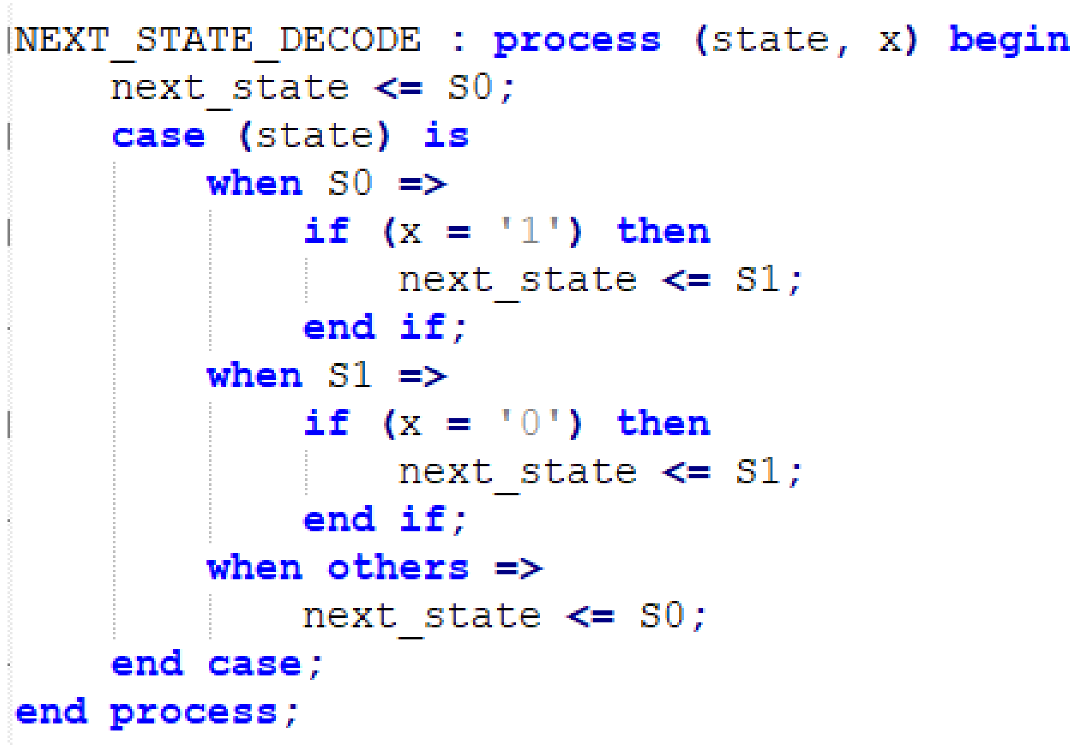
```
type state_type is (S0, S1);
signal state, next_state : state_type;
```



```
SYNC_PROC : process (clk) begin
    if rising_edge(clk) then
        if (reset = '1') then
            state <= S0;
        else
            state <= next_state;
        end if;
    end if;
end process;
```

Process 2

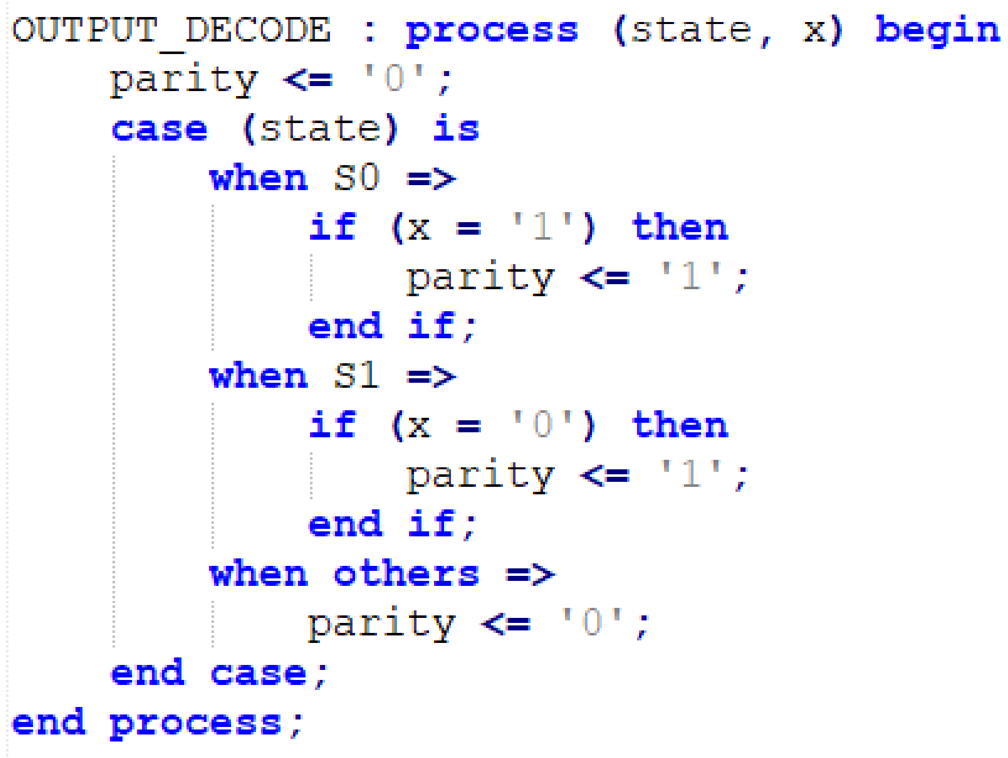
Preparing the next state



```
NEXT_STATE_DECODE : process (state, x) begin
    next_state <= S0;
    case (state) is
        when S0 =>
            if (x = '1') then
                next_state <= S1;
            end if;
        when S1 =>
            if (x = '0') then
                next_state <= S1;
            end if;
        when others =>
            next_state <= S0;
    end case;
end process;
```

Process 3

Handling the output

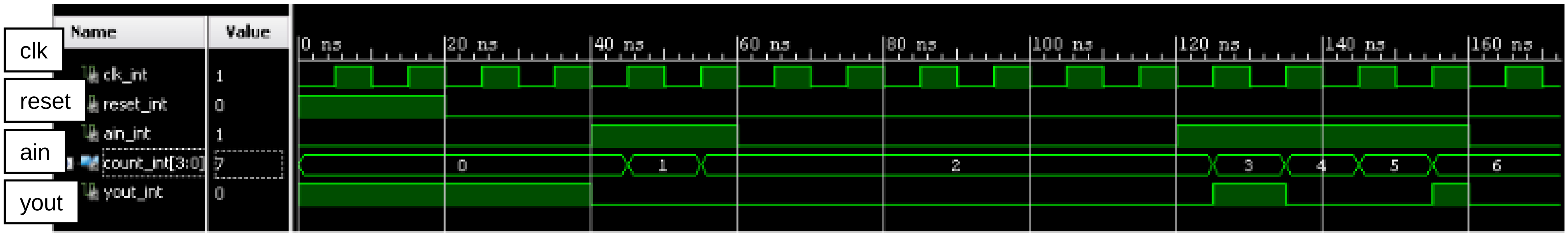


```
OUTPUT_DECODE : process (state, x) begin
    parity <= '0';
    case (state) is
        when S0 =>
            if (x = '1') then
                parity <= '1';
            end if;
        when S1 =>
            if (x = '0') then
                parity <= '1';
            end if;
        when others =>
            parity <= '0';
    end case;
end process;
```

ToDo Lecture 11

A. Implement the three process **Marley** FSM presented in these slides

B. Design a sequence detector implementing a Mealy state machine using three blocks.
The Mealy state machine has one input (ain) and one output (yout).
The output yout is 1 if and only if the total number of 1s received is divisible by 3 (hint: 0 is inclusive, however, reset cycle(s) do not count as 0- see in simulation waveform time=200).
Develop a testbench and verify the model through a behavioral simulation.



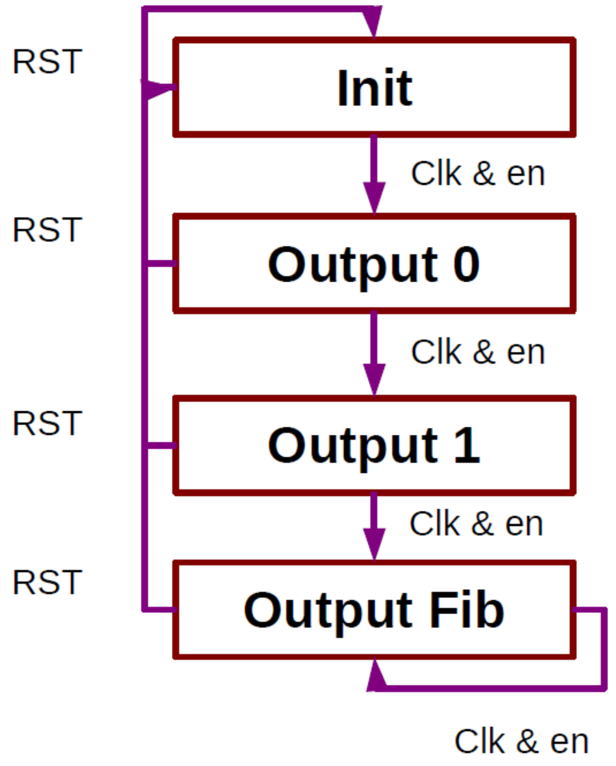
ToDo Lecture 11

C. Implement a FSM outputting the Fibonacci numbers
Reminder: 0 – 1 – 1 – 2 – 3 – 5 – … – (n-1 + n-2)

- States:
  - 1 init
  - 2 output 0
  - 3 output 1
  - 4 create Fibonacci number
  - 5 keep output

  5 can be ignored depending on your implementation

- Events:
  - 1: CLK = btnD
  - 2: en
  - 3: Reset



There are various ways, how to approach this.

Please let me know, whether you completed C!

Thank you!