FPGA tutorial

Lecture 14

10.02.2021

Jochen Steinmann

III. Physikalisches Institut B

RWTH AACHEN UNIVERSITY

III. Physikalisches Institut B

RWTH AACHEN UNIVERSITY

ToDo Lecture 13

General constrains
assume usage of a 9 bit DAC (using two‘s complement)

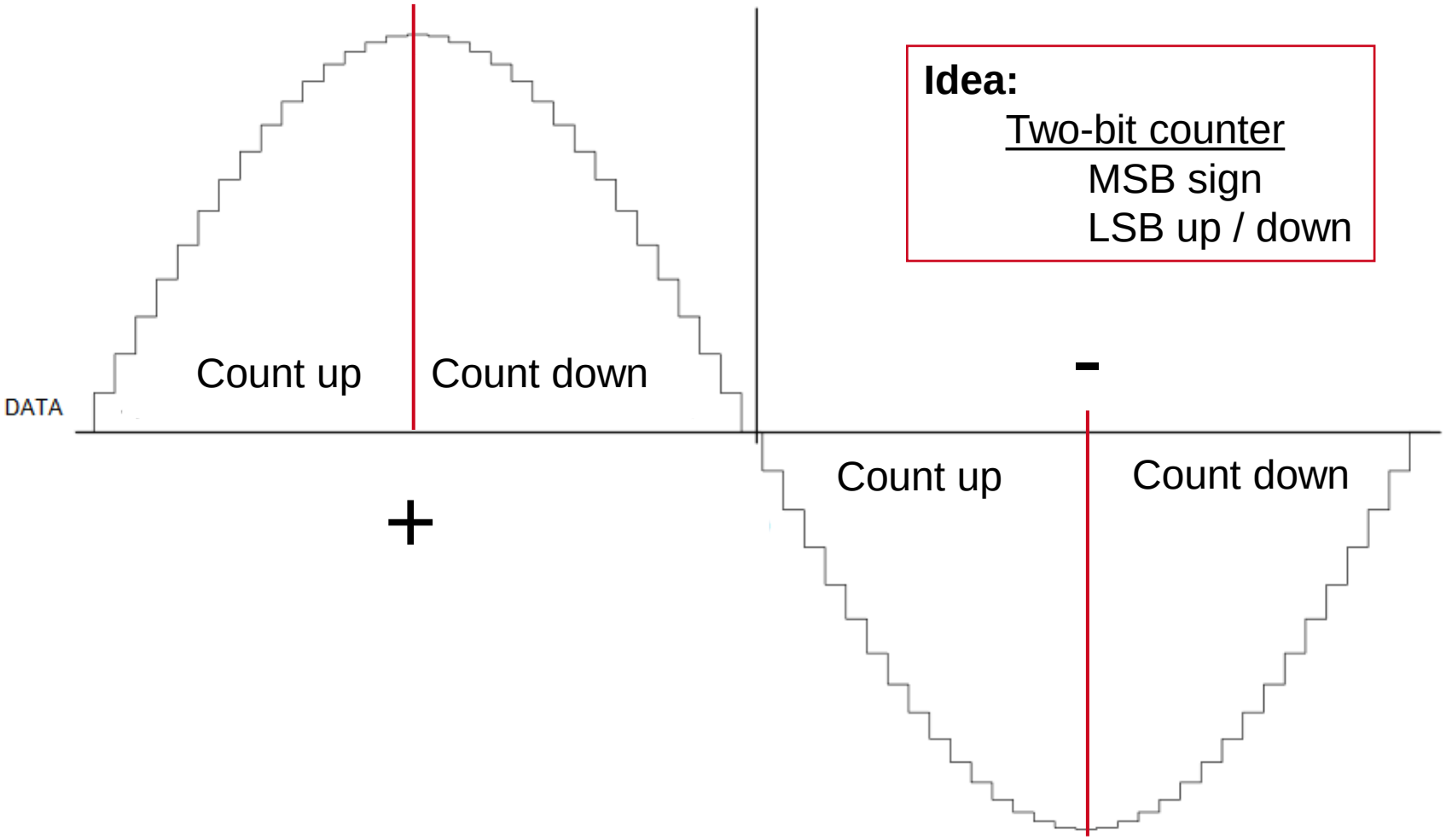
A. Create a tool for generating the sine wave lookup table (Python, C++ etc.)
   16 entries are sufficient (we need 8bit resolution)

B. Implement a DDS for generating a sine wave
   1. Counter for address of the LUT (you can reuse your modules – lecture 3 / lecture 6)
   2. Output from lookup table
   3. Encoding of positive / negative

C. Use your sine wave generator to provide
   1. Two different frequencies
   2. Two different phases per frequency (something different than n x 90°)
      use a phase difference of 120°

Sine wave generation

Using the symmetries



Idea:

Two-bit counter

MSB sign

LSB up / down

Lecture 13

Source I

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

----------------------------------------

entity dds is
    port(
        i_sl_CLK      :  in  std_logic;                                  -- clock
        i_sl_en       :  in  std_logic;                                  -- enable
        o_slv_out     :  out std_logic_vector(8  downto 0)   -- counter
    );
end dds;

----------------------------------------

architecture behavior of dds is

    component counter is
        generic(
            DATA_WIDTH    :  integer := 8
        );
        port(
            i_sl_CLK      :  in  std_logic;                                  -- clock
            i_sl_en       :  in  std_logic;                                  -- enable
            i_sl_dir      :  in  std_logic;                                  -- direction
            i_sl_rst      :  in  std_logic;                                  -- reset
            o_slv_counter :  out std_logic_vector(DATA_WIDTH-1  downto 0)   -- counter
        );
    end component counter;

    component sin_lut is
        port(
            i_slv_addr  :  in  std_logic_vector(3 downto 0); -- lut address
            o_slv_value :  out std_logic_vector(7 downto 0)  -- lut content
        );
    end component sin_lut;


    signal phase : unsigned(1 downto 0) := (others => '0'); -- phase(0) direction of counter
                                                            -- phase(1) sign of output

    signal dds_addr     : std_logic_vector(3 downto 0) := (others => '0');  -- addr of the DDS LUT
    signal dds_addr_old : std_logic_vector(3 downto 0) := (others => '0');  -- old addr

    signal sin_value    : std_logic_vector(7 downto 0) := (others => '0');  -- output of the DDS LUT
    signal sin_value2   : std_logic_vector(8 downto 0) := (others => '0');  -- output of the DDS LUT 2s complement
begin
```

Lecture 13

Source II

```
-- The Device Under Test (DUT)
i_counter : counter
    generic map(
        DATA_WIDTH    => 4
    )
    port map(
        i_sl_CLK      => i_sl_CLK,
        i_sl_en       => i_sl_en,
        i_sl_dir      => phase(0),
        i_sl_rst      => '0',
        o_slv_counter => dds_addr
    );


i_sin_lut : sin_lut
port map(
    i_slv_addr  => dds_addr,
    o_slv_value => sin_value(7 downto 0)
);

p_phase : process(dds_addr) begin
    if dds_addr = x"F" then
        phase <= phase + '1';

    elsif dds_addr = x"0" then      -- at zero, we have to change as well
        if dds_addr_old = x"1" then
            phase <= phase + '1';
        end if;
    end if;
    dds_addr_old <= dds_addr;   -- save value
end process;

p_value : process(sin_value) begin
    if phase(1) = '0' then
        sin_value2 <= '0' & sin_value;
    else
        if sin_value = x"00" then
            sin_value2 <= (others => '0');
        else
            sin_value2 <= '1' & not std_logic_vector( unsigned(sin_value) -1 );   -- build twos complement
        end if;
    end if;
end process;

o_slv_out <= sin_value2;

end behavior;
```

Lecture 13

Source III

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

----------------------------------------

entity sin_lut is
    port(
        i_slv_addr  :  in  std_logic_vector(3 downto 0); -- lut address
        o_slv_value :  out std_logic_vector(7 downto 0)  -- lut content
    );
end sin_lut;

----------------------------------------

architecture behavior of sin_lut is
    signal addr : integer range 0 to  15;
    signal outp : integer range 0 to 255;
begin

    -- some signals to keep the case statement simple
    addr <= to_integer( unsigned( i_slv_addr ) );
    o_slv_value <= std_logic_vector( to_unsigned( outp, 8));

    process(addr)
    begin
        case addr is
            when 0 =>    outp <=   0;
            when 1 =>    outp <=  26;
            when 2 =>    outp <=  53;
            when 3 =>    outp <=  78;
            when 4 =>    outp <= 103;
            when 5 =>    outp <= 127;
            when 6 =>    outp <= 149;
            when 7 =>    outp <= 170;
            when 8 =>    outp <= 189;
            when 9 =>    outp <= 206;
            when 10 =>   outp <= 220;
            when 11 =>   outp <= 232;
            when 12 =>   outp <= 242;
            when 13 =>   outp <= 249;
            when 14 =>   outp <= 253;
            when 15 =>   outp <= 255;
        end case;
    end process;

end behavior;
```

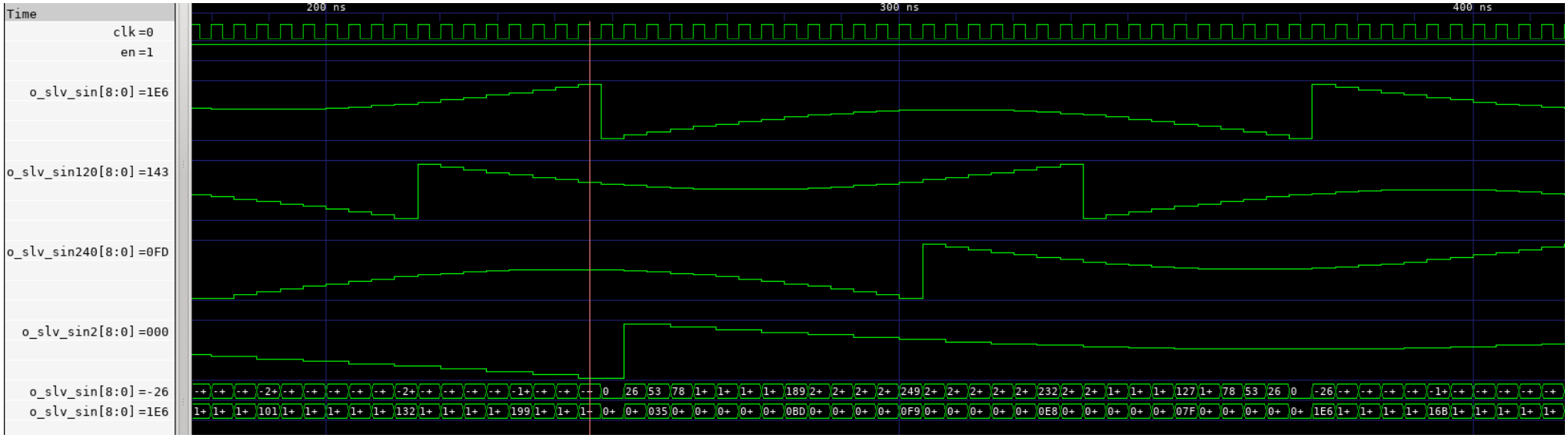
Calculated by python

Source IV

```
entity counter is
    generic(
        DATA_WIDTH    :  integer := 8
    );
    port(
        i_sl_CLK      :  in  std_logic;                                  -- clock
        i_sl_en       :  in  std_logic;                                  -- enable
        i_sl_dir      :  in  std_logic;                                  -- direction
        i_sl_rst      :  in  std_logic;                                  -- reset
        o_slv_counter :  out std_logic_vector(DATA_WIDTH-1  downto 0)    -- counter
    );
end counter;

----------------------------------------

architecture behavior of counter is
    signal cnt : unsigned(DATA_WIDTH-1 downto 0) := (others => '0');  -- define signal, because we cannot readback outputs
begin

    process(i_sl_CLK)
    begin
        if rising_edge(i_sl_CLK) then
            if (i_sl_en = '1') then
                if( i_sl_rst = '1' ) then       -- option 1 ( reset when enabled )
                    cnt <= (others => '0');
                else                            -- we have to make sure, that we
                                                -- are not resetting and counting
                                                -- at the same time
                    if(i_sl_dir = '0') then
                        cnt <= cnt + 1;
                    else
                        cnt <= cnt - 1;
                    end if;
                end if;
            elsif ( i_sl_rst = '1') then        -- option 2 ( reset when disabled )
                cnt <= (others => '0');
            end if;
--          o_slv_counter <= std_logic_vector(cnt); -- option 1a  ( o_slv_counter is always 1 clock cycle late related to cnt )
        end if;
--        o_slv_counter <= std_logic_vector(cnt);     -- option 1b  ( o_slv_counter is always 1 clock cycle late related to cnt )
    end process;

    o_slv_counter <= std_logic_vector(cnt);     -- option 2 ( o_slv_counter and cnt have the same value at the same clock cycle)

end behavior;
```

Simulation output

Lecture 13b + c



III. Physikalisches Institut B

RWTH AACHEN UNIVERSITY

Repetition

Everything is important for the exam!
Not only what I present in the following slides

VHDL

```
-----------------------------------------
-- lecture 01 simple gate
-- just forwarding the signal
-----------------------------------------

library ieee;
use ieee.std_logic_1164.all;

-----------------------------------------

entity lecture01 is
port(
          i_sl_x : in  std_logic;
          o_sl_F: out std_logic
);
end lecture01;

-----------------------------------------

architecture behaviour of lecture01 is
begin

    process(i_sl_x)
    begin
        -- compare to truth table
        if (i_sl_x = '1') then
            o_sl_F <= '1';
        else
            o_sl_F <= '0';
        end if;
    end process;
end behaviour;

-----------------------------------------
```

last line in port list without ;

<= assign a value to **o_sl_F**

Working with multiple bits at the same time

std_logic_vector

- Up to now, we just could handle a single bit by using std_logic
- std_logic_vector is a vector of std_logic

i_slv_test : in std_logic_vector (7 downto 0)

Highest bit

Lowest bit

Elements can be accessed by:

i_slv_test(4) – returns single element
i_slv_test(4 downto 1) – returns elements 4,3,2,1

How to assign values to std_logic_vector

```
signal Slv1 : std_logic_vector(7 downto 0);
signal Slv2 : std_logic_vector(7 downto 0) := (others => '0');
signal Slv3 : std_logic_vector(7 downto 0) := (others => '1');
signal Slv4 : std_logic_vector(7 downto 0) := x"AA";
signal Slv5 : std_logic_vector(7 downto 0) := "10101010";
signal Slv6 : std_logic_vector(7 downto 0) := "00000001";
```

undefined

all 0

all 1

Hexadecimal value

Binary values

```
Slv1    <=    slv2(7 downto 4) & slv3(3 downto 0);    -- concatenation
```

results in “00001111“ upper 4 bits from slv2 and lower 4 bits from slv3

Signals in VHDL

- VHDL cannot read back outputs
- Means, if we e.g. want to build a shift register or a counter, we need a internal signal, which handles the data

- When we have done the modification, we assign the value to the output

```
architecture behavior of lecture02 is
   signal shift_reg : std_logic_vector(7 downto 0) := "00000001";
            -- define signal, because we cannot readback outputs
begin

   process(i_sl_CLK)
   begin
      if rising_edge(i_sl_CLK) then

            -- do something with shift_reg

            o_slv_shift <= shift_reg;
      end if;
   end process;
end behaviour;
```

Handling the clock in VHDL

- The clock should be the only signal on the sensitivity list!
- Actions typically happen on the rising edge (transition from 0 to 1)

```
process(i_sl_CLK)
   begin
      if rising_edge(i_sl_CLK) then
```

Handling the clock in VHDL II

- Implementing a enable

```
architecture behaviour of lecture02 is
    signal shift_reg : std_logic_vector(7 downto 0) := "00000001";
                -- define signal, because we cannot readback outputs
begin

    process(i_sl_CLK)
    begin
        if rising_edge(i_sl_CLK) then
            if (i_sl_en = '1') then

            end if;
            o_slv_shift <= shift_reg;
        end if;
    end process;

end behaviour;
```

Calculation in VHDL

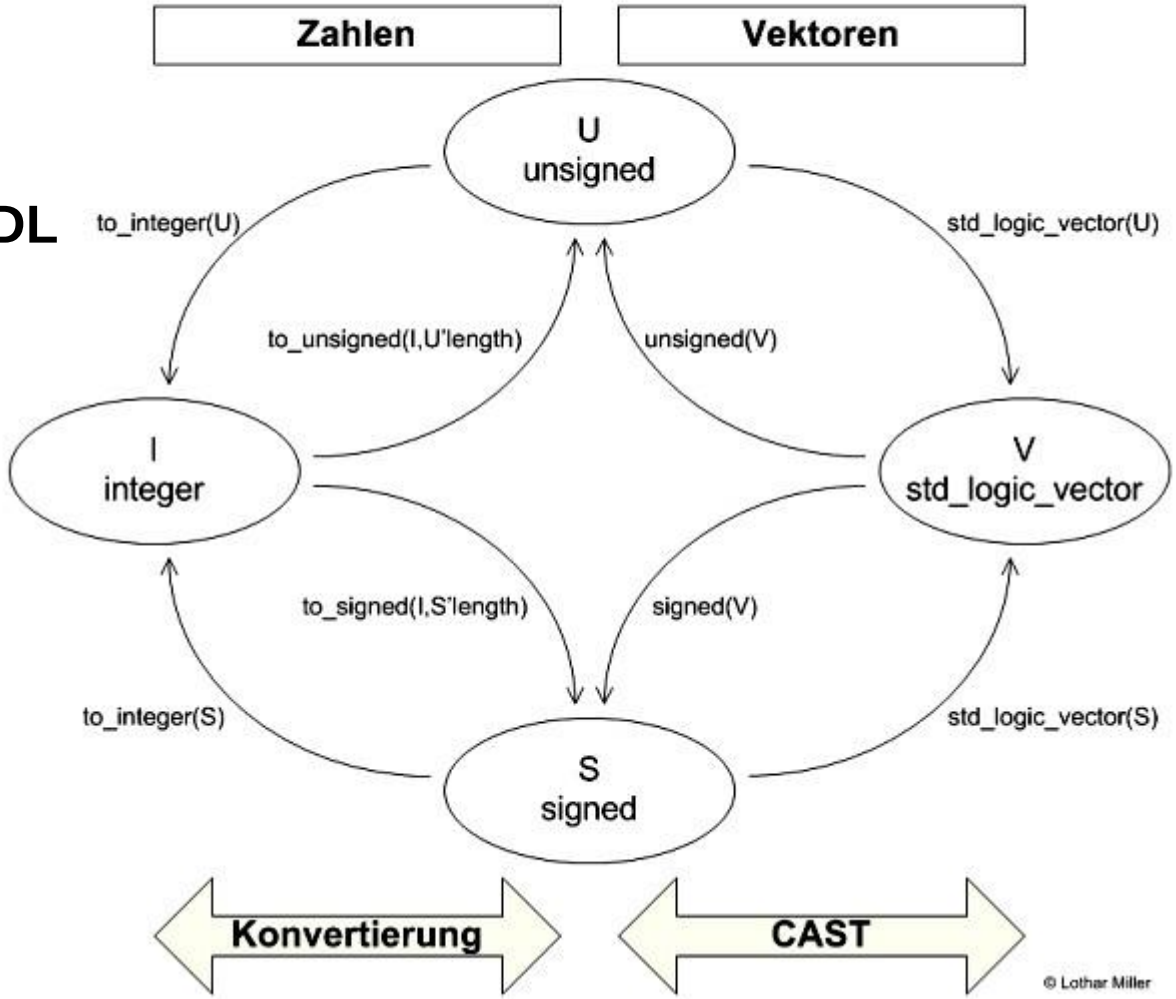
- Not possible using
  std_logic_vector

Have to convert / cast to unsigned

Recipe to do calculations in VHDL

1. Cast std_logic_vector to unsigned
2. Do calculation
3. Cast result back to std_logic_vector

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

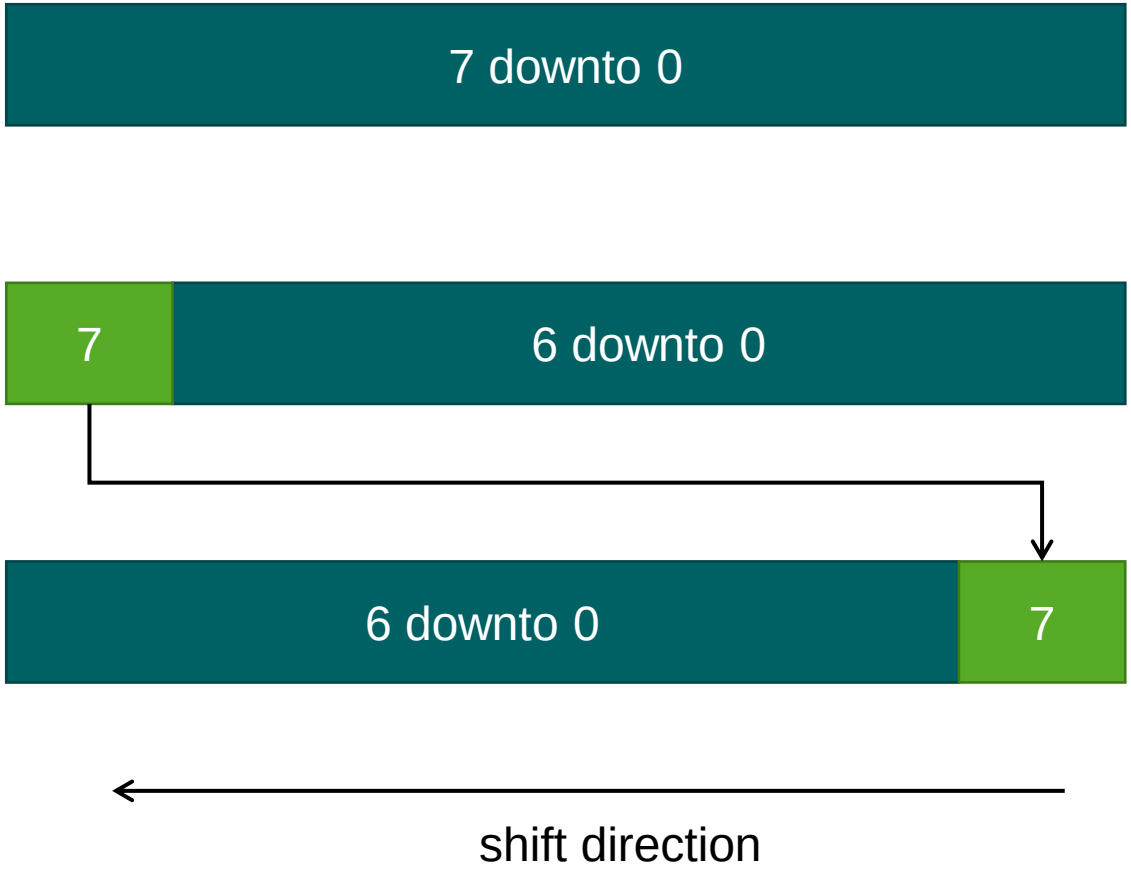


Shift register

Solution 2a

```
entity lecture02 is
    port(
        i_sl_CLK     :  in  std_logic;                      -- clock
        i_sl_en      :  in  std_logic;                      -- enable
        i_sl_dir     :  in  std_logic;                      -- direction
        o_slv_shift  :  out std_logic_vector(7 downto 0)    -- shift register
    );
end lecture02;

----------------------------------------

architecture behavior of lecture02 is
    signal shift_reg : std_logic_vector(7 downto 0) := "00000001";  -- define signal, because we cannot readback outputs
begin

    process(i_sl_CLK)
    begin
        if rising_edge(i_sl_CLK) then
            if (i_sl_en = '1') then
                if(i_sl_dir = '1') then
                    shift_reg <= shift_reg(6 downto 0) & shift_reg(7);
                else
                    shift_reg <= shift_reg(0) & shift_reg(7 downto 1);
                end if;
            end if;
            o_slv_shift <= shift_reg;
        end if;
    end process;

end behavior;
```

Explanation of shift operation

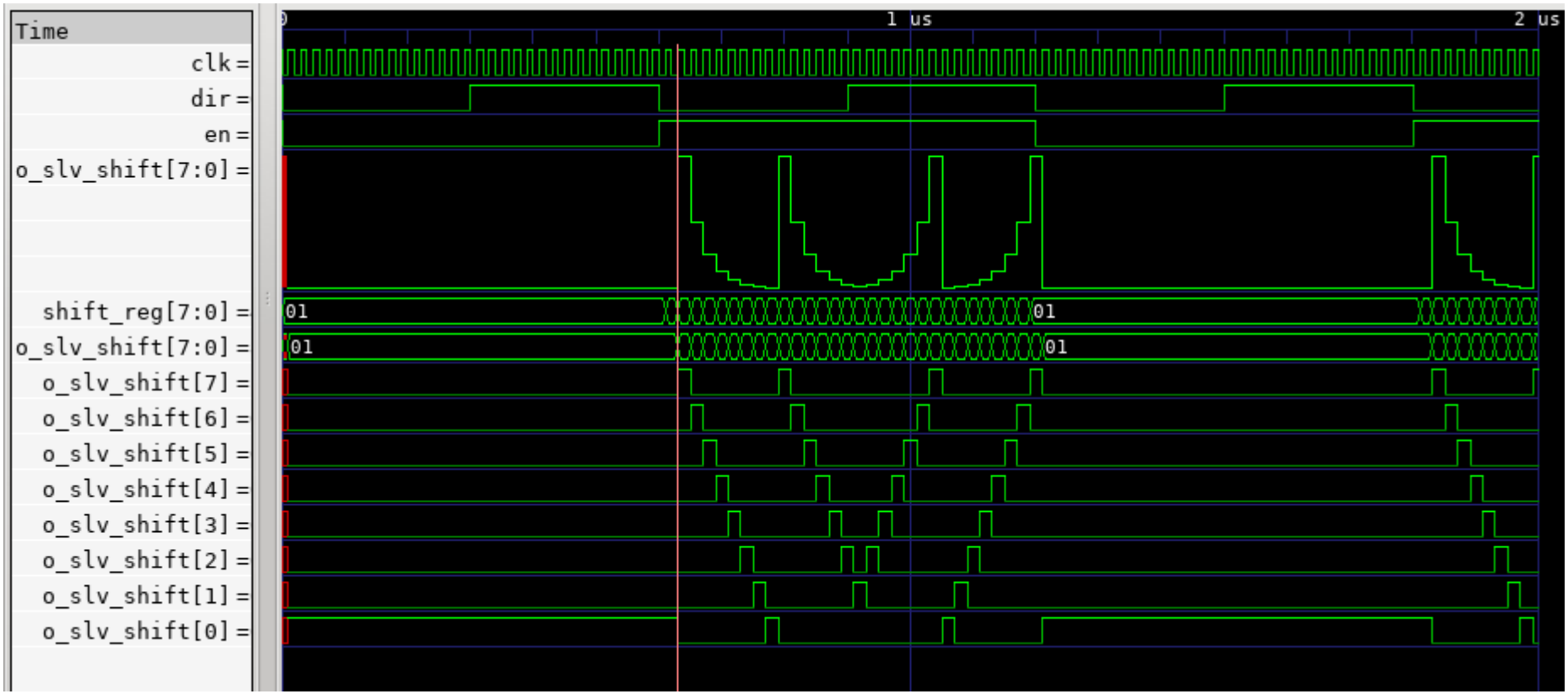


split bus in bit 7 and rest

move bit 7 to right position

Solution 2a

Simulation output



III. Physikalisches Institut B

RWTH AACHEN UNIVERSITY

Assignments in VHDL

A more detailed view

Process „runs“, when a,b,c changes!

```
process(a, b, c)
begin
        y <= '0';
        if a = '1' or b = '1' then y <= '1'; end if;
        if c = '1' then y <= '0'; end if;
end process;
```

After an event in a, b or c:

1. new value '0' is scheduled for y, but y still has its old value
2. if a or b are '1', the new scheduled value of y is changed to '1'
3. if c is '1', the scheduled value of y is changed to '0'
4. at the end of the process the scheduled value is assigned to y
5. as y is not on the sensitivity list, the process is suspended until the next event on a, b or c, and y preserves its new value

All this happens within the same simulation time!

Solution 03b

Counter with RESET

```vhdl
entity counter is
    generic(
        DATA_WIDTH    :  integer := 8
    );
    port(
        i_sl_CLK      :  in  std_logic;                                -- clock
        i_sl_en       :  in  std_logic;                                -- enable
        i_sl_dir      :  in  std_logic;                                -- direction
        i_sl_rst      :  in  std_logic;                                -- reset
        o_slv_counter :  out std_logic_vector(DATA_WIDTH-1  downto 0)  -- counter
    );
end counter;
```

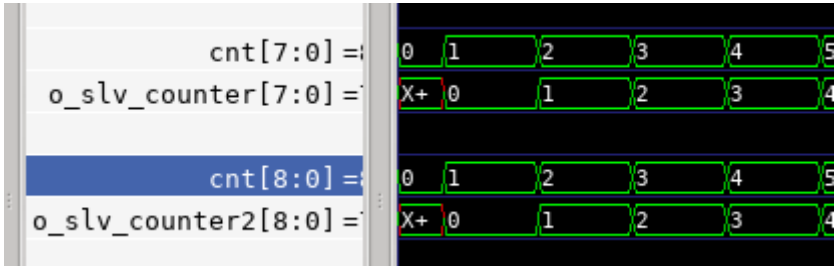
```vhdl
architecture behavior of counter is
    signal cnt : unsigned(DATA_WIDTH-1 downto 0) := (others => '0');  -- define signal, because we cannot readback outputs
begin

    process(i_sl_CLK)
    begin
        if rising_edge(i_sl_CLK) then
            if (i_sl_en = '1') then
                if( i_sl_rst = '1' ) then       -- option 1 ( reset when enabled )
                    cnt <= (others => '0');
                else                            -- we have to make sure, that we
                                                -- are not resetting and counting
                                                -- at the same time
                    if(i_sl_dir = '0') then
                        cnt <= cnt + 1;
                    else
                        cnt <= cnt - 1;
                    end if;
                end if;
            elsif ( i_sl_rst = '1') then        -- option 2 ( reset when disabled )
                cnt <= (others => '0');
            end if;
--          o_slv_counter <= std_logic_vector(cnt); -- option 1a  ( o_slv_counter is always 1 clock cycle late related to cnt )
        end if;
        o_slv_counter <= std_logic_vector(cnt);     -- option 1b  ( o_slv_counter is always 1 clock cycle late related to cnt )
    end process;

    -- o_slv_counter <= std_logic_vector(cnt);     -- option 2 ( o_slv_counter and cnt have the same value at the same clock cycle)

end behavior;
```

A short look, where to assign the outputs

Inside CLK process or outside

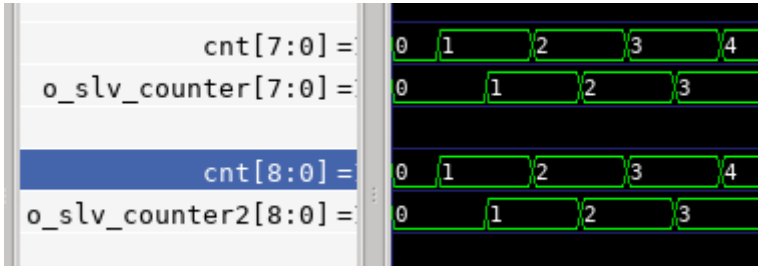
- Option 01 inside CLK process
  - Rising edge

Output delayed by 1 clock cycle



  - No rising egde

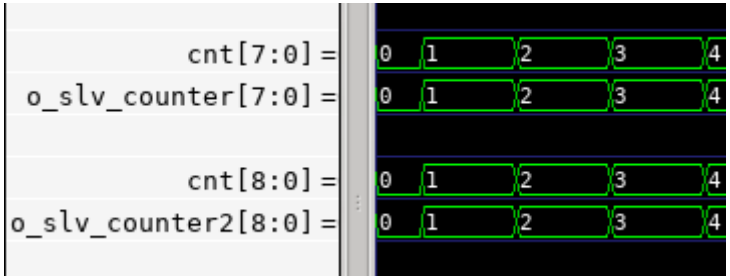
Output delayed by 0.5 clock cycle



Action on falling edge
– should be avoided

- Option 02 outside CLK process

Output delayed by 0 clock cycle



Addressdecoder

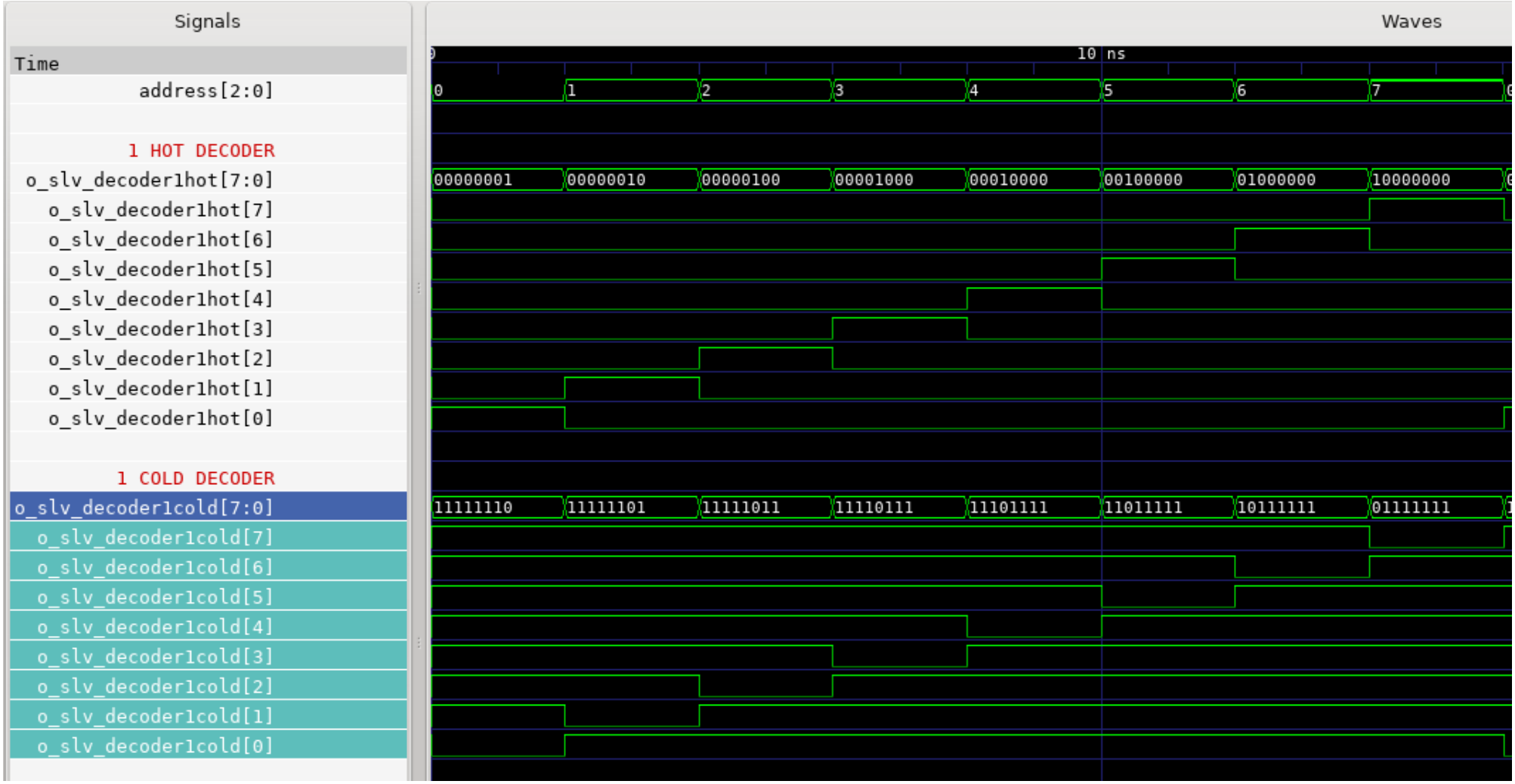
Lecture 04

III. Physikalisches Institut B
RWTH AACHEN UNIVERSITY

Addressdecoder

```
entity addressdecoder is
    generic(
        DATA_BITS     :  integer := 3;
        ONE_HOT       :  std_logic := '1'
    );
    port(
        i_sl_en       :  in  std_logic;                                -- enable
        i_slv_address :  in  std_logic_vector(DATA_BITS-1 downto 0);     -- address input
        o_slv_decoder :  out std_logic_vector( (2**DATA_BITS) -1  downto 0)   -- decoded output
    );
end addressdecoder;

----------------------------------------

architecture behavior of addressdecoder is
    signal adecoded : unsigned( (2**DATA_BITS)-1 downto 0) := (others => '0');  -- define signal, because we cannot readback outputs
begin

    process(i_slv_address, i_sl_en)
    begin
        if (i_sl_en = '1') then
            if ONE_HOT then
                adecoded <= (others => '0');
                adecoded( to_integer( unsigned(i_slv_address))  ) <= '1';
            else
                adecoded <= (others => '1');
                adecoded( to_integer( unsigned(i_slv_address))  ) <= '0';
            end if;
        end if;
    end process;

    o_slv_decoder <= std_logic_vector(adecoded);

end behavior;
```

Addressdecoder



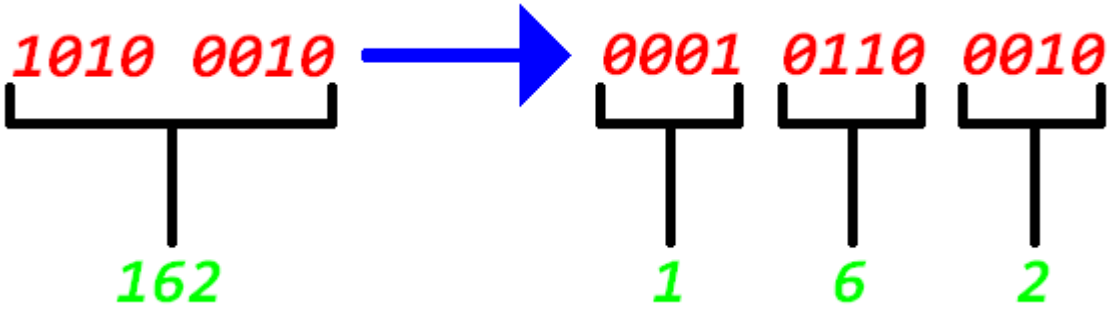
Binary Coded Decimal

Lecture 05

III. Physikalisches Institut B
RWTH AACHEN UNIVERSITY

BCD

- Example:



We would like to have this for 4 digits → Up to 9999

BCD Algorithm

1. If any column (100's, 10's, 1's) is 5 or greater add 3 to that column
2. Shift all #'s to the left 1 position
3. If **8** shifts are done, it's finished.
   Evaluate each column for the BCD values
4. Go to step 1.

BCD - Example

100's	10's	1's	Binary	Operation	
			1010 0010		162
		1	010 0010	<< #1	
		10	10 0010	<< #2	
		101	0 0010	<< #3	
		1000	0 0010	add 3	1 cycle
	1	0000	0010	<< #4	
	10	0000	010	<< #5	
	100	0000	10	<< #6	
	1000	0001	0	<< #7	
	1011	0001	0	add 3	1 cycle
1	0110	0010		<< #8	
1	6	2			

BCD decoder

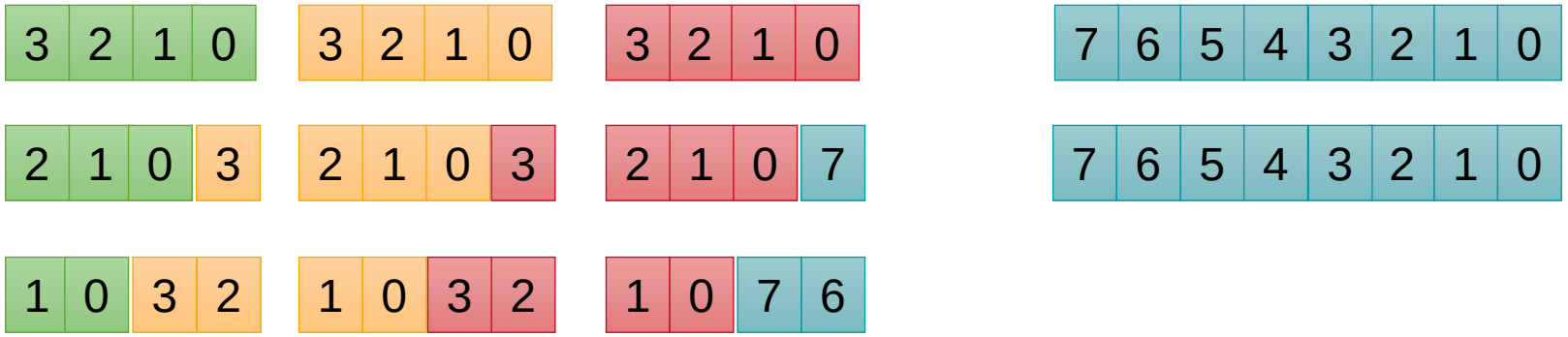
also possible with std_logic_vector
=> see moodle

```
architecture behavior of bcd is

begin
    process(i_slv_binary)
        -- temporarly variables
        variable huns : unsigned ( 3 downto 0) := (others => '0');
        variable tens : unsigned ( 3 downto 0) := (others => '0');
        variable ones : unsigned ( 3 downto 0) := (others => '0');
    begin
        -- reset all outputs
        huns := (others => '0');
        tens := (others => '0');
        ones := (others => '0');

        for i in 7 downto 0 loop

            if ( huns >= 5 ) then
                huns := huns + 3;
            end if;
            if ( tens >= 5 ) then
                tens := tens + 3;
            end if;
            if ( ones >= 5 ) then
                ones := ones + 3;
            end if;

            huns := huns(2 downto 0) & tens(3);
            tens := tens(2 downto 0) & ones(3);
            ones := ones(2 downto 0) & i_slv_binary(i);

            -- report "ones " & integer'image( to_integer(ones) );

        end loop;

        o_slv_decimal <=  std_logic_vector(huns) & std_logic_vector(tens) & std_logic_vector(ones);

    end process;
end behavior;
```

```
entity bcd is
--    GENERIC (
--    );
    port(
        i_slv_binary   :   in std_logic_vector( 7     downto 0);
        o_slv_decimal  :  out std_logic_vector( 3*4-1 downto 0)
    );
end bcd;
```

What is happening inside VHDL

```
huns := huns(2 downto 0) & tens(3);
tens := tens(2 downto 0) & ones(3);
ones := ones(2 downto 0) & i_slv_binary(i);
```



III. Physikalisches Institut B
RWTH AACHEN UNIVERSITY

Digital Filter

Digital Filter

Two kind of filters

FIR

- Finite Impulse Response
- Stable
- (no) feedback from the output
- Needs more computation effort
- Some fancy features
  - Linear phase relation
- A bit more complex to implement
  - Means in this context more difficult to implement a proper frequency

IIR

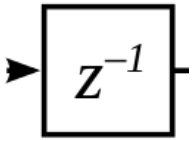
- Infinite Impulse Response
- Most efficient way of implementing filter
- Less computation needed
- Can be changed much easier on the fly
- Mostly implemented in DSP devices

Just some brief introduction / comparison!
Not very detailed.

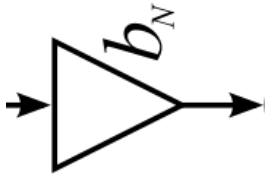
Contents of a Filter

Both are the same for FIR and IIR

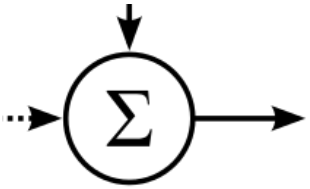
- We are not going to far into signal theory – we just use it
- What we will observe in all filters
- Delay by one clock cycle:



- Multiplication with a factor (here b_N)



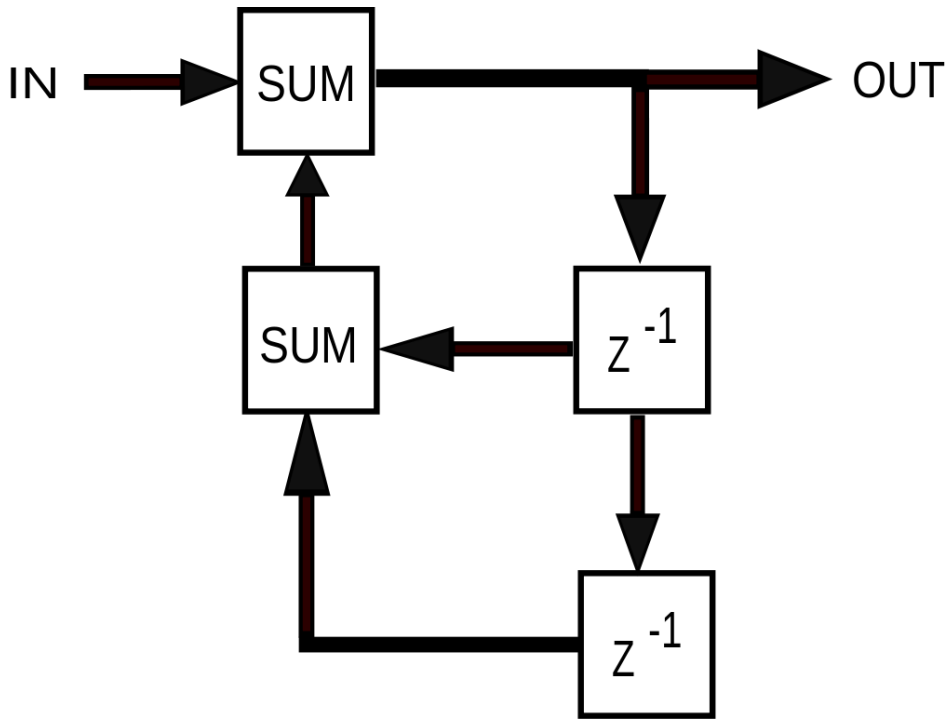
- Addition



IIR

Feedback from output

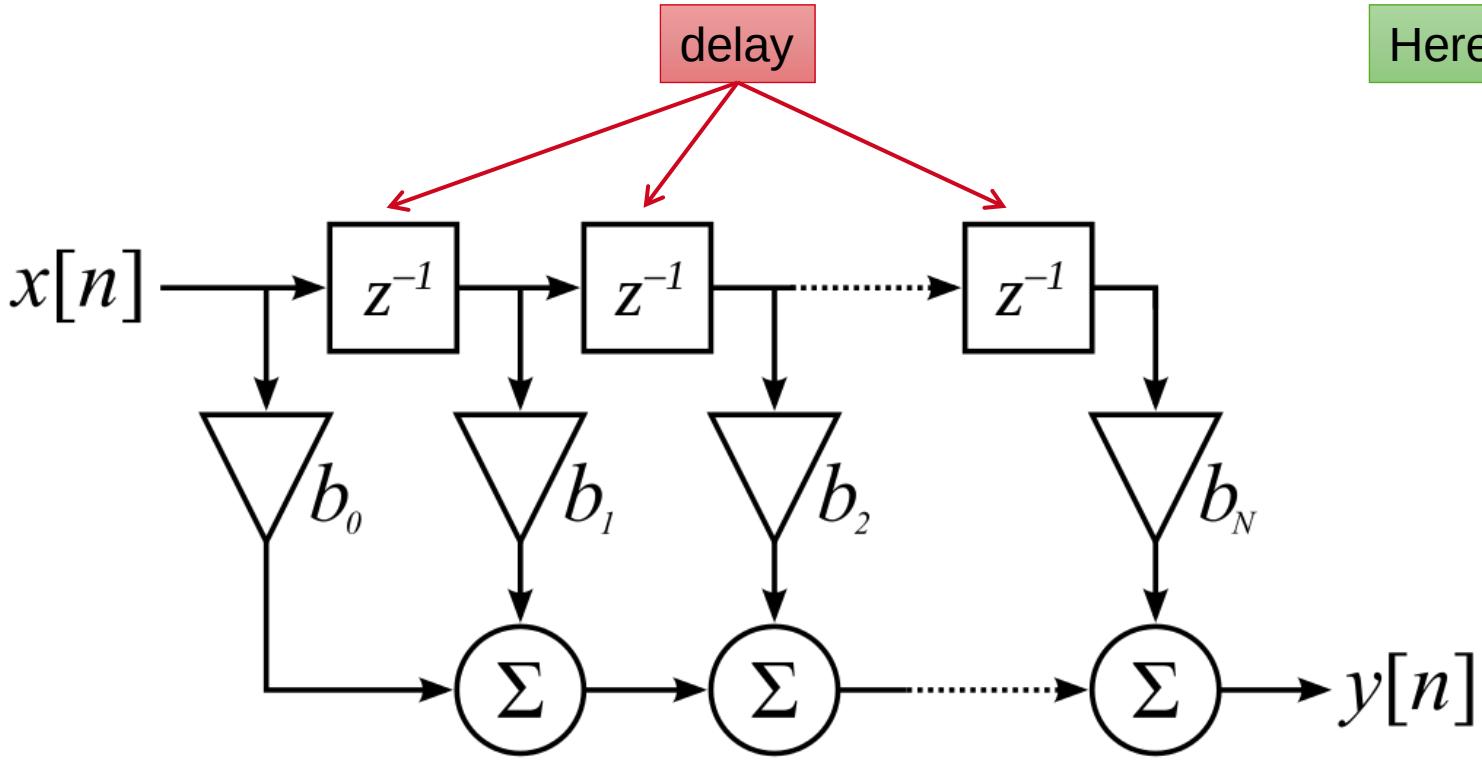
- This is also visible in the name (infinite impulse response)



Special kind of drawing

Order = Number of delays

Here 3rd order

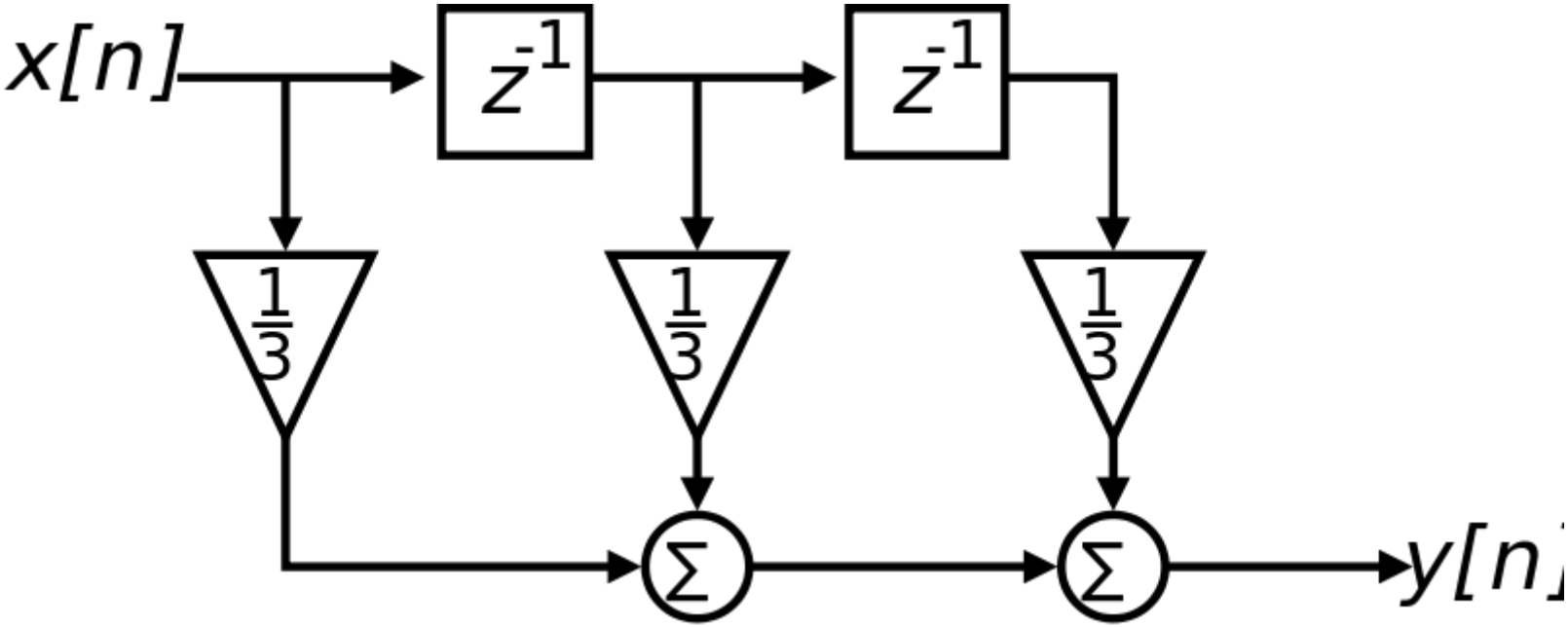


$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-N]$$
$$= \sum_{i=0}^{N} b_i \cdot x[n-i],$$

Having a look at a FIR filter

Very simple filter

- 2nd order Filter
- 3 taps



Moving Average: Y = 1/3 x[0] + 1/3 x[1] + 1/3 x[2]

$$H(z) = \frac{1}{3} + \frac{1}{3}z^{-1} + \frac{1}{3}z^{-2} = \frac{1}{3}\frac{z^2+z+1}{z^2}.$$

Problems when implementing filters

- The example on the previous slide uses float (1/3)
  - There is no float available in the FPGA
  - We have to implement it by using
    - unsigned
    - std_logic_vector

Which one would you choose?

- There is also another limiting factor
  - The division by 3 is difficult to implement (so we change to to 4)
  - Division by 4 can easily been done!

- **Why is it a better choice to calculate the average and divide afterwards?**
  - ¼ * X[0] + ¼ * X[-1] + ¼ * X[-2] + ¼ * X[-3]
  - (X[0] + X[-1] + X[-2] + X[-3]) * ¼

Mathematically the same!

Test the response of the Matched Filter

```
architecture behavior of MatchedFilter is
    signal shift_reg : slv8_array_t(3 downto 0) := (others=>(others=>'0'));

    signal coeff_reg : slv8_array_t(3 downto 0) := (
                                                    0 => (others => '1'),
                                                    1 => (others => '1'),
                                                    2 => (others => '1'),
                                                    3 => (others => '1')
                                                    );
begin

    process(i_sl_CLK)
        -- 8 bit x 8 bit = 16 bit
        -- 3 bit (4) x 16 bit = 19 bit
        variable average : unsigned(18 downto 0) := (others => '0');
    begin
        if rising_edge(i_sl_CLK) then
            if (i_sl_en = '1') then

                -- do shifting of all input values
                for I in 3 downto 1 loop
                    shift_reg(I) <= shift_reg( I - 1);
                end loop;
                shift_reg(0) <= i_slv_data;

                -- reset avarage
                average := (others => '0');
                for I in 3 downto 0 loop
                    average := average + unsigned( shift_reg(I) ) * unsigned( coeff_reg(I) );
                end loop;

            end if;
        end if;

        o_slv_average <= std_logic_vector(average);

    end process;

end behavior;
```

one loop for the shift register

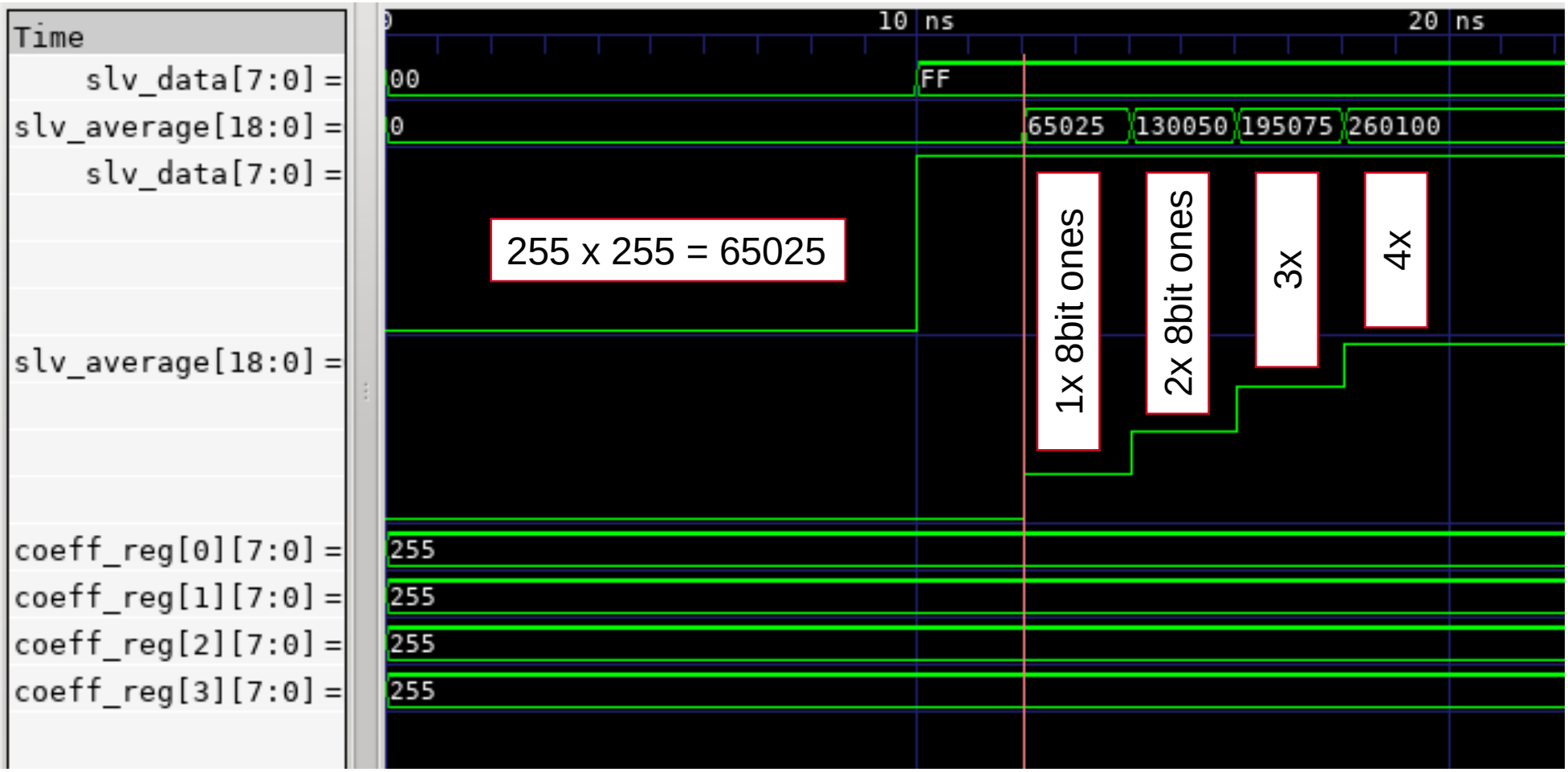
one loop for the FIR filter

Why 19 bits?

GHDL is assuming, that all variables can have their maximal value

Bit 18 will never been set to 1 in our application.

Test the response of the Matched Filter



Ring Buffer

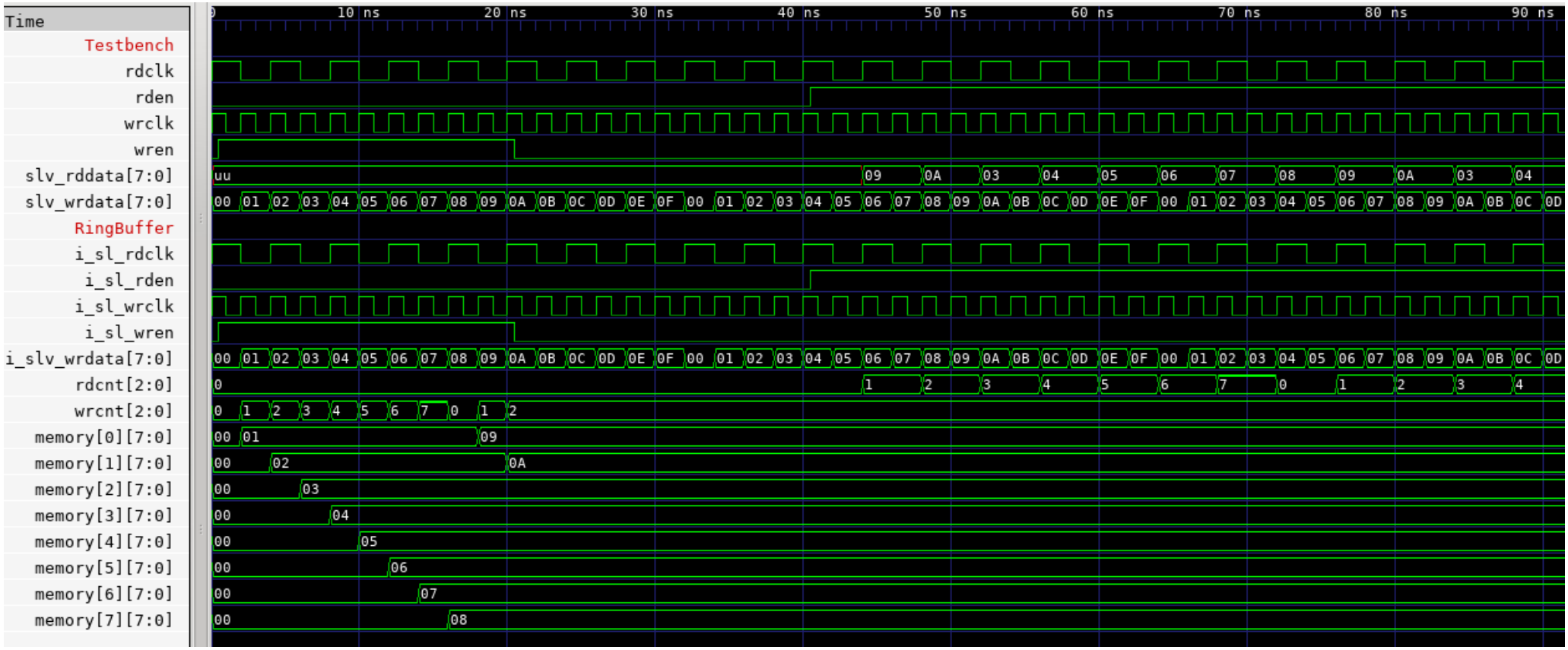
Lecture 09

III. Physikalisches Institut B
RWTH AACHEN UNIVERSITY

Lecture 09a

```
entity RingBuffer is
    port(
    -- write
        i_sl_wrCLK      :  in  std_logic;                      -- clock
        i_sl_wrEN       :  in  std_logic;                      -- enable
        i_slv_wrdata    :  in  std_logic_vector(7 downto 0);   -- data input

    -- read
        i_sl_rdCLK      :  in  std_logic;                      -- clocke
        i_sl_rdEN       :  in  std_logic;                      -- enable
        o_slv_rddata    :  out std_logic_vector(7 downto 0)    -- data input
    );
end RingBuffer;

---------------------------------------

architecture behavior of RingBuffer is
    signal memory : slv8_array_t(7 downto 0) := (others=>(others=>'0'));

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

begin

    process(i_sl_wrCLK)
    begin
        if rising_edge(i_sl_wrCLK) then
            if (i_sl_wrEN = '1') then
                memory( to_integer(wrcnt) ) <= i_slv_wrdata;

                wrcnt <= wrcnt + '1';
            end if;
        end if;
    end process;

    process(i_sl_rdCLK)
    begin
        if rising_edge(i_sl_rdCLK) then
            if (i_sl_rdEN = '1') then
                o_slv_rddata <= memory( to_integer(rdcnt)) ;

                rdcnt <= rdcnt + '1';
            end if;
        end if;
    end process;

end behavior;
```

Simulation output



FiFo

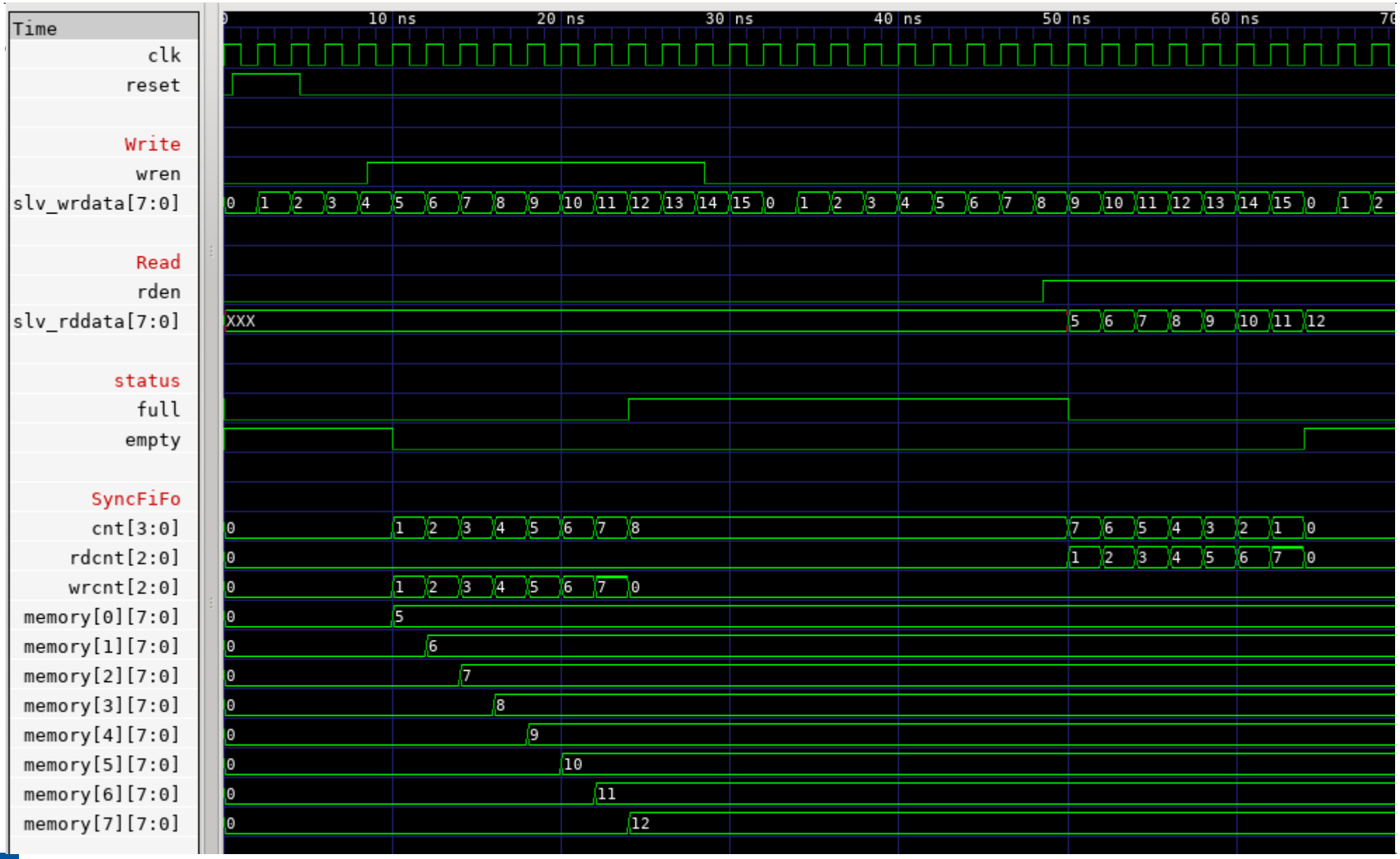
Lecture 11

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)



Finite State Machines

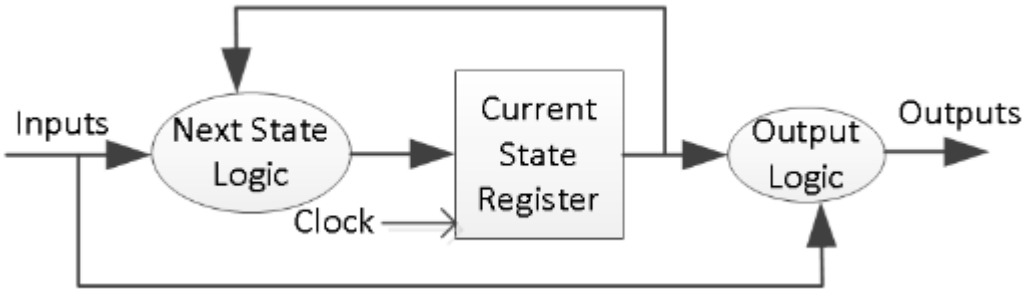
Lecture 11 + Lecture 12

III. Physikalisches Institut B
RWTH AACHEN UNIVERSITY

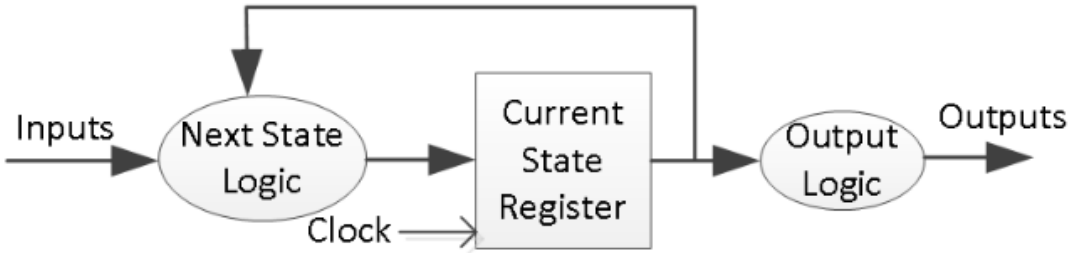
Three blocks State machines

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;
```

Thank you!

If you are interested in doing FPGA development in a thesis:
Have a look at
https://www.institut3b.physik.rwth-aachen.de/cms/ParticlePhysics3B/Forschung/LENA-und-JUNO/~geio/Bachelor-und-Masterarbeiten/