FPGA tutorial

Lecture 8

16.12.2020

Jochen Steinmann

III. Physikalisches Institut B

RWTH AACHEN UNIVERSITY

Solution

Last time

A. Implementing the moving average FIR filter (N = 3)

B. Implement different filter coefficients (b‘s)
   - B0 = 1
   - B1 = 2
   - B2 = 3
   - B3 = 4

- Plot the step response for both of the filters (gtkwave can do this)

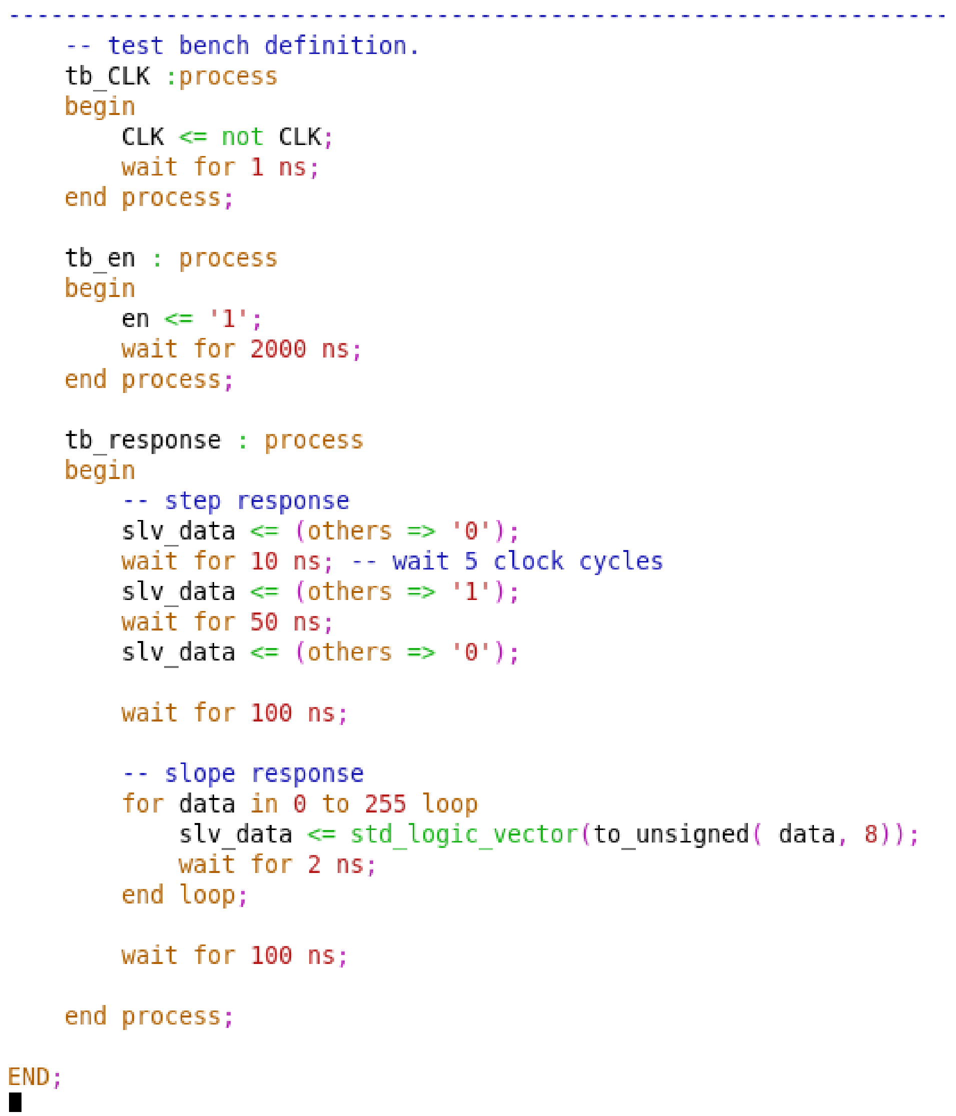
Keep in mind: Write your code, that it can be reused (generic).

Lecture 07 a

Module

```
entity MovingAverage is
    port(
        i_sl_CLK      :  in  std_logic;                          -- clock
        i_sl_en       :  in  std_logic;                          -- enable
        i_slv_data    :  in  std_logic_vector(7 downto 0);       -- data input
        o_slv_average :  out std_logic_vector(7 downto 0)        -- moving average output
    );
end MovingAverage;

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

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

begin

    process(i_sl_CLK)
        variable average : unsigned(9 downto 0) := (others => '0');
    begin
        if rising_edge(i_sl_CLK) then
            if (i_sl_en = '1') then
                -- reset avarage
                average := (others => '0');

                -- do shifting of all input values
                -- at the same time sum up all entries
                for I in 3 downto 1 loop
                    shift_reg(I) <= shift_reg( I - 1);
                    average := average + unsigned(shift_reg(I));
                end loop;

                -- in the loop, we are missing element 0
                shift_reg(0) <= i_slv_data;
                average := average + unsigned( shift_reg(0) );

            end if;
        end if;

        o_slv_average <= std_logic_vector( average(9 downto 2) );   -- divide by 4

    end process;

end behavior;
```

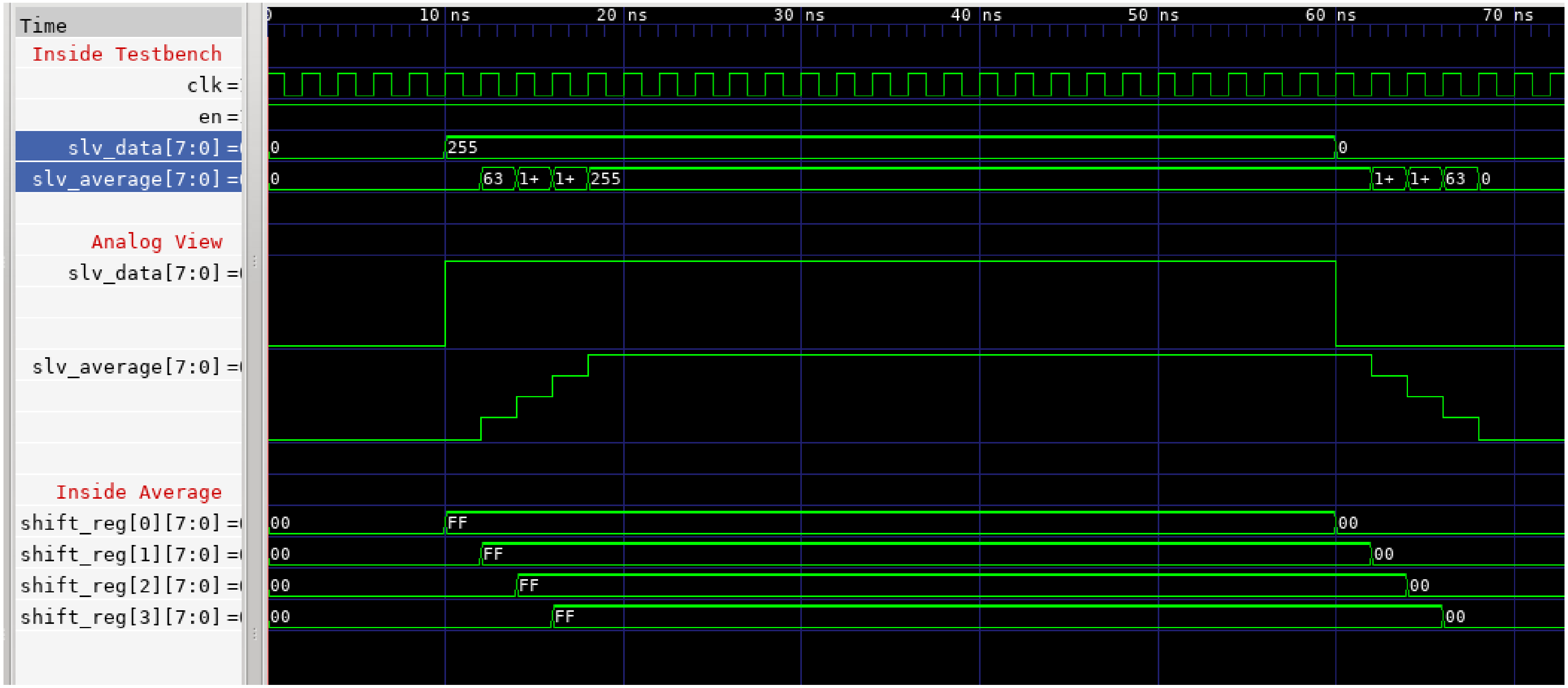
Testbench



```
-----------------------------------------------------------------
    -- test bench definition.
    tb_CLK :process
    begin
        CLK <= not CLK;
        wait for 1 ns;
    end process;

    tb_en : process
    begin
        en <= '1';
        wait for 2000 ns;
    end process;

    tb_response : process
    begin
        -- step response
        slv_data <= (others => '0');
        wait for 10 ns; -- wait 5 clock cycles
        slv_data <= (others => '1');
        wait for 50 ns;
        slv_data <= (others => '0');

        wait for 100 ns;

        -- slope response
        for data in 0 to 255 loop
            slv_data <= std_logic_vector(to_unsigned( data, 8));
            wait for 2 ns;
        end loop;

        wait for 100 ns;

    end process;

END;
```

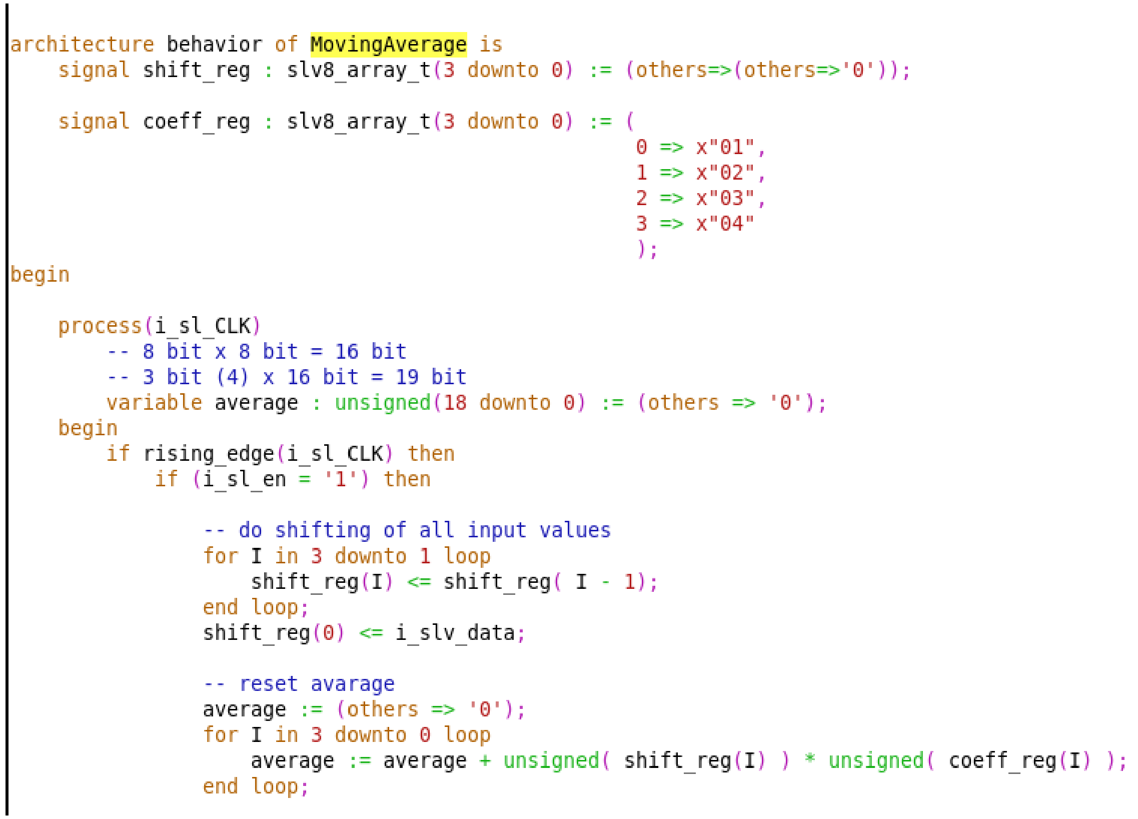
Lecture 7 a

Simulation output

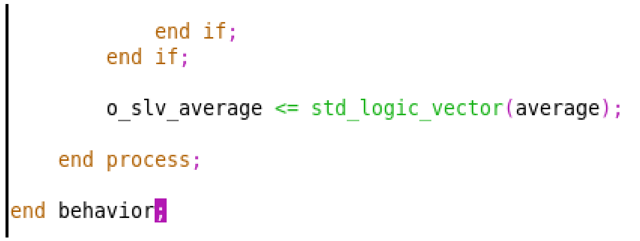


Lecture 7 b

Module

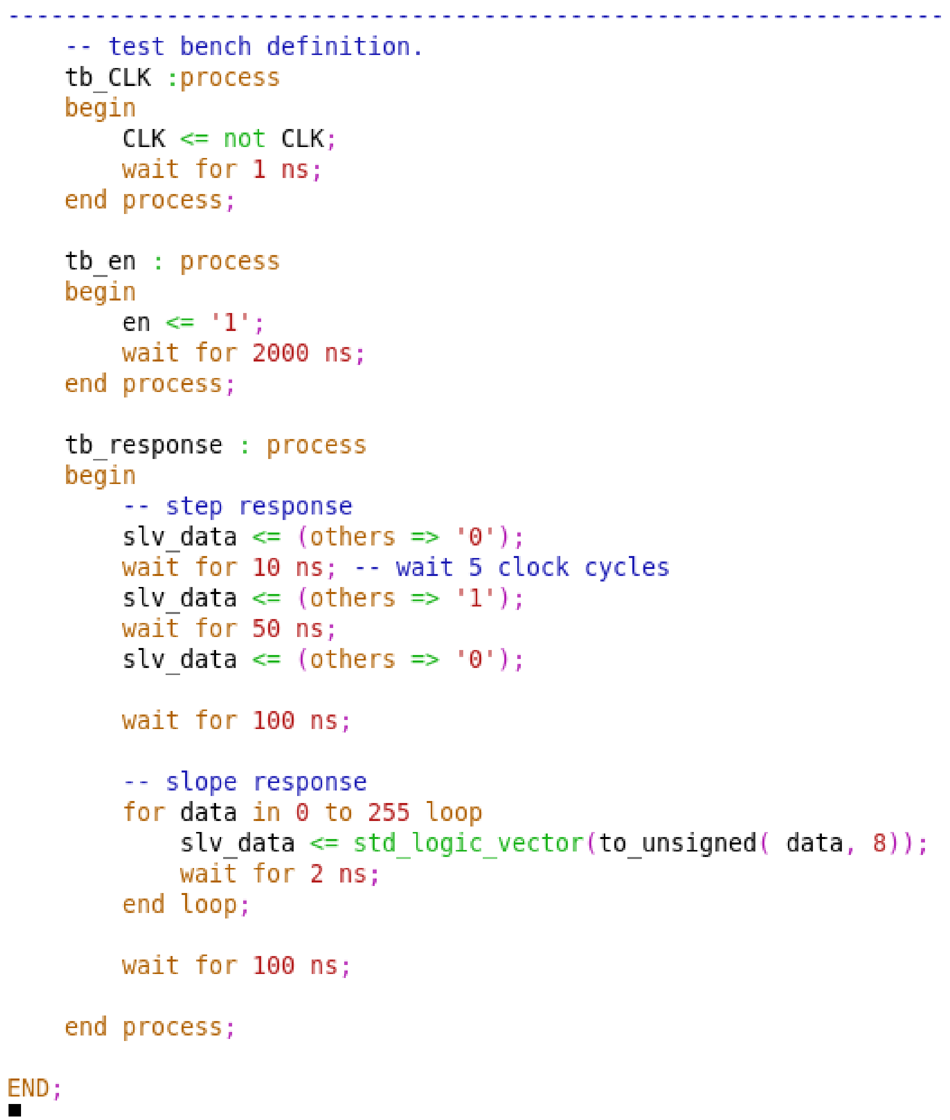


```
architecture behavior of MovingAverage is
    signal shift_reg : slv8_array_t(3 downto 0) := (others=>(others=>'0'));

    signal coeff_reg : slv8_array_t(3 downto 0) := (
                                                    0 => x"01",
                                                    1 => x"02",
                                                    2 => x"03",
                                                    3 => x"04"
                                                    );
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;
```

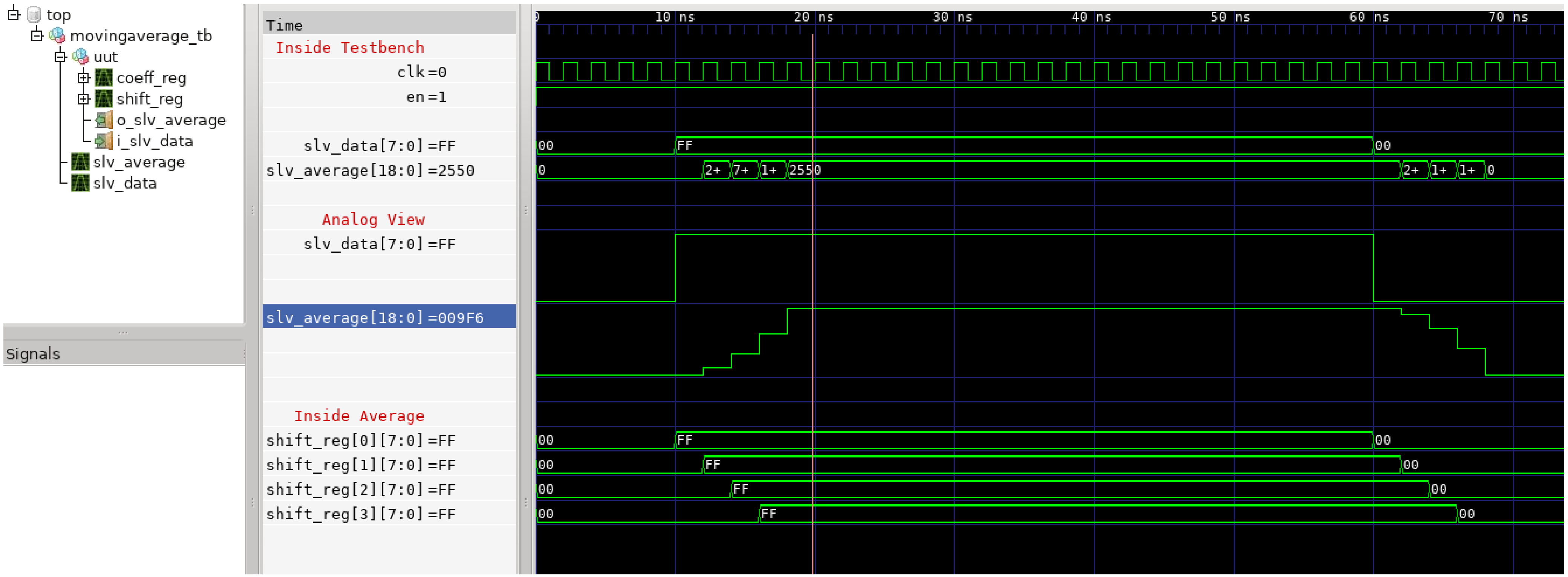
Testbench



```
-----------------------------------------------------------------
    -- test bench definition.
    tb_CLK :process
    begin
        CLK <= not CLK;
        wait for 1 ns;
    end process;

    tb_en : process
    begin
        en <= '1';
        wait for 2000 ns;
    end process;

    tb_response : process
    begin
        -- step response
        slv_data <= (others => '0');
        wait for 10 ns; -- wait 5 clock cycles
        slv_data <= (others => '1');
        wait for 50 ns;
        slv_data <= (others => '0');

        wait for 100 ns;

        -- slope response
        for data in 0 to 255 loop
            slv_data <= std_logic_vector(to_unsigned( data, 8));
            wait for 2 ns;
        end loop;

        wait for 100 ns;

    end process;

END;
```

Lecture 7 b

Simulation



NEW

VHDL

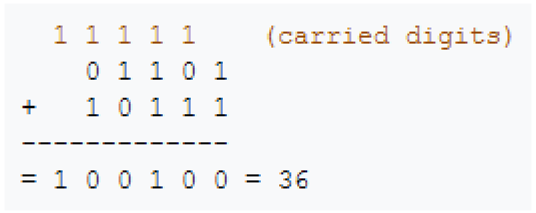
III. Physikalisches Institut B

RWTH AACHEN UNIVERSITY

Why only binary multiplication?

Binary mathematics

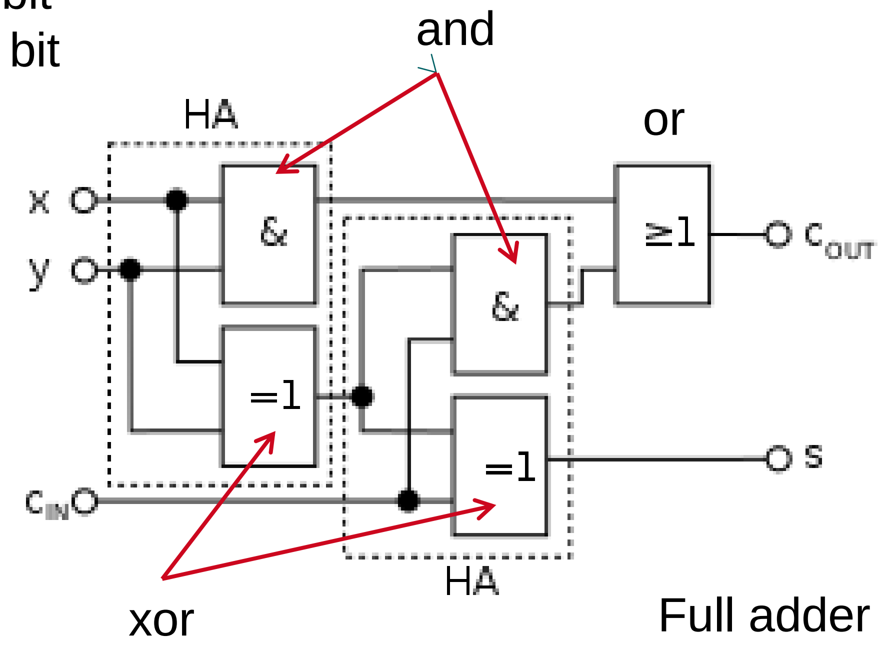
Addition / Subtraction



```
  1 1 1 1 1    (carried digits)
    0 1 1 0 1
+   1 0 1 1 1
-------------
= 1 0 0 1 0 0 = 36
```

Subtraction:
A − B = A + not B + 1

Full adder: can add 3x 1 bit
Half adder: can add 3x 1 bit



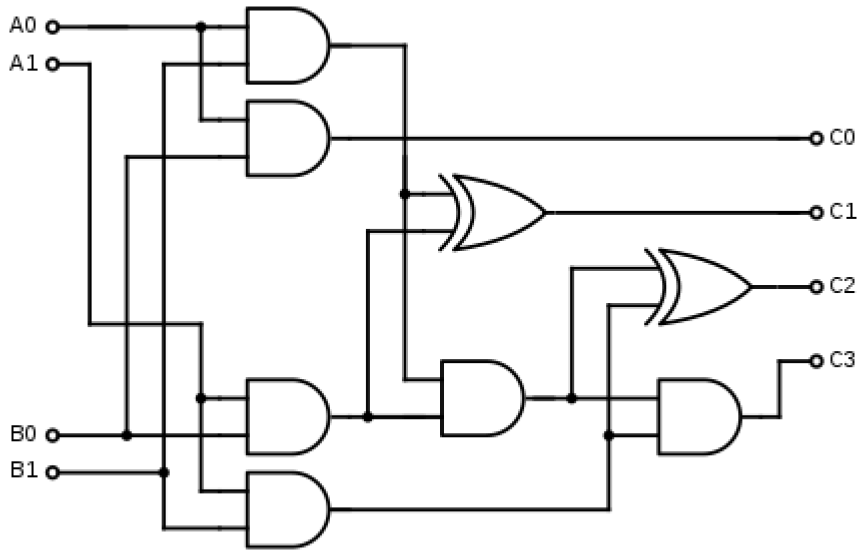
Full adder

Binary multiplication

Can be paralized

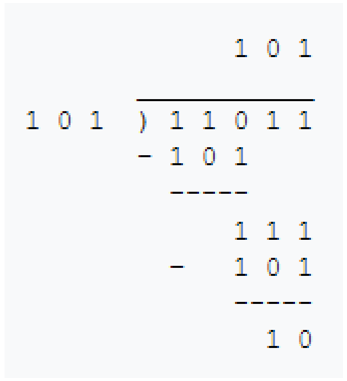
```
        1 0 1 1    (A)
      × 1 0 1 0    (B)
      ---------
        0 0 0 0    ← Corresponds to the rightmost 'zero' in B
+     1 0 1 1      ← Corresponds to the next 'one' in B
+   0 0 0 0
+ 1 0 1 1
---------------
= 1 1 0 1 1 1 0
```

Multiplication is just shifting and adding (if there is a one)



Binary division

Not that simple



```
            1 0 1
        __________
1 0 1   ) 1 1 0 1 1
        - 1 0 1
          -----
              1 1 1
          -   1 0 1
              -----
                1 0
```

- We have to subtract depending on the input value
  - This needs some clock cycles …

Digital Signal Processing

DSP

III. Physikalisches Institut B

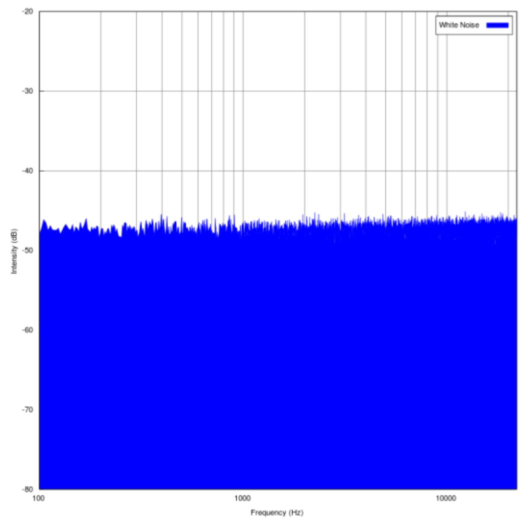
RWTH AACHEN UNIVERSITY

Noise

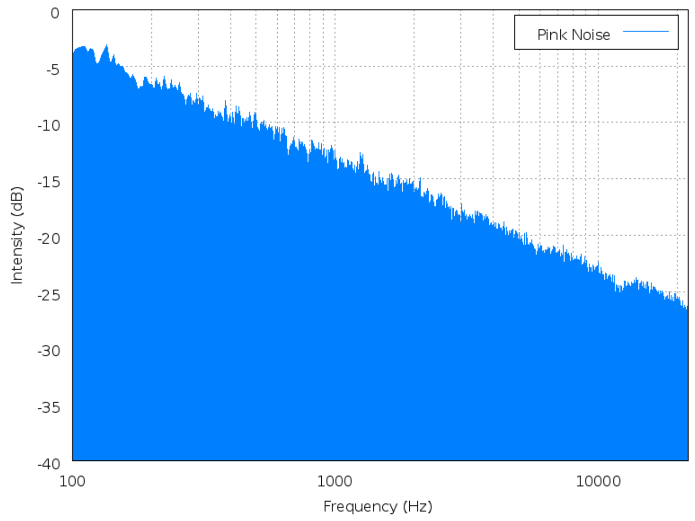
Noise ≠ Noise

Two main noise contributions

1. White noise
   - Constant frequency contribution
   - Thermal noise
   - Noise of resistors



2. Pink noise
   1. 1/f frequency dependency
   2. Some components in electronics cause this kind of noise FET, transistors



Matched Filter

How to find signal in noise …

Just scratching at the very top of the surface!

III. Physikalisches Institut B

RWTH AACHEN UNIVERSITY

Situation

Something from the real world

- **Simulation / Theory**
  - We have a signal.

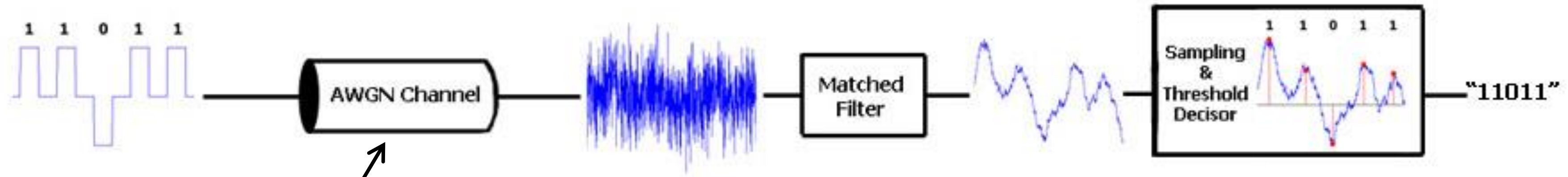
- **Reality**
  - we have a signal
  - and there is noise

- Sometimes the signal to noise ratio is rather low
  - Signal might just be a tiny bit above the noise

- We would like to detect these signals

Why do we have to care about noise and signals

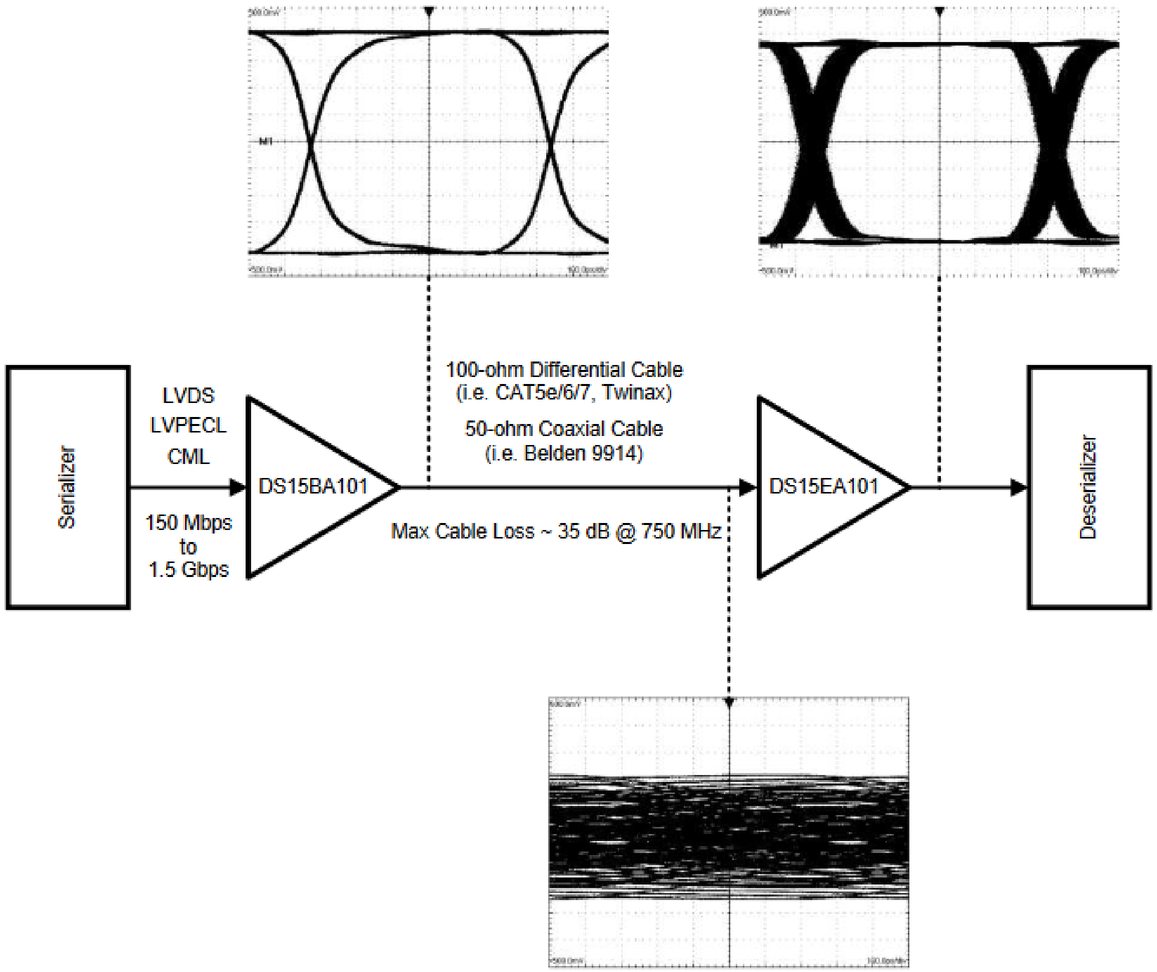
Noise is everywhere …



Additional white Gaussian noise

Why do we have to care about noise and signals

Transmission over large distances (50 to 100m)



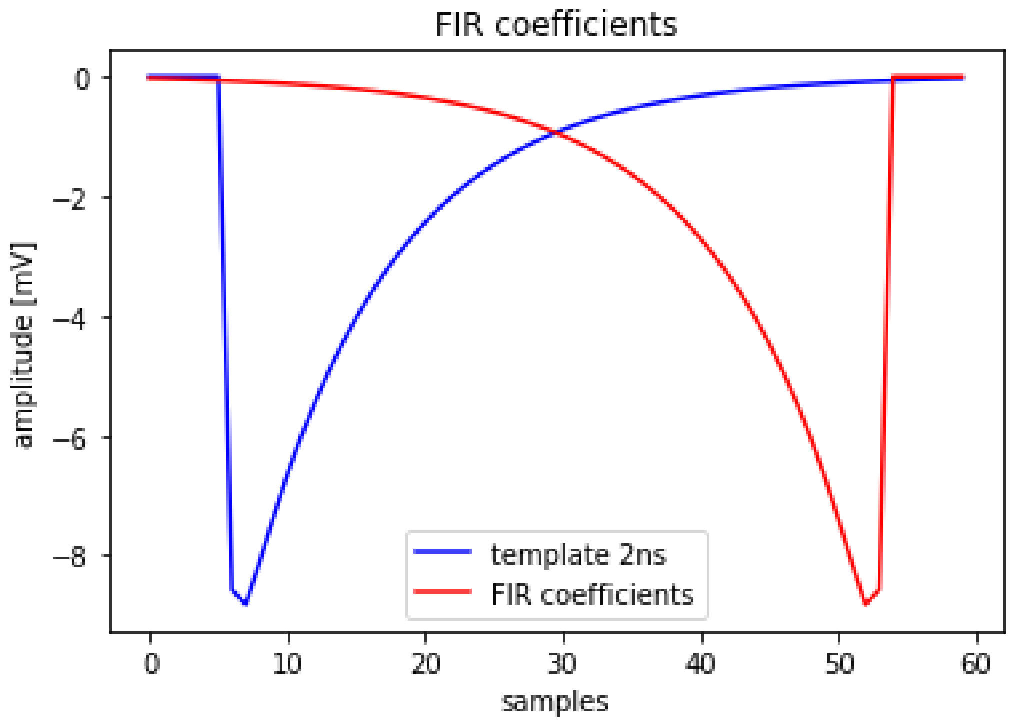
How is a matched filter implemented?

1. Create a template of your signal
2. Inverse the template in time
3. Use the inverse template as the coefficients for an FIR filter

4. Multiply a window of the signal with your template
   - Window length has to match the template length

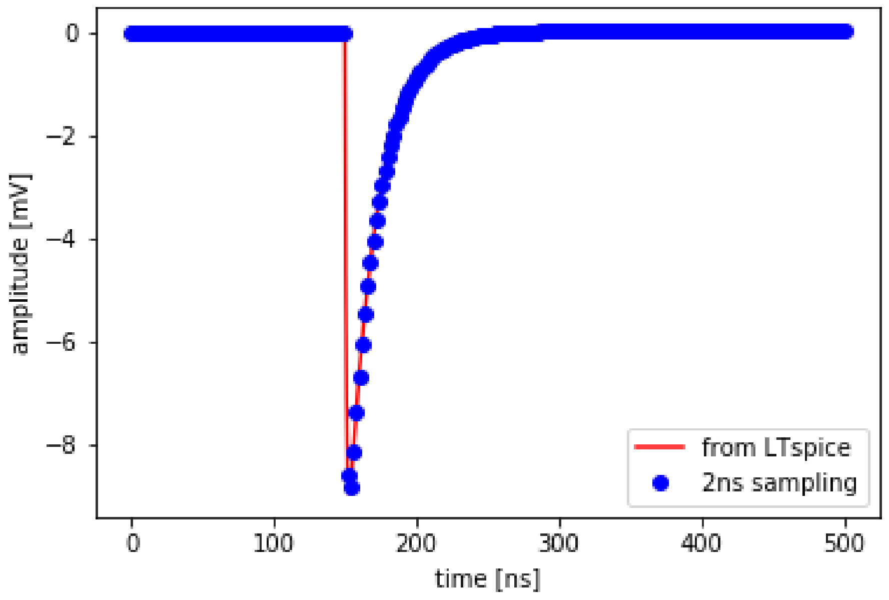
   - This is exactly what the FIR filter is doing

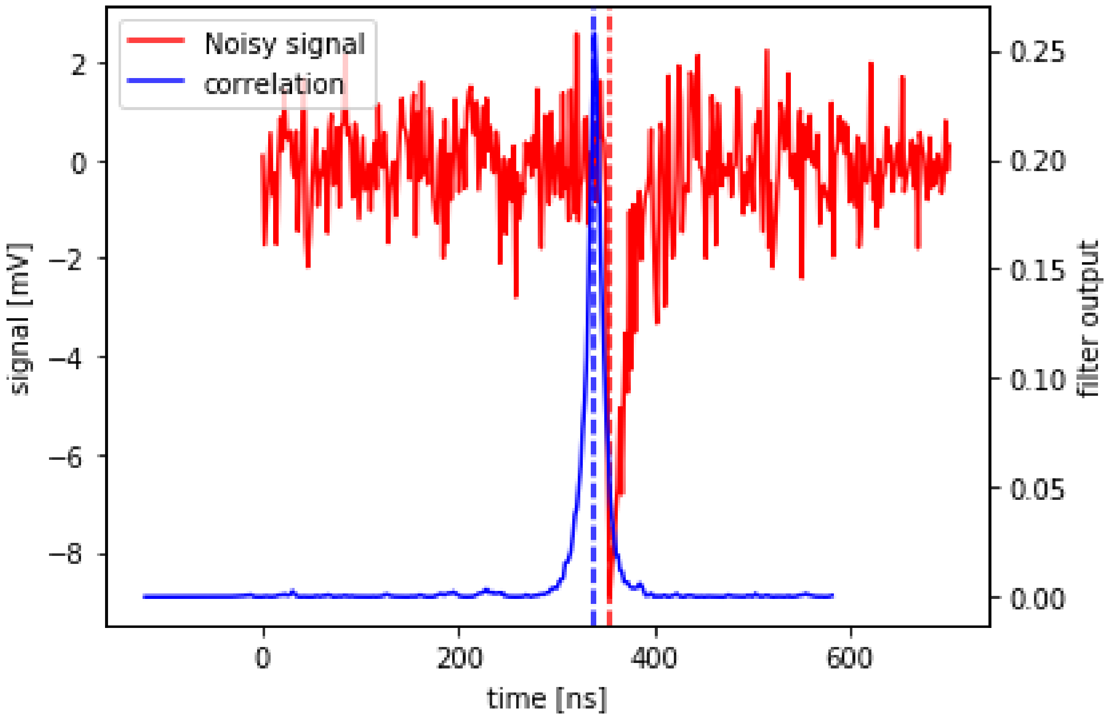
   - Zeroes in the template are useless
     - They are increasing the order of the FIR, without having an impact



Having a look at real data

Creating a template for our signal

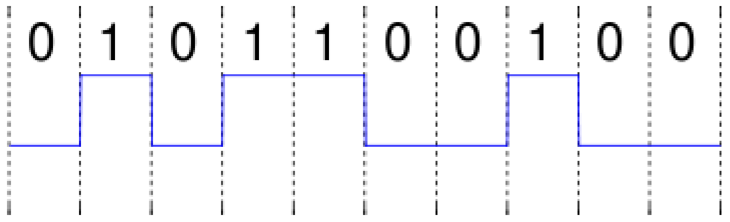




III. Physikalisches Institut B

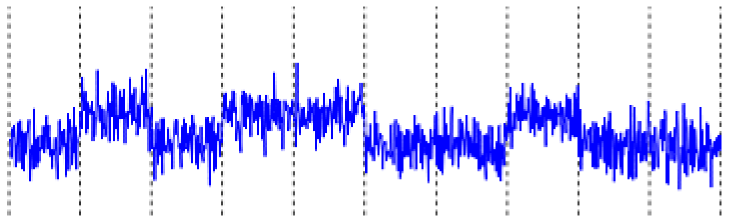
RWTH AACHEN UNIVERSITY

Example 1



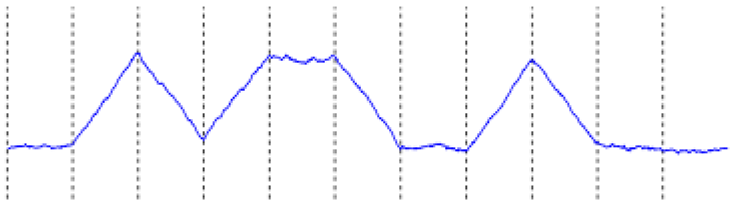
Signal at FPGA output

transmitter

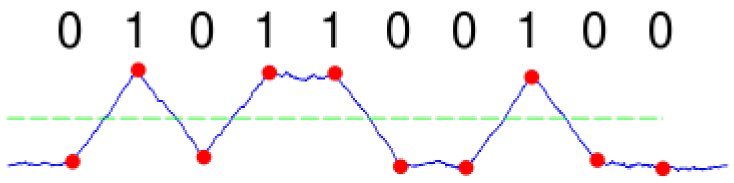


Signal ADC input

receiver



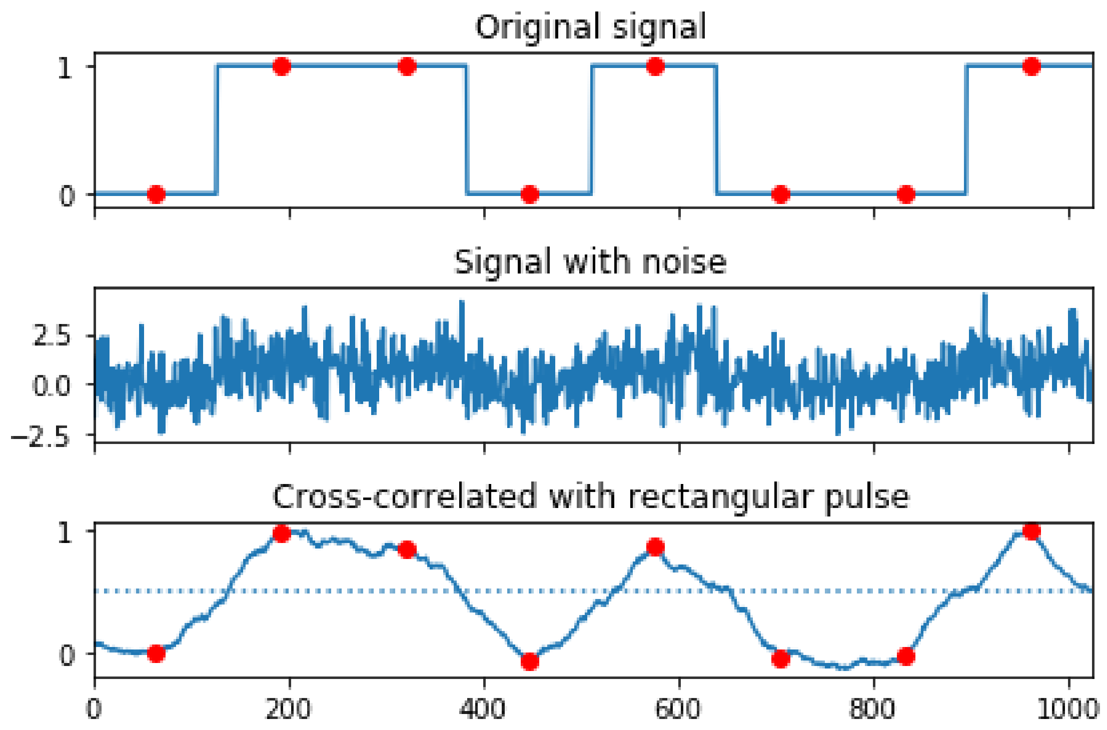
Signal at Matched Filter output



Analyzed signal

Simple Matched Filter

Example 2



It is really hard to observe a signal by eye

Template: 128 Sample ones

Implemented using floats (which we cannot do on an FPGA)

Where does the FPGA comes into the game?

Real-time…

- Using offline software, this is rather easy
  - You can use Python / C++

- But let us assume, we have 80 PMTs (Photomultiplier) all giving each about 20kHz of signal
  - You need a huge computing farm in order to apply a Matched Filter to all of these signals.
  - 20 kHz = one signal every 50 µs

- If we implement a Matched Filter in Hardware (FPGA) we can have a real-time output of the filter

ToDo for today

Template is the solution from last week

- A
  - Implement a matched filter for
    - 4 Samples of 8 bit ones
  - Test the result of this filter using different patterns
    - Response to
      - 1x 8bit
      - 2x 8bit
      - 3x 8bit
      - 4x 8bit
- B
  - Modify the matched filter, that you can change the coefficients from the testbench
  - **Do not change the array at once!**
  - Use:
    - One signal for enabling the „programming mode“
    - 3 bits for the address of the array
    - single 8 bit for the content of the array

    - Keep in mind: **Data should be transferred at the rising edge of the clock**

Merry Christmas

stay healthy