# FPGA tutorial

## Lecture 10

**13.01.2021**

**Jochen Steinmann**

III. Physikalisches Institut B

RWTH AACHEN UNIVERSITY

# Organisatorical


Solution

# ToDo:

A. Create a RingBuffer, which can
   - store 8x 8bit values
   - Two different clocks (read and write) each with its own enable

B. Change the RingBuffer that it provides these features:
   - Readout only, if write disabled
   - Output in the correct order (read pointer has to start from write pointer)
     Hint: Between read and write are some clock cycles of each clock.

   Keep in mind:
   It is not possible to set variables in different processes.

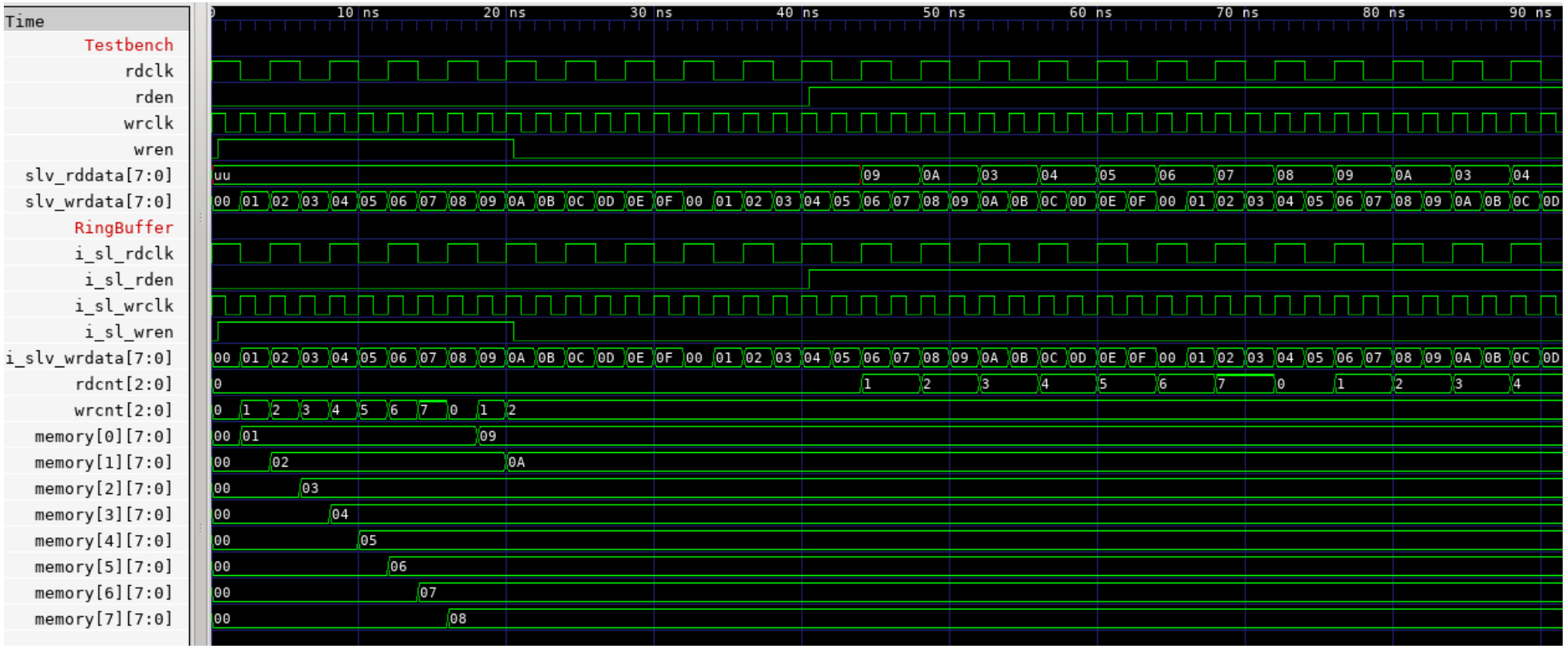
C. Start with a simple trigger module
   - If reset, run = 1
   - If data is for one clockcycle above threshold run = 0 until reset

# 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;
```


Jochen Steinmann | RWTH Aachen University

# Simulation output



# Keep the order of the data

```
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';
            else
                rdcnt <= wrcnt;
            end if;
        end if;
    end process;

end behavior;
```

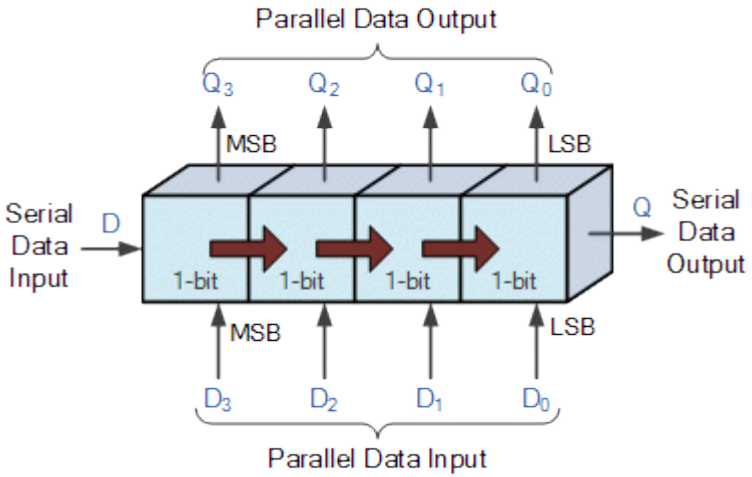
Here we are setting the read counter to the write counter. This needs one clock cycle between writing and reading

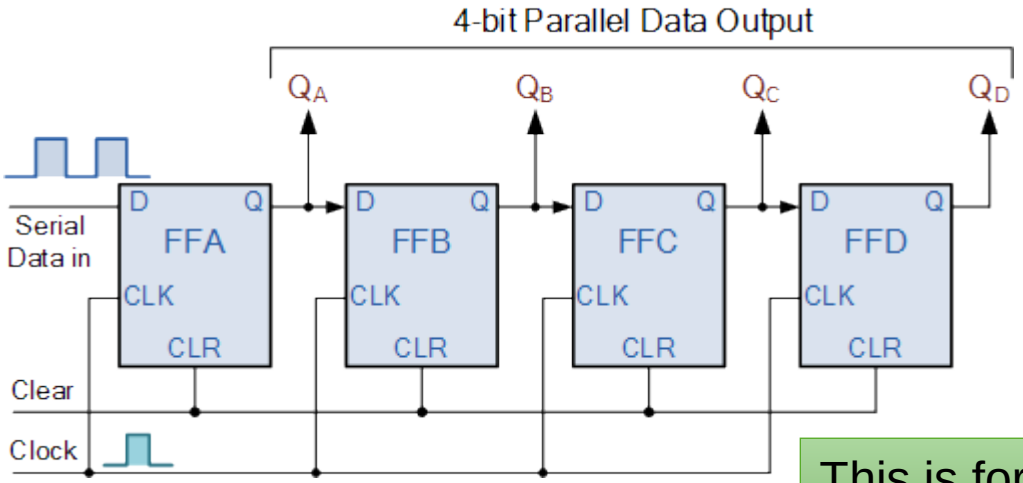

Jochen Steinmann | RWTH Aachen University

# Simulation output

# Why using a read / write pointer

- The ringbuffer has the same functionality as our wide shift register
  - So where is the difference?
  - Assumption: we want to write 8 bits (data size)
  - Shift register looks like



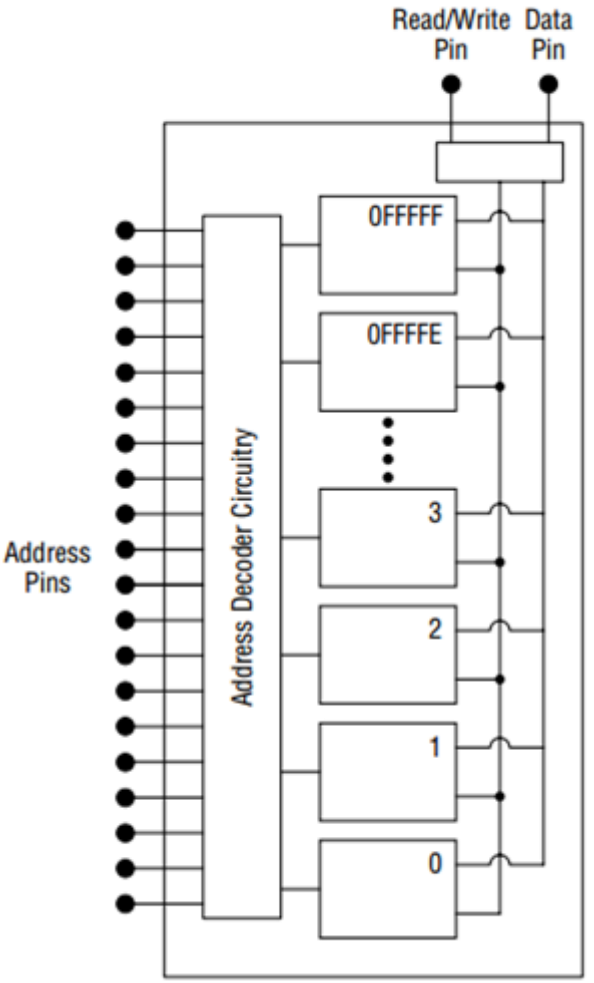




This is for a single bit

  - **If we store 8 x 8 bit, we have to update 64 bits in one clock cycle**

# Why using a read / write pointer II

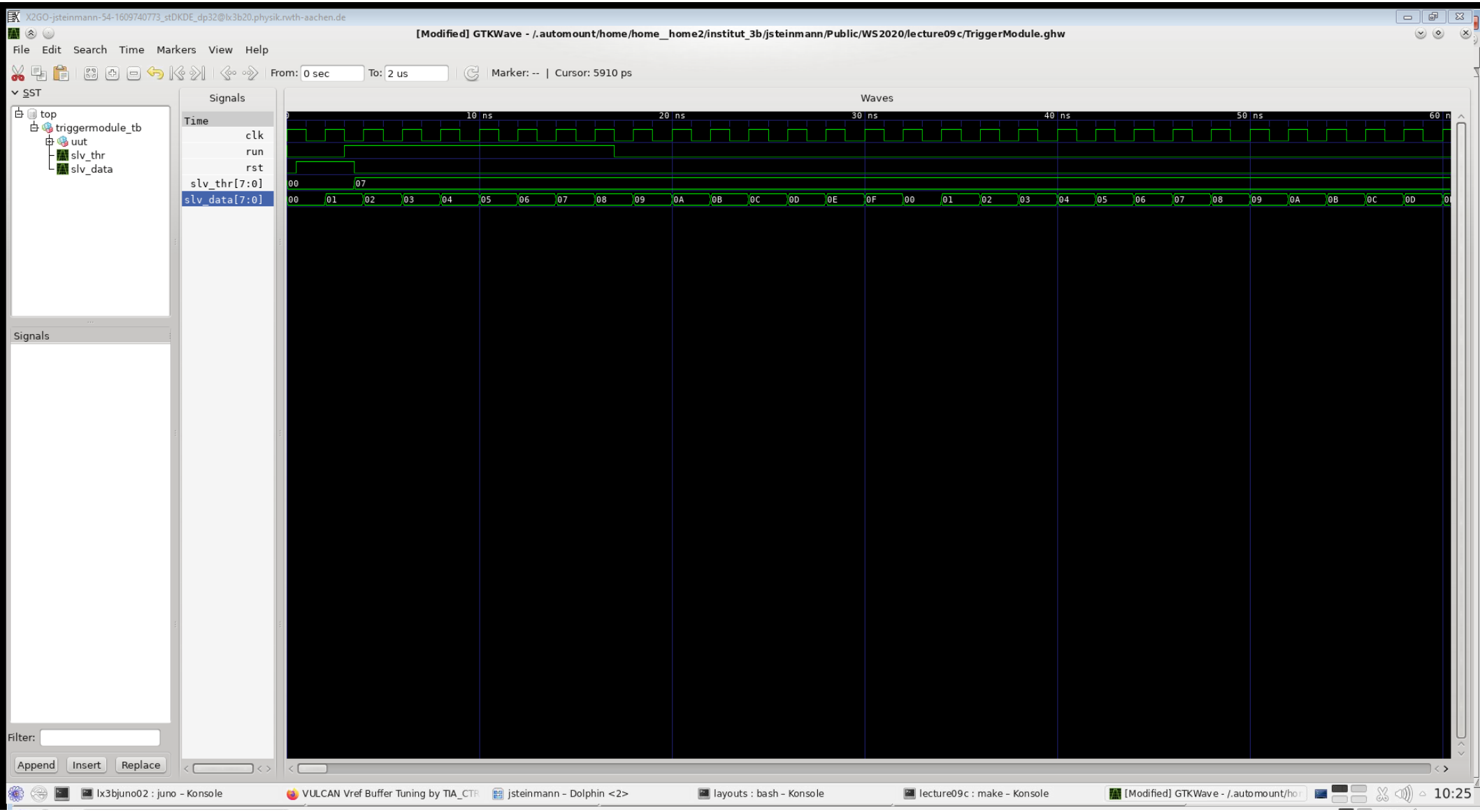
- By using a write pointer, we can use different internal structures (e.g. RAM)
- Since we are just updating one 8 bit „data cell“, it‘s easier for the FPGA to write the data, less bit changes per clock cycle.



# Trigger module

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

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

entity TriggerModule is
    port(
        i_sl_CLK        :  in  std_logic;                       -- clock
        i_sl_rst        :  in  std_logic;                       -- reset
        i_slv_data      :  in  std_logic_vector(7 downto 0);    -- data input
        i_slv_thr       :  in  std_logic_vector(7 downto 0);    -- threshold input
        o_sl_run        :  out  std_logic                       -- run output
    );
end TriggerModule;

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

architecture behavior of TriggerModule is
    signal run : std_logic := '0';
begin
    process(i_sl_CLK)
    begin
        if rising_edge(i_sl_CLK) then
            if (i_sl_rst = '1') then
                run <= '1';
            elsif ( unsigned(i_slv_data) > unsigned(i_slv_thr) ) then
                run <= '0';
            end if;
        end if;
        o_sl_run <= run;
    end process;
end behavior;
```

# Lecture 09c


[Modified] GTKWave - /.automount/home/home__home2/institut_3b/jsteinmann/Public/WS2020/lecture09c/TriggerModule.ghw
File Edit Search Time Markers View Help
From: 0 sec
To: 2 us
Marker: -- | Cursor: 5910 ps
SST
top
triggermodule_tb
uut
slv_thr
slv_data
Signals
Filter:
Append Insert Replace
Signals
Time
clk
run
rst
slv_thr[7:0]
slv_data[7:0]
Waves
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D
lx3bjuno02 : juno – Konsole
VULCAN Vref Buffer Tuning by TIA_CTF
jsteinmann – Dolphin <2>
layouts : bash – Konsole
lecture09c : make – Konsole
[Modified] GTKWave - /.automount/ho
10:25


NEW

# VHDL

III. Physikalisches Institut B
RWTH AACHEN UNIVERSITY

# When / Else statement

**a**

- Sometimes, it is required to set a signal independent of a clock

- e.g.
  - Set signal / output empty to ‚1‘ if cnt is zero, set it to zero otherwise

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

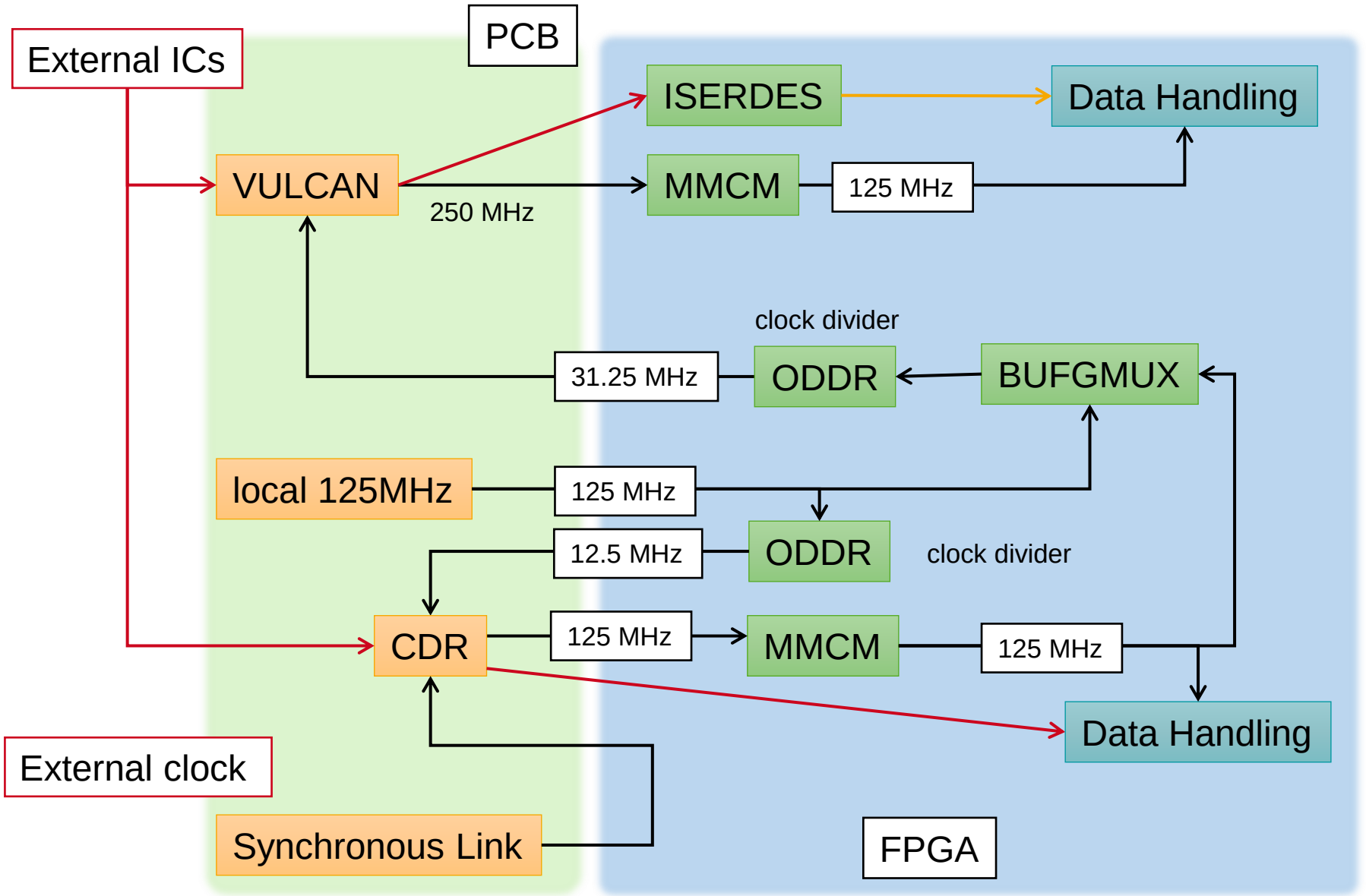
# Storing data and analysing it later

# Clocks within an FPGA

## Many different clocks

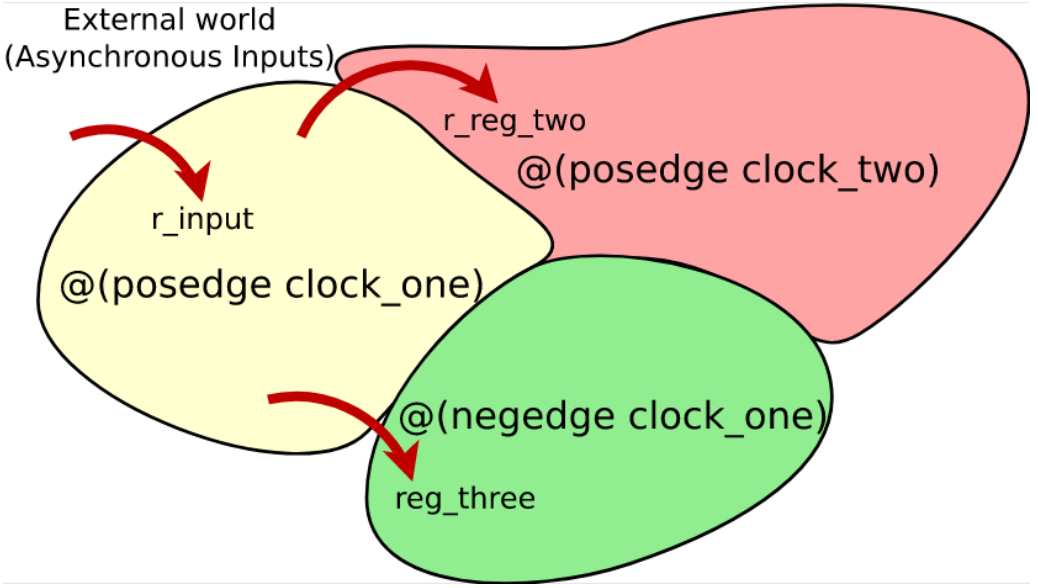
- External signals
- Clocks & data from ADCs
- Local clocks
- Generated clocks
- Each clock has to be treated as a separate clock, even, if the frequency is the same

# Clocks in a real FPGA based readout



# How to transmit data between different clocks

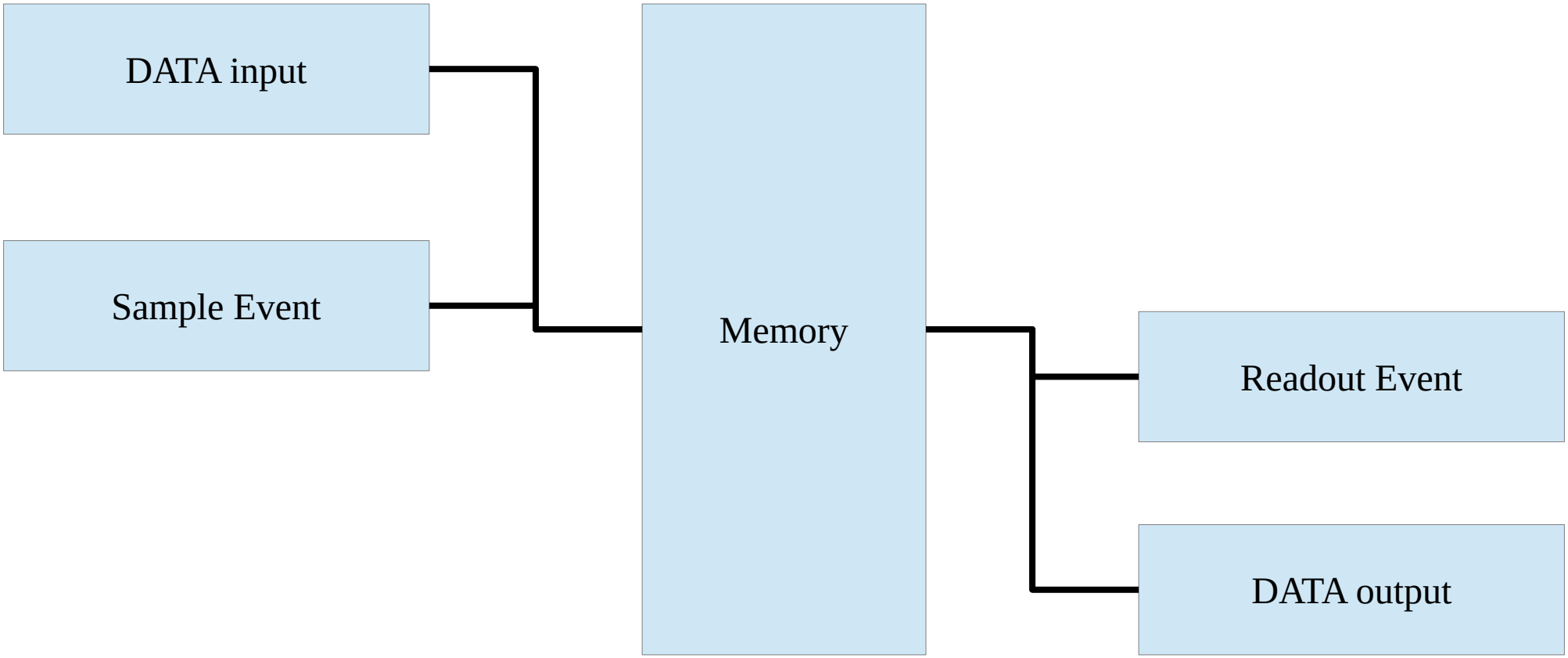
## Clock domain crossing



# Transfering Data between two clocks

- Assumption:
  - we have one “sampling” event, which generates the data
  - additionally we have one “readout” event, which displays the data
  - both events must not be clocks, they can also be events generated by buttons, etc.

# Sketch

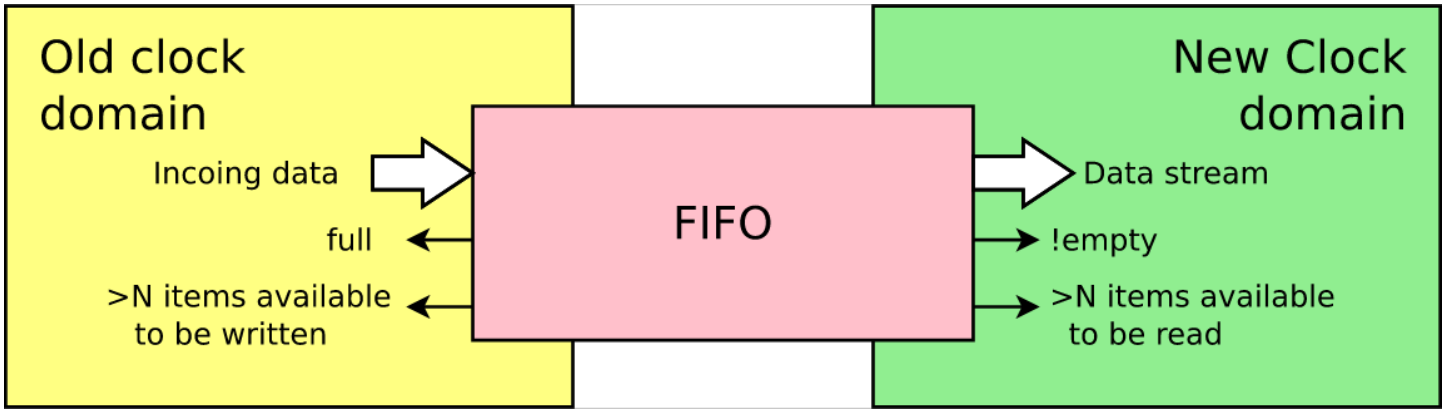


# Important

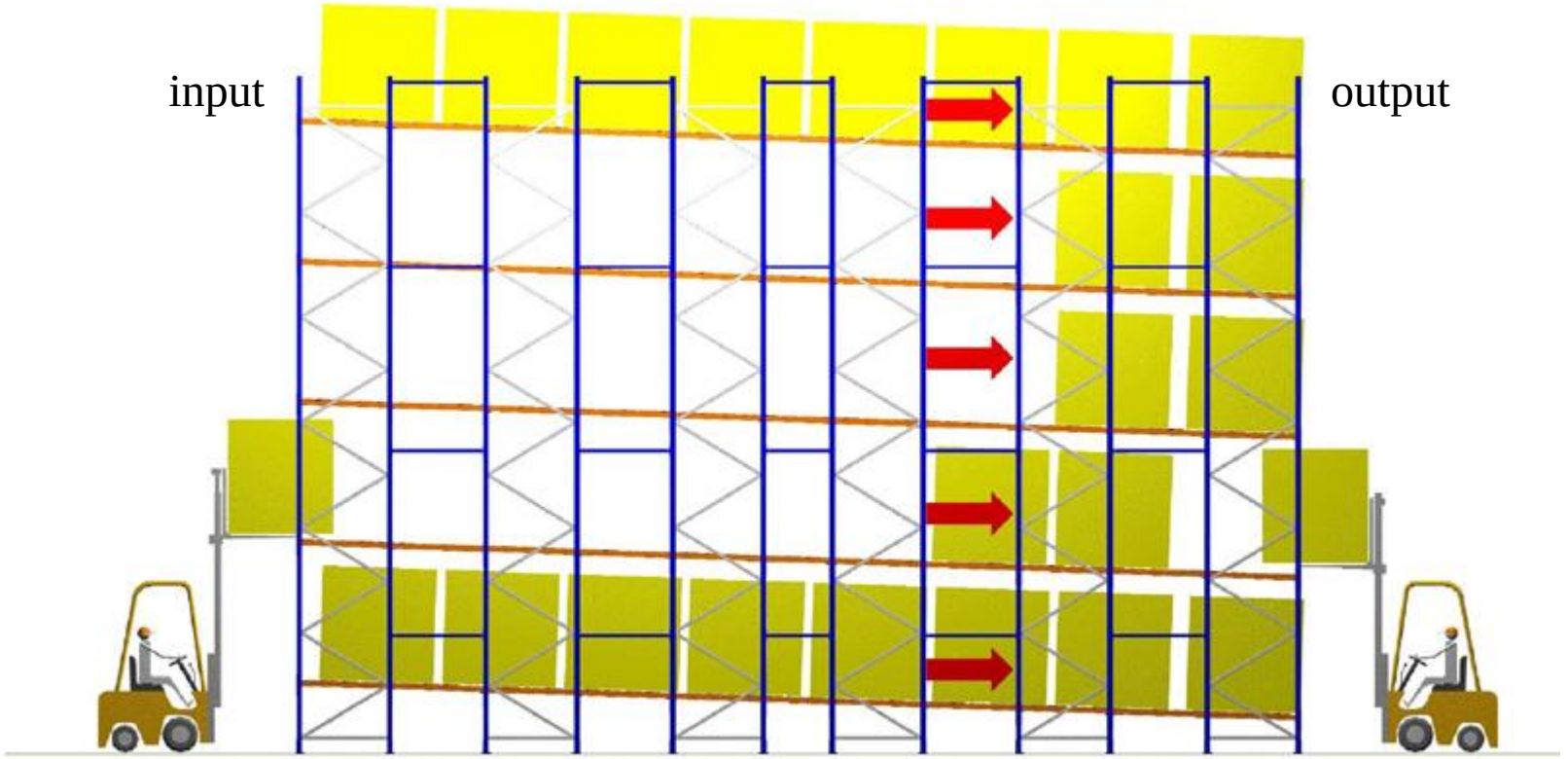
- We want to keep the order of the data
  - first input element should appear at the output as the first element

    **FIFO** → First In First Out

# Using a FiFo


Old clock domain
Incoing data
full
>N items available to be written
FIFO
New Clock domain
Data stream
!empty
>N items available to be read

# FIFO in real life



# FIFO in real life

ssi-schaefer.de/

# Use of FiFos

- Synchronous FiFos
  - read and write clock is the same
    - Data storage
    - Protocol handling
- Asynchronous FiFos
  - read and write clock can be different
  - clock domain crossing
    - data transfer from one clock region to another

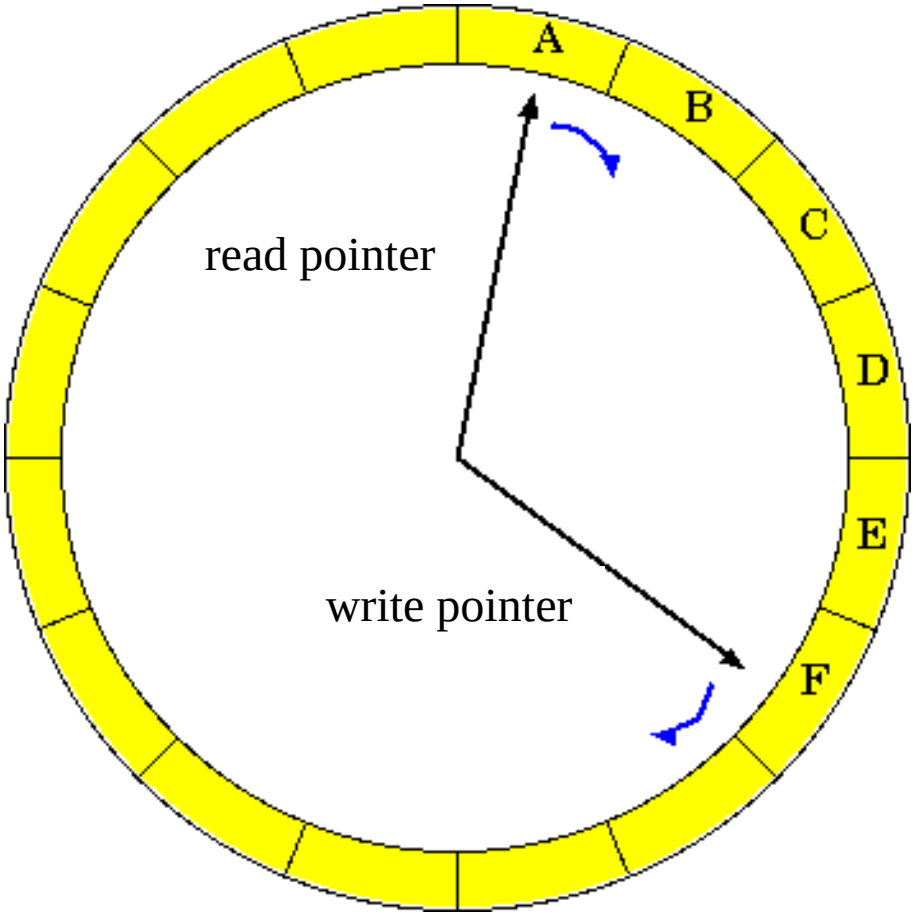
**same clock for read and write**

# FIFO and FPGA

- If you want to use complex FIFOs
  - There is the possibility to use tools from the FPGA vendor to create FiFo
- But: This is an educational course, so we are going to build our own FiFo.

# How is a FiFo working?

- Simple FiFo:
  - using ring buffer
  - read “pointer”
  - write “pointer”

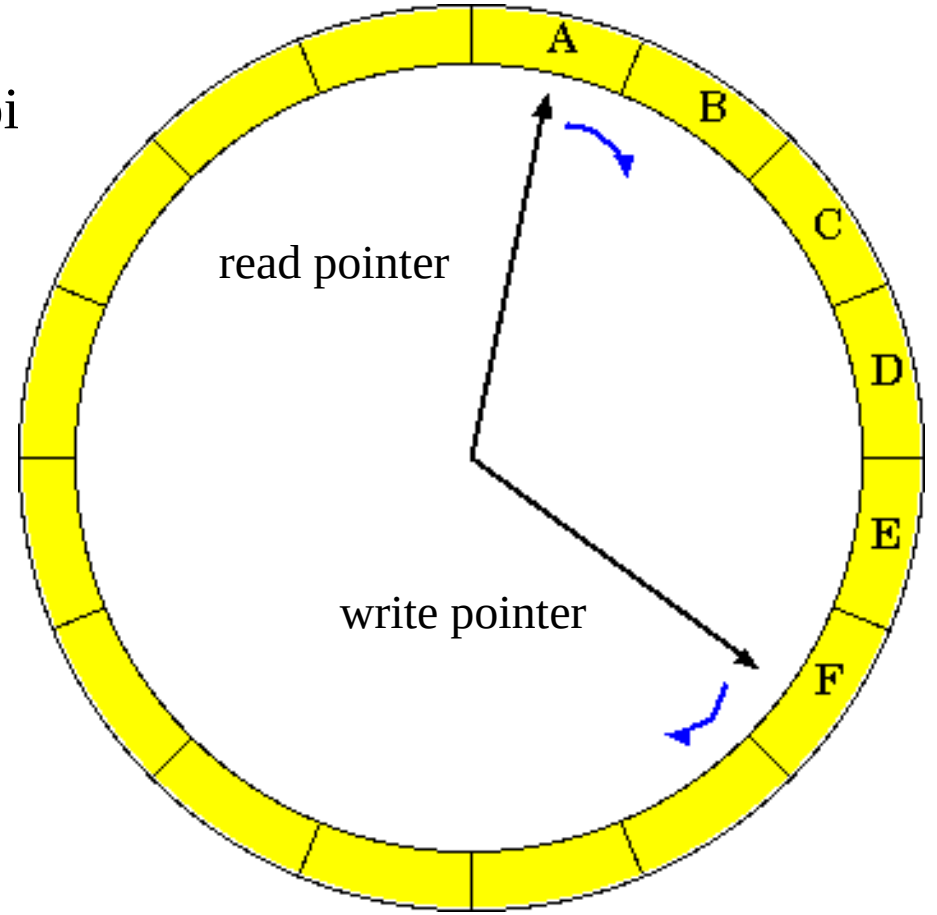


# How to input data to the FiFo

**Writing to FiFo:**
if not full
write data to position given by write poi
increase write pointer by 1

**Reading from FiFo:**
if not empty
read data from position given by
read pointer
increase read pointer by 1

**Bookkeeping:**
count number of elements currently stored in Fifo → count number of read and writes

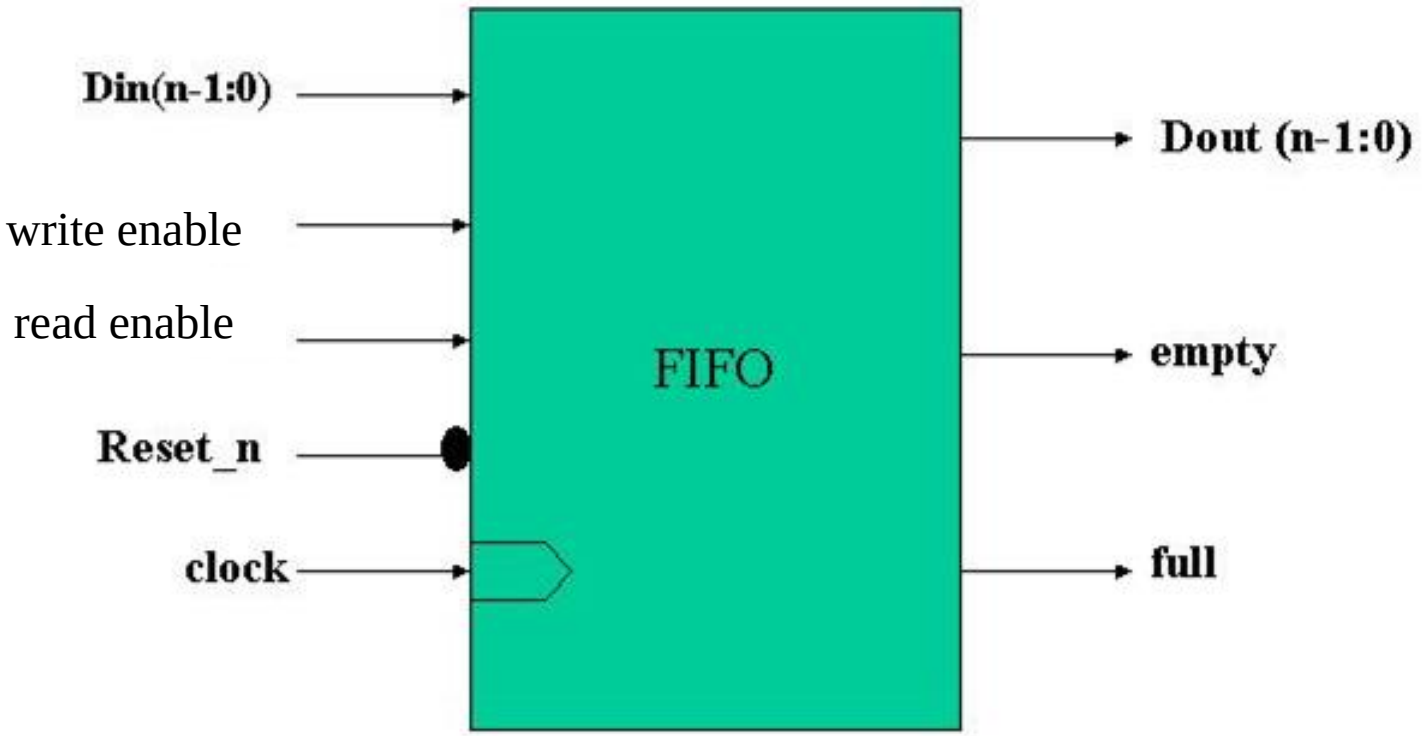



Not simply subtract read and write pointer this might result in trouble!

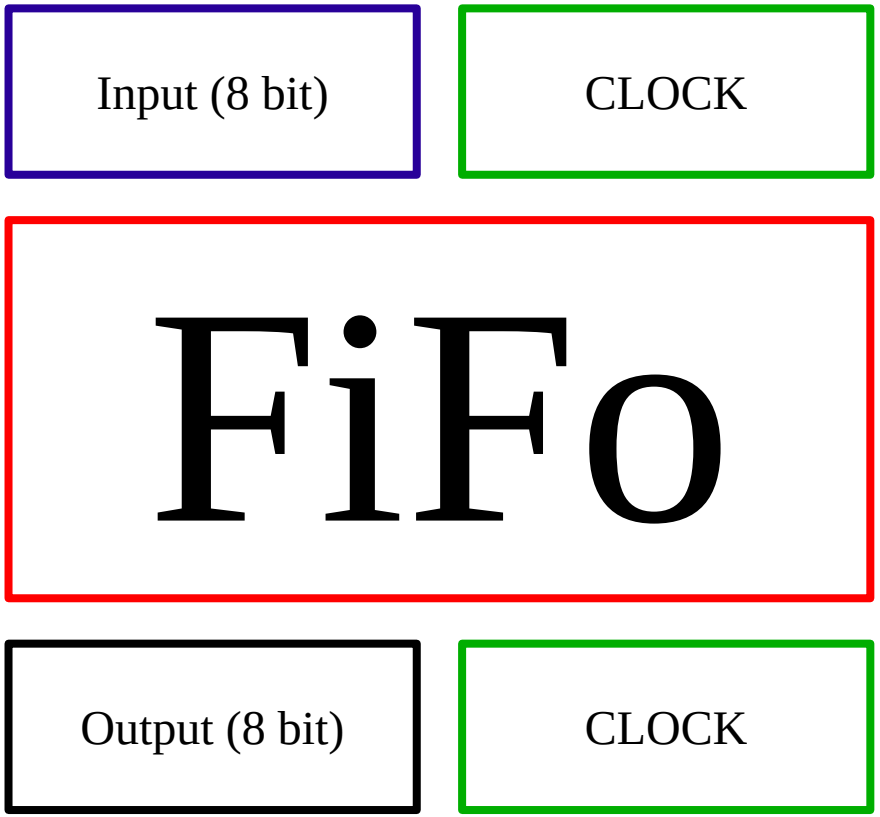
# Our Verilog FiFo

- We are building a synchronous FIFO
  - Read and Write Clock is the same
  - Data transfer on rising edge
- Storage Width => with of input and output bus
- Storage Depth => width of address bus
- Read enable
- Write enable
- Full Flag
- Empty Flag

# FIFO in electronics



# Synchronous FiFo



# Synchronous FiFo

- First start with the implementation of the FiFo
- Hint: when operating a FiFo, we can identify three blocks
  - input (input clock)
    - Write pointer
  - output (output clock)
    - Read pointer
  - bookkeeping (input & output clock)
    - Count elements

# ToDo for today

## 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.

# Thank you!