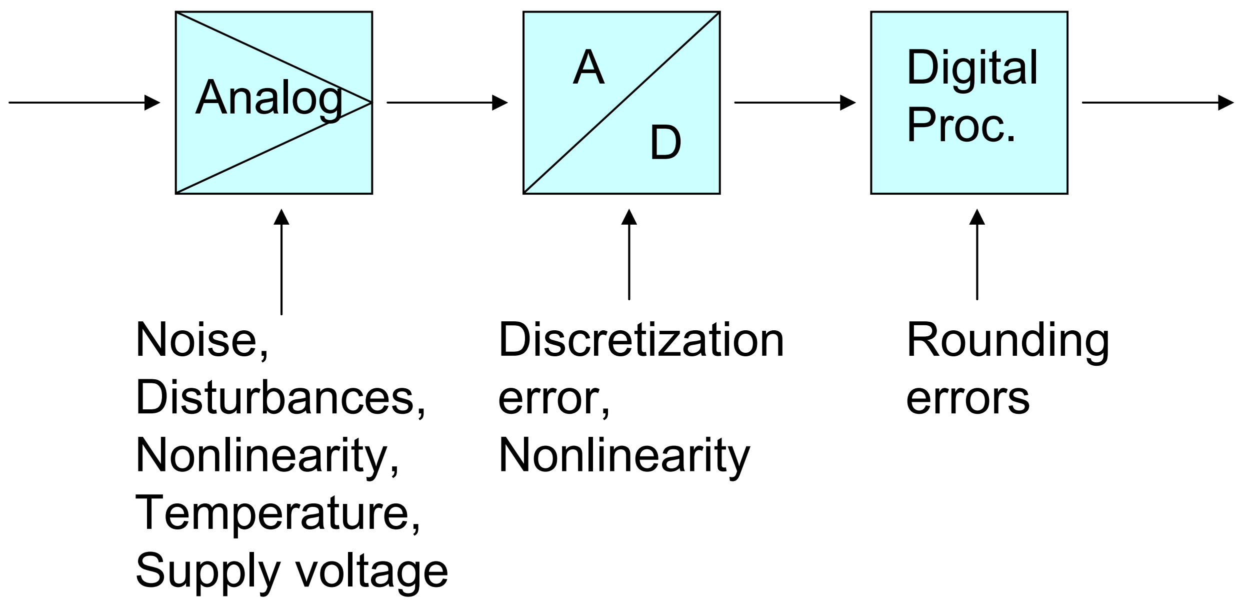
Introduction to Digital Design with VHDL

Topics

- Why digital processing?
- Basic elements and ideas
- Combinational circuits
- VHDL introduction
- Circuits with memory
- Simulating with VHDL
- Technologies – FPGA, ASIC…
- Some practical exercises

Why digital processing (1)?

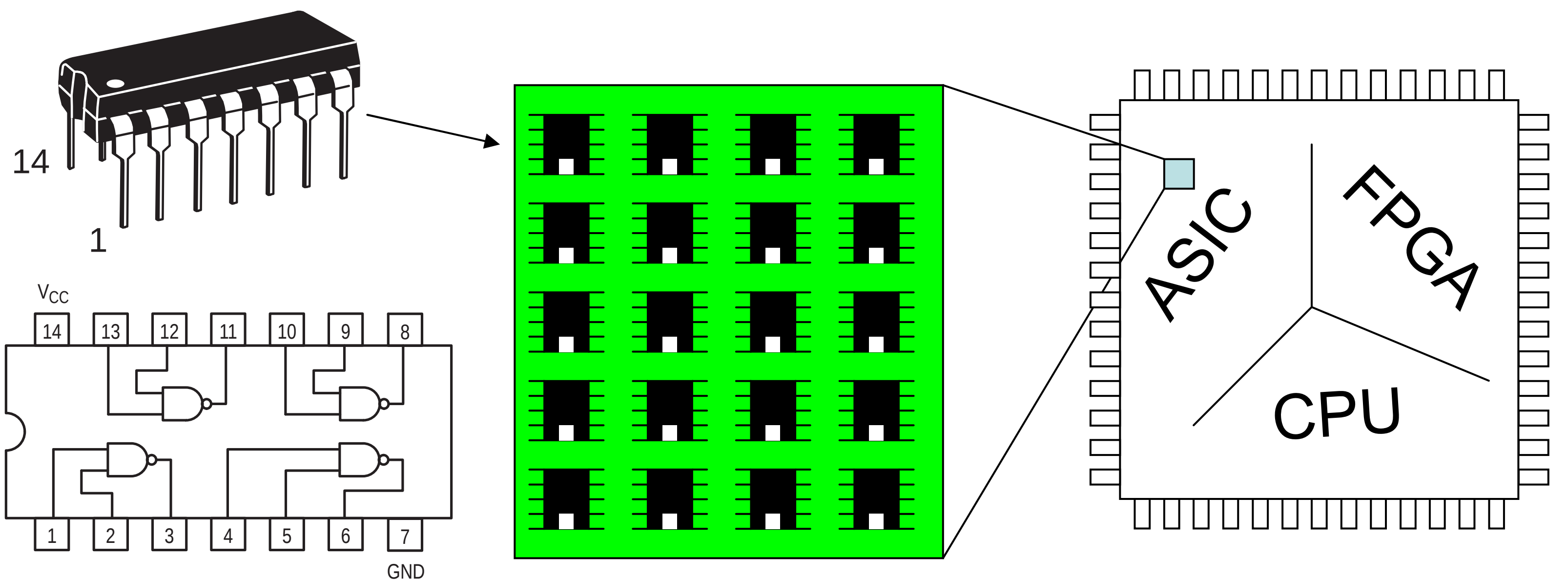
- Block diagram of some measurement device



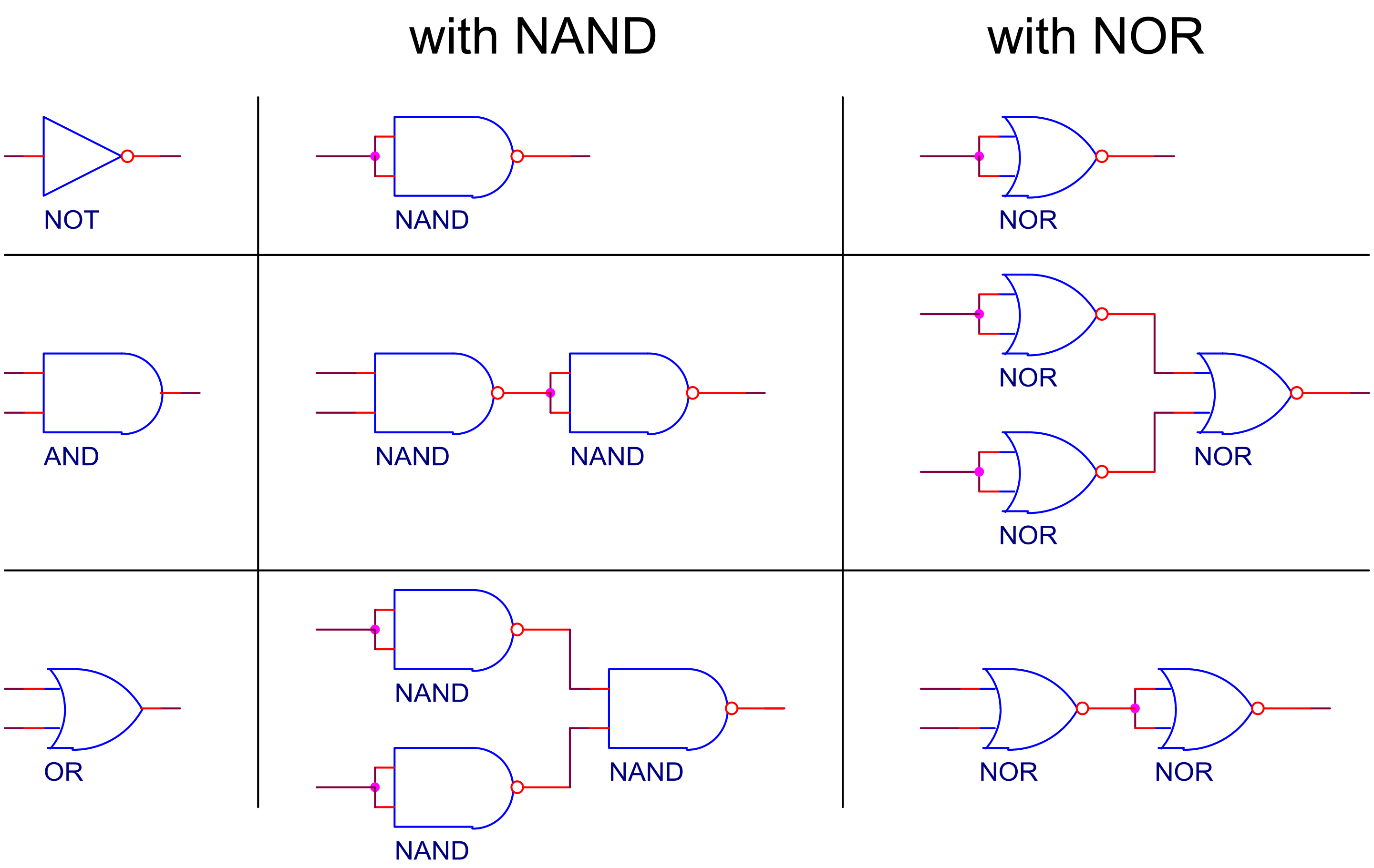
- How to arrange the full processing in order to get the best results?

Why digital processing (2)?

- Where to do what? – the tendency is to start the digital processing as early as possible in the complete chain
- How ? This is our main subject now

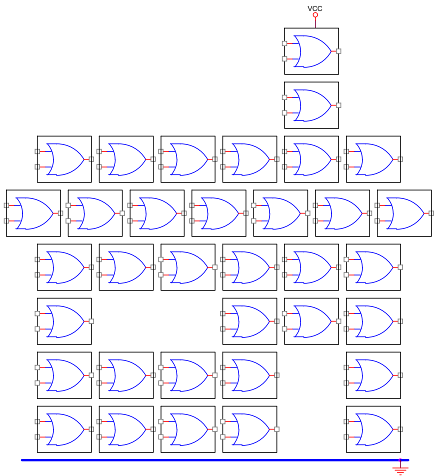


NAND or NOR can do everything…



What kind of logical elements do we need?

- Exactly like a house, that can be built using many identical small bricks, one logical circuit can be built using many identical NOR (OR-NOT) or NAND (AND-NOT) elements
- For practical reasons it is much better to have a rich set of different logical elements, this will save area and power and will speed up the circuit

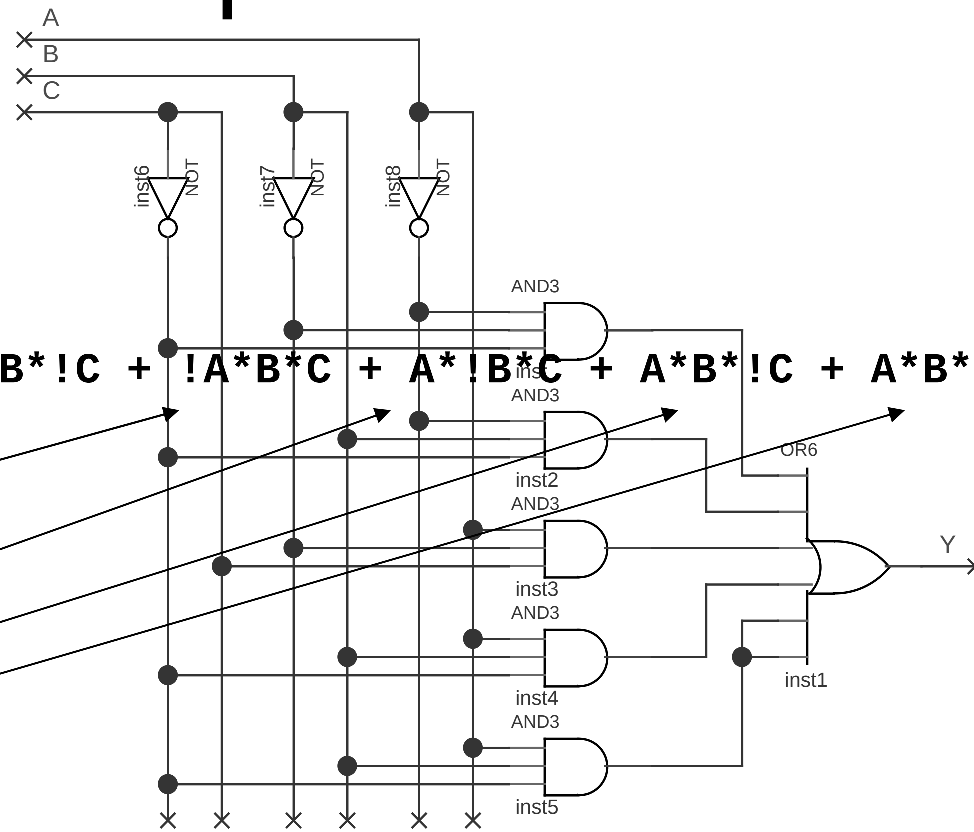


Sum of products representation

- Truth table

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Y = !A*!B*!C + !A*B*C + A*!B*C + A*B*!C + A*B*C



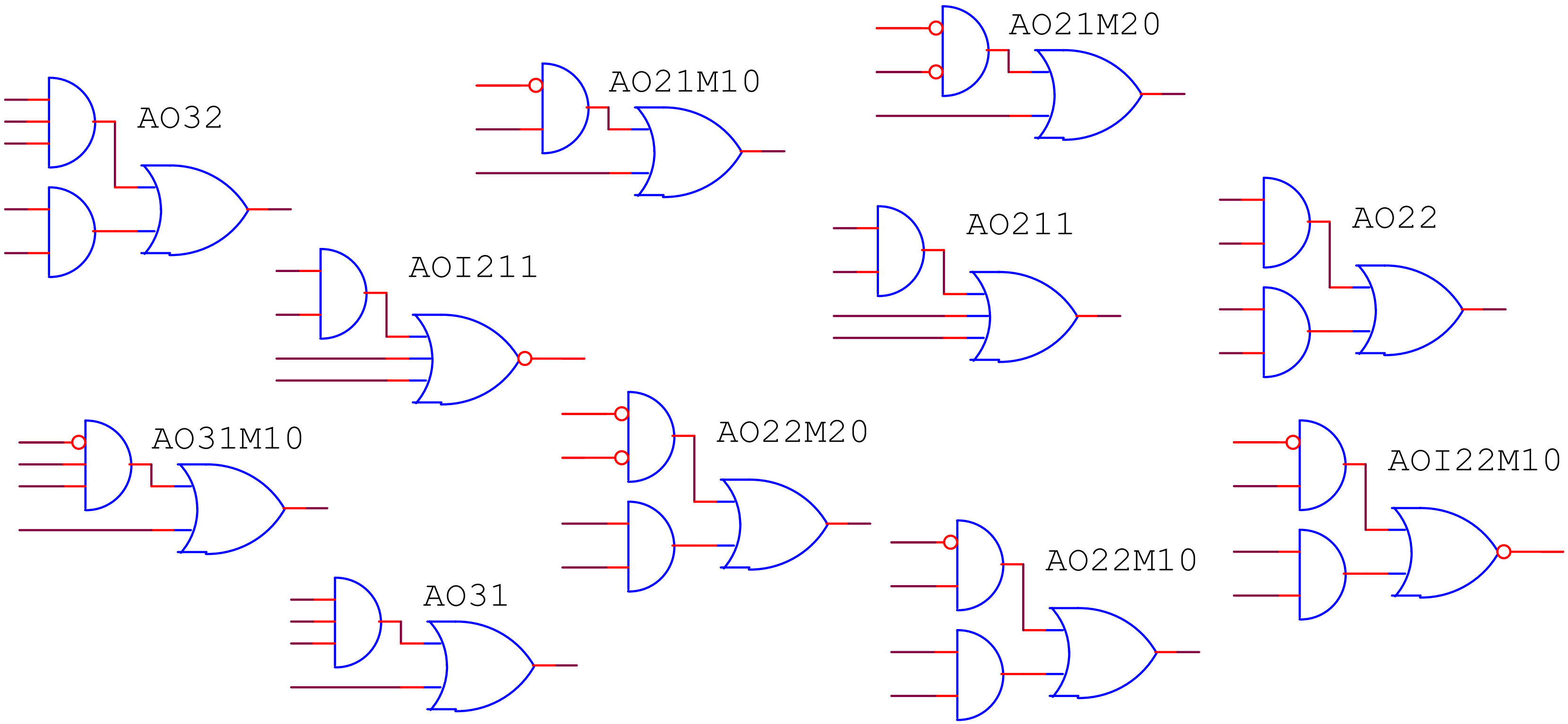
- If the function is more frequently 1, it is better to calculate the inverted function in order to have less terms: **Y <= ! (!A*!B*C + !A*B*!C + A*!B*!C)**

Conclusions(1) – PAL/CPLD/HDL

- The sum of products representation was a good move! It seems to be a universal method (with some exceptions) to build any logical function – PAL and CPLD
- Drawing of the circuit is tedious and not very reliable!
- Writing of equations seems to be easier and more reliable → languages to describe hardware (HDL - hardware description language)

Conclusions(2) – ASIC

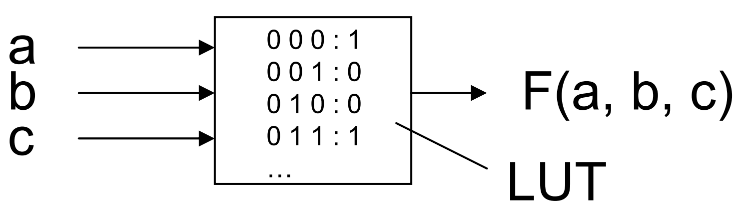
Another possibility is to have many different logic functions. Here are shown only a small subset of the variations with AND-OR-NOT primitive functions available in a typical ASIC library



All about 130 units + with different fanout capability

Conclusions(3) – LUT/FPGA

- Another possible architecture for logical functions is to implement the truth table directly as a ROM
- When increasing the number of the inputs **N**, the size of the memory grows very quickly as $\mathbf{2^N}$!
- If we have reprogrammable small memory blocks (LUT - Look Up Table), we could easily realize any function – the only limit is the number of the input signals

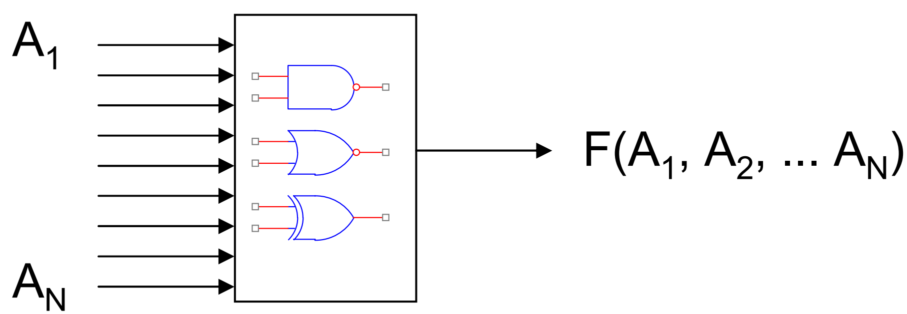


The FPGAs contain a lot of LUT with 4 to 6 inputs + something more

- For larger number of inputs we need to do something

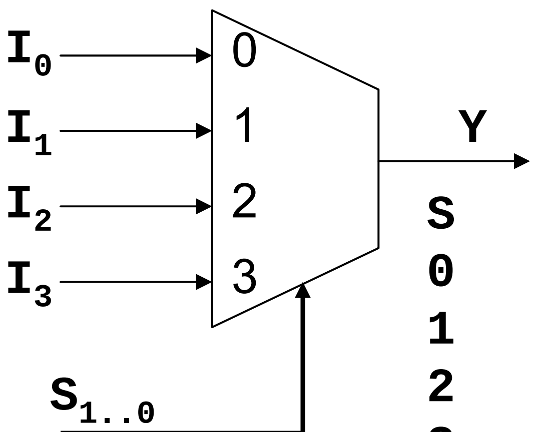
Combinational circuits

- ... are the circuits, where the outputs depend only on the present values of the inputs
- Practically there is always some **delay** in the reaction of the circuit, depending on the temperature, supply voltage, the particular input and the state of the other inputs
- it is good to know the min and max values (worst/best case)

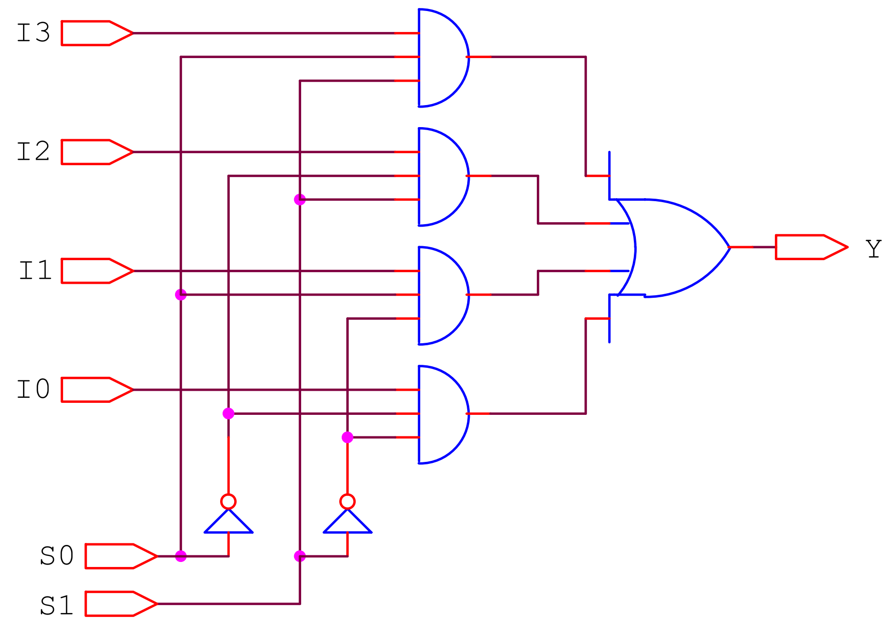


Some combinational circuits - multiplexer

- Used to control data streams – several data sources to a single receiver



S	Y
0	I0
1	I1
2	I2
3	I3

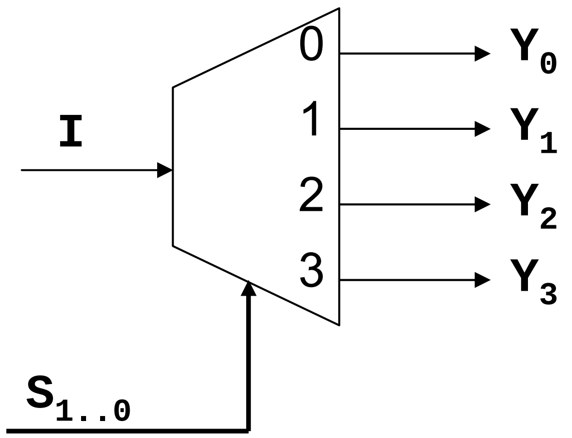



```
with S select
Y <= I0 when "00",
     I1 when "01",
     I2 when "10",
     I3 when others;
```

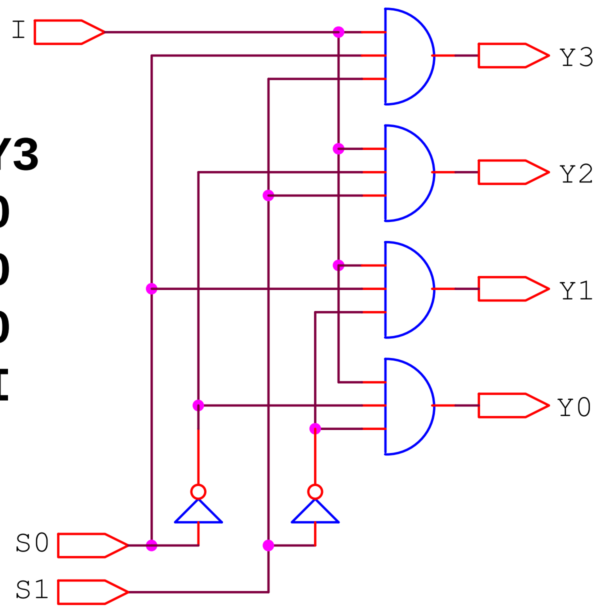
```
Y <= (not S(1) and not S(0) and I0) or
     (not S(1) and     S(0) and I1) or
     (    S(1) and not S(0) and I2) or
     (    S(1) and     S(0) and I3);
```

Some combinational circuits - demultiplexer

- To some extend an opposite to the multiplexer



S	Y0	Y1	Y2	Y3
0	I	0	0	0
1	0	I	0	0
2	0	0	I	0
3	0	0	0	I

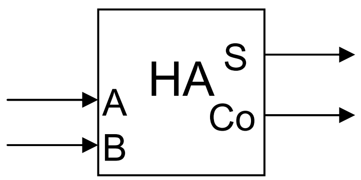


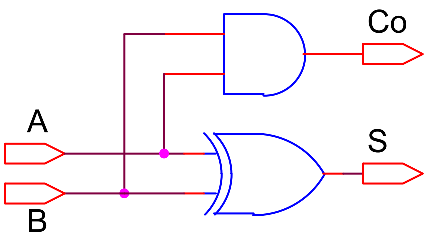

```
with S select
Y <= I & "000"          when "11",
     '0' & I & "00"     when "10",
     "00" & I & '0'     when "01",
     "000" & I          when others;
```

```
Y(0) <=     S(1) and     S(0) and I;
Y(1) <=     S(1) and not S(0) and I;
Y(2) <= not S(1) and     S(0) and I;
Y(3) <= not S(1) and not S(0) and I;
```

Half- and Full- adder

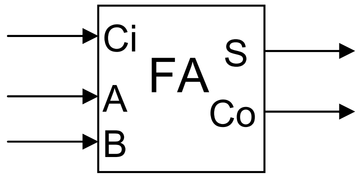
Half-adder

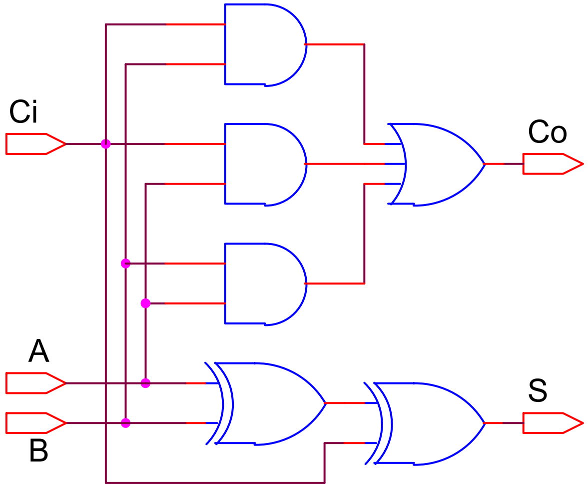





```
S  <= A xor B;
Co <= A and B;
```

Full-adder






```
S  <= A xor B xor Ci;
Co <= (A and  B) or
      (A and Ci) or
      (B and Ci);
```

VHDL

- VHDL = VHSIC Hardware Description Language
  - VHSIC = Very High Speed Integrated Circuit
- Developed on the basis of **ADA** with the support of the USA militaries, in order to help when making ***documentation*** of the digital circuits
- The next natural step is to use it for ***simulation*** of digital circuits
- And the last very important step is to use it for ***synthesis*** of digital circuits
- Standards: 1987,1993, 2000, 2002, 2006…
- Together with **Verilog** is the mostly used language for development of digital circuits
- Extensions for simulations of analogue circuits

Types of data in VHDL(1)

- **time** (**fs, ps, ns, us, ms, sec, min, hr**)
  - **1 ns, 20 ms, 5.2 us**
- **real** **(-1e38..+1e38)**
- **integer** ($-(2^{31}-1)$ .. $2^{31}-1$) with predefined subtypes **natural** (≥0) and **positive** (>0)

```
signal counter : Integer;
signal timer   : Natural;
```

- **boolean** has two possible values **FALSE** and **TRUE**
  - Not intended for electrical signals!
  - Typically used when checking some conditions, like

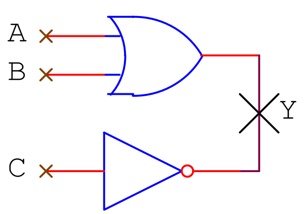
```
if a  = b then --  equal
if a /= b then --  not equal
if a >  b then --  larger
if a <  b then --  smaller
if a <= b then --  smaller or equal
if a >= b then --  larger or equal
```

↖the result of the comparison is a **boolean**

Types of data in VHDL(2)

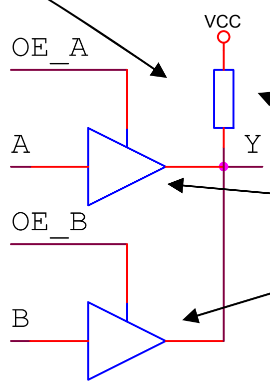
- **bit** has two possible values **'0'** and **'1'**
  - These two values are not enough to model real hardware!
- **std_logic** to the **'0'** and **'1'**, introduced 7 additional values for tri-stated (**'Z'**), unknown (**'X'**), weak 0 (**'L'**), weak 1 (**'H'**), weak unknown (**'W'**), uninitialized (**'U'**) and don‘t care (**'-'**)

This is allowed only when using **std_logic** but not **bit**!




```
Y <= not C;
Y <= A or B;
```

Example for pull-up (weak 1) and tri-stated outputs, **std_logic** is required



tri-stated output

```
Y <= 'H';
Y <= A when OE_A='1' else 'Z';
Y <= B when OE_B='1' else 'Z';
```

More complex data types

- Array
  - predefined in **IEEE.STD_LOGIC_1164**, e.g. **std_logic_vector(3 downto 0);**

```
subtype reg_data is std_logic_vector(31 downto 0);
type mem_array is array(0 to 63) of reg_data;
```

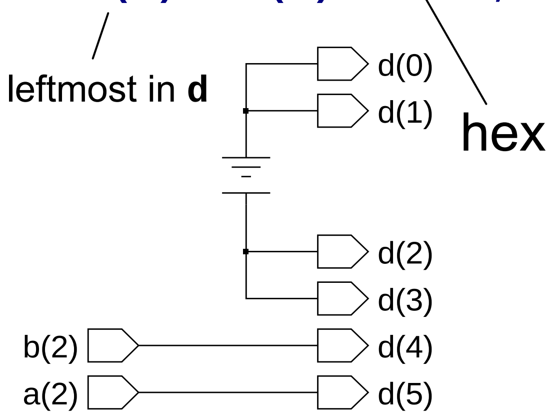
the direction of the index

- Enumerated
  - Used mainly to describe state machines

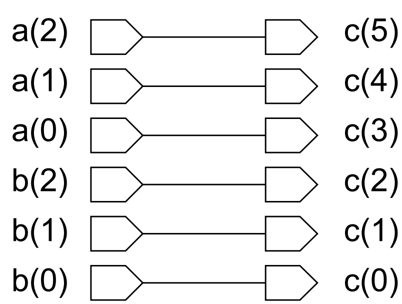
```
type state_type is (idle, run, stop, finish);
```

```
a : in  std_logic_vector(2 downto 0);
b : in  std_logic_vector(2 downto 0);
c : out std_logic_vector(5 downto 0);
d : out std_logic_vector(5 downto 0);
```

```
d <= a(2) & b(2) & X"C";
```




```
c <= a & b;
```



Mathematical operations with **std_logic_vector**s

- Using appropriate library it is possible to mix different types in mathematical operations and to apply mathematical operations (+ or -) to non-integer objects

```
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;
...
signal data11b, data_inc : std_logic_vector(10 downto 0);
...
data_inc <= data11b + 1;
```

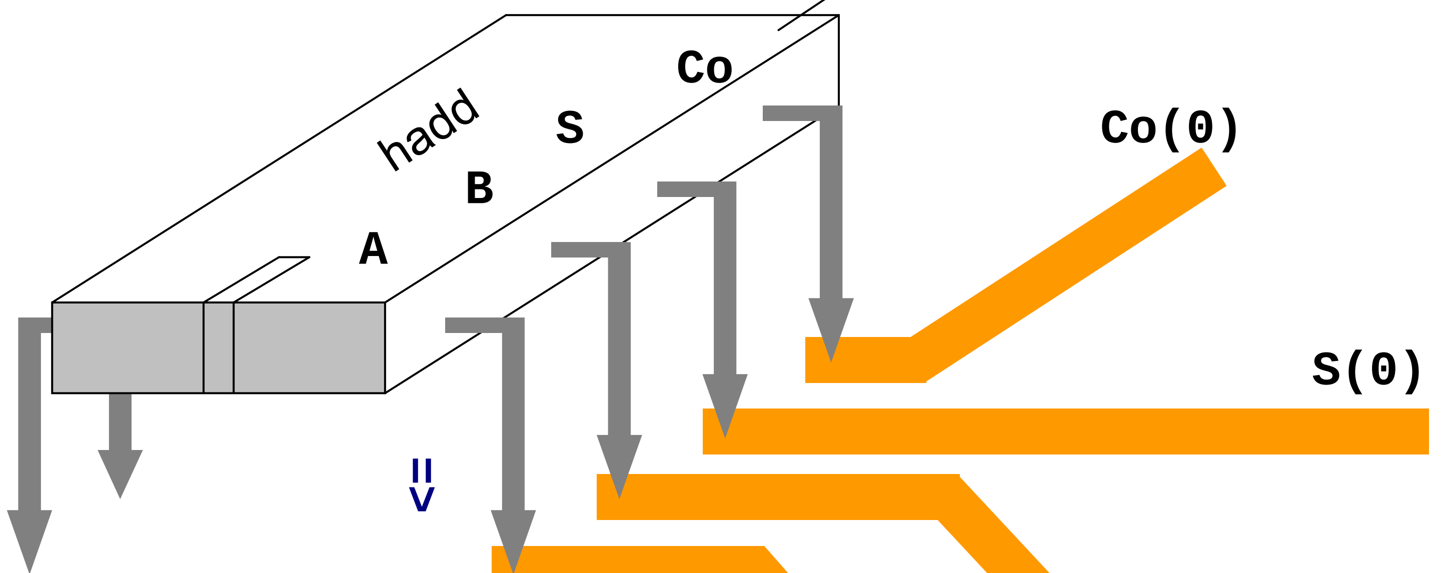
very important (→ downto)

If **data11b** is **"11111111111" (2047)**, the result will be 0!

- The same is possible with the multiplication, but be careful, the multipliers are large! Use only for power of 2!
- For synthesis the division is supported only for power of 2, in this case it is just a shift (arithmetical or logical?)
- For every technology there are libraries with optimized modules for mathematical operations

Port-signal mapping – how to remember

The **entity** is like a IC



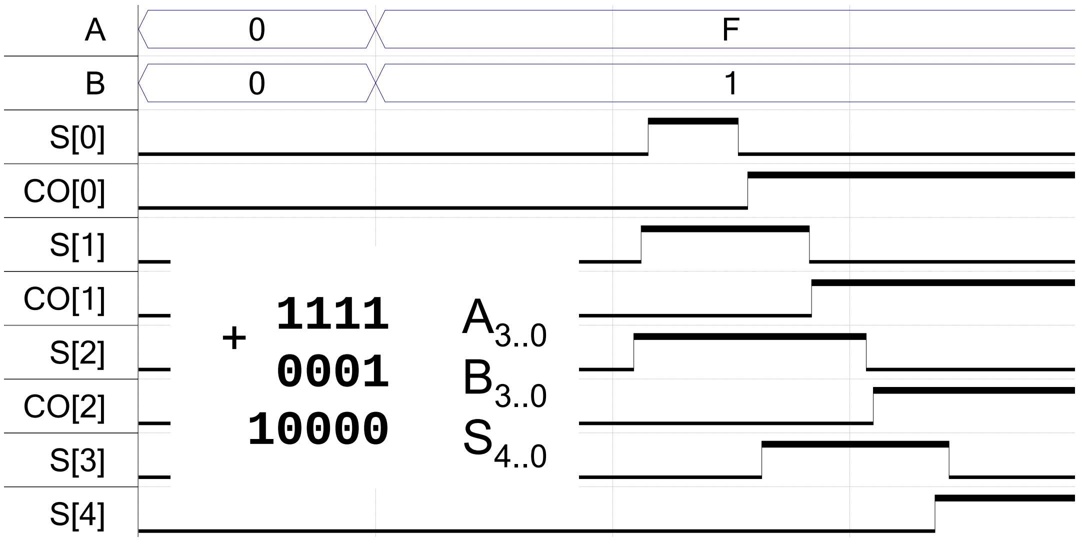

```
i0: hadd
port map(
  A  => A(0),
  B  => B(0),
  S  => S(0),
  Co => Co(0));
```

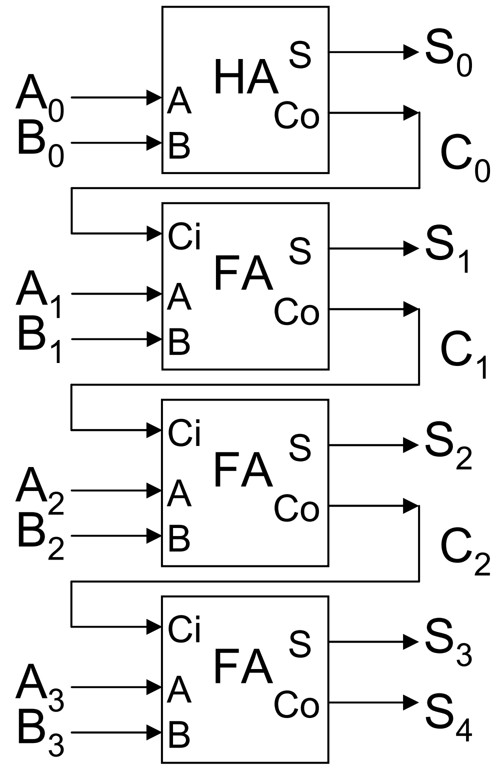
port names

signal names

The signals are like the routes on the printed circuit board (PCB)

4 bit ripple carry adder





Use 1 half-adder and 3 full-adder properly connected

```
i0: hadd port map(             A => A(0), B => B(0), Co => Co(0), S => S(0));
i1: fadd port map(Ci => C(0), A => A(1), B => B(1), Co => Co(1), S => S(1));
i2: fadd port map(Ci => C(1), A => A(2), B => B(2), Co => Co(2), S => S(2));
i3: fadd port map(Ci => C(2), A => A(3), B => B(3), Co => S(4),  S => S(3));
```

or

```
S <= ('0' & A) + ('0' & B);
```

entity
port
in out
inout
buffer
architecture
signal

Structure of an **entity** in VHDL

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```

+ other library declarations, this is the standard minimum

```
entity <entity_name> is
port (
  <port_name>  : <in|out|inout|buffer> <signal_type>;
  ...
  <port_name>  : <in|out|inout|buffer> <signal_type>);
end <entity_name>;
```

port type (annotation pointing to `<in|out|inout|buffer>`)

```
architecture <arch_name> of <entity_name> is

...
signal <internal_signal_name> : <signal_type>;
...
begin
  -- comment to the end of the line
  ...
end [<arch_name>];
```

+ optional `type`, `constant` and `component` declarations (annotation pointing to the first `...`)

Unlike C and Verilog, VHDL is not case-sensitive!

`when … else`
`process`
`if … then …`
`elsif … end if`

Priority logic constructs

```
irq_no <= "11" when IRQ(3) = '1' else
          "10" when IRQ(2) = '1' else
          "01" when IRQ(1) = '1' else
          "00" when IRQ(0) = '1' else
          "--";
valid <= IRQ(0) or IRQ(1) or IRQ(2) or IRQ(3);
```

1-st method
(dataflow style)

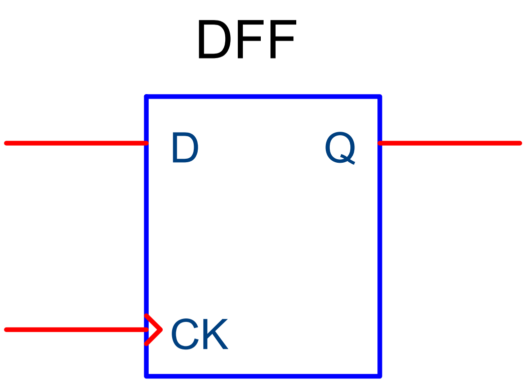
```
pri: process(IRQ)
     begin
      valid  <= '1';
      irq_no <= "--";
      if     (IRQ(3) = '1') then irq_no <= "11";
      elsif  (IRQ(2) = '1') then irq_no <= "10";
      elsif  (IRQ(1) = '1') then irq_no <= "01";
      elsif  (IRQ(0) = '1') then irq_no <= "00";
      else valid <= '0';
      end if;
     end process;
```

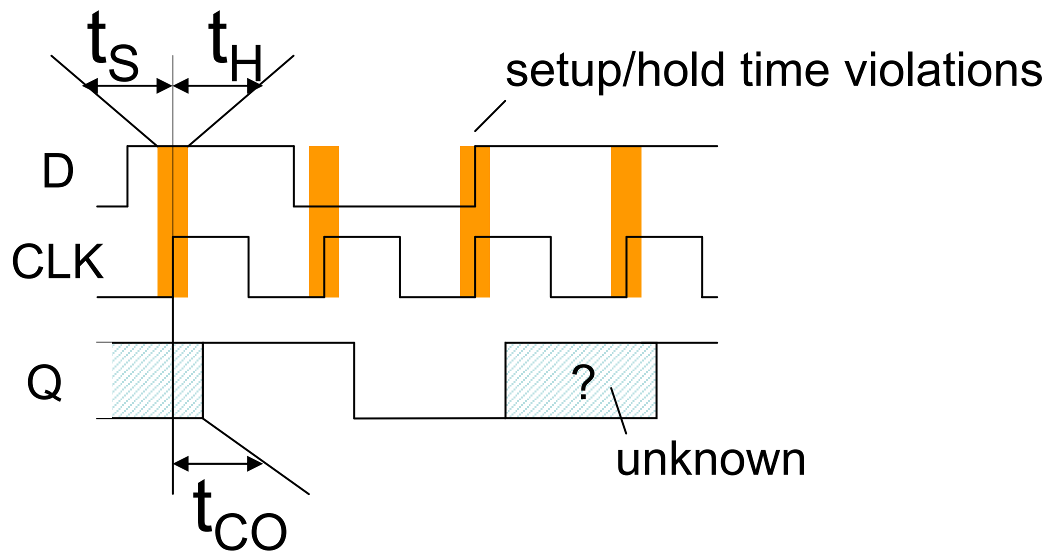
← sensitivity list

2-nd method,
using a **`process`**
(behaviour style)

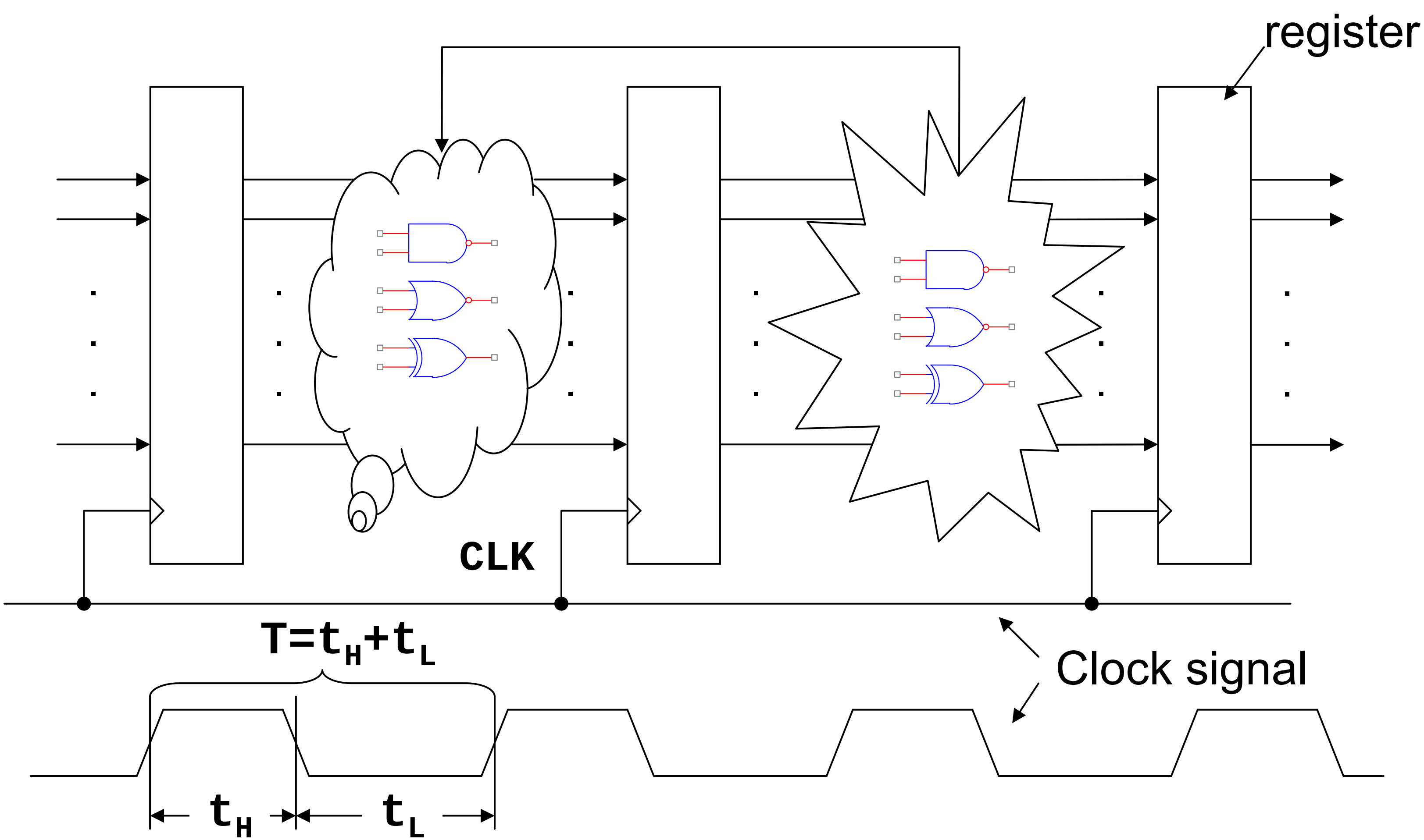
Circuits with memory – D flip-flop

- Q memorizes D on the rising (falling) edge of the clock signal
  - **Currently the most used memorizing component together with the memories (RAM)**
  - Some flip-flop types have an additional enable input and asynchronous set or reset inputs
- D must be stable t_S (setup) before and t_H (hold) after the active edge of the clock signal CLK
- The output Q settles within some time t_{CO}, if the conditions are violated (t_S , t_H) the state of the flip-flop is unknown, oscillations are possible





Synchronous circuits



At each rising clock edge the registers memorize the current values at their inputs. The outputs are updated after some small delay t_{CO}

Sequential circuits in VHDL DFF, DFFE

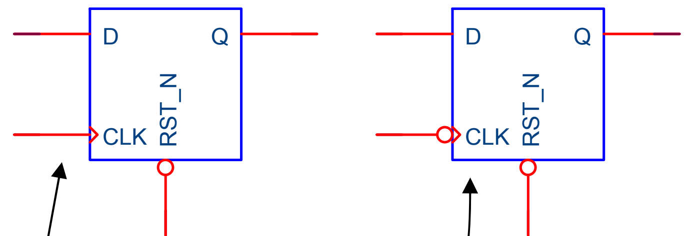
sensitivity list

DFF

```
process(clk, rst_n)
begin
    if rst_n = '0' then q <= '0';
    elsif clk'event and clk='1' then
        q  <= d;
    end if;
end process;
```

attribute

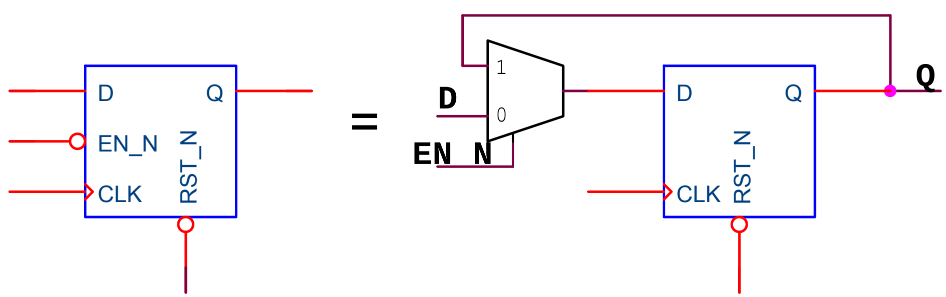
or '0'




```
rising_edge(clk)
```
or
```
falling_edge(clk)
```

DFFE = DFF with enable

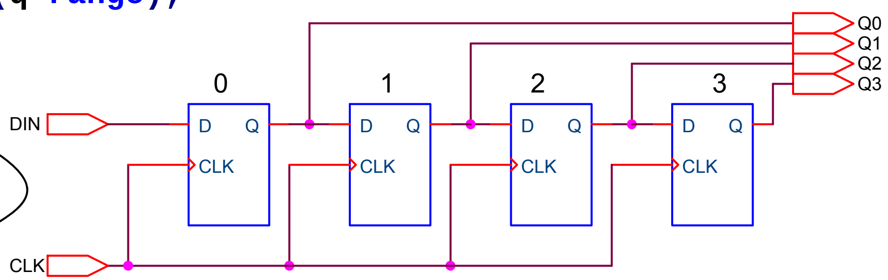
```
process(clk, rst_n)
begin
    if rst_n = '0' then q <= '0';
    elsif clk'event and clk='1' then
      if en_n = '0' then
        q  <= d;
      end if;
    end if;
end process;
```



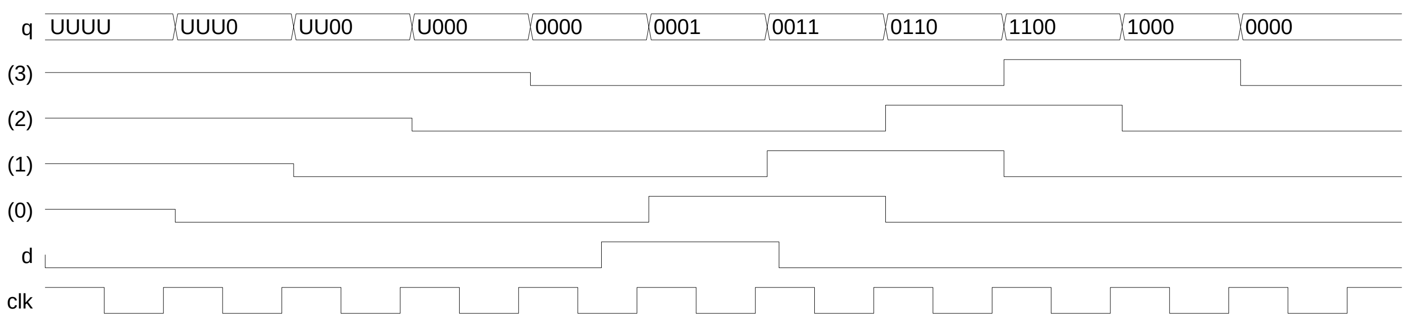
Shift register

```
entity shift_reg4 is
port (clk : in  std_logic;
      d   : in  std_logic;
      q   : out std_logic_vector(3 downto 0));
end shift_reg4;

architecture a of shift_reg4 is
signal q_i : std_logic_vector(q'range);
begin
process(clk)
begin
 if rising_edge(clk) then
  q_i <= q_i(2 downto 0) & d;
 end if;
end process;
 q <= q_i;
end;
```

Used to create delays (pipeline), for serial communication, pseudo-random generator, ring counter etc.



A entity output can not be read back, therefore the q_i here



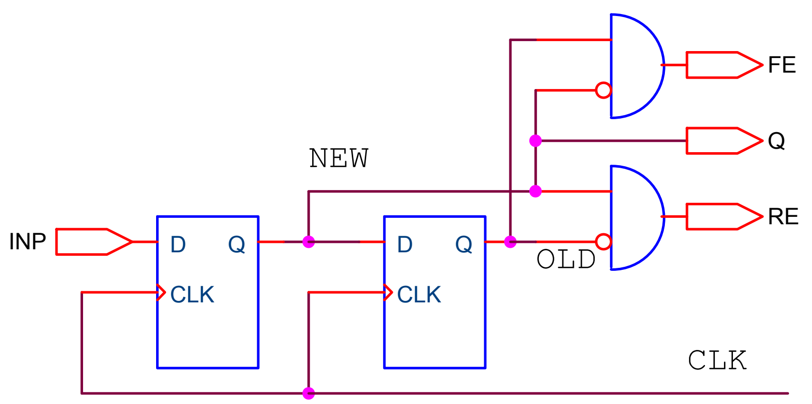
Detecting events in a synchronous design

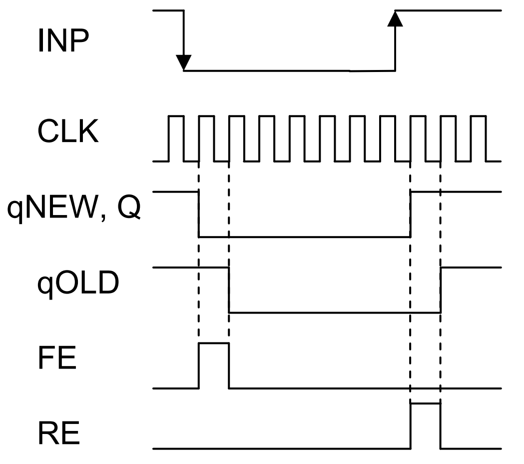
- On the first glance we could directly use VHDL constructs like

  `if rising_edge (inp) then …`

  but for many reasons this is not good
- Use a small shift register and logic to detect changes on the input signal
- Use a single clock for the whole design and generated signals like **FE** or **RE** to enable the desired action for one clock period

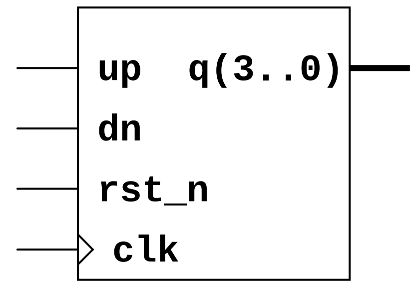
```
signal qNEW : std_logic;
signal qOLD : std_logic;
begin
process(clk)
 begin
  if rising_edge(clk) then
    qNEW <= INP;
    qOLD <= qNEW;
  end if;
 end process;
q <= qNEW;
RE <=     qNEW and not qOLD;
FE <= not qNEW and     qOLD;
```

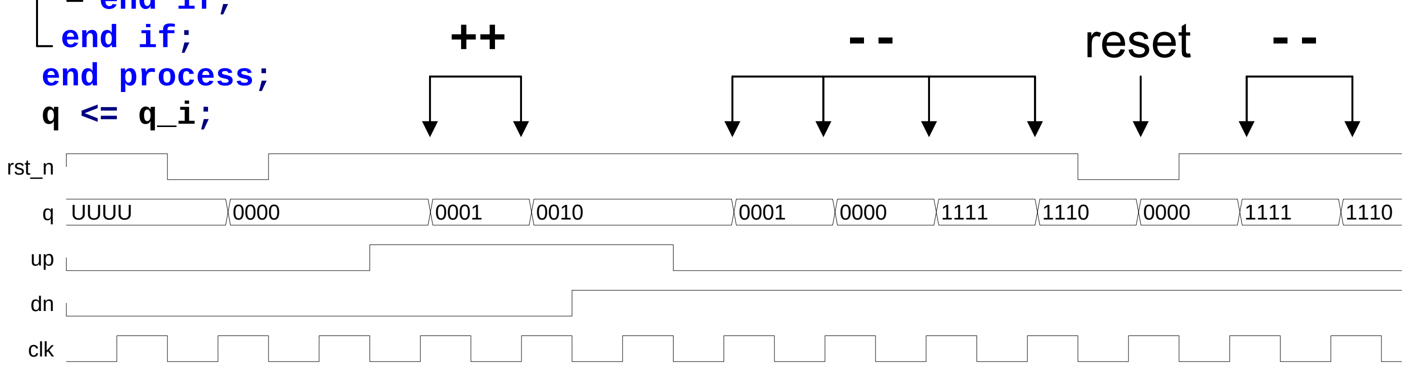




Up/down counter with synchronous reset

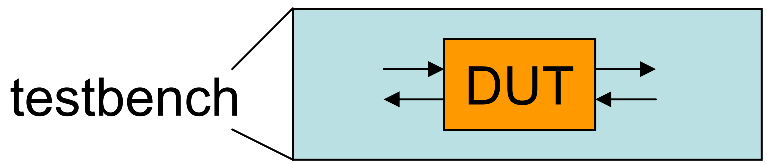
```
signal q_i  : std_logic_vector(3 downto 0);
signal mode : std_logic_vector(1 downto 0);
begin
  mode <= up & dn;
process(clk)
begin
 if clk'event and clk='1' then
   if rst_n = '0' then q_i <= (others => '0');
   else
     case mode is
       when "01" => q_i <= q_i - 1; -- down
       when "10" => q_i <= q_i + 1; -- up
       when "00" | "11" => NULL; -- do nothing
       when others => q_i <= (others => 'X'); -- should never happen!
     end case;
   end if;
 end if;
end process;
q <= q_i;
```





How to simulate – testbench

- Instantiate the design under test (DUT) into the so called **testbench**
- All signals to the DUT are driven by the **testbench**, all outputs of the DUT are read by the testbench and if possible analyzed



- Some subset of all signals at all hierarchy levels can be shown as a waveform
- The simulation is made many times at different design stages – functional, after the synthesis, after the placing and routing, sometimes together with the other chips on the board
- Many VHDL constructs used in a testbench can not be synthesized, or are just ignored when trying to make a synthesis

Simple test bench example

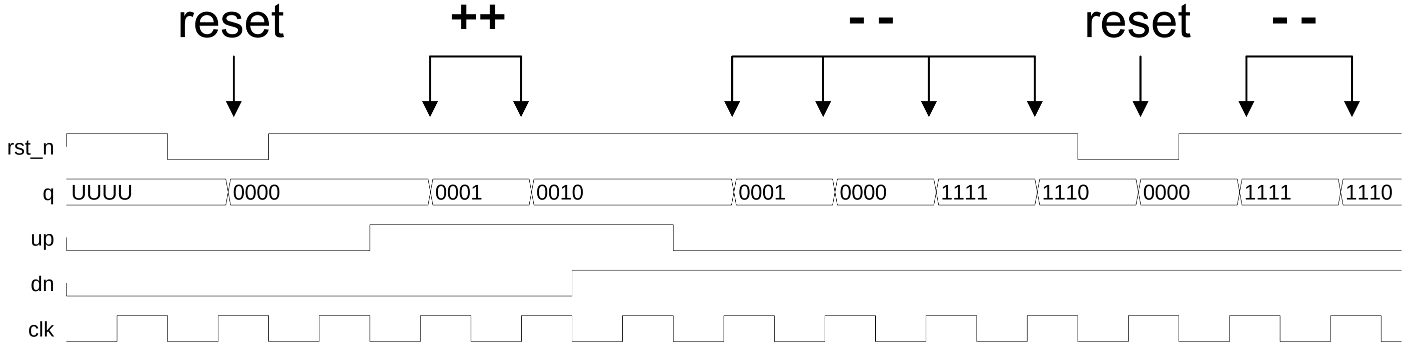
```
entity counter_updn_tb is          <-- no ports!
end counter_updn_tb;

architecture sim of counter_updn_tb is

component counter_updn is
port (clk   : in  std_logic;        } Component
      rst_n : in  std_logic;        } declaration
      up    : in  std_logic;
      dn    : in  std_logic;
      q     : out std_logic_vector(3 downto 0) );
end component;

signal rst_n : std_logic;                        } Signals used in the
signal q     : std_logic_vector(3 downto 0);     } testbench
signal up    : std_logic;
signal dn    : std_logic;
signal clk   : std_logic:= '0';  <-- initial value

begin
  clk <= not clk after 50 ns;  <-- Clock generation
```

Component instantiation

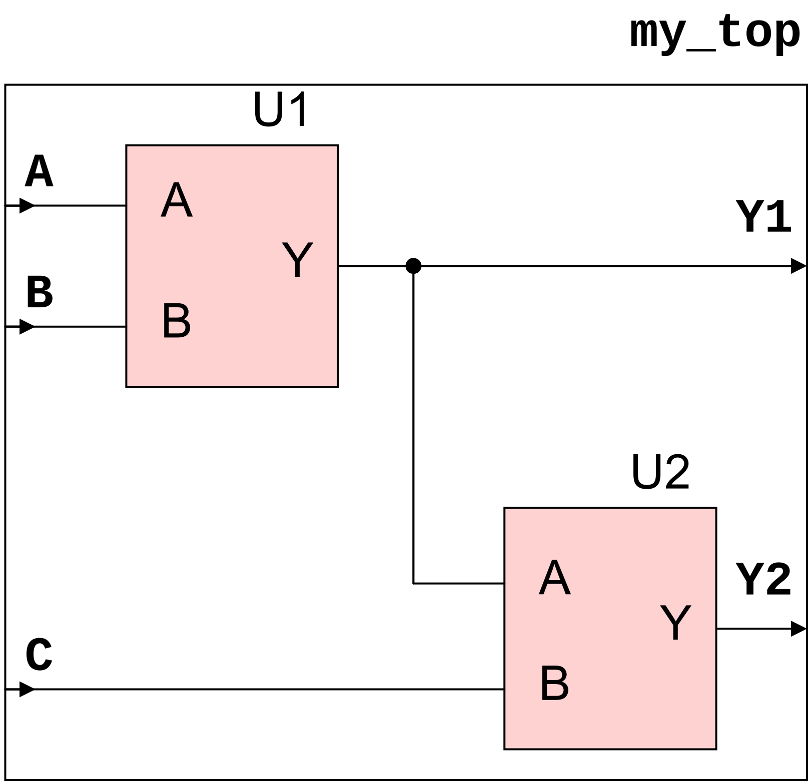
```
uut: counter_updn
port map(
    clk   => clk,
    rst_n => rst_n,
    up    => up,
    dn    => dn,
    q     => q);

end;
```

Test bench – stimuli generation

```
process
begin
    rst_n <= '1';
    up    <= '0';
    dn    <= '0';
    wait until falling_edge(clk);
    rst_n <= '0';
    wait until falling_edge(clk);
    rst_n <= '1';
    wait until falling_edge(clk);
    up <= '1';
    wait until falling_edge(clk);
    wait until falling_edge(clk);
    dn <= '1';
    wait until falling_edge(clk);
    up <= '0';
    for i in 1 to 4 loop
        wait until falling_edge(clk);
    end loop;
    rst_n <= '0';
    wait until falling_edge(clk);
    rst_n <= '1';
    wait;
end process;
```



Structural approach: top-down

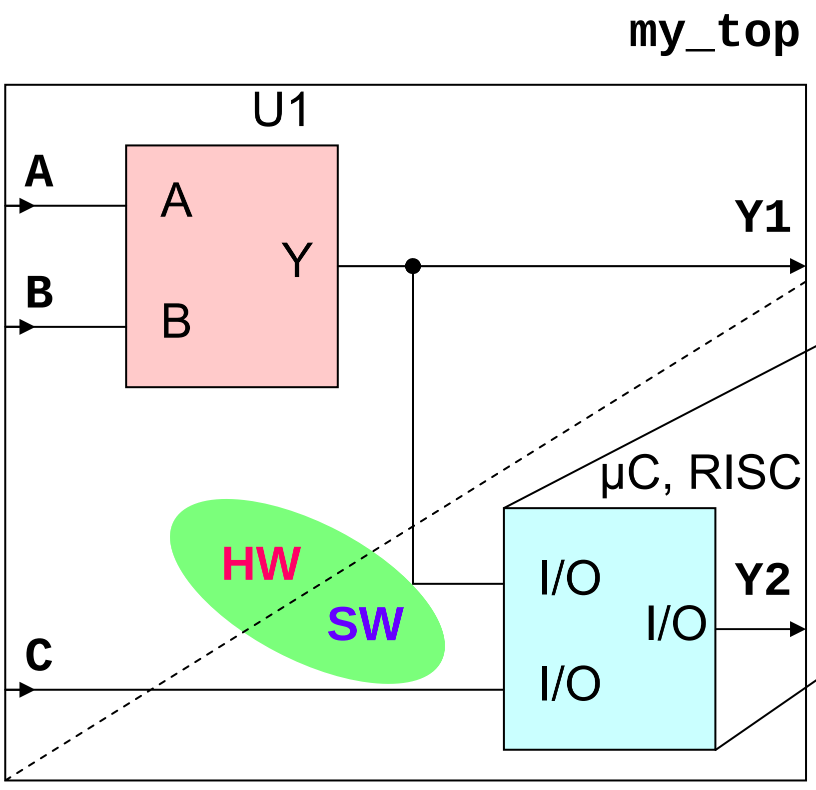


Iterative process !

- Don't delay the documentation, it is part of each design phase

- Try to understand the problem, do not stop at the first most obvious solution
- Divide into subdesigns (3..8), with possibly less connections between them, prepare block diagrams before starting with the implementation
- Clearly define the function of each block and the interface between the blocks, independently on the implementation(s) of each block
- Develop the blocks (in team) and then check the functionality
- Combine all blocks into the top module, if some of them is not finished, put temporarily a dummy

Hardware : software?



- Divide in two parts - hardware : software, taking into account the desired speed, size, flexibility, power consumption and other conditions

```
again: inc r5
        load r2, [r5]
       and r2, 0xAB
       bra cc_zero, again
        store [r3], r6
       ...
```

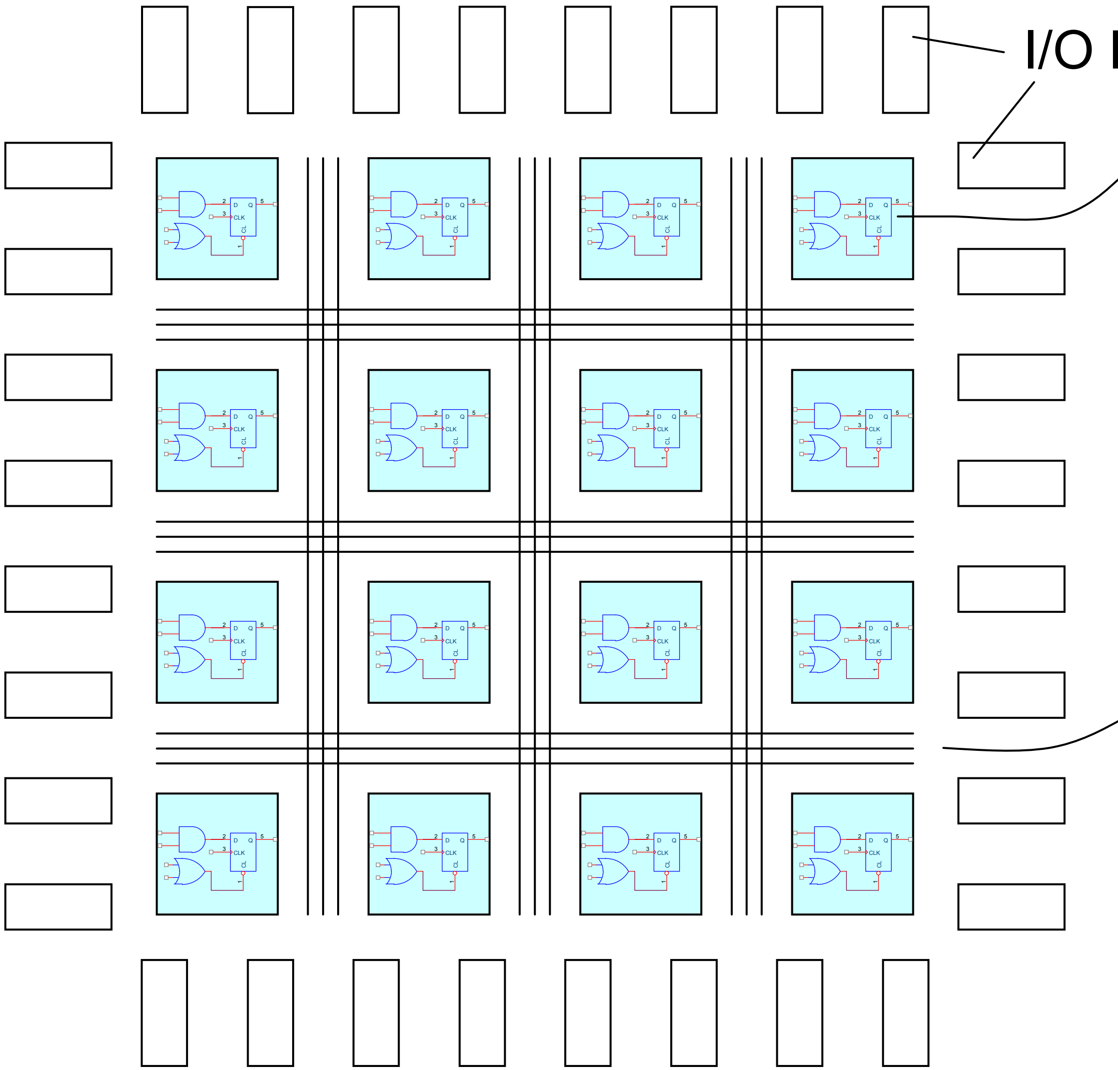
- select the processor core
- for the architecture of the hardware part proceed as described before

Technologies

Integration scales & technologies

- Small Scale Integration (SSI) ICs (74xx, 4000)
- Simple Programmable Logic Devices (SPLD) - PAL (Programmable Array Logic) & GAL (Generic Array Logic), Complex Programmable Logic Devices (CPLD)
  - Architecture, manufacturers, overview of the available products
- Field Programmable Gate Arrays (FPGA)
  - Architecture, manufacturers, overview of the available products
  - Design flow FPGA/CPLD
- Application Specific Integrated Circuits (ASIC)

FPGA – general structure



I/O Blocks and pins

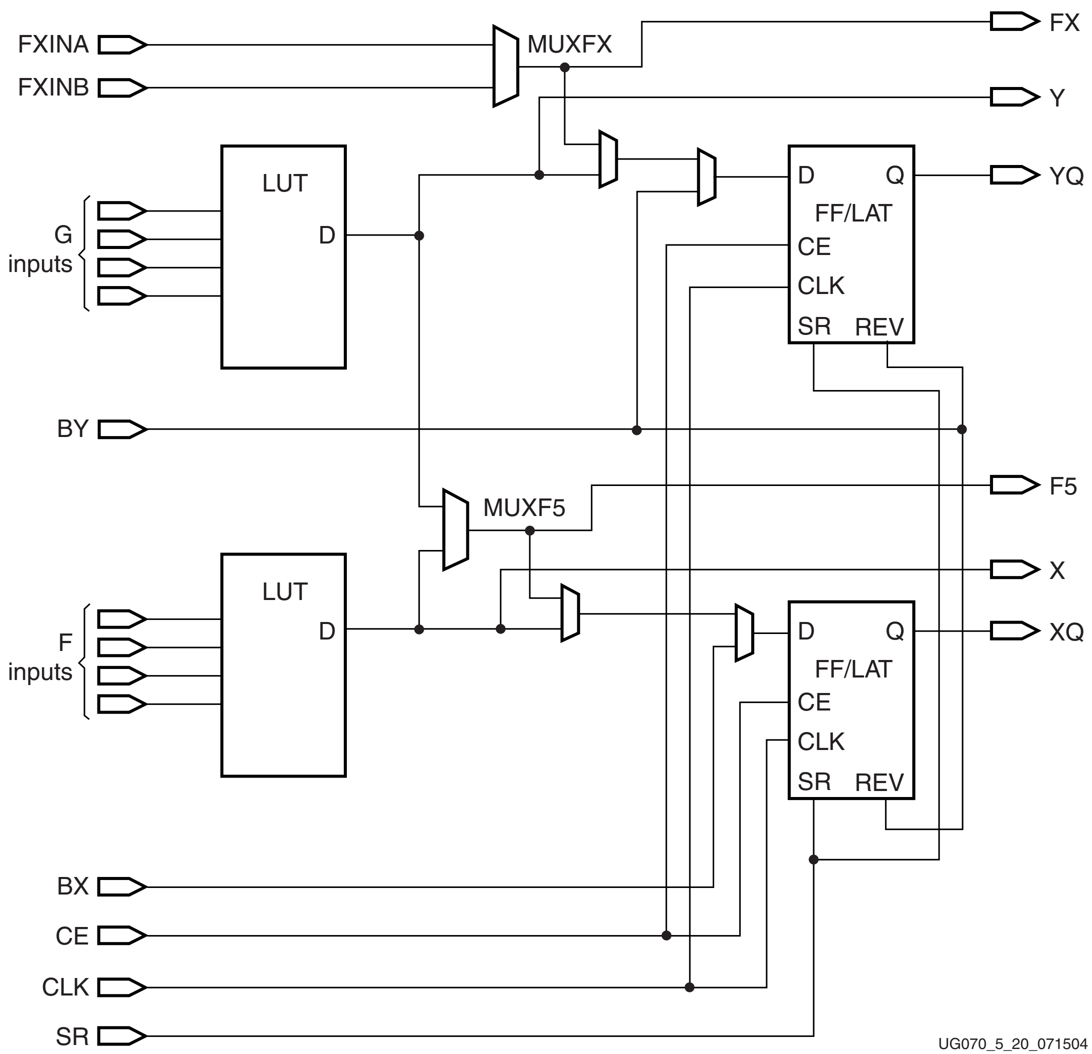
Logic Block (LE, LC, Slice)

- contains a look up table (LUT) with 4 to 6 inputs and a FF. In some FPGAs several Logic Blocks are grouped into clusters with some local routing.

Routing channels

- general purpose
- for global signals like clocks, reset, output enable, with high fanout and low skew

FPGA – Virtex 4 SLICE L/M



Each SLICE contains two LUT4, two FFs and MUXes. The two LUT4 can be combined into one LUT5.

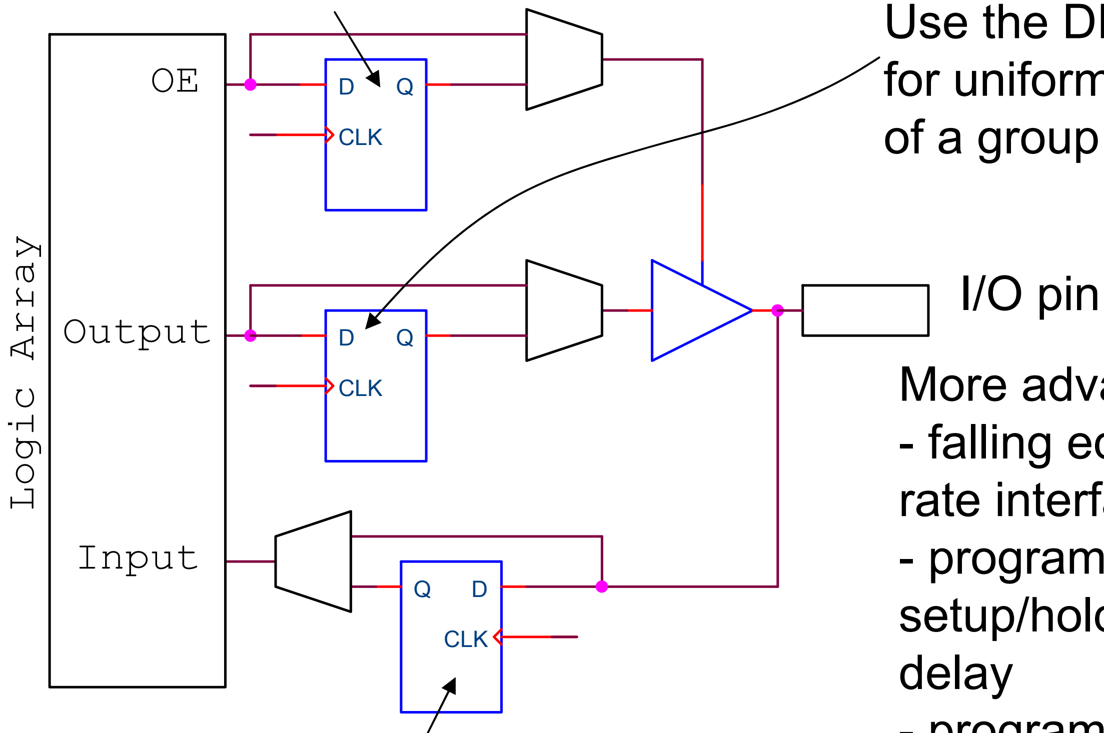
The Configurable Logic Block (CLB) contains 2x SLICEL and 2x SLICEM. The Ms can be used for distributed RAM and large shift registers.

The CLB has 8 LUT4, 8 FFs, can be used for 64 bits distributed RAM or shift register

Simple I/O block

DFF in the OE path to turn simultaneously the direction of a bus

Use the DFF in the output path for uniform clock to output delay of a group of signals.



More advanced features include
- falling edge DFFs for double data rate interfaces
- programmable delays to adjust the setup/hold time or clock to output delay
- programmable pull-up, pull-down, termination resistors or bus keeper
- programmable driver strength

Use the DFF in the input path for uniform and predictable setup/hold times of several I/Os, e.g. a data bus with 32 bits.

Low cost FPGAs overview

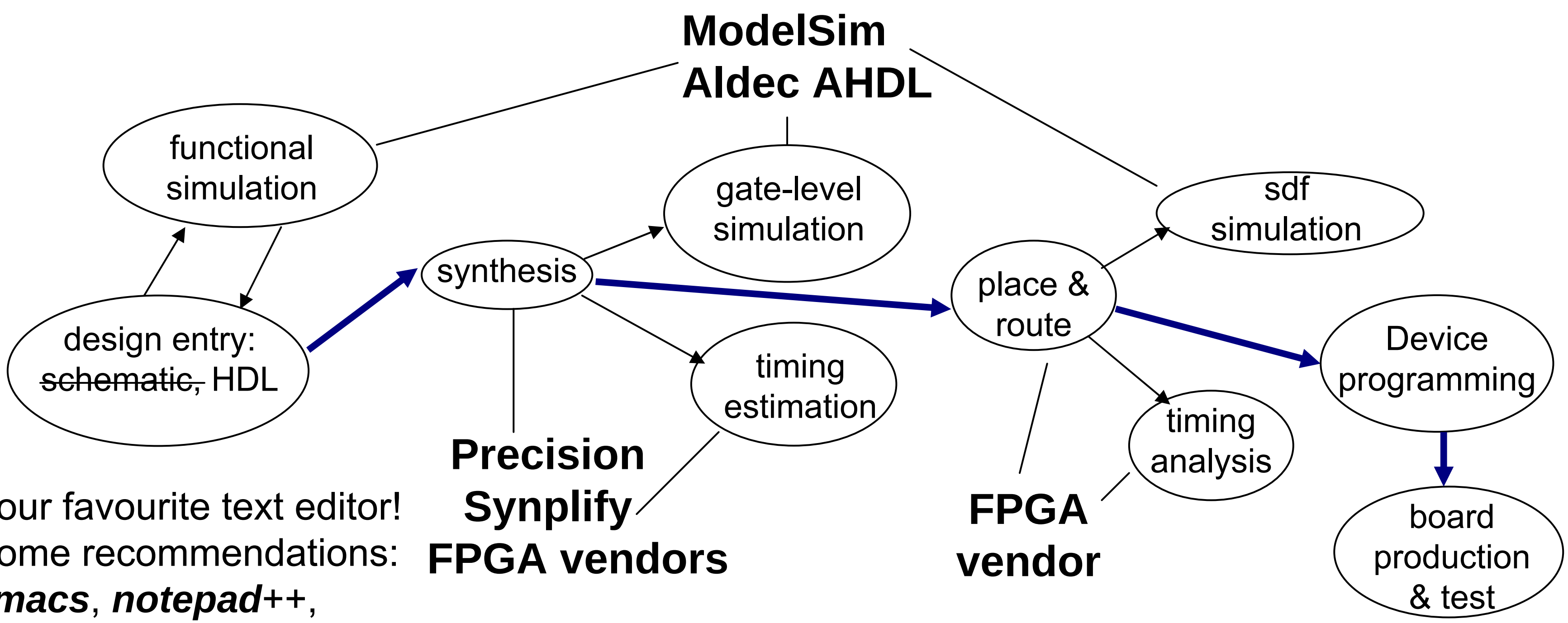
Name	LUT4 (k)	RAM kBits	18x18	PLLs	Tech
Cyclone II	4-68	120-1100	13-150	2-4	90nm
Cyclone III	5-120	400-3800	23-288	2-4	65nm (lp)
Cyclone IV E	6-114	270-3880	15-266	2-4	60nm
Spartan 3E	2-33	72-650	4-36	2-8	90nm
Spartan 3A/AN	2-25	54-576	3-32	2-8	90nm
Spartan 3D	37-53	1500-3200	84-126	8	90nm
Spartan 6 LX	4-147	216-4800	8-180	6	45nm
LatticeEC(ECP)	2-32	18-498	(0-32)	2-4	130nm
LatticeXP	3-20	54-396	---	2-4	130nm
LatticeECP2	6-68	55-1032	12-88	4-8	90nm
LatticeXP2	5-40	166-885	12-32	2-4	90nm

FPGA summary

- The price/logic goes down
- The speed goes up
- Special blocks like RAM, CPU, multiplier…
- Flexible I/O cells, including fast serial links and differential signals
- Infinitely times programmable (with some exceptions)
- External memory or interface for initialization after power up needed – copy protection impossible (with some exceptions)
- More sensitive to radiation, compared to CPLD (with some exceptions)

Manufacturers: **Actel, Altera, Lattice, Xilinx**

Design flow CPLD/FPGA



Your favourite text editor!
Some recommendations:
emacs, ***notepad***++,
nedit, with syntax
colouring and more for
`VHDL` and **`Verilog`**

**Each step can take seconds, minutes, hours ...
(place & route)**

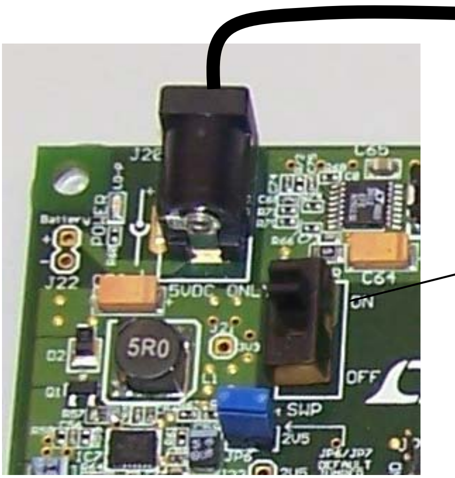
Some practical exercises

- The FPGA board - switches and LEDs
- File structure and first steps with ISE
- Logical unit with simple functions
- 8-bit up/down binary counter
- 8-bit shift register
- Angular decoder

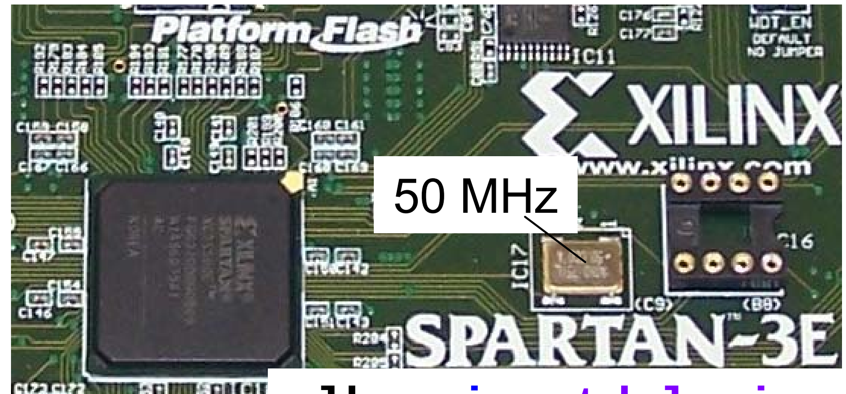
The FPGA board(1)

top.vhd

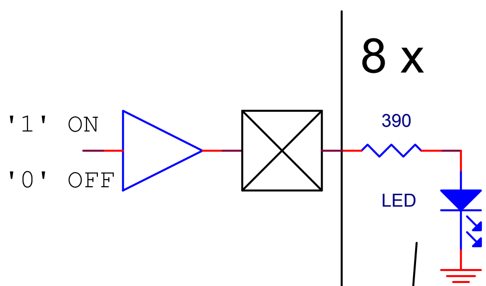
+5 V 2.5 A

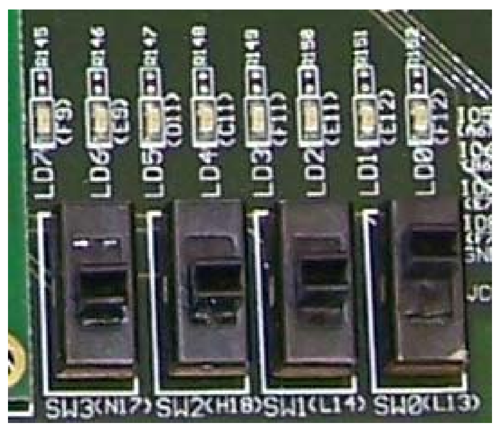


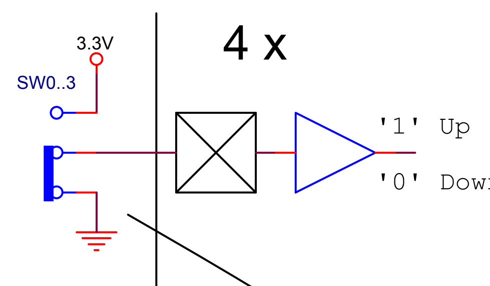
ON/OFF




```
clk : in std_logic;
```





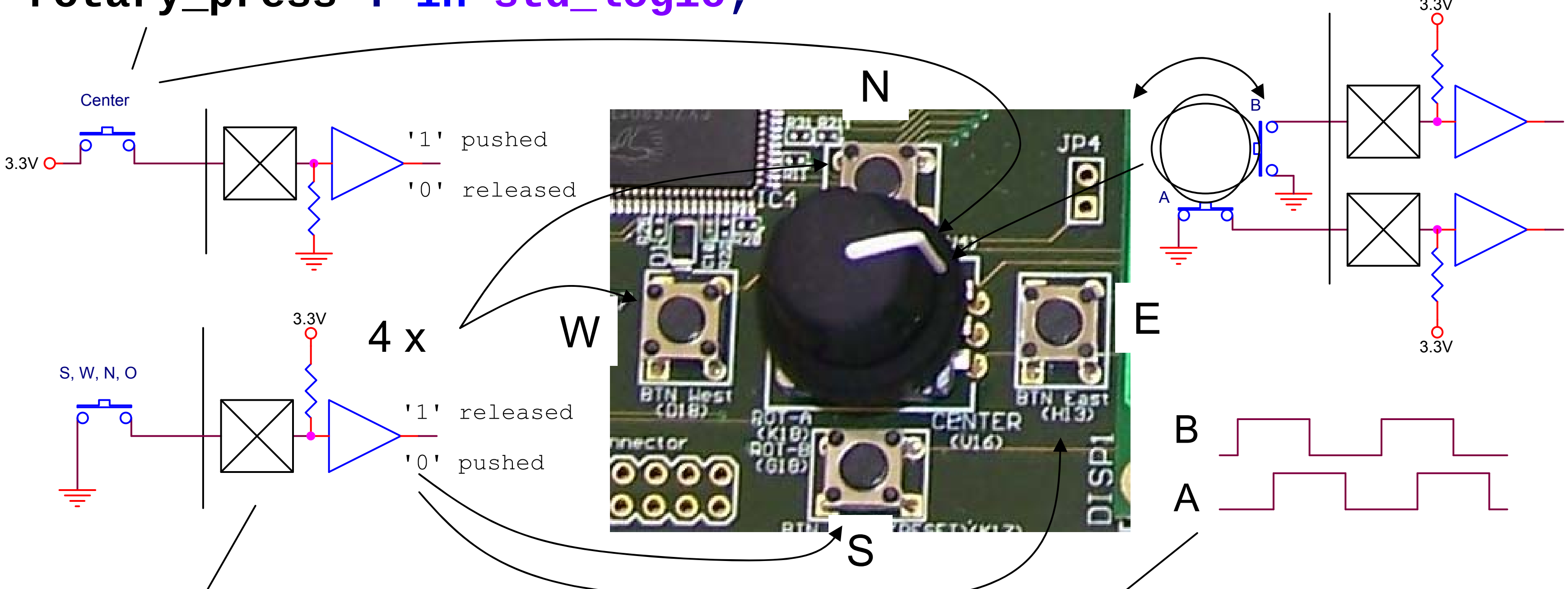



```
led    : out std_logic_vector(7 downto 0);
switch : in  std_logic_vector(3 downto 0);
```

The FPGA board(2)

top.vhd

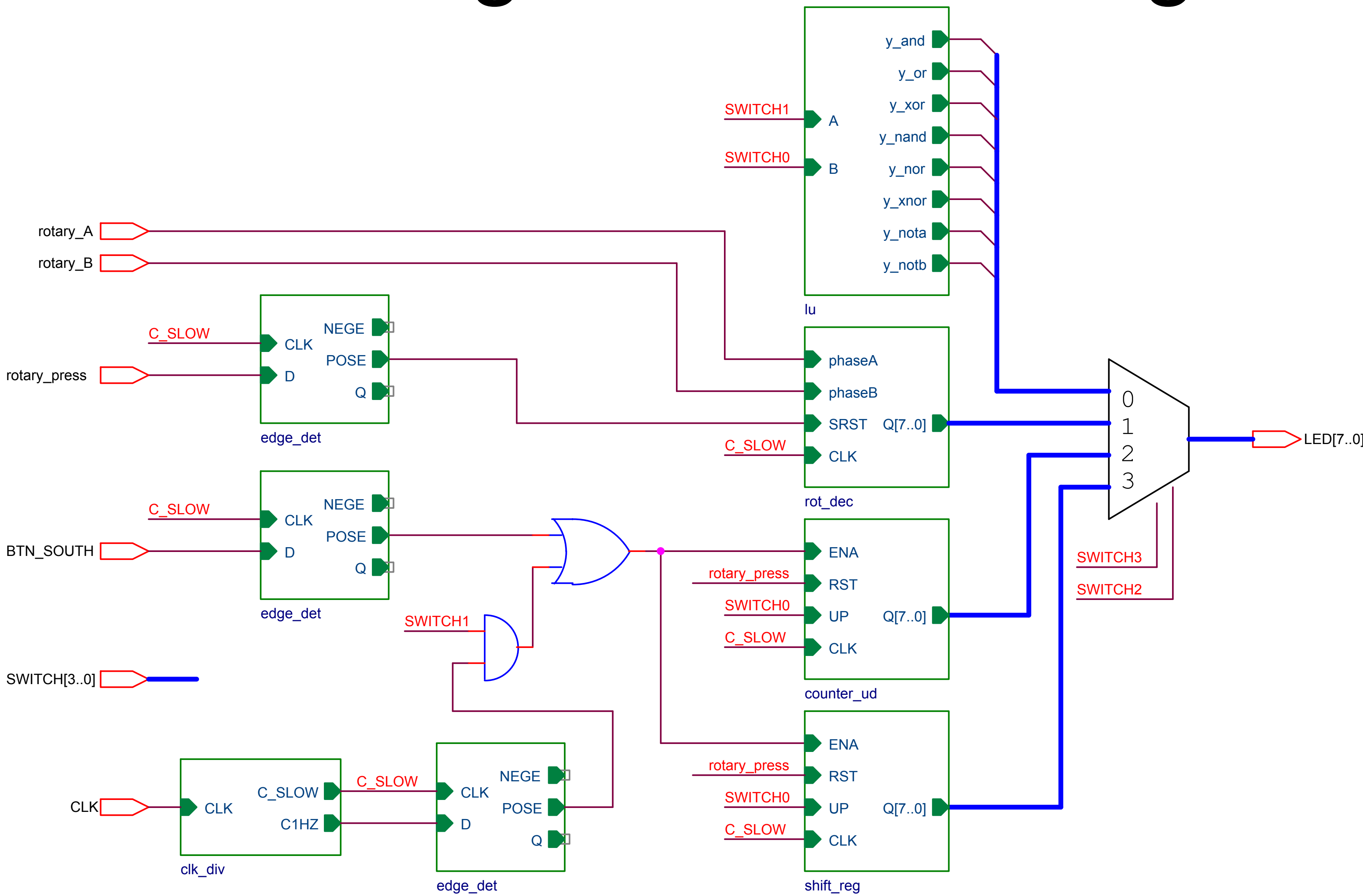
```
rotary_press : in std_logic;
```




```
btn_north : in std_logic;
btn_east  : in std_logic;
btn_south : in std_logic;
btn_west  : in std_logic;
```

```
rotary_a : in std_logic;
rotary_b : in std_logic;
```

Block diagram of our design



File structure

project_name

- **src**
  - **top.vhd, clk_div.vhd, lu.vhd, edge_det.vhd, counter_ud.vhd, shift_reg.vhd, rot_dec.vhd, mux4to1.vhd** — design sources
  - **s3esk_startup.ucf** — pinout & clock definitions
  - **top_tb.vhd** — testbench
- + many other directories created by the ISE software

First steps with ISE

- Create new design
- Add the prepared sources to the design
- Edit the proper source file(s)
- Compile the project
- Program the FPGA on the board and test your design!
- Simulate 1) behaviour; 2) post-route

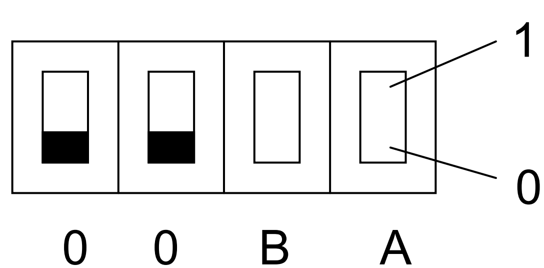
Logical unit with simple logical functions

`lu.vhd`

- Use switch(0) and switch(1) as input
- Calculate 8 functions and put the result on the 8 LEDs:

$\overline{B}$	$\overline{A}$	$\overline{A \text{ xor } B}$	$\overline{A \text{ or } B}$	$\overline{A \text{ and } B}$	A xor B	A or B	A and B

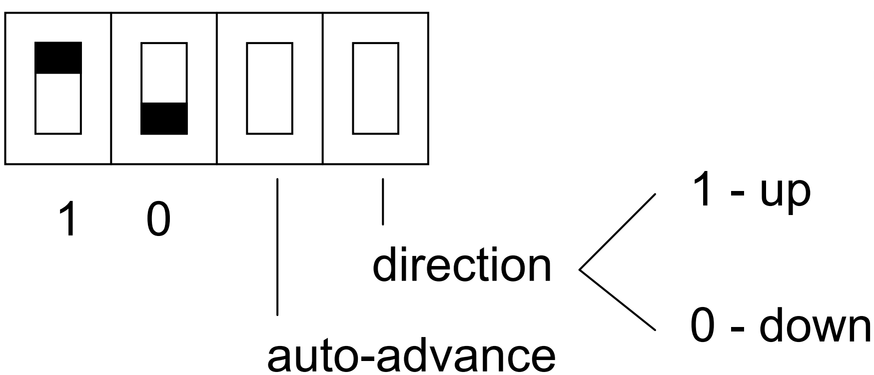
LED7 LED0

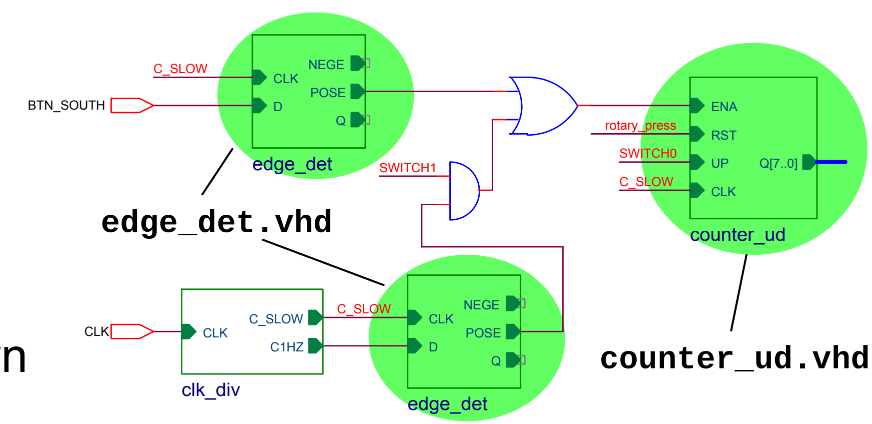


8-bit up/down binary counter

`counter_ud.vhd`
`edge_det.vhd`

- Count the “south” button or internally generated 1-2 Hz clock
- Use the rotary push button as reset
- Use switch(0) as direction (1=up, 0=down)
- Display the counter on the 8 LEDs

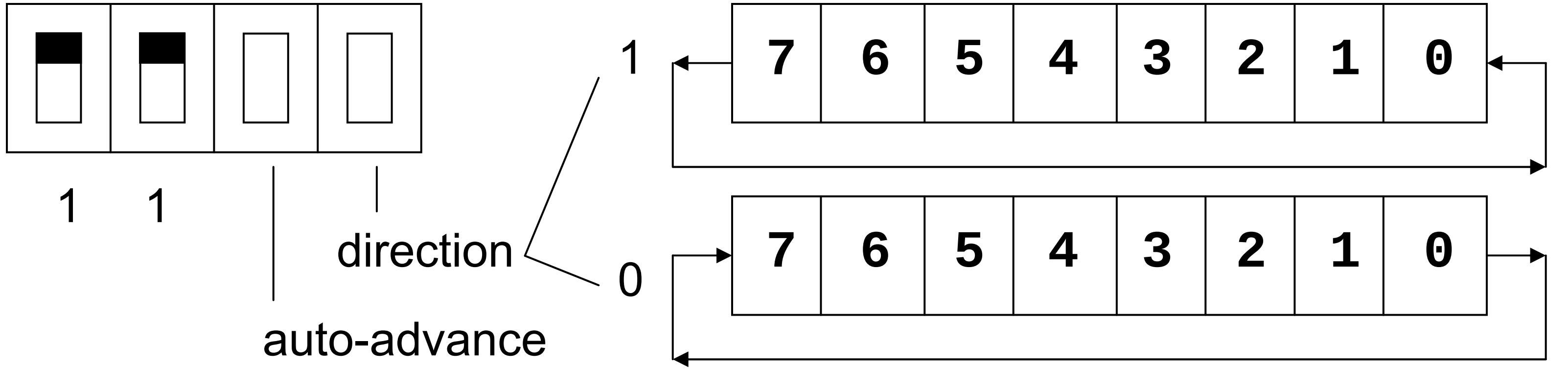




Shift register

`shift_reg.vhd`
`edge_det.vhd`

- Shift left or right, depending on the direction switch(0) when the “south” button pressed or with the internally generated 1-2 Hz clock
- Use the rotary push button as reset – load all bits with 0, except for bit 0 with 1
- Use the LEDs as display



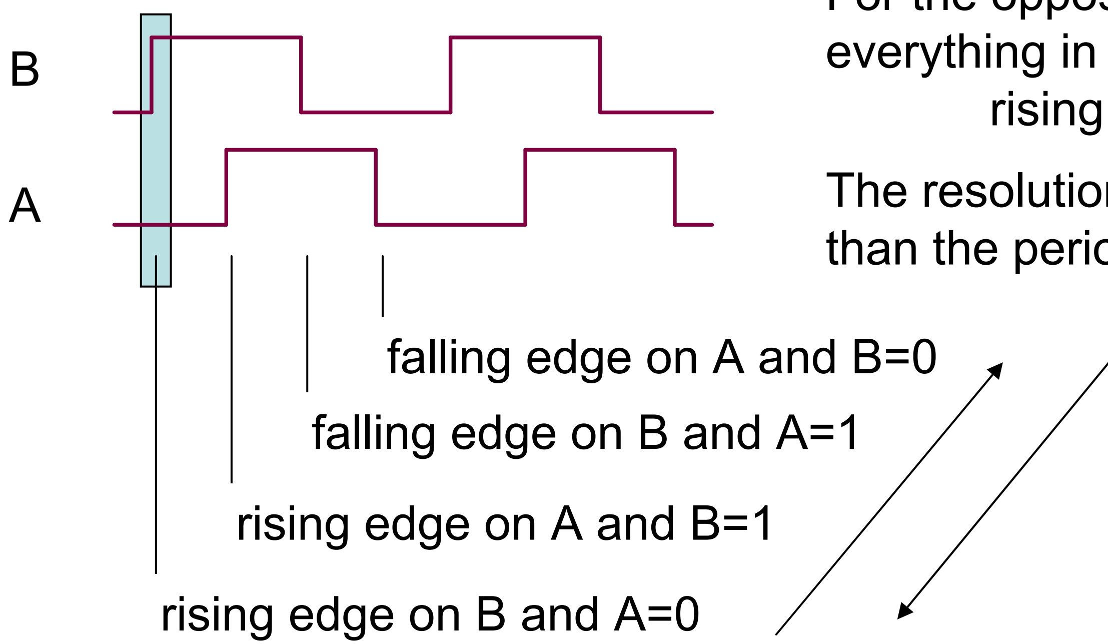
Rotary angular decoder

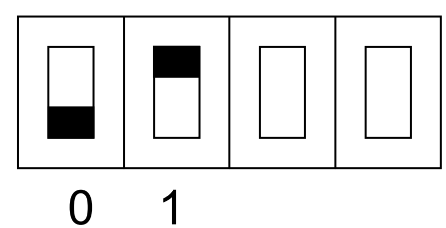
`rot_dec.vhd`
`edge_det.vhd`

- Decode the angular position of the rotary switch by counting up/down the events on the two signals A and B

For the opposite direction reverse everything in time and swap
rising ↔ falling edge

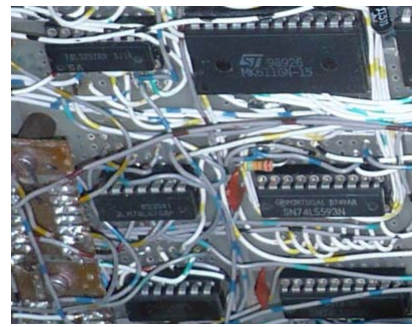
The resolution is 4x better than the period!



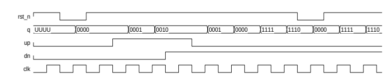


FINISH

- Now you know how it works
- Prototyping yesterday



```
signal qNEW : std_logic;
signal qOLD : std_logic;
begin
process(clk)
 begin
  if rising_edge(clk) then
    qNEW <= INP;
    qOLD <= qNEW
  end if;
 end process;
q <= qNEW;
RE <=     qNEW and not qOLD;
FE <= not qNEW and     qOLD;
```



- … and today with FPGA and HDL
- Many IP cores available – memories, interfacing, CPU cores etc.