

Working with the USB Stick

- contains all needed programs, vhdl code and documentation
- copy **terascale-DWS** folder into **home** (all future paths are relative to this folder)
- open terminal and type **df**
 - **/media/fpga/12345...** should be displayed
- copy string after ...**fpga/**
- open **terascale-DWS/setup.py**
- paste string between **//** at **TS_INSTALL_DIR**
 - **export TS_INSTALL_DIR=/media/fpga/12345.../ubuntu_16**
- go into **terascale-DWS** folder and type **source setup.py**

VHDL WORKSHOP

Philipp Horn

Structure

- Introduction
- Board
- Setup of every .vhd File
- First Example
- Starting a Simulation
- First Exercise
- Applying Design to Hardware
- Designing a Project

Introduction

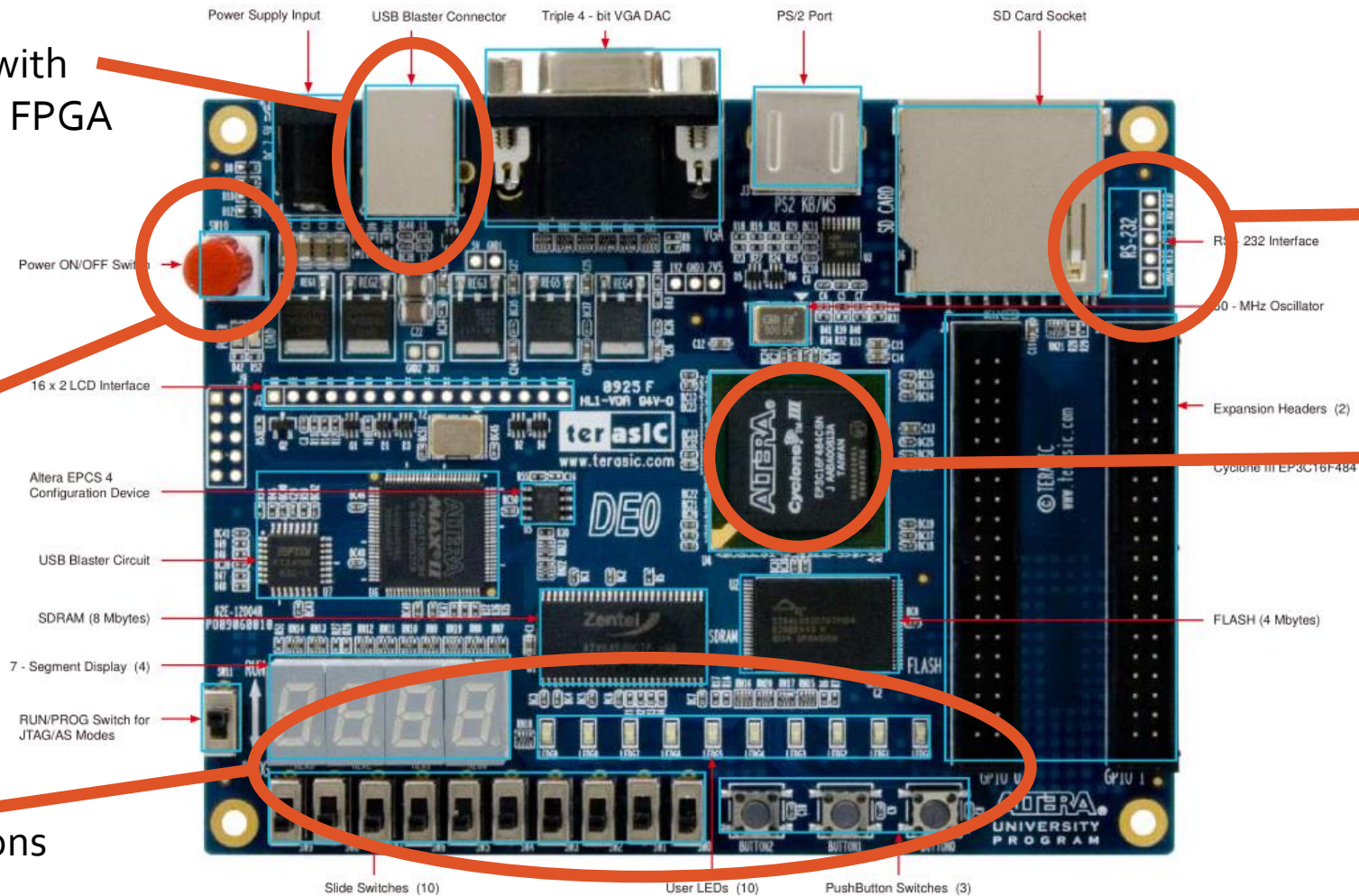
- VHDL
 - **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit **H**ardware **D**escription **L**anguage
 - describes the operation of a logical circuit (e.g. FPGA)
 - concurrent system (unlike procedural computing languages such as C)
- one .vhd text file = one module/entity (piece of hardware) containing
 - interface to outer world (e.g. LEDs, switches, ...)
 - functionality (how input is handled and connected to output)
- simulation program is used to test the logic design (modelsim)
- synthesis program translates text files to “gates and wires” that are mapped on the FPGA
- Quartus provides these programs, a complicated GUI and a lot more
- we will use hdlmake, a powerful tool to manage HDL code and write synthesis and simulation Makefiles

Board

USB connection with computer to load FPGA

ON/OFF switch

LEDs, display, switches and buttons



UART interface
(serial communication
with computer)

FPGA

Setup of every .vhd File

- **green:** comments started with --
- **blue:** keywords for VHDL syntax
- **black:** arbitrary identifiers
- **red:** library
 - Implementing data type **std_logic**
 - has 9 possible values (most important: '0', '1',)
 - single quotation mark is important
- Port:
 - Definition of interface
- Architecture:
 - Implementation of functionality

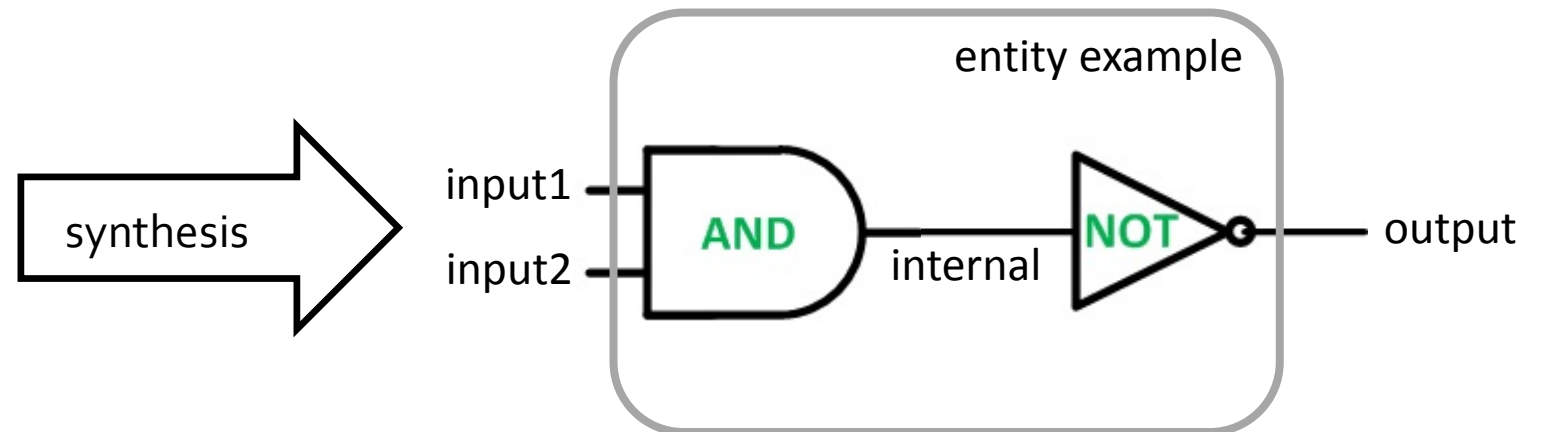
```
library IEEE;  
use IEEE.std_logic_1164.all;  
-- additional libraries are included here  
  
entity entity_label is  
    port (  
        -- input and output are defined here  
    );  
end entity_label;  
  
architecture arch_label of entity_label is  
    -- internal signals are declared here  
begin  
    -- functionality of the module is described here  
end arch_label;
```

First Example – NAND Gate

```
library IEEE;  
use IEEE.std_logic_1164.all;  
  
entity example is  
  port (  
    input1 : in std_logic;  
    input2 : in std_logic;  
    output : out std_logic := '0';  
  );  
end entity;  
  
architecture arch of example is  
  signal internal : std_logic := '0';  
begin  
  internal <= input1 and input2;  
  output <= not internal;  
end arch;
```

- Ports:
 - define two input and one output signals
 - possible to give default value

- Architecture:
 - declare one internal signal
 - Assigning values to signals and output ports
 - order is not important



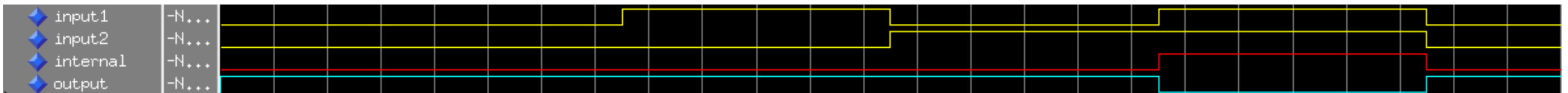
Starting a Simulation

- test logic design using simulation models (test bench)
- they control input ports and can check output ports
- test benches are provided for this workshop
- procedure for every simulation:
 - go to **firmware/sim/modelsim/example**
 - type **hdlmake** (uses Manifest.py files to collect necessary files to setup simulation and writes a Makefile)
 - type **make**

```
library IEEE;  
use IEEE.std_logic_1164.all;
```

```
entity example is  
  port (  
    input1 : in std_logic;  
    input2 : in std_logic;  
    output : out std_logic := '0'  
  );  
end entity;
```

```
architecture arch of example is  
  signal internal : std_logic := '0';  
begin  
  internal <= input1 and input2;  
  output <= not internal;  
end arch;
```



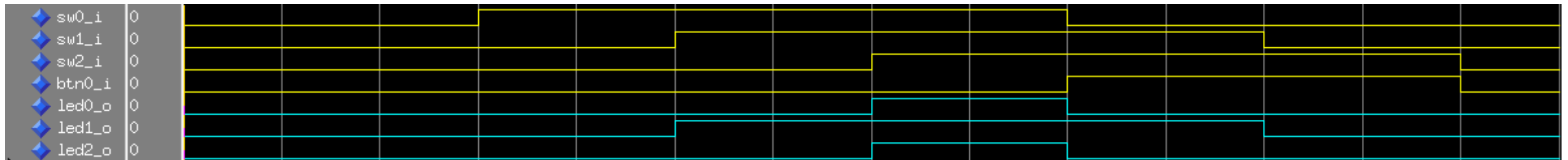
1. Exercise – Logic Gates

- open **firmware/modules/gates/gates.vhd** with a text editor
- implement following functionality:
 - switch 0, 1 and 2 are on → LED 0 is on
 - at least two of the switches 0, 1 and 2 are on → LED 1 is on
 - switch 2 is on and button 0 is not pressed → LED 2 is on
- simulation in **firmware/sim/modelsim/gates**
- use parenthesis to control priority:

- possible logic gates:

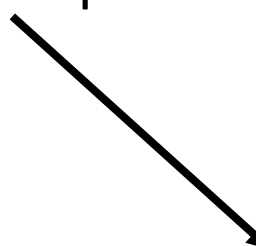
not, and, or, nand
nor, xor, xnor

```
architecture arch of example is
begin
    output <= not (input1 or input2);
end arch;
```



Applying the Design to Hardware

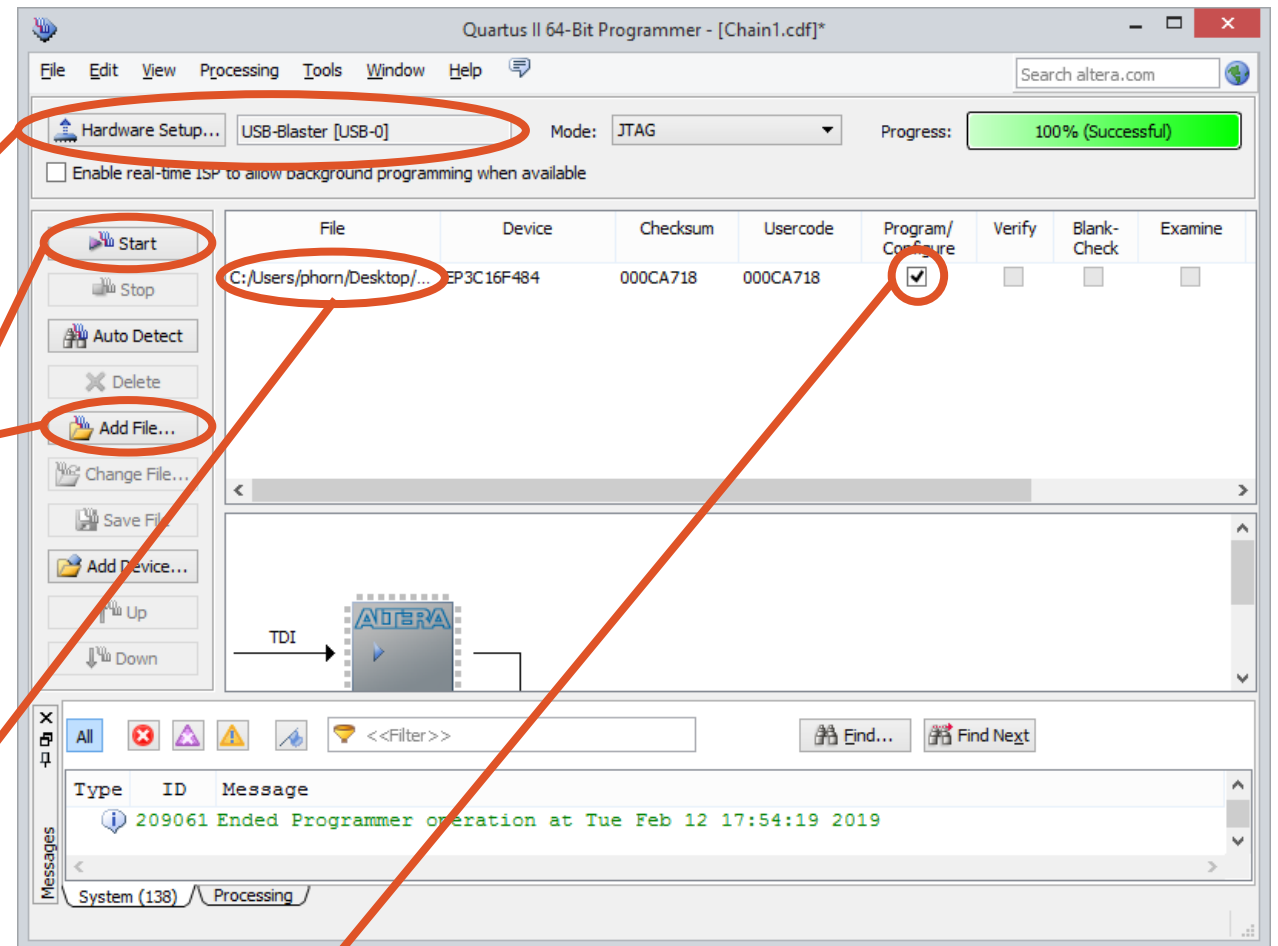
- go to **firmware/syn/de0_quartus_gates**
- **pinout.tcl**
 - script, which maps ports to pins of FPGA
 - these pins are connected to other components of the board
- type **hdlmake** (writes Makefile)
- type **make**
- **demo.sof** is generated
- regenerate file:
 - type **make clean**
 - type **make**



```
set_location_assignment PIN_H2 -to btn0_i  
set_location_assignment PIN_J6 -to sw0_i  
set_location_assignment PIN_H5 -to sw1_i  
set_location_assignment PIN_H6 -to sw2_i
```

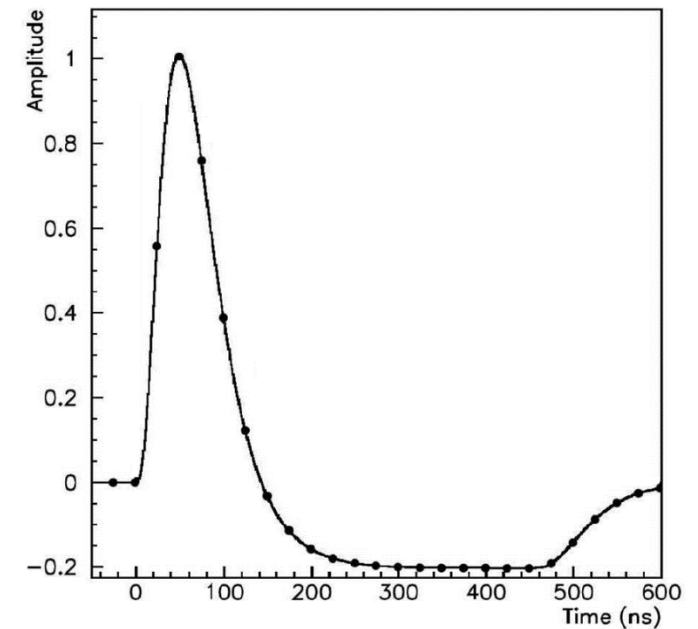
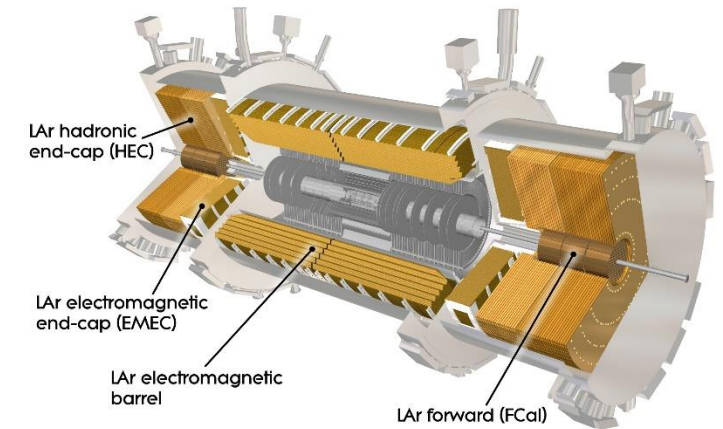
Applying the Design to Hardware

- Quartus Programmer uploads .sof files to the FPGA
 - turn board on
 - type **quartus_pgmw**
 - hardware set to USB-Blaster
 - click **Add File ...**
 - choose demo.sof
 - click **Start**
- after regenerate:
 - double-click and choose file
 - this should be ticked

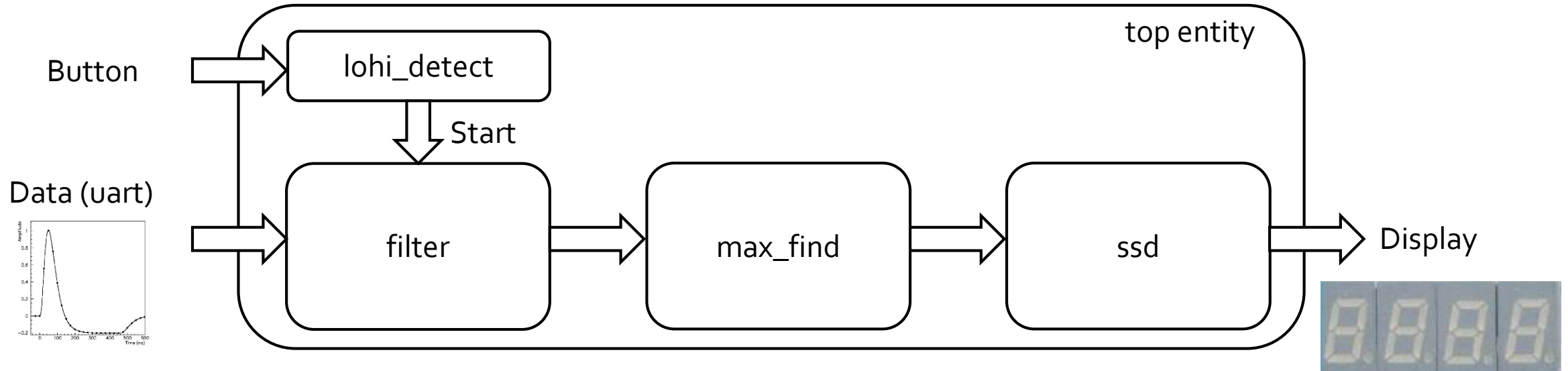


Data Processing

- ATLAS calorimeter at CERN measures energy of particles
- interaction between particle and calorimeter generates electric pulse with a amplitude corresponding to energy
- analog pulse is digitized
- ADC samples are filtered to reduce noise
- identify maximum to calculate energy of particle
- 40 MHz data rate on 182,468 readout channels
 - fast concurrent system needed



Designing a Project – Data Processing



- project is divided in several entities/modules inside the “top” entity
- each module will be developed and simulated separately
 - lohi_detect: detects rising edge of button and generates start signal
 - filter: process data stream upon receiving start signal
 - max_find: detect maximum value
 - ssd: prepare value for seven segment display

Process

```
process_label: process(sensitivity list)
begin
    -- sequential statements
end process process_label;
```

- operates procedural and are preceded by event control
- process is executed every time a signal in **sensitivity list** changes
- multiple assignments of same signal possible
 - signal adopts value of final assignment
- if condition similar to procedural computing languages

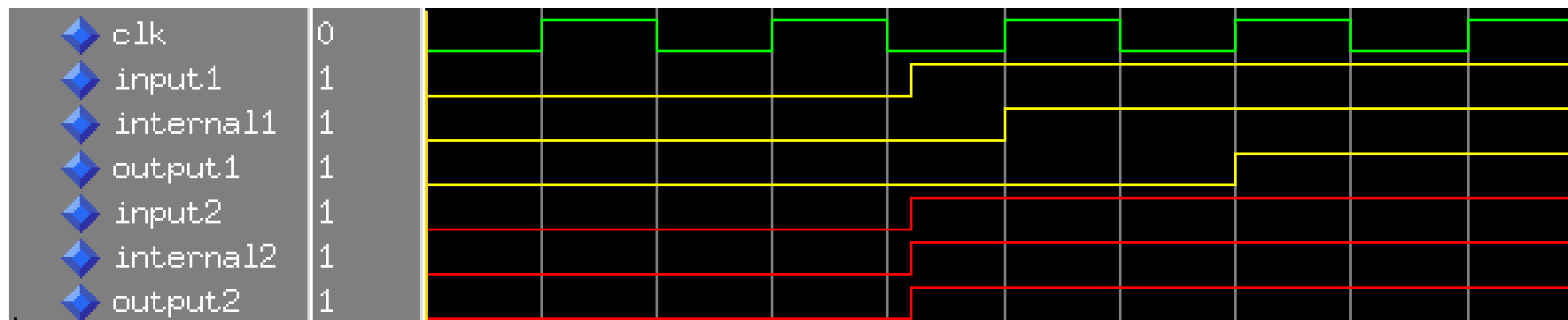
```
if cond1 then
    -- sequential statements
elsif cond2 then
    -- sequential statements
else
    -- sequential statements
end if;
```

```
process_label: process(clk)
begin
    if rising_edge(clk) then
        -- sequential statements
    end if;
end process process_label;
```

- usually process is used with a clock (**clk**)
 - signal, which alternates between '0' and '1' with a fixed frequency
- sequential statements of the left process are always executes, when the clock switches from '0' to '1'

Process - Example

- value of signal inside a process is adopted at the end of the process
- inside of process:
 - internal1 is asserted one clock cycles after input1
 - output1 is asserted two clock cycles after input1
- outside of process:
 - internal2 and output2 are asserted at the same time as input2



```
entity example2 is
port (
  clk : in std_logic;
  input1 : in std_logic;
  input2 : in std_logic;
  output1 : out std_logic := '0';
  output2 : out std_logic := '0'
);
end entity;
```

```
architecture arch of example2 is
  signal internal1 : std_logic := '0';
  signal internal2 : std_logic := '0';
begin
```

```
  process_label: process(clk)
  begin
    if rising_edge(clk) then
      internal1 <= input1;
      output1 <= internal1;
    end if;
  end process process_label;
```

```
  internal2 <= input2;
  output2 <= internal2;
```

```
end arch;
```

1. Module *LoHi Detect*

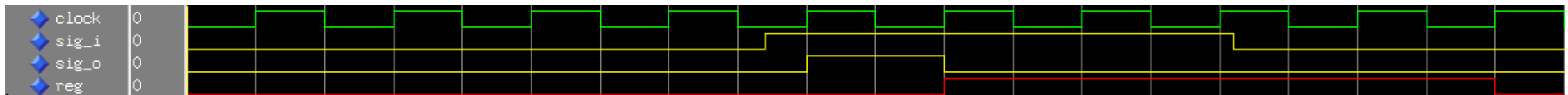
- open **firmware/modules/lohi_detect/lohi_detect.vhd**
- implement following functionality:
 - synchronize input **sig_i**
 - **sig_o** asserts for a single clock cycle after a rising edge of **sig_i**
- simulation in **firmware/sim/modelsim/lohi_detect**
- hint: internal signal **reg** is needed
- process statement with clock:

```
architecture arch of lohi_detect is
    signal reg : std_logic := '0';
    -- declare additional signals
begin

    process_label: process(clk)
    begin
        if rising_edge(clk) then
            -- write your code here
        end if;
    end process process_label;

    -- and here

end arch;
```



Additional Data Types

- **std_logic_vector:**

- array of std_logics
- default value with **others** to be length independent

```
signal A1 : std_logic_vector(6 downto 0) := (others => '0');
```

- **unsigned / signed:**

- similar to std_logic_vector
- mathematical operations are possible + - * / **
- needs additional library IEEE.numeric_std

```
signal B1 : unsigned(2 downto 0) := "110"; -- equals 6 = 4 + 2
signal B2 : signed(2 downto 0) := "110";   -- equals -2 = -4 + 2
```

- **integer / natural / positive:**

- **natural** contains zero
- use **range** to limit number of bits (default = 32)

```
signal C1 : integer := -2;
signal C2 : natural := 0;
signal C3 : integer range 5 to 100 := 7;
```

- **boolean**

- possible values: true and false
- used with any of the relational operators < > <= >= = /=

```
signal D1 : boolean := false;
```

```
if A1 = "0100110" then
if C1 <= B2 then
```

Generics

- optional possibility to pass specific information to the entity
- common usage: define number of bits for port vectors
- change default value of **bit_width** to change length of **data_i** and **data_o**
- vector with 16 bit has range (15 downto 0)
 - → **bit_width-1**

```
entity max_find is
  generic (
    bit_width : positive := 16
  );
  port (
    data_i : in unsigned(bit_width-1 downto 0);
    data_o : out unsigned(bit_width-1 downto 0) := (others => '0')
  );
end max_find;
```

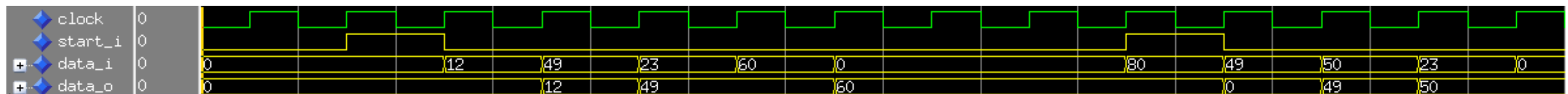
2. Module “Max Find”

- open **firmware/modules/max_find/max_find.vhd**
- implement following functionality:
 - **data_o** is maximum value of all previous **data_i**
 - **start_i** resets this maximum (has priority)
- simulation in **firmware/sim/modelsim/max_find**
- internal signal declaration necessary, because reading of output not possible:

• if condition:

```
if cond1 then
    -- sequential statements
elsif cond2 then
    -- sequential statements
else
    -- sequential statements
end if;
```

```
signal B : unsigned(bit_width-1 downto 0) := (others => '0');
```



Data Type Conversion

- Between unsigned/signed and std_logic_vector:
 - signals need to have the same width

```
A_std <= std_logic_vector(B_sig);  
A_std <= std_logic_vector(C_uns);  
  
B_sig <= signed(A_std);  
C_uns <= unsigned(A_std);
```

- Between unsigned/signed and integer:
 - specification of the intended bit width from integer required

```
D_int <= to_integer(B_sig);  
D_int <= to_integer(C_uns);  
  
B_sig <= to_signed(D_int,bit_width);  
C_uns <= to_unsigned(D_int,bit_width);
```

- not possible to convert directly between integer and std_logic_vector
 - first convert to signed or unsigned

Type Declaration

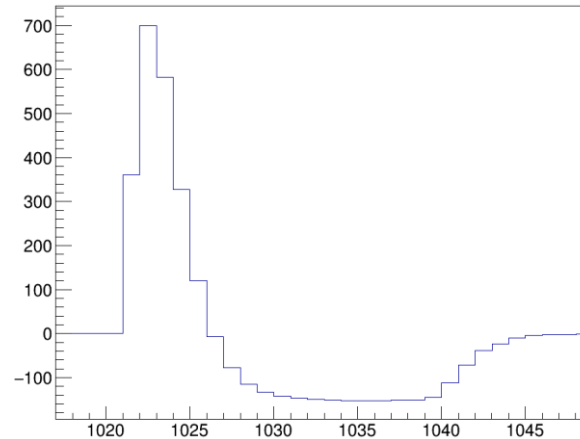
- examples of predefined types:

```
type boolean is (false, true);  
type std_logic_vector is array (natural range <>) of std_logic;
```

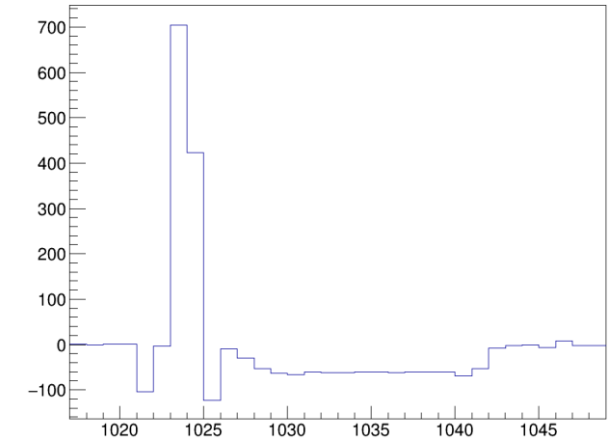
- **Boolean** is an enumerate of length two
 - **std_logic_vector** is array of **std_logic** with undefined length
- define a matrix:
 - written between **architecture** and **begin**
 - type **matrix_t** is array of std_logic_vectors with undefined length
 - signal **matrix_s** is array of std_logic_vectors with length 4

```
architecture arch of entity_label is  
    type matrix_t is array (natural range <>) of std_logic_vector(6 downto 0);  
    signal matrix_s : matrix_t(3 downto 0);  
begin
```

Wiener Filter



filter

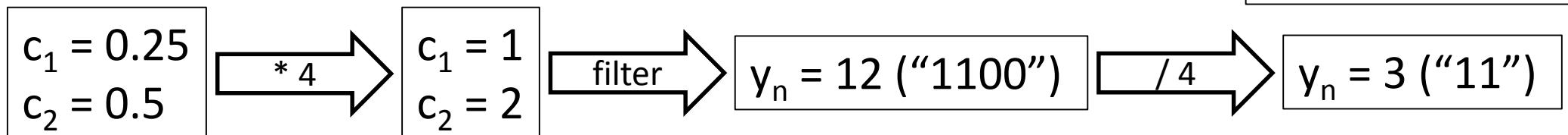


- **output** y_n is a weighted sum of the most recent 6 **input** values (x_n, x_{n-1}, \dots)
- constants (c_1, c_2, \dots) are calibrated on input and desired output

$$y_n = c_1 * x_n + c_2 * x_{n-1} + c_3 * x_{n-2} + c_4 * x_{n-3} + c_5 * x_{n-4} + c_6 * x_{n-5}$$

- constants are usually not integers \rightarrow multiply them by a common factor ($\times 2^k$)
- result has to be divided by that common factor (cut lower k bits)
- example: constants multiplied by 4 and lower 2 bits of result cut

$$y_n = c_1 * 6 + c_2 * 3$$



- type conversion:

3. Module “filter”

```
D_int <= to_integer(C_uns);
C_uns <= to_unsigned(D_int,bit_width);
```

- open **firmware/modules/filter/filter.vhd** with a text editor

$$y_n = c_1 * x_n + c_2 * x_{n-1} + c_3 * x_{n-2} + c_4 * x_{n-3} + c_5 * x_{n-4} + c_6 * x_{n-5}$$

- implement following functionality:
 - **data_o** (y_n) is a weighted sum of the most recent 6 **data_i** values (x_n, x_{n-1}, \dots)
 - use process to buffer x_{n-1}, x_{n-2}, \dots
 - convert input to integer and result to unsigned **data_uns**
- simulation in **firmware/sim/modelsim/filter**
- usage of type definition and loop possible



Conditional Signal Assignment

- outside of processes
- when / else:
 - value is assigned to **output1** based on conditions
 - more general method (any boolean expression possible)
 - counterpart to “if condition” in process
 - **input1/2/3** can be signals or values
 - **cond1** and **cond2** are bools
- with / select:
 - value is assigned to **output2** based on value of **internal**
 - rather specific method (equality checking)
 - **A, B, C, choice1** and **choice2** can be signals or values
 - value is assigned to **output2** based on possible values of **selection**

```
output1 <= input1 when cond1 else  
            input2 when cond2 else  
            input3;
```

```
with internal select  
output2 <= input1 when choice1,  
            input2 when choice2,  
            input3 when others;
```

“single disp”

- **firmware/modules/ssd/single_disp.vhd** shown on the right
- example for conditional signal assignment
- converts the single digit **number_i** (0-9) to the seven segment display vector **seg_o**
 - if **number_i** = 8 → all bits of **seg_o** needs to be active
 - **number_i** > 9 results only in the assertion of the middle segment
- hardware might require active low signals
 - ‘0’ means light is on
 - boolean decides if the signal is inverted



```
entity single_disp is
  generic(
    invert : boolean := false
  );
  port(
    number_i : in unsigned(3 downto 0);
    seg_o    : out std_logic_vector(6 downto 0)
  );
end entity;

architecture arch of single_disp is
  signal seg_s : std_logic_vector(6 downto 0);
begin

  with number_i select
    seg_s <= "0111111" when "0000", -- 0
             "0000110" when "0001", -- 1
             "1011011" when "0010", -- 2
             "1001111" when "0011", -- 3
             "1100110" when "0100", -- 4
             "1101101" when "0101", -- 5
             "1111101" when "0110", -- 6
             "0000111" when "0111", -- 7
             "1111111" when "1000", -- 8
             "1101111" when "1001", -- 9
             "1000000" when others;

  seg_o <= not seg_s when invert else
           seg_s;

end arch;
```

Mapping of entities

- reuse an entity as a module in a different entity

```
architecture arch of seconds is
  signal internal : std_logic_vector(7 downto 0);
begin

  map_label: entity work.single_disp
  generic map(
    invert => true
  )
  port map(
    number_i => "1000",
    seg_o    => internal(6 downto 0)
  );

end arch;
```

```
entity single_disp is
  generic(
    invert : boolean := false
  );
  port(
    number_i : in  unsigned(3 downto 0);
    seg_o    : out std_logic_vector(6 downto 0)
  );
end entity;
```

- **single_disp** entity (generic and ports on the top) will be needed for next module **seconds** (architecture on the left)
- **work** denotes the current working library
- generics and ports from the entity on the left
- connected values and signals on the right
- generics can be omitted (the default value would be used)
- widths need to align (shorten **internal**)

2. Exercise – Second Counter

- open **firmware/modules/ssd/seconds.vhd** with a text editor
- implement following functionality:
 - **counter** is increased with every rising edge of clock **clk**
 - when **counter** reaches **counter_max** one second passed
 - for simulation **counter_max** is set to 5
 - output seconds **sec** to seven segment display **ss_d1_o**
 - after nine seconds: start again at zero
- simulation in **firmware/sim/modelsim/seconds**
- synthesis in **firmware/syn/de0_quartus_seconds**

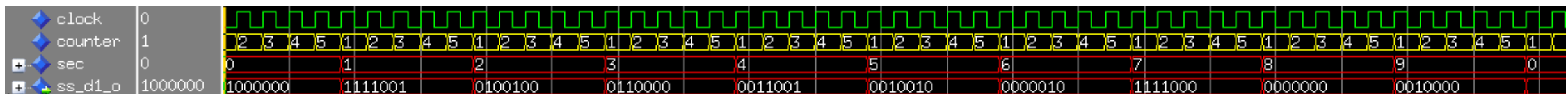
- mapping:

```

architecture arch of seconds is
begin

    map_label: entity work.single_disp
    generic map(
        invert => invert
    )
    port map(
        number_i => B,
        seg_o    => C
    );

end arch;
    
```



Generate

```
generate_label: if cond1 generate  
    output <= input;  
end generate;
```

- **if generate**: conditional creating of components
- **for generate**: repeating a group of identical components
- example: assign first 5 bits of every **std_logic_vector** of **matrix_s**

```
architecture arch of entity_label is  
    signal internal : std_logic_vector(4 downto 0) := (others => '0');  
    type matrix_t is array (natural range <>) of std_logic_vector(6 downto 0);  
    signal matrix_s : matrix_t(2 downto 0);  
begin  
    matrix_s(0)(4 downto 0) <= internal;  
    matrix_s(1)(4 downto 0) <= internal;  
    matrix_s(2)(4 downto 0) <= internal;  
end arch;
```

- combined to generate statement:

```
generate_label: for i in 2 downto 0 generate  
    matrix_s(i)(4 downto 0) <= internal;  
end generate;
```

4. Module “ssd”

- open **firmware/modules/ssd/ssd.vhd** with a text editor
- implement following functionality:
 - four digit number input **data_i**
 - use division **/** and modulo **mod** to separate digits
 - save all results in matrices **quotient** and **remainder**
 - route separate digits to **single_disp** to receive **ss_ds**
 - output four **std_logic_vectors** (seven segment display)
- simulation in **firmware/sim/modelsim/ssd**
- generate:

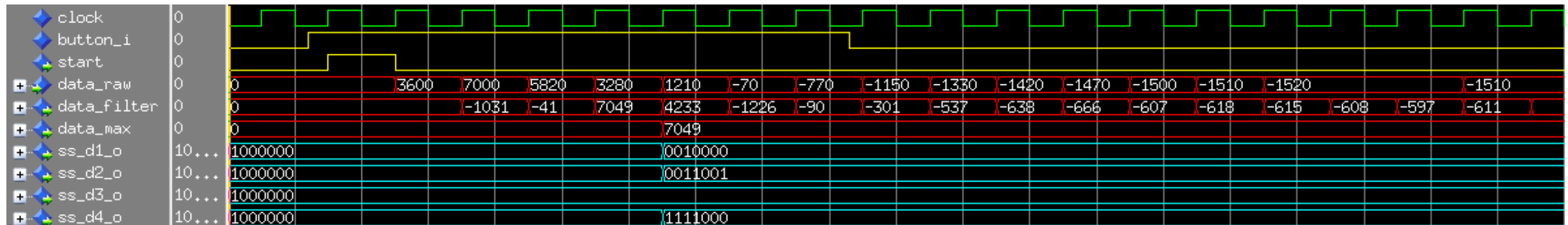
```
generate_label: for i in 3 downto 0 generate
  matrix_s(i)(4 downto 0) <= internal;
end generate;
```

Signal	Value
clock	0
data_i	0
quotient (4)	0
quotient (3)	0
quotient (2)	0
quotient (1)	0
quotient (0)	0
remainder (3)	0
remainder (2)	0
remainder (1)	0
remainder (0)	0
ss_d4_o	1000000 1111000 10...
ss_d3_o	1000000 0110000 10...
ss_d2_o	1000000 0010010 10...
ss_d1_o	1000000 0100100 10...

5. Module “top”

- solution of previous modules:
 - firmware/modules/.top/
 - copy and overwrite self written .vhd file

- open **firmware/modules/top/top.vhd** with a text editor
- implement following functionality
 - entity **data_uart** provides data (in simulation only a constant set)
 - map all 4 modules into this entity and use internal signals to connect them
- simulation in **firmware/sim/modelsim/top**
- synthesis in **firmware/syn/de0_quartus_top**



Sending data to FPGA

- go into **terascale-DWS** folder
- type **python/slowCtrl.py**
- choose displayed input file (type **1**)
- choose displayed UART port (type **5**)
- send pulses by typing **6**

```
(1) select ADC data input file [/home/fpga/terascale-DWS/python/adcData/ADC_samples.txt]
(2) set scaling factor [1.0]
(3) set single value index [0]
(4) plot ADC samples

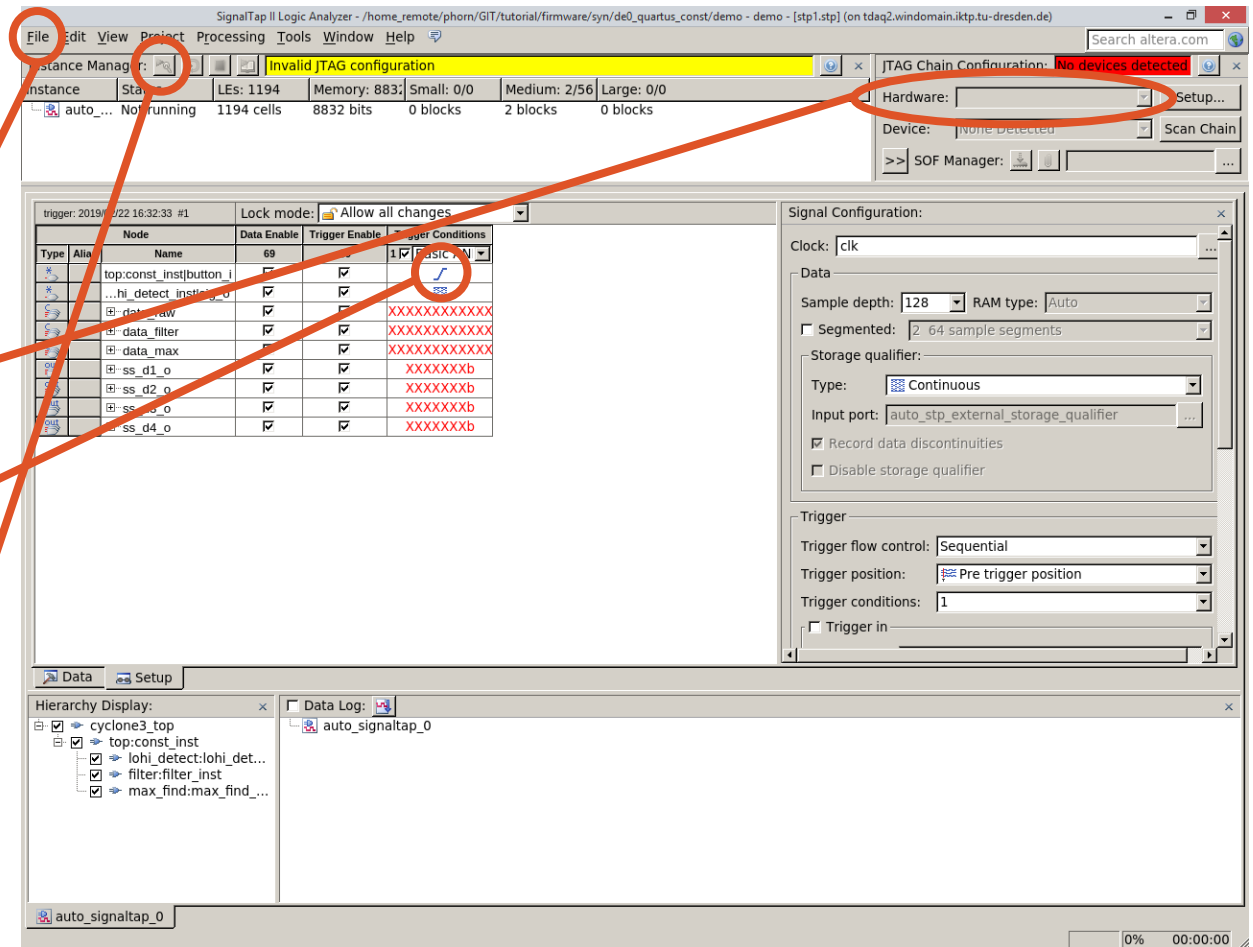
(5) open UART port [/dev/ttyS0]
(6) send pulse to fpga
(7) send single value to fpga

(0) exit

Enter option [5]: █
```

Signal Tap

- powerful debugging tool, which enables to look inside FPGA
 - type **quartus_stpw**
 - open file **stp1.stp**
 - set hardware to USB Blaster
 - trigger is already set to rising edge of button
 - arm trigger
 - push button on board



Writing a Test Bench

- test bench (**tb**) = entity used for simulation
 - example: **seconds_tb** on the right
- generics definition but without port
 - example: set **counter_max** = 5 to shorten simulation
- maps unit under test (uut) with internal signals
- output is not connected (**open**)
- clock (**clk**) generation via process
- **wait for** statement not synthesizable

```
library IEEE;
use IEEE.std_logic_1164.all;

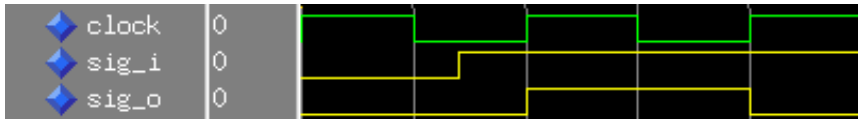
entity seconds_tb is
  generic(
    counter_max : positive := 5;
    invert      : boolean := true
  );
end seconds_tb;

architecture tb of seconds_tb is
  signal clk : std_logic := '0';
begin
  uut: entity work.seconds
  generic map(
    counter_max => counter_max,
    invert      => invert
  )
  port map(
    clk      => clk,
    ss_d1_o => open
  );

  clk_proc: process
  begin
    clk <= '0'; -- clock cycle is 20 ns
    wait for 10 ns;
    clk <= '1';
    wait for 10 ns;
  end process clk_proc;
end architecture;
```

Input generation

- additional possibilities written in test bench architecture
- asynchronous signal:
 - from **lohi_detect**



```
sim_proc: process
begin
  wait for 42 ns;
  sig_i <= '1';
  wait for 34 ns;
  sig_i <= '0';
  wait;
end process sim_proc;
```

- generate synchronous **counter**
- use **case** statement to assign input
 - from **example1**
- signal configuration done in modelsim GUI (display format, names and colors) and saved in **wave.do**

```
count_proc: process(clk)
begin
  if rising_edge(clk) then
    counter <= counter + 1;
  end if;
end process count_proc;

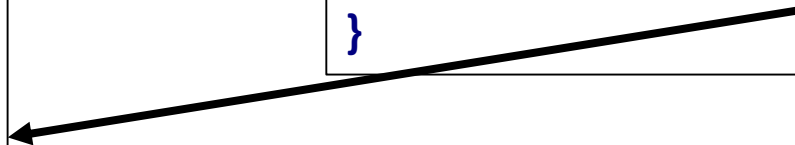
sim_proc: process(clk)
begin
  if rising_edge(clk) then
    case counter is
      when 1 =>
        input1 <= '1';
        input2 <= '0';
      when 2 =>
        input1 <= '0';
        input2 <= '1';
      when 3 =>
        input1 <= '1';
        input2 <= '1';
      when others =>
        input1 <= '0';
        input2 <= '0';
    end case;
  end if;
end process sim_proc;
```

hdlmake

- file **firmware/sim/modelsim/top/Manifest.py** is starting point for **top** simulation
- **sim_post_cmd** is a command that is issued after the simulation process has finished
- modules points to other **Manifest.py** files in local folders
- add needed files for simulation

```
files = [  
    "top_tb.vhd",  
]  
modules = {  
    "local" : [ ".././../modules/top" ],  
}
```


```
action = "simulation"  
sim_tool = "modelsim"  
sim_top = "top_tb"  
  
sim_post_cmd = "vsim -do wave.do -i top_tb"  
  
modules = {  
    "local" : [ ".././../_testbench/top_tb" ],  
}
```




Reset signal

- input to initialize signals to a predetermined state
- synchronous: reset is checked at the rising edge of clock
- asynchronous: reset in sensitivity list of process

```
entity test is
  port (
    clk  : in std_logic;
    reset : in std_logic;
    sig_i : in std_logic;
    sig_o : out std_logic
  );
end entity;
```



```
test_proc: process(clk, reset)
begin
  if reset = '1' then
    sig_o <= '0';
  elsif rising_edge(clk) then
    sig_o <= sig_i;
  end if;
end process;
```



```
test_proc: process(clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      sig_o <= '0';
    else
      sig_o <= sig_i;
    end if;
  end if;
end process;
```

Other Concepts

- **package** = collection of declarations
 - written in separate file
 - can be called and used in an entity
- **function** = describe an algorithm
 - one output and multiple inputs
- **attributes** = parameters of signal

```
signal A : std_logic_vector(7 downto 0);  
A'left  -- equals 7  
A'range -- equals 7 downto 0  
A'length -- equals 8  
  
label_gen: for i in A'range generate
```

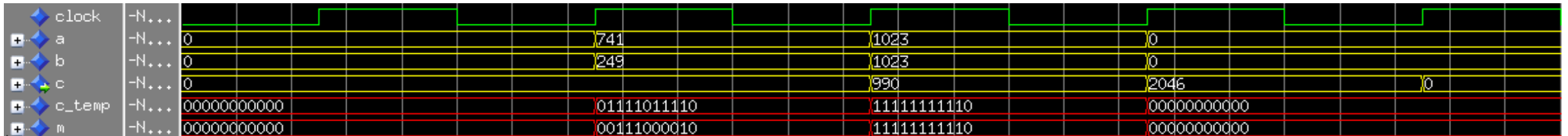
```
package utils is  
  constant six : positive := 6;  
  type matrix_t is array (natural range <>) of  
    std_logic_vector(6 downto 0);  
  function div(a : natural; b : positive) return natural;  
end package;  
  
package body utils is  
  function div(a : natural; b : positive) return natural is  
  begin  
    return a / b;  
  end function;  
end package body;
```

```
use work.utils.all;  
  
entity test is  
  generic (  
    gen : natural := 12;  
  )  
  port (  
    data_i : in  matrix_t(con downto 0);  
    data_o : out std_logic_vector(div(gen,con) downto 0)  
  );
```

Binary Addition

- open `firmware/modules/bin_add/bin_add.vhd`
- implement following functionality
 - assume `std_logic_vector` **input** to be unsigned numbers
 - use **for generate** to connect single bits via logic gates
 - internal **carry** signal needed for addition
 - use temporary signal **c_temp** for result and assign **output** with next clock cycle
- simulation in `firmware/sim/modelsim/bin_add`

$$\begin{array}{r} a = 110101110 \\ b = \quad 101011 \\ \text{carry} = 001011100 \\ \hline c = 111011001 \end{array}$$



Binary Multiplication

$$\begin{array}{r}
 a \times b = \\
 \text{matrix_s} = \\
 \begin{array}{r}
 110101110 \times 101001 = \\
 \hline
 \begin{array}{r}
 110101110 \\
 + 000000000 \\
 + 110101110 \\
 + 000000000 \\
 + 000000000 \\
 + 110101110 \\
 \hline
 = 100010011011110
 \end{array}
 \end{array}
 \end{array}$$

- open `firmware/modules/bin_mult/bin_mult.vhd` with a text editor
- implement following functionality
 - fill **a** into row of **matrix_s** if corresponding bit of **b** is asserted
 - use **bin_add** to keep adding row of **matrix_s** to **c_temp**
- simulation in `firmware/sim/modelsim/bin_mult`

