# Exercise 1:

### a) Combinational Logic

In the first exercise, you will learn to design, simulate and verify a VHDL module with standard text editors and the widely used tool ModelSim from Mentor Graphics. As a first example, you should develop a simple module which realises the logic function $a \wedge \bar{b}$ .

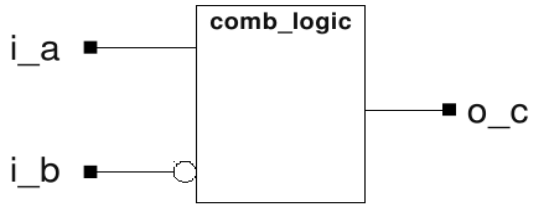



*Figure 1: comb_logic*

**Port declaration:**

- *i_a* : in std_logic
- *i_b* : in std_logic
- *o_c* : out std_logic

To write the VHDL-Code, you should use GNU emacs as your default editor. Emacs is a powerful editor with a steep learning curve, but it is a powerful tool for the development of VHDL code because of the many possibilities it offers.

Once you have written your code, you must simulate it to verify its functional behaviour. An important point you must consider with hardware description languages is the difference between simulatable and synthesizable code. Synthesizable code is the code you can implement in hardware. This code is only a small subset of VHDL. You cannot e.g. synthesize timing-conditions like “wait for 10ns”, since hardware such as a simple CMOS transistor does not understand any timings, but can only switch on and off. Nevertheless, VHDL offers such statements, along with other constructs, to simplify the simulation your modules.

Now you can start ModelSim. In order to do this properly, you must setup the ModelSim program as follows. Because of the files ModelSim creates, please be sure to start the program from the correct directory.

```
$ setup-lm modelsim 10.1b
```

The following message appears:

```
Setup for ModelTech ModelSim SE 10.1b ok.
Info: Doc: <mold>
     Start: <vsim>
```

Now you can start ModelSim with:

```
$ vsim &
```

There are several ways to work with ModelSim, two of them will be explained in this lab. The obvious way to work with the tool is with the help of the GUI. Another more common approach in EDA is the use of scripts written in TCL (Tool Command Language).

We will start using ModelSim by setting up a project with the help of the Project Navigator inside the GUI.

## Working with the Project Navigator

First of all, you must create a new project as shown in figure 2.

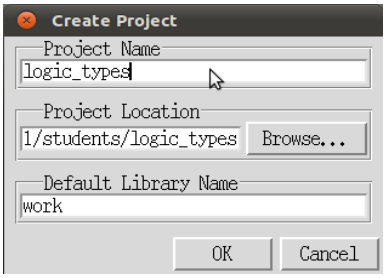



*Figure 2: Creating a project*

In the next window, you have the possibility to add existing files to your project. Here you add your *.vhd file combinational_logic.vhd, which now exists in your workspace, as you can see in the figure below. In the next step, you will compile your module.

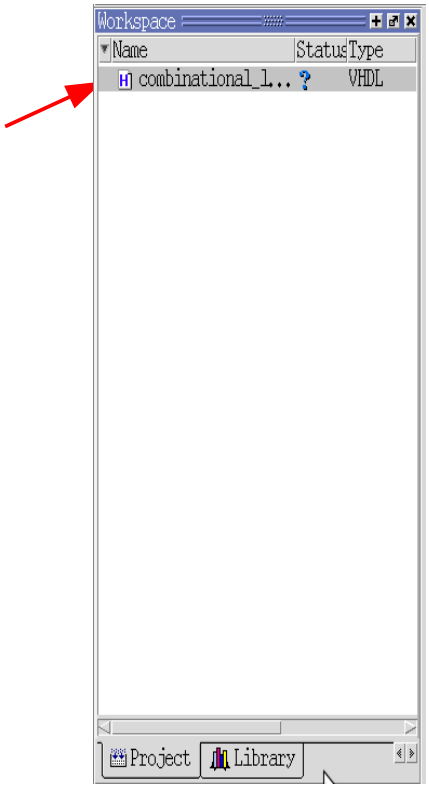



*Figure 3: Modules in workspace*


*Figure 4: Libraries in workspace*

You can compile your module by right-clicking on it in your project workspace. If the syntax of your module is correct, ModelSim will compile the module to the default library **work**. Otherwise, ModelSim shows you syntactic mistakes and the lines where they occur. If necessay, you should correct these mistakes step by step.
After a successful compilation, you know that your module has a correct syntax, but you don't know if it works correctly. To simulate the function of your module, you must add the **testbench**, a

special type of file that feeds your module with stimuli, to your project.

After compiling your testbench to the work library, you must add a simulation configuration to the project. Here you choose the testbench as design unit.

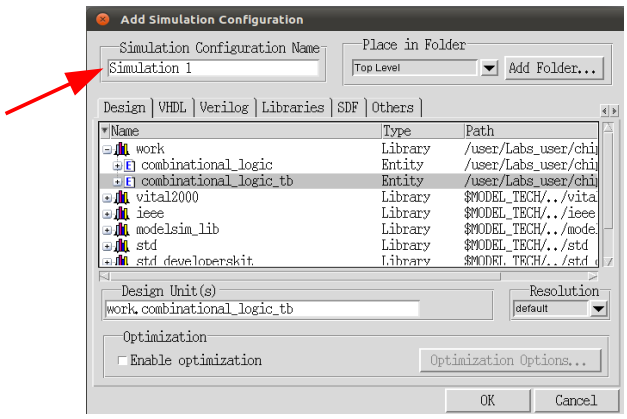



*Figure 5: Adding a simulation configuration*

By double-clicking on the simulation configuration, you start the simulation environment.

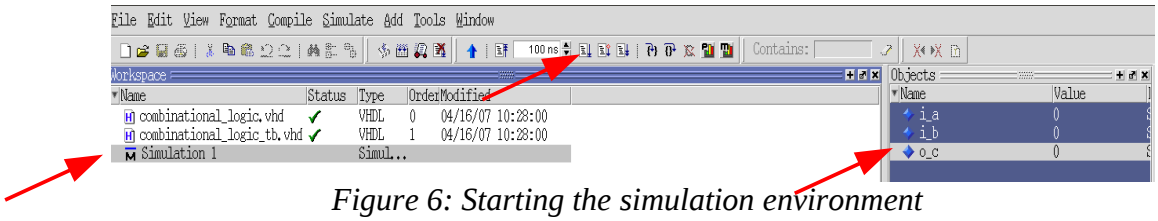



*Figure 6: Starting the simulation environment*

You can add waves to a W**aveform**, by right-clicking on the objects. If you now press ***run-all***, ModelSim generates a Waveform and you can see the signal sequence your module generates as shown in figure 7 .

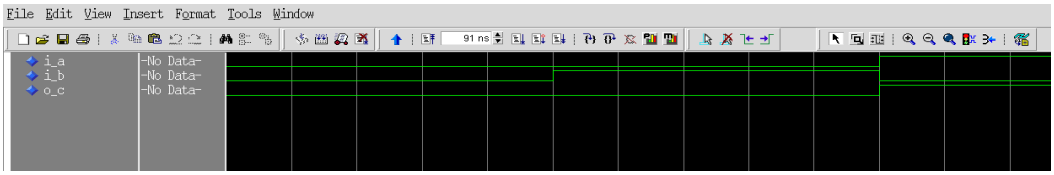



*Figure 7: Example Waveform*

By analysing this Waveform, you can see that the output o_c is only set when i_a is high and i_b is low, whis means that the function of this example is correct and satisfies the logic equation given.

**b) Developing and Testing a Simple Register**

In the next example, you must describe the function of a register and simulate it with the help of a provided script. The register you have to develop is a single-bit register with synchronous reset and should have the same function as a standard D Flip-Flop. This means that the value of the input *i_d* is assigned to the output *o_q* on each rising clock edge.

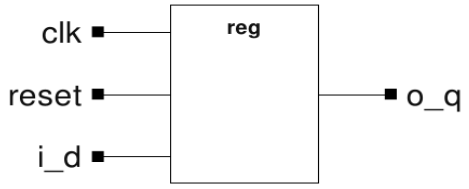



*Figure 8: reg*

**Port declaration:**

- *clk* : in std_logic
- *reset* : in std_logic
- *i_d* : in std_logic
- *o_q* : out std_logic

## Working with scripts

As you have seen, working with the Project Navigator requires many manual steps inside of the GUI to simulate your code. This can quickly become tedious for larger projects with many design-iterations, so designers often try to automate this process through the use of scripts. ModelSim offers the possibility to use standard TCL or a simpler ModelSim-specific TCL-derivative.
The provided script you have to use is built with a few, simple commands. Of course, ModelSim offers much more possibilities to run a simulation with a script. For further information, please see the ModelSim Quick Guide on the lab's homepage.

```
# vlib creates library
vlib work

# vcom compiles your VHDL-modules (use vlog for Verilog-Code)
# regard hierarchy of modules, compile order is down to top
# if not declared in another way, modules are compiled to the
# default library work
vcom reg.vhd
vcom reg_tb.vhd

# start simulator
vsim -novopt reg_tb

# add wave adds wave to waveform
# hint: you can also add the waveforms by hand, save the
# waveform file and start the waveform file with simulation
# script (do waveform.do)
add wave sim:/reg_tb/clk
add wave sim:/reg_tb/reset

# insert divider
add wave -divider inputs
add wave sim:/reg_tb/i_d
add wave -divider output
add wave sim:/reg_tb/o_q

# run simulation
run -all
```

You can start the script in the ModelSim prompt with the ***do*** command.

```
VSIM 5>  do SCRIPT_NAME.do
```

Now the waveform window should open and you can check the function of your Flip-Flop.

To get used to work with ModelSim, you will have to use each of the described methods once in the next two exercises. For thefollowing exercises, you are free to choose the method you prefer.

# Exercise 2:

### a) Using Design Hierarchy in VHDL

Design Hierarchy is one of the key concepts in VHDL development. Now that you have learned to test your code with ModelSim, you will learn to construct an easy hierarchical design with the modules you have developed. For this, you must declare components of the modules and connect instances of these components as shown in figure 1.

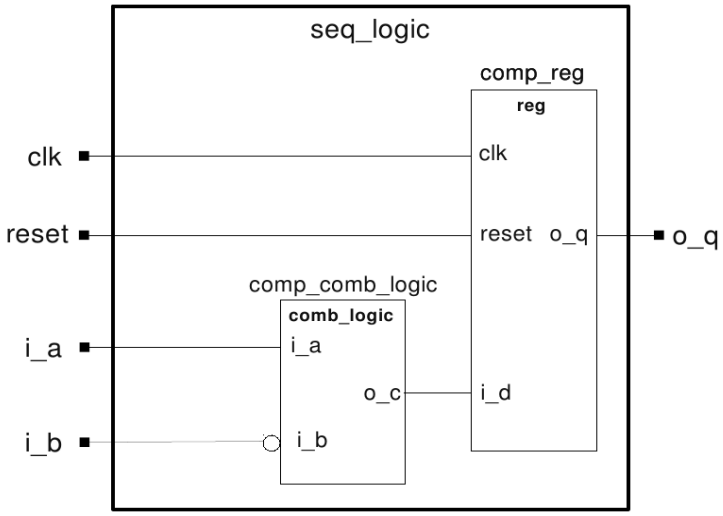



*Figure 1: seq_logic*

**Port declaration:**

- *clk* : in std_logic
- *reset* : in std_logic
- *i_a* : in std_logic
- *i_b* : in std_logic

We could also integrate the functionality of the combinational logic into the part of the code describing the register. Instead, however, we will use the VHDL component construct. One advantage of using components is the possibility for code reuse. Also, the simulation and debugging of the code is simplified, since you can now simulate small parts of your design. This is an important point, since you can easily have several thousand signals in a design.

To build a component, you must declare the component in the declaration part (the same part where you have to declare your signals) of your architecture first. The name of the component has to be the name of the entity you want to use in this case, the entities of the modules of Exercise 1 (*comb_logic*, *reg*). The generics and ports must be declared in the same way they are in the entities. Because no generics are used in this exercise, this declaration can be omitted.

```
-- component declaration
component entity_name is
      generic (generic_declarations );
      port (input_and_output_declarations);
end component entity_name;
```

Now, the component with its generics and ports is known to your architecture, but not yet used. To use this component, you must instantiate it inside your architecture.

First of all, you can label the component with a name deviating from the entity name (*comp_comb_logic*, *comp_reg*). Now you can connect the ports of the instantiated component with signals used in your architecture as shown in the figure. Ports of your entity can be used directly in the port map of the component. Again, you can omit the generic map. (If you want to use a generic map, be aware of the syntax! There is no semicolon between a generic map and a port map!)

```
-- component instantiation
label : entity_name
       generic map (component_generic    => generic)
       port map (component_input => input / signal,
                 component_output => output / signal);
```

Since you connect two components directly, you must declare an additional signal to connect the output o_c from *comp_comb_logic* with the input i_d from *comp_reg*.

Furthermore, a direct instantiation of components was defined in VHDL-93. This instantiation does not require a component declaration and has the following syntax.

```
-- direct component instantiation
label : entity library_name.entity_name(architecture_name)
         generic map (component_generic  => generic)
         port map (component_input => input / signal,
                   component_output => output / signal);
```

If you use this instantiation, consider that you must specify the library in which your entity resides.

Now set up a new ModelSim project with the Project Navigator and simulate and verify your design with the given testbench! Do not forget to compile the files of Exercise 1 to your ***work*** library, otherwise your components will be unbound and the simulation will not work.

**b) Debugging of a counter**

The next task is to debug a given six-bit counter.

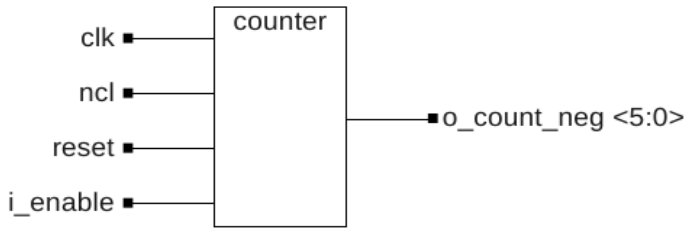



*Figure 2: counter*

**Port declaration:**

- *clk* : in std_logic
- *ncl* : in std_logic
- *reset* : in std_logic
- *i_enable* : in std_logic
- *o_count_neg* : out std_logic_vector (5 downto 0)

The counter should implement the following functions:

- count up from 0 to 63 in steps of 1.
- counting must be enabled by setting the input *i_enable* to 1, else the counter retains its old value.
- the input *reset* works as a synchronous and the input *ncl* as an asynchronous reset. Both set the counter back to 0.
- the output *o_count_neg* is the inverted value of the actual counter.

Write a simulation script and correct the mistakes in the module. You will find syntactic, as well as functional mistakes.

# Exercise 3

### a) Signals vs. Variables in VHDL

In the previous exercises, you used **signals** to describe your hardware. VHDL also offers the possibility to use **variables**, which have a different behavior.

The main functional difference between signals and variables is the “value history” associated with signals. Variables do not have a history of their previous values; a variable only knows its current value. This especially results in a different behavior when assigning a value within a process. The value of a signal will be updated at the end of the execution of the process, while variables will be updated immediately. Because of this property, variables can be overwritten with a sequential statement in a process. Overwriting a signal with multiple assignments in a sequential process will just use the last statement while ignoring all earlier assignments.
The behavior of variables can e.g. be exploited in complex calculations to hold intermediate values, but this can also lead to problems: It is not possible to see the intermediate changes of a variables' value over time in a waveform during simulation, which can complicate debugging. Also, it is not possible to use variables in the sensitivity list of the process, which can result in differences between the simulation results and the synthesized hardware (synthesizers mostly ignore sensitivity lists and just issue a warning).
For larger designs like memories, it might be advantageous to use variables, since they consume less memory resources during simulation than signals do.

As you can see in the example code below, the syntax for variables is slightly different than the syntax for signals.

```
-- usage of signals
architecture rtl of example is
      signal sig: std_logic_vector(3 downto 0) := (others => '0');
begin
      sig <= “0010”;
end;

--  usage of variables
process(sensitivity list) is
      variable var: integer;
begin
      var := var + 1;
end process;
```

In this exercise, you must design two modules which perform the same calculation. To clarify the differences between signals and variables, you must implement one of the modules using variables and the other using signals.

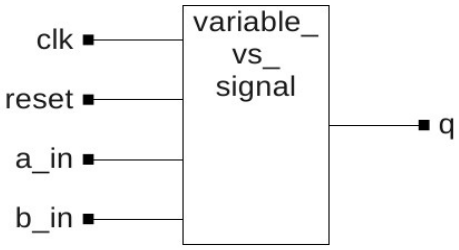



*Figure 1: variable_vs_signal*

The calculation shown in Figure 2 should be performed. The values of ***a_in*** and ***b_in*** must be fed to A and B, the result E must be fed to ***q***. Try to convert all the values from type std_logic_vector to type integer to perform the calculation. You can search on the internet for the conversion functions you will need.

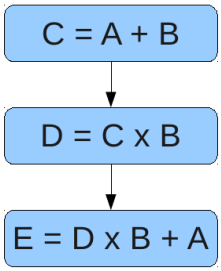



*Figure 2:*
*Calculation rule*

In this exercise, no explicit testbench is given. The stimuli for the inputs are provided by the ModelSim simulation script with the help of the ***force*** command. This is an easy approach to generate stimuli, but it will not be sufficient to test larger and more complex designs.

Try to find out the differences between variables and signals by analyzing the waveforms of both modules. If you understand the differences properly, you are free to use either signals or variables to describe your hardware in the upcoming exercises. But since we neither have complex calculations nor very large designs in our lab, the general recommendation is to use signals.

**b) Serial To Parallel Conversion**

This block should convert a serial data stream into a parallel data word using a shift register.

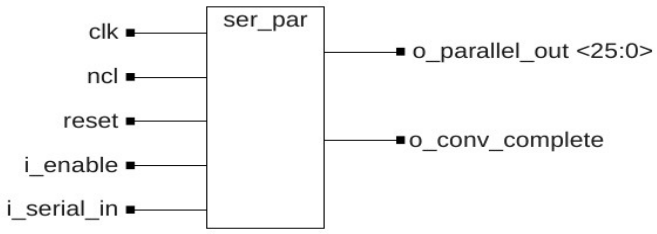



*Figure 3: ser_par*

**Port declaration:**

- *clk* : in std_logic
- *ncl* : in std_logic
- *reset* : in std_logic
- *i_enable* : in std_logic
- *i_serial_in* : in std_logic
- *o_parallel_out* : out std_logic_vector (25 downto 0)
- *o_conv_complete* : out std_logic

To do so, each new incoming bit must be shifted into the register on a rising clock edge when the *i_enable* signal is active. You can realize a shift register by using the concatenation operator **&.** This operator concatenates vectors, as shown in Figure 4.


*Figure 4: Concatenation operation*

After the conversion of each data word, a reset is triggered from the outside and the value of the shift register is set to 0. The serial data stream is sent to the converter with the MSB first and consists of 27 bits. The MSB is always 1 and is called the start bit of the data word.
When the whole data stream has been completely shifted into the register, this start bit has reached the MSB position of the shift register, which is assigned to the output o_*conv_complete*. This output signifies that a complete data word has been converted and that it is available at the output *o_parallel_out*.

# Exercise 4:

In this exercise you will learn how to implement a finite state machine (FSM) in VHDL. As examples, the two most commonly used types of FSMs were selected: the Moore and the Mealy state machines (it is assumed that the differences in the behavior of both types are already known). There a several ways to code a state machine in VHDL. Most examples found on the internet or in the literature use a 2-process model, some even use 3 processes. The most common approach at our institute is a 1-process model, therefore this model is explained in this exercise.

**a) Mealy Machines**

In the following code, you will find a simple example of how to implement a Mealy FSM with 1 process. The first thing you must do is declare a new type for your states and a signal of this type.

```
type fsm_states is (STATE_ONE, STATE_TWO);
signal state_fsm : fsm_states := STATE_ONE;
```

Now, you have a signal state_fsm with the possible values STATE_ONE or STATE_TWO. In the next step, you have to calculate your state transitions. First, you must read out the actual state of your signal *state_fsm* with a case statement. Then, you can assign a new state to this signal depending on the conditions of your FSM.

```
state_transitions : process (clk)
begin
      if rising_edge(clk) then
            if reset = '1' then
                  state_fsm <= STATE_ONE;
            else
                  case state_fsm is
                        when STATE_ONE =>
                              if condition then
                                    state_fsm <= STATE_TWO;
                              end if;
                        when STATE_TWO =>
                              if condition then
                                    state_fsm <= STATE_ONE;
                              end if;
                  end case;
            end if;
      end if;
end process state_transitions;
```

After you have calculated your state_transitions, you can assign your outputs with concurrent statements outside the process. This concurrent statement describes a simple multiplexer. Remember that the outputs of a Mealy machine depend on the state and on the inputs.

```
o_x <= '0' when ( )
          else '1';
```

Your task is to implement a Mealy machine from the given state diagram on the next page.

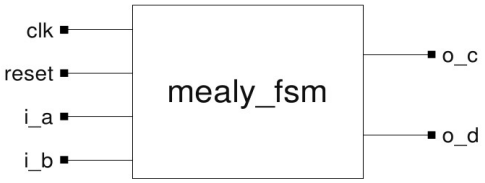



*Figure 1: mealy_fsm*

**Port declaration:**

- *clk* : in std_logic
- *reset* : in std_logic
- *i_a* : in std_logic
- *i_b* : in std_logic
- *o_c* : out std_logic
- *o_d* : out std_logic

***Hint:*** There is only one testbench, which tests both state machines. If you compare the state machines with each other in the waveform of your simulation, they should always be in the same state – with a different behavior at the outputs!

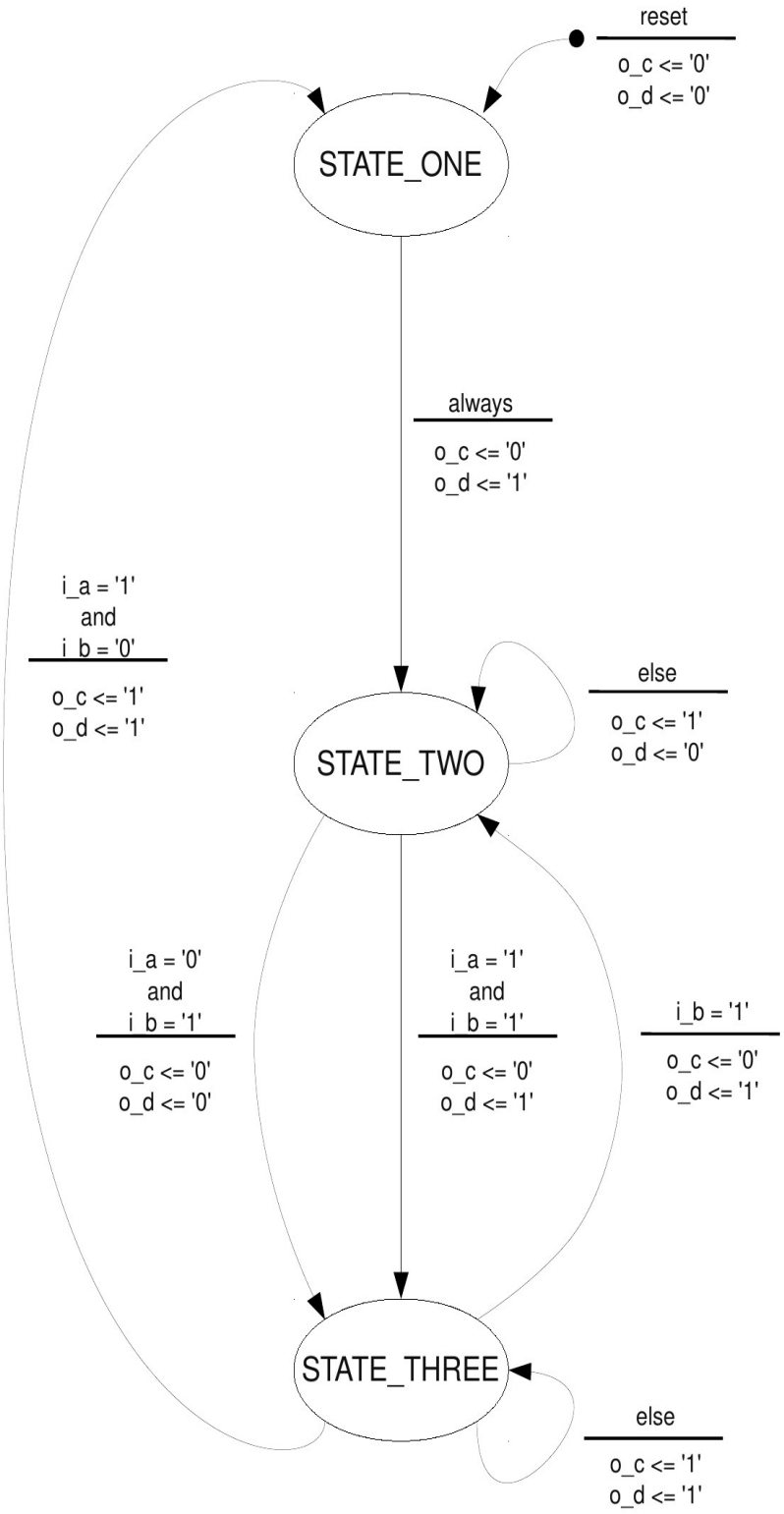



*Figure 2: State chart of Mealy machine*

**b) Moore Machines**

Now, you will implement a Moore machine of a given state diagram. This time, no example is given. Considering the differences in the design of Moore state machines compared to the Mealy design, it should be easy to change the given design from Mealy to Moore.

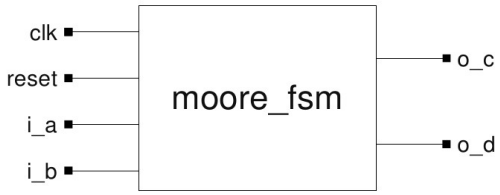



*Figure 3: moore_fsm*

**Port declaration:**

- *clk* : in std_logic
- *reset* : in std_logic
- *i_a* : in std_logic
- *i_b* : in std_logic
- *o_c* : out std_logic
- *o_d* : out std_logic

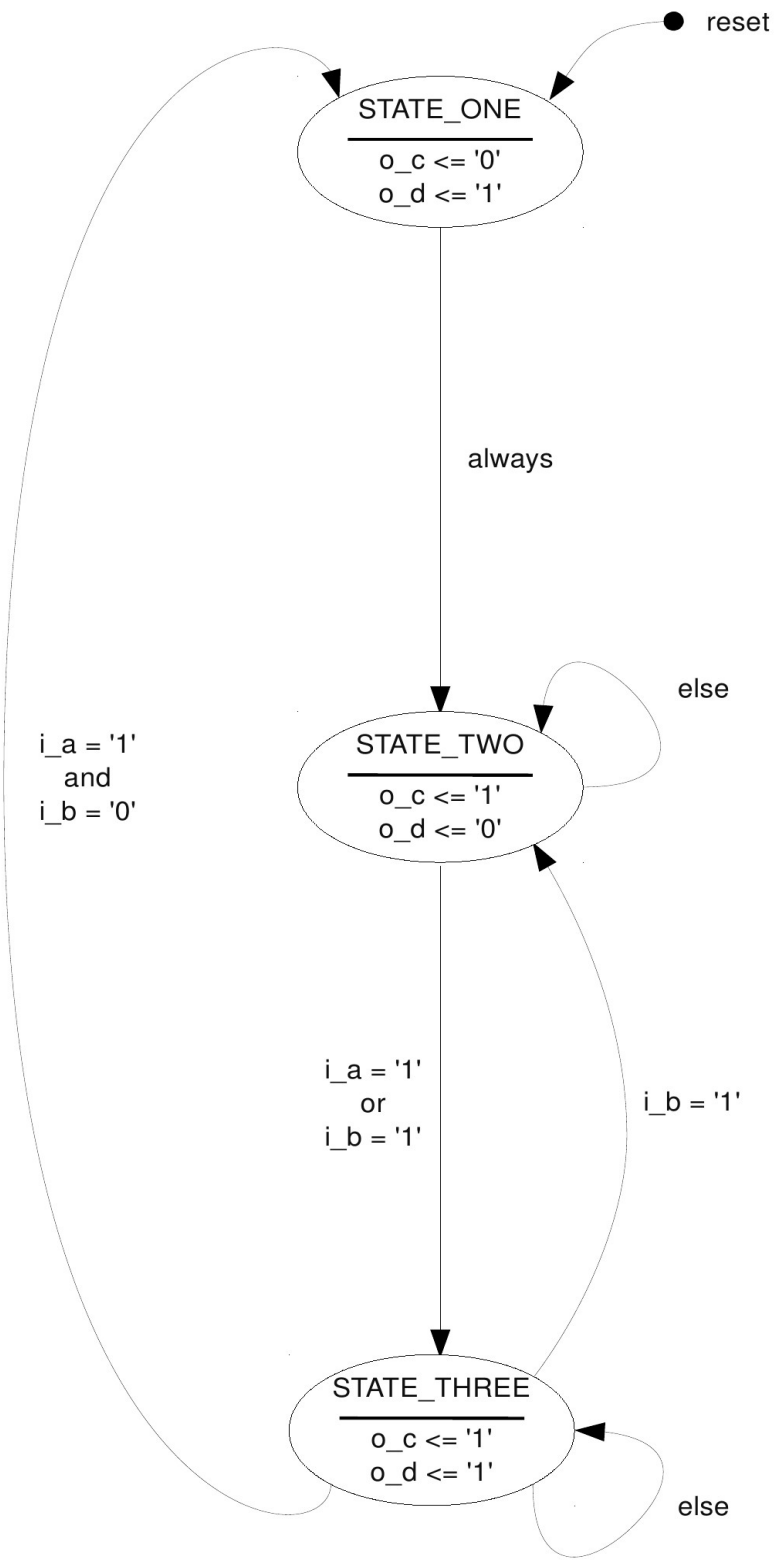



*Figure 4: State chart of Moore machine*

**c) Testbenches (*game_states*)**

As you have seen in the previous tasks, testing through simulation is an important step in the design process. At design time, the best choice is to test each module separately. This allows the designer to avoid problems and time consuming debugging after all modules have been combined into a larger design, because it is less difficult to find an error in the smaller, independent and less complex sub-modules.

Hardware description languages (HDLs) allow simulation at various levels, ranging from behavioral simulation, where only the logic behavior is simulated, up to a post-route simulation, where the HDL model is combined with technology libraries of the target architecture (in our case the FPGA) to get information on delays and rise and fall times of gates or signal lines. In this lab, we will only use a behavioural simulation of our VHDL models.
Unlike in the earlier lessons, where the test benches were provided, it is your task to write a test bench for a given VHDL file.

The unit under test will be the module *game_states*, a sub-module from the block *game_engine*. It is responsible for controlling the game flow, counting points and changing the serving player. It consists of two Mealy state machines. The state diagrams for both are provided for clarity.

The first state machine has four states:

- *SERVE* : In this state, the ball is bound to the serving player's paddle. Paddle movement is allowed. The state is exited when the serve key is pressed.
- *PLAYING* : In this state, ball and paddle movement is allowed. It is exited when one player scores a point.
- *WAITING* : After a scored point, the FSM enters this state and remains there for a period of time. Ball and paddle movement is disabled, but the ball remains at its previous location to give the players a chance to see the scored point. When exiting this state, there are two possibilities. If no player has reached the point limit to win the game, the state machine returns to the *SERVE* state. Otherwise it changes to the *WON* state.
- *WON* : This state is only entered when one player reaches the score limit. In this state ball and paddle movement remain disabled and it can only be exited by a reset of the module *game_engine*.

The second FSM has only two states:

- *PLAYER_1* : Player 1 has the right to serve. The state is exited after five serves.
- *PLAYER_2* : Player 2 has the right to serve. The state is exited after five serves.

The first thing you should do is examine the state diagrams and the VHDL code to fully understand the function of the module.
Then you can begin writing the testbench. Like in the previous lessons, the stimulus data should be read from a text file with the file ending *.vec*. It would be a good idea to look at previous testbenches to learn how to build them and how read data from files.
The testbench should move the state machine through all its transitions at least once. For each transition, you must ensure that it only occurs when all transition conditions are true.

***Hint:*** To keep the simulated time low, you should change the value of the constant *C_WAIT_TIME*.

# Exercise 5

### a) 4-bit Multiplier

In this exercise, you must design a 4-bit multiplier.

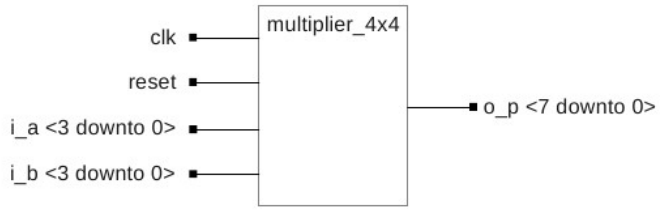



*Figure 1: multiplier_4x4*

**Port declaration:**

- *clk* : in std_logic
- *reset* : in std_logic
- *i_a* : in std_logic_vector (3 downto 0)
- *i_b* : in std_logic_vector (3 downto 0)
- *o_p* : out std_logic_vector (7 downto 0)

To design this module, you will use a structural description. This means that you describe the structure of the hardware instead of describing its behavior. With a behavioral description, we can multiply two signals in a very simple way like it is shown in the example code.

```
A <= B * C;
```

The synthesis tool will use the dedicated multipliers on the FPGA or, if no multipliers are available, synthesize the code in the usual way. Your task is to describe a 4-bit multiplier using a structural description.

To understand the structure of a binary multiplier, you should first have a look at the scheme of the binary multiplication shown in Table 1.

| | | | | | | $b_3$ $b_2$ $b_1$ $b_0$ x $a_3$ $a_2$ $a_1$ $a_0$ | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | $b_3 \times a_0$ | $b_2 \times a_0$ | $b_1 \times a_0$ | $b_0 \times a_0$ |
| | | | + | $b_3 \times a_1$ | $b_2 \times a_1$ | $b_1 \times a_1$ | $b_0 \times a_1$ | |
| | | + | $b_3 \times a_2$ | $b_2 \times a_2$ | $b_1 \times a_2$ | $b_0 \times a_2$ | | |
| | + | $b_3 \times a_3$ | $b_2 \times a_3$ | $b_1 \times a_3$ | $b_0 \times a_3$ | | | |
| **SUM:** | $p_7$ | $p_6$ | $p_5$ | $p_4$ | $p_3$ | $p_2$ | $p_1$ | $p_0$ |

*Table 1: Binary multiplication scheme*

Binary multiplication is very similar to the multiplication of decimal values and is achieved by adding a list of shifted multiplicands according to the digits of the multiplier. You can see an example multiplication in table 2.

| **1** | **1** | **0** | **1** | **x** | **1** | **0** | **0** | **1** |
|---|---|---|---|---|---|---|---|---|
| | | | | | 1 | 1 | 0 | 1 |
| | | | + | 0 | 0 | 0 | 0 | |
| | | + | 0 | 0 | 0 | 0 | | |
| | + | 1 | 1 | 0 | 1 | | | |
| **0** | **1** | **1** | **1** | **0** | **1** | **0** | **1** | |

*Table 2: Example multiplication*

In hardware, we can realize a binary multiplier with an array of multiplier units, as shown in Figure 1. These units multiply two input bits and feed the result to a full adder. With this unit, we are able to multiply two single bits, e.g. $b_1 \times a_0$ , and add a shifted multiplicand, e.g. $b_0 \times a_1$. Since the inputs $a_{in}$ and $b_{in}$ must also be connected to the next multiplier unit, they are also fed directly through as unmodified signals.

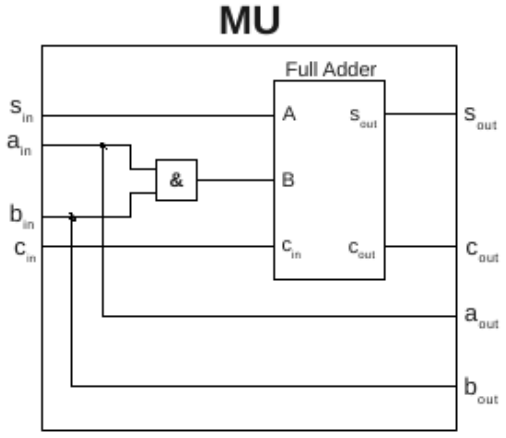



*Figure 2: Multiplier unit*

For a 4-bit multiplier, 16 MUs are needed and must be connected as shown in Figure 2. Comparing this array of MUs to the scheme shown in Table 1, you can see that both have exactly the same structure.

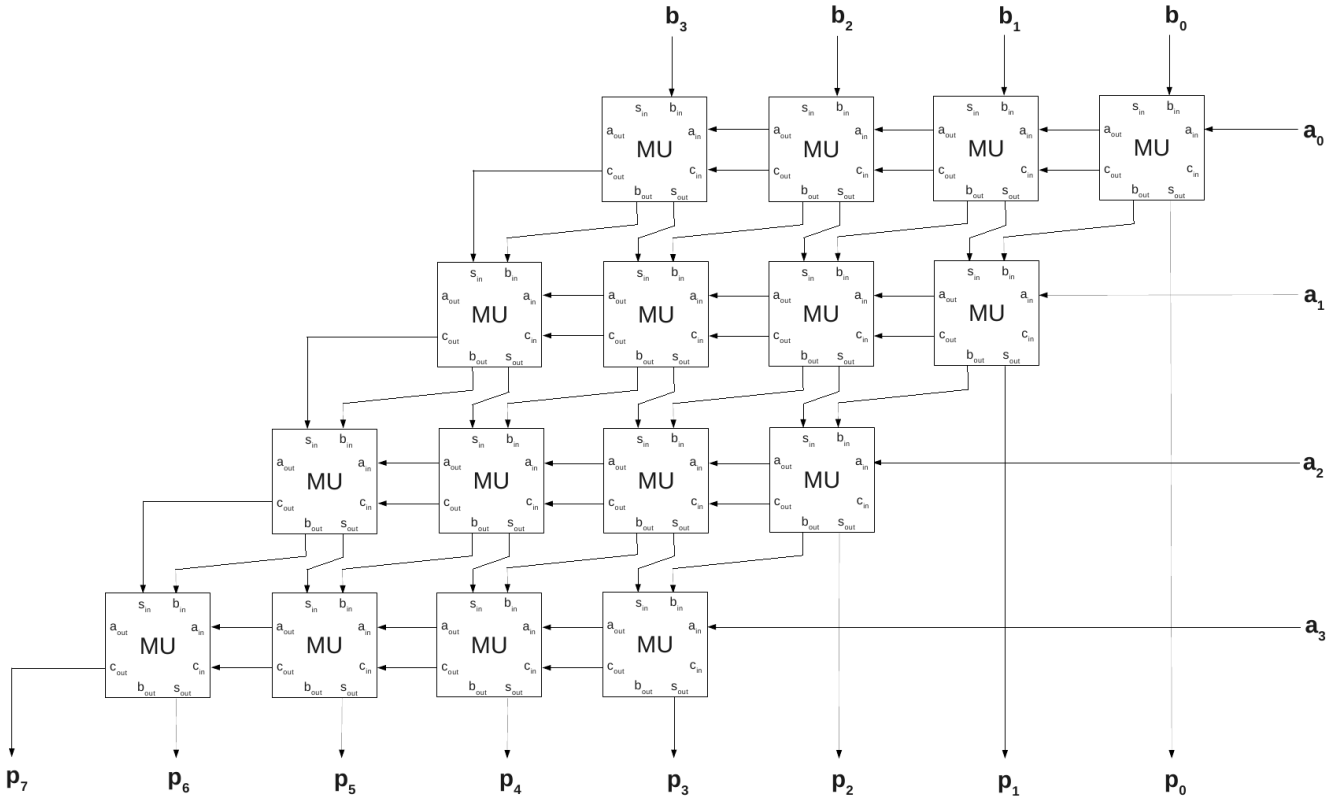



*Figure 3: 4-bit multiplier schematic*

Instead of instantiating and connecting each of these components separately, you should use the generate statement. This statement is particularly important for generating regular structures such as memories. Figure 4 shows a simple 4-bit shift register that is realized with the generate statement.

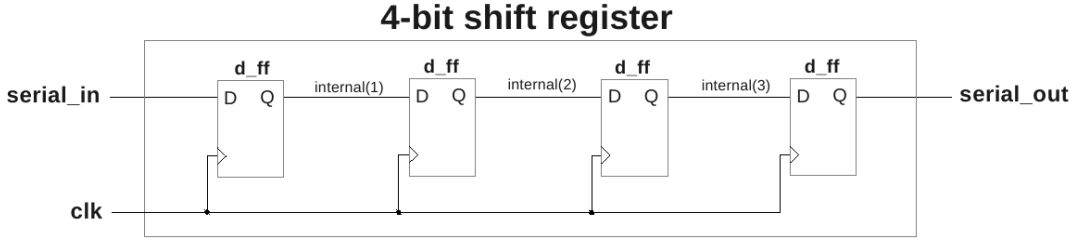



*Figure 4: 4-bit shift register*

Now have a look at the code we need to generate this shift register. It consists of 3 **if** (condition) **generate** statements within a **for** (loop variable) **generate** statement. The component d_ff is instantiated inside of the if statement and connected with signals depending on the loop variable.

```
shift_reg: for i in 4 downto 1 generate

-- connection of first flip-flop
reg_begin: if (i = 1) generate
               d_ff_begin: d_ff
                 port map (  d => serial_in,
                             clk => clk,
                             q => internal(i));
           end generate;

-- connection of middle flip-flops
reg_middle: if (i > 1) and (i < 4) generate
               d_ff_middle: d_ff
                  port map ( d => internal(i-1)
                             clk => clk,
                             q => internal(i));
            end generate;

-- connection of last flip-flop
reg_end: if (i = 4) generate
              d_ff_end: d_ff
                  port map ( d => internal(i-1),
                             clk => clk,
                             q => serial_out);
         end generate;

end generate;
```

The two main challenges when designing the 4 bit multiplier will be the differentiation between the various cases and the numbering of the interconnections required. Therefore, it is helpful to create a scheme of all interconnections before writing any code. Since the multiplier has a 2-dimensional structure, you must use a nested loop with different loop variables for the rows and columns.
Try to reduce the number of cases you have to consider. Nevertheless, you will have to use at least 6 different **if** statements inside the nested loop.

Furthermore, you cannot leave any inputs unconnected, so they must be set to '0'. Unconnected outputs only produce warnings during synthesis. To suppress these warnings, you can use the keyword **open** in the component instantiation.
The component mu is provided (you can find mu.vhd in the folder of this exercise), so you only have to declare and instantiate it.

When you have a look at the multiplier structure, you can see that it consists of a large amount of combinational logic. As you can see in Figure 5, the connection of different combinational modules may result in a very long critical path, which lowers the maximum usable frequency.

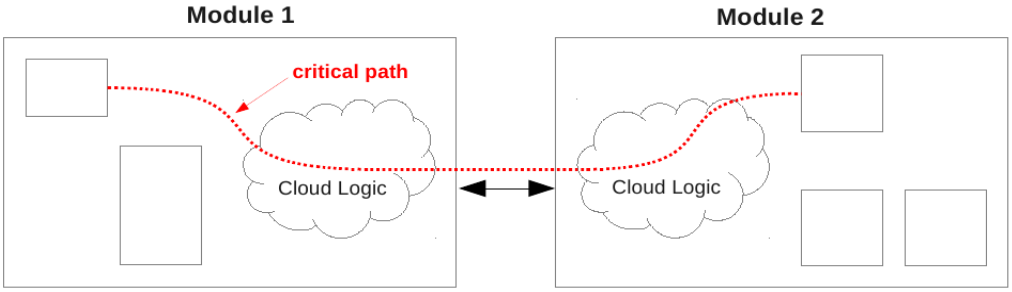



*Figure 5: Critical path without I/O registers*

To prevent this, it is good design practice to use registers for the inputs and outputs. This results in shorter critical paths, but you have to put up with additional cycles of latency (1 for every register inserted).

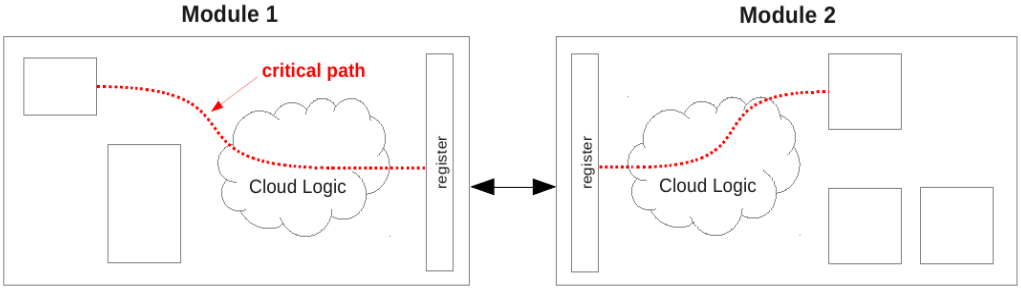



*Figure 6: Critical path with I/O registers*

Therefore, you should also use registers for the inputs and outputs of the multiplier.

The stimuli of the provided testbench tests all possible input combinations. The testbench is self-checking, which means that it recognizes each incorrect output value of your module. It expresses a warning in the ModelSim prompt every time you have an incorrect output value. This is done with an assertion and a severity warning in the VHDL code. Examine the testbench to see how that works.

# Exercise 6:

### a) Generation Of Synchronisation Pulses For The Monitor

Driving a monitor with synchronisation pulses originates from the time when cathode-ray-tube (CRT) displays were the standard display type for computers. In these monitors an electron beam forces the phosphorous coating on the inside of the front of the screen to emit visible light.

This electron beam moves from left to right and top to bottom. When the beam reaches the right or bottom end of the screen, the corresponding synchronisation impulse (horizontal or vertical) forces the monitor to retrace the electron beam to its starting position as shown in figure 1.

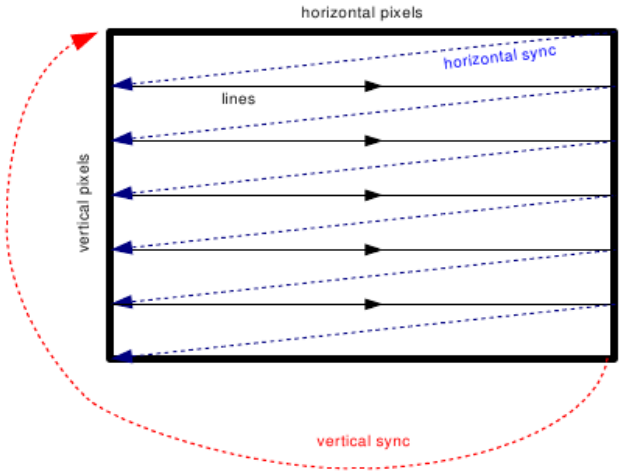



*Figure 1: Vertical and horizontal synchronization pulse*

During the time the beam requires to move back, no image data can be displayed. All other signals required by the monitor are derived from these two synchronisation signals, so it is important that their timing is correct (although modern monitors tend to be more flexible in handling non-standard timings than older ones).
The timing conditions and the generics for the sync pulse are given in the waveform in figure 2. As you can see, a front and a back porch have to be considered along with the sync pulse and the visible area (for the horizontal and for the vertical sync pulse).

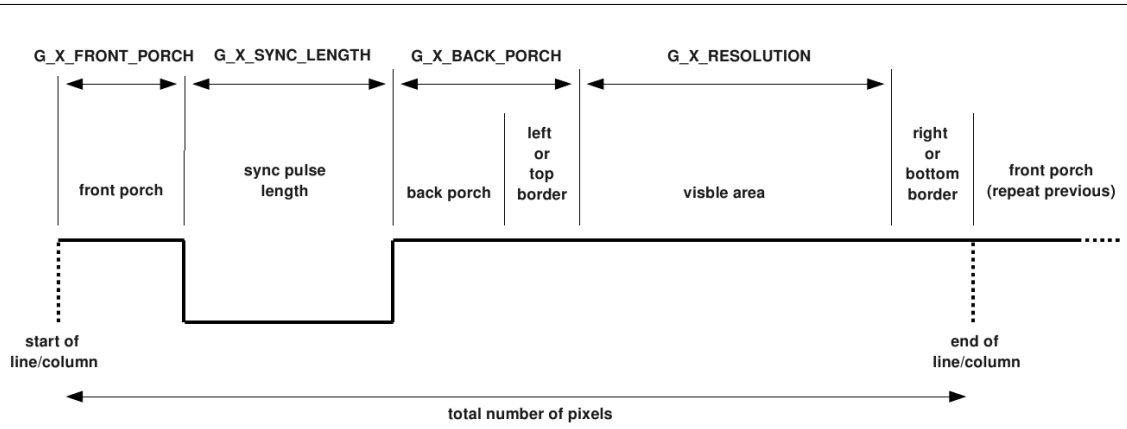



*Figure 2: Timing conditions for the vertical and horizontal sync pulse*

Your task will be to implement a synchronisation pulse generator in VHDL that functions properly for for different resolutions and refresh frequencies (although only 640x480 at a 60 Hz refresh rate will be used in the final project).

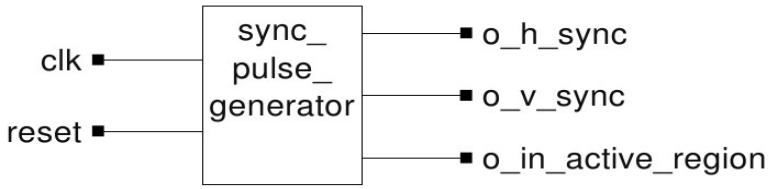



*Figure 3: sync_pulse_generator*

**Port declaration:**

- *clk* : in std_logic
- *reset* : in std_logic
- *o_h_sync* : out std_logic
- *o_v_sync* : out std_logic
- *o_in_active_region* : out std_logic

To realize a sync pulse generator which works for different resolutions and refresh frequencies, we will use generics. Generics are used to parametrize modules and can be written hierarchically (using a generic map described in Exercise 2). Therefore, you will perform calculations using only these generics instead of the values we need for our specific monitor. Afterwards, these generics will be defined in another module. Nevertheless, you can find the values required on the next page.

**List of generics:**

- *G_H_PIXEL_NUMBER* : horizontal pixel number (incl. non visible)
- *G_H_RESOLUTION* : visible horizontal resolution
- *G_H_FRONT_PORCH* : horizontal front porch length in pixels
- *G_H_BACK_PORCH* : horizontal back porch length in pixels
- *G_H_SYNC_LENGTH* : horizontal sync pulse length in pixels
- *G_H_SYNC_ACTIVE* : horizontal sync pulse polarity (1 = pos, 0 = neg)
- each of the above generics exist with a leading '*G_V*' instead of '*G_H*' for their vertical counterparts, too.

**Generic declaration:**
All generics are of type integer, except *G_H_SYNC_ACTIVE* and *G_V_SYNC_ACTIVE*, which are of type std_logic.

The output for horizontal synchronisation pulses is *o_h_sync*, *o_v_sync* for the vertical pulses. Additionally, the output *o_in_active_region* should be active ('1') when the electron beam is in the area where the picture is displayed on the screen.
In order to make the time between two impulses measurable, the synchronisation pulse generator (and the whole block *graphic_output*) is fed with a different clock signal than the rest of the circuit. It has a cycle length that represents the time required by the electron beam to display one pixel on the screen. All times needed are multiples of this pixel clock cycle length and and can thus be measured by a simple counter.

To ensure that all PC monitors work with all computers the VESA (Video Electronics Standards Association[1]) specifies the signal timings. In this lab the display should run at a 640x480 resolution with a refresh frequency of 60 Hz. The detailed timing data for this resolution is given in the manual[2] of the FPGA evaluation board used in this lab (a detailed list for most standard video modes can be found here[3]).

# "VGA industry standard" 640x480 pixel mode

## General characteristics

```
Clock frequency 25.175 MHz
Line  frequency 31469 Hz
Field frequency 59.94 Hz
```

## One line

```
  8 pixels front porch
 96 pixels horizontal sync
 40 pixels back porch
  8 pixels left border
640 pixels video
  8 pixels right border
---
800 pixels total per line
```

## One field

```
  2 lines front porch
  2 lines vertical sync
 25 lines back porch
  8 lines top border
480 lines video
  8 lines bottom border
---
525 lines total per field
```

## Other details

```
Sync polarity: H negative, V negative
Scan type: non interlaced.
```

1 http://www.vesa.org
2 http://www.digilentinc.com/Data/Products/S3BOARD/S3BOARD-rm.pdf
3 http://www.epanorama.net/documents/pc/vga_timing.html

# Exercise 7:

### a) Graphic Buffer

During one read operation from the RAM, the colour data for 32 consecutive pixels is read. This data must then be buffered and converted from a parallel data word to a serial data stream.

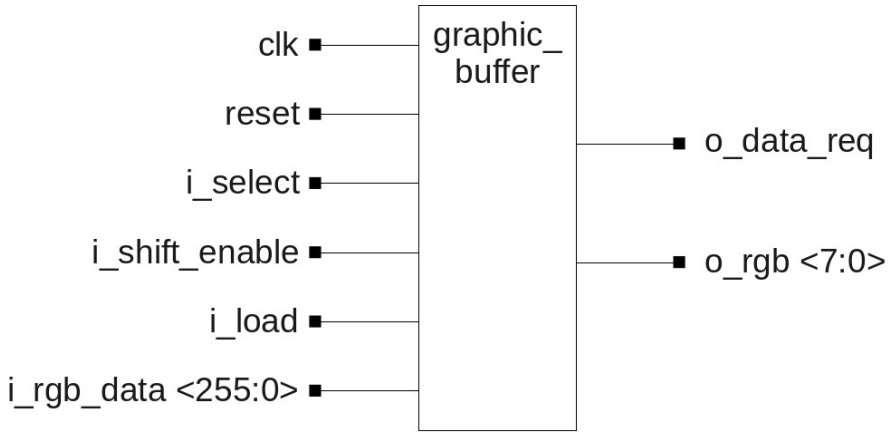



*Figure 1: graphic_buffer*

**Port declaration:**

- *clk* : in std_logic
- *reset* : in std_logic
- *i_select* : in std_logic
- *i_shift_enable* : in std_logic
- *i_load* : in std_logic
- *i_rgb_data* : in std_logic_vector (255 downto 0)
- *o_data_req* : out std_logic
- *o_rgb* : out std_logic_vector (7 downto 0)

To serialize the data,  you can use a shift register, as shown in figure 2.

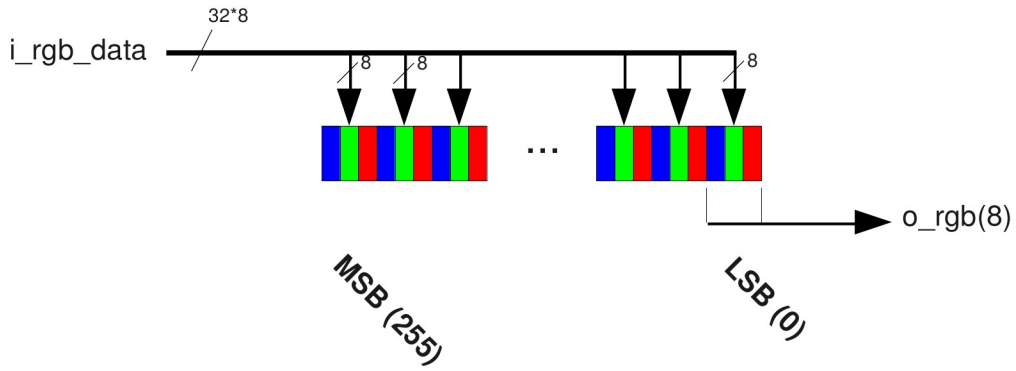



*Figure 2: Shift register used to serialize pixel data*

Two of these buffers are used, so one can be reloaded while the other is converting its data. Since both buffers may not send data simultaneously to the VGA-port, you have to consider the control of these two buffers.
The active buffer is selected by the input *i_select*. On each rising clock edge, the buffer outputs the data of a new pixel at the output *o_rgb* when both the *i_select* and the *i_shift_enable* inputs are active ('1'). It is important that the output data is valid immediately after a rising edge on the *i_select* input. The value of the output *o_rgb* is never used during the time *i_select* is 0, so it does not have to have a defined value.
The input *i_load* is used to reload the buffer with new data. Because there are two of these buffers and the load signal is connected to both, only the unselected buffer (the buffer that requested new data is disabled by the *graphic_buffer_controller* and its input *i_select* is set to 0) is reloaded.
After all 32 pixels have been displayed, the active buffer requests new data by setting the output signal *o_data_req*. It is important that this signal is already active on the rising edge after the clock cycle in which the 32nd pixel has been displayed, so that the buffer controller can switch the active buffer on the next clock edge. The data request output must be set after a reset, too.

# Exercise 8:

### a) Graphic Buffer Controller

The task of the *graphic_buffer_controller* module is to reload the two graphic buffers with data from the RAM and to select the currently active buffer. It also generates the read address for the RAM controller.

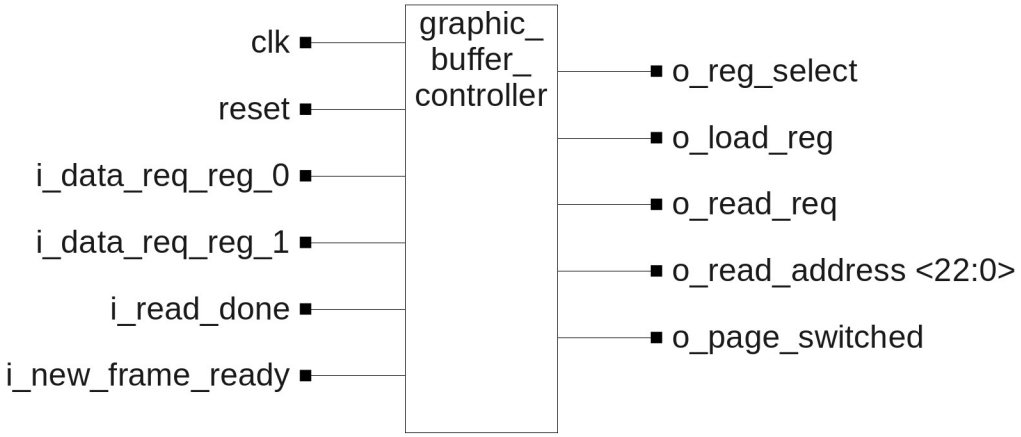



*Figure 1: graphic_buffer_controller*

**Port declaration:**

- *clk* : in std_logic
- *reset* : in std_logic
- *i_data_req_reg_0* : in std_logic
- *i_data_req_reg_1* : in std_logic
- *i_read_done* : in std_logic
- *i_new_frame_ready* : in std_logic
- *o_reg_select* : out std_logic
- *o_load_reg* : out std_logic
- *o_read_req* : out std_logic
- *o_read_address* : out std_logic_vector (22 downto 0)
- *o_page_switched* : out std_logic

**Generic declaration:**

- *G_H_RESOLUTION* : integer
- *G_V_RESOLUTION* : integer

On an incoming request for data through the inputs *i_data_req_reg_0* or *i_data_req_reg_1*, the controller immediately switches the active buffer to the buffer not requesting data on the next rising clock edge. The value of the output *o_reg_select* signifies the selected buffer (0 = buffer 0, 1 = buffer 1). If both buffers send a request simultaneously, one of them must have a higher priority. At the same time as changing the buffer, a read request is sent to the RAM controller by setting the output *o_read_req*. After the RAM controller completes the read operation (indicated by the input *i_read_done*), the unselected buffer is loaded with new data by setting the output *o_load_reg*.

After the next rising edge, the outputs *o_load_reg* and *o_read_req* must be disabled and the RAM address can be incremented. Now the controller is ready to process the next data request from one of the buffers.
The data is stored in the RAM in words of 16 bits (2 pixels). Since we are using a dynamic RAM, data has to be read in bigger bursts to mask the initial latency of opening an internal row. For this application, a burst size of 16 is chosen (32 pixels, 256 bits total). After each read operation, the address must be incremented by 16 to point to the next burst start point. After reading an entire frame (how many reads?), the address counter must be reset to the start address of the next frame.

To avoid forcing the renderer to change the image data of the currently displayed frame, a method called 'page flipping' is used. That means that the RAM is separated in two regions called pages. The start address of the first page is 0, while the second page starts at 262144 (0x40000).
While the first page is used to store the new data from the renderer, the frame that is currently displayed is read from the second page, and vice versa. When the renderer has finished a new frame (indicated by the signal *i_new_frame_ready*) the graphic output displays the rest of the current frame and then switches the page and displays the next frame. If the *i_new_frame_ready* signal is not set at the end of the frame, the old frame will be displayed again. On a reset, the first page (starting at address 0) should be selected.
The renderer itself waits until a successful page switch is signified through the output *o_page_switched* of the graphic buffer controller. The *o_page_switched* signal can be set back to 0 with the next RAM address increment after a page switch.

To implement the Graphic Buffer Controller, you must first design a state machine. You can use the template of the state chart given on the next page.

***Hints:*** If you want to increment the RAM-address, you can simply set a flag in your state-machine and increment the address in another process depending on that flag. To calculate the RAM-address, you should first examine the binary representations of the start addresses! Which bits do you have to actually change to increment the address or flip the page?

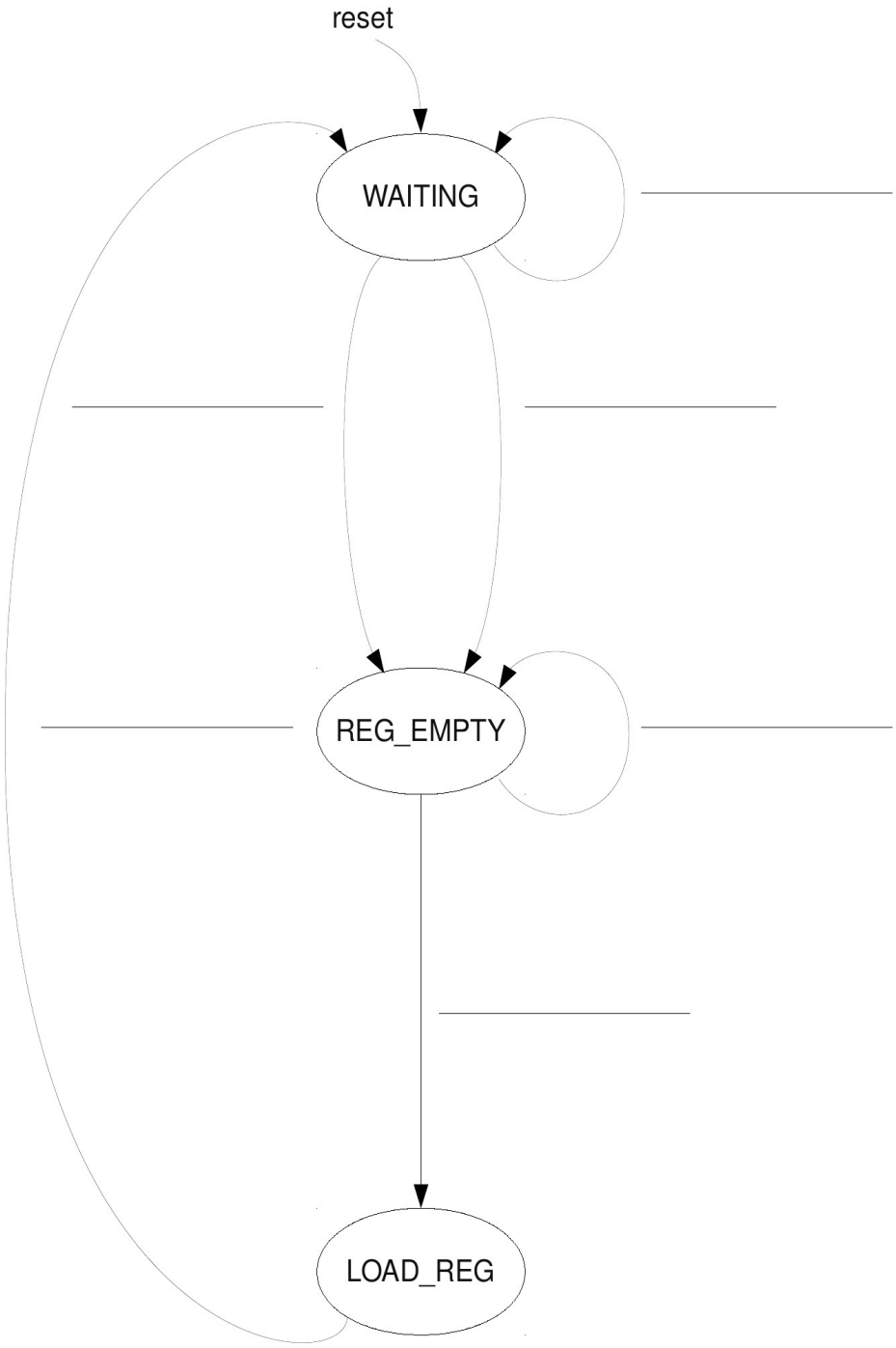



*Figure 2: State chart of the Mealy machine for the graphic_buffer_controller*

# Exercise 9:

### a) Graphic Output

The entity *graphic_output* groups the previous three modules together and provides the connections between them. It is also a multiplexer for the color component outputs *o_red*, *o_green* and *o_blue*. These outputs are fed from the currently selected register or set to 0 when the *in_active_region* signal is not set by the *sync_pulse_generator*.
The component declaration of the modules utilized should not be placed inside the architecture of *graphic_output*, but instead in a separate package file named *graphic_output_package.vhd*. This file will then be included in the *graphic_output* module.

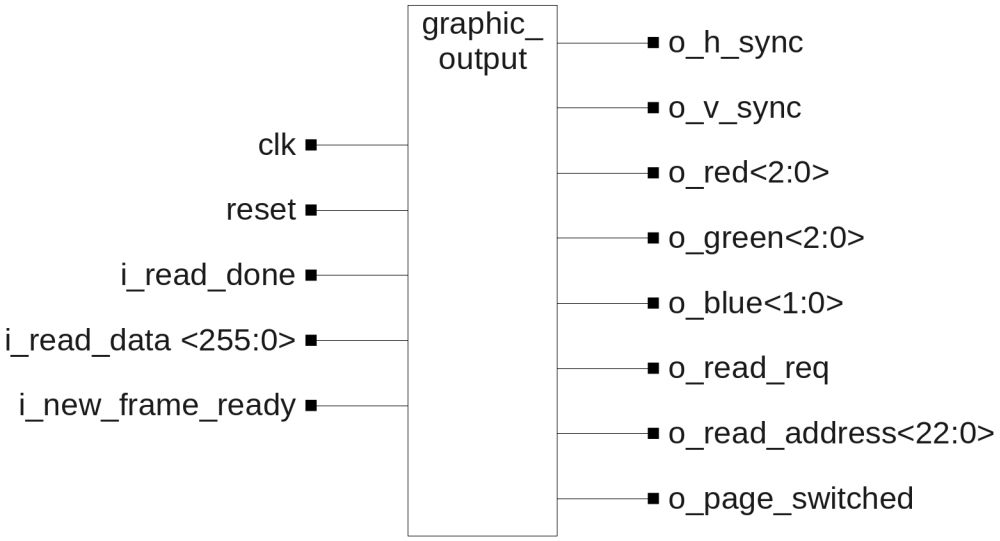



*Figure 1: graphic_output*

**Port declaration:**

- *clk* : in std_logic
- *reset* : in std_logic
- *i_read_done* : in std_logic
- *i_read_data* : in std_logic_vector (255 downto 0)
- *i_new_frame_ready* : in std_logic
- *o_h_sync* : out std_logic
- *o_v_sync* : out std_logic
- *o_red* : out std_logic_vector(2 downto 0)
- *o_green* : out std_logic_vector(2 downto 0)
- *o_blue* : out std_logic_vector(1 downto 0)
- *o_read_req* : out std_logic
- *o_read_address* : out std_logic_vector (22 downto 0)
- *o_page_switched* : out std_logic

**Generic declaration:**
The same generics as in the module *sync_pulse_generator* are used.

Your first task is to design the entity and architecture of the module *graphic_output* and to create the package file. Have a look at the schematic of the graphic_output module in the overview and think about all the interconnections before writing any VHDL code.

Since you should have previously tested the functionality of the sub-modules of graphic_output, you must now only test the interconnection of these modules and their interaction. Most importantly, the multiplexing between the two buffers must be tested. Therefore, you should have a detailed look at the signals of the two buffers and the three data outputs.
This time, no vector file with stimuli for the testbench is provided, so you must create your own vector file called **input_data.vec**. This file should feed all the inputs of *graphic_output* with stimuli. Try to create a reasonable test case by carefully examining the function of the module before writing the vector file!

After a successful simulation, the entire module and all its sub-modules will be integrated into a test project, implemented on the FPGA and tested on a real monitor.

**b) Synthesis of graphic_output**

**Working with Xilinx ISE**

To synthesize your code, we use the FPGA design software Xilinx ISE.
To setup Xilinx ISE (in this lab, we use Xilinx ISE14.6) to run on the IDA application servers, run:

```
$ setup-lm xilinx ise14.6
$ ise &
```

ISE will start with the last project opened. If no project has been created yet, click on **File** => ***New Project***. Enter a project name and change the path to your exercise directory.

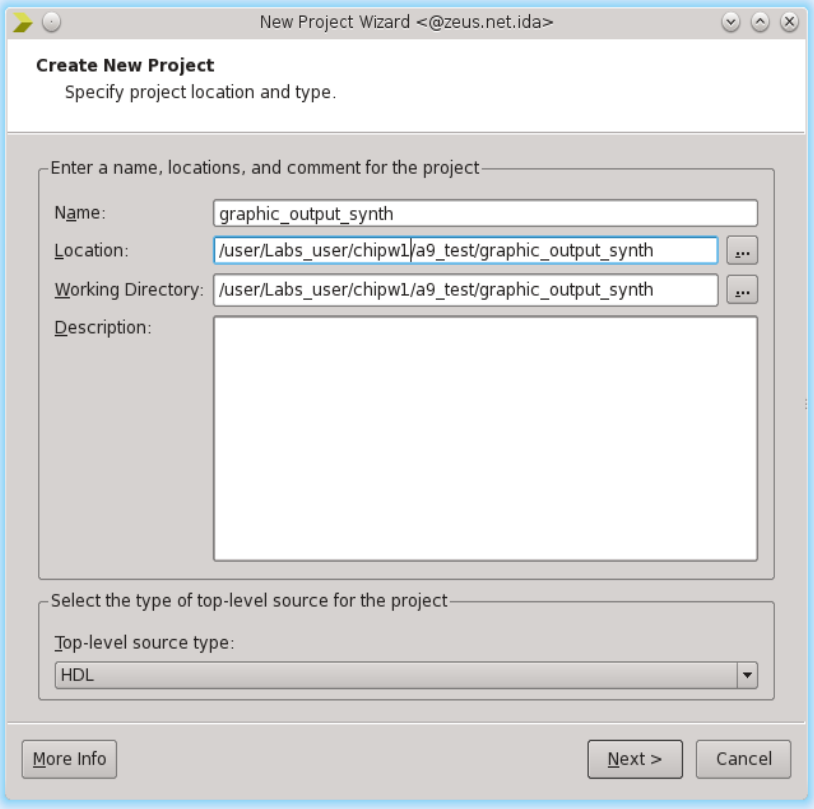



*Figure 2: Project Wizard – Create new project*

Now enter all of the required data for your FPGA and click ***Next***. Note that the device properties vary depending on the board you use. You can find the correct properties on your device. For the Nexys3 boards it should be: Device=XC6SLX16, Package=CSG324 and Speed=-3

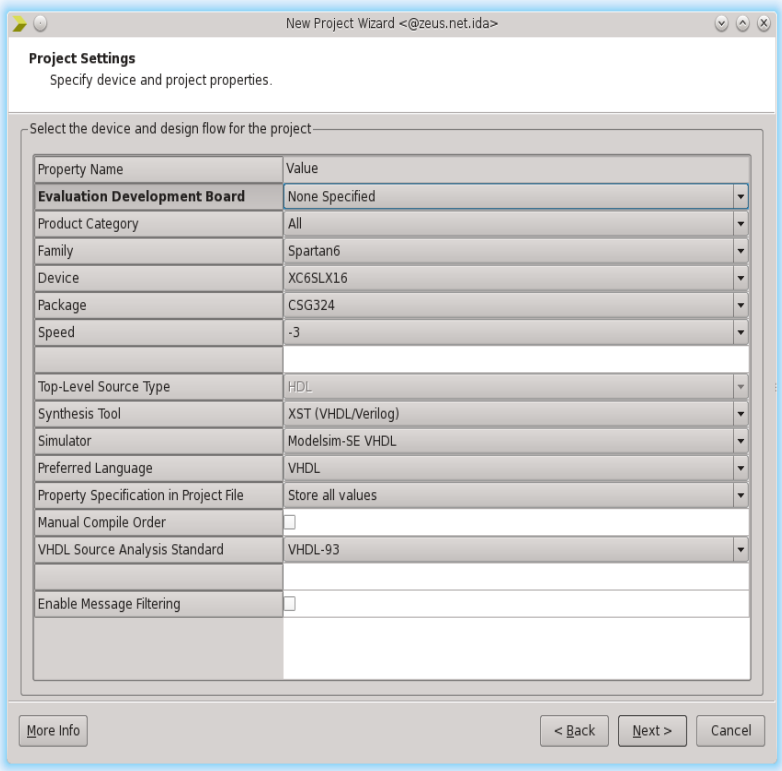



*Figure 3: Project Wizard - Device properties*

After finishing the project setup, you should see a plain project without any sources. First you have to add some sources of your previous exercises. In the main menu, select ***Project → Add Source*** and add the following files:

- graphic_buffer.vhd
- graphic_buffer_controller.vhd
- sync_pulse_generator.vhd
- graphic_output.vhd
- graphic_output_package.vhd

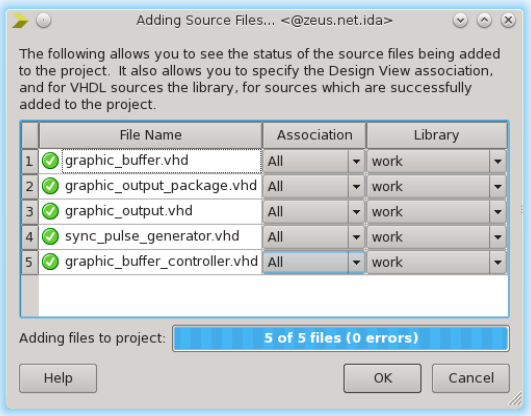



*Figure 4: Project Wizard - Add sources*

You should not use the “Add Copy of Source” option, so that you do not have to replace the file in the project directory once it is changed in its destination directory. Xilinx ISE will do a quick syntax check and if everything is fine, it will present you some green check marks.

To implement the complete test project, you must also add the following files:

- pixel_clk_generator.vhd
- rom.vhd
- graphic_output_synth.vhd
- graphic_output_synth.ucf

You can find these files in the folder of this exercise. Have a look at the *.ucf file, which contains the pin assignment of your FPGA. This file is known as a “constraints” file and is necessary to map the signals in your design to the physical FPGA inputs and outputs.

Now you see the Project Navigator window with all files (in fact you see the entities or instances of entities). Missing entities are marked with a question mark. The top entity is marked with three small squares. In most cases, the top entity is correctly chosen by ISE. If you have to change it, click on the entity you want to move to the top and right-click on it. Then select ***Set as Top Module***. To add files, right click in the Project Navigator and select either ***Add Source...*** for existing files or ***New Source...*** to create a new source file.

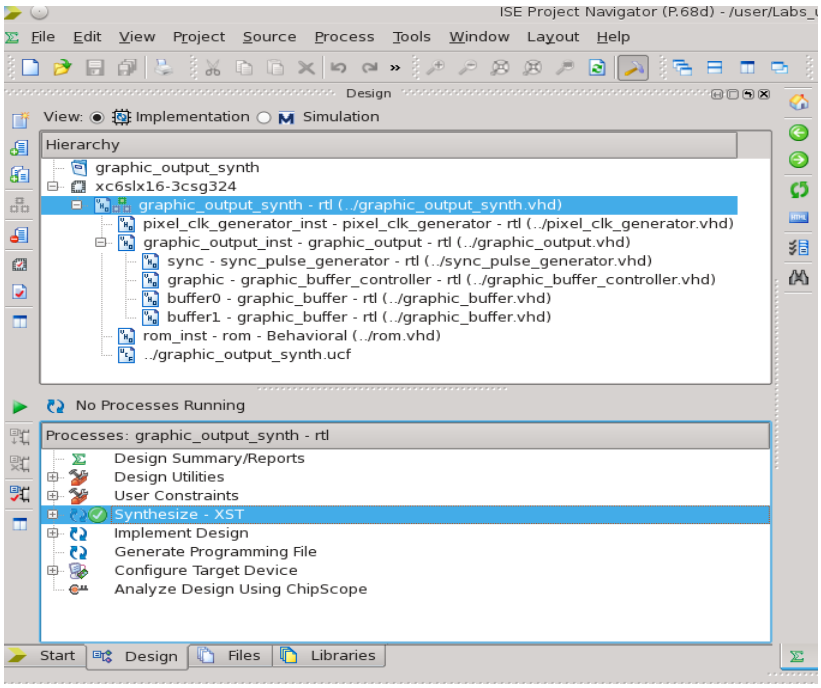



*Figure 5: Project Navigator window*

Once you have added all files, you must right click on ***Generate Programming File*** in the Processes window and select ***Properties***. Select ***Startup Options***, and change FPGA Start-Up Clock to JTAG Clock.

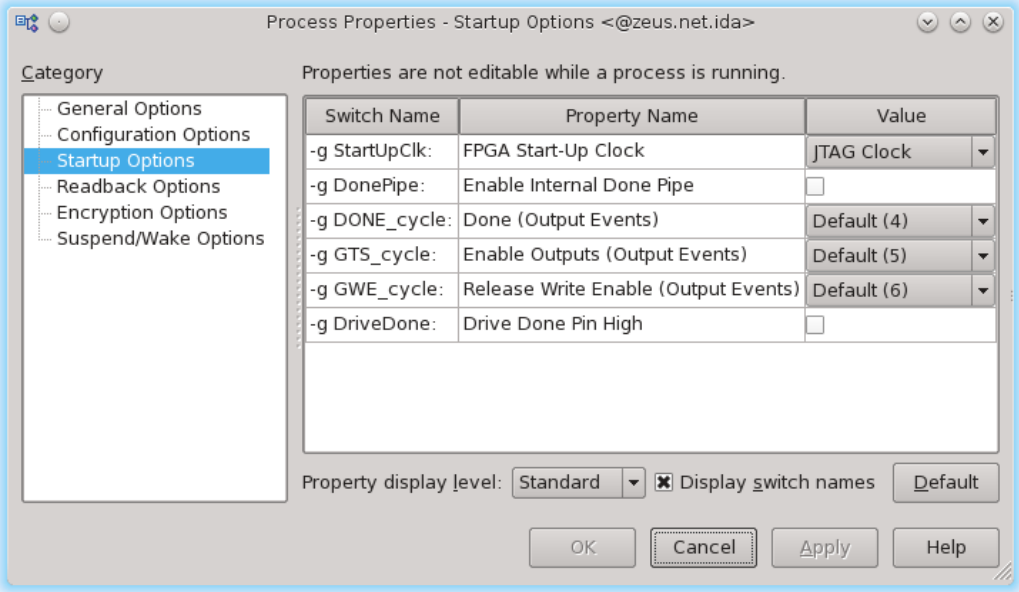



*Figure 6: Process properties*

To generate a programming file, simply double-click on ***Generate Programming File***, or you can run all three necessary steps one after another by clicking on them separately. At the end you should have a file named [*project name*].bit in your project directory. This FPGA bitstream file is then used to program the FPGA.

**Working with Digilent Adept tools**

Digilent Adept is used to program the FPGA. First thing to do is connect the FPGA board to the PC with the USB cable plugged into the micro-usb socket called "USB PROG" and connect the monitor via VGA. To use the programming tool, open a terminal window and navigate to your project directory where your bit-file is located. Check the connection by issuing:

```
$ djtgcfg enum
```

The output should look like:

```
Found 1 device(s)

Device: Nexys3
        Product Name:   Nexys3
        User Name:      Nexys3
        Serial Number:  210182475328
```

Then initialize the JTAG chain with:

```
$ djtgcfg init -d Nexys3
```

You will see something like:

```
Initializing scan chain...
Found Device ID: 34002093

Found 1 device(s):
        Device 0: XC6SLX16
```

To program the FPGA run (insert the correct name of your bit-file):

```
$ djtgcfg prog -d Nexys3 -i 0 -f [project name].bit
```

The result should be:

```
Programming device. Do not touch your board. This may take a few minutes...
Programming succeeded.
```

If the programming of the FPGA was successful, your monitor should display a test screen which alternates between Figure 7 and Figure 8. Otherwise, one of your modules is not functioning correctly.

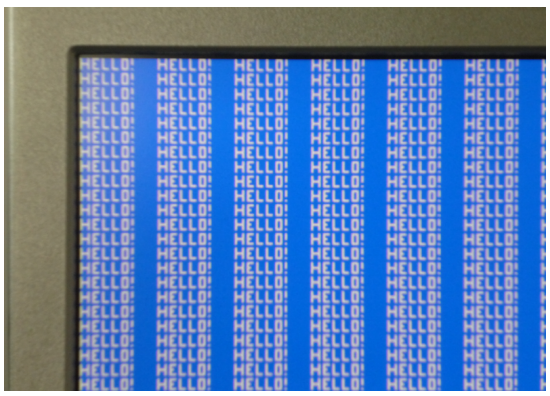



*Figure 7: First test screen*


*Figure 8: Second test screen*

# Exercise 10:

### a) RAM Controller

The communication between the FPGA and the external PSRAM is done by the module *ram_controller*.

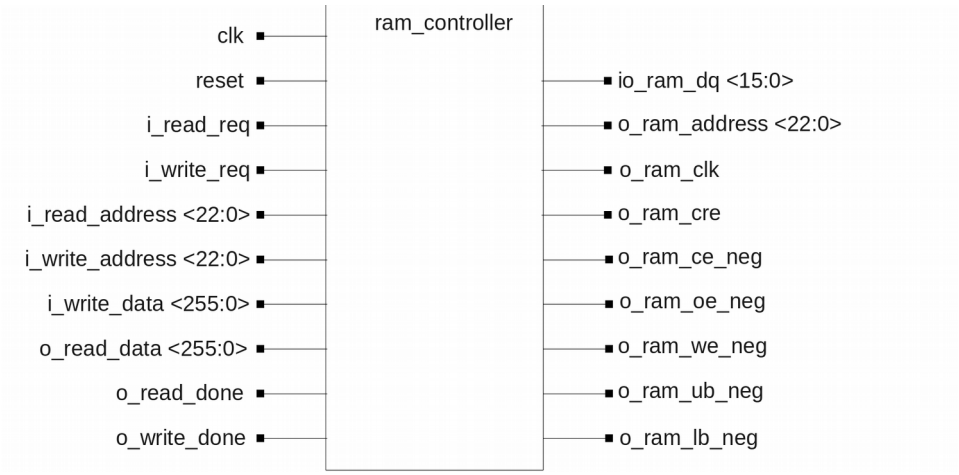



*Figure 1: ram_controller*

**Port declaration:**

- *clk* : in std_logic
- *reset* : in std_logic
- *i_read_req* : in std_logic
- *i_write_req* : in std_logic
- *i_read_address* : in std_logic_vector (22 downto 0)
- *i_write_address* : in std_logic_vector (22 downto 0)
- *i_write_data* : in std_logic_vector (255 downto 0)
- *o_read_done* : out std_logic
- *o_write_done* : out std_logic
- *o_read_data* : out std_logic_vector (255 downto 0)

- *io_ram_dq* : inout std_logic_vector (15 downto 0)
- *o_ram_address* : out std_logic_vector (22 downto 0)
- *o_ram_ce_neg* : out std_logic
- *o_oe_neg* : out std_logic
- *o_we_neg* : out std_logic
- *o_ram_ub_neg* : out std_logic
- *o_ram_lb_neg* : out std_logic

The PSRAM used on the board uses a DRAM architecture. This has some important drawbacks from a programmer's point of view: Since Data is addressed via rows internally, accessing or writing data is delayed by the time needed to open the corresponding row. Additionally, the capacitors used to store the information in DRAM lose their charge over time. This mandates periodic refreshes of all rows by opening them and writing back the result. If this refresh cycle collides with a read or write request from the FPGA, the access is delayed even further.

All together, this causes extremely long access times when the PSRAM is used as an asynchronous SRAM, since the on-chip controller assumes the worst case for every access (possible refresh collision and initial row opening latency). To overcome this problem, the PSRAM has a synchronous interface which masks the initial access latency by transferring larger bursts of data from the same row that was initially opened. This synchronous interface is more complex to design, so this part will be provided to you. However, the PSRAM has be be configured initially to use the synchronous interface via asynchronous signals.

Your task is to configure the PSRAM for synchronous operation.

The complete parameters are:

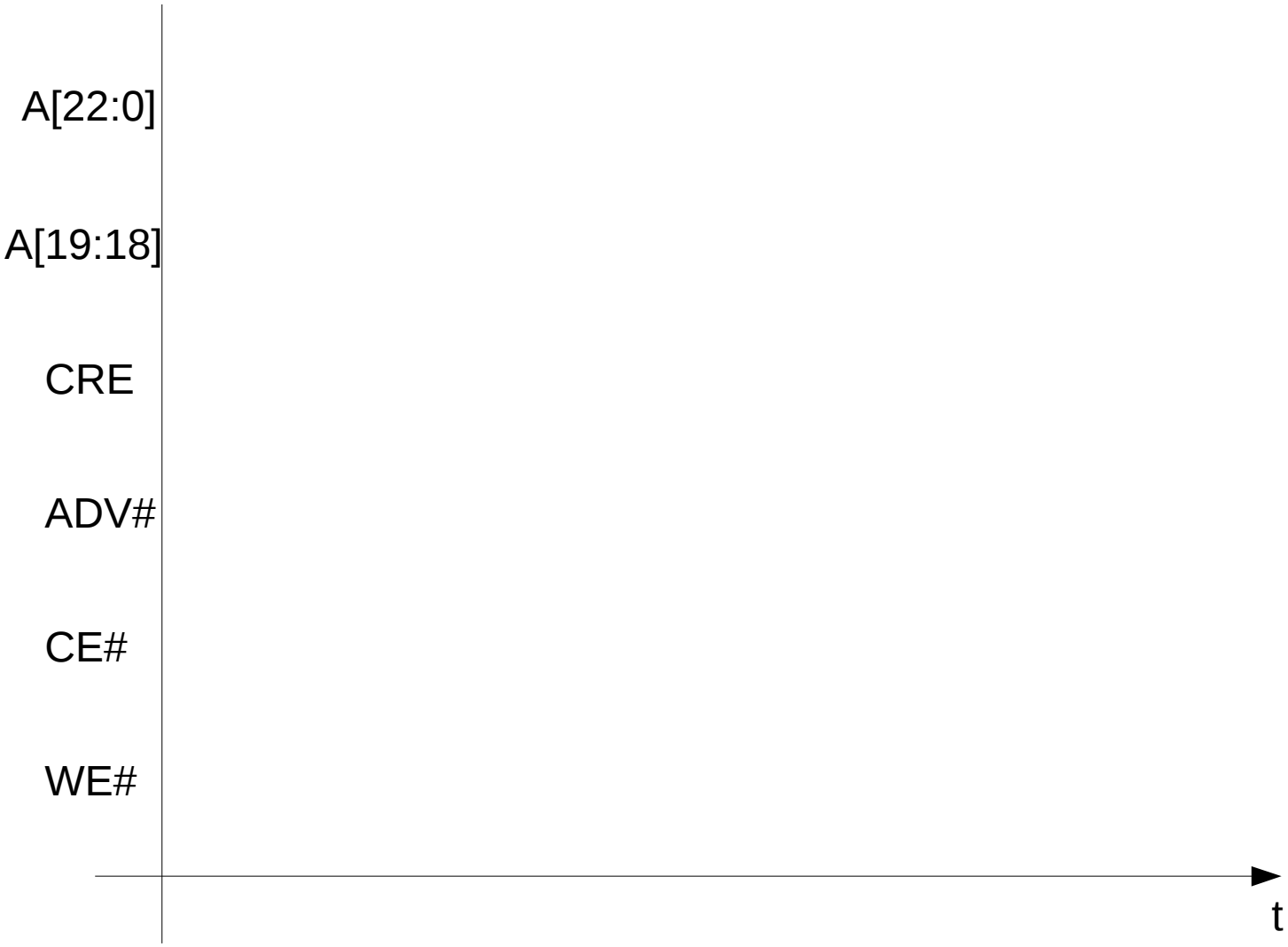
- synchronous operation
- fixed initial latency
- ... of 4 clock cycles
- wait active high
- ... asserted during delay cycles
- 1/2 drive strengh
- no burst wrap
- continous bursts until the end of row

The procedure for writing to control registers using CRE, the correct register and the bit positions as well as the timing can be found in the PSRAM's Datasheet in your documentation folder.

The controller should finish the configuration in as few clock cycles as possible. Note: Your “clk” input runs at 100Mhz, so one clock cycle equals 10ns.

To simulate the interaction between the controller and the SRAM, a simulation model of the SRAM is provided. The necessary Modelsim directives for compiling this Verilog model are already included in the .do-file.

Draw your chosen timing in the diagram on the following page.


A[22:0]
A[19:18]
CRE
ADV#
CE#
WE#
t

# Exercise 11:

### a) Pseudorandom Number Generator

This module creates a sequence of pseudo-random numbers.

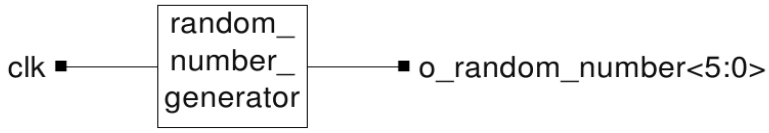



*Figure 1: random_number_generator*

**Port declaration:**

- *clk* : in std_logic
- *o_random_number* : out std_logic_vector (5 downto 0)

This is done using a linear feedback shift register. Such a register with a length of n bits creates a sequence of numbers between 1 and $2^n$-1, where each number appears only once before the entire sequence is repeated. (For detailed information on linear feedback shift registers, see [1]). The polynomial for 6 bits is:

**x(1) <= x(6) XOR x(5)**

**NOTE:** 0 is not a valid value!

This random number generator is later used in the Pong game to create a random number to be utilized for the ball movement. It will determine the initial direction and speed of the ball when the player serves. Additonally, it will be used to change the angle of the edge of the paddle every time the ball hits it.

Implement the randon number generator in VHDL, simulate its behavior and test it with the given testbench.

1 http://en.wikipedia.org/wiki/Linear_feedback_shift_register

**b) Event Trigger**

This module generates the so-called event signal for the ball and paddle movement. These signals are then used by the modules *ball_movement* and *paddle_movement*. Each time these signals become active, there is a possibility for the ball or paddles to move. This is done to divide the system clock and slow down the ball and paddle movement to a value the user can react to.
The time between two events can be changed by the generics *G_PADDLE_EVENT_INTERVAL* and *G_BALL_EVENT_INTERVAL*. These generics define the number of clock cycles between two events. When this time has elapsed, the corresponding signal must become active for one clock cycle.

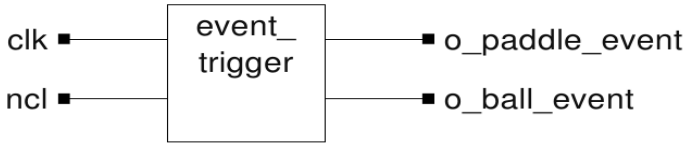


*Figure 2: event_trigger*

**Port declaration:**

- *clk* : in std_logic
- *reset* : in std_logic
- *o_paddle_event* : out std_logic
- *o_ball_event* : out std_logic

**Generic declaration:**

- *G_PADDLE_EVENT_INTERVAL* : integer
- *G_BALL_EVENT_INTERVAL* : integer

This module should then be tested with your own testbench.

**c) Input Decoder**

The input decoder scans the PS/2 data input of the FPGA board and reads data from the keyboard. Because the PS/2 port runs at its own clock frequency, there is already a process given which samples the PS/2 clock each system clock cycle and generates a signal to signify a negative PS/2 clock edge. This signal is called *new_ps2_data*. When it becomes active, the value of the input i*_ps2_data* is valid.
Now your task will be to implement the rest of the input decoder. It should support the following features:

- recognition for pressing a key and setting the corresponding output
- recognition for releasing a key and disabling the corresponding output
- a time-out function for the case that a scan-code is not completely transmitted
- a detection if both movement keys for one player are pressed simultaneously to prevent the up and down outputs of one player from being active at the same time.

A communication from the FPGA board to the keyboard is not necessary. Received data which can not be associated with the pressing or releasing of a key can be ignored. All outputs should be active as long as the corresponding keys are pressed.

For information on the PS/2 protocol, see the FPGA evaluation board manual[2].

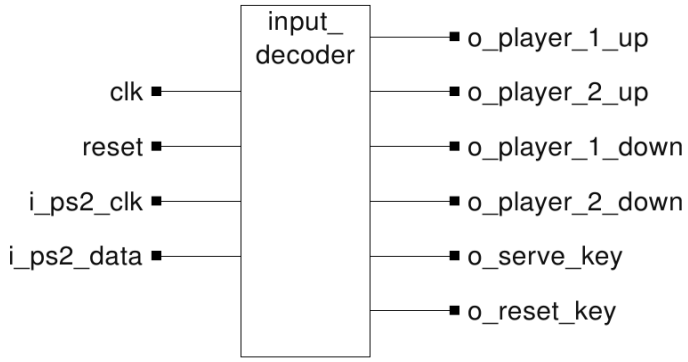



*Figure 3: input_decoder*

**Port declaration:**

- *clk* : in std_logic
- *reset* : in std_logic
- *i_ps2_clk* : in std_logic
- *i_ps2_data* : in std_logic
- *o_player_1_up* : out std_logic
- *o_player_2_up* : out std_logic
- *o_player_1_down* : out std_logic
- *o_player_2_down* : out std_logic
- *o_serve_key* : out std_logic
- *o_reset_key* : out std_logic

2 http://www.digilentinc.com/Data/Products/S3BOARD/S3BOARD-rm.pdf

# Exercise 12:

### a) Pong Completion

The last task is to implement the complete Pong game. Therefore you have to set up an ISE project with the following files you designed:

- random_number_generator
- sync_pulse_generator
- graphic_buffer
- graphic_buffer_controller
- graphic_output
- ram_controller
- event_trigger
- input_decoder

To start ISE from the application servers, run:

```
setup-lm xilinx ise14.6
ise &
```

You can find all the missing files you must include for the complete Pong game in the directory of this task. You can also find the *.ucf file you will need.
If you do not want to use the ISE GUI to set up your project, you can find a .tcl file (setup_pong_synth.tcl) in the syn directory. Like the *.do files in ModelSim, these files are used to automate your synthesis. You can start this file with the ***source*** command in the ISE command shell. If you used the correct directories, it should automatically include all the necessary files and set up the project. You can now start the synthesis with the GUI, or also with a script. The commands required to start the synthesis using a script can be easily found in the ISE help.

If your game is running, the main task of this lab is finished.