1 Introduction

In this practical evaluation we are going to design a simplified variation of the game Yahtzee with only four dice. For this exercise two modules are already provided, namely:

- A psuedo-random number generator.
- The top level with the game controller and the clock divider.

Each of these two modules will be explained in detail later on.

2 General restrictions

The system has to be realized in the template that you can download from digsys.epfl.ch (please use a VPN-connection). You may add as many VHDL-files as you require as long as you make two seperate files for them, namely (1) `"your_module_name"_entity.vhdl` that describes only the entity of the module, and (2) `"your_module_name"_behavior.vhdl` that describes only the architecture of the module. You are not allowed to rename the files contained in the template, remove/add signals to their entities, or change the behavior provided in the modules that are given. Furthermore, all fliplops used in your design must be connected to the system clock (e.g. it is not allowed to use derived and/or gated clocks), unless otherwise specifically noted. Finally, the system uses a synchronous reset, unless specifically mentioned otherwise. Make sure that you do not use spaces and/or special characters like à in your file names!
Your solution submitted to digsys.epfl.ch should be a zip file that ONLY contains:

1. All vhdl-files that are required to build your system (including those of the above described template).
2. A ".sof" file called toplevel.sof that can be used to download on a GECKO-board (generate the video).

Note that it is your responsability that all VHDL-files are syntactically correct, simulate, and are able to produce a .sof file. We will not modify any of the files in case of errors, and will not generate a sof-file if it is missing.
Important:

- From experience the most common errors made are *typo's*. Do yourself a favor and control all the file names before submitting your solution.

- If you were unable to realize one of the sub-blocks requested by one of the questions, remove these blocks from the submitted .zip file, they will then be automatically replaced by a reference solution (of course the points for these modules will not be abutted). You have, however, to generate a .sof file (put some constants in these modules, or for example a counter to be able to generate a .sof file yourself).

3 Provided blocks

To be able to realize the whole system two blocks are provided. This section describes their functionality.

3.1 The psuedo-random number generator

The psuedo-random number generator is an external ASIC connected to our FPGA. It uses a protocol that is called the I^{2}S protocol. For this exam we provide you with an "emulator" of this ASIC in VHDL. The I^{2}S-protocol used is shown in a separate file called *rng.pdf*. The signals `CSin` and `CLKin` are signals that we have to provide to the module and should be outputs of flipflops (to prevent hazards). As reaction on these signals the psuedo-random number generator (ASIC) will provide four data bits that represent the random number generated. The first bit (`A` resp. `E`) provided is the *MSB* of the random number. The random number is sign and magnitude coded. Besides these three signals, this module also has an asynchronous reset input that needs to be '1' at system startup for at least 1 ms to be able to use the module. Finally the maximal clock frequency allowed for the input `CLKin` is 128kHz.

3.2 The top level

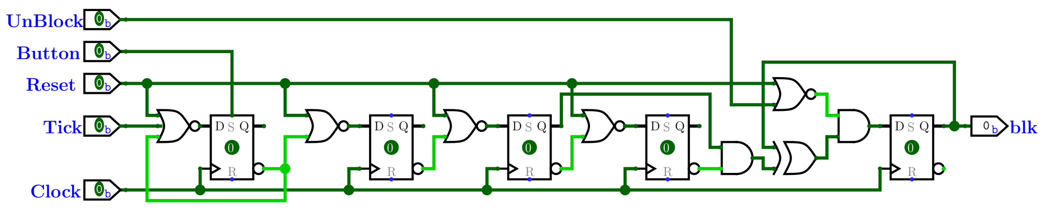
See section 6.2

4 Let's start designing

First we are going to design some blocks that we require to be able to build-up the whole game.

4.1 Button logic

The first block that we need is an interface module to the different buttons that we need for the game. The circuit for this module is shown below:



Exercise 1:

Write the complete VHDL description for this module that is called `ButtonLogic` (hence `ButtonLogic_entity.vhdl` and `ButtonLogic_behavior.vhdl`).

Requirements:

- All combinational logic elements must be described in one/multiple implicite process(es).
- All sequential logic elements must be described in one/multiple explicite process(es).
- All inputs and outputs must have exactly the same name as shown in the circuit.

Note: You are allowed to use components, but it is not a requirement.

4.2 Display module

To show the number of pips (number of dots) of the dice we use a 7-segment display. The input of this block is a 3-bit binary encoded binary number called `pips`. The output of this block is a 7-bit vector `displ` representing the 7-segment of the display, where the MSB is segment a and the LSB is segment g (in alphabetic order). The segment shows the number of pips when the input pips is in the range 1...6, otherwise it is blank (does not display anything). A segment lights up when the corresponding bit in the output `displ` is 0.

Exercise 2:

Write the complete VHDL-describtion of this completely combinational logic circuit called `Decoder` (hence `Decoder_entity.vhdl` and `Decoder_behavior.vhdl`).

Requirements:

- The input and output of this block must have exactly the names as listed above.
- The input must not be of type `std_logic` or `std_logic_vector`, but must have the correct interpretation.
- See TP1 on moodle on how the numbers 1..6 should be displayed on the seven-segment display.
- You have to use an implicite process with the `WITH (...) SELECT ....` construct to describe the functionality requested.

4.3 Shift-register

The second block is a 4-bit shift-register that we require to clock in the bits of the random-number-generator (see *rng.pdf*). Besides the inputs `Clock` (the system clock), `Reset` (the power-on reset that forces the shift-register to 0),and `Din` (this is the value of `Dout` in *rng.pdf*), it has also an input called `Tick`. When this input `Tick` is 1 the shift-register shifts one position, where the shifted-in value is the value of `Din`. If the `tick` is 0, the shift-register retains it's current value. The output `Value` of the shift-register is a vector of 4 bits that represents the number coming from the random-number-generator, where the LSB of the number is presented on bit 0, and the MSB of the number on bit 3.

Exercise 3:

Write the complete VHDL descriptions for this module that is called `ShiftToParallel` (hence `ShiftToParallel_entity.vhdl` and `ShiftToParallel_behavior.vhdl`).

Requirements:

- The inputs and outputs of this block must have exactly the names as listed above.
- The output `Value` must be of type `std_logic_vector`.

4.4 A simple state machine

To control the outputs `CSin` and `CLKin` (see *rng.pdf*) we need a finite state machine (FSM). The sequence of this FSM is shown on the bottom *rng.pdf*. Besides the inputs `Clock` (the system clock), `POR` (the power-on reset that forces asynchronously the FSM into the state `idle`), it has two other inputs, namely:

- `req`: This signal is exactly one system clock cycle high, and indicates the request of a new random-number (see *rng.pdf*).
- `Tick`: This signal indicates when 1 that the FSM has to go to the next state, and when 0 that the FSM should stay in the current state. This signal is also exactly one system clock cycle high.

You may assume that when `req` is high also `Tick` is high.

Exercise 4:

Draw the state-diagram of this FSM. Call the file `FSM.png` for a PNG image, `FSM.jpg` for a JPG image, or `FSM.pdf` for a PDF file.
To make your life easier, you may indicate the signal `Clock` by a `C` in your state diagram, the signal `POR` by `P`, the signal `Tick` by `T`, and the signal `req` by `r`.
Requirement: Name the states identical to the states shown in *rng.pdf*.

Now that we have the state-diagram we need to look at the outputs of the FSM. The outputs `CSin` and `CLKin` must be outputs from flipflops that are clocked by the system clock. Their behavior is shown in *rng.pdf*. Furthermore, this FSM has following outputs (that are activated exactly one system clock cycle):

- `doShift`: This signal indicates that a new bit should be shifted into the shift-register, and should be activated in the middle of the period that the bit is available (hence in *rng.pdf* in the middle where `A` is provided by the random-number-generator).
- `done`: This signal indicates that we have a new random number, and should be activated at the same time we go back to the state `IDLE`.

Exercise 5:

Write the complete VHDL-description of this finite state machine called `requestRandom` (hence `requestRandom_entity.vhdl` and `requestRandom_behavior.vhdl`) according to the specifications.
Requirements:

- You have to use a `TYPE` to describe the current and next state, where they correspond to the names shown in *rng.pdf*.
- The inputs and outputs of this block must have exactly the names as specified above.

5 Complexity increases

In this part you are free to design as you want, as long as you keep the specifications and it is a valid circuit that works on hardware (GECKO4Education).

5.1 Correct numbers?

As the random-number-generator can generate numbers that are different from 1...6, we need to check that the number generated is in the range 1...6. For this we need a circuit/controller that checks when we read in the random number if it is a valid number, and if not, we need to generate another random number upto the moment that it is in the range 1..6.
Specifications:
The inputs of this block are:

- `Clock`: The system clock
- `Reset`: The power-on reset that forces the module in the initial state at start-up.
- `RollDiceIn`: This signal is exactly one system clock cycle high and indicates that we require a new roll of the dice (read a new random-number in range).
- `ShiftValue`: The 4-bit value (`Value`) coming from the shift-register described in section 4.3.
- `Tick`: This is the same signal as `Tick` in section 4.4.

- `Done`: This is the same named output of the FSM described in section 4.4.

The outputs of this block are:

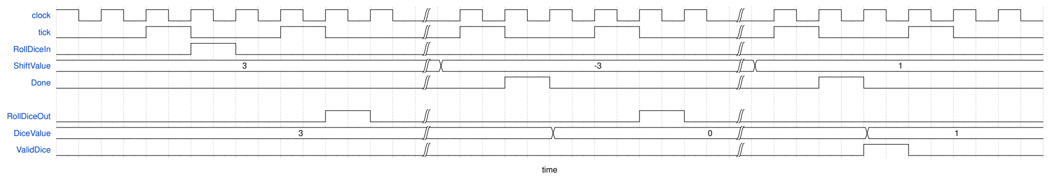
- `RollDiceOut`: This signal is exactly one system clock cycle high and indicates that a new random number needs to be read in (it will be connected to the signal `req` of the FSM described in section 4.4).
- `DiceValue`: This is a 3-bit binary encoded binary value that will be connected to the input `pips` of the module of section 4.2 and hence should be of the same type. This value changes each time the `Done`-input is one by taking the binary representation of the input `ShiftValue`. In case the value of `ShiftValue` is out of range (hence smaller than 1 or bigger than 6) it outputs a decimal 0.
- `ValidDice`: This signal is exactly one system clock cycle high and indicates that the current value on the output `DiceValue` is a correct value and the game controller can continue.

The functionality can be described as following:

1. As long as the input `RollDiceIn` is not activated, this module does nothing and retains in its current state. Once `RollDiceIn` is activated, the following sequence is performed:

2. The module waits until the input `Tick` is active.
3. Once the input `Tick` is active the module will request a new random number by activating the output `RollDiceOut` for one system clock cycle (hence `RollDiceOut` is active exactly the same time as `Tick`).
4. Then the module waits until a new random number is read-in by waiting for an activation of the input `Done`.
5. When `Done` is activated it will update the output `DiceValue` with the value specified above.
6. If the random-number is out of range it will request a new random-number by going to step 2 listed above.
7. If the random-number is in range it will activate the output `ValidDice` for one system clock cycle and go to step 1 listed above.

At system start-up all memories are resetted to 0. A sample timing diagram is shown below:



Exercise 6:

Design a module called `DiceControl`, (hence `DiceControl_entity.vhdl` and `DiceControl_behavior.vhdl`) with the specifications listed above.
Requirements: All inputs and outputs have to have the same names as listed above and you are not allowed to add or remove inputs/outputs.
Don't forget: The encoding that the random-number-generator provides to you (see section 3.1).

5.2 Sorting the dice

After the dice have been thrown three times the game controller will raise a signal called `sort` that will display the four dice in a sorted manner where the dice with the smallest and the largest number of pips are shown to the right and left, respectively. We are going to implement this completely combinational circuit by using a variation of the *merge-sort* algorithm. The idea is the following:

1. Split the set of four dice into two subsets of two dice each: $left$ and $right$, then for each subset determine the smaller die (S_{left}, respectively S_{right}) and the bigger die (B_{left}, respectively B_{right}).

2. The smaller of S_{left} and S_{right} is the die with the smallest number of pips.

3. The bigger of B_{left} and B_{right} is the die with the biggest number of pips.

4. Sort the bigger of the small set (S_{left},S_{right}) and the smaller of the big set (B_{left},B_{right}) to fill in the last two places.

This module has as inputs:

- `sort`: If this signal is 0 the dice are shown in normal order (hence `DiceOut1` = `DiceIn1`, etc.), if this signal is 1 the dice are shown in a sorted manner as described above.
- `DiceIn1`: This is the 3-bit value for die 1 coming from the block described in section 5.1.
- `DiceIn2`: This is the 3-bit value for die 2 coming from the block described in section 5.1.
- `DiceIn3`: This is the 3-bit value for die 3 coming from the block described in section 5.1.
- `DiceIn4`: This is the 3-bit value for die 4 coming from the block described in section 5.1.

This module has as outputs:

- `DiceOut1`: This is the 3-bit value going to the right most seven-segment display.
- `DiceOut2`: This is the 3-bit value going to the second from the right most seven-segment display.
- `DiceOut3`: This is the 3-bit value going to the third from the right most seven-segment display.
- `DiceOut4`: This is the 3-bit value going to the left most seven-segment display.

Exercise 7:

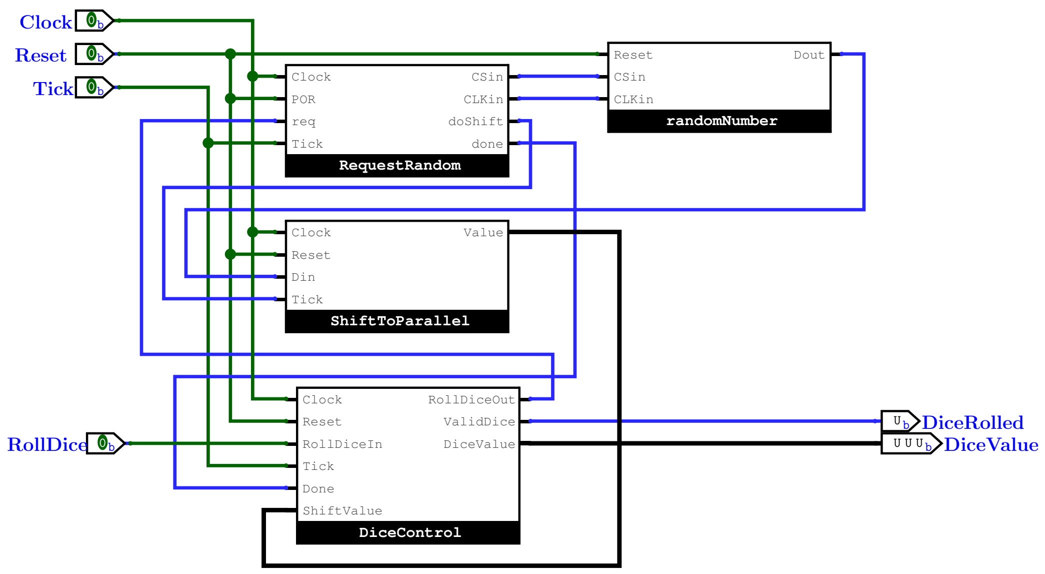
Design the module called `SortIt` (hence `SortIt_entity.vhdl` and `SortIt_behavior.vhdl`) that implements the above functionality.
Requirements: All inputs and outputs have to have the same names as listed above and you are not allowed to add or remove inputs/outputs.

6 Putting it all together

Now we are going to put it all together.

6.1 A single die

To realize one die, we are going to put all the components together that we realized so far. The functionality to be realized is shown below:



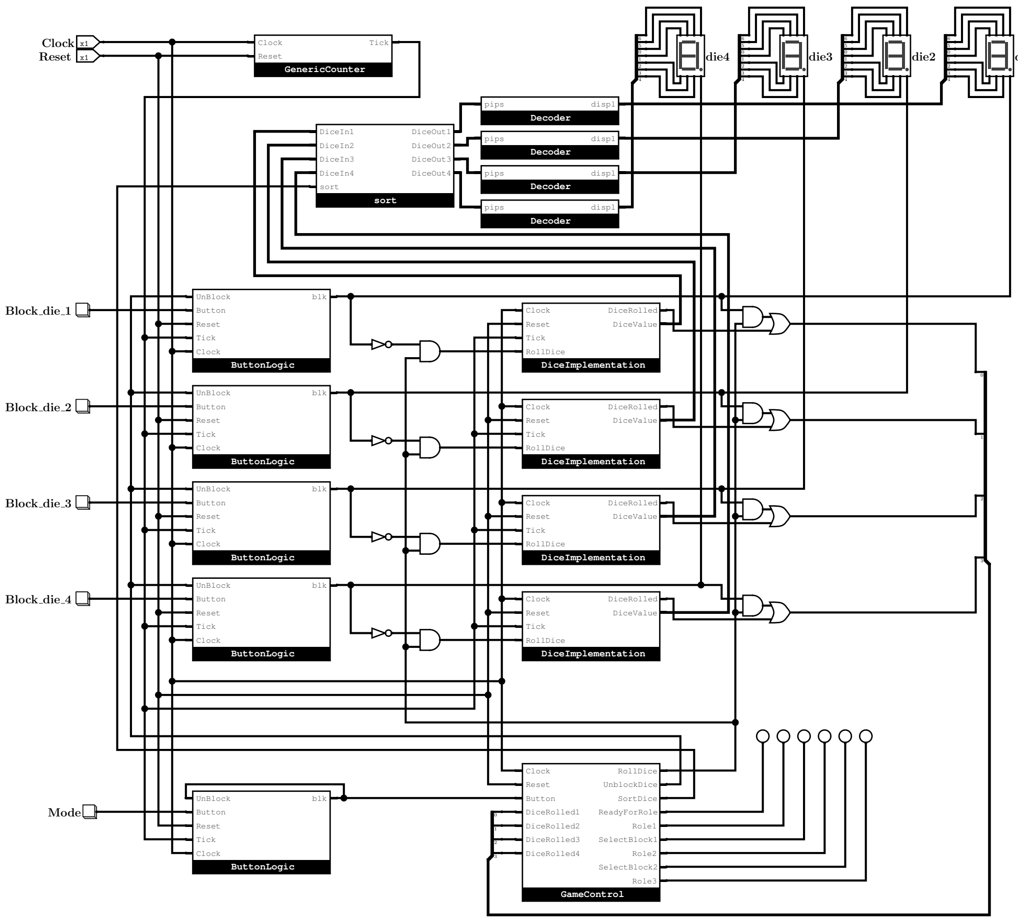
Exercise 8:

Write the complete VHDL description of this block called `DiceImplementation` (hence `DiceImplementation_entity.vhdl` and `DiceImplementation_behavior.vhdl`).
Requirements:

- The generic parameter `InitialSeed` of the block randomNumber must be provided as generic parameter of this block.
- All in- and outputs must have exactly the same names as shown above and you may not add/remove in- and/or outputs.

6.2 toplevel

The top-level was designed in and generated by *logisim-evolution* and is the following implementation:



Of course with automated code there are some small problems. The problem with this code is that the generic parameters on the top-level module have not been assigned the correct values. There are five generic parameters that you need to adapt. The system clock is 12 MHz. The parameters that need to be adapted are:

- The generic values of the `GenericCounter`. These are both set to 0 by *logisim-evolution*. We want that the frequency of the `Tick` output is 128Hz and use just the number of bits required to represent this divider number.

- Each of the `DiceImplementation` block has a generic `InitialSeed`. If we use the same number for these, the 4-dice would always have the same number of pips. Hence each of these four blocks requires another `InitialSeed`.

Exercise 9:

Modify the toplevel file such that the system runs at 128Hz, and that the `InitialSeed` for dice1=25, dice2=-25, dice3=123456, and dice4=765432. Afterwards generate the .sof file to be able to generate the video. Note that the tcl file is provided by `pins.tcl`.