DSD EXERCISE

SEQUENTIAL-STYLE COMBINATORIAL DESIGNS IN VHDL

AARHUS UNIVERSITY
SCHOOL OF ENGINEERNG
PHM, CEF

JUNE 2018

Document History

2015-10-26: PHM, Initial version.
2017-10-18: CEF, fixed exercise numbering.
2018-02-23: CEF, converted to LaTeX.
2018-02-23: CEF, fixed errors in generic interface.
2018-03-26: CEF, fixed wrong 'multiplier' text in binary-to-7-segment exercise.
2018-06-07: CEF, fixed missing generic exercise.
2018-06-07: CEF, fixed figure placements.

Goals

Designs without memory are also known as combinatorial. Combinatorial designs can be expressed with truth-tables and this is also how they are implemented in the FPGA. The goals for this exercise are:

- Get experience with sequential designs and if / case statements.
- Get even more experience with latches in your design,
- Get experience with loop statements and Generics if you are up for it.

These exercises will get around different sequential code constructions and it starts out with case statements. You may skip sub exercise (5) if you wish to try out sub exercise (6).

1 Binary to 7-Segment Decoder Using "CASE"

In this step it is your task to implement a binary to seven segment decoder using case statements.

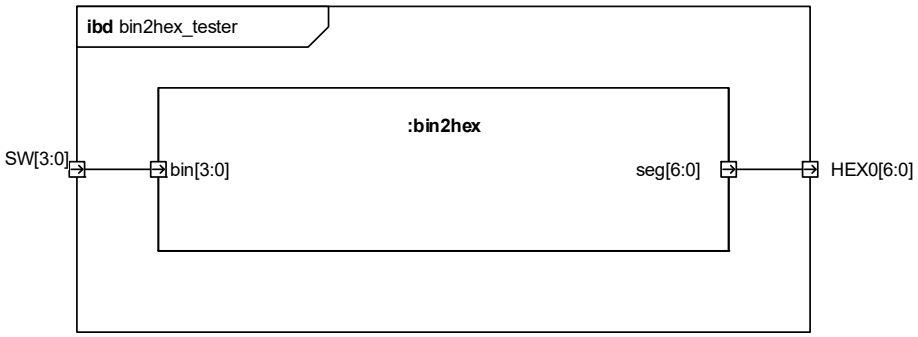

Fig. 1: *Binary to 7-segment-decoder.*

a) Create a binary-to-7-segment converter component with the interface depicted in figure 1. The component must be implemented using a case statement and include hexadecimal numbers 0..F. See the "DE-2 User Manual" for details about the seven-segment displays (named HEX displays in the text). Also note that the segments are active-low.

b) Compile and test the entity on the DE2-board. How is the design implemented according to the RTL- Viewer? Is structure different from the previous "With-select" implementation?

2 Guess Game

The purpose of the game is to guess a secret number. After entering the guess, a press on a button will evaluate the result as "Hi", "Lo" or "–". The secret number is entered manually by your opponent.

a) Design a game with interface shown in figure 2 and source code table 1 and the

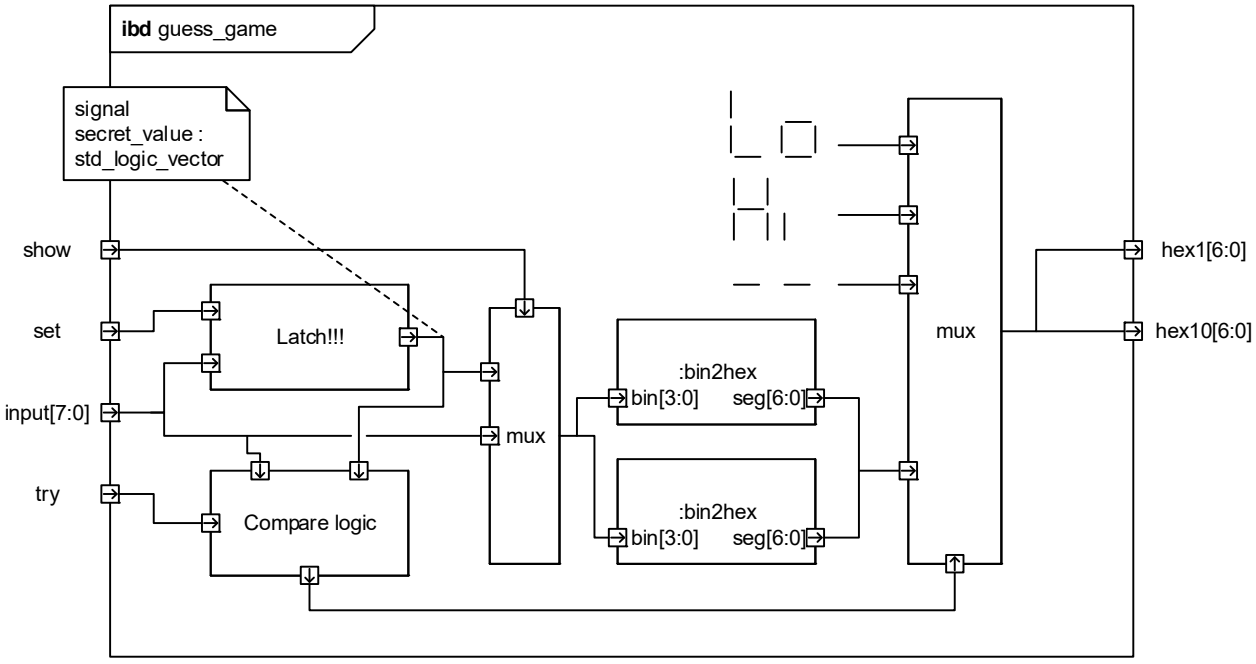

Fig. 2: *Guess game IBD.*

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

entity guess_game is
    port(inputs : in std_logic_vector(7 downto 0);
         set   : in  std_logic; -- Set predefined value
         show  : in  std_logic; -- Show predefined value
         try   : in  std_logic; -- Evaluate guess
         hex1  : out std_logic_vector(6 downto 0); -- 7-seg ones
         hex10: out std_logic_vector(6 downto 0)  -- 7-seg tens
    );
end;
```

Tab. 1: *Guess game interface.*

following functionality:

i) With no keys pressed, the displays show the current input value.

ii) With "Set" button pressed, the input value is stored as the secret number.

iii) With "Show" button pressed, the secret number is displayed.

iv) With "Try" button pressed, the guess is evaluated and the result is displayed as "Hi", "Lo" or "–".

The IBD in figure 2 shows the overall building blocks in the design.

b) Download and test your design on the DE-2 board. It could be beneficial for you to create a `guess_game_tester` that maps: inputs to switches (SW) / set, show and try to keys (KEY) / hex1, hex10 to hex displays (HEX0 HEX1). Note in your compilation output if you have any latches generated. If so, why (relate to design)? Can we avoid them?

3 Two Player Guess Game

This version of the game supports two players, and must be able to remember the secret number of the game not being played.

a) Design the two-player game as depicted in figure 3. Use if/case/when/with-select as appropriate. Argument why you have chosen one solution over others.

b) Download and test your design on the DE-2 board. "Player" should be connected to a switch (SW). Latches? How many and why?

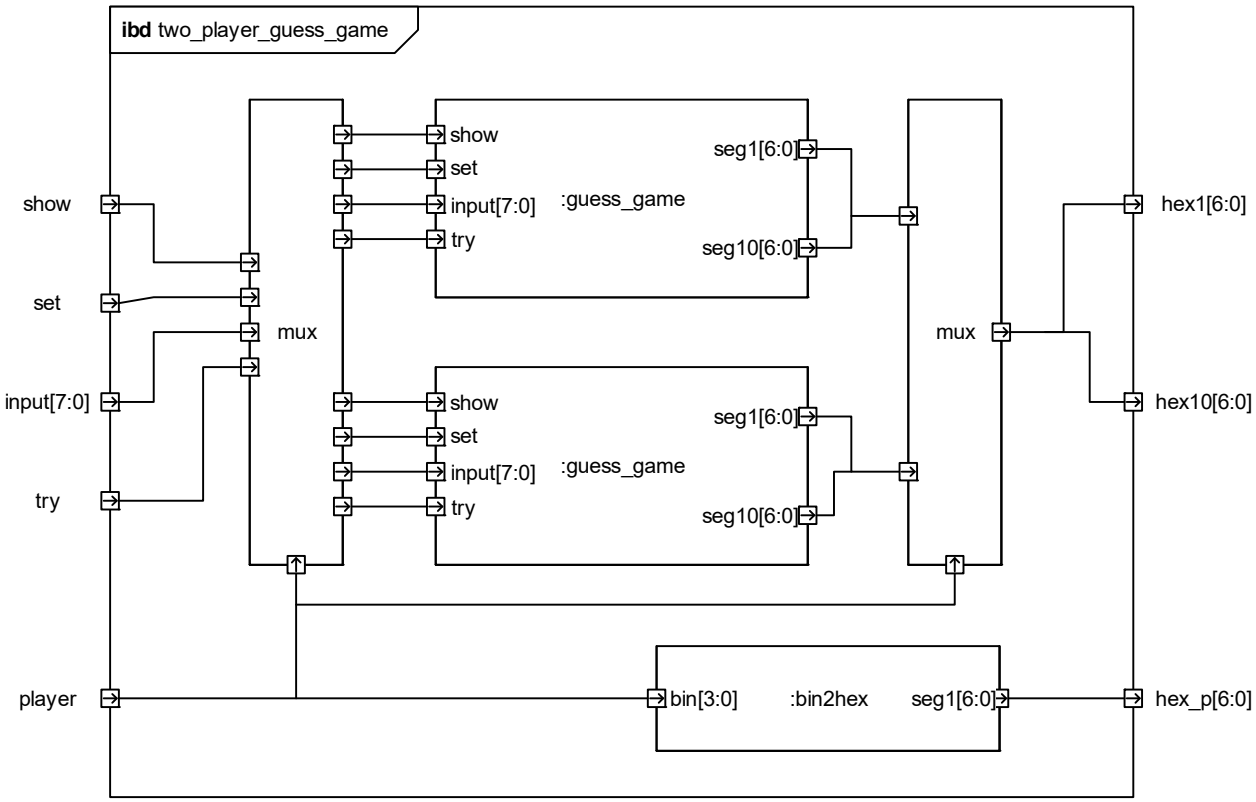

Fig. 3: *Two-player guess game.*

4 8-Input NAND Gate

This little exercise lets you work with non-/associative operators and loops. Remember that loops are used to generate multiple instances of hardware! Loops are not software loops that are iterated through at run-time.

a) Design the 8-input NAND gate depicted in figure 4 using a loop statement.

b) Perform a functional simulation of the design to verify its functionality.

c) **(OPTIONAL)** It is possible to create scalable modules using Generics (VHDL for Engineers: 15.9). Generics allow you to pass parameters to modules as shown in source code table 2. This example shows a module that takes the parameter (Generic) 'bits' and sets its port sizes accordingly. In the top-level design it is now

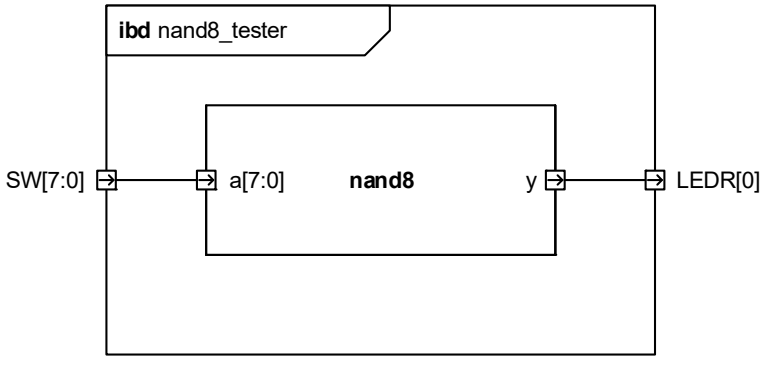

Fig. 4: *8-input NAND*

possible to scale the number of bits in use. The integer value, ‘bits’, can also be used in the architecture for loops (for examlpe a scalable NAND-n), in if-statements a.o.

Modify your 8-bit NAND gate to use a generic that lets you set the input bit width in the top-level design. Compile and test on the DE-2 board.

5 Count Ones (OPTIONAL, if you’d rather try (6))

This exercise lets you count logical ‘1’ in an array and output it on a 7-segment display to get some more experience with loop statements.

a) Design a process to implement the functionality described above. Use the binary to 7-segment converter designed previously and the interface depicted in figure 5.

b) Download at test the design on the DE-2 Board.

c) (Optional) It is possible to create scalable modules using Generics (VHDL for Engineers: 15.9). Generics allow you to pass parameters to modules as shown in source code table 2. This example shows a module that takes the parameter (Generic) ‘bits’ and sets its port sizes accordingly. In the top-level design it is now possible to scale the number of bits in use. The integer value, ‘bits’, can also be used in the architecture for loops (ex a scalable NAND-n), in if-statements a.o.

Modify your 8-bit NAND gate to use a generic that lets you set the input bit width in the top-level design. Compile and test on the DE-2 board.

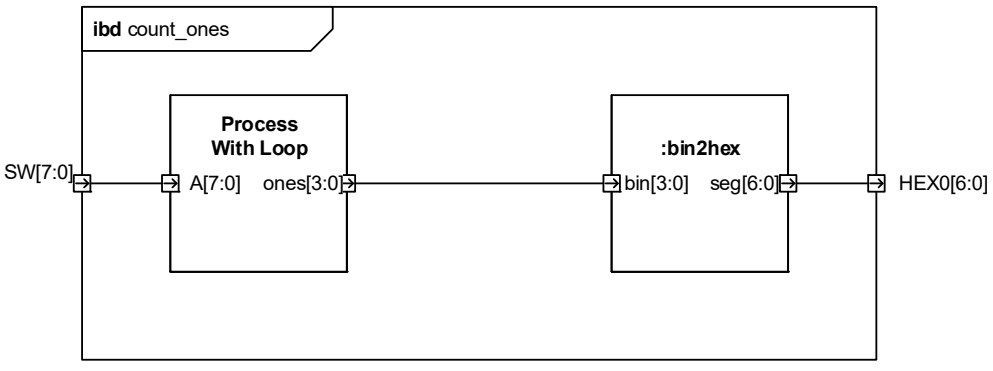

Fig. 5: *Count-ones IBD.*

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity generic_and is
5    generic (
6      bits : integer := 8 -- default value
7    );
8
9    port (
10     a, b: in  std_logic_vector(bits-1 downto 0);
11     y   : out std_logic_vector(bits-1 downto 0)
12   );
13 end;
14
15 architecture arch of generic_and is
16 begin
17   y <= a and b;
18 end;
19
20
21 library ieee;
22 use ieee.std_logic_1164.all;
23
24 entity generic_and_tester is
25   generic (
26     bits : integer := 7 -- Number of bits in tester,
27   );                    --   try an odd number
28
29   port (
30     SW   : in std_logic_vector(2*bits-1 downto 0);
31     LEDR : out std_logic_vector(bits-1 downto 0)
32   );
33 end;
34
35 architecture stuctural of generic_and_tester is
36 begin
37   u1: entity work.generic_and
38     generic map (bits => bits)
39       port map (
40           a => SW(2*bits-1 downto bits),
41           b => SW(bits-1 downto 0),
42           y => LEDR
43         );
44 end;
```

Tab. 2: *Generics in effect.*

6 Flash A/D Converter (OPTIONAL)

In this exercise you will build a 3-bit Flash A/D converter[1] using a network of resisters and multiple gpio inputs. You must use a loop-statement to convert the input state to a binary value.

a) You must start out by building some hardware. Create a 16-way ribbon-cable with a 40-pin connector in one end (Pin 1 is marked by a small arrow on the housing) and a 16-pin DIP socket in the other end, as shown in figure 6. You may just as well find a fumlebrædde from your drawers...

b) Build the circuit described in schematic in figure 7 on your fumlebrædde. Be careful to make the right connections to the ribbon cable. Note that the higher the input voltage created by the potentiometer, the higher the input voltages on the different gpio ports. An input voltage of approx. 2 volts will result in a logical '1' and below approx. 1 volt will result in logical '0'. Increasing the analogue voltage from 0V to 5V may resultin different threshold levels, than when decreasing the analogue voltage from 5V to 0V. As you increase the voltage, first `GPIO_0(4)` should go logical '1', then (2), then (0), (1), (3) and so forth.

c) Connect the 40-pin connector to the `GPIO_0` expansion header on the DE2-board. You should be able to verify that the gpio ports goes 'high' (with a multimeter / Analog Discovery) in sequence as you increase the voltage by adjusting the potentimeter.

d) Create the decoder to translate the input sequence to binary and connect it to a binary-to-seven-segment decoder. Note! As the resistor arrary is not connected in running bit order, you should use concatenation to convert the input bits to an ordered sequence:

```
my_input_vector <= gpio0_(5) & gpio0_(7) & Gpio0_(9)
```

etc. See interface in figure 8.

[1] `https://en.wikipedia.org/wiki/Flash_ADC`

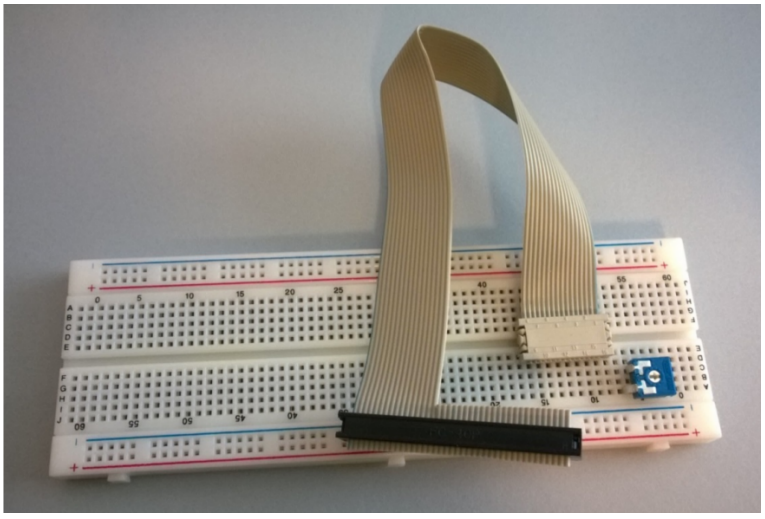

Fig. 6: *Ribbon cable and fumlebrædder.*

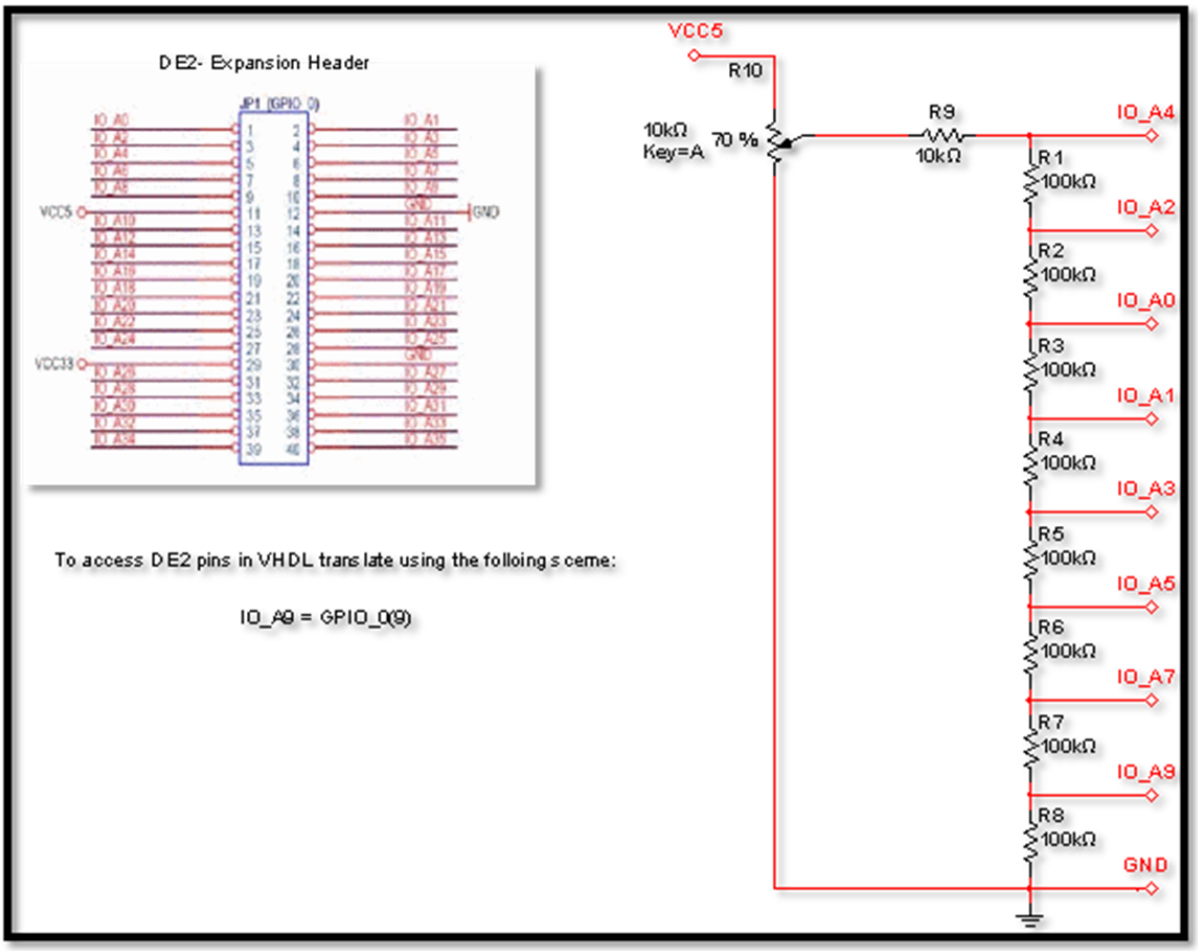

Fig. 7: *Flash ADC schematic.*

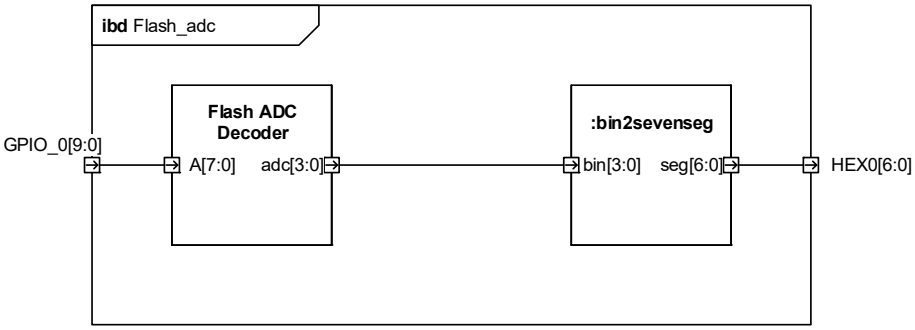

Fig. 8: *Flash ADC IBD.*