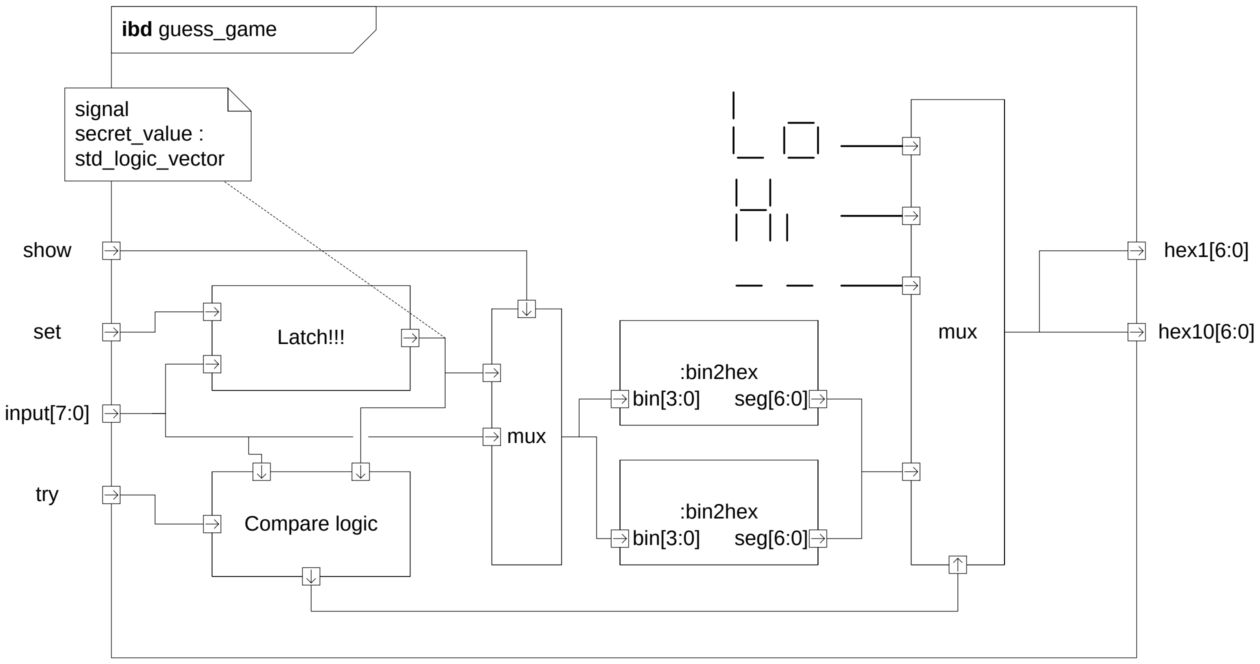
DSD Exercise



Sequential-Style Combinatorial Designs in VHDL

26-10-2015/PHM

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

PREREQUISITES

- Have Quartus II up and running
- That you have read THE DSD EXERCISE GUIDELINES!!!

THE EXERCISE ITSELF

This exercise will get around different sequential code constructions and it starts out with *case* statements. You may skip sub exercise (4) if you wish to try out sub exercise (5).

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.

1) Create a binary-to-7-segment converter component with the interface depicted in figure 3. 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.

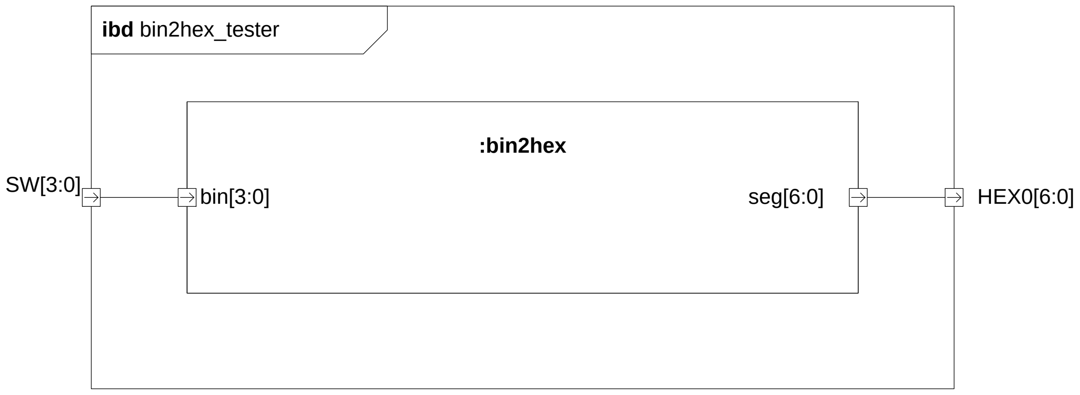


FIGURE 1: BINARY TO 7-SEGMENT DECODER

2) Compile and test the multiplier 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.

1) Design a game with interface shown in Figure 2 and the following functionality:
    a. With no keys pressed, the displays show the current input value
    b. With “Set” button pressed, the input value is stored as the secret number
    c. With “Show” button pressed, the secret number is displayed
    d. With “Try” button pressed, the guess is evaluated and the result is displayed as “Hi”, “Lo” or “--”

```
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 guess_game;
```

FIGURE 2 GUESS GAME INTERFACE

The IBD in fFigure 3 shows the overall building blocks in the design.

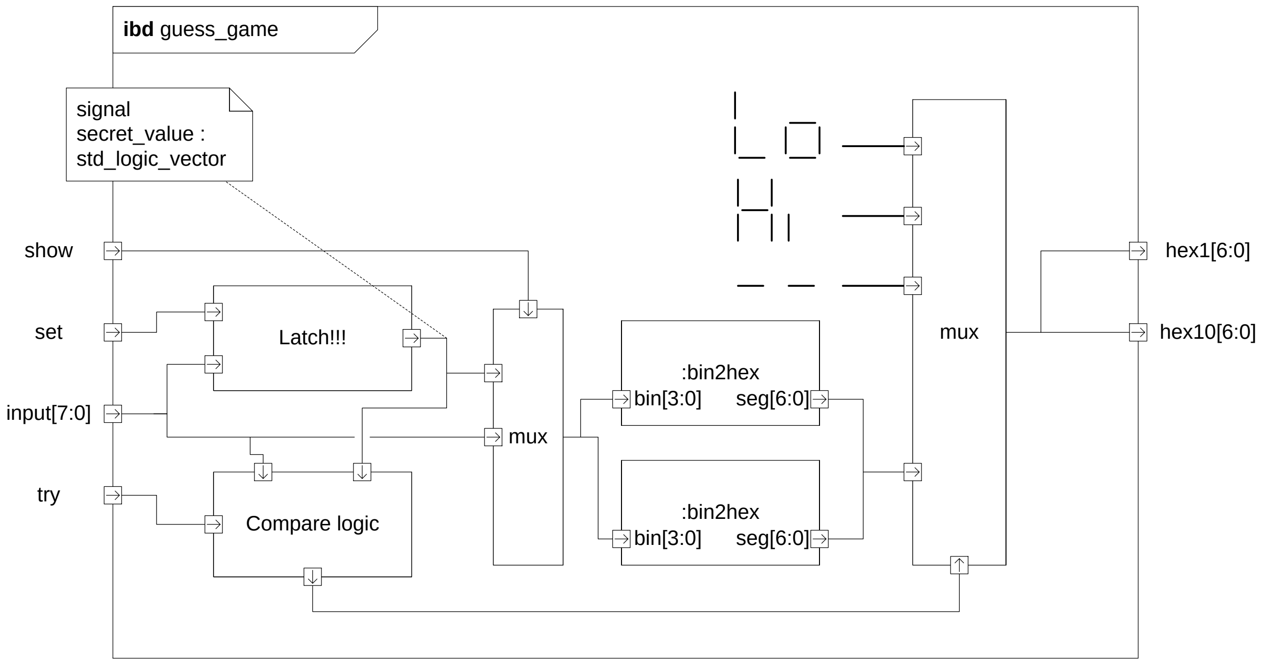


FIGURE 3 GUESS GAME IBD

2) 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?*

2) 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.

1) Design the two-player game as depicted in Figure 4. Use if/case/when/with-select as appropriate. *Argument why you have chosen one solution over others.*

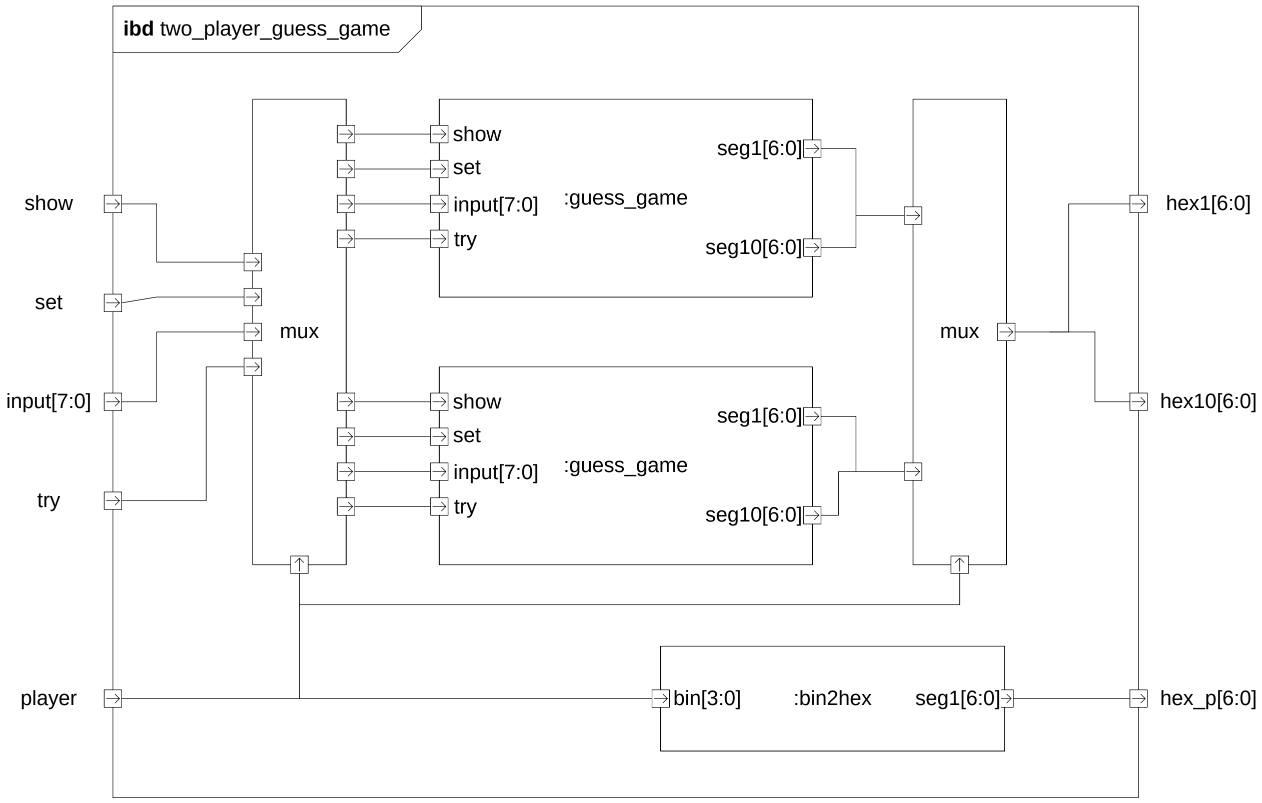


FIGURE 4 TWO-PLAYER GUESS GAME

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

3) 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.

1) Design the 8-input NAND gate depicted in Figure 5 using a loop statement.

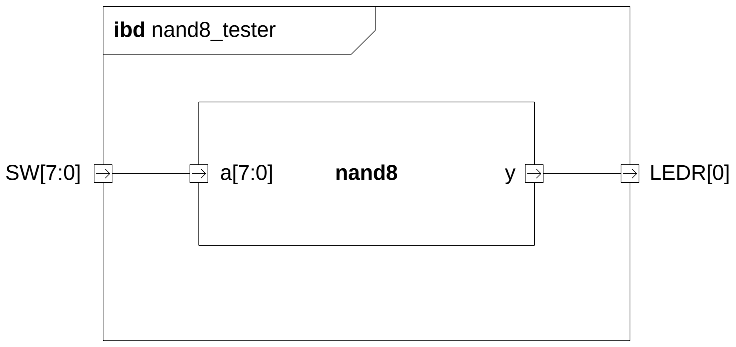


FIGURE 5 8-INPUT NAND

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

(**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 Figure 6. 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.

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

entity generic_and is
  generic (
    bits : integer := 8;  -- default value
  );
  port (
   a, b : in std_logic_vector(bits-1 downto 0);
   y    : out std_logic_vector(bits-1 downto 0);
  );
end entity;

architecture arch of generic_add is
begin
  y <= a and b;
end architecture;
```

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

entity generic_and_tester is
  generic (
    bits : integer := 10;  -- Number of bits
  );
  port (
   SW   : in std_logic_vector(2*bits-1 downto 0);
   LEDR : out std_logic_vector(bits-1 downto 0);
  );
end entity;

architecture arch of generic_add_tester is
begin
    and1 : generic_and
      generic map (bits => bits)
      port map (a,b => a,b, y => y );
end architecture;
```

FIGURE 6 GENERICS IN EFFECT

3) Optional: 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.

4) Count Ones (Optional if you'd rather try (5))

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.

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

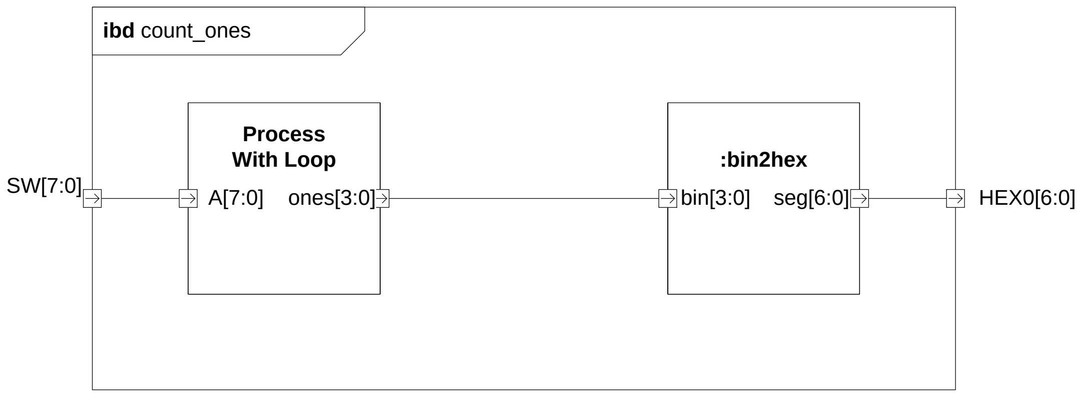


Figure 7 Count Ones IBD

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

5) Flash A/D Converter (Optional)

In this exercise you will build a 3-bit Flash A/D converter (See https://en.wikipedia.org/wiki/Flash_ADC) using a network of resisters and multiple gpio inputs. You must use a loop-statement to convert the input state to a binary value.

1) 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 8. You may just as well find a fumlebrædde from your drawers…

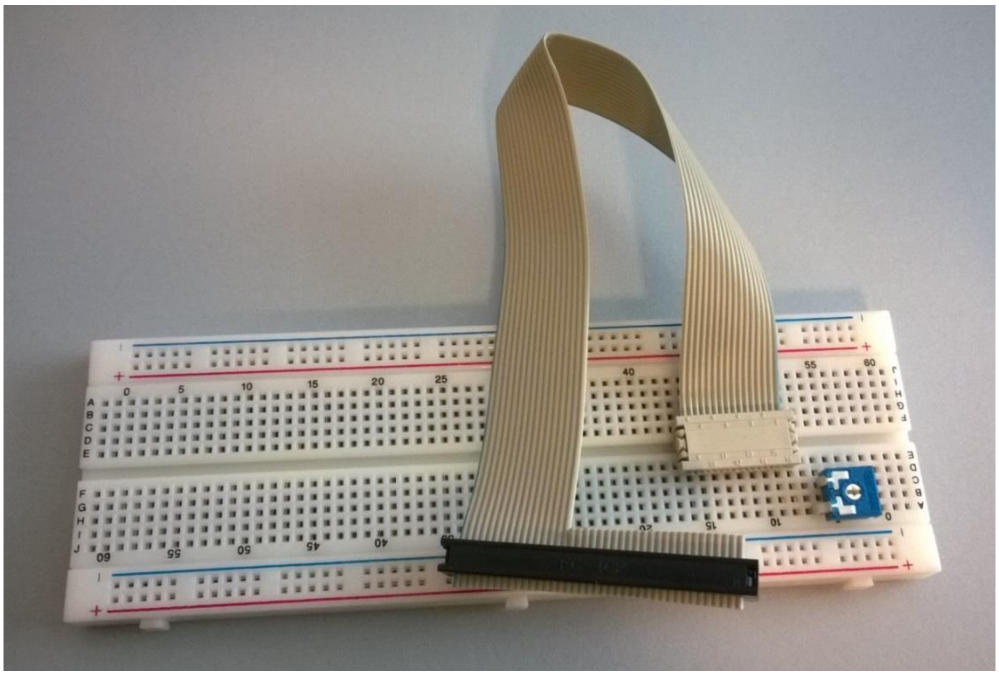


FIGURE 8 RIBBON CABLE AND FUMLEBRÆDDE

2) Build the circuit described in schematic in figure Figure 9 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 result in 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.

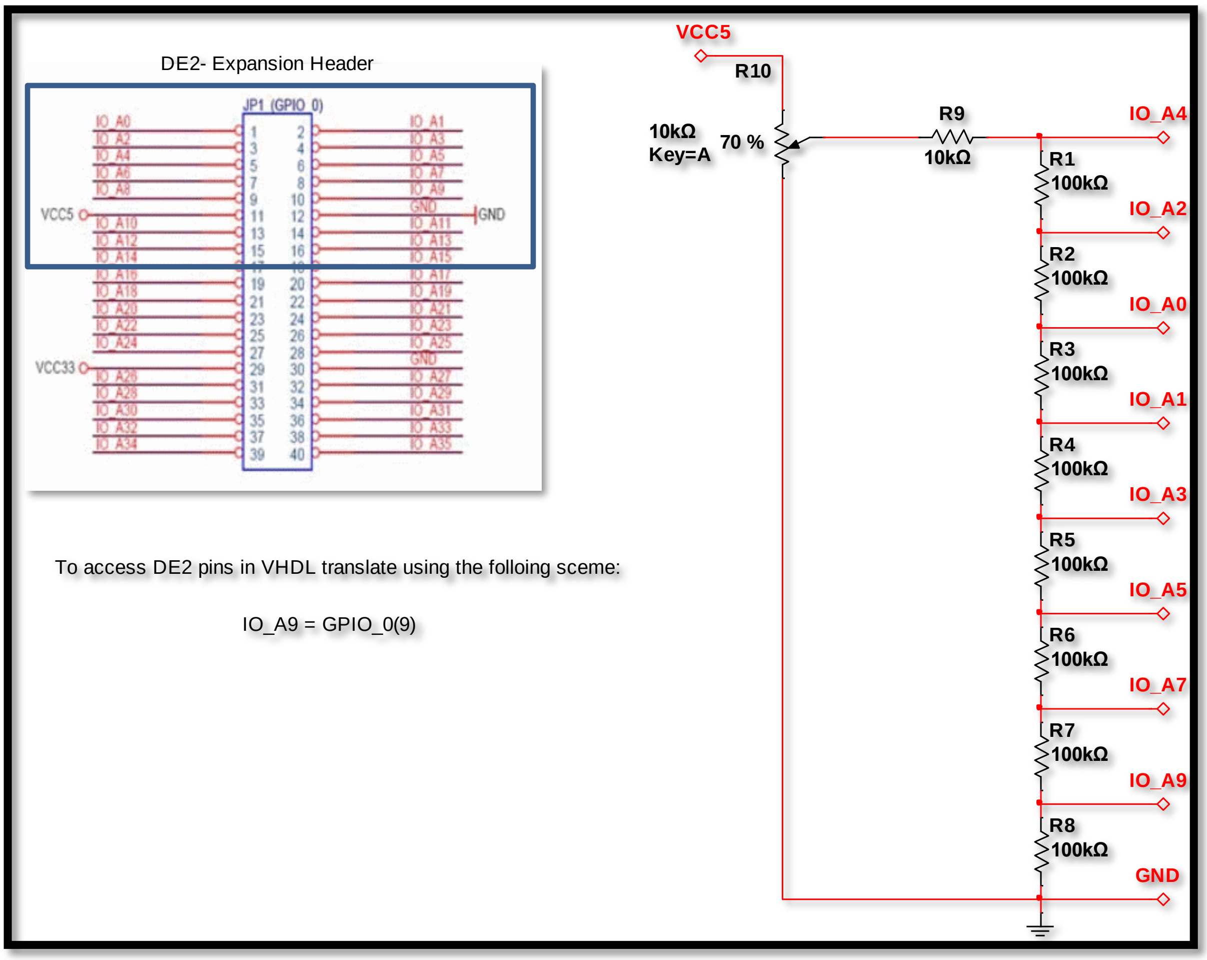


FIGURE 9 FLASH ADC SCHEMATIC

3) 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 potentiometer
4) Create the decoder to translate the input sequence to binary and connect it to a binary-to-seven-segment decoder. Note! As the resistor array is not connected in running bit order, you should use concatenation to convert the input bits to an ordered sequence: ex: my_input_vector <= gpio0_(5) & gpio0_(7) & Gpio0_(9) etc. See interface in figure Figure 10

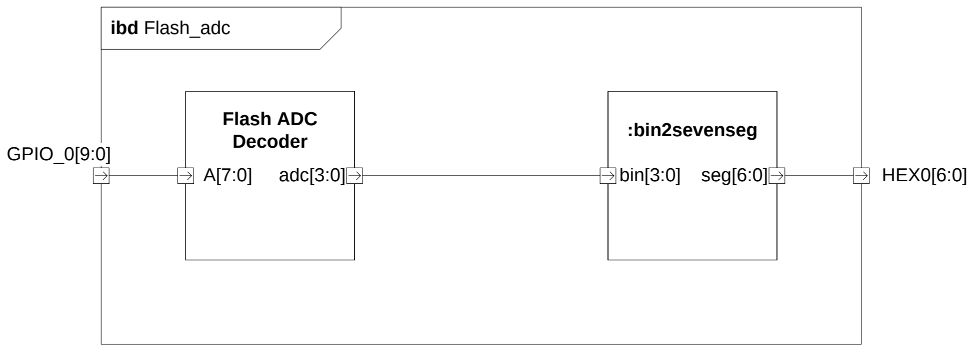


FIGURE 10 FLASH ADC IBD

5) Download and test. You should be able to change the output by turning the analogue potentiometer now ☺!