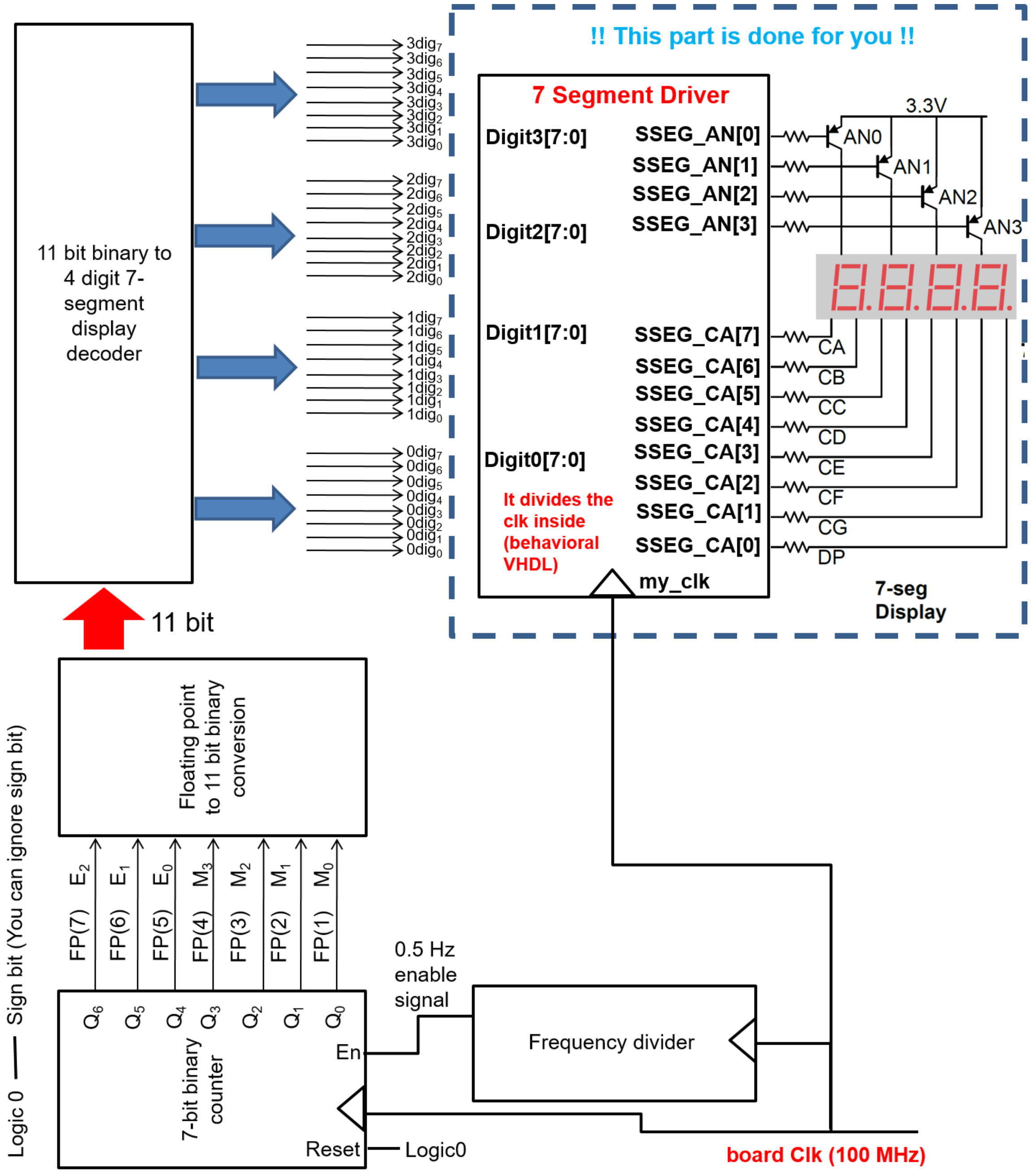
LAB5: CONVERSION OF 8-BIT FLOATING POINT NUMBERS INTO BINARY AND ITS FPGA IMPLEMENTATION TO SHOW ALL POSSIBLE VALUES

The purpose of this lab is to design a decoder that converts a companded 8-bit floating point number representation into binary values and then show all possible results in BCD format using 7-segment LED displays. This decoder is implemented on the FPGA board using VHDL design entry method. Overall schematic for the system is shown in the figure below.



Floating Point Numbers

A linear encoding using 8 bits can represent signed numbers ranging from -128 to +127 using two's-complement representation. This dynamic range supplied by 8-bit linear encoding may not be sufficient for certain applications or the same range may be desired to be covered using less bits. Therefore, nonlinear encodings are used in most practical systems. These encodings represent signals by numbers that approximate the logarithms of their values. Two standard systems, μ-law PCM and A-law PCM, are typically used.

For this laboratory assignment, we will use a simplified floating-point representation consisting of one sign bit, a 3-bit *exponent*, and a 4-bit *mantissa.* The value represented by a 8-bit floating number, S $M_3M_2M_1M_0E_2E_1E_0$, in this format is:

$$Value = (1-2S)\times M\times 2^E$$

where S is the sign bit, M is the mantissa and E is the exponent. The 4-bit mantissa, M, ranges from $(0000)_2$ to $(1111)_2$ representing 0 to 15, respectively and the exponent ranges from $(000)_2$ to $(111)_2$ representing 0 to 7, respectively. Here are some examples for the number representations in floating point number and their magnitudes' binary values in 11 bit.

Floating Point Representation	**Corresponding Value**	**11 bit binary value of the magnitude**
0-1111-111	+1920	11110000000
1-1111-111	-1920	11110000000
0-1000-001	+16	00000010000
0-1100-101	+384	00110000000

Some numbers have multiple representations in floating points. The preferred representation is the one in which the most significant bit of the mantissa is 1; this representation is said to be *normalized*. It is quite straightforward to produce the linear encoding corresponding to a floating-point representation; this operation is called *expansion*. This laboratory assignment is about building this expansion decoder. The combinational circuit that does the inverse operation is called *compression* and it is a little harder to implement. A device that performs both expansion and compression is called a *compander*. The compression part of a compander is more challenging because there are more input bits than output bits and therefore many different linear encodings must be mapped to same floating-point representation. Values that do not have floating point representations should be mapped to the closest floating point encoding. This process is called *rounding*. In this laboratory work, you do not need to worry about compression operation.

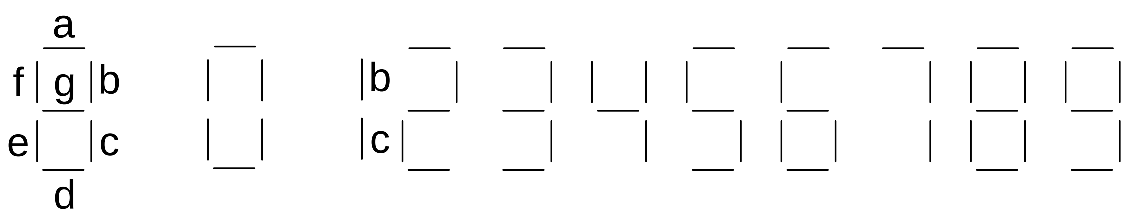
An overall block diagram for the floating-point conversion circuit is shown in the figure above. An 8-bit floating number representation is first converted into 11 bit binary value (maybe by doing multiplications). In this lab assume that the inputs are always positive number. Hence, the sign bit is 0. Therefore, you need to focus on the remaining 7 bits of the floating point number. Then, the resulting binary representation is decoded again (for example, binary to BCD and then BCD to 7-segment decoding) and displayed using 7-segment LEDs. BCD is an acronym for binary coded decimal.

FPGA IMPLEMENTATION OF THE DESIGN

The purpose of this part of the laboratory assignment is to synthesize your decoder that you designed and to actually implement it on an FPGA. You are going to use 7-bit counter with 0.5 Hz frequency to generate all possible values for the bits of the floating point number and see its value on four of the 7-segment displays.

Displaying the final decimal value on the four-digit-seven-segment LED displays sequentially

There is a four-digit-seven-segment display on the FPGA board to display four digits (Digit0, Digit1, Digit2 and Digit3). We want to display the value of the floating point number in the BCD format. In order to do that, binary-to-BCD and BCD-to-7-segment decoding may be used. Or you can convert 11-bit binary value to 4 digit 7-segment display directly. To show each digit, 7 outputs are required as shown below. The VHDL code below shows how a single digit BCD code can be decoded to show its value on a 7-segment display for a common anode configuration (the outputs must be complemented if a common cathode seven-segment display is used.). You can modify this code to accommodate four digits.



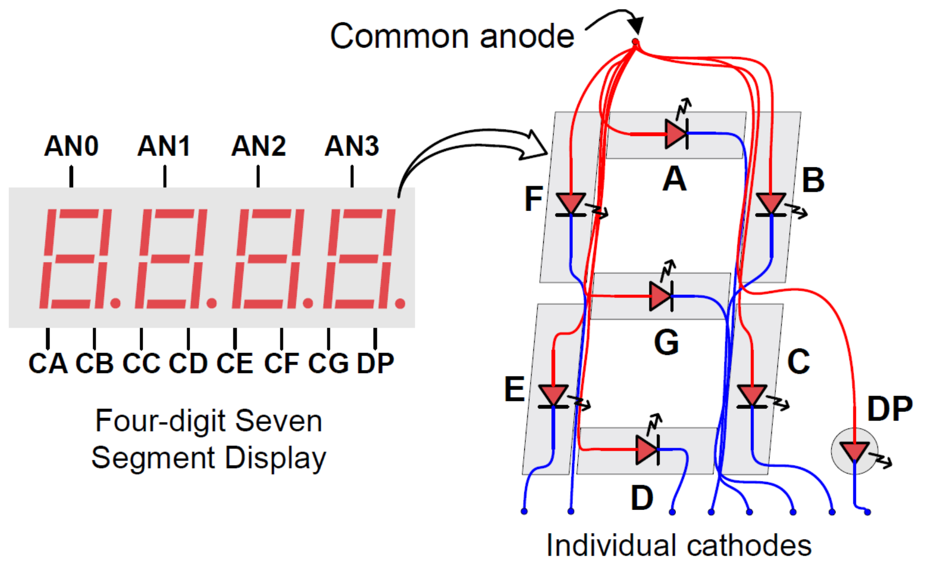
Segments of the seven-segment LED display.

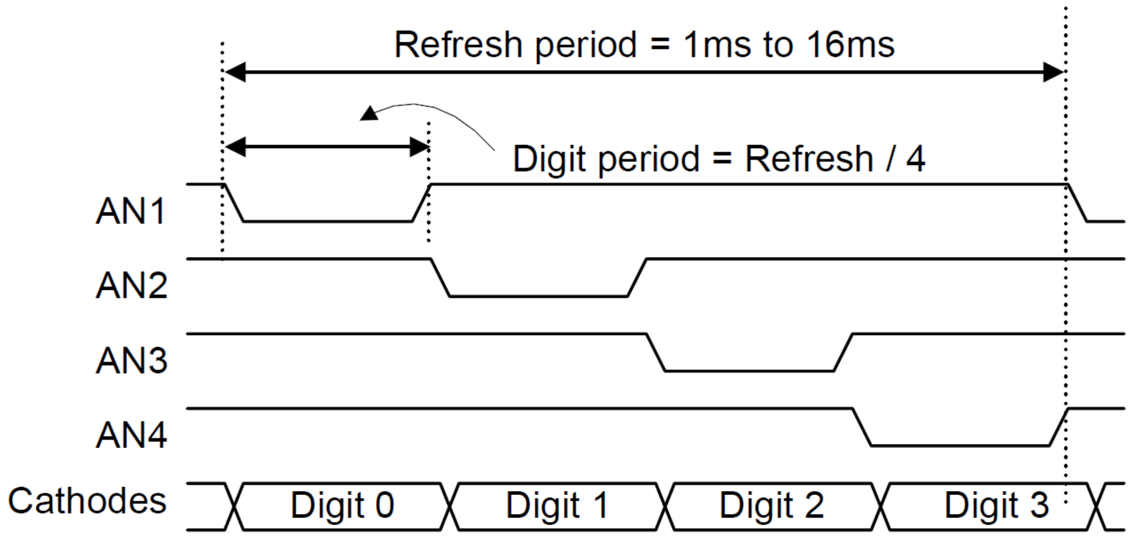
```
entity BCD_to_seven_segment is
port (          d: in std_logic_vector (3 downto 0);
                s: out std_logic_vector ( 6 downto 0) );
end BCD_to_seven_segment;

architecture dataflow of BCD_to_seven_segment is

begin

with d select
    s <="1000000" when "0000",
            "1111001" when "0001",
            "0100100" when "0010",
            "0110000" when "0011",
            "0011001" when "0100",
            "0010010" when "0101",
            "0000010" when "0110",
            "1111000" when "0111",
            "0000000" when "1000",
            "0010000" when "1001",
            "1111111" when others;
end dataflow;
```

Displaying Digits sequentially on the four-digit-seven-segment LED Displays
A seven-segment LED display is composed of individual LEDs with common anode (or cathode) arranged in "figure 8" pattern. However, seven-segment name used for the display of our FPGA board should not mislead you since it actually has eight LEDs (one LED for the decimal point called DP) as shown in the figure below [1]. Each LED on the seven-segment display can be turned on individually. In this lab, you are generating 7-bit data but the board requires 8-bit input for each digit. Since we are not going to use this decimal point, you can assign logic 1 value to the DP LED to turn it off, which is the most significant bit. Therefore, Digit3[7], Digit2[7], Digit1[7] and Digit0[7] can be constant values of logic 1.

In a typical application, each digit of seven-segment display can be used individually at any time. This means, in order to show "1" in Digit0, CB and CC pins of Digit0 must be logic0 (0 V), CA, CD,CE,CF,CG and DP pins of Digit0 must be logic1 (3.3 V) and the common anode pin of Digit0 (AN0) must be logic1 (3.3 V). To use the other three digits, we need individual CA to DP pins. This means for four digits we need to insert 32 pins (for CA to DP pins) plus 4 (for common anodes) pins, which sums to 36 pins. Since this display must be connected to the FPGA chip, this method requires the dedication of 36 pins of the FPGA chip to the seven-segment display, which is wasteful and not desirable. Instead, the same function can be implemented by just using 12 pins with time multiplexing. In this method, CA to DP inputs are made common to all four digits, which requires 8 pins of the FPGA chip. Separate 4 anode pins (AN0, AN1, AN2 and AN3) for 4 digits are required to turn on desired digit. Inserting CA to DP inputs does not turn on LEDs as long as the corresponding common anode pin is pulled up to 3.3 V. Therefore, in order to display a digit in Digit0, CA to DP values are inserted; 3.3 V is applied to AN0 but 0 V to AN1, AN2 and AN3. This way, only one digit can be displayed at a time. Therefore, Digit0, Digit1, Digit2 and Digit3 must be displayed one at a time, one after the other. If this is done very fast (faster than 60 Hz), human eye will see the individual digits turned on at the same time. It will fail to detect this sequential operation of LEDs. The time multiplexing method is explained in the figure below. Cathode values (CA to DP) must be inserted according to the corresponding digit, which is activated by its common anode value (AN0 to AN3). Note that, according to this figure, which shows how the common anode pins are connected to the FPGA chip, when the pin of the FPGA chip, which is N16 for AN0 is logic 0 (0 V), the anode of the digit0 goes to 3.3 V, thus turning it on.

We want to display four digits in this lab. You do not need to design and implement this sequential seven-segment driver operation. This is already done for you. All you need to do is to present your 4 digit data to the seven segment driver component, which is given in the *nexys3_sseg_driver.vhd* file. You can use structural vhdl to connect your design and this seven segment driver design. Thus, your design should have 32 outputs. The first 8-bit output (least significant digit) is connected to Digit0. The second 8-bit output (most significant digit) is connected to Digit1, so on and so forth. We are not going to display the sign bit of the number. Hence, do not worry about the sign bit. The entity of your design should have four sets of 8-bit output ports. These outputs will be connected to seven-segment driver design. Seven segment driver uses a clock signal to turn on individual digits one at a time with a refresh rate of 100 Hz. Thus, the overall design also needs a clock signal for display purposes even though this lab is about combinational circuits. The given VHDL code uses the board clock to generate the 100 Hz refresh rate.

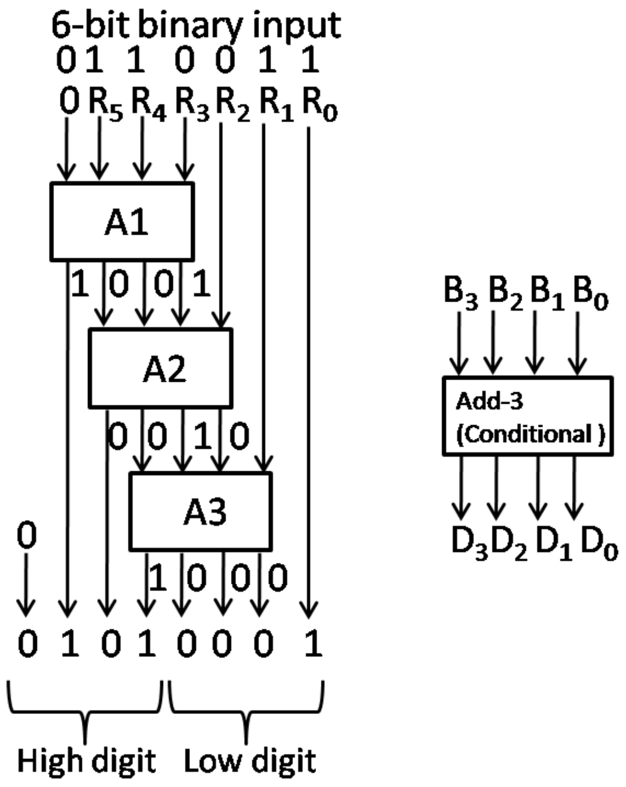




Displaying four digits on four-digit-seven-segment display sequentially [1].

Converting 11-bit binary number into four-digit BCD

You can use “brute force” to implement this converter. This means you can write down all of the cases. However, a better way is to use shift and add-3 algorithm. In this algorithm n-bit binary number is shifted left one at a time to a shift register that has enough 4-bit segments. The process ends when n shifts take place. The BCD number becomes ready in the shift register. During shifting, if the binary value in any of the BCD segment is 5 or greater, 3 is added to that value in that BCD segment. As an example, a 6-bit binary number according to this algorithm can be implemented in structural style as shown in the figure below. A combinational circuit labeled “Add-3(Conditional)” takes a 4-bit binary number and passes it to the 4-bit output as it is if its magnitude is smaller than 5. It adds 3 to this number and it passes this incremented value to the output, otherwise. We do not care what the outputs are for binary input values greater than 9 since we do not insert these values to the circuit. By using three of these circuits as shown on the left of this figure, a 6-bit binary number can be converted to BCD. This structure can be described either using structural or dataflow VHDL.



B_3	B_2	B_1	B_0	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	d	d	d	d
1	0	1	1	d	d	d	d
1	1	0	0	d	d	d	d
1	1	0	1	d	d	d	d
1	1	1	0	d	d	d	d
1	1	1	1	d	d	d	d

Conversion of a 6-bit binary number to BCD

Overall Design

As explained above your design will generate 8-bit input (in reality, you will generate 7 bit, since we ignore sign bit). You will use a slow counter with a frequency of 0.5 Hz to generate this signal named FP. You will, then, calculate its value as a 11-bit binary value. Finally, you will display this value as 4 decimal digit on 7-segment displays. Your design will generate 8-bit digit3 output, 8-bit digit2 output, 8-bit digit1 output and 8-bit digit0 output. Then, these outputs will be inputs to the design given in the *nexys3_sseg_driver.vhd* file using structural VHDL style. This design will generate the SSEG_CA(7 downto 0) and SSEG_AN(3 downto 0) outputs using the clock signal it generates from the board clock. As a starting point your VHDL code for this design can have a structure as shown below. You should complete its architecture. Your VHDL code can use only structural and/or dataflow style VHDL. Behavioral style is not allowed in this lab. If you want, you can build the required frequency dividers by first building their schematics and then turning their schematics into structural VHDL code.

```
-- Synthesizable Floating Point to Signed Magnitude Decoder, EE240 class Bogazici University
-- Implemented on Xilinx Spartan VI FPGA chip

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;
USE ieee.std_logic_arith.all ;

entity FP_decoder is
        port (
            MY_CLK: in std_logic;
            SSEG_CA: out std_logic_vector(7 downto 0);
            SSEG_AN: out std_logic_vector(3 downto 0));
end;

architecture arch_FP_decoder of FP_decoder is

--MY_CLK is coming from the board clock, connected to V10

--digit3(0) is displayed on Digit0 Segment CA
--digit3(1) is displayed on Digit0 Segment CB
--digit3(2) is displayed on Digit0 Segment CC
--digit3(3) is displayed on Digit0 Segment CD
--digit3(4) is displayed on Digit0 Segment CE
--digit3(5) is displayed on Digit0 Segment CF
--digit3(6) is displayed on Digit0 Segment CG
--digit3(7) is displayed on Digit0 Segment DP, which is always logic 1

--digit2(0) is displayed on Digit0 Segment CA
--digit2(1) is displayed on Digit0 Segment CB
--digit2(2) is displayed on Digit0 Segment CC
--digit2(3) is displayed on Digit0 Segment CD
--digit2(4) is displayed on Digit0 Segment CE
--digit2(5) is displayed on Digit0 Segment CF
--digit2(6) is displayed on Digit0 Segment CG
--digit2(7) is displayed on Digit0 Segment DP, which is always logic 1

--digit1(0) is displayed on Digit0 Segment CA
--digit1(1) is displayed on Digit0 Segment CB
--digit1(2) is displayed on Digit0 Segment CC
--digit1(3) is displayed on Digit0 Segment CD
--digit1(4) is displayed on Digit0 Segment CE
--digit1(5) is displayed on Digit0 Segment CF
--digit1(6) is displayed on Digit0 Segment CG
--digit1(7) is displayed on Digit0 Segment DP, which is always logic 1

--digit0(0) is displayed on Digit0 Segment CA
--digit0(1) is displayed on Digit0 Segment CB
--digit0(2) is displayed on Digit0 Segment CC
--digit0(3) is displayed on Digit0 Segment CD
--digit0(4) is displayed on Digit0 Segment CE
--digit0(5) is displayed on Digit0 Segment CF
--digit0(6) is displayed on Digit0 Segment CG
--digit0(7) is displayed on Digit0 Segment DP, which is always logic 1

--digit3(7 downto 0), digit2(7 downto 0), digit1(7 downto 0) and digit0(7 downto 0) will be
-- input to the design given in the nexys3_sseg_driver.vhd file using structural vhdl style.
-- This design will generate the SSEG_CA(7 downto 0) and SSEG_AN(3 downto 0)
-- outputs using the board clock signal.

end arch_FP_decoder;
```

You may need to use a multiplier in this design. A VHDL multiplier example is given below.

```
LIBRARY ieee ;
```

```
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;
USE ieee.std_logic_arith.all ;
entity mymultiply is
            port(
                        a : in STD_LOGIC_VECTOR(15 downto 0);
                        b : in STD_LOGIC_VECTOR(7 downto 0);
                        mult : out STD_LOGIC_VECTOR(23 downto 0);
                  );
end mymultiply;
architecture dataflow of mymultiply is
begin
          mult <= a * b;
end dataflow;
```

Pin Assignment

After successfully completing the design, compilation, ISim simulations and synthesis, you should show that your design work on the FPGA board. FPGA board has certain wiring. Only specific pins of the FPGA are connected to seven-segment displays. The input and output ports of your design should be connected to these pins of FPGA. You can specify which pin of the FPGA is connected to your input and output of your design before starting synthesis. You can achieve this pin assignment either manually by entering the corresponding pin numbers below using the graphical interface of the Xilinx ISE CAD tool or using the .ucf file below. By looking at the manual of the Nexys3 Board, we can find which FPGA pins are connected to which components. The following is a list of pin connection we want and their meanings.

```
SSEG_CA<0>  -> T17
SSEG_CA<1>  -> T18
SSEG_CA<2>  -> U17
SSEG_CA<3>  -> U18
SSEG_CA<4>  -> M14
SSEG_CA<5>  -> N14
SSEG_CA<6>  -> L14
SSEG_CA<7>  -> M13

SSEG_AN<0>  -> N16
SSEG_AN<1>  -> N15
SSEG_AN<2>  -> P18
SSEG_AN<3>  -> P17

MY_CLK  ->V10
```

The .ucf file should be like this.

```
#
# Pin assignments for the Nexys3 Spartan VI Board.
#
Net SSEG_CA<0> LOC=T17  |  IOSTANDARD=LVCMOS33; # Connected to CA
Net SSEG_CA<1> LOC=T18  |  IOSTANDARD=LVCMOS33; # Connected to CB
Net SSEG_CA<2> LOC=U17  |  IOSTANDARD=LVCMOS33; # Connected to CC
Net SSEG_CA<3> LOC=U18  |  IOSTANDARD=LVCMOS33; # Connected to CD
Net SSEG_CA<4> LOC=M14  |  IOSTANDARD=LVCMOS33; # Connected to CE
Net SSEG_CA<5> LOC=N14  |  IOSTANDARD=LVCMOS33; # Connected to CF
Net SSEG_CA<6> LOC=L14  |  IOSTANDARD=LVCMOS33; # Connected to CG
Net SSEG_CA<7> LOC=M13  |  IOSTANDARD=LVCMOS33; # Connected to DP

Net SSEG_AN<0> LOC=N16  |  IOSTANDARD=LVCMOS33; # Connected to AN0
Net SSEG_AN<1> LOC=N15  |  IOSTANDARD=LVCMOS33; # Connected to AN1
Net SSEG_AN<2> LOC=P18  |  IOSTANDARD=LVCMOS33; # Connected to AN2
Net SSEG_AN<3> LOC=P17  |  IOSTANDARD=LVCMOS33; # Connected to AN3

Net MY_CLK LOC=V10  |  IOSTANDARD=LVCMOS33; # Connected to board clock
```

Preparation (Prelab)

For Floating point to 11 bit binary magnitude conversion

- Write a VHDL code for this component. Use only structural or dataflow style.
- Run ISim and verify that it functions as expected by running a few test sequences.

For the 11-bit binary to BCD Conversion (or 11-bit to directly 4 digit 7-segment conversion)

- Write a VHDL code for this component. Use only structural or dataflow style.
- Run ISim and verify that it functions as expected by running a few test sequences.

For the four digit BCD to four digit 7-Segment Conversion

- Use the binary-to-seven-segment code given above.

For the Overall Design

- Combine your design components and the design given to you in *nexys3_sseg_driver.vhd* under FP_decoder architecture and run ISim on it and verify that it functions as expected by running a few test sequences.

In Lab

- Use Xilinx ISE to synthesize the design based on Xilinx Spartan VI family. Use either the given ucf file or enter manually to make pin assignment.
- Generate the bit file and upload it to your FPGA.
- On the hardware show that your design works.
- See if the LEDs show what you expect every 2 seconds as your input counts up automatically.

Summary

- Open a new Xilinx ISE project.
- Complete VHDL definition of your design.
- Simulate your design using Isim and verify correct functionality.
- Use Xilinx ISE to synthesize your design on the FPGA board.
- Demonstrate your TA that your decoder calculates the values of the floating point number that is updated every 2 seconds and displays correctly on the four of the7 segment LED displays on the board.

References

[1] "Nexys 3 Manual." (accessed 18 March 2021).