ECE 340

Embedded Systems

Spring 2022

LAB1

Hardware Design

FPGA Design of a Floating Point Unit (FPU)

Introduction

Lab1 is about design, functional simulation and implementation (synthesis, placement and routing) of a Floating Point Adder in Verilog using the Xilinx Vivado® toolset. Vivado is the Xilinx tool used for hardware design, including RTL simulation, synthesis, placement, routing and bitstream generation. As part of Lab1, you will be able to download and test your design on Zedboard under different configurations.
The Zedboard digital system development platform features Xilinx's Zynq-7000 FPGA, 512 MB external DDR3 memory and enough I/O devices and ports to host a wide variety of digital systems.

Xilinx tools set up

Make sure that your Linux environment is set up for the Xilinx Vivado toolset. Run (or, even better, place in your start-up shell script) the following commands[1]:

```
source /tools/Xilinx/Vivado/2020.2/settings64.sh
source /tools/Xilinx/Vitis/2020.2/settings64.sh

# License Files for Xilinx tools
export XILINXD_LICENCE_FILE=2100@10.64.82.9
```

The previous script commands refer to *bash* shell, and the start-up shell script is *.bashrc*, which should be placed in your home directory. Every time you launch a new terminal, the *.bashrc* script is automatically executed.

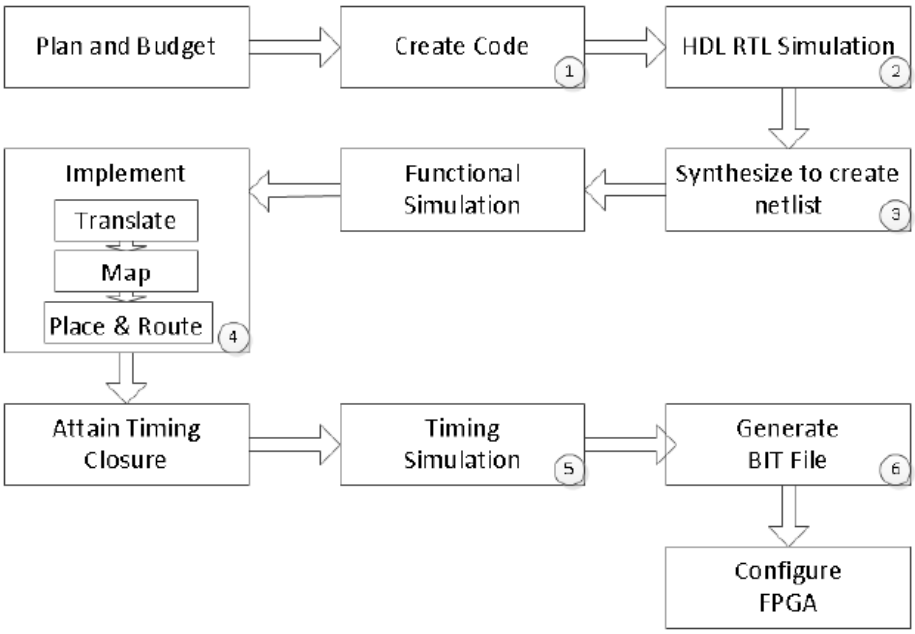
Objectives

The objective of these lab sessions are the following:

a) to introduce the concepts of hardware design of a realistically complex component (Floating Point Adder) using Verilog HDL,
b) to introduce functional and timing simulation and debugging using testbenches and the Xilinx Vivado simulator,
c) to introduce the concepts of user-constrained design synthesis, placement, routing and bitmap generation on a Xilinx FPGA board,
d) to introduce the downloading and demonstration of the design on the board, and
e) to introduce the functionality of the Zedboard.

The diagram shows the typical design flow that will be followed in this lab.

[1] For the Xilinx 2020.2 toolset. Modify accordingly for more recent tool versions.



Floating Point Addition

Calculations that take place over a very wide range of numbers or require very high precision usually resort to floating point representations. The Floating Point Adder that you will develop in this section is a hardware model for the IEEE Standard 754-1985 for single precision floating point numbers.

According to the 754 IEEE standard, a floating-point number contains a sign bit, an exponent, and a mantissa. For the 32-bit standard, the format is illustrated in the figure below.

31 30 ..... 23 22 ... 0

Sign (S)	Exponent (E)	Mantissa (M)

The value of number F is given by the following equation:

$F = (-1)^S * 2^{E-127} * (1 + M)$ (1)

This equation is valid only for

$0 < Exponent < 255$ (2)

The values 0 and 255 are used as special-purpose flags to denote, among others, $\pm\infty$, denormalized numbers, and NaNs (Not a Number). For this lab, you should assume that the input values to the FP Adder are given so that they satisfy condition (2). Moreover, you may assume that the addition does not result in underflow or overflow so that there is no need to check for these conditions. Figure 1 shows the stages of an FP addition.

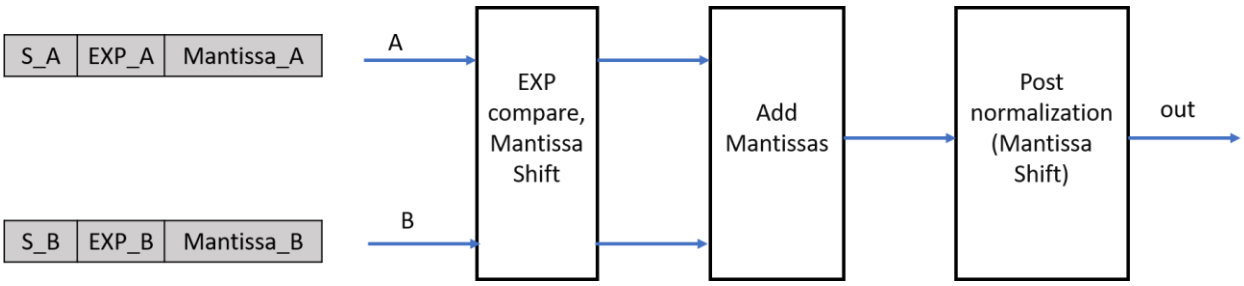


Figure 1. Stages of FP addition.

Step1. Design and behavioral simulation of single cycle FP Adder

Step 1 of the lab is to design and simulate a Floating Point Unit that performs single-precision addition in a single clock cycle. Make sure that the FP inputs and the output are registered so that you can isolate the pure computation in a single clock cycle. Also, make sure that you correctly implement the case when the output is equal to zero.

The purpose of this module is to get a general understanding of designing, simulating, and debugging your hardware described in Verilog. This task is usually accomplished using a test bench which is also written in Verilog. Therefore, it is important to understand that a Verilog test bench is as error-prone as a Verilog hardware description. You should simulate, test, and debug your Verilog code and show its correct functionality on the input tests provided to you.

Start this lab by copying the two input files of the module as well as the input test file (*.hex) into your local folder and study the skeleton Verilog files. Your main task is to study the FP addition algorithm (perhaps doing an example with paper and pencil), and complete the Verilog code in the *fpadd_single.v* file. Note that in Zedboard (to be used in later steps), the asynchronous reset signal is active high (posedge).

One of the challenges of the FP addition is to implement a functionality to count the number of leading zeros of a number B. For example, if B = 23'b000_0010_1110_0110_1110_0001, then N = 5. You will need this functionality at the post-normalization phase.

In the test bench file, the FP Adder is instantiated and its output is compared against the expected output of the test file. Note the use of the $*readmemh* Verilog API to read the input and expected output as text files with hexadecimal numbers. An *initial* statement is used to initialize system signals such as *rst*, *clk*. Another behavioral statement generates the clock signal with a period *cycle.* This requires that the *clk* signal reverses after *cycle/2* ns.

1. Once you are satisfied that your code makes sense, launch Vivado from a Linux terminal to create a new design project (make sure that you have already set the tools up as described in the Overview document):

   vivado &

   The Vivado Project Navigator GUI will open.

2. You create a new project: *Create New Project*
3. At the *Create New Vivado Project* page click *Next*
4. For Project Location, browse to your preferred directory. Type the Project Name (e.g.

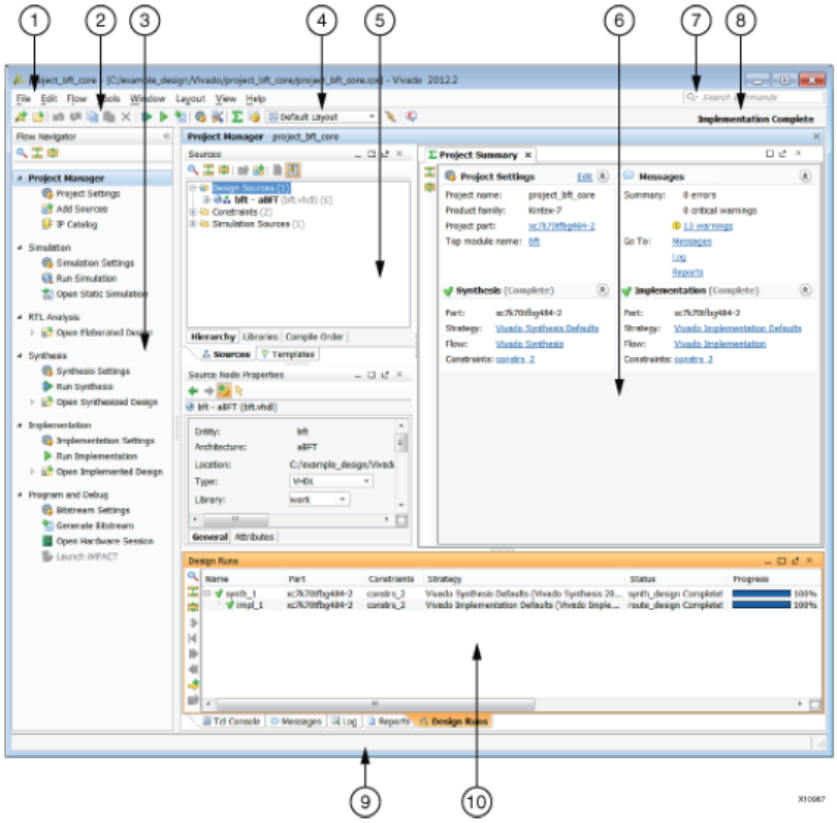


Figure 2. Vivado IDE Viewing Environment

FPAdder_Single) and make sure that the *Create Project subdirectory* is checked. Click *Next.*

5. In the *Project Type* form select *RTL Project*. Click *Next.*
6. In the *Add Sources* form select the Verilog files of your design. Click *Next.*
7. In the *Add Constraints* form click *Next.* You will include design constraints, only when you implement your design in the FPGA board.
8. In the *Default Part* form specify *Boards* and start searching for *Zedboard*. Select *Zedboard Zynq Evaluation and Development Kit.*
9. In the *New Project Summary* form click *Finish* to create the project.

Figure 2 shows the Vivado IDE environment:

1. Menu Bar
2. Main Toolbar
3. Flow Navigator
4. Layout Selector
5. Data Windows Area
6. Workspace
7. Menu Command Search Field
8. Project Status Bar
9. Status Bar
10. Results Window Area

The next step is to use the Vivado built-in simulator and examine the behavioral simulation

results[2]. Behavioral simulation only simulates the Verilog code without any inference to gates or transistors and is used to prove the logical correctness of your Verilog code. In the *Flow Navigator*, under the *Simulation* tab, select the *Simulation Settings* button and make sure that the Target Simulator is Vivado, the Simulator Language is mixed (or Verilog), and that the top module name is the name of your testbench module. Press OK. In the *Flow Navigator,* press *Run Simulation* and *Run Behavioral Simulation.*

The Verilog code will be compiled and executed using the testbench as the top level module. The workspace of the Vivado IDE will change and new windows will appear when you execute the testbench. One window shows the simulation results as waveforms, another shows the Objects of the testbench, and yet another the hierarchy of the design and the *glbl* instances (*Scopes*). The *Main Toolbar* can be used to guide the simulation. The most widely used icons are: *restart* which resets the simulation and the waveform and *run* <*time*> which runs the simulator for time N. *For* example, *run 10000 ns* will run the design for 10000 ns and will terminate. You can move around the hierarchy of your design from the *Scopes* window, select signals and variables that you want to trace and simulate from the *Objects* window, and drag them to the waveform window.

Test your simulation to verify its correctness. The number of errors in the testbench should be zero.

Go to the project directory and check the directory structure that Vivado created to store the files for project support. The <*Project Name*>.*srcs* directory contains all project source files and design constraints (to be explained later in step 4). The <*Project Name*>.*sim* directory contains automatically generated files for each simulation campaign. The <*Project Name*>.*data* directory is a placeholder for the Vivado program database.

Step 2. Design and behavioral simulation of a pipelined FP Adder

Implementing the functionality of the FP Adder in a single cycle may be easier, but it may be stretching the clock period when the module is implemented in the FPGA. This is because a single clock period has to accommodate more operations (in this case the complete FP addition). Therefore, the designer may have to reduce the clock frequency to give the electric signals more time to move between flip-flops. A high-performance implementation requires that the functionality of the FP adder be pipelined with at least two stages.

In step2 you will expand your design to compute the FP addition in two pipeline stages. It is up to you to decide how to allocate the functionality of the adder across the two stages. You need to make sure that there is a balanced allocation so that each stage performs approximately 50% of the total computation. You should create a new project for step2.

Another thing you should try is to analyze your RTL code. In the *Flow Navigator*, select *RTL Analysis*, expand the *Elaborated Design* and then select *Schematic*. Elaboration is a step during simulation and synthesis in which the design hierarchy is flattened producing an interconnection of modules. The graphical schematic can be viewed in a new window.

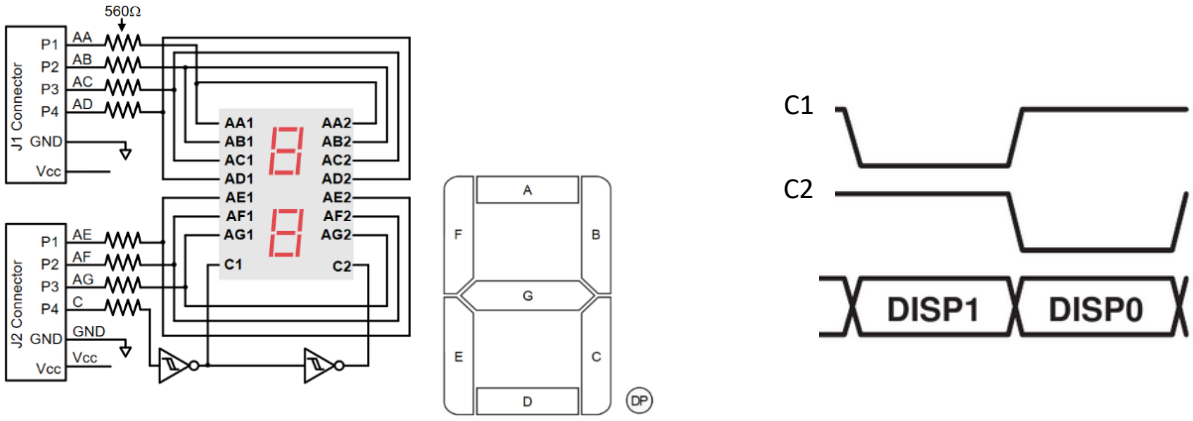
Step 3. FP Adder in FPGA. 7 segment display output

The 2-stage FP adder that you verified in the previous step, will now be implemented on the Zedboard board (step 3). However, we need to be able to provide inputs to the adder, and also to see the result of the addition on the Zedboard. In other words, we need to implement some form of I/O using the peripherals available on the Zedboard. The Zedboard provides 8 LEDs that can be used to output 8 (out of the 32) bits of the result. However, we will also use a more sophisticated form of output: the seven-segment displays (7seg).

[2] If you prefer, you can use other simulators such as Modelsim

7 seg displays

The 7-segment display module consists of seven LEDs (hence its name) arranged in a rectangular fashion. Each of the seven LEDs is called a segment because, when illuminated, the segment forms part of a numerical digit to be displayed. Figure 3 shows the block diagram of the 7seg display (2 digits) to be used in step 3, which is a peripheral connected to the PMOD ports of the Zedboard. We will use such peripherals to depict a total of 4 digits (16 bits).



(a) (b)

Figure 3. (a) The seven-segment display (2 digits) block diagram. (b) Waveforms showing the two Anodes (C1 and C2) used to select a digit and the values {A,B,C,D,E,F} for the two digits. Note that these 7 values are multiplexed and are used to drive a digit when the corresponding anode is active low.

To turn on a specific LED of a digit:

- The anode C1 (or C2) that selects the digit is driven low
- A subset of the values {A,B,C,D,E,F} is also driven low. For example, the value 0 appears at the top digit of Figure 3a, when C1=0 and all values become 1 except for value G which remains 0.

Figure 3b shows that time multiplexing is needed to drive both digits of the 7seg display and this process is repeated continuously. In other words, even if the two LEDs are being driven in sequence (they are never selected both at the same time), the human eye does not capture this discontinuity because the switching happens very fast.

When the anode becomes inactive (=1), the LED values are being discharged. The anode needs to stay active (=0) for some minimum amount of time to be able to charge the LEDs and drive the digits. In step 3, you should generate the periodic anode signal with a period equal to 0.32 us (=320 ns), which is a multiple of the 10 ns clock period of the FPGA (i.e. the period of C1 in Figure 3b is 320 ns).

In step 3, you will need to write Verilog code to implement the functionality of a 2-digit 7seg display as a PMOD peripheral. For example, the module SevenSegmentDisplay will have the following ports:

```
SevenSegDisplay SSD (.clk(clk), .rst(rst), .DataIn(data), .an(an0), .a(a),  .b(b), .c(c), .d(d), .e(e0), .f(f), .g(g));
```

The DataIn port is the 8-bit input to the module. The seven 1-bit signals {a,b,c,d,e,f} are the output of your module and will be connected to the FPGA output pins, and from there to the PMOD ports as specified in the constraints file (see next subsection). For example, if DataIn = 8'h76, the seven signals will be switching between the values {a=1, b=1, c=1, d=0, e=0, f=0, g=0} and {a=1, b=0, c=1, d=1, e=1, f=1, g=1} every 0.32us.

In step 3, you have to design, simulate and, implement a 7seg display module. The *fpadd_system.v* top level Verilog file should instantiate the FP Adder module, and two 7seg display modules. Moreover, 8 output bits should be driven to the onboard LEDs. The two inputs will be hardwired to a specific value chosen by the user.

Hardware implementation

After this step, you should be able to implement a small hardware system on an FPGA board as shown in Figure 4. You will also learn details on the implementation flow of the Vivado toolset. Since this is the first time that you use the Xilinx implementation tools, this lab is described as a walk-through in which all steps are explained. Launch Vivado from a Linux terminal and create a new design project (*e.g. FPAdd_7seg*):

% vivado &

Create the new project which includes the Verilog files you created in step 3 without the test bench. We will not need a test bench file because we will run the design on the board instead of simulating it. The file *fpadd_system.v* is the top level of the design.

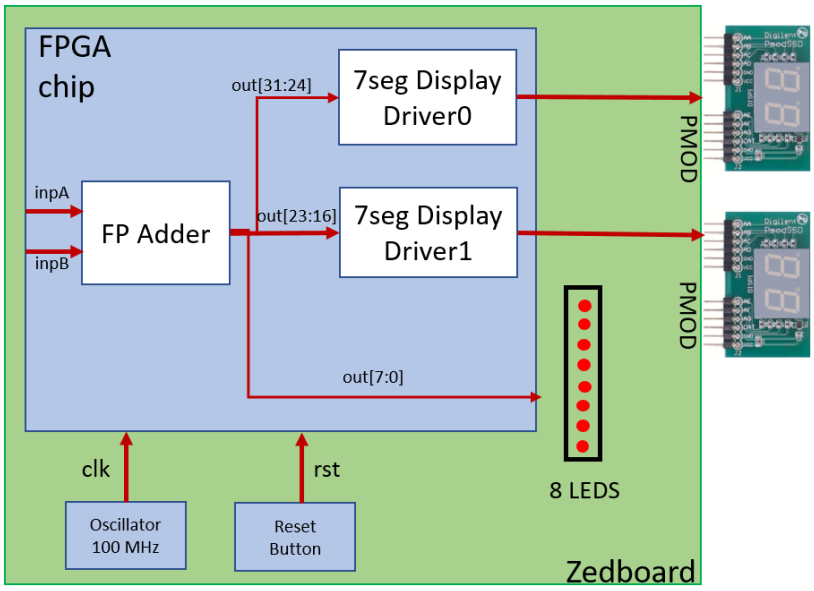


Figure 4. Zedboard platform for step 3.

Design Constraints

Besides the Verilog source code, an implementation to an FPGA bitstream requires that the designer defines design constraints. These constraints specify the FPGA I/O pins which will be assigned to the I/O module ports (e.g. clock, reset, LEDs, PMOD for 7seg displays, etc.), their

physical characteristics, timing constraints such as clock period and clock duty cycle, and so on. In particular, timing is the most important constraint of a design since it has a direct impact on the speed and area of the generated circuit. There is an inverse relationship between area (i.e. number of LUTs, Flip Flops, block RAMs, etc.) and speed (i.e. clock frequency): if the designer requires higher clock frequencies this will typically result in larger circuits. Can you guess why?

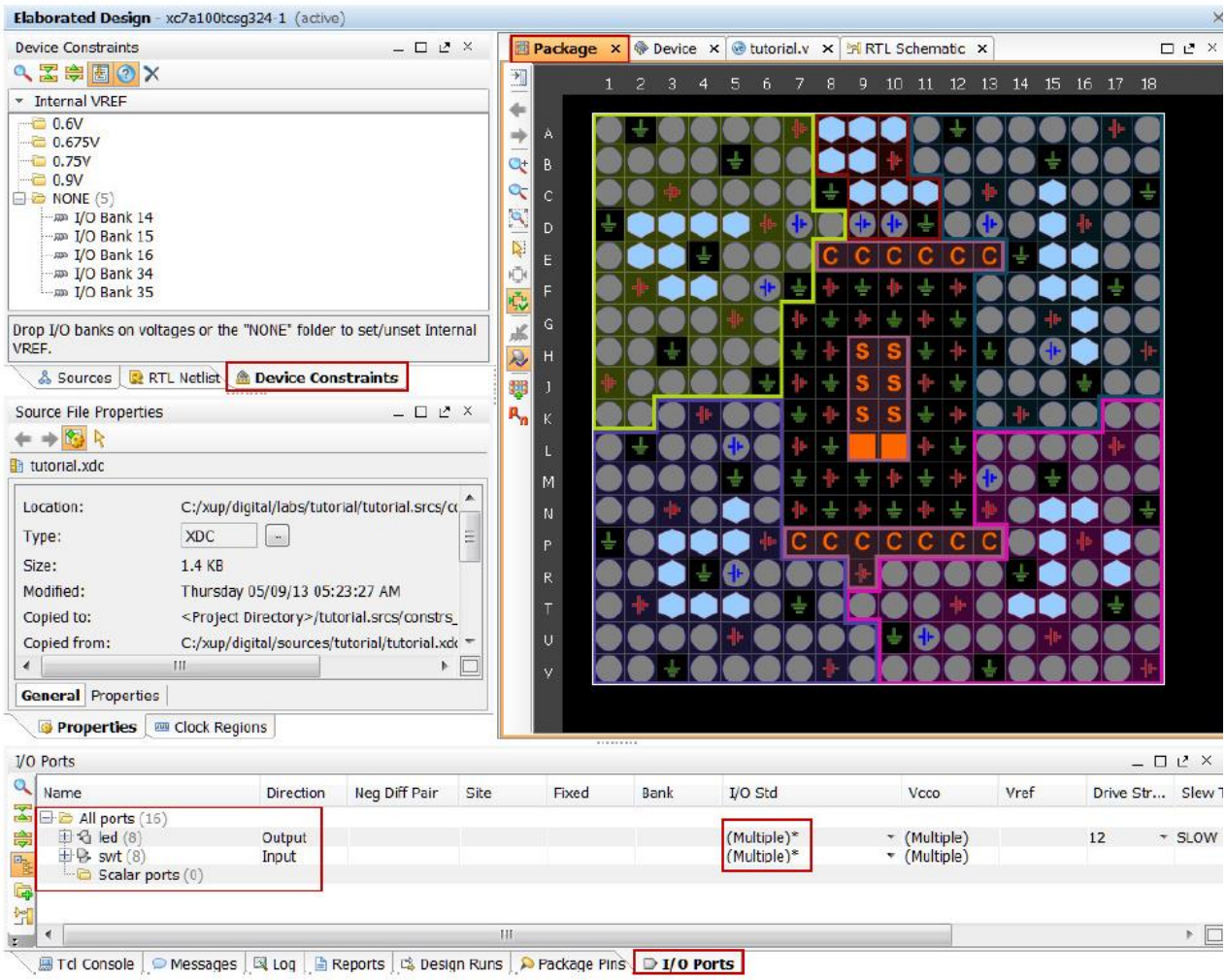
Select *Add Sources* in the *Flow Navigator*, then select *Add or Create Constraints* and click *Next.* Select *Add Files,* and select the *fp_add.xdc* text file. For this lab, the incomplete XDC file (XDC stands for Xilinx Design Constraints) is given to you. The XDC file is where the Vivado tool is instructed by the user to perform time-constrained placement and routing and to place the I/O ports of the design onto specific FPGA pins that have a dedicated function in the board (e.g. they are connected to the LEDs or PMOD ports). This is where you specify the target clock frequency that you want your system to run.

You will need to fill in the details, i.e. the entries that contain *XXX*. You will have to consult the Zedboard schematic for the pin locations of the *clk, rst*, *leds* (named LD0 to LD7), and PMODs in the Zedboard schematic. It is actually even easier in the Zedboard because the pin locations are the small codes near the leds. For example, led[0] is electrically connected to pin T22 of the FPGA. Study the syntax of the Tcl script (pronounced as tikl) language that is used in the XDC file and try to extend it to all I/O pins.

The XDC file also specifies a clock period of 10 ns (100 MHz). This is the clock frequency that you request from the synthesis tool to target when producing the final netlist. Apparently, the actual clock that you will provide to the circuit cannot be higher than 100 MHz (Figure 4).

An alternative way to specify design constraints is by the drop-down *Layout Selector* menu. In the Flow Navigator, open the Elaborated Design, select *I/O Planning* and notice that the *Package* GUI view is displayed in the *Workspace* area and the I/O ports tab in the *Results Window Area* (see figure below). Move the cursor over the *Package* view, highlighting different pins. Notice the pin site number is shown at the bottom of the Vivado GUI, along with the pin type (User IO, GND, VCCO…) and the type of the I/O bank it belongs to.

Expand the led ports by clicking on the + box and make sure that all I/O ports use the LVCMOS33 I/O standard (change it if they do not). To assign a pin location for led[0], click under the site column across the led[0] row to see a drop-down box appear. Type T22, tick *Fixed,* and hit the Enter key to assign the pin.



Notice that any selection that you perform in this GUI automatically updated the XDC file which has to be reloaded every time you make a change. The *Package* GUI is just a more user-friendly method to specify design constraints. The specification of pin site T22 and LVCMOS33 I/O standard for led[0] is specified by the following two Tcl commands in the *fp_add.xdc* file:

```
set_property package_pin T22 [get_ports{led[0]}]

set_property ionstandard_LVCMOS33 [get_ports{led[0]}]
```

Select *File→Save Constraints* and click *OK* to update the existing constraints file.

Synthesis

Synthesis is a process that parses the Verilog design files and generates a gate-level netlist for the whole design. After synthesis, there is no immediate correspondence between the gate-level netlist and the FPGA architecture. In other words, the netlist is in a generic gate level format.

Click on *Run Synthesis* under *Synthesis* tab. The synthesis process will be run on the top level *fpadd_system.v* file (and all its hierarchical files). When the process is completed after a few minutes a *Synthesis Completed* dialog box with three options is displayed. Select the *Open Synthesized Design* option and click *OK* as we want to look at the synthesis output before

progressing to the implementation stage. Click *Yes* to close the elaborated design if the dialog box is displayed.

Select the *Project Summary* tab (Default Layout) and understand the various windows. You can see useful information such as the utilization of the FPGA in terms of various resources such as LUT, I/Os both in Graph and Table format. Click on the *Schematic* tab under the *Open Synthesized Design* to view the synthesized design in the schematic view. Study the schematics of the design modules and observe that IBUF and OBUF gates are automatically instantiated to the design as the input and output are buffered. These gates will be later mapped to LUTs and Flip flops in the FPGA CLBs.

Verify that the *<Project Name>.runs* directory has been created by Vivado under *the <Project Name>* directory. Under the *runs* directory, *synth_1* directory is created which holds several files and scripts. For example, the **.dcp* files (Design Checkpoint), contain a continuously refined database of the design.

Note: you should always heed the *Warning* messages of Vivado and try to fix them. In a lot of cases, your Verilog code may produce correct simulation results, but it may not produce correct results on the FPGA board. Warning may reveal such problems.

Implementation (Translation, Mapping, Placement & Routing)

The *implementation* is a multi-step phase that transforms the synthesized gate level netlist into a Xilinx proprietary bitstream that can be downloaded to program the FPGA.

The *translation* phase merges multiple design files into a single netlist. This is the main database of the netlist which also includes the constraint file (XDC).

The *Mapping* phase groups gates from the netlist into physical components of the FPGA (LUTs, Flip-flops, DSP modules, IO buffers, etc.).

The *Placement* and *Routing* phase places the physical components onto the 2D chip, connects the components, and extracts timing data into reports. This is the most time-consuming task of the implementation phase.

Once a design is implemented, the user should create a file that can be downloaded and program the FPGA. This file is called a bitstream: a BIT file (*bit* extension), which can program the FPGA in one of two different ways:

- By a download cable such as a Platform USB. The bitstream is downloaded from the computer hard disk to the FPGA.
- By writing a non-volatile memory such as a Xilinx Platform Flash PROM. The bit file should first be converted into a PROM file.

Click on *Run Implementation* under the *Implementation* tab. The implementation task will run on the synthesis **.dcp* output files. When the process is completed an *Implementation Completed* dialog box with three options is displayed. Select the *Open Implemented Design* option and click *OK* as we want to look at the implementation output in the *Device* view tab. Click *Yes* to close the synthesized design.

Study the implemented design in the *Device* view and zoom in to look at the details of the final design and how it is mapped on the FPGA fabric. At the *Netlist* panel, select one of the nets and notice that the net is highlighted in the FPGA.

Close the implemented view and select the *Project Summary* (Default Layout view) and observe the results. How much is the actual resource utilization? Select the *Post-implementation* tabs under the *Timing and Utilization* Windows.

Verify that the *impl_1* directory has been created under the <*Project Name*>*.runs* directory. The *impl_1* directory contains several files including the report files. For example, several binary **.dcp* files (e.g. *fpadd_system_routed.dcp*) which contain the database of the design.

Timing simulation

Now that the FPAdder has been placed and routed in the Zynq-7000 FPGA fabric, we have all the information to perform a detailed timing simulation. Click on *Run Simulation→ Run Post-implementation Timing Simulation* under the *Simulation* tab. The Vivado simulator will be launched using the implemented design and your testbench as the top level module. Note that you have to resort to a previous project with a testbench to be able to run the simulation.

Run again the testbench and notice that the waveform shows the real and not the zero delay of the behavioral simulation. Close the simulator by selecting *File→Close simulation*

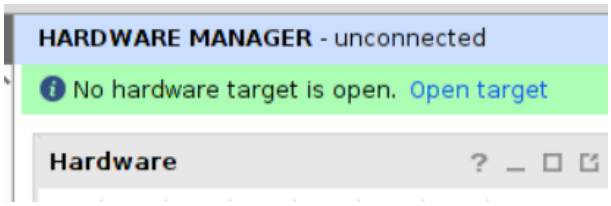
Bitstream generation and FPGA Programming

This step will allow us to download the design to the FPGA and see the N-bit gray counter running on the Zedboard. First, you should connect the board, power it ON and make sure that the micro-USB cable is connected between the board and the PC. All the jumpers JP7 to JP11 should be connected to the GND as shown in the figure. To be able to program the FPGA, we should have already installed the cable drivers.

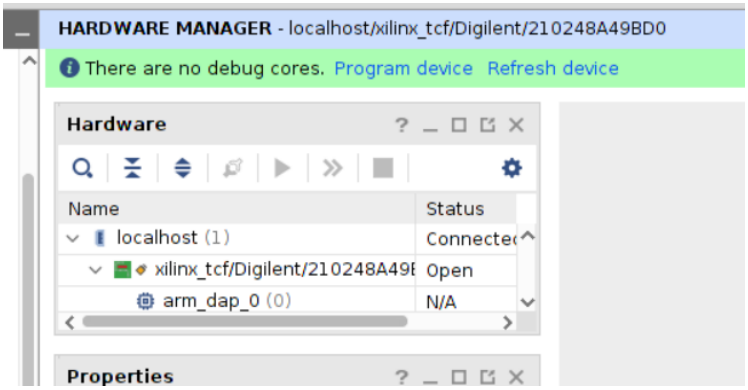
Once all these preparatory steps have been taken, select the *Generate Bitstream* entry under *Program and Debug* in the *Flow Navigator* panel. The bitstream generation process is used to generate the "executable" file that will be downloaded to the FPGA. It runs on the implemented design and, when completed, a *Bitstream Generation Completed* dialog box with three options will be displayed. The <*Project Name*>*.bit* file is generated under the <*Project Name*>*.runs*/*impl_1* subdirectory.

The final step is to transfer the bitstream *<Project Name>.bit* from your hard disk to the FPGA. First, switch on the FPGA board. Make sure that the POWER LED is on.

Select the *Open Hardware Manager* option (at the bottom of the Flow Navigator), and click *OK*. The Hardware session window will open indicating "unconnected" status as shown below. Click on the *Open Target* link and then *Auto Connect*.



The JTAG cable will be searched and the *Xilinx_tcf* should be detected and identified as a hardware target. It will also show the hardware devices detected in the chain. The hardware status changes from Unconnected to the server name and the device (i.e. the FPGA) is highlighted. Also notice that the status indicates that it is not programmed. Click *Program Device* and verify that the *<Project Name>.bit* bitstream file is selected as the programming file in the General tab.



Click *OK* to program the FPGA with the selected bitstream file. The blue DONE light will light immediately after the FPGA is programmed.

Once the last bit of the bitstream file makes it into the FPGA the design will start running. Hopefully, everything has been done correctly and you can see the correct output values of the FP Adder appearing in the two 7seg displays and the LEDs.

Close the hardware session by selecting *File→Close Hardware Manager*.

Step 4. Use Buttons to provide multiple inputs to the FP Adder

The only drawback of the platform you implemented in step 3 is that you cannot easily provide multiple inputs to the FP Adder. We are planning to fix this issue in step 4. In this part of Lab1, you will include an additional Data Memory which will provide the two inputs of the FP Adder as shown in Figure 5.

You will need to write another Verilog file to implement a Data Memory (*DataMemory.v*) of size WxH = (2*32) x NUM bits and instantiate this memory with the input values of the **.hex* file of step 1. These values will be assigned to the data memory upon reset. The Data Memory is being read each time the user presses a Zedboard button. Each time the user presses the button (shown in Figure 5), an internal read

pointer *ptr* of the Data Memory will move to the next entry *Mem[ptr]* and will provide the corresponding two input FP32 numbers to the FP adder. The pointer will wrap around to entry 0 when it reaches the NUM-1 end of the Memory.

However, there must also be an interface circuit between the button and the enable signal of the Data Memory. You also need to construct an edge detection module (level to pulse, L2P). This is an FSM-based module that receives as input a multi-cycle pulse signal and generates as output a single-cycle pulse signal. The input pulse signal is generated by the button in the board and can last any number of clock cycles. It corresponds to the user pressing the button. However, such an input pulse that lasts multiple cycles N can trigger as many counter increases as the number of cycles N, which is not what the user requires for the system.

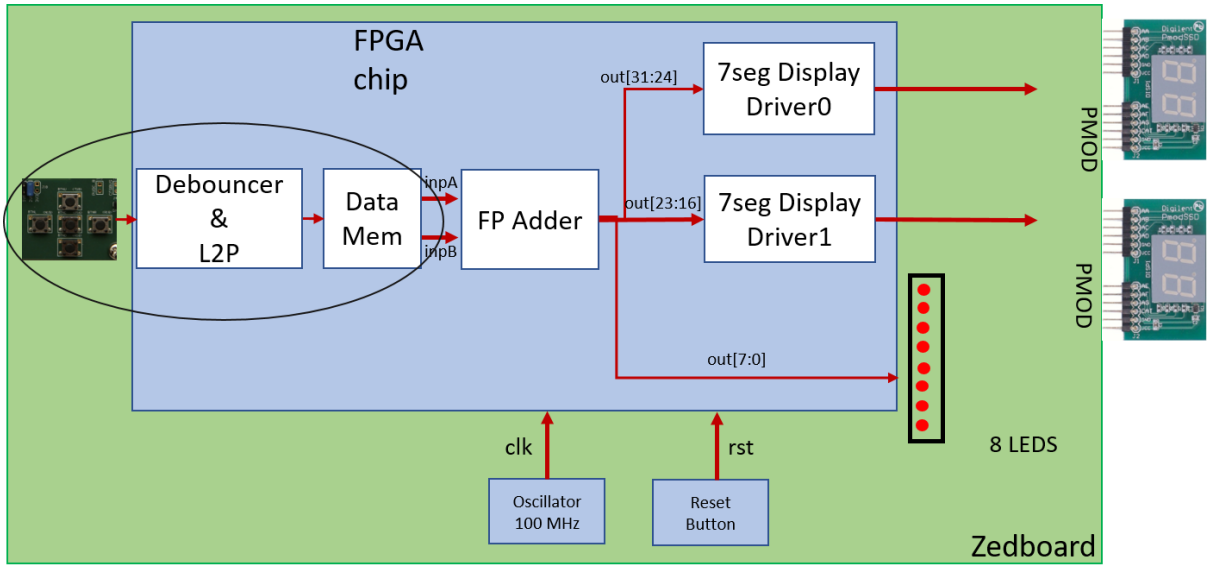


Figure 5. Zedboard platform for step 4. The additional hardware of step 4 is shown circled.

This FSM-based edge detector will produce an output that can be connected to the input enable signal of the Data Memory. The input to the edge detector is a button of the Zedboard.

What do you observe? Does your system work?

Debouncing

A problem arises from the mechanical "bounce" inherent in switches: as a metal contact opens and closes it may bounce a couple of times, creating a sequence of on/off transitions in rapid succession. So, you need to use debouncing circuitry to filter out these unwanted transitions (Figure 6). You should design a Verilog implementation of a digital debouncer that requires that an input transition be stable for (approximately) T=0.1 sec before reporting a transition on its output. You should use an instance of the module to debounce any switch inputs you use in your designs.

Use a counter to produce an *enable* signal when the input noisy signal has flipped and has remained stable for T=0.1secs. If the input signal changes sooner than T secs, we assume that it was mechanical noise, and we do not pass the noisy signal to the output (clean) signal.

When you finish the implementation of step 4, you should be able to scan the Data Memory for input data each time you press the button, and observe the (hopefully) correct output at the two 7seg displays and the LEDs.

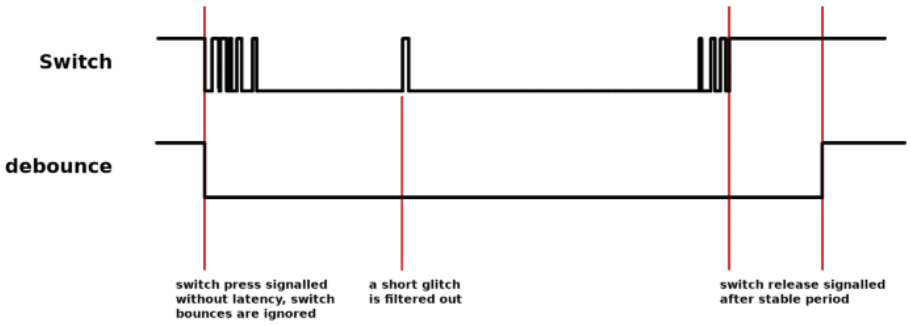


Figure 6. The Debouncer functions as a low pass filter to eliminate the spurious signals due to mechanical bounces.