ECE385
DIGITAL SYSTEMS LABORATORY
Introduction to Xilinx Vivado

A Simple Design in SystemVerilog Using Xilinx Vivado 2022.2 Edition

Design a full-adder and use it to design a 2-bit adder:

You need to be on a machine with Linux or Windows 10 or 11, either on your own machine or in the ECE Open Lab (3022 ECEB). A Linux version is also available online for your own Linux system. To perform this exercise on your own computer, you will need to install the Vivado software (you may download the installer from the links below) or install it on a virtual machine. Follow these steps to fully install the software:

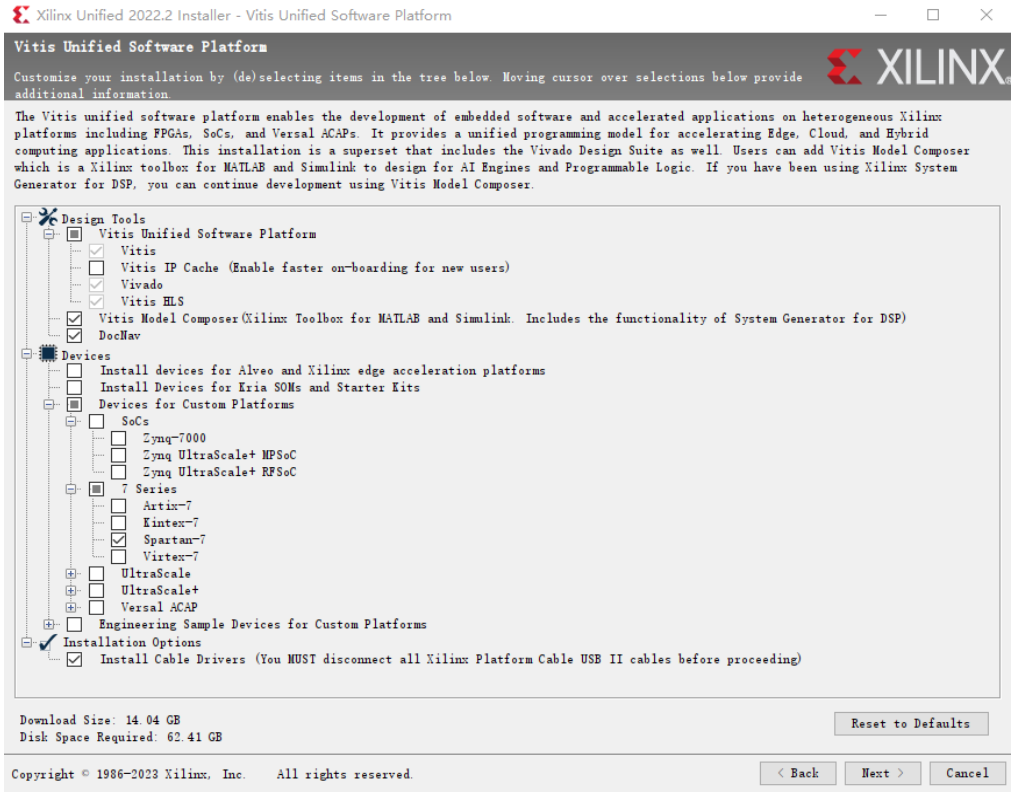


Figure 1 - Minimal Install for ECE 385

Option 1: Download and Install Directly from AMD Xilinx Website

- Install Vivado ML Edition 2022.2, Vitis Unified Software Platform, and Spartan-7 device support from the AMD Xilinx website (https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2022-2.html). You should download the ***Xilinx Unified Installer 2022.2***.

- You will need to create a Xilinx account. Enter the information at Xilinx download center, press download.
- Open the installer, press ***Next***. Fill in your User information for your Xilinx account, choose ***Download and Install Now***, click ***Next***. Select ***Vitis*** as the product to install, click ***Next***. Mark related boxes for installation guided by Figure 1, click ***Next***. Select installation destination, disk usage is around 63 GB. Click ***Next*** and start installation.

Once the installation has completed, you are now ready to begin using Vivado. Below is a sample 2-bit adder project.

Option 2: Use the provided Virtual Machine:

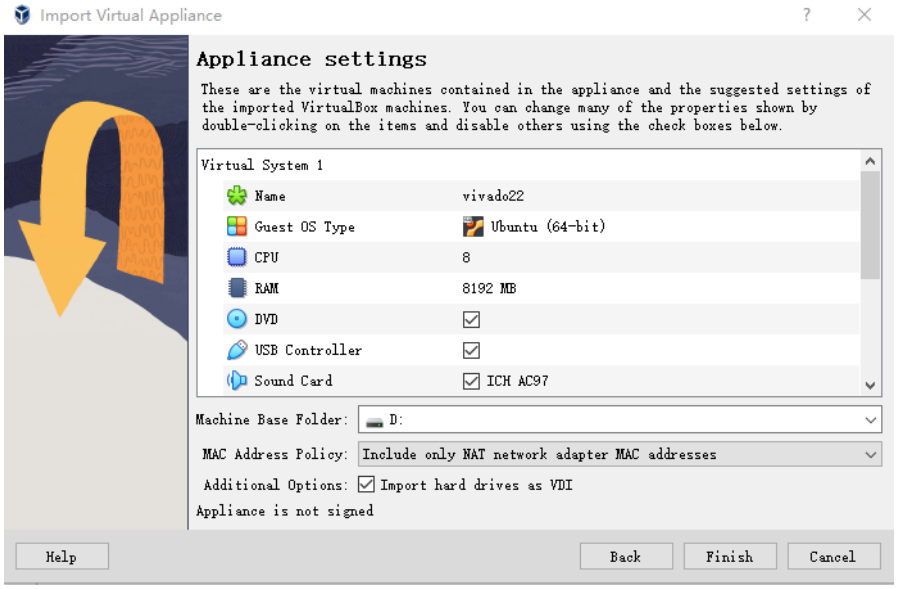


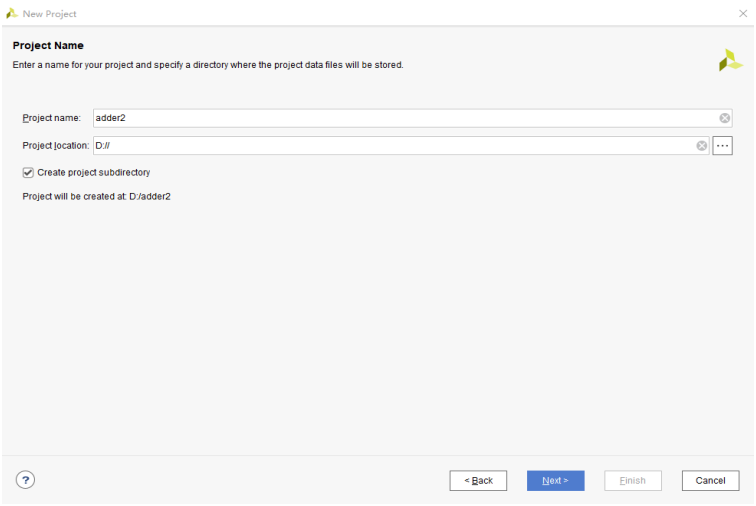
Figure 2 - Virtual Machine Settings

- Use the provided Virtual Machine that has Vivado ML Edition 2022.2, Vitis Unified Software Platform, and Spartan-7 device support already installed. (You will need at least 130 GB Disk space for the VM). The virtual machine can be found on the course Canvas page).
- Download and install Oracle VM VirtualBox version 7.0.8 from https://www.virtualbox.org/wiki/Downloads following instructions. (Skip this step if you already have VirtualBox or VMWare installed.)
- Make sure you install any specific host addons (e.g., for the USB drivers).
- Open Oracle VM VirtualBox, select ***Tools - Import***. Choose the downloaded .ova file, select ***Next***.
- Select a folder for the ***Machine Base Folder***. You need to make sure the drive has more than 130 GB. Note that the machine base folder may be on an external USB drive if your main drive does not have sufficient free space. Select ***Finish*** *and* wait for the VM to be imported. See Figure 2 for the default settings. Note that if you have fewer than 8 CPU cores you will need to reduce the number of virtual cores. Similarly, if you only have 8GB of RAM total, you will need to reduce the total RAM to 7GB or lower. Computers

with < 8GB of RAM will be unable to run VirtualBox with Vivado.

Create a New Project:

- Open ***Vivado 2022.2***.
- From ***Quick Start*** select ***Create Project***. Click ***Next*** to pass the intro screen.
- The window in Figure 3 will appear. Name the project ***adder2*** and choose the project location (make sure there are no spaces and no special characters in any of your entries). In addition, make sure the path is as short as possible. Select ***Next***.
- Select ***RTL Project*** for the Project Type and do not mark the boxes yet. Click ***Next***.
- Do not add any sources yet, click ***Next***.
- Do not add any constraints yet, click ***Next***.
- A window in Figure 4 will appear, search, and select ***XC7S50CSGA324-1***, click ***Next***.
- Click ***Finish***.
- You should now see the ***Project Manager*** window.



An Important Note on Project Location:

If you are using any of the EWS labs (including 3022 ECEB), do not work directly from the U:\ (EWS Home) drive. Not only will the performance very poor when other students are in the labs - some later labs, especially those requiring Vitis SDK, will not build correctly as some of the underlying tools do not recognize network paths. Instead put your project in a simple directory in the C:\Users\<netid> folder, for example:

C:\Users\ljames23\ece385\lab2\

Note that this path has no spaces and is relatively short. In addition, other students will not have permission to access your "Users" folder, so you will not have to worry about students plagiarizing your code. Keep in mind that you will have to copy the project back into your U:\ drive to switch computers, as the "Users" folder is not synchronized across computers.

An alternative is to work from a USB 3 flash drive or portable SSD. Just be sure to back up your data often as USB drives often fail.

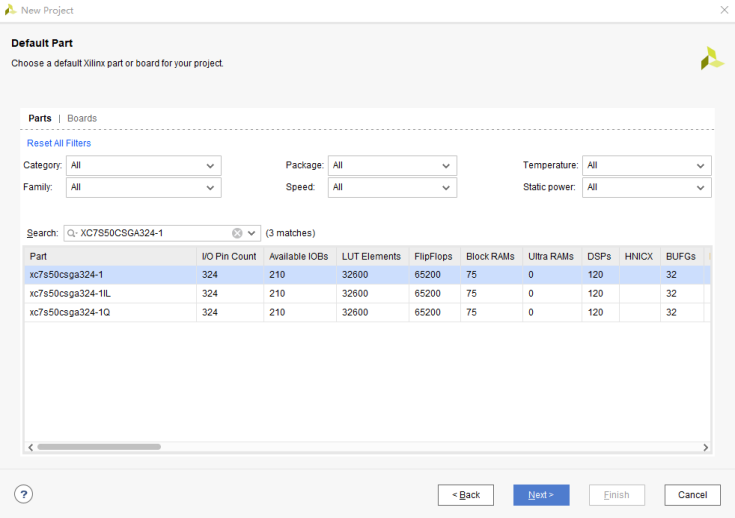


Figure 4 - Selecting the Correct Part

Create a Full-Adder Entity:
Now you can start entering your design. In the ***File*** menu, select ***Add Sources***. Select ***Add or create design sources***. Choose ***Create File***. A window in Figure 5 will pop up. Select ***SystemVerilog*** for the file type and name the file ***full_adder.sv***. Press ***OK***. Press ***Finish***. Name the module ***full_adder*** and press ***OK***. You will be presented with a blank SystemVerilog file full_adder.sv with module full_adder as shown in Figure 6. Enter the code below to define the full adder unit and save the file. Figure 7 shows schematic representations of what the code defines; if you do not understand what the code means, you should review the Introduction to SystemVerilog on pages IST.1-25.

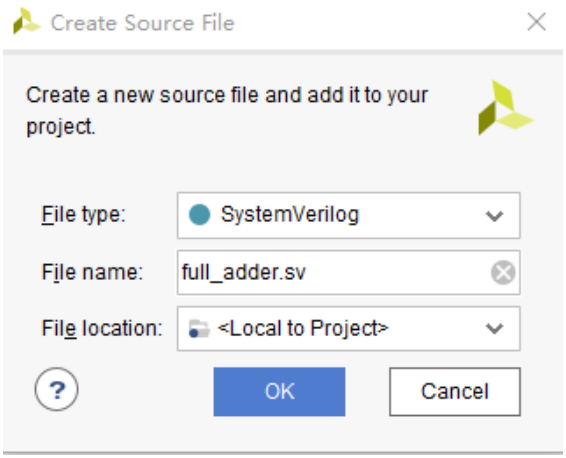


Figure 5 - Create New Source

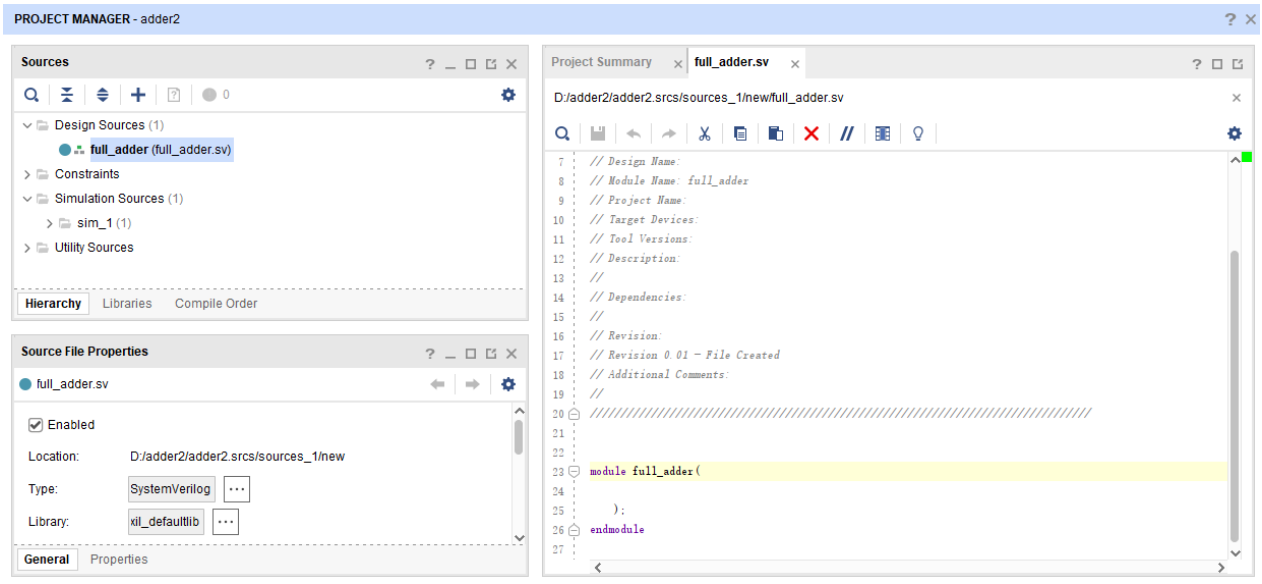


Figure 6 - Project Manager

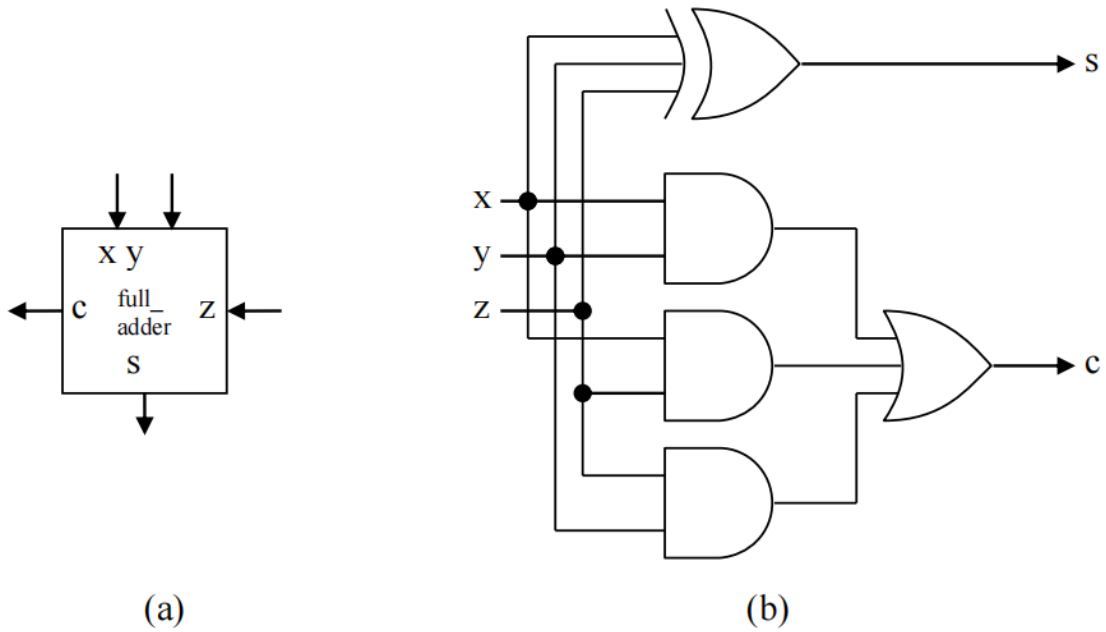


Figure 7 – a) Module of Full Adder. b) Internal Circuit

```
module full_adder (input x, y, z,
      output s, c);

        assign s = x^y^z;
```

```
    assign c = (x&y)|(y&z)|(x&z);

endmodule
```

Create an Entity for 2-Bit Adder:
Now, follow the same steps to create a new file that defines a 2-bit adder with inputs A (2-bits), B (2-bits) and Cin, and outputs Sum (2-bits) and Cout. The code is given below. Save the file as adder2.sv.

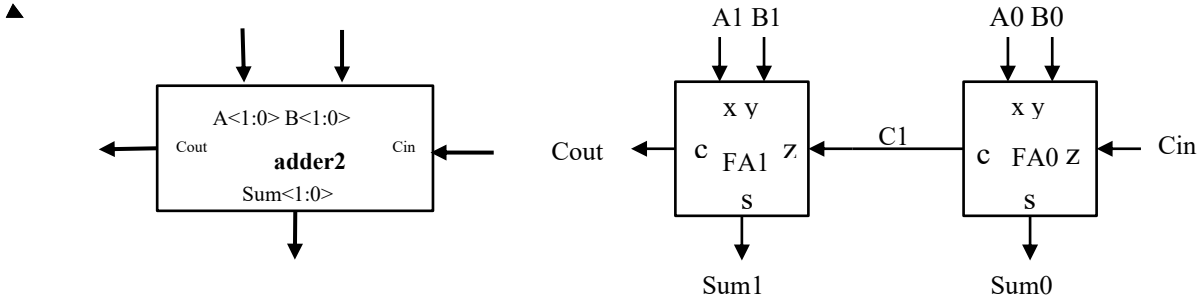


Figure 8 - a) adder2 Top-level Connections b) adder2 Internal Submodule Instantiations and Connections

---

```
module adder2 (input logic [1:0] A, B,
    input logic c_in,
    output logic [1:0] S,
    output logic c_out);

        // Internal carries in the 2-bit adder
        logic    c1;

        full_adder  FA0 (.x (A[0]), .y (B[0]), .z (c_in), .s (S[0]), .c (c1));
        full_adder  FA1 (.x (A[1]), .y (B[1]), .z (c1), .s (S[1]), .c (c_out));

endmodule
```

Once you have entered and saved the code for adder2.sv, click the ***Run Synthesis*** button (▶ Run Synthesis) in the project manager at the side bar. Vivado will check the code for correct syntax and generate errors or warnings if there is anything wrong or questionable about your code. A message pops up when the program has finished; you should get no errors or warnings if you have entered the code correctly. If you find errors in synthesis, correct the code, save the file, and run ***Run Synthesis*** again. Vivado also builds the hierarchy in the ***Design Sources*** window, which should look like Figure 9. Now you have created a 2-bit adder and you can simulate the design.

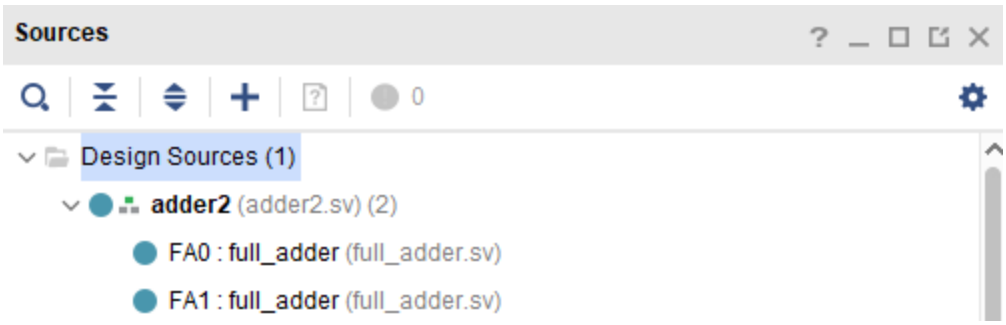


Figure 9 - Design Sources Panel

Using Simulator:
To run provided testbenches in Vivado using vSim, please follow the instructions starting from IVT. 19. To create your own testbenches, please watch the "Introduction to vSim" tutorial video on the course website.

Entering Pin Assignments:
Click the ***Layout - I/O Planning*** in the menu to open the I/O Planner. If that option is not available, remember to open your synthesized design first. You will see the pin layout of the Spartan 7 chip on the top right of the window, as well as the pin assignment table at the bottom of the window. Enter the assignments from Table 1 in sequence. The planner should look like Figure 10. Save the assignment as a constraint file (.xdc) and run synthesis again. Index with single-colored polygon background means that this pin has been assigned a signal. Index with polygon and a rectangle means this pin has been assigned a signal.

Port Name	Location	I/O Std	Comments
Cin	PIN_G1	LVCMOS33	On-board slider switch (SW0)
A[0]	PIN_F2	LVCMOS33	On-board slider switch (SW1)
A[1]	PIN_F1	LVCMOS33	On-board slider switch (SW2)
B[0]	PIN_E2	LVCMOS33	On-board slider switch (SW3)
B[1]	PIN_E1	LVCMOS33	On-board slider switch (SW4)
S[0]	PIN_C13	LVCMOS33	On-Board LED (LD0)
S[1]	PIN_C14	LVCMOS33	On-Board LED (LD1)
Cout	PIN_D14	LVCMOS33	On-Board LED (LD2)

Table 1: Pin Assignments

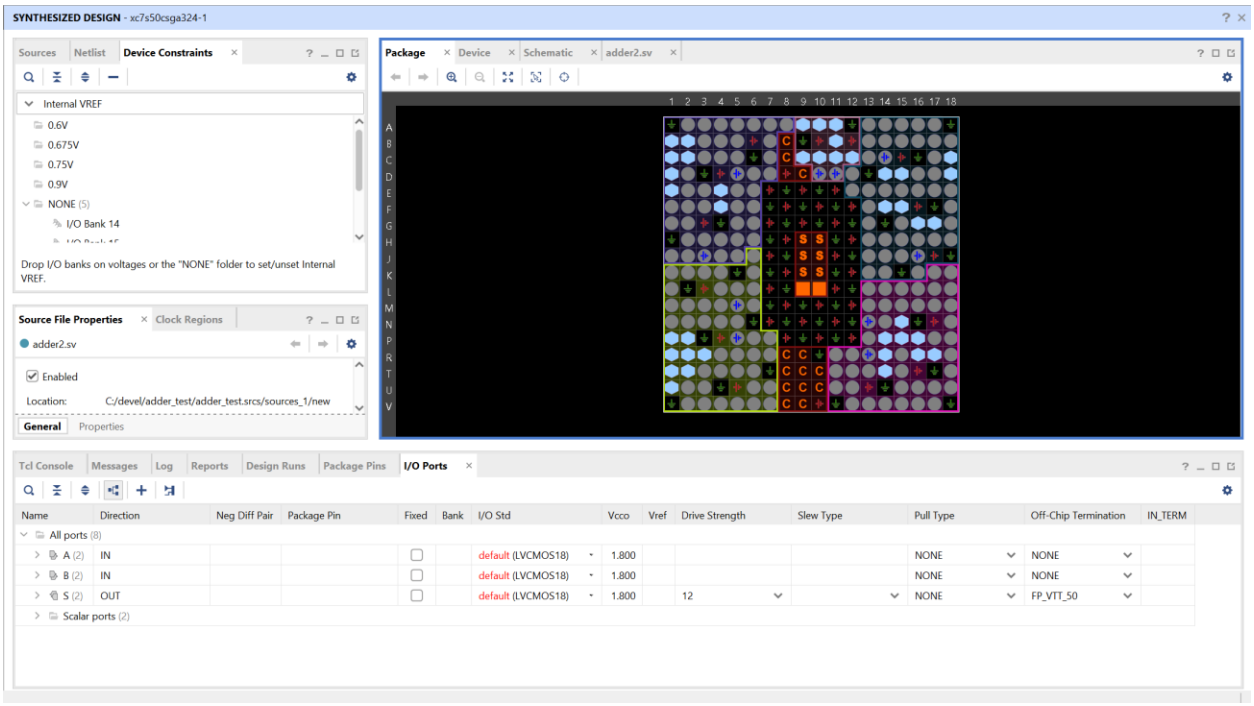


Figure 10 - Pin Packager Tool

Run Implementation

Next click the ***Run Implementation*** button (Run Implementation) in the sidebar. This time, Vivado will translate the synthesized design into a physical implementation on the target FPGA or SoC device. It consists of several steps, including placement, routing, and timing analysis.

Programming the FPGA:

Plug in the FPGA to your computer's USB port. You should see the ***PWR SEL*** led light up (blue) on your board. Turn on the board by flipping the ***ON/OFF*** switch on the board. You should see several LEDs on the board light up. When you finished Running Implementation, clicking on the button ***Generate Bitstream*** Generate Bitstream . Note that if you did not finish the pin assignment step, or you have unassigned pins, this step will fail with error messages. Please make sure you have all the signals on the top-level assigned to a pin on the board. If you are using the virtual machine, you will have to 'capture' the Urbana board's USB connection to allow the VM to communicate directly with it. You can do this by clicking on the USB icon in VirtualBox and checking the Xilinx JTAG+Serial device as shown in Figure 11.

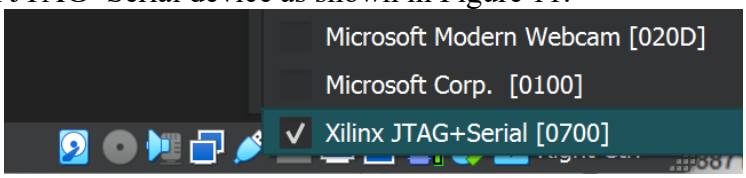


Figure 11 - Capturing USB in VirtualBox

When you have the bitstream ready, press ***Open Hardware Manager***. Choose ***Open target - Auto connect.*** You should see the window shown as Figure 12.

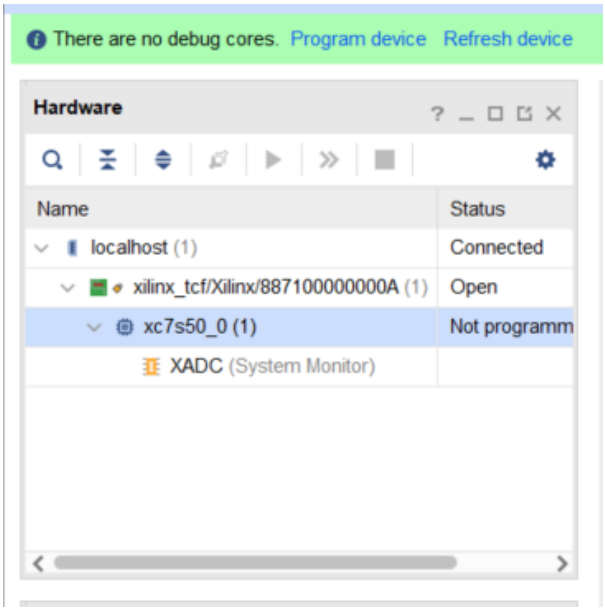


Figure 12 - Programming Panel

Press ***Program device - Program***. Note that you should be able to see the generated bitstream file (.bit). Now your board has been programmed with the hardware described by your SystemVerilog code! Try toggling the various switches and confirm the behavior on the LEDs operate as expected.

Summary of Steps for Performing Experiments Involving SystemVerilog in the Lab

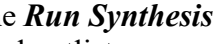
Creating a Project

- Open ***Vivado 2022.2***.
- From ***Quick Start*** select ***Create Project***. Click ***Next*** to pass the intro screen.
- Name the project and choose the project location (make sure there are no spaces and no special characters in any of your entries). Select ***Next***.
- Select ***RTL Project*** for the Project Type, do not mark the boxes yet. Click ***Next***.
- Add source files (skip if you do not have any yet), click ***Next***.
- Add constraint files (skip if you do not have any yet), click ***Next***.
- Select ***XC7S50CSGA324-1***, click ***Next***.
- Click ***Finish***.
- You should now see the ***Project Manager*** window that a project has been created.

Adding Files to a Project

- Often for some labs (but not all) you are provided with some files as a starting point. If you didn't add your files in the step above, or if you wish to add additional files, select ***Add Sources*** in the ***File*** menu. Select the file(s) you want to add by clicking ***Add Files*** or ***Add Directories***. Constraint files are usually .xdc files including pin assignments, timing constraints, clock definitions, I/O standards, and other design-specific requirements; design sources are HDL code .sv files; simulation files are usually testbench .sv files. **Make sure you check "Copy Sources into Project".** If not, your project will have links to files which are likely in a temporary directory, and this may corrupt your project.
- Vivado is reasonable smart about figuring out the top-level of your project, but sometimes you may need to manually set the top-level. As the name implies, this is the module which represents the entire physical design, which presumably contains and instantiates the other submodules. You may do this by right clicking on the relevant file under Design Sources and clicking ***Set as Top.***

Simulating, Analyzing, Synthesizing, and Implementing the Design

- Click the ***Run Simulation*** and select ***Run Behavioral Simulation*** to start running simulation. It will elaborate the design and come up with a waveform. If you have a testbench included, you will be able to see the design running given the input of your testbench. If there are no errors, you will see a window pop up saying Synthesis successfully completed. You should see ***Error*** or ***Warning*** at the Messages window below. You should follow the error messages and find the log files for the reason for the errors.
- Click the ***Run Synthesis*** () to begin synthesis. This step, Vivado generates a gate-level netlist representing the design. The netlist contains instances of FPGA-specific logic elements, such as lookup tables, flip-flops, and interconnects, based on the technology mapping and optimization. If there are no errors, you will see a window pop up saying Synthesis successfully completed. You should see ***Error*** or ***Warning*** at the Messages window below. You should follow the error messages and find the log files for the reason for the errors.

- You will need to fix any errors in your design for the above steps before continuing. Additionally, you may receive warnings from the various steps. You should correct all warnings from the ***Run Synthesis*** step. Warnings from the other steps may be acceptable but check with a TA if you are encountering issues.
- Next click the ***Run Implementation*** button (Run Implementation) in the sidebar. This time, Vivado will translate the synthesized design into a physical implementation on the target FPGA or SoC device. It consists of several steps, including placement, routing, and timing analysis.

Viewing the Synthesized Design

- Once the design is fully compiled, you can view the synthesized circuit in ***RTL ANALYSIS > Open Elaborated Design > Schematic*** in an interactive GUI.

Setting pin mappings and programming the Urbana Board

Once your design simulates correctly, you can download the design bit-stream onto the Urbana board and test it.

- First you must provide pin assignments for inputs and outputs of the top-level. Click the ***Layout - I/O Planning***.
- This will open the Pin Planner Editor. The list of circuit input and output pins should have been automatically listed at the bottom portion of the window. The list of assignments can be found in the lab manual entry for the current lab. Type in these assignments. (If you prefer, you can double-click the fields in the editor and select signals and pins from a dropdown menu.)
- Once you have entered the pin assignments for all ports, save the assignment editor view and follow the ***Run Synthesis*** (Run Synthesis) and ***Run implementation*** (Run Implementation) flow again.

Downloading the design to the Urbana Board

The Programmer sends the completed bitstream to the FPGA.

- Connect the programming cable to the computer's USB port and the board's ***USB Blaster*** connector.
- Power on the dev board by flipping the switch on the top right of the board to ***ON***.
- Click ***Program and Debug > Generate Bitstream*** (Generate Bitstream) in the ***Flow Navigator***
- Click ***Yes*** in the pop-up window. Wait until another popup window and click ***Ok***
- If the previous step was successful, Click ***Program and Debug > Open Hardware Manager>Open Target.*** Then click, ***Auto Connect***
- If the previous step was successful, Click ***Program and Debug > Open Hardware Manager> Program Device.*** You can also right click, ***xc7s50_0*** and select ***Program Devic***e.
- Check the box under ***Program/Configure*** to select the programming file.
- Click ***Program*** in the pop up. The FPGA will be programmed with your design.
- Once the programming process is completed, test your design for functionality.

- If your design does not work or exhibits bugs, alter your design, recompile, and reprogram the FPGA. If you cannot easily determine how to fix your design, design a simulation to capture the incorrect behavior to better understand what is going wrong.

I/O Pins available on the Urbana Development Board

Pins used in the lab exercises are listed in the manual description for each experiment. A full listing of the devices and associated pins on the Urbana board development board can be found in the Urbana datasheet, which can be found on Canvas under resources.

Vivado Tutorial Exercise – Bit-Serial Processor in SystemVerilog

The 8-bit serial processor will be very similar to the design you have done for Experiment 2.1. The only difference is that you will now use two 8-bit shift registers to store the data inputs and the result. You will first load the registers with the values that you want to be used as operands. Then you will specify the operation that you want to perform on the data and specify the register that should hold the result. The result will be computed using bit-wise operators. Every cycle during computation, the least significant bits from each register will be shifted out, desired logic operation will be performed on the two bits, and the result bits will be shifted into the registers as the most significant bits. After repeating this procedure eight times, the specified register will contain the 8-bit result.

Task: Download the 4-bit module files from the course website (under Lab 2.2). Use it to build an 8-bit serial processor project using the provided modules by creating a project, adding the provided files to the project and test the circuit operation using the included testbench. Follow the SystemVerilog tutorial ISV and the instructions for testbench (**IVT. 19-22**) to complete the exercise. Include a copy of the schematic block diagram and the simulation waveform (with annotations) in your Lab 2.2 lab report.

Your circuit should have the following inputs and outputs (once it has been extended to 8-bits):

Inputs

```
Clk, Reset, Execute, LoadA, LoadB – logic
Din – logic [7:0]
F – logic [2:0]
R – logic [1:0]
```

Descriptions of the inputs:

`Clk`: Clock signal for the processor
`Reset`: Reset signal to initialize the controller to the start state.
`Execute`: Signal that triggers the execution of a computation cycle.
`LoadA`: Signal that loads Register A with the data specified by the input switches (`Din<7:0>`).

LoadB: Signal that loads Register B with the data specified by the input switches (Din<7:0>).
Din<7: 0>: Input switches for the input data
F<2:0>: Function select
R<1:0>: Router select

Outputs
Aval, Bval – logic [7:0]
hex_seg – logic [7:0]
hex_grid – logic [3:0]
LED – logic [3:0]

Descriptions of the outputs:
Aval<7:0>, Bval<7:0>: Binary values in registers A, B
hex_seg – logic [7:0]: Segment (Cathode) control for left HEX driver.
hex_grid – logic [3:0]: Grid (Anode) control for left HEX driver.
LED<3:0>: Concatenated values of Execute, LoadA, LoadB, & Reset signals, to be displayed on LEDs.

Note: For Experiments 2.2-7, you should use the same input/output port names as provided for each experiment in this manual.

The block diagram for the circuit is shown in Figure 13. The register unit contains two 8-bit registers, Register A and Register B. The computation unit performs the desired logical operation based on the function select (F<2:0>). The routing unit selects the input bits to Register A and Register B after the computation. The control unit provides control signals (e.g. load, shift) to the register unit.

Control Unit

The control unit will accept Clock, LoadA, LoadB, Execute, and Reset as inputs. It will output the control signals to the register unit to instruct the registers when to load the values from the switches and when to shift the register values. The Reset signal, when set to high, will put the controller in the reset/start state. A 0->1 transition on the Execute signal will trigger the execution of a computation cycle. Once the computation cycle has started, the controller should ignore the Execute signal until the appropriate logical operation is performed on all bits and the registers contain appropriate 8-bit values. That is, during execution, the Execute signal is a 'Don't Care'. The processor should halt at the end of a computation cycle until the Execute signal is set to low. Another computation cycle can then begin when the Execute signal again switches from low to high. Load A and Load B should only be allowed to take effect in the reset/start state and should be ignored during the execution of a computation cycle.

Register Unit

The register unit will contain two 8-bit registers, Register A and Register B. The control signals to parallel load the values of the data inputs (Din<7:0>) to one or both registers and to

shift in the result bits from the routing unit will come from the control unit. During a computation cycle each register will shift-out one bit at a time to the computation unit as the result bit from the routing unit shifts in. The 8-bit values of the registers will also be provided as outputs that the user can observe.

Computation Unit

The computation unit will perform a logical operation based on the function select (F<2:0>) on the operands provided by the register unit. The computation unit will output three one-bit values to the routing unit – A, B, and F(A, B). Table 1 provides the functions associated with the value of F<2:0> and the values that should be transmitted to the register unit from the routing unit based on R<1:0>.

Routing Unit

The routing unit will accept A, B, and F(A, B) as inputs and will output A' (shift-in to Register A, newA) and B' (shift-in to Register B, newB) based on R<1:0> as shown in Table 1.

HexDriver Units

Each HexDriver unit accepts one 16-bit binary input and outputs the anode and cathode controls necessary to correctly display this as 4 hex digits on a 7-segment display. You will instance 1 HexDriver unit, because you need to display 16 bits of data.

Pin Assignment Table

For the simulation portion of Lab 2.2, you will not need to have pin assignments. However, you will need to have pin assignments for the ILA (Integrated Logic Analyzer) portion (refer to Appendix II of the Lab 2 manual for recommended pin assignments for this portion). If you do not provide explicit pin assignments, Vivado will raise an error during the implementation process. It will indicate that there are unplaced I/Os (inputs/outputs) and prompt you to assign the pins manually. Also, on some versions of Vivado, the pin placement graphical tool is occasionally buggy. It will prompt you to unplace unrelated signals when placing certain signals, which will make it seem like it's impossible to map all the pins for a given assignment. If you get this bug, try re-sorting the list of pins. For example, sorting them by "Package Pin" seems to work around this bug.

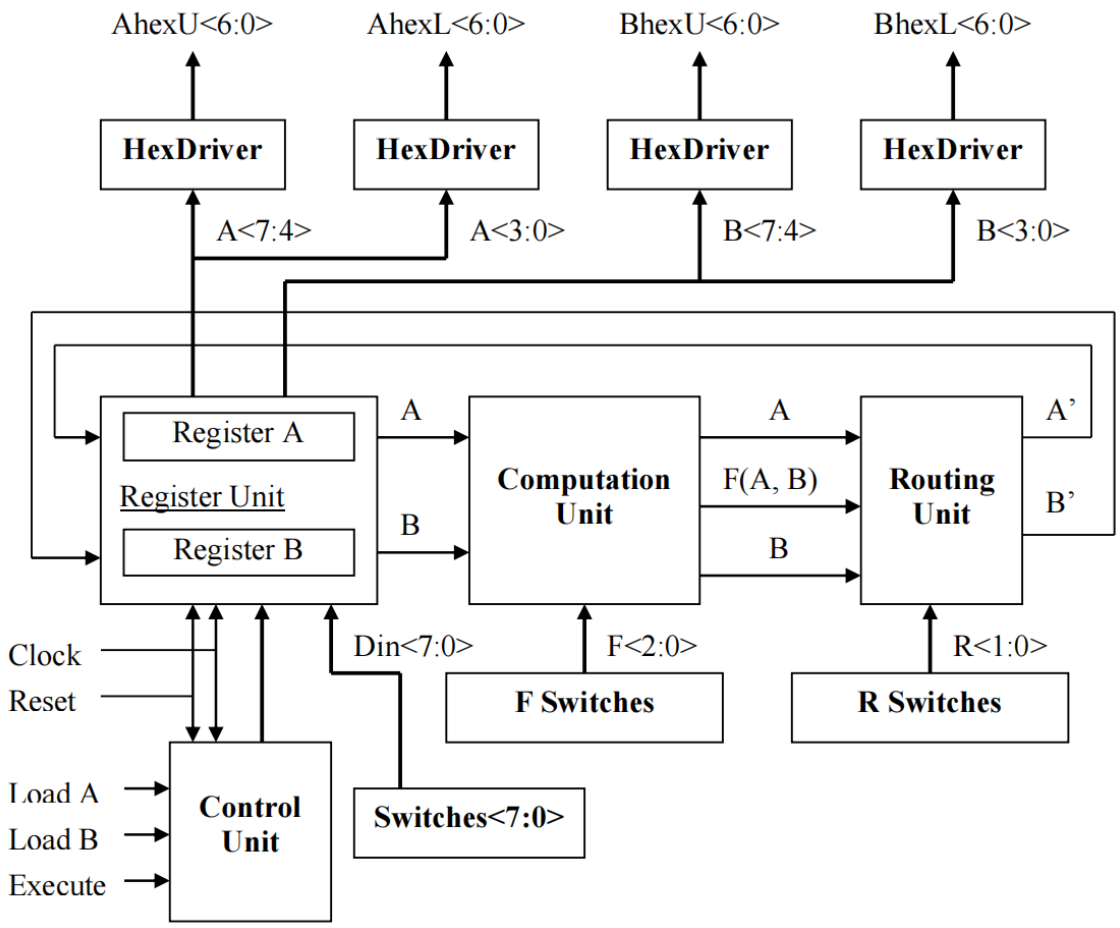


Figure 13 - Block Diagram for the 8-bit Logic Processor

Table 2. Function Table

Function Selection Inputs			Computation Unit Output
F<2>	**F<1>**	**F<0>**	**f(A, B)**
0	0	0	A AND B
0	0	1	A OR B
0	1	0	A XOR B
0	1	1	1111
1	0	0	A NAND B
1	0	1	A NOR B
1	1	0	A XNOR B
1	1	1	0000

Routing Selection		Router Output	
R<1>	**R<0>**	**A'**	**B'**
0	0	A	B
0	1	A	F
1	0	F	B
1	1	B	A

Design Resources and Statistics

On modern Xilinx FPGA, the basic logic building block is called a slice. In the case of the Spartan 7 FPGA, each slice consists of 4 6-input look-up tables (LUTs) and 8 1-bit flip flops. In FPGAs, all the combinational logic is implemented in LUTs. To find out the usage of LUTs after compilation, go to the ***IMPLEMENTATION*** tab under ***Flow Navigator***. Go to ***Open Implemented Design***. In the Implemented Design view, you will see various sections with statistics such as:

- Report Utilization: This section displays information about the utilization of FPGA resources like LUTs (Look-Up Tables), FFs (Flip-Flops), BRAMs (Block RAMs), DSPs (Digital Signal Processors), etc.
- Report Timing Summary: This section provides information about the design's timing characteristics, including critical paths, slack, and clock frequencies.
- Report Power: This section shows power-related statistics, including estimated power consumption, power supply network information, and dynamic power reports.

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	745300	1182240	63.04
LUTRAM	437361	591840	73.90
FF	999619	2364480	42.28
BRAM	335	2160	15.51
DSP	3	6840	0.04
IO	122	832	14.66
BUFG	10	1800	0.56
MMCM	2	30	6.67
PLL	3	60	5.00

Figure 14 - Implementation Summary Table

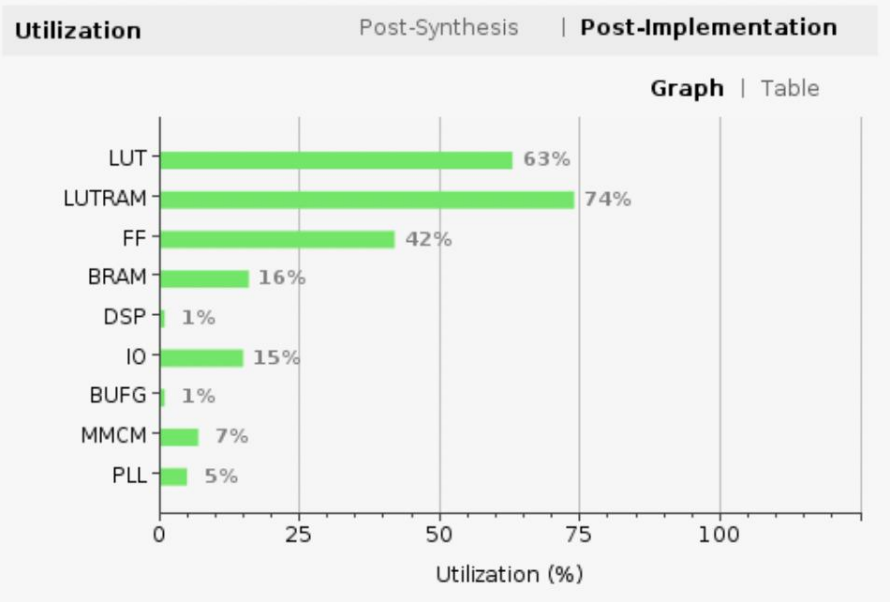


Figure 15 - Implementation Summary Graph

As shown in Figures 14 and 15, the total number of LUTs (or Look-Up tables) used as well as the utilization percent is reported. LUTs are programmable logic elements that allow FPGAs to implement arbitrary combinational logic functions. Next, the number of FF (flip-flops or registers) is reported. Flip-flops provide the storage elements necessary for sequential circuit behavior such as storage units and state machines. Below, we can find a report of BRAM usage. There is an on-chip RAM block on the Xilinx Spartan-7 XC7S50 FPGA, sizing 2700 Kbits, which we call block memory (BRAM). BRAM is not used in most of the SystemVerilog labs, but in the SoC labs, it is required to store the software for the MicroBlaze processor. The reported number gives you the usage of BRAM in the design. Next, we can see the total number of DSP (Digital Signal Processing) blocks used. These DSP blocks are dedicated resources within the FPGA that are optimized for performing DSP-related operations, such as multiply-accumulate (MAC) operations, complex arithmetic, and other mathematical computations commonly found in digital signal processing algorithms.

You will also see Design Timing Summary under ***IMPLEMENTATION*** > ***Open Implemented Design*** > ***Report Timing Summary***. Click ***OK*** to generate the timing report using the default options. To perform a detailed analysis of the operating frequency, Vivado needs to be provided with input and output timing constraints of the circuit, using the Xilinx Design Constraints (.xdc) format. These constraints usually arise when the circuit designed for the FPGA is not a standalone circuit, but rather it is merely a part of a larger system. The other parts of the system often have their own operating constraints, which eventually become the input and output constraints for the FPGA circuit. However, since most of the FPGA designs in our labs are standalone systems with manual inputs, it is not necessary to come up with complicated I/O constraints yet (this will change when we need to interface with other integrated circuits that operate on the order of nanoseconds, rather than LEDs and buttons that operate on the order of milliseconds). However, **for all designs which use clocked elements**, we at least need to specify

the main clock speed for our design which is fixed via the crystal oscillator on the Urbana board. You may add this simple timing constraint by opening your synthesized design and clicking ***IMPLEMENTATION*** > ***Open Implemented Design*** > ***Edit Timing Constraints.*** Then click the + icon next to Create Clock. The tricky thing here is that the Clock name is counterintuitively what other timing constraints will refer to the clock as, so you can enter a generic name such as "clk_100" here – it is even acceptable to leave blank. However, it is important that Source Objects reflect the actual hardware port of the clock is (e.g., from the top-level module port definition). A good way to double check the name is to click on the (…) button and then search for all the inputs and find the clock signal (for the early labs this will be called "Clk") and double click on it. The clock period of the external oscillator is 10ns, so the defaults for the clock period/rise/and fall time are fine. Make sure you save the changes to the .xdc when you return to the constraints editor. Note that the timing constraints and the I/O pin assignments are both stored in the same .xdc file, so for later labs this material will be provided.

For the Power analysis, you can find the power report under **IMPLEMENTATION** > **Open implemented Design** > **Report Power**. You will find a diagram like in Figure 15. You can find the Dynamic power and the Device Static power. Dynamic power refers to the power consumed by an FPGA during its operation because of switching activities within the design whereas Device Static power refers to the power consumed by an FPGA even when there is no switching activity or clock signal. Dynamic power composed of Signals, Logic and I/O power. Signals Power refers to the power consumed by the FPGA due to switching activities of the internal signals within the design; Logic Power refers to the power consumed by the combinational logic elements within the FPGA; I/O Power refers to the power consumed by the FPGA's input/output (I/O) interfaces and circuitry.

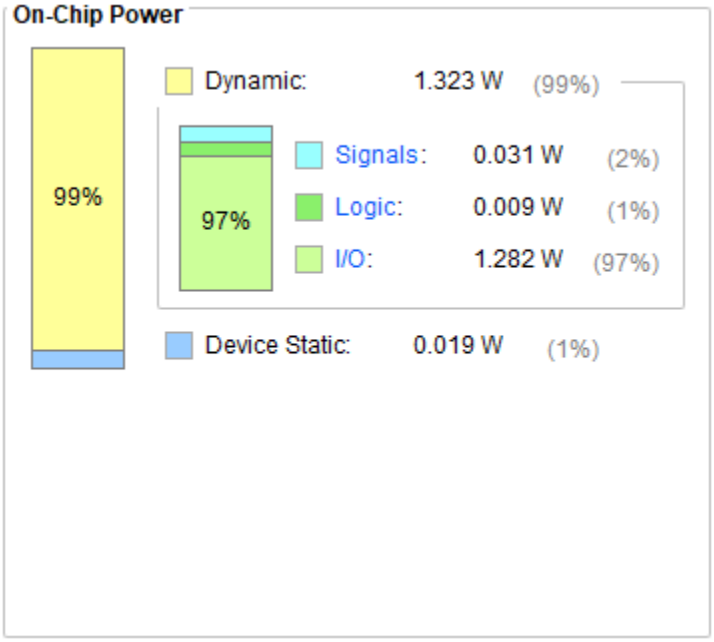


Figure 16 - Power Summary

Instructions for Testbench for the Bit-Serial Logic Processor

These instructions are a tutorial for creating a testbench that generates simulation waveforms and results for the bit-serial processor in the Vivado 2022.2 environment, with the use of its built-in simulator. It is **required in the demo**.

A testbench envelopes the design under test (DUT), provides it with test stimuli, collects the results, and analyzes it, as illustrated below. In the bit-serial processor, the DUT is the processor module. We will use the testbench to provide some inputs for testing. The results can be viewed as waveforms in Vivado.

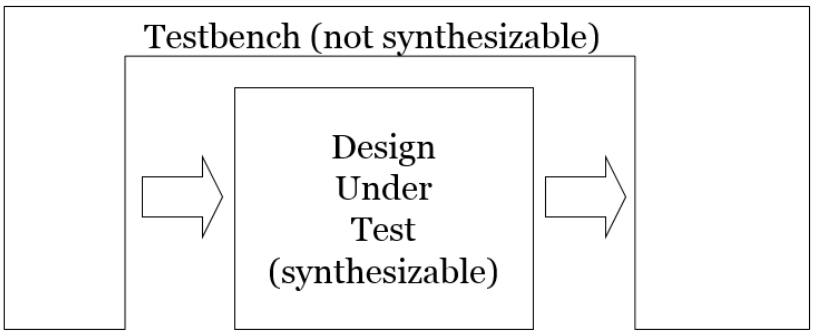


Figure 17 - Conceptual Testbench

Note that we have so far emphasized the synthesizability of SystemVerilog coding because the code in concern describes the design itself. On the other hand, a testbench is used in simulations and is not required to be synthesizable on hardware. Therefore, some syntax used in the testbench is not synthesizable. Pay attention to the differences and avoid using syntax that is not synthesizable in the designs!

Preliminary Work:
Build the bit-serial processor project and include all the provided code. For the "Processor (Main Module)" part, modify the provided 4-bit logic processor to the 8-bit variant. Make sure the top-level "Processor" module uses all the port names given in the lab manual because they will be used in the testbench. Notice that if you wish to simulate the 4-bit logic processor (to familiarize yourself with the simulation workflow) – that testbench is provided and called 'testbench.sv'.

Writing the Testbench:
The testbench for the 8-bit logic processor is provided as **testbench-8.sv. Make sure that when you switch from the 4-bit to the 8-bit processor, you also change the testbench name, as in the section below.**

Adding testbench file to your project:
In the ***File*** menu, select ***Add Sources***. Select ***Add or create simulation sources***. Choose a simulation set or make a new one, adding the testbench file you created (if you created a new file, select ***SystemVerilog*** for the file type). Press ***OK***. Press ***Finish***.

If you now have multiple simulation sources folders, you can right click and make the folder active to perform different tests as shown in Figure 18. Notice that if you have multiple testbenches the currently active one will be shown in bold. You can select different test-benches by right clicking on the desired testbench file and selecting “Set as Top”.

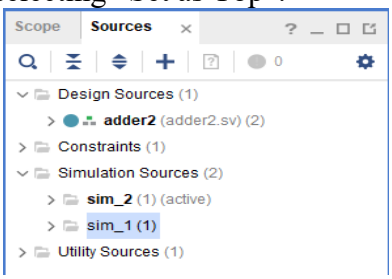


Figure 18 - Simulation Sources

Finally, the default Vivado settings run the simulation for only 100ns, which is often insufficient to see the full behavior of a complex circuit. Instead, we recommend that you use the $finish task at the end of a testbench so you will automatically run the simulation for the desired amount of time. You need to modify a project setting; go to: **Flow->Settings->Simulation Settings** to open the settings that control waveform simulation of your design. Click on the **Simulation** tab under the Project Settings-> Simulation page and change the value of **xsim.simulate.runtime** to **all**. This ensures your design will simulate until the **$finish** task is reached and not an arbitrary predefined time. Click Apply. You will only need to do this once per new project.

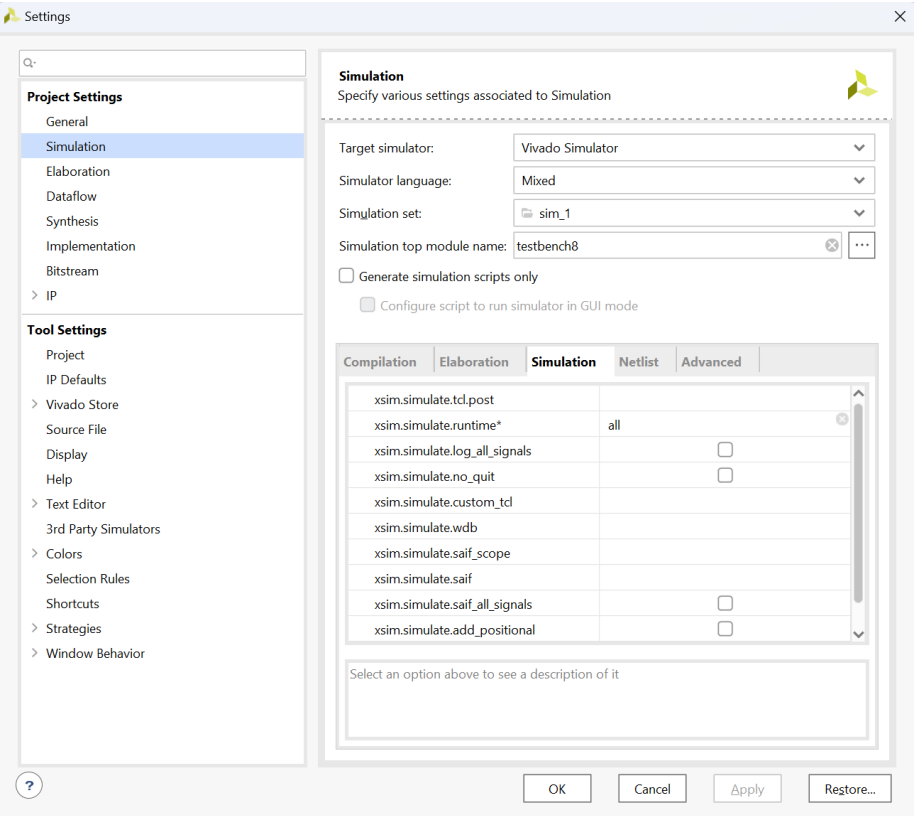


Figure 19 - Changing Simulation Settings

Note that your active testbench should be Now we're ready to run the testbench. Under Flow Navigator, click on ***SIMULATION > Run Simulation > Run Behavioral Simulation.*** Behavioral simulation is focused on validating the design's functionality. Note that if you finished the synthesis, you could do ***post-synthesis simulation***, which is geared towards assessing the timing behavior and performance of the design after synthesis.

The result will be shown on a new window on Vivado as shown in Figure 20. Click on the button to zoom full. And use buttons to zoom to a readable waveform.

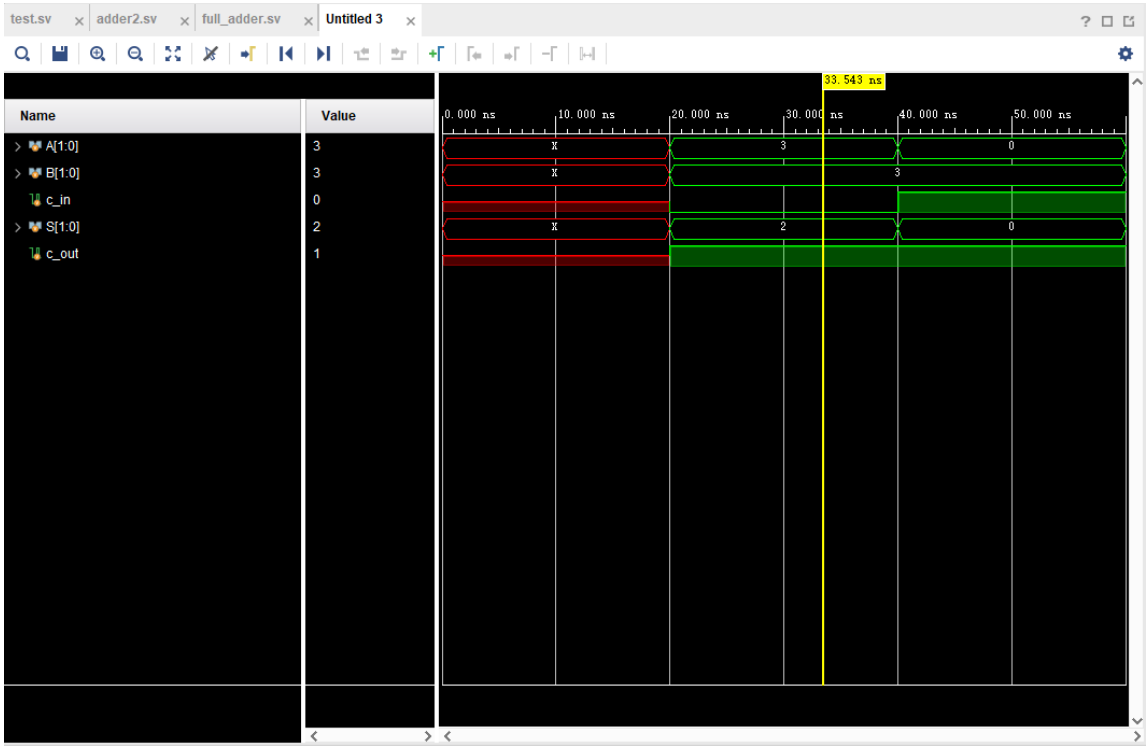


Figure 20 - Vsim Waveform for Adder

Note: By default, the buses are displayed in binary value. To change the display format, right click on the signal name and select ***Radix -> <format>***. For example, if you wish to see the binary value, you should select ***Radix -> Binary***.

To run a functional simulation of your design, click the ***Run-All*** button ( ) in the toolbar. If you wish to modify any portion of the waveform and re-simulate, you should first click the ***Restart*** button ( ) to erase the existing outputs.

Note that the red lines indicate that signals are unknown values in the beginning. Those are expected because there were no inputs at the beginning 10 ns.

Monitoring Internal Signals:

In Vivado, we also can monitor internal signals in the hierarchical design in addition to the ones specified in the testbench. To do this, navigate the design hierarchy on the left, select the signal to be observed, and then right click and choose ***Add Wave Window***. The selected signal will be added

into the simulation waveform window. Figure 21 shows an example of adding the state variable in the controller. You can also add waves by directly dragging them into the simulation window.

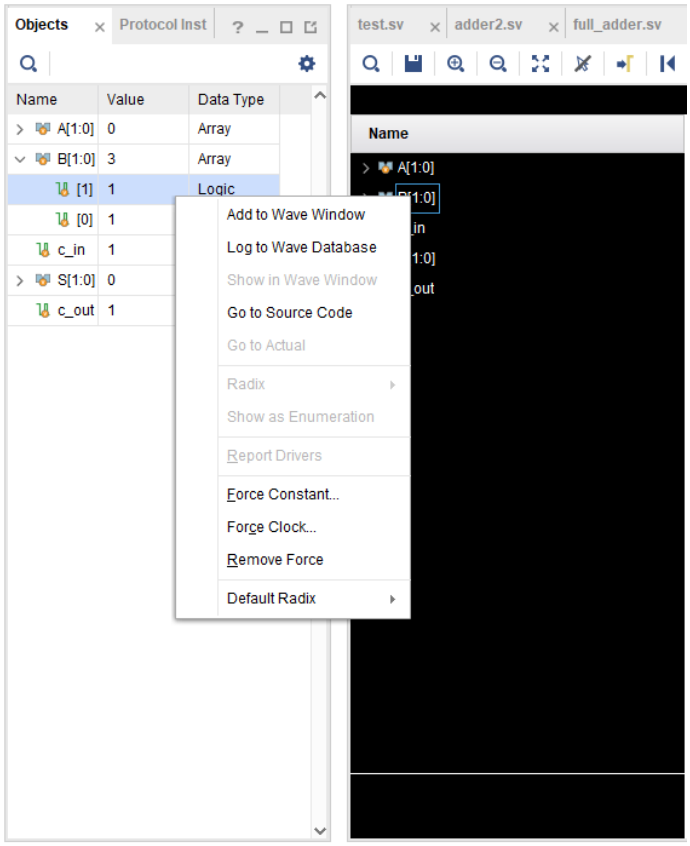


Figure 21 – Monitoring Additional Signals

After all signals of your choice are added, go to the command line below and type in:

```
restart
```

This will reset the waveform window such that the testbench starts from the beginning again.

```
run 100ns
```

This will run the testbench for 100ns. Alternatively, you may use the run command without any time parameter to simulate to $finish. The waveforms of the selected internal signals should be available for view now.

Troubleshooting:
If a certain part of your design is not showing up in the design hierarchy window, it's usually because the design file fails to compile successfully in Vivado. Look into the error messages in the *Transcript* to find out what went wrong.