



ECE4700J Computer Architecture

Summer 2022

Lab #1 Getting Start with Verilog Design Using Xilinx Vivado

Due: 11:59pm May 27th, 2022 (Beijing Time)

Logistics

- This lab is an individual exercise.
- All the design code should be in SystemVerilog or Verilog, we strongly encourage you to use SystemVerilog to be prepared for later assignments.
- This lab must be checked off by a TA before end of lab on Friday, May 27th, 2022 (Beijing Time). It is highly recommended you complete the lab before the following week's lab release.
- All code and reports (if available) MUST be submitted to the assignment of Canvas.
- Internet usage is allowed and encouraged.
- No late submission is allowed for this lab.

Overview

In this lab, you will install Vivado and setup the environment for Verilog programming, and then do some simple Verilog designs.

- Install the Xilinx Vivado ML Edition 2021.2.
- Set up the environment correctly.
- Be able to implement some simple Verilog/SystemVerilog designs and synthesize/simulate it using Vivado.

Xilinx Vivado Introduction and Tutorial

I. Introduction

Xilinx Vivado ML Edition - 2021.2 is a stable version of Xilinx software. In this semester, it's mainly used for Verilog programming, synthesizing and simulation.

II. Installation

1. Download & Sign Up

Download *Xilinx Vivado ML Edition - 2021.2* from [Downloads \(xilinx.com\)](https://www.xilinx.com/downloads). You need to first [sign up](#) at xilinx.com.



Version

[2022.1](#)[2021.2](#)[2021.1](#)[2020.3](#)[Vivado Archive](#)[ISE Archive](#)[CAE Vendor Libraries](#)[Archive](#)

Vivado ML Edition - 2021.2 Full Product Installation

Important Information

Vivado ML 2021.2 is now available for download:

- New device support for Artix® UltraScale+™: XCAU20P and XCAU25P
- Improved Intelligent Design Runs for push-button timing closure
- New example designs available in Vivado®
- Ease of use enhancements for HLS flows

We **strongly recommend** to use the web installers as it reduces download time and saves significant disk space.

Please see [Installer Information](#) for details.

Note:

- Download verification is only supported with Google Chrome and Microsoft Edge web browsers.
- Beginning this release we will be offering only 2 Editions for Vivado ML. Please go to [product page](#) for more details.
- Vivado ML 2021.1 and later versions require upgrading your license server tools to the Flex 11.17.2.0 versions.

[Download Includes](#)[Download Type](#)[Last Updated](#)[Answers](#)[Documentation](#)[Support Forums](#)[Vivado ML Edition](#)[Full Product Installation](#)[Oct 27, 2021](#)[2021.x - Vivado Known Issues](#)[Release Notes](#)
[OS Support Update](#)
[What's New in Vivado](#)[Installation and Licensing](#)

Based on your OS type and network, you can choose to download the “Self-Extracting Web Installer” or “All OS Unified Installer Single-File Download (SFD)”.

📄 [Xilinx Unified Installer 2021.2: Windows Self Extracting Web Installer \(EXE - 212.41 MB\)](#)

MD5 SUM Value : 76b60fc6a74338066f4e4dd0a855ec93

Download Verification ⓘ

[Digests](#)[Signature](#)[Public Key](#)

📄 [Xilinx Unified Installer 2021.2: Linux Self Extracting Web Installer \(BIN - 272.8 MB\)](#)

MD5 SUM Value : d4fe2978f735e4353f6ccff3405b488b

Download Verification ⓘ

[Digests](#)[Signature](#)[Public Key](#)

📄 [Xilinx Unified Installer 2021.2 SFD \(TAR/GZIP - 71.9 GB\)](#)

MD5 SUM Value : c6f91186f332528a7b74a6a12a759fb6

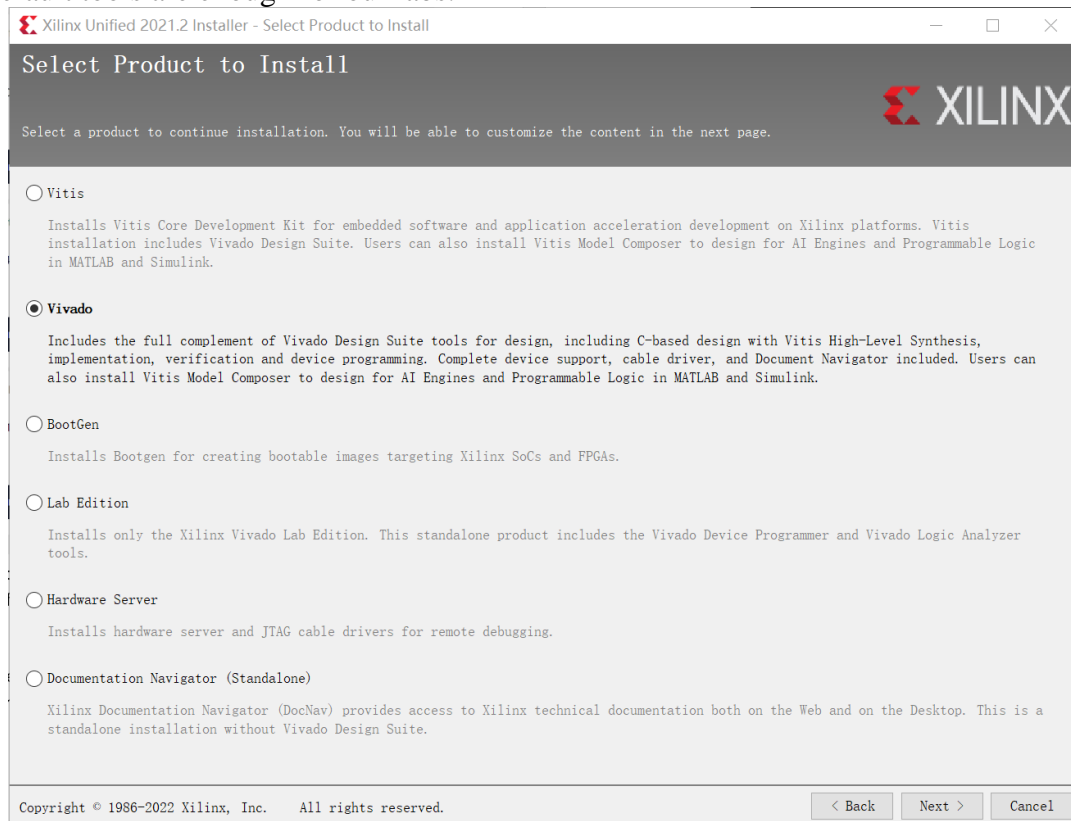
Download Verification ⓘ

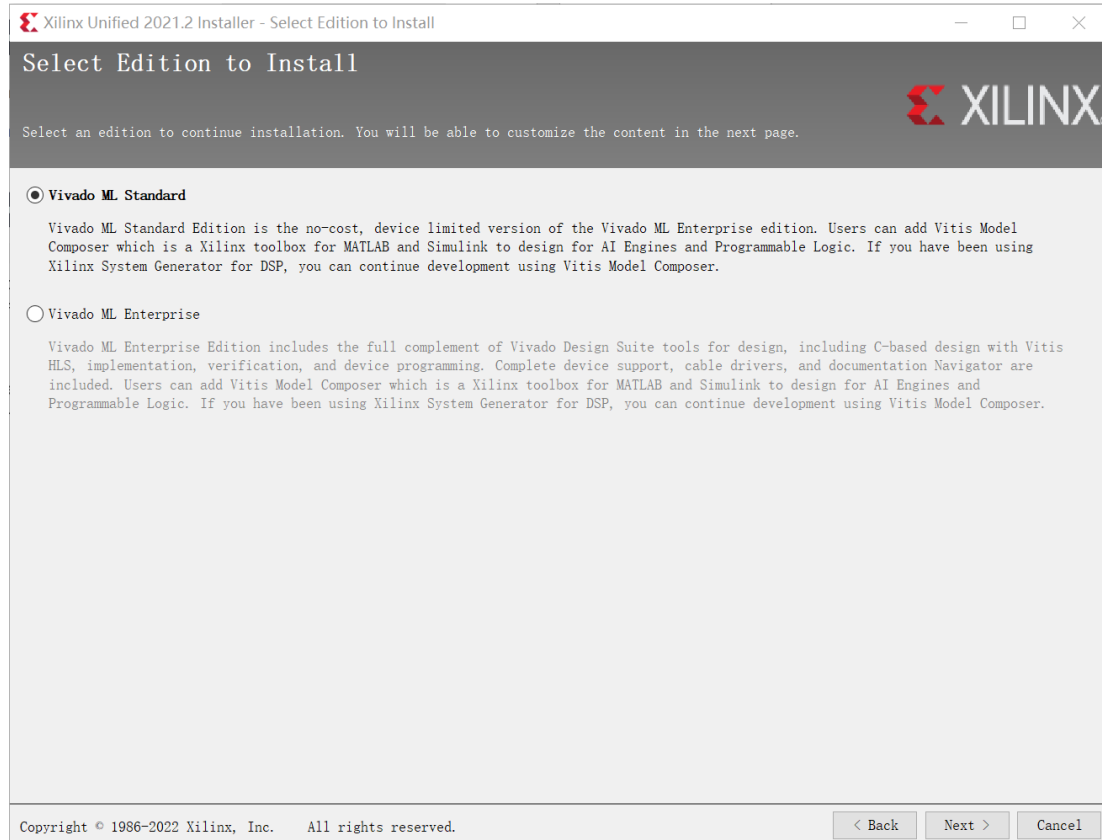
[Digests](#)[Signature](#)[Public Key](#)

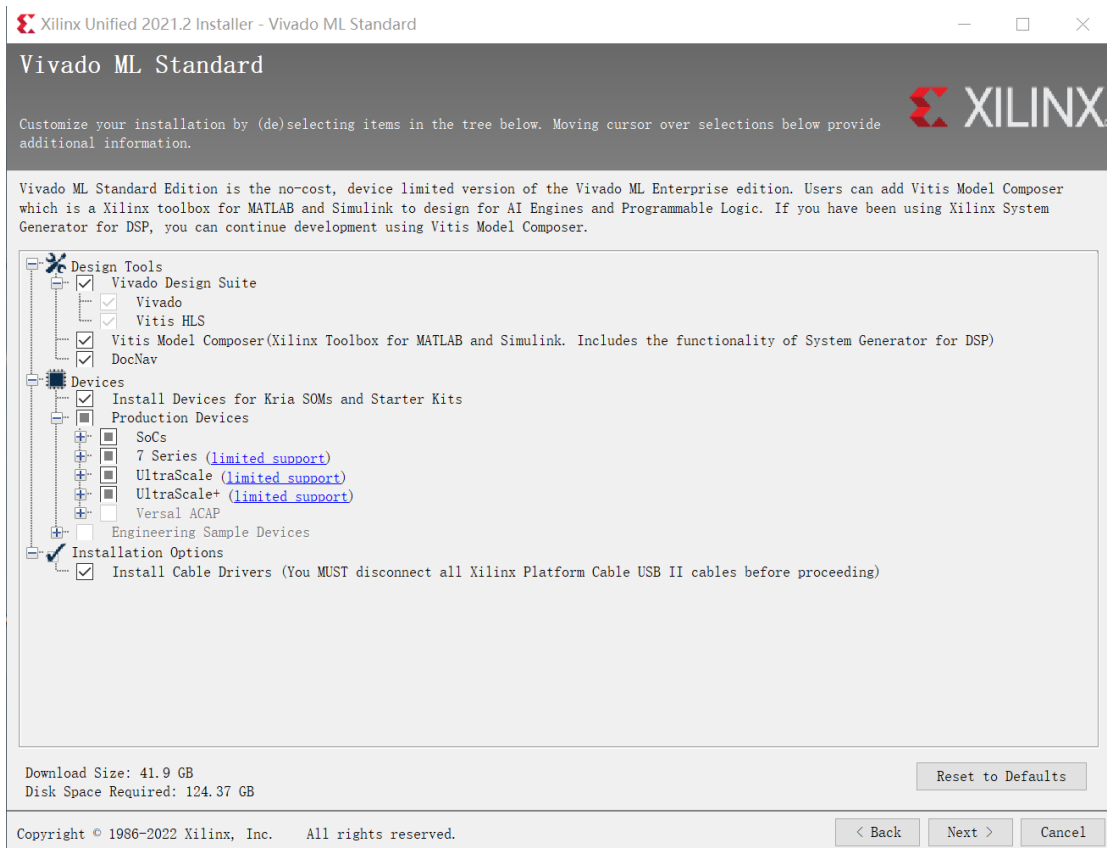


The former installer will download the main program by itself and the latter one has contained all necessary files and you needn't download anything more.
Here, we use the former one to continue demo.

2. Execute the installer and you need to sign in. Choose **Vivado ML Standard** which is free. Default tools are enough for our labs.



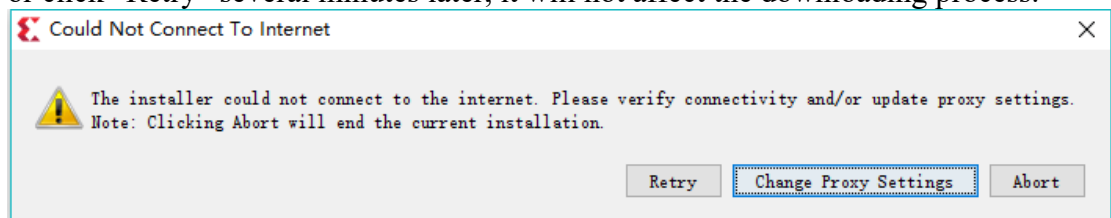




And then you can specify the installation directory and start installing.

Tips:

- When downloading, there may be warnings as the figure below, If the installer is still running and the progress bar is progressing, please just ignore it or click “Retry” several minutes later, it will not affect the downloading process.



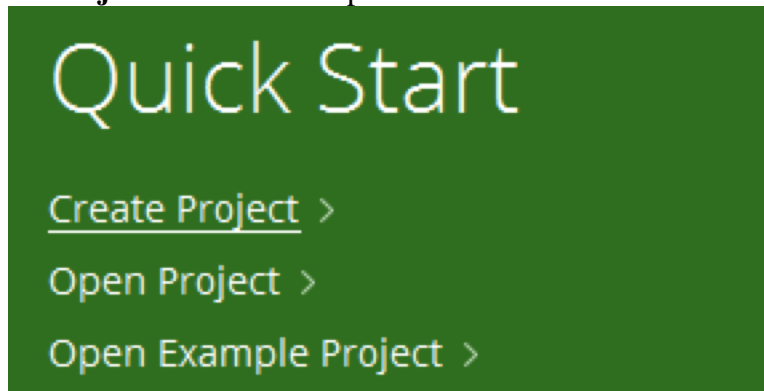
- If you find the downloading speed to be slow, connect to SJTU VPN may help.
- Start early.** The installation of Vivado does take a while.

III. Create Project and Do Verilog Programming

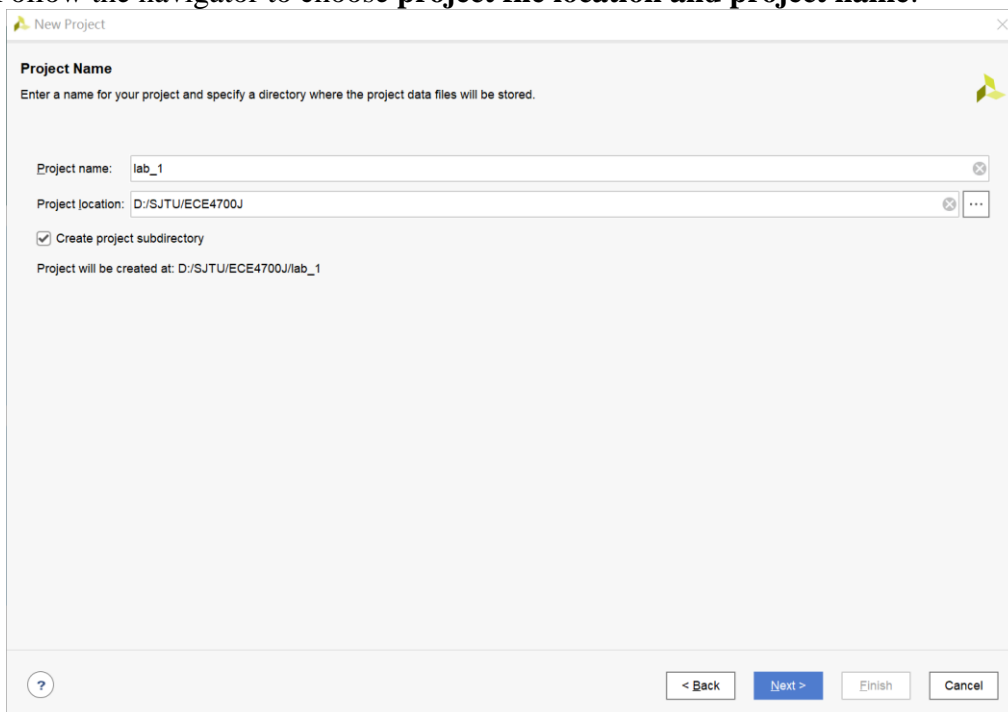
1. Open **Vivado** to open the software.



2. Click **Create Project** in the left side panel.

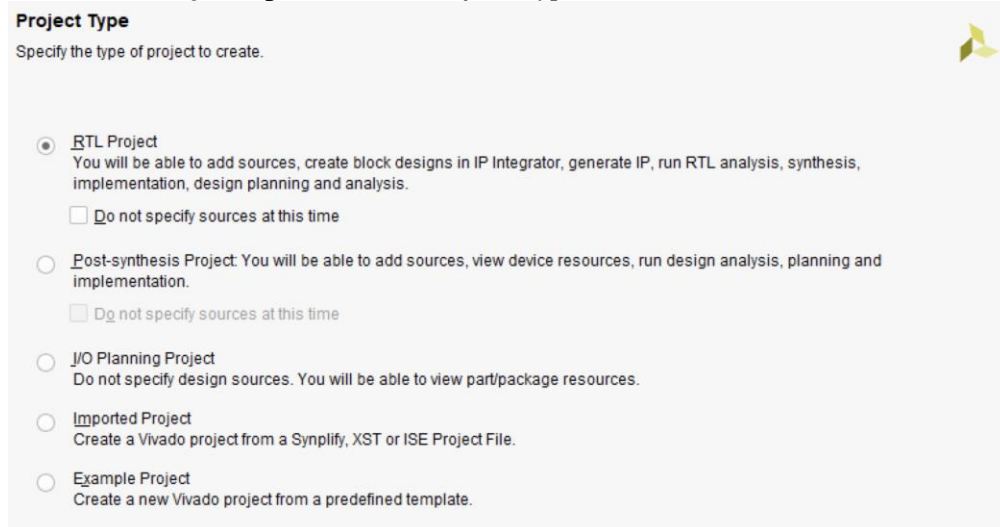


3. Follow the navigator to choose **project file location and project name**:



Project Type:

Select **RTL Project** option in the *Project Type* form, and click **Next**.

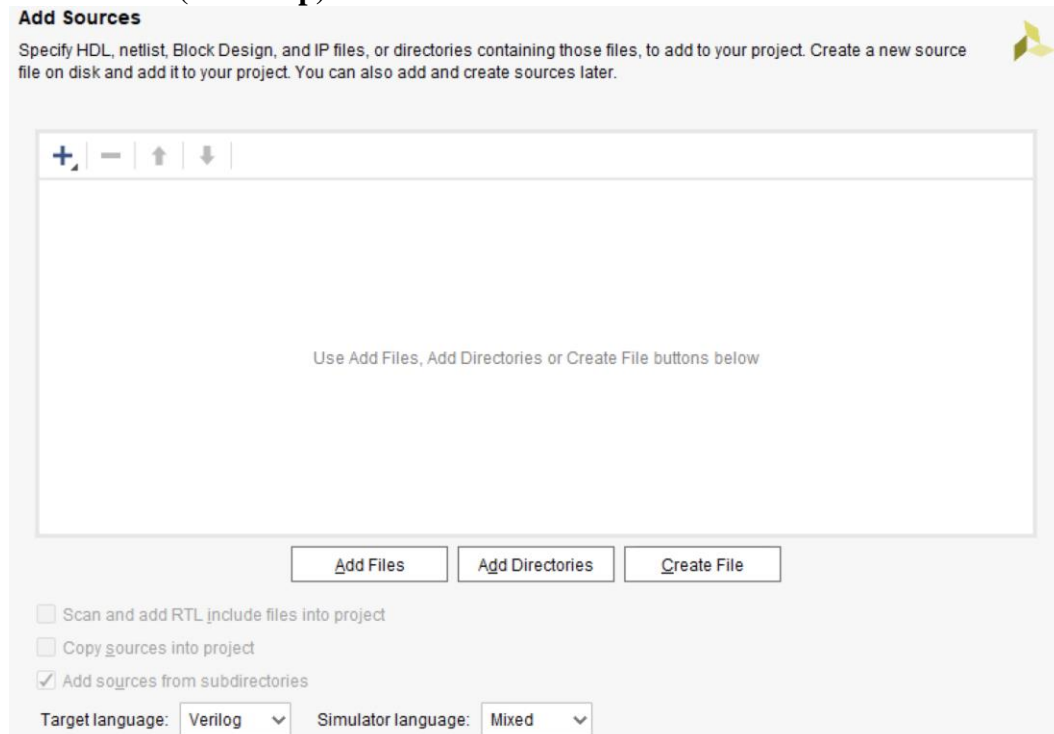


Project Type
Specify the type of project to create.

- ☒ **RTL Project**
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.
☐ Do not specify sources at this time
- ☐ **Post-synthesis Project**: You will be able to add sources, view device resources, run design analysis, planning and implementation.
☐ Do not specify sources at this time
- ☐ **I/O Planning Project**
Do not specify design sources. You will be able to view part/package resources.
- ☐ **Imported Project**
Create a Vivado project from a Synplify, XST or ISE Project File.
- ☐ **Example Project**
Create a new Vivado project from a predefined template.

you may check “Do not specify sources at this time” box if you want to add source file later.

Add Source (Can Skip):



Add Sources
Specify HDL, netlist, Block Design, and IP files, or directories containing those files, to add to your project. Create a new source file on disk and add it to your project. You can also add and create sources later.

Use Add Files, Add Directories or Create File buttons below

☐ Scan and add RTL include files into project
☐ Copy sources into project
☒ Add sources from subdirectories

Target language: Verilog
 Simulator language: Mixed



Optional:

Using the drop-down buttons, select **Verilog** as the *Target Language* and *Simulator Language* in the *Add Sources* form.

Add Constrain (Can Skip):

Add Constraints (optional)
Specify or create constraint files for physical and timing constraints.

Use Add Files or Create File buttons below

☐ Copy constraints files into project

Choose Default Part:

Different parts in Vivado means different FPGA target boards. In ECE4700J course, since we only need to synthesize and simulate the design (we don't need to do on-board experiments), you can feel free to select your favorite part and it doesn't matter. However, you must ensure that the resources (i.e., the LUTs, DSPs, RAMs, hardware resources) on this part is enough for your design. We recommend you to choose big boards as the part. For example, we recommend you to use the part **xczu7eg-ffvf1517-2-i** (if you have installed the Ultrascale+ series devices in step II.2)

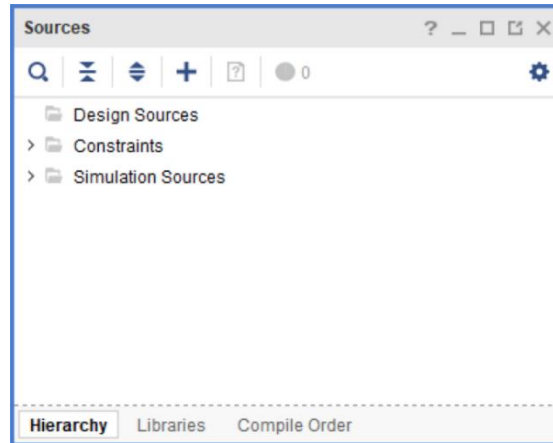
Finish Creating Project:

When you have finished the above steps, a new project is created.

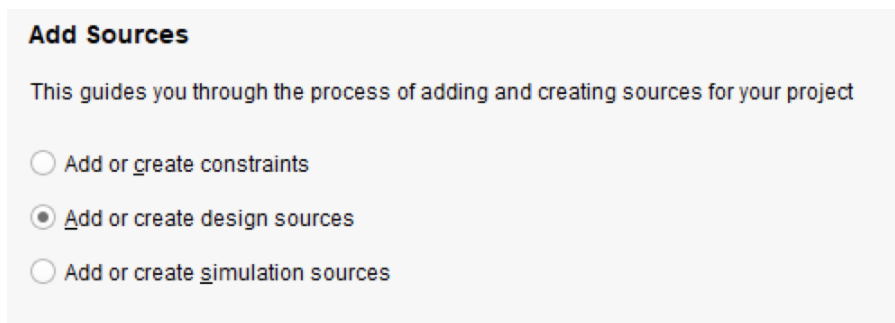
IV. Design and Behavioral Simulation

1. Create Design

Find source windows and click “+” to add source.

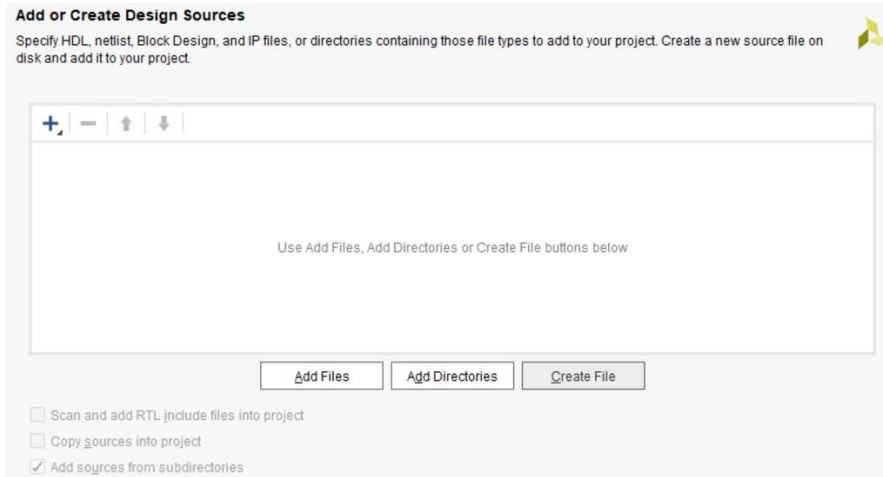


Choose “Add or create design sources”.

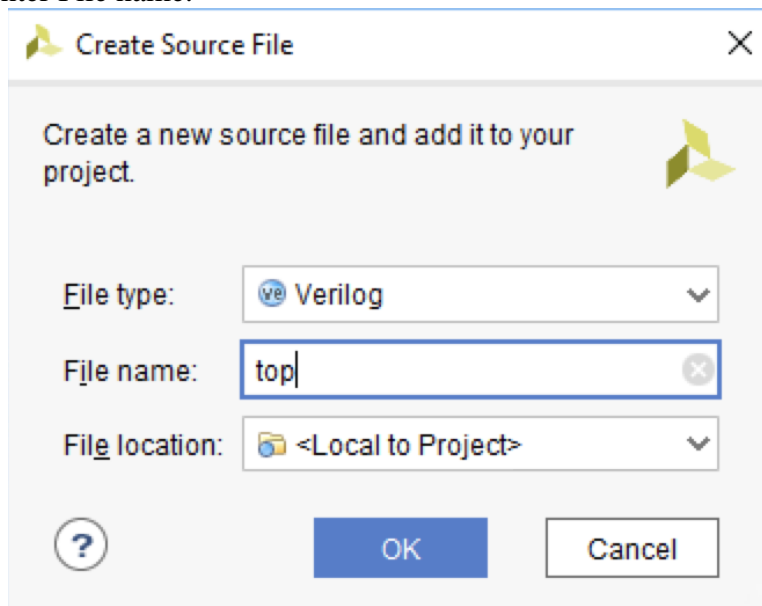


If you have finished writing the design files in other IDEs (like VSCode), you can directly add these files by clicking “Add Files”.

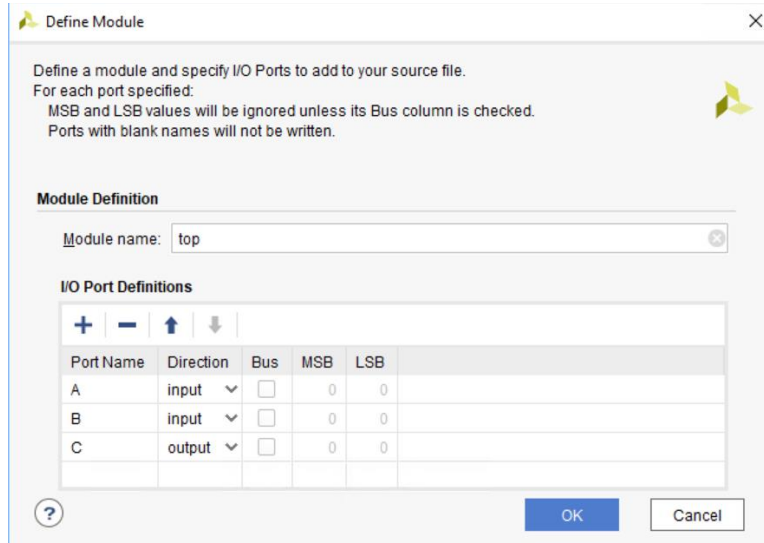
If you want to create design files, click “Create File”.



Choose File type to be Verilog/SystemVerilog (ECE4700J will mainly use SystemVerilog), enter File name.



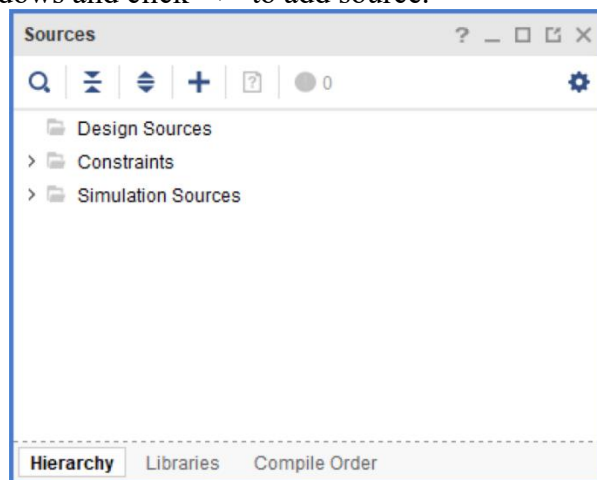
Click “OK” and “Finish”, then a “Define Module” window will pop up. You can skip this step, or you can also name your module ports here.



Then you can code in your Verilog file to implement your design.

2. Create Simulation

Find source windows and click “+” to add source.



Choose “Add or create design sources”.

Add Sources

This guides you through the process of adding and creating sources for your project

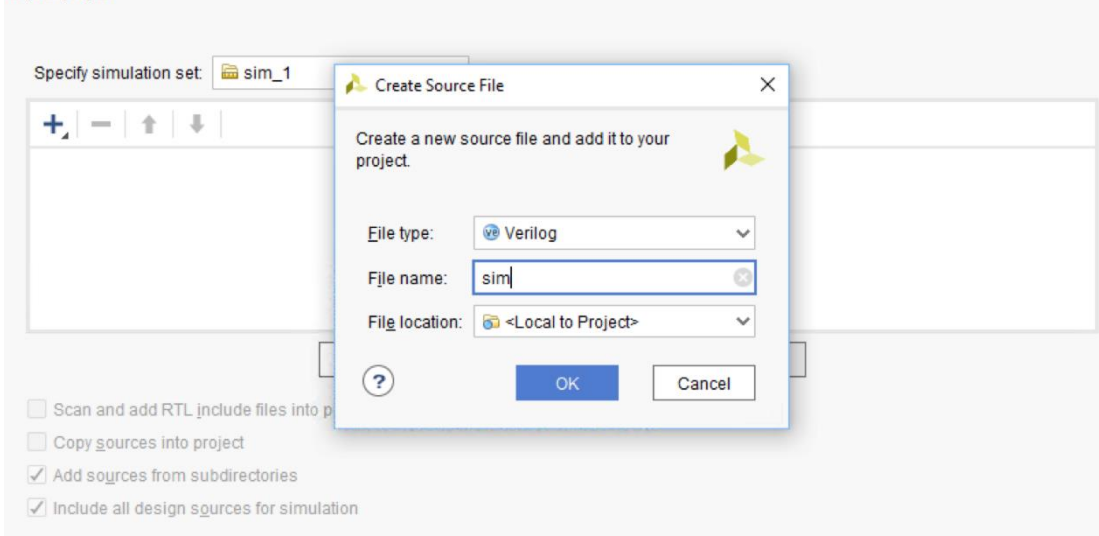
- ☐ Add or create constraints
- ☐ Add or create design sources
- ☒ Add or create simulation sources

If you have finished writing the simulation files (testbenches) in other IDEs (like VSCode), you can directly add these files by clicking Add Files.

If you want to create testbenches, click “Create File”, Name file and choose location.

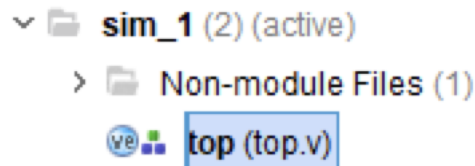
Add or Create Simulation Sources

Specify simulation specific HDL files, or directories containing HDL files, to add to your project. Create a new source file on disk and add it to your project.



Click OK and Finish.

Open the simulation Verilog file in Source Hierarchy to write your simulation.

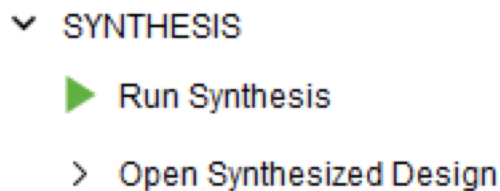


Choose corresponding simulation file and run simulation by clicking “Run Simulation” – “Run Behavioral Simulation” in the left **Flow Navigator** bar and get the **behavioral simulation** (i.e., the pre-synthesis simulation) result.

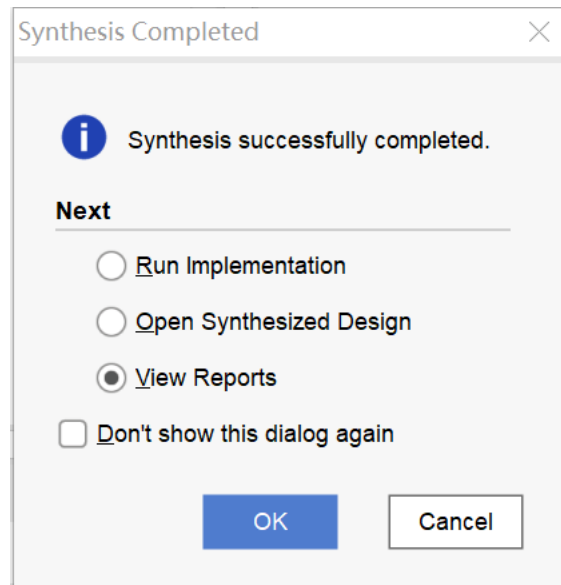
V. Synthesis and Post-Synthesis Simulation

1. Synthesis the design

Run Synthesis in Flow Navigator with default setting.



After the synthesis process is finished, you may open the synthesis design or synthesis reports.



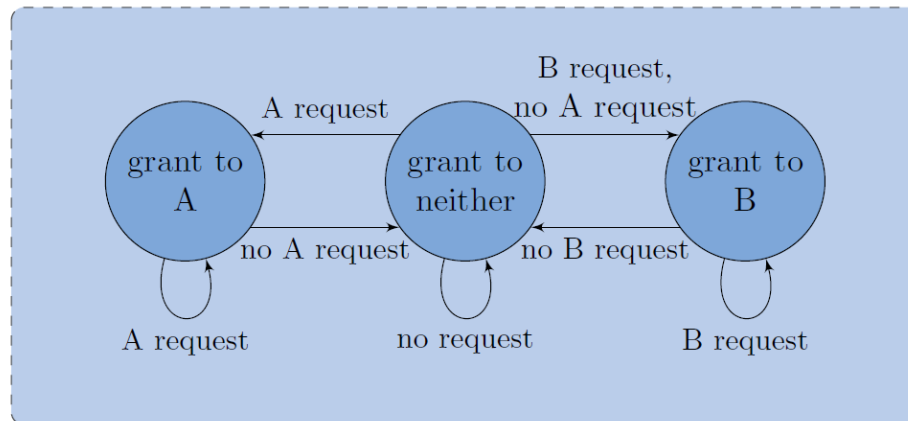
2. Do the post-synthesis simulation

Run post-synthesis simulation by clicking “Run Simulation” – “Run Post-Synthesis Timing/Functional Simulation” in the left Flow Navigator bar and get the post-synthesis simulation result.

Assignments

I. Design of a Simple Arbiter (Mandatory)

You will now write and debug a design from scratch, though you probably want to use designs we've shown you as templates.



This is a Moore state machine for an arbiter, which conceptually would be connected to two requesters, A and B, at some higher level, and should provide signals to each indicating whether control has been granted to it.

Begin by examining the state machine in this figure. What are the inputs and the outputs? What state needs to be stored, and given this, what registers do you need? What does the combinational logic look like?

The Verilog you write to implement the state machine above should be put into a file called **arbiter.v**. You will also need a testbench, **arbiter_test.v**, to test the module. You should try to write a testbench that produces the correct output for the module in different situations to test against.

We will provide you with starter files of **arbiter.v** and **arbiter_test.v**, after you finished your design, you should synthesize it and do post-synthesis timing simulation. Your design needs to pass the timing of synthesis (i.e., no timing violation during simulation), and shows the correct output.

During the live demo, you should show TA with the waveform of your post-synthesis timing simulation and describe why it can prove that your design is correct. After the live demo, you also need to submit your **arbiter.v** and **arbiter_test.v** (contrasted into a zip/tar file) on canvas before the due date.



II. Priority Selectors Design (Optional)

1. Introduction

In this optional assignment, you will be designing a number of different priority selectors. **You will be given an up to 10% bonus to the total score of Lab 1** if you are able to correctly implement the optional assignment. This will be counted towards your final grade.

A priority selector, or priority arbiter, has n pairs of request and grant lines. As the names imply, the selector chooses one of the asserted request lines and asserts its corresponding grant line. In the most general case, the selector can assert k grant lines, where $k \leq n$, though for this project, $k = 1$. Priority selectors are heavily used in computer architecture, where we often have limited resources that need to be assigned optimally for best performance. The modules you write for this assignment will both remind you about the concepts of digital design you learned in VE 270 and begin to prepare you for the following labs and projects.

2. Assignment

In this assignment you will be asked to design and test the following devices:

- A 4-bit fixed priority selector, using behavioral SystemVerilog:
 - Using assign statements for the combinational logic.
 - Using always blocks for the combinational logic.
- A 4-bit and an 8-bit fixed priority selector using a hierarchy of 2-bit selectors.

a) Before you start

We provided a testbench (test.v) for you to test your design, and we provided code for two types of AND gates (in And.v). You should not modify them during your design.

In this assignment you **must** use SystemVerilog.

b) Part A and B: 4-bit Fixed Priority Selector

For this section, you will design two different 4-bit fixed priority selectors. Both are to be declared as:

```
module ps4(  
    input      [3:0] req,  
    input      en,  
    output logic [3:0] gnt  
);
```

The signal `req[3]` is the highest priority request, and priority goes down to `req[0]`, which is the lowest priority request. In all cases no more than one of the grant lines



should be asserted at any given time. If `en` is low, then no grant lines should be asserted. For example, if `en=1` and `req=4'b1111`, then `gnt=4'b1000`.

The design for this section should be put into a file named `P1a.v`. Using assign statements (and no always blocks) implement a 4-bit priority selector. You may want to use a K-map to find the logic equations needed for the grant lines, or you might be able to solve it ad hoc.

The `test.v` provided is a testbench written for this design. You should add `test.v` as your simulation file in Vivado and do simulations.

Once you have completed the above task, you need to do it again, but this time put the design in a file named `P1b.v`. This time you are to use if/else statements inside an always block.

c) Part C: Hierarchical Priority Selectors

Consider the Verilog you wrote in `P1a.v` and `P1b.v`. If you were going to make a 128-bit priority selector, your design would be quite long and unreadable. One way to avoid this problem is to build your module in a way that it can be combined with itself to make a larger version. For example, it is fairly easy to use three 2-bit and gates to create a single 4-bit and gate. Think through how you would go about building this. The figure below shows how this is done in the case of the and. How would you go about doing this with priority selectors? It's not easy. . .


```

module and2(
    input      [1:0] a,
    output logic x
);
    assign x=a[0] & a[1];
endmodule

module and4(
    input      [3:0] in,
    output logic out
);
    logic [1:0] tmp;
    and2 left(.a(in[1:0]),.x(tmp[0]));
    and2 right(.a(in[3:2]),.x(tmp[1]));
    and2 top(.a(tmp),.x(out));
endmodule

```

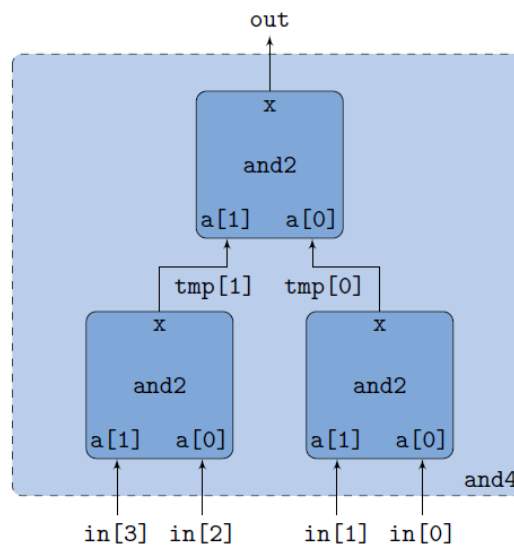


Figure 2: Hierarchical Module Example: a 4-bit and built with three 2-bit and modules.

Let's consider a 2-bit priority selector as a building block. In order to make a 4-bit device, you'd need three of 2-bit selectors in a tree structure, similar the and's above. The right might get the two lowest priority request and grant lines, while the left got the two highest priority request and grant lines. Then the left and right would ask the top to choose which of them got the grant. In our previous 4-bit module we had a way to be told if we could grant to anyone, the enable line (en). But we didn't have a way of saying "Hey, I have a request that would like a grant." We will need to add that functionality,



and you should call it req up for “requesting something from the device above me.” This signal should be asserted if either request is asserted no matter the value of the enable.

You will need two modules, one is the 2-bit priority selector and the other is the 4-bit. These modules should be declared as follows:

```
module ps2(  
    input      [1:0] req,  
    input      en,  
    output logic [1:0] gnt,  
    output logic req_up  
);  
  
module ps4(  
    input      [3:0] req,  
    input      en,  
    output logic [3:0] gnt,  
    output logic req_up  
);
```

To build the 2-bit selector, use either P1a.v or P1b.v as a template, and don't forget to include the req up signal. Then you need to build the 4-bit device. Use Figure. 2 as an example of the concept of the tree structure, and in particular, be sure you understand the Verilog before you proceed.

Your design for this section should be written into a file named P1c.v. In addition, you will need to modify the testbench to use this new module and its additional output. Once you have this version working, build a ps8 module using ps4 and ps2 modules, and save it in the same file. The module declarations should follow from ps4, above. For full credit, your design should not contain any additional “glue” logic. It should only contain the minimum number of ps modules and wires connecting them. To test this new module, write a new testbench and save it as testC.v. You will need to turn it in.

3. Submission

After you are confident in your solution, make sure you have the following files in your directory:

- a) P1a.v
- b) P1b.v
- c) P1c.v
- d) testC.v

Then you can compress your files, and submit it on canvas. This optional assignment doesn't need live demos.



References:

1. Vivado Design Suite User Guide: Design Flows Overview (UG892) <https://docs.xilinx.com/r/en-US/ug892-vivado-design-flows-overview/Vivado-System-Level-Design-Flows>
2. Vivado Design Suite User Guide: Synthesis (UG901) <https://docs.xilinx.com/v/u/en-US/ug901-vivado-synthesis>
3. Umich EECS470 WN 2021 Lab1
4. Umich EECS470 WN 2021 Project1

Acknowledgement:

- Jon Beaumont (University of Michigan)
- VE270 TA Group
- Xilinx



Deliverables:

- Live Demo of Mandatory Assignment:
 - Show TA with the waveform of your post-synthesis timing simulation and describe why it can prove that your design is correct.
- Individual Deliverables:
 - Submit design files of the Mandatory Assignment: arbiter.v and arbiter_test.v (contrasted into a zip/tar file) via canvas assignment.
 - If you choose to finish the optional assignment, submit your design files via canvas by following the instruction in “Submission” section of the optional assignment.

Grading Policy

Live Demo – 70%

Canvas Files Submission and Correctness – 30%