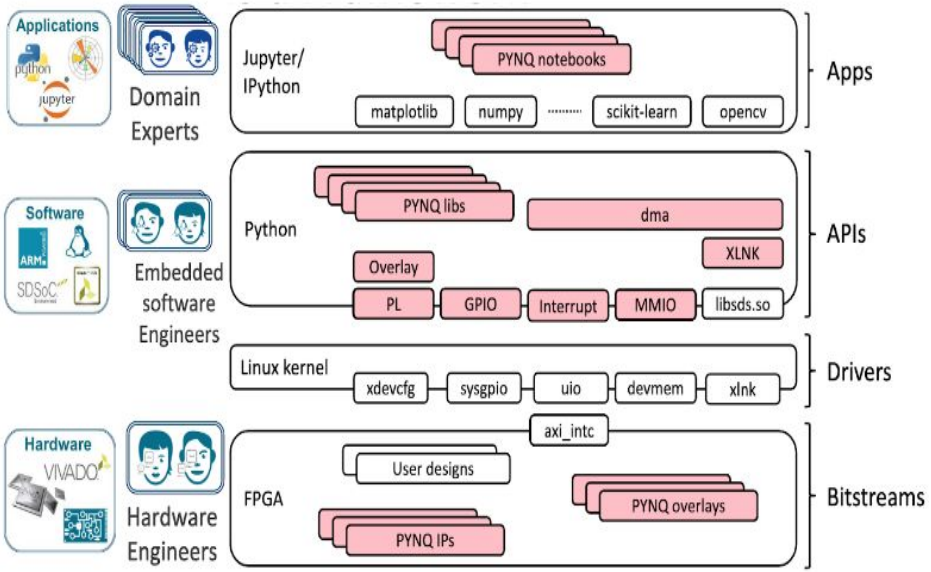
Xilinx & Vivado IDE

Speaker: Jia-Ming Lin

Outlines

- Recap. PYNQ Framework
- FPGA Architecture
- FPGA Design Process
- Design Optimization
- Lab1:
  - Creating hardware design by using Vivado HLS
  - Generating Bitstream
  - Validating on FPGA.

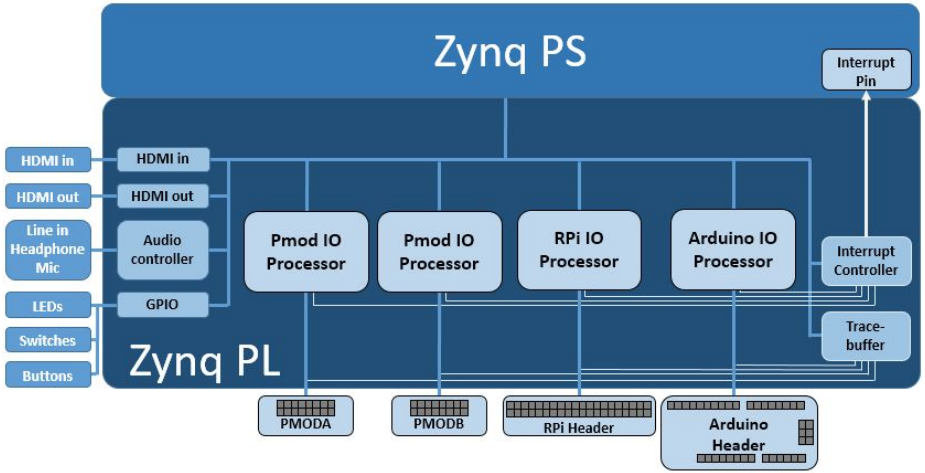
Recap. PYNQ Framework



- **Interactive computing environment** based on Jupyter Notebook
- Using Python to invoke **hardware libraries and overlay** in Programmable Logic
- Through PYNQ, we can easier to create high performance applications with
  - Parallel hardware execution
  - Hardware acceleration algorithm
  - High frame-rate video processing.

What is an Overlay?

- Hardware library, consisted of one or several IPs
- Extends user application from Processing System(CPU) to Processing Logic(FPGA)
  - Speed-up or customize the hardware platform for a particular application.
- Example: PYNQ base overlay



Case Study: Video input/output

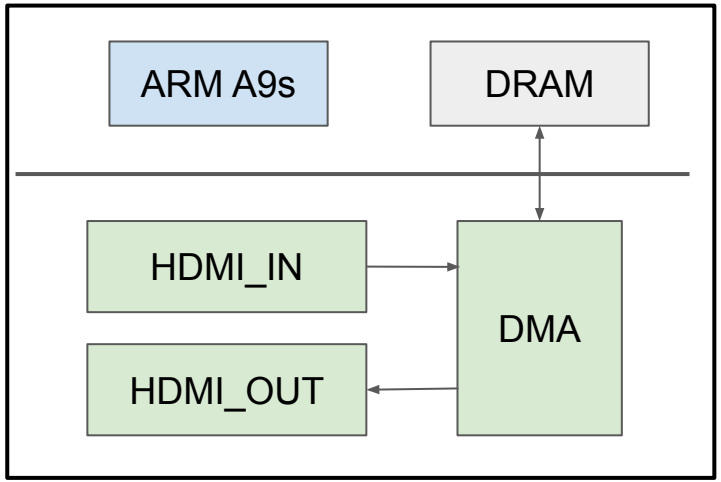
- Two Methods
  a. OpenCV on ARM A9s
  b. HDMI_IN and HDMI_OUT on Processing Logic
- Which one is better? In terms of throughput(FPS), energy consumption(Watt)

Method (a)

- USB Webcam on USB port of Zynq PS(CPU)
- Capture image and stored to DRAM
- OpenCV package is used

Source code reference:
https://gist.github.com/cathalmccabe/b0ab8917f748840f0d3959f7eabf0f82

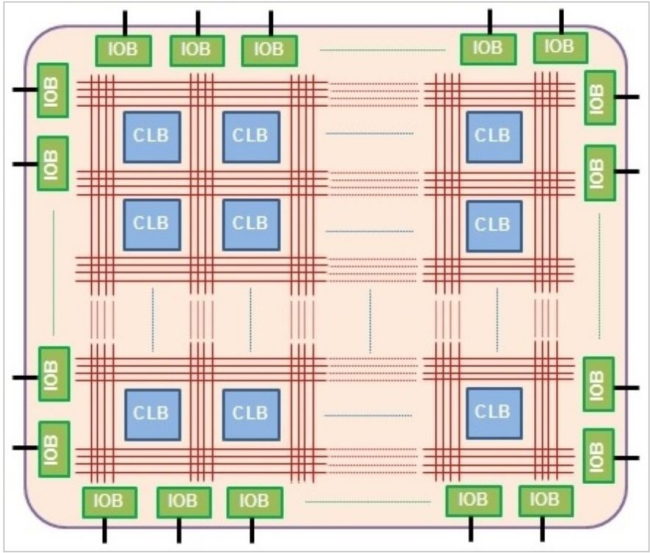
Method (b)



FPGA Architecture

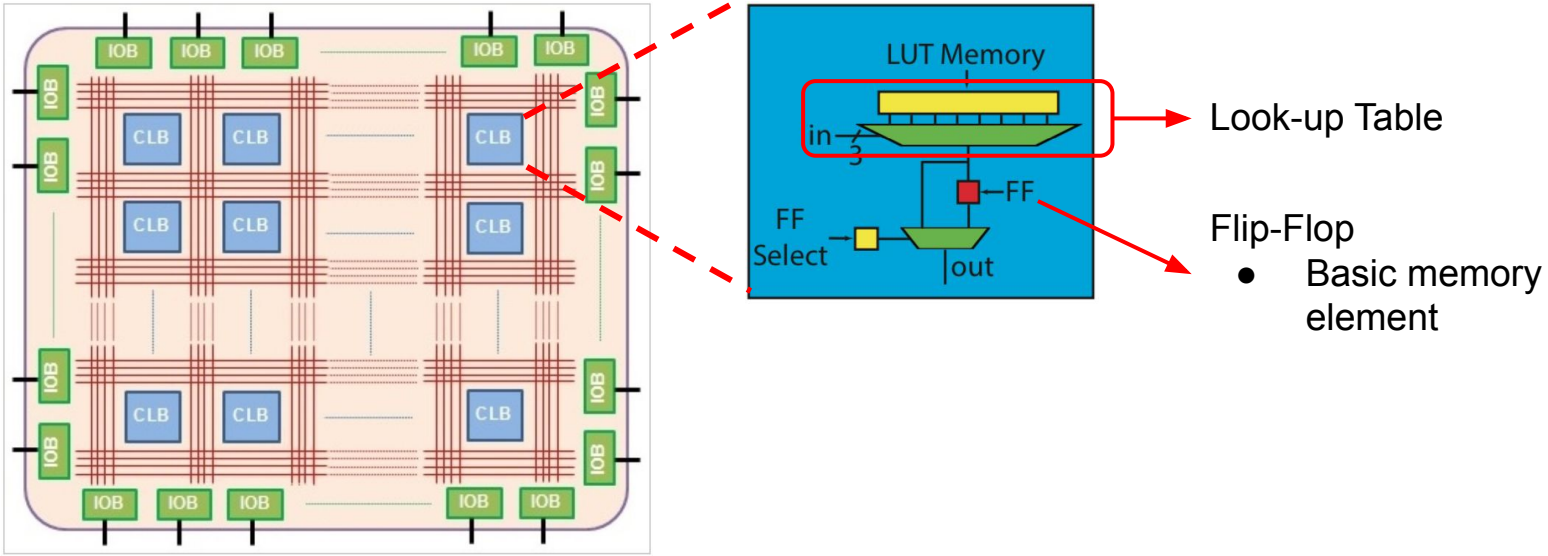
Field Programmable Gate Arrays(FPGAs)

- Recap, **Programmable Logic Device**
  - Contains an array of AND gate & another array of OR gate
  - Users can configure these devices in order to implement the Boolean functions
  - E.x. some form of sum-of-product.
- Idea behind **FPGA**,
  - Programmable/configurable logic blocks,
  - Programmable interconnections between logic blocks
    - Configure how the logic blocks are connected
- Two dominant FPGA makers
  - Xilinx and Altera



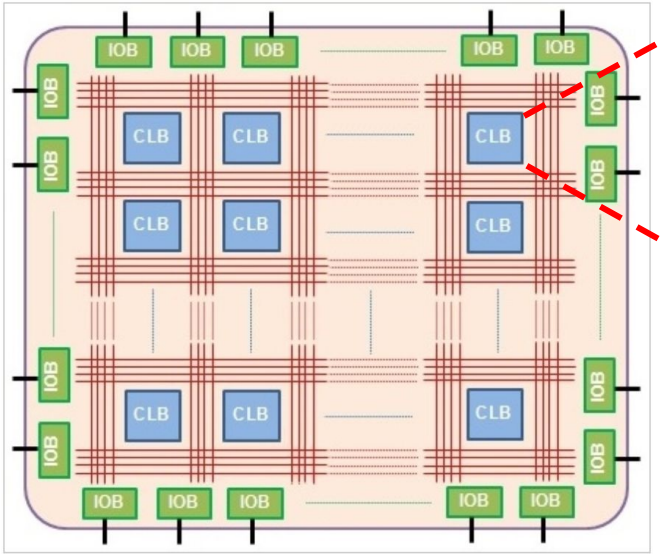
Configurable Logic Block(CLB) or Slice

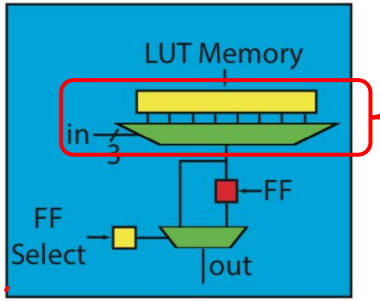
- Based on Look-up Tables(LUTs), Flip-Flop and other specialized functions(e.g. Full Adder)

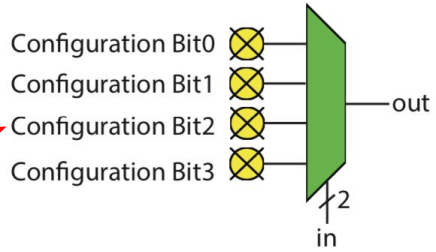


Configurable Logic Block(CLB) or Slice

- Configuring the FPGA







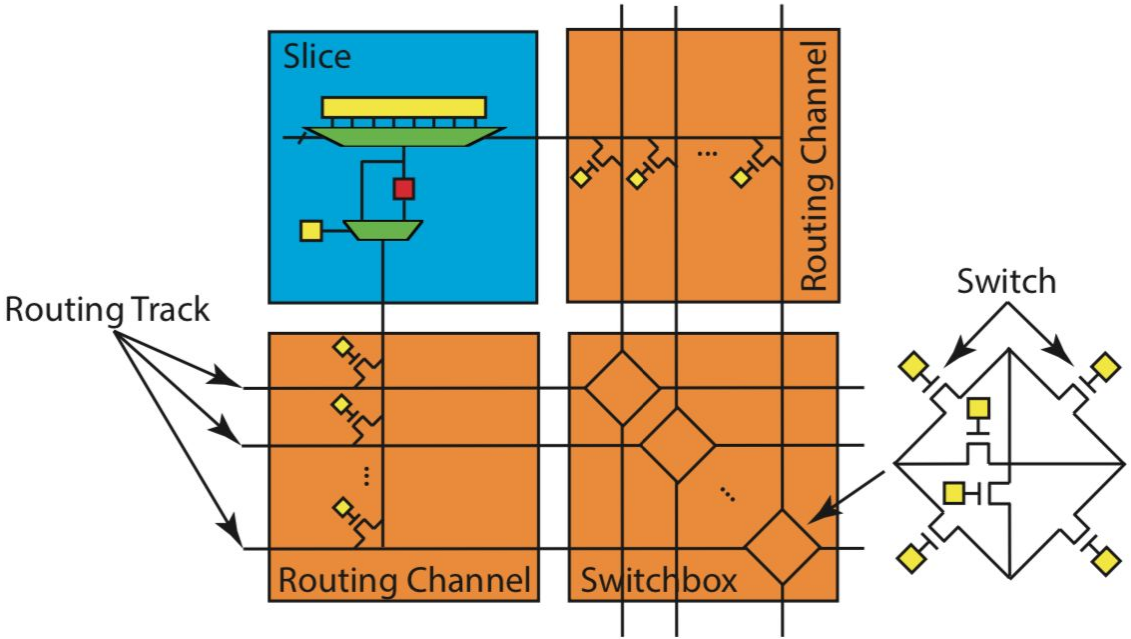
in[1]	in[0]	out
0	0	0
0	1	0
1	0	0
1	1	1

out = in[1] & in[0]

- LUTs are configured using **Bitstream**
- Example, truth table for a 2 input AND gate

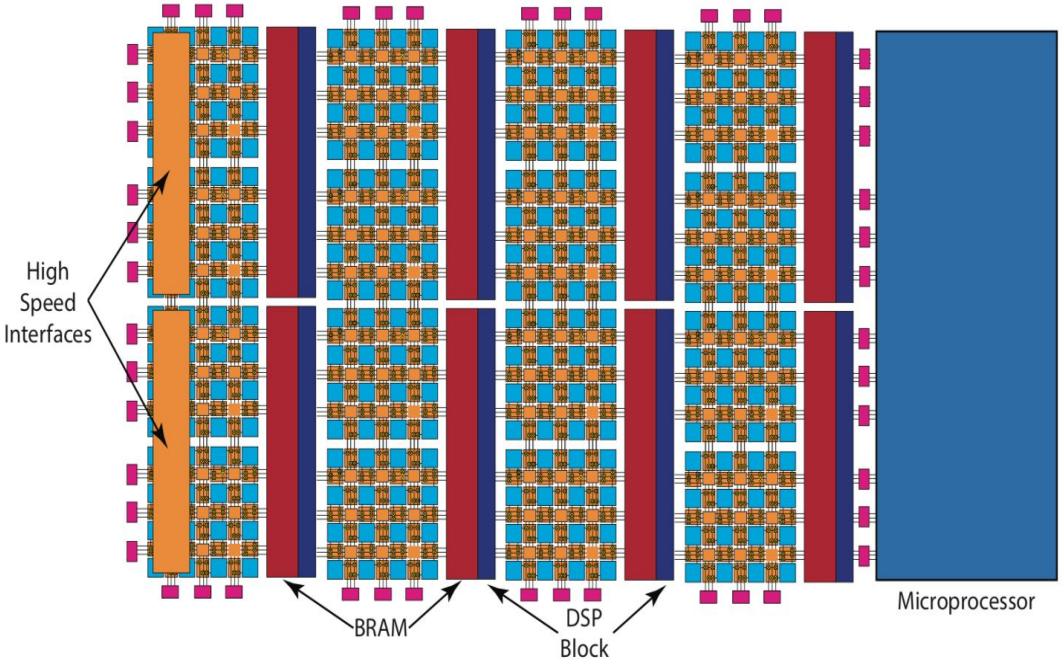
Programmable Interconnections

- Flexible network connections of CLBs(or Slices)



- At each interconnect, there is a **switch transistor** which is default OFF
- Each switch is controlled by a 1-bit register
- Configuring routing is simply put 0 or 1 into the register

Modern FPGAs



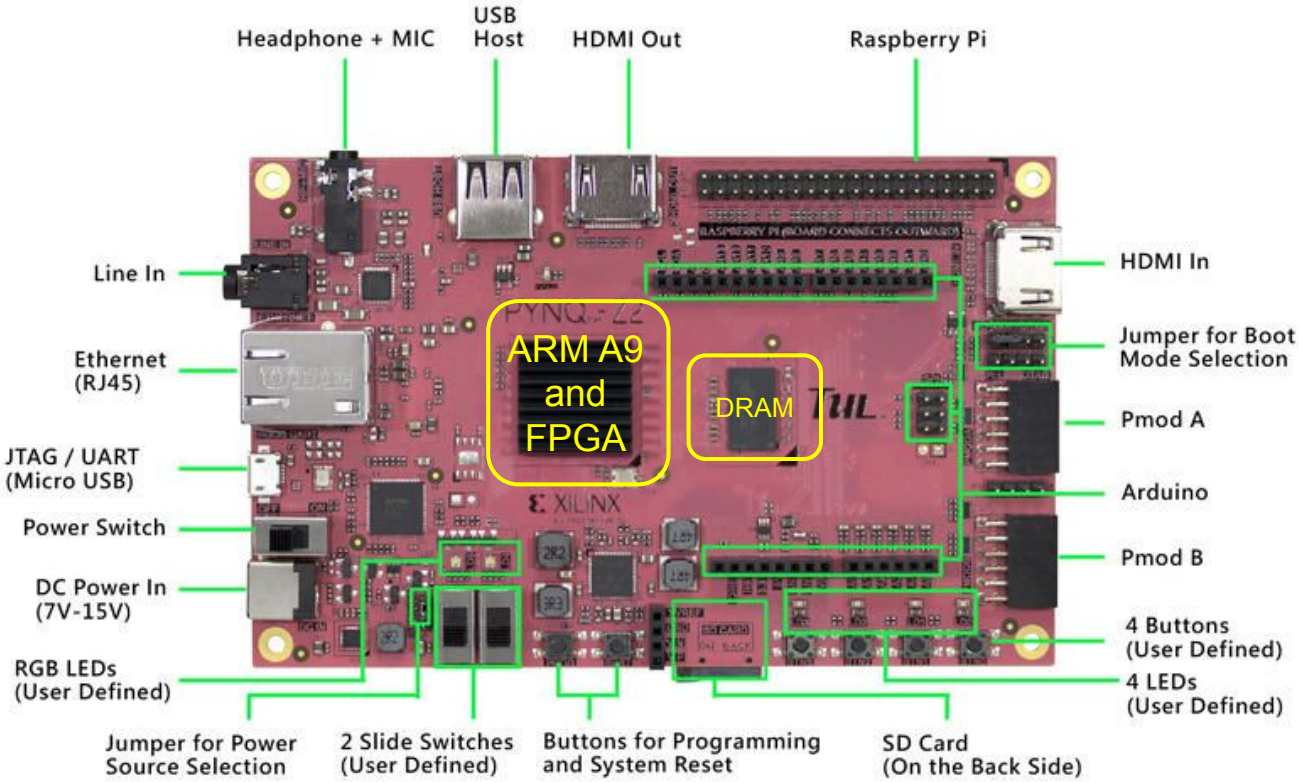
- As the number of transistors on FPGA increasing...
- More “hard” resources
- **DSP48**
  - Mul, Add, MAC
  - Higher frequence
  - Not flexible as CLB
- **Microprocessors**
  - ARM or x86
- **Block RAM (BRAM)**
  - Transfer data
  - Store large data on-chip

Comparison of three different on- and off-chip memory storage options

	External Memory	BRAM	FFs
count	1-4	thousands	millions
size	GBytes	KBytes	Bits
total size	GBytes	MBytes	100s of KBytes
width	8-64	1-16	1
total bandwidth	GBytes/sec	TBytes/sec	100s of TBytes/sec

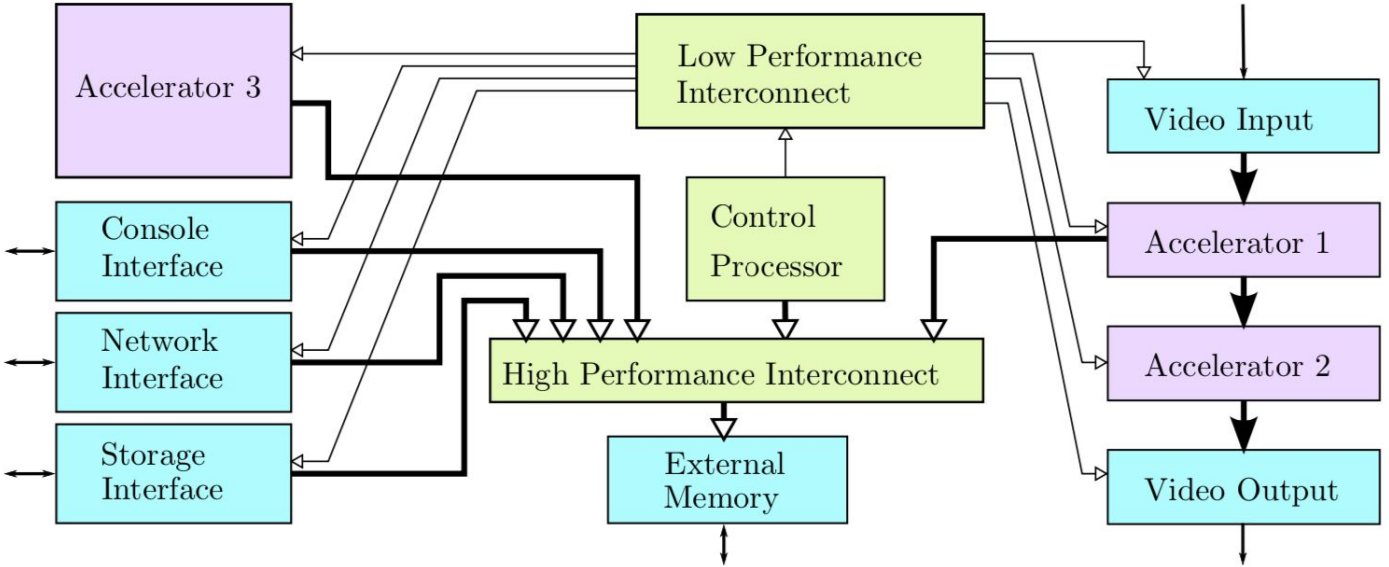
- **External Memory**: highest density, lowest bandwidth
- **FFs**: highest total bandwidth, limited amount of data storage capability.
- **BRAM**: intermediate value between external memory and FFs

PYNQ-Z2


Headphone + MIC
USB Host
HDMI Out
Raspberry Pi
Line In
HDMI In
Ethernet (RJ45)
Jumper for Boot Mode Selection
ARM A9 and FPGA
DRAM
Pmod A
JTAG / UART (Micro USB)
Arduino
Power Switch
Pmod B
DC Power In (7V-15V)
4 Buttons (User Defined)
RGB LEDs (User Defined)
4 LEDs (User Defined)
Jumper for Power Source Selection
2 Slide Switches (User Defined)
Buttons for Programming and System Reset
SD Card (On the Back Side)

FPGA Design Process

Block diagram showing a hypothetical embedded FPGA design



- **I/O Interface Core**: highly constrained by timing, RTL implemented
- **Standard Core**: Generic, fixed-function processing,
  - Processor core, on-chip memory, interconnections
- **Accelerator core**:
  - Where we focus on and develop by using HLS.

Design Optimization

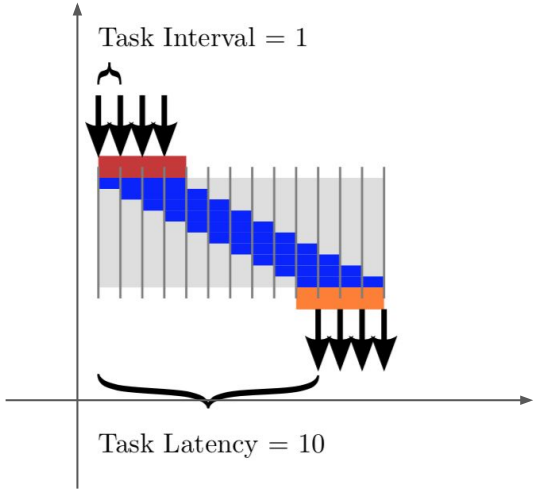
Benchmarking Metrics for Hardware Design

- Accuracy
  - Quality of result for a given task
- Throughput
  - Analytics on high volume data
  - Real-time performance (e.g., video at 30 fps)
- Latency
  - For interactive applications (e.g., autonomous navigation)
- Energy and Power
  - Edge and embedded devices have limited battery capacity
  - Data centers have stringent power ceilings due to cooling costs
- Hardware Cost

Reference: https://www.rle.mit.edu/eems/wp-content/uploads/2019/09/2019_icip_tutorial.pdf

Definitions

- Task
  - Function invocation, e.g. int32_t func(int32_t a)
- Task latency
  - Time between task start and when it finishes
- Task interval
  - Time between one task starts and the next starts



- **Horizontal Axis**: time
- **Vertical Axis**: Functional units in design
  - E.g. Mul(*), Add(+)
- Four executions of a design, starting a new task every cycle

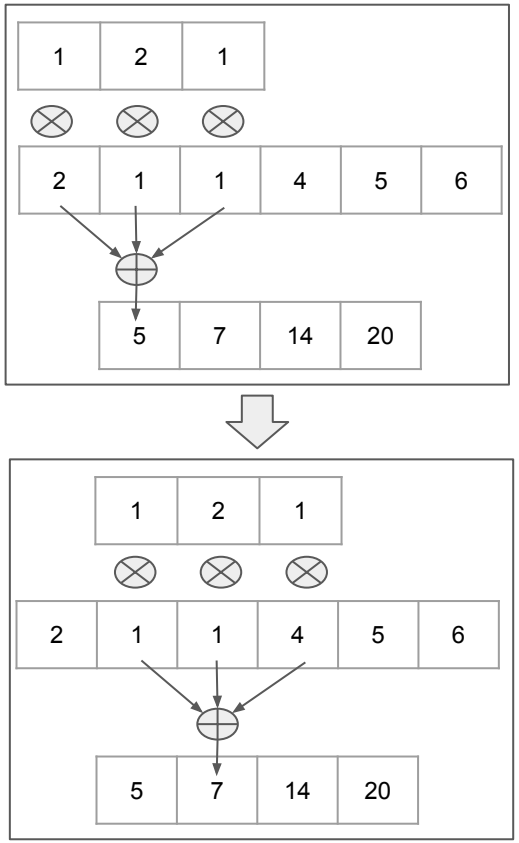
Area/Throughput Tradeoffs

- Example: FIR filter
  - 1-Dim convolution
  - Key question: what circuit is generated from this code?

```
#define NUM_TAPS 4

void fir(int input, int *output, int taps[NUM_TAPS])
{
  static int delay_line[NUM_TAPS] = {};

  int result = 0;
  for (int i = NUM_TAPS - 1; i > 0; i--) {
    delay_line[i] = delay_line[i - 1];
  }
  delay_line[0] = input;

  for (int i = 0; i < NUM_TAPS; i++) {
    result += delay_line[i] * taps[i];
  }

  *output = result;
}
```

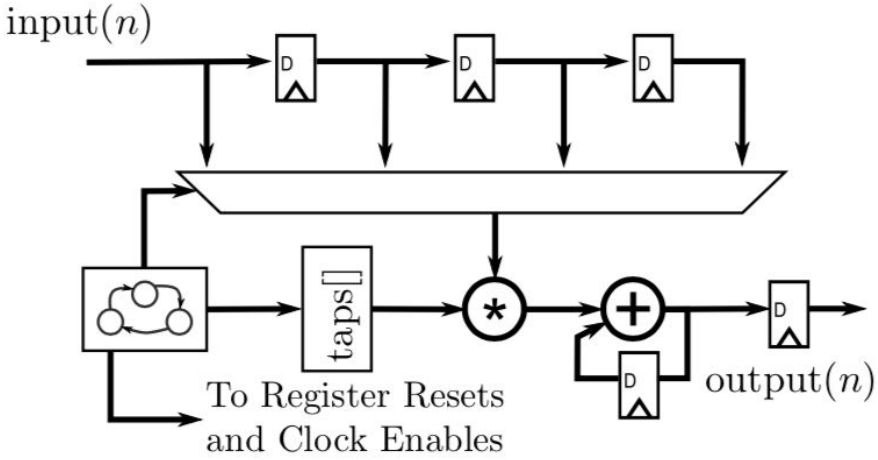
Left shift one

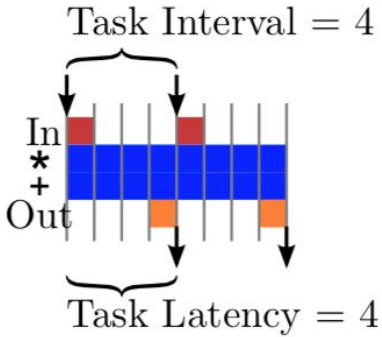
Inner-product



Area/Throughput Tradeoffs

- Architecture generated from previous slide



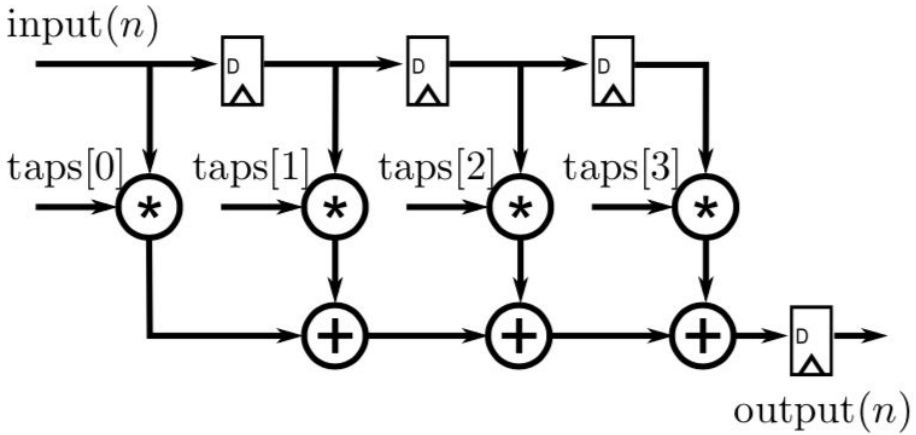


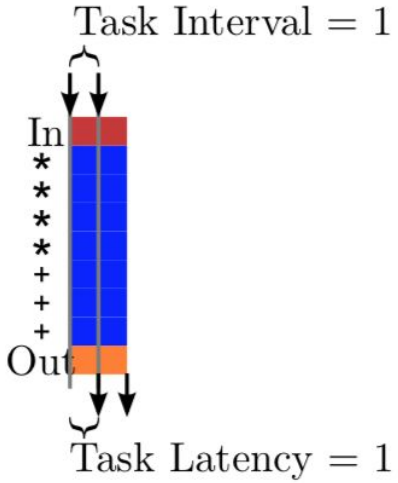
Design 1

- **Advantage**: Less resources on multiplication and add operations
- **Disadvantage**: Higher latency
- **Disadvantage**: more resources on control logics

Area/Throughput Tradeoffs

- With little modifications on HLS(Next week)





Design 2

- **Advantage**: Higher processing rate, less latency
- **Advantage**: Simpler architecture
- **Disadvantage**: Need more resources on Mul and Add

In this course

Chapter	FIR 2	CORDIC 3	DFT 4	FFT 5	SPMV 6	MatMul 7	Histogram 8	Video 9	Sorting 10	Huffman 11
Loop Unrolling	X		X	X	X		X		X	
Loop Pipelining	X		X	X	X		X	X	X	X
Bitwidth Optimization	X	X								X
Function Inlining	X									X
Hierarchy	X			X			X	X	X	X
Array Optimizations			X	X	X	X	X	X	X	X
Task Pipelining				X			X	X	X	X
Testbench					X	X			X	X
Co-simulation					X					
Streaming						X		X	X	
Interfacing								X		

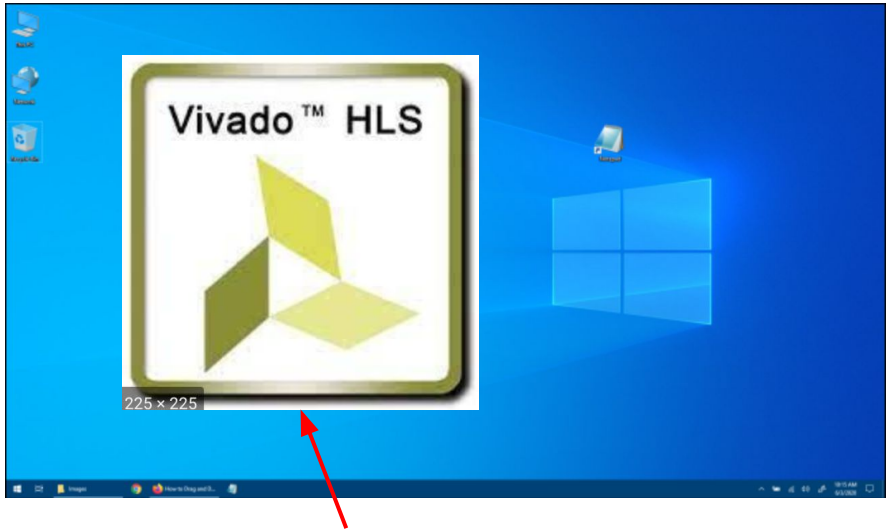
Lab 1: Overlay Tutorial

Overview

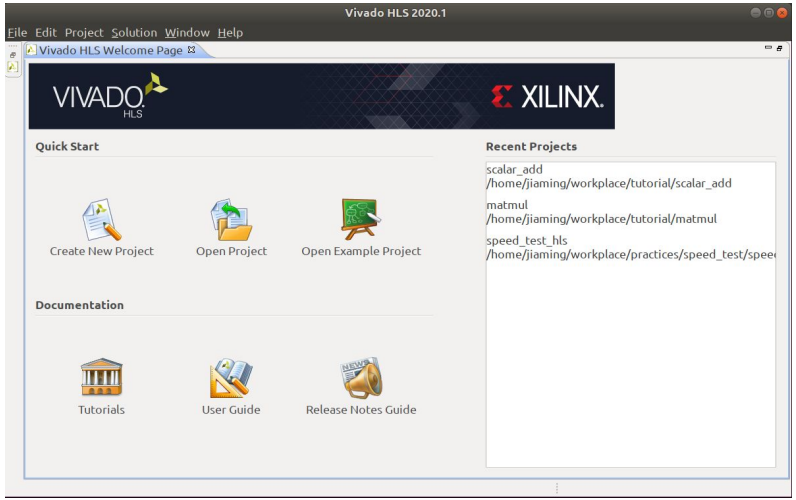
- Developing a Single IP
  - Intellectual Property(IP) is a reusable IC design that is the intellectual property of one party.
  - Using Vivado HLS
- Generating Bitstream and download to FPGA
- Interact with our own IP by using PYNQ

Developing a Single IP using Vivado HLS

Open the Vivado HLS 2020.1 IDE

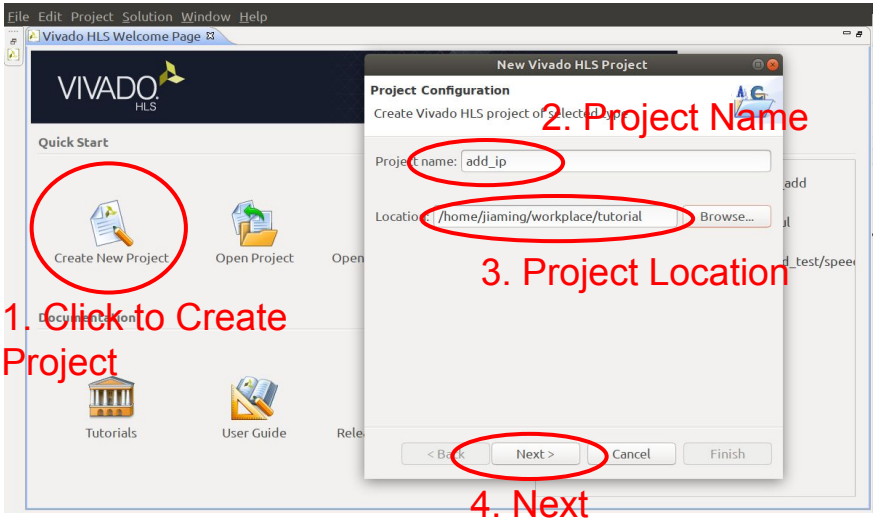


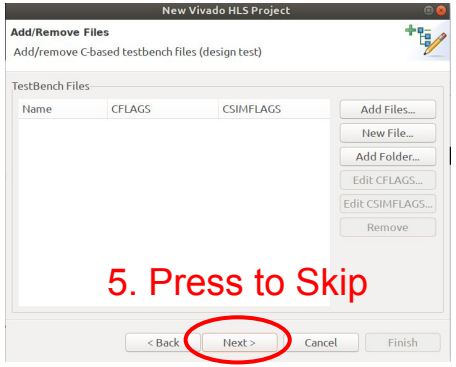
Short path on desktop

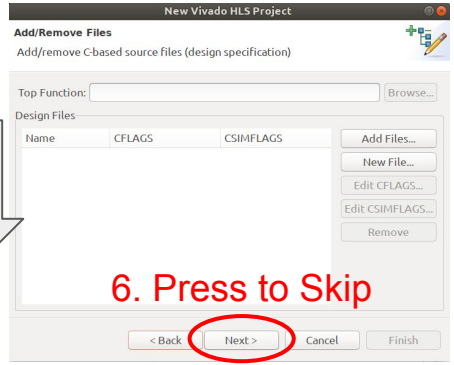


Developing a Single IP using Vivado HLS

Create the Project

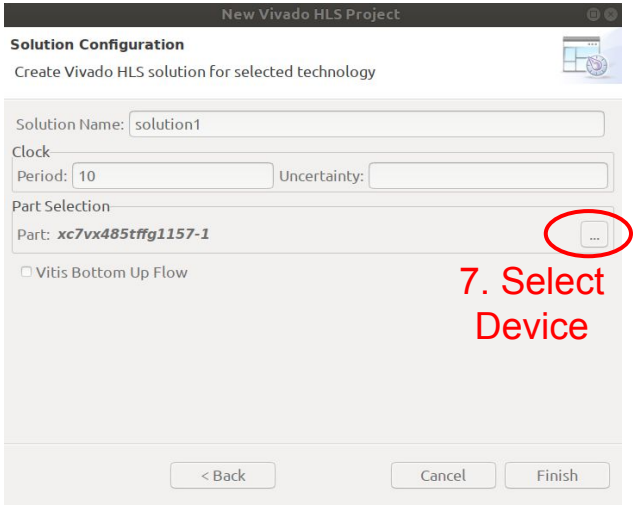


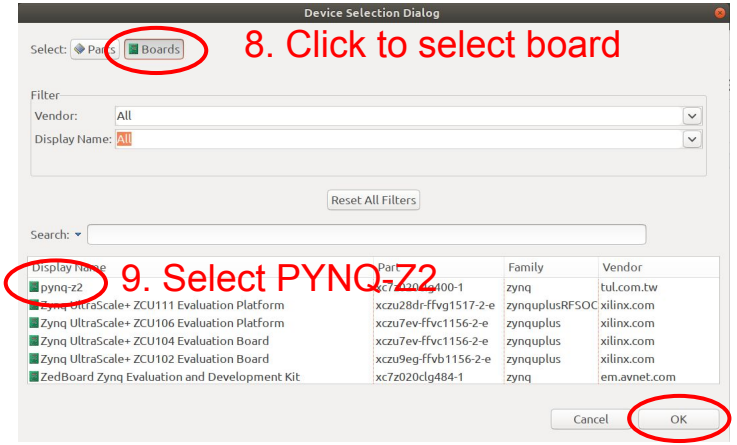




Developing a Single IP using Vivado HLS

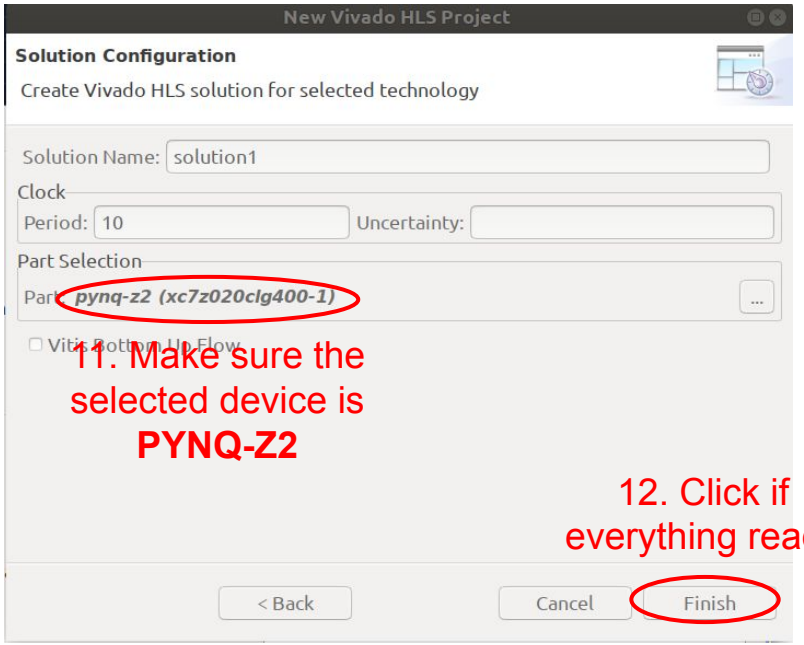
Create the Project

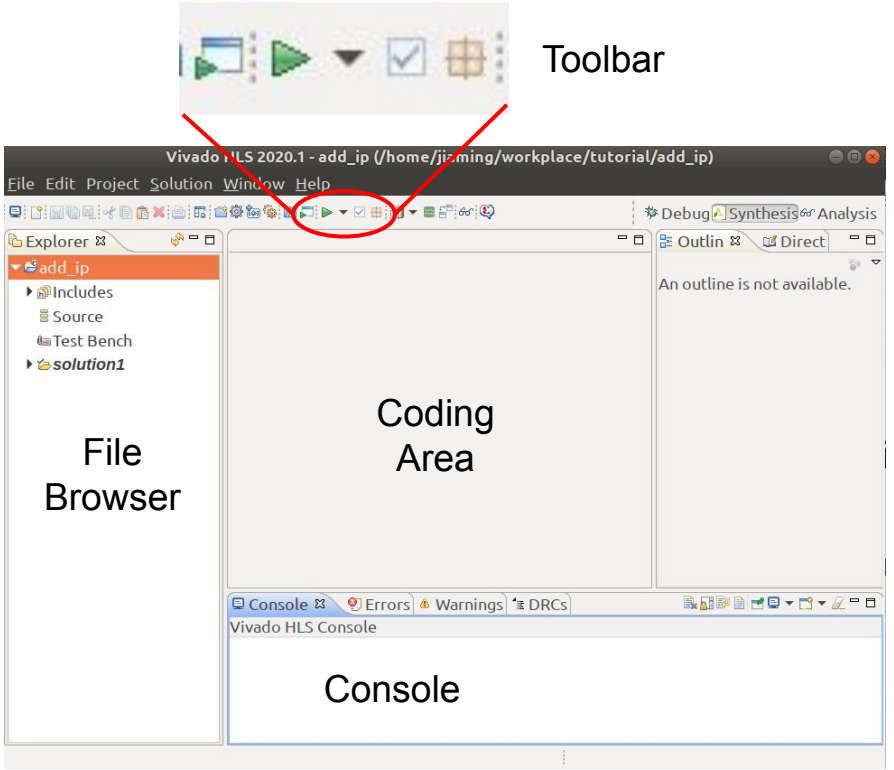




Developing a Single IP using Vivado HLS

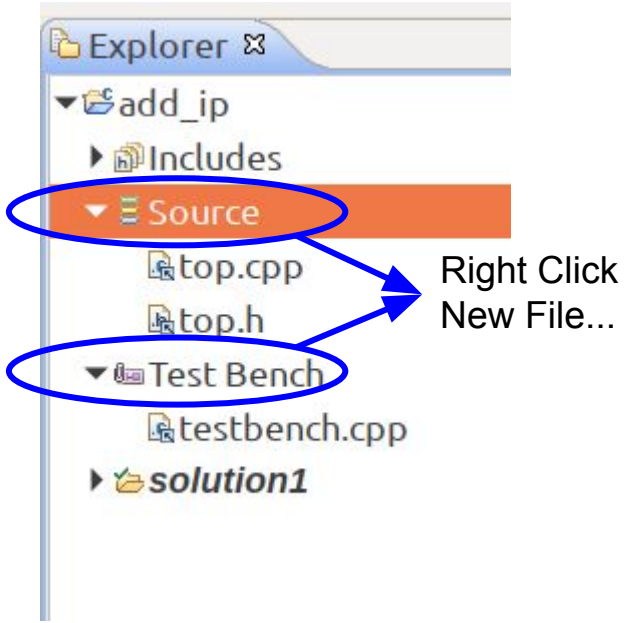
Create Project





Developing a Single IP using Vivado HLS

Create New Files



Three new files…

- **Under Source,**
  (1) **top.h**: Define datatype or function, ...
  (2) **top.cpp**: Hardware design implementation
- **Under Test Bench,**
  (1) **testbench.cpp**: Check hardware/software result is matched.

Developing a Single IP using Vivado HLS

top.h

```
#include <iostream>
#include <ap_int.h>

using namespace std;

// 32 bit integer
typedef ap_int<32> data_t;

void add(data_t a, data_t b, data_t &c);
```

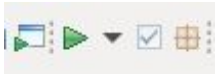
top.cpp

```
#include "top.h"

void add(data_t a, data_t b, data_t& c) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=c

    c = a + b;
}
```

testbench.cpp

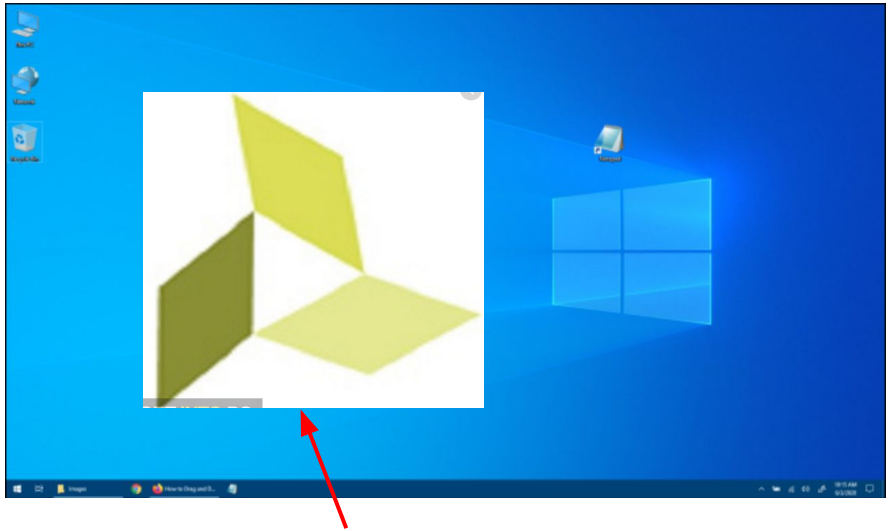
```
#include "top.h"

int main(){
    data_t a = 5;
    data_t b = 6;
    data_t c;

    add(a,b,c);

    int err = 0;

    if (c != (a+b)){
        err+=1;
        cout << "Error, HW c = "
             << c << endl;
    }
    return err;
}
```

Developing a Single IP using Vivado HLS

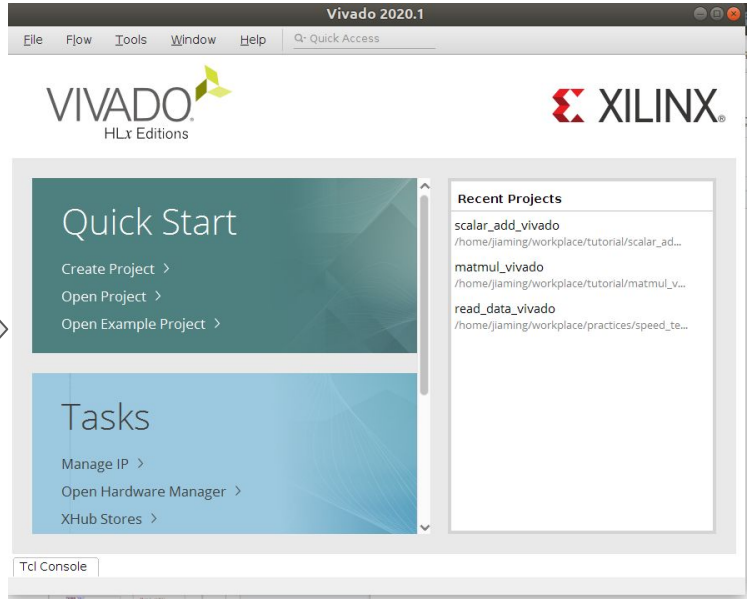
- Toolbar: 
  - **C Simulation**: Check the design output is same with software implementation
  - **C Synthesis**:
    - Generate RTL design(e.g. Verilog) from HLS C code
    - Performance and Resource consumption report
  - **C/RTL Cosimulation**: Check RTL output is matched with software implementation.
  - **Export RTL**: wrap the design to an IP module
- In this Lab, click "**C Simulation**" → "**C Synthesis**" → "**Export RTL**"

Generating Bitstream and Download to FPGA

- Open Vivado

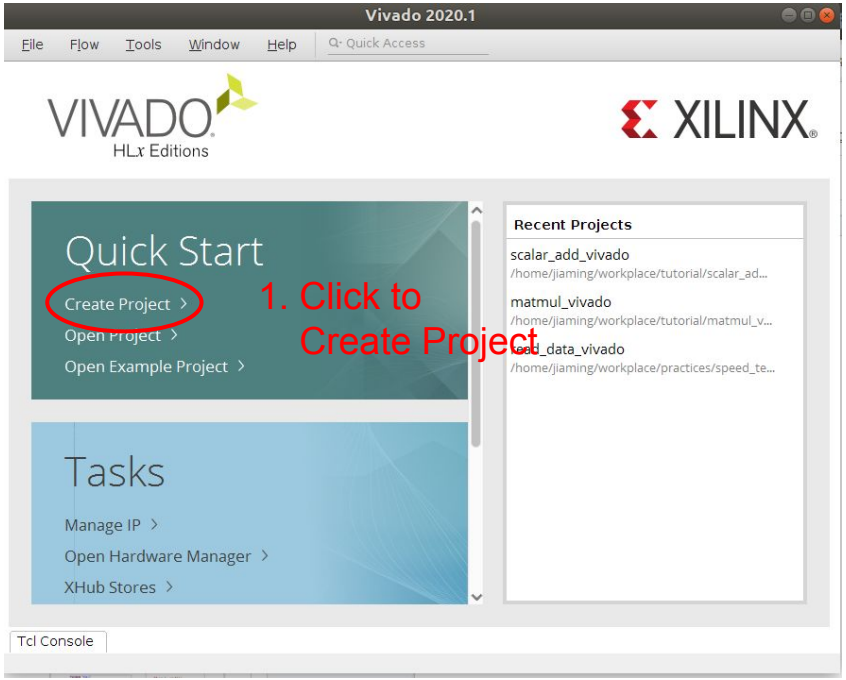


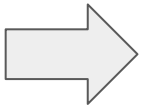
Short path on desktop

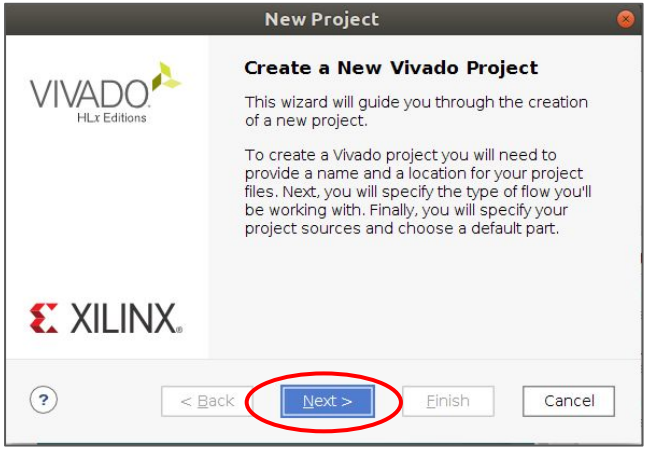


Generating Bitstream and Download to FPGA

- Create Vivado Project



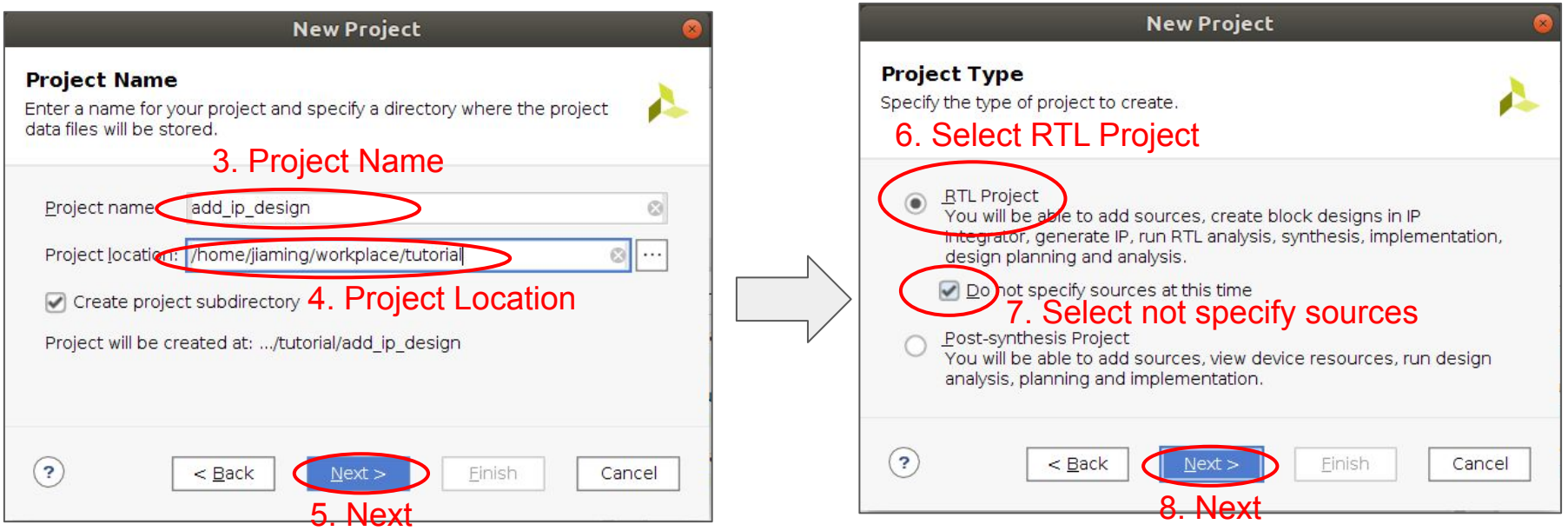




2. Click Next

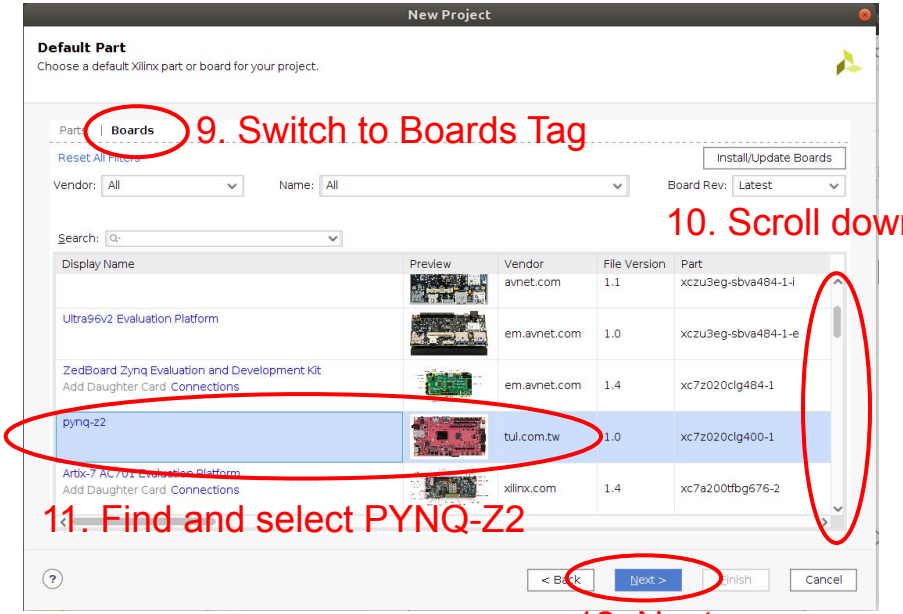
Generating Bitstream and Download to FPGA

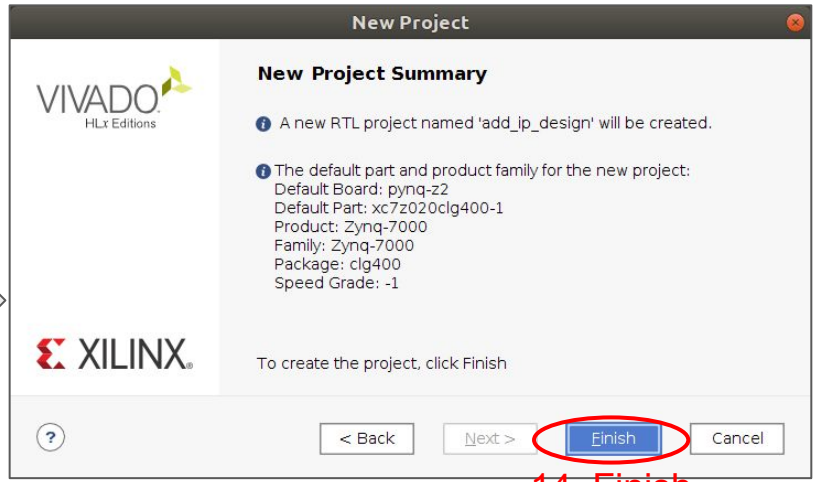
- Create Vivado Project



Generating Bitstream and Download to FPGA

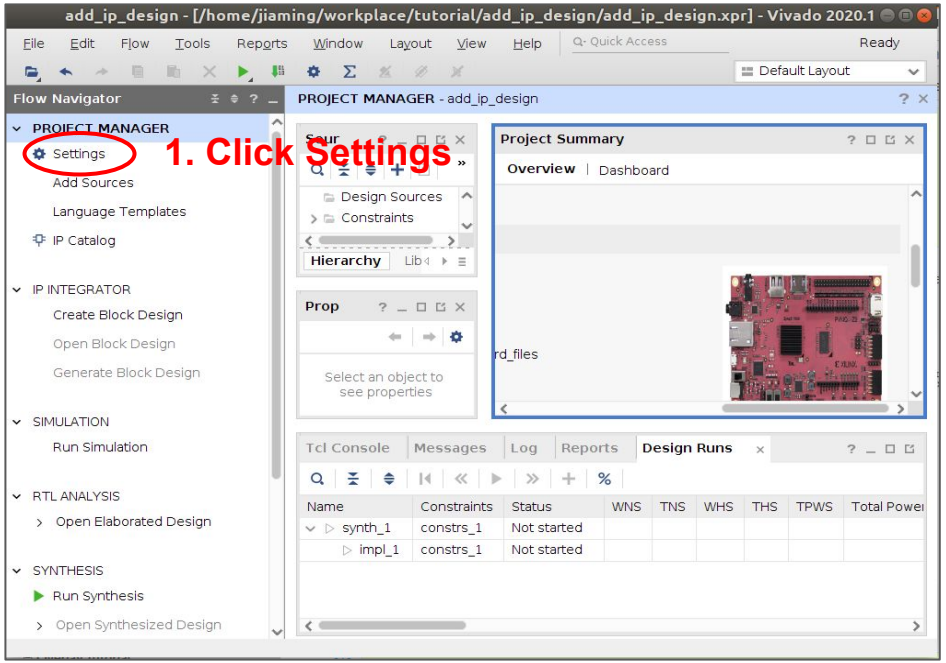
- Create Vivado Project

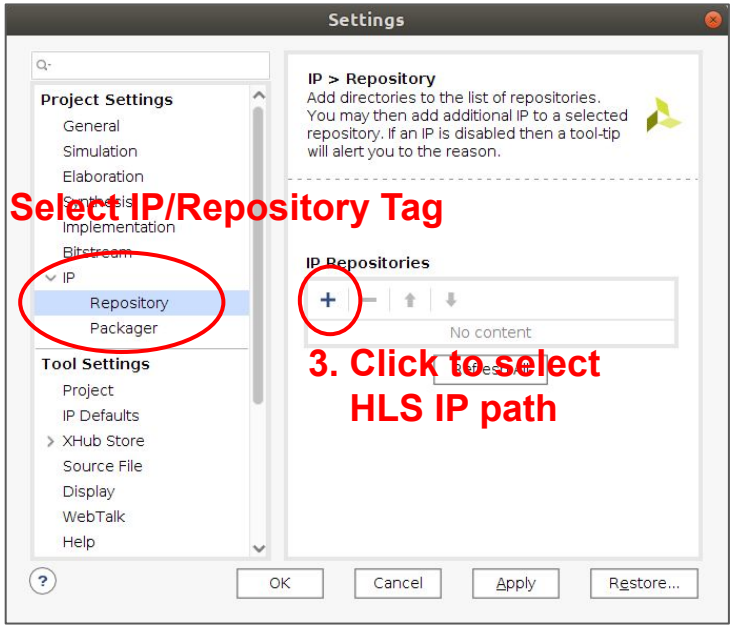




Generating Bitstream and Download to FPGA

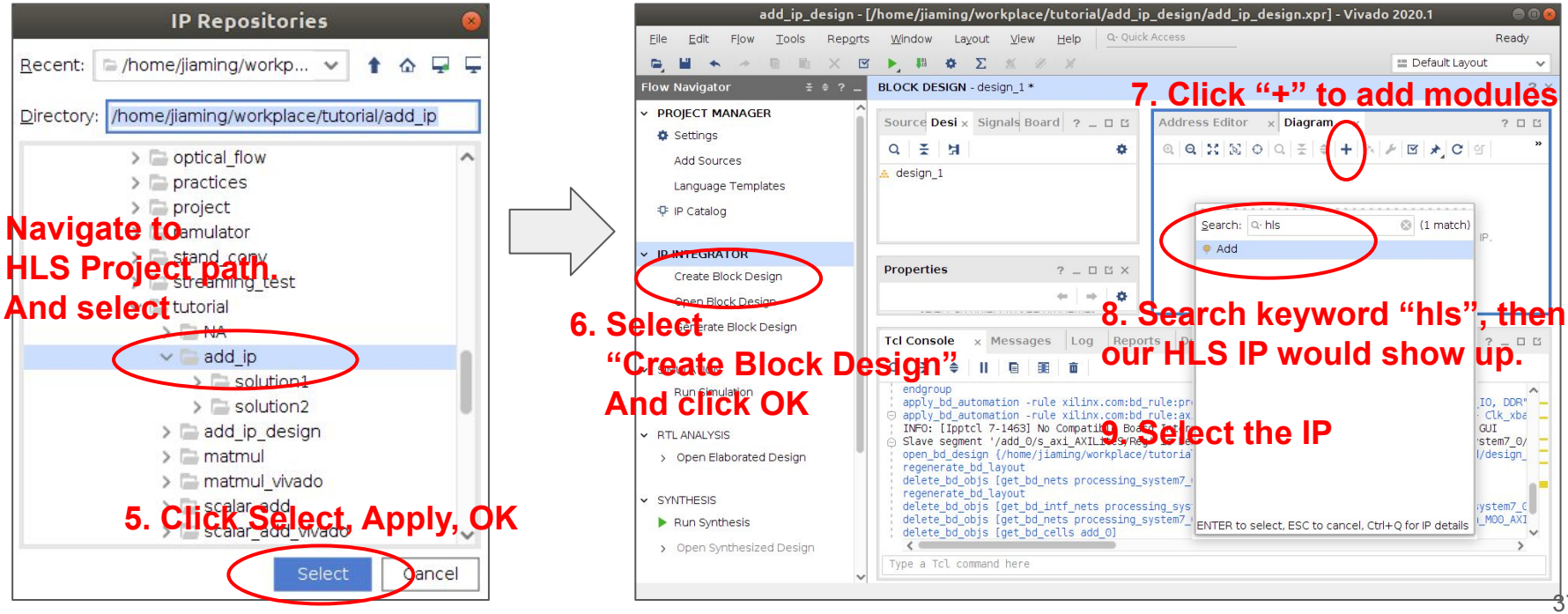
- Import HLS IP





Generating Bitstream and Download to FPGA

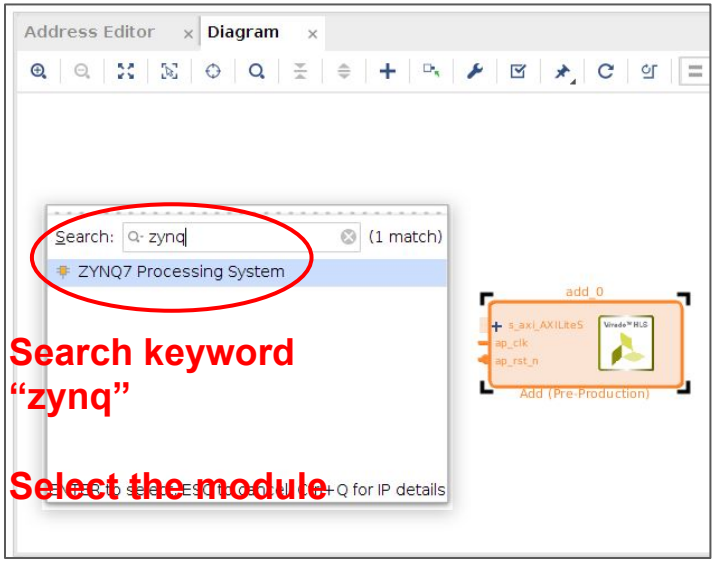
- Import HLS IP

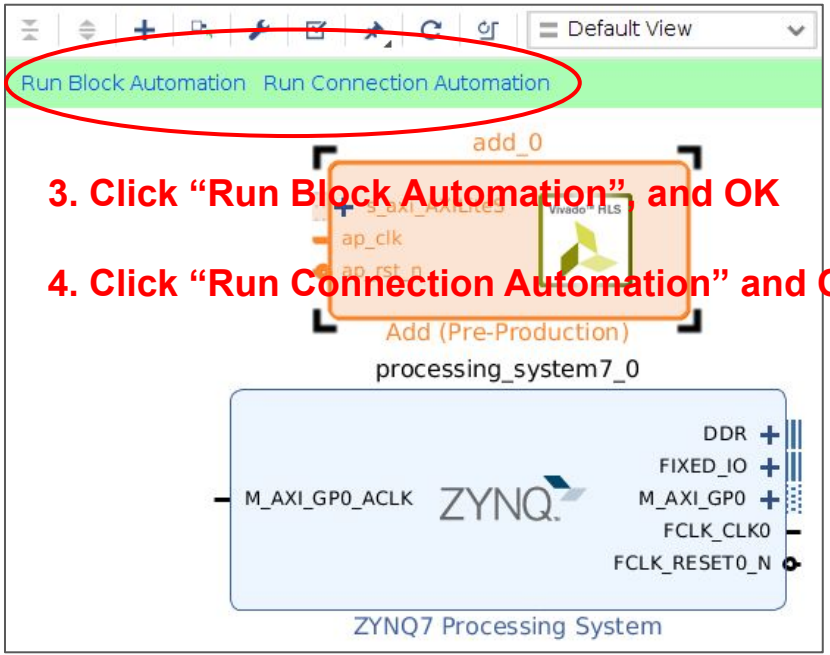


Generating Bitstream and Download to FPGA

- Constructing Block Design

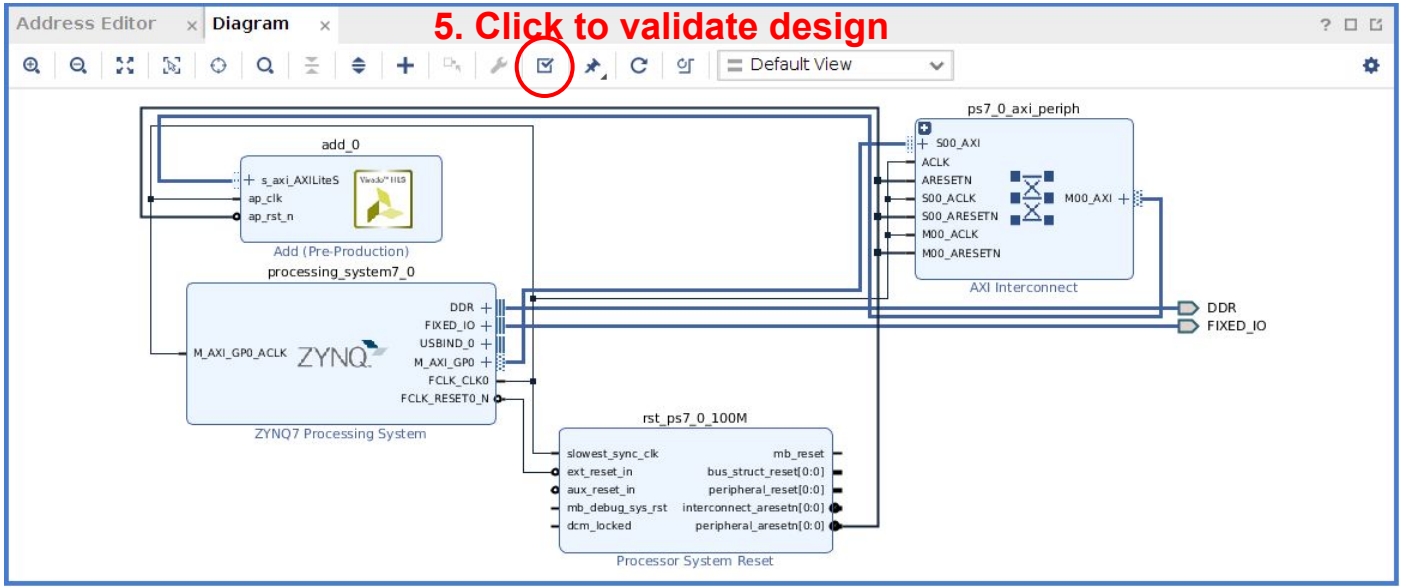
The Created Block Design View





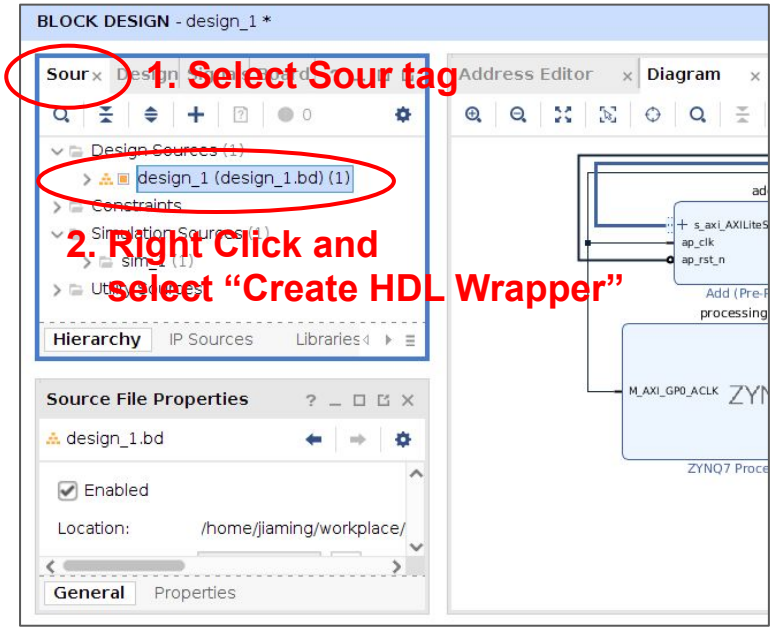
Generating Bitstream and Download to FPGA

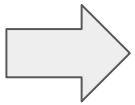
- Constructing Block Design

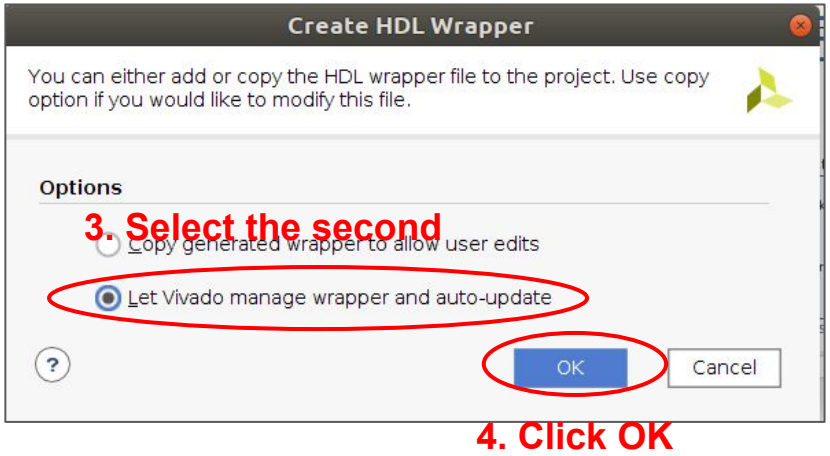


Generating Bitstream and Download to FPGA

- Generate Bitstream

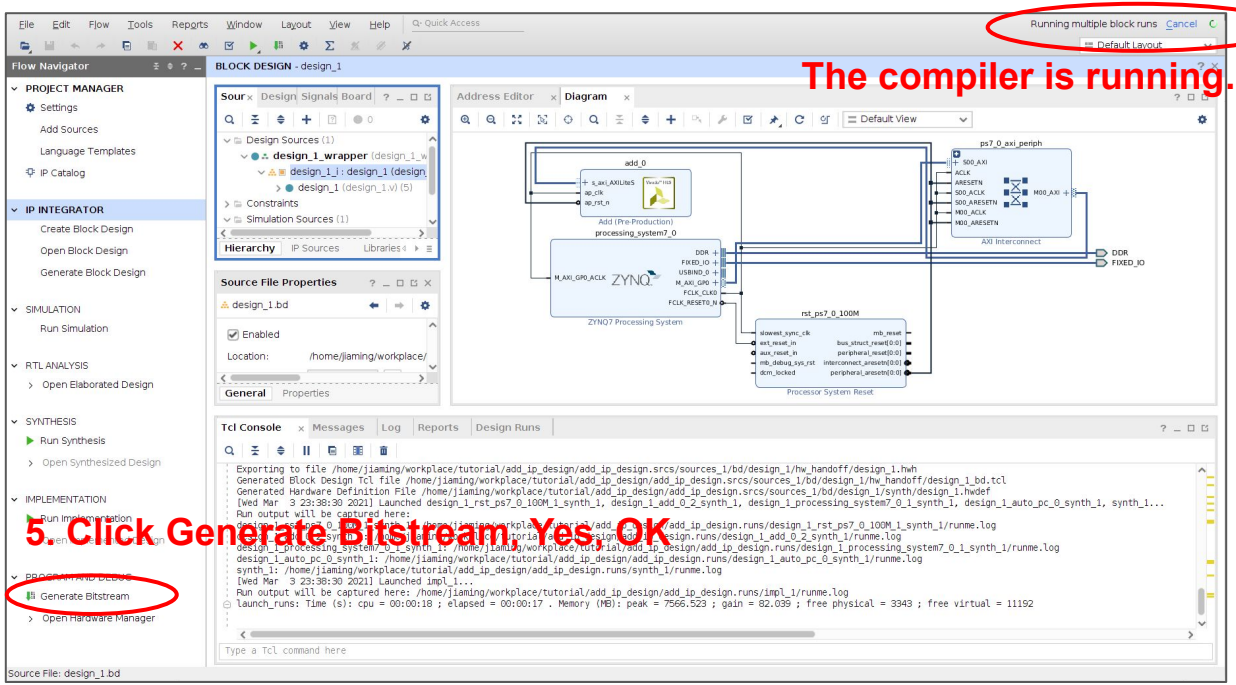






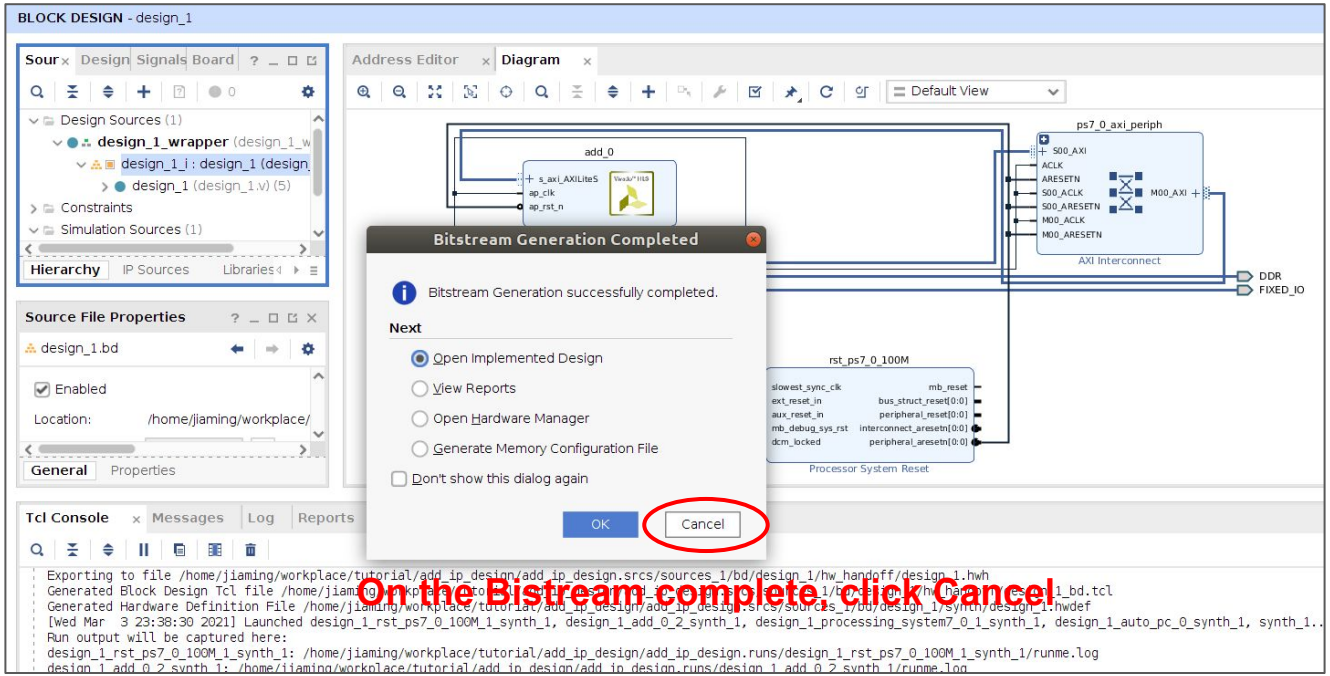
Generating Bitstream and Download to FPGA

- Generate Bitstream



Generating Bitstream and Download to FPGA

- Generate Bitstream

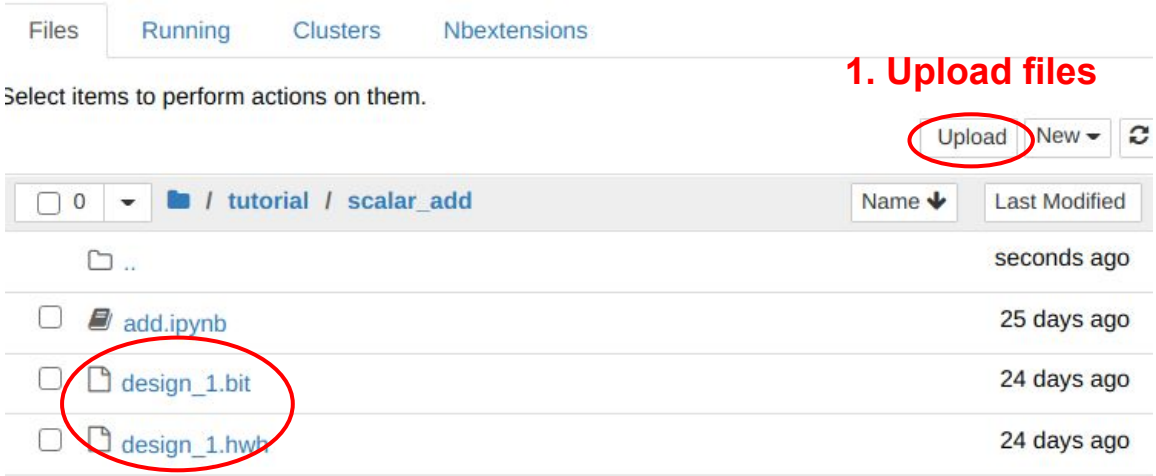


Generating Bitstream and Download to FPGA

- Two Design files
  - **<Your Design Path>/<Your Project Name>.srcs**
    /add_ip_design.srcs/sources_1/bd/design_1/hw_handoff/**design_1.hwh**
  - **<Your Design Path>/<Your Project Name>.runs**/impl_1/**design_1_wrapper.bit**
- Power on PYNQ-Z2 and create new folder from Jupyter-Notebook

Generating Bitstream and Download to FPGA

- Transfer two design files to PYNQ-Z2 by using file upload in Jupyter-Notebook



Generating Bitstream and Download to FPGA

- Create new Notebook, named “add.ipynb”

```
In [1]: from pynq import Overlay

In [3]: overlay = Overlay("./design_1.bit")
        overlay? # show the modules in the overlay

        Bitstream loaded

In [5]: Add = overlay.add_0

In [6]: Add.register_map

Out[6]: RegisterMap {
          a_V = Register(a_V=0),
          b_V = Register(b_V=0),
          c_V = Register(c_V=0),
          c_V_ctrl = Register(c_V_ap_vld=1, RESERVED=0)
        }

In [9]: Add.register_map.a_V = 12
        Add.register_map.b_V = 4
        Add.register_map.c_V

Out[9]: Register(c_V=16)

In [10]: Add.write(0x10, 6) # a
         Add.write(0x18, 7) # b
         Add.read(0x20) # c

Out[10]: 13
```

Download bitstream to FPGA

Interact with the IP using register map

Interact with the IP using offset

Summary

- FPGA Architecture
  - CLB,
  - Programmable Interconnection
  - Modules in Morden FPGA
  - Memory Hierarchy
- Design Optimization
  - Task
  - Task Latency
  - Task Interval
- Next Week
  - PP4FPGA chapter 2: Hardware Implementation of Finite Impulse Response and Optimizations.