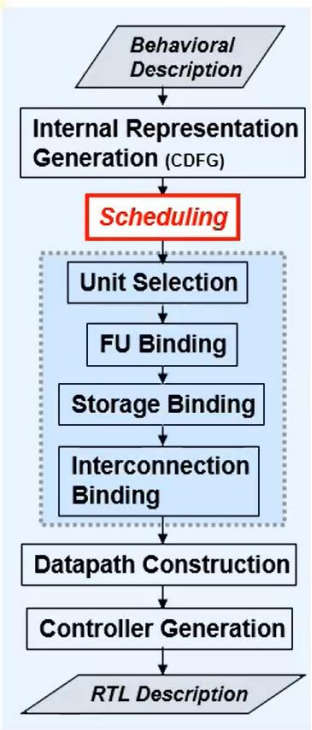
Lab1 Report

- Overall HLS development flow:
  - Concept:

    HLS design process maps an algorithm description (in C/C++) to an RTL (in Verilog/VHDL) implementation. That is to say, we can more focus on the top-level design(algorithm) than to consider the logic or circuit(architecture) behaviors.

  - Flow:

    How can a high-level language code be analyzed and optimized such that it can be transformed into the RTL description?

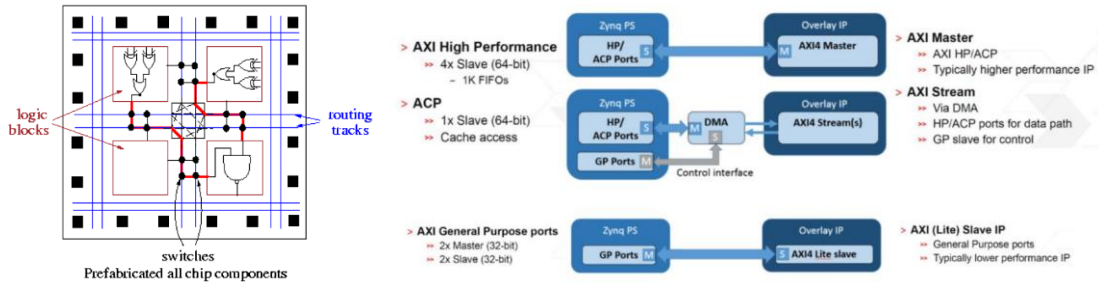


- Behavioral description:
  - ✓ C or C++ or signal-flow graph, etc.
- Internal Representation:
  - ✓ Control-data-flow graph
- Operation scheduling:
  - ✓ Assigning operation to control steps
  - ✓ Timing arrangement
- Resource allocation and binding:
  - ✓ Specified operations to functional units.
  - ✓ Assigning variables to registers

- Kernel development flow:
  1. Once completing the C++ code implement, we need to "debug" the code. Hence, we need the testing code and run the C-simulation to verify the design.
  2. Next, we need to set up the directive and run the synthesis to guide the HLS optimization process. In this step we can see some metrics that show the usage of resources and the performance in estimation.
  3. To verifying the RTL implementation, we need to run Co-simulation step, which can tell us whether the RTL code been generated is functionally correct or not.

➢ Different Zynq PS-PL Interfaces:

PS stands for process system, which is the brain of the FPGA, whereas PL stands for programmable logic, which is the part that the logics and the interconnections are prefabricated. By setting different interface between PS and PL, we can get different level of the resource's usage and the performance.



➢ Data type and interface synthesis support:

Figure 41: **Data Type and Interface Synthesis Support**

Argument Type	Scalar		Array			Pointer or Reference			HLS::Stream
Interface Mode	Input	Return	I	I/O	O	I	I/O	O	I and O
ap_ctrl_none									
ap_ctrl_hs		D							
ap_ctrl_chain									
axis									
s_axilite									
m_axi									
ap_none	D					D			
ap_stable									
ap_ack									
ap_vld								D	
ap_ovld							D		
ap_hs									
ap_memory			D	D	D				
bram									
ap_fifo									D
ap_bus									

Supported D = Default Interface Not Supported

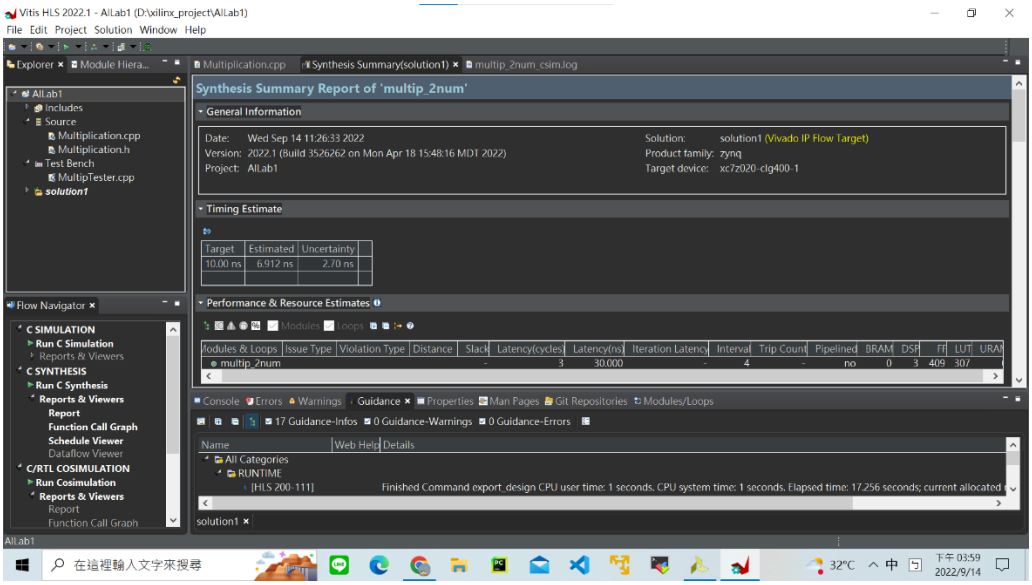
X14293

1. Block-level protocol: ap_ctrl_none, ap_ctrl_hs, ap_ctrl_chain
2. AXI interface protocol: axis, s_axilite, m_axi
3. I/O No protocol: ap_none, ap_stable
4. Wire handshake protocol: ap_ack, ap_vld, ap_ovld, ap_hs
5. Memory interface protocol(RAM, FIFO): ap_memroy, bram, ap_fifo
6. Bus protocol: ap_bus

➢ Lab1:

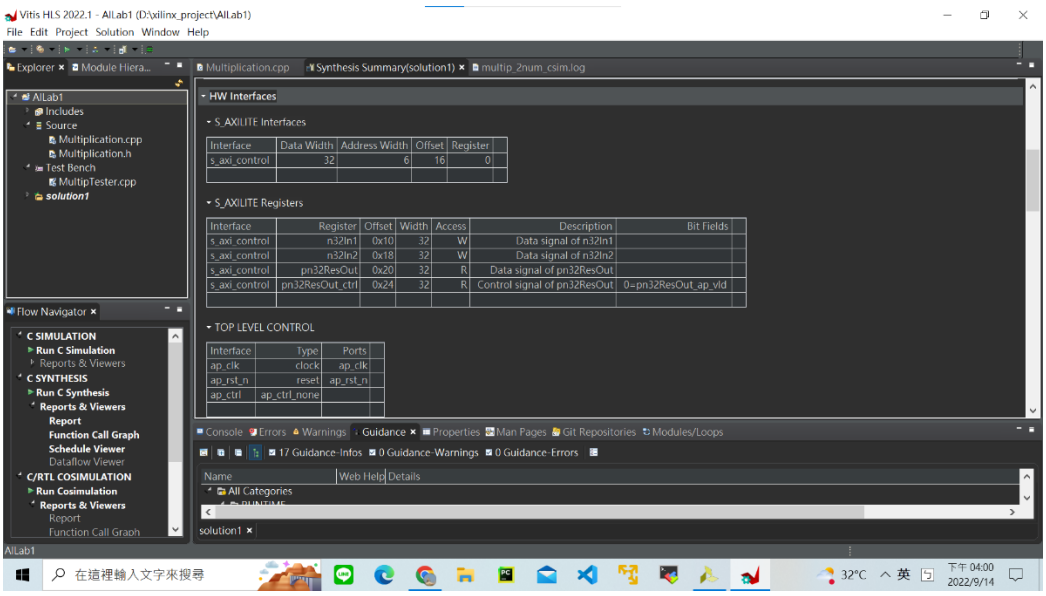
In lab1, we use pragma to set the input and output directives and follow the steps to complete the simulation and synthesis. After confirming the kernel function is correct, we can export the RTL to become an IP, which will be imported in the Vivado block design.

In this lab, I try to set the clock period to **5ns** and run the synthesis, end out occurring the **timing violation**. Hence, I set the clock period to **10ns**. I also try some different clock periods and find that the estimate resource usages are not the same.

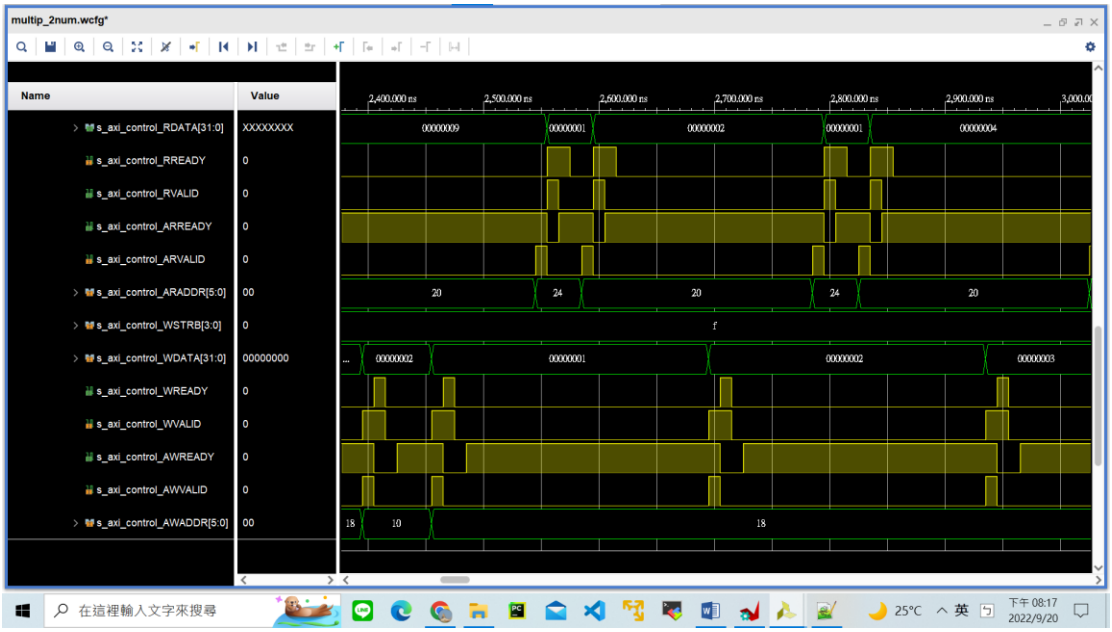


Modules & Loops	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
multip_2num	24	28	24	3	3	3

Estimate performance and resource

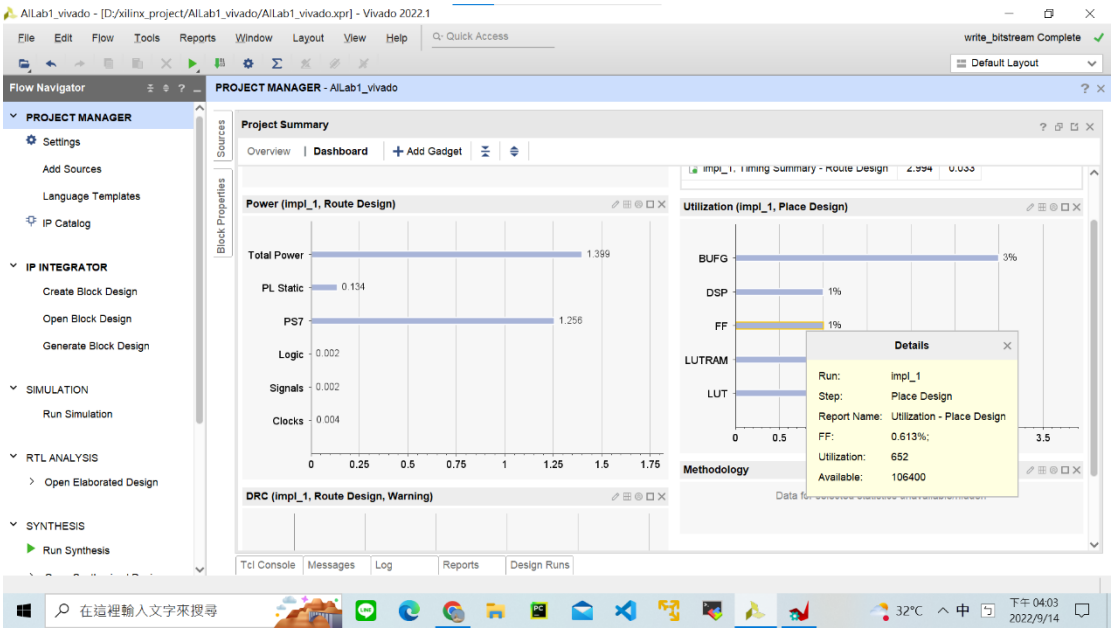


HW interface

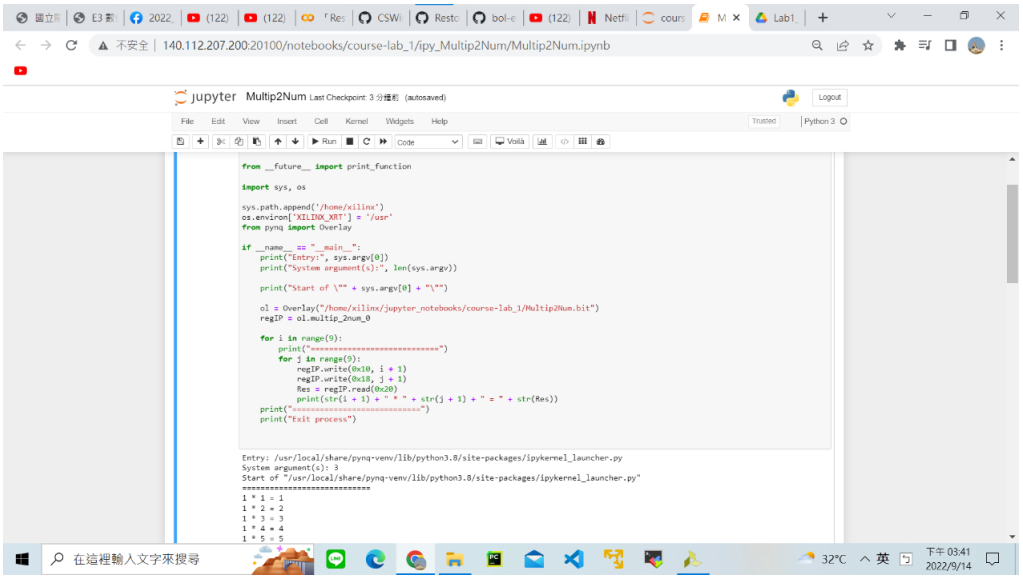


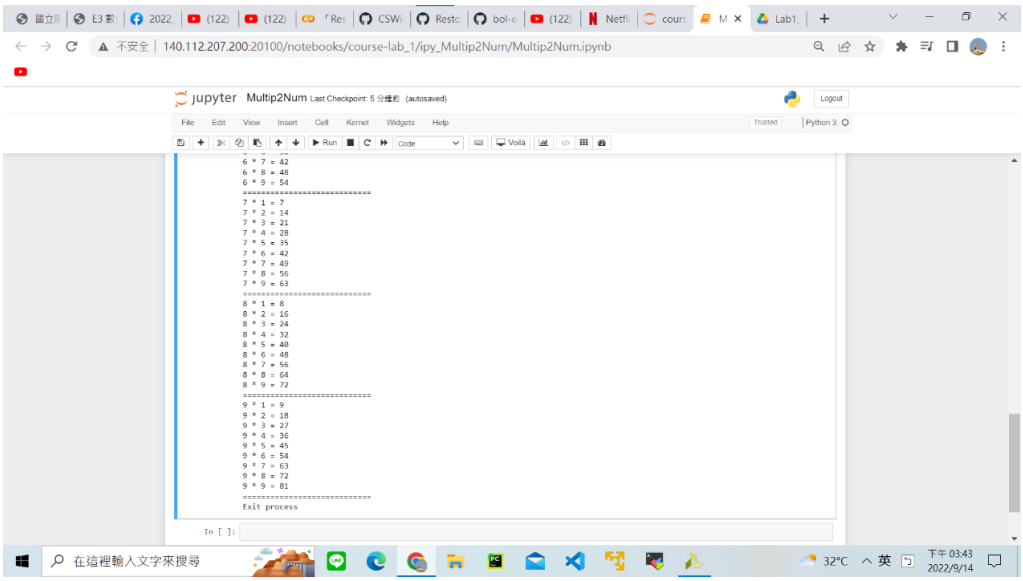
Waveform

From the waveform, we can see that the N32in1 port gets the value when it reads the address 0x10, and then store it in the register until the next read. Once N32in2 value is arrived, the output may start to compute. I observe that when the s_axi_control_AWREADY is high, there are some read, writes happen. When the s_axi_control RREADY and s_axi_control_RVALID is high, the value of the computed output will be written to the address 0x20, that is the reason why we need to read the answer by using the code regIP.read(0x20) to get the computed output value in the Jupyter-notebook.



Utilization





Result of the jupyter-notebook

```
// ==============================================================
// Vitis HLS - High-Level Synthesis from C, C++ and OpenCL v2022.1 (64-bit)
// Tool Version Limit: 2022.04
// Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.
// ==============================================================
// control
// 0x00 : reserved
// 0x04 : reserved
// 0x08 : reserved
// 0x0c : reserved
// 0x10 : Data signal of n32In1
//        bit 31~0 - n32In1[31:0] (Read/Write)
// 0x14 : reserved
// 0x18 : Data signal of n32In2
//        bit 31~0 - n32In2[31:0] (Read/Write)
// 0x1c : reserved
// 0x20 : Data signal of pn32ResOut
//        bit 31~0 - pn32ResOut[31:0] (Read)
// 0x24 : Control signal of pn32ResOut
//        bit 0  - pn32ResOut_ap_vld (Read/COR)
//        others - reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

#define XMULTIP_2NUM_CONTROL_ADDR_N32IN1_DATA     0x10
#define XMULTIP_2NUM_CONTROL_BITS_N32IN1_DATA     32
#define XMULTIP_2NUM_CONTROL_ADDR_N32IN2_DATA     0x18
#define XMULTIP_2NUM_CONTROL_BITS_N32IN2_DATA     32
#define XMULTIP_2NUM_CONTROL_ADDR_PN32RESOUT_DATA 0x20
#define XMULTIP_2NUM_CONTROL_BITS_PN32RESOUT_DATA 32
#define XMULTIP_2NUM_CONTROL_ADDR_PN32RESOUT_CTRL 0x24
```

HLS register definitions