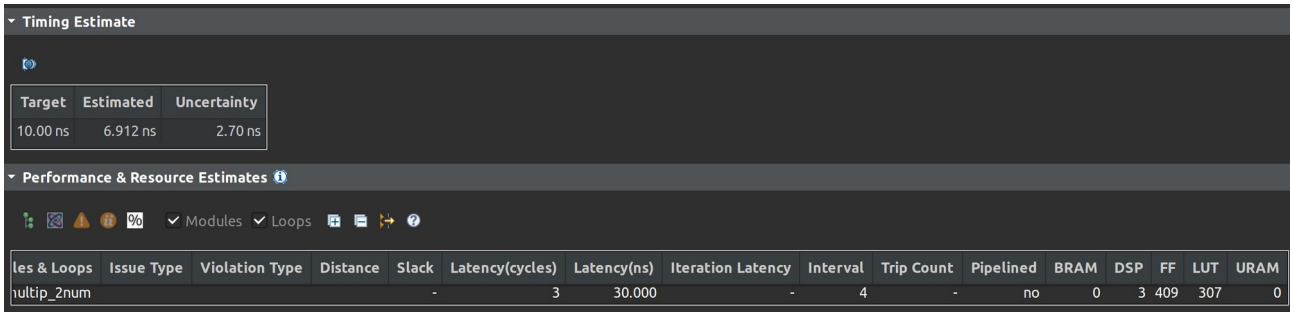
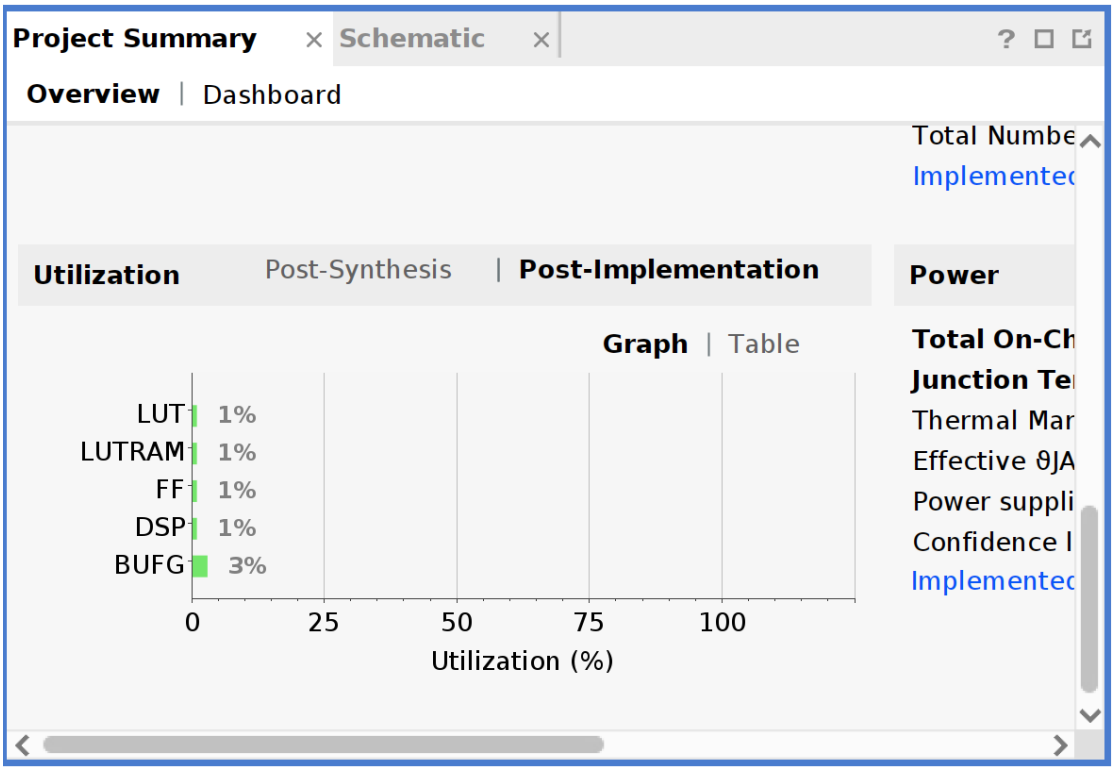
SoC Design Lab1

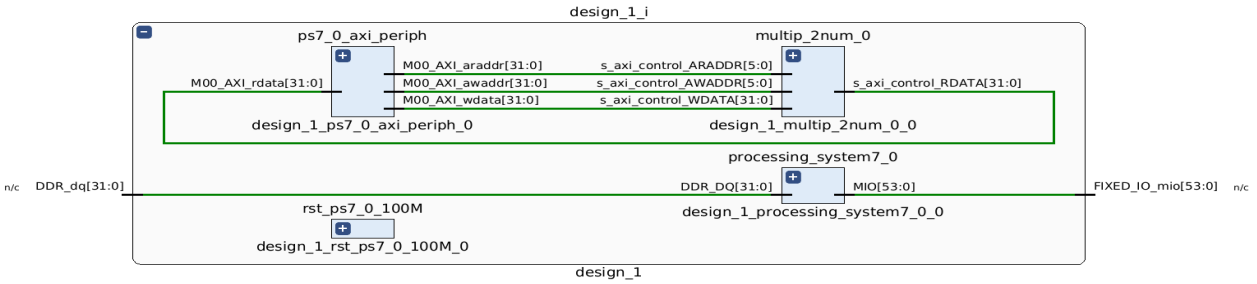
1. Performance:



2. Utilization:



3. Interface



HW Interfaces

S_AXILITE Interfaces

Interface	Data Width	Address Width	Offset	Register
s_axi_control	32	6	16	0

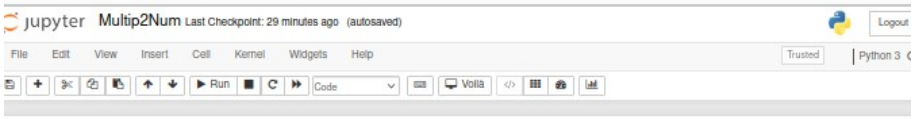
S_AXILITE Registers

Interface	Register	Offset	Width	Access	Description	Bit Fields
s_axi_control	n32In1	0x10	32	W	Data signal of n32In1	
s_axi_control	n32In2	0x18	32	W	Data signal of n32In2	
s_axi_control	pn32ResOut	0x20	32	R	Data signal of pn32ResOut	
s_axi_control	pn32ResOut_ctrl	0x24	32	R	Control signal of pn32ResOut	0=pn32ResOut_ap_vld

TOP LEVEL CONTROL

Interface	Type	Ports
ap_clk	clock	ap_clk
ap_rst_n	reset	ap_rst_n
ap_ctrl	ap_ctrl_hs	ap_done ap_idle ap_ready ap_start

4. Jupyter notebook result

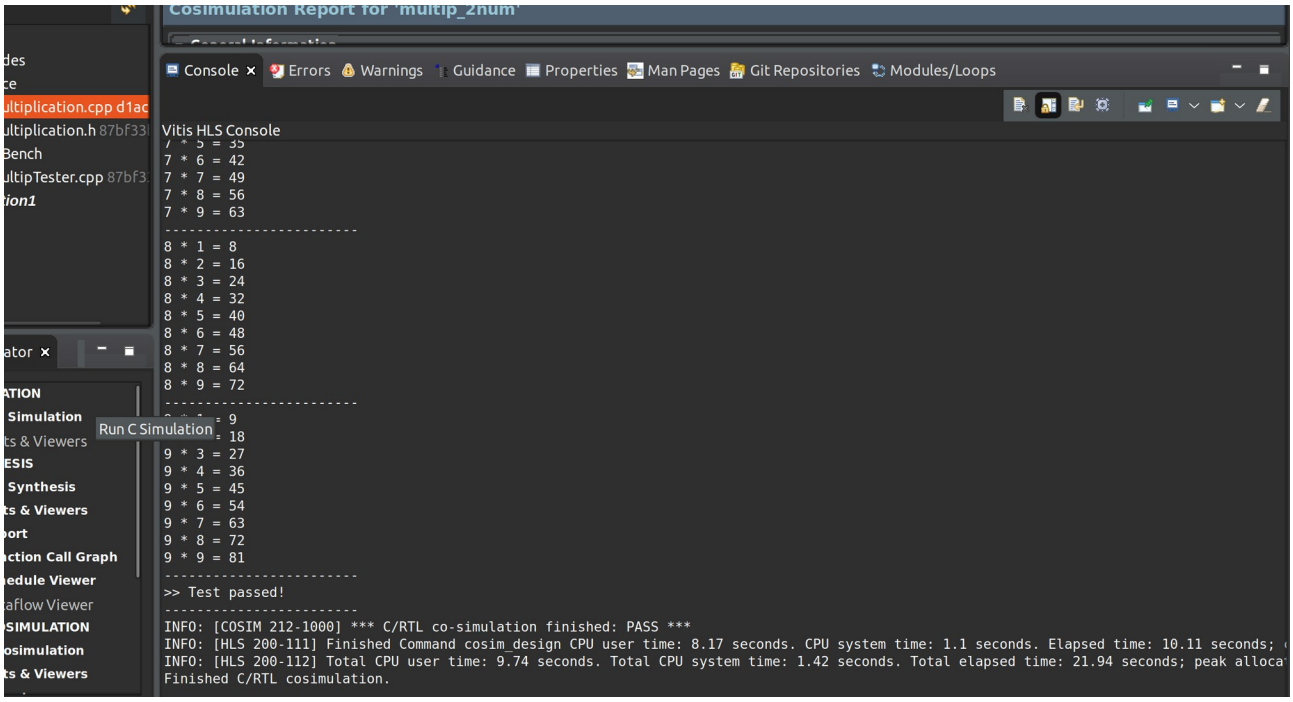


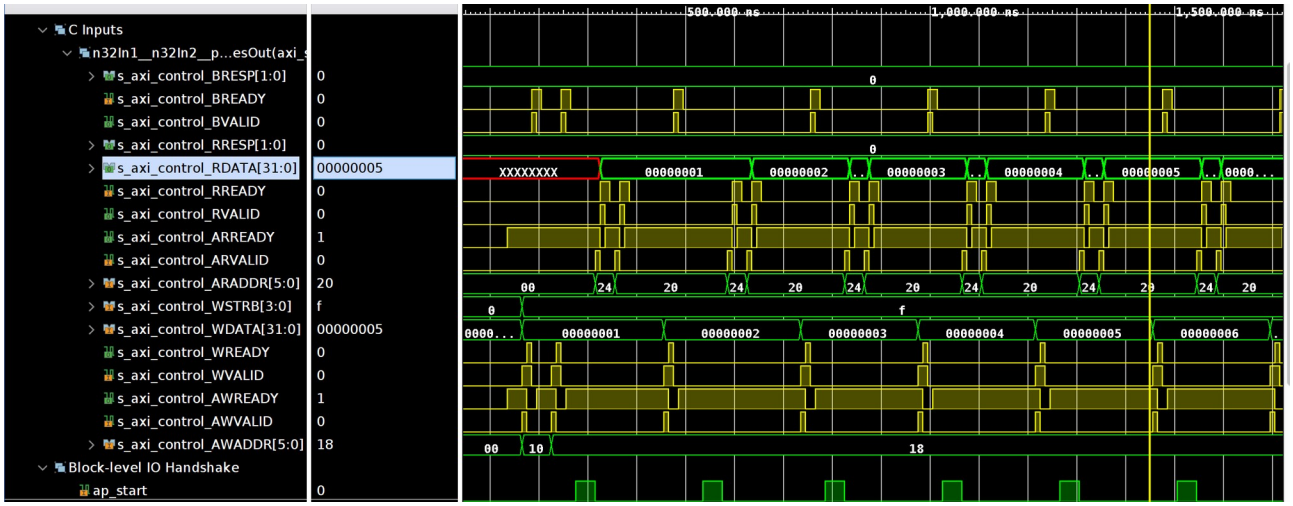
In [2]:

```
# coding: utf-8

# In[ ]:


from __future__ import print_function

import sys, os

sys.path.append('/home/xilinx')
os.environ['XILINX_XRT'] = '/usr'
from pynq import Overlay

if __name__ == "__main__":
    print("Entry:", sys.argv[0])
    print("System argument(s):", len(sys.argv))

    print("Start of \"" + sys.argv[0] + "\"")

    ol = Overlay("/home/xilinx/jupyter_notebooks/Multip2Num.bit")
    regIP = ol.multip_2num_0

    for i in range(9):
        print("============================")
        for j in range(9):
            regIP.write(0x10, i + 1)
            regIP.write(0x18, j + 1)
            Res = regIP.read(0x20)
            print(str(i + 1) + " * " + str(j + 1) + " = " + str(Res))
    print("============================")
    print("Exit process")


```

```
Entry: /usr/local/share/pynq-venv/lib/python3.8/site-packages/ipykernel_launcher.py
System argument(s): 3
Start of "/usr/local/share/pynq-venv/lib/python3.8/site-packages/ipykernel_launcher.py"
============================
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
1 * 9 = 9
============================
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
============================
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
3 * 5 = 15
3 * 6 = 18
3 * 7 = 21
3 * 8 = 24
3 * 9 = 27
============================
4 * 1 = 4
4 * 2 = 8
4 * 3 = 12
4 * 4 = 16
4 * 5 = 20
4 * 6 = 24
4 * 7 = 28
4 * 8 = 32
4 * 9 = 36
============================
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
============================
6 * 1 = 6
6 * 2 = 12
6 * 3 = 18
6 * 4 = 24
6 * 5 = 30
6 * 6 = 36
6 * 7 = 42
6 * 8 = 48
6 * 9 = 54
============================
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
============================
8 * 1 = 8
8 * 2 = 16
8 * 3 = 24
8 * 4 = 32
8 * 5 = 40
8 * 6 = 48
8 * 7 = 56
8 * 8 = 64
8 * 9 = 72
============================
9 * 1 = 9
9 * 2 = 18
9 * 3 = 27
9 * 4 = 36
9 * 5 = 45
9 * 6 = 54
9 * 7 = 63
9 * 8 = 72
9 * 9 = 81
============================
Exit process
```

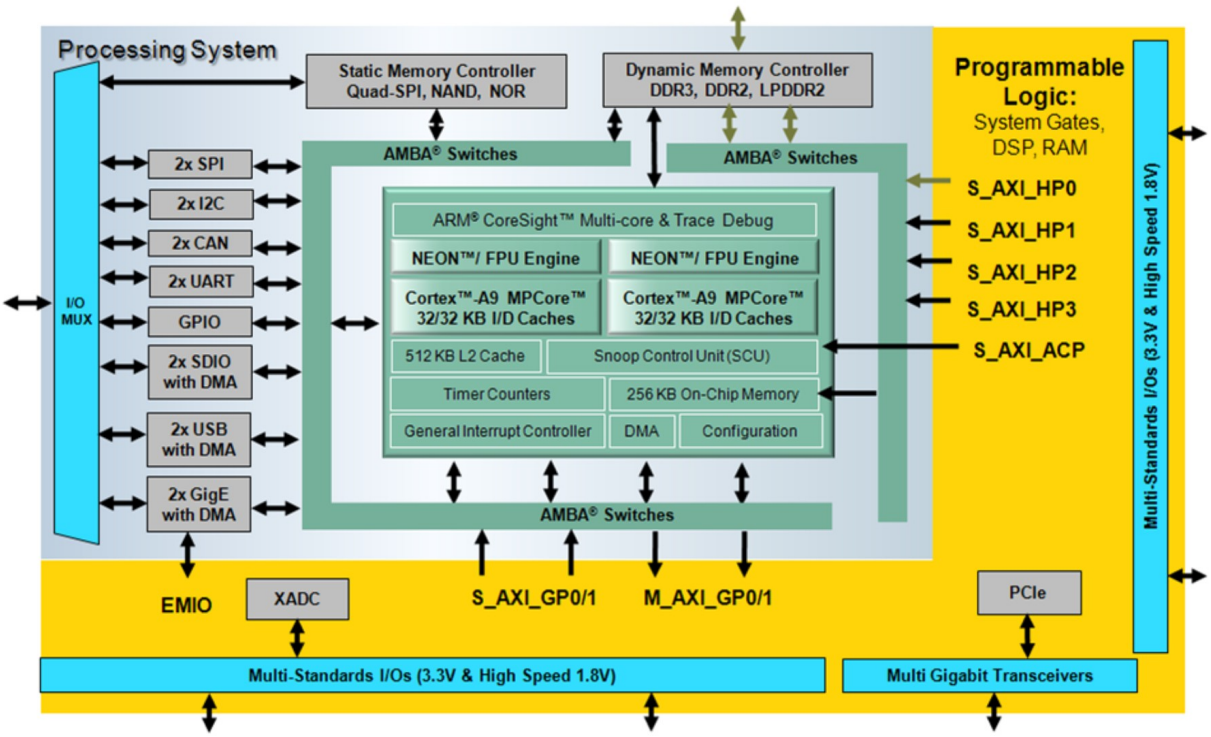
5. Cosimulation & waveform



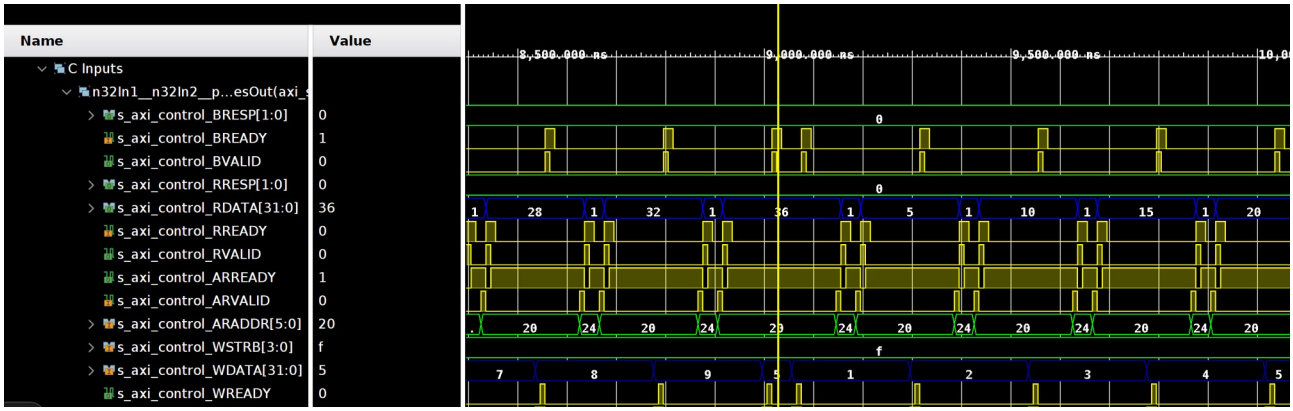


6. Introduction of the system:

In this practice, the codes provided is to create a multiplication module, and use AXILite bus to communicate with zynq processing system. The zynq processing system is as follow:

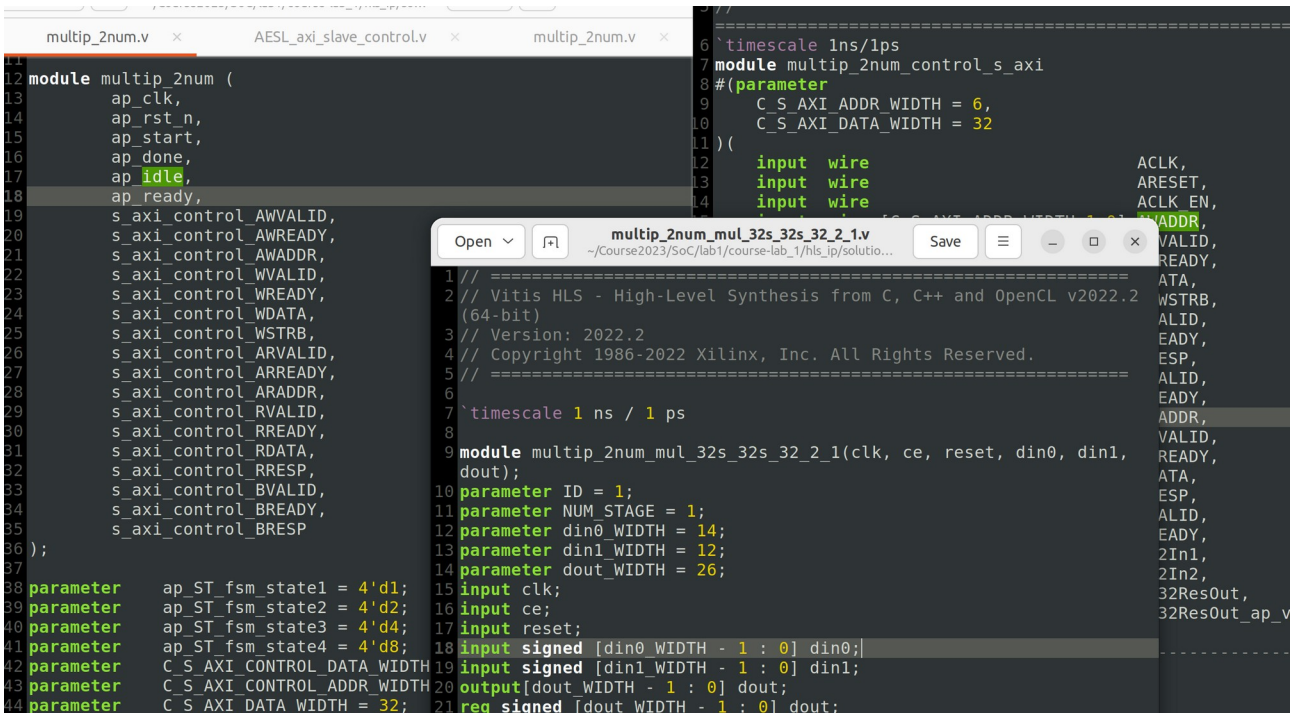


7. Observation & Learned:
Checking the waveform, we could observe how the data transfers and is received through the axilite bus. Let's take a look at the example for i = 5 and j = 1~4, the waveform shown as below,



Taking a look at the blue signal, the input is the "s_axi_control__WDATA" and the value is 5 and then set from 1 to 4 as shown in the picture, and after passing 13 cycles, the value of "s_axi_control_RDATA" becomes 5, which is the output in our case.

After looking at the signals shown in the waveform, I begin to interest in the structure of verilog code it generates. Following is the code it generates after running cosimulation.
We could see that there are three modules: multip_2num_control_s_axi(control the bus control signal), multip_2num_mul_32s_32s_32_2_1, and multip_2num(which calls the multip_2num_control_s_axi and multip_2num_mul_32s_32s_32_2_1), and we could see that how it generate the corresponding axi bus control signals in this case.



And one thing I notice is the block-level control signals as professor has mentioned in class, although I don't see it shows in pragma syntax or the directive, it indeed exists in the code, which is the module "multip_2num", and the code is as follows,

...
input ap_start;
output ap_done;
output ap_idle;
output ap_ready;
…