Brief Introduction about the overall system:

HLS 全名為 high level synthesis。顧名思義，就是用高階的語言去進行硬體合成，在 xilinx 的 vitis_hls 則是用 c/c++去進行高階演算法的描述，最終則在高階演算進行驗證後才做合成，這樣可以讓開發者多專注在演算法上而非邏輯閘的或是 register transfer level 上面的行為描述。

在這次 lab1 當中會用到兩個 tool: vitis_hls & vivado。
Vitis_hls 主要將 C++的演算法合成出硬體 IP。
Vivado 則是將 IP 拿來做接線 block design 的應用產生給 software 的 interface 使用。

在 vitis_hls, High Level Synthesis 當中主要分成三個階段:

[Vitis_hls]

C sim (c-simulation):

只要是在模擬 C++ code 的演算法的正確性。

C synthesis:

將 C++的 code 進行合成 RTL 硬體語言。

Co-sim (C/RTL co-simulation):

比對 C-simulation 跟 RTL-simulation 的模擬結果是否一致。

[Vivado]

選擇 FPGA 開發平台將 vitis_hls 產生出來的 IP 進行 block design 接線。最終生成.bit & .hwh，並可以在真實的 FPGA 版上進行部屬應用(在這次 lab 是使用 jupyter notebook 做應用，並使用 pynq.Overlay 的方式來呼叫 IP)。

What is observed & learned

在這次 lab 是我第一次接觸 vivado & vitis 的 tool，所以當然我學到了一些基本 tool 使用的方式以及 HLS 的概念。

我也觀察到了在 python 當中使用 overlay 的方式去對 hardware 去做讀寫的方式:

```
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)
```

regIP 則相當於 interface 的概念。
那為什麼是去把 i, j 寫到 0x10, 0x18 的位置呢?
我就去找，發現在 csynth.rpt 有些地方值得觀察。

```
* 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   |
+---------------+------------------+--------+-------+--------+-----------------------------+------------------------+
```

```
================================================================
== SW I/O Information
================================================================
* Top Function Arguments
+------------+-----------+----------+
| Argument   | Direction | Datatype |
+------------+-----------+----------+
| n32In1     | in        | int      |
| n32In2     | in        | int      |
| pn32ResOut | out       | int*     |
+------------+-----------+----------+

* SW-to-HW Mapping
+------------+---------------+----------+------------------------------------------+
| Argument   | HW Interface  | HW Type  | HW Info                                  |
+------------+---------------+----------+------------------------------------------+
| n32In1     | s_axi_control | register | name=n32In1 offset=0x10 range=32         |
| n32In2     | s_axi_control | register | name=n32In2 offset=0x18 range=32         |
| pn32ResOut | s_axi_control | register | name=pn32ResOut offset=0x20 range=32     |
| pn32ResOut | s_axi_control | register | name=pn32ResOut_ctrl offset=0x24 range=32 |
+------------+---------------+----------+------------------------------------------+
```

可以發現到我們是用 axi_lite 的 protocol 將 n32In1 & n32In2 的值寫到這兩個位置的 register 上面，再從 0x20 的地方讀出來。而這也符合 C++ code 的變數名稱:

```
void multip_2num(int32_t n32In1, int32_t n32In2, int32_t* pn32ResOut)
{
#pragma HLS INTERFACE s_axilite port=pn32ResOut
#pragma HLS INTERFACE s_axilite port=n32In2
#pragma HLS INTERFACE s_axilite port=n32In1
#pragma HLS INTERFACE ap_ctrl_none port=return

    *pn32ResOut = n32In1 * n32In2;

    return;
}
```

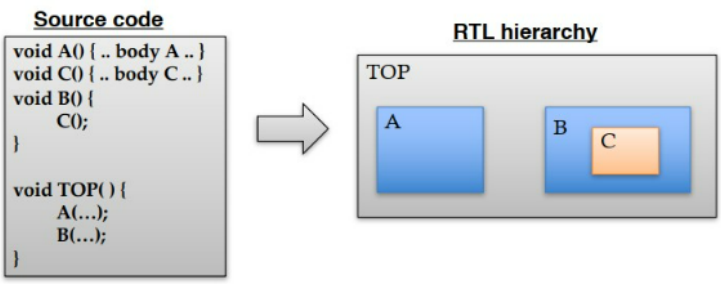
此外 pragma 是什麼?
Pragma 其實是 directives 的一種，也就對 code 進行一些限制可以用 directive.tcl 的方式，也可用 inline pragma 寫在 source code 裡面都是一樣的意思。

例如你想要你的 C++在某一些部分進行 unroll 或是 pipeline 的方式去實踐硬體，都是透過 pragma 或是下 directive 所能達到的。

至於 HLS 是怎麼樣的一個思維，我們可以用下面的突來想像他對應的 RTL 架構:

C/C++	硬件
函数	模块(module)
参数	输入/输出端口(port)
算子	函数单元
标量	线(wire)或寄存器
数组	内存(memory)
控制流	控制逻辑

通常情况下RTL代码/硬件模块层次与原始C/C++代码层次一致。



Screen dump

<u>Performance:</u>

+ Performance & Resource Estimates:

PS: '+' for module; 'o' for loop; '*' for dataflow

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
+ multip_2num	-	0.39	3	30.000	-	4	-	no	-	3 (1%)	409 (~0%)	307 (~0%)	-

== Performance Estimates

+ Timing:

* Summary:

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	6.912 ns	2.70 ns

+ Latency:

* Summary:

Latency (cycles)		Latency (absolute)		Interval		Pipeline
min	max	min	max	min	max	Type
3	3	30.000 ns	30.000 ns	4	4	no

+ Detail:

* Instance:

N/A

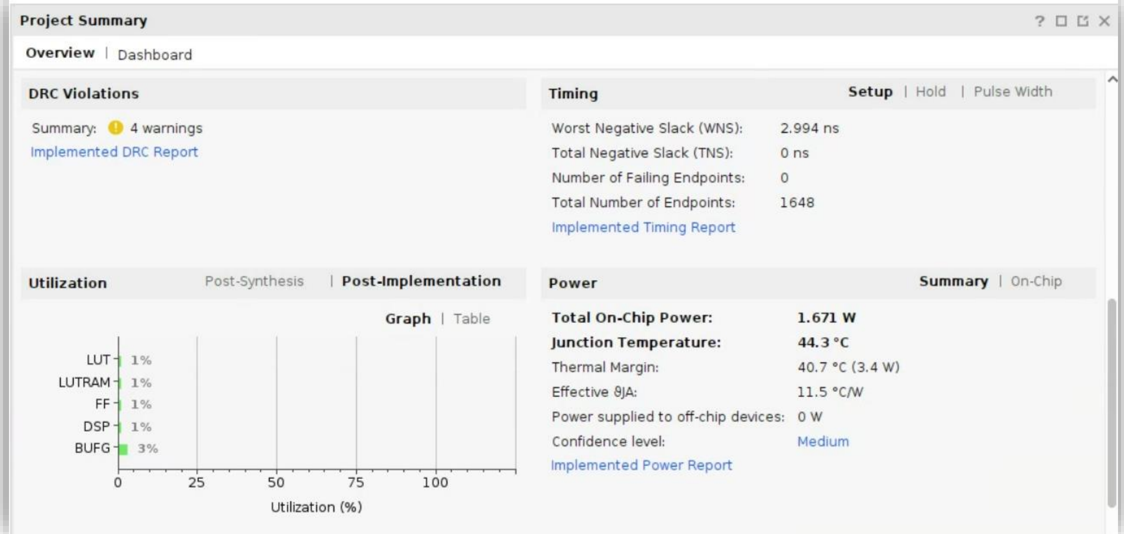
* Loop:

N/A

Interface:

```
================================================================
== Interface
================================================================
* Summary:
+-----------------------+-----+-----+------------+--------------+--------------+
|       RTL Ports       | Dir | Bits|  Protocol  | Source Object|    C Type    |
+-----------------------+-----+-----+------------+--------------+--------------+
|s_axi_control_AWVALID  |   in|    1|       s_axi|       control|       pointer|
|s_axi_control_AWREADY  |  out|    1|       s_axi|       control|       pointer|
|s_axi_control_AWADDR   |   in|    6|       s_axi|       control|       pointer|
|s_axi_control_WVALID   |   in|    1|       s_axi|       control|       pointer|
|s_axi_control_WREADY   |  out|    1|       s_axi|       control|       pointer|
|s_axi_control_WDATA    |   in|   32|       s_axi|       control|       pointer|
|s_axi_control_WSTRB    |   in|    4|       s_axi|       control|       pointer|
|s_axi_control_ARVALID  |   in|    1|       s_axi|       control|       pointer|
|s_axi_control_ARREADY  |  out|    1|       s_axi|       control|       pointer|
|s_axi_control_ARADDR   |   in|    6|       s_axi|       control|       pointer|
|s_axi_control_RVALID   |  out|    1|       s_axi|       control|       pointer|
|s_axi_control_RREADY   |   in|    1|       s_axi|       control|       pointer|
|s_axi_control_RDATA    |  out|   32|       s_axi|       control|       pointer|
|s_axi_control_RRESP    |  out|    2|       s_axi|       control|       pointer|
|s_axi_control_BVALID   |  out|    1|       s_axi|       control|       pointer|
|s_axi_control_BREADY   |   in|    1|       s_axi|       control|       pointer|
|s_axi_control_BRESP    |  out|    2|       s_axi|       control|       pointer|
|ap_clk                 |   in|    1|  ap_ctrl_hs|   multip_2num|  return value|
|ap_rst_n               |   in|    1|  ap_ctrl_hs|   multip_2num|  return value|
|ap_start               |   in|    1|  ap_ctrl_hs|   multip_2num|  return value|
|ap_done                |  out|    1|  ap_ctrl_hs|   multip_2num|  return value|
|ap_idle                |  out|    1|  ap_ctrl_hs|   multip_2num|  return value|
|ap_ready               |  out|    1|  ap_ctrl_hs|   multip_2num|  return value|
+-----------------------+-----+-----+------------+--------------+--------------+
```

Utilization:

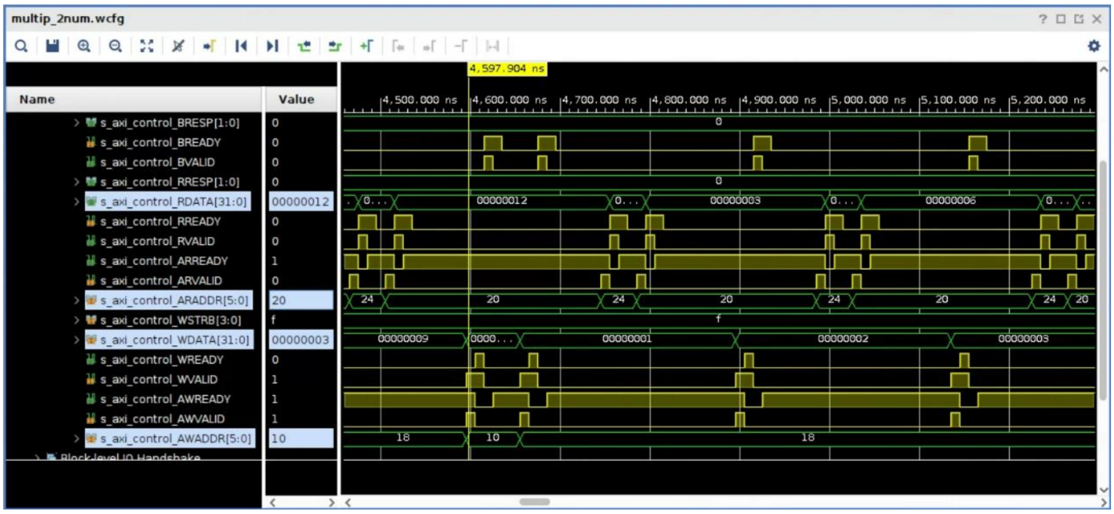


== Utilization Estimates

* Summary:

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	0	3	309	282	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	25	-
Register	-	-	100	-	-
Total	0	3	409	307	0
Available	280	220	106400	53200	0
Utilization (%)	0	1	~0	~0	0

Co-simulation transcript/waveform:



Jupyter notebook execution results:

```python
# 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
```