

# HLS Lab A- Interface Synthesis

D10943004 林亮昕

## ● Lab 1: Function Return and Block-level Protocols

下方是一個有兩個加法運算的 c-code

```
int adders(int in1, int in2, int in3) {  
    int sum;  
    sum = in1 + in2 + in3;  
    return sum;  
}
```

利用 Vitis HLS 將上方的 c-code 進行合成後觀察到以下結果:

### A. Show the default block-level, port-level protocol table

在不加任何 pragma 的情況下，預設的 block-level protocol 是 ap\_ctrl\_hs，ap\_ctrl\_hs 除了 data port (in1, in2, in3)外還包含四個控制的 port (1-bit):

Table 4-1: Block Level I/O Protocol ap\_ctrl\_hs

Signals	Description
ap_start	<p>This signal controls the block execution and must be asserted to logic 1 for the design to begin operation.</p> <p>It should be held at logic 1 until the associated output handshake ap_ready is asserted. When ap_ready goes high, the decision can be made on whether to keep ap_start asserted and perform another transaction or set ap_start to logic 0 and allow the design to halt at the end of the current transaction.</p> <p>If ap_start is asserted low before ap_ready is high, the design might not have read all input ports and might stall operation on the next input read.</p>
ap_ready	<p>This output signal indicates when the design is ready for new inputs.</p> <p>The ap_ready signal is set to logic 1 when the design is ready to accept new inputs, indicating that all input reads for this transaction have been completed.</p> <p>If the design has no pipelined operations, new reads are not performed until the next transaction starts.</p> <p>This signal is used to make a decision on when to apply new values to the inputs ports and whether to start a new transaction should using the ap_start input signal.</p> <p>If the ap_start signal is not asserted high, this signal goes low when the design completes all operations in the current transaction.</p>
ap_done	<p>This signal indicates when the design has completed all operations in the current transaction.</p> <p>A logic 1 on this output indicates the design has completed all operations in this transaction. Because this is the end of the transaction, a logic 1 on this signal also indicates the data on the ap_return port is valid.</p> <p>Not all functions have a function return argument and hence not all RTL designs have an ap_return port.</p>
ap_idle	<p>This signal indicates if the design is operating or idle (no operation).</p> <p>The idle state is indicated by logic 1 on this output port. This signal is asserted low once the design starts operating.</p> <p>This signal is asserted high when the design completes operation and no further operations are performed.</p>

此外如果執行的 cycle>1 還會有 ap\_clk 和 ap\_rst 兩個 port。

而預設的 port-level protocol 則是 ap\_none。ap\_none 只保留 data port。

B. How to specify the block-level protocol?

利用`#pragma` 指令可以指定 block-level protocol ，下方的指令是一個例子將 block-level protocol 指定為 `ap_ctrl_none`

```
#pragma HLS INTERFACE mode=ap_ctrl_none port=return
```

C. Show Interface table, and cross-reference signals and corresponding protocol

`ap_ctrl_hs` 的 interface 如下

```

=====
== Interface
=====
* Summary:

```

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
<code>ap_start</code>	in	1	<code>ap_ctrl_hs</code>	adders	return value
<code>ap_done</code>	out	1	<code>ap_ctrl_hs</code>	adders	return value
<code>ap_idle</code>	out	1	<code>ap_ctrl_hs</code>	adders	return value
<code>ap_ready</code>	out	1	<code>ap_ctrl_hs</code>	adders	return value
<code>ap_return</code>	out	32	<code>ap_ctrl_hs</code>	adders	return value
<code>in1</code>	in	32	<code>ap_none</code>	<code>in1</code>	scalar
<code>in2</code>	in	32	<code>ap_none</code>	<code>in2</code>	scalar
<code>in3</code>	in	32	<code>ap_none</code>	<code>in3</code>	scalar

```

=====

```

`ap_ctrl_none` 的 interface 如下

```

=====
== Interface
=====
* Summary:

```

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
<code>in1</code>	in	32	<code>ap_none</code>	<code>in1</code>	scalar
<code>in2</code>	in	32	<code>ap_none</code>	<code>in2</code>	scalar
<code>in3</code>	in	32	<code>ap_none</code>	<code>in3</code>	scalar
<code>ap_return</code>	out	32	<code>ap_ctrl_none</code>	adders	return value

```

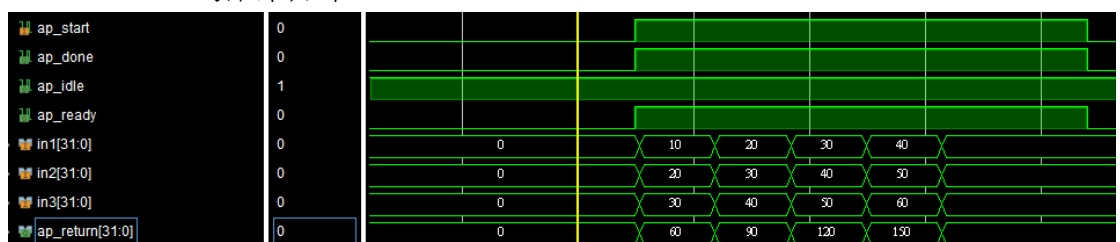
=====

```

兩者最大的差別在於 `ap_ctrl_none` 只保留 data port

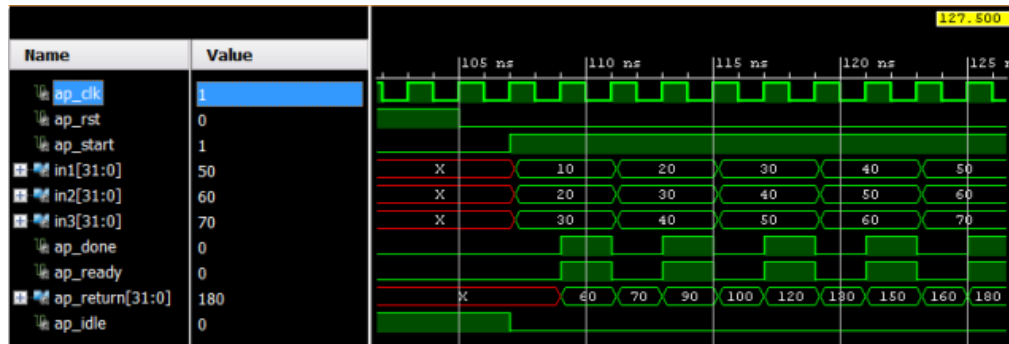
D. Show co-simulation waveform, explain the `ap_ctrl_hs` interface behavior\

co-simulation 的結果如下:



由於 c-code 中兩個加法的運算很簡單，可以在一個 cycle 內做完，此時合成的電路會是 combinational。當 ap\_start 為 1 時，電路會開始運算，ap\_ready 為 1 時代表電路可以接受新的輸入，執行完畢後 ap\_done 會變成 1，ap\_idle 則代表此時電路是否有在運算(由於是 combinational 所以 ap\_idle 一直是 1)。

當合成的電路 cycle>1 時，其波形應該如下：



#### E. Use ap\_ctrl\_none -> Cosim failures

ap\_ctrl\_none 可以進行 cosimulation 的情況有三個：

```
@E [SIM-345] Cosim only supports the following 'ap_ctrl_none' designs: (1)
combinational designs; (2) pipelined design with task interval of 1; (3) designs with
array streaming or hls_stream ports.
@E [SIM-4] *** C/RTL co-simulation finished: FAIL ***
```

由於兩個加法的運算很簡單，可以在一個 cycle 內做完(符合上方第一格條件)，此時合成的電路會是 combinational，因此 ap\_ctrl\_none 不會有 error。但當運算的複雜度提升如：

```
int adders(int in1, int in2, int in3) {
#pragma HLS INTERFACE mode=ap_ctrl_none port=return
    int sum;
    sum = in1*10023452345 + in2 + in3;
    return sum;
}
```

此時 HLS 的結果無法在一個 cycle 內完成：

```
+ Latency:
  * Summary:
  +-----+-----+-----+-----+-----+-----+
  | Latency (cycles) | Latency (absolute) | Interval | Pipeline |
  | min | max | min | max | min | max | Type |
  +-----+-----+-----+-----+-----+-----+
  | 5 | 5 | 16.250 ns | 16.250 ns | 6 | 6 | no |
  +-----+-----+-----+-----+-----+-----+
```

這時用 ap\_ctrl\_none 就無法成功

```
ERROR: [COSIM 212-540] Code only supports the following 'ap_ctrl_none' designs: (1) combinational designs; (2) pipelined design with task interval of 1; (3) designs with array streaming or hls_stream or AXI4 stream ports.
ERROR: [COSIM 212-3] *** C/RTL co-simulation file generation failed. ***
ERROR: [COSIM 212-4] *** C/RTL co-simulation finished: FAIL ***
```

- Lab 2: Port I/O Protocols

與 Lab1 類似，Lab2 的 c-code 一樣有兩個加法運算，但是用 pointer 的形式描述

```
void adders_io(int in1, int in2, int *in_out1) {
#pragma HLS INTERFACE mode=ap_hs port=in_out1
#pragma HLS INTERFACE mode=ap_vld port=in2
#pragma HLS INTERFACE mode=ap_vld port=in1

    *in_out1 = in1 + in2 + *in_out1;
}
```

利用 Vitis HLS 將上方的 c-code 進行合成後觀察到以下結果:

A. List all the port-level protocol from Vitis HLS (2022.1) manual

所有的 port-level protocol 如下:

1. ap\_none: 只有 data port
2. ap\_hs: 包含 ap\_ack, ap\_vld, ap\_ovld 三個控制訊號
3. ap\_memory, bram: 接到記憶體의 I/O
4. ap\_fifo: 接到 FIFO 的 I/O
5. axis, m\_axi, s\_axilite: 用於 AXI stream, AXI lite, 與 AXI

B. Show the interface table & waveform to explain the signal behavior

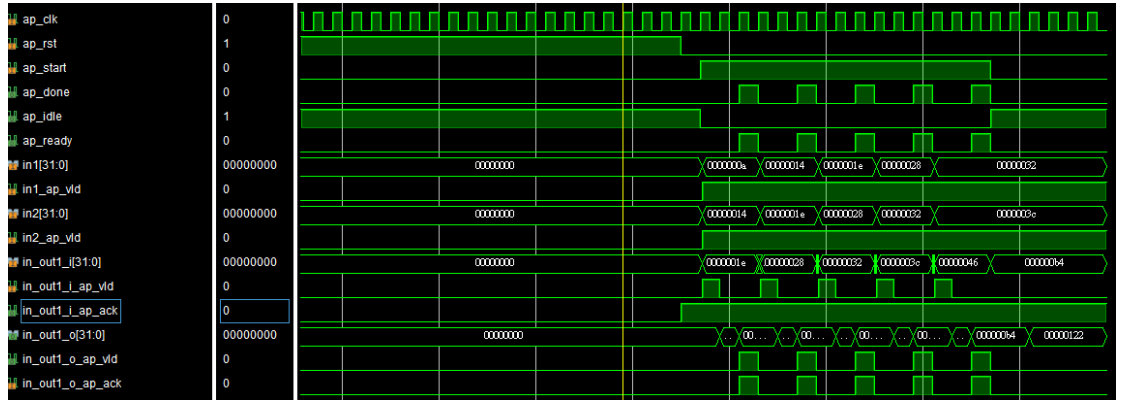
Interface table 如下:

```
=====
== Interface
=====
* Summary:
```

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	adders_io	return value
ap_rst	in	1	ap_ctrl_hs	adders_io	return value
ap_start	in	1	ap_ctrl_hs	adders_io	return value
ap_done	out	1	ap_ctrl_hs	adders_io	return value
ap_idle	out	1	ap_ctrl_hs	adders_io	return value
ap_ready	out	1	ap_ctrl_hs	adders_io	return value
in1	in	32	ap_vld	in1	scalar
in1_ap_vld	in	1	ap_vld	in1	scalar
in2	in	32	ap_vld	in2	scalar
in2_ap_vld	in	1	ap_vld	in2	scalar
in_out1_i	in	32	ap_hs	in_out1	pointer
in_out1_i_ap_vld	in	1	ap_hs	in_out1	pointer
in_out1_i_ap_ack	out	1	ap_hs	in_out1	pointer
in_out1_o	out	32	ap_hs	in_out1	pointer
in_out1_o_ap_vld	out	1	ap_hs	in_out1	pointer
in_out1_o_ap_ack	in	1	ap_hs	in_out1	pointer

```
=====
```

波形如下



在 block-level 的部分，當 ap\_start 為 1 時，電路開始進行運算，當每筆測資計算完時 ap\_done 會為 1，此時 ap\_ready 也會為 1，代表可以接受下一筆測資，每筆測資的運算時間為兩個 cycle，當最後運算都結束時 ap\_idle 會回到 1。

在 port-level 的部分，in1\_ap\_vld/in2\_ap\_vld/in\_out1\_i\_ap\_vld 為 1 代表 in1/in2/in\_out1 的輸入資料已經準備好，in\_out\_o\_ap\_vld 為 1 代表冊茲已經運算完準備由 in\_out1 的進行輸出。

另外，in\_out\_i\_ap\_ack/in\_out\_o\_ap\_ack 則是代表運算電路/測試軟體已經準備好要接收資料。

### ● Lab 3: Implementing Arrays as RTL Interfaces

Lab3 的 c-code 是利用 for loop 在 8 個 channel 上分別作累加運算，每個 channel 各有 4 筆 data，故總共有 32 筆輸入與 32 筆輸出。

```
// The data comes in organized in a single array.
// - The first sample for the first channel (CHAN)
// - Then the first sample for the 2nd channel etc.
// The channels are accumulated independently
// E.g. For 8 channels:
// Array Order : 0 1 2 3 4 5 6 7 8 9 10 etc. 16 etc...
// Sample Order: A0 B0 C0 D0 E0 F0 G0 H0 A1 B1 C2 etc. A2 etc...
// Output Order: A0 B0 C0 D0 E0 F0 G0 H0 A0+A1 B0+B1 C0+C2 etc. A0+A1+A2 etc...

void array_io (dout_t d_o[N], din_t d_i[N]) {
    int i, rem;

    // Store accumulated data
    static dacc_t acc[CHANNELS];
    dacc_t temp;

    // Accumulate each channel
    For_Loop: for (i=0;i<N;i++) {
        rem=i%CHANNELS;
        temp = acc[rem] + d_i[i];
        acc[rem] = temp;
        d_o[i] = acc[rem];
    }
}
```

利用 Vitis HLS 將上方的 c-code 進行合成後觀察到以下結果：

- A. Rolled loop, use dual-port RAM. What does the synthesis report show?  
 在沒有 unroll 且沒有 pipeline 的情況下，指定 I/O 使用 dual-port 與 single-port RAM 並沒有差別，最後 HLS 都會讓 interface 為 single-port RAM，因為每個 iteration 需要兩個 cycle 來讀取：

Loop Name	Latency (cycles)		Iteration Latency	Initiation achieved	Interval target	Trip Count	Pipelined
	min	max					
- For_Loop	64	64	2	-	-	32	no

而在沒有 unroll 但有 pipeline 的情況下，指定 I/O 使用 dual-port 與 single-port RAM 同樣沒有差別，最後 HLS 都會讓 I/O 為 dual-port RAM 來實現同時讀寫，因此每個 iteration 只需要一個 cycle：

Loop Name	Latency (cycles)		Iteration Latency	Initiation achieved	Interval target	Trip Count	Pipelined
	min	max					
- For_Loop	32	32	2	1	1	32	yes

- B. Unrolled the loop, compare the latency for the cases of combination of input(single/dual port), output (single/dual port), explain why?  
 由於 unroll 的關係，只有在 input port 與 output port 同時是 dual port 的時候才有辦法實現 factor=2 的 unrolling，此時 cycle=17 大約是原本 cycle=33 的一半。值得注意的是 dual port 有分 ram\_2p、ram\_s2p、ram\_t2p。ram\_2p 支援兩讀或一寫一讀，ram\_s2p 支援一寫一讀，ram\_t2p 則支援兩讀、一寫一讀、與兩寫。故 output 的 dual port 形式需設為 ram\_t2p 才能有以上 unrolling 的效果。
- C. Unroll & array\_partition with different type = block/cyclic/complete, factor = 2, 4. Observe latency, resource used and explain why
- 在 block type 的 array partition 下，factor=4 的 cycle 和使用的硬體資源都比 factor=2 來得少上不少。這是因為 c-code 中是以四個數字為一組進行累加，當 factor=2 時僅能一次得到兩個數字，必須用額外的 registers 將這兩個數字(或其相加的結果)紀錄起來等到同組的另外兩個數字出現才能進行輸出，這中間的時間大約差了 8 個 cycle，故 factor=2 需要的 cycle 為 25 個 cycle( $\approx 32/2+8$ )，factor=4 的 cycle=10( $\approx 32/4$ )多上不少。

2. 在 cyclic type 的 array partition 下，factor=4 的 cycle 比 factor=2 來得少上不少，但使用的硬體資源則是差不多。這是因為在 cyclic type 的 array partition 下，資料是交錯進來的，故 factor=4 只需要 9 個 cycle( $\approx 32/4$ )但 factor=2 需要 17 個 cycle( $\approx 32/2$ )。
3. 在 complete type 的 array partition 下，所有的資料會同時進來進行運算，雖然只需要 1 個 cycle 且所需的硬體資源較 cyclic type 和 block type 來得少，但所需的 IO bandwidth 卻相當的大。

總結來說，factor=4 的 array partition 在 cyclic type 與 block type 下都能得到不錯的結果，這是由於資料的模式所得到的結果，因此根據不同資料模式選擇不同的 partition 方法可以得到相當程度的優化。

## ● Lab 4: Implementing AXI4 Interfaces

Lab4 的 c-code 基本上與 Lab3 相同:

```
void axi_interfaces (dout_t d_o[N], din_t d_i[N]) {
    int i, rem;

    // Store accumulated data
    static dacc_t acc[CHANNELS];

    // Accumulate each channel
    For_Loop: for (i=0; i<N; i++) {
        rem=i%CHANNELS;
        acc[rem] = acc[rem] + d_i[i];
        d_o[i] = acc[rem];
    }
}
```

利用 Vitis HLS 將上方的 c-code 進行合成後觀察到以下結果:

- A. AXI Stream: Unroll the loop, and observe how many axis channel created. Compare the area with Lab1-3.  
為了避免在 stream 下產生 data dependency 造成 cycle 上的浪費，AXI channel 的數量與 c-code 中 channel 的數量需要一致=8，因此只需要 4( $=32/8$ )個 cycle，所需要的硬體資源與 Lab3 中以 cyclic type 進行 array partition 的方式差不多。
- B. AXI Lite: It is used to communicate with hos program. Show \_hw.h and explain its content  
透過將 block level protocol 設置為 AXI Lite 可以讓外部的 host program 透過 AXI Lite 的 protocol 去控制完成的 IP(但 data port 還是透過 AXI Stream)。IP 在控制層面主要分成 interrupt 和 control signals 兩種，透過 AXI Lite protocol 將這些訊號由不同的 registers 中的位元代表，如下方 xaxi\_interfaces\_hw.h 中所示:

```

// control
// 0x0 : Control signals
//   bit 0 - ap_start (Read/Write/COH)
//   bit 1 - ap_done (Read/COR)
//   bit 2 - ap_idle (Read)
//   bit 3 - ap_ready (Read/COR)
//   bit 7 - auto_restart (Read/Write)
//   bit 9 - interrupt (Read)
//   others - reserved
// 0x4 : Global Interrupt Enable Register
//   bit 0 - Global Interrupt Enable (Read/Write)
//   others - reserved
// 0x8 : IP Interrupt Enable Register (Read/Write)
//   bit 0 - enable ap_done interrupt (Read/Write)
//   bit 1 - enable ap_ready interrupt (Read/Write)
//   others - reserved
// 0xc : IP Interrupt Status Register (Read/COR)
//   bit 0 - ap_done (Read/COR)
//   bit 1 - ap_ready (Read/COR)
//   others - reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

#define XAXI_INTERFACES_CONTROL_ADDR_AP_CTRL 0x0
#define XAXI_INTERFACES_CONTROL_ADDR_GIE   0x4
#define XAXI_INTERFACES_CONTROL_ADDR_IER   0x8
#define XAXI_INTERFACES_CONTROL_ADDR_ISR   0xc

```

例如當位址為 0x0 的 register 中的第 0 個 bit 為 1 時，該 IP 的 ap\_start 為 1，因此 IP 會開始運作。