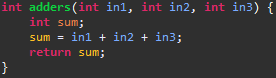
HLS Lab A- Interface Synthesis

D10943004 林亮昕

* Lab 1: Function Return and Block-level Protocols

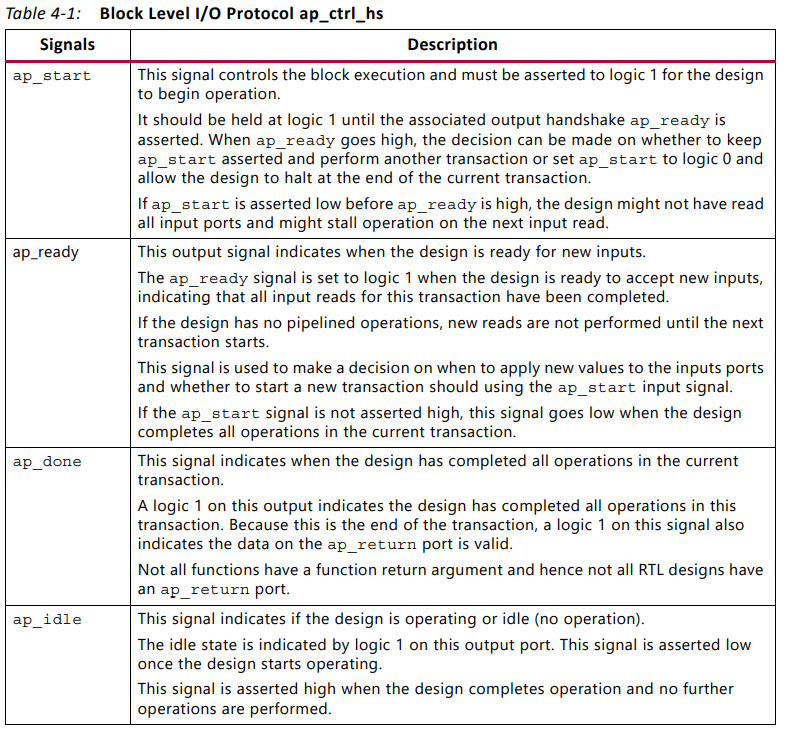
下方是一個有兩個加法運算的c-code



利用Vitis HLS將上方的c-code進行合成後觀察到以下結果:

1. 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):



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。

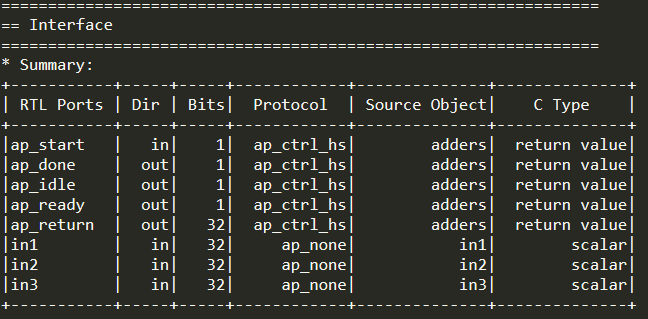
1. How to specify the block-level protocol?

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

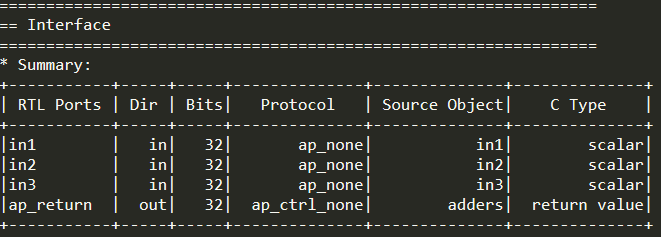


1. Show Interface table, and cross-reference signals and corresponding protocol

ap\_ctrl\_hs的interface如下



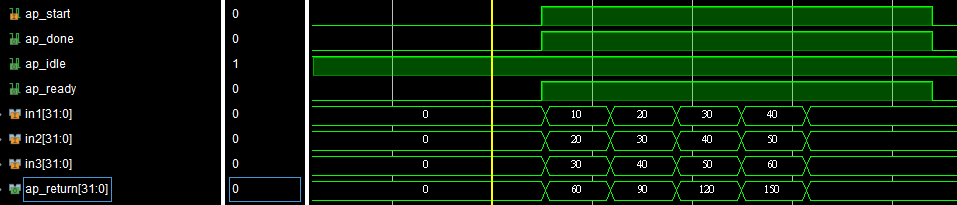
ap\_ctrl\_none的interface如下



兩者最大的差別在於ap\_ctrl\_none只保留data port

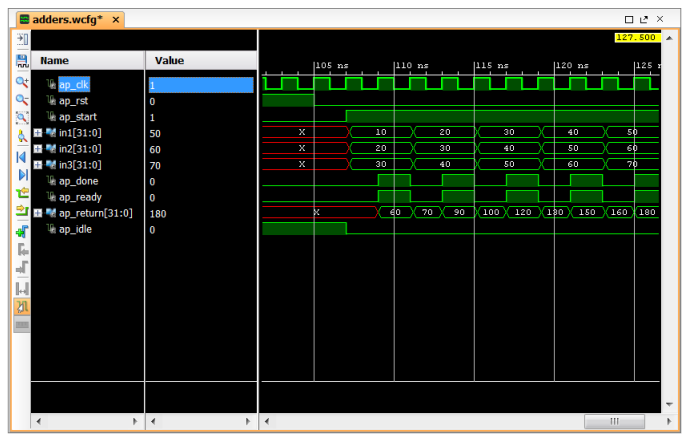
1. Show co-simulation waveform, explain the ap\_ctrl\_hs interface behavior\

co-simulation的結果如下:



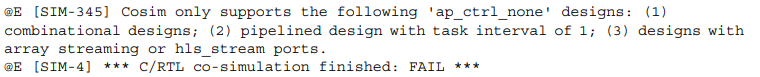
由於c-code中兩個加法的運算很簡單，可以在一個cycle內做完，此時合成的電路會是combinational。當ap\_start為1時，電路會開始運算，ap\_ready為1時代表電路可以接受新的輸入，執行完畢後ap\_done會變成1，ap\_idle則代表此時電路是否有在運算(由於是combinational所以ap\_idle一直是1)。

當合成的電路cycle>1時，其波形應該如下:

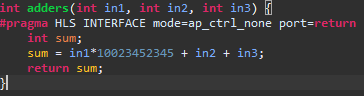


1. Use ap\_ctrl\_none -> Cosim failures

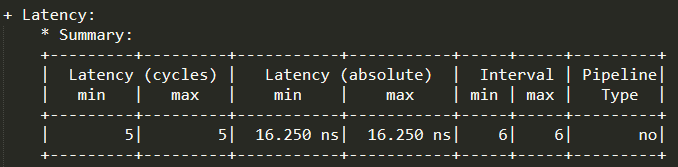
ap\_ctrl\_none可以進行cosimulation的情況有三個:



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:



此時HLS的結果無法在一個cycle內完成:

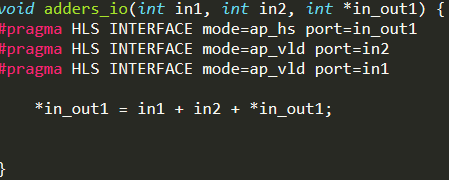


這時用ap\_ctrl\_none就無法成功



* 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



利用Vitis HLS將上方的c-code進行合成後觀察到以下結果:

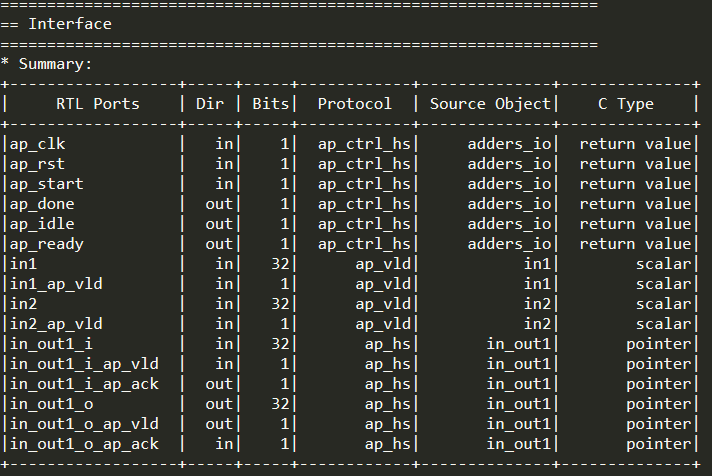
1. List all the port-level protocol from Vitis HLS (2022.1) manual

所有的port-level protocol如下:

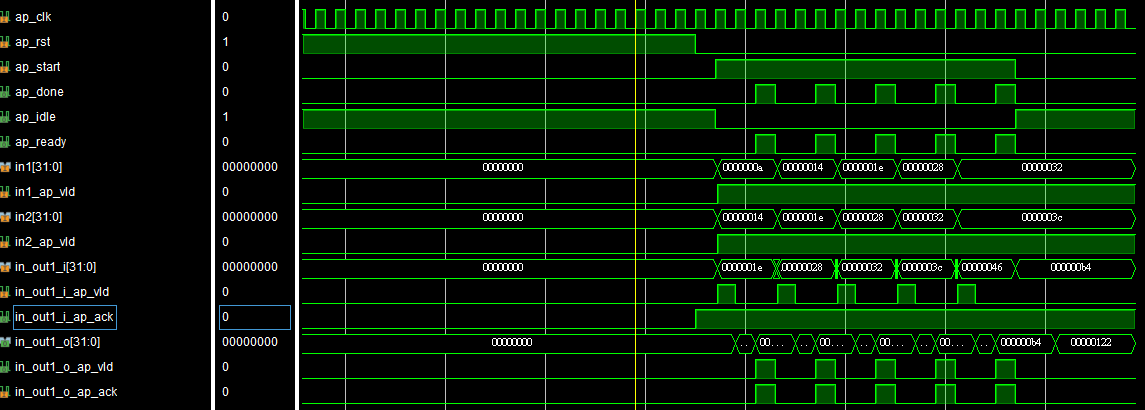
* 1. ap\_none: 只有data port
  2. ap\_hs: 包含ap\_ack, ap\_vld, ap\_ovld三個控制訊號
  3. ap\_memory, bram: 接到記憶體的I/O
  4. ap\_fifio: 接到FIFO的I/O
  5. axis, m\_axi, s\_axilite: 用於AXI stream, AXI lite, 與AXI

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

Interface table如下:



波形如下



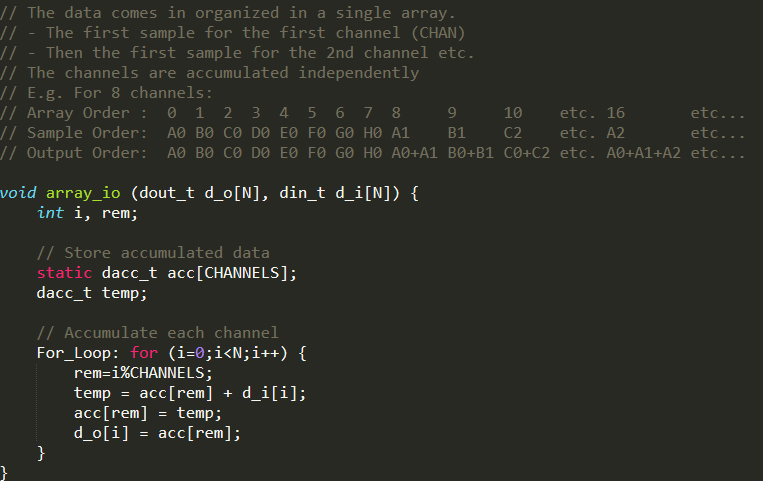
在block-level的部分，當ap\_start為1時，電路開始進行運算，當每筆測資計算完時ap\_done會為1，此時ap\_ready也會為1，代表可以接受下一筆測資，每筆測資的運算時間為兩個cycle，當最後運算都結束時ap\_idle會回到1。

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

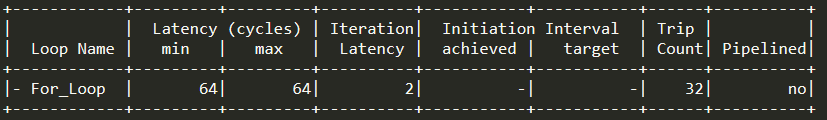
Lab3的c-code是利用for loop在8個channel上分別作累加運算，每個channel各有4筆data，故總共有32筆輸入與32筆輸出。



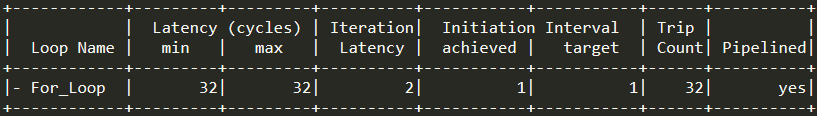
利用Vitis HLS將上方的c-code進行合成後觀察到以下結果:

1. 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:



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:



1. Unrolled the loop, compare the latency for the cases of combination of input(single/dual port), output (single/dual port), explain why?

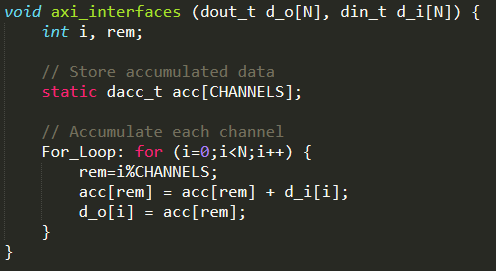
由於unroll的關係，只有在input port與output port同時是dual port的時候才有辦法實現factor=2的unrolling，此時cycle=17大約是原本cycle=33的一半。值得注意的是dual port有分ram\_2p、ram\_s2p、ram\_t2p。ram\_2p支援兩讀或一寫一讀，ram\_s2p支援一寫一讀，ram\_t2p則支援兩讀、一寫一讀、與兩寫。故output的dual port形式需設為ram\_t2p才能有以上unrolling的效果。

1. Unroll & array\_partition with different type = block/cyclic/complete, factor = 2, 4. Observe latency, resource used and explain why
2. 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(32/2+8)，factor=4的cycle=10(32/4)多上不少。
3. 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(32/4)但factor=2需要17個cycle(32/2)。
4. 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:



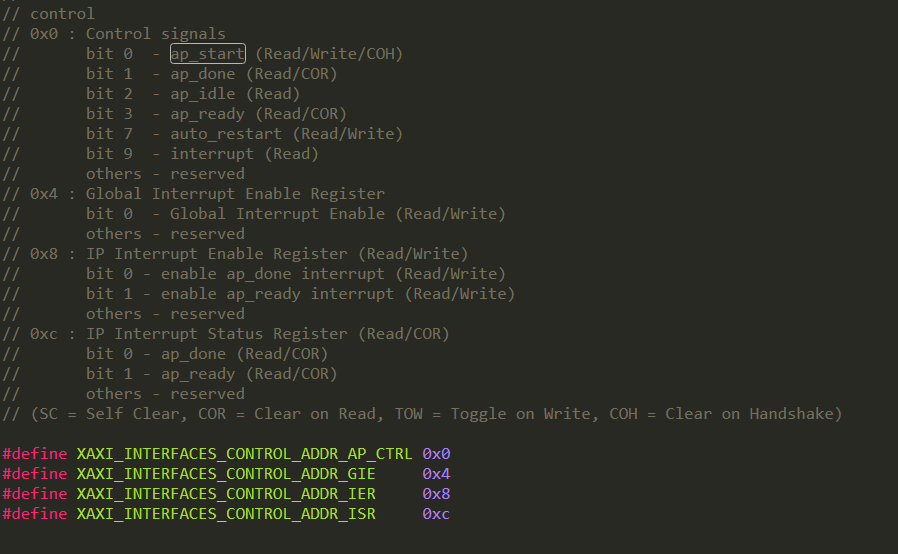
利用Vitis HLS將上方的c-code進行合成後觀察到以下結果:

1. 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。

1. 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:



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。