

LAB A Design Optimization

Lab 1 Optimization a Matrix Multiplier

@單純不做 Pipeline 情況之 Matrix Multiplier

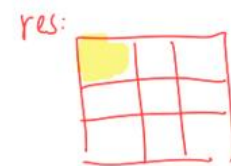
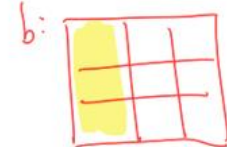
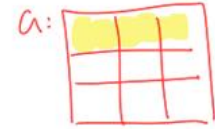
```

*****
#include "matrixmul.h"

void matrixmul(
    mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
    mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
    result_t res[MAT_A_ROWS][MAT_B_COLS])
{
    // Iterate over the rows of the A matrix
    Row: for(int i = 0; i < MAT_A_ROWS; i++) {
        // Iterate over the columns of the B matrix
        Col: for(int j = 0; j < MAT_B_COLS; j++) {
            res[i][j] = 0;
            // Do the inner product of a row of A and col of B
            Product: for(int k = 0; k < MAT_B_ROWS; k++) {
                res[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}

```

Product: for loop:



我們可以看到，每次進行運算的時候，product 迴圈內需要做三次運算。

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	13.33	3.576	1.67

Latency (clock cycles)

Summary

Latency	Interval	
min	max	min
79	79	79

Detail

Instance

Loop

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- Row	78	78	26	-	-	3	no
+ Col	24	24	8	-	-	3	no
++ Product	6	6	2	-	-	3	no

Loop
↓
Loop
 $(24+1+1) \times 3 = 78$

1 CLK 進入 Loop
↓
 $24 = 8 \times 3$
 $6 = 2 \times 3$
6+1+1
↓
1 CLK 出 Loop

Figure 7-4: Synthesis Report for the Matrix Multiplier

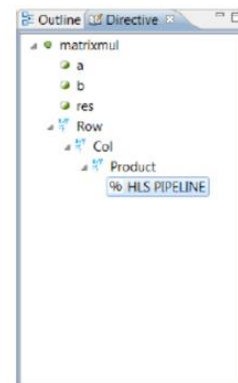
故在不做 pipeline 的情況下，我們可以得知，其各自運算所需要的時間，包含最內層 2 個 cycle 並且 Trip Count = 3，代表重複三次此迴圈，並且在 Col loop 的計算下，需要考慮進入 for loop 與出去 for loop 的部分，各自需要 1 個 clk，代表一次 COL loop 需要 $(6+1+1)=8$ ，並且 trip Count = 3，故需要 $8 \times 3 = 24$ ，即可計算出如同上圖的 Latency。

@ 將 Pipeline 加置於最內層之 Loop

將 Pipeline 加在最後內層 Loop

```
*****
#include "matrixmul.h"

void matrixmul(
    mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
    mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
    result_t res[MAT_A_ROWS][MAT_B_COLS])
{
    // Iterate over the rows of the A matrix
    Row: for(int i = 0; i < MAT_A_ROWS; i++) {
        // Iterate over the columns of the B matrix
        Col: for(int j = 0; j < MAT_B_COLS; j++) {
            res[i][j] = 0;
            // Do the inner product of a row of A and col of B
            Product: for(int k = 0; k < MAT_B_ROWS; k++) {
                res[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```



如果將 Pipeline pragma 放置於最內層，則有可能造成下列相關問題。

```
INFO: [XFORM 203-541] Flattening a loop nest 'Row' (matrixmul.cpp:54:37) in function
'matrixmul'.
...
INFO: [SCHED 204-61] Pipelining loop 'Product'.
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint (II = 1,
distance = 1)
    between 'store' operation (matrixmul.cpp:60) of variable 'tmp_8', matrixmul.cpp:60
on array 'res' and 'load' operation ('res_load', matrixmul.cpp:60) on array 'res'.
INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 2, Depth: 2.
```

Performance Estimates

Timing (ns)

Summary			
Clock	Target	Estimated	Uncertainty
ap_clk	13.33	4.306	1.67

Latency (clock cycles)

Summary				
Latency		Interval		
min	max	min	max	Type
82	82	82	82	none

Detail

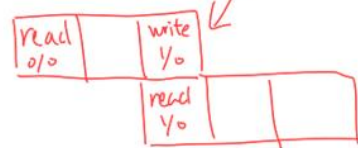
Instance

Loop

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- Row_Col	81	81	9	-	-	9	no
+ Product	6	6	2	2	1	3	yes

① "data dependency 造成無法 II=1"

② res[i][j] = 0 導致無法全部 flatten

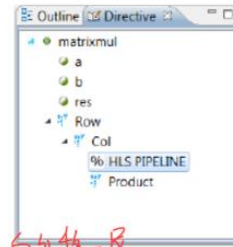


也就是，此時，因為 data dependency 造成的 II 不為 1 情況，以及因為 res = 0 而導致的無法全數 flatten。

如此的情況下，data dependency 能夠靠著 Pipeline 更外層的 for loop 或者靠著 data reshape/partition 來解決此情況。

@ Pipeline the Col loop

```
*****  
#include "matrixmul.h"  
  
void matrixmul(  
    mat_a_t a[MAT_A_ROWS][MAT_A_COLS],  
    mat_b_t b[MAT_B_ROWS][MAT_B_COLS],  
    result_t res[MAT_A_ROWS][MAT_B_COLS])  
{  
    // Iterate over the rows of the A matrix  
    Row: for(int i = 0; i < MAT_A_ROWS; i++) {  
        // Iterate over the columns of the B matrix  
        Col: for(int j = 0; j < MAT_B_COLS; j++) {  
            res[i][j] = 0;  
            // Do the inner product of a row of A and col of B  
            Product: for(int k = 0; k < MAT_B_ROWS; k++) {  
                res[i][j] += a[i][k] * b[k][j];  
            }  
        }  
    }  
}
```

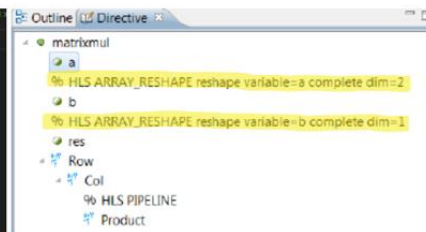


Pipeline 向外移一層
但確保了 data dependency
不會發生

這樣的作法，確保了 data dependency 的不發生，但卻產生另一問題，也就是 Block RAM 最多僅能讀寫 2 次，但在此處讀取了 3 次，這樣的情況將會觸發 limited memory ports。而這件事情可以透過 Reshape 與 Partition 來進行解決。

@ Reshape the Array

```
*****  
#include "matrixmul.h"  
  
void matrixmul(  
    mat_a_t a[MAT_A_ROWS][MAT_A_COLS],  
    mat_b_t b[MAT_B_ROWS][MAT_B_COLS],  
    result_t res[MAT_A_ROWS][MAT_B_COLS])  
{  
    // Iterate over the rows of the A matrix  
    Row: for(int i = 0; i < MAT_A_ROWS; i++) {  
        // Iterate over the columns of the B matrix  
        Col: for(int j = 0; j < MAT_B_COLS; j++) {  
            res[i][j] = 0;  
            // Do the inner product of a row of A and col of B  
            Product: for(int k = 0; k < MAT_B_ROWS; k++) {  
                res[i][j] += a[i][k] * b[k][j];  
            }  
        }  
    }  
}
```

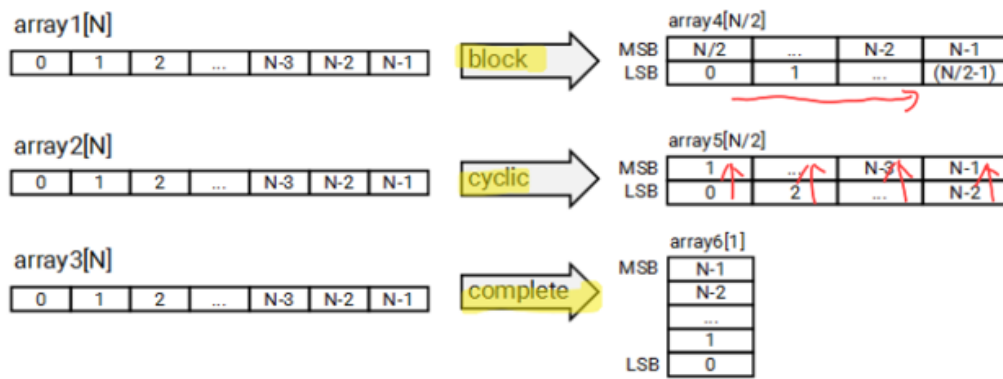


這邊藉由 Reshape the Array 來協助處理 limited memory ports。

@reshape the array aprgma

```
void foo (...) {
  int array1[N];
  int array2[N];
  int array3[N];
  #pragma HLS ARRAY_RESHAPE variable=array1 block factor=2 dim=1
  #pragma HLS ARRAY_RESHAPE variable=array2 cycle factor=2 dim=1
  #pragma HLS ARRAY_RESHAPE variable=array3 complete dim=1
  ...
}
```

The `ARRAY_RESHAPE` pragma transforms the arrays into the form shown in the following figure.
Figure: `ARRAY_RESHAPE` Pragma



X14807-110217

Reshape the pragma 分成 block cyclic，complete 來進行矩陣的分切，各自以不同方式進行分切，block 會將矩陣分割為連續記憶體，按照輸入順序與維度進行分切，而 cyclic 則會以循環的方式，將原本的順序打散自依照維度分切的記憶體。而 Complete 將會將記憶體全部切成各自獨立的小塊，最消耗資源，但也最容易解決 data dependency 的問題。

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	13.33	7.566	1.67

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
11	11	11	11	none

Detail

Instance

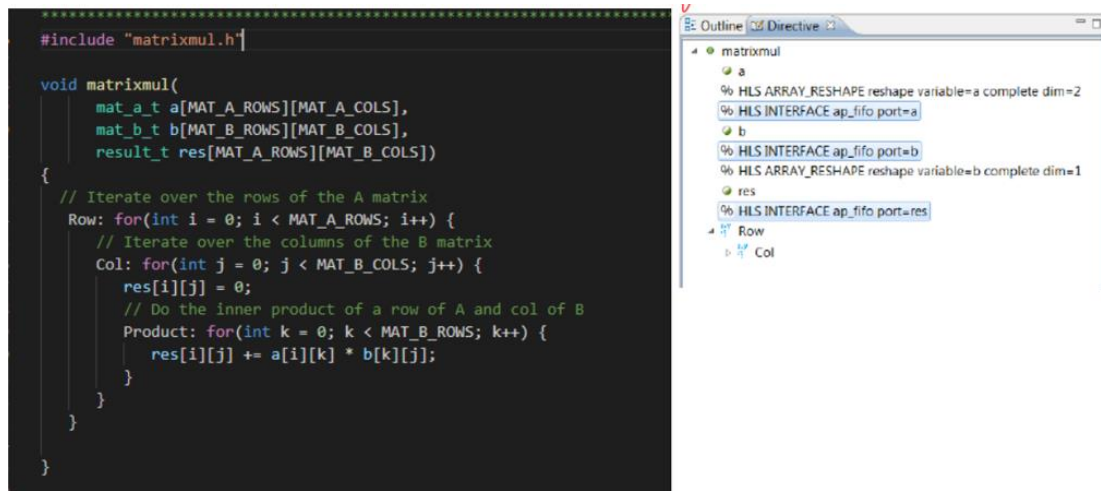
Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_Col	9	9	2	1	1	9	yes

$$(1 \times 9 + 1 + 1) = 11$$

這邊可以看到透過 reshape array 來達成整體全部 pipeline 的效果，總共只需要 latency = 11 即可完成全部運算。

@Apply FIFO to interfaces



```
#include "matrixmul.h"

void matrixmul(
    mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
    mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
    result_t res[MAT_A_ROWS][MAT_B_COLS])
{
    // Iterate over the rows of the A matrix
    Row: for(int i = 0; i < MAT_A_ROWS; i++) {
        // Iterate over the columns of the B matrix
        Col: for(int j = 0; j < MAT_B_COLS; j++) {
            res[i][j] = 0;
            // Do the inner product of a row of A and col of B
            Product: for(int k = 0; k < MAT_B_ROWS; k++) {
                res[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

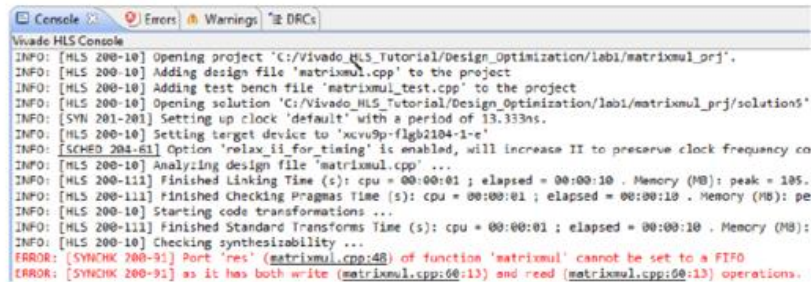
Outline Directive

- matrixmul
 - a
 - % HLS ARRAY_RESHAPE reshape variable=a complete dim=2
 - % HLS INTERFACE ap_fifo port=a
 - b
 - % HLS INTERFACE ap_fifo port=b
 - % HLS ARRAY_RESHAPE reshape variable=b complete dim=1
 - res
 - % HLS INTERFACE ap_fifo port=res
 - Row
 - Col

如果要將 FIFO 應用於資料上，則我們可以將 data，以 stream 形式流動，這也就是為什麼要宣告為 FIFO 的型態。

因存取時為對連續的同一地址進行4次寫入。
即 $[0][0] \rightarrow [0][0] \rightarrow [0][0] \rightarrow [0][0] \rightarrow [0][0] \dots$
即：
$$\begin{cases} res[0][0] = res[0][0] + a[0][0] + b[0][0] \\ res[0][1] = res[0][1] + a[0][1] + b[0][1] \\ \vdots \end{cases}$$

這並非 FIFO 的數據流動方式，此為隨機存取



```
Vivado HLS Console
INFO: [HLS 200-10] Opening project 'C:/Vivado_HLS_Tutorial/Design_Optimization/lab1/matrixmul_prj'.
INFO: [HLS 200-10] Adding design file 'matrixmul.cpp' to the project
INFO: [HLS 200-10] Adding test bench file 'matrixmul_test.cpp' to the project
INFO: [HLS 200-10] Opening solution 'C:/Vivado_HLS_Tutorial/Design_Optimization/lab1/matrixmul_prj/solutions'
INFO: [SYN 201-201] Setting up clock 'default' with a period of 13.333ns.
INFO: [HLS 200-10] Setting target device to 'xcvu9p-flgb2104-1-e'
INFO: [SCHED 204-61] Option 'relax_ii_for_timing' is enabled, will increase II to preserve clock frequency co
INFO: [HLS 200-10] Analyzing design file 'matrixmul.cpp' ...
INFO: [HLS 200-111] Finished Linking Time (s): cpu = 00:00:01 ; elapsed = 00:00:10 . Memory (MB): peak = 185.
INFO: [HLS 200-111] Finished Checking Pragmas Time (s): cpu = 00:00:01 ; elapsed = 00:00:10 . Memory (MB): pe
INFO: [HLS 200-10] Starting code transformations ...
INFO: [HLS 200-111] Finished Standard Transforms Time (s): cpu = 00:00:01 ; elapsed = 00:00:10 . Memory (MB):
INFO: [HLS 200-10] Checking synthesizability ...
ERROR: [SYNCHK 200-91] Port 'res' (matrixmul.cpp:48) of function 'matrixmul' cannot be set to a FIFO
ERROR: [SYNCHK 200-91] as it has both write (matrixmul.cpp:60:13) and read (matrixmul.cpp:60:13) operations.
```

然而，同時對於多個資料進行讀取的動作，這件事情是違反了 FIFO 的標準，如果想要使用 FIFO 作為 Input，Output 則需遵守其規則，在本篇 LAB 當中，將於 LAB2 當中重新詮釋如何正確的使用 FIFO。

@ Pipeline all the function

```
#include "matrixmul.h"

void matrixmul(
    mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
    mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
    result_t res[MAT_A_ROWS][MAT_B_COLS])
{
    // Iterate over the rows of the A matrix
    Row: for(int i = 0; i < MAT_A_ROWS; i++) {
        // Iterate over the columns of the B matrix
        Col: for(int j = 0; j < MAT_B_COLS; j++) {
            res[i][j] = 0;
            // Do the inner product of a row of A and col of B
            Product: for(int k = 0; k < MAT_B_ROWS; k++) {
                res[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

若是對於整體的 function 進行 PipeLine，將會有可能導致，其消耗過多資源，儘管能做到最短時間完成，但付出的硬體資源需要謹慎思考是否必要。

solution4為Reshapearray後(無 FIFO) 所消耗之資源

solution6 為全部 Pipeline 所消耗之資源

FF, LUT → 明顯消耗增多

但 Solution6 之 Latency 與 Interval 亦明顯減少

Timing (ns)			
Clock		solution4	solution6
ap_clk	Target	13.33	13.33
	Estimated	7.566	7.566

Latency (clock cycles)			
		solution4	solution6
Latency	min	11	5
	max	11	5
Interval	min	11	5
	max	11	5

Utilization Estimates		
	solution4	solution6
BRAM_18K	0	0
DSP48E	2	18
FF	18	343
LUT	187	565
URAM	0	0

```
INFO: [XFORM 203-502] Unrolling all loops for pipelining in function 'matrixmul'
(matrixmul.cpp:49).INFO: [HLS 200-489] Unrolling loop 'Row' (matrixmul.cpp:54) in
function 'matrixmul' completely with a factor of 3.

INFO: [HLS 200-489] Unrolling loop 'Col' (matrixmul.cpp:56) in function 'matrixmul'
completely with a factor of 3.

INFO: [HLS 200-489] Unrolling loop 'Product' (matrixmul.cpp:59) in function
'matrixmul' completely with a factor of 3.
```

如上圖所示。

@LAB 2 C Code Optimized for I/O Access

```
#include "matrixmul.h"

void matrixmul(
    mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
    mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
    result_t res[MAT_A_ROWS][MAT_B_COLS])
{
    #pragma HLS ARRAY_RESHAPE variable=b complete dim=1
    #pragma HLS ARRAY_RESHAPE variable=a complete dim=2
    #pragma HLS INTERFACE ap_fifo port=a
    #pragma HLS INTERFACE ap_fifo port=b
    #pragma HLS INTERFACE ap_fifo port=res

    mat_a_t a_row[MAT_A_ROWS];
    mat_b_t b_copy[MAT_B_ROWS][MAT_B_COLS];
    int tmp = 0;

    // Iterate over the rows of the A matrix
    Row: for(int i = 0; i < MAT_A_ROWS; i++) {
        // Iterate over the columns of the B matrix
        Col: for(int j = 0; j < MAT_B_COLS; j++) {
            #pragma HLS PIPELINE rewind
            // Do the inner product of a row of A and col of B
            tmp = 0;
            // Cache each row (so it's only read once per function)
            if (j == 0)
                Cache_Row: for(int k = 0; k < MAT_A_ROWS; k++)
                    a_row[k] = a[i][k];
            // Cache all cols (so they are only read once per function)
            if (i == 0)
                Cache_Col: for(int k = 0; k < MAT_B_ROWS; k++)
                    b_copy[k][j] = b[k][j];
            Product: for(int k = 0; k < MAT_B_ROWS; k++) {
                tmp += a_row[k] * b_copy[k][j];
            }
            res[i][j] = tmp;
        }
    }
}
```

input a → input b
output res
b 延著維度 1 reshape
a 延著維度 2 reshape
因為在運用時 i, k, j
是 k 維度在一起運算
所以延著「共享的維度」是 reshape 的最佳解

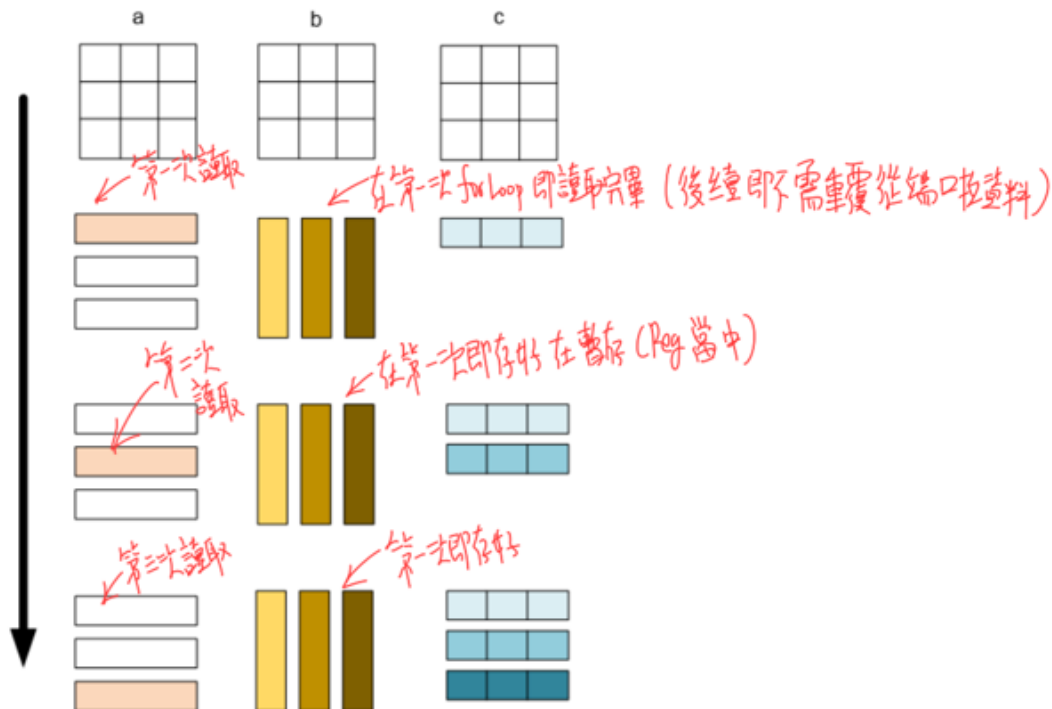
No1 Loop	No2 Loop	No3 Loop
$\bar{i}=0$	$\bar{i}=1$	$\bar{i}=2$
$j=0$	$j=0$	$j=0$
$a[0][0]$	$a[1][0]$	$a[2][0]$
$a[0][1]$	$a[1][1]$	$a[2][1]$
$a[0][2]$	$a[1][2]$	$a[2][2]$

→ 符合 FIFO
← 第 j Loop 即讀完所有 b
← 且符合 FIFO

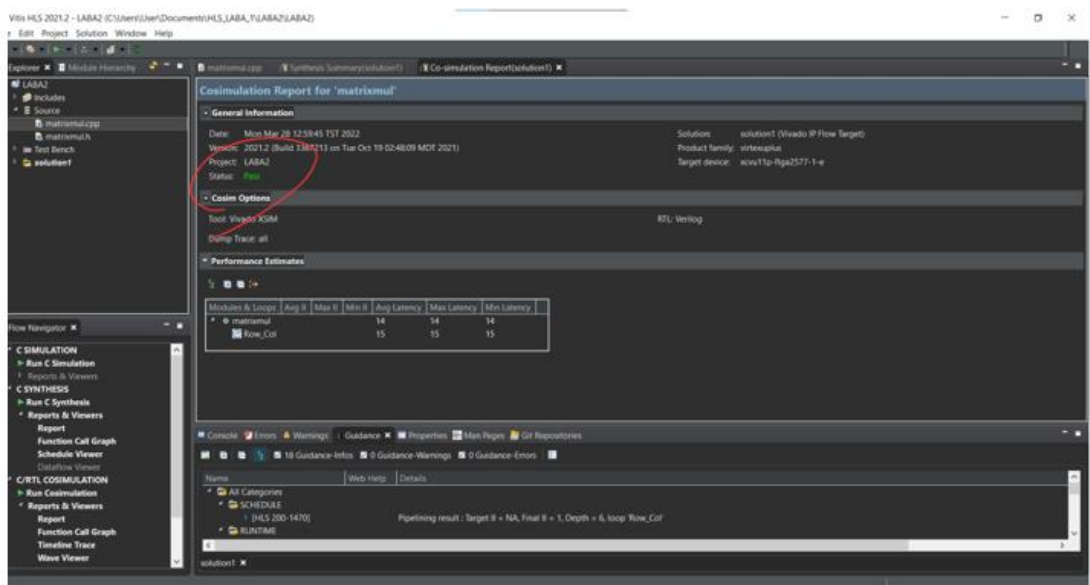
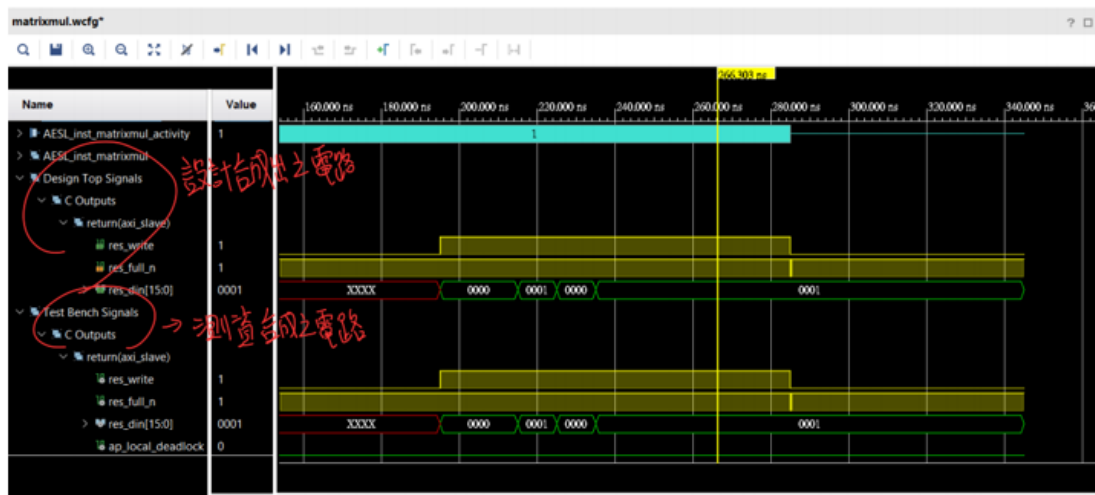
$\bar{i}=0$	$\bar{i}=1$	$\bar{i}=2$
$j=0$	$j=1$	$j=2$
$b[0][0]$	$b[0][1]$	$b[0][2]$
$b[1][0]$	$b[1][1]$	$b[1][2]$
$b[2][0]$	$b[2][1]$	$b[2][2]$

用 temp 解決
 $res[i][j] = res[i][j] + Psum$

如同上圖所示，在本次 LAB 當中，將採用 FIFO 作為 Input 與 Output，並且在輸入與輸出，均遵守 FIFO 之規則，按照順序，不可進行隨機存取。



而在本次 Code 當中，亦對於資料讀取，有較好的優化，總計資料不重複讀取許多次，B 的矩陣部分將於一開始即完全讀取完畢，並存入暫存器中，避免重複向儲存著 B 本體的記憶體進行讀取，浪費資源與時間。



從以上可以看到，輸出所合成之電路，確實與測資提供之驗算一致。

@ 簡易狀況之 Data dependency

The code snippet shows a loop where `r[i]` is updated based on its previous value, creating a true data dependency. The HLS PIPELINE is set to off.

Timing Estimate:

Target	Estimated	Uncertainty
10.00 ns	2.085 ns	2.70 ns

Performance & Resource Estimates:

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
aaa	-	-	21	210.000	-	22	-	no	0	0	15	114	0
add	-	-	20	200.000	2	-	10	no	-	-	-	-	-

Handwritten notes: "RAW truly dependency" and "不做 pipe line" (Don't do pipeline).

在較為簡單狀況的 data dependency 當中我們想要檢視，是否 Tool 會協助我們進行處理。(上圖為不做 Pipeline 的 data dependency)

The code snippet shows a loop where a temporary variable `temp` is used to store the result of `a[i] + b[i]`, which is then used to update `r[i]`. This eliminates the true data dependency. The HLS PIPELINE is set to off.

Timing Estimate:

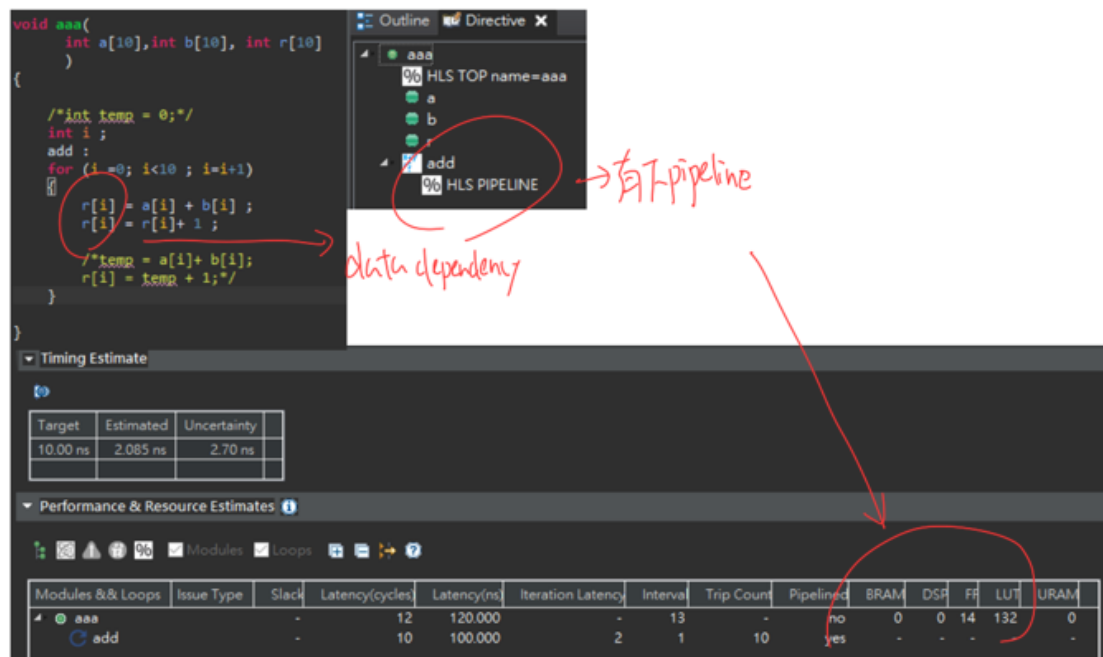
Target	Estimated	Uncertainty
10.00 ns	2.085 ns	2.70 ns

Performance & Resource Estimates:

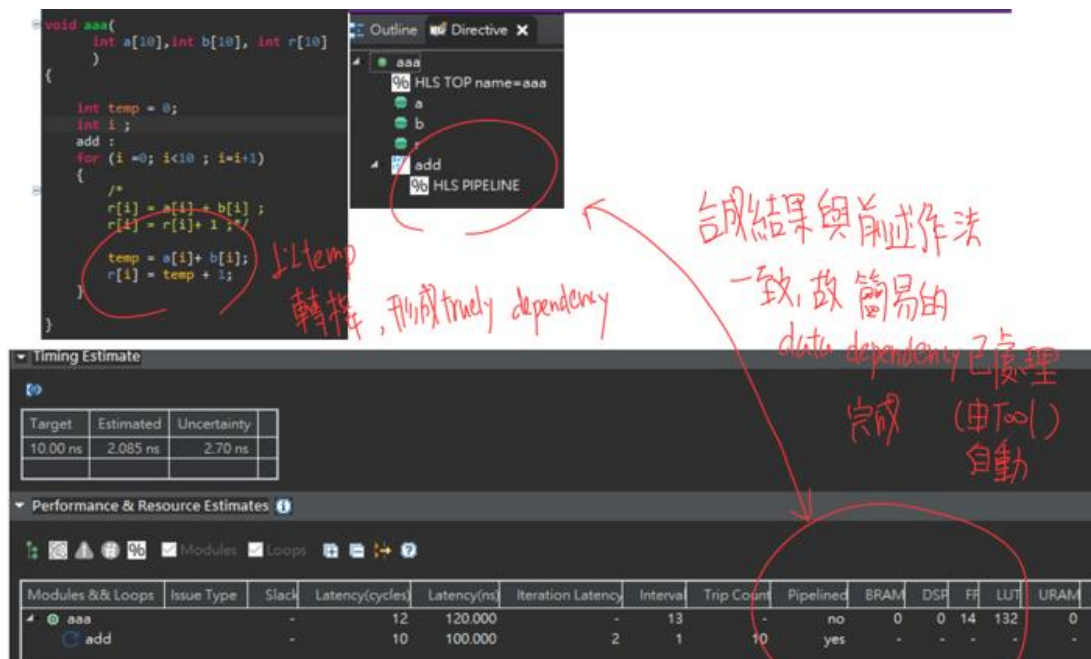
Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
aaa	-	-	21	210.000	-	22	-	no	0	0	15	114	0
add	-	-	20	200.000	2	-	10	no	-	-	-	-	-

Handwritten notes: "用temp來作, 這跟 r[i] 本身 truly dependency 但合出來相同 → 即較為簡單的数据 dependency 已經處理完成" (Using temp to do this, this is the same as the true dependency of r[i] itself, but the result is the same → that is, the simpler data dependency has been handled).

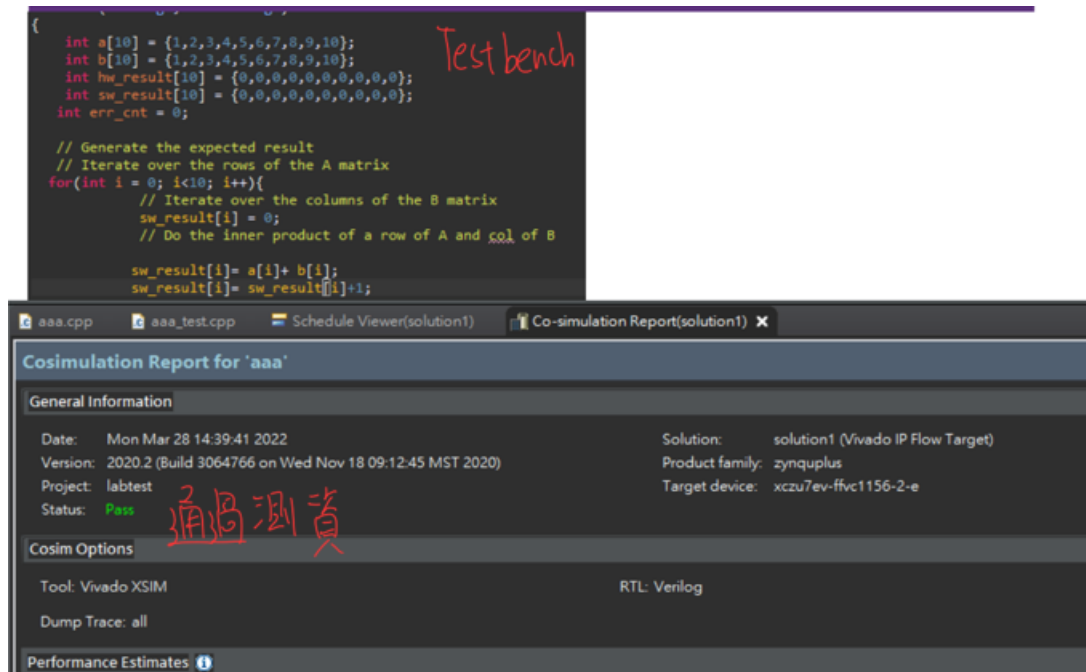
本圖為 以 temp 來使 data dependency 的情況為 True dependency，可以看到合成之情況為完全相同。



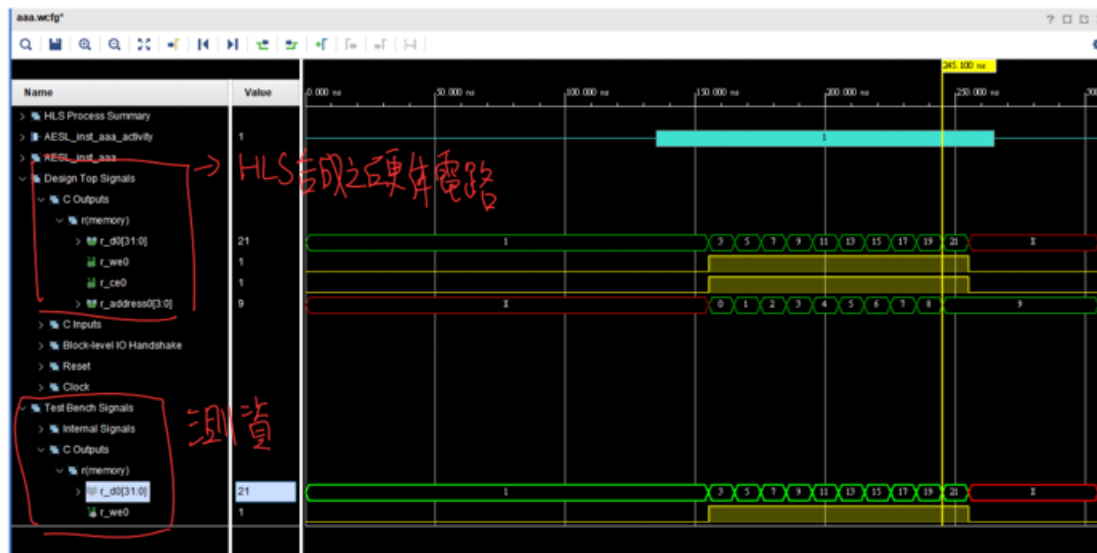
上圖為有做 PipeLine 之簡易 data dependency 情況。



上圖為有做 PipeLine 並且以 temp 製造 True dependency 之情況，可以看到均由 Tool 協助進行完成，就算為 True dependency 在較為簡易的狀況，仍可由 Tool 協助完成除錯。



將上述電路進行測資相關驗證，可以佐證，測資為通過情形。



並檢視其合成後之電路，亦可以確認，合成後之電路與測資電路輸出相符合，驗證假想: Tool 將會識別較為簡易的 Data dependency 仍可自動排除。

@較為嚴重之 data dependency

The screenshot shows a C++ code snippet with a loop 'add' containing a true data dependency. The code is as follows:

```
void aaa(  
    int a[12], int b[12], int r[12]  
)  
{  
    int temp = 0;  
    int i;  
    add :  
    for (i = 0; i < 10; i = i + 1)  
    {  
        r[i] = a[i] + b[i] + a[i+1] + a[i+2] + b[i+1] + b[i+2];  
        r[i+2] = r[i+1] + r[i] + 1;  
        //r[i] = a[i] + b[i] + a[i+1] + a[i+2] + b[i+1] + b[i+2];  
        //temp = r[i];  
        //r[i+2] = r[i+1] + temp + 1;  
    }  
}
```

The Vivado HLS interface shows the 'add' loop with a data dependency violation. The error message states: "The II Violation in module 'aaa' (loop 'add'): Unable to enforce a carried dependence constraint (II = 1, distance = 1, offset = 1) between 'store' operation ('r_addr_2_write_in15', 'r_addr_2_write_in15')."

Performance & Resource Estimates table:

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
aaa	II Violation	-	22	220.000	-	2	10	no	0	0	25	584	0
add	II Violation	-	20	200.000	-	2	2	yes	-	-	-	-	-

Handwritten notes in red: "將 a, b 切 partition 即可解決 limit memory port 之問題" and "true dependency".

將 a 與 b 切分 partition 避免前述所提到的 limit memory port 問題。但於合成的時候，因為 output r 的 carry dependency 的問題，導致無法合成出 II=1 之情況，於是亦將 r 切分為不同之記憶體，採用 array partition 並且以 complete 切分，則可以處理上述問題。

The screenshot shows the 'Cosimulation Report for 'aaa'' and the 'Performance & Resource Estimates' table. The 'Cosimulation Report' shows the following information:

- General Information: Date: Mon Mar 28 16:44:37 2022, Version: 2020.2 (Build 3064766 on Wed Nov 18 09:12:45 MST 2020), Project: labtest, Status: Pass.
- Cosim Options: Tool: Vivado XSIM, Dump Trace: all.

The 'Performance & Resource Estimates' table is as follows:

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
aaa	-	-	12	120.000	-	13	-	no	0	0	18	1015	0
add	-	-	10	100.000	-	2	1	yes	-	-	-	-	-

Handwritten notes in red: "切 complete 形式將耗費許多資源" and "將 output matrix 以 complete 形式切 Array partition".

可以看到將 r 以 array partition 以及 complete 切分，可以解決上述 II=2 之情況，但將會耗費許多資源，為了驗證合成電路是否正確，亦有執行 co-simulation，確保合成電路，與測資一致。



上圖為合成電路與測資電路之輸出，確認為一致，驗證較為嚴重的 data dependency 或許可以用 array partition 進行處理。

@ Limit memory port issue

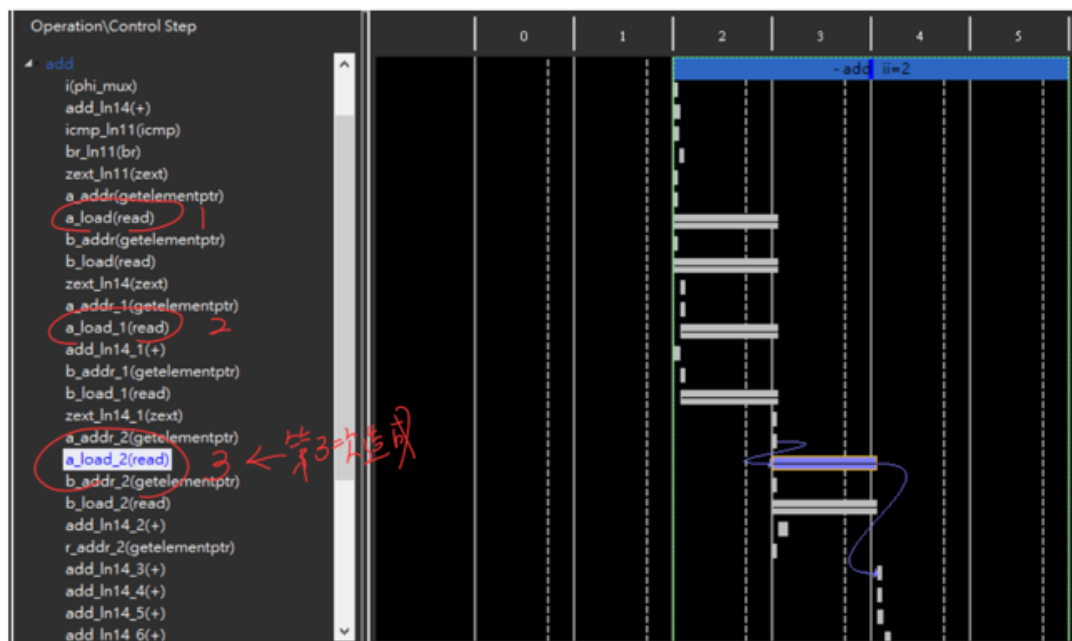
```
void aaa(
    int a[13], int b[13], int r[13]
)
{
    //int temp = 0;
    int i;
    add :
    for (i = 0; i < 11; i = i + 1)
    {
        r[i] = a[i] + b[i] + a[i+1] + a[i+2] + b[i+1] + b[i+2];
        r[i+2] = r[i+2] + r[i+1] + r[i+1] + 1;

        /*temp = a[i] + b[i];
        r[i] = temp + 1;*/
    }
}
```

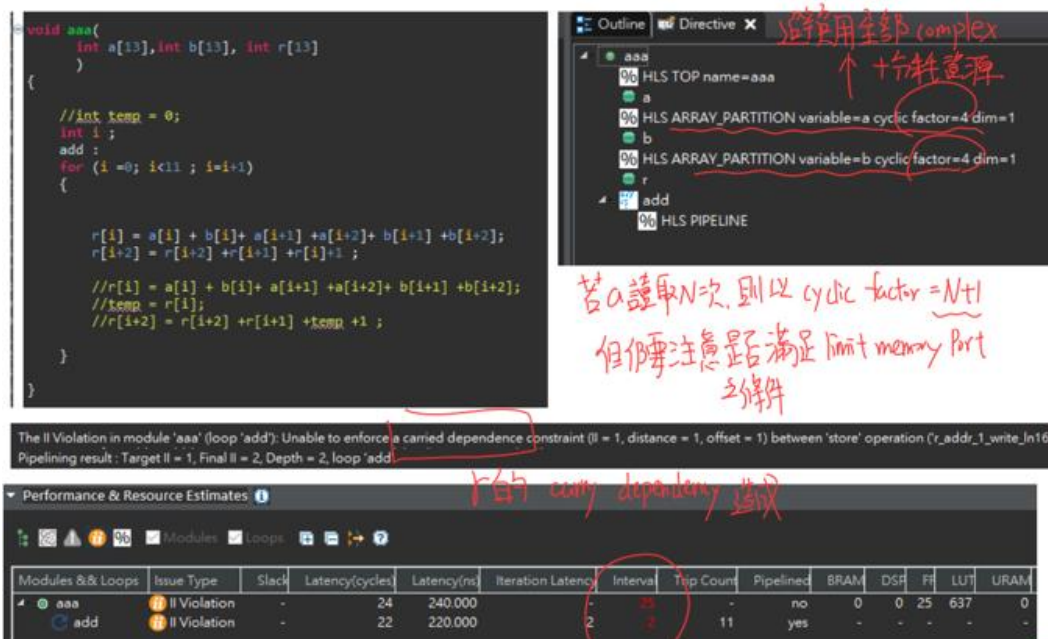
Performance & Resource Estimates

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSR	FF	LUT	URAM
aaa	II Violation	-	27	270.000	-	28	-	no	0	0	189	414	0
add	II Violation	-	24	240.000	-	2	11	yes	-	-	-	-	-

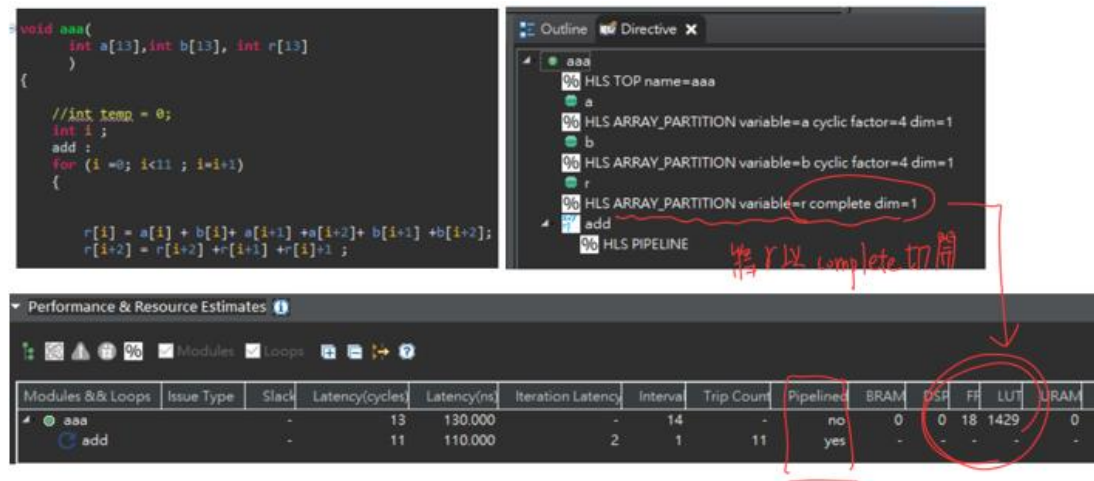
從上圖程式碼中，可以見到，確實出現 limit memory port 將可能造成的問題，也就是對於 a,b 記憶體重複進行讀取，這樣是無法在一連續記憶體當中，做到 II=2，於是就可以透過前述之技巧，以 array partition 或者 array reshape 進行相關切分，來避免此種問題。



本圖為陳述，重複第三次之讀取，將造成 limit memory port 的問題，造成 II=2 之情況。

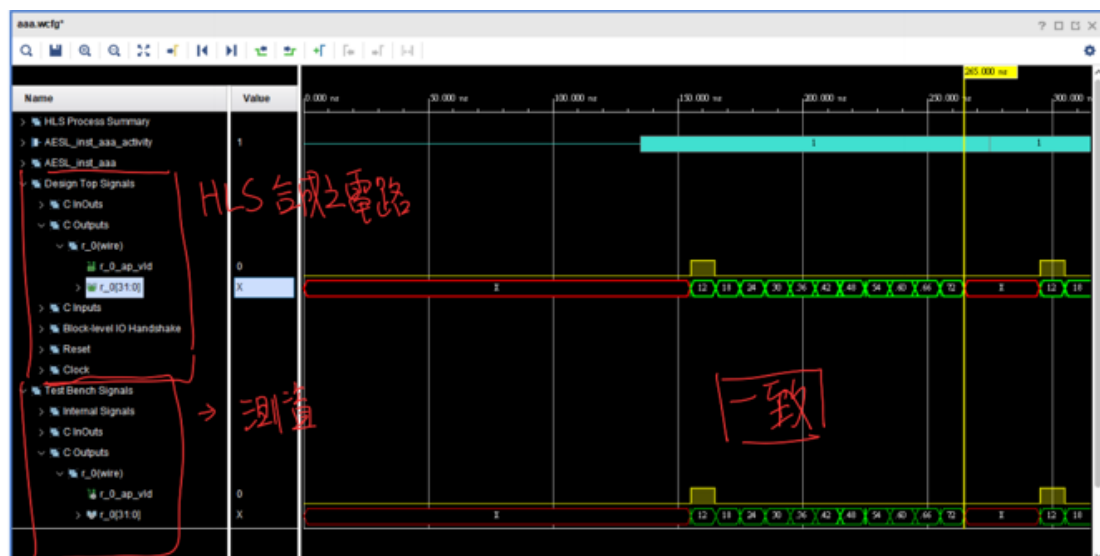


而在本篇的 Code 當中，將 a,b 以 array partition 進行切分，並且 factor 設定為 4，，盡量避免全部均使用 array partition 當中的 complete 切分。這樣將會消耗過多的硬體資源，並不是一件好事情，故需要適時使用 array reshape 與 array partition。而 output 則以前述之 array partition complete 進行切分，以避免上述之 carry dependency 問題。



將 r 以 complete 切開
使 LUT 使用量增加

本圖為驗證，使用 array partition complete 將會造成使用之 LUT 用量大增，但可以使 II=1，此點可以用於進行相關權衡。



本圖可以驗證，HLS 合成之電路，與測資之電路輸出均一致，驗證正確。

```

void aaa(
    int a[12], int b[12], int r[12]
)
{
    //int kcomp = 0;
    int i;
    add :
    for (i = 0; i < 10; i = i + 1)
    {
        r[i] = a[i] + b[i] + a[i+1] + a[i+2] + b[i+1] + b[i+2];
        r[i+2] = r[i+1] + r[i] + 1;

        //r[i] = a[i] + b[i] + a[i+1] + a[i+2] + b[i+1] + b[i+2];
        //kcomp = r[i];
        //r[i+2] = r[i+1] + kcomp + 1;
    }
}

```

```

aaa
HLS TOP name=aaa
a
b
r
add
HLS PIPELINE

```

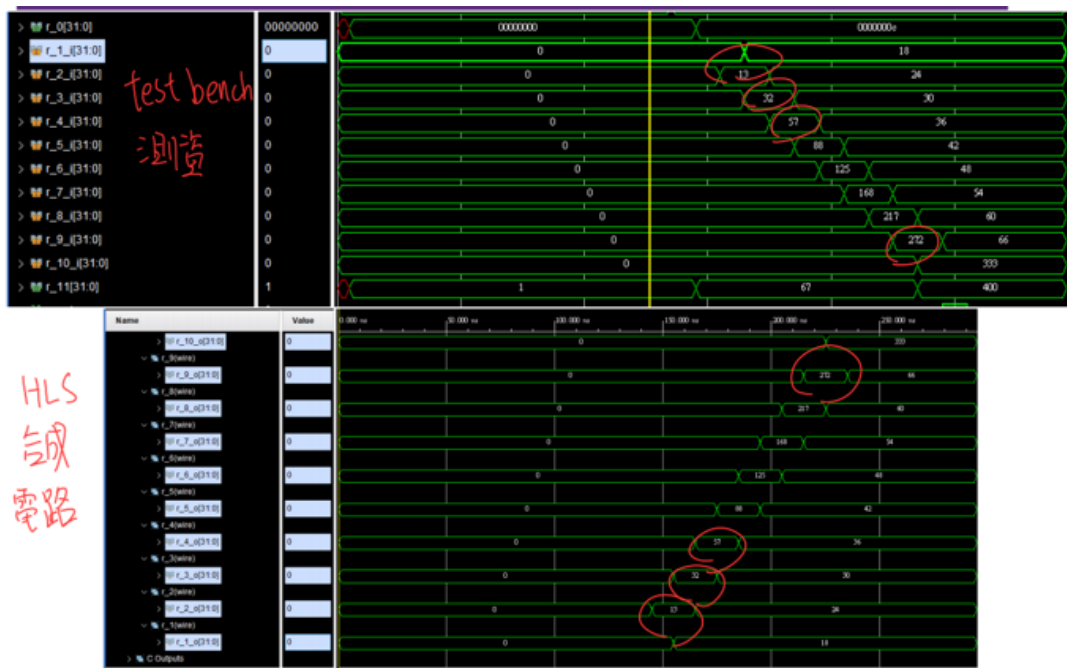
Annotations in image:
 - HLS ARRAY_PARTITION variable=a complete dim=1
 - HLS ARRAY_PARTITION variable=b complete dim=1
 - HLS ARRAY_PARTITION variable=r complete dim=1

全部都切
complete array partition
將造成使用量爆增!

Performance & Resource Estimates

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
aaa		-	-	-	-	-	-	no	0	0	647	2679	0
add		-	-	-	1	1	-	yes	-	-	-	-	-

本圖證明切分為全部 complete 將導致資源用量大增，但將會於以下輸出圖見到，輸出的 r，也將會被全部分割成獨立的狀況，可以於波形圖下方見到。

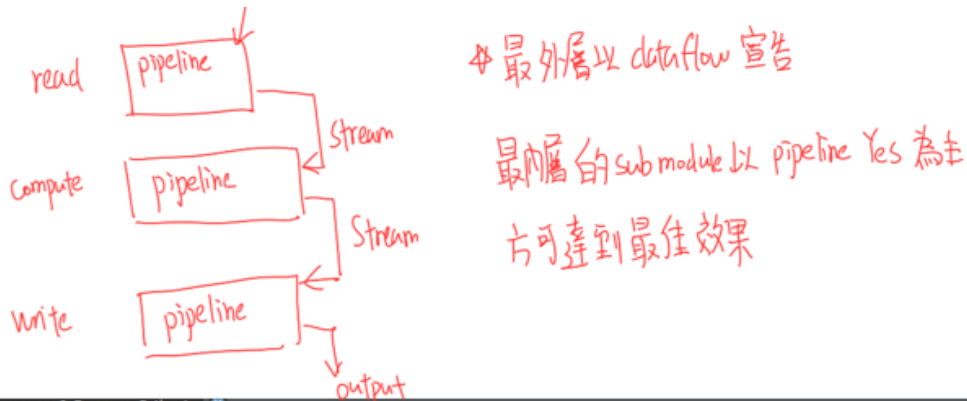


本圖可以看到 HLS 合成電路(下方)與上方測資輸出，均為一致，證明驗證正確。

@ Data Flow



本篇將會示範如何做到示範當中的較為有效的準則，以上圖為例，可以見到從 Global memory 之資料輸入後，經由 Pipeline 的各自 module 進行相關處理，並且於 module 中，以 data flow 進行流通，這樣會有較佳的效果!而我認為這也是進行 HLS 相關撰寫時，需要遵守的準則。整體框架必須以 data flow 模式，進行連接，確保資料流通順暢，而各自 module 則於最內層 for loop 做到 Pipeline，這樣方可確保 module 內之資料不會被卡住，確保資料於 module 內亦流通順暢。

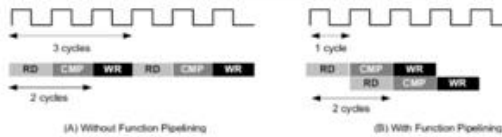


Performance & Resource Estimates

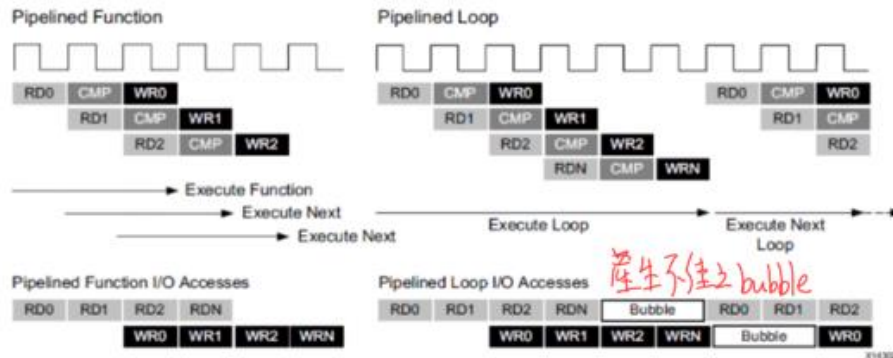
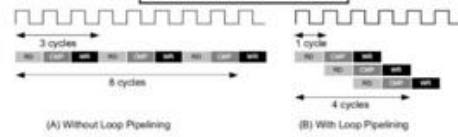
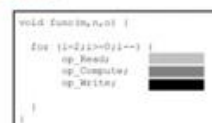
Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSR	FF	LUT	URAM
adder		-	4113	4.113E4	-	4107	-	dataflow	2	0	1777	2133	0
read_input3		-	4106	4.106E4	-	4106	-	no	0	0	112	249	0
mem_rd		-	4097	4.097E4	3	1	4096	yes	-	-	-	-	-
write_result		-	4104	4.104E4	-	4104	-	no	0	0	205	232	0
mem_wr		-	4097	4.097E4	3	1	4096	yes	-	-	-	-	-
compute_add		-	4098	4.098E4	-	4098	-	no	0	0	102	204	0
execute		-	4096	4.096E4	2	1	4096	yes	-	-	-	-	-

從上圖可以知道，除了整體框架以 dataflow 流動外，各自 module 最內層仍需以 pipeline 為主，確保 II=1，資料流通順暢。

① Function Pipeline



② Loop pipeline

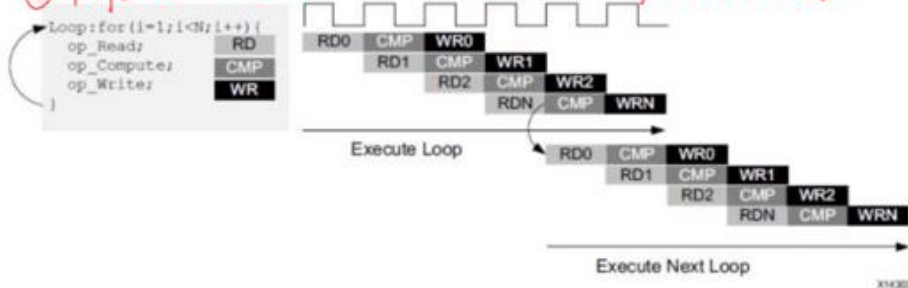


Function Pipeline: 確保 function 之間能夠以 Pipeline 完成，避免單一 function 執行時，其他待機 function 的浪費。

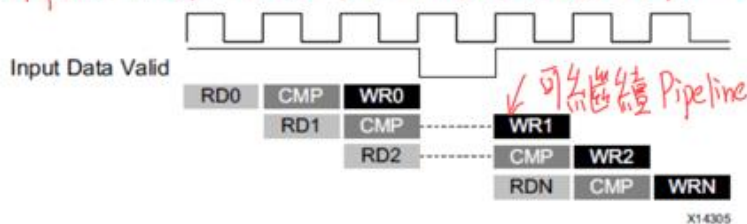
Loop pipeline，在 module 內的運算，確保其循環進行 pipeline，避免運算時間之浪費。

而 loop pipeline 所產生的不佳 bubble，能夠以下圖之 pipeline rewind 來進行處理。

③ Pipe line rewind: 防止反覆 call loop 造成之 bubble

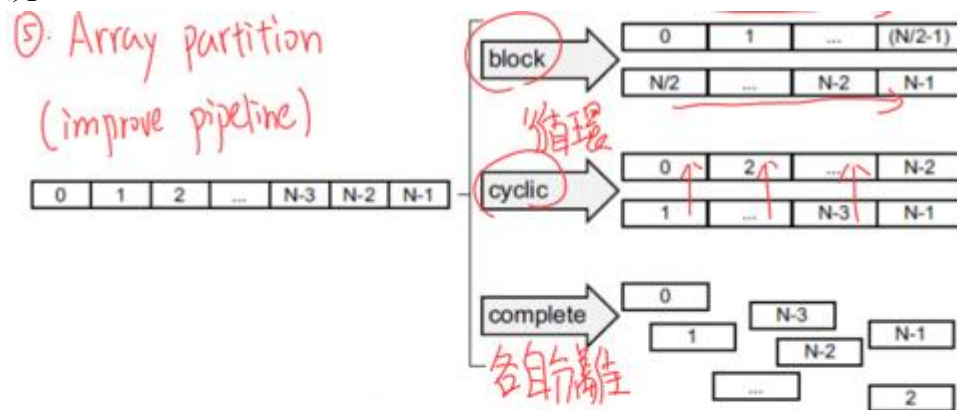


④ Pipeline Flush: 解決 input data 未進入造成之 pipeline 中止情況



以 Pipeline rewind，確保反覆 call loop 造成之 bubble 能夠有所消除。

以 Pipeline Flush，解決 Input data 尚未進入 pipeline 中，導致 pipeline 驟停之情況。



```
void foo (...) {
    int my_array[10][6][4];
    ...
    my_array[10][6][4] = ...
}
```

三維

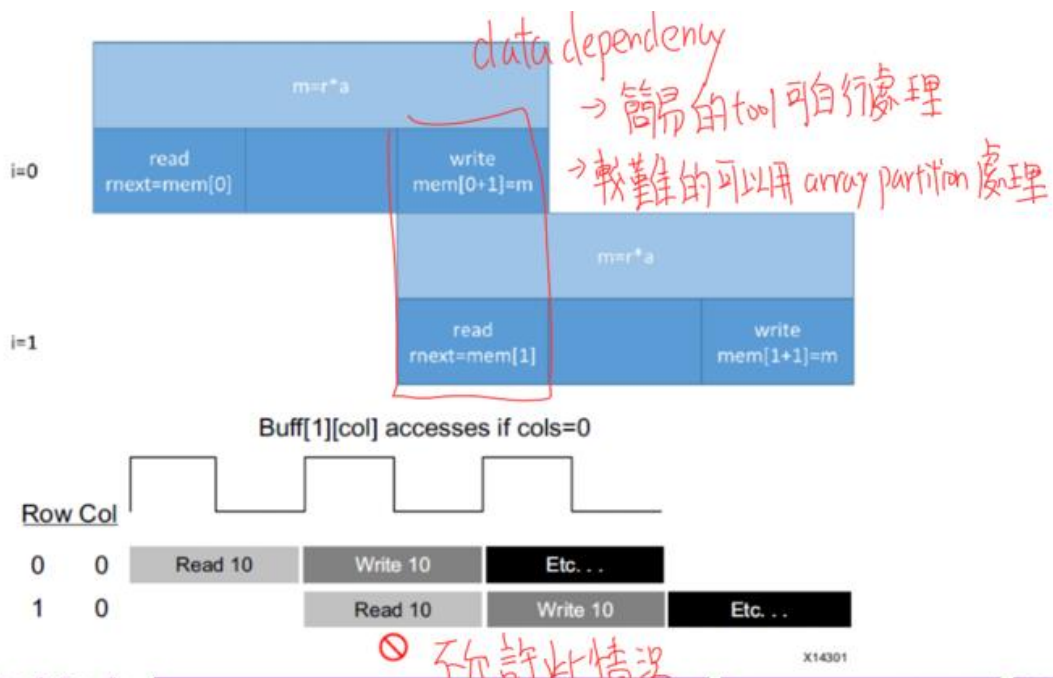
以第3維度分割

my_array[10][6][4] → partition dimension 3 → 4's [0][0]

my_array[10][6][4] → partition dimension 1 → 10's [6][4]

my_array[10][6][4] → partition dimension 0 → 10x6x4 = 240 registers → 全部切開 240's (10x6x4)

以 array partition 之情況來提升 Pipeline 的效能。



另外亦可以透過 array partition 來緩解 data dependency 之情況。

⑥ Loop unrolling

```
void top(...) {
    ...
    for_mult: for (i=0; i>0; i++) {
        a[i] = b[i] * c[i];
    }
    ...
}
```

Rolled Loop

Read b[3]	Read b[2]	Read b[1]	Read b[0]
Read c[3]	Read c[2]	Read c[1]	Read c[0]
*	*	*	*
Write a[3]	Write a[2]	Write a[1]	Write a[0]

單片乘法器 4 次

Partially Unrolled Loop

Read b[3]	Read b[1]
Read c[3]	Read c[1]
Read b[2]	Read b[0]
Read c[2]	Read c[0]
*	*
*	*
Write a[3]	Write a[1]
Write a[2]	Write a[0]

2 片乘法器 2 次

Unrolled Loop

Read b[3]
Read c[3]
Read b[2]
Read c[2]
Read b[1]
Read c[1]
Read b[0]
Read c[0]
*
*
*
*
Write a[3]
Write a[2]
Write a[1]
Write a[0]

4 片乘法器 1 次

以 Loop unrolling 來協助，以更多的硬體資源來達成更少的 Interval，但因為將會需要更多硬體資源，所以需要評估是否適合。

```
loop_region:
{
    add:
    for (i = 0; i < N; i++)
    {
        c[i] = a[i] + b[i];
    }
    sub:
    for (i = 0; i < N; i++)
    {
        d[i] = a[i] - b[i];
    }
}
```

Latency (clock cycles)

		s_default	s_merge
Latency	min	18	9
	max	18	9
Interval	min	19	10
	max	19	10

Utilization Estimates

	s_default	s_merge
BRAM_18K	0	0
DSP48E	0	0
FF	23	12
LUT	37	25

⑦ Loop merge



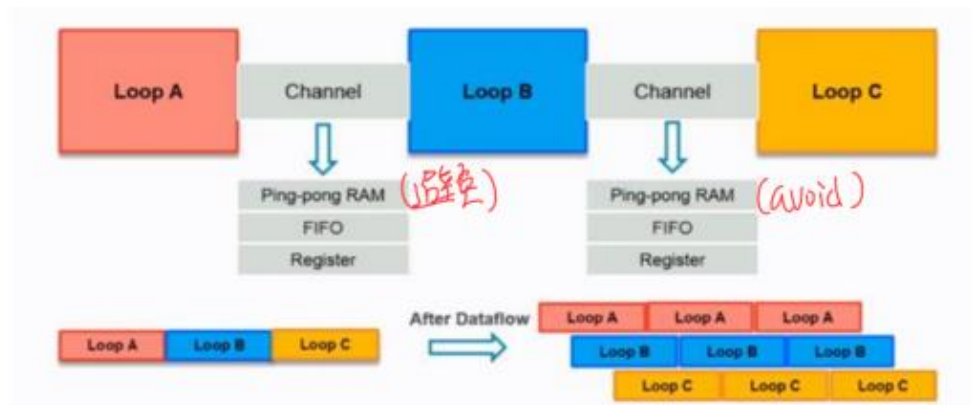
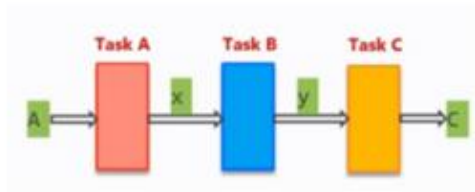
Latency		Interval		Type
min	max	min	max	
18	18	19	19	none

LI 減少

消耗資源亦減少

以 Loop merge，來協助合併迴圈，並且減少 II，亦可以減少消耗資源。

⑧ Dataflow 優化



Data flow 優化，將各自 module 以 Pipeline 形式合成，並且確保整體大架構以 data flow 形式進行合成，方可縮短 Interval 達成更好的藉由硬體加速的效能。

@其他

除了前述的 Loop 與 Dataflow，亦須活用 array partition 與 FIFO，來進行相關 HLS 設計，FIFO 能夠使 data stream 流動更順暢，但同時對於資料的存取順序更為要求，不可以為隨機存取。故可能需要對於 Source Code 進行相關修改。另外亦可以將 Input data 以適當之形式進行暫存，減少大量頻繁對於記憶體的重複讀取，消耗大量時間。

主要核心之設計守則：“大框架下以 Data flow 合成，確保資料於 sub module 中流動順暢，個別內層 Sub module 之最內層以 PipeLine 為主，確保 II = 1，使內部運算與資料傳遞不堵塞”。