

HLS Lab B -FIR

110061608 吳承哲

➤ Outline

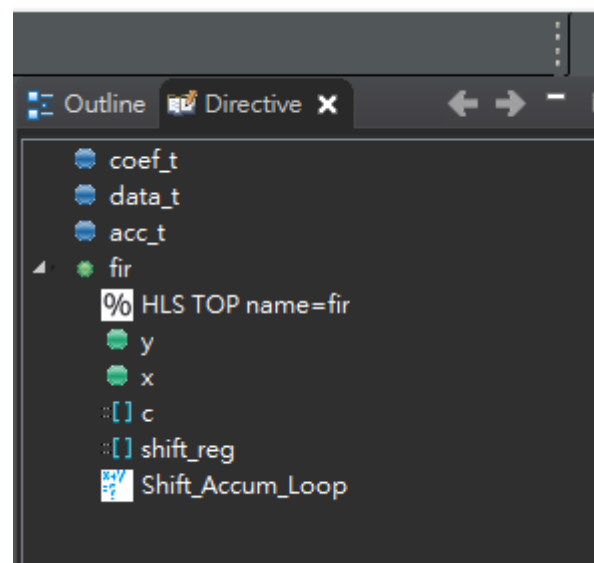
- FIR11
- FIR128_baseline
- FIR128_Q1 : Variable Bitwidths
- FIR128_Q2 : Pipelining
- FIR128_Q3 : Removing Conditional Statements
- FIR128_Q4 : Loop Partitioning
- FIR128_Q5 : Memory Partitioning
- FIR128_Q6 : Best Design
- Conclusion
- GitHub link

➤ FIR11

Code

```
13 #include "fir.h"
14 #define N 11
15 #include "ap_int.h"
16 typedef int coef_t;
17 typedef int data_t;
18 typedef int acc_t;
19
20 void fir (data_t *y,data_t x){
21     coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0,53};
22     // Write your code here
23     static
24     data_t shift_reg[N];
25     acc_t acc;
26     int i;
27     acc = 0;
28     Shift_Accum_Loop:
29     for (i = N - 1; i >= 0; i--) {
30         if (i == 0) {
31             acc += x * c[0];
32             shift_reg[0] = x;
33         } else {
34             shift_reg[i] = shift_reg[i - 1];
35             acc += shift_reg[i] * c[i];
36         }
37     }
38     *y = acc;
39 }
40
```

Hierarchy



Performance

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	6.912 ns	2.70 ns

Performance & Resource Estimates

Modules & Loops

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
fir	-	-	-	-	20	200.000	-	21	-	no	0	3	869	364	0
fir_Pipeline_Shift_Accum_Loop	-	-	-	-	16	160.000	-	16	-	no	0	2	666	283	0
Shift_Accum_Loop	-	-	-	-	14	140.000	5	1	11	yes	-	-	-	-	-

Utilization

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	0	3	831	333	0
Memory	-	-	-	-	-
Multiplexer	-	-	-	31	-
Register	-	-	38	-	-
Total	0	3	869	364	0
Available	280	220	106400	53200	0
Utilization (%)	0	1	~0	~0	0

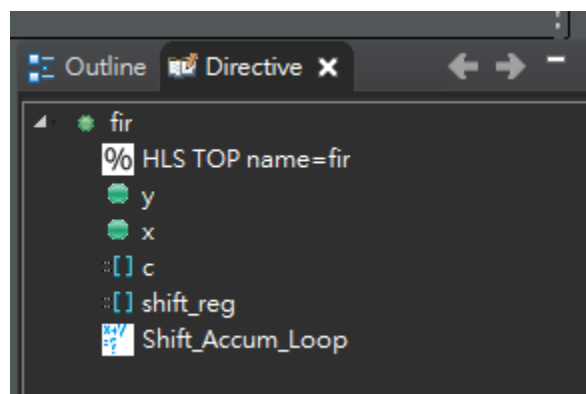
以上 fir11 都是經過 HLS 自動優化，只是實現基本 fir11 的功能，並成功合成出來。

➤ FIR128_baseline

Code

```
12
13 #include "fir.h"
14 #define N 128
15 #include "ap_int.h"
16 void fir (data_t *y,data_t x){
17
18     coef_t c[N] = {10, 11, 11, 8, 3, -3, -8, -11, -11, -10,
19
20     // Write your code here
21     static
22     data_t shift_reg[N];
23     acc_t acc;
24     int i;
25     acc = 0;
26     Shift_Accum_Loop:
27     for (i = N - 1; i >= 0; i--) {
28         if (i == 0) {
29             acc += x * c[0];
30             shift_reg[0] = x;
31         } else {
32             shift_reg[i] = shift_reg[i - 1];
33             acc += shift_reg[i] * c[i];
34         }
35     }
36     *y = acc;
37 }
38
```

Hierarchy



Performance

▼ Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	6.912 ns	2.70 ns

▼ Performance & Resource Estimates

Modules

Loops

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
▲ fir			-		135	1.350E3		136	-	no	3	1	611	335	0
▲ fir.Pipeline_Shift_Accum_Loop			-		133	1.330E3		133	-	no	3	1	576	276	0
Shift_Accum_Loop			-		131	1.310E3	5	1	128	yes	-	-	-	-	-

Utilization

Utilization Estimates					
Summary					
Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	39	-
FIFO	-	-	-	-	-
Instance	3	1	576	276	0
Memory	-	-	-	-	-
Multiplexer	-	-	-	20	-
Register	-	-	35	-	-
Total	3	1	611	335	0
Available	280	220	106400	53200	0
Utilization (%)	1	~0	~0	~0	0

以上是 FIR128 經過 HLS 自動優化後的結果，因為處理的 coefficient 變多所以花的 cycle 數和硬體也變多，接下來的幾個 question 會提出幾種優化的方式，最後並設計出 Best design。

- FIR128_Q1 : Variable Bitwidths

Code

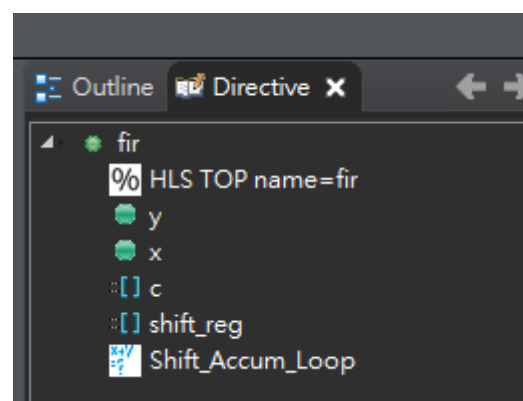
```
#include "fir.h"
#define N 128
#include "ap_int.h"
typedef ap_int<8> reg_t;

void fir(data_t y,data_t x){

    coef_t c[N] = {10, 11, 11, 8, 3, -3, -8, -11, -11, -10, -10, -10};

    // Write your code here
    static
    reg_t shift_reg[N];
    acc_t acc;
    int i;
    acc = 0;
Shift_Accum_Loop:
    for (i = N - 1; i >= 0; i--) {
        if (i == 0) {
            acc += x * c[0];
            shift_reg[0] = x;
        } else {
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * c[i];
        }
    }
    *y = acc;
}
```

Hierarchy



Performance

Performance Estimates					
Timing					
Clock		8bit	16bit	32bit	64bit
ap_clk	Target	10.00 ns	10.00 ns	10.00 ns	10.00 ns
	Estimated	6.508 ns	6.508 ns	6.912 ns	6.912 ns
Latency					
		8bit	16bit	32bit	64bit
Latency (cycles)	min	134	134	135	135
	max	134	134	135	135
Latency (absolute)	min	1.340 us	1.340 us	1.350 us	1.350 us
	max	1.340 us	1.340 us	1.350 us	1.350 us
Interval (cycles)	min	135	135	136	136
	max	135	135	136	136

Utilization

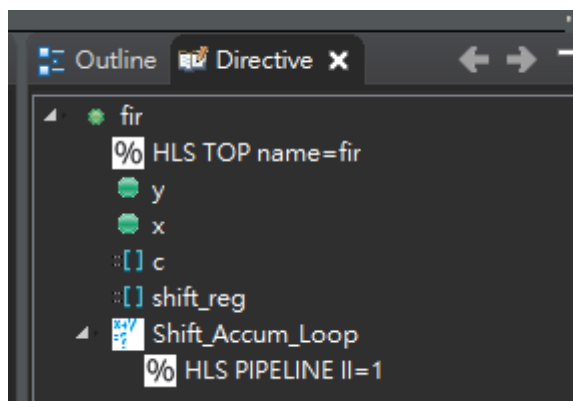
Utilization Estimates				
	8bit	16bit	32bit	64bit
BRAM_18K	2	2	3	3
DSP	0	1	1	1
FF	337	353	611	611
LUT	326	291	335	335
URAM	0	0	0	0

以上是透過改變 shift register 的 bitwidth 來觀察 performance 和 utilization 的變化，隨著 bit 數的增加，Latency 都差不多，但是從 16bit 變成 32bit 時，硬體使用的數量會大幅度的上升，因為要搬移的 data bitwidth 增加，所以需要用到的硬體也會變多，還有因為 32 bits 以上之後的 bit 都是多餘的，所以用到的硬體才不會繼續增加。

➤ FIR128_Q2 : Pipelining

Code 的部分沒有做更動，主要是加入 Pipeline 的 pragma，並改變 II 從 1 ~ 5 觀察發生的變化。

Hierarchy



Performance

Performance Estimates						
Timing						
Clock		II_1	II_2	II_3	II_4	II_5
ap_clk	Target	10.00 ns	10.00 ns	10.00 ns	10.00 ns	10.00 ns
	Estimated	6.508 ns	6.508 ns	6.508 ns	6.508 ns	6.508 ns
Latency						
		II_1	II_2	II_3	II_4	II_5
Latency (cycles)	min	134	261	388	516	516
	max	134	261	388	516	516
Latency (absolute)	min	1.340 us	2.610 us	3.880 us	5.160 us	5.160 us
	max	1.340 us	2.610 us	3.880 us	5.160 us	5.160 us
Interval (cycles)	min	135	262	389	517	517
	max	135	262	389	517	517

Utilization

Utilization Estimates					
	II_1	II_2	II_3	II_4	II_5
BRAM_18K	2	2	2	2	2
DSP	0	0	0	0	0
FF	337	175	176	151	151
LUT	326	296	313	283	283
URAM	0	0	0	0	0

從以上結果可以觀察出隨著 II 的增加，雖然 latency 上升，但是硬體的使用會減少，II 從 1 變到 2 的時候大概就是 latency 變成 2 倍，但使用的 FF 也減少一半。從 2 變成 3 時，可以看出 latency 上升但硬體使用的情況沒發生什麼變化，在 FIR 的運算中，依次 iteration 應該是 3 個 cycle，所以 II=2 或 3 使用硬體的效率差不多，但是會浪費掉一個 cycle，而 II=4 和 5 可以看出硬體的使用量有稍微降低一些，但是從 latency 的角度來看，總共耗費太多 cycle 了。

➤ FIR128_Q3 : Removing Conditional Statements

Code

```
#include "fir.h"
#define N 128
#include "ap_int.h"
typedef ap_int<8> reg_t;
void fir (data_t *y,data_t x){

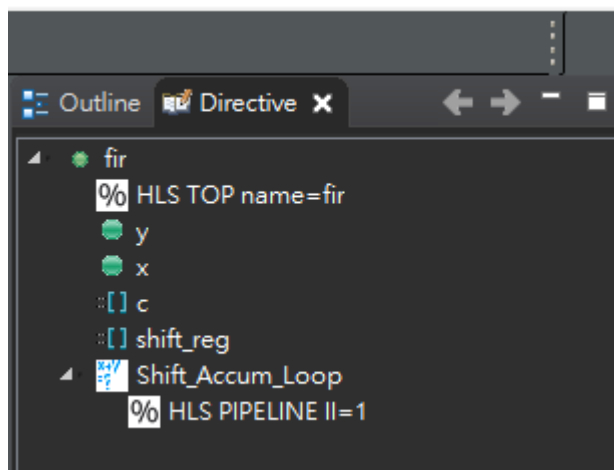
    coef_t c[N] = {10, 11, 11, 8, 3, -3, -8, -11, -11, -10, -10};

    // Write your code here
    static
    reg_t shift_reg[N];
    acc_t acc;
    int i;
    acc = 0;
Shift_Accum_Loop:
    for (i = N - 1; i >= 0; i--) {

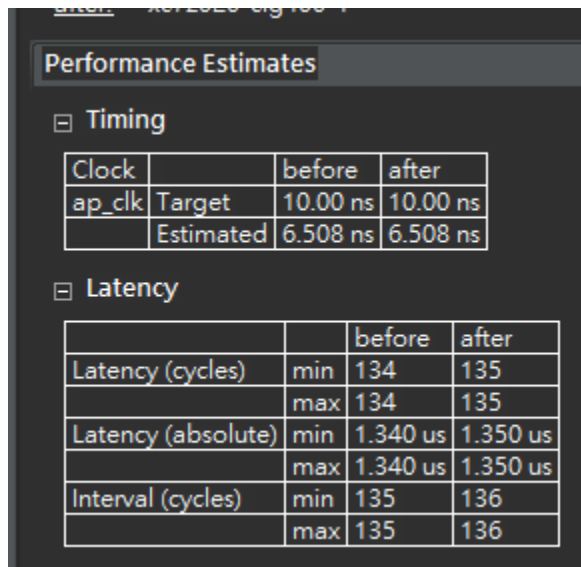
        shift_reg[i] = shift_reg[i - 1];
        acc += shift_reg[i] * c[i];
    }
    shift_reg[0] = x;
    acc += x * c[0];
    *y = acc;
}
```

以上是將 if-else 移除掉之後，再將 $i = 0$ 的情況，補在 for-loop 後面，就可以達到原本的功能。

Hierarchy



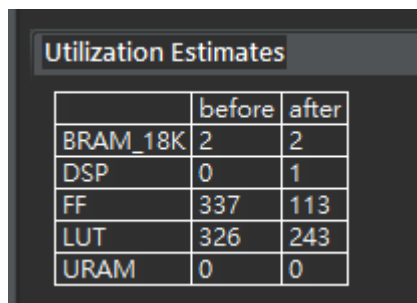
Performance



The image shows a 'Performance Estimates' window with two sections: 'Timing' and 'Latency'. The 'Timing' section contains a table with clock information. The 'Latency' section contains a table with various latency and interval metrics, each with 'before' and 'after' values.

Performance Estimates			
Timing			
Clock		before	after
ap_clk	Target	10.00 ns	10.00 ns
	Estimated	6.508 ns	6.508 ns
Latency			
		before	after
Latency (cycles)	min	134	135
	max	134	135
Latency (absolute)	min	1.340 us	1.350 us
	max	1.340 us	1.350 us
Interval (cycles)	min	135	136
	max	135	136

Utilization



The image shows a 'Utilization Estimates' window with a table listing resource usage. The table has three columns: resource name, 'before' value, and 'after' value.

	before	after
BRAM_18K	2	2
DSP	0	1
FF	337	113
LUT	326	243
URAM	0	0

根據上述的部分可以觀察出，經過 remove condition 的動作
雖然 latency 的部分沒有降低，但是可以看出在硬體使用的
部分少了許多。

➤ FIR128_Q4 : Loop Partitioning

Code

```
#include "fir.h"
#define N 128
#include "ap_int.h"
typedef ap_int<8> reg_t;
void fir (data_t *y,data_t x){

    coef_t c[N] = {10, 11, 11, 8, 3, -3, -8, -11, -11, -10, -10, -10, -10, -10, -10, -10};

    // Write your code here
    static
    reg_t shift_reg[N];
    acc_t acc;
    int i;
Time_delay_loop:
    for (i = N - 1; i >= 0; i--) {

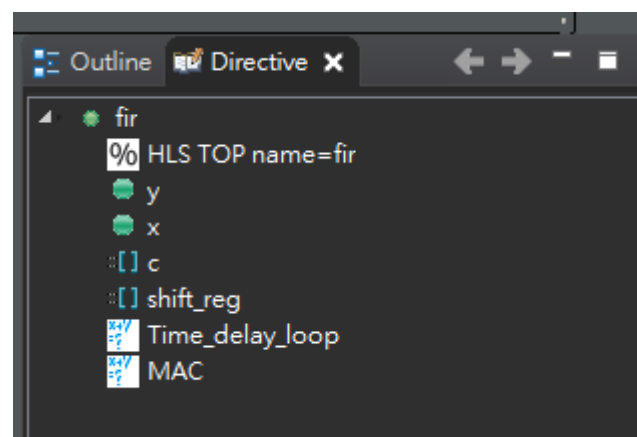
        shift_reg[i] = shift_reg[i - 1];
    }
    shift_reg[0] = x;

    acc = 0;
MAC:
    for (i = N - 1; i >= 0 ; i--) {
        acc += shift_reg[i] * c[i];
    }

    *y = acc;
```

以上是將 FIR 拆分成兩個 loop，一個是 TDL 另一個是 MAC。

Hierarchy



Performance

Performance Estimates			
Timing			
Clock		merge	fission
ap_clk	Target	10.00 ns	10.00 ns
	Estimated	6.508 ns	6.508 ns
Latency			
		merge	fission
Latency (cycles)	min	135	268
	max	135	268
Latency (absolute)	min	1.350 us	2.680 us
	max	1.350 us	2.680 us
Interval (cycles)	min	136	269
	max	136	269

Utilization

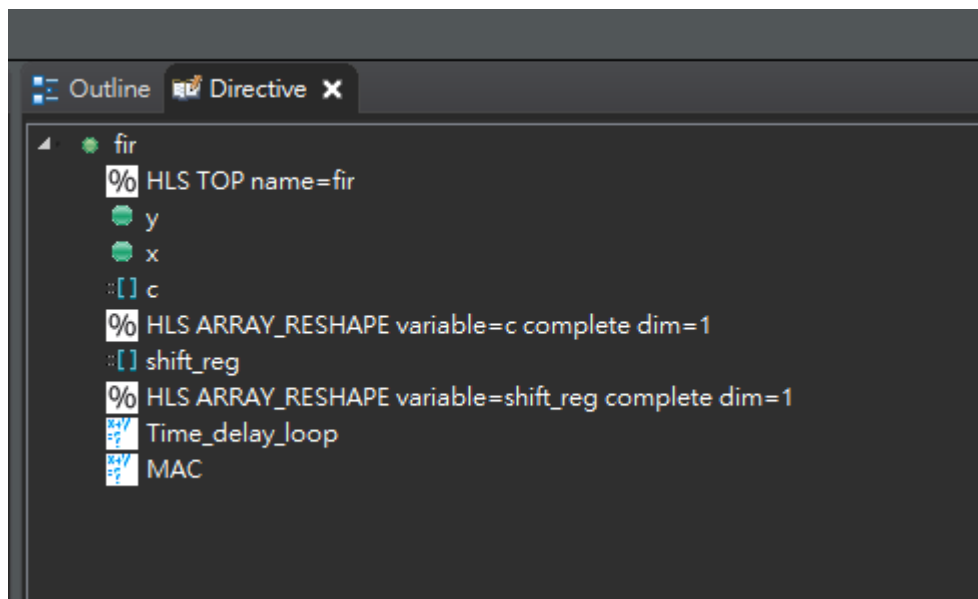
Utilization Estimates		
	merge	fission
BRAM_18K	2	2
DSP	1	1
FF	113	128
LUT	243	270
URAM	0	0

以上的結果只有經過 HLS 自動優化，如果只是單純將 loop 拆開來沒有做任何處理，performance 不一定會提高。

➤ FIR128_Q5 : Memory Partitioning

Code 的部分是使用 loop fission 的 code，並透過
insert array_reshape 的 pragma 來觀察發生的變化。

Hierarchy



Performance

Performance Estimates				
Timing				
Clock		origin	reshape	reshape_pipelinefunction
ap_clk	Target	10.00 ns	10.00 ns	10.00 ns
	Estimated	6.508 ns	6.912 ns	7.108 ns
Latency				
		origin	reshape	reshape_pipelinefunction
Latency (cycles)	min	268	267	24
	max	268	267	24
Latency (absolute)	min	2.680 us	2.670 us	0.240 us
	max	2.680 us	2.670 us	0.240 us
Interval (cycles)	min	269	268	1
	max	269	268	1

Utilization

Utilization Estimates			
	origin	reshape	reshape_pipelinefunction
BRAM_18K	2	0	0
DSP	1	1	1
FF	128	4418	6014
LUT	270	14180	3618
URAM	0	0	0

從以上結果可以看出，單純將 array complete reshape 的話，想要達到相同的 performance，在硬體的使用上就會大幅度提升，這邊我還有實驗一下，pipeline function 的話會將兩個 loop fully unroll，可以看出 latency 大幅度的減少，但是硬體的使用上 FF 變高但 LUT 的用量減少，可以利用以上觀察出的結果，設計出 best design。

➤ FIR128_Q6 : Best Design

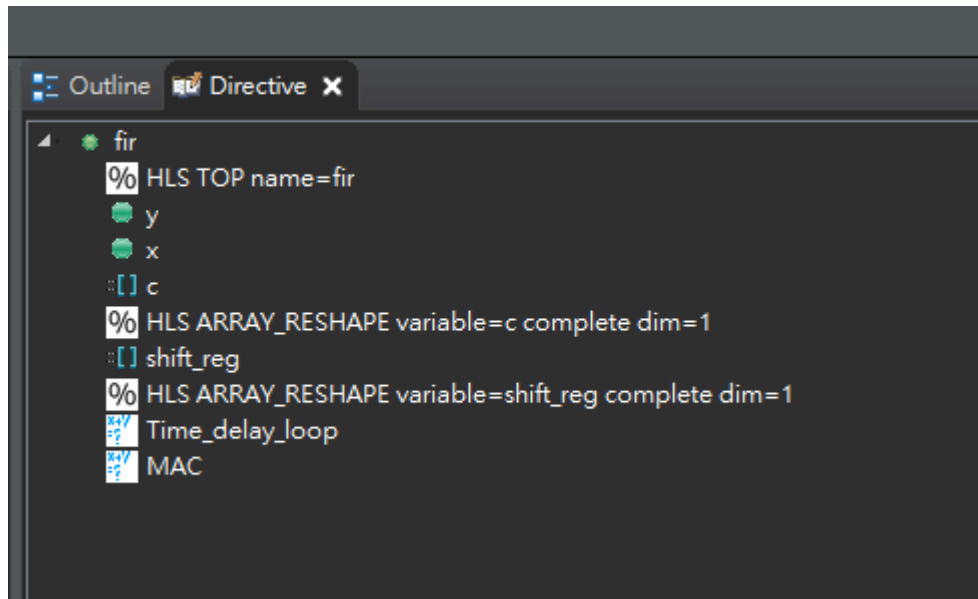
Code

```
12
13 #include "fir.h"
14 #define N 128
15 #include "ap_int.h"
16 typedef ap_int<8> reg_t;
17 typedef ap_int<5> c_t;
18 void fir (data_t *y,data_t x){
19
20     c_t c[N] = {10, 11, 11, 8, 3, -3, -8, -11, -11, -10, -10, -10,
21
22     // Write your code here
23     static
24     reg_t shift_reg[N];
25     acc_t acc = 0;
26     int i;
27     /* Time_delay_loop:
28     for (i = N - 1; i >= 0; i--) {
29         shift_reg[i] = shift_reg[i - 1];
30     }
31     shift_reg[0] = x;
32
33     MAC:
34     for (i = N - 1; i >= 0 ; i--) {
35         acc += shift_reg[i] * c[i];
36     }
37
38     *y = acc;
39 }
40 */
41
42 FIR:
43 for (i = N - 1; i >= 0; i--) {
44
45     shift_reg[i] = shift_reg[i - 1];
46     acc += shift_reg[i] * c[i];
47 }
48 shift_reg[0] = x;
49 acc += shift_reg[0] * c[0];
50 *y = acc;
51 }
52
53
```

Code 的部分我將 loop merge 和 loop fission 的部分放

在一起，可以用不同的 solution 來比較。

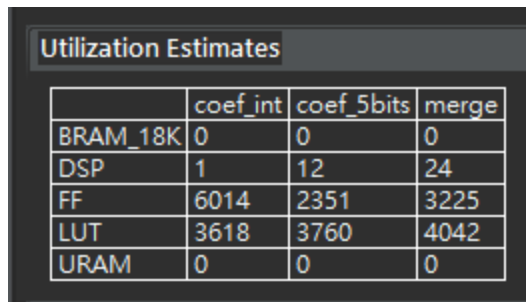
Hierarchy



Performance

Performance Estimates				
Timing				
Clock		coef_int	coef_5bits	merge
ap_clk	Target	10.00 ns	10.00 ns	10.00 ns
	Estimated	7.108 ns	7.013 ns	7.134 ns
Latency				
Latency (cycles)	min	24	4	5
	max	24	4	5
Latency (absolute)	min	0.240 us	40.000 ns	50.000 ns
	max	0.240 us	40.000 ns	50.000 ns
Interval (cycles)	min	1	1	1
	max	1	1	1

Utilization



	coef_int	coef_5bits	merge
BRAM_18K	0	0	0
DSP	1	12	24
FF	6014	2351	3225
LUT	3618	3760	4042
URAM	0	0	0

bitwidth 的部分，我將 shift register 設為 8bit，
coefficient array 設為 5bit，可以從上述結果看出從
integer(32bits)變成 5 bits，整體的 latency 和硬體使用
來看，都提升許多，pipeline 的部分我是做 function
pipeline，我也有試過將 inner loop pipeline 但是效果
沒有 function pipeline 好，最後我還有將 loop merge
並做 function pipeline，他會多 loop fission 一個
cycle， 後來發現是 DSP 產生出的電路不同，
如下圖：

Merge_DSP

+ Instance		
- DSP		
Instance	Module	Expression
mac_muladd_8s_4ns_11s_13_4_1_U2	mac_muladd_8s_4ns_11s_13_4_1	$i0 * i1 + i2$
mac_muladd_8s_4ns_11s_13_4_1_U7	mac_muladd_8s_4ns_11s_13_4_1	$i0 + i1 * i2$
mac_muladd_8s_4ns_11s_13_4_1_U10	mac_muladd_8s_4ns_11s_13_4_1	$i0 * i1 + i2$
mac_muladd_8s_4ns_11s_13_4_1_U15	mac_muladd_8s_4ns_11s_13_4_1	$i0 + i1 * i2$
mac_muladd_8s_4ns_11s_13_4_1_U18	mac_muladd_8s_4ns_11s_13_4_1	$i0 * i1 + i2$
mac_muladd_8s_4ns_11s_13_4_1_U23	mac_muladd_8s_4ns_11s_13_4_1	$i0 + i1 * i2$
mac_muladd_8s_4ns_12s_13_4_1_U1	mac_muladd_8s_4ns_12s_13_4_1	$i0 + i1 * i2$
mac_muladd_8s_4ns_12s_13_4_1_U8	mac_muladd_8s_4ns_12s_13_4_1	$i0 * i1 + i2$
mac_muladd_8s_4ns_12s_13_4_1_U9	mac_muladd_8s_4ns_12s_13_4_1	$i0 + i1 * i2$
mac_muladd_8s_4ns_12s_13_4_1_U16	mac_muladd_8s_4ns_12s_13_4_1	$i0 * i1 + i2$
mac_muladd_8s_4ns_12s_13_4_1_U17	mac_muladd_8s_4ns_12s_13_4_1	$i0 + i1 * i2$
mac_muladd_8s_4ns_4ns_12_4_1_U24	mac_muladd_8s_4ns_4ns_12_4_1	$i0 * i1 + i2$
mac_muladd_8s_5s_12s_13_4_1_U3	mac_muladd_8s_5s_12s_13_4_1	$i0 + i1 * i2$
mac_muladd_8s_5s_12s_13_4_1_U4	mac_muladd_8s_5s_12s_13_4_1	$i0 * i1 + i2$
mac_muladd_8s_5s_12s_13_4_1_U5	mac_muladd_8s_5s_12s_13_4_1	$i0 + i1 * i2$
mac_muladd_8s_5s_12s_13_4_1_U6	mac_muladd_8s_5s_12s_13_4_1	$i0 * i1 + i2$
mac_muladd_8s_5s_12s_13_4_1_U11	mac_muladd_8s_5s_12s_13_4_1	$i0 + i1 * i2$
mac_muladd_8s_5s_12s_13_4_1_U12	mac_muladd_8s_5s_12s_13_4_1	$i0 * i1 + i2$
mac_muladd_8s_5s_12s_13_4_1_U13	mac_muladd_8s_5s_12s_13_4_1	$i0 + i1 * i2$
mac_muladd_8s_5s_12s_13_4_1_U14	mac_muladd_8s_5s_12s_13_4_1	$i0 * i1 + i2$
mac_muladd_8s_5s_12s_13_4_1_U19	mac_muladd_8s_5s_12s_13_4_1	$i0 + i1 * i2$
mac_muladd_8s_5s_12s_13_4_1_U20	mac_muladd_8s_5s_12s_13_4_1	$i0 * i1 + i2$
mac_muladd_8s_5s_12s_13_4_1_U21	mac_muladd_8s_5s_12s_13_4_1	$i0 + i1 * i2$
mac_muladd_8s_5s_12s_13_4_1_U22	mac_muladd_8s_5s_12s_13_4_1	$i0 * i1 + i2$

Fission_DSP

+ Instance		
- DSP		
Instance	Module	Expression
am_addmul_8s_8s_5s_13_4_1_U2	am_addmul_8s_8s_5s_13_4_1	$(i0 + i1) * i2$
am_addmul_8s_8s_5s_13_4_1_U3	am_addmul_8s_8s_5s_13_4_1	$(i0 + i1) * i2$
am_addmul_8s_8s_5s_13_4_1_U6	am_addmul_8s_8s_5s_13_4_1	$(i0 + i1) * i2$
am_addmul_8s_8s_5s_13_4_1_U7	am_addmul_8s_8s_5s_13_4_1	$(i0 + i1) * i2$
am_addmul_8s_8s_5s_13_4_1_U10	am_addmul_8s_8s_5s_13_4_1	$(i0 + i1) * i2$
am_addmul_8s_8s_5s_13_4_1_U11	am_addmul_8s_8s_5s_13_4_1	$(i0 + i1) * i2$
ama_addmuladd_8s_8s_4ns_12s_14_4_1_U1	ama_addmuladd_8s_8s_4ns_12s_14_4_1	$i0 + (i1 + i2) * i3$
ama_addmuladd_8s_8s_4ns_12s_14_4_1_U4	ama_addmuladd_8s_8s_4ns_12s_14_4_1	$(i0 + i1) * i2 + i3$
ama_addmuladd_8s_8s_4ns_12s_14_4_1_U5	ama_addmuladd_8s_8s_4ns_12s_14_4_1	$i0 + (i1 + i2) * i3$
ama_addmuladd_8s_8s_4ns_12s_14_4_1_U8	ama_addmuladd_8s_8s_4ns_12s_14_4_1	$(i0 + i1) * i2 + i3$
ama_addmuladd_8s_8s_4ns_12s_14_4_1_U9	ama_addmuladd_8s_8s_4ns_12s_14_4_1	$i0 + (i1 + i2) * i3$
ama_addmuladd_8s_8s_4ns_12s_14_4_1_U12	ama_addmuladd_8s_8s_4ns_12s_14_4_1	$(i0 + i1) * i2 + i3$

我猜想大概是合成出的電路不同，導致整體的

latency 也會不一樣。

➤ Conclusion

透過這一連串的 optimization，讓我對於 Vitis HLS 的使用又有一些新的理解，不管是 memory 的 partition 或是 pragma 的使用都更加熟悉如何使用，還有一些 coding style 的技巧也學到了一些，真的是受益良多。

➤ Github link

https://github.com/sssh311318/HLS_LabB