Xilinx HLS Example – FP_ACCUM

Self-paced Report
R09943020 電子所碩一 杜承諺

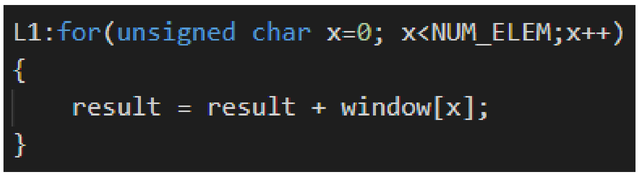
1. HLS C-sim/Synthesis/Cosim (Screenshot + brief intro)

系統介紹:

- float32 的累加器。
- Input: float array，大小為 128
- Output: sum，所有值相加結果

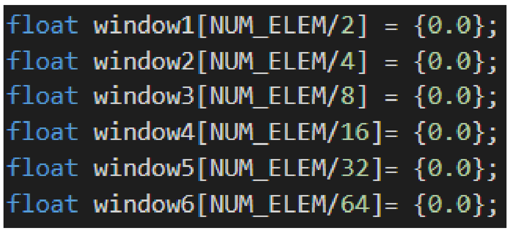
Source code:

- **Source code 提供兩種實作方式來計算相加結果**
- **傳統累加器**



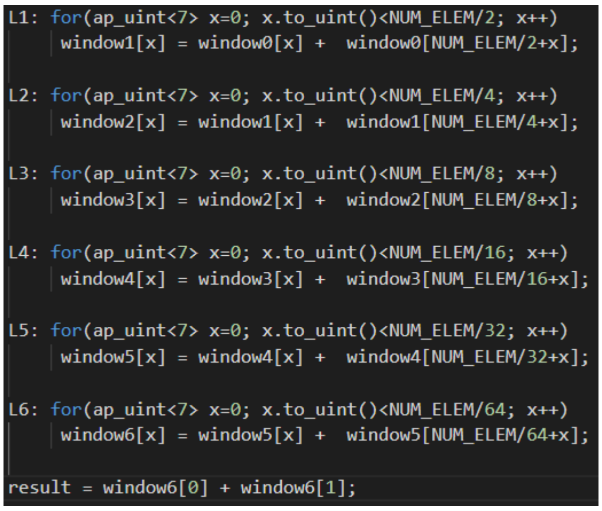
```
L1:for(unsigned char x=0; x<NUM_ELEM;x++)
{
    result = result + window[x];
}
```

- 加法樹
  - 另外宣告 6 個 array 儲存每層加法樹的 partial sum



```
float window1[NUM_ELEM/2]  = {0.0};
float window2[NUM_ELEM/4]  = {0.0};
float window3[NUM_ELEM/8]  = {0.0};
float window4[NUM_ELEM/16]= {0.0};
float window5[NUM_ELEM/32]= {0.0};
float window6[NUM_ELEM/64]= {0.0};
```

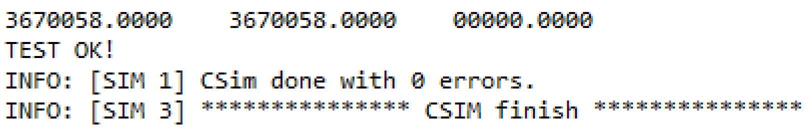
  - 每層一次計算兩兩相加



```
L1: for(ap_uint<7> x=0; x.to_uint()<NUM_ELEM/2; x++)
      window1[x] = window0[x] +  window0[NUM_ELEM/2+x];

L2: for(ap_uint<7> x=0; x.to_uint()<NUM_ELEM/4; x++)
      window2[x] = window1[x] +  window1[NUM_ELEM/4+x];

L3: for(ap_uint<7> x=0; x.to_uint()<NUM_ELEM/8; x++)
      window3[x] = window2[x] +  window2[NUM_ELEM/8+x];

L4: for(ap_uint<7> x=0; x.to_uint()<NUM_ELEM/16; x++)
      window4[x] = window3[x] +  window3[NUM_ELEM/16+x];

L5: for(ap_uint<7> x=0; x.to_uint()<NUM_ELEM/32; x++)
      window5[x] = window4[x] +  window4[NUM_ELEM/32+x];

L6: for(ap_uint<7> x=0; x.to_uint()<NUM_ELEM/64; x++)
      window6[x] = window5[x] +  window5[NUM_ELEM/64+x];

result = window6[0] + window6[1];
```

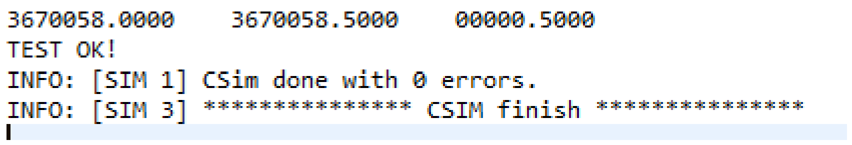
C-sim

- 累加器



```
3670058.0000    3670058.0000    00000.0000
TEST OK!
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

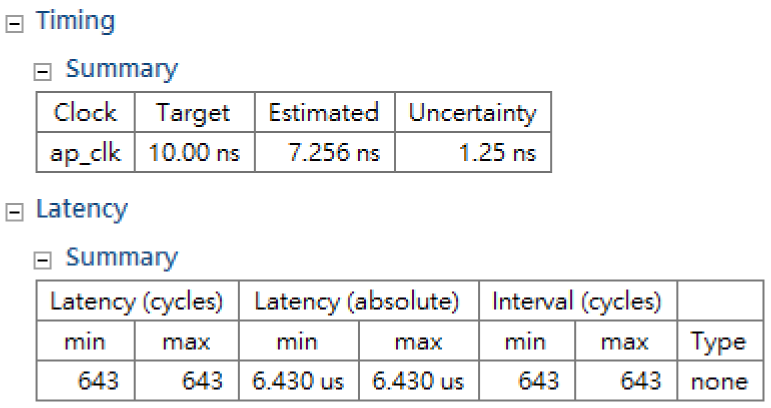
- 加法樹



```
3670058.0000    3670058.5000    00000.5000
TEST OK!
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Synthesis

- 累加器



⊟ Timing

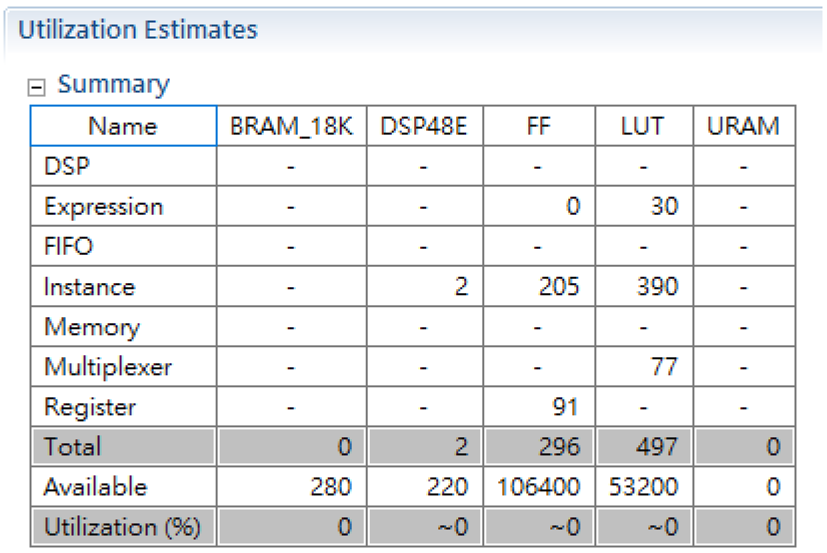
⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.256 ns	1.25 ns

⊟ Latency

⊟ Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
643	643	6.430 us	6.430 us	643	643	none

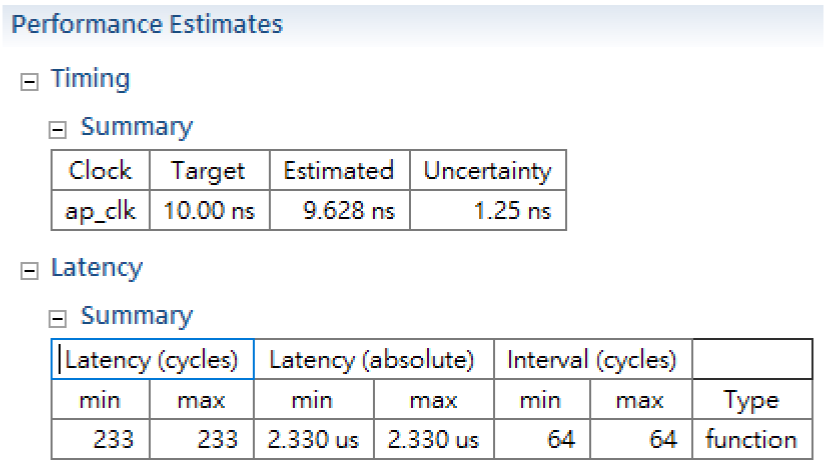


Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	30	-
FIFO	-	-	-	-	-
Instance	-	2	205	390	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	77	-
Register	-	-	91	-	-
Total	0	2	296	497	0
Available	280	220	106400	53200	0
Utilization (%)	0	~0	~0	~0	0

- 加法樹



Performance Estimates

⊟ Timing

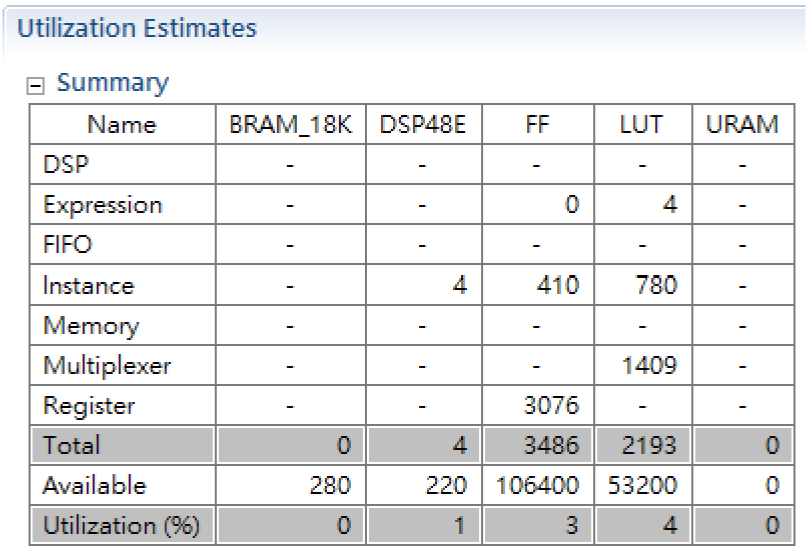
⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	9.628 ns	1.25 ns

⊟ Latency

⊟ Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
233	233	2.330 us	2.330 us	64	64	function



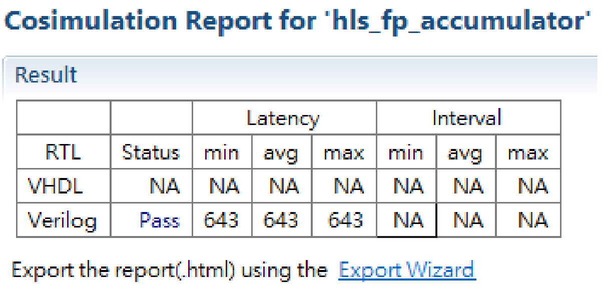
Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	4	-
FIFO	-	-	-	-	-
Instance	-	4	410	780	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	1409	-
Register	-	-	3076	-	-
Total	0	4	3486	2193	0
Available	280	220	106400	53200	0
Utilization (%)	0	1	3	4	0

Cosim

- 累加器



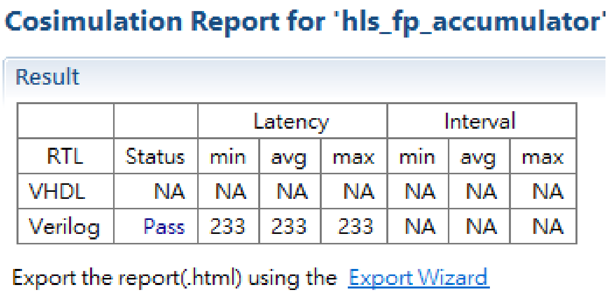
Cosimulation Report for 'hls_fp_accumulator'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	643	643	643	NA	NA	NA

Export the report(.html) using the Export Wizard

- 加法樹



Cosimulation Report for 'hls_fp_accumulator'

Result

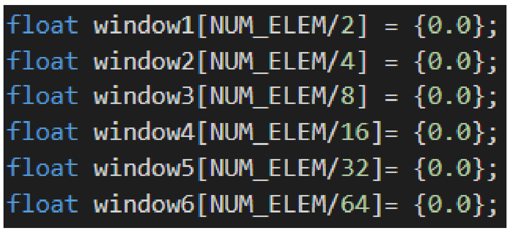
		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	233	233	233	NA	NA	NA

Export the report(.html) using the Export Wizard

2. Improvement throughput/area

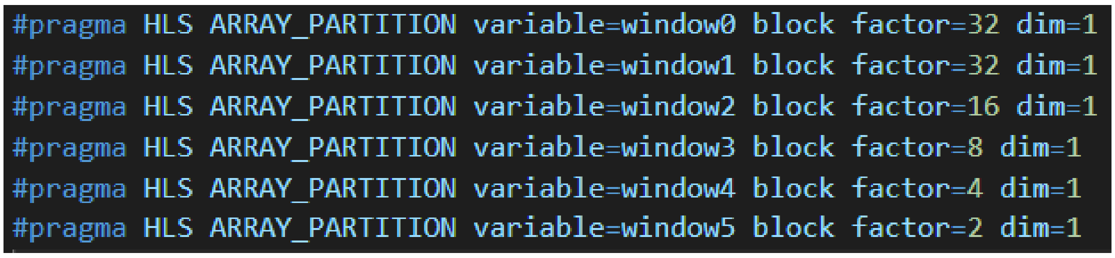
使用 ARRAY PARTITION 平行加法樹的運算，加速了 6.3 倍

- 觀察 source code 發現原本的做法開了 6 個 array 來暫存 adder tree 每層的值。



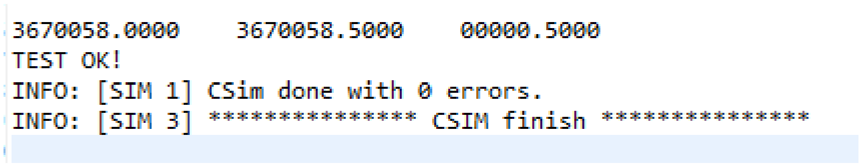
```
float window1[NUM_ELEM/2] = {0.0};
float window2[NUM_ELEM/4] = {0.0};
float window3[NUM_ELEM/8] = {0.0};
float window4[NUM_ELEM/16]= {0.0};
float window5[NUM_ELEM/32]= {0.0};
float window6[NUM_ELEM/64]= {0.0};
```

- 但是因為 BRAM 的限制導致加法樹同一層無法在同一 cycle 平行計算，最多只能一次計算兩個值相加。
- Array Partition
    - ■ 利用 array partition 增加每一層的 bandwidth
    - ■ 使所有加法器可以平行計算



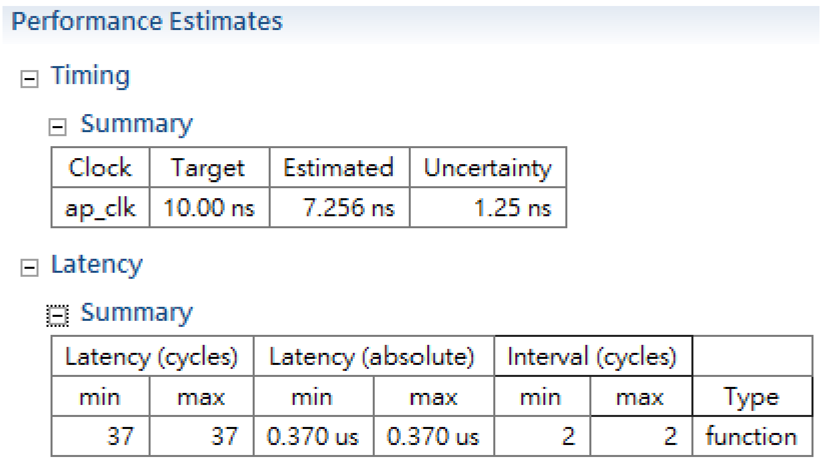
```
#pragma HLS ARRAY_PARTITION variable=window0 block factor=32 dim=1
#pragma HLS ARRAY_PARTITION variable=window1 block factor=32 dim=1
#pragma HLS ARRAY_PARTITION variable=window2 block factor=16 dim=1
#pragma HLS ARRAY_PARTITION variable=window3 block factor=8 dim=1
#pragma HLS ARRAY_PARTITION variable=window4 block factor=4 dim=1
#pragma HLS ARRAY_PARTITION variable=window5 block factor=2 dim=1
```

- Csim



```
3670058.0000    3670058.5000    00000.5000
TEST OK!
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

- Synthesis



Performance Estimates

Timing

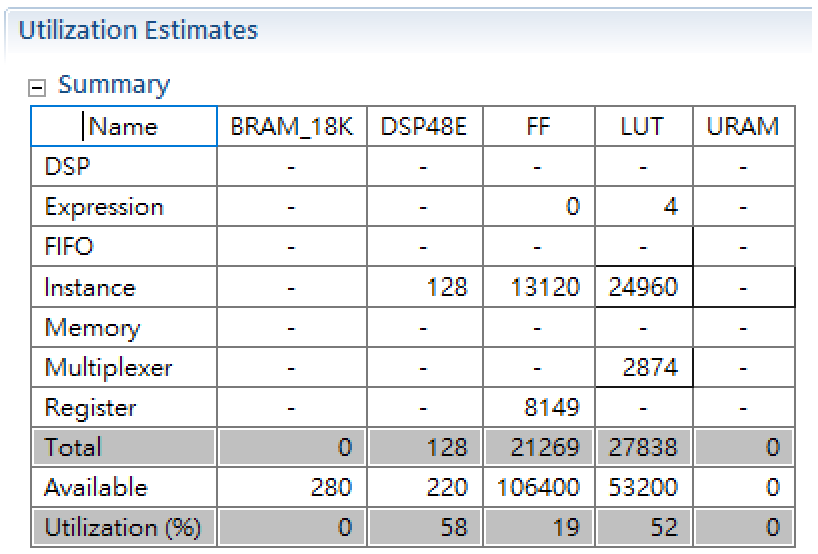
Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.256 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
37	37	0.370 us	0.370 us	2	2	function

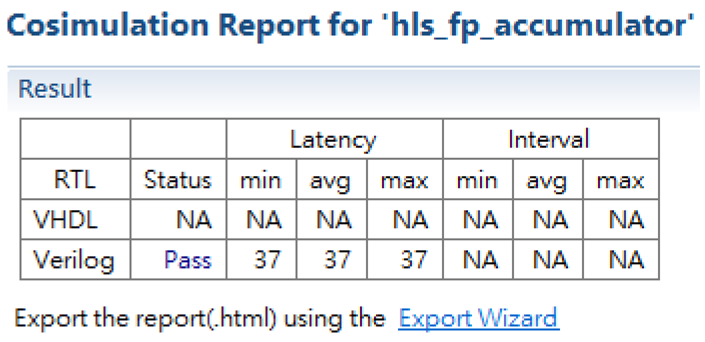


Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	4	-
FIFO	-	-	-	-	-
Instance	-	128	13120	24960	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	2874	-
Register	-	-	8149	-	-
Total	0	128	21269	27838	0
Available	280	220	106400	53200	0
Utilization (%)	0	58	19	52	0

- Cosim



Cosimulation Report for 'hls_fp_accumulator'

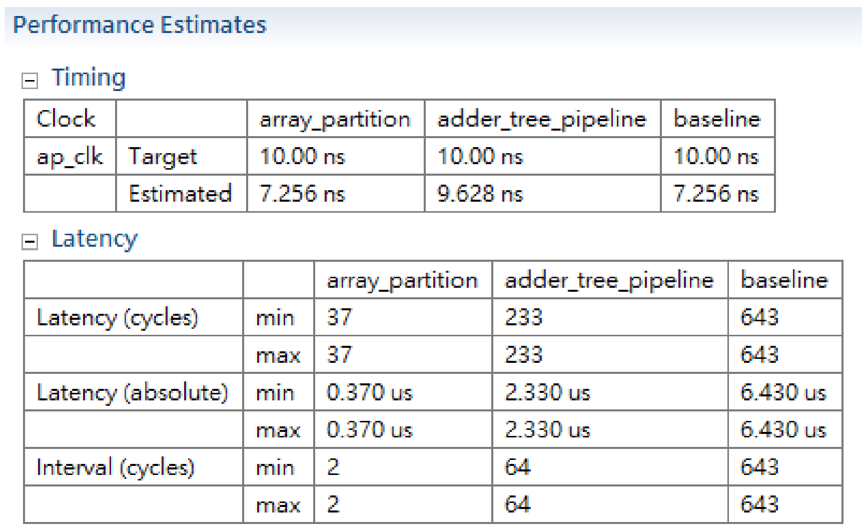
Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	37	37	37	NA	NA	NA

Export the report(.html) using the Export Wizard

比較三種作法

- 累加器使用三者之中最少的資源，但 latency 也最長
- 使用 Array Partition 後，相較於原本 adder tree 的做法，加速了 **6.3 倍**，**interval 更是縮短了 32 倍**



Performance Estimates

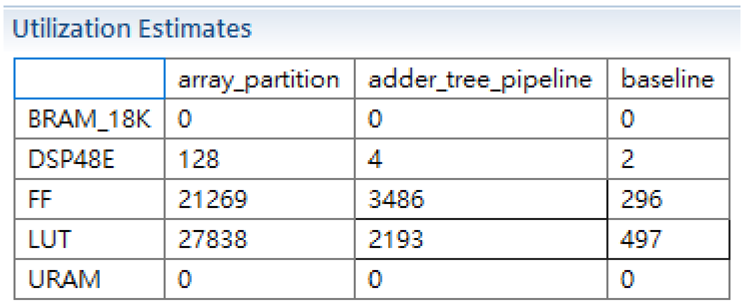
Timing

Clock		array_partition	adder_tree_pipeline	baseline
ap_clk	Target	10.00 ns	10.00 ns	10.00 ns
	Estimated	7.256 ns	9.628 ns	7.256 ns

Latency

		array_partition	adder_tree_pipeline	baseline
Latency (cycles)	min	37	233	643
	max	37	233	643
Latency (absolute)	min	0.370 us	2.330 us	6.430 us
	max	0.370 us	2.330 us	6.430 us
Interval (cycles)	min	2	64	643
	max	2	64	643

- 可以看出累加器(baseline)只用到兩個 DSP，總體使用資源最少
- Array partition 使用了 128 DSP，跟許多 FF, LUT 來存放 partial sum

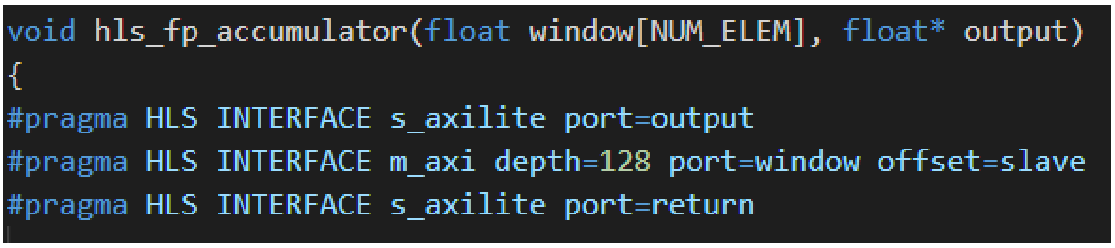

Utilization Estimates

	array_partition	adder_tree_pipeline	baseline
BRAM_18K	0	0	0
DSP48E	128	4	2
FF	21269	3486	296
LUT	27838	2193	497
URAM	0	0	0

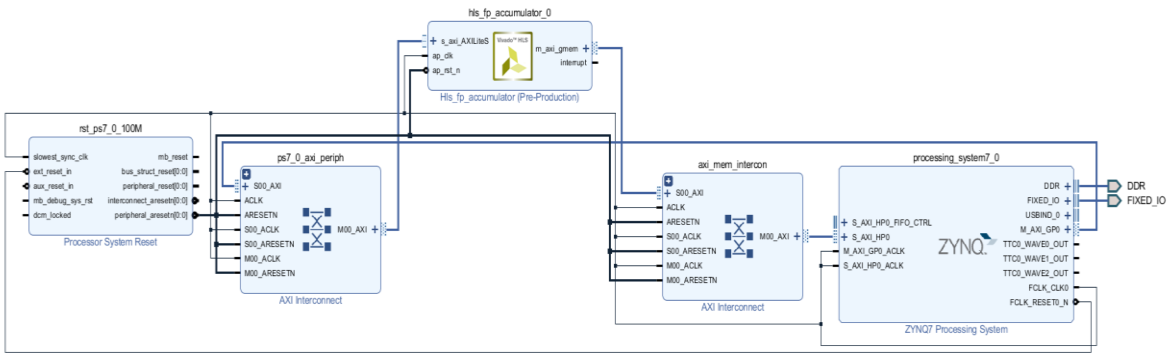
總結:
累加器與加法樹是速度與面積的 trade-off，可以根據應用選擇相應的計算總和的方式。

3. System level bring up (Pynq)
原先的 source code 中只有 HLS 的實作，我另外**自己實作**了 host program 以及相關的 I/O 介面

I/O 介面
- Input: float window[NUM_ELEM]使用 m_axi 介面
- Output: 使用 AXI_Lite 介面傳輸
- Control: 使用 AXI_Lite 介面傳輸

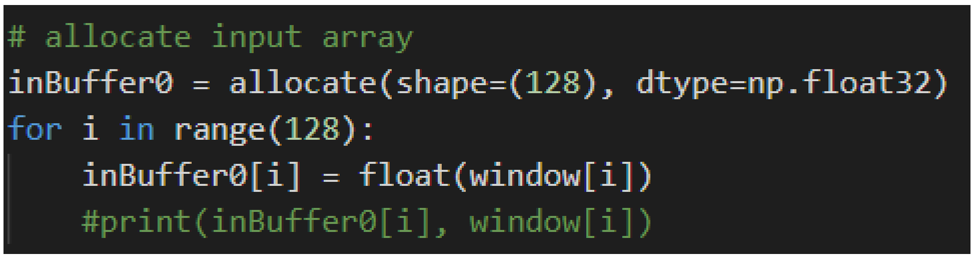

```
void hls_fp_accumulator(float window[NUM_ELEM], float* output)
{
#pragma HLS INTERFACE s_axilite port=output
#pragma HLS INTERFACE m_axi depth=128 port=window offset=slave
#pragma HLS INTERFACE s_axilite port=return
```

Vivado Block Diagram



Host program

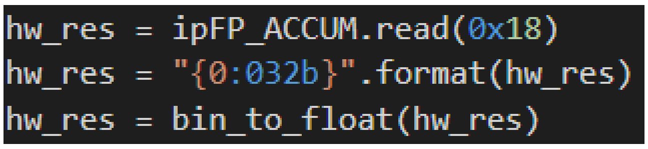
- 使用與 Lab2 一樣的 MAXI 介面
- 另外因為我們處理的是 float 的計算，在 python 中 float 變數是 64bit，所以要先透過 numpy 進行 data type 的轉換(float → float32)再送入 inBuffer



```
# allocate input array
inBuffer0 = allocate(shape=(128), dtype=np.float32)
for i in range(128):
    inBuffer0[i] = float(window[i])
    #print(inBuffer0[i], window[i])
```

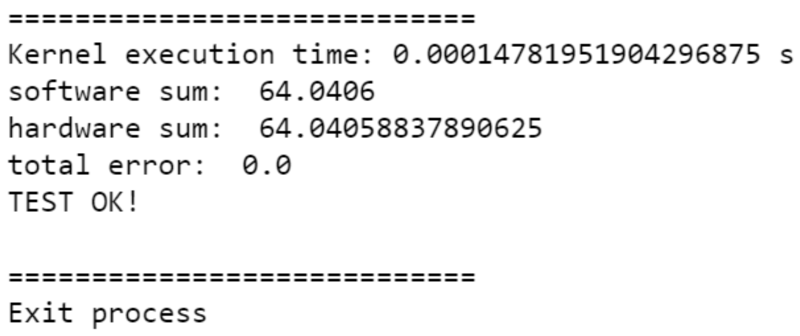
- 透過 axilite 介面讀出來的值 python 會把他當作 int32，所以要做一些 data type 的轉換才能讓 host program 成功得到正確的 output。

  Int32 → binary string → float32



```
hw_res = ipFP_ACCUM.read(0x18)
hw_res = "{0:032b}".format(hw_res)
hw_res = bin_to_float(hw_res)
```

- 執行結果



```
============================
Kernel execution time: 0.00014781951904296875 s
software sum:  64.0406
hardware sum:  64.04058837890625
total error:  0.0
TEST OK!

============================
Exit process
```

4. Github submit