pp4fpga Prefix Sum and Histogram report

電子所碩一 R09943019 李唐

System introduction

Prefix Sum

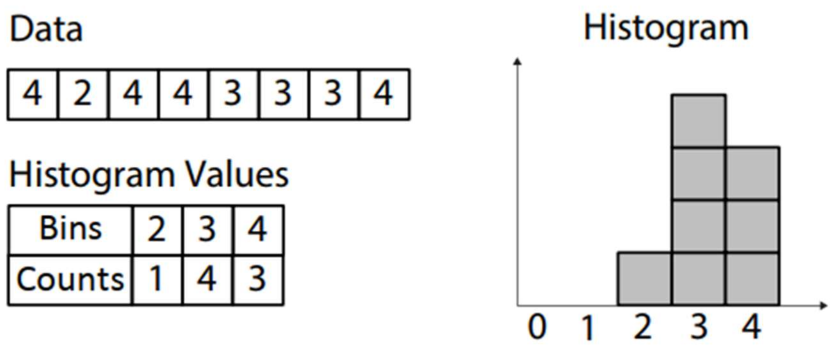
Cumulative sum of a sequence of numbers:

$$out_n = out_{n-1} + in_n$$

This recurrence equation limits the parallelism and throughput of the design. We need to carefully design and rewrite the C code in order to achieve pipeline throughput as much as possible.

Histogram

Probability distribution of a discrete signal. Count the number of times each value appears in the sequence.



Bins	2	3	4
Counts	1	4	3

- Prefix sum only performs one accumulation, while in histogram we have to compute one accumulation for each bin.
- In prefix sum we need to add input value to accumulation each time, while in histogram we only add 1 to each accumulation.

In the design of histogram, we would encounter similar problem in pipeline throughput. Moreover, some dependency issues need to be addressed.

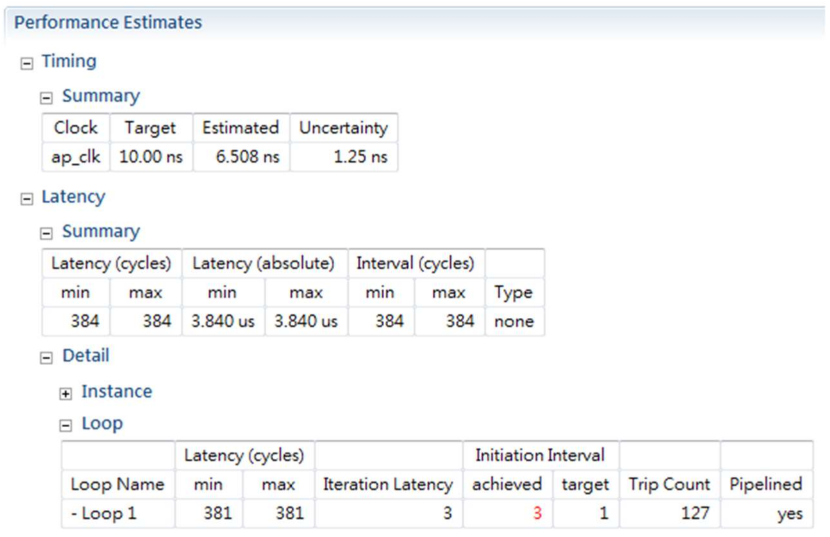
Screen dump and Observations

Prefix Sum

- **Solution 1: Baseline implementation**

```
void prefixsum(int in[SIZE], int out[SIZE]) {
    out[0] = in[0];
    for(int i=1; i < SIZE; i++) {
#pragma HLS PIPELINE
        out[i] = out[i-1] + in[i];
    }
}
```

- Synthesis report/RTL Co-simulation



Performance Estimates

⊟ Timing

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	6.508 ns	1.25 ns

⊟ Latency

⊟ Summary

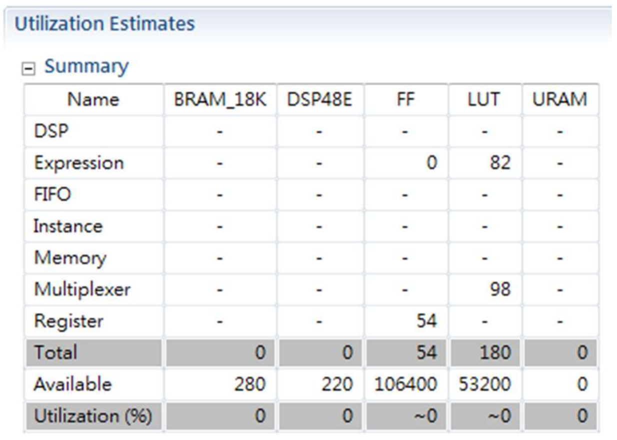
Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
384	384	3.840 us	3.840 us	384	384	none

⊟ Detail

⊞ Instance

⊟ Loop

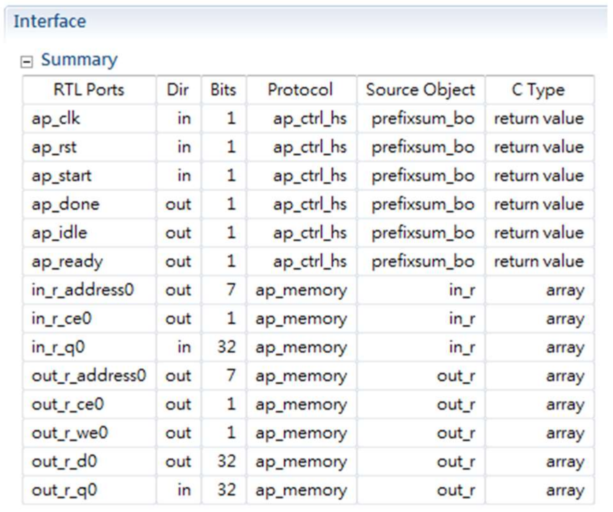
	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	381	381	3	3	1	127	yes



Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	82	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	98	-
Register	-	-	54	-	-
Total	0	0	54	180	0
Available	280	220	106400	53200	0
Utilization (%)	0	0	~0	~0	0



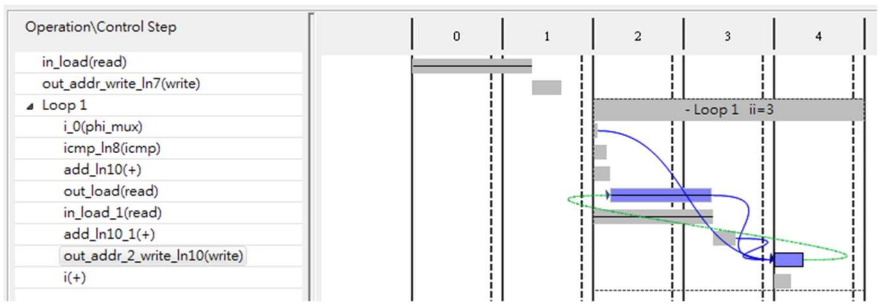
Interface

⊟ Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	prefixsum_bo	return value
ap_rst	in	1	ap_ctrl_hs	prefixsum_bo	return value
ap_start	in	1	ap_ctrl_hs	prefixsum_bo	return value
ap_done	out	1	ap_ctrl_hs	prefixsum_bo	return value
ap_idle	out	1	ap_ctrl_hs	prefixsum_bo	return value
ap_ready	out	1	ap_ctrl_hs	prefixsum_bo	return value
in_r_address0	out	7	ap_memory	in_r	array
in_r_ce0	out	1	ap_memory	in_r	array
in_r_q0	in	32	ap_memory	in_r	array
out_r_address0	out	7	ap_memory	out_r	array
out_r_ce0	out	1	ap_memory	out_r	array
out_r_we0	out	1	ap_memory	out_r	array
out_r_d0	out	32	ap_memory	out_r	array
out_r_q0	in	32	ap_memory	out_r	array

From the synthesis result, we find that the pipeline to the inner loop only achieves II=3 instead of II=1, which leads to a long latency.

- Analysis perspective



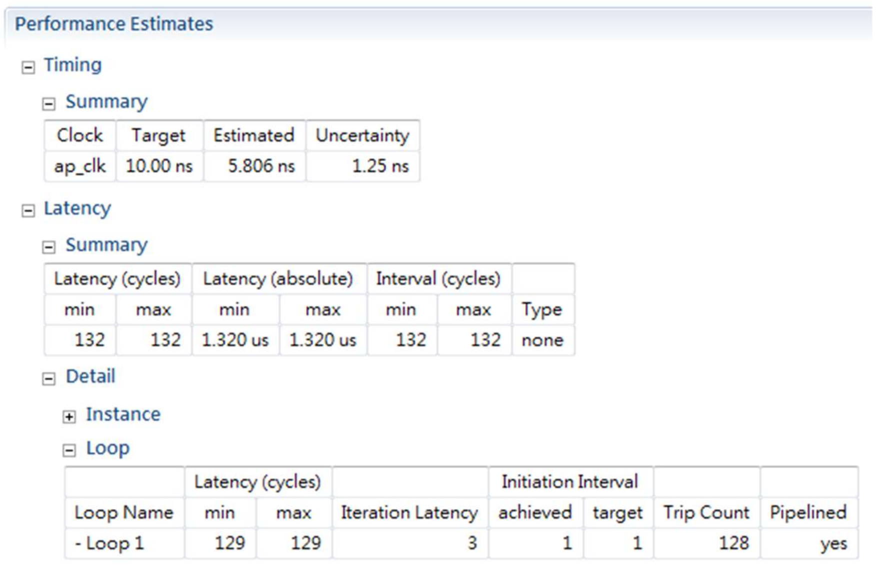
From the above timeline, we can observe that the read operation from array out[i-1] requires 2 cycles while the write operation into array out[i] requires another cycle. Also, the addition with in[i] can be performed in the same cycle as soon as out[i-1] value is read out.

Obviously, there exists **read after write (RAW)** dependency between inter loop iterations. If the current iteration is i (write to out[i]), the next iteration i+1 would read from out[i] which would lead to confliction when II=1,2 due to the dependency.

- **Solution 2: introduce local variable for holding accumulation**

```
void prefixsum(int in[SIZE], int out[SIZE]) {
    int A = in[0];
    for(int i=0; i < SIZE; i++) {
#pragma HLS PIPELINE
        A = A + in[i];
        out[i] = A;
    }
}
```

- Synthesis report/RTL Co-simulation



Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	5.806 ns	1.25 ns

Latency

Summary

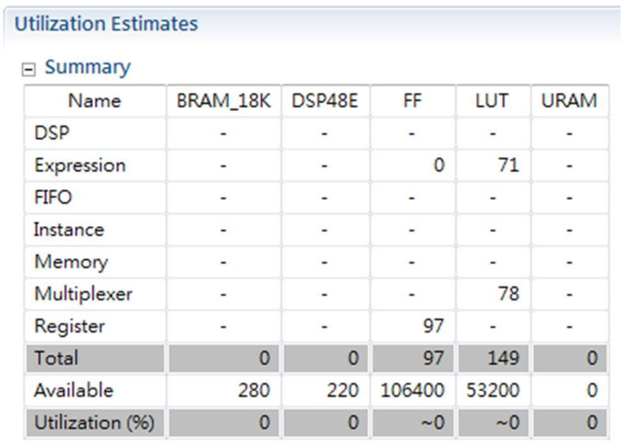
Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
132	132	1.320 us	1.320 us	132	132	none

Detail

Instance

Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	129	129	3	1	1	128	yes

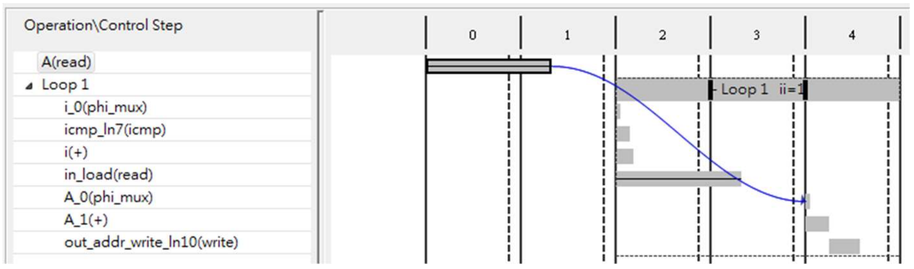

Utilization Estimates

Summary

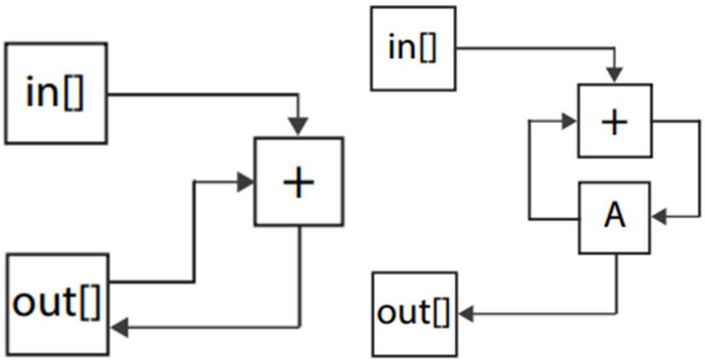
Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	71	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	78	-
Register	-	-	97	-	-
Total	0	0	97	149	0
Available	280	220	106400	53200	0
Utilization (%)	0	0	~0	~0	0

From the synthesis result, we find that the pipeline to the inner loop can now achieves II=1, which greatly improves the overall latency.

- Analysis perspective



If we place the accumulation result in the local buffer A, then we can keep doing accumulation on each cycle without dependency on array out[i].

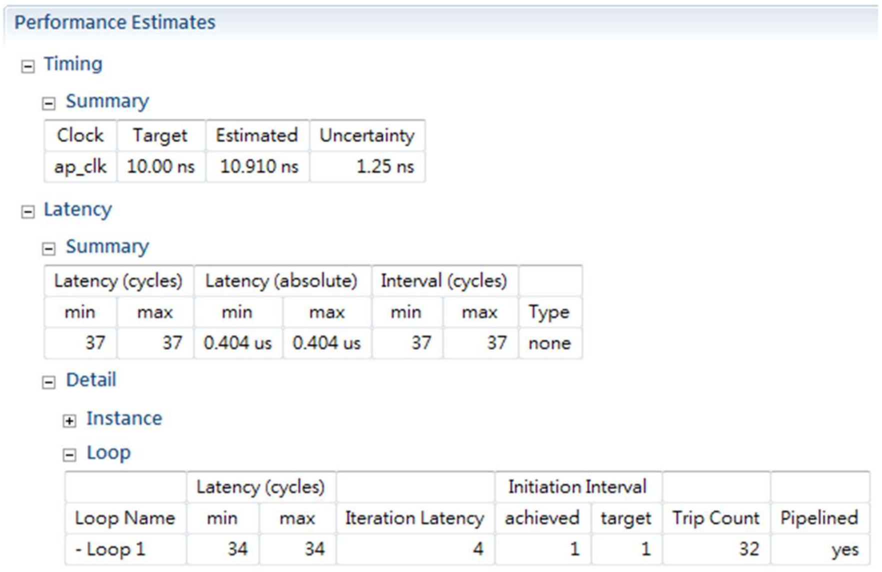


- **Solution 3: unrolling and array partition**

```
void prefixsum(int in[SIZE], int out[SIZE]) {
#pragma HLS ARRAY_PARTITION variable=out cyclic factor=4 dim=1
#pragma HLS ARRAY_PARTITION variable=in cyclic factor=4 dim=1
    int A = in[0];
    for(int i=0; i < SIZE; i++) {
#pragma HLS UNROLL factor=4
#pragma HLS PIPELINE
```

```
            A = A + in[i];
            out[i] = A;
        }
    }
```

■ Synthesis report/RTL Co-simulation



Performance Estimates

⊟ Timing

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	10.910 ns	1.25 ns

⊟ Latency

⊟ Summary

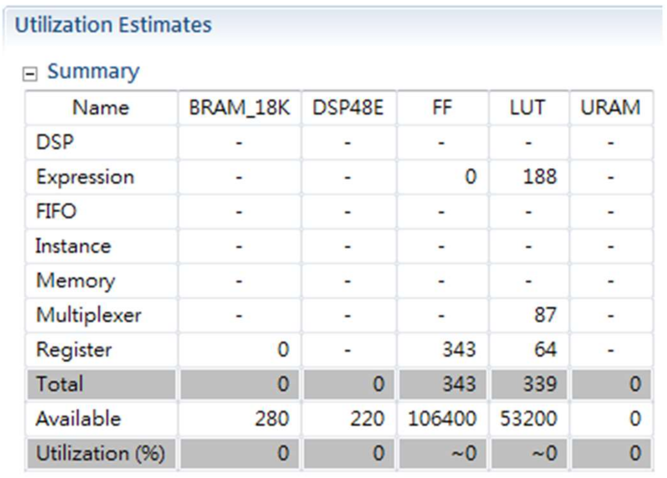
Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
37	37	0.404 us	0.404 us	37	37	none

⊟ Detail

⊞ Instance

⊟ Loop

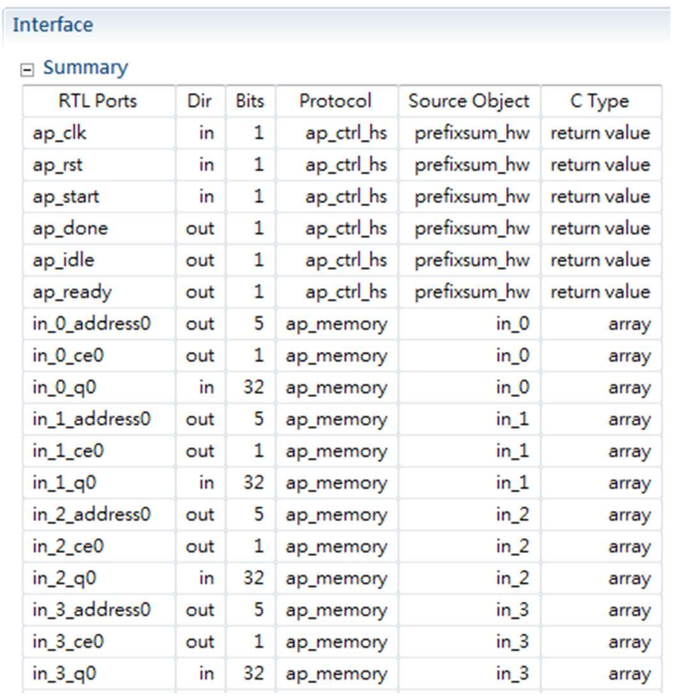
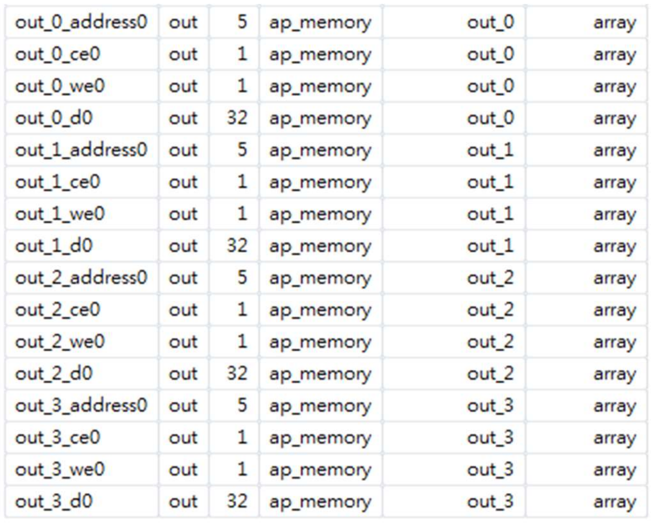
	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	34	34	4	1	1	32	yes



Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	188	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	87	-
Register	0	-	343	64	-
Total	0	0	343	339	0
Available	280	220	106400	53200	0
Utilization (%)	0	0	~0	~0	0

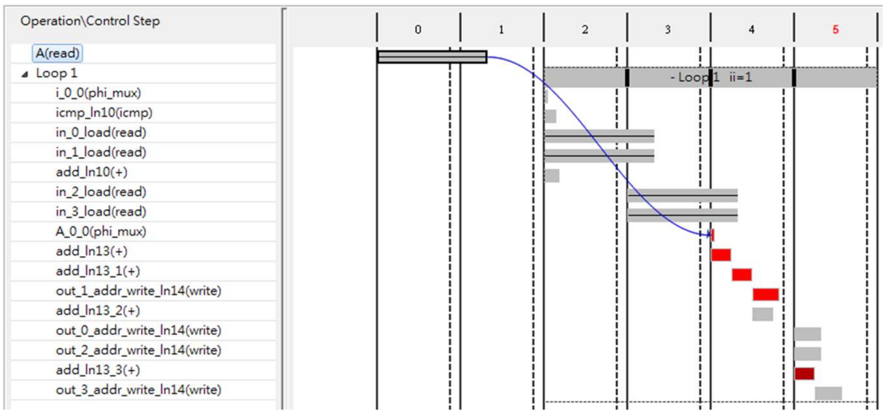
Interface

⊟ Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	prefixsum_hw	return value
ap_rst	in	1	ap_ctrl_hs	prefixsum_hw	return value
ap_start	in	1	ap_ctrl_hs	prefixsum_hw	return value
ap_done	out	1	ap_ctrl_hs	prefixsum_hw	return value
ap_idle	out	1	ap_ctrl_hs	prefixsum_hw	return value
ap_ready	out	1	ap_ctrl_hs	prefixsum_hw	return value
in_0_address0	out	5	ap_memory	in_0	array
in_0_ce0	out	1	ap_memory	in_0	array
in_0_q0	in	32	ap_memory	in_0	array
in_1_address0	out	5	ap_memory	in_1	array
in_1_ce0	out	1	ap_memory	in_1	array
in_1_q0	in	32	ap_memory	in_1	array
in_2_address0	out	5	ap_memory	in_2	array
in_2_ce0	out	1	ap_memory	in_2	array
in_2_q0	in	32	ap_memory	in_2	array
in_3_address0	out	5	ap_memory	in_3	array
in_3_ce0	out	1	ap_memory	in_3	array
in_3_q0	in	32	ap_memory	in_3	array
out_0_address0	out	5	ap_memory	out_0	array
out_0_ce0	out	1	ap_memory	out_0	array
out_0_we0	out	1	ap_memory	out_0	array
out_0_d0	out	32	ap_memory	out_0	array
out_1_address0	out	5	ap_memory	out_1	array
out_1_ce0	out	1	ap_memory	out_1	array
out_1_we0	out	1	ap_memory	out_1	array
out_1_d0	out	32	ap_memory	out_1	array
out_2_address0	out	5	ap_memory	out_2	array
out_2_ce0	out	1	ap_memory	out_2	array
out_2_we0	out	1	ap_memory	out_2	array
out_2_d0	out	32	ap_memory	out_2	array
out_3_address0	out	5	ap_memory	out_3	array
out_3_ce0	out	1	ap_memory	out_3	array
out_3_we0	out	1	ap_memory	out_3	array
out_3_d0	out	32	ap_memory	out_3	array

From the synthesis result, we find that unrolling the loop leads to shorter latency but much more resource. Moreover, in order to fully utilize the unrolling, we need to partition both the in[i] and out[i] arrays to avoid memory bandwidth limitation.

- Analysis perspective



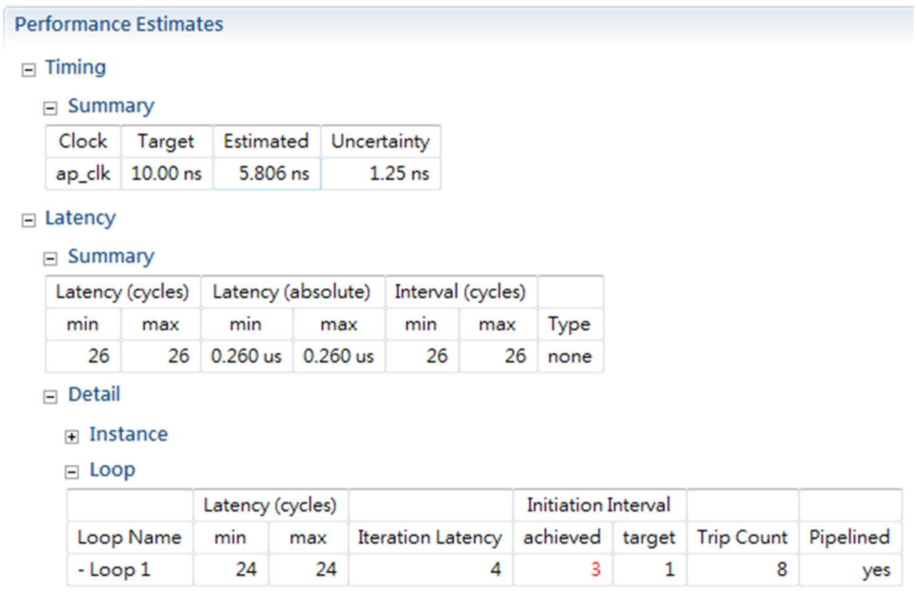
Unrolling the loop leads to longer critical path during accumulation, which can be resolved by using larger clock period. But actually in our PNYQ system, there is no timing violation.

✧ Histogram

- **Solution 1: Baseline implementation**

```
void histogram(int in[INPUT_SIZE], int hist[VALUE_SIZE]) {
    int val;
    for(int i = 0; i < INPUT_SIZE; i++) {
#pragma HLS PIPELINE
        val = in[i];
        hist[val] = hist[val] + 1;
    }
}
```

- Synthesis report/RTL Co-simulation



Performance Estimates

⊟ Timing

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	5.806 ns	1.25 ns

⊟ Latency

⊟ Summary

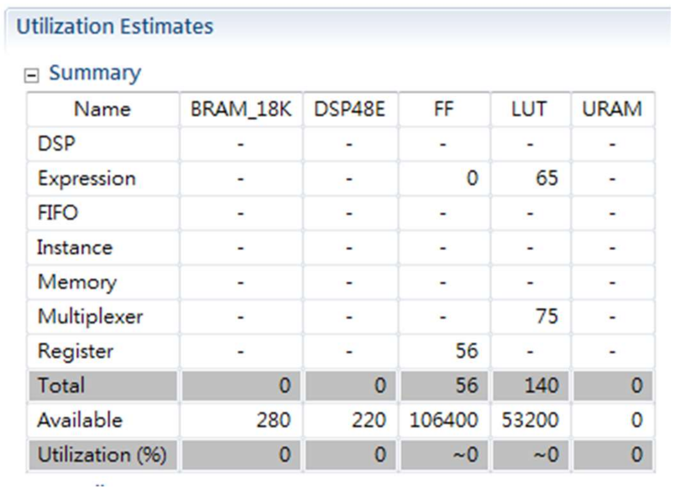
Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
26	26	0.260 us	0.260 us	26	26	none

⊟ Detail

⊞ Instance

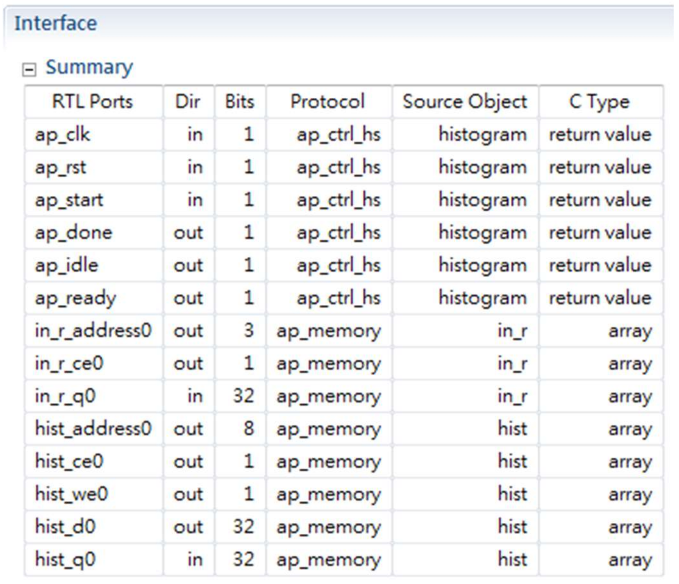
⊟ Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	24	24	4	3	1	8	yes


Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	65	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	75	-
Register	-	-	56	-	-
Total	0	0	56	140	0
Available	280	220	106400	53200	0
Utilization (%)	0	0	~0	~0	0

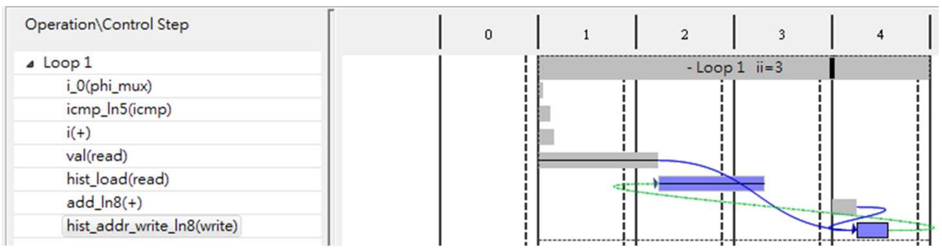

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	histogram	return value
ap_rst	in	1	ap_ctrl_hs	histogram	return value
ap_start	in	1	ap_ctrl_hs	histogram	return value
ap_done	out	1	ap_ctrl_hs	histogram	return value
ap_idle	out	1	ap_ctrl_hs	histogram	return value
ap_ready	out	1	ap_ctrl_hs	histogram	return value
in_r_address0	out	3	ap_memory	in_r	array
in_r_ce0	out	1	ap_memory	in_r	array
in_r_q0	in	32	ap_memory	in_r	array
hist_address0	out	8	ap_memory	hist	array
hist_ce0	out	1	ap_memory	hist	array
hist_we0	out	1	ap_memory	hist	array
hist_d0	out	32	ap_memory	hist	array
hist_q0	in	32	ap_memory	hist	array

From the synthesis result, we find that the pipeline to the inner loop only achieves II=3 instead of II=1, which leads to a long latency.

■ Analysis perspective



From the timeline, we can find that the same RAW dependency problem occurs in hist[val] array. That is, the confliction occurs when the val value is same between inter loop iterations.

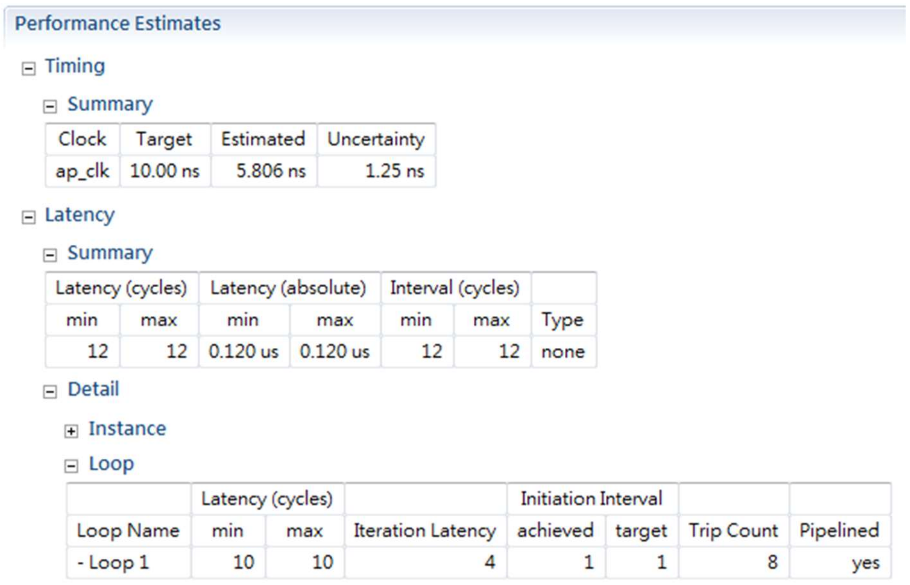
✧ **Solution 2: RAW false dependence**

void histogram_dep(int in[INPUT_SIZE], int hist[VALUE_SIZE]) {

#pragma HLS DEPENDENCE variable=hist inter RAW distance=2

```
        int val;
        int old = -1;
        for(int i = 0; i < INPUT_SIZE; i++) {
#pragma HLS PIPELINE
            val = in[i];
            assert(old != val);
            hist[val] = hist[val] + 1;
            old = val;
        }
}
```

✧ Synthesis report/RTL Co-simulation



Performance Estimates

⊟ Timing

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	5.806 ns	1.25 ns

⊟ Latency

⊟ Summary

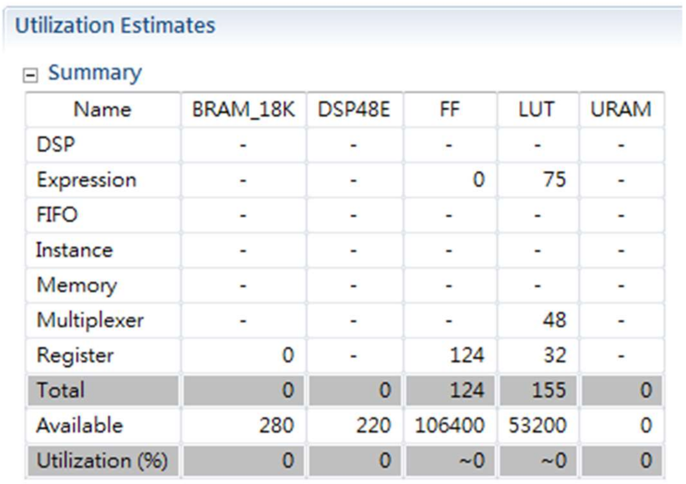
Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
12	12	0.120 us	0.120 us	12	12	none

⊟ Detail

⊞ Instance

⊟ Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	10	10	4	1	1	8	yes



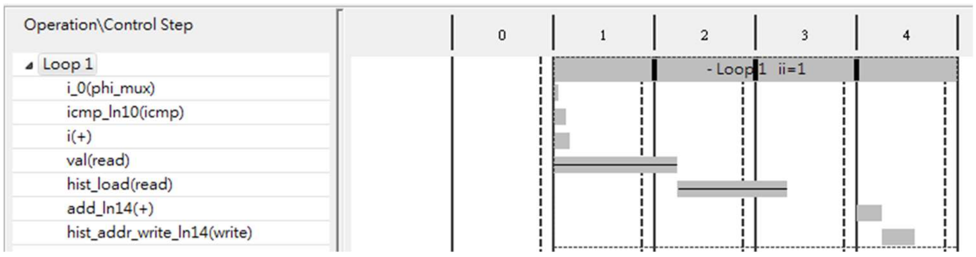
Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	75	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	48	-
Register	0	-	124	32	-
Total	0	0	124	155	0
Available	280	220	106400	53200	0
Utilization (%)	0	0	~0	~0	0

From the synthesis result, we find that the pipeline to the inner loop can now achieves II=1, which greatly improves the overall latency. (FF resource is increased due to additional register old and further pipeline)

✧ Analysis perspective



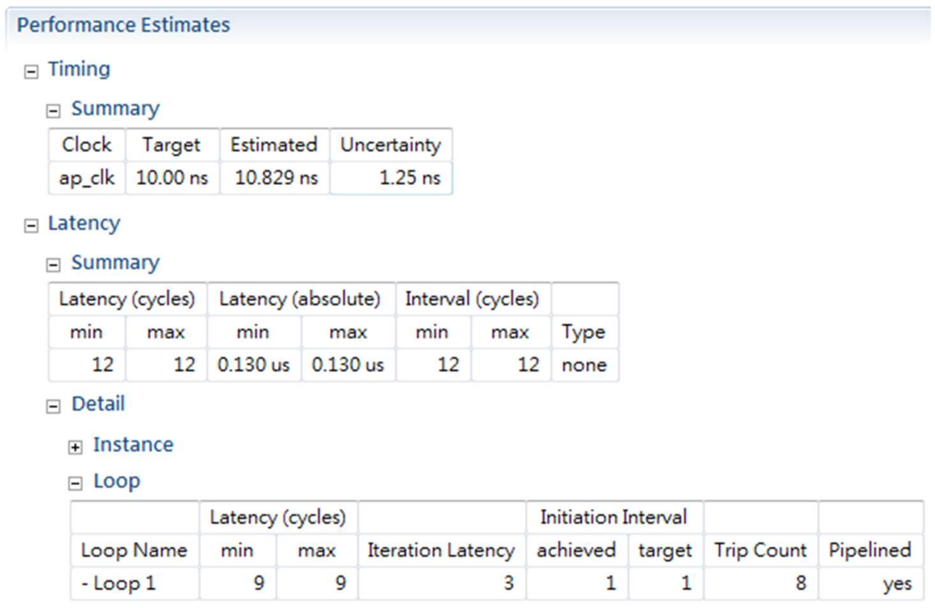
If we assume that the val value for the current iteration and the next iteration won't be the same, then II=1 can be realized logically. So we need to add the **dependence pragma** to tell the HLS that we can assure there would be false dependence to the array hist[val].

Another solution would be to fully partition the array hist[val] so that it's implemented with Flip-Flop resources. The data stored in Flip-Flop can be accessed immediately after the write operation. However, fully partition would lead to large resources utilization when the number of bin is high.

✧ **Solution 3: introduce local variable for holding previous value**

```
void histogram_opt(int in[INPUT_SIZE], int hist[VALUE_SIZE]) {
    int acc = 0;
    int i, val;
    int old = in[0];
#pragma HLS DEPENDENCE variable=hist intra RAW false
    for(i = 0; i < INPUT_SIZE; i++) {
#pragma HLS PIPELINE II=1
        val = in[i];
        if(old == val) {
            acc = acc + 1;
        } else {
            hist[old] = acc;
            acc = hist[val] + 1;
        }
        old = val;
    }
    hist[old] = acc;
}
```

✧ Synthesis report/RTL Co-simulation



Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	10.829 ns	1.25 ns

Latency

Summary

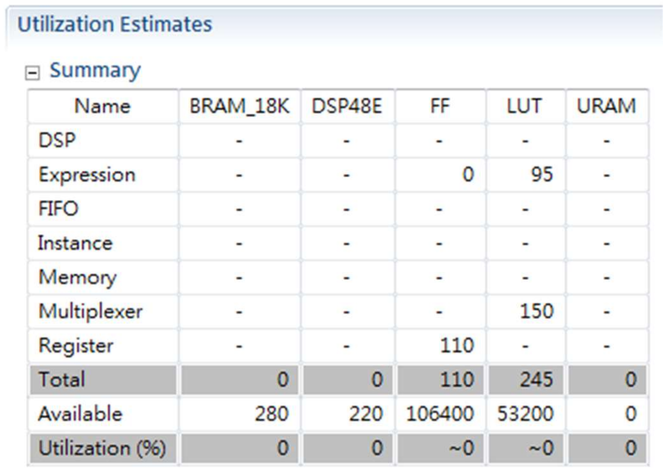
Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
12	12	0.130 us	0.130 us	12	12	none

Detail

Instance

Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	9	9	3	1	1	8	yes



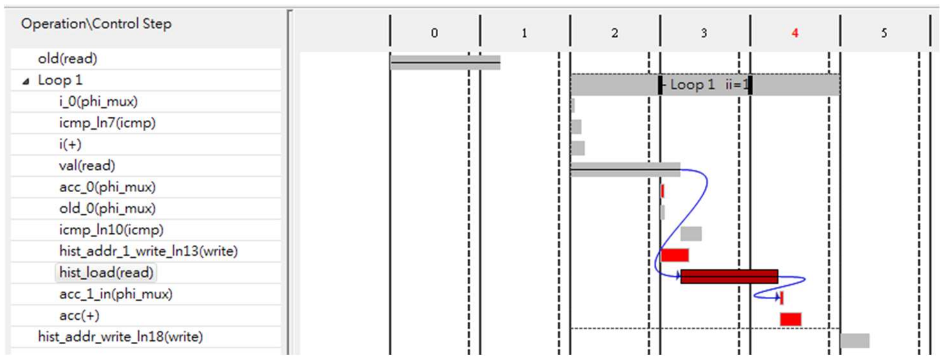
Utilization Estimates

Summary

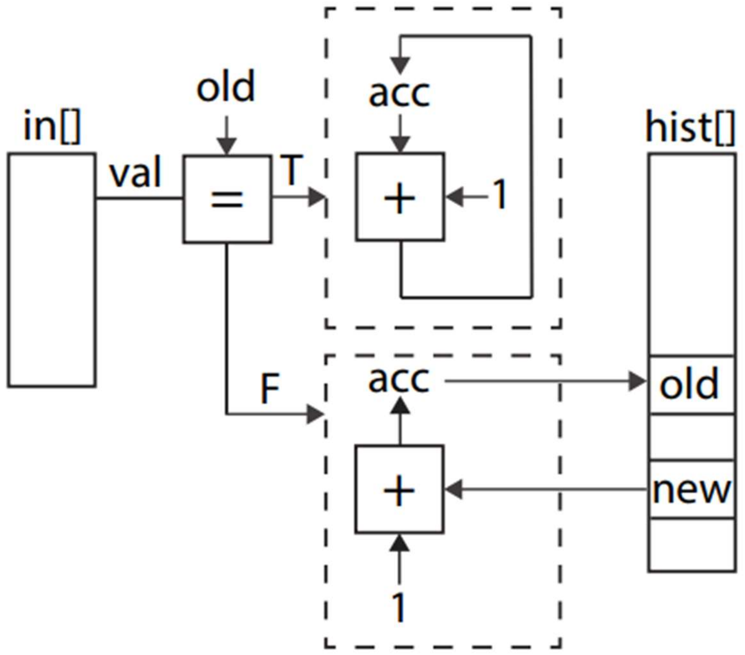
Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	95	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	150	-
Register	-	-	110	-	-
Total	0	0	110	245	0
Available	280	220	106400	53200	0
Utilization (%)	0	0	~0	~0	0

From the synthesis result, we find that the pipeline also achieves II=1 while the latency is the same as solution 2.

✧ Analysis perspective



The assumption in solution 2 may not hold for all cases. Better design introduces two variables: old and acc to hold the val and hist[val] in the previous iteration. In each iteration we check whether current val equals to old. If it is, then we do the accumulation directly in the local buffer acc. If it's not, then we store the acc back to the BRAM while read the new hist[val] from the BRAM. By rewriting the code like this, we can resolve the RAW dependency problem.



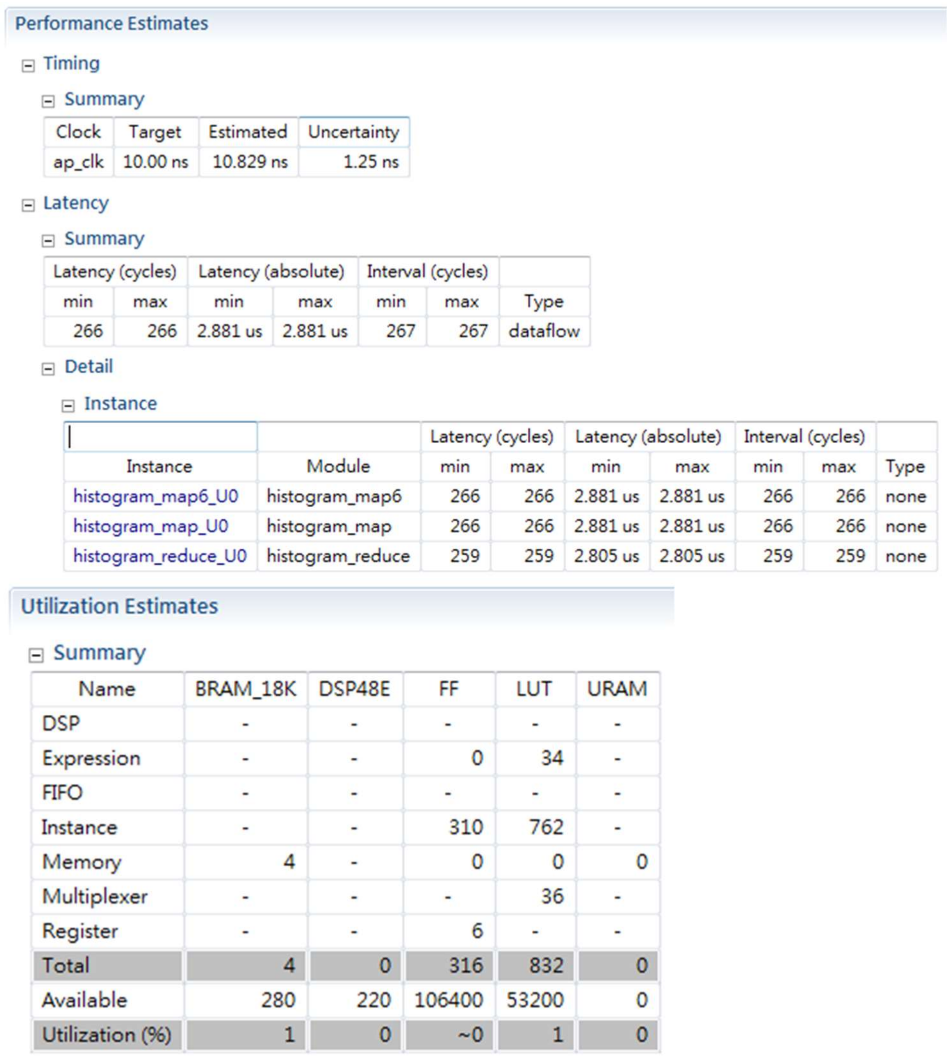
✧ **Solution 4: map-reduce implementation with dataflow directive**

```
void histogram_reduce(int hist1[VALUE_SIZE], int hist2[VALUE_SIZE],
int output[VALUE_SIZE]) {
        for(int i = 0; i < VALUE_SIZE; i++) {
#pragma HLS PIPELINE II=1
                output[i] = hist1[i] + hist2[i];
        }
}

void histogram_parallel(int inputA[INPUT_SIZE/2],
int inputB[INPUT_SIZE/2], int hist[VALUE_SIZE]){
#pragma HLS DATAFLOW
        int hist1[VALUE_SIZE];
        int hist2[VALUE_SIZE];

        histogram_opt (inputA, hist1); // function in solution 3
        histogram_opt (inputB, hist2);
        histogram_reduce(hist1, hist2, hist);
}
```

✧ Synthesis report/RTL Co-simulation



Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	10.829 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
266	266	2.881 us	2.881 us	267	267	dataflow

Detail

Instance

		Latency (cycles)		Latency (absolute)		Interval (cycles)		
Instance	Module	min	max	min	max	min	max	Type
histogram_map6_U0	histogram_map6	266	266	2.881 us	2.881 us	266	266	none
histogram_map_U0	histogram_map	266	266	2.881 us	2.881 us	266	266	none
histogram_reduce_U0	histogram_reduce	259	259	2.805 us	2.805 us	259	259	none

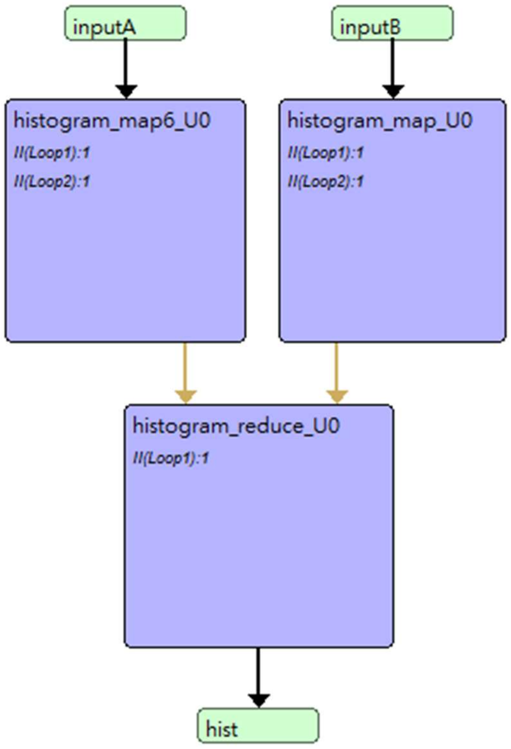
Utilization Estimates

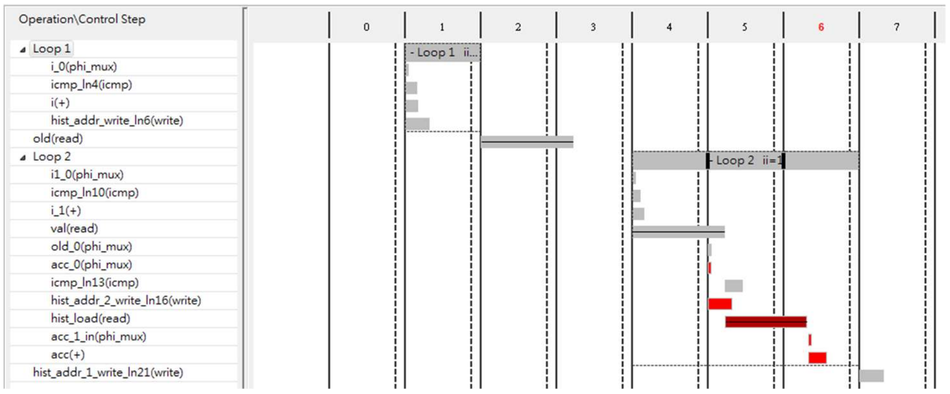
Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	34	-
FIFO	-	-	-	-	-
Instance	-	-	310	762	-
Memory	4	-	0	0	0
Multiplexer	-	-	-	36	-
Register	-	-	6	-	-
Total	4	0	316	832	0
Available	280	220	106400	53200	0
Utilization (%)	1	0	~0	1	0

From the synthesis result, we find that the latency becomes much longer compared with previous solutions.

✧ Analysis perspective





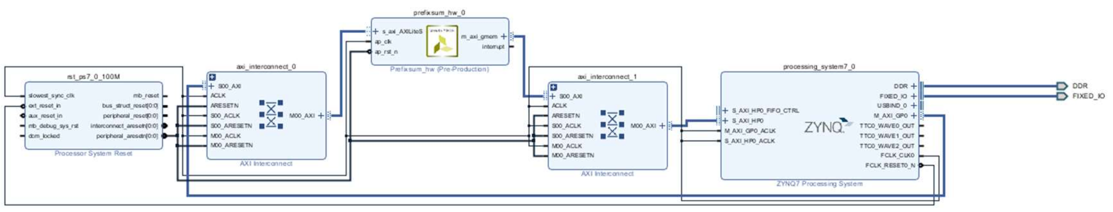
The design creates two Processing Elements (PE) to independently work on two partitioned input sets (mapping), which increases the processing throughput. Then the outputs from these two instances are combined in the same histogram array as the final output (reducing).

The reasons behind longer latency are possibly:

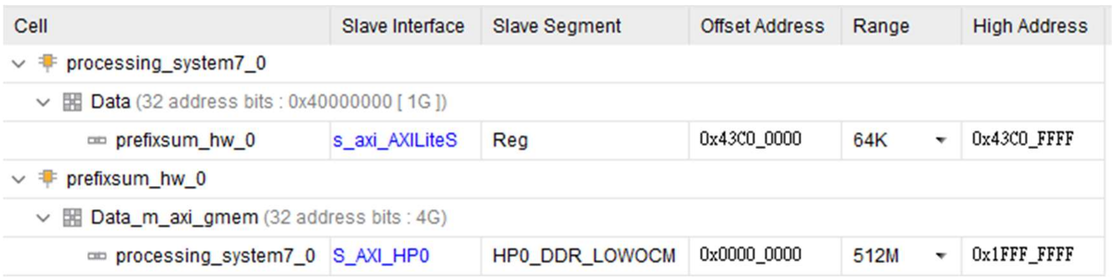
- In each PE, the histogram has to be initialized before computation, which requires 256 cycles in total since the range of values consider is 256 (much larger than input size).
- In the reducing process, we need 256 cycles to combine the outputs from PEs to generate the combined histogram.

Dataflow directive makes the function operations partially overlap, which reduces the latency and increase the overall throughput. Nevertheless, the bottleneck in mapping process still makes the total latency longer compared with previous solutions. This design would perform better than other solutions only when the testing dataset is large enough.

- ZYNQ implementation
  - Block diagram: AXI4-Master interface

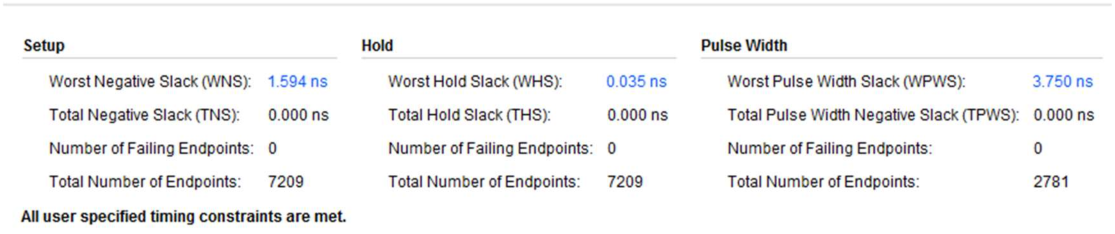


  - Address mapping



Cell	Slave Interface	Slave Segment	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
prefixsum_hw_0	s_axi_AXILiteS	Reg	0x43C0_0000	64K	0x43C0_FFFF
prefixsum_hw_0					
Data_m_axi_gmem (32 address bits : 4G)					
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF

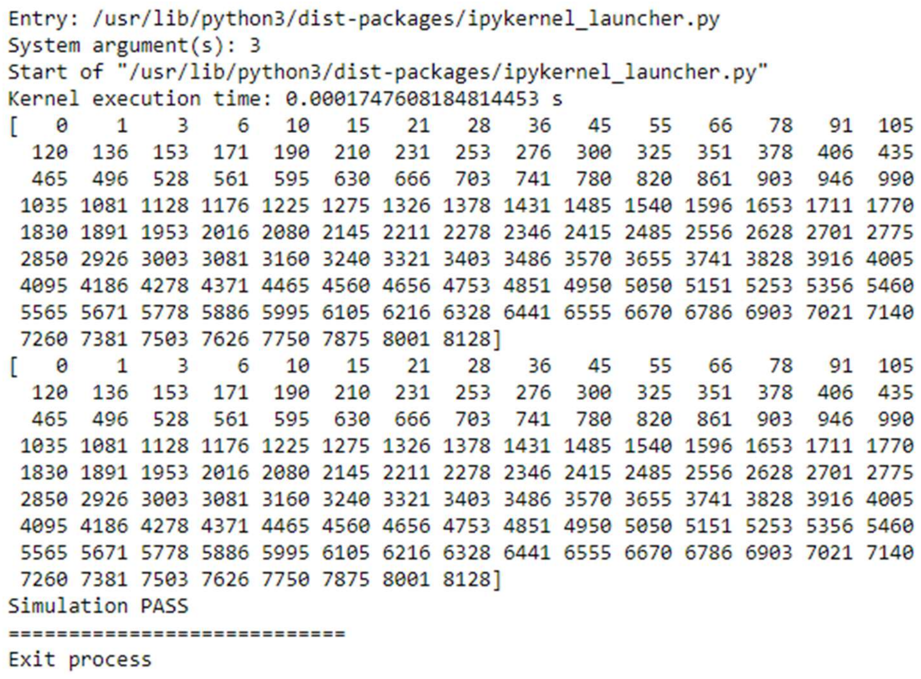
  - No timing violation



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.594 ns	Worst Hold Slack (WHS):	0.035 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	7209	Total Number of Endpoints:	7209	Total Number of Endpoints:	2781

All user specified timing constraints are met.

  - PYNQ result



```
Entry: /usr/lib/python3/dist-packages/ipykernel_launcher.py
System argument(s): 3
Start of "/usr/lib/python3/dist-packages/ipykernel_launcher.py"
Kernel execution time: 0.00017476081848144453 s
[   0    1    3    6   10   15   21   28   36   45   55   66   78   91  105
  120  136  153  171  190  210  231  253  276  300  325  351  378  406  435
  465  496  528  561  595  630  666  703  741  780  820  861  903  946  990
 1035 1081 1128 1176 1225 1275 1326 1378 1431 1485 1540 1596 1653 1711 1770
 1830 1891 1953 2016 2080 2145 2211 2278 2346 2415 2485 2556 2628 2701 2775
 2850 2926 3003 3081 3160 3240 3321 3403 3486 3570 3655 3741 3828 3916 4005
 4095 4186 4278 4371 4465 4560 4656 4753 4851 4950 5050 5151 5253 5356 5460
 5565 5671 5778 5886 5995 6105 6216 6328 6441 6555 6670 6786 6903 7021 7140
 7260 7381 7503 7626 7750 7875 8001 8128]
[   0    1    3    6   10   15   21   28   36   45   55   66   78   91  105
  120  136  153  171  190  210  231  253  276  300  325  351  378  406  435
  465  496  528  561  595  630  666  703  741  780  820  861  903  946  990
 1035 1081 1128 1176 1225 1275 1326 1378 1431 1485 1540 1596 1653 1711 1770
 1830 1891 1953 2016 2080 2145 2211 2278 2346 2415 2485 2556 2628 2701 2775
 2850 2926 3003 3081 3160 3240 3321 3403 3486 3570 3655 3741 3828 3916 4005
 4095 4186 4278 4371 4465 4560 4656 4753 4851 4950 5050 5151 5253 5356 5460
 5565 5671 5778 5886 5995 6105 6216 6328 6441 6555 6670 6786 6903 7021 7140
 7260 7381 7503 7626 7750 7875 8001 8128]
Simulation PASS
============================
Exit process
```

- GitHub submission

  https://github.com/nodetibylno/MSoC_self_paced_learning

- Reference

1. R. Kastner, J. Matai, and S. Neuendorffer , Parallel Programming for FPGAs, arXiv , 2018.