LAB_A - Interface Synthesis

Student ID: R11922029

Name: 吳泓毅

Department: NTU CSIE

github

LAB1: Review the function return and block-level protocols

Note: If you follow the instructions in Lab1, you will get different results with the tutorial. I considered that it is caused by updating of vitis version, which makes optimization that the code become combinational logic. So I modified the code to generate the result that this Lab wanted to show

Origin code

```
#include "adders.h"

int adders(int in1, int in2, int in3) {

// Prevent IO protocols on all input ports
#pragma HLS INTERFACE ap_none port=in3
#pragma HLS INTERFACE ap_none port=in2
#pragma HLS INTERFACE ap_none port=in1


    int sum;

    sum = in1 + in2 + in3;

    return sum;

}
```

Modified code

```
#include "adders.h"

int adders(int in1, int in2, int in3,int in4,int in5,int in6,int in7) {
#pragma HLS INTERFACE mode=ap_ctrl_none port=return

// Prevent IO protocols on all input ports
#pragma HLS INTERFACE ap_none port=in1
#pragma HLS INTERFACE ap_none port=in2
#pragma HLS INTERFACE ap_none port=in3
#pragma HLS INTERFACE ap_none port=in4
#pragma HLS INTERFACE ap_none port=in5
#pragma HLS INTERFACE ap_none port=in6
#pragma HLS INTERFACE ap_none port=in7


    int sum;

    sum = in1 + in2 + in3 + in4 + in5 + in6 + in7;

    return sum;

}
```

Function of code

A simple adder

Synthesis of code

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
1	1	3.250 ns	3.250 ns	2	2	no

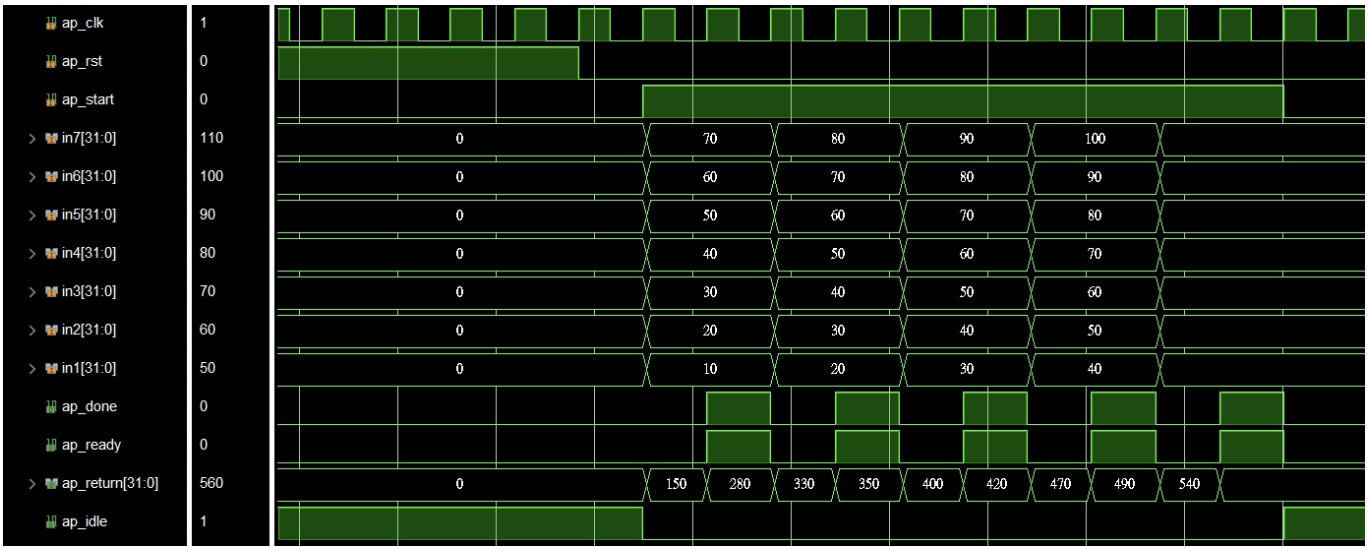
Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	adders	return value
ap_rst	in	1	ap_ctrl_hs	adders	return value
ap_start	in	1	ap_ctrl_hs	adders	return value
ap_done	out	1	ap_ctrl_hs	adders	return value
ap_idle	out	1	ap_ctrl_hs	adders	return value
ap_ready	out	1	ap_ctrl_hs	adders	return value
ap_return	out	32	ap_ctrl_hs	adders	return value
in1	in	32	ap_none	in1	scalar
in2	in	32	ap_none	in2	scalar
in3	in	32	ap_none	in3	scalar
in4	in	32	ap_none	in4	scalar
in5	in	32	ap_none	in5	scalar
in6	in	32	ap_none	in6	scalar
in7	in	32	ap_none	in7	scalar

Observation:

- We can see that ap_clk and ap_rst are added, it represents that the design takes more than one clock cycle to complete.
- Block-level I/O protocol has been added to control the RTL design: ap_start, ap_done, ap_idle, and ap_ready.
- This design has a 32-bit output port for function return ap_return.

Waveform

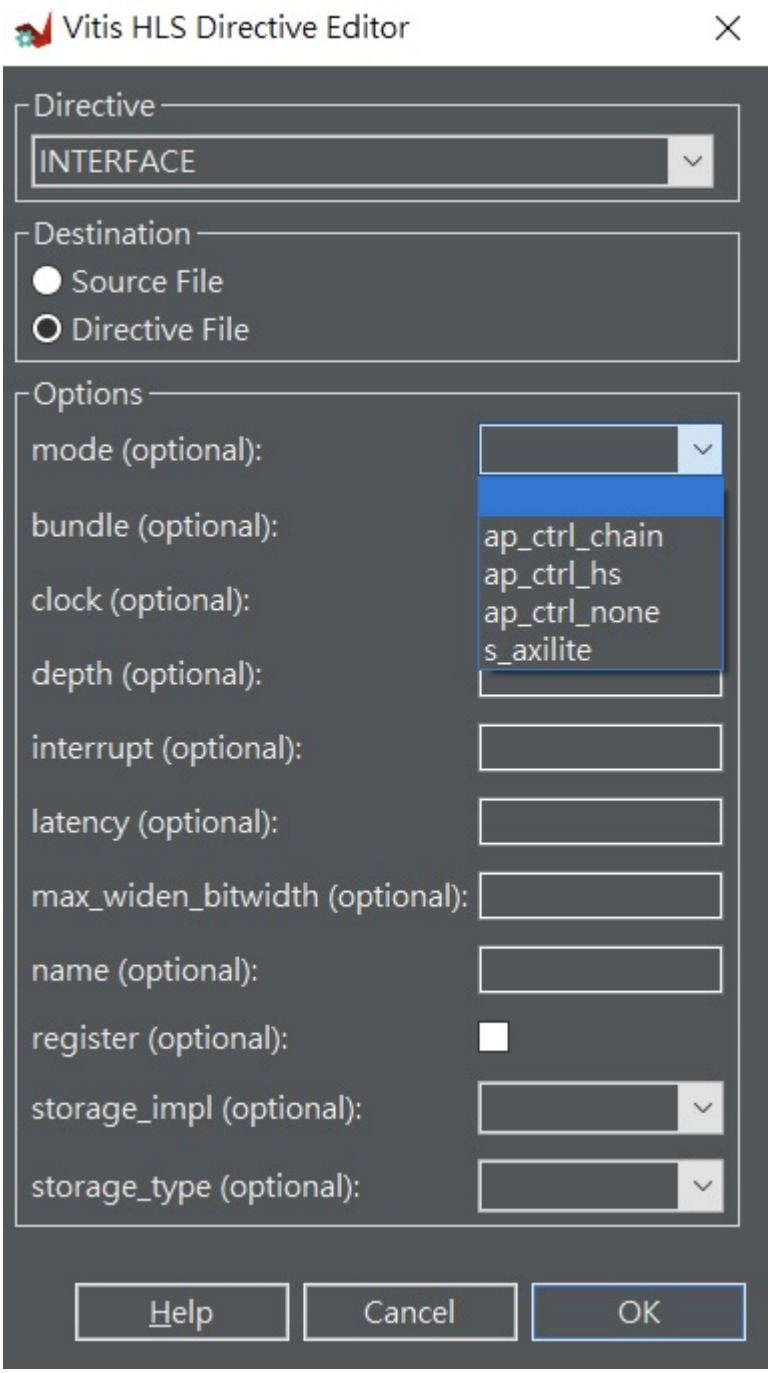


Observation:

- The design does not start until ap_start is set to 1
- The design indicates it is no longer idle by setting port ap_idle low.
- Output signal ap_ready goes high to indicate the design is ready for new inputs on the next clock.
- Output signal ap_done indicates when the design is finished and that the value on output port ap_return is valid.
- Because ap_start is held high, the next transaction starts on the next clock cycle.
- ap_done and ap_ready is set to 1 at the same time, which means it's a non-pipeline design.

Block-level protocols

Now we create a new solution and try to add other block-level protocols. We can see that there are 4 modes for block-level protocol



Explanation of each block-level protocol

- ap_ctrl_none: No block-level I/O control protocol.
- ap_ctrl_hs: The block-level I/O control handshake protocol. This protocol is the default protocol.
- ap_ctrl_chain: The block-level I/O protocol for control chaining. This I/O protocol is primarily used for chaining pipelined blocks together.
- s_axilite: May be applied in addition to ap_ctrl_hs or ap_ctrl_chain to implement the block-level I/O protocol as an AXI Slave Lite interface in place of separate discrete I/O ports.

Specify block-level protocol

using

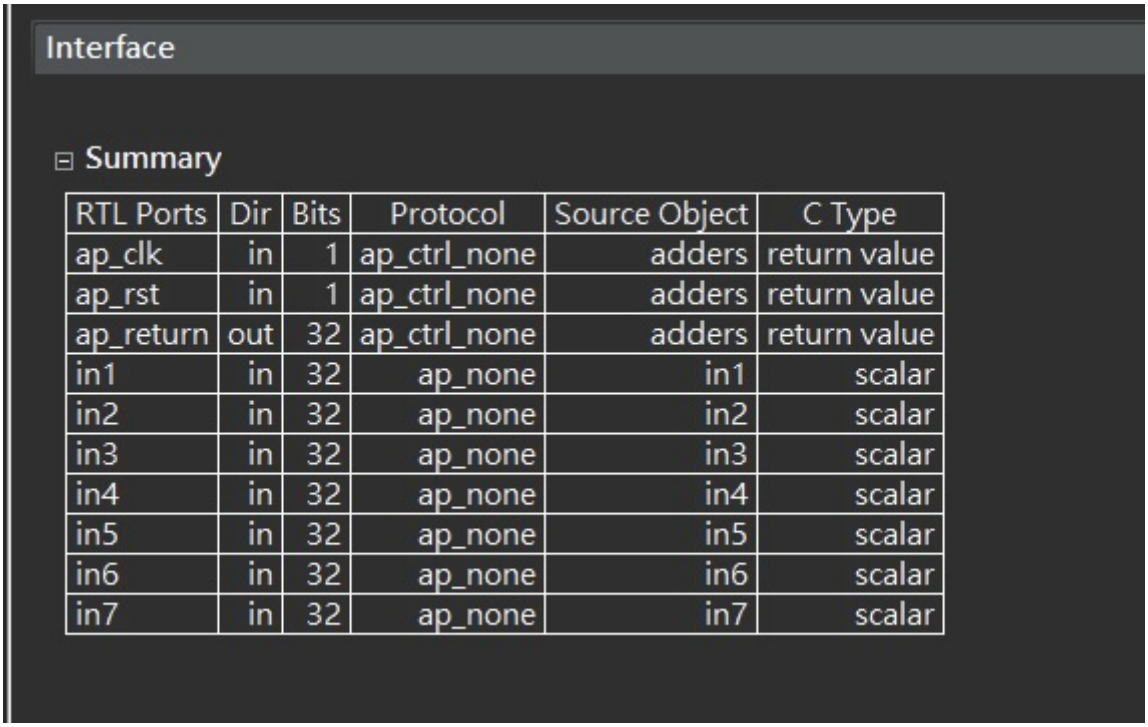
```
#pragma HLS INTERFACE block-level-protocol port=return
```

ap_ctrl_none

We choose ap_ctrl_none to specifiy that we don't use any block-level protocol.

```
*****************************************************************************/
#include "adders.h"

int adders(int in1, int in2, int in3,int in4,int in5,int in6,int in7) {
#pragma HLS INTERFACE mode=ap_ctrl_none port=return


// Prevent IO protocols on all input ports
#pragma HLS INTERFACE ap_none port=in1
#pragma HLS INTERFACE ap_none port=in2
#pragma HLS INTERFACE ap_none port=in3
#pragma HLS INTERFACE ap_none port=in4
#pragma HLS INTERFACE ap_none port=in5
#pragma HLS INTERFACE ap_none port=in6
#pragma HLS INTERFACE ap_none port=in7


    int sum;

    sum = in1 + in2 + in3 + in4 + in5 + in6 + in7;

    return sum;

}
```

Synthesis of code



RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_none	adders	return value
ap_rst	in	1	ap_ctrl_none	adders	return value
ap_return	out	32	ap_ctrl_none	adders	return value
in1	in	32	ap_none	in1	scalar
in2	in	32	ap_none	in2	scalar
in3	in	32	ap_none	in3	scalar
in4	in	32	ap_none	in4	scalar
in5	in	32	ap_none	in5	scalar
in6	in	32	ap_none	in6	scalar
in7	in	32	ap_none	in7	scalar

Obervation:

- in summary of the design, we can see that block-level I/O protocol(ap_start, ap_done, ap_idle, and ap_ready) aren't added to the design

Co-simulation with ap_ctrl_none

When do co-simulation, it will fail and show error message below. This is because RTL CoSimulation feature requires a block-level I/O protocol to sequence the test bench and RTL design for CoSimulation automatically.

ERROR: [COSIM 212-345] Cosim only supports the following 'ap_ctrl_none' designs: (1) combinational designs; (2) pipelined design with II of 1; (3) designs with array streaming or hls_stream or AXI4 stream ports. ERROR: [COSIM 212-5] *** C/RTL co-simulation file generation failed. *** ERROR: [COSIM 212-4] *** C/RTL co-simulation finished: FAIL ***

Reason for modified code

According to error message above, it shows that cosim can supports ap_ctrl_none design with combinational designs. And the origin code can pass co-simulation with ap_ctrl_none block level protocol. So I considered the origin code is optimized to combinatonal logic in Vitis HLS 2022.1

LAB2: Understand the default I/O protocol for ports and learn how to select an I/O protocol.

Origin code

```
*************************************************************************/
#include "adders_io.h"

void adders_io(int in1, int in2, int *in_out1) {

    *in_out1 = in1 + in2 + *in_out1;


}
```

Function of code

A simple adder

Compare to Lab1

We don't have function return value in Lab2, but we pass the result through function argument *in_out1. Because in_out exists LHS and RHS of function statements, so in_out is implemented as separate input and output ports.

Port-level I/O protocol in Vitis 2022.1

- Port-Level I/O: No protocol
  ap_none: simplest interface type and has no other signals associated with it. Neither the input nor output data signals have associated control ports that indicate when data is read or written.

- Port-Level I/O: Wire Handshakes
  ap_vld: an valid port for input or output
  ap_ack: an acknowlegement port for input or output
  ap_hs: using both ap_vld and ap_ack, can be applied to arrays that are read or written in sequential order.
  ap_ovld: for use with in-out arguments. When the in-out is split into separate input and output ports, mode ap_none is applied to the input port and ap_vld applied to the output port. This is the default for pointer arguments that are both read and written.

- Port-Level I/O: Memory Interface Protocol
  ap_memory: default interface of array arguments, interface appears as discrete ports.
  bram: Almost identical to ap_memory, the only difference is the way Vivado IP integrator shows the blocks.
  ap_fifo: When the design requires access to a memory element and the access is always performed in a sequential manner, that is, no random access is required. We use ap_fifo.

Port-level I/O protocol in Lab2

Synthesis of code without adding any directive

Interface

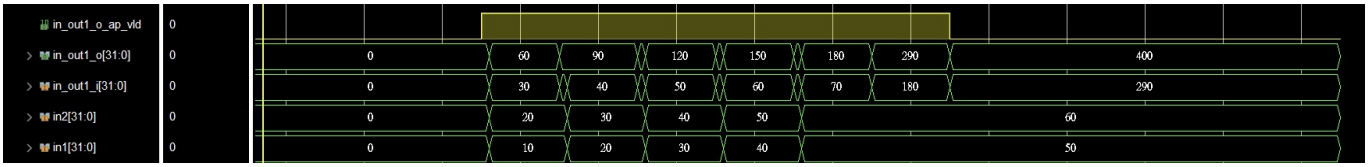
Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_start	in	1	ap_ctrl_hs	adders_io	return value
ap_done	out	1	ap_ctrl_hs	adders_io	return value
ap_idle	out	1	ap_ctrl_hs	adders_io	return value
ap_ready	out	1	ap_ctrl_hs	adders_io	return value
in1	in	32	ap_none	in1	scalar
in2	in	32	ap_none	in2	scalar
in_out1_i	in	32	ap_ovld	in_out1	pointer
in_out1_o	out	32	ap_ovld	in_out1	pointer
in_out1_o_ap_vld	out	1	ap_ovld	in_out1	pointer

Observation:

- We can ensure that the default port-level protocol of in1 and in2 is ap_none, and because in_out is both an input and output, so it is separated to two parts(in_out1_i and in_out1_o) and ap_ovld protocol is assigned to in_out1_i and in_out1_o

Waveform of code without adding any directive



Observartion:

- It can see that in_out has been separate to two signal in_out1_o and in_out1_i, and in_out1_o_ap_vld added to in_out1_o

Code with adding directives

```
############################################################
## This file is generated automatically by Vitis HLS.
## Please DO NOT edit it.
## Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.
############################################################
set_directive_top -name adders_io "adders_io"
set_directive_interface -mode ap_vld "adders_io" in1
set_directive_interface -mode ap_ack "adders_io" in2
set_directive_interface -mode ap_hs "adders_io" in_out1
```

Synthesis of code with adding directives

Interface

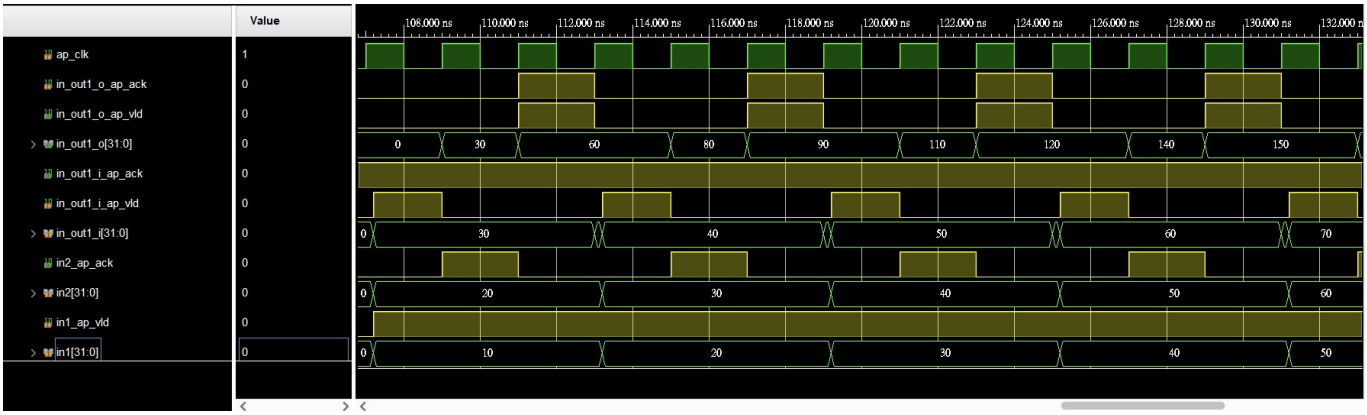
Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	adders_io	return value
ap_rst	in	1	ap_ctrl_hs	adders_io	return value
ap_start	in	1	ap_ctrl_hs	adders_io	return value
ap_done	out	1	ap_ctrl_hs	adders_io	return value
ap_idle	out	1	ap_ctrl_hs	adders_io	return value
ap_ready	out	1	ap_ctrl_hs	adders_io	return value
in1	in	32	ap_vld	in1	scalar
in1_ap_vld	in	1	ap_vld	in1	scalar
in2	in	32	ap_ack	in2	scalar
in2_ap_ack	out	1	ap_ack	in2	scalar
in_out1_i	in	32	ap_hs	in_out1	pointer
in_out1_i_ap_vld	in	1	ap_hs	in_out1	pointer
in_out1_i_ap_ack	out	1	ap_hs	in_out1	pointer
in_out1_o	out	32	ap_hs	in_out1	pointer
in_out1_o_ap_vld	out	1	ap_hs	in_out1	pointer
in_out1_o_ap_ack	in	1	ap_hs	in_out1	pointer

Observation:

- For in1, because we set protocol as ap_vld, so the data on port in1 is only read when port in1_ap_vld is active-High
- For in2, we set protocol as ap_ack, so in2_ap_ack will be active-High when data port in2 is read
- For in_out1, we set protocol as ap_hs, so both ap_vld and ap_ack are added to in_out1_i and in_out1_o

Waveform of code with adding any directives



Lab3 Review how array ports are implemented and can be partitioned

Origin code

```
*******************************************************************************/
#ifndef ARRAY_IO_H_
#define ARRAY_IO_H_

#include <stdio.h>

typedef short din_t;
typedef short dout_t;
typedef int dacc_t;

#define CHANNELS 8
#define SAMPLES  4
#define N CHANNELS * SAMPLES

void array_io (dout_t d_o[N], din_t d_i[N]);

#endif
```

```
#include "array_io.h"

// The data comes in organized in a single array.
// - The first sample for the first channel (CHAN)
// - Then the first sample for the 2nd channel etc.
// The channels are accumulated independently
// E.g. For 8 channels:
// Array Order :  0  1  2  3  4  5  6  7  8     9     10    etc. 16       etc...
// Sample Order:  A0 B0 C0 D0 E0 F0 G0 H0 A1    B1    C2    etc. A2       etc...
// Output Order:  A0 B0 C0 D0 E0 F0 G0 H0 A0+A1 B0+B1 C0+C2 etc. A0+A1+A2 etc...

void array_io (dout_t d_o[N], din_t d_i[N]) {
    int i, rem;

    // Store accumulated data
    static dacc_t acc[CHANNELS];
    dacc_t temp;

    // Accumulate each channel
    For_Loop: for (i=0;i<N;i++) {
        rem=i%CHANNELS;
        temp = acc[rem] + d_i[i];
        acc[rem] = temp;
        d_o[i] = acc[rem];
    }
}
```

Function of code

Do accumulated adding

Solution 1

Synthesis of code

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	array_io	return value
ap_rst	in	1	ap_ctrl_hs	array_io	return value
ap_start	in	1	ap_ctrl_hs	array_io	return value
ap_done	out	1	ap_ctrl_hs	array_io	return value
ap_idle	out	1	ap_ctrl_hs	array_io	return value
ap_ready	out	1	ap_ctrl_hs	array_io	return value
d_o_address0	out	5	ap_memory	d_o	array
d_o_ce0	out	1	ap_memory	d_o	array
d_o_we0	out	1	ap_memory	d_o	array
d_o_d0	out	16	ap_memory	d_o	array
d_i_address0	out	5	ap_memory	d_i	array
d_i_ce0	out	1	ap_memory	d_i	array
d_i_q0	in	16	ap_memory	d_i	array

Observation:

- Function argument d_o has been synthesized to a RAM port(I/O protocol ap_memory), data port(d_o_d0), address port(d_o_address_0), chip enable(d_o_ce0), and write-enable port(do_we0).
- Function argument d_i has been synthesized to a similar RAM interface, but has an input data port (d_i_q0) and no write-enable port because this interface only reads data.
- The bits for d_o_address0 and d_i_address0 is 5, the reason is that we have total 32 elements(channels 8 x samples 4) in the array, so we need 5 bits to represent 32 address.

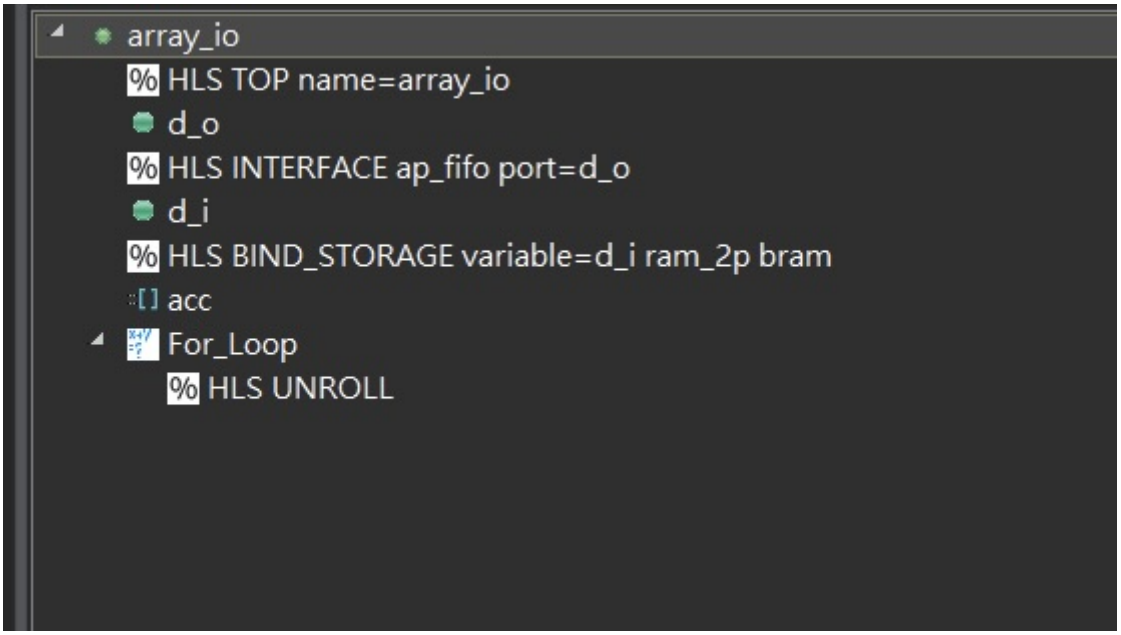
Solution 2

Compare to Solution 1

We can see that in for loop, there is one input read and one output write. So even if multiple inputs and outputs are available, we cannot get benefit from it.

To solve problem mentioned above, we unroll the for loop, let it be a SIMD structure, and we also let d_i be a two port RAM to let 2 inputs send to the function at the same time. Last, we set d_o interface as ap_fifo.

Setting of directives of Solution 2



Synthesis of code

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	array_io	return value
ap_rst	in	1	ap_ctrl_hs	array_io	return value
ap_start	in	1	ap_ctrl_hs	array_io	return value
ap_done	out	1	ap_ctrl_hs	array_io	return value
ap_idle	out	1	ap_ctrl_hs	array_io	return value
ap_ready	out	1	ap_ctrl_hs	array_io	return value
d_o_din	out	16	ap_fifo	d_o	pointer
d_o_full_n	in	1	ap_fifo	d_o	pointer
d_o_write	out	1	ap_fifo	d_o	pointer
d_i_address0	out	5	ap_memory	d_i	array
d_i_ce0	out	1	ap_memory	d_i	array
d_i_q0	in	16	ap_memory	d_i	array
d_i_address1	out	5	ap_memory	d_i	array
d_i_ce1	out	1	ap_memory	d_i	array
d_i_q1	in	16	ap_memory	d_i	array

Observation:

- We can see that d_i has been implemented as a dual-port RAM interface and d_o has been implemented as a FIFO interface with a port called d_o_full_n to determine whether FIFO is full or not

Report comparison

Latency

		solution1	solution2
Latency (cycles)	min	34	33
	max	34	33
Latency (absolute)	min	0.136 us	0.132 us
	max	0.136 us	0.132 us
Interval (cycles)	min	35	34
	max	35	34

Utilization Estimates

	solution1	solution2
BRAM_18K	0	0
DSP	0	0
FF	82	1274
LUT	129	2118
URAM	0	0

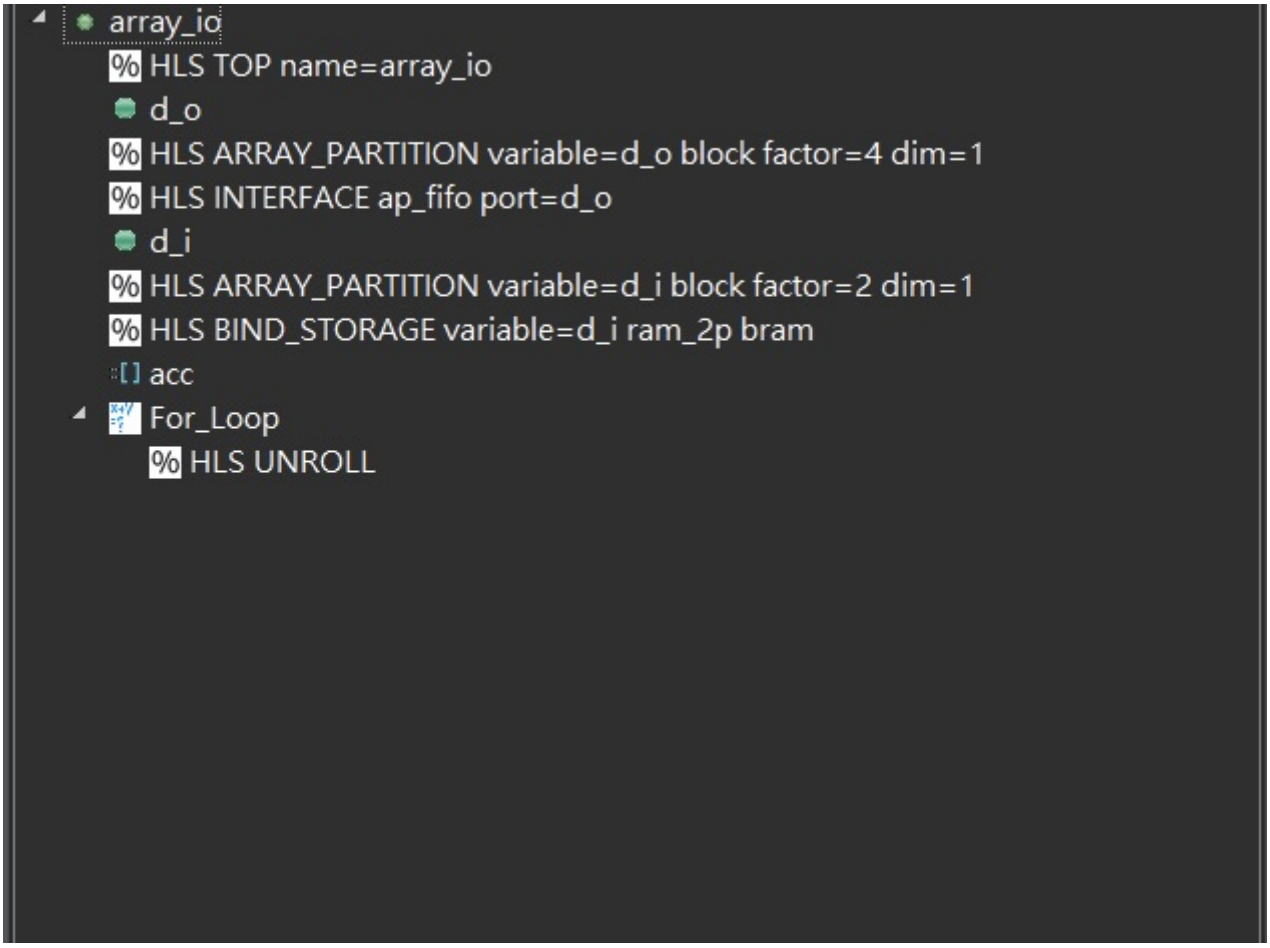
Observation:

- When we compare the result with solution1, we can see that the latency of two solutions almost the same and solution2's hardware utilization is more than solution1 a lot. The reason is that we only use one output port(ap_fifo) in d_o. Even thought our input numbers and computation resources increased. Single output port is the main problem. So we need to let d_o also has multiple ports.

Solution 3

According to solution2's problem, we need to partition d_o array to have more output ports. We partition the array to 4 blocks and also partition d_i to two blocks, each block has dual-port

Setting of directives of Solution 3



Synthesis of code

Interface

- **Summary**

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	array_io	return value
ap_rst	in	1	ap_ctrl_hs	array_io	return value
ap_start	in	1	ap_ctrl_hs	array_io	return value
ap_done	out	1	ap_ctrl_hs	array_io	return value
ap_idle	out	1	ap_ctrl_hs	array_io	return value
ap_ready	out	1	ap_ctrl_hs	array_io	return value
d_o_0_din	out	16	ap_fifo	d_o_0	pointer
d_o_0_full_n	in	1	ap_fifo	d_o_0	pointer
d_o_0_write	out	1	ap_fifo	d_o_0	pointer
d_o_1_din	out	16	ap_fifo	d_o_1	pointer
d_o_1_full_n	in	1	ap_fifo	d_o_1	pointer
d_o_1_write	out	1	ap_fifo	d_o_1	pointer
d_o_2_din	out	16	ap_fifo	d_o_2	pointer
d_o_2_full_n	in	1	ap_fifo	d_o_2	pointer
d_o_2_write	out	1	ap_fifo	d_o_2	pointer
d_o_3_din	out	16	ap_fifo	d_o_3	pointer
d_o_3_full_n	in	1	ap_fifo	d_o_3	pointer
d_o_3_write	out	1	ap_fifo	d_o_3	pointer
d_i_0_address0	out	4	ap_memory	d_i_0	array
d_i_0_ce0	out	1	ap_memory	d_i_0	array
d_i_0_q0	in	16	ap_memory	d_i_0	array
d_i_0_address1	out	4	ap_memory	d_i_0	array
d_i_0_ce1	out	1	ap_memory	d_i_0	array
d_i_0_q1	in	16	ap_memory	d_i_0	array
d_i_1_address0	out	4	ap_memory	d_i_1	array
d_i_1_ce0	out	1	ap_memory	d_i_1	array
d_i_1_q0	in	16	ap_memory	d_i_1	array
d_i_1_address1	out	4	ap_memory	d_i_1	array
d_i_1_ce1	out	1	ap_memory	d_i_1	array
d_i_1_q1	in	16	ap_memory	d_i_1	array

Observation:

- We can see that d_o has been implemented as four separate FIFO interfaces. d_i has been implemented as two separate RAM interfaces, each uses a dual-port interface.

Report comparison

Latency

		solution1	solution2	solution3
Latency (cycles)	min	34	33	10
	max	34	33	10
Latency (absolute)	min	0.136 us	0.132 us	40.000 ns
	max	0.136 us	0.132 us	40.000 ns
Interval (cycles)	min	35	34	11
	max	35	34	11

Utilization Estimates

	solution1	solution2	solution3
BRAM_18K	0	0	0
DSP	0	0	0
FF	82	1274	867
LUT	129	2118	2135
URAM	0	0	0

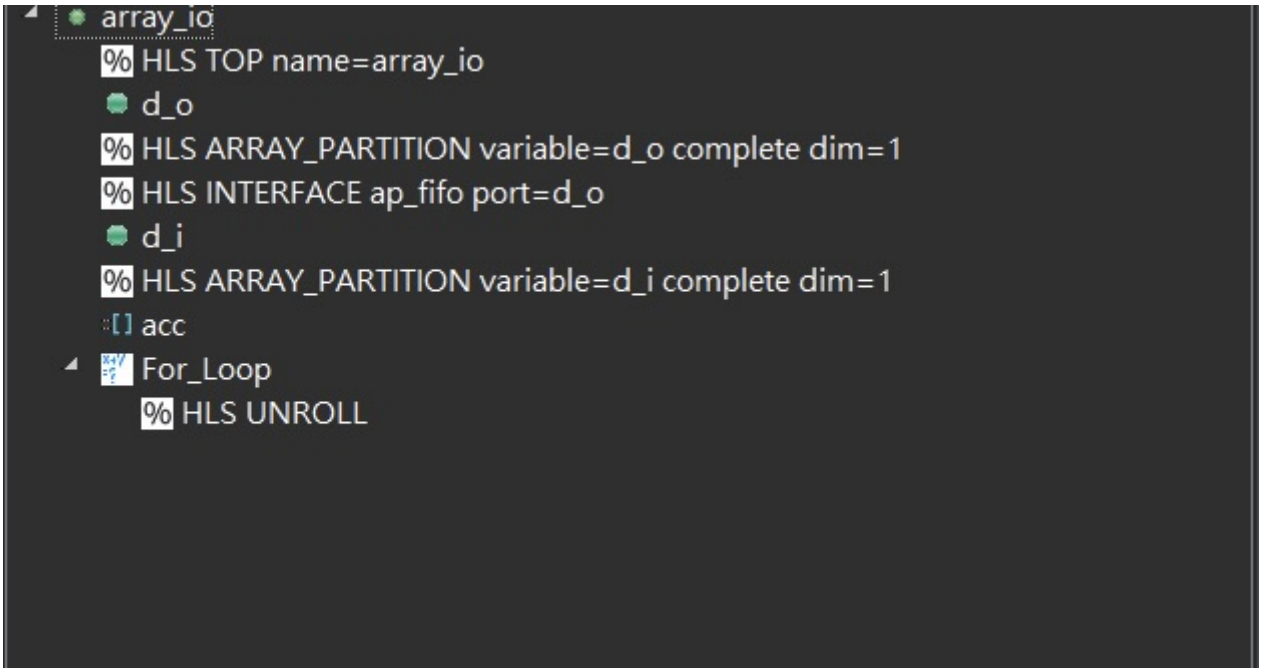
Observation:

- Because we make both input and output be 4 ports, so the latency is decreased.

Solution 4

In this solution, we partitioned input and output array completely,

Setting of directives of Solution 4



Synthesis of code

Interface

- **Summary**

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	array_io	return value
ap_rst	in	1	ap_ctrl_hs	array_io	return value
ap_start	in	1	ap_ctrl_hs	array_io	return value
ap_done	out	1	ap_ctrl_hs	array_io	return value
ap_idle	out	1	ap_ctrl_hs	array_io	return value
ap_ready	out	1	ap_ctrl_hs	array_io	return value
d_o_0	out	16	ap_vld	d_o_0	pointer
d_o_0_ap_vld	out	1	ap_vld	d_o_0	pointer
d_o_1	out	16	ap_vld	d_o_1	pointer
d_o_1_ap_vld	out	1	ap_vld	d_o_1	pointer
d_o_2	out	16	ap_vld	d_o_2	pointer
d_o_2_ap_vld	out	1	ap_vld	d_o_2	pointer
d_o_3	out	16	ap_vld	d_o_3	pointer
d_o_3_ap_vld	out	1	ap_vld	d_o_3	pointer
d_o_4	out	16	ap_vld	d_o_4	pointer
d_o_4_ap_vld	out	1	ap_vld	d_o_4	pointer
d_o_5	out	16	ap_vld	d_o_5	pointer
d_o_5_ap_vld	out	1	ap_vld	d_o_5	pointer
d_o_6	out	16	ap_vld	d_o_6	pointer
d_o_6_ap_vld	out	1	ap_vld	d_o_6	pointer
d_o_7	out	16	ap_vld	d_o_7	pointer
d_o_7_ap_vld	out	1	ap_vld	d_o_7	pointer

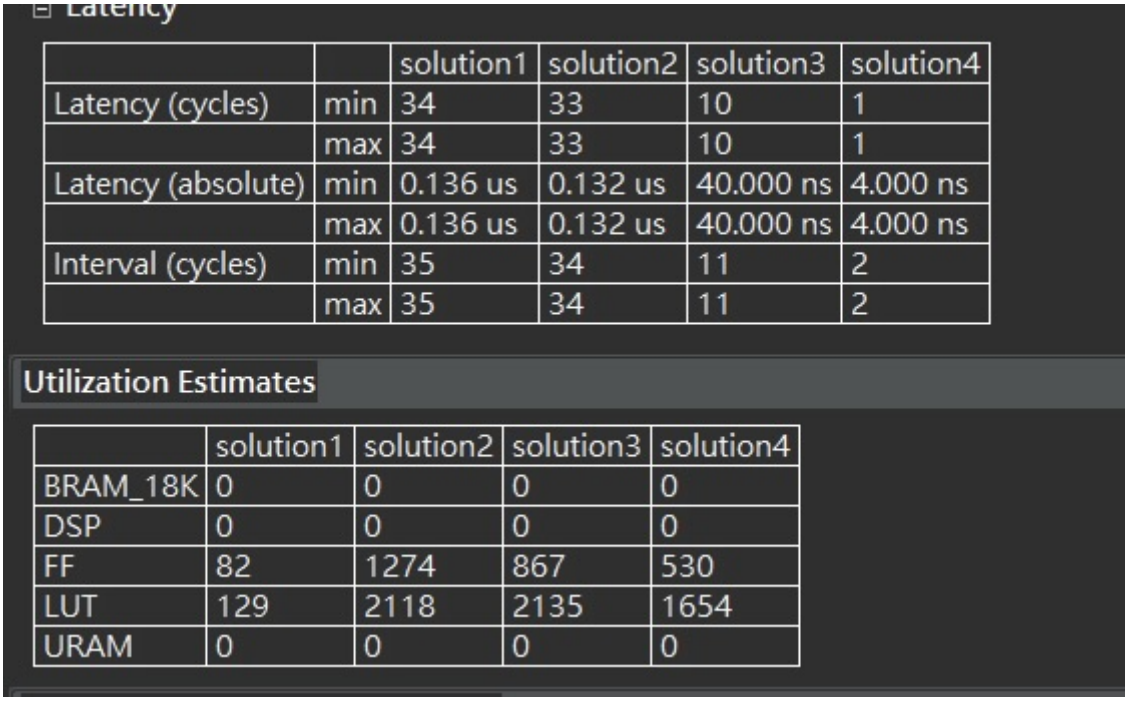
d_o_8	out	16	ap_vld	d_o_8	pointer
d_o_8_ap_vld	out	1	ap_vld	d_o_8	pointer
d_o_9	out	16	ap_vld	d_o_9	pointer
d_o_9_ap_vld	out	1	ap_vld	d_o_9	pointer
d_o_10	out	16	ap_vld	d_o_10	pointer
d_o_10_ap_vld	out	1	ap_vld	d_o_10	pointer
d_o_11	out	16	ap_vld	d_o_11	pointer
d_o_11_ap_vld	out	1	ap_vld	d_o_11	pointer
d_o_12	out	16	ap_vld	d_o_12	pointer
d_o_12_ap_vld	out	1	ap_vld	d_o_12	pointer
d_o_13	out	16	ap_vld	d_o_13	pointer
d_o_13_ap_vld	out	1	ap_vld	d_o_13	pointer
d_o_14	out	16	ap_vld	d_o_14	pointer
d_o_14_ap_vld	out	1	ap_vld	d_o_14	pointer
d_o_15	out	16	ap_vld	d_o_15	pointer
d_o_15_ap_vld	out	1	ap_vld	d_o_15	pointer
d_o_16	out	16	ap_vld	d_o_16	pointer
d_o_16_ap_vld	out	1	ap_vld	d_o_16	pointer
d_o_17	out	16	ap_vld	d_o_17	pointer
d_o_17_ap_vld	out	1	ap_vld	d_o_17	pointer
d_o_18	out	16	ap_vld	d_o_18	pointer
d_o_18_ap_vld	out	1	ap_vld	d_o_18	pointer
d_o_19	out	16	ap_vld	d_o_19	pointer
d_o_19_ap_vld	out	1	ap_vld	d_o_19	pointer
d_o_20	out	16	ap_vld	d_o_20	pointer
d_o_20_ap_vld	out	1	ap_vld	d_o_20	pointer
d_o_21	out	16	ap_vld	d_o_21	pointer
d_o_21_ap_vld	out	1	ap_vld	d_o_21	pointer
d_o_22	out	16	ap_vld	d_o_22	pointer
d_o_22_ap_vld	out	1	ap_vld	d_o_22	pointer
d_o_23	out	16	ap_vld	d_o_23	pointer
d_o_23_ap_vld	out	1	ap_vld	d_o_23	pointer
d_o_24	out	16	ap_vld	d_o_24	pointer
d_o_24_ap_vld	out	1	ap_vld	d_o_24	pointer
d_o_25	out	16	ap_vld	d_o_25	pointer
d_o_25_ap_vld	out	1	ap_vld	d_o_25	pointer
d_o_26	out	16	ap_vld	d_o_26	pointer
d_o_26_ap_vld	out	1	ap_vld	d_o_26	pointer
d_o_27	out	16	ap_vld	d_o_27	pointer
d_o_27_ap_vld	out	1	ap_vld	d_o_27	pointer
d_o_28	out	16	ap_vld	d_o_28	pointer
d_o_28_ap_vld	out	1	ap_vld	d_o_28	pointer
d_o_29	out	16	ap_vld	d_o_29	pointer
d_o_29_ap_vld	out	1	ap_vld	d_o_29	pointer
d_o_30	out	16	ap_vld	d_o_30	pointer
d_o_30_ap_vld	out	1	ap_vld	d_o_30	pointer
d_o_31	out	16	ap_vld	d_o_31	pointer
d_o_31_ap_vld	out	1	ap_vld	d_o_31	pointer
d_i_0	in	16	ap_none	d_i_0	pointer
d_i_1	in	16	ap_none	d_i_1	pointer
d_i_2	in	16	ap_none	d_i_2	pointer
d_i_3	in	16	ap_none	d_i_3	pointer
d_i_4	in	16	ap_none	d_i_4	pointer

d_i_6	in	16	ap_none	d_i_6	pointer
d_i_7	in	16	ap_none	d_i_7	pointer
d_i_8	in	16	ap_none	d_i_8	pointer
d_i_9	in	16	ap_none	d_i_9	pointer
d_i_10	in	16	ap_none	d_i_10	pointer
d_i_11	in	16	ap_none	d_i_11	pointer
d_i_12	in	16	ap_none	d_i_12	pointer
d_i_13	in	16	ap_none	d_i_13	pointer
d_i_14	in	16	ap_none	d_i_14	pointer
d_i_15	in	16	ap_none	d_i_15	pointer
d_i_16	in	16	ap_none	d_i_16	pointer
d_i_17	in	16	ap_none	d_i_17	pointer
d_i_18	in	16	ap_none	d_i_18	pointer
d_i_19	in	16	ap_none	d_i_19	pointer
d_i_20	in	16	ap_none	d_i_20	pointer
d_i_21	in	16	ap_none	d_i_21	pointer
d_i_22	in	16	ap_none	d_i_22	pointer
d_i_23	in	16	ap_none	d_i_23	pointer
d_i_24	in	16	ap_none	d_i_24	pointer
d_i_25	in	16	ap_none	d_i_25	pointer
d_i_26	in	16	ap_none	d_i_26	pointer
d_i_27	in	16	ap_none	d_i_27	pointer
d_i_28	in	16	ap_none	d_i_28	pointer
d_i_29	in	16	ap_none	d_i_29	pointer
d_i_30	in	16	ap_none	d_i_30	pointer
d_i_31	in	16	ap_none	d_i_31	pointer

Observation:

- According to the synthesis report, we can see that d_i and d_o has been implemented to 32 separate ports. The port-level portocol of d_i is ap_none since it's scalar.

Report comparison



Latency

		solution1	solution2	solution3	solution4
Latency (cycles)	min	34	33	10	1
	max	34	33	10	1
Latency (absolute)	min	0.136 us	0.132 us	40.000 ns	4.000 ns
	max	0.136 us	0.132 us	40.000 ns	4.000 ns
Interval (cycles)	min	35	34	11	2
	max	35	34	11	2

Utilization Estimates

	solution1	solution2	solution3	solution4
BRAM_18K	0	0	0	0
DSP	0	0	0	0
FF	82	1274	867	530
LUT	129	2118	2135	1654
URAM	0	0	0	0

Observation:

- Because we can add port to each element of input and output array, so we only need one clock latency to get the outputs.

LAB4: Create an optimized implementation of the design and add AXI4 interfaces

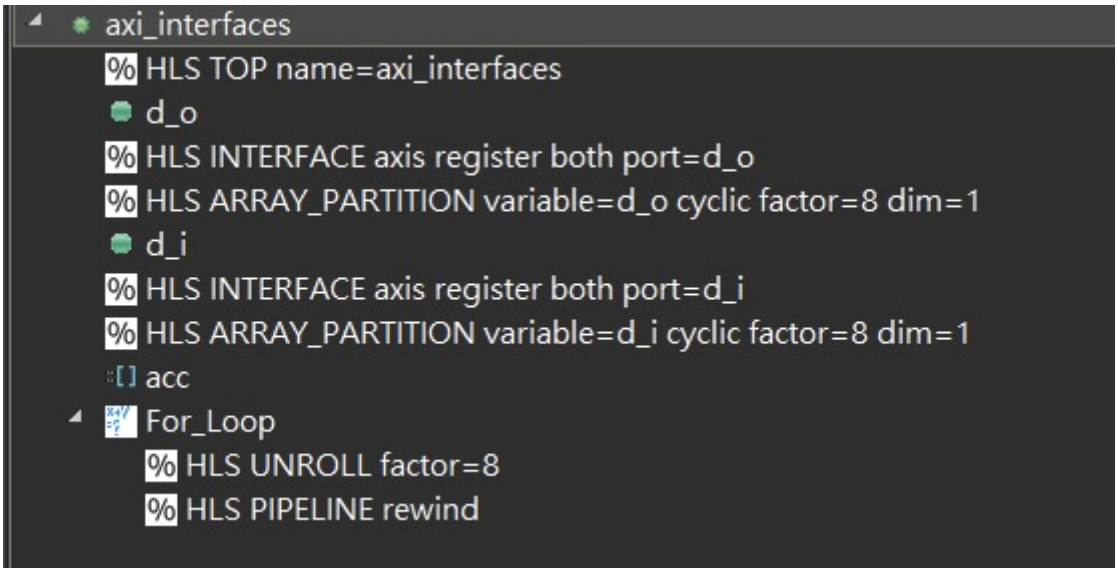
Origin code

```
#include "axi_interfaces.h"

// The data comes in organized in a single array.
// - The first sample for the first channel (CHAN)
// - Then the first sample for the 2nd channel etc.
// The channels are accumulated independently
// E.g. For 8 channels:
// Array Order :  0  1  2  3  4  5  6  7  8     9     10    etc. 16       etc...
// Sample Order:  A0 B0 C0 D0 E0 F0 G0 H0 A1    B1    C2    etc. A2       etc...
// Output Order:  A0 B0 C0 D0 E0 F0 G0 H0 A0+A1 B0+B1 C0+C2 etc. A0+A1+A2 etc...

void axi_interfaces (dout_t d_o[N], din_t d_i[N]) {
    int i, rem;

    // Store accumulated data
    static dacc_t acc[CHANNELS];

    // Accumulate each channel
    For_Loop: for (i=0;i<N;i++) {
        rem=i%CHANNELS;
        acc[rem] = acc[rem] + d_i[i];
        d_o[i] = acc[rem];
    }
}
```

Function of code

Do accumulated adding

Setting of directive



In this Lab, we let input and output array be implemented as AXI4-Stream interfaces. We use cyclic parition with a factor of 8 to create 8 separate partitions. And we unroll the for loop with factor of 8 to let HLS generate 8 copies of code segment, and use pipeline and enable loop rewinding to let initial interval be 1.

Synthesis of Code

Interface

- Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	axi_interfaces	return value
ap_rst_n	in	1	ap_ctrl_hs	axi_interfaces	return value
ap_start	in	1	ap_ctrl_hs	axi_interfaces	return value
ap_done	out	1	ap_ctrl_hs	axi_interfaces	return value
ap_idle	out	1	ap_ctrl_hs	axi_interfaces	return value
ap_ready	out	1	ap_ctrl_hs	axi_interfaces	return value
d_i_0_TVALID	in	1	axis	d_i_0	pointer
d_i_0_TDATA	in	16	axis	d_i_0	pointer
d_i_0_TREADY	out	1	axis	d_i_0	pointer
d_o_0_TREADY	in	1	axis	d_o_0	pointer
d_o_0_TDATA	out	16	axis	d_o_0	pointer
d_o_0_TVALID	out	1	axis	d_o_0	pointer
d_i_1_TVALID	in	1	axis	d_i_1	pointer
d_i_1_TDATA	in	16	axis	d_i_1	pointer
d_i_1_TREADY	out	1	axis	d_i_1	pointer
d_o_1_TREADY	in	1	axis	d_o_1	pointer
d_o_1_TDATA	out	16	axis	d_o_1	pointer
d_o_1_TVALID	out	1	axis	d_o_1	pointer
d_i_2_TVALID	in	1	axis	d_i_2	pointer
d_i_2_TDATA	in	16	axis	d_i_2	pointer
d_i_2_TREADY	out	1	axis	d_i_2	pointer
d_o_2_TREADY	in	1	axis	d_o_2	pointer
d_o_2_TDATA	out	16	axis	d_o_2	pointer
d_o_2_TVALID	out	1	axis	d_o_2	pointer
d_i_3_TVALID	in	1	axis	d_i_3	pointer
d_i_3_TDATA	in	16	axis	d_i_3	pointer
d_i_3_TREADY	out	1	axis	d_i_3	pointer
d_o_3_TREADY	in	1	axis	d_o_3	pointer
d_o_3_TDATA	out	16	axis	d_o_3	pointer
d_o_3_TVALID	out	1	axis	d_o_3	pointer
d_i_4_TVALID	in	1	axis	d_i_4	pointer
d_i_4_TDATA	in	16	axis	d_i_4	pointer
d_i_4_TREADY	out	1	axis	d_i_4	pointer
d_o_4_TREADY	in	1	axis	d_o_4	pointer
d_o_4_TDATA	out	16	axis	d_o_4	pointer
d_o_4_TVALID	out	1	axis	d_o_4	pointer
d_i_5_TVALID	in	1	axis	d_i_5	pointer
d_i_5_TDATA	in	16	axis	d_i_5	pointer
d_i_5_TREADY	out	1	axis	d_i_5	pointer
d_o_5_TREADY	in	1	axis	d_o_5	pointer
d_o_5_TDATA	out	16	axis	d_o_5	pointer
d_o_5_TVALID	out	1	axis	d_o_5	pointer
d_i_6_TVALID	in	1	axis	d_i_6	pointer
d_i_6_TDATA	in	16	axis	d_i_6	pointer
d_i_6_TREADY	out	1	axis	d_i_6	pointer
d_o_6_TREADY	in	1	axis	d_o_6	pointer
d_o_6_TDATA	out	16	axis	d_o_6	pointer
d_o_6_TVALID	out	1	axis	d_o_6	pointer
d_i_7_TVALID	in	1	axis	d_i_7	pointer
d_i_7_TDATA	in	16	axis	d_i_7	pointer
d_i_7_TREADY	out	1	axis	d_i_7	pointer
d_o_7_TREADY	in	1	axis	d_o_7	pointer
d_o_7_TDATA	out	16	axis	d_o_7	pointer
d_o_7_TVALID	out	1	axis	d_o_7	pointer

Observation:

- We can see that 16 axis channels is built, each channel has 3 ports(d_x_x_TVALID,d_x_x_TDATA,d_x_x_TREADY).

Result compare to Lab3

Lab4

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
4	5	16.000 ns	20.000 ns	4	4	loop rewind stp(delay=0 clock cycles(s))

Detail

Instance

Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- For_Loop	4	4	2	1	1	4	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	523	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	316	-
Register	-	-	527	-	-
Total	0	0	527	839	0
Available	4320	6840	2364480	1182240	960
Available SLR	1440	2280	788160	394080	320
Utilization (%)	0	0	~0	~0	0
Utilization SLR (%)	0	0	~0	~0	0

Lab3

Latency

		solution1	solution2	solution3	solution4
Latency (cycles)	min	34	33	10	1
	max	34	33	10	1
Latency (absolute)	min	0.136 us	0.132 us	40.000 ns	4.000 ns
	max	0.136 us	0.132 us	40.000 ns	4.000 ns
Interval (cycles)	min	35	34	11	2
	max	35	34	11	2

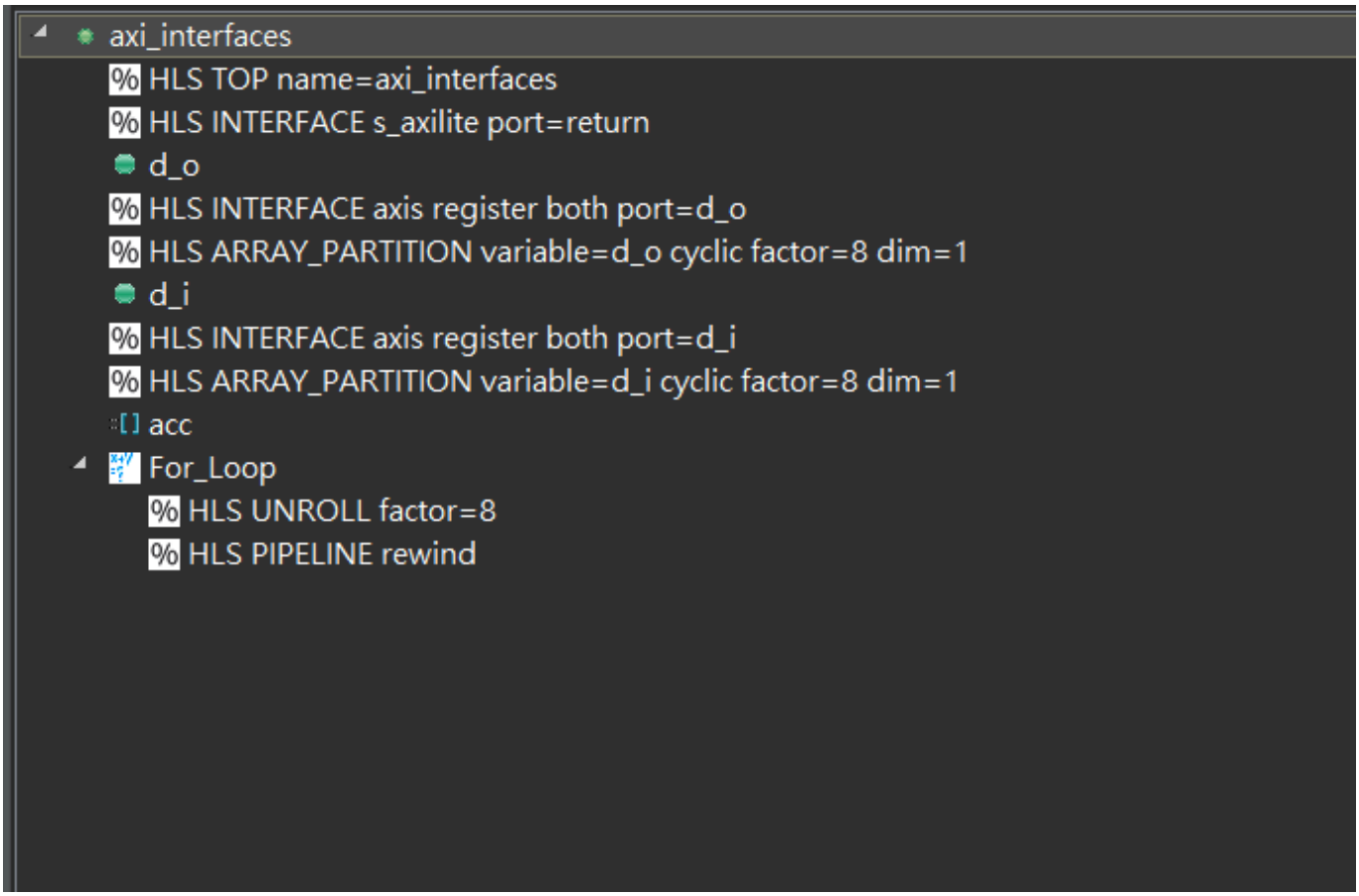
Utilization Estimates

	solution1	solution2	solution3	solution4
BRAM_18K	0	0	0	0
DSP	0	0	0	0
FF	82	1274	867	530
LUT	129	2118	2135	1654
URAM	0	0	0	0

Observation:

- Compare the result with Lab3, although we have more latency than solution4, but because we do pipeline and rewind, so if there are sequential inputs, we can generate the result every cycle. The hardware utilization of Lab4 is less than solution4 because we use total 16 channels and solution4 in lab3 generate 64 channels.

Implement AXI4-LITE Interfaces



We add interface with mode s_axilite to specify AXI4-Lite interface.

Synthesis of code

Interface

- **Summary**

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
s_axi_control_AWVALID	in	1	s_axi	control	return void
s_axi_control_AWREADY	out	1	s_axi	control	return void
s_axi_control_AWADDR	in	4	s_axi	control	return void
s_axi_control_WVALID	in	1	s_axi	control	return void
s_axi_control_WREADY	out	1	s_axi	control	return void
s_axi_control_WDATA	in	32	s_axi	control	return void
s_axi_control_WSTRB	in	4	s_axi	control	return void
s_axi_control_ARVALID	in	1	s_axi	control	return void
s_axi_control_ARREADY	out	1	s_axi	control	return void
s_axi_control_ARADDR	in	4	s_axi	control	return void
s_axi_control_RVALID	out	1	s_axi	control	return void
s_axi_control_RREADY	in	1	s_axi	control	return void
s_axi_control_RDATA	out	32	s_axi	control	return void
s_axi_control_RRESP	out	2	s_axi	control	return void
s_axi_control_BVALID	out	1	s_axi	control	return void
s_axi_control_BREADY	in	1	s_axi	control	return void

s_axi_control_BRESP	out	2	s_axi	control	return void
ap_clk	in	1	ap_ctrl_hs	axi_interfaces	return value
ap_rst_n	in	1	ap_ctrl_hs	axi_interfaces	return value
interrupt	out	1	ap_ctrl_hs	axi_interfaces	return value
d_i_0_TVALID	in	1	axis	d_i_0	pointer
d_i_0_TDATA	in	16	axis	d_i_0	pointer
d_i_0_TREADY	out	1	axis	d_i_0	pointer
d_o_0_TREADY	in	1	axis	d_o_0	pointer
d_o_0_TDATA	out	16	axis	d_o_0	pointer
d_o_0_TVALID	out	1	axis	d_o_0	pointer
d_i_1_TVALID	in	1	axis	d_i_1	pointer
d_i_1_TDATA	in	16	axis	d_i_1	pointer
d_i_1_TREADY	out	1	axis	d_i_1	pointer
d_o_1_TREADY	in	1	axis	d_o_1	pointer
d_o_1_TDATA	out	16	axis	d_o_1	pointer
d_o_1_TVALID	out	1	axis	d_o_1	pointer
d_i_2_TVALID	in	1	axis	d_i_2	pointer
d_i_2_TDATA	in	16	axis	d_i_2	pointer
d_i_2_TREADY	out	1	axis	d_i_2	pointer
d_o_2_TREADY	in	1	axis	d_o_2	pointer
d_o_2_TDATA	out	16	axis	d_o_2	pointer
d_o_2_TVALID	out	1	axis	d_o_2	pointer
d_i_3_TVALID	in	1	axis	d_i_3	pointer
d_i_3_TDATA	in	16	axis	d_i_3	pointer
d_i_3_TREADY	out	1	axis	d_i_3	pointer
d_o_3_TREADY	in	1	axis	d_o_3	pointer
d_o_3_TDATA	out	16	axis	d_o_3	pointer
d_o_3_TVALID	out	1	axis	d_o_3	pointer
d_i_4_TVALID	in	1	axis	d_i_4	pointer
d_i_4_TDATA	in	16	axis	d_i_4	pointer
d_i_4_TREADY	out	1	axis	d_i_4	pointer
d_o_4_TREADY	in	1	axis	d_o_4	pointer
d_o_4_TDATA	out	16	axis	d_o_4	pointer
d_o_4_TVALID	out	1	axis	d_o_4	pointer
d_i_5_TVALID	in	1	axis	d_i_5	pointer
d_i_5_TDATA	in	16	axis	d_i_5	pointer
d_i_5_TREADY	out	1	axis	d_i_5	pointer
d_o_5_TREADY	in	1	axis	d_o_5	pointer
d_o_5_TDATA	out	16	axis	d_o_5	pointer
d_o_5_TVALID	out	1	axis	d_o_5	pointer
d_i_6_TVALID	in	1	axis	d_i_6	pointer
d_i_6_TDATA	in	16	axis	d_i_6	pointer
d_i_6_TREADY	out	1	axis	d_i_6	pointer
d_o_6_TREADY	in	1	axis	d_o_6	pointer
d_o_6_TDATA	out	16	axis	d_o_6	pointer
d_o_6_TVALID	out	1	axis	d_o_6	pointer
d_i_7_TVALID	in	1	axis	d_i_7	pointer
d_i_7_TDATA	in	16	axis	d_i_7	pointer

d_i_7_TREADY	out	1	axis	d_i_7	pointer
d_o_7_TREADY	in	1	axis	d_o_7	pointer
d_o_7_TDATA	out	16	axis	d_o_7	pointer
d_o_7_TVALID	out	1	axis	d_o_7	pointer

Observation:

- We can see that block-level I/O protocol ports(ap_start,ap_done,etc.) have been replaced with AXI4LITE interface and output interrupt signal has been added to the design.

xaxi_interfaces_hw.h

After Export RTL step, xaxi_interfaces_hw.h will be generated in solution2/imp1/ip/drivers/axi_interfaces_v1_0/src/ folder. This file contains the address to access and control the block-level interface signals. For example, setting control register 0x0 bit 0 to the value 1 will enable the ap_start port, or alternatively, setting bit 7 will enable the auto-restart and the design will re-start automatically at the end of each transaction.

```
// ==============================================================
// Vitis HLS - High-Level Synthesis from C, C++ and OpenCL v2022.1 (64-bit)
// Tool Version Limit: 2022.04
// Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.
// ==============================================================
// control
// 0x0 : Control signals
//       bit 0  - ap_start (Read/Write/COH)
//       bit 1  - ap_done (Read/COR)
//       bit 2  - ap_idle (Read)
//       bit 3  - ap_ready (Read/COR)
//       bit 7  - auto_restart (Read/Write)
//       bit 9  - interrupt (Read)
//       others - reserved
// 0x4 : Global Interrupt Enable Register
//       bit 0  - Global Interrupt Enable (Read/Write)
//       others - reserved
// 0x8 : IP Interrupt Enable Register (Read/Write)
//       bit 0 - enable ap_done interrupt (Read/Write)
//       bit 1 - enable ap_ready interrupt (Read/Write)
//       others - reserved
// 0xc : IP Interrupt Status Register (Read/COR)
//       bit 0 - ap_done (Read/COR)
//       bit 1 - ap_ready (Read/COR)
//       others - reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

#define XAXI_INTERFACES_CONTROL_ADDR_AP_CTRL 0x0
#define XAXI_INTERFACES_CONTROL_ADDR_GIE     0x4
#define XAXI_INTERFACES_CONTROL_ADDR_IER     0x8
#define XAXI_INTERFACES_CONTROL_ADDR_ISR     0xc
```