High-Level FPGA Design

Lab 2: HLS Introduction – Gain IP Synthesis & Directives

In this lab you will:

- write synthesizable C/C++ suitable for Vitis HLS
- simulate C/RTL, run synthesis, interpret results
- apply synthesis directives PIPELINE and BIND_OP

Tools and Sources

- You should have Vitis HLS (2025.1 recommended)
- Maps:

```
proj/
├── src/
│   ├── 02_gain/          ← lab2 C++ sources
│   │   └── gain.cpp
│   ├── 03_gain/          ← lab3 C++ sources
│   │   ├── gain.cpp
│   │   ├── tb_gain.cpp
│   │   └── gain.h
│
├── vitis/                ← Vitis workspace (location)
│   ├── gain_hls/         ← lab2 HLS component
│   ├── gain_ip/          ← lab3 HLS component
```

Note: keep your folder path short and simple, avoid spaces, non-ASCII or special characters!

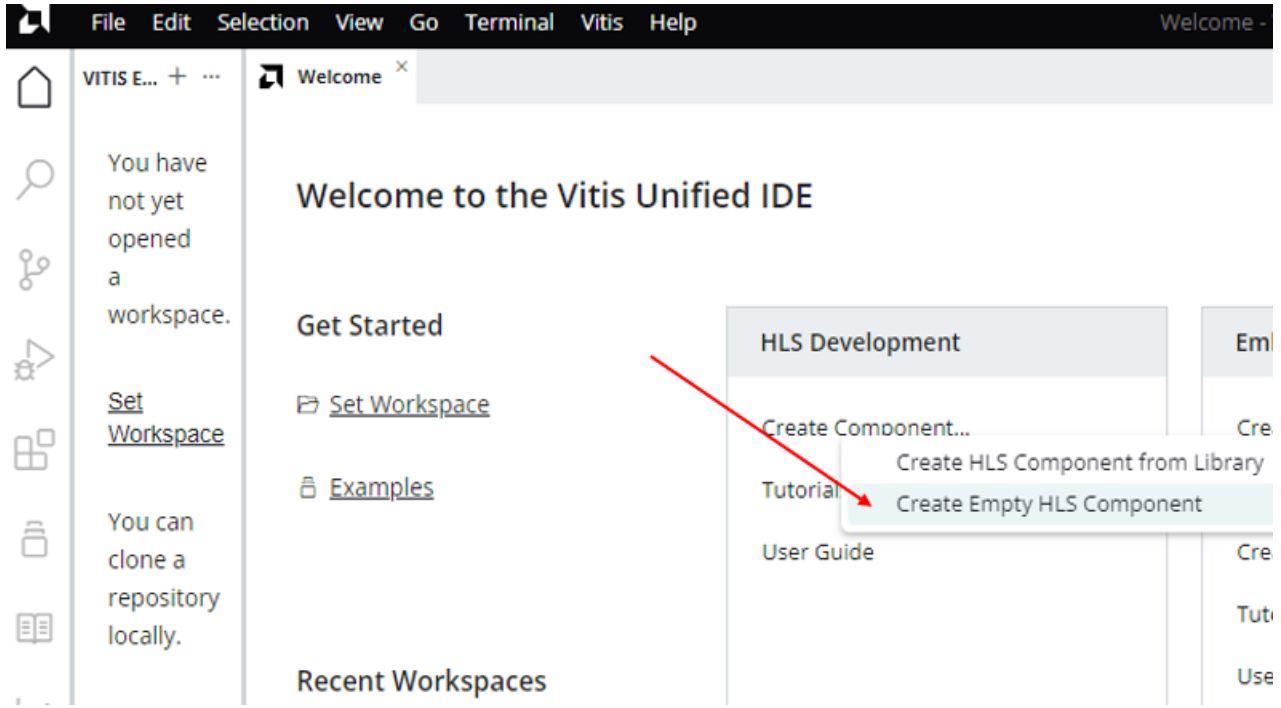
2-1 Create a New Vitis HLS Project

- Launch Vitis 2025.1

- HLS Development, Create Component…, Create Empty HLS Component

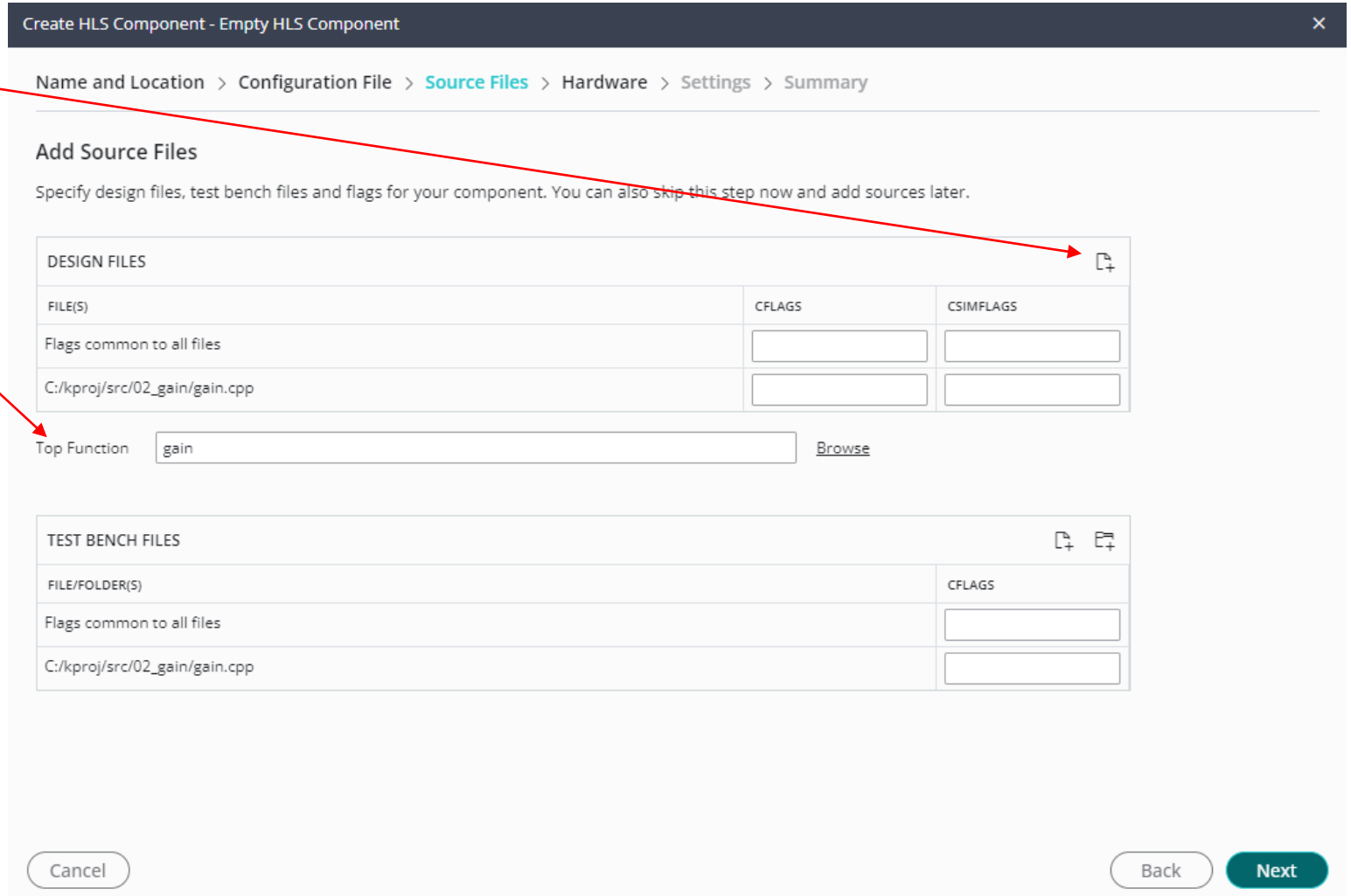
- Component name: **gain_hls**
  Component location: **C:\proj\vitis**

- Configuration File: Empty File

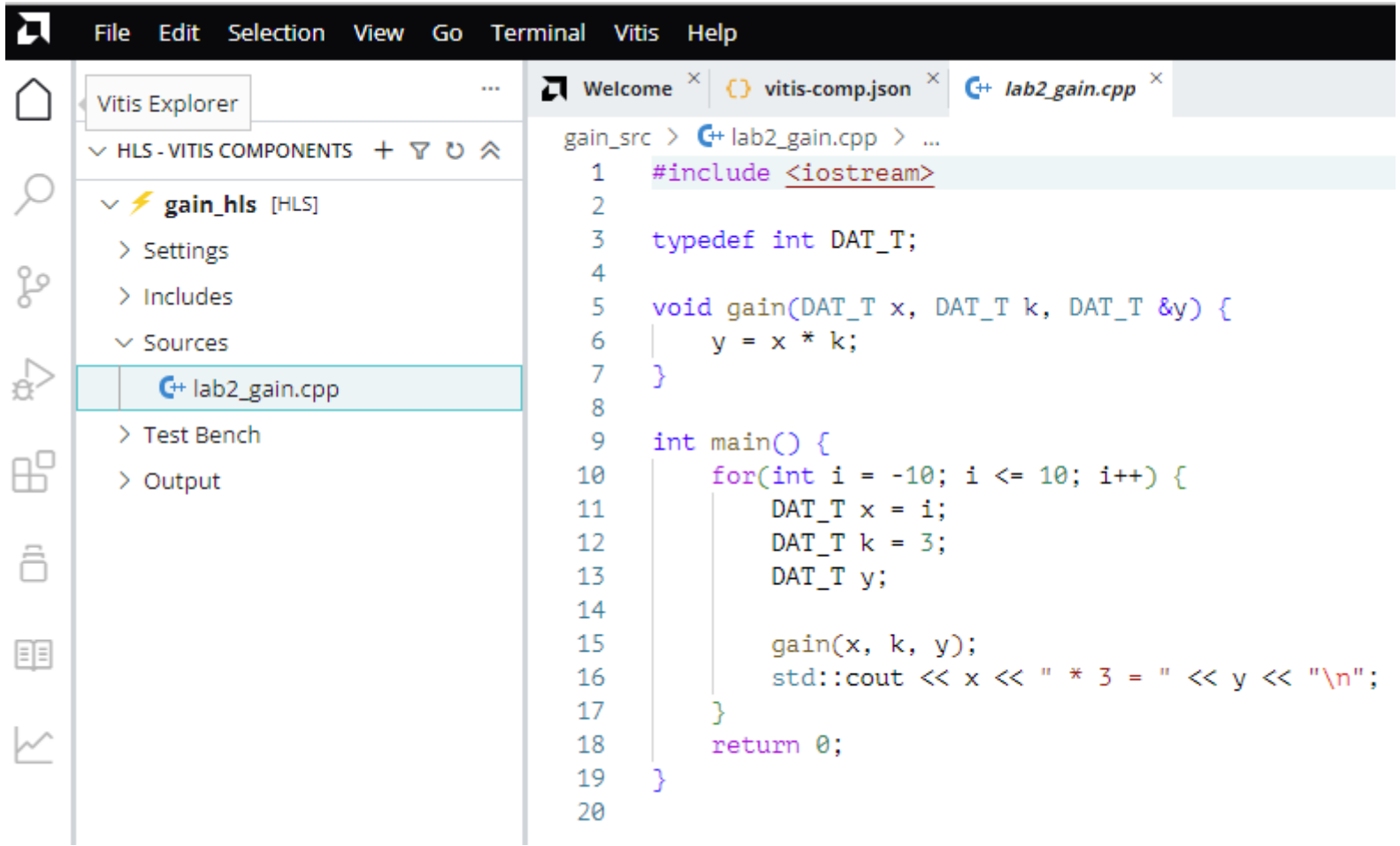


2-1 Create HLS Component > Source Files

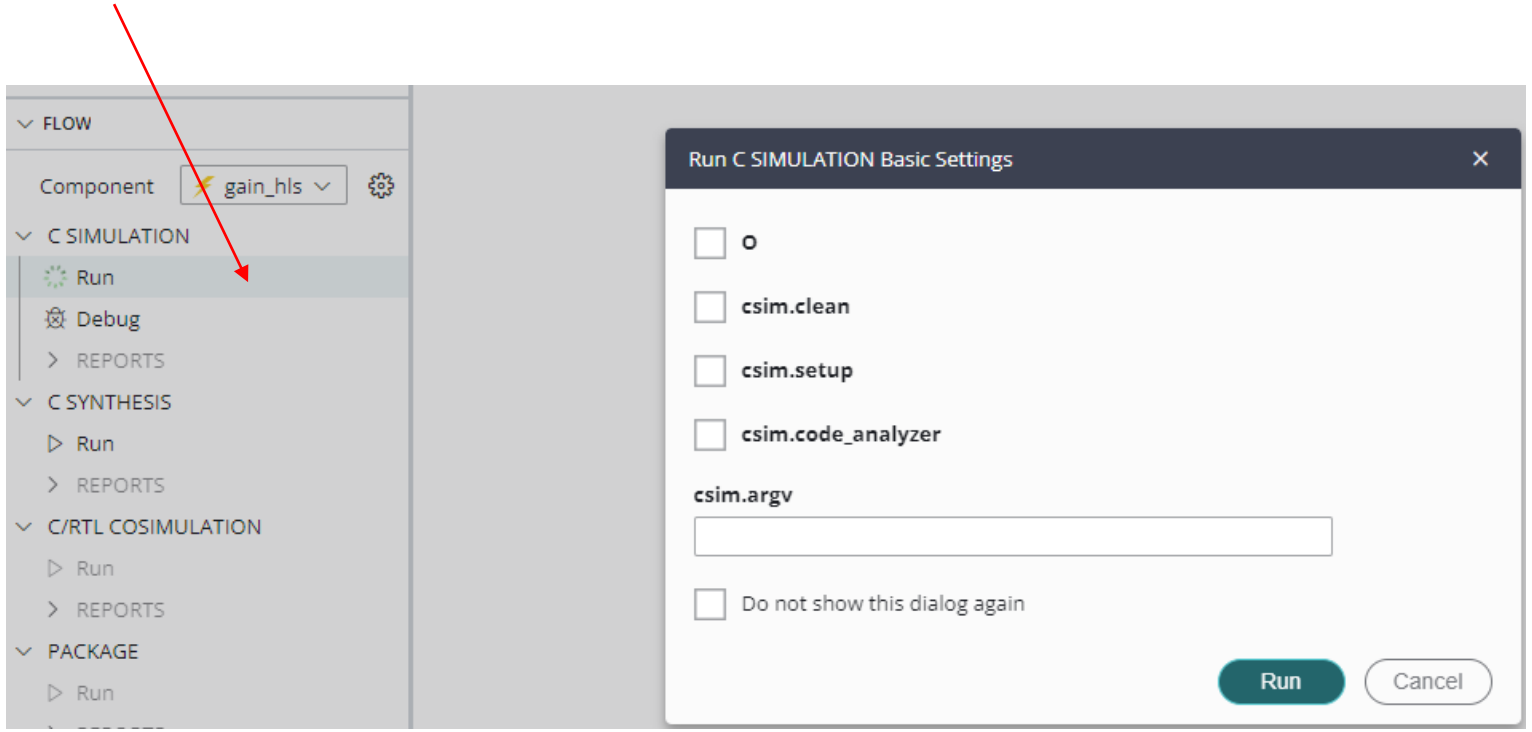
- Add Source Files
  gain.cpp
- Top function: **gain**
- Add Test Bench Files
  gain.cpp
- Click Next
- Select Part xc7z010clg400-1, Next
- Next and Finish, if required Update Workspace



Vitis Explorer



2-2 Run C Simulation



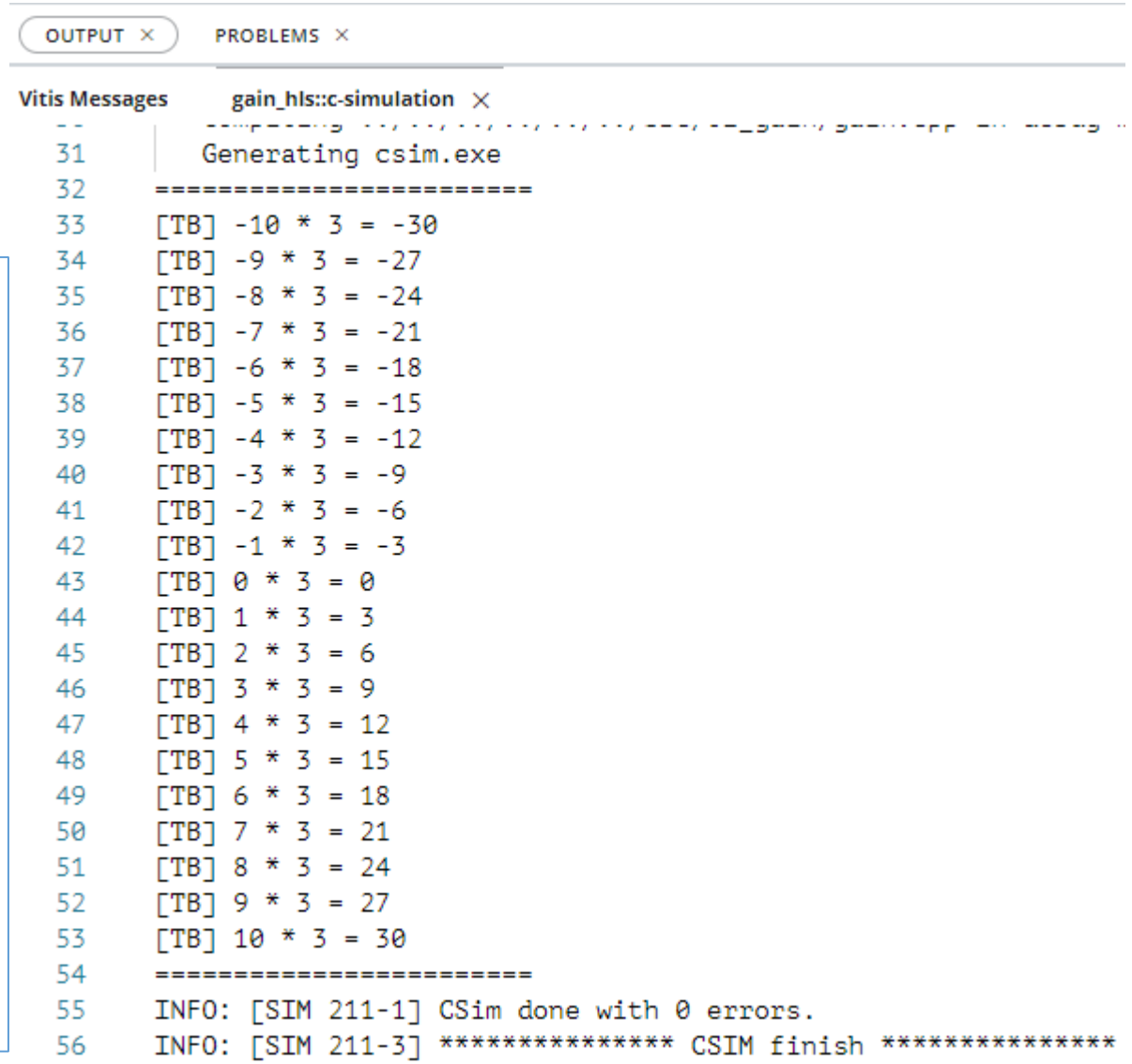
Note: C simulation is fast, purely software based and validates algorithm correctness

2-2 Run C Simulation

Verify output, you should confirm:

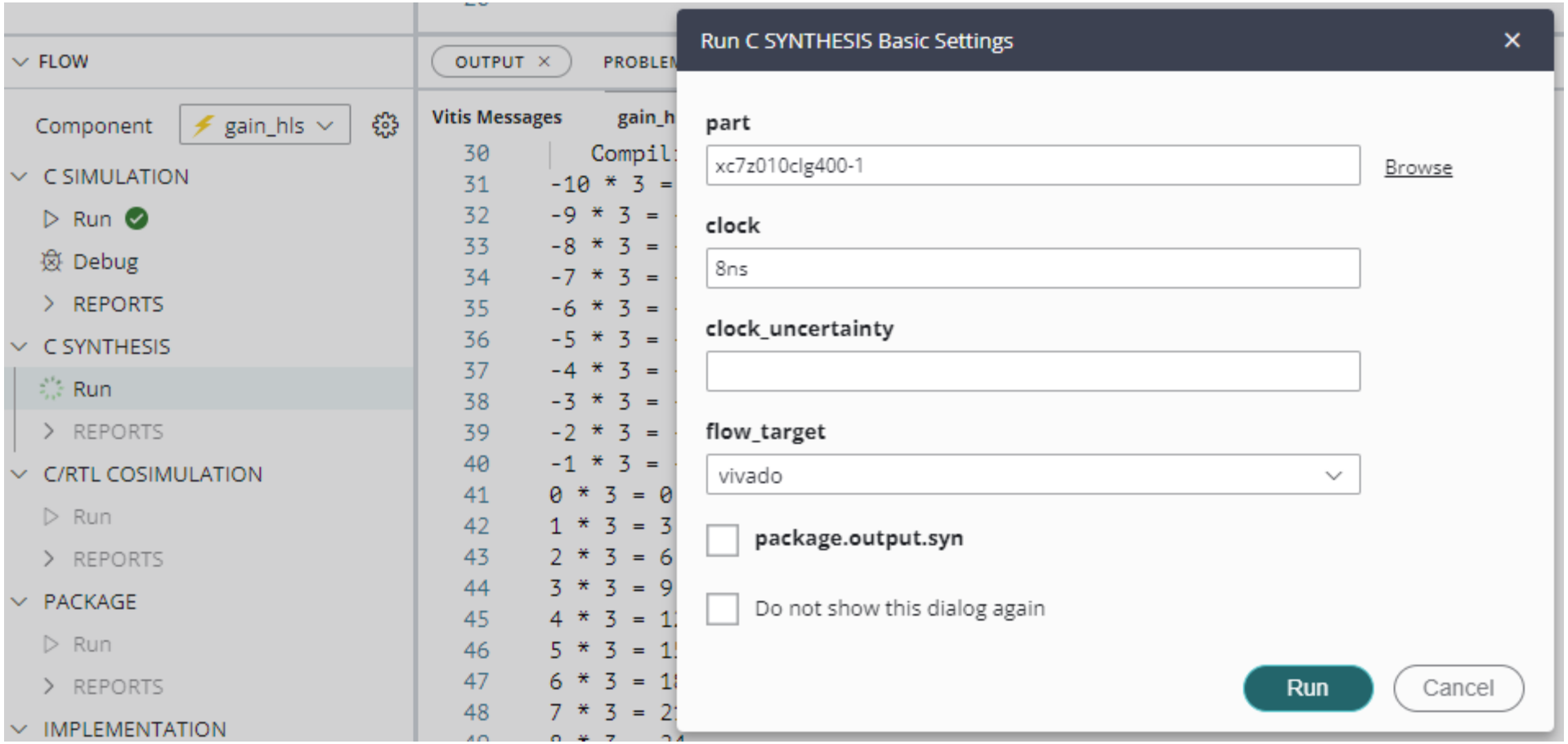
- Correct multiplication
- No overflow for small values
- Output matches expectations

```cpp
#include <iostream>

typedef int DAT_T;

void gain(DAT_T x, DAT_T k, DAT_T &y) {
    y = x * k;
}

int main() {
    std::cout << "\n=======================\n";
    for(int i = -10; i <= 10; i++) {
        DAT_T x = i;
        DAT_T k = 3;
        DAT_T y;


        gain(x, k, y);
        std::cout << "[TB] " << x << " * " << k << " = " << y << "\n";
    }
    std::cout << "\n=======================\n";
    return 0;
}
```

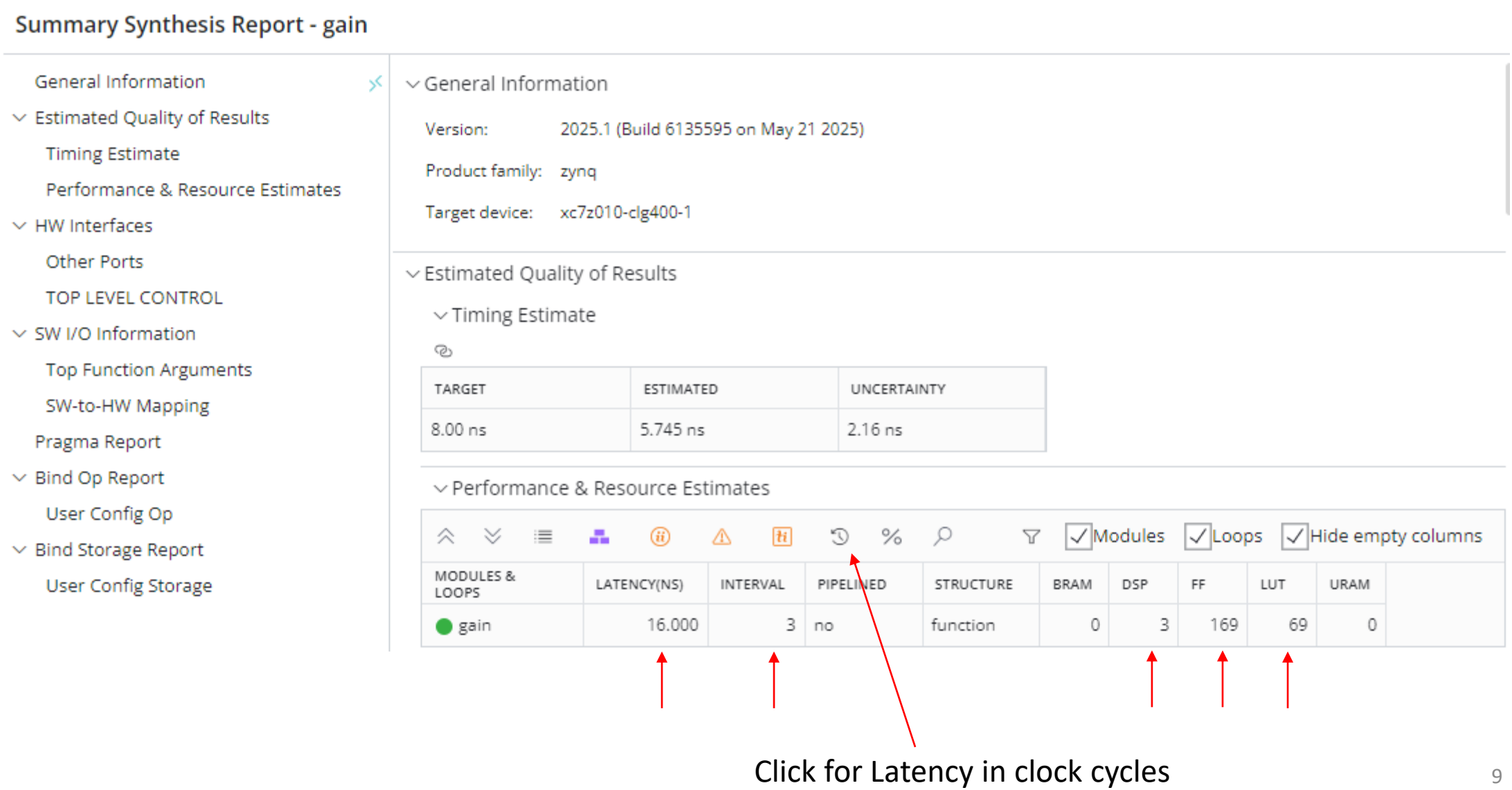


2-3 Synthesize the design

- **Run C Synthesis**
- set clock 8ns



2-3 Open Synthesis Report



2-3 Open Synthesis Report > HW Interfaces

HW Interfaces

Other Ports

PORT	MODE	DIRECTION	BITWIDTH
k	ap_none	in	32
x	ap_none	in	32
y	ap_vld	out	32

TOP LEVEL CONTROL

INTERFACE	TYPE	PORTS
ap_clk	clock	ap_clk
ap_rst	reset	ap_rst
ap_ctrl	ap_ctrl_hs	ap_done ap_idle ap_ready ap_start

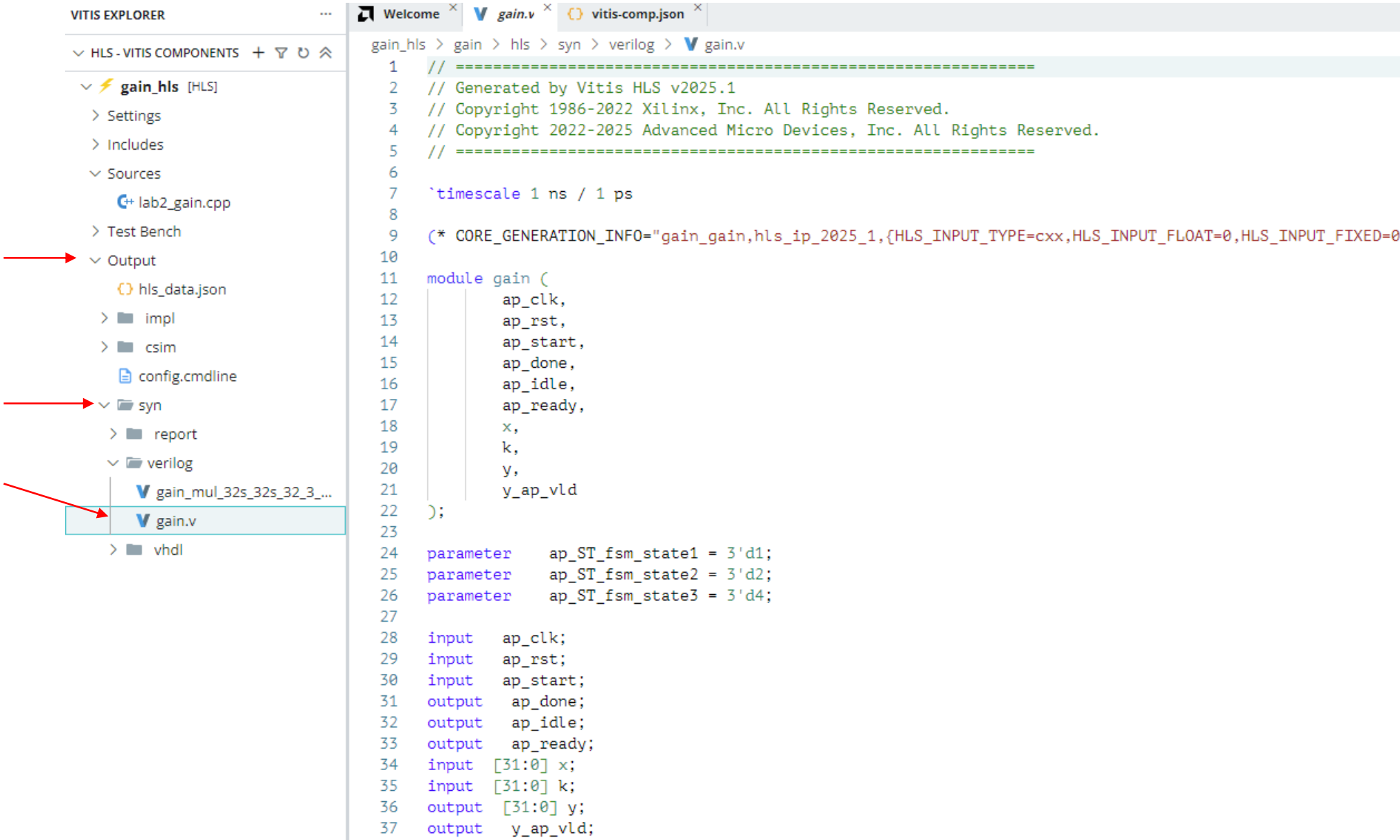
SW I/O Information

Pragma Report

Bind Op Report

NAME	DSP	PRAGMA	VARIABLE	OP	IMPL	LATENCY
gain (1)	3					
mul_32s_32s_32_3_1_U1	3	no	mul_ln6	mul	auto	2

2-3 Open Synthesis RTL Output



2-3 Explore Synthesis - Change Data Type

gain_src > lab2_gain.cpp > ...

```
1   #include <iostream>
2
3   typedef int16_t DAT_T;
4
5   void gain(DAT_T x, DAT_T k, DAT_T &y) {
6       y = x * k;
7   }
```

int16_t
int8_t
float

- Run C Synthesis
- Check Performance & Resource Estimates
  - e.g. 16-bit multipy fits one DSP

TARGET	ESTIMATED	UNCERTAINTY
8.00 ns	5.580 ns	2.16 ns

Performance & Resource Estimates

Modules Loops

MODULES & LOOPS	LATENCY(NS)	INTERVAL	PIPELINED	STRUCTURE	BRAM	DSP	FF	LUT	URAM
gain	0.0	1	no	function	0	1	0	6	0

HW Interfaces

Other Ports

PORT	MODE	DIRECTION	BITWIDTH
k	ap_none	in	16
x	ap_none	in	16
y	ap_vld	out	16

2-3 Experiment, change operator

- integer divider
  - high Latency
- integer adder
  - small

TARGET	ESTIMATED	UNCERTAINTY
8.00 ns	4.135 ns	2.16 ns

Performance & Resource Estimates

Modules Loops

MODULES & LOOPS	LATENCY(CYCLES)	INTERVAL	PIPELINED	STRUCTURE	BRAM	DSP	FF	LUT	URAM
gain	35	36	no	function	0	0	430	402	0

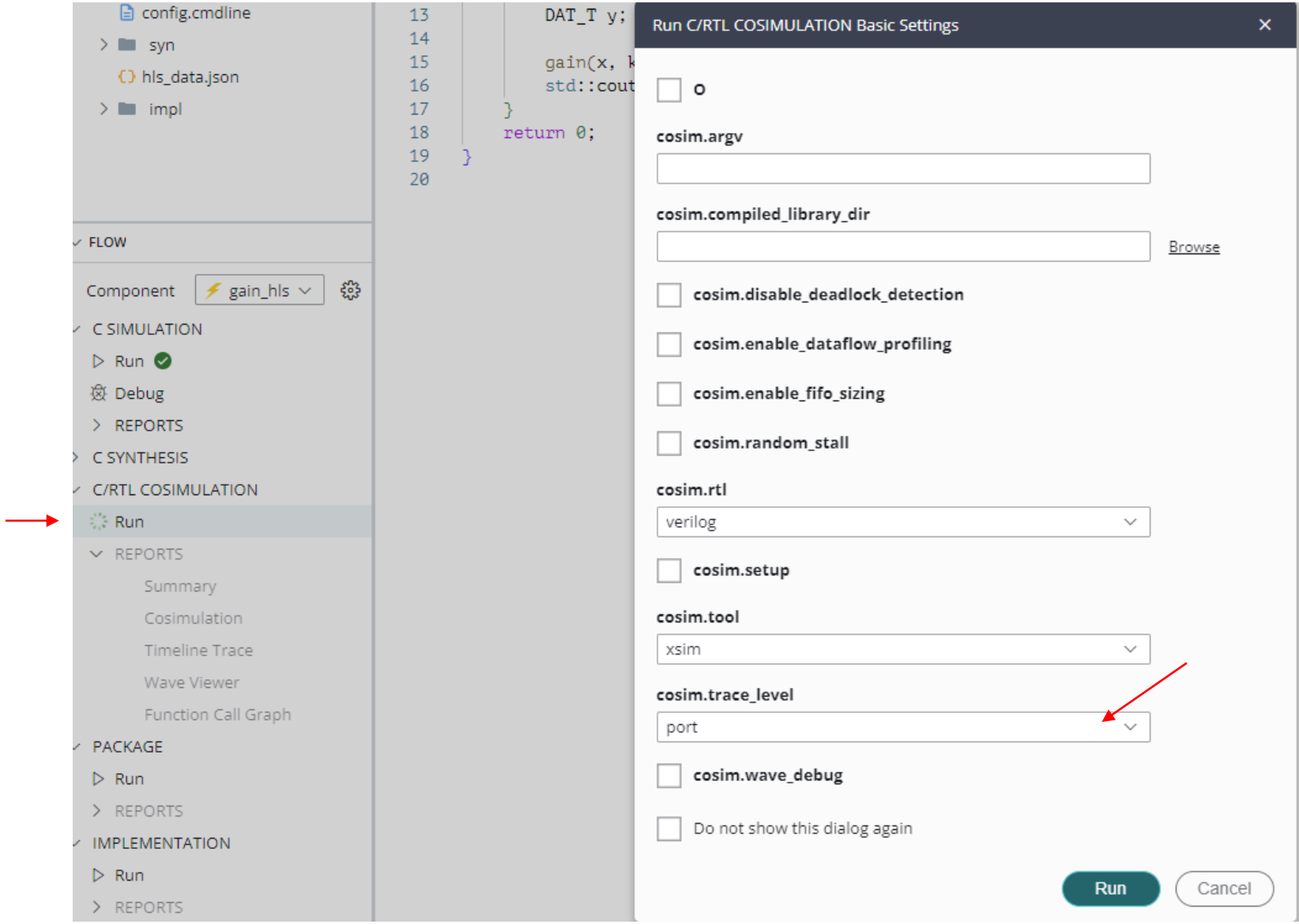
HW Interfaces

Other Ports

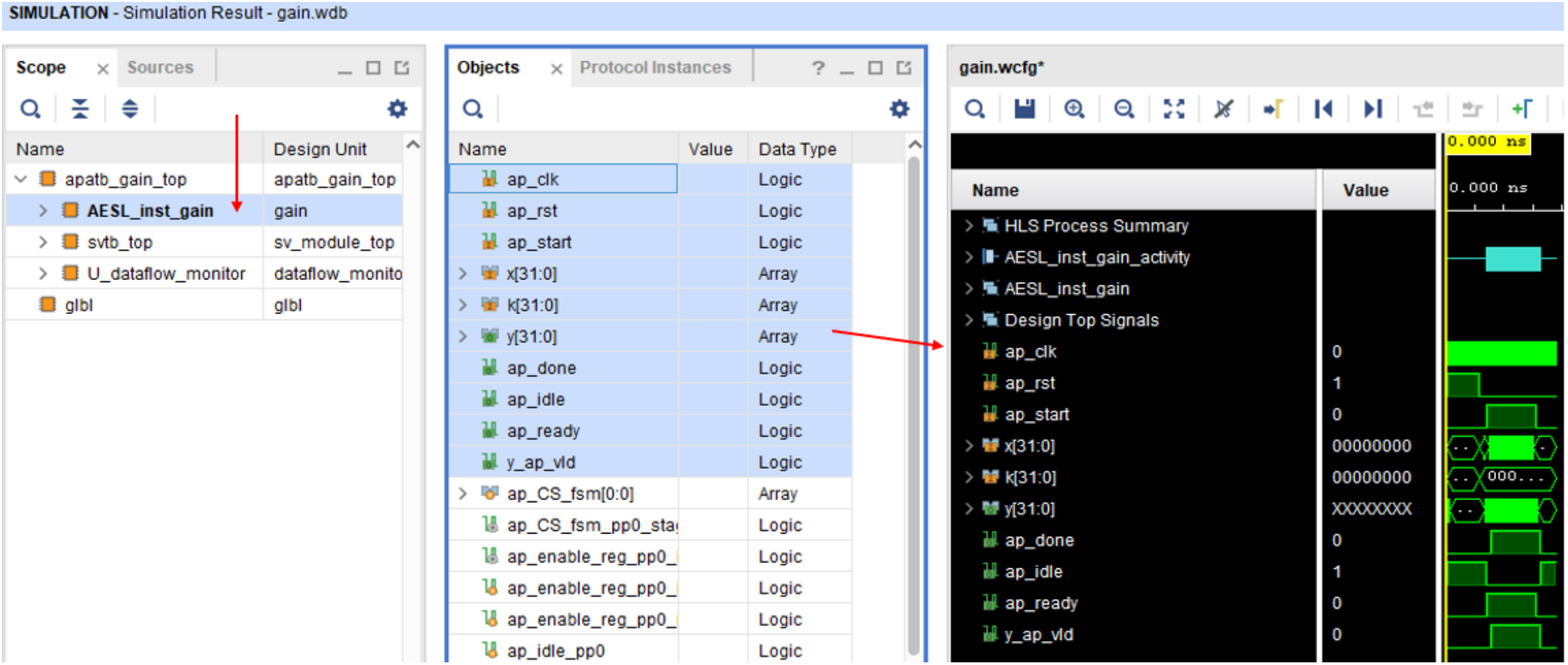
PORT	MODE	DIRECTION	BITWIDTH
k	ap_none	in	32
x	ap_none	in	32
y	ap_vld	out	32

2-4 C/RTL Co-Simulation

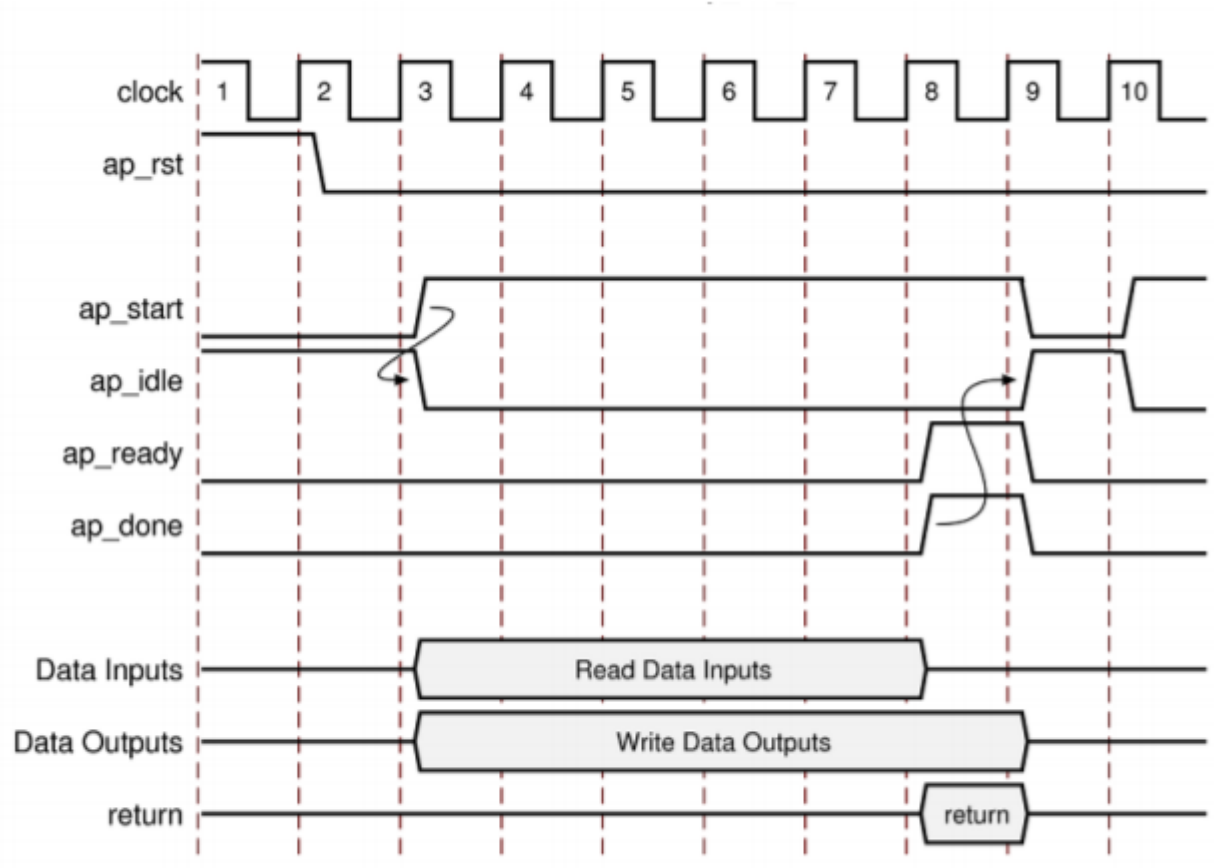
- Change data type back to int
- Run **C Synthesis**
- Run **C/RTL Cosimulation**
- Set: cosim.trace_level = port
- From **REPORTS** open **Wave Viewer**
  - The wave traces can be observed in Vivado Simulator

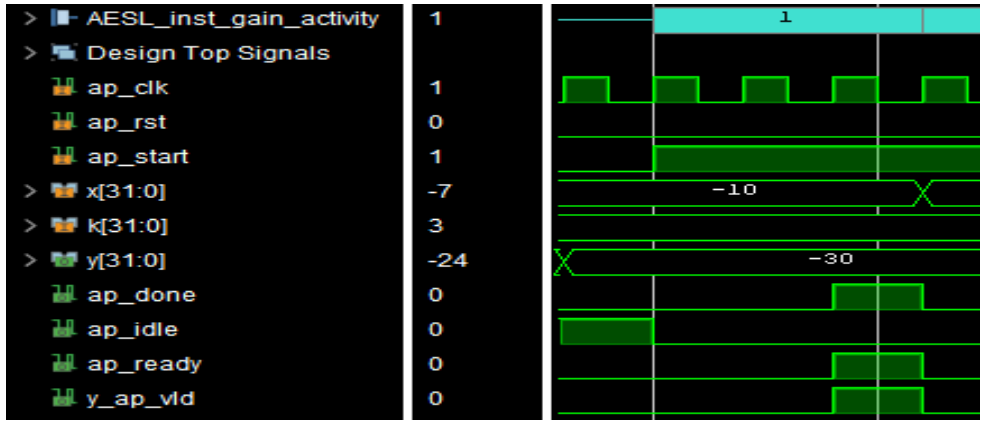


2-4 Vivado Simulation Result



2-4 Vivado Simulation Result

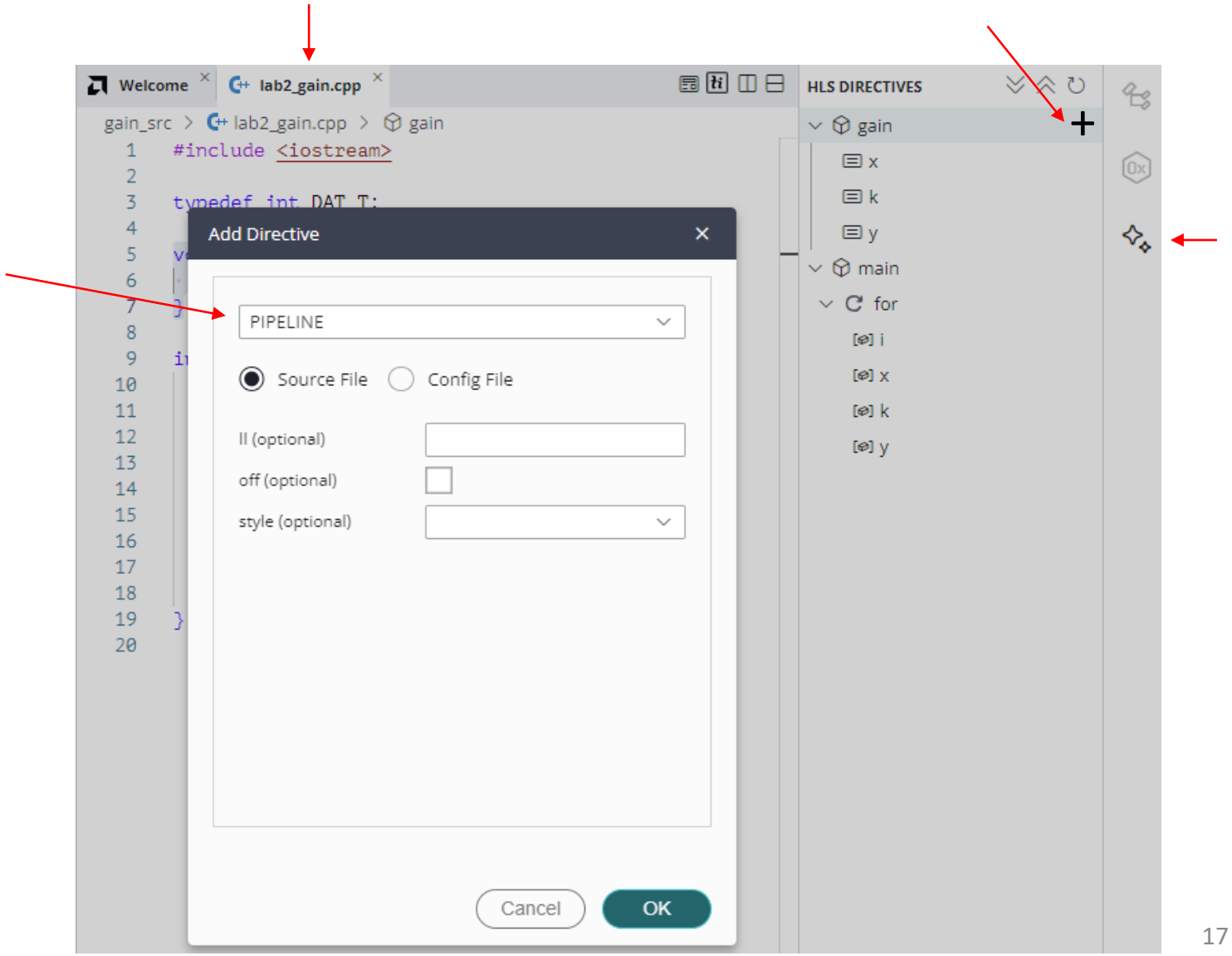




- The design starts when `ap_start` is asserted `High`.

2-5 HLS Directives

- View HLS Directives
- Add PIPELINE Directive to gain()
  - Click + near gain
  - Find PIPELINE, OK
- Run C Synthesis



2-5 HLS Directive PIPELINE

- Check synthesis report
  - compare

TARGET	ESTIMATED	UNCERTAINTY
8.00 ns	5.745 ns	2.16 ns

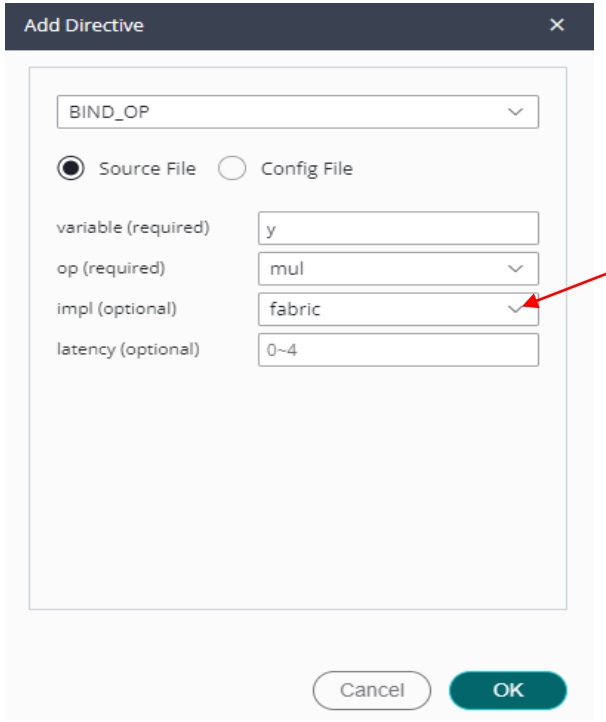
Performance & Resource Estimates

Modules Loops

MODULES & LOOPS	LATENCY(CYCLES)	INTERVAL	PIPELINED	STRUCTURE	BRAM	DSP	FF	LUT	URAM
gain	2	1	yes	function	0	3	169	51	0

2-5 HLS Directive BIND_OP

- Change data type to int_16t
- Add BIND_OP Directive to variable y and select
  - op=mul
  - impl=fabric
- Run C Synthesis
- Check synthesis report



Meaning: Do not use DSP

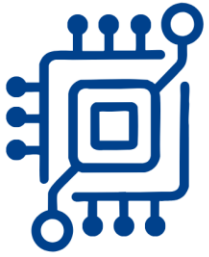
High-Level FPGA Design

Lab 3: Explore HLS Data Types – Gain IP

In this lab you will:

- write structured synthesizable C/C++ (header file, separate testbench)
- use arbitrary precision and fixed-point data types (ap_int, ap_fixed)
- understand how arithmetic operators on various data type map to FPGA hardware

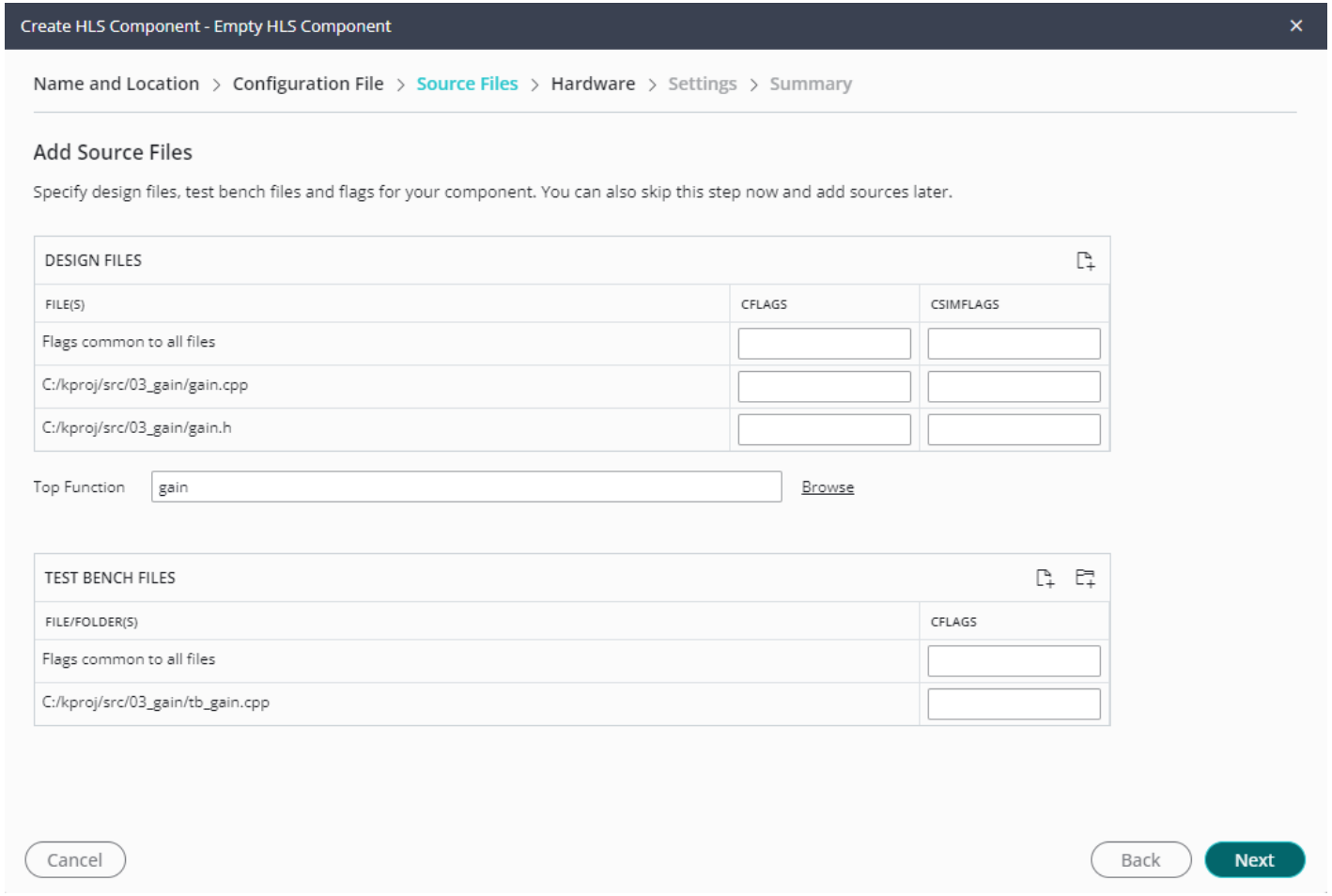
UNIVERSITY OF LJUBLJANA | FE Faculty of Electrical Engineering



CC CHIP.SI
SLOVENIAN CHIPS COMPETENCE CENTER

3-1 Create New Component

- File, New Component, Name: **gain_ip**
- Source: gain.cpp, gain.h
- Testbench: tb_gain.cpp
- Part xc7z010clg400-1



Arbitrary Precision Integer

- Data types now defined in header file
- **ap_int<N>**, N-bit integer

```
#ifndef _GAIN_H_
#define _GAIN_H_

#include <ap_int.h>
#include <ap_fixed.h>

typedef ap_int<16> DAT_T;

void gain(DAT_T x, DAT_T k, DAT_T& y);
#endif
```

Note

- include <ap_int.h>
- requires C++ compiler (template usage)
- unsigned arbitrary precision integer is **ap_uint<N>**

3-2 Run C Simulation

Testbench

- include header
- set k and values range
- Run C Simulation k=30 and k=50

```
DAT_T k = 30;

const int xmin = -1000;
const int xmax =  1000;
const int N = 10;
```

```
k = 30
x = -800, y = -24000, y_ref = -24000
x = -600, y = -18000, y_ref = -18000
x = -400, y = -12000, y_ref = -12000
x = -200, y = -6000, y_ref = -6000
x = 0, y = 0, y_ref = 0
x = 200, y = 6000, y_ref = 6000
x = 400, y = 12000, y_ref = 12000
x = 600, y = 18000, y_ref = 18000
x = 800, y = 24000, y_ref = 24000
x = 1000, y = 30000, y_ref = 30000
```

Overflow for ap_int<16>

```
k = 50
x = -800, y = 25536, y_ref = -40000
x = -600, y = -30000, y_ref = -30000
x = -400, y = -20000, y_ref = -20000
x = -200, y = -10000, y_ref = -10000
x = 0, y = 0, y_ref = 0
x = 200, y = 10000, y_ref = 10000
x = 400, y = 20000, y_ref = 20000
x = 600, y = 30000, y_ref = 30000
x = 800, y = -25536, y_ref = 40000
x = 1000, y = -15536, y_ref = 50000
```

3-2 Change Data Type & Run C Simulation

16 x 16 bit = 32 bits

- Declare result data type RES_T in header file
- Modify function declaration
- Modify variable y declaration in function and in the testbench
- Check with C Simulation

```
typedef ap_int<32> RES_T;

void gain(DAT_T x, DAT_T k, RES_T& y);
```

3-3 Explore Synthesis - Data Types & PIPELINE

- Run C Synthesis
  - Set clk: 4ns
- Check Performance & Resource Estimates
  - e.g. 16-bit or 24-bit input
- Add directive PIPELINE
- Run C Synthesis
- Fill the table

DAT_T, RES_T	Latency	Interval	DSP	FF	LUT
16, 32	3	4	1	73	26
24, 48					
16, 32, pipeline					
24, 48, pipeline					

Fixed-point Data Type

ap_fixed<W, I>

- W-bit binary number with I integer bits

```
#include <ap_fixed.h>

typedef ap_fixed<16,1> DAT_T;
typedef ap_fixed<16,1> RES_T;
```

Example: **ap_fixed<5,2>**

```
01.000, decimal 1
00.100, decimal 0.5
01.100, decimal 1.5
10.000, decimal -2 (binary complement)
Values range: -2 to 1.875
```

Note: ap_fixed<N,1> are real numbers in range from -1 to 1 (1 is excluded)
Normalized values in interval (-1,1) are common in digital signal processing.

3-4 Synthesis

- **Run C Synthesis,** set clock 8ns
- Is the circuit utilization different compared to 16-bit integer?

3-5 Fixed-point C Simulation

- Define new data type for k
  - range: -32 to 31.999
- Modify function declaration
- Modify the testbench
- Check with C Simulation

```
typedef ap_fixed<16,1> DAT_T;
typedef ap_fixed<16,1> RES_T;
typedef ap_fixed<16,6> GAIN_T;
```

```
void gain(DAT_T x, GAIN_T k, RES_T& y);
```

k = 20

xmin = -0.1

xmax = 0.1

```
k = 20
x = -0.100006, y = -0.00012207, y_ref = -2.00012
x = -0.0800171, y = 0.399658, y_ref = -1.60034
x = -0.0600281, y = 0.799438, y_ref = -1.20056
x = -0.0400085, y = -0.800171, y_ref = -0.800171
```

3-5 Updated Test Bench

```
int main() {
    GAIN_T k = 20;
    std::cout << "k = " << k << std::endl;

    const float xmin = -0.1f;
    const float xmax =  0.1f;
    const int N = 10;
    const float step = (xmax - xmin) / (N - 1);

    for (int n = 0; n < N; n++) {
       DAT_T x = xmin + n * step;
       RES_T y;
       gain(x, k, y);

       float y_ref = k * x;

       std::cout << "x= " << x << ", y= " << y << ", ref= " << y_ref << "\n";
    }
    return 0;
}
```

3-5 Fixed-point Saturation

ap_fixed<W, I, QMODE, OMODE>

- Quantization QMODE
  - Default is truncate bits, fast, less accurate
  - **AP_RND** — round to nearest
- Overflow OMODE
  - Default is wrap around
  - **AP_SAT** — saturate to min/max representable value
- Modify RES_T
- Check with C Simulation
  - Saturation instead of overflow

```
typedef ap_fixed<16,1, AP_RND, AP_SAT> RES_T;
```

3-6 Explore Synthesis - Data Types & PIPELINE

- Run C Synthesis
  - Set clk: 4ns
- Check Performance & Resource Estimates
- Fill the table

DAT_T, RES_T	Latency	Interval	DSP	FF	LUT
16, 1	3	4	1	73	26
16, 1, saturate					
16, 1, saturate, pipeline					
14, 1, saturate, pipeline					

3-7 Prepare IP Component for Red Pitaya

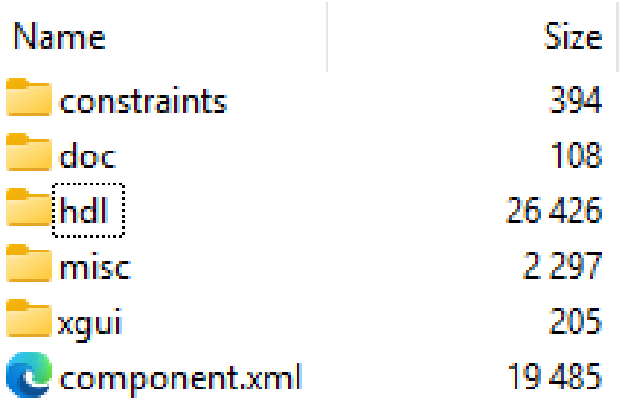
- Change input and output data to 14-bits (14-bit ADC/DAC)

```
typedef ap_fixed<14, 1> DAT_T;
typedef ap_fixed<14, 1, AP_RND, AP_SAT> RES_T;
```

- Add directives to gain()
  - PIPELINE
  - INTERFACE mode=ap_ctrl_none
- Run C Synthesis

3-7 Package IP Component

- Run PACKAGE
  - set output file: **gain_ip** and description: HLS gain component
- Explore Output
  - open ZIP file with IP sources `proj\vitis\gain_ip\gain_ip.zip`



*Lab Experiments

- add offset: y = x * k + offset
- synthesize with different data types
- compare fixed-point and floating-point utilization

High-Level FPGA Design

Lab4: Averaging Filter in HLS IP

In this lab you will:

- Re implement the 16 bit 8 tap averaging filter in Vitis HLS
- Express pipeline, latency, and parallelism using HLS pragmas
- Achieve 1 sample per clock throughput (II = 1)
- Analyze HLS synthesis reports and compare them to RTL expectations

Lab Sources

- Project folder structure

```
proj/
  ├── src/
  │     ├── 04_avg_filter/        ← lab4 C++ sources
  │     │     ├── avg_filter.cpp
  │     │     ├── avg_filter.h
  │     │     └── tb_ avg_filter.cpp
  │
  ├── vitis/                ← Vitis workspace (location)
  │     ├── avg_filter/     ← lab4 HLS component
```

4-1 Create a New Vitis HLS Component

- Create Component…, Create Empty HLS Component
- Component name: **avg_filter**
  Configuration File: Empty File
- Add Source Files: avg_filter.cpp, avg_filter.h
  - Top function: **avg_filter**
- Add Test Bench Files tb_avg_filter.cpp
- Part xc7z010clg400-1, clk=8ns

4-1 Check design description

- data type of 16-bit integer data (defined in ap_int.h)
- internal **static** array (to store values between function calls)
- the function describes only shift register

```
#include "avg_filter.h"

void avg_filter(
    ap_int<16> in,
    ap_int<16> &out
) {
    static ap_int<16> shift_reg[NTAPS];

    // Shift register
    for (int i = NTAPS - 1; i > 0; i--) {
        shift_reg[i] = shift_reg[i - 1];
    }
    shift_reg[0] = in;

    out = shift_reg[NTAPS-1]; // temporary output
}
```

```
#ifndef AVG_FILTER_H
#define AVG_FILTER_H

#include <ap_int.h>

#define NTAPS 8

void avg_filter(
    ap_int<16> in,
    ap_int<16> &out
);

#endif
```

4-2 Initial C Synthesis

- Run **C Synthesis** and check Performance & Resource Estimates
- Check **Bind Storage: ram_2p**

Latency	Interval	DSP	FF	LUT
12	13	0	47	137

4-3 Simulation

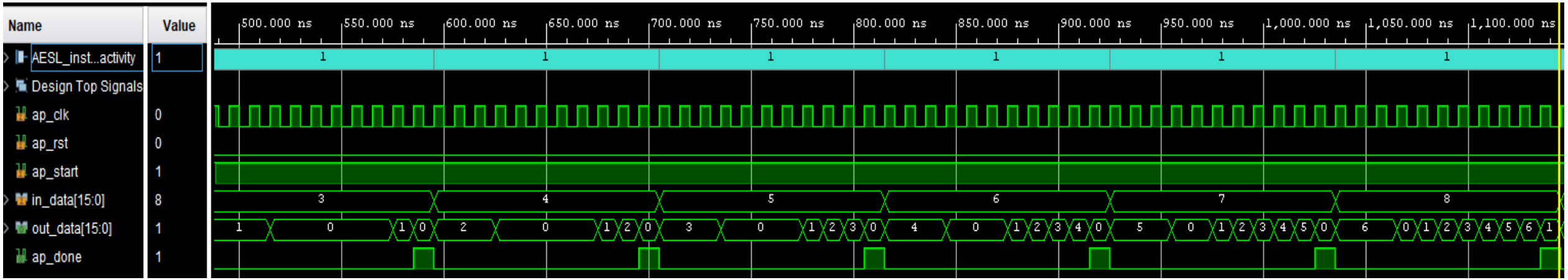
- Run C **Simulation**
- Add for loop to a Test Bench
  - Check delayed output

```
for (in = 1; in < 20; in++) {
    avg_filter(in, out);
    std::cout << "[TB] in=" << in << " out=" ..
}
```

```
[TB] in=1 out=0
[TB] in=2 out=0
[TB] in=3 out=0
[TB] in=4 out=0
[TB] in=5 out=0
[TB] in=6 out=0
[TB] in=7 out=0
[TB] in=8 out=1
[TB] in=9 out=2
[TB] in=10 out=3
```

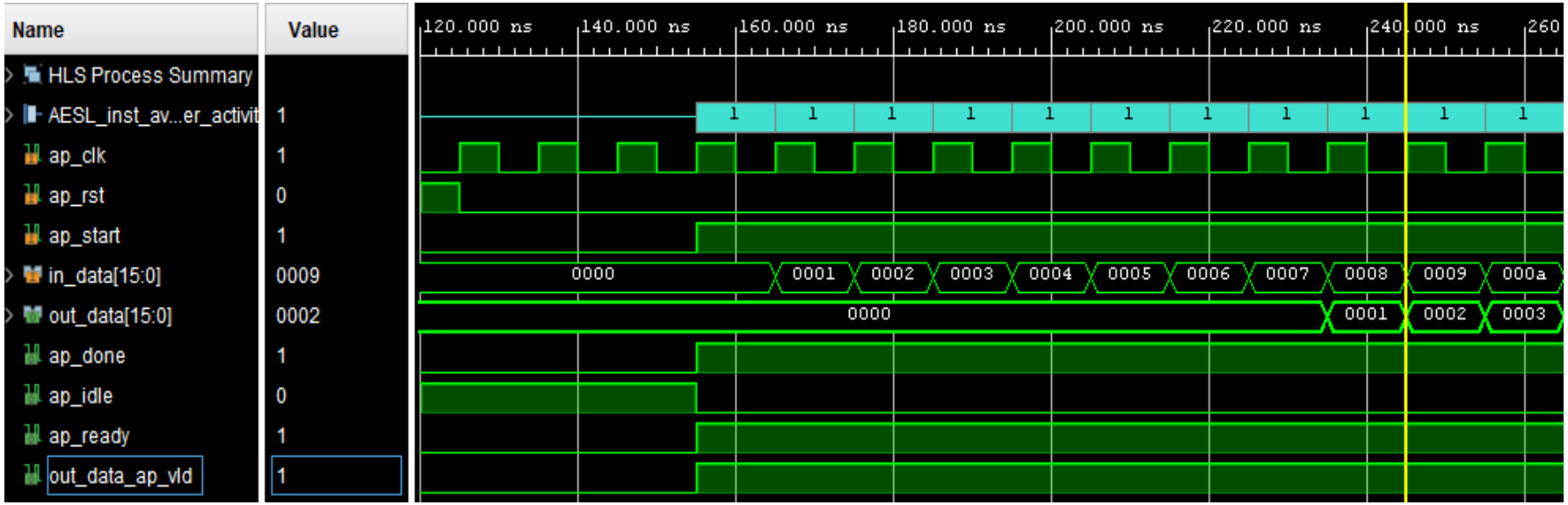
4-3 C/RTL Co-Simulation

- Run **Cosimulation**, set: cosim.trace_level = port
- From **REPORTS** open **Wave Viewer**
- Observe circuit interface control signals and output data
  - Check for the first valid nonzero output



4-4 Directives

- Open **avg_filter.cpp** in Vitis Editor and add label to for loop, SLOOP: for …
- Add directive **UNROLL** on SLOOP
- Run **C Synthesis** and Examine Synthesis Report, write data to the table
- Check operation with C/RTL Cosimulation



4-5 Compute Average

- Add accumulation loop (declare sum) and output division
- Add **UNROLL** directive to this loop

```
ap_int<32> sum = 0;

// Accumulate
MLOOP: for (int i = 0; i < NTAPS; i++) {
#pragma HLS UNROLL
        sum += shift_reg[i];
 }
// divide by 8
 out = sum >> 3;
```

4-5 Compute Average, Simulate

- Check operation with **C Simulator**
  - test with input sequence: 0, 2, 4, 6, 8, 10, … `avg_filter(2*in, out);`

input	0	2	4	6	8	10	12	14	16
output	0	0	0	1	2	3	5	7	9

  - and with constant input: 100

input	100	100	100	100	100	100	100	100	100
output	12	25	37	50	62	75	87	100	100

*4-6 Optimize Average Computation Algorithm

- Modify code to compute running sum (sum variable should be static!)
- Check operation with **C Simulator**
  - Compare results with previous algorithm ouput
- Run **C Synthesis,** Examine Synthesis Report
- Run Implementation, Check Resource Usage