DIGITAL DESIGN

LAB7 INTRODUCTION TO HLS

2022 SUMMER TERM

CONTENTS

- 1. Introduction to HLS Tools
- 2. Principles of HLS
- 3. Optimization Constraints
- 4. Example: Matrix Multiplier

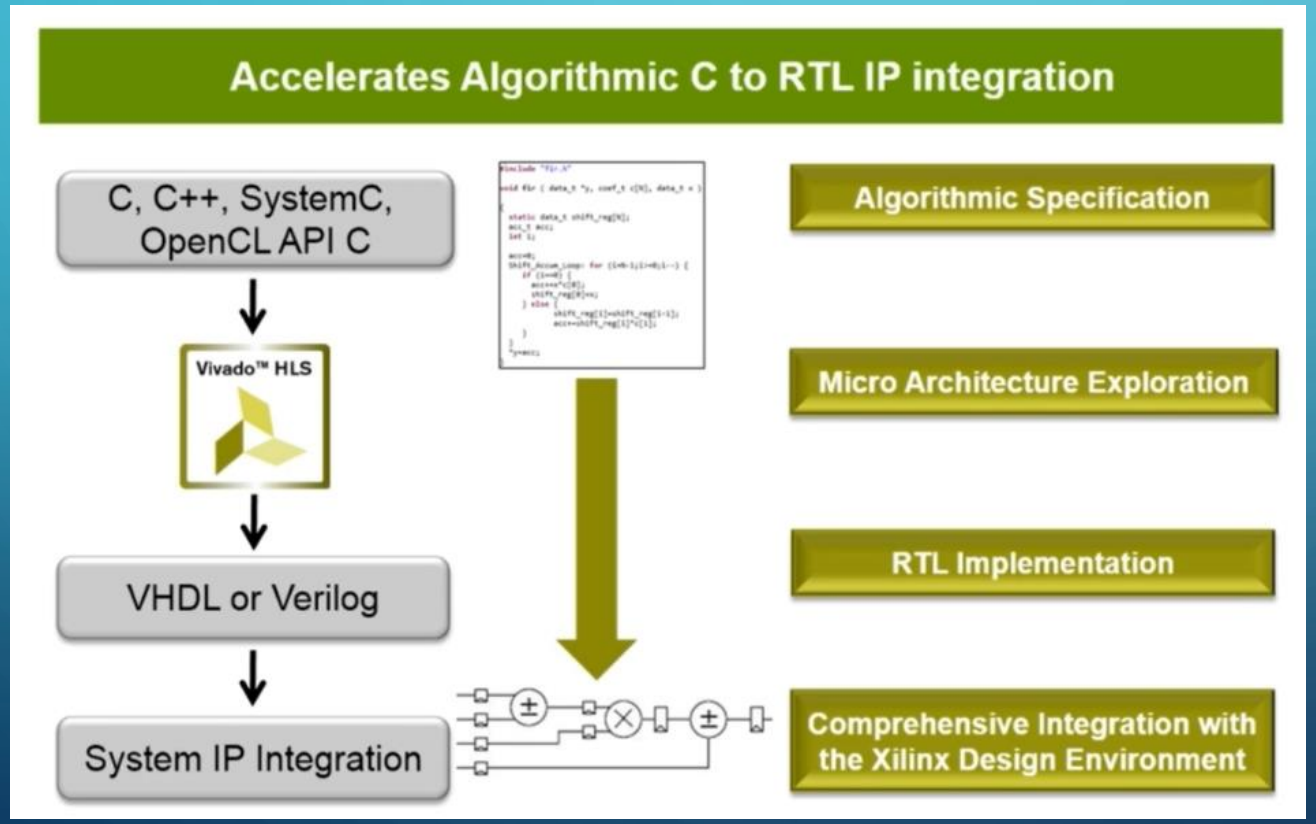
INTRODUCTION TO HLS TOOLS

- 1. High-Level Synthesis:
  - The Xilinx® Vivado® High-Level Synthesis (HLS) compiler provides a programming environment similar to those available for application development on both standard and specialized processors. Vivado HLS shares key technology with processor compilers for the interpretation, analysis, and optimization of **C/C++ programs**.
  - By targeting an FPGA as the execution fabric, Vivado HLS enables a software engineer to optimize code for throughout, power, and latency without the need to address the performance bottleneck of a single memory space and limited computational resources. This allows the implementation of computationally intensive software algorithms into actual products, not just functionality demonstrators.

Refer to UG998 for further understanding

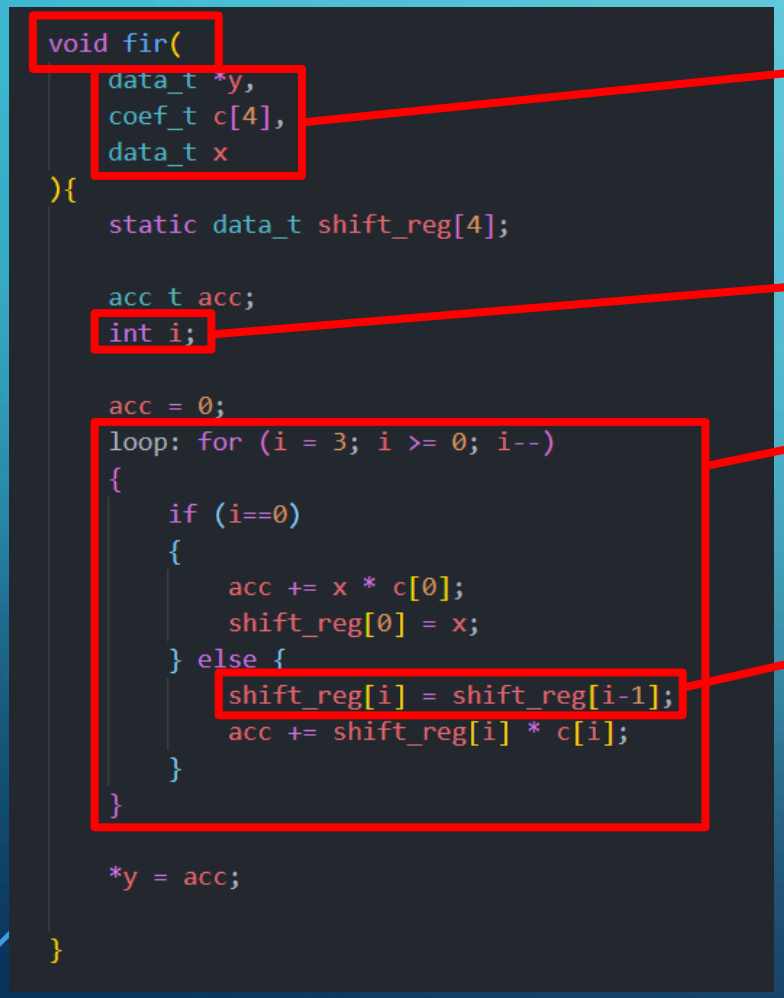
INTRODUCTION TO HLS TOOLS

- 1. High-Level Synthesis:
- Algorithm → C/CPP code → HDL → Circuits (on FPGA)



INTRODUCTION TO HLS TOOLS

- 2. Basic Components:



```
void fir(
    data_t *y,
    coef_t c[4],
    data_t x
){
    static data_t shift_reg[4];

    acc_t acc;
    int i;

    acc = 0;
    loop: for (i = 3; 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;

}
```

- **Functions**: All code is made up of functions which represent the design hierarchy. (Module)
- **Top-level IO**: The arguments of the top-level function determine the hardware RTL interface ports (both input and output).
- **Types**: All variables are of a defined type. The type can influence the area and performance. (Wire/Reg)
- **Loops**: Functions typically contain loops. How these are handled can have a major impact on area and performance.
- **Arrays**: Arrays are used often in C code. They can influence the device IO and become performance bottlenecks. (BRAM)
- **Operators**: Operators in the C code may require sharing to control area or specific hardware implementations to meet performance

PRINCIPLES OF HLS

- 1. Operations

```
A[i] = B[i] * C[i];
D[i] = B[i] * E[i];
F[i] = A[i] + D[i];
```

Figure 4-1: **Example Code for Three Operations**

Vivado HLS analyzes and abstracts into memory accessing and algorithm operations, optimize based on dependency, parallelize the execution instead of sequential operations. **More hardware**

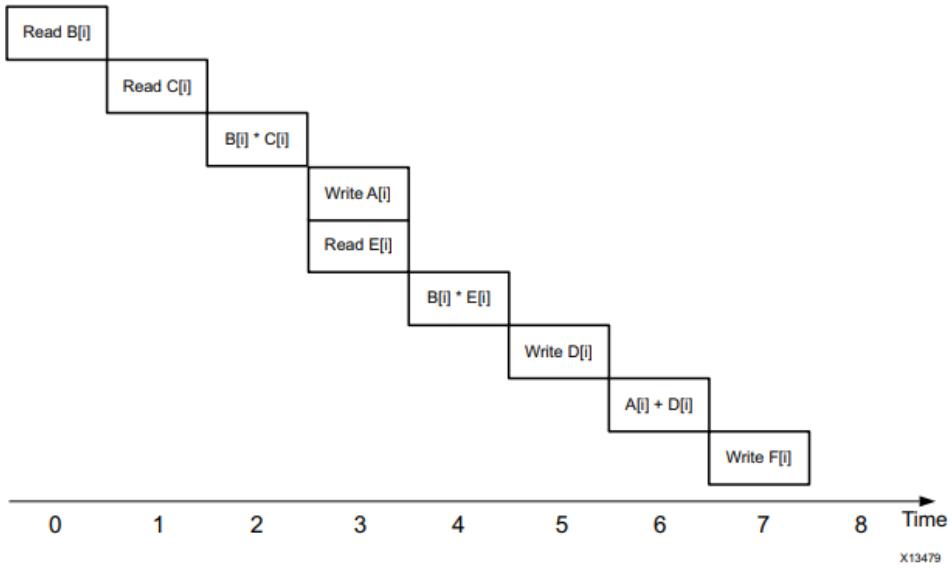


Figure 4-2: **Execution of Example Code on a Processor**

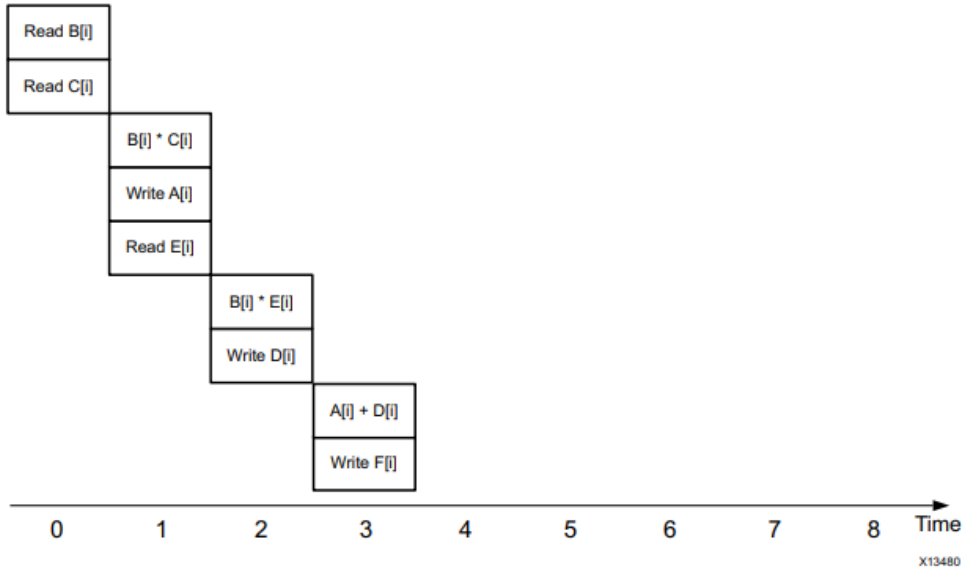
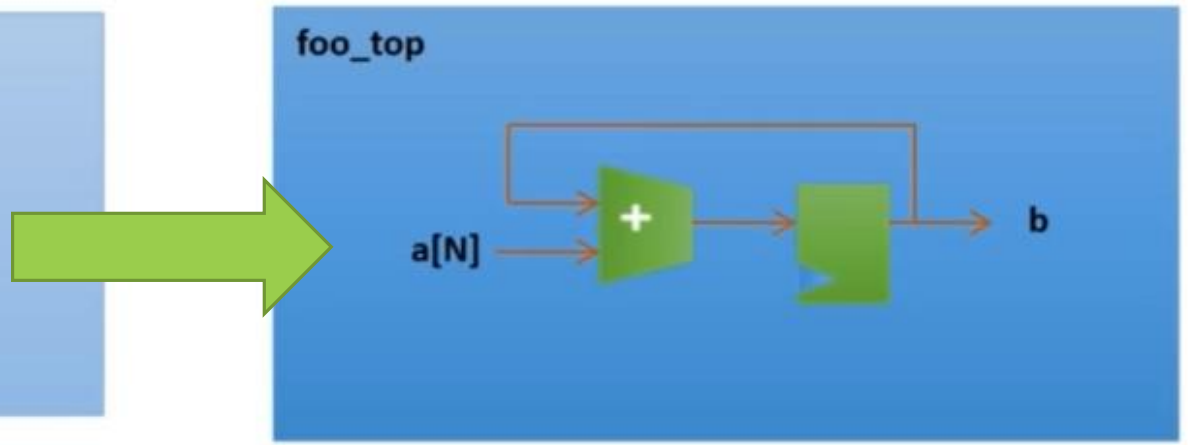


Figure 4-3: **Default Execution of HLS Code on an FPGA**

PRINCIPLES OF HLS

- 2. Loops
- N loops will result in N clock cycles for calculation.
- Can be optimized by adding constrains(will be discussed in detail later)

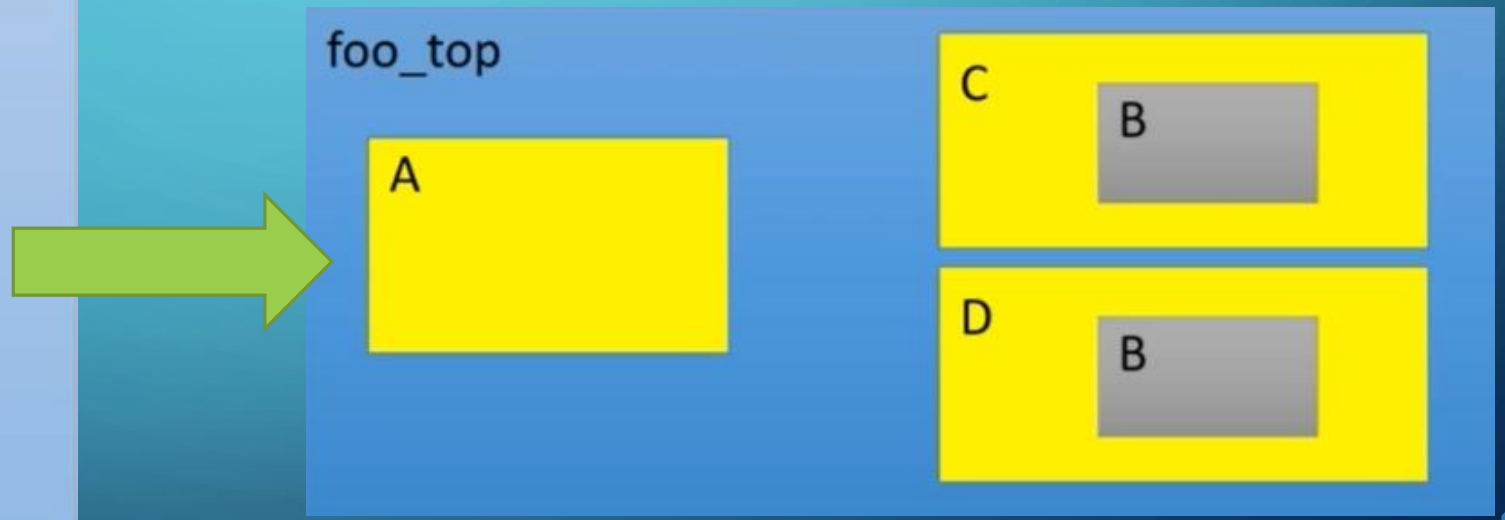
```
void foo_top (…) {
  ...
  Add: for (i=3;i>=0;i--) {
    b = a[i] + b;
  ...
  }
```



PRINCIPLES OF HLS

- 3. Submodules
- Every subfunctions inside a function will generate a submodule, forming **Hierarchy**

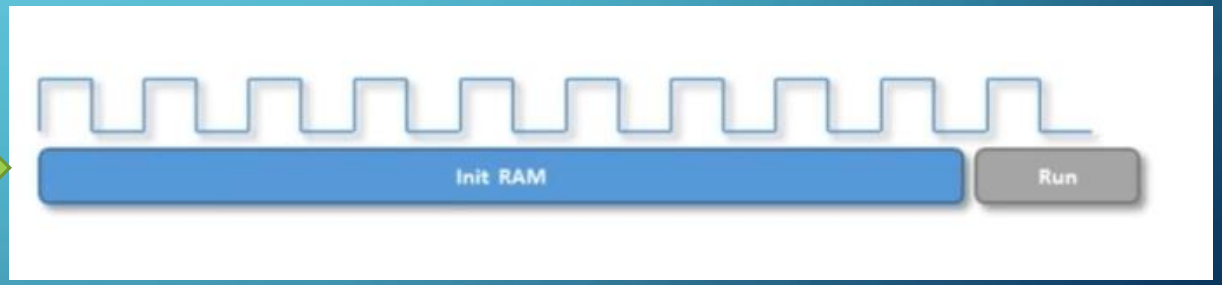
```
void A() { ..body A..}
void B() { ..body B..}
void C() {
        B();
}
void D() {
        B();
}


void foo_top() {
        A(...);
        C(...);
        D(...)
}
```



PRINCIPLES OF HLS

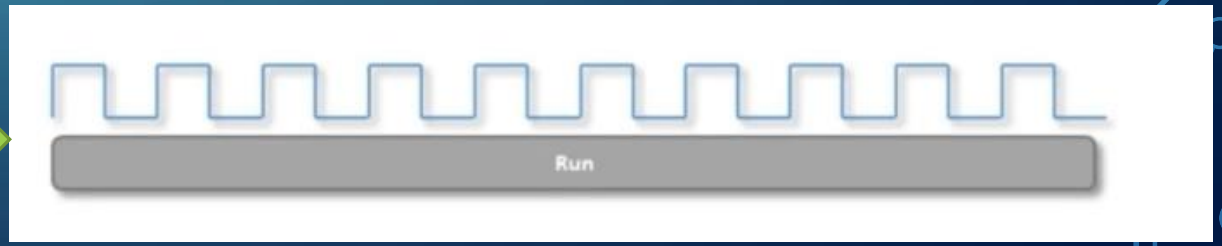
- 4. Arrays
  - Static and Const have great impact on array implementation
  - Initialized arrays implies a ROM and need cycles to initialize every time the function is executed. In hardware, a ROM uses some clock cycles to initialize when power is on.

```
int coeff[8] = {-4, -3, -2, -1, 0, 1, 2, 3}
```



- Instead, static arrays can hold the value since last changes, matching the functionality of a RAM, thus it will use the BRAM resource on FPGA.

```
static int coeff[8] = {-4, -3, -2, -1, 0, 1, 2, 3}
```



PRINCIPLES OF HLS

- 5. Pointers
  - The HLS compiler supports pointer usage that can be completely known at compile time.

```
int A[10];
int *pA;
pA = A;
```

  - Another supported model for pointer is accessing external memory. HLS defines an external memory as outside of the scope of compiler-generated RTL. The address can be represented as pointers.

```
void foo (int *data_in,...) {
    int item1, item2, item3;
    item1 = *data_in;
    item2 = *(data_in + 1);
    item3 = *(data_in + 2);
}
```

PRINCIPLES OF HLS

- 6. Limitation
- Vivado HLS can compile almost any C/C++ program. The only coding limitation for Vivado HLS is with dynamic language constructs typical in processors with a single memory space. When using Vivado HLS, the main dynamic constructs to consider are **memory allocation** and **pointers** with runtime information.

Table 4-2: **Functions Used in Dynamic Memory Management**

C	C++
malloc()	new()
calloc()	delete()
free()	

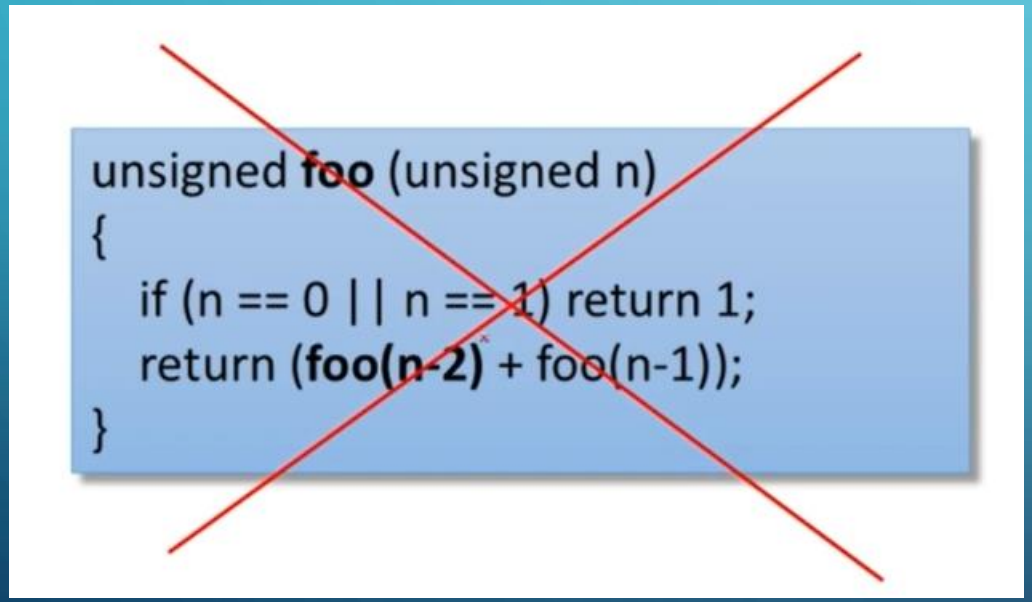
```
int *A = malloc(10*sizeof(int));
```

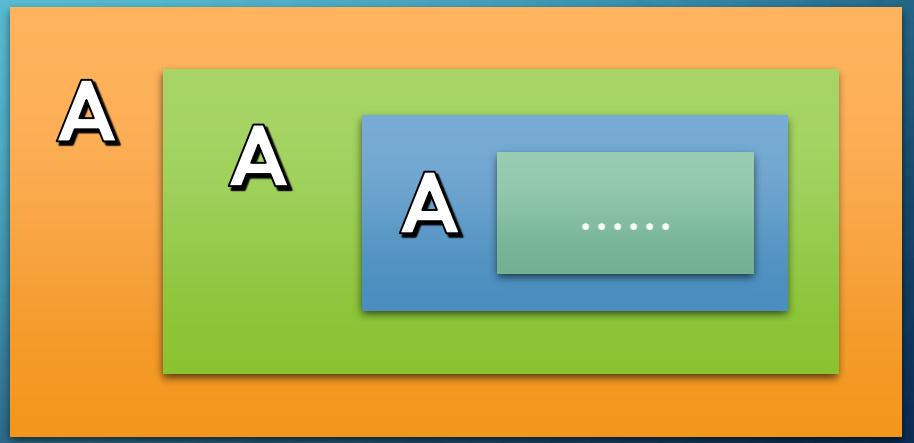
Figure 4-8: **Dynamic Memory Allocation**

HLS cannot synthesize code that includes any of the keywords in Table 4-2 even if the allocation is constant, as in the example shown in Figure 4-8. Use static arrays instead.

PRINCIPLES OF HLS

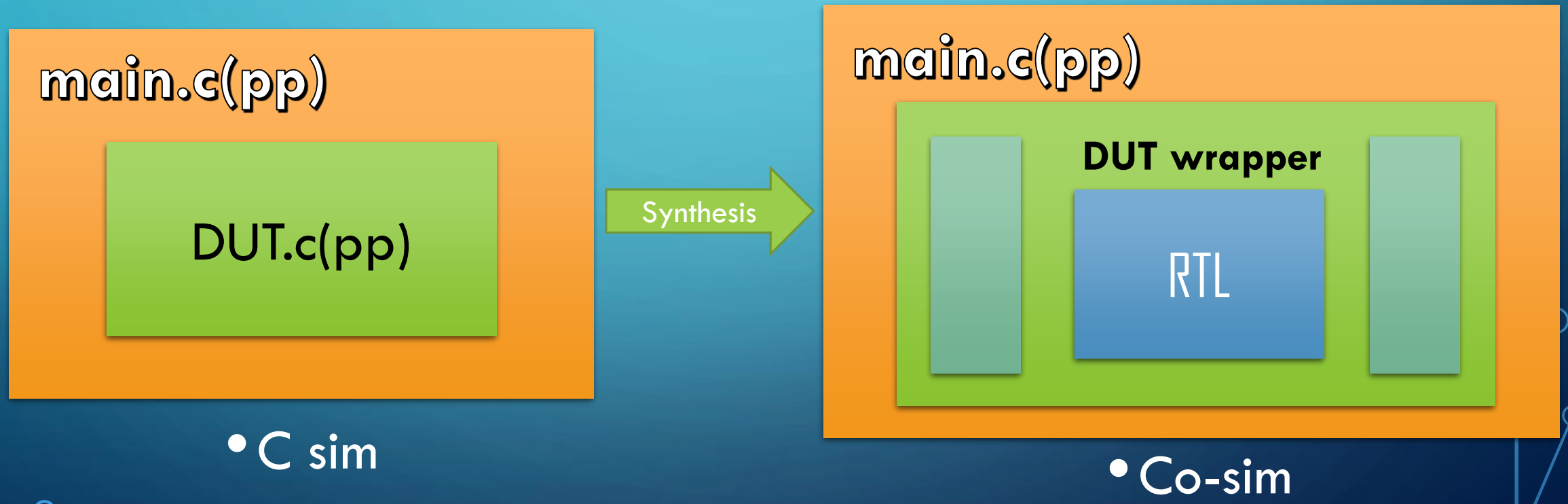
- 6. Limitation
- **Recursion** is not supported, since it cannot synthesize how many modules to generate when calls itself.





PRINCIPLES OF HLS

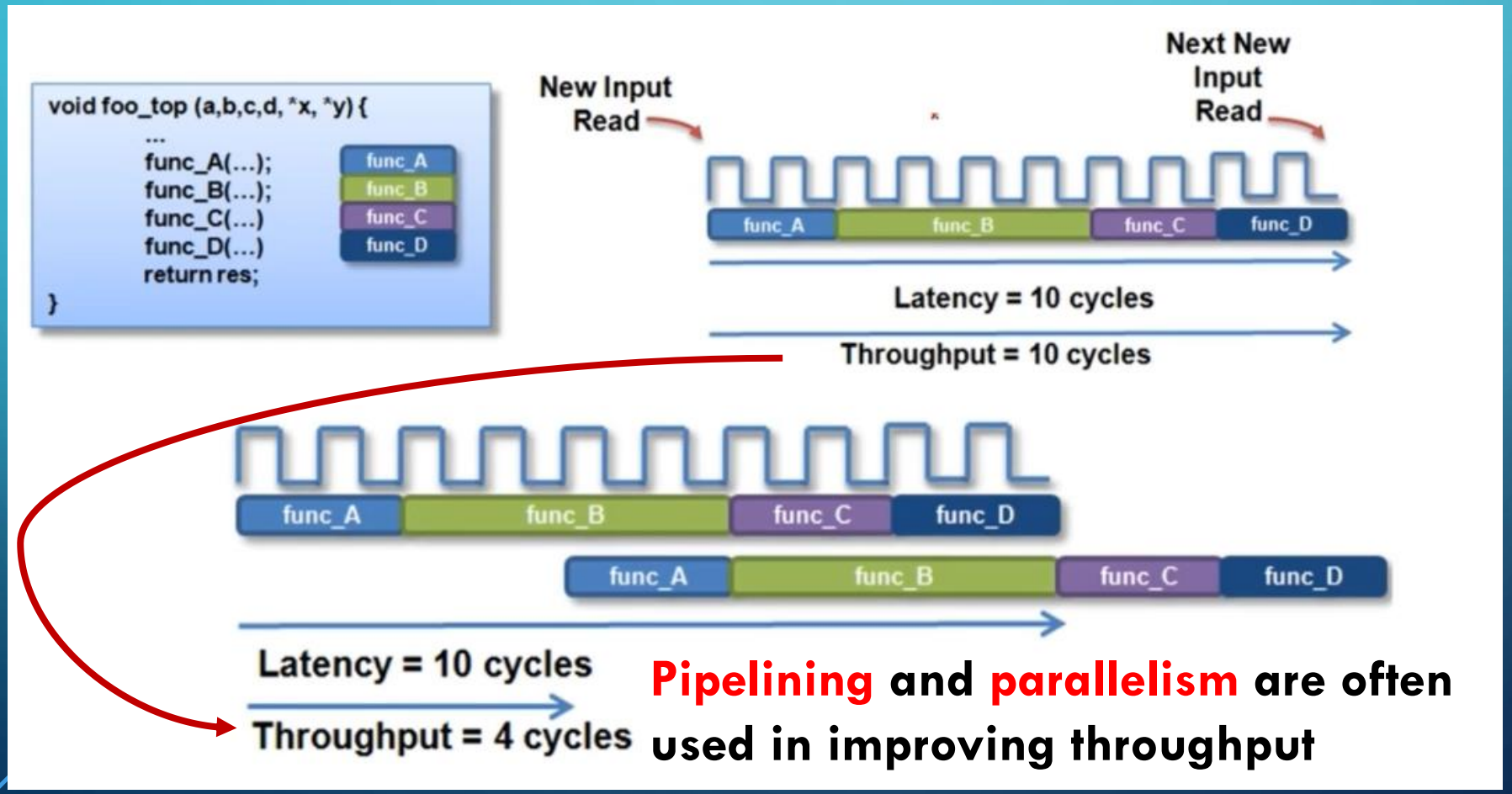
- 7. Testbench
- Automatically construct a testbench in co-simulation (use C code to simulate RTL code)



- C sim
- Co-sim

OPTIMIZATION CONSTRAINTS

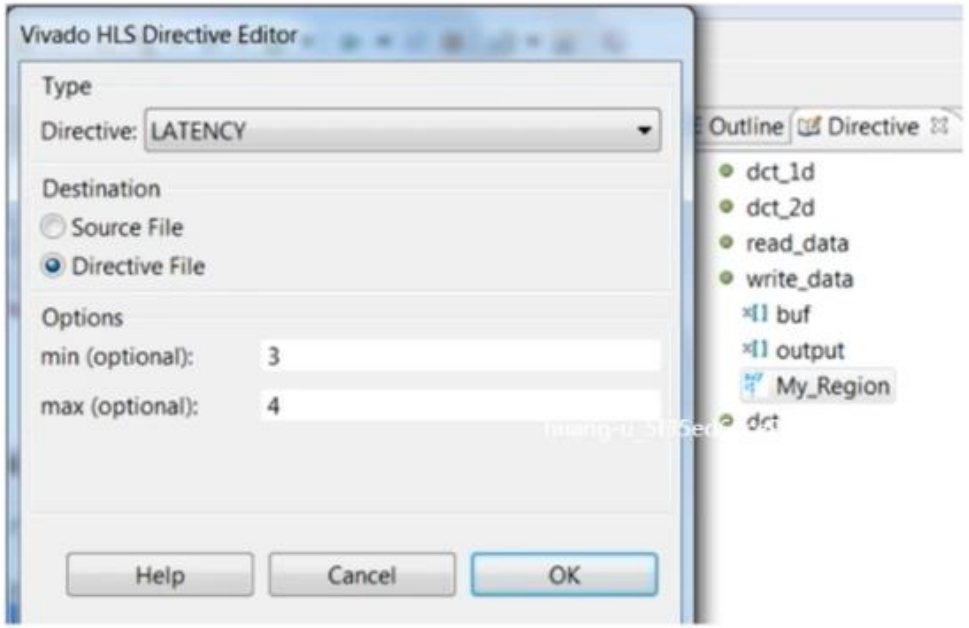
- Latency and Throughput(Interval)



OPTIMIZATION CONSTRAINTS

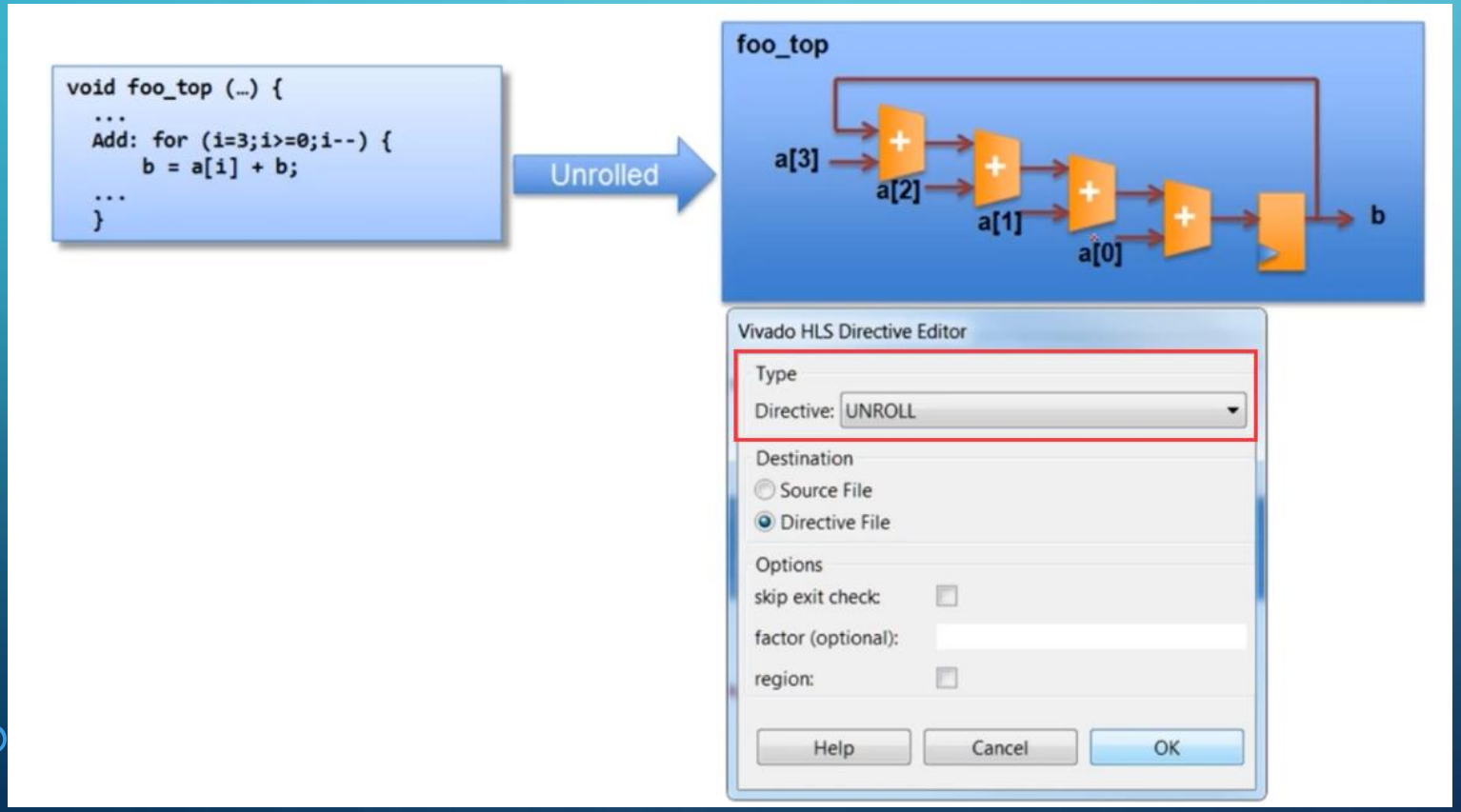
- 1. Latency Constraints
- Latency derivatives can be applied on functions, loops, and regions.

```
int write_data (int buf, int output) {

if (x < y) {
      return (x + y);
} else {

      My_Region: {

            return (y – x) * (y + x);

      }

}
```



OPTIMIZATION CONSTRAINTS

- 2. Loop Unroll
- Some loops can be unrolled to improve performance.



OPTIMIZATION CONSTRAINTS

- 3. Loop Flattening
- Some loops can be flattened from 2D to 1D to decrease loop iterations.

```
Loop_outer: for (i=3;i>=0;i--) {
   Loop_inner: for (j=3;j>=0;j--) {
       [Loop body]
   }
}
```

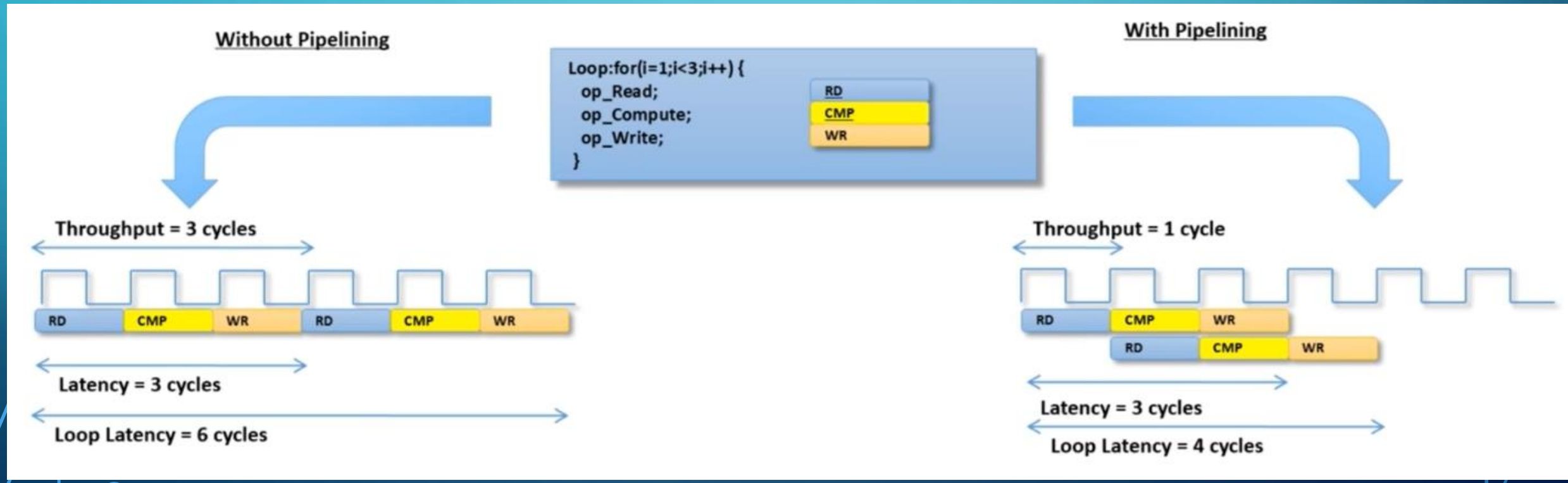

```
  L2: for (k=15,k>=0;k--) {

     [Loop body l3 ]
}
```

If there are statements inside two loops, they cannot be flattened.

```
Loop_outer: for (i=3;i>N;i--) {
   [Loop body]
   Loop_inner: for (j=3;j>=M;j--) {
       [Loop body]
   }
}
```

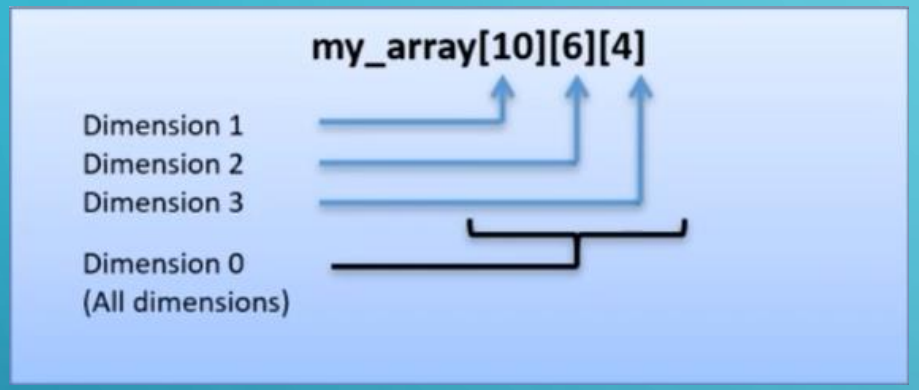
OPTIMIZATION CONSTRAINTS

- 4. Pipelining
- Pipeline constrains can be added to optimize loop interval.



OPTIMIZATION CONSTRAINTS

- 5. Array partition



Partition into different dimensions will generate multiple RAMs which can be accessed simultaneously for higher parallelism and pipeline.

Partition into dimension 0 means 240 bit registers instead of BRAM

my_array_0[10][6]
my_array_1[10][6]
my_array_2[10][6]
my_array_3[10][6]

Partition dimension 3

Partition dimension 1

my_array[10][6][4]

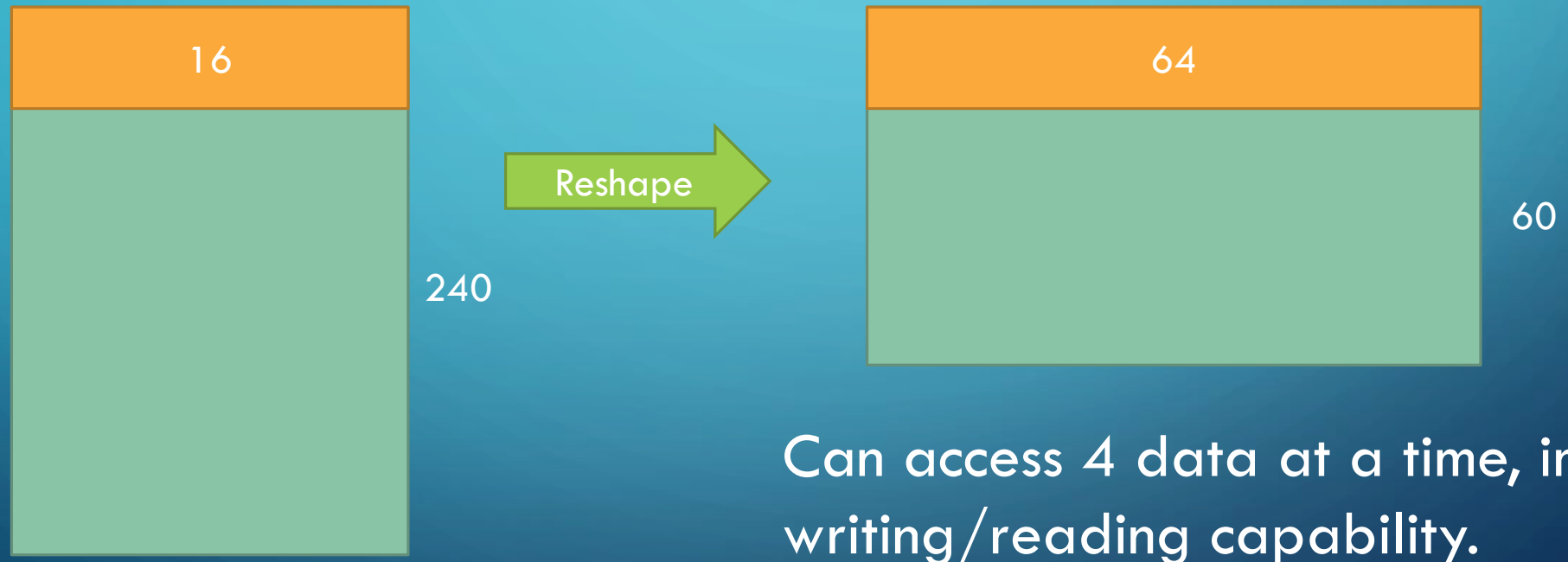
Partition dimension 0

my_array_0[6][4]
my_array_1[6][4]
my_array_2[6][4]
my_array_3[6][4]
my_array_4[6][4]
my_array_5[6][4]
my_array_6[6][4]
my_array_7[6][4]
my_array_8[6][4]
my_array_9[6][4]

10*6*4=240 240 individual registers

OPTIMIZATION CONSTRAINTS

- 6. Array Reshaping
- Change the shape of an array to improve io bandwidth.



Can access 4 data at a time, improve writing/reading capability.

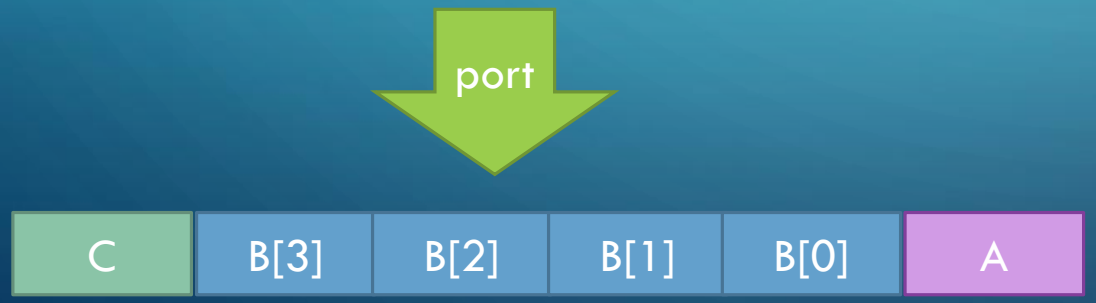
OPTIMIZATION CONSTRAINTS

- Reshaping vs Partitioning
- Both are useful for increasing the memory or data bandwidth.
- **Reshaping**
  - Simply increases the width of the data word
  - Does not increase the number of memory ports
- **Partitioning**
  - Increases the memory ports, thus more I/o to deal with
  - Use it only if you have to use independent addressing

OPTIMIZATION CONSTRAINTS

- 7. Data Packing
- Package data into a struct to share control logic.

```
typedef struct{
        unsigned char A;
        unsigned char B[4];
        unsigned char C;
} my_data;
void foo(my_data *a)
```



Grouped structure

- First element in the struct becomes the LSB
- Last struct element becomes the MSB
- Arrays are partitioning completely

On the Interface

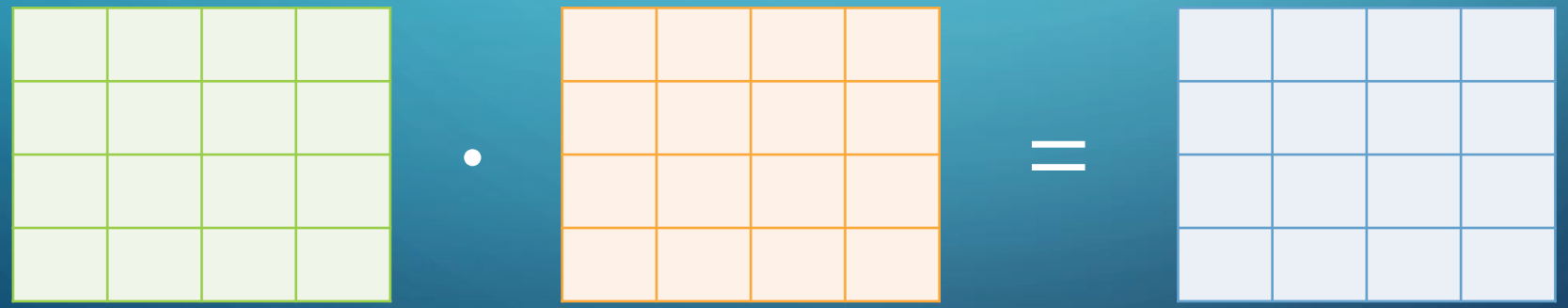
- This means a single port

Internally

- Single bus
- May result in simplified control logic, faster and lower latency designs

EXAMPLE: MATRIX MULTIPLIER (4X4)*(4X4)

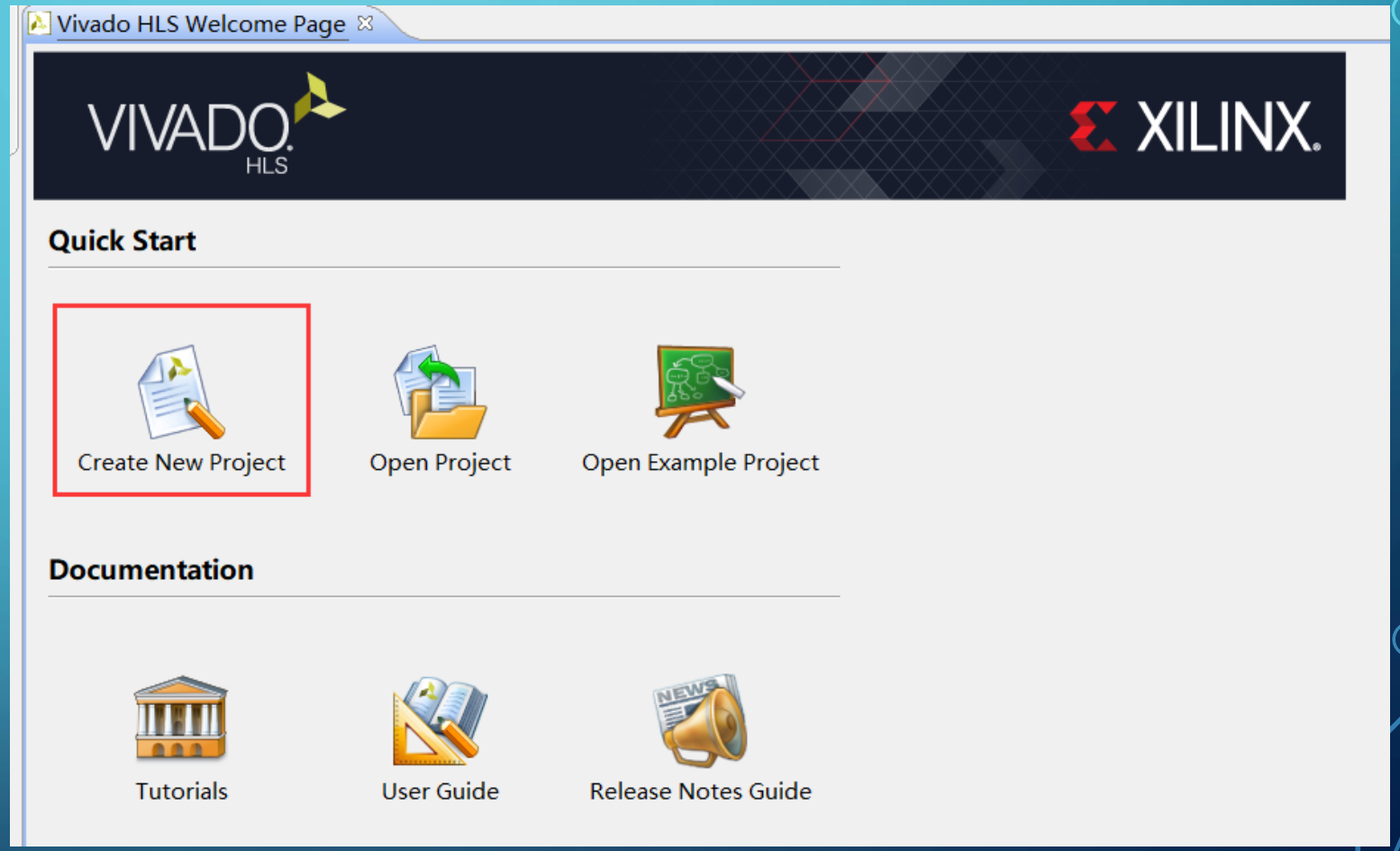
- Let's try to use HLS to generate a matrix multiplier by ourselves.
- Input: **A[4][4]** (8-bit), **B[4][4]** (8-bit)
- Output: **C[4][4]** (18-bit)
- Objective: C=A·B



EXAMPLE: MATRIX MULTIPLIER

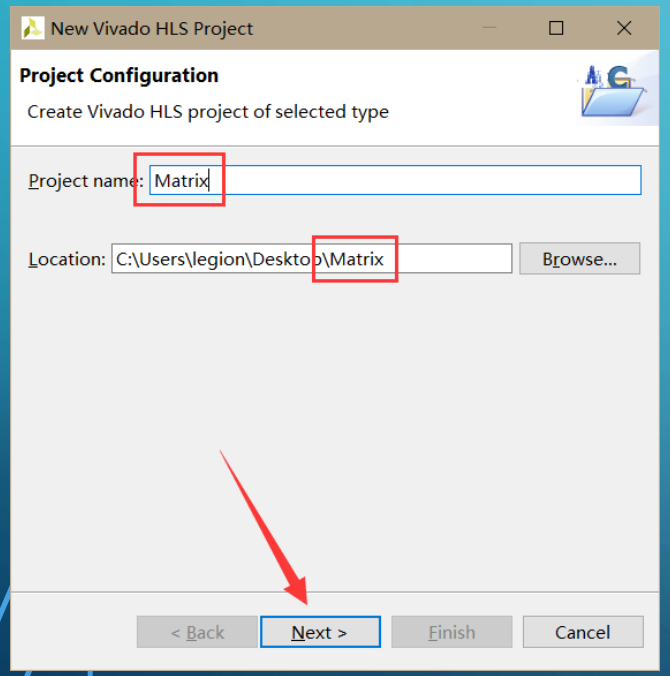
- 1. Open Vivado HLS

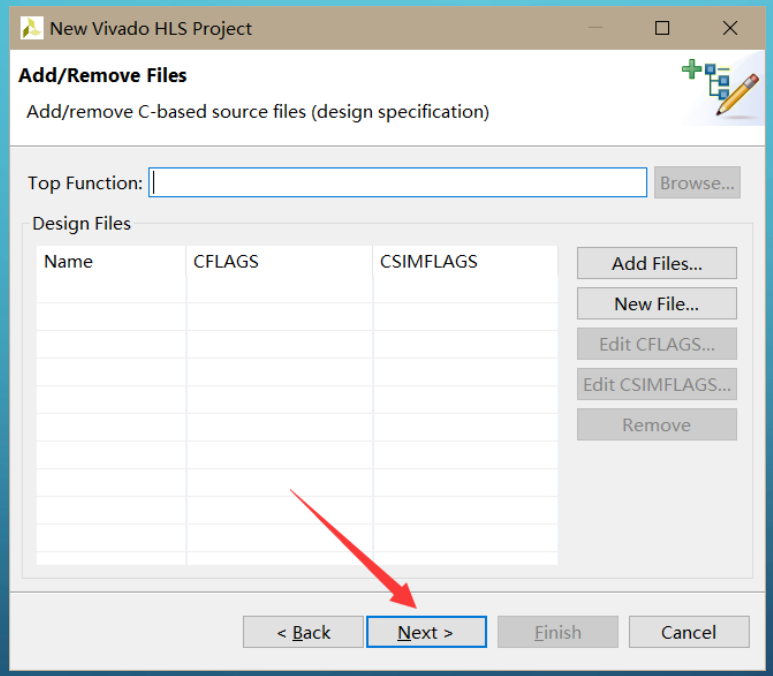
- 2. Create New Project

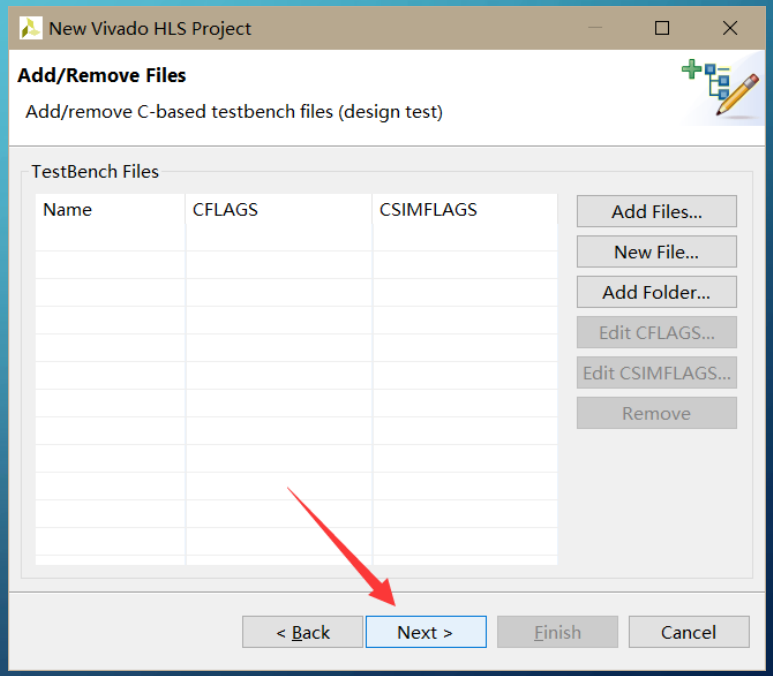


EXAMPLE: MATRIX MULTIPLIER

- 3. Specify the Project name and location (anywhere you like)
- 4. Click “Next” all the way through to “Solution Configuration”

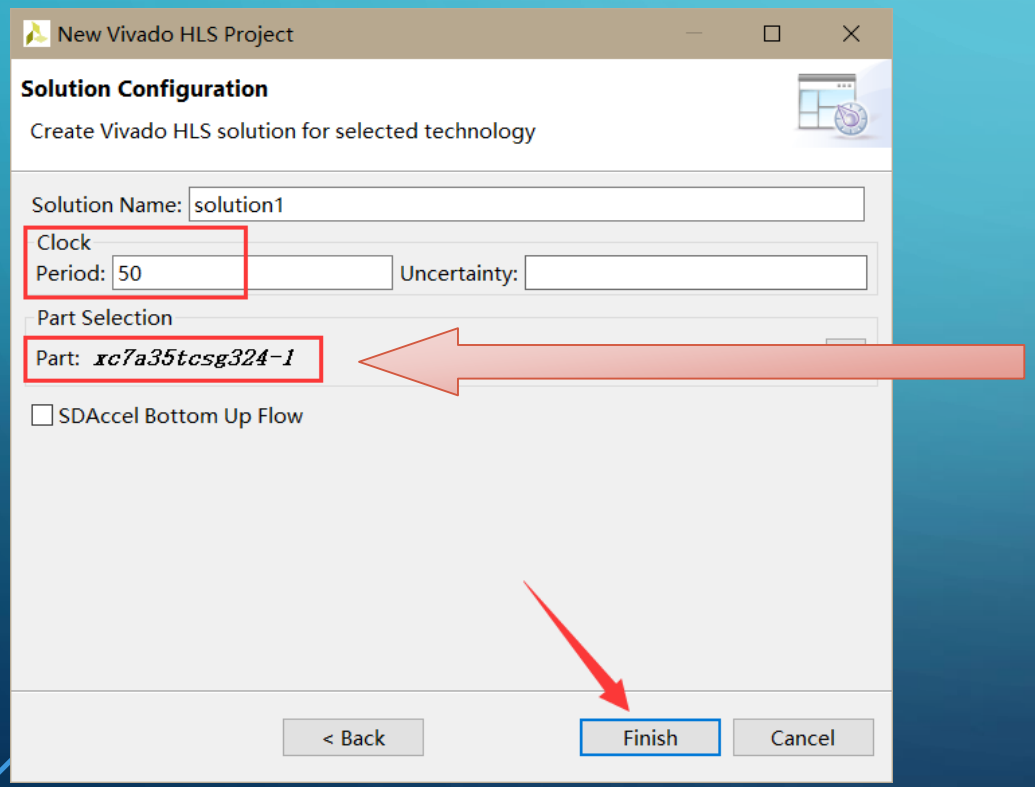


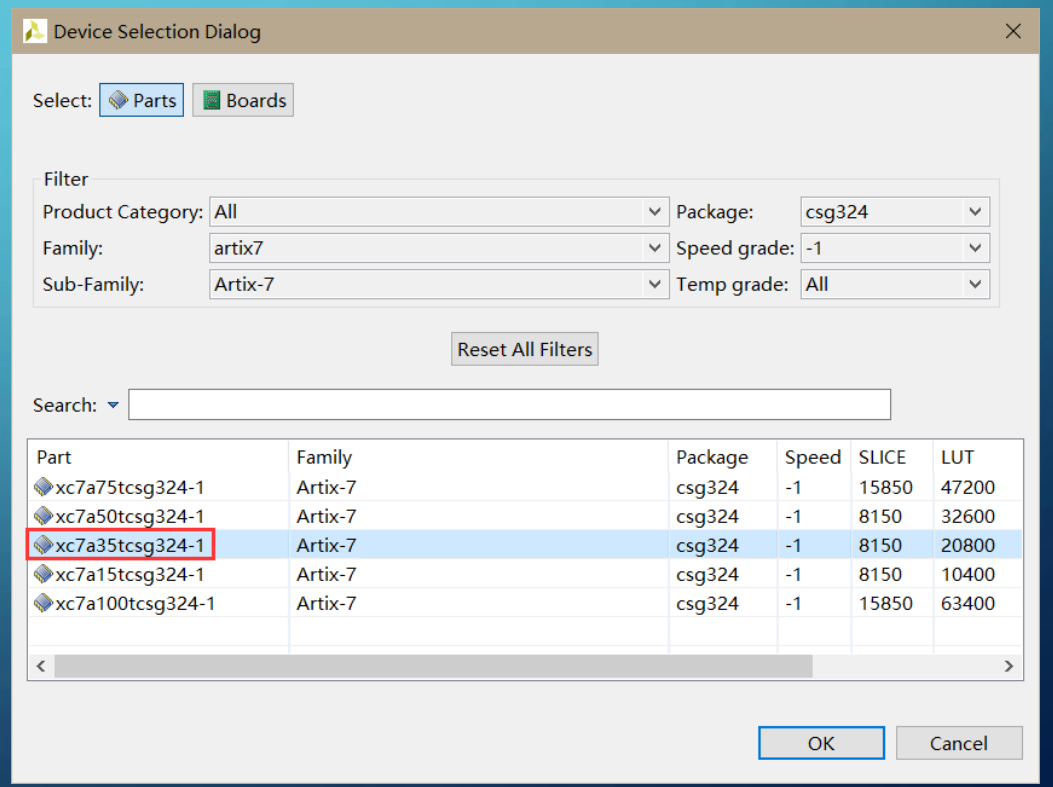




EXAMPLE: MATRIX MULTIPLIER

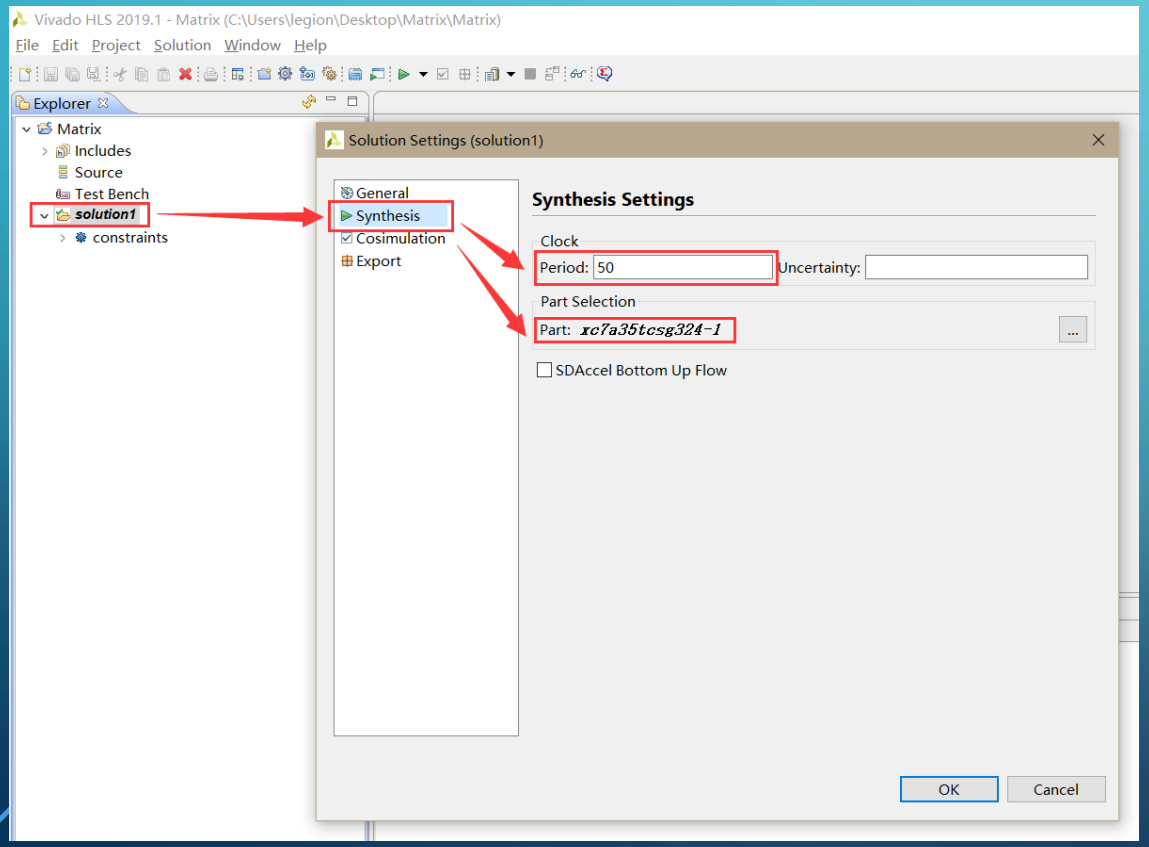
- 5. Assume a Clock Period, which the Compiler will try to synthesize a circuit suitable to run in this timing. We can choose “50 ns” here, and this can be changed later.
- 6. Choose the FPGA part in RGO1 “xc7a35tcsg324-1”
- 7. Click “Finish” to end the new project configuration.





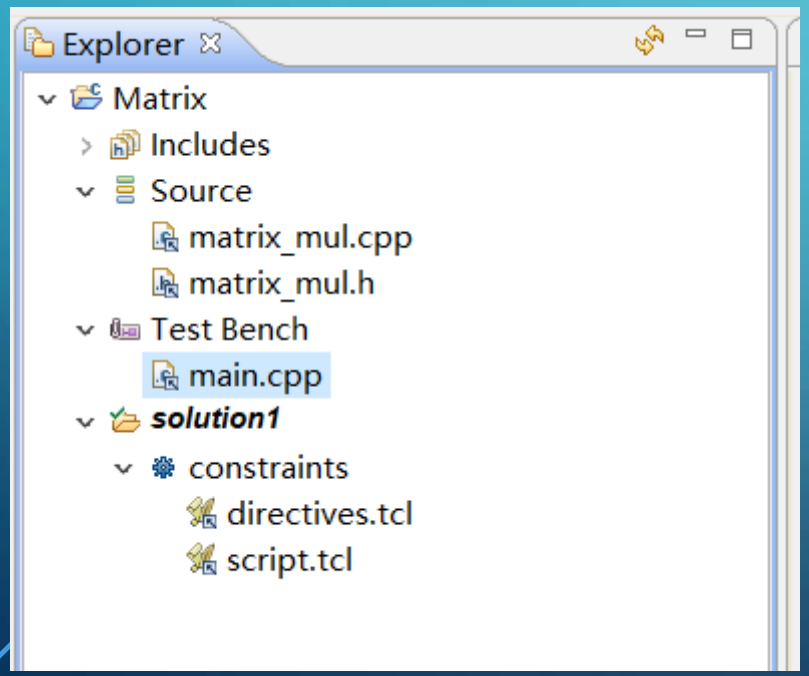
EXAMPLE: MATRIX MULTIPLIER

- 8. The Clock period and part can be configured in Solution Settings.
- Right click “solution1” → “Solution Settings..” → “Sythesis”



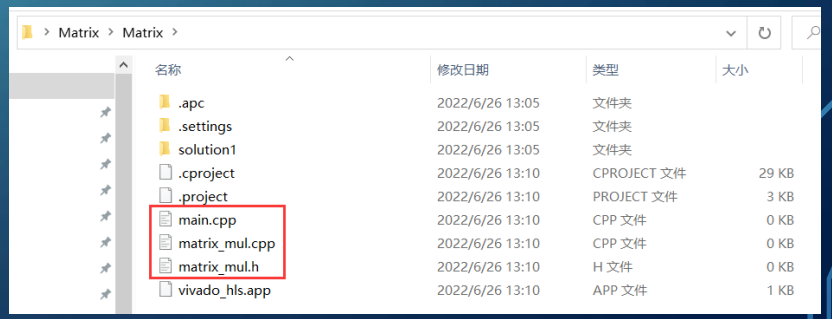
EXAMPLE: MATRIX MULTIPLIER

- 9. Create the c(pp) files we will use.
- Right click “Source” → “New File..” → create “matrix_mul.cpp” and its header.
- Right click “Test Bench” → “New File..” → create “main.cpp”.



Codes to synthesize is placed in “Source” directory, and Testbench is placed in “Test Bench” directory.

Our directives can be written into source code or specified in another files in “constraints” directory, which will be used at compile time.



EXAMPLE: MATRIX MULTIPLIER

- 10. matrix_mul.h
- Add “#ifndef” to avoid error if we “#include” the header many times.
- A new data type “ap_int<>” is used for fixed point numbers with any bit width, so include the header “ap_fixed.h” .
- Type the name and arguments of functions we defined in “matrix_mul.cpp”.

matrix_mul.cpp | matrix_mul.h | main.cpp

```
1 #ifndef __Matrix_Mul__
2 #define __Matrix_Mul__ // Avoid error in multiple including
3
4 #include "ap_fixed.h" // Type in HLS for any precision(bit), ap_int<8> <=> char
5
6 void matrix_mul(ap_int<8> A[4][4], ap_int<8> B[4][4], ap_int<16> C[4][4]);
7
8 #endif
```

EXAMPLE: MATRIX MULTIPLIER

- 10. matrix_mul.cpp (Basic)
- Basic function of a matrix multiplier can be written into two for loops in C++.
- Input A[4][4], B[4][4] and output C[4][4] are all defined as arguments with type ap_int<length>

matrix_mul.cpp | matrix_mul.h | main.cpp

```
1  #include "matrix_mul.h"
2
3  void matrix_mul(ap_int<8> A[4][4], ap_int<8> B[4][4], ap_int<18> C[4][4]){
4      for(int i=0; i<4; i++){
5          for(int j=0; j<4; j++){
6              C[i][j] = 0;
7              for(int k=0; k<4; k++){
8                  C[i][j] = C[i][j] + A[i][k]*B[k][j];
9              }
10         }
11     }
12 }
13
```

EXAMPLE: MATRIX MULTIPLIER

- 10. matrix_mul.cpp (Optimized)
- Constraints can be added for compiling, and can be generated by HLS tools.

matrix_mul.cpp | matrix_mul.h | main.cpp

```
1  #include "matrix_mul.h"
2
3  void matrix_mul(ap_int<8> A[4][4], ap_int<8> B[4][4], ap_int<16> C[4][4]){
4
5      #pragma HLS INTERFACE s_axilite port=return
6      #pragma HLS INTERFACE s_axilite port=A
7      #pragma HLS INTERFACE s_axilite port=B
8      #pragma HLS INTERFACE s_axilite port=C
9      #pragma HLS ARRAY_RESHAPE variable=A complete dim=2
10     #pragma HLS ARRAY_RESHAPE variable=B complete dim=1
11 //  #pragma HLS ARRAY_PARTITION variable=B complete dim=1
12 //  #pragma HLS ARRAY_PARTITION variable=A complete dim=2
13
14
15     for(int i=0; i<4; i++){
16         for(int j=0; j<4; j++){
17
18         #pragma HLS LATENCY min=5 max=5
19         #pragma HLS PIPELINE II=1
20
21             C[i][j] = 0;
22             for(int k=0; k<4; k++){
23                 // #pragma HLS UNROLL
24                 C[i][j] = C[i][j] + A[i][k]*B[k][j];
25             }
26         }
27     }
28 }
29
```

Outline | Directive

- matrix_mul.h
- matrix_mul(ap_int<8>[][], ap_int<8>[][], ap_int<16>[][]) : void

Outline | Directive

- matrix_mul
  - # HLS INTERFACE s_axilite port=return
  - A
  - # HLS ARRAY_RESHAPE variable=A complete dim=2
  - # HLS INTERFACE s_axilite port=A
  - B
  - # HLS ARRAY_RESHAPE variable=B complete dim=1
  - # HLS INTERFACE s_axilite port=B
  - C
  - # HLS INTERFACE s_axilite port=C
  - for Statement
    - for Statement
      - # HLS LATENCY min=5 max=5
      - # HLS PIPELINE II=1
      - for Statement

Click the right part will highlight and jump to the location in source code

EXAMPLE: MATRIX MULTIPLIER

- 11. main.cpp (Testbench)
- The main.cpp is used as a testbench in both “C simulation” and “Co-simulation”, and can be written in a completely “software” style.

matrix_mul.cpp | matrix_mul.h | main.cpp

```
#include "matrix_mul.h"
#include <iostream>

int main(){
    ap_int<8> A[4][4];
    ap_int<8> B[4][4];
    ap_int<18> C[4][4];

    for(int i = 0; i < 4; i++)
        for(int j = 0; j < 4; j++){
            A[i][j] = i*4 + j;
            B[i][j] = A[i][j];
        }

    matrix_mul(A, B, C);

    for(int i = 0; i < 4; i++)
        for(int j = 0; j < 4; j++){
            std::cout << "C[" << i << "," << j << "]=" << C[i][j] << std::endl;
        }

    return 0;
}
```

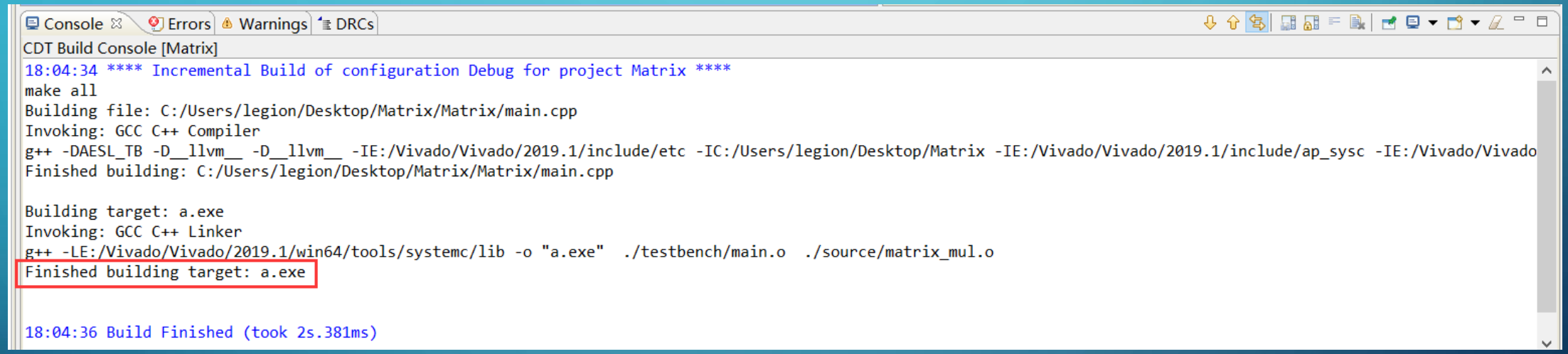
Initialize input matrix A and B

Call the function (initialize a module)

Print each element of C matrix to console

EXAMPLE: MATRIX MULTIPLIER

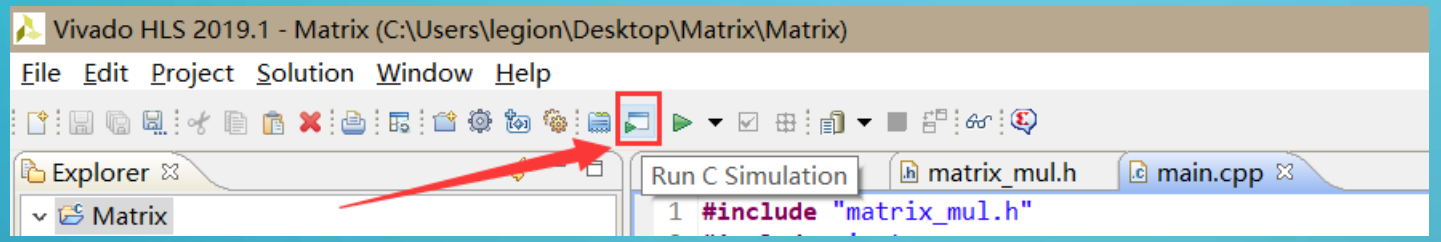
- 12. Build the project
- Press “Ctrl + B” to build the C code, there is no syntax error if a .exe target is built.

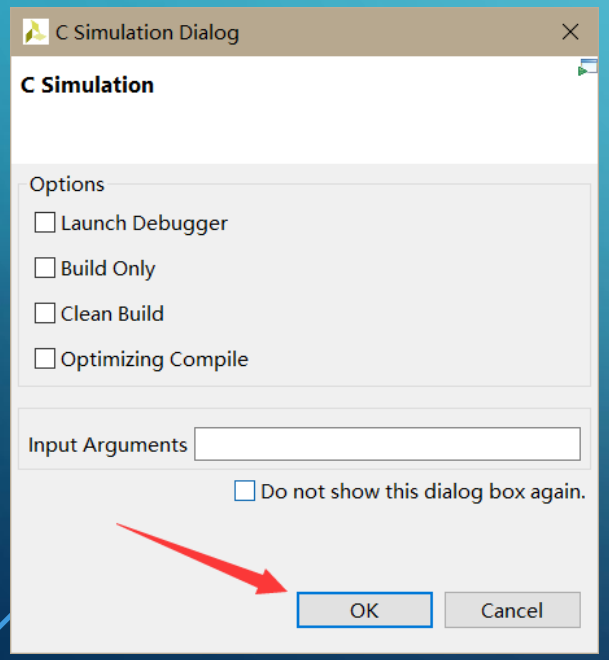


Or we can just run “C simulation” directly, which will also build the project first.

EXAMPLE: MATRIX MULTIPLIER

- 13. C simulation
- Click “Run C simulation” at the tool bar. Then a window will jump out.





Click “OK” if there’s nothing else in the options.

EXAMPLE: MATRIX MULTIPLIER

- 13. C simulation
- C sim only check the C code. And the result in console will be logged in "_csim.log"

matrix_mul.cpp | matrix_mul.h | main.cpp | _csim.log

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../main.cpp in debug mode
   Compiling ../../../matrix_mul.cpp in debug mode
   Generating csim.exe
C[0,0]=56
C[0,1]=62
C[0,2]=68
C[0,3]=74
C[1,0]=152
C[1,1]=174
C[1,2]=196
C[1,3]=218
C[2,0]=248
C[2,1]=286
C[2,2]=324
C[2,3]=362
C[3,0]=344
C[3,1]=398
C[3,2]=452
C[3,3]=506
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************

```

The page of log file will open automatically after the simulation.

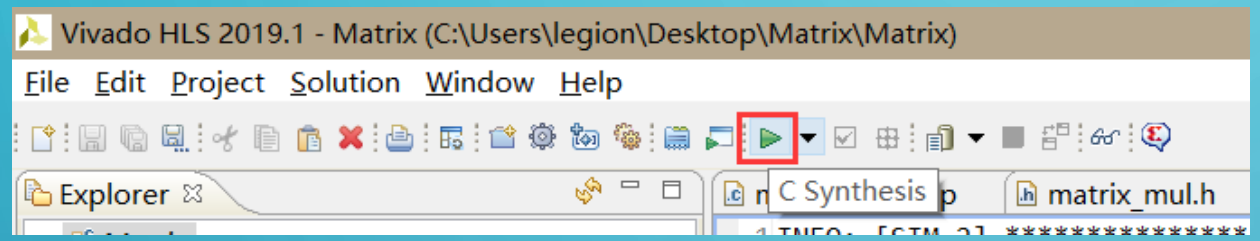
We can check the printed results.

0 errors in the Csim.

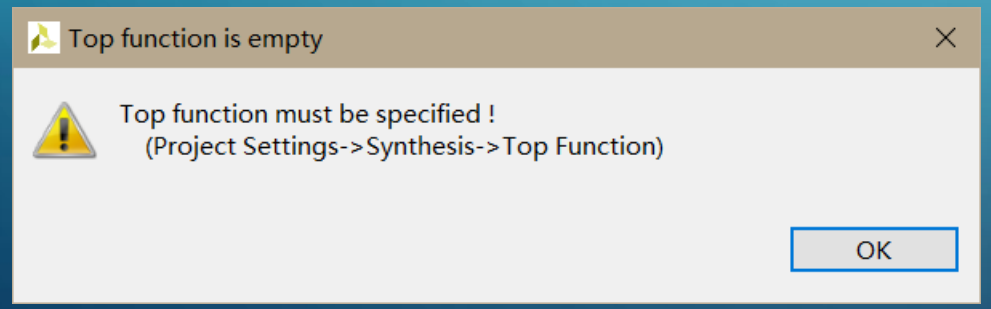
Information and warnings during the simulation can be find in the console. Go check the INFO if something goes wrong.

EXAMPLE: MATRIX MULTIPLIER

- 13. C Synthesis
- Then we can click this to run C synthesis.

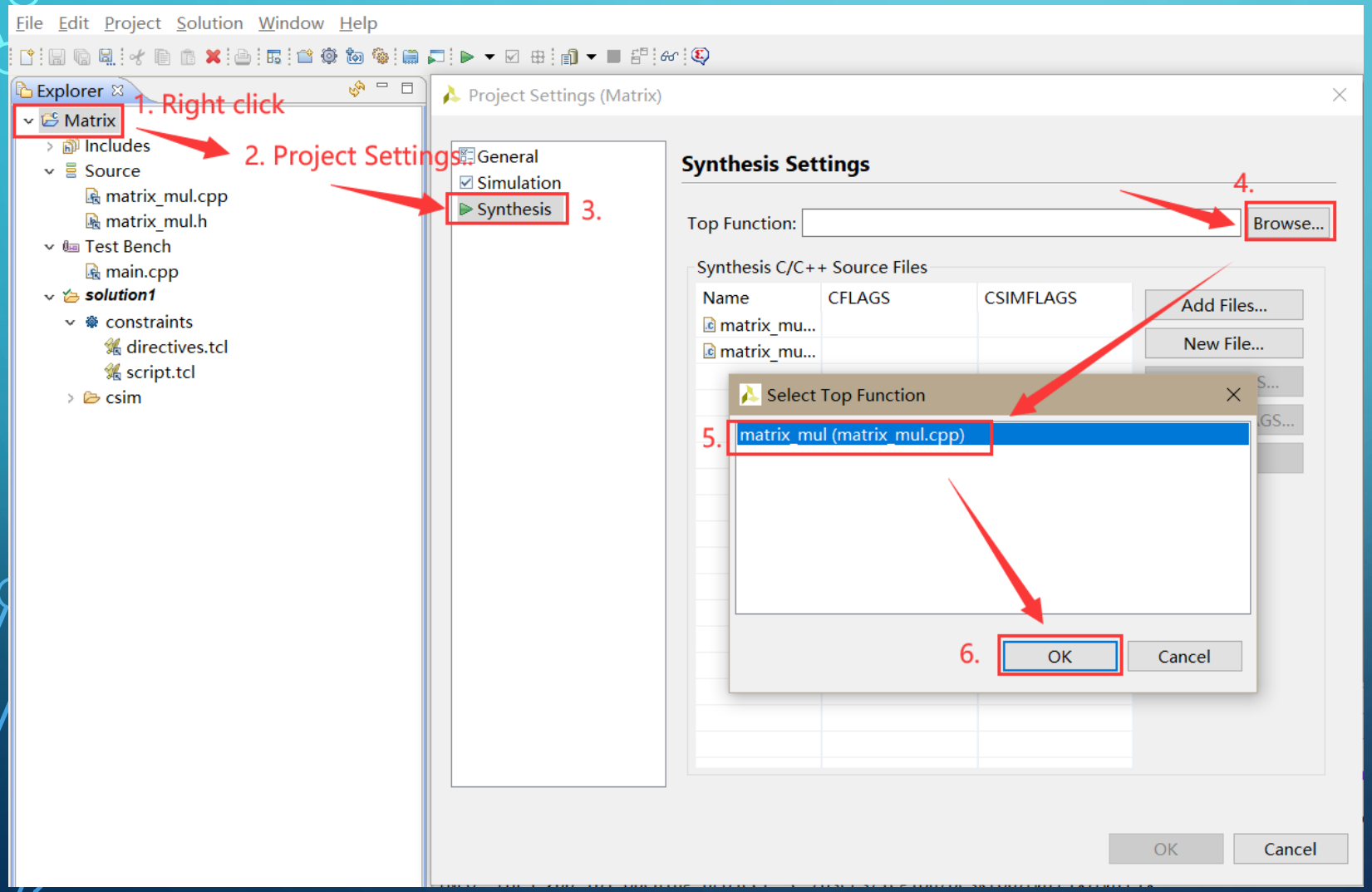


Oops, the top module hasn't been specified now, as there may be many functions in the source file matrix_mul.cpp, we should set the hierarchy for the synthesizer.



EXAMPLE: MATRIX MULTIPLIER

- 13. C Synthesis

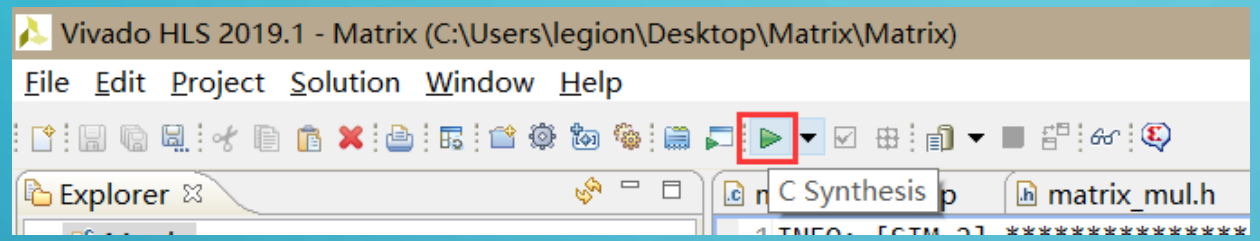


Specify the top function in the following steps.

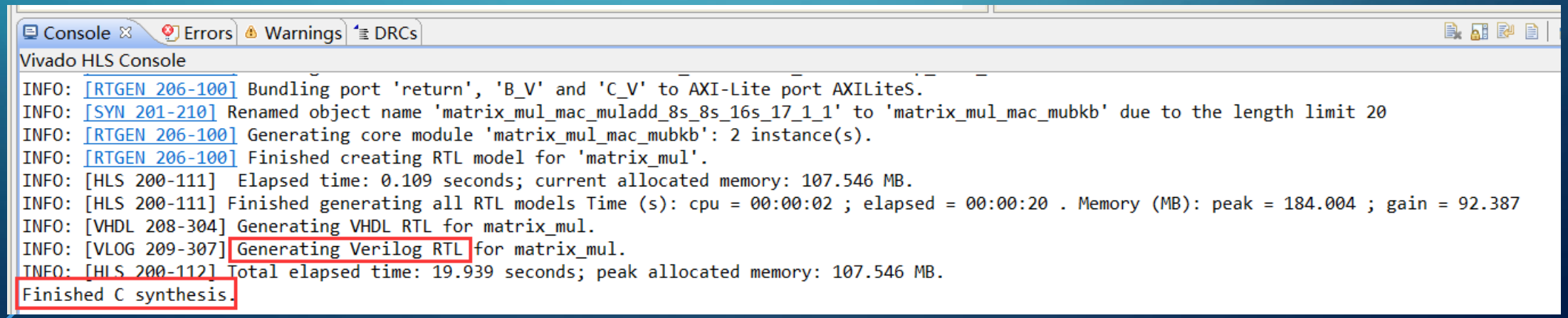
Then click “OK”

EXAMPLE: MATRIX MULTIPLIER

- 13. C Synthesis
- Then we can click again to run C synthesis.



Wait for a while to run the C synthesis.
The RTL code will also be generated, check it to see how complex it will be.



EXAMPLE: MATRIX MULTIPLIER

- 13. C Synthesis
- Check the synthesis report.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	50.00	9.332	6.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
169	169	169	169	none

Detail

Instance

N/A

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	168	168	42	-	-	4	no
+ Loop 1.1	40	40	10	-	-	4	no
++ Loop 1.1.1	8	8	2	-	-	4	no

The estimated timing now is only about 9 ns, so it is possible to set the ap_clk to 10 ns, result in 5x throughput for the multiplier.

The latency is 169 clock cycles, the detailed latency is shown below in the Loop table.

EXAMPLE: MATRIX MULTIPLIER

- 13. C Synthesis

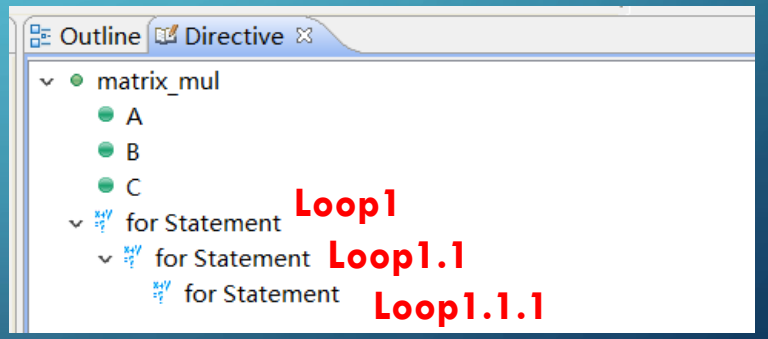
Summary

Latency		Interval		
min	max	min	max	Type
169	169	169	169	none

The **latency** means we need 169 clock cycles to calculate matrix multiplication once;
Interval means that 169 cycles are needed between two multiplications.
The latency and interval are equal if there is no pipeline.

Loop

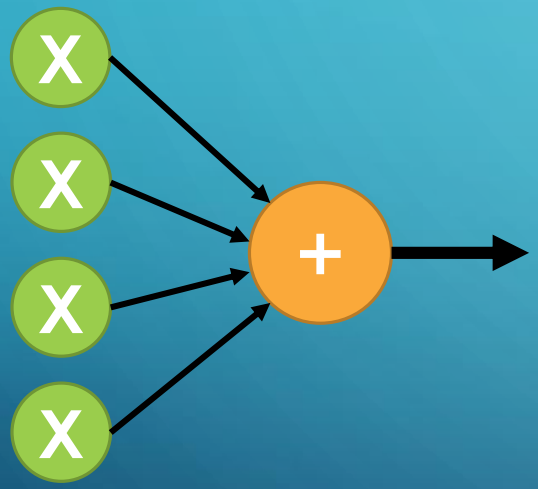
	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	168	168	42	-	-	4	no
+ Loop 1.1	40	40	10	-	-	4	no
++ Loop 1.1.1	8	8	2	-	-	4	no



Loop1.1.1: iteration latency: need 2 cycles in each loop iteration. Repeat for 4 times, so the total latency is 8.
Loop1.1: iteration latency: 8-cycles in Loop1.1.1 + 2-cycles of state change.
Loop1: iteration latency: 40-cycles in Loop1.1 + 2-cycles of state change.
Loop1: total latency: 4*42-cycles

EXAMPLE: MATRIX MULTIPLIER

- 14. Optimization
- If we want to calculate the last loop A[i]*B[i] at the same time, we will need only 16 cycles to calculate the whole matrix.



Method 1: Unroll the Loop1.1.1, which means unroll the “for loop” into 1 iteration, the compiler will generate 4 multiplier and 1 adder.

```
for(int k=0; k<4; k++){
    #pragma HLS UNROLL
    C[i][j] = C[i][j] + A[i][k]*B[k][j];
```

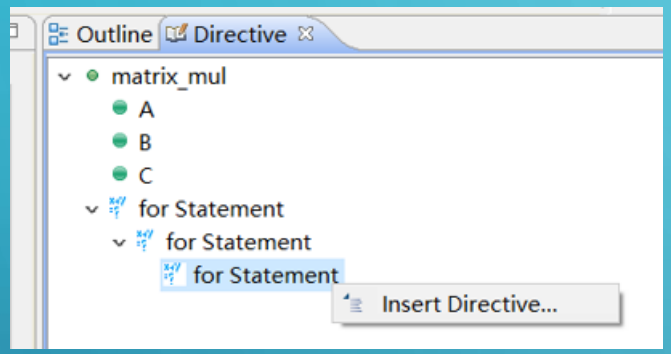
Method 2: Tell the tool that this loop should use only 1 cycle to calculate. This means adding a constraint for the compiler. Doesn’t matter whether it can be achieved or not, it will report a warning if the constraint cannot be realized by the synthesizer.

```
#pragma HLS PIPELINE II=1
    C[i][j] = 0;
    for(int k=0; k<4; k++){
        #pragma HLS UNROLL
        C[i][j] = C[i][j] + A[i][k]*B[k][j];
    }
```

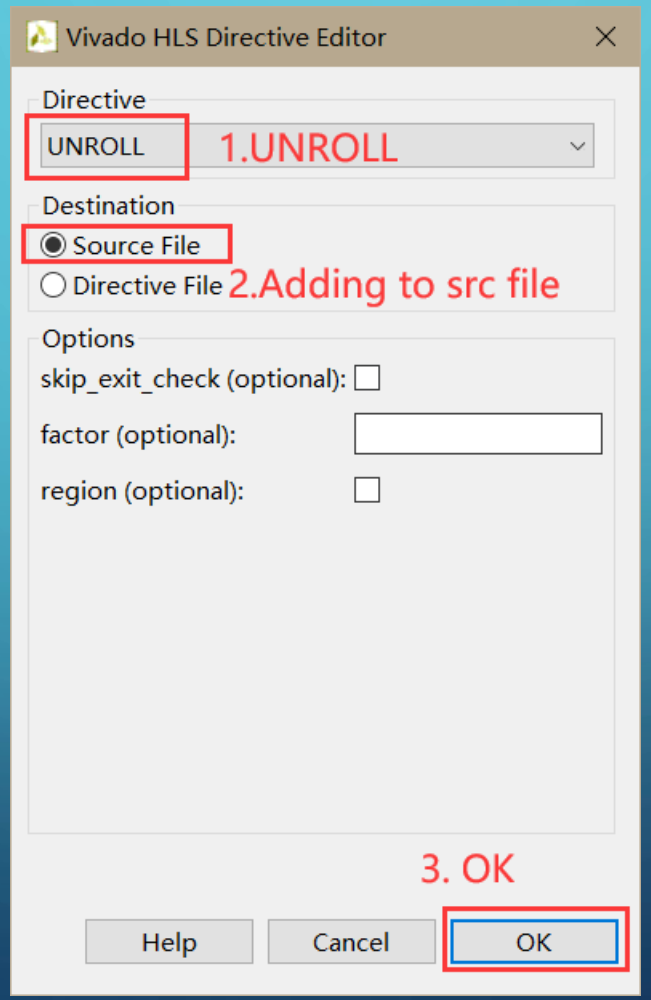
EXAMPLE: MATRIX MULTIPLIER

- 14. Optimization: UNROLL

Method 1: Loop Unroll



Go to the directives at the top-right corner, choose the inner for loop and “Insert Directives”




```
for(int k=0; k<4; k++){
    #pragma HLS UNROLL
    C[i][j] = C[i][j] + A[i][k]*B[k][j];
```

```
22              for(int k=0; k<4; k++){
23  #pragma HLS UNROLL
24                  C[i][j] = C[i][j] + A[i][k]*B[k][j];
25              }
26          }
```

Edit the option box as we need, and a “#pragma” statement will be generated to the destination we specified.

You can just write the statement without using the tool.

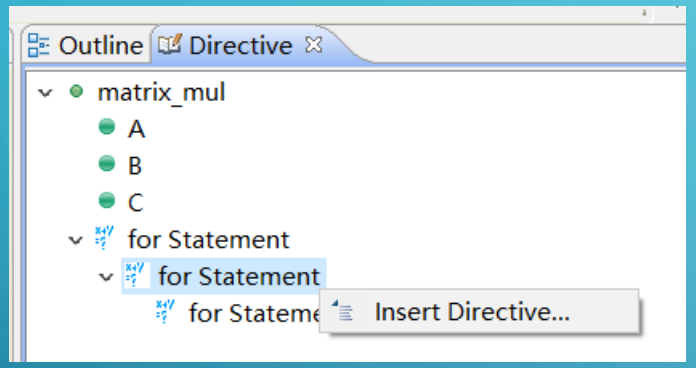
EXAMPLE: MATRIX MULTIPLIER

- 14. Optimization: PIPELINE

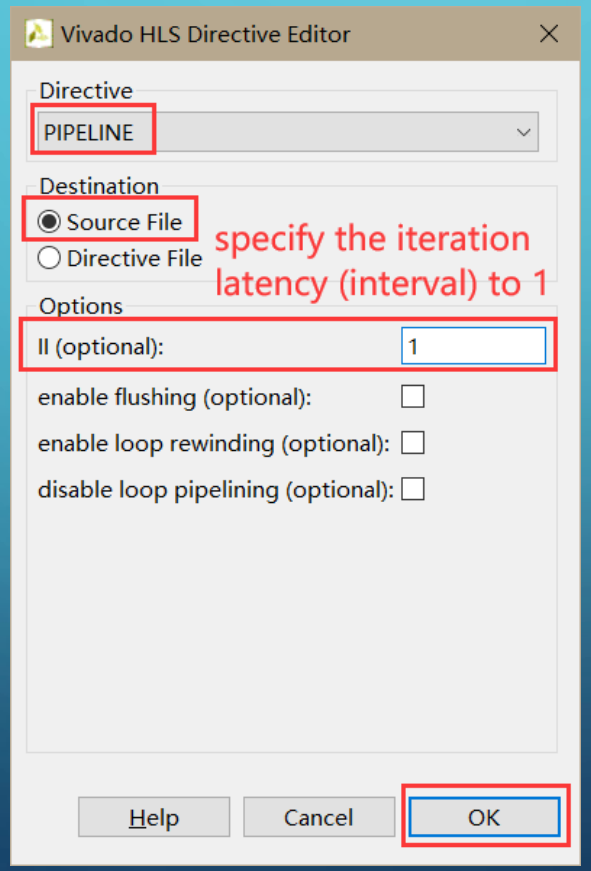
```
#pragma HLS PIPELINE II=1
    C[i][j] = 0;
    for(int k=0; k<4; k++){
        #pragma HLS UNROLL
        C[i][j] = C[i][j] + A[i][k]*B[k][j];
    }
```

Method 2: Pipeline: Iteration Latency

A more strict constraint is to specify the iteration interval (II) in the Loop1.1



Choose the second loop Loop1.1 “Insert Directives”




```
15      for(int i=0; i<4; i++){
16          for(int j=0; j<4; j++){
17  #pragma HLS PIPELINE II=1
18
19              C[i][j] = 0;
20              for(int k=0; k<4; k++){
21  //              #pragma HLS UNROLL
22                  C[i][j] = C[i][j] + A[i][k]*B[k][j];
23              }
24          }
25      }
26  }
```

Specify the iteration latency for the whole Loop1.1 to pipeline, then a “#pragma” statement will be generated to the destination we specified.

EXAMPLE: MATRIX MULTIPLIER

- 15. C Synthesis
- Then we can repeat the above steps to see the report.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	50.00	14.675	6.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
34	34	34	34	none

Detail

Instance

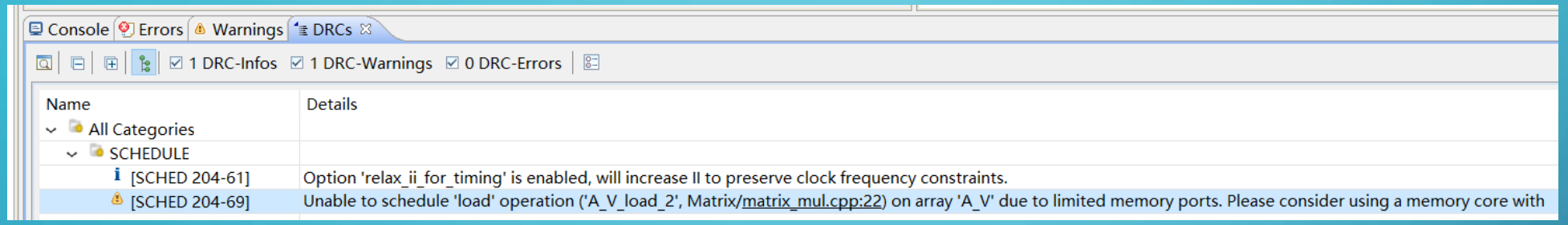
Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	32	32	3	2	1	16	yes

The latency seems decreased, but still not achieved the 16 cycles we want, and a warning is reported.

EXAMPLE: MATRIX MULTIPLIER

- 15. C Synthesis



Interface

Summary

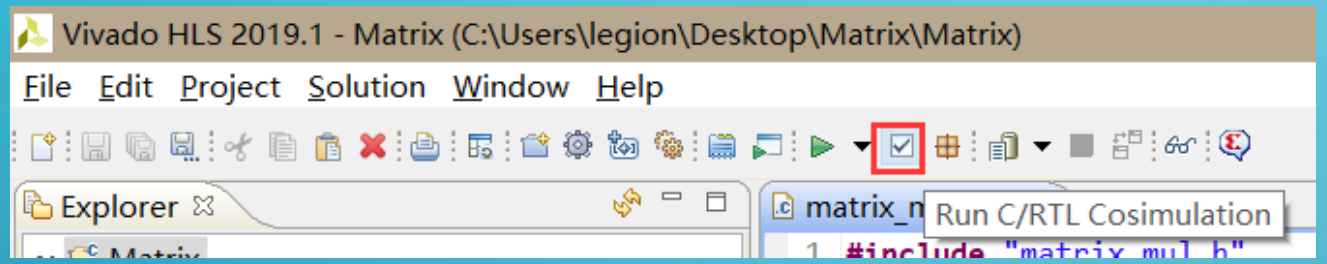
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	matrix_mul	return value
ap_rst	in	1	ap_ctrl_hs	matrix_mul	return value
ap_start	in	1	ap_ctrl_hs	matrix_mul	return value
ap_done	out	1	ap_ctrl_hs	matrix_mul	return value
ap_idle	out	1	ap_ctrl_hs	matrix_mul	return value
ap_ready	out	1	ap_ctrl_hs	matrix_mul	return value
A_V_address0	out	4	ap_memory	A_V	array
A_V_ce0	out	1	ap_memory	A_V	array
A_V_q0	in	8	ap_memory	A_V	array
A_V_address1	out	4	ap_memory	A_V	array
A_V_ce1	out	1	ap_memory	A_V	array
A_V_q1	in	8	ap_memory	A_V	array
B_V_address0	out	4	ap_memory	B_V	array
B_V_ce0	out	1	ap_memory	B_V	array
B_V_q0	in	8	ap_memory	B_V	array
B_V_address1	out	4	ap_memory	B_V	array
B_V_ce1	out	1	ap_memory	B_V	array
B_V_q1	in	8	ap_memory	B_V	array
C_V_address0	out	4	ap_memory	C_V	array
C_V_ce0	out	1	ap_memory	C_V	array
C_V_we0	out	1	ap_memory	C_V	array
C_V_d0	out	18	ap_memory	C_V	array

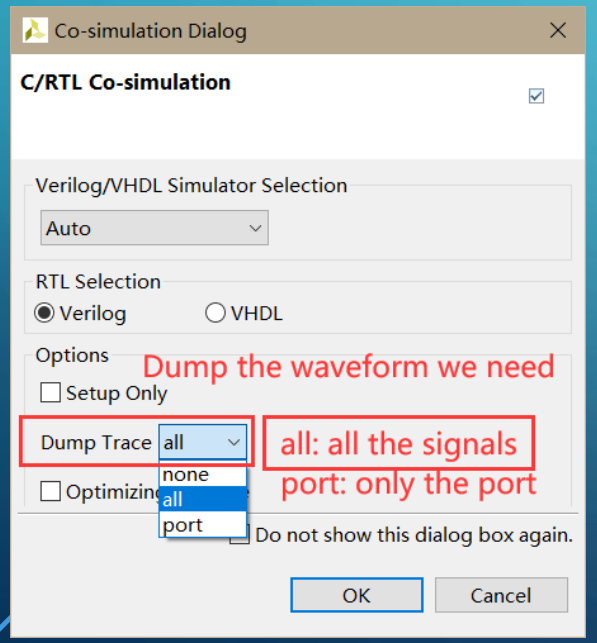
See the “Interface” part in the report.
A and B are synthesized as 2 two-port RAM, and we can only read out 2 data in a cycle, so there still need 2 cycles in each iteration, result in 34 cycles in total.

Not very clear? We can check out the waveform!

EXAMPLE: MATRIX MULTIPLIER

- 16. Co-Simulation
- We can do the C/RTL co-simulation to see the wave

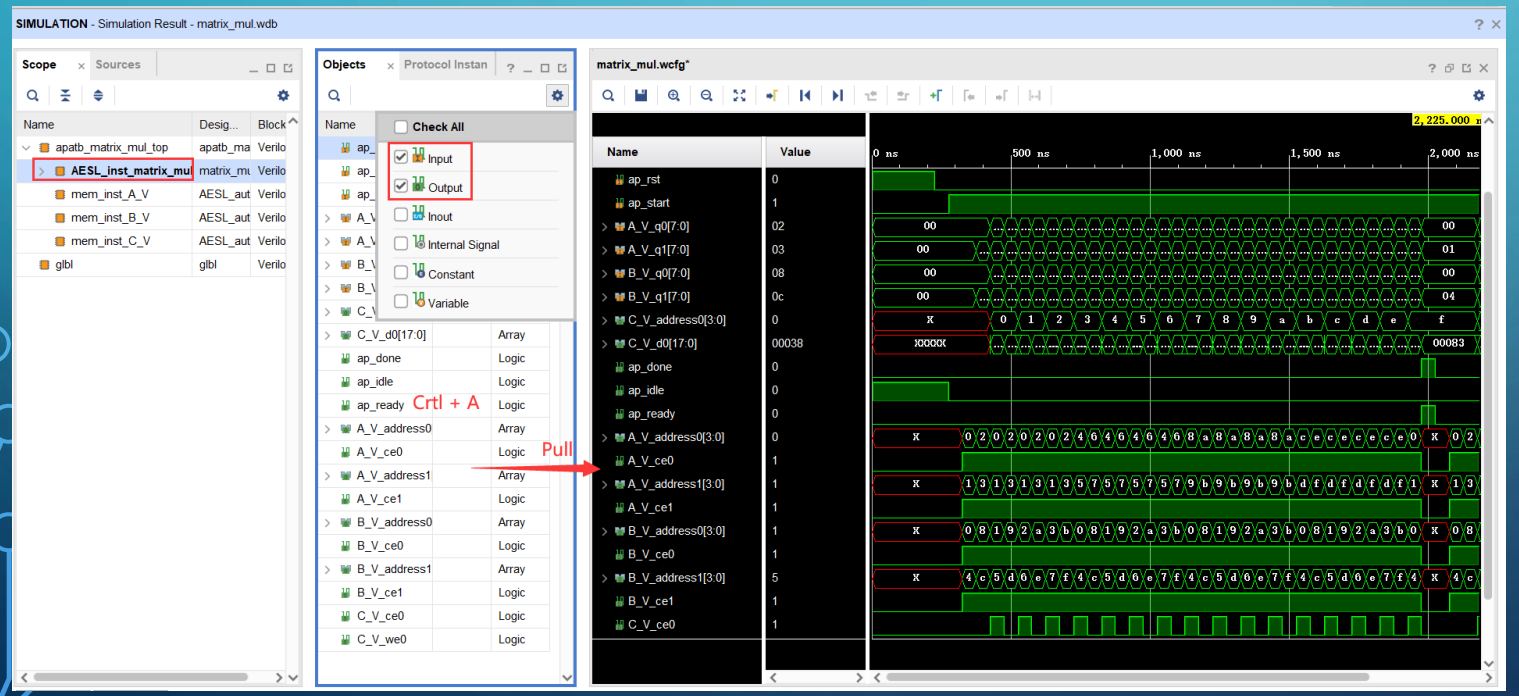




Dump (save) the wave we need to debug.
Choose all for now to check all the waveforms.
It takes time to do the cosimulation.

EXAMPLE: MATRIX MULTIPLIER

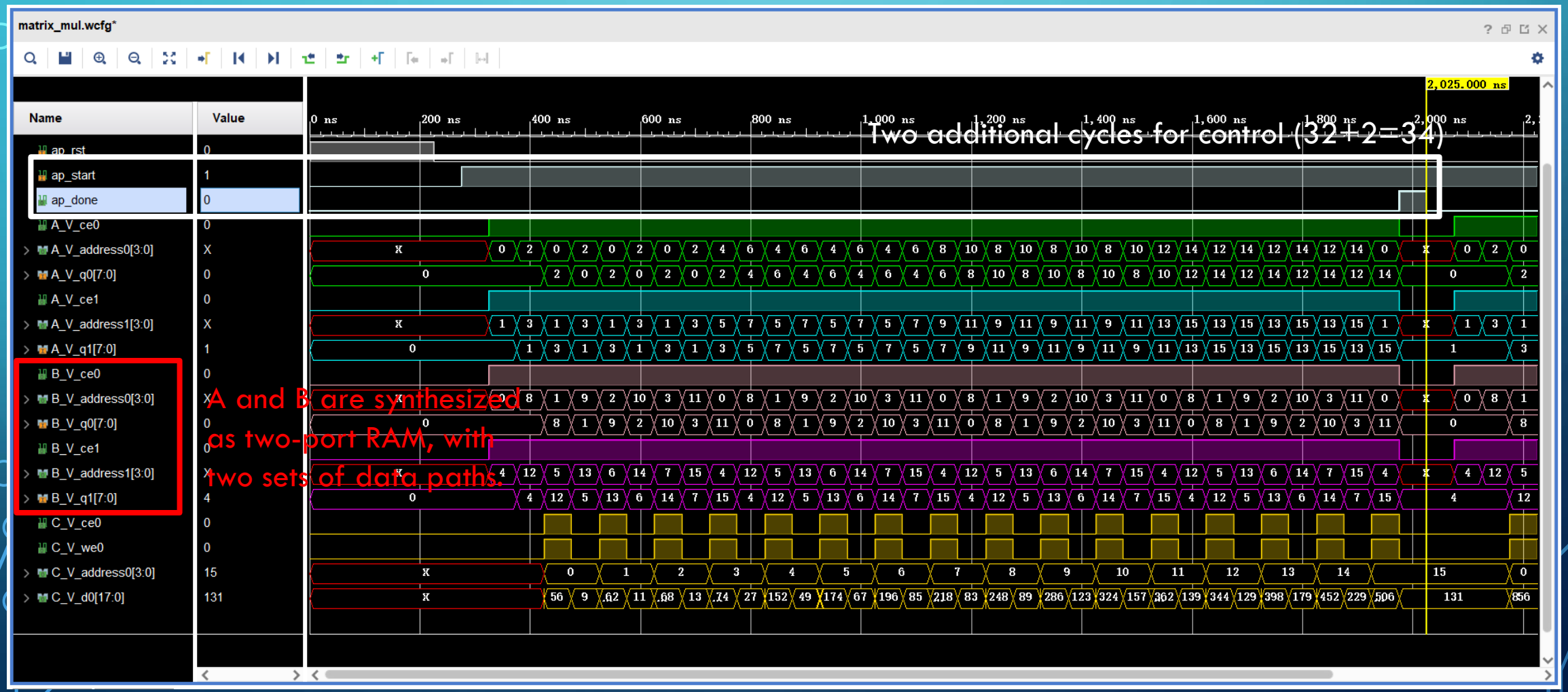
- 16. Co-Simulation
- When cosim is finished, open Wave Viewer, the Vivado will be opened.



Dump (save) the wave we need to debug.
Choose all for now to check all the waveforms.
It takes time to do the cosimulation.

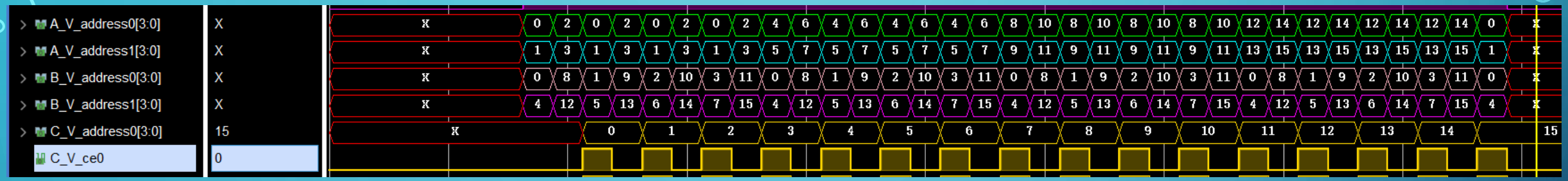
EXAMPLE: MATRIX MULTIPLIER

- 16. Co-Simulation



EXAMPLE: MATRIX MULTIPLIER

- 16. Co-Simulation



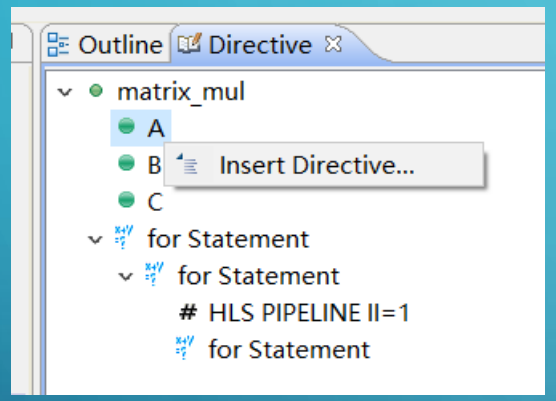
Analyze the addresses of A, B and C, we can find that it takes 2 cycles to load out all 4 data in each iteration, thus the C_V_ce0 write once every two cycles.

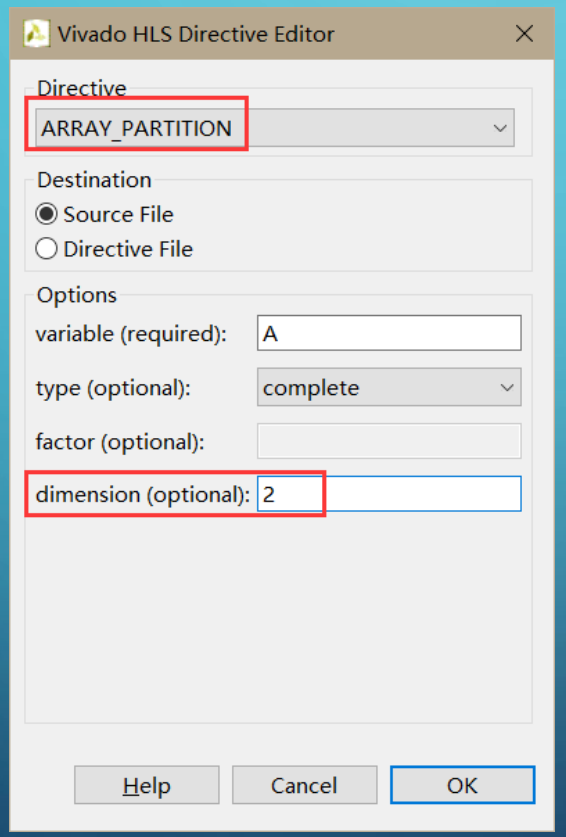
How to realize our goal (1 cycle per iteration)?

Array Reshaping and Partition!

EXAMPLE: MATRIX MULTIPLIER

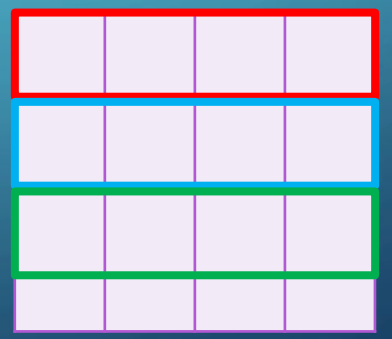
- 17. Optimization: Array Partition
- Go back to the source code, if we cut the 2D array into 4 parts, we can read out 4 data per cycle!

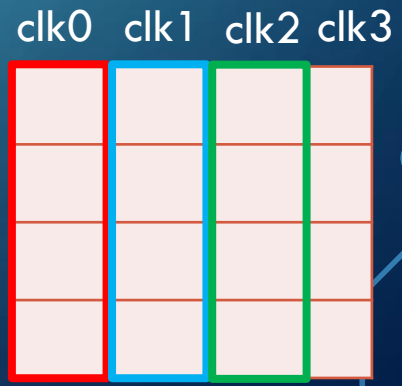





```
#pragma HLS ARRAY_PARTITION variable=A complete dim=2
#pragma HLS ARRAY_PARTITION variable=B complete dim=1
```

Then cut B as well, but in dim=1.
Just copy the A statement.





EXAMPLE: MATRIX MULTIPLIER

- 17. Optimization: Array Partition
- Now the total latency of 18 cycles is achieved.
- Check it in the Synthesis report and wave.
- Remember to close Vivado before co-sim!

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	50.00	14.675	6.25

Latency (clock cycles)

Summary

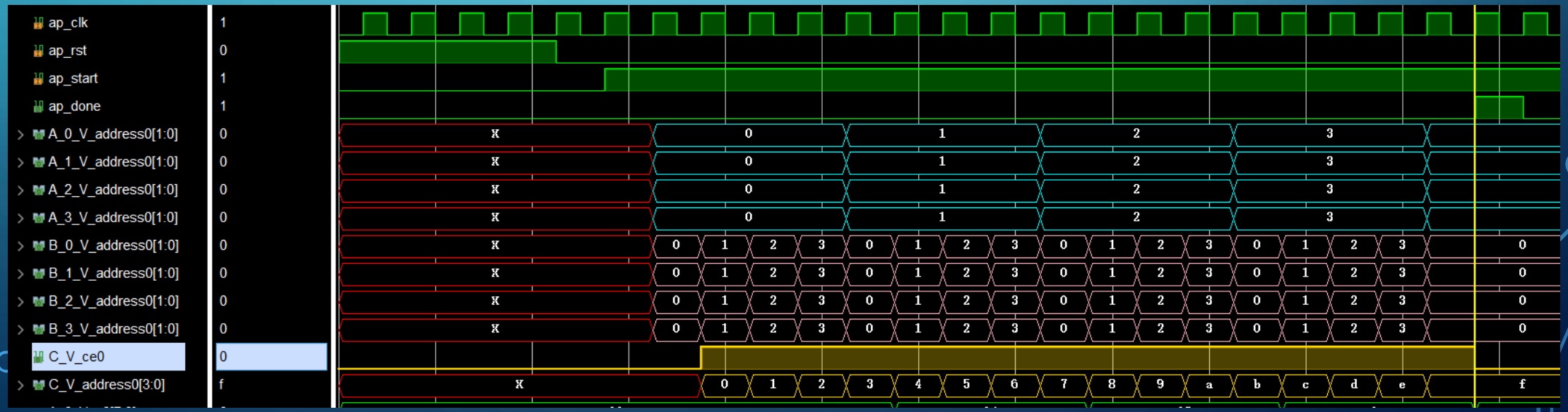
Latency		Interval		
min	max	min	max	Type
18	18	18	18	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	16	16	2	1	1	16	yes



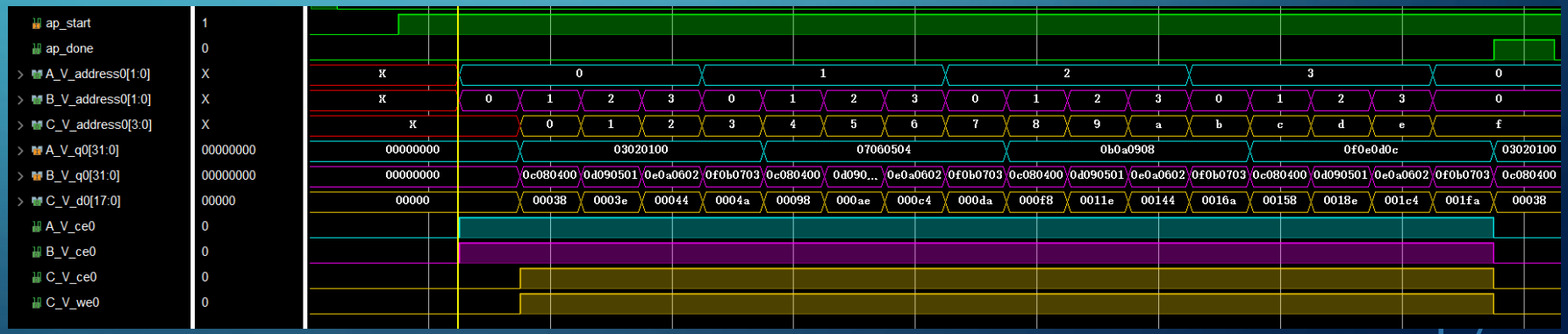
EXAMPLE: MATRIX MULTIPLIER

- 18. PRACTICE: Array Reshape
- Try the **Array reshaping** Constraint
- Check the Synthesis Report and Wave Viwer, can you describe the difference between Partition and Reshaping?
- Hint: Check the data length in “Interface”, and waveform.

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	matrix_mul	return value
ap_rst	in	1	ap_ctrl_hs	matrix_mul	return value
ap_start	in	1	ap_ctrl_hs	matrix_mul	return value
ap_done	out	1	ap_ctrl_hs	matrix_mul	return value
ap_idle	out	1	ap_ctrl_hs	matrix_mul	return value
ap_ready	out	1	ap_ctrl_hs	matrix_mul	return value
A_V_address0	out	2	ap_memory	A_V	array
A_V_ce0	out	1	ap_memory	A_V	array
A_V_q0	in	32	ap_memory	A_V	array
B_V_address0	out	2	ap_memory	B_V	array
B_V_ce0	out	1	ap_memory	B_V	array
B_V_q0	in	32	ap_memory	B_V	array
C_V_address0	out	4	ap_memory	C_V	array
C_V_ce0	out	1	ap_memory	C_V	array
C_V_we0	out	1	ap_memory	C_V	array
C_V_d0	out	18	ap_memory	C_V	array



SUMMARY

- Through this example, we not only generated Verilog (RTL) code to realize a complex module with io interfaces, but also optimized the circuit from 169 clock cycles to only 18 cycles using optimization constraints.
- This is how fast and functional the HLS tool is.
- With the same working flow, we can write much more complex modules with only C code and a little hardware basic knowledge.
- This is why HLS is often known as a popular tool for IC beginners and a powerful productivity tool for skilled IC designers.