香港中文大學
The Chinese University of Hong Kong

CENG3430 Rapid Prototyping of Digital Systems

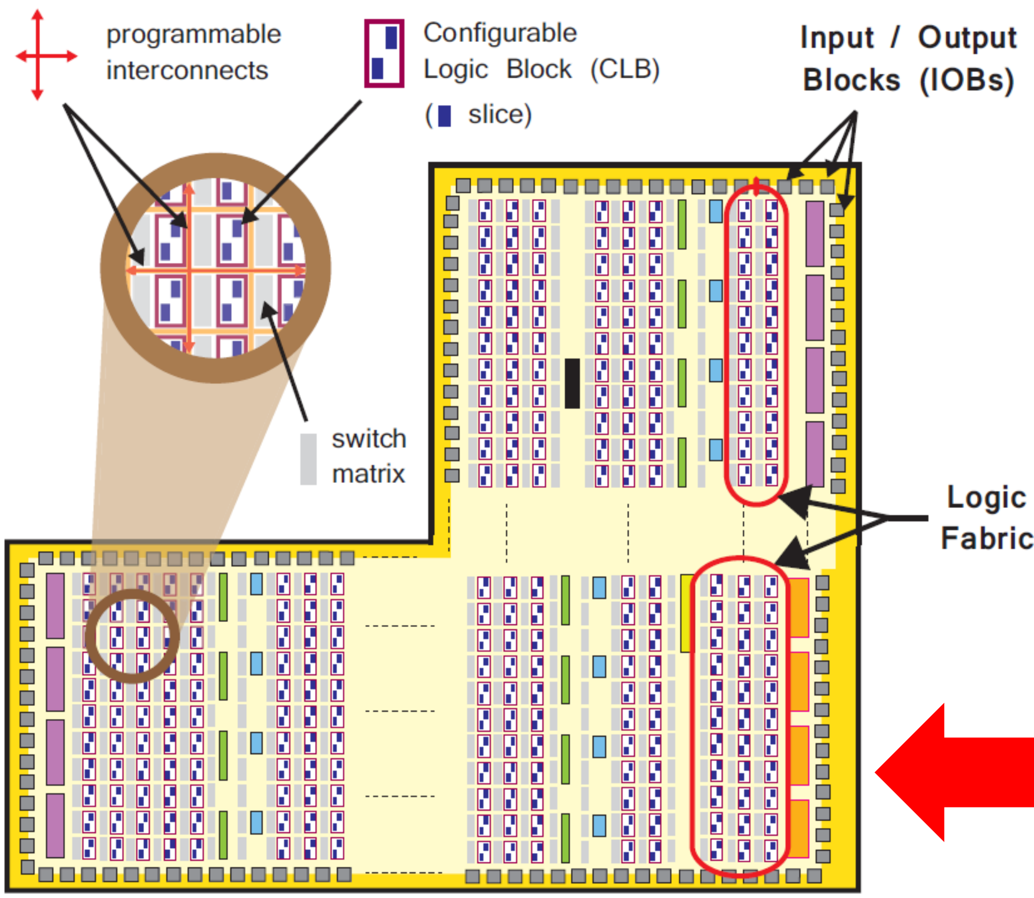
Lecture 09:
Rapid Prototyping (III) – High Level Synthesis

Ming-Chang YANG

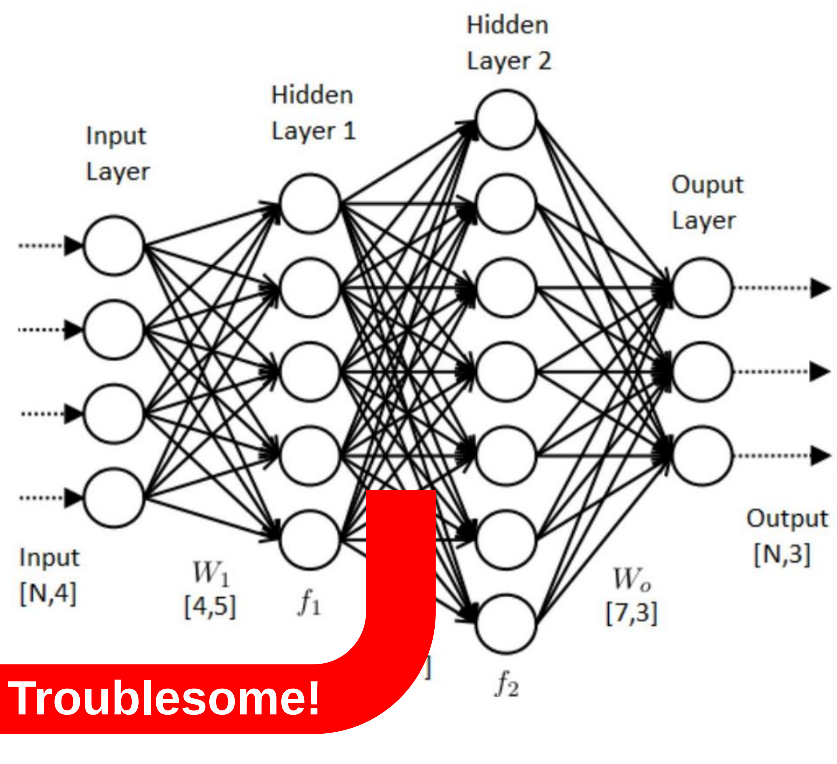
mcyang@cse.cuhk.edu.hk

High-Performance Logic in PL?

- **Programmable Logic (PL)** is ideal for high-speed and high-parallel logic and arithmetic.
  - – However, it might be very hard to implement sometimes.



Ex: Neural Network



Outline

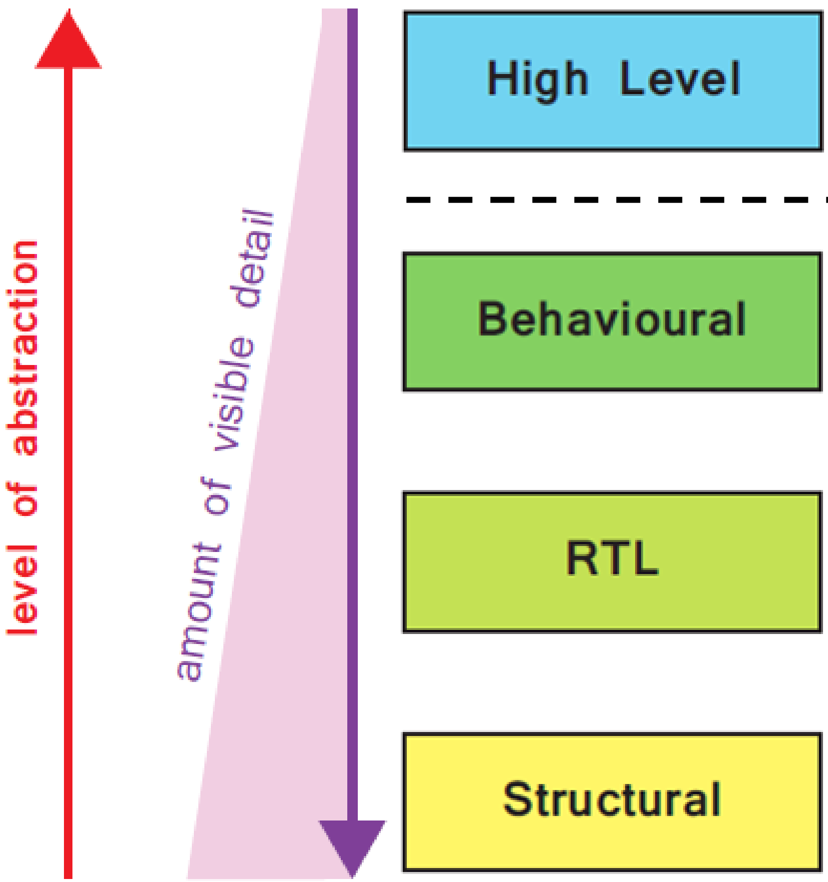
- High-Level Synthesis Concept

- Vivado High-Level Synthesis
  - Inputs and Outputs
  - High-Level Synthesis Process
    - Interface Synthesis
    - Algorithm Synthesis
  - Algorithm Optimizations
    - Loop
    - Array

- Lab Exercise: Accelerating Floating Point Matrix Multiplication with HLS

High-Level Synthesis (HLS)

- **High-level synthesis (HLS)** simplifies the circuit description by abstracting/hiding low-level details with high-level (i.e., algorithmic-level) representations.



*The **high level** expresses designs at an algorithmic level of abstraction.*

C/C++/SystemC design entry

*The **behavioral HDL** describes how the circuit "behaves" (what we use!)*

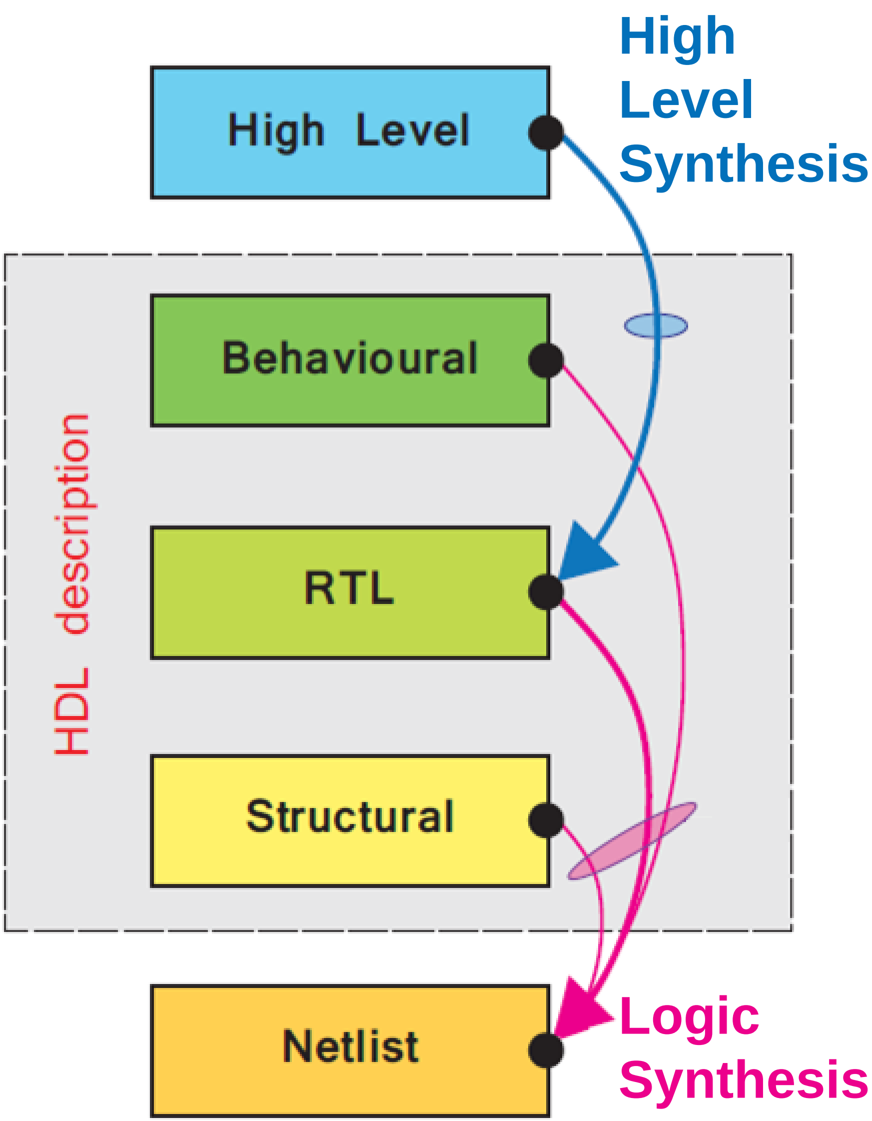
*The **register transfer level (RTL)** interprets operations occurring between registers.*

*The **structural level** involves instantiating, configuring and connecting hardware elements down to the levels of LUTs and FFs.*

HDL design entry

HLS vs. Logic Synthesis

- **High-level synthesis** means synthesizing the **high-level code** into an **HDL description**.
- In FPGA design, the term "synthesis" usually refers to **logic synthesis**.
  - The process of interpreting **HDL code** into the **netlist**.
- In the HLS design flow, both types of "synthesis" are applied (**one** after **the other**)!



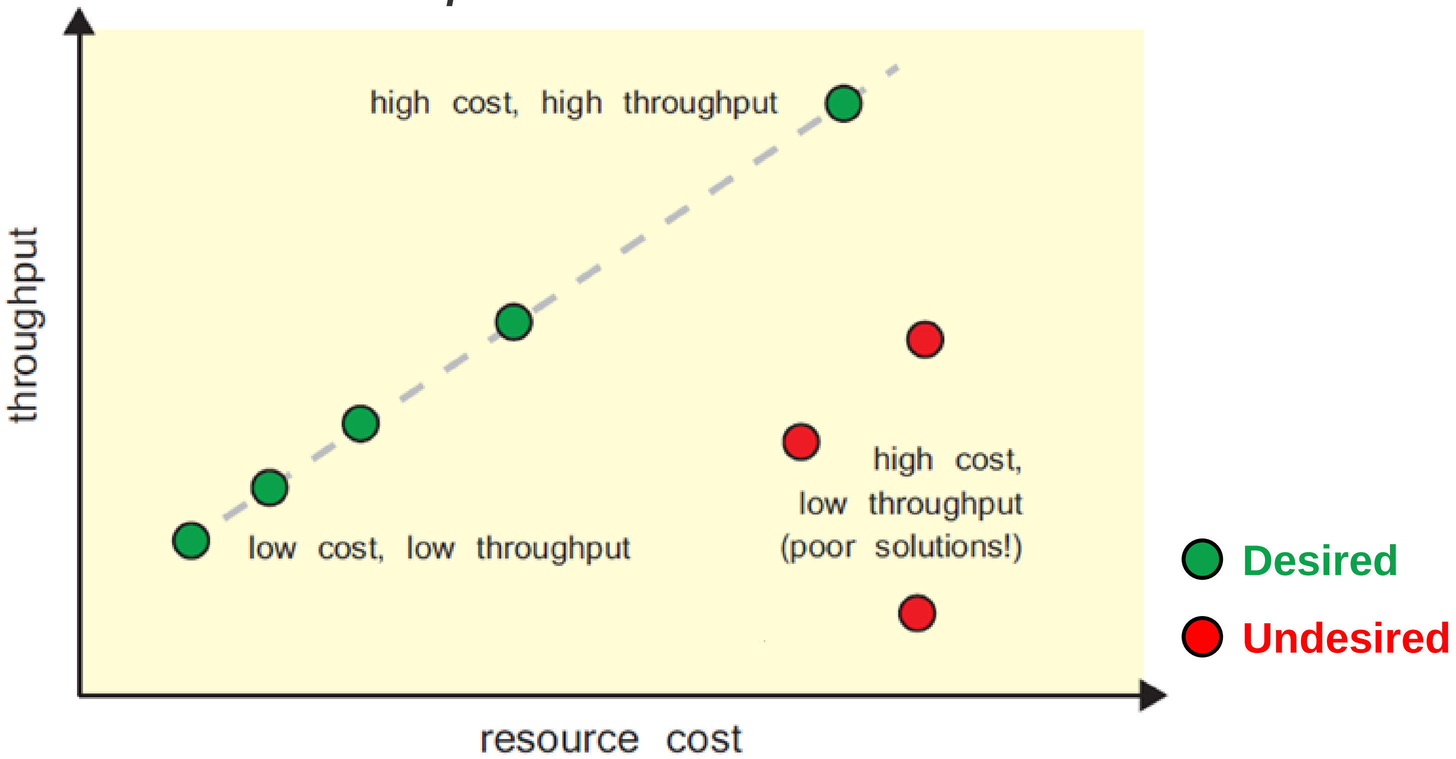
Why High-Level Synthesis (HLS)?

1) HLS from high-level languages is convenient.
   – Engineers are comfortable with languages such as C/C++.
2) The designers simply direct the process, while the HLS tools (i.e., Vivado HLS) implement the details.
   – Designs can be generated rapidly; but the designer must trust the HLS tools in implementing lower-level functionality.
3) HLS separates the functionality and implementation.
   – The source code *does not fix* the actual implementation.
   – *Variations* on the implementations can be created quickly by applying appropriate **"directives"** to the HLS process.
      • But there is no need to explicitly change to the source code.

In one word: HLS shoots for **productivity**.

Design Metrics in HLS

- Hardware design always faces a trade-off between:
  1) **Area**, or **Resource Cost** — *the amount of hardware required to realize the desired functionality*;
  2) **Speed** (specifically **throughput** or **latency**) — *the rate at which the circuit can process data*.



Outline

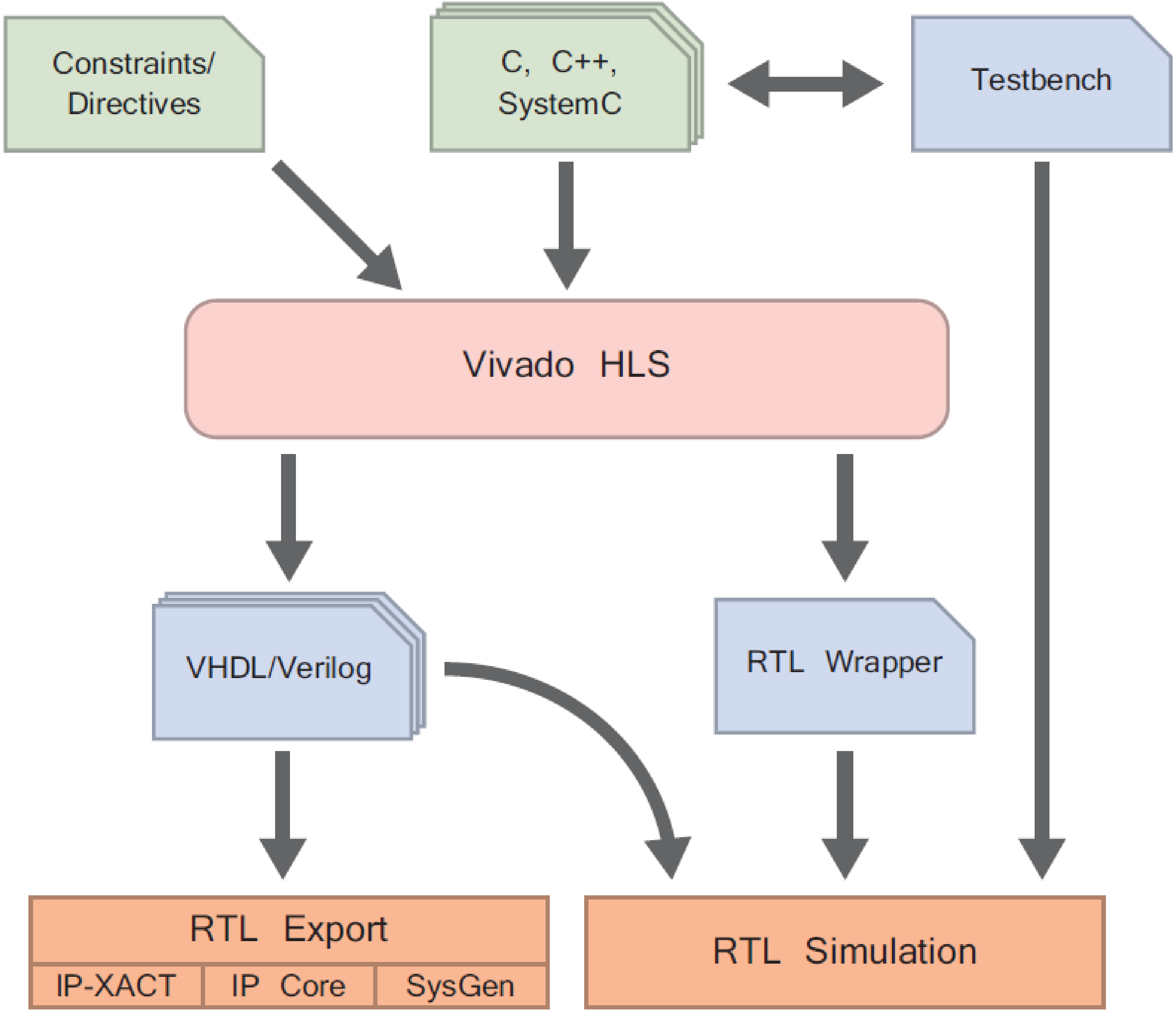
- High-Level Synthesis Concept

- Vivado High-Level Synthesis
  - Inputs and Outputs
  - High-Level Synthesis Process
    - Interface Synthesis
    - Algorithm Synthesis
  - Algorithm Optimizations
    - Loop
    - Array

- Lab Exercise: Accelerating Floating Point Matrix Multiplication with HLS

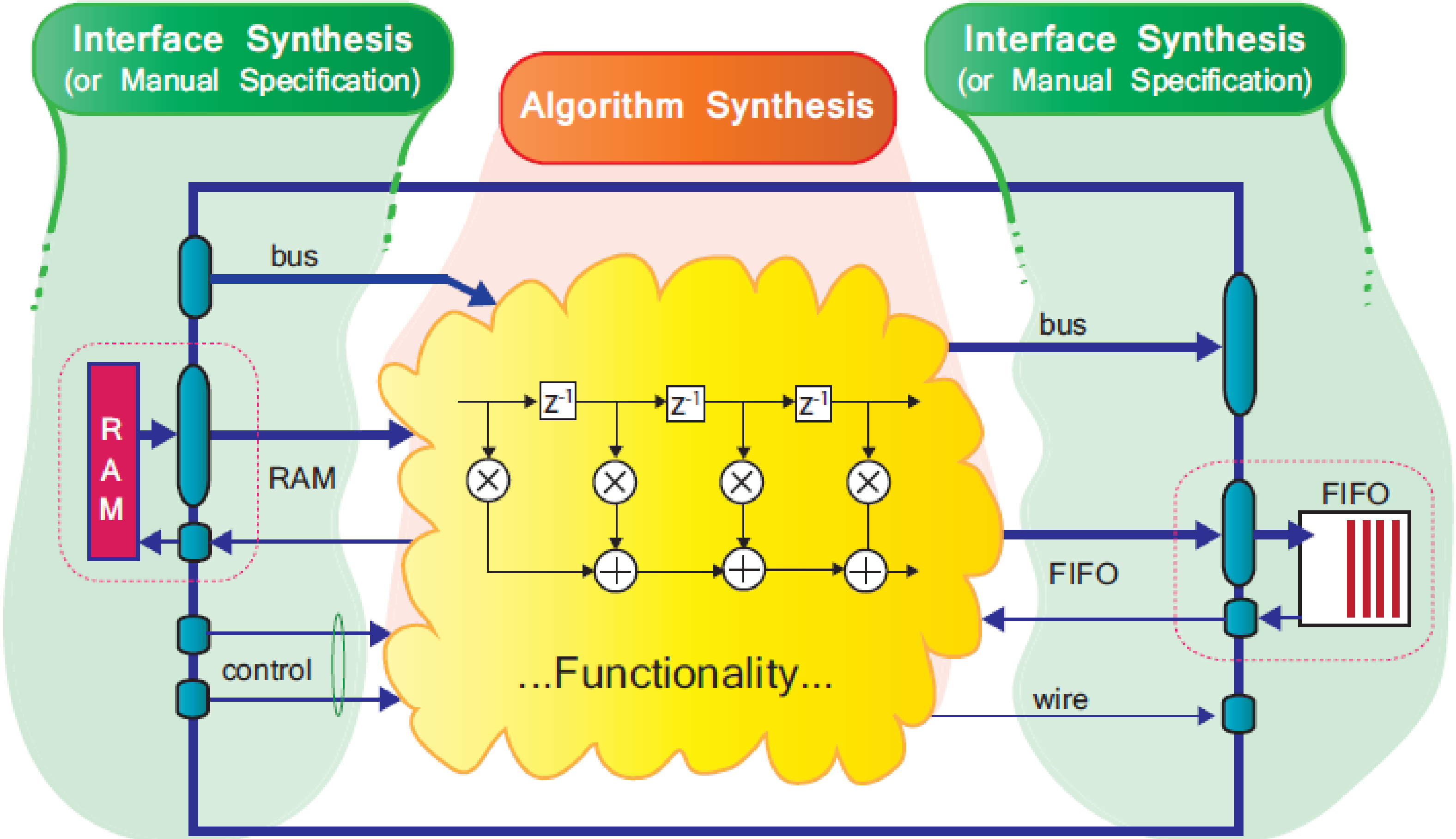
Vivado HLS


Vivado™ HLS


Constraints/
Directives
C, C++,
SystemC
Testbench
Vivado HLS
VHDL/Verilog
RTL Wrapper
RTL Export
IP-XACT
IP Core
SysGen
RTL Simulation

Vivado HLS Process (It's automatic!)

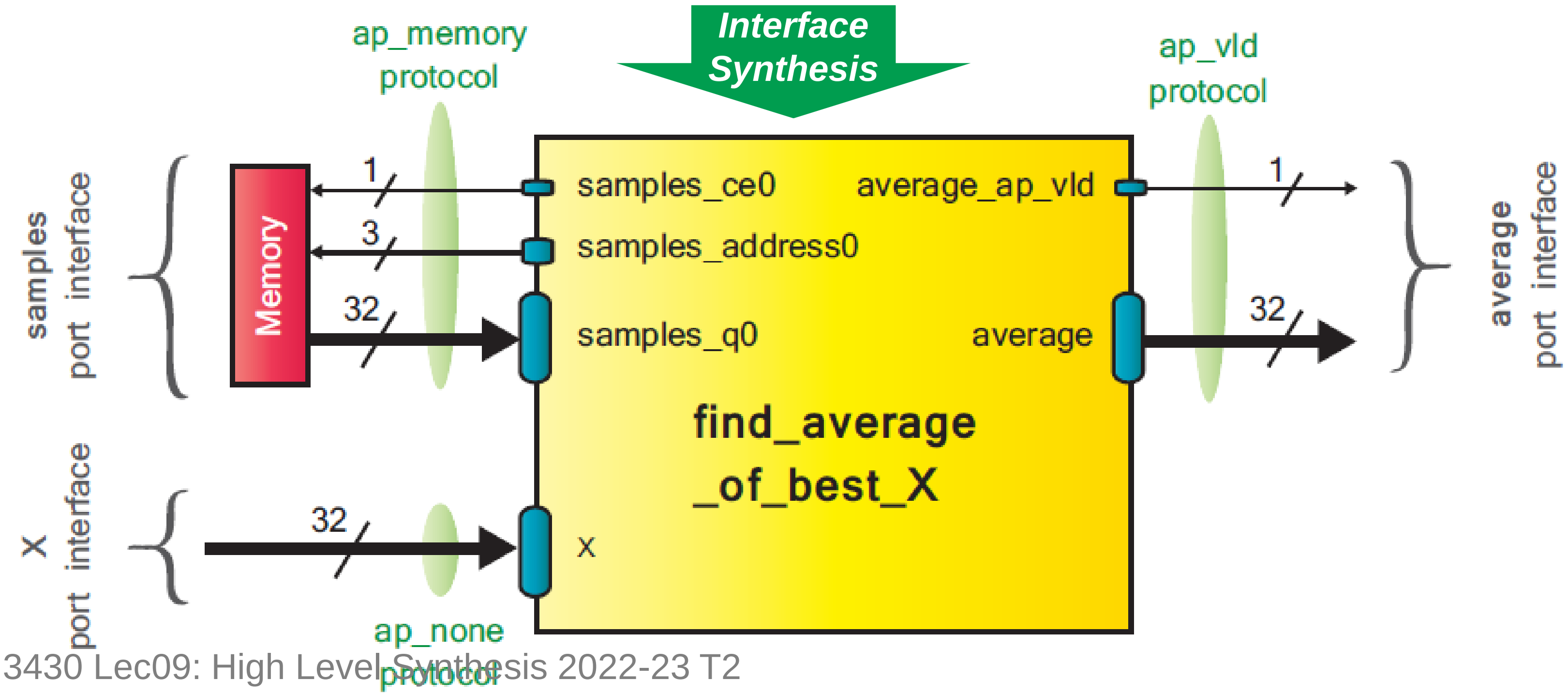
- The HLS process internally involves two major tasks:
  1) The **interface** of the design, i.e., its top-level connections;
  2) The **functionality** of the design, i.e., the algorithm(s).



Vivado HLS: Interface Synthesis

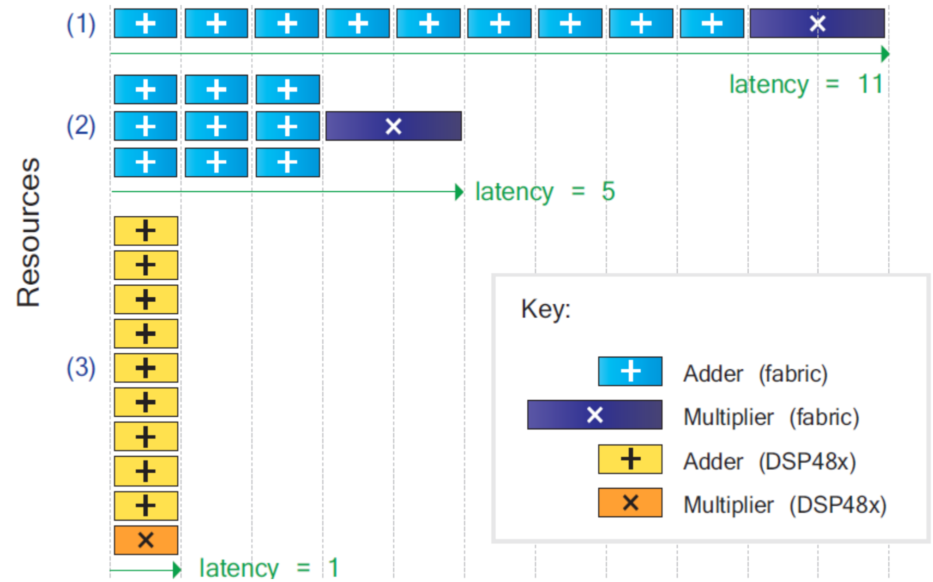
- The **interface** can be created manually or inferred automatically from the code (**interface synthesis**).
  - The **ports** are inferred from the top-level function arguments and return values of the source C/C++ file;
  - The **protocols** are inferred from the behavior of the ports.

```
void find_average_of_best_X (int *average, int samples[8], int X)
```



Vivado HLS: Algorithm Synthesis

- The **algorithm synthesis** comprises three primary stages, which occur in the following order:
  1) **Extraction of Data Path and Control:** Analyze the high-level code and interpret the required functionality;
  2) **Scheduling and Binding:** Translate high-level code & bind the RTL operations onto the target device;
     - It may result in different (i) *latency*, (ii) *throughput*, and (iii) *amount of resources used*.
     - By default, Vivado HLS optimizes the *area*.



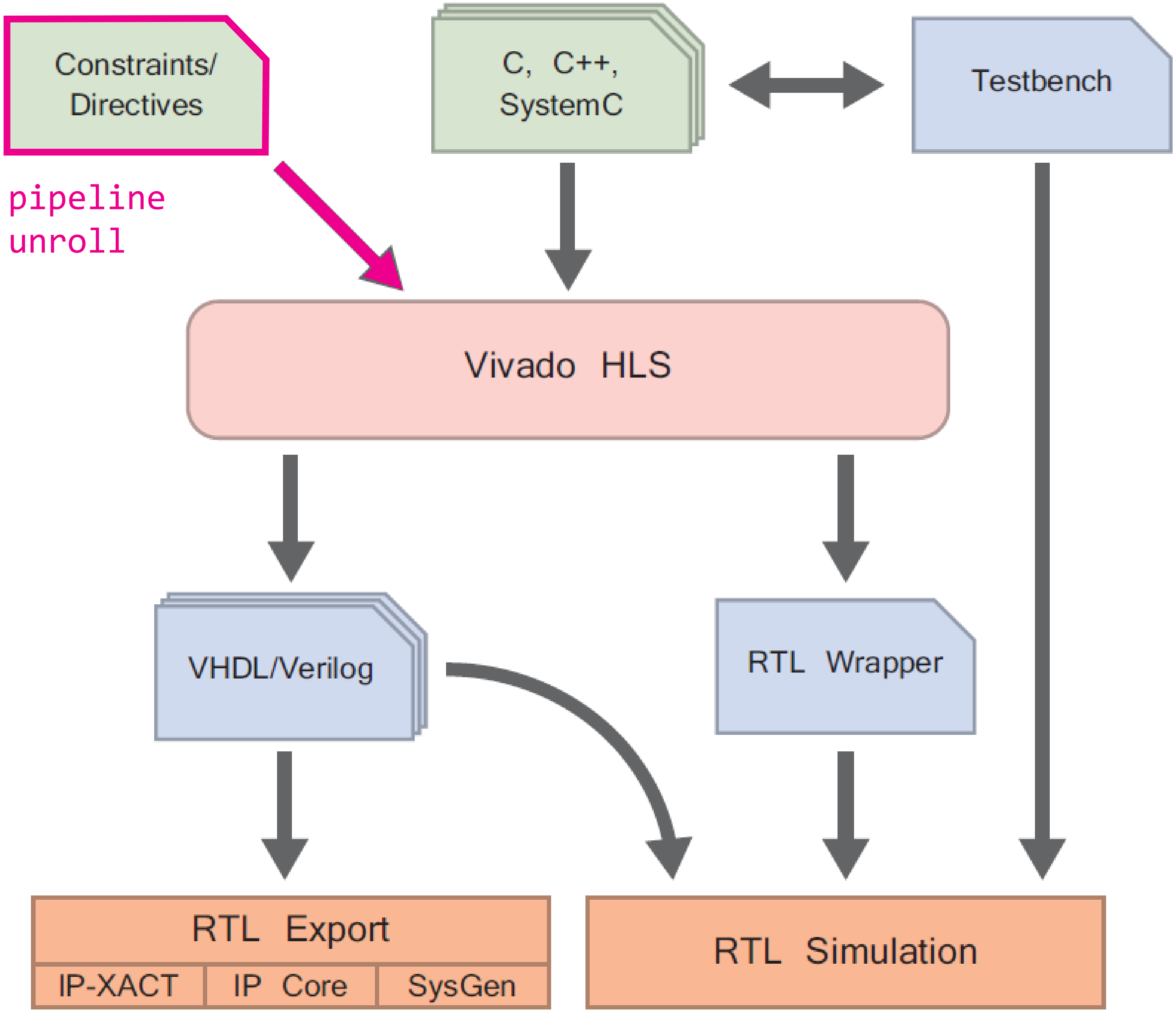
Ex: Calculating the average of ten numbers.

  3) **Optimizations:** Direct the RTL result towards desired optimizations via constraints and directives (i.e., pragmas) without explicitly changing the high-level code!

Revisit HLS: How Directives Work?

```
#pragma HLS pipeline
#pragma HLS unroll
...
```

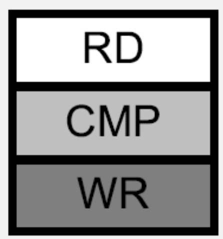
Vivado™ HLS

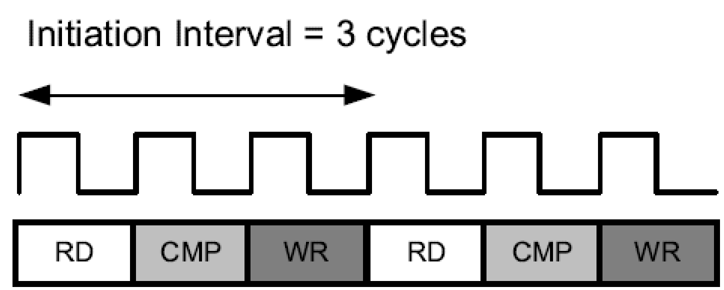


Loop Optimizations

- **Loops** are used extensively in programming.
  - It constitute a natural method of expressing operations that are repetitive in some way.
- By default, Vivado HLS seeks to optimize area.
  - I.e., loops time-share a minimal set of hardware resources.
    - The operations in a loop are executed sequentially.
    - The next iteration can only begin when the last is done.

```
Loop:for(i=1;i<3;i++) {
 op_Read;
 op_Compute;
 op_Write;
}
```



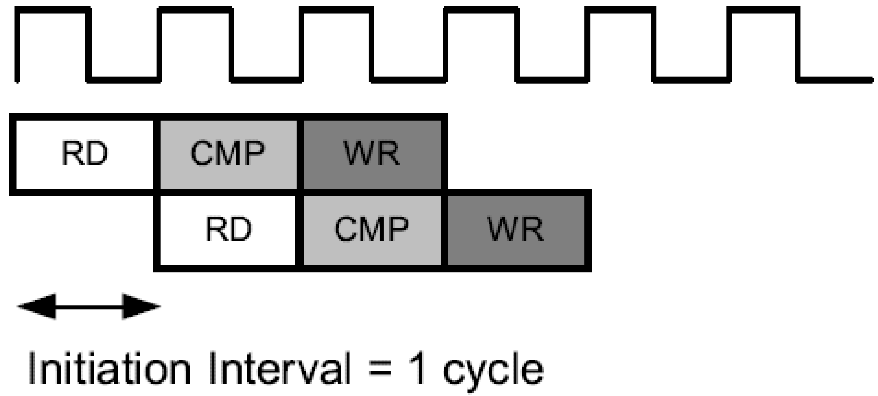


- Loop optimizations can be made using **directives**.
  - Allowing the resulting implementation to be altered with just few or even no changes to the software code.

Loop Optimization #1: Pipelining

- **Loop pipelining** allows the operations in a loop to be implemented in a concurrent manner.

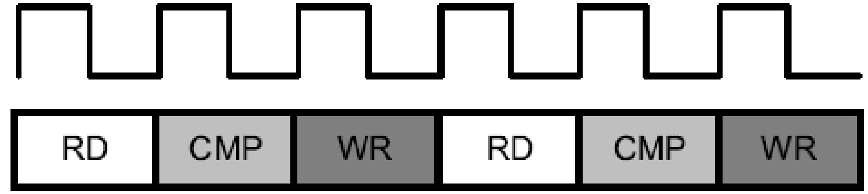
```
Loop:for(i=1;i<3;i++) {
 #pragma HLS pipeline
 op_Read;
 op_Compute;
 op_Write;
}
```



  - The initiation interval (II) is the number of clock cycles between the start times of consecutive loop iterations.
- To pipeline a loop, put the directive "`#pragma HLS pipeline [II=1]`" at the beginning of that loop.
  - Vivado HLS automatically tries to pipeline the loop with the minimum initiation interval (`II`) (i.e., `II=1`).

Class Exercise 9.1

```
Loop:for(i=1;i<3;i++) {
 op_Read;
 op_Compute;
 op_Write;
}
```



- Assume it takes a total of six cycles to complete the loop originally. How many cycles are needed if we pipeline the loop with initiation interval (II) set to 2?

Loop Optimization #2: Unrolling

- **Loop unrolling** creates copies of the loop body to lead to higher parallelism and throughput.
  - Unrolling a loop by a factor of N means to create N copies.
    - N < the total number of loop iterations? It is called a “partial unroll”.
    - N = the total number of loop iterations? It is called a “full unroll”.

Rolled Loops

```
int sum = 0;
for(int i = 0; i < 10; i++) {
    #pragma HLS unroll factor=2
    sum += a[i];
}
```

Loops Unrolled by a Factor of 2

```
int sum = 0;
for(int i = 0; i < 10; i+=2) {
    sum += a[i];
    sum += a[i+1];
}
```

- To unroll a loop, put the directive “`#pragma HLS unroll [factor=N]`” at the beginning of that loop.
  - The loop will be fully unrolled by default.

Class Exercise 9.2

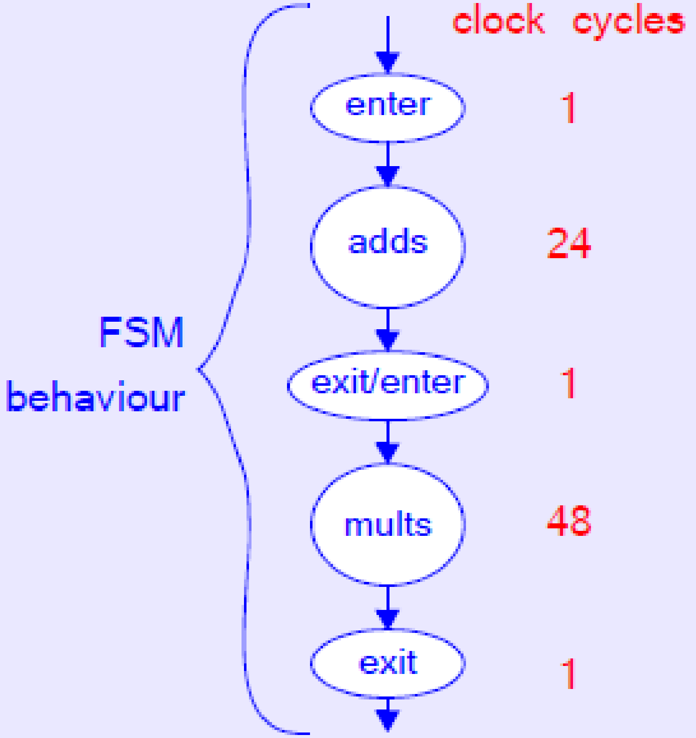
- How many loop iterations are needed if we partially unroll the loop with a factor of 2?

```
void  top(...) {     ...
 for_mult:for (i=3;i>0;i--)  {
       a[i] = b[i] * c[i];
  }
 ...
}
```

Loop Optimization #3: Merging (1/2)

- In some cases, there might be multiple loops occurring one after the other in the code.
  - For instance, the addition loop is followed by a similar loop which multiplies the elements of the two arrays.

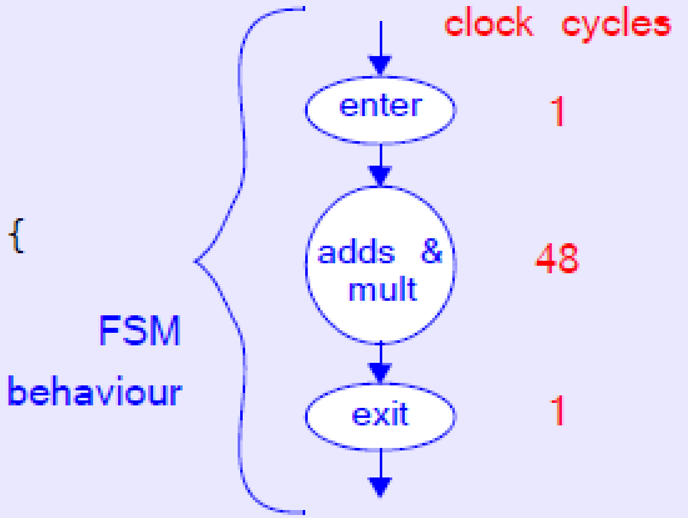
```
void add_mult (short c[12], short m[12], short a[12], short b[12])
{
  short j;

  add_loop: for (j=0;j<12;j++) {
        c[j] = a[j] + b[j];
     }

  mult_loop: for (j=0;j<12;j++) {
        m[j] = a[j] * b[j];
     }
}
```



Loop Optimization #3: Merging (2/2)

- One possible optimization is to **merge** the two loops.
  - That is, both the addition and multiplication operations are conducted within the single loop body.

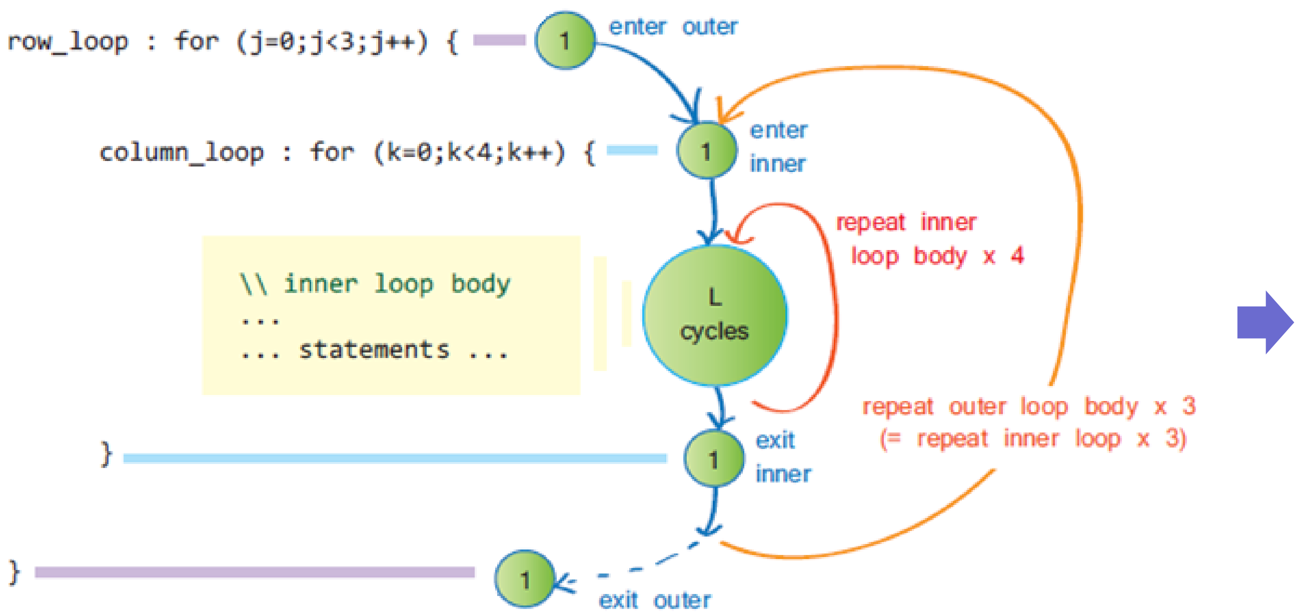
```
void add_mult (short c[12], short m[12], short a[12], short b[12])
{
  short j;


  add_mult_loop: for (j=0;j<12;j++) {
        c[j] = a[j] + b[j];
        m[j] = a[j] * b[j];
     }
}
```



- To merge loops, put directive "`#pragma HLS loop_merge`" at the beginning of a function/loop body.
  - There is no need to explicitly change to the source code!

Loop Optimization #4: Flattening

- We may also "**flatten**" nested loops to remove the loop hierarchy via "`#pragma HLS loop_flatten`".
  - It saves clock cycles transitioning into/out of an (inner) loop.

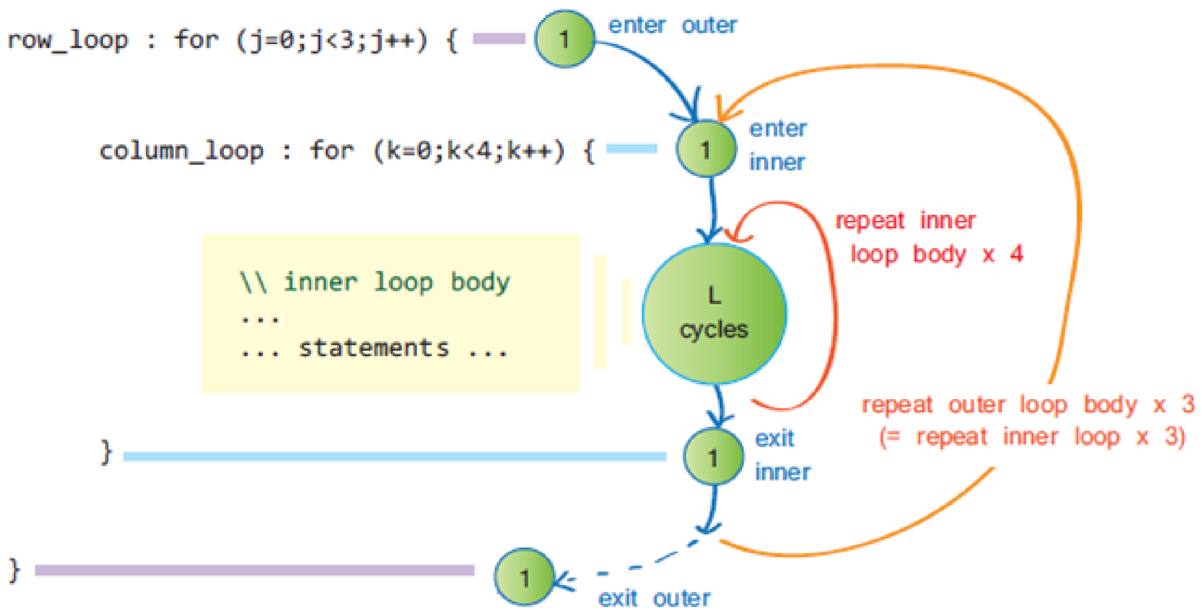



```
// flattened loop
for (i=0;i<3*4;i++) {

   ... statements ...

}
```

  - Then, we may achieve better pipeline optimization or apply larger unrolling factors for higher parallelism.
    - It explains why Vivado HLS flattens the nested loops automatically when the inner loop is pipelined.

Class Exercise 9.3

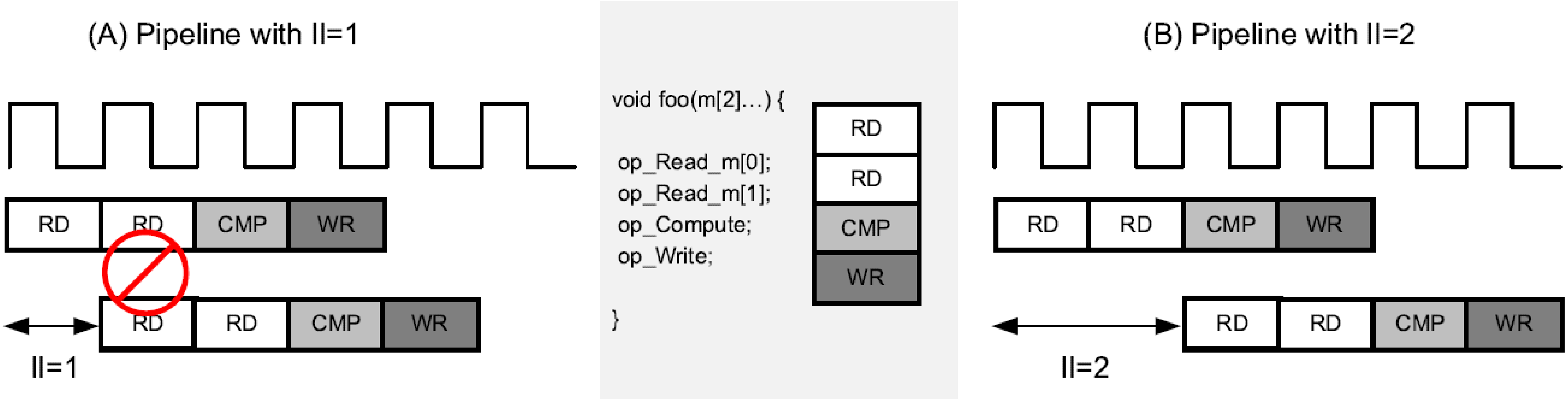
- Consider the following example. How many clock cycles can be saved if the nested loops are flattened?




```
// flattened loop
for (i=0;i<3*4;i++) {

   ... statements ...

}
```

Factor Limiting the Parallelism?

- Loop optimizations aim to achieve higher parallelism.
- One limiting factor for parallelism is the number of available hardware resources.
  - If the loop is pipelined with an initiation interval of one, there are two read operations.
    - If the memory has only one port, then two read operations cannot be executed simultaneously and must be executed in two cycles.
  - Thus, the minimal initiation interval (II) can only be two.

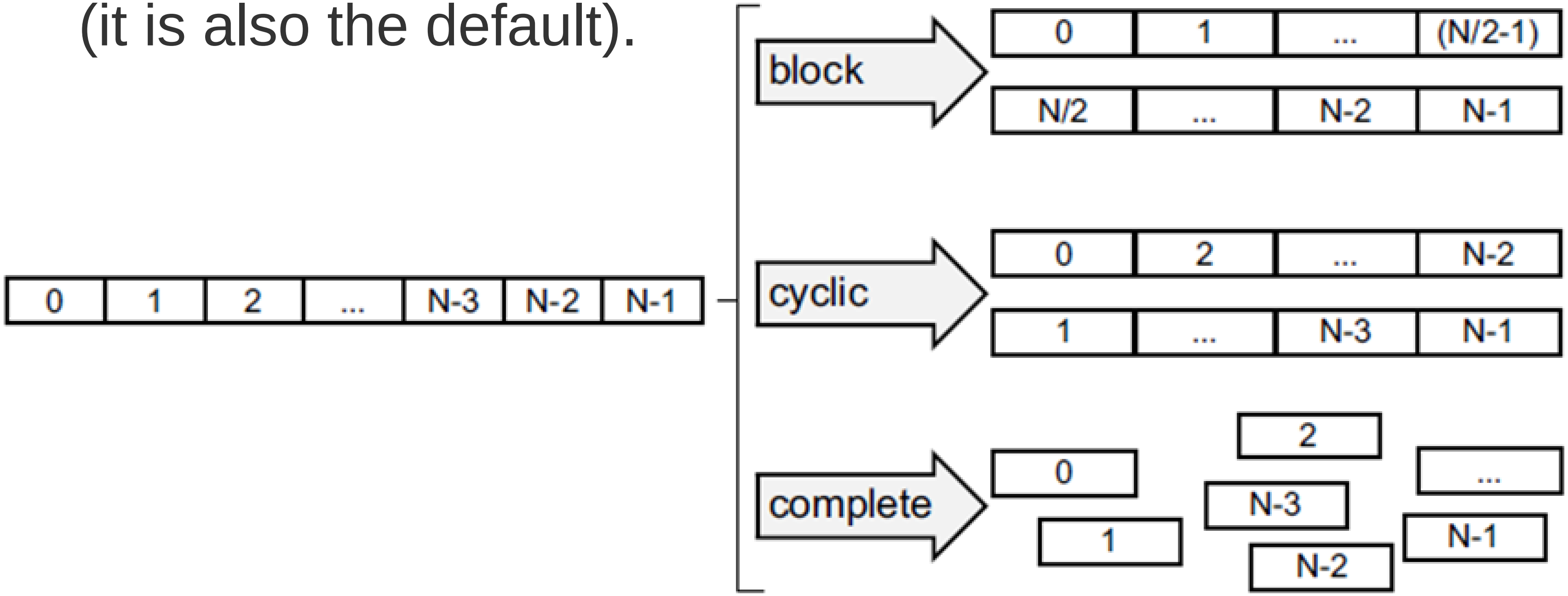


Array Optimization: Partitioning (1/3)

- **Arrays** are usually mapped to the Block RAM (BRAM) of PL, where BRAM has limited read/write ports.
- **Partitioning** an array into smaller arrays increases the port number and may improve the throughput.
- To partition an array, put directive "`#pragma HLS array_partition [arguments]`" within the boundaries where the array variable is defined.
  - **`variable=<name>`**: Specifies the array to be partitioned.
  - **`<type>`**: Optionally specifies the partition type.
  - **`factor=<int>`**: Specifies the number of smaller arrays that are to be created/partitioned.
  - **`dim=<int>`**: Specifies which dimension of a multi-dimensional array to partition.

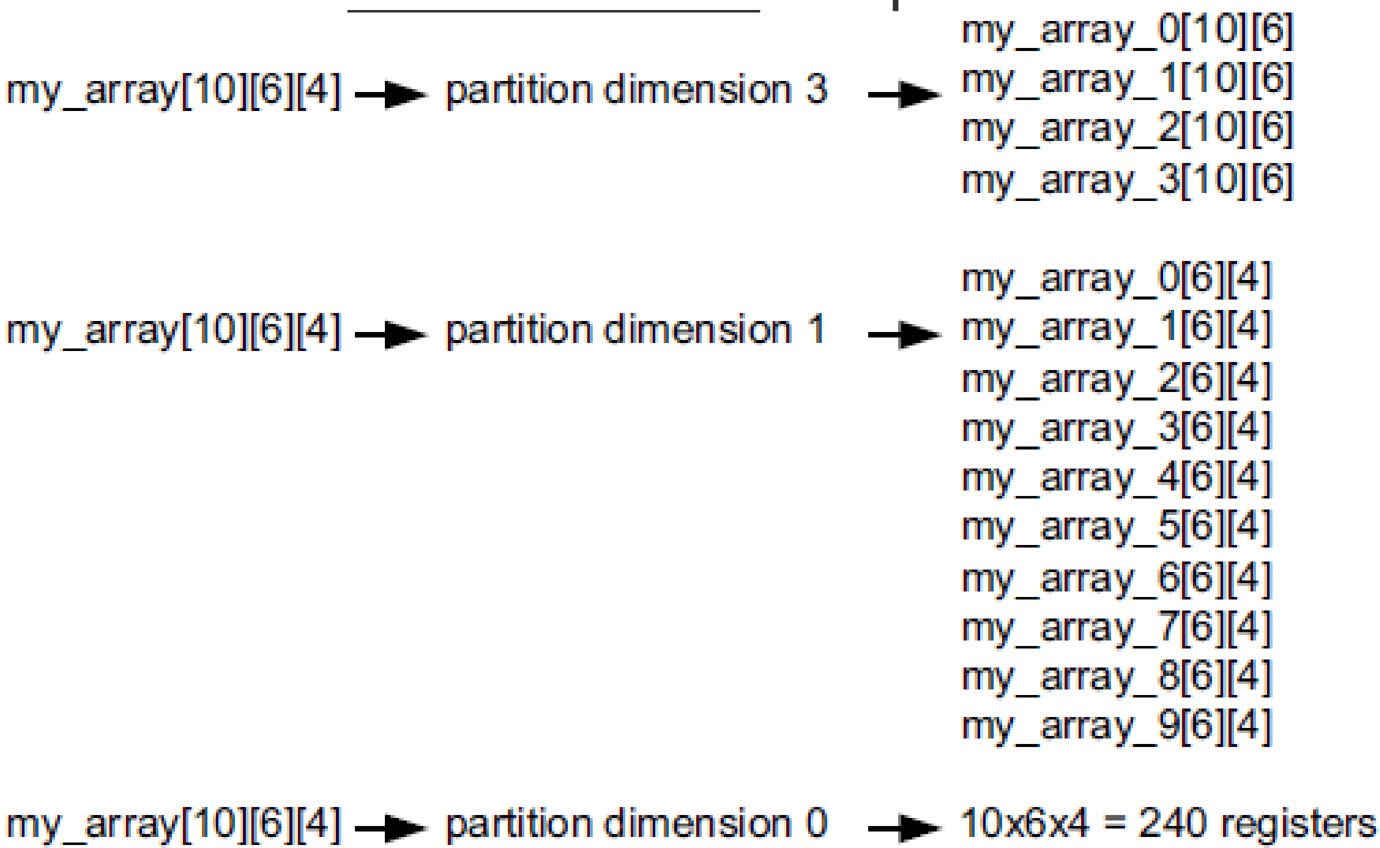
Array Optimization: Partitioning (2/3)

- The **<type>** argument specifies the partition type:
  - **block**: Splits the array into N equal blocks, where N is the integer defined by the `factor` argument.
  - **cyclic**: Creates smaller arrays by interleaving elements from the original array.
  - **complete**: Decomposes the array into individual elements (it is also the default).



Array Optimization: Partitioning (3/3)

- The **<dim>** argument specifies which dimension of a multi-dimensional array to partition.
  - Non-zero value: Only the <u>specified dimension</u> is partitioned.
  - A value of 0: <u>All dimensions</u> are partitioned.



Class Exercise 9.4

- Consider the matrix multiplication, how should matrices a and b be partitioned for better parallelism?

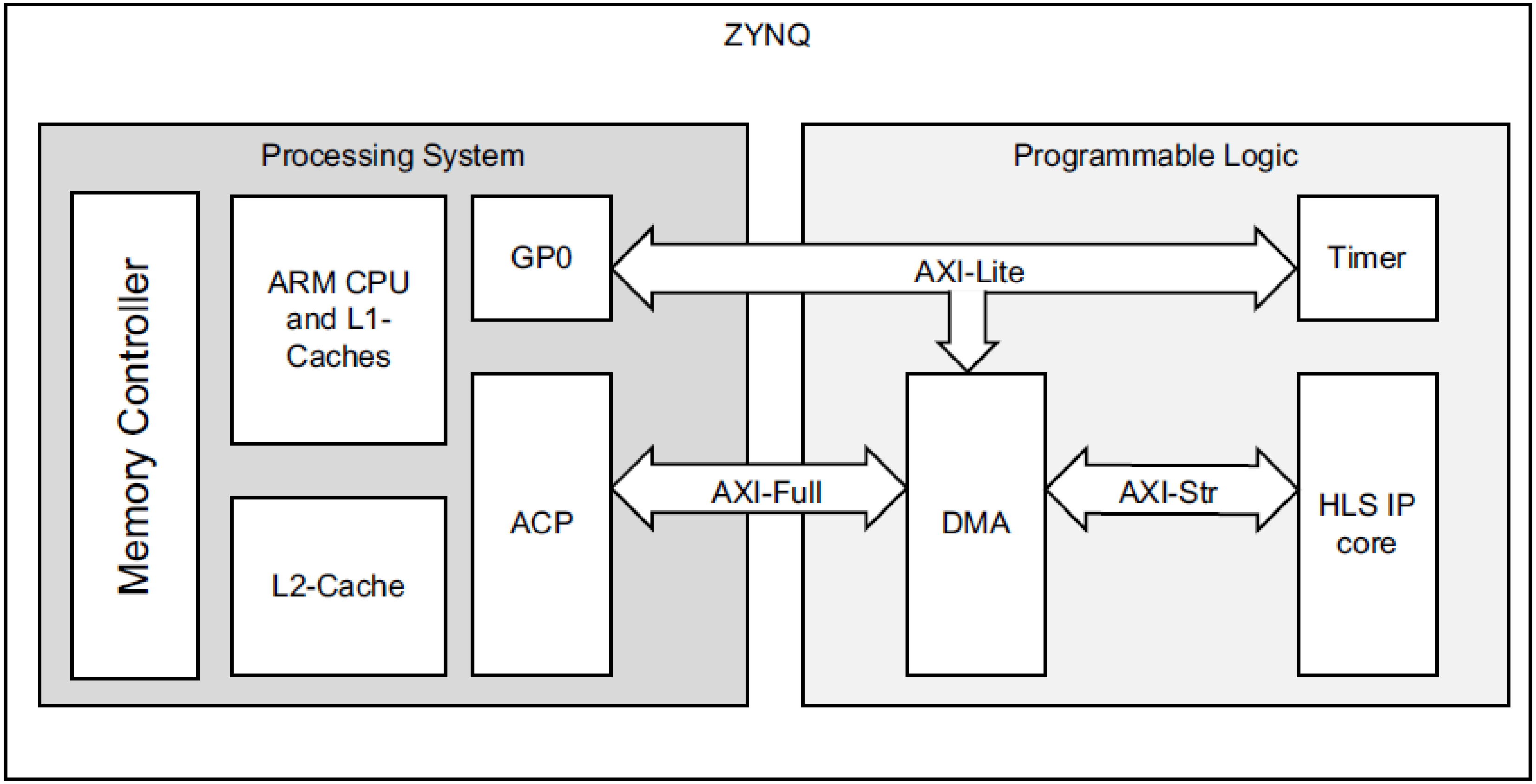
$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

Outline

- High-Level Synthesis Concept
- Vivado High-Level Synthesis
  - Inputs and Outputs
  - High-Level Synthesis Process
    - Interface Synthesis
    - Algorithm Synthesis
  - Algorithm Optimizations
    - Loop
    - Array
- Lab Exercise: Accelerating Floating Point Matrix Multiplication with HLS

Lab Exercise: Matrix Multiplication (1/4)

- In this lab, we will develop an accelerator for the floating-point multiplication on 32x32 matrices.
  - The accelerator is connected to an AXI DMA peripheral in PL and then to the accelerator coherence port (ACP) in PS.

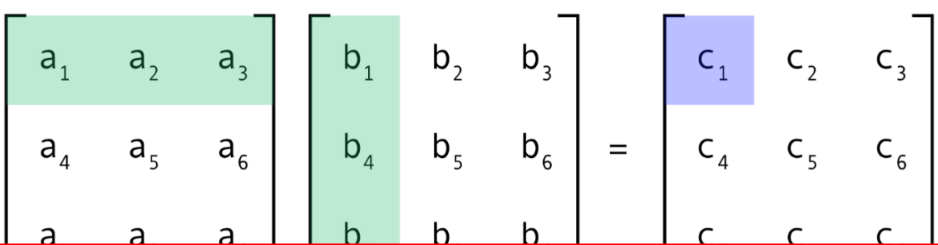


Lab Exercise: Matrix Multiplication (2/4)

- The function to be optimized is defined in "`mmult.h`":

```
template <typename T, int DIM>
void mmult_hw(T A[DIM][DIM], T B[DIM][DIM], T C[DIM][DIM])
{
  // matrix multiplication of a A*B matrix
  L1:for (int ia = 0; ia < DIM; ++ia)
  {
    L2:for (int ib = 0; ib < DIM; ++ib)
    {
      T sum = 0;
      L3:for (int id = 0; id < DIM; ++id)
      {
        sum += A[ia][id] * B[id][ib];
      }
    C[ia][ib] = sum;
  }
}
```

← L1 iterates over the rows of the input matrix A.
← L2 iterates over columns of the input matrix B.
← L3 multiplies each element of row vector A with an element of column vector B and accumulates it to the elements of a row of the output matrix C.



How? Utilize "directives" properly to direct HLS!

Lab Exercise: Matrix Multiplication (3/4)

- **Resource Cost** (Post-Implementation Utilization)

Utilization - Post-Implementation

Resource	Utilization	Available	Utilization %
LUT	4195	53200	7.89
LUTRAM	250	17400	1.44
FF	5054	106400	4.75
BRAM	8	140	5.71
DSP	5	220	2.27
BUFG	1	32	3.13

Should NOT over-utilize the resources!

Graph **Table**

Post-Synthesis **Post-Implementation**

Lab Exercise: Matrix Multiplication (4/4)

- **Performance** (Latency and HW/SW Speedup)

Performance Estimates

⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.41	1.25

Should NOT violate timing constraint!

(i.e., the estimated clock period should be less than the target one)

⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
332872	332872	332873	332873	none

The higher, the slower!

SDK Log | Terminal 1

Serial: (COM6, 115200, 8, 1, None, None - CONNECTED) - Encoding: (ISO-8859-1)

```
DMA Init done
Loop time for 1024 iterations is -2 cycles
Running Matrix Mult in SW


Total run time for SW on Processor is 25880 cycles over 1024 tests.
Cache cleared
Total run time for AXI DMA + HW accelerator is 333830 cycles over 1024 tests
Acceleration factor: 0.77
```

The lower, the slower!

Summary

- High-Level Synthesis Concept
- Vivado High-Level Synthesis
  - Inputs and Outputs
  - High-Level Synthesis Process
    - Interface Synthesis
    - Algorithm Synthesis
  - Algorithm Optimizations
    - Loop
    - Array
- Lab Exercise: Accelerating Floating Point Matrix Multiplication with HLS