香港中文大學
The Chinese University of Hong Kong

*CENG3430 Rapid Prototyping of Digital Systems*

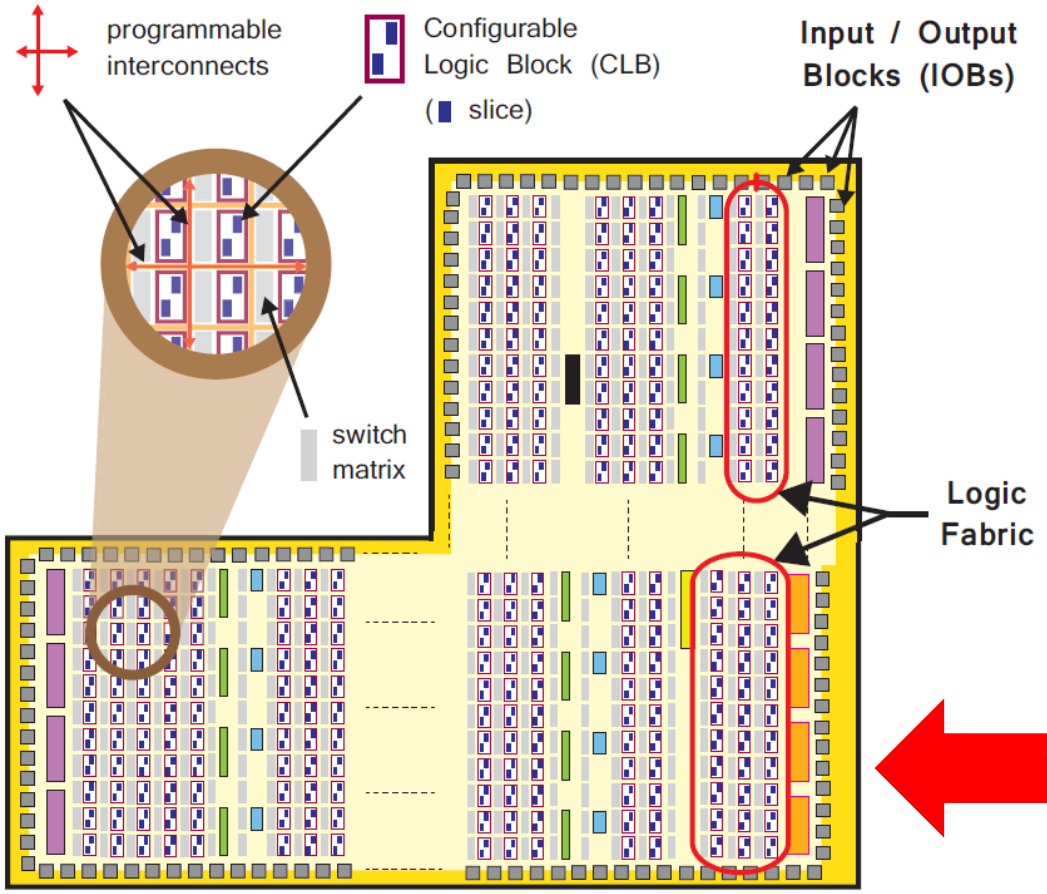
# Lecture 09: Rapid Prototyping (III) – High Level Synthesis

**Ming-Chang YANG**

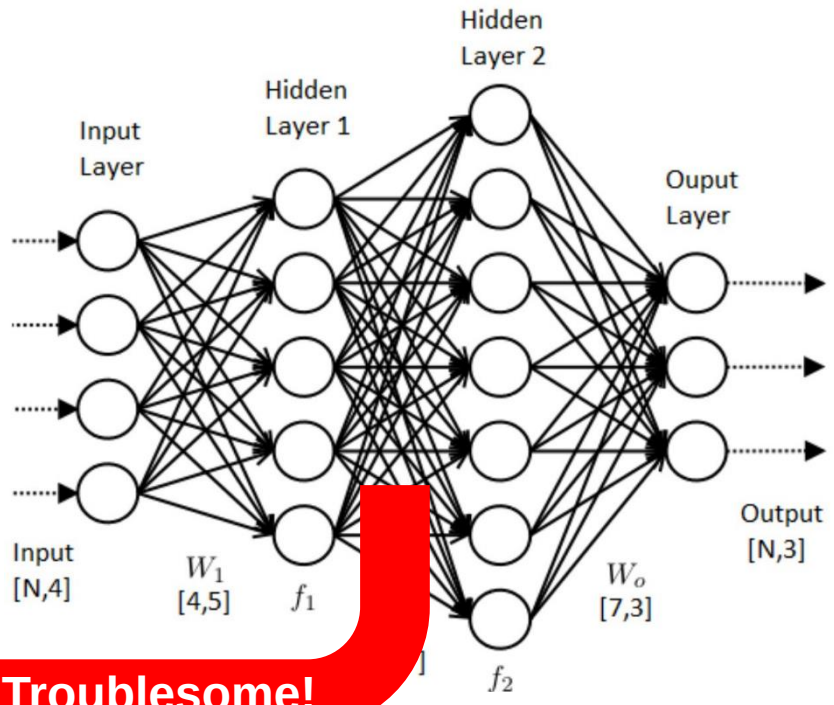
mcyang@cse.cuhk.edu.hk

# What else can we do with ZedBoard?

- PL section is also ideal for implementing high-speed and high-parallel logic and arithmetic, thanks to its highly-programmable logic fabrics.




Ex: Neural Network




**Troublesome!**

# Outline

- High-Level Synthesis Concepts
  - What is, and Why High-Level Synthesis
  - Design Metrics in HLS
- Vivado High-Level Synthesis
  - Inputs and Outputs
  - High-Level Synthesis Process
    - Interface Synthesis and Algorithm Synthesis
    - Case Studies on Optimizations: Loop and Array
  - Wrap-up: Vivado HLS Design Flow
- Lab Exercise: Accelerating Floating Point Matrix Multiplication with HLS

# Machine Learning Drives HLS Boom

- **High-Level Synthesis** is experiencing a new wave of popularity, driven by its ability to handle **machine-learning** matrices and iterative design efforts.

"When you implement the inference engine, you take the original network model and run it on an edge or mobile device," according to Mike Fingeroff, HLS technologist at Mentor, a Siemens Business. "You reduce the area and power of the IC, but you look at performance, especially if it's part of something like an ADAS in an automated vehicle that has some requirement for real-time performance. To get the best performance you have to tailor the hardware to a specific network and optimize the model to contribute to that."

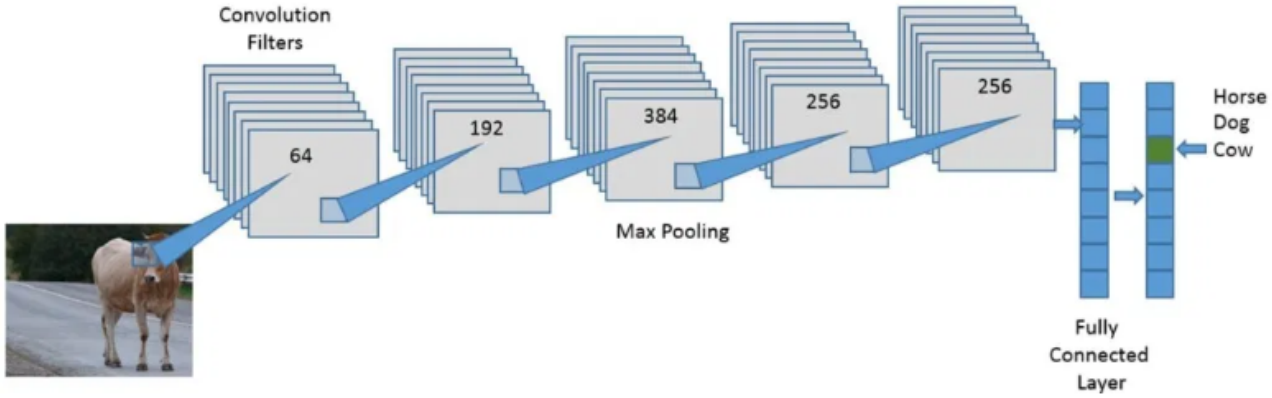
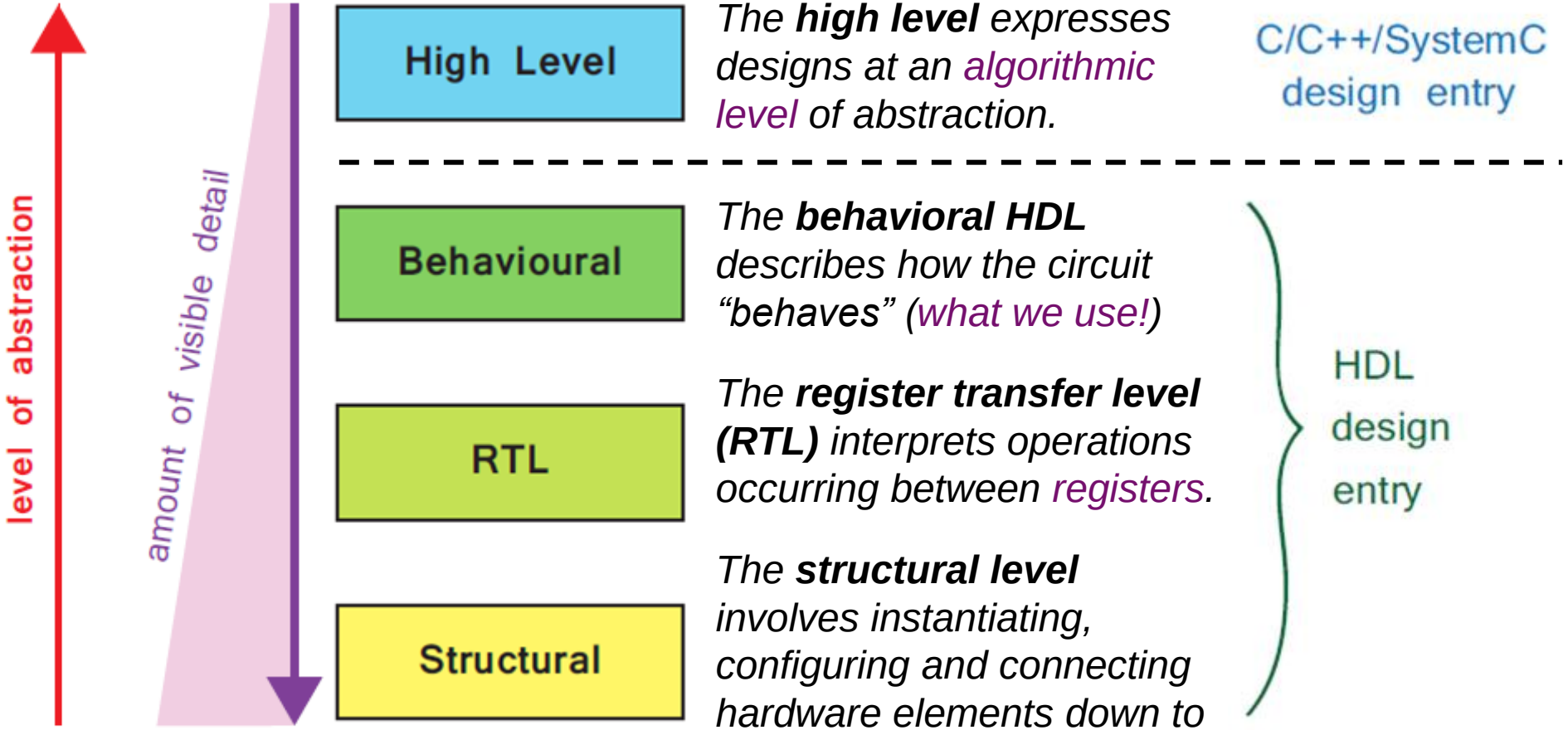



**Fig. 1: Higher levels of abstraction are necessary for complex convolutional designs. Source: Mentor, a Siemens Business**


https://semiengineering.com/machine-learning-drives-high-level-synthesis-boom/

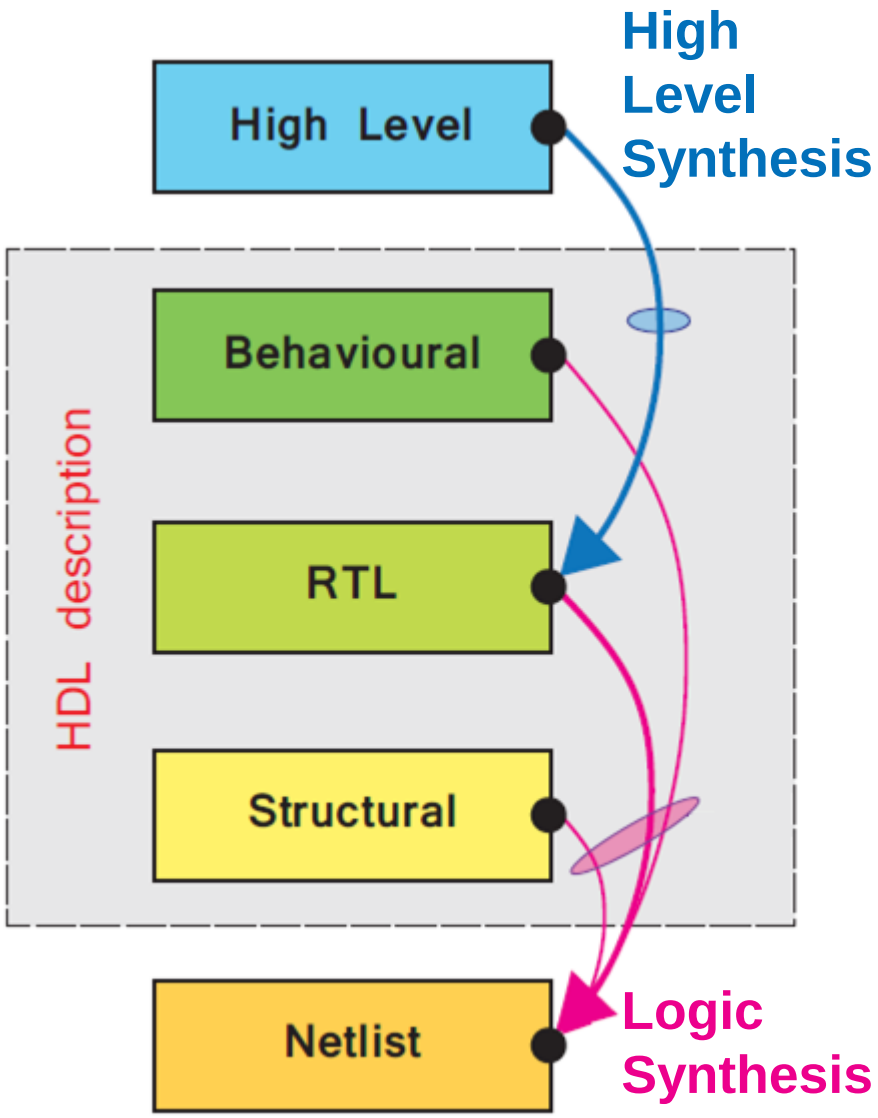
# What is High-Level Synthesis (HLS)?

- By abstracting/hiding low-level details with high-level representations, **high-level synthesis (HLS)** simplifies the description of the circuit dramatically.



# High-Level Synthesis & 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**.
- When taking a HLS design, both types of "syntheses" are applied (one after the other)!



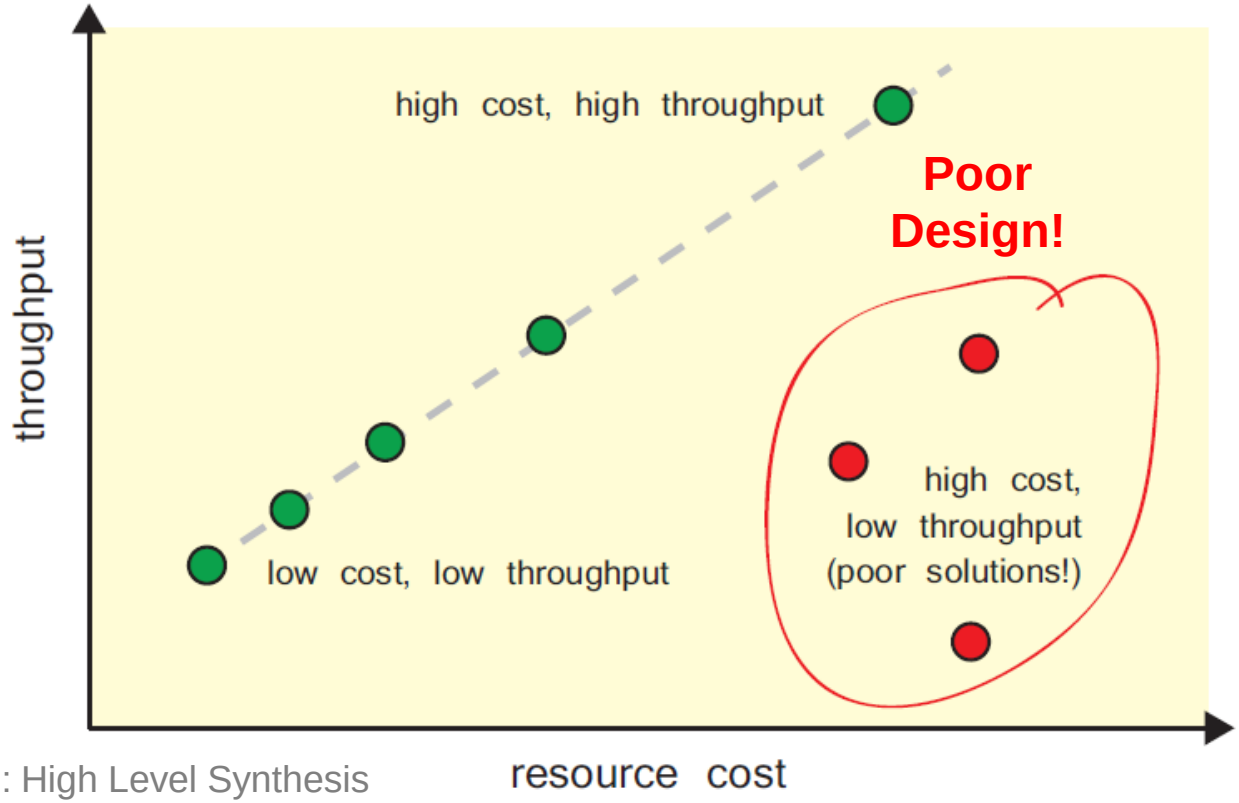
# Why High-Level Synthesis (HLS)?

- The designers simply direct the process, while the HLS tools (i.e., Vivado HLS) implement the details.
  - Designs can be generated much more rapidly.
    - The designer must trust the HLS tools in implementing lower-level functionality correctly and efficiently.
- 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.
    - Rather than having to fundamentally re-work the source code.
- HLS from software languages is convenient.
  - Engineers are comfortable with languages such as C/C++.

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.

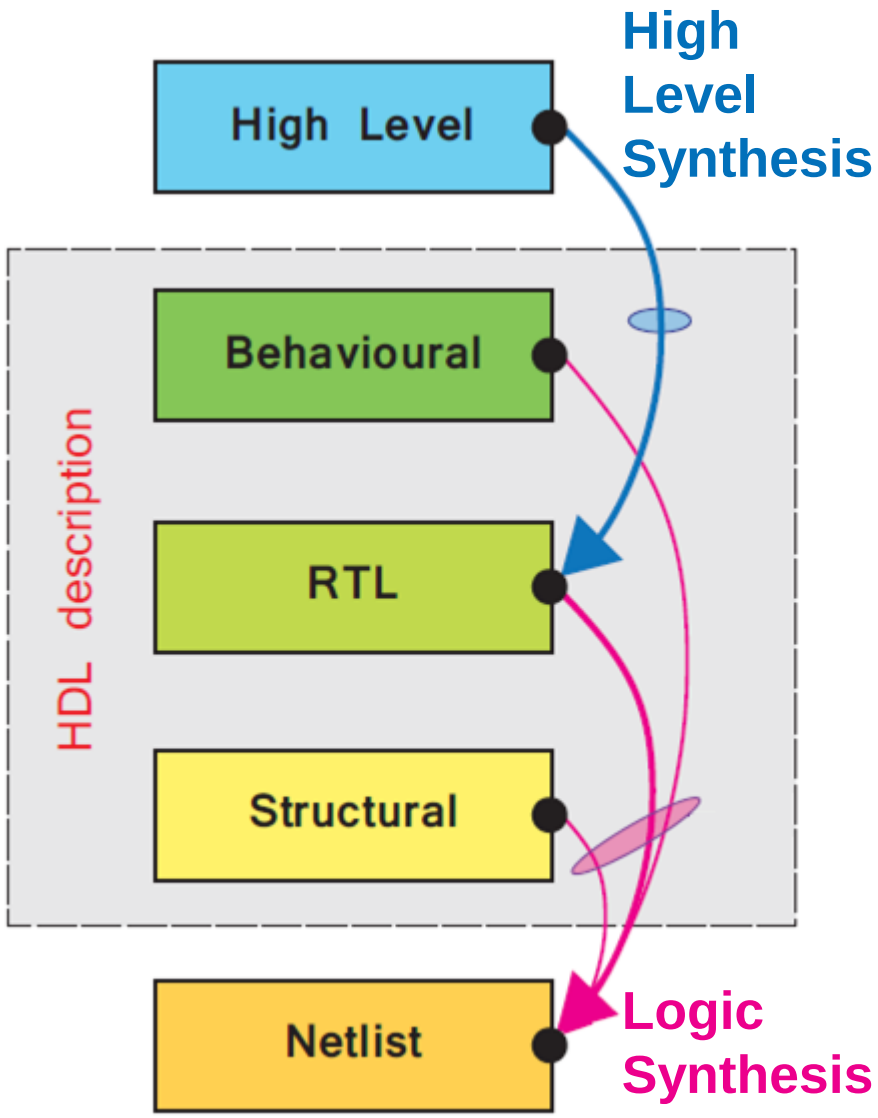


# Outline

- High-Level Synthesis Concepts
  - What is, and Why High-Level Synthesis
  - Design Metrics in HLS
- Vivado High-Level Synthesis
  - Inputs and Outputs
  - High-Level Synthesis Process
    - Interface Synthesis and Algorithm Synthesis
    - Case Studies on Optimizations: Loop and Array
  - Wrap-up: Vivado HLS Design Flow
- Lab Exercise: Accelerating Floating Point Matrix Multiplication with HLS

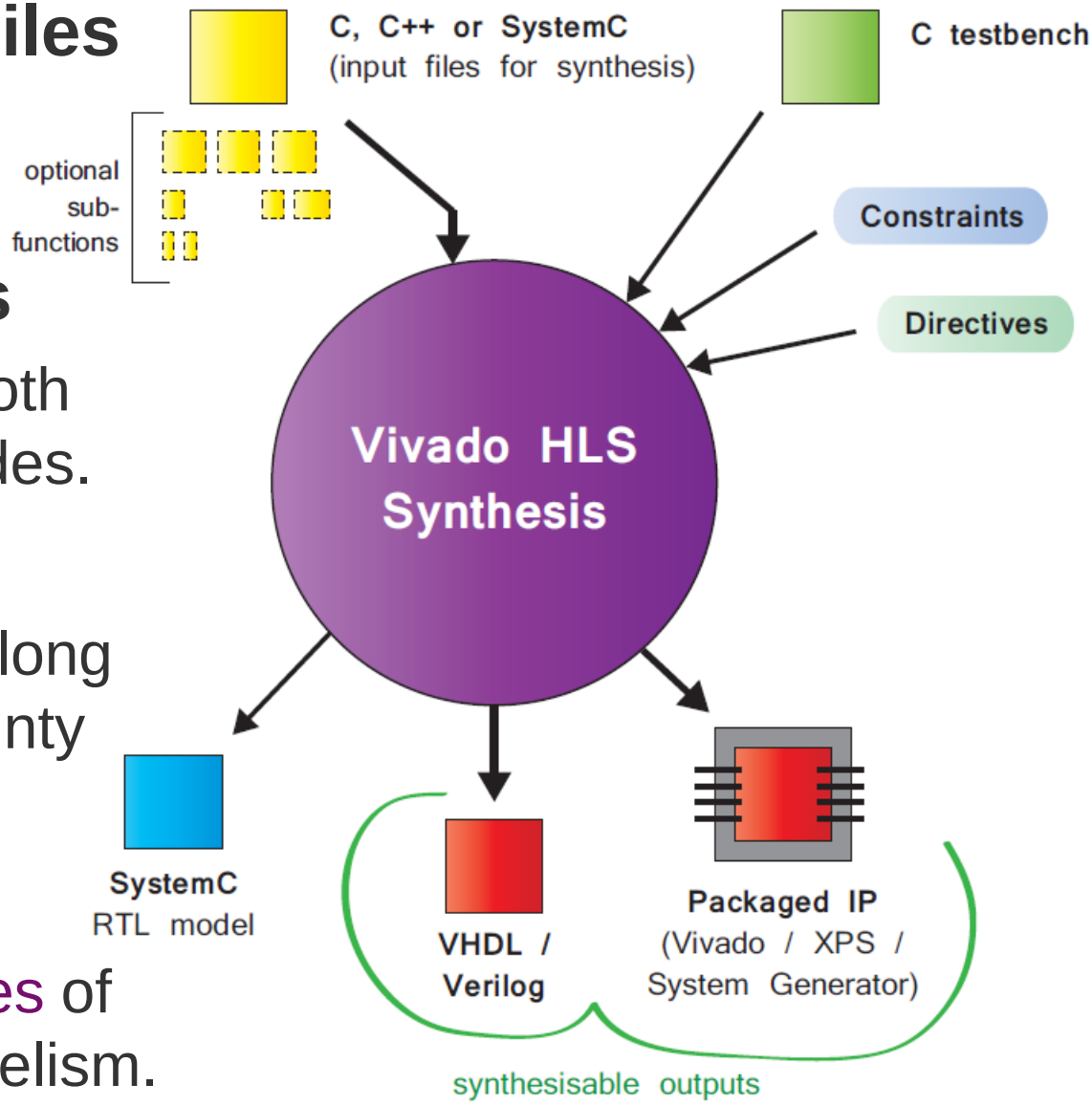
# Vivado HLS

- In short, Vivado HLS:
  1) First transforms a *C*, *C++*, or *SystemC* design into an RTL implementation;
     - That is, all C-based designs in HLS are for implementations in programmable logic (PL).
       - Which are distinct from software code intended to run on the processor.
  2) Then synthesizes the RTL implementation onto the programmable logic (PL) of a Xilinx FPGA or Zynq device.



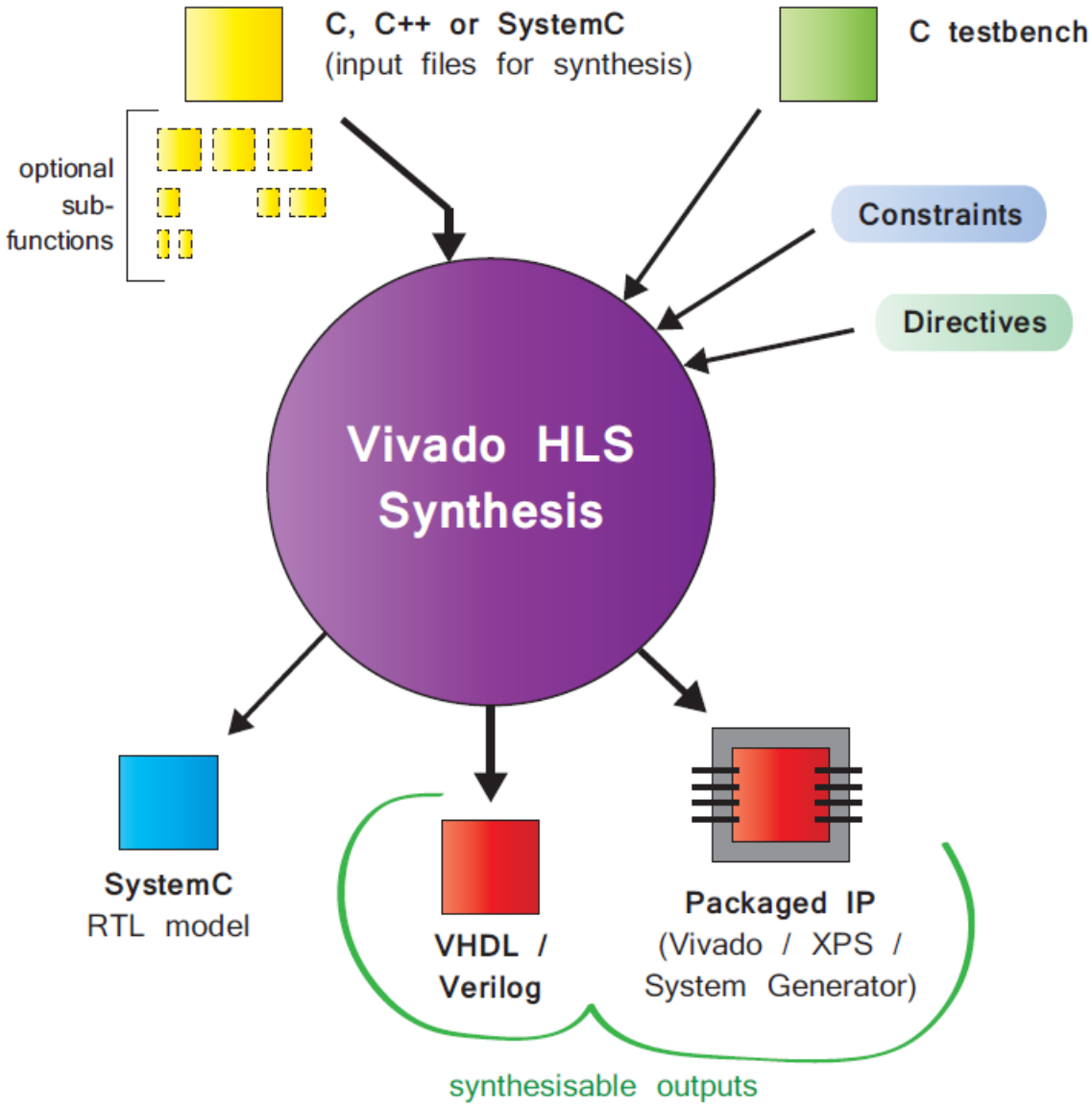
# Inputs to Vivado HLS

1) **C/C++/SystemC Files**
   - Functions to be synthesized.
2) **C Testbench Files**
   - Basis for verifying both C code and RTL codes.
3) **Constraints**
   - Timing constraints along with a clock uncertainty and device details.
4) **Directives**
   - Implementation styles of pipelining and parallelism.



# Outputs from Vivado HLS

- The possible **outputs** produced could be:
  1) **VHDL or Verilog files/codes**
  2) **Packaged IP** (for Block Design in Vivado)
  3) **SystemC model**
- The designer can choose based on different **"prototyping styles"**.

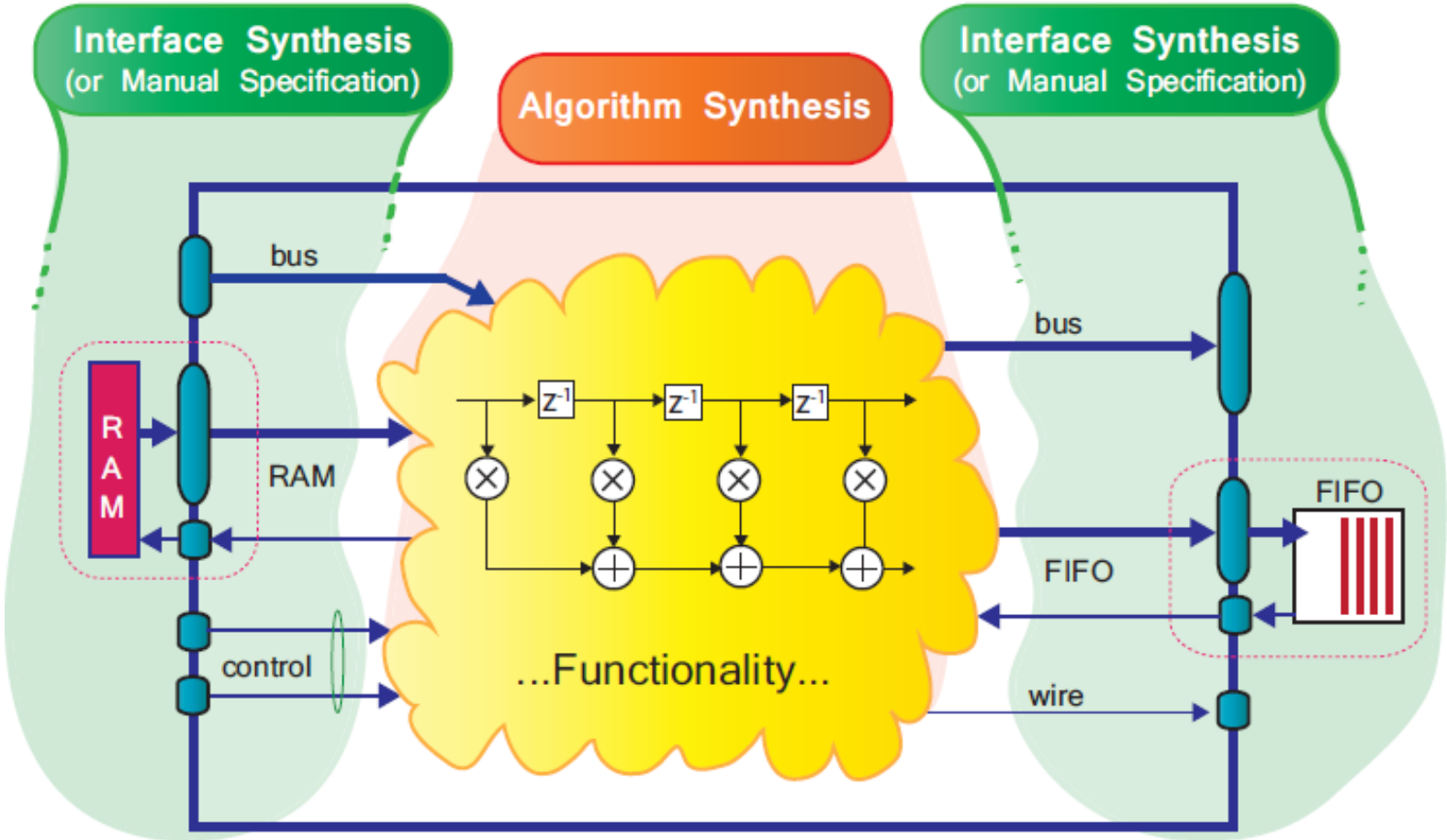


# Outline

- High-Level Synthesis Concepts
  - What is, and Why High-Level Synthesis
  - Design Metrics in HLS
- Vivado High-Level Synthesis
  - Inputs and Outputs
  - High-Level Synthesis Process
    - Interface Synthesis and Algorithm Synthesis
    - Case Studies on Optimizations: Loop and Array
  - Wrap-up: Vivado HLS Design Flow
- Lab Exercise: Accelerating Floating Point Matrix Multiplication with HLS

# Vivado HLS Process

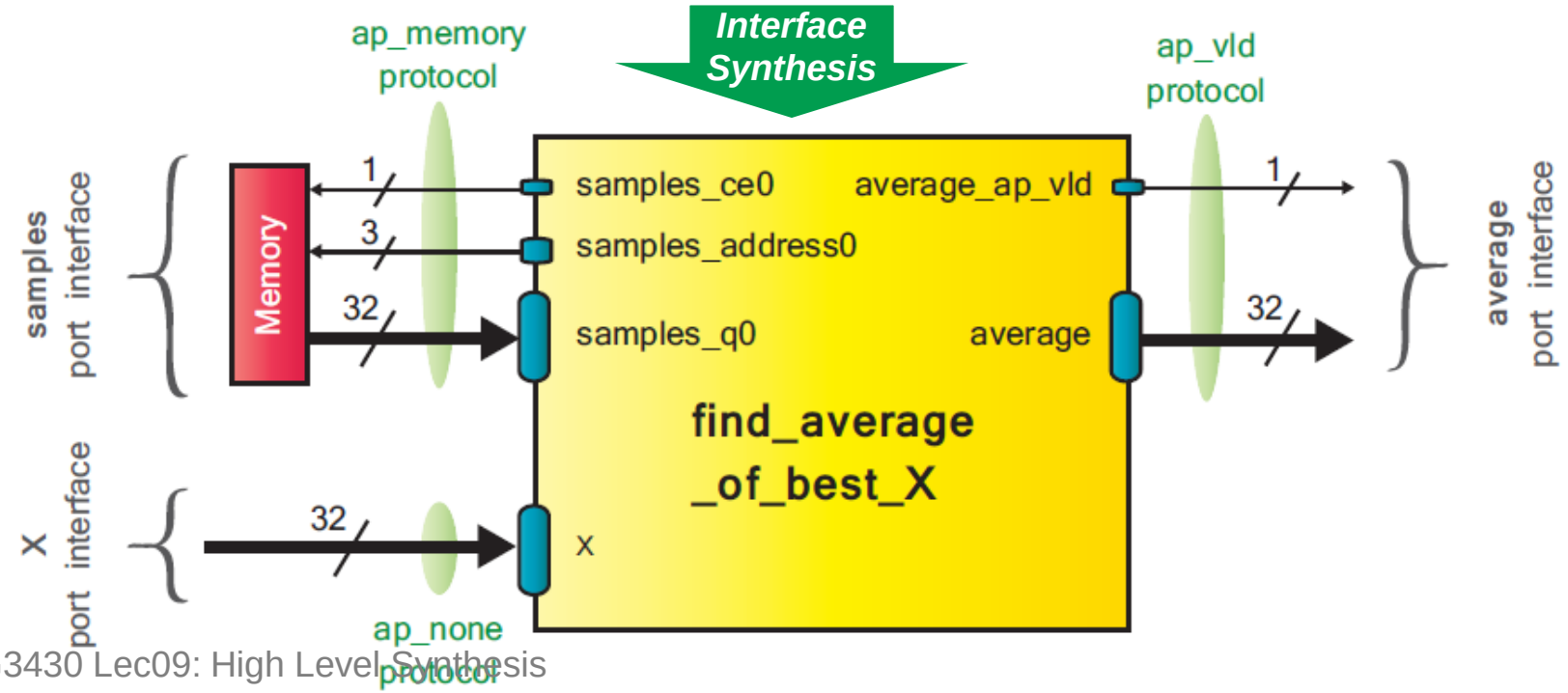
- The HLS process involves two major aspects:
  1) The **interface** of the design, i.e. its top-level connections,
  2) The **functionality** of the design, i.e. the algorithm(s).



# 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 (1/4)

- The **algorithm synthesis** comprises three primary stages, which occur in the following order:
  1) **Extraction of Data Path and Control**
  2) **Scheduling and Binding**
  3) **Optimizations**

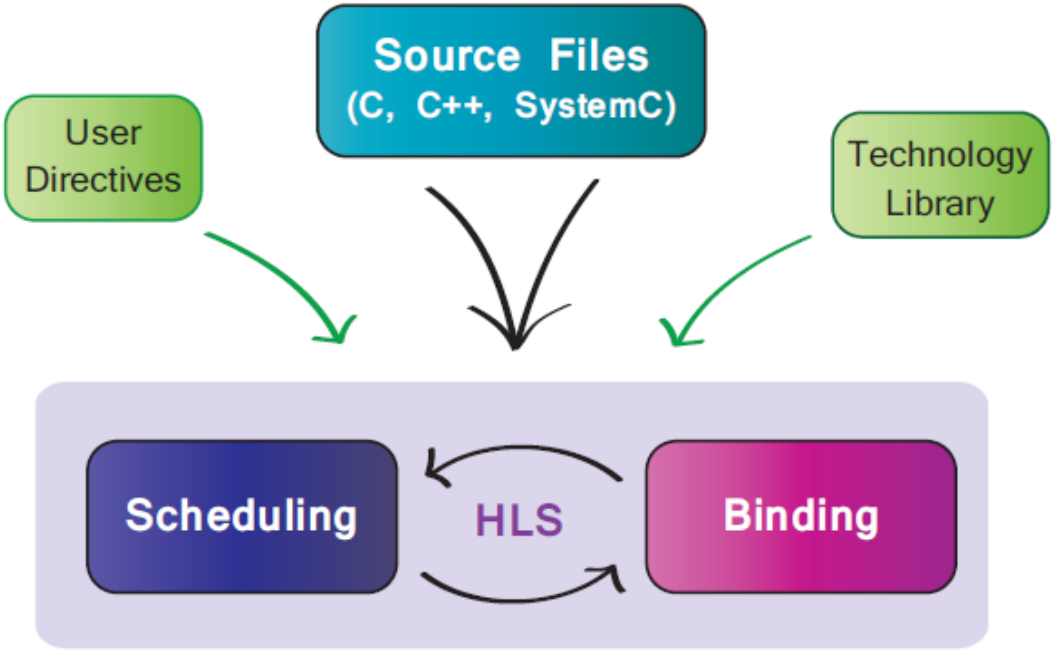
## 1) Extraction of Data Path and Control

- The first stage of HLS is to analyze the C/C++/SystemC code and interpret the required functionality.
- The implementation will normally have a **datapath** component, and a **control** component.
  - **Datapath**: operations performed on the data samples,
  - **Control**: the circuitry required to co-ordinate dataflow processing.

# Vivado HLS: Algorithm Synthesis (2/4)

## 2) Scheduling and Binding

- **Scheduling** is the translation of the RTL statements interpreted from the C code into a set of operations, each with an associated duration in terms of clock cycles.
- **Binding** is the process of associating the scheduled operations with the physical resources of the target device.

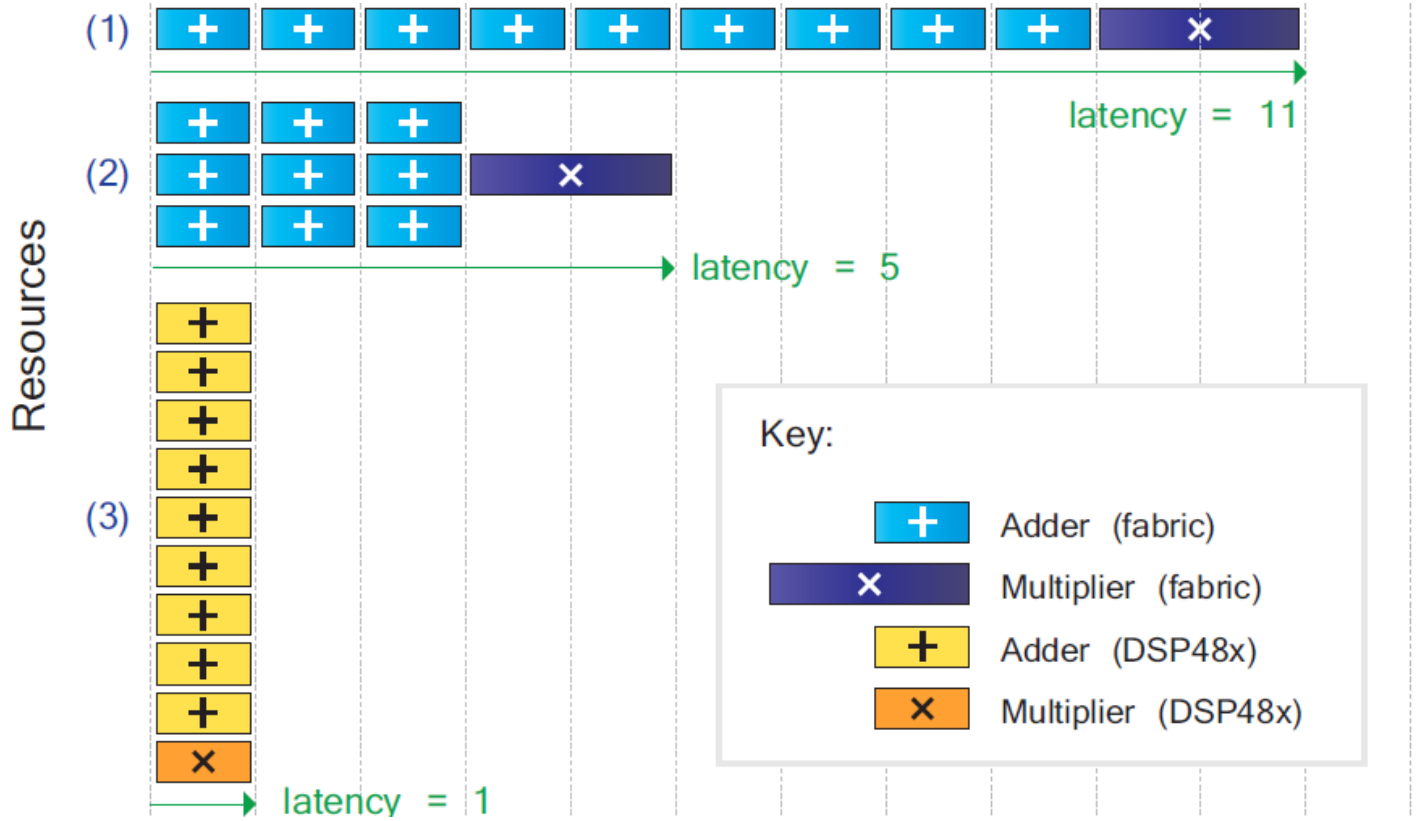


# Vivado HLS: Algorithm Synthesis (3/4)

## 2) Scheduling and Binding (Cont'd)

– The resulting implementation has a set of metrics including (i) latency, (ii) throughput, and (iii) the amount of resources.

- By default, the HLS process optimizes **area** (i.e., the first strategy).




Example: Calculating the average of an array of ten numbers.

# Vivado HLS: Algorithm Synthesis (4/4)

## 3) Optimizations

- The designer can dictate the HLS process towards the desired implementation goals:
  - **Constraints:** The designer places a limit on the design.
    - For instance, the minimum clock period may be specified.
      - This makes it easy to ensure that the implementation meets the requirements of the system into which it will be integrated.
  - **Directives:** The designer can exert more specific influence over aspects of the RTL implementation.
    - HLS tool provides `pragmas` that can be used to optimize the design.
    - For example, how the HLS treats loops or arrays identified in the C code, or the latency of particular operations.
      - This can yield significant changes to the RTL output
      - Therefore, with knowledge of the available directives, the designer can optimize according to application requirements.

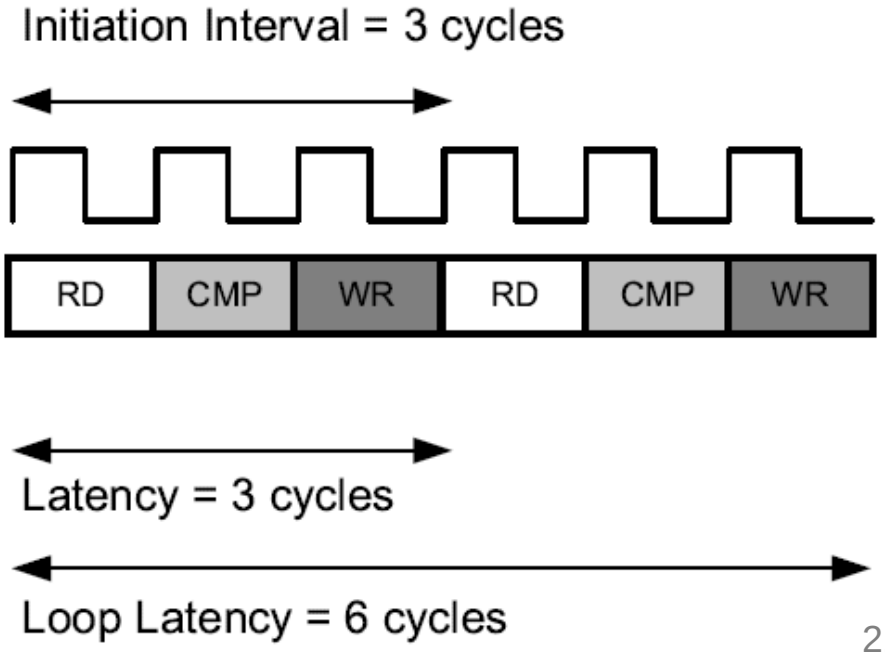
# Outline

- High-Level Synthesis Concepts
  - What is, and Why High-Level Synthesis
  - Design Metrics in HLS
- Vivado High-Level Synthesis
  - Inputs and Outputs
  - High-Level Synthesis Process
    - Interface Synthesis and Algorithm Synthesis
    - Case Studies on Optimizations: Loop and Array
  - Wrap-up: Vivado HLS Design Flow
- Lab Exercise: Accelerating Floating Point Matrix Multiplication with HLS

# Optimizations: Loop

- **Loops** are used extensively in programming, and constitute a natural method of expressing operations that are repetitive in some way.
- By default, Vivado HLS seeks to optimize area: Loops are automatically "rolled" (a.k.a. **rolled loops**).
  - That is, loops time-share a minimal set of hardware.
    - The operations in a loop are executed sequentially.
    - The next iteration can only begin when the last is done.

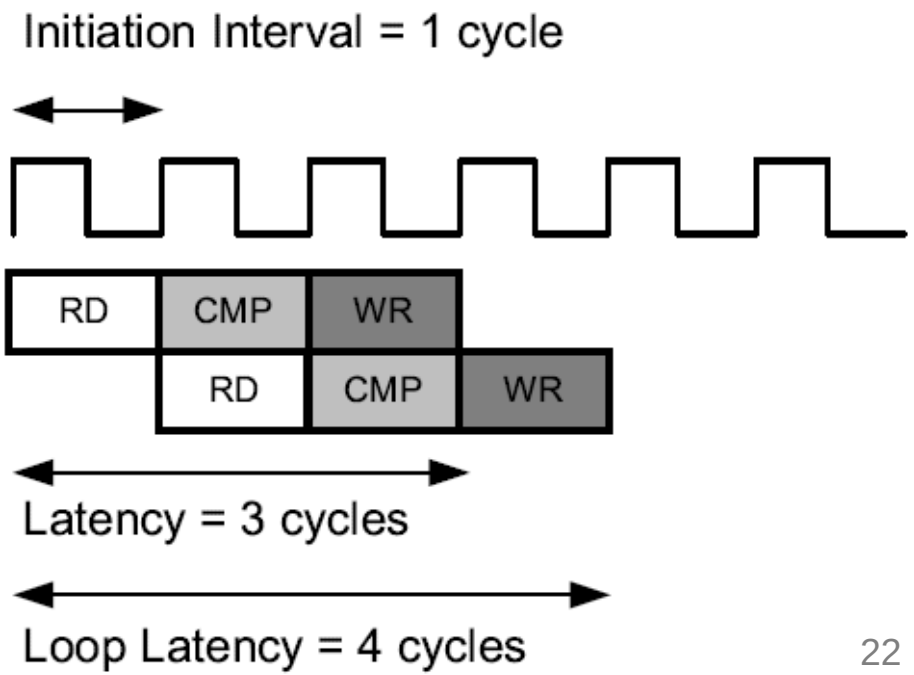
```
Loop:for(i=1;i<3;i++) {
 op_Read;
 op_Compute;
 op_Write;
}
```

# Optimization #1: Loop Pipelining (1/2)

- Several loop optimizations can be made using **directives** in Vivado HLS.
  - Allowing the resulting implementation to be altered with just few or even no changes to the software code.
- **Loop pipelining** allows the operations in a loop to be implemented in a concurrent manner.
  - The initiation interval (II) is the number of clock cycles between the start times of consecutive loop iterations.

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

# Optimization #1: Loop Pipelining (2/2)

- To pipeline a loop, put the directive "`#pragma HLS pipeline [II=1]`" at the beginning of the loop body.
  - Vivado HLS automatically tries to pipeline the loop with the minimum initiation interval (`II`).
  - Without the optional `II=1` , the best possible initiation interval 1 is used, meaning that input samples can be accepted on every clock cycle.

```
for (index_a = 0; index_a < A_NROWS; index_a++) {
     for (index_b = 0; index_b < B_NCOLS; index_b++) {
#pragma HLS PIPELINE II=1
          float result = 0;
          for (index_d = 0; index_d < A_NCOLS; index_d++) {
              float product_term = in_A[index_a][index_d] * in_B[index_d][index_b];
              result += product_term;
          }
          out_C[index_a * B_NCOLS + index_b] = result;
     }
}
```

# Optimization #2: Loop Unrolling (1/2)

- **Loop unrolling** is a technique to exploit parallelism by creating copies of the loop body.
  - Unrolling a loop by a factor of N creates N copies of the loop body, and the loop variable referenced by each copy is updated accordingly.
    - If the factor N is less than the total number of loop iterations (10 in the below example), it is called a "partial unroll".
    - If the factor N is the same as the number of loop iterations, it is called a "full unroll".

**Rolled Loops**

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

# Optimization #2: Loop Unrolling (2/2)

- **Loop unrolling** creates more operations in each loop iteration, resulting higher parallelism and throughput.

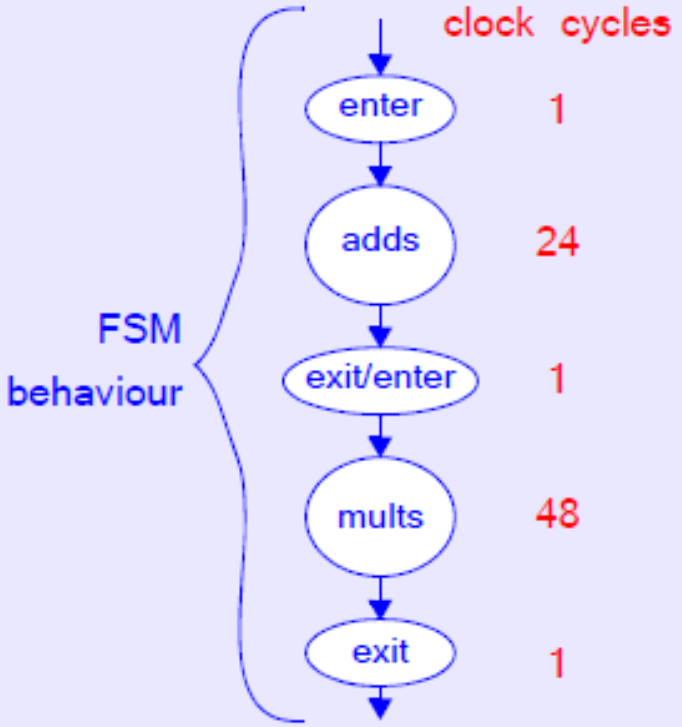
- To unroll a loop, put the directive "`#pragma HLS unroll [factor=N]`" at the beginning of the loop.
  - Without the optional `factor=N`, the loop will be fully unrolled by default.

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

# Optimization #3: Merging Loops (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.

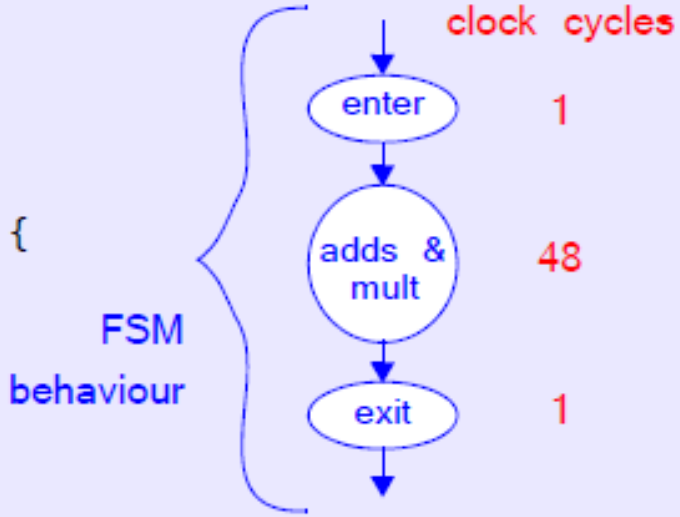
```
void add_mult (short c[12], short m[12], short a[12], short b[12])
{
  short j;


  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];
    }
}
```



# Optimization #3: Merging Loops (2/2)

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

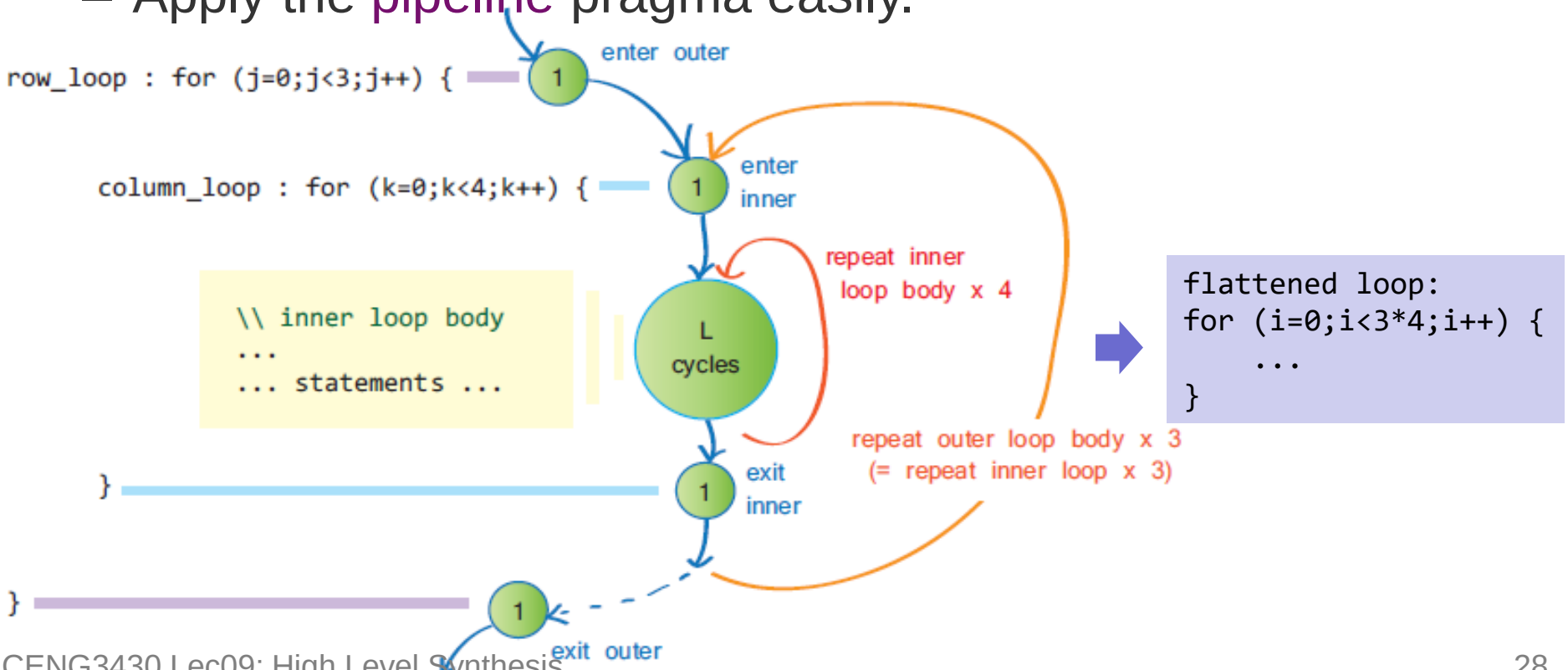
```
void add_mult (short c[12], short m[12], short a[12], short b[12])
{
  short j;


  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.

# Optimization #4: Flattening Loops

- We may "**flatten**" nested loops to unroll inner loop(s) via the directive "`#pragma HLS loop_flatten`".
  - Avoid extra clock cycles transitioning into or out of a loop;
  - Apply larger unrolling parameters to explore the parallelism;
  - Apply the pipeline pragma easily.



# Factors Limiting the Parallelism (1/2)

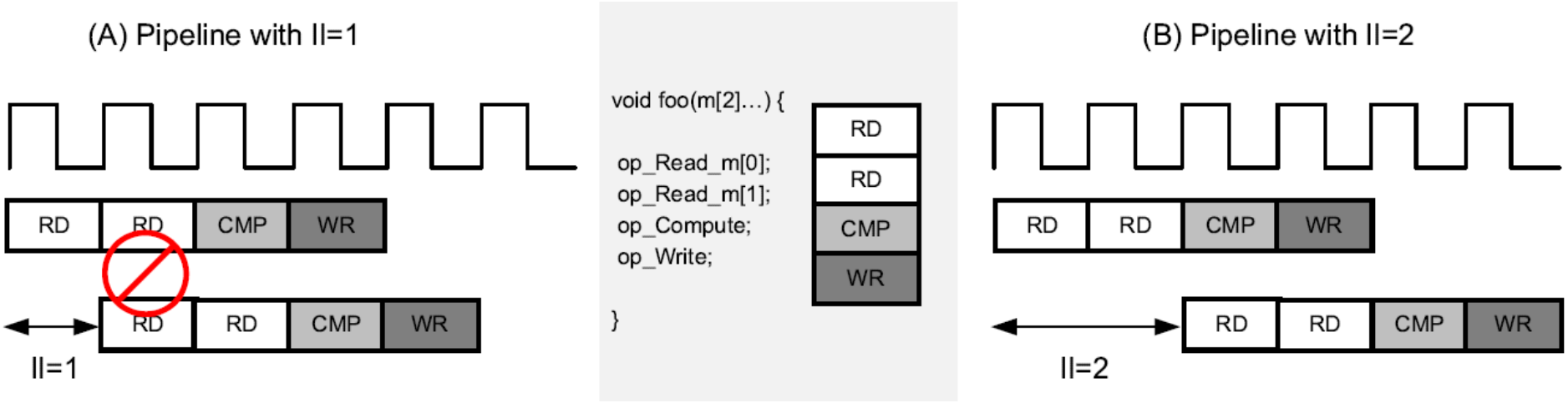
- Loop optimizations aim at exploiting the **parallelism** between loop iterations.
  - However, parallelism between loop iterations can be limited mainly by data dependence or hardware resources.
- **Loop-carried Dependence:** A data dependence from an operation in an iteration to another in a subsequent iteration.
  - The subsequent iteration cannot start until the current iteration has finished.
  - Array accesses are a common source of loop-carried dependences.
  - Automatic dependence analysis can be too conservative: Directive "`#pragma HLS dependence`" allows you to explicitly specify and avoid a false dependence.

```
while (a != b) {
    if (a > b)
        a -= b;
    else
        b -= a;}
```

```
for (i = 1; i < N; i++)
    mem[i] = mem[i-1] + i;
```

# Factors Limiting the Parallelism (2/2)

- Another 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 can only be two.

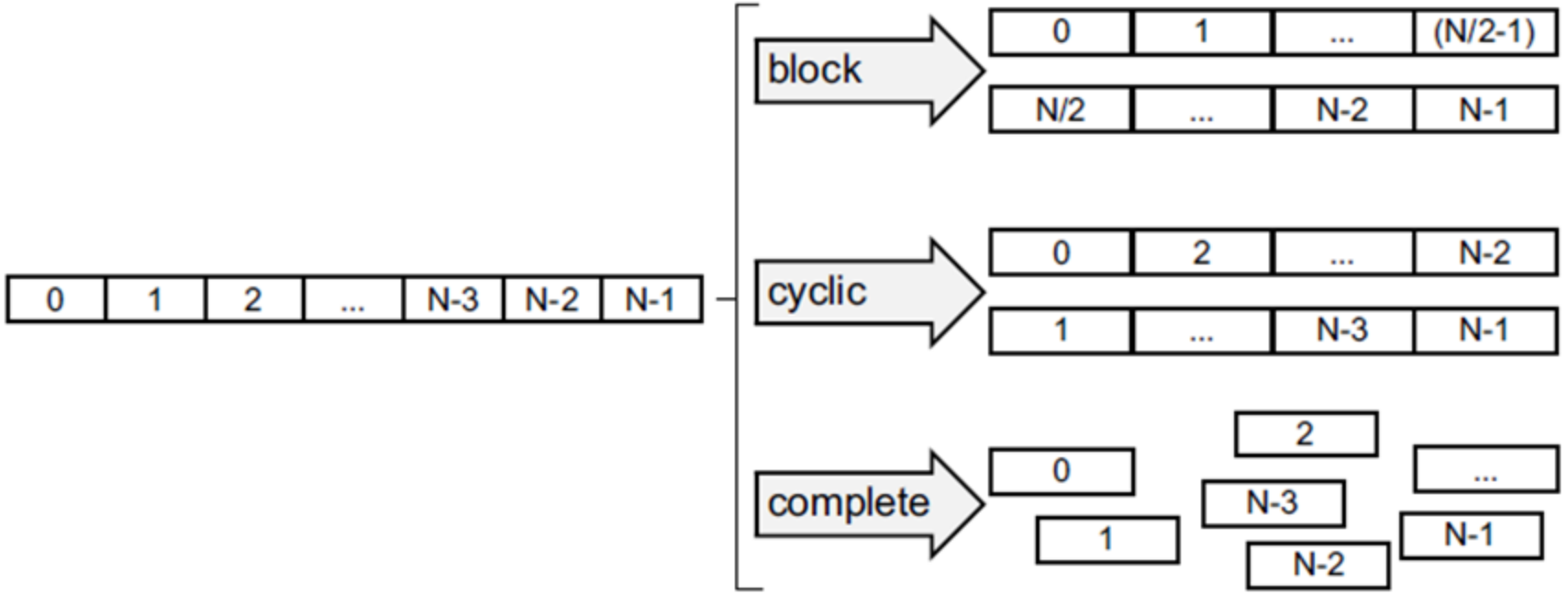


# Optimizations: Array Partition (1/3)

- **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.

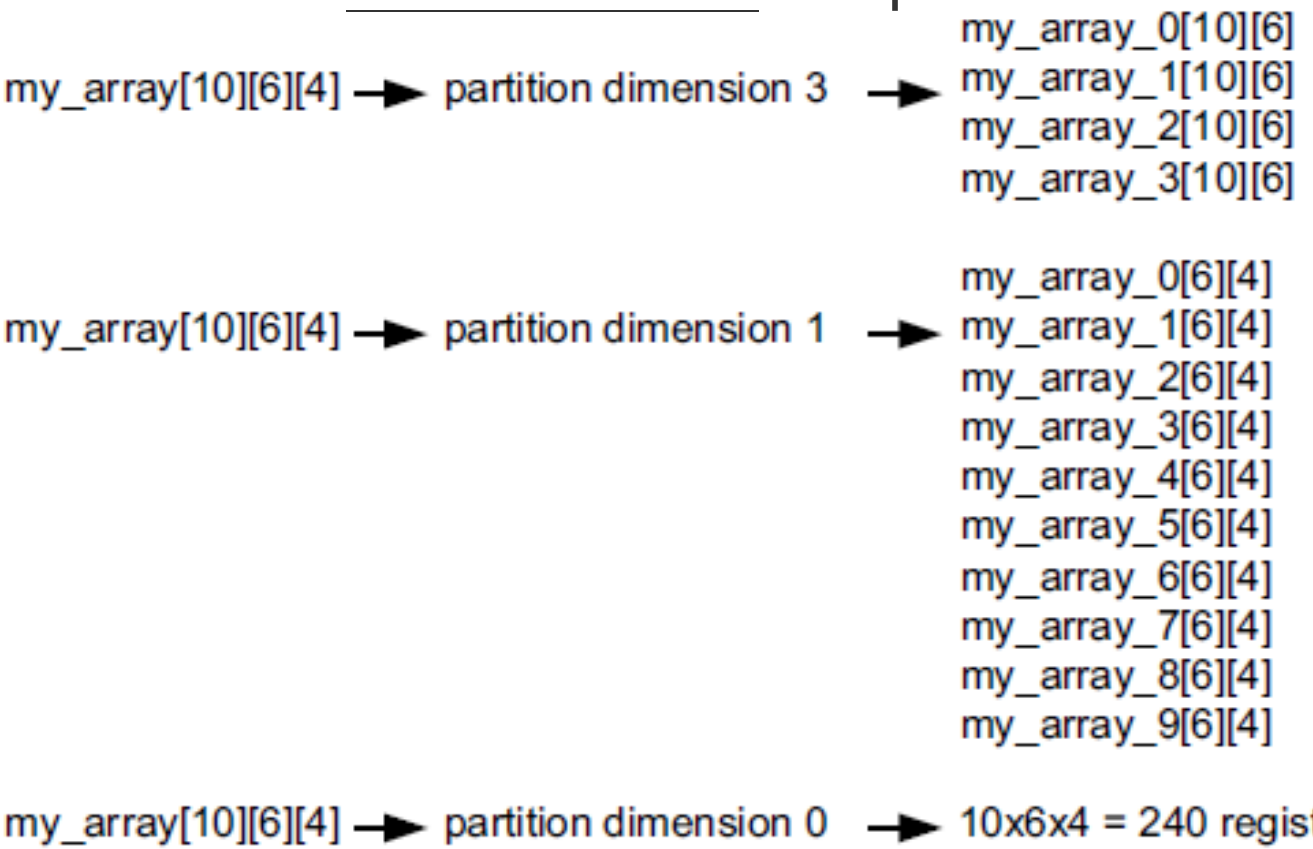
# Optimizations: Array Partition (2/3)

- 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.



# Optimizations: Array Partition (3/3)

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

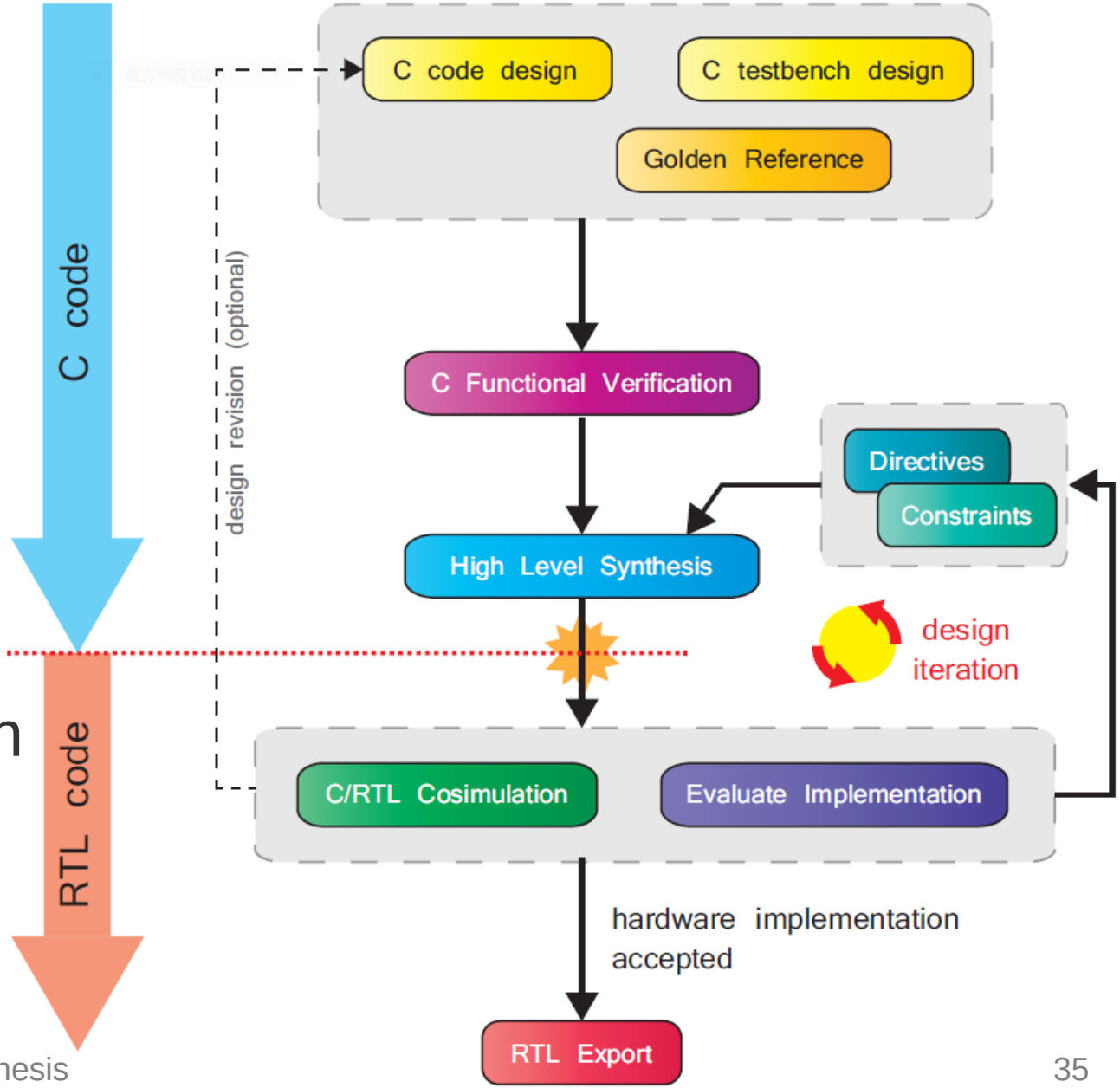


# Outline

- High-Level Synthesis Concepts
  - What is, and Why High-Level Synthesis
  - Design Metrics in HLS
- Vivado High-Level Synthesis
  - Inputs and Outputs
  - High-Level Synthesis Process
    - Interface Synthesis
    - Algorithm Synthesis
    - Algorithm Case Study: Loops
  - Wrap-up: Vivado HLS Design Flow
- Lab Exercise: Accelerating Floating Point Matrix Multiplication with HLS

# Wrap-up: Vivado HLS Design Flow (1/5)

① Inputs to HLS
② Functional Verification
③ High-Level Synthesis
④ C/RTL Co-simulation
⑤ Evaluation of Implementation
⑥ Design Iterations
⑦ RTL Export



# Wrap-up: Vivado HLS Design Flow (2/5)

## ① Inputs to the HLS Process

- A **C/C++/SystemC function**, along with
- A **C-based testbench** for verifying the operation.

## ② Functional Verification

- It is necessary to verify the **functional integrity** of the C/C++/SystemC code before synthesizing it into RTL code.
  - This can be achieved by writing a testbench in the same high-level language, and checking the results against the “golden reference”.

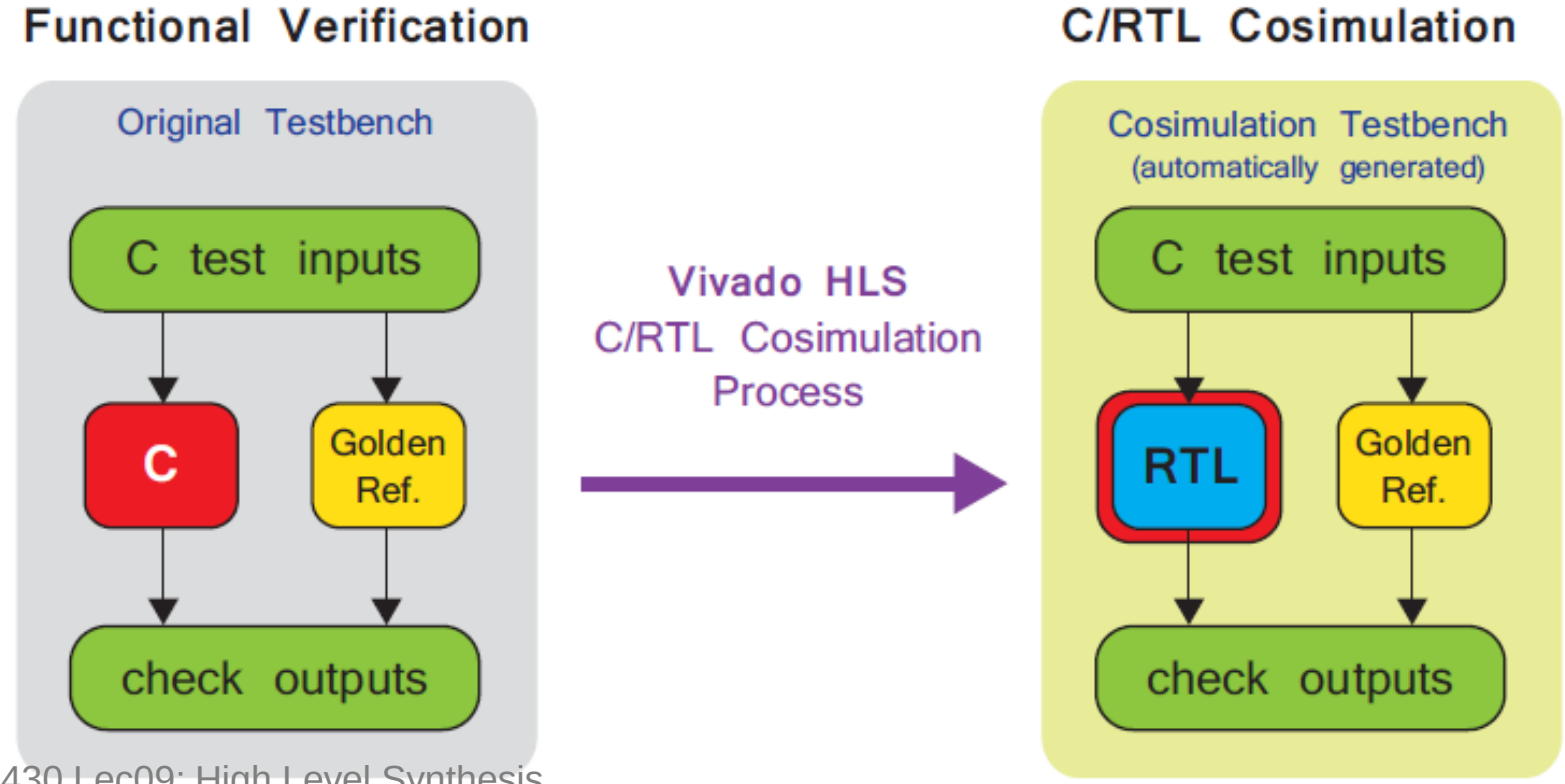
## ③ High-Level Synthesis

- It **transforms** C-based codes, together with user-supplied directives/constraints, into the RTL description of the circuit.
- Once completed, a set of output files is produced.
  - Including design files in the desired RTL language, and other log, report files, testbenches, scripts, etc.

# Wrap-up: Vivado HLS Design Flow (3/5)

④ **C/RTL Co-simulation** (Optional)

- The produced RTL implementation can be **checked** against the original C/C++/SystemC code.
- This process re-uses the original, C-based testbench, saving the effort of generating a new RTL testbench.



# Wrap-up: Vivado HLS Design Flow (4/5)

## ⑤ Evaluation of Implementation

- It is also necessary to **evaluate** the RTL output in terms of its implementation and performance.
  - For example, the numbers of resources it requires in the PL, the latency of the design, maximum supported clock frequency, etc.

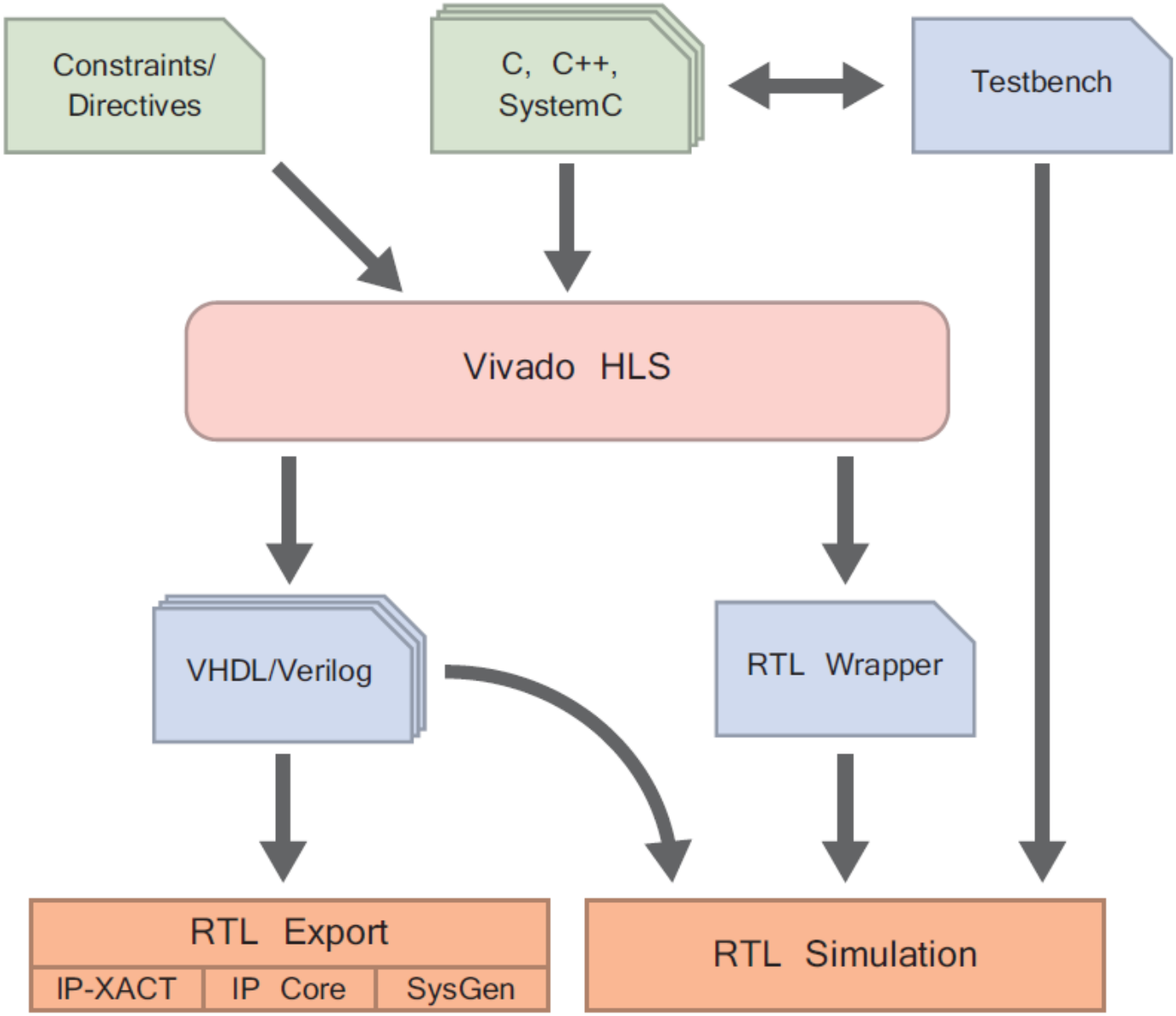
## ⑥ Design Iterations

- The constraints and directives can be **refined** based on the results of implementation evaluation.
- It is also possible to prompt more fundamental review and refinement of original algorithm, as designed in C code.

## ⑦ RTL Export

- The **RTL files** (i.e., VHDL or Verilog codes) can be used directly, or be packaged as an **IP core** for easing the integration with other Xilinx tools, such as IDE and SDK.

# Wrap-up: Vivado HLS Design Flow (5/5)

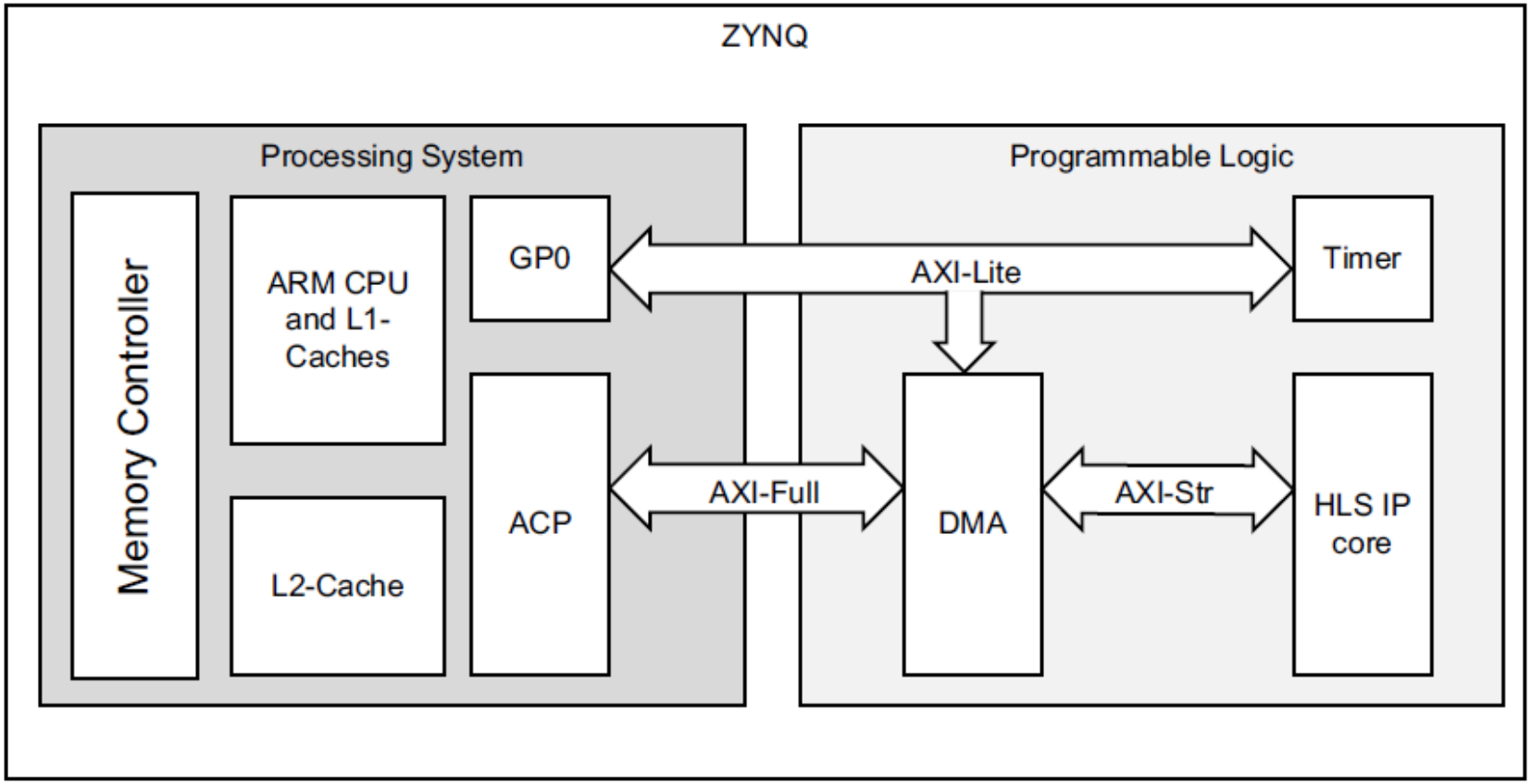


# Outline

- High-Level Synthesis Concepts
  - What is, and Why High-Level Synthesis
  - Design Metrics in HLS
- Vivado High-Level Synthesis
  - Inputs and Outputs
  - High-Level Synthesis Process
    - Interface Synthesis
    - Algorithm Synthesis
    - Algorithm Case Study: Loops
  - Wrap-up: Vivado HLS Design Flow
- 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 to? Simply put 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 |

The higher, the worse!

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 |

Latency (clock cycles)

Summary

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

Detail

Instance

Loop

The higher, the worse!

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 worse!

# Summary

- High-Level Synthesis Concepts
  - What is, and Why High-Level Synthesis
  - Design Metrics in HLS
- Vivado High-Level Synthesis
  - Inputs and Outputs
  - High-Level Synthesis Process
    - Interface Synthesis
    - Algorithm Synthesis
    - Algorithm Case Study: Loops
  - Wrap-up: Vivado HLS Design Flow
- Lab Exercise: Accelerating Floating Point Matrix Multiplication with HLS