AMD
XILINX

# Introduction to High-Level Synthesis with Vitis HLS Tool

# Objectives

- After completing this module, you will be able to:
  - Describe the need for high-level synthesis
  - Describe the efficient paradigms for programming FPGAs
  - Identify the basic terminology used in HLS
  - Identify the steps to extract RTL from C using the Vitis HLS tool
  - Perform C language support for the Vitis HLS tool
  - Describe the C validation and RTL verification process in the Vitis HLS tool

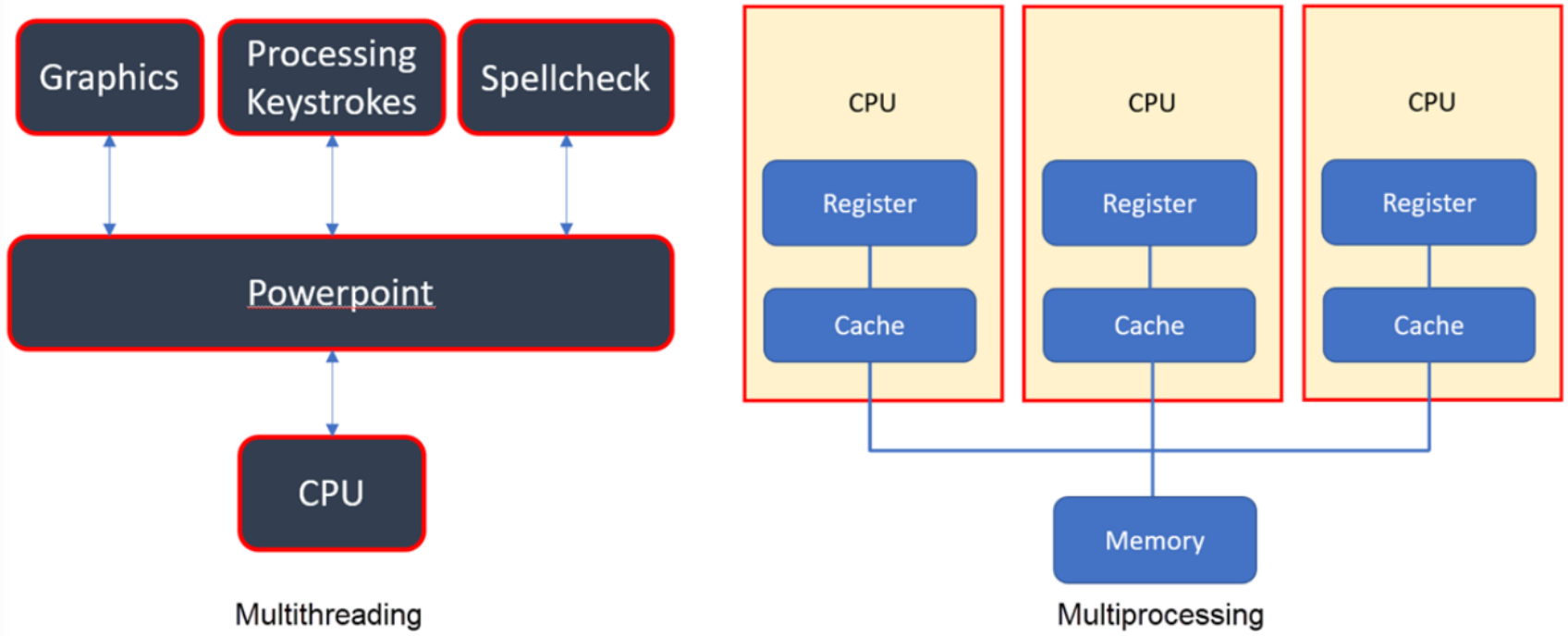
# Outline

- Three Paradigms for Programming FPGAs
- HLS Design Flow
- Basics of High-Level Synthesis
- High-Level Synthesis with Vitis HLS
- Validation and Verification Flow
- Vitis HLS Support
- Summary

# Three Paradigms for Programming FPGAs

# Architecture Matters

- Multithreading and/or Multiprocessing can include multiple system processes, or it can consist of one process that has multiple threads within it.
- Multi-threaded programming using a shared memory system became very popular as it allowed the software developer to design applications with parallelism in mind but with a fixed CPU architecture.
- Multi-processing programming using multiple CPU cores and hyperthreading to improve throughput as shown in the figure on the right.

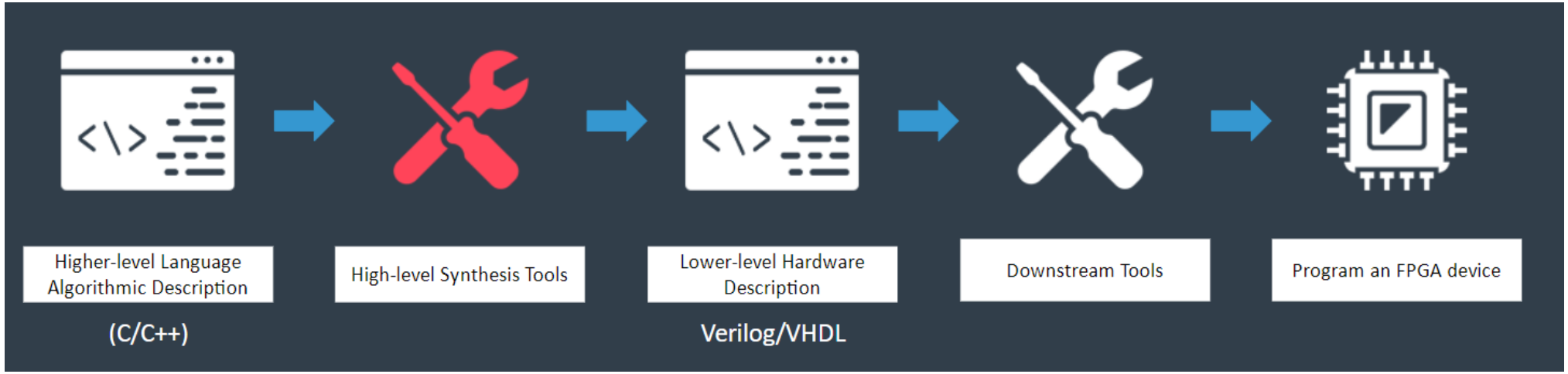







AMD XILINX

# Three Paradigms for Programming FPGAs



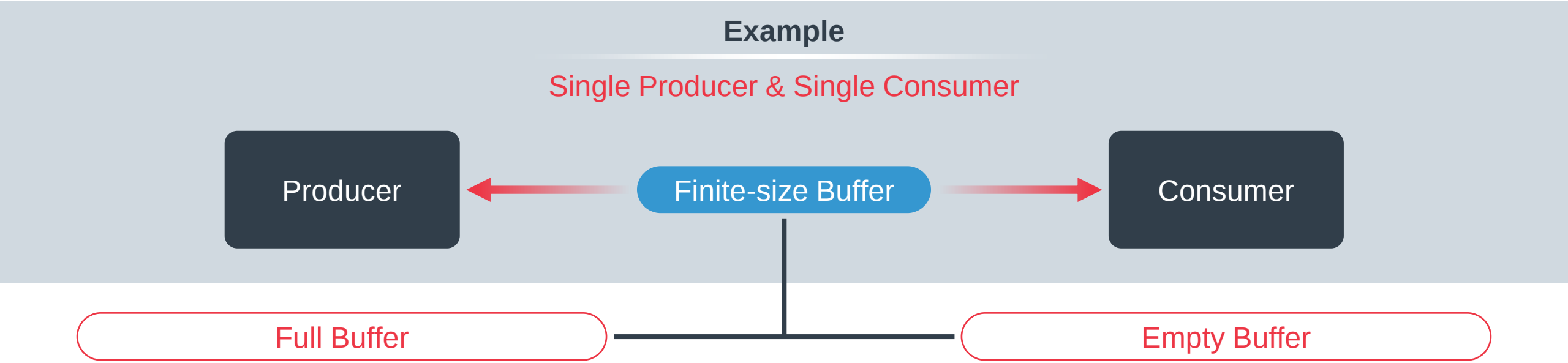

- Retain the advantages of a programming language to write efficient code that can be translated into hardware
- Additional work (rewriting the code) require for the desired performance goals even if the C/C++ code can be automatically converted into hardware.

Producer-Consumer Paradigm

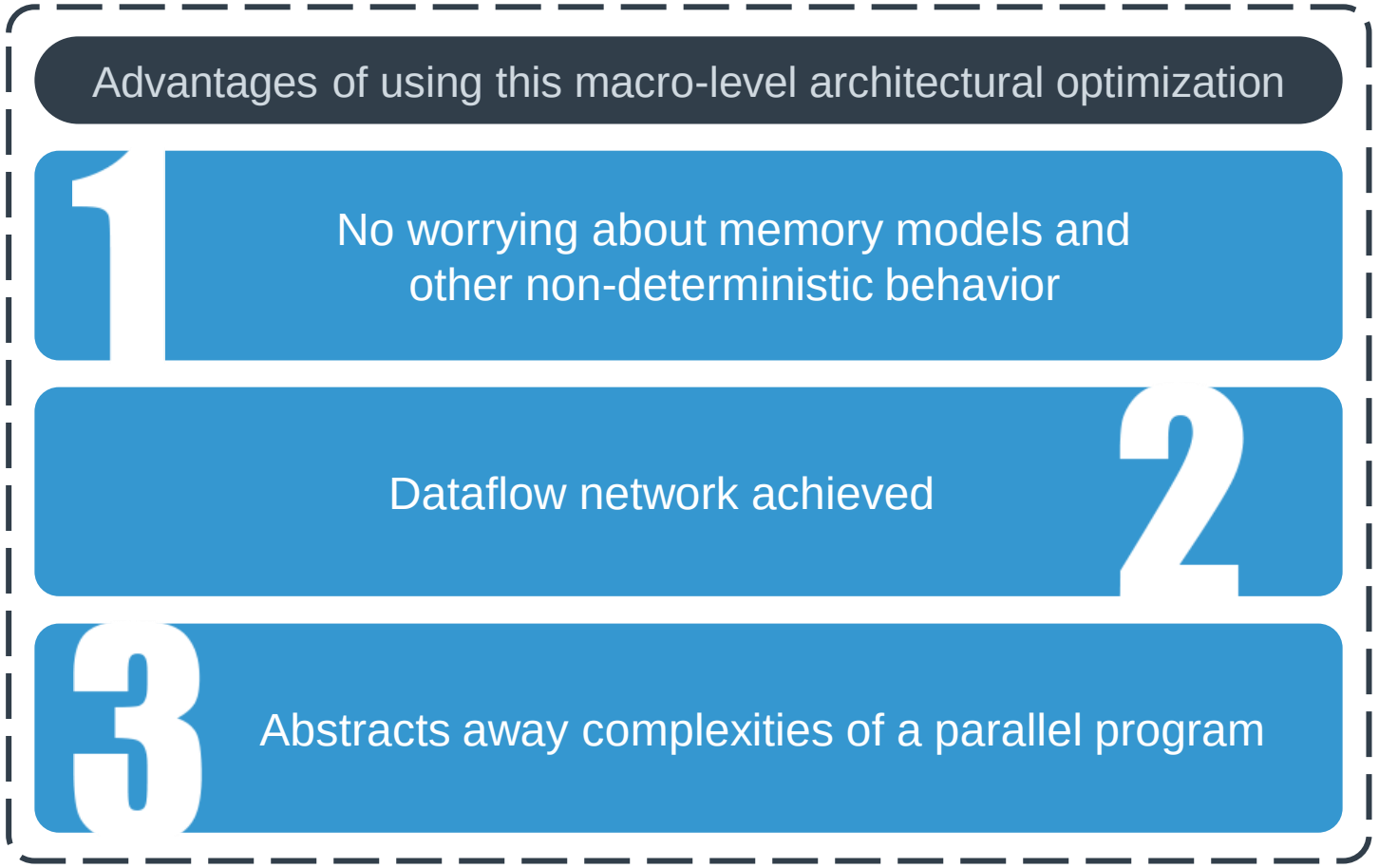
Stream data Paradigm

Pipelining Paradigm

# Producer-Consumer Paradigm

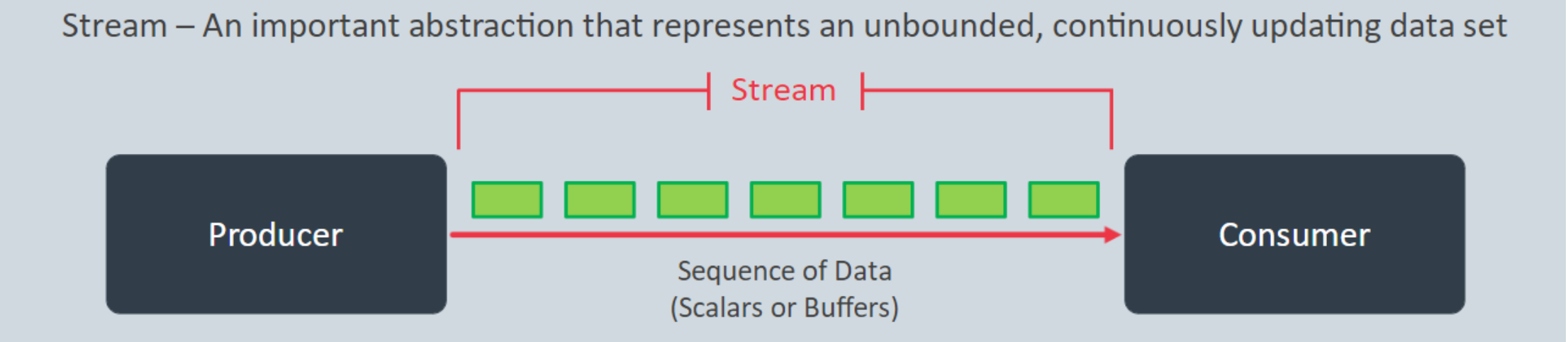


# Producer-Consumer Paradigm




Performance of such a dataflow network relies on the designer's ability to continually feed data to the network such that data keeps streaming through the system

# Stream data Paradigm

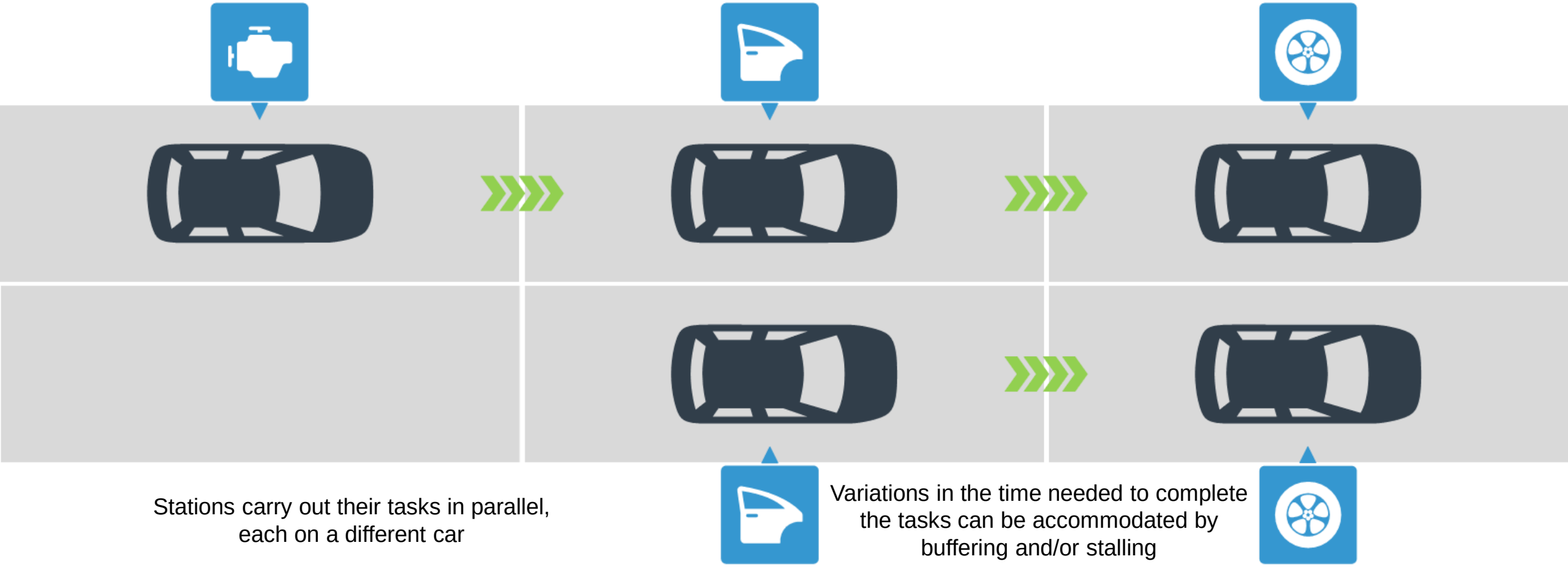



- Streaming paradigm forces you to think in terms of data access patterns/sequences
- Random memory access to data in software are virtually free; in hardware, it takes resources
- Make sequential memory accesses that can be converted into streams

# Pipelining Paradigm

A classical micro-level architectural optimization that can be applied to multiple levels of abstraction

**Example – Production line of a car factory**



Stations carry out their tasks in parallel, each on a different car

Variations in the time needed to complete the tasks can be accommodated by buffering and/or stalling

Divide the work and resources among the stages so all take the same time to complete their tasks





AMD XILINX

# Pipelining Paradigm

Producer-consumer pipeline will only be efficient if each task produces/consumes data at a high rate

**Static Optimization**

Pipelining uses same resources to execute the same function over time and requires complete knowledge about the number of times the task is executed and the latency of each task

Low-level instruction pipelining technique cannot be applied to dataflow-type networks

# Pipelining Paradigm

Software written for CPUs is fundamentally different from software written for FPGAs

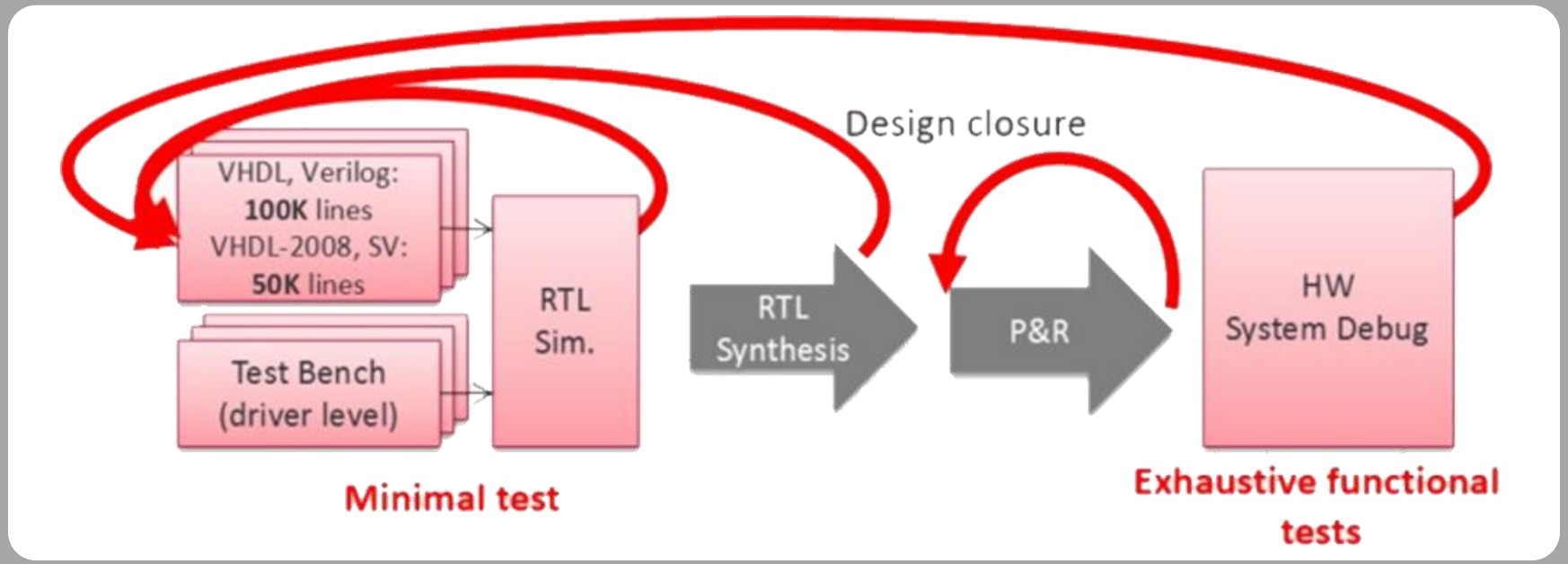
Recommended high-level actions

- Verify the source code changes
- Focus on the macro-architecture and draw the desired activity timeline where the horizontal axis represents time
- Start coding your program
- Partition the original algorithm into smaller components
- Have an overall vision about the rates of processing
- Think about the granularity of the streaming

# HLS Design Flow

# Traditional RTL vs Vitis HLS Design Flow

## Traditional RTL Flow



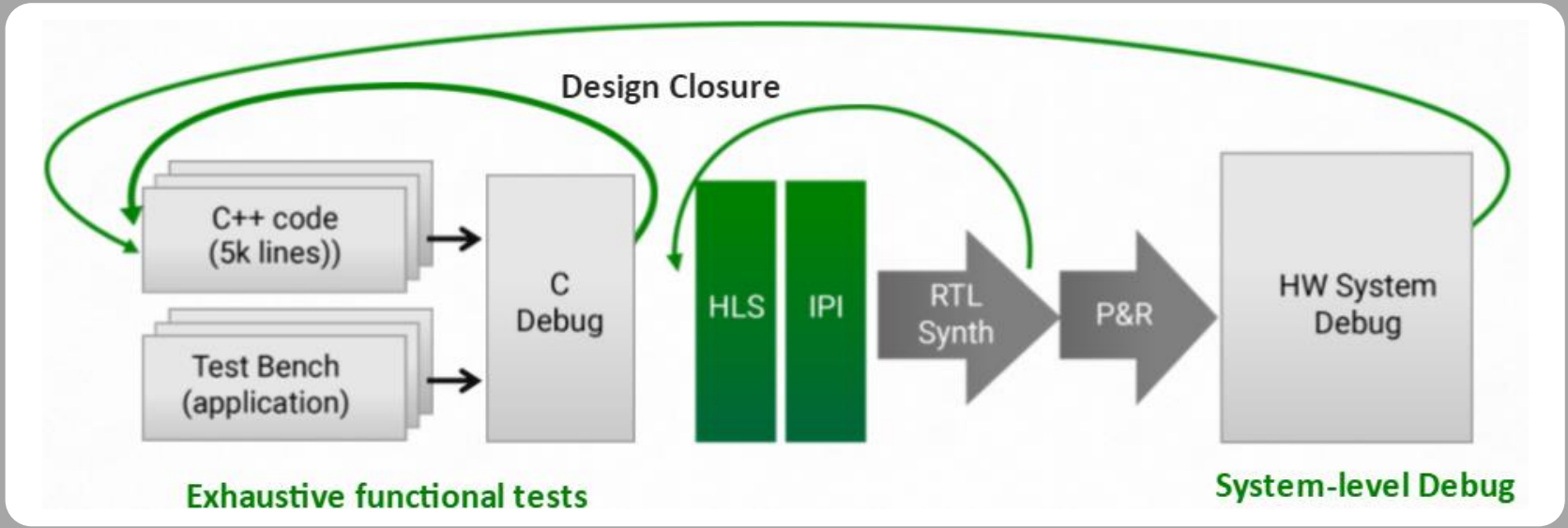

Typical system starts with a software model → Distributed to the hardware and embedded software teams

Choose an RTL microarchitecture that meets the system requirements

End-product has orders of magnitude better performance per watt

# Traditional RTL vs Vitis HLS Design Flow

## Vitis HLS Tool Design Flow




When coupled with the Vivado IP integrator:

- HLS tool provides designers and system architects with a **faster and more robust way of delivering quality designs**

**Traditional Flow Example**

- **240 people months**
  - 10 people
  - 2 years

**HLS-Based Flow Example**

- **16 people months**
  - 2 people
  - 8 months

15x Faster





AMD XILINX

# Need for High-Level Synthesis

- Algorithmic-based approaches are getting popular due to accelerated design time and time to market (TTM)
  - Larger designs pose challenges in design and verification of hardware at HDL level
- Industry trend is moving towards hardware acceleration to enhance performance and productivity
  - CPU-intensive tasks can be offloaded to hardware accelerator in FPGA
  - Hardware accelerators require a lot of time to understand and design
- Vitis HLS tool converts algorithmic description written in C-based design flow into hardware description (RTL)
  - Elevates the abstraction level from RTL to algorithms
- High-level synthesis is essential for maintaining design productivity for large designs

# Vitis HLS Tool Flow

Inputs to the HLS tool:

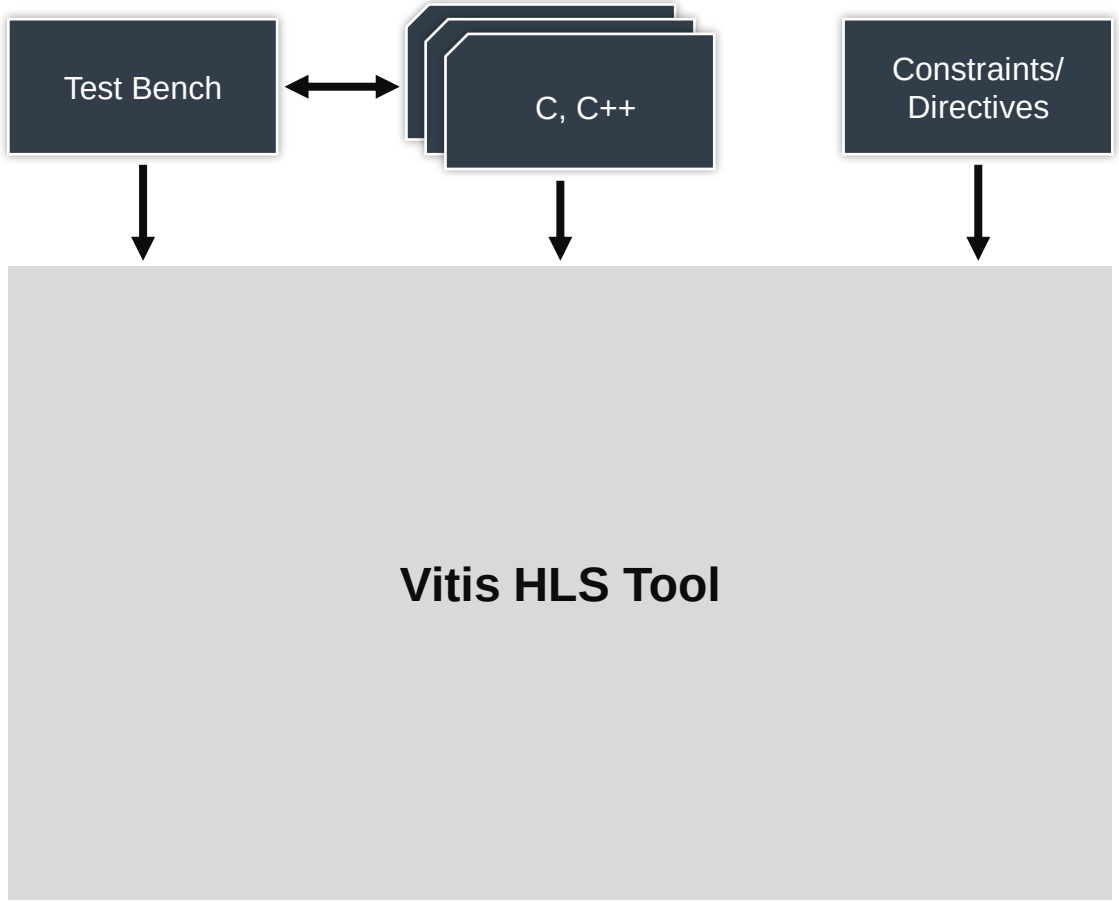
01 C, C++ functions with their sub-functions

02 C functions with RTL blackbox content

03 Constraints: clock period, clock uncertainty, device target, etc.

04 Directives that direct the synthesis process to implement a specific behavior or optimization

05 C test bench used to simulate the C function prior to synthesis

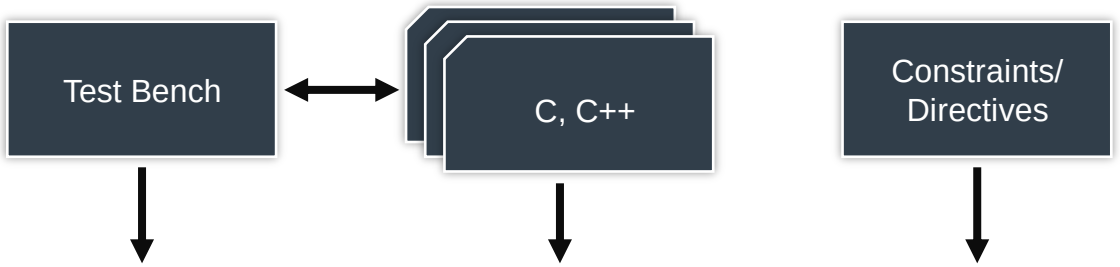







AMD XILINX

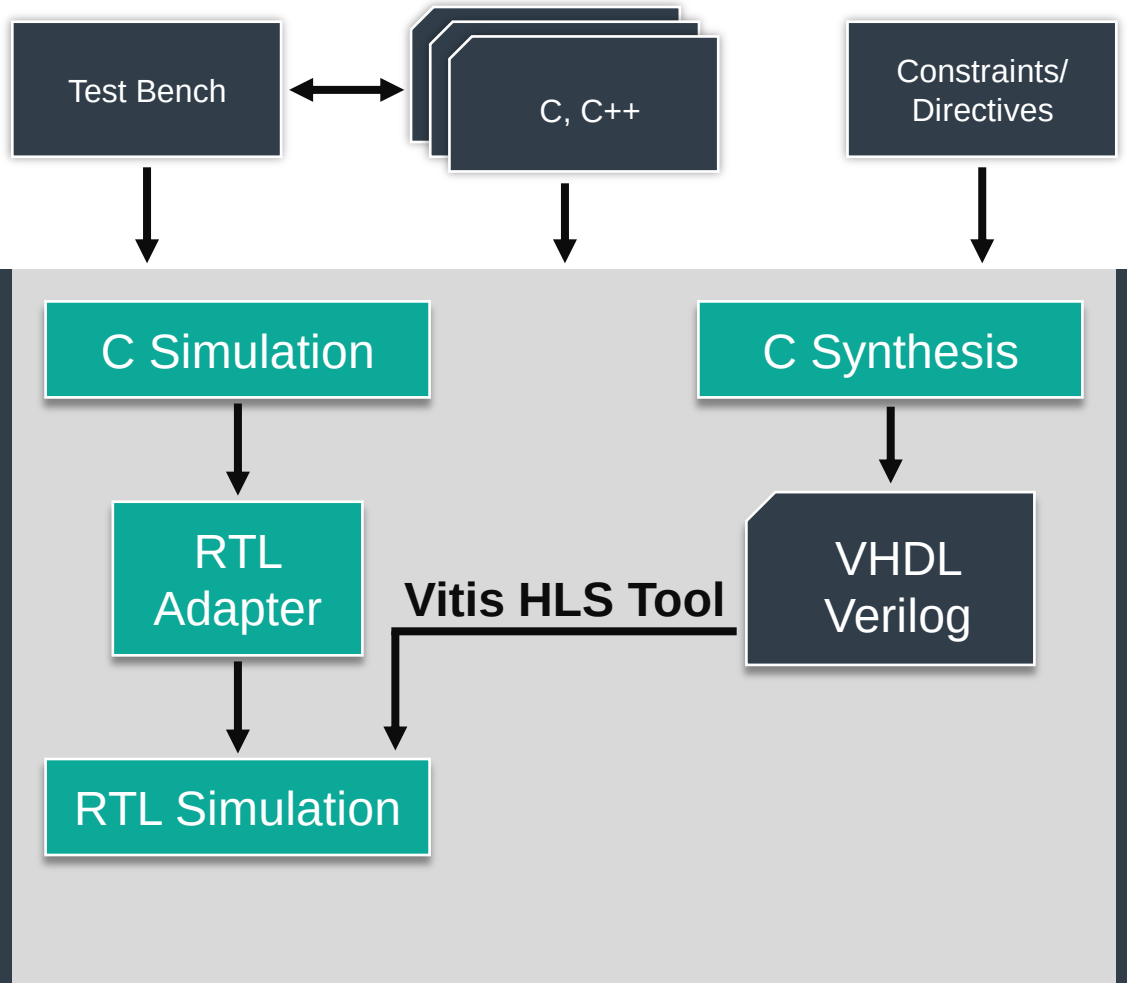
# Vitis HLS Tool Flow




**Process:**

- First step is to verify the functionality of the C/C++ code using C simulation feature
- Next – C synthesis, which includes:
  - Scheduling of all the tasks
  - Mapping of the available resources
  - Optimizing the code using synthesis directives
  - Generating the RTL as an output

# Vitis HLS Tool Flow




Once the RTL is generated, it is validated using the RTL Co-simulation feature

**Output of the HLS tool:**

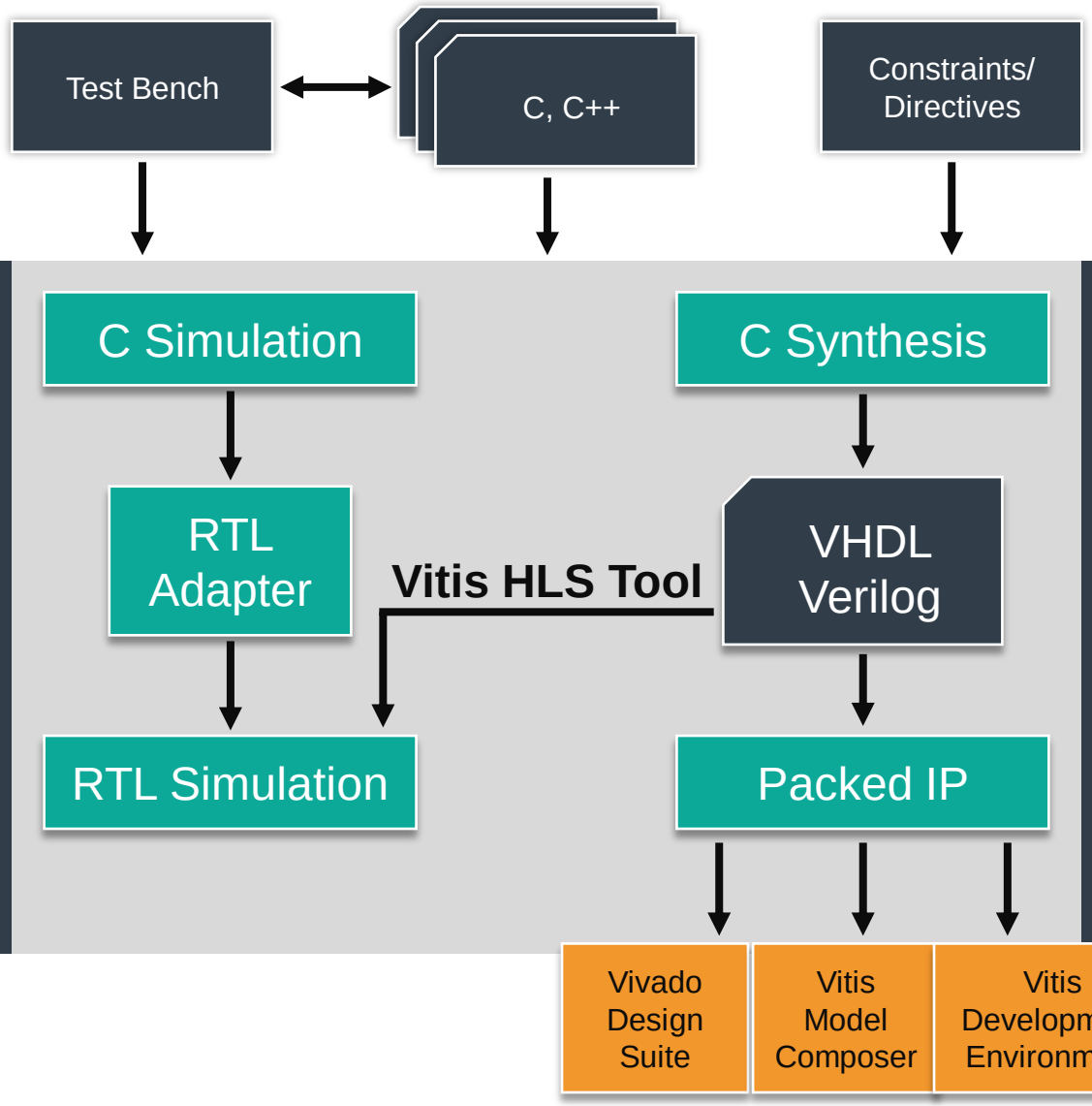
- **Compiled object files (.xo):** Created compiled hardware functions for use in the Vitis application acceleration development flow
- **RTL implementation files in HDL formats**: Primary output from Vitis HLS tool flow
- **Report files:** Generated as a result of simulation, synthesis, C/RTL co-simulation, and generating output

# Vitis HLS Tool Flow

HLS tool packages these implementation files as an IP block

Exports it to IP catalog for use with other tools in the Xilinx design flow such as:

- Vivado Design Suite
- Vitis Model Composer
- Vitis Development Environment







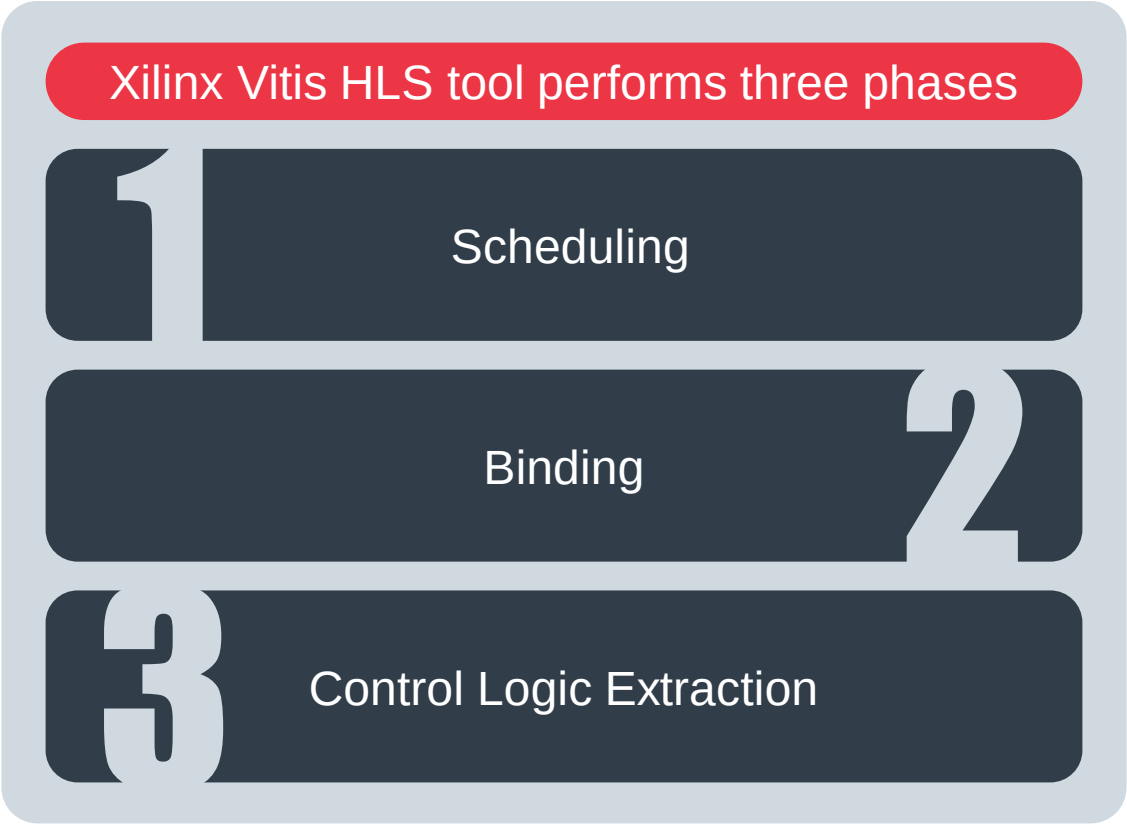

AMD XILINX

# Basics of High-Level Synthesis

# Basics of High-Level Synthesis

Vitis HLS tool

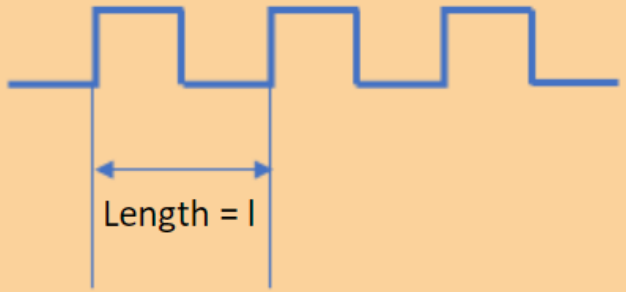
- Allows C, C++, and OpenCL™ functions to become hard wired onto the device logic fabric and RAM/DSP blocks
- Implements hardware kernels in the Vitis application acceleration development flow and develops RTL IP for FPGA designs in Vivado® Design Suite



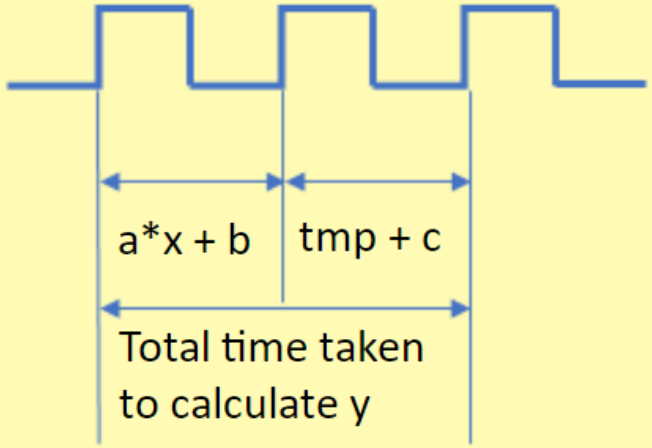
# Scheduling

- Scheduling determines which operations occur during each clock cycle based on :

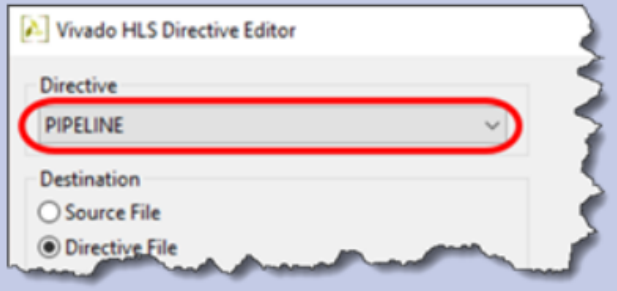
Length of the clock cycle or clock frequency




Time it takes for the operation to complete as defined by the target device




User-specified optimization directives

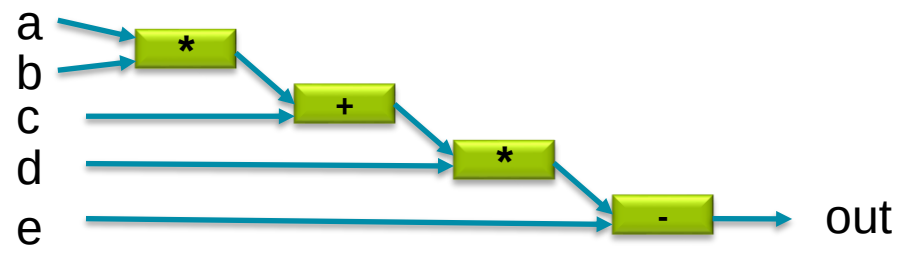



Different user-specified directives like pipeline, dataflow etc.

# Scheduling

- If the clock period is shorter or a slower FPGA is targeted
  - HLS automatically schedules the operations over more clock cycles as well as some operations might need to be implemented as multicycle resources

```
void foo (
 …
  t1 = a * b;
  t2 = c + t1;
  t3 = d * t2;
  out = t3 – e;
}
```




Schedule 1


- If the clock period is longer or a faster FPGA is targeted
  - More operations are completed within a singer clock cycle

Schedule 2


- The code also impacts the schedule on the contrary
  - Code implications and data dependencies must be obeyed

# Binding

- Binding phase determines which hardware resources implements each scheduled operation
  - Operators map to cores

- Binding Decision: to share
  - Given this schedule:

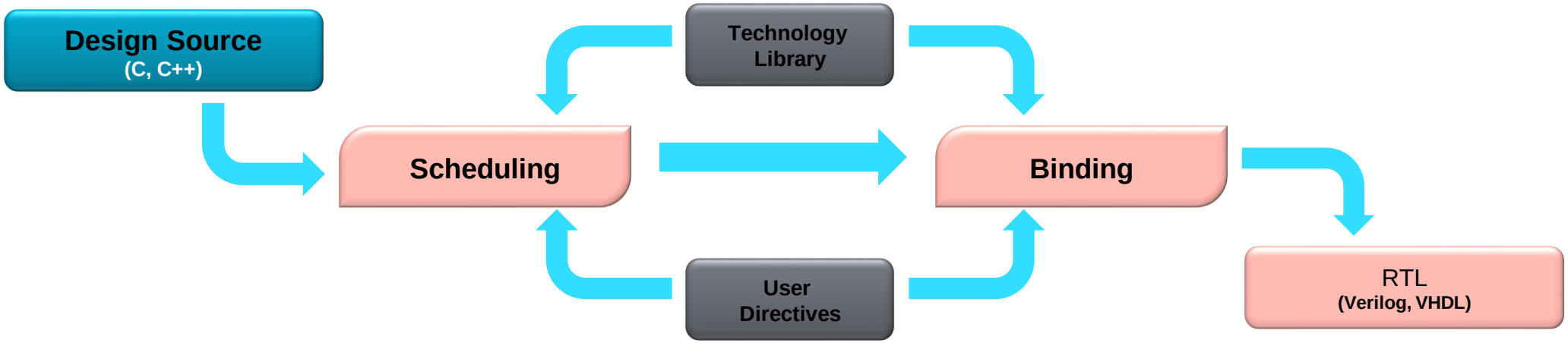

    - Binding must use 2 multipliers, since both are in the same cycle
    - It can decide to use an adder and subtractor or *share* one addsub

- Binding Decision: or not to share
  - Given this schedule:


    - Binding may decide to share the multipliers (each is used in a different cycle)
    - Or it may decide the cost of sharing (muxing) would impact timing and it may decide *not to share* them
    - It may make this same decision in the first example above too

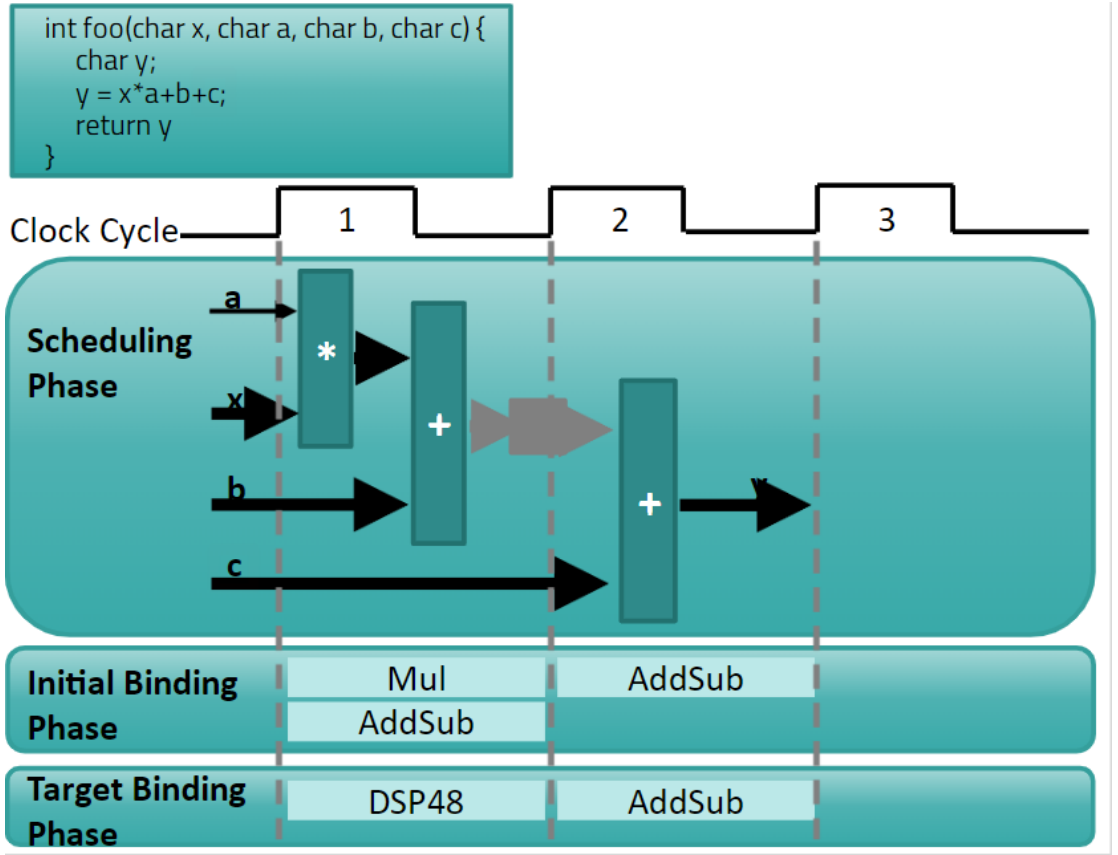
# Scheduling and Binding

- Scheduling & Binding
  - Scheduling and Binding are at the heart of HLS
- Scheduling determines in which clock cycle an operation will occur
  - Takes into account the control, dataflow and user directives
  - The allocation of resources can be constrained
- Binding determines which library cell is used for each operation
  - Takes into account component delays, user directives

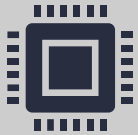


# Scheduling and Binding Example

- Scheduling Phase:
  - Scheduling of the operations during each clock cycle
  - Multiplication and first addition performed in the first clock cycle
  - An internal register stores this result
  - Second addition and the output generation happen in the second cycle
- Initial binding phase:
  - Implements the multiplier operation using a combinational multiplier (Mul)
  - Implements both add operations using a combinational adder/subtractor(AddSub)
- Target binding phase:
  - Implements both the multiplier and one of the addition operations using a DSP48 resource
- Final hardware implementation:
  - Implements the arguments to the top-level functions as input and output ports



# Control Logic Extraction



Extracts the control logic to create a finite state machine that sequences the operations in the RTL design

Why was the control logic extractions not happening in the HLS tool in the scheduling and binding example?

There were no loops!

Differences between combinational circuit and sequential circuit to be implemented into FPGAs

# Control Logic Extraction & I/O Port Implementation Example

Array in the Top-Level Function → Block RAM

Other Options → FIFOs, Distributed RAM, Individual Registers

HLS assumes that the block RAM is outside the top-level function and automatically creates ports such as:

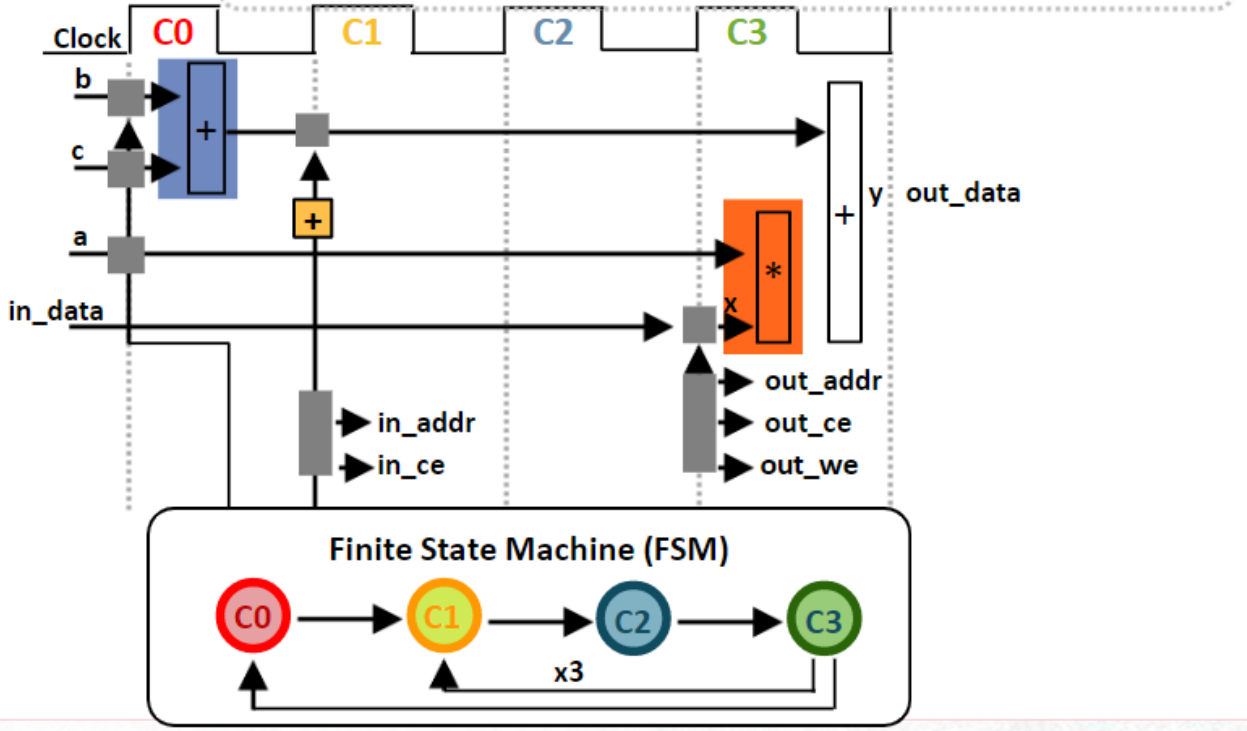
- Data ports
- Address ports
- Chip-enable
- Write-enable

# Control Logic Extraction

- Example performs the same multiplication and addition operations but inside a ‘for’ loop
  - HLS automatically extract the control logic from the C code and creates a Finite State Machine(FSM) in the RTL design to sequence these operations
  - FSM controls when the registers stores data and the state of any I/O control signals.
  - It starts in the state C0. On the next clock, it enters state C1, then state C2, and then C3.
  - Each time the design enters state C3, it reuses the result of the addition
  - Process continues until all output is written
  - Design then returns to the state C0 to read the next values of b and c to start the process again

```
void foo (int in [3], char a, char b, char c, int out [3]) {
  int x,y;
  for (int i = 0; i < 3; i++) {
    x = in[ i ];
    y = a*x + b + c;
    out [ i ] = y;
  }
}
```

C0 - adder b + c
C1 - generates the address for *in* and *adder* to increment to count how many times design iterates {C1, C2 and C3}
C2 - Block Ram returns the data for *in* and stores it as variable *x*
C3 - Reads the data from port *a* with other values to perform the calculation and generates the first *y* out



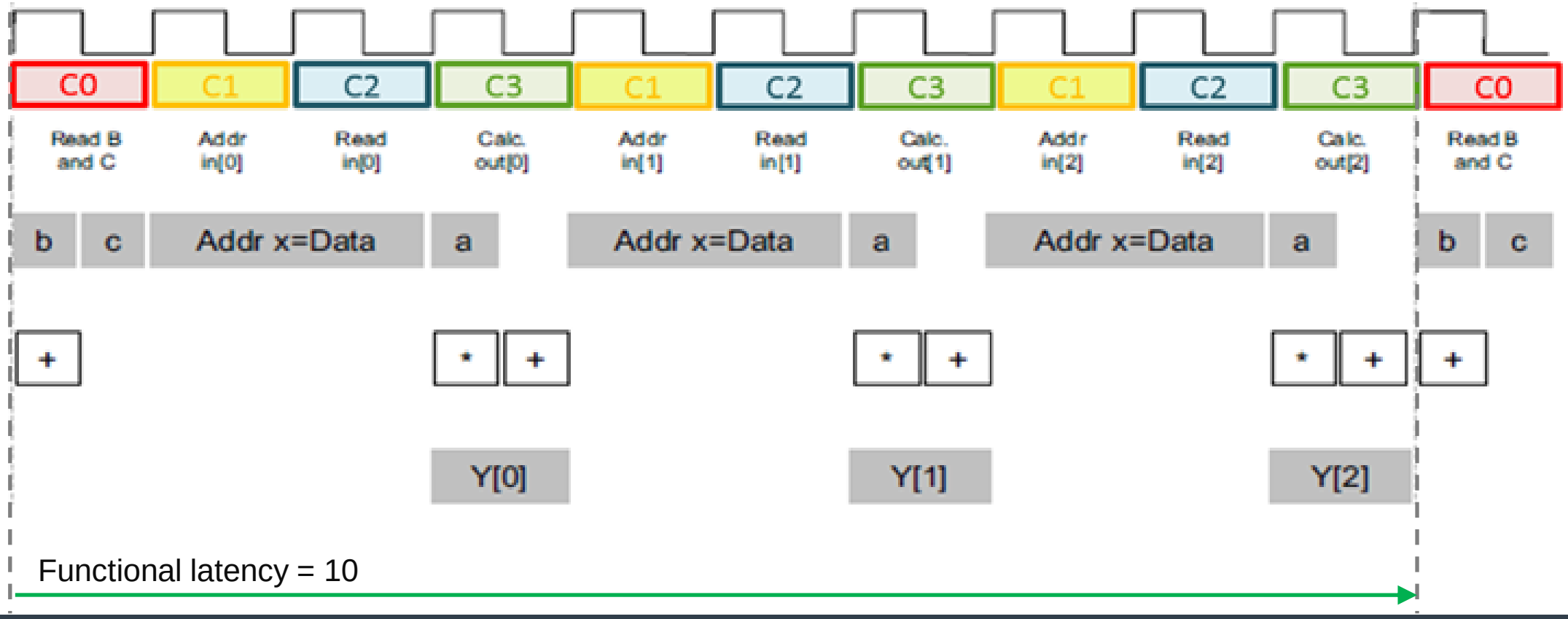





AMD XILINX

# High-Level Synthesis with Vitis HLS tool

# Terminology for Measuring in Clock Cycles

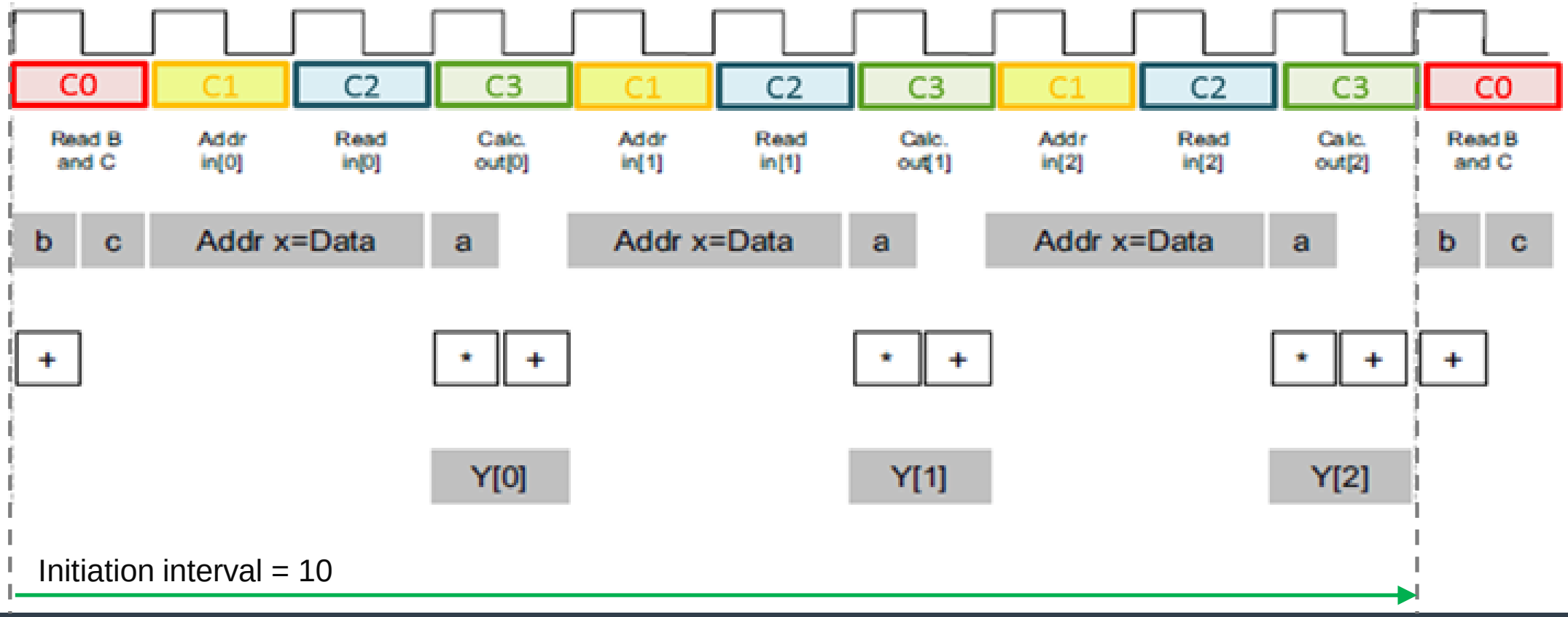



**Latency:** Number of clock cycles required for the function to go from input to output generation

When the output is an array, the latency is measured to the last array value output

- Ten clock cycles in this case

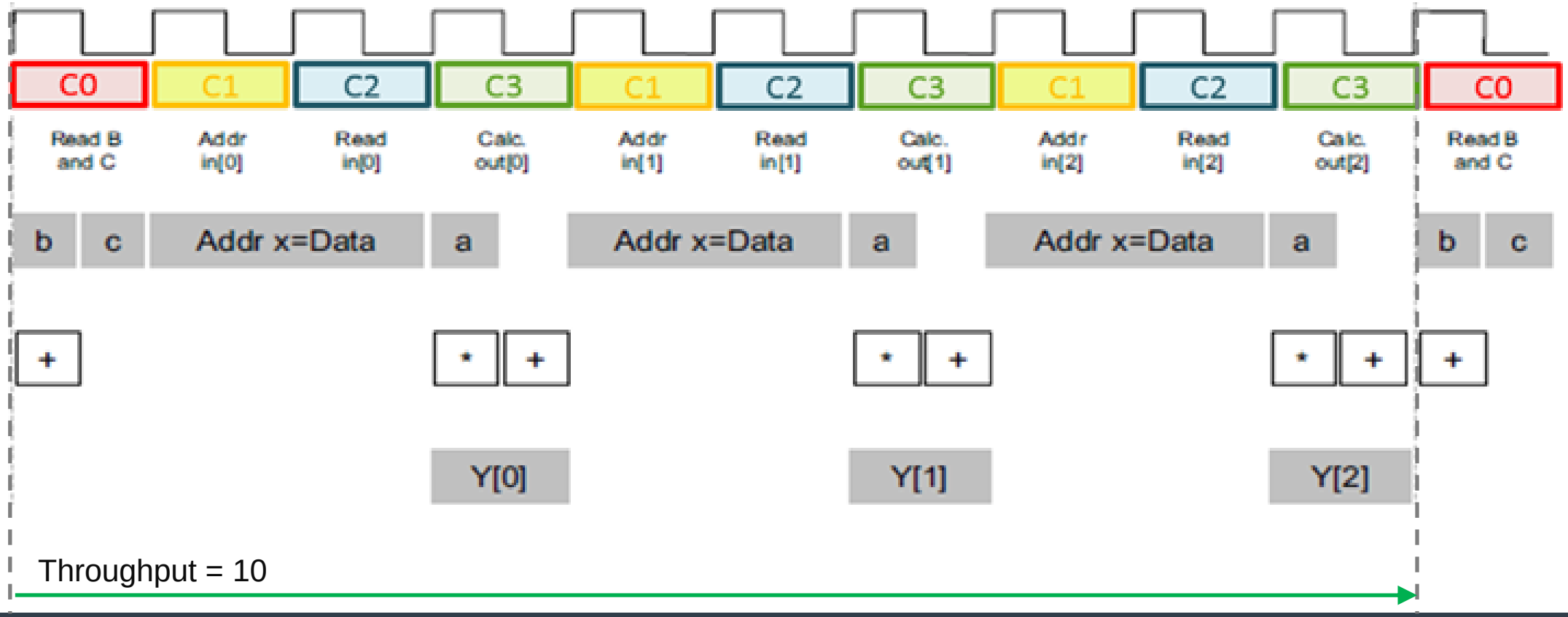
# Terminology for Measuring in Clock Cycles




**Initiation interval (II):** Number of clock cycles before the function can accept new input data

- 10 clock cycles in this case

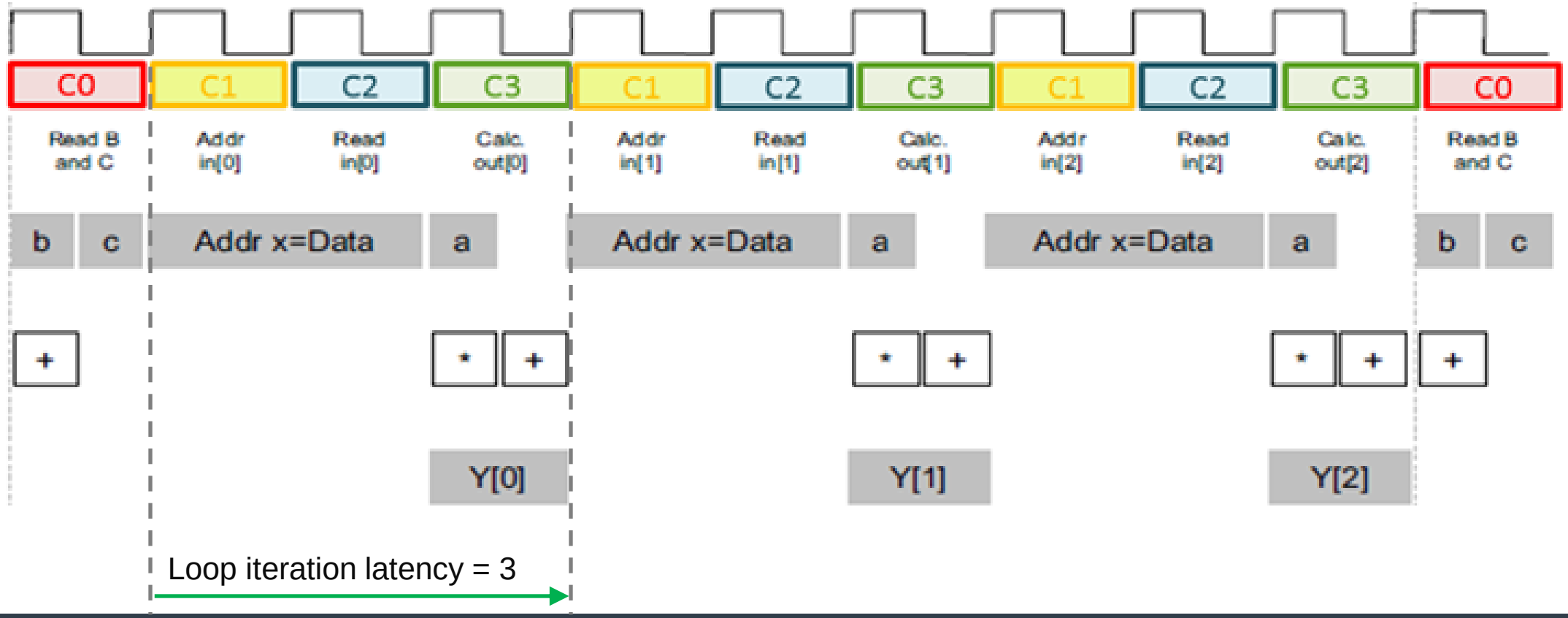
# Terminology for Measuring in Clock Cycles




**Throughput:** Number of cycles between the new input samples

- 10 clock cycles in this case

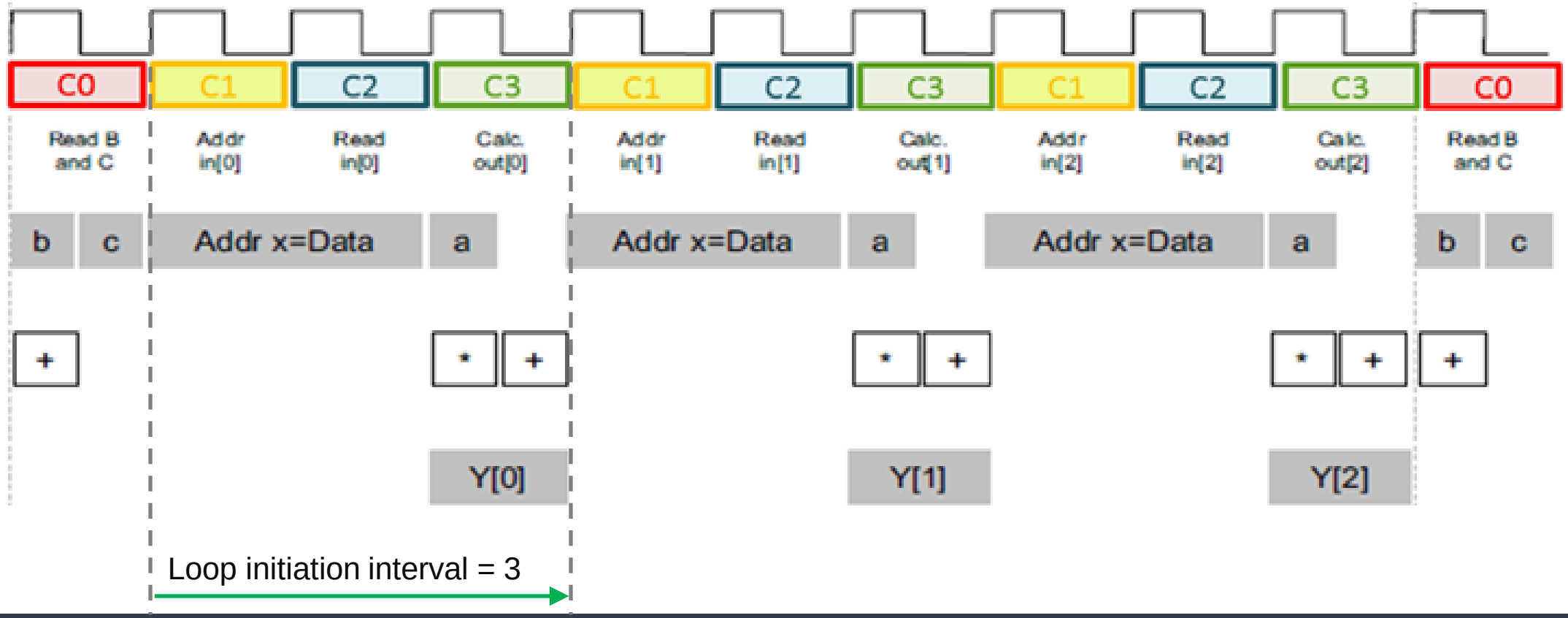
# Terminology for Measuring in Clock Cycles




**Loop iteration latency:** It is the number of clock cycles it takes to complete one iteration of the loop

- Three clock cycles in this case

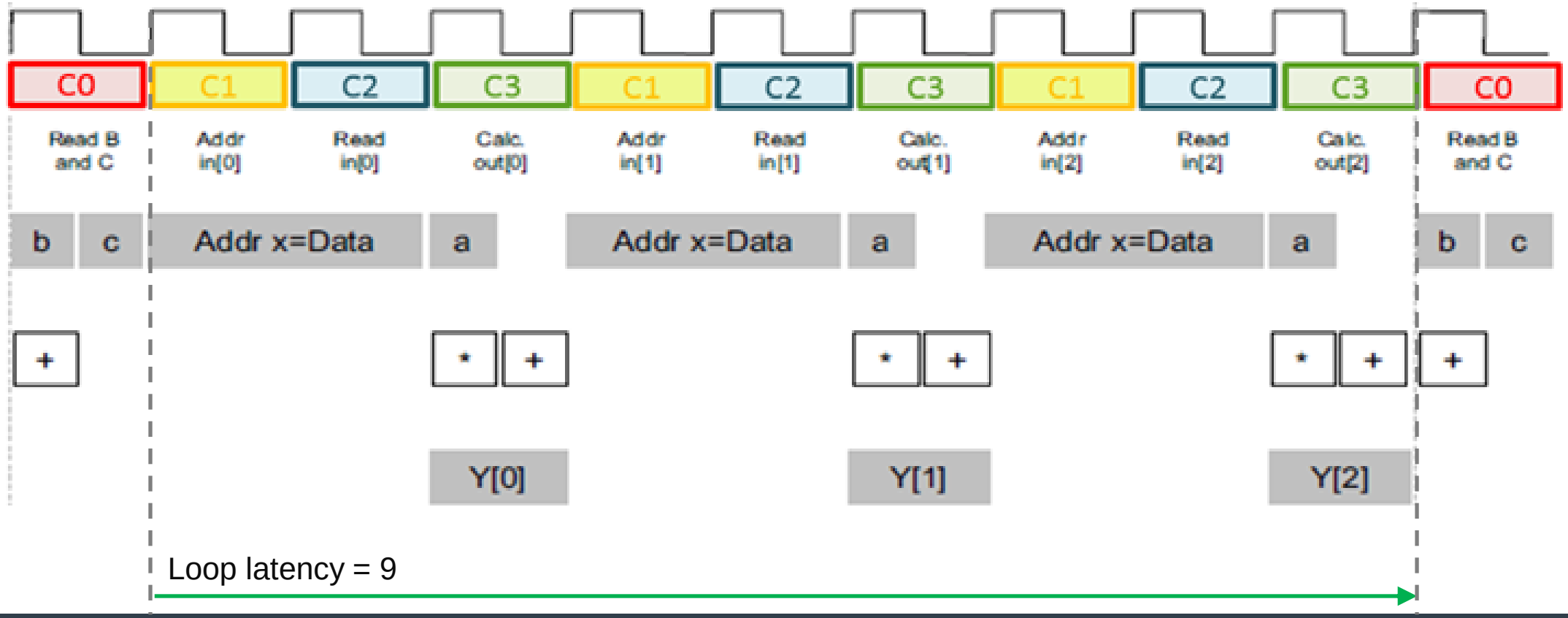
# Terminology for Measuring in Clock Cycles




**Loop initiation interval:** Number of clock cycles before the next iteration of the loop starts to process data

- Three clock cycles in this case

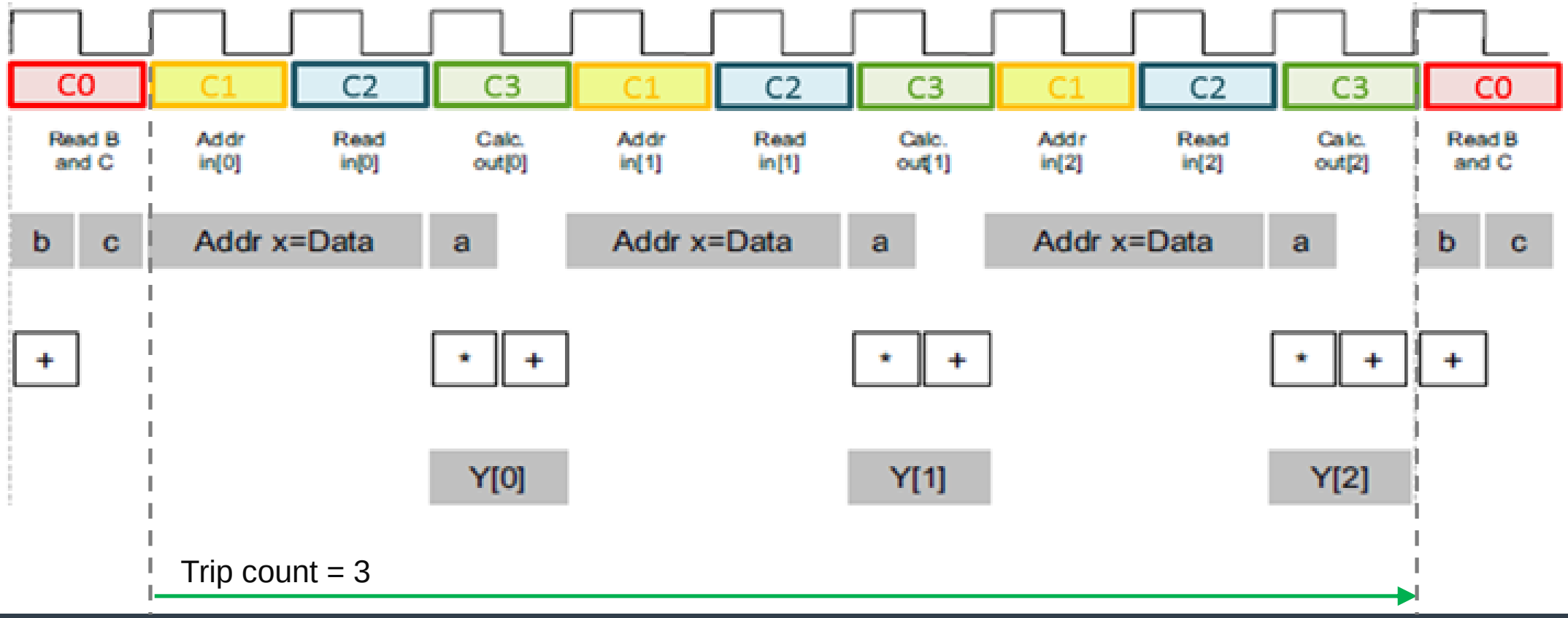
# Terminology for Measuring in Clock Cycles




**Loop latency:** Number of cycles to execute all iterations of the loop

- Nine cycles in this case

# Terminology for Measuring in Clock Cycles




**Trip count:** Number of iterations in the loop; three in this case

**Data rate:** Equal to the 1/throughput * clock frequency

# Key Attributes of C Code for HLS

**Function**

HLS tool converts this function into a hardware

**Top-Level I/O**

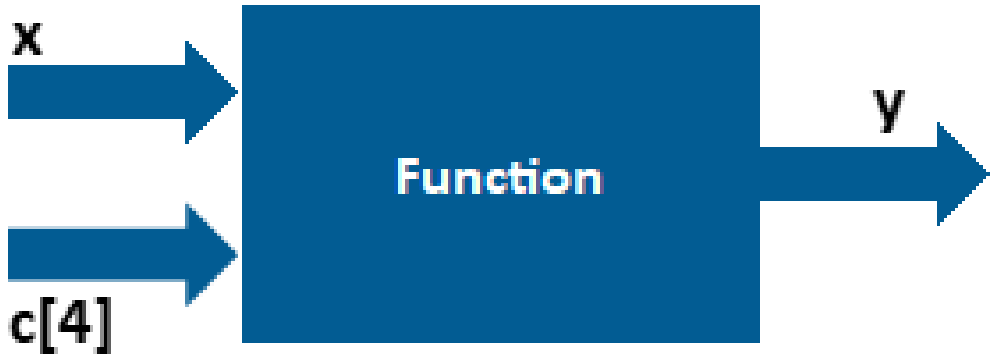
HLS tool converts each argument of a top-level function into a physical connection, which is an IO interface port of the RTL

**Types**

The type of the physical connection is defined by the type of the argument

Influences the area and performance

```
void fir ( data_t *y,
           coef_t  c[4],
           data_t  x)
```




These predetermined types are supported by the Vitis HLS tool





AMD XILINX

# C to RTL Conversion

- Does the HLS tool synthesize all parts of the C code in the same manner?

  No, during C to RTL conversion, the HLS tool synthesizes different parts of the code differently

Top level functions

Loops

Other C functions

Arrays

# The Key Attributes of C code

```
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: the same in hardware

**Top Level IO :** The arguments of the top-level function determine the hardware RTL interface ports

**Types:** All variables are of a defined type. The type can influence the area and performance

**Loops:** Functions typically contain loops. How these are handled can have a major impact on area and performance

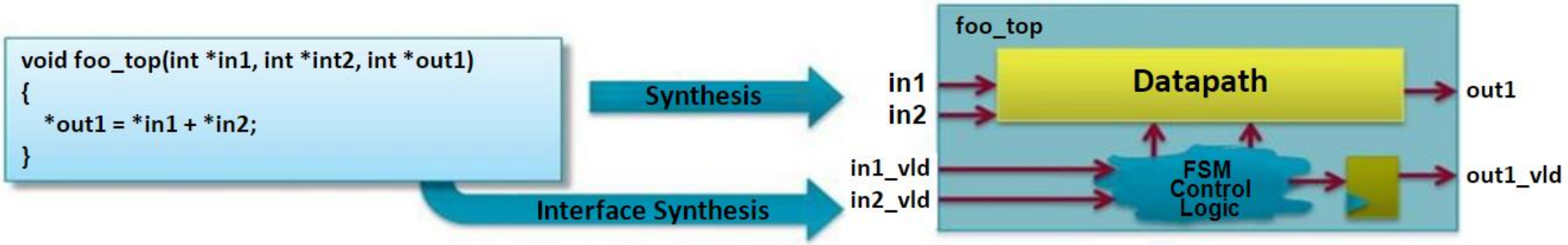
**Arrays:** Arrays are used often in C code. They can influence the device IO and become performance bottlenecks

**Operators:** Operators in the C code may require sharing to control area or specific hardware implementations to meet performance

**Let's examine the default synthesis behavior of these …**

# Top-Level Functions

- Top-level function arguments become I/O ports on the RTL designs
- These ports can optimally be implemented with an interface synthesis, hardware protocol
- The top-level function foo_top has two input arguments in1 and in2 and one output argument out1. These arguments have become the ports of the equivalent generated hardware



# Other C Functions

- Each function is translated into an RTL block
  - Verilog module, VHDL entity

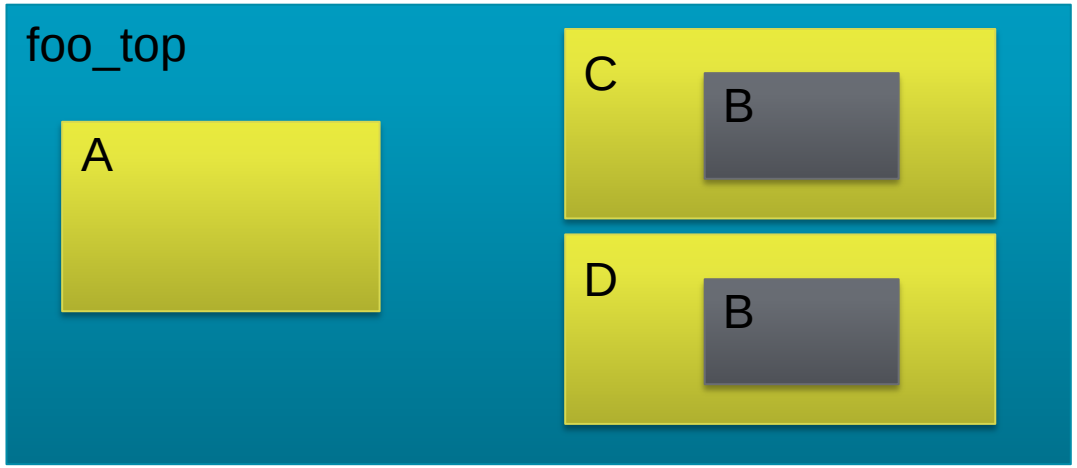
**Source Code**

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


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

my_code.c

**RTL hierarchy**




**Each function/block can be shared like any other component (add, sub, etc) provided it's not in use at the same time**

  - By default, each function is implemented using a common instance
  - Functions may be inlined to dissolve their hierarchy
    - Small functions may be automatically inlined

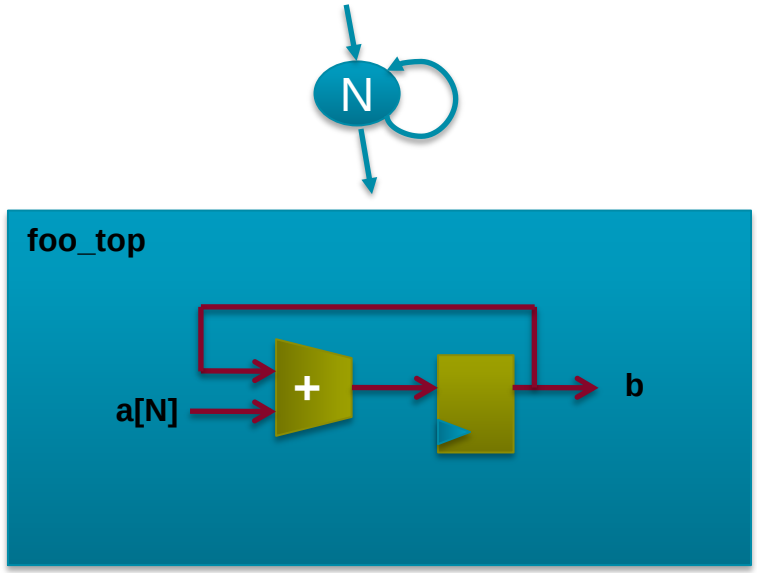
# Loops

- In Vitis HLS, Loops in the C functions are kept rolled and are pipelined by default to improve performance
  - Each C loop iteration ➔ Implemented in the same state
  - Each C loop iteration ➔ Implemented with same resources

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

**Loops require labels if they are to be referenced by Tcl directives**
**(GUI will auto-add labels)**

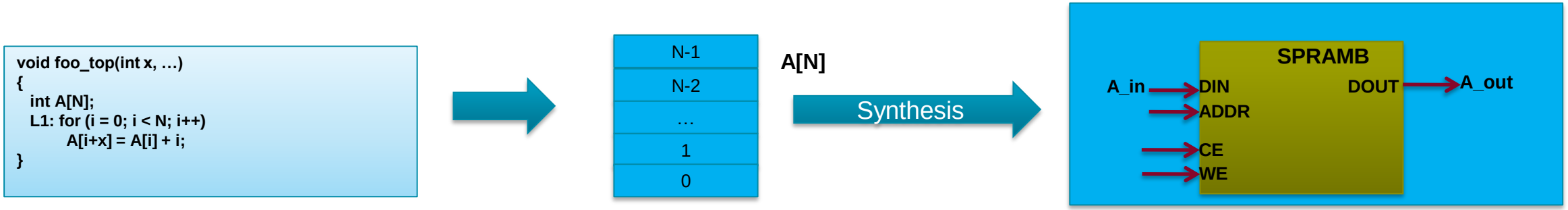
Synthesis




- Loops can be unrolled if their indices are statically determinable at elaboration time
  - Not when the number of iterations is variable
- Unrolled loops result in more elements to schedule but greater operator mobility
  - Let's look at an example ….

# Arrays

- An array in C code is implemented by a memory in the RTL
  - By default, arrays are implemented as block RAMs




- The array can be targeted to any memory resource in the library
  - The ports (Address, CE active high, etc.) and sequential operation (clocks from address to data out) are defined by the library model
  - All RAMs are listed in the Vitis HLS Library Guide： block RAM (BRAM), LUT RAM, or UltraRAM
- Arrays can be merged with other arrays and reconfigured
  - To implement them in the same memory or one of different widths & sizes
- Arrays can be partitioned into individual elements
  - Implemented as smaller RAMs or registers

# Types = Operator Bit-sizes

**Code**

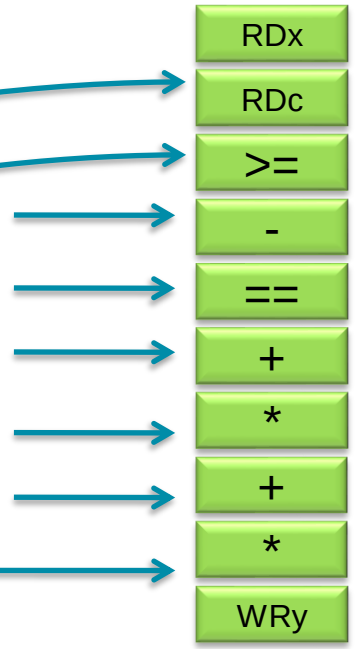
```
void fir (
  data_t *y,
  coef_t c[4],
  data_t x
  ) {

  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;
}
```

**From any C code example ...**

**Operations**




**Operations are extracted…**

**Types**

**Standard C types**

| | | |
|---|---|---|
| long long (64-bit) | short (16-bit) | unsigned types |
| int (32-bit) | char (8-bit) | |
| float (32-bit) | double (64-bit) | |

**Arbitary Precision types**

**C++:** ap_(u)int types (1-1024)
ap_fixed types

Can be used to define any variable to be a specific bit-width (e.g. 17-bit, 47-bit etc).

**The C types define the size of the hardware used: handled automatically**

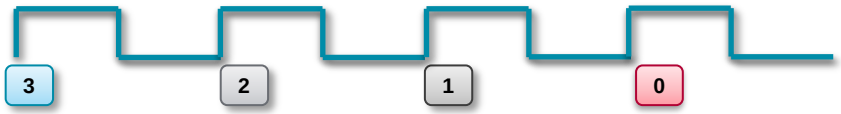




AMD XILINX

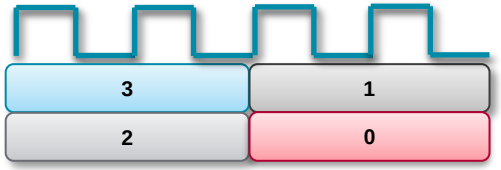
# Operators

- Operator sizes are defined by the type
  - The variable type defines the size of the operator
- Vitis HLS will try to minimize the number of operators
  - By default Vitis HLS will seek to minimize area after constraints are satisfied
- User can set specific limits & targets for the resources used
  - Allocation can be controlled
    - An upper limit can be set on the number of operators or cores allocated for the design: This can be used to force sharing
    - e.g limit the number of multipliers to 1 will force Vitis HLS to share




**Use 1 mult, but take 4 cycle even if it could be done in 1 cycle using 4 mults**

  - Resources can be specified
    - The cores used to implement each operator can be specified
    - e.g. Implement each multiplier using a 2-stage pipelined core (hardware)

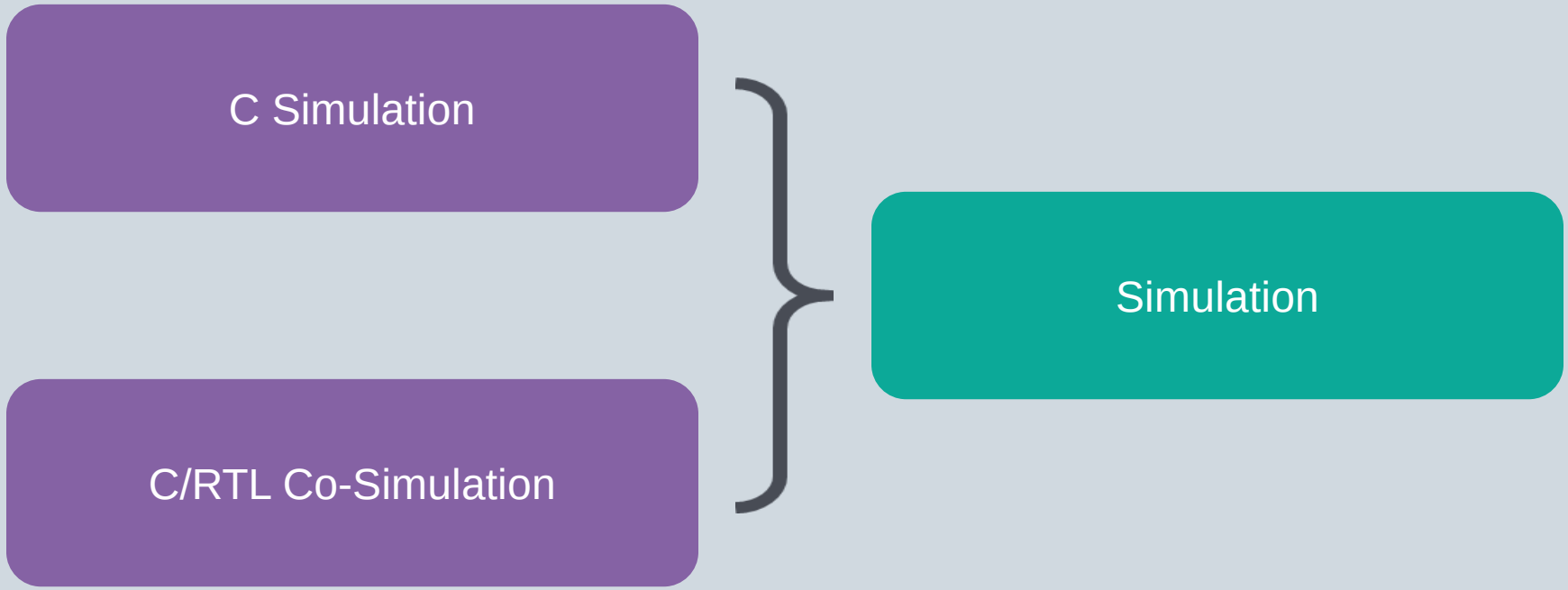



**Same 4 mult operations could be done with 2 pipelined mults (with allocation limiting the mults to 2)**

# Validation and Verification Flow

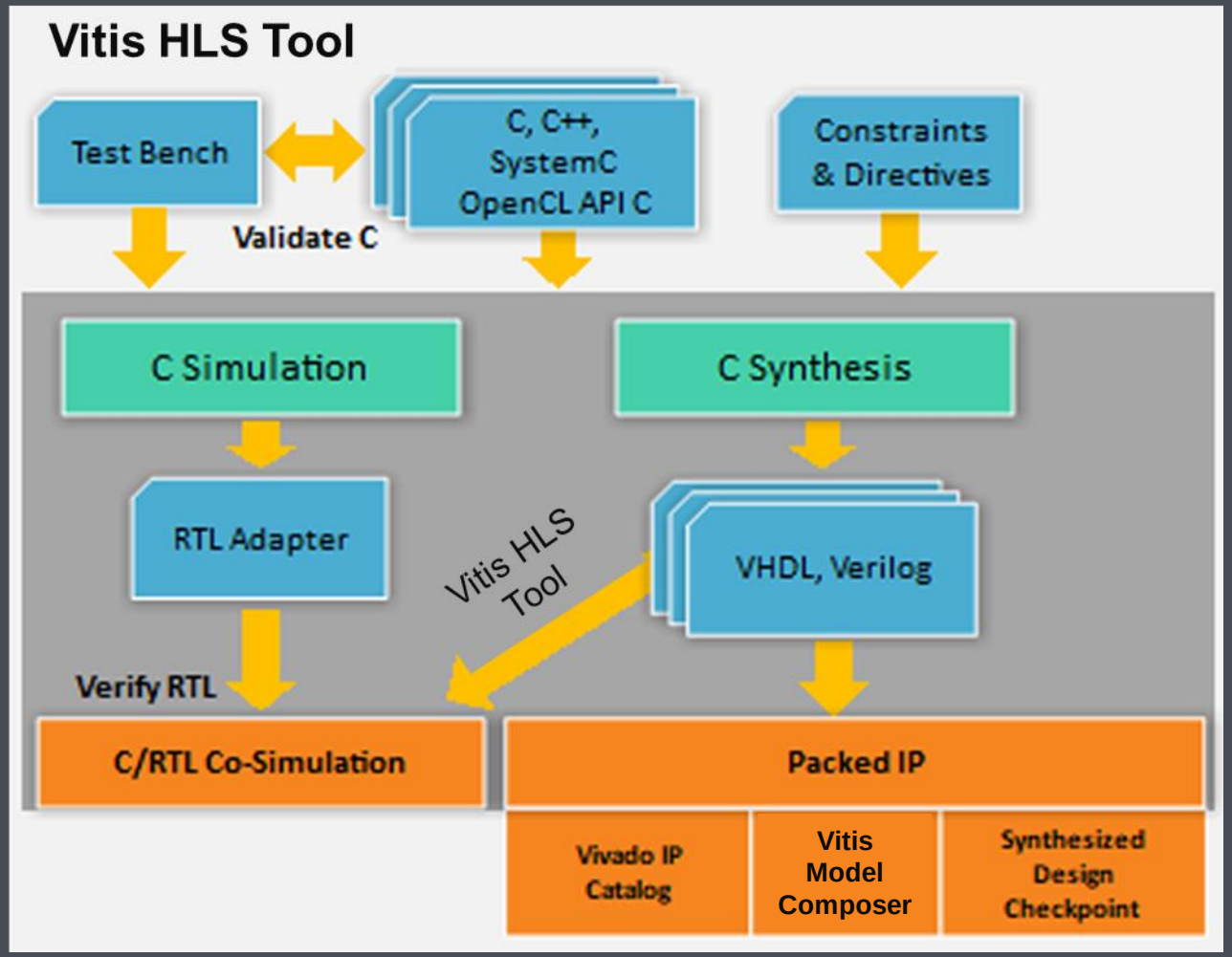
# C Validation and RTL Verification

Verification of the design using the HLS tool is a two-step process



# C Validation and RTL Verification

- At pre-synthesis stage, the C simulation checks the functionality of the C algorithm
- C validation is fast and free, and it uses the C test bench
- Post-synthesis verification is automated through the C/RTL co-simulation feature
  - Reuses the C test bench to perform verification on the output RTL





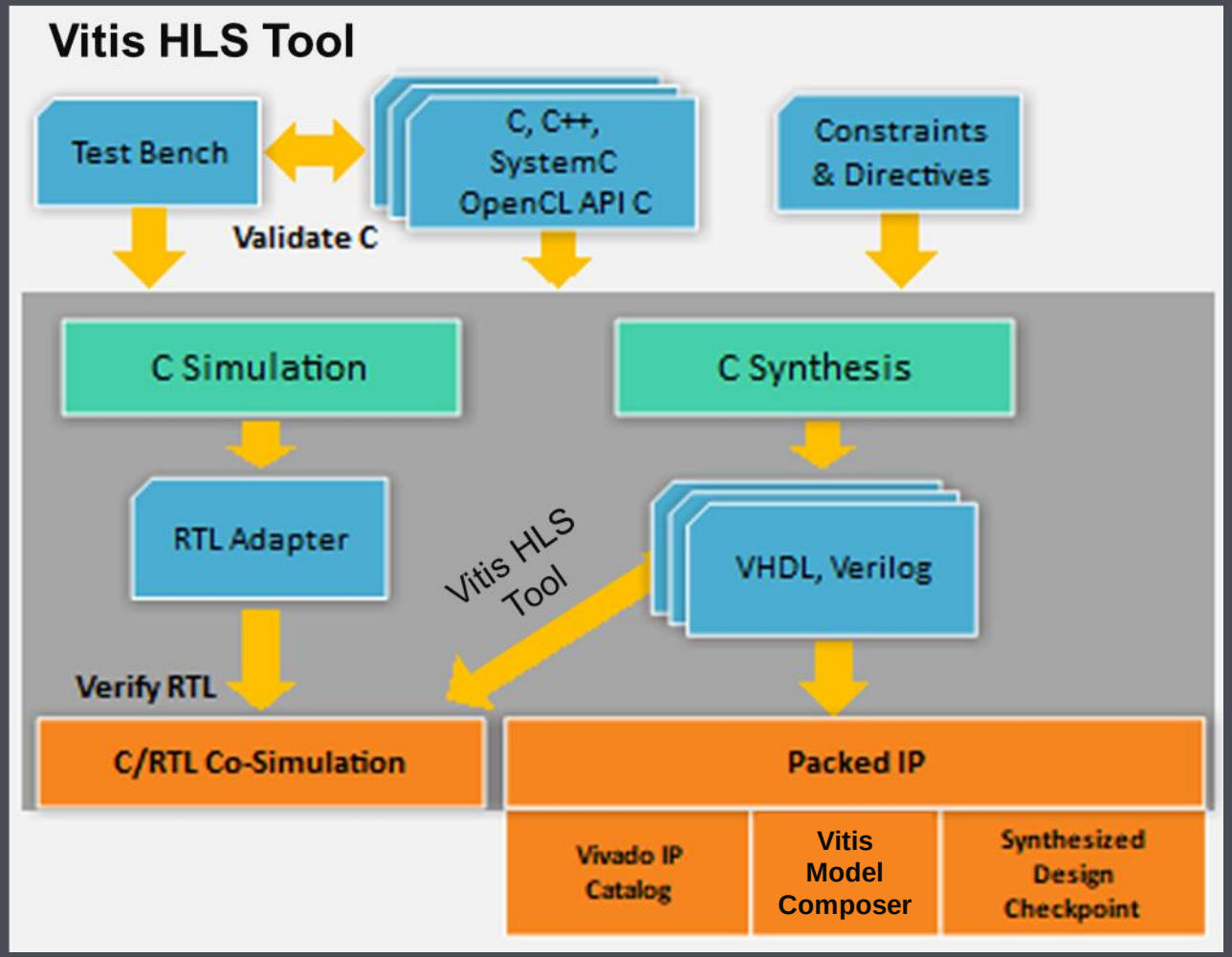



AMD XILINX

# C Validation and RTL Verification

C/RTL co-simulation involves three steps:

- C simulation is executed and the inputs to the top-level function are saved as “input vectors”
- The “input vectors” are used in a C/RTL co-simulation using the RTL created by the Vitis HLS tool. The outputs from the RTL are saved as “output vectors”
- The “output vectors” are applied to the C test bench after the function for synthesis to verify the results are correct








AMD XILINX

# C Function Test Bench

Xilinx has provided some best practices that should be followed when writing a C test bench:

C test bench should compare the results with known good values

Automatically confirms that the C validation and RTL verification is correct

Return value of the C function test bench is set to:

- Zero: If the results are correct
- Non-zero value: If the results are incorrect

```
int main () {

  int ret=0;
  ...
  ret = system("diff  --brief   -w output.dat  output.golden.dat");

  if (ret != 0) {
       printf("Test failed  !!!\n");
       ret=1;
  } else {
       printf("Test passed !\n");
  }
  ...
  return ret;
}
```

# Determine or Create the Top-Level Function

In any C program, the top-level function is called main()
In the Vitis HLS tool design flow, we can specify any sub-function below main()

Guidelines

- Only one function is allowed as the top-level function for synthesis
- Any sub-functions in the hierarchy under the top-level function for synthesis are also synthesized
- If we want to synthesize functions that are not in the hierarchy under the top-level function, we must merge the functions into a single top-level function for synthesis
- Separate the test bench and the design file

# Determine or Create the Top-Level Function

Given a case where functions func_A and func_B are to be implemented in FPGA

**main.c**

```
int main () {
        ...
        func_A(a,b,*i1);
        func_B(c,*i1,*i2);
        func_C(*i2,ret)

        return ret;
}
```

func_A
func_B
func_C

Re-partition the design to create a **new** single top-level function inside main()

**main.c**

```
#include func_AB.h
int main (a,b,c,d) {
        ...
        // func_A(a,b,i1);
        // func_B(c,i1,i2);
        func_AB (a,b,c, *i1, *i2);
        func_C(*i2,ret)

        return ret;
}
```

func_AB
func_C

**func_AB.c**

```
#include func_AB.h
func_AB(a,b,c, *i1, *i2) {
        ...
        func_A(a,b,*i1);
        func_B(c,*i1,*i2);
        ...
}
```

func_A
func_B

**Recommendation is to separate testbench and design files***

*Else add file as design file ***and*** testbench

# Using the Flow Navigator

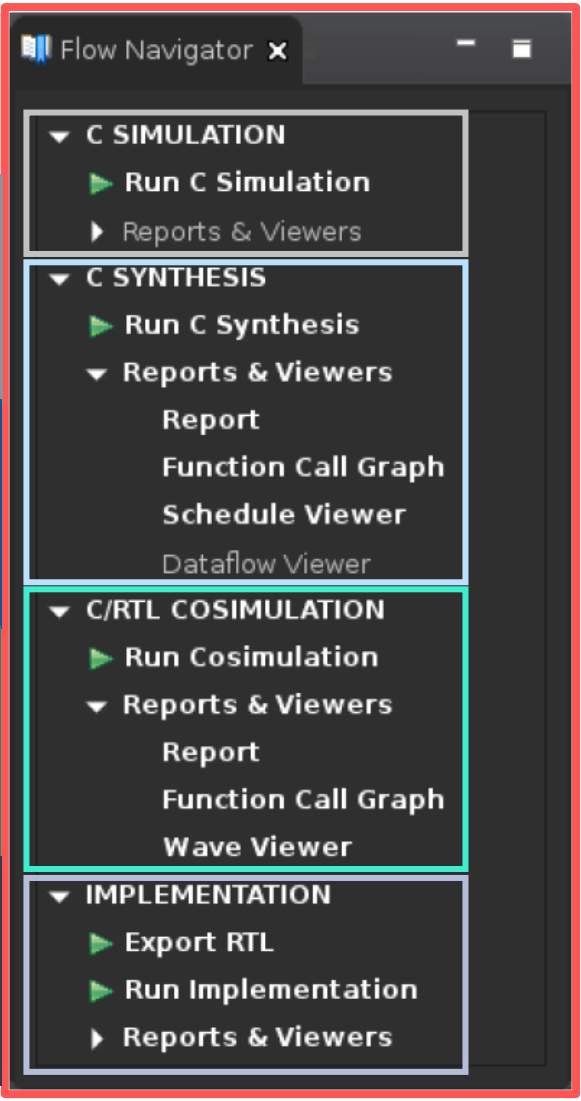
Process flow representation of the Vitis HLS tool design flow
All viewers and reports are also available

01 **C SIMULATION**: Opens the C Simulation dialog box and lists the available reports after simulation

02 **C SYNTHESIS**: Opens the C Synthesis dialog box and lists the available reports after synthesis

03 **C/RTL COSIMULATION**: Opens the C/RTL Cosimulation dialog box and lists the available reports after C/RTL simulation

04 **IMPLEMENTATION**: Specifies the format and location of the exported RTL file from the Vitis HLS tool and also runs Vivado synthesis and implementation








AMD XILINX

# Summary

# Quick Q&A

- How is hardware extracted from C code?
  - Control and datapath can be extracted from C code at the top level
  - The same principles used in the example can be applied to sub-functions
    - At some point in the top-level control flow, control is passed to a sub-function
    - Sub-function may be implemented to execute concurrently with the top-level and or other sub-functions
- How is this control and dataflow turned into a hardware design?
  - Vitis HLS maps this to hardware through scheduling and binding processes
- How are the functions, loops, arrays and IO ports mapped into FPGA?
- What is the number of clock cycles before a function accepts new input data called?

# Summary

- In high-level synthesis(HLS)
  - C and C++ code is synthesized to RTL
  - Operations in the code map to hardware resources
  - RTL verification is accelerated using the same C test bench
  - HLS C libraries allow common hardware design constructs and functions to be easily modeled in C and synthesized to RTL
- When a C or C++ function synthesizes to RTL, the Vitis HLS tool performs:
  - Scheduling
  - Binding
  - Control logic extraction


# Thank You

# Disclaimer and Attribution



AMD
XILINX