Del Algoritmo al Hardware:
Aprendizaje Automático en Sistemas Embebidos

From Algorithm to Hardware: Machine Learning in Embedded Systems

1 al 11 de Abril, 2025. Universidad Nacional de Mar del Plata - Mar del Plata - Argentina.

High-Level Synthesis: Bridging Software and Hardware

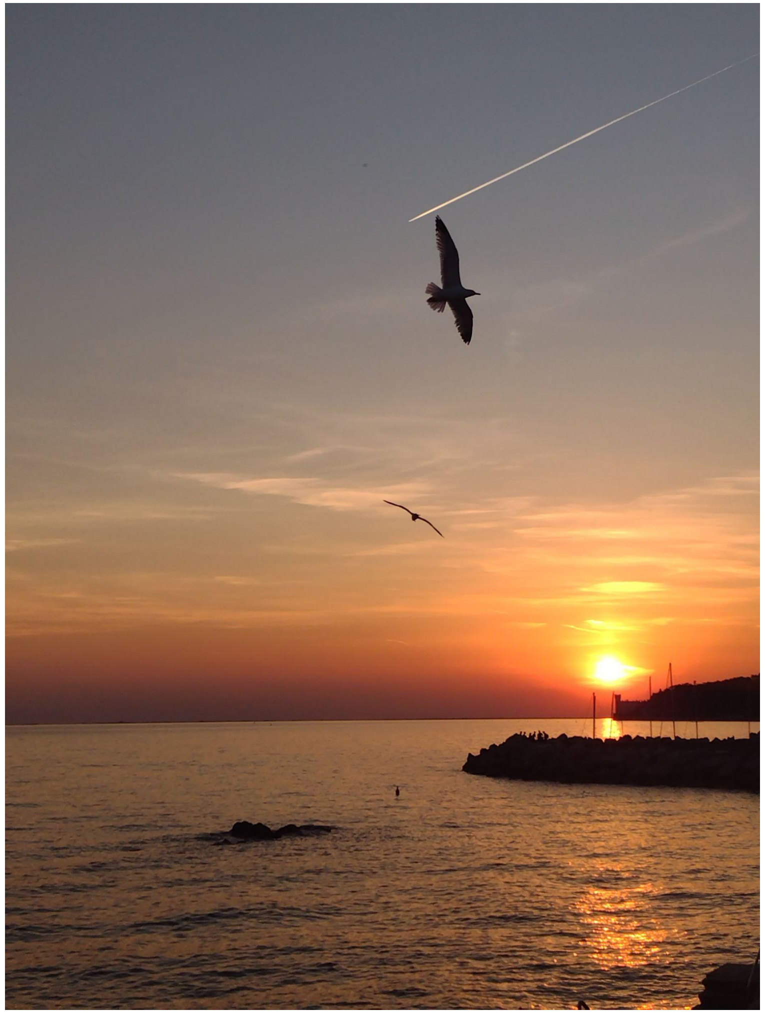
Romina Soledad Molina, Ph.D.
MLab-STI, ICTP

Mar del Plata, Argentina - 2025 -

Universidad Nacional de Mar del Plata
Fundación Williams
The Abdus Salam International Centre for Theoretical Physics

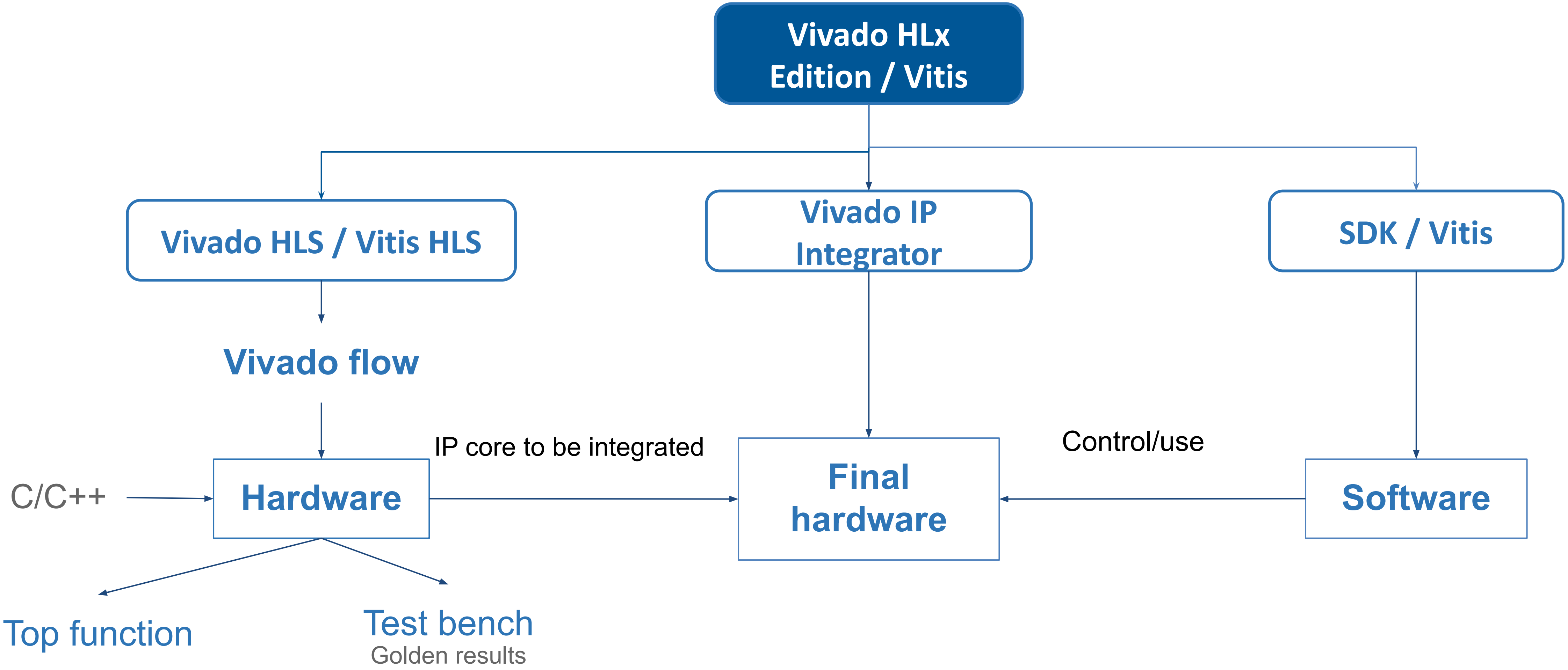
Outline

- Introduction.
- High-level synthesis.
- HLS Component Development Flow.
- C-to-RTL Conversion.
- Language Support.
- Hardware design: Directives/Optimizations.
- Vitis HLS GUI.
- Demo: HLS and Matrix Multiplication.

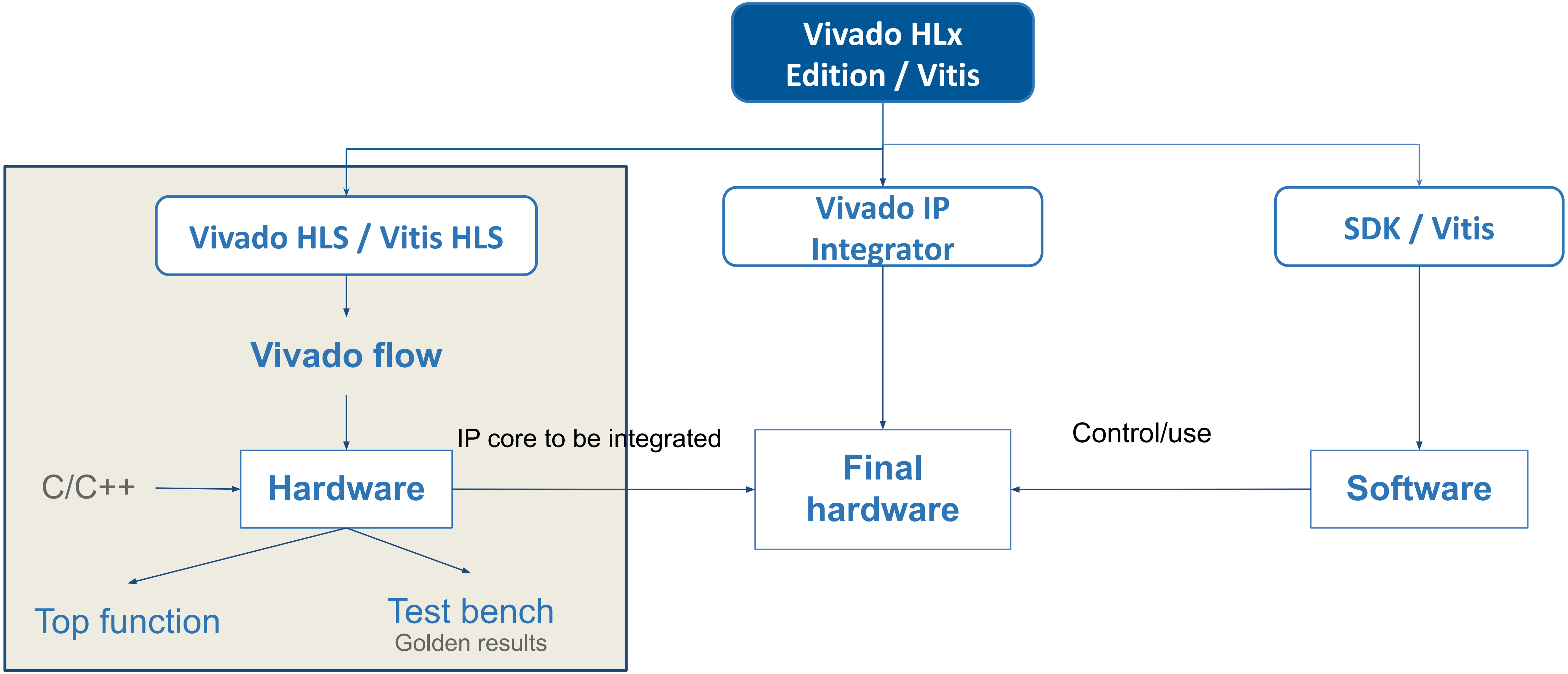


Introduction

Introduction

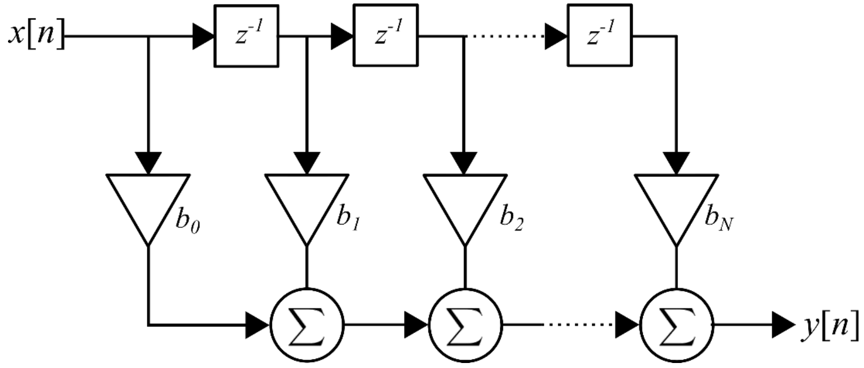


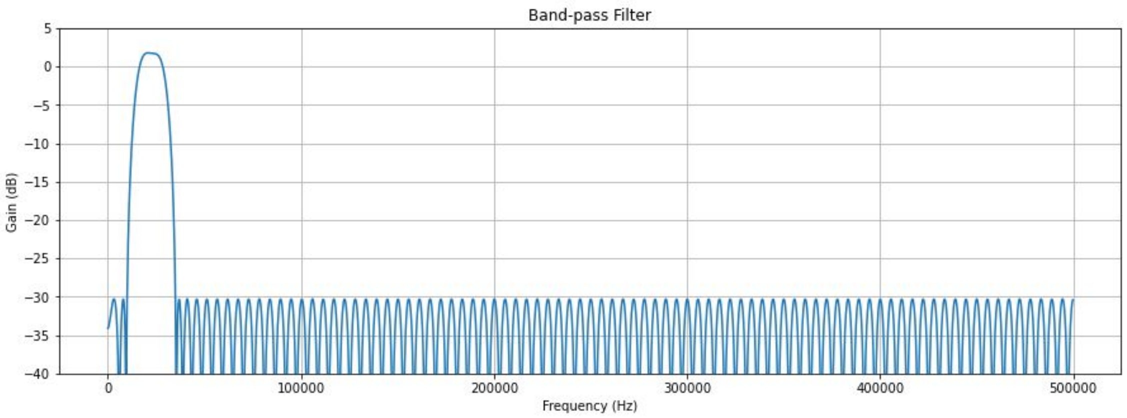
Introduction



Introduction

- **Traditional RTL FIR Filter Design**
    - Define interfaces
    - Define architecture
        - FSM
        - Datapath
    - Write RTL code
    - Write RTL test bench.
- **The design choice is already made.**





Introduction

- *Question: Why the need of High-Level Synthesis tools?*

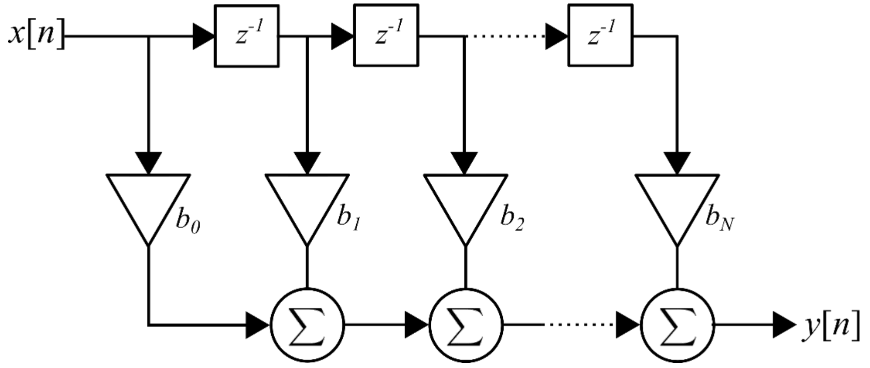
Introduction

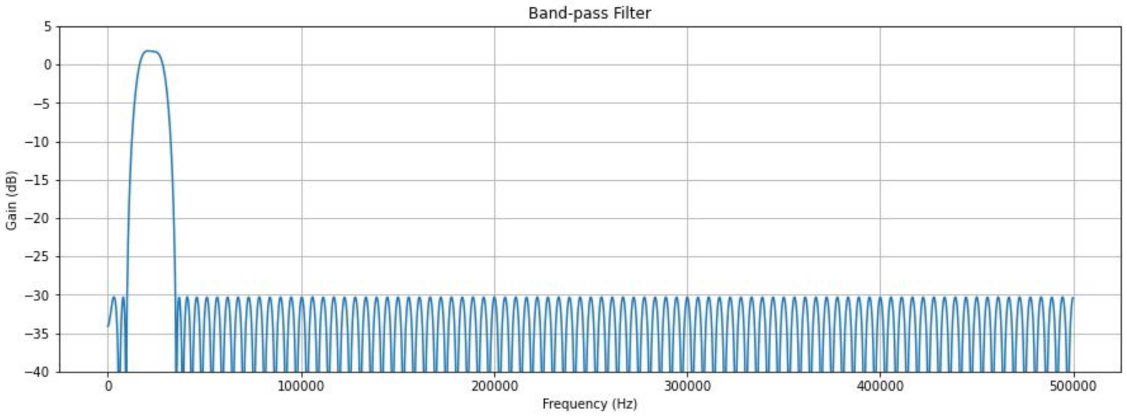
- *Question: Why the need of High-Level Synthesis tools?*
    - Some reasons could be:
        - Productivity boosting.
        - Trend to use FPGA as hardware accelerators.
        - Reduce Time-To-Market.
        - Design Space Exploration.
        - Early metric estimations.
        - Functionality verification through C-based test bench.
        - Reuse.

Introduction

- **HLS-based FIR Filter Design**

```
void fir (data_t *y,  data_t *x ) {
        static data_t shift_reg[N];
        acc_t acc;
        data_t data;
        int i;
        acc=0;
        Shift_Accum_Loop: for (i=N-1;i>=0;i--) {
                if (i==0) {
                        shift_reg[0]= *x;
                        data = *x;
                } else {
                        shift_reg[i]=shift_reg[i-1];
                        data = shift_reg[i];
                 }
        acc+=data*firCoeff[i];
        }
        acc = acc >> 16;
        *y=acc;
}
```





High-Level Synthesis

Introduction

- **High-Level Synthesis**
    - It provides the facility to create RTL from a high level of abstraction.

Introduction

- **High-Level Synthesis**
  - It provides the facility to create RTL from a high level of abstraction.
  - Several implementations are possible from the same source code description.

Introduction

- **High-Level Synthesis**
  - It provides the facility to create RTL from a high level of abstraction.
  - Several implementations are possible from the same source code description.
  - Implements the design based on defaults and user applied directives.

Introduction

- **High-Level Synthesis**
  - It provides the facility to create RTL from a high level of abstraction.
  - Several implementations are possible from the same source code description.
  - Implements the design based on defaults and user applied directives.
  - It allows the optimization of the input code using directives to:
    - Reduce latency
    - Improve performance and throughput
    - Reduce resource utilization

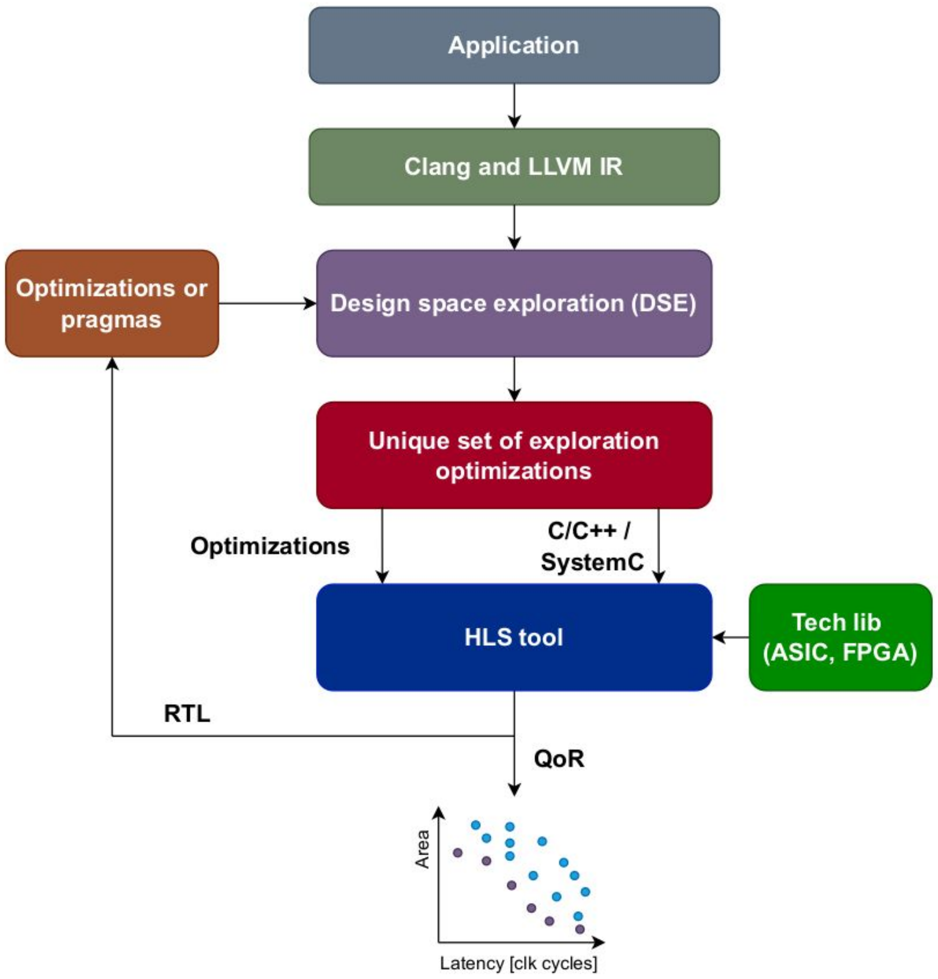
Introduction

- **High-Level Synthesis**

    - It provides the facility to create RTL from a high level of abstraction.
    - Several implementations are possible from the same source description.
    - Implements the design based on defaults and user applied directives.
    - It allows the optimization of the input code using directives to:
        - Reduce latency.
        - Improve performance and throughput.
        - Reduce resource utilization.

Without optimization, HLS tool will look to minimize latency and improve concurrency.

Design space exploration



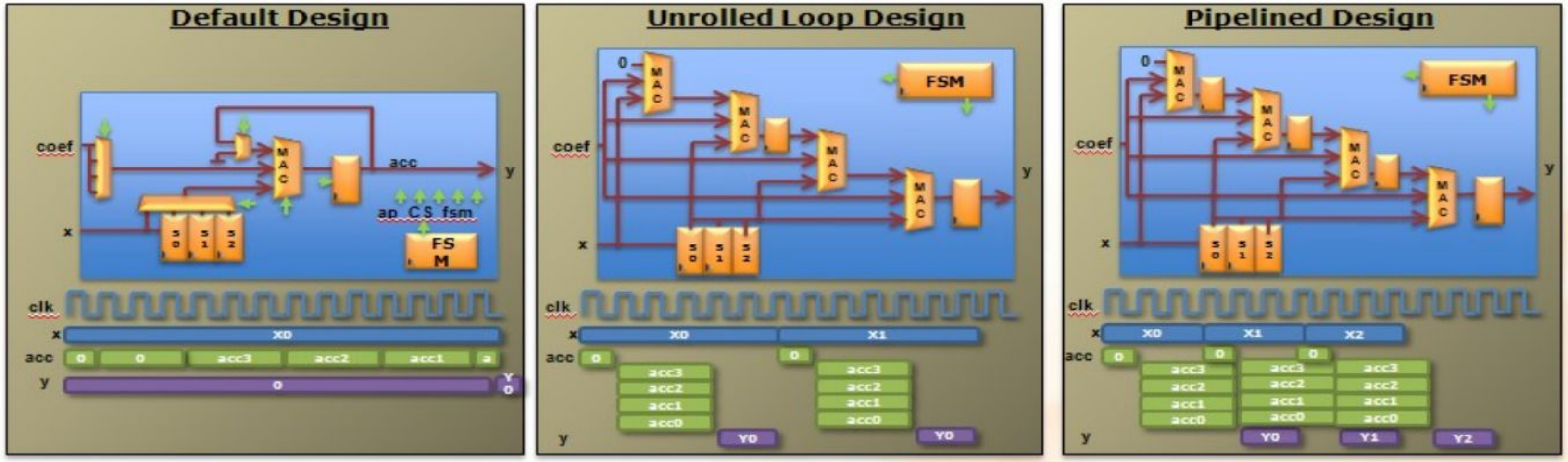
Design space exploration

```
…
  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];
     }
```

Same hardware is used for each loop iteration :
- Small area
- Long latency
- Low throughput

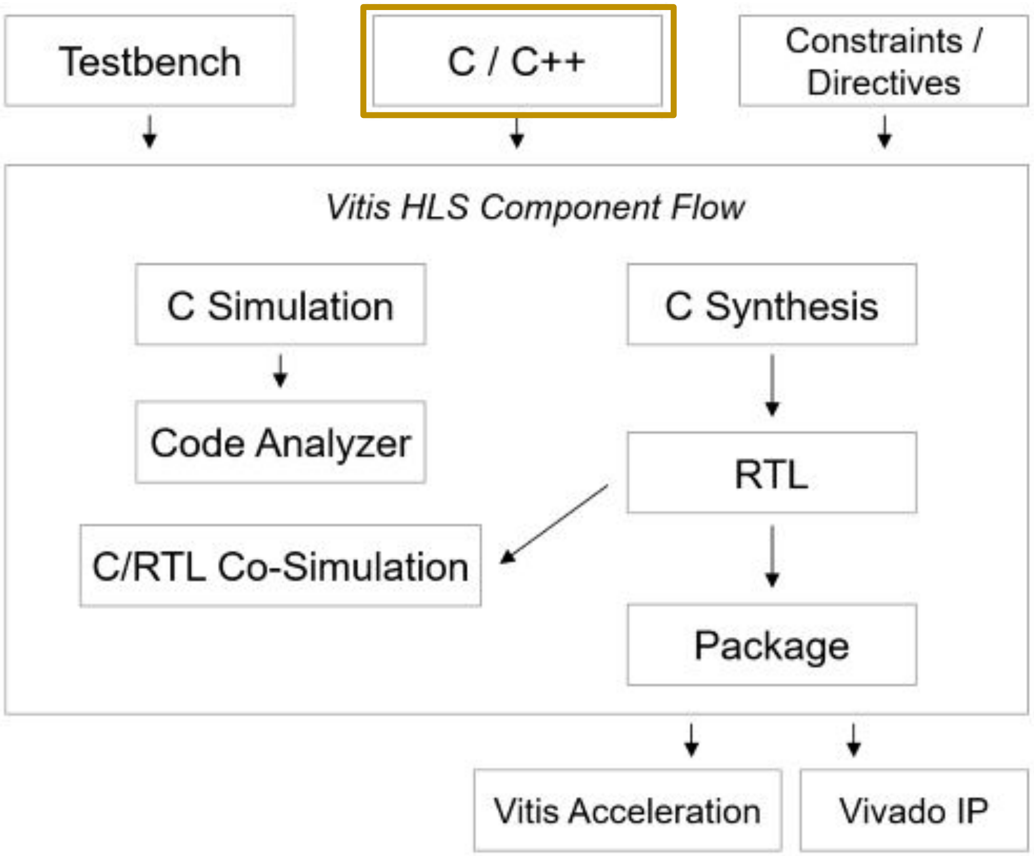
Different hardware for each loop iteration :
- Higher area
- Short latency
- Better throughput

Different iterations executed concurrently:
- Higher area
- Short latency
- Best throughput



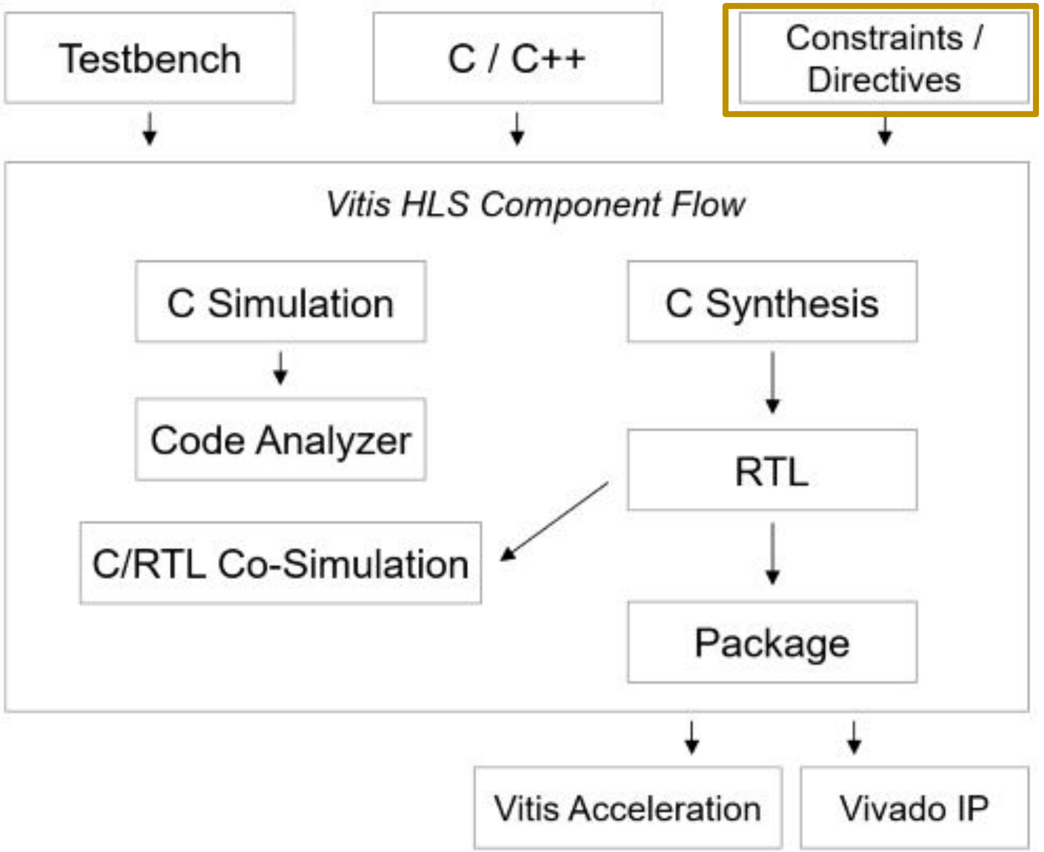
HLS Component Development Flow

HLS Component Development Flow



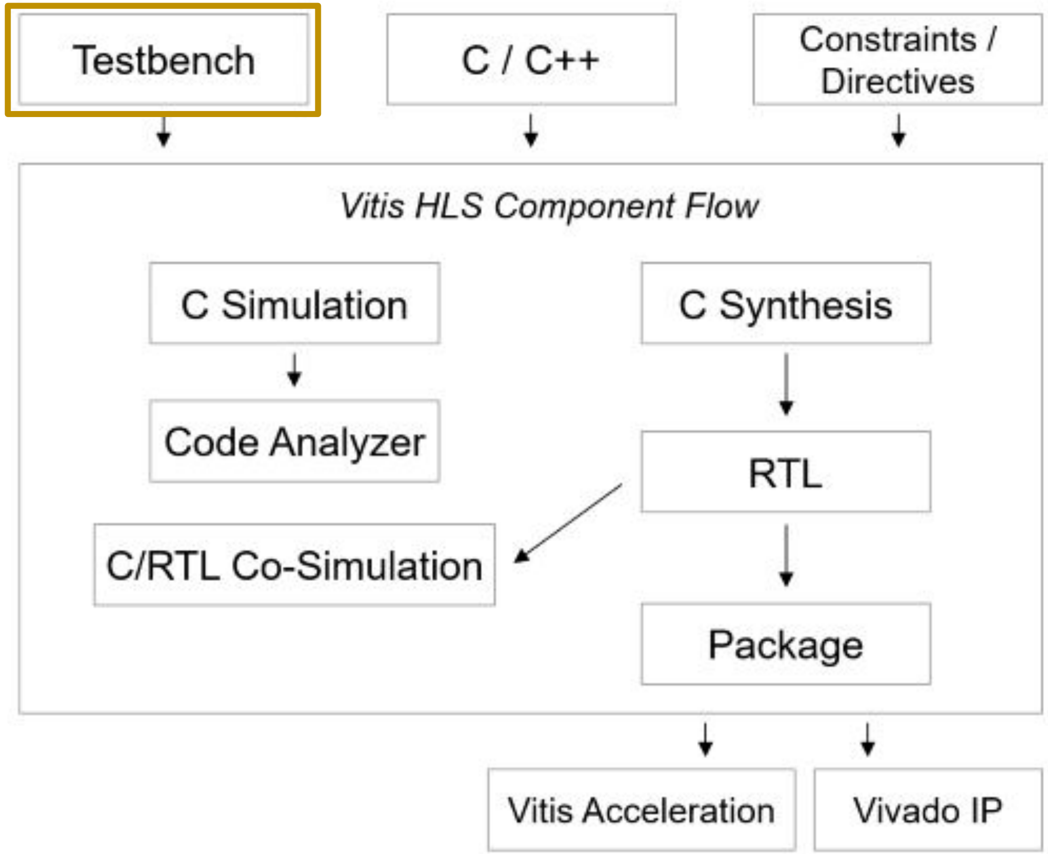
https://docs.amd.com/r/en-US/ug1399-vitis-hls/Introduction-to-Vitis-HLS-Components

HLS Component Development Flow



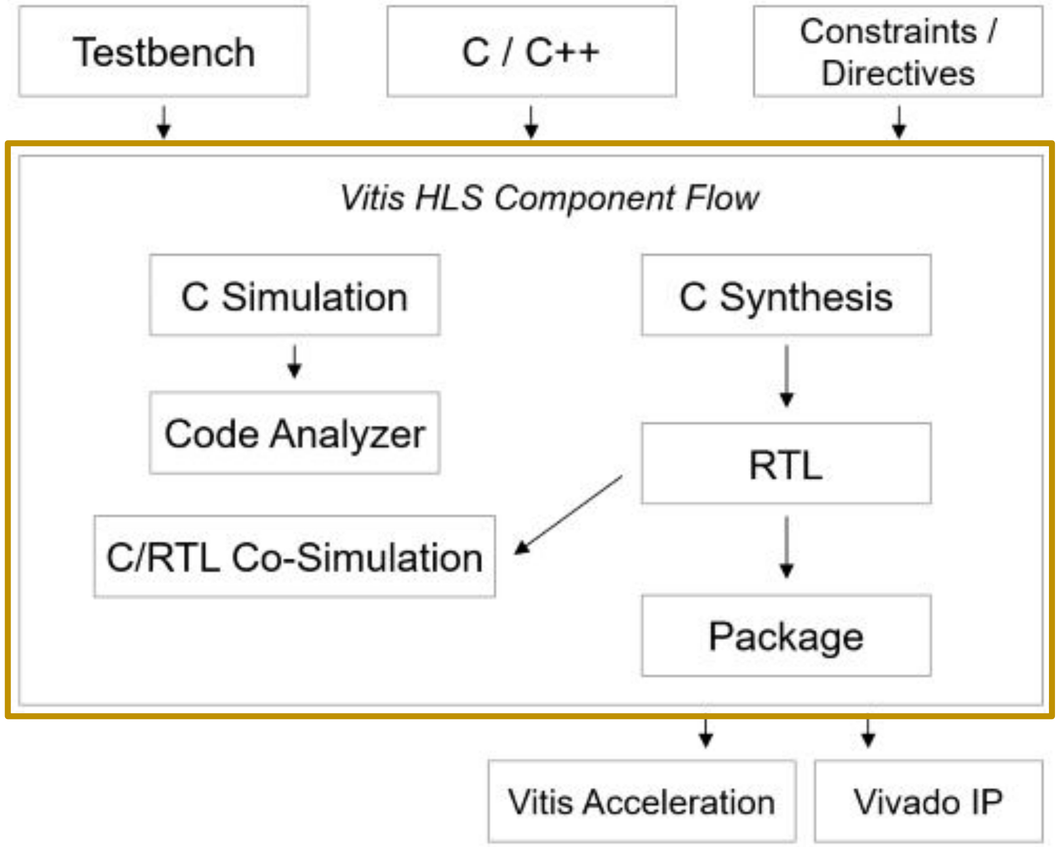
https://docs.amd.com/r/en-US/ug1399-vitis-hls/Introduction-to-Vitis-HLS-Components

HLS Component Development Flow



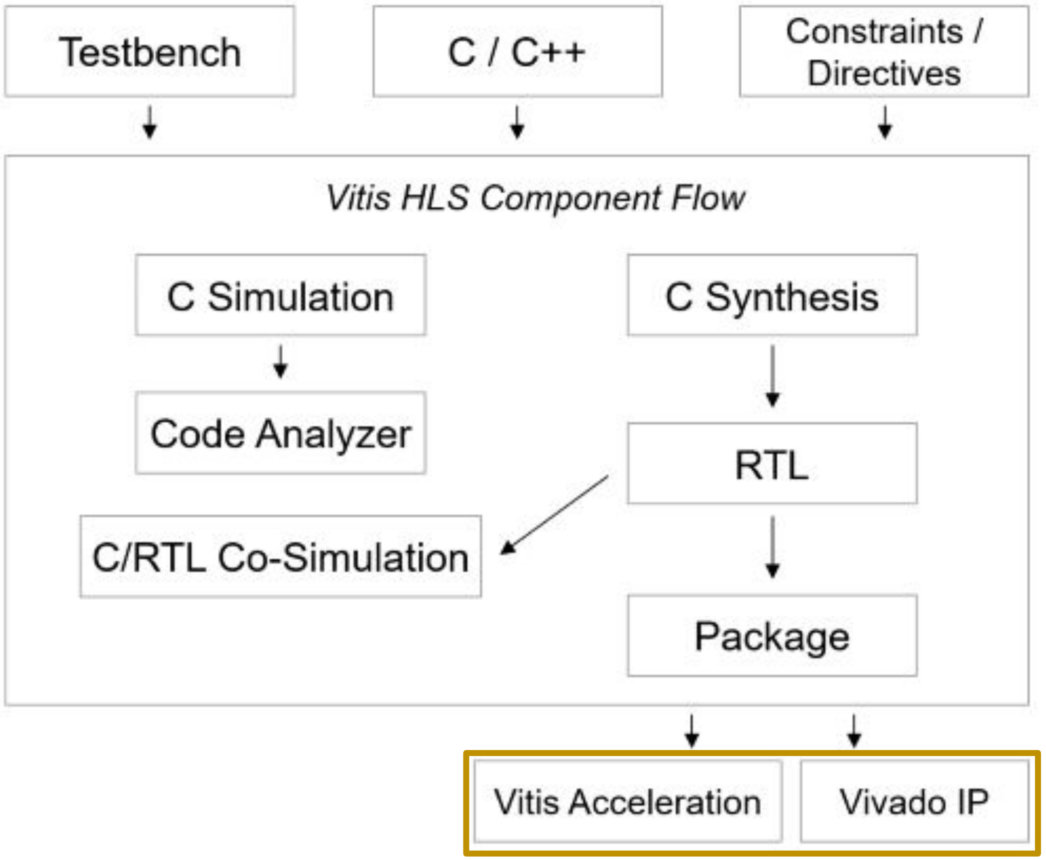
https://docs.amd.com/r/en-US/ug1399-vitis-hls/Introduction-to-Vitis-HLS-Components

HLS Component Development Flow



https://docs.amd.com/r/en-US/ug1399-vitis-hls/Introduction-to-Vitis-HLS-Components

HLS Component Development Flow



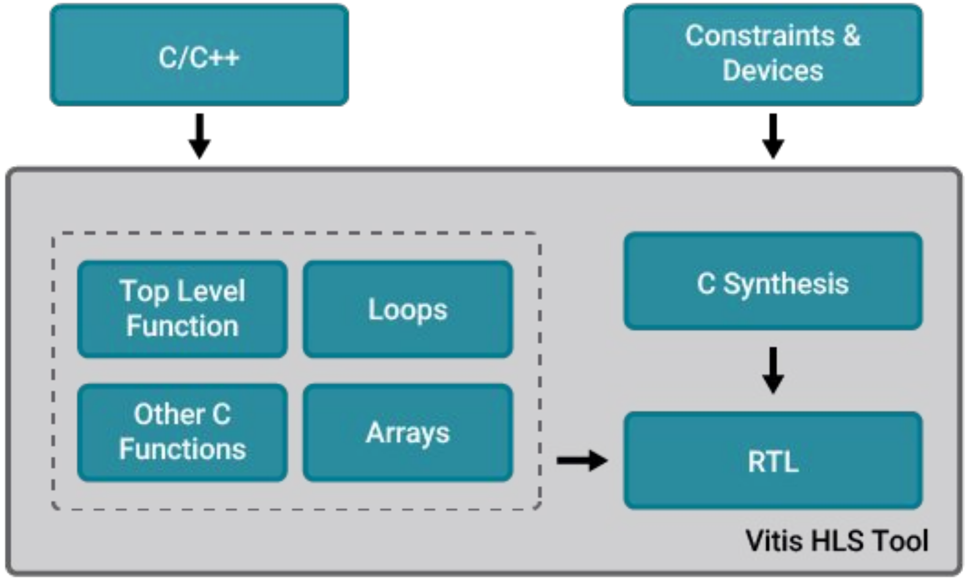
https://docs.amd.com/r/en-US/ug1399-vitis-hls/Introduction-to-Vitis-HLS-Components

C-to-RTL Conversion

C-to-RTL Conversion

The Vitis HLS tool synthesizes different parts of C code differently:

- **Top-level function arguments of the C/C++ code** are synthesized into **RTL I/O** ports and are automatically implemented with an interface synthesis hardware protocol -> **Only one top function.**



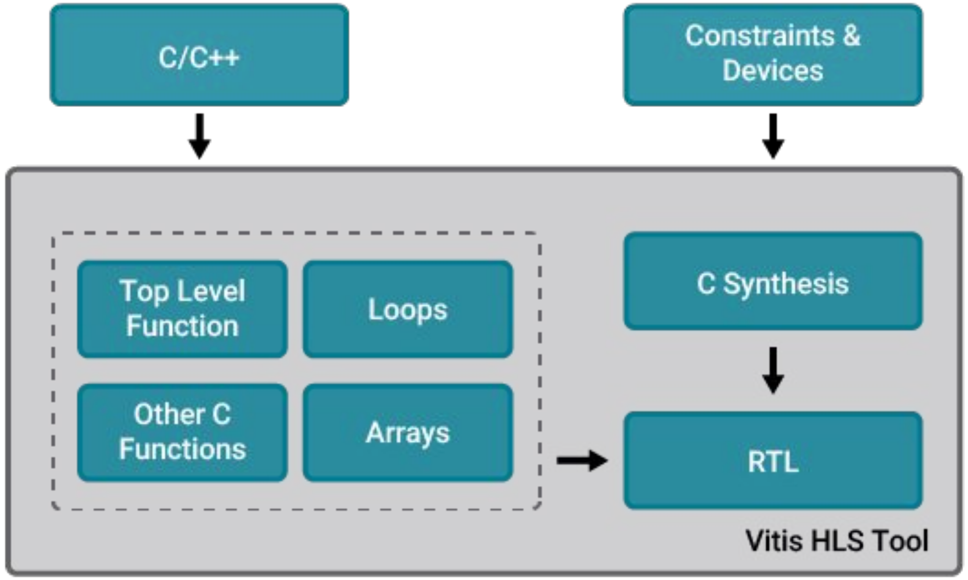
Source: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html

C-to-RTL Conversion

The Vitis HLS tool synthesizes different parts of C code differently:

- **Top-level function arguments of the C/C++ code** are synthesized into **RTL I/O** ports and are automatically implemented with an interface synthesis hardware protocol -> **Only one top function.**

```
void functionA(char x, char a, char b, char c, char y) {
        char tmp = 0;
        tmp = x*a;
         y = tmp+c;
}
```



Source: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html

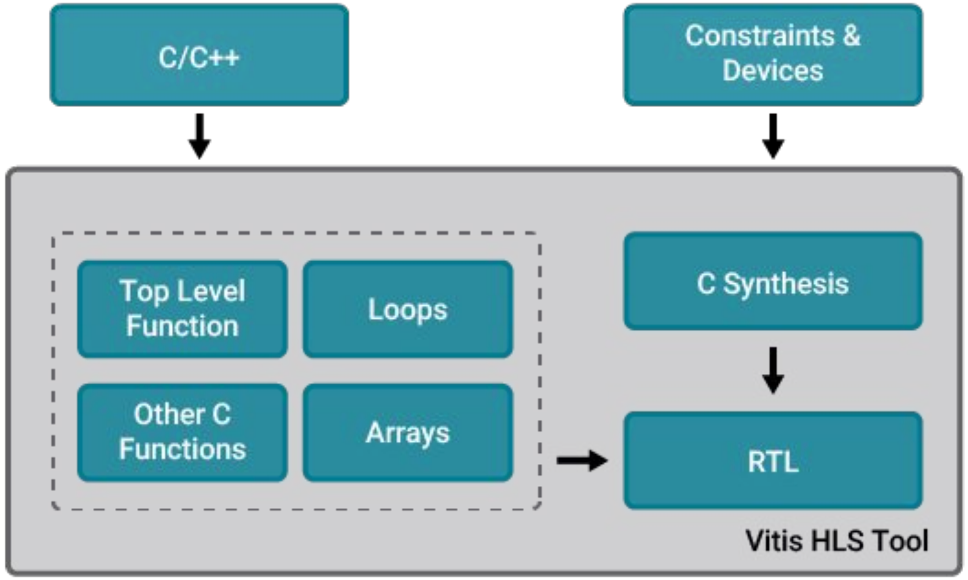
C-to-RTL Conversion

The Vitis HLS tool synthesizes different parts of C code differently:

- **Top-level function arguments of the C/C++ code** are synthesized into **RTL I/O** ports and are automatically implemented with an interface synthesis hardware protocol -> **Only one top function.**

- **Other C functions are synthesized to RTL blocks**—maintaining the design hierarchy.

- **Arrays in the C code** can be targeted to any memory resource, such as BRAM, LUTRAM, and URAM.

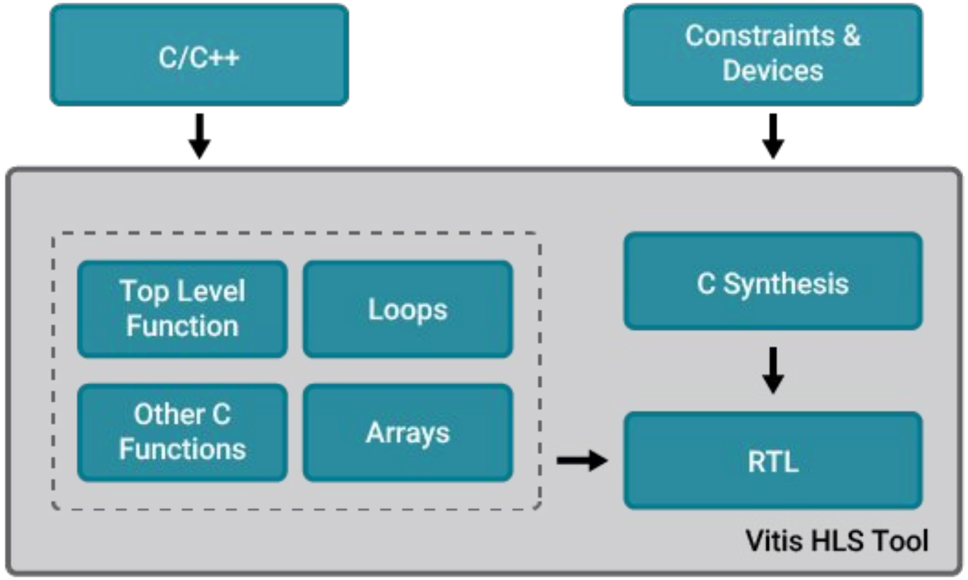


Source: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html

C-to-RTL Conversion

- **Performance metrics**, such as latency, initiation interval, loop iteration latency, and resource utilization, can be reviewed with synthesis reports.

- Vitis HLS tool **pragmas and optimization directives** allow for configuring the synthesis results for the C/C++ code.

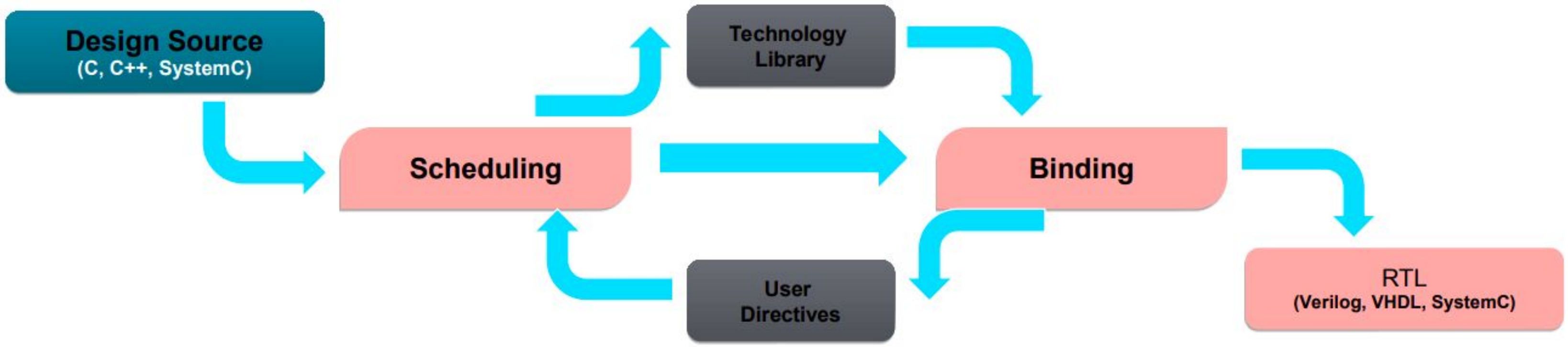


Source: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html

C-to-RTL Conversion

Scheduling and binding

- **Scheduling** determines in which clock cycle an operation will occur.
- **Binding** determines which library cell is used for each operation.
    - For example, for a functional unit like adder, there can be many options like ripple-carry adder, carry-look-ahead-adder etc.



Source: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html

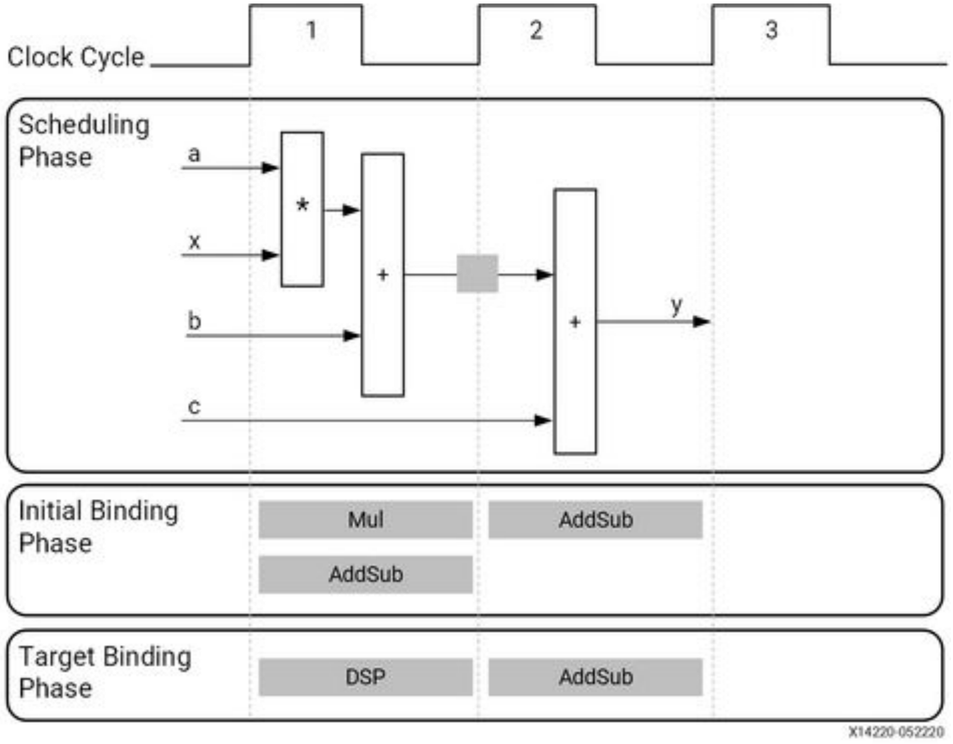
C-to-RTL Conversion
Scheduling and binding

```
int foo(char x, char a, char b, char c) {
    char y;
    y = x*a+b+c;
    return y;
}
```

Source: https://docs.amd.com/r/2020.2-English/ug1399-vitis-hls/Scheduling-and-Binding-Example

C-to-RTL Conversion
Scheduling and binding

```
int foo(char x, char a, char b, char c) {
    char y;
    y = x*a+b+c;
    return y;
}
```



Source: https://docs.amd.com/r/2020.2-English/ug1399-vitis-hls/Scheduling-and-Binding-Example

C-to-RTL Conversion

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;
}
```

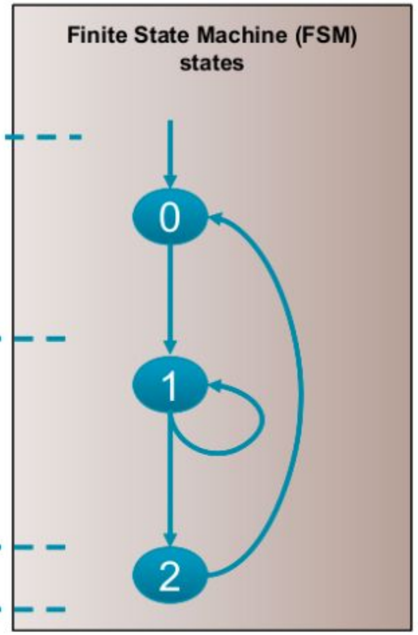
Function Start

For-Loop Start

For-Loop End

Function End

Control Behavior



From any C code example ..

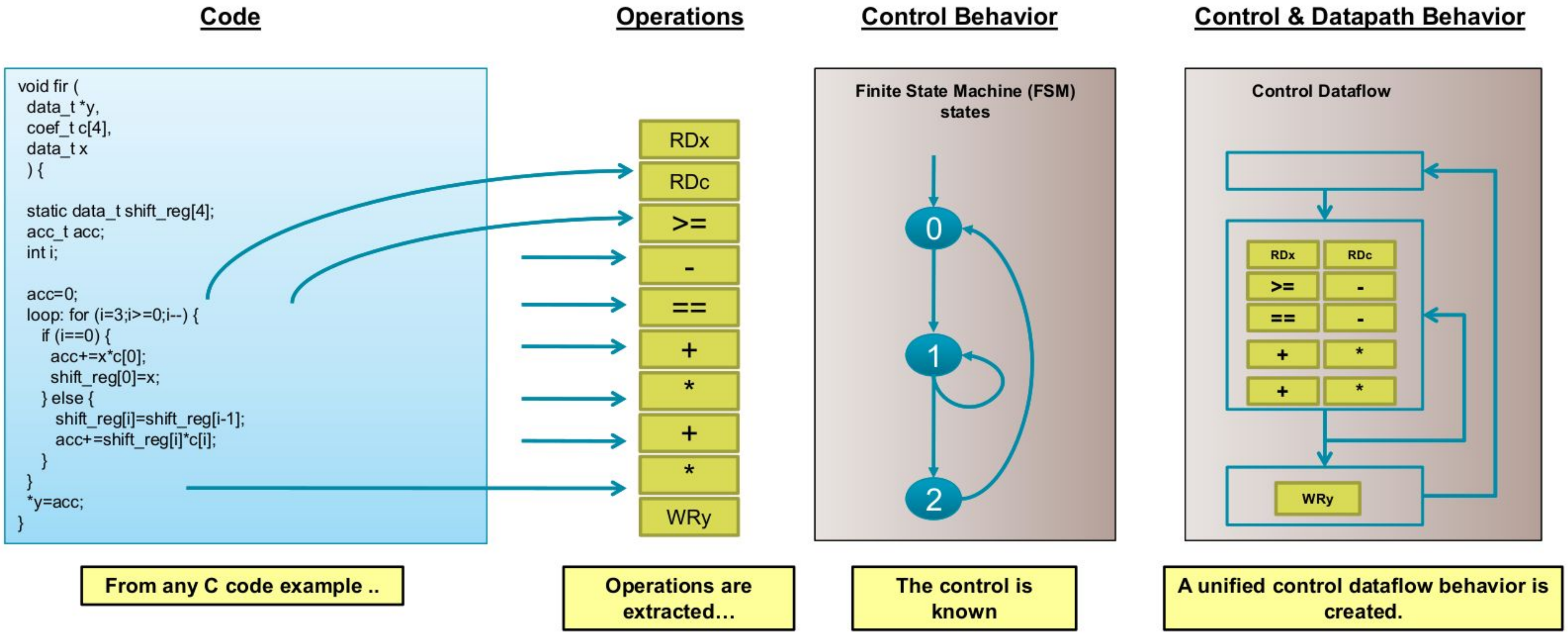
The loops in the C code correlated to states of behavior

This behavior is extracted into a hardware state machine

Intro to HLS 11- 8

XILINX ALL PROGRAMMABLE.

C-to-RTL Conversion



XILINX ALL PROGRAMMABLE.

Language Support

Language Support

- Vivado/Vitis HLS supports C, C++, SystemC, and OpenCL API C kernel.
- Supports arbitrary precision types for all input languages.
- Floating point support.
- Support for OpenCV functions.

Source: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html

Language Support

- The function must contain the entire functionality of the design.
- None of the functionality can be performed by system calls to the operating system.
- The C/C++ constructs must be of a fixed or bounded size.
- The implementation of those constructs must be unambiguous.

Source: https://docs.amd.com/r/2020.2-English/ug1399-vitis-hls/Vitis-HLS-Coding-Styles

Language Support

- The function must contain the entire functionality of the design.
- None of the functionality can be performed by system calls to the operating system.
- The C/C++ constructs must be of a fixed or bounded size.
- The implementation of those constructs must be unambiguous.

- Unsupported C/C++ Constructs:
  - System calls.
  - Dynamic memory usage.
  - Recursive functions.

Source: https://docs.amd.com/r/2020.2-English/ug1399-vitis-hls/Vitis-HLS-Coding-Styles

Data type precision

- **Standard C Types**
  - Integers:
    - long long (64 bits)
    - int (32 bits)
    - Short (16 bits)
  - Characters:
    - char (8 bits)
  - Floating Point:
    - Float (32 bits)
    - Double (64 bits)
- **Arbitrary Precision Types**
  - C:
    - ap_(u)int
  - C++:
    - ap_(u)int
    - ap_fixed
  - C++ / SystemC:
    - sc_(u)int
    - sc_fixed

Hardware design: Directives/Optimizations

Hardware design - Directives/Optimizations

Minimize latency: UNROLL, LOOP_FLATTEN, LOOP_MERGE.
Minimize throughput: DATAFLOW, PIPELINE.
Improve bottleneck: RESOURCE, ARRAY_PARTITION, ARRAY_RESHAPE.

Hardware design - Directives/Optimizations

Minimize latency: UNROLL, LOOP_FLATTEN, LOOP_MERGE.
Minimize throughput: DATAFLOW, PIPELINE.
Improve bottleneck: RESOURCE, ARRAY_PARTITION, ARRAY_RESHAPE.

```
#pragma HLS UNROLL
#pragma HLS PIPELINE
#pragma HLS ARRAY_PARTITION variable=layer3_out complete dim=0
```

Hardware design - Directives/Optimizations

Minimize latency: UNROLL, LOOP_FLATTEN, LOOP_MERGE.
Minimize throughput: DATAFLOW, PIPELINE.
Improve bottleneck: RESOURCE, ARRAY_PARTITION, ARRAY_RESHAPE.

```
#pragma HLS UNROLL
#pragma HLS PIPELINE
#pragma HLS ARRAY_PARTITION variable=layer3_out complete dim=0
```

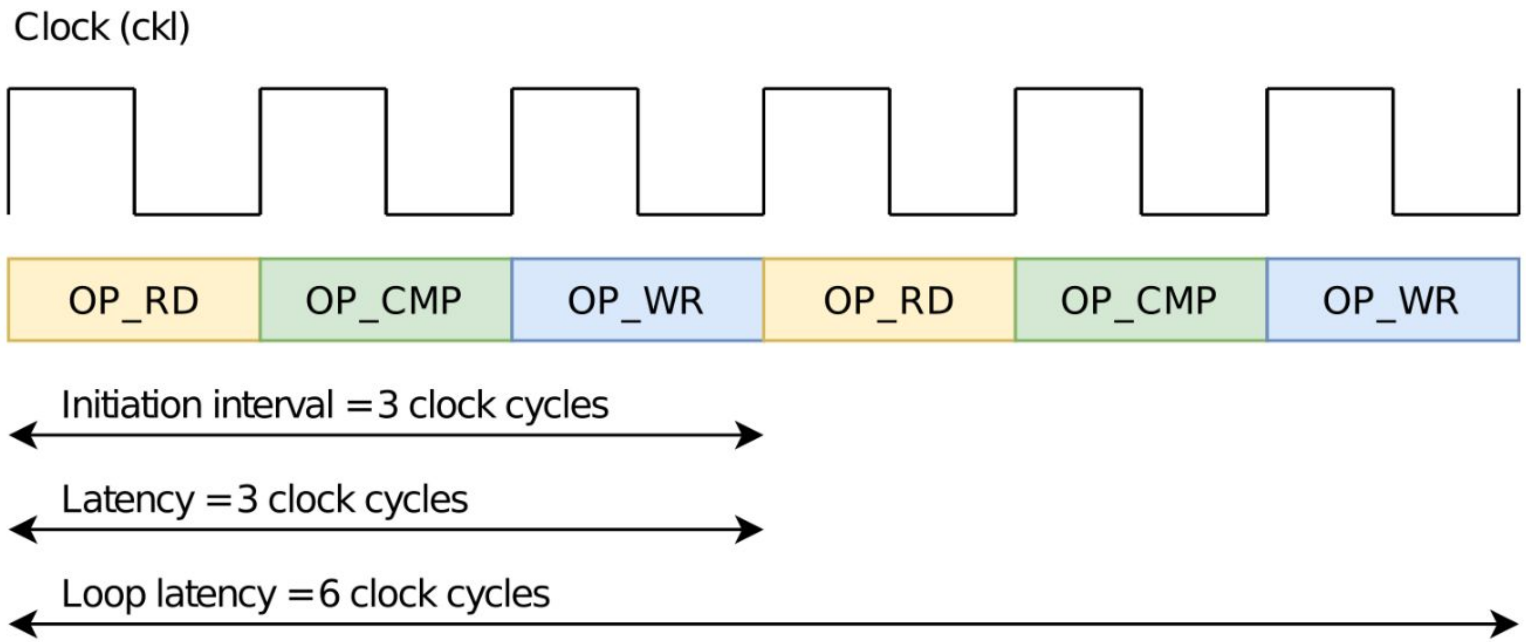
Source code: directives are included in the code.
Directive file: directives are specified in a separated file.

Hardware design - Directives/Optimizations

Loop handle

```
Loop_1: for(i=1; i<3; i++){
      OP_RD;
      OP_CMP;
      OP_WR;
}
```

OP_RD	Read
OP_CMP	Computation
OP_WR	Write



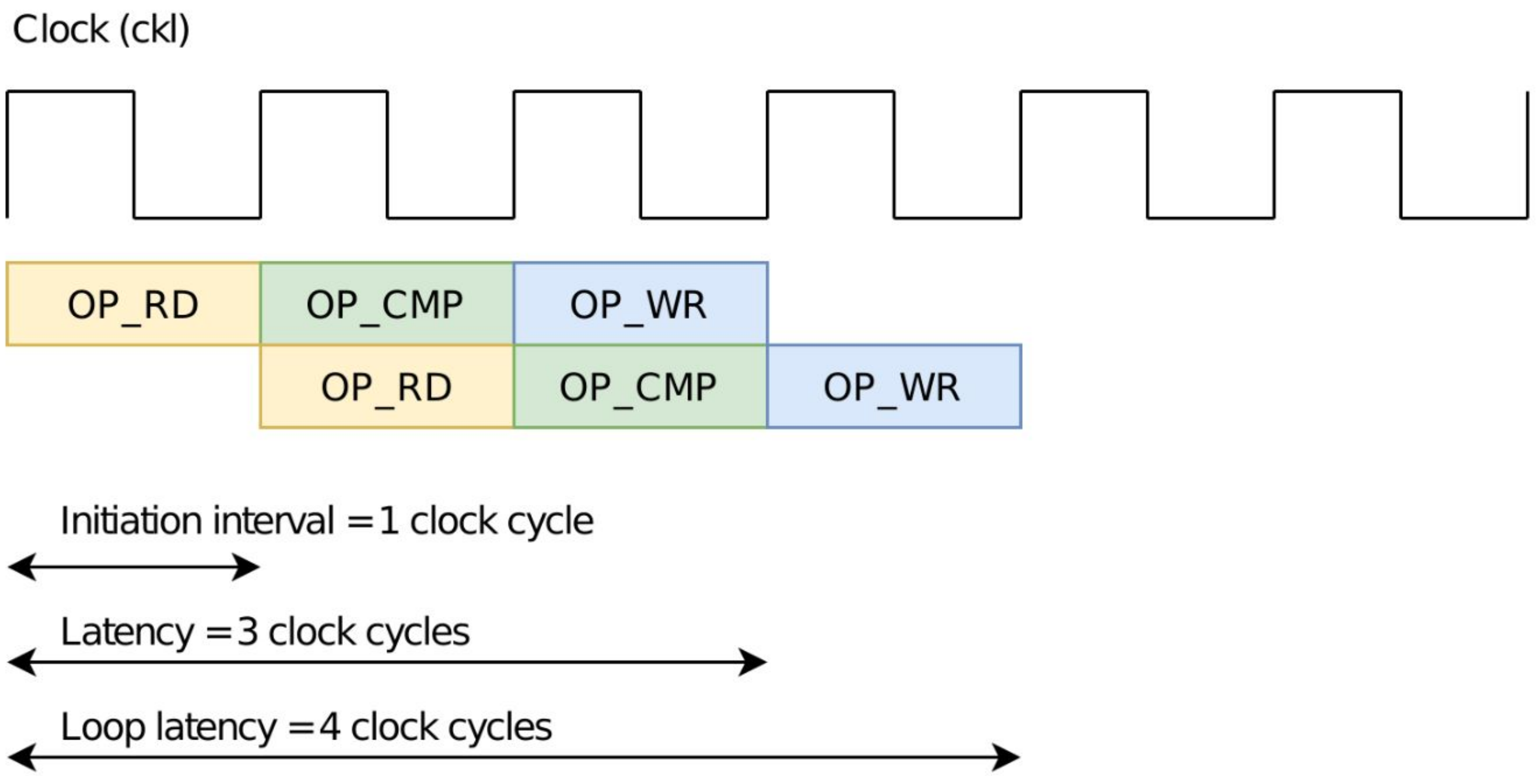
Hardware design - Directives/Optimizations

Loop handle

Loop + Pipeline

```
Loop_1: for(i=1; i<3; i++){
      OP_RD;
      OP_CMP;
      OP_WR;
}
```

OP_RD	Read
OP_CMP	Computation
OP_WR	Write



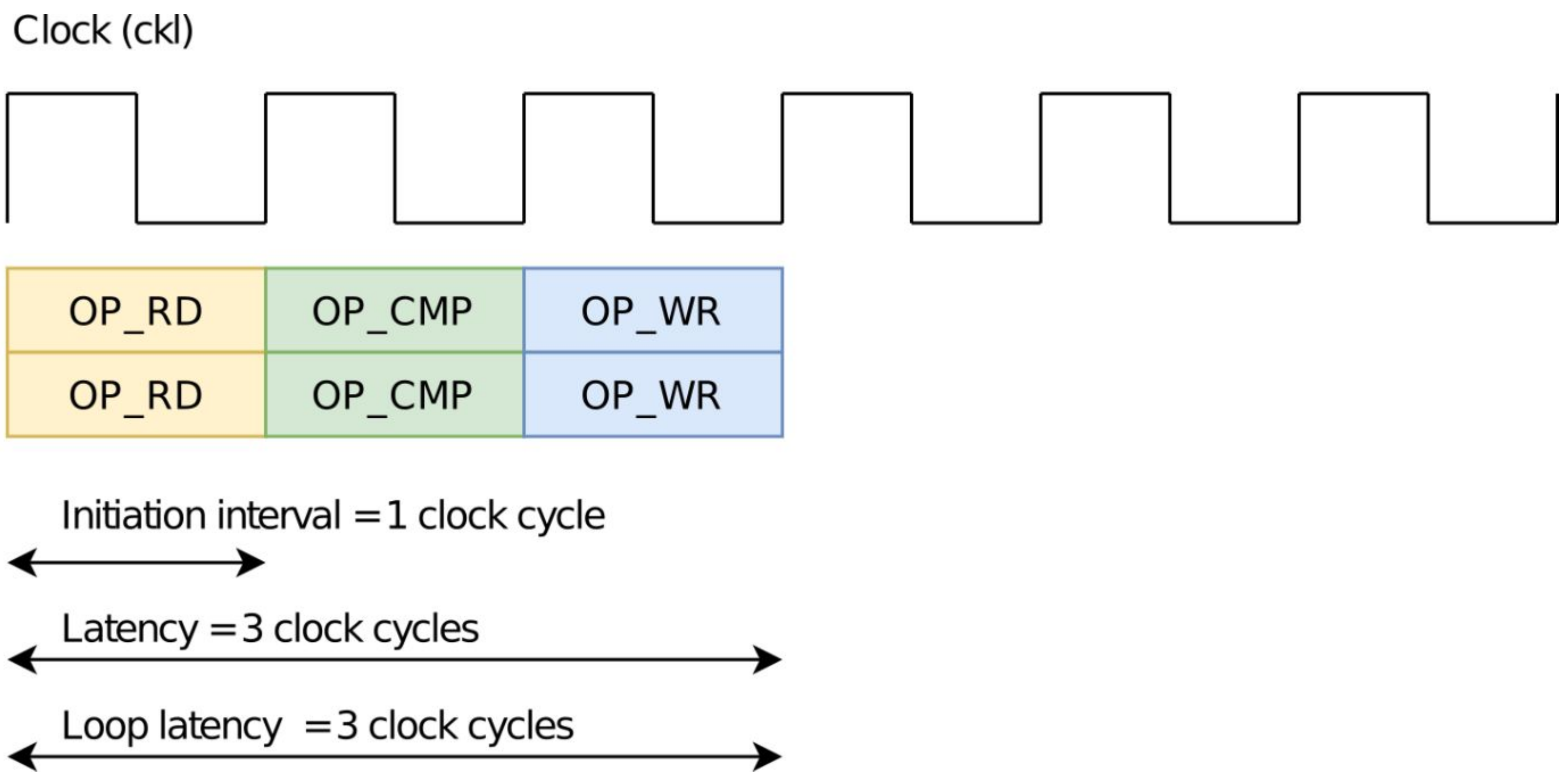
Hardware design - Directives/Optimizations

Loop handle

Loop + Unroll

```
Loop_1: for(i=1; i<3; i++){
      OP_RD;
      OP_CMP;
      OP_WR;
}
```

OP_RD	Read
OP_CMP	Computation
OP_WR	Write



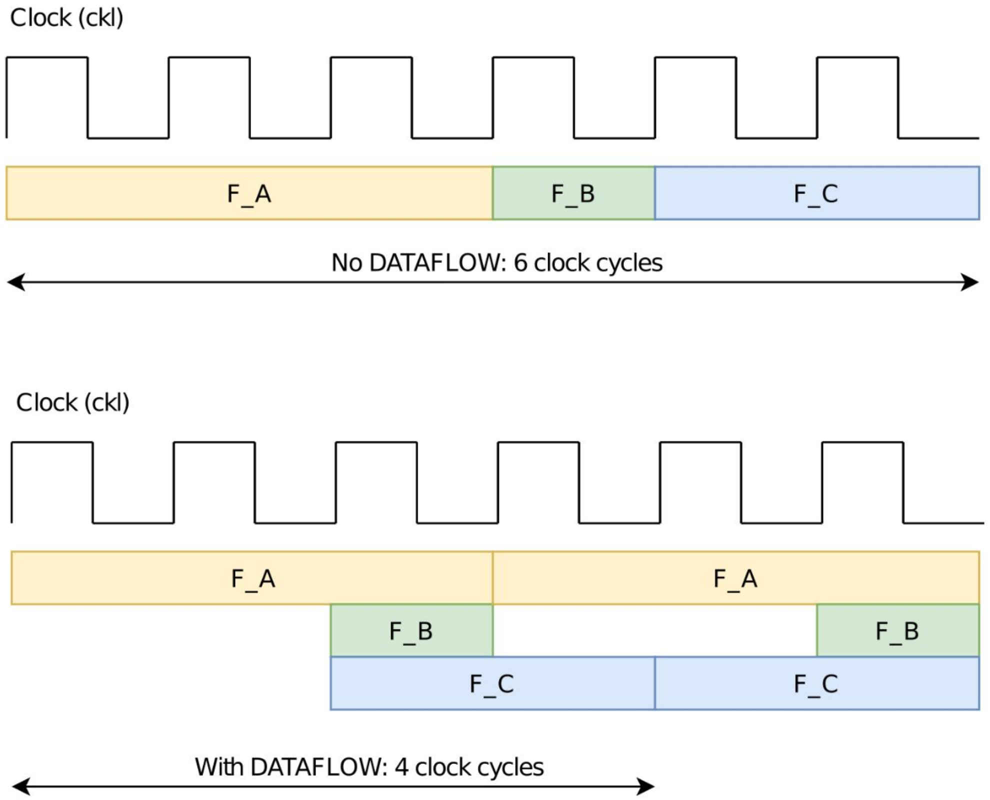
Hardware design - Directives/Optimizations

Loop handle

Loop + Dataflow

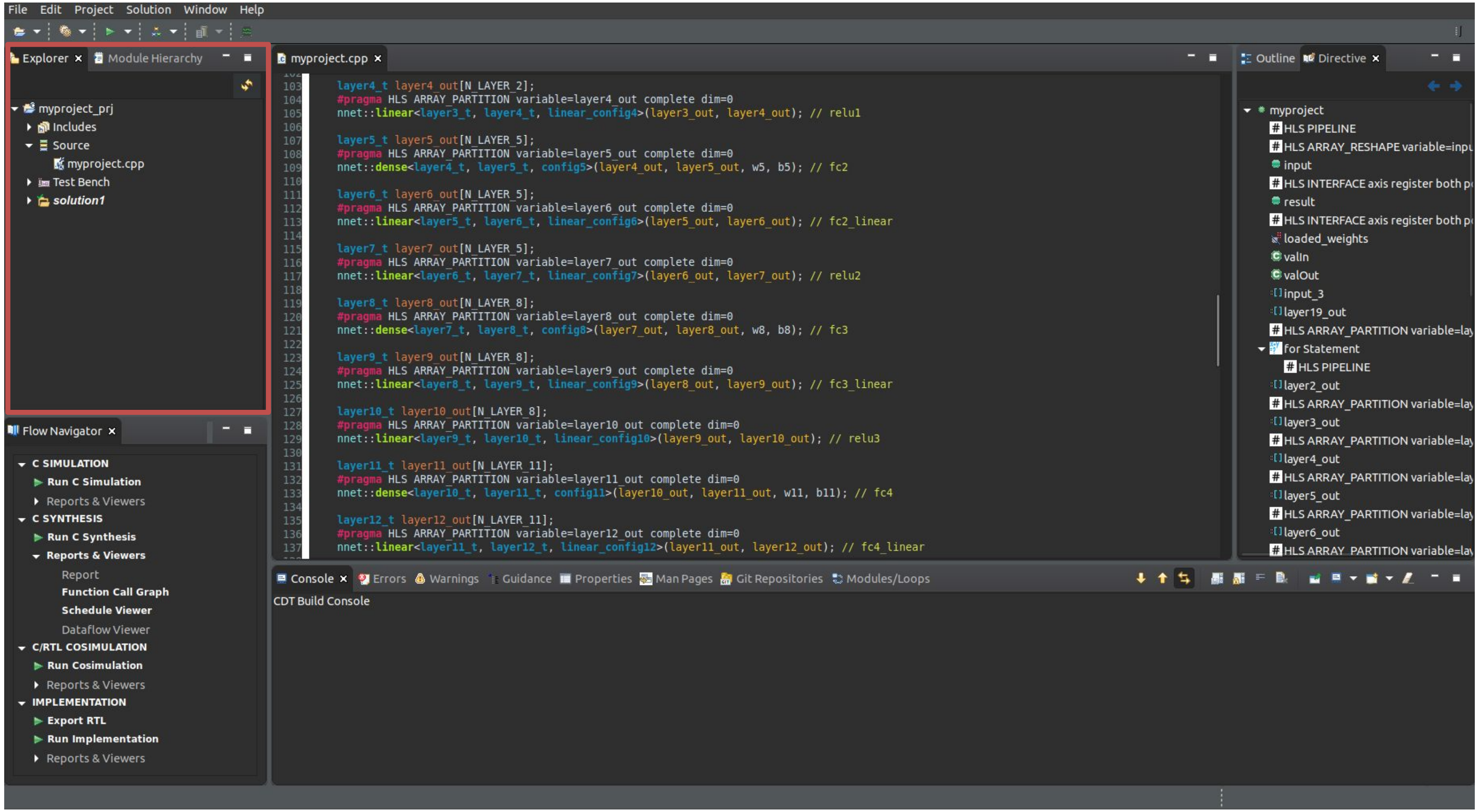
```
void function(a, b, c, d) {
     f_A(a, b, i1);
     f_B(c, i1, i2);
     f_C(i2, d);

return d;
}
```

F_A	Function f_A(a, b, i1)
F_B	Function f_B(c, i1, i2)
F_C	Function f_C(i2, d)

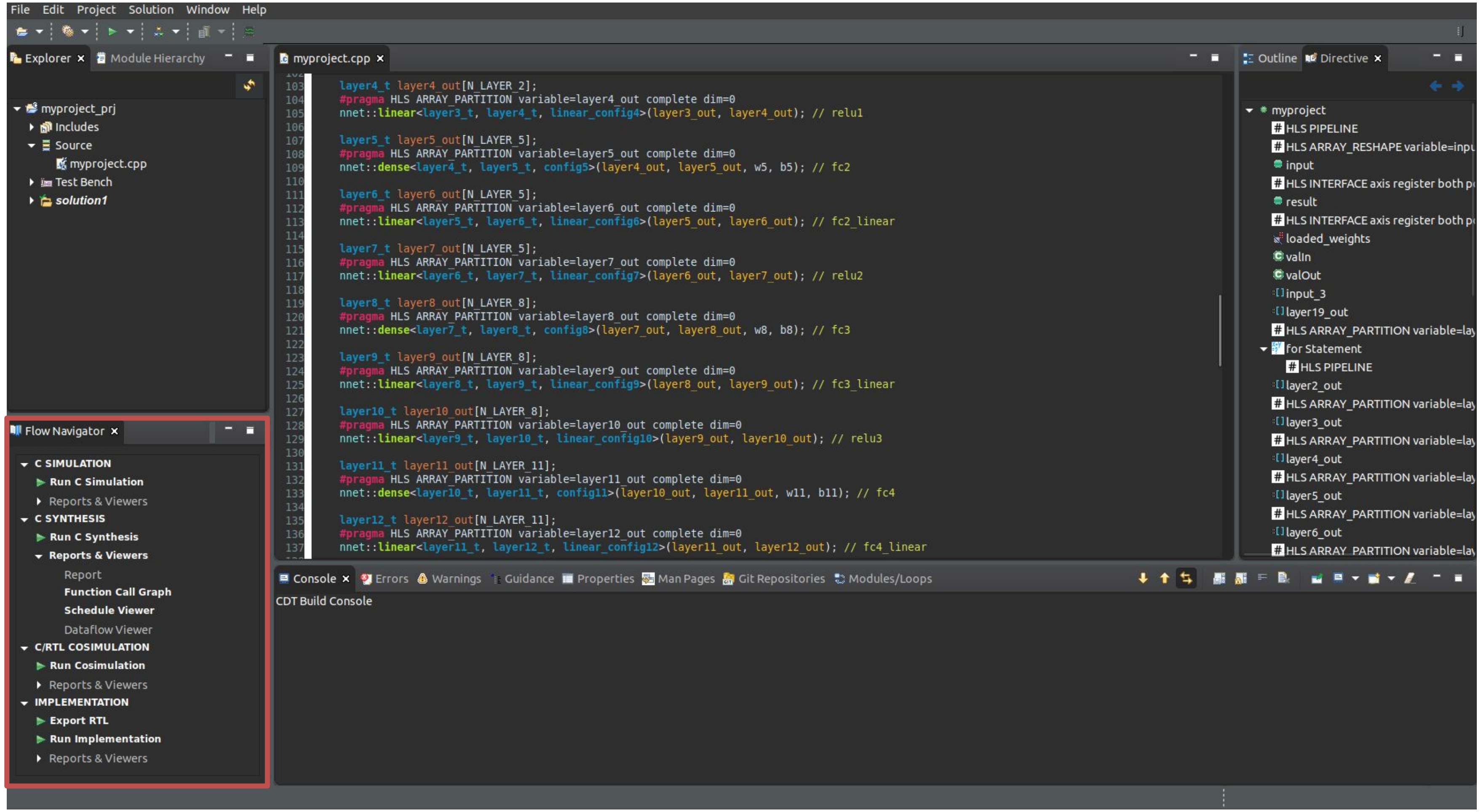


Vitis HLS GUI

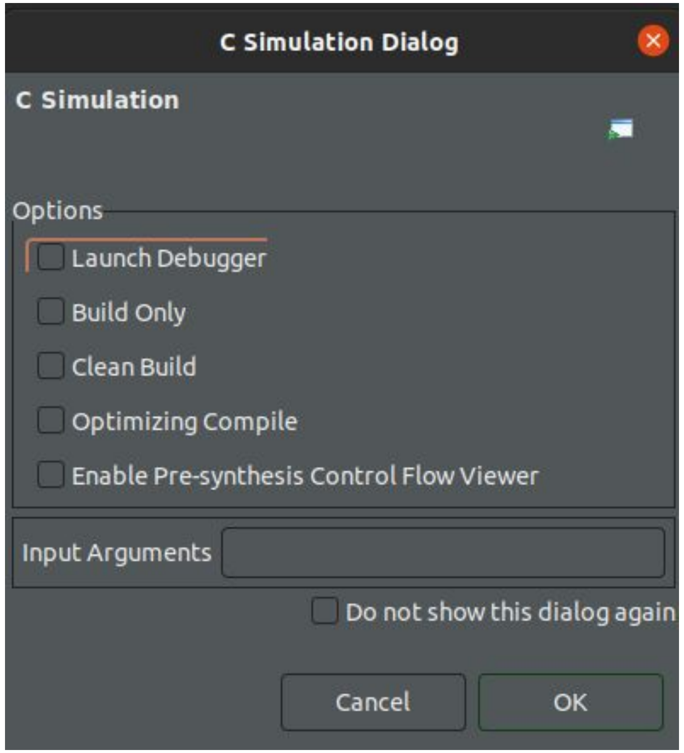
Vitis HLS GUI

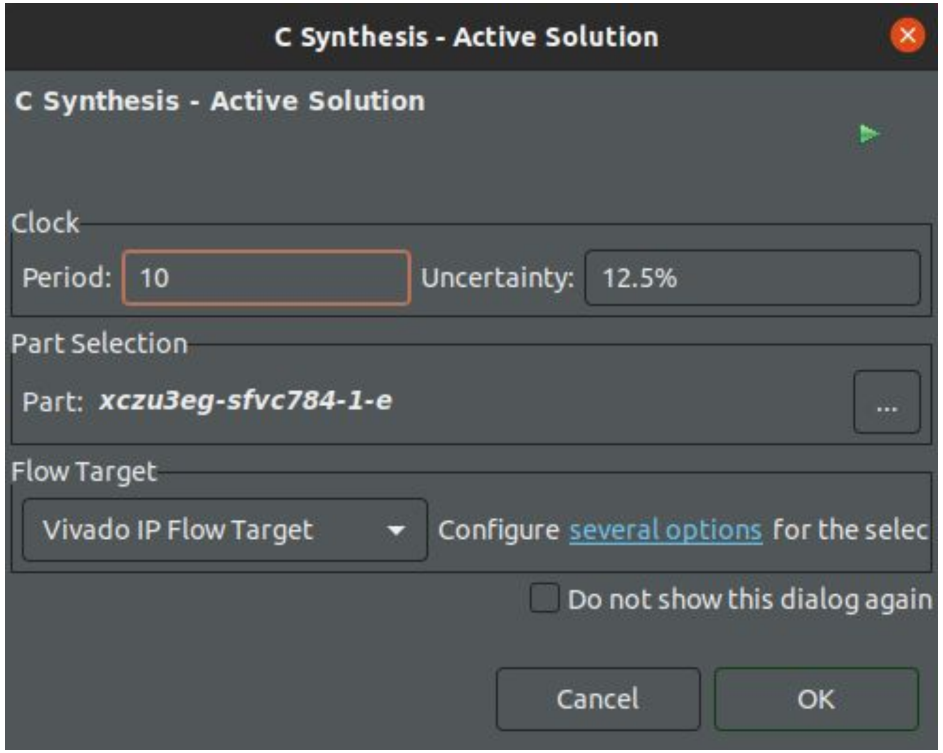


Vitis HLS GUI

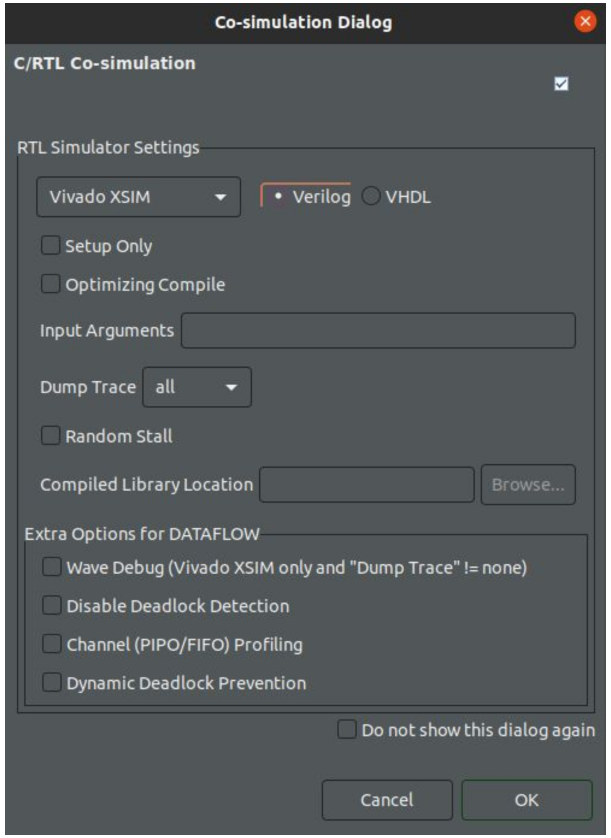


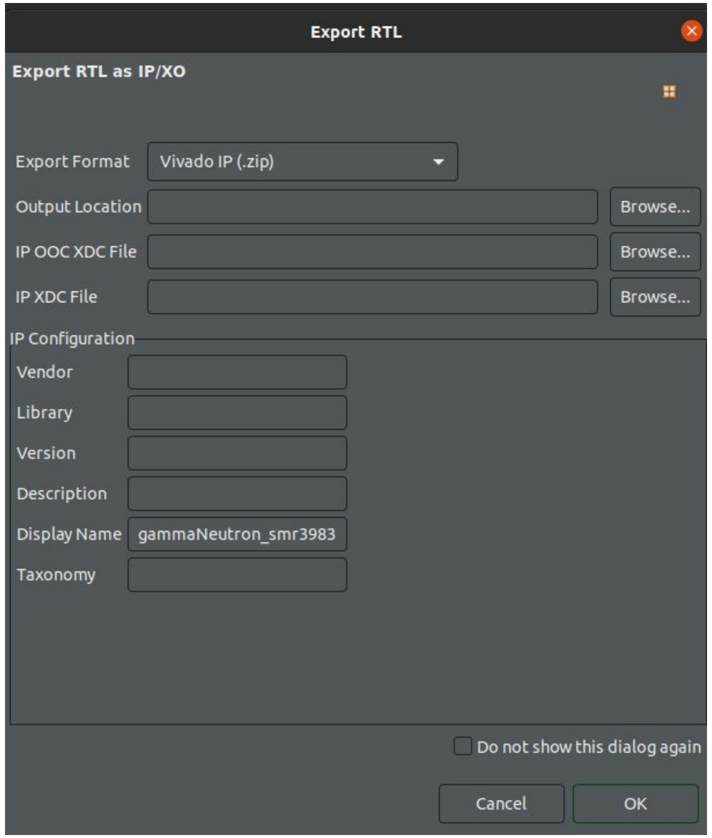
Vitis HLS GUI



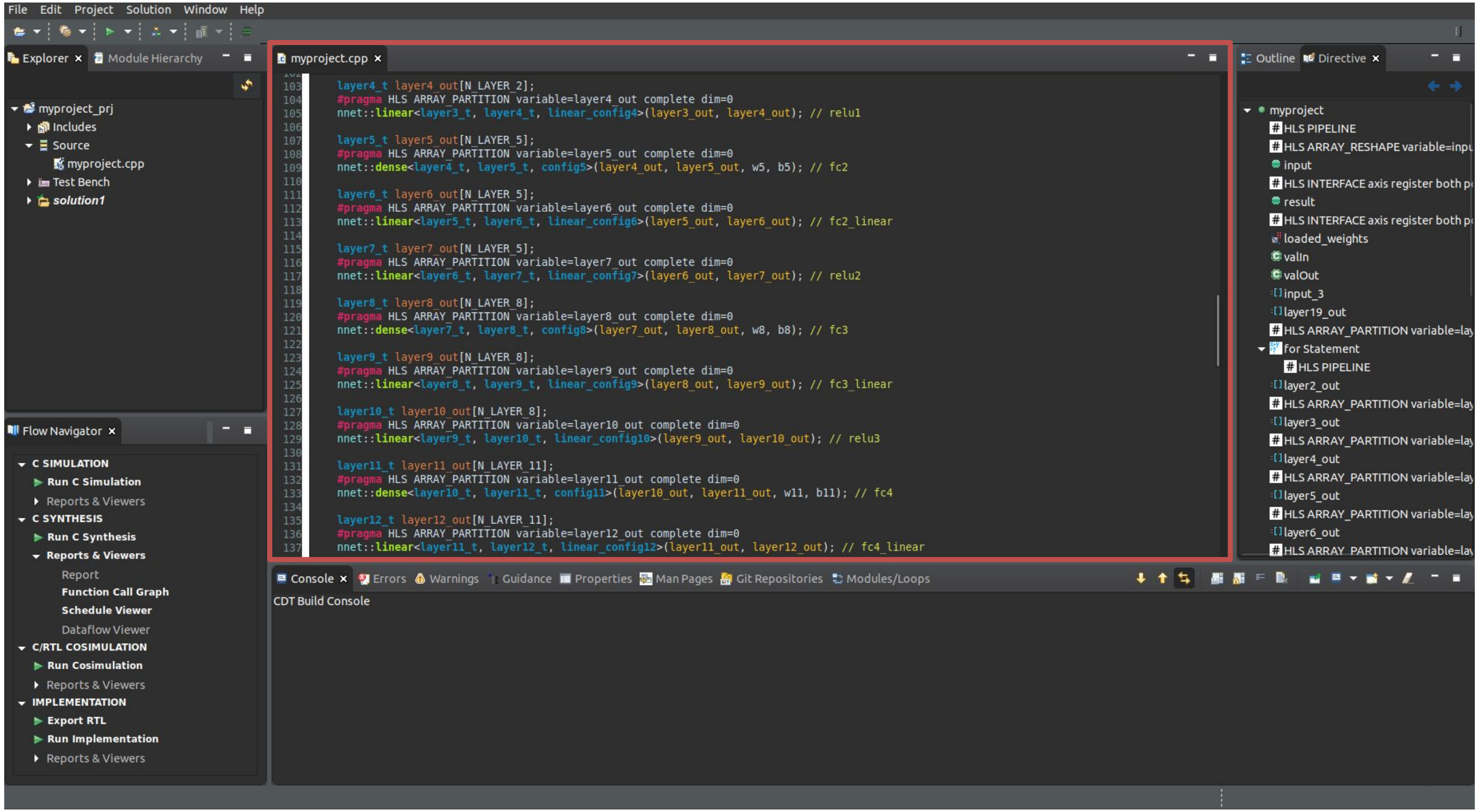


Vitis HLS GUI

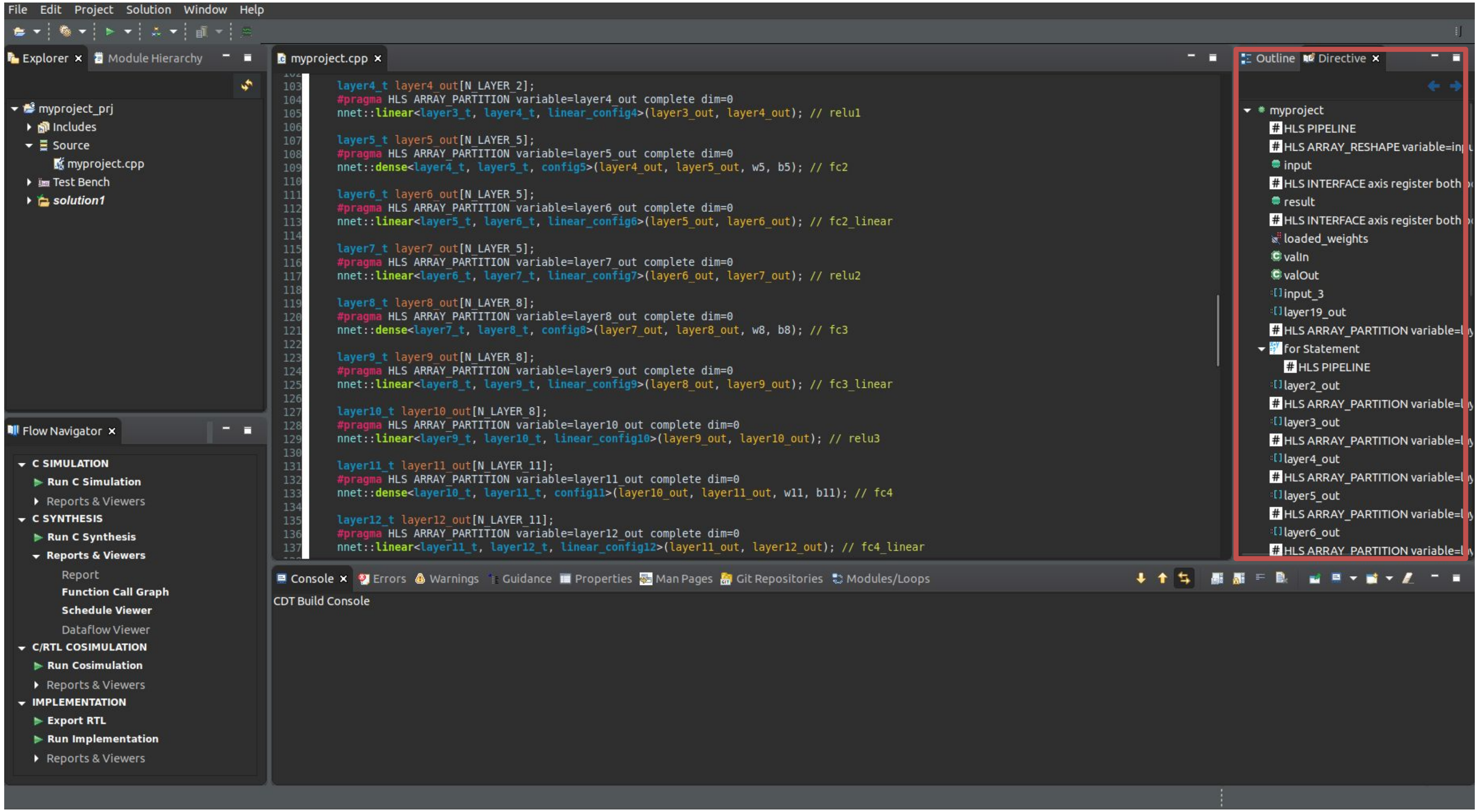




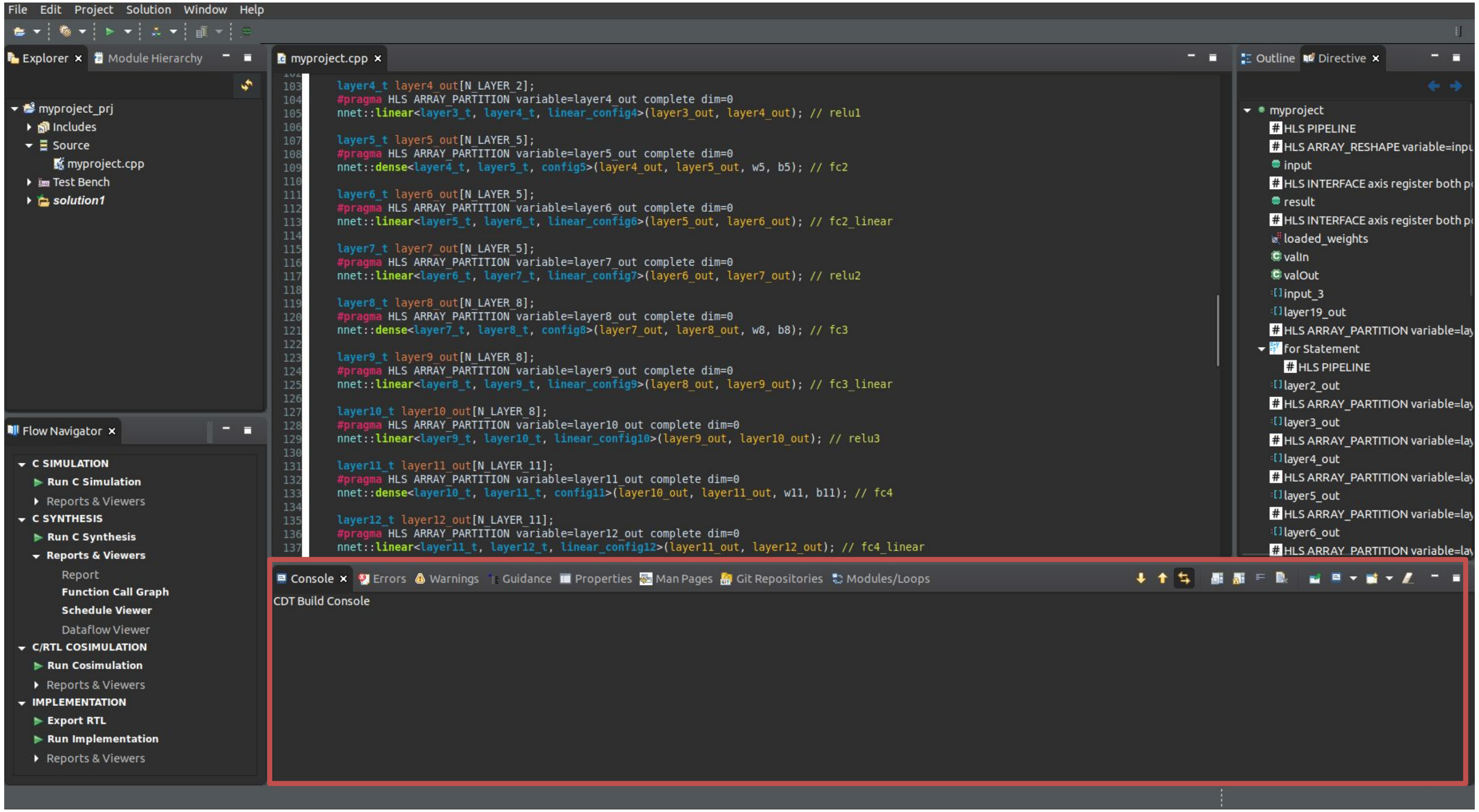
Vitis HLS GUI



Vitis HLS GUI

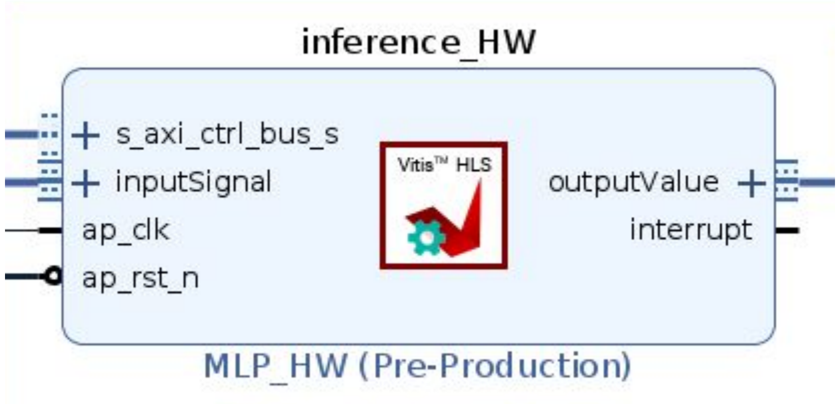


Vitis HLS GUI



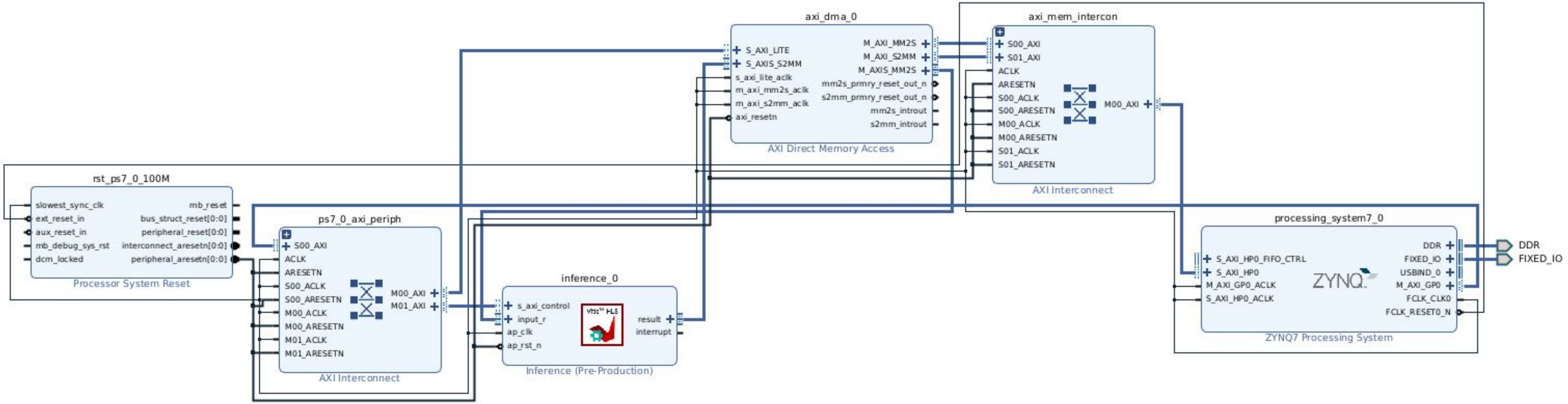
Vitis HLS GUI

Generated IP core



Vivado IP integration

Vivado IP integration



Demo:
HLS and Matrix Multiplication

Demo: HLS and Matrix Multiplication

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & \cdots & a_{11}b_{1n} + \cdots + a_{1n}b_{nn} \\ \vdots & \ddots & \vdots \\ a_{n1}b_{11} + \cdots + a_{nn}b_{n1} & \cdots & a_{n1}b_{1n} + \cdots + a_{nn}b_{nn} \end{bmatrix}$$

Thank you!

Romina Soledad Molina, Ph.D.
MLab-STI, ICTP

Mar del Plata, Argentina - 2025 -