



# *Joint ICTP-IAEA School on Systems-on-Chip Based on FPGA for Scientific Instrumentation and Reconfigurable Computing*

## High-level Synthesis

**Fernando Rincón**

*University of Castilla-La Mancha*  
*fernando.rincon@uclm.es*

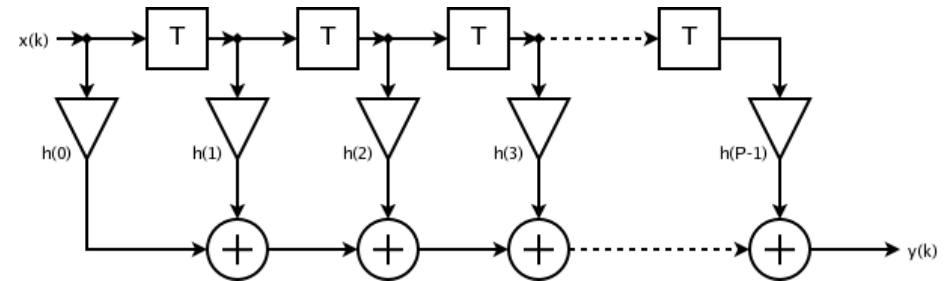


# Contents

- What is High-level Synthesis?
- Why HLS?
- How Does it Work?
- HLS Coding
- An example: Matrix Multiplication
  - Design analysis
- Validation Flow
- RTL Export
- IP Integration
- Software Drivers
- HLS Libraries

# Why HLS?

- Let's design a FIR filter
- First decisions:
  - Define the interface
    - types for  $x$ ,  $y$  and  $h$
    - $h$  provided through a ROM, a register file?
  - Define the architecture:
    - Finite state machine
      - Number of states
    - Datapath
      - Type of multipliers and adders (latencies may affect number of states)
      - Bit-size of the resources
- Then write RTL code (Verilog or VHDL)
- And also a RTL testbench



# Why HLS?

One possible implementation

Data types and structure can  
Be generalized up to a certain  
point

Operations are assumed to be  
Solved in one clock cycle

I/O interface should later be wrapped  
for the appropriate bus

The design choice is already made

```
ARCHITECTURE behavior OF fir_filter IS
  SIGNAL coeff_int      : coefficient_array;
  SIGNAL data_pipeline  : data_array;
  SIGNAL products       : product_array;
BEGIN

  PROCESS(clk, reset_n)
    VARIABLE sum : SIGNED((data_width + coeff_width +
                          integer(ceil(log2(real(taps)))) - 1) DOWNTO 0);
  BEGIN

    IF(reset_n = '0') THEN

      data_pipeline <= (OTHERS => (OTHERS => '0'));
      coeff_int <= (OTHERS => (OTHERS => '0'));
      result <= (OTHERS => '0');

    ELSIF(clk'EVENT AND clk = '1') THEN

      coeff_int <= coefficients;
      data_pipeline <= SIGNED(data) & data_pipeline(0 TO taps-2);

      sum := (OTHERS => '0');
      FOR i IN 0 TO taps-1 LOOP
        sum := sum + products(i);
      END LOOP;

      result <= STD_LOGIC_VECTOR(sum);

    END IF;
  END PROCESS;

  product_calc: FOR i IN 0 TO taps-1 GENERATE
    products(i) <= data_pipeline(i) * SIGNED(coeff_int(i));
  END GENERATE;

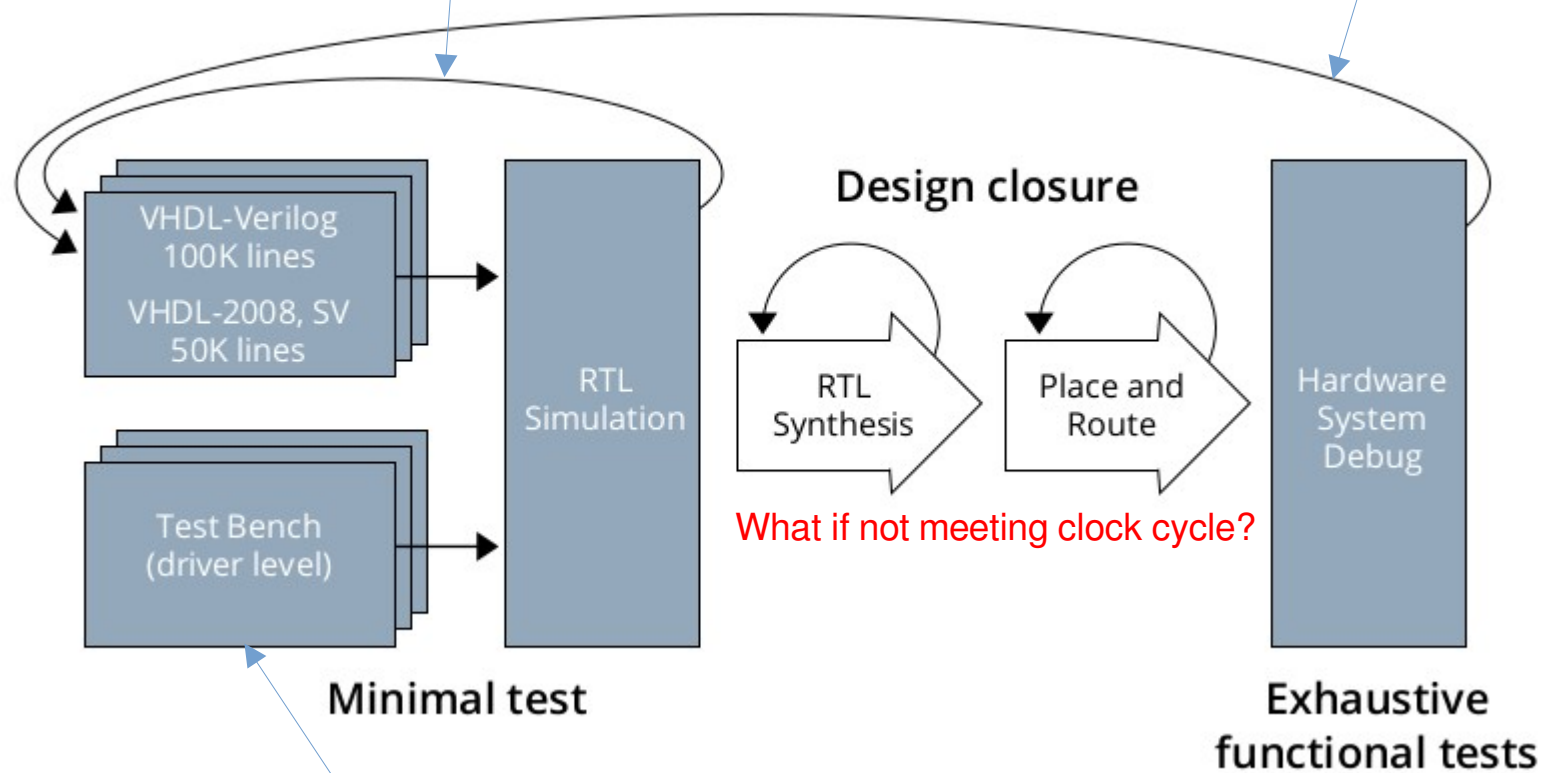
END behavior;
```

# Why HLS?

Costly architecture redesign

Even more costly. High impact in Design time

## Traditional RTL Design Flow



What if not meeting clock cycle?

What if I want to integrate the FIR using another Bus interface?

RTL tests are hard to write

# What is High-level Synthesis?

- Compilation of behavioral algorithms into RTL descriptions

## *Input*

### **Behavioral description:**

- Algorithm

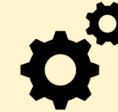
### **Constraints:**

- I/O description
- Timing
- Memory



### **High Level Synthesis:**

- Microarchitecture evaluation
- FSM extraction
- Operations & datapath extraction
- Interface synthesis



## *Output*

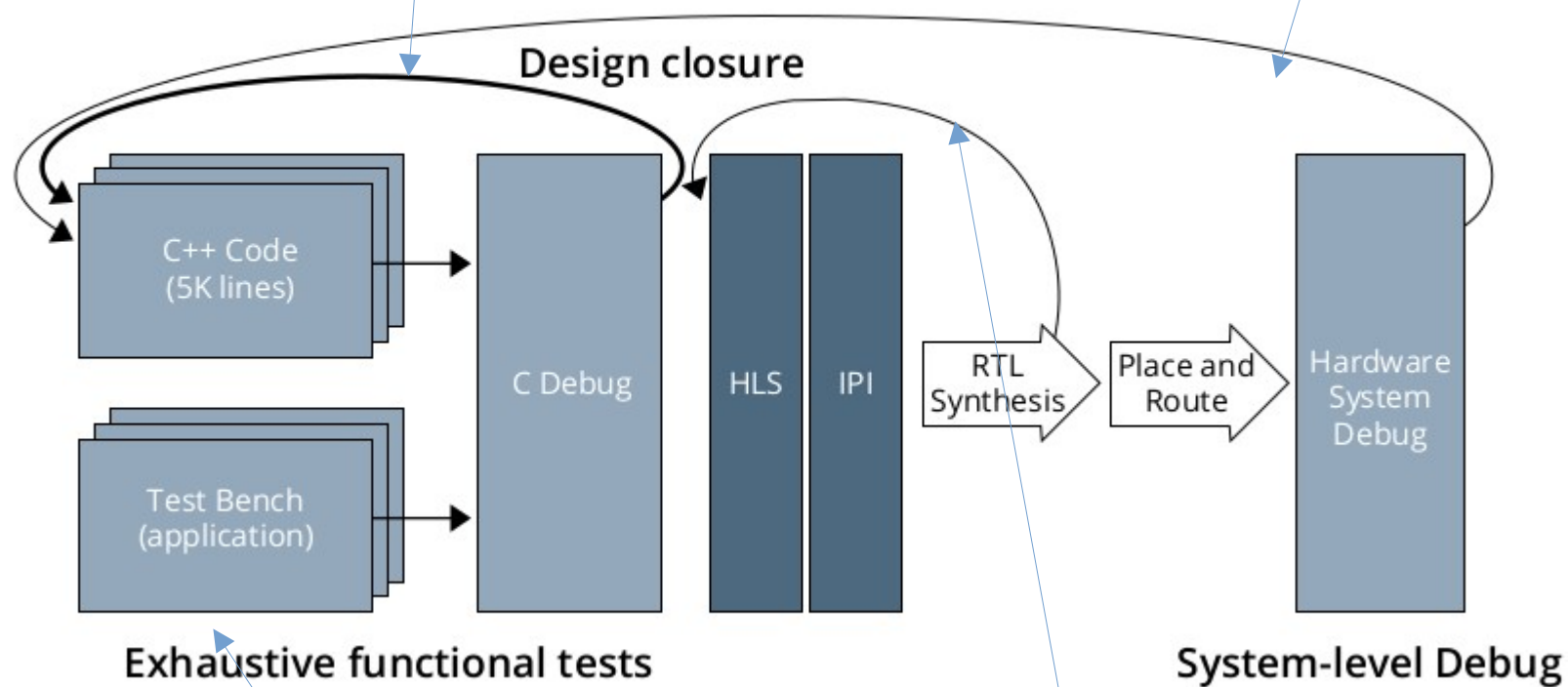
**RTL IP**

# Why HLS?

Standard debug tasks.  
Focused in algorithm not architecture

Always costly, but much less

## Vivado HLx Design Flow



Exhaustive functional tests

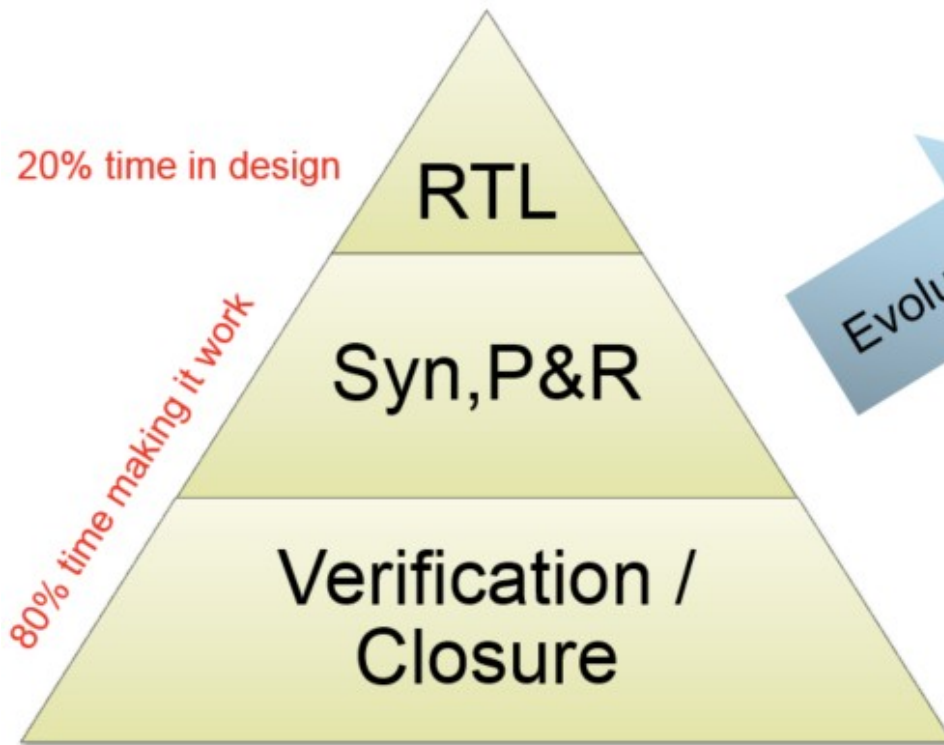
System-level Debug

Same C test for all stages

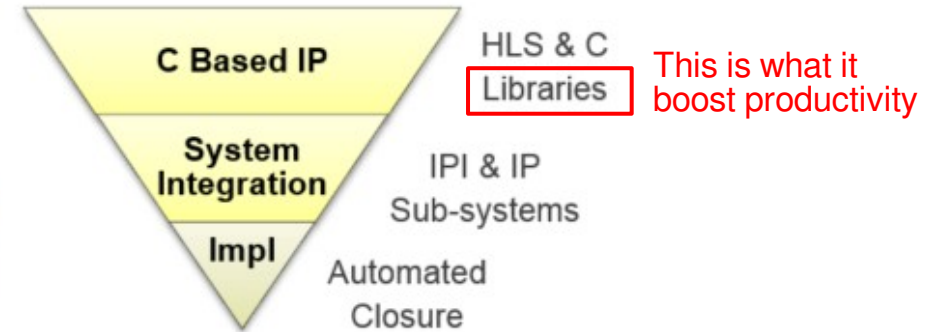
Solution optimization through directives.  
Fast design space exploration

# Why HLS?

## Vivado RTL-Based Design

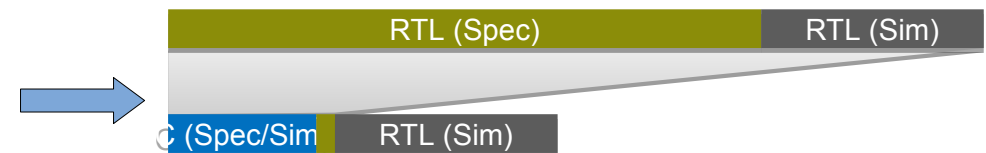


## Vivado C and IP-Based Design



First Design	10X-15X Faster
Derivative Design	40X Faster
Typical QoR	0.7 – 1.2X

Video Design Example			
Input	C Simulation Time	RTL Simulation Time	Improvement
10 frames 1280x720	10s	~2 days (ModelSim)	~12000x





# Why HLS?

- Need for **productivity boosting** at design level
  - Fast Design Space Exploration
  - Reduce Time-to-market
  - Trend to use FPGAs as Hw accelerators
- Electronic System Level Design is based in
  - Hw/Sw Co-design
    - SystemC / SystemVerilog / C++
    - Transaction-Level Modelling
  - One common C-based description of the system
  - Iterative refinement
  - Integration of models at a very different level of abstraction
  - **But need an efficient way to get to the silicon (HLS)**
- Rising the level of abstraction enables Sw programmers to have access to silicon

# HLS Benefits

- **Design Space Exploration**
  - Early estimation of main design variables: latency, performance, consumption
    - Would imply endless recoding in VHDL or Verilog
  - Can be targeted to different technologies
- **Verification**
  - Reuse of C-based testbenches
  - Can be complemented with formal verification
- **Reuse**
  - Higher abstraction provides better reuse opportunities
  - Cores can be exported to different bus technologies
  - Vitis HLS provides a number of HLS libraries:
    - Vision, finances, hpc, ...

# 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 :

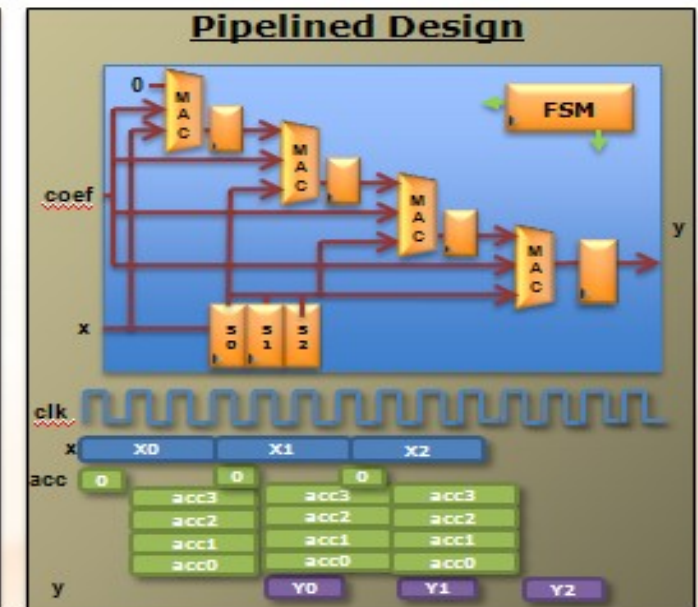
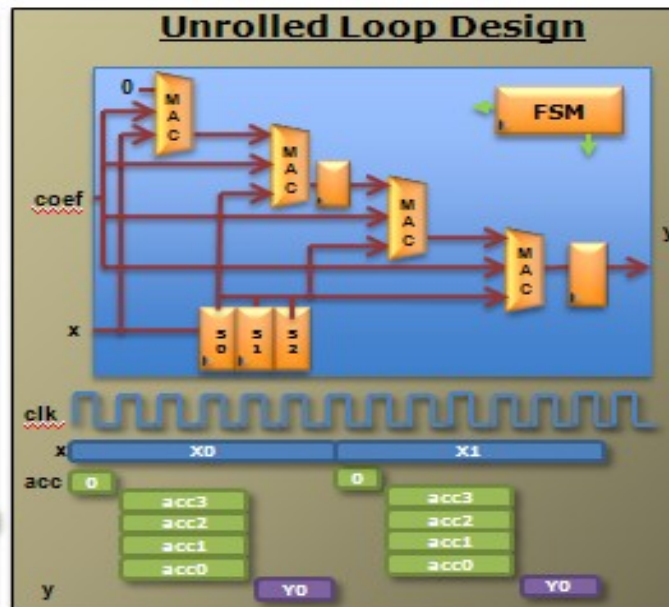
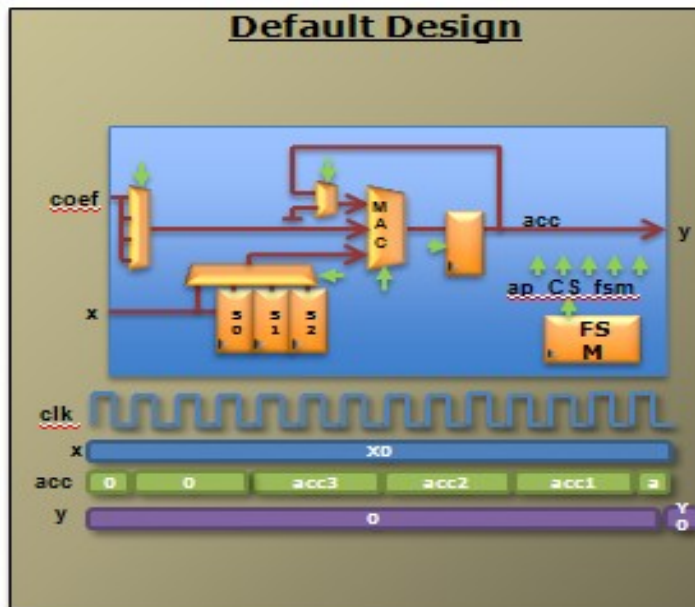
- Small area
- Long latency
- Low throughput

Different hardware for each loop iteration :

- Higher area
- Short latency
- Better throughput

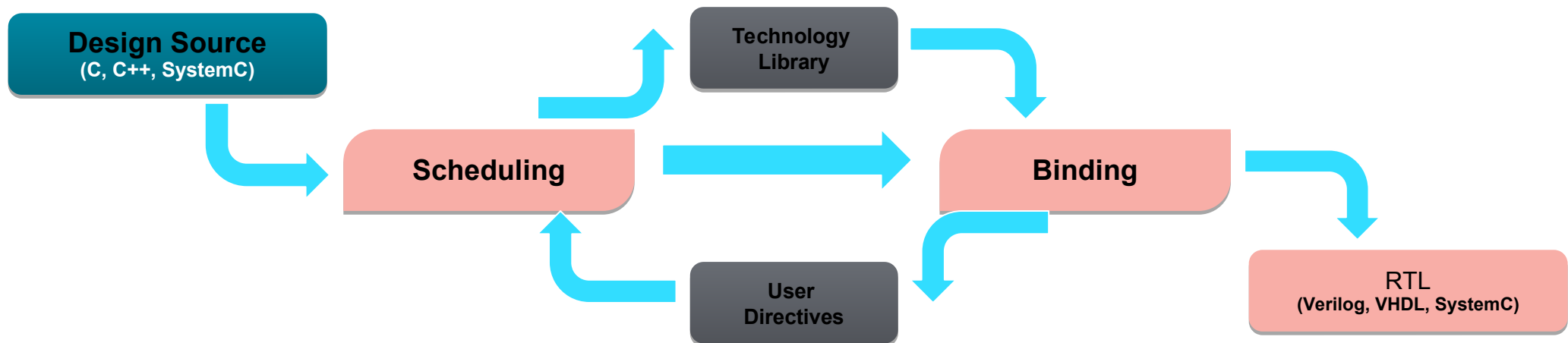
Different iterations executed concurrently:

- Higher area
- Short latency
- Best throughput



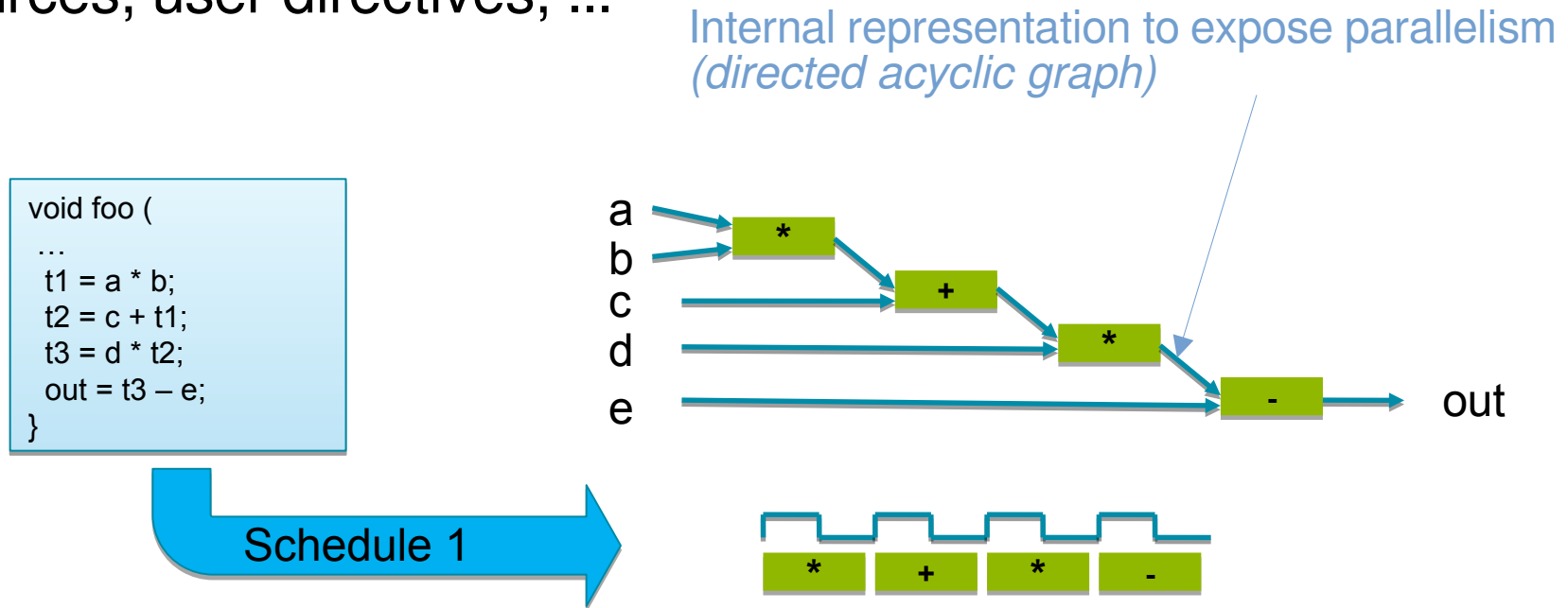
# How Does it Work? - Scheduling & Binding

- 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, ...



# How Does it Work? - Scheduling

- Operations are mapped into clock cycles, depending on timing, resources, user directives, ...

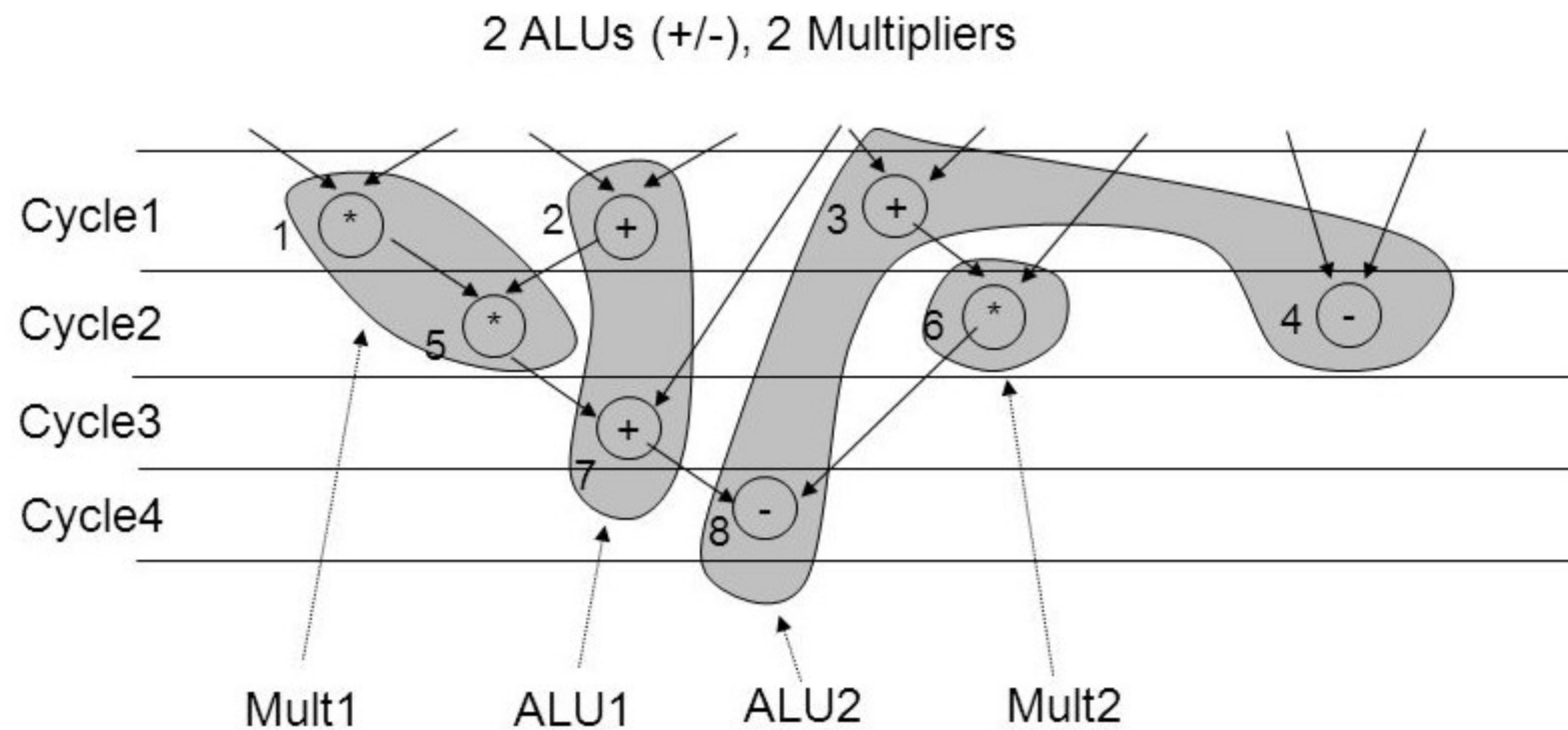


When a faster technology or slower clock ...



# How Does it Work? - Allocation & Binding

Operations are assigned to available functional units in the library



# How Does it Work? - Control Extraction

## 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;  
}
```

FIR C code example ..

Function Start

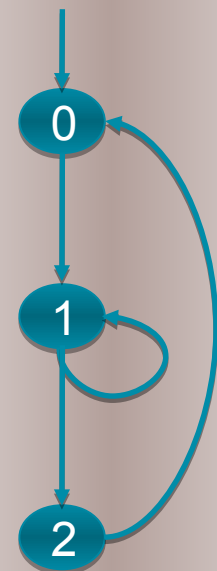
For-Loop Start

For-Loop End

Function End

## Control Behavior

Finite State Machine (FSM) states



The loops in the C code correlated to states of behavior

This behavior is extracted into a hardware state machine

# How does it work? - Datapath Extraction

## Code

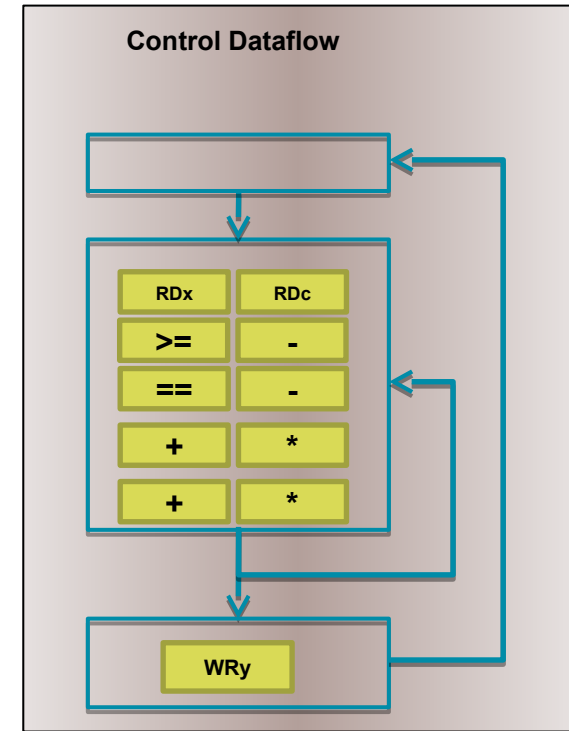
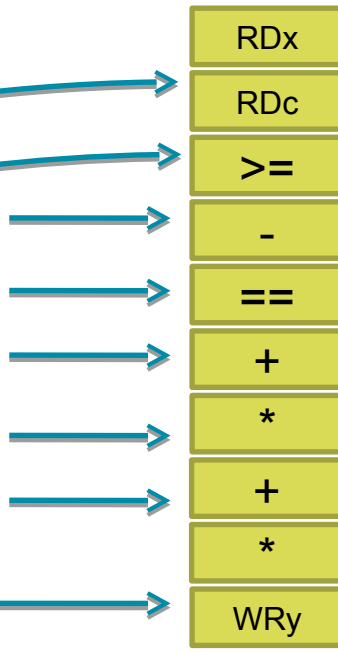
## Operations

## Control & Datapath Behavior

```
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;
}
```



FIR C code example ..

Operations are extracted...

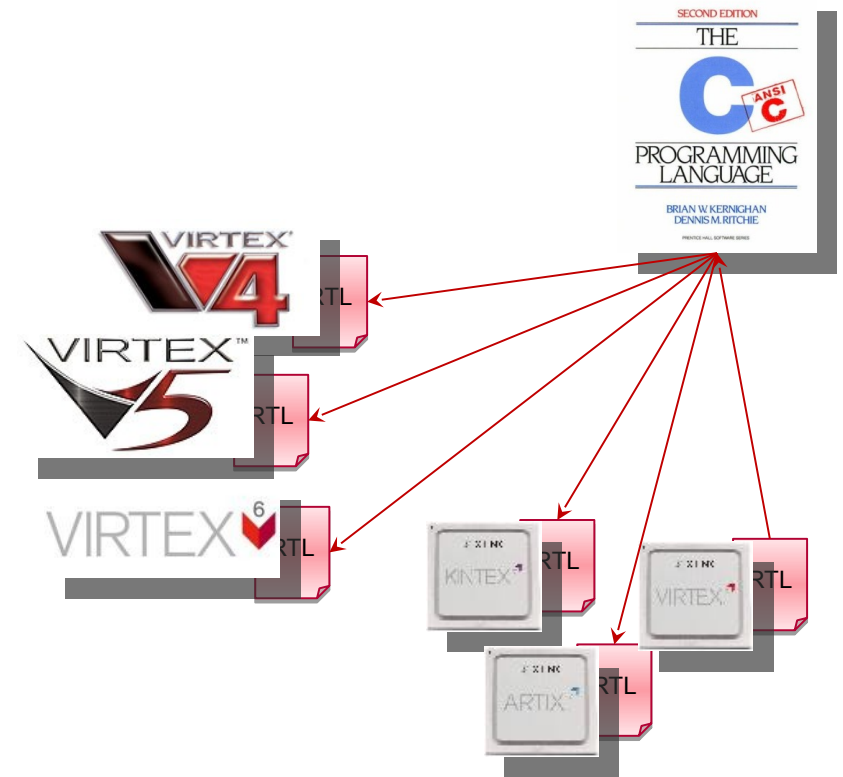
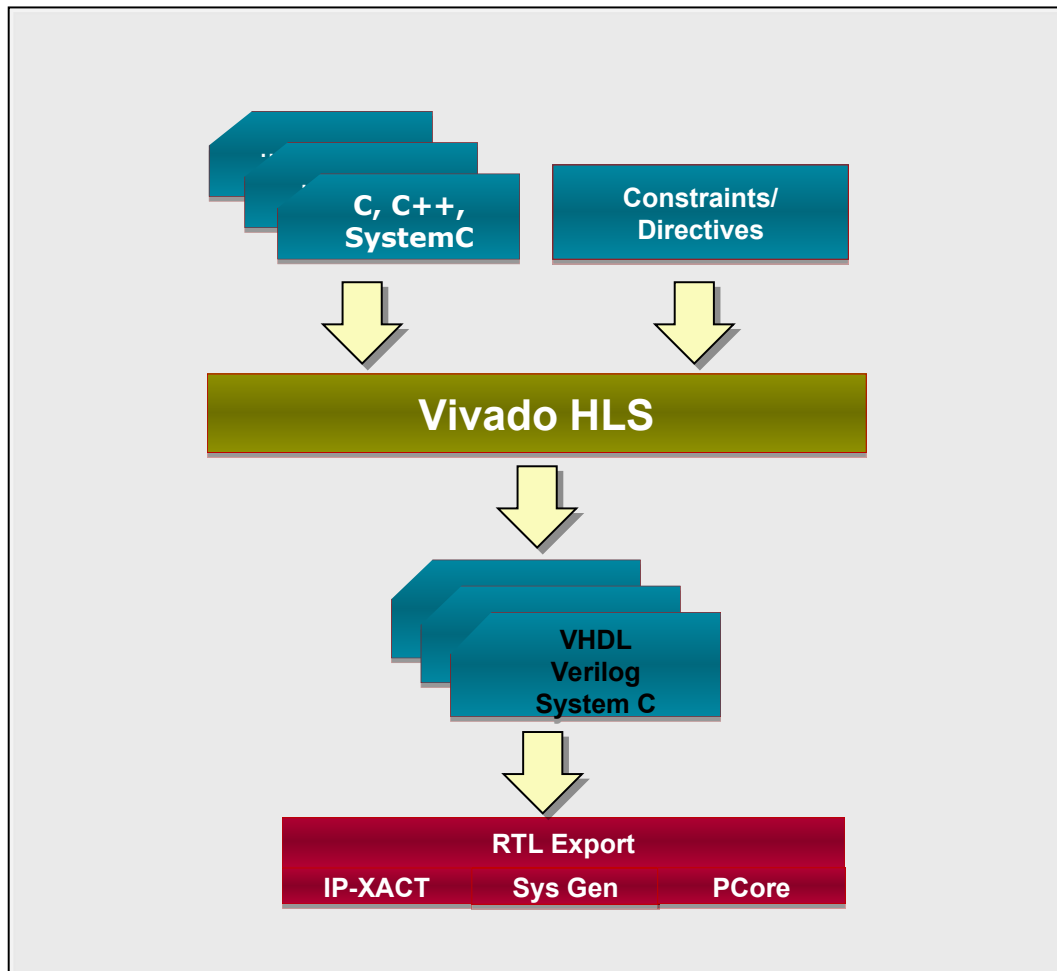
A unified control dataflow behavior is created.

*Scheduling + Binding*



# Vitis HLS

- High-level Synthesis Suite from Xilinx



# Source Code: Language Support

- Vivado HLS supports C, C++, SystemC and OpenCL API C kernel
  - Provided it is statically defined at compile time
  - Default extensions: .c for C / .cpp for C++ & SystemC
- Modeling with bit-accuracy
  - Supports arbitrary precision types for all input languages
  - Allowing the exact bit-widths to be modeled and synthesized
- Floating point support
  - Support for the use of float and double in the code
- Support for OpenCV functions
  - Enable migration of OpenCV designs into Xilinx FPGA
  - Libraries target real-time full HD video processing

# Source Code: Key Attributes

- Only one top-level function is allowed

```
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:** Represent the design hierarchy

**Top Level IO :** Top-level arguments determine Interface ports

**Types:** Type influences area and performance

**Loops:** Their scheduling has major impact on area and performance

**Arrays:** Mapped into memory. May become main performance bottlenecks

**Operators:** Can be shared or replicated to meet performance

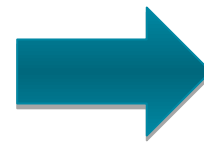
# Functions & RTL Hierarchy

- Each function is translated into an RTL block.
- Can be shared or inlined (dissolved)

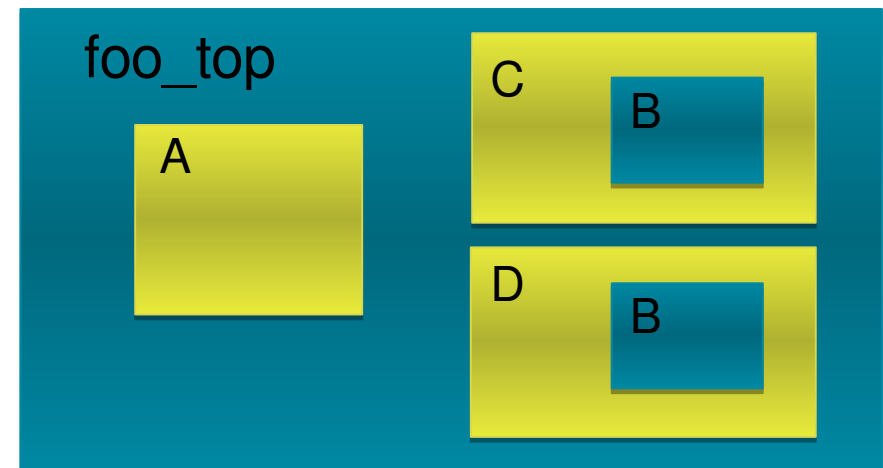
## Source Code

```
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



# Operator Types

- They define the size of the hardware used
- **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` => (1-1024)
  - C++:
    - `ap_(u)int` => (1-1024)
    - `ap_fixed`
  - C++ / SystemC:
    - `sc_(u)int` => (1-1024)
    - `sc_fixed`

# Loops

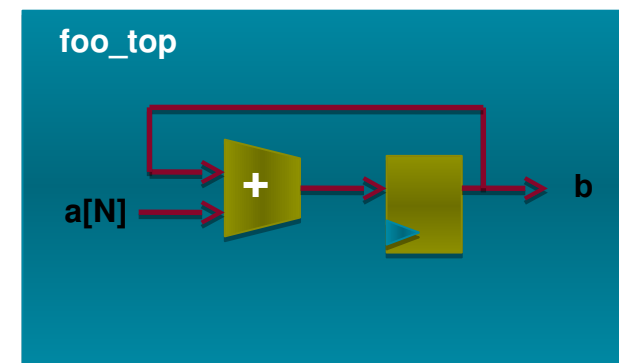
- **Rolled by default**

- Each iteration implemented in the **same state**
- Each iteration implemented with the **same resources**



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

Synthesis



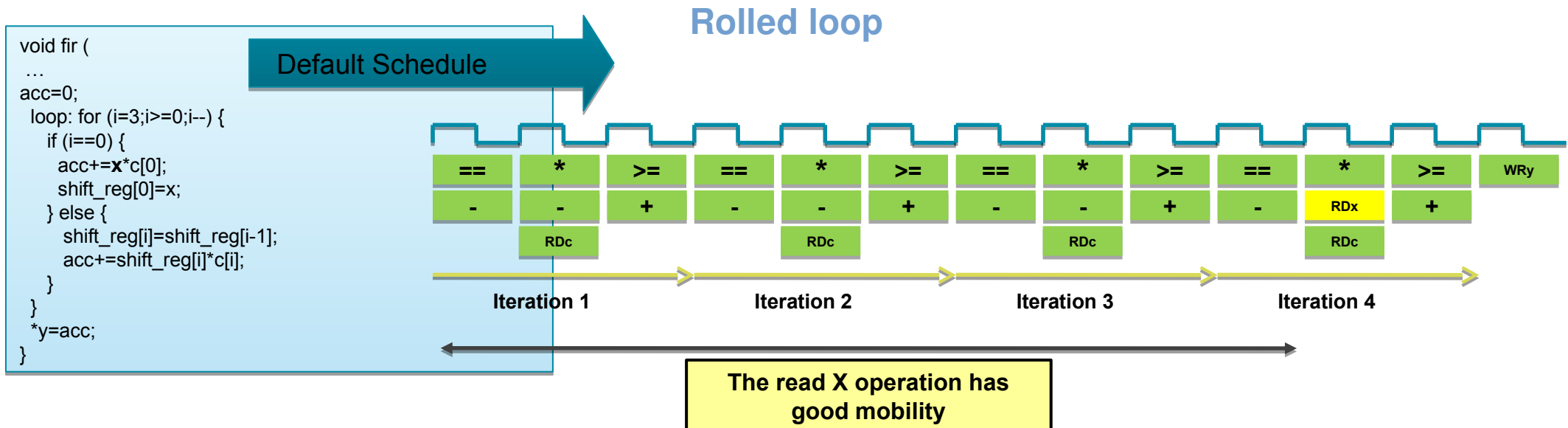
- **Loops can be unrolled** if their indexes are **statically determinable** at elaboration time

- Not when the number of iterations is variable
- Result in more elements to schedule but **greater operator mobility**

# Data Dependencies: Good

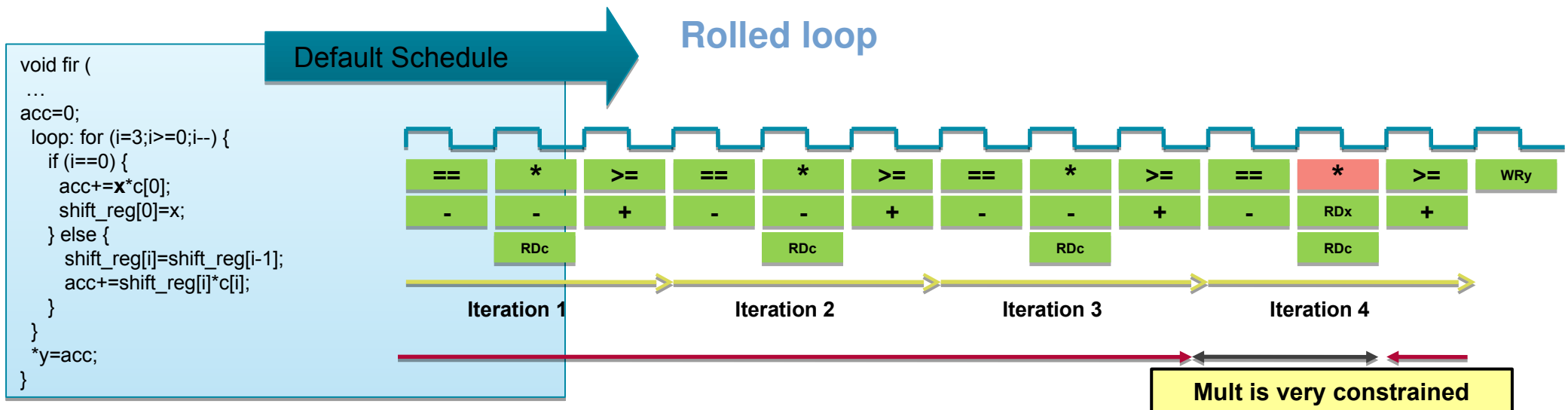
- Example of good mobility

- The read on data port X can occur anywhere from the start to iteration 4
  - The only constraint on RDx is that it occur before the final multiplication
- Vivado HLS has a lot of freedom with this operation
  - It waits until the read is required, saving a register
  - Input reads can be optionally registered



# Data Dependencies: Bad

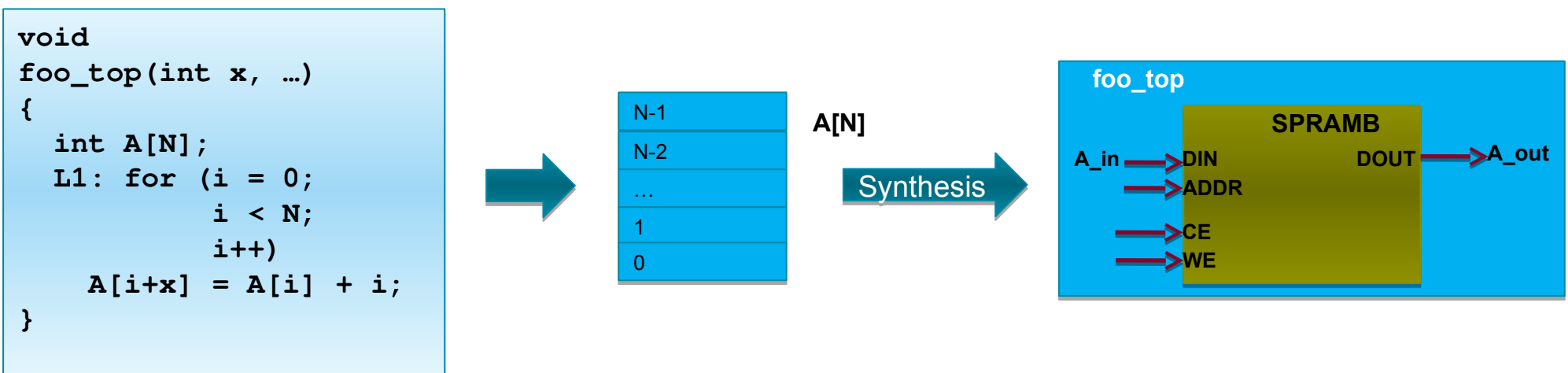
- The final multiplication must occur before the read and final addition
- Loops are rolled by default
  - Each iteration cannot start till the previous iteration completes
  - The final multiplication (in iteration 4) must wait for earlier iterations to complete
- The structure of the code is forcing a particular schedule
  - There is little mobility for most operations





# Arrays

- By default implemented as RAM
  - Dual port if performance can be improved otherwise Single Port RAM
  - optionally as a FIFO or registers bank
- Can be targeted to any memory resource in the library
- Can be merged with other arrays and reconfigured
- Arrays can be partitioned into individual elements
  - Implemented as smaller RAMs or registers



# Top-Level IO Ports

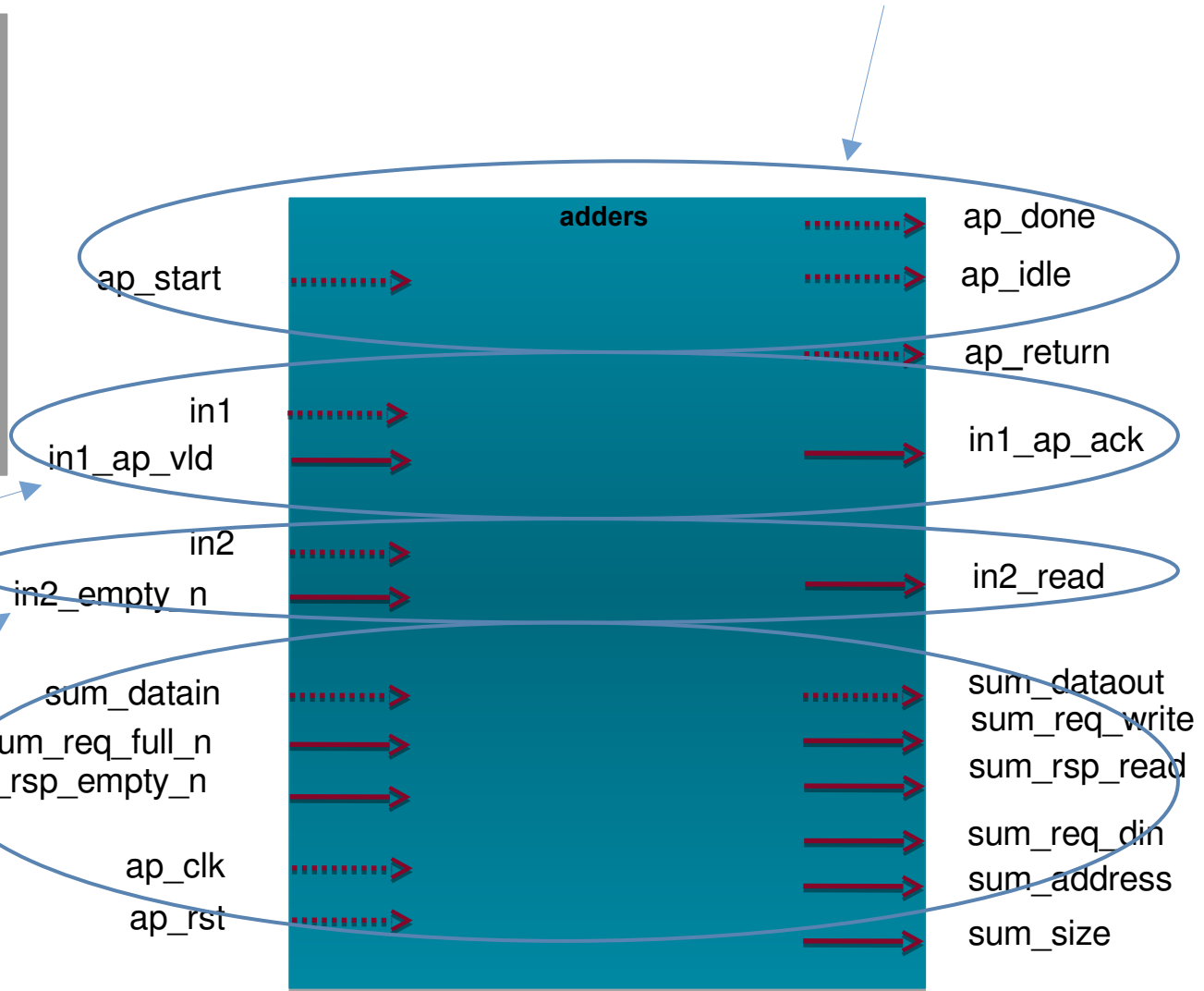
```

#include "adders.h"
int adders(int in1, int in2,
           int *sum) {

    int temp;
    *sum = in1 + in2 + *sum;
    temp = in1 + in2;

    return temp;
}
    
```

Function activation



Input data can include strobcs, acks, ...

Or be modelled as fifo channels

Data passed by reference is modelled as R/W  
(not necessarily as memory  
As in this case)

# An example: Matrix Multiplication

## Solution 1: naive implementation (no optimization)

```
typedef int mat_a_t;
typedef int mat_b_t;
typedef int result_t;

void matrixmul(
    mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
    mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
    result_t res[MAT_A_ROWS][MAT_B_COLS])
{
    // Iterate over the rows of the A matrix
    Row: for(int i = 0; i < MAT_A_ROWS; i++) {

        // Iterate over the columns of the B matrix
        Col: for(int j = 0; j < MAT_B_COLS; j++) {

            // Inner product of a row of A and col of B
            res[i][j] = 0;

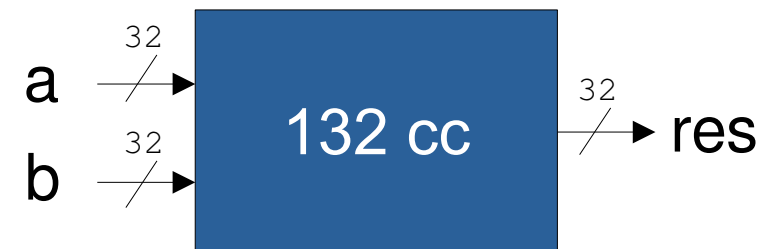
            Product: for(int k = 0; k < MAT_B_ROWS; k++) {
                res[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

3x3 square matrixes

Clock cycle: 8.50 ns

Loop	Latency	Iteration latency	Trip count	Initiation interval
Row	132	44	3	0
Col	42	14	3	0
Product	12	4	3	0

Resources	BRAM	DSP	FF	LUT
Total	0	3	158	271



# Schedule Viewer

- Perspective for design analysis

The screenshot shows the Vitis HLS 2021.2 interface for a project named 'matmult'. The 'Module Hierarchy' pane on the left shows a tree structure with 'matrixmul' expanded to show 'Row' and 'Col' sub-modules, with 'Product' selected. The 'Schedule Viewer' pane on the right displays a Gantt chart for the 'Product' module, showing operations like 'mul\_ln83(\*)' and 'add\_ln83(+)' mapped across clock cycles 0 to 5. A blue callout box labeled 'Scheduling' with the text 'Mapping of operations to cycles' points to the Gantt chart. The 'Flow Navigator' pane at the bottom left shows a list of simulation and synthesis steps, with 'Schedule Viewer' selected. A blue callout box labeled 'Performance Profile' with the text 'Hw resources Latencies + Intervals' points to the 'Performance Profile' section in the Flow Navigator. The 'Performance Profile' table at the bottom right shows the following data:

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iterati
matrixmul				-	160	1.600E3	
Row				-	159	1.590E3	
Col				-	51	510.000	
Product				-	15	150.000	

# Schedule Viewer

The screenshot displays the Vitis HLS Schedule Viewer interface. The main window shows a Gantt chart with a time axis from 0 to 7. The chart is divided into three main sections: Row (top), Col (middle), and Product (bottom). The Row section contains operations like `i_1(read)`, `zext_ln83(zext)`, `tmp(bitconcatenate)`, `sub_ln83(-)`, `icmp_ln76(icmp)`, `add_ln76(+)`, `br_ln76(br)`, and `br_ln78(br)`. The Col section contains `j(phi_mux)`, `zext_ln81(zext)`, `add_ln81(+)`, `zext_ln81_1(zext)`, and `res_addr(getelemer)`. The Product section contains `res_addr(getelemer)`. A blue box labeled "Scheduled states" points to the Gantt chart. A blue box labeled "Module hierarchy" points to the left sidebar, which lists the operations and loops. A blue box labeled "Cross references" points to the C++ source code in the right window, which shows the implementation of the matrix multiplication. The source code includes comments and a loop structure: `for(int i = 0; i < MAT_A_ROWS; i++)` for the Row, `for(int j = 0; j < MAT_B_COLS; j++)` for the Col, and `for(int k = 0; k < MAT_B_ROWS; k++)` for the Product. The code also includes a `#pragma HLS PIPELINE off` directive.

**Module hierarchy**  
Operations, loops  
And functions

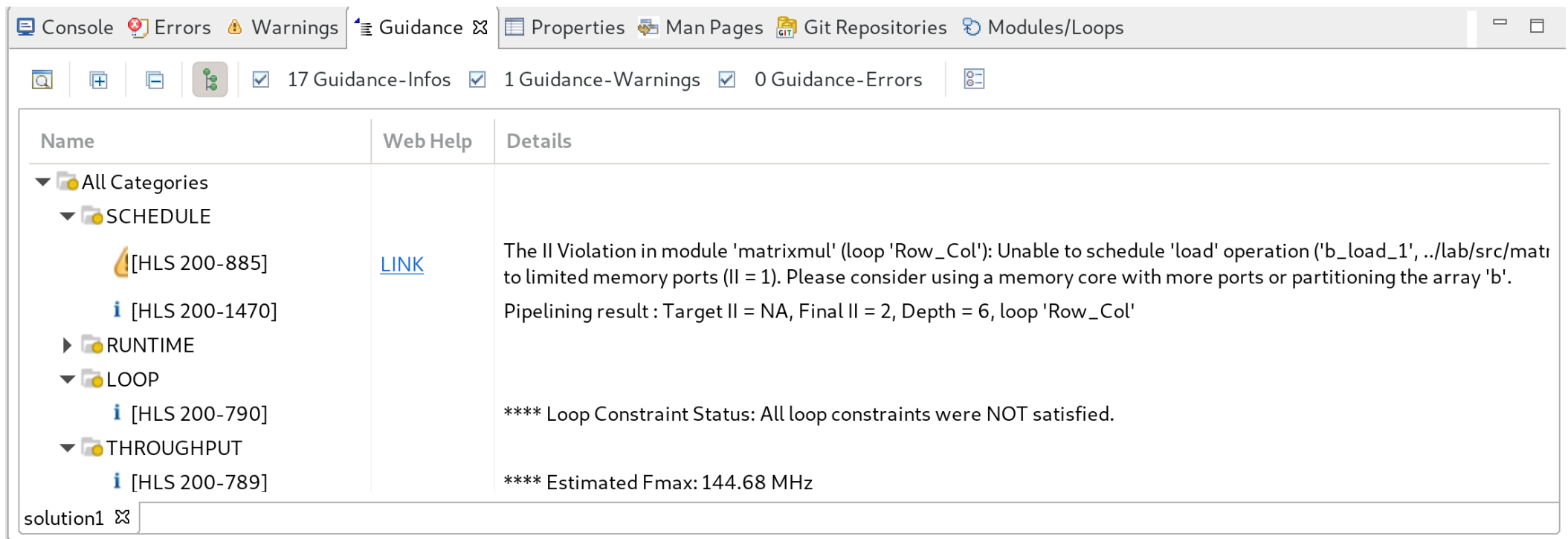
**Scheduled states**

**Cross references**

```
61 // Design name: matrixmul
62 // Purpose:
63 // This is a C++ version of a matrix m
64 // Reference:
65 // Revision History:
66 //*****
75 // Iterate over the rows of the A matrix
76 Row: for(int i = 0; i < MAT_A_ROWS; i++)
77 #pragma HLS PIPELINE off
78   Col: for(int j = 0; j < MAT_B_COLS; j++)
79     // Iterate over the columns of the B matrix
80     // Do the inner product of a row of
81     res[i][j] = 0;
82     Product: for(int k = 0; k < MAT_B_ROWS; k++)
83       res[i][j] += a[i][k] * b[k][j];
84   }
85 }
86 }
87 }
```

# Guidance

- Outlines the main problems and proposes solutions



The screenshot shows the Guidance window in the Vivado IDE. The window title bar includes tabs for Console, Errors, Warnings, Guidance, Properties, Man Pages, Git Repositories, and Modules/Loops. Below the title bar, there are icons for search, home, and refresh, along with status indicators: 17 Guidance-Infos, 1 Guidance-Warnings, and 0 Guidance-Errors.

Name	Web Help	Details
▼ All Categories		
▼ SCHEDULE		
🔥 [HLS 200-885]	<a href="#">LINK</a>	The II Violation in module 'matrixmul' (loop 'Row_Col'): Unable to schedule 'load' operation ('b_load_1', ../lab/src/matr to limited memory ports (II = 1). Please consider using a memory core with more ports or partitioning the array 'b'. Pipelining result : Target II = NA, Final II = 2, Depth = 6, loop 'Row_Col'
📘 [HLS 200-1470]		
▶ RUNTIME		
▼ LOOP		
📘 [HLS 200-790]		**** Loop Constraint Status: All loop constraints were NOT satisfied.
▼ THROUGHPUT		
📘 [HLS 200-789]		**** Estimated Fmax: 144.68 MHz

At the bottom left of the window, the text "solution1" is visible next to a refresh icon.

# MM Pipelined version

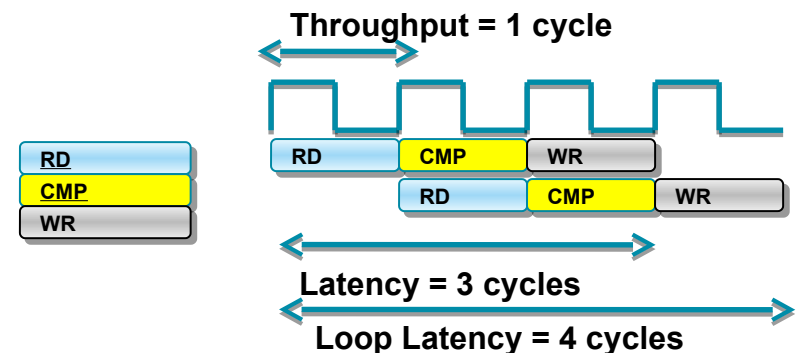
## Solution 2: pipelining

```
void matrixmul(  
    mat_a_t a[MAT_A_ROWS][MAT_A_COLS],  
    mat_b_t b[MAT_B_ROWS][MAT_B_COLS],  
    result_t res[MAT_A_ROWS][MAT_B_COLS])  
{  
    // Iterate over the rows of the A matrix  
    Row: for(int i = 0; i < MAT_A_ROWS; i++) {  
  
        // Iterate over the columns of the B matrix  
        Col: for(int j = 0; j < MAT_B_COLS; j++) {  
  
            // Inner product of a row of A and col of B  
            res[i][j] = 0;  
  
            Product: for(int k = 0; k < MAT_B_ROWS; k++) {  
                #pragma HLS PIPELINE  
                res[i][j] += a[i][k] * b[k][j];  
            }  
        }  
    }  
}
```

Clock cycle: 8.50 ns

Loop	Latency	Iteration latency	Trip count	Initiation interval
Row_col	99	11	9	1
Product	7	4	3	2

Resources	BRAM	DSP	FF	LUT
Total	0	3	137	322



# MM Custom bit size

## Solution 3: 10 bit inputs

```
typedef ap_int<18> mat_a_t;
typedef ap_int<18> mat_b_t;
typedef ap_int<18> result_t;

void matrixmul(
    mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
    mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
    result_t res[MAT_A_ROWS][MAT_B_COLS])
{
    // Iterate over the rows of the A matrix
    Row: for(int i = 0; i < MAT_A_ROWS; i++) {

        // Iterate over the columns of the B matrix
        Col: for(int j = 0; j < MAT_B_COLS; j++) {

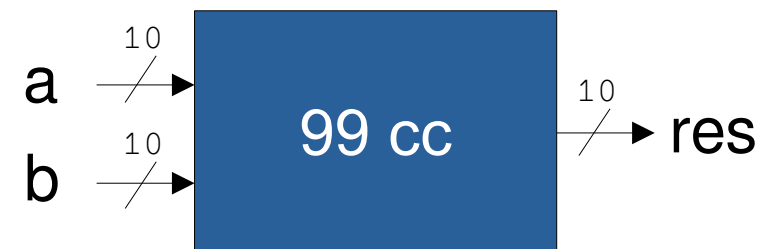
            // Inner product of a row of A and col of B
            res[i][j] = 0;

            Product: for(int k = 0; k < MAT_B_ROWS; k++) {
                #pragma HLS PIPELINE II=2
                res[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

Clock cycle: 8.50 ns

Loop	Latency	Iteration latency	Trip count	Initiation interval
Row_col	99	11	9	1
Product	7	4	3	2

Resources	BRAM	DSP	FF	LUT
Total	0	3	137	322





# MM Array Partition

## Solution 4: fully partition a & b

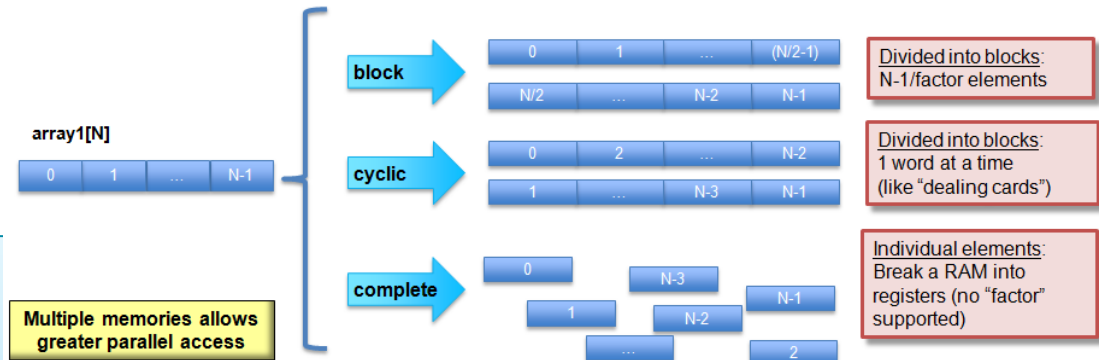
```

void matrixmul(
    mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
    mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
    result_t res[MAT_A_ROWS][MAT_B_COLS])
{
    #pragma HLS ARRAY_PARTITION variable=b complete dim=1
    #pragma HLS ARRAY_PARTITION variable=a complete dim=2
    // Iterate over the rows of the A matrix
    Row: for(int i = 0; i < MAT_A_ROWS; i++) {

        // Iterate over the columns of the B matrix
        Col: for(int j = 0; j < MAT_B_COLS; j++) {

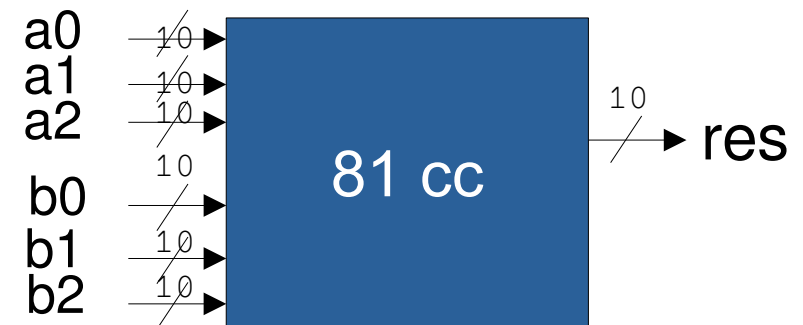
            // Inner product of a row of A and col of B
            res[i][j] = 0;

            Product: for(int k = 0; k < MAT_B_ROWS; k++) {
                #pragma HLS PIPELINE II=2
                res[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
    
```



Loop	Latency	Iteration latency	Trip count	Initiation interval
Row_col	81	9	9	1
Product	6	3	3	2

Resources	BRAM	DSP	FF	LUT
Total	0	1	64	243



# MM Floating-Point

## Solution 5: floating point

```
typedef float mat_a_t;
typedef float mat_b_t;
typedef float result_t;

void matrixmul(
    mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
    mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
    result_t res[MAT_A_ROWS][MAT_B_COLS])
{
    // Iterate over the rows of the A matrix
    Row: for(int i = 0; i < MAT_A_ROWS; i++) {

        // Iterate over the columns of the B matrix
        Col: for(int j = 0; j < MAT_B_COLS; j++) {

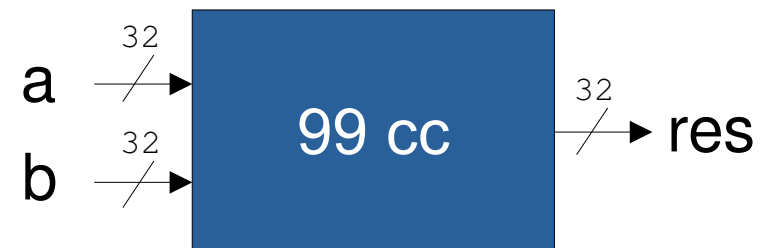
            // Inner product of a row of A and col of B
            res[i][j] = 0;

            Product: for(int k = 0; k < MAT_B_ROWS; k++) {
                #pragma HLS PIPELINE II=2
                res[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

Clock cycle: 7.96 ns

Loop	Latency	Iteration latency	Trip count	Initiation interval
Row_col	216	24	9	0
Product	20	11	3	5

Resources	BRAM	DSP	FF	LUT
Total	0	5	489	1002



# MM Interface Synthesis

## Function activation interface

Can be disabled  
`ap_control_none`

RTL ports	dir	bits	Protocol	C Type
<code>ap_clk</code>	in	1	<code>ap_ctrl_hs</code>	return value
<code>ap_rst</code>	in	1	<code>ap_ctrl_hs</code>	return value
<code>ap_start</code>	in	1	<code>ap_ctrl_hs</code>	return value
<code>ap_done</code>	out	1	<code>ap_ctrl_hs</code>	return value
<code>ap_idle</code>	out	1	<code>ap_ctrl_hs</code>	return value
<code>ap_ready</code>	out	1	<code>ap_ctrl_hs</code>	return value
<code>in_a_address0</code>	out	8	<code>ap_memory</code>	array
<code>in_a_ce0</code>	out	1	<code>ap_memory</code>	array
<code>in_a_q0</code>	in	32	<code>ap_memory</code>	array
<code>in_b_address0</code>	out	8	<code>ap_memory</code>	array
<code>in_b_ce0</code>	out	1	<code>ap_memory</code>	array
<code>in_b_q0</code>	in	32	<code>ap_memory</code>	array
<code>in_c_address0</code>	out	8	<code>ap_memory</code>	array
<code>in_c_ce0</code>	out	1	<code>ap_memory</code>	array
<code>in_c_we0</code>	out	1	<code>ap_memory</code>	array
<code>in_c_d0</code>	out	32	<code>ap_memory</code>	array

## Synthesized memory ports

Also dual-ported

In the array partitioned  
Version, 3 mem ports.  
One per partial product

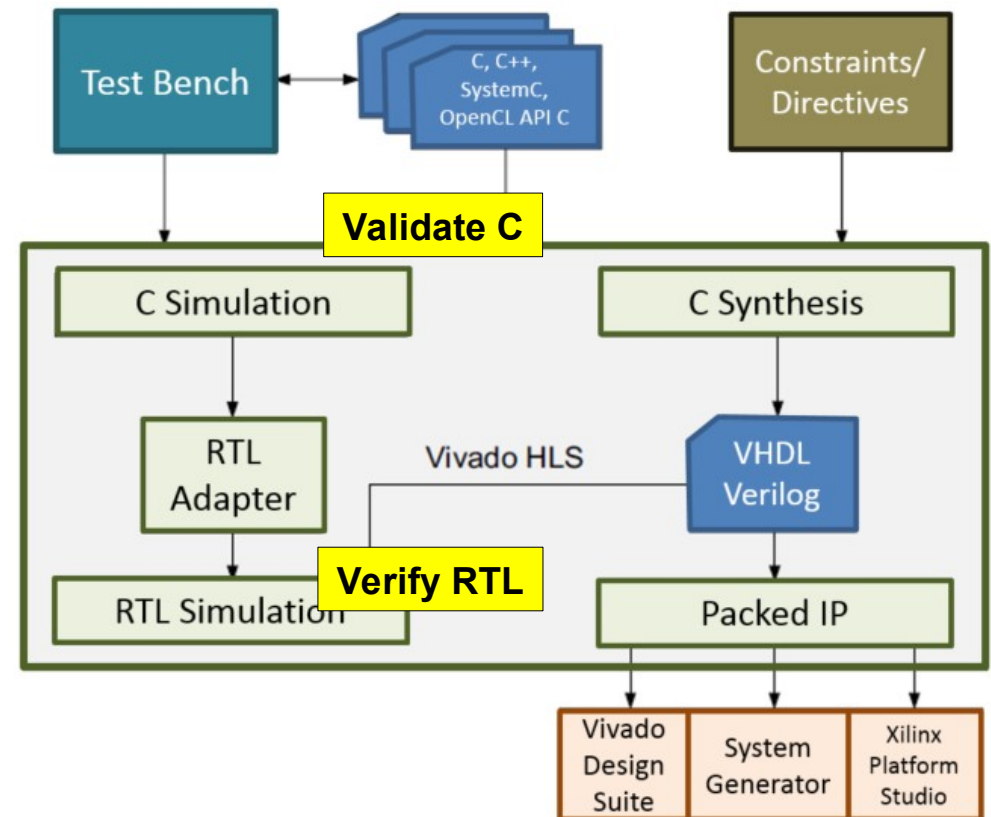
# Interface synthesis

- I/O ports can be mapped to different bus interfaces
- Let's map the MM to an AXI Lite bus
  - `#pragma HSL INTERFACE s_axilite port=a bundle=myBus`
  - The bundle is used to group more than one port into the same bus

RTL ports	dir	bits	Protocol	RTL ports	dir	bits	Protocol
ap_clk	in	1	ap_ctrl_hs	s_axi_myBus_WSTRB	in	4	s_axi
ap_rst_n	in	1	ap_ctrl_hs	s_axi_myBus_ARVALID	in	1	s_axi
ap_start	in	1	ap_ctrl_hs	s_axi_myBus_ARREADY	out	1	s_axi
ap_done	out	1	ap_ctrl_hs	s_axi_myBus_ARADDR	in	8	s_axi
ap_idle	out	1	ap_ctrl_hs	s_axi_myBus_RVALID	out	1	s_axi
ap_ready	out	1	ap_ctrl_hs	s_axi_myBus_RREADY	in	1	s_axi
s_axi_myBus_AWVALID	in	1	s_axi	s_axi_myBus_RDATA	out	32	s_axi
s_axi_myBus_AWREADY	out	1	s_axi	s_axi_myBus_RRESP	out	2	s_axi
s_axi_myBus_AWADDR	in	1	s_axi	s_axi_myBus_BVALID	out	1	s_axi
s_axi_myBus_WVALID	in	1	s_axi	s_axi_myBus_BREADY	in	1	s_axi
s_axi_myBus_WREADY	out	1	s_axi	s_axi_myBus_BRESP	out	2	s_axi
s_axi_myBus_WDATA	in	32	s_axi				

# Validation Flow

- Two steps for design verification
  - Before synthesis
  - After synthesis
- Pre-synthesis: C Validation
  - Validate the algorithm is correct
- Post-synthesis: RTL Verification
  - Verify the RTL is correct
- C validation
  - A **HUGE** reason to use HLS
    - Fast, free verification
  - Validate the algorithm is correct before synthesis
    - Follow the test bench tips given over
- RTL Verification
  - Vivado HLS can co-simulate the RTL with the original test bench



# Test benches

- The test bench should be in a separate file
- Or excluded from synthesis
  - The Macro `__SYNTHESIS__` can be used to isolate code which will not be synthesized

Design to be synthesized

## Test Bench

Nothing in this `ifndef` will be read by Vivado HLS

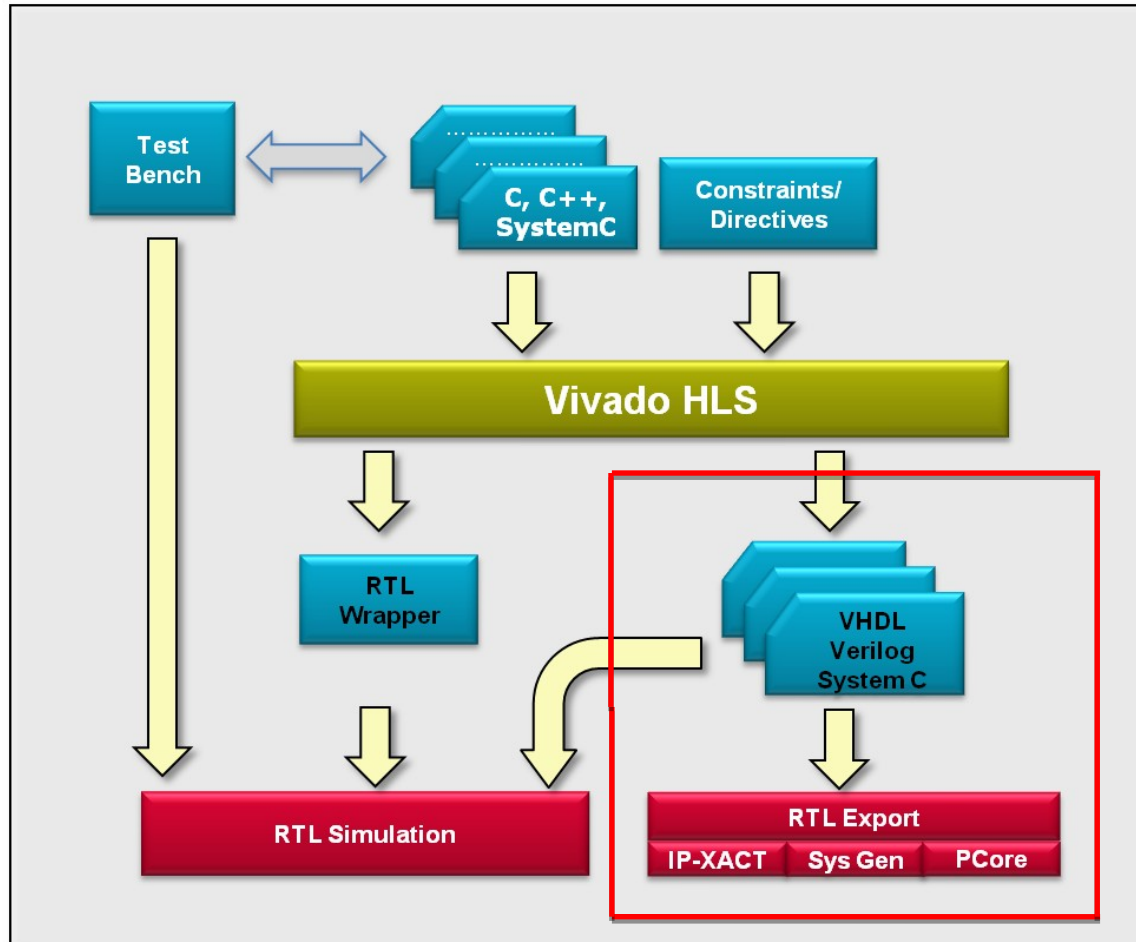
```
// test.c
#include <stdio.h>
void test (int d[10]) {
    int acc = 0;
    int i;
    for (i=0;i<10;i++) {
        acc += d[i];
        d[i] = acc;
    }
}
#ifndef __SYNTHESIS__
int main () {
    int d[10], i;
    for (i=0;i<10;i++) {
        d[i] = i;
    }
    test(d);
    for (i=0;i<10;i++) {
        printf("%d %d\n", i, d[i]);
    }
    return 0;
}
#endif
```

# Test benches: ideal test bench

- Self checking
  - RTL verification will re-use the C test bench
  - If the test bench is self-checking
    - Allows RTL Verification to be run without a requirement to check the results again
- RTL verification “passes” if the test bench return value is 0 (zero)

```
int main () {  
  
    // Compare results  
    int ret = system("diff --brief -w output.dat output.golden.dat");  
    if (ret != 0) {  
        printf("Test failed !!!\n", ret); return 1;  
    } else {  
        printf("Test passed !\n", ret); return 0;  
    }  
}
```

# RTL Export



RTL output in Verilog, VHDL and SystemC

Scripts created for RTL synthesis tools

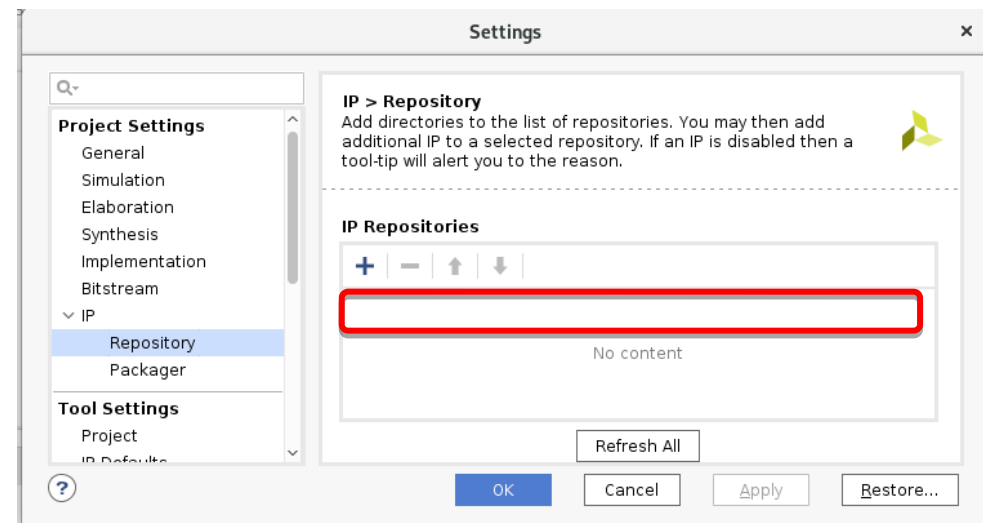
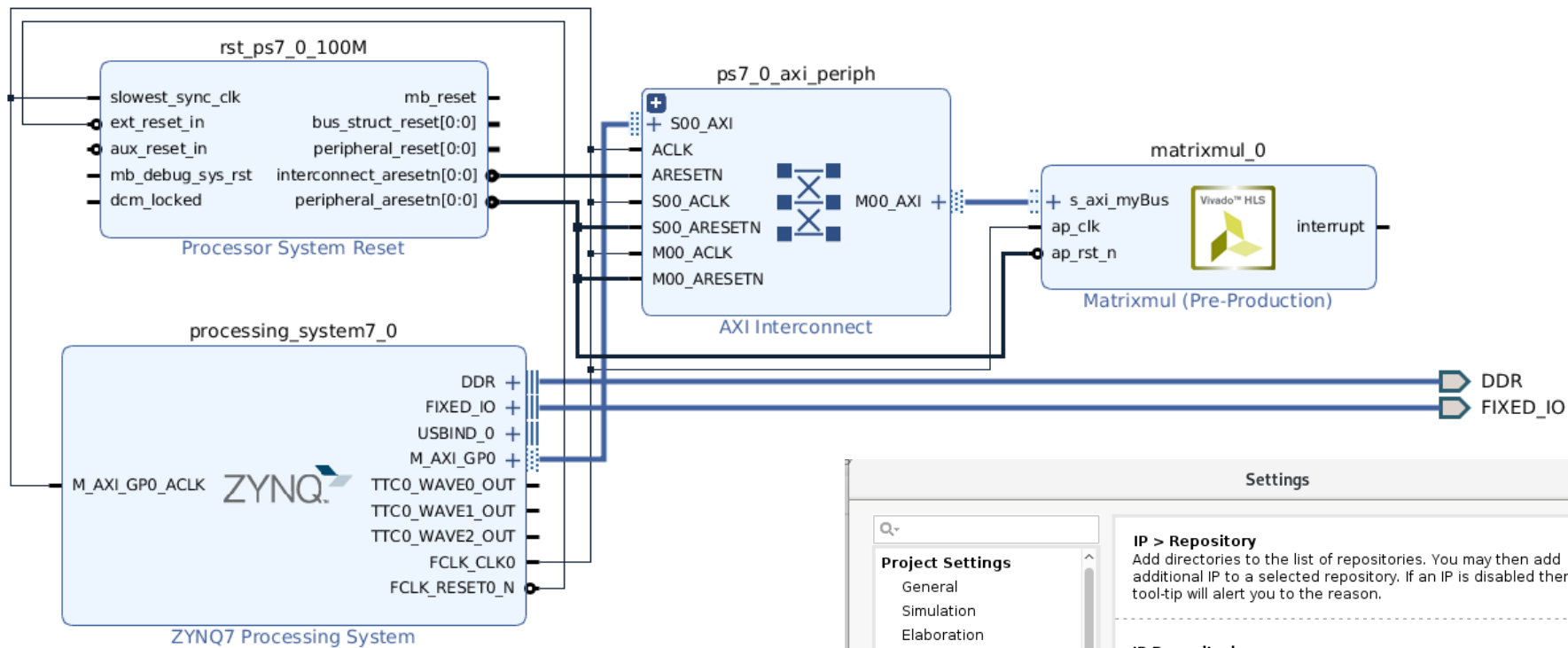
RTL Export to IP-XACT, SysGen, and Pcore formats

IP-XACT and SysGen => Vivado HLS for 7 Series and Zynq families  
PCore => Only Vivado HLS Standalone for all families



# IP integration

- Exported cores can be directly integrated in Vivado



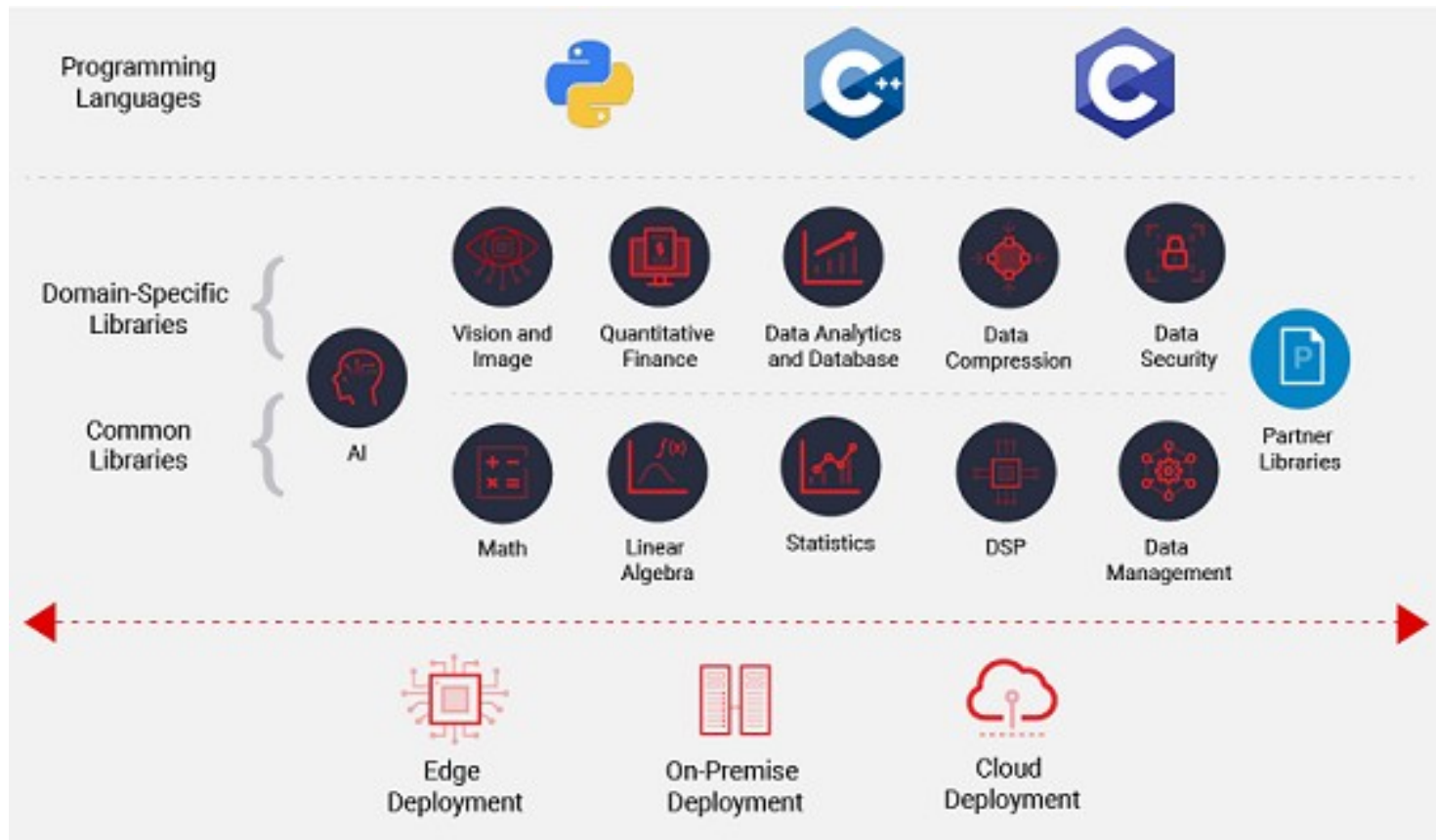
# Software Drivers

- And both drivers for baremetal and User Space linux are generated

```
70 #define XIL_COMPONENT_IS_READY 1
71 #endif
72
73 /***** Function Prototypes *****/
74 #ifndef __linux__
75 int XMatrixmul_Initialize(XMatrixmul *InstancePtr, u16 DeviceId);
76 XMatrixmul_Config* XMatrixmul_LookupConfig(u16 DeviceId);
77 int XMatrixmul_CfgInitialize(XMatrixmul *InstancePtr, XMatrixmul_Config *ConfigPtr);
78 #else
79 int XMatrixmul_Initialize(XMatrixmul *InstancePtr, const char* InstanceName);
80 int XMatrixmul_Release(XMatrixmul *InstancePtr);
81 #endif
82
83
84 u32 XMatrixmul_Get_a_BaseAddress(XMatrixmul *InstancePtr);
85 u32 XMatrixmul_Get_a_HighAddress(XMatrixmul *InstancePtr);
86 u32 XMatrixmul_Get_a_TotalBytes(XMatrixmul *InstancePtr);
87 u32 XMatrixmul_Get_a_BitWidth(XMatrixmul *InstancePtr);
88 u32 XMatrixmul_Get_a_Depth(XMatrixmul *InstancePtr);
89 u32 XMatrixmul_Write_a_Words(XMatrixmul *InstancePtr, int offset, word_type *data, int length);
90 u32 XMatrixmul_Read_a_Words(XMatrixmul *InstancePtr, int offset, word_type *data, int length);
91 u32 XMatrixmul_Write_a_Bytes(XMatrixmul *InstancePtr, int offset, char *data, int length);
92 u32 XMatrixmul_Read_a_Bytes(XMatrixmul *InstancePtr, int offset, char *data, int length);
93 u32 XMatrixmul_Get_b_BaseAddress(XMatrixmul *InstancePtr);
94 u32 XMatrixmul_Get_b_HighAddress(XMatrixmul *InstancePtr);
95 u32 XMatrixmul_Get_b_TotalBytes(XMatrixmul *InstancePtr);
96 u32 XMatrixmul_Get_b_BitWidth(XMatrixmul *InstancePtr);
97 u32 XMatrixmul_Get_b_Depth(XMatrixmul *InstancePtr);
98 u32 XMatrixmul_Write_b_Words(XMatrixmul *InstancePtr, int offset, word_type *data, int length);
99 u32 XMatrixmul_Read_b_Words(XMatrixmul *InstancePtr, int offset, word_type *data, int length);
100 u32 XMatrixmul_Write_b_Bytes(XMatrixmul *InstancePtr, int offset, char *data, int length);
```

# HLS Libraries

- Vitis accelerated libraries
  - Valid for classic Vivado flow
  - Compatible with the new OpenCL-based flow



# An example: Vision libraries

- Based on the OpenCV standard
- Big number of OpenCV operations available for synthesis
- Full OpenCV for test
- Interface synthesis for common Xilinx bus interfaces

XILINX Applications Products Developers Support

🏠 Vitis Vision Library  
2020.2

Search docs

**Vitis Vision Library User Guide**

- Overview
- Getting Started with Vitis Vision
- Getting Started with HLS
- Design Examples Using Vitis Vision Library

**Vitis Vision Library API Reference**

- Overview
- xf::cv::Mat Image Container Class
- Vitis Vision Library Functions

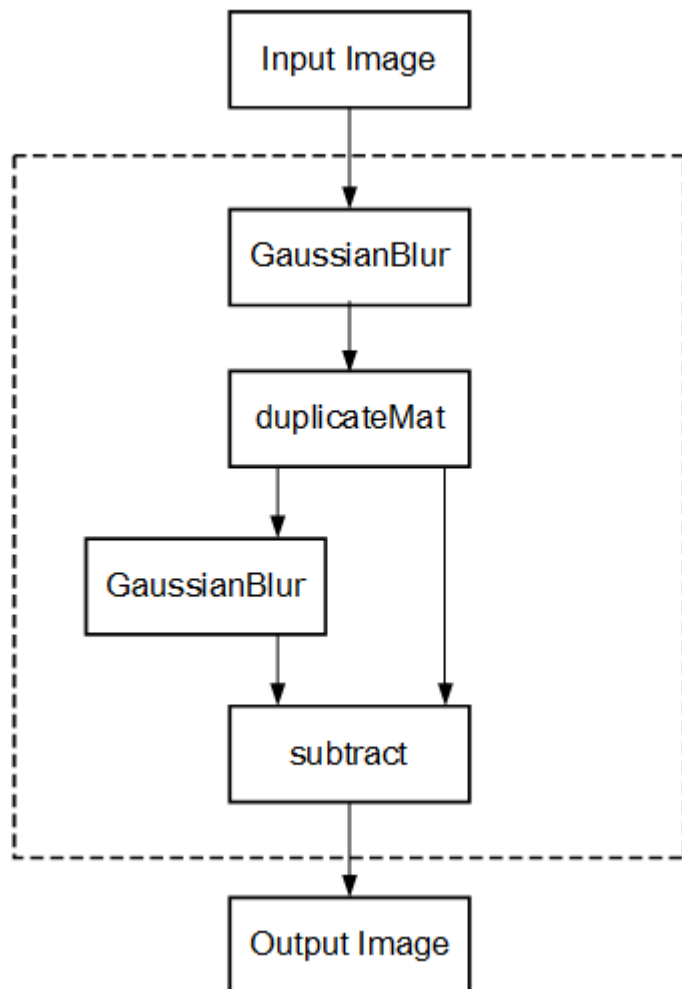
**This Page**

- Show Source

- xf::cv::absDiff
- xf::cv::convertTo
- Vitis Vision Library Functions
  - Absolute Difference
  - Accumulate
  - Accumulate Squared
  - Accumulate Weighted
  - AddS
  - Addweighted
  - Autoexposurecorrection
  - Autowhitebalance
  - Badpixelcorrection
  - Brute-force (Bf) Feature Matcher
  - Bilateral Filter
  - Bit Depth Conversion
  - Bitwise AND
  - Bitwise NOT
  - Bitwise OR
  - Bitwise XOR
  - Blacklevelcorrection
  - Box Filter
  - BoundingBox
  - Canny Edge Detection
  - Channel Combine
  - Channel Extract
  - Color Conversion
  - Color correction matrix

# An example: Vision libraries

- Difference of Gaussian Filter



```
void gaussiandiference(ap_uint<PTR_WIDTH>* img_in, float sigma, ap_uint<PTR_WIDTH>* img_out, int rows, int cols) {  
  
#pragma HLS INTERFACE m_axi      port=img_in      offset=slave  bundle=gmem0  
#pragma HLS INTERFACE m_axi      port=img_out      offset=slave  bundle=gmem1  
#pragma HLS INTERFACE s_axilite  port=sigma  
#pragma HLS INTERFACE s_axilite  port=rows  
#pragma HLS INTERFACE s_axilite  port=cols  
#pragma HLS INTERFACE s_axilite  port=return  
  
xf::cv::Mat<TYPE, HEIGHT, WIDTH, NPC1> imgInput(rows, cols);  
xf::cv::Mat<TYPE, HEIGHT, WIDTH, NPC1> imgin1(rows, cols);  
xf::cv::Mat<TYPE, HEIGHT, WIDTH, NPC1> imgin2(rows, cols);  
xf::cv::Mat<TYPE, HEIGHT, WIDTH, NPC1, 15360> imgin3(rows, cols);  
xf::cv::Mat<TYPE, HEIGHT, WIDTH, NPC1> imgin4(rows, cols);  
xf::cv::Mat<TYPE, HEIGHT, WIDTH, NPC1> imgOutput(rows, cols);  
  
#pragma HLS DATAFLOW  
  
// Retrieve xf::cv::Mat objects from img_in data:  
xf::cv::Array2xfMat<PTR_WIDTH, TYPE, HEIGHT, WIDTH, NPC1>(img_in, imgInput);  
  
// Run xfOpenCV kernel:  
xf::cv::GaussianBlur<FILTER_WIDTH, XF_BORDER_CONSTANT, TYPE, HEIGHT, WIDTH, NPC1>(imgInput, imgin1, sigma);  
xf::cv::duplicateMat<TYPE, HEIGHT, WIDTH, NPC1, 15360>(imgin1, imgin2, imgin3);  
xf::cv::GaussianBlur<FILTER_WIDTH, XF_BORDER_CONSTANT, TYPE, HEIGHT, WIDTH, NPC1>(imgin2, imgin4, sigma);  
xf::cv::subtract<XF_CONVERT_POLICY_SATURATE, TYPE, HEIGHT, WIDTH, NPC1, 15360>(imgin3, imgin4, imgOutput);  
  
// Convert output xf::cv::Mat object to output array:  
xf::cv::xfMat2Array<PTR_WIDTH, TYPE, HEIGHT, WIDTH, NPC1>(imgOutput, img_out);  
  
return;  
} // End of kernel
```

# References

- M. Fingeroff, “High-Level Synthesis Blue Book”, X libris Corporation, 2010
- P. Coussy, A. Morawiec, “High-Level Synthesis: from Algorithm to Digital Circuit”, Springer, 2008
- “High-Level Synthesis Flow on Zynq” Course materials from the Xilinx University Program, 2016