Heterogeneous FPGA acceleration using Xilinx Vitis Development Environment

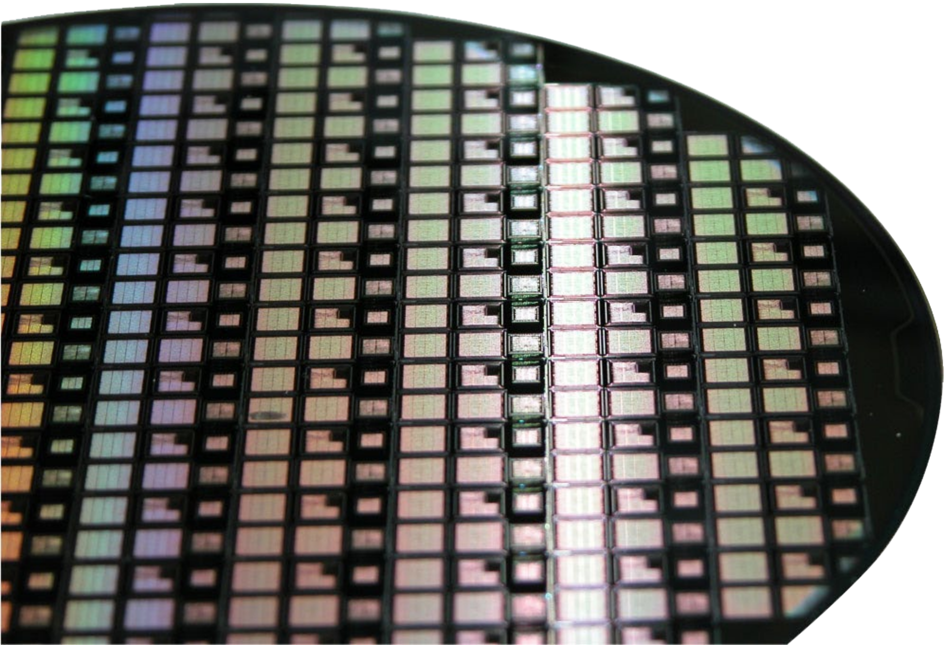
Energy Efficient Parallel Computing Systems for Data Analytics

21/05/2024

Arpan Prasad

Federico Villani

Integrated Systems Laboratory

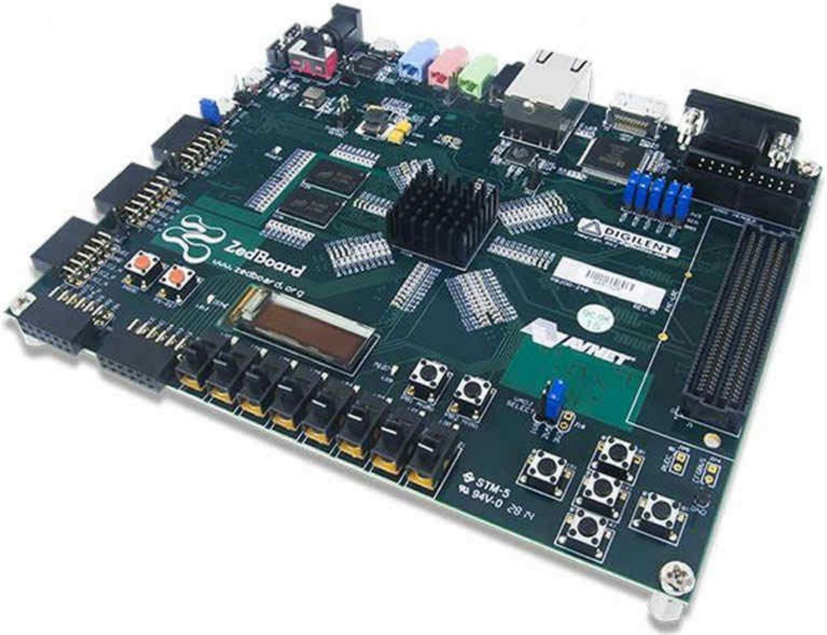


Goals

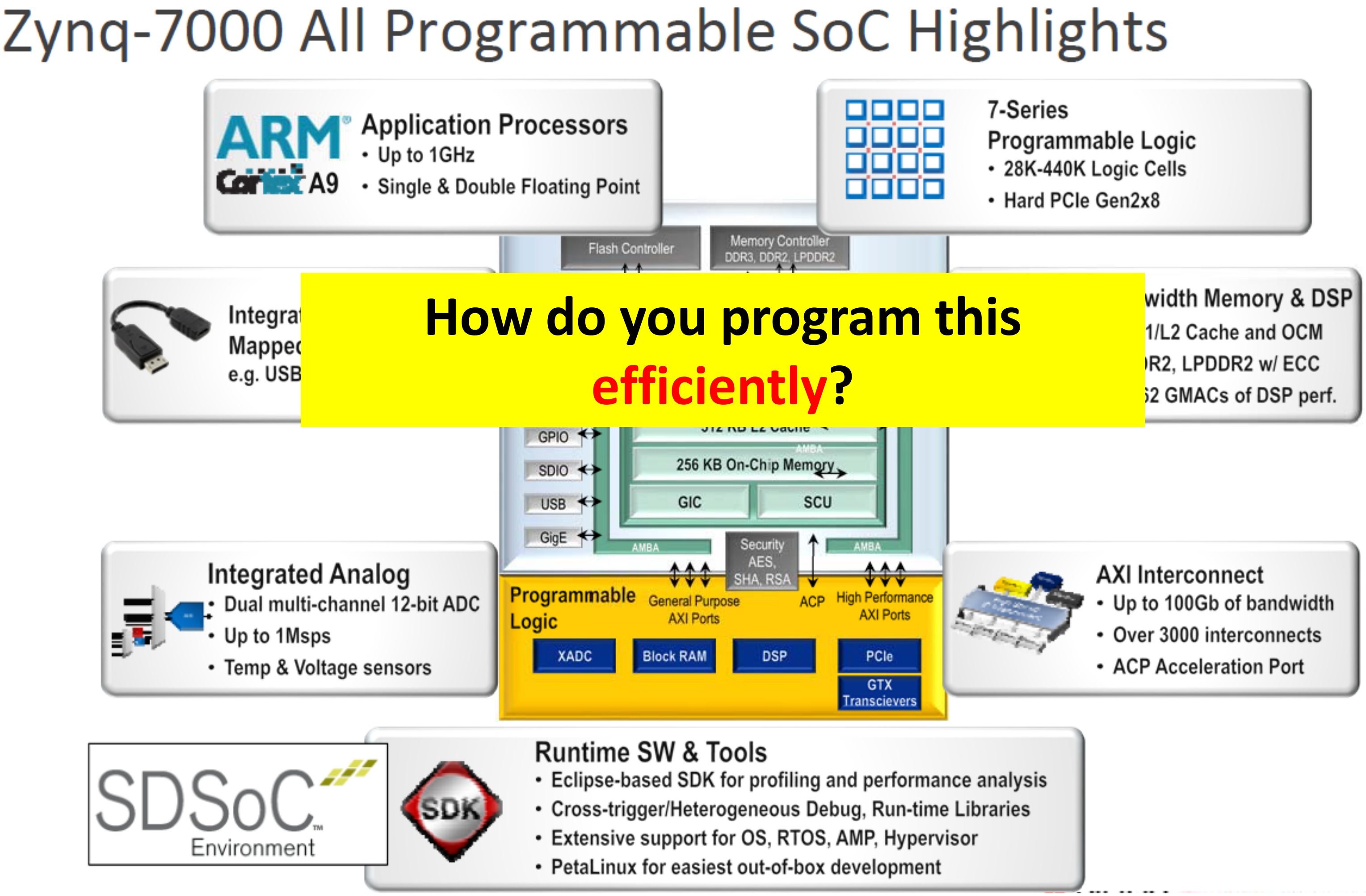
- *Learn how to accelerate an application on a SoC with programmable logic* (Xilinx Zynq)
- *Use an advanced design flow to do so* (Xilinx Vitis and Vitis HLS)
- *Use High-Level-Synthesis to design the HW accelerator*

Zynq-7000 board

- *Software-Defined System on Chip (SDSoC)*
- *ARM Cores + FPGA Fabric for custom hardware mapping*



Vitis Development Environment



Zynq-7000 All Programmable SoC Highlights

Application Processors
- Up to 1GHz
- Single & Double Floating Point

7-Series
Programmable Logic
- 28K-440K Logic Cells
- Hard PCIe Gen2x8

Integrat
Mappe
e.g. USB

width Memory & DSP
1/L2 Cache and OCM
R2, LPDDR2 w/ ECC
2 GMACs of DSP perf.

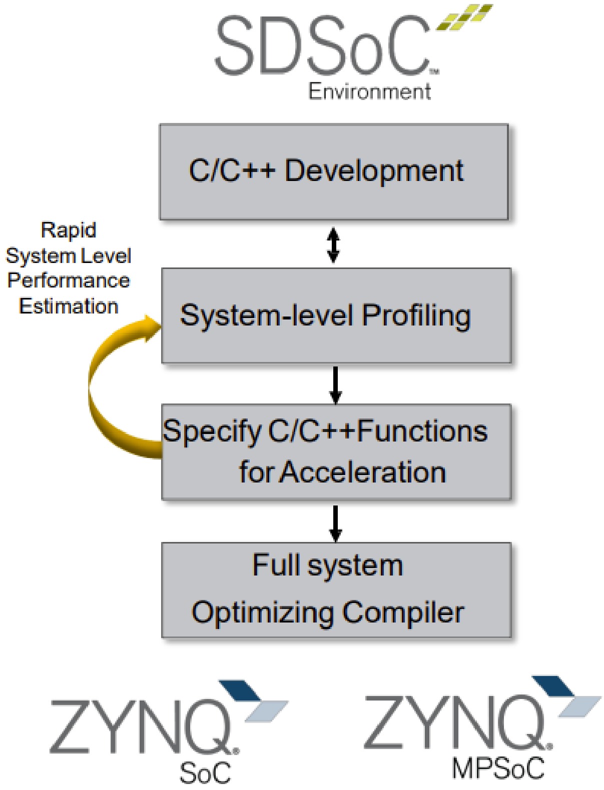
Integrated Analog
- Dual multi-channel 12-bit ADC
- Up to 1Msps
- Temp & Voltage sensors

AXI Interconnect
- Up to 100Gb of bandwidth
- Over 3000 interconnects
- ACP Acceleration Port

Runtime SW & Tools
- Eclipse-based SDK for profiling and performance analysis
- Cross-trigger/Heterogeneous Debug, Run-time Libraries
- Extensive support for OS, RTOS, AMP, Hypervisor
- PetaLinux for easiest out-of-box development

How do you program this efficiently?

SDx vs Vitis development environment



SDSoC Environment

C/C++ Development

System-level Profiling

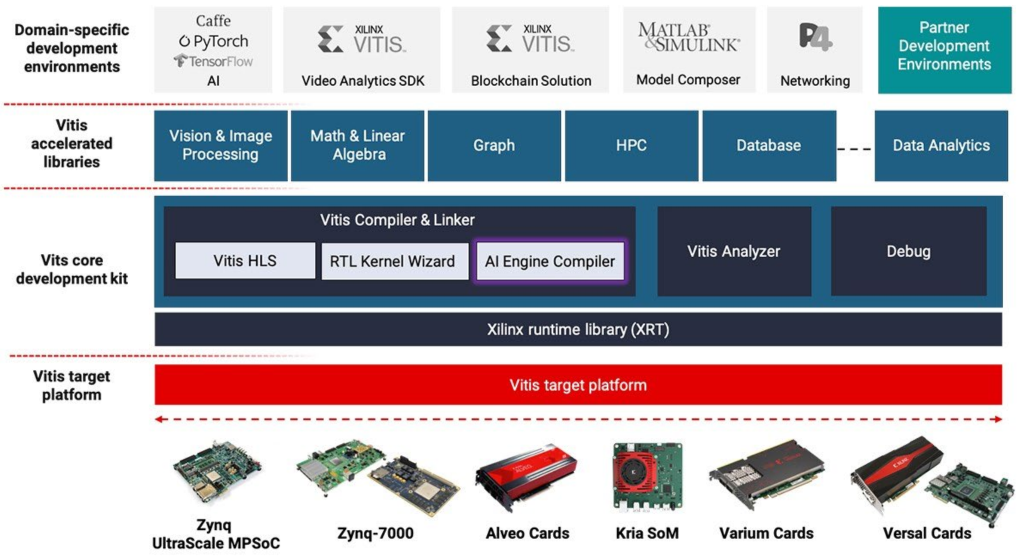
Specify C/C++Functions for Acceleration

Full system Optimizing Compiler

Rapid System Level Performance Estimation

ZYNQ SoC

ZYNQ MPSoC



Domain-specific development environments: Caffe, PyTorch, TensorFlow — AI | XILINX VITIS — Video Analytics SDK | XILINX VITIS — Blockchain Solution | MATLAB & SIMULINK — Model Composer | P4 — Networking | Partner Development Environments

Vitis accelerated libraries: Vision & Image Processing | Math & Linear Algebra | Graph | HPC | Database | Data Analytics

Vits core development kit: Vitis Compiler & Linker (Vitis HLS, RTL Kernel Wizard, AI Engine Compiler) | Vitis Analyzer | Debug

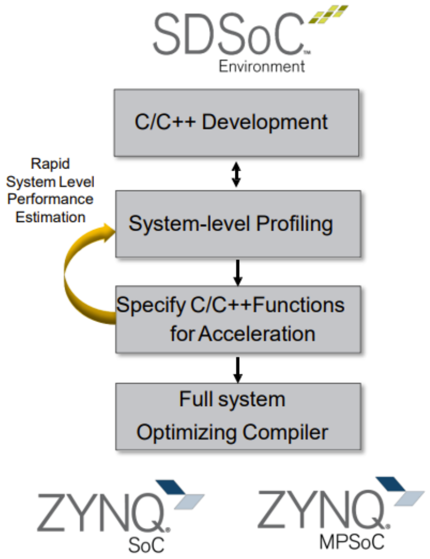
Xilinx runtime library (XRT)

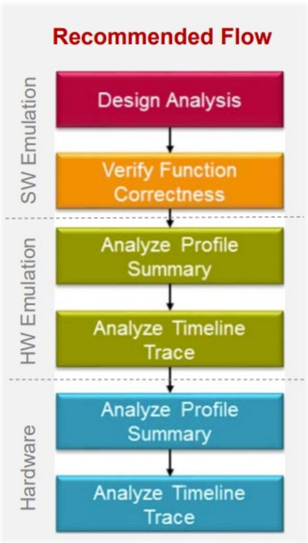
Vitis target platform: Vitis target platform

Zynq UltraScale MPSoC | Zynq-7000 | Alveo Cards | Kria SoM | Varium Cards | Versal Cards

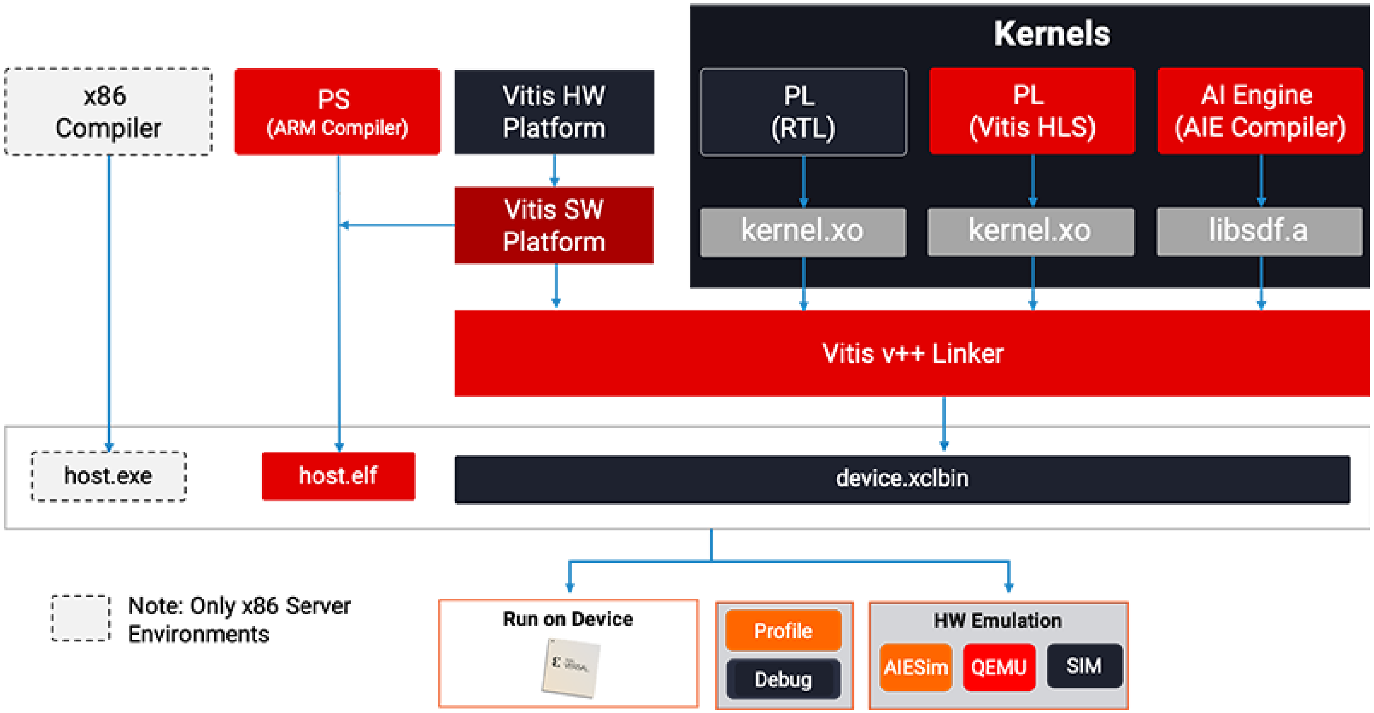
SDx vs Vitis- What changes

- 3 SDX tools -> integrated into Vitis and Vitis_hls
- More oriented to data centres
- Different development flow, tightly integrated with Linux
- In today's exercise => Bare metal development


SDSoC Environment
C/C++ Development
Rapid System Level Performance Estimation
System-level Profiling
Specify C/C++Functions for Acceleration
Full system Optimizing Compiler
ZYNQ SoC
ZYNQ MPSoC



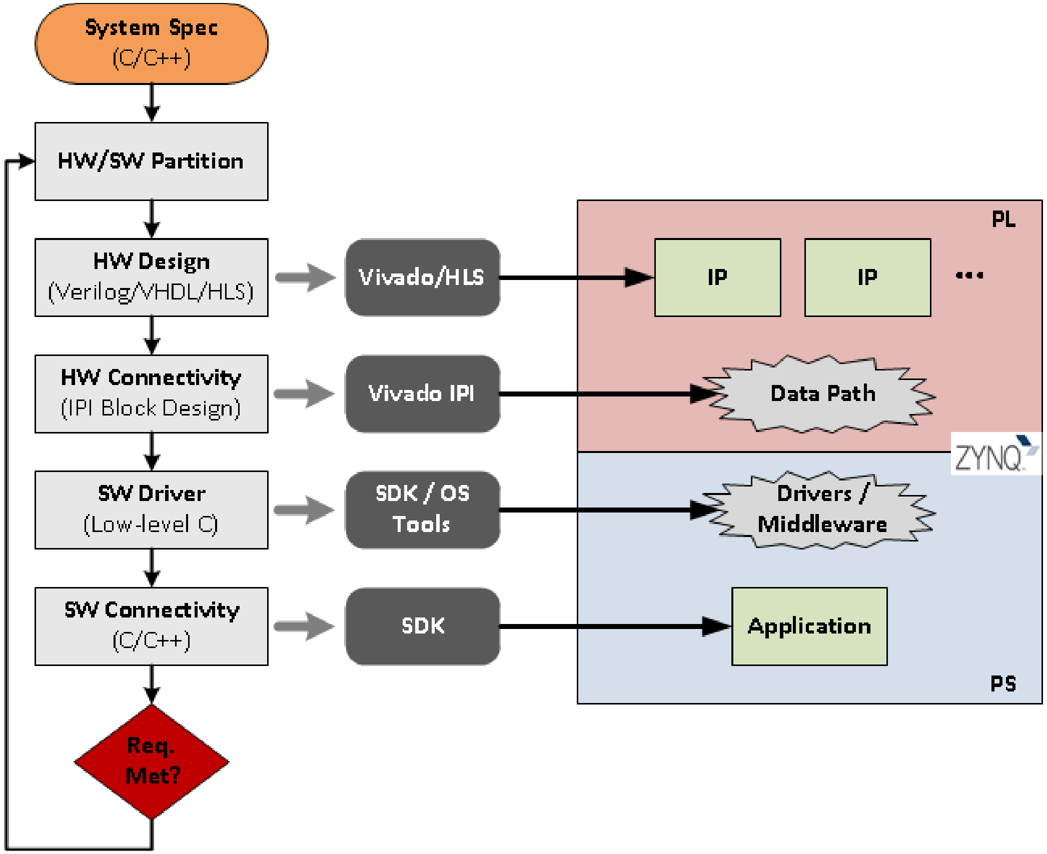
What is integrated inside vitis



Why "integrated development environment"? - Standard flow

Very High Level of expertise required:

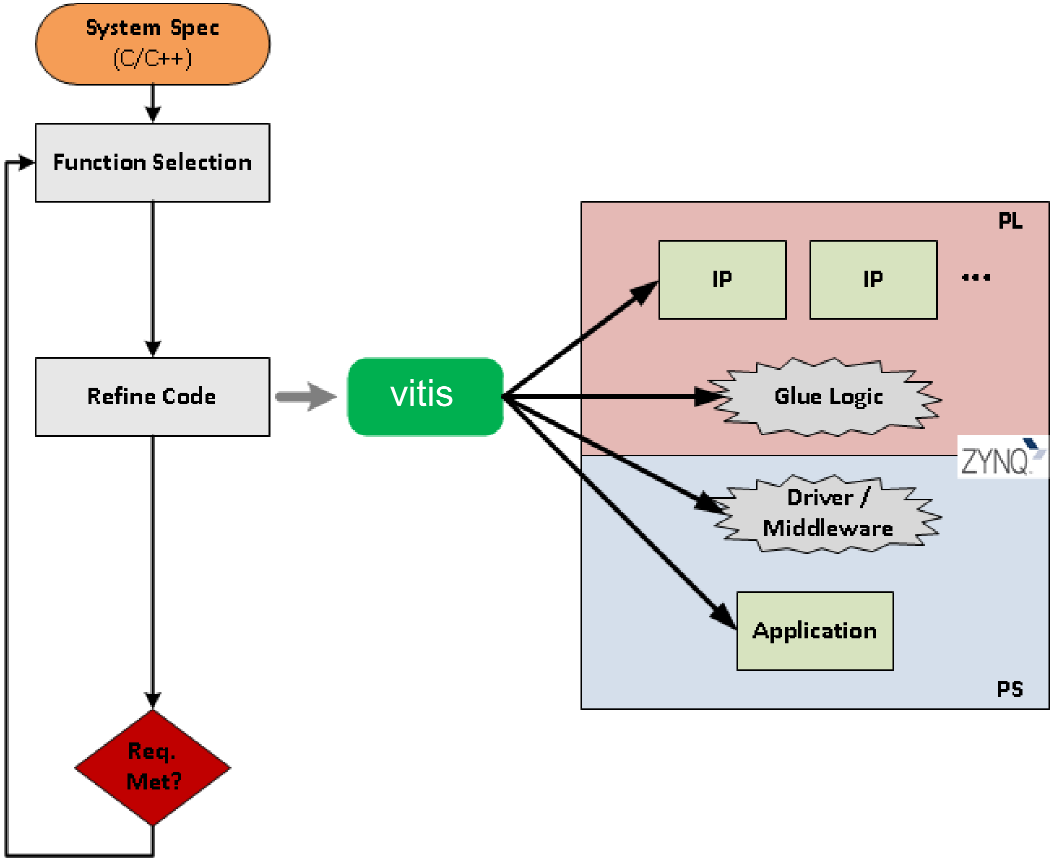
- Various hardware design entities
- Hardware connectivity at system level
- Driver development to drive custom hardware
- Integration in application and target OS



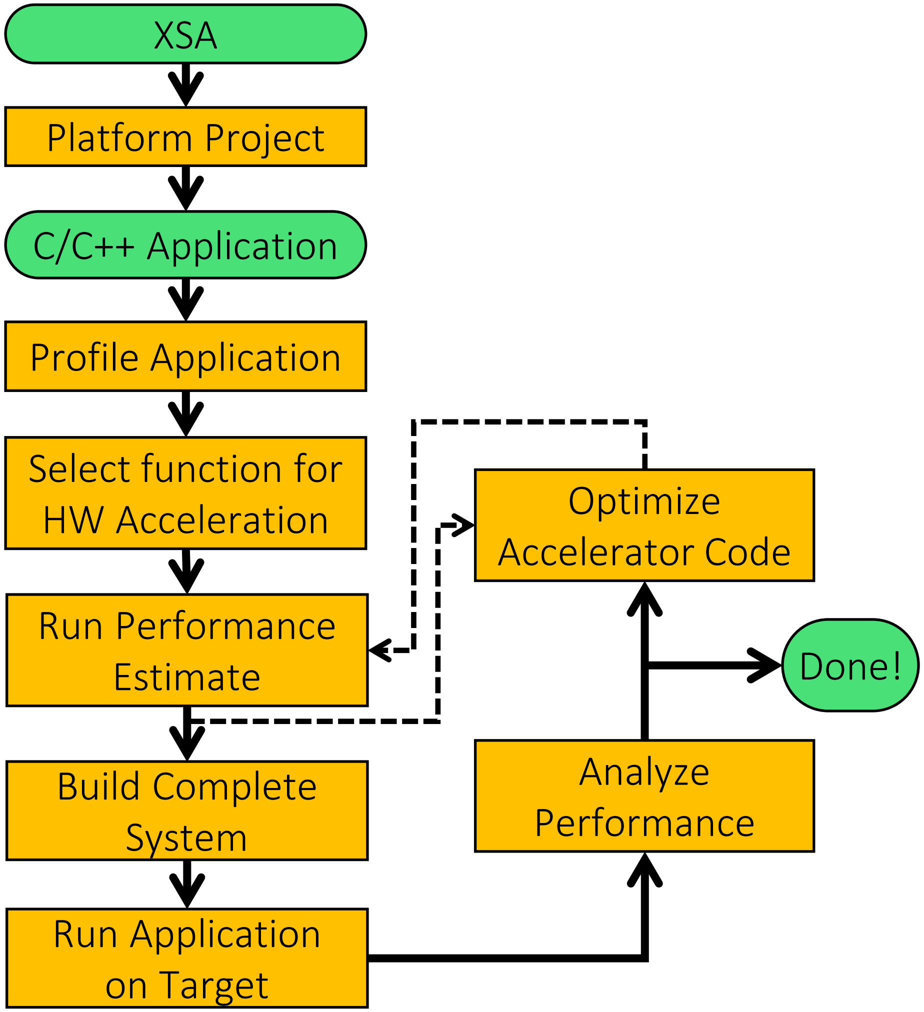
Why "integrated development environment"? - Flow with vitis

Vitis consolidates a multi-step/multi-tool process into a single tool and reduces hardware/software partitioning Code needs refinement!

Note: It is still complicated, as you will see in the exercise, but much less than it would be otherwise!



Vitis development flow overview



What gets accelerated?

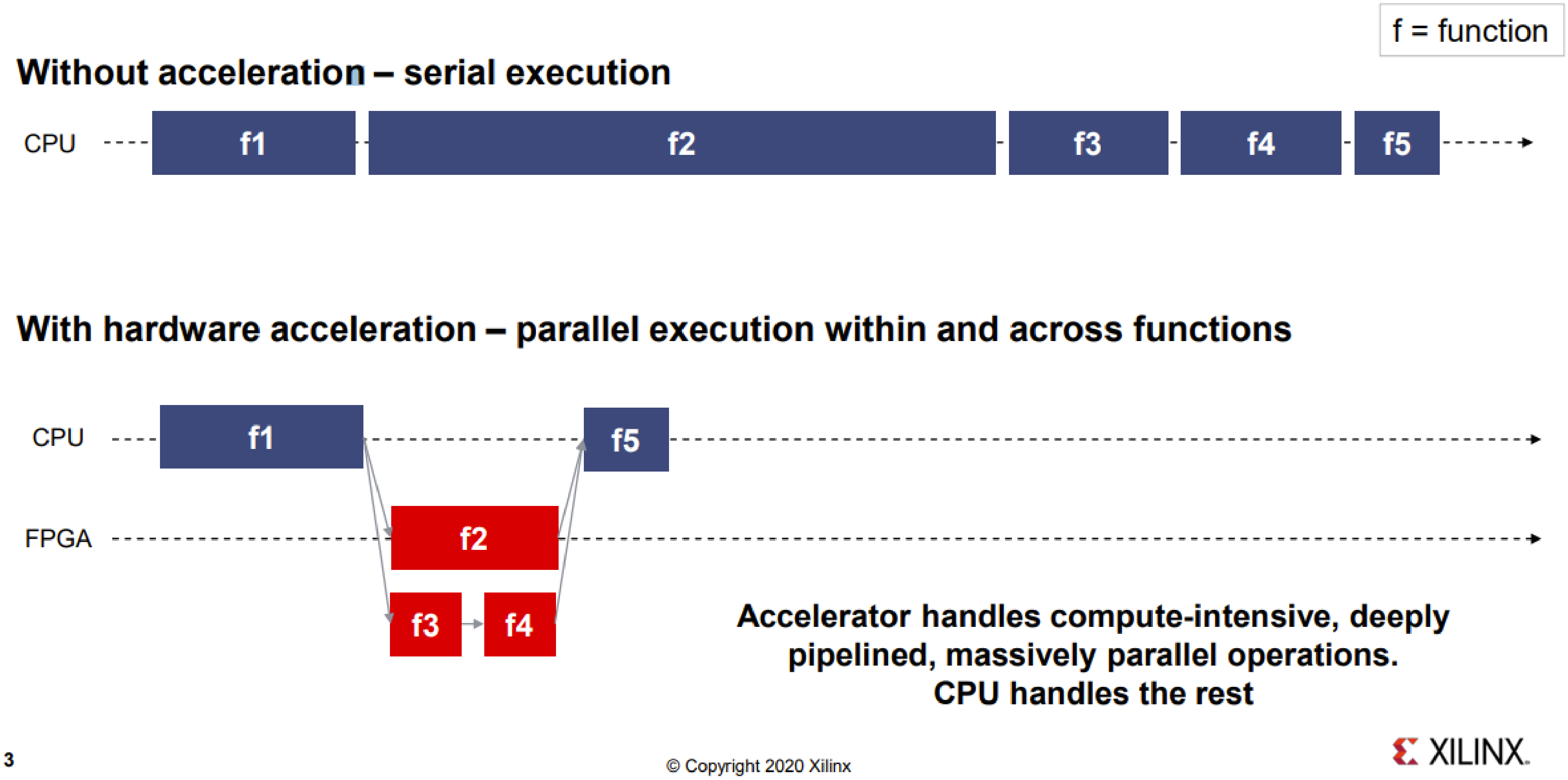
- Rule of thumb:

  Do All-in-Software first,
  *if **requirements** are not met,*
  evaluate hardware acceleration.

- No optimization without profiling!
  - Find performance bottlenecks, i.e. where speedup with HW acceleration could be made
  - A function is not automatically suited for HW acceleration just because it takes a lot of time to be executed!

- Trade-off between ***data movement cost*** and ***acceleration benefits***

What gets accelerated?



Without acceleration – serial execution

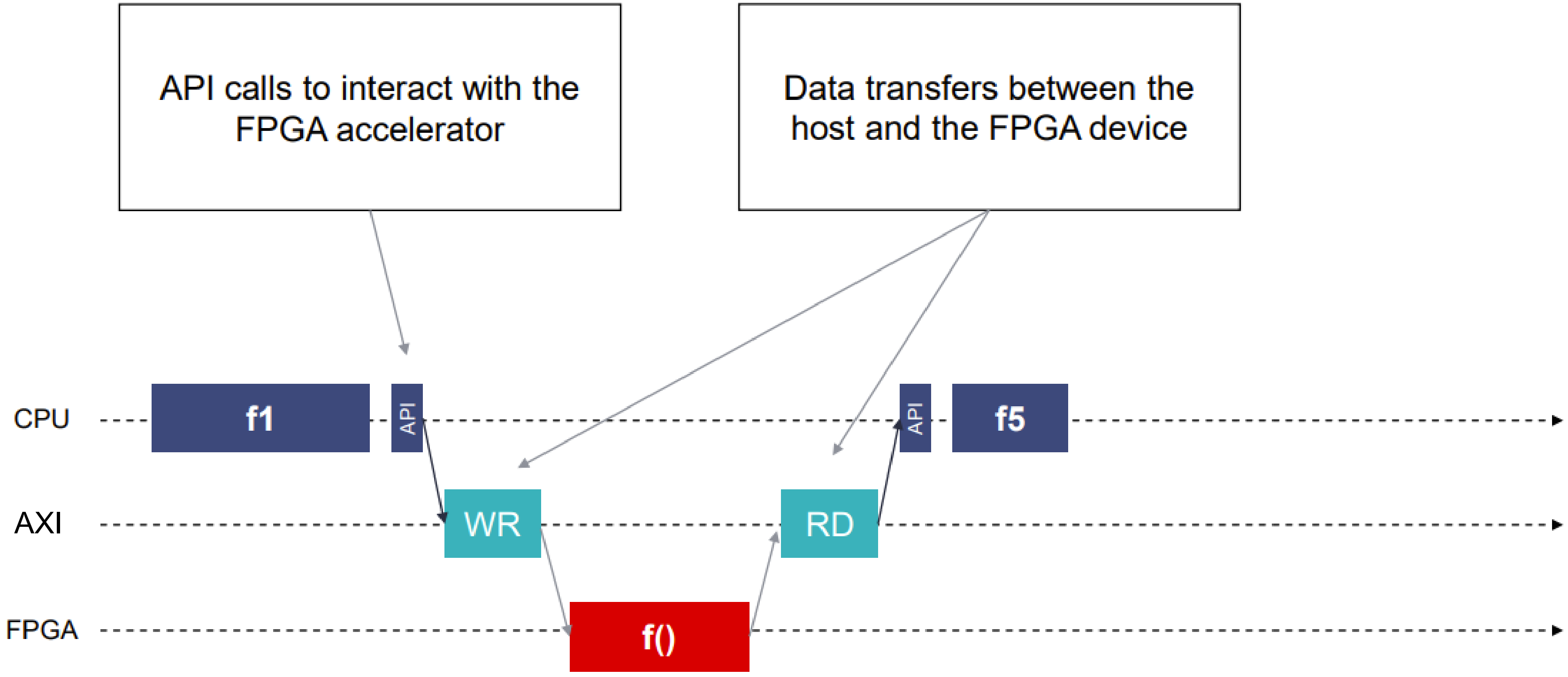
With hardware acceleration – parallel execution within and across functions

Accelerator handles compute-intensive, deeply pipelined, massively parallel operations. CPU handles the rest

3

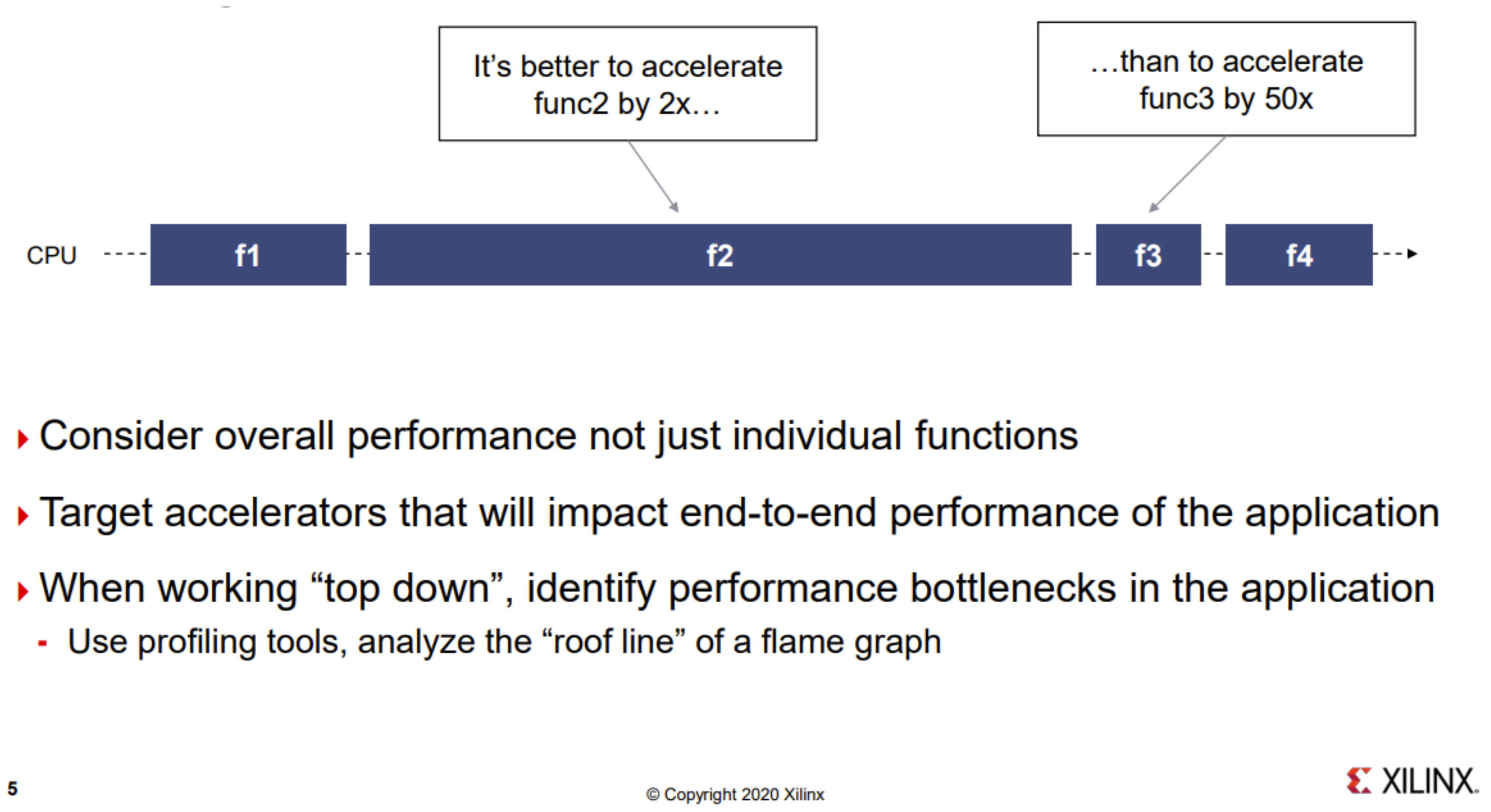
XILINX.

What gets accelerated?


API calls to interact with the FPGA accelerator
Data transfers between the host and the FPGA device
CPU
AXI
FPGA
f1
API
WR
f()
RD
API
f5

XILINX.

What should you accelerate?



It’s better to accelerate func2 by 2x…

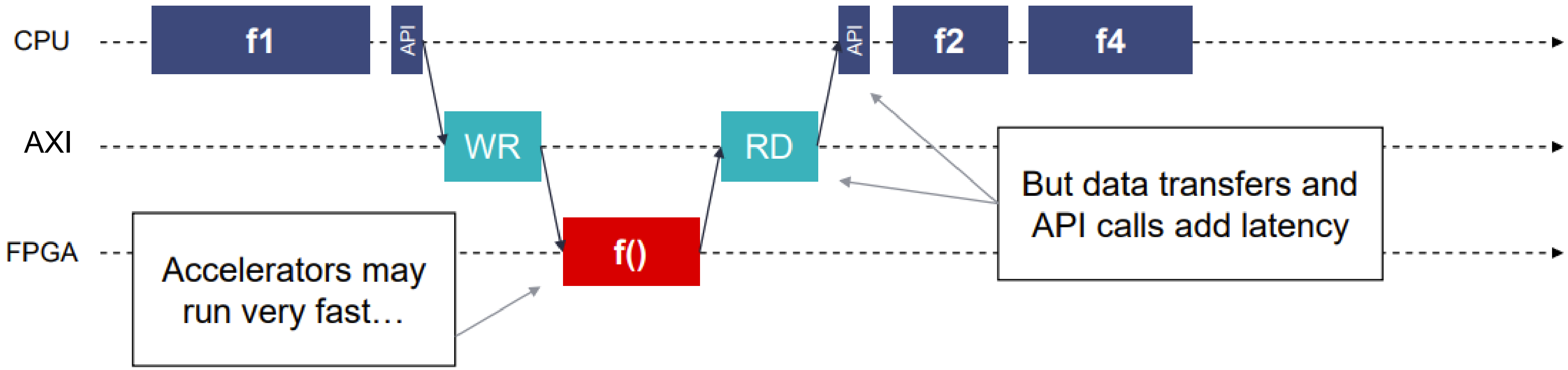
…than to accelerate func3 by 50x

- Consider overall performance not just individual functions
- Target accelerators that will impact end-to-end performance of the application
- When working “top down”, identify performance bottlenecks in the application
  - Use profiling tools, analyze the “roof line” of a flame graph

5

XILINX.

Which functions have potential?



Accelerators may run very fast…

But data transfers and API calls add latency

- Look for functions where {compute time} is much greater than {data transfer time}
  - Good: Monte Carlo – a few inputs, a lot of computations
  - Not so good: Vector addition – 2x more inputs than computations
- Functions that perform a lot of processing per invocation are preferable over small functions that are called many times
  - Minimizes API calls and event management overhead

Vitis development flow – in detail

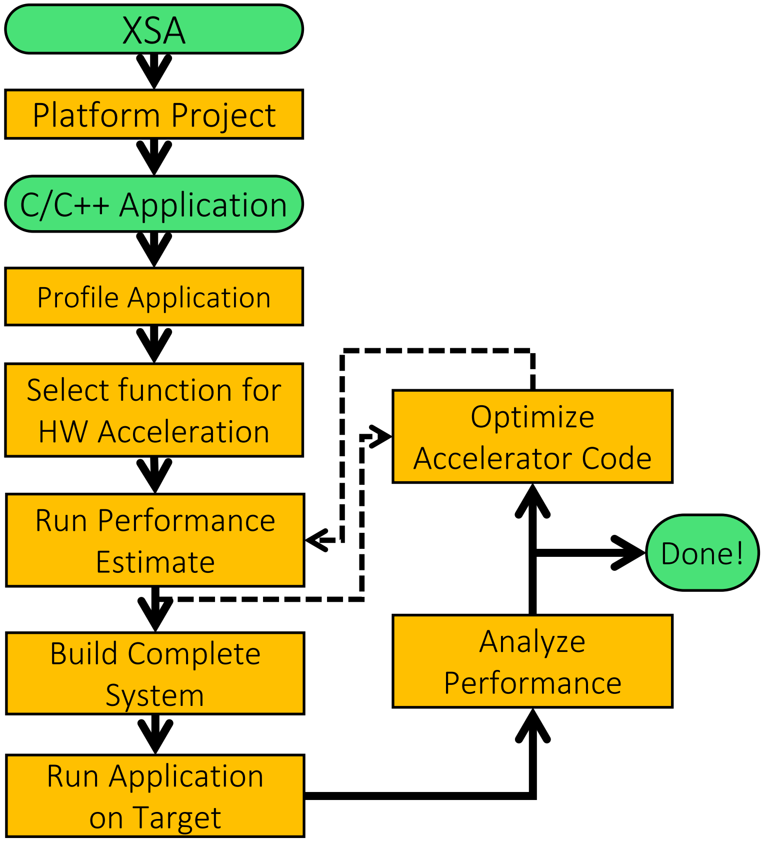
- Start with pure software system
  - Running on Zynq SoC (ARM cores)
- Profile application
  - In-system profiling (ARM)
- Select Functions for HW acceleration
  - Select C functions, will be fed to High-Level-Synthesis
- Performance estimate
  - Performance will be most likely be very bad before accelerator code & data transfer is optimized

Optimize until estimate meets requirements

- Build complete system
  - Fully automated, but takes a long time (hours)
- Analyze performace

Optimize further if performance not met

(same structure as the exercise)



How does the V++ compiler map programs to HW/SW?

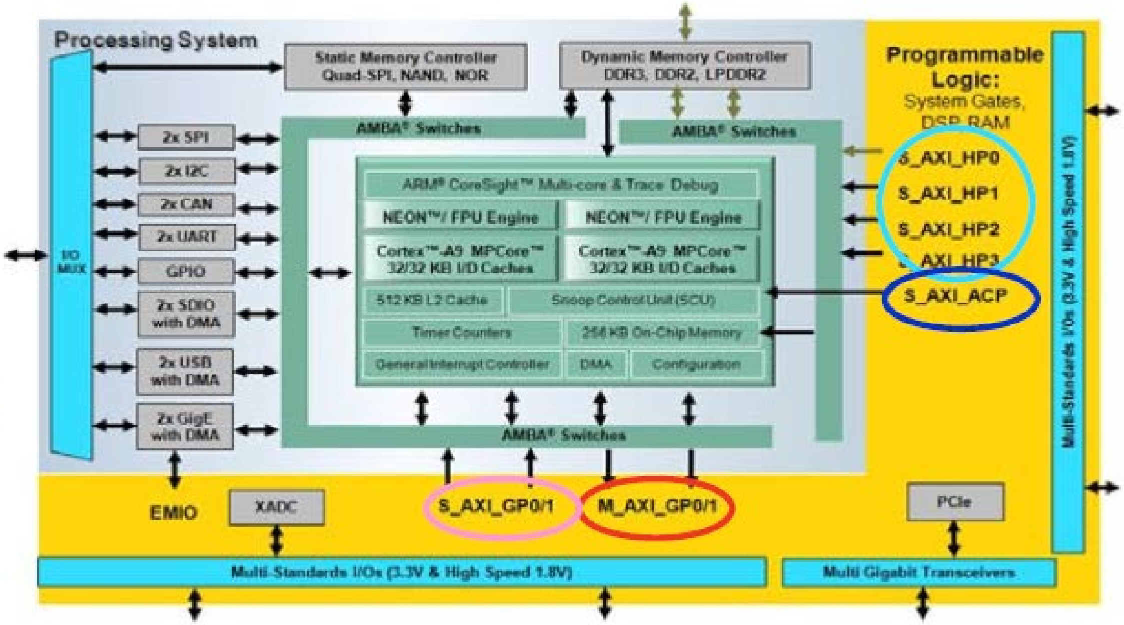
- Summary: What is needed to accelerate a ‘function’:
  - Accelerator (HLS or HDL)
  - Data Motion Network
  - Software Driver and Call

Data movement

- How do you interact with your accelerator?
- Which PS interface should be used?
  - High Performance, General Purpose, Cache Coherence Port
- Where does the input/output of the accelerator connect to?
  - DDR/PS, external interface
- Which is the optimal data movement techniques?
  - DMA-driven (simple or scatter gather)
  - Software driven
  - How is cache coherency managed?
  - What is the used memory model (contiguous or paged?)
- How is the signaling between the PS and the accelerator implemented?
  - Interrupt-driven, event interface, polled?

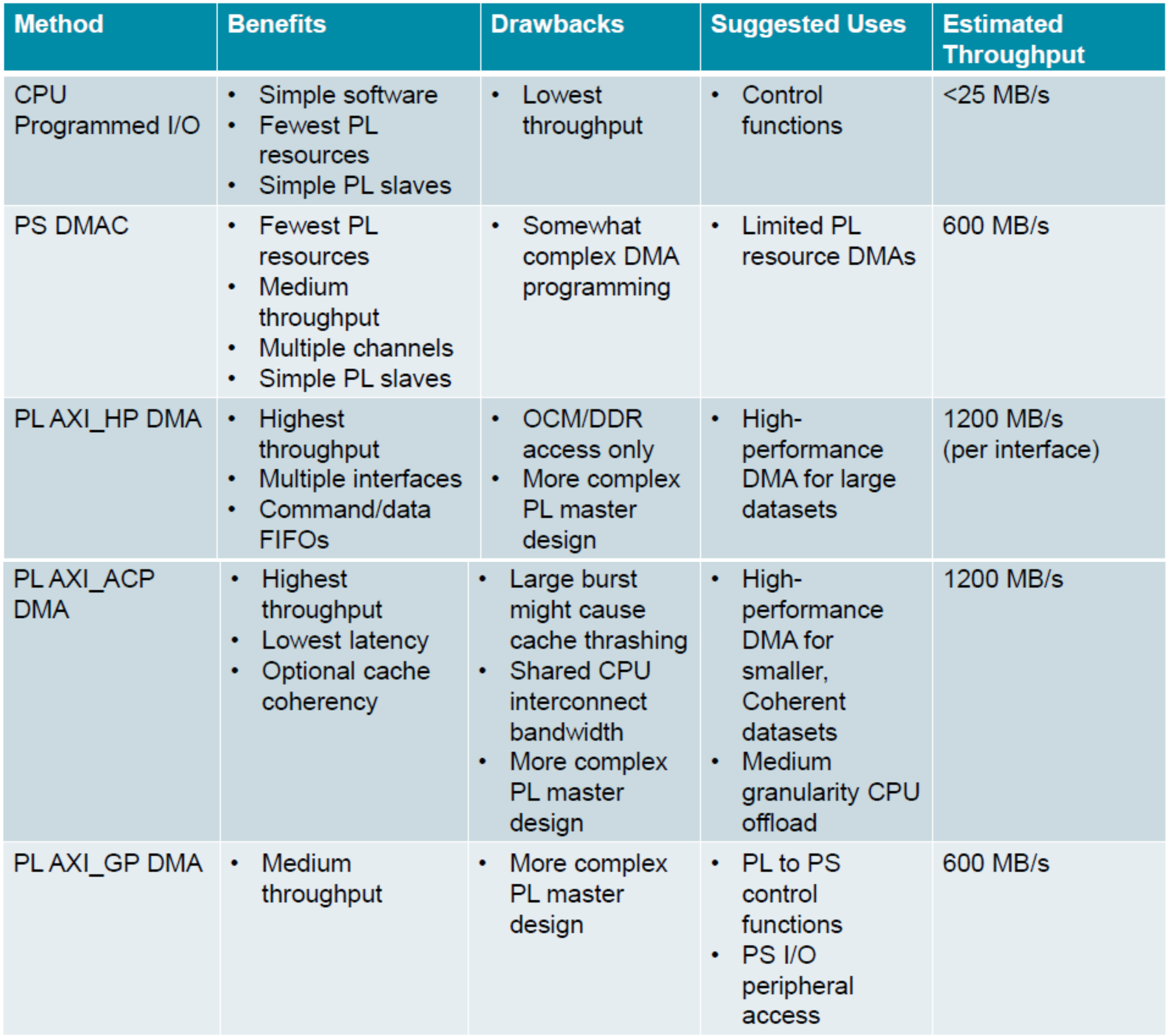
Data movement

- The AMBA AXI ports of the PS-PL interface provide the primary mechanism for the flow of data between the PS and PL
  - Two general-purpose master ports
  - Two general-purpose slave ports
  - Four high-performance slave ports
  - One accelerator coherency port (ACP) slave port



(depends on the SoC used...)

Data movement methods comparison



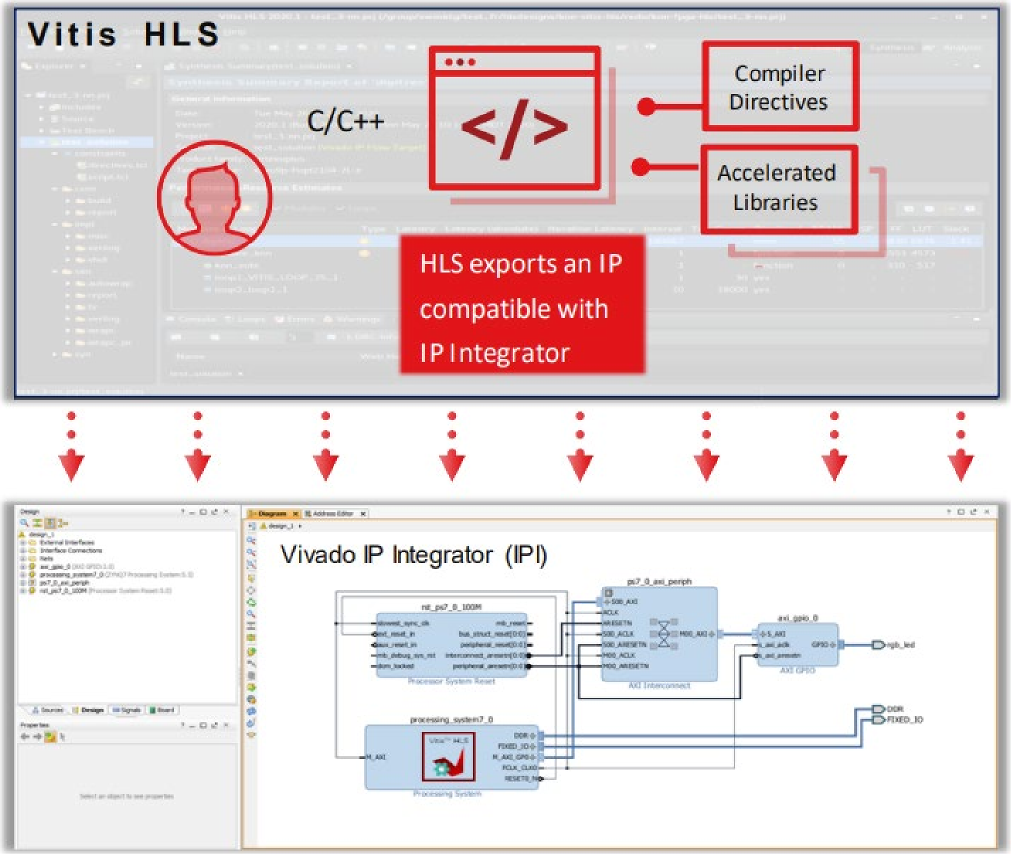
Method	Benefits	Drawbacks	Suggested Uses	Estimated Throughput
CPU Programmed I/O	• Simple software • Fewest PL resources • Simple PL slaves	• Lowest throughput	• Control functions	<25 MB/s
PS DMAC	• Fewest PL resources • Medium throughput • Multiple channels • Simple PL slaves	• Somewhat complex DMA programming	• Limited PL resource DMAs	600 MB/s
PL AXI_HP DMA	• Highest throughput • Multiple interfaces • Command/data FIFOs	• OCM/DDR access only • More complex PL master design	• High-performance DMA for large datasets	1200 MB/s (per interface)
PL AXI_ACP DMA	• Highest throughput • Lowest latency • Optional cache coherency	• Large burst might cause cache thrashing • Shared CPU interconnect bandwidth • More complex PL master design	• High-performance DMA for smaller, Coherent datasets • Medium granularity CPU offload	1200 MB/s
PL AXI_GP DMA	• Medium throughput	• More complex PL master design	• PL to PS control functions • PS I/O peripheral access	600 MB/s

How does the V++ compiler map programs to HW/SW?

- Summary: What is needed to accelerate a ‘function’:
  - **Accelerator (HLS or HDL)**
  - Data Motion Network
  - Software Driver and Call

High-level synthesis: HLS

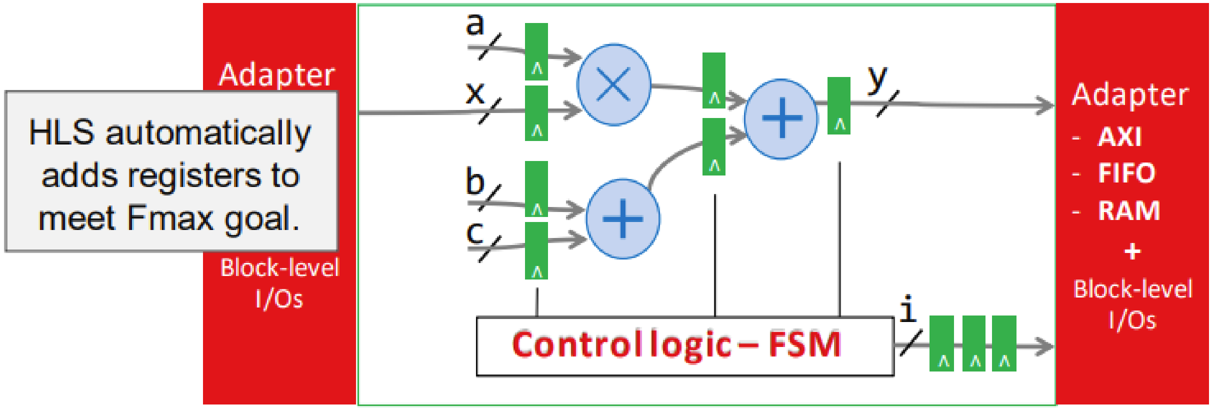
- High-Level Synthesis
  - Creates an RTL implementation from C, C++, System C, OpenCL API C kernel code
  - Extracts control and dataflow from the source code
  - Implements the design based on defaults and user applied directives
- Many implementation are possible from
  the same source description
  - Smaller designs, faster designs, optimal designs
  - Enables design exploration



HLS exports an IP compatible with IP Integrator

C code to kernel or IP

```
void f(int in[4], int out[4]) {
  int a,b,c,x,y;
  for(int i = 0; i < 4; i++) {
    x = in[i]; y = a*x + b + c; out[i] = y;
  }
```



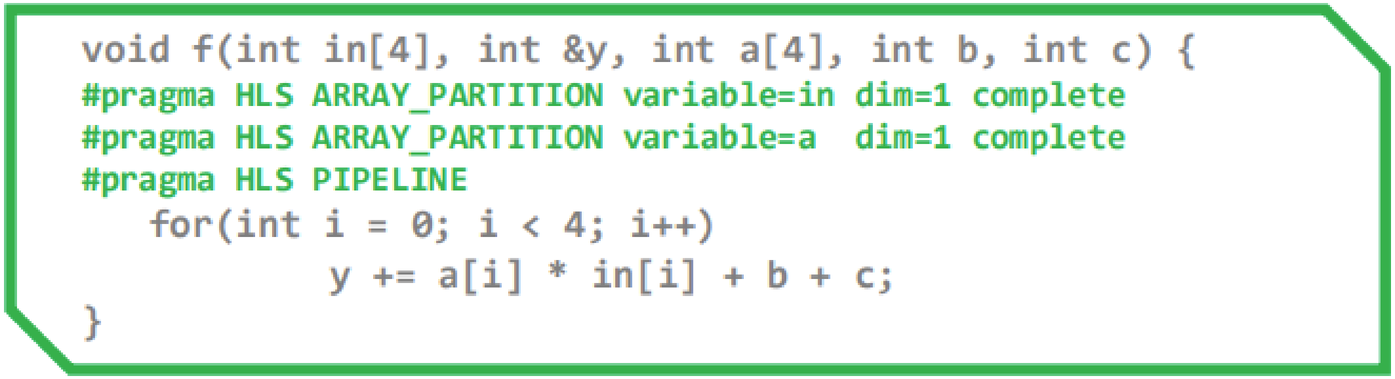
HLS automatically adds registers to meet Fmax goal.

>> 10

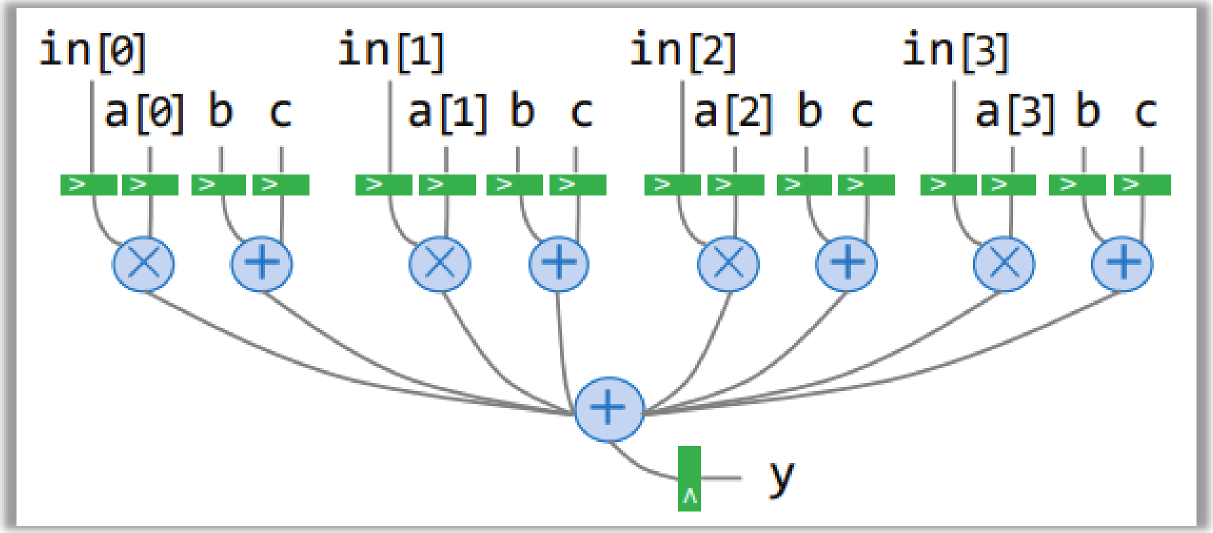
XILINX

Design exploration via pragmas

> Pragmas change the circuit topology…

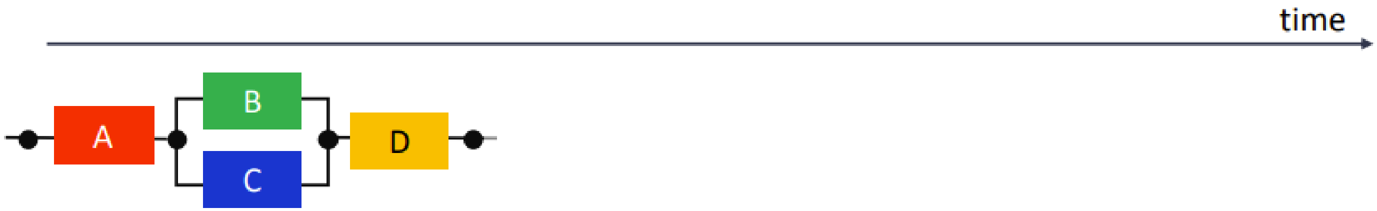


```
void f(int in[4], int &y, int a[4], int b, int c) {
#pragma HLS ARRAY_PARTITION variable=in dim=1 complete
#pragma HLS ARRAY_PARTITION variable=a  dim=1 complete
#pragma HLS PIPELINE
   for(int i = 0; i < 4; i++)
            y += a[i] * in[i] + b + c;
}
```

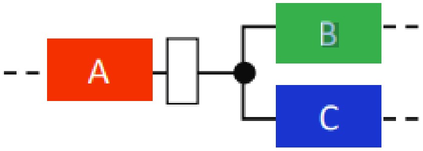


>> 11

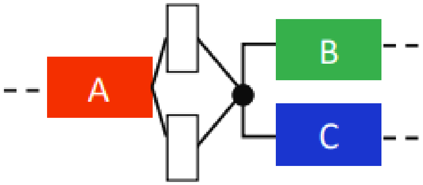
Task parallelism - example



“diamond” shape connectivity.
3 iterations.



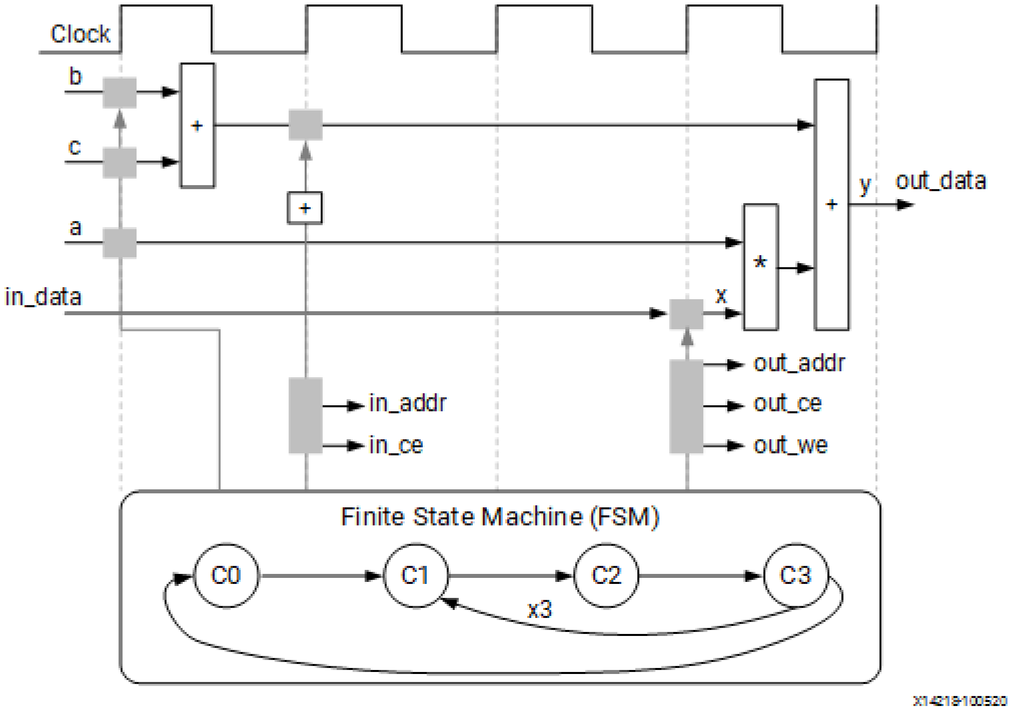
By default…
As **A** produces its buffer, **B** and **C** wait…
Then **B** and **C** consume and **A** waits…



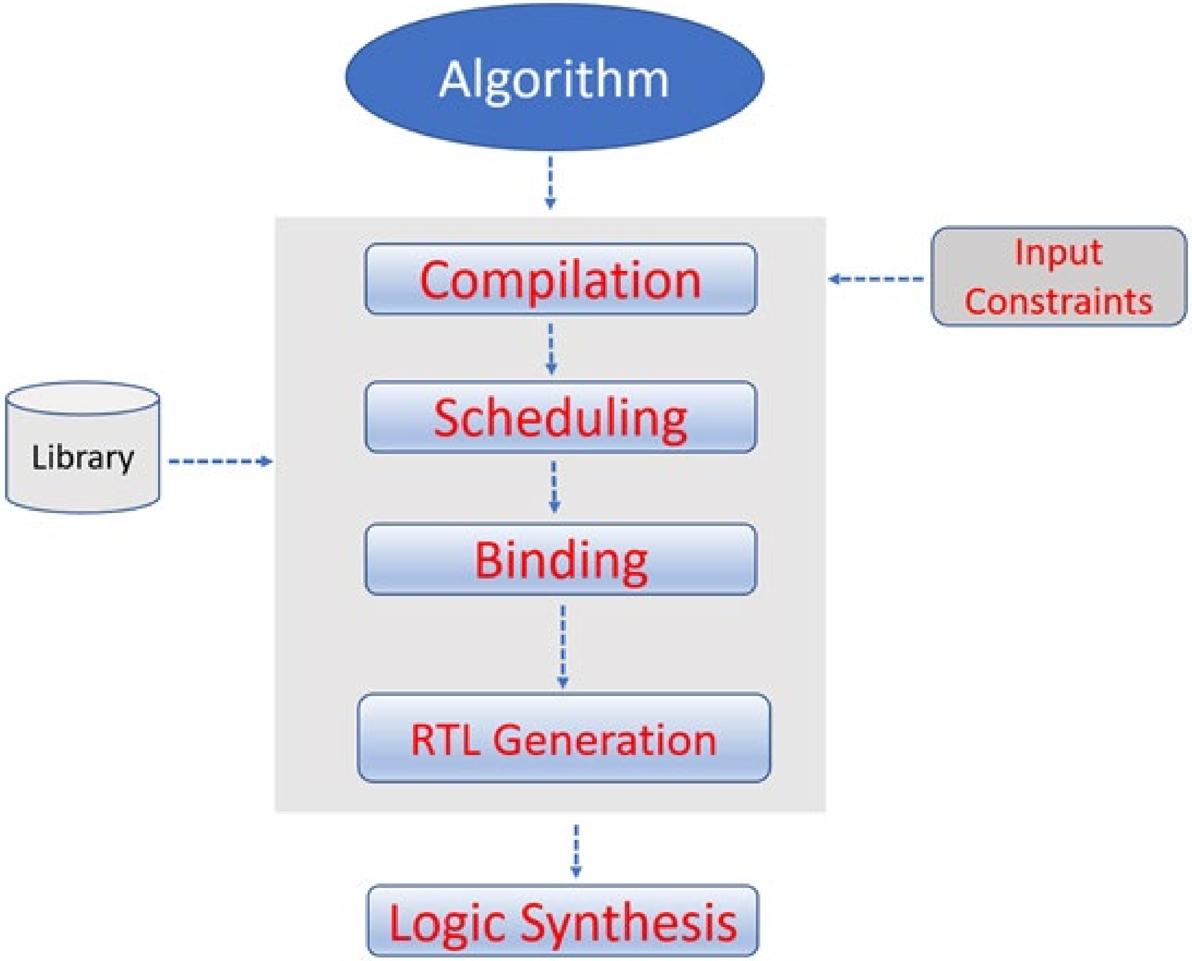
With **ping-pong** buffers…
A produces in one buffer, **B** and **C** read the other.
Next invocation, buffers switch roles: tasks can work continuously

HLS – Control logic extraction

```
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;
  }
}
```

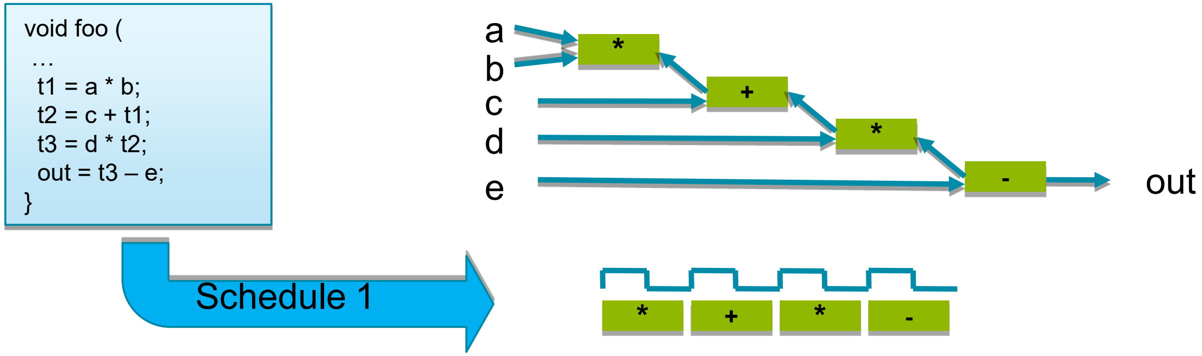


HLS – tasks


Algorithm
Compilation
Input Constraints
Library
Scheduling
Binding
RTL Generation
Logic Synthesis

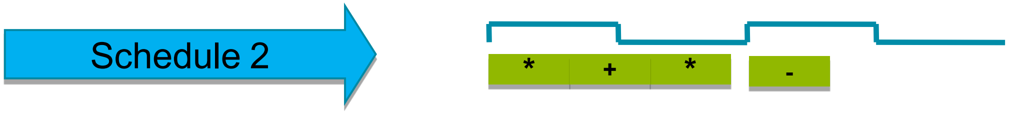
Scheduling

- The operations in the control flow graph are mapped into clock cycles



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

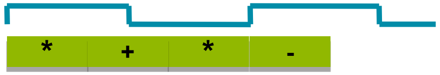
- The technology and user constraints impact the schedule
  - A faster technology (or slower clock) may allow more operations to occur in the same clock cycle



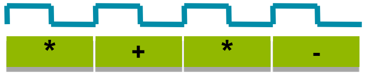
- The code also impacts the schedule
  - Code implications and data dependencies must be obeyed

Binding

- Binding is where operations are mapped to cores from the hardware library
  - Operators map to cores
- Binding Decision: to share
  - Given this schedule:



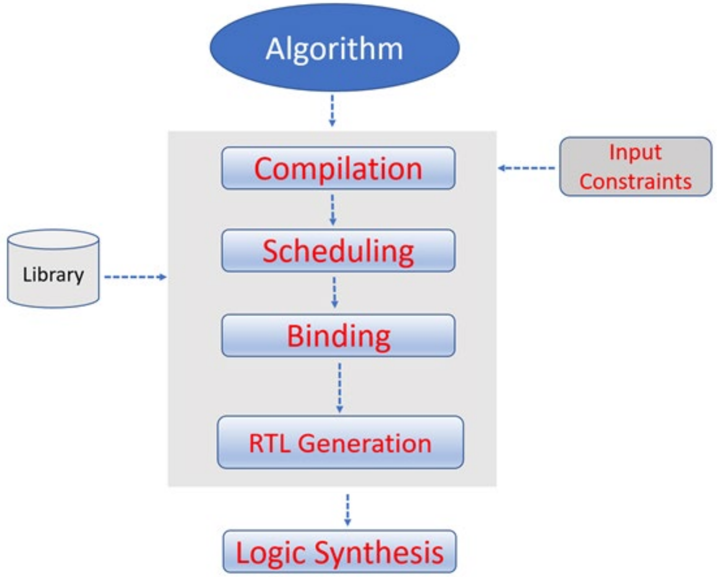
    - Binding must use 2 multipliers, since both are in the same cycle
    - It can decide to use an adder and subtractor or *share* one addsub
- Binding Decision: or not to share
  - Given this schedule:

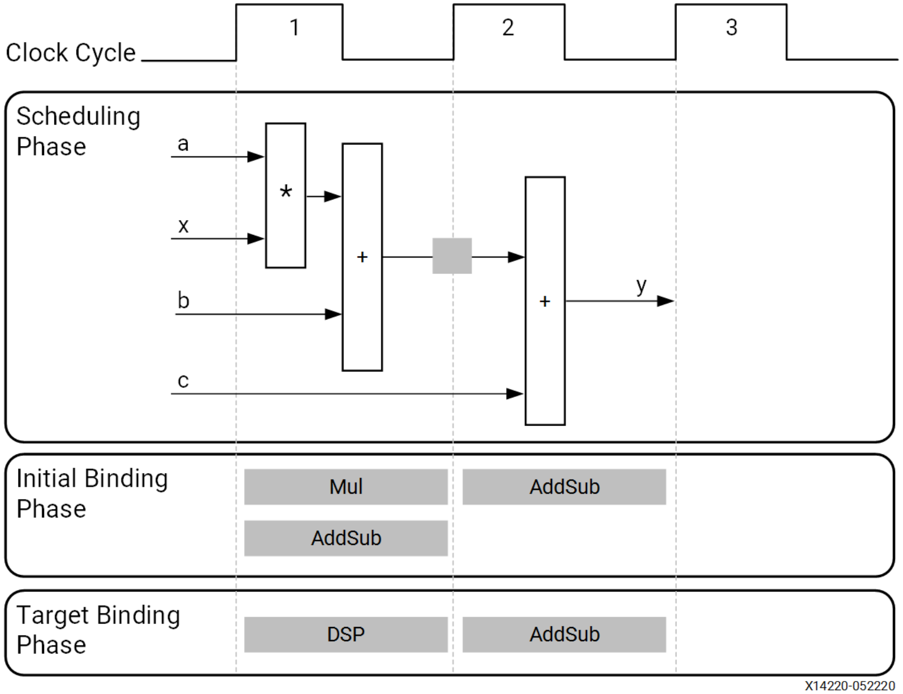


    - Binding may decide to share the multipliers (each is used in a different cycle)
    - Or it may decide the cost of sharing (muxing) would impact timing and it may decide *not to share* them
    - It may make this same decision in the first example above too

HLS – binding

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





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 …

Functions & RTL hierarchy

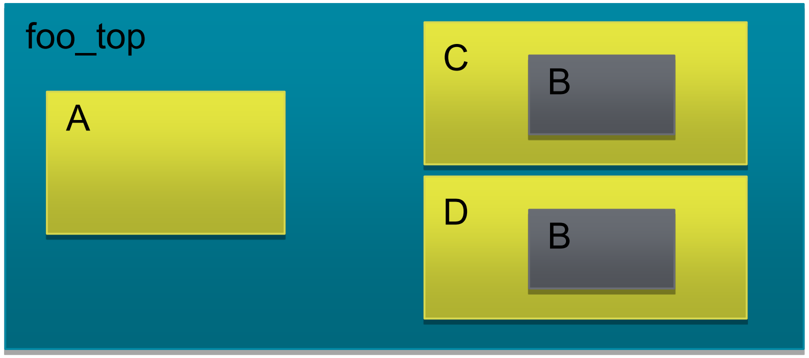
- Each function is translated into an RTL block
  - Verilog module, VHDL entity

Source Code

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


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

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

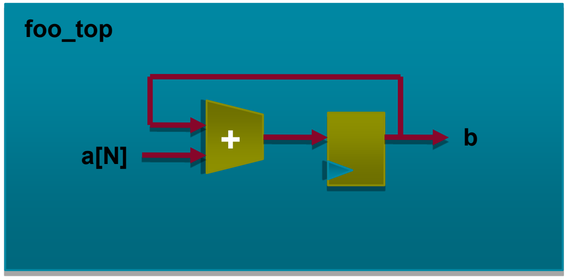
Loops

- By default, loops are rolled
  - 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)

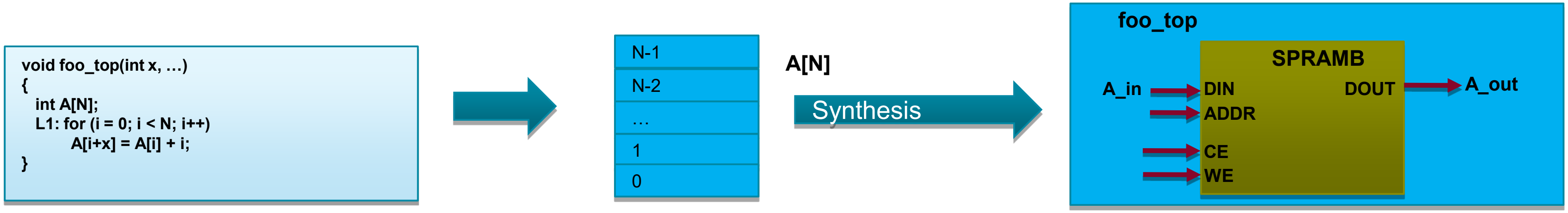
Synthesis



- 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

Arrays in HLS

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

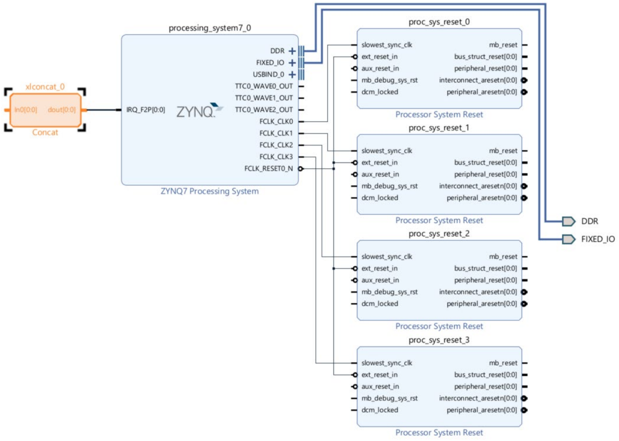


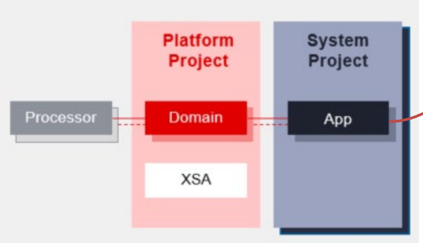
```
void foo_top(int x, …)
{
  int A[N];
  L1: for (i = 0; i < N; i++)
        A[i+x] = A[i] + i;
}
```

- 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 Vivado HLS Library Guide
- 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

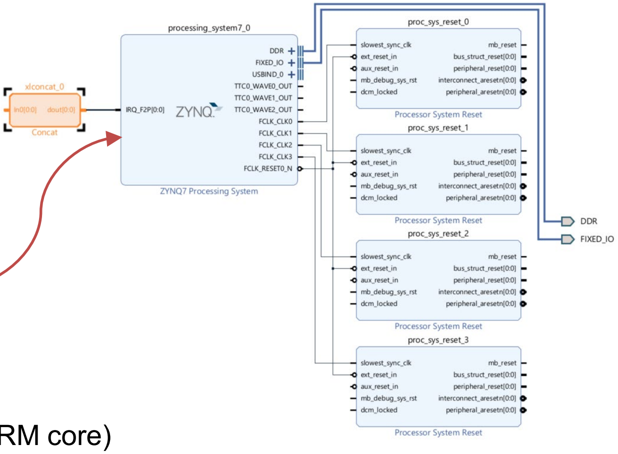
Vitis flows:

Original Block diagram

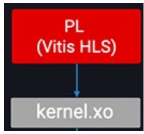




Application project (running on ARM core)

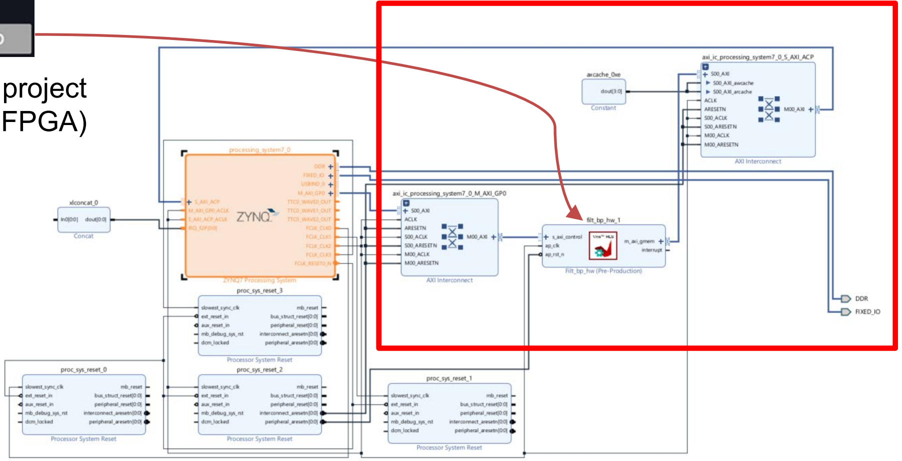


SW programming



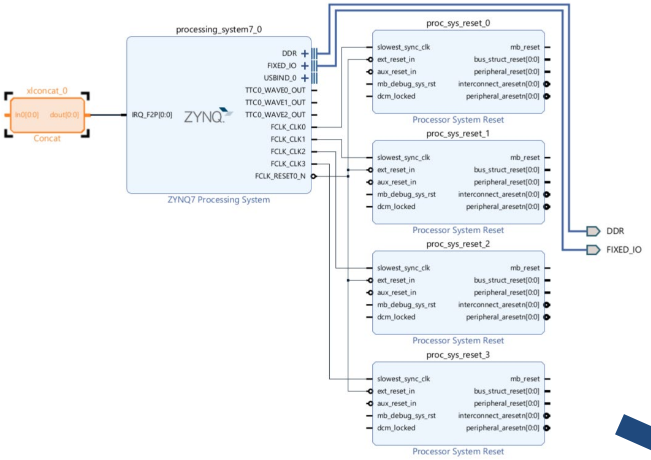
HLS Kernel project (running on FPGA)

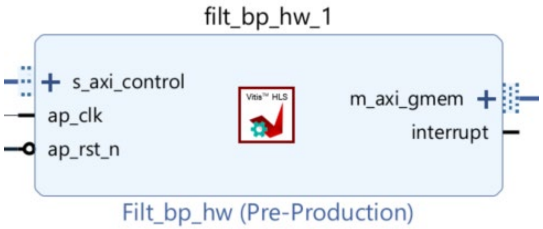
This is automagically generated by Vitis HLS!

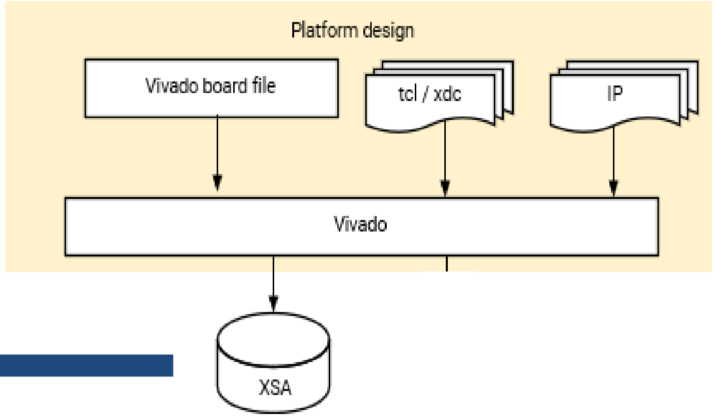


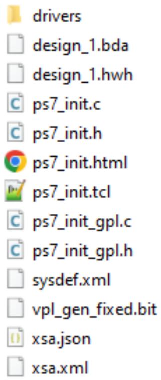
SW + HW programming

How is my hardware packaged for SW development: XSA archive

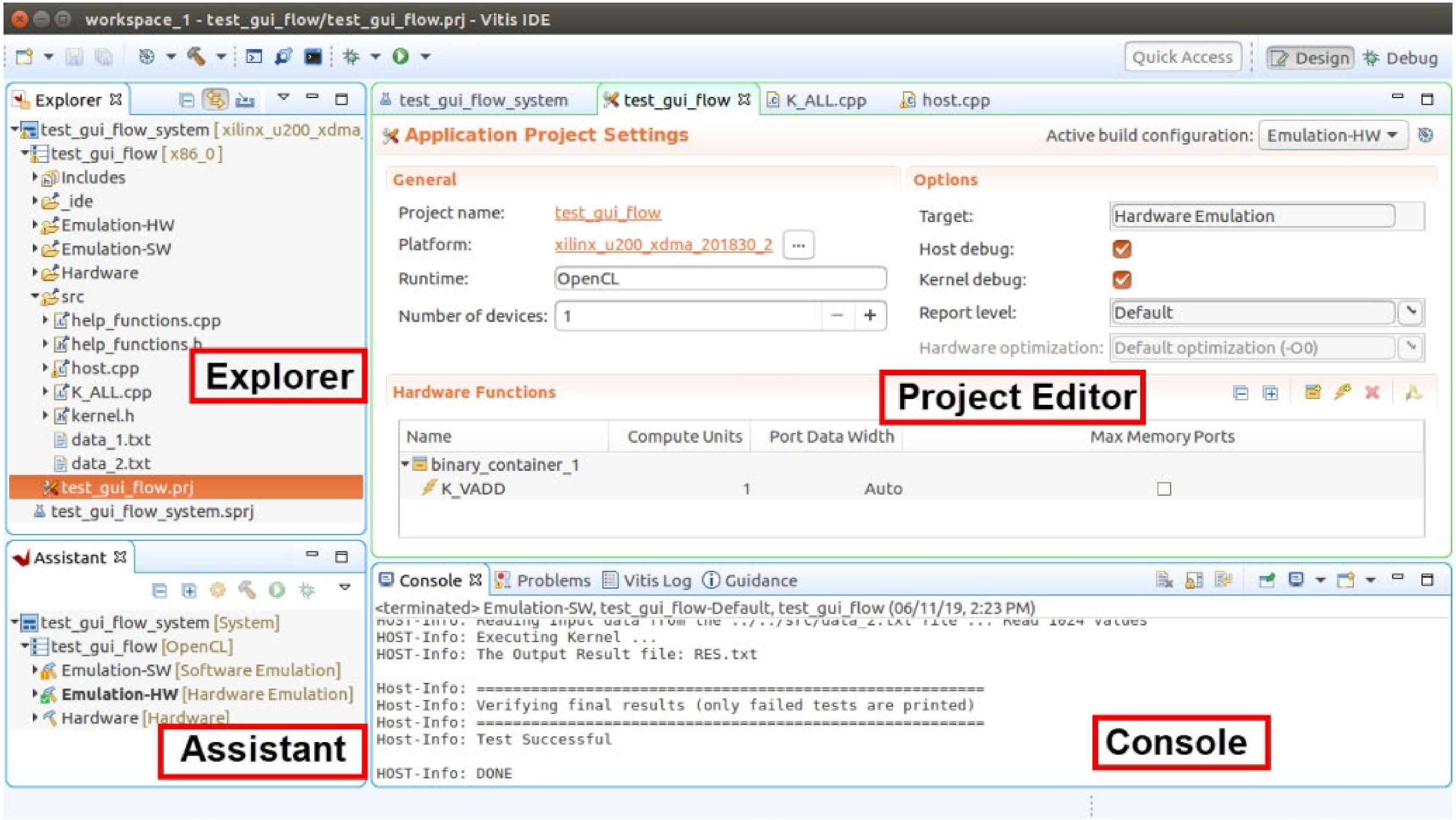







drivers
design_1.bda
design_1.hwh
ps7_init.c
ps7_init.h
ps7_init.html
ps7_init.tcl
ps7_init_gpl.c
ps7_init_gpl.h
sysdef.xml
vpl_gen_fixed.bit
xsa.json
xsa.xml

Vitis IDE overview:



Application Project Settings

Active build configuration: Emulation-HW

General

Project name: test_gui_flow

Platform: xilinx_u200_xdma_201830_2

Runtime: OpenCL

Number of devices: 1

Options

Target: Hardware Emulation

Host debug: ☑

Kernel debug: ☑

Report level: Default

Hardware optimization: Default optimization (-O0)

Hardware Functions

Name	Compute Units	Port Data Width	Max Memory Ports
binary_container_1			
K_VADD	1	Auto	□

```
<terminated> Emulation-SW, test_gui_flow-Default, test_gui_flow (06/11/19, 2:23 PM)
HOST-Info: Reading input data from the ../../src/data_2.txt file ... Read 1024 values
HOST-Info: Executing Kernel ...
HOST-Info: The Output Result file: RES.txt

Host-Info: ============================================================
Host-Info: Verifying final results (only failed tests are printed)
Host-Info: ============================================================
Host-Info: Test Successful

HOST-Info: DONE
```

© Copyright 2020 Xilinx

XILINX

References

- L. H. Crockett, R. A. Elliot, M. A. Enderwitz and R. W. Stewart, *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*, First Edition, Strathclyde Academic Media, 2014.
- Official Xilinx SDSoC Training Slides
- Official Vitis HLS Training Slides