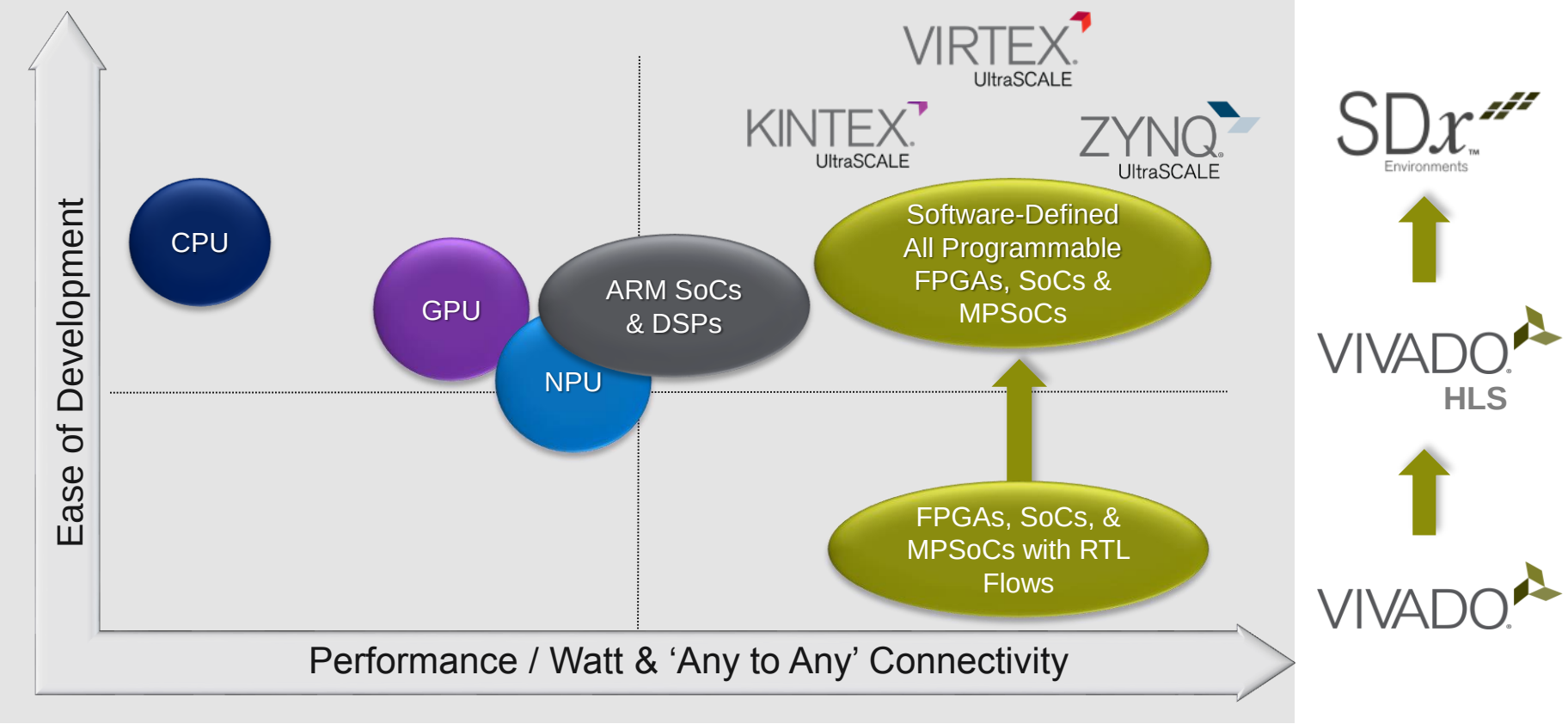
XILINX
ALL PROGRAMMABLE™

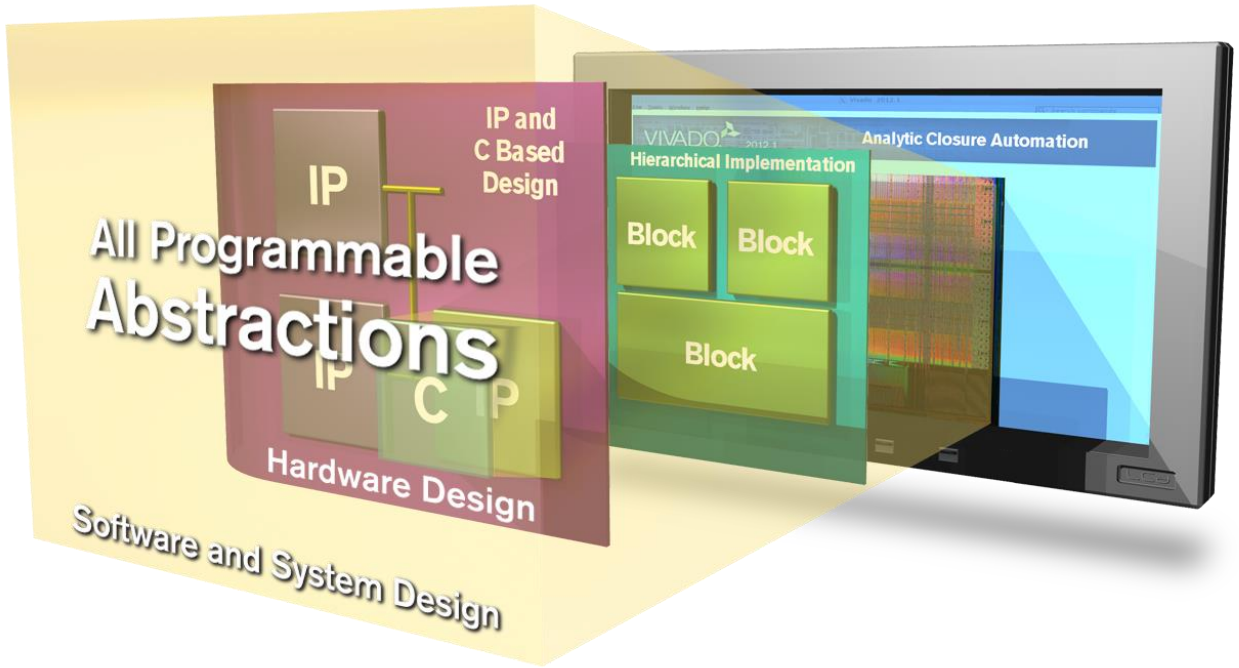
High-level programming tools for FPGA

Smarter Systems Compute Acceleration Options



Note: Software programmable devices often paired with FPGA for connectivity and co-processing

Enabling Smarter Systems

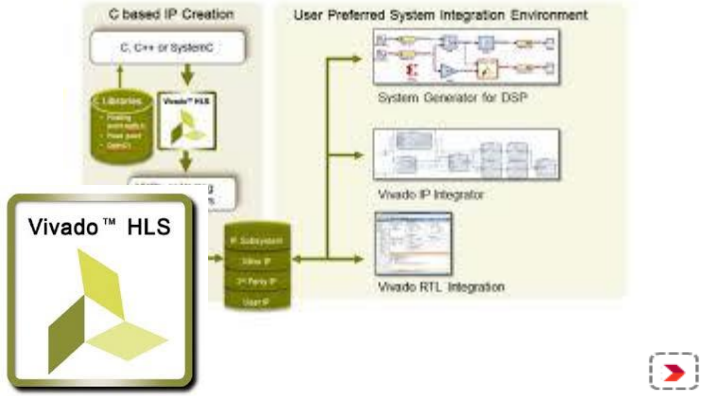


- **All Programmable Abstractions**
  - Software, model, platform and IP-based design environments for system, software and hardware developers
  - Improve productivity and access to All Programmable FPGA, SoCs and 3D IC

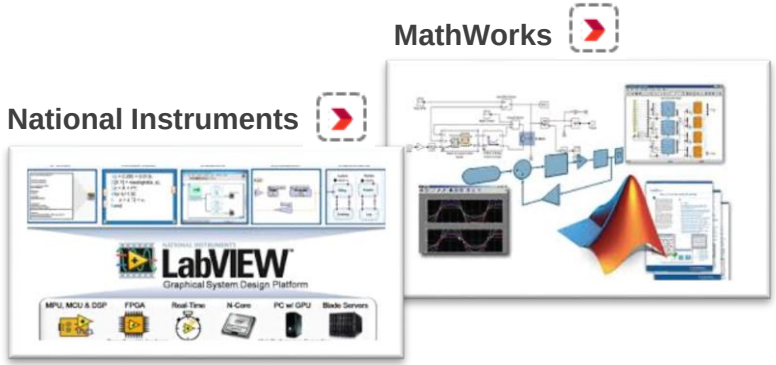
Enabling Smarter Systems

Abstractions

Software Automation



Hardware Automation



System Automation

SDSoC: Development Environment

Replace SoCs with the 100x performance gain of Zynq SoCs and MPSoCs

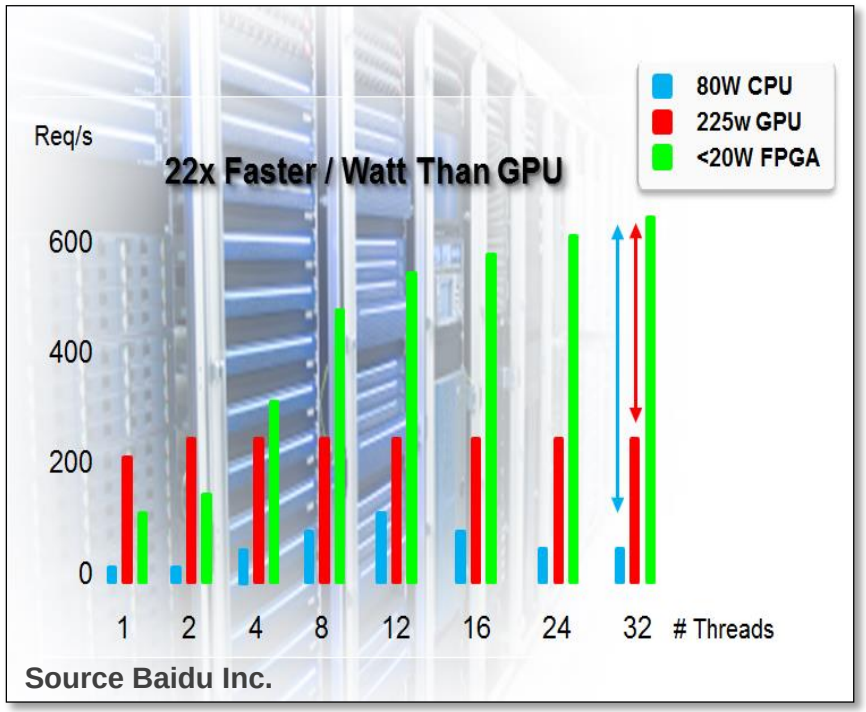
Algorithm	CPU/DSP	Zynq	Advantage
Forward projection	ARM: 3 sec/view	0.016 sec/view	188x
Motion detection	ARM: 0.7 FPS	67 FPS	90x
Noise reduction-Sobel	ARM: 1 FPS	67 FPS	60x
Canny edge detection	ARM: 0.66 FPS	40 FPS	45x
3D image reconstruction	ARM: 75k	8k	9x
DPD	ARM: 506 ms	31.3 ms	16x
FIR	TI DSP: 64020 ns	1200 ns	53x
FFT	TI DSP: 1036 ns	128 ns	8x

ARM = ARM Cortex A9 @ 800MHz
TI DSP = C66X

- **ASSP-like programming experience**
  - Eclipse™-based IDE
- **System level profiling**
  - Rapid system performance estimation
  - Automated performance measurement
- **Full system optimizing compiler**
  - Software configurable acceleration using C/C++
- **Also useful for system architects**
  - Rapid "what if" architectural tradeoffs

SDAccel: Development Environment

Replace CPU/GPU with up to 25X performance/watt value of FPGAs

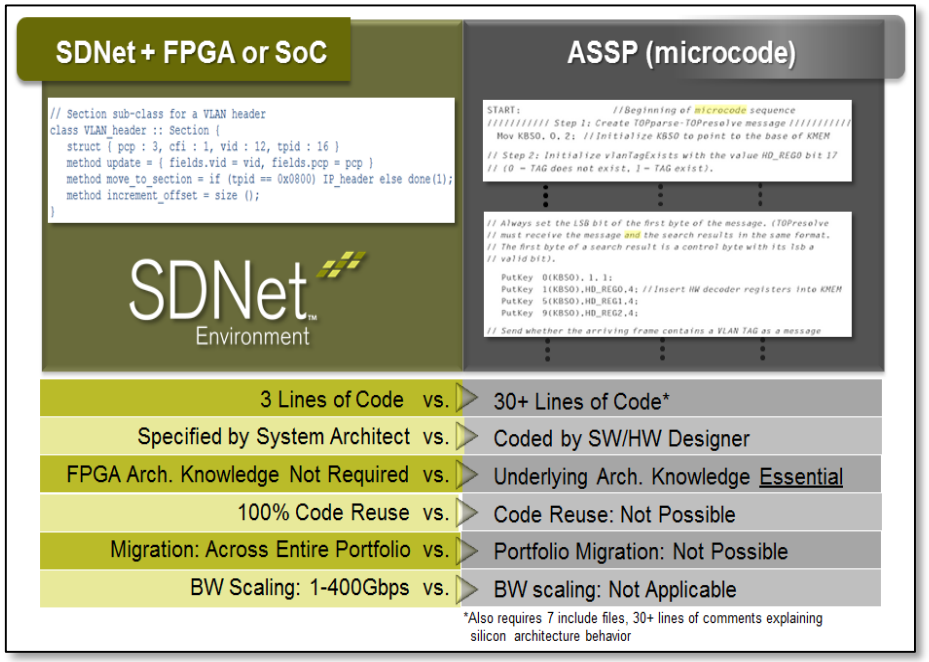


Ideal for data center, A&D and medical imaging accelerated computing

- **CPU/GPU like development experience**
  - OpenCL, C/C++ Kernel acceleration
  - Khronos OpenCL conformant
- **Architecturally optimizing compiler**
  - 25x Performance/Watt advantage over GPUs / CPUs
- **Complete CPU/GPU like run-time experience**
  - Run-time reconfigurable accelerators
  - Production ready platforms

SDNet: SW Defined Specification Environment

Replaces NPUs with All Programmable performance, flexibility, and security



10x productivity advantage

- **‘Softly Defined’ networks**
  - Content intelligence data plane hardware supporting hit-less in service updates
  - Dynamically collaborates with control plane
  - Address performance, flexibility, and security challenges of agile service-oriented networking
- **SDNet’s specification driven environment**
  - Generates packet data plane HW subsystem
  - Generates subsystem firmware
  - Generates optimized packet processing engines and flows
    - e.g. parsing, editing, search, and feature optimized QoS policy
  - Generates test benches

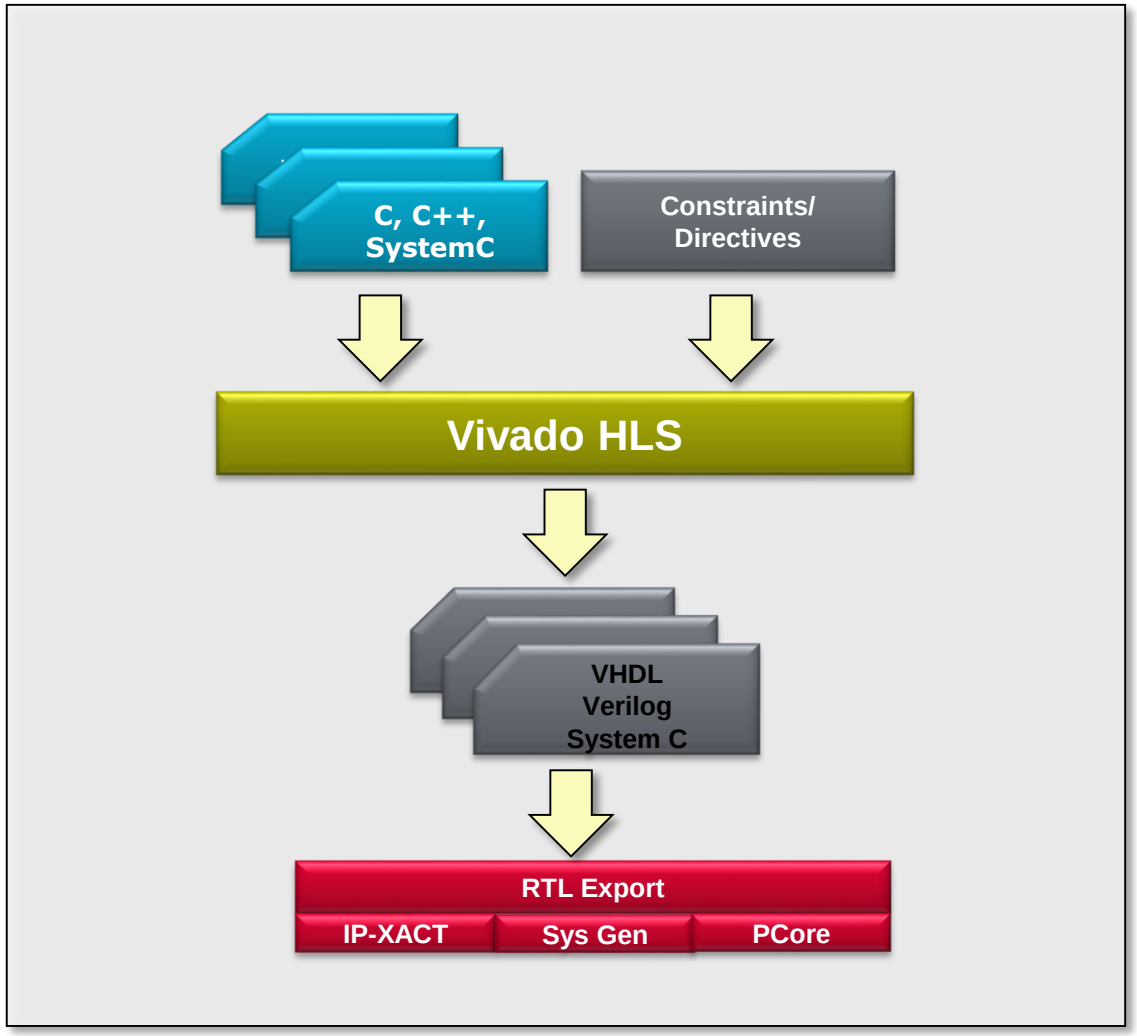
Bottom-up Agenda

- Vivado HLS: the core
- System integration level
  - SDSoC: replace SoC
  - SDAccel: replace CPU/GPU
  - SDNet: replace NPU/ASSP
- Xilinx University Program

Vivado HLS: High-Level Synthesis

High-Level Synthesis: HLS

- **High-Level Synthesis**
  - Creates an RTL implementation from C level source 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



Design Exploration with Directives

One body of code:
Many hardware outcomes

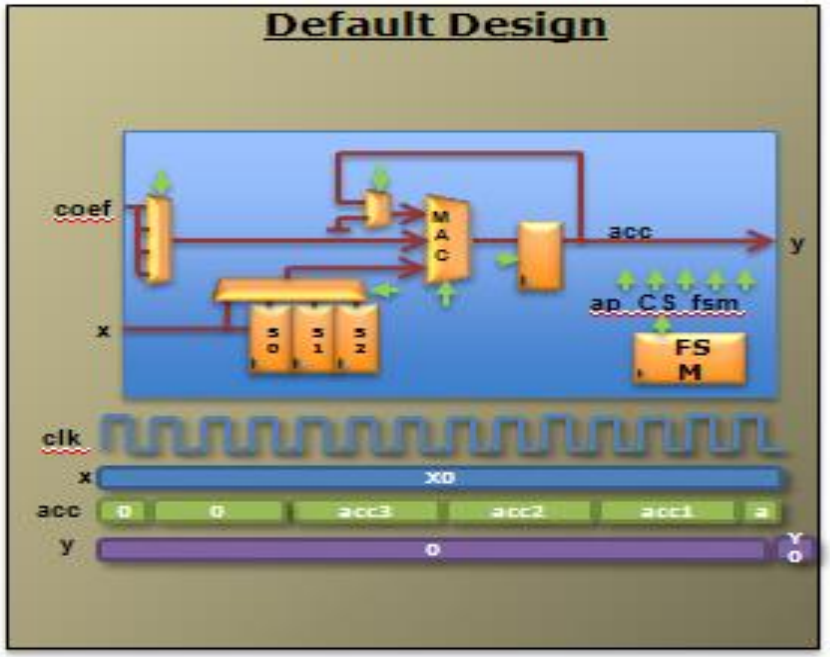
```
…
loop: for (i=3;i>=0;i--) {
  if (i==0) {
    acc+=x*c[0];
    shift_reg[0]=x;
  } else {
    shift_reg[i]=shift_reg[i-1];
    acc+=shift_reg[i]*c[i];
  }
}
….
```

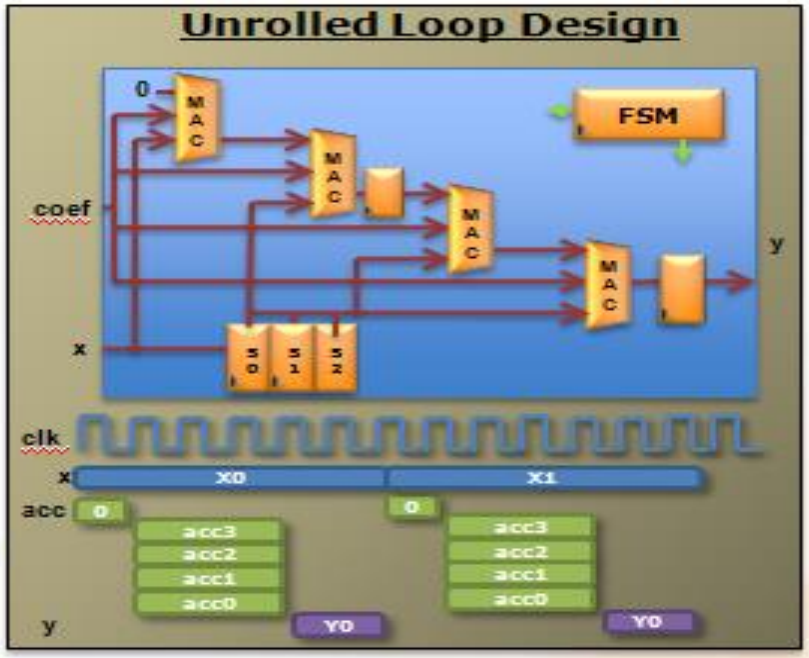
Before we get into details, let's look under the hood ….

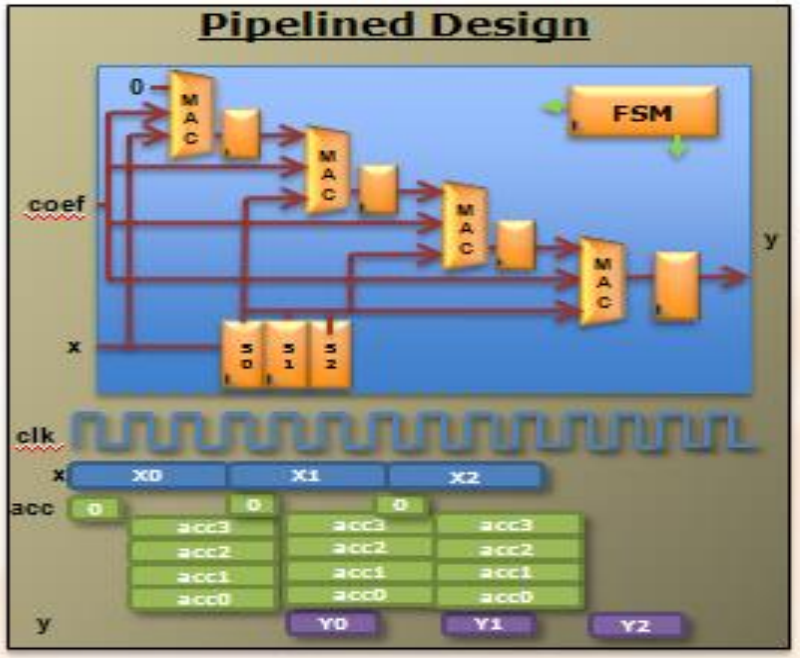
The same hardware is used for each iteration of the loop:
- **Small area**
- **Long latency**
- **Low throughput**

Different hardware is used for each iteration of the loop:
- **Higher area**
- **Short latency**
- **Better throughput**

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







HLS: 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;
}
```

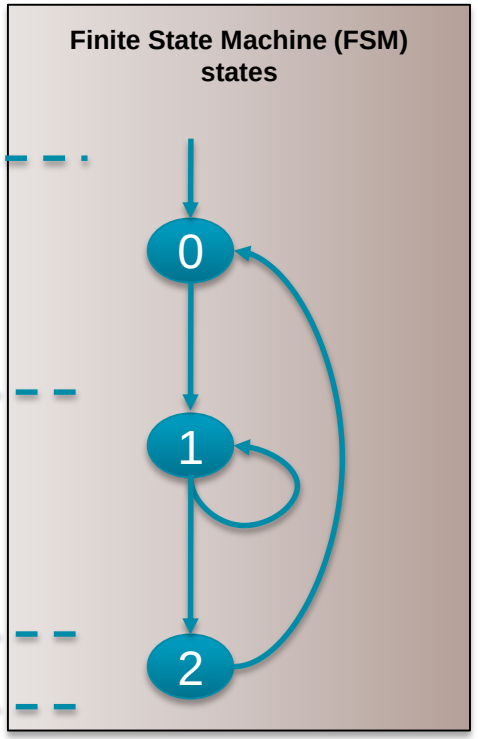
Function Start

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

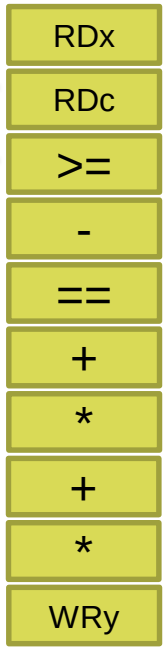
This behavior is extracted into a hardware state machine

HLS: Control & Datapath Extraction

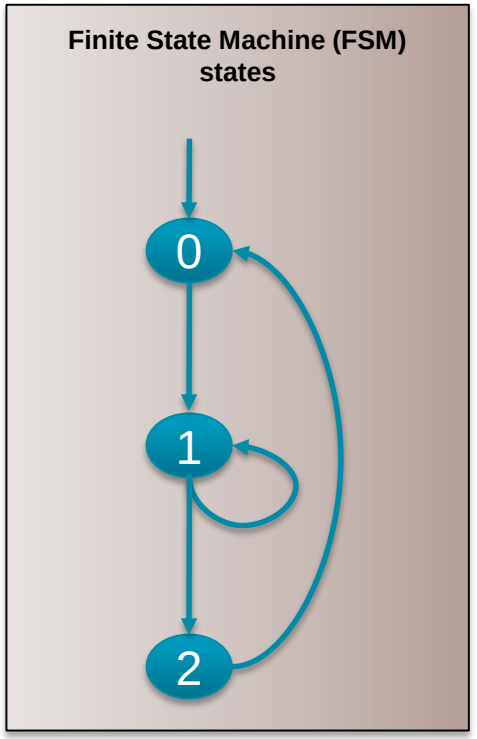
Code

```
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;
}
```

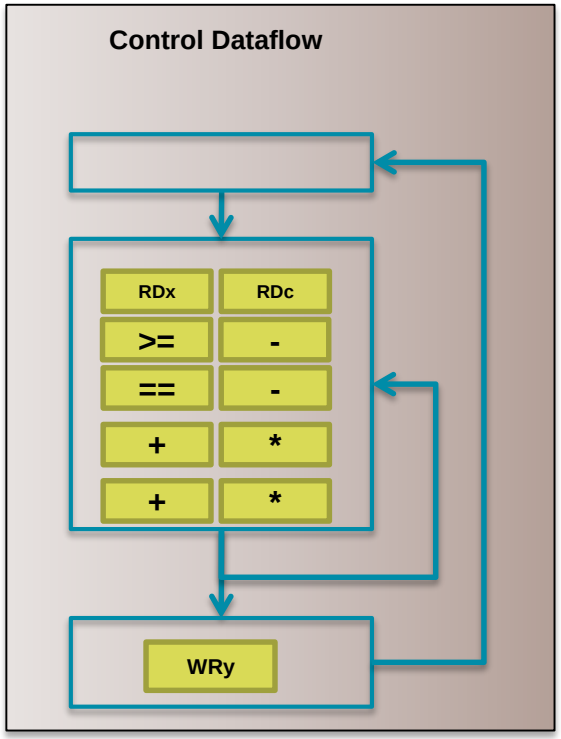
Operations



Control Behavior



Control & Datapath Behavior



From any C code example ..

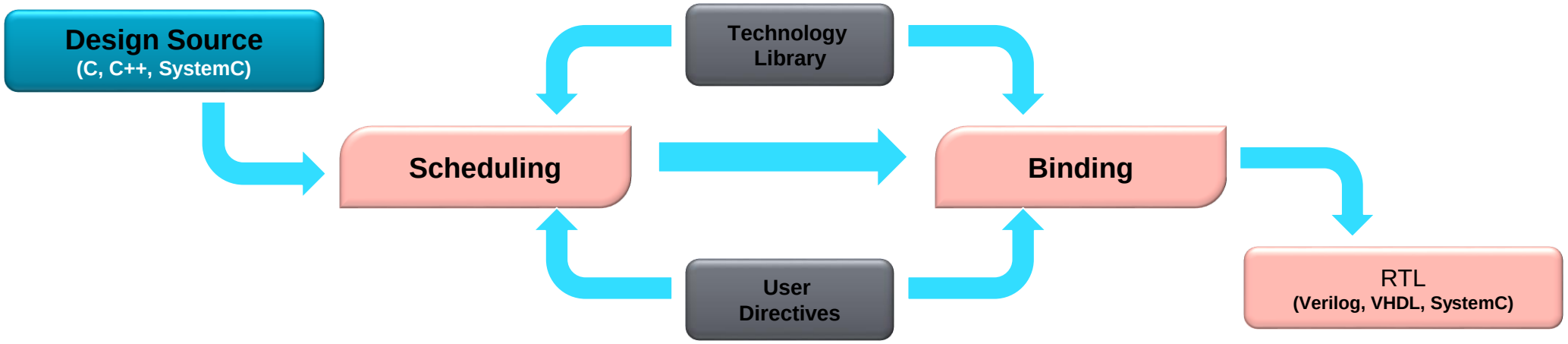
Operations are extracted…

The control is known

A unified control dataflow behavior is created.

High-Level Synthesis: Scheduling & Binding

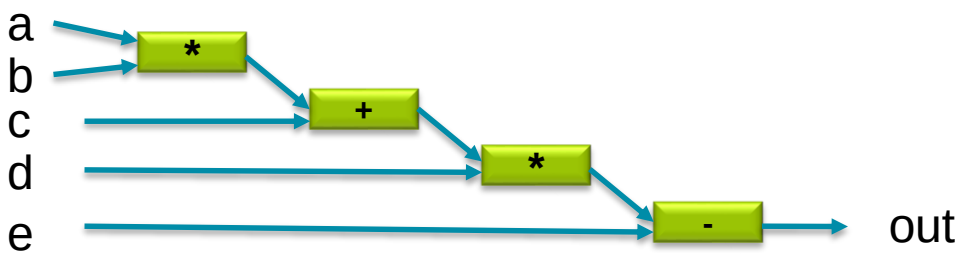
- **Scheduling and Binding are at the heart of HLS**
- **Scheduling determines in which clock cycle an operation will occur**
  - Takes into account the control, dataflow and user directives
  - The allocation of resources can be constrained
- **Binding determines which library cell is used for each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



Scheduling

- **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;
}
```



Schedule 1

- **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

Schedule 2

- **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

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

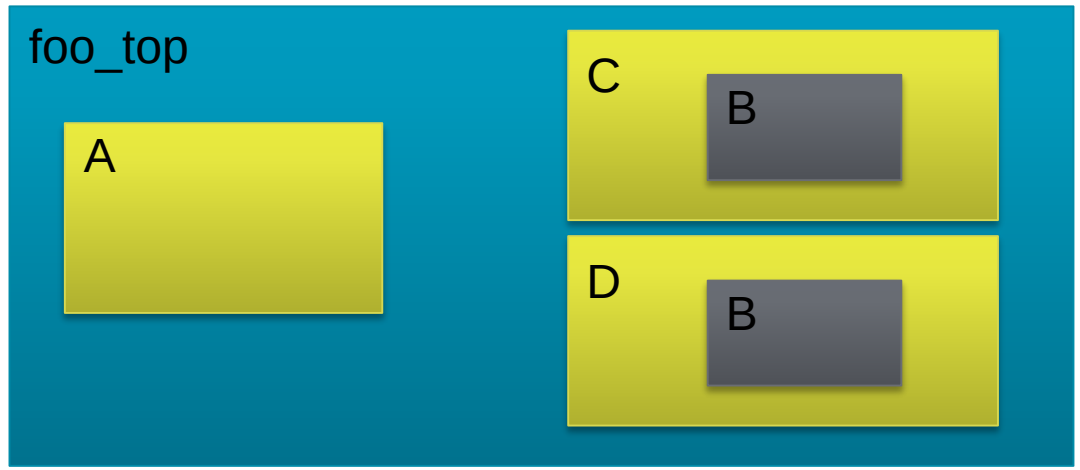
- **Each function is translated into an RTL block**
  - Verilog module, VHDL entity

Source Code

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


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

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

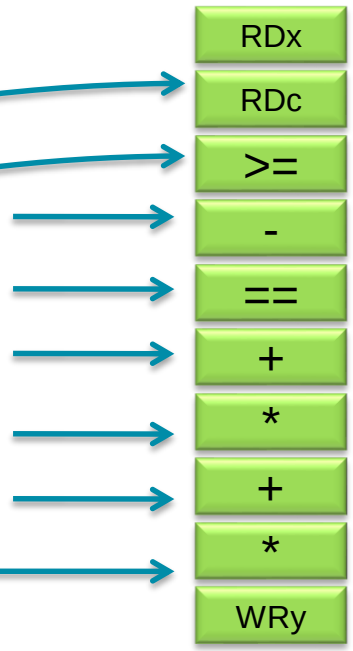
  - By default, each function is implemented using a common instance
  - Functions may be inlined to dissolve their hierarchy
    - Small functions may be automatically inlined

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;
}
```

Operations



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)
C++:	ap_(u)int types (1-1024) ap_fixed types
C++/SystemC:	sc_(u)int types (1-1024) sc_fixed types

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

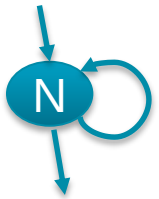
From any C code example ...

Operations are extracted…

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

Loops

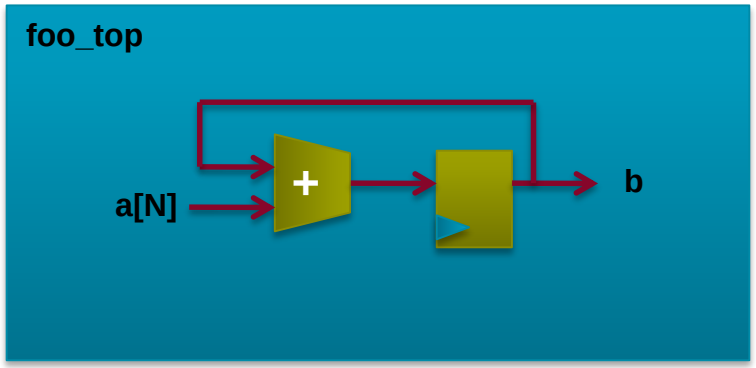
- **By default, loops are rolled**
  - Each C loop iteration ➔ Implemented in the same state
  - Each C loop iteration ➔ Implemented with same resources




```
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)

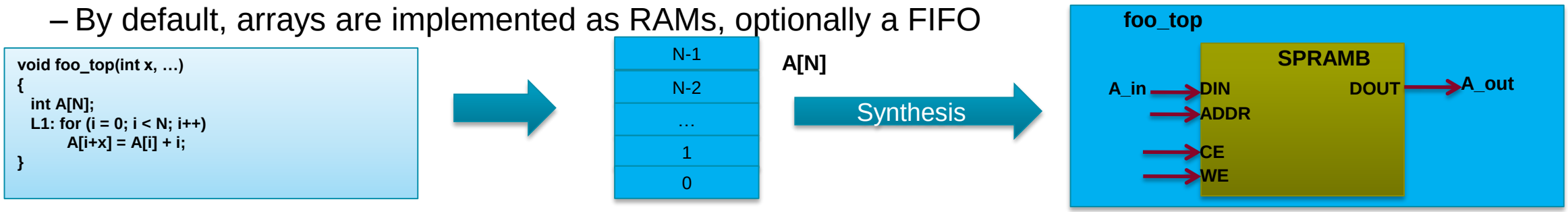
Synthesis



  - 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
    - Let's look at an example ….

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



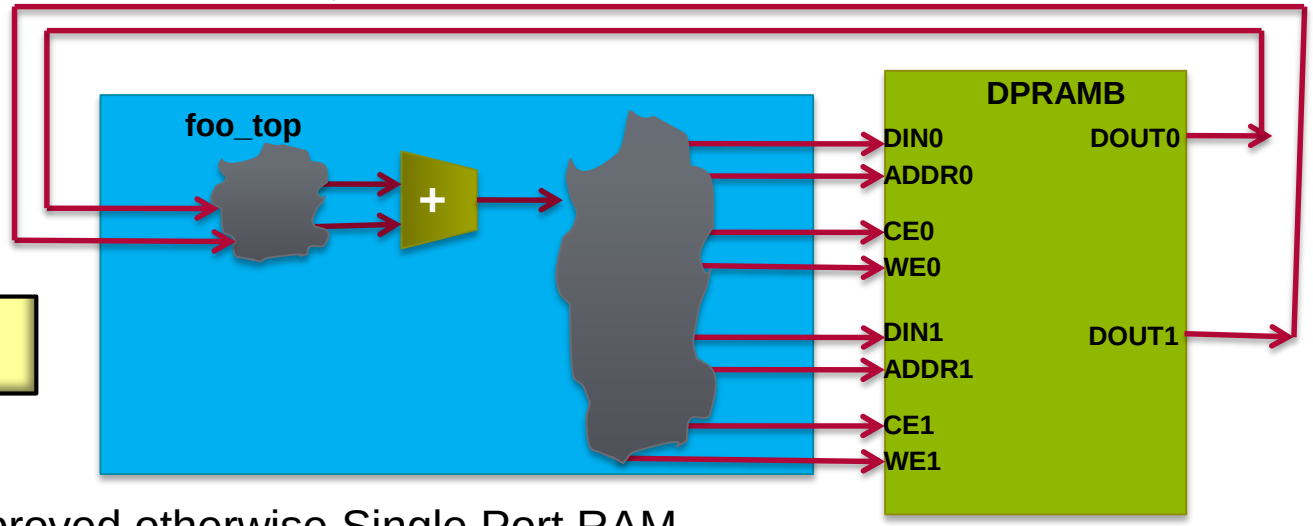
- **The array can be targeted to any memory resource in the library**
  - The ports (Address, CE active high, etc.) and sequential operation (clocks from address to data out) are defined by the library model
  - All RAMs are listed in the Vivado HLS Library Guide
- **Arrays can be merged with other arrays 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

Top-Level IO Ports

- **Top-level function arguments**
  - All top-level function arguments have a default hardware port type
- **When the array is an argument of the top-level function**
  - The array/RAM is “off-chip”
  - The type of memory resource determines the top-level IO ports
  - Arrays on the interface can be mapped & partitioned
    - E.g. partitioned into separate ports for each element in the array

```
void foo_top( int A[3*N] , int x)
{
   L1: for (i = 0; i < N; i++)
         A[i+x] = A[i] + i;
}
```

Synthesis



Number of ports defined by the RAM resource

- **Default RAM resource**
  - Dual port RAM if performance can be improved otherwise Single Port RAM

Comprehensive C Support

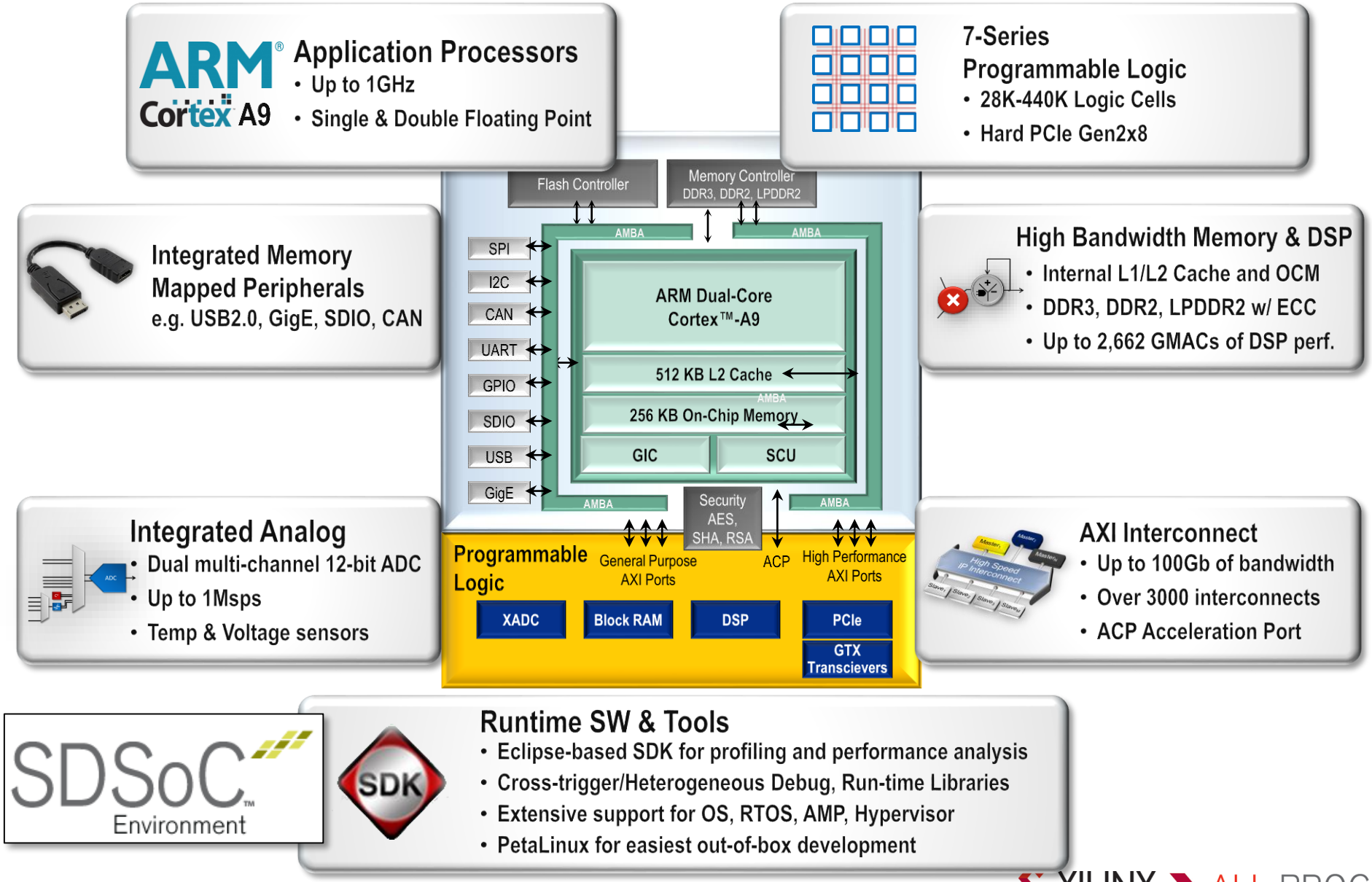
- **A Complete C Validation & Verification Environment**
  - Vivado HLS supports complete bit-accurate validation of the C model
  - Vivado HLS provides a productive C-RTL co-simulation verification solution
- **Vivado HLS supports C, C++ and SystemC**
  - Functions can be written in any version of C
  - Wide support for coding constructs in all three variants of C
  - Require to be statically defined at **compile time**
- **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

Summary

- **In HLS**
  - C becomes RTL
  - Operations in the code map to hardware resources
  - Understand how constructs such as functions, loops and arrays are synthesized
- **HLS design involves**
  - Synthesize the initial design
  - Analyze to see what limits the performance
    - User directives to change the default behaviors
    - Remove bottlenecks
  - Analyze to see what limits the area
    - The types used define the size of operators
    - This can have an impact on what operations can fit in a clock cycle
- **Use directives to shape the initial design to meet performance**
  - Increase parallelism to improve performance
  - Refine bit sizes and sharing to reduce area

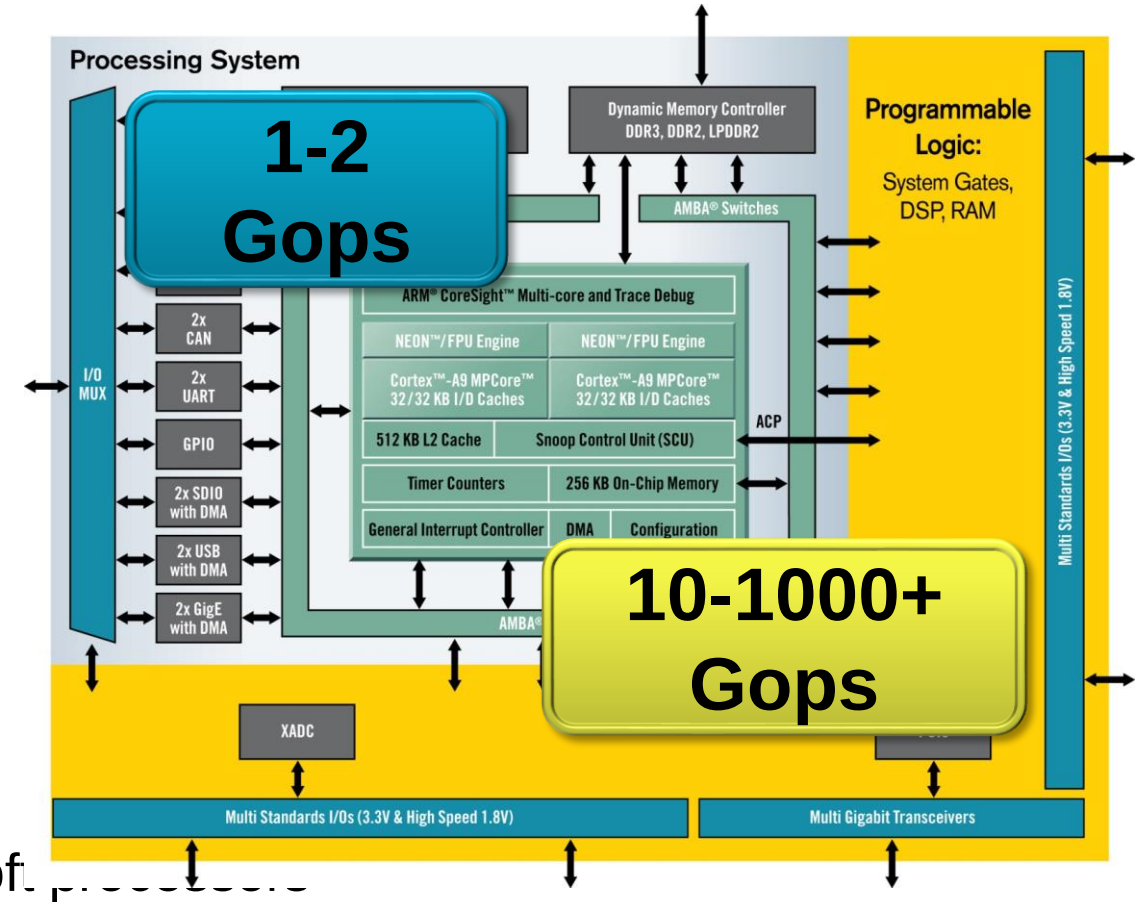
SDSoC

Zynq-7000 All Programmable SoC Highlights



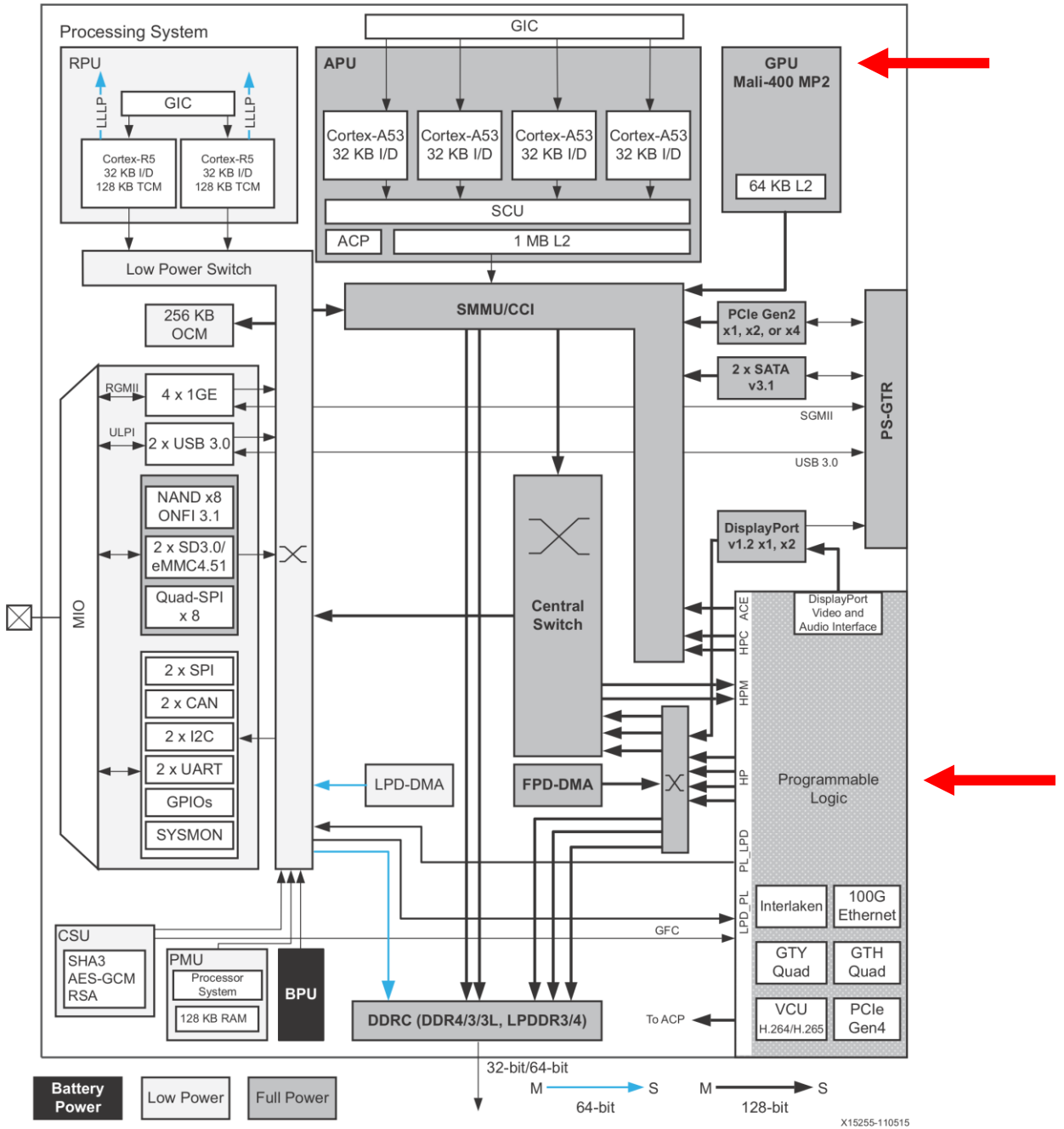
Advantage of FPGA Hardware Accelerators

- **Processing System (PS)**
  - Two ARM® Cortex™-A9 with NEON™ extensions
  - Floating Point support
  - Up to 1 GHz operation
  - L2 Cache – 512KB Unified
  - On-Chip Memory of 256KB
  - Integrated Memory Controllers
  - Run full Linux
- **Programmable Logic (PL)**
  - 28K-444K logic cells
  - High bandwidth AMBA interconnect
  - ACP port - cache coherency for additional soft processors

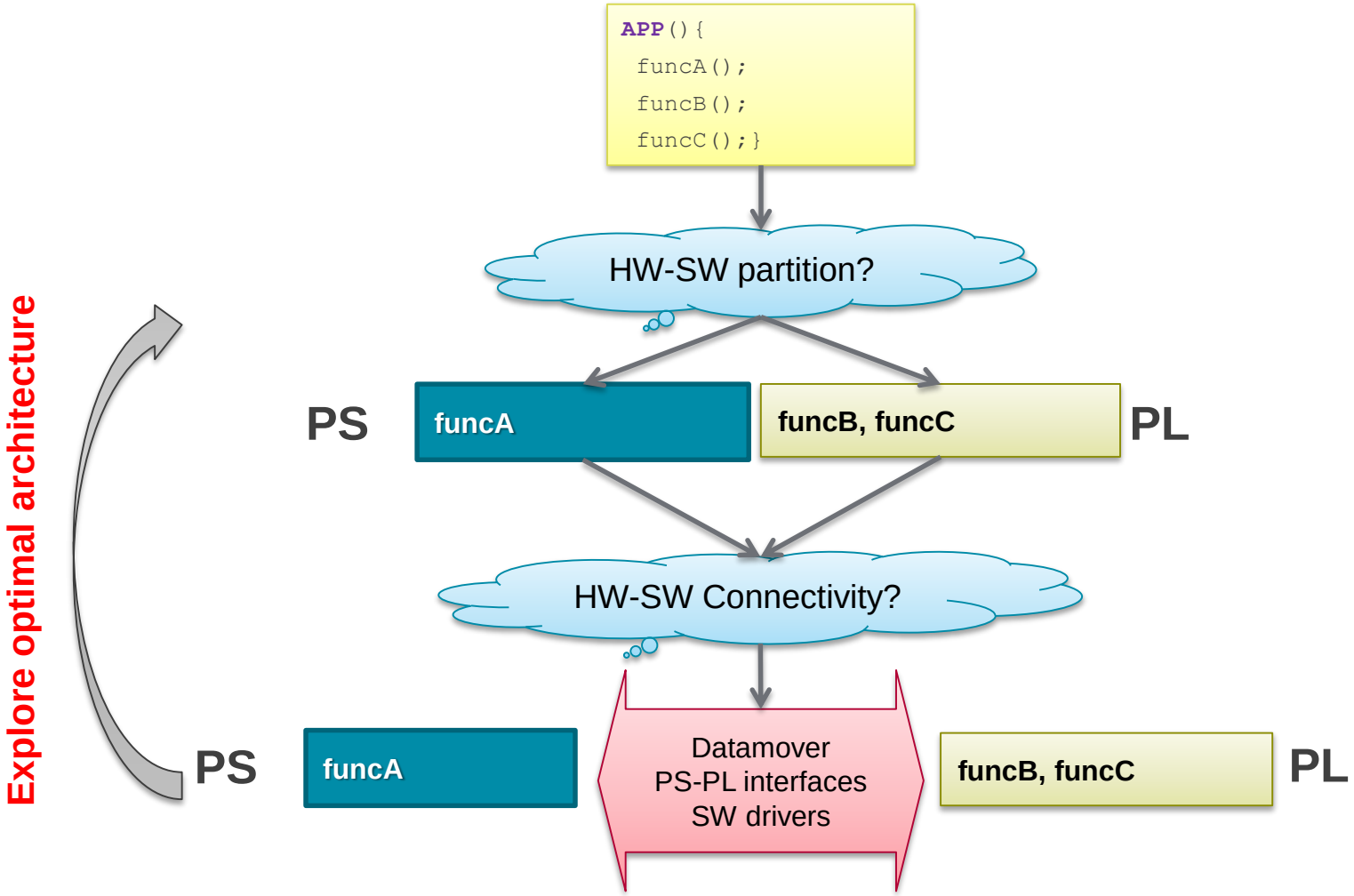


Programmable Logic Provides Superior Performance

Zynq UltraScale+ MPSoC



All Programmable SoCs Development Flow



SDSoC Development Environment

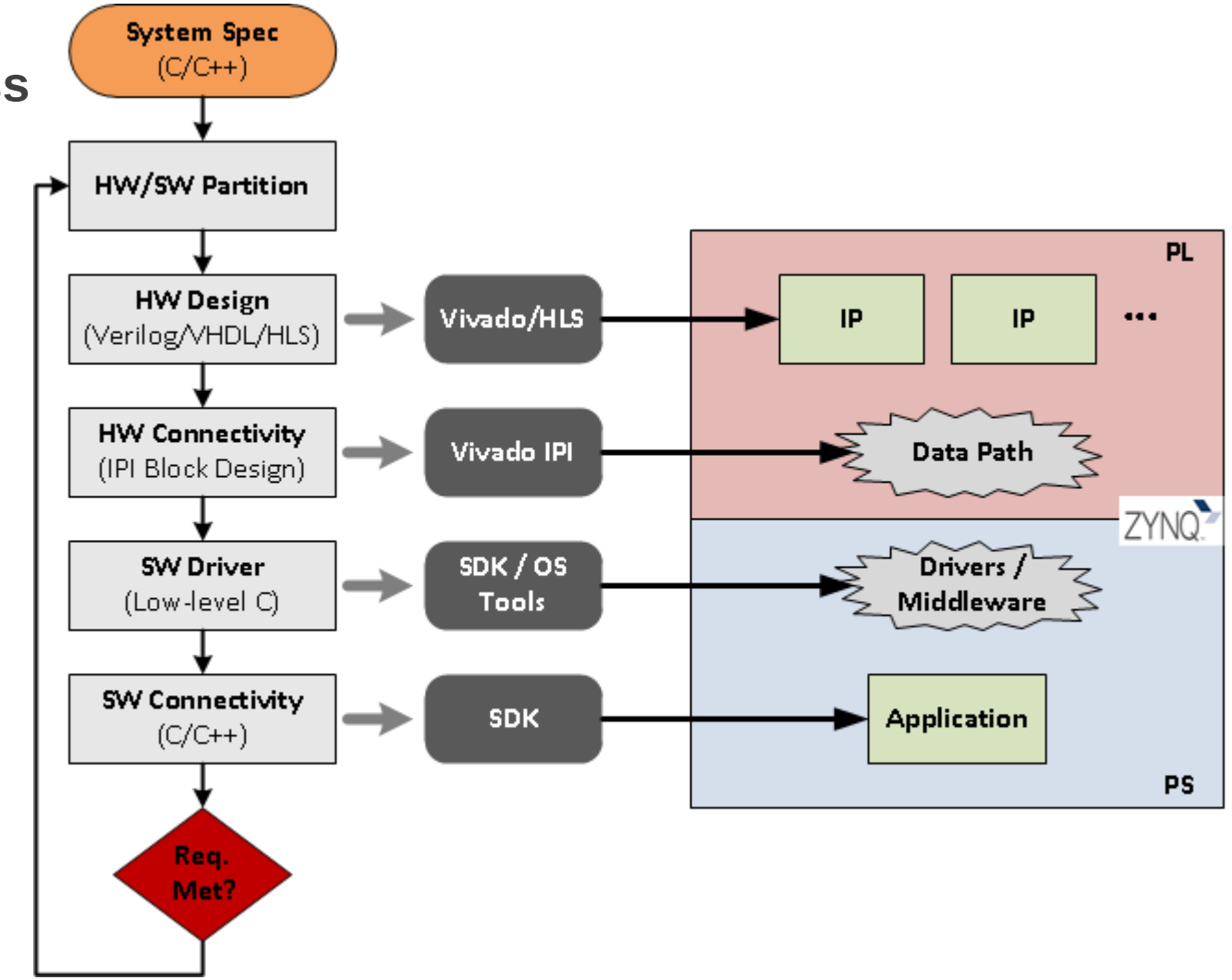
- **What is it?**
  - The SDSoC development environment provides a familiar embedded C/C++ application development experience including
    - An easy to use Eclipse IDE
    - A comprehensive design environment for heterogeneous Zynq SoC and MPSoC
    - C/C++ full-system optimizing compiler
    - Infrastructure to combine processing system, accelerators, data movers, signaling, and drivers
- **Enables function acceleration in Programmable Logic (PL) with a click of button using various technologies**
    - Vivado IPI
    - Vivado HLS
    - SDK
    - Profiler

Benefits of SDSoC Development Environment

- **Shorter development cycles**
  - Estimation shows improvement over software-only solution for system and accelerators
  - Iterative improvement can be made at early stages of development without the need to build hardware
- **Simplified interface and partitioning between hardware and software**
  - Software-based flow vs RTL
  - User interfaces with software-only tools; hardware management is abstracted
  - Removes much of the hardware/software distinction and throw-over-the-wall effect
- **Automated initial design**
  - Users can tweak code at macro- and micro-architectural levels
  - Designers still have manual control over the constructed Vivado HLS tool and Vivado Design Suite projects

Development Flow Without SDSoC

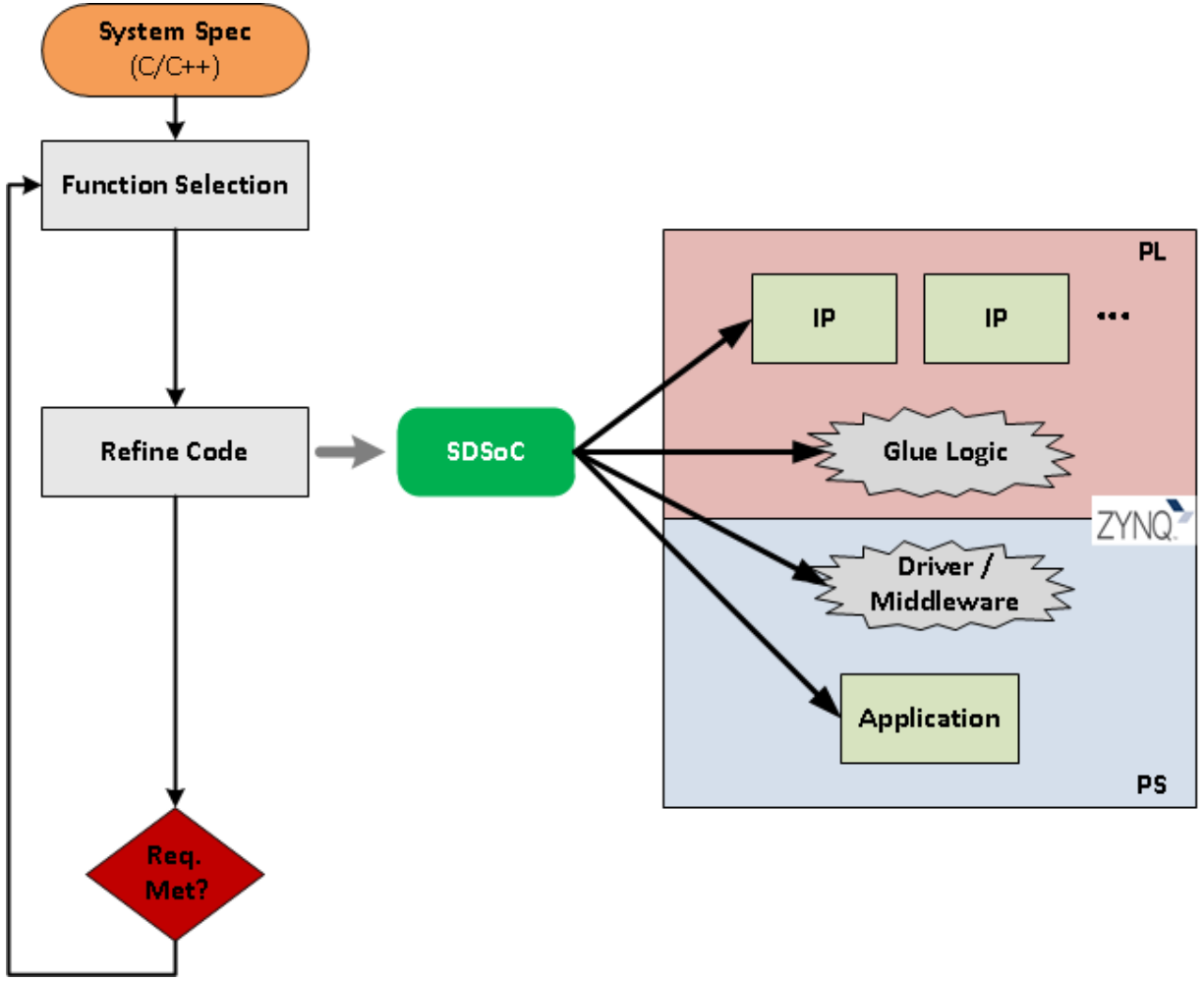
- **Overall process requires expertise at all steps in the development process**
  - Various hardware design entries
  - Hardware connectivity at system level
  - Driver development to drive custom hardware
  - Integrating with application and target OS



© Copyright 2015 Xilinx

SDSoC Development Environment

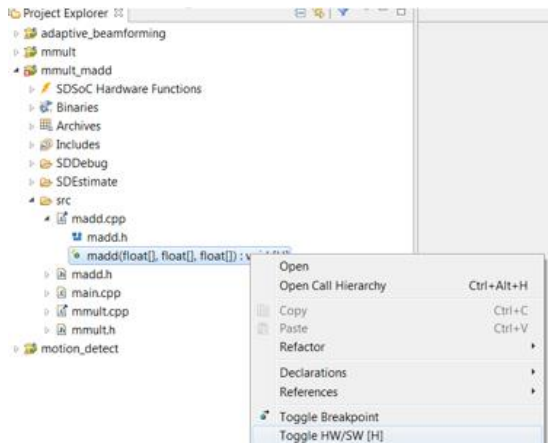
- **SDSoC development environment consolidates a multi-step/multi-tool process into a single tool and reduces hardware/software partitioning to simple function selection**
  - Code typically needs to be refined to achieve optimal results



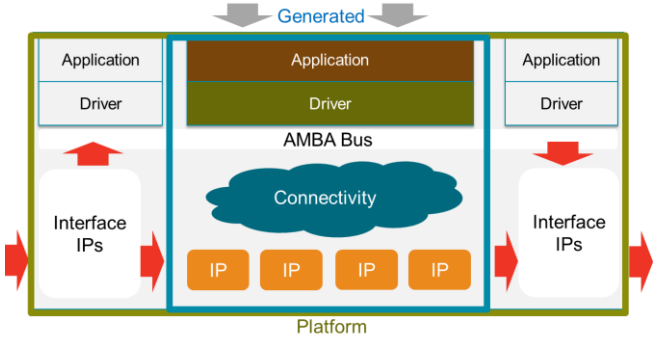
Software Development Environment

C/C++ Application

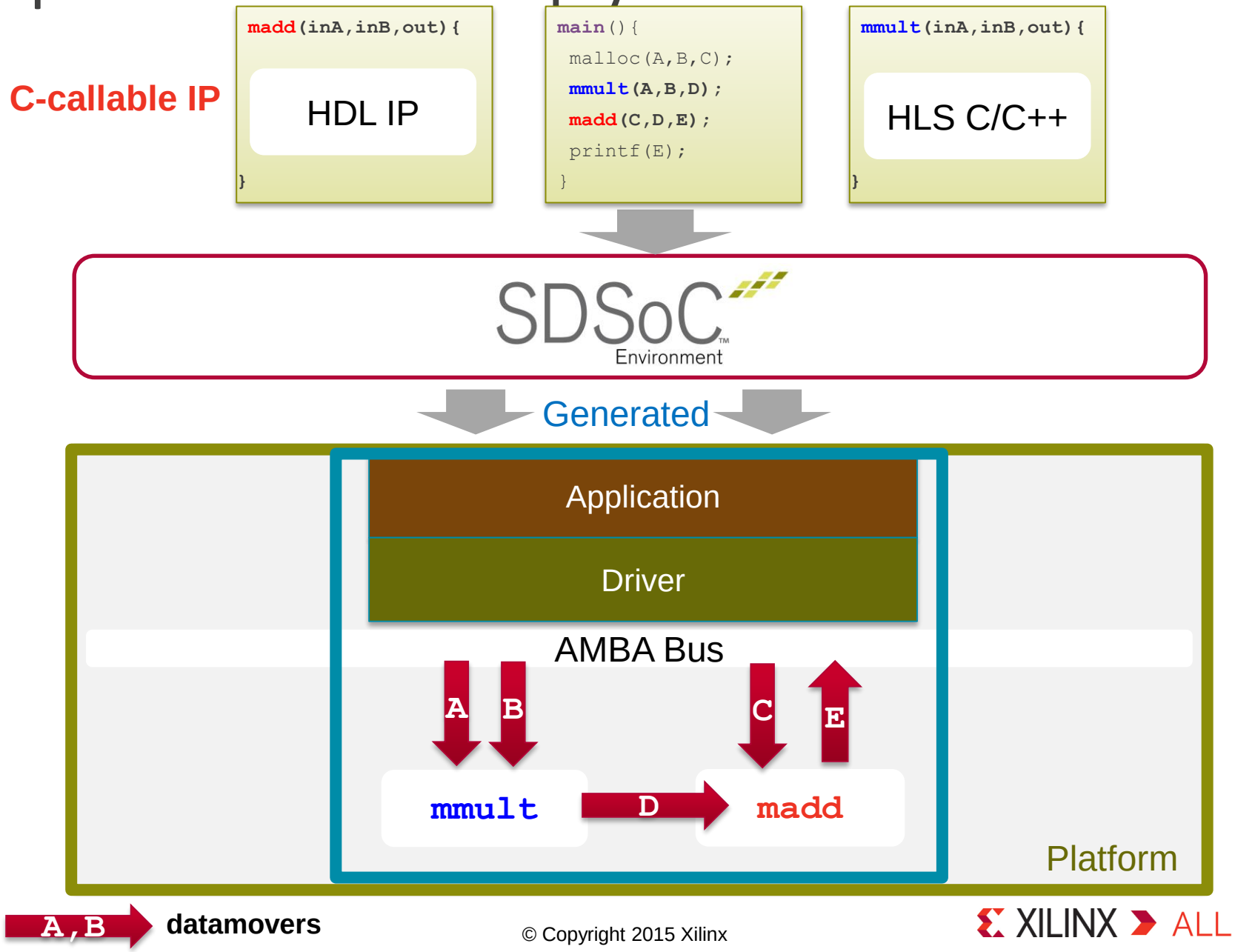
Estimator | **Compiler** | **Debugger** | **Profiler**



Leverages proven Xilinx tools in the backend



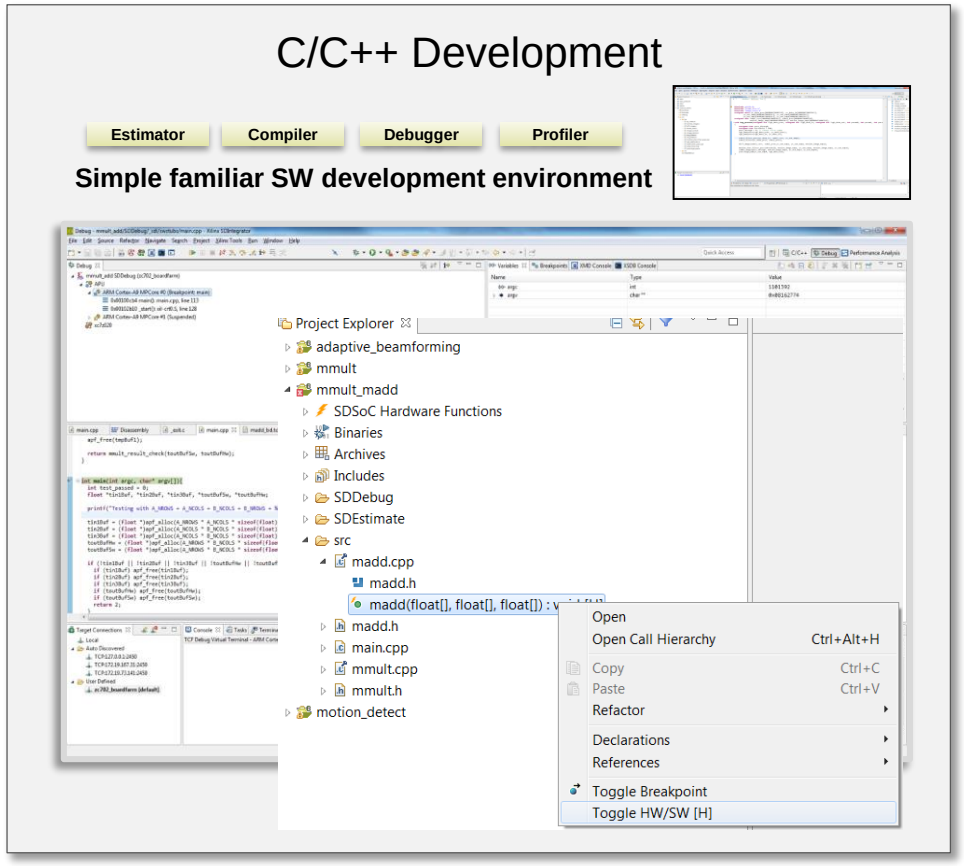
An Example – Matrix Multiply + Add



Supported Development Platforms

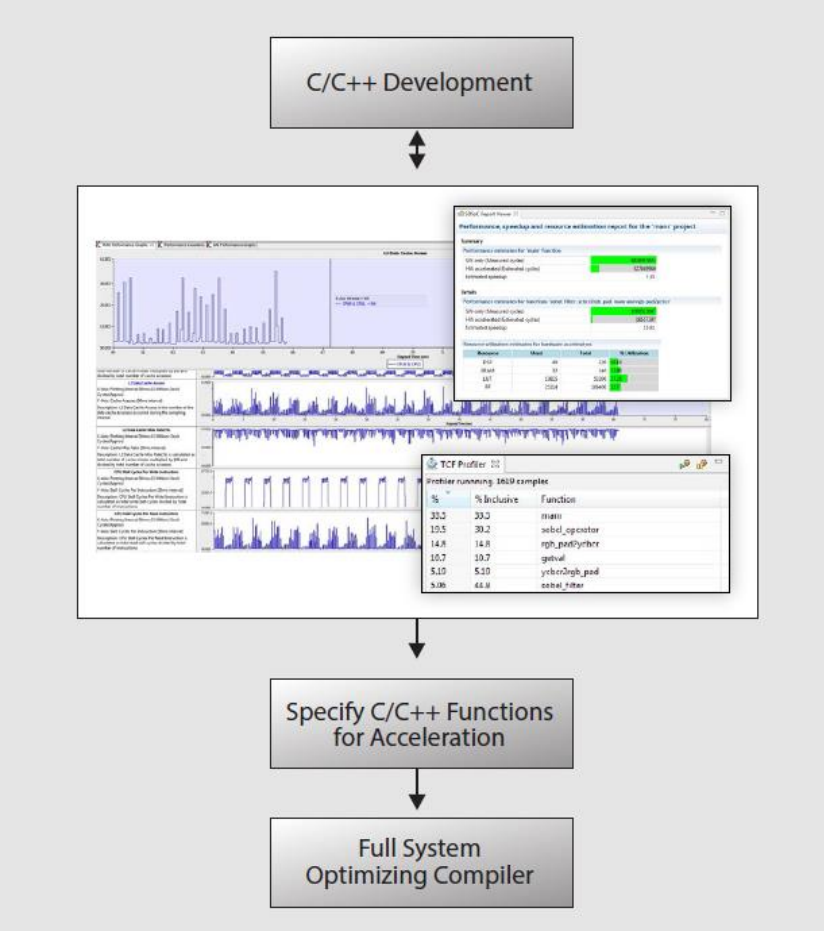
- **Start today with:**
  - Xilinx development platforms: ZC702 & ZC706
  - Alliance Member platforms: Zedboard, MicroZed, ZYBO, Zynq SDR, ZC702+HDMI IO, SVDK, …
  - Visit www.xilinx.com/sdsoc for the complete list
- Customer platform flow supported today (UG1146)

SDSoC's ASSP-like Programming Experience



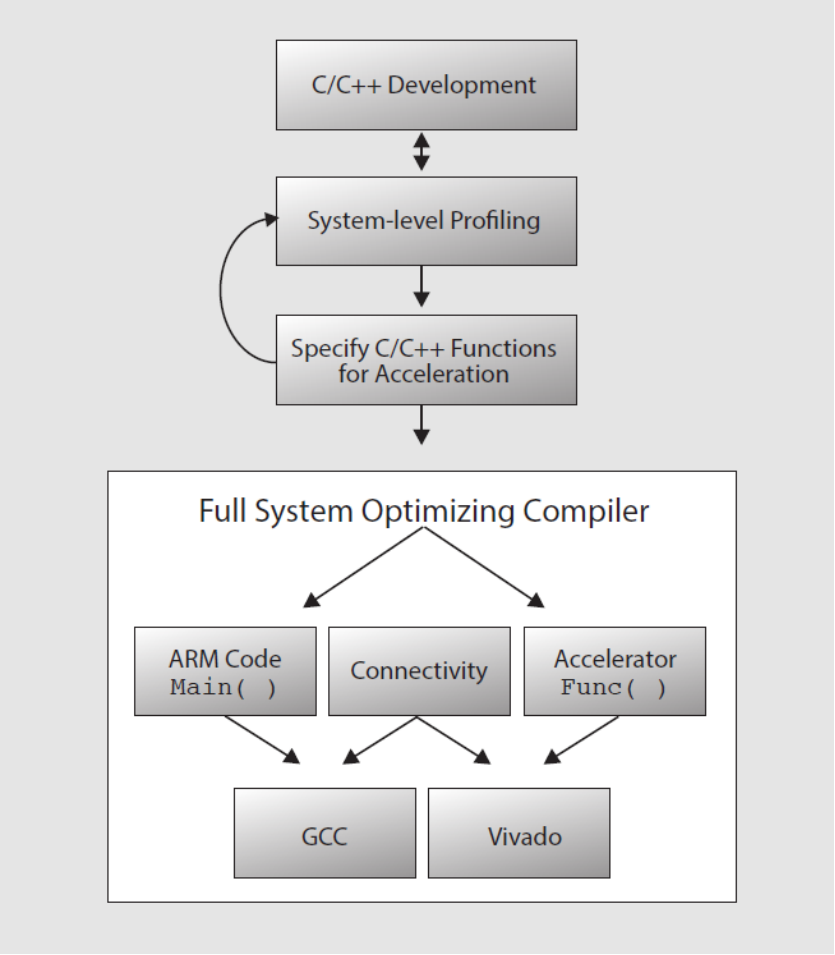
- **Easy to use Eclipse IDE**
- **One click to accelerate functions in Programmable Logic (PL)**
- **Optimized libraries**
  - Xilinx, ARM and Partners
  - DSP, Video, fixed point, linear algebra, BLAS, OpenCV
- **Support for Linux, FreeRTOS, bare metal**
  - Additional OS support in future releases

SDSoC's System Level Profiling



- **Rapid system performance estimation**
  - Full system estimation (programmable logic, data communication, processing system)
  - Reports SW/HW cycle level performance and hardware utilization
- **Automated performance measurement**
  - Runtime measurement by instrumentation of cache, memory, and bus utilization

Full System Optimizing Compiler



- **Rapid software configurable application acceleration using C/C++**
  - Automated function acceleration in programmable logic
  - Up to 100X increase in performance vs. software
  - System optimized for latency, bandwidth, and hardware utilization

Automated Generation of System Connectivity

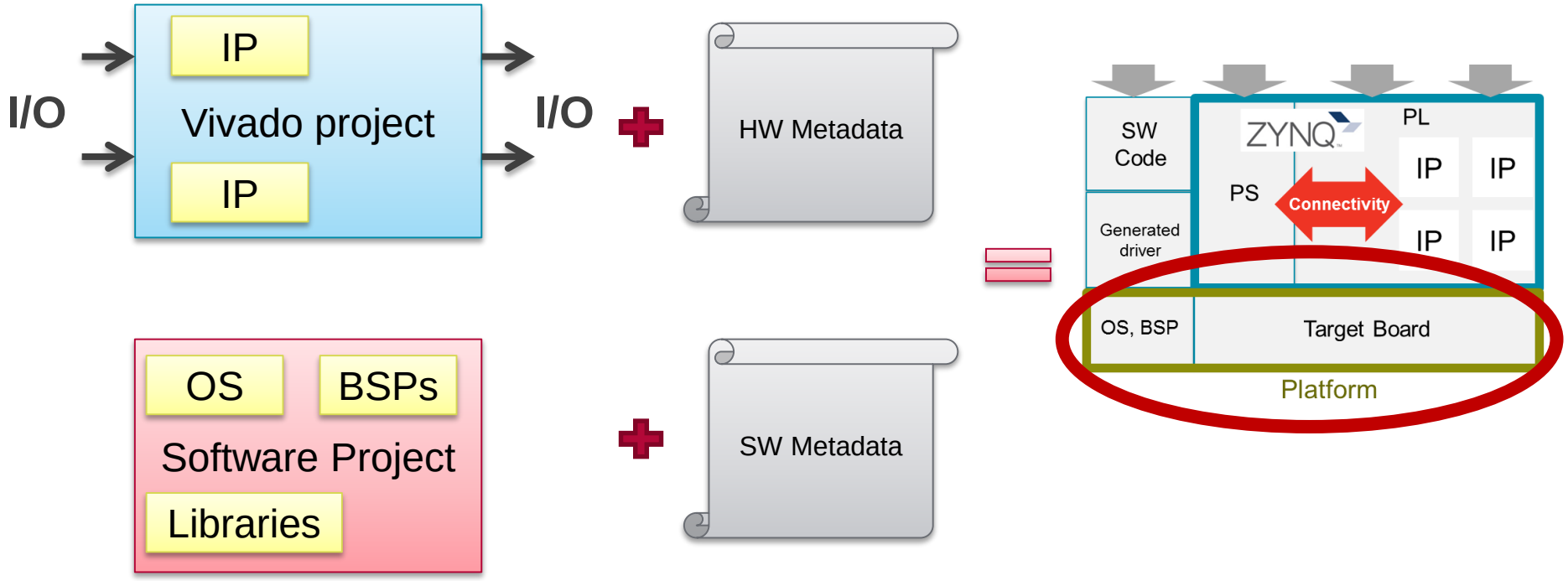
	PS-PL Interface				
Find the lowest latency DataMover		ACP	HP cache	HP non-cache	GP
	SW only	180,957	181,009	365,766	
	Simple DMA	27,023	38,705	26,797	
	SGDMA	30,804	43,225	30,818	
	Processor Direct	45,868	81,941	46,057	
	FIFO				427,878

32X32 floating point matrix multiply (latency in processor cycles)

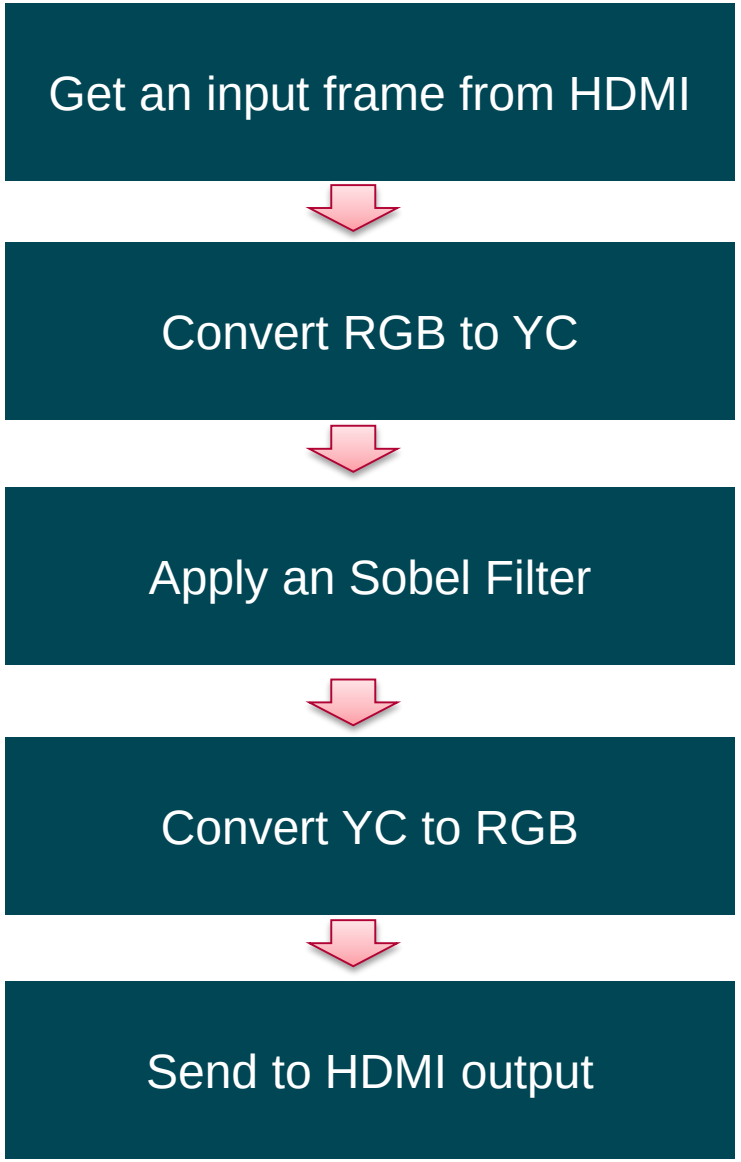
- **Explore system performance and power**
  - Rapidly configure, generate macro and micro architectures
  - Explore optimal interconnect and memory interfaces
  - Automatic insertion of AXI-Performance Monitor (APM) to obtain detailed cache/port/memory performance data
- **Shorten development time over traditional Hardware/Software flows**

Custom Platform Creation

- **Target customer's own board for production design with SDSoC**
- **Add metadata to existing Vivado project and existing Software projects (OS, BSPs and libraries)**



Optimization of Sobel filtering on video stream



Reference application

```
void img_process(unsigned int *rgb_data_in,
                 unsigned int *rgb_data_out,
                 unsigned short *yc_data_in,
                 unsigned short *yc_sobel_out)
{

    rgb_pad2ycbcr(rgb_data_in, yc_data_in);
    sobel_filter(yc_data_in, yc_sobel_out);
    ycbcr2rgb_pad(yc_sobel_out, rgb_data_out);
}
```

Naïve Sobel Filter Implementation

```
void sobel_filter(unsigned short *yc_in,unsigned short *yc_out, short *x_op, short *y_op)
{
  int row, col;
  for(row = 0; row < NUMROWS; row++){
    for(col = 0; col < NUMCOLS; col++){
      unsigned short input_data, unsigned char edge;
      if((col < NUMCOLS) & (row < NUMROWS))
        input_data = yc_in[row*NUMCOLS+col];
      if( isEdge(row, col)) edge=0;
      else{
        short x_weight = 0, y_weight = 0;
        for(char I = 0; i < 3; i++){
          for(char j = 0; j < 3; j++){
            unsigned short temp = (yc_in[index(row, col, I, j)]);
            x_weight += (temp * x_op[i][j]);
            y_weight += (temp * y_op[i][j]);
        } }
        edge = ABS(x_weight) + ABS(y_weight);
      }
      yc_out[index(row, col)] = edge;
    }
  }
}
```

Iterate over an input video image

Applying 3x3 sobel filter

Writing to output image

Video 1: Run the Naïve Sobel on a Zynq board

- **0.1 FPS @ 1080p**

Problem 1: Non-Sequential Overlapped Memory Access

```
void sobel_filter(unsigned short *yc_in,unsigned short *yc_out, short *x_op, short *y_op)
{
  int row, col;
  for(row = 0; row < NUMROWS; row++){
    for(col = 0; col < NUMCOLS; col++){
      unsigned short input_data, unsigned char edge;
      if((col < NUMCOLS) & (row < NUMROWS))
        input_data = yc_in[row*NUMCOLS+col];
      if( isEdge(row, col)) edge=0;
      else{
        short x_weight = 0, y_weight = 0;
        for(char I = 0; i < 3; i++){
          for(char j = 0; j < 3; j++){
            unsigned short temp = (yc_in[index(row, col, I, j)]);
            x_weight += (temp * x_op[i][j]);
            y_weight += (temp * y_op[i][j]);
        } }
        edge = ABS(x_weight) + ABS(y_weight);
      }
      yc_out[index(row, col)] = edge;
    }
  }
}
```

9 memory access per iteration

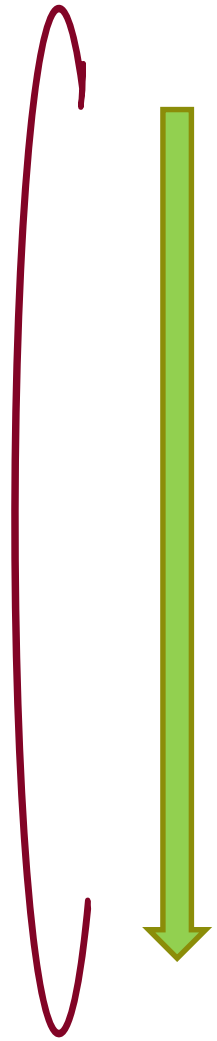
Solution 1: Use ap_linebuffer and ap_window Classes

```
void sobel_filter(unsigned short *yc_in,unsigned short *yc_out, short *x_op, short *y_op)
{
  int row, col;
  ap_linebuffer<unsigned char, 3, NUMCOLS> buff_A;
  ap_window<unsigned char,3,3> buff_C;
  for(row = 0; row < NUMROWS; row++){
    for(col = 0; col < NUMCOLS; col++){
      unsigned short input_data, unsigned char edge;
      update_linebuffer(buff_A, yc_in[index(row,col)]);
      update_window(buff_C);
      if((col < NUMCOLS) & (row < NUMROWS))
        input_data = yc_in[row*NUMCOLS+col];
      if( isEdge(row, col)) edge=0;
      else{
        short x_weight = 0, y_weight = 0;
        for(char I = 0; i < 3; i++){
          for(char j = 0; j < 3; j++){
            unsigned short temp = (buff_C[index(row, col, I, j)]);
            x_weight += (temp * x_op[i][j]);
            y_weight += (temp * y_op[i][j]);
        } }
        edge = ABS(x_weight) + ABS(y_weight);
      }
      yc_out[index(row, col)] = edge;
    }
  }
}
```

Video 2: Run the Solution 1 on a Zynq board

- **1 FPS @ 1080p**
- **10x speedup**

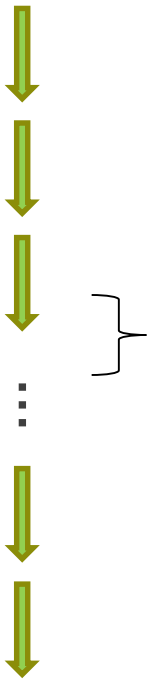
Problem 2: Loop Iterations Are Sequential

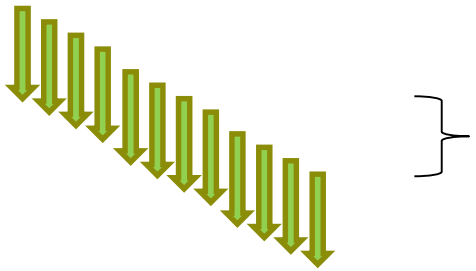


```
void sobel_filter(unsigned short *yc_in,unsigned short *yc_out, short *x_op, short *y_op)
{
  int row, col;
  for(row = 0; row < NUMROWS; row++){
    for(col = 0; col < NUMCOLS; col++){
      unsigned short input_data, unsigned char edge;
      if((col < NUMCOLS) & (row < NUMROWS))
        input_data = yc_in[row*NUMCOLS+col];
      if( isEdge(row, col)) edge=0;
      else{
        short x_weight = 0, y_weight = 0;
        for(char I = 0; i < 3; i++){
          for(char j = 0; j < 3; j++){
            unsigned short temp = (yc_in[index(row, col, I, j)]);
            x_weight += (temp * x_op[i][j]);
            y_weight += (temp * y_op[i][j]);
        } }
        edge = ABS(x_weight) + ABS(y_weight);
      }
      yc_out[index(row, col)] = edge;
    }
  }
}
```

Executing sequentially

Solution 2: Pipeline Loop Iterations





60*1920*1080 =

124,416,000 cycles

60+(1920*1080) =

2,073,660 cycles (60x speedup)

Assuming 60 cycles / loop iteration

Solution 2: Add Pipeline #pragma

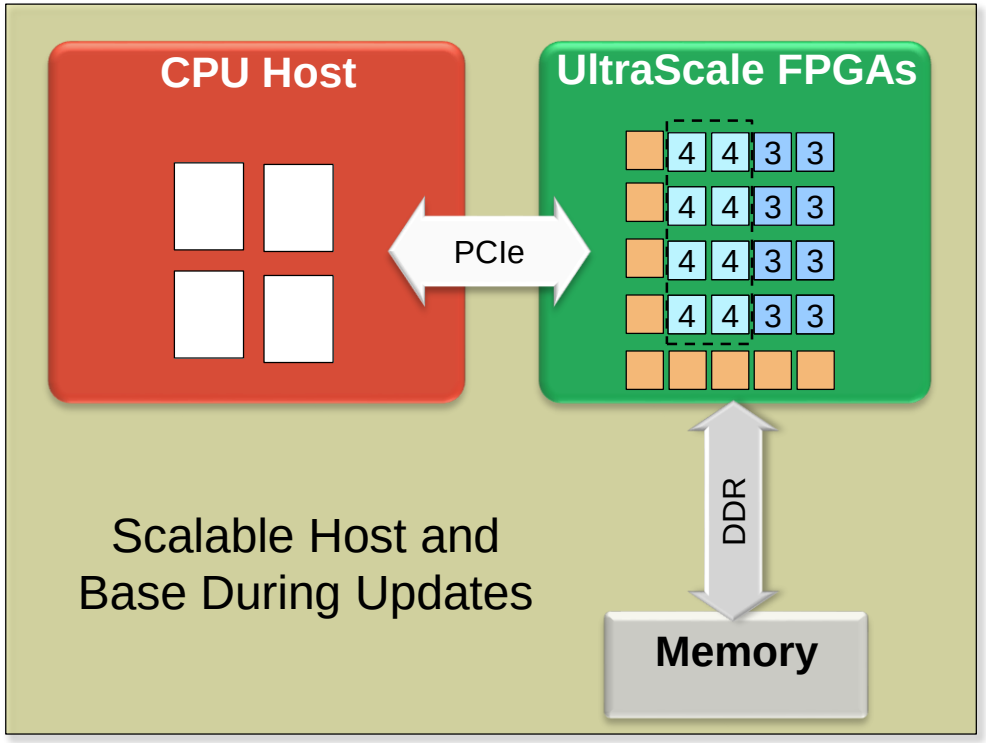
```
void sobel_filter(unsigned short *yc_in,unsigned short *yc_out, short *x_op, short *y_op)
{
  int row, col;
  ap_linebuffer<unsigned char, 3, NUMCOLS> buff_A;
  ap_window<unsigned char,3,3> buff_C;
  for(row = 0; row < NUMROWS; row++){
    for(col = 0; col < NUMCOLS; col++){
#pragma AP PIPELINE II = 1
      unsigned short input_data, unsigned char edge;
      update_linebuffer(buff_A, yc_in[index(row,col)]);
      update_window(buff_C);
      if((col < NUMCOLS) & (row < NUMROWS))
        input_data = yc_in[row*NUMCOLS+col];
      if( isEdge(row, col)) edge=0;
      else{
        short x_weight = 0, y_weight = 0;
        for(char I = 0; i < 3; i++){
          for(char j = 0; j < 3; j++){
            unsigned short temp = (buff_C[index(row, col, I, j)]);
            x_weight += (temp * x_op[i][j]);
            y_weight += (temp * y_op[i][j]);
        } }
        edge = ABS(x_weight) + ABS(y_weight);
      }
      yc_out[index(row, col)] = edge;
    }
  }
}
```

Video 3: Run the Solution 2 on a Zynq board

- **60 FPS @ 1080p**
- **60x speedup**

SDAccel

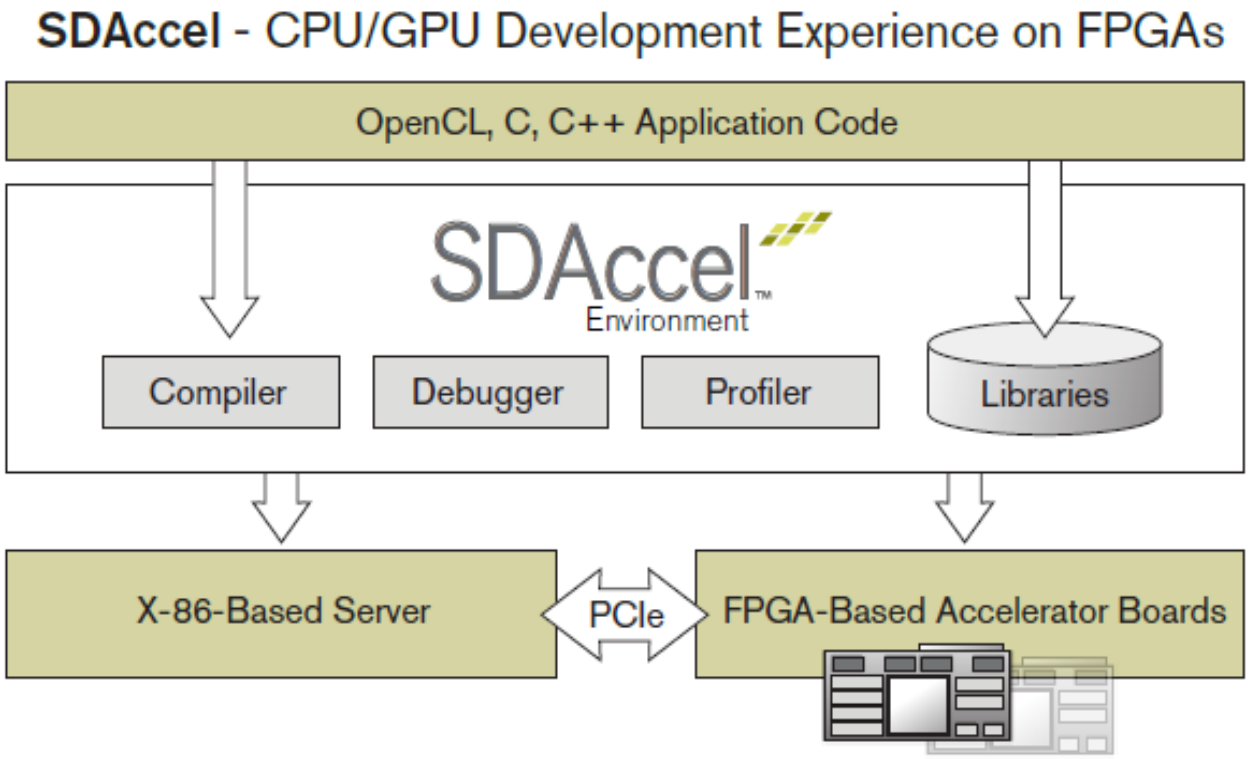
CPU/GPU-like Runtime Experience on FPGAs



CPU/GPU Runtime Experience

- On-demand loadable acceleration units
- Always on interfaces (Memory, Ethernet PCIe, Video)
- Optimize resources thru hardware reuse

SDAccel: Development Environment for C, C++ and OpenCL



Libraries	Availability
OpenCL built-ins	Included
Video, DSP, Linear Algebra	Included
OpenCV, BLAS	Provided by Auviz Systems

Readily Available Boards

CONVEY COMPUTER™ ALPHA DATA picocomputing

Typical deployment workflow

- **Optimize on x86 platform with emulator and auto generated cycle accurate models**
  - Identify application for acceleration
  - Program and optimize kernel on host
  - Compile and execute application for CPU
  - Estimate performance
  - Debug FPGA kernels with cycle accurate models on CPU
- **Deployment on FPGA**
  - Compile for FPGA (longest step)
  - Execute and validate performance on card

Implementation

- **Each OpenCL work-group mapped on 1 IP block**
- **Pipelining of work-items in a work-group possible with #pragma**
- **Support several kernels per program**
- **Generate FPGA container with bitstream + metadata**
- **Different compilation/execution modes**
  - CPU-only mode for host and kernels
  - RTL version of kernels for co-simulation
  - Real FPGA execution
- **Estimation of resource utilization by the tool**
- **OpenCL host API handles accelerator invocation**

Specify work-group size for better implementation

```
__kernel
__attribute__((reqd_work_group_size(4,4,1)))
void mmult32(__global int* A, __global int* B,
__global int* C) {
  // 2D Thread ID
  int i = get_local_id(0);
  int j = get_local_id(1);
  __local int Blocal[256];
  int result=0, k=0;
  B_local[i*16 + j] = B[i*16 + j];
  barrier(CLK_LOCAL_MEM_FENCE);
  for(k = 0; k < 16; k++)
    result += A[i*16 + k]*B_local[k*16 + j];
  C[i*16 + j] = result;
}
```

- **Is compiled into**

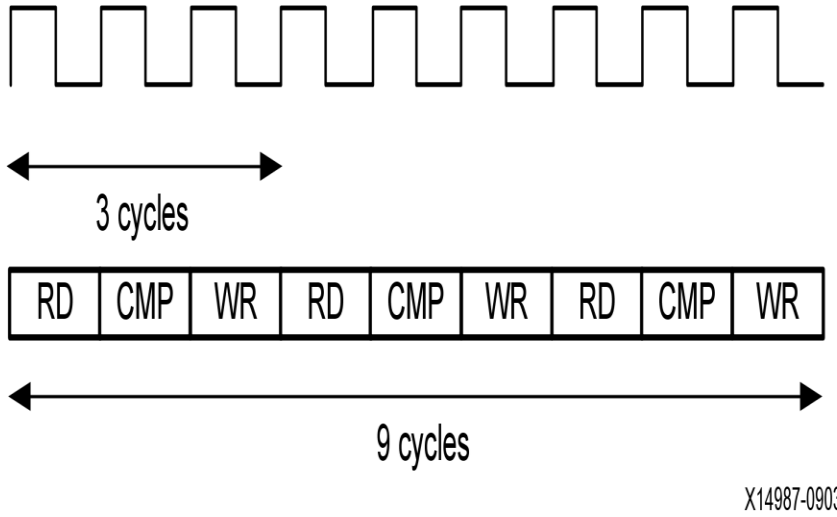
```
__kernel void mmult32(global int* A, global int* B,
global int* C) {
  localid_t id;
  int B_local[16*16];
  for(id[2] = 0; id[2] < 1; id[2]++)
    for(id[1] = 0; id[1] < 4; id[1]++)
      for(id[0] = 0; id[0] < 4; id[0]++) {
        ...
      }
  ...
}
```

Loop unrolling

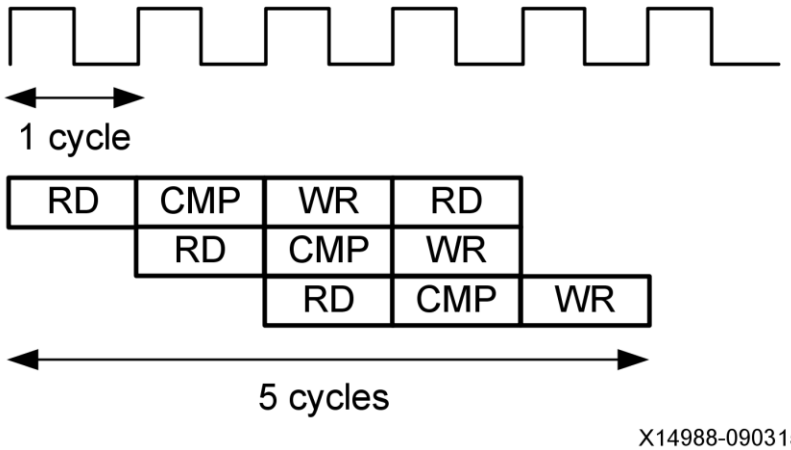
```
kernel void
vmult(local int* a, local int* b, local int* c) {
  int tid = get_global_id(0);
  __attribute__((opencl_unroll_hint(2)))
  for (int i = 0; i < 4; i++) {
    int idx = tid*4 + i;
    a[idx] = b[idx]*c[idx];
  }
}
```

Work-item pipelining

```
__attribute__ ((reqd_work_group_size(3,1,1)))
kernel void foo(...) {
  int tid = get_global_id(0);
  op_Read(tid);
  op_Compute(tid);
  op_Write(tid);
}
```




```
__attribute__ ((reqd_work_group_size(3,1,1)))
kernel void foo(...) {
  __attribute__((xcl_pipeline_workitems)) {
    int tid = get_global_id(0);
    op_Read(tid);
    op_Compute(tid);
    op_Write(tid);
  }
}
```



Partitioning memories inside of compute units

- **Useful to avoid access conflict**
- **__local int buffer[16] __attribute__((xcl_array_partition(cyclic,4,1)));**
  - Cyclic partition on 4 memories along dimension 1
- **___local int buffer[16] __attribute__((xcl_array_partition(block,4,1)));**
  - Bloc partition on 4 memories along dimension 1
- **__local int buffer[16] __attribute__((xcl_array_partition(complete,1)));**
  - Complete partitioning along dimension 1: 1 register/element ☺

Using HLS C/C++ as OpenCL kernel

- Need to use parameter interface compatible with SDAccel OpenCL

```
void matrix_multiplication(int *a, int *b, int *output) {
#pragma HLS INTERFACE m_axi port=a offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=b offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=a bundle=control
#pragma HLS INTERFACE s_axilite port=b bundle=control
#pragma HLS INTERFACE s_axilite port=output bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control
  // Matrices of size 16*16
  const int rank = 16;
  int sum = 0;
  // Cache the external matrices
  int bufa[rank*rank];
  int bufb[rank*rank];
  int bufc[rank*rank];
  memcpy(bufa, a, sizeof(bufa));
  memcpy(bufb, b, sizeof(bufb));
  for (unsigned int c = 0; c < rank; c++) {
    for (unsigned int r = 0; r < rank; r++) {
      sum = 0;
      for (int index = 0; index < rank; index++) {
#pragma HLS pipeline
        int aIndex = r*rank + index;
        int bIndex = index*rank + c;
        sum += bufa[aIndex]*bufb[bIndex];
      }
      bufc[r*rank + c] = sum;
    }
  }
  memcpy(output, bufc, sizeof(bufc));
  return;
}
```

SDNet

Introducing SDNet

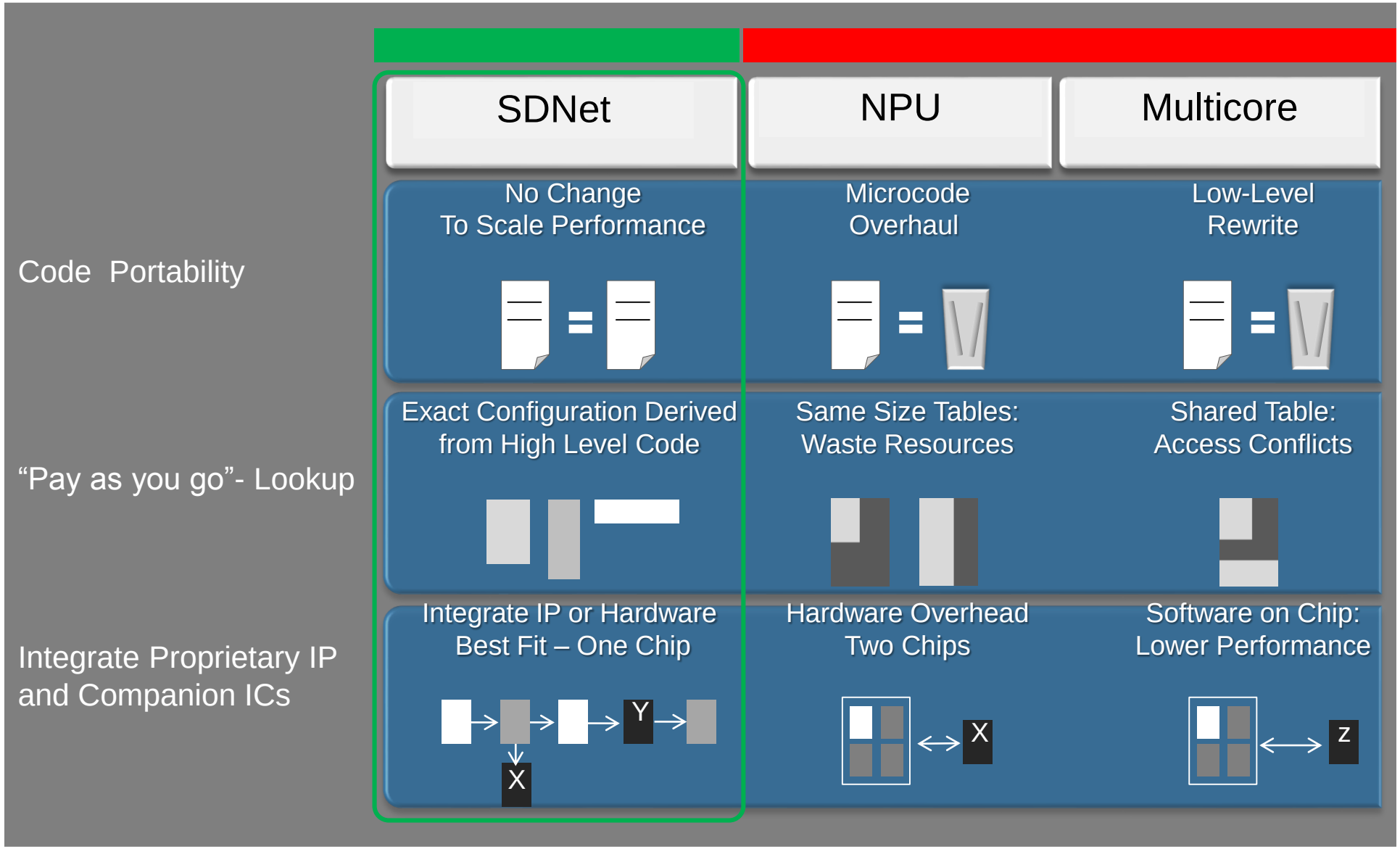
Technology Innovation with 25+ patents

- High Level Specification-to-Silicon
- FPGA Designers
- Systems/Software Engineers

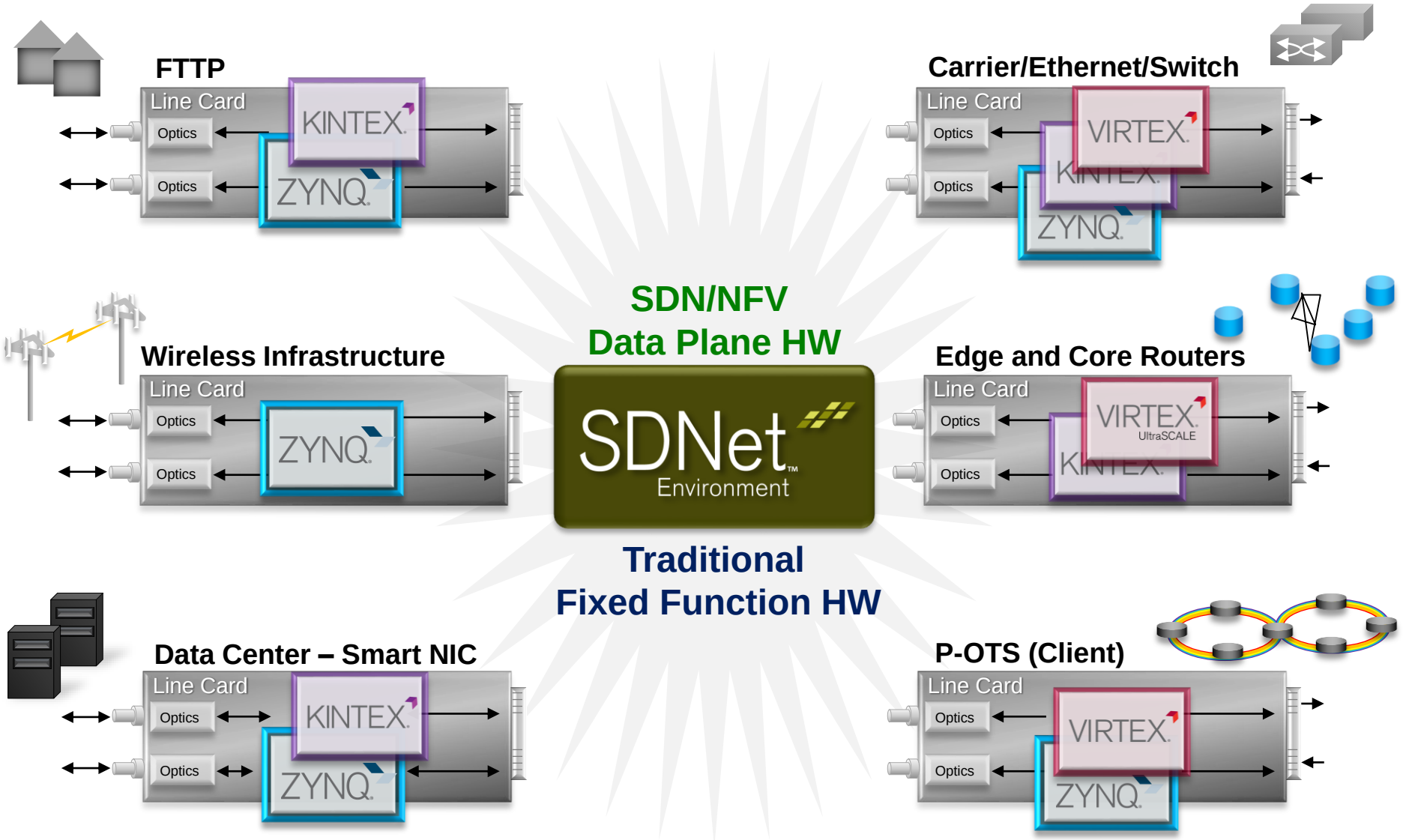
Novel approach to Programmable Data Plane Architecture

- High level specification enabling auto-generation of Data Plane COREs for FPGAs
- 10x productivity increase over SW design on Multicores and NPUs
- First-in-class Data Plane architecture able to collaborate with the Control Plane

SDNet: Data Plane Implementation

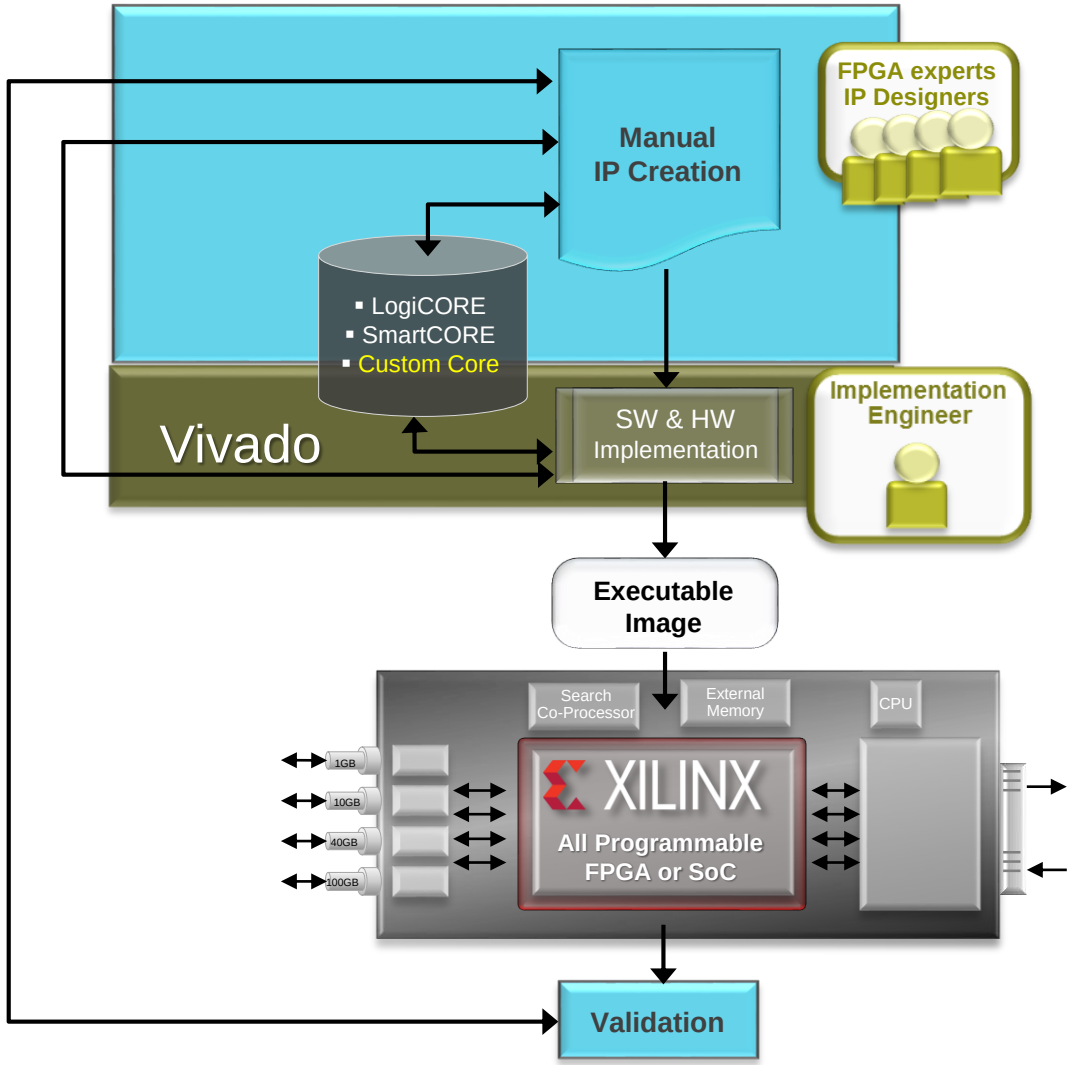


Data Plane Function Acceleration – Core to Edge

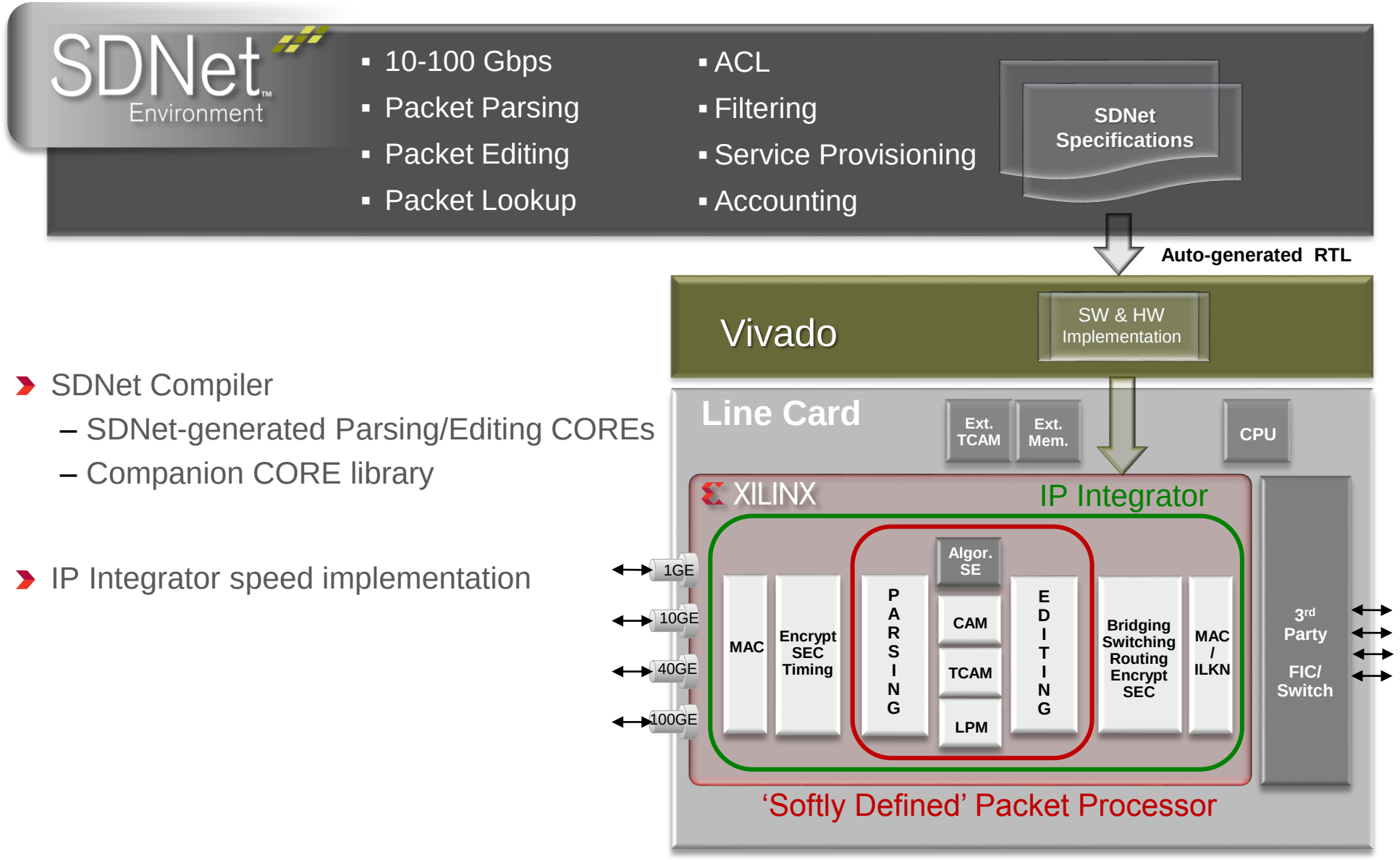


Traditional Data Plane Implementation with FPGA

- Large FPGA Team
- IP Design/Support – Complex
- Extensibility (ECOs) - Costly



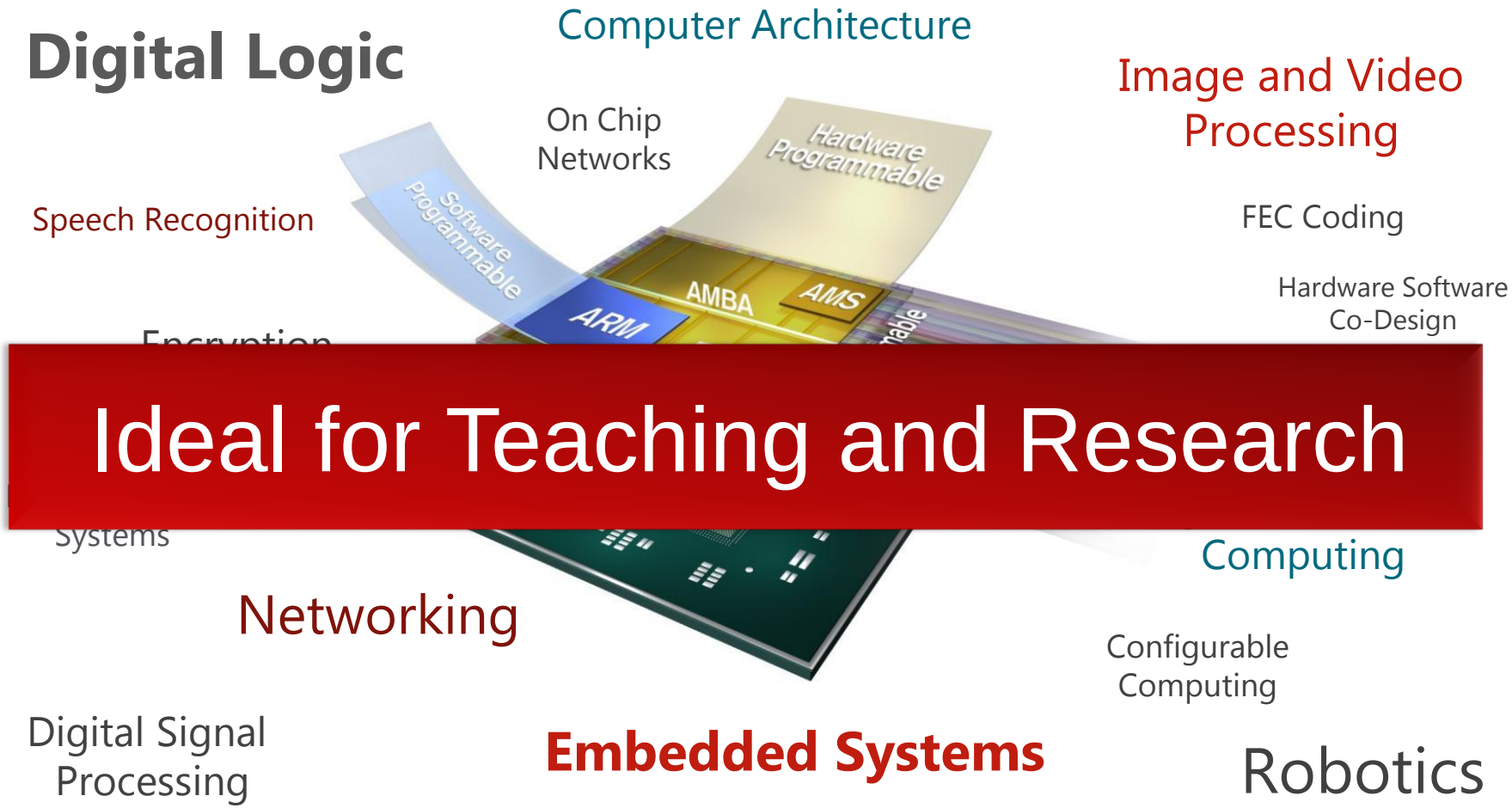
Methodology: High Level Description-to-Silicon



- SDNet Compiler
  - SDNet-generated Parsing/Editing COREs
  - Companion CORE library
- IP Integrator speed implementation

Xilinx University Program

Xilinx “All Programmable” solution span the digital spectrum



XILINX

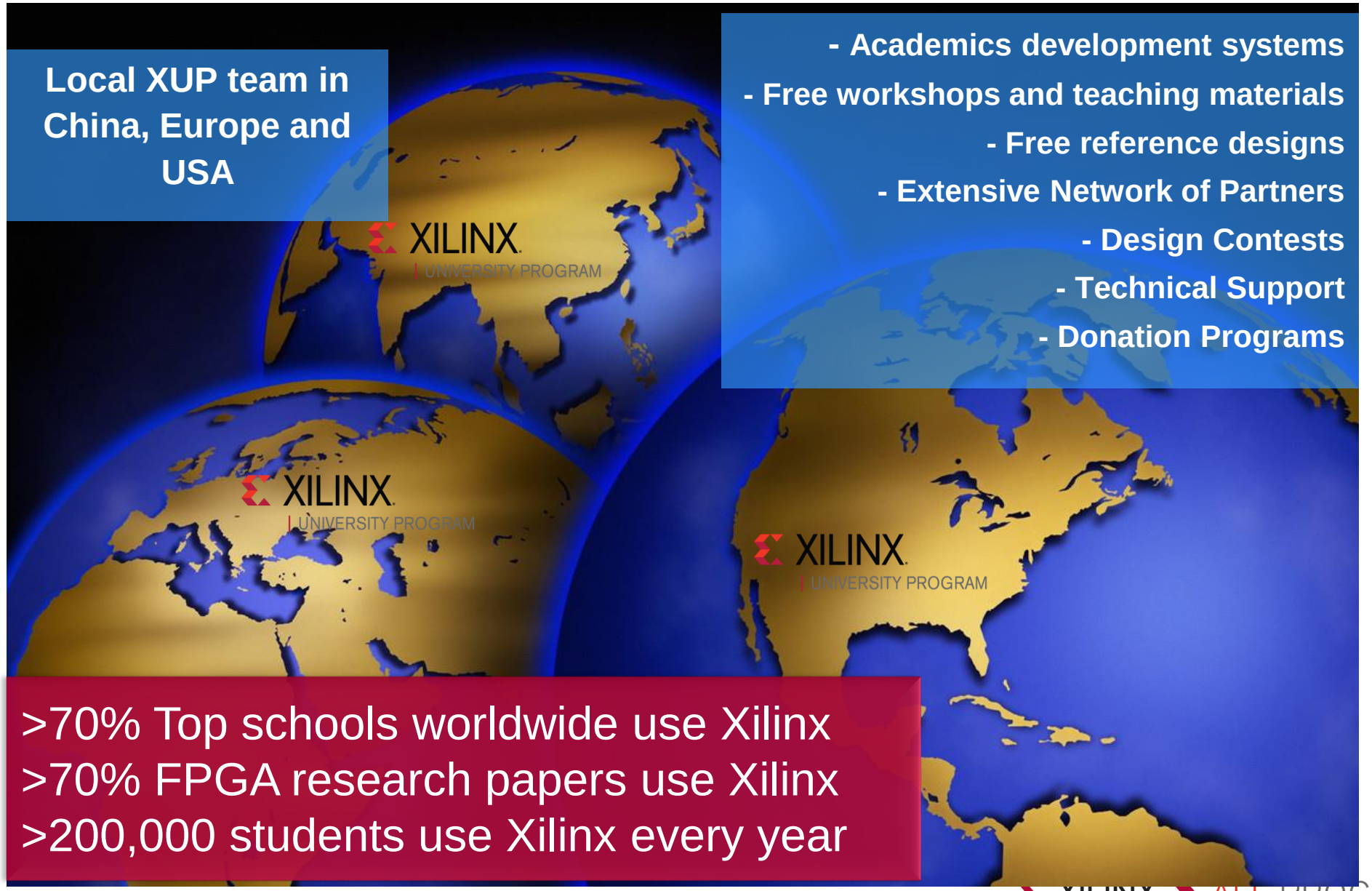
UNIVERSITY PROGRAM

Provide the best possible enabling technology for academics

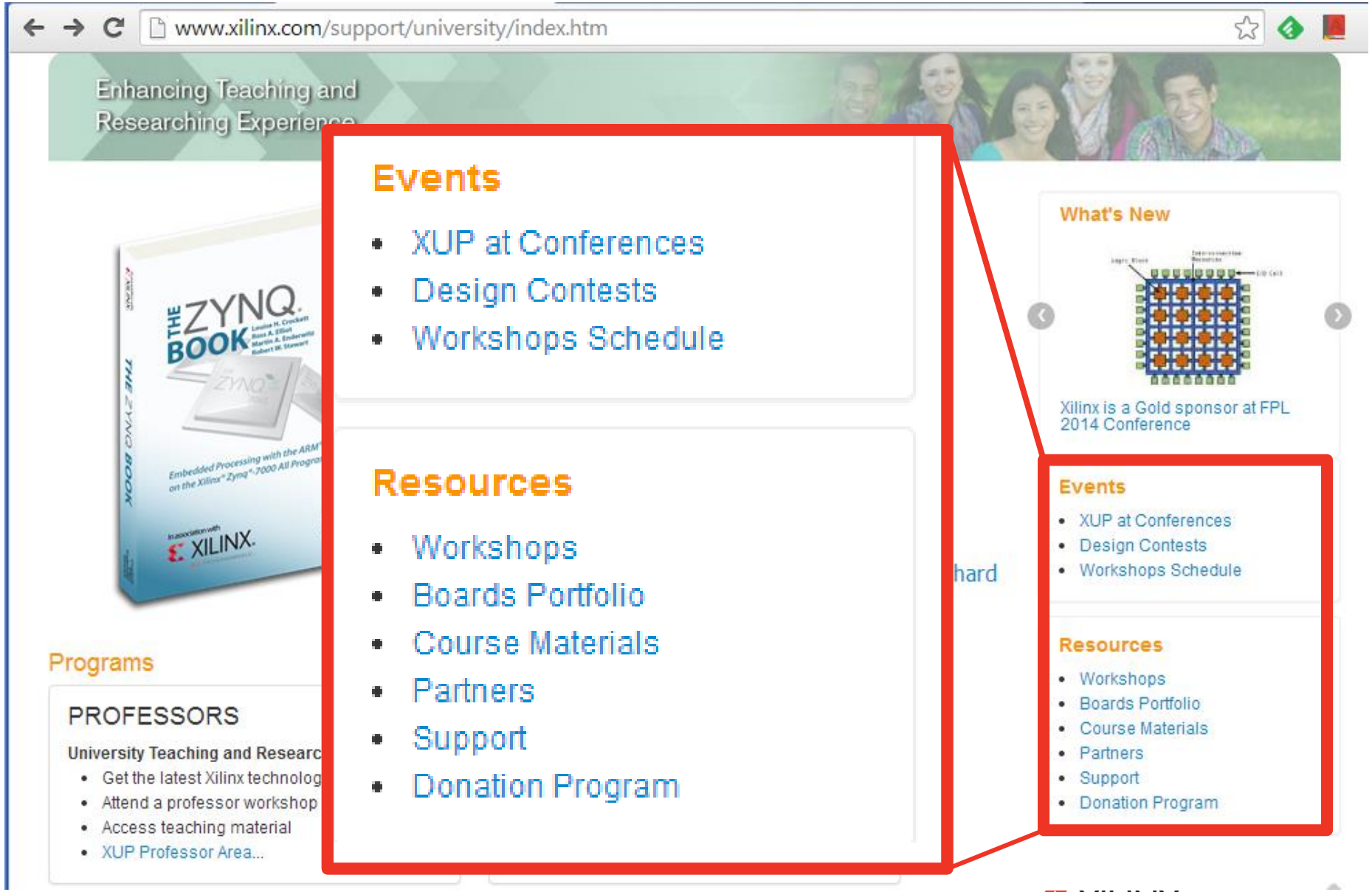
Identify, foster and support the best teaching and research practices

Partner to disseminate best practice as widely as possible

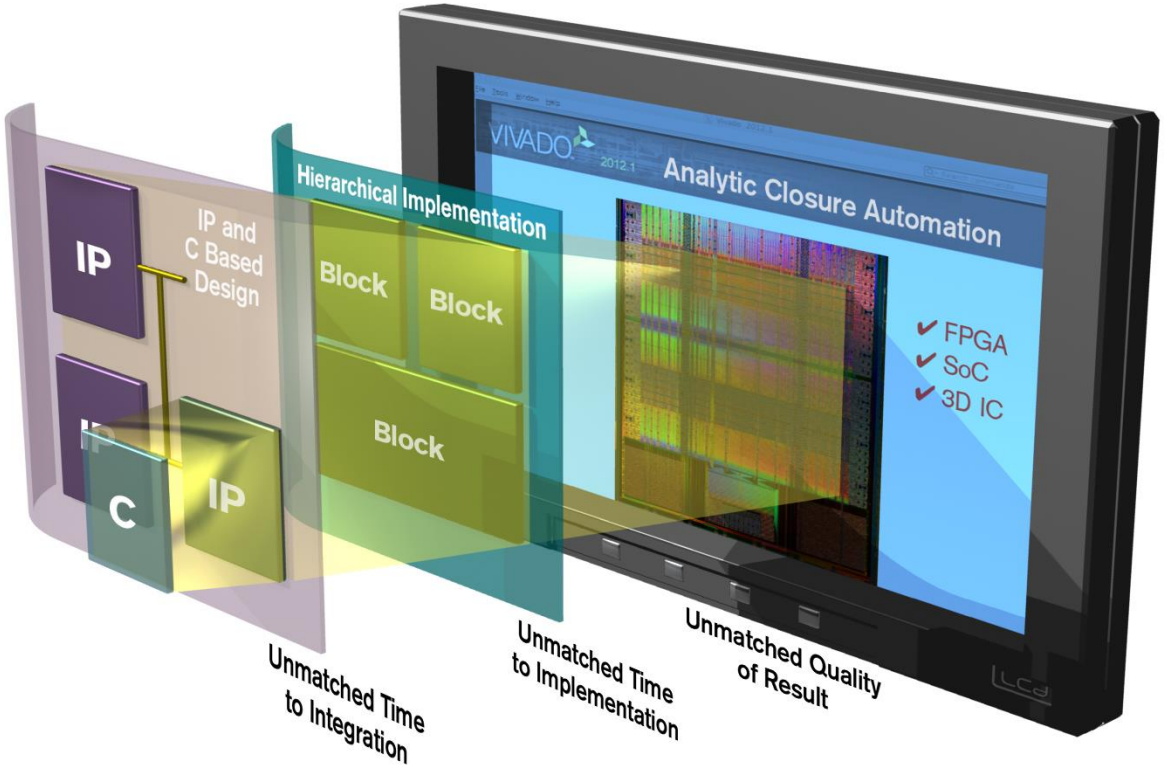
Xilinx is committed to University Program



Xilinx University Website



The ASIC Strength Design Suite



VIVADO™

- ✓ Delivering 4x productivity, turning months to weeks
- ✓ C and standards-based IP integration with fast verification
- ✓ WebPACK free for download
- ✓ System edition available at special university pricing and through donation program

XUP 7-Series Boards powered by Vivado

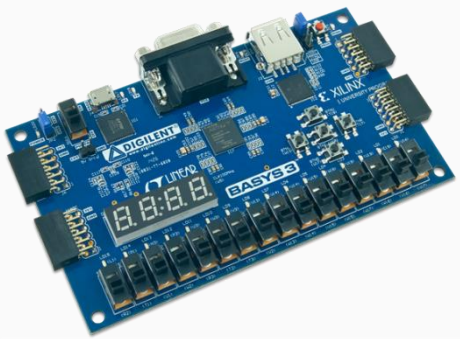
FPGA development boards

ARTIX

Nexys4 DDR

Artix-7 100T
128MiB DDR
Ethernet, SD card
USB, PMODs
Accelerometer, VGA
Microphone, Audio amp

$159 (Academic)



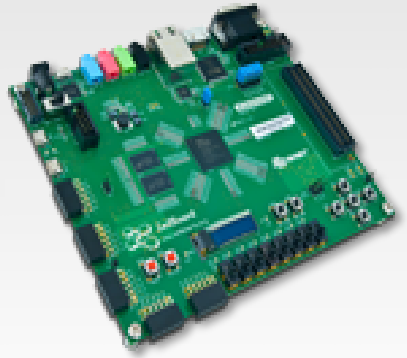
Basys3

Artix-7 35T
16 switches & LEDs
USB-UART
4 PMODs
VGA

$79 (Academic)

SoC development boards

ZYNQ



ZedBoard

Zynq 7020
512MB DDR3
256Mb Quad-SPI Flash
FMC, PMODs, USB
HDMI VGA, OLED
I²S Audio Codec

$319 (Academic)

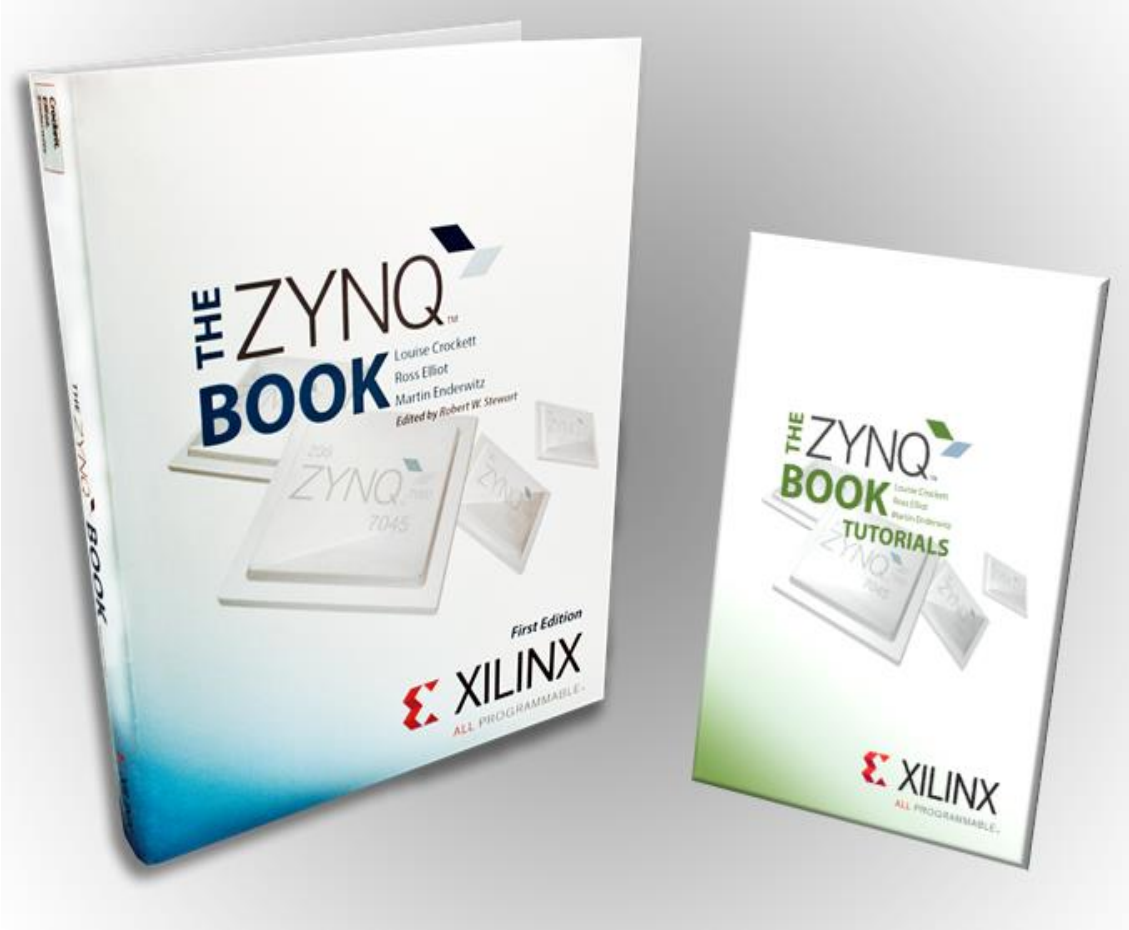
ZYBO

Zynq 7010
512MB DDR3
128Mb Serial Flash
Ethernet, Micro SD
USB, PMODs
HDMI, VGA
Audio Codec w/ in & out

$125 (Academic)

The Zynq Book

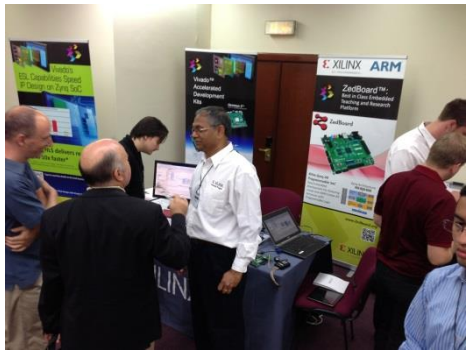
Embedded Processing with ARM Cortex-A9 on the Xilinx Zynq All Programmable SoC



- **Perfect introduction book for emerging Zynq developers**
- **Comprehensive Tutorials**
- **Over 15,000 downloads today**
- **Best Sellers on Amazon**
- **Free PDF download www.zynqbook.com**

XUP In Action

OpenHW 2014
开源硬件与嵌入式计算大赛
Open Source Hardware and
Embedded Computing Contest 2014



FPL 2014

FPGA 2014

FPT'14
Shanghai

Workshop

Embedded System Design Flow on Zynq using Vivado

High-Level Synthesis Flow using Vivado

Xilinx 7 Series FPGA Architecture and Vivado Tool Flow

無限創意
乘ARM飛翔
UNLIMITED COMPUTING ON ARM
ARM Cortex™-A平台
打造智慧運算的無限可能

FCCM 2014

EMECS-thon

MIT SoC Design Competition
ONE BOARD, ONE MONTH, UNLIMITED POSSIBILITIES...

XILINX
UNIVERSITY PROGRAM

Any questions?

Reach Us @

http://www.xilinx.com/university

Conclusion

- **Modern FPGAs are complex MP-SoC with programmable logic, CPU, GPU, peripherals, IO… the ultimate accelerator**
- **Require different abstractions to program with productivity at different levels**
- **SDSoC to ease system-level programming**
- **SDAccel: OpenCL for host-accelerator style programming**
  - Add specific optimizations for FPGA power & efficiency tradeoffs
    - data sizes
    - #CU
    - Pipelines
    - Bus
    - Memories
    - Pipes
    - Specific interfaces & IO…
- **SDNet: DSL for software defined network**

Follow Xilinx

facebook.com/XilinxInc

twitter.com/XilinxInc

youtube.com/XilinxInc

linkedin.com/company/xilinx

plus.google.com/+Xilinx