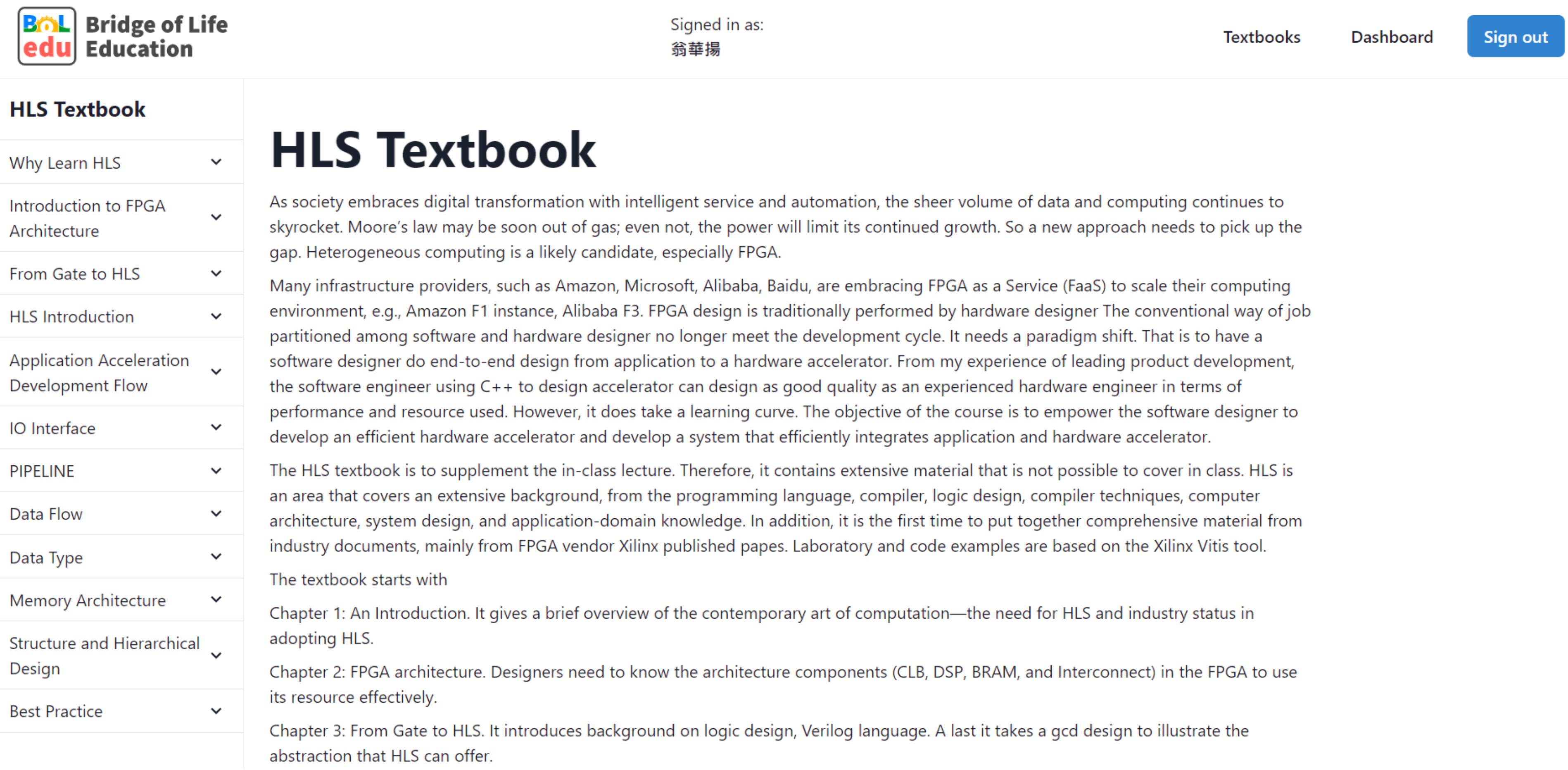
HLS 101

Lecturer: Hua-Yang Weng   Date: 2022/08/03



HLS Textbook

As society embraces digital transformation with intelligent service and automation, the sheer volume of data and computing continues to skyrocket. Moore's law may be soon out of gas; even not, the power will limit its continued growth. So a new approach needs to pick up the gap. Heterogeneous computing is a likely candidate, especially FPGA.

Many infrastructure providers, such as Amazon, Microsoft, Alibaba, Baidu, are embracing FPGA as a Service (FaaS) to scale their computing environment, e.g., Amazon F1 instance, Alibaba F3. FPGA design is traditionally performed by hardware designer The conventional way of job partitioned among software and hardware designer no longer meet the development cycle. It needs a paradigm shift. That is to have a software designer do end-to-end design from application to a hardware accelerator. From my experience of leading product development, the software engineer using C++ to design accelerator can design as good quality as an experienced hardware engineer in terms of performance and resource used. However, it does take a learning curve. The objective of the course is to empower the software designer to develop an efficient hardware accelerator and develop a system that efficiently integrates application and hardware accelerator.

The HLS textbook is to supplement the in-class lecture. Therefore, it contains extensive material that is not possible to cover in class. HLS is an area that covers an extensive background, from the programming language, compiler, logic design, compiler techniques, computer architecture, system design, and application-domain knowledge. In addition, it is the first time to put together comprehensive material from industry documents, mainly from FPGA vendor Xilinx published papes. Laboratory and code examples are based on the Xilinx Vitis tool.

The textbook starts with

Chapter 1: An Introduction. It gives a brief overview of the contemporary art of computation—the need for HLS and industry status in adopting HLS.

Chapter 2: FPGA architecture. Designers need to know the architecture components (CLB, DSP, BRAM, and Interconnect) in the FPGA to use its resource effectively.

Chapter 3: From Gate to HLS. It introduces background on logic design, Verilog language. A last it takes a gcd design to illustrate the abstraction that HLS can offer.

- See the below textbook for details
  - https://boledu-next-chakra.vercel.app/textbooks/hls-textbook
- Course:
  - NTUEE EEE5060 Application Acceleration with High-Level-Synthesis
  - NTUEE EEE5029 Multimedia System-on-chip Design

Outline

- Why HLS?
- HLS IP Flow
- Pragma Introduction
- Design Flow
- Labs

Outline

- Why HLS?
- HLS IP Flow
- Pragma Introduction
- Design Flow
- Labs

Why HLS?

Why HLS?

- Software Defined Hardware (SDH)
- Computational Efficiency
- Design and Verification Productivity
- Improve Quality of Results (QoR)
- Fast system prototyping
- Fast architecture exploration
  - C++ ⟵⟶ python v.s. verilog ⟵⟶ HLS

Why HLS?

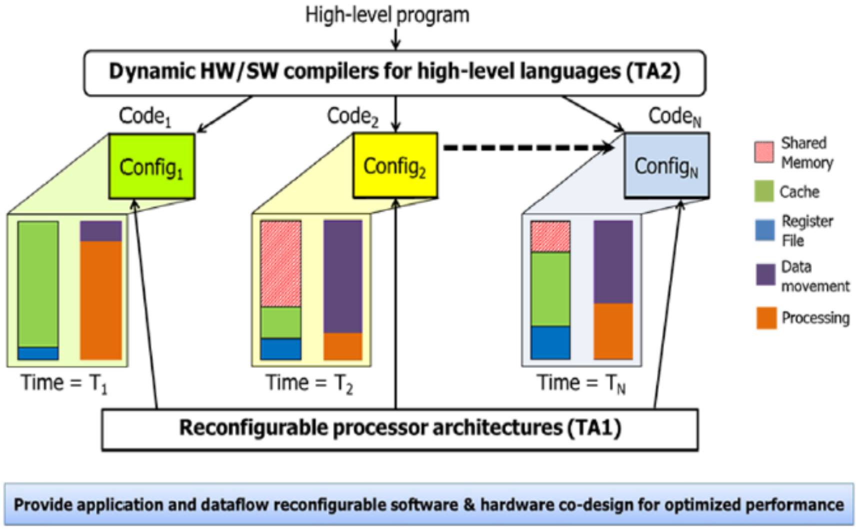
- Software Defined Hardware (SDH)
  - Defense Advanced Research Projects Agency
- Computational Efficiency
- Design and Verification Productivity
- Improve Quality of Results (QoR)
- Fast system prototyping
- Fast architecture exploration
  - C++ ⟷ python v.s. verilog ⟷ HLS

DARPA – Software Defined Hardware (SDH) Program

***"In modern warfare, decisions are driven by information. …
The ability to exploit this data to understand and predict the world around us is an asymmetric advantage for the Department of Defense (DoD)."***

Goal of SDH:

- *Compute efficiency (GOPs/Watt)* in SDH system to be at efficiencies within 5X of ASICs and 500-1000X better than CPU implementation.
- The same programmability as current NumPy/Python implementation



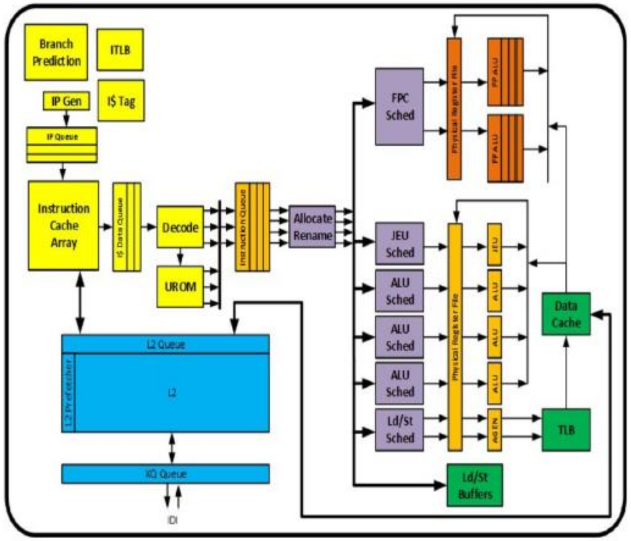
SDH Program Concept and Structure

Slides from NTUEE EEE5060 Application Acceleration with High-Level-Synthesis

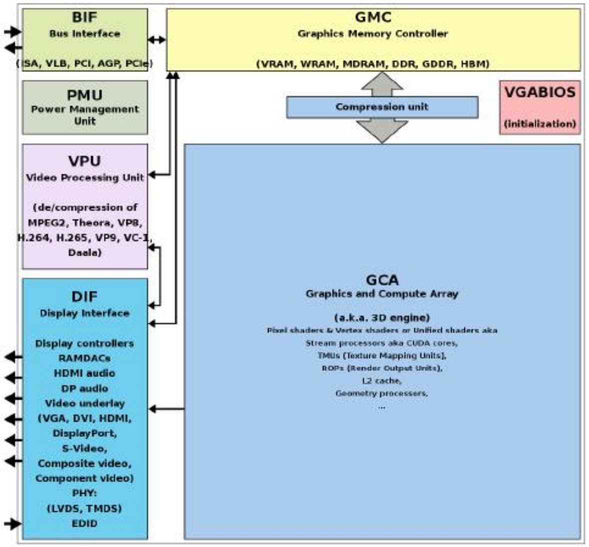
Why HLS?

- Software Defined Hardware (SDH)
- Computational Efficiency
- Design and Verification Productivity
- Improve Quality of Results (QoR)
- Fast system prototyping
- Fast architecture exploration
  - C++ ⟵⟶ python v.s. verilog ⟵⟶ HLS

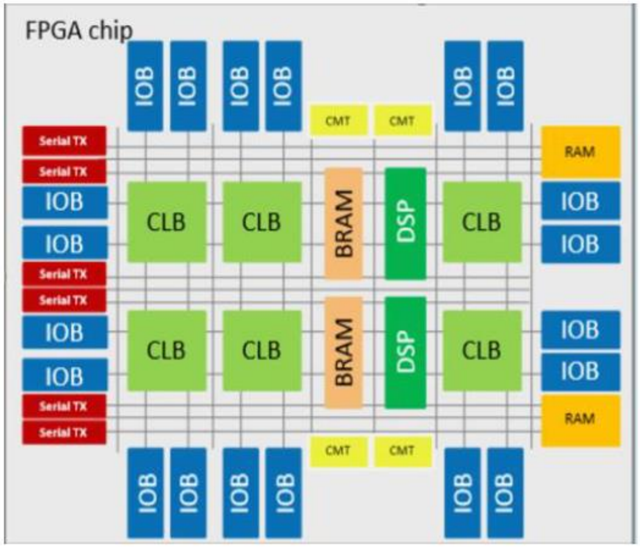
Computational Power



- Highly flexibility
- Low computing efficiency
- General software processing


BIF
Bus Interface
(ISA, VLB, PCI, AGP, PCIe)
GMC
Graphics Memory Controller
(VRAM, WRAM, MDRAM, DDR, GDDR, HBM)
PMU
Power Management Unit
Compression unit
VGABIOS
(initialization)
VPU
Video Processing Unit
(de/compression of MPEG2, Theora, VP8, H.264, H.265, VP9, VC-1, Daala)
DIF
Display Interface
Display controllers
RAMDACs
HDMI audio
DP audio
Video underlay
(VGA, DVI, HDMI, DisplayPort, S-Video, Composite video, Component video)
PHY:
(LVDS, TMDS)
EDID
GCA
Graphics and Compute Array
(a.k.a. 3D engine)
Pixel shaders & Vertex shaders or Unified shaders aka
Stream processors aka CUDA cores,
TMUs (Texture Mapping Units),
ROPs (Render Output Units),
L2 cache,
Geometry processers,
...

- Better flexibility
- Average computing efficiency
- Suited to simple logic and SIMD computationally intensive task



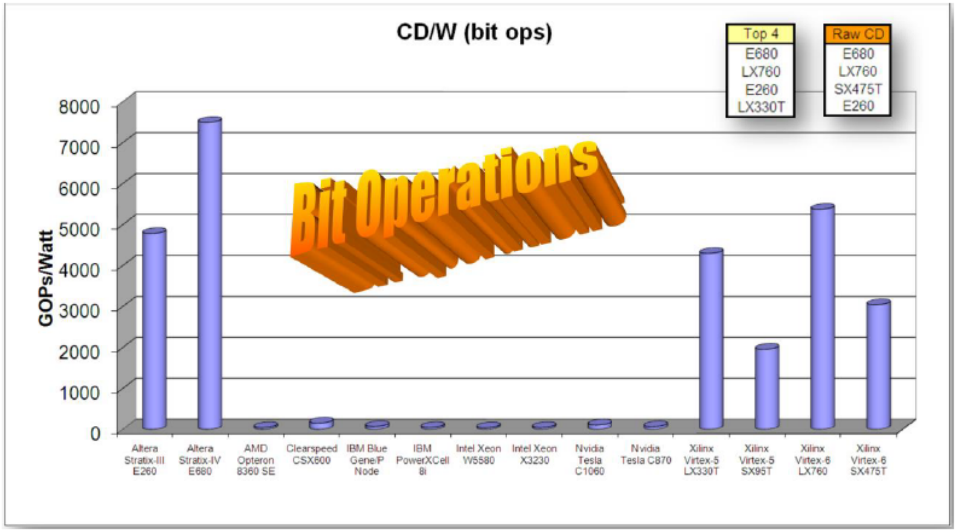
- Good flexibiliy
- Better computing efficiency
- Parallel computing, real-time processing, low power consumption
- Suited to hardware acceleration of specific algorithm

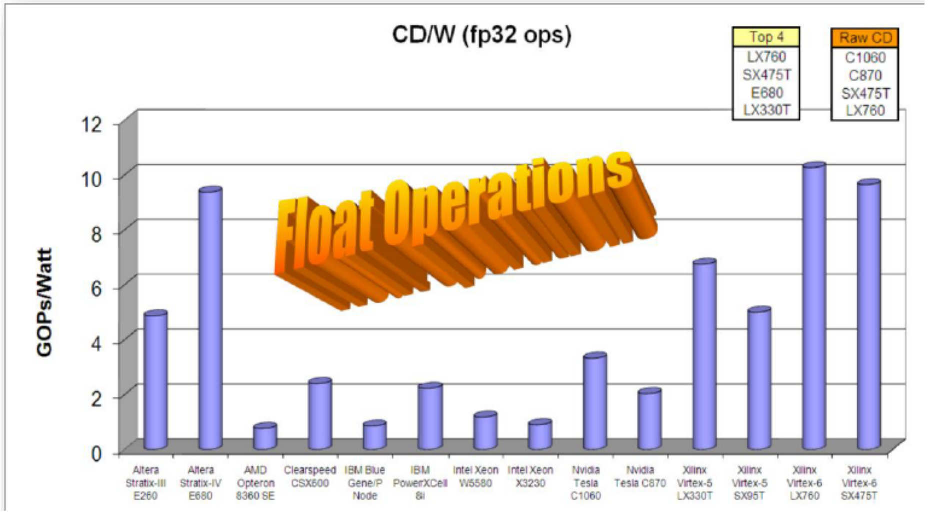
What area that FPGA is more compute efficient than GPU

- Irregular parallelism
- Customized data types
- Customized datapath, e.g. dataflow
- Efficient memory access semantics (random access, FIFO, stack etc.)

Why is FPGA not as popular as GPU?

CD/W: Computational Density (GOPs/s) per Watt



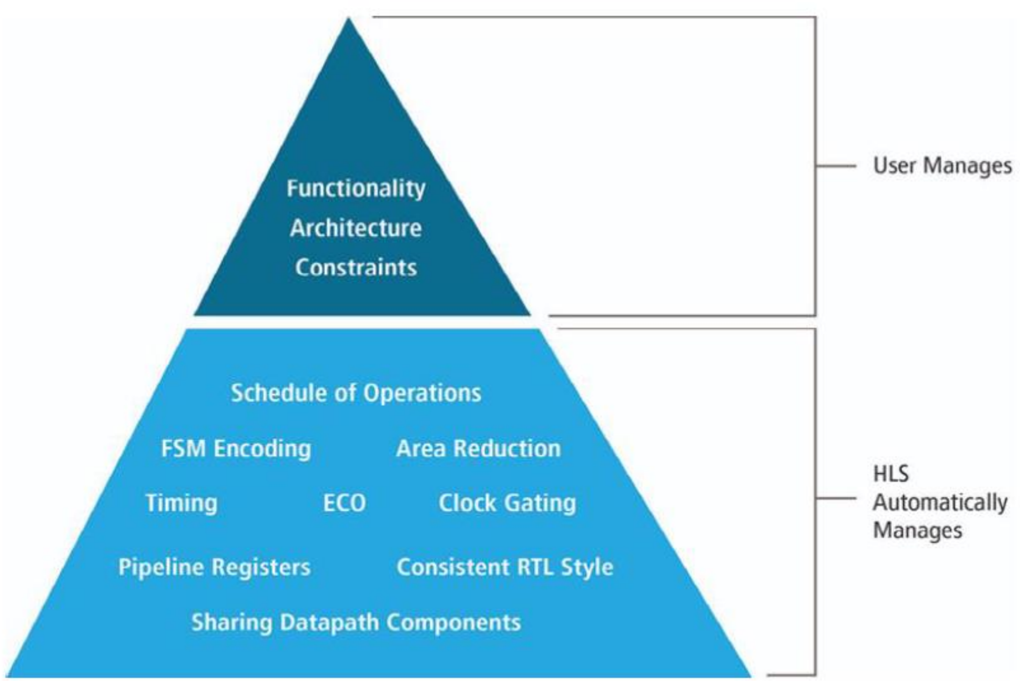


http://cas.ee.ic.ac.uk/people/gac1/DATE2011/Stitt.pdf

Why HLS?

- Software Defined Hardware (SDH)
- Computational Efficiency
- Design and Verification Productivity
- Improve Quality of Results (QoR)
- Fast system prototyping
- Fast architecture exploration
    - C++ ←→ python v.s. verilog ←→ HLS

Design and Verification Productivity



Functionality
Architecture
Constraints

User Manages

Schedule of Operations
FSM Encoding
Area Reduction
Timing
ECO
Clock Gating
Pipeline Registers
Consistent RTL Style
Sharing Datapath Components

HLS Automatically Manages

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

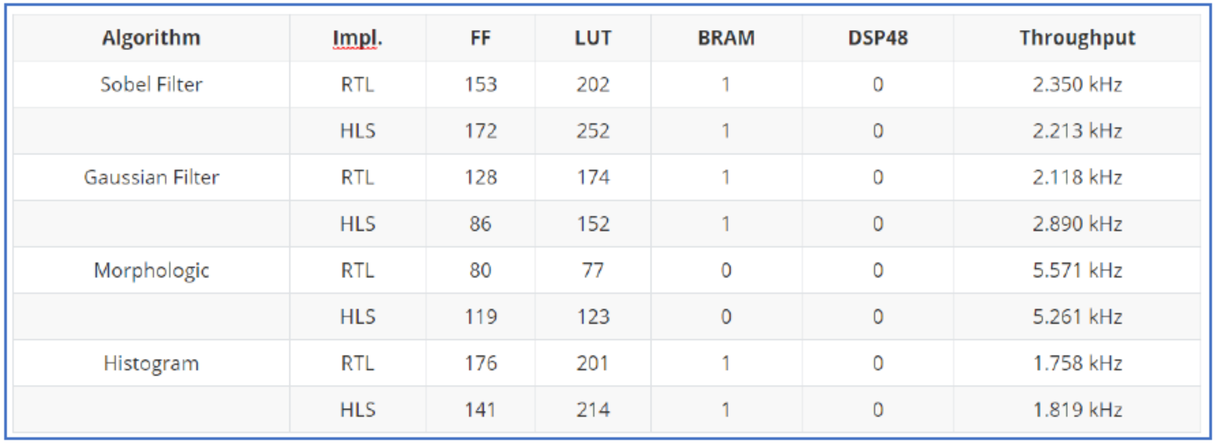
http://www.ecs.umass.edu/ece/labs/vlsicad/ece667/reading/hls-survey.pdf

Why HLS?

- Software Defined Hardware (SDH)
- Computational Efficiency
- Design and Verification Productivity
- Improve Quality of Results (QoR)
- Fast system prototyping
- Fast architecture exploration
  - C++ ⟷ python v.s. verilog ⟷ HLS

Improve Quality of Results (QoR)

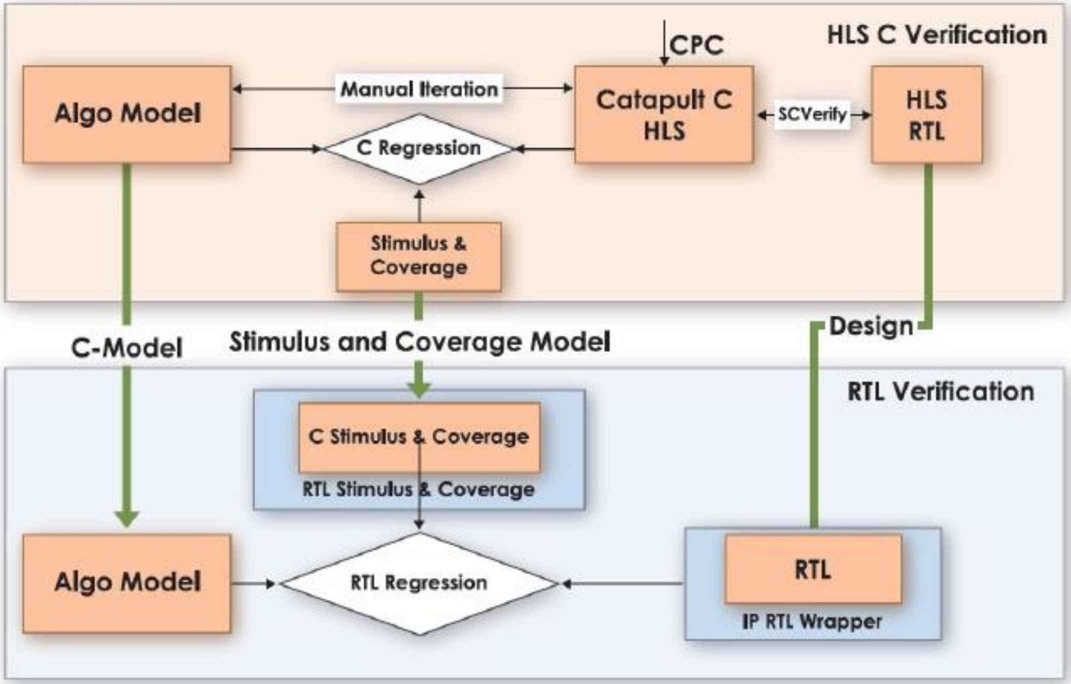
- **Advanced optimization algorithm** – leverage on continued research for better and intelligent synthesis algorithms
- **Allow more design space exploration** – quickly create many different implementation from one high-level description of the design. e.g. Explore SIMD parallelism with a single parameters.



Algorithm	Impl.	FF	LUT	BRAM	DSP48	Throughput
Sobel Filter	RTL	153	202	1	0	2.350 kHz
	HLS	172	252	1	0	2.213 kHz
Gaussian Filter	RTL	128	174	1	0	2.118 kHz
	HLS	86	152	1	0	2.890 kHz
Morphologic	RTL	80	77	0	0	5.571 kHz
	HLS	119	123	0	0	5.261 kHz
Histogram	RTL	176	201	1	0	1.758 kHz
	HLS	141	214	1	0	1.819 kHz

"A Comparative Study between RTL and HLS for Image Processing Application with FPGAs" https://escholarship.org/uc/item/9vx1s37b

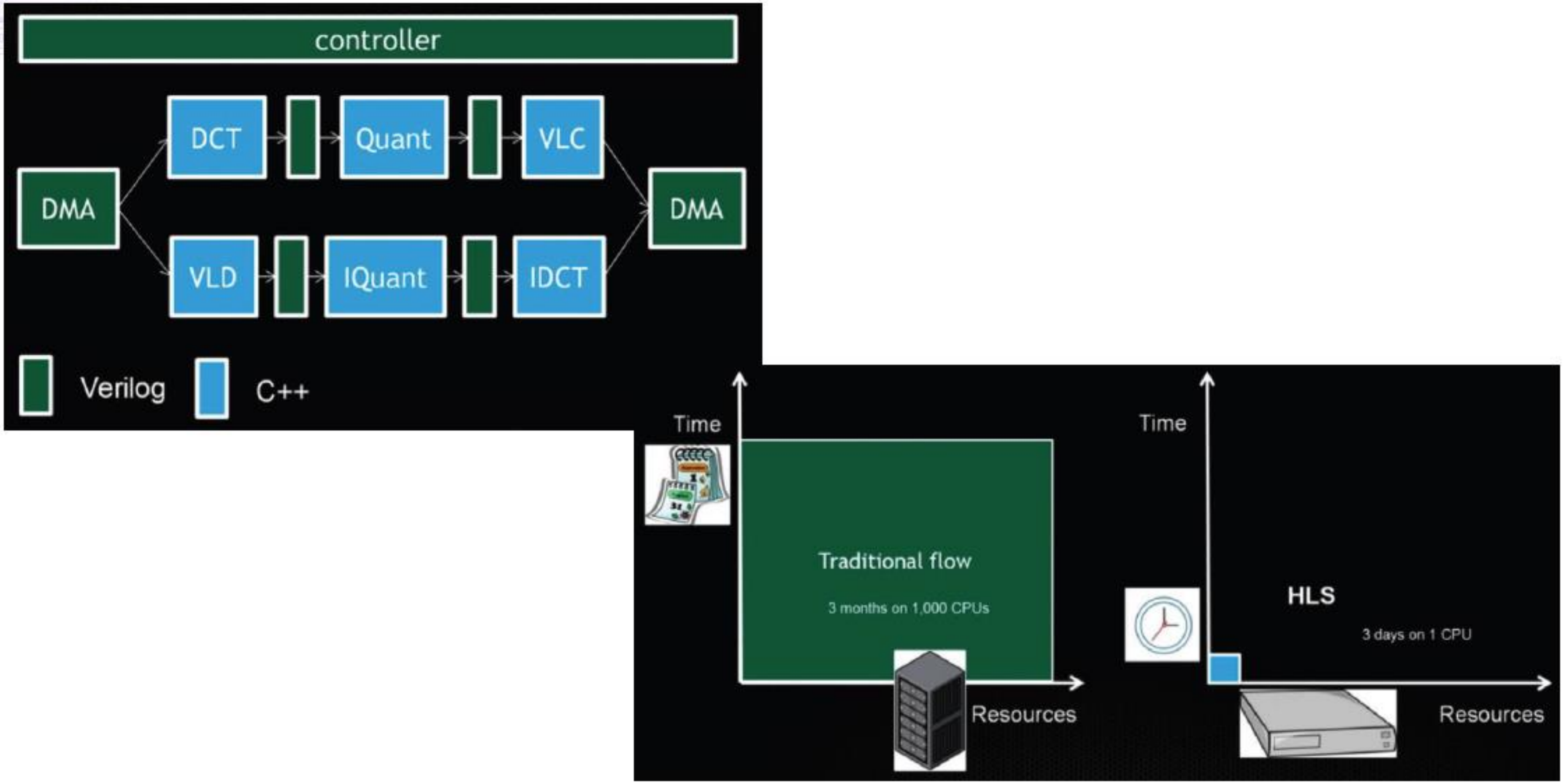
Example from Qualcomm



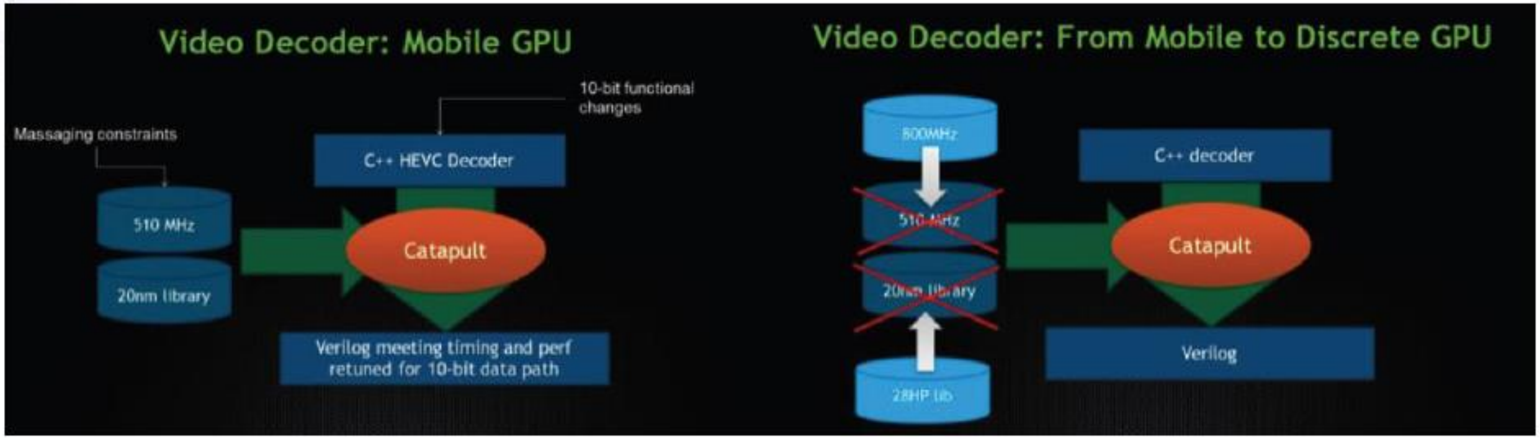
- **The HLS design code space is much smaller at the C-level** than at the RTL, making it easier to verify and correct; the **100x faster simulation speeds** enable us to detect problems and close coverage magnitudes faster than in RTL
- With the HLS methodology, what is verified in C stays verified in the RTL domain. As a result, most of the bugs are found and corrected in C.
- When HLS/HLV is done, the remaining work in the RTL environment is mostly at the interface level.

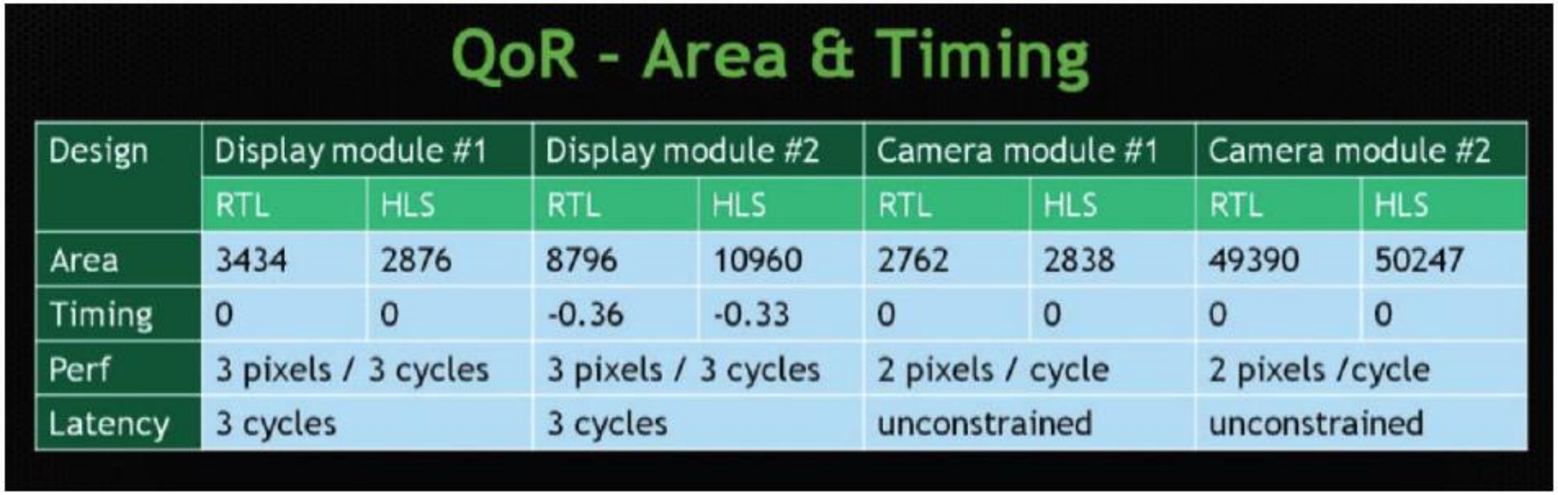
Slides from NTUEE EEE5029 Multimedia System-on-chip Design

Example from NVIDIA (Image Decoder)



Example from NVIDIA (Image Decoder)


Video Decoder: Mobile GPU
10-bit functional changes
Massaging constraints
C++ HEVC Decoder
510 MHz
20nm library
Catapult
Verilog meeting timing and perf retuned for 10-bit data path
Video Decoder: From Mobile to Discrete GPU
800MHz
~~510 MHz~~
~~20nm library~~
28HP lib
C++ decoder
Catapult
Verilog



QoR - Area & Timing

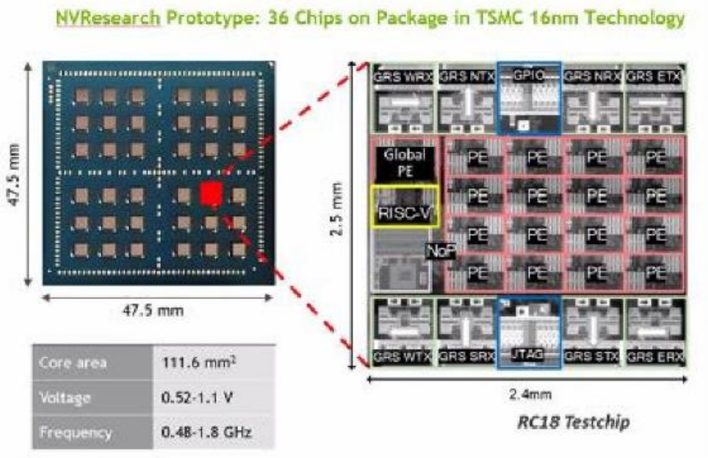
Design	Display module #1		Display module #2		Camera module #1		Camera module #2	
	RTL	HLS	RTL	HLS	RTL	HLS	RTL	HLS
Area	3434	2876	8796	10960	2762	2838	49390	50247
Timing	0	0	-0.36	-0.33	0	0	0	0
Perf	3 pixels / 3 cycles		3 pixels / 3 cycles		2 pixels / cycle		2 pixels /cycle	
Latency	3 cycles		3 cycles		unconstrained		unconstrained	

Industry case - Nvidia

Nvidia Research – Machine Learning Accelerator

"10X Improvement in RTL design and verification effort compared to manual RTL"

- Enable full SoC level performance - < 2.6% from RTL in cycle count
- Low Design Effort – Spec-to-Tapeout in 6 months with < 10 researchers

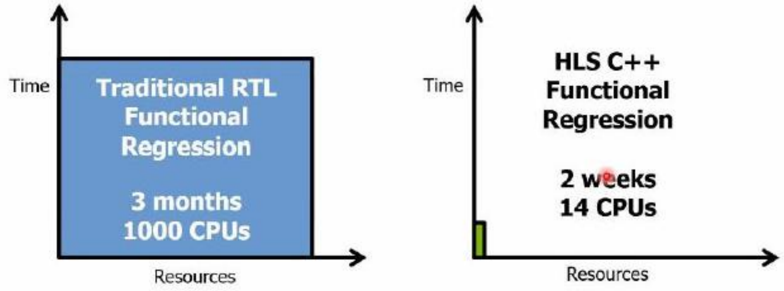

NVResearch Prototype: 36 Chips on Package in TSMC 16nm Technology

Core area	111.6 mm^2
Voltage	0.52-1.1 V
Frequency	0.48-1.8 GHz

RC18 Testchip

Nvidia Xavier 12nFF SoC

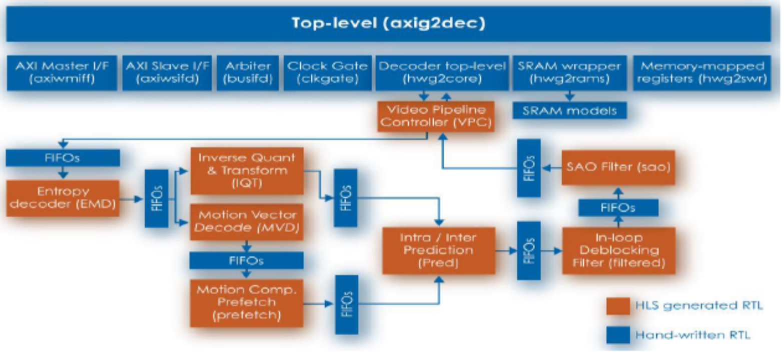
- C++ functional verification runtime ~500x less resource than RTL
- Fast verification makes rapid product changes possible
    - VP9/HEVC code from 8 to 10 bit color depth in 2 weeks
    - Change from 20nm/500Mhz to 28/nm/800Mhz in 3 days with HLS

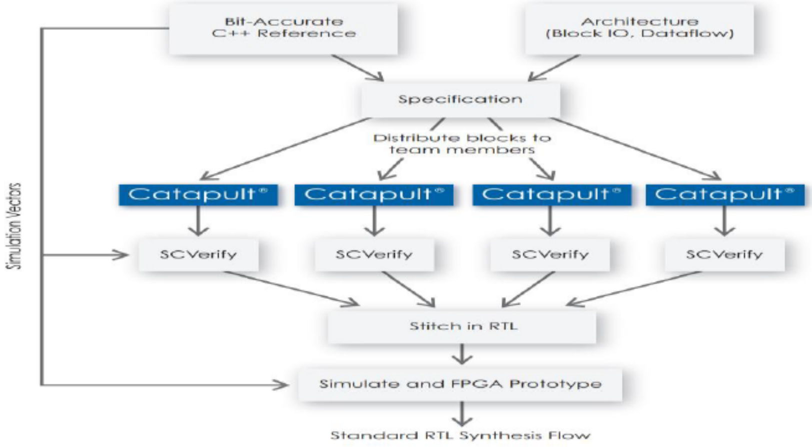


https://www.mentor.com/hls-lp/multimedia/player/nvidia-design-and-verification-of-a-machine-learning-accelerator-soc-using-an-object-oriented-hls-based-design-flow-2cea13e3-93cf-4539-bac6-01f75c263fc1

Industry Case – Google Designs VP9 CODEC in Half the Time

- Time to Verified RTL: 2x faster
  - Built in under 6 monts v.s. 1 year for RTL
  - 69k lines of C++ v.s. 1.2 millon lines of Verilog
- Simulation Speed: 500x faster
  - RTL simulation: 70 servers and 2 days
  - C simulation: 3 servers in 2 hours
- > 99% bugs caught in C simulation
- Benefits from the view of Google
  - 90% less code, less bug
  - Flexibility – SW-like process, late-stage algorithm changes
  - Rapid HW prototyping – rapidly evaluate new idea, algorithms

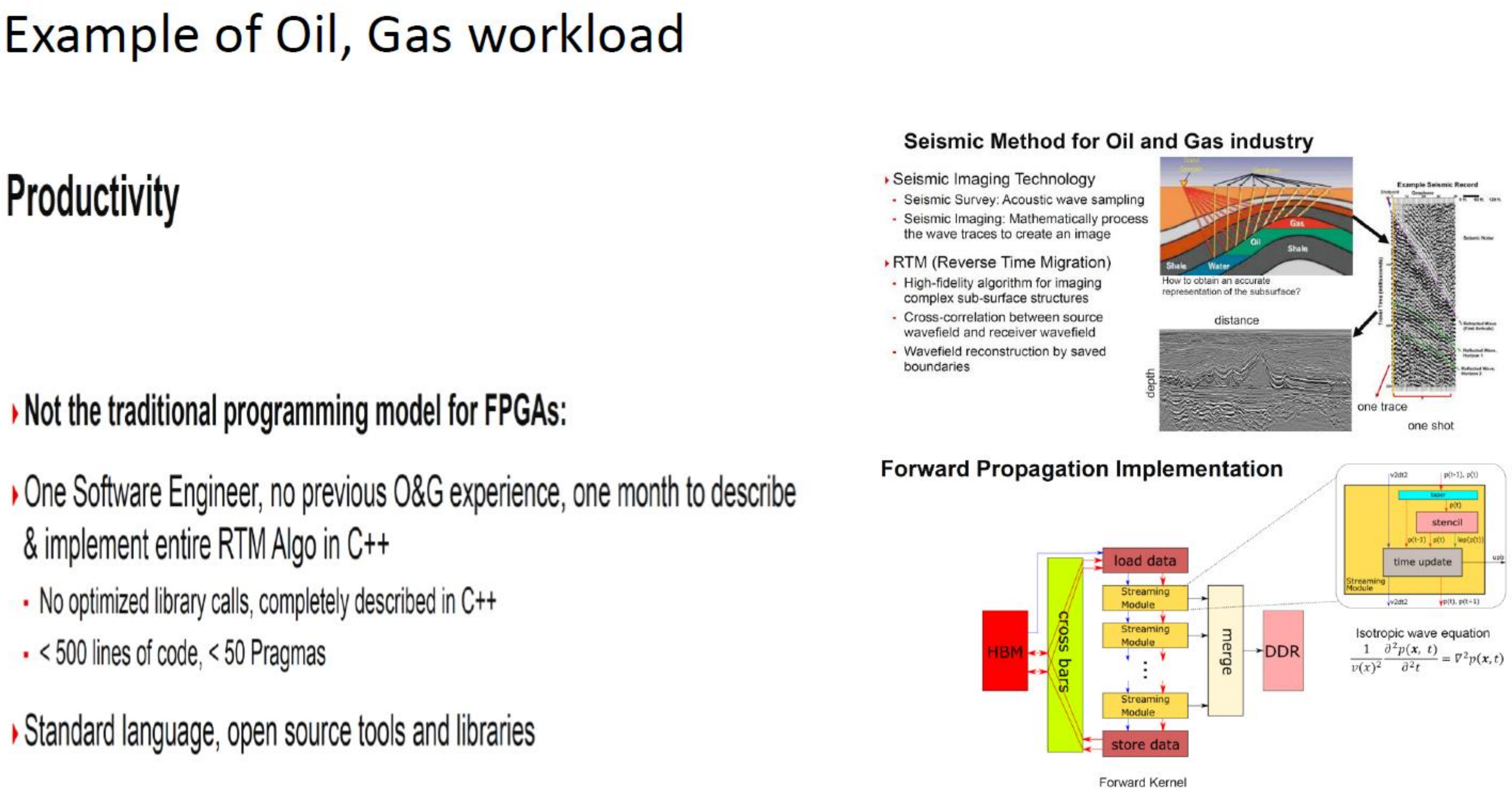

Top-level (axig2dec)
AXI Master I/F (axiwmiff)
AXI Slave I/F (axiwsifd)
Arbiter (busifd)
Clock Gate (clkgate)
Decoder top-level (hwg2core)
SRAM wrapper (hwg2rams)
Memory-mapped registers (hwg2swr)
Video Pipeline Controller (VPC)
SRAM models
FIFOs
Entropy decoder (EMD)
FIFOs
Inverse Quant & Transform (IQT)
Motion Vector Decode (MVD)
FIFOs
Motion Comp. Prefetch (prefetch)
FIFOs
FIFOs
Intra / Inter Prediction (Pred)
FIFOs
SAO Filter (sao)
FIFOs
FIFOs
In-loop Deblocking Filter (filtered)
HLS generated RTL
Hand-written RTL


Bit-Accurate C++ Reference
Architecture (Block IO, Dataflow)
Specification
Distribute blocks to team members
Catapult®
Catapult®
Catapult®
Catapult®
SCVerify
SCVerify
SCVerify
SCVerify
Stitch in RTL
Simulate and FPGA Prototype
Standard RTL Synthesis Flow
Simulation Vectors

EDU

https://go.mentor.com/4uNV1

Application Specific



Example of Oil, Gas workload

Productivity

- Not the traditional programming model for FPGAs:
- One Software Engineer, no previous O&G experience, one month to describe & implement entire RTM Algo in C++
  - No optimized library calls, completely described in C++
  - < 500 lines of code, < 50 Pragmas
- Standard language, open source tools and libraries

Seismic Method for Oil and Gas industry

- Seismic Imaging Technology
  - Seismic Survey: Acoustic wave sampling
  - Seismic Imaging: Mathematically process the wave traces to create an image
- RTM (Reverse Time Migration)
  - High-fidelity algorithm for imaging complex sub-surface structures
  - Cross-correlation between source wavefield and receiver wavefield
  - Wavefield reconstruction by saved boundaries

How to obtain an accurate representation of the subsurface?

Forward Propagation Implementation

Forward Kernel

Isotropic wave equation

$$\frac{1}{v(x)^2}\frac{\partial^2 p(\boldsymbol{x},\ t)}{\partial^2 t} = \nabla^2 p(\boldsymbol{x}, t)$$

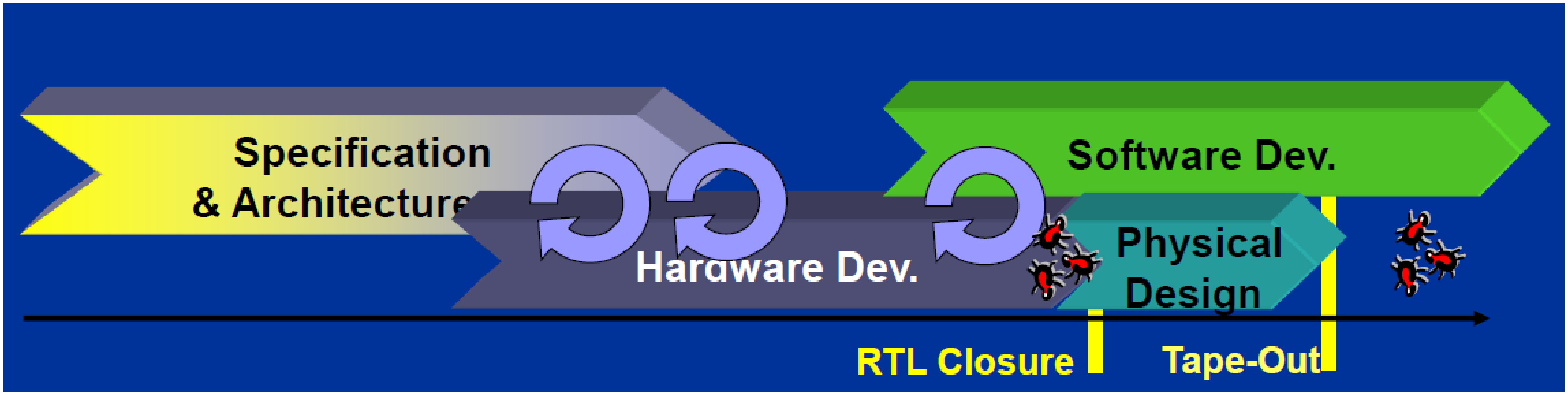
Slides from NTUEE EEE5060 Application Acceleration with High-Level-Synthesis

Why HLS?

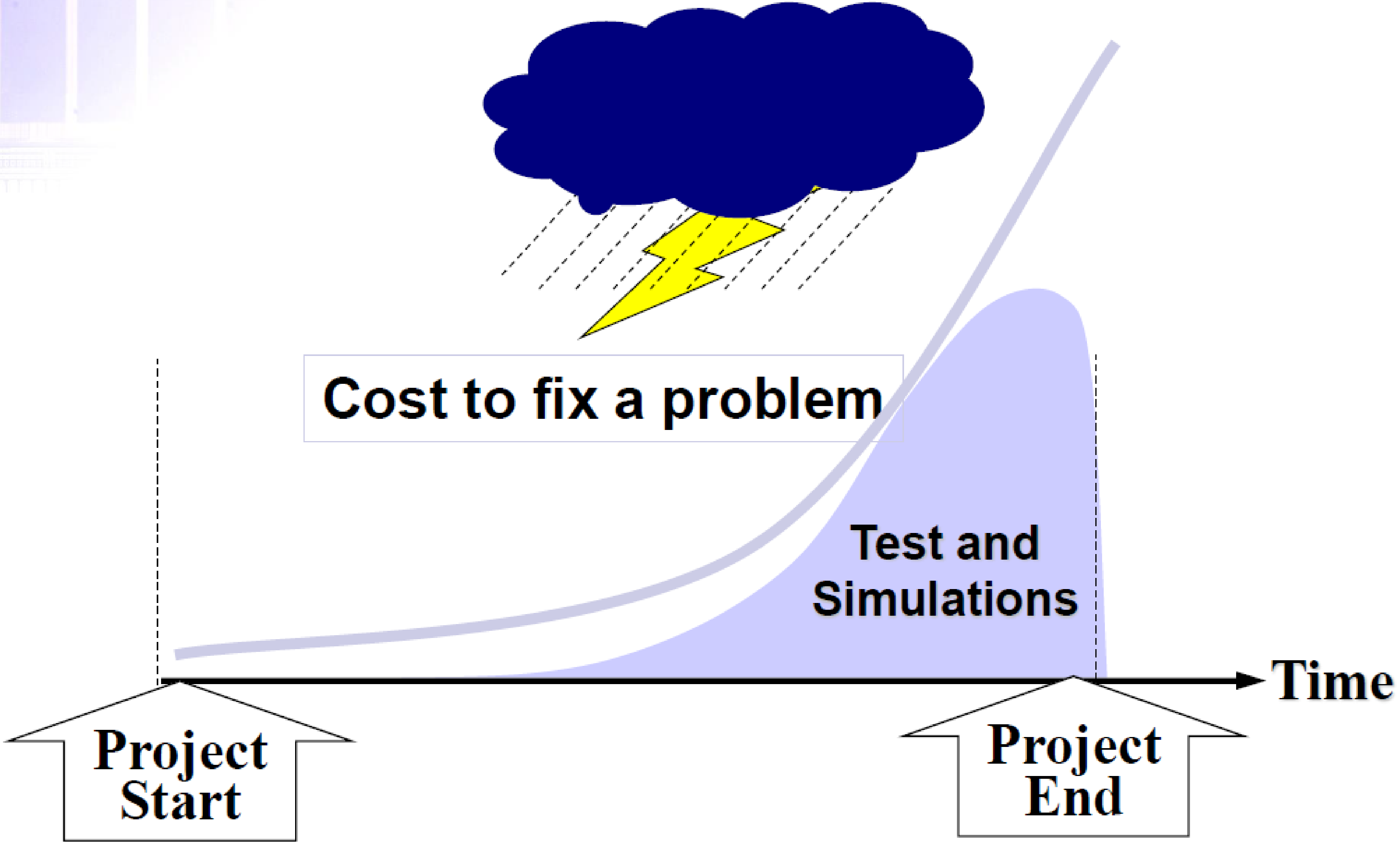
- Software Defined Hardware (SDH)
- Computational Efficiency
- Design and Verification Productivity
- Improve Quality of Results (QoR)
- Fast system prototyping: ESL
- Fast architecture exploration
    - C++ ⟷ python v.s. verilog ⟷ HLS

Typical SOC design flow

- Overlap in specification/architecture phase and RTL-design phase; multiple design changes
  - Architecture design done informally
- SW development starting late in the project

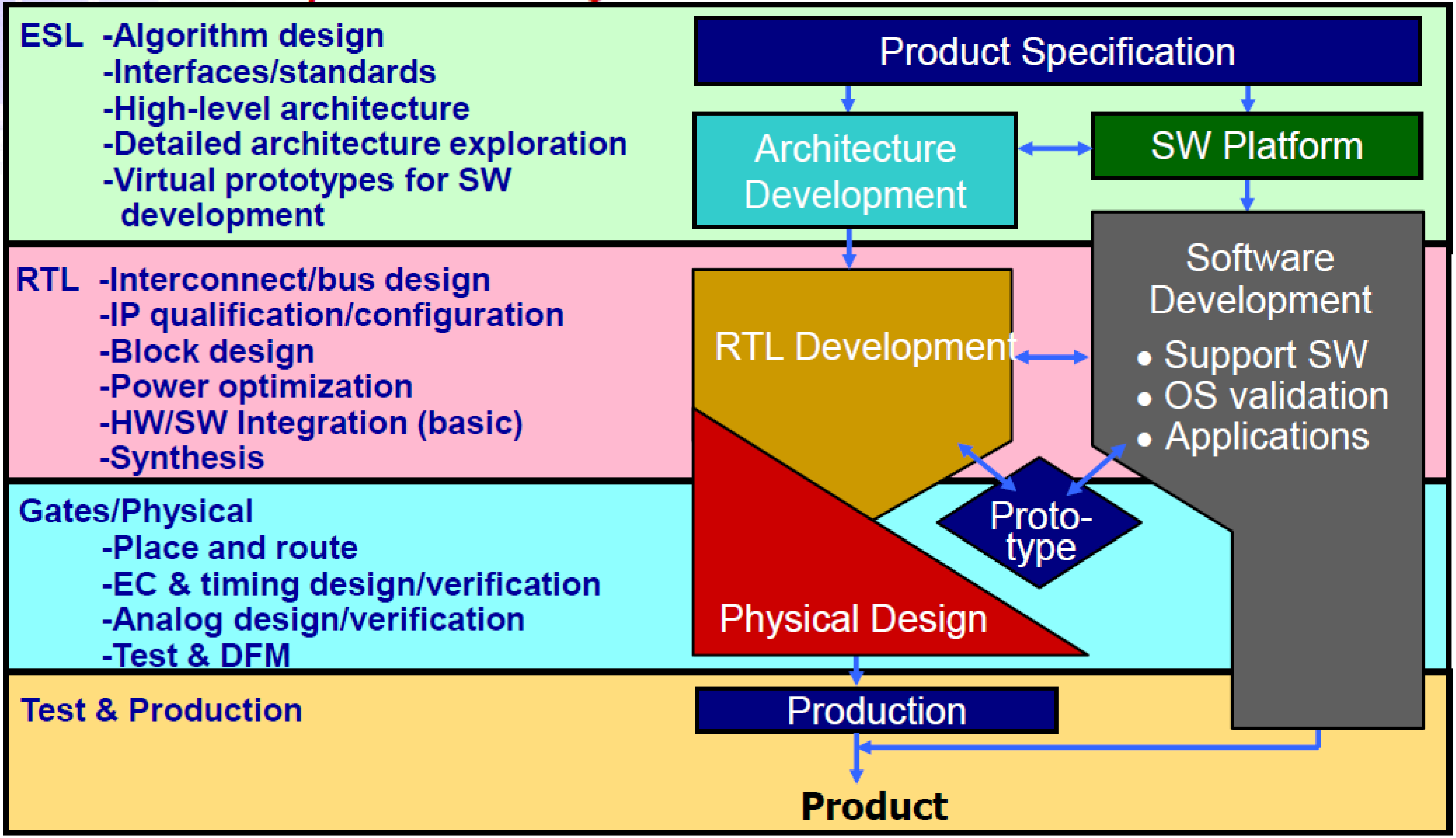


Verification at the Backend



Emerging SoC Design Flow (2/2)

ESL : Electronic System Level Design



ESL
- -Algorithm design
- -Interfaces/standards
- -High-level architecture
- -Detailed architecture exploration
- -Virtual prototypes for SW development

RTL
- -Interconnect/bus design
- -IP qualification/configuration
- -Block design
- -Power optimization
- -HW/SW Integration (basic)
- -Synthesis

Gates/Physical
- -Place and route
- -EC & timing design/verification
- -Analog design/verification
- -Test & DFM

Test & Production

Product Specification

Architecture Development

SW Platform

RTL Development

Software Development
- Support SW
- OS validation
- Applications

Proto-type

Physical Design

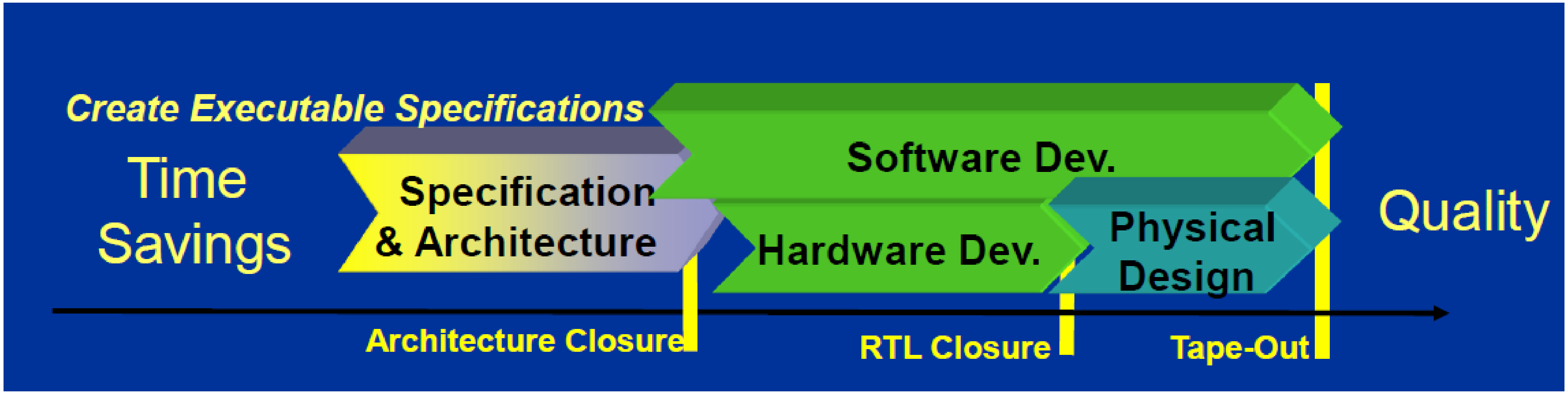
Production

Product

Source: Synopsys, Inc.

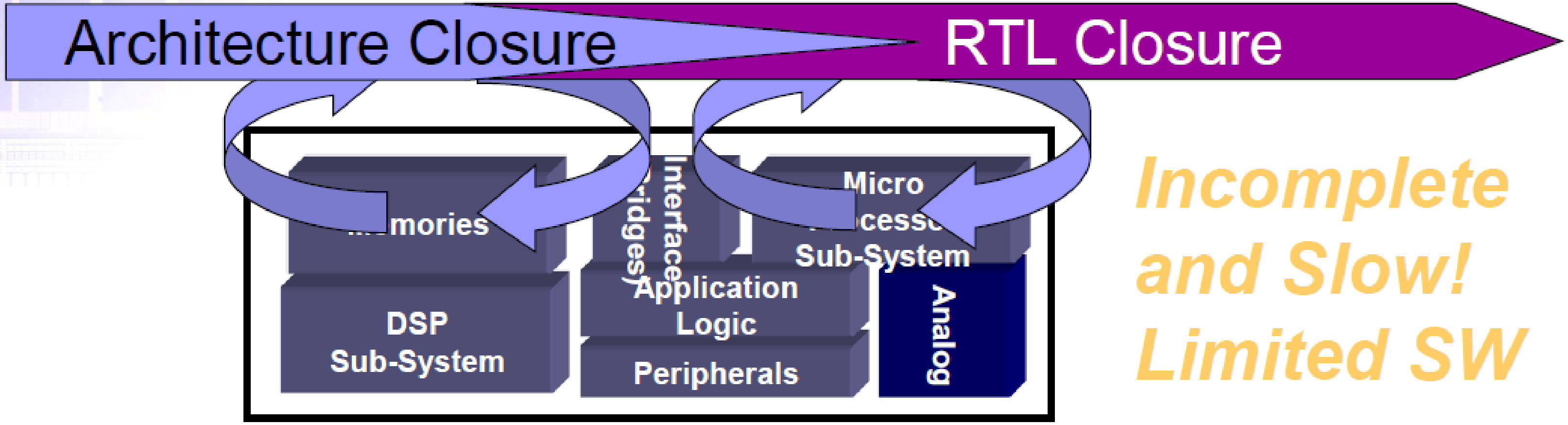
ESL: New SOC Design Flow

- Architecture closure
  - □ Achieve a reduction # of RTL iterations
  - □ Can perform concurrent HW and SW design
  - □ Shorten the time it takes to get to golden RTL



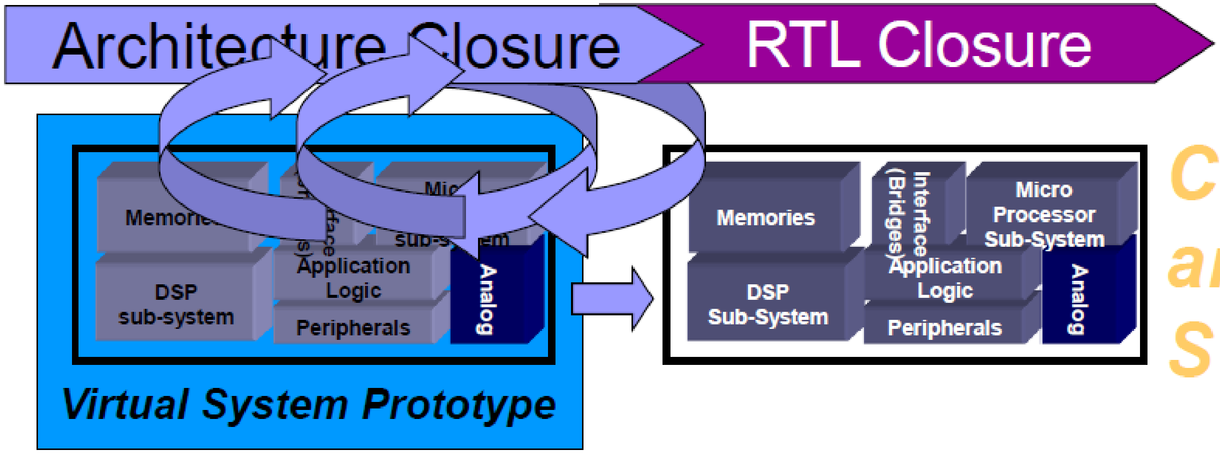
SOC Design Flows

Typical Flow: *Step 1 and 2 performed on RTL model*



Incomplete and Slow! Limited SW

New Flow: *Step 1 on transaction level, step 2 on RTL model*

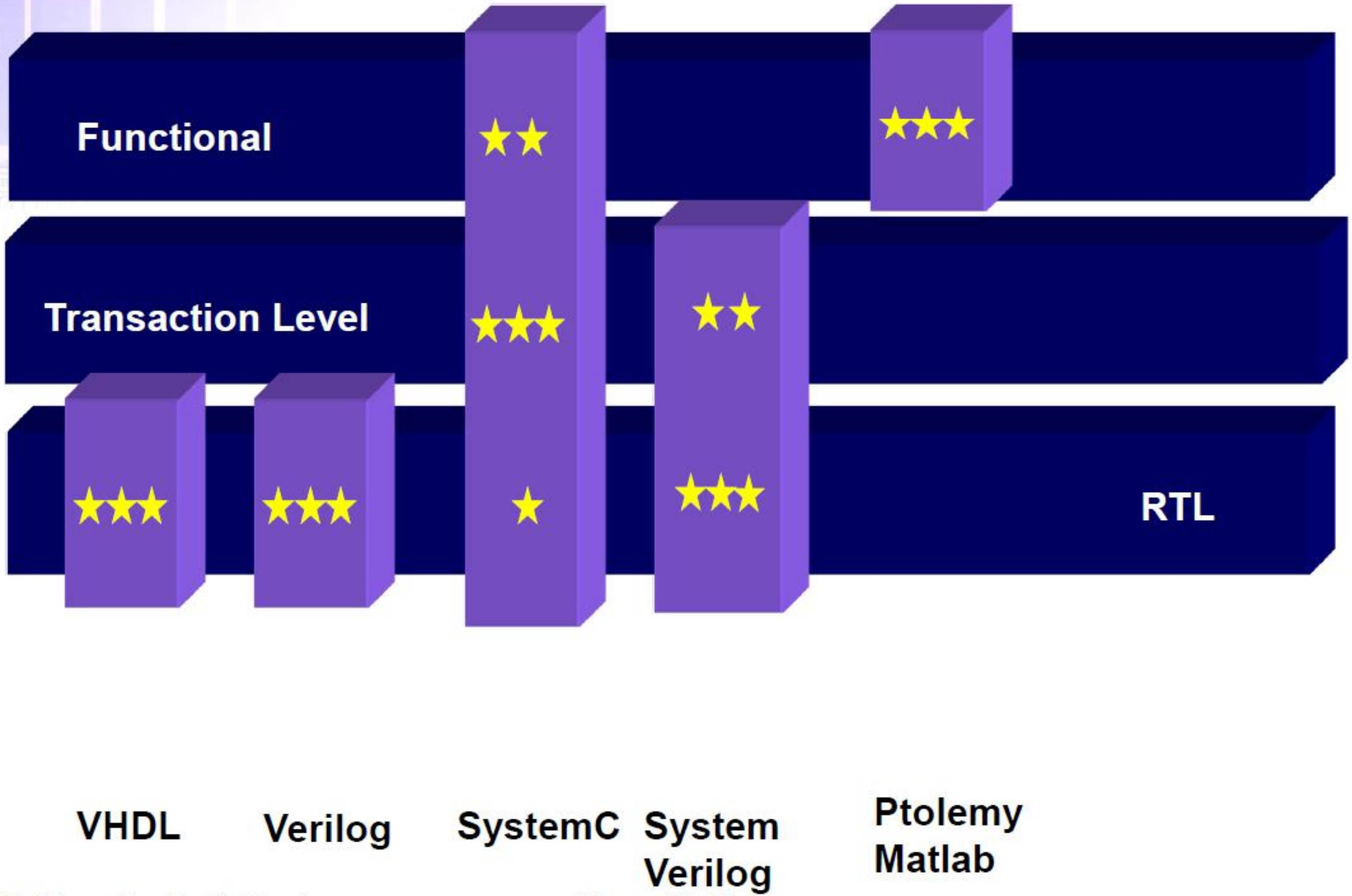


Complete and Fast! SW Early

Multimedia SoC Design *Shao-Yi Chien* **49**

Languages

★★★ excellent
★★ good
★ ok

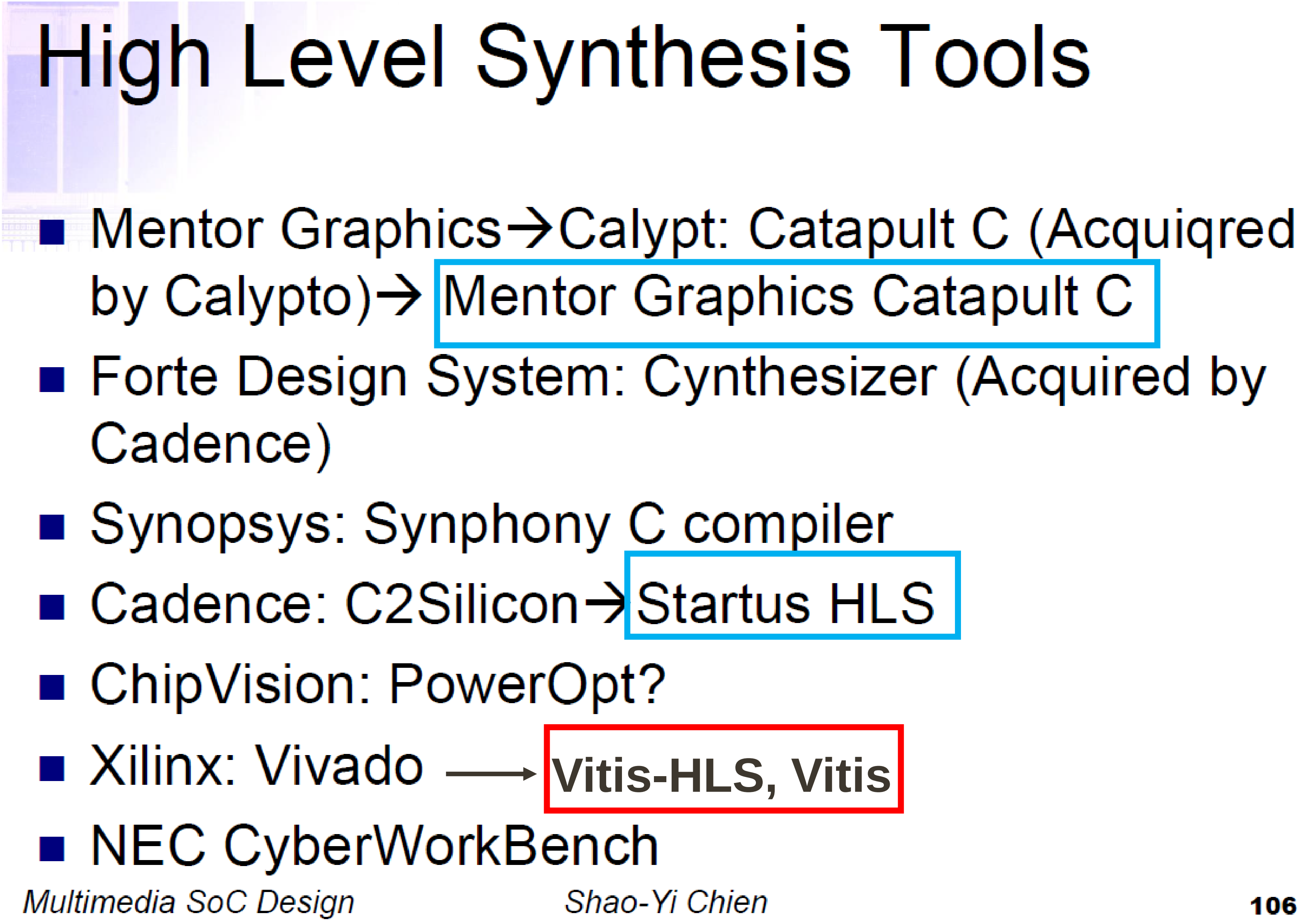


	VHDL	Verilog	SystemC	System Verilog	Ptolemy Matlab
Functional			★★		★★★
Transaction Level			★★★	★★	
RTL	★★★	★★★	★	★★★	

Multimedia SoC Design *Shao-Yi Chien* **75**

SystemC

- Not a new language
- A special class library
- Based on C++
  - Includes all the advantages/disadvantages of C++
- Good reference implementation
- C++ compatibility supports SW compatibility
- Only limited path to implementation
- TLM methodology and experience exists
- Oriented towards HDS verification, architecture exploration, and fast higher level simulation

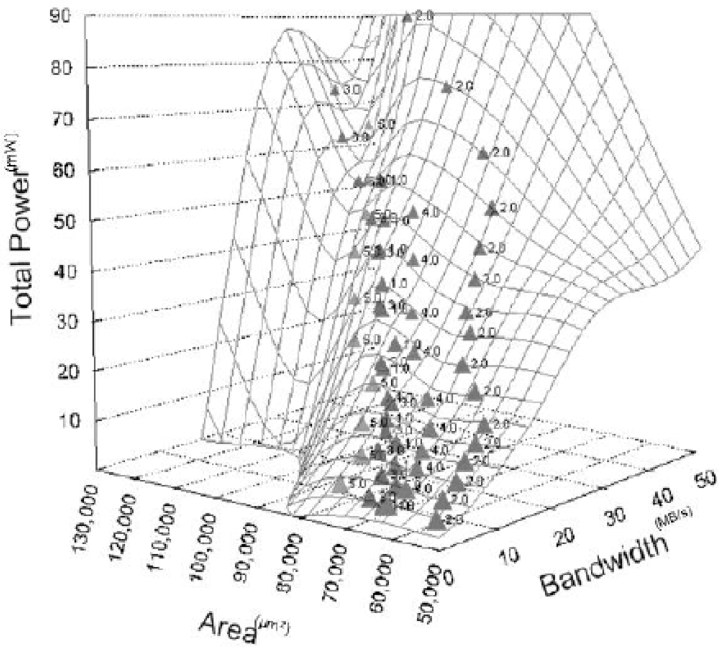


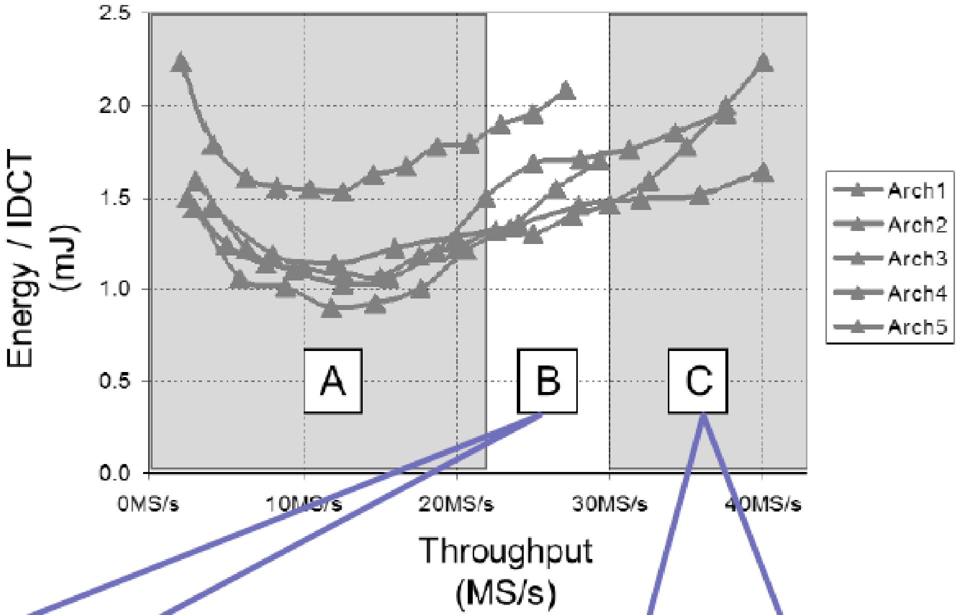
High Level Synthesis Tools

- Mentor Graphics→Calypt: Catapult C (Acquiqred by Calypto)→ Mentor Graphics Catapult C
- Forte Design System: Cynthesizer (Acquired by Cadence)
- Synopsys: Synphony C compiler
- Cadence: C2Silicon→Startus HLS
- ChipVision: PowerOpt?
- Xilinx: Vivado → **Vitis-HLS, Vitis**
- NEC CyberWorkBench

Why HLS?

- Software Defined Hardware (SDH)
- Computational Efficiency
- Design and Verification Productivity
- Improve Quality of Results (QoR)
- Fast system prototyping: ESL
- Fast architecture exploration
  - C++ ←→ python v.s. verilog ←→ HLS





Unpipelined microarchitecture taking 16-clock cycles per loop iteration

Highly pipelined microarchitecture

Outline

- Why HLS?
- HLS IP Flow
- Pragma Introduction
- Design Flow
- Labs

HLS IP Flow

0. Coding in C++

1. **C-Simulation (SW-Emulation)**
   - Check the C source code evaluation with the golden (Similar to SystemC)

2. **C-Synthesis**
   - Perform C -> RTL synthesis

3. **Co-Simulation (Hardware-Emulation)**
   - Using standard RTL verification tools

4. Generate bitstream (FPGA)
   - RTL to Gate-level synthesis + P&R for IC flow

See UG871 for details:
https://docs.xilinx.com/v/u/en-US/ug871-vivado-high-level-synthesis-tutorial

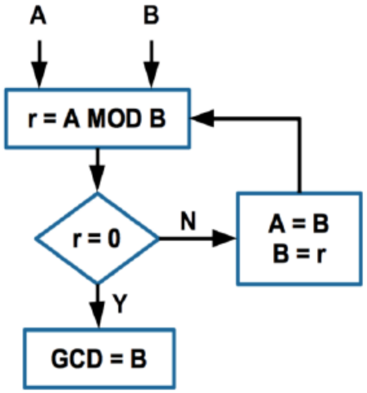
0. Coding in C++

- Coding in C++ rather than tedious RTL level
- Benefits:
  - No sequential logic bugs
  - Unified coding language
    - Design: C++
    - Verification: C++
    - Application: C++
- Disadvantages:
  - Stiff learning curve
  - RTL is still the mainstream in Digital IC Design Flow
    - FAE, customer, ….

Illustration – GCD

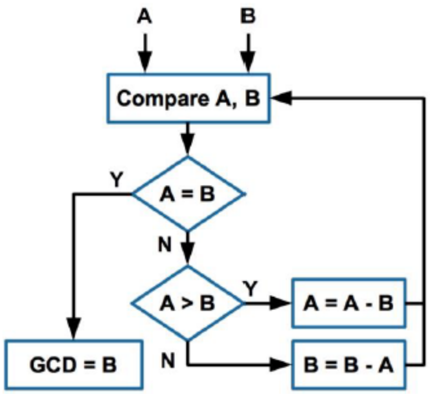
Euclidean Algorithm

$$\gcd(a,b) = \gcd(b,r)$$
$$\text{where, } a = qb + r$$



Simplified Euclidean GCD Algorithm

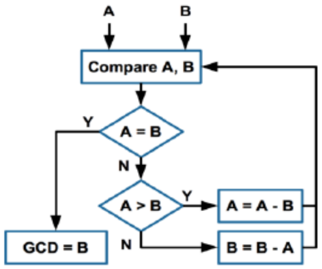
$$\gcd(a,b) = \gcd(b,(a-b))$$
$$= \gcd(a,(b-a))$$

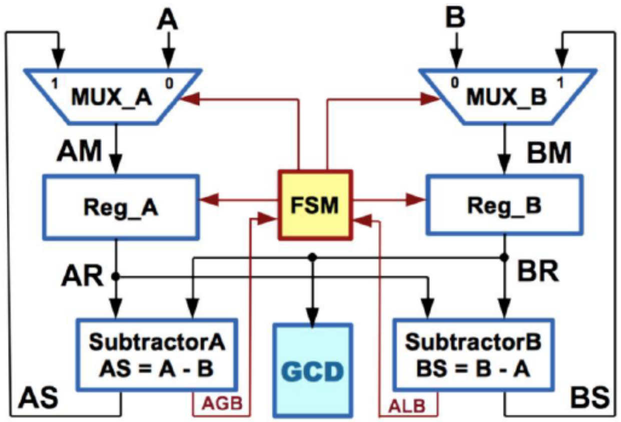


```
module gcd_behavior #(parameter width = 32)
             ( input  [width-1: 0] A_in, B_in,
               output [width-1:0] Y );
reg [width-1:0] A, B, Y, swap
Integer  done;

always @( A_in or B_in ) begin
  while ( A ! = B ) begin
    if( A > B )   A<= A – B;
    else          B<= B – A;
  end
end
Y = B;
endmodule
```

- RTL synthesis tool only copies the circuit for the while/for loop
- But the # of loop could not be determined at compiling time
- The circuit could not be synthesized
- It needs a structure implementation

Illustration – GCD (RTL)



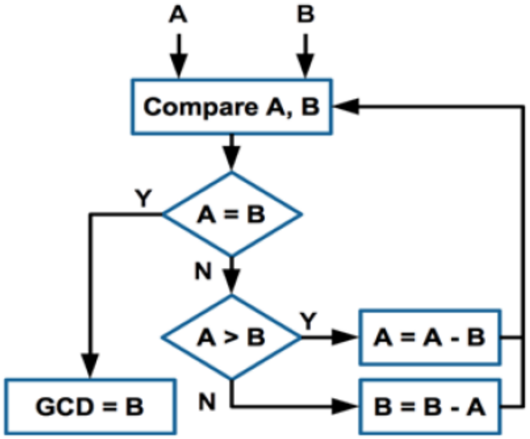


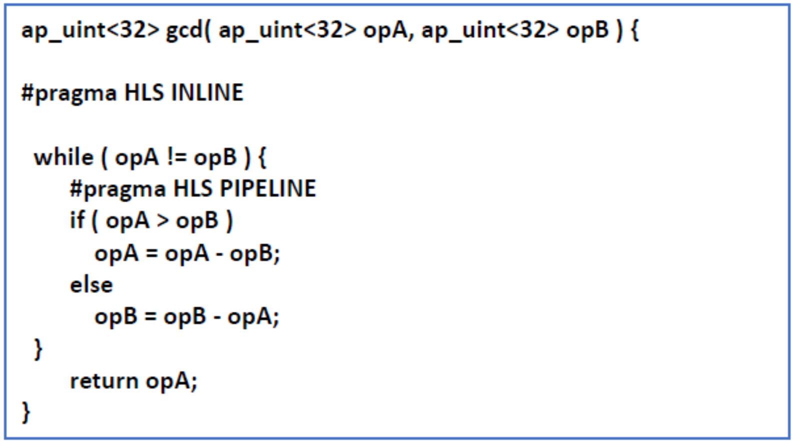
```
module gcd_fsm(
                 input  clock, reset, go,
                 input  AGB, ALB,
                 output  A_en, B_en,
                 A_mux_sel, B_mux_sel,
                 out_mux_sel, ouput done );
reg running = 0;
always @( posedge clock) begin
   if( go )    running <= 1;
   else if (done)  running <= 0;
end
reg [5:0] ctrl_sig;
assign { A_en, B_en, A_mux_sel, B_mux_sel, done }
  always @(*)  begin
     if( !running )    ctrl_sig = 5’b11_00_0;
     else if( AGB )    ctrl_sig = 5’b10_1x_0;
     else if( ALB  )   ctrl_sig = 5’b11_11_0;
     else              ctrl_sig = 5’b00_xx_1;
  end
endmodule
```

```
module gcd_datapath #(parameter width = 16)
                ( input clock,
                  input A_en, B_en, A_mux_sel, B_mux_sel,
                  out_mux_sel,
                  input [width-1:0] A_in, B_in;
                  output AGB, ALB,
                  output [width-1:0] Y; )
reg [width-1:0] A, B;
assign Y = A;

// Datapath Logic
wire [width-1:0] out =  ( out_mux_sel) ? B: A-B:;
wire [width-1:0] A_next = ( A_mux_sel ) ? out : A_in;
wire [width-1:0] B_next = ( B_mux_sel ) ? A : B_in;

// Generate output control signals
wire AGB = ( A > B);
wire ALB =  (A < B);

// edge-triggered flip-flop
always @( posedge clock) begin
   if( A_en )    A <= A_next;
   if (B_en)     B <= B_next;
end
endmodule
```

A Glimpse of High-Level-Synthesis

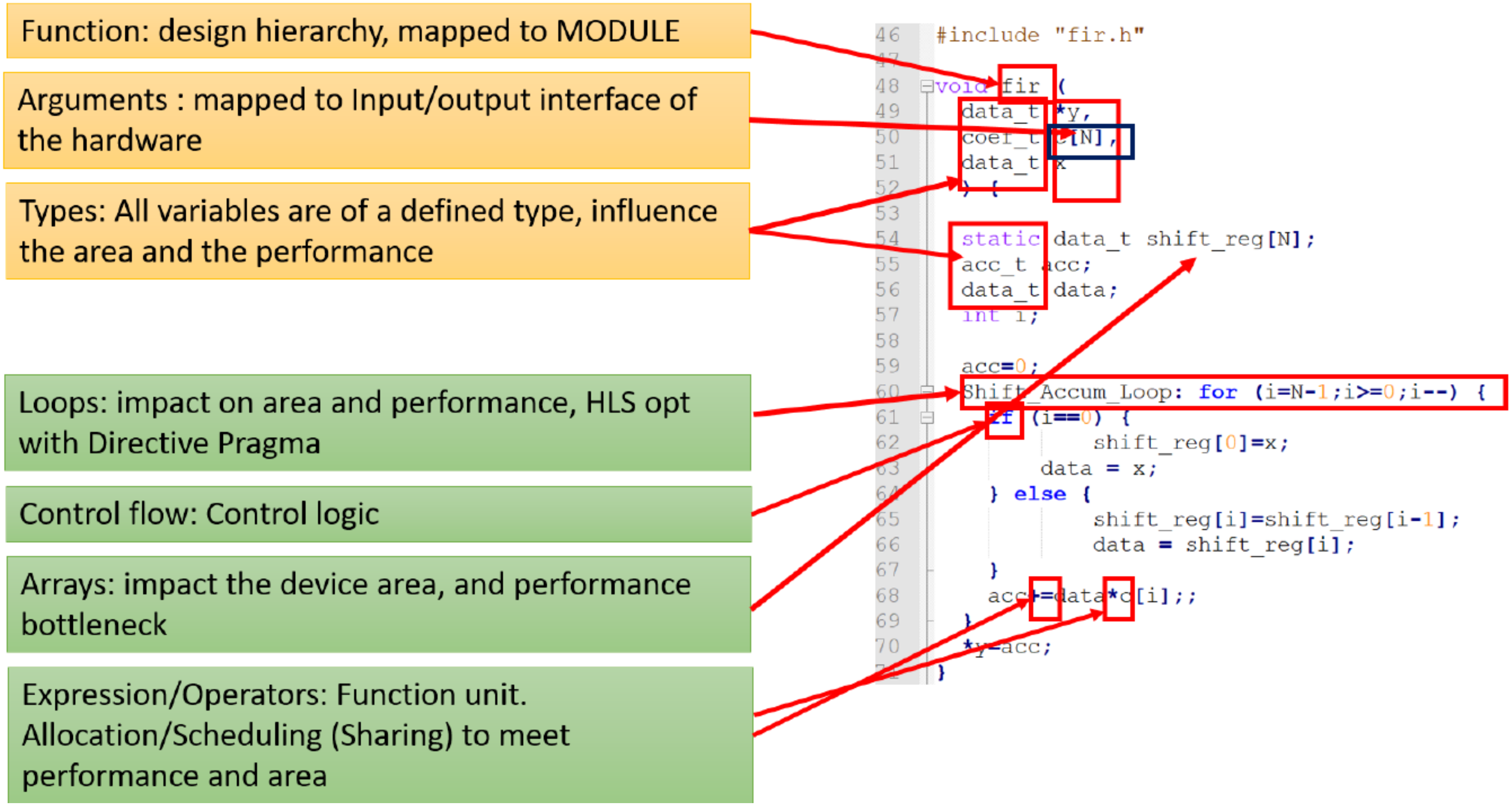
- HLS build synchronous design
  - No timing -> no clock, reset
  - No port width – imply by data type
  - Port direction – lhs, rhs
    - Input: only read, “pass by value”
    - Ouptut: function return, a reference, or a pointer
    - Inout: a reference or a pointer
- Loop:
  - Automatic control/datapath synthesis





```
ap_uint<32> gcd( ap_uint<32> opA, ap_uint<32> opB ) {

#pragma HLS INLINE

  while ( opA != opB ) {
      #pragma HLS PIPELINE
      if ( opA > opB )
         opA = opA - opB;
      else
         opB = opB - opA;
  }
      return opA;
}
```

Mapping of Key Attributes of C Code



Function: design hierarchy, mapped to MODULE

Arguments : mapped to Input/output interface of the hardware

Types: All variables are of a defined type, influence the area and the performance

Loops: impact on area and performance, HLS opt with Directive Pragma

Control flow: Control logic

Arrays: impact the device area, and performance bottleneck

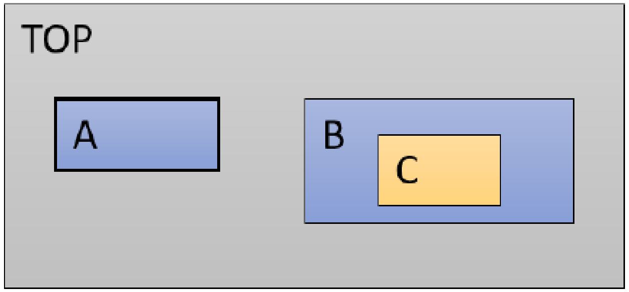
Expression/Operators: Function unit. Allocation/Scheduling (Sharing) to meet performance and area

```
#include "fir.h"

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

  static data_t shift_reg[N];
  acc_t acc;
  data_t data;
  int i;

  acc=0;
  Shift_Accum_Loop: for (i=N-1;i>=0;i--) {
     if (i==0) {
           shift_reg[0]=x;
        data = x;
     } else {
           shift_reg[i]=shift_reg[i-1];
           data = shift_reg[i];
     }
     acc+=data*c[i];;
  }
  *y=acc;
}
```

Function Hierarchy

- Top-level function becomes the top level of the RTL
- Sub-functions are synthesized into blocks in the RTL design
- Inlined to dissolve the hierarchy
    - Provide greater optimization opportunity

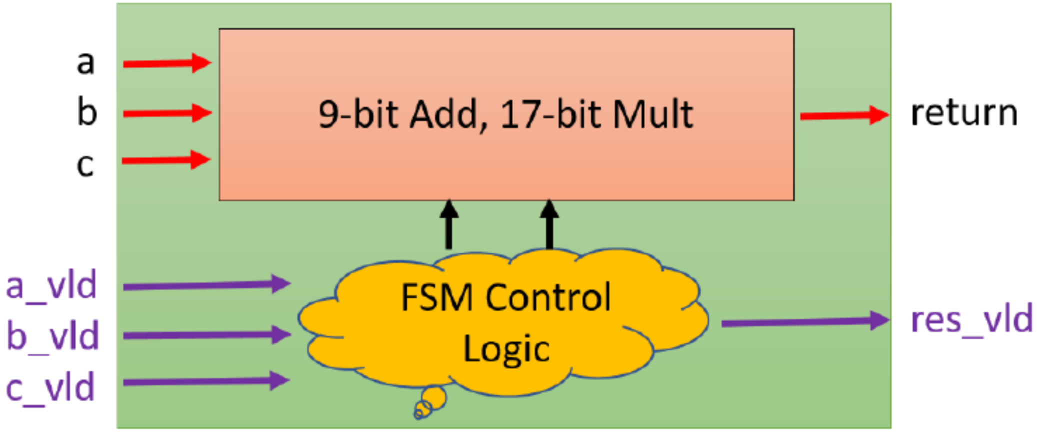
```
void A0 { ... Body A ...}
void C0 { ... Body C ...}
void B0 {   C0; }
void TOP() {
              A  ( ... )
              B  ( ... )
}
```



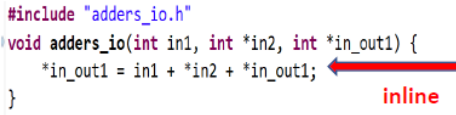
Function Arguments

- Function arguments mapped to ports on the RTL blocks
- Additional control ports are added to the design for control/synchronization among blocks
- Input/output (I/O) protocols
  - Allow automatically synchronize data exchange among blocks

```
int17  foo_top(int8* a, int8* b, int8* c, int17* ret)
{
                int sum, multi;

                sum = *a + *b;
                multi = sum * *c;
                return multi;
}
```

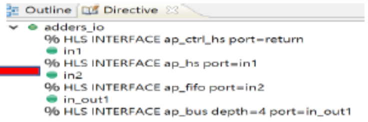


How top module connects to system

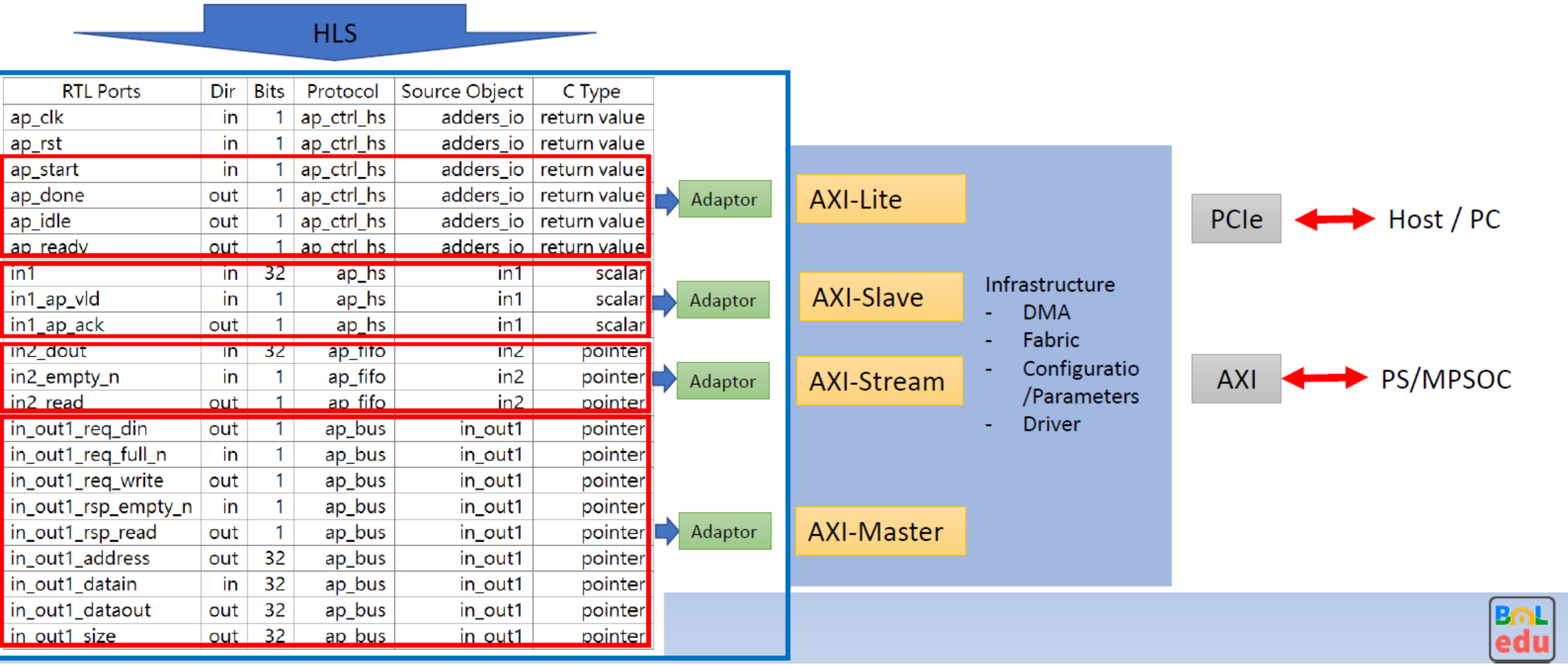


```
#include "adders_io.h"
void adders_io(int in1, int *in2, int *in_out1) {
    *in_out1 = in1 + *in2 + *in_out1;
}
```

inline



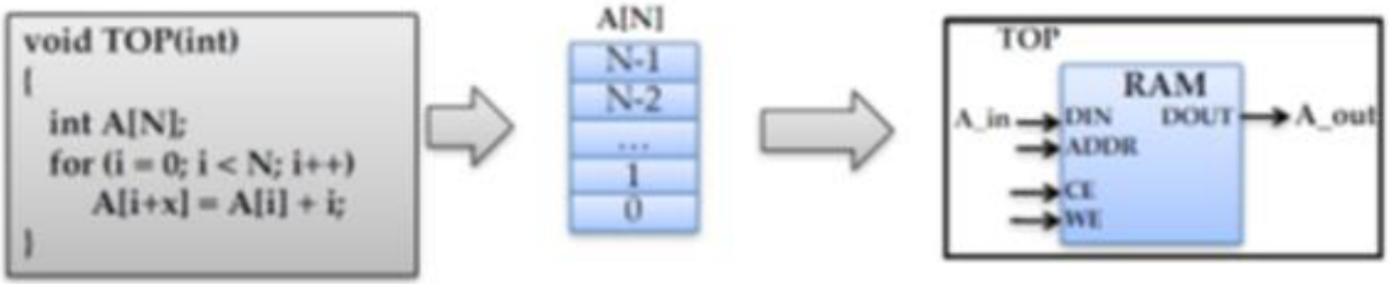
```
adders_io
  % HLS INTERFACE ap_ctrl_hs port=return
  in1
  % HLS INTERFACE ap_hs port=in1
  in2
  % HLS INTERFACE ap_fifo port=in2
  in_out1
  % HLS INTERFACE ap_bus depth=4 port=in_out1
```



RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	adders_io	return value
ap_rst	in	1	ap_ctrl_hs	adders_io	return value
ap_start	in	1	ap_ctrl_hs	adders_io	return value
ap_done	out	1	ap_ctrl_hs	adders_io	return value
ap_idle	out	1	ap_ctrl_hs	adders_io	return value
ap_ready	out	1	ap_ctrl_hs	adders_io	return value
in1	in	32	ap_hs	in1	scalar
in1_ap_vld	in	1	ap_hs	in1	scalar
in1_ap_ack	out	1	ap_hs	in1	scalar
in2_dout	in	32	ap_fifo	in2	pointer
in2_empty_n	in	1	ap_fifo	in2	pointer
in2_read	out	1	ap_fifo	in2	pointer
in_out1_req_din	out	1	ap_bus	in_out1	pointer
in_out1_req_full_n	in	1	ap_bus	in_out1	pointer
in_out1_req_write	out	1	ap_bus	in_out1	pointer
in_out1_rsp_empty_n	in	1	ap_bus	in_out1	pointer
in_out1_rsp_read	out	1	ap_bus	in_out1	pointer
in_out1_address	out	32	ap_bus	in_out1	pointer
in_out1_datain	in	32	ap_bus	in_out1	pointer
in_out1_dataout	out	32	ap_bus	in_out1	pointer
in_out1_size	out	32	ap_bus	in_out1	pointer

Arrays

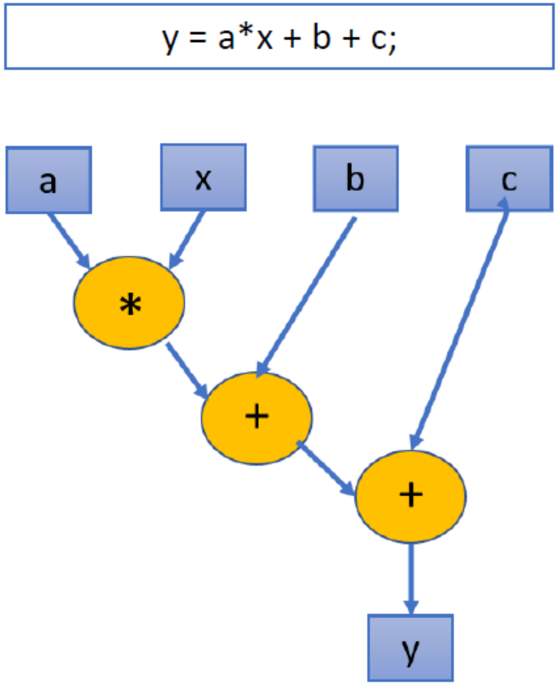
- Typically implemented by a memory block
  - Read & write array mapped to RAM
  - Constant array mapped to ROM
- By using directives
  - An array can be partitioned and map to multiple RAMs (ARRAY_PARTITIION)
  - Multiple arrays can be merged and mapped to one RAM (ARRAY_RESHAPE)
  - A array can be partitioned into individual elements and map to registers



```
void TOP(int)
{
  int A[N];
  for (i = 0; i < N; i++)
     A[i+x] = A[i] + i;
}
```

Expressions – Data Flow Graph

- Expression is translated to datapath and its control path (FSM)
- Start by analyzing the data dependencies between the various steps in the expression shown above. This analysis leads to a Data Flow Graph (DFG)



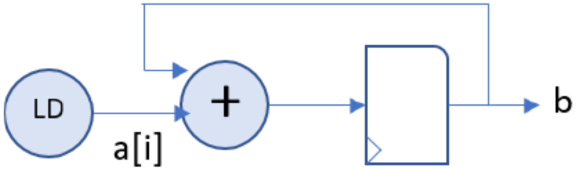
Control Flow: Loop

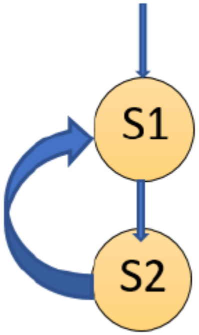
- Loops are the main area of parallelism in an algorithm
- Loops can be
  - pipelined,
  - Unrolled, Partially unrolled,
  - Merged
  - Flattened
- HLS generates the datapath and control logic

Control Flow – Rolled

- By default, loops are rolled
  - Each loop iteration corresponds to a “sequence” of states (DAG)
  - The state sequence will be repeated multiple times based on the loop trip count.
  - The resource (adder) is repeatedly used in the loop iteration.
  - Efficient use the resource, but longer latency

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





Loop - Unroll

- Rolled loops can be made unrolled or partially unrolled by

  #pragma UNROLL [factor = n]

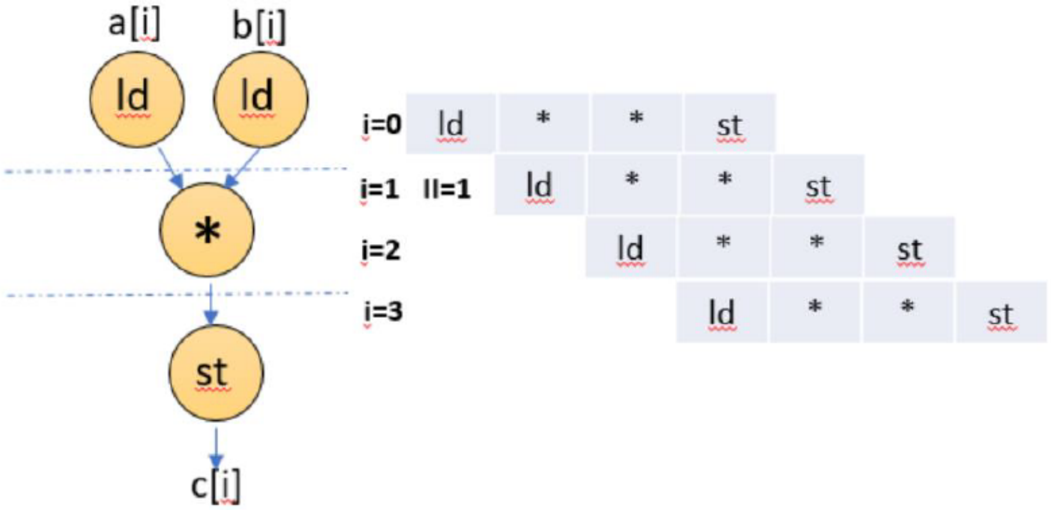
- Pros
  - Decrease loop overhead
  - Increase parallelism for scheduling
- Cons
  - Increase operator count, negatively impact area, power and timing

```
// Rolled
void TOP(...) {
   ...
   for(i = 0; i < 4; i++)
      c[i] = a[i] * b[i];
}
```

```
// Unrolled
void TOP(...) {
   ...
   c[0] = a[0] * b[0];
   c[1] = a[1] * b[1];
   c[2] = a[2] * b[2];
   c[3] = a[3] * b[3];
}
```

Loop - Pipeline

- One of the most important optimization
- Allow a new iteration to begin before the previous iteration is complete
- Key matric: Initiation Interval (II)



i=0		ld	*	*	st			
i=1	II=1		ld	*	*	st		
i=2				ld	*	*	st	
i=3					ld	*	*	st

1. C-Simulation (SW-Emulation)

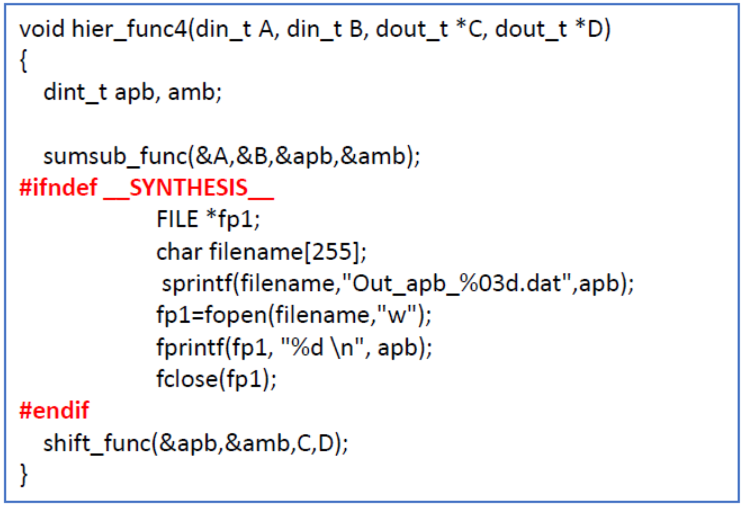
- Verify your C/C++ code with the golden

- Since the hardware is designed in C++, the testbench is also C++.

- It is just like Freshman C/C++ course.

Non-Synthesizable Code for Testbench

- The great power of HLS is the C-simulation and RTL co-simulation testbench are the same.
  - Similar to systemC ESL validation
- Use __SYNTHESIS__ for testbench code



```
void hier_func4(din_t A, din_t B, dout_t *C, dout_t *D)
{
   dint_t apb, amb;

   sumsub_func(&A,&B,&apb,&amb);
#ifndef __SYNTHESIS__
                FILE *fp1;
                char filename[255];
                 sprintf(filename,"Out_apb_%03d.dat",apb);
                fp1=fopen(filename,"w");
                fprintf(fp1, "%d \n", apb);
                fclose(fp1);
#endif
   shift_func(&apb,&amb,C,D);
}
```

2. C-Synthesis

- Analysis the C/C++ code and transform to RTL code
- Tools:
  - FPGA: Vivado-HLS (deprecated) → Vitis-HLS
  - IC: Stratus-HLS, … etc.

- Tools guaranteed the logic.

- Pragmas are needed to control its behavior
  - UNROLL factor=2
  - PIPELINE II=1
  - …

Resource Allocation, Scheduling, Binding

- **Resource allocation**: Each operation is mapped to a hardware resource, annotated with both timing and area information

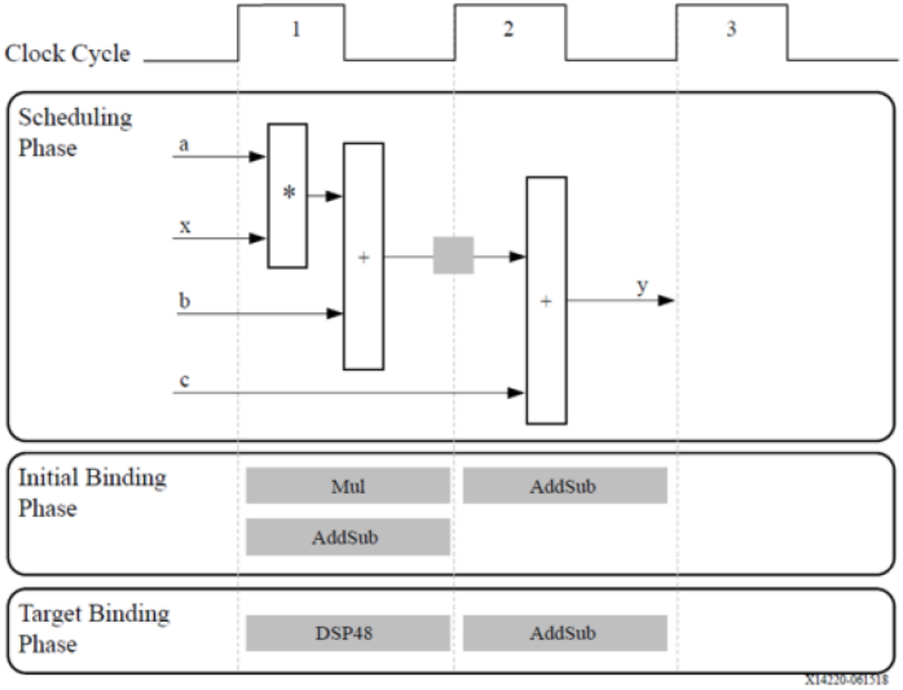
  #pragma HLS allocation operation instance = add limit = 1

- **Scheduling**: decide which clock cycle to perform what operations

- **Binding**: mapped to the hardware resource.

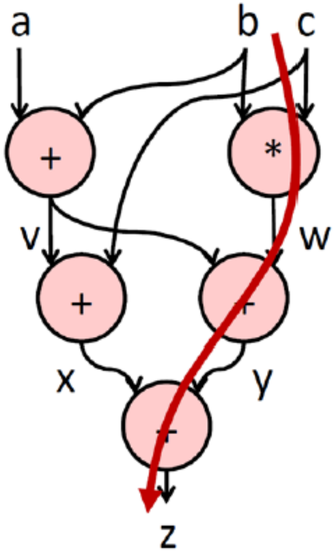
  #pragma HLS bind_op variable=<variable> op=<type> impl=<value> latency=<int>

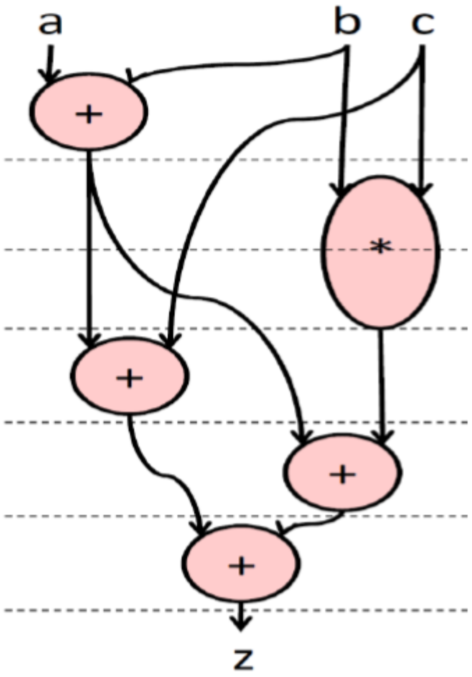
y = a*x + b + c;



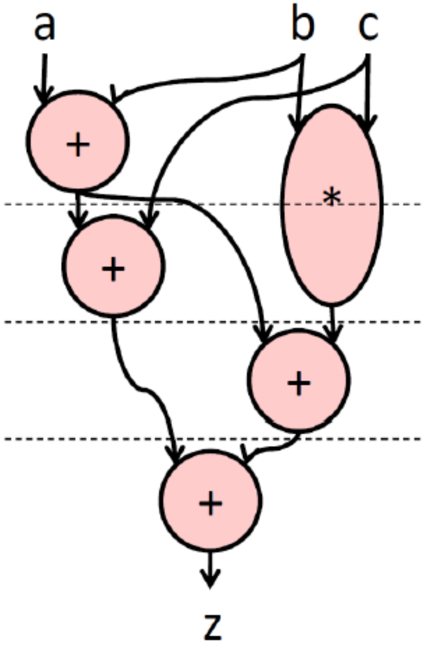
Example - Expression Datapath Resource allocation & Scheduling

```
v = a + b;
w = b * c;
x = v + c;
y = v + w;
z = x + y;
```



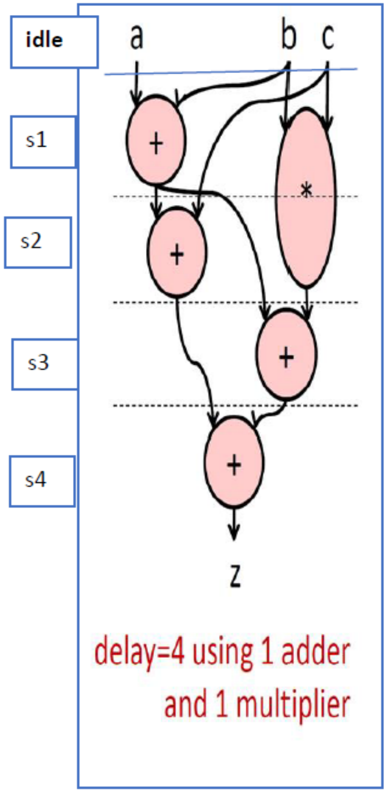


delay=6 using 1 MAC



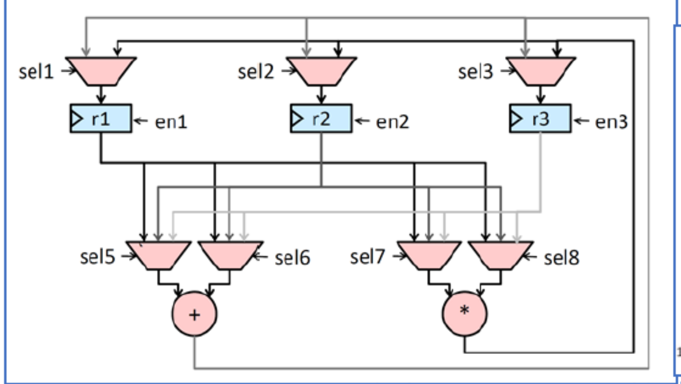
delay=4 using 1 adder and 1 multiplier

Example – Expression Datapath Binding & Resource Sharing



delay=4 using 1 adder and 1 multiplier

Generating Datapath



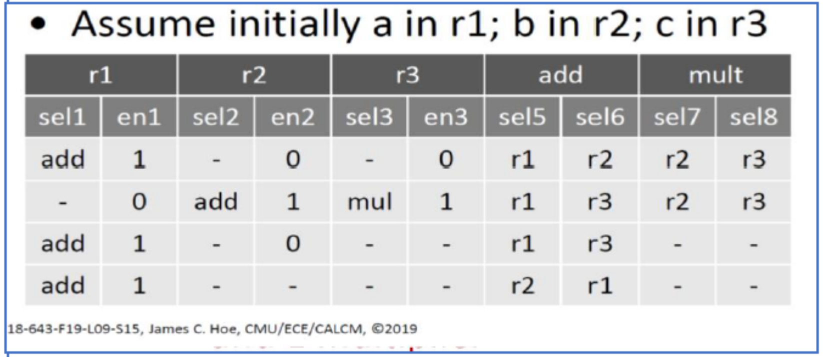
Generating State Machines

- Multiple cycles
- States: (idle, s1, s2, s3, s4
- Encode States: 000,100,101,110,111)

Generate Control Signals

- Combines the Control Signal State & State Machine, e.g.
- e.g. en1 = s1 | s3 | s4;

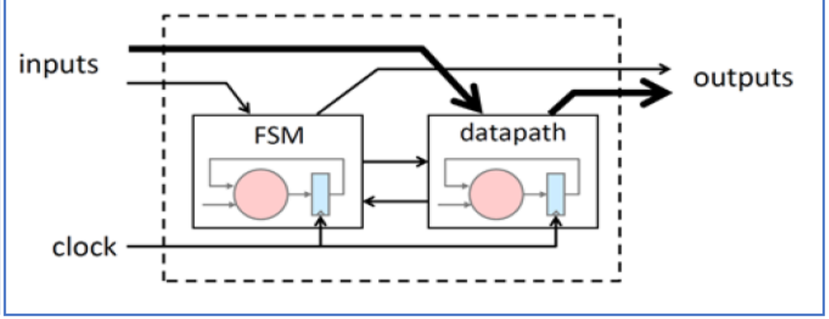
Assign State for Control Signals



- Assume initially a in r1; b in r2; c in r3

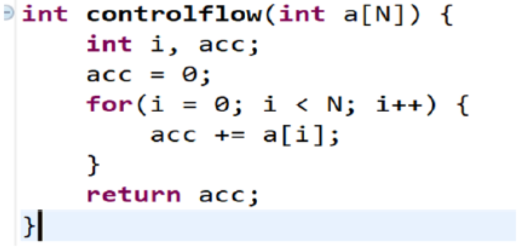
r1		r2		r3		add		mult	
sel1	en1	sel2	en2	sel3	en3	sel5	sel6	sel7	sel8
add	1	-	0	-	0	r1	r2	r2	r3
-	0	add	1	mul	1	r1	r3	r2	r3
add	1	-	0	-	-	r1	r3	-	-
add	1	-	-	-	-	r2	r1	-	-

18-643-F19-L09-S15, James C. Hoe, CMU/ECE/CALCM, ©2019



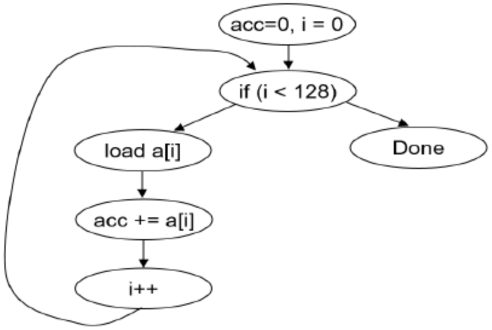
Example – Control Flow

0: Loop – add array

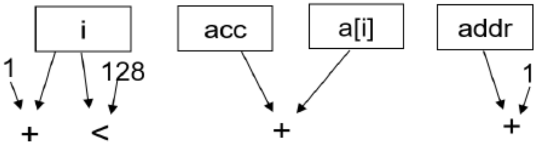


```
int controlflow(int a[N]) {
    int i, acc;
    acc = 0;
    for(i = 0; i < N; i++) {
        acc += a[i];
    }
    return acc;
}
```

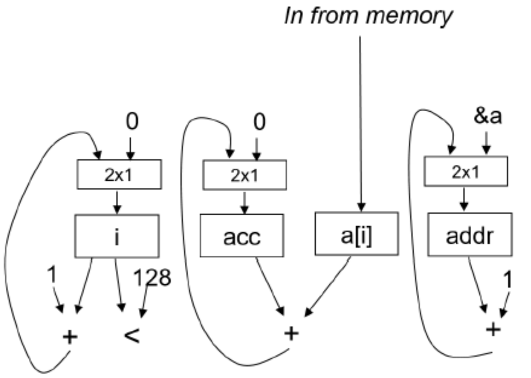
1: Decompose into states



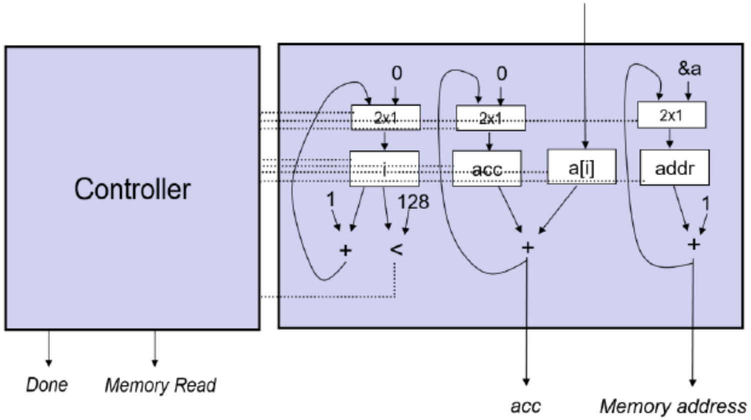
2: Identify state variables & operands



3: Determine sources of state variable input



4: add input/output
5: Generate Controller (State Machines) for datapath control



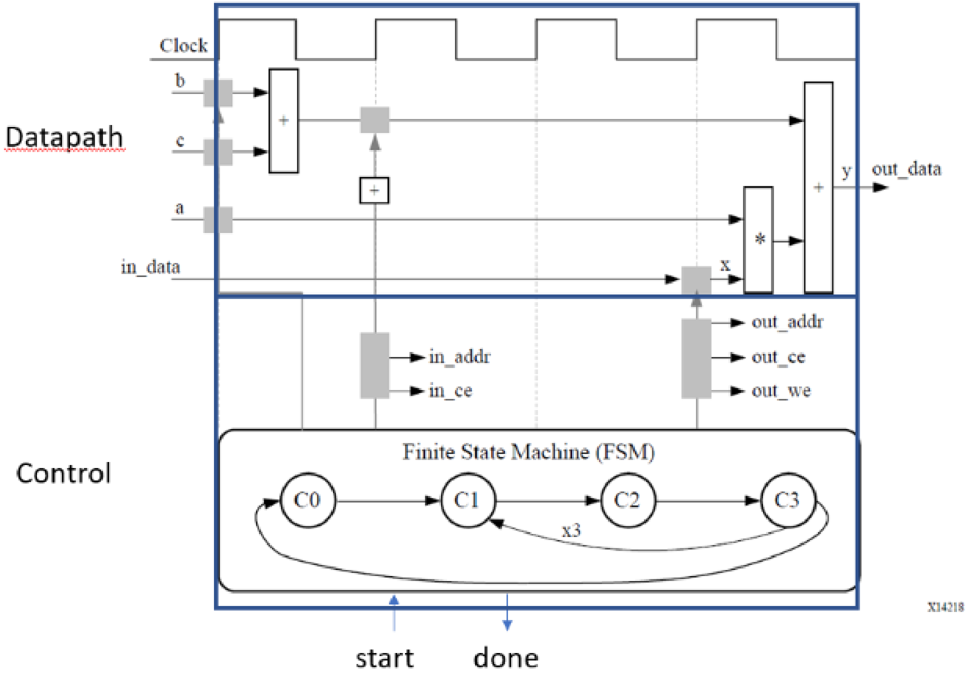
Slides from NTUEE EEE5060 Application Acceleration with High-Level-Synthesis

HLS extra the control logic in the form of a finite-state-machine, in which each of state C0, C1, C2, C3 perform the following tasks

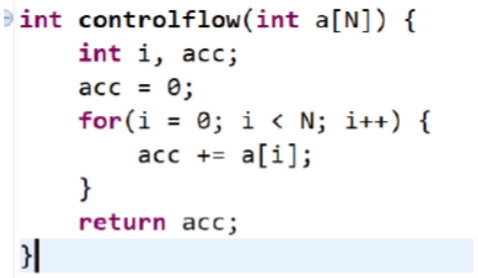
- C0: perform (b+c) and loop initiation. The result is latched at the end of the C0 state
- C1: Generate in memory access control signal, including in_addr, in_ce
- C2: wait for the RAM return in[i] data
- C3: Perform the multiplication of x*a and addition. Generate the out RAM control signals, out_addr, out_ce, out_we

The full sequence of states are: C0, {C1, C2, C3}, {C1, C2, C3}, {C1, C2, C3}, and return to C0

If directive "#pragma PIPELINE" is specified, HLS generates a pipelined datapath for the operations in the loop and its corresponding loop controller logic.

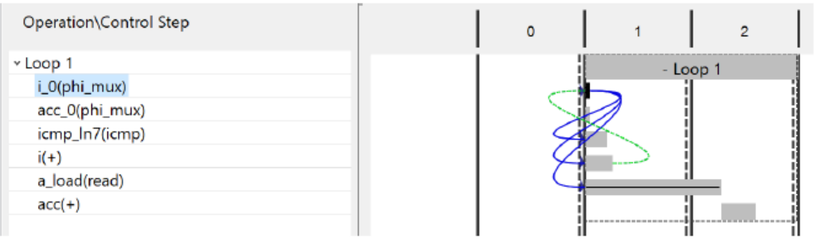


Example – Control Flow (Vivado HLS)



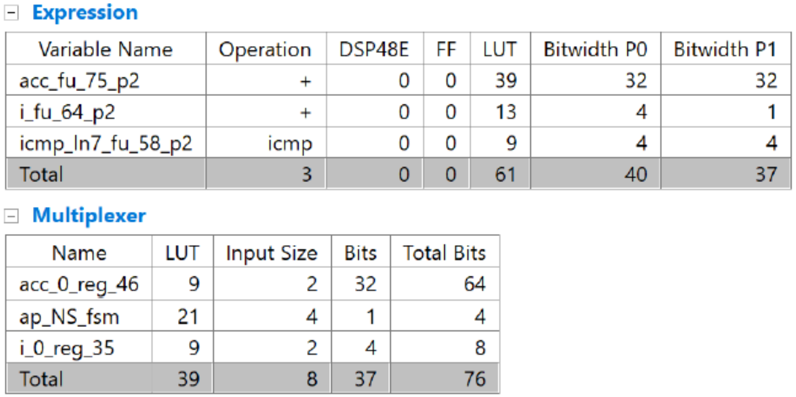
```
int controlflow(int a[N]) {
    int i, acc;
    acc = 0;
    for(i = 0; i < N; i++) {
        acc += a[i];
    }
    return acc;
}
```

Scheduling



Operation\Control Step	0	1	2
Loop 1		- Loop 1	
i_0(phi_mux)			
acc_0(phi_mux)			
icmp_ln7(icmp)			
i(+)			
a_load(read)			
acc(+)			

Resource



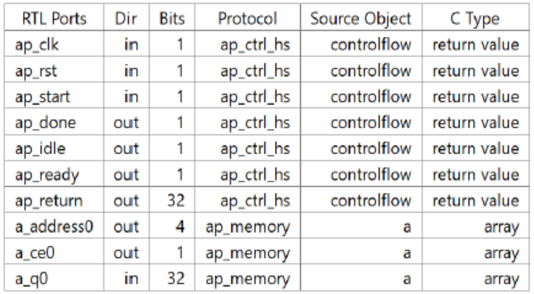
Expression

Variable Name	Operation	DSP48E	FF	LUT	Bitwidth P0	Bitwidth P1
acc_fu_75_p2	+	0	0	39	32	32
i_fu_64_p2	+	0	0	13	4	1
icmp_ln7_fu_58_p2	icmp	0	0	9	4	4
Total	3	0	0	61	40	37

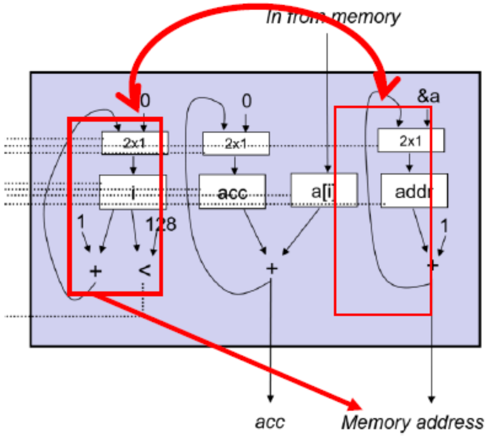
Multiplexer

Name	LUT	Input Size	Bits	Total Bits
acc_0_reg_46	9	2	32	64
ap_NS_fsm	21	4	1	4
i_0_reg_35	9	2	4	8
Total	39	8	37	76

Interface



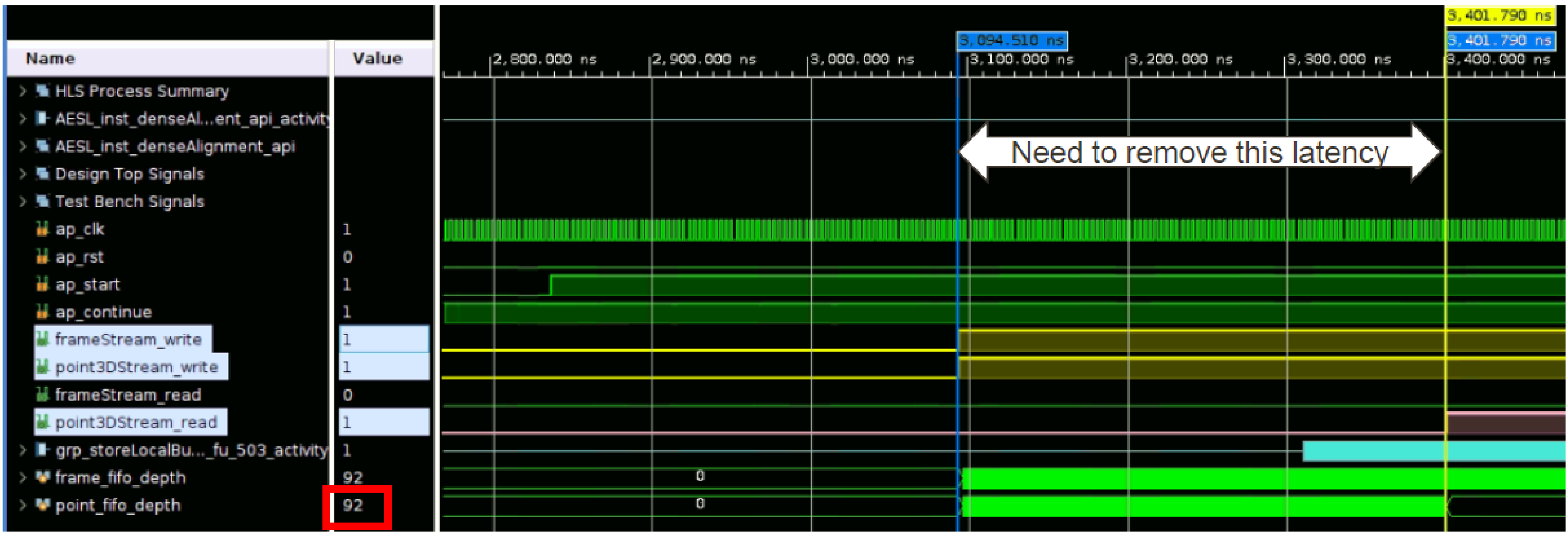
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	controlflow	return value
ap_rst	in	1	ap_ctrl_hs	controlflow	return value
ap_start	in	1	ap_ctrl_hs	controlflow	return value
ap_done	out	1	ap_ctrl_hs	controlflow	return value
ap_idle	out	1	ap_ctrl_hs	controlflow	return value
ap_ready	out	1	ap_ctrl_hs	controlflow	return value
ap_return	out	32	ap_ctrl_hs	controlflow	return value
a_address0	out	4	ap_memory	a	array
a_ce0	out	1	ap_memory	a	array
a_q0	in	32	ap_memory	a	array



- Memory address is the same as "variable i"
- **But, address of array "a" is different?**

3. Co-Simulation (Hardware-Emulation)

- Using standard RTL verification tools
- Waveform viewers
- System-level considerations
  - E.g. FIFOs, deadlocks, … etc.



Name	Value
HLS Process Summary	
AESL_inst_denseAl...ent_api_activity	
AESL_inst_denseAlignment_api	
Design Top Signals	
Test Bench Signals	
ap_clk	1
ap_rst	0
ap_start	1
ap_continue	1
frameStream_write	1
point3DStream_write	1
frameStream_read	0
point3DStream_read	1
grp_storeLocalBu..._fu_503_activity	1
frame_fifo_depth	92
point_fifo_depth	92

Need to remove this latency

4.Generate bitstream (FPGA)

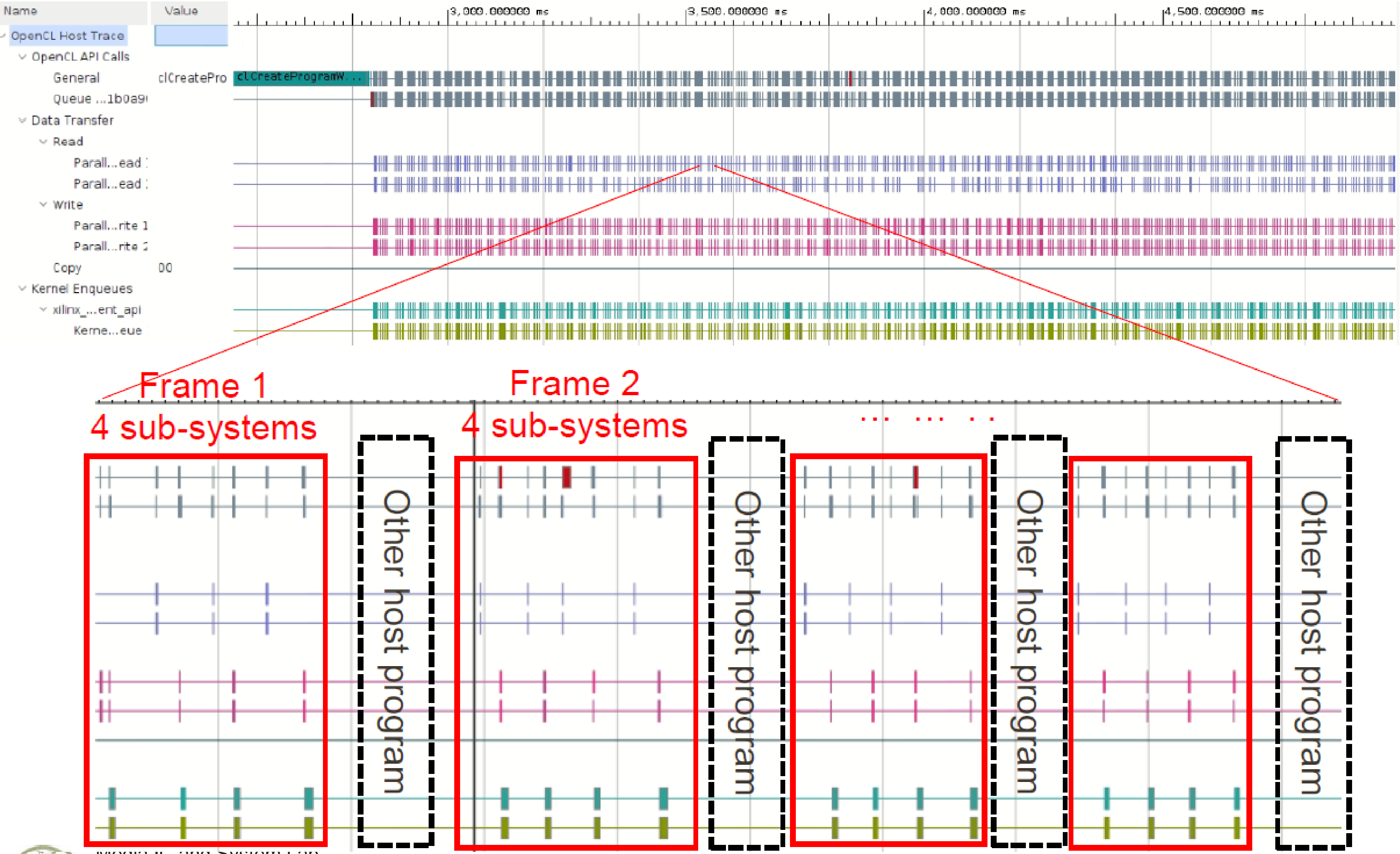
- Tools: Vivado

- Automation in FPGA tools without clicks
- Configure the synthesis and placement via FPGA .tcl
- This step takes around 1~2 hr

- ~~Post layout verification~~ → Run on FPGA

Application Timeline (1/2)

100 frames testbench



Frame 1

4 sub-systems

Other host program

Frame 2

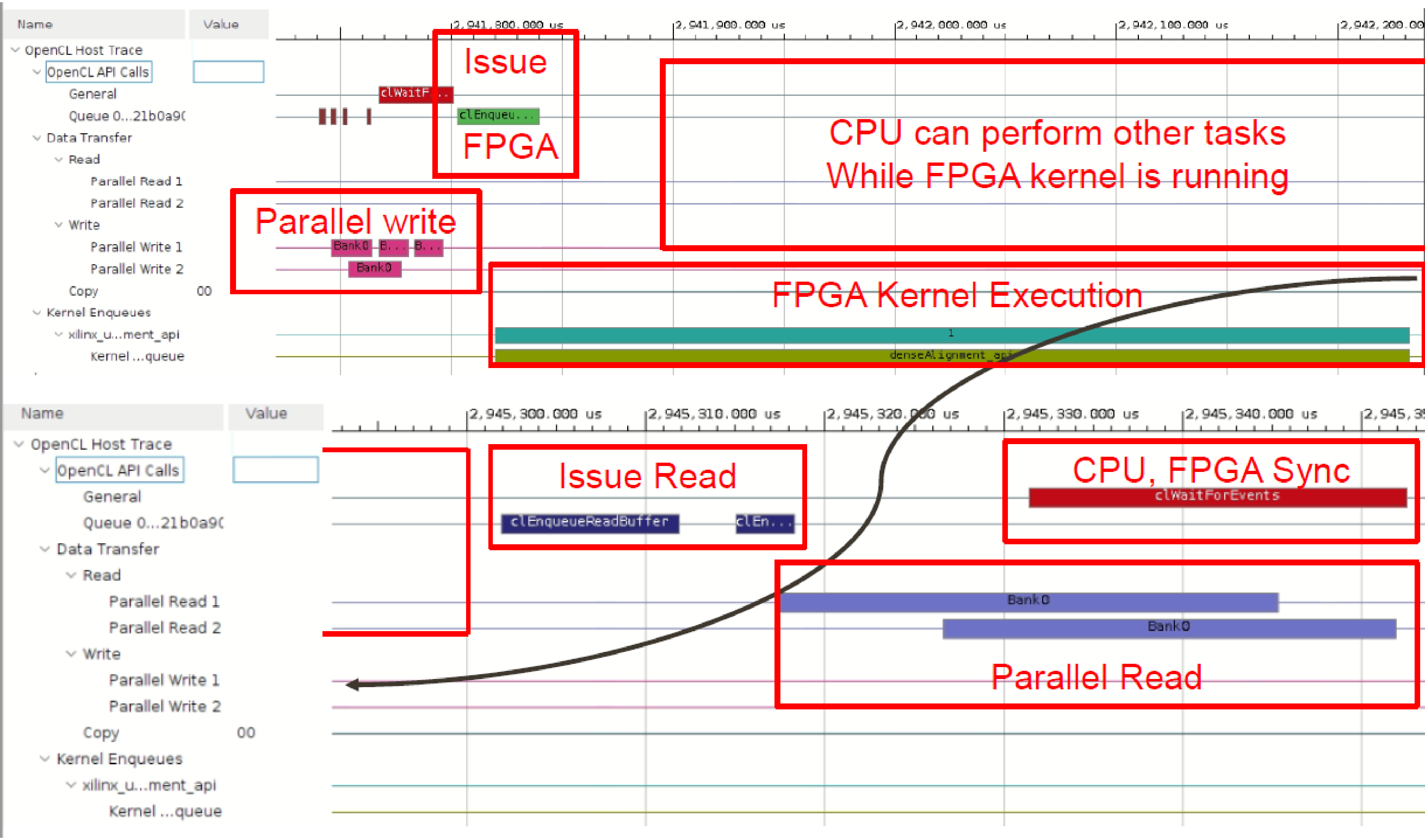
4 sub-systems

Other host program

… … . .

Other host program

Other host program



Issue

FPGA

CPU can perform other tasks
While FPGA kernel is running

Parallel write

FPGA Kernel Execution

Issue Read

CPU, FPGA Sync

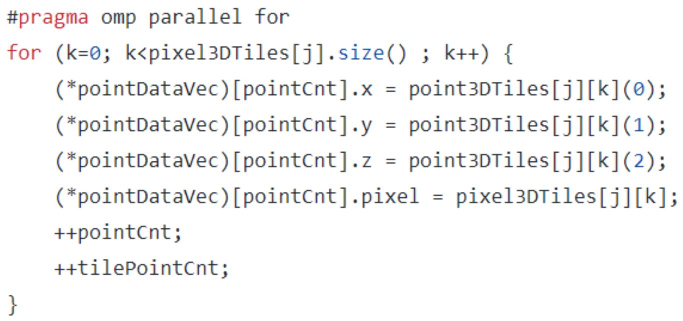
Parallel Read

Outline

- Why HLS?
- HLS IP Flow
- Pragma Introduction
- Design Flow
- Labs

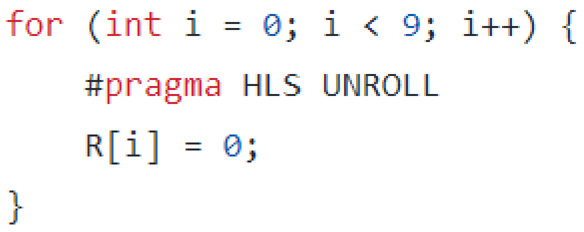
#Pragma Introduction

- Special purpose directive for turning on or off some compiler-specific features.
- Example: OpenMP
  - Multi-thread programming

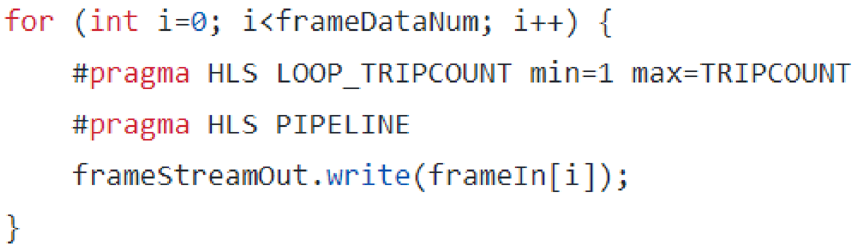


```
#pragma omp parallel for
for (k=0; k<pixel3DTiles[j].size() ; k++) {
    (*pointDataVec)[pointCnt].x = point3DTiles[j][k](0);
    (*pointDataVec)[pointCnt].y = point3DTiles[j][k](1);
    (*pointDataVec)[pointCnt].z = point3DTiles[j][k](2);
    (*pointDataVec)[pointCnt].pixel = pixel3DTiles[j][k];
    ++pointCnt;
    ++tilePointCnt;
}
```

- Example: HLS
  - Unroll, pipeline, …..



```
for (int i = 0; i < 9; i++) {
    #pragma HLS UNROLL
    R[i] = 0;
}
```



```
for (int i=0; i<frameDataNum; i++) {
    #pragma HLS LOOP_TRIPCOUNT min=1 max=TRIPCOUNT
    #pragma HLS PIPELINE
    frameStreamOut.write(frameIn[i]);
}
```

#Pragma Introduction

- Interface Synthesis pragma HLS interface
- Task-level Pipeline pragma HLS dataflow, pragma HLS stream
- Pipeline pragma HLS pipeline
- Loop Unrolling pragma HLS unroll, pragma HLS dependence
- Array Optimization pragma HLS array_partition, pragma HLS array_reshape
- Resource Optimization pragma HLS allocation, pragma HLS function_instantiate
- Others
  - https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas

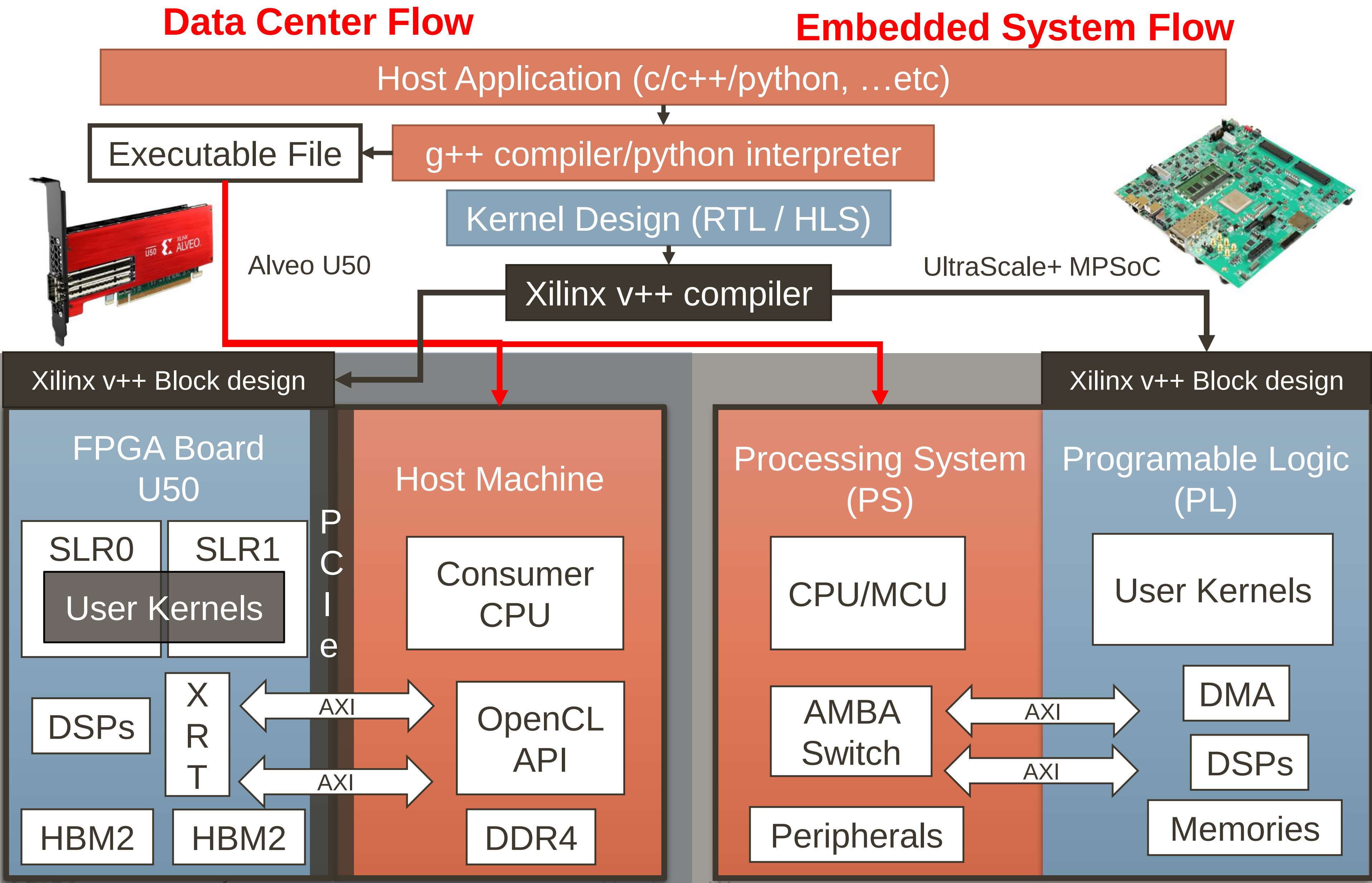
Outline

- Why HLS?
- HLS IP Flow
- Pragma Introduction
- Design Flow
- Labs

Design Flow

1. Platform select
   - Data center flow
   - Embedded system flow
2. Develop software algorithm
3. Software profile
4. Set Acceleration Goal
5. Applicability of the Hardware
6. Hardware Architecture Plan
7. HLS coding

1. Platform select



Data Center Flow

Embedded System Flow

Host Application (c/c++/python, …etc)

Executable File

g++ compiler/python interpreter

Kernel Design (RTL / HLS)

Alveo U50

UltraScale+ MPSoC

Xilinx v++ compiler

Xilinx v++ Block design

FPGA Board U50

SLR0

SLR1

User Kernels

DSPs

XRT

HBM2

HBM2

PCIe

AXI

AXI

Host Machine

Consumer CPU

OpenCL API

DDR4

Xilinx v++ Block design

Processing System (PS)

CPU/MCU

AMBA Switch

Peripherals

AXI

AXI

Programable Logic (PL)

User Kernels

DMA

DSPs

Memories

2. Develop Software Algorithm

- C++ is a better choice for HLS development flow
- Python or other language is okay, but need to translate to C++ for HLS hardware synthesis
  - Rewrite the code in C/C++
  - Cython or other transforms may/may-not help
- Pure C++ code is the simplest case
  - If function calls deeper API, then need to ensure the code in API is synthesizable
- Other examples: FINN (Python)

3. Software profile: Identify the function to be accelerated

Number of subsystems	Background Rendering	GUI	Pose Estimation	Pose Refinement	Model Rendering	Swap Window	Total Time	Avg GN Iter #	Avg LS Iter #
4	2.75	0.53	6.51	**10.43**	0.31	7.60	28.51	3.16	0.17

- You can use timers such as std::chrono
  - https://en.cppreference.com/w/cpp/chrono

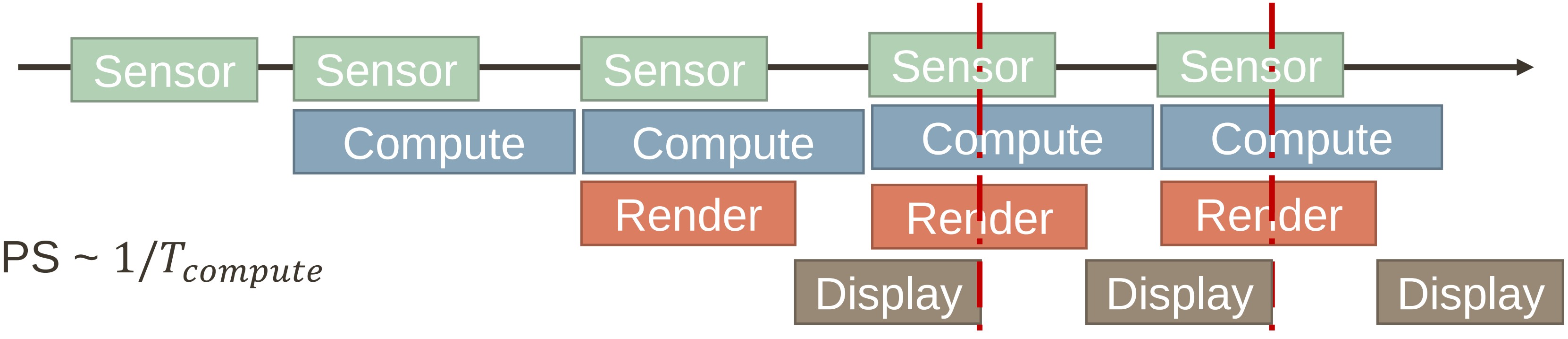
- The platform and the underlying computational cores matters a lot
  - High-end CPU, low-end CPU, MCU, GPU, ……

4. Set Acceleration Goal

- Set your goal
- What is the assumption of this goal, under what scenario?
- Example:

Acceleration Goals: Frame latency ≤ 2.5 ms

- Assuming surgeon head motion → 20 deg/sec
- Assuming **4** subsystems → 1 surgical target + 3 surgical instruments
- Assuming pipelined sense-compute-render-display system
- Assuming **bottleneck** of the pipeline bounded by **compute core**
- **Current software application latency → 10 ms**



FPS ~ $1/T_{compute}$

Period ~ $T_{compute}$

4. Set Acceleration Goal

Number of subsystems	Background Rendering	GUI	Pose Estimation	Pose Refinement	Model Rendering	Swap Window	Total Time	Avg GN Iter #	Avg LS Iter #
4	2.75	0.53	6.51	**10.43**	0.31	7.60	28.51	3.16	0.17

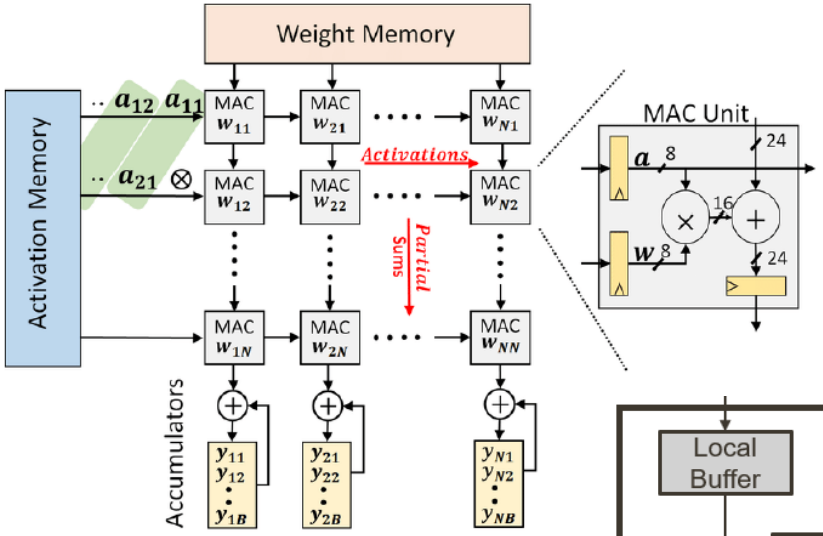
- Is this goal competitive?
  - x4 times faster, not too impressive (Impressive -> 2 ~ 3 orders)
- Achievable ?
- **Roughly estimate the cycles needed**
  - What's the time complexity of the function? How much degree of parallelism can be achieved to reach this goal? (Resource enough?)
- **Determine if it is PCIe-bound (Data center flow)**
  - 800x800x3 + 10000 x (1 + 3 x 4) byte @ 33us = **57.84 GB/s**
  - U50 PCIe bandwidth (Host -> PCIe -> FPGA) Read(Write): **11GB/s** -> ***PCIe-bounded***

5. Applicability of the Hardware

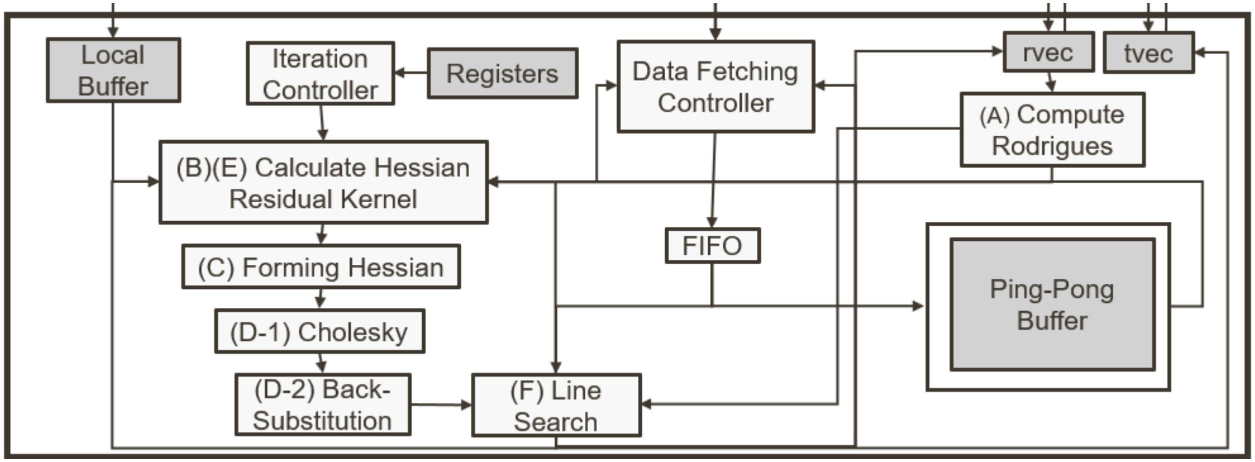
- How general is your hardware?
  - ASIC-like?
  - DSP-like?
- Example (ASIC-like)
- *Applicable to most **D**irect **D**ense **P**hotometric **R**efinement problems (**DDPR**)*
  - *Not restricted to planar or marker objects → General 3D rigid objects*
  - *Suits the front-end refinement of **V**isual **O**dometry (VO) if depth provided*
- Example (DSP-like)
  - Custom ISA + common processing units

6. Hardware Architecture Plan

- What is the possible compute architecture? dataflow?
- E.g. Systolic array, dedicated dataflow?



※ Separate data movement (IO) from Computation (PE)



Design Flow

1. Platform select
   - Data center flow
   - Embedded system flow
2. Develop software algorithm
3. Software profile
4. Set Acceleration Goal
5. Applicability of the Hardware
6. Hardware Architecture Plan
7. HLS coding

HLS Code Prerequisites

- No unsupported data type (Due to dynamic allocating)
  - Std::vector, new (malloc), *pointers,

- No unsupported relative high-level-functions
  - E.g. Eigen, *OpenCV, Open3D, ….. Most of the libraries.

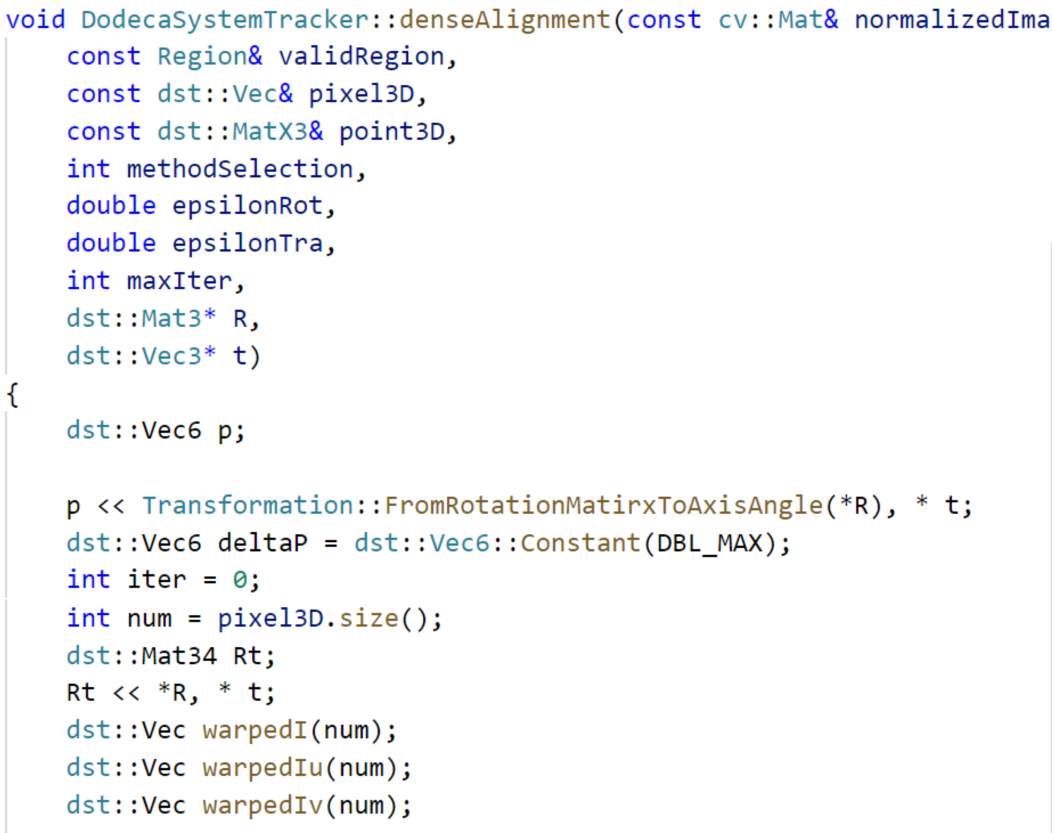
(※ Prevent RTL-like or hardware unfriendly coding style during design)

Original Host Code (1/3)

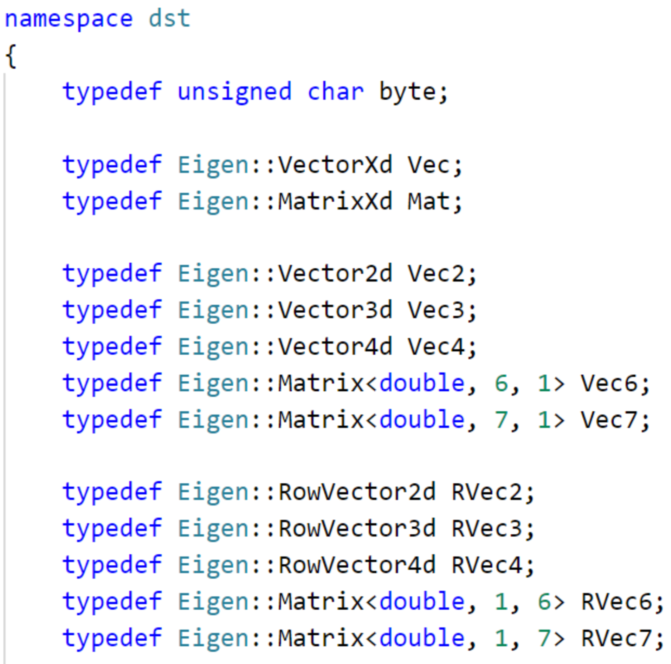
1. Not supported data type (Due to dynamic allocating)

(**Eigen** -> C++ template library for linear algebra, equivalent to numpy in python)

Need to rewrite in array or pointer!



```
void DodecaSystemTracker::denseAlignment(const cv::Mat& normalizedImage,
    const Region& validRegion,
    const dst::Vec& pixel3D,
    const dst::MatX3& point3D,
    int methodSelection,
    double epsilonRot,
    double epsilonTra,
    int maxIter,
    dst::Mat3* R,
    dst::Vec3* t)
{
    dst::Vec6 p;

    p << Transformation::FromRotationMatirxToAxisAngle(*R), * t;
    dst::Vec6 deltaP = dst::Vec6::Constant(DBL_MAX);
    int iter = 0;
    int num = pixel3D.size();
    dst::Mat34 Rt;
    Rt << *R, * t;
    dst::Vec warpedI(num);
    dst::Vec warpedIu(num);
    dst::Vec warpedIv(num);
```



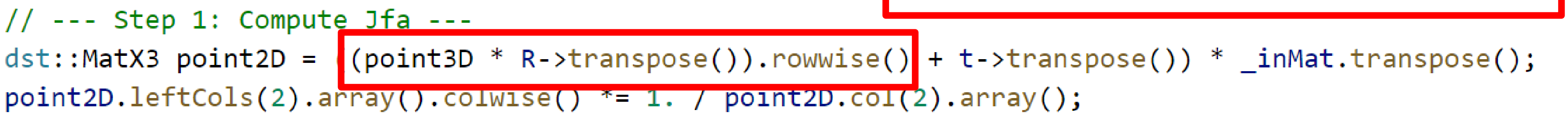
```
namespace dst
{
    typedef unsigned char byte;

    typedef Eigen::VectorXd Vec;
    typedef Eigen::MatrixXd Mat;

    typedef Eigen::Vector2d Vec2;
    typedef Eigen::Vector3d Vec3;
    typedef Eigen::Vector4d Vec4;
    typedef Eigen::Matrix<double, 6, 1> Vec6;
    typedef Eigen::Matrix<double, 7, 1> Vec7;

    typedef Eigen::RowVector2d RVec2;
    typedef Eigen::RowVector3d RVec3;
    typedef Eigen::RowVector4d RVec4;
    typedef Eigen::Matrix<double, 1, 6> RVec6;
    typedef Eigen::Matrix<double, 1, 7> RVec7;
```

Original Host Code (2/3)

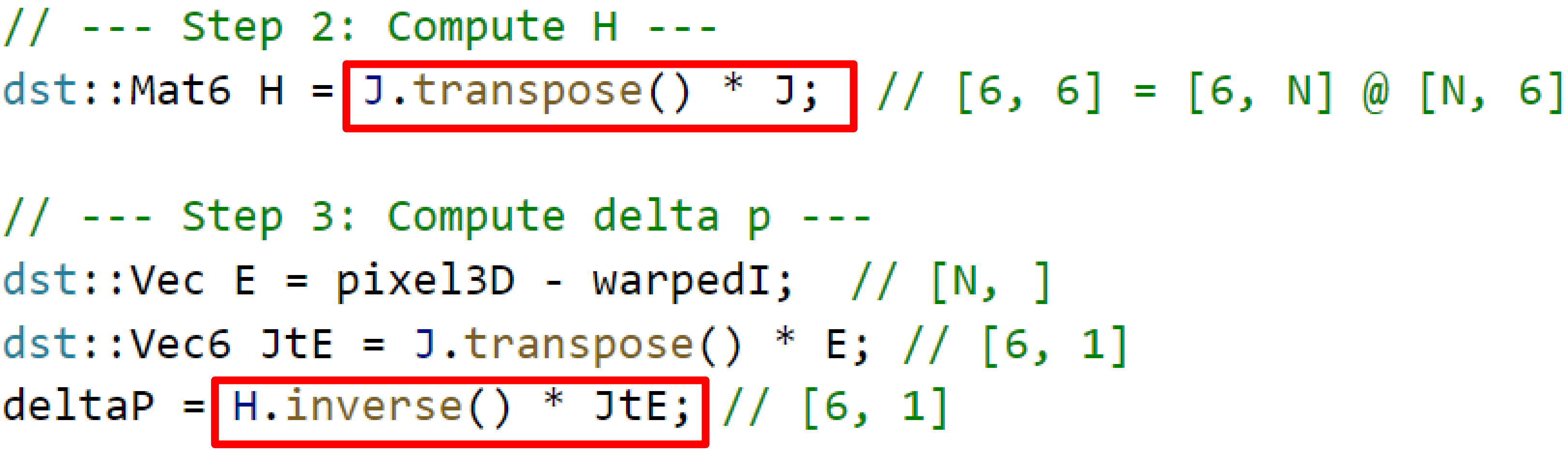
2. Not supported relative high-level-functions

(Eigen class array-wise operation)

Need to rewrite the detail implementations!



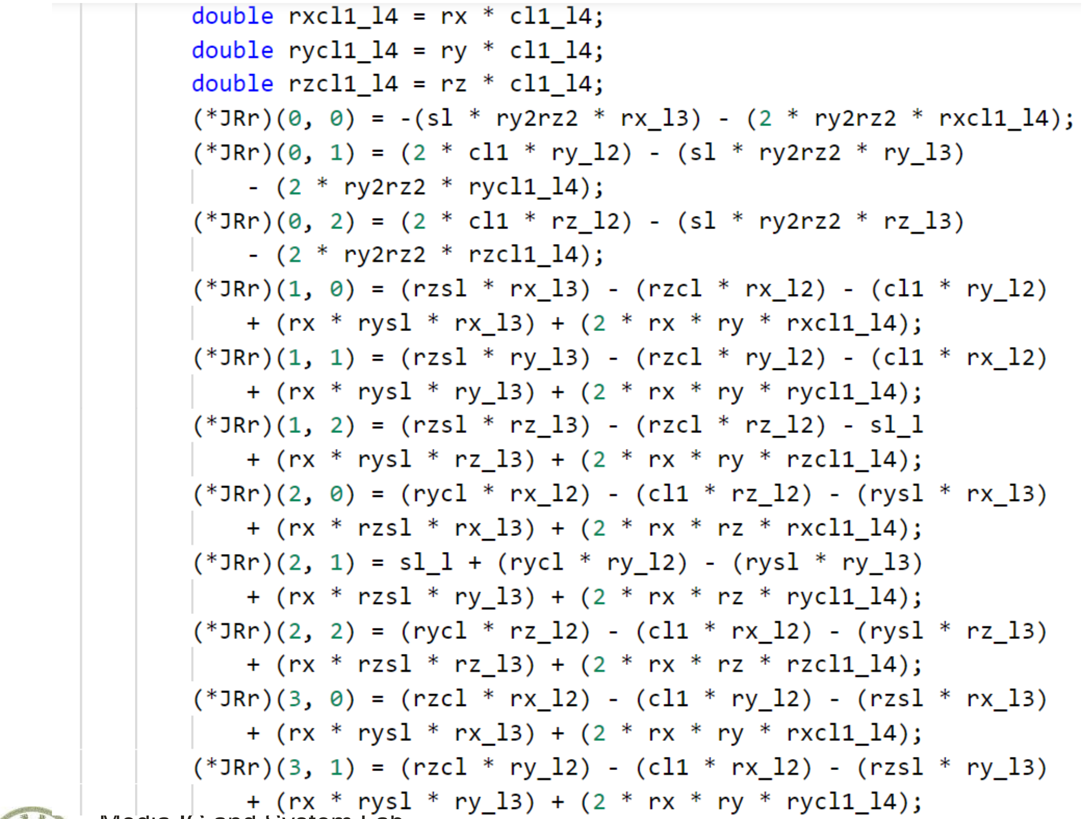
```
// --- Step 1: Compute Jfa ---
dst::MatX3 point2D = ((point3D * R->transpose()).rowwise() + t->transpose()) * _inMat.transpose();
point2D.leftCols(2).array().colwise() *= 1. / point2D.col(2).array();
```



```
// --- Step 2: Compute H ---
dst::Mat6 H = J.transpose() * J; // [6, 6] = [6, N] @ [N, 6]

// --- Step 3: Compute delta p ---
dst::Vec E = pixel3D - warpedI;  // [N, ]
dst::Vec6 JtE = J.transpose() * E; // [6, 1]
deltaP = H.inverse() * JtE; // [6, 1]
```

Original Host Code (3/3)

(※ Prevent RTL-like or hardware unfriendly coding style during design)



```
double rxcl1_l4 = rx * cl1_l4;
double rycl1_l4 = ry * cl1_l4;
double rzcl1_l4 = rz * cl1_l4;
(*JRr)(0, 0) = -(sl * ry2rz2 * rx_l3) - (2 * ry2rz2 * rxcl1_l4);
(*JRr)(0, 1) = (2 * cl1 * ry_l2) - (sl * ry2rz2 * ry_l3)
    - (2 * ry2rz2 * rycl1_l4);
(*JRr)(0, 2) = (2 * cl1 * rz_l2) - (sl * ry2rz2 * rz_l3)
    - (2 * ry2rz2 * rzcl1_l4);
(*JRr)(1, 0) = (rzsl * rx_l3) - (rzcl * rx_l2) - (cl1 * ry_l2)
    + (rx * rysl * rx_l3) + (2 * rx * ry * rxcl1_l4);
(*JRr)(1, 1) = (rzsl * ry_l3) - (rzcl * ry_l2) - (cl1 * rx_l2)
    + (rx * rysl * ry_l3) + (2 * rx * ry * rycl1_l4);
(*JRr)(1, 2) = (rzsl * rz_l3) - (rzcl * rz_l2) - sl_l
    + (rx * rysl * rz_l3) + (2 * rx * ry * rzcl1_l4);
(*JRr)(2, 0) = (rycl * rx_l2) - (cl1 * rz_l2) - (rysl * rx_l3)
    + (rx * rzsl * rx_l3) + (2 * rx * rz * rxcl1_l4);
(*JRr)(2, 1) = sl_l + (rycl * ry_l2) - (rysl * ry_l3)
    + (rx * rzsl * ry_l3) + (2 * rx * rz * rycl1_l4);
(*JRr)(2, 2) = (rycl * rz_l2) - (cl1 * rx_l2) - (rysl * rz_l3)
    + (rx * rzsl * rz_l3) + (2 * rx * rz * rzcl1_l4);
(*JRr)(3, 0) = (rzcl * rx_l2) - (cl1 * ry_l2) - (rzsl * rx_l3)
    + (rx * rysl * rx_l3) + (2 * rx * ry * rxcl1_l4);
(*JRr)(3, 1) = (rzcl * ry_l2) - (cl1 * rx_l2) - (rzsl * ry_l3)
    + (rx * rysl * ry_l3) + (2 * rx * ry * rycl1_l4);
```

Use for loop instead to gain the benefit of HLS

Then the 4 steps…

- C-Simulation
- C-Synthesis
- Co-Simulation
- Generate Bitstream

Outline

- Why HLS?
- HLS IP Flow
- Pragma Introduction
- Design Flow
- Labs

Lab Introduction

- Design files from NTUEE EEE5060 Application Acceleration with High-Level-Synthesis

- Lab1, Lab2: Embedded system flow
  - Vitis-hls, Vivado
  - MPSoC FPGAs: Pynq, Ultrascale+

- Lab3: Data center flow
  - Vitis
  - Data Center FPGAs: Alveo