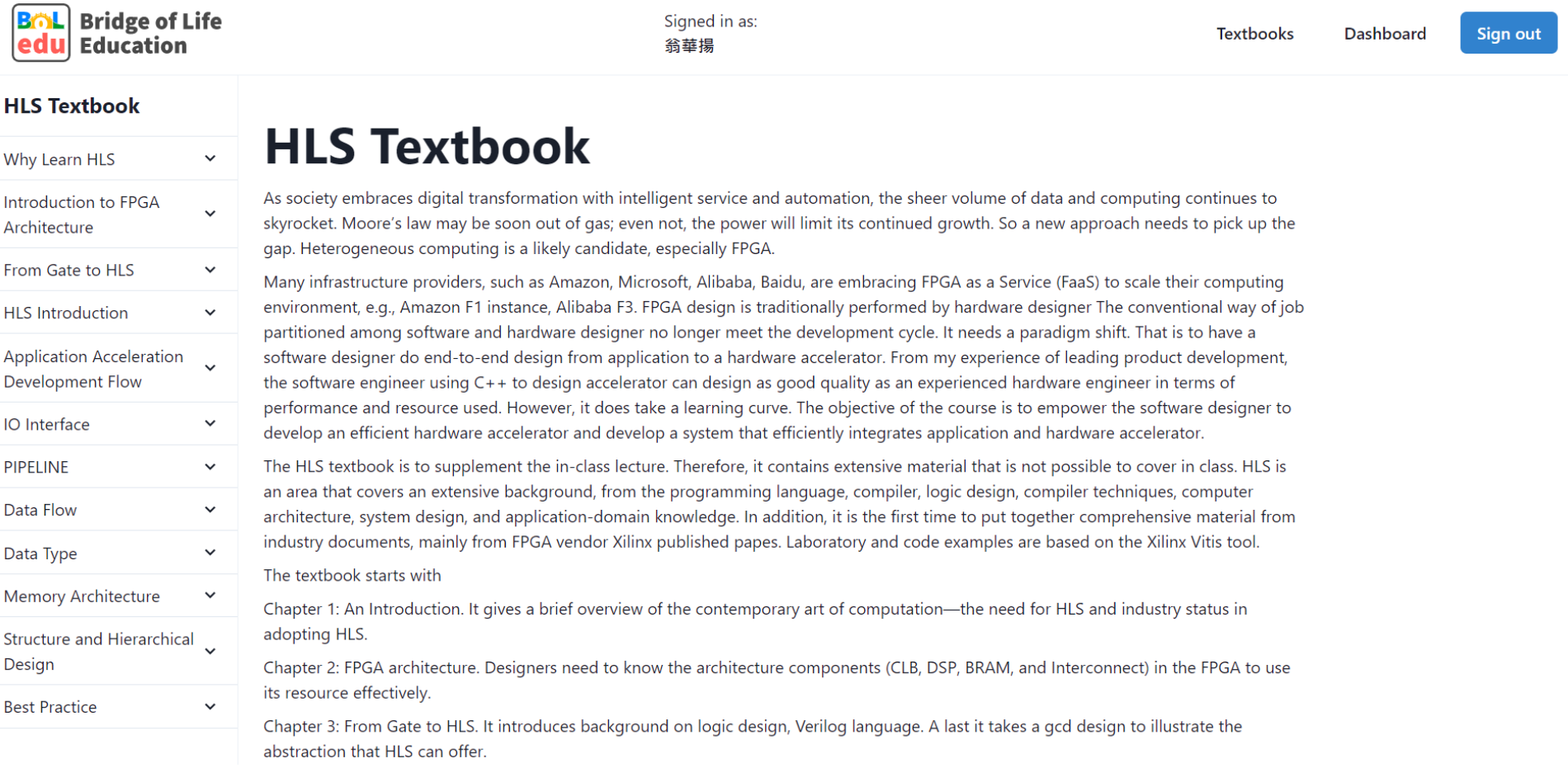
HLS Introduction & Lab

2023.08.02

Fan, Yung-Wei



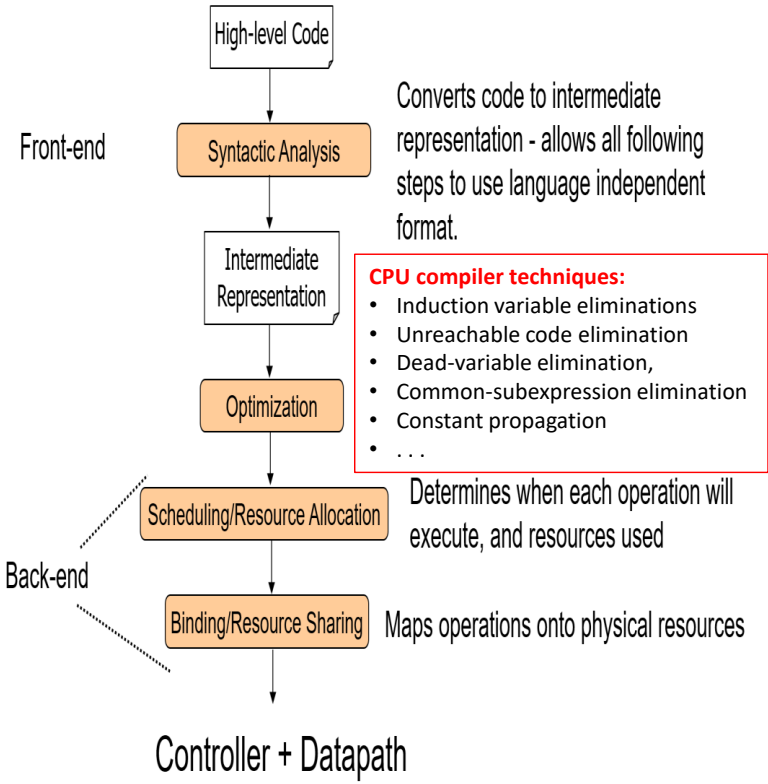
- See the below textbook for details
  - https://boledu-next-chakra.vercel.app/textbooks/hls-textbook
- Course:
  - NTUEE EEE5060 Application Acceleration with High-Level-Synthesis
  - NTUEE EEE5029 Multimedia System-on-chip Design

Outline

- Why HLS?
- HLS Introduction
- HLS Design Flow
- Lab

High Level Synthesis - HLS

- Convert (C/C++/OpenCL) into a RTL circuit
  - Optimize for power, performance, area, timing
  - Use Directives (Pragma) to direct compile/optimization process
- Vendors – FPGA vendors, IC
  - Xilinx Vitis-HLS, Intel HLS Compiler
  - Siemens/Mentor Catapult, Cadence Stratus HLS, (Synopsys Synphony HLS)
- Focus on the Back-end part
  - Scheduling/Resource Allocation
  - Binding/Resource Sharing



Code generation from a Domain-Specific Language for C-based HLS of hardware accelerators
https://dl.acm.org/doi/pdf/10.1145/2656075.2656081

Why HLS?

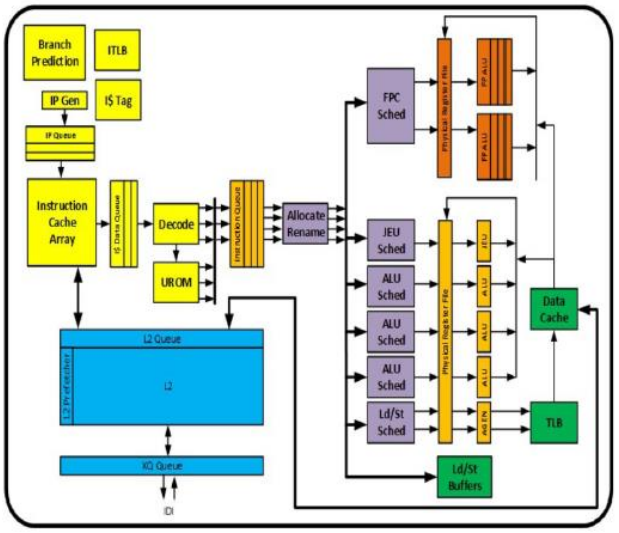
Why HLS?

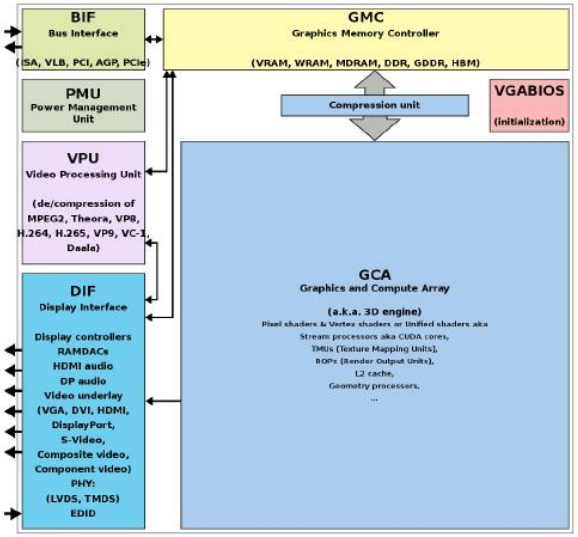
- Computational Efficiency
- FPGA development is made easy by HLS
  - C++ ←→ python v.s. verilog ←→ HLS
- Design and Verification Productivity

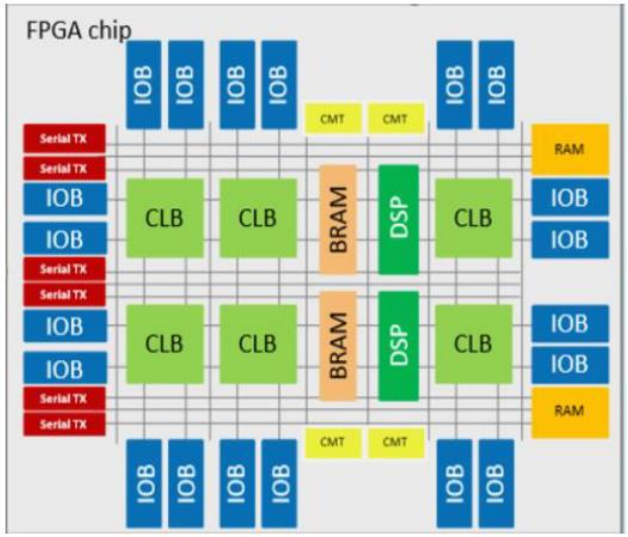
Why HLS?

- Computational Efficiency
- FPGA development is made easy by HLS
  - C++ ←→ python v.s. verilog ←→ HLS
- Design and Verification Productivity

• Compute Efficiency (GOPs/Watt)





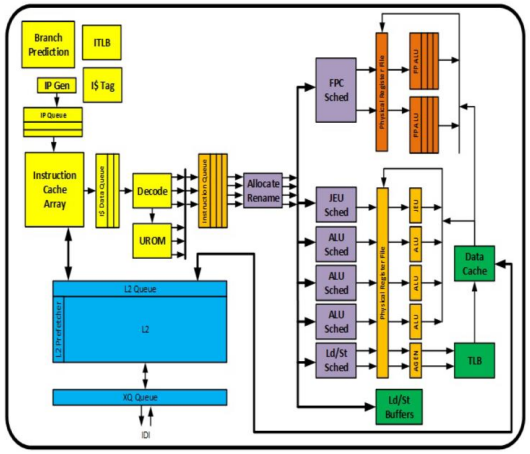


- Highly flexibility
- Low computing efficiency
- General software processing

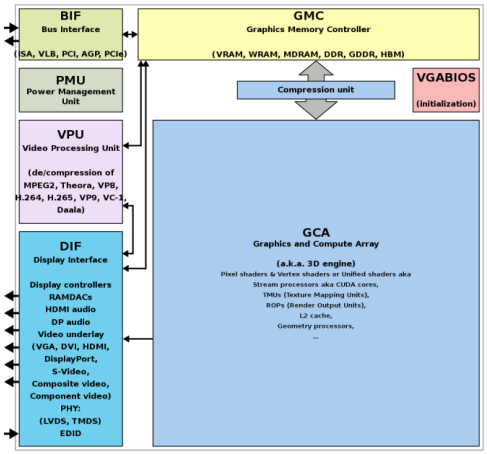
- Better flexibility
- Average computing efficiency
- Suited to simple logic and SIMD computationally intensive task

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

Ease of programmability vs. efficiency

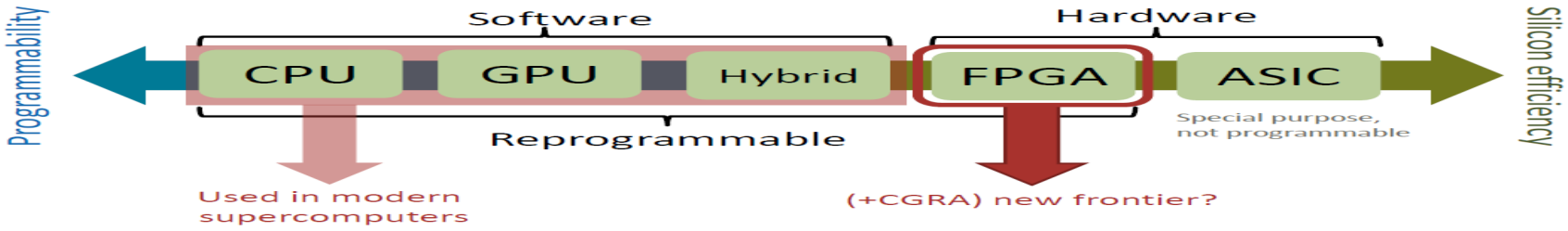


CPU:1



GPU:100

FPGA:1000



Computational Power Hits a Plateau

Load-store v.s. Dataflow architecture

- Load-store (“von Neumann”)
  - **Energy per instruction: 70pJ**

v.s.

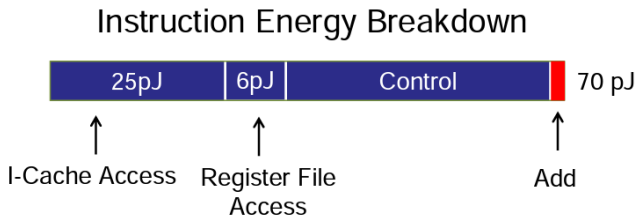
- Dataflow (“non von Neumann”)
  - **Energy per operation: 1 ~3pJ**

Rough Energy costs for various operations in 45nm 0.9V

Integer	
Add	
8 bit	0.03pJ
32 bit	0.1pJ
Mult	
8 bit	0.2pJ
32 bit	3 pJ

FP	
FAdd	
16 bit	0.4pJ
32 bit	0.9pJ
FMult	
16 bit	1pJ
32 bit	4pJ

Memory	
Cache	(64bit)
8KB	10pJ
32KB	20pJ
1MB	100pJ
DRAM	1.3-2.6nJ



Application Acceleration is a **dataflow management problem**

ISSCC 2014: Mark Horowitz, Computing's Energy Problem and What We Can Do About It: https://ieeexplore.ieee.org/document/6757323

Why is FPGA 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?

Why HLS?

- Computational Efficiency
- FPGA development is made easy by HLS
  - C++ ←→ python v.s. verilog ←→ HLS
- Design and Verification Productivity

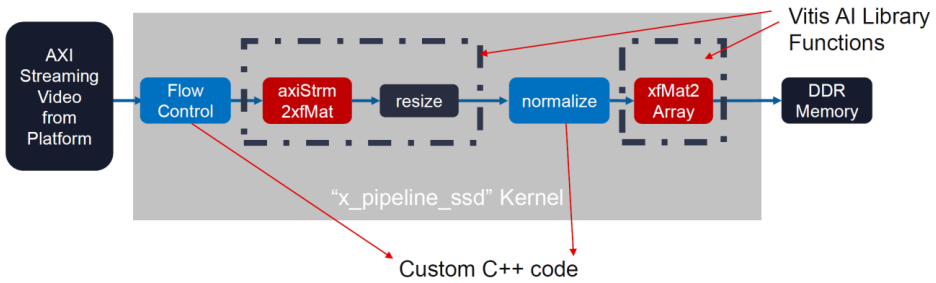
Difficulties in developing Application Accelerators

- HW language are low-level and very difficult
- Know nothing about hardware design
- How software application interacts with FPGA

FPGA Development Made Easy by HLS

- HW language are low-level and very difficult

- Use C, C++, OpenCL, Python, or TensorFlow



Code Snippet

Top Level Function Definition

Interface definitions

Dataflow the processing

Internal streaming variable declarations

Synchronize to start of frame

Convert from axi stream to xf::Mat

Resize the image

Subtract the mean values and apply scale

Convert from xf::Mat to memory mapped interface

```
void x_pipeline_ssd(hls::stream<ap_axiu<24, 3, 1, 1> >& image_in,
                    ap_uint<AXI_WIDTH> *image_out,
                    int width_in,  int height_in, int width_out,
                    int height_out, int use_mean,  float scale_r,
                    float scale_g, float scale_b, unsigned char mean_r,
                    unsigned char mean_g, unsigned char mean_b)
```

```
#pragma HLS INTERFACE axis port = image_in
#pragma HLS INTERFACE m_axi port = image_out offset = slave bundle = image_out_gmem depth = 131072
#pragma HLS INTERFACE s_axilite port = image_out bundle = control
#pragma HLS INTERFACE s_axilite port = width_in bundle = control
#pragma HLS INTERFACE s_axilite port = height_in bundle = control
#pragma HLS INTERFACE s_axilite port = width_out bundle = control
#pragma HLS INTERFACE s_axilite port = height_out bundle = control
#pragma HLS INTERFACE s_axilite port = use_mean bundle = control
#pragma HLS INTERFACE s_axilite port = scale_r bundle = control
#pragma HLS INTERFACE s_axilite port = scale_g bundle = control
#pragma HLS INTERFACE s_axilite port = scale_b bundle = control
#pragma HLS INTERFACE s_axilite port = mean_r bundle = control
#pragma HLS INTERFACE s_axilite port = mean_g bundle = control
#pragma HLS INTERFACE s_axilite port = mean_b bundle = control
#pragma HLS INTERFACE s_axilite port = return bundle = control
```

```
#pragma HLS DATAFLOW
```

```
hls::stream<ap_axiu<24, 0, 0, 0> > px_in_synced;
xf::cv::Mat<XF_8UC3, MAX_IN_HEIGHT, MAX_IN_WIDTH, NPC> in_mat(height_in, width_in);
#pragma HLS STREAM variable=in_mat.data depth=256 dim=1
xf::cv::Mat<XF_8UC3, MAX_IN_HEIGHT, MAX_IN_WIDTH, NPC> in_rgb(height_in, width_in);
#pragma HLS STREAM variable=in_rgb.data depth=256 dim=1
xf::cv::Mat<XF_8UC3, MAX_OUT_HEIGHT, MAX_OUT_WIDTH, NPC> out_rgb(height_out, width_out);
#pragma HLS STREAM variable=out_rgb.data depth=256 dim=1
xf::cv::Mat<XF_8UC3, MAX_OUT_HEIGHT, MAX_OUT_WIDTH, NPC> out_mat(height_out, width_out);
#pragma HLS STREAM variable=out_mat.data depth=256 dim=1
```

```
flow_control(image_in, px_in_synced, height_in, width_in);
```

```
xf::cv::axiStrm2xfMat<24, XF_8UC3, MAX_IN_HEIGHT, MAX_IN_WIDTH, NPC>(px_in_synced,in_mat);
```

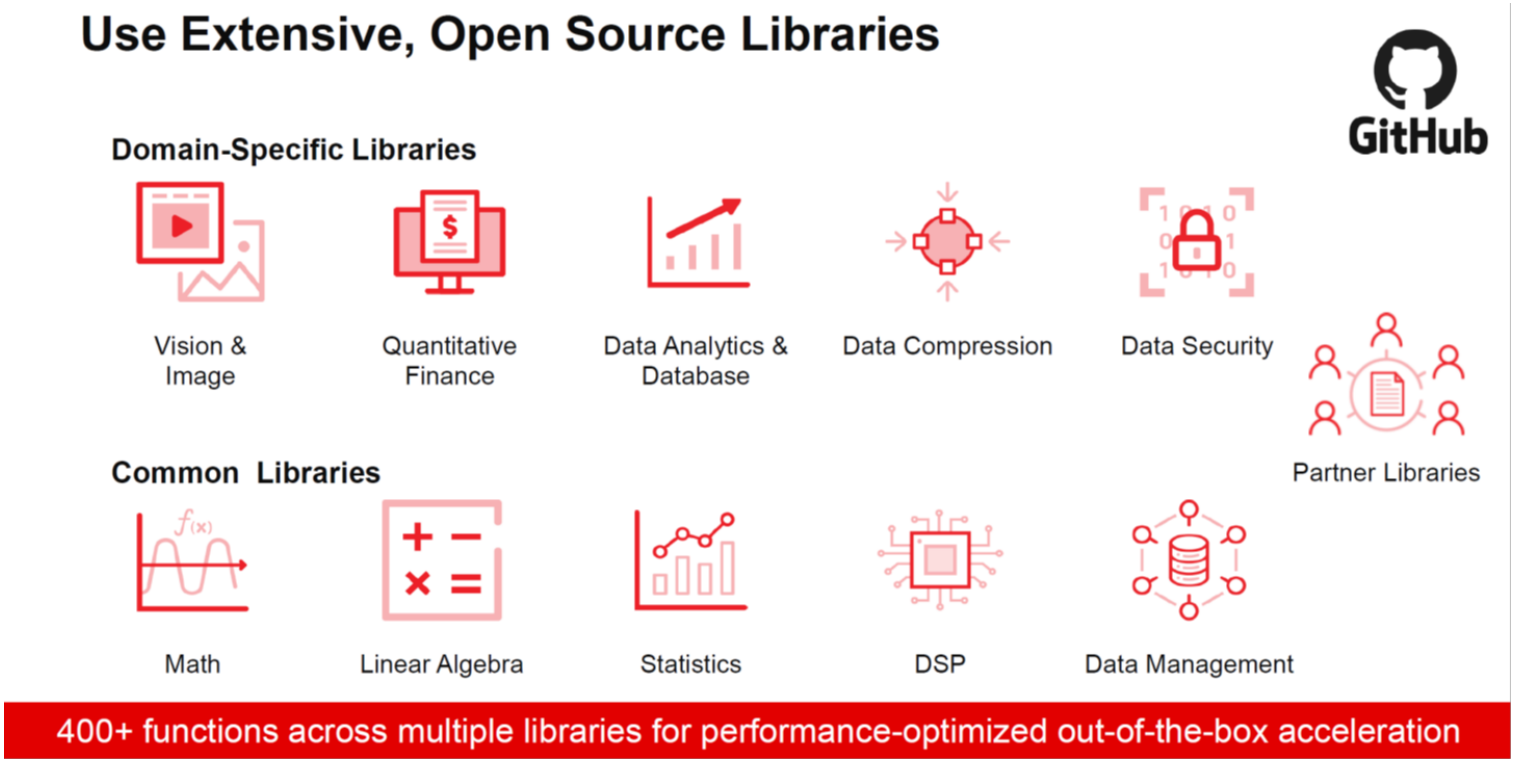
```
xf::cv::resize<XF_INTERPOLATION_NN,
               XF_8UC3,
               MAX_IN_HEIGHT,
               MAX_IN_WIDTH,
               MAX_OUT_HEIGHT,
               MAX_OUT_WIDTH,
               NPC,
               MAX_DOWN_SCALE>(in_mat, out_rgb);
```

```
image_normalize(out_rgb, out_mat, use_mean, scale_r, scale_g, scale_b, mean_r, mean_g, mean_b);
```

```
xf::cv::xfMat2Array<AXI_WIDTH, XF_8UC3, MAX_OUT_HEIGHT, MAX_OUT_WIDTH, NPC>(out_mat, image_out);
}
```


HLS

Speedup Development by Libraries



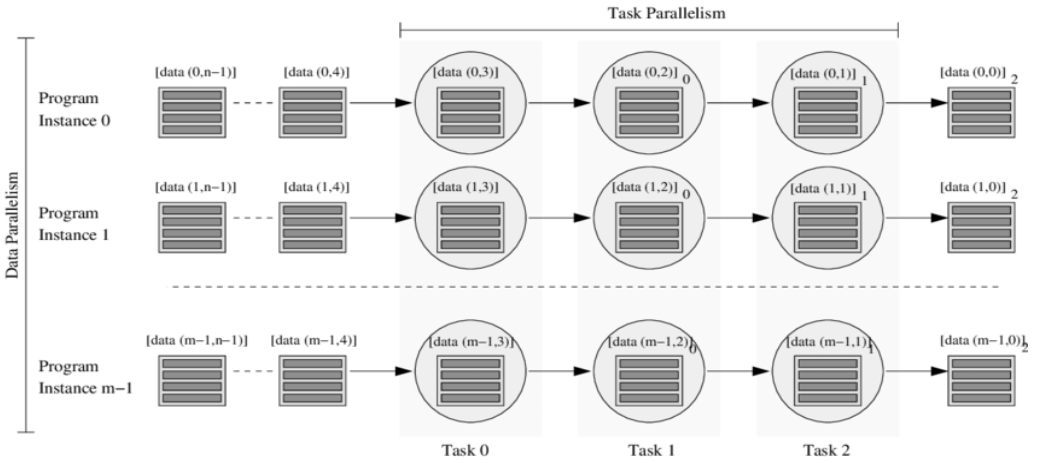
FPGA Development Made Easy by HLS

- Know nothing about hardware design

- Parallel programming concept apply

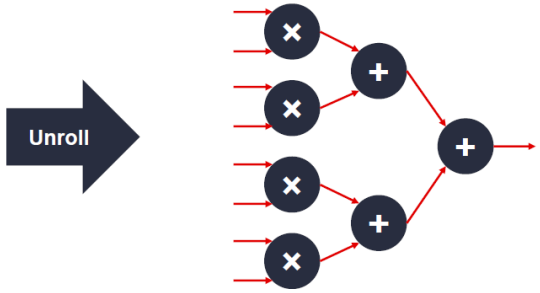
Think “Parallel”

- Data-level Parallelism
- Task-Level Parallelism



- Instruction (operator) - Level Parallelism

```
for (int i=0; i<N; i++)
{
  acc += A[i] * B[i];
}
```

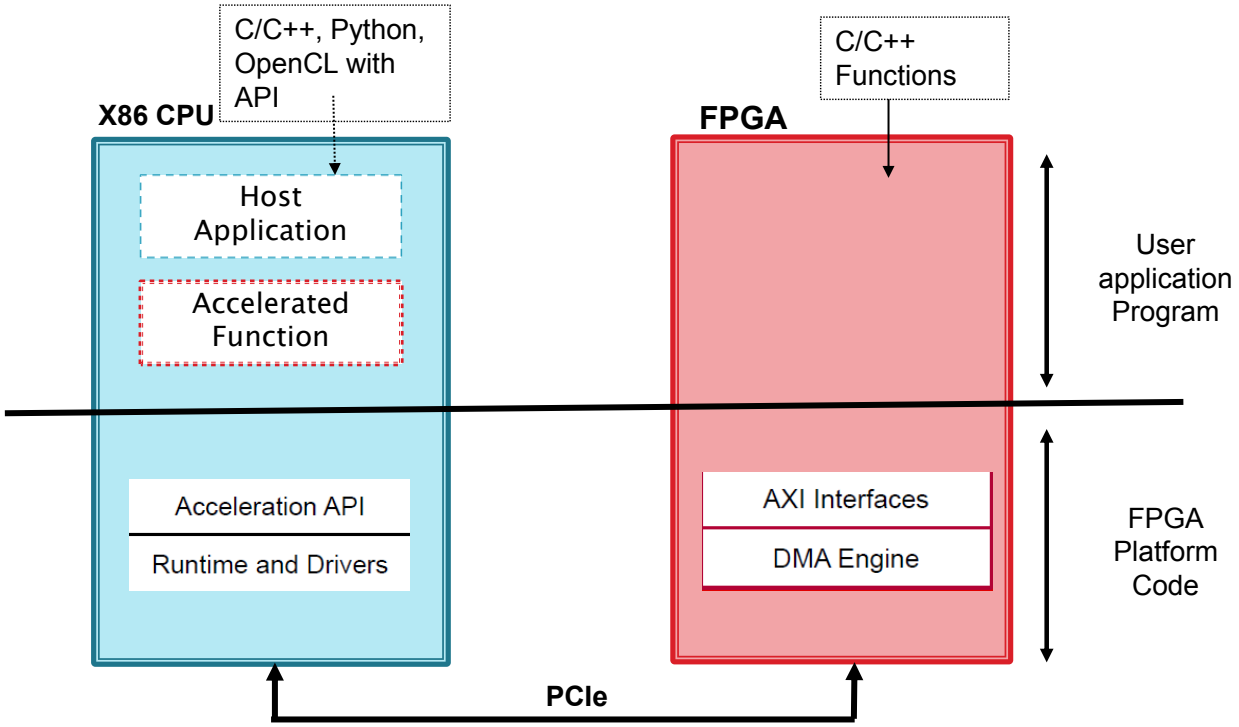


FPGA Development Made Easy by HLS

- How software application interacts with FPGA

- Off-shelf Platform ready

Software Interacts with FPGA



Moving function to FPGA creates a lot of overhead.
Can we really gain performance and reduce power?

Why HLS?

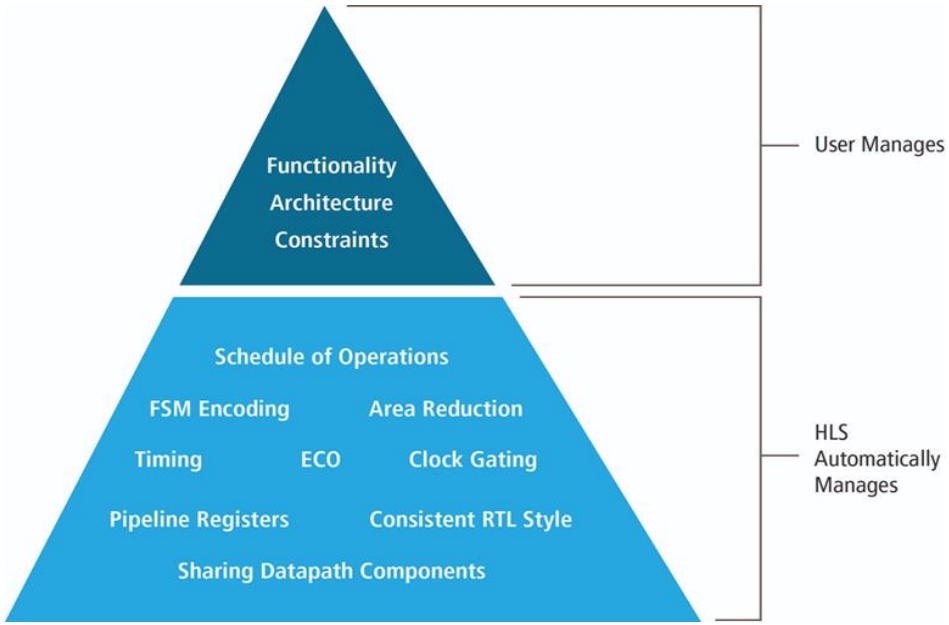
- Computational Efficiency
- FPGA development is made easy by HLS
  - C++ ←→ python v.s. verilog ←→ HLS
- Design and Verification Productivity

Design and Verification Productivity

- Fast architecture exploration
- Improve Quality of Results (QoR)
- Industry uses cases

How is possible that HLS can have better QoR than RTL code?

Design and Verification Productivity



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

Design Exploration with Directives

One body of code:
Many hardware outcomes

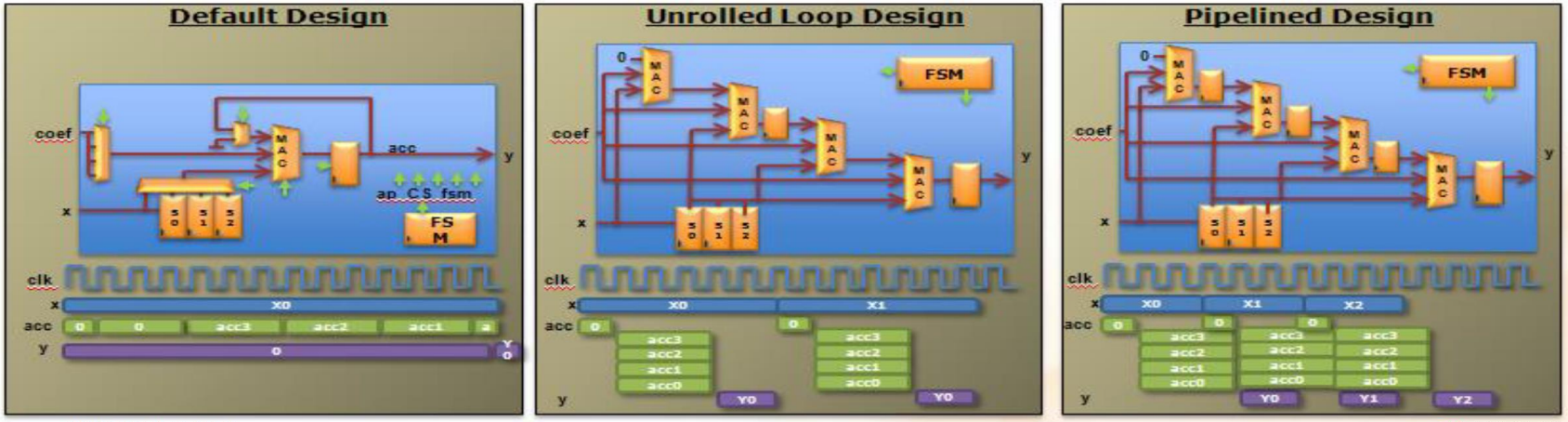
```
...
loop: for (i=3;i>=0;i--) {
  if (i==0) {
    acc+=x*c[0];
    shift_reg[0]=x;
  } else {
    shift_reg[i]=shift_reg[i-1];
    acc+=shift_reg[i]*c[i];
  }
}
....
```

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



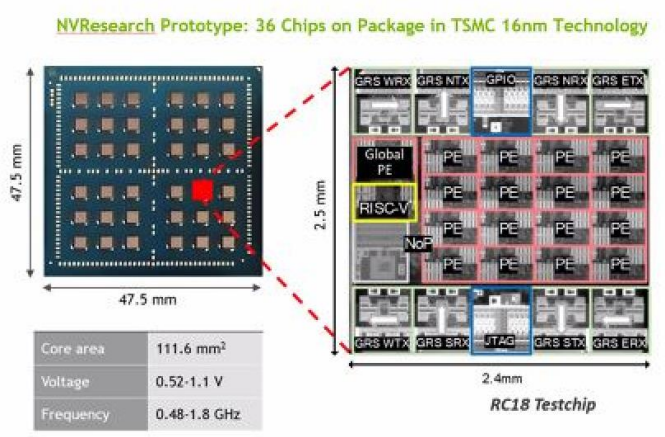
XILINX.

Industry case - Nvidia

Nvidia Research – Machine Learning Accelerator

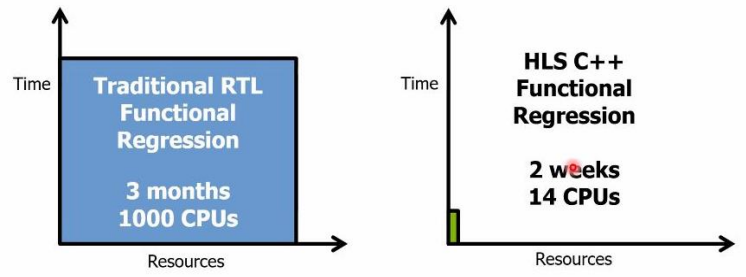
"10X Improvement in RTL design and verification effort compared to manual RTL"

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



Nvidia Xavier 12nFF SoC

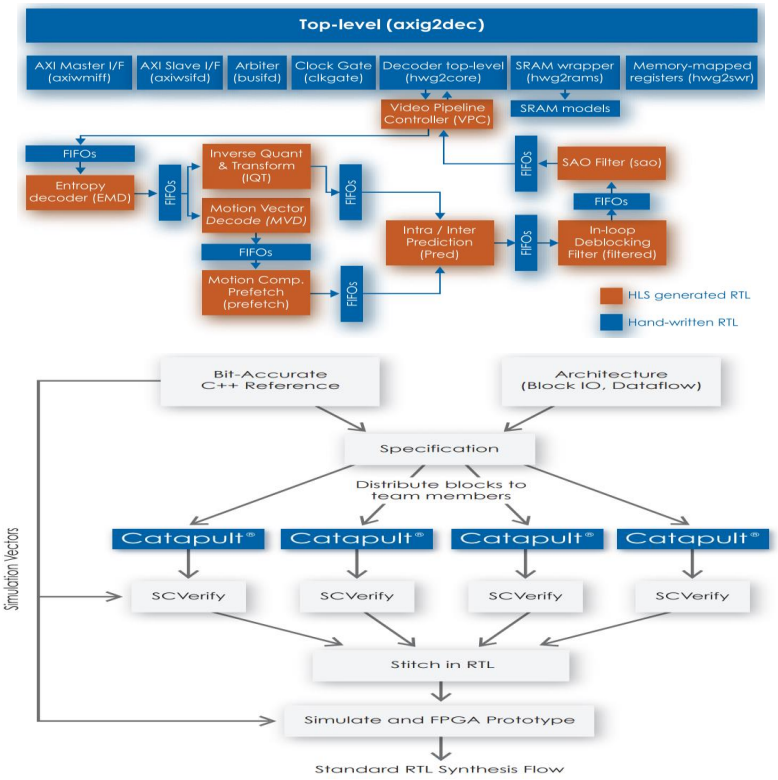
- C++ functional verification runtime ~500x less resource than RTL
- Fast verification makes rapid product changes possible
  - VP9/HEVC code from 8 to 10 bit color depth in 2 weeks
  - Change from 20nm/500Mhz to 28/nm/800Mhz in 3 days with HLS



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



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

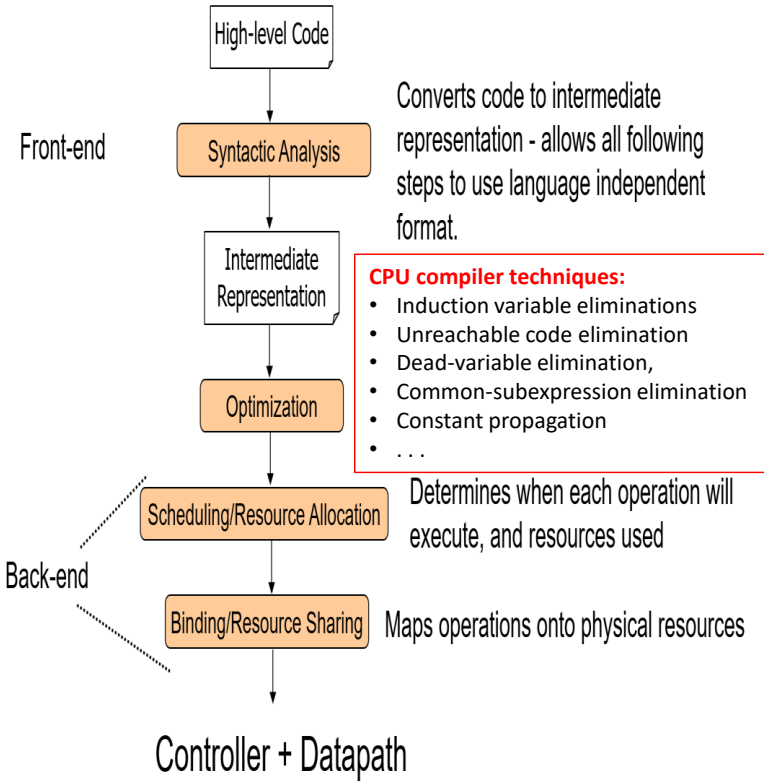
HLS Introduction

HLS Introduction

- C to RTL Mapping
  - Function, Array, Loop
  - Pragmas / Directives
- Software and Hardware are different
  - Unsupported C/C++ Constructs
- Example - GCD

High Level Synthesis - HLS

- Convert (C/C++/OpenCL) into a RTL circuit
  - Optimize for power, performance, area, timing
  - Use Directives (Pragma) to direct compile/optimization process
- Vendors – FPGA vendors, IC
  - Xilinx Vitis-HLS, Intel HLS Compiler
  - Siemens/Mentor Catapult, Cadence Stratus HLS, (Synopsys Synphony HLS)
- Focus on the Back-end part
  - Scheduling/Resource Allocation
  - Binding/Resource Sharing



Code generation from a Domain-Specific Language for C-based HLS of hardware accelerators
https://dl.acm.org/doi/pdf/10.1145/2656075.2656081

HLS Introduction

- C to RTL Mapping
  - Function, Array, Loop
  - Pragmas / Directives
- Software and Hardware are different
  - Unsupported C/C++ Constructs
- Example - GCD

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

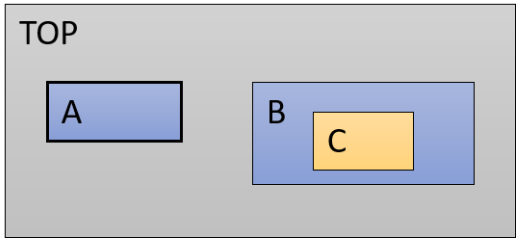
Expression/Operators: Function unit. Allocation/Scheduling (Sharing) to meet performance and area

```
#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

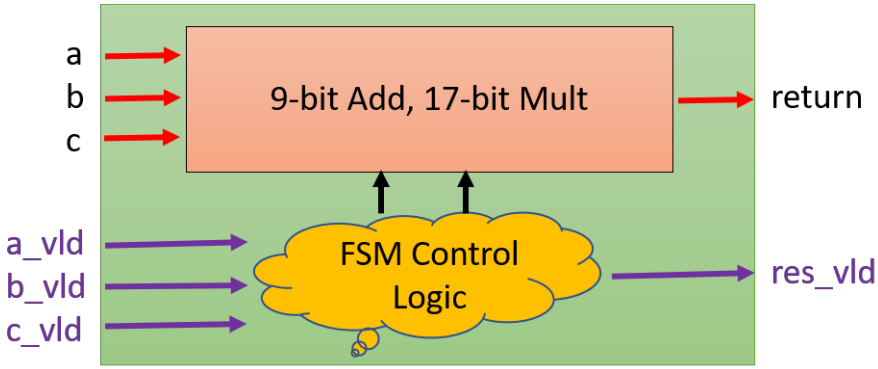
```
void A { ... Body A ...}
void C { ... Body C ...}
void B {   C; }
void TOP() {
                A  ( ... )
                B  ( ... )
}
```



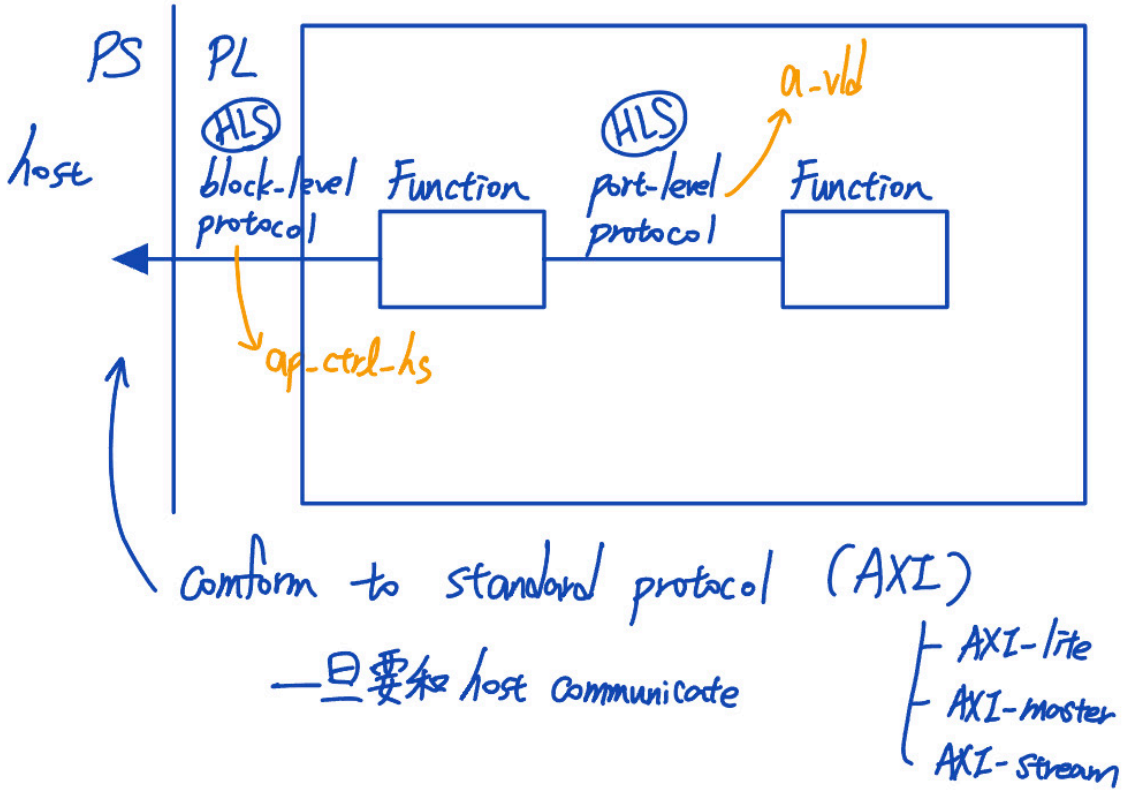
Function Arguments

- Function arguments mapped to ports on the RTL blocks
- **Global variable** if accessed only local to the function, no io port created.
- Insert control ports (Port-level Protocol) to automatically synchronize data exchange among blocks
- Insert Block-level Protocol on Top level function to communicate with Host
- Arbitrary precision bit-width to reduce resource and latency

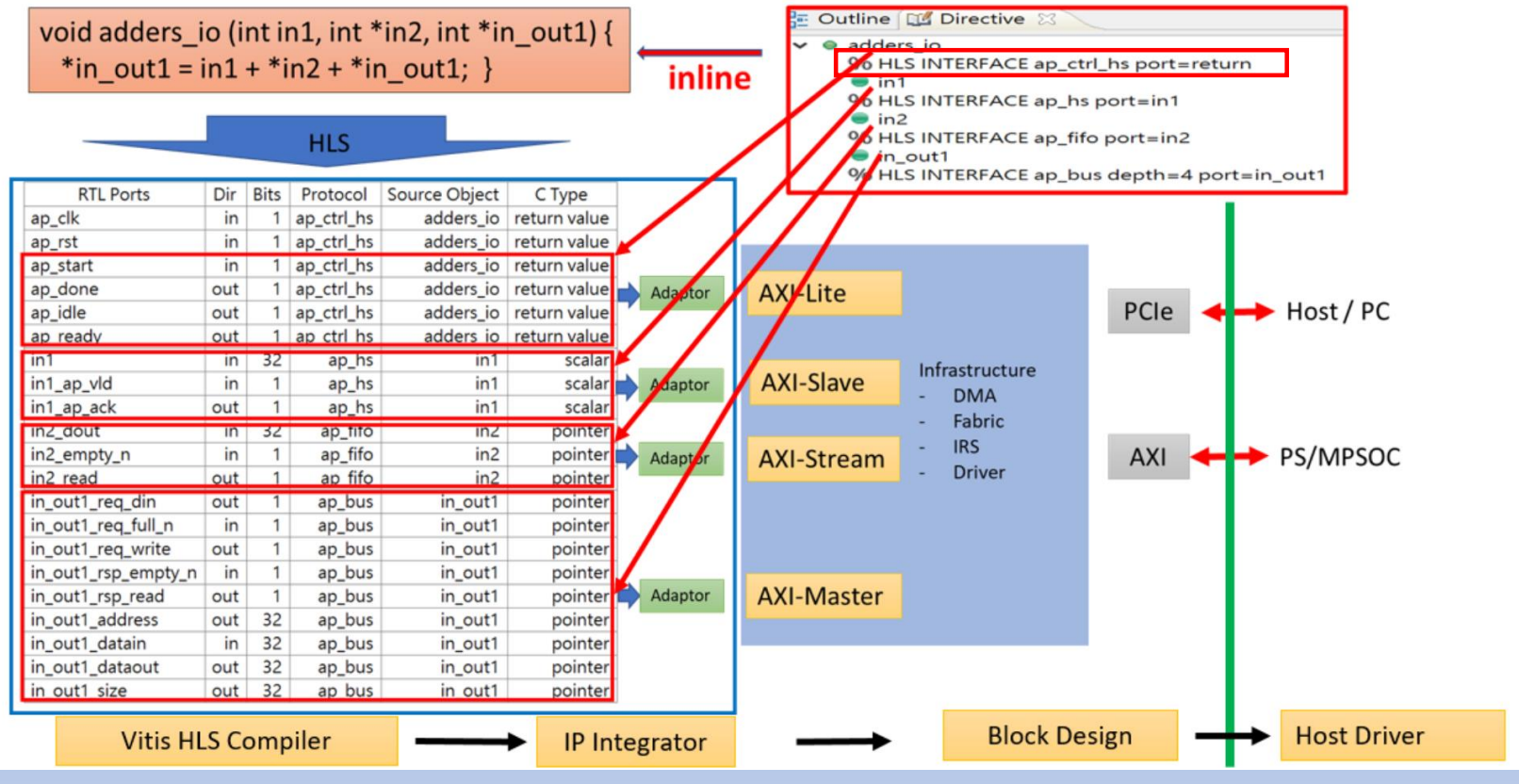
```
int17  foo_top(int8* a, int8* b, int8* c, int17* ret)
{
        int sum, multi;

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



Function, Top function (Kernel) explained

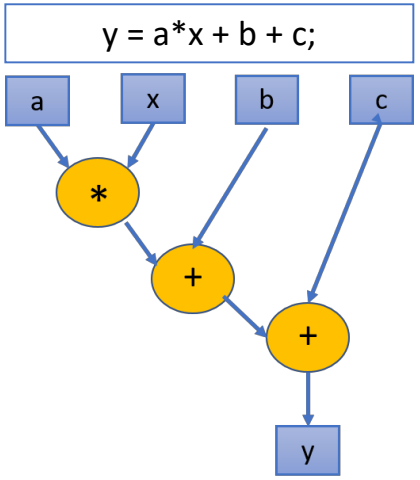


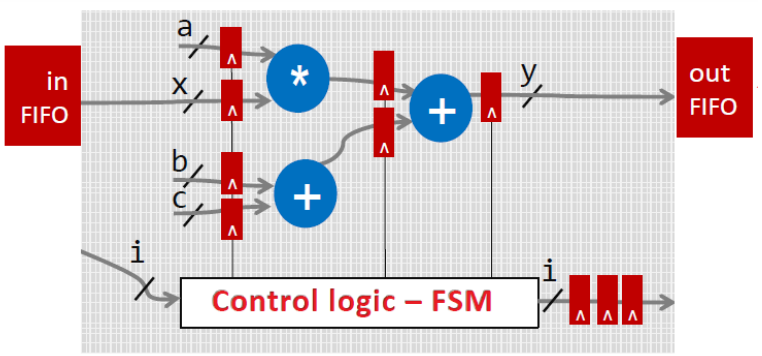
Host/Kernel Communication



Expressions – Data Flow Graph

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





Resource Allocation, Scheduling, Binding

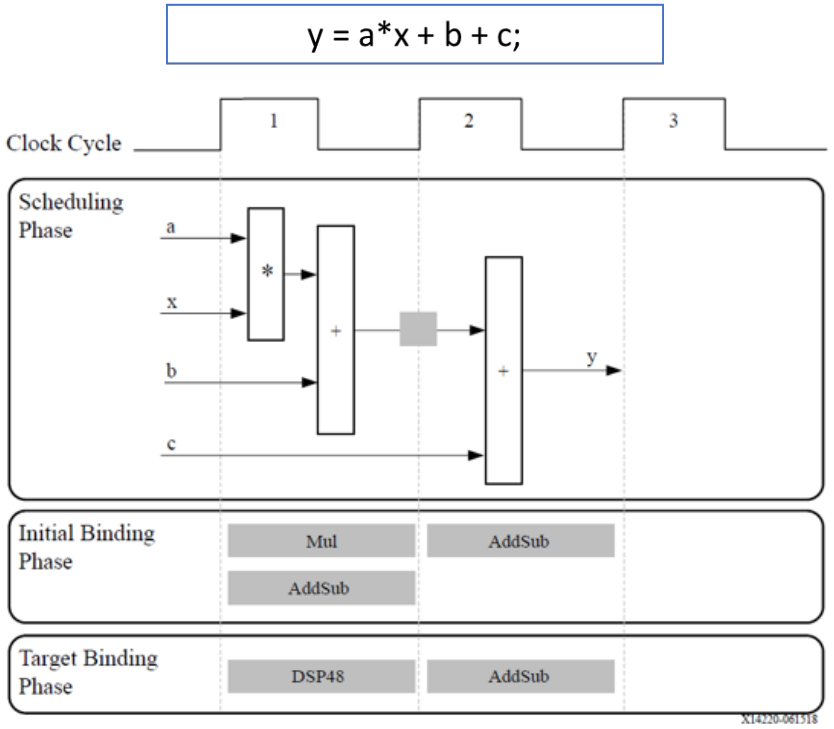
- **Resource allocation**: Each operation is mapped to a hardware resource, annotated with both timing and area information

  #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>



Arrays

- Typically implemented by a memory block
  - Read & write array mapped to RAM
  - Constant array mapped to ROM
- Memory access is often the performance bottleneck
  - HLS default memory model assumes 2-port BRAM
  - Array can be reshaped and/or partitioned to remove bottleneck

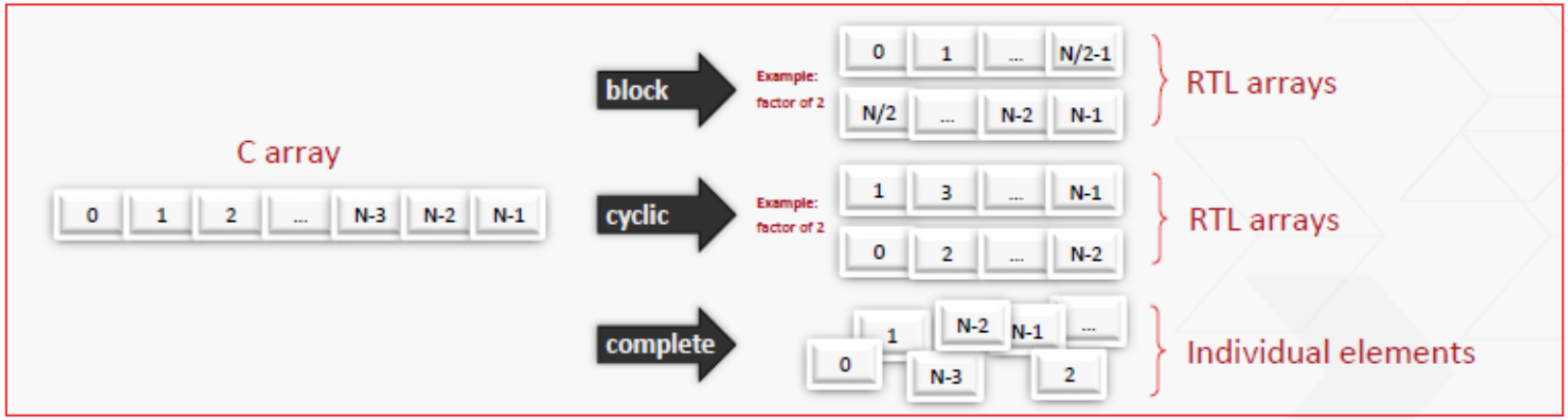

```
void foo (...) {
...
SUM_LOOP:for(i=2;i<N;++i) {
  sum += mem[i] + mem[i-1] + mem[i-2];
  ...
  }
}
```

See UG902 to get full throughput on this example
- *(Chap 3 – Array Accesses and Performance)*

Example: Code implies three reads from a RAM, prevents full throughput

Partition, Reshape Your Arrays

- Partitioning splits an array into independent arrays
  - Array can be partitioned on any of their dimensions for better throughput
- Reshaping combines array elements into wider containers
  - Different arrays into a single physical memory
  - New RTL memories are automatically generated without changes to C code



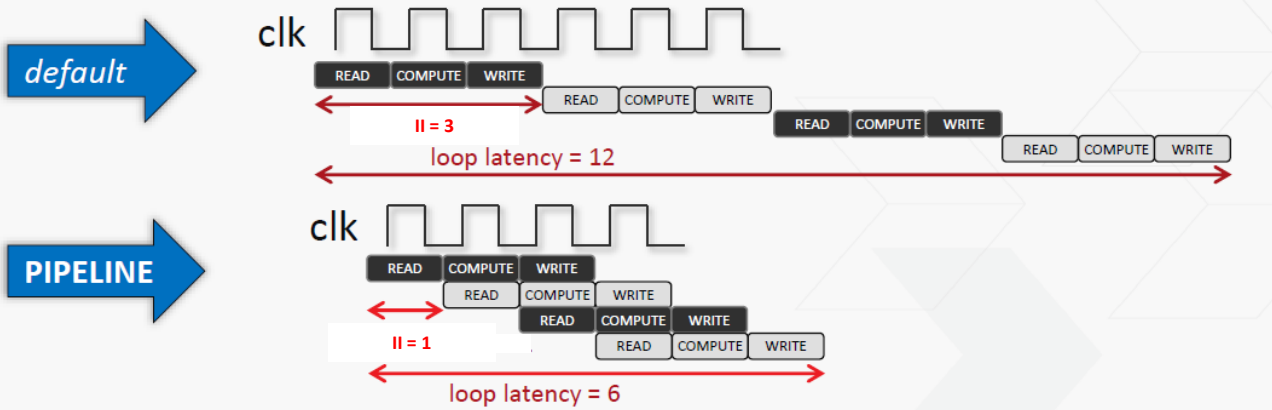
Control Flow: Loop

- 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

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)

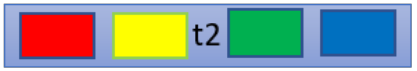
```
void F (...)  {
...
add: for (i=0;i<=3;i++) {
# PRAGMA HLS PIPELINE
      op_READ;
      op_COMPUTE;
      op_WRITE;
}
...
```



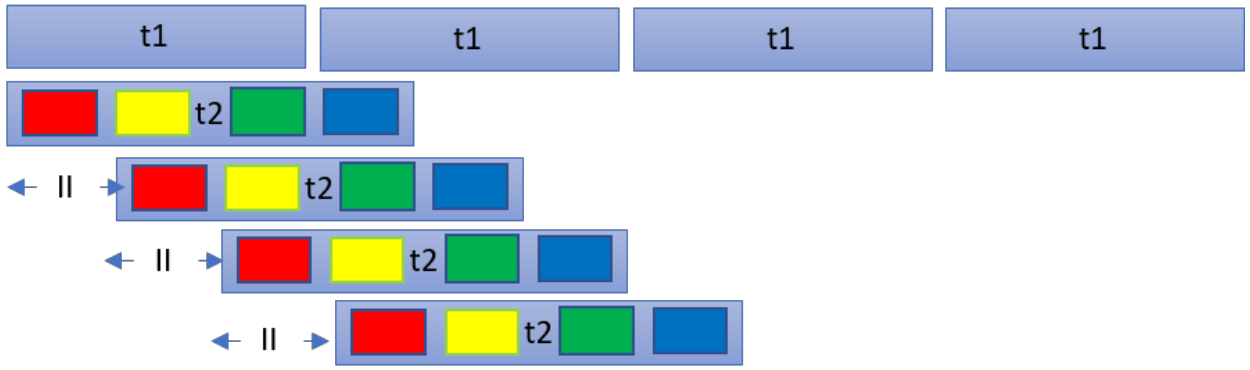
Pipeline Latency v.s. Throughput

A function, latency t1

Decompose a function into 4 steps, latency t2

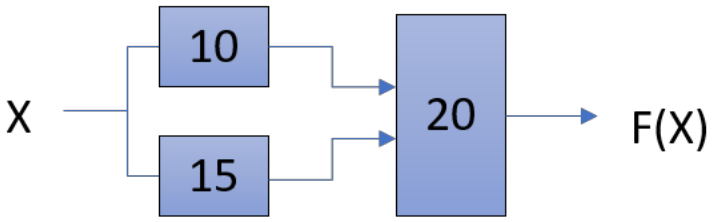


Compare the latency to execute 4 iterations of non-pipelined function (4*t1) v.s. pipelined functions (t2 + 3 II)

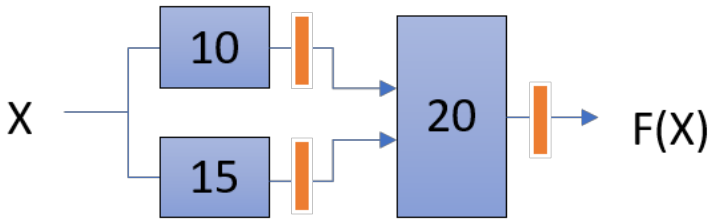


II: Initiation Interval

Pipeline May Take Longer Latency



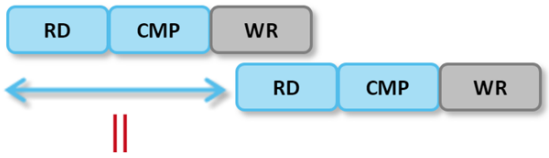
Latency = 35
Throughput = 1/35



T_{clk} (clock period) = 20
Latency = 2 * Tclk = 40
Throughput = 1/20

Initiation Interval (II) Concept

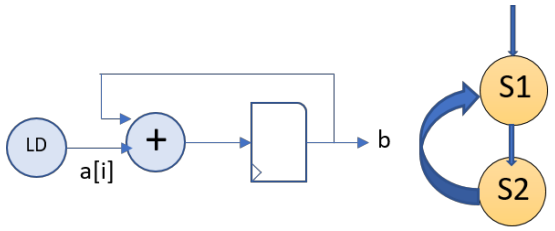
- Latency = time between start and finish of a task
- Throughput = number of tasks finished in a given time
- Throughput = 1/Latency?
- Initiation Interval (II) = Number of clock between new input samples
- Iteration Latency = # of clock to execute one iteration (L)
- Loop Latency = # of clock to execute all the operations in a loop
- Ultimate goal is to achieve II = 1 (most critical performance metric)



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 F ( . . . ) {
. . .
add: for (i=0; i <= 3; i++) {
    b += a[i] + b;
. . .
```



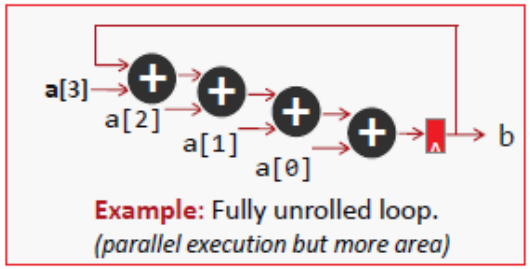
Loop - Unroll

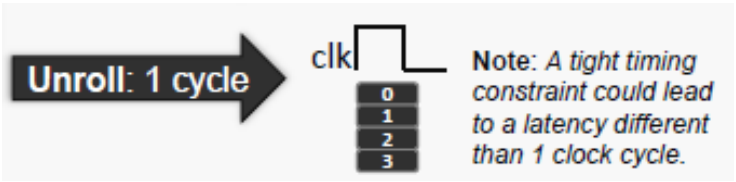
- Rolled loops can be made unrolled or partially unrolled by

  #pragma UNROLL [factor = n]

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

```
void F ( . . . ) {
. . .
add: for (i=0; i <= 3; i++) {
   #pragma UNROLL
   b += a[i];
. . .
```



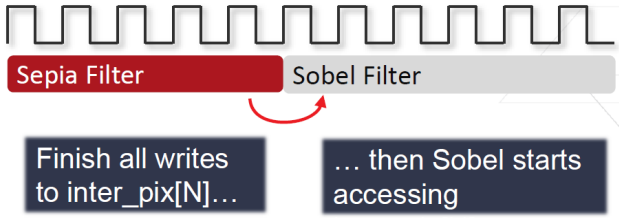


Task-Level Parallelism - Dataflow

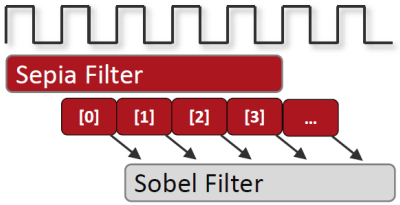
- **By default a C function producing data for another is fully executed first**

```
// This memory can be a FIFO during optimization
rgb_pixel inter_pix[MAX_HEIGHT][MAX_WIDTH];

// Primary processing functions
sepia_filter(in_pix,inter_pix);
sobel_filter(inter_pix,out_pix2);
```

Sepia Filter
Sobel Filter



- **Dataflow allows Sobel to start as soon as data is ready**
  - Functions operate concurrently and continuously
  - The interval (hence throughput) is improved
  - Channel buffer has to be filled before consumed for ping-pong



- **Dataflow creates memory channels**
  - Created between loops or functions to store data samples
  - “Ping-pong” channel holds all the data
  - “FIFO” for sequential access, no need to store all the data

#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

HLS Introduction

- C to RTL Mapping
  - Function, Array, Loop
  - Pragmas / Directives
- Software and Hardware are different
  - Unsupported C/C++ Constructs
- Example - GCD

Data Dependency

RAW (Read After Write)
- **True dependency**
- S1-iteration(u) -> S2-iteration (v)
- S1 computes a value S2 uses

WAR (Write After Read)
- S1 read from a memory location update by S2
- Renaming to resolve WAR

WAW (Write After Write)
- S1 write to a memory that write by S2
- Renaming to resolve WAW

```
for(i=0; I < N; i++)
{
   A[i] = A[i-1] * a;
 }
```

```
For( … i++) {
  A[i-1] = b – a;
  B[i] = A[i] + 1 ;
}
```

```
For( … i++) {
  B[i] = A[i-1] + 1;
  A[i] = B[i+1] + b
  B[i+2] = b – a; }
```

Latency ->

Iteration->

l/A0	*	s/A1			
	l/A1	*	s/A2		

Latency ->

Iteration->

s/A0	l/A1	S/B1			
	s/A1	*	s/B2		

Latency ->

Iteration->

l/A0	S/B1	l/B2	s/A1	s/B3	
	l/A1	s/B2	l/B3	s/A2	s/B3
		l/A2	s/B3	l/B4	

We don’t have compiler/processor take care it for us.

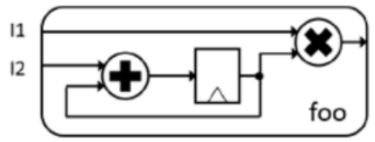
https://www.geeksforgeeks.org/dependency-graph-in-compiler-design/

Register renaming: http://people.ee.duke.edu/~sorin/ece252/lectures/4.2-tomasulo.pdf

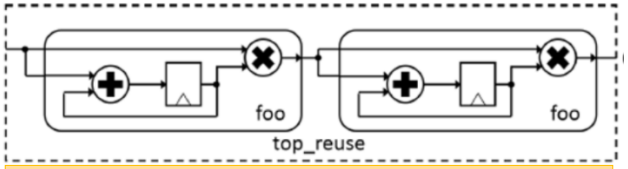
Software (C/C++) and Hardware (HDL Simulator) Behave Differently

1. In software, a statement is evaluated once in a sequential manner v.s. in event-driven hardware scheduling, out-of-order, and re-evaluate if RHS variables/signals change.

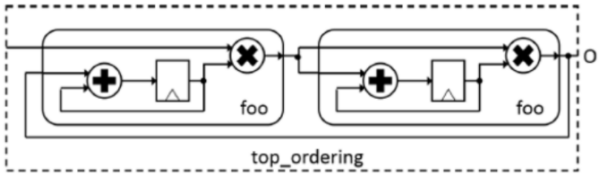
2. In software C program, statements are evaluated in a blocking manner, vs. in hardware, it is non-blocking, i.e., runs concurrently.




```
void foo(int I1, int I2, int *O) {
   static int L; // latch

   *O=I1*L; // read current-L
   L=I2+L; // assign next-L
}
```




```
void fxn_reuse_try(int I, int *O) {
   int tmp; // output of left module

   foo(I,I,&tmp); // left in figure
   foo(tmp,tmp,O); // right in figure
}
```




```
void fxn_ordering_try(int I, int *O) {
   int tmp1; // output of left module
   int tmp2; // output of right module

   foo<1>(I,tmp2,&tmp1); // left in figure
   foo<2>(tmp1,tmp1,&tmp2); // right in figure
   *O=tmp2; }
```

Unsupported C/C++ Constructs

- System Calls
- Dynamic Memory Usage (malloc)
- No C++ dynamic polymorphism nor dynamic virtual function call
  - Static/Compile-time polymorphism (function/operator overloading) is ok
- Pointer Limitation
- Recursive Functions

All resource must be statically allocated at compilation stage

Refer to ug902

System Calls

- e.g., printf(), malloc(), getc(), time(), sleep()
- HLS defined macro **__SYNTHESIS__** to exclude non-synthesized code
- **__SYNTHESIS__** is only defined in HLS
- Maintain the same copy of the source code for C-simulation and C/RTL co-simulation

```
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);
}
```

Dynamic Memory Usage

- Memory allocation: malloc(), alloc(), and free()
- User-defined macro NO_SYNTH

```
#include "malloc_removed.h"
#include <stdlib.h>
// #define NO_SYNTH

dout_t malloc_removed(din_t din[N], dsel_t width) {
#ifdef NO_SYNTH
   long long *out_accum = malloc (sizeof(long long));
   int* array_local = malloc (64 * sizeof(int));
#else
   long long _out_accum;
   long long *out_accum = &_out_accum;
   int _array_local[64];
   int* array_local = &_array_local[0];
#endif
……
}
```

Pointer Limitation

- **General Pointer Casting**
  - Not support general pointer casting
  - Support pointer casting between native C/C++ types.
- **Pointer Arrays**
  - Support pointer array to a scalar or an array of scalars
  - Not support array of pointers point to additional pointers
- **Function Pointer – not supported**

Recursive Function – A GCD Example

- Recursive functions cannot be synthesized because their function call depth is data-dependent, thus non-determined at compiler time.
- **Tail-recursion is a loop in disguise**, the simple function can easily be transformed as right.

```
unsigned foo (unsigned m, unsigned n) {
   if (m == 0) return n;
   if (n == 0) return m;
   return foo(n, m%n);
}
```

```
unsigned foo (unsigned m, unsigned n) {
   while( m!=0 & n!=0 ) {
      unsigned int mmodn=m%n;
      m=n;
      n=mmodn;
   }
   if (m == 0) return n;
   else return m;
}
```


https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Recursive-Functions

HLS Introduction

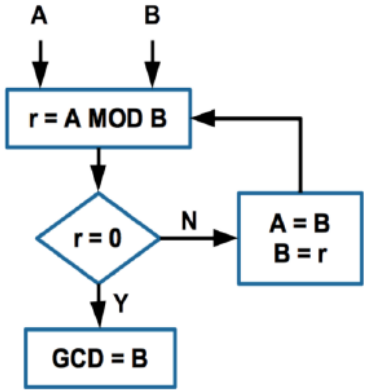
- C to RTL Mapping
  - Function, Array, Loop
  - Pragmas / Directives
- Software and Hardware are different
  - Unsupported C/C++ Constructs
- Example - GCD

Illustration – GCD

Euclidean Algorithm

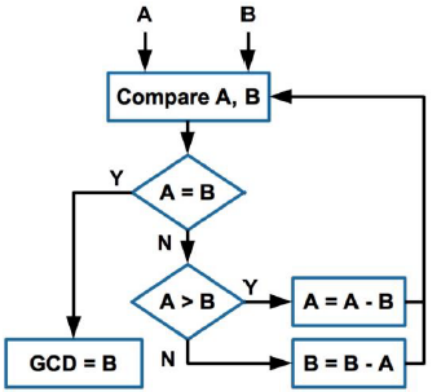
$$\gcd(a,b) = \gcd(b,r)$$

where, a = qb + r



Simplified Euclidean GCD Algorithm

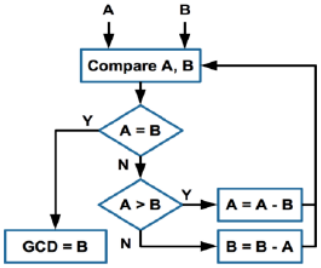
$$\gcd(a,b) = \gcd(b,(a-b))$$
$$= \gcd(a,(b-a))$$

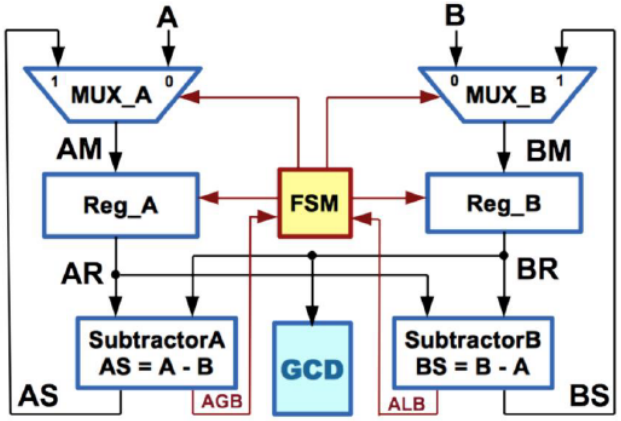



```
module gcd_behavior #(parameter width = 32)
              ( input  [width-1: 0] A_in, B_in,
                output [width-1:0] Y );
reg [width-1:0] A, B, Y, swap
Integer  done;

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)



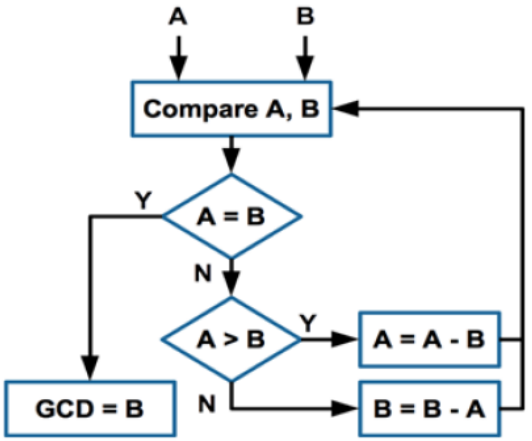



```
module gcd_fsm(
                  input  clock, reset, go,
                  input  AGB, ALB,
                  output  A_en, B_en,
                  A_mux_sel, B_mux_sel,
                  out_mux_sel, ouput done );
reg running = 0;
always @( posedge clock) begin
   if( go )    running <= 1;
   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;
}
```

HLS Design Flow

HLS Design Flow

- Design Flow
- HLS IP Flow

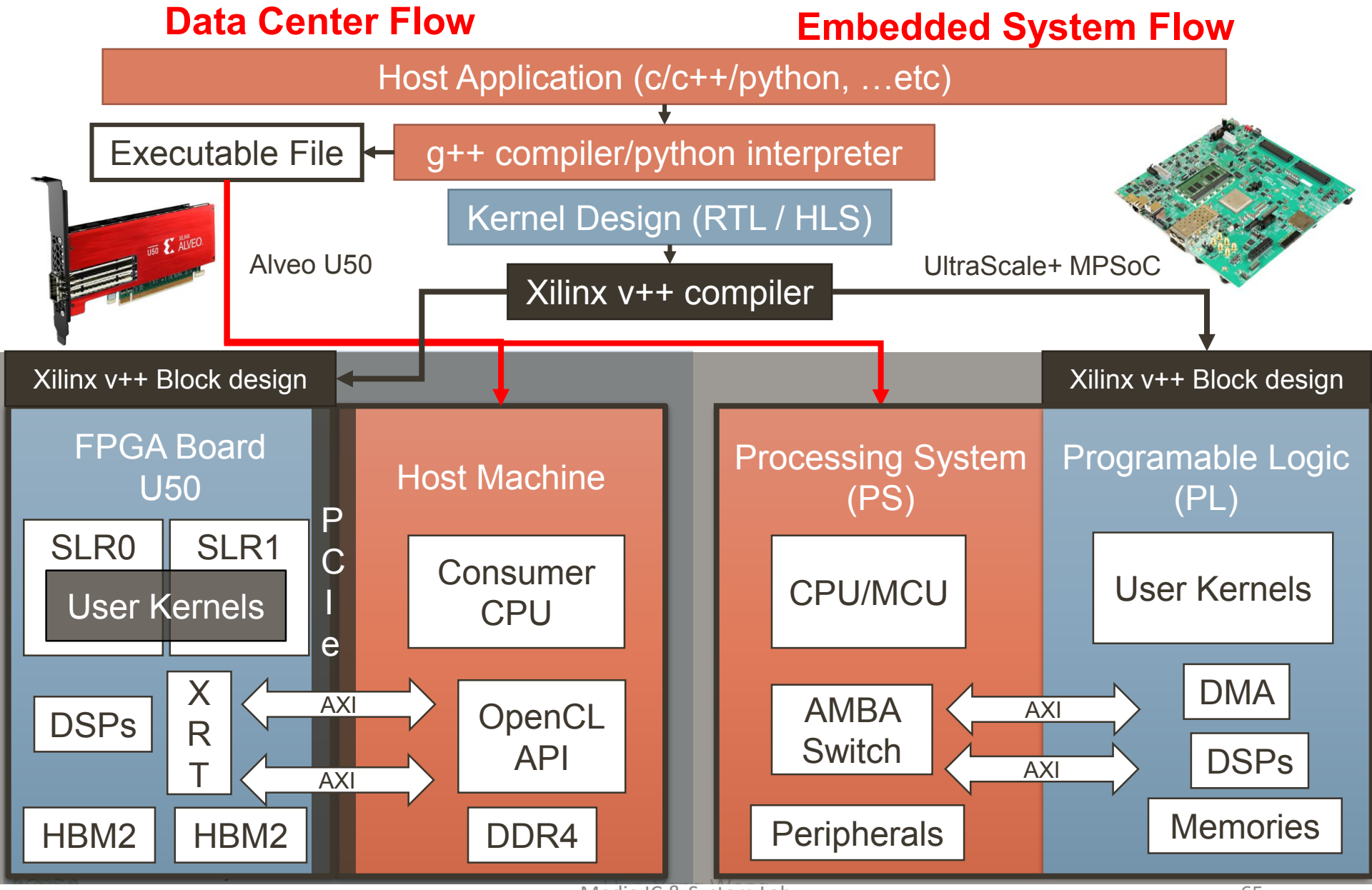
HLS Design Flow

- Design Flow
- HLS IP Flow

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



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

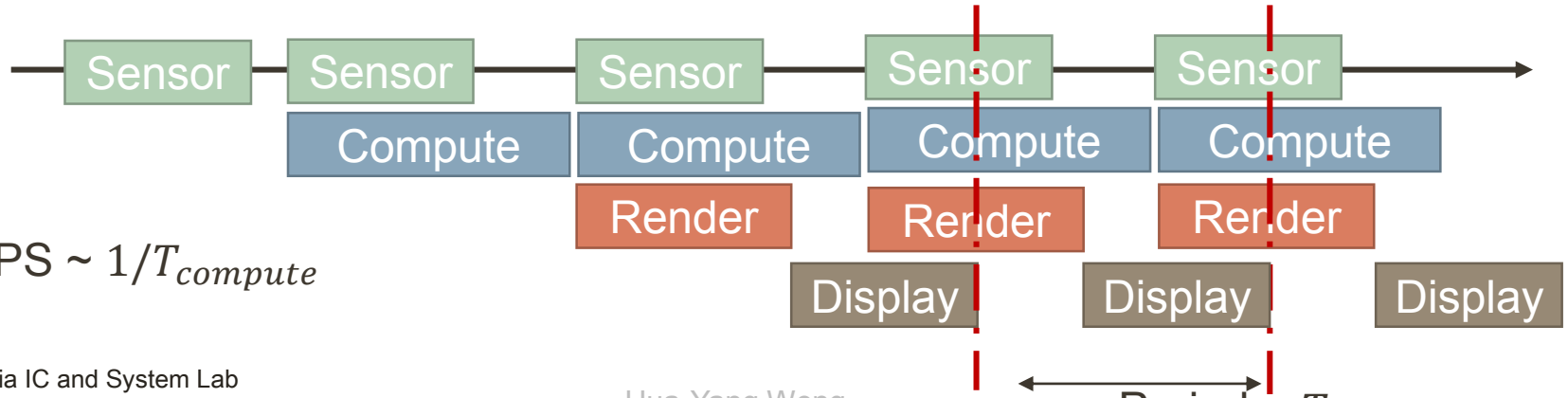
- You can use timers such as std::chrono
  - https://en.cppreference.com/w/cpp/chrono
- The platform and the underlying computational cores matters a lot
  - High-end CPU, low-end CPU, MCU, GPU, ……

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**



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

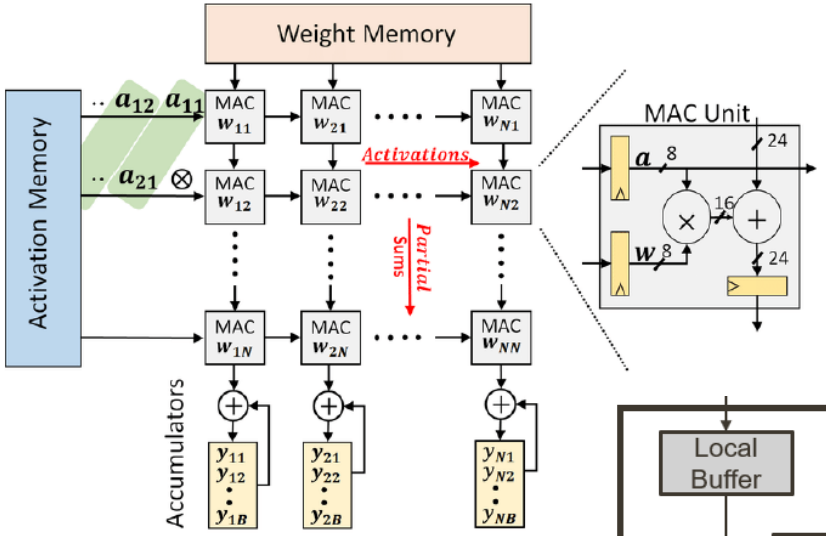
- Is this goal competitive?
  - x4 times faster, not too impressive (Impressive -> 2 ~ 3 orders)
- Achievable ?
- **Roughly estimate the cycles needed**
  - What's the time complexity of the function? How much degree of parallelism can be achieved to reach this goal? (Resource enough?)
- **Determine 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

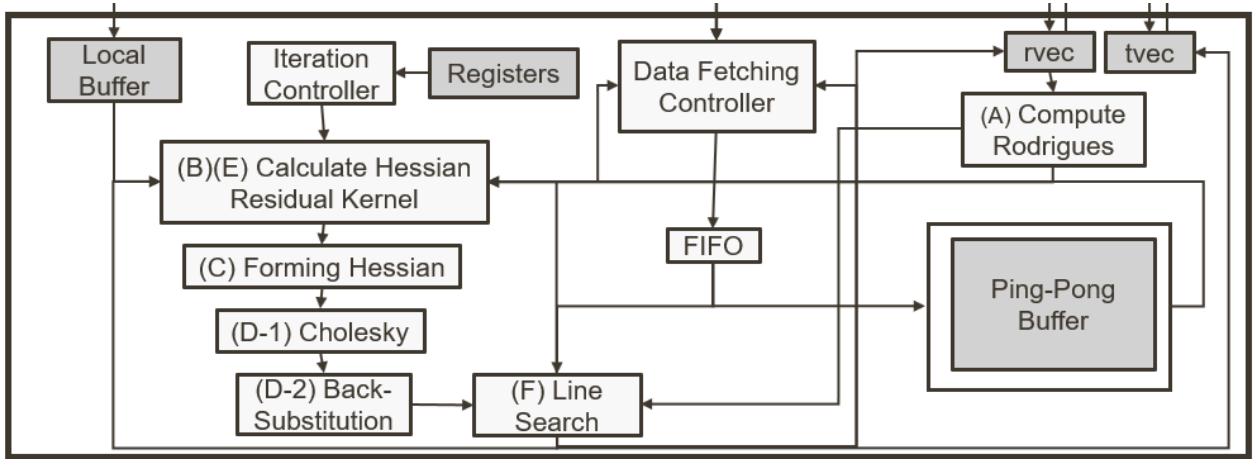
- How general is your hardware?
  - ASIC-like?
  - DSP-like?
- Example (ASIC-like)
- *Applicable to most **D**irect **D**ense **P**hotometric **R**efinement problems (**DDPR**)*
  - *Not restricted to planar or marker objects → General 3D rigid objects*
  - *Suits the front-end refinement of **V**isual **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 Design Flow

- Design Flow
- HLS IP Flow

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

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, ….

1. C-Simulation (SW-Emulation)

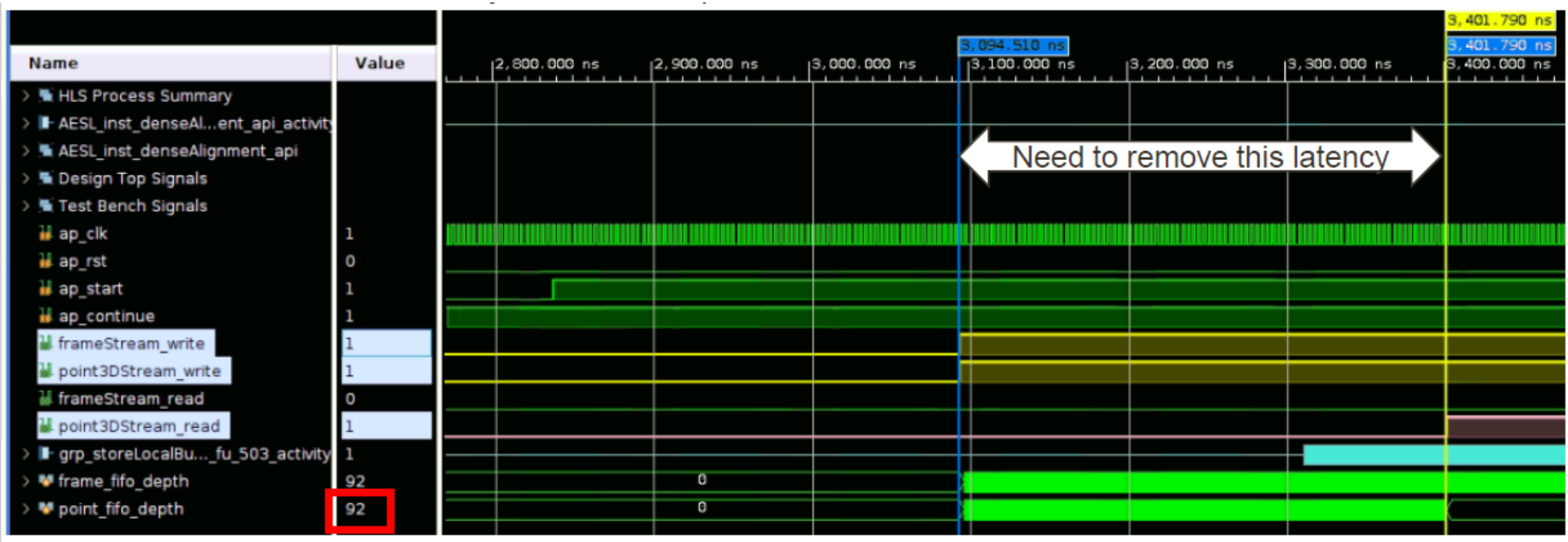
- Verify your C/C++ code with the golden
- Since the hardware is designed in C++, the testbench is also C++.
- It is just like Freshman C/C++ course.

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
  - …

3. Co-Simulation (Hardware-Emulation)

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



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

Lab

Lab Introduction

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

We will only go through Lab1

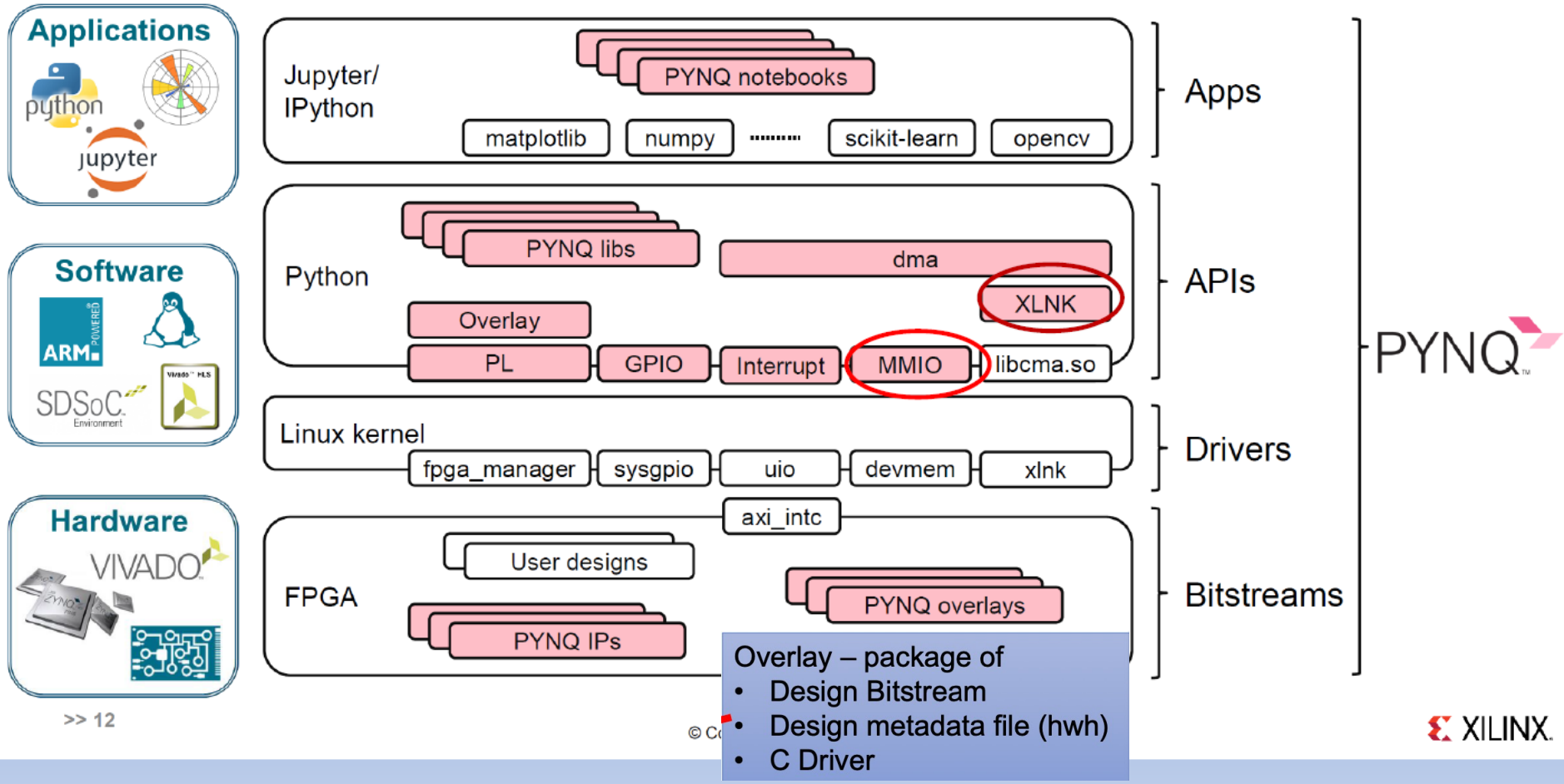
- Lab1, Lab2: Embedded system flow
  - Vitis-hls, Vivado
  - MPSoC FPGAs: Pynq, Ultrascale+

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

AXI

- AXI4
  - **High**-performance memory-mapped requirements.
- AXI-Lite
  - **Low-**throughput memory-mapped communication
- AXI-Stream
  - For high-speed **streaming** data

PYNQ is a Framework



>> 12

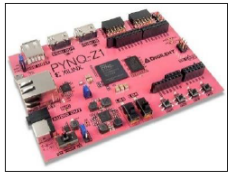
XILINX

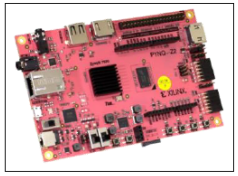
PYNQ Document: https://pynq.readthedocs.io/en/v2.5/getting_started.html#

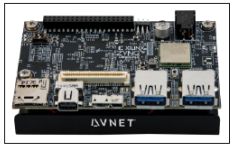
PYNQ-enabled boards

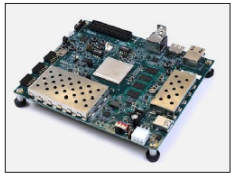
- **Python productivity for Zynq**
  - Open source
  - Build image for other Zynq boards
- **Downloadable SD card image**
  - Zynq 7000
    - PYNQ-Z1 (Digilent)
    - PYNQ-Z2 (TUL)
  - Zynq MPSoC
    - Ultra96 (Avnet)
    - ZCU104 (Xilinx)
  - Zynq RFSoC
    - ZCU111 RFSoC (Xilinx)

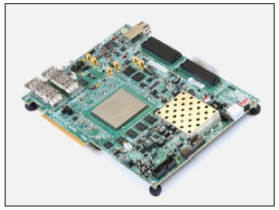
We are using this board


PYNQ-Z1


PYNQ-Z2

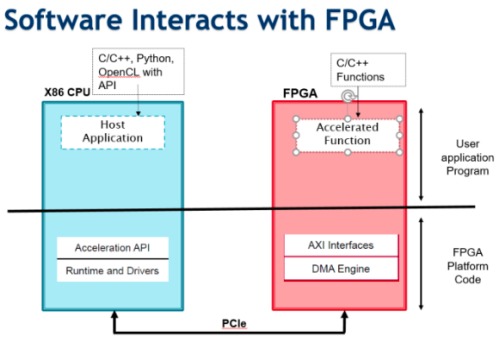

Ultra96

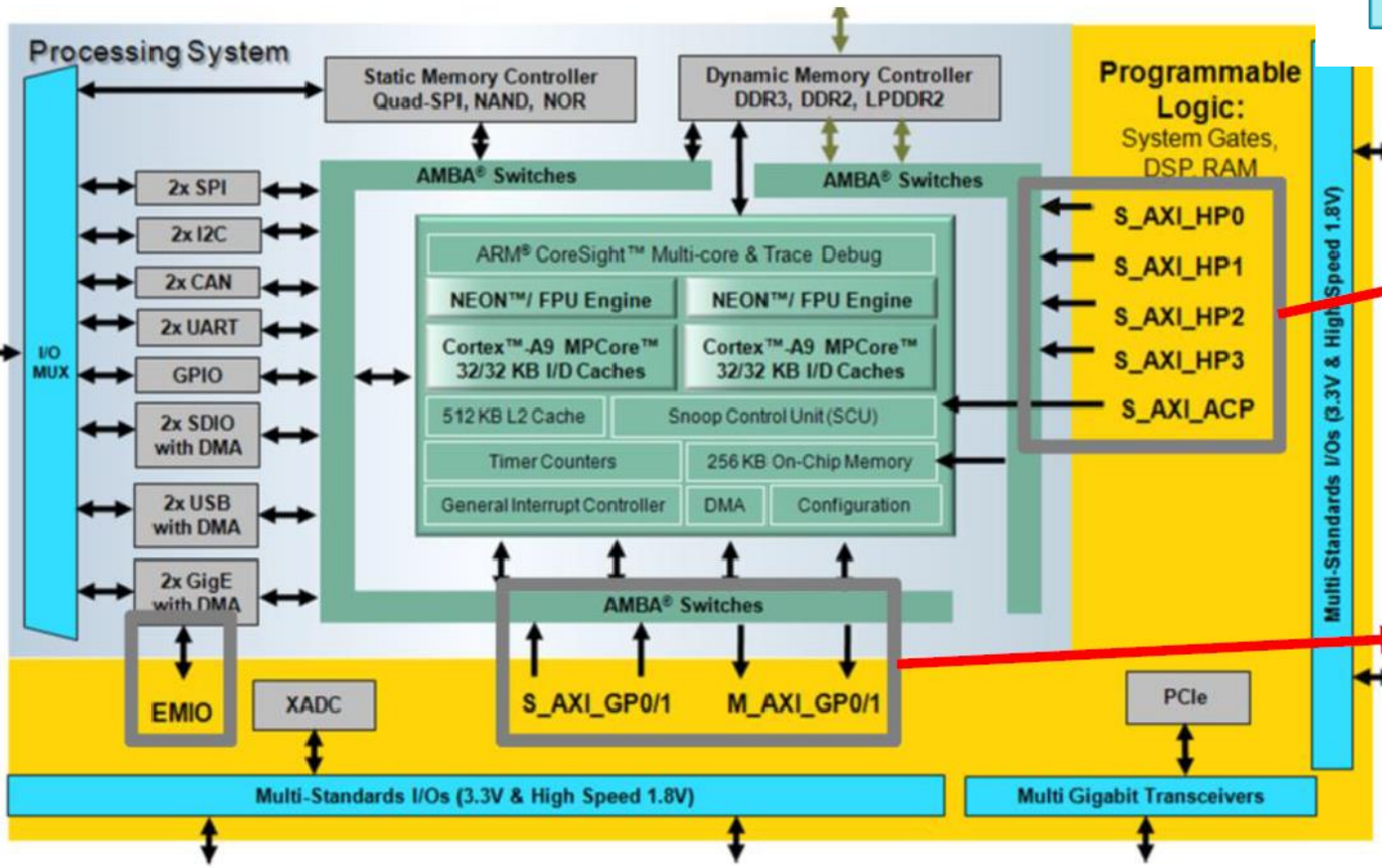

ZCU104


ZCU111

XILINX.

Zynq Block Diagram



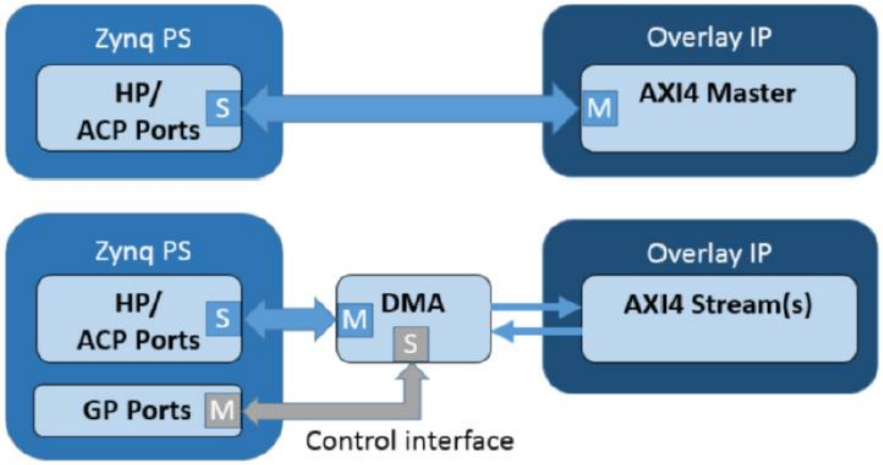


- **AXI High Performance**
  - 4x Slave (64-bit)
    - 1K FIFOs
- **ACP**
  - 1x Slave (64-bit)
  - Cache access

- **AXI General Purpose ports**
  - 2x Master (32-bit)
  - 2x Slave (32-bit)

Zynq PS-PL Interface

- **AXI High Performance**
  - 4x Slave (64-bit)
    - 1K FIFOs
- **ACP**
  - 1x Slave (64-bit)
  - Cache access



- **AXI Master**
  - AXI HP/ACP
  - Typically higher performance IP
- **AXI Stream**
  - Via DMA
  - HP/ACP ports for data path
  - GP slave for control

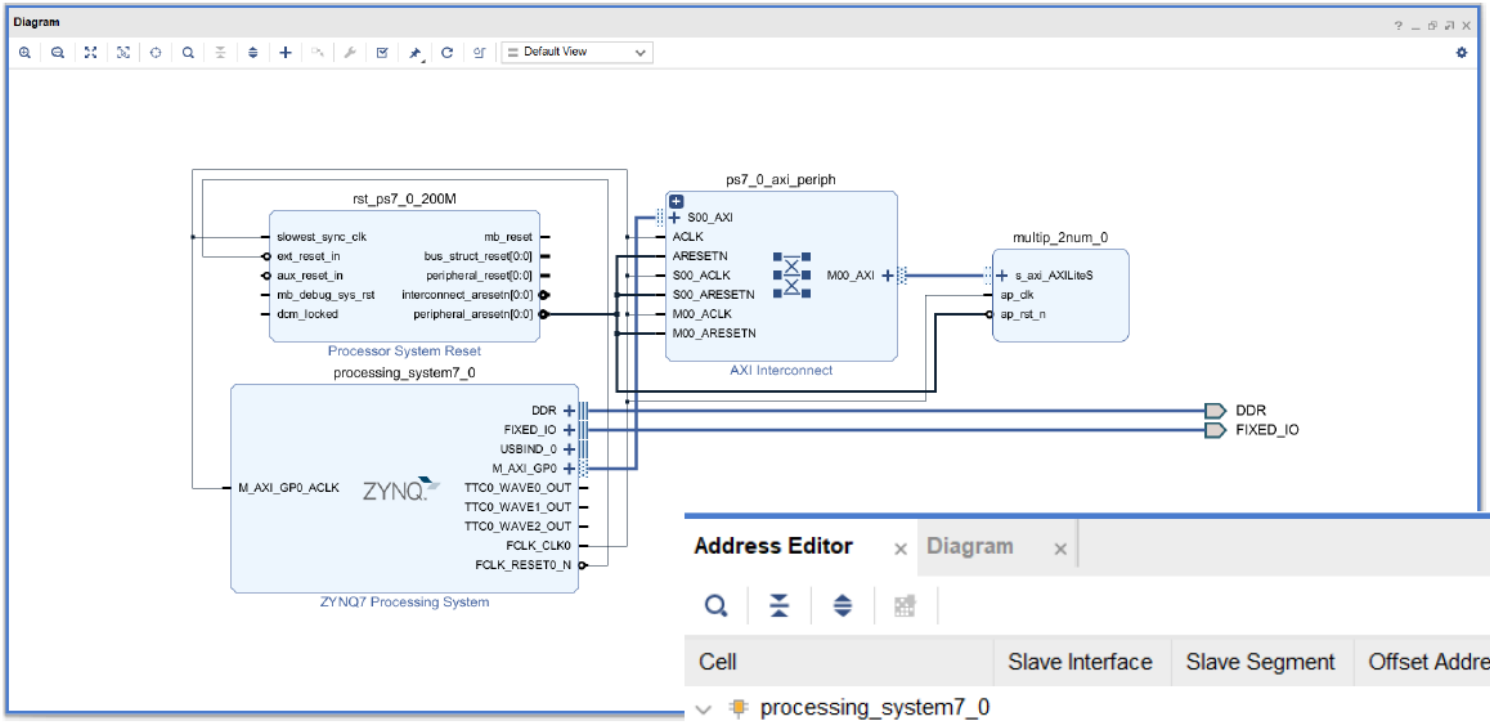
- **AXI General Purpose ports**
  - 2x Master (32-bit)
  - 2x Slave (32-bit)

- **AXI (Lite) Slave IP**
  - General Purpose ports
  - Typically lower performance IP

Lab#1 - Multiplication

```
void multip_2num(int32_t n32In1, int32_t n32In2, int32_t* pn32ResOut)
{
#pragma HLS INTERFACE s_axilite port=pn32ResOut
#pragma HLS INTERFACE s_axilite port=n32In2
#pragma HLS INTERFACE s_axilite port=n32In1
#pragma HLS TOP name=multip_2num

    *pn32ResOut = n32In1 * n32In2;

    return;
}
```

BD for Multiplication AXI-Lite



Address Editor | Diagram

Cell	Slave Interface	Slave Segment	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
multip_2num_0	s_axi_AXILiteS	Reg	0x43C0_0000	64K	0x43C0_FFFF

HWH/HLS Register Map for Multiplication

```
<MODULES>
  <MODULE ... FULLNAME="/multip_2num_0" ...>
    <PARAMETER NAME="C_S_AXI_AXILITES_BASEADDR" VALUE="0x43C00000"/>
    <PARAMETER NAME="C_S_AXI_AXILITES_HIGHADDR" VALUE="0x43C0FFFF"/>
  </PARAMETERS>
```

```
void multip_2num(int32_t n32In1, int32_t n32In2, int32_t* pn32ResOut)
{
#pragma HLS INTERFACE s_axilite port=pn32ResOut
#pragma HLS INTERFACE s_axilite port=n32In2
#pragma HLS INTERFACE s_axilite port=n32In1
#pragma HLS TOP name=multip_2num

    *pn32ResOut = n32In1 * n32In2;

    return;
}
```

Synthesis(solution1)(multip_2num_csynth.rpt) | xmultip_2num_hw.h

```
// ==========================================================
// Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC v2019.2 (64-bit)
// Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
// ==========================================================
// AXILiteS
// 0x00 : reserved
// 0x04 : reserved
// 0x08 : reserved
// 0x0c : reserved
// 0x10 : Data signal of n32In1
//        bit 31~0 - n32In1[31:0] (Read/Write)
// 0x14 : reserved
// 0x18 : Data signal of n32In2
//        bit 31~0 - n32In2[31:0] (Read/Write)
// 0x1c : reserved
// 0x20 : Data signal of pn32ResOut
//        bit 31~0 - pn32ResOut[31:0] (Read)
// 0x24 : Control signal of pn32ResOut
//        bit 0  - pn32ResOut_ap_vld (Read/COR)
//        others - reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

#define XMULTIP_2NUM_AXILITES_ADDR_N32IN1_DATA     0x10
#define XMULTIP_2NUM_AXILITES_BITS_N32IN1_DATA     32
#define XMULTIP_2NUM_AXILITES_ADDR_N32IN2_DATA     0x18
#define XMULTIP_2NUM_AXILITES_BITS_N32IN2_DATA     32
#define XMULTIP_2NUM_AXILITES_ADDR_PN32RESOUT_DATA 0x20
#define XMULTIP_2NUM_AXILITES_BITS_PN32RESOUT_DATA 32
#define XMULTIP_2NUM_AXILITES_ADDR_PN32RESOUT_CTRL 0x24
```

Host Code for Multiplication

- Import MMIO (Involving in Overlay)
- Define memory mapped region
  - The base/offset is defined in hwh file
- Read and Write 32-bit values
- In read function, it shall check the ap_vld (0x24)

```
ol = Overlay("/home/xilinx/IPBitFile/Multip2Num.bit")
regIP = ol.multip_2num_0

for i in range(9):
    print("============================")
    for j in range(9):
        regIP.write(0x10, i + 1)
        regIP.write(0x18, j + 1)
        Res = regIP.read(0x20)
        print(str(i + 1) + " * " + str(j + 1) + " = " + str(Res))
```

Let’s Start