EPFL

Lab. On HW-SW Digital Systems Codesign
EE-390(a)

Session 2
Introduction to co-design with high-level synthesis (HLS)

Prof. David Atienza
Dr. Denisa Constantinescu, Dr. Miguel Peón-Quirós
Mr. Rubén Rodríguez-Álvarez, Ms. Stasa Kostic, Mr. Karan Pathak

Session objectives

- Introduction to high-level synthesis (HLS)
- Levels of optimization and parallelism
- Mapping of function arguments to peripheral ports
- Examples of peripherals in HLS
- Integration with SW in Linux
- Memory & DSP resources in the Zynq 7000 FPGA family

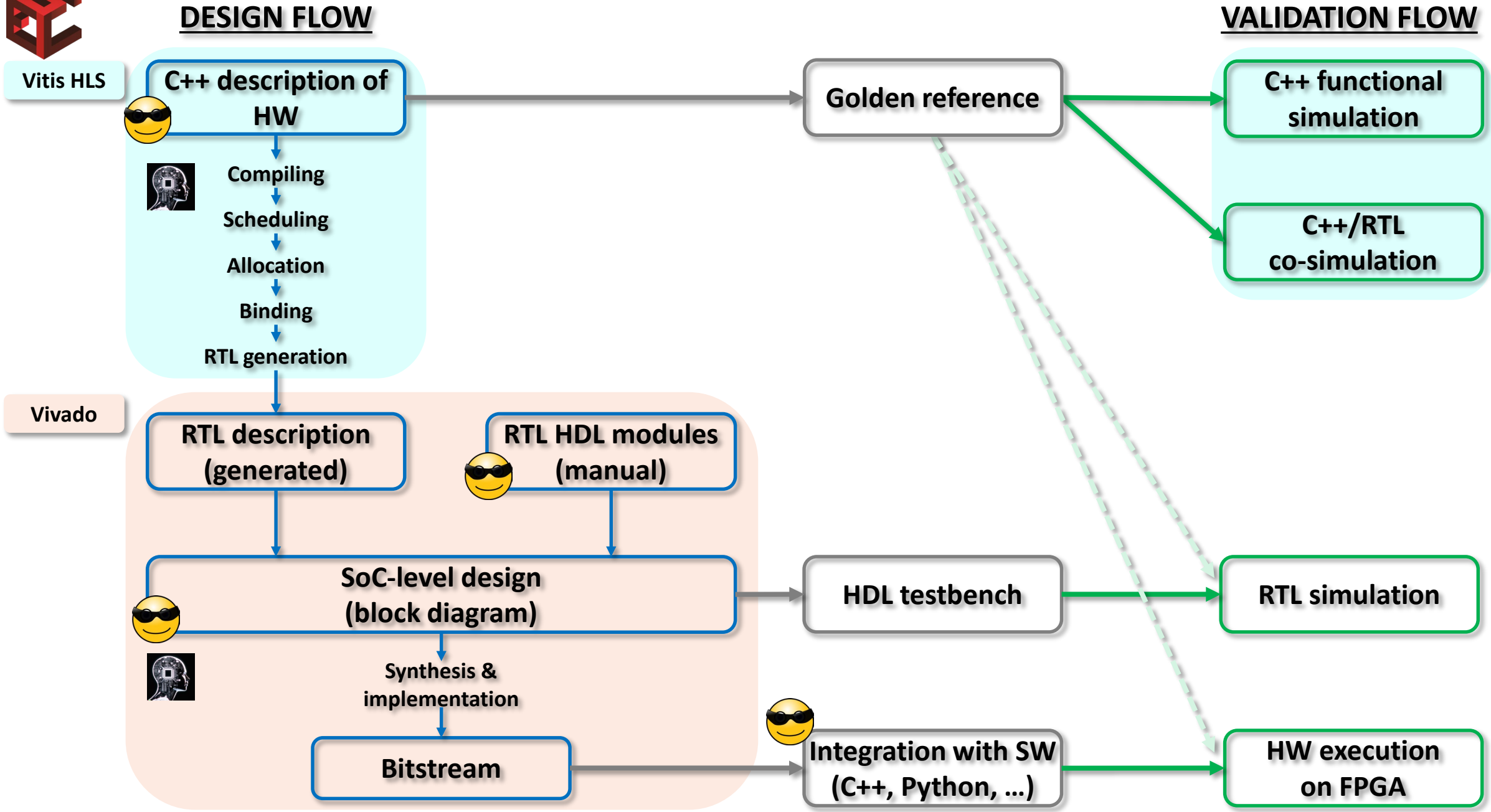
Introduction to high-level synthesis (HLS)

What is high-level synthesis (HLS)?

- HLS is an automated process that takes a behavioral specification of a system and generates a register-transfer level (RTL) structure that implements the specified behavior[1]
- HLS increases the level of abstraction for the description of digital circuits
  - In HDL, we describe a system in terms of finite-state machines and register-level transfers in a datapath
  - In HLS, we can describe a system at the algorithmic and dataflow levels
    - We can write a loop and the tool will automatically:
      - generate a datapath with the required number and type of operators,
      - produce the scheduling of operations over cycles, and
      - infer the FSMs
      - If we change the optimization criteria, e.g., number of pipeline stages, the tool will automatically adjust the FSM
- The designer describes the macro-architecture of the algorithm in C/C++
  - Focusing on the design purpose and its interactions with other components
- Micro-architectural decisions such as FSMs, datapaths, register pipelines, etc. are left to the HLS tool
  - Enables easy exploration of different microarchitectures from the same microarchitecture description

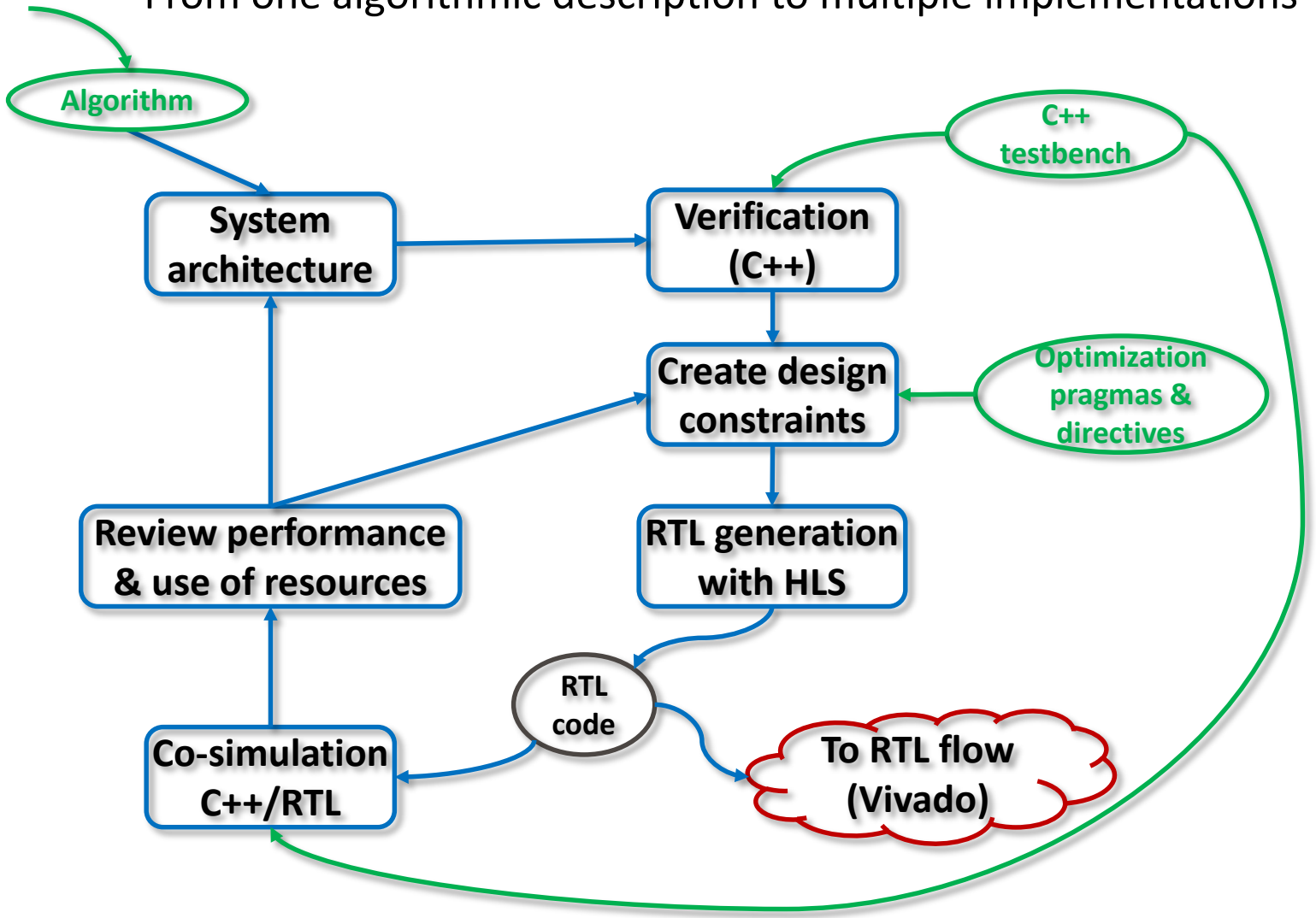
[1] Vitis High-Level Synthesis User Guide – UG1399

High-level synthesis design flow for HW accelerators



Design exploration with HLS

- HLS enables faster exploration of microarchitecture optimizations
  - From one algorithmic description to multiple implementations

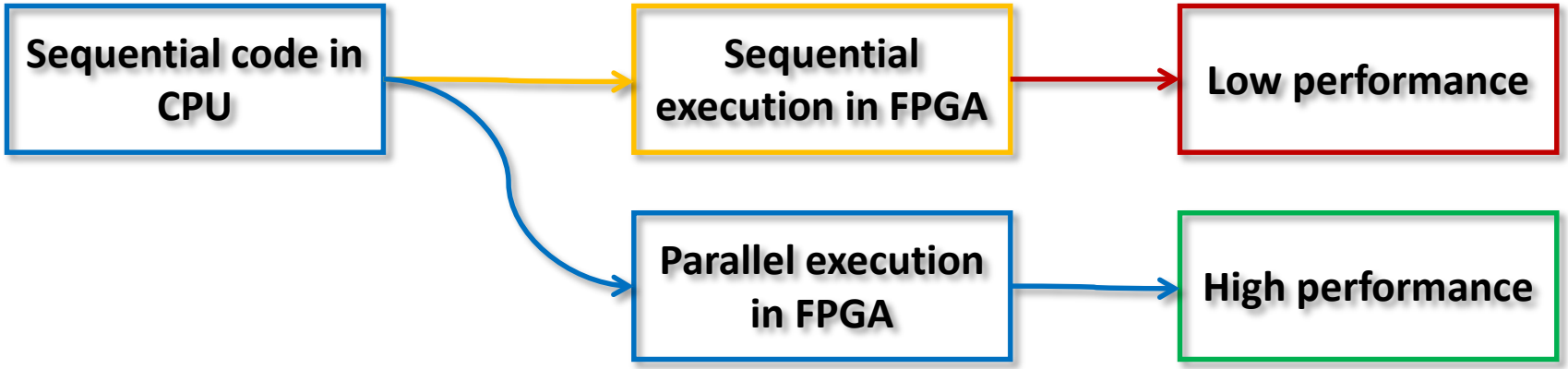


1. Design the architecture
2. Describe it algorithmically in C/C++
3. Verify functionality at behavioral level
4. Use HLS tools to generate the RTL implementation for a given clock speed and input constraints
5. Verify the functionality of the generated RTL
6. Explore different microarchitectures changing HLS optimization pragmas and directives

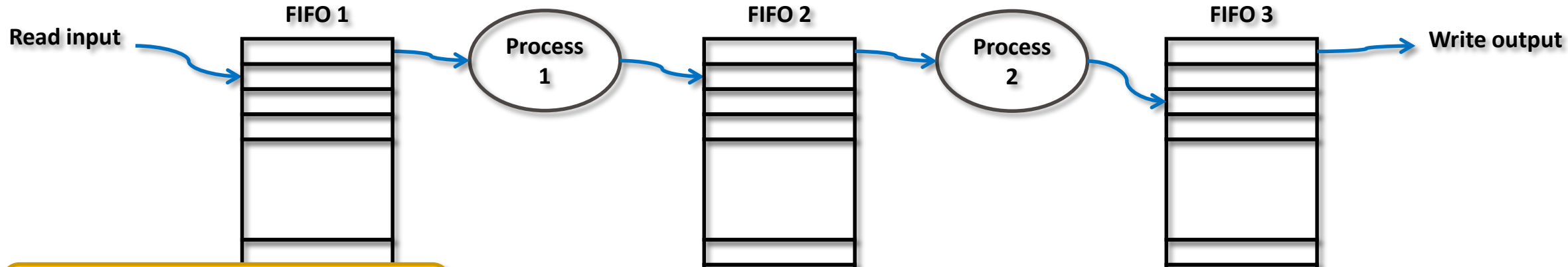
The SW implementation of an algorithm and its HW description in HLS may require very different C/C++ code to be efficient

Tips to write efficient HW descriptions using HLS

- The HLS tool must be able to infer parallelism from sequential code



- Producer-consumer paradigm

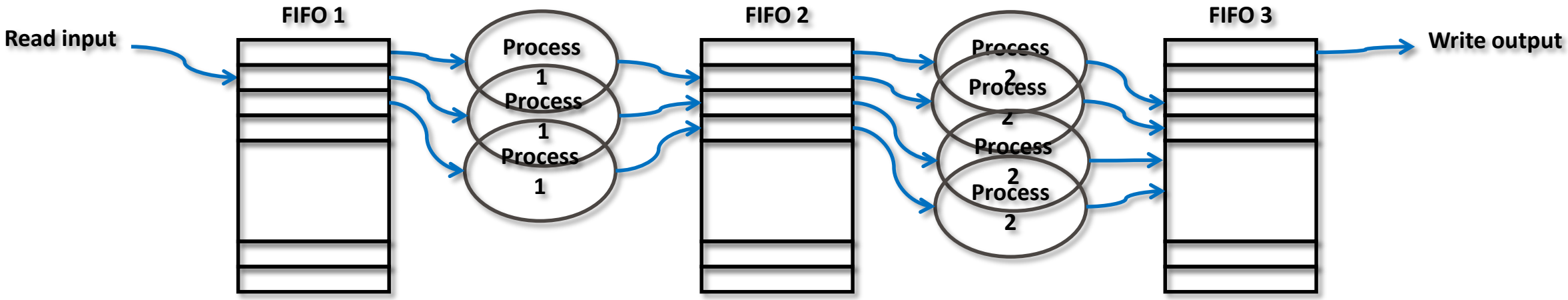


Tip: Favor consumer-producer architectures. Then, analyze the temporal interactions and the flow of data/tasks

This paradigm enables task parallelism and decoupling of execution times at the highest design level

Consumer-producer paradigm in HLS

- Producer-consumer paradigm



- Enables decoupling of tasks and pipelining on different data items.
  - On 1 CPU → No increase of performance
  - On n CPUs → ~ Linear speed-up up to some limit (coarse granularity)
  - On FPGA → Each process can be replicated (fine granularity)
    → Each process can also be internally pipelined

Three levels of parallelism:
1) **Independent tasks that work on each step independently**
2) **Multiple tasks at each step**
3) **Each task can be pipelined to have multiple data elements "on-the-fly"**

Levels of optimization and parallelism

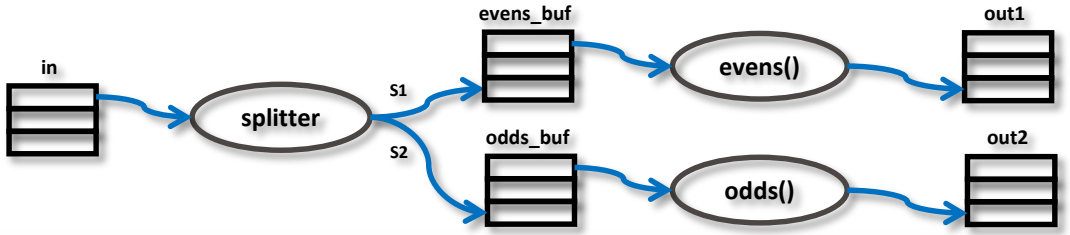
Levels of optimization in HLS

- Macro-architectural level optimizations
  - Data-driven parallelism
  - Control-driven parallelism
  - Mixed control/data-driven parallelism
  - Specified by the block-level control protocol

- Micro-architectural level optimizations
  - Loop optimizations
    - Pipelining
    - Unrolling (pipelining implies unrolling)
    - Merge/fusion
  - Array optimization & reshaping
    - Loop optimizations determine access pattern
    - If array is in top-level interface → Control signals to interface with external memory
    - If array is internal to the design → Control signals + memory model for the RTL tool
  - Function optimizations
    - Task-level parallelism
    - Separation into load-compute-store subfunctions to enable parallelism

Data-driven parallelism

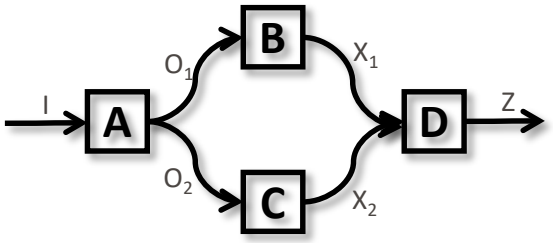
- Using task and stream abstractions from Xilinx HLS libraries
  - A task is a function that is executed infinitely → HW module that processes an input as soon as it's available
    - Represented with hls::task
    - No explicit calls to the function
  - Streams provide read() and write() methods through hls::stream<T>
  - In SW, equivalent to a thread executing a function, with input and output channels or FIFOs
- Useful when not interacting with SW, no explicit start/stop conditions, just flow of data




```
20  void odds_and_evens(hls::stream<int> &in, hls::stream<int> &out1,
21                      hls::stream<int> &out2) {
22    hls_thread_local hls::stream<int> s1; // channel connecting t1 and t2
23    hls_thread_local hls::stream<int> s2; // channel connecting t1 and t3
24    // t1 infinitely runs function splitter, with input in and outputs s1 and s2
25    hls_thread_local hls::task t1(splitter, in, s1, s2);
26    // t2 infinitely runs function odds, with input s1 and output out1
27    hls_thread_local hls::task t2(odds, s1, out1);
28    // t3 infinitely runs function evens, with input s2 and output out2
29    hls_thread_local hls::task t3(evens, s2, out2);
30  }
```

```
1   #include "test.h"
2
3   void splitter(hls::stream<int> &in, hls::stream<int> &odds_buf,
4                 hls::stream<int> &evens_buf) {
5     int data = in.read();
6     if (data % 2 == 0)
7     evens_buf.write(data);
8     else
9     odds_buf.write(data);
10  }
11
12  void odds(hls::stream<int> &in, hls::stream<int> &out) {
13    out.write(in.read() + 1);
14  }
15
16  void evens(hls::stream<int> &in, hls::stream<int> &out) {
17    out.write(in.read() + 2);
18  }
```

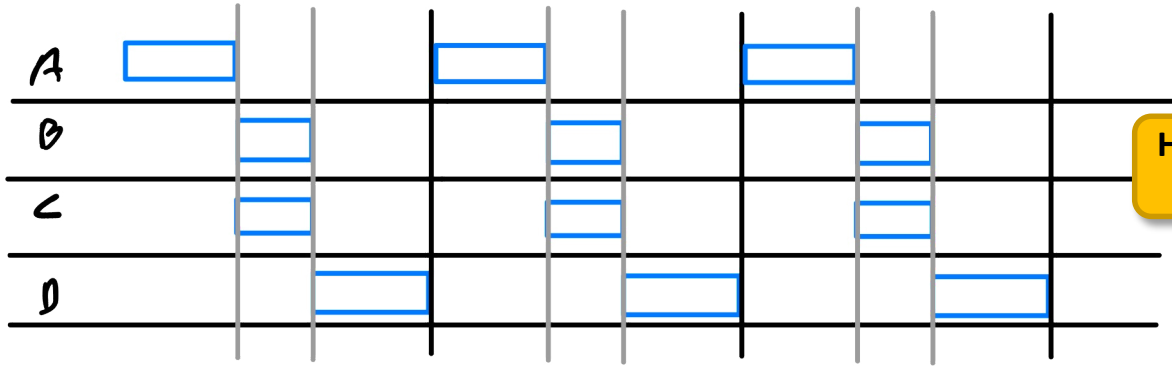
Control-driven parallelism



- Useful when the system has start/stop signals
  - E.g., accelerator synchronization with SW
- Takes a series of sequential function calls
  - Creates a task-level pipeline architecture of concurrent processes
- Multiple sequential functions can be started simultaneously
  - A function can be restarted before it finishes (pipelining with respect to itself)

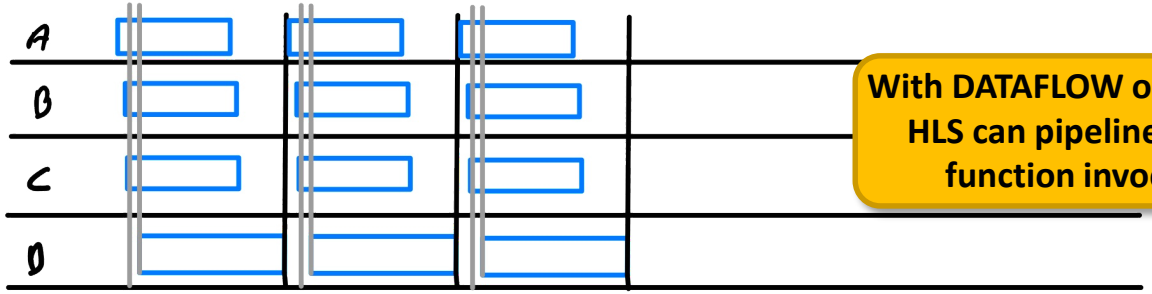
```
A(I, O1, O2);
B(O1, X1);
C(O2, X2);
D(X1, X2, Z);
```

Let's assume that the complete process is invoked 3 times



HLS will normally be able to parallelize B and C

  - A subsequent function can start before the previous finishes (pipelining to chain functions)



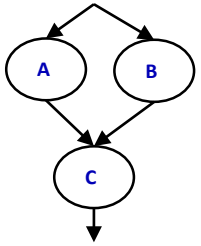
With DATAFLOW optimization, HLS can pipeline multiple function invocations

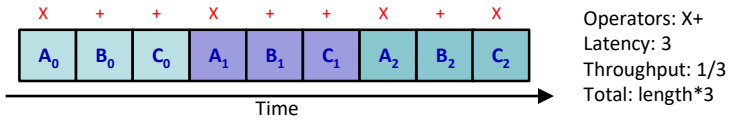
E.g., process 3 independent images, each one analyzed row-by-row. The functions can be pipelined by rows for each image

Loop optimization primer

- Consider the following loop:

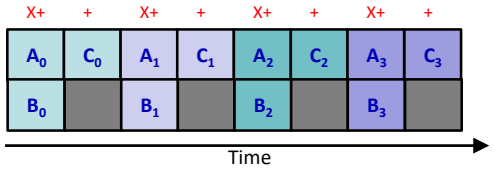
```
for (i = 0; i < length; ++ i) {
      A: a = in[i] * 2;
      B: b = in[i] + 5;
      C: out[i] = a+b;
}
```





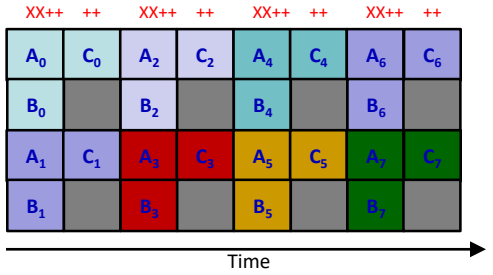
Operators: X+
Latency: 3
Throughput: 1/3
Total: length*3

A)Parallelizing inner operations:



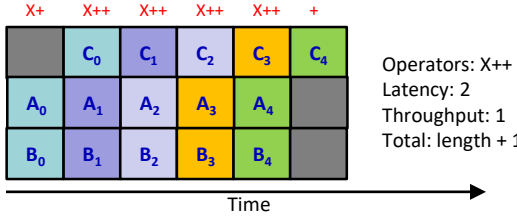
Operators: X+
Latency: 2
Throughput: 1/2
Total: length*2

B) Parallelizing + unrolling inner operations:



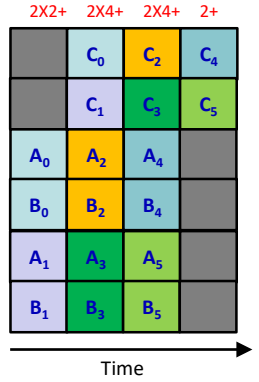
Operators: XX++
Latency: 2
Throughput: 2/2
Total: ceil(length/2)*2

C)Parallelizing + pipelining iterations



Operators: X++
Latency: 2
Throughput: 1
Total: length + 1

D) Parallelizing + unrolling + pipelining



Operators: XX++++
Latency: 2
Throughput: 2
Total: ceil(length/2) + 1

Mapping of function arguments to peripheral ports

Functions in Vitis HLS

- A function is a HW component that can be instantiated multiple times
  - Each function is implemented as a HW component with ports and start/end signals
  - Function calls → component interactions, can be parallelized
    - Recursion is not supported
  - Inlining fuses functionality and use of resources

- Task-level parallelism is implemented at the function level
  - I.e., in general, Vitis HLS infers parallelism only between function calls
  - Loops that should execute in parallel have to be pushed into separate functions
    - Sequential loops can be pipelined
  - Re-architect functions into load-compute-store subfunctions to enable parallelism

Basic mapping of resources

- Function's local data is private
  - Functions share data only over ports
  - Local variables are converted into registers (flip-flops) or memories (e.g., BRAM)
    - E.g., shift registers are automatically inferred, e.g., when elements of a local array are displaced in a loop
- Arrays are converted into:
  - Registers
  - Memory as local storage (e.g., BRAM)
    - Multiple options (pragmas) to control the type of resource: BRAM, distributed RAM, etc.
  - Memory as global storage (e.g., DRAM)
- No support for dynamic memory
  - The designer needs to know the amount of memory used by the algorithm

Mapping of top-level arguments to peripheral ports

- A register file is exposed through a slave AXI4 interface
  - Assuming control-driven paradigm over AXI4 buses
- Scalar arguments are converted into registers in the register file
- Arrays or pointers to arrays are converted into:
  - One register in the register file to store the array address (pointer value)
  - Logic to implement an AXI4 master to access the array
- Depending on the block-level control paradigm chosen
  - HLS generates signals for start/done control of the module

```
#include <stdint.h>
#include <ap_int.h>

ap_uint<32> TopFunction(ap_uint<32> length, ap_uint<32> * input)
{
#pragma HLS INTERFACE s_axilite port=length
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE m_axi port=input offset=slave

  ap_uint<32> result = 0;

  for (uint32_t ii = 0; ii < length; ++ ii)
    result += input[ii];

  return result;
}
```

Creates an entry in the RF for this scalar argument

Creates an entry in the RF for the result

Specifies that the module will have a control schema based on control/status registers visible in the AXI4 bus

Creates a master interface to retrieve the data of the vector, and an entry in the RF to specify the pointer address

Mapping of top-level arguments to peripheral ports

- A register file is exposed through a slave AXI4 interface
  - Assuming control-driven paradigm over AXI4 buses
- Scalar arguments are converted into registers in the register file
- Arrays or pointers to arrays are converted into:
  - One register in the register file to store the array address (pointer value)
  - Logic to implement an AXI4 master to access the array
- Depending on the block-level control paradigm chosen
  - HLS generates signals for start/done control of the module

```
#include <stdint.h>
#include <ap_int.h>

ap_uint<32> TopFunction(ap_uint<32> length, ap_uint<32> * input)
{
#pragma HLS INTERFACE s_axilite port=length
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE m_axi port=input offset=slave

  ap_uint<32> result = 0;

  for (uint32_t ii = 0; ii < length; ++ ii)
    result += input[ii];

  return result;
}
```

Vitis HLS generates automatically a file (e.g., xadder_hw.h) with the register and control blocks definition

```
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/COH)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read/COR)
//        bit 7  - auto_restart (Read/Write)
//        bit 9  - interrupt (Read)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0 - enable ap_done interrupt (Read/Write)
//        bit 1 - enable ap_ready interrupt (Read/Write)
//        others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0 - ap_done (Read/TOW)
//        bit 1 - ap_ready (Read/TOW)
//        others - reserved
// 0x10 : Data signal of ap_return
//        bit 31~0 - ap_return[31:0] (Read)
// 0x18 : Data signal of length_r
//        bit 31~0 - length_r[31:0] (Read/Write)
// 0x1c : reserved
// 0x20 : Data signal of input_r
//        bit 31~0 - input_r[31:0] (Read/Write)
// 0x24 : Data signal of input_r
//        bit 31~0 - input_r[63:32] (Read/Write)
// 0x28 : reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write,
//  COH = Clear on Handshake)
```

Examples of peripherals in HLS

Example I: Combinational adder

```c
#include <stdint.h>
#include <stdio.h>
#include "adder.h"

int main(int , char ** )
{
    uint32_t a = 5, b = 3, res;

    Adder(a, b, res);
    printf("Result: %u\n", res);


    return 0;
}
```

```c
#include <cstdint>
#include "adder.h"

void Adder(uint32_t a, uint32_t b, uint32_t &res)
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=res

    res = a + b;
}
```

```c
// ==============================================================
// Vitis HLS - High-Level Synthesis from C, C++ and OpenCL v2020.2 (64-bit)
// Copyright 1986-2020 Xilinx, Inc. All Rights Reserved.
// ==============================================================
// control
// 0x00 : reserved
// 0x04 : reserved
// 0x08 : reserved
// 0x0c : reserved
// 0x10 : Data signal of a
//        bit 31~0 - a[31:0] (Read/Write)
// 0x14 : reserved
// 0x18 : Data signal of b
//        bit 31~0 - b[31:0] (Read/Write)
// 0x1c : reserved
// 0x20 : Data signal of res
//        bit 31~0 - res[31:0] (Read)
// 0x24 : Control signal of res
//        bit 0  - res_ap_vld (Read/COR)
//        others - reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

#define XADDER_CONTROL_ADDR_A_DATA   0x10
#define XADDER_CONTROL_BITS_A_DATA   32
#define XADDER_CONTROL_ADDR_B_DATA   0x18
#define XADDER_CONTROL_BITS_B_DATA   32
#define XADDER_CONTROL_ADDR_RES_DATA 0x20
#define XADDER_CONTROL_BITS_RES_DATA 32
#define XADDER_CONTROL_ADDR_RES_CTRL 0x24
```

- Top level accelerator function: Adder()
  - Two ports to receive operands
  - One port to return the result
- Ports implemented as AXI4 registers in a slave interface
- The module does not implement any specific control protocol (ap_ctrl_none)[1]
  - It's a purely combinational circuit where res is continuously updated

[1] A single valid signal is implemented as a clear-on-read (COR) bit. This enables chaining a consumer and a producer using that single signal as flow control.

Example II: Vector adder

```
1  #include <ap_int.h>
2
3  void AddVectors(ap_uint<32> * input1, ap_uint<32> * input2,
4                  ap_uint<32> * output,
5                  ap_uint<32> length, ap_uint<32> accum)
6  {
7  #pragma HLS INTERFACE s_axilite port=length
8  #pragma HLS INTERFACE s_axilite port=accum
9  #pragma HLS INTERFACE s_axilite port=return
10 #pragma HLS INTERFACE m_axi depth=1024 port=input1 offset=slave
11 #pragma HLS INTERFACE m_axi depth=1024 port=input2 offset=slave
12 #pragma HLS INTERFACE m_axi depth=1024 port=output offset=slave
13
14   if (length > 8*1024*1024)
15     length = 8*1024*1024;
16
17   for (ap_uint<32> ii = 0; ii < length; ++ ii) {
18     output[ii] = input1[ii] + input2[ii] + accum;
19   }
20 }
```

Xilinx offers integer types with arbitrary precision to help adjusting the bitwidth of the datapaths

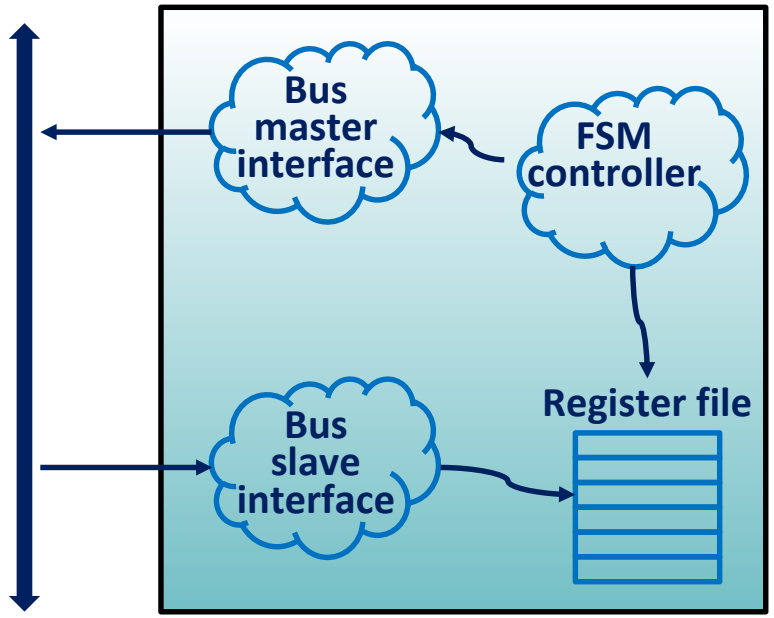
An array or pointer parameter is translated into:
- **One register in the slave interface for the address**
- **One master port**

If the protocol for the function name or "return" is s_axilite, start/done/ready/idle signals are implemented in an AXI4 register, and the corresponding control FSM is generated

- An FSM is created to read values, perform the additions and write the results using external ports
  - A datapath with enough adders to obtain an initiation interval of 1 is automatically created
- Vitis HLS adds the logic to control the start/done/idle/ready signals. The module starts working when the "start" bit in the control register is set to 1
- We can "bundle" the ports of the arrays to decide if they use one master interface each, or if some of them share an AXI interface. Add `bundle=name` to the `m_axi` lines as required

Result of the synthesis of the vector adder

- The accelerator contains a main controller
  - The controller is connected to the registers to react to commands
    - The values of the registers can be connected directly as wires to the FSM
  - The controller can write into the registers, which will be read by the processor in the future
- The main controller implements the task of the accelerator
  - Using master interfaces to access memories or other peripherals



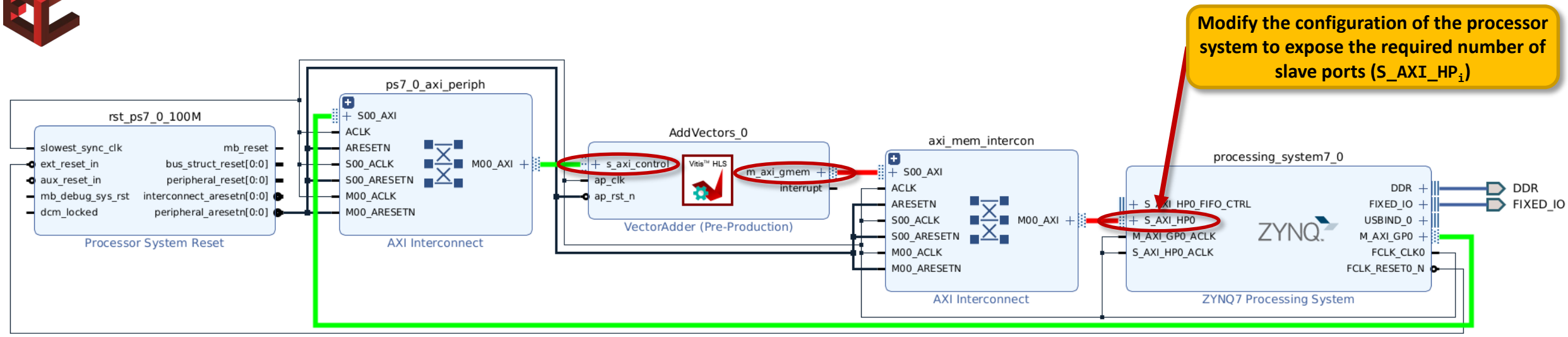
Index	Name
0	Control & status
1	Global interrupt enable register
2	IP interrupt enable register
3	IP interrupt status register
4	Address of input1
7	Address of input2
10	Address of output
13	Length
15	Accumulator

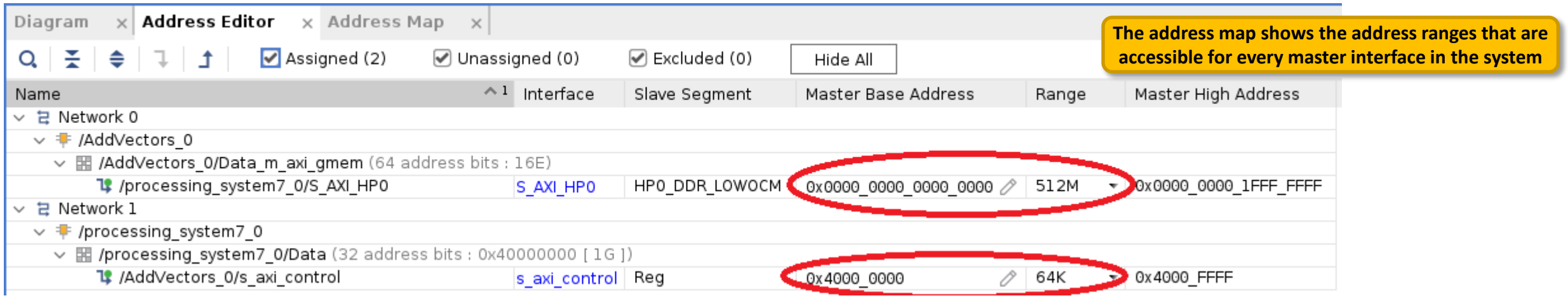
How do we know the offset of the registers?

How do we know the bit positions of the start/done/idle/ready signals in the control & status register?

Integration with SW in Linux

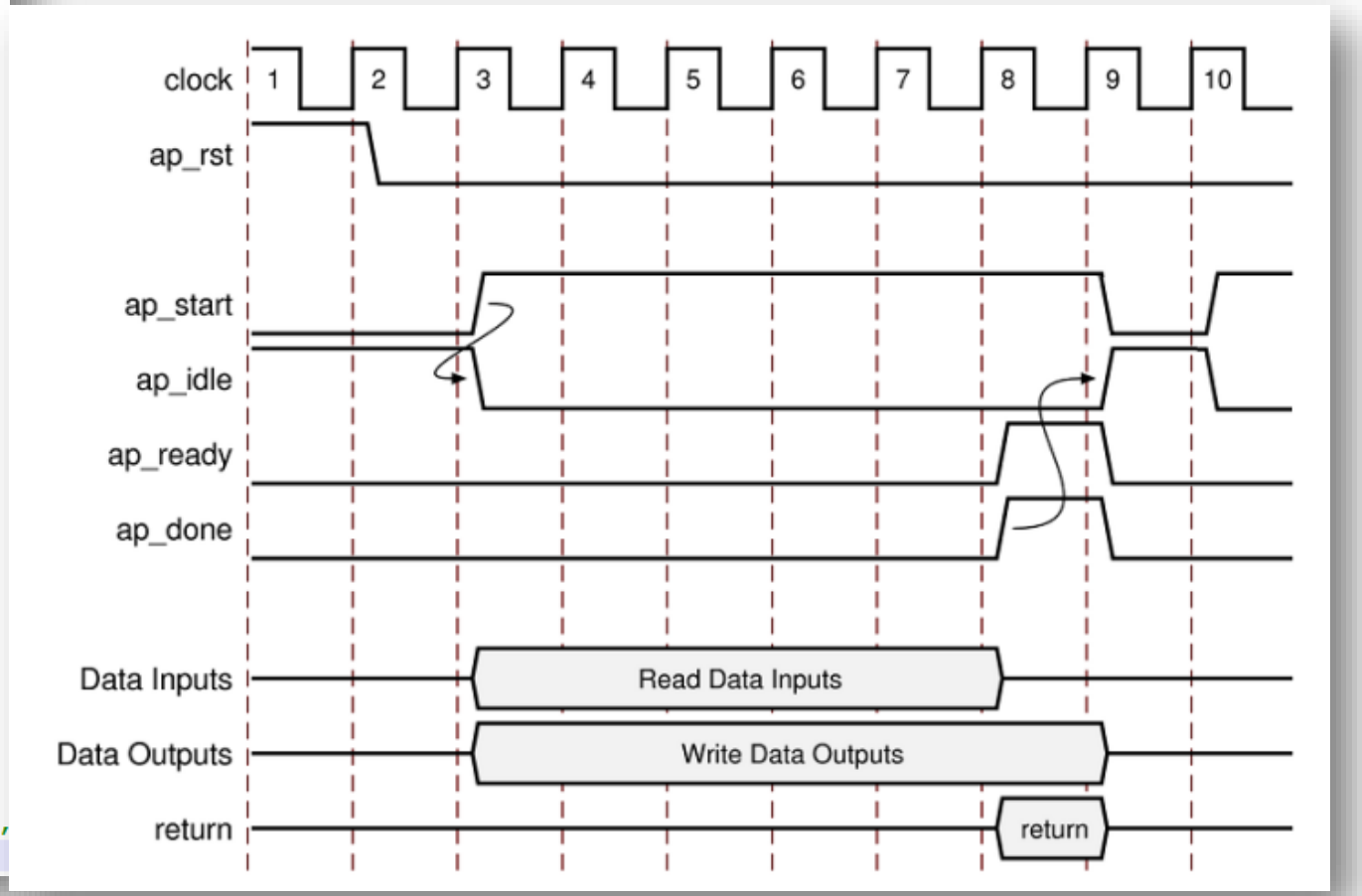
Integration of the vector adder in a SoC





HW-SW handshake protocol for AXI4 peripherals

```
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/COH)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read/COR)
//        bit 7  - auto_restart (Read/Write)
//        bit 9  - interrupt (Read)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0 - enable ap_done interrupt (Read/Write)
//        bit 1 - enable ap_ready interrupt (Read/Write)
//        others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0 - ap_done (Read/TOW)
//        bit 1 - ap_ready (Read/TOW)
//        others - reserved
// 0x10 : Data signal of input1
//        bit 31~0 - input1[31:0] (Read/Write)
// 0x1c : Data signal of input2
//        bit 31~0 - input2[31:0] (Read/Write)
// 0x28 : Data signal of output_r
//        bit 31~0 - output_r[31:0] (Read/Write)
// 0x34 : Data signal of length_r
//        bit 31~0 - length_r[31:0] (Read/Write)
// 0x3c : Data signal of accum
//        bit 31~0 - accum[31:0] (Read/Write)
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write,
//  COH = Clear on Handshake)
```



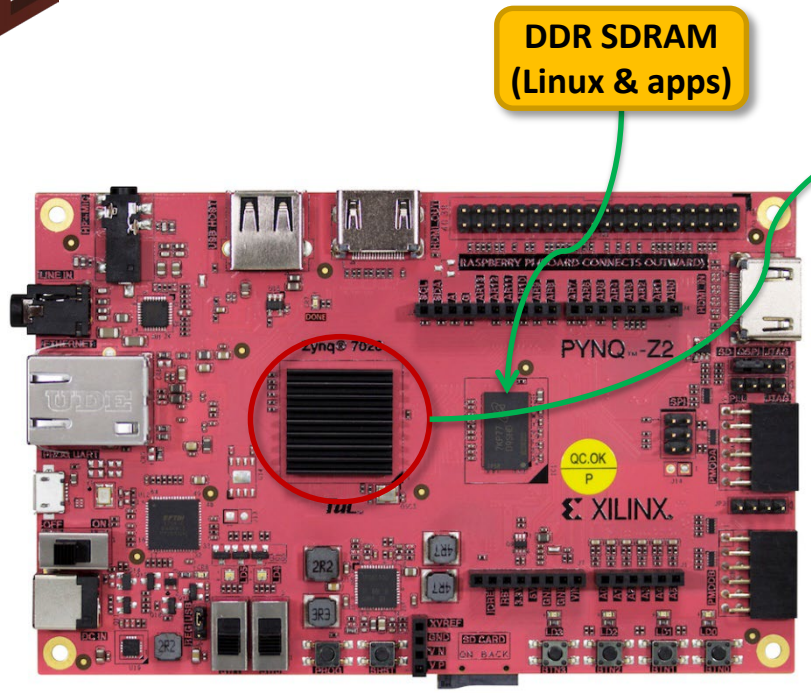
- In SW, we use the control signals of the register at offset 0
  - Program the input registers
  - Write '1' into bit 0 (LSB) to start the accelerator
  - Wait until '1' is read from bit 1 (done) → **This resets start**
  - Access the results

Alternatively, we can connect the interrupt line to the processor and enable the interrupts with the global and IP interrupt enable registers

- ap_done is COR, so the done signal can only be read once!
- Reading ap_done completes the handshake, which clears ap_start (COH: clear-on-handshake)

Memory & DSP resources in the Zynq 7000 FPGA family

Storage resources in the Pynq board



Inside the Zynq 7020 FPGA

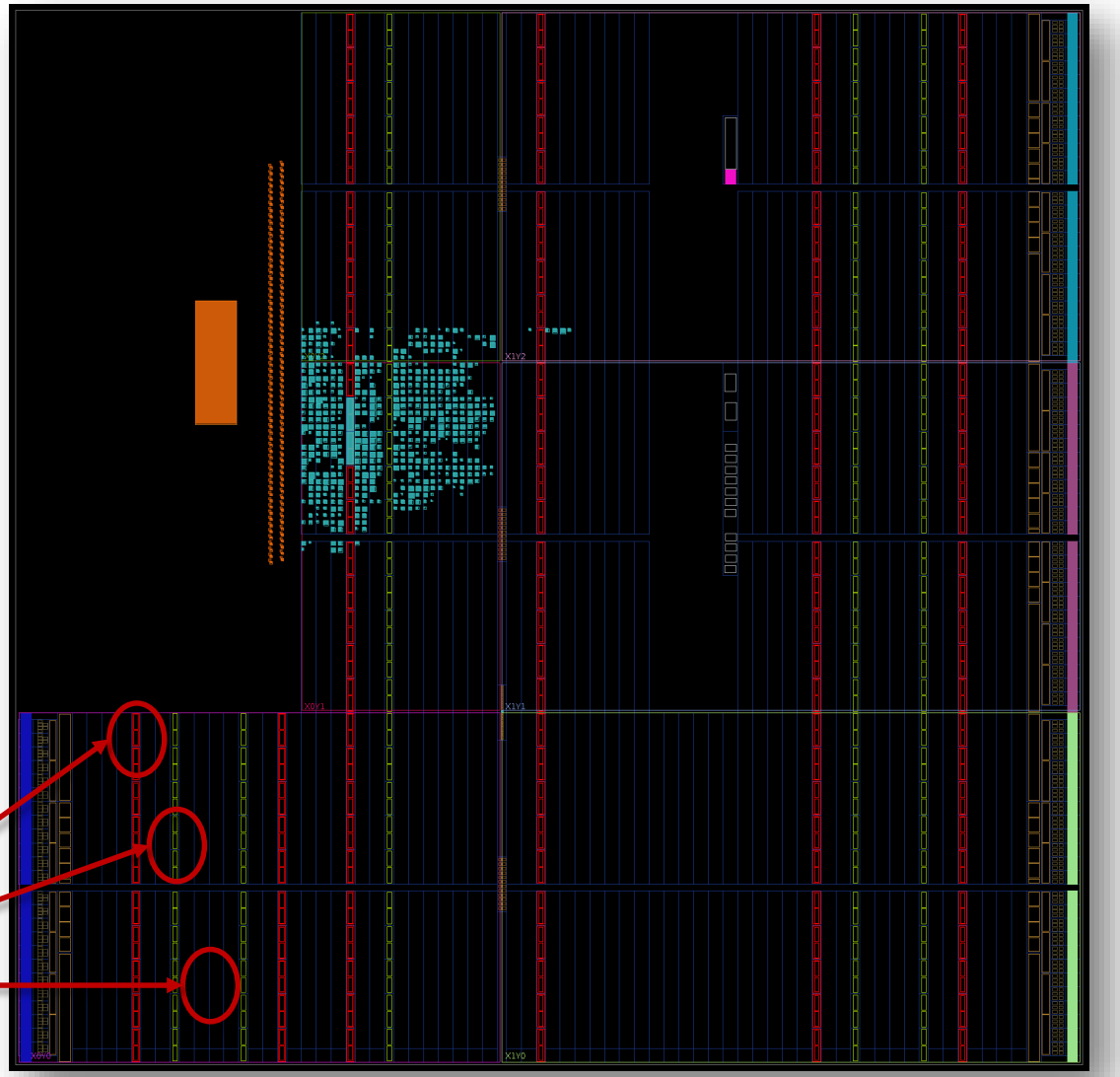
- Registers (flip-flops, FF)
  - Synchronous write
  - Continuous read (multiplexers)
- Distributed RAM (LUTRAM)
  - Implemented in the lookup tables of the slices → Consume logic resources!
  - Asynchronous
- Block RAM (BRAM)
  - Implemented as small SRAMs in the FPGA fabric → Don't have reset!
  - Synchronous write (0 cycles latency)
  - Synchronous read (1 cycle latency!!!)
- Vivado can infer the memory resource type from the HDL description
  - We can also guide HLS to use one specific type of resource

BRAMs offer vast bandwidth

- The Z-7020 has 140 BRAMs
  - 1024*4*140 = 573 440 B → 560 KiB
- Each of the 140 blocks has 2 ports
  - 140 blocks * 2 ports * 4 B * 100 MHz = 834 GB/s! (> 1.5 TB/s at 200 MHz!)
- BRAMs, logic cells and DSPs are organized in columns across the FPGA fabric

Place logic close to storage

Enables parallel computation on data

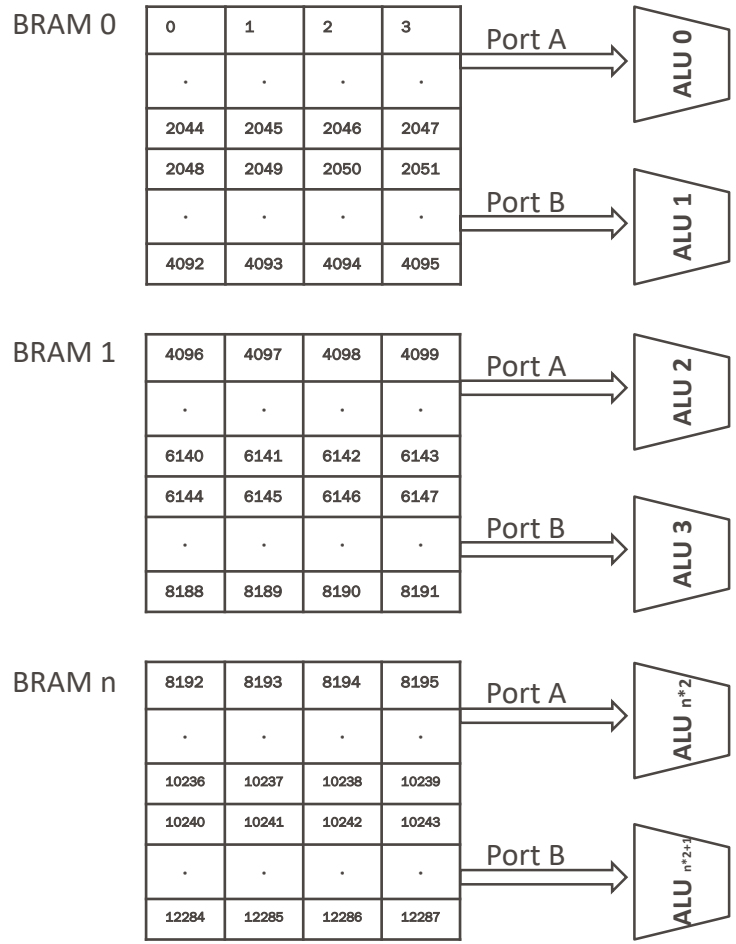


BRAM characteristics

- Xilinx FPGAs contain hundreds (or thousands) of small synchronous static RAMs (SRAM)
- BRAMs are dual-port memories
  - Each port is totally independent
- Each BRAM of 38 Kbits can be configured as:
  - 1024 words of 32 (36) bits
  - 2048 words of 16 (18) bits
  - 4096 words of 8 (9) bits
  - Also: 32Kx1, 16Kx2, 8Kx4
  - Each port can use a different configuration!
- Each BRAM can also be configured as
  - 2 independent BRAMs of 18 Kbits

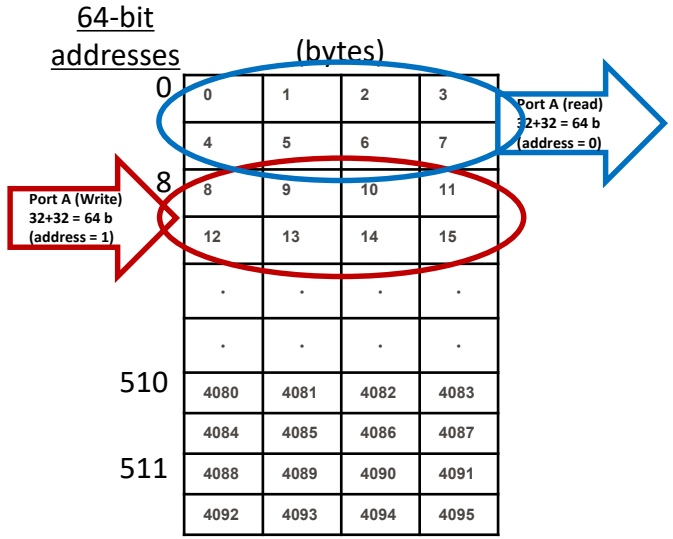
Each BRAM block contains additional bits that can be used as data or ECC

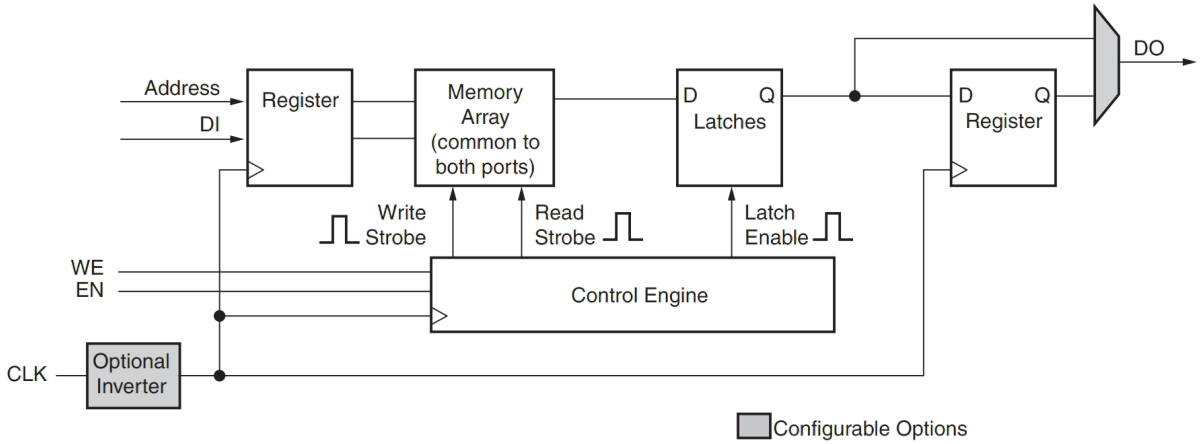
BRAM blocks contain logic to implement ECC without additional resources



BRAMs are dual port memories

- Each port can perform independent reads and writes, for a total of 4 operations per cycle
  - What happens if one port reads and writes from/to the same address?
    - Conflicts possible if both ports access to the same address simultaneously
- The ports can be “linked” to double the BRAM width:
  - 1 read/write port of 512 words of 64 (72) bits --- or 64Kx1
- Each port can have its own clock
- BRAMs contain output registers to reduce critical paths
- BRAMs also implement FIFO functionality





Source: Xilinx UG473

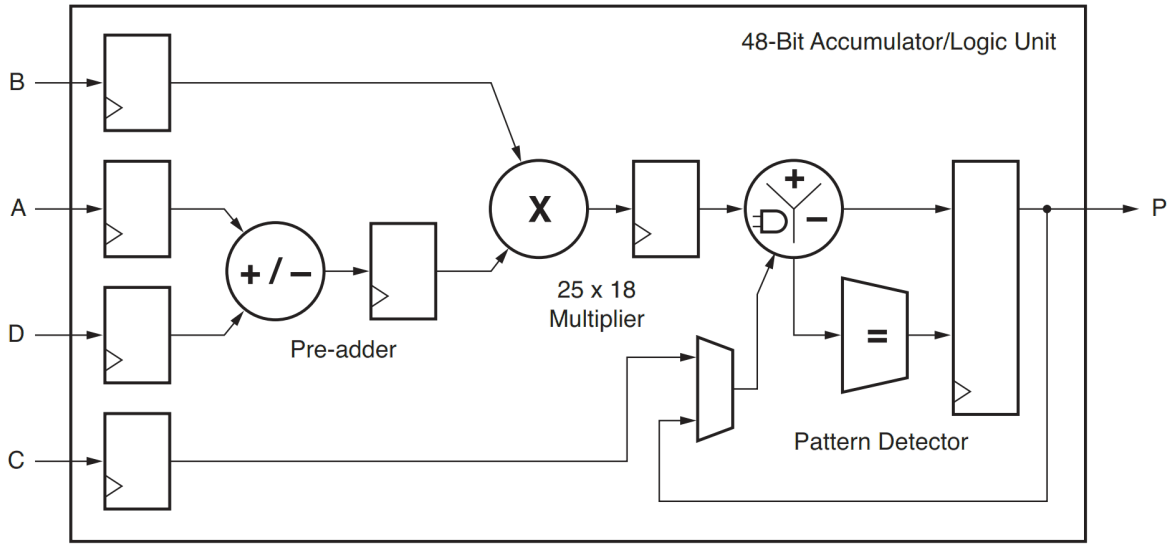
BRAMs are synchronous memories

- Writes happen on the next clock edge
  - Before the next clock edge:
    - writeAddr becomes stable with the address to write to
    - dataIn becomes stable with the data to write
    - writeEn becomes stable to ‘1’
  - At the clock edge, the input data is written into the memory array

- Reads happen on the next clock edge; data are available one cycle later!
  - Before the next clock edge:
    - readAddr becomes stable with the address to read from
    - If present, readEn becomes stable to ‘1’
  - After the next clock edge:
    - After the memory access time, the contents of the designated address become available on dataOut

DSP slices

- Multipliers and accumulators can be implemented with dedicated HW blocks
  - Our Z-7020 has 220 DSP slices
- Each DSP slice can implement:
  - 25x18 multiplier
  - 48-bit accumulator
  - SIMD arithmetic (2x24-bit or 4x12-bit) for add/subtract/accumulate
  - Other logic functions (see docs)
- DSP slices implement pipelining and can be cascaded using dedicated buses
- HLS will infer automatically the use of DSPs
  - If sizes and operations match DSP functionality



Source: Xilinx UG479

Bibliography (in Moodle)

- Vitis High-Level Synthesis User Guide – UG1399 (v2022.2, Dec. 7th, 2022)
  - Heavily quoted during this session
  - Read carefully Sections I-III of the document
  - Use Sections IV, VI as reference as needed
- "Zynq-7000 SoC Technical Reference Manual (UG585)"
  - https://docs.xilinx.com/v/u/en-US/ug585-Zynq-7000-TRM
- "7 series FPGAs memory resources user guide (UG473)"
  - https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf
- "7-series DSP48E1 slice user guide (UG479)"
  - https://docs.xilinx.com/v/u/en-US/ug479_7Series_DSP48E1

Supported data types in HLS

- Vitis HLS supports (mostly) all standard C/C++ data types
  - `signed/unsigned char, short, int, long, float, double`
    - Also: `int8_t, int16_t, int32_t, int64_t`
  - Partial support for IEEE-754
    - HLS synthesis will respect the order of FP operations → May block some optimizations
  - Support of `math.h` operations
    - `std::complex<double>` is not supported
- Vitis HLS also supports integer types of arbitrary precision
  - Useful to reduce the bitwidth of ports and hence of the datapath operators
  - Integer types:

```
#include “ap_int.h”
ap_[u]int<W>, with 1 ≤ W ≤ 1024
ap_uint<14> count;
```

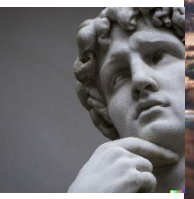
  - Fixed point types:

```
#include “ap_fixed.h”
ap_[u]fixed<W,I,Q,O,N>
ap_fixed<> cnnWeights[SIZE];
```

Value	Meaning (ap_fixed)
W	Word length in bits
I	Integer bits to the left of the (binary) point
Q	Quantization mode. Can be AP_RND, AP_RND_ZERO, AP_RND_MIN_INF, AP_RND_INF, AP_RND_CONV, AP_TRN, AP_TRN_ZERO – Check the documentation!
O	Overflow mode: AP_SAT, AP_SAT_ZERO, AP_SAT_SYM, AP_WRAP, AP_WRAP_SM
N	Number of saturation bits in overflow wrap modes

SIMD can be implemented using the type hls::vector<T, N>

All hls::vector operations are mapped on parallel HW (best if both, bitwidth of T and N, are powers of 2)



Questions?

Prof. David Atienza

EPFL – Embedded Systems Laboratory
david.atienza@epfl.ch