

FPGA Programmierung mit



Studierendentage 2026

23.03 – 27.03.2026

Fachbereich Physik

Ruprecht-Karls-Universität Heidelberg

About me



Sebastian Dittmeier
PostDoc

Experimental
Particle Physics

- Started working with FPGAs > 10 years ago
- PhD in Physics 2018 (Heidelberg)
- Also studied in Heidelberg before
- Focus: Trigger and data acquisition system for particle physics experiments

- Trigger developments for DEAP3600
- Pixel sensor readout and testbeam telescope systems in the context of Mu3e
- Hardware track trigger and FPGA accelerated track reconstruction for ATLAS



How does this course work?

- Daily sheet of attendance → attend at least 4/5 days to get 1 CP
- Slides in English → Should we talk in German or English?
- Code examples used in the labs @ https://github.com/sdittmeier/hls_course
- Slides and code linked on the [website](#) of this course

- For the labs: we will need ssh connection to curvaton@physi.uni-heidelberg.de with X11 forwarding → for Windows you can use e.g. [MobaXterm](#)
- Account log-in information → distribution on paper
- Time limited access for the duration of this course
- You should work in groups of 2 students

What can you expect in the coming days?

Monday, 23.03.2026

What is an FPGA? And why physicists care about them!
Motivation, Architecture, Electronics, Programming Model, Basics of HLS

Tuesday, 24.03.2026

Thinking in Hardware
Timing Terminology, Paradigms, HLS Data Types, Pointers, Streams
Lab: Matrix Multiplication, Pointers, and Streams

Wednesday, 25.03.2026

Interfaces and Improving Performance
Interfaces, Directives, Improving Throughput and Latency, Bottlenecks
Lab: YUV-Filter, Loops Optimizations

Thursday, 26.03.2026

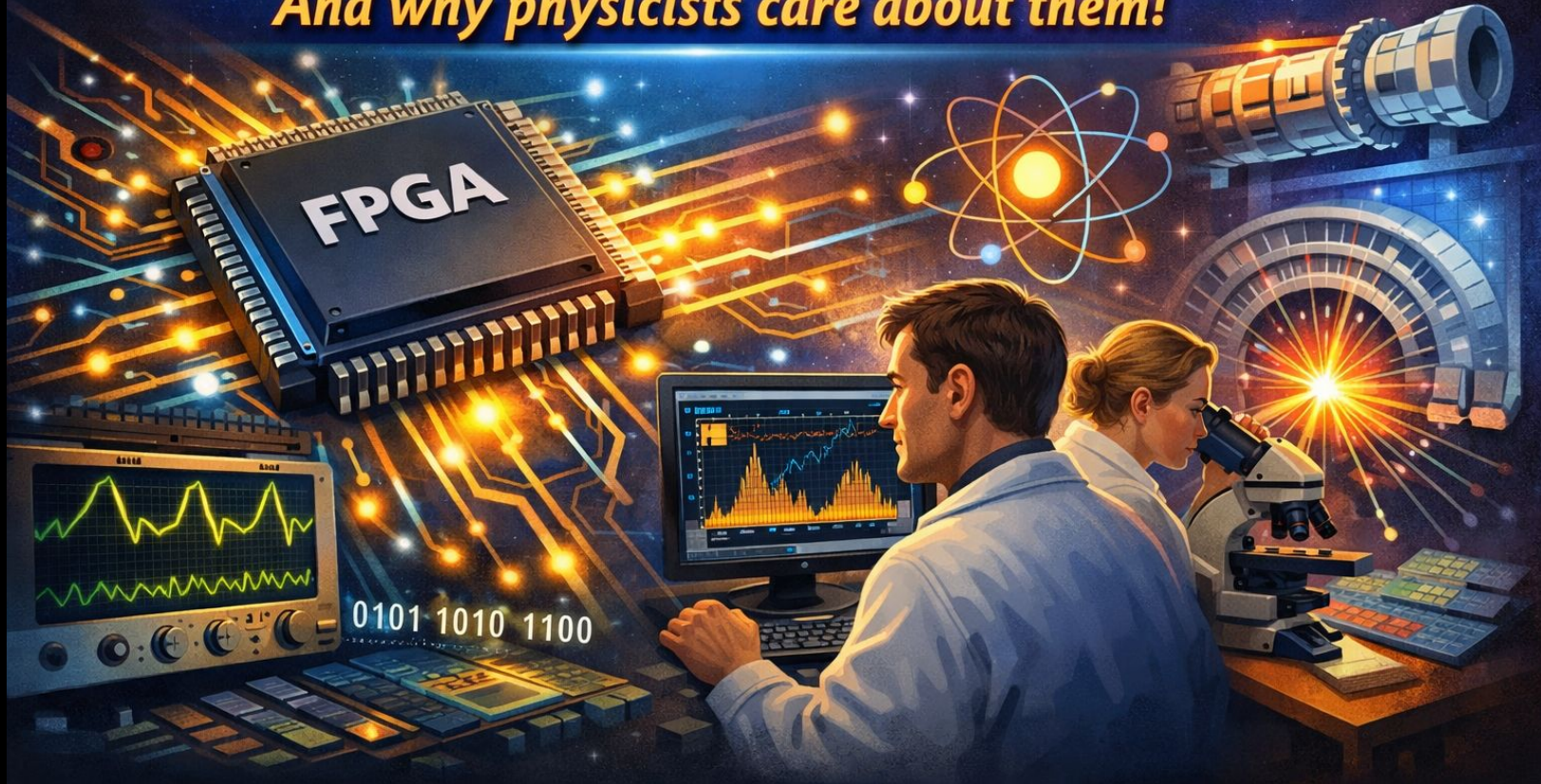
Improving Area and Resources
Allocation & Structure Control, Hierarchies, Arbitrary Precision
Lab: DCT, Matrix Multiplication revisited

Friday, 27.03.2026

Task Level Parallelism, and how to deal with Timing Violations
Lab: Freely running kernel, Timing violation

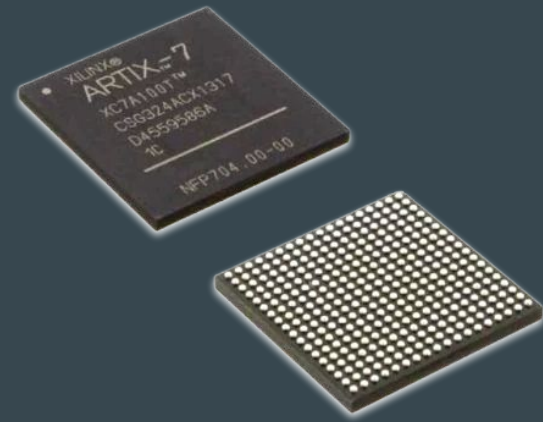
What is an **FPGA**?

And why physicists care about them!

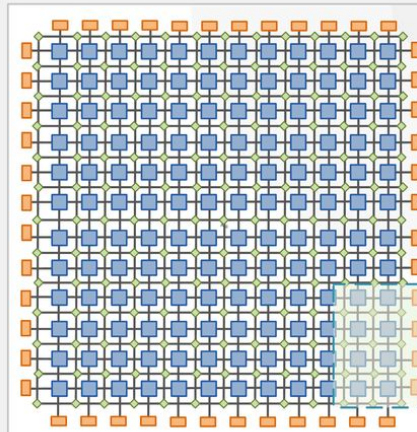


Field Programmable Gate Array

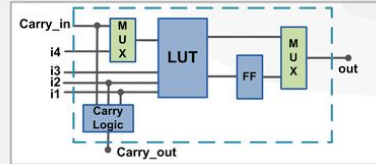
- Digital electronic device with many pins
- **Programmable** hardware
 - No function without YOUR “program”
 - The algorithm you program IS the processor
- Hardware can be (re-)programmed after manufacturing in “the **field**”
- Historically made from logic **gates** (AND, OR, XOR, ...)
- Organized as an **array** of many identical programmable blocks



a) Island type FPGA architecture

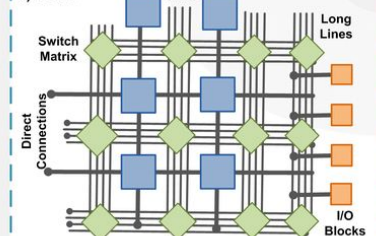


c) Basic architecture of a CLB



[source](#)

b) Zoom



Applications of FPGAs in industry



Telecommunications

- 5G/6G base stations
- Signal processing, beamforming, error correction



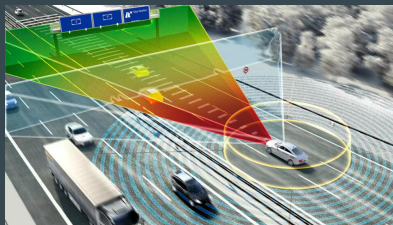
Data centers & cloud computing

- Network acceleration (SmartNICs)
- Compression, encryption, packet filtering



Finance & high-frequency trading

- Ultra-low latency decision making
- Microseconds = money



When ultra-low **latency**, strict **determinism**, and massive **parallelism** are critical, **FPGAs** are the solution.



Which device should I pick for my task?



CPU



GPU



FPGA



ASIC

Device comparison



Aspect

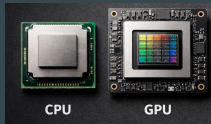
CPU

Full name	Central Processing Unit
Primary purpose	General-purpose computing
Flexibility	★★★★ Very high
Performance (for specific tasks)	★ Moderate
Parallelism	★ Low - moderate
Power efficiency	★ Moderate
Latency	★★ Moderate
Programming	★★★★ Easy (C/C++, Python, etc.)
Reconfigurable after manufacture	✓ Yes
Development cost	★★★★ Low
Time to market	★★★★ Fastest
Typical use cases	OS, apps, control logic

Use cases in High Energy Physics

High-level trigger,
offline analysis

Device comparison



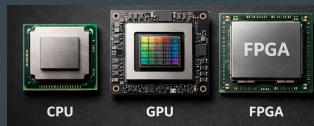
Aspect

CPU

GPU

Full name	Central Processing Unit	Graphics Processing Unit
Primary purpose	General-purpose computing	Massively parallel computation
Flexibility	★★★★ Very high	★★★ High
Performance (for specific tasks)	★ Moderate	★★ High (parallel tasks)
Parallelism	★ Low - moderate	★★★★★ Very high
Power efficiency	★ Moderate	★★ Moderate - high
Latency	★★ Moderate	★ High (batch-oriented)
Programming	★★★★★ Easy (C/C++, Python, etc.)	★★★ Moderate (CUDA, OpenCL)
Reconfigurable after manufacture	✓ Yes	✓ Yes
Development cost	★★★★★ Low	★★★ Low - medium
Time to market	★★★★★ Fastest	★★★ Fast
Typical use cases	OS, apps, control logic	ML, graphics, simulations
Use cases in High Energy Physics	High-level trigger, offline analysis	High-level trigger, offline analysis, especially ML

Device comparison



Aspect

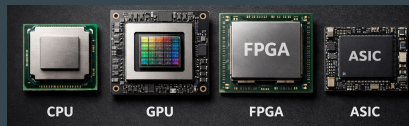
CPU

GPU

FPGA

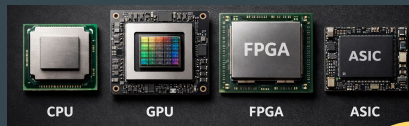
Full name	Central Processing Unit	Graphics Processing Unit	Field-Programmable Gate Array
Primary purpose	General-purpose computing	Massively parallel computation	Custom hardware logic (reconfigurable)
Flexibility	★★★★ Very high	★★★ High	★★ Medium (reprogrammable)
Performance (for specific tasks)	★ Moderate	★★ High (parallel tasks)	★★★★ Very high
Parallelism	★ Low - moderate	★★★★★ Very high	★★★★ High (custom pipelines)
Power efficiency	★ Moderate	★★ Moderate - high	★★★★ High
Latency	★★ Moderate	★ High (batch-oriented)	★★★★ Very low
Programming	★★★★★ Easy (C/C++, Python, etc.)	★★★★ Moderate (CUDA, OpenCL)	★★ Moderate - Hard (HLS, Verilog/VHDL)
Reconfigurable after manufacture	✓ Yes	✓ Yes	✓ Yes
Development cost	★★★★★ Low	★★★★ Low - medium	★★ Medium
Time to market	★★★★★ Fastest	★★★★ Fast	★★ Medium
Typical use cases	OS, apps, control logic	ML, graphics, simulations	Networking, DSP, low-latency systems
Use cases in High Energy Physics	High-level trigger, offline analysis	High-level trigger, offline analysis, especially ML	Level-1 trigger systems, High-level trigger co-processors

Device comparison



Aspect	CPU	GPU	FPGA	ASIC
Full name	Central Processing Unit	Graphics Processing Unit	Field-Programmable Gate Array	Application-Specific Integrated Circuit
Primary purpose	General-purpose computing	Massively parallel computation	Custom hardware logic (reconfigurable)	Fixed, task-specific hardware
Flexibility	★★★★ Very high	★★★ High	★★ Medium (reprogrammable)	★ Very low
Performance (for specific tasks)	★ Moderate	★★ High (parallel tasks)	★★★★ Very high	★★★★★ Highest
Parallelism	★ Low - moderate	★★★★★ Very high	★★★★ High (custom pipelines)	★★★★★ Custom (optimized)
Power efficiency	★ Moderate	★★ Moderate - high	★★★★ High	★★★★★ Very high
Latency	★★ Moderate	★ High (batch-oriented)	★★★★ Very low	★★★★★ Lowest
Programming	★★★★★ Easy (C/C++, Python, etc.)	★★★★ Moderate (CUDA, OpenCL)	★★ Moderate - Hard (HLS, Verilog/VHDL)	★ Very hard (hardware design)
Reconfigurable after manufacture	✓ Yes	✓ Yes	✓ Yes	✗ No
Development cost	★★★★★ Low	★★★★ Low - medium	★★ Medium	★ Very high
Time to market	★★★★★ Fastest	★★★★ Fast	★★ Medium	★ Slow
Typical use cases	OS, apps, control logic	ML, graphics, simulations	Networking, DSP, low-latency systems	Data-center accelerators, automotive, chips in phones, routers
Use cases in High Energy Physics	High-level trigger, offline analysis	High-level trigger, offline analysis, especially ML	Level-1 trigger systems, High-level trigger co-processors	Detector sensors, front-end readout

Device comparison



Aspect

CPU

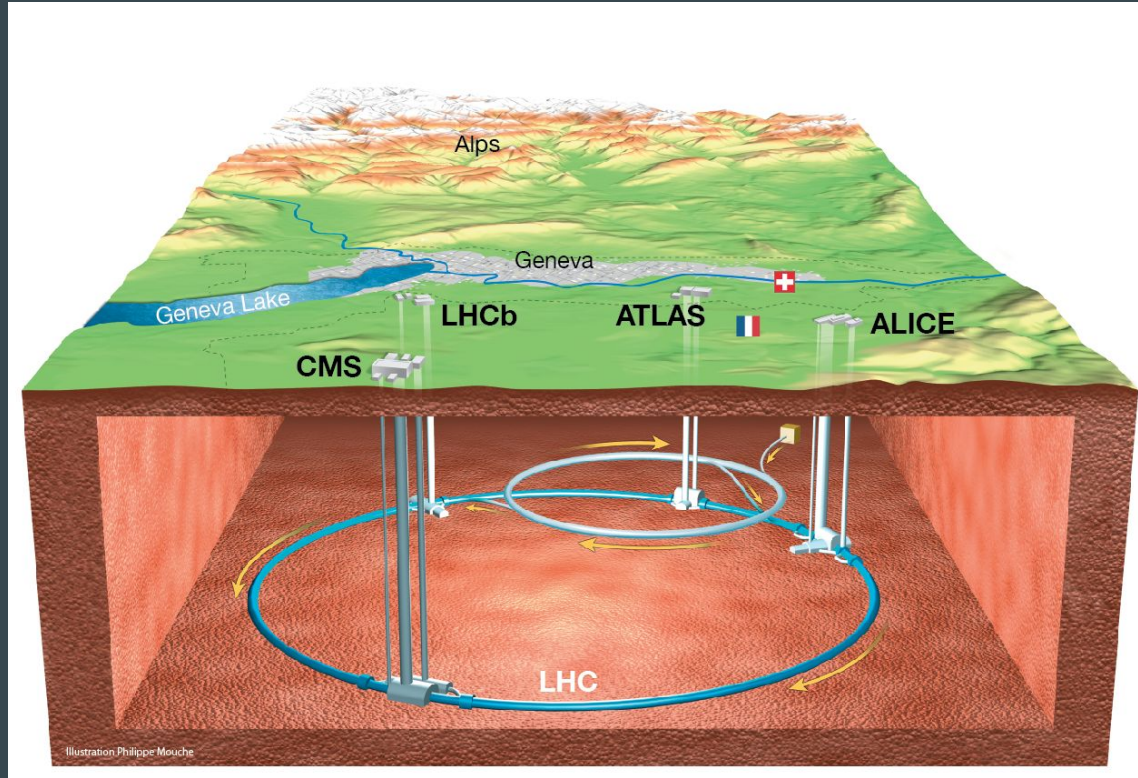
GPU

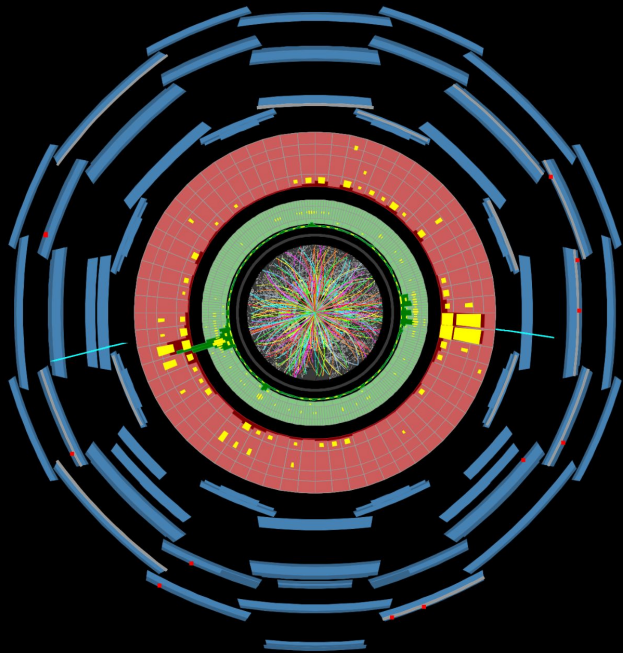
FPGA

ASIC

Full name	Central Processing Unit	Graphics Processing Unit	Field-Programmable Gate Array	Application-Specific Integrated Circuit
Primary purpose	General-purpose computing	Massively parallel computation	Custom hardware logic (reconfigurable)	Fixed, task-specific hardware
Flexibility	★★★★★ Very high	★★★★ High	★★ Medium (reprogrammable)	★ Very low
Performance (for specific tasks)	★ Moderate	★★★ High (parallel tasks)	★★★★ Very high	★★★★★ Highest
Parallelism	★ Low - moderate	★★★★★ Very high	★★★★ High (custom pipelines)	★★★★★ Custom (optimized)
Power efficiency	★ Moderate	★★ Moderate - high	★★★★ High	★★★★★ Very high
Latency	★★ Moderate	★ High (batch-oriented)	★★★★ Very low	★★★★★ Lowest
Programming	★★★★★ Easy (C/C++, Python, etc.)	★★★★ Moderate (CUDA, OpenCL)	★★ Moderate - Hard (HLS, Verilog/VHDL)	★ Very hard (hardware design)
Reconfigurable after manufacture	✓ Yes	✓ Yes	✓ Yes	✗ No
Development cost	★★★★★ Low	★★★★ Low - medium	★★ Medium	★ Very high
Time to market	★★★★★ Fastest	★★★★ Fast	★★ Medium	★ Slow
Typical use cases	OS, apps, control logic	ML, graphics, simulations	Networking, DSP, low-latency systems	Data-center accelerators, automotive, chips in phones, routers
Use cases in High Energy Physics	High-level trigger, offline analysis	High-level trigger, offline analysis, especially ML	Level-1 trigger systems, High-level trigger co-processors	Detector sensors, front-end readout

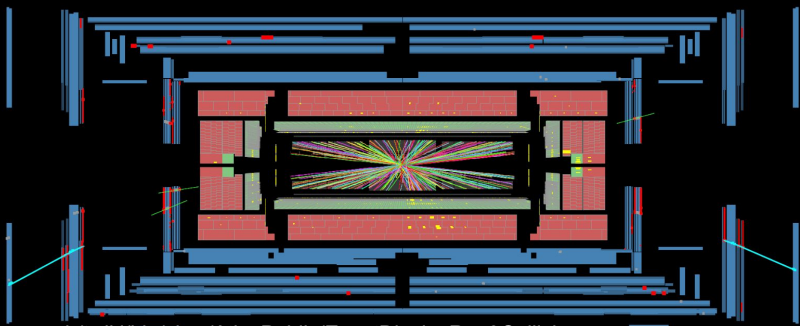
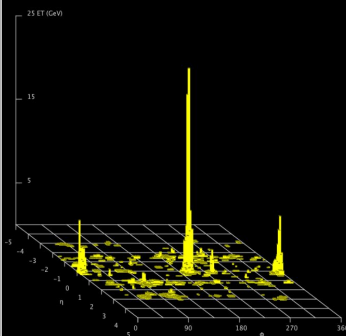
Applications of FPGAs in particle physics

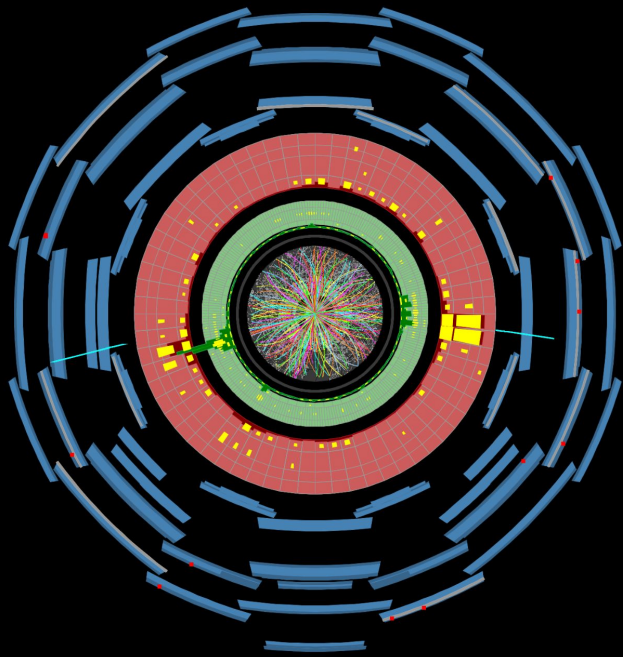




Run Number: 450227, Event Number: 6327489

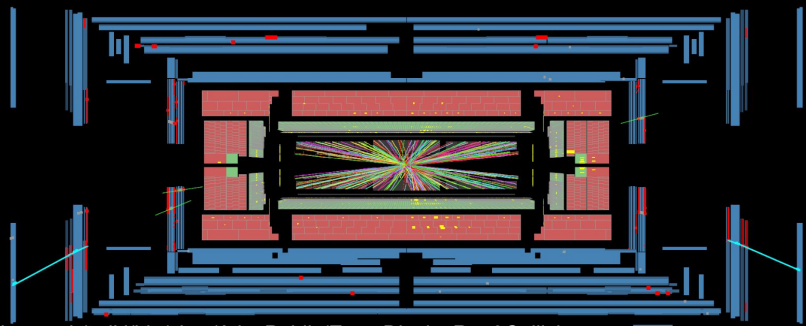
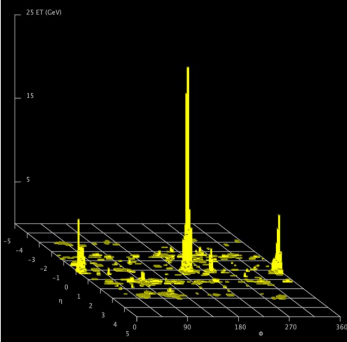
Date: 2023-04-21 19:24:16 CEST



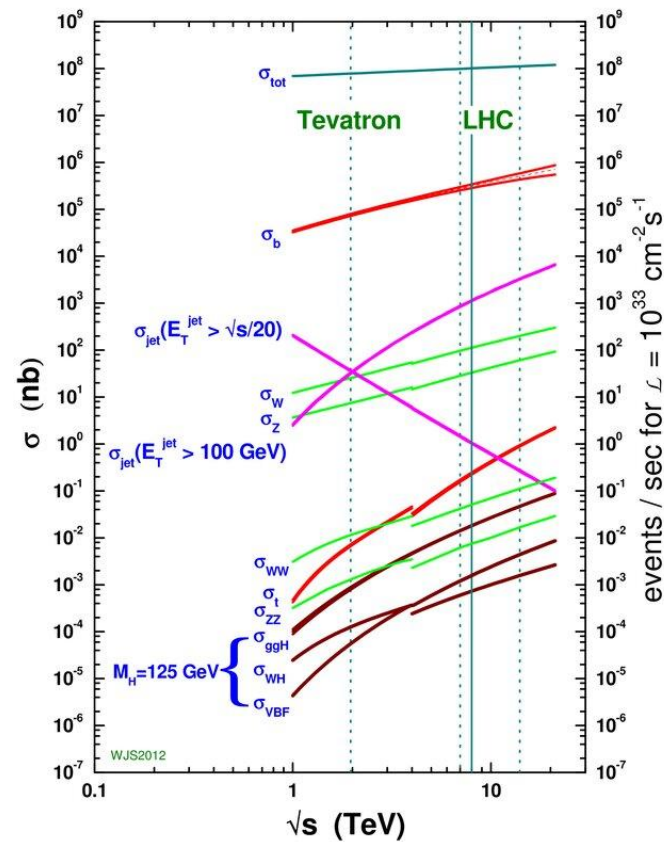


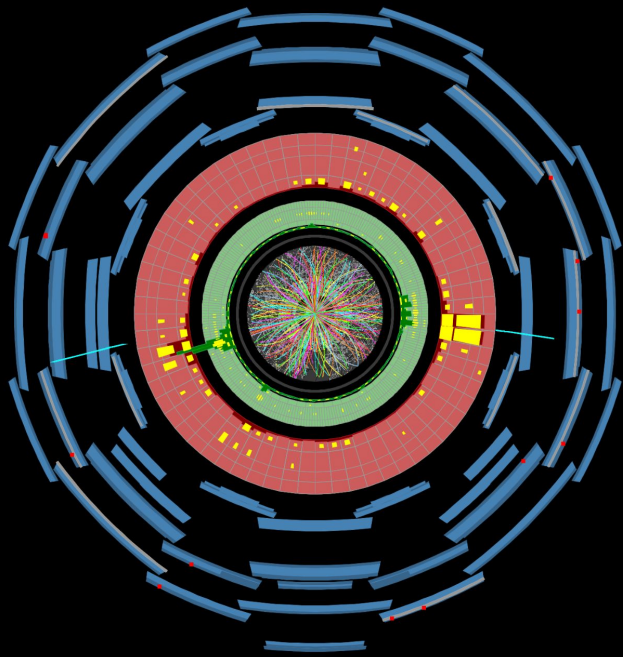
Run Number: 450227, Event Number: 6327489

Date: 2023-04-21 19:24:16 CEST



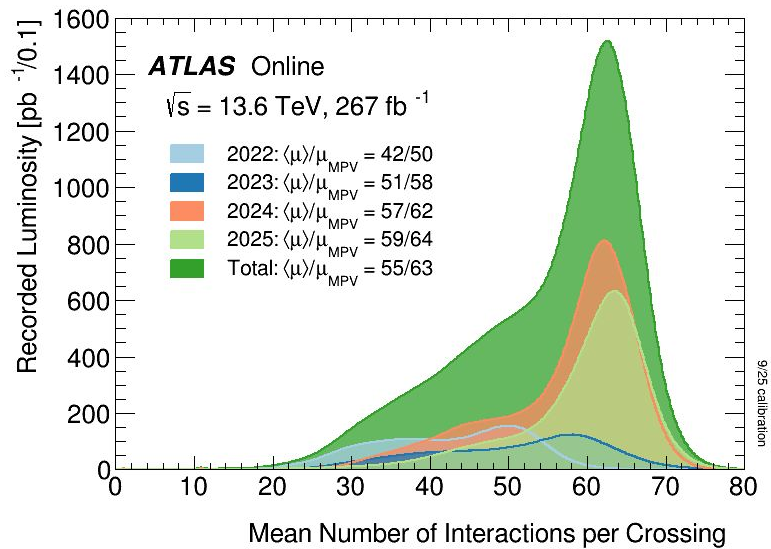
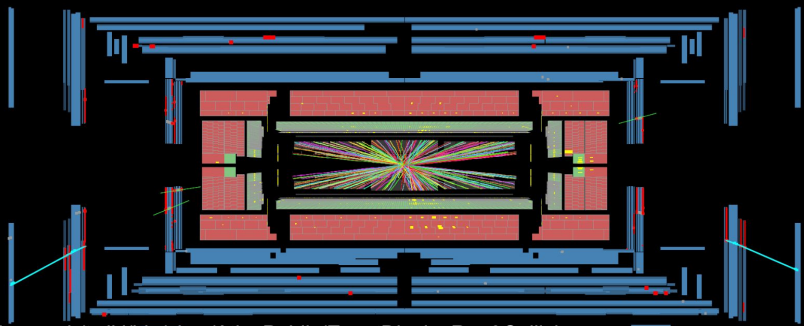
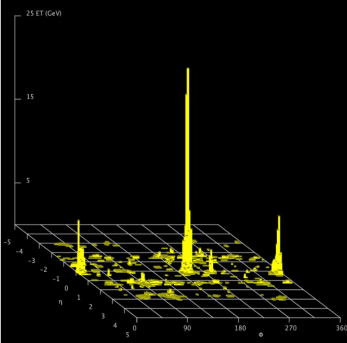
proton - (anti)proton cross sections





Run Number: 450227, Event Number: 6327489

Date: 2023-04-21 19:24:16 CEST



<https://twiki.cern.ch/twiki/bin/view/AtlasPublic/LuminosityPublicResultsRun3>

<https://twiki.cern.ch/twiki/bin/view/AtlasPublic/EventDisplayRun3Collisions>



27, Event Number: 6327489

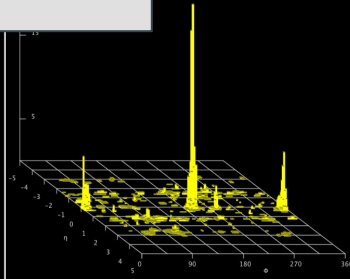
TDAQ

From collision
@ 100 Terabytes
per second

to storage
@ few Gigabytes
per second



Photo by Azamat Esenaliev from [Pexels](#)



Applications of FPGAs in particle physics

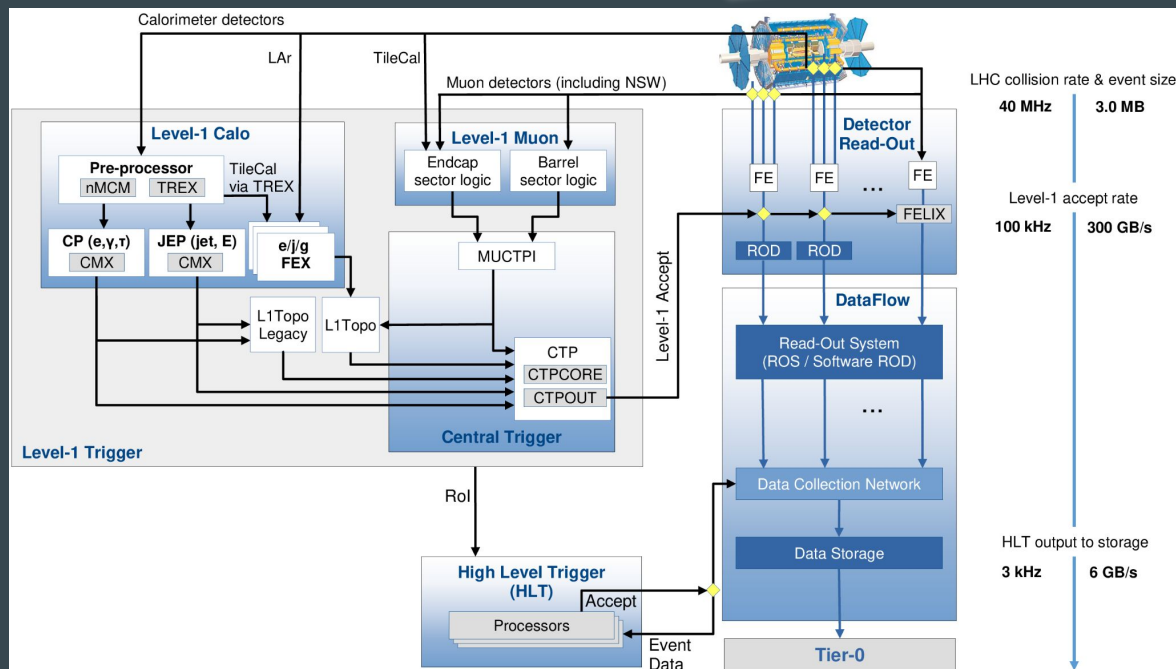
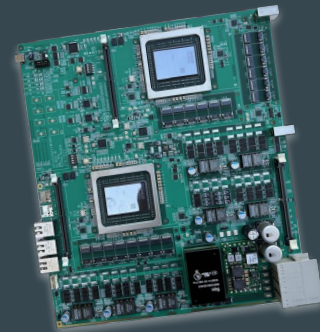
Real-time **trigger** systems

- Decide within microseconds which collision events to keep
- Handle data rates of TB/s

Fast **data acquisition**

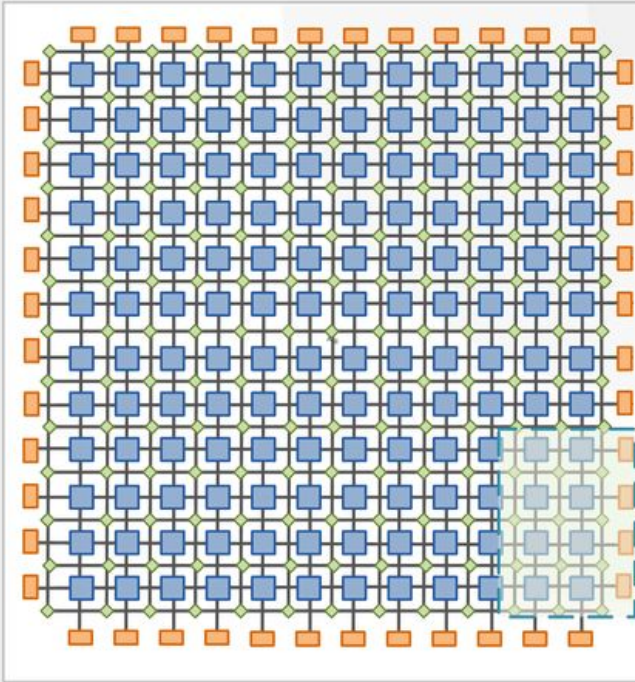
- Readout from detectors
- Deterministic latency
→ crucial for synchronization

FPGAs enable **real-time physics** where CPUs/GPUs are too slow or unpredictable.



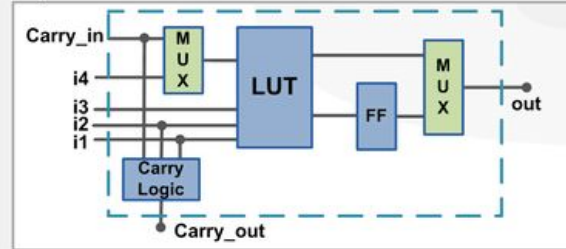
Conceptual FPGA Architecture

a) Island type FPGA architecture

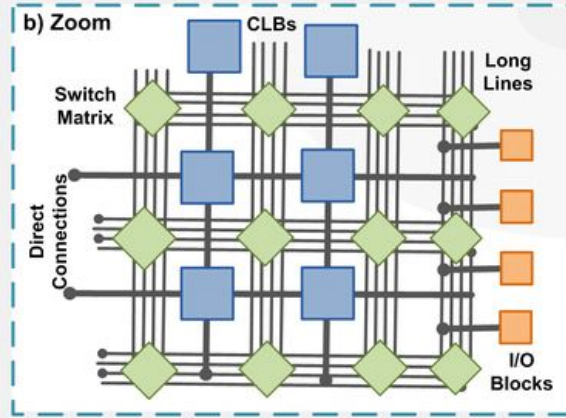


c) Basic architecture of a CLB

[source](#)



b) Zoom



Conceptual FPGA Architecture

a) Island type FPGA architecture

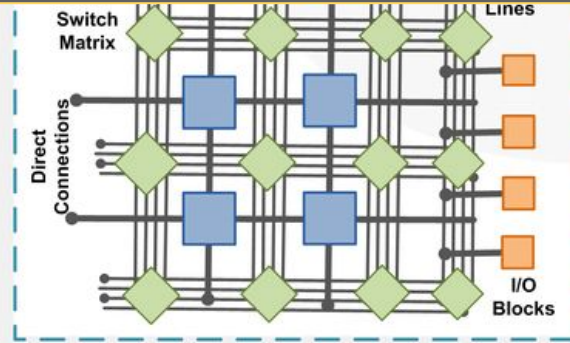
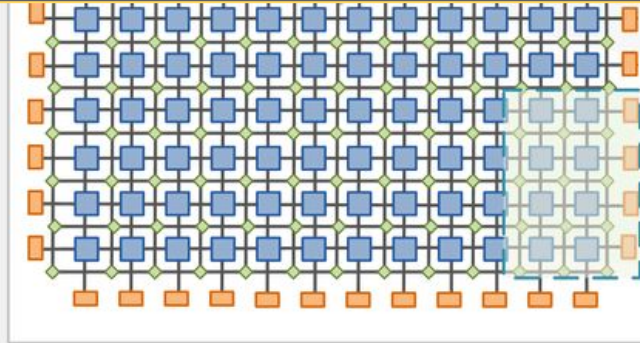


c) Basic architecture of a CLB



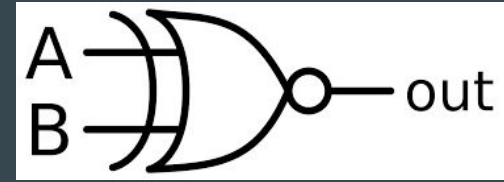
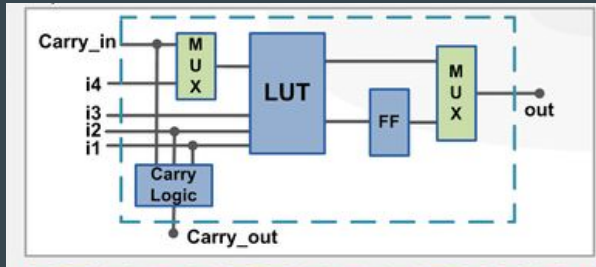
[source](#)

1. Everything instantiated runs in parallel – always!
2. Compute is spatially distributed on the FPGA
3. Data flows through deterministic, clocked pipelines



Configurable Logic Block

- Main resource for general purpose **combinatorial** and **sequential** circuits
- Logic, arithmetic & memory features
- Made of logic elements:
look-up tables & **registers**



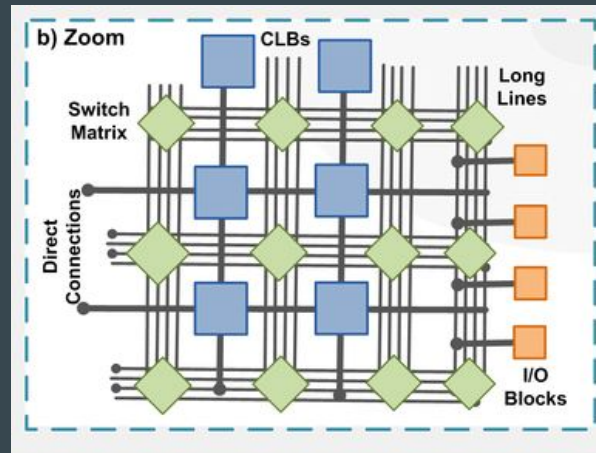
I1	I0	O0
0	0	1
0	1	0
1	0	0
1	1	1

from truth table to LUT

ADDR	D
0	1
1	0
2	0
3	1

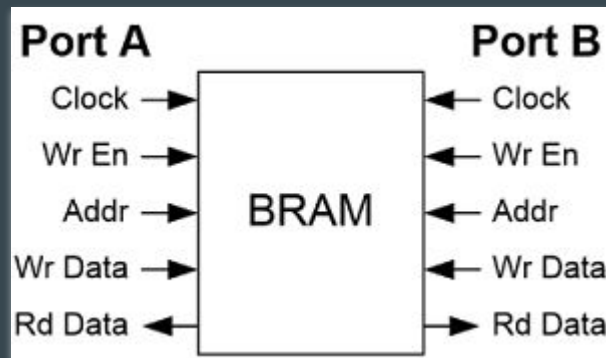
Interconnect

- **Programmable network** of signal pathways between inputs and outputs of functional elements
- **Switch matrix** to access general-routing resources
- External connectivity: **programmable I/O** cells
- **Volatile configuration** memory



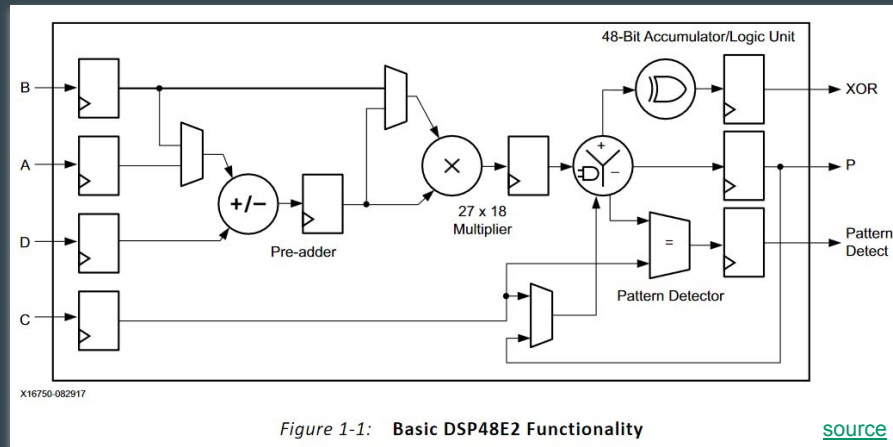
Block RAM *example: AMD UltraScale*

- Registers store state of a **single** bit
- Need for **larger data storage** → **BRAM**
 - Stores 36 Kbits of data with configurable port width
 - Two write and two read ports
 - Independent clocks
 - Cascadable → build larger blocks
 - Optional address sequencing → **FIFO** (this will become important for dataflow!)
- Need for **even larger data storage** → UltraRAM
 - Single-clocked, two port, synchronous memory blocks of 288 Kbits
 - Fixed port width
 - Cascade for larger blocks



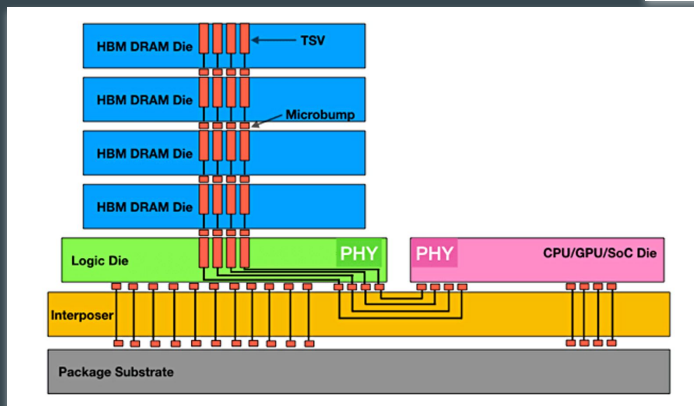
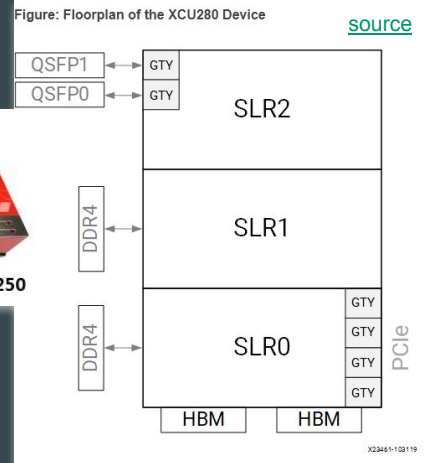
Digital Signal Processor (DSP) blocks *example: AMD UltraScale*

- Hardware optimized arithmetic, especially multiply-accumulate (MAC)
 - 27×18 two's complement multiplier
 - Cascadable 48b accumulators
 - SIMD add/sub/accu (2 x 24b, 4 x 12b)
- Small multipliers/ adders/ counters: use CLB carry logic
- 3 pipeline registers for full-speed multiply / MAC



External memories *example: AMD UltraScale*

- DDR and High-Bandwidth Memory (HBM)
 - Accelerator cards offer additional external memory
 - Alveo U250: 4 x 16 GB DDR4 @ 2400 MT/s
 - Alveo V80: 2 x 16 GB HBM2e @ max 820 GB/s
- We will use this to transfer data between host and accelerator card



Super Logic Regions (SLR) *example: AMD UltraScale*

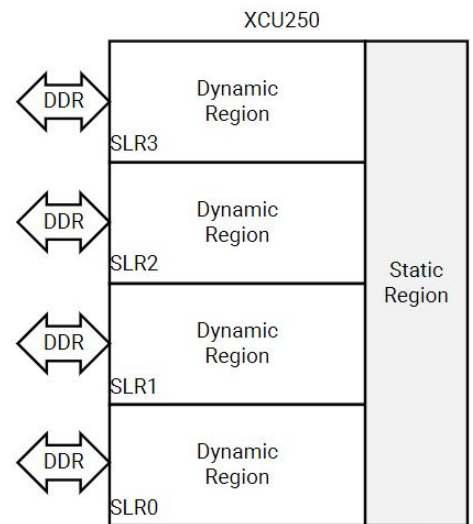
- 3D IC devices using stacked silicon interconnect
- Special routing lines between SLR:
super long lines (SLLs)
- Implications on **SLR utilization**, **SLR crossing**, **timing**

Table 7: U250 Dynamic Region Resources

Resource	Total	SLR0	SLR1	SLR2	SLR3
Look-Up Tables (LUTs) (K)	1341	338	332	332	339
Registers (K)	2749	693	680	680	696
36 Kb Block RAMs	2000	500	500	500	500
288 Kb UltraRAMs	1280	320	320	320	320
DSP Slices	11508	2877	2877	2877	2877

[source](#)

Figure 3: Floorplan of the XCU250 Device

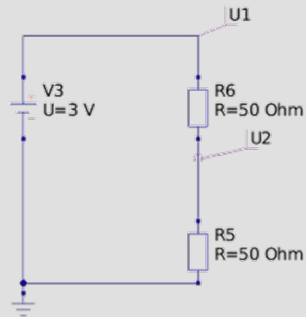


X21420-090718

Why do we care about timing?
Let's look at some basic electronics

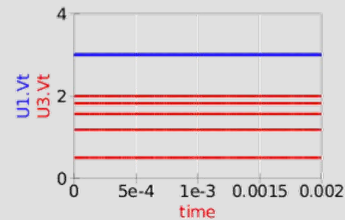
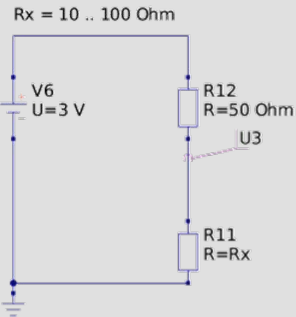
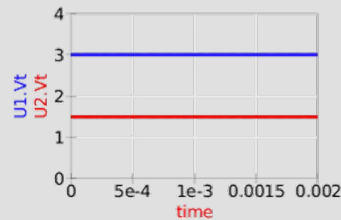
Passive Components

Resistive voltage divider

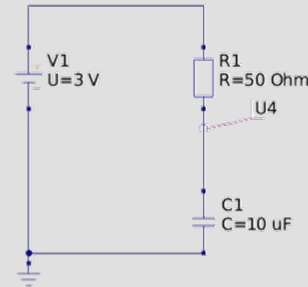


$$U_{R5} = R5 \cdot I$$

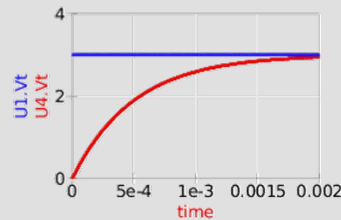
$$I = U_1 / (R5 + R6)$$



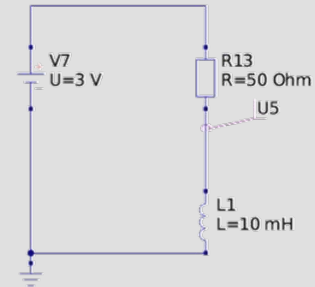
Resistor + Capacitor



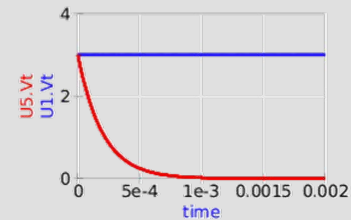
$$U_C = U_1 \cdot (1 - e^{-t/RC})$$



Resistor + Inductor

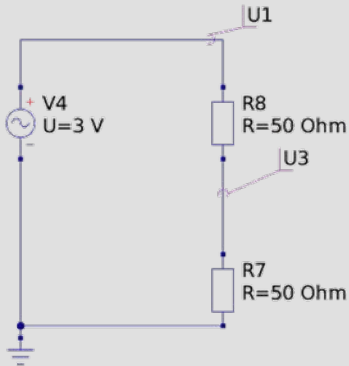


$$U_L = U_1 \cdot e^{-tR/L}$$

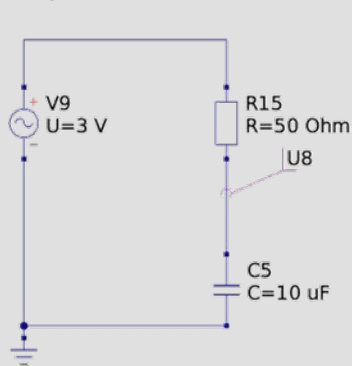


Frequency Response

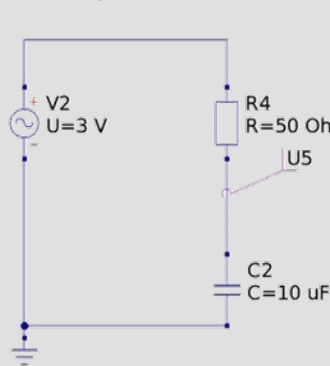
Sine-signal
Resistive divider



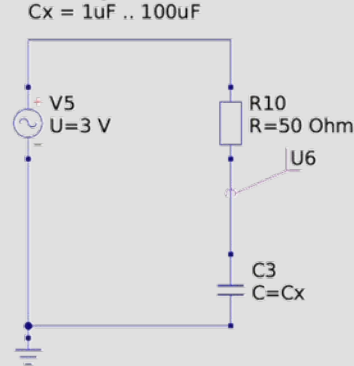
Sine-signal
Capacitive divider



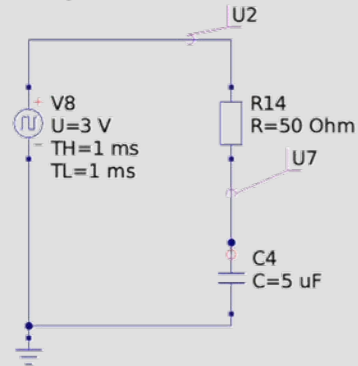
Sine-signal
RC low-pass



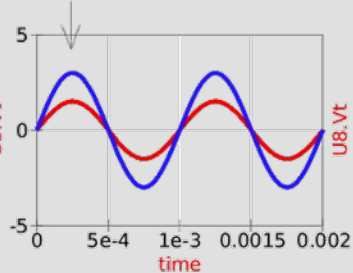
Sine-signal
RC low-pass
 $C_x = 1\mu\text{F} \dots 100\mu\text{F}$



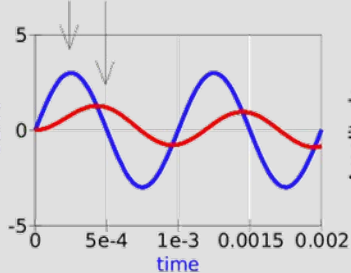
Square-signal
Capacitive divider



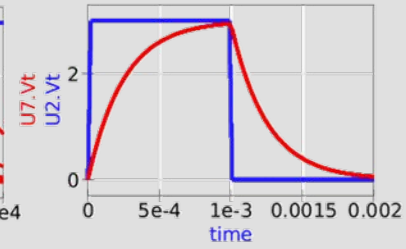
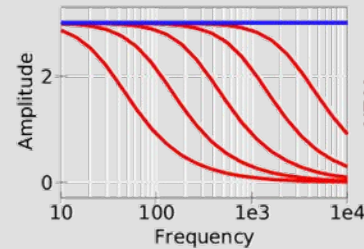
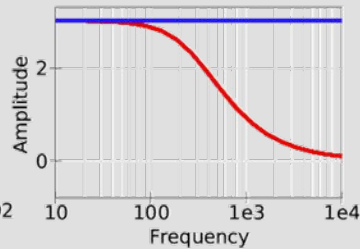
In phase



Phase shift

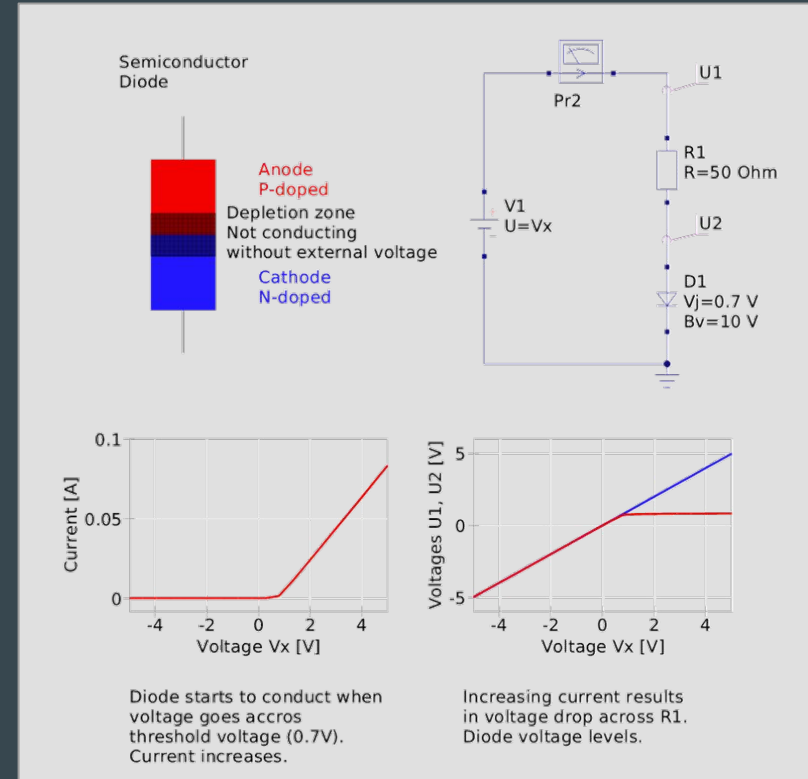


Amplitude at C compared to U1



Semiconductors

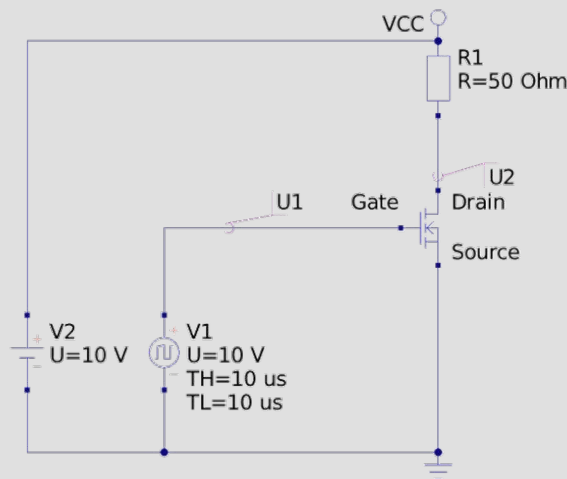
- „Doping“ (N,P) of normally non-conducting materials
=> semi-conductor
- Non-linear response.
Turn-on/off voltages (thresholds)



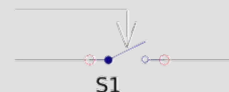
MOSFET

Transistor

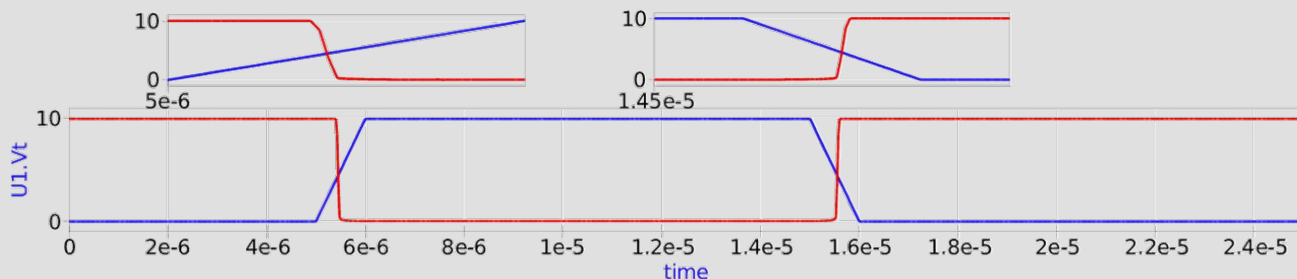
- 2 diodes, 3 terminals
- Acts as a switch
- Low power control input (Gate) steers high power main channel (Drain - Source)
- Asymmetric connection: Better to GND than to VCC



This MOSFET is equivalent to a switch. It conducts when the input voltage is high (above threshold). It breaks, when the input voltage is low (below threshold). Output voltage is opposite to input voltage (inverted). Resistance to VCC (via R1) and GND is different.

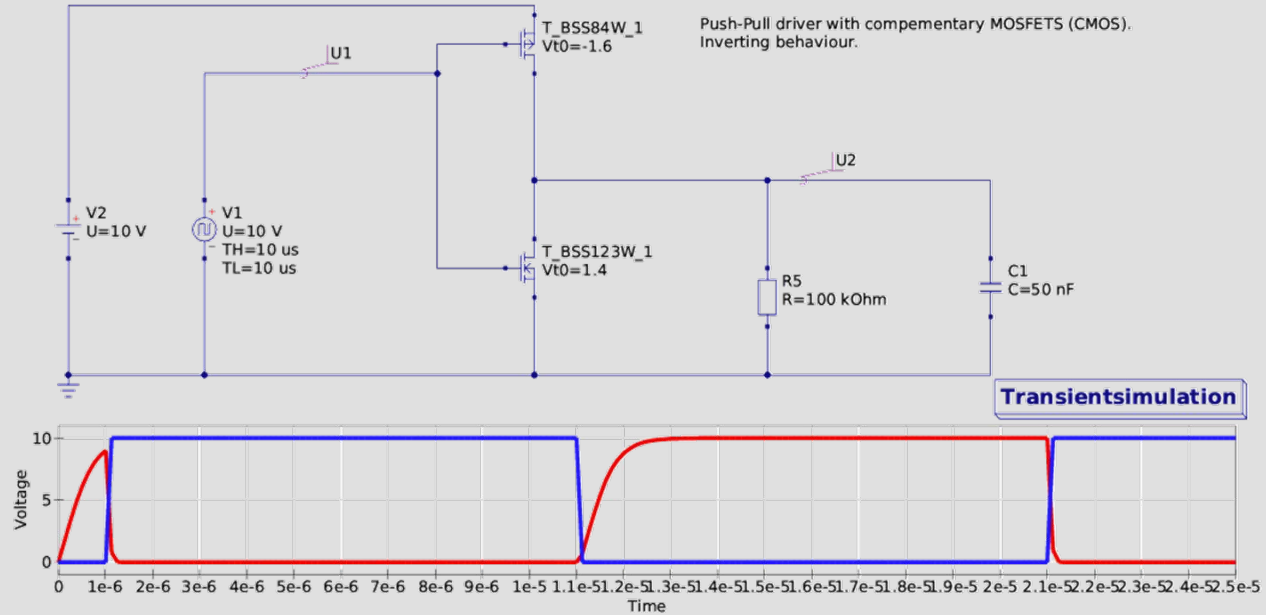


Gate has parasitic capacitance (important) and resistance (less important). Static gate current is 0. Dynamic gate current matters. The delay time from input to output depends on technology and circuit. Always > 0



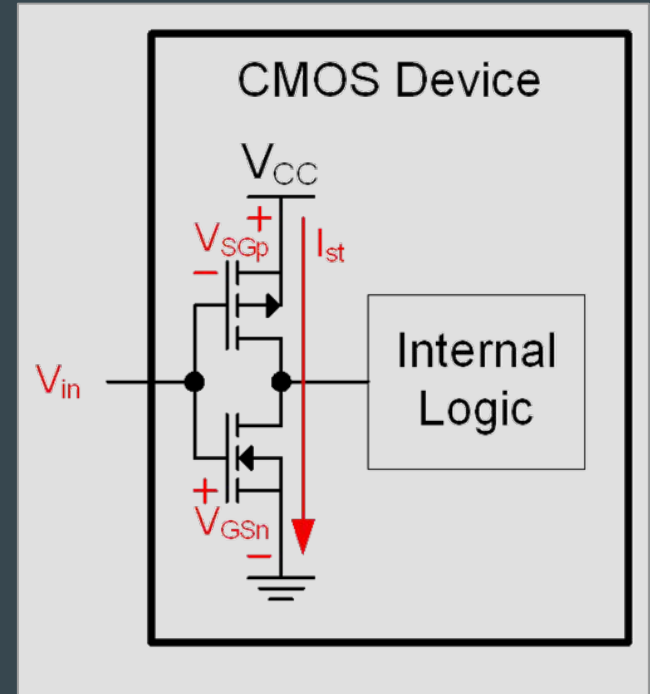
CMOS

Two complementary
(N,P) MOSFETs build
a push-pull driver



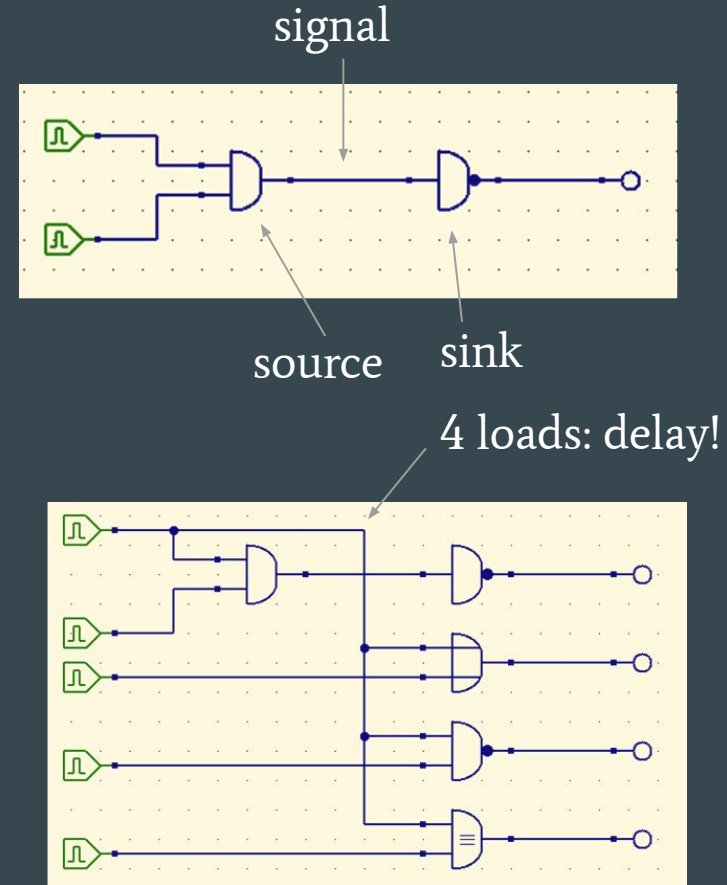
Input Properties of CMOS logic: MOSFET gate

- Ideal
 - Infinite high resistance
 - No parasitics
 - → zero input current, zero delay, zero power
- Real (roughly)
 - Very high non-infinite input resistance (static power)
 - Parasitic input capacity
 - Signal delay (gate R-C low-pass)
 - Frequency limits
 - Parallel inputs increase „C“
 - Charge/discharge → (dynamic power)
 - Leakage current



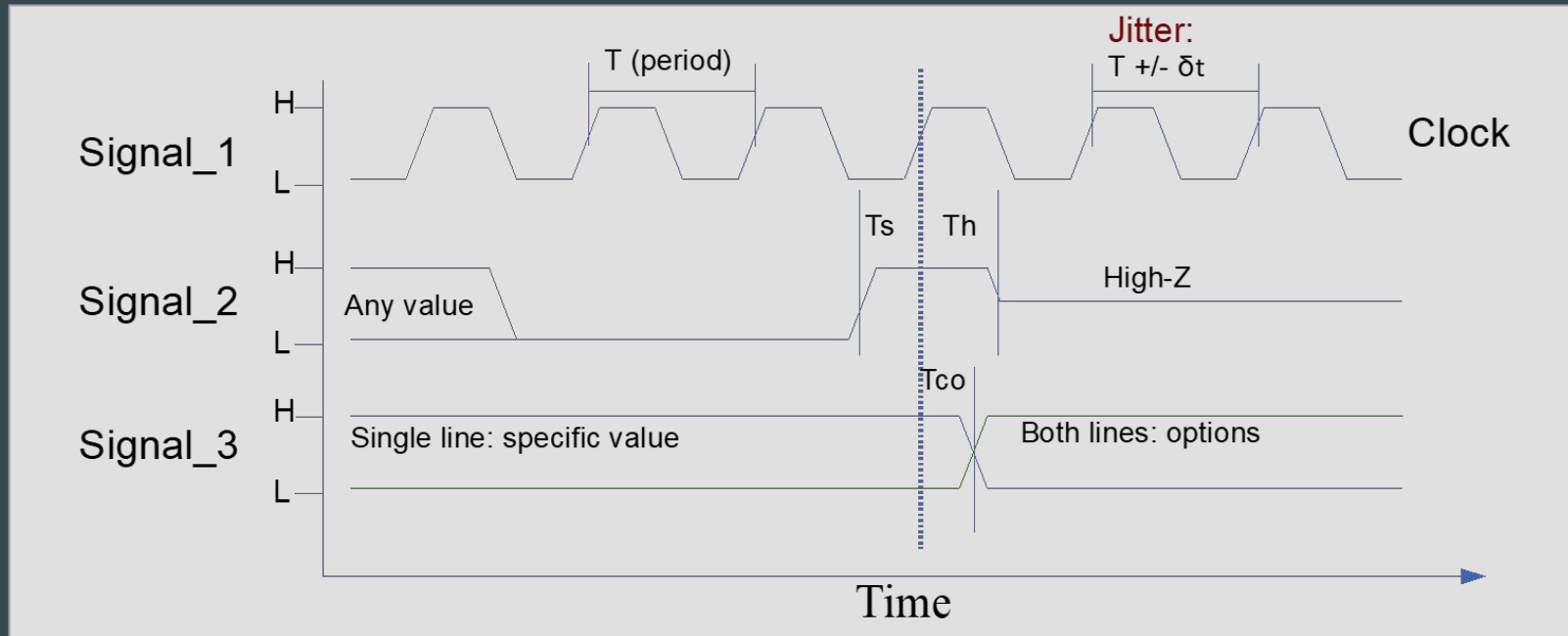
Signals

- Electronic circuits consists of nodes (e.g. gates) connected by metal wires – either off chip or on chip
- All nodes connected to the same electrical connection form one „signal“
- A signal must have at least one source (driver) and one sink (receiver)



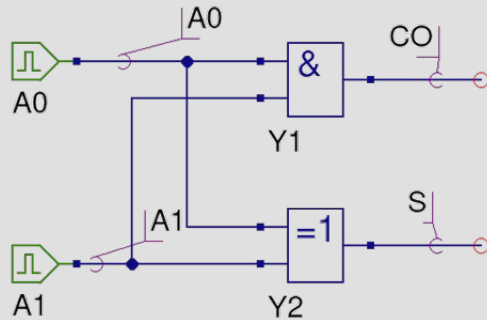
Timing Diagrams

Essential tool to display/analyze digital circuit functionality



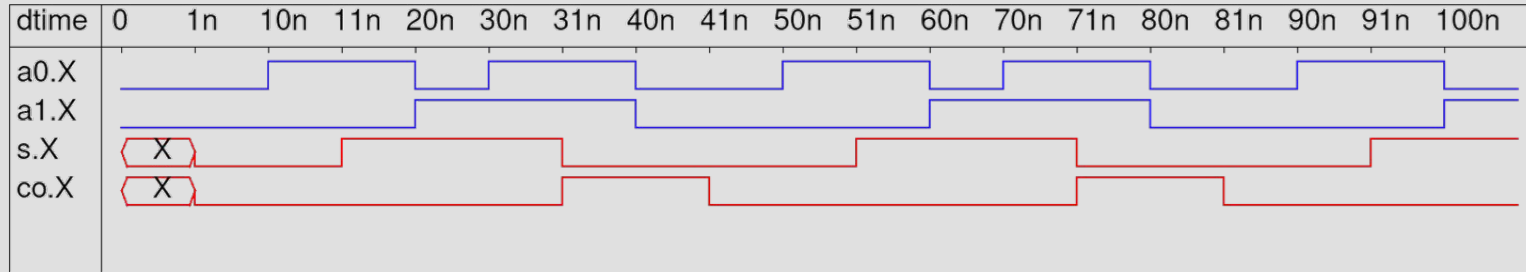
Combinatorial Logic Example: Adder

Half-adder with discrete gates



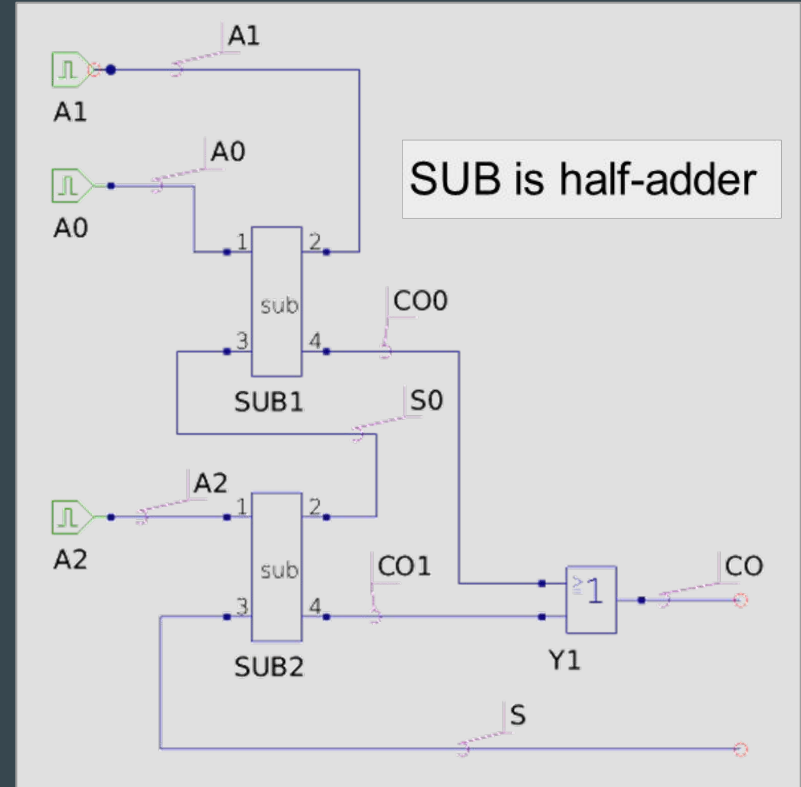
A1	A0	S	CO
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- Half-adder: addition of 2 bits
- Output sum (1 bit) and carry (1 bit)
- Counts up to 2

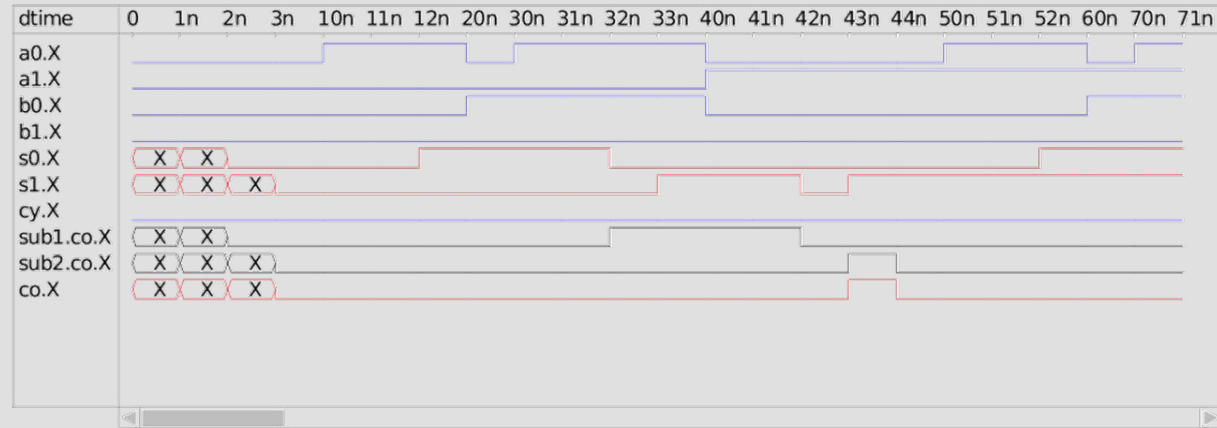
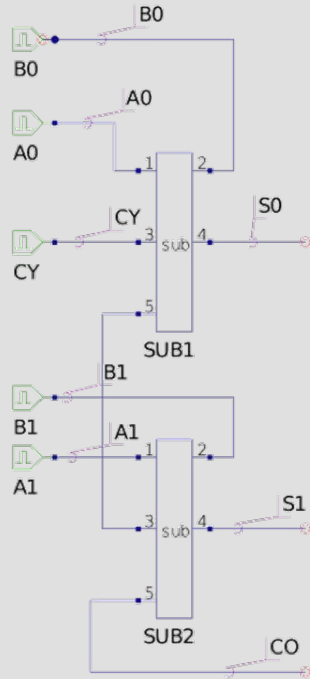


Full-Adder

- A full adder for 2 bits requires an additional carry input
- This is realized by cascading two half-adders, the second one for the carry input
- Counts up to 3



Multiple-Bit Adders



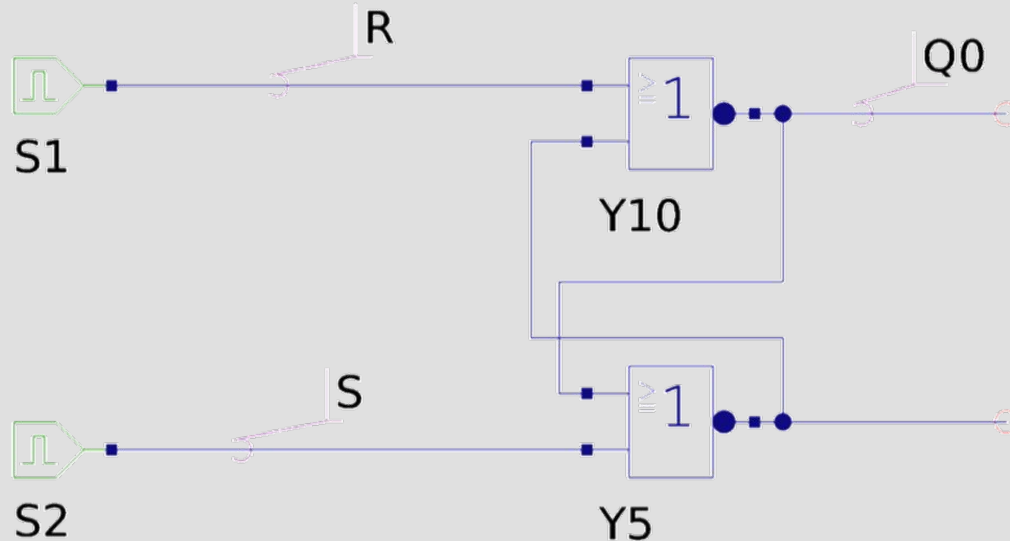
- Multiple bit adders are realized by individually adding the bits of the input words and cascading the carry outputs
- Every carry stage adds a delay

Critical Path

- The multi-stage „ripple-carry“ adder implements the logic with multiple levels
- The result of one level can be computed only once the result of the previous stages is available
- In such implementations there is always one **CRITICAL PATH**, which determines the **maximum delay** of the entire logic function
- In case of the **adder**, the critical path is the **chain of carry signals**
- After a change in the input, the carry output becomes valid last
- Every additional bit of the input width adds additional 2 gate delays
- The actual number of logic levels for a given function depends on device architecture

Storage Elements – RS-Latch

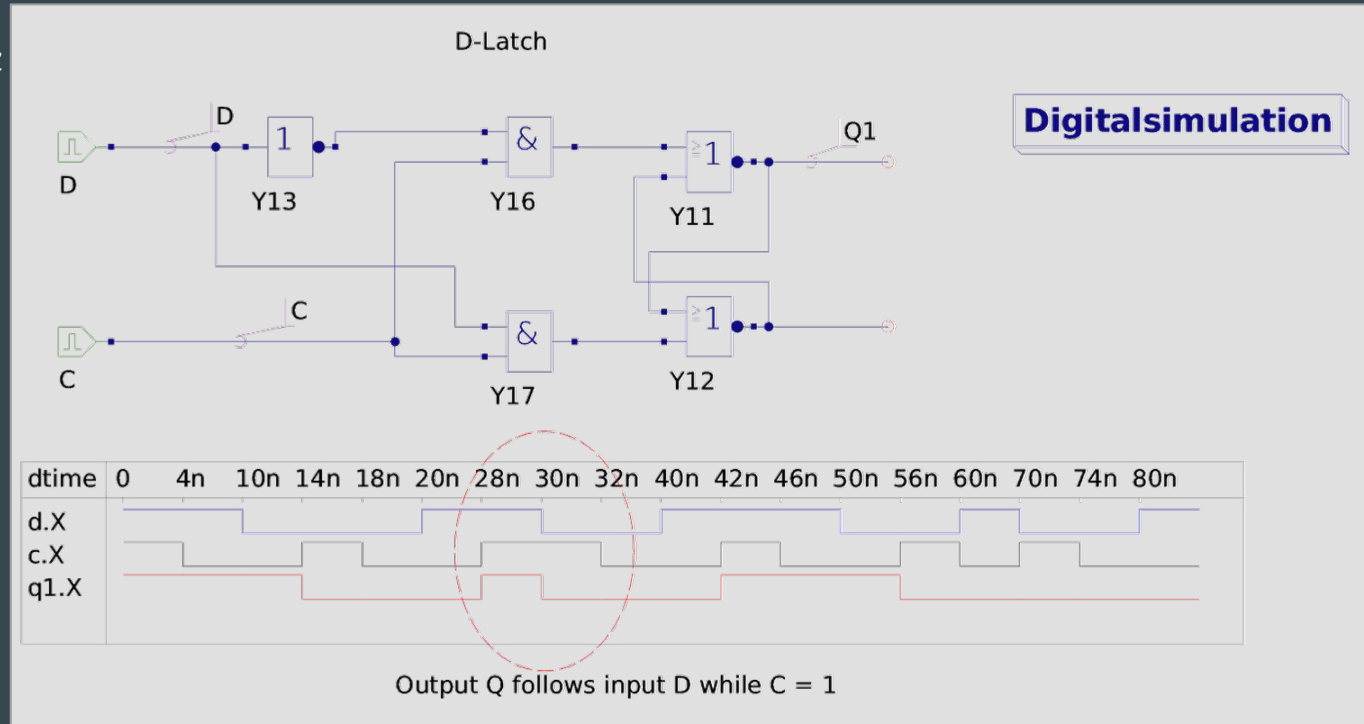
- Feedback creates storage elements
- RS-latches: very sensitive to spikes (short signal pulses due to noise or cross-talk)



R	S	Q0
0	0	Keep
1	0	0
0	1	1
1	1	Illegal

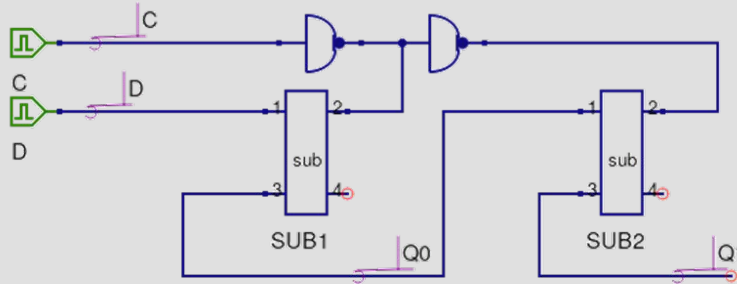
Storage Elements – D-Latch

- Additional logic to qualify input
- Input changes ignored while C low



Storage Elements – D-FlipFlop

- More precise timing: edge-sensitivity with D-FlipFlop (DFF)
- Cascaded D-latches with inverted control → foundation for SEQUENTIAL logic

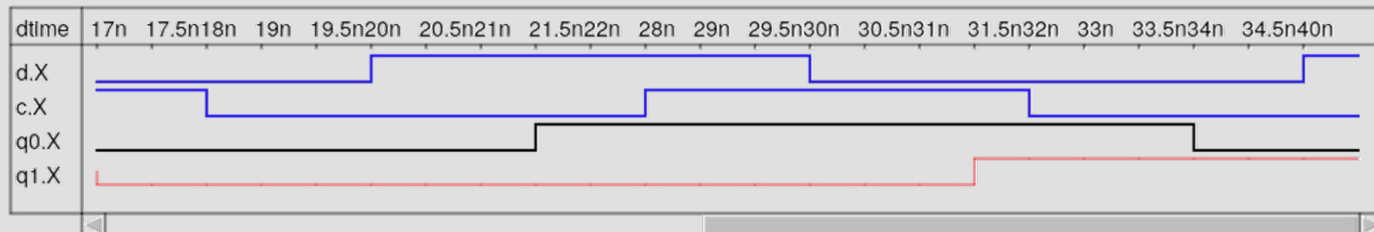


Two cascaded D-latches with inverted enable signal build a D-type flip-flop (DFF or just FF), an important digital component.

The output is updated with the rising edge of the enable signal, called "clock". The input must be stable some time before (setup time) and after (hold time) the rising edge.

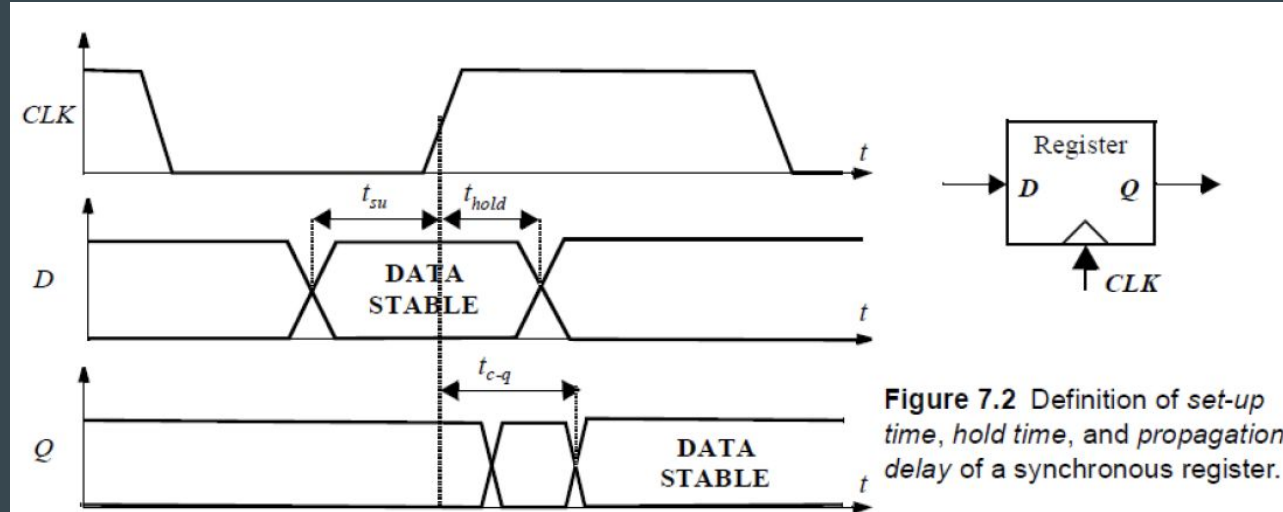
Digitalsimulation

Rising edge on C triggers update of Q1



D-FlipFlop Timing Characteristics

- The data signal needs to be stable before and after the clock rising edge
- The DFF may be in a meta-stable state if the data is unstable during the clock transition
- DFF output changes some time after the clock transition.
That's propagation delay.

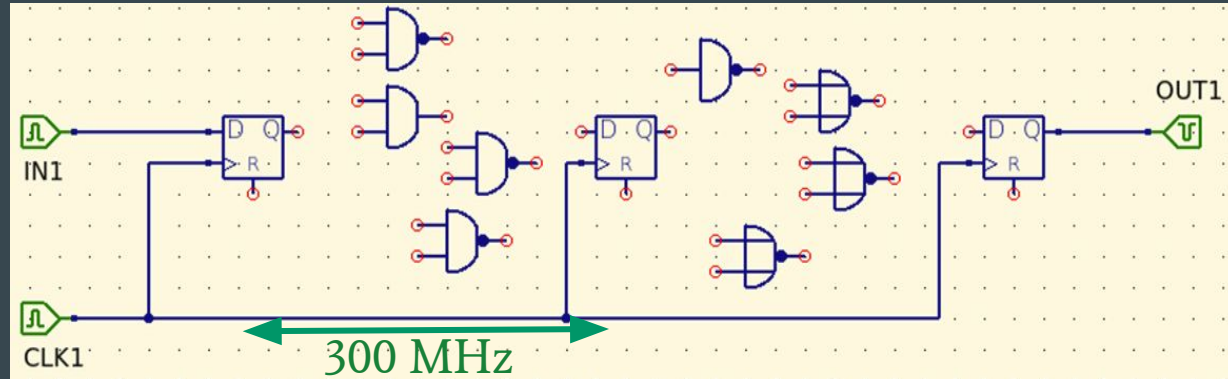
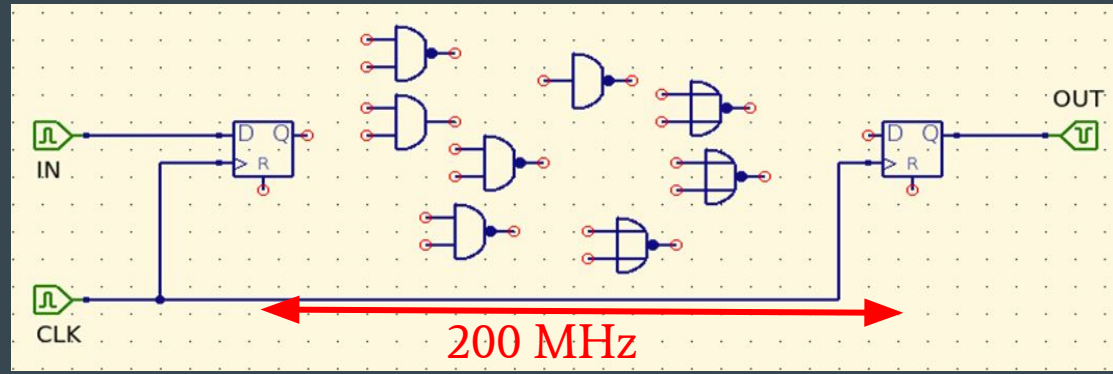


Sequential Logic

- Combinatorial logic delays make global timing analysis difficult/impossible
 - How do we know a computation is completed?
- Embedding combinatorial elements between sequential elements (FFs / registers) driven by a common (or well known) clock signal (synchronous) allows for precise timing
 - However, the clock period defines the speed of all operations
 - Cannot profit from faster combinatorial subcircuits
- Sequential logic is the standard approach

Pipelining

- Complex logic between two register stages => slow clock
- „Break“ logic into 2 (or more) stages
- Pipelined“ stages => faster clock
- Tradeoff: latency (2 cycles/1 cycle) vs frequency
- Not always possible



Takeaways for FPGA programming about timing

- You target some specific latency or throughput, which requires to run your hardware at a certain frequency
- What does it mean if my design does **not meet timing**?
 - It may only be possible to run at slower clock speed then specified and **fail** your requirements
 - If you try to run at your specified clock speed it may just **fail**!

Takeaways for FPGA programming about timing

- Timing closure = understanding the physics of signal propagation
- Signals with many sinks = high fan-out → Logic replication
 - Large capacitive load
 - Slower signal transition
- Long combinational paths → Pipelining
 - Accumulated propagation delay
 - Critical path determines maximum clock frequency
- Routing delay → Partitioning, localization, reduced parallelism
 - FPGAs are physically large → placement matters
 - Signals hop through switch matrix → adds resistance and capacitance
 - Routing delay can dominate logic delay

Programming Models

HDL versus HLS

Development flows: RTL design flow using HDL

[source](#)

```
type demuxStateType is (stall, run);
signal loadState, loadNextState : fifoStateType;
signal readState, readNextState : fifoStateType;
signal demuxState : demuxStateType;

signal ingressPifofrom : std_logic;
signal validForEgressFifo : std_logic_vector (13 downto 0); -- a shiftregister
signal loadIngressFifo, ingressFifoFull, ingressFifoEmpty : std_logic_vector
signal egressFifoFull, egressFifoEmpty, readEgressFifo, fifoSelect : std_logic;
signal readIngressFifo, loadEgressFifo : std_logic;
signal loadEgressFifo : std_logic_vector (7 downto 0);

signal wbDataForInputReg : std_logic;

-- data needs two levels of pipelining
signal wbInputDataStage0, wbInputDataStage1 : std_logic;

process (wbClk)
begin
    if rising_edge(wbClk) then
        readIngressFifo <= '0';
        loadState <= loadNextState;
        wbDataForInputReg <= wbDataForInput;
    end if;
end process;
```

HDL Code

```
create_clock -period 5.0 -name wbClk [get_ports wbClk]
create_clock -period 2.5 -name bftClk [get_ports bftClk]

set_property IOSTANDARD LVCMOS18 [all_inputs]
set_property IOSTANDARD LVCMOS18 [all_outputs]

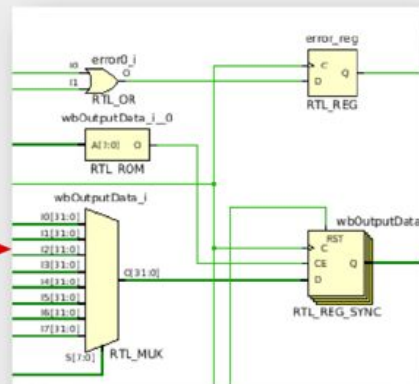
set_property PACKAGE_PIN A10 [get_ports {v0Output}]
set_property PACKAGE_PIN A14 [get_ports {error}]
set_property PACKAGE_PIN A19 [get_ports {reset}]
set_property PACKAGE_PIN A20 [get_ports {wbClk}]
set_property PACKAGE_PIN A24 [get_ports {wbDataForInput}]
set_property PACKAGE_PIN V11 [get_ports {wbInputData[0]}]
```

XDC

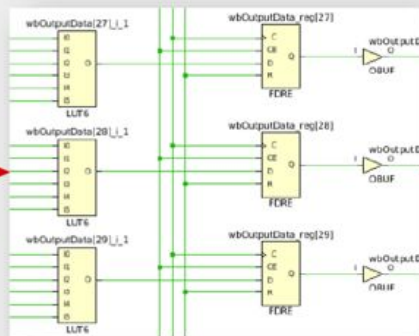


Elaborate

Synthesize

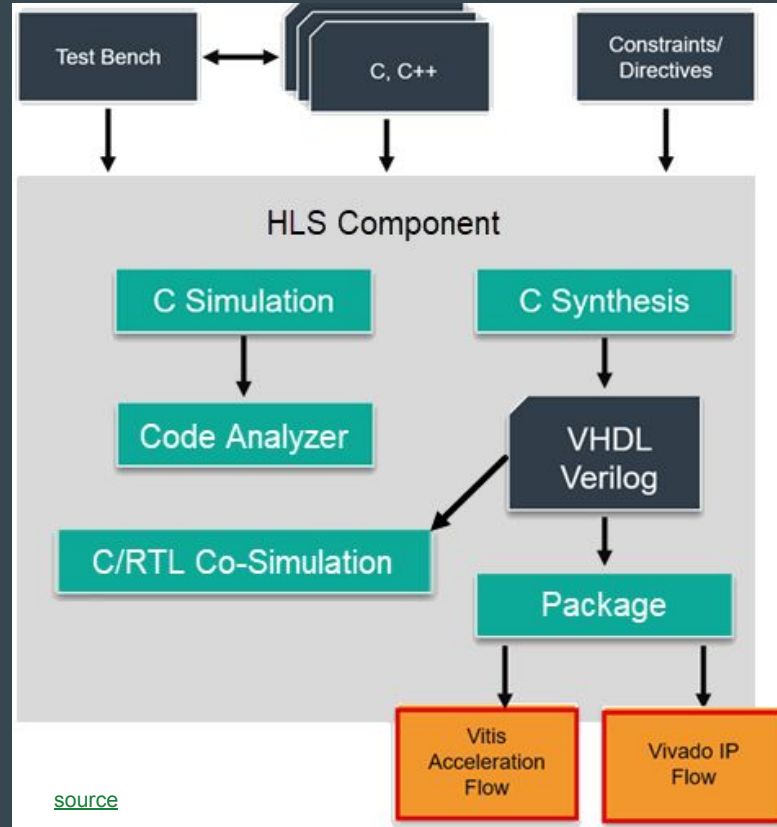


Analyze

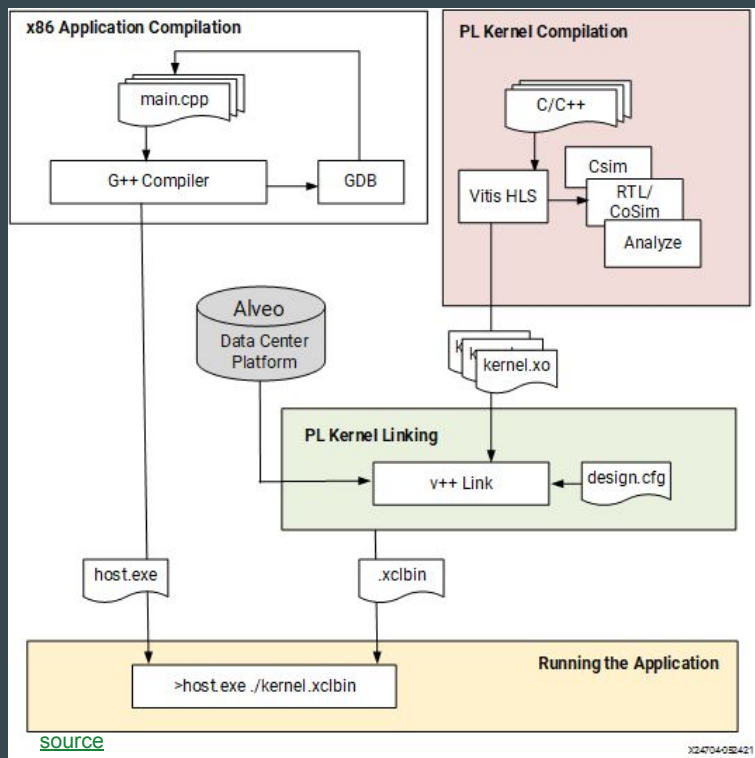


Implementation

Development flows: HLS design flow



Development flows: Vitis Acceleration flow



Comparison

Hardware Description Language (HDL)

- Cycle-accurate hardware description
- Explicit control over timing and parallelism
- Manual RTL design (FSMs, pipelines, datapaths)
- Fine-grained resource control (LUTs, FFs, BRAM, DSPs)
- Deterministic performance and latency
- Longer development time
- Steeper learning curve (hardware-centric thinking)
- Easier to hit aggressive timing constraints
- Well-suited for custom interfaces and low-level IP

High Level Synthesis (HLS)

- Compile C/C++ → hardware
- Algorithm-level abstraction
- Automatic scheduling, pipelining, and resource allocation
- Faster development and prototyping
- Software-friendly workflow
- Easier design-space exploration (pragma tuning)
- Less explicit control over exact cycle behavior
- May require iteration to match hand-written RTL performance
- Ideal for compute-heavy kernels (DSP, ML, video, etc.)

What **HLS** does **NOT** do for you

HLS removes HDL syntax, but it does not remove hardware thinking!

- It does **NOT** automatically parallelize your algorithm
- It does **NOT** automatically pipeline everything
- It does **NOT** automatically fix memory bottlenecks
- It does **NOT** infer your performance intent
- It does **NOT** behave like a CPU
- It does **NOT** support full C++

Differences between software C++ and HLS C++

Software C++ describes execution ⇔ HLS C++ describes hardware structure

- No dynamic memory allocation in HLS!

```
int *a = new int[n];
```

Hardware must be fully defined at synthesis time

- No recursion! Hardware must be finite, static, and fully allocated at compile time

```
int factorial(int n) {  
    if(n == 0) return 1;  
    return n *  
    factorial(n-1);  
}
```

Instead: iterative loops, FSMs, pipelines

Differences between software C++ and HLS C++

Software C++ describes execution $\Leftrightarrow$ HLS C++ describes hardware structure

- Loops mean hardware: either reusing or replicating hardware $\rightarrow$ pragmas!

```
for(int i=0;i<4;i++)  
    y[i] = a[i] + b[i];
```

- Control flow becomes hardware control logic
Branching in hardware $\rightarrow$ compute both values
+ Multiplexer to select result

```
if(a > b)  
    x = a;  
else  
    x = b;
```

Differences between software C++ and HLS C++

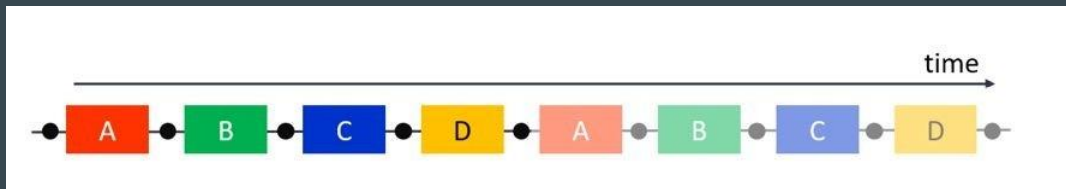
Software C++ describes execution ⇔ HLS C++ describes hardware structure

- Variables are registers or wires
 - Local variables → register
 - Arrays → BRAM
 - **Static** variable → persistent register
- Time is explicit in HLS
 - Each operation takes clock cycles
 - Latency is fixed
 - Scheduling is static
- No operating system
 - No dynamic runtime or system environment

```
int tmp;  
int array[1024];  
static int counter = 0;  
counter++;
```

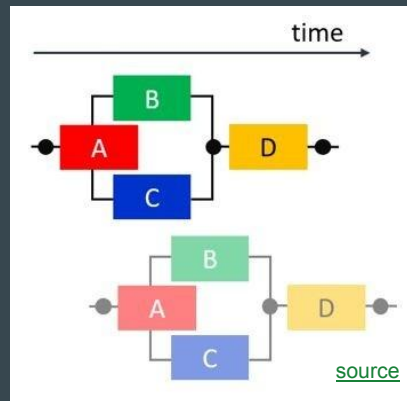
FPGA Processing model

Instead of sequentially executing tasks, as on CPU (or GPU with massive data parallelism)



An FPGA shines most if we can achieve

- **task parallelism and pipelining within a run**
- **pipelining of runs**
- and **pipelining within a task**

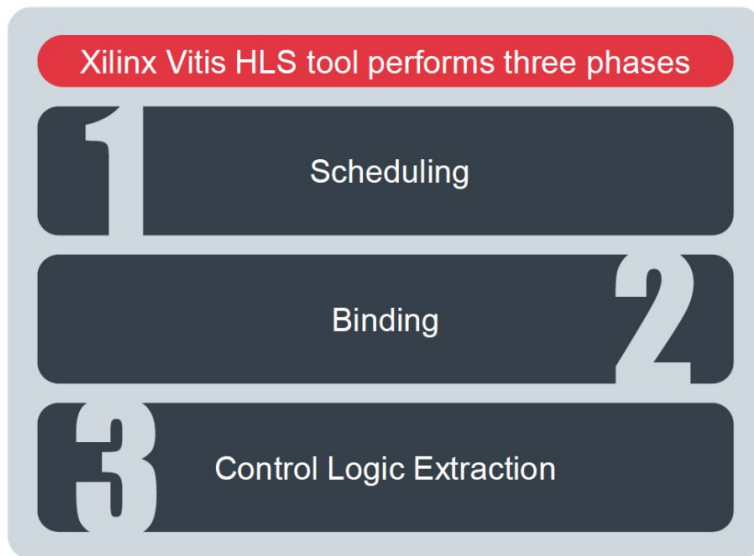


Basics of High-Level Synthesis

Basics of High-Level Synthesis

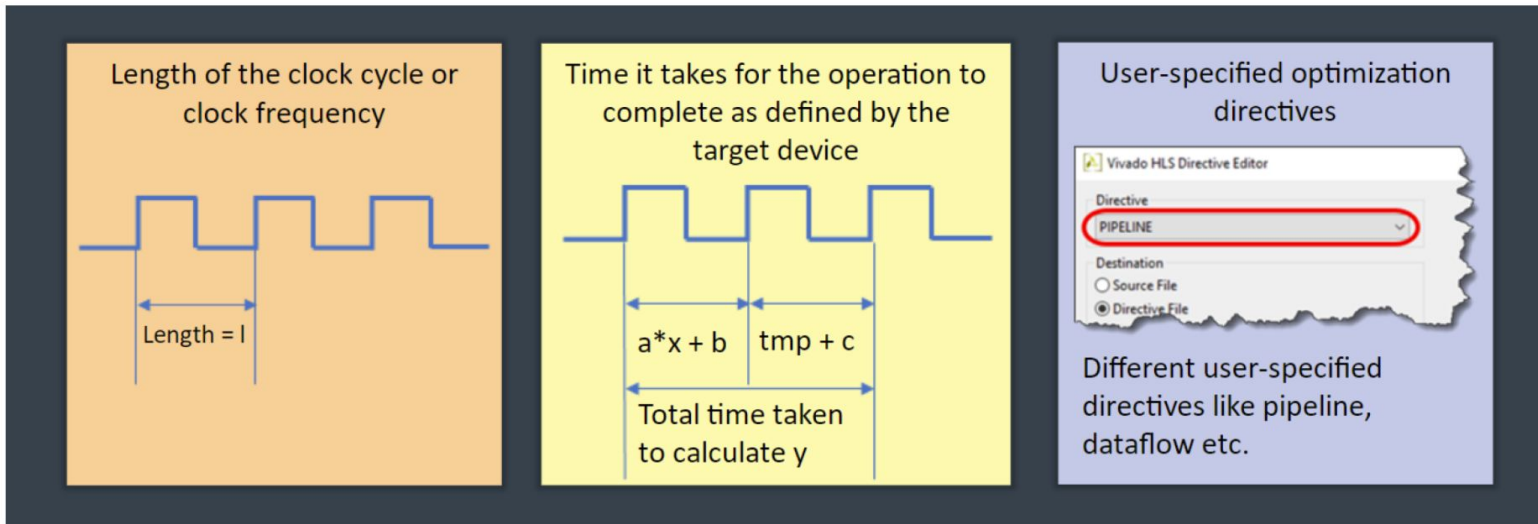
Vitis HLS tool

- Allows C, C++, and OpenCL™ functions to become hard wired onto the device logic fabric and RAM/DSP blocks
- Implements hardware kernels in the Vitis application acceleration development flow and develops RTL IP for FPGA designs in Vivado® Design Suite



Scheduling

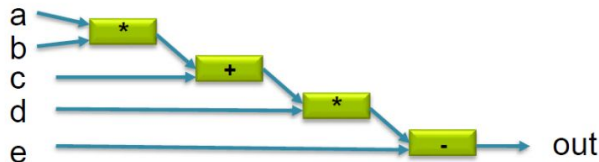
- ▶ Scheduling determines which operations occur during each clock cycle based on :



Scheduling

- ▶ If the clock period is shorter or a slower FPGA is targeted
 - HLS automatically schedules the operations over more clock cycles as well as some operations might need to be implemented as multicycle resources

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



Schedule 1

- ▶ If the clock period is longer or a faster FPGA is targeted
 - More operations are completed within a single clock cycle



Schedule 2

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



Binding

- ▶ Binding phase determines which hardware resources implements each scheduled operation
 - Operators map to cores

- ▶ Binding Decision: to share

- Given this schedule:



- Binding must use 2 multipliers, since both are in the same cycle
 - It can decide to use an adder and subtractor or share one addsub

- ▶ Binding Decision: or not to share

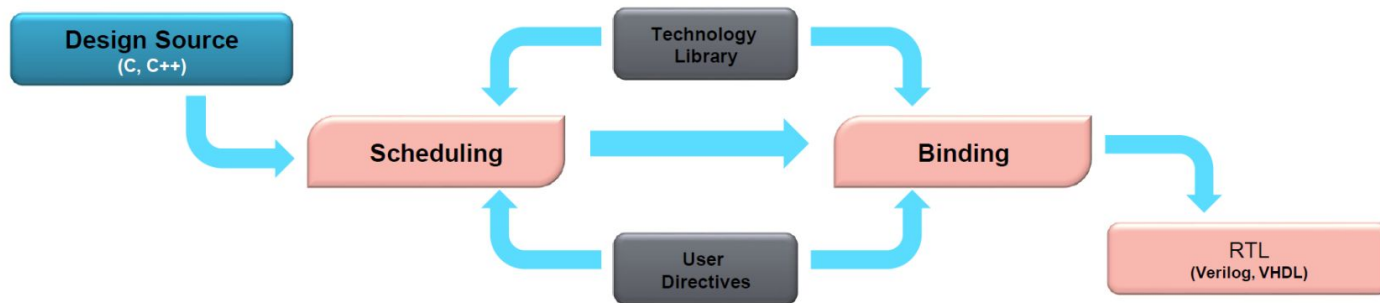
- Given this schedule:



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

Scheduling and Binding

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



Scheduling and Binding Example

► Scheduling Phase:

- Scheduling of the operations during each clock cycle
- Multiplication and first addition performed in the first clock cycle
- An internal register stores this result
- Second addition and the output generation happen in the second cycle

► Initial binding phase:

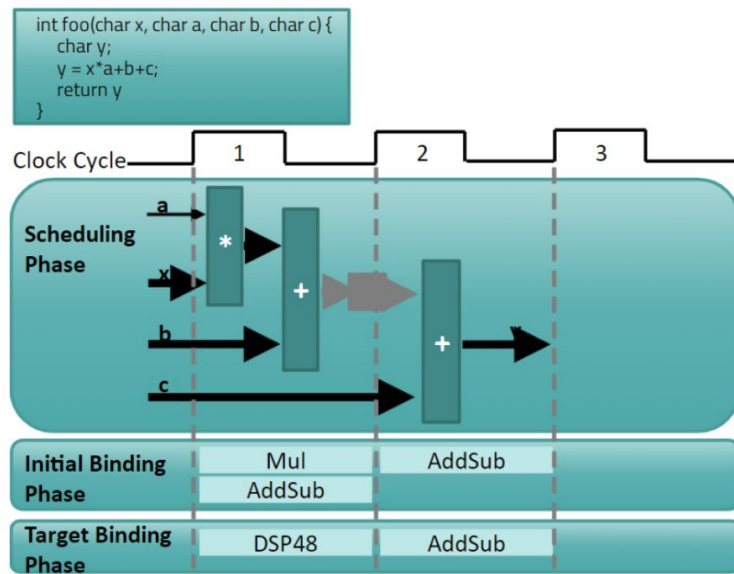
- Implements the multiplier operation using a combinational multiplier (Mul)
- Implements both add operations using a combinational adder/subtractor (AddSub)

► Target binding phase:

- Implements both the multiplier and one of the addition operations using a DSP48 resource

► Final hardware implementation:

- Implements the arguments to the top-level functions as input and output ports



Control Logic Extraction



Extracts the control logic to create a finite state machine that sequences the operations in the RTL design

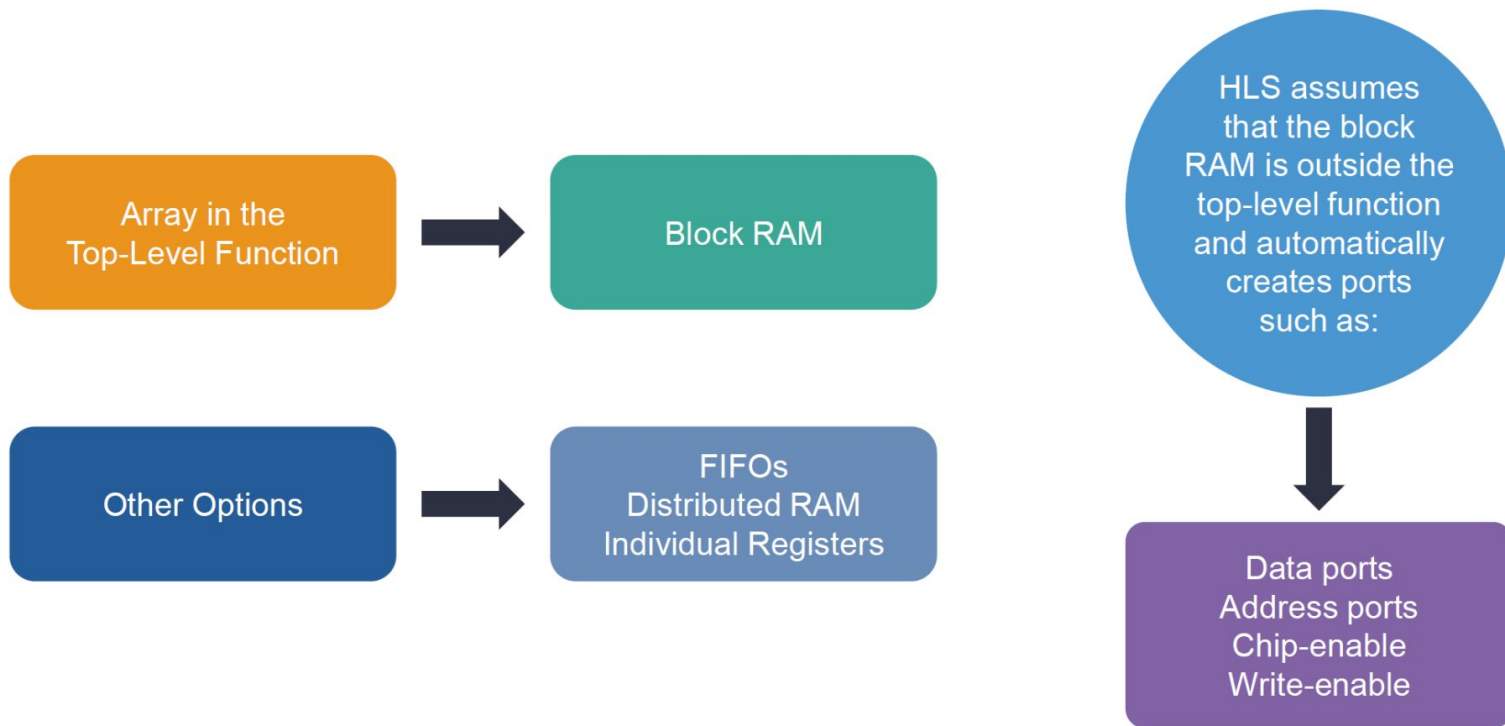


Why was the control logic extractions not happening in the HLS tool in the scheduling and binding example?

There were no loops!

Differences between combinational circuit and sequential circuit to be implemented into FPGAs

Control Logic Extraction & I/O Port Implementation Example

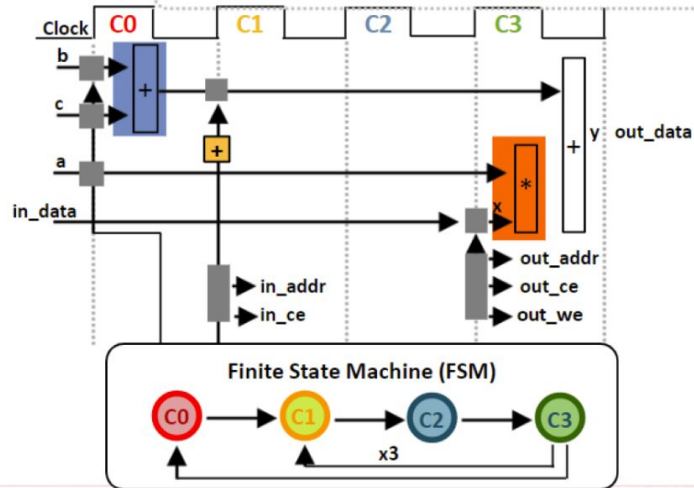


Control Logic Extraction

- ▶ Example performs the same multiplication and addition operations but inside a 'for' loop
 - HLS automatically extract the control logic from the C code and creates a Finite State Machine(FSM) in the RTL design to sequence these operations
 - FSM controls when the registers stores data and the state of any I/O control signals.
 - It starts in the state C0. On the next clock, it enters state C1, then state C2, and then C3.
 - Each time the design enters state C3, it reuses the result of the addition
 - Process continues until all output is written
 - Design then returns to the state C0 to read the next values of b and c to start the process again

```
void foo (int in [3], char a, char b, char c, int out [3]) {
    int x,y;
    for (int i = 0; i < 3; i++) {
        x = in[i];
        y = a*x + b + c;
        out[i] = y;
    }
}
```

- C0** - adder b + c
- C1** - generates the address for *in* and *adder* to increment to count how many times design iterates {C1, C2 and C3}
- C2** - Block Ram returns the data for *in* and stores it as variable *x*
- C3** - Reads the data from port *a* with other values to perform the calculation and generates the first *y* out



Key Attributes of C Code for HLS

Function

HLS tool converts this function into a hardware

Top-Level I/O

HLS tool converts each argument of a top-level function into a physical connection, which is an IO interface port of the RTL

Types

The type of the physical connection is defined by the type of the argument

Influences the area and performance

```
void fir ( data_t *y,  
          coef_t c[4],  
          data_t x)
```



These predetermined types are supported by the Vitis HLS tool

C to RTL Conversion

- ▶ Does the HLS tool synthesize all parts of the C code in the same manner?

No, during C to RTL conversion, the HLS tool synthesizes different parts of the code differently

Top level functions

Loops

Other C functions

Arrays

The Key Attributes of C code

```

void fir(
    data_t *y,
    coef_t c[4],
    data_t x
){

    static data_t shift_reg[4];
    acc_t acc;
    int i;

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

```

Functions: All code is made up of functions which represent the design hierarchy: the same in hardware

Top Level IO: The arguments of the top-level function determine the hardware RTL interface ports

Types: All variables are of a defined type. The type can influence the area and performance

Loops: Functions typically contain loops. How these are handled can have a major impact on area and performance

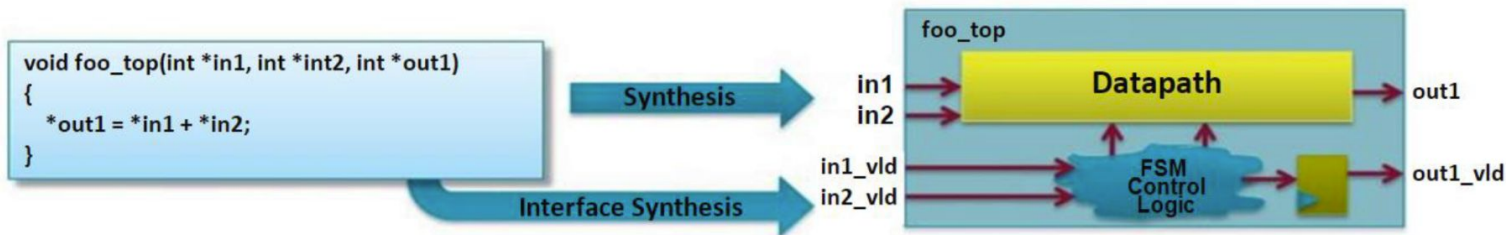
Arrays: Arrays are used often in C code. They can influence the device IO and become performance bottlenecks

Operators: Operators in the C code may require sharing to control area or specific hardware implementations to meet performance

Let's examine the default synthesis behavior of these ...

Top-Level Functions

- Top-level function arguments become I/O ports on the RTL designs
- These ports can optimally be implemented with an interface synthesis, hardware protocol
- The top-level function `foo_top` has two input arguments `in1` and `in2` and one output argument `out1`. These arguments have become the ports of the equivalent generated hardware



Other C Functions

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

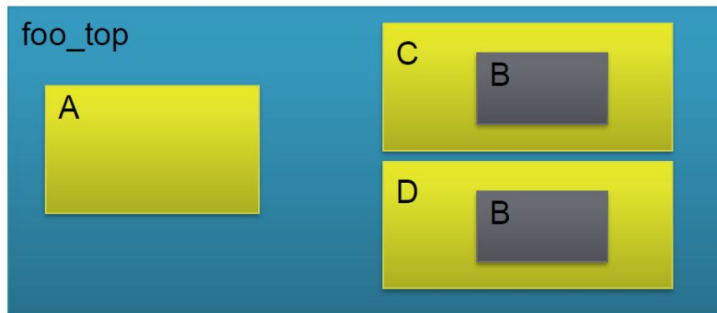
Source Code

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

my_code.c



RTL hierarchy



Each function/block can be shared like any other component (add, sub, etc) provided it's not in use at the same time

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

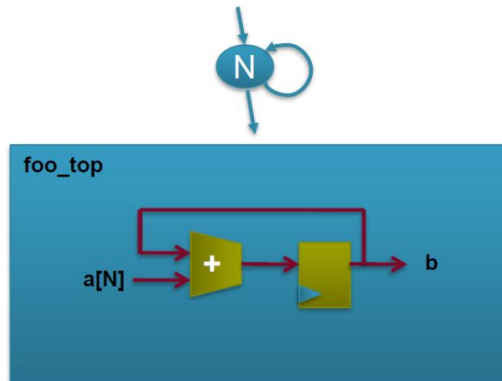
Loops

- ▶ In Vitis HLS, Loops in the C functions are kept rolled and are pipelined by default to improve performance
 - Each C loop iteration → Implemented in the same state
 - Each C loop iteration → Implemented with same resources

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

Loops require labels if they are to be referenced by Tcl directives
(GUI will auto-add labels)

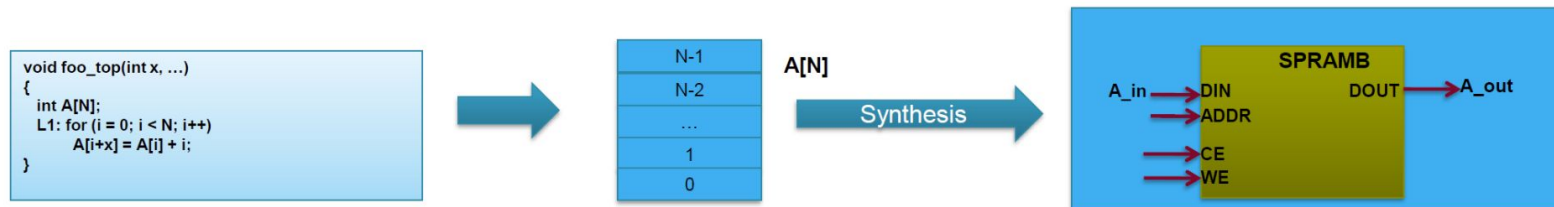
Synthesis



- Loops can be unrolled if their indices are statically determinable at elaboration time
 - Not when the number of iterations is variable
- Unrolled loops result in more elements to schedule but greater operator mobility
 - Let's look at an example

Arrays

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



- ▶ The array can be targeted to any memory resource in the library
 - The ports (Address, CE active high, etc.) and sequential operation (clocks from address to data out) are defined by the library model
 - All RAMs are listed in the Vitis HLS Library Guide: block RAM (BRAM), LUT RAM, or UltraRAM
- ▶ Arrays can be merged with other arrays and reconfigured
 - To implement them in the same memory or one of different widths & sizes
- ▶ Arrays can be partitioned into individual elements
 - Implemented as smaller RAMs or registers

Types = Operator Bit-sizes

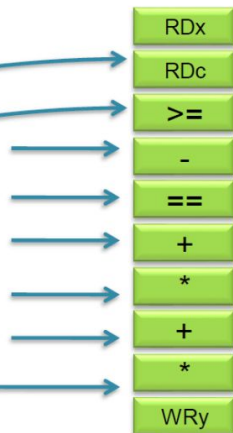
Code

```
void fir (
  data_t *y,
  coef_t c[4],
  data_t x
) {

  static data_t shift_reg[4];
  acc_t acc;
  int i;

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

Operations



Types

Standard C types

long long (64-bit)	short (16-bit)	unsigned types
int (32-bit)	char (8-bit)	
float (32-bit)	double (64-bit)	

Arbitrary Precision types

C++: ap_(u)int types (1-1024)
 ap_fixed types

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

From any C code example ...

Operations are
extracted...

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

Operators

- ▶ Operator sizes are defined by the type
 - The variable type defines the size of the operator
- ▶ Vitis HLS will try to minimize the number of operators
 - By default Vitis HLS will seek to minimize area after constraints are satisfied
- ▶ User can set specific limits & targets for the resources used
 - Allocation can be controlled
 - An upper limit can be set on the number of operators or cores allocated for the design: This can be used to force sharing
 - e.g limit the number of multipliers to 1 will force Vitis HLS to share



Use 1 mult, but take 4 cycle even if it could be done in 1 cycle using 4 mults

- Resources can be specified
 - The cores used to implement each operator can be specified
 - e.g. Implement each multiplier using a 2-stage pipelined core (hardware)

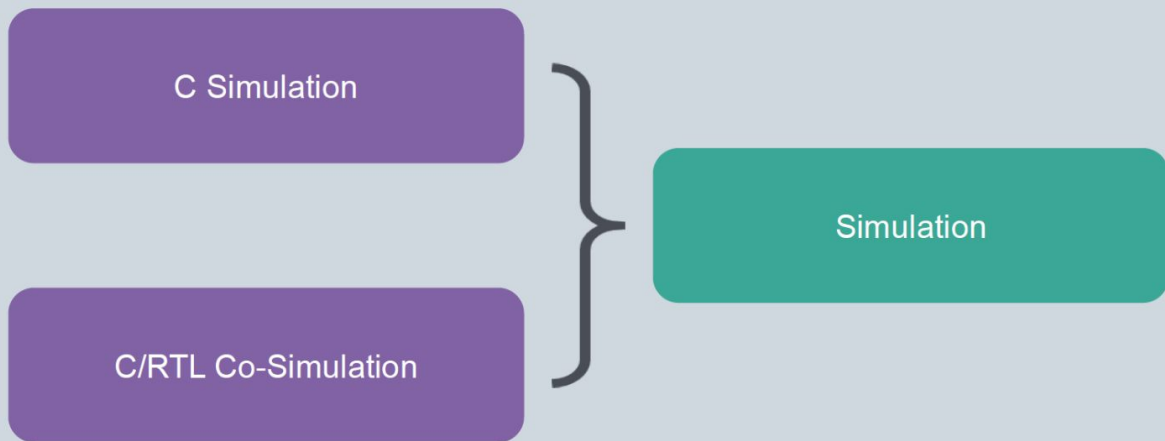


Same 4 mult operations could be done with 2 pipelined mults (with allocation limiting the mults to 2)

Validation and Verification Flow

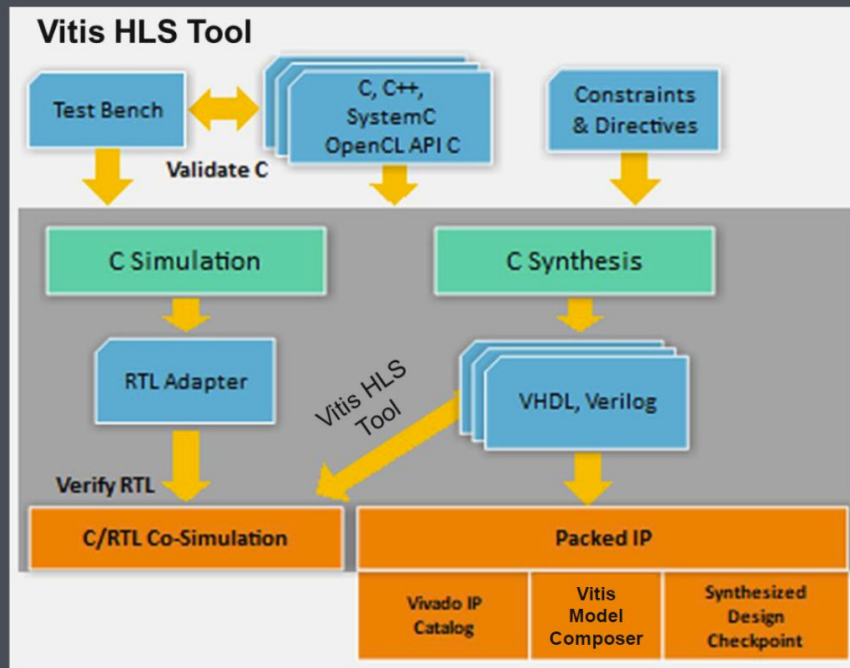
C Validation and RTL Verification

Verification of the design using the HLS tool is a two-step process



C Validation and RTL Verification

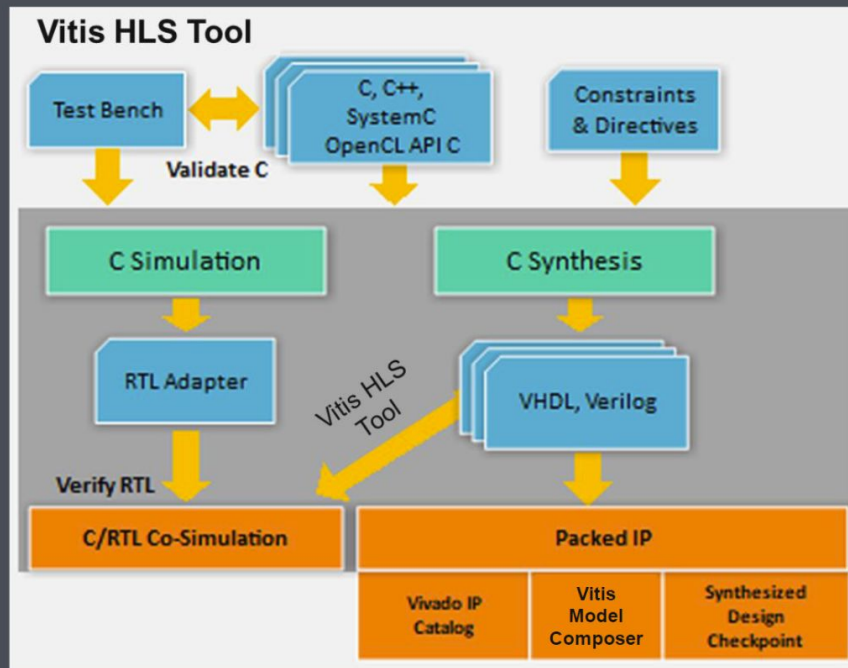
- At pre-synthesis stage, the C simulation checks the functionality of the C algorithm
- C validation is fast and free, and it uses the C test bench
- Post-synthesis verification is automated through the C/RTL co-simulation feature
 - Reuses the C test bench to perform verification on the output RTL



C Validation and RTL Verification

C/RTL co-simulation involves three steps:

- C simulation is executed and the inputs to the top-level function are saved as “input vectors”
- The “input vectors” are used in a C/RTL co-simulation using the RTL created by the Vitis HLS tool. The outputs from the RTL are saved as “output vectors”
- The “output vectors” are applied to the C test bench after the function for synthesis to verify the results are correct



C Function Test Bench

Xilinx has provided some best practices that should be followed when writing a C test bench:

C test bench should compare the results with known good values

Automatically confirms that the C validation and RTL verification is correct

Return value of the C function test bench is set to:

- Zero: If the results are correct
- Non-zero value: If the results are incorrect

```
int main () {  
  
    int ret=0;  
  
    ...  
    ret = system("diff -brief -w output.dat output.golden.dat");  
  
    if (ret != 0) {  
        printf("Test failed !!!\n");  
        ret=1;  
    } else {  
        printf("Test passed !\n");  
    }  
  
    ...  
    return ret;  
}
```

Determine or Create the Top-Level Function

In any C program, the top-level function is called main()

In the Vitis HLS tool design flow, we can specify any sub-function below main()

Guidelines

Only one function is allowed as the top-level function for synthesis

Any sub-functions in the hierarchy under the top-level function for synthesis are also synthesized

If we want to synthesize functions that are not in the hierarchy under the top-level function, we must merge the functions into a single top-level function for synthesis

Separate the test bench and the design file

Determine or Create the Top-Level Function

Given a case where functions func_A and func_B are to be implemented in FPGA

```
main.c
int main () {
    ...
    func_A(a,b,*i1);
    func_B(c,*i1,*i2);
    func_C(*i2,ret)
    return ret;
}
```

func_A
func_B
func_C

Re-partition the design to create a new single top-level function inside main()

```
main.c
#include func_AB.h
int main (a,b,c,d) {
    ...
    // func_A(a,b,i1);
    // func_B(c,i1,i2);
    func_AB (a,b,c, *i1, *i2);
    func_C(*i2,ret)
    return ret;
}
```

func_AB
func_C

Recommendation is to separate testbench and design files*

*Else add file as design file **and** testbench

```
func_AB.c
#include func_AB.h
func_AB(a,b,c, *i1, *i2) {
    ...
    func_A(a,b,*i1);
    func_B(c,*i1,*i2);
    ...
}
```

func_A
func_B

Hands-on session!

Tutorial – Getting started with HLS

We will look at a simple function to start

Instructions for this tutorial on [github](#)

Start at /day1/vadd

```
extern "C" {  
    void vadd(  
        const unsigned int *in1, // Read-Only Vector 1  
        const unsigned int *in2, // Read-Only Vector 2  
        unsigned int *out,       // Output Result  
        int size                 // Size in integer  
    )  
    {  
#pragma HLS INTERFACE m_axi port=in1 bundle=aximm1  
#pragma HLS INTERFACE m_axi port=in2 bundle=aximm2  
#pragma HLS INTERFACE m_axi port=out bundle=aximm1  
        for(int i = 0; i < size; ++i)  
        {  
            out[i] = in1[i] + in2[i];  
        }  
    }  
}
```

What can you expect in the coming days?

Monday, 23.03.2026

What is an FPGA? And why physicists care about them!
Motivation, Architecture, Electronics, Programming Model, Basics of HLS

Tuesday, 24.03.2026

Thinking in Hardware
Timing Terminology, Paradigms, HLS Data Types, Pointers, Streams
Lab: Matrix Multiplication, Pointers, and Streams

Wednesday, 25.03.2026

Interfaces and Improving Performance
Interfaces, Directives, Improving Throughput and Latency, Bottlenecks
Lab: YUV-Filter, Loops Optimizations

Thursday, 26.03.2026

Improving Area and Resources
Allocation & Structure Control, Hierarchies, Arbitrary Precision
Lab: DCT, Matrix Multiplication revisited

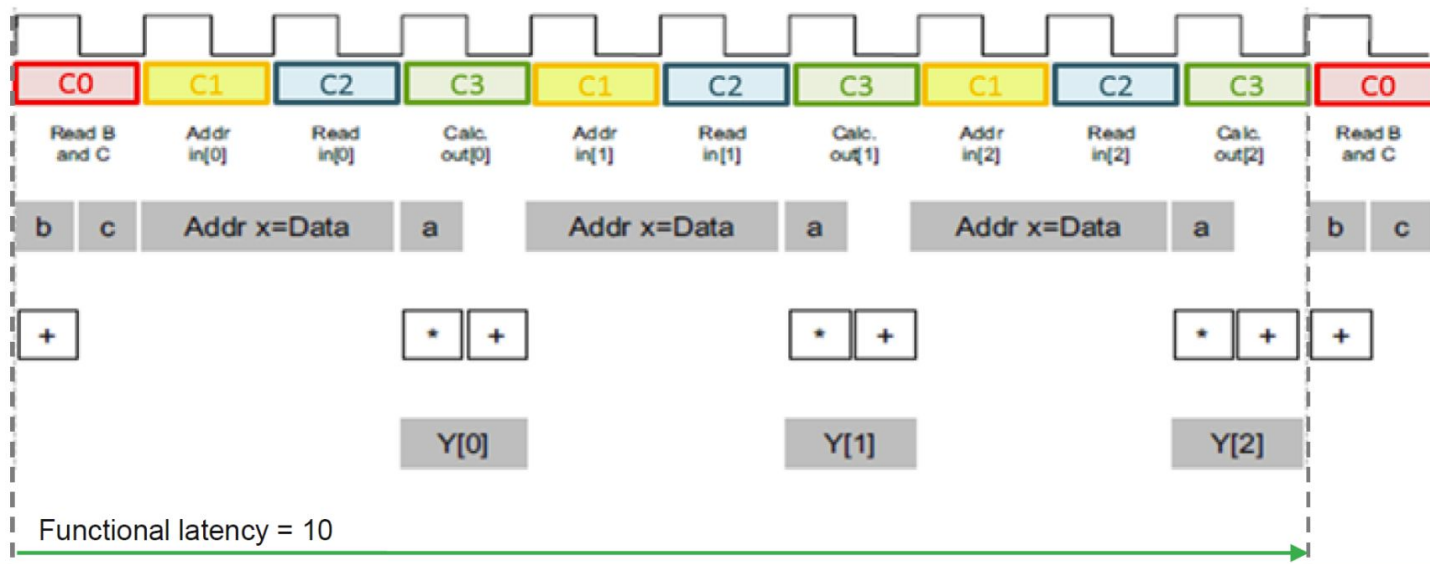
Friday, 27.03.2026

Task Level Parallelism, and how to deal with Timing Violations
Lab: Freely running kernel, Timing violation

Thinking in Hardware



Terminology for Measuring in Clock Cycles

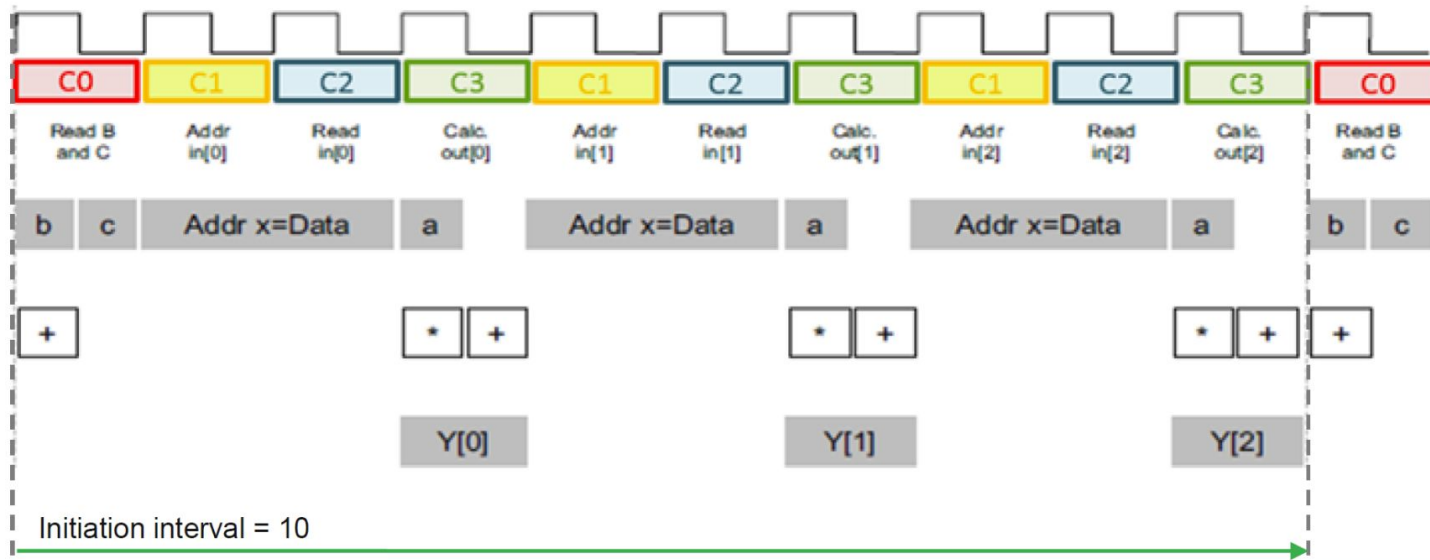


Latency: Number of clock cycles required for the function to go from input to output generation

When the output is an array, the latency is measured to the last array value output

- Ten clock cycles in this case

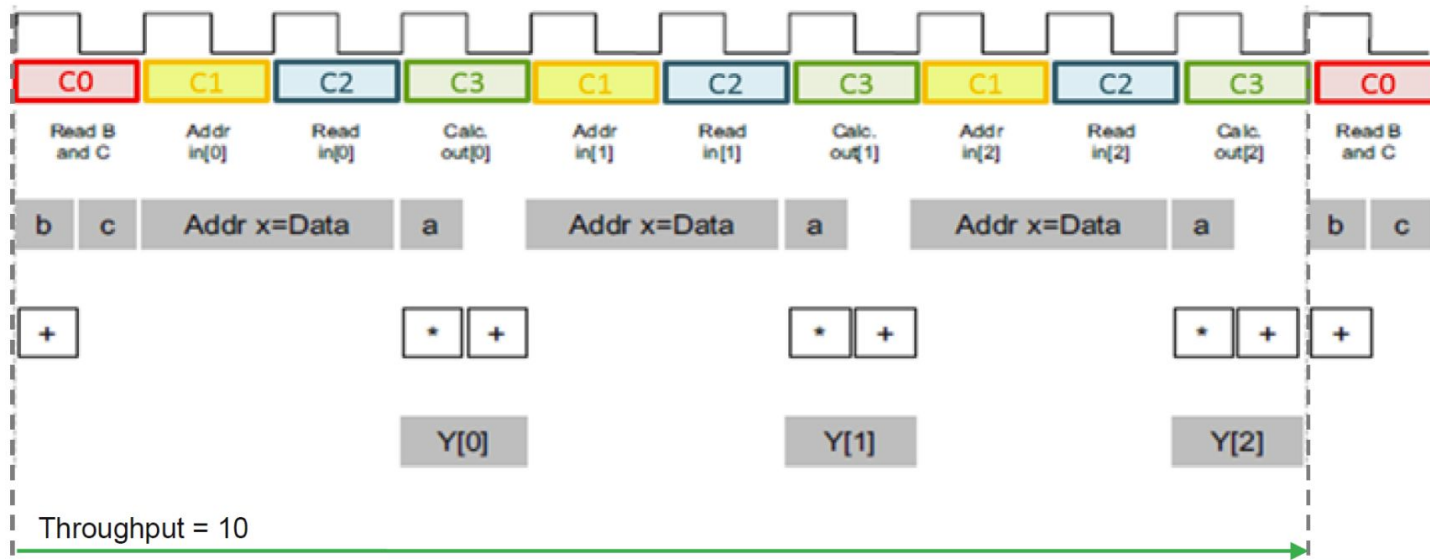
Terminology for Measuring in Clock Cycles



Initiation interval (II): Number of clock cycles before the function can accept new input data

- 10 clock cycles in this case

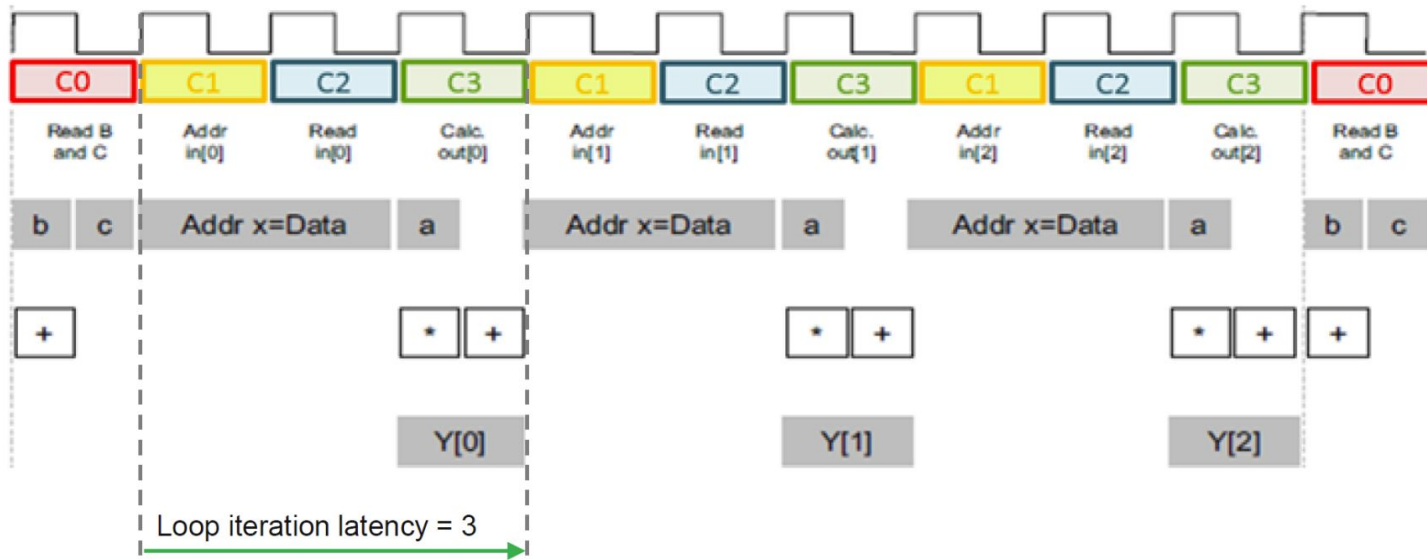
Terminology for Measuring in Clock Cycles



Throughput: Number of cycles between the new input samples

- 10 clock cycles in this case

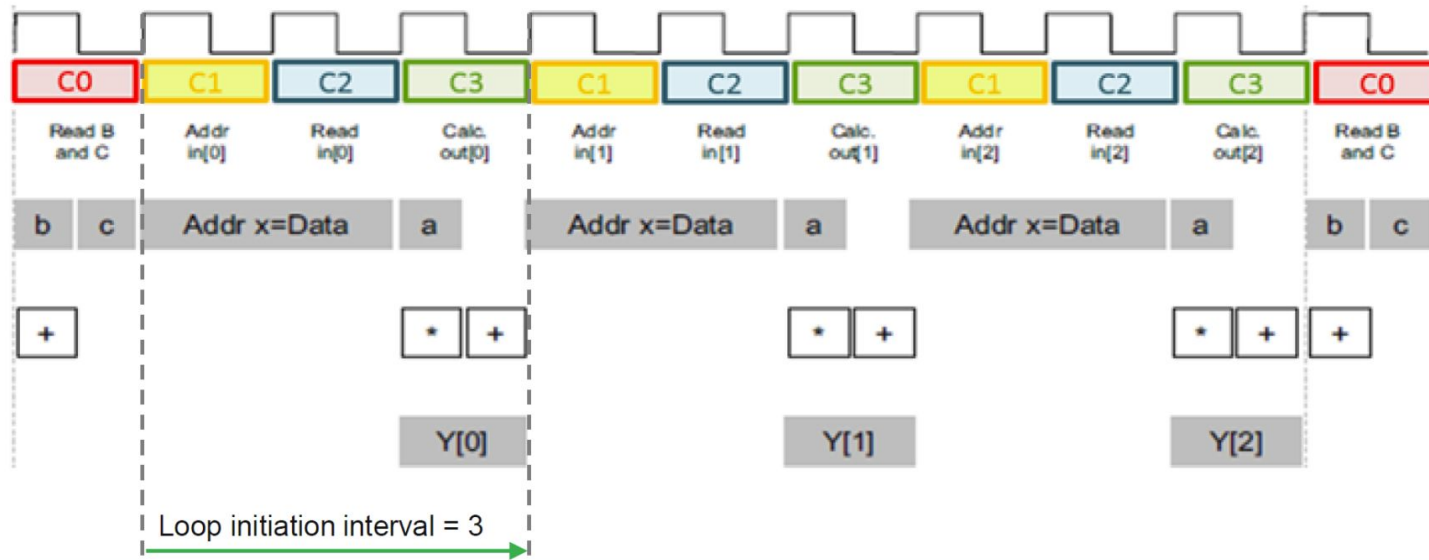
Terminology for Measuring in Clock Cycles



Loop iteration latency: It is the number of clock cycles it takes to complete one iteration of the loop

- Three clock cycles in this case

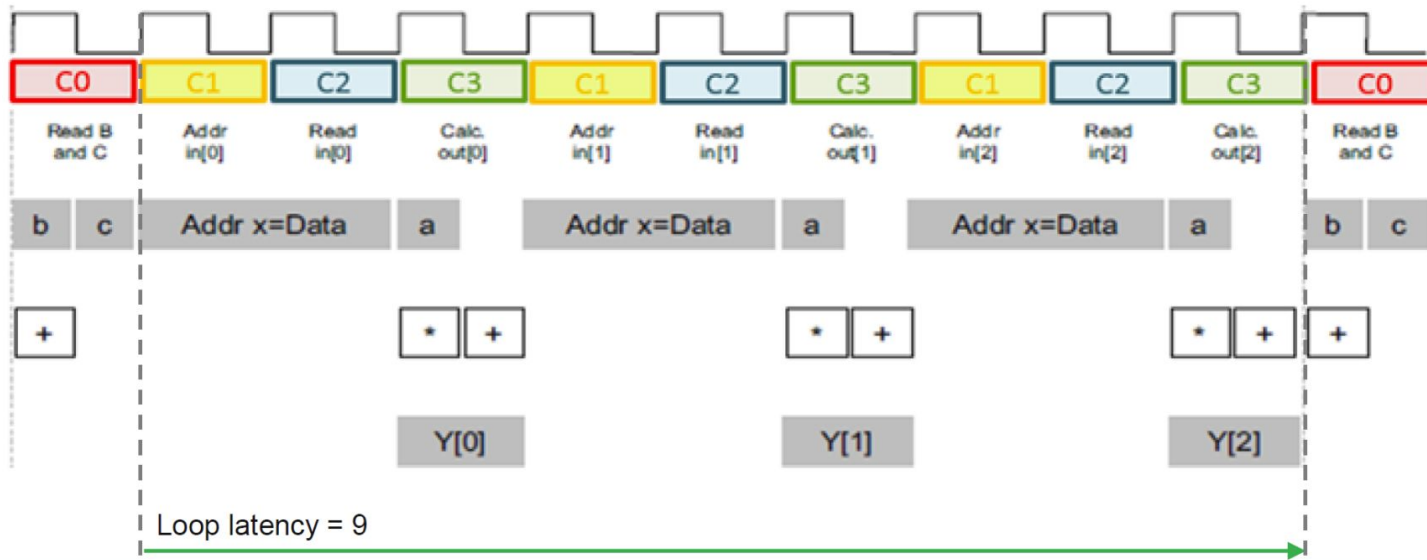
Terminology for Measuring in Clock Cycles



Loop initiation interval: Number of clock cycles before the next iteration of the loop starts to process data

- Three clock cycles in this case

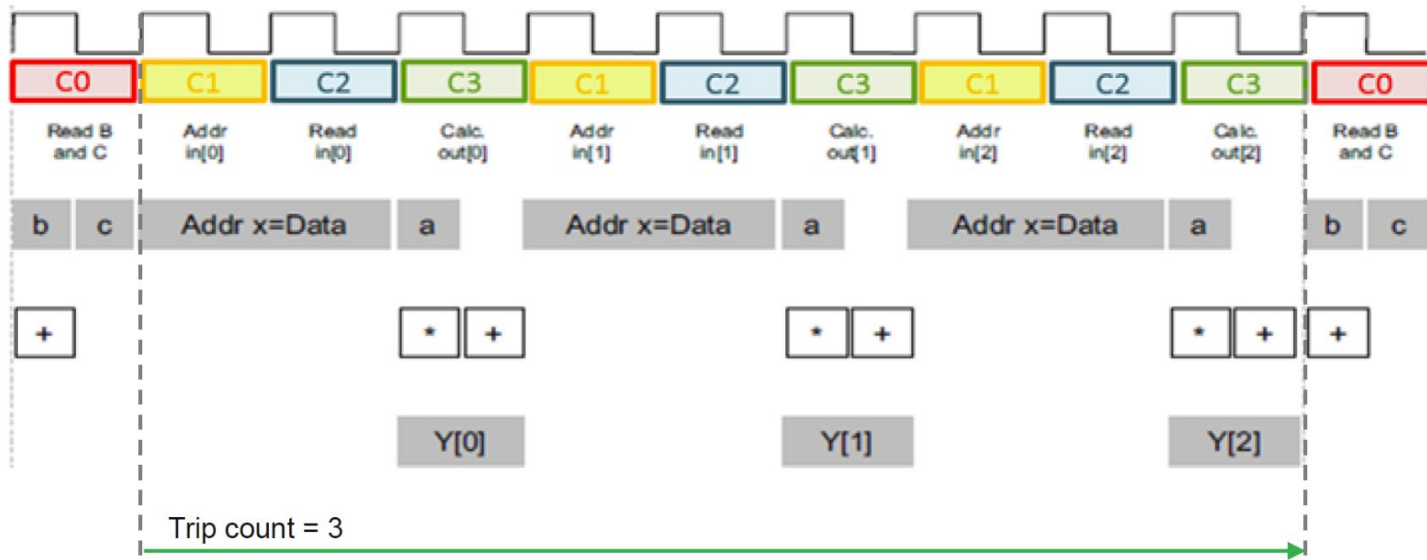
Terminology for Measuring in Clock Cycles



Loop latency: Number of cycles to execute all iterations of the loop

- Nine cycles in this case

Terminology for Measuring in Clock Cycles



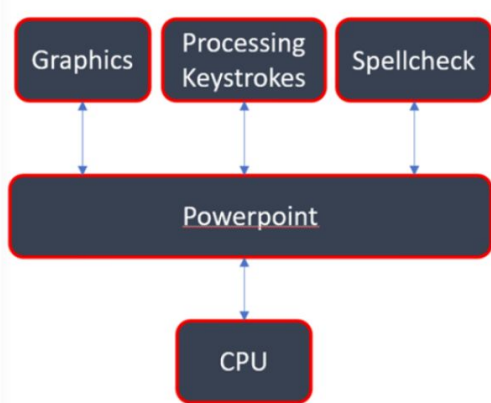
Trip count: Number of iterations in the loop; three in this case

Data rate: Equal to the $1/\text{throughput} \times \text{clock frequency}$

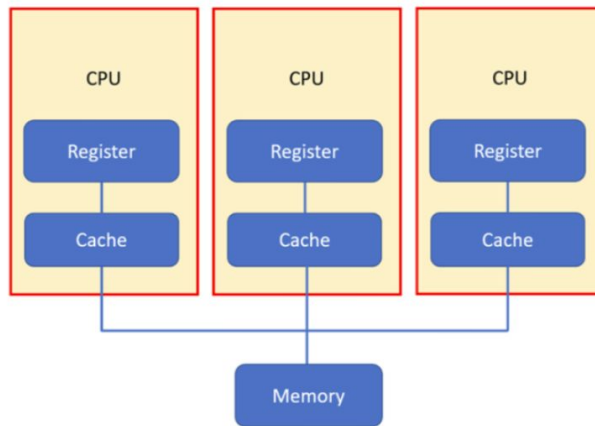
Three Paradigms for Programming FPGAs

Architecture Matters

- ▶ Multithreading and/or Multiprocessing can include multiple system processes, or it can consist of one process that has multiple threads within it.
- ▶ Multi-threaded programming using a shared memory system became very popular as it allowed the software developer to design applications with parallelism in mind but with a fixed CPU architecture.
- ▶ Multi-processing programming using multiple CPU cores and hyperthreading to improve throughput as shown in the figure on the right.

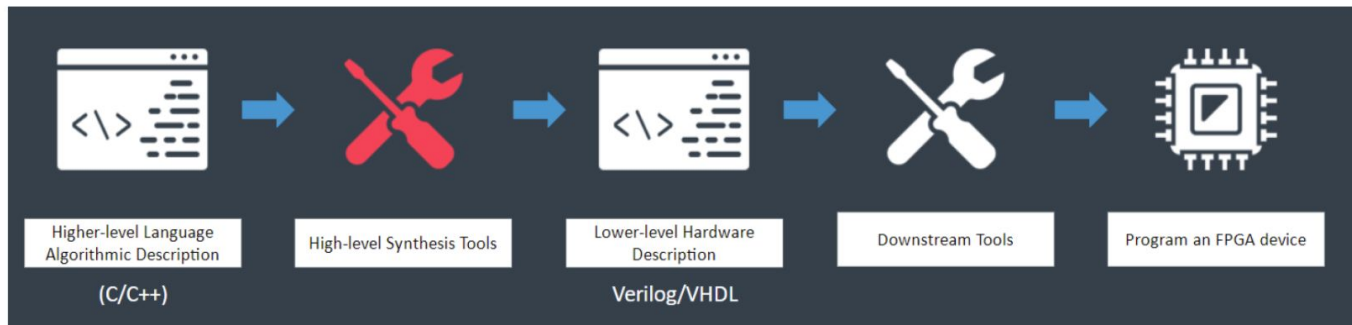


Multithreading



Multiprocessing

Three Paradigms for Programming FPGAs



- Retain the advantages of a programming language to write efficient code that can be translated into hardware
- Additional work (rewriting the code) require for the desired performance goals even if the C/C++ code can be automatically converted into hardware.

Producer-Consumer Paradigm

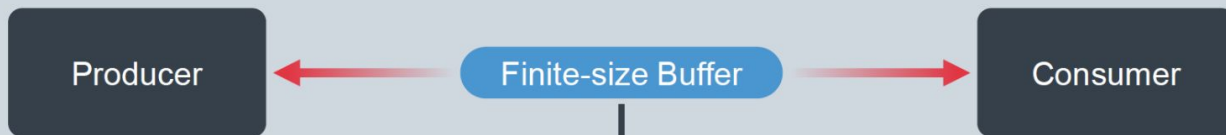
Stream data Paradigm

Pipelining Paradigm

Producer-Consumer Paradigm

Example

Single Producer & Single Consumer



Full Buffer

Producer blocks the data



Consumer removes an item for
producer to fill the buffer again



Empty Buffer



Consumer stalls



Producer puts data into the buffer
and wakes up the consumer

Producer-Consumer Paradigm

Advantages of using this macro-level architectural optimization

1

No worrying about memory models and other non-deterministic behavior

2

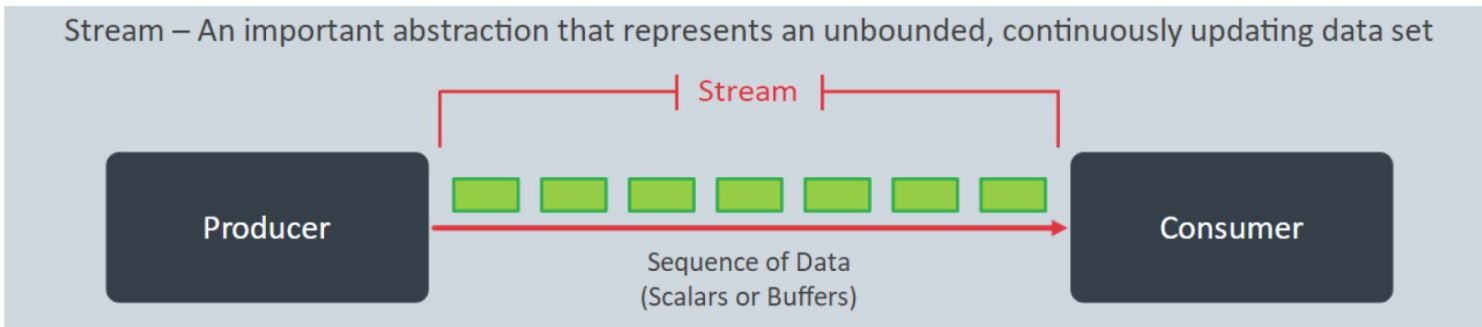
Dataflow network achieved

3

Abstracts away complexities of a parallel program

Performance of such a dataflow network relies on the designer's ability to continually feed data to the network such that data keeps streaming through the system

Stream data Paradigm

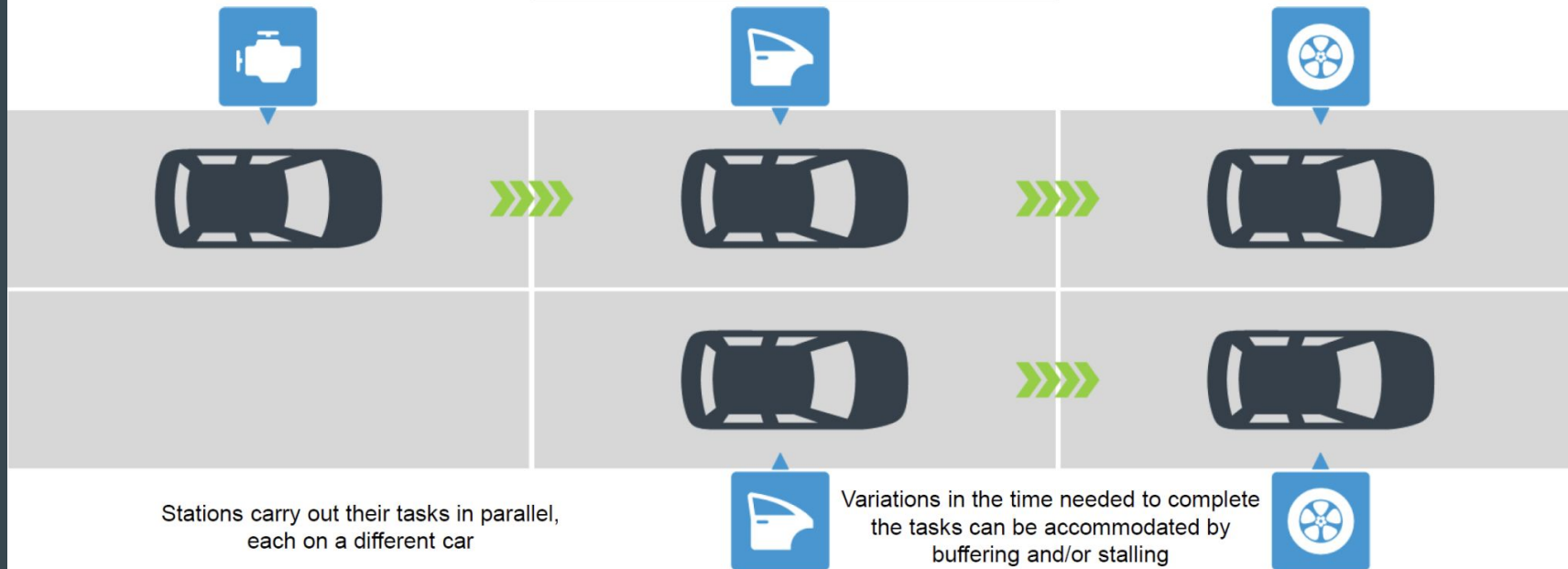


- Streaming paradigm forces you to think in terms of data access patterns/sequences
- Random memory access to data in software are virtually free; in hardware, it takes resources
- Make sequential memory accesses that can be converted into streams

Pipelining Paradigm

A classical micro-level architectural optimization that can be applied to multiple levels of abstraction

Example – Production line of a car factory



Divide the work and resources among the stages so all take the same time to complete their tasks

Pipelining Paradigm

Producer-consumer pipeline will only be efficient if each task produces/consumes data at a high rate

Static Optimization

Pipelining uses same resources to execute the same function over time and requires complete knowledge about the number of times the task is executed and the latency of each task

Low-level instruction pipelining technique cannot be applied to dataflow-type networks

Pipelining Paradigm

Software written for CPUs is fundamentally different from software written for FPGAs

Recommended high-level actions

Verify the source code changes

Focus on the macro-architecture and draw the desired activity timeline where the horizontal axis represents time

Start coding your program

Partition the original algorithm into smaller components

Have an overall vision about the rates of processing

Think about the granularity of the streaming

Vitis HLS Support

Comprehensive C Support

- ▶ A Complete C Validation & Verification Environment
 - Vitis HLS supports complete bit-accurate validation of the C model
 - Vitis HLS provides a productive C-RTL co-simulation verification solution
- ▶ Vitis HLS supports C, C++ and OpenCL API C kernel
 - Functions can be written in C and C++ 11/C++ 14
 - Wide support for coding constructs in C and C++ 11/C++ 14
- ▶ Modeling with bit-accuracy
 - Supports arbitrary precision types for all input languages
 - Allowing the exact bit-widths to be modeled and synthesized
- ▶ Floating point support
 - Support for the use of float and double in the code

Unsupported Constructs: Overview

- System calls
 - System calls cannot be synthesized : performing some tasks on the OS in which the C program is running
 - Vitis HLS tool ignores these commonly used system calls such as printf, fscanf..
 - Vitis HLS tool defines macro `_SYNTHESIS_`, which allows excluding non-synthesizable code from the design
- Dynamic Memory Usage
 - All the constructs of C are supported in HLS provided they are statically defined at compile time
 - If a function is not fully realized, it cannot be synthesized : `malloc()`, `alloc()`, `free()`
 - HLS tool doesn't support C++ objects that are dynamically created or destroyed
- Pointer Limitations
 - No support for general pointer casting but support for pointer casting between native C/C++ types
 - It supports pointer arrays for synthesis, provided that each pointers to a scalar or an array of scalar. Arrays of pointers cannot point to additional pointers
 - Function points are not supported
- Recursive Functions
- Standard Template Libraries

Library Support

- ▶ HLS C library allow common hardware design constructs and functions to be easily modeled in C and synthesized to RTL
- ▶ Use each of the C library in the design by including the library header file
- ▶ Four types of libraries as follows:
 - Arbitrary Precision Data Types Library
 - HLS Math library (hls_math.h)
 - HLS Streaming Library
 - HLS IP libraries
 - Vitis Vision Library

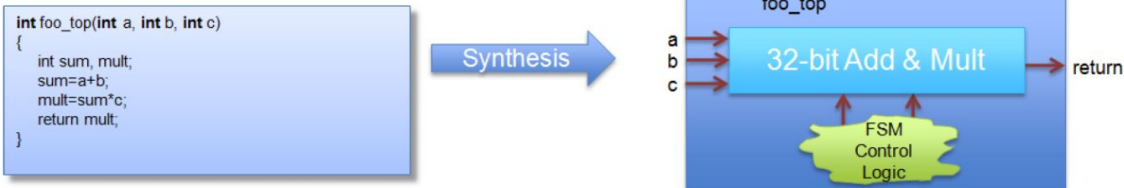
Data Types and Bit-Accuracy

Data Types and Bit-Accuracy

- ▶ C and C++ have standard types created on the 8-bit boundary
 - char (8-bit), short (16-bit), int (32-bit), long long (64-bit)
 - Also provides stdint.h (for C), and stdint.h and cstdint (for C++)
 - Types: int8_t, uint16_t, uint32_t, int_64_t etc.
 - They result in hardware which is not bit-accurate and can give sub-standard QoR
- ▶ Vitis HLS tool provides both integer and fixed-point arbitrary precision types for C++
 - Allow any arbitrary bit-width to be specified
 - Designers can:
 - Improve the QoR of the hardware by specifying exact data widths
 - Simulate the design to ensure that there is no loss of accuracy

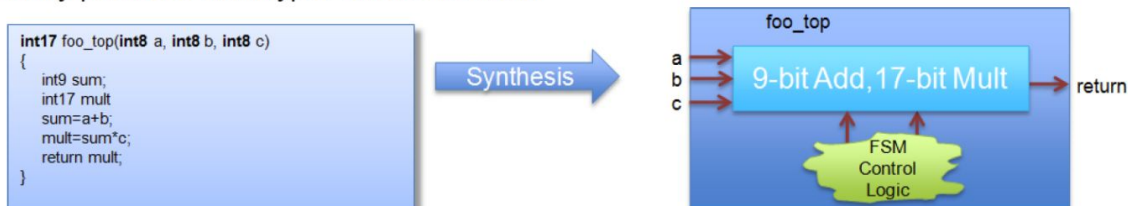
Why is arbitrary precision Needed?

- ▶ Code using native C int type



- ▶ However, if the inputs will only have a max range of 8-bit

- Arbitrary precision data-types should be used

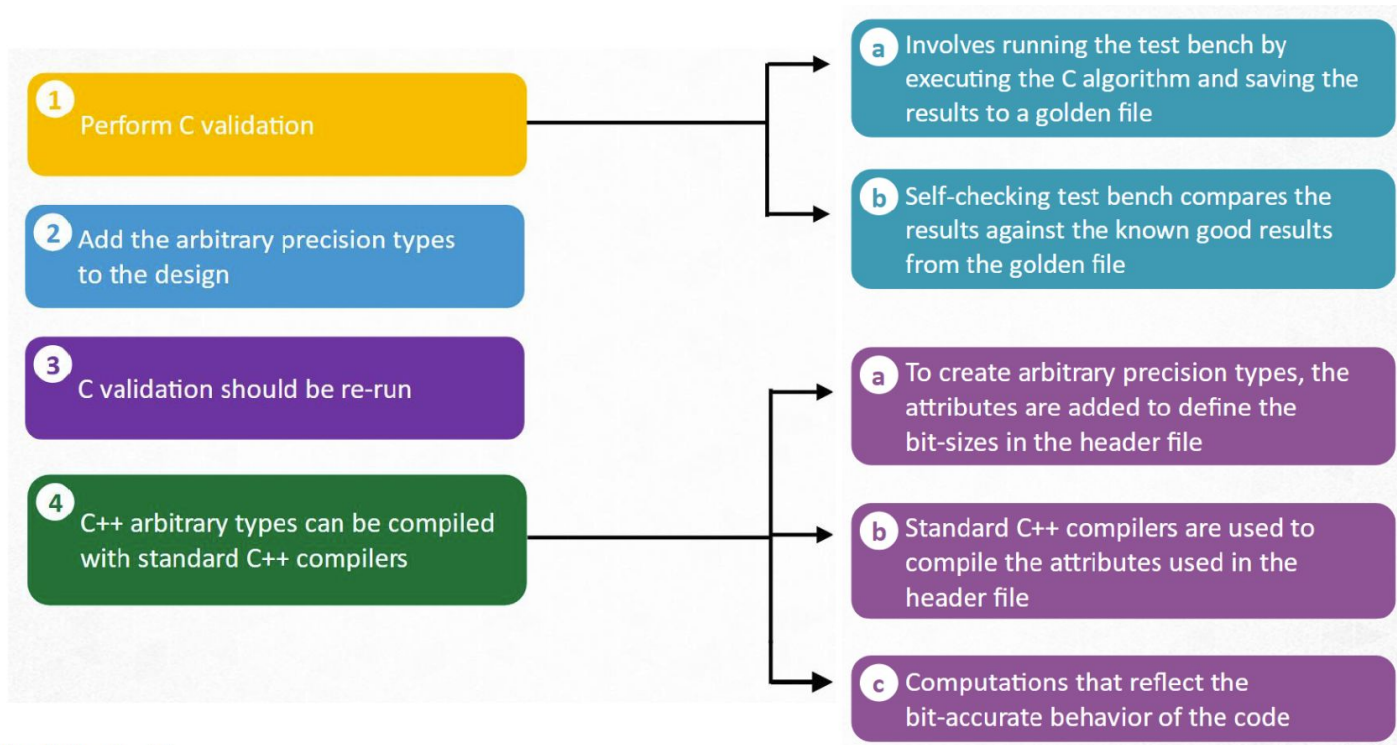


- It will result in smaller & faster hardware with the full required precision
- Full precision can be simulated/validated with C simulation and hardware will behave the same

HLS & C Types

- ▶ There are 3 basic types you can use for HLS
 - Standard C/C++ Types
 - Vitis HLS enhancements to C++: `ap_int`,
 - Vitis HLS enhancements to C++: `ap_fixed`

Arbitrary Precision Flow



Arbitrary Precision : C++ ap_int types

► For C++

- Vitis HLS types ap_int can be used
- Range: 1 to 1024 bits
 - Signed: ap_int<W>
 - Unsigned: ap_uint<W>
- The bit-width is specified by W
- User must include the ap_int.h header file in the source files

```
#include ap_int.h
```

Include header file

```
void foo_top (...) {
```

```
    ap_int<9>
```

```
    var1;    // 9-bit
```

```
    ap_uint<10>
```

```
    var2;    // 10-bit unsigned
```

► C++ compilation

- Use g++ at the Vitis HLS CLI (shell)
 - Include the path to the Vitis HLS tool header file

```
shell> g++ -o my_test test.c test_tb.c -I$VITIS_HLS_HOME/include
```

AP_INT operators & conversions

- Fully Supported for all Arithmetic operator

Operations	
Arithmetic	+ - * / % ++ --
Logical	~ !
Bitwise	& ^
Relational	> < <= >= == !=
Assignment	*= /= %= += -= <<= >>= &= ^= =

- Methods for type conversion

Methods		Example
To integer	Convert to an integer type	res = var.to_int();
To unsigned integer	Convert to an unsigned integer type	res = var.to_uint();
To 64-bit integer	Convert to a 64-bit long long type	res = var.to_int64();
To 64-bit unsigned integer	Convert to an unsigned long long type	res = var.to_uint64();
To double	Convert to double type	res = var.to_double();

AP_INT Bit Manipulation methods

Methods		Example
Length	Returns the length of the variable.	<code>res=var.length;</code>
Concatenation	Concatenation low to high	<code>res=var_hi.concat(var_lo);</code> Or <code>res= (var_hi,var_lo)</code>
Range or Bit-select	Return a bit-range from high to low or a specific bit.	<code>res=var.range(high bit,low bit);</code> Or <code>res=var[bit-number]</code>
(n)and_reduce	(N)And reduce all bits.	<code>bool t = var.and_reduce();</code>
(n)or_reduce	(N)Or reduce all bits	<code>bool t = var.or_reduce();</code>
X(n)or_reduce	X(N)or reduce all bits	<code>bool t = var.xor_reduce();</code>
Reverse	Reverse the bits in the variable	<code>var.reverse();</code>
Test bit	Tests if a bit is true	<code>bool t = var.test(bit-number)</code>
Set bit value	Sets the value of a specific bit	<code>var.set_bit(bit-number, value)</code>
Set bit	Set a specific bit to one	<code>var.set(bit-number);</code>
Clear bit	Clear a specific bit to zero	<code>var.clear(bit-number);</code>
Invert Bit	Invert a specific bit	<code>var.invert(bit-number);</code>
Rotate right	Rotate the N-bits to the right	<code>var.rrotate(N);</code>
Rotate left	Rotate the N-bits to the left	<code>var.lrotate(N);</code>
Bitwise Invert	Invert all bits	<code>var.b_not();</code>
Test sign	Test if the sign is negative (return true)	<code>bool t = var.sign();</code>

Arbitrary Precision : C++ ap_fixed types

- ▶ Support for fixed point datatypes in C++
 - Include the path to the ap_fixed.h header file
 - Both signed (ap_fixed) and unsigned types (ap_ufixed)

```
#include ap_fixed.h
void foo_top (...) {
    ap_fixed<9, 5, AP_RND_CONV, AP_SAT> var1;           // 9-bit,
                                                         // 5 integer bits, 4 decimal places
    ap_ufixed<10, 7, AP_RND_CONV, AP_SAT> var2;         // 10-bit unsigned
                                                         // 7 integer bits, 3 decimal places
```

- ▶ Advantages of Fixed Point types
 - The result of variables with different sizes is automatically taken care of
 - The binary point is automatically aligned
 - Quantization: Underflow is automatically handled
 - Overflow: Saturation is automatically handled

Alternatively, make the result variable large enough such that overflow or underflow does not occur

AP_FIXED operators & conversions

- Fully Supported for all Arithmetic operator

Operations	
Arithmetic	+ - * / % ++ --
Logical	~ !
Bitwise	& ^
Relational	> < <= >= == !=
Assignment	*= /= %= += -= <<= >>= &= ^= =

- Methods for type conversion

Methods		Example
To integer	Convert to an integer type	res = var.to_int();
To unsigned integer	Convert to an unsigned integer type	res = var.to_uint();
To 64-bit integer	Convert to a 64-bit long long type	res = var.to_int64();
To 64-bit unsigned integer	Convert to an unsigned long long type	res = var.to_uint64();
To double	Convert to double type	res = var.to_double();
To ap_int	Convert to an ap_int	res = var.to_ap_int();

AP_FIXED methods

► Methods for bit manipulation

Methods		Example
Length	Returns the length of the variable.	<code>res=var.length;</code>
Concatenation	Concatenation low to high	<code>res=var_hi.concat(var_lo);</code> Or <code>res= (var_hi,var_lo)</code>
Range or Bit-select	Return a bit-range from high to low or a specific bit.	<code>res=var.range(high bit,low bit);</code> Or <code>res=var[bit-number]</code>

Quantization Modes: Rounding

- ▶ **AP_RND_ZERO**: rounding to zero
 - For positive numbers, the redundant bits are truncated
 - For negative numbers, add MSB of removed bits to the remaining bits.
 - The effect is to round towards zero.
 - 01.01 (1.25 using 4 bits) rounds to 01.0 (1 using 3 bits)
 - 10.11 (-1.25 using 4 bits) rounds to 11.0 (-1 using 3 bits)

- ▶ **AP_RND_CONV**: rounded to the nearest value
 - The rounding depends on the least significant bit
 - If the least significant bit is set, rounding towards plus infinity
 - Otherwise, rounding towards minus infinity
 - 00.11 (0.75 using 4-bit) rounds to 01.0 (1.0 using 3-bit)
 - 10.11 (-1.25 using 4-bit) rounds to 11.0 (-1.0 using 3-bit)

Quantization Modes: Truncation

- ▶ **AP_TRN: truncate**
 - Remove redundant bits. Always rounds to minus infinity
 - This is the default.
 - 01.01(1.25) → 01.0 (1)

- ▶ **AP_TRN_ZERO: truncate to zero**
 - For positive numbers, the same as AP_TRN
 - For positive numbers: 01.01(1.25) → 01.0(1)
 - For negative numbers, round to zero
 - For negative numbers: 10.11 (-1.25) → 11.0(-1)

Overflow Modes

- ▶ Overflow mode
 - Determines the behavior when an operation generates more bits than can be satisfied by the MSB

- ▶ Overflow Modes (saturation)
 - AP_SAT, AP_SAT_ZERO, AP_SAT_SYM

- ▶ Overflow Modes (wrap)
 - AP_WRAP, AP_WRAP_SM
 - The number of saturation bits, N, is considered when wrapping

Overflow Mode: Saturation

- ▶ **AP_SAT: saturation**
 - This overflow mode will convert the specified value to MAX for an overflow or MIN for an underflow condition
 - MAX and MIN are determined from the number of bits available
- ▶ **AP_SAT_ZERO: saturates to zero**
 - Will set the result to zero, if the result is out of range
- ▶ **AP_SAT_SYM: symmetrical saturation**
 - In 2's complement notation one more negative value than positive value can be represented
 - If it is desirable to have the absolute values of MIN and MAX symmetrical around zero, AP_SAT_SYM can be used
 - Positive overflow will generate MAX and negative overflow will generate -MAX
 - 0110(6) => 011(3)
 - 1011(-5) => 101(-3)

Overflow Mode: Wrap Sign Magnitude

▶ AP_WRAP_SM, N = 0

- This mode uses sign magnitude wrapping
- Sign bit set to the value of the least significant deleted bit
- If the most significant remaining bit is different from the original MSB, all the remaining bits are inverted
- IF MSBs are same, the other bits are copied over
 - Step 1: First delete redundant MSBs. 0100(4) => 100(-4)
 - Step 2: The new sign bit is the least significant bit of the deleted bits. 0 in this case
 - Step 3: Compare the new sign bit with the sign of the new value
- If different, invert all the numbers. They are different in this case
 - 011 (3) 11

▶ AP_WRAP_SM, N > 0

- Uses sign magnitude saturation
- Here N MSBs will be saturated to 1
- Behaves similar to case where N = 0, except that positive numbers stay positive and negative numbers stay negative

Floating Point Support

- ▶ Synthesis for floating point
 - Data types (IEEE-754 standard compliant)
 - Single-precision
 - 32 bit: 24-bit fraction, 8-bit exponent
 - Double-precision
 - 64 bit: 53-bit fraction, 11-bit exponent
- ▶ Support for Operators
 - Vitis HLS supports the Floating Point (FP) cores for each Xilinx technology
 - If Xilinx has a FP core, Vitis HLS supports it
 - It will automatically be synthesized
 - If there is no such FP core in the Xilinx technology
 - Design should be re-coded to fixed-point types

Using Floating-Point Types Example

- ▶ Floats and doubles are commonly used with math.h for C and cmath.h for C++

double	foo_d = 3.1459;	Using ap_fixed requires: • C++ • \$VITIS_HLS_HOME/include/ap_fixed.h
float	foo_f = 3.1459 ;	
ap_fixed<14,4>	foo_fx = -1.4142;	
int	foo_i = 42;	

- ▶ While converting C functions to C++ to take advantage of math.h support, be sure that the new C++ code compiles correctly before synthesizing with the Vitis HLS tool

extern "C" float sqrtf(float);	Required if it is a C++ function
--------------------------------	----------------------------------

- ▶ Using a sqrtf(function)
 - Remove the need for the type converters in the hardware/Save the area and improves the timing

double	var_d = sqrt(foo_d);	//64-bit sqrt core
float	var_f = sqrtf(foo_f);	//This will lead to a single precision core
	var_f = sqrt(foo_f);	//Still 64-bit, with format conversion cores(single to double and back)

ap_fixed<14,4>	var_fx = sqrtf(foo_fx);	//fixed-point to single precision conversion	Using sqrt instead of sqrtf would imply a single-to-double conversion and back
		//Fixed -> 32-bit sqrt core -> floating to fixed conversion	
int	var_i = sqrtf(foo_i);	//int to float conversion Int -> 32-bit sqrt -> float to int	

Support for Math Functions

- ▶ Many other functions from the C/C++ standard math library for which are no floating-point operator cores available
 - These functions are implemented in a bit-approximate manner
 - Results for these functions may differ within a few units of least precision(ULP) to the C/C++ standards
- ▶ Stand C math library
 - C simulation results and the C/RTL co-simulation results may be different
- ▶ Hls_math.h library
 - C simulation and C/RTL co-simulation results are identical
 - However, there results of C simulation are not the same as those using standard C libraries
- **Recommendation**
 - Replace math.h or cmath.h with hls_math.h or to add the files hls_lib.c and keep math/cmath.h

Support for Math Functions

Function	Float	Double	Accuracy (ULP)	Implementation Style
fabs	Supported	Supported	Exact	Synthesized
fabsf	Supported	Not Applicable	Exact	Synthesized
floorf	Supported	Not Applicable	Exact	Synthesized
fmax	Supported	Supported	Exact	Synthesized
fmin	Supported	Supported	Exact	Synthesized
logf	Supported	Not Applicable	1	Synthesized
floor	Supported	Supported	Exact	Synthesized
fpclassify	Supported	Supported	Exact	Synthesized
isfinite	Supported	Supported	Exact	Synthesized
isinf	Supported	Supported	Exact	Synthesized
isnan	Supported	Supported	Exact	Synthesized
isnormal	Supported	Supported	Exact	Synthesized
log	Supported	Supported	1 for float, 16 for double	Synthesized
log10	Supported	Supported	2 for float, 3 for double	Synthesized
1/x (reciprocal)	Supported	Supported	Exact	LogiCORE IP
recip	Supported	Supported	1	Synthesized
recipf	Supported	Not Applicable	1	Synthesized
round	Supported	Supported	Exact	Synthesized
rsqrt	Supported	Supported	1	Synthesized
rsqrtf	Supported	Not Applicable	1	Synthesized
1/sqrt (reciprocal sqrt)	Supported	Supported	Exact	LogiCORE IP
signbit	Supported	Supported	Exact	Synthesized
sin	Supported	Supported	10	Synthesized
sincos	Supported	Supported	1 for float, 5 for double	Synthesized
sincosf	Supported	Not Applicable	1	Synthesized
sinf	Supported	Not Applicable	1	Synthesized
sinhf	Supported	Not Applicable	6	Synthesized
sqrt	Supported	Supported	Exact	LogiCORE IP
tan	Supported	Supported	20	Synthesized
tanf	Supported	Not Applicable	3	Synthesized
trunc	Supported	Supported	Exact	Synthesized

Arbitrary Precision Floating-Point Library `ap_float<W,E>`

- `W` = total bitwidth of variable (Max 80 bits)
- `E` = bitwidth of the exponent (from 4 to 16 bits)
- 1 bit sign + `W-E-1` bits for fraction bits (from 4 to 64 bits)
- e.g. C++ float: `ap_float<32,8>`
- Supports common C++ floating-point arithmetic operators
- No automatic conversion from `ap_float<>` to `float`!
- Accumulation of `ap_float<>` possible with helper class `ap_float_acc<>`, using `ap_fixed<>` underlying datatype

Bit-Width Propagation

If the arguments to a function are sized accurately, Vitis HLS can propagate this information through the design. There is no need to create arbitrary precision types for every variable.

→ recommendation: correctly size the arguments of all functions! no need to size local variables

```
#include "ap_int.h"
```

```
ap_int<24> foo(int x, int y) {  
    int tmp;
```

```
    tmp = (x * y);  
    return tmp
```

```
}
```

32b mul with truncation to 24 bits

```
typedef ap_int<12> din_t;  
typedef ap_int<24> dout_t;
```

```
dout_t func_sized(din_t x, din_t y)
```

```
{  
    int tmp;
```

```
    tmp = (x * y);  
    return tmp
```

```
}
```

24b mul

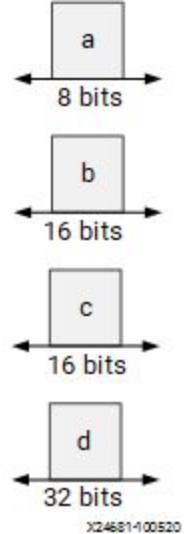
Composite data types

Structs

- Structs in the code are disaggregated by default. They are decomposed into separate objects for each of their member elements.
- Use AGGREGATE pragma to collect all elements in a single wide vector
 - will be padded to align on a 4-byte boundary
 - first element is aligned on the Most Significant Bit (MSB), final element on the Least Significant Bit (LSB)
- Structs in the interface are aggregated by default
 - Exception: struct contains stream (only allowed in Vivado IP flow, not in Vitis kernel flow)

```
struct example {  
  ap_int<5> a;  
  unsigned short int b;  
  unsigned short int c;  
  int d;  
};  
void foo()  
{  
  example s0;  
  #pragma HLS disaggregate variable=s0  
}
```

Disaggregated struct



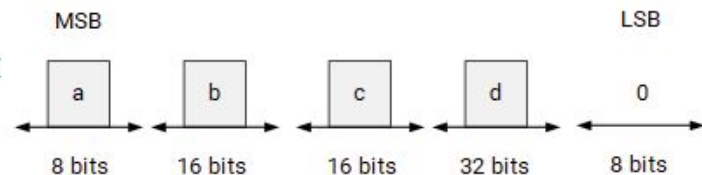
Aligned structs

- Data types with custom width are allocated with sizes which are a power of 2
- `bool` type will also be padded to 8 bits!
- Padding depends on order and size of elements

```
// 96 bit  
struct data_t {  
    short varA;  
    int varB;  
    short varC;  
};
```

```
// 64 bit  
struct data_t {  
    short varA;  
    short varC;  
    int varB;  
};
```

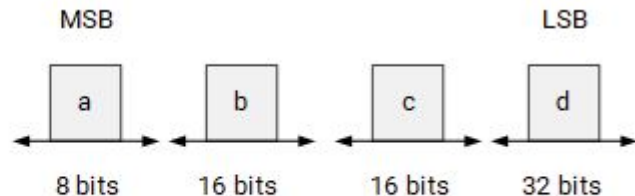
```
struct __attribute__((aligned(2))) example {  
    ap_int<5> a;  
    unsigned short int b;  
    unsigned short int c;  
    int d;  
};
```



Packed structs

- Packs elements so that the size of the struct is based on the actual size of each element
- Can also be achieved using `compact=bit` option of the `AGGREGATE` pragma

```
struct_attribute__((packed)) example {  
    ap_int<5> a;  
    unsigned short int b;  
    unsigned short int c;  
    int d;  
};
```



X24680-102220

C++ classes and templates

- Fully supported, but top-level for synthesis must be a function!
- Instantiate the class in a top-level function instead.
- Global variables are not recommended to be used in classes
- Templates can be used to create unique instances
- Static variables in a template function are duplicated for each different value of template arguments

```
#include "cpp_FIR.h"

// Top-level function with class instantiated
data_t cpp_FIR(data_t x)
{
    static CFir<coef_t, data_t, acc_t> fir1;

    cout << fir1;

    return fir1(x);
}
```


Enumerated types

```
#include <stdio.h>
```

```
enum mad_layer {
```

```
    MAD_LAYER_I    = 1,
```

```
    MAD_LAYER_II   = 2,
```

```
    MAD_LAYER_III  = 3
```

```
};
```

```
enum mad_mode {
```

```
    MAD_MODE_SINGLE_CHANNEL = 0,
```

```
    MAD_MODE_DUAL_CHANNEL  = 1,
```

```
    MAD_MODE_JOINT_STEREO  = 2,
```

```
    MAD_MODE_STEREO        = 3
```

```
};
```

```
enum mad_emphasis {
```

```
    MAD_EMPHASIS_NONE = 0,
```

```
    MAD_EMPHASIS_50_15_US = 1,
```

```
    MAD_EMPHASIS_CCITT_J_17 = 3
```

```
};
```

```
typedef    signed int mad_fixed_t;
```

```
typedef struct mad_header {
```

```
    enum mad_layer layer;
```

```
    enum mad_mode mode;
```

```
    int mode_extension;
```

```
    enum mad_emphasis emphasis;
```

```
    unsigned long long bitrate;
```

```
    unsigned int samplerate;
```

```
    unsigned short crc_check;
```

```
    unsigned short crc_target;
```

```
    int flags;
```

```
    int private_bits;
```

```
} header_t;
```

```
typedef struct mad_frame {
```

```
    header_t header;
```

```
    int options;
```

```
    mad_fixed_t sbsample[2][36][32];
```

```
} frame_t;
```

```
# define MAD_NSBSAMPLES(header) \
```

```
    ((header)->layer == MAD_LAYER_I ? 12 : \
```

```
    ((header)->layer == MAD_LAYER_III && \
```

```
    ((header)->flags & 17)) ? 18 : 36))
```

```
void types_composite(frame_t *frame);
```

- enum used as argument in top-level: 32b
- Internal used: optimized for required bits

Type qualifiers

Volatile

- Impacts how many reads or writes are performed when pointers are accessed multiple times on function interfaces
- Accesses preserved (top-level)
 - No burst
 - No port widening
 - No dead-code elimination
- Arbitrary precision types do not support volatile for arithmetics

Static

- Hold values between function calls (register / flip-flop)
- Value must be maintained across invocations of function
- NOT true that ONLY static variables become registers
- Initialization per default to 0 (or explicitly set)

Const

- Value of variable is never updated
- Must be initialized
- Vitis removes unnecessary hardware for consts
- Arrays → ROM
- ROMs are also automatically inferred if never written to array

Dead code example

The C/C++ undefined behaviors can lead to a different behavior in simulation and synthesis.

In the example you should not expect that `A[i]` gets the value of `val` from the previous loop iteration if neither `i==0`, nor `(cond)` are true. You should even not expect that the increment (`val++`) will happen.

```
for (int i=0; i<N; i++) {  
    int val; // uninitialized value  
    if (i==0) val=0;  
    else if (cond) val=1;  
    // val may have indeterminate value here  
    A[i] = val; // undefined behavior  
    val++; // dead code  
}
```

Solution: either initialize `val` at each iteration, or move the declaration of `val` above the loop

Global variables

- Fully synthesizable, can be freely used in the code
→ not inferred as arguments to the top-level function!
- Example uses three global variables (idx, Ain, Aout)

```
#include "top.h"

void top(const int idx, const int Ain[N], int Aout[Nhalf]) {

    int Aint[N];

    // Move elements in the input array

    ILOOP: for (int i = 0; i < N; i++) {

        int iadj = (i + idx) % N;

        Aint[i] = Ain[i] + Ain[iadj];

    } // end ILOOP

    // sum the 1st and 2nd halves
    OLOOP: for (int i = 0; i < Nhalf; i++) {

        Aout[i] = (Aint[i] + Aint[Nhalf + i]);

    } // end OLOOP

} // end top()
```

Vector data types `hls::vector<T, N>`

- T: The type of elements that the vector holds, can be a user-defined type which must provide common arithmetic operations.
- N: The number of elements that the vector holds, must be a positive integer.
- The best performance is achieved when both the bit-width of T and N are integer powers of 2

Vector data types

- Can be used to define SIMD operations
- Mapped to hardware during synthesis that will be executed in parallel

```
typedef hls::vector<int, 8> t_int8Vec;
t_int8Vec intVectorA, intVectorB;
.
.
.
void processVecStream(hls::stream<t_int8Vec>
&inVecStream1, hls::stream<t_int8Vec> &inVecStream2,
hls::stream<int8Vec> &outVecStream)
{
    for(int i=0; i<32; i++)
    {
        #pragma HLS pipeline II=1
        t_int8Vec aVec = inVecStream1.read();
        t_int8Vec bVec = inVecStream2.read();
        //performs a vector operation on 8 integers
        in parallel
        t_int8Vec cVec = aVec * bVec;
        outVecStream.write(cVec);
    }
}
```

Using Pointers with Limitation

Using Pointers

Used extensively in C code and are well supported for synthesis

Used as arguments to the top-level function

Important to understand

How pointers are implemented during synthesis because they can sometimes cause issues in achieving the desired RTL interface

If the designer uses a pointer in multiple functions, then Vitis HLS inlines all these functions and tries to increase the run time

Using Pointers

Structs as Pointers

Structs can be implemented differently as a pointer or as a pass-by-value argument

Pointer Arithmetic

Introducing the pointer arithmetic limits the possible interfaces that can be synthesized in RTL

No sequential access to pointer data

This interface supplies an address with which to index the data when the data is accessed

Converting Pointers Using Malloc

Fixed-sized resources can be created, and the existing pointer can simply be made to point to these fixed-sized resources

This technique can prevent manual recoding of the existing design

Multi-Access Pointers

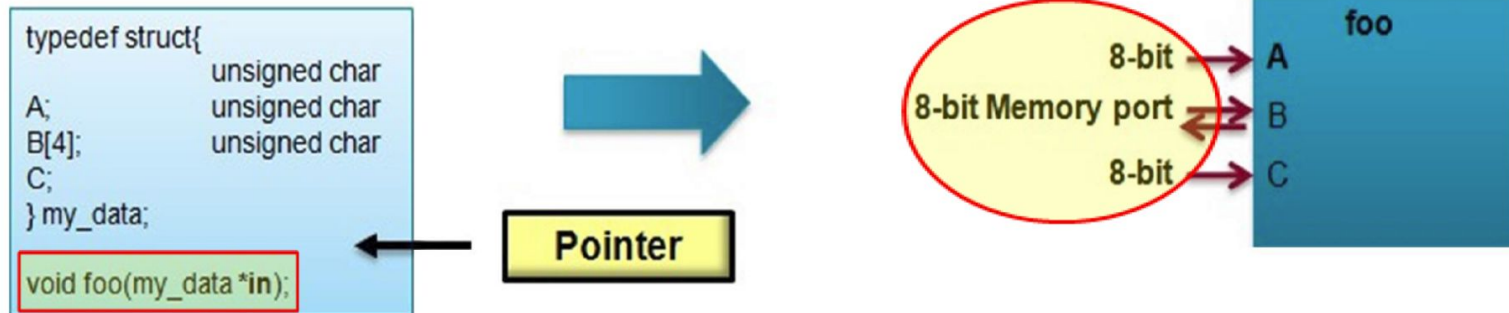
Multiple accesses occur when a pointer is read from or written to multiple times in the same function

Volatile qualifier can be used on any function argument that is accessed multiple times

In the case of multi-access pointers, C code cannot be rigorously validated with C simulation

Requires a depth specification for RTL simulation

Structs as Pointers Passed as a Pointer

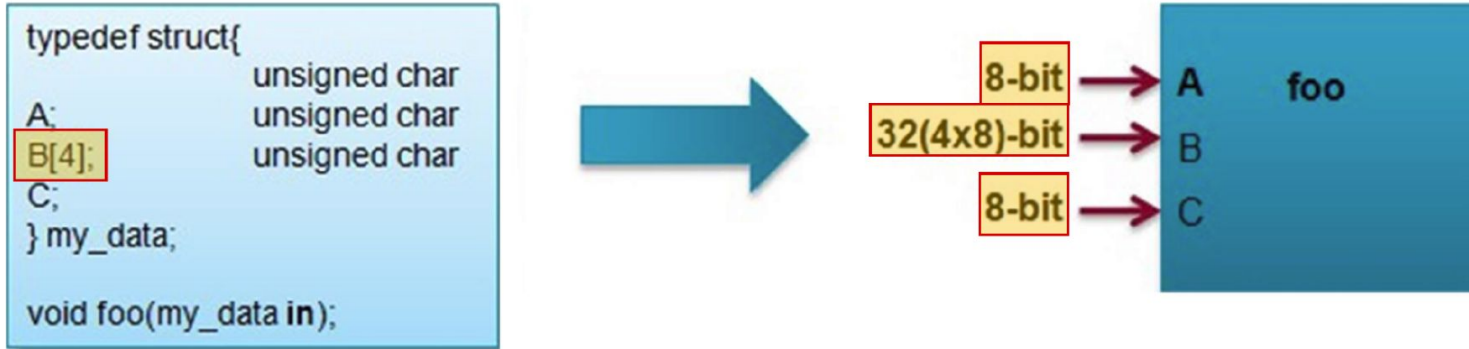


8-bit port for each variable with 'char' data type

Port size for the B variable is $4 \times 8 = 32$ bits

When a struct is passed as a pointer, the data will be accessed in a **pass-by-reference method** → port created for all members of the struct is 8 bits each

Structs as Pointers Passed by Pass-by-Value Method



When a struct is passed by pass-by-value method, the data will be accessed in a **pass-by-value method** → port created for A and C = 8 bits, but the port created for B = 32 bits

Since B is an array of 4 char members, each member of the array takes 8 bits, and B takes a total of 32 bits

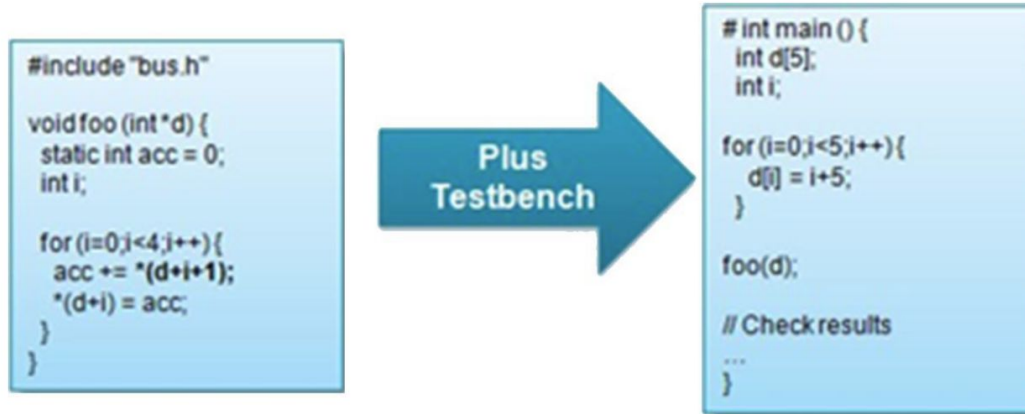
Structs as Pointers

No limitations in the size or complexity of the structs

There can be as many array dimensions and as many members in a struct as required

Only limitation with the implementation of the structs is when the arrays are to be implemented as streaming

Pointer Arithmetic

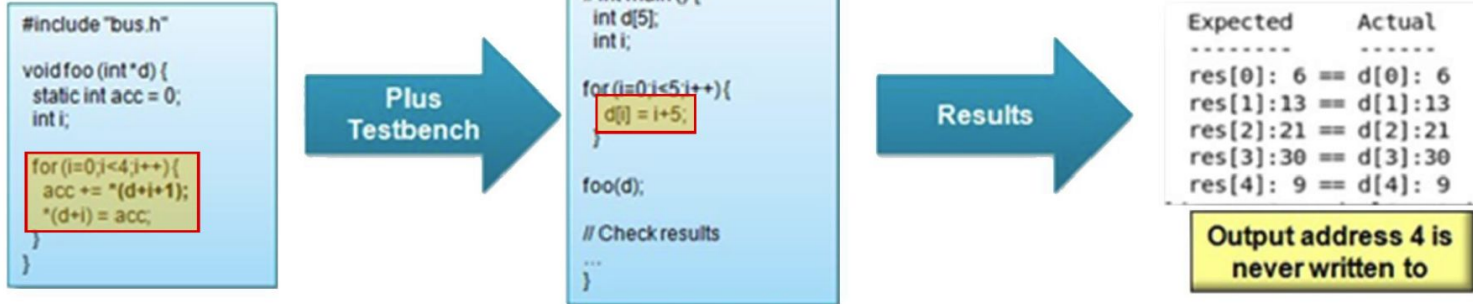


Simple pointer arithmetic used to accumulate the data values

Introducing the pointer arithmetic limits the possible interfaces that can be synthesized in RTL

Wire, handshake, and FIFO interfaces → data must arrive and is written in sequence, starting from element zero

Pointer Arithmetic



In the code example → output address 4 is never written

Pointer Arithmetic

If wire, handshake, and FIFO interfaces are used in this case → RTL co-simulation fails

```
Expected    Actual
-----
res[0]: 6 != d[0]: 5
res[1]:13 != d[1]: 6
res[2]:21 != d[2]: 7
res[3]:30 != d[3]: 8
res[4]: 9 == d[4]: 9
SystemC: simulation stopped by user.
@E [SIM-3] Simulation failed: testbench return error code "1".
@E [SIM-1] *** AutoSim finished: FAIL ***
```

**Only a bus interface can be used
when I/O pointers use non-trivial
arithmetic**

- Modify the code with an array on the interface
- Implement this in synthesis with ap_memory
- Interface can index the data with an address and can perform out-of-order, or nonsequential, accesses

Converting Pointers Using Malloc

Any system calls that manage the memory allocation are using resources that exist in the memory

System calls are created and released during run time

To synthesize hardware

Design must be fully self-contained, specifying all the required resources

To synthesize hardware which uses malloc()

Must be transformed into the equivalent bounded representations of a defined size, such as an array

Coding changes can impact the functionality of the design

Converting Pointers Using Malloc

Workaround

1. Add the user-defined macro NO_SYNTN to the code
 - Used to select between the synthesizable and non-synthesizable versions
2. Enable macro NO_SYNTN, execute the C simulation, and save the results
3. Disable the macro and execute the C simulation to verify that the results are identical
4. Perform synthesis with the user-defined macro disabled

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

dout_t malloc_removed(din_t din[N], dsel_t width) {
  #ifdef NO_SYNTN
    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
  int i, j;

  LOOP_SHIFT: for (i=0; i<N-1; i++) {
    if (i<width)
      *(array_local+i)=din[i];
    else
      *(array_local+i)=din[i]>>2;
  }
  *out_accum=0;
  LOOP_ACCUM: for (j=0; j<N-1; j++) {
    *out_accum += *(array_local+j);
  }

  return *out_accum;
}
```

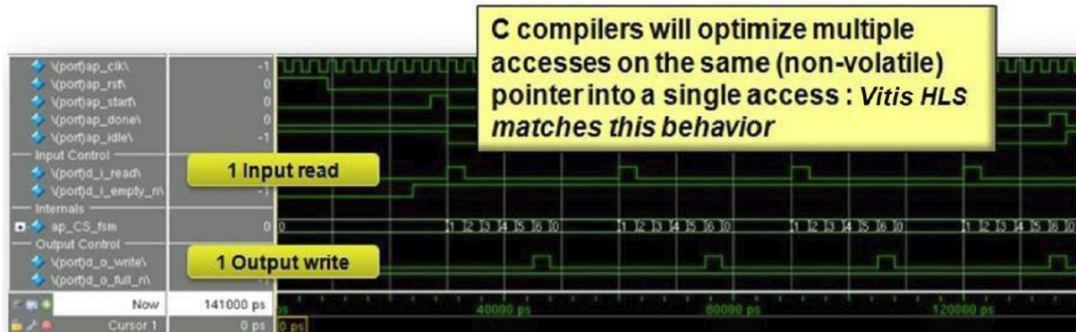
Multi-Access Pointers

```
void fifo (
    int*d_o,
    int*d_i

){
    static int acc = 0;
    int cnt;

    acc += *d_i;
    acc += *d_i;
    *d_o = acc;

    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
}
```



- Vitis HLS tool matches the behavior of the gcc compiler
- Optimizes these reads and writes into a single read and a single write operation on each port when the RTL is examined
- To make this design read and write to the RTL ports multiple times, there is a need to use a volatile qualifier

Multi-Access Pointers

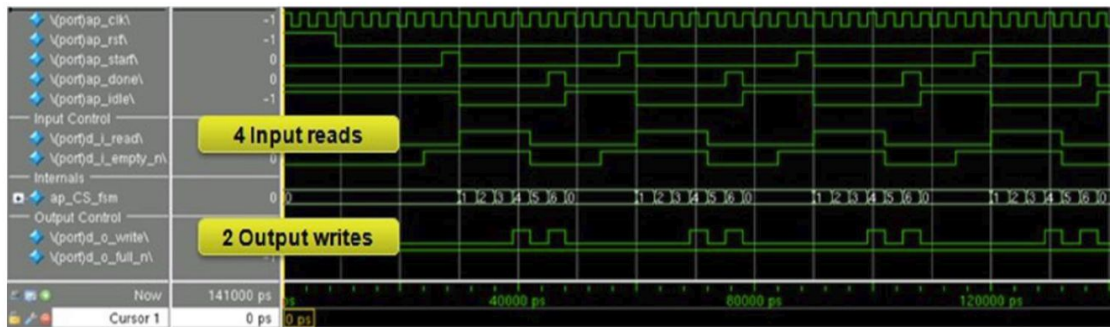
```

void fifo (
    volatile int *d_o,
    volatile int *d_i
){
    static int acc = 0;
    int cnt;

    acc += *d_i;
    acc += *d_i;
    *d_o = acc;

    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
}

```



- Performs four reads, but the same data is read four times
- Performs two separate writes, each with the correct data, but the test bench captures data only for the final write

Streams

Designing with Streams

- ▶ Streams can be used instead of multi-access pointers
 - None of the issues
- ▶ Streams simulate like an infinite FIFO in software
 - Implemented as a FIFO of user-defined size in hardware
- ▶ Streams have support for multi-access
 - Streams interface to the testbench
 - Streams can be read in the testbench to check the intermediate values
 - Streams store values: no chance of the volatile effect
- ▶ Streams are supported on the interface and internally
 - Define the stream as static to make it internal only

Using Streams

- ▶ Streams are C++ classes
 - Modeled in C++ as an infinite depth FIFO
 - Can be written to or read from
- ▶ Ideal for Hardware Modeling
 - Ideal for modeling designs in which the data is known to be streaming (data is always in sequential order)
 - Video or Communication designs typically stream data
 - No need to model the design in C++ as a “frame”
 - Streams allow it to be modeled as per-sample processing
 - Just fill the stream in the test bench
 - Read and write to the stream as if it's an infinite FIFO
 - Read from the stream in the test bench to confirm results
- ▶ Streams are by default implemented as a FIFO of depth 2
 - Can be specified by the user: needed for decimation/interpolation designs

Stream Example

- ▶ Create using `hls::stream` or define the `hls` namespace

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

typedef ap_uint<128> uint128_t;          // 128-bit user defined type

hls::stream<uint128_t> my_wide_stream; // A stream declaration
```

```
#include <ap_int.h>
#include <hls_stream.h>
using namespace hls;                    // Use hls namespace

typedef ap_uint<128> uint128_t;          // 128-bit user defined type

stream<uint128_t> my_wide_stream;        // hls:: no longer required
```

- ▶ Blocking and Non-Block accesses supported

```
// Blocking Write
hls::stream<int> my_stream;

int src_var = 42;
my_stream.write(src_var);
// OR use: my_stream << src_var;
```

```
// Blocking Read
hls::stream<int> my_stream;

int dst_var;
my_stream.read(dst_var);
// OR use: dst_var = my_stream.read();
```

```
hls::stream<int> my_stream;

int src_var = 42;
bool stream_full;

// Non-Blocking Write
if (my_stream.write_nb(src_var)) {
    // Perform standard operations
} else {
    // Write did not happen
}
```

```
// Full test
stream_full = my_stream.full();
```

```
hls::stream<int> my_stream;

int dst_var;
bool stream_empty;

// Non-Blocking Read
if (my_stream.read_nb(dst_var)) {
    // Perform standard operations
} else {
    // Read did not happen
}

// Empty test
fifo_empty = my_stream.empty();
```

Hands-on session!

Lab – Matrix Multiplication, Pointers, and Streams

Code for this lab again on [github](#)

Start at /day2, you will find

- /matmul: Matrix multiplication → repeat steps of the tutorial [part4](#) and [part5](#)
Can you find ways to optimize this kernel?
- /pointers: Working with pointers in HLS → steps of [part 4](#) only
/basic_arithmetic: example for illustration → any issues?
/malloc: does this work? if not, what is the error? how can you fix it?
- /streams: Implement a moving average calculation with a FIFO → steps of [part 4](#)

What can you expect in the coming days?

Monday, 23.03.2026

What is an FPGA? And why physicists care about them!
Motivation, Architecture, Electronics, Programming Model, Basics of HLS

Tuesday, 24.03.2026

Thinking in Hardware
Timing Terminology, Paradigms, HLS Data Types, Pointers, Streams
Lab: Matrix Multiplication, Pointers, and Streams

Wednesday, 25.03.2026

Interfaces and Improving Performance
Interfaces, Directives, Improving Throughput and Latency, Bottlenecks
Lab: YUV-Filter, Loops Optimizations

Thursday, 26.03.2026

Improving Area and Resources
Allocation & Structure Control, Hierarchies, Arbitrary Precision
Lab: DCT, Matrix Multiplication revisited

Friday, 27.03.2026

Task Level Parallelism, and how to deal with Timing Violations
Lab: Freely running kernel, Timing violation

Interfaces and Improving Performance

Improving throughput

FPGA

#pragma HLS

#pragma IN5

#S USES ARRAY_PARTITION

BUFFER

BUFFER

MEMORY

MEMORY

PERFORMANCE

Improving latency

FPGA

#pragma HLS

#pragma HLS PIPELINE

#pragma IN5 HS INE

MEMORY

COMPUTE

#pragma HLS

#pragma HLS

#S ININE



Managing Interface

The Key Attributes of C code : IO

Code

```
void fir (
    data_t *y,
    coef_t c[4],
    data_t x
){
    static data_t shift_reg[4];
    acc_t acc;
    int i;

    acc=0;
    loop: for (i=3;i>=0;i--) {
        if (i==0) {
            acc+=x*c[0];
            shift_reg[0]=x;
        } else {
            shift_reg[i]=shift_reg[i-1];
            acc+=shift_reg[i] * c[i];
        }
    }
    *y=acc>>2;
}
```

Functions: All code is made up of functions which represent the design hierarchy: the same in hardware

Input & Outputs: The arguments of the top-level function must be transformed to hardware interfaces with an IO protocol

Types: All variables are of a defined type. The type can influence the area and performance

Loops: Functions typically contain loops. How these are handled can have an impact on area and performance.

Arrays: Arrays are used often in C code. They can impact the device area and become performance bottlenecks.

Operators: Operators in the C code may require sharing to control area or be assigned to specific hardware implementations to meet performance

Example : Combinational Design

▶ Simple Adder Example

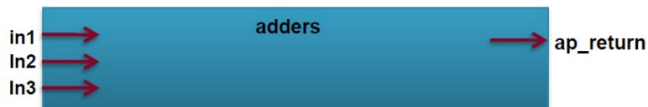
- Output is the sum of 3 inputs

```
#include "adders.h"
int adders(int in1, int in2, int in3) {

    int sum;
    sum = in1 + in2 + in3;
    return sum;
}
```

▶ Synthesized with 100 ns clock

- All adders can fit in one clock cycle
- **Combinational design**



- The function return becomes RTL port ap_return
- No handshakes are required or created in this example

=====

Performance Estimates

=====

+ Summary of timing analysis:

* Estimated clock period (ns): 3.45

+ Summary of overall latency (clock cycles):

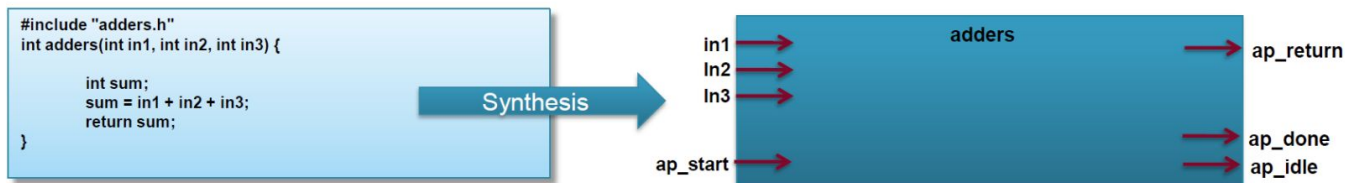
* Best-case latency: 0

* Average-case latency: 0

* Worst-case latency: 0

Example : Sequential Design

- ▶ The same adder design is now synthesized with a 3ns clock
 - The design now takes more than 1 cycle to complete
 - Vitis HLS creates a **sequential** design with the default port types



- ▶ By Default ..
 - Block level handshake signals are added to the design
 - ap_start: when to start
 - ap_done: when the design has completed the operation
 - ap_idle: when the design is idle

Vitis HLS IO Options

▶ Vitis HLS has four types of IO

1. Data ports created by the original top-level C function arguments
2. IO protocol signals added at the Block-Level
3. IO protocol signals added at the Port-Level
4. IO protocol signals added externally as IP Interfaces

▶ Data Ports

- These are the function arguments/parameters

▶ Block-Level Interfaces (optional)

- An interface protocol which is added at the block level
- Controls the addition of block level control ports: start, idle, done, and ready

▶ Port-Level interfaces (optional)

- IO interface protocols added to the individual function arguments

▶ IP interfaces (optional)

- Added as external adapters when exported as an IP

Interface Synthesis

- ▶ Type of interface depends on
 - Data type
 - Direction of the parameters of the top-level function
 - Target flow for the active solution
 - Default interface configuration settings
 - Any specified INTERFACE pragmas or directives
- ▶ Interface defines three elements of the kernel
 - Channels for data to flow into or out of the HLS design
 - Port protocol that is used to control the flow of data through the data channel, defining when the data is valid and when it can be read or written (port-level protocols)
 - Execution control scheme for the HLS design, specifying the operation of the kernel or IP as pipelined or sequential (block-level protocols)

Block Level Protocol

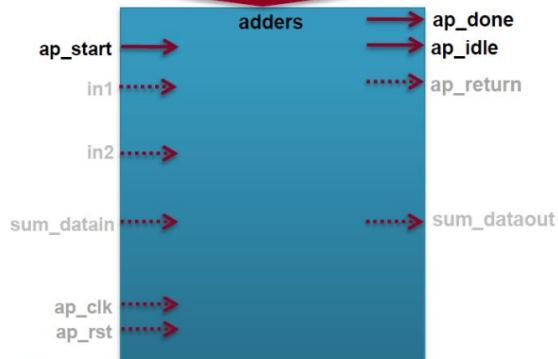
► Adder Example

```
#include "adders.h"
int adders(int in1, int in2, int *sum) {

    int temp;
    *sum = in1 + in2 + *sum;
    temp = in1 + in2;

    return temp;
}
```

Synthesis



Managing Interface 13
- 12

©2022 Advanced Micro Devices, Inc.

- **Block Level Protocol**

- An IO protocol added at the RTL block level
- Controls and indicates the operational status of the block

- **Block Operation Control**

- Controls when the RTL block starts execution (ap_start)
- Indicates if the RTL block is idle (ap_idle) or has completed (ap_done)

- **Complete and function return**

- The ap_done signal also indicates when any function return is valid

- **Ready (not shown here)**

- If the function is pipelined an additional ready signal (ap_ready) is added
- Indicates a new sample can be supplied before done is asserted

AMD
XILINX

Port-Level IO Protocols

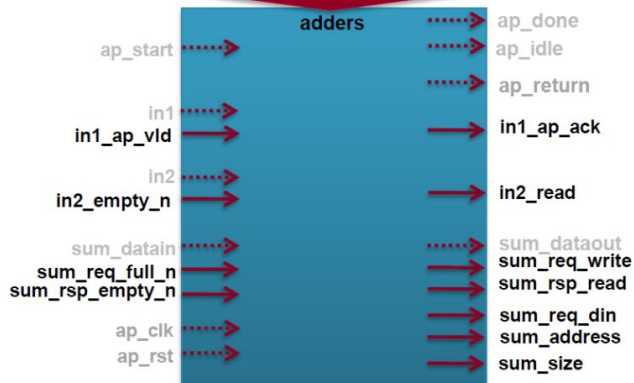
► Adder Example

```
#include "adders.h"
int adders(int in1, int in2, int *sum) {

    int temp;
    *sum = in1 + in2 + *sum;
    temp = in1 + in2;

    return temp;
}
```

Synthesis



• Port IO Protocols

- An IO protocol added at the port level
- Sequences the data to/from the data port

• Interface Synthesis

- The design is automatically synthesized to account for IO signals (enables, acknowledges etc.)

• Select from a pre-defined list

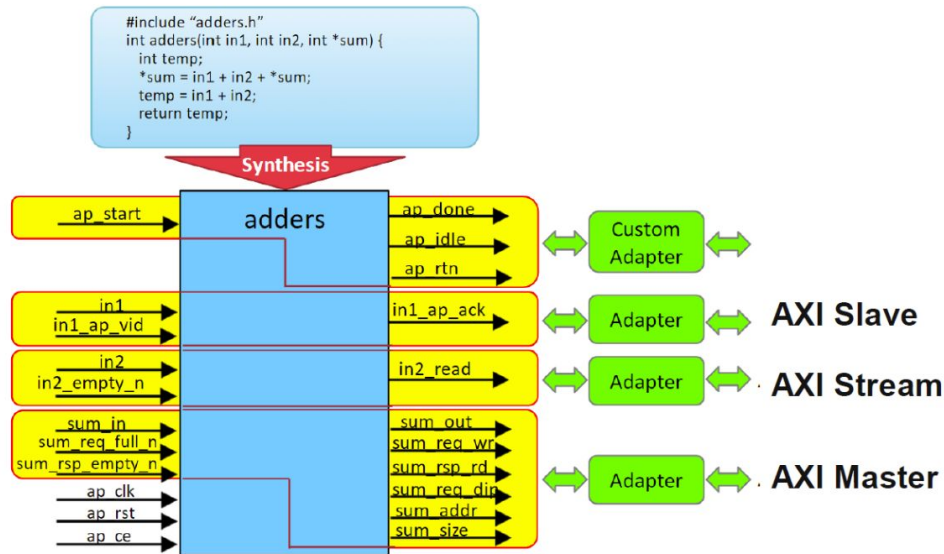
- The IO protocol for each port can be selected from a list
- Allows the user to easily connect to surrounding blocks

• Non-standard Interfaces

- Supported in C++ using an arbitrary protocol definition

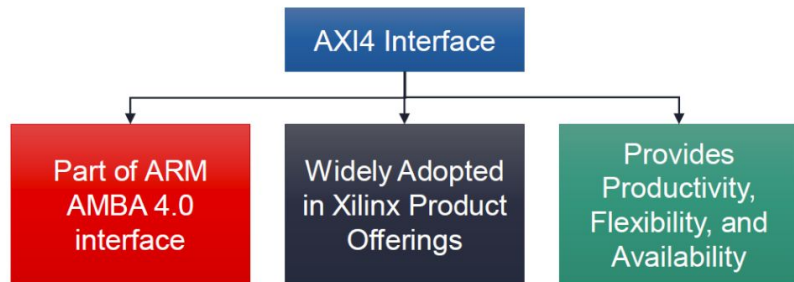
Vitis HLS IO Optional: AXI Adapter Protocols

- ▶ Bus interface are used for the both Vivado IP flow and Vitis kernel flow
- ▶ Bus interface include:
 - AXI4 Slave
 - AXI4 Stream
 - AXI4 Master
- ▶ IP adapter cores are used to connect the I/O protocols to bus interfaces
- ▶ These Interface are implemented in the RTL wrapper used in the IP generation flow



AXI Adapter Interface Protocols

- ▶ The AXI4 interfaces supported by Vitis HLS include
 - the AXI4-Stream interface (axis)
 - AXI4-Lite (s_axilite)
 - AXI4 master (m_axi) interfaces
- ▶ The AXI4 adapter interfaces are the default interfaces
 - Used by Vitis HLS for the Vitis kernel flow
 - Though they are also supported in the Vivado IP flow.

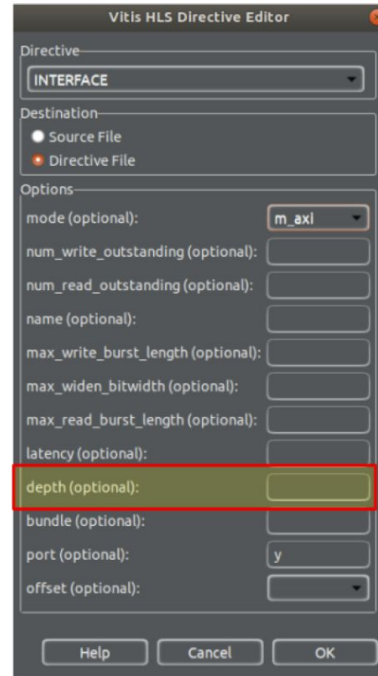


Argument	Scalar		Array			Pointer or Reference		
	pass-by-value		pass-by-reference			pass-by-reference		
Interface Mode	Input	Returns	I	IO	O	I	IO	O
ap_ctrl_none								
ap_ctrl_hs		D						
ap_ctrl_chain								
axis								
s_axilite								
m_axi								
ap_none	D					D		
ap_stable								
ap_ack								
ap_vld								D
ap_ovld							D	
ap_hs								D
ap_memory			D	D	D			
bram								
ap_fifo								

Supported. D = Default Interface Not Supported

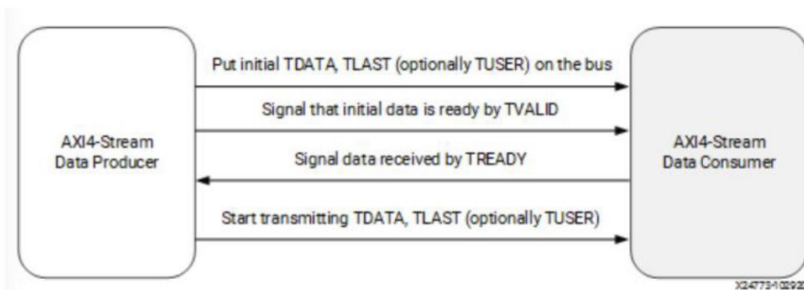
AXI4 Master Interface

- ▶ AXI4 memory-mapped (m_axi) interface is used on array or pointer/reference arguments
 - allow kernels to read and write data in global memory (DDR, HBM, PLRAM).
 - Default to pointer and array arguments only in Vitis kernel flow
- ▶ Vitis HLS implements in one of the following modes:
 - Individual data transfers: reads or writes a single element of data for each address
 - Burst mode data transfers: reads or writes data using a single base address followed by multiple sequential data samples, which makes this mode capable of higher data throughput
- ▶ HLS provides an option to group the multiple ports into the same AXI_Master Interface:
 - All the ports which use the same bundle name are grouped
 - Depth option is given for the C/RTL co-simulation, which is required for pointers



AXI4-Stream Interface

- ▶ AXIS interface can be applied to any input argument and any array or pointer output argument and cannot be used with arguments which are both read and write (inout arguments)
- ▶ AXIS interface is always sign-extended to the next byte
- ▶ Implemented are register interfaces or non-registered interfaces. This is controlled using the INTERFACE directive –register option
- ▶ The use of `hls::axis` (and `ap_axiu/ap_axis`)
 - limited to interfaces of the top-level function as it is the programmatic method to support AXI4-Stream with side channels.
 - cannot be used on internal functions or variables.
 - For internal streams you must use **`hls::stream`** objects.



- ▶ AXI4-Stream is a protocol designed for transporting arbitrary unidirectional data.
 - TDATA width of bits is transferred per clock cycle.
 - The transfer is started once the producer sends the TVALID signal and the consumer responds by sending the TREADY signal (once it has consumed the initial TDATA).
 - At this point, the producer will start sending TDATA and TLAST (TUSER if needed to carry additional user-defined sideband data).
 - TLAST signals the last byte of the stream. So the consumer keeps consuming the incoming TDATA until TLAST is asserted.



Improving Performance

Improving Performance

- ▶ Vitis HLS has many ways to improve performance
 - Automatic (and default) optimizations
 - Latency directives
 - Pipelining to allow concurrent operations
- ▶ Vitis HLS support techniques to remove performance bottlenecks
 - Manipulating loops
 - Partitioning and reshaping arrays
- ▶ Optimizations are performed using directives
 - Let's look first at how to apply and use directives in Vitis HLS

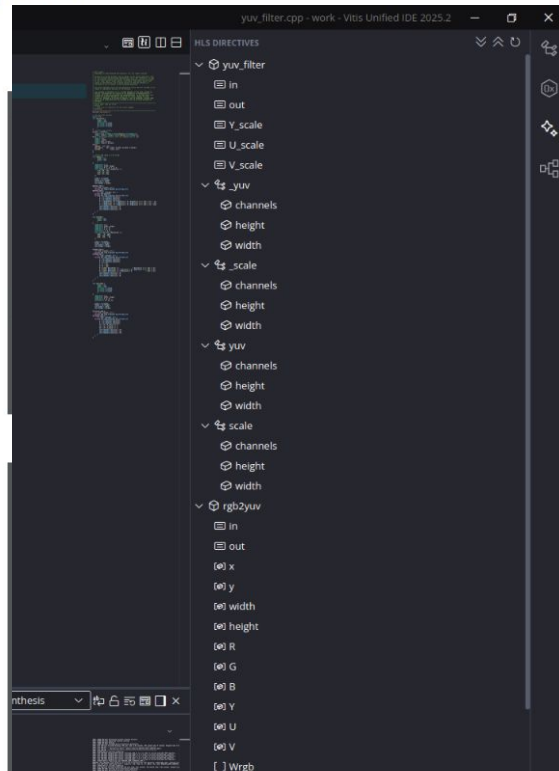
Optimization Directives: ~~For~~ or Pragma

- ▶ Directives can be placed into the C source by default
 - Pragas are added (and will remain) in the C source file
 - Pragas (#pragma) will be used by every solution which uses the code

```

74
75 RGB2YUV_LOOP_X:
76     for (x=0; x<width; x++) {
77         #pragma HLS loop_tripcount min=200 max=1920
78 RGB2YUV_LOOP_Y:
79         for (y=0; y<height; y++) {
80             #pragma HLS pipeline
81             #pragma HLS loop_tripcount min=200 max=1280
82                 R = in->channels.ch1[x][y];
83                 G = in->channels.ch2[x][y];
84                 B = in->channels.ch3[x][y];
85                 Y = ((Wrgb[0][0] * R + Wrgb[0][1] * G + Wrgb[0][2] * B + 128) >> 8) + 16;
86                 U = ((Wrgb[1][0] * R + Wrgb[1][1] * G + Wrgb[1][2] * B + 128) >> 8) + 128;
87                 V = ((Wrgb[2][0] * R + Wrgb[2][1] * G + Wrgb[2][2] * B + 128) >> 8) + 128;
88                 out->channels.ch1[x][y] = Y;
89                 out->channels.ch2[x][y] = U;
90                 out->channels.ch3[x][y] = V;
91             }
92         }
93     }

```



Improving Performance 21- 7

©2022 Advanced Micro Devices, Inc.

AMD
XILINX

Optimization Directives: ~~For~~ or Pragma

- ▶ Directives can be placed into the C source by default
 - Pragas are added (and will remain) in the C source file
 - Pragas (`#pragma`) will be used by every solution which uses the code

```

74
75 RGB2YUV_LOOP_X:
76     for (x=0; x<width; x++) {
77         #pragma HLS loop_tripcount min=200 max=1920
78     RGB2YUV_LOOP_Y:
79         for (y=0; y<height; y++) {
80             #pragma HLS pipeline
81             #pragma HLS loop_tripcount min=200 max=1280
82             R = in->channels.ch1[x][y];
83             G = in->channels.ch2[x][y];
84             B = in->channels.ch3[x][y];
85             Y = ((Wrgb[0][0] * R + Wrgb[0][1] * G + Wrgb[0][2] * B + 128) >> 8) + 16;
86             U = ((Wrgb[1][0] * R + Wrgb[1][1] * G + Wrgb[1][2] * B + 128) >> 8) + 128;
87             V = ((Wrgb[2][0] * R + Wrgb[2][1] * G + Wrgb[2][2] * B + 128) >> 8) + 128;
88             out->channels.ch1[x][y] = Y;
89             out->channels.ch2[x][y] = U;
90             out->channels.ch3[x][y] = V;
91         }
92     }
93 }

```

We just add them directly in the editor :)



Improving Throughput

Pipeline & dataflow

Dataflow vs Pipelining Optimization

▶ Dataflow Optimization

- Dataflow optimization is “coarse grain” pipelining at the function and loop level
- Increases concurrency between functions and loops
- Only works on functions or loops at the top-level of the hierarchy
 - Cannot be used in sub-functions

▶ Function & Loop Pipelining

- “Fine grain” pipelining at the level of the operators (*, +, >>, etc.)
- Allows the operations inside the function or loop to operate in parallel
- Unrolls all sub-loops inside the function or loop being pipelined
 - Loops with variable bounds cannot be unrolled: This can prevent pipelining
 - Unrolling loops increases the number of operations and can increase memory and run time

Pipeline Directive

- ▶ The pipeline directive pipelines functions or loops
 - This example pipelines the function with an Initiation Interval (II) of 2
 - The II is the same as the throughput but this term is used exclusively with pipelines



- ▶ Omit the target II and Vitis HLS will Automatically pipeline for the fastest possible design
 - Specifying a more accurate maximum may allow more sharing (smaller area)
- ▶ The directive on loops provides loop rewinding option

Directive

PIPELINE

Destination

☐ Source File

☒ Directive File

Options

II (optional): 2

off (optional): ☐

rewind (optional): ☐

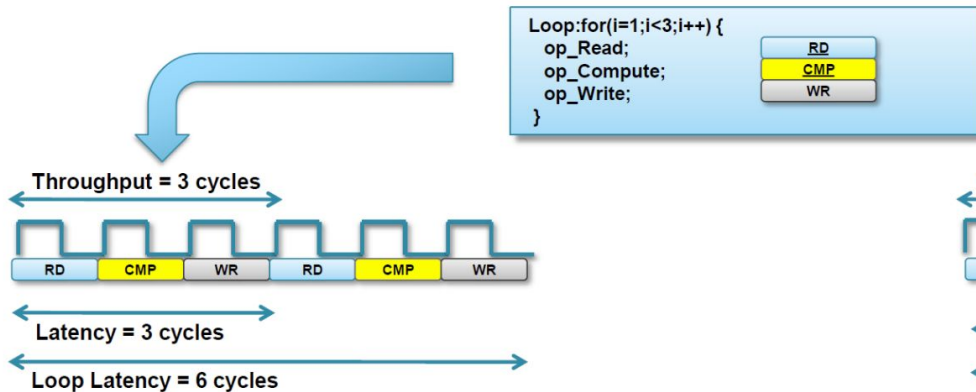
style (optional):

Pipeline for Performance

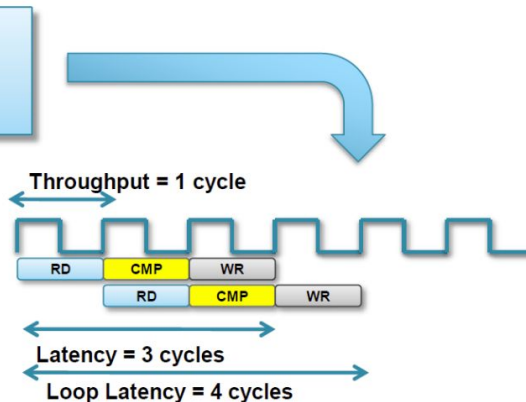
- ▶ When the PIPELINE directive is used, it **automatically unrolls** all loops and sub-loops in the hierarchy below, creating a great deal of logic
- ▶ Loops with **variable bounds** cannot be unrolled, and any loops and functions in the hierarchy above these loops cannot be pipelined
- ▶ Pipeline of the functions and loops can **maximize** performance
- ▶ Vitis HLS **auto-pipelines the loops by default** to improve performance
- ▶ If the sub-function is not pipelined, the function above it might show limited improvement when it is pipelined.

Loop Pipelining

Without Pipelining



With Pipelining



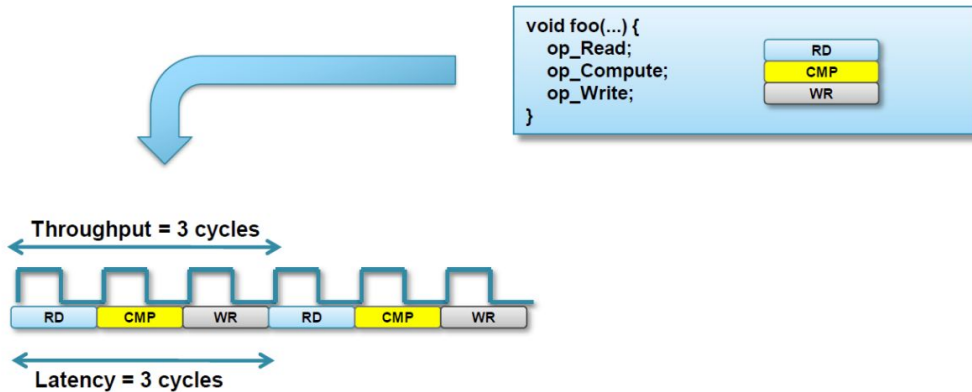
- ▶ There are 3 clock cycles before operation RD can occur again
 - Throughput = 3 cycles
- ▶ There are 3 cycles before the 1st output is written
 - Latency = 3 cycles
 - For the loop, 6 cycles

- > The latency is the same
 - >> The throughput is better
 - Less cycles, higher throughput
- > The latency for all iterations, the loop latency, has been improved

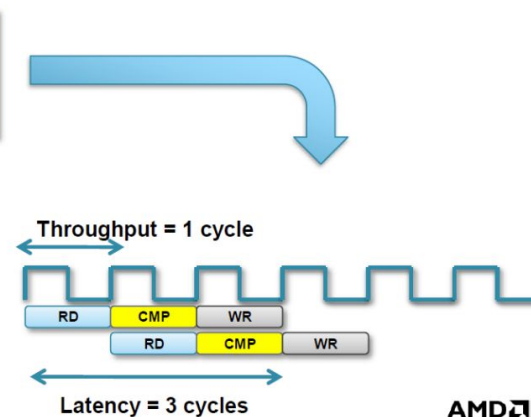
Function Pipelining

- ▶ There are 3 clock cycles before operation RD can occur again
 - Throughput = 3 cycles
- ▶ There are 3 cycles before the 1st output is written
 - Latency = 3 cycles
- ▶ The latency is the same
- ▶ The throughput is better
 - Less cycles, higher throughput

Without Pipelining



With Pipelining



Pipelining: Be Careful Where You Put it

How the optimization directives are applied determines the output

- ▶ Vitis HLS will attempt to unroll all loops nested below a PIPELINE directive
 - May not succeed for various reason and/or may lead to unacceptable area
 - Loops with variable bounds cannot be unrolled
 - Unrolling Multi-level loop nests may create a lot of hardware
 - Pipelining the inner-most loop will result in best performance for area
 - If the inner most loop is modest and fixed, try the next one(or two) out
 - Outer loops will keep the inner pipeline fed

```
void foo(in1[ ][ ], in2[ ][ ], ...) {
  ...
  L1:for(i=1;i<N;i++){
    L2:for(j=0;j<M;j++){
      #pragma HLS PIPELINE
      out[i][j] = in1[i][j] + in2[i][j];
    }
  }
}
```

1adder, 3 accesses

```
void foo(in1[ ][ ], in2[ ][ ], ...) {
  ...
  L1:for(i=1;i<N;i++){
    #pragma HLS PIPELINE
    L2:for(j=0;j<M;j++){
      out[i][j] = in1[i][j] + in2[i][j];
    }
  }
}
```

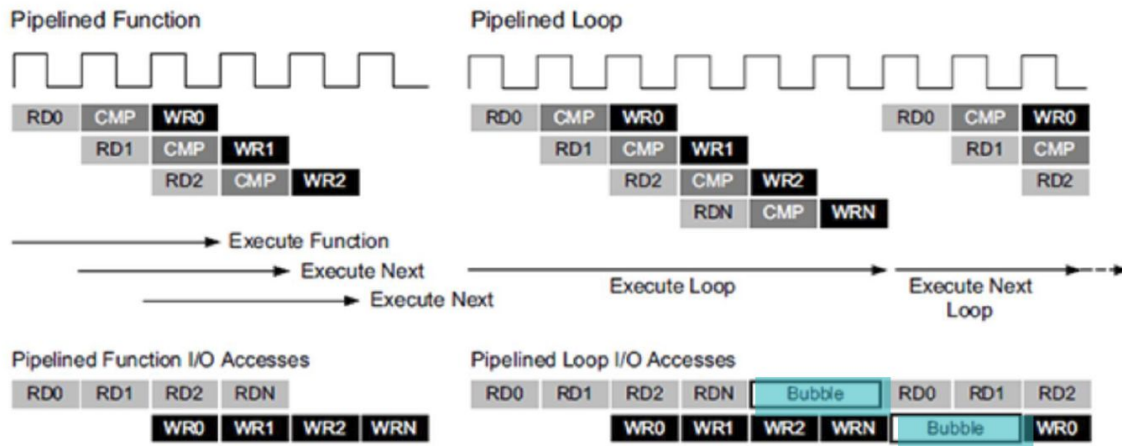
Unrolls L2
M adders, 3M accesses

```
void foo(in1[ ][ ], in2[ ][ ], ...) {
  #pragma HLS PIPELINE
  ...
  L1:for(i=1;i<N;i++){
    L2:for(j=0;j<M;j++){
      out[i][j] = in1[i][j] + in2[i][j];
    }
  }
}
```

Unrolls L1 and L2
N*M adders, 3(N*M) accesses

Pipelined Functions and Loops

The difference is how the inputs and outputs of the pipeline are processed

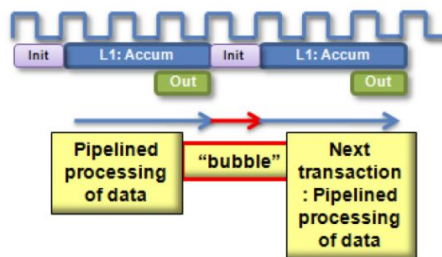


- ▶ In case of functions, the pipeline runs forever and never ends
- ▶ In case of loops, the pipeline executes until all iterations of the loop are completed
- ▶ Bubble is nothing but
 - Point when no new inputs are read as the loop completes the execution of the final iterations
 - Point when no new outputs are written as the loop starts new loop iterations

Pipelining the Top-Level Loop

- ▶ Loop Pipelining top-level loop may give a “bubble”
 - A “bubble” here is an interruption to the data stream
 - Given the following

```
void foo_top (in1, in2, ...) {  
    static accum=0;  
    ...  
    L1:for(i=1;i<N;i++) {  
        accum = accum + in1 + in2;  
    }  
    out1_data = accum;  
    ...  
}
```

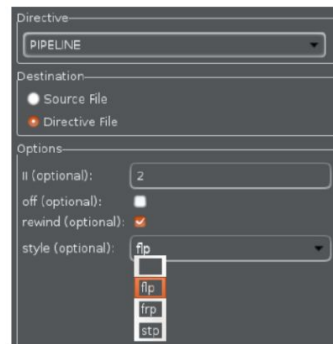
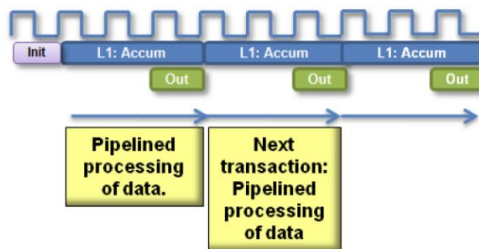


- The function will process a stream of data
- The next time the function is called, it still needs to execute the initial (init) operations
 - These operations are any which occur before the loop starts
 - These operations may include interface start/stop/done signals
- Due to these operations, an unexpected interruption of the data stream, a bubble can happen

Continuous Pipelining the Top-Level loop

- ▶ Use the “rewind” option for continuous pipelining
 - Immediate re-execution of the top-level loop
 - The operation rewinds to the start of the loop

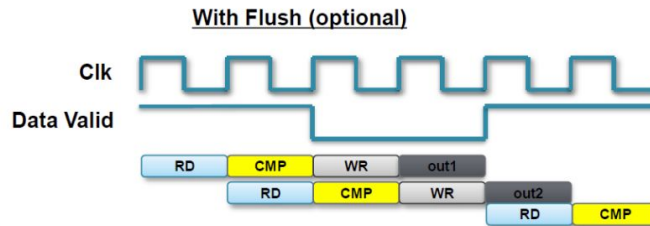
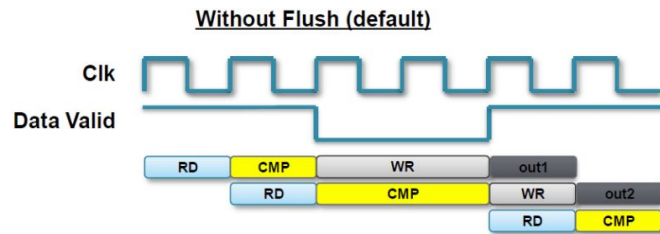
```
void foo_top (in1, in2, ...) {
    static accum=0;
    ...
    L1:for(i=1;i<N;i++) {
        accum = accum + in1 + in2;
    }
    out1_data = accum;
    ...
}
```



- ▶ When used top-level loops, at the end of the loop iteration count, loop starts to re-execute immediately
 - Only affects the top-level loops
- ▶ When used on the function, it rewinds the function execution to the start of the loop by pushing any initialization statements before the start of the loop into the loop
 - These statements cannot contain if-else branches

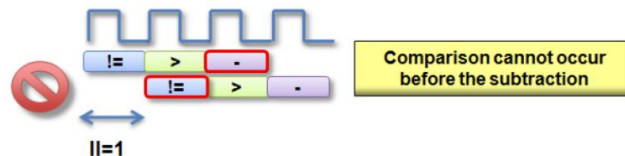
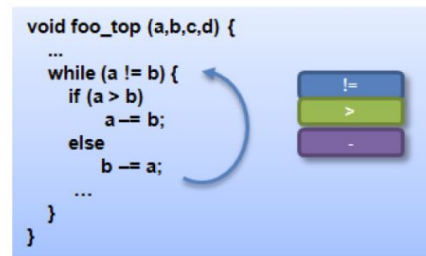
Pipeline Flush

- ▶ Pipelines continue to execute as long as data is available at the input
 - If there is no data available to process, the pipeline will stall.
- ▶ Without Flush:
 - Operation that is being performed is stalled when the valid signal goes Low
 - Write operation is stalled here
- ▶ With Flush:
 - A pipeline can be emptied or flushed using the flush option
 - When a pipeline is flushed, the pipeline stops reading new inputs when none are available as determined by a data valid signal at the start of the pipeline
 - It continues processing, shutting down each successive pipeline stage until the final input has been processed through to the output of the pipeline



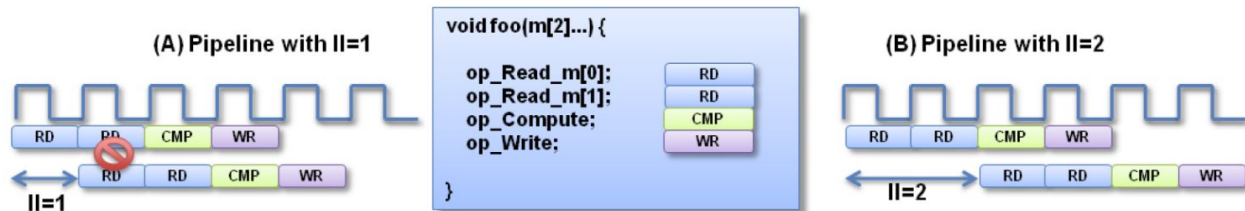
Issues That Prevent Pipelining

- ▶ Pipelining functions unrolls all loops
 - Loops with variable bounds cannot be unrolled
 - This will prevent pipelining
 - Re-code to remove the variables bounds: max bounds with an exit
- ▶ Feedback prevent/limits pipelines
 - Feedback within the code will prevent or limit pipelining
 - The pipeline may be limited to higher initiation interval (more cycles, lower throughput)
- ▶ Resource Contention may prevent pipelining
 - Can occur within input and output ports/arguments
 - This is a classis way in which arrays limit performance



Resource Contention: Unfeasible Initiation Intervals

- Sometimes the II specification cannot be met
 - In this example there are 2 read operations on the same port

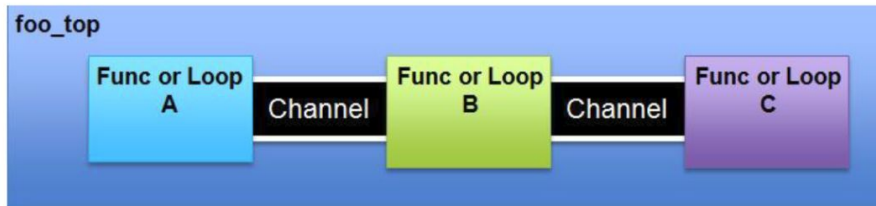


- An II=1 cannot be implemented
 - The same port cannot be read at the same time
 - Similar effect with other resource limitations
 - For example, if functions or multipliers etc. are limited
- Vitis HLS will automatically increase the II to 2 as the tool will always try to create a design, even if constraints must be violated

Dataflow Optimization

► Dataflow Optimization

- Creates a parallel process architecture
- Useful on a set of sequential tasks as it creates an architecture of concurrent processes for these tasks
- Allows the execution of the tasks to overlap
- Improves performance over a statically pipelined solution
- Increases the overall throughput of the design and reduces latency
- Replaces the strict, centrally-controlled pipeline with a more flexible and distributed handshaking architecture
- DATAFLOW directive works on function/loop level – allowing the parallel execution of multiple loops or functions



Dataflow for Performance

- ▶ Given a design with multiple functions
 - The code and dataflow are as shown

```
void foo_top (a,b,c,d, *x, *y) {
    ...
    func_A(a,b,t1);
    func_B(a,t1,t2);
    func_C(c,t2,&x);
    func_D(d,x,&y);
}
```



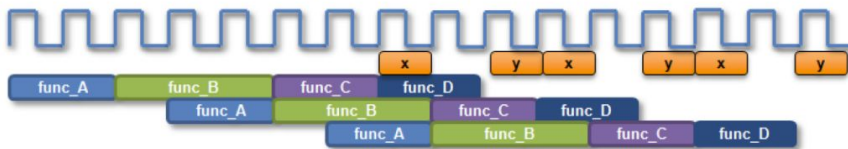
- ▶ Implementation requires nine cycles before a new input can be processed by func_A



The latency is 9 cycles

The throughput is also 9 cycles

- ▶ Func_A can begin processing a new input every three clock cycles



The latency is still 9 cycles

The throughput is now 3

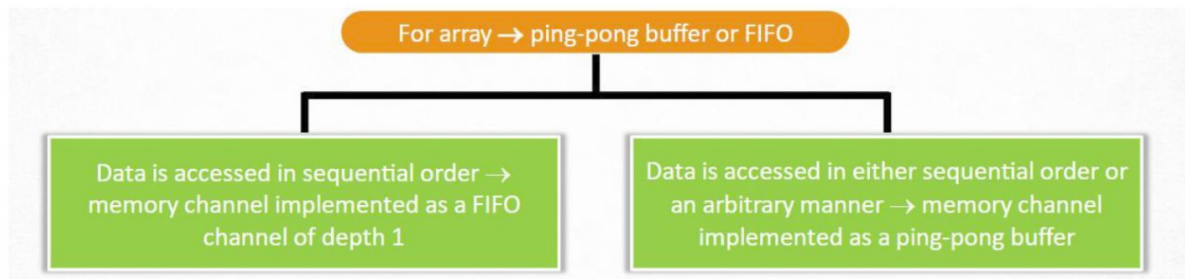
Configuring the Dataflow Channel

- ▶ Vitis HLS tool analyzes the functions or a loop body to create individual channels that model the dataflow to store the results of each task in the dataflow region.
- ▶ These channels can be either
 - Simple FIFOs with handshake signals for scalar variables
 - Ping-pong buffers with memcopy elements, for non-scalar variables like arrays
- ▶ Channels contain handshake signals to indicate when the FIFO or ping-pong buffer is full or empty
- ▶ Dataflow optimization has an area overhead
- ▶ Individual buffers allow Vitis HLS tool to free each task to execute at its own pace

To use FIFO's the access must be sequential. If HLS determines that the access is not sequential then it will halt and issue a message. If HLS can not determine the sequential nature then it will issue warning and continue.

FIFO vs Ping-Pong Buffer

- ▶ For scalar, pointer, reference parameters, function return, Vitis HLS tool implements channel -> FIFO
- ▶ Ping-Pong buffer implementation is nothing but two block RAMs, each defined by the maximum size of the consumer or producer array
- ▶ A ping-pong buffer ensures that the channel always has the capacity to hold all samples without a loss, providing the improved interval and safe and reliable data transfer.



Dataflow Optimization Limitations

- ▶ Optimizes the flow of data between tasks for maximum performance
- ▶ Does not require tasks to be chained one after the other; however, there are some limitations in how the data is transferred
- ▶ Here are few coding styles that prevent dataflow optimization:
 - Single producer-consumer violations
 - Bypassing tasks
 - Feedback between tasks
 - Conditional execution of tasks
 - Loops with multiple exit conditions

Single producer-consumer violations

- ▶ Must be single producer consumer; the following code violates the rule and dataflow does not work

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {

    int temp1[N];
    Loop1: for(int i = 0; i < N; i++) {
        temp1[i] = data_in[i] * scale;
    }
    Loop2: for(int j = 0; j < N; j++) {
        data_out1[j] = temp1[j] * 123;
    }
    Loop3: for(int k = 0; k < N; k++) {
        data_out2[j] = temp1[k] * 456;
    }
}
```

The Fix

```
void Split (in[N], out1[N], out2[N]) {
    // Duplicated data
    L1:for(int i=1;i<N;i++) {
        out1[i] = in[i];
        out2[i] = in[i];
    }
}

void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {

    int temp1[N], temp2[N], temp3[N];
    Loop1: for(int i = 0; i < N; i++) {
        temp1[i] = data_in[i] * scale;
    }
    Split(temp1, temp2, temp3);
    Loop2: for(int j = 0; j < N; j++) {
        data_out1[j] = temp2[j] * 123;
    }
    Loop3: for(int k = 0; k < N; k++) {
        data_out2[j] = temp3[k] * 456;
    }
}
```


Bypassing Tasks

- ▶ You cannot bypass a task; the following code violates this rule and dataflow does not work

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {
    int temp1[N], temp2[N], temp3[N];
    Loop1: for(int i = 0; i < N; i++) {
        temp1[i] = data_in[i] * scale;
        temp2[i] = data_in[i] >> scale;
    }
    Loop2: for(int j = 0; j < N; j++) {
        temp3[j] = temp1[j] + 123;
    }
    Loop3: for(int k = 0; k < N; k++) {
        data_out[j] = temp2[k] + temp3[k];
    }
}
```

The fix: make it systolic like datapath

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {
    int temp1[N], temp2[N], temp3[N], temp4[N];
    Loop1: for(int i = 0; i < N; i++) {
        temp1[i] = data_in[i] * scale;
        temp2[i] = data_in[i] >> scale;
    }
    Loop2: for(int j = 0; j < N; j++) {
        temp3[j] = temp1[j] + 123;
        temp4[j] = temp2[j];
    }
    Loop3: for(int k = 0; k < N; k++) {
        data_out[j] = temp4[k] + temp3[k];
    }
}
```


Optimize for throughput

Performance Bottleneck

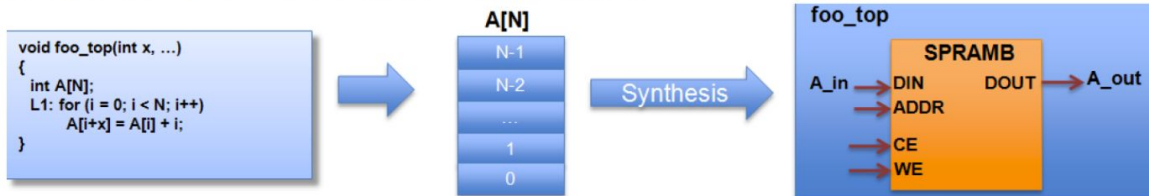
Optimize for throughput

- ▶ When pipelining fails to meet the required performance
- ▶ Examine the design in the analysis perspective
- ▶ Issues that might be encountered while pipelining the functions and loops
 - Addressed using optimization directives and configurations
 - Reduce bottlenecks in data structures

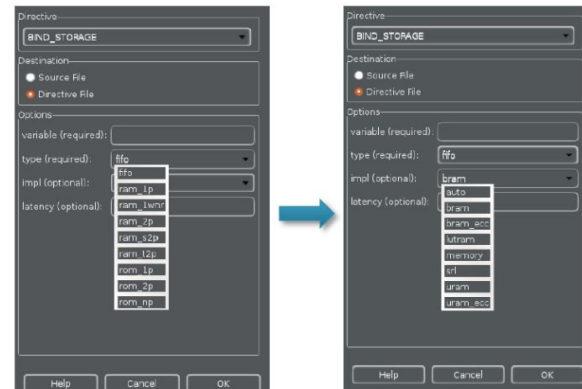
Directives and Configurations	Description
CONFIG ARRAY PARTITION	Determines how arrays are partitioned, including global arrays and if the partitioning impacts array ports
CONFIG COMPILE	Controls synthesis-specific optimizations such as the automatic loop pipelining and floating point math optimizations
CONFIG SCHEDULE	Determines the effort level to use during the synthesis scheduling phase and the verbosity of the output messages
CONFIG_UNROLL	Allows all loops below the specified number of loop iterations to be automatically unrolled

Review: Arrays in HLS

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

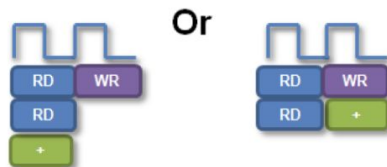


- ▶ To use a FIFO instead of a block RAM -> Stream directive
- ▶ Arrays are automatically specified as streaming:
 - If an array is set as interface type `ap_fifo`, `axis`, or `ap_hs`, etc
 - If the array are used in a region where the DATAFLOW is applied.
- ▶ All the other arrays must be specified as streaming using the `STREAM` directive if a FIFO is required for the implementation



Arrays : Performance bottlenecks

- ▶ Arrays are intuitive and useful software constructs
 - They allow the C algorithm to be easily captured and understood
- ▶ Array accesses can often be performance bottlenecks
 - Arrays are targeted to a default RAM
 - Single-port RAM: impossible to pipeline for loop to process a new loop iteration every clock cycle
 - Dual-port RAM: Allows only two accesses per clock cycle
 - Three reads are required to calculate the value of the sum, and so three accesses per clock cycle are required to pipeline the loop with a new iteration every clock cycle.



Even with a dual-port RAM, we cannot perform all reads and writes in one cycle

- ▶ Solution:
 - Array can be partitioned and reshaped to produce higher data bandwidth
 - Allows more optimal configuration of the array
 - Provides better implementation of the memory resource

Array and RAM selection

- ▶ If no directive is selected
 - Vitis HLS will determine the RAM to use
 - It will use a Dual-port if it improves throughput
 - Else it will use a single-port
- ▶ If no RAM target is specified ,
 - RTL synthesis will determine if RAM is implemented as BRAM or LUTRAM
- ▶ If RAM target is specified
 - Vitis HLS tool will obey the target selected

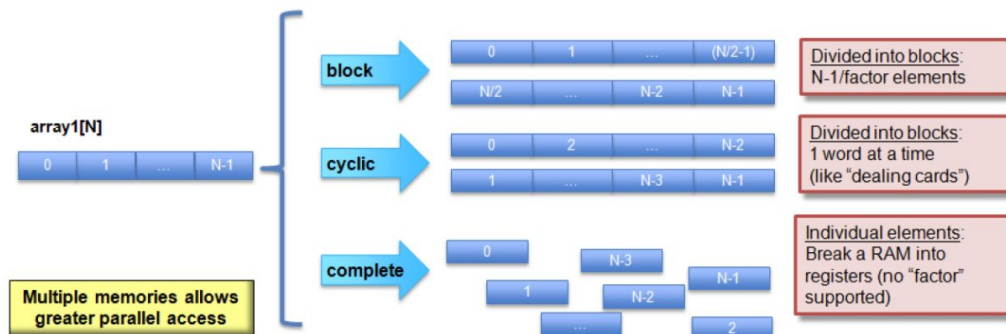
Specify BIND_STORAGE directive

Which type of RAM is used

Which RAM ports are created:
a single port or a dual port

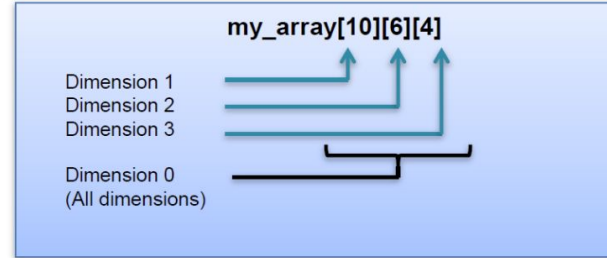
Array Partitioning

- ▶ Partitioning breaks an array into smaller arrays or individual registers to improve parallel access to data and remove block RAM bottlenecks
 - Types of array partition:
 - Block: Original array is split into equally sized blocks of consecutive elements of the original array
 - Complete: Default operation is to split the array into its elements. This corresponds to resolving a memory into registers
 - Cyclic: Original array is split into equally sized blocks, interleaving the elements of the original array



Array Dimensions

- ▶ The array options can be performed on dimensions of the array



Examples

my_array[10][6][4] → partition dimension 3 →

```

my_array_0[10][6]
my_array_1[10][6]
my_array_2[10][6]
my_array_3[10][6]
  
```

my_array[10][6][4] → partition dimension 1 →

my_array[10][6][4] → partition dimension 0 → 10x6x4 = 240 individual registers

```

my_array_0[6][4]
my_array_1[6][4]
my_array_2[6][4]
my_array_3[6][4]
my_array_4[6][4]
my_array_5[6][4]
my_array_6[6][4]
my_array_7[6][4]
my_array_8[6][4]
my_array_9[6][4]
  
```

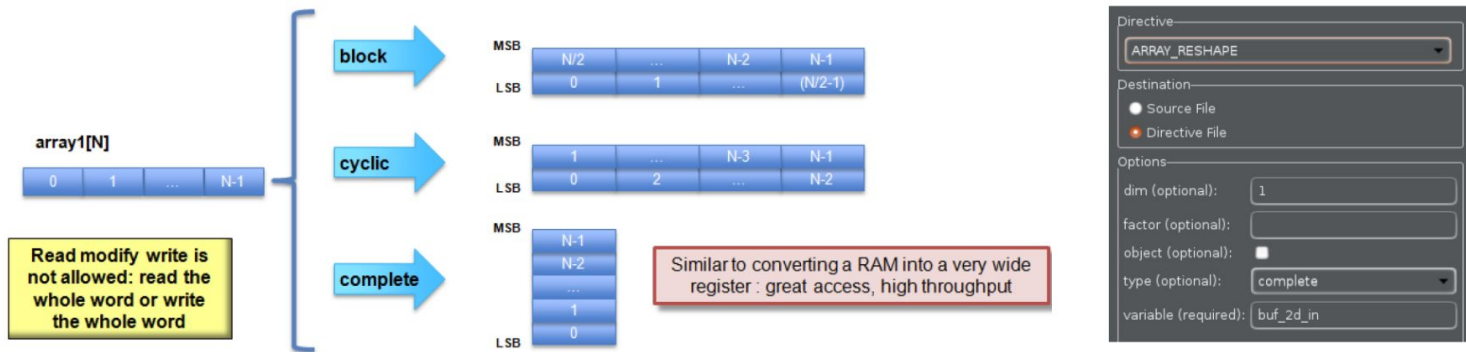

Configuring Array Partitioning

- ▶ Vitis HLS can automatically partition arrays to improve throughput
 - This is controlled via the array configuration command
 - Enable mode `throughput_driven`
- ▶ Auto-partition arrays with constant indexing
 - When the array index is not a variable
 - Arrays below the threshold are auto-partitioned
 - Set the threshold using option `complete_threshold`

Name	Value	Default	Reset
▼ config_array_partition			
complete_threshold	4	4	
throughput_driven	auto	auto	

Array Reshaping

- ARRAY_RESHAPE directive allows more data to be accessed in a single clock cycle
- Reduces the number of block RAMs while still allowing the beneficial attributes of partitioning — parallel access to the data
- Vitis HLS tool may automatically unroll any loops consuming this data and will try to improve the throughput
- Reshaping recombines partitioned arrays back into a single array



Improving Performance 21- 36

©2022 Advanced Micro Devices, Inc.

AMD
XILINX

Reshaping vs Partitioning

- ▶ Both are useful for increasing the memory or data bandwidth
- ▶ Reshaping
 - Simply increases the width of the data word
 - Does not increase the number of memory ports
- ▶ Partitioning
 - Increases the memory ports; thus more I/O to deal with
 - Use it only if you have to use independent addressing
- ▶ Common error message: cue to use reshaping or partitioning

INFO: [SCHED 61] Pipelining loop 'SUM_LOOP'.

WARNING: [SCHED 69] Unable to schedule 'load' operation ('mem_load_2', bottleneck.c:62) on array 'mem' due to limited memory ports.

INFO: [SCHED 61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.

Bottleneck Example

- ▶ Array accesses (block RAM) can be bottlenecks inside functions or loops
 - Still prevents the II of 1 despite PIPELINE and DATAFLOW
 - Prevents processing of one sample per clock

```
@I [SCHED-61] Pipelining loop 'SOBEL_LOOP'.
@W [SCHED-69] Unable to schedule 'store' operation (image_demo.cpp:172) of
variable 'y' on array 'buff_A' due to limited resources (II = 2).
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 3, Depth: 13.
```

```
void sobel_filter(RGB inter_pix[MAX_HEIGHT][MAX_WIDTH],
                 AXI_PIXEL out_pix[MAX_HEIGHT][MAX_WIDTH],
                 int rows, int cols) {
    ...
    for (row = 0; row < rows + 1; row++) {
        SOBEL_LOOP: for (col = 0; col < cols + 1; col++)
        {
            if (col < cols) {
                buff_A[2][col] = buff_A[1][col];
                buff_A[1][col] = buff_A[0][col];
                buff_B[1][col] = buff_B[0][col];
                temp = buff_A[0][col];
            }
            if (col < cols & row < rows) {
                tempx = inter_pix[row][col];
                buff_A[0][col] = rgb2y(tempx);
                buff_B[0][col] = tempx;
            }
            ...
        }
    }
}
```

3 Writes

+

3 Reads

+

Dual-Port
RAMMaximum of
2 RAM ports

+

1 Sample Per Clock

=

Too Few RAM ports

Solution: Use ARRAY_PARTITION directive

```
void sobel_filter(RGB inter_pix[MAX_HEIGHT][MAX_WIDTH],
                 AXI_PIXEL out_pix[MAX_HEIGHT][MAX_WIDTH],
                 int rows, int cols) {
    ...
    #pragma HLS ARRAY_PARTITION variable=buff_A dim=1 complete
    for (row = 0; row < rows + 1; row++) {
        SOBEL_LOOP: for (col = 0; col < cols + 1; col++)
        {
            if (col < cols) {
                buff_A[2][col] = buff_A[1][col];
                buff_A[1][col] = buff_A[0][col];
                buff_B[1][col] = buff_B[0][col];
                temp = buff_A[0][col];
            }
            if (col < cols & row < rows) {
                tempx = inter_pix[row][col];
                buff_A[0][col] = rgb2y(tempx);
                buff_B[0][col] = tempx;
            }
            ...
        }
    }
}
```

3 Writes

+

3 Reads

+

3 Dual-Port
RAMsMaximum
Number of
RAM ports

+

1 Sample Per Clock

=

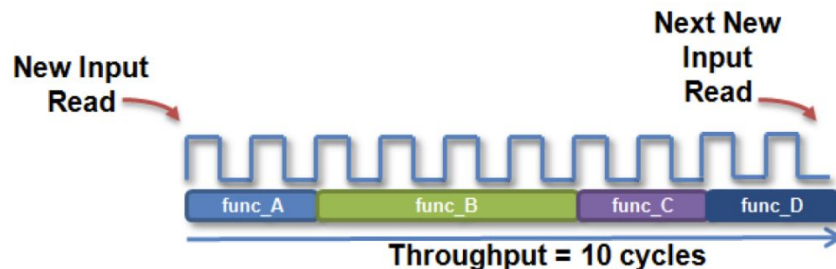
Success

Improving Latency

Latency and Throughput – The Performance Factors

► Design Throughput

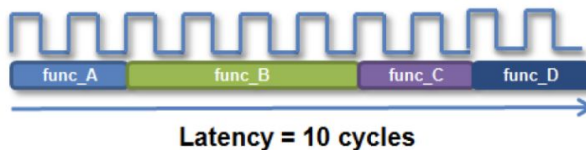
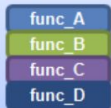
- The throughput of the design is the number of cycles between new inputs
 - By default (no concurrency) this is the same as latency
 - Next start/read is when this transaction ends



► Design Latency

- The latency of the design is the number of cycle it takes to output the result
 - In this example the latency is 10 cycle

```
void foo_top (a,b,c,d, *x, *y) {  
    ...  
    func_A(...);  
    func_B(...);  
    func_C(...);  
    func_D(...);  
    return res;  
}
```

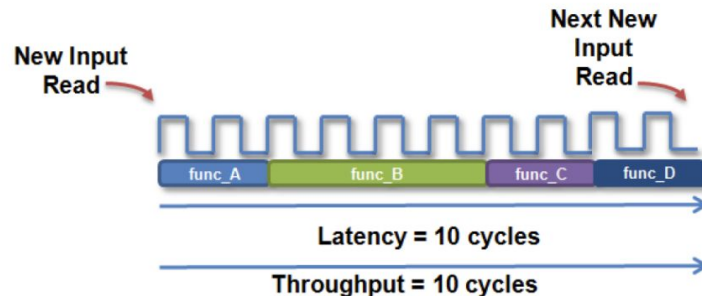


Latency and Throughput

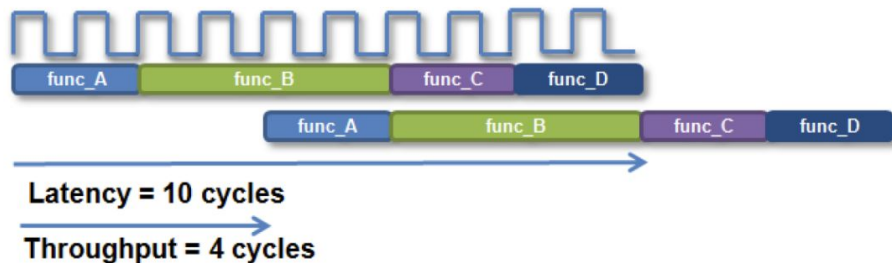
- ▶ In the absence of any concurrency
 - Latency is the same as throughput

```
void foo_top (a,b,c,d, *x, *y) {
    ...
    func_A(...);
    func_B(...);
    func_C(...);
    func_D(...);
    return res;
}
```

func_A
func_B
func_C
func_D



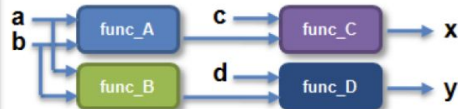
- ▶ Pipelining for higher throughput
 - Vitis HLS can pipeline functions and loops to improve throughput
 - Latency and throughput are related
 - We will discuss optimizing for latency first, then throughput



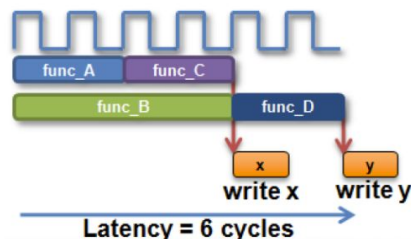
Vitis HLS: Minimize latency

- ▶ Vitis HLS will by default minimize latency
 - Throughput is prioritized above latency (no throughput directive is specified here)
 - In this example
 - The functions are connected as shown
 - Assume function B takes longer than any other functions

```
void foo_top (a,b,c,d, ^x, ^y) {
    ...
    func_A(a,b,t1);
    func_B(a,b,t2);
    func_C(c,t1,&x);
    func_D(d,t2,&y);
}
```



- ▶ Vitis HLS will automatically take advantage of the parallelism
 - It will schedule functions to start as soon as they can
 - Note it will not do this for loops within a function: by default they are executed in sequence



- func_A and func_B can start at the same time
- func_C can start as soon as func_A completes
- func_D must wait for func_B to complete
- Outputs are written as soon as they are ready

Reducing Latency

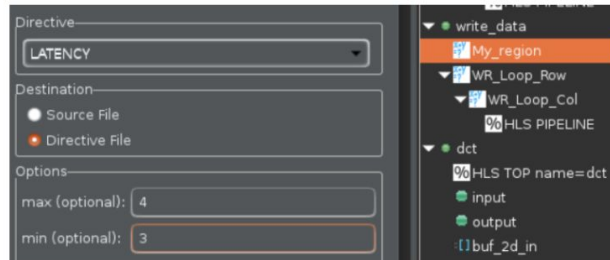
- ▶ Vitis HLS has the following directives to reduce latency
 - LATENCY
 - Allows a minimum and maximum latency constraint to be specified
 - LOOP_FLATTEN
 - Allows nested loops to be collapsed into a single loop with improved latency
 - LOOP_MERGE
 - Merge consecutive loops to reduce overall latency, increase sharing, and improve logic optimization
- ▶ Benefit to the latency because it typically costs a clock cycle to enter and leave a loop
- ▶ The fewer the transitions between loops, the less the number of clock cycles a design

Region Specific Latency Constraint

- ▶ Latency directives can be applied on functions, loops and regions
- ▶ Use regions to specify specific locations for latency constraints
 - A region is any set of named braces {...a region...}
 - The region My_Region is shown in this example
 - This allows the constraint to be applied to a specific range of code
 - Here, only the else branch has a latency constraint

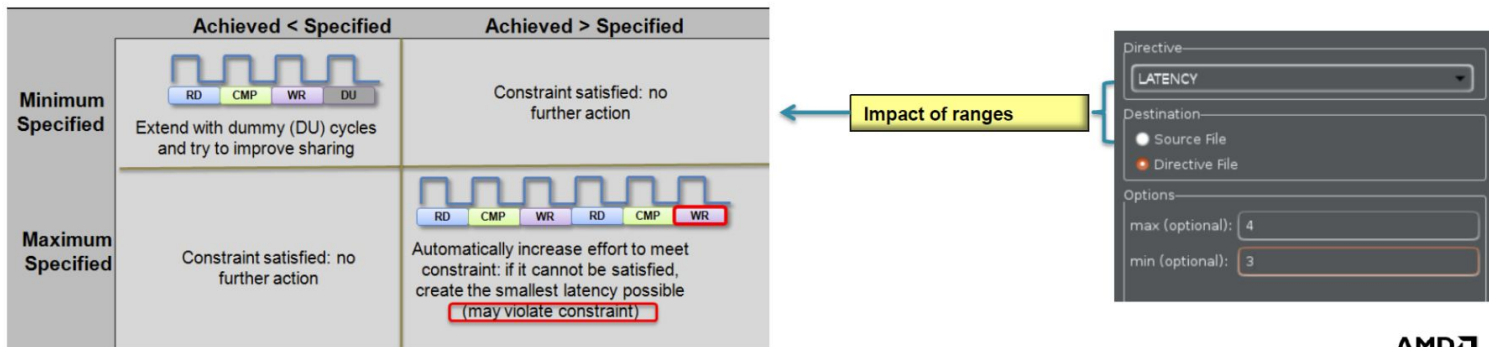
```
int write_data (int buf, int output) {  
    if (x < y) {  
        return (x + y);  
    } else {  
        My_Region: {  
            return (y - x) * (y + x);  
        }  
    }  
}
```

Select the region in the Directives tab & right-click to apply latency directive



Latency Constraints

- ▶ When a maximum and/or minimum LATENCY constraint is placed on a scope
 - Tool tries to ensure that all operations in the function complete within the range of clock cycles specified
 - If the loop is unrolled, the latency directive sets a maximum limit on all loop operations
- ▶ Latency specified VS latency achieved:
 - Max latency constraint cannot be met: Relax the latency constraint and tries to achieve the best possible result
 - Min latency constraint is set: Produce a design with lower latency and inserts dummy clock cycles to meet the minimum latency
 - No latency range is specified: Schedules for minimum latency



Improving Performance 21- 45

©2022 Advanced Micro Devices, Inc.

AMD
XILINX

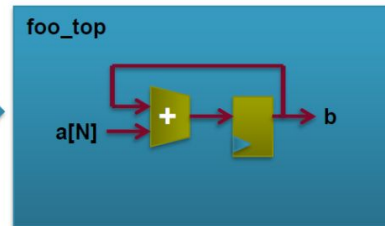
Review: Loops

- ▶ By default, loops are rolled
 - Each C loop iteration → Implemented in the same state
 - Each C loop iteration → Implemented with same resources
 - Loops can be unrolled if their indices are statically determinable at elaboration time
 - Not when the number of iterations is variable
- ▶ Optimizations by making change to the loop structure as if the code were changed
 - Unroll
 - Partially unroll
 - Flatten
 - Merge

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

Synthesis

Loops require labels if they are to be referenced
by Tcl directives
(GUI will auto-add labels)



Rolled Loops Enforce Latency

- ▶ A rolled loop can only be optimized so much
 - Given this example, where the delay of the adder is small compared to the clock frequency

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

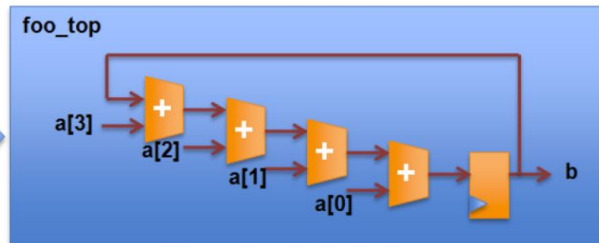


- This rolled loop will never take less than 4 cycles
 - No matter what kind of optimization is tried
 - This minimum latency is a function of the loop iteration count

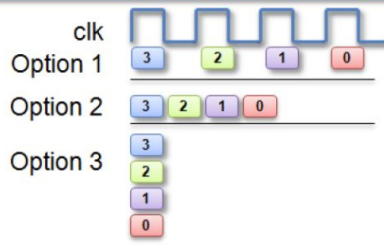
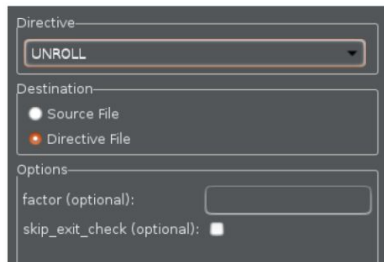
Unrolled Loops can Reduce Latency

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

Unrolled



Select loop “Add” in
the directives pane
and right-click



Unrolled loops allow
greater option &
exploration

Options explained on next
slide

Unrolled loops are likely to result in more hardware
resources and higher area

Partial Unrolling

- ▶ Fully unrolling loops can create a lot of hardware
- ▶ Loops can be partially unrolled
 - Provides the type of exploration shown in the previous slide
- ▶ Partial Unrolling
 - A standard loop of N iterations can be unrolled to by a factor
 - For example unroll by a factor 2, to have N/2 iterations
 - Similar to writing new code as shown on the right →
 - The break accounts for the condition when N/2 is not an integer
 - If “i” is known to be an integer multiple of N
 - The user can remove the exit check (and associated logic)
 - Vitis HLS is not always be able to determine this is true (e.g. if N is an input argument)
 - User takes responsibility: verify!

```
Add: for(int i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
}
```

```
Add: for(int i = 0; i < N; i += 2) {
    a[i] = b[i] + c[i];
    if (i+1 >= N) break;
    a[i+1] = b[i+1] + c[i+1];
}
```

Effective code after
compiler
transformation

```
for(int i = 0; i < N; i += 2) {
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1] + c[i+1];
}
```

An extra adder for
N/2 cycles trade-off

Loop Flattening

- ▶ Vitis HLS can automatically flatten nested loops
 - A faster approach than manually changing the code
- ▶ Flattening should be specified on the inner most loop
 - It will be flattened into the loop above
 - The “off” option can prevent loops in the hierarchy from being flattened

Directive:

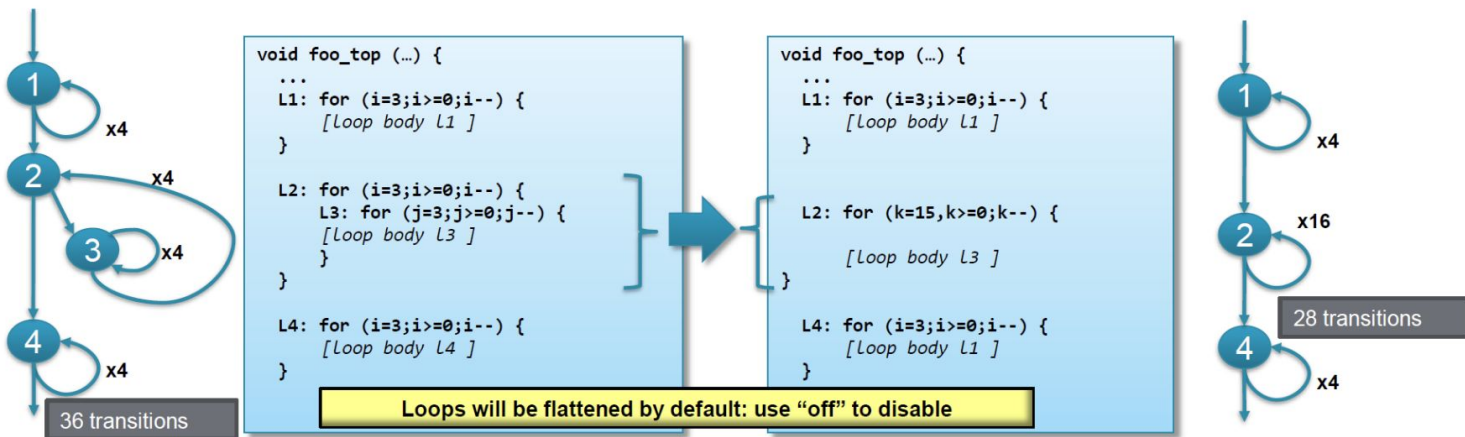
Destination:

☐ Source File

☒ Directive File

Options:

off (optional): ☐



Perfect and Semi-Perfect Loops

► Only perfect and semi-perfect loops can be flattened

- The loop should be labeled or directives cannot be applied

▪ Perfect Loops

- Only the inner most loop has body (contents)
- There is no logic specified between the loop statements
- The loop bounds are constant

▪ Semi-perfect Loops

- Only the inner most loop has body (contents)
- There is no logic specified between the loop statements
- The outer most loop bound can be variable

– Other types

- Should be converted to perfect or semi-perfect loops

```
Loop_outer: for (i=3;i>=0;i--) {
  Loop_inner: for (j=3;j>=0;j--) {
    [Loop body]
  }
}
```

```
Loop_outer: for (i=3;i>N;i--) {
  Loop_inner: for (j=3;j>=0;j--) {
    [Loop body]
  }
}
```

```
Loop_outer: for (i=3;i>N;i--) {
  [Loop body]
  Loop_inner: for (j=3;j>=M;j--) {
    [Loop body]
  }
}
```

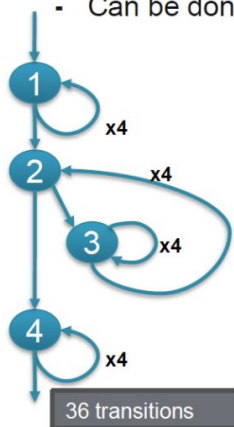
Loop Merging

- ▶ Vitis HLS can automatically merge loops
 - A faster approach than manually changing the code
 - Allows for more efficient architecture explorations
 - FIFO reads, which must occur in strict order, can prevent loop merging
 - Can be done with the “force” option : user takes responsibility for correctness

Directive
 LOOP_MERGE

Destination
☐ Source File
☒ Directive File

Options
 force (optional): ☐

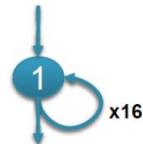


```
void foo_top (...) {
  ...
  L1: for (i=3;i>=0;i--) {
    [Loop body L1 ]
  }
  L2: for (i=3;i>=0;i--) {
    L3: for (j=3;j>=0;j--) {
      [Loop body L3 ]
    }
  }
  L4: for (i=3;i>=0;i--) {
    [Loop body L4 ]
  }
}
```

Already flattened



```
void foo_top (...) {
  ...
  L123: for (l=16,l>=0;l--) {
    if (cond1)
      [Loop body L1 ]
    [Loop body L3 ]
    if (cond4)
      [Loop body L4 ]
  }
}
```



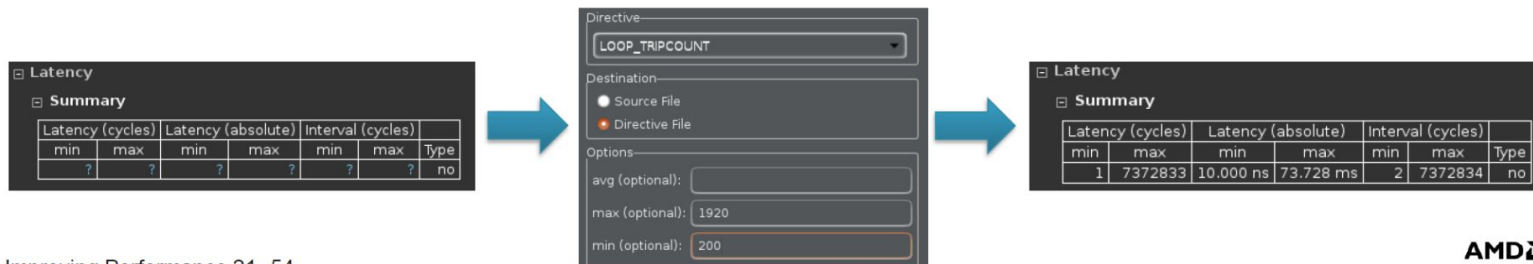
Loop Merge Rules

- ▶ If loop bounds are all variables, they must have the same value
- ▶ If loops bounds are constants, the maximum constant value is used as the bound of the merged loop
 - As in the previous example where the maximum loop bounds become 16 (implied by L3 flattened into L2 before the merge)
- ▶ Loops with both variable bound and constant bound **cannot** be merged
- ▶ The code between loops to be merged cannot have side effects
 - Multiple execution of this code should generate same results
 - A=B is OK, A=A+1 is not
- ▶ Reads from a FIFO or FIFO interface must always be in sequence
 - A FIFO read in one loop will not be a problem
 - FIFO reads in multiple loops may become out of sequence
 - This prevents loops being merged

Loop Reports

- ▶ Vitis HLS reports the latency of loops
 - Shown in the report file and GUI
- ▶ Given a variable loop index, the latency cannot be reported
 - Vitis HLS does not know the limits of the loop index
 - This results in latency reports showing unknown values
- ▶ The loop tripcount (iteration count) can be specified
 - Apply to the loop in the directives pane
 - Allows the reports to show an estimated latency

Impacts reporting – not synthesis



Improving Performance 21- 54

©2022 Advanced Micro Devices, Inc.

AMD
XILINX

Techniques for Minimizing Latency - Summary

► Constraints

- Vitis HLS accepts constraints for latency

► Loop Optimizations

- Latency can be improved by minimizing the number of loop boundaries
 - Rolled loops (default) enforce sharing at the expense of latency
 - The entry and exits to loops costs clock cycles
- Reduce latency by:
 - Fully unrolling loops (but creating a lot of hardware)
 - Partial unrolling
 - Converting imperfect to perfect loops
 - Merging sequential loops: Allows for more efficient architecture explorations

Summary

Summary

- ▶ Directives may be added through GUI
 - Tcl command is added into script.tcl file
 - Pragmas are added into the source file
- ▶ Latency is minimized by default
 - Constraints can be set
- ▶ Loops may have impact on the latency
- ▶ Throughput may be improved by pipelining at
 - The task, function, and loop level
- ▶ Arrays may create performance bottleneck if not handled properly

Summary

► Optimizing Performance

- Throughput optimization
 - Perform Dataflow optimization at the top-level
 - Pipeline individual functions and/or loops
 - Pipeline the entire function: beware of lots of operations, lots to schedule and it's not always possible
- Latency optimization
 - Specify latency directives
 - Unroll loops
 - Merge and Flatten loops to reduce loop transition overheads
- Array Optimizations
 - Focus on bottlenecks often caused by memory and port accesses
 - Removing bottlenecks improves latency and throughput
 - Use Array Partitioning, Reshaping, and Data packing directives to achieve throughput

Hands-on session!

Lab – Improving performance with pragmas

Code for this lab again on [github](#)

Start at /day 3, you will find

- /yuv_filter: Image processing
pipeline and dataflow pragmas
→ detailed instructions in README.md
- /loop_optimizations: Generic loop optimization strategies
see the effect of various pragmas on loops
→ detailed instructions in README.md

What can you expect in the coming days?

Monday, 23.03.2026

What is an FPGA? And why physicists care about them!
Motivation, Architecture, Electronics, Programming Model, Basics of HLS

Tuesday, 24.03.2026

Thinking in Hardware
Timing Terminology, Paradigms, HLS Data Types, Pointers, Streams
Lab: Matrix Multiplication, Pointers, and Streams

Wednesday, 25.03.2026

Interfaces and Improving Performance
Interfaces, Directives, Improving Throughput and Latency, Bottlenecks
Lab: YUV-Filter, Loops Optimizations

Thursday, 26.03.2026

Improving Area and Resources
Allocation & Structure Control, Hierarchies, Arbitrary Precision
Lab: DCT, Matrix Multiplication revisited

Friday, 27.03.2026

Task Level Parallelism, and how to deal with Timing Violations
Lab: Freely running kernel, Timing violation

Improving Area and Resources

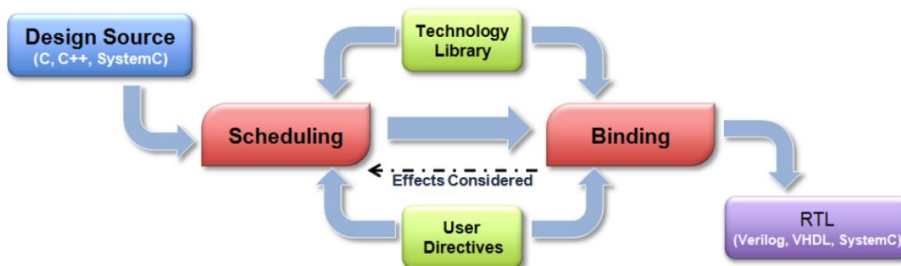




Improving Area and Resources

Review: Control Scheduling & Binding

- ▶ Scheduling & Binding
 - Scheduling and Binding are the processes at the heart of HLS



- ▶ BIND_OP/ BIND_STORAGE
 - Can be used to specify a device resource for implementation
 - Can be used to assign a specific memory type in the RTL
- ▶ The allocation directive
 - Can be used to limit the number of operation in scheduling & binding stages

Directives to improve Area

Directives and Configurations	Description
ALLOCATION	Specify a limit for the number of operations, cores, or functions used. This can force the sharing of hardware resources and might increase latency
ARRAY_RESHAPE	Reshapes an array from one with many elements to one with greater word-width. Useful for improving block RAM accesses without using more block RAM
LOOP_MERGE	Merges consecutive loops to reduce overall latency, increase sharing, and improve logic optimization
OCCURRENCE	Used when pipelining functions or loops to specify that the code in a location is executed at a lesser rate than the code in the enclosing function or loop
STREAM	Specifies that a specific memory channel is to be implemented as a FIFO or RAM during dataflow optimization
Config Dataflow	This configuration specifies the default memory channel and FIFO depth in a dataflow region
BIND_STORAGE	Define a specific implementation for a storage element, or memory, in the RTL
BIND_OP	Define a specific implementation for an operation in the RTL

Reducing Area Usage

Improving Area/Resource Utilization

- ▶ Control the number of elements
 - Directives can be used to control scheduling and binding phases, which in turn control the number of elements used
- ▶ Control the design hierarchy
 - Like RTL synthesis, removing the hierarchy can help optimize across function and loop boundaries
 - Functions can be inlined
 - Loops can be unrolled
- ▶ Array implementation
 - VITIS HLS provides directives for combining memories
 - Allowing a single large memory to be used instead of multiple smaller memories
- ▶ Bit-width optimization
 - Bit width impact the size of the storage elements and operators
 - Arbitrary precision types ensure correct operator sizing

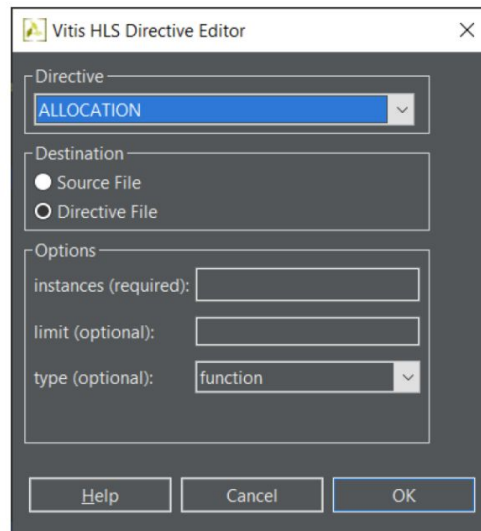
Allocation: Limit the Numbers

► Allocation directive limits different types

- Type: Operation
 - The instances are the operators
 - Add, mul, urem, etc.
- Type: Core
 - The instances are the cores
 - Adder, Addsub, PipeMult2s, etc
- Type: Functions
 - The functions in the code
 - Discussed in more detail later

► Allocations are defined for a scope

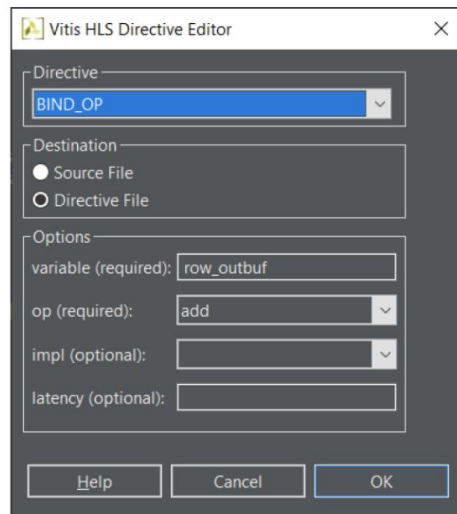
- Like all directives, allocations are set for the scope they are applied in
- If the directive is applied to a function, loop or region, it does not include objects outside that scope



Operators and Cores are listed in the VITIS HLS Library Guide

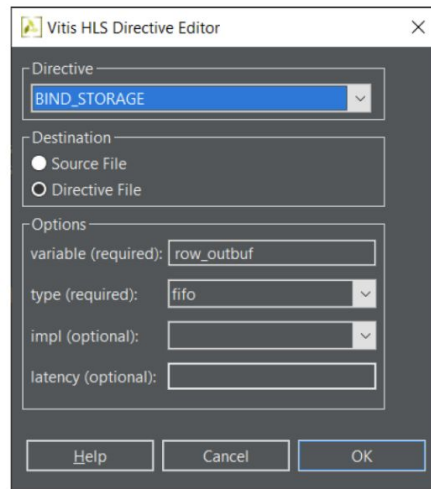
BIND_OP: Specific a device resource

- ▶ Specifies that for a given variable, an operation(mul, add,sub) should be mapped to a device resource for implementation
 - If not, Vitis HLS tool automatically determines the resource to use
- ▶ You can also specify:
 - -op: Operation to bind to a specific implementation resource
 - -impl: Implementation to use for the specified operation
 - -latency: Latency for the operation



BIND_STORAGE: Assign a specific memory

- ▶ Assigns a variable(array or function argument) to a specific memory type in the RTL
 - If not, Vitis HLS tool determines the memory type to assign
 - HLS implements the memory using specified implementations(impl) in the hardware
 - Controls the array implementation as a single/dual-port RAM
- ▶ You can also specify:
 - -type: Type of memory to bind to the specified variable
 - Support types: fifo, ram_1p, ram_1wnr, ram_2p, ram_s2p, ram_t2p, rom_1p, rom_2p, and rom_np
 - -impl: Implementation for the specified memory type
 - Support implementations: bram, bram_ecc, lutram, uram, uram_ecc and srl
 - -latency: Default latency for the binding of the storage type to the implementation



Review: Functions & RTL Hierarchy

- ▶ Each function in the design gets synthesized into an RTL block, maintaining the design hierarchy

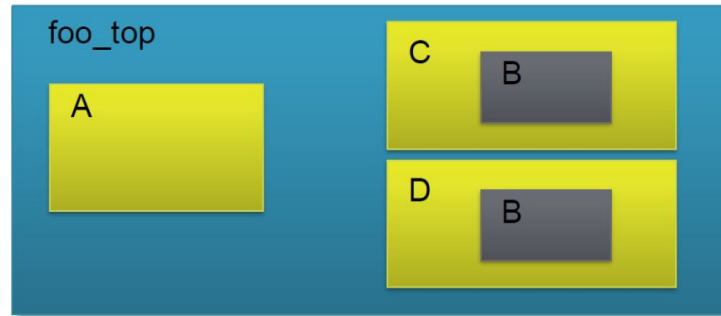
Source Code

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

my_code.c



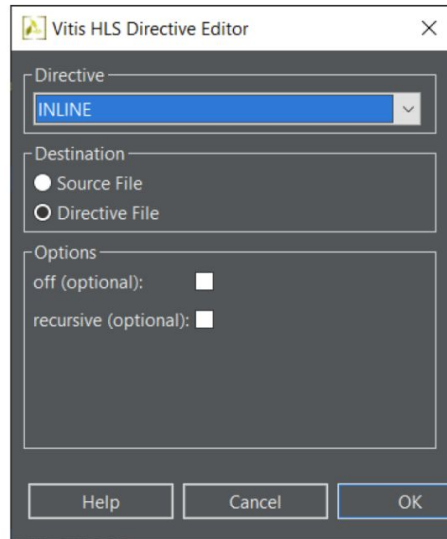
RTL hierarchy



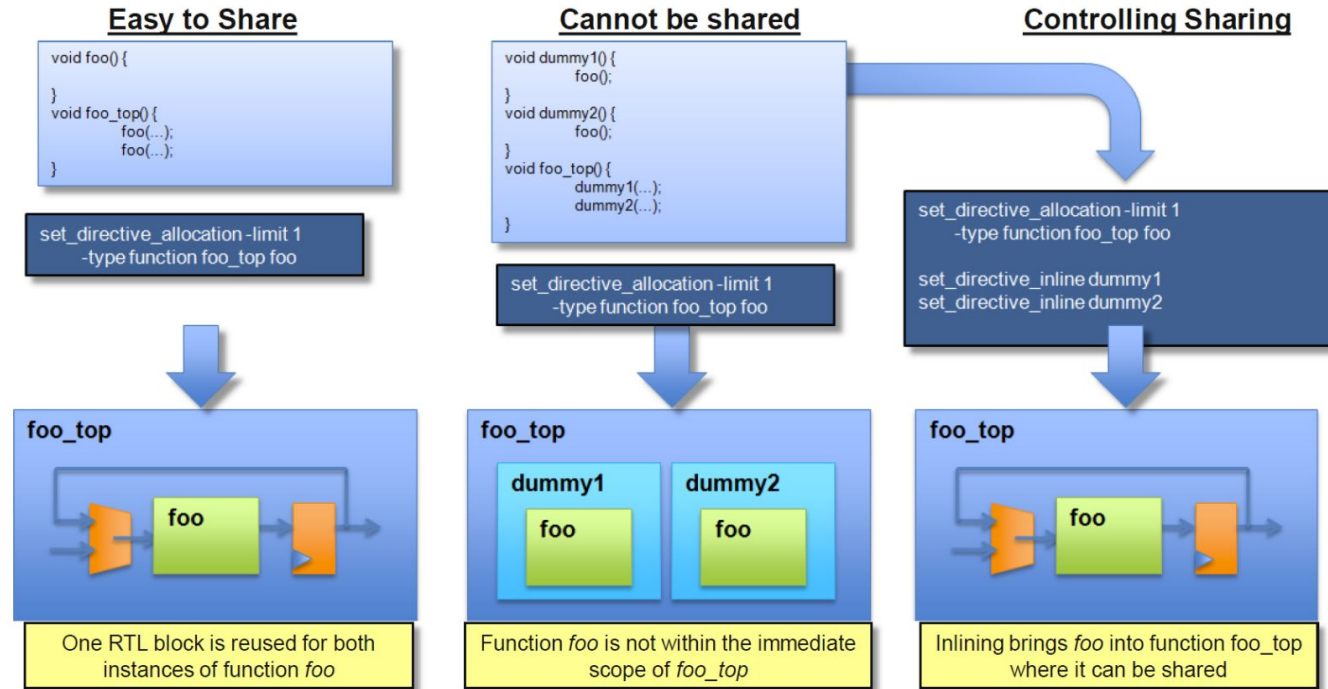
Functions can be inlined – the hierarchy removed & the function dissolved into the surrounding function

Function Inlining

- ▶ Inlining a function may improve area by allowing the components within the function to be better shared or optimized with the logic in the calling function.
- ▶ Vitis HLS performs some inlining automatically
 - This is performed on small logic functions if Vitis HLS determines area or performance will benefit
- ▶ User Control
 - Optionally recursively down the hierarchy
 - Optionally inlining can be explicitly prevented
 - Turn inlining off
- ▶ Inlining functions allows for greater optimization
 - Like ungrouping RTL hierarchies: optimization across boundaries
 - Like ungrouping RTL hierarchies it can result in lots of operations & impact run time



Inline and Allocation: Shape the Hierarchy

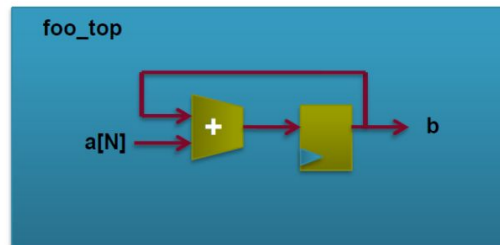


Loops

- ▶ By default, loops are rolled and pipelined to improve performance
 - Each C loop iteration → Implemented in the same state with same resources

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

Synthesis



- ▶ For Area optimization

Keeping loops rolled maximizes sharing across loop iterations: each iteration of the loop uses the same hardware resources

Loop Merging

- ▶ Loop merging can remove the redundant computation among multiple (related) loops
 - Improving area and performance

```
My_Region: {  
  #pragma HLS merge loop  
  for (i = 0; i < N; ++i)  
    A[i] = B[i] + 1;  
  for (i = 0; i < N; ++i)  
    C[i] = A[i] / 2;  
}
```

Merge

```
for (i = 0; i < N; ++i) {  
  A[i] = B[i] + 1;  
  C[i] = A[i] / 2;  
}
```

Effective code after compiler transformation

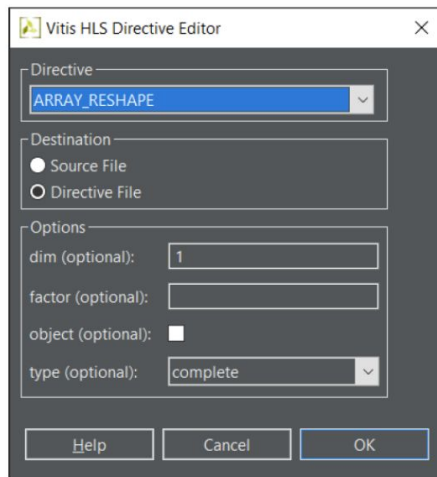
- ▶ Allows the logic within the loops to be optimized together
- ▶ Optimization cannot occur across loop boundaries
- ▶ Allows for more efficient architecture explorations

```
for (i = 0; i < N; ++i)  
  C[i] = (B[i] + 1) / 2;
```

Removes A[i], any address logic and any potential memory accesses

Mapping Arrays

- ▶ The arrays in the C model may not be ideal for the available RAMs
 - The code may have many small arrays, each array is mapped into a block RAM or UltraRAM
 - Basic block RAM unit is 18K, the array may not utilize the RAMs very well
- ▶ Array Mapping
 - Mapping combines smaller arrays into larger arrays
 - Reduce the number of block RAMs required
 - Specify the array variable to be mapped
 - Give all arrays to be combined the same instance name
 - If an array is larger than 18K, they are automatically mapped into multiple 18K units.
- ▶ `ARRAY_RESHAPE` directive supports mapping small arrays into a larger one



Arbitrary Precision Integers

- ▶ C and C++ have standard types created on the 8-bit boundary
 - char (8-bit), short (16-bit), int (32-bit), long long (64-bit)
 - Also provides stdint.h (for C), and stdint.h and cstdint (for C++)
 - Types: int8_t, uint16_t, uint32_t, int_64_t etc.
 - Implemented hardware sometimes need different sizes of data types
 - They result in hardware which is not bit-accurate and can give sub-standard QoR
 - Signals with arbitrary widths are also converted into fixed sized widths when native C data types are used
- ▶ Vitis HLS provides both integer and fixed-point arbitrary precision data types for C++
 - Allow any arbitrary bit-width to be specified
 - Will simulate with bit-accuracy

```
#include ap_cint.h
void foo_top (...) {
    int1        var1;      // 1-bit
    uint1       var1u;     // 1-bit unsigned
    int2        var2;      // 2-bit
    ...
    int1024     var1024;    // 1024-bit
    uint1024    var1024u;  // 1024-bit unsigned
    ...
}
```

my_code.c

```
#include ap_int.h
void foo_top (...) {
    ap_int<1>    var1;      // 1-bit
    ap_uint<1>  var1u;     // 1-bit unsigned
    ap_int<2>    var2;      // 2-bit
    ...
    ap_int<1024> var1024;   // 1024-bit
    ap_uint<1024> var1024u; // 1024-bit unsigned
    ...
}
```

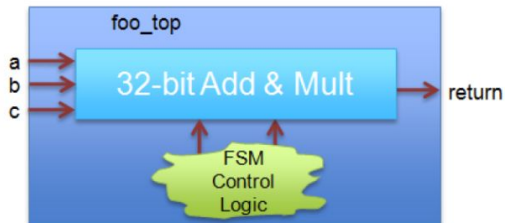
my_code.cpp

Why are Arbitrary Precision types Needed?

▶ Code using native C int type

```
int foo_top(int a, int b, int c)
{
    int sum, mult;
    sum=a+b;
    mult=sum*c;
    return mult;
}
```

Synthesis



▶ However, if the inputs will only have a max range of 8-bit

- Arbitrary precision data-types should be used

```
int17 foo_top(int8 a, int8 b, int8 c)
{
    int9 sum;
    int17 mult
    sum=a+b;
    mult=sum*c;
    return mult;
}
```

Synthesis



- It will result in smaller & faster hardware with full precision, full precision can be simulated/validated with C simulation and hardware will behave the same

DEPENDENCE Directive

Loop-Independent Dependence

Same element is accessed in a single loop iteration

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

Loop-Carried Dependence

Same element is accessed from a different loop iteration

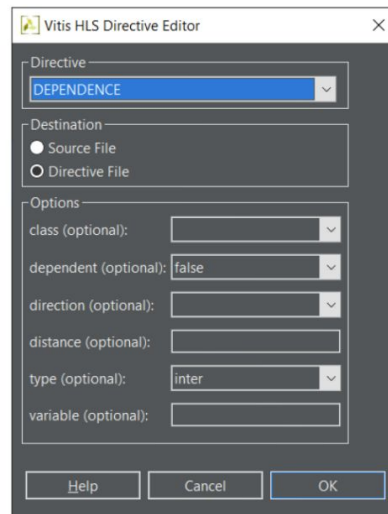
```
for (i=1; i<N; i++) {  
    A[0]=0;  
    A[i]=A[i-1]*2;  
}
```

DEPENDENCE directive allows you to explicitly define the dependencies and eliminate a false dependence

DEPENDENCE Directive

- ▶ DEPENDENCE directive allows to explicitly define the dependencies and eliminate a false dependence
 - Provides the tool with additional information about the dependencies
 - Allows designers to inform tool of "external" conditions
 - Makes sure there are no dependencies between loop iterations

```
for (row = 0; row < rows + 1; row++) {  
  L2: for (col = 0; col < cols + 1; col++) {  
    #pragma HLS PIPELINE II=1  
    #pragma HLS dependence variable=buff_A type=inter dependent=false  
    #pragma HLS dependence variable=buff_B type=inter dependent=false  
    if (col < cols) {  
      buff_A[2][col] = buff_A[1][col]; // read from buff_A[1][col]  
      buff_A[1][col] = buff_A[0][col]; // write to buff_A[1][col]  
      buff_B[1][col] = buff_B[0][col];  
      temp = buff_A[0][col]; } } }
```

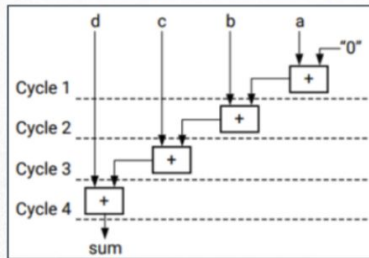


Optimizing Logic Expressions

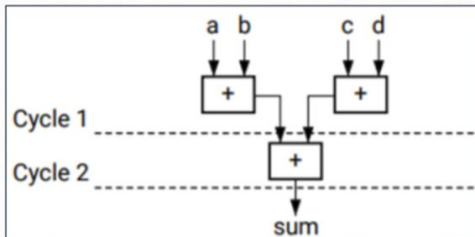
- ▶ Expression balancing rearranges operators to construct a balanced tree and reduce latency
 - For integer operations expression balancing is on by default but may be disabled using the `EXPRESSION_BALANCE` pragma or directive.
 - For floating-point operations, expression balancing is off by default but may be enabled using the `config_compile -unsafe_math_optimizations` command.

```
data_t foo_top (data_t a, data_t b, data_t c, data_t d) {
    data_t sum;
    sum = 0;
    sum += a;
    sum += b;
    sum += c;
    sum += d;
    return sum;
}
```

Without Expression Balancing

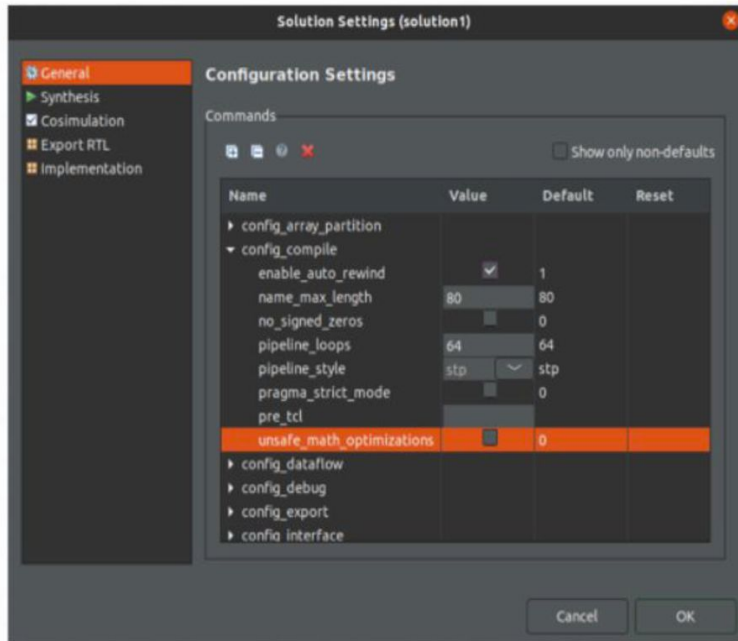


With Expression Balancing



Optimizing Logic Expressions

- ▶ To configure the tool to enable expression balancing with float and double types
- ▶ Place the pragma in the C source within the boundaries of the required location.
 - #pragma HLS expression_balance off
 - Turns off expression balancing at this location.
 - Specifying #pragma HLS expression_balance enables expression balancing in the specified scope. Adding off disables it.
- ▶ Solution settings > General > Config_compile>unsafe_math_optimizations



Summary

Summary

- ▶ Resource utilization can be reduced using allocation and binding controls
 - Allocation can define how many resource are used
- ▶ The design structure can be controlled by
 - Inlining functions: direct impact on RTL hierarchy & optimization possibilities
 - Loops: direct impact on reuse of resources
 - Arrays: direct impact on the RAM
- ▶ Major area optimization techniques
 - Minimize bit widths
 - Map smaller arrays into larger arrays
 - Make better use of existing RAMs
 - Control loop hierarchy
 - Control function call hierarchy
 - Control the number of operators and cores
- ▶ Arbitrary precision data types help controlling both the area and resource utilization

Hands-on session!

Lab – Improving Area and Resources with pragmas

Code for this lab again on [github](#)

Start at /day 4, you will find

- /dct: Discrete Cosine Transformation
Improve performance, area and resources with pragmas
→ detailed instructions in README.md
- /matmul_revisited: Matrix multiplication → some more free coding exercise!
Improve performance with architectural changes and pragmas
→ detailed instructions in README.md

What can you expect in the coming days?

Monday, 23.03.2026

What is an FPGA? And why physicists care about them!
Motivation, Architecture, Electronics, Programming Model, Basics of HLS

Tuesday, 24.03.2026

Thinking in Hardware
Timing Terminology, Paradigms, HLS Data Types, Pointers, Streams
Lab: Matrix Multiplication, Pointers, and Streams

Wednesday, 25.03.2026

Interfaces and Improving Performance
Interfaces, Directives, Improving Throughput and Latency, Bottlenecks
Lab: YUV-Filter, Loops Optimizations

Thursday, 26.03.2026

Improving Area and Resources
Allocation & Structure Control, Hierarchies, Arbitrary Precision
Lab: DCT, Matrix Multiplication revisited

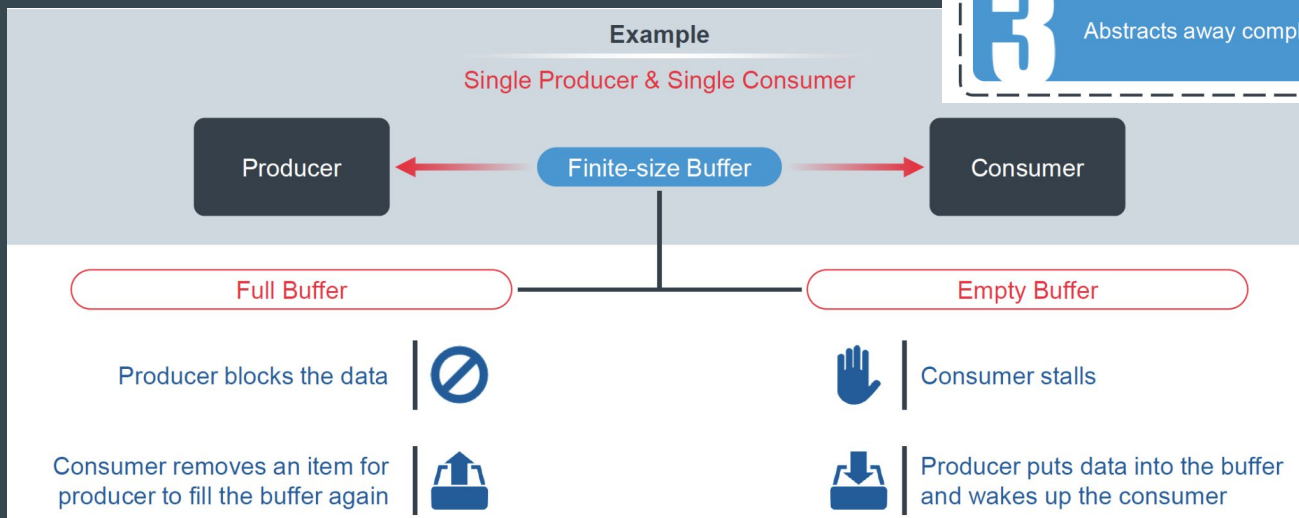
Friday, 27.03.2026

Task Level Parallelism, and how to deal with Timing Violations
Lab: Freely running kernel, Timing violation

Task Level Parallelism, and how to deal with Timing Violations



Producer-Consumer Paradigm



Advantages of using this macro-level architectural optimization

1

No worrying about memory models and other non-deterministic behavior

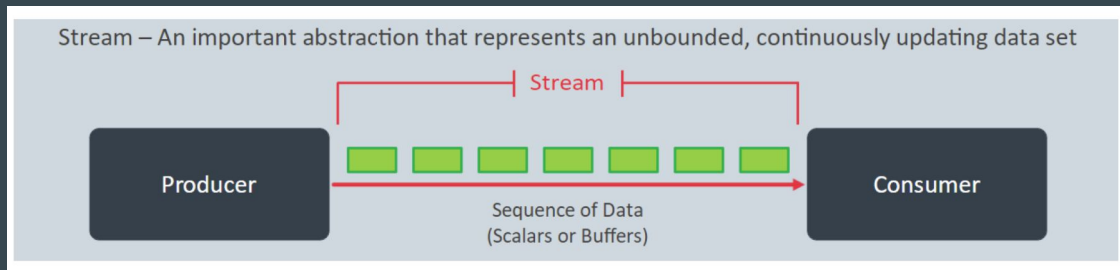
2

Dataflow network achieved

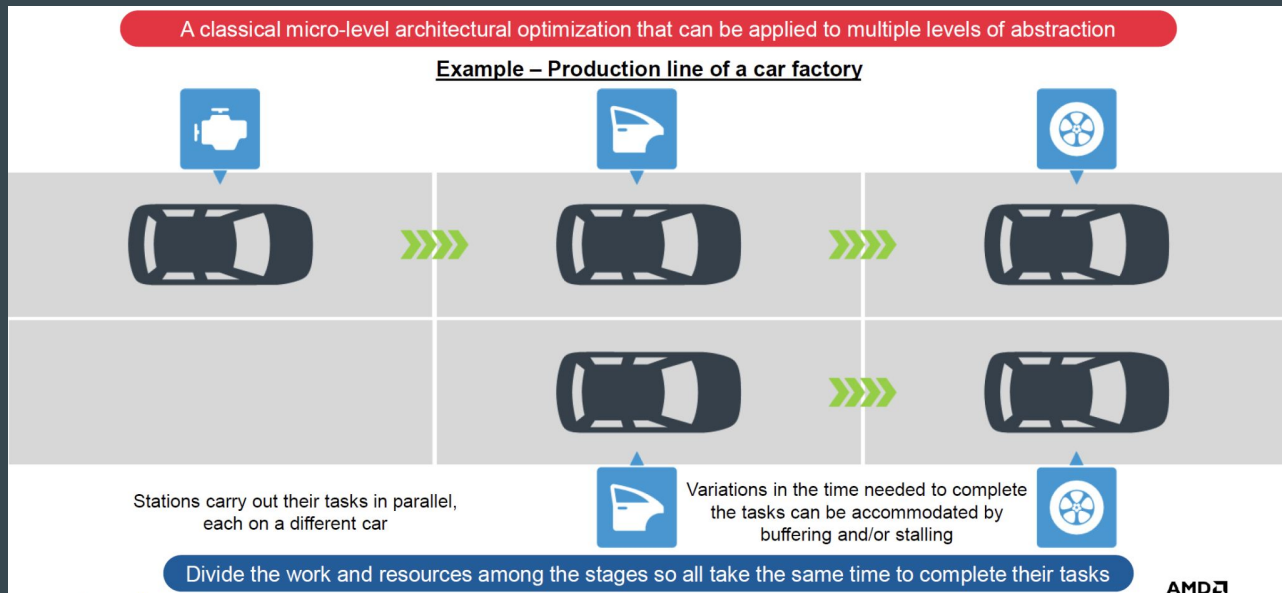
3

Abstracts away complexities of a parallel program

Stream data Paradigm



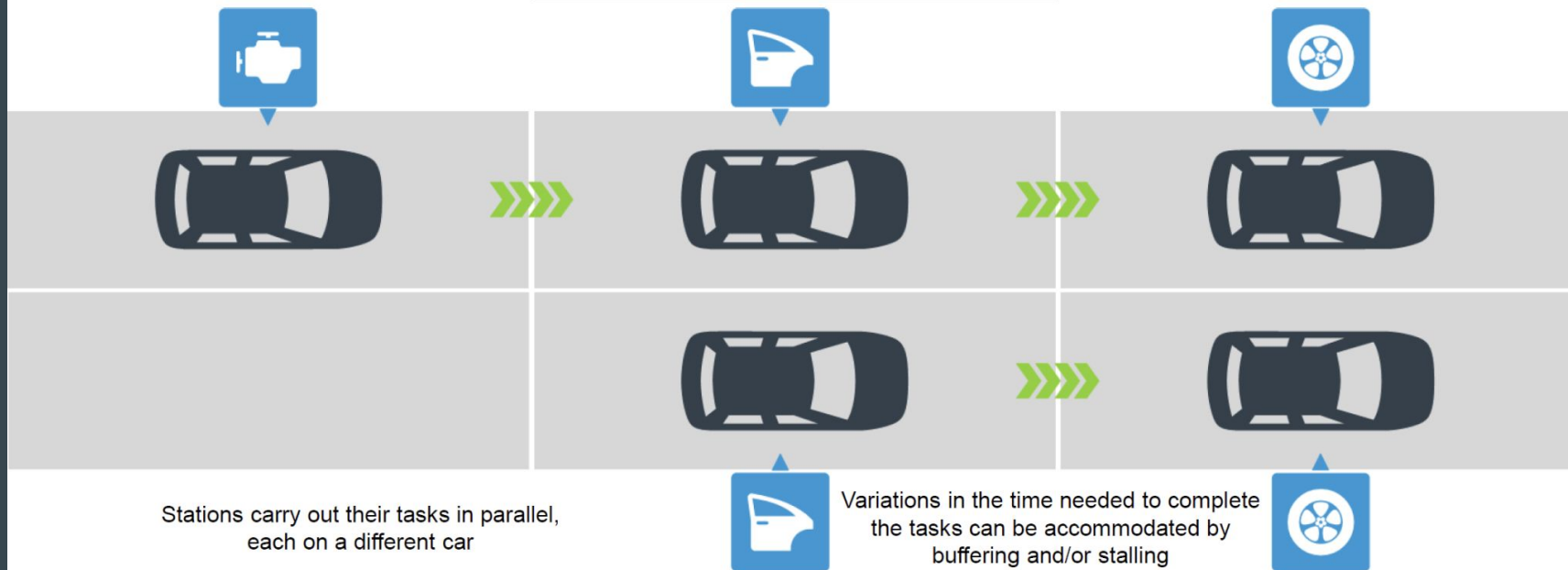
Pipelining Paradigm



Pipelining Paradigm

A classical micro-level architectural optimization that can be applied to multiple levels of abstraction

Example – Production line of a car factory



Divide the work and resources among the stages so all take the same time to complete their tasks

Task-level Parallelism (TLP)

Data-driven task-level parallelism

- No interaction with external memory
- Functions executed in parallel
- Always running
- No data dependencies

Control-driven task-level parallelism

- Some interaction with external memory
- Data dependencies between parallel tasks
- Dataflow optimization in Vitis HLS

Data-driven TLP

- Statically instantiate and connect tasks and channels explicitly
- Tasks are not controlled by function call/return semantics
- Typically only stream type input and output
- Always running, waiting for data on their input stream
- Modeling tasks
`hls::task`
- Modeling channels
`hls::stream`
`hls::stream_of_blocks`

```
#include "test.h"
```

```
void splitter(hls::stream<int> &in, hls::stream<int> &odds_buf, hls::stream<int> &evens_buf) {  
    int data = in.read();  
    if (data % 2 == 0)  
        evens_buf.write(data);  
    else  
        odds_buf.write(data);  
}
```

```
void odds(hls::stream<int> &in, hls::stream<int> &out) {  
    out.write(in.read() + 1);  
}
```

```
void evens(hls::stream<int> &in, hls::stream<int> &out) {  
    out.write(in.read() + 2);  
}
```

```
void odds_and_evens(hls::stream<int> &in, hls::stream<int> &out1, hls::stream<int> &out2) {  
    hls_thread_local hls::stream<int> s1; // channel connecting t1 and t2  
    hls_thread_local hls::stream<int> s2; // channel connecting t1 and t3  
  
    // t1 infinitely runs function splitter, with input in and outputs s1 and s2  
    hls_thread_local hls::task t1(splitter, in, s1, s2);  
    // t2 infinitely runs function odds, with input s1 and output out1  
    hls_thread_local hls::task t2(odds, s1, out1);  
    // t3 infinitely runs function evens, with input s2 and output out2  
    hls_thread_local hls::task t3(evens, s2, out2);  
}
```

Data-driven TLP

- `hls_thread_local` keeps the object and underlying thread alive across multiple calls of instantiating function
→ only required for C sim to match RTL sim
- Task objects implicitly manage a thread that runs infinitely
- Supplied function (evens/odds) is task body, has an implicit infinite loop wrapped around it
- Supplied functions can contain pipelined loops, that need to be flushable (FLP)
- **IMPORTANT:**
`hls::task` is not a function call!

```
#include "test.h"
```

```
void splitter(hls::stream<int> &in, hls::stream<int> &odds_buf, hls::stream<int> &evens_buf) {  
    int data = in.read();  
    if (data % 2 == 0)  
        evens_buf.write(data);  
    else  
        odds_buf.write(data);  
}
```

```
void odds(hls::stream<int> &in, hls::stream<int> &out) {  
    out.write(in.read() + 1);  
}
```

```
void evens(hls::stream<int> &in, hls::stream<int> &out) {  
    out.write(in.read() + 2);  
}
```

```
void odds_and_evens(hls::stream<int> &in, hls::stream<int> &out1, hls::stream<int> &out2) {  
    hls_thread_local hls::stream<int> s1; // channel connecting t1 and t2  
    hls_thread_local hls::stream<int> s2; // channel connecting t1 and t3  
  
    // t1 infinitely runs function splitter, with input in and outputs s1 and s2  
    hls_thread_local hls::task t1(splitter, in, s1, s2);  
    // t2 infinitely runs function odds, with input s1 and output out1  
    hls_thread_local hls::task t2(odds, s1, out1);  
    // t3 infinitely runs function evens, with input s2 and output out2  
    hls_thread_local hls::task t3(evens, s2, out2);  
}
```

Data-driven TLP

- `hls::stream<type, depth>` behaves like a FIFO with specified depth
- Access to different streams are not ordered
- Streams can be defined globally or locally

In summary, `hls::task` model is

- recommended if your design requires a completely data-driven, pure streaming type of behavior, with no sort of control.
- useful in modeling feedback (cyclic dependency between tasks)
- useful in modeling dynamic multi-rate designs (producer writes or consumer reads at a rate that is data dependent)

```
#include "test.h"
```

```
void splitter(hls::stream<int> &in, hls::stream<int> &odds_buf, hls::stream<int> &evens_buf) {  
    int data = in.read();  
    if (data % 2 == 0)  
        evens_buf.write(data);  
    else  
        odds_buf.write(data);  
}
```

```
void odds(hls::stream<int> &in, hls::stream<int> &out) {  
    out.write(in.read() + 1);  
}
```

```
void evens(hls::stream<int> &in, hls::stream<int> &out) {  
    out.write(in.read() + 2);  
}
```

```
void odds_and_evens(hls::stream<int> &in, hls::stream<int> &out1, hls::stream<int> &out2) {  
    hls_thread_local hls::stream<int> s1; // channel connecting t1 and t2  
    hls_thread_local hls::stream<int> s2; // channel connecting t1 and t3  
  
    // t1 infinitely runs function splitter, with input in and outputs s1 and s2  
    hls_thread_local hls::task t1(splitter, in, s1, s2);  
    // t2 infinitely runs function odds, with input s1 and output out1  
    hls_thread_local hls::task t2(odds, s1, out1);  
    // t3 infinitely runs function evens, with input s2 and output out2  
    hls_thread_local hls::task t3(evens, s2, out2);  
}
```

Control-driven TLP

- Model parallelism while relying on the sequential semantics of C++, rather than continuously running threads
- Functions executed in concurrent, pipelined fashions (within loops), and/or arguments that are C++ scalar or array variables (on-chip / off-chip memory)
- Control-driven TLP provides:
 - A subsequent function can start before the previous finishes
 - A function can be restarted before it finishes
 - Two or more sequential functions can be started simultaneously

```
#include "diamond.h"

void diamond(data_t vecIn[N], data_t vecOut[N])
{
    data_t c1[N], c2[N], c3[N], c4[N];
    #pragma HLS dataflow
    funcA(vecIn, c1, c2);
    funcB(c1, c3);
    funcC(c2, c4);
    funcD(c3, c4, vecOut);
}
```

Control-driven TLP

- While using the dataflow model, HLS automatically inserts synchronization and communication between the tasks
- Creates task-level pipeline architecture of concurrent processes
- Channels can be FIFO (`hls::stream`) or PIPO or `hls::stream_of_blocks`
- Dataflow model not limited to a chain of processes, can be used on any directed acyclic graph, or cyclic structure with streams
- Can produce overlapping within iteration with FIFOs, and across iterations when connected with PIPOs and FIFOs

```
#include "diamond.h"
```

```
void diamond(data_t vecIn[N], data_t vecOut[N])  
{  
    data_t c1[N], c2[N], c3[N], c4[N];  
    #pragma HLS dataflow  
    funcA(vecIn, c1, c2);  
    funcB(c1, c3);  
    funcC(c2, c4);  
    funcD(c3, c4, vecOut);  
}
```


Control-driven TLP

- Canonical coding style
dataflow network matches function calls in source code → more predictable
- Non-canonical coding style
HLS needs to analyze loops, etc.
less direct for the user
- The dataflow pragma can be applied in two different contexts: Inside a
 - Function (aka function dataflow region).
 - Special kind of for loop (aka loop dataflow region) such that, with the loop being the only statement in function (and other requirements)

```
#include "diamond.h"
```

```
void diamond(data_t vecIn[N], data_t vecOut[N])  
{  
    data_t c1[N], c2[N], c3[N], c4[N];  
    #pragma HLS dataflow  
    funcA(vecIn, c1, c2);  
    funcB(c1, c3);  
    funcC(c2, c4);  
    funcD(c3, c4, vecOut);  
}
```

```
void noncanonical(int Input0, int Input1[], int  
&Output0, int Output1[]) {  
    #pragma HLS dataflow  
    int C0, C1[N], C2;  
    C0 = Input0 * 3;  
    for (int i = 0; i < N; i++) {  
        C1[i] = Input1[0] + 2;  
    }  
    func2(C0, C1, C2);  
    func3(C2, Output0, Output1);  
}
```

Nested dataflow pitfall I

- The performance of nested dataflow regions can be greatly impacted if any non-dataflow regions are present other than at the deepest level of the function/region hierarchy (for example, the leaf node of the dataflow graph).
- This is because sequential FSMs do not allow overlapped execution of the (possibly dataflow) regions that they contain.

```
void proc1(...) {  
    #pragma HLS dataflow  
    ...  
}  
  
void proc2(...) {  
    ... // missing dataflow pragma  
    proc1();  
    ...  
}  
  
void top(...) {  
    #pragma HLS dataflow  
    ...  
    proc2(...);  
    ...  
}
```

Nested dataflow pitfall II

- The performance of nested dataflow regions can be greatly impacted if any non-dataflow regions are present other than at the deepest level of the function/region hierarchy (for example, the leaf node of the dataflow graph).
- This is because sequential FSMs do not allow overlapped execution of the (possibly dataflow) regions that they contain.

```
void proc1(...) {  
    #pragma HLS dataflow  
    ...  
}  
  
void proc2(...) {  
    a = x + y; // this code is NOT implemented as a  
               dataflow process but as a sequential FSM  
    for (int i = 0; i < N; i++) {  
        #pragma HLS dataflow  
        proc1(a, ...);  
        ...  
    }  
}  
  
void top(...) {  
    #pragma HLS dataflow  
    ...  
    proc2(...);  
    ...  
}
```

Advanced dataflow regions

- Unrolling and partitioning for parameterized task-level parallelism
- A dataflow pragma is required to ensure that the correctness of the region is checked before unrolling, and after unrolling the code must satisfy the canonical dataflow conditions.
- Partitioned arrays, where each partition is passed as an independent variable (array or scalar) to one process.

```
void dut(int in[M], int out[M]) {  
    #pragma HLS dataflow  
    hls_thread_local hls::stream<int> chan[N+1]; //  
    arrays of hls::streams are fully partitioned  
    automatically  
    read_in(in, chan[0]);  
    hls_thread_local hls::task t[N]; // array of worker  
    processes  
    for (int i=0; i<N; i++) {  
        #pragma HLS unroll  
        #pragma HLS dataflow  
        t[i](worker, chan[i], chan[i+1]);  
    }  
    write_out(chan[N], out);  
}
```

Advanced dataflow regions

- Unrolling and partitioning for parameterized task-level parallelism
- A dataflow pragma is required to ensure that the correctness of the region is checked before unrolling, and after unrolling the code must satisfy the canonical dataflow conditions.
- Partitioned arrays, where each partition is passed as an independent variable (array or scalar) to one process.
- Also with partial partitioning and a partial unrolling

```
void dut(int a[10], int b[10]) {  
    #pragma HLS array_partition cyclic factor=2  
    variable=a  
  
    #pragma HLS array_partition cyclic factor=2  
    variable=b  
  
    for (int i=0; i<10; i++) {  
        #pragma HLS dataflow  
        #pragma HLS unroll factor=2  
        compute(a[i], b[i]); // read a[i], write b[i]  
    }  
}
```

Loop-carried Dependencies

- `hls::stream` to transfer data backwards
- Ensure that read data from feedback streams not until data has been produced by later processes, eg by skipping first execution
 - Static variable
 - Loop counter of surrounding dataflow loop
 - `hls_thread_local` variable in `hls::task` context

```
void read_and_add(int a[N], hls::stream<int> &b, hls::stream<int>
&f, int i) {
#pragma HLS pipeline II=1 style=flp // flushing pipeline is needed
to avoid deadlocks
    int t=0;
    if (i>0) t = f.read(); // reading previous sum, except first time
    b.write(a[i]+t); // adding and passing
}

void write_and_feedback(hls::stream<int> &a, int b[N],
hls::stream<int> &f, int i) {
#pragma HLS pipeline II=1 style=flp
    int aa=a.read();
    if (i<N-1) f.write(aa); // passing back current sum
    b[i] = aa; // writing it out
}

void dut(int a[N], int b[N]) {
    for (int i=0; i<N; i++) {
#pragma HLS DATAFLOW
        static hls::stream<int> f; // feedback stream, here could also
be declared without static, above the loop
        hls::stream<int> c; // forward stream

        read_and_add(a, c, f, i);
        write_and_feedback(c, b, f, i);
    }
}
```

Dataflow Memory Channels

- PIPO or FIFO, depends on user's choice
 - Scalars: FIFO automatically
 - Array: sequential access → may choose FIFO/PIPO → PIPO default
 - Array: random access → PIPO
- FIFO: less memory, risk of deadlock
- PIPO: more memory, never deadlock
- Vitis HLS IDE can help with determining optimal FIFO sizes from RTL co-simulation

```
void top ( ... ) {  
    #pragma HLS dataflow  
        int A[1024];  
    #pragma HLS stream type=pipo variable=A depth=3  
  
    producer(A, B, ...); // producer writes A and B  
    middle(B, C, ...);   // middle reads B and writes C  
    consumer(A, C, ...); // consumer reads A and C  
}
```


Stream-of-Blocks

- Streaming of blocks of data, each block is array or multi-dim array
- The producer or consumer process that wants to access the stream first needs to acquire access to it
→ write_lock, read_lock
- Released when lock goes out of scope
- Producer can also read the block it has write access to
- Requires less storage compared to PIPO:
 $2 \times N$ instead of $2 \times M \times N$

```
#include "hls_streamofblocks.h"
typedef int buf[N];
void producer (hls::stream_of_blocks<buf> &s, ...) {
    for (int i = 0; i < M; i++) {
        // Allocation of hls::write_lock acquires the block for the producer
        hls::write_lock<buf> b(s);
        for (int j = 0; j < N; j++)
            b[f(j)] = ...;
        // Deallocation of hls::write_lock releases the block for the consumer
    }
}

void consumer(hls::stream_of_blocks<buf> &s, ...) {
    for (int i = 0; i < M; i++) {
        // Allocation of hls::read_lock acquires the block for the consumer
        hls::read_lock<buf> b(s);
        for (int j = 0; j < N; j++)
            ... = b[g(j)] ...;
        // Deallocation of hls::write_lock releases the block to be reused by
        the producer
    }
}

void top(...) {
#pragma HLS dataflow
    hls::stream_of_blocks<buf> s;

    producer(b, ...);
    consumer(b, ...);
}
```

Mixing Data-Driven and Control-Driven Models

Control-Driven TLP	Data-Driven TLP
Designs with multiple processes running the same number of executions	Task-level parallelism is observable in C simulation and RTL simulation.
HLS Design requires control signals to start/stop the process	HLS Design uses a completely data-driven approach that does not require control signals to start/stop the process
Design requires interaction with an external software application	Designs with data-dependent multi-rate behavior: • Producer writes data or consumer reads data at a rate that is data dependent • Easier to model for designs that require feedback between processes
Design requires non-local memory access	Design uses pure streaming data transfers
Requires RTL simulation to model the effect of the parallelism	

Example Mixed Model

- Dataflow region read_in / write_out
- Task-channel region with dynamic instantiation of 4 tasks with special channels (split / merge → see `hls::split`, `hls::merge`)
- Tasks are statically instantiated, not “called” in the loop
- If a control-based process (i.e. regular dataflow) produces a stream for an `hls::task`, then it must be called before the declaration of the tasks in the code
- If a control-based process consumes a stream from an `hls::task`, then it must be called after the declaration of the tasks in the code

```
void dut(int in[N], int out[N], int n) {  
    #pragma HLS dataflow  
    hls_thread_local hls::split::round_robin<int, NP>  
    split1;  
    hls_thread_local hls::merge::round_robin<int, NP>  
    merge1;  
  
    read_in(in, n, split1.in);  
  
    // Task-Channels  
    hls_thread_local hls::task t[NP];  
    for (int i=0; i<NP; i++) {  
        #pragma HLS unroll  
        t[i](worker, split1.out[i], merge1.in[i]);  
    }  
  
    write_out(merge1.out, out, n);  
}
```

Example Mixed Model

- Being declared as a task, `worker` runs infinitely
- Data-driven task-level parallelism is closer to conventional RTL development, for instance as used in hardware trigger systems in high energy physics, where continuous streams of data from the experiment have to be processed
- We can separate out this function to create a separate freely running kernel

```
void dut(int in[N], int out[N], int n) {  
    #pragma HLS dataflow  
        hls_thread_local hls::split::round_robin<int, NP>  
split1;  
        hls_thread_local hls::merge::round_robin<int, NP>  
merge1;  
  
    read_in(in, n, split1.in);  
  
    // Task-Channels  
    hls_thread_local hls::task t[NP];  
    for (int i=0; i<NP; i++) {  
        #pragma HLS unroll  
            t[i](worker, split1.out[i], merge1.in[i]);  
    }  
  
    write_out(merge1.out, out, n);  
}
```

Freely running kernel model

- Purely data-driven task-level parallelism
- For instance used in hardware trigger systems in high energy physics, where continuous streams of data from the experiment have to be processed
- No block-level I/O protocol, kernel starts automatically and never stops!
- With dataflow or tasks
→ difference in simulation behavior!

```
extern "C" {  
void krnl_free_running_dataflow(  
    hls::stream<ap_axiu<WORD_LENGTH,0,0,0>>& in,  
    hls::stream<ap_axiu<WORD_LENGTH,0,0,0>>& out)  
{  
    #pragma HLS interface ap_ctrl_none port=return  
  
    hls::stream<ap_uint<WORD_LENGTH>> s1;  
    #pragma HLS stream variable=s1 depth=16  
    hls::stream<ap_uint<WORD_LENGTH>> s2;  
    #pragma HLS stream variable=s2 depth=16  
  
    #pragma HLS dataflow  
    read_df(in, s1);  
    compute_df(s1, s2);  
    write_df(s2, out);  
}  
}
```

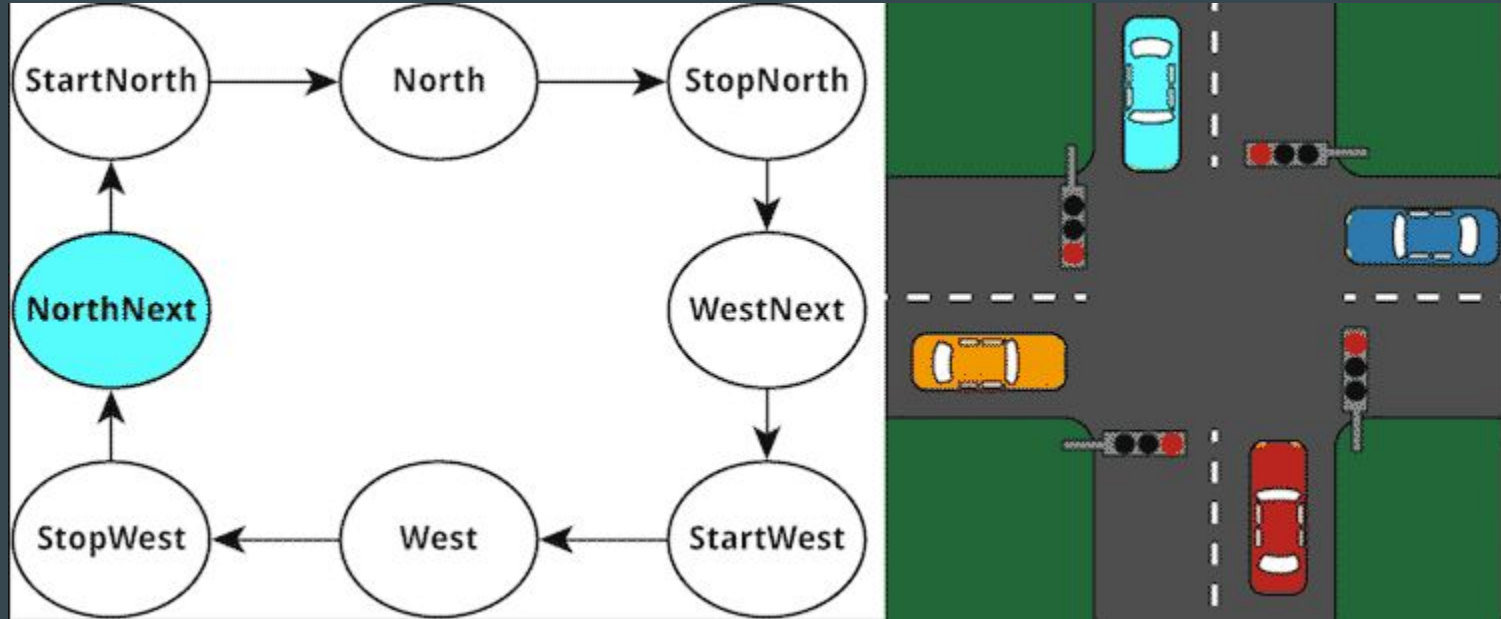
Freely running kernel model

- Purely data-driven task-level parallelism
- For instance used in hardware trigger systems in high energy physics, where continuous streams of data from the experiment have to be processed
- No block-level I/O protocol, kernel starts automatically and never stops!
- With dataflow or tasks
→ difference in simulation behavior!

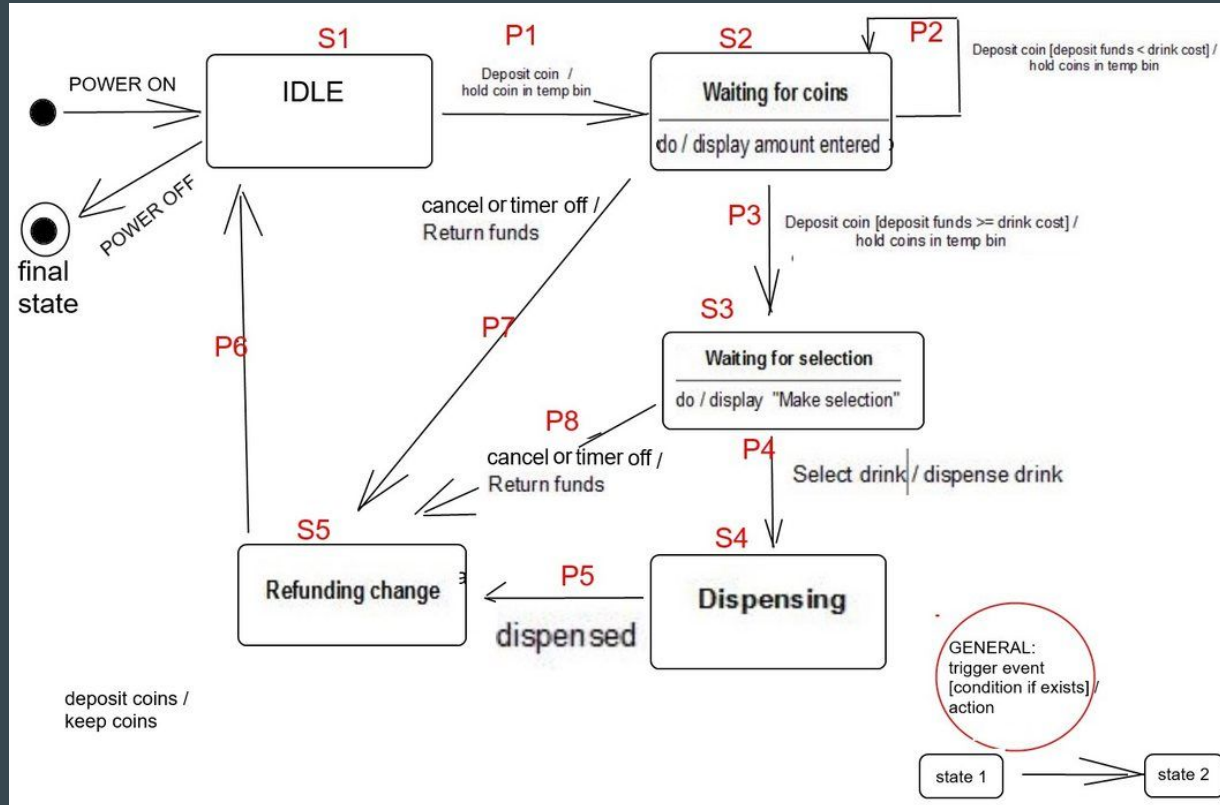
```
extern "C" {  
void krnl_free_running_tasks(  
    hls::stream<ap_axiu<WORD_LENGTH,0,0,0>>& in,  
    hls::stream<ap_axiu<WORD_LENGTH,0,0,0>>& out)  
{  
    #pragma HLS interface ap_ctrl_none port=return  
  
    hls_thread_local hls::stream<ap_uint<WORD_LENGTH>> s1;  
    #pragma HLS stream variable=s1 depth=16  
    hls_thread_local hls::stream<ap_uint<WORD_LENGTH>> s2;  
    #pragma HLS stream variable=s2 depth=16  
  
    hls_thread_local hls::task t1(read, in, s1);  
    hls_thread_local hls::task t2(compute, s1, s2);  
    hls_thread_local hls::task t3(write, s2, out);  
}  
}
```

Finite State Machines

FSMs: control logic for hardware



Transitions can be conditional



FSMs in HLS

```
void input_fsm(streams in and out) {

    enum StateInputFSM : uint8_t {
        WAIT_FOR_HEADER = 1,
        MODULE_HEADER = 2,
        MODULE_DATA = 4,
        READ_FOOTER = 8
    };

    static StateInputFSM state =
StateInputFSM::WAIT_FOR_HEADER;

    // variables ...

    infinite_input_FSM_loop : // Infinite loop
    while (true) {
        #pragma HLS Pipeline II=1 style=flp/
        // non-blocking read from the input stream ...
```

```
// statemachine logic
if(state == StateInputFSM::WAIT_FOR_HEADER) {
    if(in_flag == EVT_HDR_FLAG && in_valid){
        s_header.write(in_data);
        state = StateInputFSM::MODULE_HEADER;
    }
} else if (state == StateInputFSM::MODULE_HEADER) {
    if(in_valid){
        if(in_flag == EVT_FTR_FLAG) {
            ...
            state = StateInputFSM::READ_FOOTER_1;
        } else {
            ...
            state = StateInputFSM::MODULE_DATA;
        }
    }
    ...
```

Missing pipeline targets

Pipeline Dependencies

- Read-after-write (true dependency), instruction depends on the result of a previous operation
- Write-after-read (anti-dependency), instruction cannot update before a previous instruction reads
- Write-after-write, keeping specific order of writing, otherwise other instructions get corrupted

RAW:

I1: $t = a * b;$

I2: $c = t + 1;$

WAR:

I1: $b = t + a;$

I2: $t = 3;$

WAW:

I1: $t = a * b;$

I2: $c = t + 1;$

I3: $t = 1;$

Memory Dependencies

- No issue within same iteration of the loop, but I1 writes to a location that is read by I2 of the next iteration of the loop
→ RAW dependence

```
int top(int a) {  
    int r=1,rnext,m,i,out;  
    static int mem[256];  
L1: for(i=0;i<=254;i++) {  
#pragma HLS PIPELINE II=1  
I1:     m = r * a; mem[i+1] = m;    // line 7  
I2:     rnext = mem[i]; r = rnext; // line 8  
    }  
    return r;  
}
```

WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint

(II = 1, distance = 1) between 'store' operation (top.cpp:7) of variable 'm', top.cpp:7 on array 'mem' and 'load' operation ('rnext', top.cpp:8) on array 'mem'.

INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.

Removing false dependencies

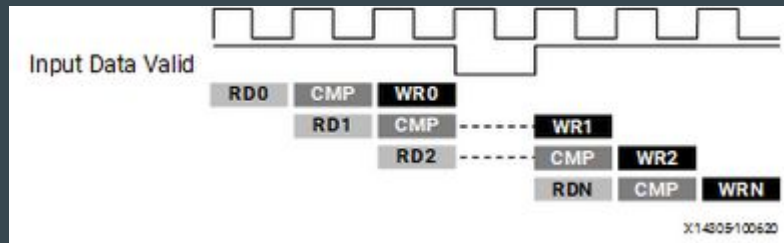
- False dependencies are dependencies that arise when the compiler is too conservative
- Two main types
 - Inter: between different iterations of same loop
 - Intra: within the same iteration of a loop
- Be careful! Specifying a FALSE dependency, when in fact the dependency is not FALSE, can result in incorrect hardware.

```
void histogram(int in[INPUT_SIZE], int hist[VALUE_SIZE]) {  
    int acc = 0;  
    int i, val;  
    int old = in[0];  
    #pragma HLS DEPENDENCE variable=hist type=intra  
    direction=RAW dependent=false  
    for(i = 0; i < INPUT_SIZE; i++)  
    {  
        #pragma HLS PIPELINE II=1  
        val = in[i];  
        if(old == val)  
        {  
            acc = acc + 1;  
        }  
        else  
        {  
            hist[old] = acc;  
            acc = hist[val] + 1;  
        }  
  
        old = val;  
    }  
  
    hist[old] = acc;  
}
```


Timing violations

A common cause of timing violations

- Pipeline types
 - Standard pipelines stall when no input data is available
 - Flushable pipelines continue to produce output, shut down pipeline stages successively
- Requires blocking signals to stall the pipeline → can become drivers of high-fanout nets, especially on pipelines that are deep in the number of physical stages and work on significant data sizes! → prime cause of timing closure issues that cannot be fixed in synthesis and place and route!
- Resolve with free-running pipeline (only within dataflow region)
 - No blocking signals, reduces fanouts, fully flushable
 - Additional resources for FIFO buffers for blocking output ports and mux delay at those ports
- Tool automatically selects right pipeline, but may be overwritten by the user



Target Clock Period Is Too Aggressive

- If the requested clock is faster than the hardware implementation can achieve, the scheduler cannot fit operations within one cycle.
- Example:
 - Target clock: 3 ns
 - A floating-point multiply may require >5 ns
 - Result → timing violation during scheduling.
- Fixes
 - Relax the clock constraint
 - Allow multi-cycle operations
 - Pipeline the design

Long Combinational Paths in One Cycle

- If many operations are chained in a single clock cycle, the delay becomes too large.
- If not pipelined, HLS may schedule all operations in one cycle, creating a long critical path.

```
y = a * b + c * d + e * f + g;
```

- Fixes
 - Use PIPELINE
 - Break expressions into stages
 - Insert intermediate variables

Large Arithmetic Operators

- Some operators have high latency:
- Common heavy operators:
 - Floating-point division
 - Floating-point sqrt
 - 64-bit multiplication
 - modulus (%) and division (/)
- Division is often implemented with multi-cycle hardware, which can cause violations if forced into one cycle.
- Fixes
 - Use fixed-point instead of floating-point
 - Allow multi-cycle operators
 - Use approximations or lookup tables

Unrolled Loops Creating Huge Logic

- Loop unrolling replicates hardware. Excessive unrolling creates large combinational structures.

```
#pragma HLS UNROLL factor=64
for (i=0;i<64;i++)
    sum += a[i] * b[i];
```

- This creates 64 multipliers and a large adder tree, which may exceed the clock timing.
- Fixes
 - Reduce unroll factor
 - Pipeline the loop instead
 - Use partial unrolling

Large Reduction Trees

- Operations like summation across many elements generate adder trees.

```
for (i=0; i<128; i++)  
    sum += a[i];
```

- If fully unrolled → very deep adder tree → long critical path.
- Fixes
 - Pipeline the reduction
 - Use tree balancing
 - Accumulate across cycles

Complex Control Logic

- Large nested conditions or state machines can create slow logic.

```
if (a) {  
    if (b) {  
        if (c) ...
```

- Deep control structures → long condition evaluation paths.
- Fixes
 - Simplify conditions
 - Precompute flags
 - Pipeline the control

Bit-Width Explosion

- Large arbitrary precision types increase combinational delay.

```
ap_int<512> a, b;
```

```
c = a + b;
```

- 512-bit adders are very slow.
- Fixes
 - Reduce bit width
 - Use narrower types where possible

Hands-on session!

Lab – Free running kernel, initiation interval, timing issues

Code for this lab again on [github](#)

Start at /day 5, you will find

- /fix_initiation_intervals: solve issues preventing reaching maximum throughput
→ detailed instructions in README.md
- /free_running_kernel: investigate differences in two implementations
→ detailed instructions in [README.md](#)
- If you did not manage to finish any previous lab, you may use the remaining time to revisit these

That's a wrap!

Wrap-up

- I hope you learned something over the course of this week
- High Level Synthesis has significantly reduced the entry barrier for getting started with FPGA programming
- We have done a deep dive into HLS and optimization techniques, but of course we could not cover everything
- There are cool applications of HLS → Machine learning on FPGAs with [hls4ml](#)

Wrap-up

- If you want to get even more control over the hardware you create on FPGAs
→ I recommend getting started with VHDL or Verilog
- If you are interested in following up on FPGAs → You are welcome to join the “Reconfigurable Embedded Systems” lecture this summer term
- If you are interested in applying your new FPGA programming skills in the field of high energy physics, or in Machine Learning on FPGAs with **hls4ml**, and you are looking for a thesis project
→ get in touch! (dittmeier@physi.uni-heidelberg.de)

Your feedback is very welcome!

<https://forms.gle/LHHXLjZguE9aG83n8>

