Traineeships in Advanced Computing for High Energy Physics (TAC-HEP)

GPU & FPGA module training: Part-2

Week-1: FPGA: Parallelism in program execution

Lecture-2: March 22nd 2023

Varun Sharma

University of Wisconsin – Madison, USA

Correct Time

Next Week onwards:

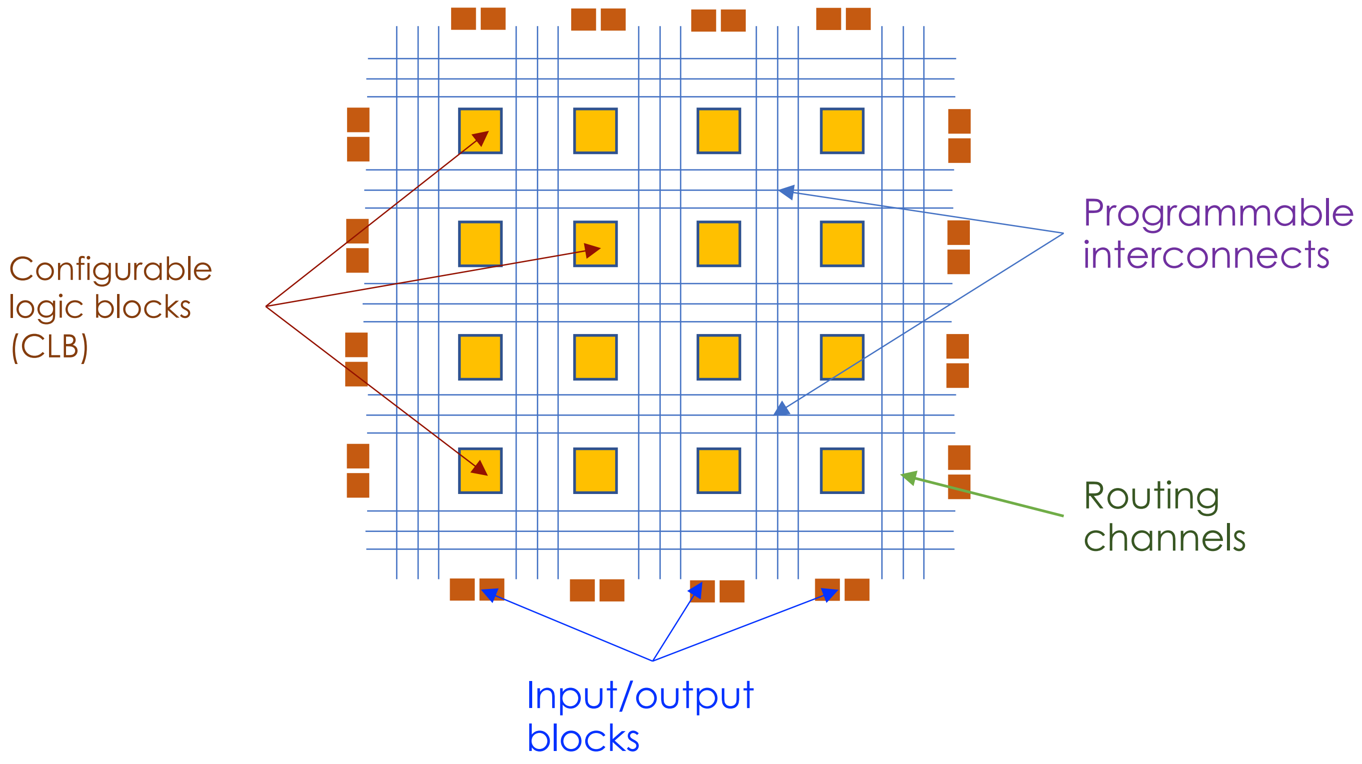
- Tuesdays: 9:00-10:00 CT / 10:00-11:00 ET / 16:00-17:00 CET
- Wednesday: 11:00-12:00 CT / 12:00-13:00 ET / 18:00-19:00 CET

Content

Last lecture we covered introduction to FPGA, its architecture and sub-components

Today we will see how parallelism works in FPGA program execution

Fig. 1



CPU/FPGA Advantages

CPU advantages	FPGA Advantages
• Better with floating point numbers • Programming a CPU in normally easier than programming an FPGA (does not require to understand digital electronics) • Faster compilation • Easier code portability • Lower unit cost	• More flexible processing • More flexible input/output • Parallel processing • Better with multi-clock systems • Better with time-critical operations

More and more often, FPGAs and CPUs (or GPUs) are complementary:
They co-exist in the same system and perform different tasks

FPGA/ASIC Advantages

FPGA Advantages	ASIC Advantages
Faster time-to-market - no layout, masks or other manufacturing steps are needed **Lower constant/initial cost** **Simpler design cycle** - due to software that handles much of the routing, placement, and timing **More predictable project cycle** due to elimination of potential re-spins, wafer capacities, etc. **Re-programmability**: a new configuration can be uploaded	**Full custom capability (including analog) -** since device is manufactured to design specs **Lower unit costs** – For mass production **Smaller form factor** - since device is manufactured to design specs **Higher clock speeds**

Uses of FPGAs outside HEP

- Telecommunication
- Automotive
- Aerospace and Defense
- Medical Electronics
- ASIC Prototyping
- Audio
- Broadcast
- Consumer Electronics
- Data Center
- Distributed Monetary Systems
- High Performance Computing
- Industrial
- Scientific Instruments
- Security systems
- Video & Image Processing
- Digital signal processing
- Bioinformatics
- Controllers
- Computer hardware emulation
- Voice recognition
- Cryptography

TAC-HEP 2023

FPGA Parallelism

Program execution on a Processor

A processor executes a program as a sequence of instructions

- Translated into useful computation for a software application
- Compiler transforms the C/C++ into assemble language

$z = a + b;$ $\rightarrow$ `ADD $R1,$R2,$R3`

- The assemble code defines the addition operation to compute the value of z in terms of the internal registers of a processor
- The complete assembly program to compute the value of z is as follows:

```
LD      a, $R1
LD      b, $R2
ADD     $R1,$R2,$R3
ST      $R3, c
```

- Even a simple operation, such as the addition of two values, results in multiple assembly instructions

Program execution on a Processor

- Depending on the location of a and b, the LD operations take a different number of clock cycles to complete:
  - Processor cache : few 10 clock cycles
  - DDR memory: ~100/~1000 clock cycles
  - Hard drives: even longer

- Software engineers spend a lot of time restructuring their algorithms
  - Increase the spatial locality of data in memory to increase the cache hit rate and decrease the processor time spent per instruction.

Program execution on FPGA

FPGA is an inherently parallel processing fabric capable of implementing any logical and arithmetic function that can run on a processor

- Main difference: **Vivado HLS compiler**
  - Transforms software descriptions into RTL, is not hindered by the restrictions of a cache and a unified memory space
- Computation of z is compiled by Vivado HLS into several LUTs required to achieve the size of the output operand
- E.g.: In C code, variable a, b, and z are defined with the short data type (16-bit data container)
  - Variables gets implemented as 16 LUTs by Vivado HLS

General rule: 1 LUT is equivalent to 1 bit of computation

Program execution on FPGA

- LUTs used for the computation of z are exclusive to this operation ONLY.
  - Unlike a processor, where all computations share the same ALU
- FPGA implementation instantiates independent sets of LUTs for each computation in the software algorithm
- FPGA differs from processor: memory architecture & cost of memory access
- FPGA implementation, the Vivado HLS compiler arranges memories into multiple storage banks as close as possible to the point of use in the operation
  - Results in an instantaneous memory bandwidth, exceeding the capabilities of a processor

Program execution on FPGA

With regard to computational throughput and memory bandwidth, the Vivado HLS compiler exercises the capabilities of the FPGA fabric through the processes:

- Scheduling
- Pipelining
- Dataflow

Transparent to the user, these processes are integral stages of the software compilation process that extract the best possible circuit-level implementation of the software application.

Scheduling

Process of identifying the data and control dependencies between different operations

- To Determine which operation occur during each clock cycle based on:
  - Length of the clock cycle or clock frequency
  - Time it takes for the operation to complete, as defined by the target device
  - User-specified optimization directives

- Vivado HLS analyzes dependencies between adjacent operations as well as across time
- Group operations to execute in the same clock cycle and set up the hardware to allow the overlap of function calls.
- The overlap of function call executions removes the processor restriction that requires the current function call to fully complete before the next function call to the same set of operations can begin.

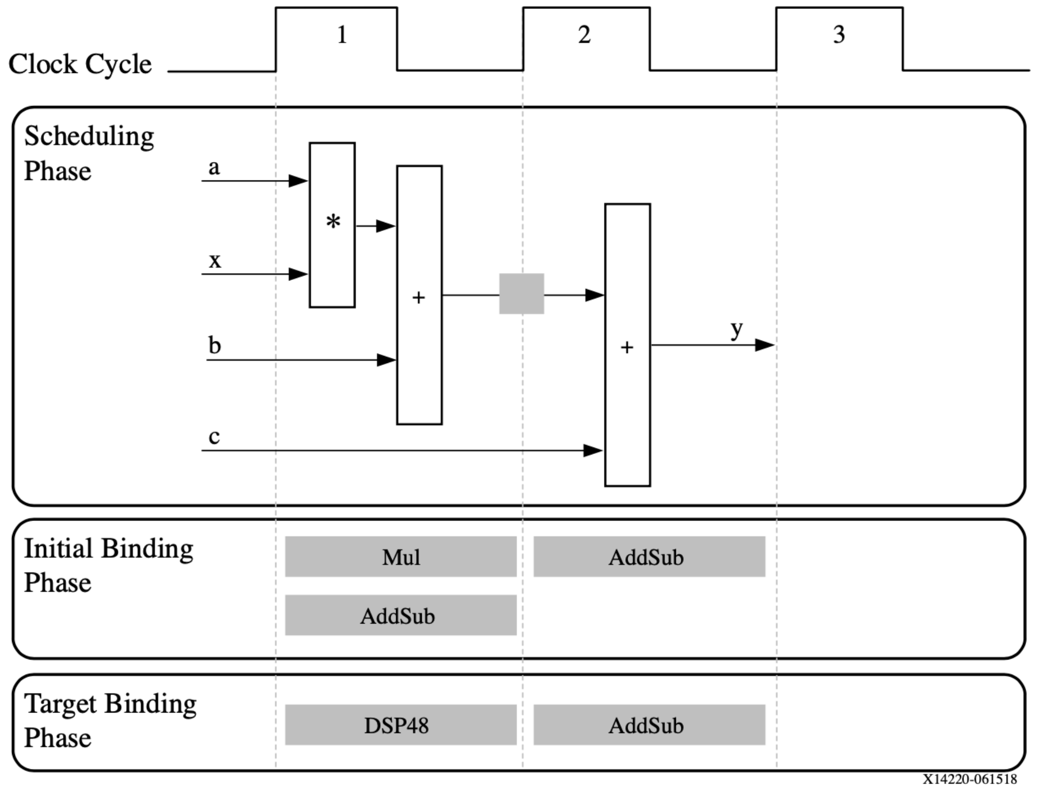
Scheduling

E.g.: Scheduling phases for a simple code

```
int foo(char x, char a, char b, char c) {
 char y;
 y = x*a+b+c;
 return y;
}
```

First cycle: reads *x*, *a*, and *b* data ports
Second cycle: reads data port c & generates output *y*

Fig. 2



Scheduling

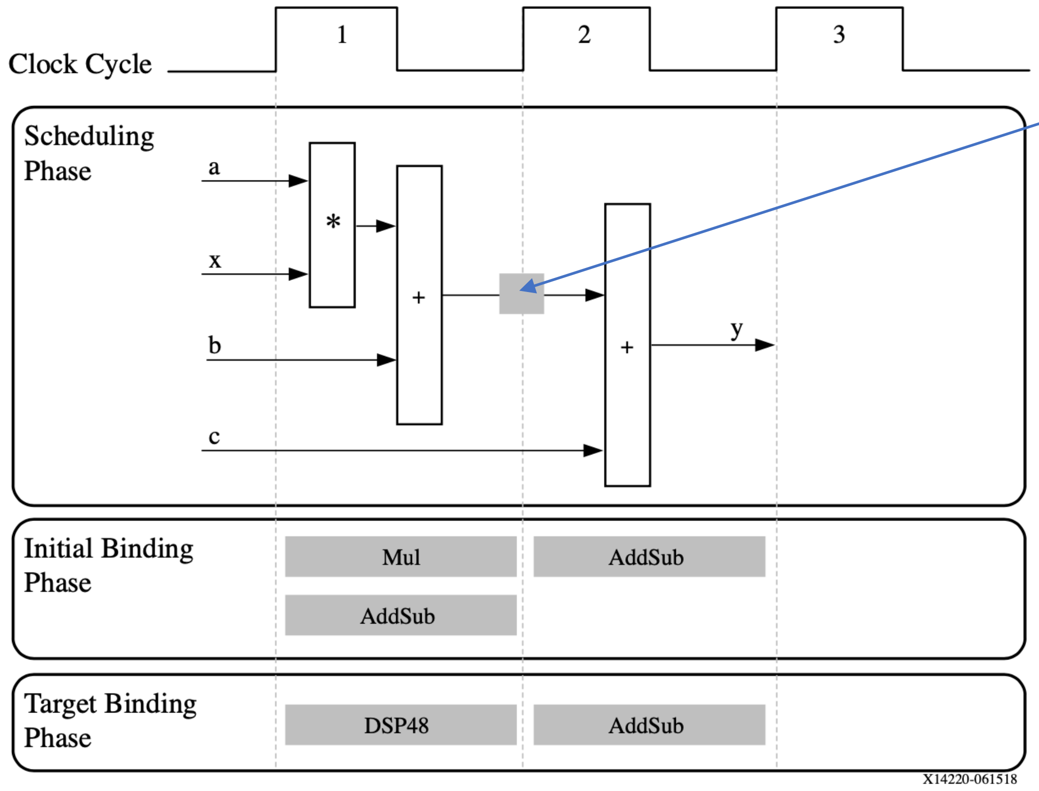
E.g.: Scheduling phases for a simple code

```
int foo(char x, char a, char b, char c) {
 char y;
 y = x*a+b+c;
 return y;
}
```

First cycle: reads x, *a*, and *b* data ports
Second cycle: reads data port c & generates output *y*

Internal register storing a variable

Fig. 3



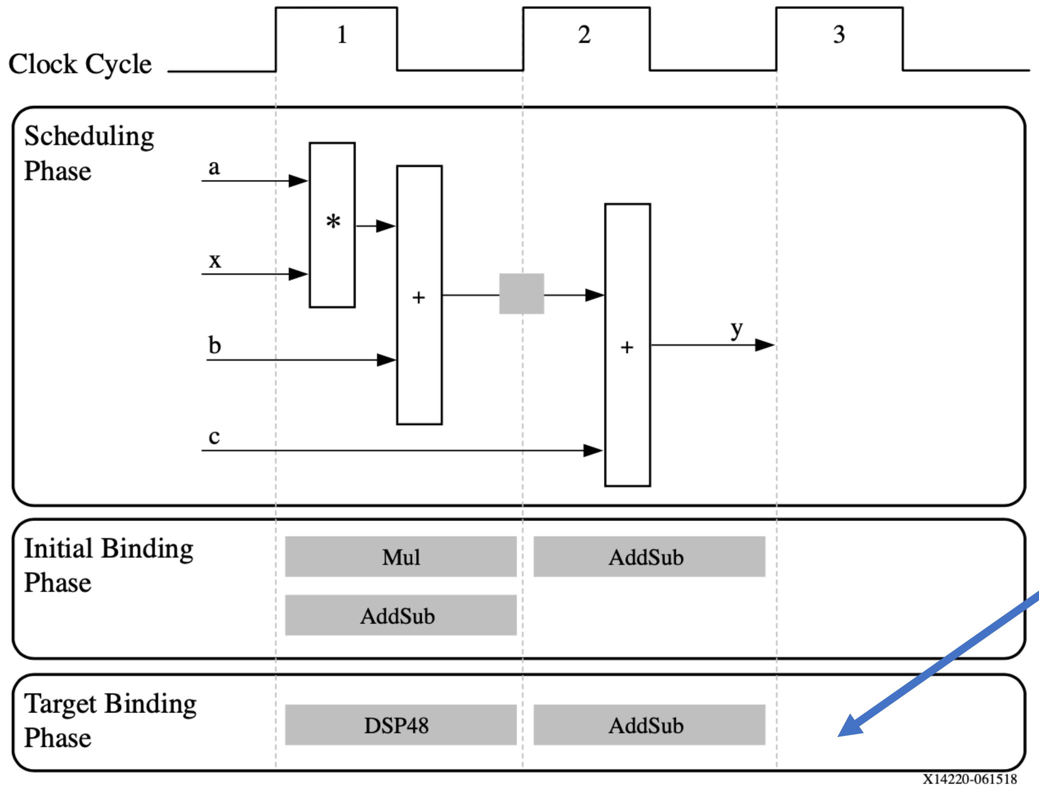
First cycle: Multiplication and the first addition
Second cycle: Second addition and output generation

Scheduling

E.g.: Scheduling phases for a simple code

```
int foo(char x, char a, char b, char c) {
 char y;
 y = x*a+b+c;
 return y;
}
```

Fig. 4



In this example, the arguments are simple data ports but in hardware implementation they are I/O ports.

The input data ports are all **8-bits** wide ***(char)***.

Output data port is **32-bit** wide as function return is a 32-bit ***int*** data type

Optimised for the ideal balance of high-performance and efficient implementation

Pipelining

Technique to avoid data dependencies and increase the level of parallelism

- Preserving the original functionality, required circuit is divided into a chain of independent stages
- All stages in the chain run in parallel on the same clock cycle
- The only difference is the source of data for each stage
- Each stage in the computation receives its data values from the result computed by the preceding stage during the previous clock cycle

$$y = (a \times x) + b + c$$

- Vivado HLS compiler instantiates one multiplier and two adder blocks for above example

Pipelining

C implementation

$$y = (a \times x) + b + c$$

Pipelined implementation

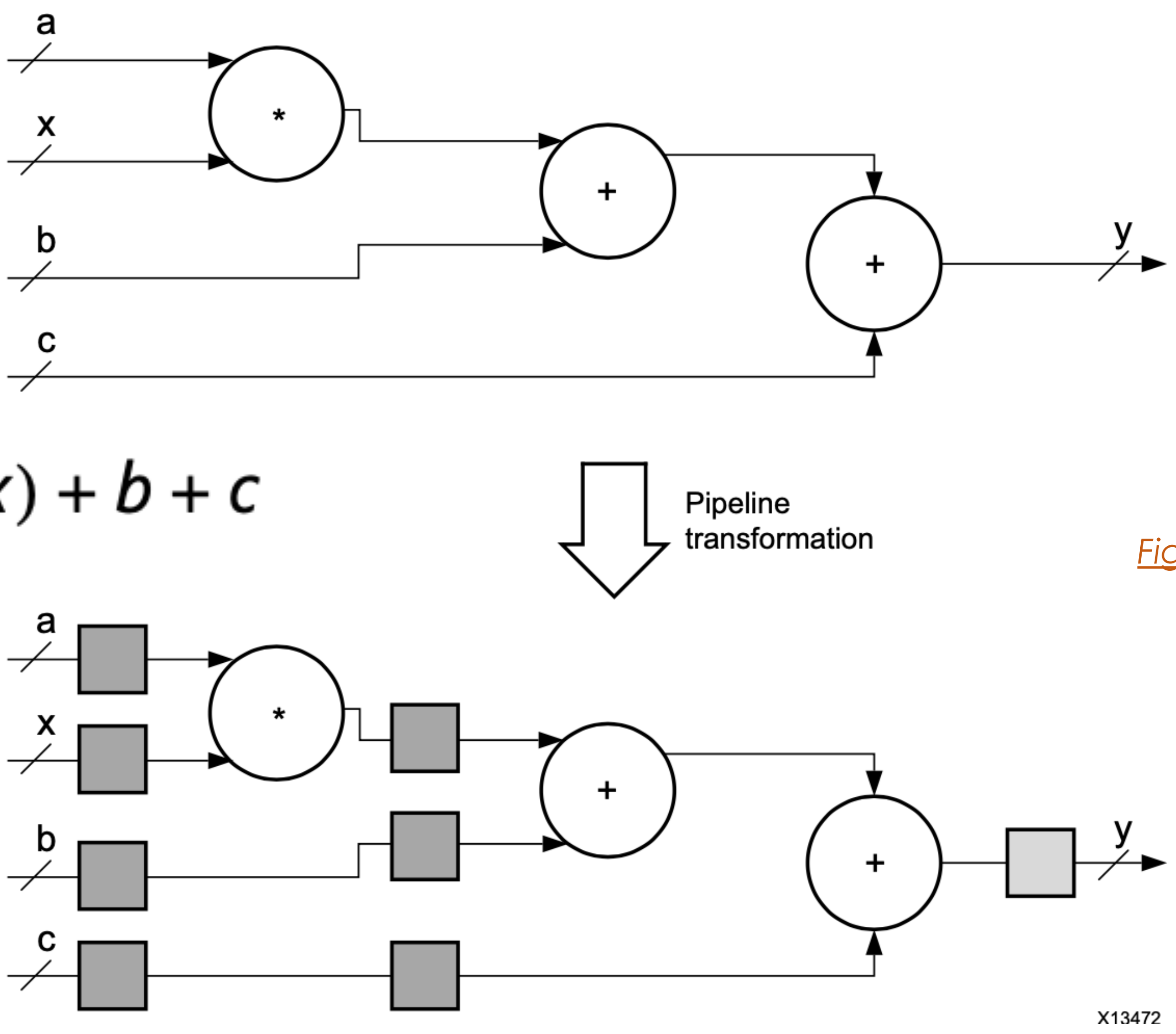


Fig. 5

Pipelining

- Boxes: registers implemented by FF blocks
- Each box column counted as single clock cycle
- Result in 3 clock cycles.
- Addition of registers, leads to separated compute sections for each block
  - Multiplier & two adders can run in parallel and reduce latency

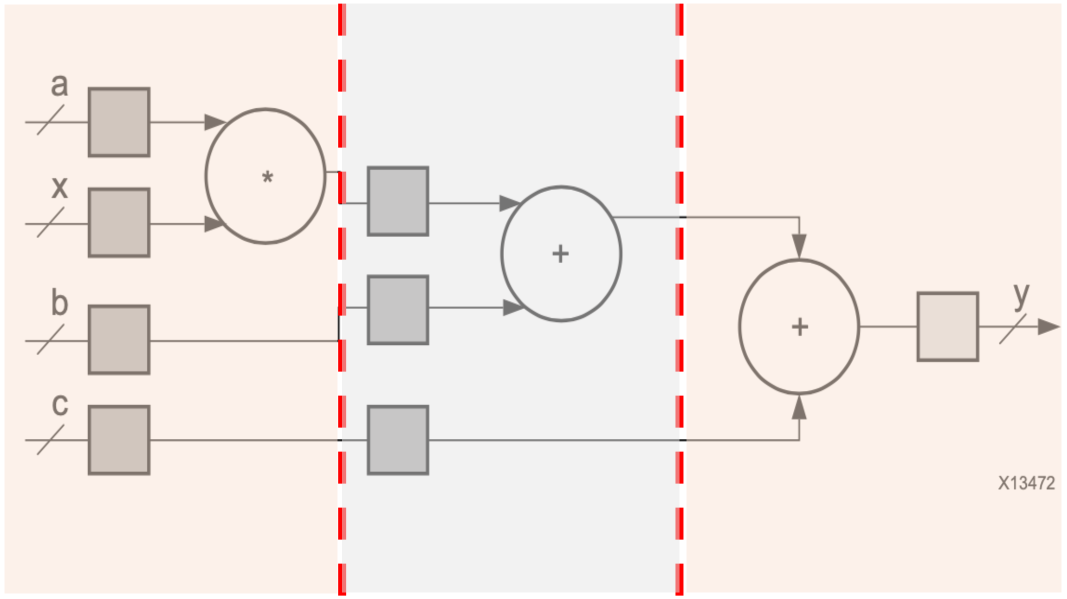


Fig. 6

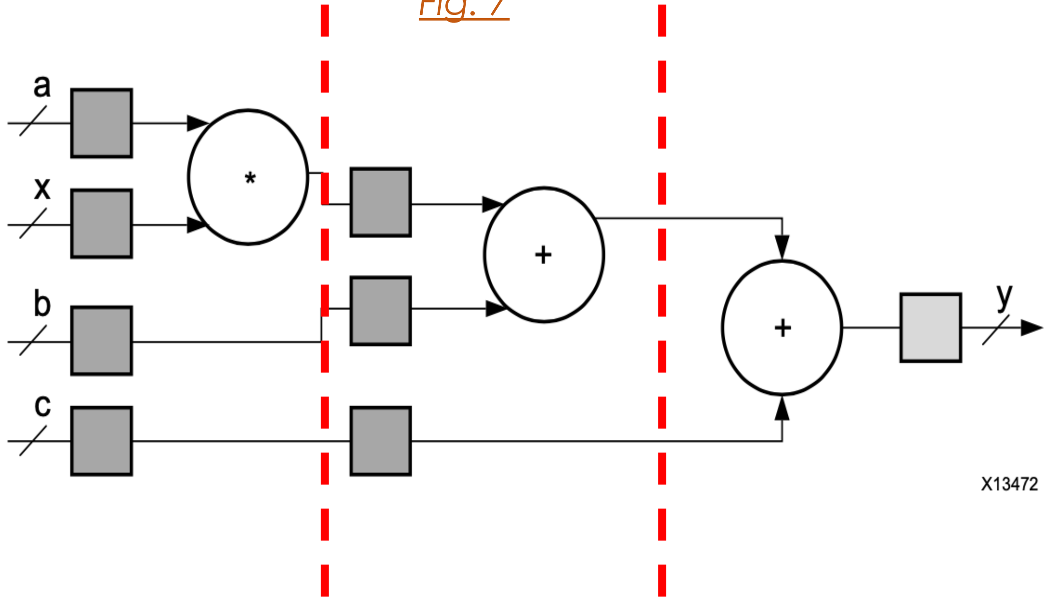
Pipelining

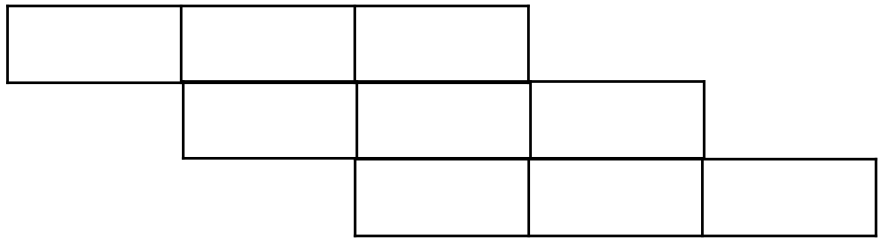
- Both sections of the datapath run in parallel
  - Essentially computing the ***y*** and ***y'*** in parallel
  - ***y'*** result of the next execution

  - First computation of y: pipeline fill time = 3 CLK

  - After this initial computation, a new value of y is available at the output on every clock cycle, because the computation pipeline contains overlapped data sets for the current and subsequent y computations

Fig. 7



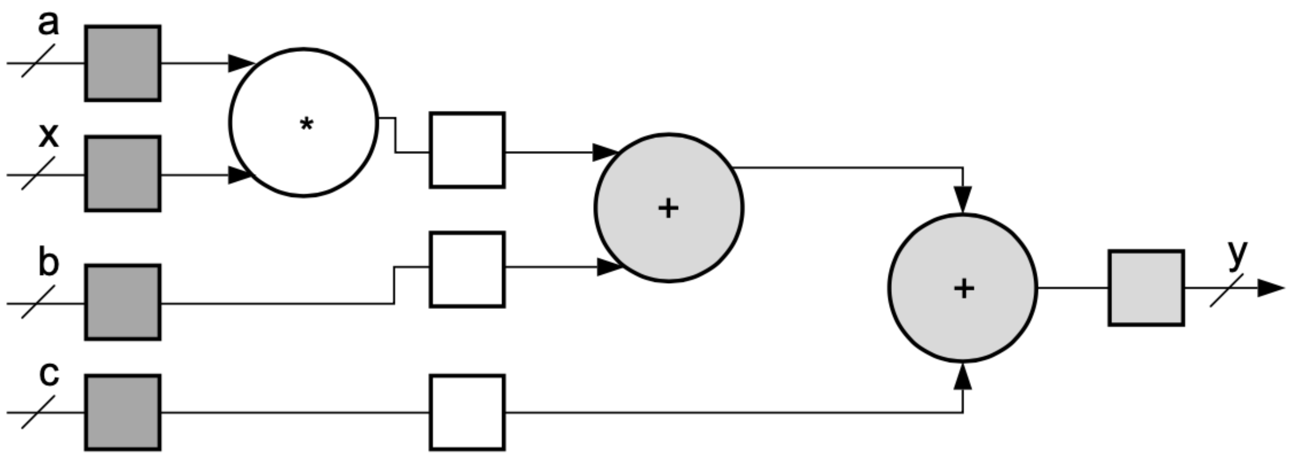


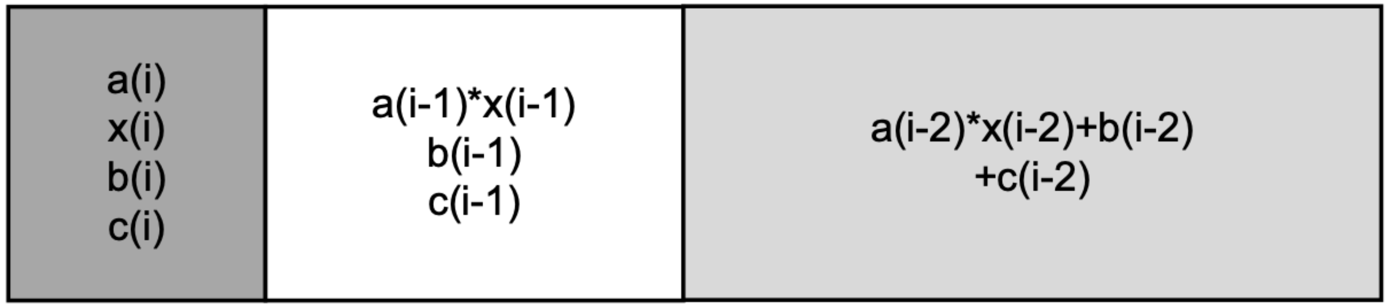
Pipelining

- Raw data: dark gray,
- Semi-computed data: white
- Final data: light gray

All exist simultaneously & each stage result is captured in its own set of registers

Although the latency for such computation is in multiple cycles, there is new result with every cycle





a(i)
x(i)
b(i)
c(i)
	a(i-1)*x(i-1)
	b(i-1)
	c(i-1)
		a(i-2)*x(i-2)+b(i-2)
		+c(i-2)

Fig. 8

Dataflow

- Similar to pipelining but parallelism at coarse-grain level

- Parallel execution of functions within a single program
  - By evaluating the interactions between different functions of a program based on their inputs and outputs

- Case-1: Independent (simplest)
  - Separate resources for different functions and run the blocks independently
- Case-2: Dependent (complex)
  - One function provides result for another function (*consumer-producer scenario*)

Dataflow

Consumer-producer scenario:

- Producer creates a complete data set before the consumer can start its operation
  - Parallelism by instantiating a pair of BRAM memories arranged as memory banks *ping* and *pong*
  - Each function can access only one memory bank, *ping* or *pong*, for the duration of a function call
  - Guarantees **functional correctness** but limits parallelism

- Consumer can start working with partial results from the producer
  - Both functions are connected through the use of a FIFO memory circuit
  - FIFO act as queue, provides data-level synchronization between the modules
  - both hardware modules are executing during any time of functional call
  - **Exception**: consumer module waits for some data to be available from the producer before beginning computation (*Initiation interval – II)*

Path to firmware

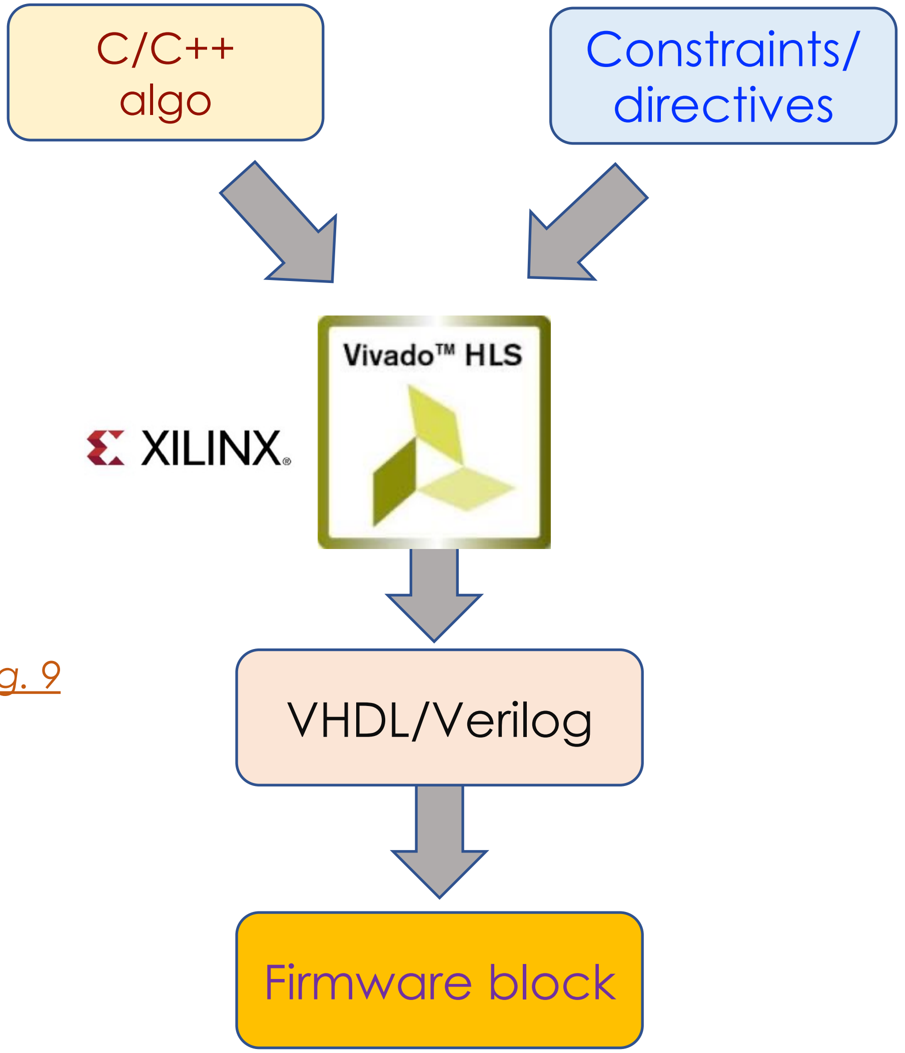


Fig. 9

High Level Synthesis (HLS)

- Compile from C/C++ to VHDL/Verilog
- Pre-processor directives and constraints used to optimize the design

Hardware Description Languages

- VHDL/Verilog
- Programming languages which describe electronic circuits

Drastic decrease in firmware development time!

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf

TAC-HEP 2023

Questions?

TAC-HEP 2023

Additional material

Jargons

- **ICs - Integrated chip:** assembly of hundreds of millions of transistors on a minor chip
- **PCB:** Printed Circuit Board
- **LUT - Look Up Table aka 'logic'** - generic functions on small bitwidth inputs. Combine many to build the algorithm
- **FF - Flip Flops** - control the flow of data with the clock pulse. Used to build the pipeline and achieve high throughput
- **DSP - Digital Signal Processor** - performs multiplication and other arithmetic in the FPGA
- **BRAM - Block RAM** - hardened RAM resource. More efficient memories than using LUTs for more than a few elements
- **PCIe or PCI-E - Peripheral Component Interconnect Express**: is a serial expansion bus standard for connecting a computer to one or more peripheral devices
- **InfiniBand** is a computer networking communications standard used in high-performance computing that features very high throughput and very low latency
- **HLS** - High Level Synthesis - compiler for C, C++, SystemC into FPGA IP cores
- **HDL** - Hardware Description Language - low level language for describing circuits
- **RTL** - Register Transfer Level - the very low level description of the function and connection of logic gates
- **FIFO** – First In First Out memory
- **Latency** - time between starting processing and receiving the result
  - Measured in clock cycles or seconds
- **II - Initiation Interval** - time from accepting first input to accepting next input