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

FPGA module training

Week-3

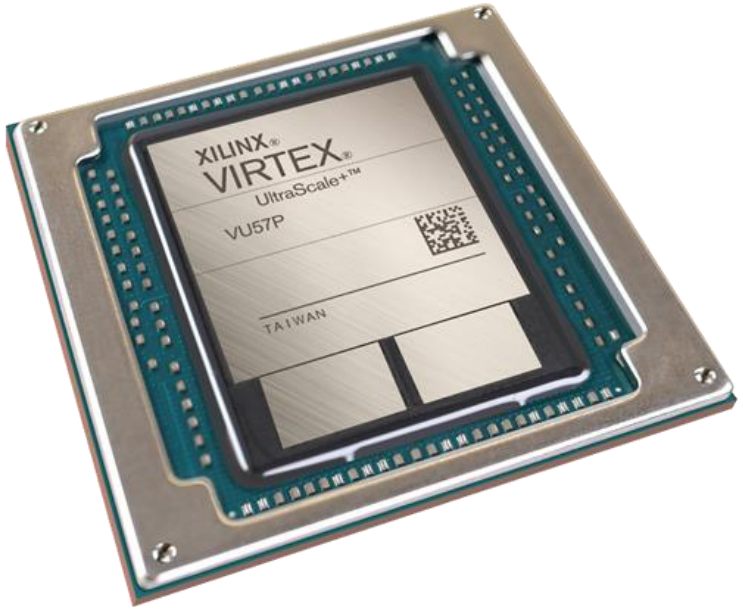
Lecture-5: February 11th 2025

Varun Sharma

University of Wisconsin – Madison, USA

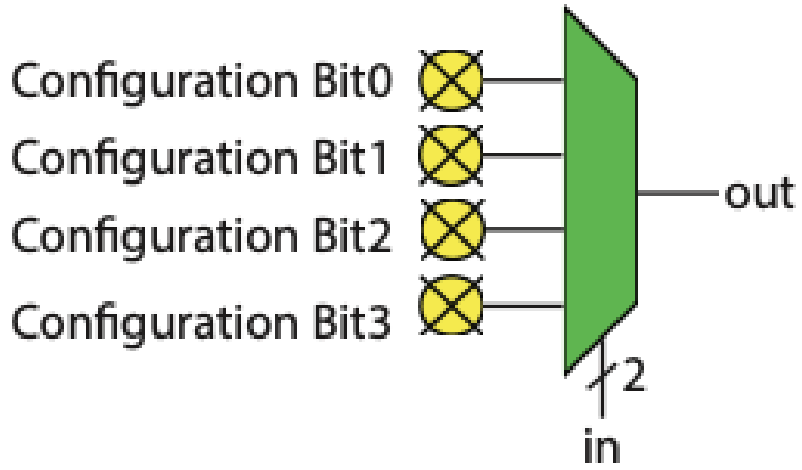
Content

- FPGA Processing
  - Scheduling, Pipelining, DataFlow
- Clock Frequency, Latency, Pipelining



Look-up Tables (LUTs)

a) Lookup Table (LUT)



b)

in[1]	in[0]	out
0	0	0
0	1	0
1	0	0
1	1	1

out = in[1] & in[0]

Configuration bits are memory elements that store the truth table of the function implemented by the LUT (AND in this case)

These bits store the output value for every possible combination of input

Number of configuration bits in a LUT depends on the # of inputs => 2^N

Other Storage Elements: URAM

- Ultra RAM blocks are dual-port, synchronous 288Kb RAM with a fixed configuration
- Available in Xilinx's UltraScale+ devices

- Eight times more storage capacity than the BRAMs
- URAMs generally have higher latency access compared to BRAMs
- **Usage:** Large buffers, Video processing …

TAC-HEP 2025

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

$z = a + b;$ → `ADD $R1,$R2,$R3`

- The assembly 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 (Register-Transfer level),
  - 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

Execution steps on FPGA

- Vivado HLS compiler exercises the capabilities of the FPGA fabric using following processes:
  - Scheduling and binding
  - 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**

- 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 -- ***Pipelining***

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

- If the clock period is longer or a faster FPGA is targeted, more operations are completed within a single clock cycle, and all operations might complete in one clock cycle.

- Conversely, if the clock period is shorter or a slower FPGA is targeted, high-level synthesis automatically schedules the operations over more clock cycles, and some operations might need to be implemented as multicycle resources

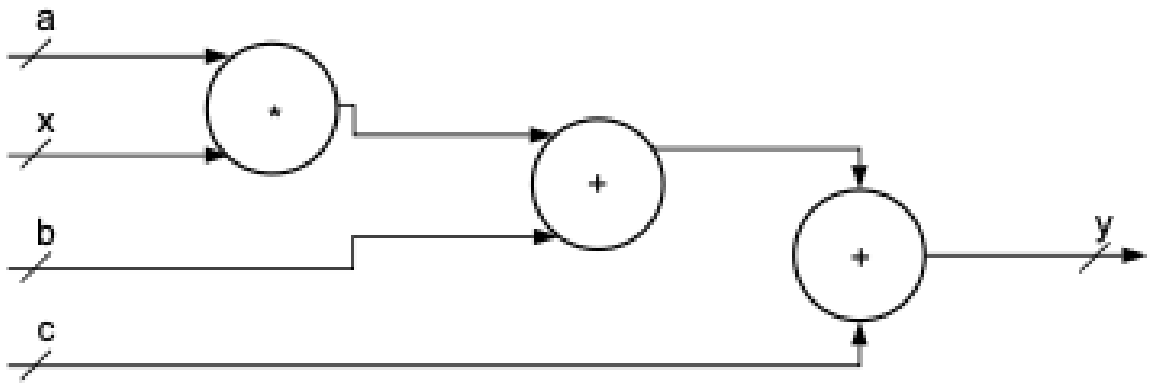
Binding

- Determines which hardware resource implements each scheduled operation

- To implement the optimal solution, high-level synthesis uses information about the target device

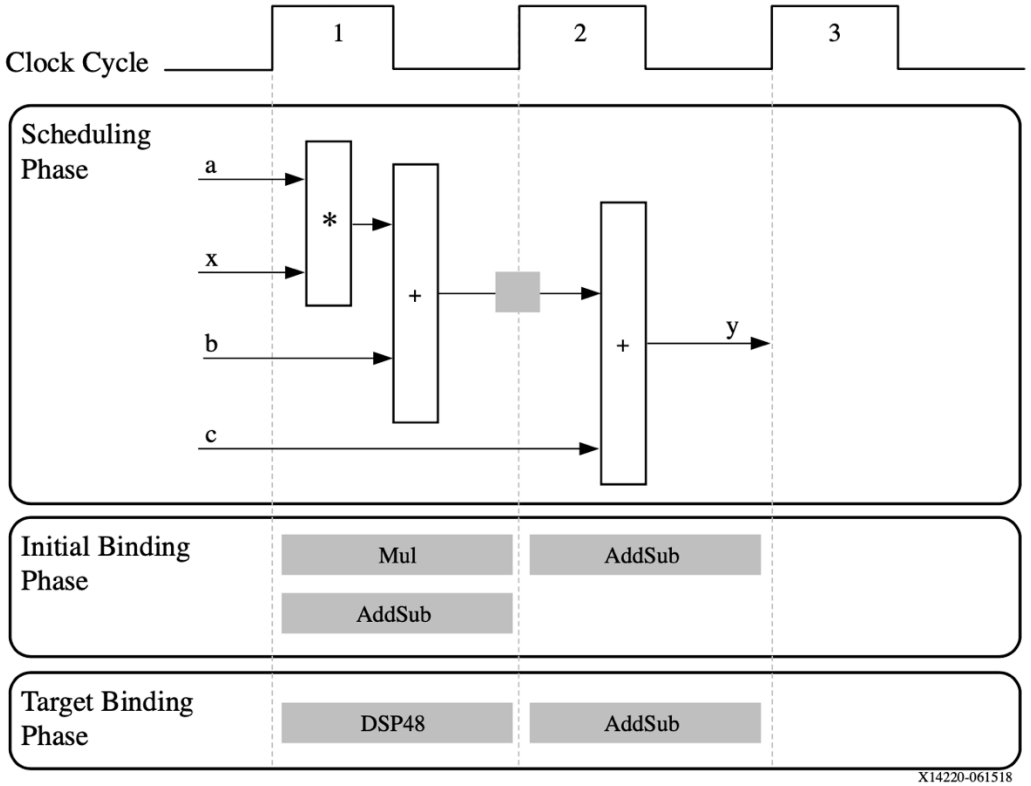
$$y = (a \times x) + b + c$$

Example



Scheduling & Binding

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



Scheduling

First cycle:

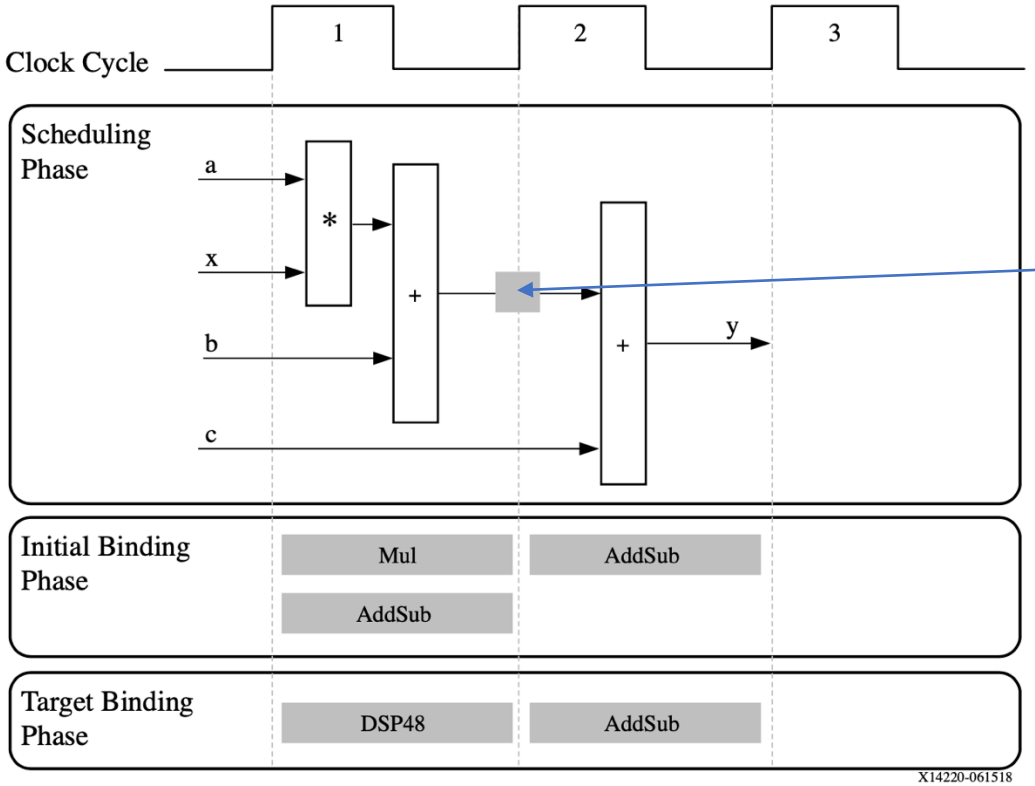
- Reads ***b***, ***a***, and ***b*** data ports

Second cycle:

- Reads data port ***c***
- Generates output ***y***

Scheduling & Binding

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



Scheduling

First cycle:

- Reads ***b***, ***a***, and ***b*** data ports

Second cycle:

- Reads data port ***c***
- Generates output ***y***

Internal register storing a variable

Binding

First cycle:

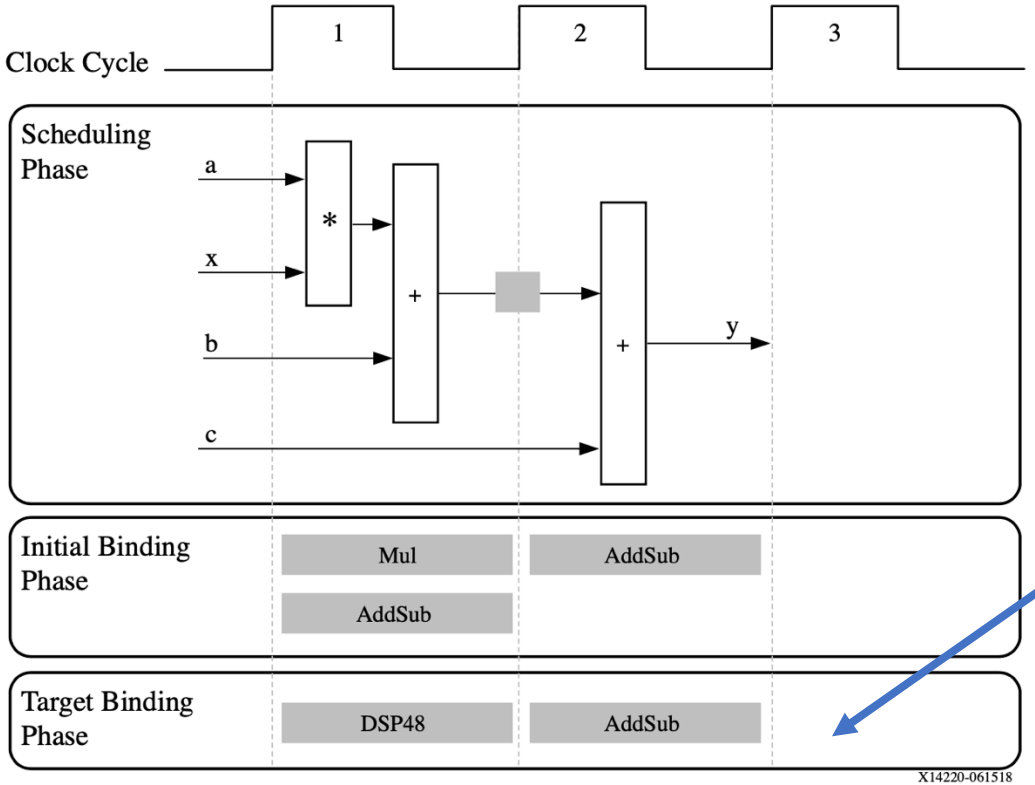
Multiplication & the first addition

Second cycle:

Second addition & output generation

Scheduling

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



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

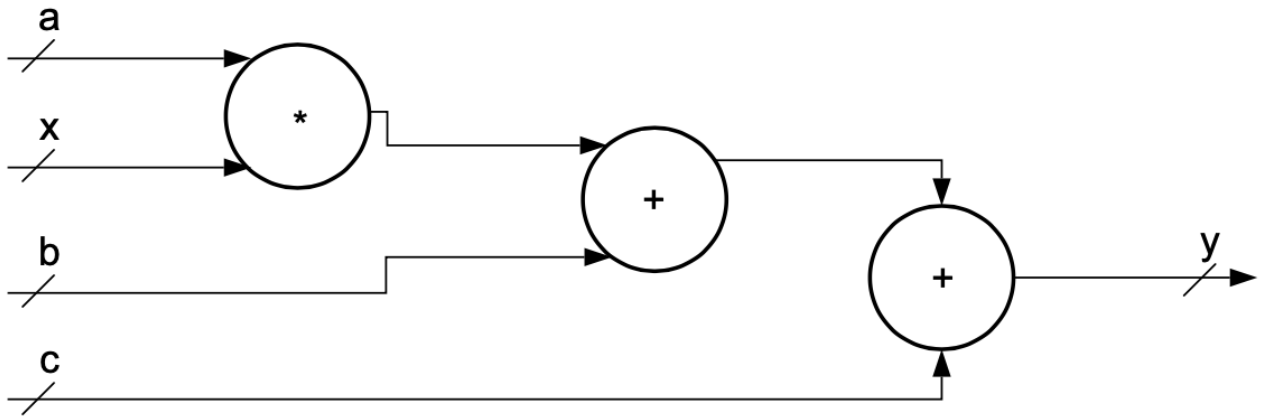
- Preserving the original functionality, required circuit is divided into a chain of independent stages
- All stages in the chain run in parallel on the same clock cycle
- The only difference is the source of data for each stage
- Each stage in the computation receives its data values from the result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

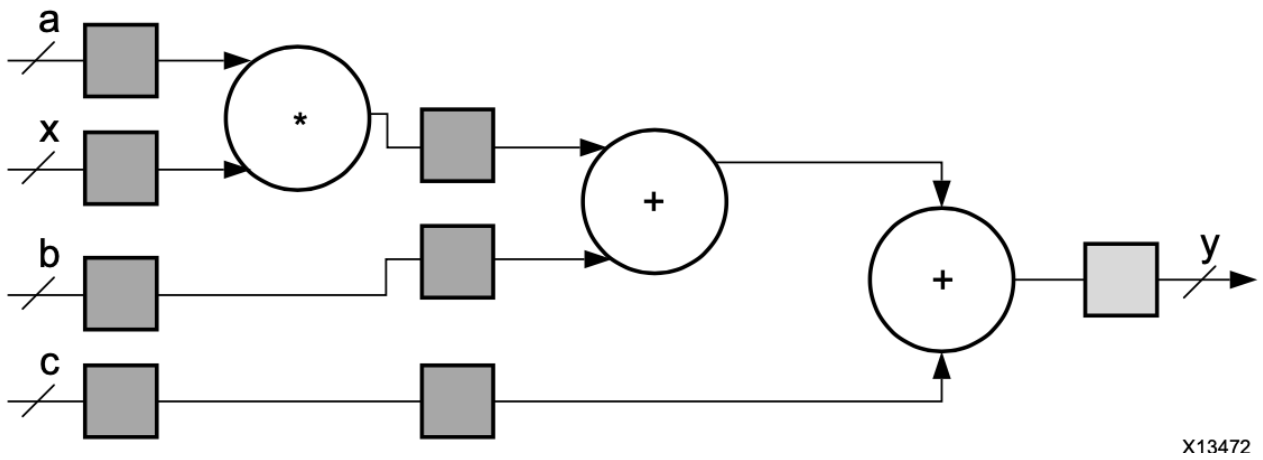
C implementation



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

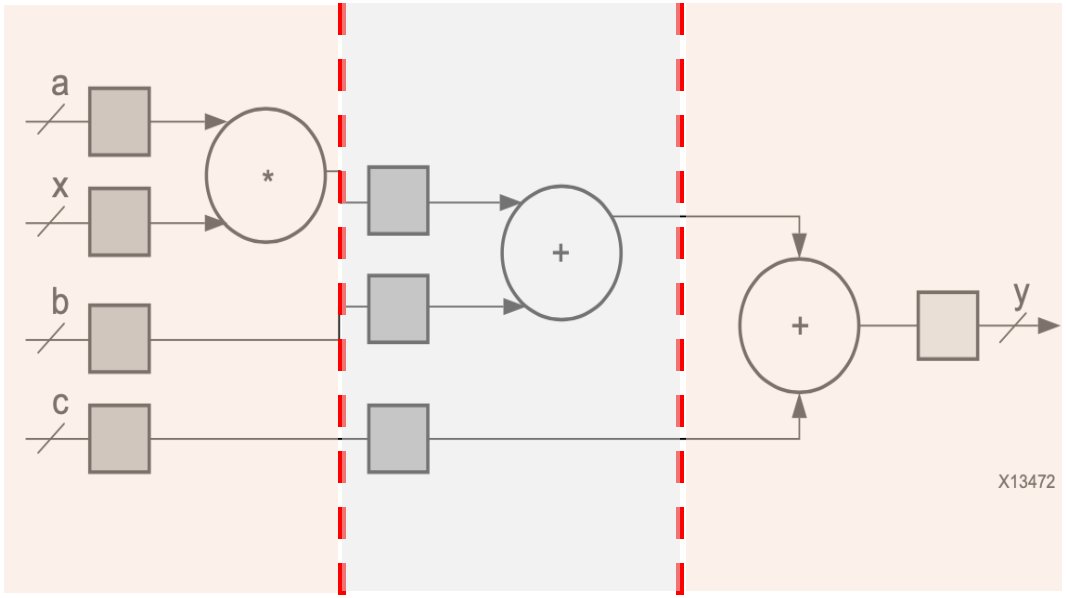
Pipeline transformation

Pipelined implementation



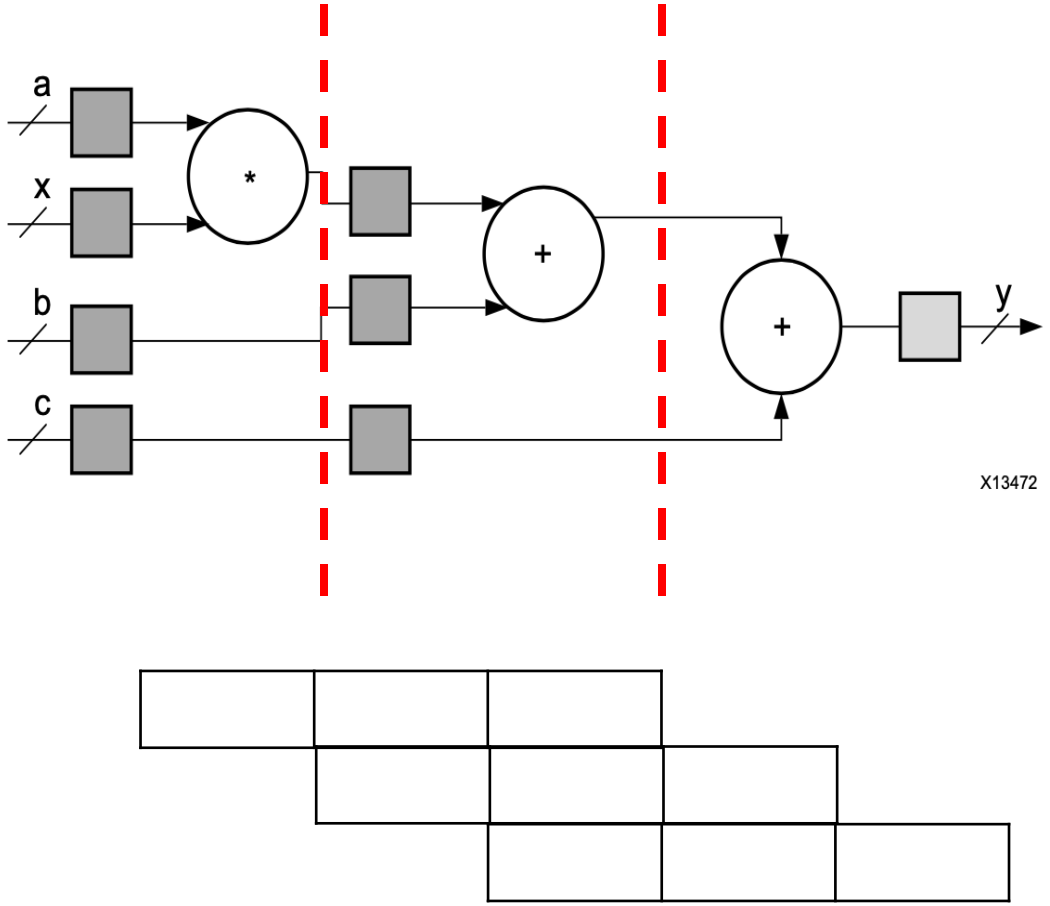
Pipelining

- 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



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

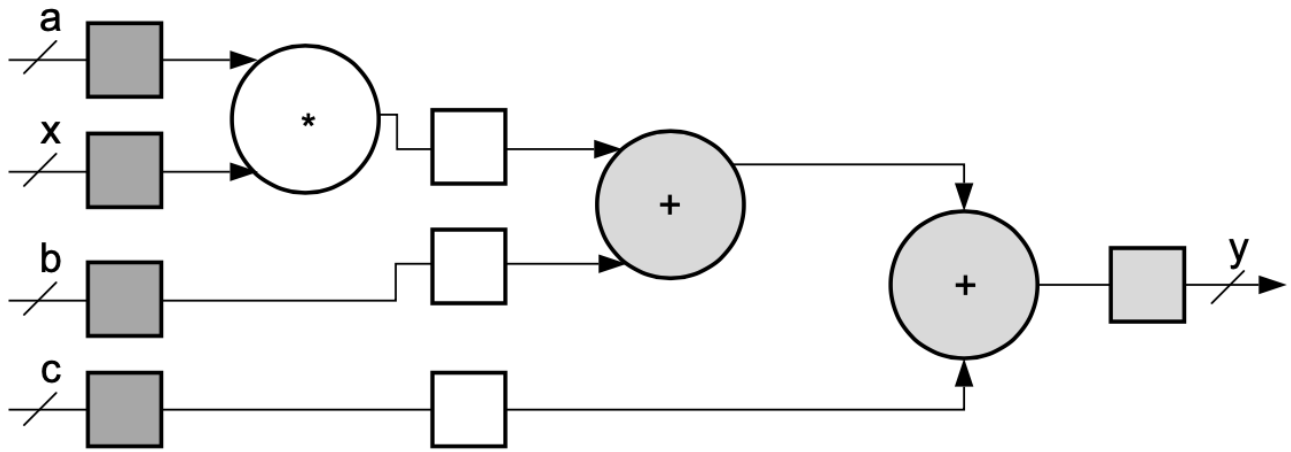


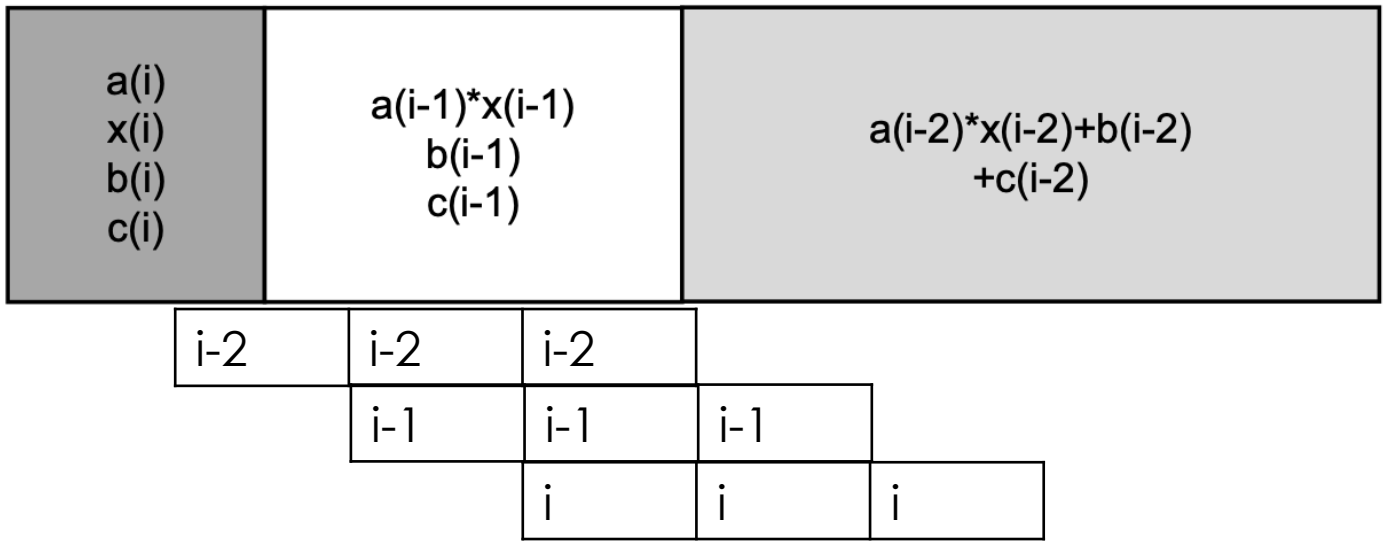
Pipelining

- 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





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:

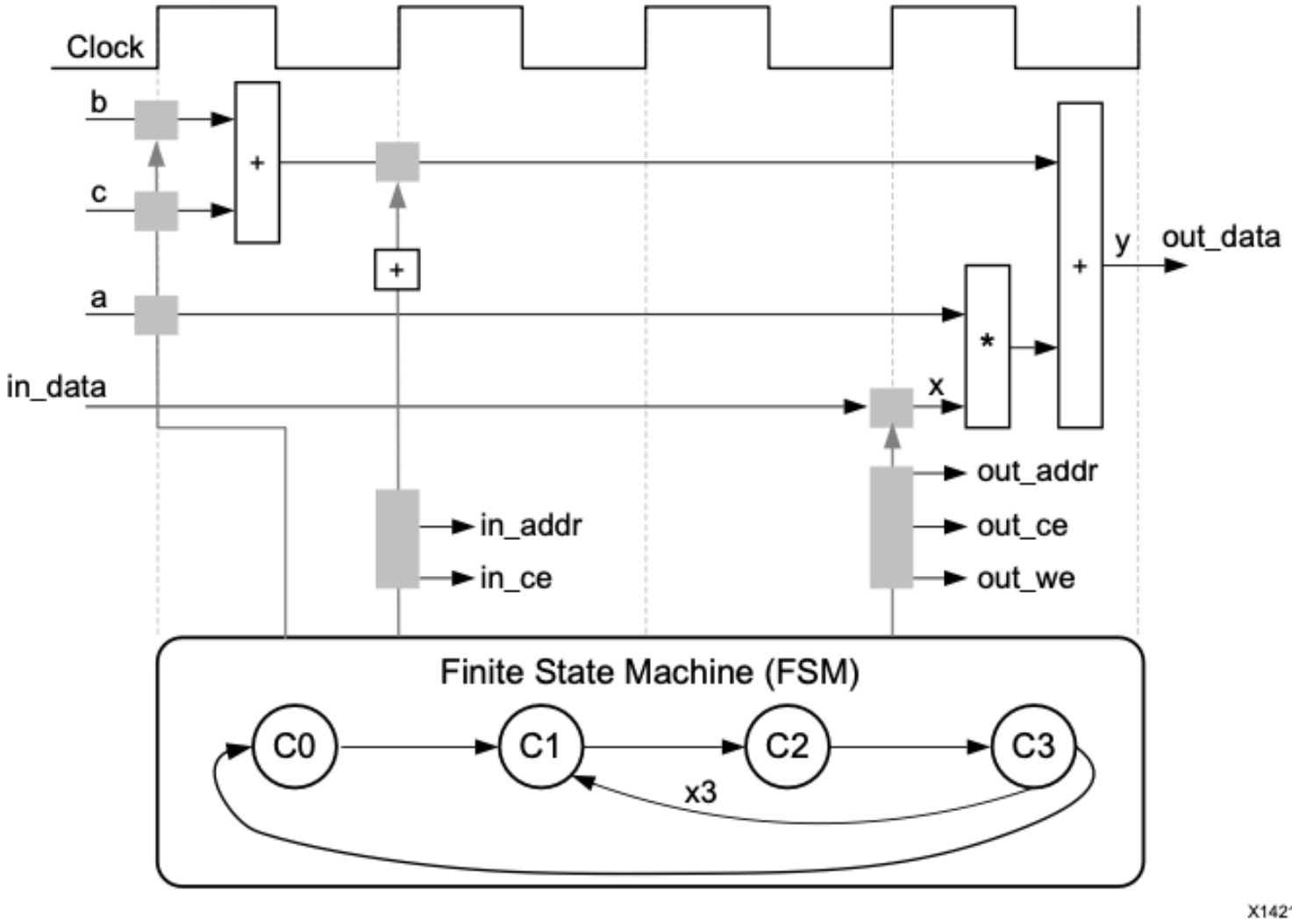
- Producer creates a complete data set before the consumer can start its operation
  - Parallelism by instantiating a pair of BRAM memories arranged as memory banks *ping* and *pong*
  - Each function can access only one memory bank, *ping* or *pong*, for the duration of a function call
  - Guarantees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

Another Example

```
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;
  }
}
```

Another Example: Implementation

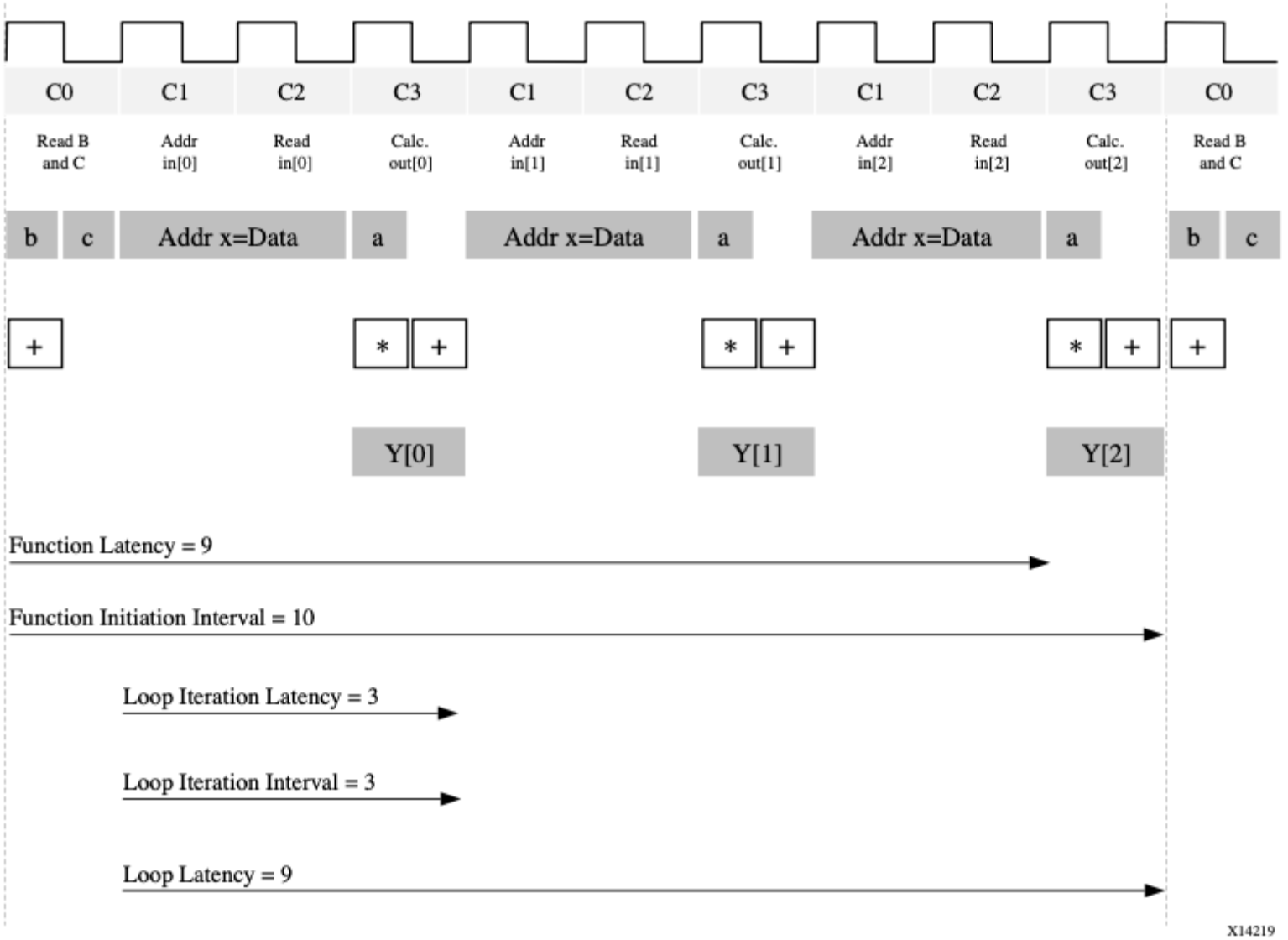


The FSM starts in the state C0

On the next clock, it enters state C1, then state C2, and then state C3. It returns to state C1 (and C2, C3)

A total of three times before returning to state C0

Latency & Initiation Interval



TAC-HEP 2025

Questions?

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