

# HLS-Based Design Using FPGAs

Mahmut Taylan Kandemir

CSE 531

ACK: V. Narayanan, Z. Li

# From Design to Physical Layout

- Circuits Schematic
  - Specify the layout of every wire and transistor
  - Everything manually, basically drawing circuits
- Register-Transfer Level (RTL)
  - HDL (Hardware Descriptive Language)
    - VHDL, Verilog, SystemVerilog
  - Some abstraction
  - Build large design without knowing too much about detail
- High Level Synthesis (HLS)
  - C, C++
  - More abstraction
  - No register, cycle-to-cycle specification
  - Focus on the function, the tool creates the RTL

# HLS

- HLS analyzes and exploits the concurrency in an algorithm.
- HLS inserts registers as necessary to limit critical paths and achieve a desired clock frequency.
- HLS generates control logic that directs the data path.
- HLS implements interfaces to connect to the rest of the system.
- HLS maps data onto storage elements to balance resource usage and bandwidth.
- HLS maps computation onto logic elements performing user specified and automatic optimizations to achieve the most efficient implementation.

```
void example(int A[50], int B[50]) {  
    //Set the HLS native interface types  
    #pragma HLS INTERFACE axis port=A  
    #pragma HLS INTERFACE axis port=B  
    int i;  
    for(i = 0; i < 50; i++){  
        B[i] = A[i] + 5;  
    }  
}
```

# HLS C to gates

- Arithmetic Operators
  - Unary Minus (Negation)  $-a$
  - Addition (Sum)  $a + b$
  - Subtraction (Difference)  $a - b$
  - Multiplication (Product)  $a * b$
  - Division (Quotient)  $a / b$
  - Modulus (Remainder)  $a \% b$

# HLS C to gates

- Bitwise Operators
  - Bitwise Left Shift  $a \ll b$
  - Bitwise Right Shift  $a \gg b$
  - Bitwise One's Complement  $\sim a$
  - Bitwise AND  $a \& b$
  - Bitwise OR  $a | b$
  - Bitwise XOR  $a \wedge b$

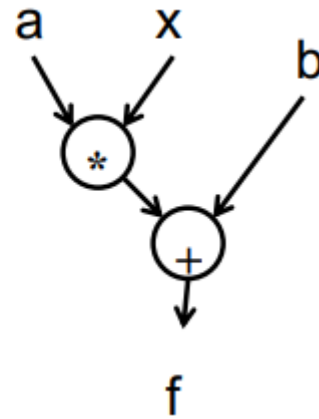
# HLS C to gates

- Comparison Operators
  - Less Than  $a < b$
  - Less Than or Equal To  $a \leq b$
  - Greater Than  $a > b$
  - Greater Than or Equal To  $a \geq b$
  - Not Equal To  $a \neq b$
  - Equal To  $a == b$
  - Logical Negation  $!a$
  - Logical AND  $a \&\& b$
  - Logical OR  $a || b$

# HLS C to gates

- Straight line code

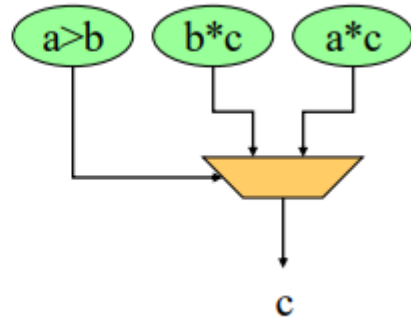
$f = a * x + b$



# HLS C to gates

- Conditions

```
if (a>b)  
    c=b*c;  
else  
    c=a*c;
```






# HLS C to gates

- Function Calls

```
int f(int a, int b)
    return(sqrt(a*a+b*b));
```

```
for(i=0;i<MAX;i++)
    D[i]=f(A[i],B[i]);
```

```
int f(int a, int b)
    return(sqrt(a*a+b*b));
for(i=0;i<MAX;i++)
    D[i]=f(A[i],B[i]);
```

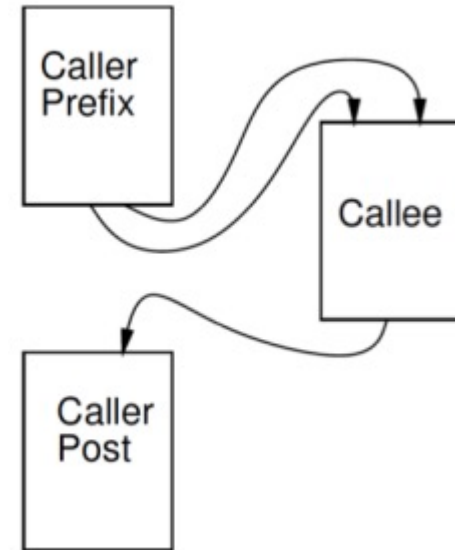


```
for(i=0;i<MAX;i++)
    D[i]=sqrt(A[i]*A[i]
              +B[i]*B[i]);
```

- **Inlinening** a function
  - Copying the body of function into the point of call
  - Replacing the function arguments with the arguments supplied in the call

# HLS C to gates

- Implement a given function as an operation
- Send arguments as input tokens
- Get result back as token



# FIR example

```
#define NUM_TAPS 4

void fir(int input, int *output, int taps[NUM_TAPS])
{
    static int delay_line[NUM_TAPS] = {};

    int result = 0;
    for (int i = NUM_TAPS - 1; i > 0; i--) {
        delay_line[i] = delay_line[i - 1];
    }
    delay_line[0] = input;

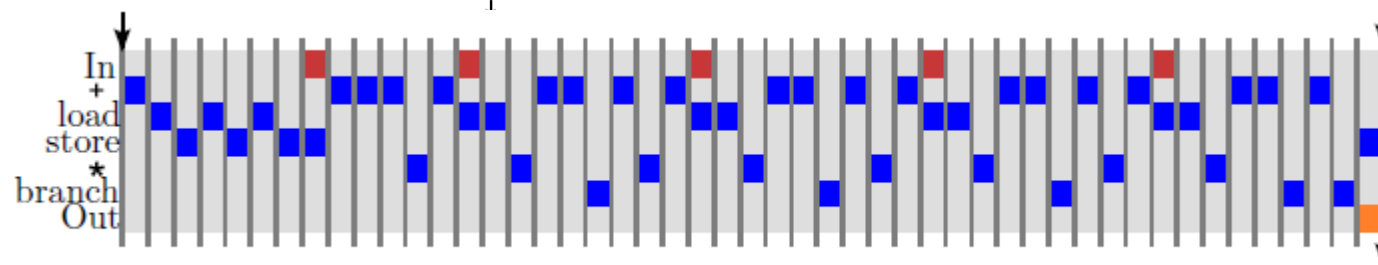
    for (int i = 0; i < NUM_TAPS; i++) {
        result += delay_line[i] * taps[i];
    }

    *output = result;
}
```

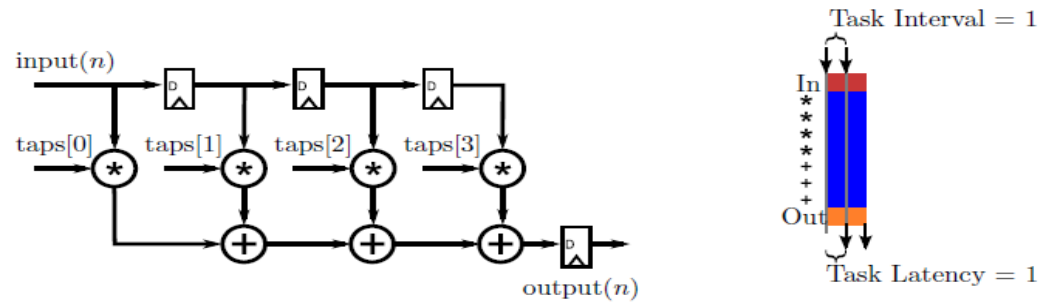
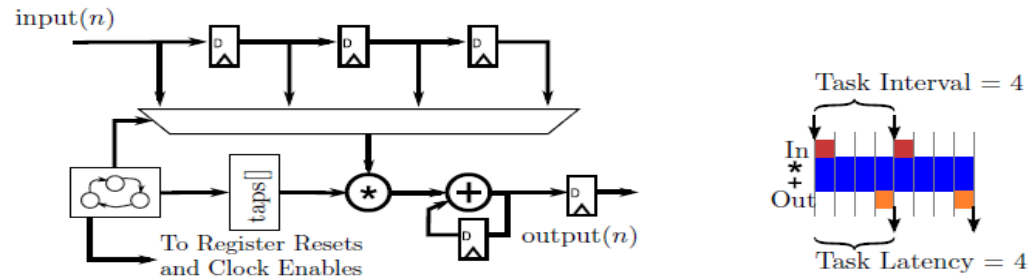
# CPU execution

```
fir:
.frame r1,0,r15 # vars= 0, regs= 0, args= 0
.mask 0x00000000
addik r3,r0,delay_line.1450
lwi r4,r3,8 # Unrolled loop to shift the delay line
swi r4,r3,12
lwi r4,r3,4
swi r4,r3,8
lwi r4,r3,0
swi r4,r3,4
swi r5,r3,0 # Store the new input sample into the delay line
addik r5,r0,4 # Initialize the loop counter
addk r8,r0,r0 # Initialize accumulator to zero
addk r4,r8,r0 # Initialize index expression to zero
$L2:
mul r3,r4,4 # Compute a byte offset into the delay_line array
addik r9,r3,delay_line.1450
lw r3,r3,r7 # Load filter tap
lwi r9,r9,0 # Load value from delay line
mul r3,r3,r9 # Filter Multiply
addk r8,r8,r3 # Filter Accumulate
addik r5,r5,-1 # update the loop counter
bneid r5,$L2
addik r4,r4,1 # branch delay slot, update index expression

rtsd r15, 8
swi r8,r6,0 # branch delay slot, store the output
.end fir
```



# HLS synthesis

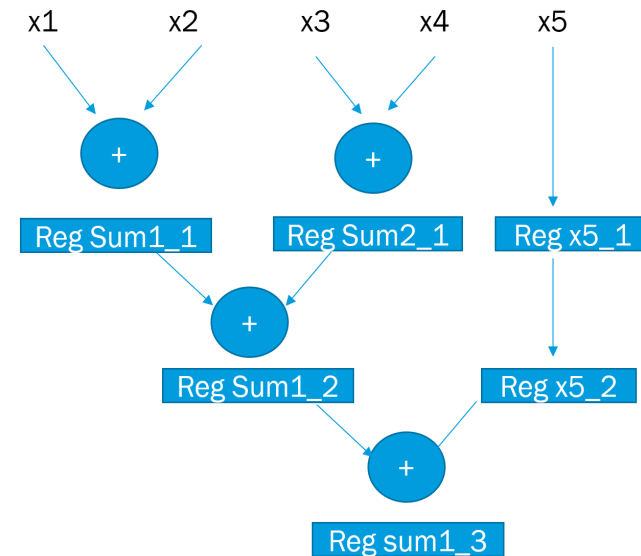


# Exploiting Resource-Level Parallelism

- FPGA potential
  - High raw computation capacity
    - Lots of CLBs for general logic
    - Many DSPs for arithmetic operation
    - Interconnects and memory elements (registers, distributed RAMs, etc)
- How to exploit computation through HLS C approach?
  - Utilize computation resources *as much as possible*
  - Exploit **parallelism** and **regularity** in the code
  - Use pre-build modules *as much as possible*

# Parallelism Overview

- Computation Graph (Dataflow)
  - Program as a graph of operations
  - How data(signal) is produced and consumed
- Where's the parallelism?
  - All operations show in graph run concurrently, the data flow into different operations cycle by cycle



# Parallelism Overview (HLS perspective)

- Control flow
  - Program as a sequence of instructions
    - A sequence of C statements, up to bottom, line by line
    - Compiled to a sequence of assembly code
  - Using a cycle table to describe when an instruction is executed in which cycle
  - Where's the parallelism?





# Parallelism Overview

- **Data-Level**

- – Perform same computation on different data items


- **Thread/Task-Level**

- – Perform separable (perhaps heterogeneous)

- **Instruction-Level**

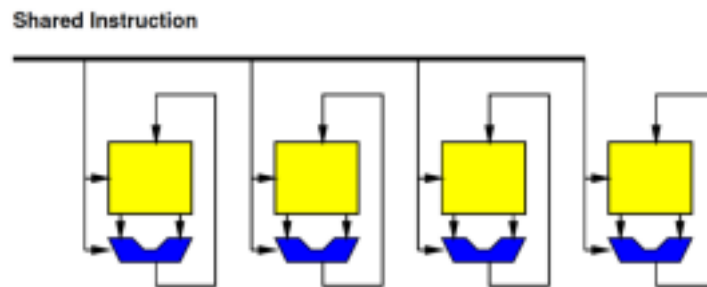
- - Within a single sequential thread/task, perform multiple operations on each cycle

- **Pipeline**

- - Introduce new inputs before finishing
    - - can be instruction/task level
- 

# Data-Level Parallelism

- We often call it SIMD (single instruction multiple data)
- Microcontroller CPU implementation
  - CPU vendor specify in its ISA ( Instruction Set Architecture)
    - Intel AVX instructions
  - Compiler(gcc) compile your C program to AVX instructions, CPU executes in its special hardware unit
  - CPU implementation usually has limited data width and ALU operations
    - 64 bit, add/mult operation, limited ALUs (vector lanes)



```
vadd(int *a, int *b, int *res,  
     int vector_length) {  
    for (int i=0;i<vector_length;i++)  
        res[i]=a[i]+b[i];  
}
```

# Data-Level Parallelism

- A typical example
- May not match physical hardware length
  - $\text{Vector\_length} < \text{vector lanes}$
  - $\text{Vector\_length} > \text{vector lanes}$
  - $\text{Vector\_length} \% \text{vector lanes} > 0$

```
vadd(int *a, int *b, int *res,  
     int vector_length) {  
    for (int i=0; i<vector_length; i++)  
        res[i]=a[i]+b[i];  
}
```

# Data-Level Parallelism

- Instead, an FPGA can be configured to do any # of vector lanes/operations/bitwidth.
  - It's a programmable chip!
- Designer's job is to exploit such parallelism in HLS C code
  - That is, the parallelism will not come naturally
  - It depends on
    - How the code is written
    - Whether the input parameters are fixed or not



# Data-Level Parallelism

- HLS pragmas for data level parallelism
  - Pragma **HLS unroll**
  - Unrolls loops to create multiple independent operations rather than a single collection of operations.
  - The UNROLL pragma transforms loops by creating multiples copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel.



# Data-Level Parallelism

Loop unrolling by a factor of 2 effectively transforms the code to look like the following code where the break construct is used to ensure the functionality remains the same, and the loop exits at the appropriate point.

Fully unroll is usually used for data level parallelism

```
for(int i = 0; i < X; i++) {  
    pragma HLS unroll factor=2  
    a[i] = b[i] + c[i];  
}
```



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

# Data-Level Parallelism

- What would the code look like when it is fully unrolled?



# Memory and Array

- Array in HLS
  - Arrays are typically implemented as a memory (RAM, ROM or FIFO) after synthesis.
  - Details in
    - <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Arrays>
- Array accesses can often create bottlenecks to performance. When implemented as a memory, the number of memory ports limits access to the data.
- Some care must be taken to ensure arrays that only require read accesses are implemented as ROMs in the RTL.
- Need to support parallel access as well



# Dual Port Memory

- Single Port memory has one read and write port
  - Allows one read/write per cycle
- Dual Port memory has two read and write port
  - Allows two read/write per cycle

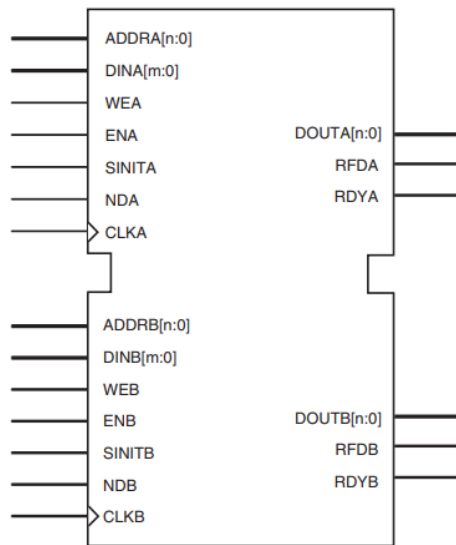
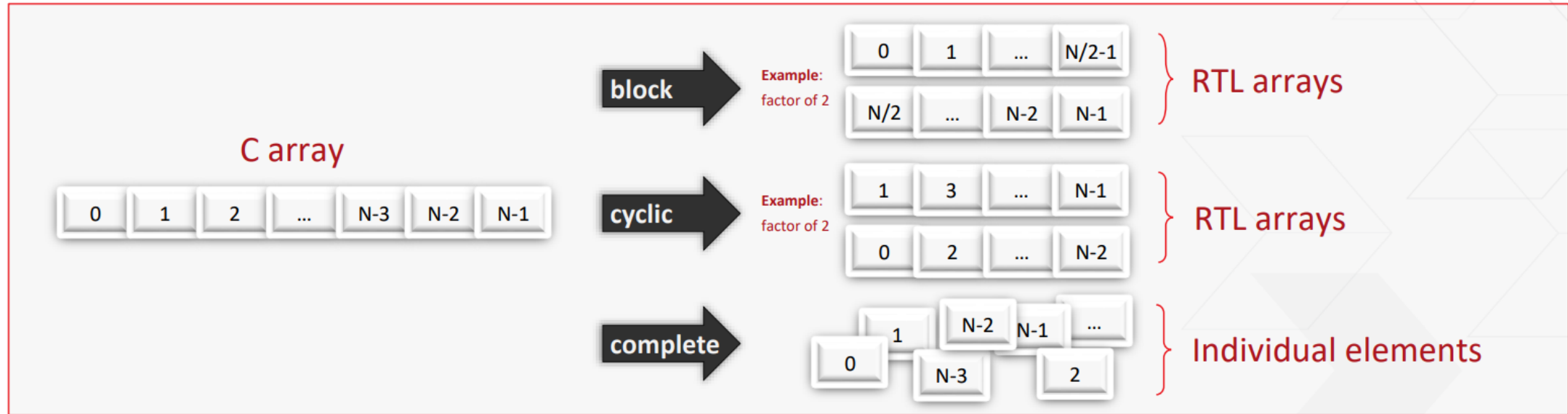


Figure 1: Core Schematic Symbol

# Array Partition

- Partitions an array into smaller arrays or individual elements and provides the following:
  - Results in RTL with multiple small memories or multiple registers instead of one large memory.
  - Effectively increases the amount of read and write ports for the storage.
  - Potentially improves the throughput of the design.
  - Requires more memory instances or registers.

# Array Partition



# Take away

- All kinds of parallelism require parallel access to memory
- Depending on the access pattern, array structures needs to be modified accordingly

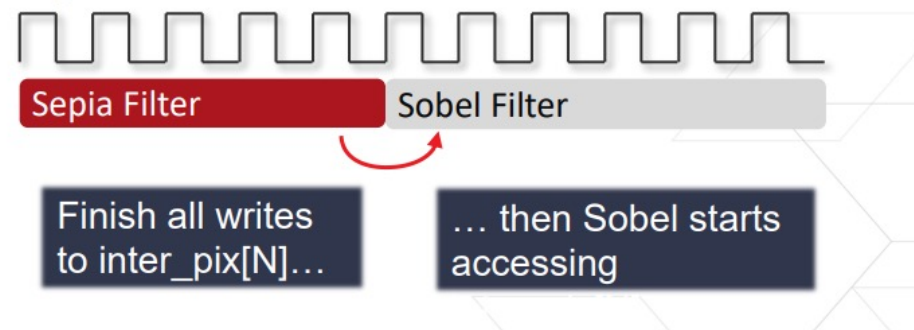


# Task-Level Parallelism

```
// This memory can be a FIFO during optimization  
rgb_pixel inter_pix[MAX_HEIGHT][MAX_WIDTH];  
  
// Primary processing functions  
sepia_filter(in_pix, inter_pix);  
sobel_filter(inter_pix, out_pix2);
```

Sepia Filter

Sobel Filter



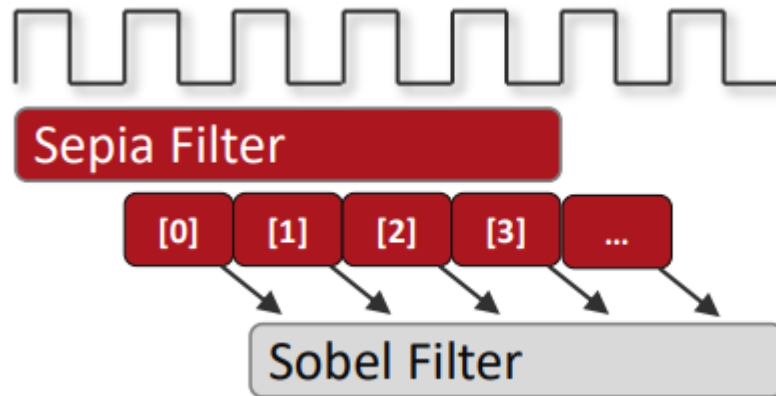
# Task-Level Parallelism

- We want Sobel to start *as soon as* data is ready
  - Functions operate *concurrently* and *continuously*
  - The interval (hence throughput) is improved
  - Channel buffer(FIFO) has to be filled before consumed

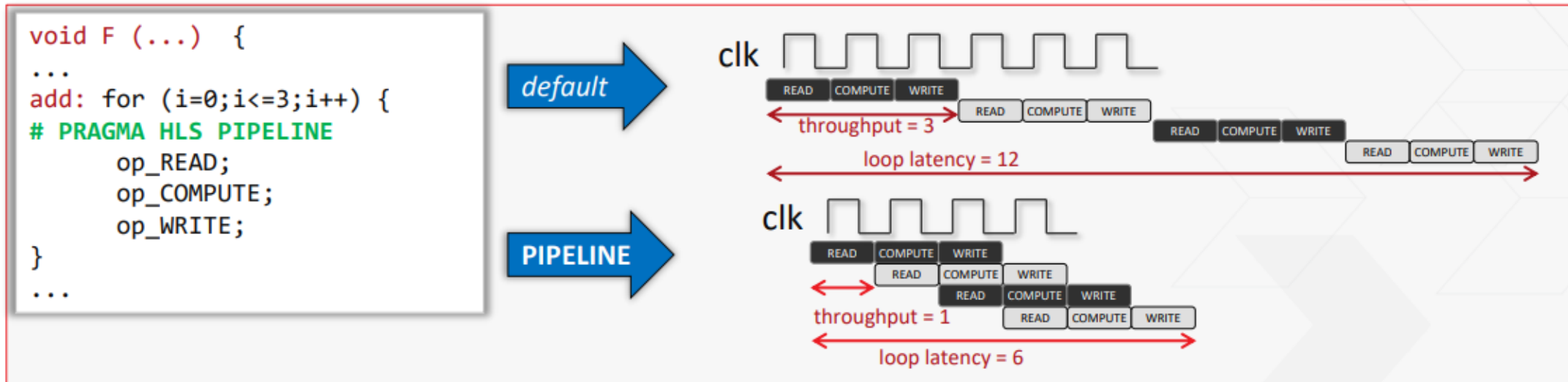


# Task-Level Parallelism

- DATAFLOW allows concurrent execution of two (or more) functions
- Dataflow creates memory channels



# Instruction-Level Parallelism



#pragma HLS PIPELINE II=2

II=Iteration Interval

Latency=II\*#of iterations+(latency of single loop - II)

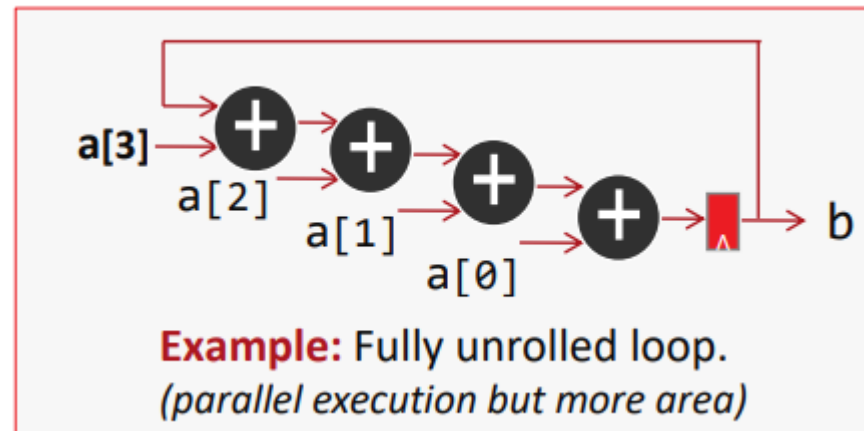
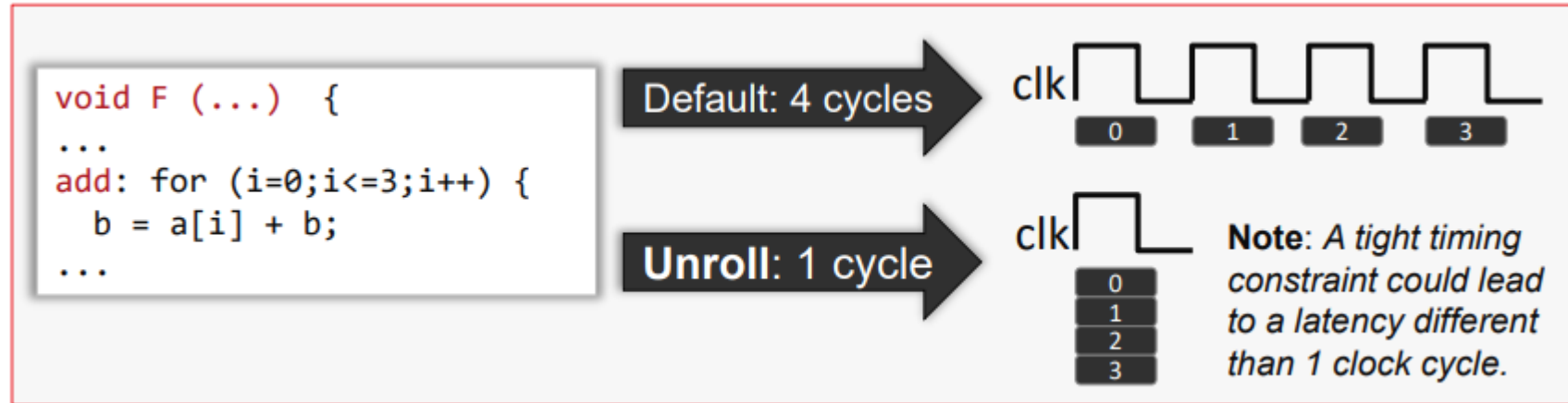


# Instruction-Level Parallelism

- PIPELINE applies to loops or functions
- Instructs HLS to process variables continuously
- Initiation Interval (II): Number of clock cycles before the function can accept new inputs



# Instruction-Level Parallelism



# Other Common Loop Manipulation Techniques

- <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls>
  - Loop merge
  - Loop flatten
  - Loop fission
- 
- A lot of well-known compiler optimizations can be used to increase parallelism opportunities in FPGAs
  - The primary goal is to maximize parallel execution via high resource utilization
- 