



Improving Performance

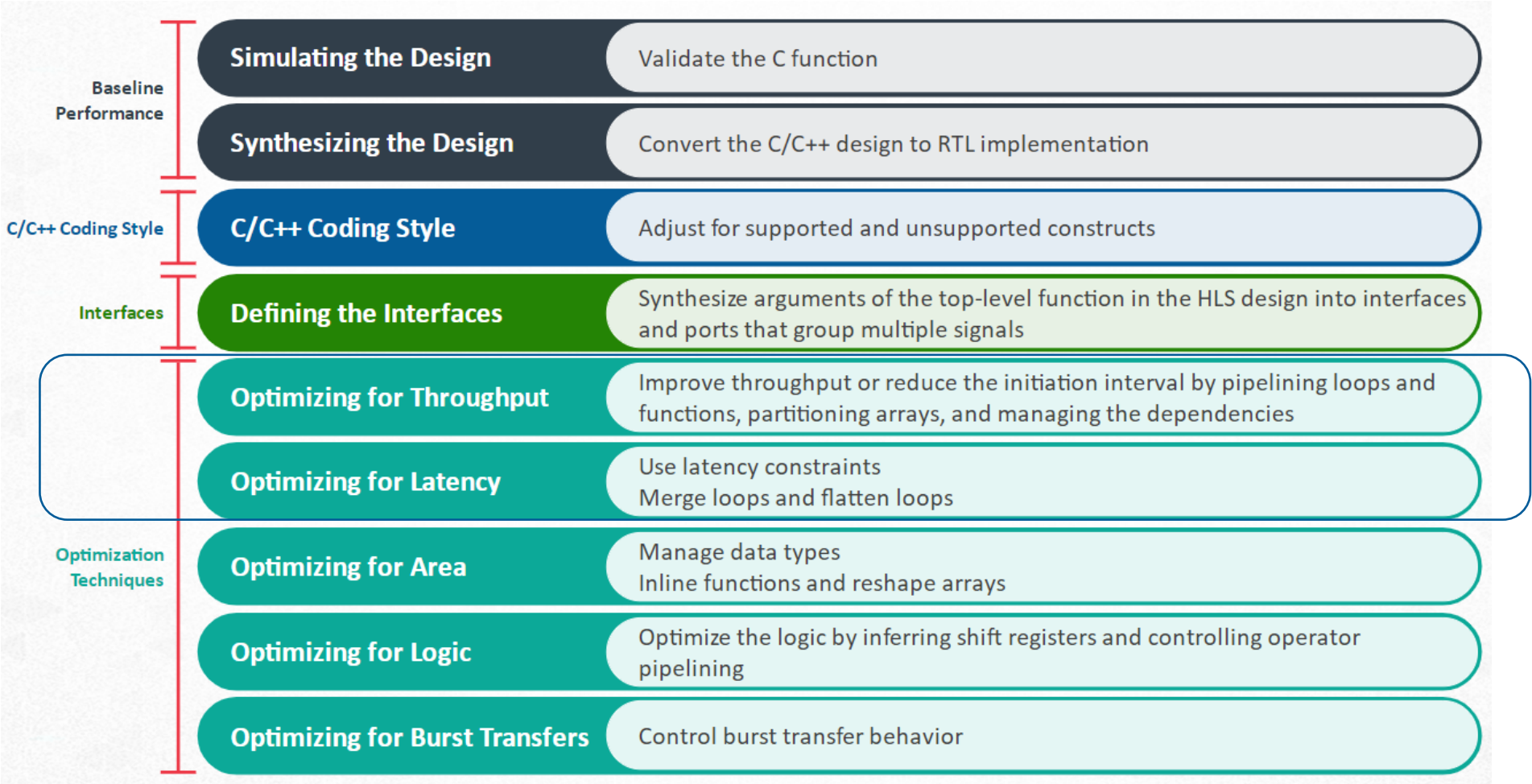
Objectives

- ▶ After completing this module, you will be able to:
 - Describe the main steps of HLS Design Methodology
 - Add directives to your design
 - List number of ways to improve performance
 - State directives which are useful to improve latency
 - Describe how loops may be handled to improve latency
 - Recognize the dataflow technique that improves throughput of the design
 - Describe the pipelining technique that improves throughput of the design
 - Identify some of the bottlenecks that impact design performance

Outline

- ▶ Adding Directives
- ▶ HLS Design Methodology
- ▶ Improving Throughput
- ▶ Improving Latency
- ▶ Performance Bottleneck
- ▶ Summary

Vitis HLS Design Methodology

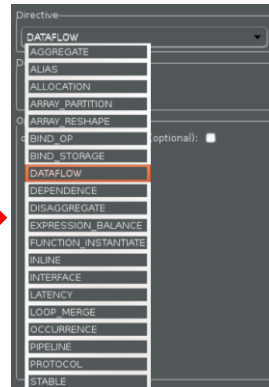
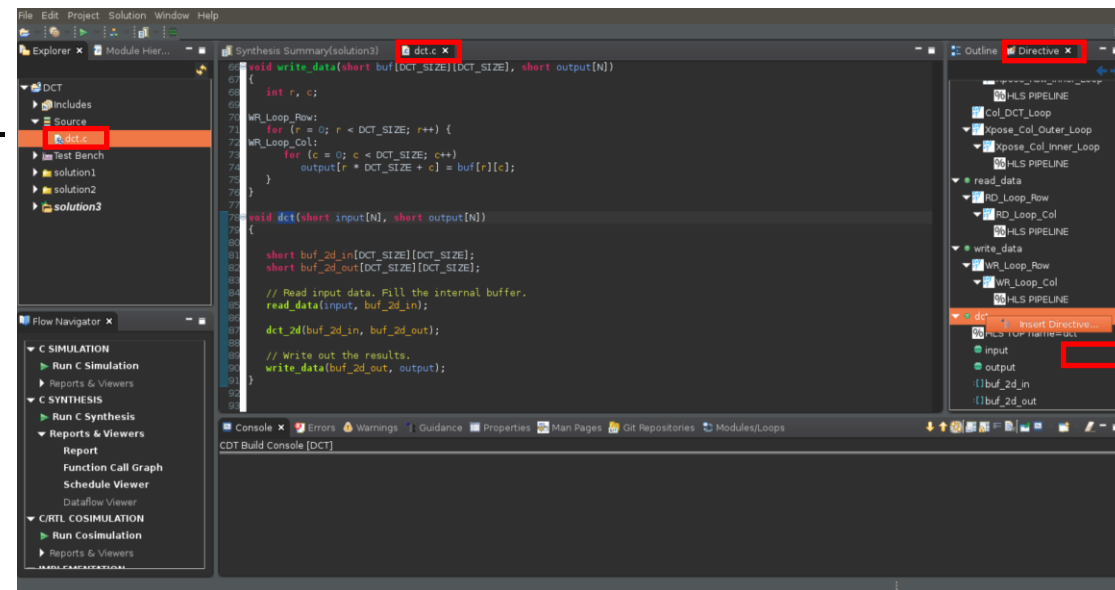


Improving Performance

- ▶ 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


Applying Directives

- ▶ If the source code is open in the GUI Information pane
 - The Directive tab in the Auxiliary pane shows all the locations and objects upon which directives can be applied (in the opened C file, not the whole design)
 - Functions, Loops, Regions, Arrays, Top-level arguments
 - Select the object in the Directive Tab
 - “dct” function is selected
 - Right-click to open the editor dialog box
 - Select a desired directive from the drop-down menu
 - “DATAFLOW” is selected
 - Specify the Destination
 - Source File
 - Directive File



Optimization Directives: Tcl or Pragma

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



```

78 void dct(short input[N], short output[N])
79 {
80 #pragma HLS DATAFLOW
    
```

Directive: **DATAFLOW**

Destination:

☒ Source File

☐ Directive File

Options:

disable_start_propagation (optional): ☐

- ▶ Directives can be placed in the directives file
 - The Tcl command is written into directives.tcl
 - There is a directives.tcl file in each solution
 - Each solution can have different directives

Once applied the directive will be shown in the Directives tab (right-click to modify or delete)



```

% HLS DATAFLOW
    
```

Directive: **DATAFLOW**

Destination:

☐ Source File

☒ Directive File

Options:

disable_start_propagation (optional): ☐

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): ☐

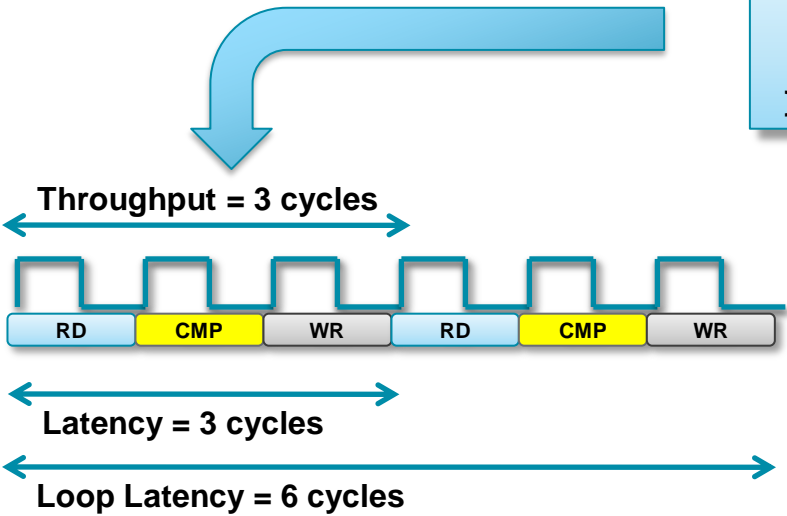
style (optional):

Pipeline for Performance

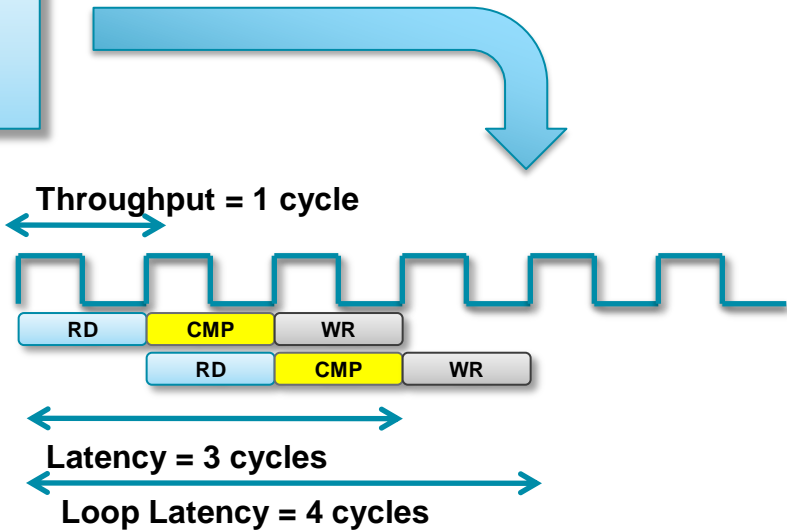
- ▶ When the PIPELINE directive is used, it **automatically unrolls** all loops and sub-loops in the hierarchy below, creating a great deal of logic
- ▶ Loops with **variable bounds** cannot be unrolled, and any loops and functions in the hierarchy above these loops cannot be pipelined
- ▶ Pipeline of 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



```

Loop:for(i=1;i<3;i++) {
  op_Read;
  op_Compute;
  op_Write;
}

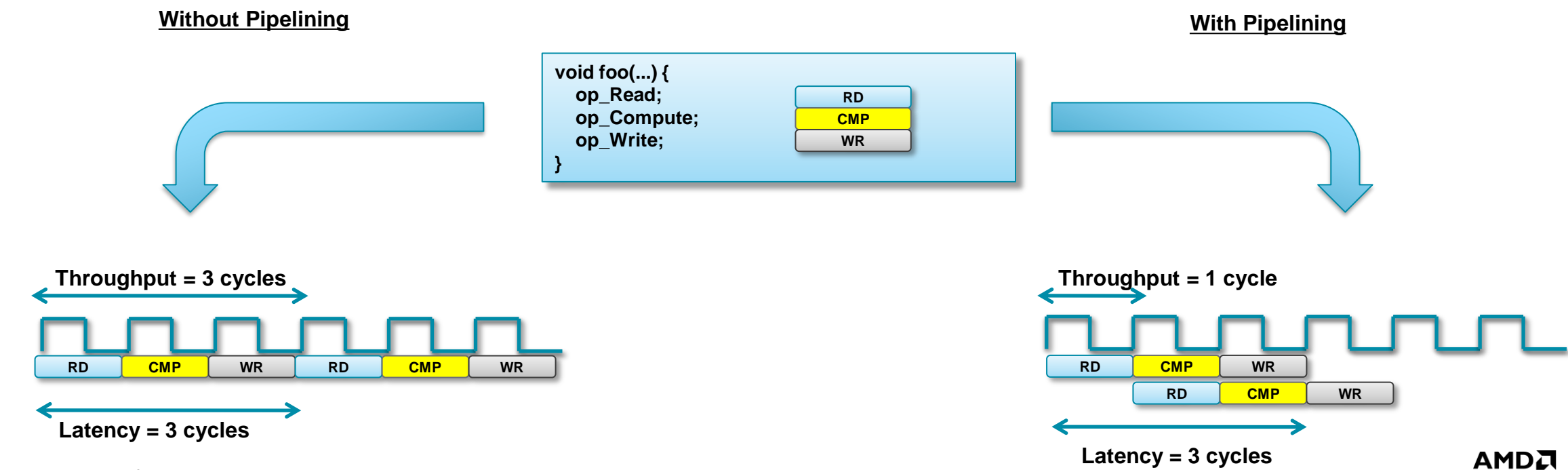
```

- ▶ 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



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];
    }
  }
}
```

1 adder, 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];
    }
  }
}
```

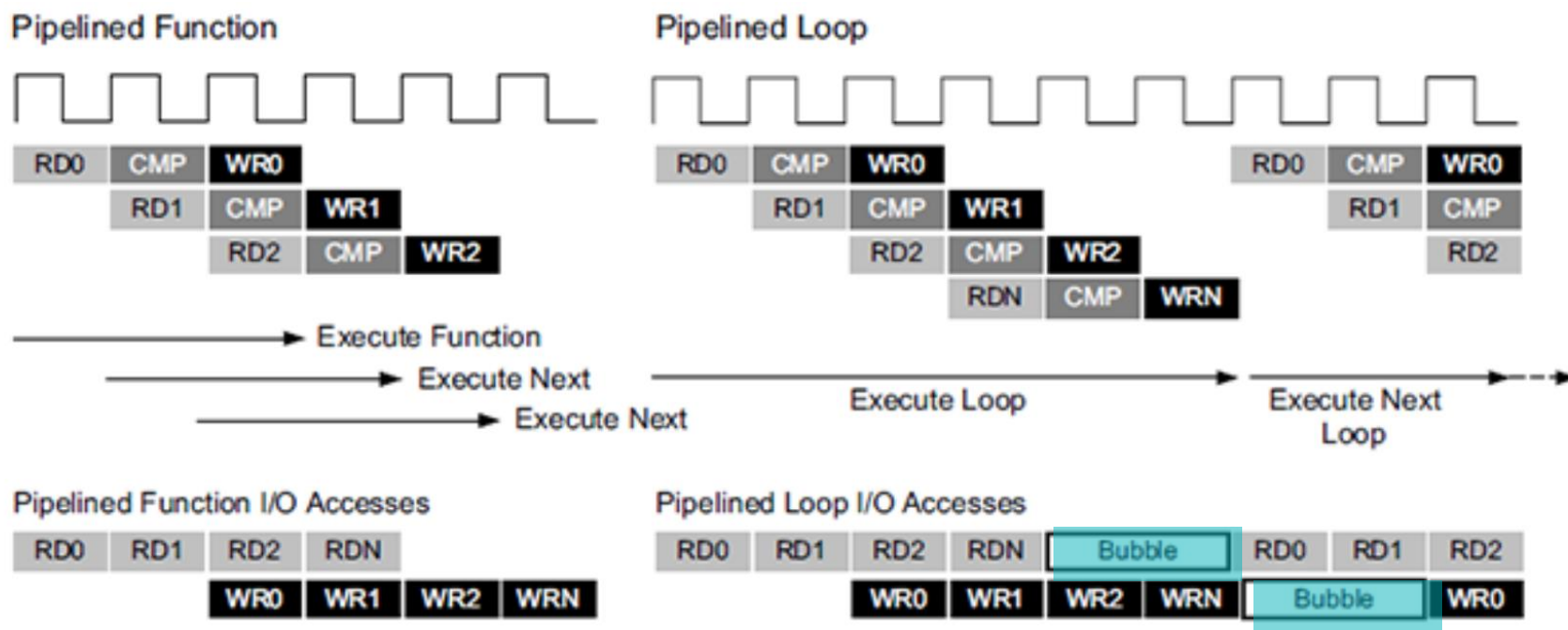
Unrolls L2
M adders, 3M accesses

```
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



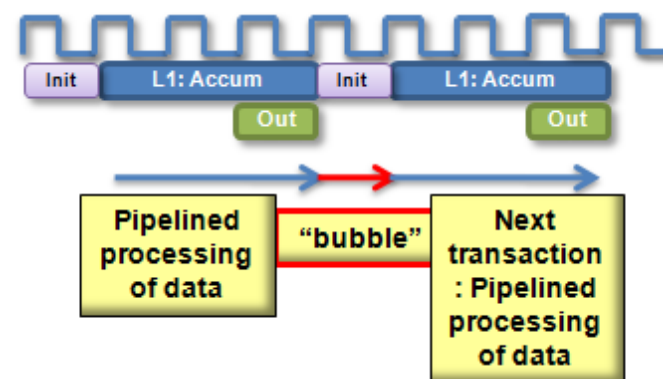
- ▶ In case of functions, the pipeline runs forever and never ends
- ▶ In case of loops, the pipeline executes until all iterations of the loop are completed
- ▶ Bubble is nothing but
 - Point when no new inputs are read as the loop completes the execution of the final iterations
 - Point when no new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

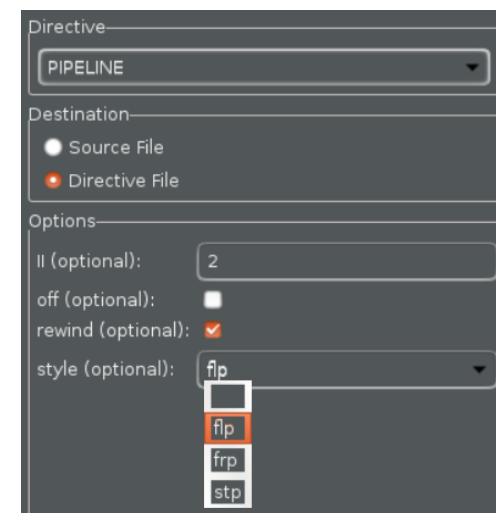
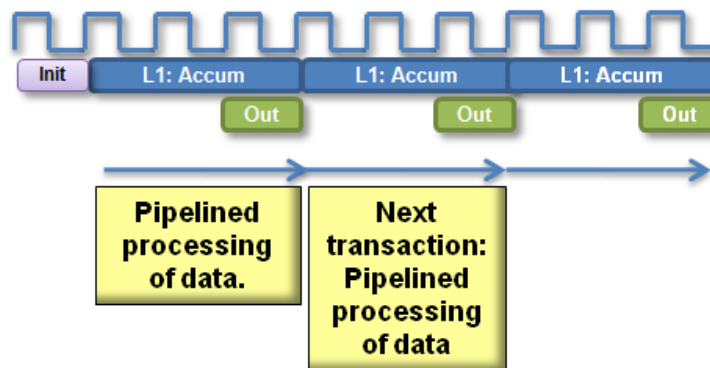


- The function will process a stream of data
- The next time the function is called, it still needs to execute the initial (init) operations
 - These operations are any which occur before the loop starts
 - These operations may include interface start/stop/done signals
- Due to these operations, an unexpected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

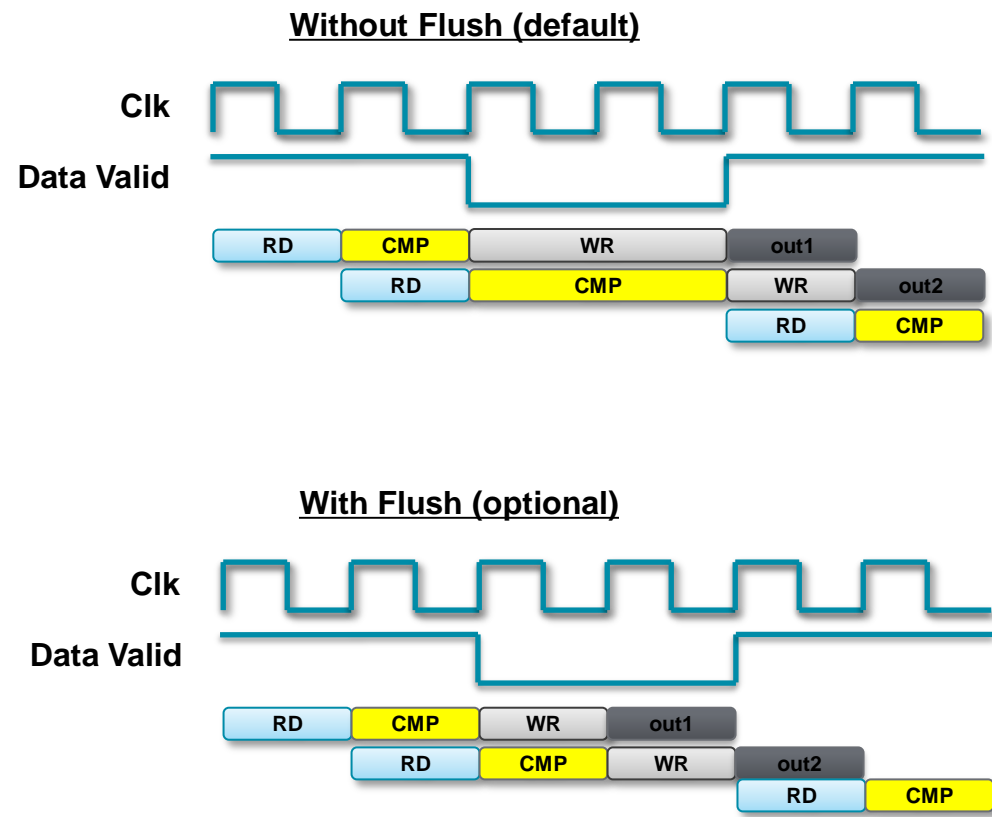
```
void foo_top (in1, in2, ...) {
  static accum=0;
  ...
  L1:for(i=1;i<N;i++) {
    accum = accum + in1 + in2;
  }
  out1_data = accum;
  ...
}
```



- ▶ 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

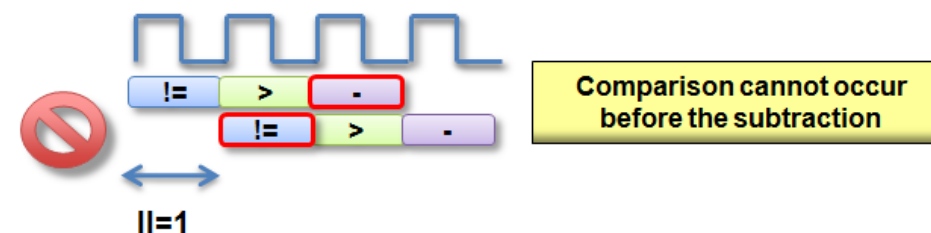
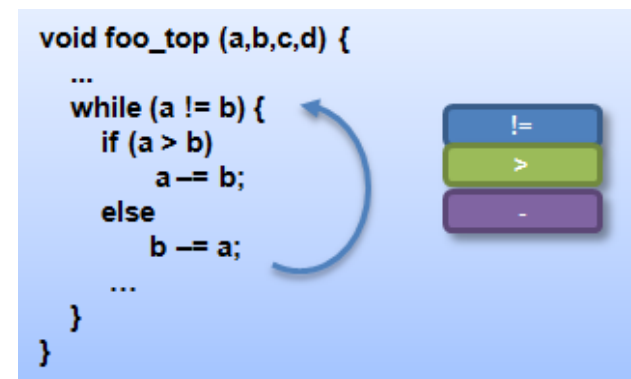
Pipeline Flush

- ▶ 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



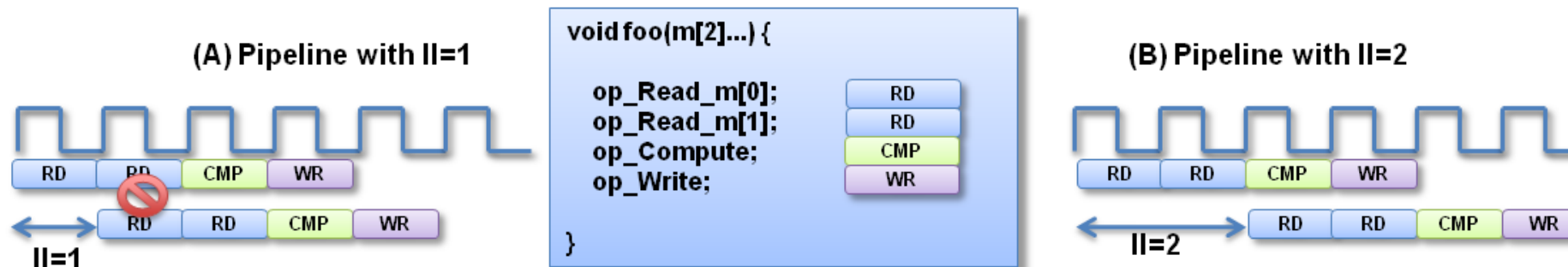
Issues That Prevent Pipelining

- ▶ 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 classic 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

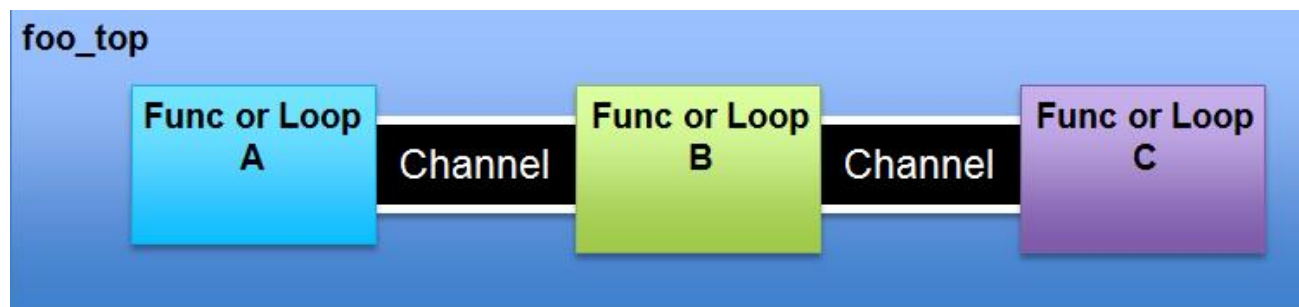


- An II=1 cannot be implemented
 - The same port cannot be read at the same time
 - Similar effect with other resource limitations
 - For example, if functions or multipliers etc. are limited
- Vitis HLS will automatically increase the II to 2 as the tool will always try to create a design, even if constraints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

Dataflow Optimization

► Dataflow Optimization

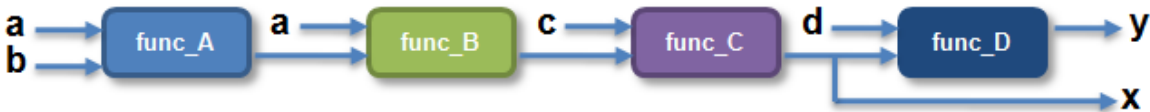
- Creates a parallel process architecture
- Useful on a set of sequential tasks as it creates an architecture of concurrent processes for these tasks
- Allows the execution of the tasks to overlap
- Improves performance over a statically pipelined solution
- Increases the overall throughput of the design 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**



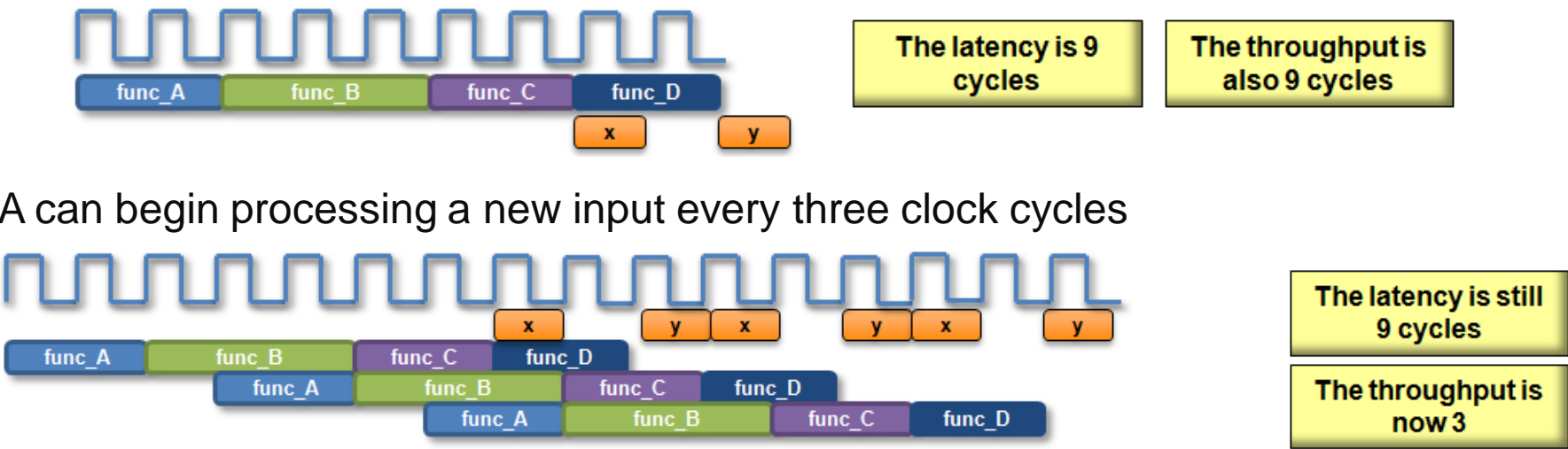
Dataflow for Performance

- ▶ 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



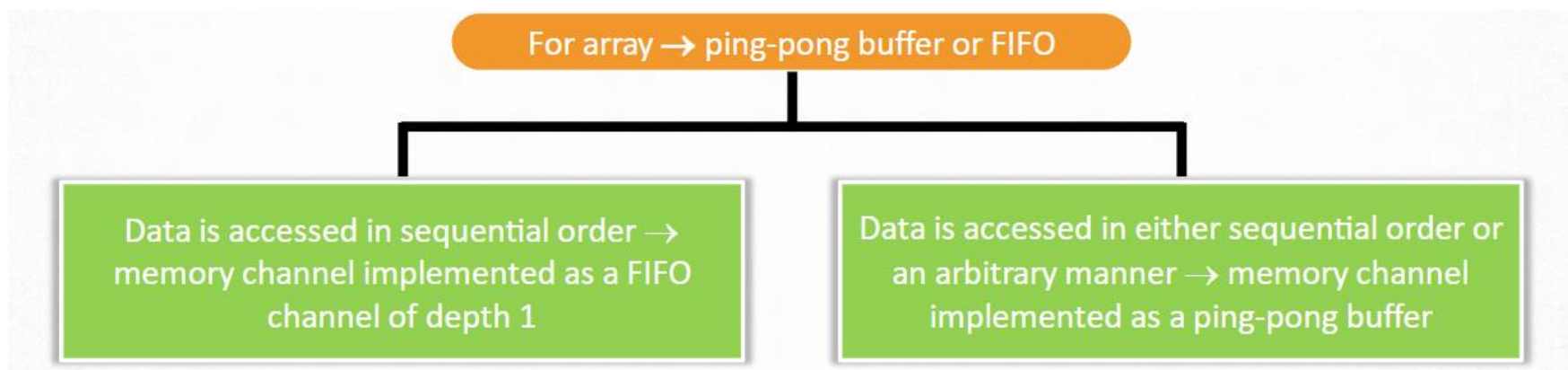
Configuring the Dataflow Channel

- ▶ 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 mempy 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 customer pr 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 templ[N];
    Loop1: for(int i = 0; i < N; i++) {
        templ[i] = data_in[i] * scale;
    }
    Loop2: for(int j = 0; j < N; j++) {
        data_out1[j] = templ[j] * 123;
    }
    Loop3: for(int k = 0; k < N; k++) {
        data_out2[j] = templ[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 templ[N], temp2[N], temp3[N];
    Loop1: for(int i = 0; i < N; i++) {
        templ[i] = data_in[i] * scale;
    }
    Split(templ, 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

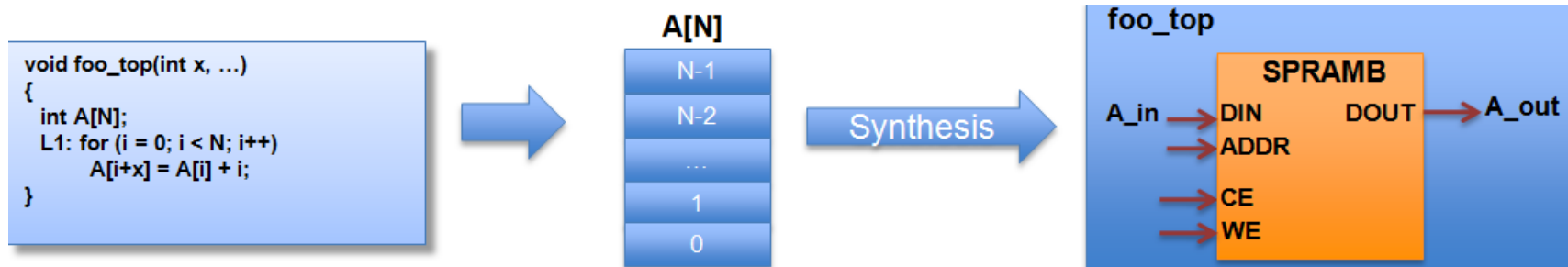
Optimize for throughput

- ▶ 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

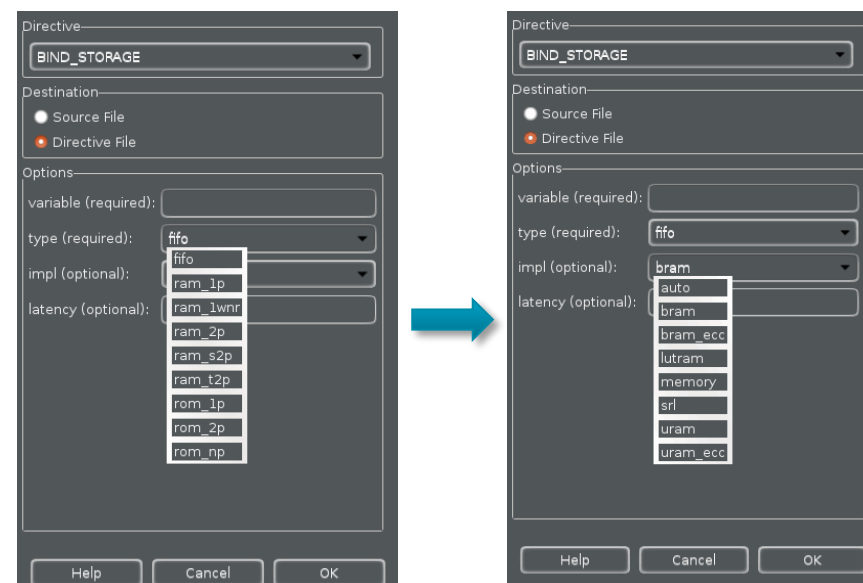
Directives and Configurations	Description
CONFIG ARRAY PARTITION	Determines how arrays are partitioned, including global arrays and if the partitioning impacts array ports
CONFIG COMPILE	Controls synthesis-specific optimizations such as the automatic loop pipelining and floating point math optimizations
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

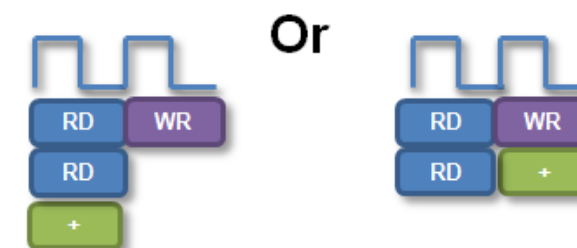


- ▶ To use a FIFO instead of a block RAM -> Stream directive
- ▶ Arrays are automatically specified as streaming:
 - If an array is set as interface type `ap_fifo`, `axis`, or `ap_hs`, etc
 - If the array are used in a region where the DATAFLOW is applied.
- ▶ All the other arrays must be specified as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

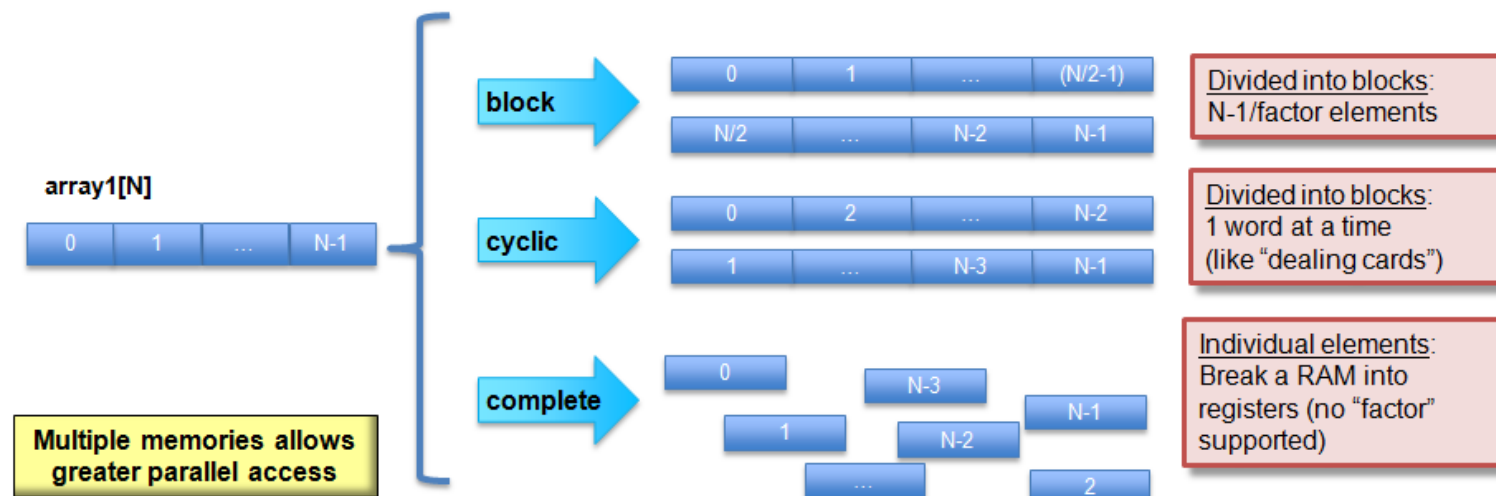
Specify `BIND_STORAGE` directive

Which type of RAM is used

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

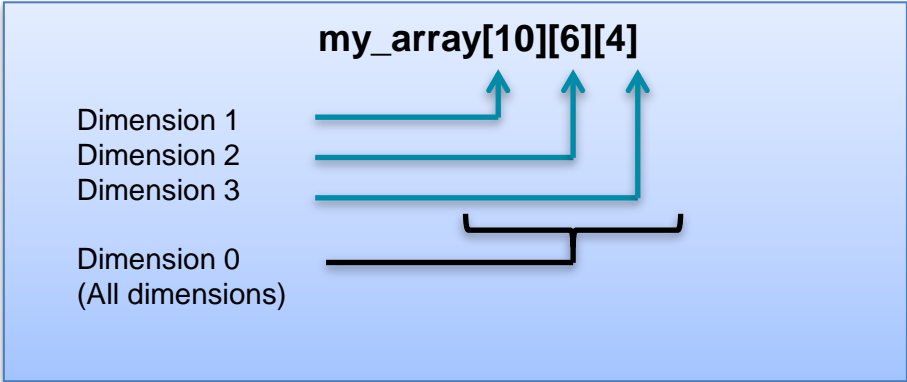
Array Partitioning

- ▶ 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]

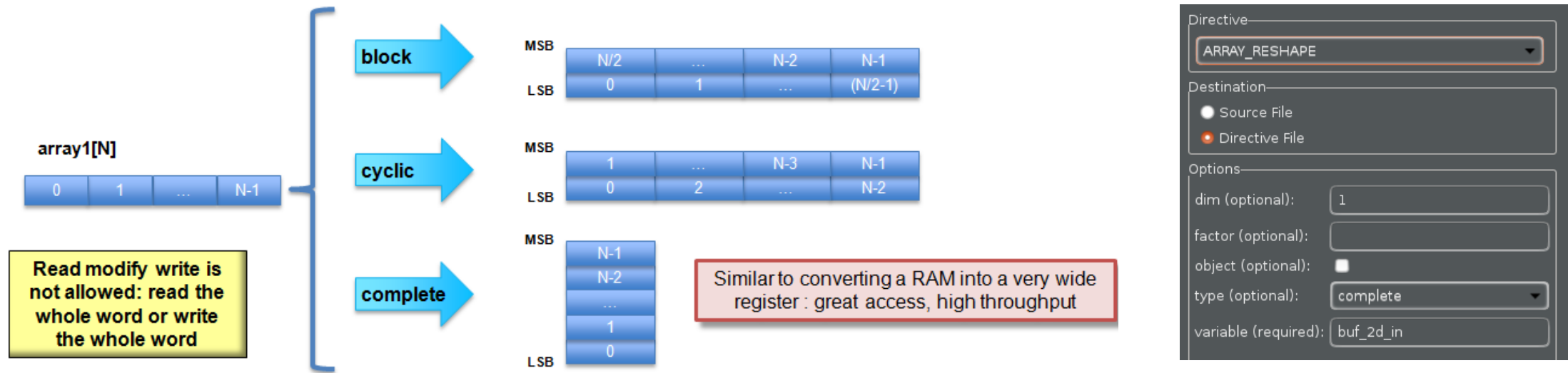
Configuring Array Partitioning

- ▶ 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	<input type="text" value="4"/>	4	
throughput_driven	<input type="text" value="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



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.
```

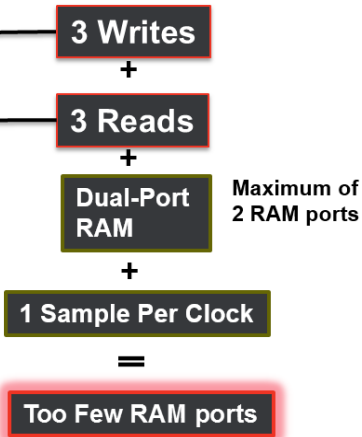
```
INFO: [SCHED 61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
```

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

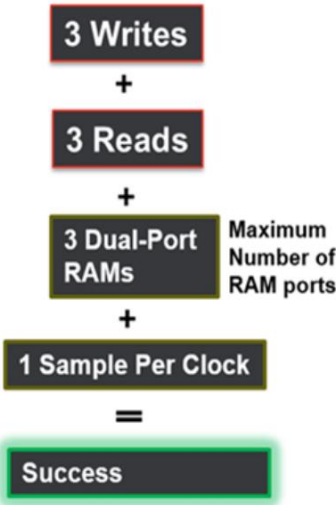
```
@I [SCHED-61] Pipelining loop 'I SOBEL_LOOP'.
@W [SCHED-69] Unable to schedule 'store' operation (image_demo.cpp:172) of
variable 'y' on array 'buff_A' due to limited resources (II = 2).
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 3, Depth: 13.
```

```
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) {
                temp_x = inter_pix[row][col];
                buff_A[0][col] = rgb2y(temp_x);
                buff_B[0][col] = temp_x;
            }
        }
    }
    ...
}
```



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) {
                temp_x = inter_pix[row][col];
                buff_A[0][col] = rgb2y(temp_x);
                buff_B[0][col] = temp_x;
            }
        }
    }
    ...
}
```

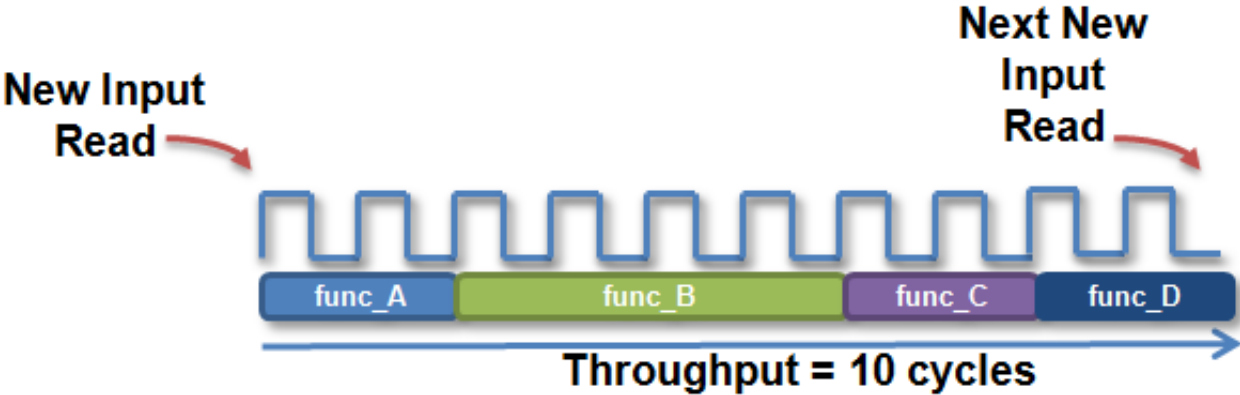


Improving Latency

Latency and Throughput – The Performance Factors

► Design Throughput

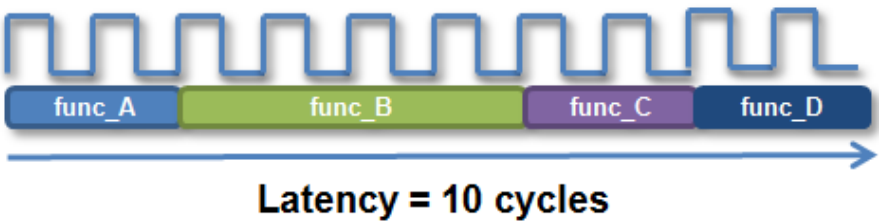
- The throughput of the design is the number of cycles between new inputs
 - By default (no concurrency) this is the same as latency
 - Next start/read is when this transaction ends



► 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

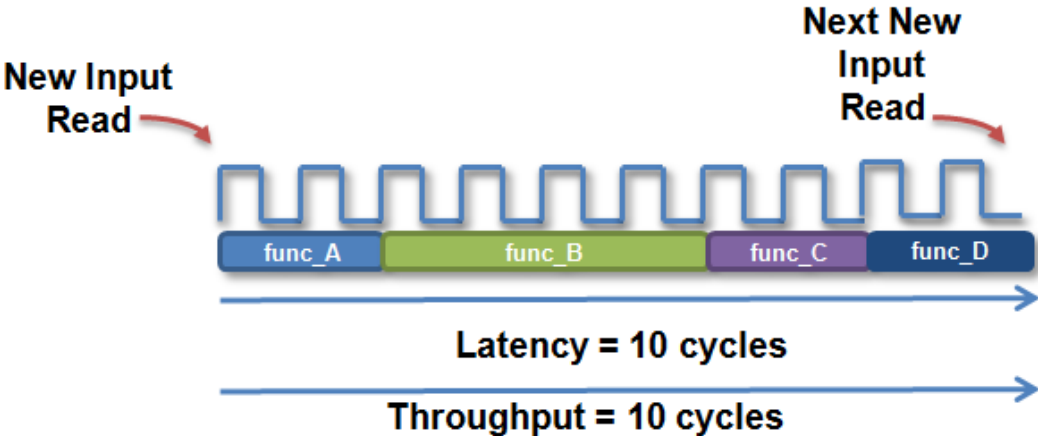
```
void foo_top (a,b,c,d, *x, *y) {  
    ...  
    func_A(...);  
    func_B(...);  
    func_C(...);  
    func_D(...);  
    return res;  
}
```



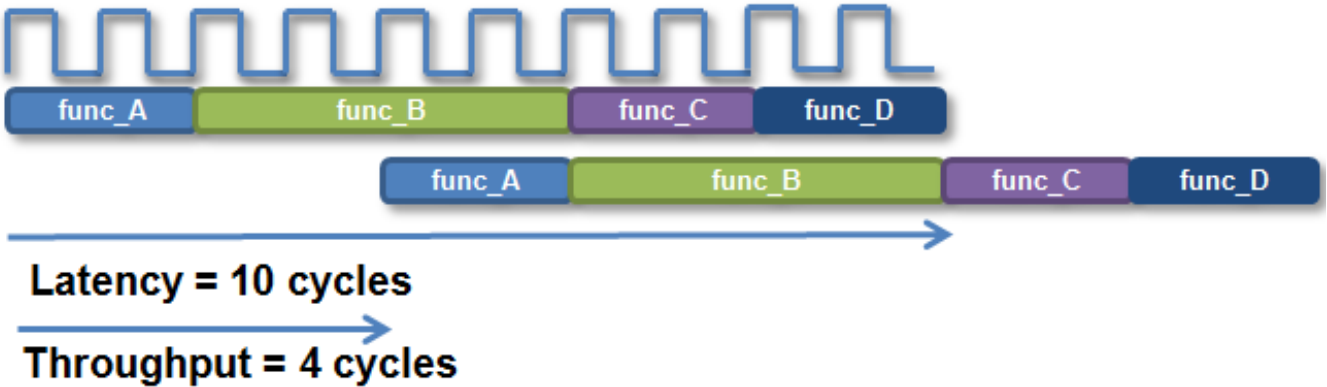
Latency and Throughput

- ▶ 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;  
}
```



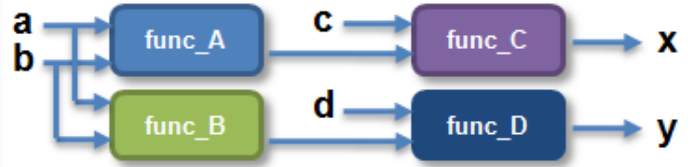
- ▶ Pipelining for higher throughput
 - Vitis HLS can pipeline functions and loops to improve throughput
 - Latency and throughput are related
 - We will discuss optimizing for latency first, then throughput



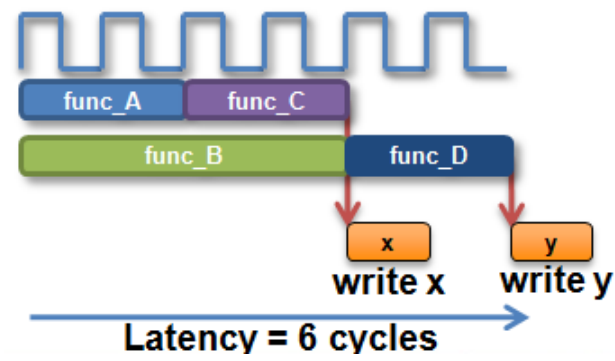
Vitis HLS: Minimize latency

- ▶ 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

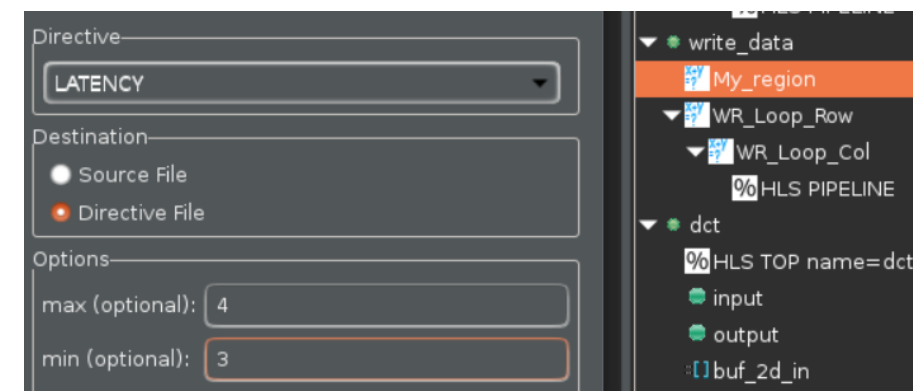
- ▶ Vitis HLS has the following directives to reduce latency
 - LATENCY
 - Allows a minimum and maximum latency constraint to be specified
 - LOOP_FLATTEN
 - Allows nested loops to be collapsed into a single loop with improved latency
 - LOOP_MERGE
 - Merge consecutive loops to reduce overall latency, 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

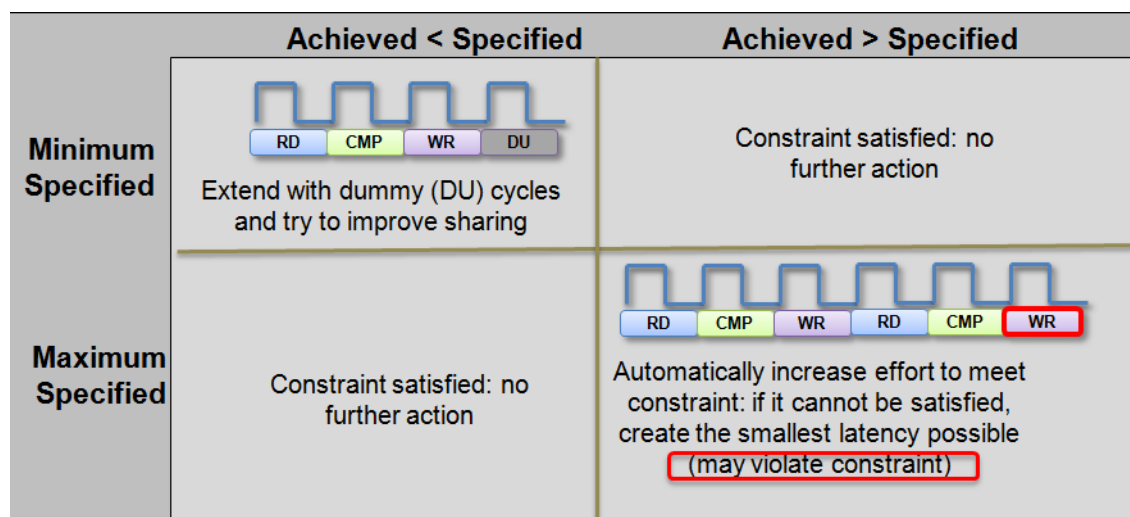
```
int write_data (int buf, int output) {  
    if (x < y) {  
        return (x + y);  
    } else {  
        My_Region: {  
            return (y - x) * (y + x);  
        }  
    }  
}
```

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



Impact of ranges

Directive
 LATENCY

Destination
☐ Source File
☒ Directive File

Options
 max (optional): 4
 min (optional): 3

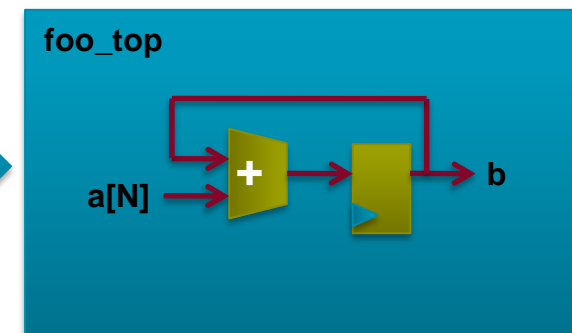
Review: Loops

- ▶ 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;  
    }  
    ...  
}
```

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

Synthesis



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;  
    }  
    ...  
}
```

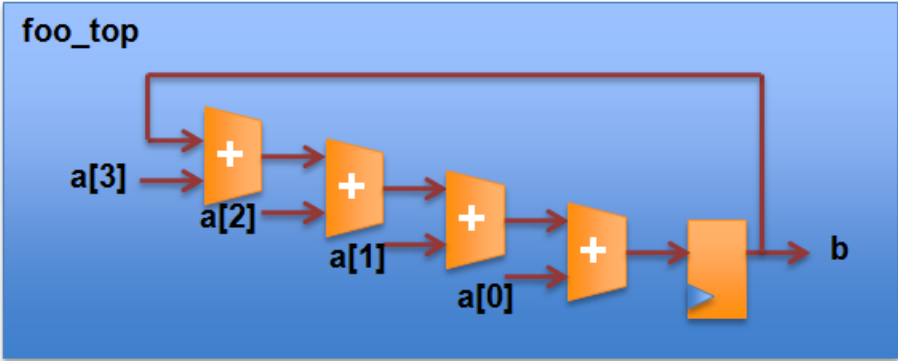


- This rolled loop will never take less than 4 cycles
 - No matter what kind of optimization is tried
 - This minimum latency is a function of the loop iteration count

Unrolled Loops can Reduce Latency

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

Unrolled



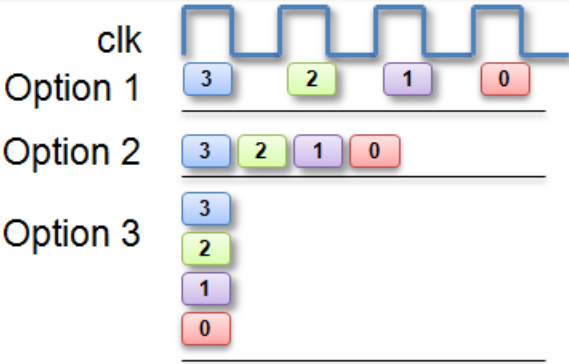
Select loop “Add” in the directives pane and right-click



Directive
UNROLL

Destination
☐ Source File
☒ Directive File

Options
factor (optional):
skip_exit_check (optional): ☐



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

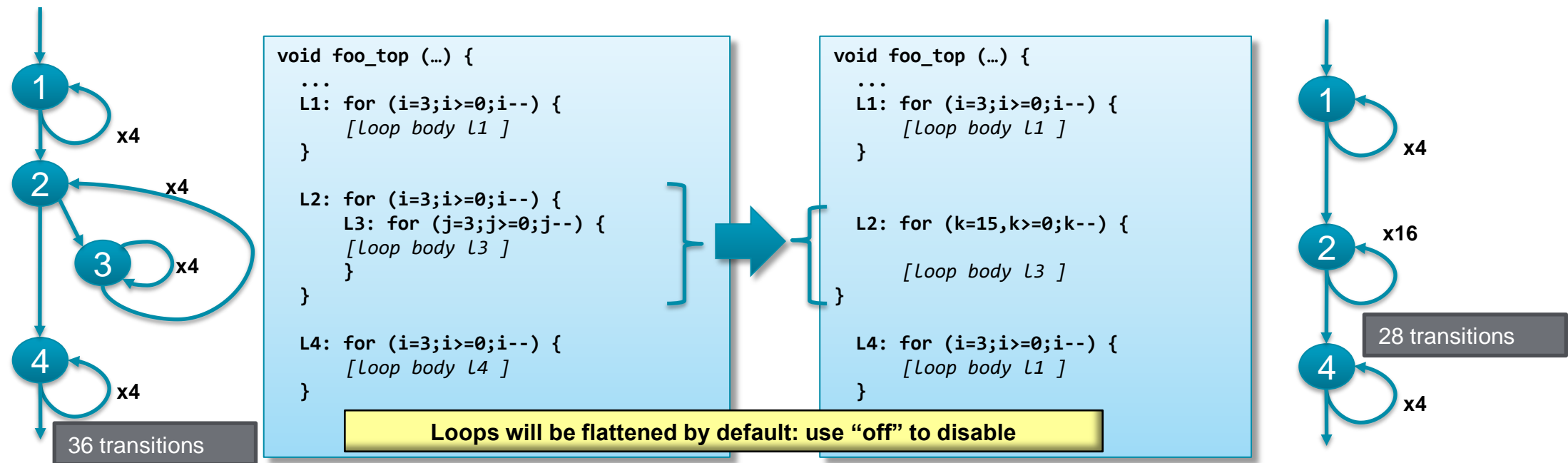
LOOP_FLATTEN

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

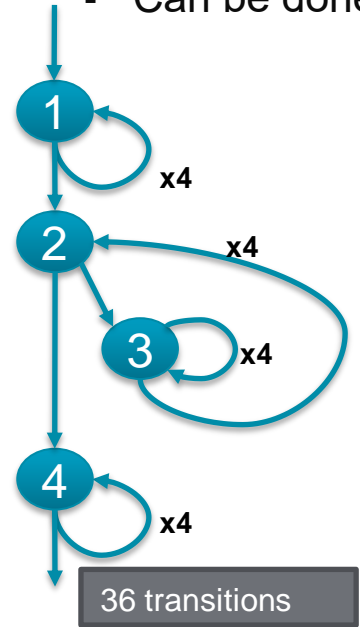
LOOP_MERGE

Destination

☐ Source File

☒ Directive File

Options
force (optional): ☐

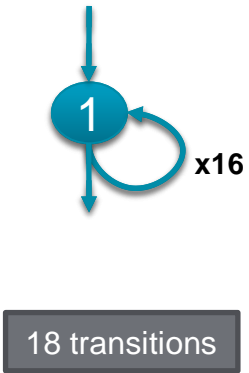


```
void foo_top (...) {  
    ...  
    L1: for (i=3;i>=0;i--) {  
        [Loop body L1 ]  
    }  
    L2: for (i=3;i>=0;i--) {  
        L3: for (j=3;j>=0;j--) {  
            [Loop body L3 ]  
        }  
    }  
    L4: for (i=3;i>=0;i--) {  
        [Loop body L4 ]  
    }  
}
```

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

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
?	?	?	?	?	?	no



Directive

LOOP_TRIPCOUNT

Destination

☐ Source File

☒ Directive File

Options

avg (optional):

max (optional): 1920

min (optional): 200



Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
1	7372833	10.000 ns	73.728 ms	2	7372834	no

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



Thank You

Disclaimer and Attribution

The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18

© Copyright 2021 Advanced Micro Devices, Inc. All rights reserved. Xilinx, the Xilinx logo, AMD, the AMD Arrow logo, Alveo, Artix, Kintex, Kria, Spartan, Versal, Vitis, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

