

Improving Performance



Objectives

- > **After completing this module, you will be able to:**
 - >> 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

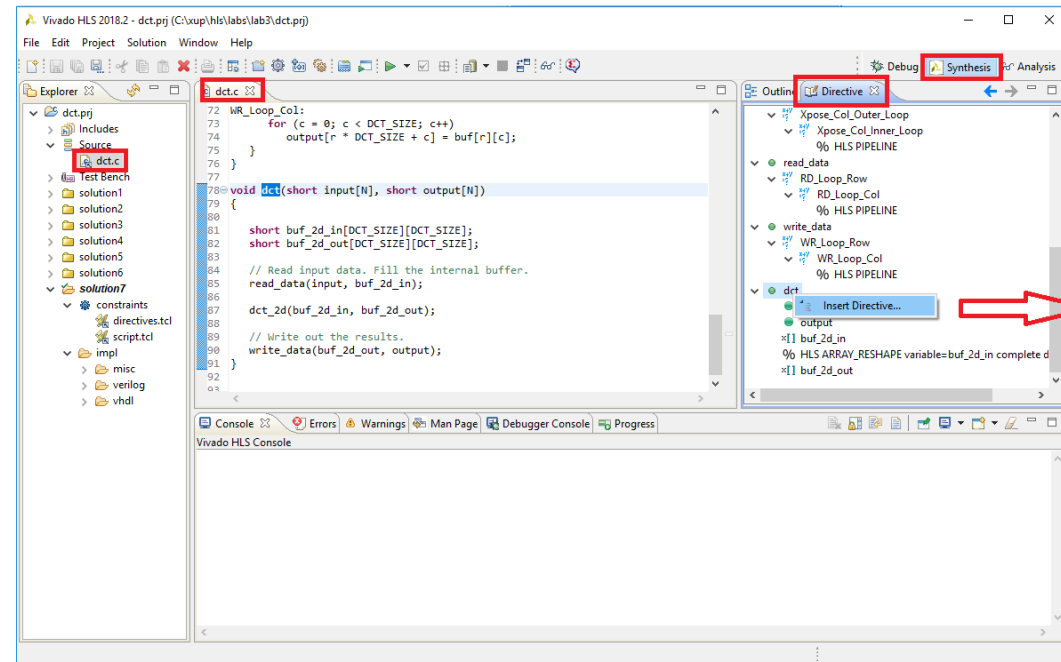
- > *Adding Directives*
- > Improving Latency
 - >> Manipulating Loops
- > Improving Throughput
- > Performance Bottleneck
- > Summary

Improving Performance

- > **Vivado HLS has a number of ways to improve performance**
 - >> Automatic (and default) optimizations
 - >> Latency directives
 - >> Pipelining to allow concurrent operations
- > **Vivado 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 Vivado 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



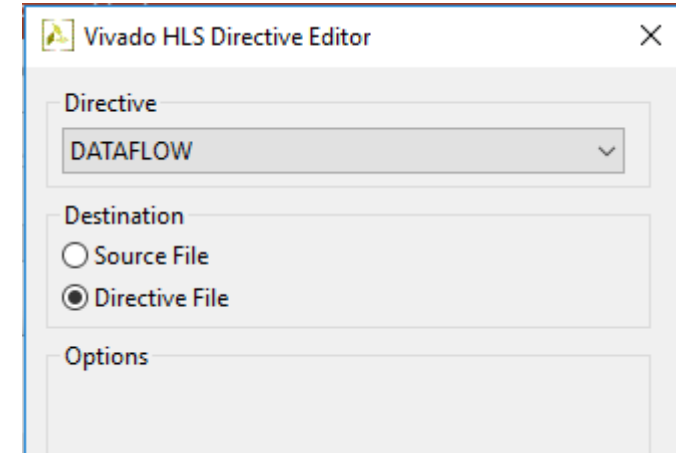
Optimization Directives: Tcl or Pragma

> Directives can be placed 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)

dct
% HLS DATAFLOW

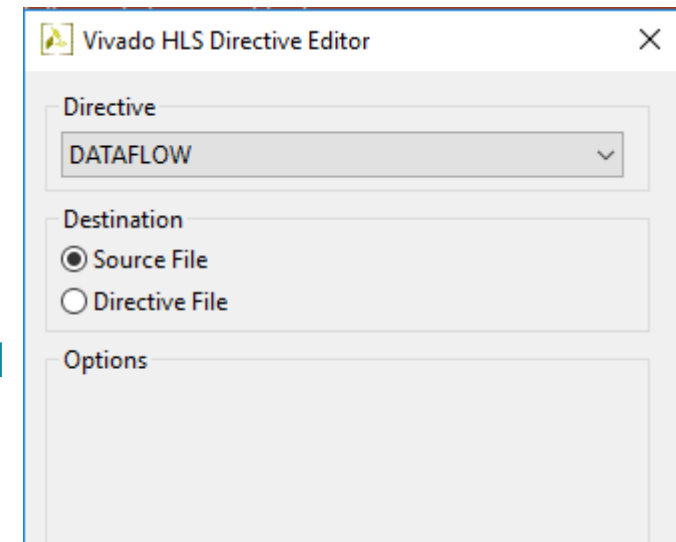


> Directives can be placed into the C source

- >> Pragmas are added (and will remain) in the C source file
- >> Pragmas (#pragma) will be used by every solution which uses the code

dct
HLS DATAFLOW

```
78 void dct(short input[N], short output[N])  
79 {  
80 #pragma HLS DATAFLOW  
81
```

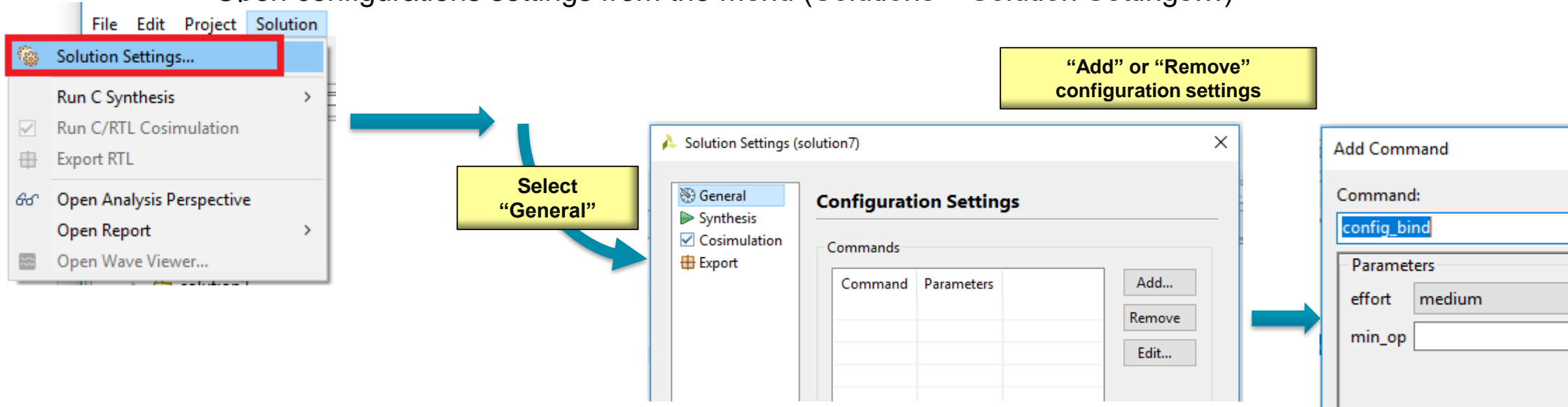


Solution Configurations

> Configurations can be set on a solution

>> Set the default behavior for that solution

- Open configurations settings from the menu (Solutions > Solution Settings...)

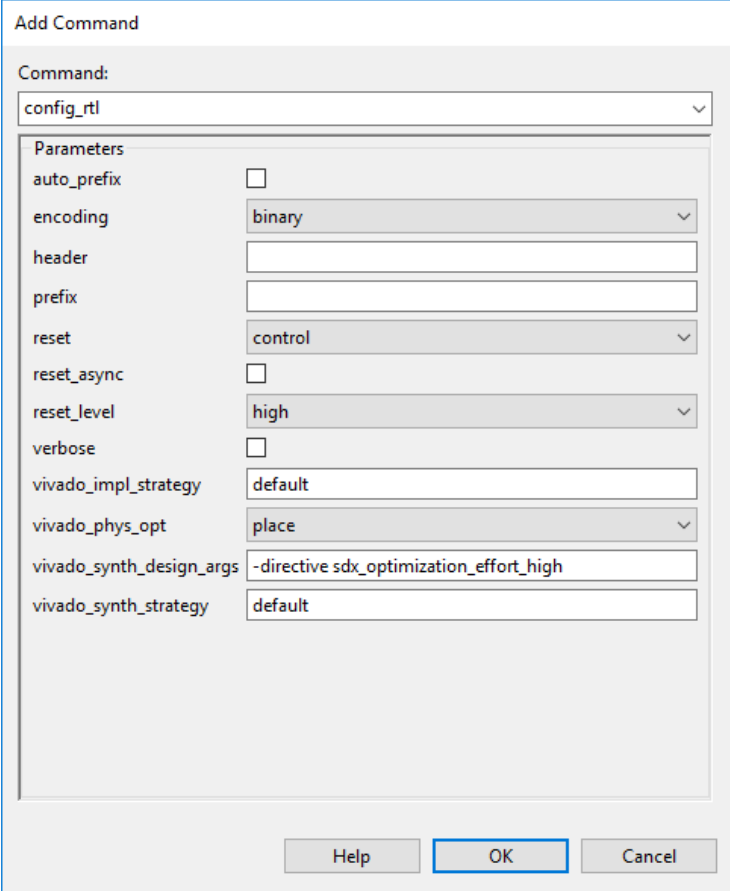


>> Choose the configuration from the drop-down menu

- Array Partitioning, Binding, Dataflow Memory types, Interface, RTL Settings, Core, Compile, Schedule efforts

Example: Configuring the RTL Output

- > **Specify the FSM encoding style**
 - >> By default the FSM is auto
- > **Add a header string to all RTL output files**
 - >> Example: Copyright Acme Inc.
- > **Add a user specified prefix to all RTL output filenames**
 - >> The RTL has the same name as the C functions
 - >> Allow multiple RTL variants of the same top-level function to be used together without renaming files
- > **Reset all registers**
 - >> By default only the FSM registers and variables initialized in the code are reset
 - >> RAMs are initialized in the RTL and bitstream
- > **Synchronous or Asynchronous reset**
 - >> The default is synchronous reset
- > **Active high or low reset**
 - >> The default is active high



The screenshot shows the 'Add Command' dialog box with the 'config_rtl' command selected. The parameters are configured as follows:

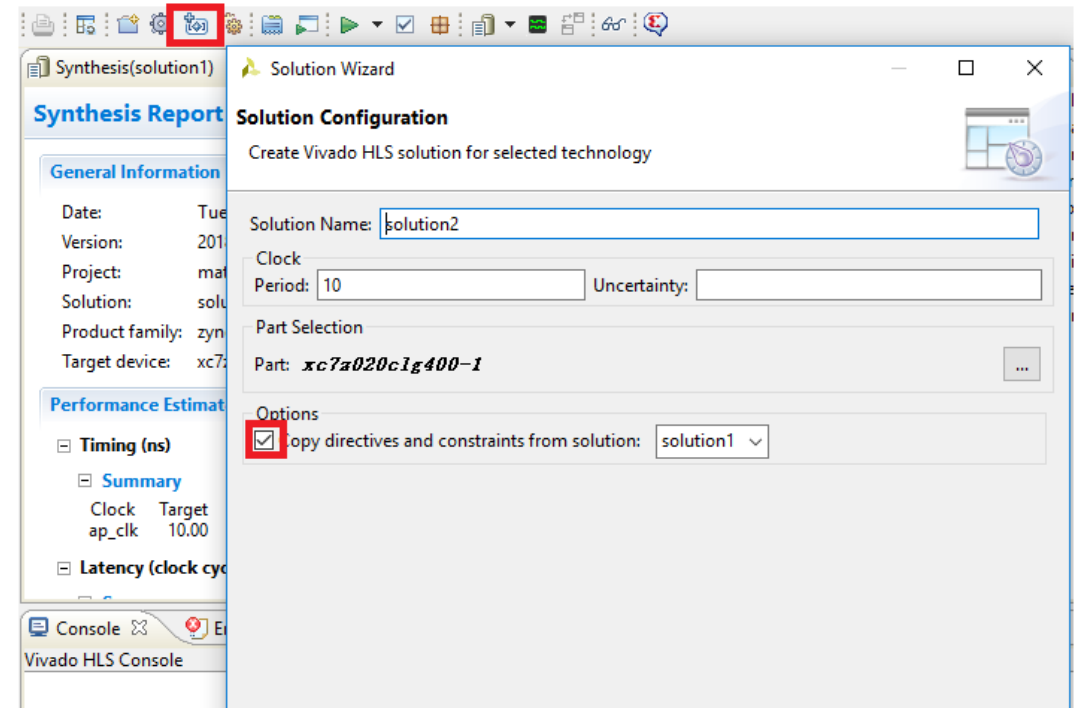
Parameter	Value
auto_prefix	<input type="checkbox"/>
encoding	binary
header	
prefix	
reset	control
reset_async	<input type="checkbox"/>
reset_level	high
verbose	<input type="checkbox"/>
vivado_impl_strategy	default
vivado_phys_opt	place
vivado_synth_design_args	-directive sdx_optimization_effort_high
vivado_synth_strategy	default

Buttons at the bottom: Help, OK, Cancel.

The remainder of the configuration commands will be covered throughout the course

Copying Directives into New Solutions

- > **Click the New Solution Button**
- > **Optionally modify any of the settings**
 - >> Part, Clock Period, Uncertainty
 - >> Solution Name
- > **Copy existing directives**
 - >> By default selected
 - >> Uncheck if do not want to copy
 - >> No need to copy pragmas, they are in the code



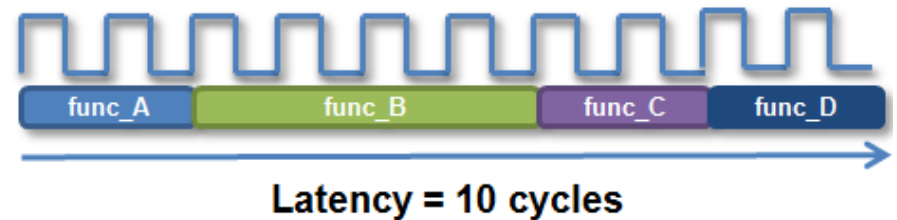
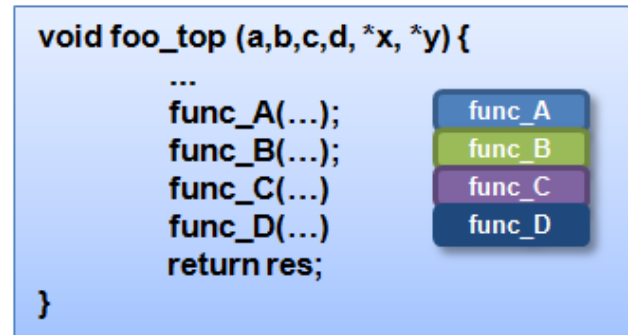
Improving Latency



Latency and Throughput – The Performance Factors

> 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 cycles

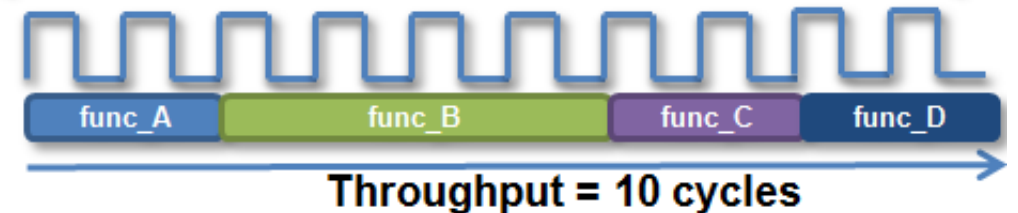


> Design Throughput

- >> 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

New Input
Read

Next New
Input
Read

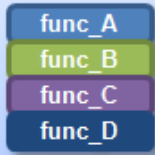


Latency and Throughput

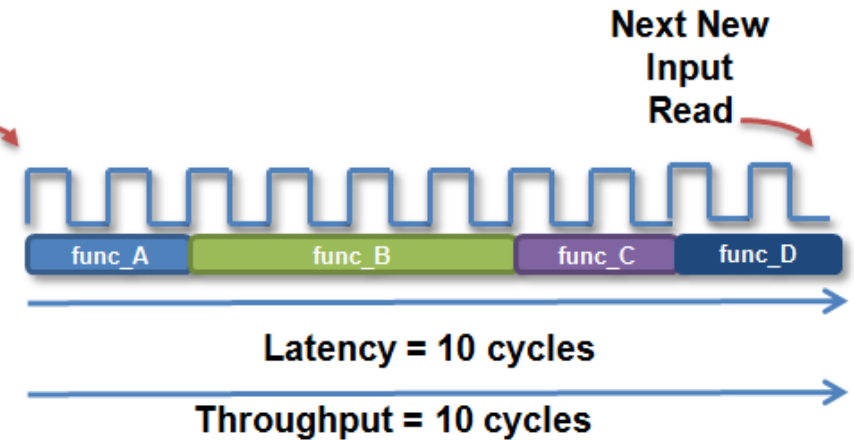
> In the absence of any concurrency

- >> 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;  
}
```

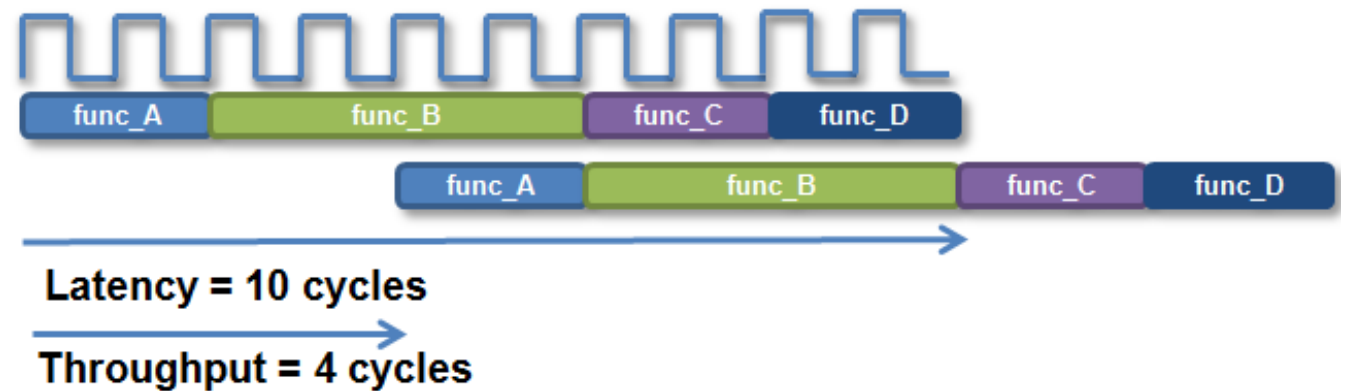


New Input
Read



> Pipelining for higher throughput

- >> Vivado HLS can pipeline functions and loops to improve throughput
- >> Latency and throughput are related
- >> We will discuss optimizing for latency first, then throughput



Vivado HLS: Minimize latency

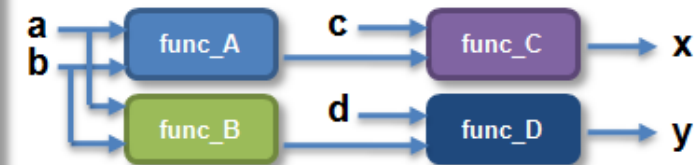
> Vivado HLS will by default minimize latency

>> 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

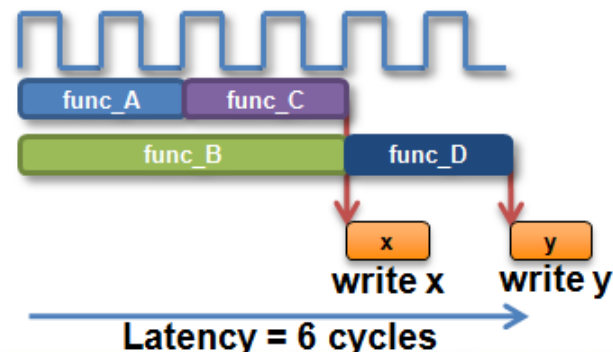
```
void foo_top (a,b,c,d, *x, *y) {  
    ...  
    func_A(a,b,t1);  
    func_B(a,b,t2);  
    func_C(c,t1,&x)  
    func_D(d,t2,&y)  
}
```



> Vivado 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

> Vivado 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

>> UNROLL

Default Behavior: Minimizing Latency

> Functions

- >> Vivado HLS will seek to minimize latency by allowing functions to operate in parallel
 - As shown on the previous slide

> Loops

- >> Vivado HLS will not schedule loops to operate in parallel by default
 - Dataflow optimization must be used or the loops must be unrolled
 - Both techniques are discussed in detail later

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



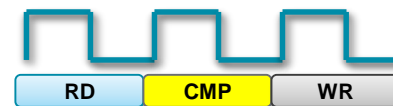
> Operations

- >> Vivado HLS will seek to minimize latency by allowing the operations to occur in parallel
- >> It does this within functions and within loops

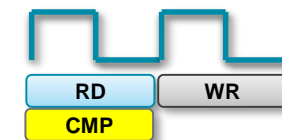
```
void foo(...) {  
  op_Read;  
  op_Compute;  
  op_Write;  
}
```



Example with Sequential Operations



Example of Minimizing latency with Parallel Operations

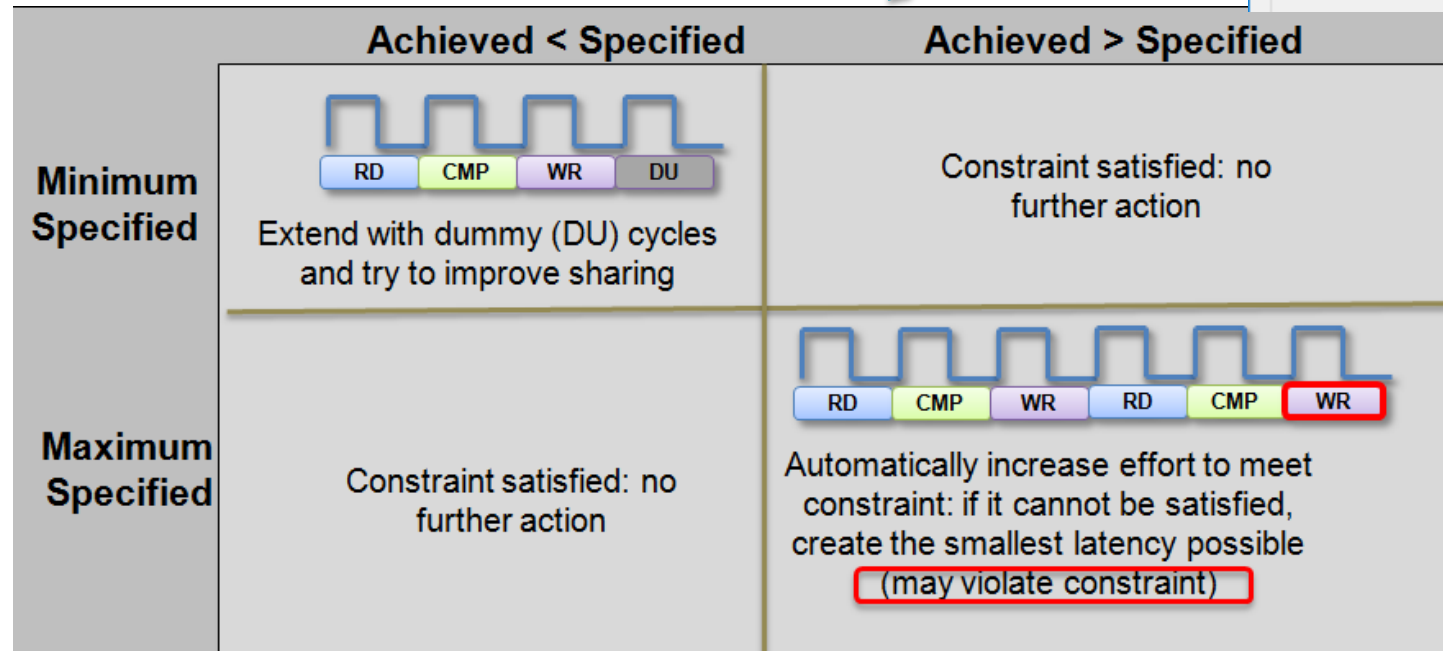
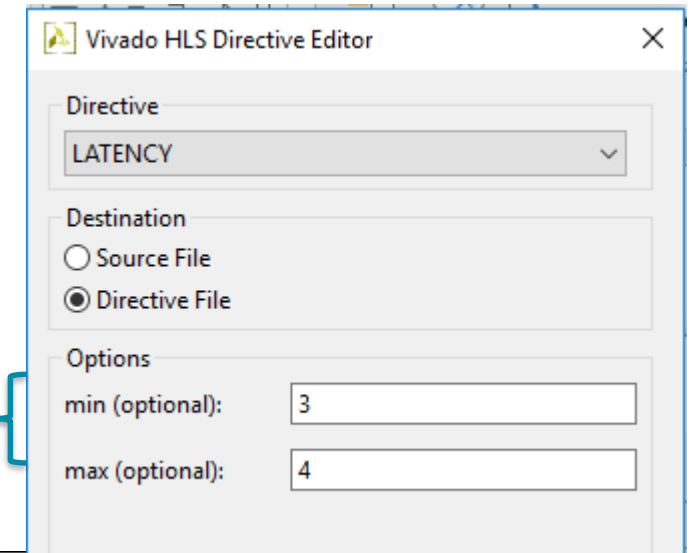


Latency Constraints

> Latency constraints can be specified

- >> Can define a minimum and/or maximum latency for the location
 - This is applied to all objects in the specified scope
- >> No range specification: schedule for minimum
 - Which is the default

Impact of ranges

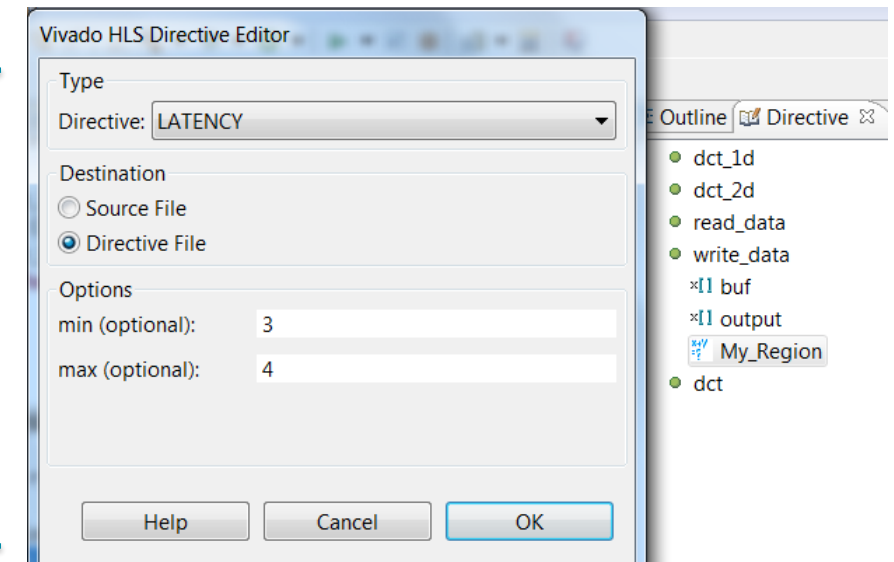


Region Specific Latency Constraint

- > 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



Improving Latency - Manipulating Loops



Review: Loops

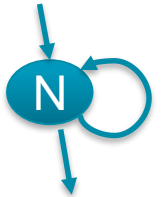
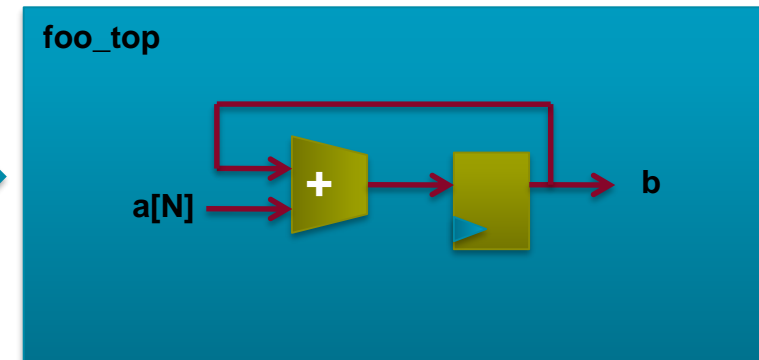
> By default, loops are rolled

- >> 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

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;  
    }  
    ...  
}
```

Clock
Adder Delay



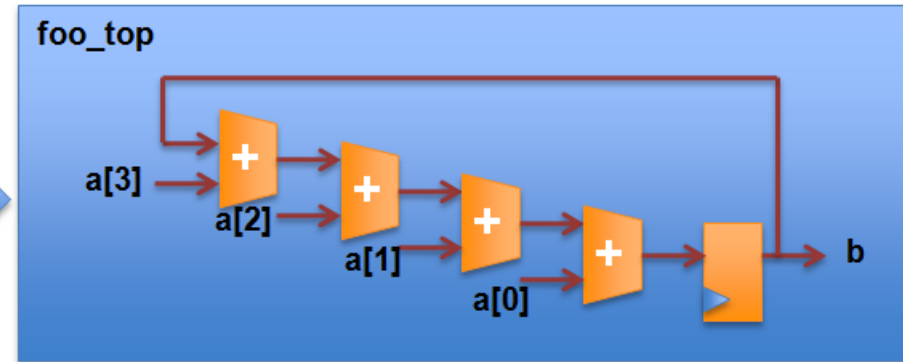
>> This rolled loop will never take less than 4 cycles

- No matter what kind of optimization is tried
- This minimum latency is a function of the loop iteration count

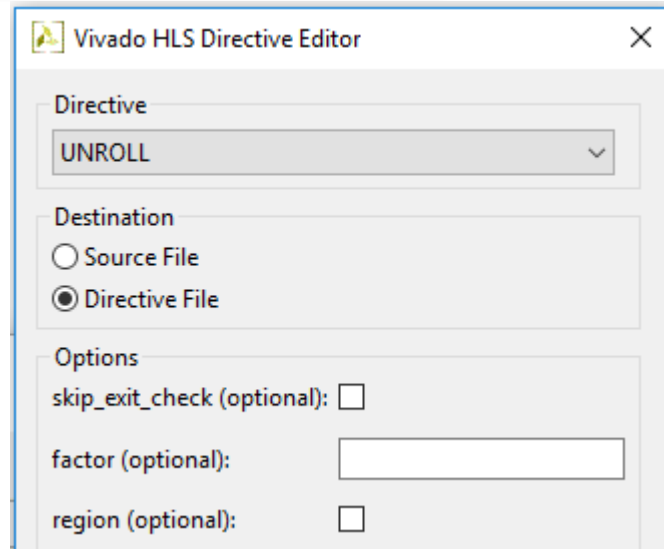
Unrolled Loops can Reduce Latency

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

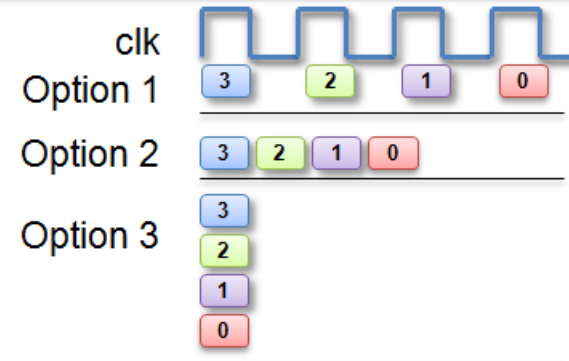
Unrolled



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



Options explained on next slide



Unrolled loops allow greater option & exploration

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)
 - Vivado 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];  
}
```

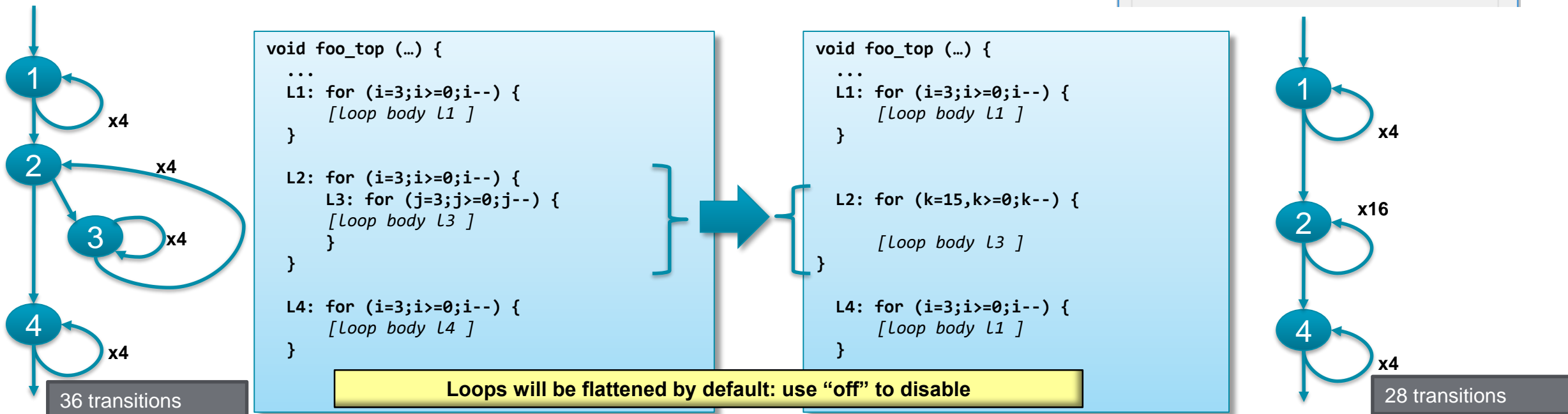
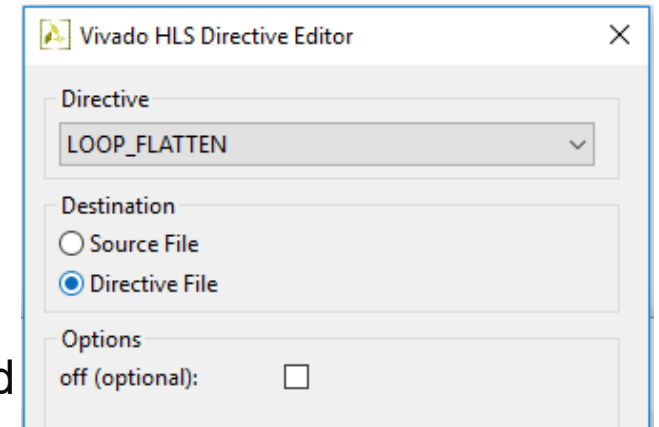
Effective code after
compiler
transformation

```
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

- > **Vivado 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



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]  
    }  
}
```

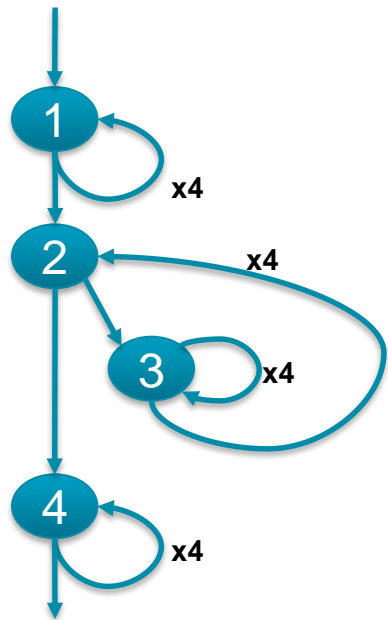
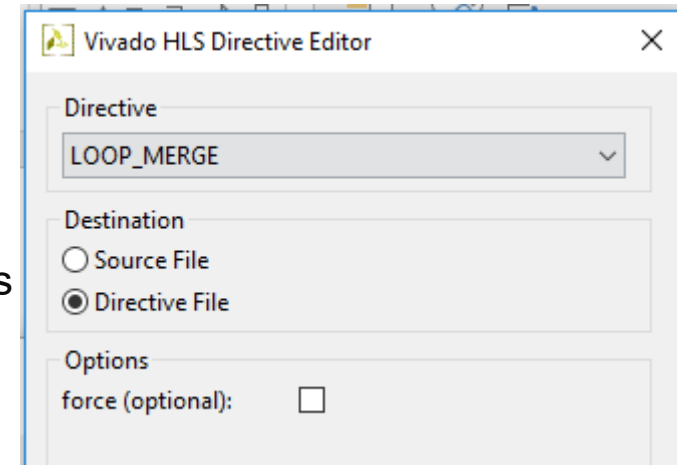
```
Loop_outer: for (i=3;i>N;i--) {  
    Loop_inner: for (j=3;j>=0;j--) {  
        [Loop body]  
    }  
}
```

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


Loop Merging

> Vivado 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



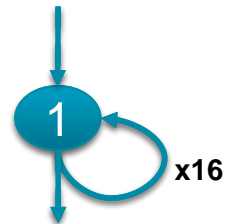
36 transitions

```
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 ]  
    }  
}
```



18 transitions

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

- > **Vivado HLS reports the latency of loops**
 - >> Shown in the report file and GUI
- > **Given a variable loop index, the latency cannot be reported**
 - >> Vivado 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

Performance Estimates

▣ **Timing (ns)**

▣ **Summary**

Clock	Target	Estimated	Uncertainty
default	10.00	8.74	1.25

▣ **Latency (clock cycles)**

▣ **Summary**

Latency		Interval		
min	max	min	max	Type
?	?	?	?	none



Vivado HLS Directive Editor

Directive
LOOP_TRIPCOUNT

Destination
☒ Source File
☐ Directive File

Options
min (optional): 200
max (optional): 1280
average (optional):



Performance Estimates

▣ **Timing (ns)**

▣ **Summary**

Clock	Target	Estimated	Uncertainty
default	10.00	8.74	1.25

▣ **Latency (clock cycles)**

▣ **Summary**

Latency		Interval		
min	max	min	max	Type
561205	34417925	561206	34417926	none

Techniques for Minimizing Latency - Summary

> Constraints

- >> Vivado 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

Improving Throughput



Improving Throughput

> 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);  
}
```



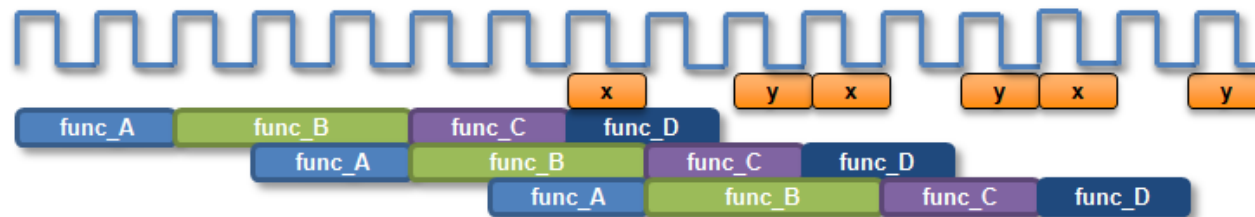
> Vivado HLS will schedule the design



The latency is 9 cycles

The throughput is also 9 cycles

> It can also automatically optimize the dataflow for throughput



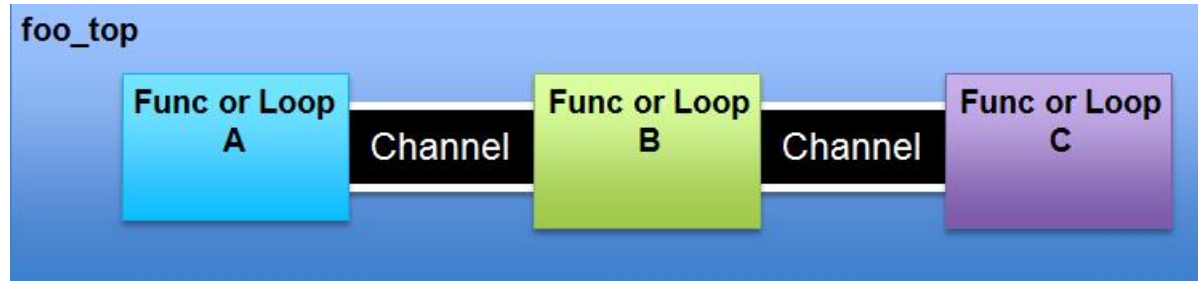
The latency is still 9 cycles

The throughput is now 3

Dataflow Optimization

> Dataflow Optimization

- >> Can be used at the top-level function
- >> Allows blocks of code to operate concurrently
 - The blocks can be functions or loops
 - Dataflow allows loops to operate concurrently
- >> It places channels between the blocks to maintain the data rate



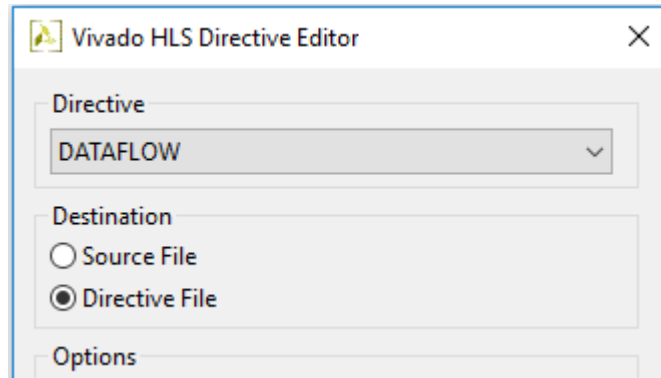
- For arrays the channels will include memory elements to buffer the samples
- For scalars the channel is a register with hand-shakes

> Dataflow optimization therefore has an area overhead

- >> Additional memory blocks are added to the design
- >> The timing diagram on the previous page should have a memory access delay between the blocks
 - Not shown to keep explanation of the principle clear

Dataflow Optimization Commands

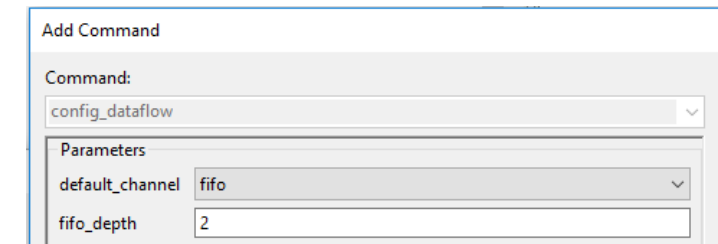
- > **Dataflow is set using a directive**
 - >> Vivado HLS will seek to create the highest performance design
 - Throughput of 1



Dataflow Optimization through Configuration Command

> Configuring Dataflow Memories

- >> Between functions Vivado HLS uses ping-pong memory buffers by default
 - The memory size is defined by the maximum number of producer or consumer elements
- >> Between loops Vivado HLS will determine if a FIFO can be used in place of a ping-pong buffer
- >> The memories can be specified to be FIFOs using the Dataflow Configuration
 - Menu: Solution > Solution Settings > config_dataflow
 - With FIFOs the user can override the default size of the FIFO
 - Note: Setting the FIFO too small may result in an RTL verification failure



> Individual Memory Control

- >> When the default is ping-pong
 - Select an array and mark it as Streaming (directive STREAM) to implement the array as a FIFO
- >> When the default is FIFO
 - Select an array and mark it as Streaming (directive STREAM) with option “off” to implement the array as a ping-pong

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.

Dataflow : Ideal for streaming arrays & multi-rate functions

> Arrays are passed as single entities by default

>> This example uses loops but the same principle applies to functions

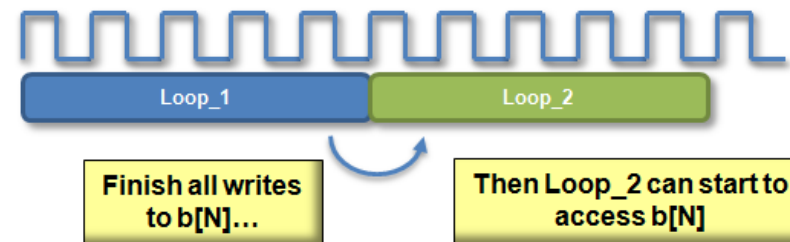
```
int a[N], b[N], c[N];
```

```
Loop_1: for (i=0; i<=N-1; i++) {  
    b[i] = a[i] + in1;  
}
```

Loop_1

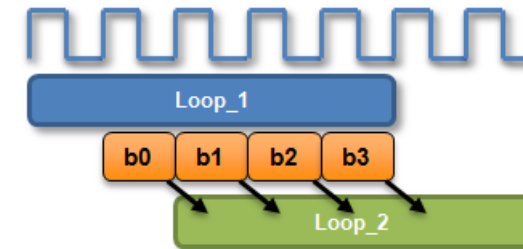
```
Loop_2: for (i=0; i<=N-1; i++) {  
    c[i] = b[i] * in2;  
}
```

Loop_2



> Dataflow pipelining allows loop_2 to start when data is ready

- >> The throughput is improved
- >> Loops will operate in parallel
- If dependencies allow



> Multi-Rate Functions

>> Dataflow buffers data when one function or loop consumes or produces data at different rate from others

> IO flow support

>> To take maximum advantage of dataflow in streaming designs, the IO interfaces at both ends of the datapath should be streaming/handshake types (ap_hs or ap_fifo)

Dataflow Limitations (1)

- > 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;  
    }  
}
```

Dataflow Limitations (2)

- > 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];  
    }  
}
```

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

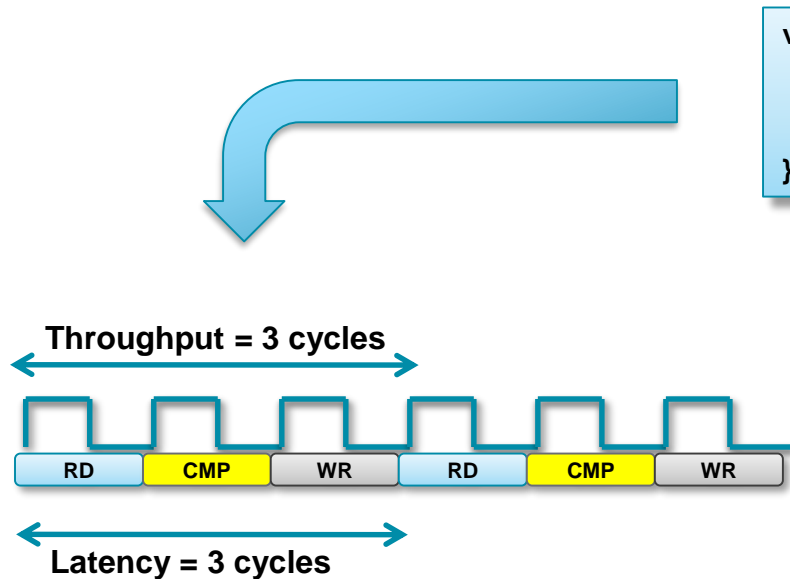
> Function & Loop Pipelining

- >> “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

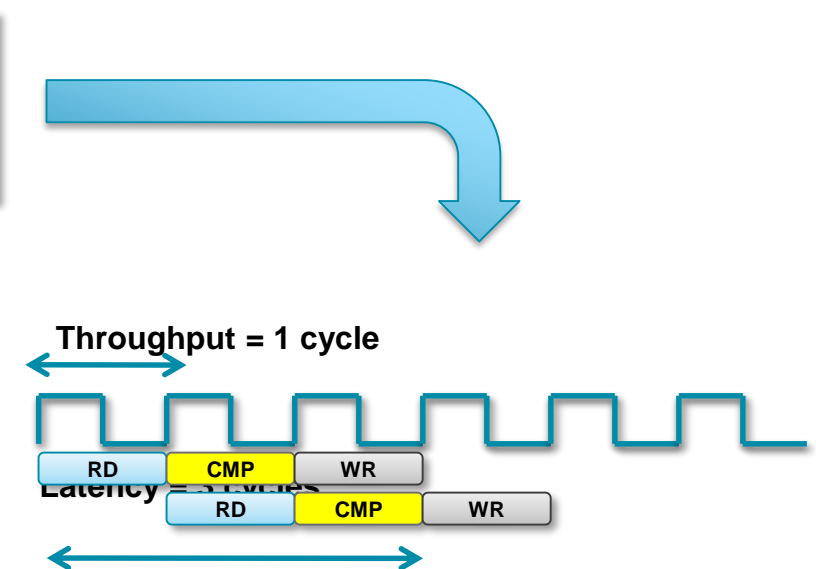
Function Pipelining

- > 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
 - >> Less cycles, higher throughput
- > The throughput is better

Without Pipelining

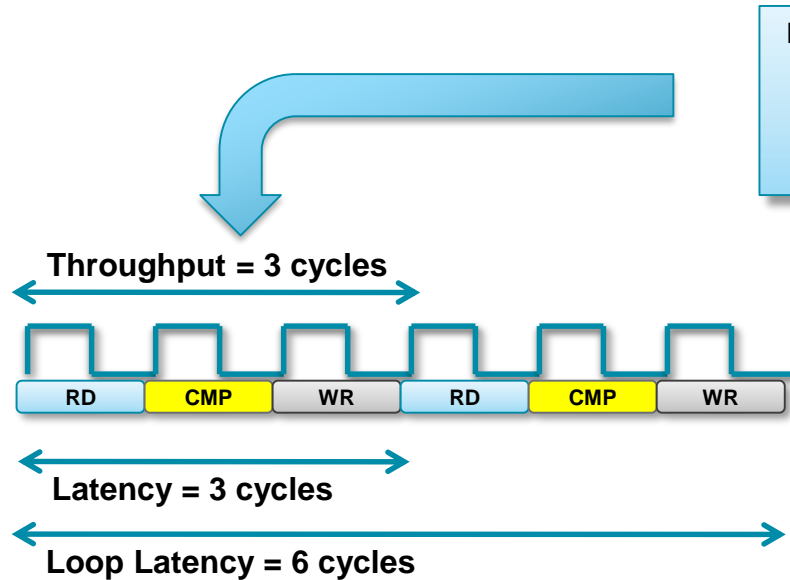


With Pipelining

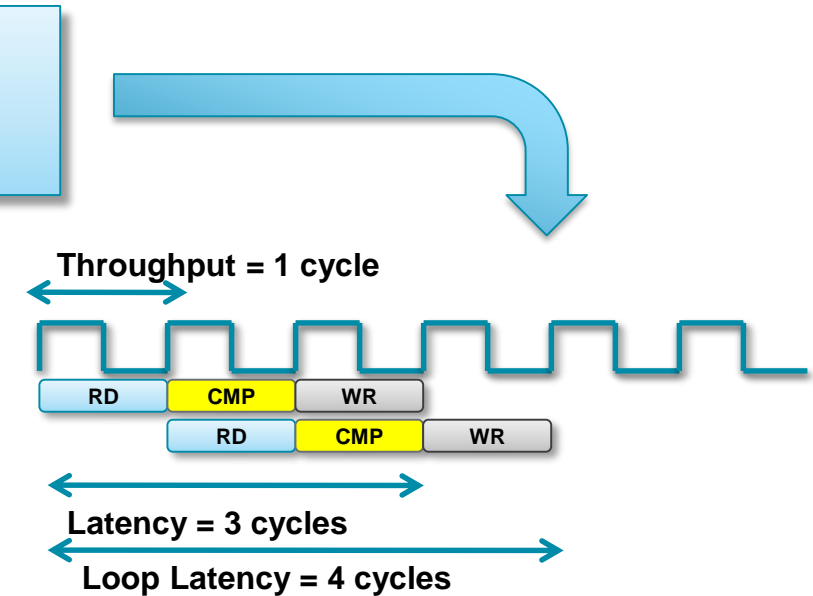


Loop Pipelining

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

Pipelining and Function/Loop Hierarchy

- > **Vivado 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
 - Or next one (or two) out if inner-most is modest and fixed
 - e.g. Convolution algorithm
 - 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

Pipeline Directive

> The pipeline directive pipelines functions or loops

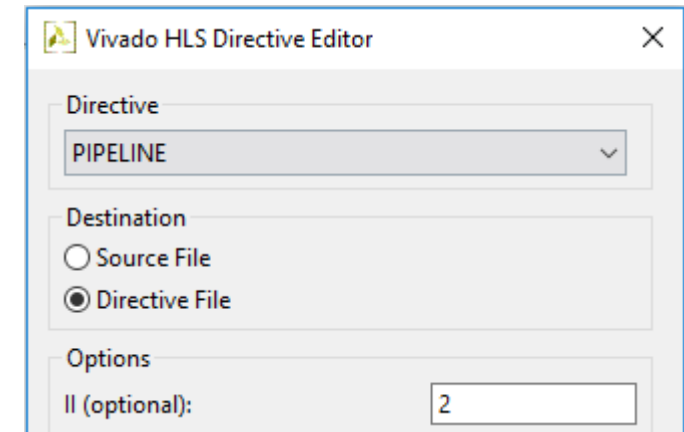
- >> This example pipelines the function with an Initiation Interval (II) of 2
 - The II is the same as the throughput but this term is used exclusively with pipelines



> Omit the target II and Vivado HLS will Automatically pipeline for the fastest possible design

- >> Specifying a more accurate maximum may allow more sharing (smaller area)

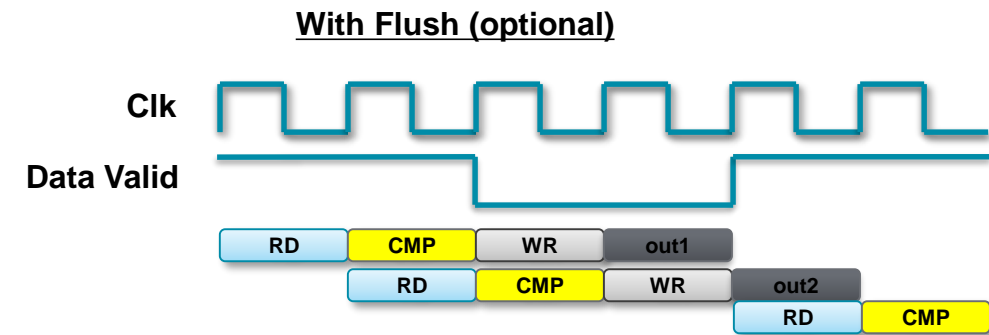
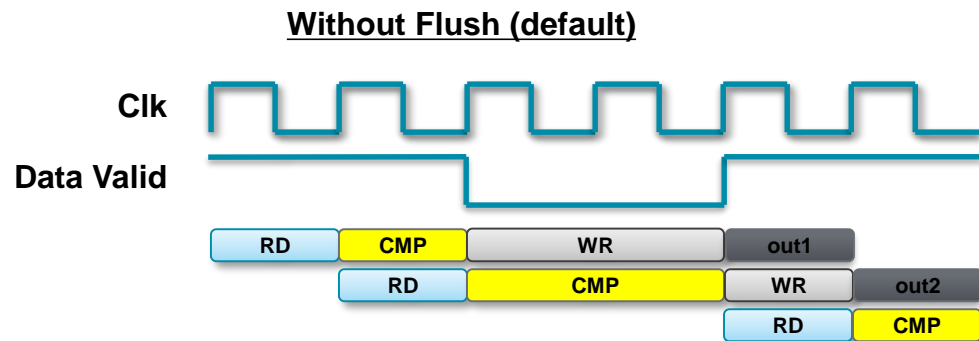
> The directive on loops provides loop rewinding option



Pipeline Flush

> Pipelines can optionally be flushed

- >> Flush: when the input enable goes low (no more data) all existing results are flushed out
 - The input enable may be from an input interface or from another block in the design
- >> The default is to stall all existing values in the pipeline



> With Flush

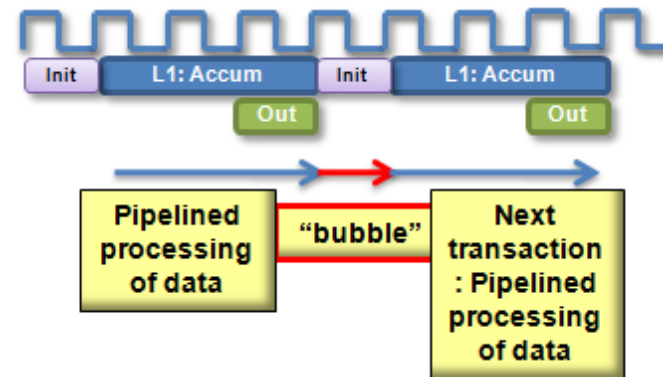
- >> When no new input reads are performed
- >> Values already in the pipeline are flushed out

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;  
    ...  
}
```

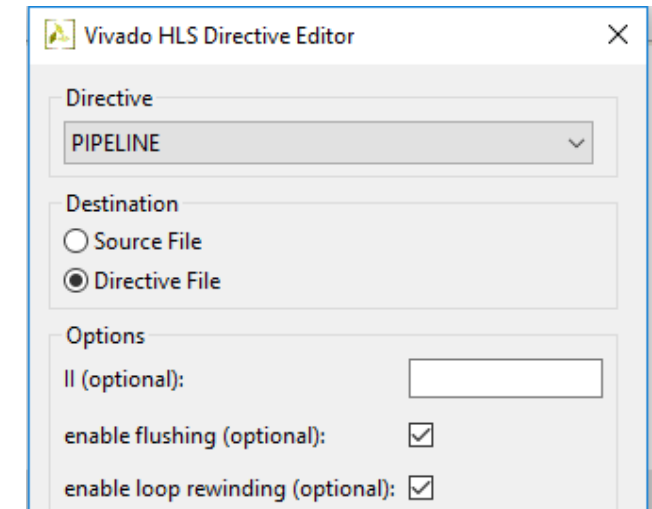
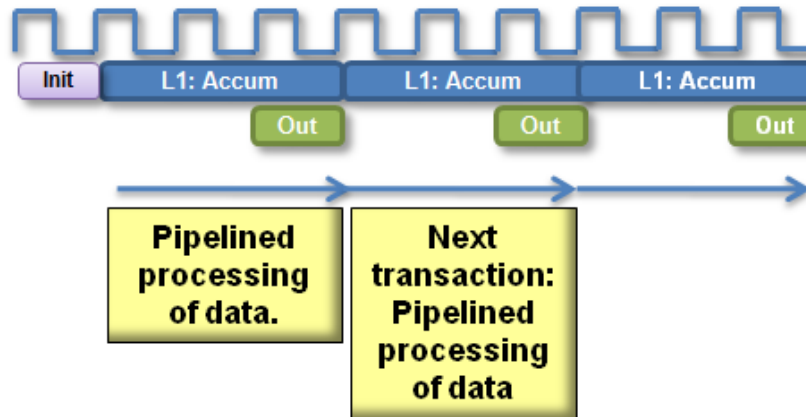


- >> The function will process a stream of data
- >> The next time the function is called, it still needs to execute the initial (init) operations
 - These operations are any which occur before the loop starts
 - These operations may include interface start/stop/done signals
- >> This can result in an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

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
 - Ignores any initialization statements before 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;  
    ...  
}
```



- > The rewind portion only effects top-level loops
 - >> Ensures the operations before the loop are never re-executed when the function is re-executed

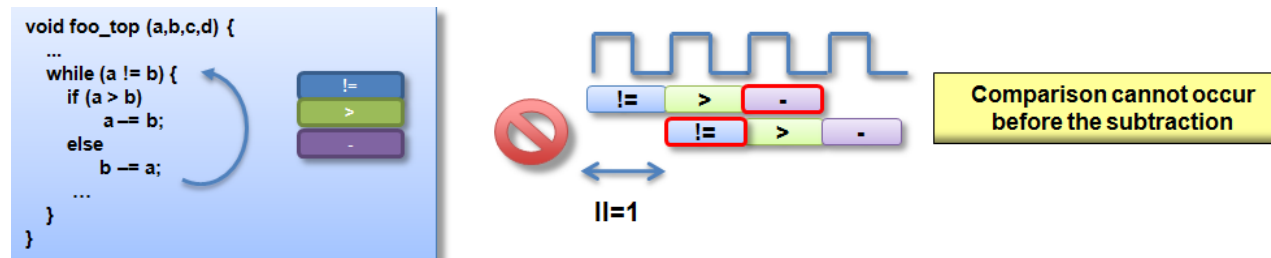
Issues which prevent Pipelining

> 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)



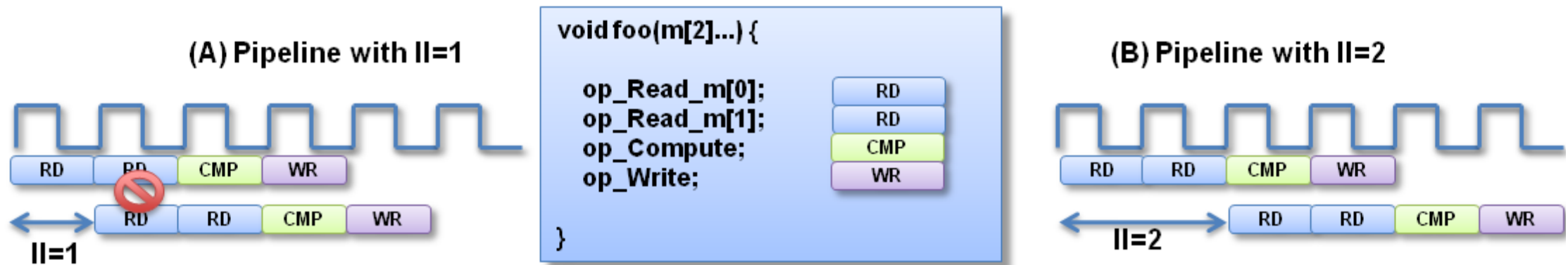
> Resource Contention may prevent pipelining

- >> 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

> Vivado HLS will automatically increase the II

>> Vivado HLS will always try to create a design, even if constraints must be violated

Performance Bottleneck



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
 - May not be the most ideal memory for performance

```
void foo_top (...) {  
    ...  
    for (i = 2; i < N; i++)  
        mem[i] = mem[i-1] + mem[i-2];  
}  
}
```



- Cannot pipeline with a throughput of 1



Or



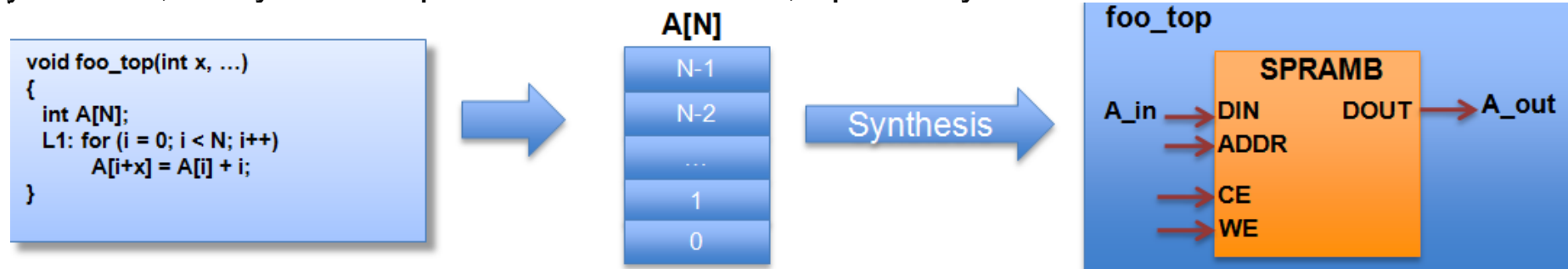
Even with a dual-port RAM, we cannot perform all reads and writes in one cycle

- > **Vivado HLS allows arrays to be partitioned and reshaped**
 - >> Allows more optimal configuration of the array
 - >> Provides better implementation of the memory resource

Review: Arrays in HLS

> An array in C code is implemented by a memory in the RTL

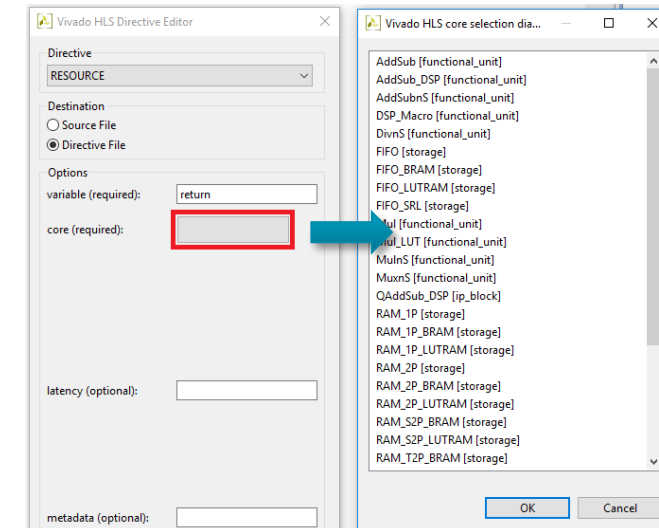
>> By default, arrays are implemented as RAMs, optionally a FIFO



> The array can be targeted to any memory resource in the library

>> The ports and sequential operation are defined by the library model

- All RAMs are listed in the Vivado HLS Library Guide



List of available Cores

Array and RAM selection

> If no RAM resource is selected

- >> Vivado HLS will determine the RAM to use
 - It will use a Dual-port if it improves throughput
 - Else it will use a single-port

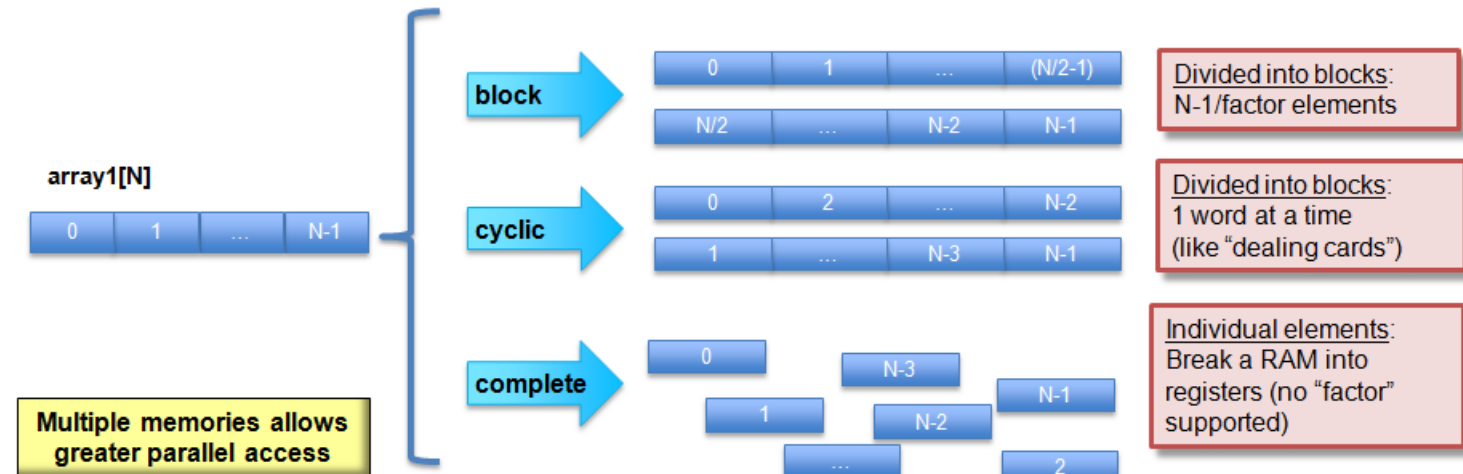
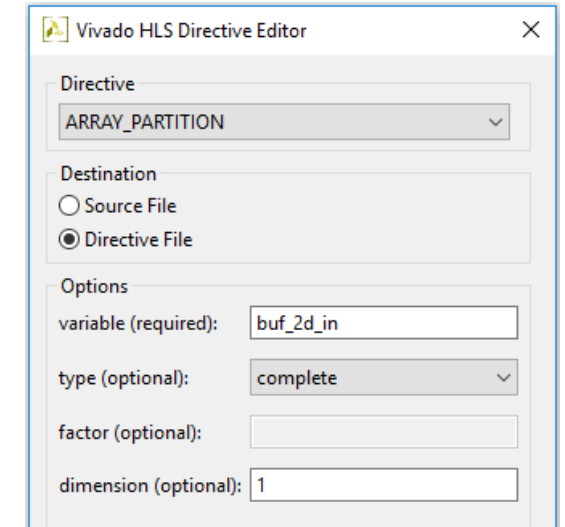
> BRAM and LUTRAM selection

- >> If none is made (e.g. resource RAM_1P used) RTL synthesis will determine if RAM is implemented as BRAM or LUTRAM
- >> If the user specifies the RAM target (e.g. RAM_1P_BRAM or RAM_1P_LUTRAM is selected) Vivado HLS will obey the target
 - If LUTRAM is selected Vivado HLS reports registers not BRAM

Array Partitioning

> Partitioning breaks an array into smaller elements

- >> If the factor is not an integer multiple the final array has fewer elements
- >> Arrays can be split along any dimension
 - If none is specified dimension zero is assumed
 - Dimension zero means all dimensions
- >> All partitions inherit the same resource target
 - That is, whatever RAM is specified as the resource target
 - Except of course “complete”



Configuring Array Partitioning

> Vivado 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

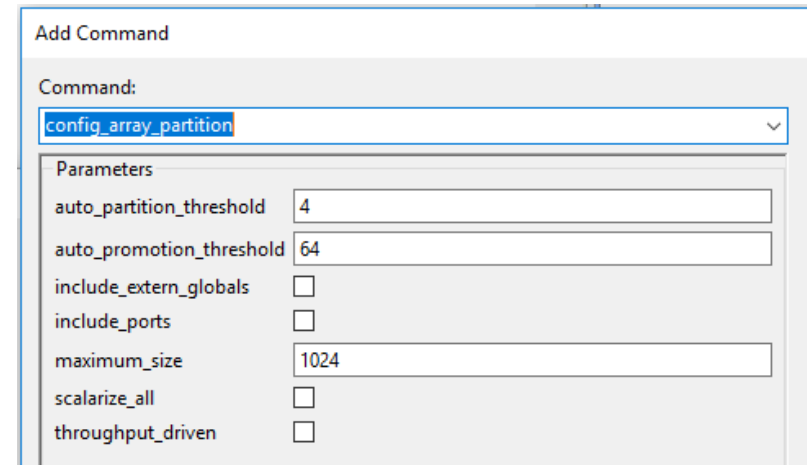
- >> When the array index is not a variable
- >> Arrays below the threshold are auto-partitioned
- >> Set the threshold using option elem_count_limit

> Partition all arrays in the design

- >> Select option scalarize_all

> Include all arrays in partitioning

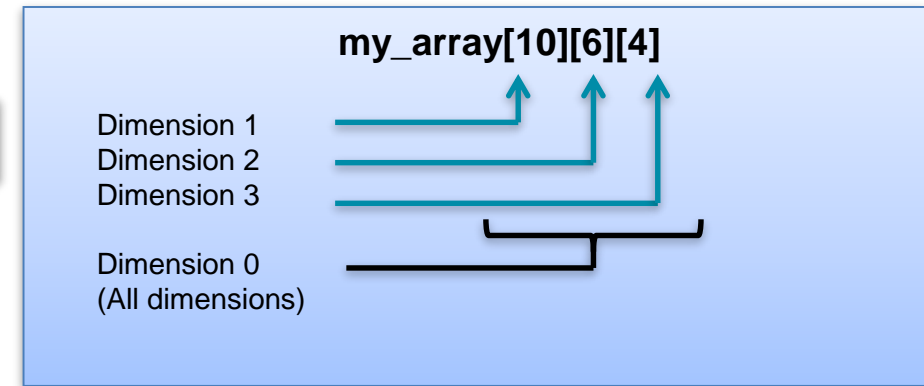
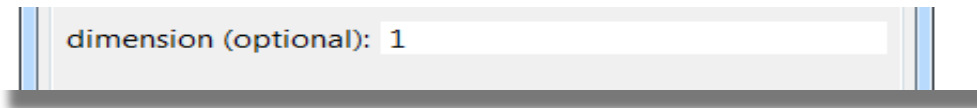
- >> The include_ports option will include any arrays on the IO interface when partitioning is performed
 - Partitioning these arrays will result in multiple ports and change the interface
 - This may however improve throughput
- >> Any arrays defined as a global can be included in the partitioning by selecting option include_extern_globals
 - By default, global arrays are not partitioned



Add Command	
Command:	config_array_partition
Parameters	
auto_partition_threshold	4
auto_promotion_threshold	64
include_extern_globals	<input type="checkbox"/>
include_ports	<input type="checkbox"/>
maximum_size	1024
scalarize_all	<input type="checkbox"/>
throughput_driven	<input type="checkbox"/>

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 →

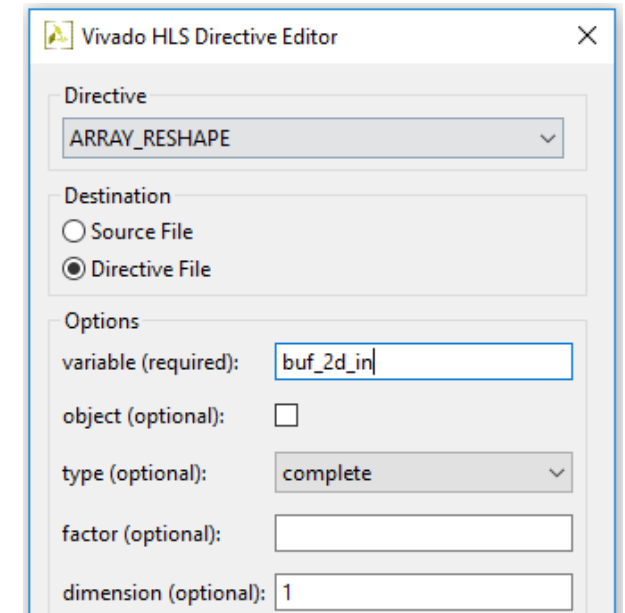
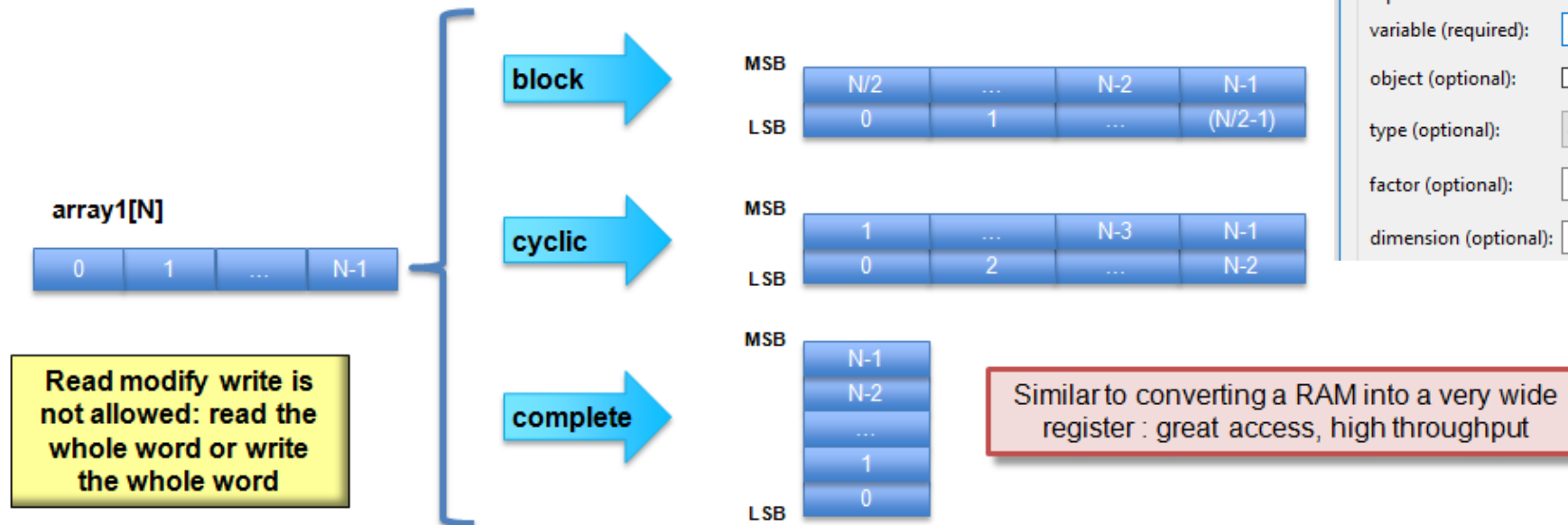
my_array_0[6][4]
my_array_1[6][4]
my_array_2[6][4]
my_array_3[6][4]
my_array_4[6][4]
my_array_5[6][4]
my_array_6[6][4]
my_array_7[6][4]
my_array_8[6][4]
my_array_9[6][4]

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

Array Reshaping

> Reshaping recombines partitioned arrays back into a single array

- >> Same options as array partition
- >> However, reshape automatically recombines the parts back into a single element
- >> The “new” array has the same name
 - Same name used for resource targeting



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**

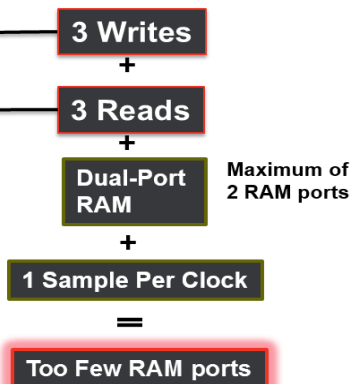
```
INFO: [SCHED 61] Pipelining loop 'SUM_LOOP'.  
WARNING: [SCHED 69] Unable to schedule 'load' operation ('mem_load_2', bottleneck.c:62) on  
array 'mem' due to limited memory ports.  
INFO: [SCHED 61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
```

Bottleneck Example

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

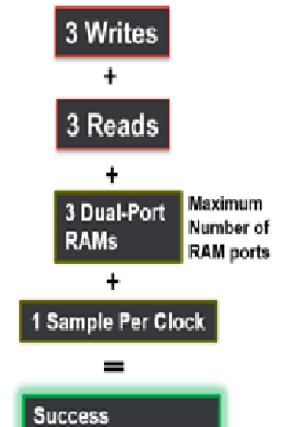
```
@I [SCHED-61] Pipelining loop 'I SOBEL_LOOP'.  
@W [SCHED-69] Unable to schedule 'store' operation (image_demo.cpp:172) of  
variable 'y' on array 'buff_A' due to limited resources (II = 2).  
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 3, Depth: 13.
```

```
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;  
            }  
            ...  
        }  
    }  
}
```



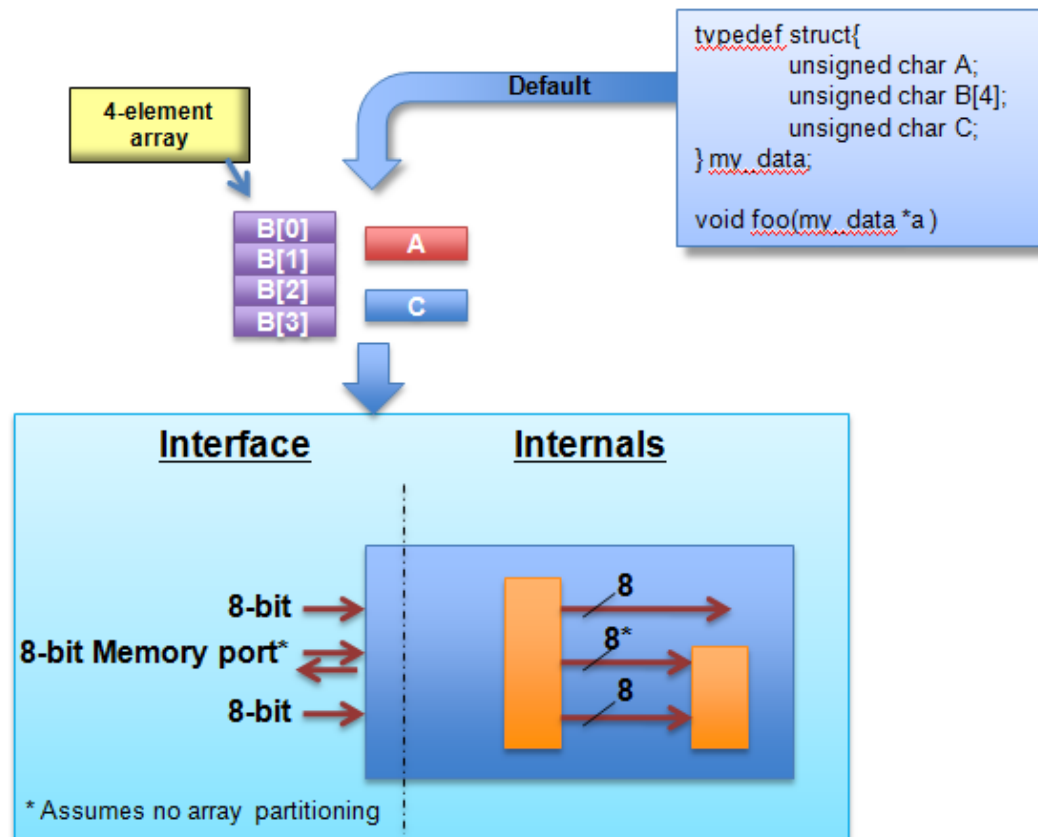
Solution: Use ARRAY_PARTITION directive

```
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=tempA complete dim=1  
    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;  
            }  
            ...  
        }  
    }  
}
```



Structs and Arrays: The Default Handling

- > **Structs are a commonly used coding construct**
 - >> By default, structs are separated into their separate elements

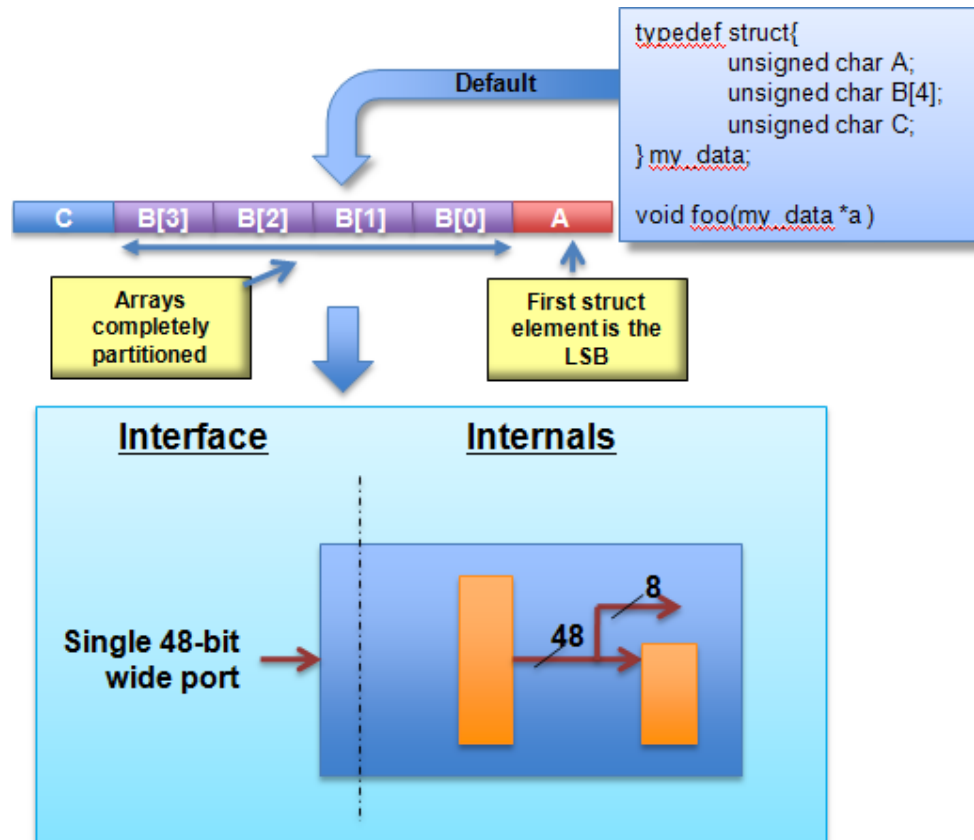


- **Treated as separate elements**
- **On the Interface**
 - This means separate ports
- **Internally**
 - Separate buses & wires
 - Separate control logic, which may be more complex, slower and increase latency
- **Use the DATA_PACK directive to group them into a single element**

Data Packing

> Data packing groups structs internally and at the IO Interface

>> Creates a single wide bus of all struct elements



• Grouped structure

- First element in the struct becomes the LSB
- Last struct element becomes the MSB
- Arrays are partitioning completely

• On the Interface

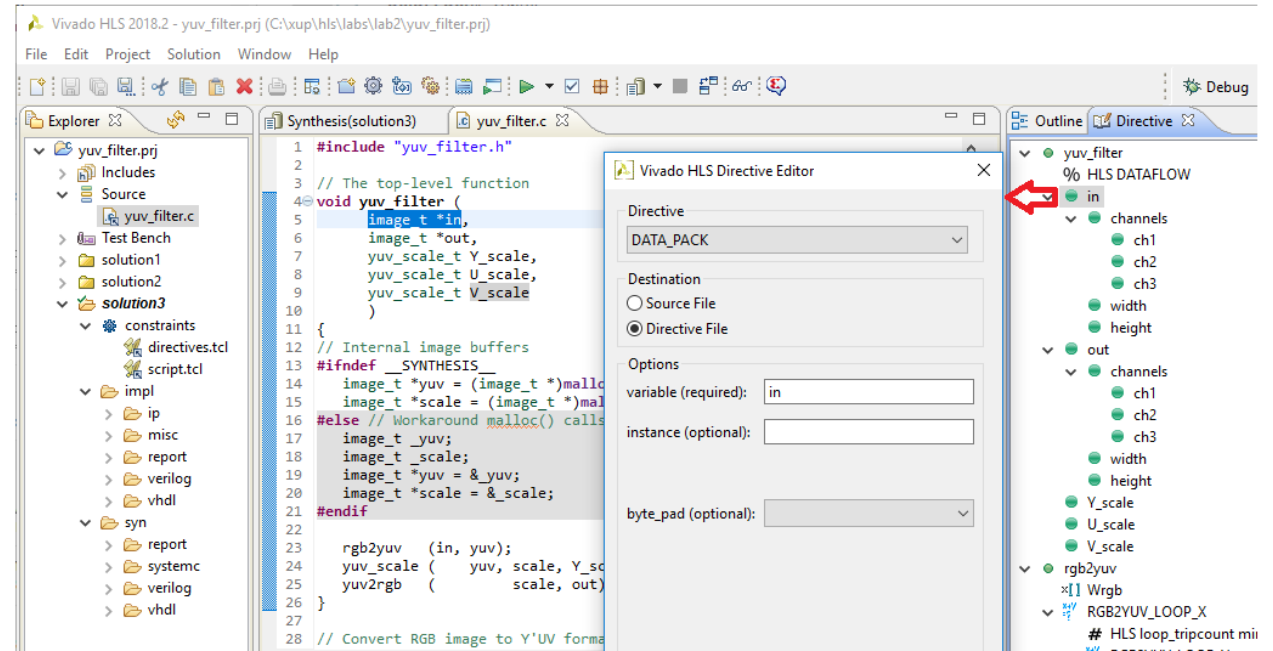
- This means a single port

• Internally

- Single bus
- May result in simplified control logic, faster and lower latency designs

Using Data Pack Directive

- > Apply the DATA_PACK directive
 - >> Select and specify the struct variable to pack



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

>> Latency optimization

- Specify latency directives
- Unroll loops
- Merge and Flatten loops to reduce loop transition overheads

>> 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

>> 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