

---

# **Vitis HLS Design Optimizations and Model Composer Integration**

Joseph Cavallaro  
Rice University  
6 October 2022

# Last Lecture

---

- ❑ Project 3 – Vitis HLS for Matrix Multiplication
- ❑ Resources for Project 3 on Canvas
  - ⇒ Sample tutorial files in “lab 1” as starting point for Project 3
- ❑ Vitis HLS Pragmas for pipelining and scheduling

# Today

---

- ❑ Continue discussion of Performance Optimizations from where we left off on Improving Throughput
- ❑ Additional Project 3 notes, examples, and materials on Canvas.

# Improving Throughput

## ➤ Given a design with multiple functions

- The code and dataflow are as shown

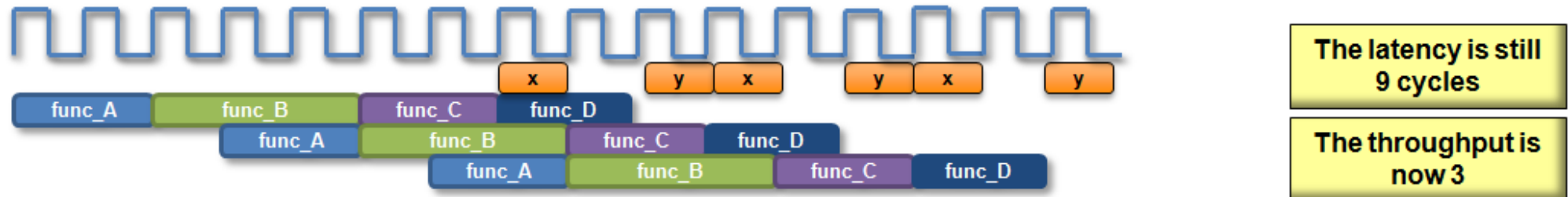
```
void foo_top (a,b,c,d, *x, *y) {  
    ...  
    func_A(a,b,t1);  
    func_B(a,t1,t2);  
    func_C(c,t2,&x)  
    func_D(d,x,&y)  
}
```



## ➤ Vivado HLS will schedule the design



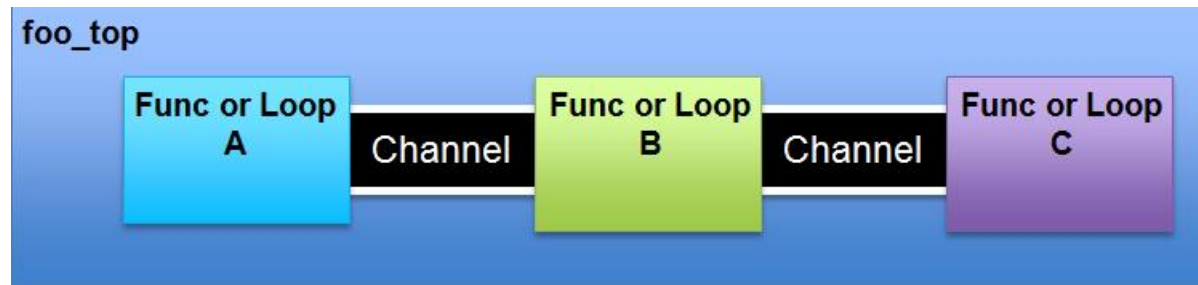
## ➤ It can also automatically optimize the dataflow for throughput



# 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

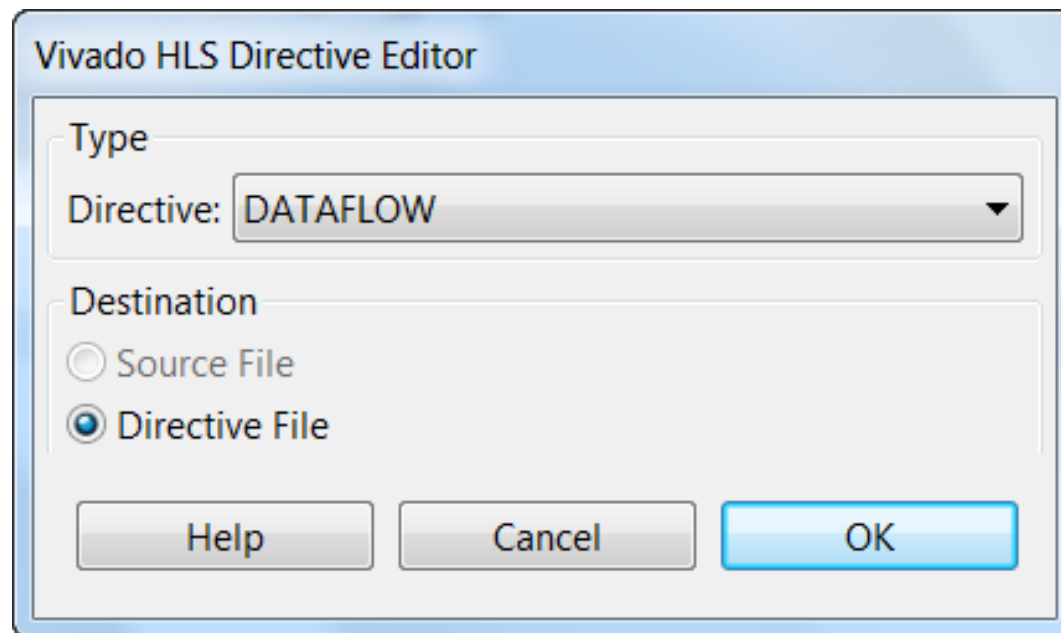
## ➤ Dataflow optimization therefore has an area overhead

- 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

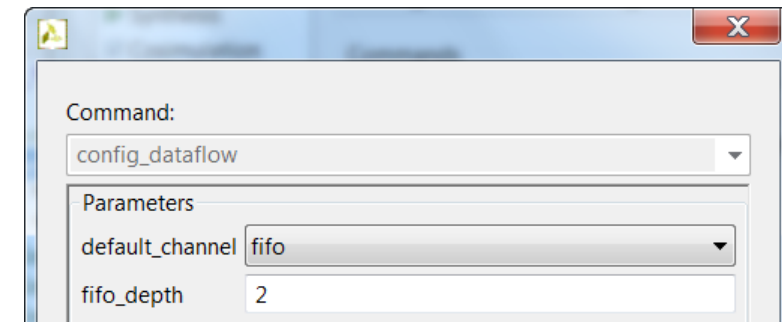
- Vivado HLS will seek to create the highest performance design
  - Throughput of 1



# 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

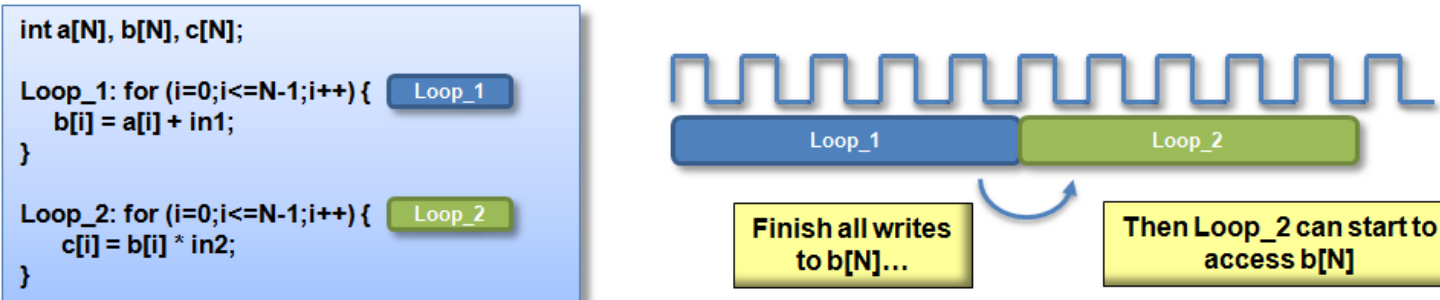
- When the default is ping-pong
  - Select an array and mark it as Streaming (directive STREAM) to implement the array as a FIFO
- When the default is FIFO
  - Select an array and mark it as Streaming (directive STREAM) with option “off” to implement the array as a ping-pong

**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

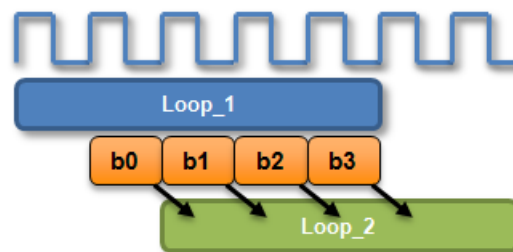
## ➤ Arrays are passed as single entities by default

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



## ➤ 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)



# Pipelining: Dataflow, Functions & Loops

## ➤ Dataflow Optimization

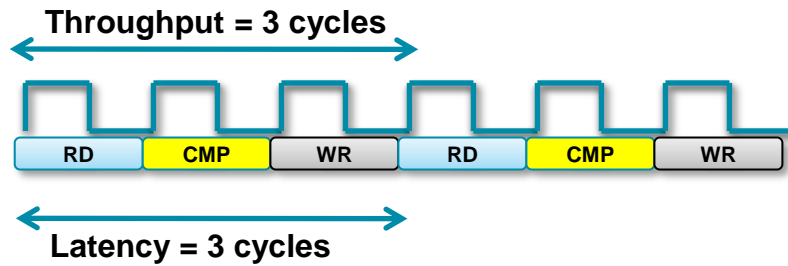
- Dataflow optimization is “coarse grain” pipelining at the function and loop level
- Increases concurrency between functions and loops
- Only works on functions or loops at the top-level of the hierarchy
  - Cannot be used in sub-functions

## ➤ 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

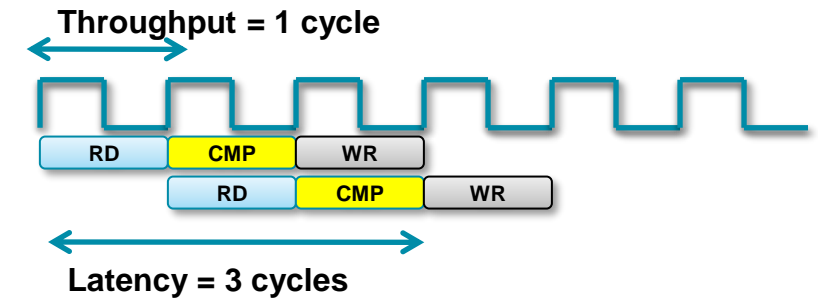
# Function Pipelining

## Without Pipelining



- There are 3 clock cycles before operation RD can occur again
  - Throughput = 3 cycles
- There are 3 cycles before the 1<sup>st</sup> output is written
  - Latency = 3 cycles

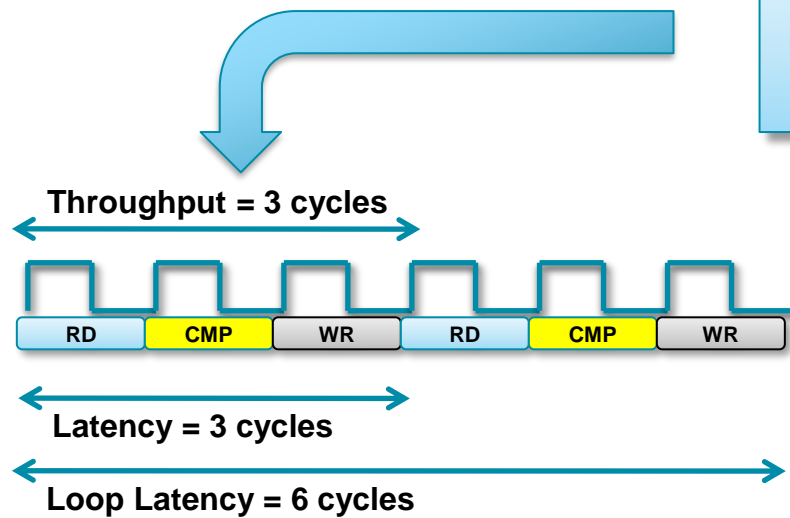
## With Pipelining



- The latency is the same
- The throughput is better
  - Less cycles, higher throughput

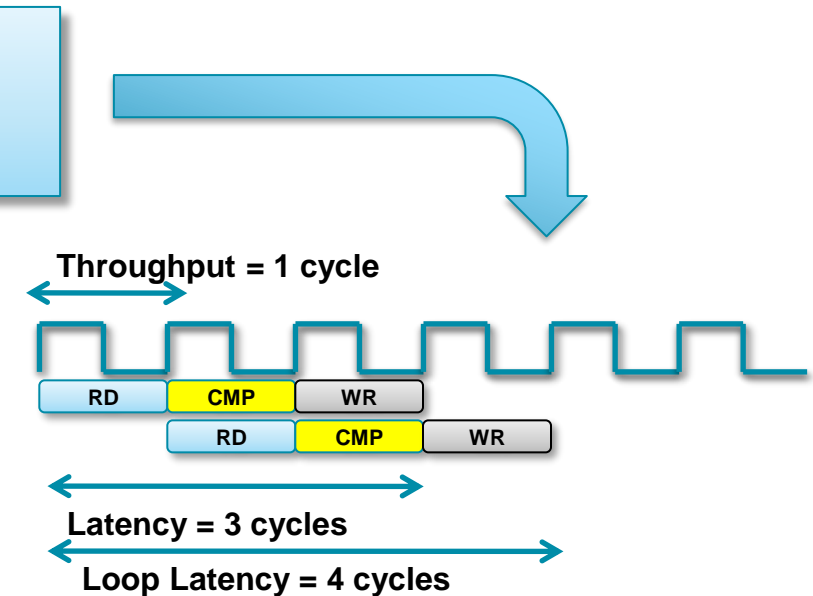
# Loop Pipelining

Without Pipelining



- There are 3 clock cycles before operation RD can occur again
  - Throughput = 3 cycles
- There are 3 cycles before the 1<sup>st</sup> output is written
  - Latency = 3 cycles
  - For the loop, 6 cycles

With Pipelining



- 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 AP 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 AP 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 AP 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

# Pipelining Commands

## ➤ 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

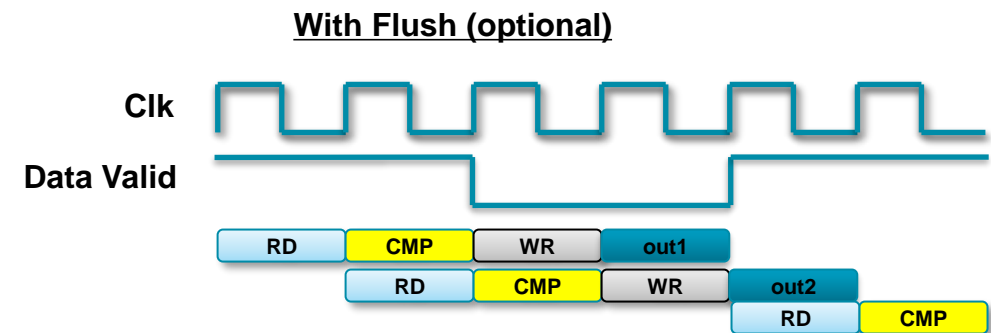
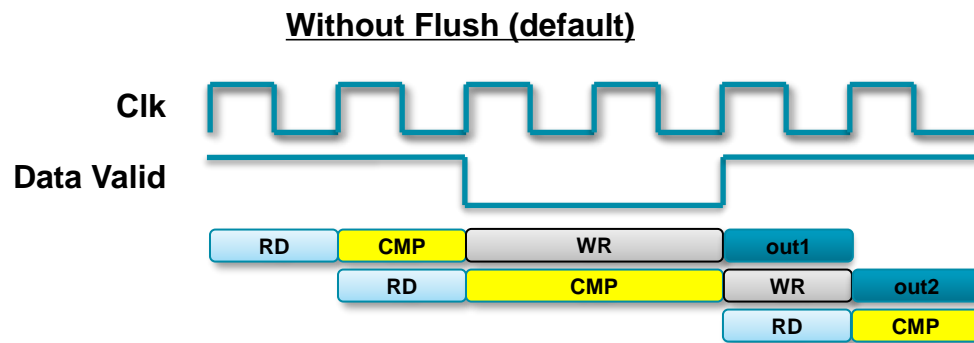
- Specifying a more accurate maximum may allow more sharing (smaller area)

The image shows the 'Vivado HLS Directive Editor' dialog box. It has a title bar and three main sections: 'Type', 'Destination', and 'Options'. In the 'Type' section, 'Directive: PIPELINE' is selected in a dropdown menu. In the 'Destination' section, 'Directive File' is selected with a radio button. In the 'Options' section, 'II (optional):' has a text field with the value '2', and 'enable flushing:' has an unchecked checkbox. At the bottom are 'Help', 'Cancel', and 'OK' buttons.

# 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

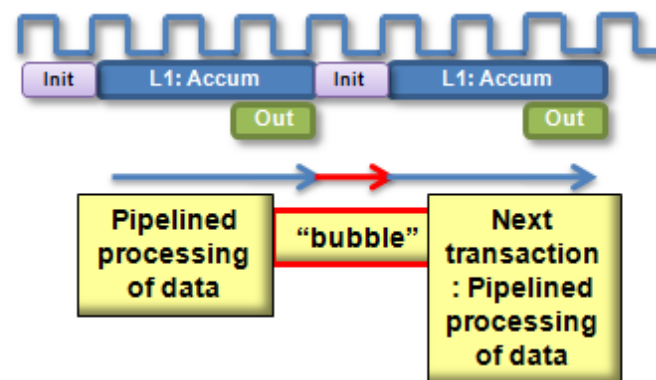
- When no new input reads are performed
- Values already in the pipeline are flushed out

# 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;  
    ...  
}
```



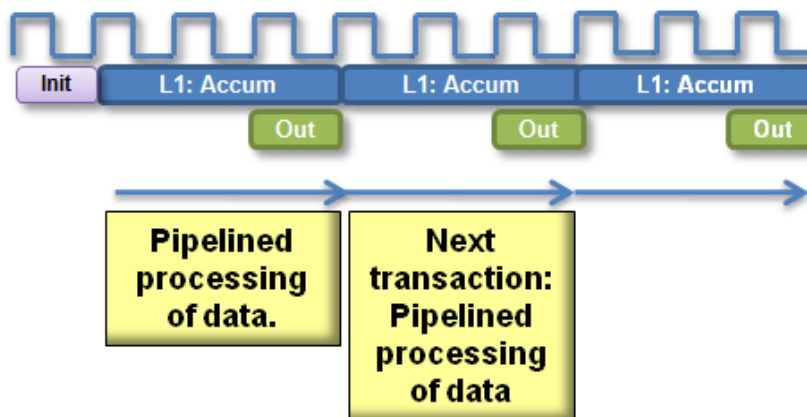
- The function will process a stream of data
- The next time the function is called, it still needs to execute the initial (init) operations
  - These operations are any which occur before the loop starts
  - These operations may include interface start/stop/done signals
- This can result in an unexpected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



Vivado HLS Directive Editor

Type  
Directive: PIPELINE

Destination  
☐ Source File  
☒ Directive File

Options  
II (optional):  
enable flushing: ☒  
enable loop rewinding: ☒

Help Cancel OK

## ➤ The rewind portion only effects top-level loops

- Ensures the operations before the loop are never re-executed when the function is re-executed



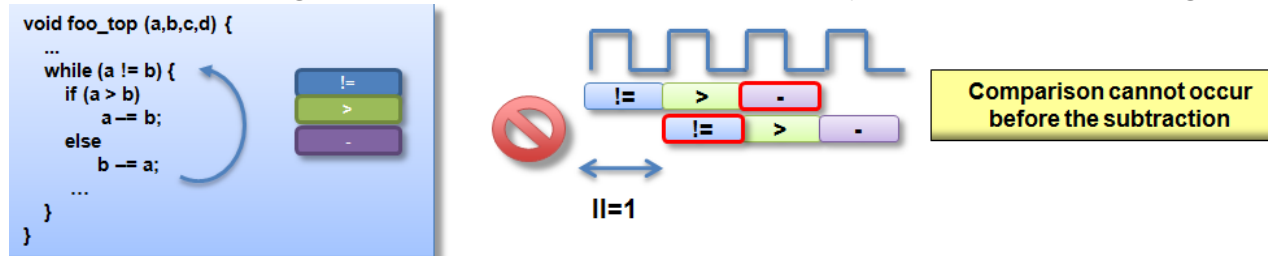
# Issues which prevent Pipelining

## ➤ 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)



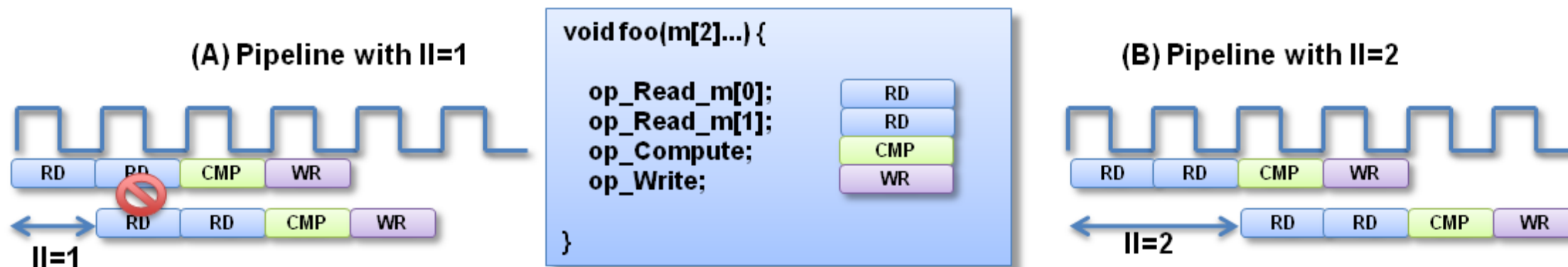
## ➤ Resource Contention may prevent pipelining

- 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

# Outline

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

# 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];  
}
```



Or



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

- Cannot pipeline with a throughput of 1

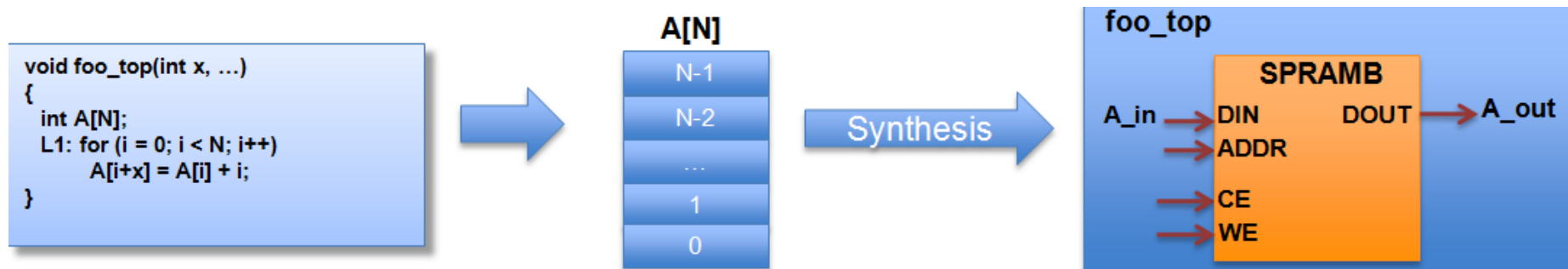
## ➤ Vivado HLS allows arrays to be partitioned and reshaped

- Allows more optimal configuration of the array
- Provides better implementation of the memory resource

# Review: Arrays in HLS

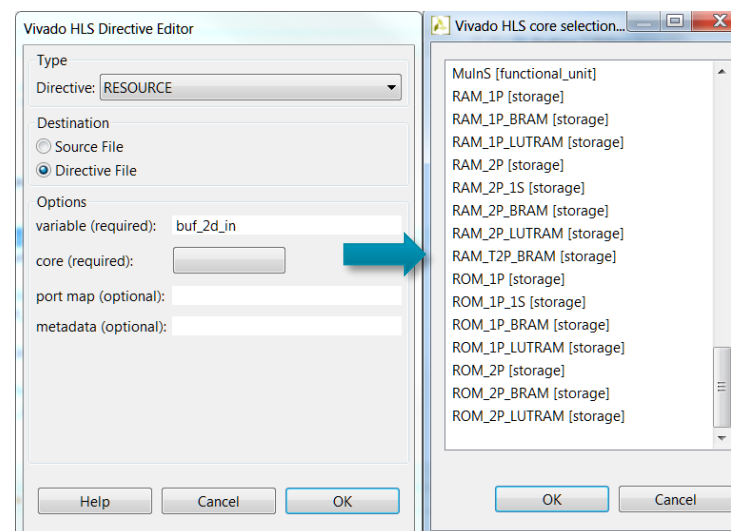
## ➤ An array in C code is implemented by a memory in the RTL

- 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

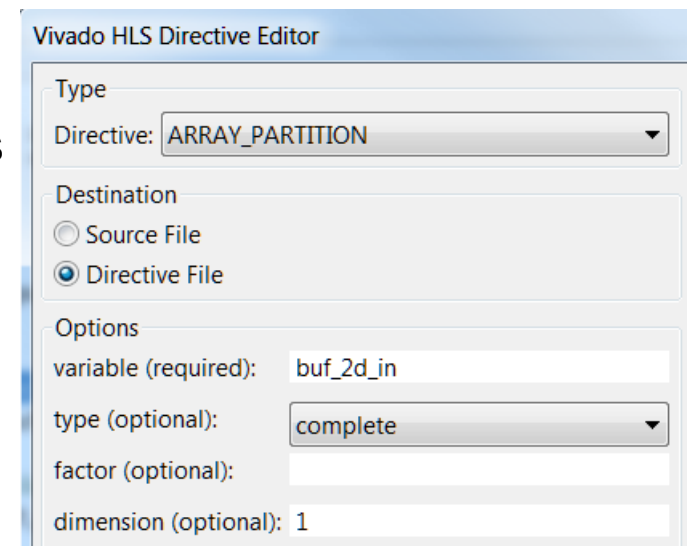
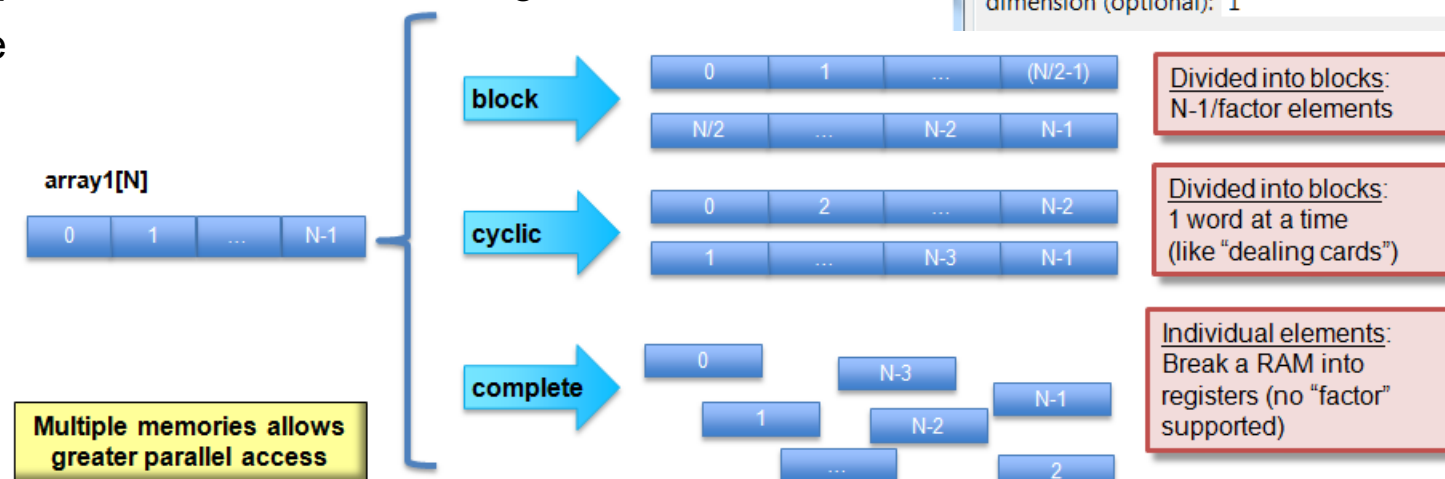
## ➤ BRAM and LUTRAM selection

- 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

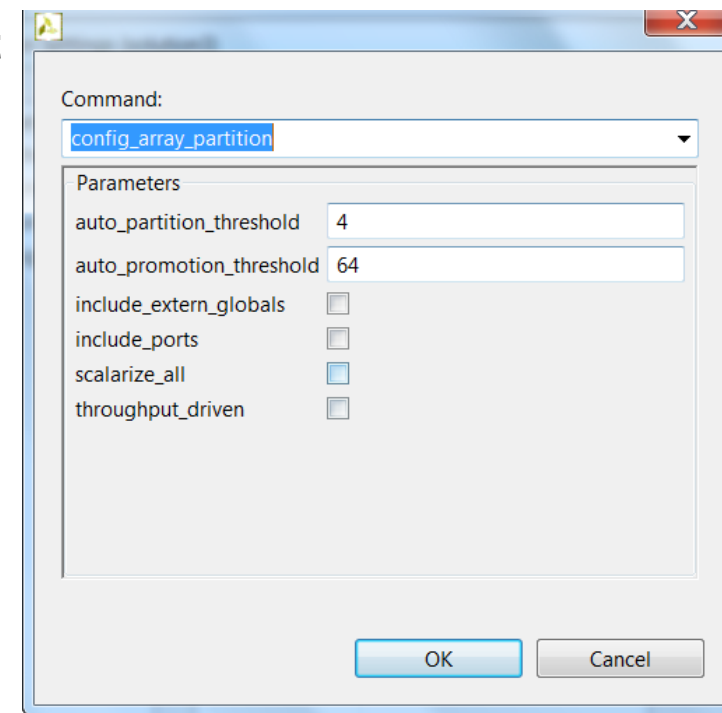
- When the array index is not a variable
- Arrays below the threshold are auto-partitioned
- Set the threshold using option elem\_count\_limit

## ➤ Partition all arrays in the design

- Select option scalarize\_all

## ➤ Include all arrays in partitioning

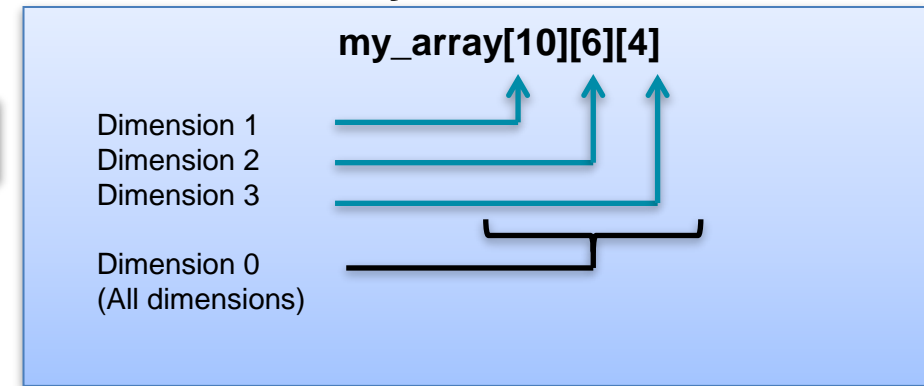
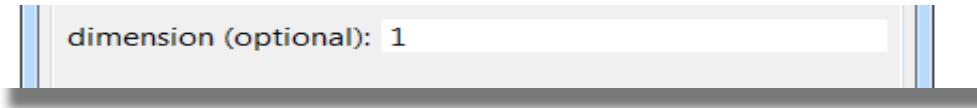
- The include\_ports option will include any arrays on the IO interface when partitioning is performed
  - Partitioning these arrays will result in multiple ports and change the interface
  - This may however improve throughput
- Any arrays defined as a global can be included in the partitioning by selecting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





# 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 ➔

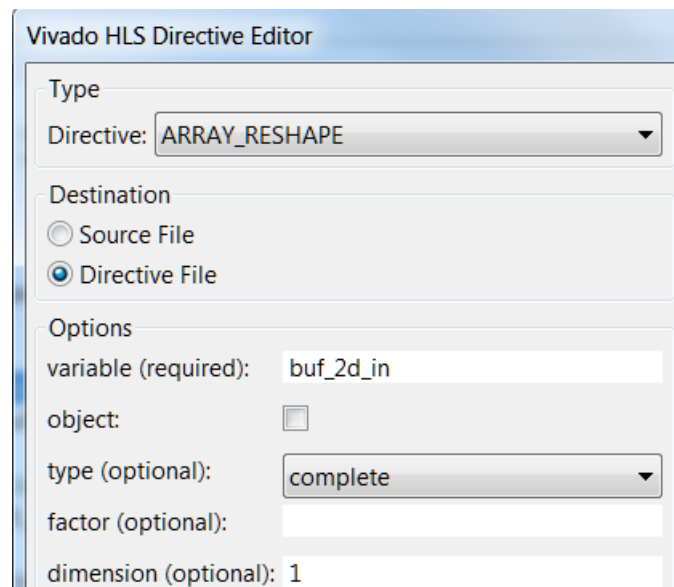
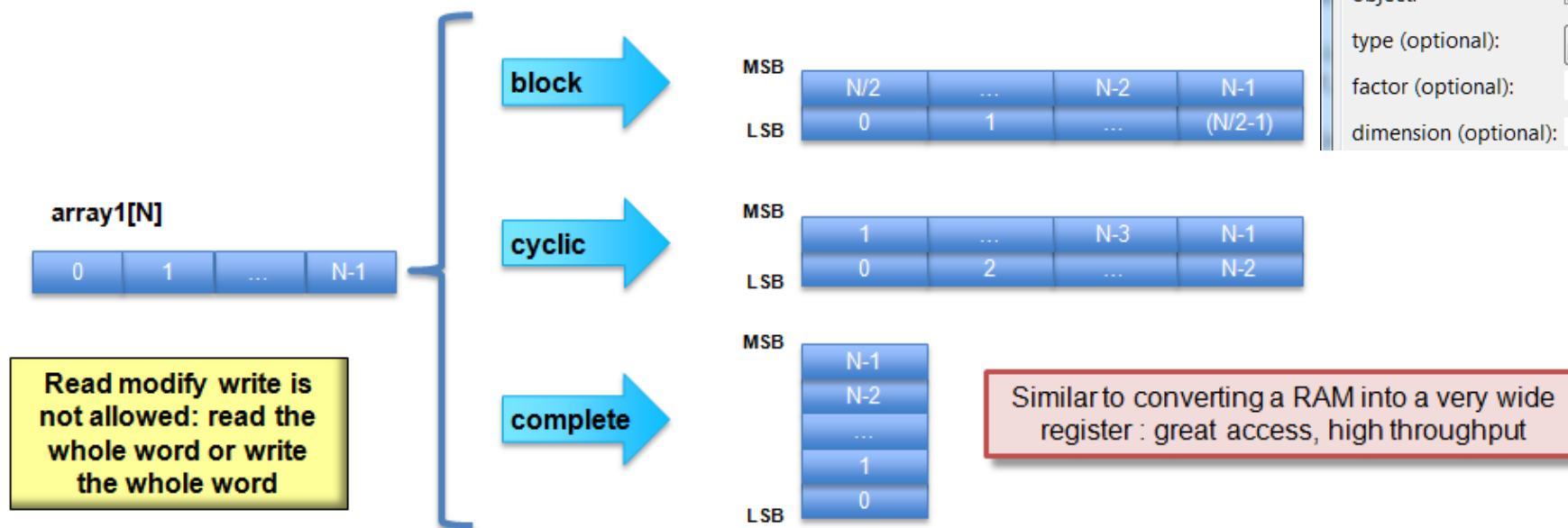
my\_array\_0[6][4]  
my\_array\_1[6][4]  
my\_array\_2[6][4]  
my\_array\_3[6][4]  
my\_array\_4[6][4]  
my\_array\_5[6][4]  
my\_array\_6[6][4]  
my\_array\_7[6][4]  
my\_array\_8[6][4]  
my\_array\_9[6][4]

my\_array[10][6][4] ➔ partition dimension 0 ➔ 10x6x4 = 240 individual registers

# Array Reshaping

## ➤ Reshaping recombines partitioned arrays back into a single array

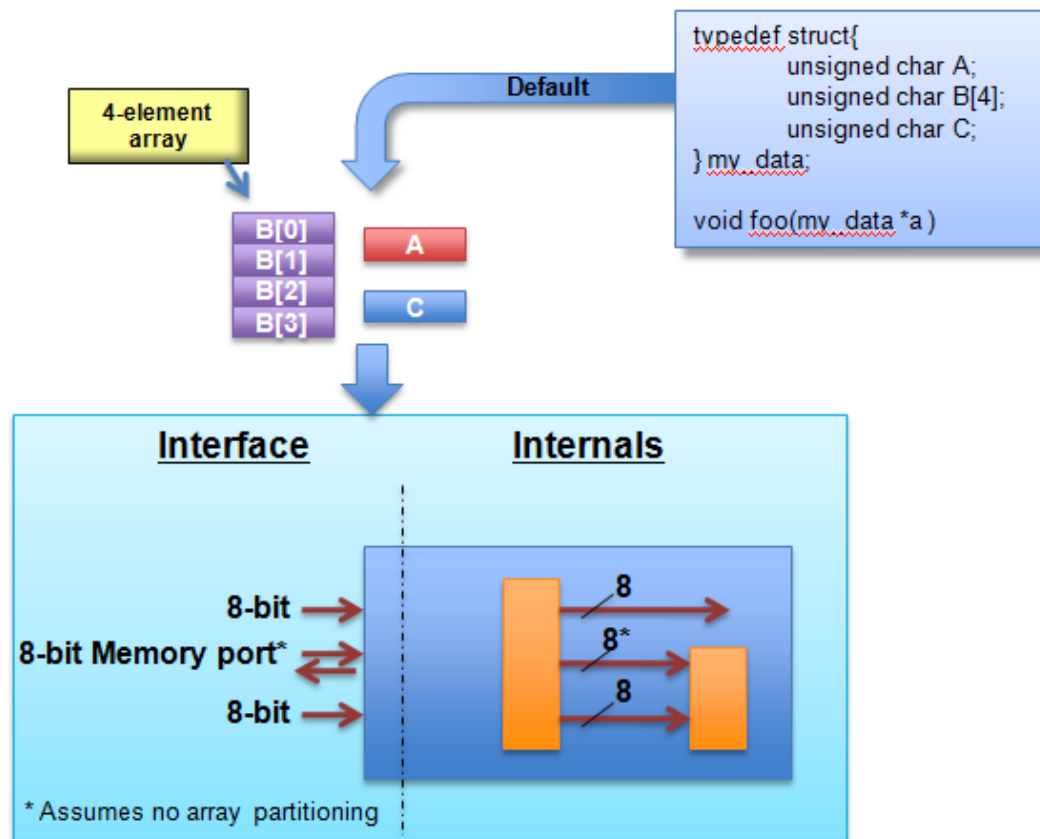
- Same options as array partition
- However, reshape automatically recombines the parts back into a single element
- The “new” array has the same name
  - Same name used for resource targeting



# Structs and Arrays: The Default Handling

## ➤ Structs are a commonly used coding construct

- By default, structs are separated into their separate elements

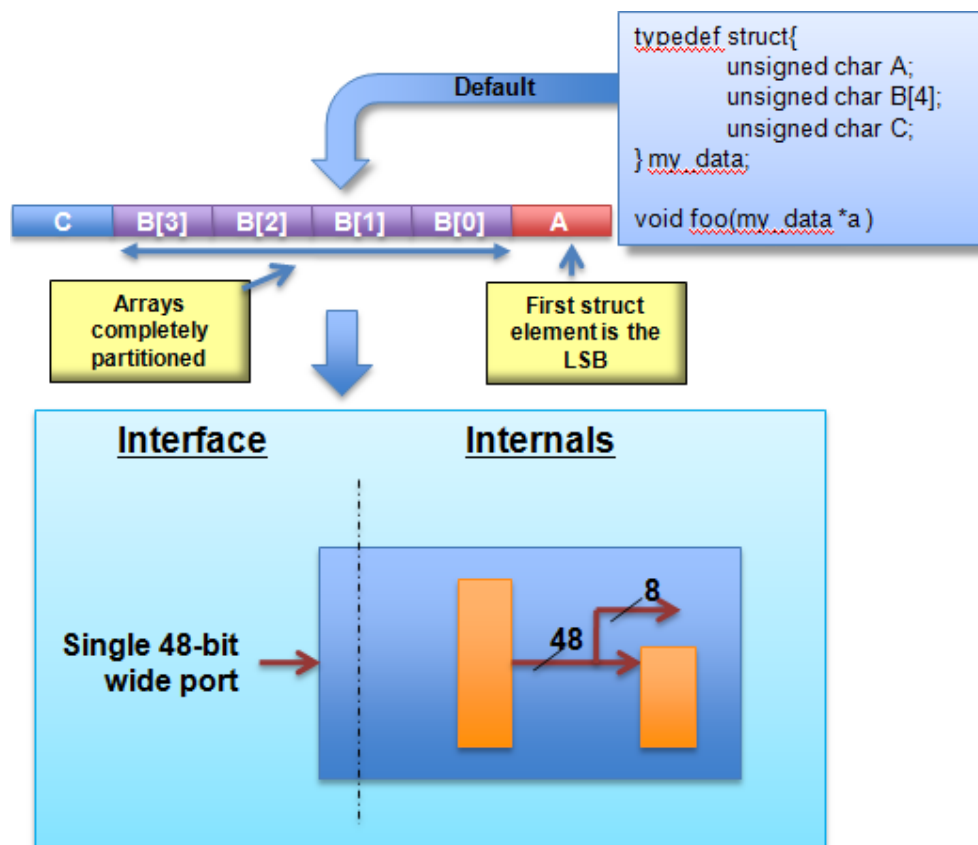


- **Treated as separate elements**
- **On the Interface**
  - This means separate ports
- **Internally**
  - Separate buses & wires
  - Separate control logic, which may be more complex, slower and increase latency

# 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

# Vitis HLS Book on Canvas

---

- ❑ Please read Chapter 7, Matrix Multiplication, for background and discussion on optimizations to help with Project 3.
- ❑ Please also read Chapter 3, CORDIC, too, since this will help with Project 4.
- ❑ New ebook is on Canvas at: Files / Books\_Readings / Kastner\_HLS\_Book
- ❑ Chapter 1, Introduction, is also a good overview of goals for HLS.

# Vitis HLS treeadd files and Export RTL

---

- ❑ Goals:
  - ❑ View simple C function and testbench
  - ❑ Build Vitis HLS project to include files and Zedboard
  - ❑ Run C simulation to verify syntax and function
  - ❑ Run debugger to view and single step through code
  - ❑ Synthesize the design and view time steps and execution
  - ❑ Export RTL for “Vivado IP for System Generator” (Still old name now for Model Composer).
- 
- ❑ Note: Export RTL default for just plain “Vivado IP” is for use with Vivado and Vitis for ARM core C program control and not for Model Composer

# Include Vitis HLS Block in Model Composer

- ❑ Default location for “solution1” export will be a subfolder under the project name folder where you built the Vitis HLS project.
- ❑ After exporting RTL from Vitis HLS, start Model Composer
- ❑ You will need to add a Vitis HLS block and point to your solution folder
- ❑ In the example, a Model Composer model has been created using the Vitis HLS block already.
- ❑ When opening that Model Composer model, there are “extra” signals on the Vitis HLS block
- ❑ These signals enable the block to run in simulation
- ❑ When we export as IP catalog, then these will be signals that the ARM core will use in the AXI bus interface

# Files on Canvas – Vitis HLS to Model Composer

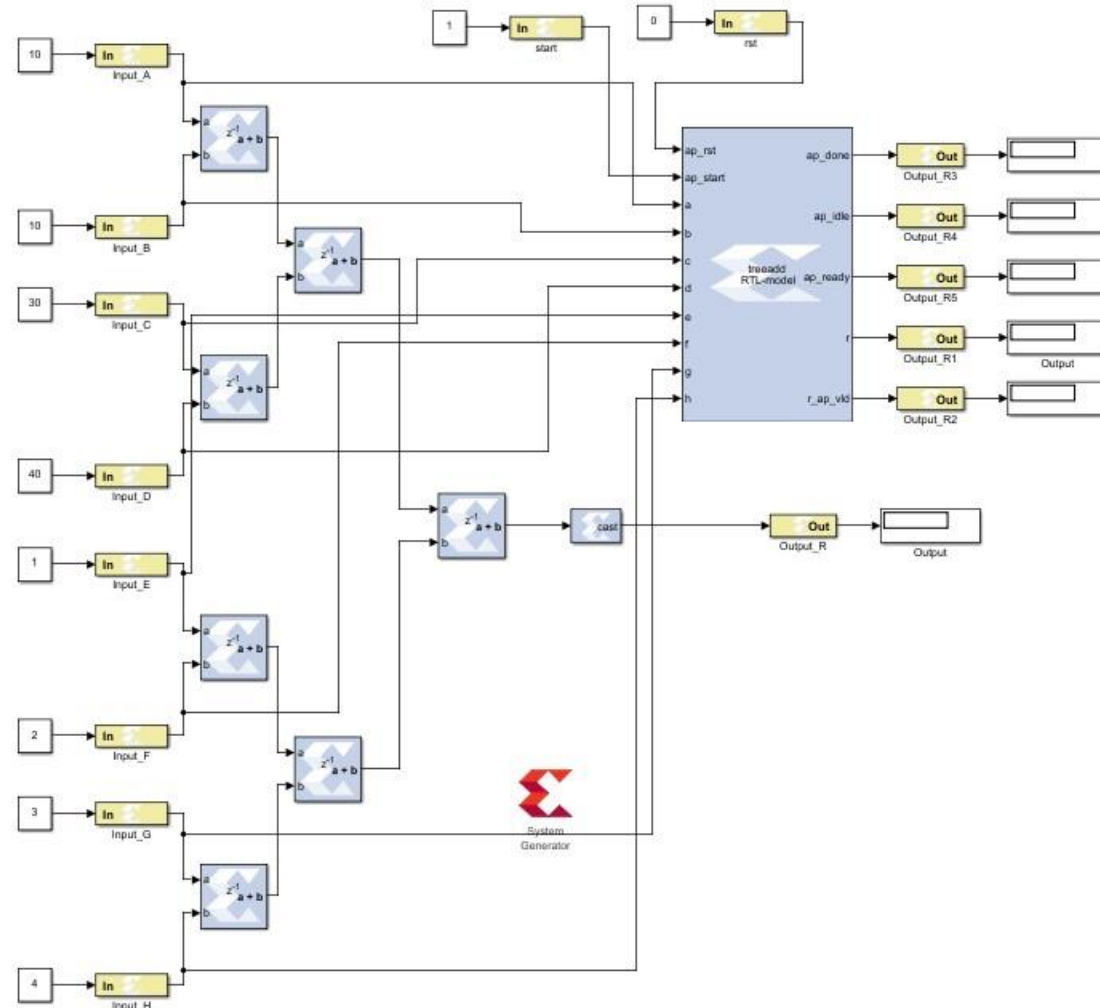
---

- ❑ Located in:
- ❑ ELEC 522/Files/CAD\_Tool\_Examples/Treeadd\_Vitis\_HLS\_MC
  
- ❑ treeadd.cpp - function to synthesize
- ❑ treeadd.h - header file
- ❑ treeadd\_test.cpp – testbench file
- ❑ treeadd\_HLS\_MC.slx – Premade Model Composer File
  
- ❑ This version has a similar treeadd.cpp file from last week without AXI Lite ports for Model Composer control. Model Composer uses the Gateways as I/O Pins through JTAG and not memory mapped under ARM processor core “C” program control.



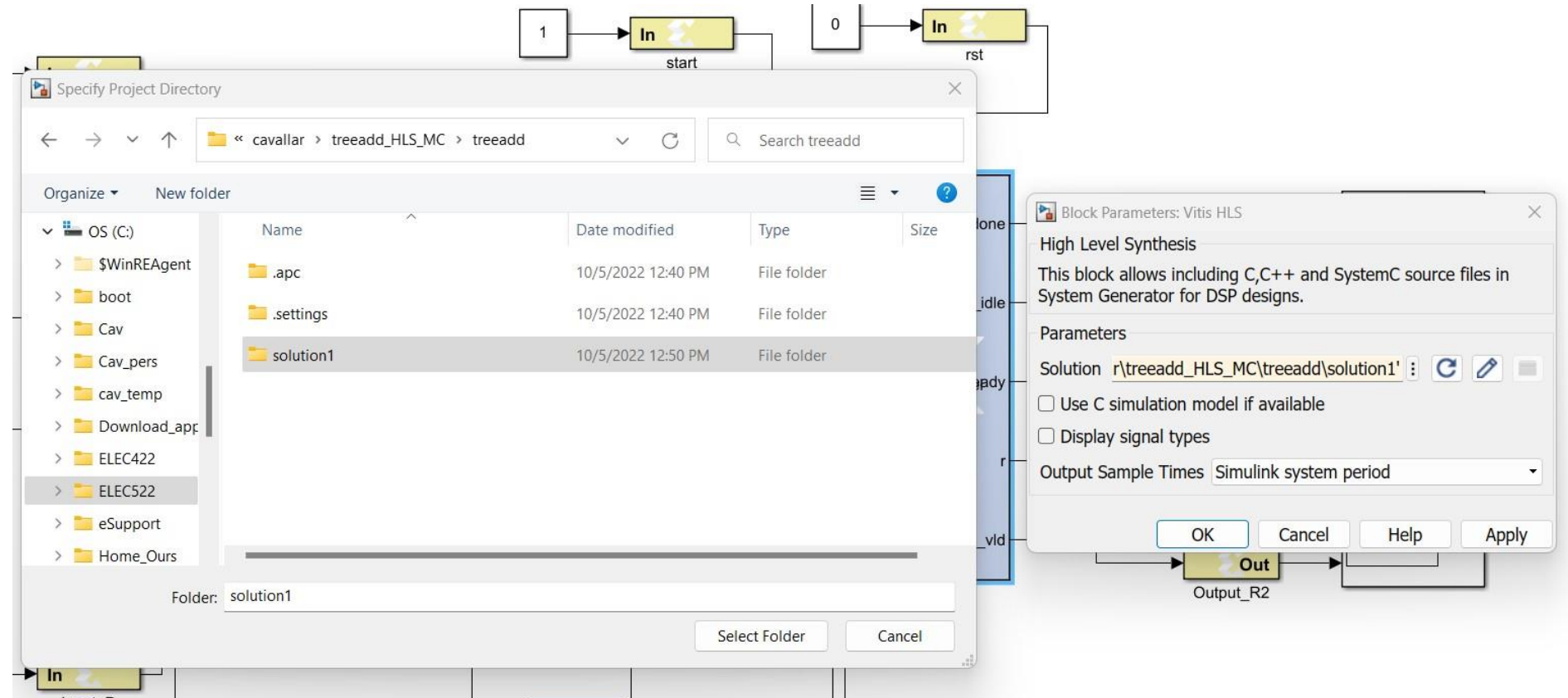
# Model Composer with Vitis HLS block and preset inputs

- Overall Model Composer file with constant inputs.



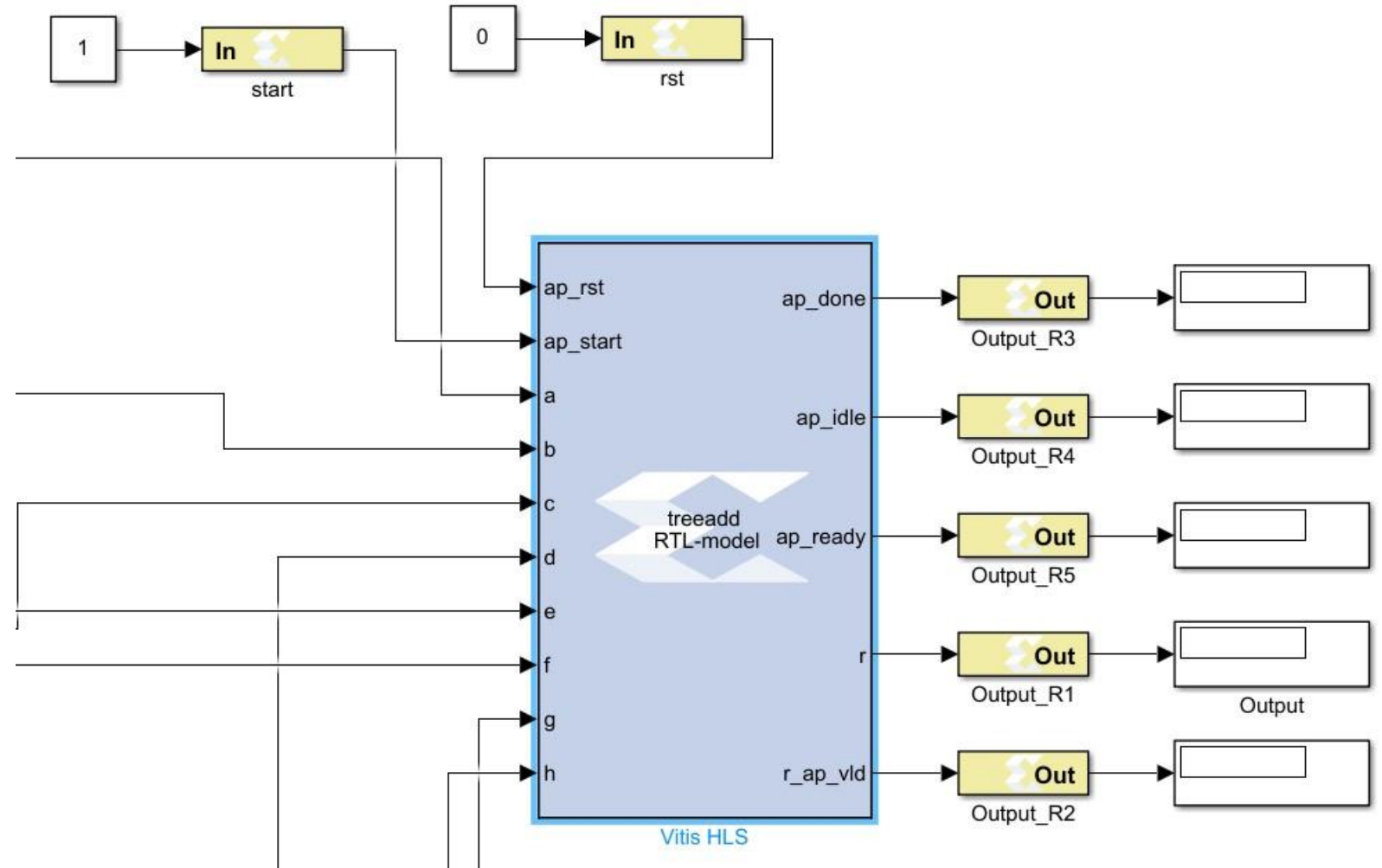
# Solution1 folder linked from your Folder

- ❑ Update Vitis HLS block to point to solution from your folder (instead of mine.)



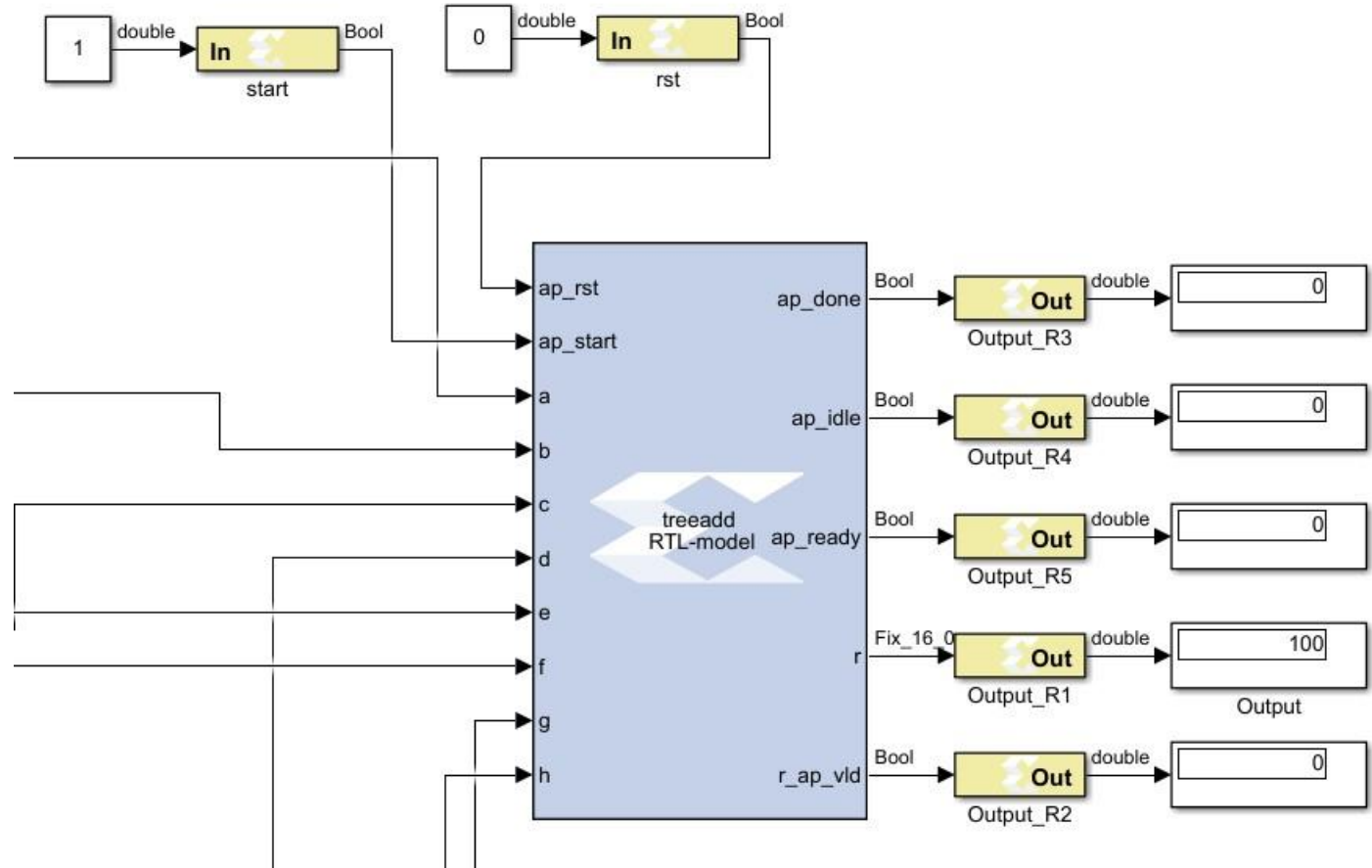
# Vitis HLS Model Composer Block Configured before Simulation

- Note how some control inputs have Boolean values set.
- Note too some extra control outputs terminated in gateways and displays.
- Some gateway labels added for clarity.



# Model Composer Simulation

- ❑ Correct output on the “r” result output based on fixed constant inputs.
- ❑ Note other done signals not used and producing zero values.





# Other Older HLS Tutorials on Canvas

- ❑ The treeadd demo that we did last week does an “Export RTL for Vivado” only and then Vitis C language ARM control instead of Model Composer export and Simulink HW Co-Sim
- ❑ (In same folder as the lab 1): Lab 2 focuses on:
  - ⇒ Add directives in your design
  - ⇒ Understand the effect of INLINE directive
  - ⇒ Improve performance using PIPELINE directive
  - ⇒ Distinguish between DATAFLOW directive and Configuration Command functionality
- ❑ Lab 3: Apply memory partitions techniques to improve resource utilization
- ❑ Lab 4: Create a processor system using IP Integrator in Vivado (Lab 4 Not really needed for Project 3 and for older tool flow.)
- ❑ These versions have not been fully tested with the 2022.1 tools but should be general examples.

# Additional Examples on Vectoradd

- ❑ For vector inputs in Vitis HLS several examples will be posted.
- ❑ When included in Model Composer, Matlab timeseries for data can be used.
- ❑ Different PRAGMAs can be used that require different data organization.
- ❑ Will be posted on Canvas soon after testing with the 2022.1 tools. An announcement will be made when ready.

 vectoradd\_HLS\_fifo\_MC

 vectoradd\_HLS\_MC

 vectoradd\_HLS\_nopipe\_MC

 vectoradd\_HLS\_reshape\_MC

# Next Lecture

---

- ❑ Next Tuesday October 11 is Fall Break.
- ❑ Next Thursday October 12 will be a Project 3 help session.
- ❑ Discuss “ap\_fixed” data types for arbitrary precision fixed point (integer and fraction) data