Developing the Accelerator Using HLS

Objectives

After completing this module, you will be able to:

- Describe the high-level synthesis flow
- Describe the capabilities of the Vivado HLS tool
- List the types of I/O abstracted in the Vivado HLS tool
- Create bus interfaces

Introduction

- **Introduction**
- Vivado HLS Tool Flow
- Interface Synthesis
- Creating Bus Interfaces
- Summary

Need for High-Level Synthesis

- **Algorithmic-based approaches are popular due to accelerated design time and time-to-market pressures**
  - Larger designs pose challenges in design and verification of hardware
- **Industry trend is moving towards hardware acceleration to enhance performance and productivity**
  - CPU-intensive tasks are now offloaded to hardware accelerator
  - Hardware accelerators require a lot of time to understand and design
- **Vivado HLS tool converts algorithmic description written in C-based design flow into hardware description (RTL)**
  - Elevates the abstraction level from RTL to algorithms
- **High-level synthesis is essential for maintaining design productivity for large designs**

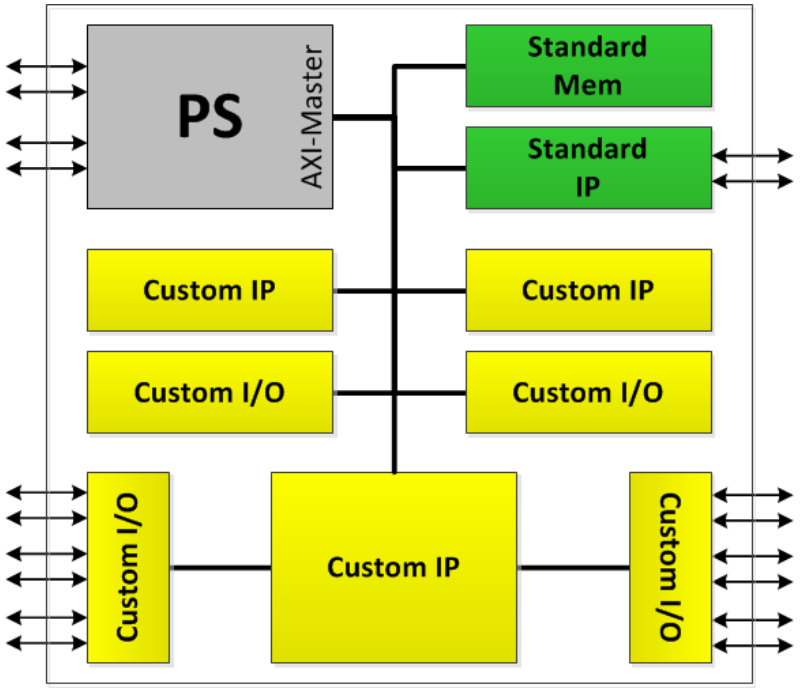
Introduction to High-Level Synthesis

- **Hardware extraction from C code**
  - Control and datapath can be extracted from C code at the top level
  - Same principles used in the example can be applied to sub-functions
    - At some point in the top-level control flow, control is passed to a sub-function
    - Sub-function can be implemented to execute concurrently with the top level and or other sub-functions

- **Scheduling and binding processes create hardware design from control flow graph considering the constraints and directives**
  - Scheduling process maps the operations into cycles
  - Binding process determines which hardware resource, or core, is used for each operation
  - Binding decisions are considered during scheduling because the decisions in the binding process can influence the scheduling of operations
    - For example, using a pipelined multiplier instead of a standard combinational multiplier

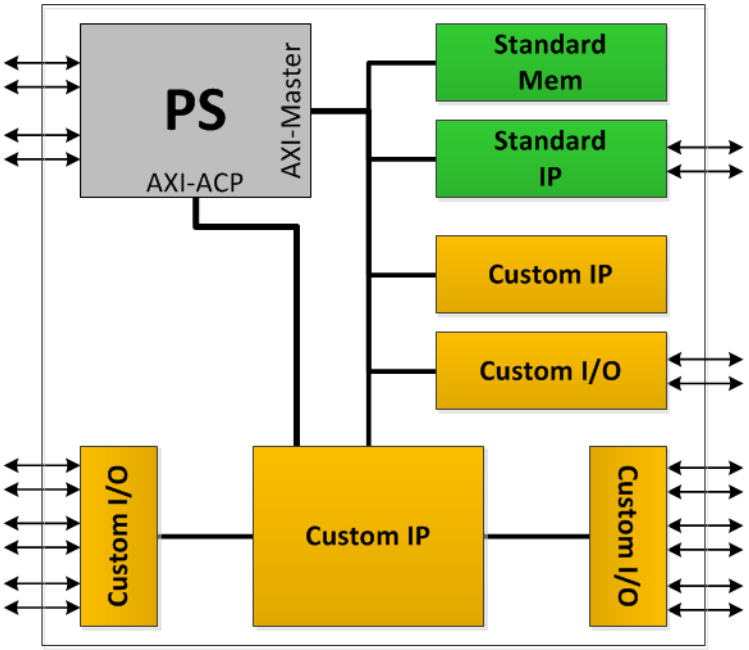
Data Flow Model

- **Custom IP for complex function and data flows**
- **PS used for control and resource management**
  - Not data processing



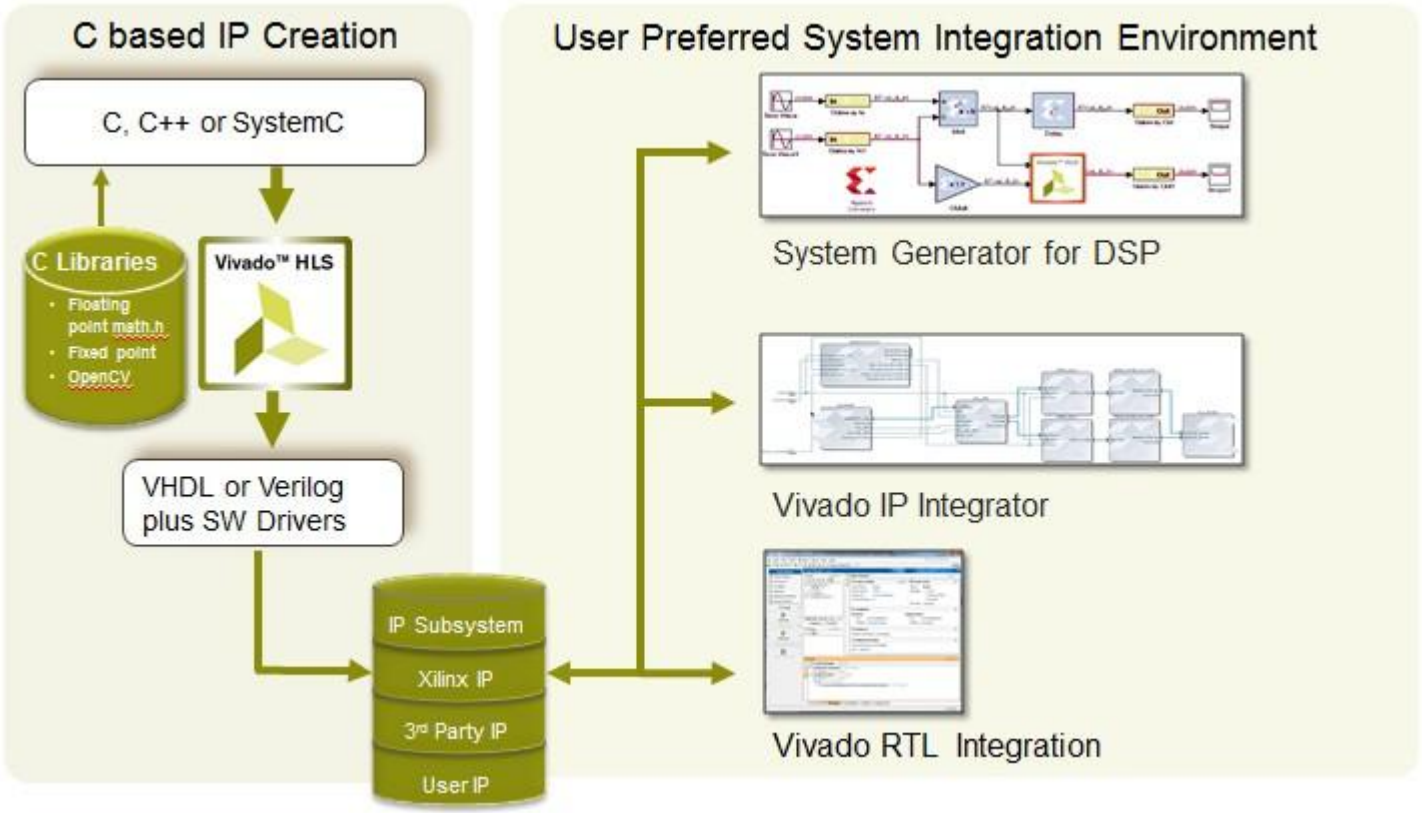
Acceleration Model

- Joint PS-PL use focus
- Balances software/hardware partition
- PS primary compute platform
- PL for hardware acceleration



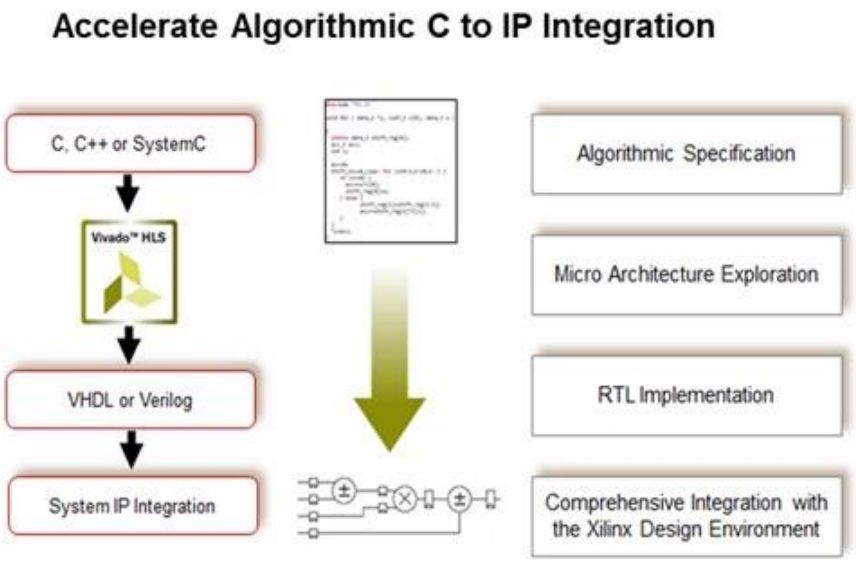
- Communications between
  - GP ports used for accelerator management
  - Data moved on high-efficiency ports (ACP/HPx)
  - Interrupts used to signal significant events

System Integration



High-Level Synthesis: HLS

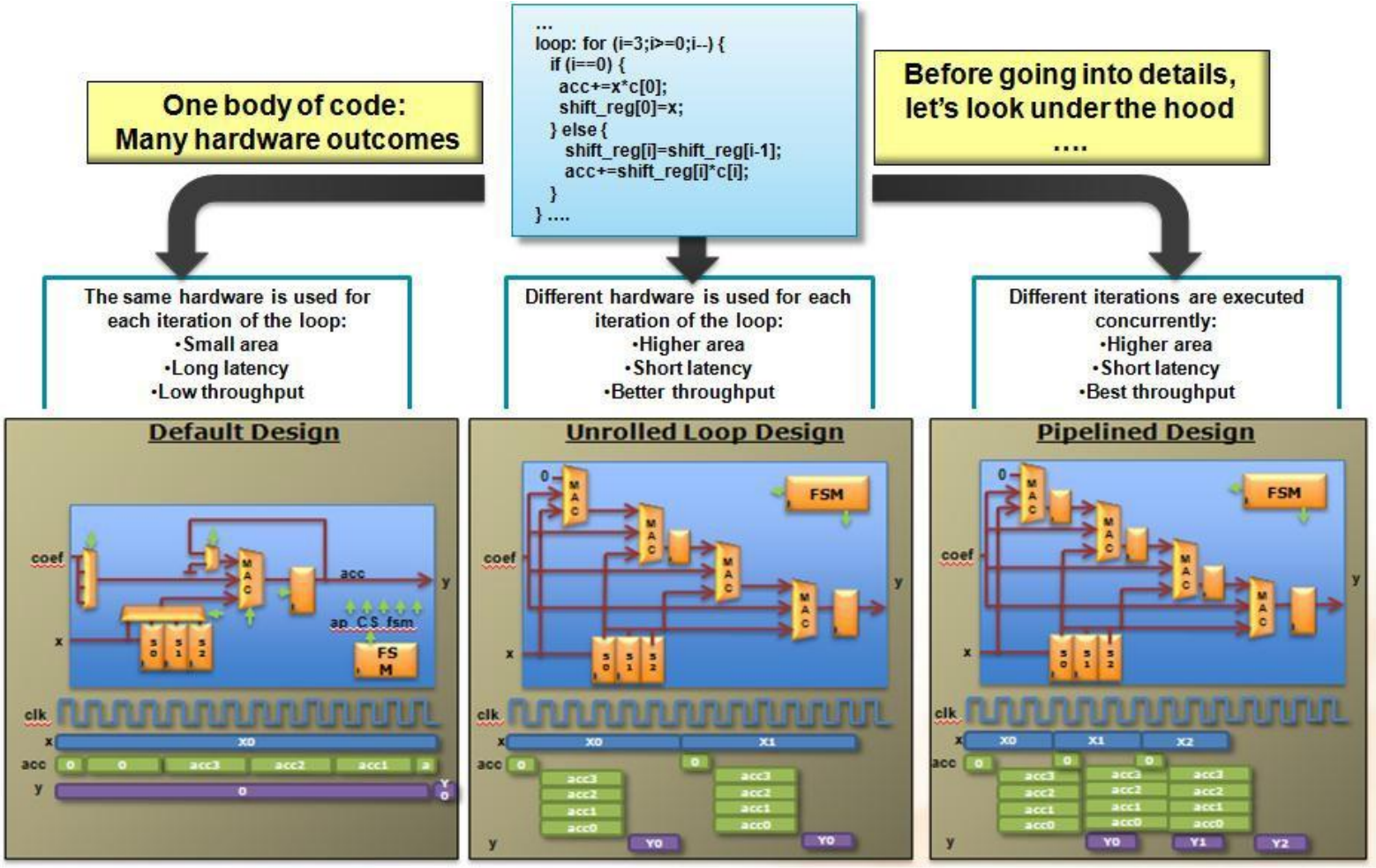
- **High-level synthesis**
  - Creates an RTL implementation from source code
    - C, C++, SystemC
    - Coding style impacts hardware realization
    - Limitations on certain constructs and access to libraries
  - Extracts control and dataflow from the source code
  - Implements the design based on defaults and user-applied directives



- **Many implementations are possible from the same source description**
  - Smaller designs, faster designs, optimal designs

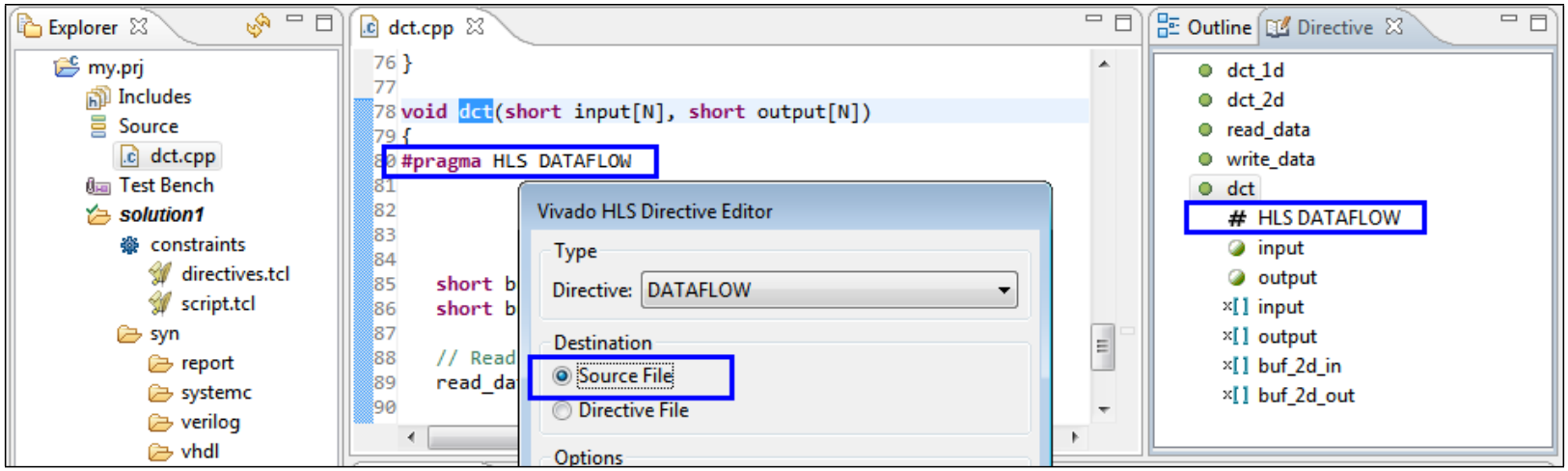
- **Enables design exploration**

Design Exploration with Directives



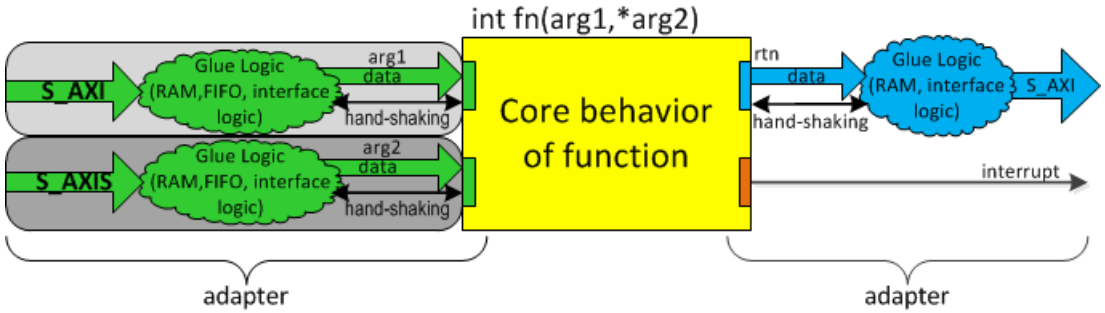
Optimization Directives: Pragma

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



Designer Challenges

- **Body of function remains generally unchanged**
  - May need to mark un-synthesizable code with PRAGMA
- **Main task of designer falls to how to move data in/out of core**
  - C function argument become physical ports
  - Physical ports become AXI and sideband signals
  - PRAGMAs function as design constraints



Library Support

- **Library support**
  - Floating-point support
    - Support for single-precision and double-precision, floating-point functions from math.h
  - Fixed-point support
    - Simulate and implement fixed-point algorithms using the <ap_int.h> library
  - OpenCV video function support
    - Enable migration of OpenCV designs into Zynq® All Programmable SoC
    - Libraries target real-time full HD video processing
  - DSP function support
    - Instantiate and parameterize FIR compiler and FFT LogiCORE™ IP as function calls from your C++ code

Unsupported Constructs: Overview

- **System calls and function pointers**
  - Dynamic memory allocation
    - malloc() and free()
  - Standard I/0 and file I/O operations
    - fprintf() / fscanf(), etc.
  - System calls
    - time(), sleep(), etc.
- **Data types**
  - Forward declared type
  - Recursive type definitions
    - Type contains members with the same type
- **Non-standard pointers**
  - Pointer casting between general data types
    - OK with native integers types
  - If a double pointer is used in multiple functions, Vivado HLS tool will inline all the functions
    - Slower synthesis, may increase area and run time

Synthesis Macro

- **Macro __SYNTHESIS__ is auto defined**
  - Defined when Vivado HLS tool elaborates designs
    - Not defined inside Vivado HLS tool when C compilation is performed
  - This can be used to remove unsynthesizable code from the design

```
// test.c
#include <stdio.h>

void test (int d[10]) {
  int acc = 0;
  int i;
#ifndef __SYNTHESIS__
FILE *fp1;
#endif


  for (i=0;i<10;i++){
    acc += d[i];
    d[i] = acc;
#ifndef __SYNTHESIS__
    fprintf(fp1, "%d \n", acc);
#endif

  }
}
```

File access to the OS is ignored during synthesis

Code is ***only*** executed if __SYNTHESIS__ is ***not defined***

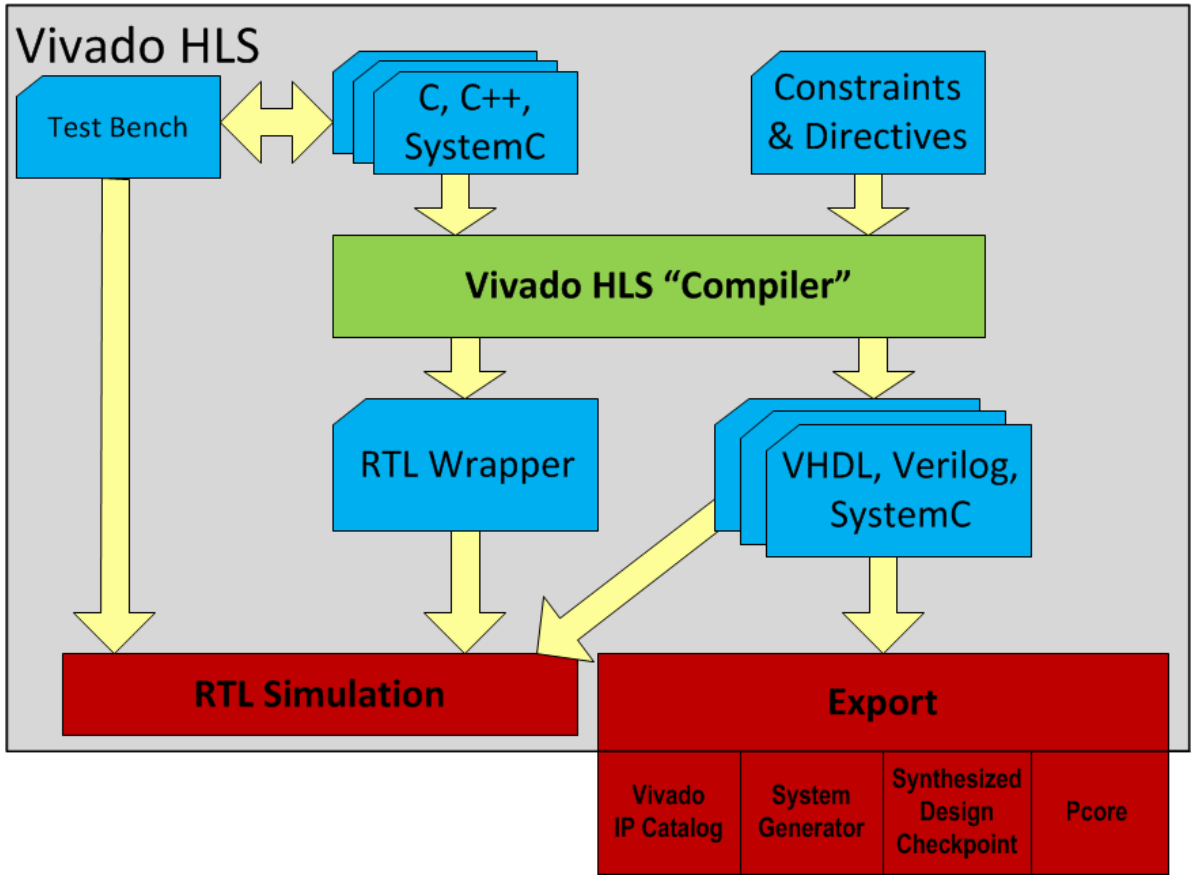
Vivado HLS Tool Flow

- Introduction
- **Vivado HLS Tool Flow**
- Interface Synthesis
- Creating Bus Interfaces
- Summary

Vivado HLS Tool is a Complete Environment

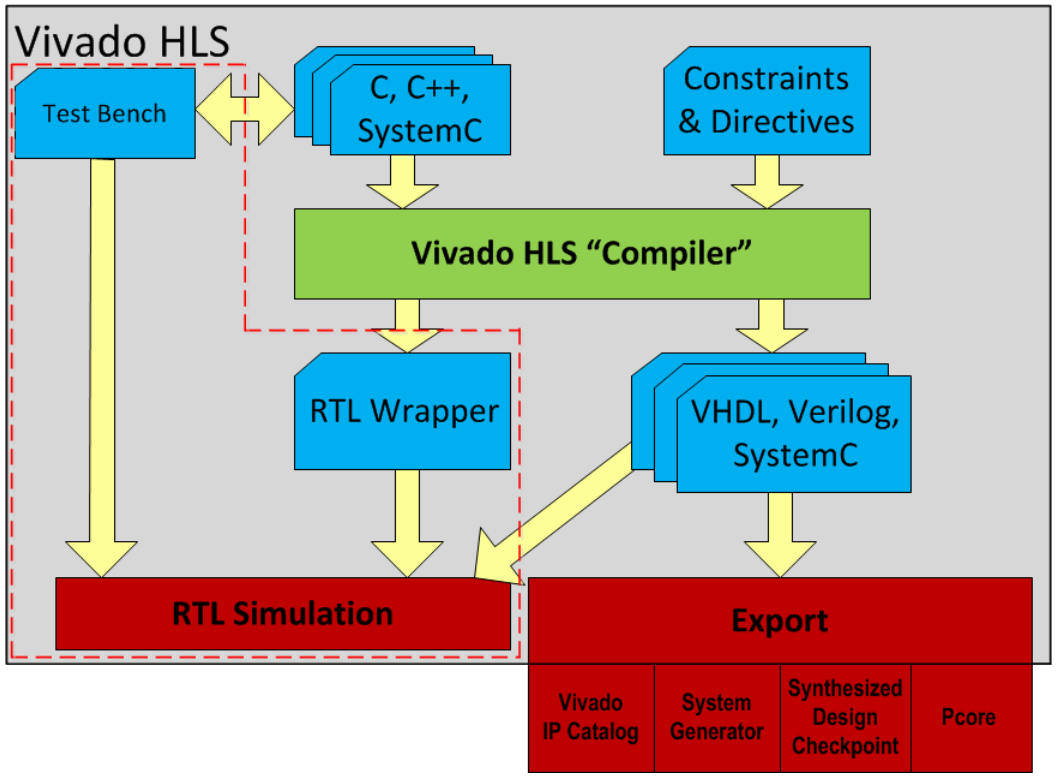
- **Simulation**
  - C/C++/SystemC support
  - VHDL/Verilog support through Vivado simulator
- **Synthesis**
  - Generates VHDL or Verilog output
- **Design analysis and exploration**
  - Visualization tools
- **IP export**
  - IP-XACT, PCore, System Generator, and SysGen format

Vivado HLS Tool: High-Level Synthesis



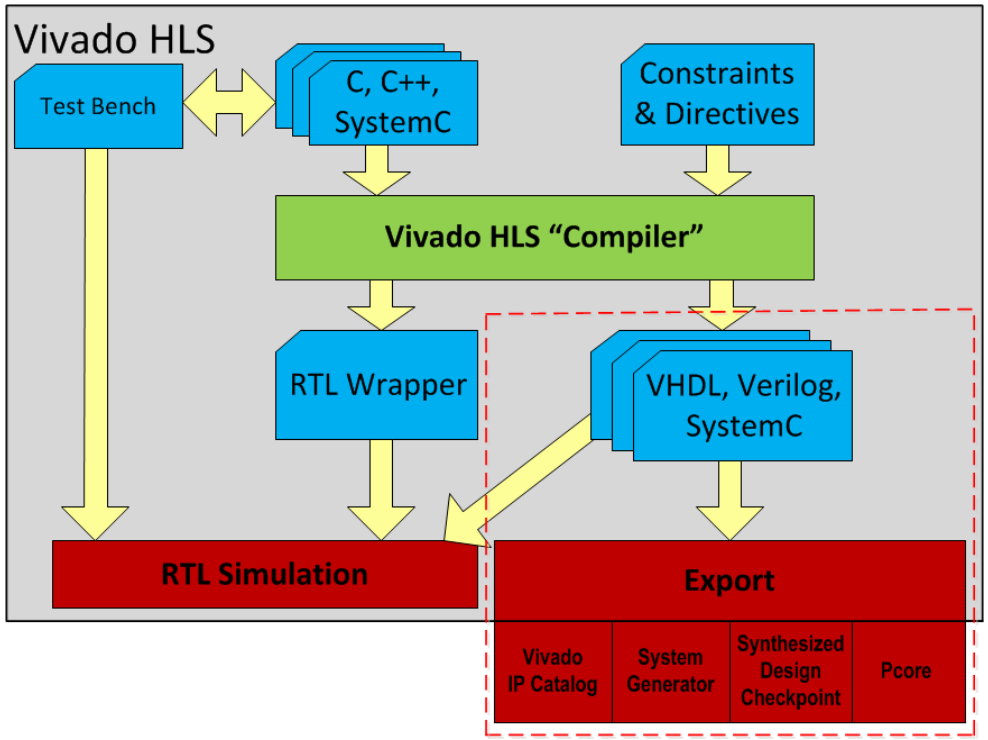
Vivado HLS Tool: RTL Verification

- **RTL output**
  - Verilog
  - VHDL
  - SystemC
- **Automatic re-use of C-level testbench**
- **RTL verification executed within HLS**
- **Support for third-party HDL simulators**



Vivado HLS Tool: Exporting RTL as IP

- **RTL output in**
  - Verilog
  - VHDL
  - SystemC
- **RTL design exported as IP**
  - Used in Xilinx and other tools
- **RTL synthesis can be executed within HLS**
  - For exploring results



Summary of Synthesis Flows (1)

Flow	Purpose	Output
Run C Simulation	Validate that the C algorithm has the correct functionality	Testbench should be self checking
Run C Synthesis	Synthesize C to RTL	RTL output files
Multiple Solutions	Exploration of best area/performance trade-offs	RTL output files, user reports
C/RTL Co-simulation	Verify the RTL gives the same results as the C code using the same testbench (stimuli)	Testbench should be self checking
Export RTL	Export the RTL as IP/System Generator block/pcore to use with other Xilinx tools	IP package (optionally the results of RTL synthesis for your review)

Summary of Synthesis Flows (2)

- **Vivado HLS tool will create results in an RTL synthesis directory**
  - These results confirm that RTL synthesis matches those estimated by Vivado HLS tool
- **Export RTL output in the *impl* directory should be used for final RTL synthesis**
  - Take this RTL IP and combine it with the other blocks in RTL project/IPI project/System Generator
  - Then, everything is synthesized, placed, routed, and bitstream generated

Interface Synthesis

- Introduction
- Vivado HLS Tool Flow
- **Interface Synthesis**
- Creating Bus Interfaces
- Summary

Key Attributes of C Code

- Functions: All code is made up of functions that represent the design hierarchy; the same in hardware
- Top-level I/O: The arguments of the top-level function determine the hardware RTL interface ports
- Types: All variables are of a defined type. The type can influence the area and performance
- Loops: Functions typically contain loops. How these are handled can have a major impact on area and performance
- Arrays: Arrays are used often in C code. They can impact the device I/O and become performance bottlenecks
- Operators: Operators in the C code may require sharing to control area or specific hardware implementations to meet performance

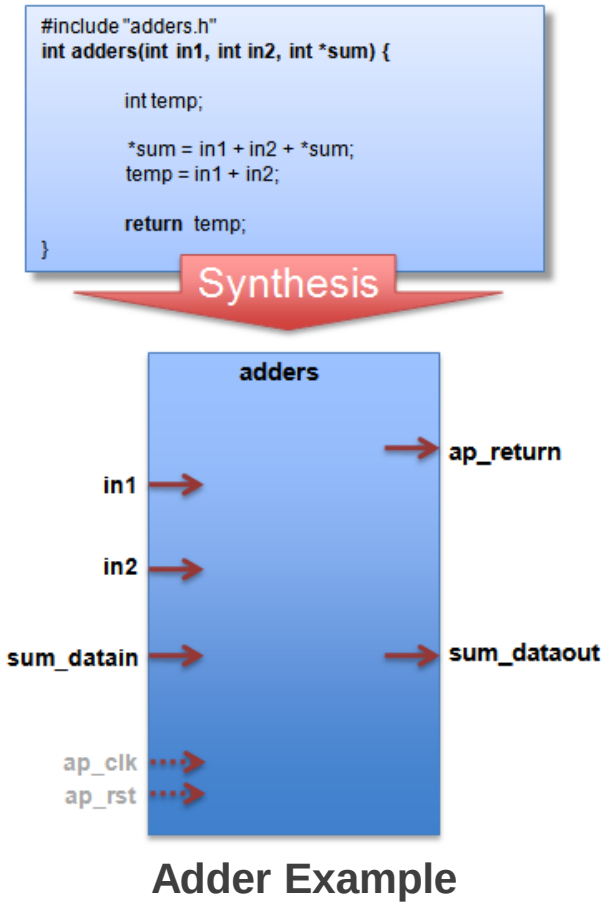
```
void fir (
  data_t *y,
  coef_t c[4],
  data_t x
  ) {

  static data_t shift_reg[4];
  acc_t acc;
  int i;

  acc=0;
  loop: for (i=3;i>=0;i--) {
    if (i==0) {
      acc+=x*c[0];
      shift_reg[0]=x;
    } else {
      shift_reg[i]=shift_reg[i-1];
      acc+=shift_reg[i] * c[i];
}
  }
  *y=acc;
}
```

Vivado HLS Tool I/O Options

- **Data ports**
  - Directly derived from the function arguments/parameters
- **Block-level interfaces (optional)**
  - An interface protocol that is added at the block level
  - Controls the addition of block level control ports: start, idle, done, and ready
- **Port-level interfaces (optional)**
  - I/O interface protocols added to the individual function arguments
- **Bus interfaces (optional)**
  - Added as external adapters when the RTL is exported as an IP block

Vivado HLS Tool I/O Options: Function Arguments

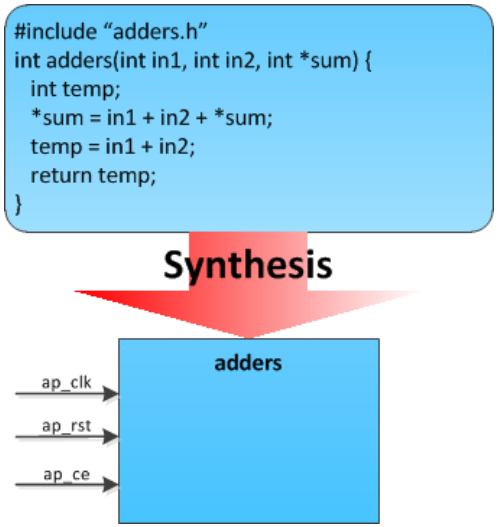
- **Function arguments**
  - Synthesized into data ports
- **Function return**
  - Any return is synthesized into an output port called ap_return
- **Pointers (and C++ references)**
  - Can be read from and written to
  - Separate input and output ports for pointer reads and writes
- **Arrays (not shown here)**
  - Like pointers can be synthesized into read and/or write ports



Adder Example

Vivado HLS Tool: Basic Ports

- **Clock added to all RTL blocks**
  - One clock per function/block
  - SystemC designs may have a unique clock for each CTHREAD
- **Reset added to all RTL blocks**
  - Only the FSM and any variables initialized in the C are reset by default
  - Reset and polarity options are controlled via the RTL configuration
    - Solutions/Solution Settings
- **Optional clock enable**
  - An optional clock enable can be added via the RTL configuration
  - When de-asserted it will cause the block to “freeze”
    - All connected blocks are assumed to be using the same CE
    - When the I/O protocol of this block freezes, it is expected other blocks will do the same
    - Else a valid output may be read multiple times

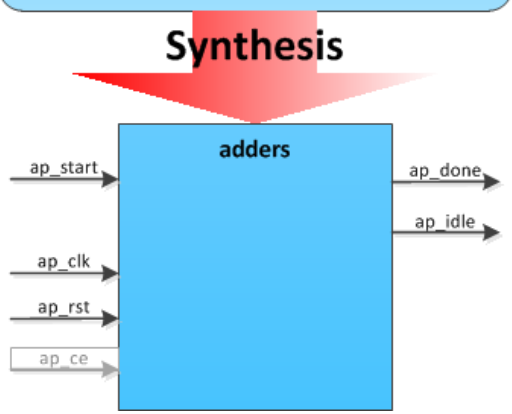


Adder Example

Vivado HLS Tool: Block-Level Signals

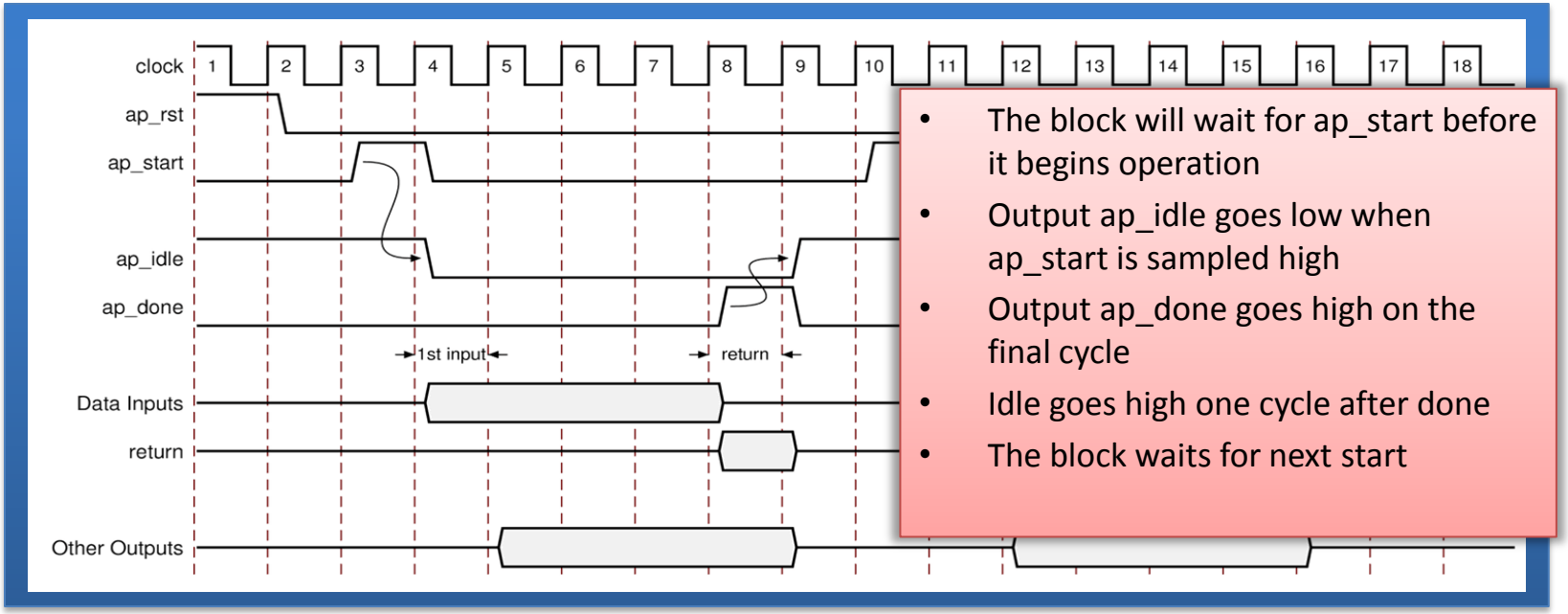
- **Block-level protocol**
  - An I/O protocol added at the RTL block level
  - Controls and indicates the operational status of the block
- **Block operation control**
  - Controls when the RTL block starts execution (ap_start)
  - Indicates if the RTL block is idle (ap_idle) or has completed (ap_done)
- **Complete and function return**
  - ap_done signal also indicates when any function return is valid
- **Ready (not shown here)**
  - If the function is pipelined an additional ready signal (ap_ready) is added
  - Indicates a new sample can be supplied before done is asserted

```
#include "adders.h"
int adders(int in1, int in2, int *sum) {
  int temp;
  *sum = in1 + in2 + *sum;
  temp = in1 + in2;
  return temp;
}
```



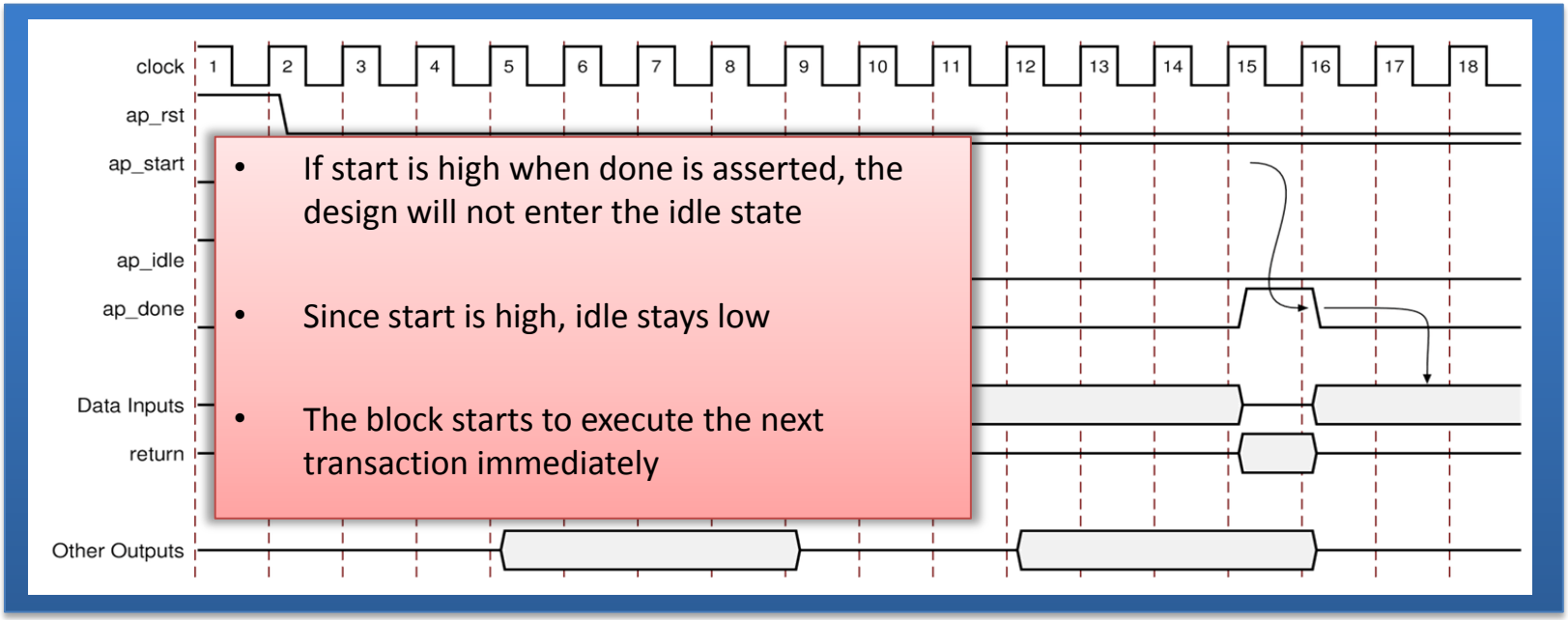
Adder Example

AP_START: Pulsed



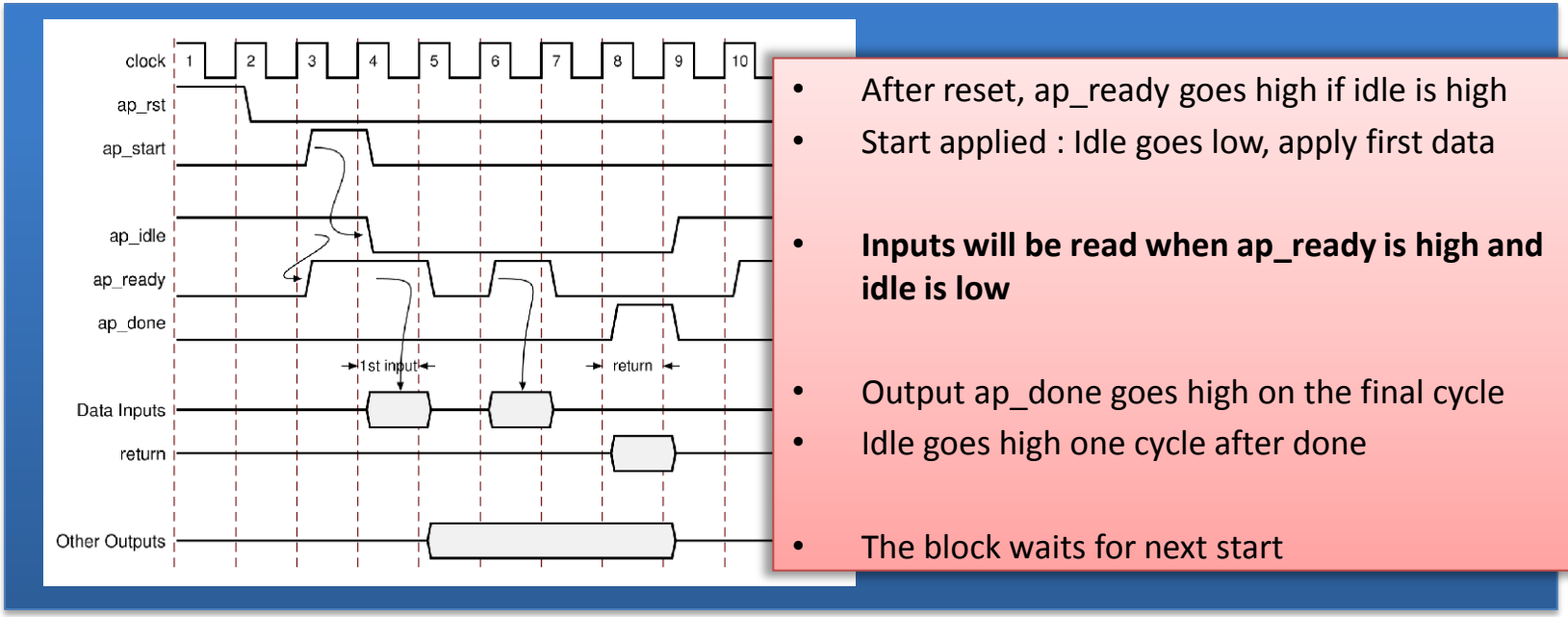
- **Input Data**
  - Can be applied when ap_idle goes low
    - The first read is performed 1 clock cycle after idle goes low
  - Input reads can occur in any cycle up until the last cycle
- **Output Data**
  - Any function return is valid when ap_done is asserted high
  - Other outputs may output their data at any time after the first read
    - The output can only be guaranteed to be valid with ap_done if it is registered
    - It is recommended to use a port level IO protocol for other outputs

AP_START: Constant High



- **Input and Output data operations**
  - As before
- **The key difference here is that the design is never idle**
  - The next data read is performed immediately

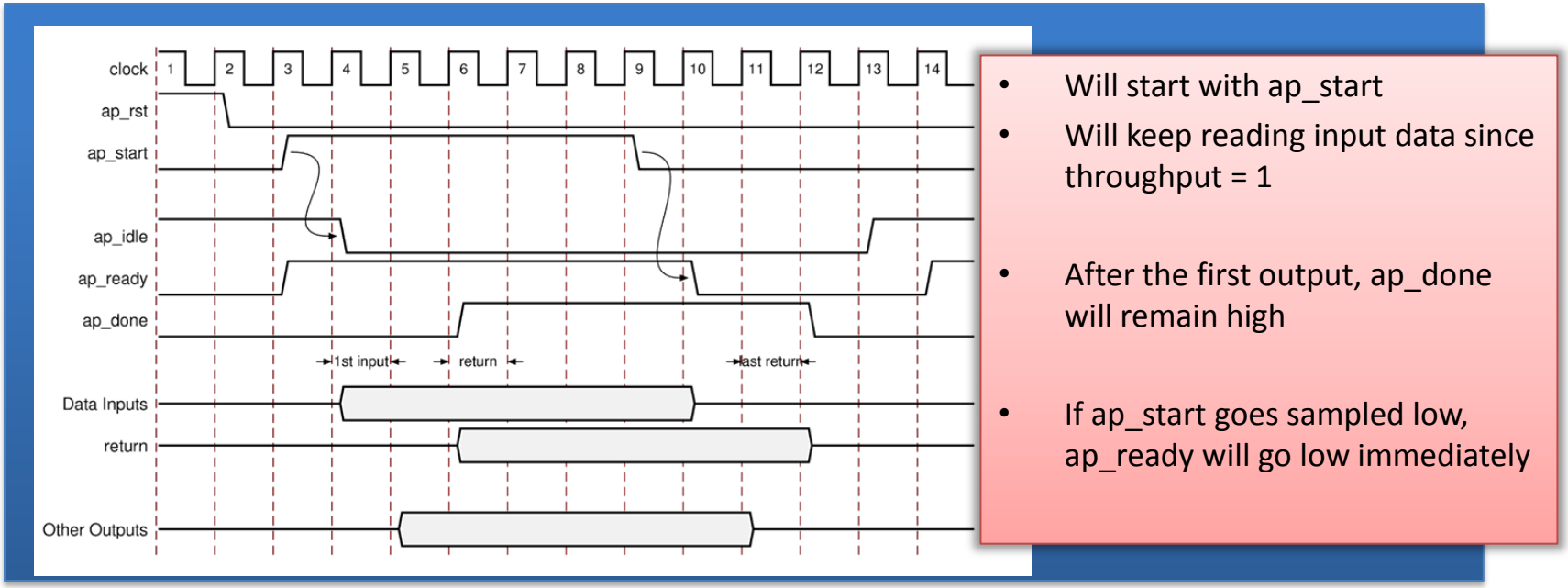
Pipelined Designs



- **Input Data**
  - Will be read when ap_ready is high and ap_idle is low
    - Indicates the design is ready for data
  - Signal ap_ready will change at the rate of the throughput
- **Output Data**
  - As before, function return is valid when ap_done is asserted high
  - Other outputs may output their data at any time after the first read
    - It is recommended to use a port level IO protocol for other outputs

This example shows an II of 2

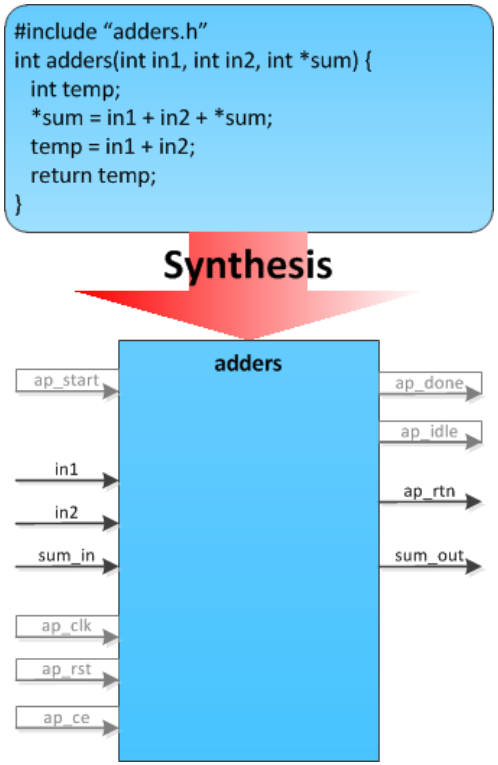
Pipelined Designs: Throughput = 1



- **Input Data when TP=1**
  - It can be expected that ap_ready remains high and data is continuously read
  - The design will only stop processing when ap_start is de-asserted
- **Output Data when TP=1**
  - After the first output, ap_done will remain high while there are samples to process
    - Assuming there is no data decimation (output rate = input rate)

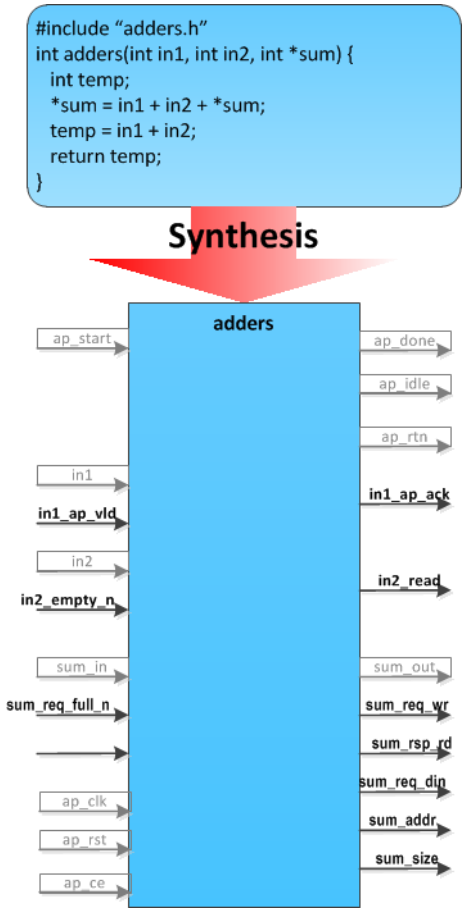
Vivado HLS Tool: Dealing with Arguments

- **Arguments become ports**
  - Pass-by-value becomes input only
  - Pass-by-reference becomes an input and an output
- **Return value becomes an output**



Vivado HLS Tool: Port I/O Protocols

- **Port I/O protocols**
  - I/O protocol added at the port level
  - Sequences the data to/from the data port
- **Interface synthesis**
  - Design is automatically synthesized to account for I/O signals (enables, acknowledges, etc.)
- **Select from a pre-defined list**
  - I/O protocol for each port can be selected from a list
  - Allows for easy connection to surrounding blocks
- **Non-standard interfaces**
  - Supported in C/C++ using an arbitrary protocol definition
  - Supported natively in SystemC



Adder Example

Let's Look at an Example

```
#include "adders.h"
int adders(int in1, int in2, int *sum) {

                    int temp;

                     *sum = in1 + in2 + *sum;
          temp = in1 + in2;

                    return  temp;
}
```

"Sum" is a pointer which is read and written to : an Inout

Argument Type	Variable			Pointer Variable			Array			Reference Variable		
	Pass-by- value			Pass-by-reference			Pass-by-reference			Pass-by-reference		
Interface Type	I	IO	O	I	IO	O	I	IO	O	I	IO	O
ap_none	D			D						D		
ap_stable												
ap_ack												
ap_vld						D						D
ap_ovld					D						D	
ap_hs												
ap_memory							D	D	D			
ap_fifo												
ap_bus												
ap_ctrl_none												
ap_ctrl_hs			D									

The port for "Sum" can be any of these interface types

The default for port for "Sum" will be type ap_ovld

Now, let's look at what these interfaces are …

Key:

I	: input
IO	: inout
O	: output
D	: Default Interface

Supported Interface

Unsupported Interface

Interface Types

Multiple interface protocols are available
Every combination of C argument and port protocol is not supported
It may require a code modification to implement a specific IO protocol

Argument Type	Variable			Pointer Variable			Array			Reference Variable		
	Pass-by- value			Pass-by-reference			Pass-by-reference			Pass-by-reference		
Interface Type	I	IO	O	I	IO	O	I	IO	O	I	IO	O
ap_none	D			D						D		
ap_stable												
ap_ack												
ap_vld						D						D
ap_ovld					D						D	
ap_hs												
ap_memory							D	D	D			
ap_fifo												
ap_bus												
ap_ctrl_none												
ap_ctrl_hs			D									

No IO Protocol

Wire handshake protocols

Memory protocols : RAM

Bus protocols

Block Level Protocol

Block level protocols can be applied to the return port - but the port can be omitted and just the function name specified

Key:

I : input
IO : inout
O : output
D : Default Interface

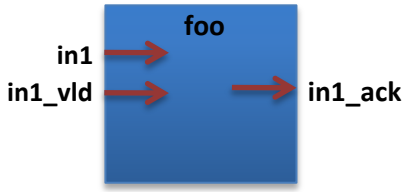
Supported Interface

Unsupported Interface

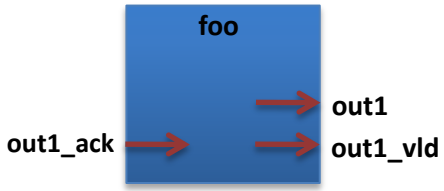
The wire protocols are all derivatives of protocol ap_hs

ap_hs is compatible with AXI-Stream

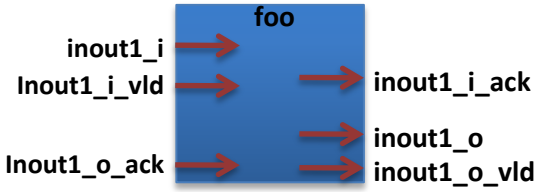
- Inputs
  - Arguments which are only read
  - The valid is input port indicating when to read
  - Acknowledge is an output indicating it was read



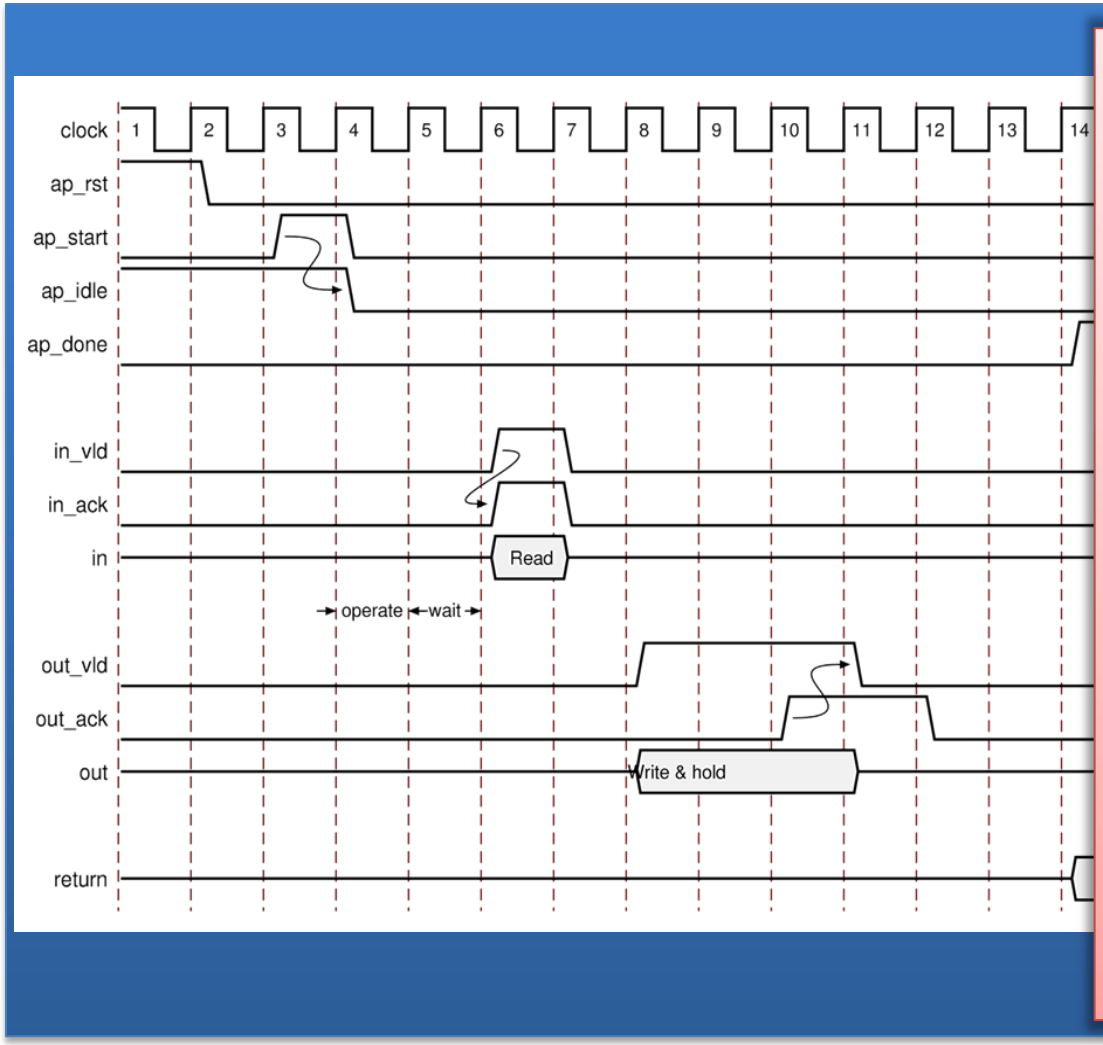
- Outputs
  - Arguments which are only written to
  - Valid is an output indicating data is ready
  - Acknowledge is an input indicating it was read



- Inouts
  - Arguments which are read from and written to
  - These are split into separate in and out ports
  - Each half has handshakes as per Input and Output



Handshake IO Protocol



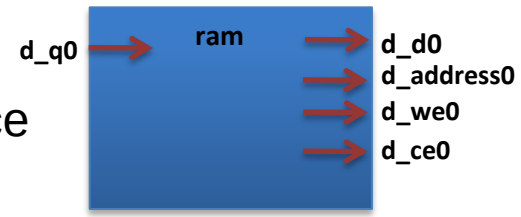
- *After start, idle goes low and the RTL operates <u>until a read is required</u>*
- The RTL will stall (wait) until the input valid is asserted
- If there is more than one input valid, each can stall the RTL
- It will acknowledge on the same cycle it reads the data (reads on next clock edge)
- An output valid is asserted when the port has data
- The RTL will stall (hold the data and wait) until an input acknowledge is received
- *Done will be asserted when the function is complete*

Memory IO Protocols: Ports Generated

RAM Ports

- Created by protocol ap_memory
- Given an array specified on the interface
- Ports are generated for data, address & control
  - Example shows a single port RAM
  - A dual-port resource will result in dual-port interface

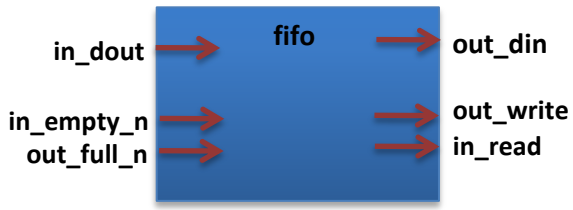
```
#include "ram.h"
void ram (int d[DEPTH], ...) {
  ...
}
```



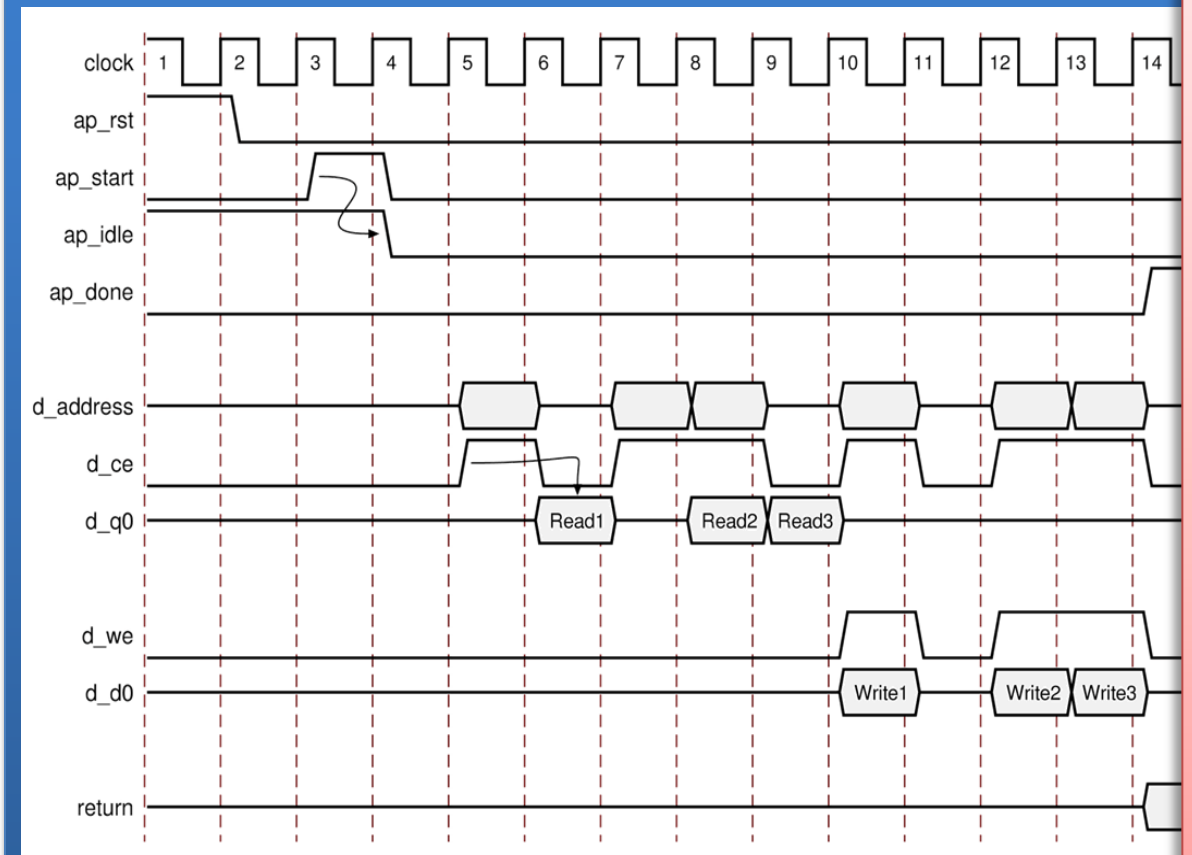
FIFO Ports

- Created by protocol ap_fifo
- Can be used on arrays, pointers and references
- Standard Read/Write, Full/Empty ports generated
  - Must use separate arrays for read and write
  - Pointers/References: split into In and Out ports

```
#include "fifo.h"
void fifo (int out[DEPTH],
           int in[DEPTH]) {

  ...
}
```

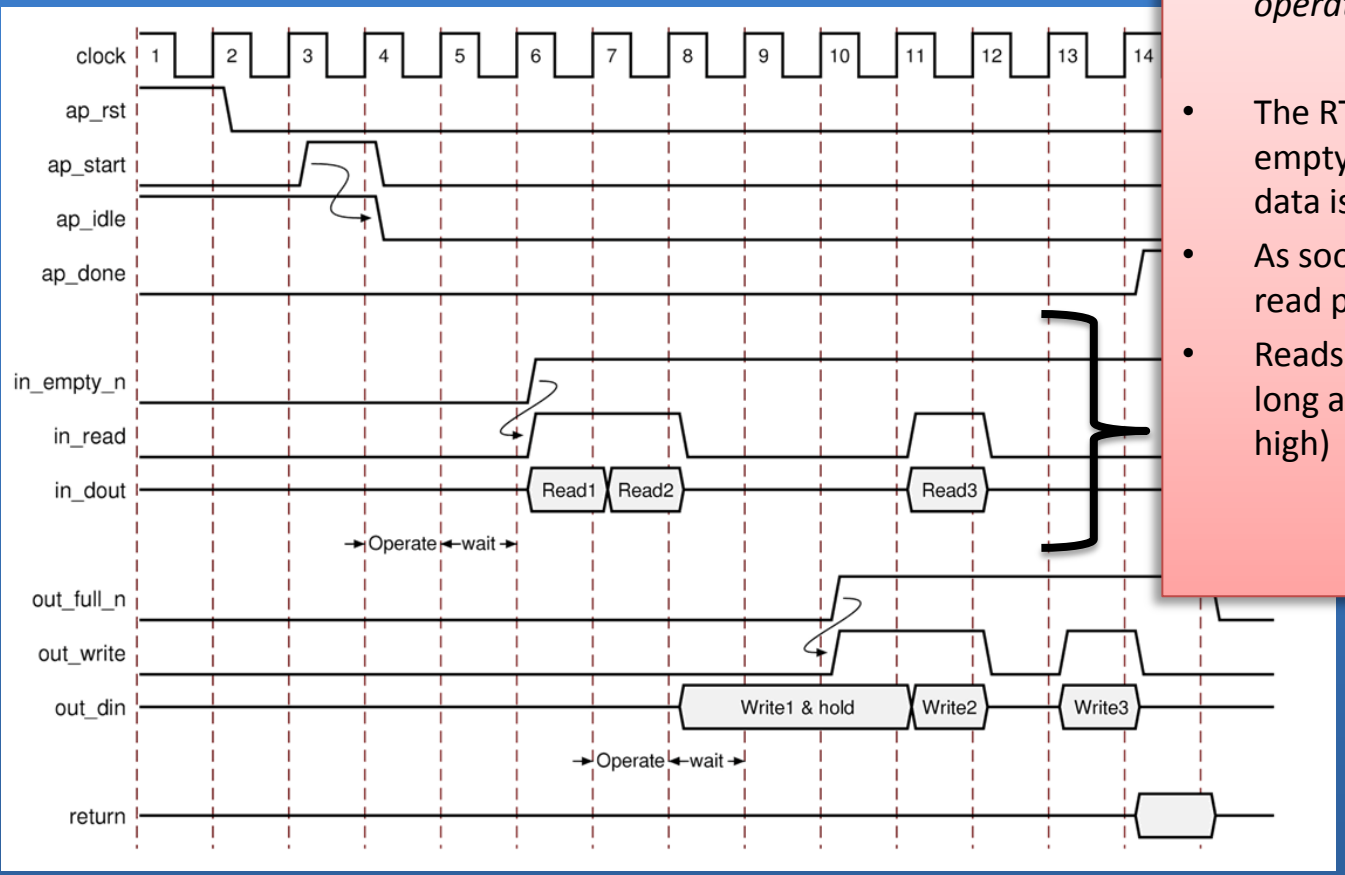


Memory IO Protocol



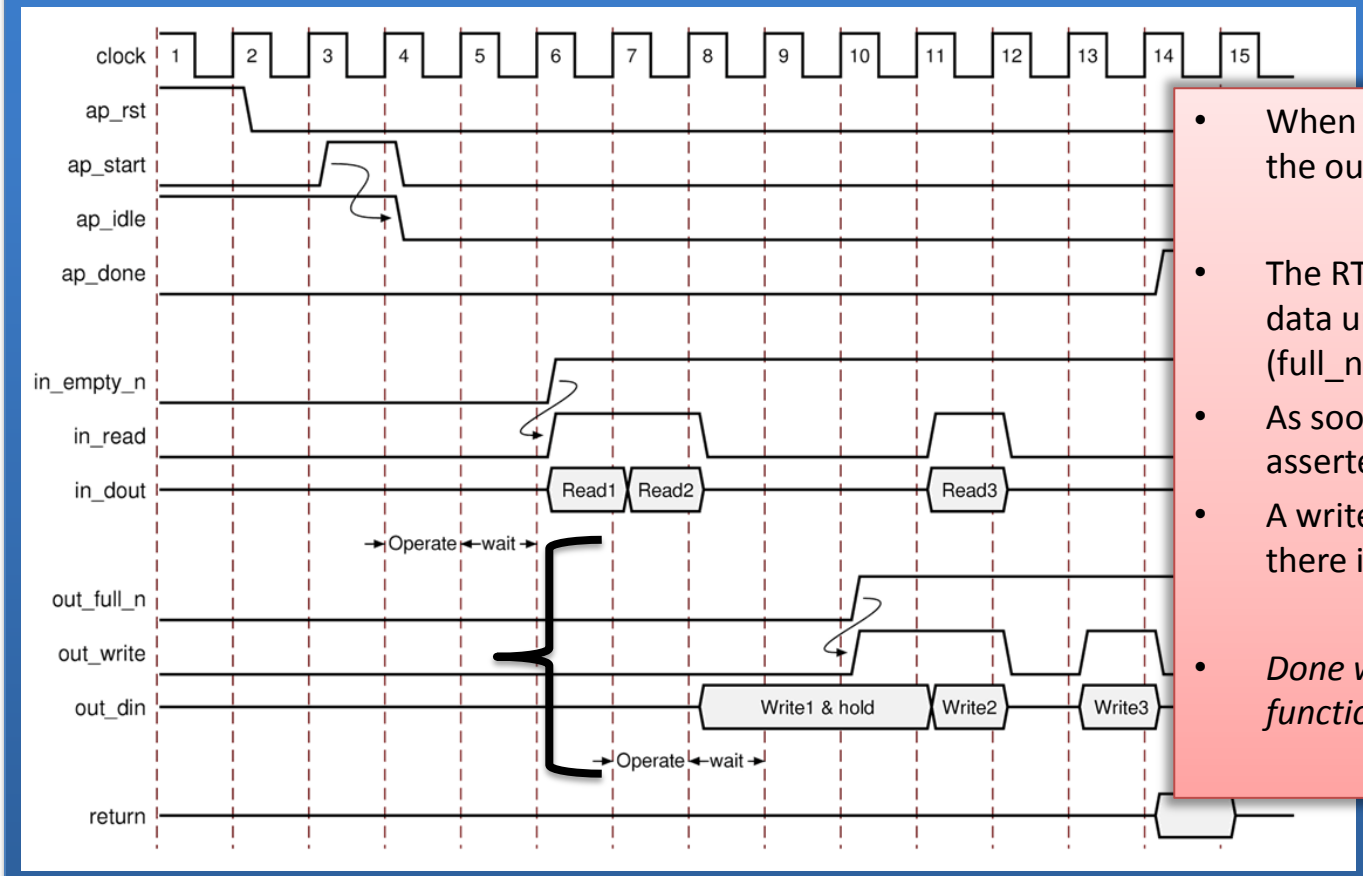
- *After start, idle goes low and the RTL operates*
- When a read is required, an address is generated and CE asserted high
- Data is available on data input port d_q0 in the next cycle
- The read operations may be pipelined
- When a write is required, the address and data are placed on the output ports
- Both CE &WE are asserted high
- Writes may be pipelined
- *Done will be asserted when the function is complete*

FIFO IO Protocol (Read)



- *After start, idle goes low and the RTL operates <u>until a read is required</u>*
- The RTL will stall (wait) until the empty_n is asserted high to indicate data is available
- As soon as data is available, the fifo read port will go high
- Reads will occur when required, so long as data is available (empty_n is high)

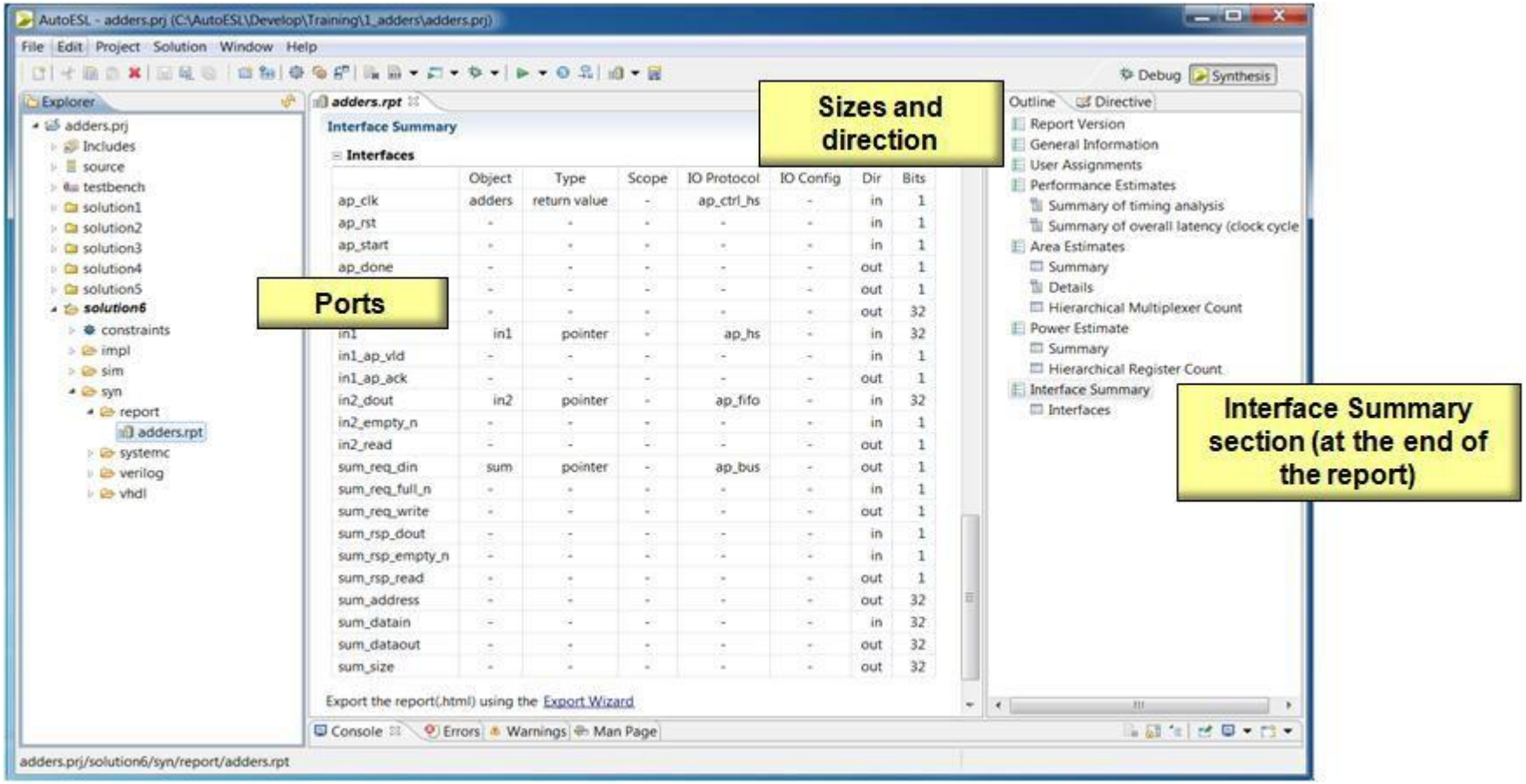
FIFO IO Protocol (Write)



- When ready, data will be written to the output port
- The RTL will then stall and hold the data until the FIFO is no longer full (full_n high)
- As soon as data can written, write is asserted high
- A write will occur when required if there is room in the fifo (full_n high)
- *Done will be asserted when the function is complete*

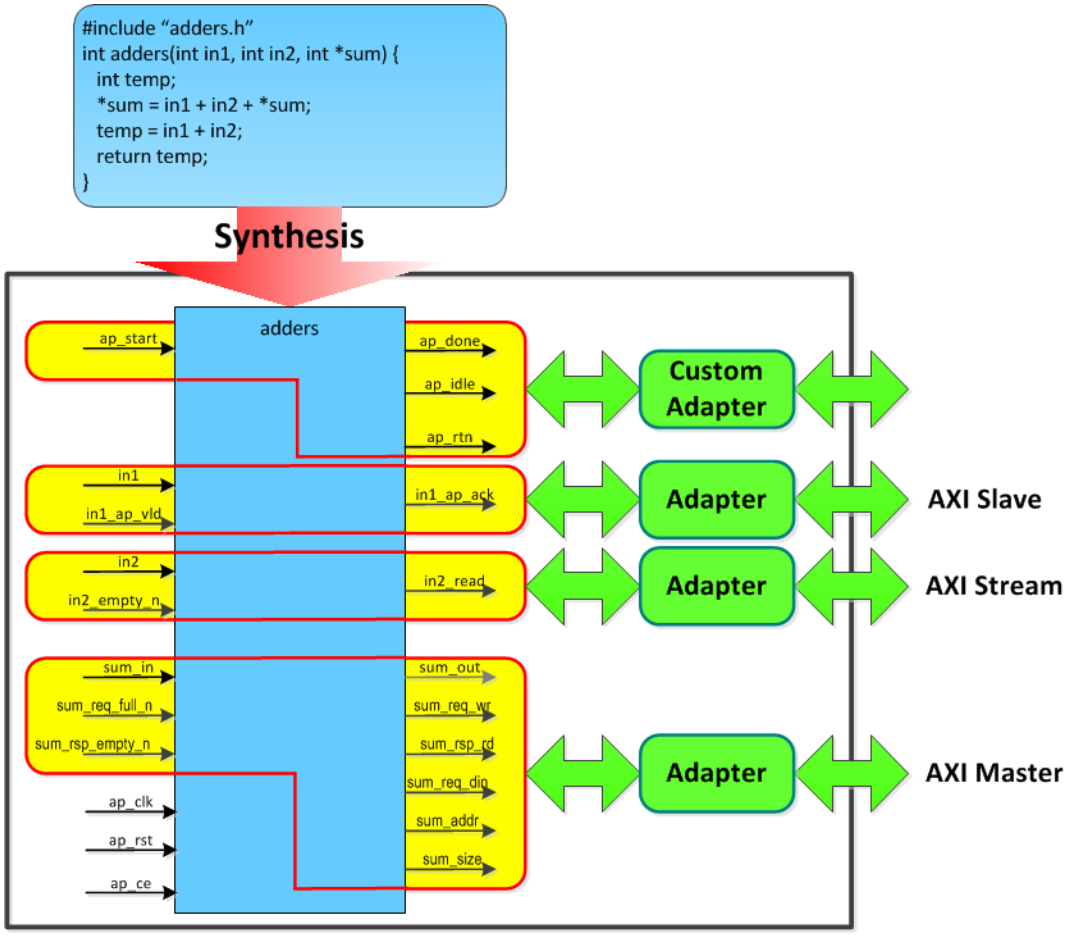
Vivado HLS Tool Interfaces

- Summary can be found in the Synthesis report



Vivado HLS Tool I/O Options: Bus Interfaces

- **Bus interfaces**
  - For use in Vivado IP integrator environment
- **Bus protocols**
  - AXI4-stream
  - AXI4-master
  - AXI4-slave



Creating Bus Interfaces

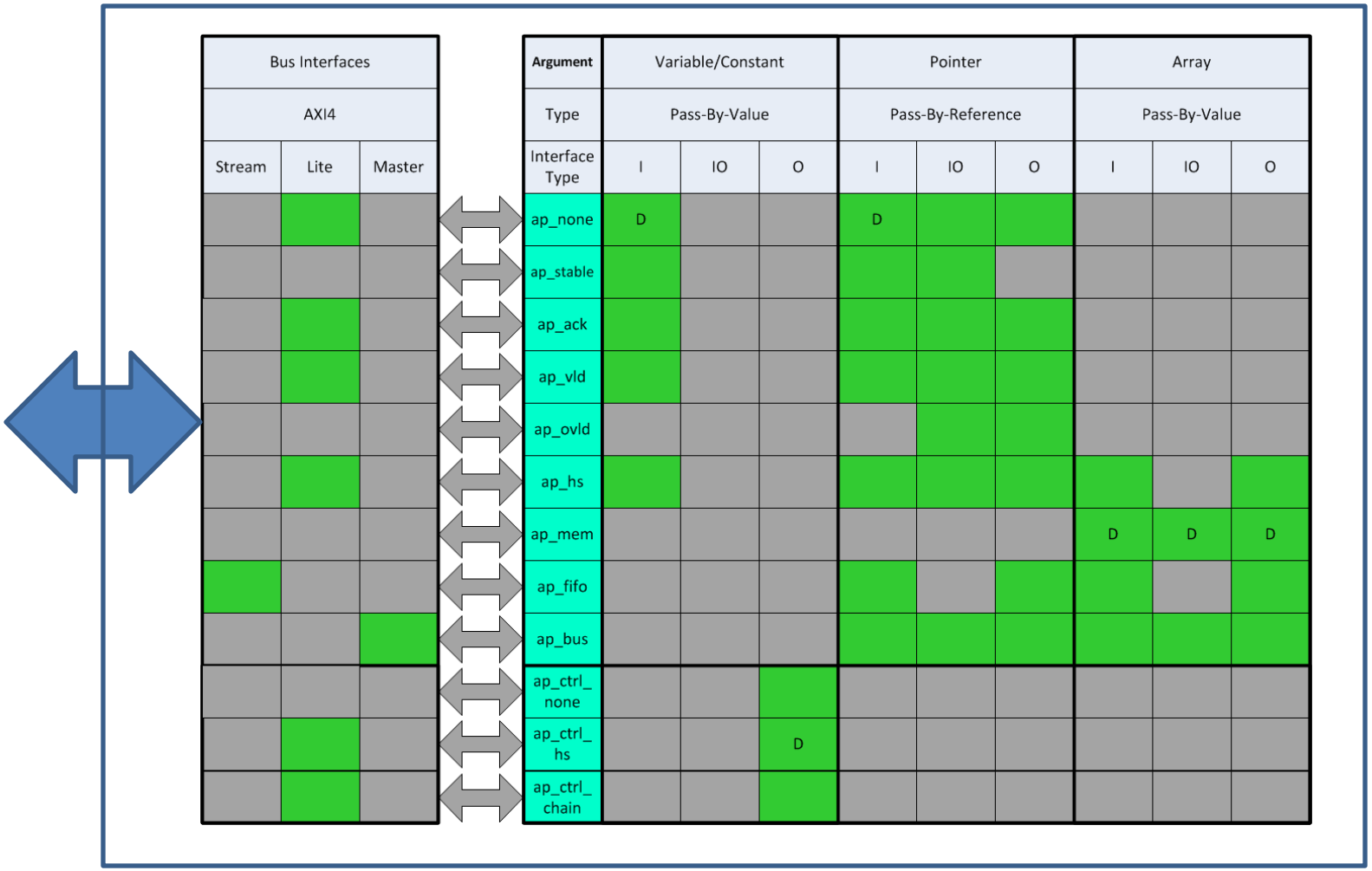
- Introduction
- Vivado HLS Tool Flow
- Interface Synthesis
- **Creating Bus Interfaces**
- Summary

Creating a Bus Interface: Overview

- **Begin with verified design**
- **Select the port-level I/O protocol appropriate for adapter interface**
  - Each AXI/signal type requires a specific type of adapter
- **Specify the appropriate RESOURCE for each adapter**
- **Optionally group and rename ports to create interfaces**

Building Up an Interface

- Interface = adapters + arguments



Slave Interface Example

Simple example

```
int foo_top (int *a, int *b, int *c, int *d) {
// defint RTL interfaces
#pragma HLS INTERFACE ap_hs       port=a
#pragma HLS INTERFACE ap_none     port=b
#pragma HLS INTERFACE ap_vld      port=c
#pragma HLS INTERFACE ap_ack      port=d
#pragma HLS INTERFACE ap_ctrl_hs port=return register

// define the interfaces and group into AXI4 slave “slv0”
#pragma HLS RESOURCE core=AXI4LiteS metadata=”-bus_bundle slv0" variable=a
#pragma HLS RESOURCE core=AXI4LiteS metadata=”-bus_bundle slv0" variable=b

// define the interfaces and group into AXI4 slave “slv1”
#pragma HLS RESOUCE core=AXI4LiteS metadata=”-bus_bundle slv1" variable=return
#pragma HLS RESOUCE core=AXI4LiteS metadata=”-bus_bundle slv1" variable=c
#pragma HLS RESOUCE core=AXI4LiteS metadata=”-bus_bundle slv1" variable=d

   *a += *b;
   return (*c + *d);
}
```

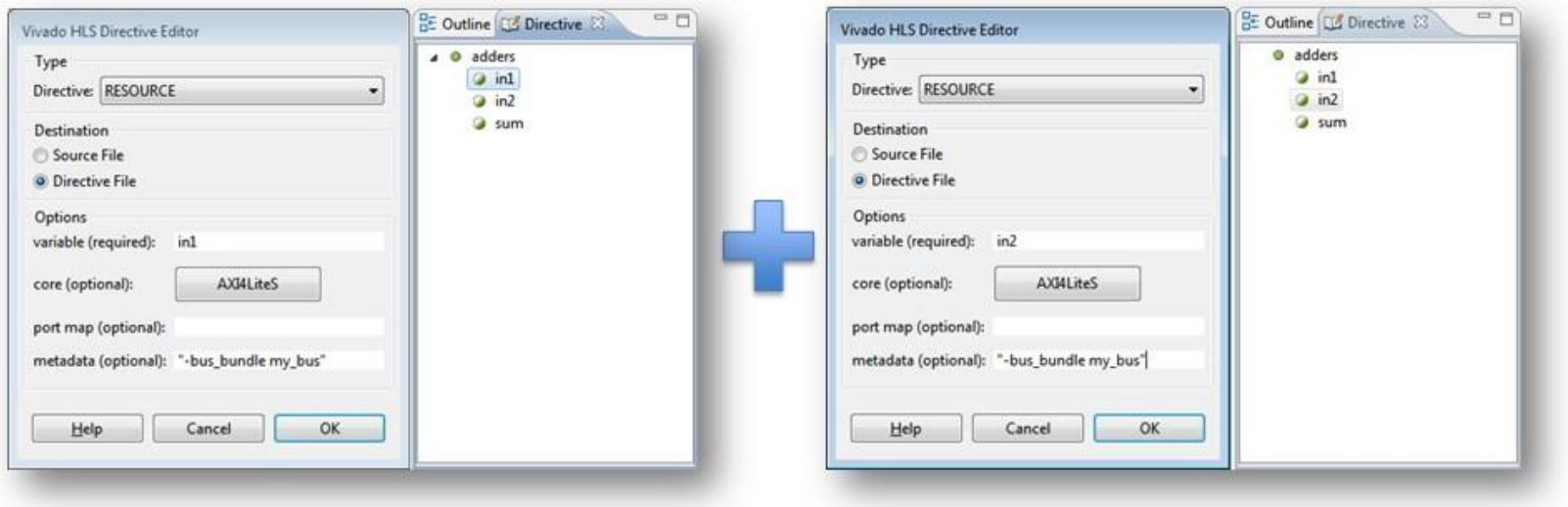
Port a: Synthesized with two-way handshake (ap_hs)
Port b: Synthesized with no IO protocol (ap_none)
Port c: Synthesized with input valid protocol (ap_vld)
Port d: Synthesized with output acknowledge (ap_ack)
Block IO protocols signal added to the design

Port a and b grouped into slave adapter slv0

Port c, d, and return and the block level protocol (associated with the return port) grouped into slave adapter slv1

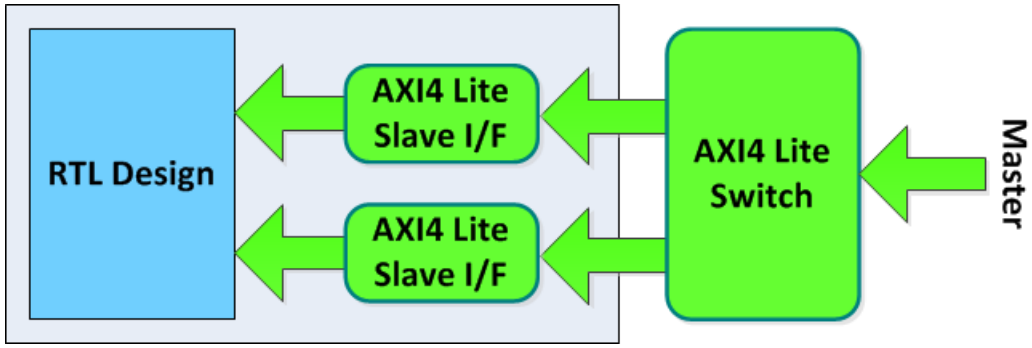
Creating Interfaces

- **Interfaces can be created using a GUI or as meta data**
  - Allow buses or signals to be grouped
  - Optional step
  - Multiple resources can have the same name
    - Use `-bus_bundle` option in the meta data field
    - Will be grouped into the same slave interface
  - In the example below: RTL buses/ports "in1" and "in2" are grouped into a common AXI4-Lite slave interface named "my_bus"



Slave Interface Example RTL

- **This example shows the IP consisting of the RTL design and two slave interfaces**
  - One AXI slave interface might control the IP
  - The other can be used to move data
- **Note that a switch is present so that multiple slaves can be connected to a single master**
  - Alternatively, each slave port can be directly connected to a single master



Summary

- Introduction
- Vivado HLS Tool Flow
- Interface Synthesis
- Creating Bus Interfaces
- **Summary**

Summary

- **Vivado HLS tool converts C-based sources to RTL**
- **Vivado HLS tool provides complete platform**
  - C validation to IP creation
- **Use directives to meet performance**
  - Override default behavior
- **Interface synthesis includes handshake and control ports**
- **AXI interfaces can be created**
  - Choice of bus adapter is a function of the C variable type (pointer, etc.)