Managing Interface

Objectives

- After completing this module, you will be able to:
  - List the types of IO abstracted in Vitis HLS
  - List various basic and optional IO ports handled in Vitis HLS
  - State how a design can be synthesized as combinatorial and sequential
  - Distinguish between block-level and port-level IO protocols
  - State how pointer interfaces are implemented

Outline

- Introduction
- Vitis HLS Design Methodology
- Block Level Protocols
- Port Level Protocols
- AXI Adapter Interface Protocols
- Summary

Introduction to HLS Design Methodology

Vitis HLS methodology:

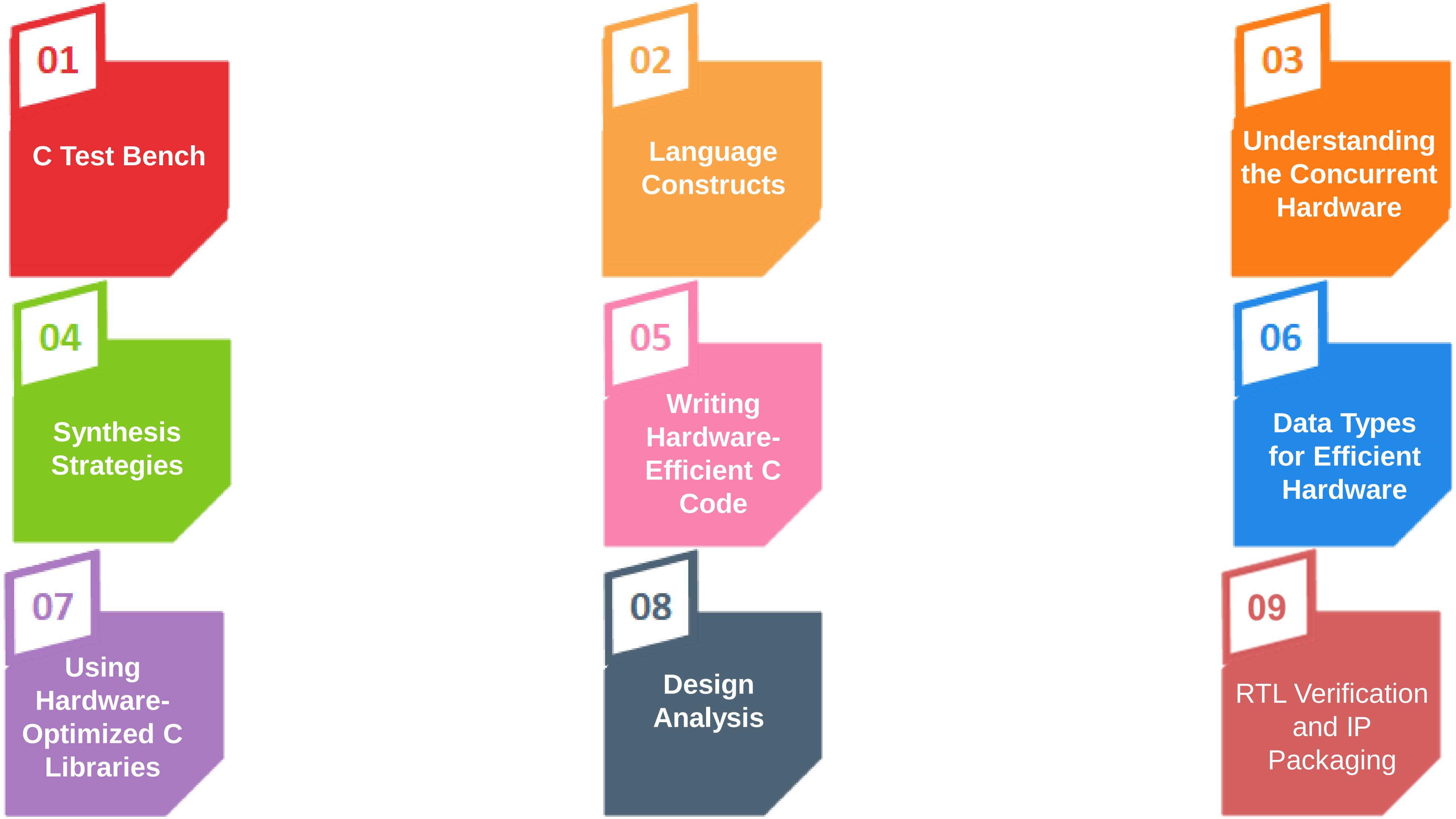
- Provides details on programming techniques to apply to C/C++ code
- Discussion of the various optimization methods
- Intended to provide real-world techniques and design details to improve performance
- Number of optimizations that are applied to the C/C++ code using directives/pragmas in the code
- Highly recommended to review the methodology and obtain a global perspective of high-level synthesis optimizations

HLS Design Methodology

Baseline Performance	Simulating the Design	Validate the C function
	Synthesizing the Design	Convert the C/C++ design to RTL implementation
C/C++ Coding Style	C/C++ Coding Style	Adjust for supported and unsupported constructs
Interfaces	Defining the Interfaces	Synthesize arguments of the top-level function in HLS design into interfaces and ports that group multiple signals
Optimization Techniques	Optimizing for Throughput	Improve throughput or reduce the initiation interval by pipelining loops and functions, partitioning arrays, and managing the dependencies
	Optimizing for Latency	Use latency constraints Merge loops and flatten loops
	Optimizing for Area	Manage data types Inline functions and reshape arrays
	Optimizing for Logic	Optimize the logic by inferring shift registers and controlling operator pipelining
	Optimizing for Burst Transfers	Control burst transfer behavior

High-Level Synthesis (HLS)

HLS Design Methodology provides a set of best practices while creating C/C++ based designs



01 C Test Bench

02 Language Constructs

03 Understanding the Concurrent Hardware

04 Synthesis Strategies

05 Writing Hardware-Efficient C Code

06 Data Types for Efficient Hardware

07 Using Hardware-Optimized C Libraries

08 Design Analysis

09 RTL Verification and IP Packaging

The Key Attributes of C code : IO

Code

```
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>>2;
}
```

Functions: All code is made up of functions which represent the design hierarchy: the same in hardware

Input & Outputs: The arguments of the top-level function must be transformed to hardware interfaces with an IO protocol

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 an impact on area and performance.

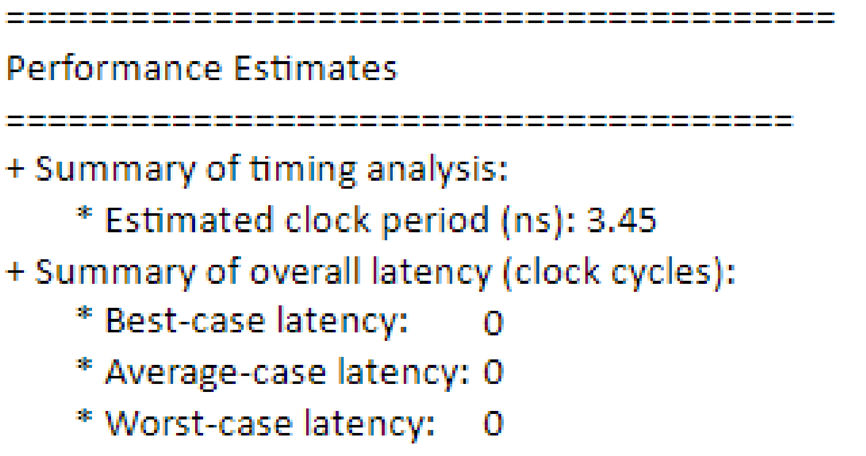
Arrays: Arrays are used often in C code. They can impact the device area and become performance bottlenecks.

Operators: Operators in the C code may require sharing to control area or be assigned to specific hardware implementations to meet performance

Example : Combinational Design

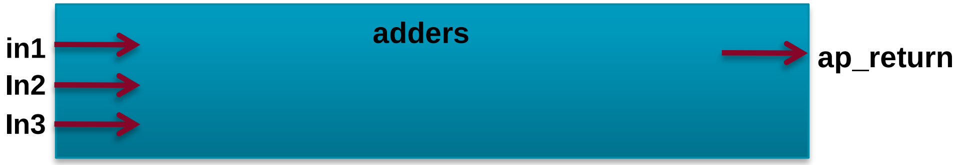
- Simple Adder Example
  - Output is the sum of 3 inputs

```
#include "adders.h"
int adders(int in1, int in2, int in3) {

        int sum;
        sum = in1 + in2 + in3;
        return sum;
}
```



```
==========================================
Performance Estimates
========================================
+ Summary of timing analysis:
    * Estimated clock period (ns): 3.45
+ Summary of overall latency (clock cycles):
    * Best-case latency:       0
    * Average-case latency: 0
    * Worst-case latency:     0
```

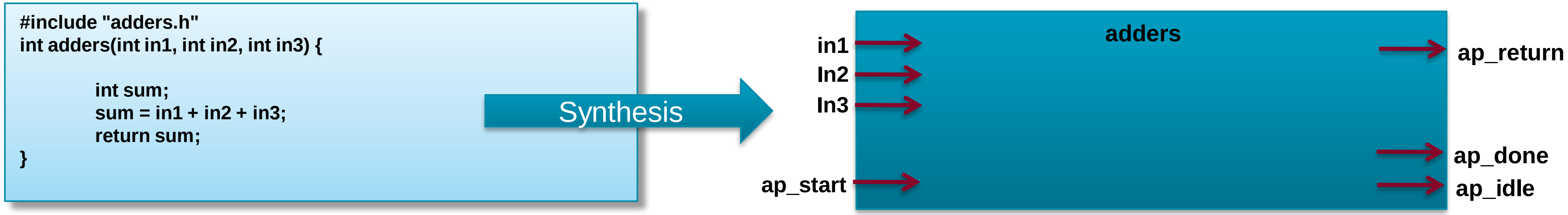
- Synthesized with 100 ns clock
  - All adders can fit in one clock cycle
  - **Combinational design**



  - The function return becomes RTL port ap_return
  - No handshakes are required or created in this example

Example : Sequential Design

- The same adder design is now synthesized with a 3ns clock
  - The design now takes more than 1 cycle to complete
  - Vitis HLS creates a **<u>sequential</u>** design with the default port types



```
#include "adders.h"
int adders(int in1, int in2, int in3) {

        int sum;
        sum = in1 + in2 + in3;
        return sum;
}
```

- By Default ..
  - Block level handshake signals are added to the design
  - ap_start: when to start
  - ap_done: when the design has completed the operation
  - ap_idle: when the design is idle

Vitis HLS IO Options

- Vitis HLS has four types of IO
  1. Data ports created by the original top-level C function arguments
  2. IO protocol signals added at the Block-Level
  3. IO protocol signals added at the Port-Level
  4. IO protocol signals added externally as IP Interfaces
- Data Ports
  - These are the function arguments/parameters
- Block-Level Interfaces (optional)
  - An interface protocol which is added at the block level
  - Controls the addition of block level control ports: start, idle, done, and ready
- Port-Level interfaces (optional)
  - IO interface protocols added to the individual function arguments
- IP interfaces (optional)
  - Added as external adapters when exported as an IP

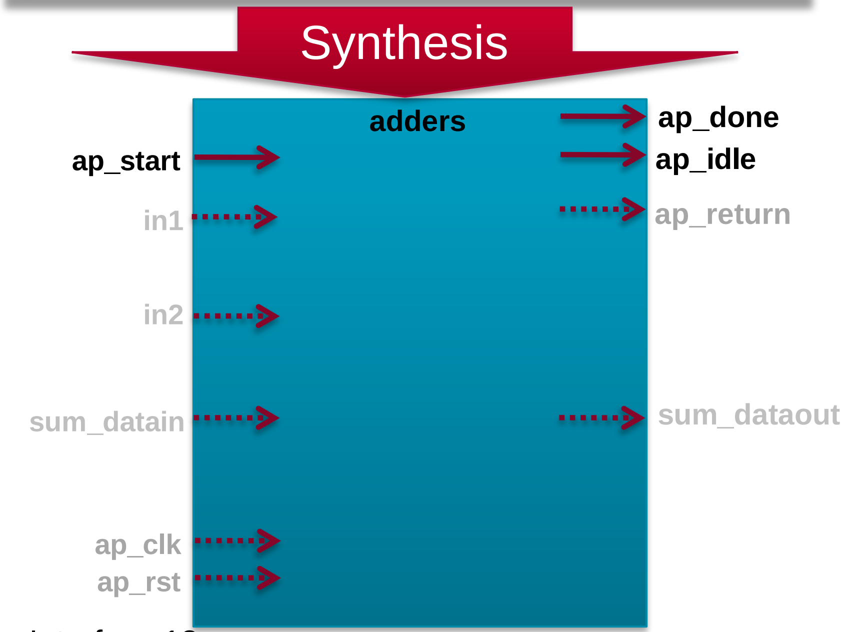
Interface Synthesis

- Type of interface depends on
  - Data type
  - Direction of the parameters of the top-level function
  - Target flow for the active solution
  - Default interface configuration settings
  - Any specified INTERFACE pragmas or directives
- Interface defines three elements of the kernel
  - Channels for data to flow into or out of the HLS design
  - Port protocol that is used to control the flow of data through the data channel, defining when the data is valid and when it can be read or written (port-level protocols)
  - Execution control scheme for the HLS design, specifying the operation of the kernel or IP as pipelined or sequential (block-level protocols)

Block Level Protocol

Adder Example

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

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

        return  temp;
}
```

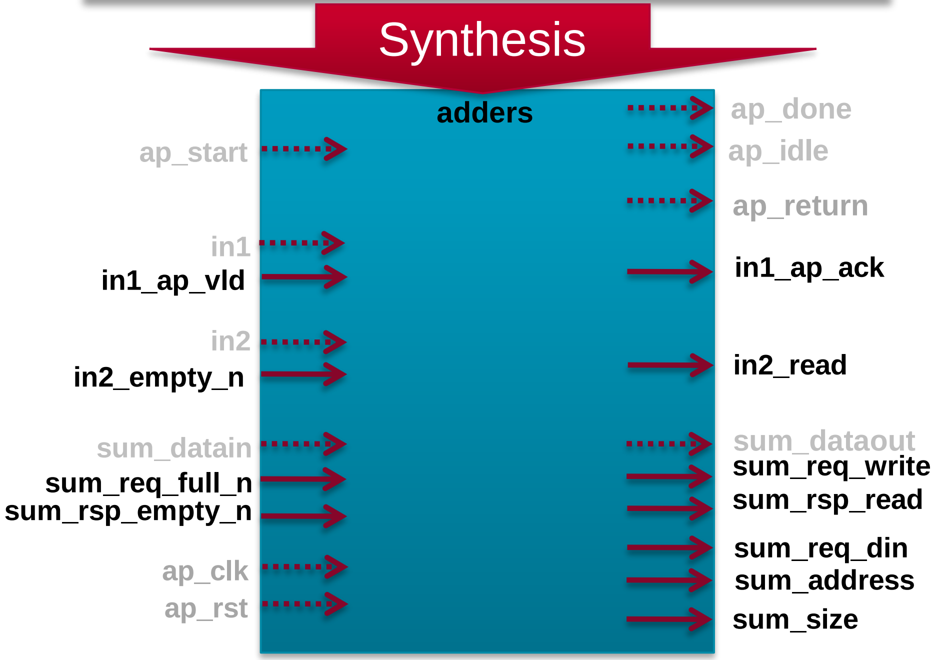


- **Block Level Protocol**
  - An IO protocol added at the RTL block level
  - Controls and indicates the operational status of the block
- **Block Operation Control**
  - Controls when the RTL block starts execution (ap_start)
  - Indicates if the RTL block is idle (ap_idle) or has completed (ap_done)
- **Complete and function return**
  - The 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

Port-Level IO Protocols

Adder Example

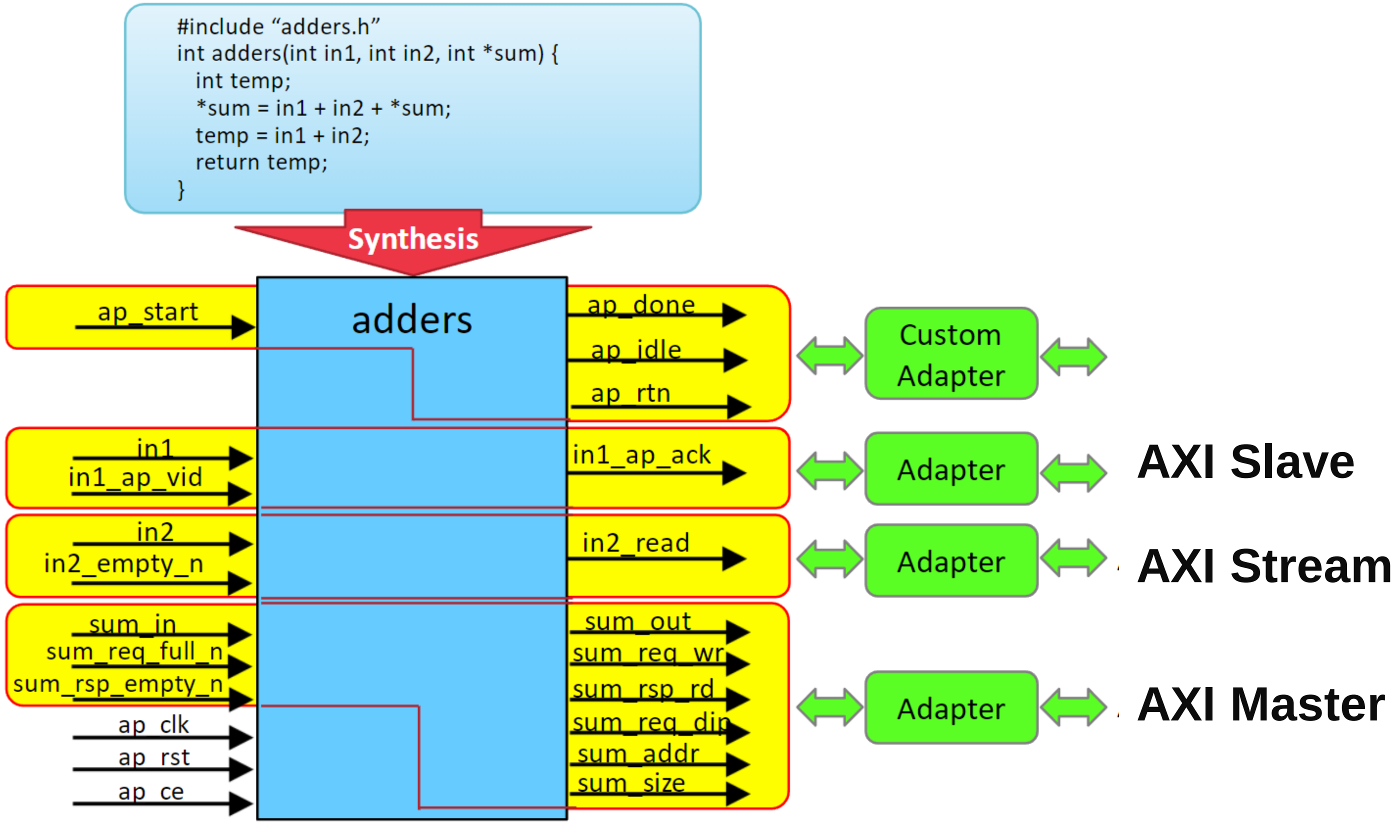
```
#include "adders.h"
int adders(int in1, int in2, int *sum) {

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

            return  temp;
}
```



- **Port IO Protocols**
  - An IO protocol added at the port level
  - Sequences the data to/from the data port
- **Interface Synthesis**
  - The design is automatically synthesized to account for IO signals (enables, acknowledges etc.)
- **Select from a pre-defined list**
  - The IO protocol for each port can be selected from a list
  - Allows the user to easily connect to surrounding blocks
- **Non-standard Interfaces**
  - Supported in C++ using an arbitrary protocol definition

Vitis HLS IO Optional: AXI Adapter Protocols

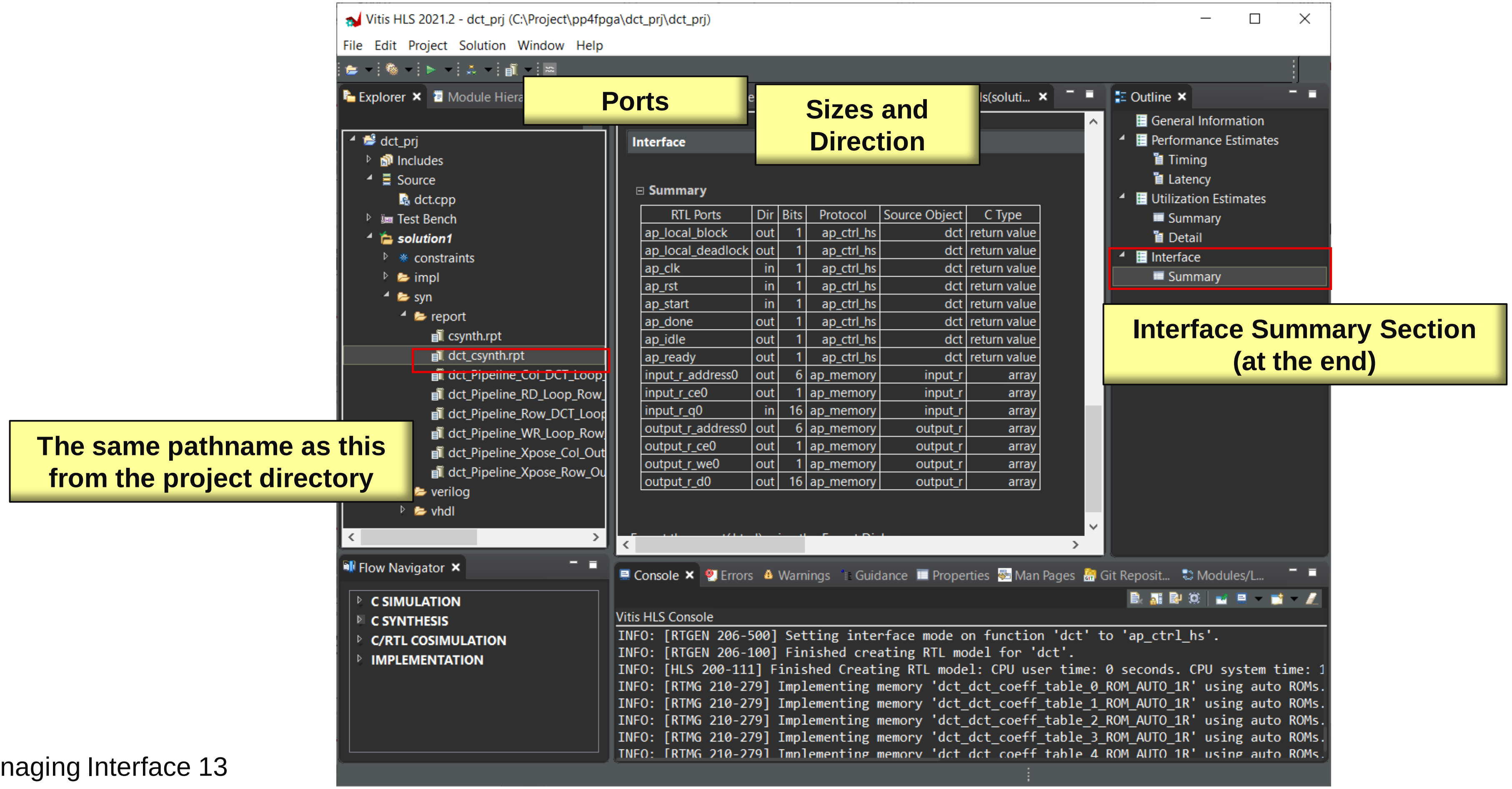
- Bus interface are used for the both Vivado IP flow and Vitis kernel flow
- Bus interface include:
  - AXI4 Slave
  - AXI4 Stream
  - AXI4 Master
- IP adapter cores are used to connect the I/O protocols to bus interfaces
- These Interface are implemented in the RTL wrapper used in the IP generation flow



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

Vitis HLS Interfaces Summary

- Where do you find the summary?
  - In the Synthesis report



Ports

Sizes and Direction

Interface Summary Section (at the end)

The same pathname as this from the project directory

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_local_block	out	1	ap_ctrl_hs	dct	return value
ap_local_deadlock	out	1	ap_ctrl_hs	dct	return value
ap_clk	in	1	ap_ctrl_hs	dct	return value
ap_rst	in	1	ap_ctrl_hs	dct	return value
ap_start	in	1	ap_ctrl_hs	dct	return value
ap_done	out	1	ap_ctrl_hs	dct	return value
ap_idle	out	1	ap_ctrl_hs	dct	return value
ap_ready	out	1	ap_ctrl_hs	dct	return value
input_r_address0	out	6	ap_memory	input_r	array
input_r_ce0	out	1	ap_memory	input_r	array
input_r_q0	in	16	ap_memory	input_r	array
output_r_address0	out	6	ap_memory	output_r	array
output_r_ce0	out	1	ap_memory	output_r	array
output_r_we0	out	1	ap_memory	output_r	array
output_r_d0	out	16	ap_memory	output_r	array

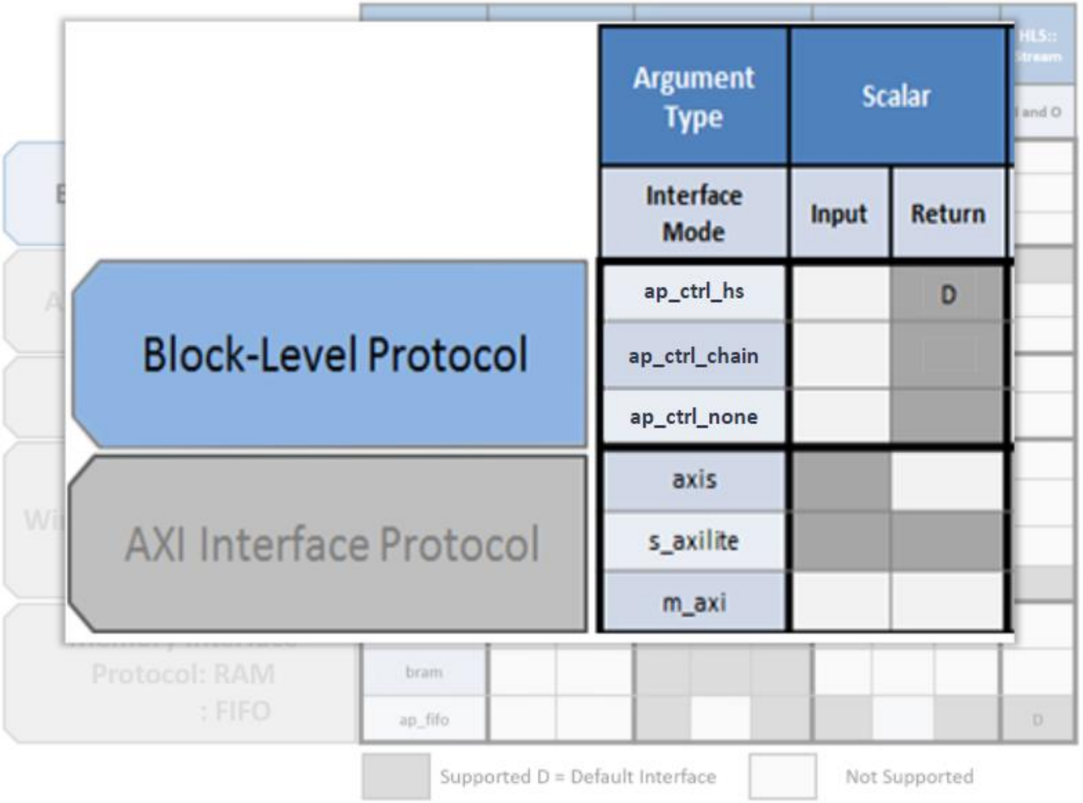
Vitis HLS Console

```
INFO: [RTGEN 206-500] Setting interface mode on function 'dct' to 'ap_ctrl_hs'.
INFO: [RTGEN 206-100] Finished creating RTL model for 'dct'.
INFO: [HLS 200-111] Finished Creating RTL model: CPU user time: 0 seconds. CPU system time: 1
INFO: [RTMG 210-279] Implementing memory 'dct_dct_coeff_table_0_ROM_AUTO_1R' using auto ROMs.
INFO: [RTMG 210-279] Implementing memory 'dct_dct_coeff_table_1_ROM_AUTO_1R' using auto ROMs.
INFO: [RTMG 210-279] Implementing memory 'dct_dct_coeff_table_2_ROM_AUTO_1R' using auto ROMs.
INFO: [RTMG 210-279] Implementing memory 'dct_dct_coeff_table_3_ROM_AUTO_1R' using auto ROMs.
INFO: [RTMG 210-279] Implementing memory 'dct_dct_coeff_table_4_ROM_AUTO_1R' using auto ROMs.
```

Block Level Protocols

Block-level protocol

- Three types of block-level protocols:
  - Ap_ctrl_hs
  - Ap_ctrl_chain
  - Ap_ctrl_none
- Function return -> specified as an AXI4-Lite protocol using s_axilite
  - All the ports in the block-level protocol are grouped into the AXI4-Lite protocol
  - Common practice when another device, such as a CPU or a processor, is used to configure and control when the block starts and stops operation



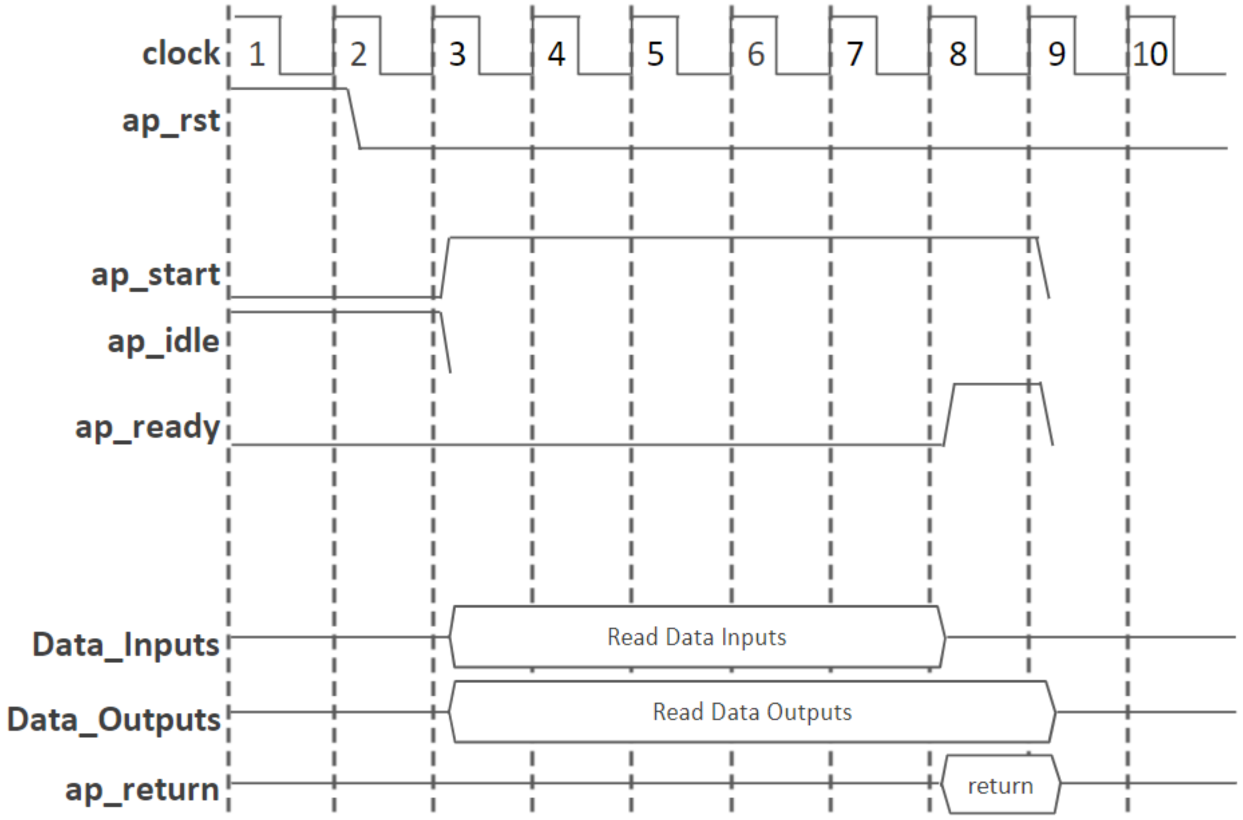
	Argument Type	Scalar	
	Interface Mode	Input	Return
Block-Level Protocol	ap_ctrl_hs		D
	ap_ctrl_chain		▨
	ap_ctrl_none		▨
AXI Interface Protocol	axis	▨	
	s_axilite	▨	▨
	m_axi		

▨ Supported D = Default Interface □ Not Supported

HLS:: Stream
Protocol: RAM
: FIFO
bram
ap_fifo
D

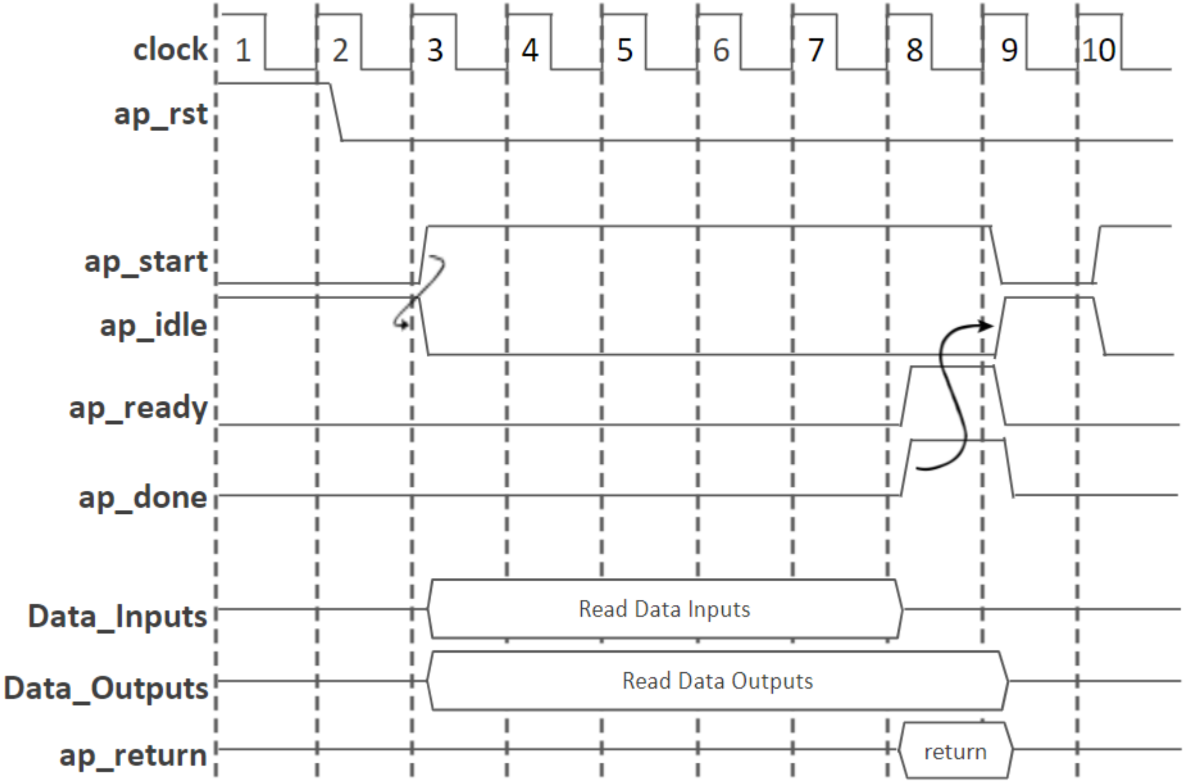
AP_CTRL_HS

- After the reset signal is applied:
  - 1. Design starts when “ap_start” control signal goes high
  - 2. ap_start signal should remain high until ap_ready goes high
  - 3.Output “ap_idle” goes low when “ap_start” is sampled high
  - 4. Data can now be read on the input ports and write on the output port
    - First input data can be sampled on the fist clock edge after “ap_idle” goes low.

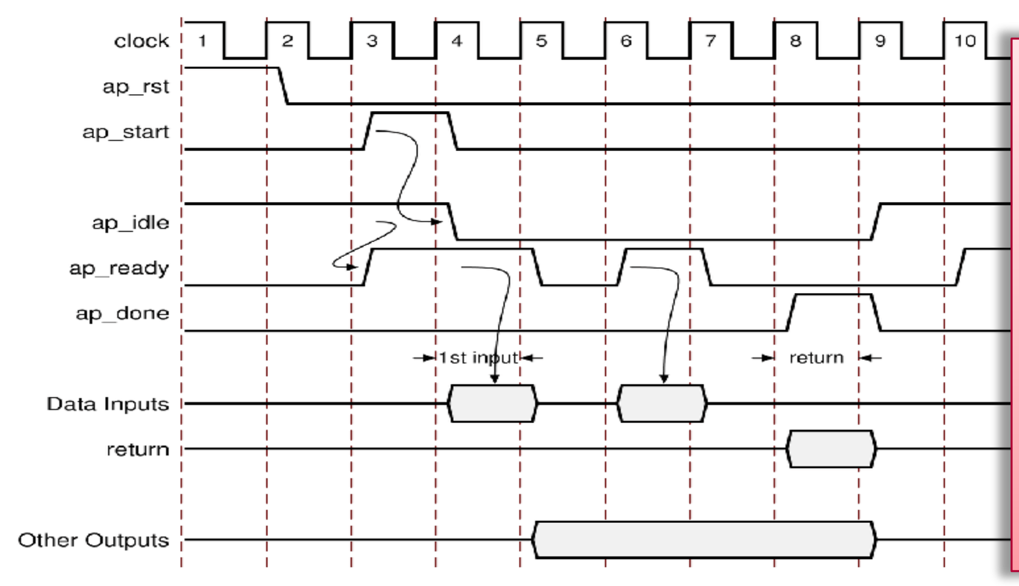


AP_CTRL_HS

- 5. When the clock completes all operations, any return value is written to port ap_return
  - If there was no function return, there is no ap_return port on the RTL
  - Other outputs may be written to at any time until the block completes and are independent of this I/O protocol
- 6. Output “ap_done” goes high when the block completes operation
- 7. Idle signal goes high one cycle after “ap_done” and remains high until the next time “ap_start” is sampled high.
- 8. If the “ap_start” signal is high when “ap_done” goes High:
  - “ap_idle” signal remains Low
  - Block immediately starts its next execution (or next transaction)
  - Next input can be read on the next clock edge



Pipelined Designs

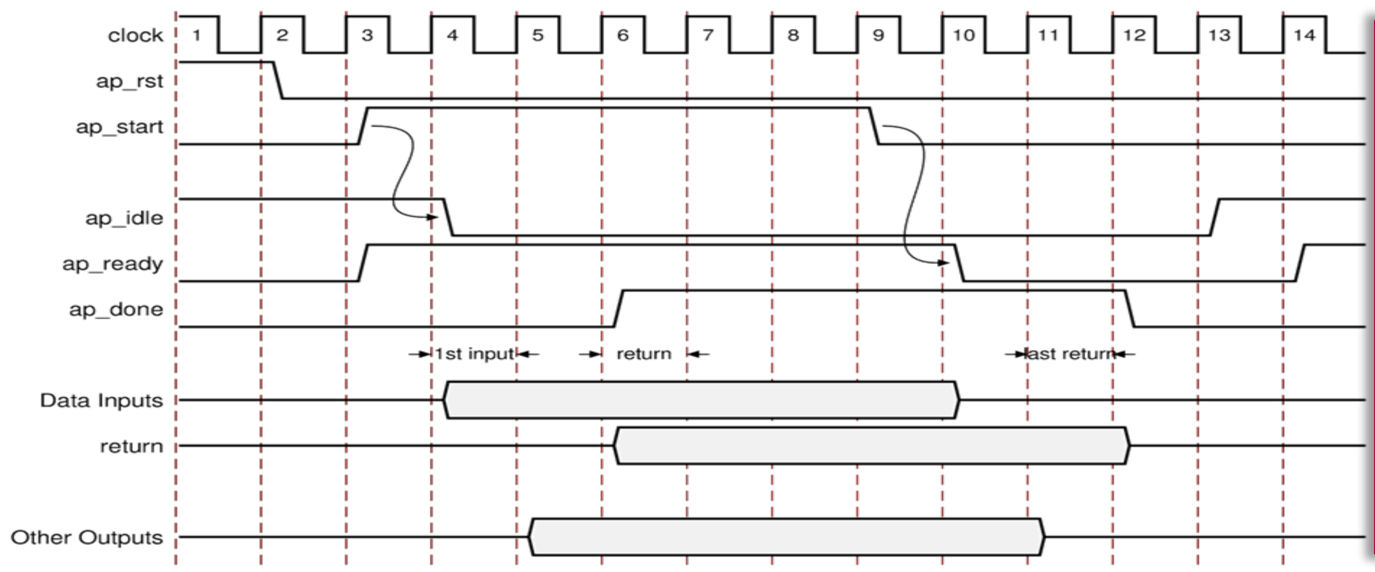


- After reset, ap_ready goes high if idle is high
- Start applied : Idle goes low, apply first data
- **Inputs will be read when ap_ready is high and idle is low**
- Output ap_done goes high on the final cycle
- Idle goes high one cycle after done
- The block waits for next start

- **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 II
- **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: II = 1

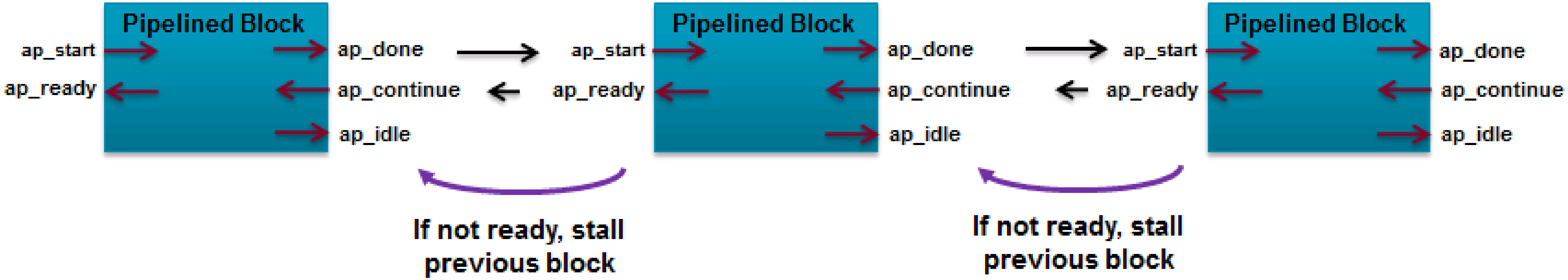


- Will start with ap_start
- Will keep reading input data since II = 1
- After the first output, ap_done will remain high
- If ap_start goes is sampled low, ap_ready will go low immediately

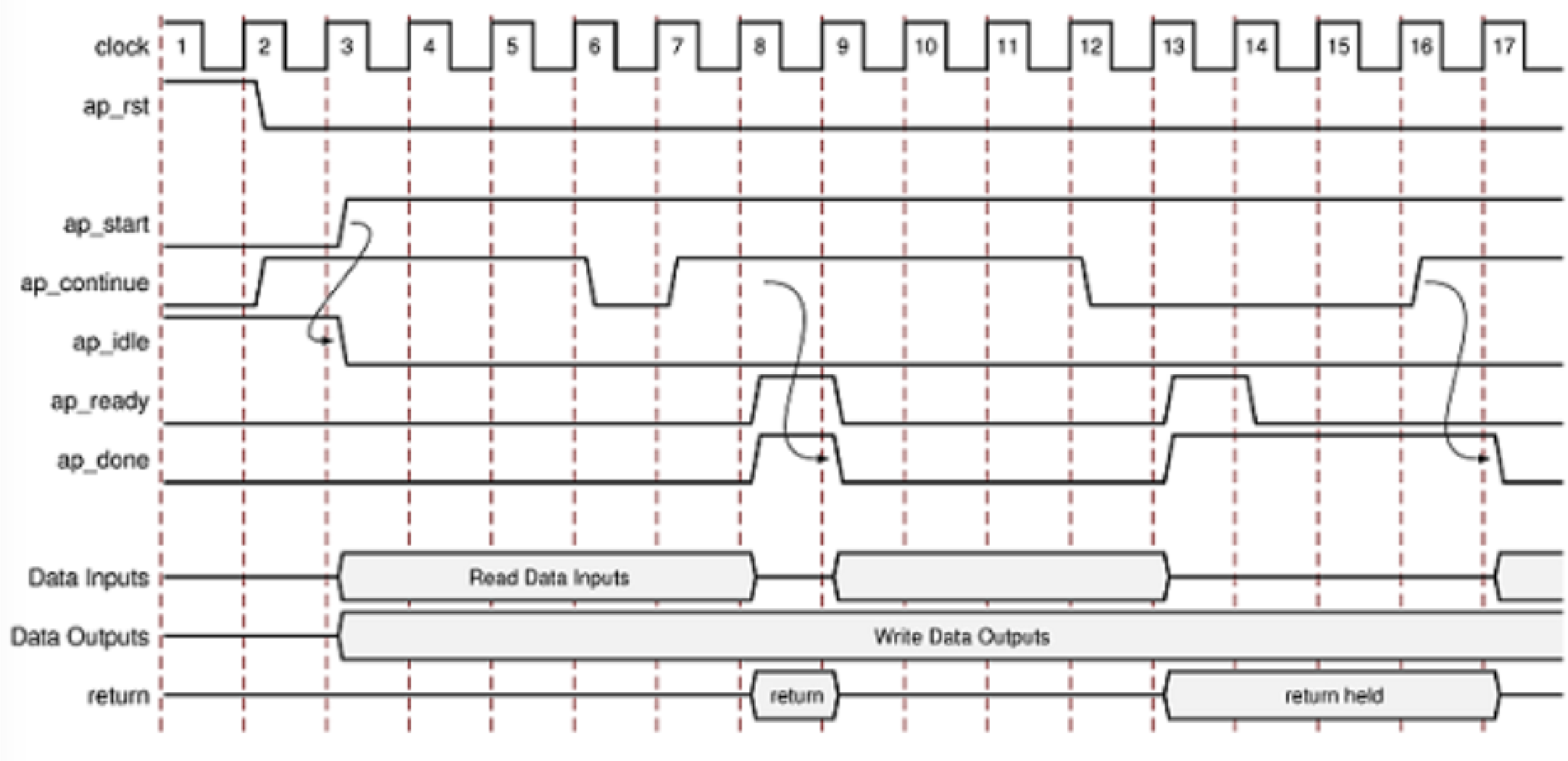
- **Input Data when II=1**
  - It can be expected that ap_ready remains high and data is continuously read
  - The design will only stop processing when ap_start is de-asserted
- **Output Data when II=1**
  - After the first output, ap_done will remain high while there are samples to process
    - Assuming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

AP_CTRL_CHAIN

- Protocol to support pipeline chains: ap_ctrl_chain
  - Similar to ap_ctrl_hs but with additional signal ap_continue
  - Allows blocks to be easily chained in a pipelined manner
  - ap_ctrl_chain protocol provides back-pressure in systems which allows the downstream blocks to prevent any more data being generated



AP_CTRL_CHAIN



- Asserting the ap_continue signal Low
  - informs the design that the downstream block that consumes the data is not ready to accept new data
  - the design stalls when it reaches the final state of the current transaction
  - the output data is presented on the interface
- the ap_done signal can be asserted High and the design remains in this state until ap_continue is asserted High

If you use the ap_ctrl_none control protocol in your design, you must meet at least one of the conditions for C/RTL co-simulation as described in Interface Synthesis Requirement

AP_CTRL_NONE

- Simplest protocol and handshake signal ports (ap_start, ap_idle, ap_ready, and ap_done) are not created
- The only ports in the RTL design are those specified in the source code
- It is highly recommended to use the auto-restart mode for data-driven kernels, because it allows the host code to start and stop when needed, in an orderly fashion.

Requires

Producer blocks provide data to the input port and output port at the correct time or hold it for the length of transaction

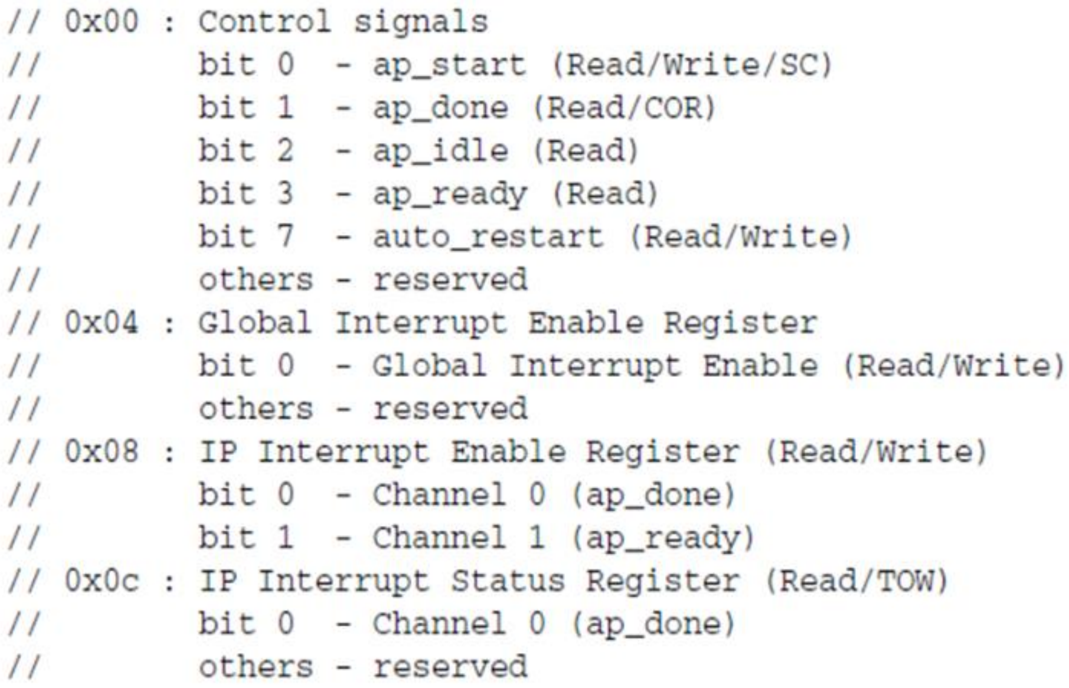
Customer blocks to read the output ports at the correct time

Cautions

If the design is specified to use the block-level IO protocol ap_ctrl_none and the design contains any hls::stream variables which employ non-blocking behavior, C/RTL co-simulation is not guaranteed to be complete.

S_AXILITE

- AXI Slave Lite I/O protocol used to control the HLS block: s_axilite
  - Common practice used when the processing system (PS) or processor is used to configure the HLS block
- Multiple ports can be grouped into the same AXI4 Slave Lite protocol
- When the design is exported to the IP catalog, the output includes C function and header files for the use with the code running on a processor
  - Hardware header file provides a complete list of the memory mapped locations for the ports grouped into the AXI4 Slave Lite protocol
  - Device will read any inputs grouped into the AXI4 Slave Lite protocol from the register in the protocol



```
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/SC)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0  - Channel 0 (ap_done)
//        others - reserved
```

Block-level for C/RTL Co-simulation

- Block Level Handshakes can be disabled
  - Select the function in the directives tab, right-click
    - Select Interface & then **ap_ctrl_none** for no block level handshakes
    - Select Interface & then **ap_ctrl_hs** to re-apply the default

```
# Tcl commands

set_directive_interface -mode ap_ctrl_none “dct“
# Default is on
# set_directive_interface -mode ap_ctrl_hs “dct“
```

- Interface Synthesis Requirements

To use the C/RTL co-simulation feature to verify the RTL design, at least one of the following conditions must be true:

- Top-level function must be synthesized using an ap_ctrl_chain or ap_ctrl_hs block-level protocol
- Design must be purely combinational
- Top-level function must have an initiation interval of 1
- Interfaces must be all arrays that are streaming and implemented with axis or ap_hs interface modes

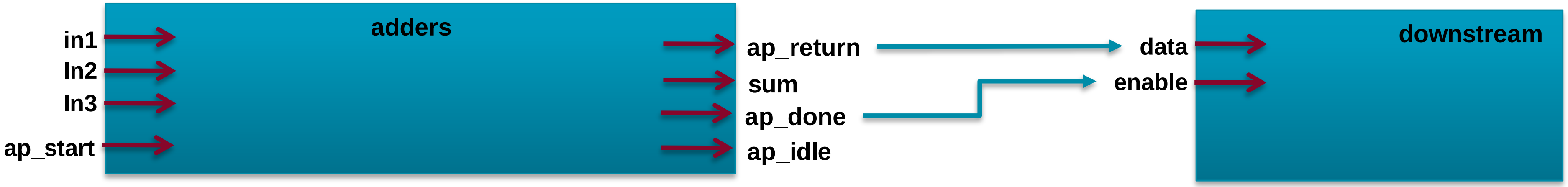
If at least one of these conditions is not met, C/RTL co-simulation halts with the following message:

```
@E [SIM-345] Cosim only supports the following 'ap_ctrl_none' designs: (1)
combinational designs; (2) pipelined design with task interval of 1; (3) designs with
array streaming or hls_stream ports.
@E [SIM-4] *** C/RTL co-simulation finished: FAIL ***
```

Port Level Protocols

Port Level IO protocols

- We've seen how the function return is validated
  - The block level output signal ap_done goes high to indicate the function return is valid
  - This allows downstream blocks to correctly sample the output port



- For other outputs, Port Level IO protocols can be added
  - Allowing upstream and downstream blocks to synchronize with the other data ports
  - The type of protocol depends on the type of C port
    - Pass-by-value scalar
    - Pass-by-reference pointers
    - Pass-by-reference arrays

The starting point is the type of argument used by the C function

Interface Types

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

Notes:

1. Supported
2. D = Default Interface
3. Not Supported

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

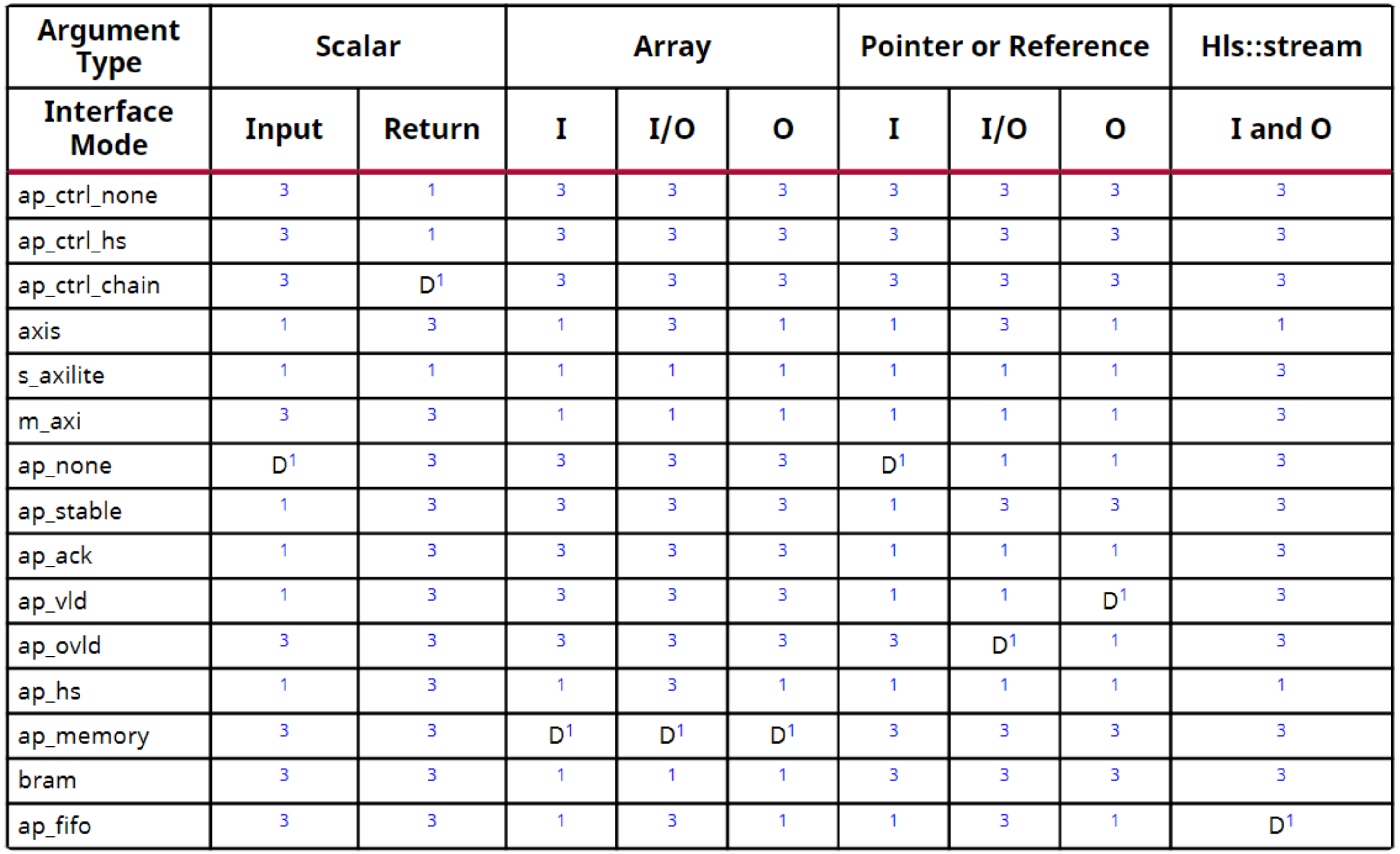
Block Level Protocol

AXI4 Interfaces

No IO Protocol

Wire handshake protocols

Memory protocols
: RAM
: FIFO



Argument Type	Scalar		Array			Pointer or Reference			Hls::stream
Interface Mode	Input	Return	I	I/O	O	I	I/O	O	I and O
ap_ctrl_none	3	1	3	3	3	3	3	3	3
ap_ctrl_hs	3	1	3	3	3	3	3	3	3
ap_ctrl_chain	3	D^1	3	3	3	3	3	3	3
axis	1	3	1	3	1	1	3	1	1
s_axilite	1	1	1	1	1	1	1	1	3
m_axi	3	3	1	1	1	1	1	1	3
ap_none	D^1	3	3	3	3	D^1	1	1	3
ap_stable	1	3	3	3	3	1	3	3	3
ap_ack	1	3	3	3	3	1	1	1	3
ap_vld	1	3	3	3	3	1	1	D^1	3
ap_ovld	3	3	3	3	3	3	D^1	1	3
ap_hs	1	3	1	3	1	1	1	1	1
ap_memory	3	3	D^1	D^1	D^1	3	3	3	3
bram	3	3	1	1	1	3	3	3	3
ap_fifo	3	3	1	3	1	1	3	1	D^1

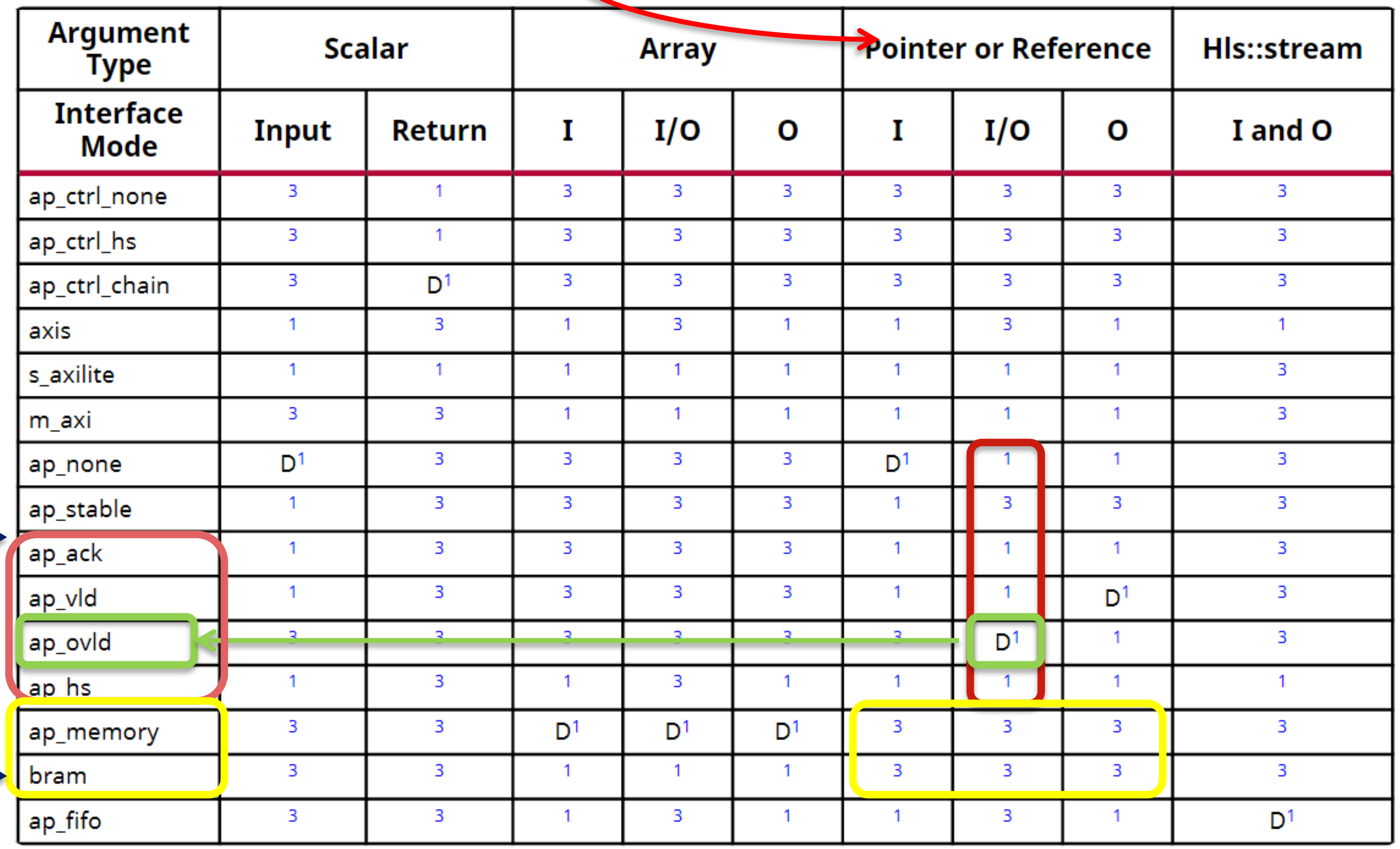
Let’s Look at an Example

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

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

            int temp;

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

            return  temp;
}
```

The port for “Sum” can be any of these interface types

The default for port for “Sum” will be type ap_ovld



Argument Type	Scalar		Array			Pointer or Reference			Hls::stream
Interface Mode	Input	Return	I	I/O	O	I	I/O	O	I and O
ap_ctrl_none	3	1	3	3	3	3	3	3	3
ap_ctrl_hs	3	1	3	3	3	3	3	3	3
ap_ctrl_chain	3	D[1]	3	3	3	3	3	3	3
axis	1	3	1	3	1	1	3	1	1
s_axilite	1	1	1	1	1	1	1	1	3
m_axi	3	3	1	1	1	1	1	1	3
ap_none	D[1]	3	3	3	3	D[1]	1	1	3
ap_stable	1	3	3	3	3	1	3	3	3
ap_ack	1	3	3	3	3	1	1	1	3
ap_vld	1	3	3	3	3	1	1	D[1]	3
ap_ovld	3	3	3	3	3	3	D[1]	1	3
ap_hs	1	3	1	3	1	1	1	1	1
ap_memory	3	3	D[1]	D[1]	D[1]	3	3	3	3
bram	3	3	1	1	1	3	3	3	3
ap_fifo	3	3	1	3	1	1	3	1	D[1]

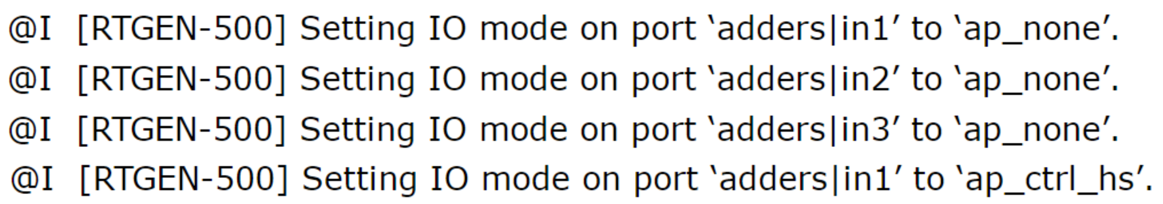
Key:

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

Default IO Protocols

- The default port protocols
  - Input pass-by-value arguments and pointers are implemented as simple wire ports with no associated handshaking signal
    - They should be held stable for the entire transaction
    - There is no way to know when the input will be read
  - Output pointers are implemented with an associated output vided signal to indicate when the output is valid.
    - The output wire have an accompanying output valid signal which can be used to validate them
  - Arrays will default to RAM interfaces (2-port RAM is the default RAM)

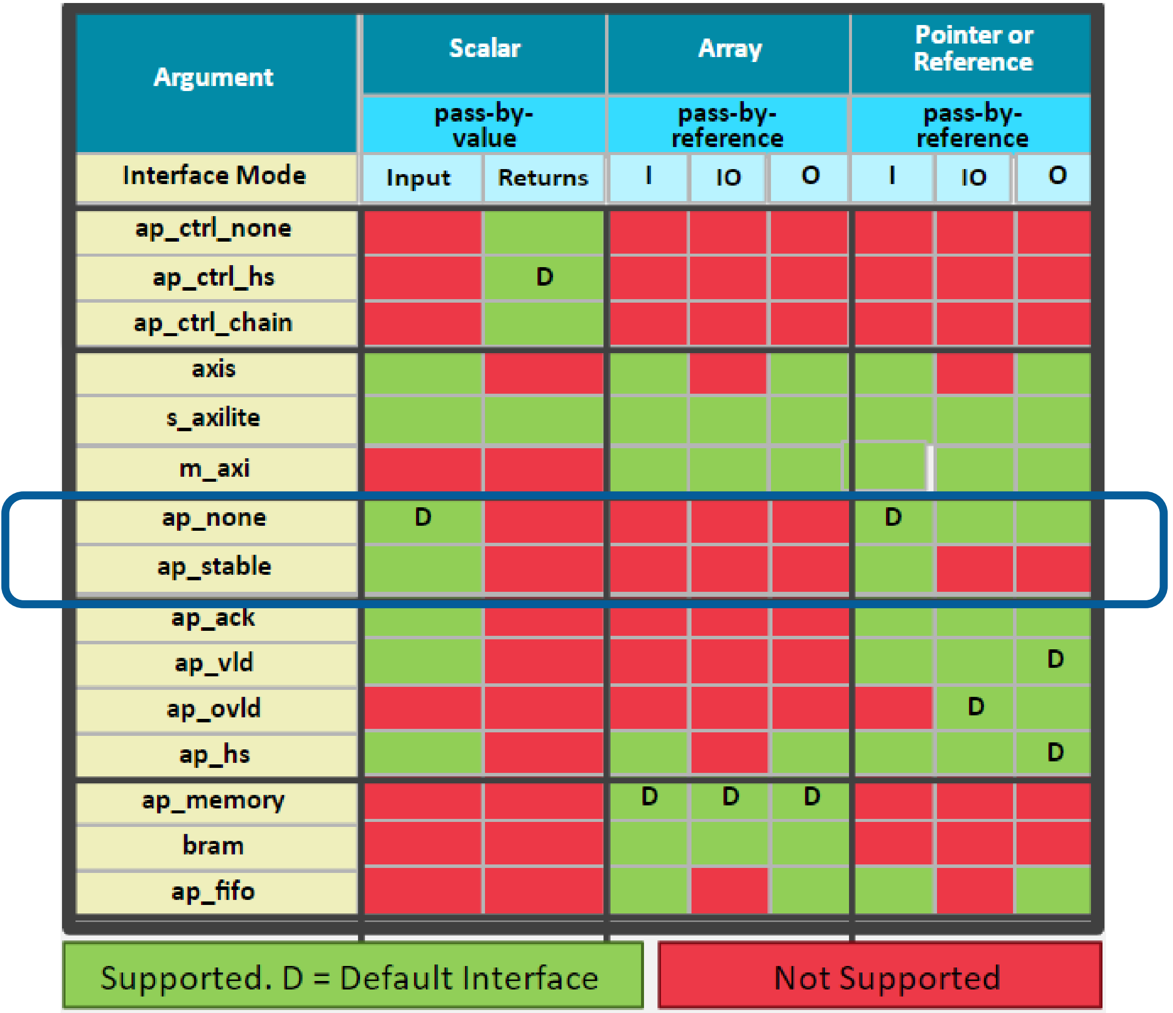
C Argument Type	Default I/O protocol
Inputs	ap_none
Outputs	ap_vld
Inout In port of inout Out port of inout	ap_ovld ap_none ap_vld
Arrays	Ap_memory



```
@I  [RTGEN-500] Setting IO mode on port 'adders|in1' to 'ap_none'.
@I  [RTGEN-500] Setting IO mode on port 'adders|in2' to 'ap_none'.
@I  [RTGEN-500] Setting IO mode on port 'adders|in3' to 'ap_none'.
@I  [RTGEN-500] Setting IO mode on port 'adders|in1' to 'ap_ctrl_hs'.
```

No IO Protocol

- AP_NONE: The default protocol for scalar input ports
  - Protocol ap_none means that no additional protocol signals are added
  - The port will be implemented as just a data port
- Other ports can be specified as ap_none
  - Except arrays which must be a RAM or FIFO interface
- AP_STABLE: An ap_none with fanout benefits
  - The ap_stable type informs High-Level Synthesis that the data applied to this port remains stable during normal operation, but is not a constant value that could be optimized, and the port is not required to be registered
  - Typically used for ports that provides configuration data - data that can change but remains stable during normal operation (configuration data is typically only changed during or before a reset)



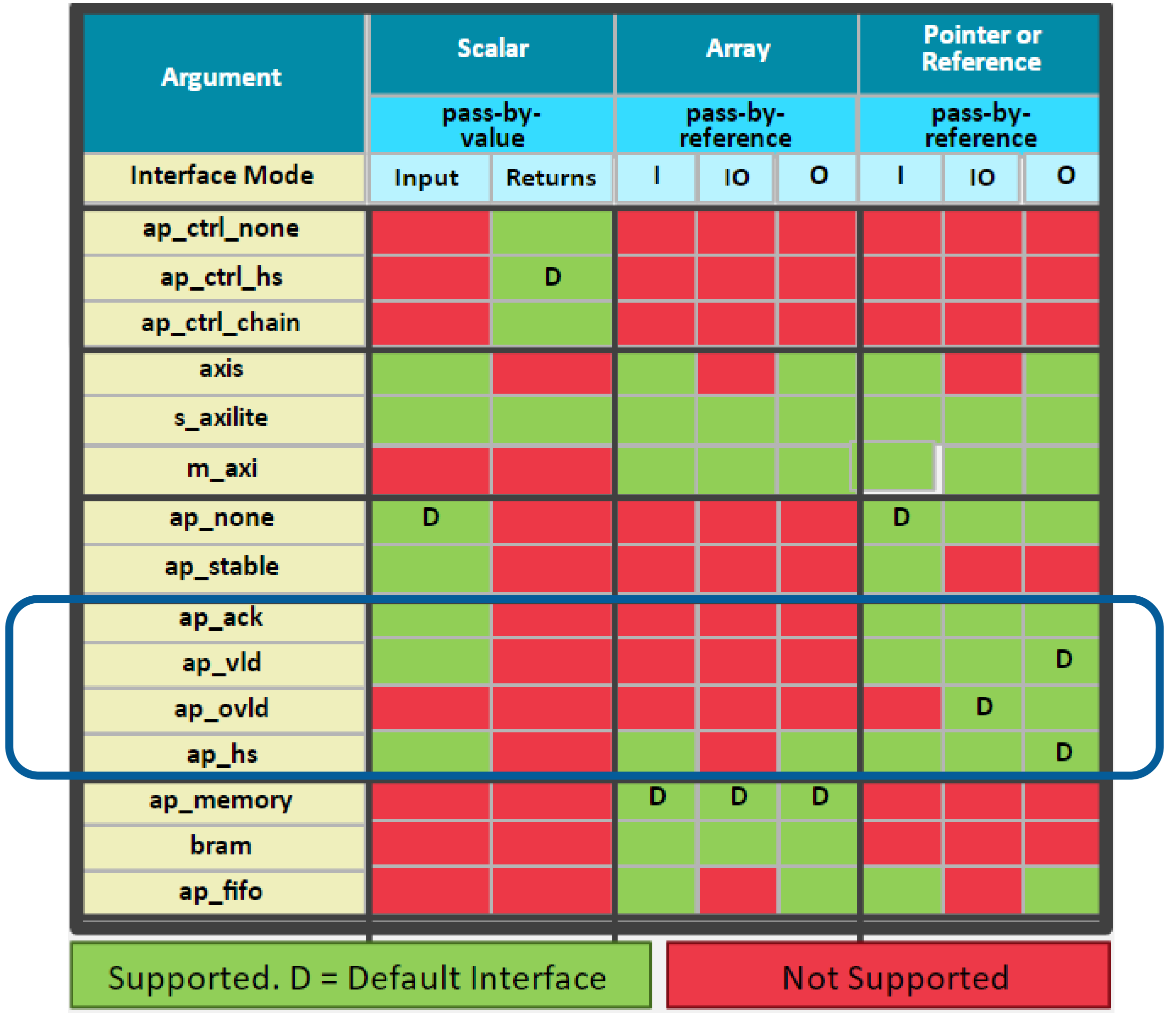
Argument	Scalar		Array			Pointer or Reference		
	pass-by-value		pass-by-reference			pass-by-reference		
Interface Mode	Input	Returns	I	IO	O	I	IO	O
ap_ctrl_none	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥
ap_ctrl_hs	🟥	🟩 D	🟥	🟥	🟥	🟥	🟥	🟥
ap_ctrl_chain	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥
axis	🟩	🟥	🟩	🟥	🟩	🟩	🟥	🟩
s_axilite	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
m_axi	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩
ap_none	🟩 D	🟥	🟥	🟥	🟥	🟩 D	🟩	🟩
ap_stable	🟩	🟥	🟥	🟥	🟥	🟩	🟥	🟥
ap_ack	🟩	🟥	🟥	🟥	🟥	🟩	🟩	🟩
ap_vld	🟩	🟥	🟥	🟥	🟥	🟩	🟩	🟩 D
ap_ovld	🟥	🟥	🟥	🟥	🟥	🟥	🟩 D	🟩
ap_hs	🟩	🟥	🟩	🟥	🟩	🟩	🟩	🟩 D
ap_memory	🟥	🟥	🟩 D	🟩 D	🟩 D	🟥	🟥	🟥
bram	🟥	🟥	🟩	🟩	🟩	🟥	🟥	🟥
ap_fifo	🟥	🟥	🟩	🟥	🟩	🟩	🟥	🟩

🟩 Supported. D = Default Interface

🟥 Not Supported

Wire Handshakes Protocols

- The wire protocols add a valid and/or acknowledge port to each data port
- The wire protocols are all derivatives of protocol ap_hs
  - ap_ack: add an additional acknowledge port
  - ap_vld: add an additional valid port
  - ap_ovld: add an additional valid port to an output
  - ap_hs: adds both
- Output control signals are used to inform other blocks
  - Data has been read at the input by this block (ack)
  - Data is valid at the output (vld)
  - The other block must accept the control signal (this block will continue)
- Input control signals are used to inform this block
  - The output data has been read by the consumer (ack)
  - The input from the producer is valid (vld)
  - **<u>This block will stall</u>** while waiting for the input controls



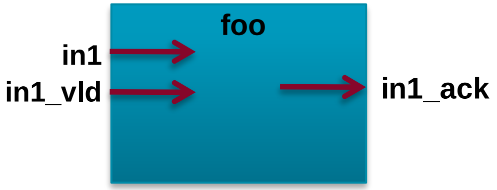
Argument	Scalar		Array			Pointer or Reference		
	pass-by-value		pass-by-reference			pass-by-reference		
Interface Mode	Input	Returns	I	IO	O	I	IO	O
ap_ctrl_none	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥
ap_ctrl_hs	🟥	🟩 D	🟥	🟥	🟥	🟥	🟥	🟥
ap_ctrl_chain	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥
axis	🟩	🟥	🟩	🟥	🟩	🟩	🟥	🟩
s_axilite	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
m_axi	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩
ap_none	🟩 D	🟥	🟥	🟥	🟥	🟩 D	🟩	🟩
ap_stable	🟩	🟥	🟥	🟥	🟥	🟩	🟥	🟥
ap_ack	🟩	🟥	🟥	🟥	🟥	🟩	🟩	🟩
ap_vld	🟩	🟥	🟥	🟥	🟥	🟩	🟩	🟩 D
ap_ovld	🟥	🟥	🟥	🟥	🟥	🟥	🟩 D	🟩
ap_hs	🟩	🟥	🟩	🟥	🟩	🟩	🟩	🟩 D
ap_memory	🟥	🟥	🟩 D	🟩 D	🟩 D	🟥	🟥	🟥
bram	🟥	🟥	🟩	🟩	🟩	🟥	🟥	🟥
ap_fifo	🟥	🟥	🟩	🟥	🟩	🟩	🟥	🟩

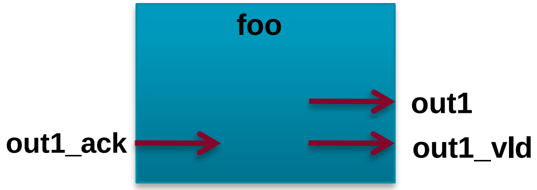
🟩 Supported. D = Default Interface
🟥 Not Supported

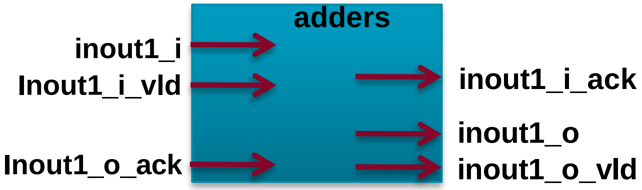
Wire Handshakes Protocols: Ports Generated

- The wire protocols are all derivatives of protocol ap_hs
  - 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

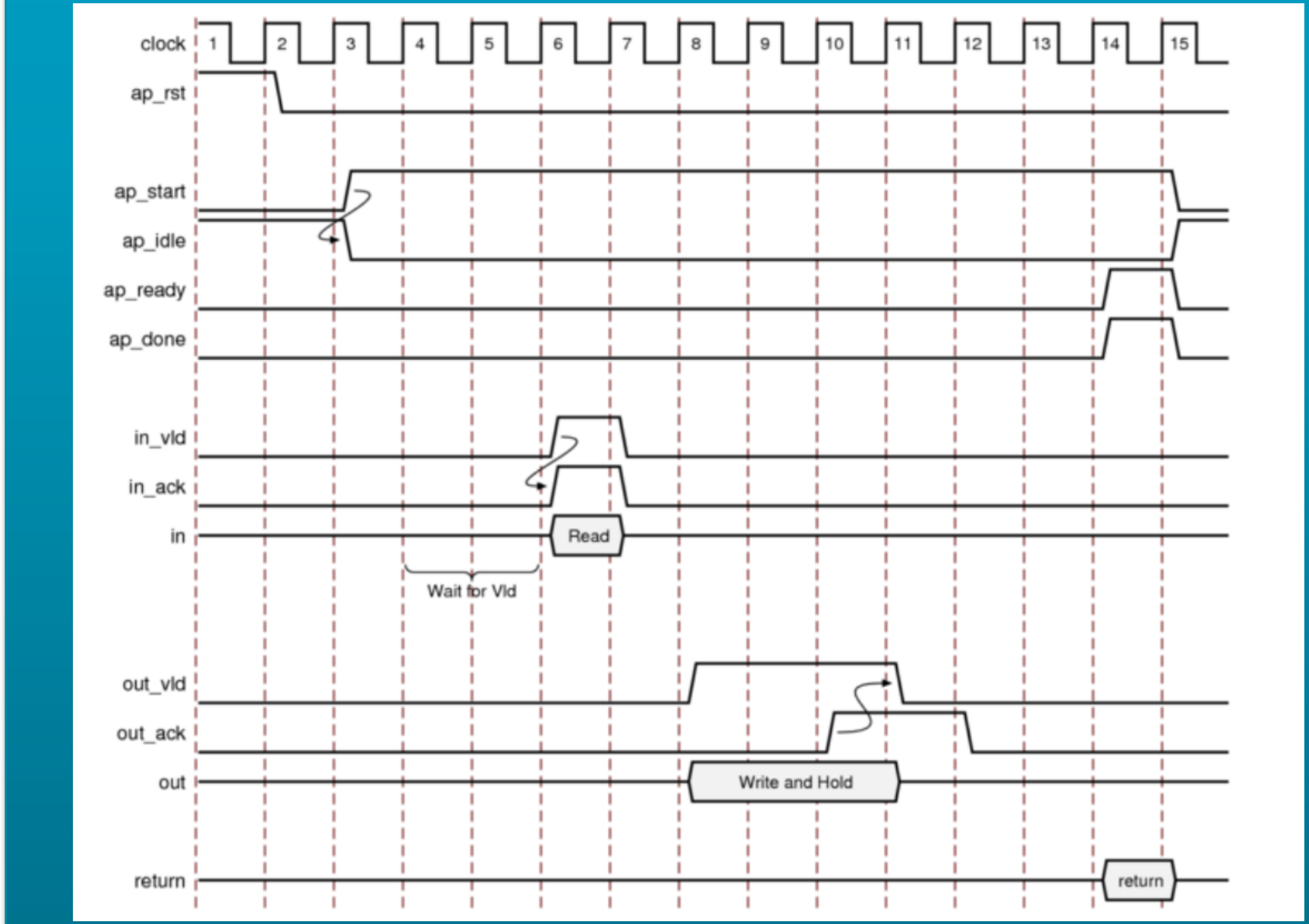
ap_hs is compatible with AXI-Stream







Handshake IO Protocol



- *After start is applied, the block begins normal operation.*

- 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

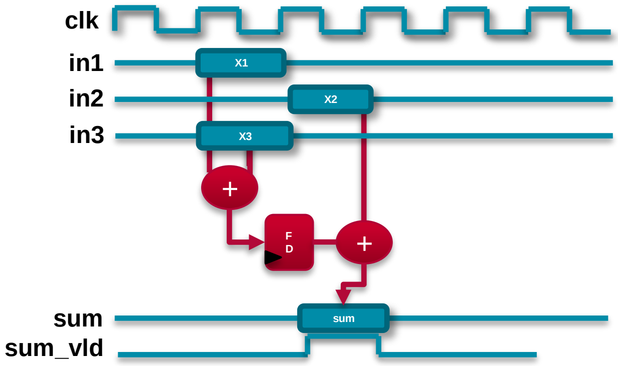
- *Done will be asserted when the function is complete*

Other Handshake Protocols

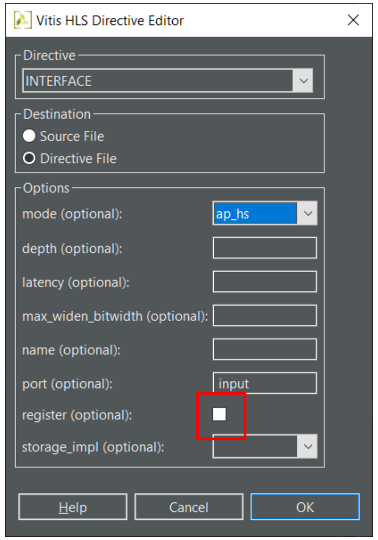
- The other wire protocols are derivatives of ap_hs in behavior
  - Some subtleties are worth discussing when the full two-way handshake is **not** used
- Using the Valid protocols (ap_vld, ap_ovld)
  - Outputs: Without an associated input acknowledge it is a requirement that the consumer takes the data when the output valid is asserted
    - Protocol ap_ovld only applies to output ports (is ignored on inputs)
  - Inputs: No particular issue
    - Without an associated output acknowledge, the producer does not know when the data has been read (but the done signal can be used to update values)
- Using the Acknowledge Protocol (ap_ack)
  - Outputs: Without an associated output valid the consumer will not know when valid data is ready but the design will stall until it receives an acknowledge (**Dangerous**: Lock up potential)
  - Inputs: Without an associated input valid, the design will simply read when it is ready and acknowledge that fact.

Registering IO Reads and Writes

- Vitis HLS does not register Input and Outputs by default
  - It will chain operations to minimize latency
  - The inputs will be read when the design requires them
  - The outputs will be written as soon as they are available



- Inputs and outputs can be registered
  - Inputs will be registered in the first cycle
    - Input pointers and partitioned arrays will be registered when they are required
  - Outputs will be registered and held until the next write operation
    - Which for scalars will be the next transaction (can't write twice to the same port) unless the block is pipelined


Vitis HLS Directive Editor

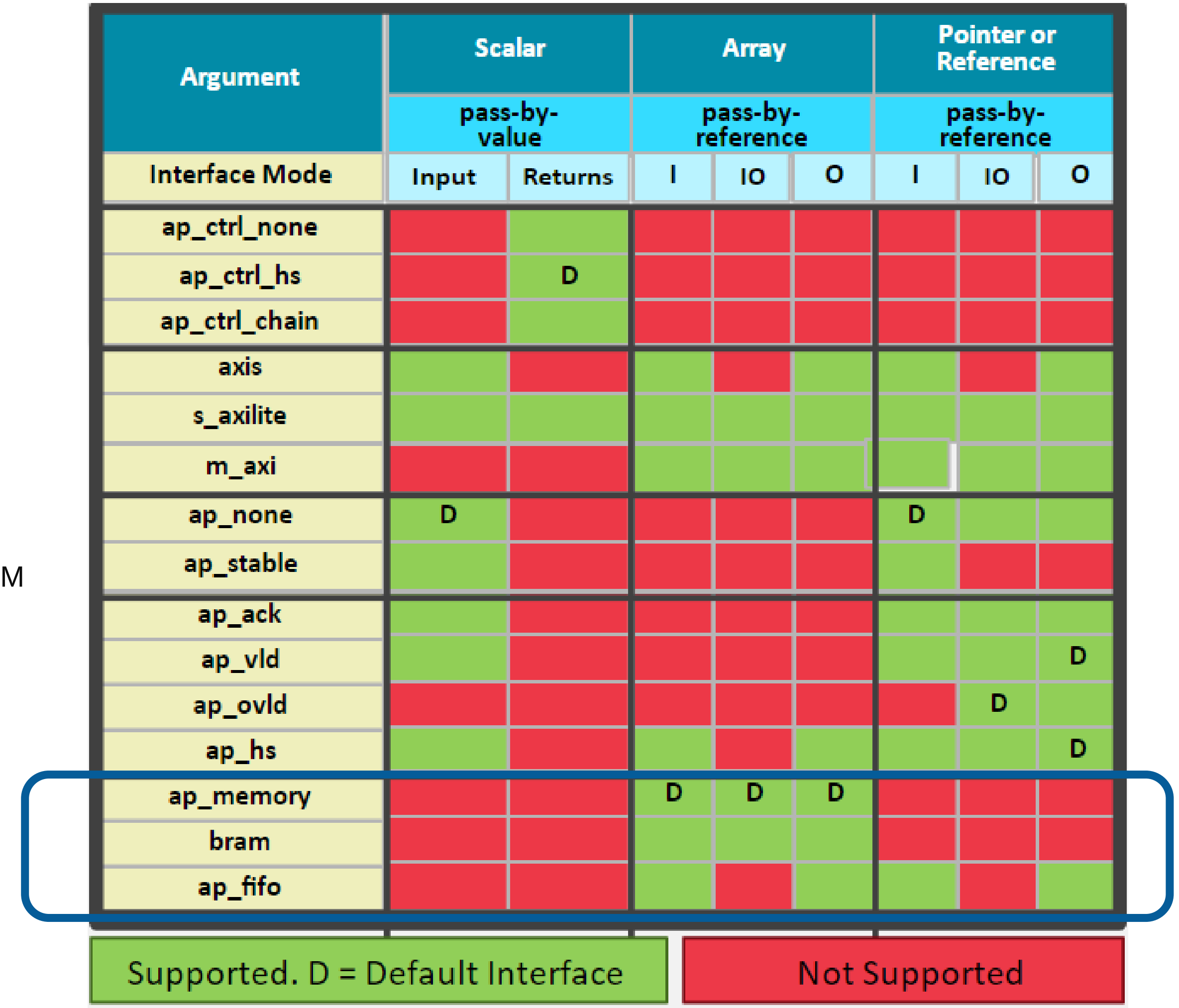
Directive
INTERFACE

Destination
○ Source File
● Directive File

Options
mode (optional): ap_hs
depth (optional):
latency (optional):
max_widen_bitwidth (optional):
name (optional):
port (optional): input
register (optional): □
storage_impl (optional):

Memory IO Protocols

- Memory protocols can be inferred from array and pointer arguments
  - Array arguments can be synthesized to RAM or FIFO ports
    - When FIFOs specified on array ports, the ports must be read-only or write-only
  - Pointer (and References in C++) can be synthesized to FIFO ports
- RAM ports
  - Support arbitrary/random accesses
  - Specify the array targets using the BIND_STORAGE pragma for ap_memory interface
    - If no target is specified, Vitis HLS determines whether to use a single or dual-port RAM interface
  - Requires two cycles read access: generate address, read data
    - Pipelining can reduce this overhead by overlapping generation with reading
- FIFO ports
  - Require read and writes to be sequential/streaming
  - Always uses a standard FIFO (single-port) model
  - Single cycle for both reads and writes



Argument	Scalar		Array			Pointer or Reference		
	pass-by-value		pass-by-reference			pass-by-reference		
Interface Mode	Input	Returns	I	IO	O	I	IO	O
ap_ctrl_none	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥
ap_ctrl_hs	🟥	🟩 D	🟥	🟥	🟥	🟥	🟥	🟥
ap_ctrl_chain	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥
axis	🟩	🟥	🟩	🟥	🟩	🟩	🟥	🟩
s_axilite	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
m_axi	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩
ap_none	🟩 D	🟥	🟥	🟥	🟥	🟩 D	🟩	🟩
ap_stable	🟩	🟥	🟥	🟥	🟥	🟩	🟥	🟥
ap_ack	🟩	🟥	🟥	🟥	🟥	🟩	🟩	🟩
ap_vld	🟩	🟥	🟥	🟥	🟥	🟩	🟩	🟩 D
ap_ovld	🟥	🟥	🟥	🟥	🟥	🟥	🟩 D	🟩
ap_hs	🟩	🟥	🟩	🟥	🟩	🟩	🟩	🟩 D
ap_memory	🟥	🟥	🟩 D	🟩 D	🟩 D	🟥	🟥	🟥
bram	🟥	🟥	🟩	🟩	🟩	🟥	🟥	🟥
ap_fifo	🟥	🟥	🟩	🟥	🟩	🟩	🟥	🟩

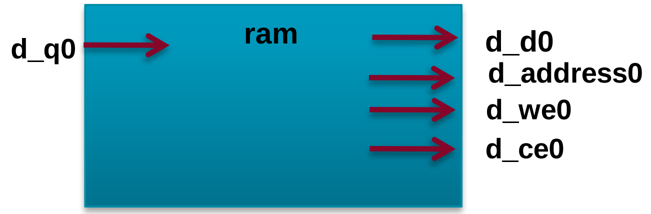
🟩 Supported. D = Default Interface

🟥 Not Supported

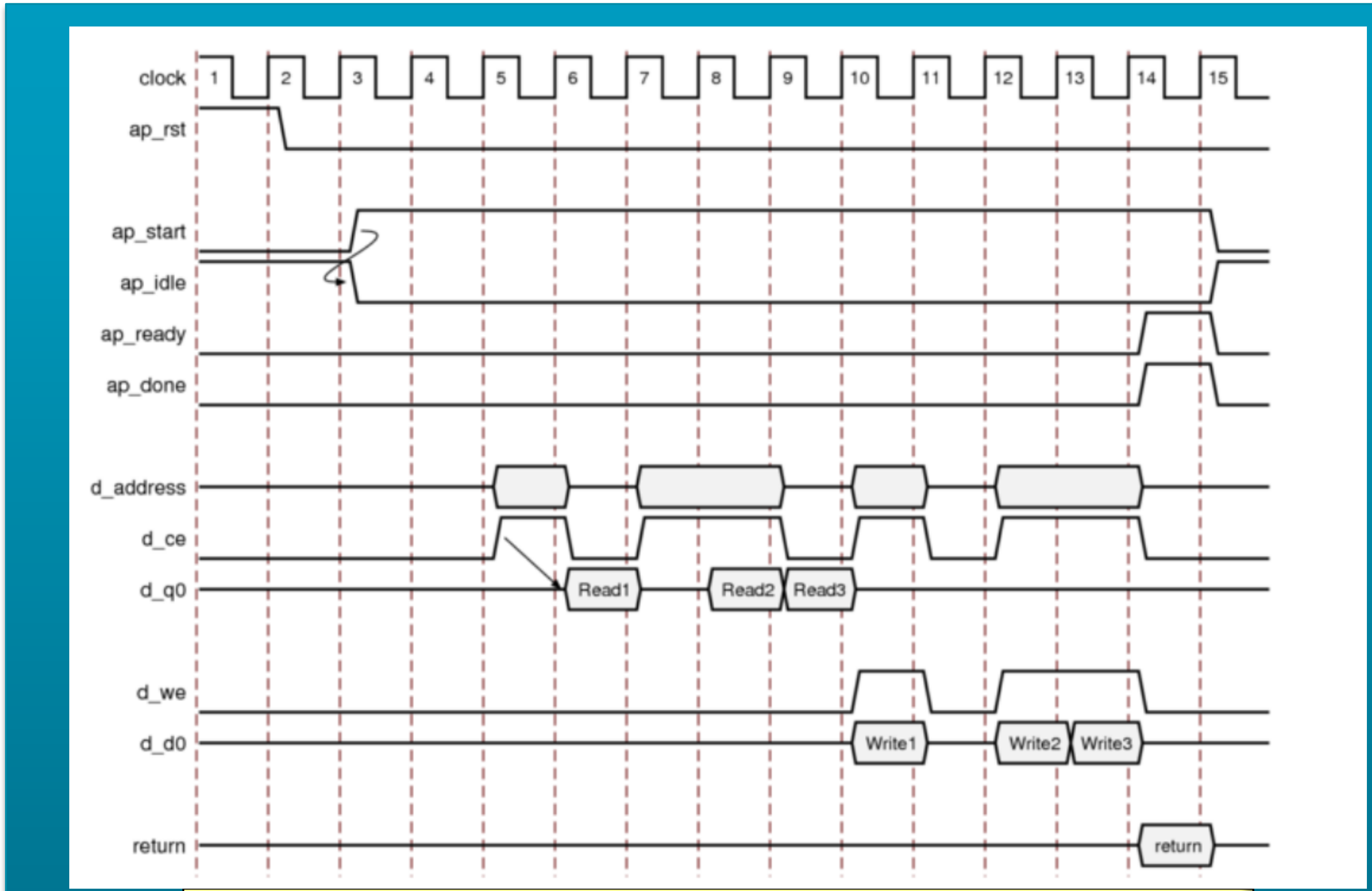
Memory IO Protocols: Ap_Memory

- 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
    - Vitis will automatically implement a dual-port interface to increase the bandwidth if it will reduce the II(initiation interval)
  - Using BIND_STORAGE pragma to specify the array targets
    - If not, Vitis HLS determines whether to use a single or dual port RAM interface

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



Ap_Memory Interface Timing Diagram



There is no stall behavior initiated by a RAM interface

- *After start, idle goes low and the RTL operates*

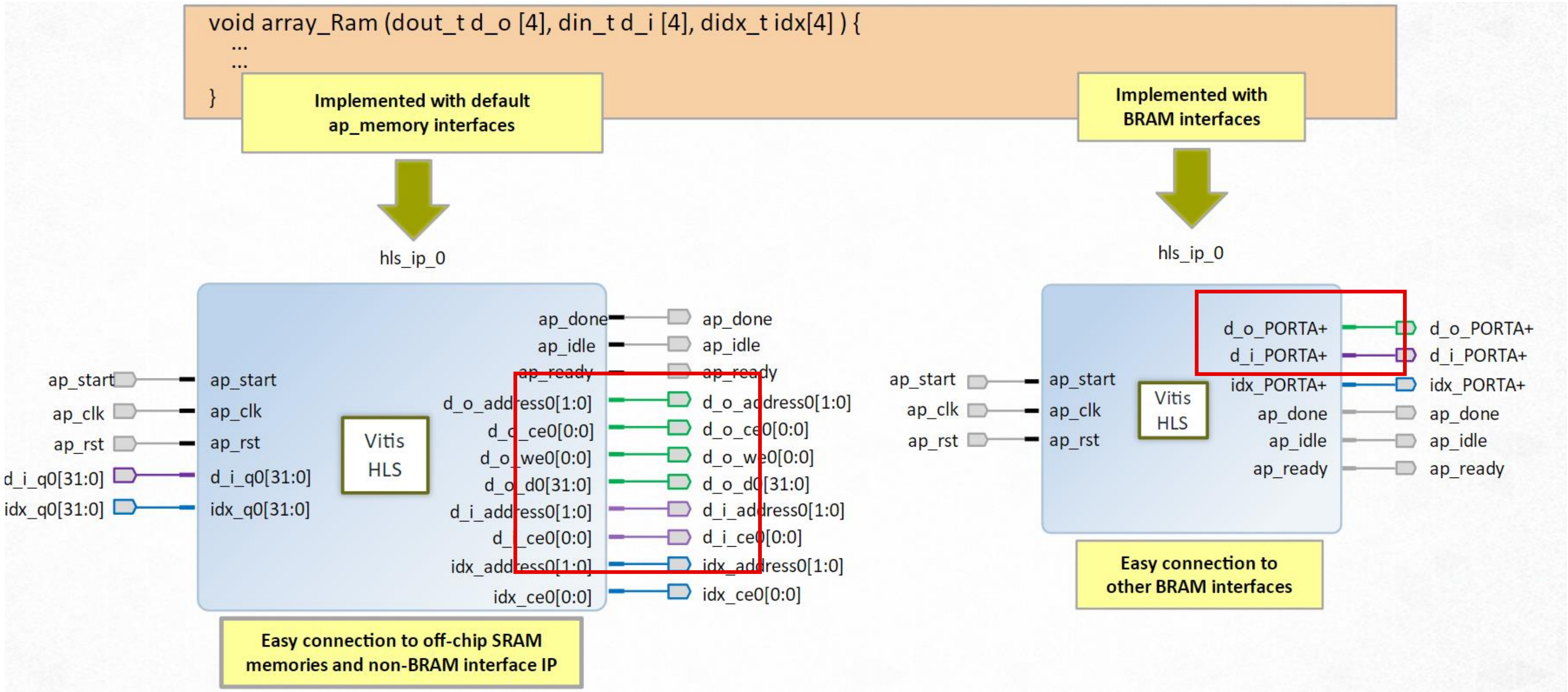
- When a read is required, an address is generated and CE asserted high
- Data is available on data input port d_q0 in the next cycle
- The read operations may be pipelined

- 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*

Memory IO Protocol : Block RAM(bram)

- Block RAM interface is functionally identical to the ap_memory interface
- The only difference is how the ports are implemented when the design is used in the IPI
  - An ap_memory interface is displayed as multiple and separate ports that can be connected to off-chip SRAM and non-block RAM
  - A block RAM interface is displayed as a single grouped port that can be connected to a Xilinx block RAM using a single point-to-point connection

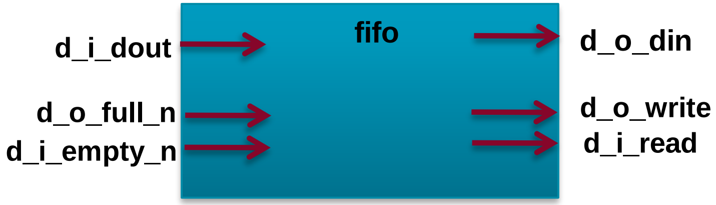


```
void array_Ram (dout_t d_o [4], din_t d_i [4], didx_t idx[4] ) {
   …
   …
}
```

Memory IO Protocols: AP_FIFO

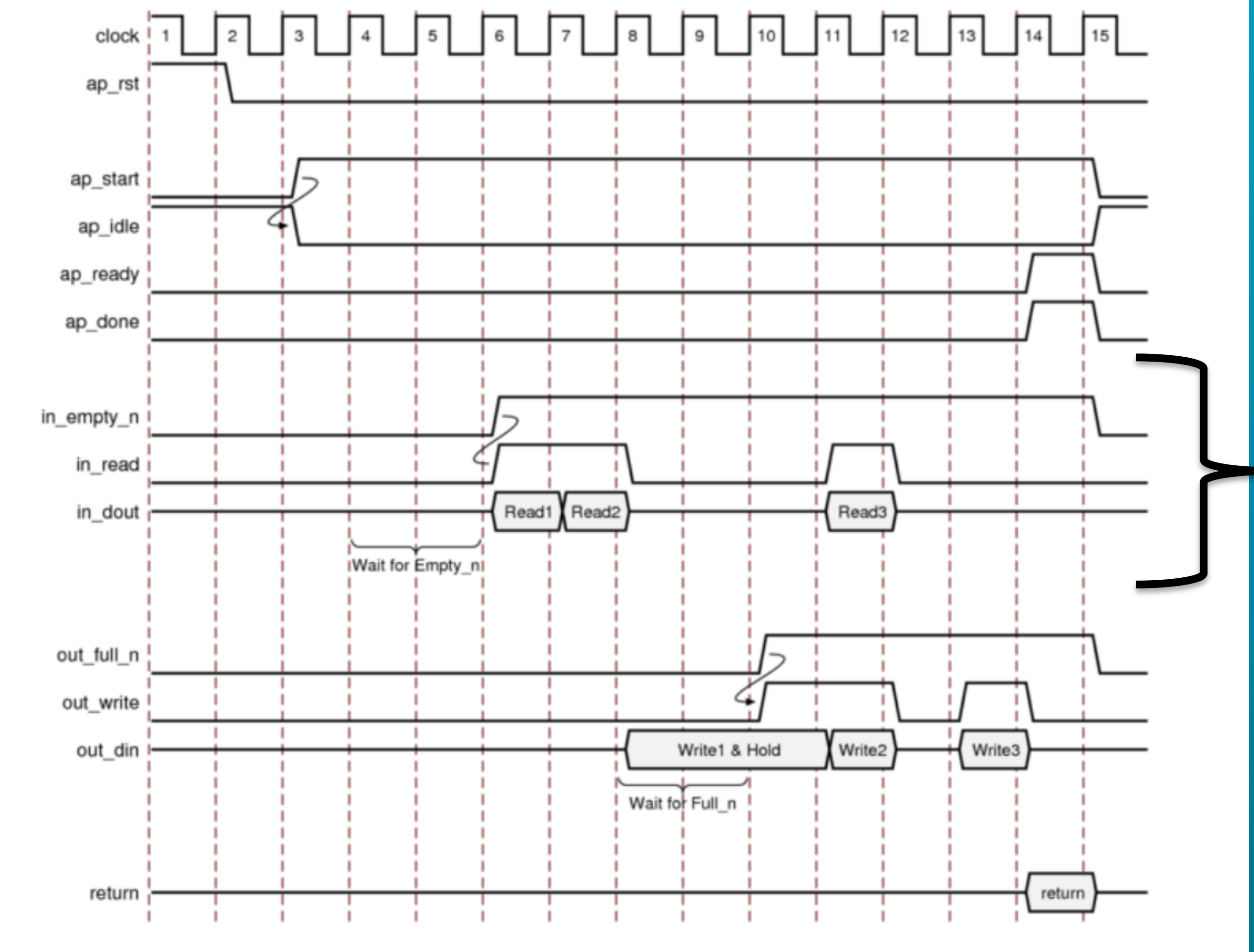
- When the output port is written to:
  - Output valid signal interface is hardware-efficient approach
  - When design requires access to memory and access performed in sequential manner
- FIFO Ports
  - Can be used on arrays, pointers or references are specified
  - Allows the port to be connected to a FIFO
  - Enables complete, two-way empty-full communication
  - Standard Read/Write, Full/Empty ports generated
    - Arrays: must use separate arrays for read and write
    - Pointers/References: split into In and Out ports

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

   ...
}
```



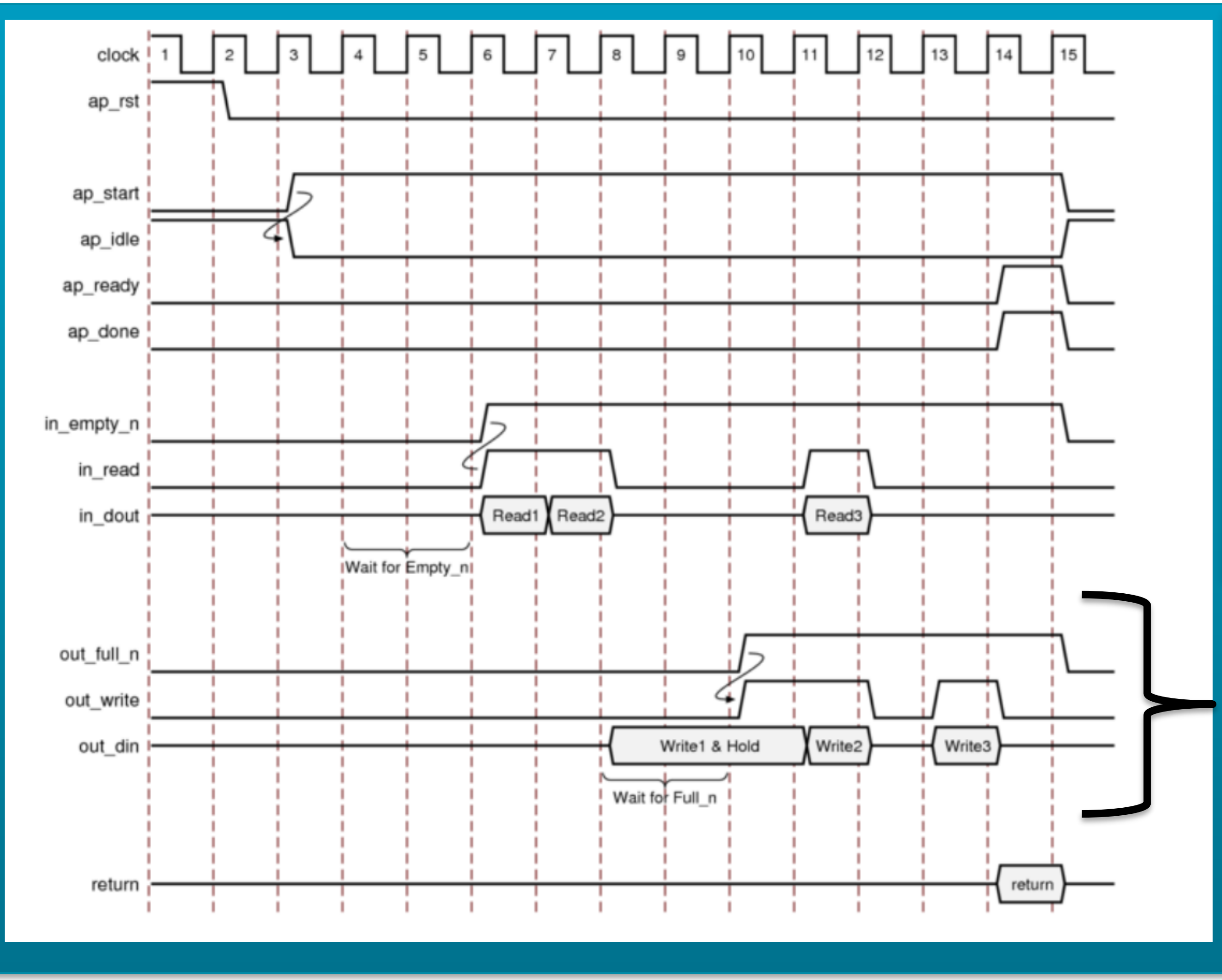
[Public]

AP_FIFO Read Interface Timing Diagram



- *After Start is applied, the block begins normal operation*
- If the input port is ready to be read but the FIFO is empty as indicated by input port in_empty_n Low, the design stalls and waits for data to become available
- When the FIFO contains data as indicated by input port in_empty_n High, an output acknowledges and the in_read is asserted High to indicate that the data was read in this cycle

AP_FIFO Write Interface Timing Diagram



- If an output is ready to be written to but the FIFO is full as indicated by out_full_n Low, the data is placed on the output port but the design stalls and waits for the space to become available in the FIFO

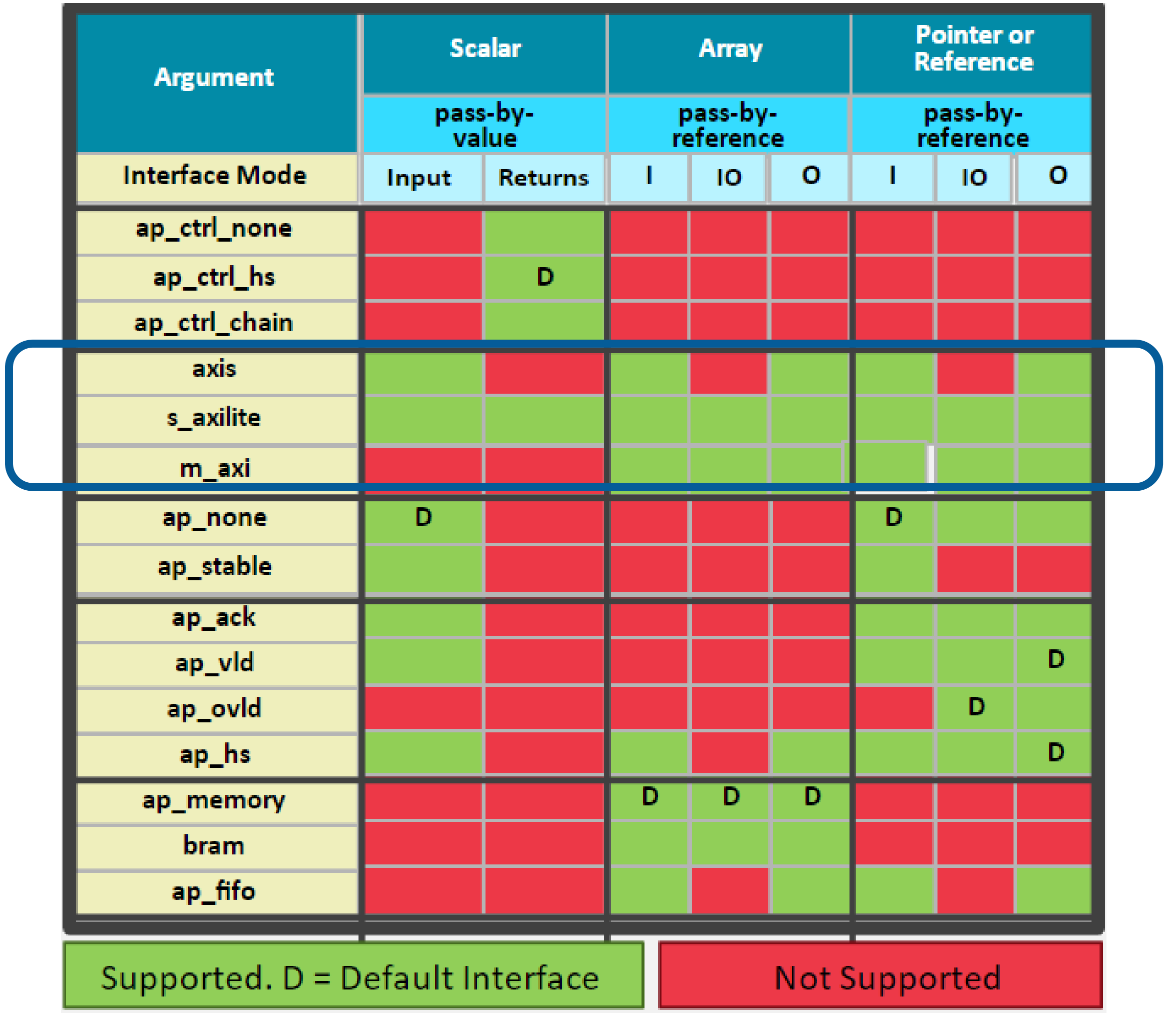
- When space becomes available in the FIFO as indicated by out_full_n High, the output acknowledges and the signal out_write is asserted to indicated that the output data is available

AXI Adapter Interface Protocols

AXI Adapter Interface Protocols

- The AXI4 interfaces supported by Vitis HLS include
  - the AXI4-Stream interface (axis)
  - AXI4-Lite (s_axilite)
  - AXI4 master (m_axi) interfaces
- The AXI4 adapter interfaces are the default interfaces
  - Used by Vitis HLS for the Vitis kernel flow
  - Though they are also supported in the Vivado IP flow.

AXI4 Interface
Part of ARM AMBA 4.0 interface
Widely Adopted in Xilinx Product Offerings
Provides Productivity, Flexibility, and Availability



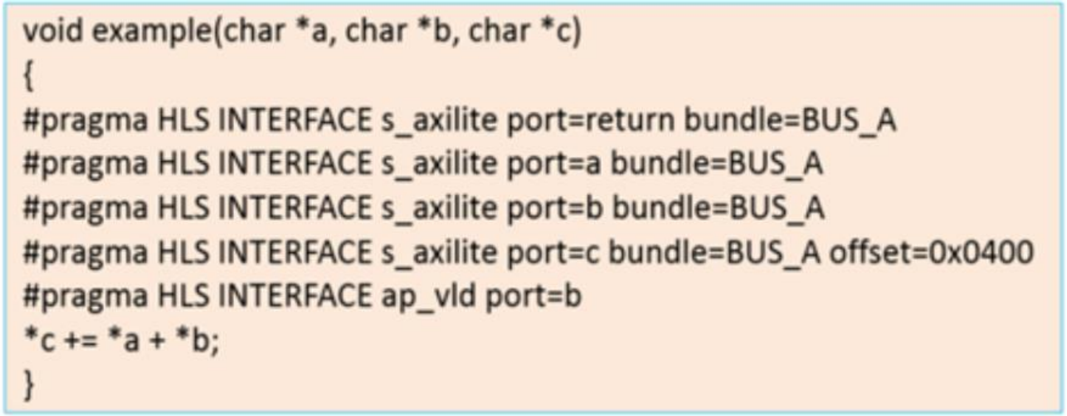
Argument	Scalar		Array			Pointer or Reference		
	pass-by-value		pass-by-reference			pass-by-reference		
Interface Mode	Input	Returns	I	IO	O	I	IO	O
ap_ctrl_none	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥
ap_ctrl_hs	🟥	🟩 D	🟥	🟥	🟥	🟥	🟥	🟥
ap_ctrl_chain	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥
axis	🟩	🟥	🟩	🟥	🟩	🟩	🟥	🟩
s_axilite	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
m_axi	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩
ap_none	🟩 D	🟥	🟥	🟥	🟥	🟩 D	🟩	🟩
ap_stable	🟩	🟥	🟥	🟥	🟥	🟩	🟥	🟥
ap_ack	🟩	🟥	🟥	🟥	🟥	🟩	🟩	🟩
ap_vld	🟩	🟥	🟥	🟥	🟥	🟩	🟩	🟩 D
ap_ovld	🟥	🟥	🟥	🟥	🟥	🟥	🟩 D	🟩
ap_hs	🟩	🟥	🟩	🟥	🟩	🟩	🟩	🟩 D
ap_memory	🟥	🟥	🟩 D	🟩 D	🟩 D	🟥	🟥	🟥
bram	🟥	🟥	🟩	🟩	🟩	🟥	🟥	🟥
ap_fifo	🟥	🟥	🟩	🟥	🟩	🟩	🟥	🟩

🟩 Supported. D = Default Interface

🟥 Not Supported

AXI4-Lite Interface

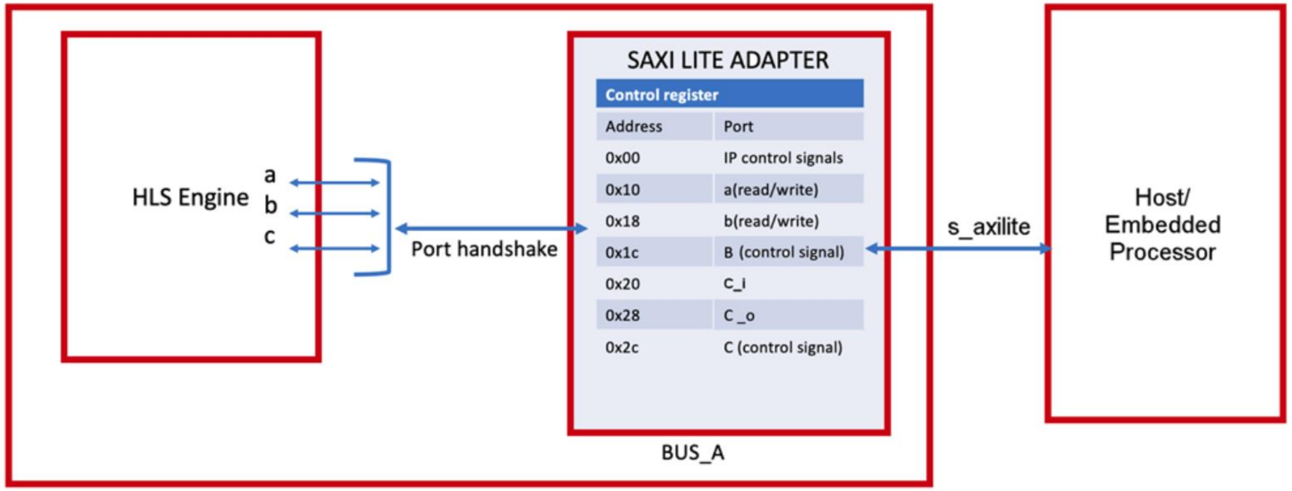
- S_axilite interfaces is implemented as an adapter that captures the data that was communicated from the host in registers on the adapter
- The AXI4-Lite interface performs several functions within a Vivado IP or Vitis kernel:
  - It maps a block-level control mechanism which can be used to start and stop the kernel.
  - It provides a channel for passing scalar arguments, pointers to scalar values, function return values, and address offsets for m_axi interfaces from the host to the IP or kernel
  - It output C driver files for use with the code running on a processor
- When the AXI4-slave lite interface is selected:
  - Default mode for input ports is ap_none
  - Default mode for output ports is ap_vld
  - Default mode for function return port is ap_ctrl_hs



```
void example(char *a, char *b, char *c)
{
#pragma HLS INTERFACE s_axilite port=return bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=a bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=b bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=c bundle=BUS_A offset=0x0400
#pragma HLS INTERFACE ap_vld port=b
*c += *a + *b;
}
```

S_AXILITE Example

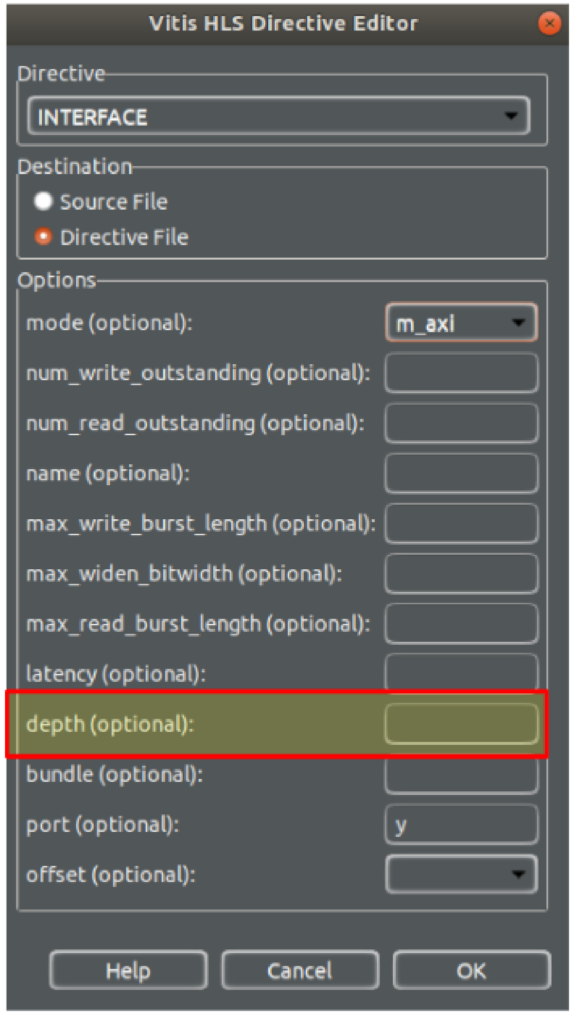
- S_axilite interfaces is implemented as an adapter that captures the data that was communicated from the host in registers on the adapter. The synthesized example will be part of a system that has three important elements as shown in the figure below:
  - Host application running on an x86 or embedded processor interacting with the IP or kernel
  - SAXI Lite Adapter: The INTERFACE pragma implements an s_axilite adapter. The adapter has two primary functions: implementing the interface protocol to communicate with the host, and providing a Control Register Map to the IP or kernel.
  - The HLS engine or function that implements the design logic
- By default, Vitis HLS automatically assigns the address for each port that is grouped into an s_axilite interface.



Control register	
Address	Port
0x00	IP control signals
0x10	a(read/write)
0x18	b(read/write)
0x1c	B (control signal)
0x20	C_I
0x28	C _o
0x2c	C (control signal)

AXI4 Master Interface

- AXI4 memory-mapped (m_axi) interface is used on array or pointer/reference arguments
  - allow kernels to read and write data in global memory (DDR, HBM, PLRAM).
  - Default to pointer and array arguments only in Vitis kernel flow
- Vitis HLS implements in one of the following modes:
  - Individual data transfers: reads or writes a single element of data for each address
  - Burst mode data transfers: reads or writes data using a single base address followed by multiple sequential data samples, which makes this mode capable of higher data throughput
- HLS provides an option to group the multiple ports into the same AXI_Master Interface:
  - All the ports which use the same bundle name are grouped
  - Depth option is given for the C/RTL co-simulation, which is required for pointers



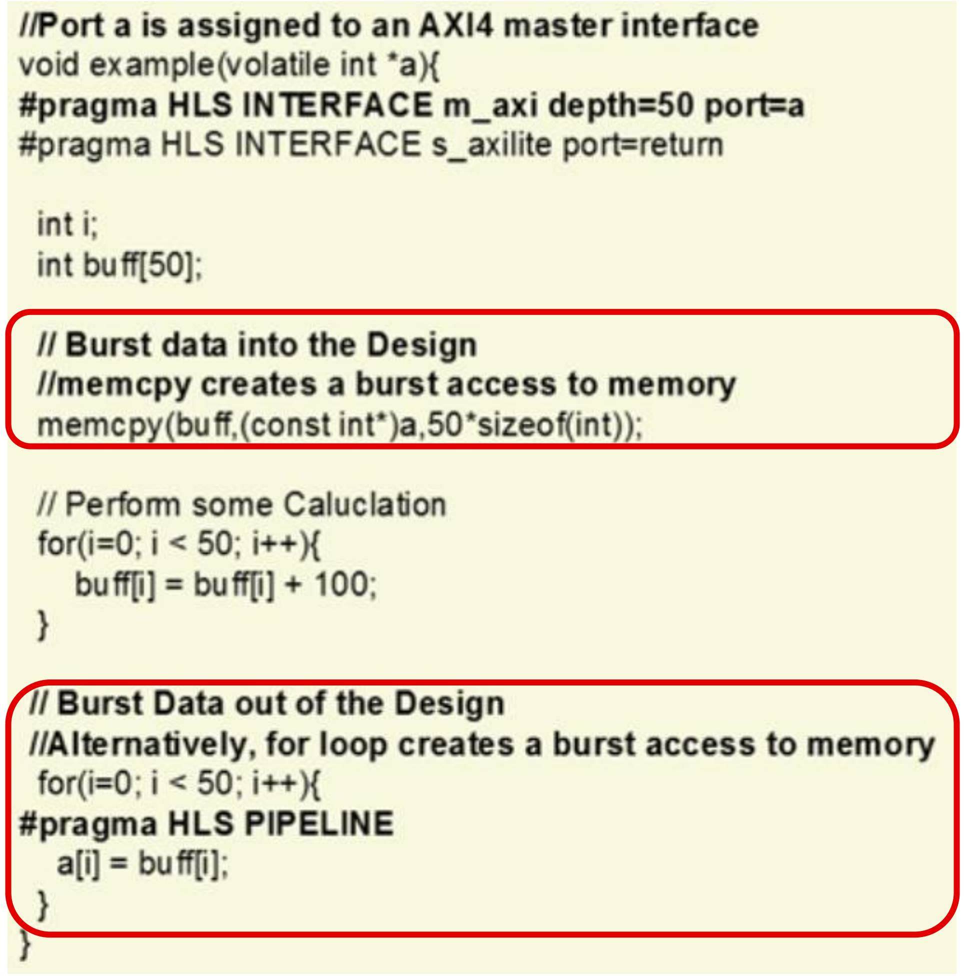
Directive: INTERFACE

Destination:
○ Source File
● Directive File

Options:
mode (optional): m_axi
num_write_outstanding (optional):
num_read_outstanding (optional):
name (optional):
max_write_burst_length (optional):
max_widen_bitwidth (optional):
max_read_burst_length (optional):
latency (optional):
depth (optional):
bundle (optional):
port (optional): y
offset (optional):

AXI4 Master: Brust Access with a Single Port

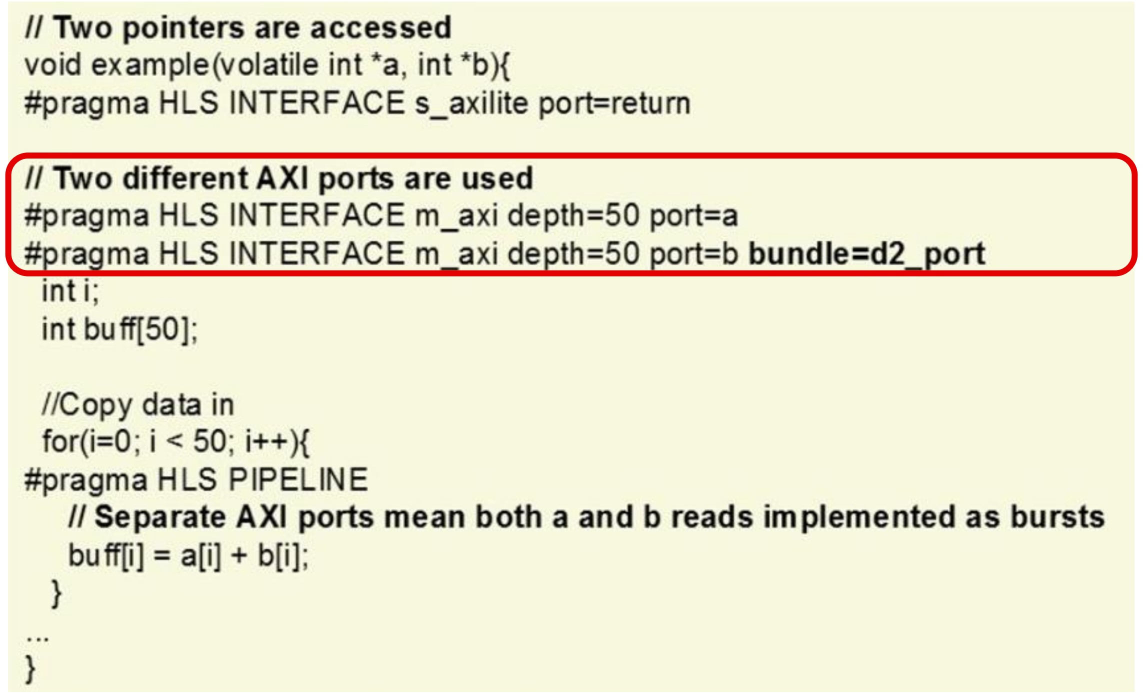
- Burst mode of operation is possible when you use
  - C memcpy function
  - a pipelined for loop
- Burst accesses are inferred from “for” loops automatically by the HLS tool
- Requirements:
  - Loops must be pipelined
  - Addresses must be accessed in increasing order
  - Memory accesses cannot be guarded by a conditional statement
  - Do not flatten nested loops
  - Only one read and write per AXI port allowed in a “for” loop
  - Only one read and one write is allowed in a “for” loop unless the ports are bundled in different AXI ports



```
//Port a is assigned to an AXI4 master interface
void example(volatile int *a){
#pragma HLS INTERFACE m_axi depth=50 port=a
#pragma HLS INTERFACE s_axilite port=return

 int i;
 int buff[50];

 // Burst data into the Design
 //memcpy creates a burst access to memory
 memcpy(buff,(const int*)a,50*sizeof(int));

 // Perform some Caluclation
 for(i=0; i < 50; i++){
    buff[i] = buff[i] + 100;
 }

 // Burst Data out of the Design
 //Alternatively, for loop creates a burst access to memory
 for(i=0; i < 50; i++){
#pragma HLS PIPELINE
   a[i] = buff[i];
 }
}
```

AXI4 Master: Brust Access with Multiple Port

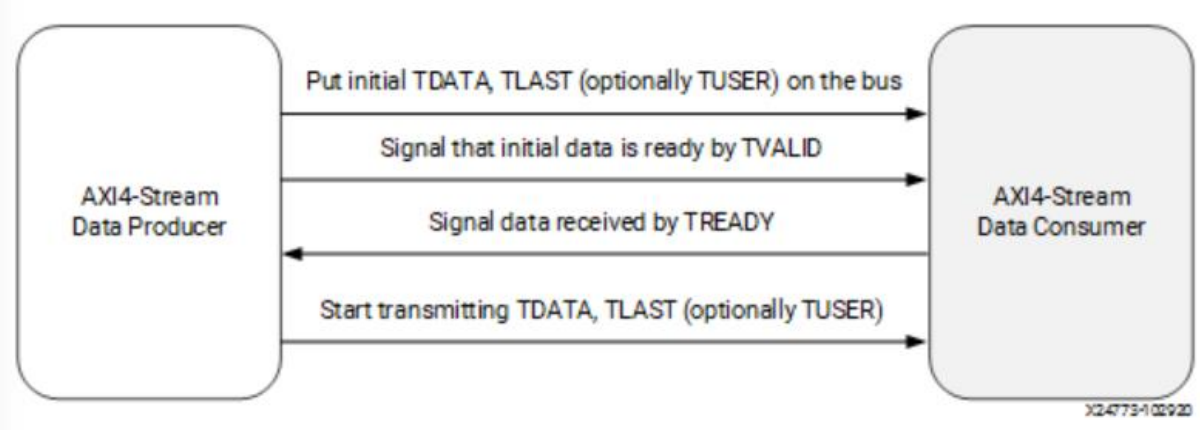
- Performing two reads in burst mode using different AXI interfaces
- Vitis HLS tool implements the port reads as brust transfers
  - Port a is specified without using the bundle option and is implemented in the default AXI interface
  - Port b is specified without using a named bundle option and is implemented in the separate AXI interface called d2_port
- In the multiple access bursts:
  - Only one access (read or write) per port can be inferred from a “for” loop
  - No simultaneous read and write access
  - If multiple ports are used, multiple read or writes can be performed



```
// Two pointers are accessed
void example(volatile int *a, int *b){
#pragma HLS INTERFACE s_axilite port=return

// Two different AXI ports are used
#pragma HLS INTERFACE m_axi depth=50 port=a
#pragma HLS INTERFACE m_axi depth=50 port=b bundle=d2_port
  int i;
  int buff[50];

  //Copy data in
  for(i=0; i < 50; i++){
#pragma HLS PIPELINE
    // Separate AXI ports mean both a and b reads implemented as bursts
    buff[i] = a[i] + b[i];
  }
...
}
```

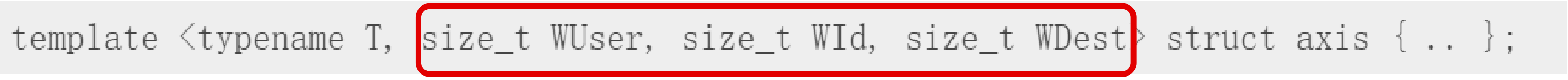
AXI4-Stream Interface

- AXIS interface can be applied to any input argument and any array or pointer output argument and cannot be used with arguments which are both read and write (inout arguments)
- AXIS interface is always sign-extended to the next byte
- Implemented are register interfaces or non-registered interfaces. This is controlled using the INTERFACE directive –register option
- The use of hls::axis (and ap_axiu/ap_axis)
  - limited to interfaces of the top-level function as it is the programmatic method to support AXI4-Stream with side channels.
  - cannot be used on internal functions or variables.
  - For internal streams you must use **hls::stream** objects.


AXI4-Stream Data Producer
AXI4-Stream Data Consumer
Put initial TDATA, TLAST (optionally TUSER) on the bus
Signal that initial data is ready by TVALID
Signal data received by TREADY
Start transmitting TDATA, TLAST (optionally TUSER)
X24773-102920

- AXI4-Stream is a protocol designed for transporting arbitrary unidirectional data.
  - TDATA width of bits is transferred per clock cycle.
  - The transfer is started once the producer sends the TVALID signal and the consumer responds by sending the TREADY signal (once it has consumed the initial TDATA).
  - At this point, the producer will start sending TDATA and TLAST (TUSER if needed to carry additional user-defined sideband data).
  - TLAST signals the last byte of the stream. So the consumer keeps consuming the incoming TDATA until TLAST is asserted.

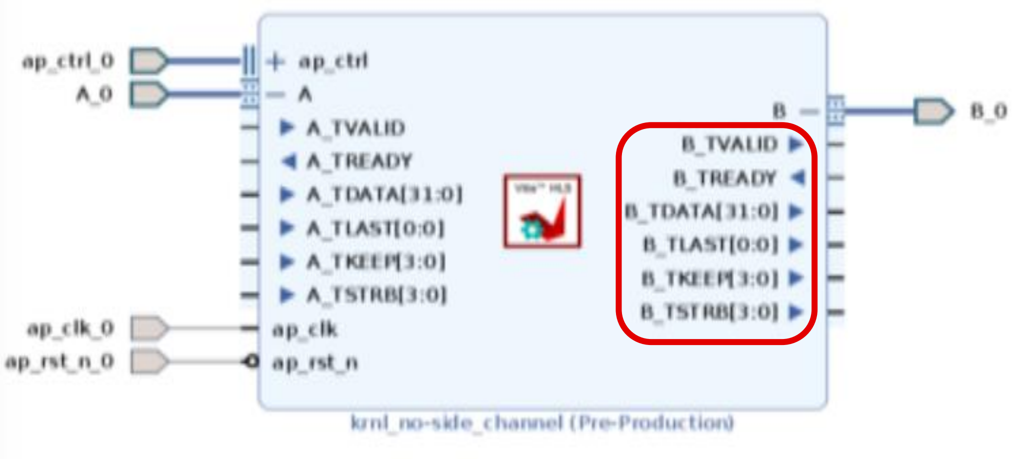
AXI4-Stream without Side channels



```
template <typename T, size_t WUser, size_t WId, size_t WDest> struct axis { .. };
```

- The AXI4-Stream interface is implemented as a struct type in Vitis HLS and has the above signature (defined in ap_axi_sdata.h)

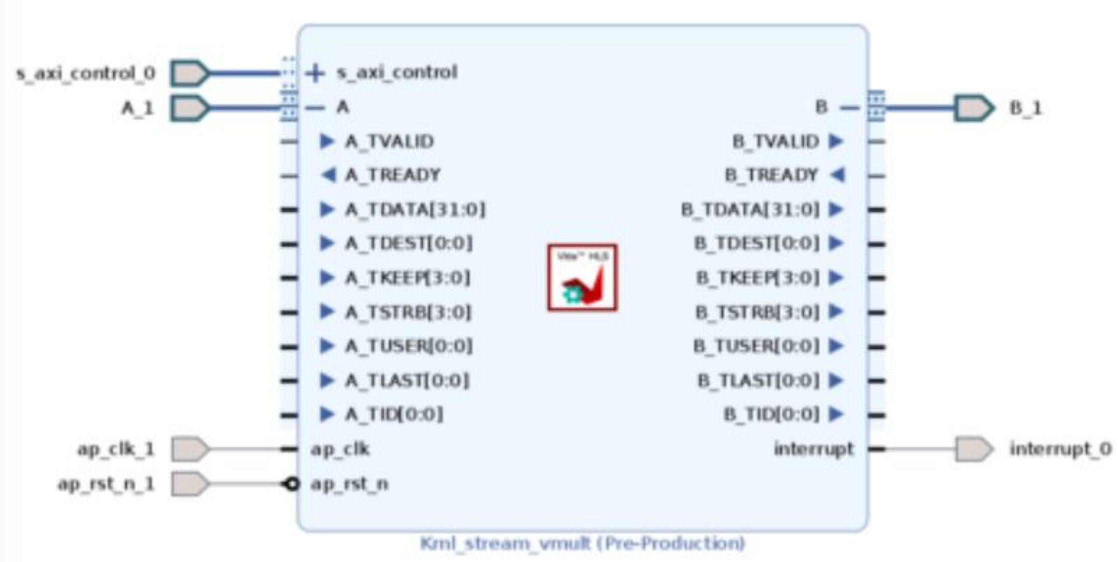
- An AXI4-Stream is used without side-channels when the function argument, ap_axis or ap_axiu data type, does not contain any AXI4 side-channel elements
  - In the right example, both interfaces are implemented using an AXI4-Stream
  - that is, when the WUser, WId, and WDest parameters are set to 0.
  - typedef ap_axiu<DWIDTH, 0, 0, 0> trans_pkt;



- After synthesis, both arguments are implemented with a data port (TDATA) and the standard AXI4-Stream protocol ports, TVALID, TREADY, TKEEP, TLAST, and TSTRB, as shown in the right figure.

AXI4-Stream with Side channels

- Optional but part of the AXI4-Stream Standard
- Directly referenced and controlled in the C/C++ code using a struct, provided the member elements of the struct match the names of the AXI4-Stream side-channel signals
- The right example shows how the side-channels can be used directly in the C/C++ code and implemented on the interface.
- The code uses #include "ap_axi_sdata.h" to provide an API to handle the side-channels of the AXI4-Stream interface. In the following example a signed 32-bit data type is used:
  - typedef ap_axiu<DWIDTH, 1, 1, 1> trans_pkt;
  - TID, TUSER,TDEST signals are added



Summary

Summary

- Vitis HLS has four types of IO
  - Data ports created by the original C function arguments
  - IO protocol signals added at the Block-Level
  - IO protocol signals added at the Port-Level
  - IO protocol signals added externally as AXI Adapter Interfaces
- Block Level protocols provide default handshake
  - Block Level handshakes are added to the RTL design
  - Enables system level control & sequencing
- Port Level Protocols provide wide support for all standard IO protocols
  - The protocol is dependent on the C variable type
- AXI Adapter Interface implement an adapter to manage communication according to the protocol

[Public]

AMD
XILINX

Thank You

Disclaimer and Attribution

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