

Block and Port Level Protocols



Objectives

- > **After completing this module, you will be able to:**
 - >> List the types of IO abstracted in Vivado HLS
 - >> List various basic and optional IO ports handled in Vivado 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*
- > Block Level Protocols
- > Port Level Protocols
- > Summary

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 101 : 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;
}
```

> Synthesized with 100 ns clock

>> All adders can fit in one clock cycle

>> **Combinational design**



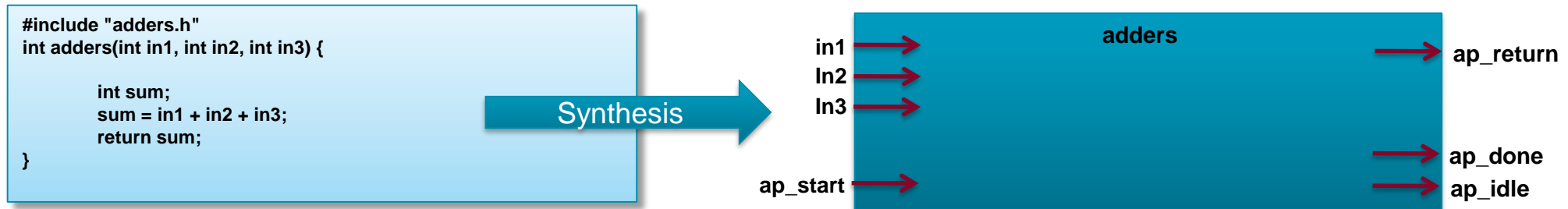
```
== Performance Estimates
+ Summary of timing analysis:
  * Estimated clock period (ns): 3.45
+ Summary of overall latency (clock cycles):
  * Best-case latency: 0
  * Average-case latency: 0
  * Worst-case latency: 0
```

>> The function return becomes RTL port ap_return

>> No handshakes are required or created in this example

Example 102: Sequential Design

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



- > **By Default ..**
 - >> Block level handshake signals are added to the design

Vivado HLS IO Options

> Vivado 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

Vivado HLS Basic Ports

> Adder Example

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

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

    return temp;
}
```

Synthesis

ap_clk
ap_rst
ap_ce

- **Clock added to all sequential RTL blocks**
 - One clock per function/block
 - SystemC designs may have a unique clock for each CTHREAD
- **Reset added to all sequential 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 sequential RTL (config_interface) configuration
 - When de-asserted it will cause the block to “freeze”
 - All connected blocks are assumed to be using the same CE
 - When the IO protocol of this block freezes, it is expected other blocks do the same
 - Else a valid output may be read multiple times

Vivado HLS Optional IO : Function Arguments

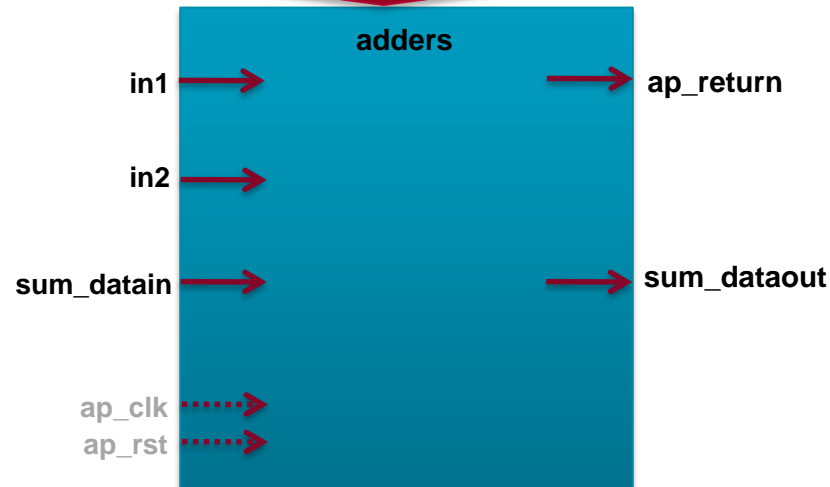
> Adder Example

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

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

    return temp;
}
```

Synthesis



- **Function Arguments**
 - Synthesized into data ports
- **Function Return**
 - Any return is synthesized into an output port called ap_return
- **Pointers (& 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

Vivado HLS Optional IO : Block Level Protocol

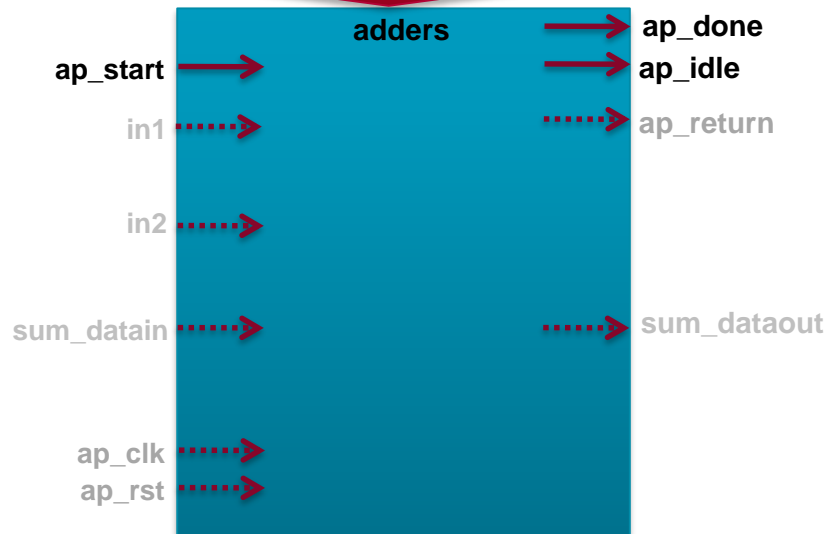
> Adder Example

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

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

    return temp;
}
```

Synthesis



- **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

Vivado HLS Optional IO : Port IO Protocols

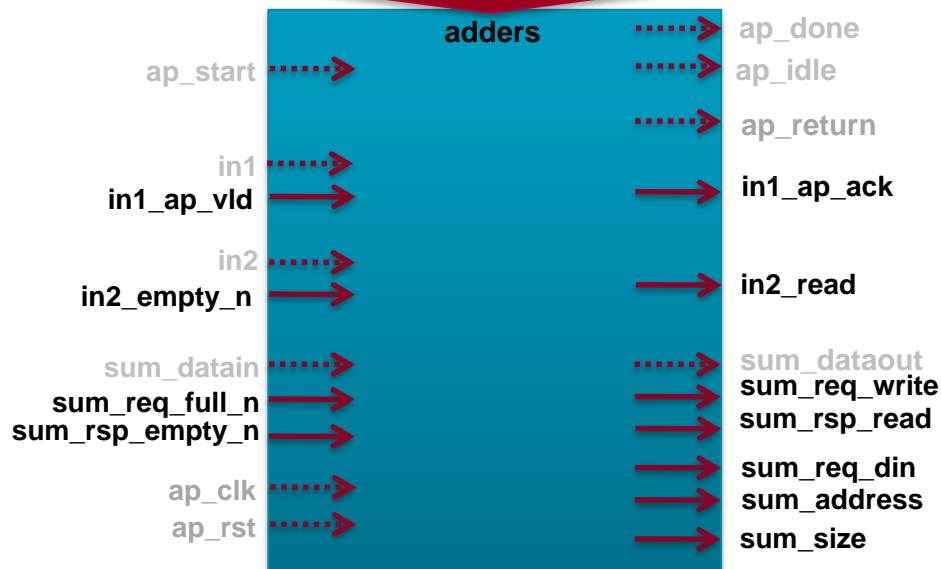
> Adder Example

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

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

    return temp;
}
```

Synthesis



• 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/C++ using an arbitrary protocol definition
- Supported natively in SystemC

Vivado HLS Interfaces Summary

> Where do you find the summary?

>> In the Synthesis report

The screenshot shows the Vivado HLS 2018.2 interface. The 'Explorer' pane on the left shows the project structure, with the 'report' directory expanded and 'yuv2rgb_csynth.rpt' highlighted. The main window displays the synthesis report for 'yuv2rgb_csynth.rpt'. The report is organized into sections: 'Ports' and 'Sizes and Direction'. The 'Interface Summary' section is highlighted in the 'Outline' pane on the right, indicating its location at the end of the report. A yellow box highlights the 'yuv2rgb_csynth.rpt' file in the Explorer pane, with a note stating: 'The same pathname as this from the project directory'. Another yellow box highlights the 'Interface Summary Section (at the end)' in the Outline pane. A third yellow box highlights the 'Ports' and 'Sizes and Direction' sections in the main report area.

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	yuv2rgb	return value
ap_rst	in	1	ap_ctrl_hs	yuv2rgb	return value
ap_start	in	1	ap_ctrl_hs	yuv2rgb	return value
ap_done	out	1	ap_ctrl_hs	yuv2rgb	return value
ap_continue	in	1	ap_ctrl_hs	yuv2rgb	return value
ap_idle	out	1	ap_ctrl_hs	yuv2rgb	return value
ap_ready	out	1	ap_ctrl_hs	yuv2rgb	return value
in_channels_ch1_dout	in	8	ap_fifo	in_channels_ch1	pointer
in_channels_ch1_empty_n	in	1	ap_fifo	in_channels_ch1	pointer
in_channels_ch1_read	out	1	ap_fifo	in_channels_ch1	pointer
in_channels_ch2_dout	in	8	ap_fifo	in_channels_ch2	pointer
in_channels_ch2_empty_n	in	1	ap_fifo	in_channels_ch2	pointer
in_channels_ch2_read	out	1	ap_fifo	in_channels_ch2	pointer
in_channels_ch3_dout	in	8	ap_fifo	in_channels_ch3	pointer
in_channels_ch3_empty_n	in	1	ap_fifo	in_channels_ch3	pointer
in_channels_ch3_read	out	1	ap_fifo	in_channels_ch3	pointer
in_width_dout	in	16	ap_fifo	in_width	pointer
in_width_empty_n	in	1	ap_fifo	in_width	pointer
in_width_read	out	1	ap_fifo	in_width	pointer
in_height_dout	in	16	ap_fifo	in_height	pointer
in_height_empty_n	in	1	ap_fifo	in_height	pointer
in_height_read	out	1	ap_fifo	in_height	pointer
out_channels_ch1_address0	out	22	ap_memory	out_channels_ch1	array
out_channels_ch1_ce0	out	1	ap_memory	out_channels_ch1	array
out_channels_ch1_we0	out	1	ap_memory	out_channels_ch1	array
out_channels_ch1_d0	out	8	ap_memory	out_channels_ch1	array

Vivado HLS Optional IO : IP Adapters

> Adder Example

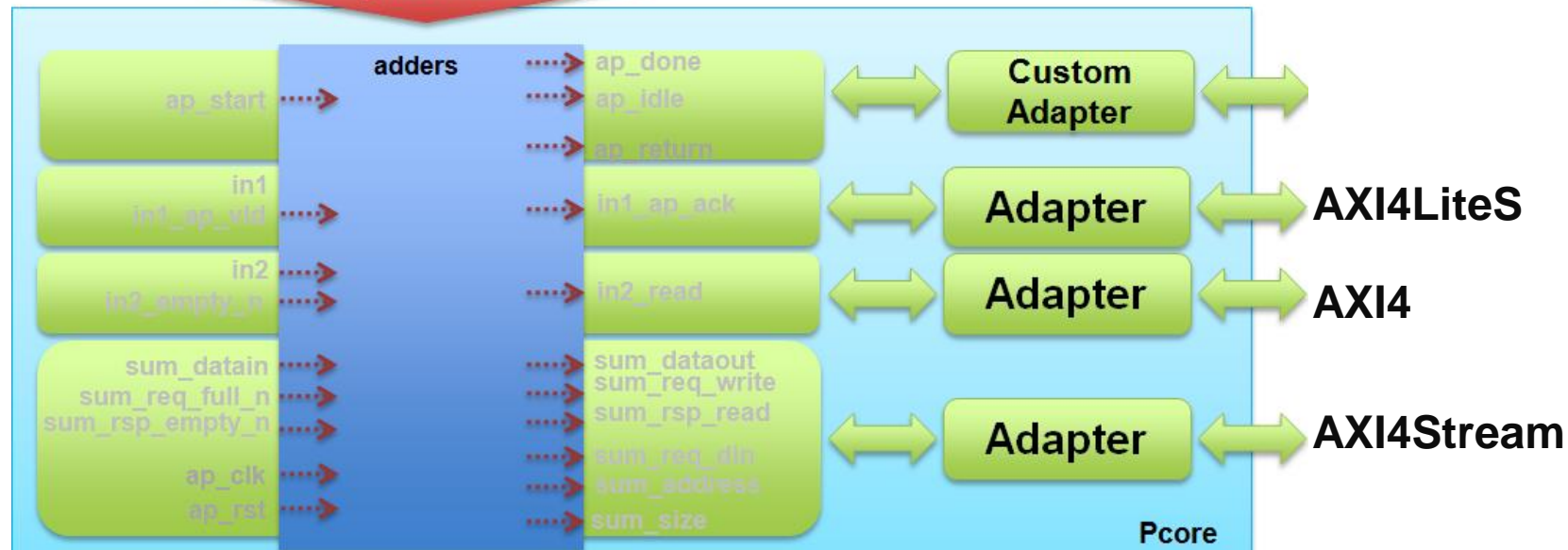
```
#include "adders.h"
int adders(int in1, int in2, int *sum) {

    int temp;

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

    return temp;
}
```

Synthesis

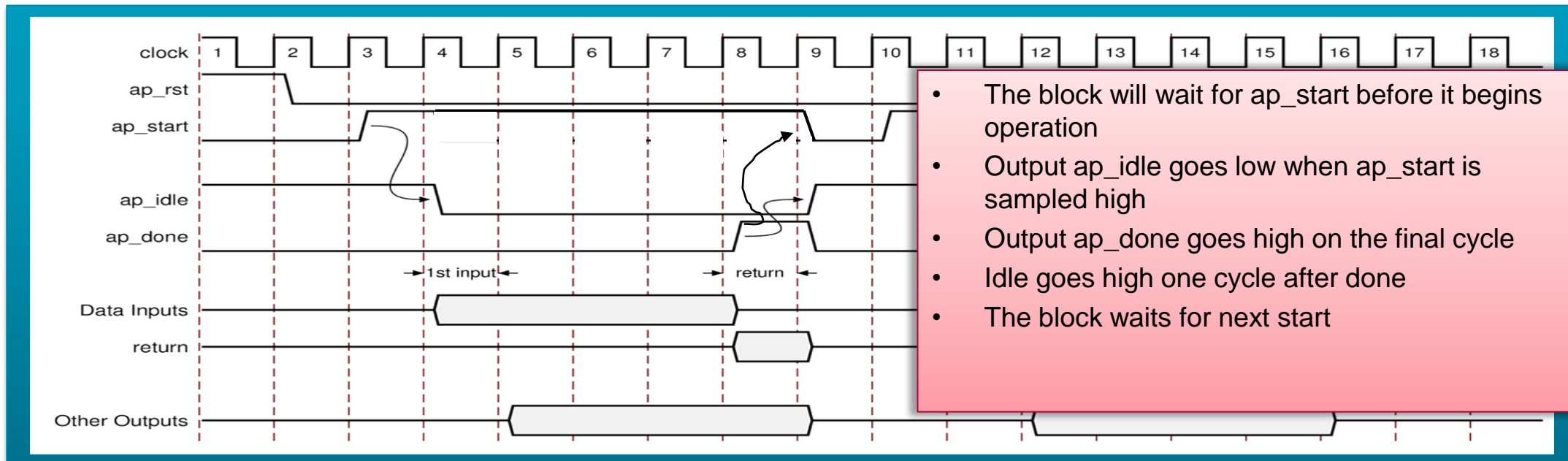


- Added when the IP block is exported
- Available in the IP directory
 - “ip”, “sysgen”

Block Level Protocols



AP_START: Single Cycle



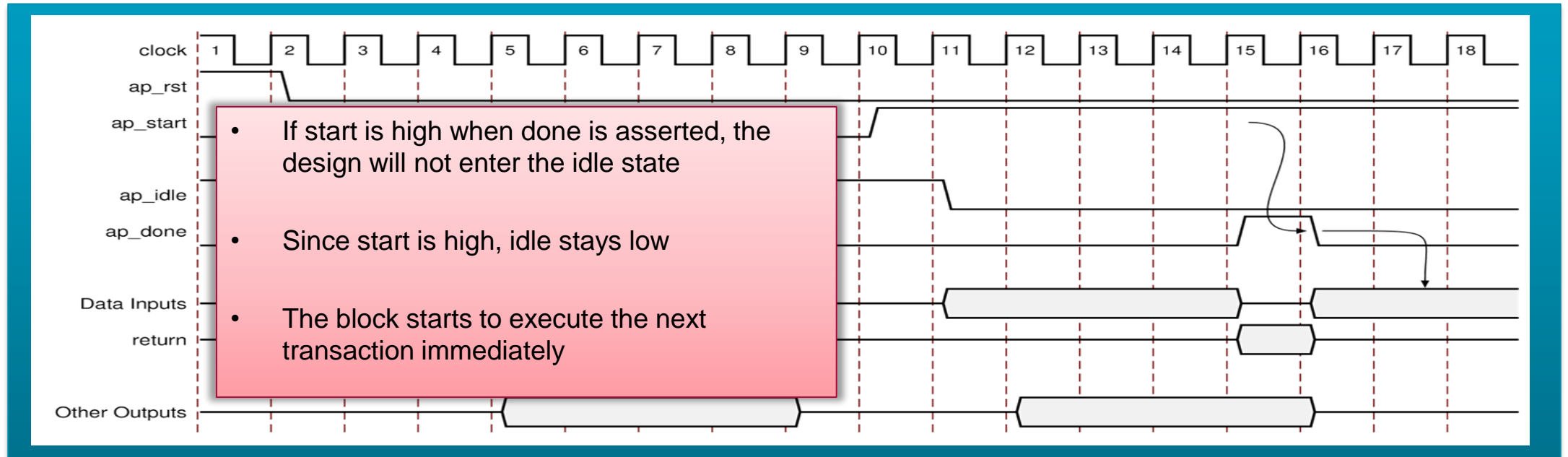
- **Input Data**

- 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**

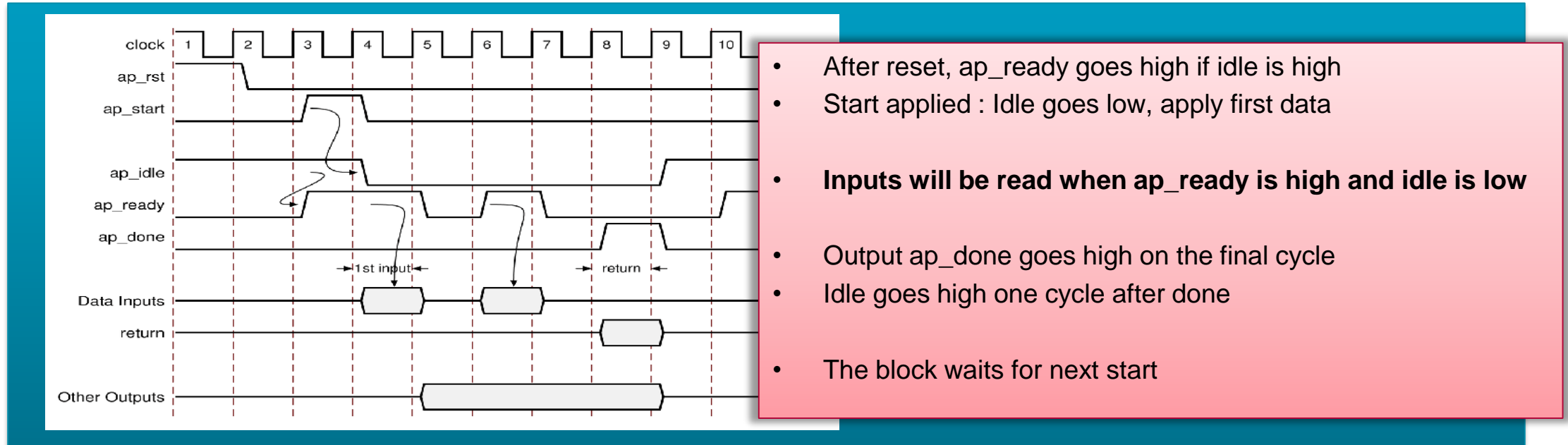
- Any function return is valid when ap_done is asserted high
- Other outputs may output their data at any time after the first read
 - The output can only be guaranteed to be valid with ap_done if it is registered
 - It is recommended to use a port level IO protocol for other outputs

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**

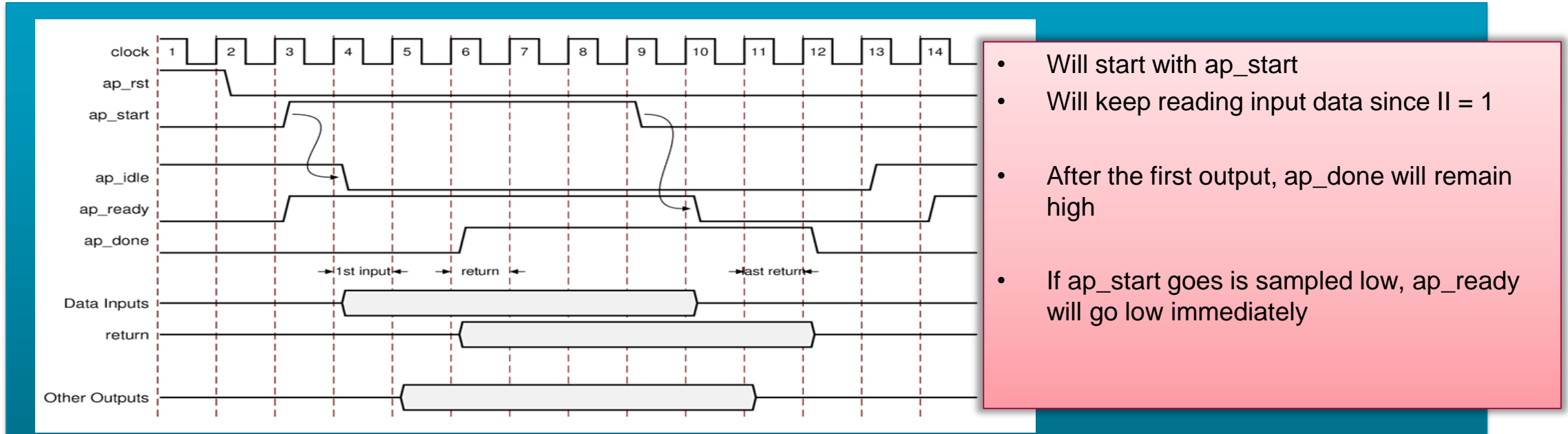
- Will be read when ap_ready is high and ap_idle is low
 - Indicates the design is ready for data
- Signal ap_ready will change at the rate of the II

- **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



- **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

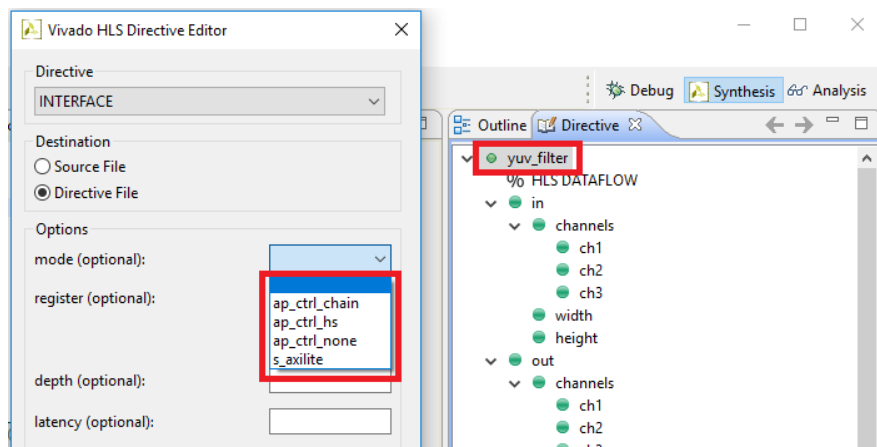
- **Output Data when II=1**

- 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)

Disabling Block Level Handshakes

> 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"
```

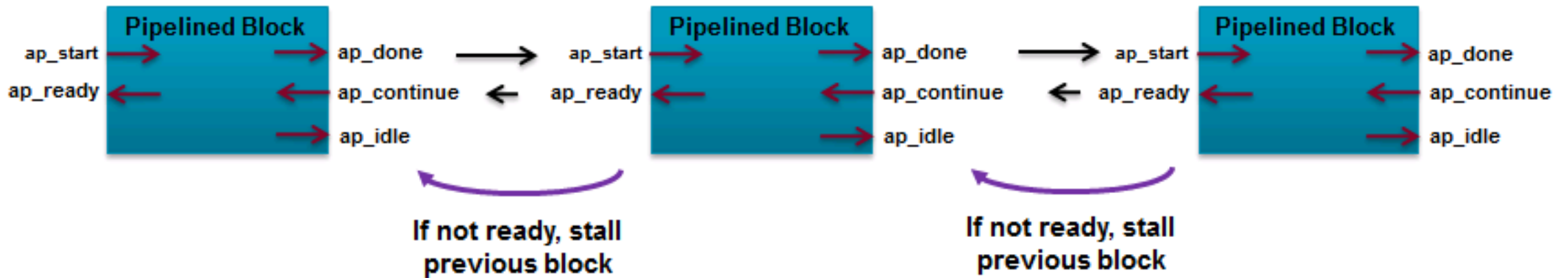
> Requirement: Manually verify the RTL

- >> Without block level handshakes autosim cannot verify the design
 - Will only work in simple combo and II=1 cases
 - Handshakes are required to know when to sample output signals

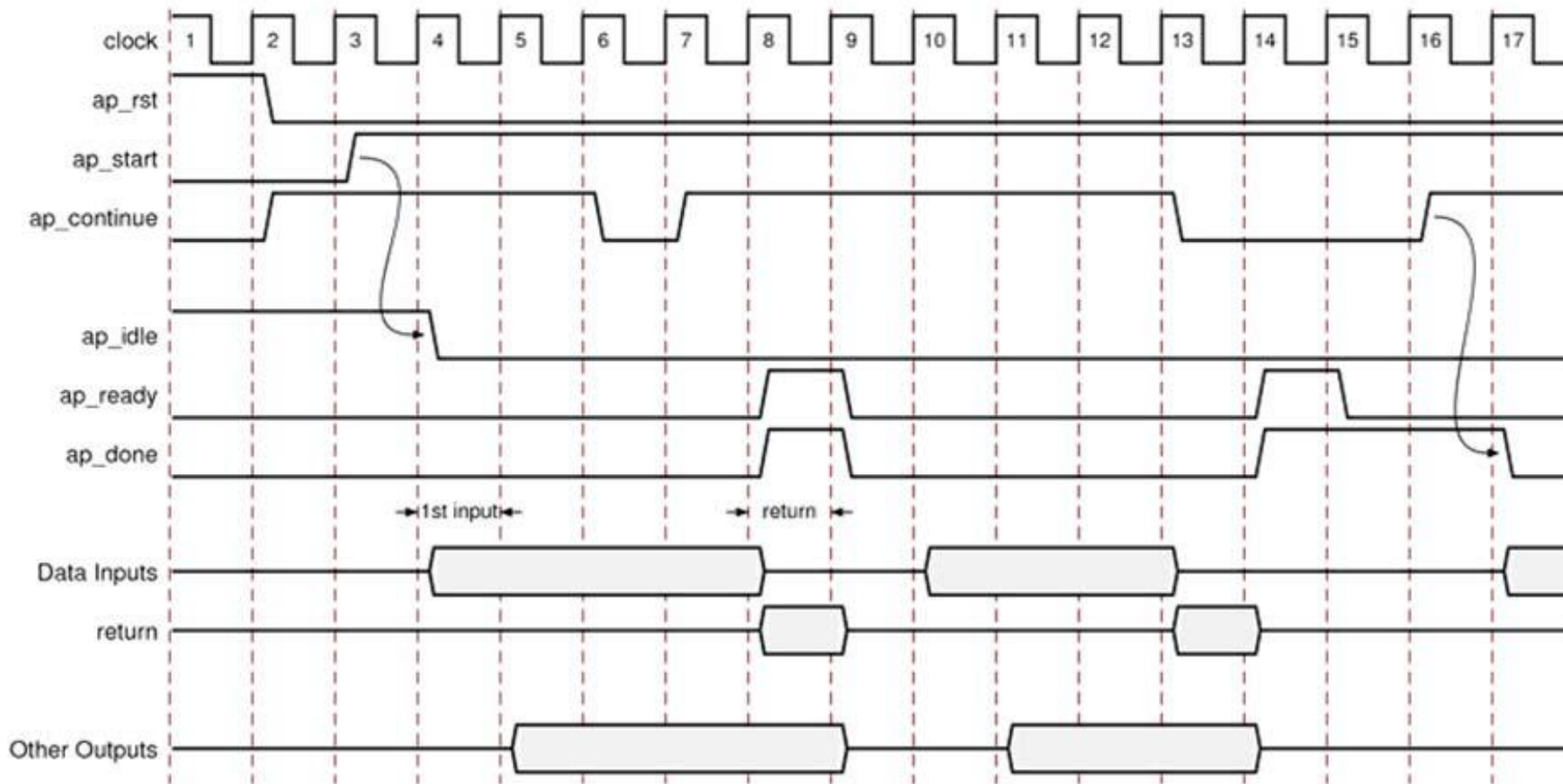
It is recommended to leave the default and use Block Level Handshakes

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



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
- When **ap_continue** is asserted Low, 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

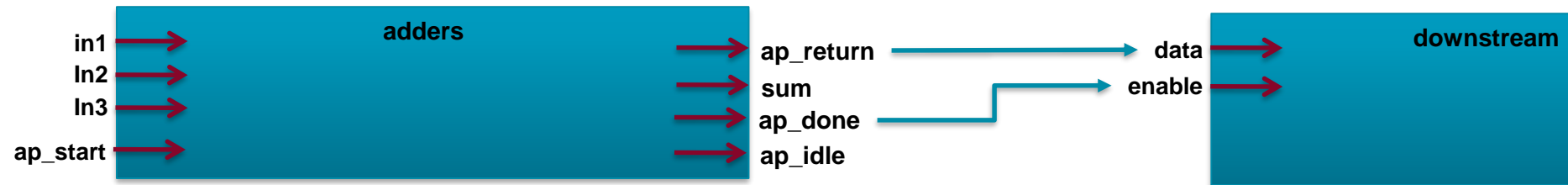
Port Level Protocols



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 (& references in C++)
 - Pass-by-reference arrays

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

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

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

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

Key:

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

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									
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									
ap_memory			D	D	D				
bram									
ap_fifo									D
ap_bus									

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

Block Level Protocol




AXI4 Interfaces

No IO Protocol

Wire handshake protocols

Memory protocols : RAM : FIFO

Bus protocols

 Supported
  D = Default Interface
  Not Supported

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

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									
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									
ap_memory			D	D	D				
bram									
ap_fifo									D
ap_bus									

Default IO Protocols

> The default port protocols

- >> Inputs: ap_none
- >> Outputs: ap_vld
- >> Inout: ap_ovld
 - In port gets ap_none
 - Out port gets ap_vld
- >> Arrays: ap_memory
- >> All shown as the default (D) on previous slide

The Vivado HLS shell/console always shows the results of interface synthesis

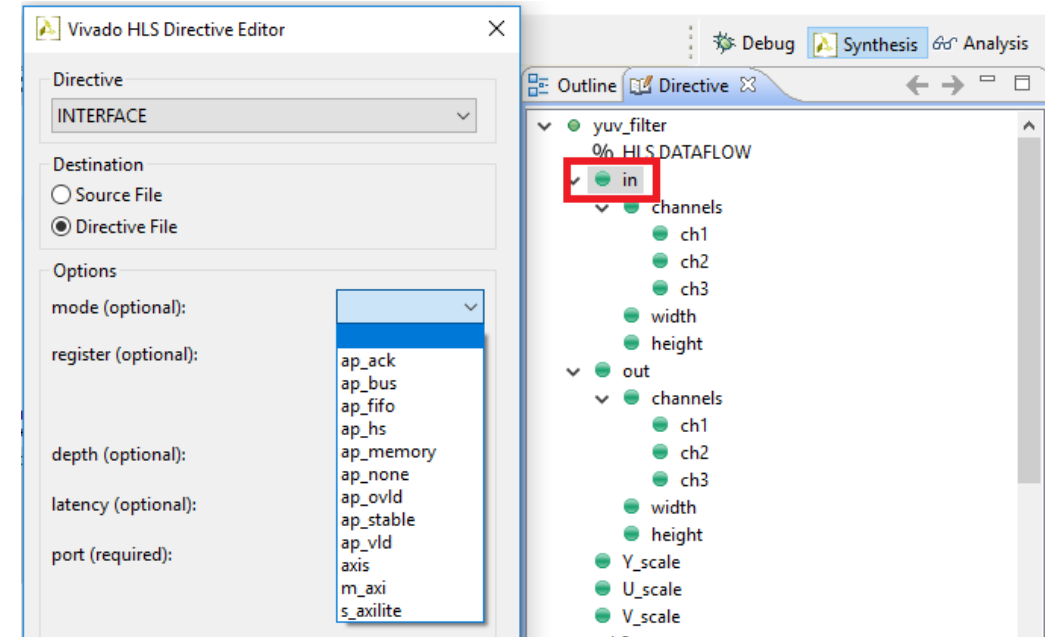
```
@I [RTGEN-500] Setting interface mode on port 'dct/input_r' to 'ap_memory'.  
@I [RTGEN-500] Setting interface mode on port 'dct/output_r' to 'ap_memory'.  
@I [RTGEN-100] Finished creating RTL model for 'dct'.
```

> Result of the default protocols

- >> No protocol for input ports
 - They should be held stable for the entire transaction
 - There is no way to know when the input will be read
- >> Output writes 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)

Specifying IO Protocols

- > **Select the port in the Directives pane to specify a protocol**
 - >> Select the port
 - >> Right-click and choose Interface
 - >> Select the protocol from the drop-down menu
- > **Or apply using Tcl or Pragma**



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

Block Level Protocol




AXI4 Interfaces

No IO Protocol

Wire handshake protocols

Memory protocols : RAM : FIFO

Bus protocols

 Supported
  D = Default Interface
  Not Supported

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

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									
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									
ap_memory			D	D	D				
bram									
ap_fifo									D
ap_bus									

No IO Protocol

- > **AP_NONE: The default protocol for 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)

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

Block Level Protocol

AXI4 Interfaces

No IO Protocol

Wire handshake protocols

Memory protocols : RAM : FIFO

Bus protocols

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									
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									
ap_memory			D	D	D				
bram									
ap_fifo									D
ap_bus									



Supported D = Default Interface



Not Supported

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

Wire 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)
 - >> **This block will stall** while waiting for the input controls

Wire 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

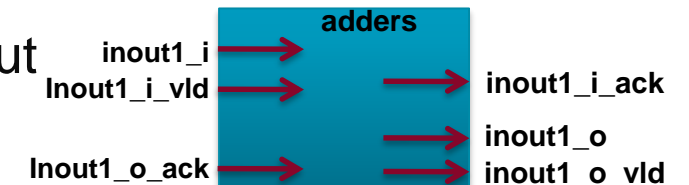
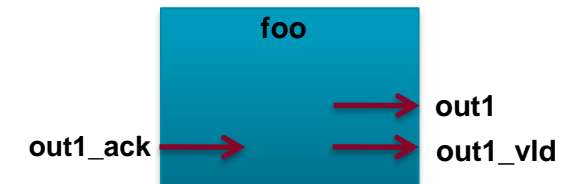
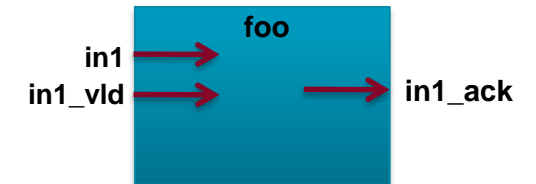
>> Outputs

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

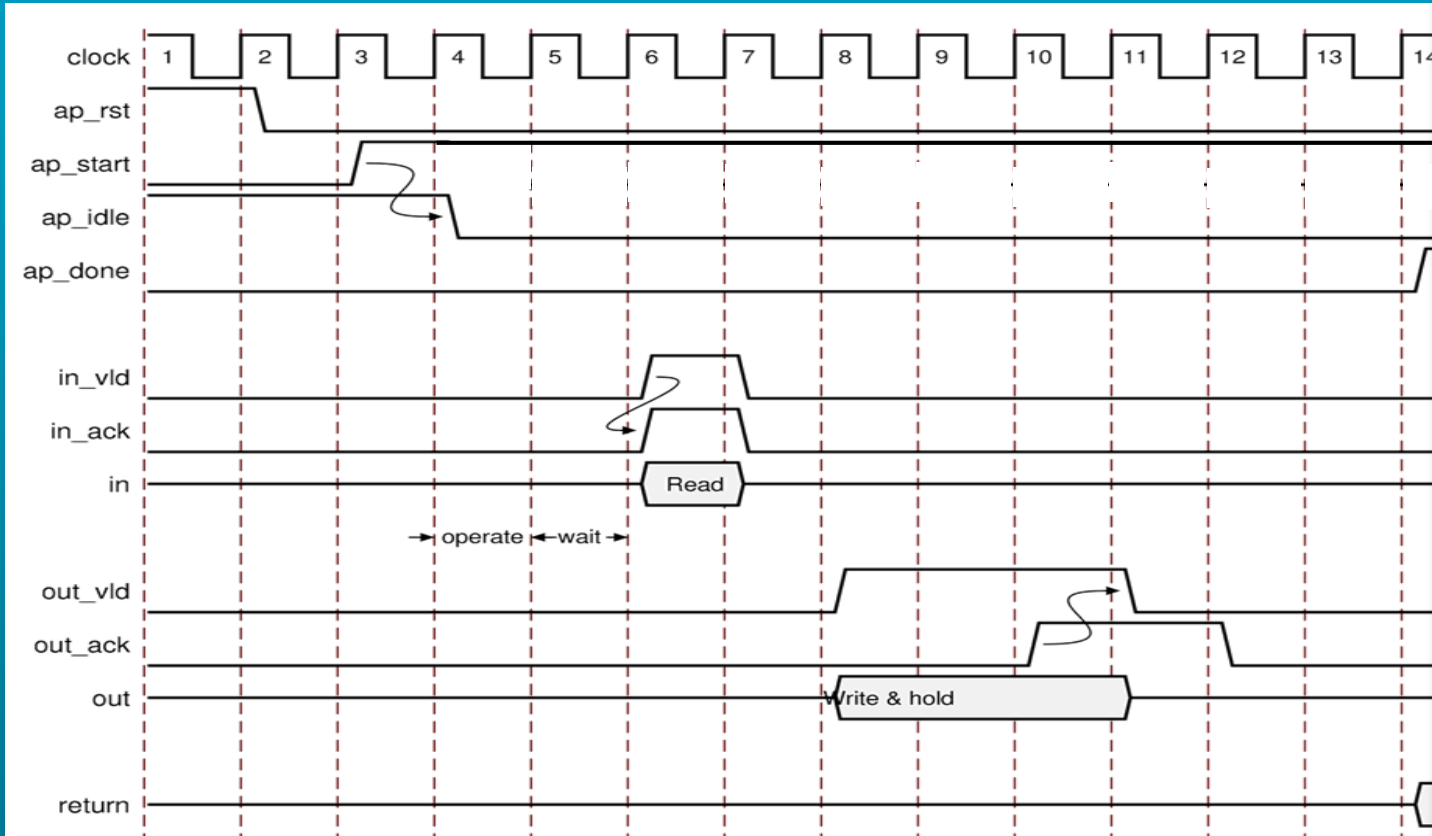
>> Inouts

- 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



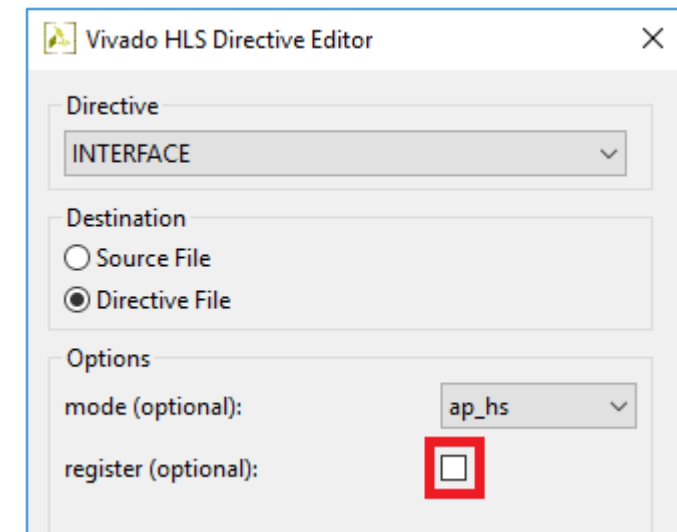
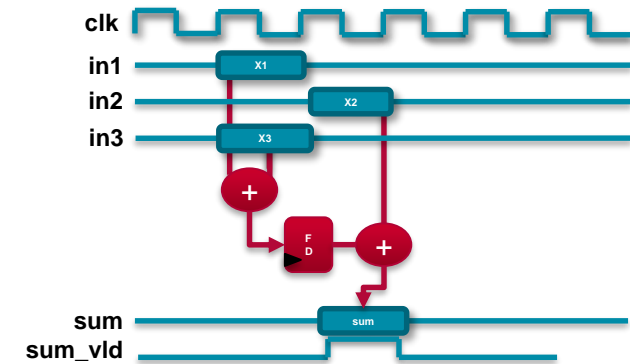
- After start, idle goes low and the RTL operates until a read is required
- The RTL will stall (wait) until the input valid is asserted
- If there is more than one input valid, each can stall the RTL
- It will acknowledge on the same cycle it reads the data (reads on next clock edge)
- An output valid 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

- > **Vivado 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



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

Block Level Protocol




AXI4 Interfaces

No IO Protocol

Wire handshake protocols

Memory protocols : RAM : FIFO

Bus protocols

 Supported
  D = Default Interface
  Not Supported

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

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									
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									
ap_memory			D	D	D				
bram									
ap_fifo									D
ap_bus									

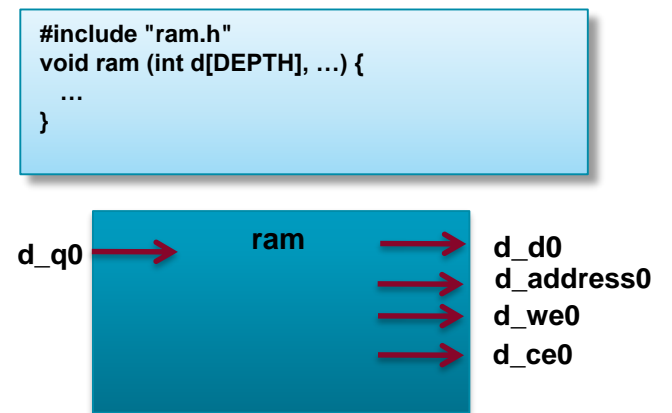
Memory IO Protocols

- > **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
 - >> May be implemented with Dual-Port RAMs to increase the bandwidth
 - The default is a 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

Memory IO Protocols: Ports Generated

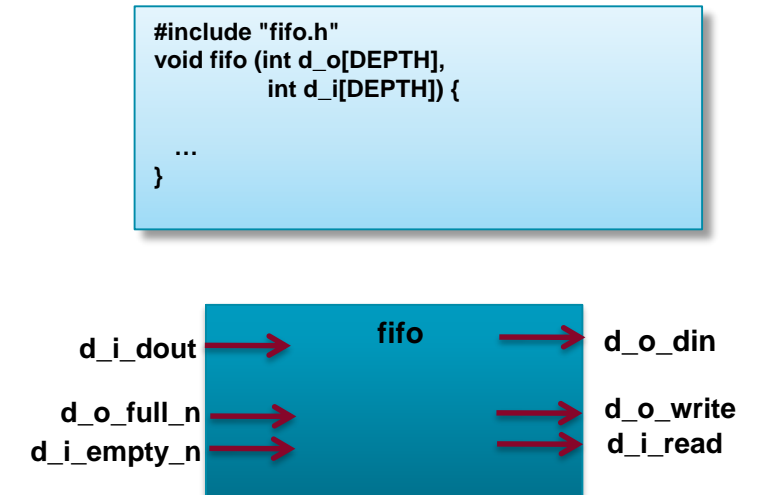
> RAM Ports

- >> 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
- >> Specify the off-chip RAM as a Resource
 - Use the RESOURCE directive on the array port

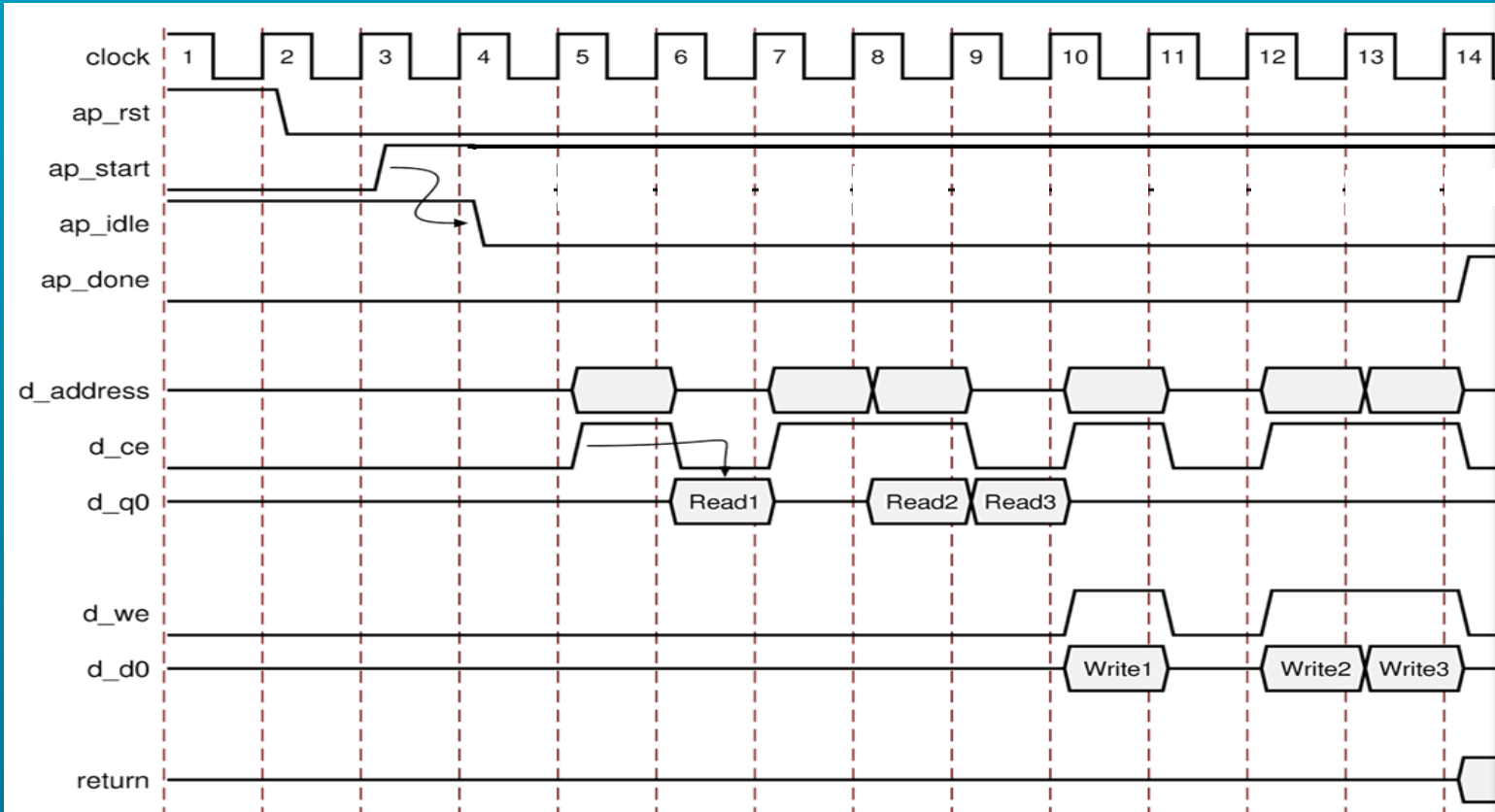


> FIFO Ports

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



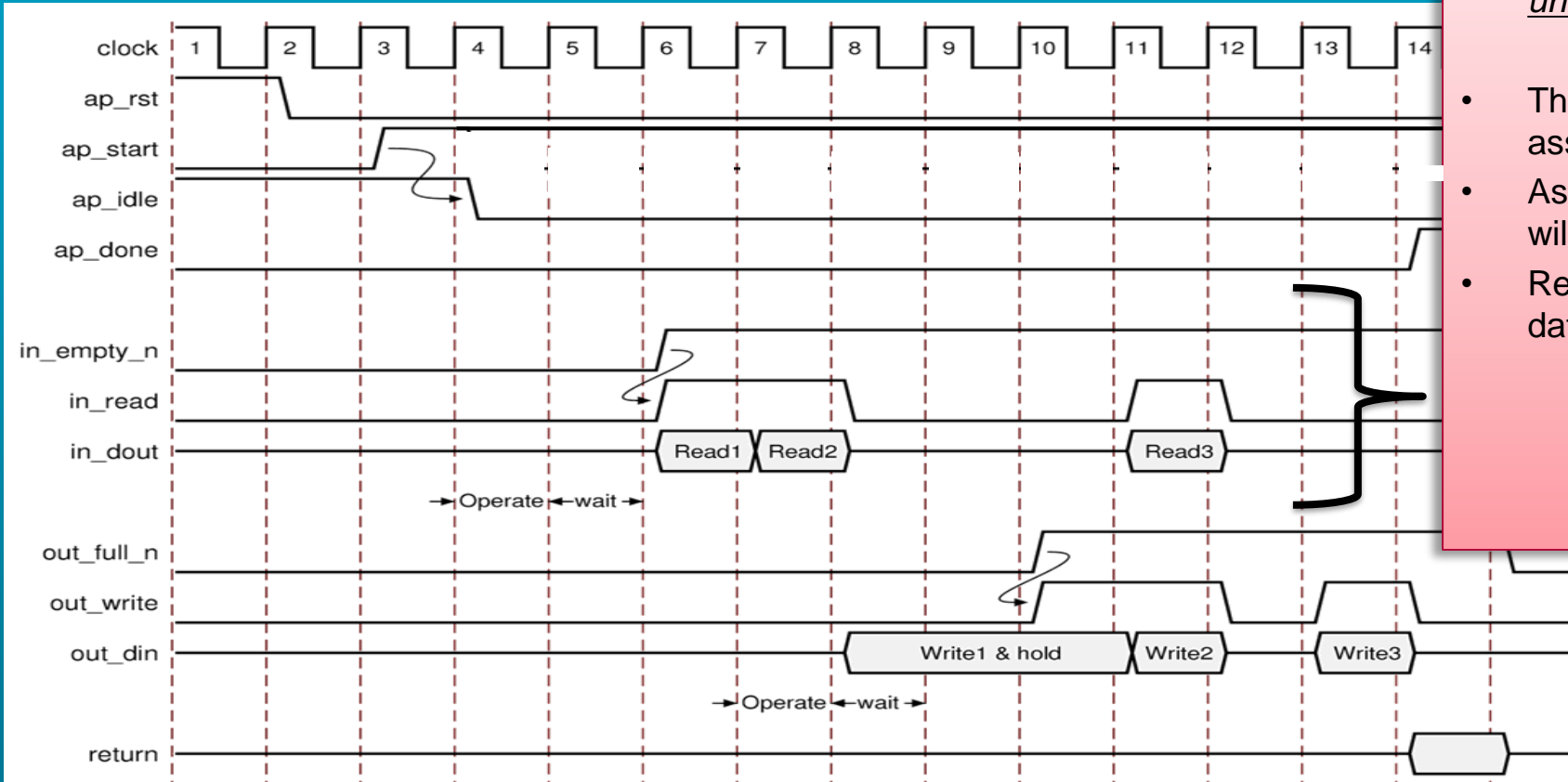
Memory IO Protocol (ap_memory)



- *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*

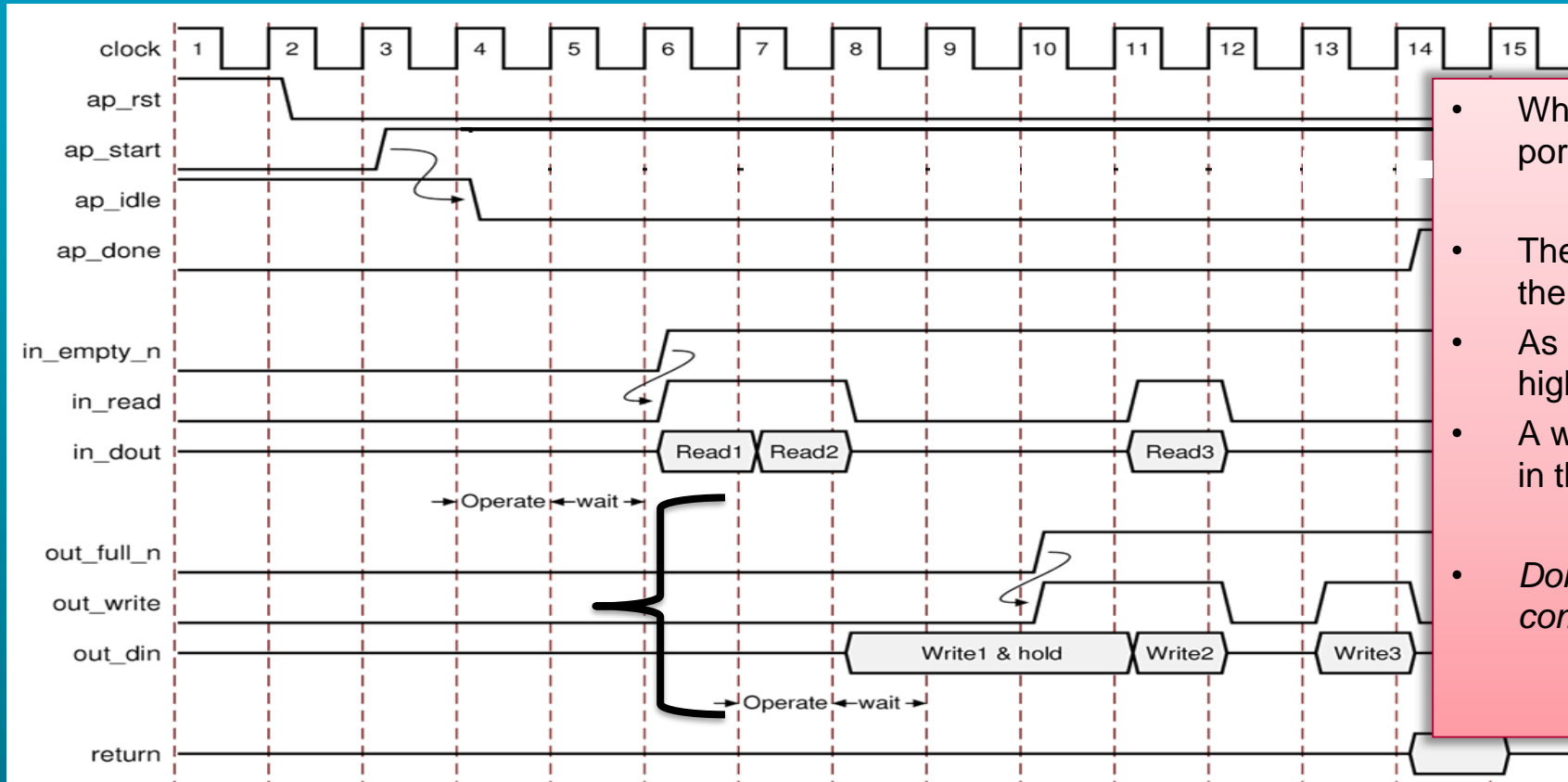
There is no stall behavior initiated by a RAM interface

FIFO IO Protocol (ap_fifo, Read)



- After start, idle goes low and the RTL operates until a read is required
- 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 (ap_fifo, 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 be 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*

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

Block Level Protocol




AXI4 Interfaces

No IO Protocol

Wire handshake protocols

Memory protocols : RAM : FIFO

Bus protocols

 Supported
  D = Default Interface
  Not Supported

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

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									
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									
ap_memory			D	D	D				
bram									
ap_fifo									D
ap_bus									

Bus IO Protocol

- > **Vivado HLS supports a Bus IO protocol**
 - >> It is that of a generic bus
 - Not an industry standard
 - >> Can be used to communicate with a bus bridge
 - >> The Vivado HLS bus protocol is principally used to connect to adapters
- > **The Bus IO protocol supports memcpy**
 - >> The bus IO protocol supports the C function memcpy
 - >> This provides a high performance interface for bursting data in a DMA like fashion
- > **The Bus IO protocol supports complex pointer arithmetic at the IO**
 - >> Pointers at the IO can be synthesized to ap_fifo or ap_bus
 - If using ap_fifo, the accesses must be sequential
 - If pointer arithmetic is used, the port must use ap_bus

Standard and Burst Mode

> Standard Mode

- >> Each access to the bus results in a request then a read or write operation
- >> Multiple read or writes can be performed in a single transaction

Single read and write in Standard Mode

```
void foo (int *d) {  
    static int acc = 0;  
    int i;  
  
    acc += d[i];  
    d[i] = acc;  
}
```

> Burst Mode

- >> Use the memcpy command
- >> Copies data between array & a pointer argument
- >> The pointer argument can be a bus interface
 - This example uses a size of 4
 - This is more efficient for higher values

Multiple reads and writes in Standard Mode

```
#include "bus.h"  
  
void foo (int *d) {  
    static int acc = 0;  
    int i;  
  
    for (i=0;i<4;i++) {  
        acc += d[i];  
        d[i] = acc;  
    }  
}
```

```
void foo (int *d) {  
    static int acc = 0;  
    int i;  
  
    for (i=0;i<4;i++) {  
        acc += *(d+i);  
        *(d+i) = acc;  
    }  
}
```

```
#include "bus.h"  
  
void foo (int *d) {  
    int buf1[4], buf2[4];  
    int i;  
  
    memcpy(buf1,d,4*sizeof(int));  
  
    for (i=0;i<4;i++) {  
        buf2[i] = buf1[3-i];  
    }  
  
    memcpy(d,buf2,4*sizeof(int));  
}
```

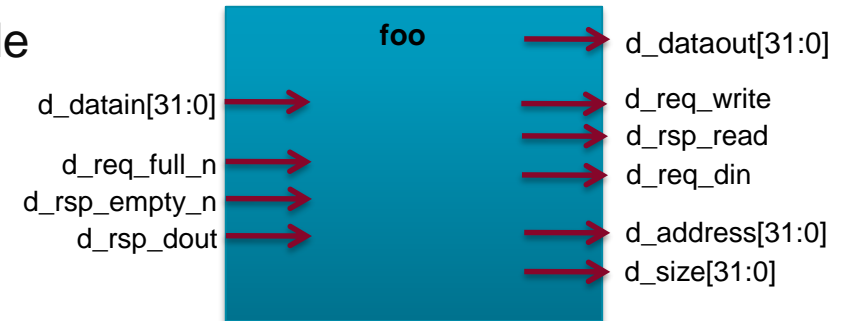
Bus IO Protocol: Ports Generated

> Bus request then access protocol

- >> The protocol will request a read/write bus access
- >> Then read or write data in single or burst mode

```
#include "foo.h"

void foo (int *d) {
  ...
}
```



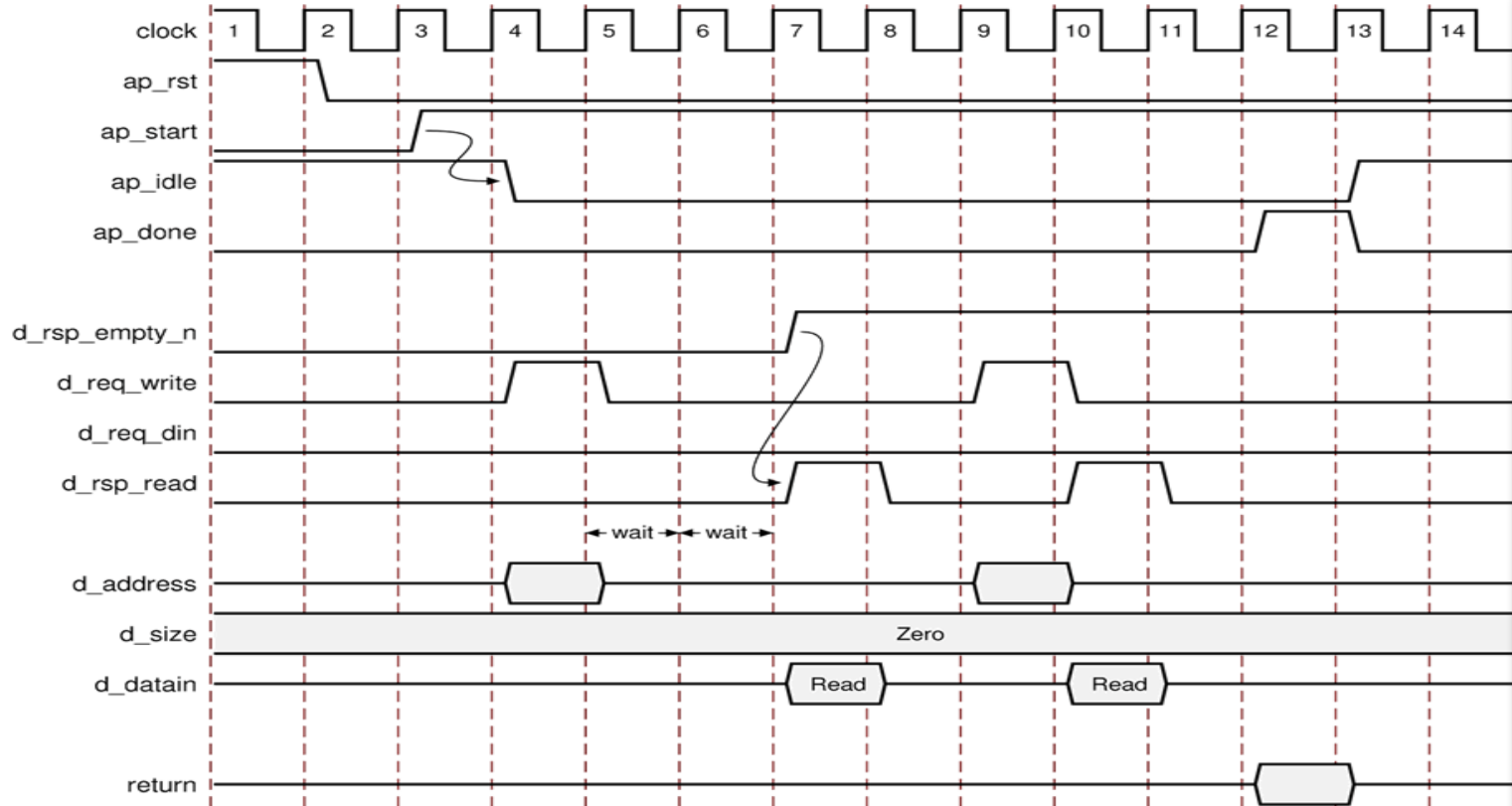
Input Ports	Description
d_datain	Input data.
d_req_full_n	Active low signal indicates the bus bridge is full. The design will stall, waiting to write.
d_rsp_empty_n	Active low signal indicates the bus bridge is empty and can accept data.

Standard Mode: The address port gives the index address e.g. $*(d+i)$, $addr = \text{value of "i"}$

Output Ports	Description
d_dataout	Output data.
d_req_write	Asserted to initiate a bus access.
d_req_din	Asserted if the bus access is to write. Remains low if the access is to read.
d_rsp_read	Asserted to start a bus read (completes a bus access request and starts reading data)
d_address	Offset for the base address.
d_size	Indicates the burst (read or write) size.

Bus IO Protocol (Standard, Read)

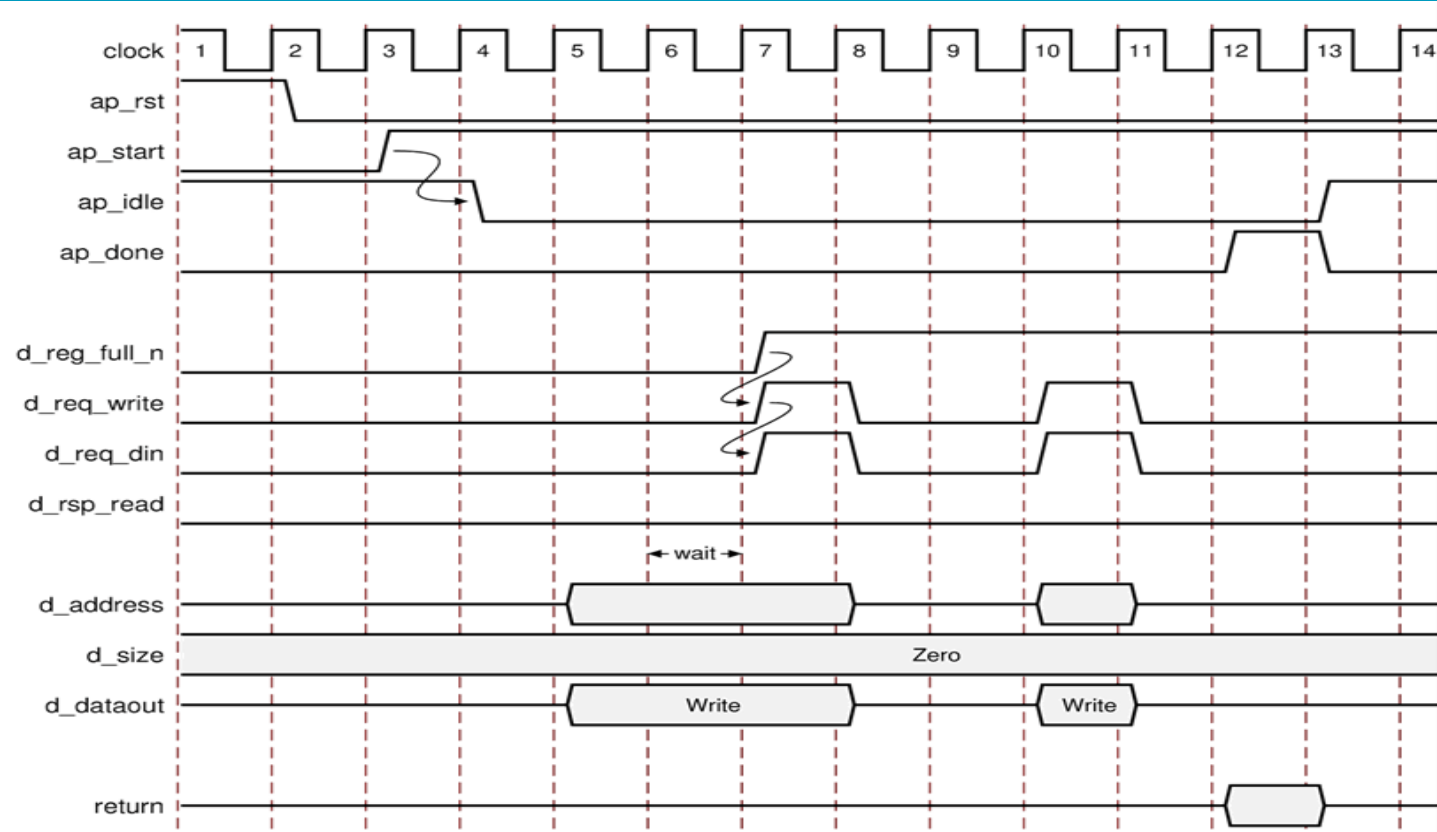
```
void foo (int *d) {  
    static int acc = 0;  
    int i;  
    for (i=0;i<4;i++) {  
        acc += d[i+1];  
        d[i] = acc;  
    }  
}
```



- *After start, idle goes low and the RTL operates until a read is required*
- A `req_write` high with `req_din` low indicates a read request
- A read address is supplied: value is the pointer index value
- The RTL will stall (wait) until the `empty_n` is asserted high to indicate data is available
- As soon as data is available, `rsp_read` will go high
- Reads will occur when required, so long as data is available (`empty_n` is high)

Bus IO Protocol (Standard, Write)

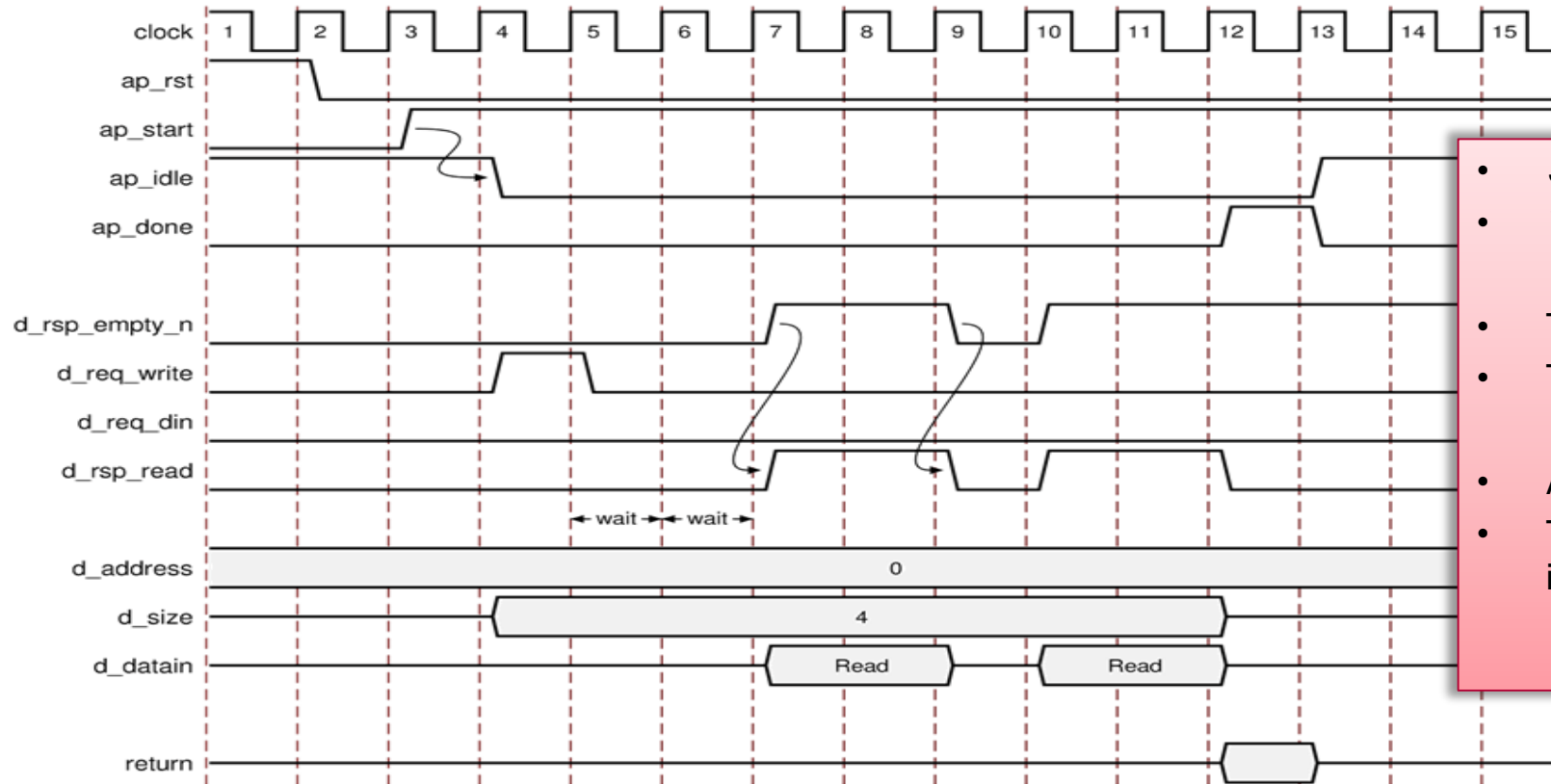
```
void foo (int *d) {  
    static int acc = 0;  
    int i;  
    for (i=0;i<4;i++) {  
        acc += d[i+1];  
        d[i] = acc;  
    }  
}
```



- After start, idle goes low and the RTL operates until a write is required
- When a write is required, data and address are applied
- The RTL will stall (wait) until the full_n is asserted high to indicate space is available
- As soon as data is available, rsp_write and req_din will go high
- A req_write high with req_din high indicates a write request
- There is no acknowledge for writes in this interface
- Write will occur when required, so long as data is available (full_n is high)

Bus IO Protocol (Burst, Read)

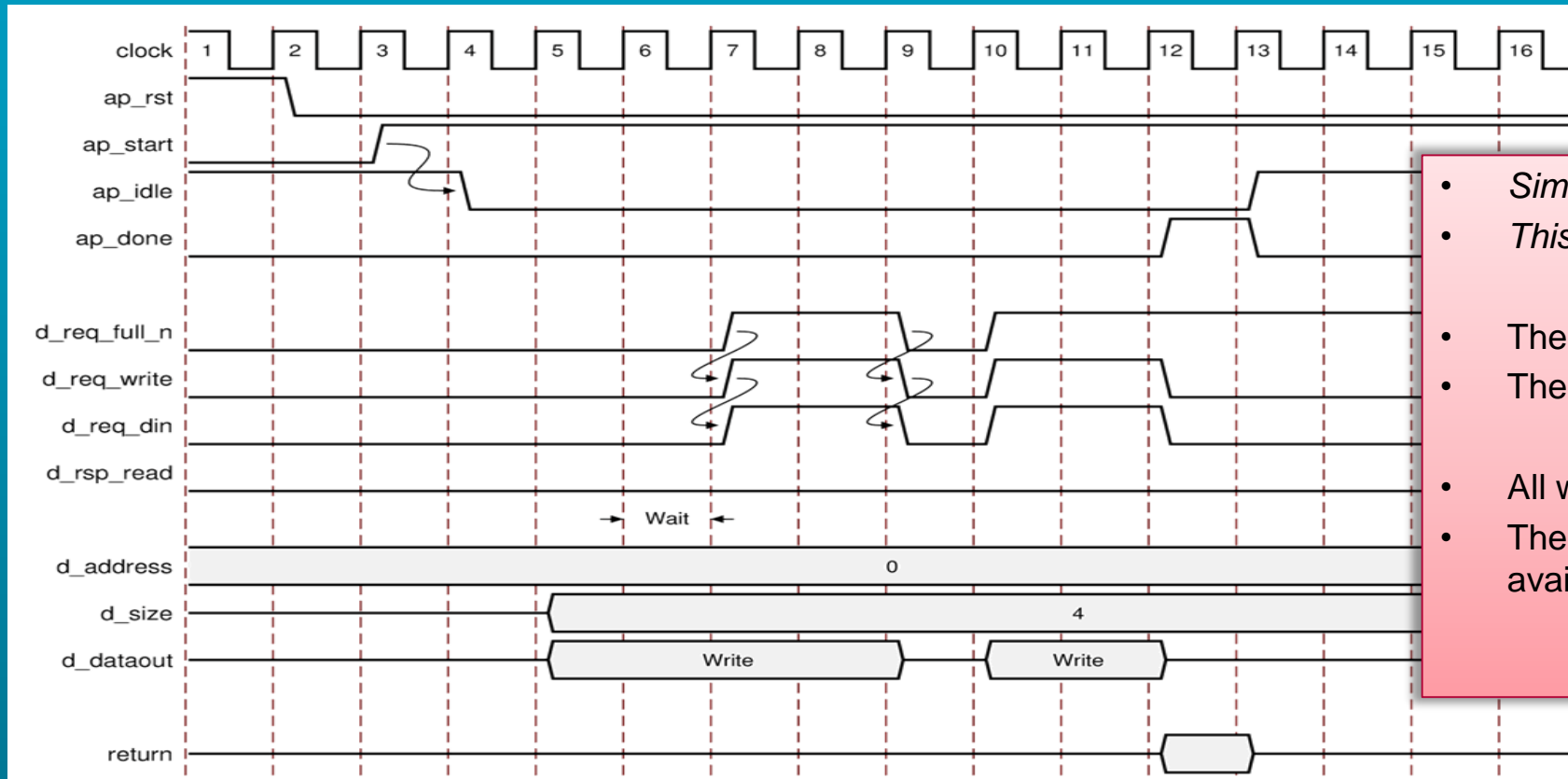
```
void foo (int *d) {  
    int buf1[4], buf2[4];  
    int i;  
    memcpy(buf1,d,4*sizeof(int));  
  
    for (i=0;i<4;i++) {  
        buf2[i] = buf1[3-i];  
    }  
    memcpy(d,buf2,4*sizeof(int));  
}
```



- *Similar to a Standard Mode*
- *This example uses a burst=4*
- The base address is zero
- The burst size is placed on port size
- All reads are done consecutively
- The empty_n signal will stall the RTL until data is available

Bus IO Protocol (Burst, Write)

```
void foo (int *d) {  
    int buf1[4], buf2[4];  
    int i;  
    memcpy(buf1,d,4*sizeof(int));  
  
    for (i=0;i<4;i++) {  
        buf2[i] = buf1[3-i];  
    }  
    memcpy(d,buf2,4*sizeof(int));  
}
```



- *Similar to a Standard Mode*
- *This example uses a burst=4*
- The base address is zero
- The burst size is placed on port size
- All writes are done consecutively
- The full_n signal will stall the RTL until data is available

Summary



Summary

- > **Vivado 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 IP 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