Bridge of Life Education

SOC Design

Kernel IO Interface

Topics

- Overview
- Block Level Protocol
  - ap_ctrl_hs
  - ap_ctrl_chain - advanced
  - ap_ctrl_none (datflow) - advanced
- Port Level Protocol
  - s_axilite
  - m_axi - axi master
  - axis - axi stream
- Pragma & Examples

Overview

Mapping of Key Attributes of C Code

Function: design hierarchy, mapped to MODULE

Arguments : mapped to Input/output interface of the hardware

Types: All variables are of a defined type, influence the area and the performance

Loops: impact on area and performance, HLS opt with Directive Pragma

Control flow: Control logic

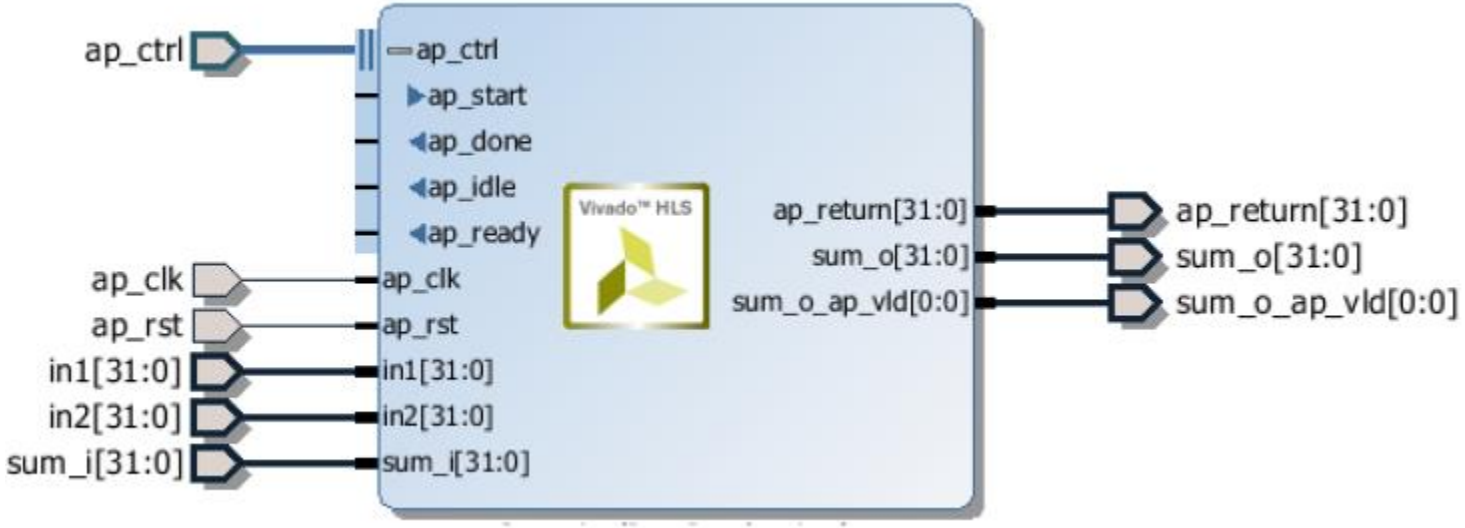
Arrays: impact the device area, and performance bottleneck

Operators: Function unit. Allocation/Scheduling (Sharing) to meet performance and area

```
46  #include "fir.h"
47
48  void fir (
49      data_t *y,
50      coef_t c[N],
51      data_t x
52  ) {
53
54
55
56
57      int i;
58
59      acc=0;
60      Shift_Accum_Loop: for (i=N-1;i>=0;i--) {
61          if (i==0) {
62              shift_reg[0]=x;
63              data = x;
64          } else {
65              shift_reg[i]=shift_reg[i-1];
66              data = shift_reg[i];
67          }
68          acc+=data*c[i];;
69      }
70      *y=acc;
71  }
```

Overview

```
#include "sum_io.h"
dout_t sum_io (     din_t       in1,
                    din_t        in2,
                    dio_t       *sum) {
            dout_t temp;
            *sum = in1 + in2 + *sum;
            temp = in1 + in2;
            return temp;
}
```



Top Function Communicates with Host

- **Block Protocol**: specified on the s_axilite interface (**ap_ctrl_hs**, ap_ctrl_chain, ap_ctrl_none)
- **Port Protocol**: AXI4-Master, AXI-Lite , AXI4-Stream
  - Scalar inputs: AXI4-Lite interface (s_axilite)
  - Pointer to an Array:
    - AXI4 memory-mapped interface (m_axi) to access memory and
    - s_axilite interface to specify the base address to the memory address space.
  - Function Return: ap_return port added to the s_axilite interface
  - Arguments specified as hls::stream: default to AXI4-Stream interface (axis)
- HLS will produce an associated set of C driver files during the Export RTL process

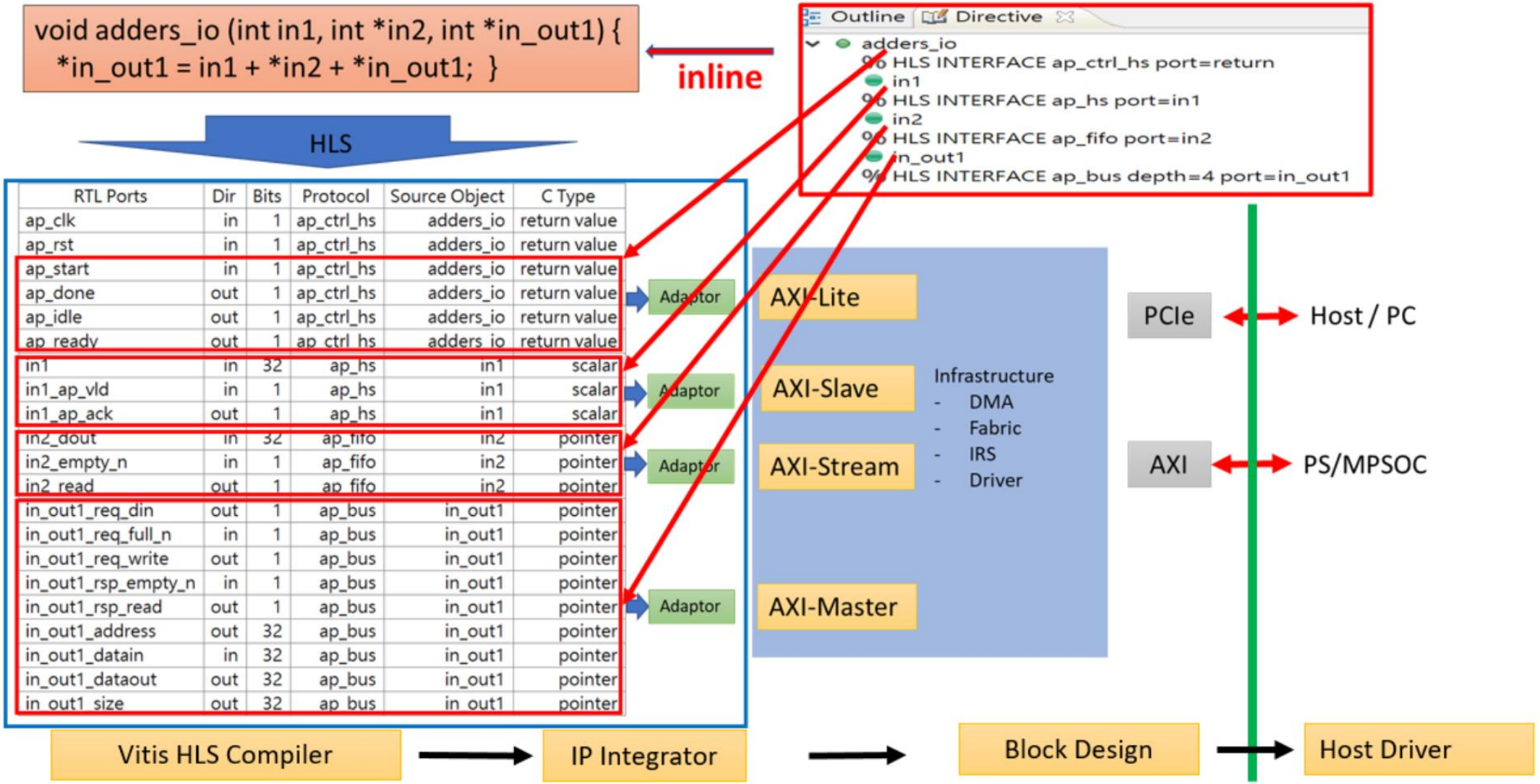
Block-Level Interface Protocol

```
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/COH)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
```

- Host application/driver controls kernel functions
- Block-level protocol (ap_ctrl_hs, ap_ctrl_chain) specifies:
  - When the design can start to perform the operation
  - When the operation ends
  - When the design is idle and ready for new inputs
- After the block-level protocol starts the operation of the block, the port-level I/O protocols are used to sequence data into and out of the block


Allocate Buffers in Host Memory
Create Buffers in Global Memory
Set Kernel Arguments
Data Transfer
Execute Kernel
Synchronize Event

Construction Flow for Host/Kernel Communication



Block Level Interface Protocol

- AP_CTRL_HS (Default: Sequential Mode)
- AP_CTRL_CHAIN (Pipeline Mode)
- AP_CTRL_NONE (Free-running Mode)

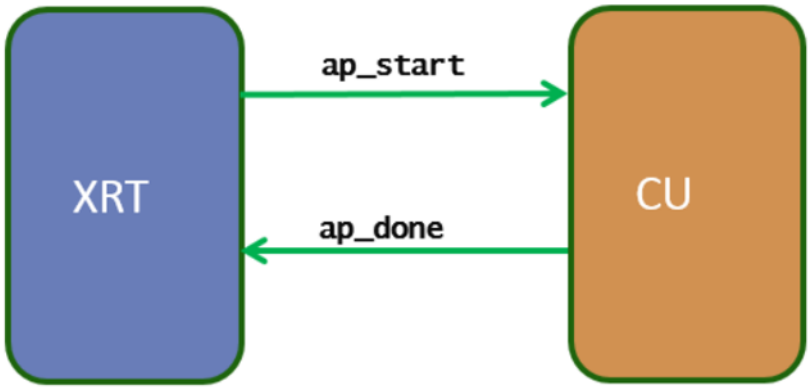
AP_CTRL_HS (Default)

Block Level Protocol Specified on port: return

```
// kernel without chain
void krnl_simple_mmult(int* a, int* b, int* c, int* d, int* output, int dim) {
....
#pragma HLS INTERFACE   ap_ctrl_hs          port = return
```

AP_CTRL_HS (Sequential Executed Kernel)

- Host and Kernel Synchronization by
  - ap_start
  - ap_done
- Kernel can only be restarted (ap_start), after it completes the current execution (ap_done)
- Serving one execution request a time



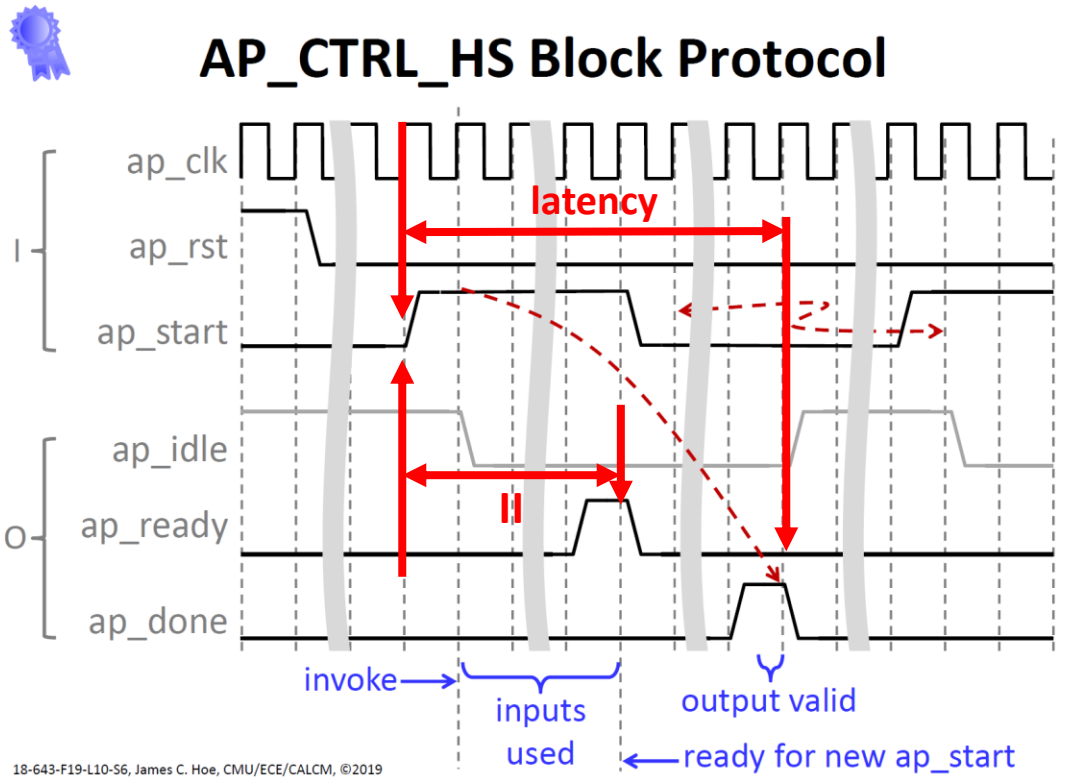
https://xilinx.github.io/XRT/master/html/xrt_kernel_executions.html

AP_CTRL_HS Protocol

- ap_start (i): set 1 to start until ap_ready asserted.
- ap_ready (o): design is ready to accept new input
- ap_done (o): design completes all operation.

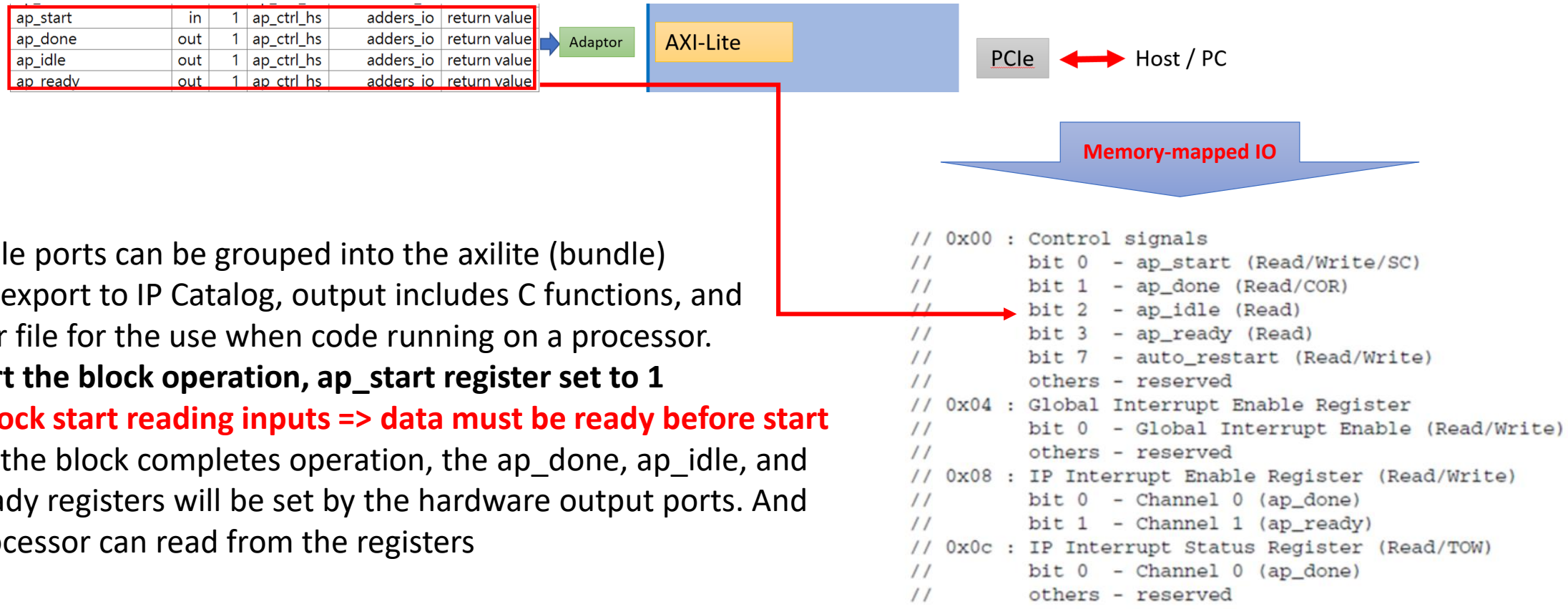
  Indicates data on ap_return is valid
- ap_idle (o): indicate design is idle if high.
- (ap_return): return data

- Pipeline/Nonpipeline depends on ap_ready timing



Ug902,p#78: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf

Host (PS/Processor) to control HLS block – s_axilite



- Multiple ports can be grouped into the axilite (bundle)
- When export to IP Catalog, output includes C functions, and header file for the use when code running on a processor.
- **To start the block operation, ap_start register set to 1**
- **The block start reading inputs => data must be ready before start**
- When the block completes operation, the ap_done, ap_idle, and ap_ready registers will be set by the hardware output ports. And PS/Processor can read from the registers

AP_CTRL_HS : HOST/XRT

- The HOST/XRT driver writes a 1 in ap_start to start the kernel
- The HOST/XRT driver waits for ap_done asserted by kernel (guaranteeing the output data is fully produced by the kernel).
- Repeat 1-2 for the next kernel execution

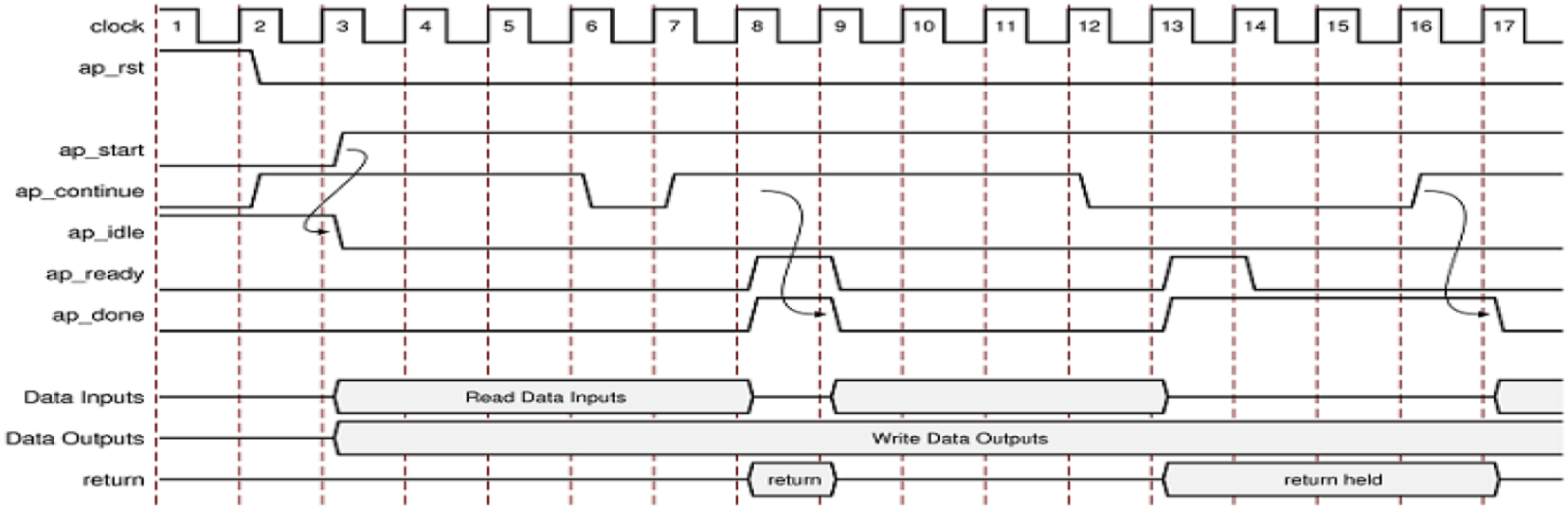
AP_CTRL_CHAIN - Advanced

AP_CTRL_CHAIN (Pipelined kernel)

For cascaded block, **a mechanism for consumer block to back-pressure on producer block**.
ap_continue – active high indicates the downstream block is ready to consumes for new data input.

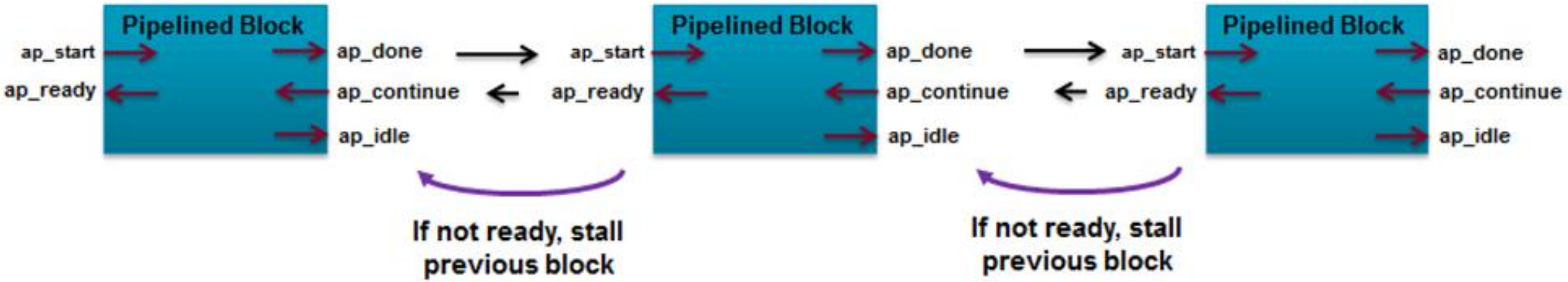
- ap_continue is low, prevents the upstream block from generating additional data.
- down stream ap_ready can drive ap_continue port.
- ap_continues is High when ap_done is High, continue operating
- ap_continue Low, ap_done is High, stops operating, ap_done remains high, data remains valid on the ap_return port.

Figure 100: **Behavior of ap_ctrl_chain Interface**



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

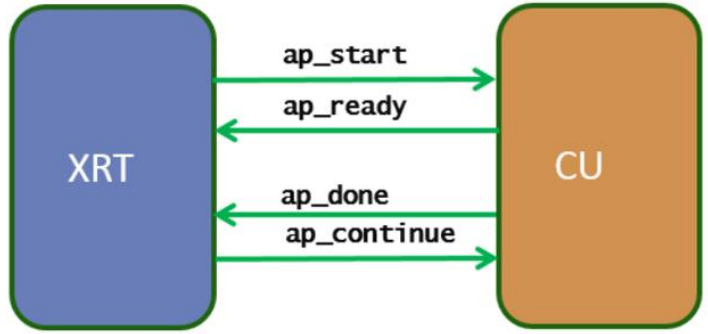


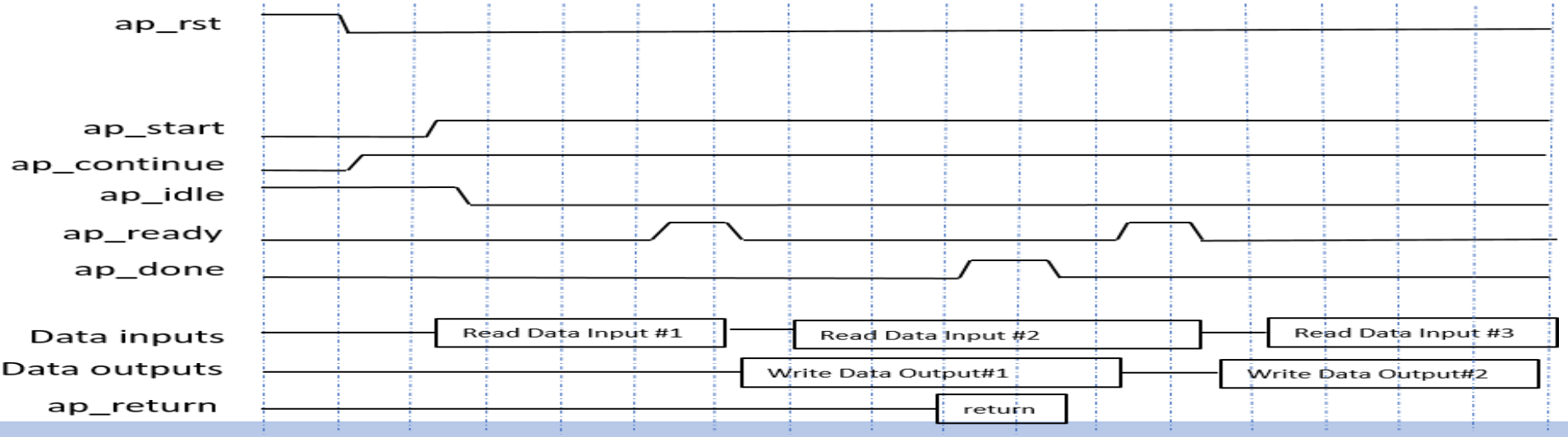
Host Control Pipeline Kernel Execution

Table 9: **Control Register Signals**

Bit	Name	Description
0	ap_start	Asserted when the kernel can start processing data. Cleared on handshake with ap_done being asserted.
1	ap_done	Asserted when the kernel has completed operation. Cleared on read.
2	ap_idle	Asserted when the kernel is idle.
3	ap_ready	Asserted by the kernel when it is ready to accept the new data
4	ap_continue	Asserted by the XRT to allow kernel keep running
7	auto_restart	Used to enable automatic kernel restart as described in Working with Auto-Restarting Kernels.
31:5	Reserved	Reserved

1. Input Synchronization(ap_start, ap_ready)
   - XRT writes a 1 in ap_start to start the kernel
   - XRT waits for ap_ready
   - XRT write 1 in ap_start to start the kernel again
2. Output Synchronization (ap_done, ap_continue)
   - XRT waits for ap_done asserted by the kernel (output data is produced)
   - XRT write1 a 1 in ap_continue to keep kernel running.
3. The two processes (Input Synchronization, Output Synchronization) run asynchronously)
4. Enqueue multiple requests, and data buffers ahead of time, e.g. clEnqueueMigrateMemObjects





https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Control-Requirements-for-XRT-Managed-Kernels

Performance Difference between pipeline, non-pipeline

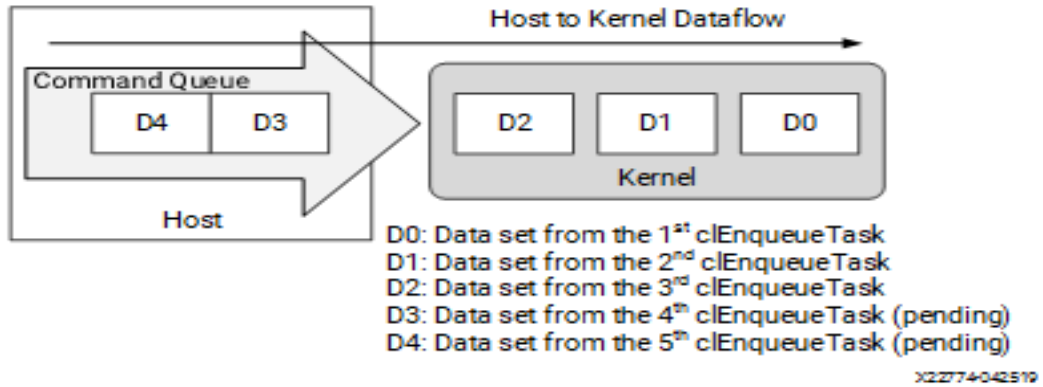
```
// kernel without chain
void krnl_simple_mmult(int* a, int* b, int* c, int* d, int* output, int dim) {
....
#pragma HLS INTERFACE ap_ctrl_hs port = return
```

```
// kernel with chain
void krnl_chain_mmult(int* a, int* b, int* c, int* d, int* output, int dim) {
....
#pragma HLS INTERFACE ap_ctrl_chain port = return

#pragma HLS DATAFLOW
   mm2s(a, strm_a, strm_ctrl_trans1, strm_ctrl_trans2);
   mmult(strm_a, b, strm_ctrl_trans2, strm_b, strm_ctrl_trans3);
   mmult(strm_b, c, strm_ctrl_trans3, strm_c, strm_ctrl_trans4);
   mmult(strm_c, d, strm_ctrl_trans4, strm_d, strm_ctrl_trans5);
   s2mm(strm_d, output, strm_ctrl_trans5);
}
```

```
// host code
for (int i = 0; i < NUM_TIMES; i++) {
      krnl_chain_mmult.setArg(0, buffer_in1[i]));
      krnl_chain_mmult.setArg(1, buffer_in2[i]));
      krnl_chain_mmult.setArg(2, buffer_in3[i]));
      krnl_chain_mmult.setArg(3, buffer_in4[i]));
      krnl_chain_mmult.setArg(4, buffer_output[i]));
      krnl_chain_mmult.setArg(5, MAT_DIM));

       // Copy input data to device global memory
       q.enqueueMigrateMemObjects({buffer_in1[i], buffer_in2[i], buffer_in3[i],
buffer_in4[i]}, 0 /* 0 means from host*/));
       q.enqueueTask(krnl_chain_mmult));
}
```

```
----------------------------------------------------
                Performance Summary
----------------------------------------------------
------------------------+---------------------------
Kernel(10 iterations)   |    Wall-Clock Time (s)
------------------------+---------------------------
krnl_chain_mmult        | 0.00305935
krnl_simple_mmult       | 0.00534336
------------------------+---------------------------
Speedup:                | 1.74657
------------------------+---------------------------
```

Figure: Host to Kernel Dataflow



https://github.com/Xilinx/Vitis_Accel_Examples/tree/2020.2/cpp_kernels/kernel_chain

AP_CTRL_NONE – For Dataflow

AP_CTRL_NONE (Continuously Running Kernel)

Host communicates with kernel through stream - Kernel starts execution when the data is available at its input

- Only when it has no memory-mapped input and output.
- No need to start the kernel by clEnqueueTask from the host.
- Useful for data-driven designs with streaming data coming from and going to the I/O pins (Ethernet, SerDes) of the FPGA, or streamed from or to a different kernel (kernel-to-kernel streaming).
- Don't use clSetKernelArg to pass a scalar argument to ap_ctrl_none kernel; only use xclRegWrite (API implemented in 2019.2) API

> **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

https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Auto-Restarting-Kernels

Port Level – AXI Interface

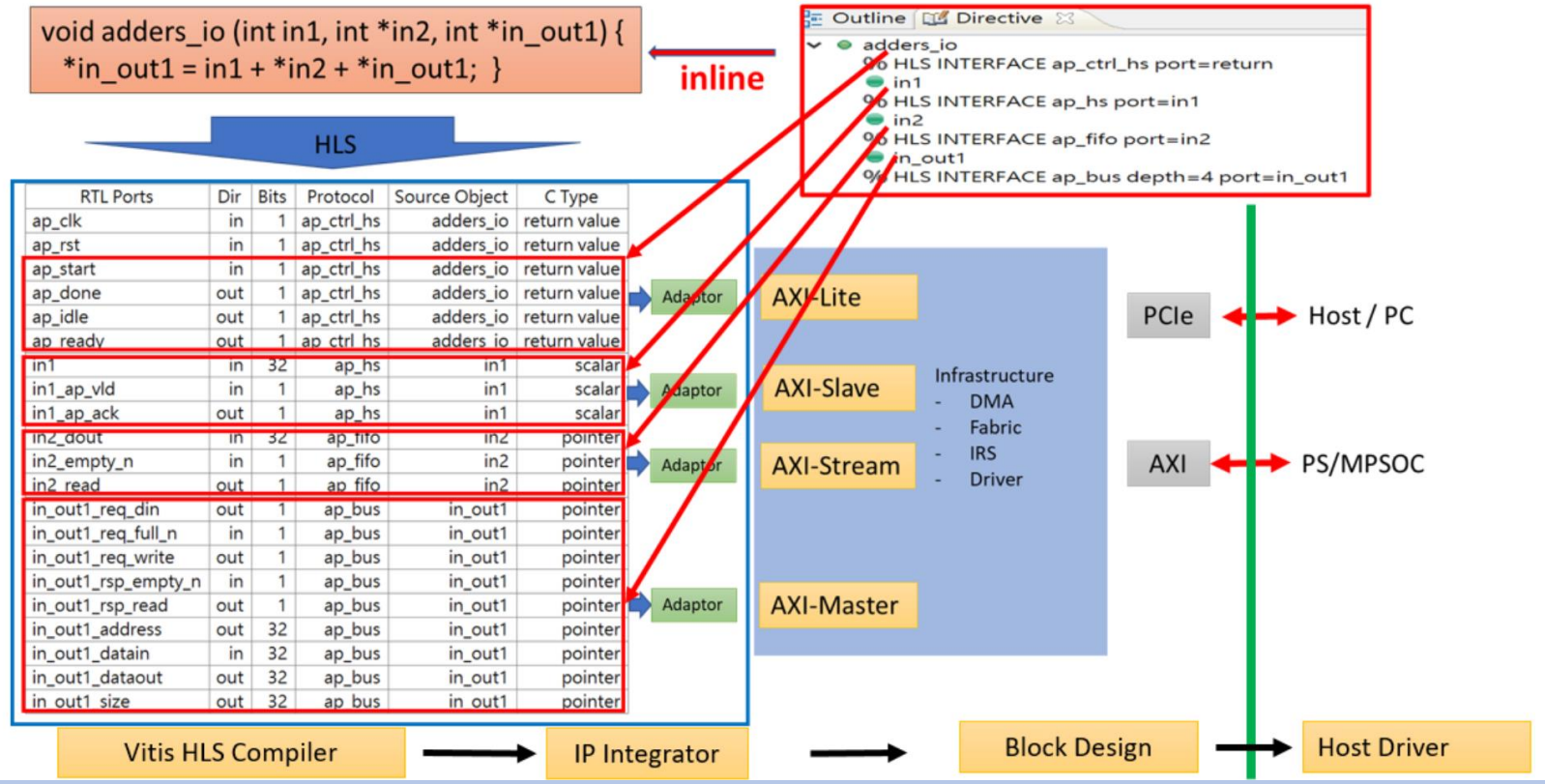
s_axilite,

m_axi

axis

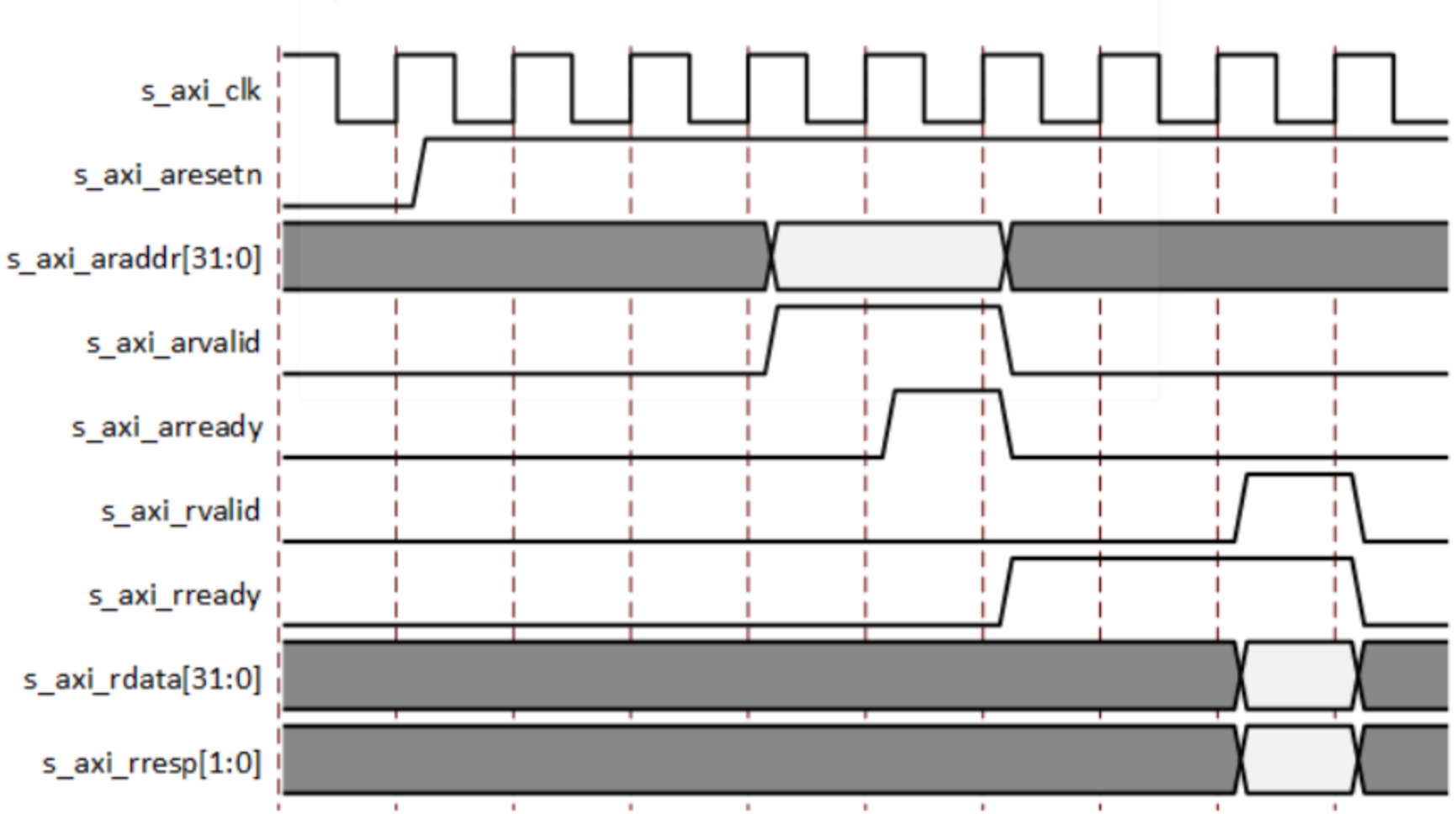
axis, s_axilite, m_axi

- s_axilite - AXI4-Lite – group multiple argument into the same AXI4-Lite
- m_axi - AXI4 Master – array or pointer, group multiple ports (bundle)
- axis - AXI4-Stream – for input or output but not io, array

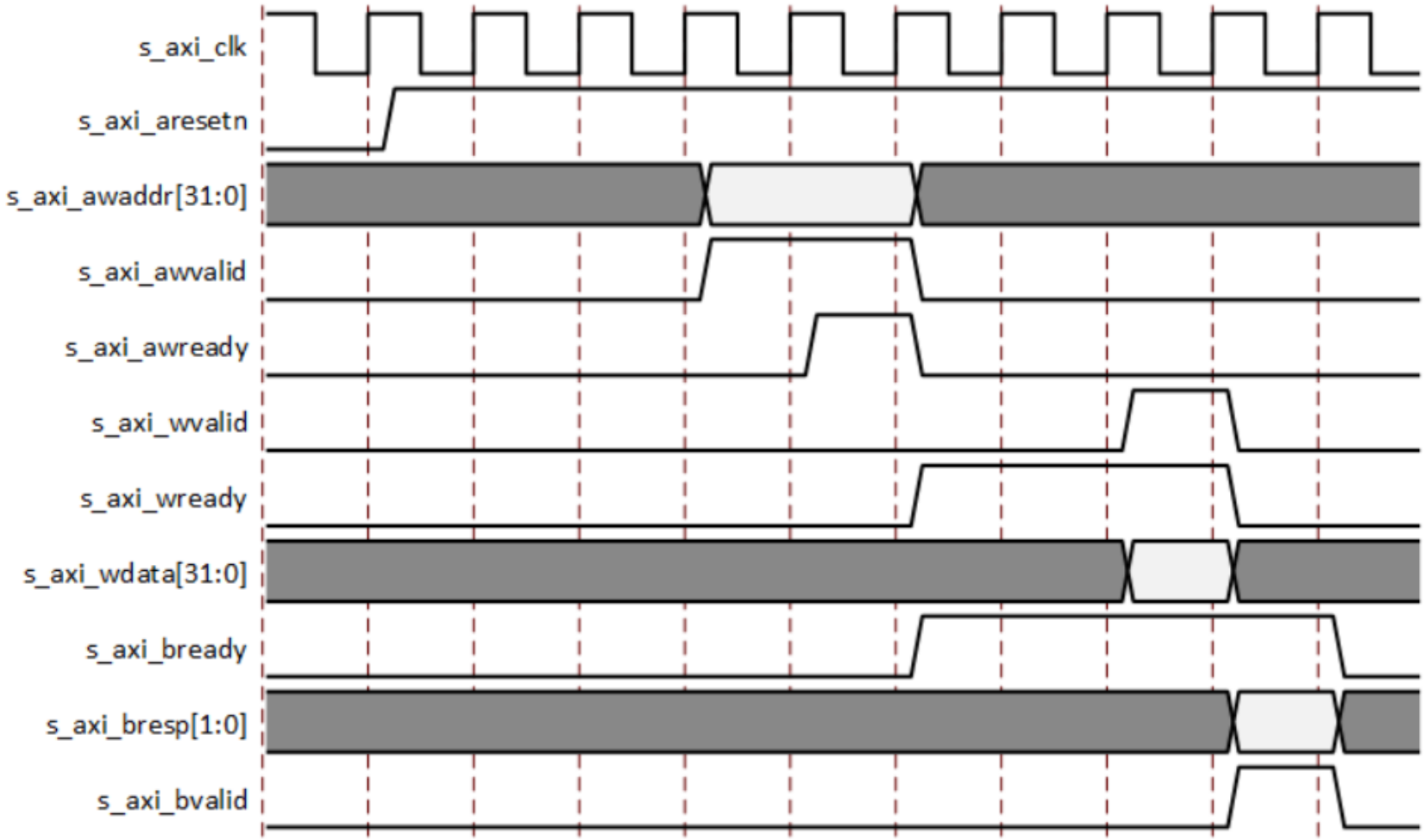


Port Level – S-AXILITE

AXI4-Lite Read Transaction



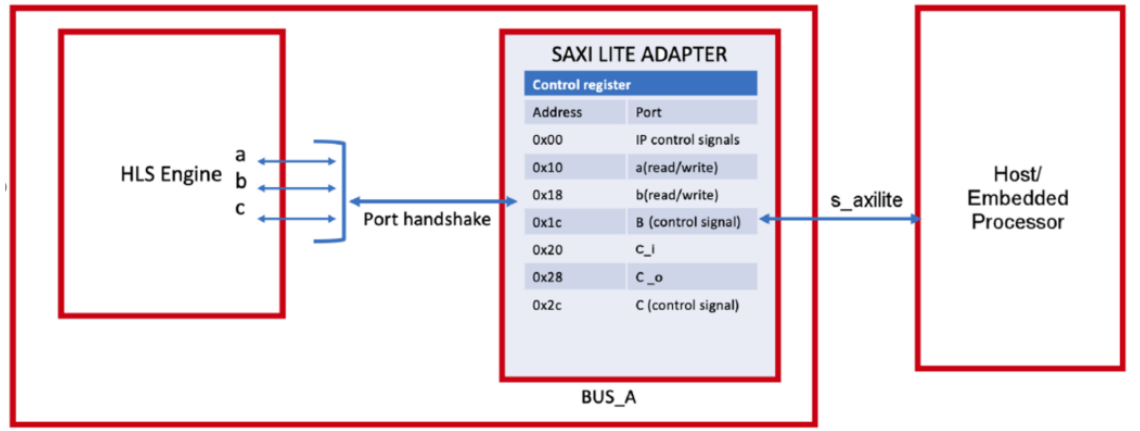
AXI4-Lite Write Transaction



AXI-Lite Interface (s_axilite)

- Allow design controlled by a CPU
- **Port return**: specify the block-level protocol. (s_axilite: defaut)
- Bundle: Group multiple arguments, or separate axilite
- Offset: specified starting address, or tool automatically assigns
- Output C driver files for use with code running on a processor
- Address 0x0000-0x000F for block I/O protocol and interrupt control
- When the AXI4-lite interface is selected
  - Default mode for input ports is ap_none
  - Default mode for output port is ap_vld
  - Default mode for function return port is ap_ctrl_hs
- Controlling Hardware – _hw.h
  - Start block operation ap_start register set to 1
  - Block complete ap_done, ap_idle, and ap_ready set by hardware

```
#include <stdio.h>
void example(char *a, char *b, char *c)
{
  #pragma HLS INTERFACE s_axilite    port=return bundle=BUS_A
  #pragma HLS INTERFACE ap_ctrl_hs   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
  #pragma HLS INTERFACE ap_vld       port=b

  *c += *a + *b;
}
```



S_AXILITE and Block Level Protocol

- Port return: specify the block-level protocol. The interface pragma is specified as

  #pragma HLS INTERFACE s_axilite port=return bundle=BUS_A

  It is the default block-level protocol.
- You can also assign the block control protocol to the interface. As an example

  #pragma HLS INTERFACE s_axilite port=return bundle=BUS_A

  #pragma HLS INTERFACE ap_ctrl_hs port=return bundle=BUS_A
- In the Control Register Map, HLS reserve address 0x00 – 0x0C for the block level protocols and interrupt controls.
  - The Control register (0x00) contains ap_start (bit-0), ap_done (bit-1), ap_ready(bit-3) and ap_idle (bit_2); in the case of ap_ctrl_chain, it contains ap_continue. These signals are accessed through the s_axilite adapter.

S-AXILITE and Port-Level Protocol

```
#include <stdio.h>
void example(char *a, char *b, char *c)
{
   #pragma HLS INTERFACE s_axilite port=return bundle=BUS_A
   #pragma HLS INTERFACE ap_ctrl_hs  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
   #pragma HLS INTERFACE ap_vld                  port=b

   *c += *a + *b;
}
```

#pragma HLS INTERFACE s_axilite port=b bundle=BUS_A
#pragma HLS INTERFACE ap_vld port=b

```
// 0x18 : Data signal of b
//                      bit 7~0 - b[7:0] (Read/Write)
//                      others - reserved
// 0x1c : Control signal of b
//                      bit 0 - b_ap_vld (Read/Write/SC)
//                      others - reserved
```

#pragma HLS INTERFACE s_axilite port=a bundle=BUS_A

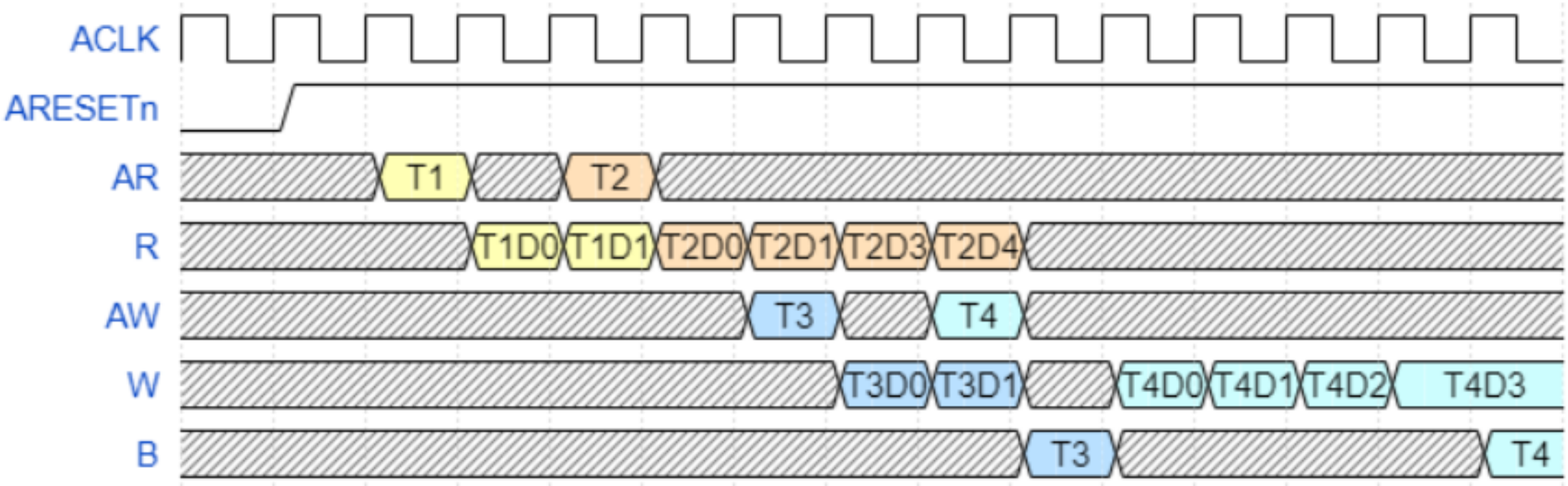
```
// 0x10 : Data signal of a
//                      bit 7~0 - a[7:0] (Read/Write)
//                      others - reserved
```

#pragma HLS INTERFACE s_axilite port=c bundle=BUS_A

```
// 0x20 : Data signal of c_i
//                      bit 7~0 - c_i[7:0] (Read/Write)
//                      others - reserved
// 0x24 : reserved
// 0x28 : Data signal of c_o
//                      bit 7~0 - c_o[7:0] (Read)
//                      others - reserved
// 0x2c : Control signal of c_o
//                      bit 0 - c_o_ap_vld (Read/COR)
//                      others - reserved
```

Port Level – AXI4_Master (M_AXI)

AXI Master Transaction Waveform



Example of multiple transactions: T1 (arlen=1), T2 (arlen=3) read transactions and T3 (awlen=1), T4 (awlen=3) write transactions.

Single v.s. Burst Transfer

- Pointer and array argument (default to m_axi)
- Offset = slave, transfer address is defined by axilite.
- Default alignment is set to 64 bytes
- Maximum read/write burst length is set to 16 by default

Single Transfer

Burst Transfer

- memcpy
- For-loop + PIPELINE
  - Pipeline the loop
  - Access address **in increasing order**
  - Do not place accesses inside a **conditional statement**
  - For nested loops, do not flatten loops, because this inhibits the burst operation
  - **Only one read and one write is allowed in a for loop** unless the ports are bundled in different AXI ports.

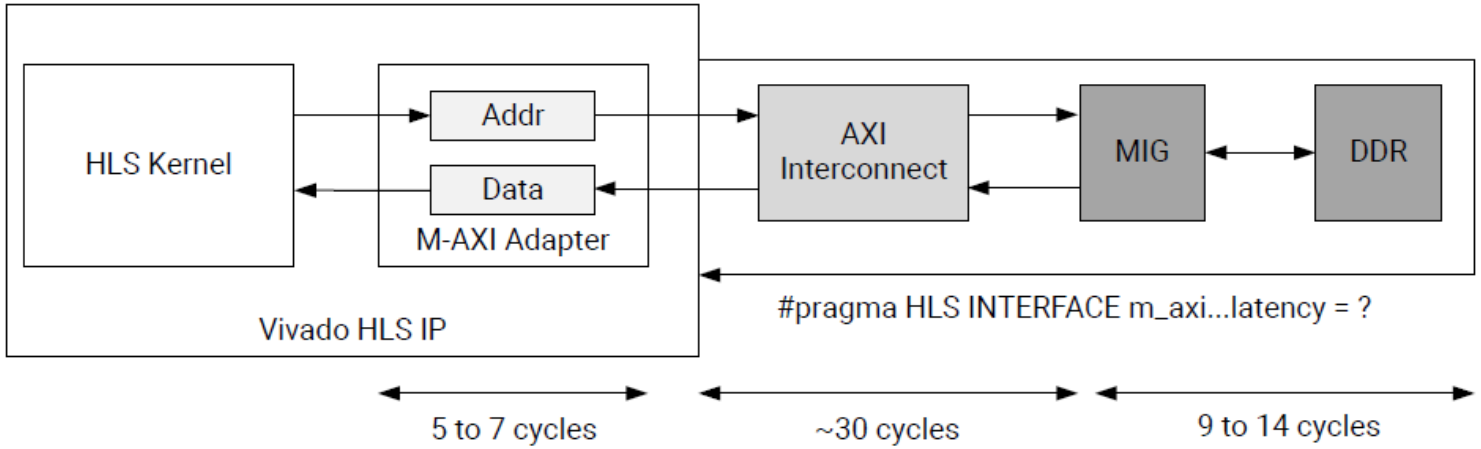
```
void bus (int *d) {
    static int acc = 0;
    acc += *d;
    *d = acc;
}
```

```
void example(volatile int *a){
   #pragma HLS INTERFACE m_axi      depth=50    port=a
   #pragma HLS INTERFACE s_axilite  port=return
   int i, buff[5];

   //memcpy creates a burst access to memory
   memcpy(buff, (const int*) a, 50*sizeof(int));

   for(i=0; i < 50; i++){
      #pragma HLS PIPELINE
      buff[i] = buff[i] + 100; }

  // for loop + PIPLEINE
   for(i=0; i < 50; i++){
      #pragma HLS PIPELINE
      a[i] = buff[i]; }
}
```

AXI Master Characteristics

- Max burst-length <= 256 (limited by AXI bus protocol ARLEN[7:0])
- Max transfer size <= 4KB
- Do not cross 4KB boundary
- Bus width – power of 2, between 32 bits and 512 bits
  - What if not matches actual data size, e.g. video data RGB 24-bit?

Latency & Bus utilization



- **depth**: Specifies the maximum number of samples for the test bench to process.
- **latency**: Specifies the expected latency of AXI4 interface.
- **max_read_burst_length**, **max_write_burst_length**: Specifies the maximum number of data values read during a burst transfer.
- **num_read_outstanding**, **num_write_outstanding**: Specifies how many read requests can be made to the AXI4 bus without a response before the design stalls. It needs storage to hold num_read_outstanding requests

  Buffer size = num_read_outstanding * max_read_burst_length * word_size.

Question: With 128-bit bus width, what is the max burst length to specify?

Best Practices for Designing with M_AXI Interfaces
https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Best-Practices-for-Designing-with-M_AXI-Interfaces

Experiment on AXI Burst Performance

- AXI burst performance affected by
  - Latency
  - Burst Length
  - Num_outstanding
  - Bus width
- Measure the time to read/write a buffer from DDR

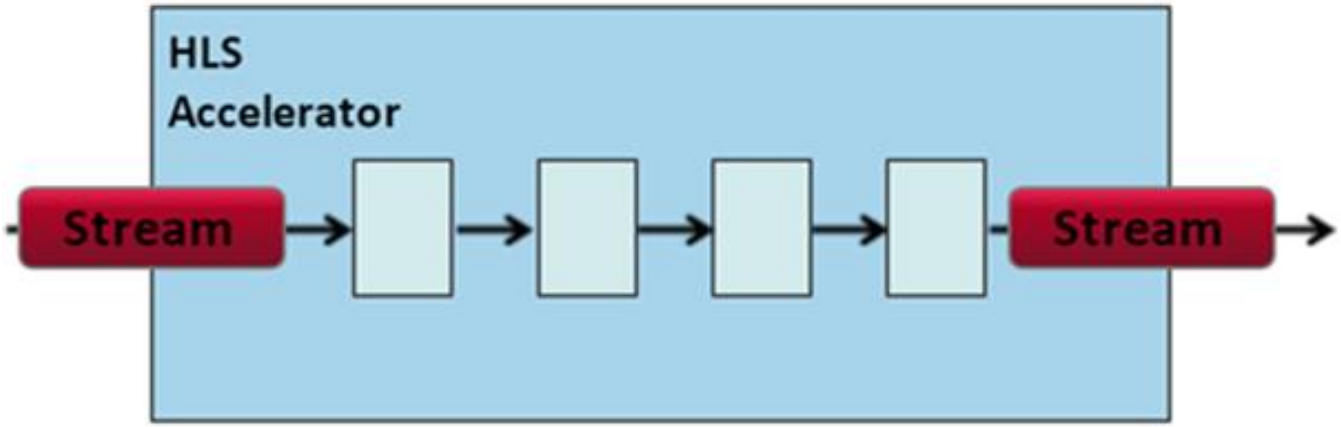
```
// Data Width - 256
void test_kernel_maxi_256bit_1(int64_t buf_size, int direction, int64_t* perf,
ap_int<256>* mem) {
// Data Width - 512
// void test_kernel_maxi_512bit_1(int64_t buf_size, int direction, int64_t* perf,
ap_int<512>* mem) {

#pragma HLS INTERFACE m_axi port = mem bundle = aximm0  \
                    num_write_outstanding = 4   \
                    max_write_burst_length = 4  \
                    num_read_outstanding = 4    \
                    max_read_burst_length = 4   \
                    offset = slave
#pragma HLS DATAFLOW
   hls::stream<int64_t> cmd;
   testKernelProc(mem, buf_size, direction, cmd, 32);
   perfCounterProc(cmd, perf, direction, 4, 4);
}
```

```
Kernel->AXI Burst READ performance
Data Width = 512 burst_length = 4 num_outstanding = 4 buffer_size = 16.00 MB | throughput = 3.92988 GB/sec
Data Width = 512 burst_length = 16 num_outstanding = 4 buffer_size = 16.00 MB | throughput = 13.1114 GB/sec
Data Width = 512 burst_length = 32 num_outstanding = 4 buffer_size = 16.00 MB | throughput = 16.8218 GB/sec
Data Width = 512 burst_length = 4 num_outstanding = 32 buffer_size = 16.00 MB | throughput = 16.8222 GB/sec
Data Width = 512 burst_length = 16 num_outstanding = 32 buffer_size = 16.00 MB | throughput = 16.8295 GB/sec
Data Width = 512 burst_length = 32 num_outstanding = 32 buffer_size = 16.00 MB | throughput = 16.8219 GB/sec
```

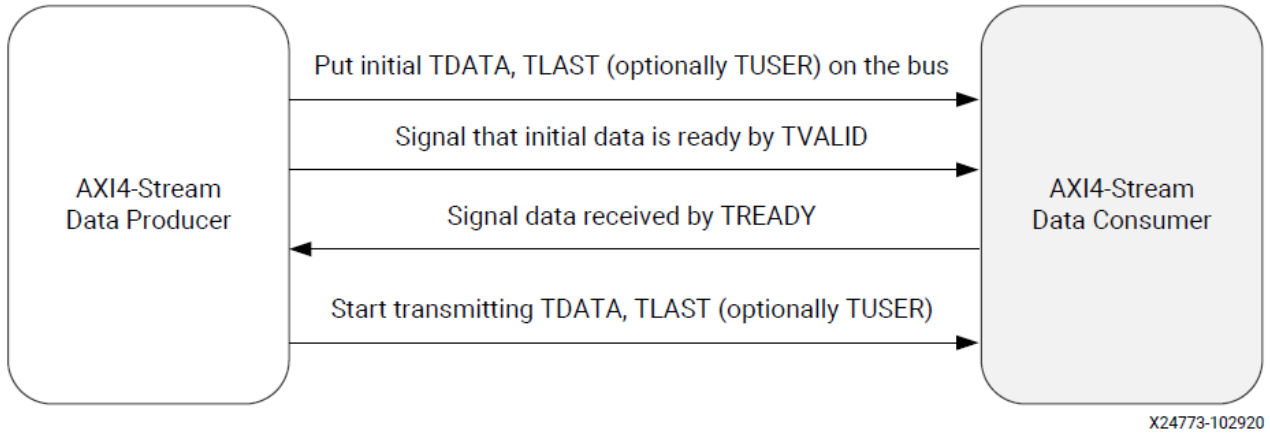
Port Level – AXI4_STREAM (AXIS)

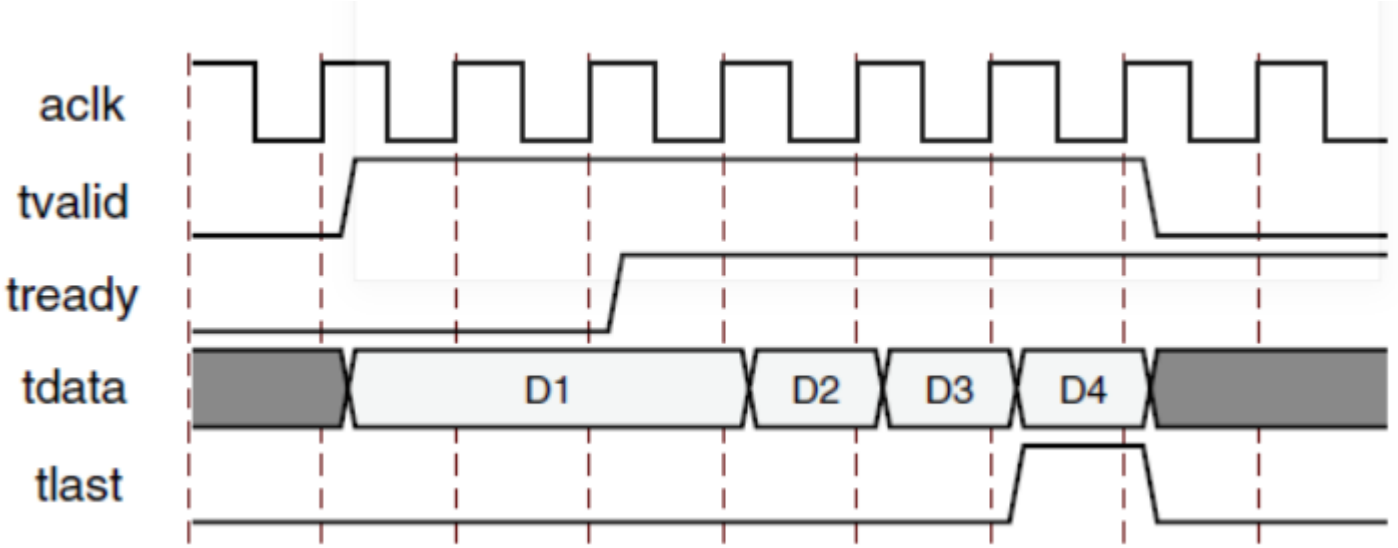
AXI4-Stream - AXIS

- Specify on input or output only , not on input/output – array, pointer
- Use mostly in dataflow between functions
- Multiple variables group into same stream (struct, DATA_PACK)



AXI4-Stream Transfer Protocol

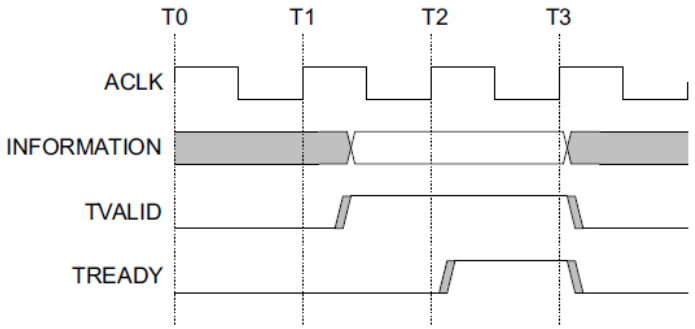




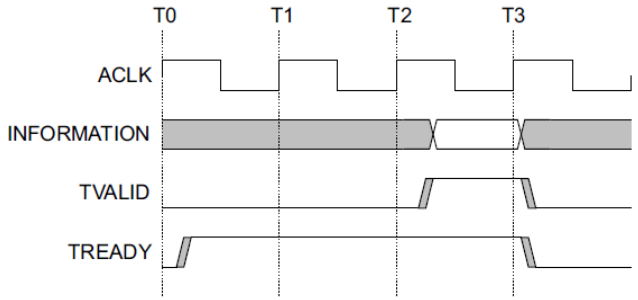
Data Transfer Handshake : TVALID, TREADY

- For a transfer to occur, both **TVALID** and **TREADY** must be asserted
- A Transmitter is not permitted to wait until **TREADY** is asserted before asserting **TVALID**
- Once **TVALID** is asserted, it must remain asserted until the handshake occurs
- A Receiver is permitted to wait for **TVALID** to be asserted before asserting **TREADY**

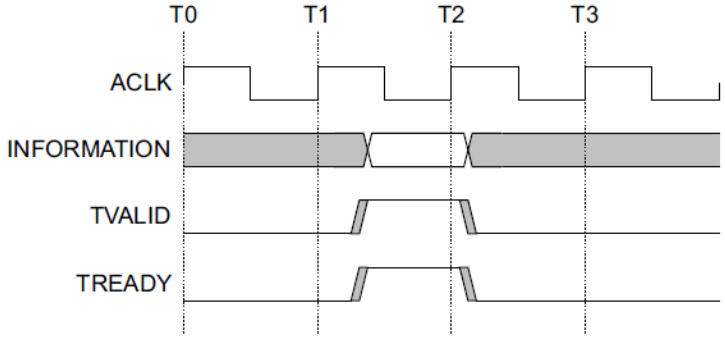
TVALID asserted before TREADY



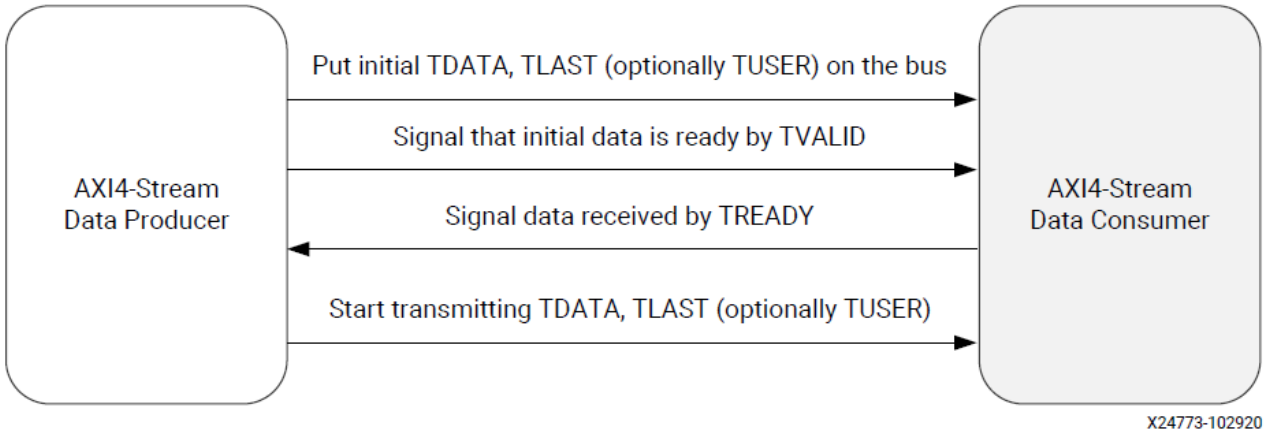
TREADY asserted before TVALID

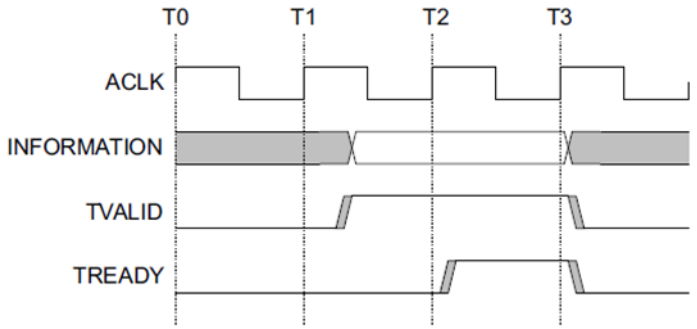


TVALID and TREADY asserted simultaneously



AXI4-Stream – HLS::AXIS






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

  static constexpr size_t NewWUser = (WUser == 0) ? 1 : WUser;
  static constexpr size_t NewWId = (WId == 0) ? 1 : WId;
  static constexpr size_t NewWDest = (WDest == 0) ? 1 : WDest;

 T                          data;
 ap_uint<bytewidth<T>> keep;
 ap_uint<bytewidth<T>> strb;
 ap_uint<NewWUser>     user;
 ap_uint<1>                last;
 ap_uint<NewWId>       id;
 ap_uint<NewWDest>     dest;
}
```

ap_axis<Wdata, WUser, WId, WDest>

hls::axis<ap_int<WData>, WUser, WId, WDest>

ap_axiu<WData, WUser, WId, WDest>

hls::axis<ap_uint<WData>, WUser, WId, WDest>

Stream for Dataflow : hls:stream<>

- Include <hls_stream.h>
- **hls::stream<Type, Depth>**
  - Type:
    - C++ native data type
    - HLS arbitrary precision type, e.g. ap_int<>
    - User-defined struct containing above types
  - Depth: depth of FIFO for co-simulation verification
- Used for top-level function arguments, and between functions
- Top interface can be
  - FIFO interface: ap_fifo (default) - support non-blocking behavior
  - Handshake interface: ap_hs
  - AXI4-Stream : axis
- Inside function – FIFO with depth = 2 (#pragma HLS STREAM depth = <int>), note: depth specify actual resource allocation.
- Use passed-by-reference to pass streams into and out of functions
- Only in C++ based designs

Stream Example

> Create using hls::stream or define the hls namespace

```
#include <ap_int.h>
#include <hls_stream.h>


typedef ap_uint<128> uint128_t;                // 128-bit user defined type

hls::stream<uint128_t> my_wide_stream;    // A stream declaration
```

```
#include <ap_int.h>
#include <hls_stream.h>
using namespace hls;                          // Use hls namespace

typedef ap_uint<128> uint128_t;          // 128-bit user defined type

stream<uint128_t> my_wide_stream;    // hls:: no longer required
```

> Blocking and Non-Block accesses supported

```
// Blocking Write
hls::stream<int> my_stream;

int src_var = 42;
my_stream.write(src_var);
// OR use:  my_stream << src_var;
```

```
// Blocking Read
hls::stream<int> my_stream;

int dst_var;
my_stream.read(dst_var);
// OR use: dst_var = my_stream.read();
```

```
hls::stream<int> my_stream;

int src_var = 42;
bool stream_full;

// Non-Blocking Write
if (my_stream.write_nb(src_var)) {
     // Perform standard operations
} else {
     // Write did not happen
}

// Full test
stream_full = my_stream.full();
```

```
hls::stream<int> my_stream;

int dst_var;
bool stream_empty;

// Non-Blocking Read
if (my_stream.read_nb(dst_var)) {
      // Perform standard operations
} else {
       // Read did not happen
}

// Empty test
fifo_empty = my_stream.empty();
```

Stream arrays and structs are not supported for RTL simulation at this time: must be verified manually.

e.g. hls::stream<uint8_t> chan[4])