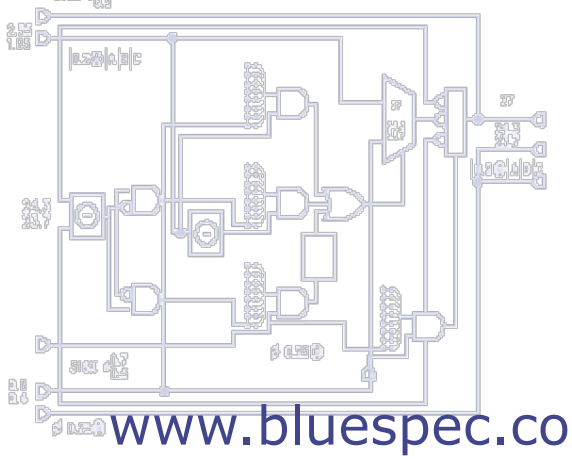
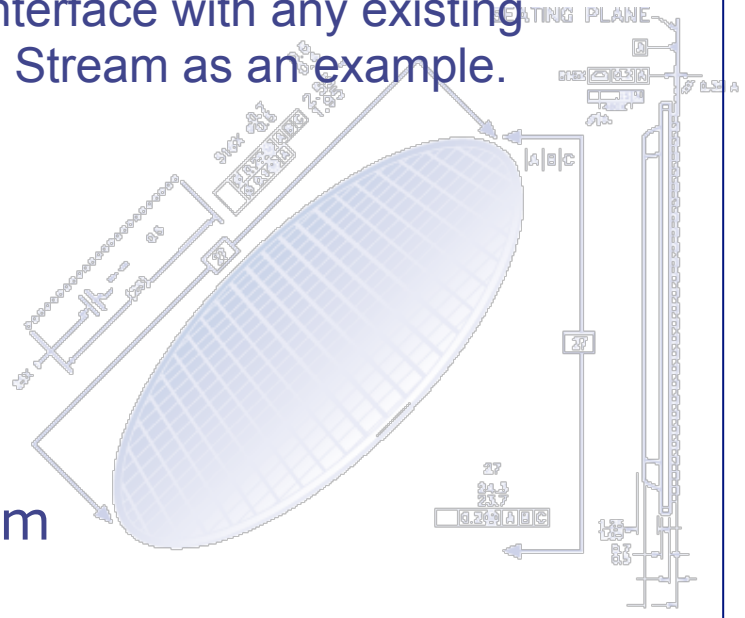
BSV Training

Eg09: Interfacing to AXI4 Stream

How BSV interfaces can be customized to interface with any existing RTL interface spec, using ARM AMBA AXI4 Stream as an example.





www.bluespec.com

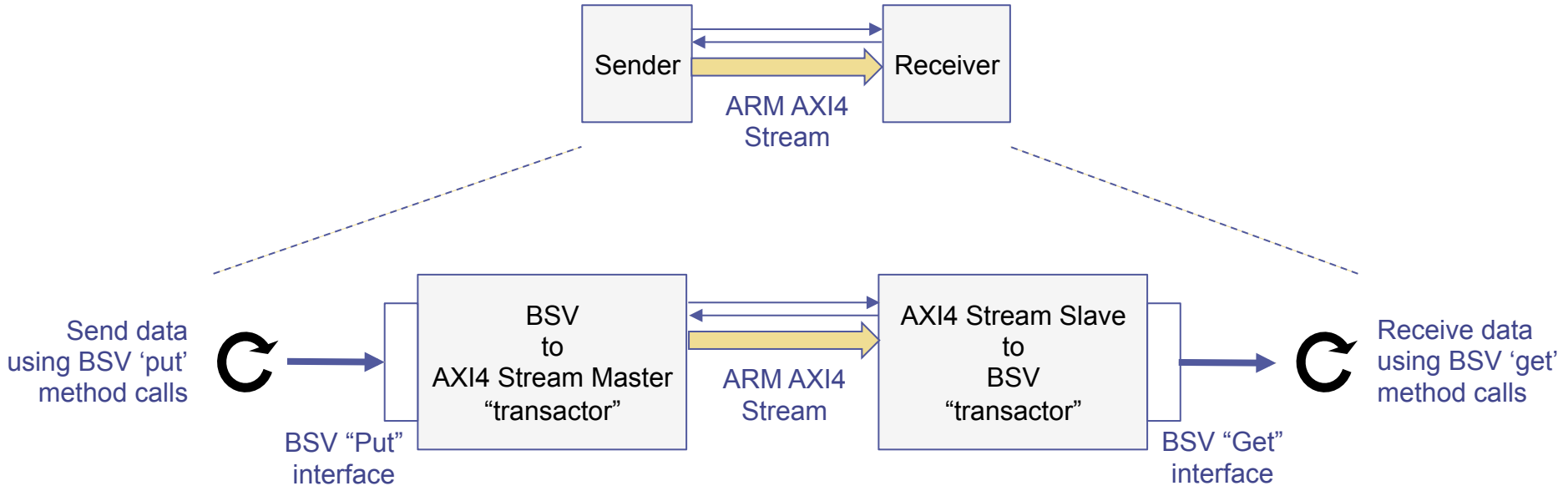
A note on training logistics

- This example concerns interfacing BSV to existing RTL blocks and protocols. As such, it may not be of interest when an entire design (including testbench) is done in BSV.

- This example can therefore be deferred/ postponed/ omitted if you are not currently trying to interface BSV to existing RTL, and studied later only if/ when that becomes a priority.

Before proceeding, please review the lecture: Lec_Interop_RTL

Eg09: Interfacing with existing HW and protocols



- The "AXI4 Stream" bus and protocol is a standard part of ARM's AMBA™ AXI™ interconnect family
  - Ref: ARM document "AMBA 4 AXI4-Stream Protocol", AXRM IHI 0051A (ID030510)
- In this example, we show the techniques for interfacing BSV to an existing bus and protocol such as AXI4 Stream.

The AXI4-Stream spec document

Ref: ARM document "AMBA 4 AXI4-Stream Protocol", AXRM IHI 0051A (ID030510)

AMBA® 4 AXI4-Stream Protocol

Version: 1.0

Specification

ARM®

2.1 Signal list

The interface signals are listed in Table 2-1. For additional information on these signals see further sections of this chapter.

Table 2-1 uses the following parameters to define the signal widths:

n Data bus width in bytes.

i **TID** width. Recommended maximum is 8-bits.

d **TDEST** width. Recommended maximum is 4-bits.

u **TUSER** width. Recommended number of bits is an integer multiple of the width of the interface in bytes.

Table 2-1 Interface signals list

Signal	Source	Description
ACLK	Clock source	The global clock signal. All signals are sampled on the rising edge of **ACLK**.
ARESETn	Reset source	The global reset signal. **ARESETn** is active-LOW.
TVALID	Master	**TVALID** indicates that the master is driving a valid transfer. A transfer takes place when both **TVALID** and **TREADY** are asserted.
TREADY	Slave	**TREADY** indicates that the slave can accept a transfer in the current cycle.
TDATA[(8n-1):0]	Master	**TDATA** is the primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes.
TSTRB[(n-1):0]	Master	**TSTRB** is the byte qualifier that indicates whether the content of the associated byte of **TDATA** is processed as a data byte or a position byte.
TKEEP[(n-1):0]	Master	**TKEEP** is the byte qualifier that indicates whether the content of the associated byte of **TDATA** is processed as part of the data stream. Associated bytes that have the **TKEEP** byte qualifier deasserted are null bytes and can be removed from the data stream.
TLAST	Master	**TLAST** indicates the boundary of a packet.
TID[(i-1):0]	Master	**TID** is the data stream identifier that indicates different streams of data.
TDEST[(d-1):0]	Master	**TDEST** provides routing information for the data stream.
TUSER[(u-1):0]	Master	**TUSER** is user defined sideband information that can be transmitted alongside the data stream.

bluespec

The AXI4-Stream spec document

Ref: ARM document "AMBA 4 AXI4-Stream Protocol", AXRM IHI 0051A (ID030510)

2.2 Transfer signaling

This section gives details of the handshake signaling and defines the relationship of the **TVALID** and **TREADY** handshake signals.

2.2.1 Handshake process

The **TVALID** and **TREADY** handshake determines when information is passed across the interface. A two-way flow control mechanism enables both the master and slave to control the rate at which the data and control information is transmitted across the interface. For a transfer to occur both the **TVALID** and **TREADY** signals must be asserted. Either **TVALID** or **TREADY** can be asserted first or both can be asserted in the same **ACLK** cycle.

A master 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 slave is permitted to wait for **TVALID** to be asserted before asserting the corresponding **TREADY**.

If a slave asserts **TREADY**, it is permitted to deassert **TREADY** before **TVALID** is asserted.

The following sections give examples of the handshake sequence.

TVALID before TREADY handshake

In Figure 2-1 the master presents the data and control information and asserts the **TVALID** signal HIGH. Once the master has asserted **TVALID**, the data or control information from the master must remain unchanged until the slave drives the **TREADY** signal HIGH, indicating that it can accept the data and control information. In this case, transfer takes place once the slave asserts **TREADY** HIGH. The arrow shows when the transfer occurs.

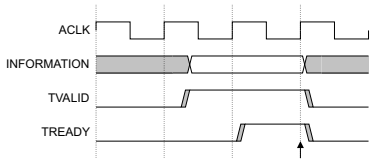


Figure 2-1 TVALID before TREADY handshake

TREADY before TVALID handshake

In Figure 2-2 the slave drives **TREADY** HIGH before the data and control information is valid. This indicates that the destination can accept the data and control information in a single cycle of **ACLK**. In this case, transfer takes place once the master asserts **TVALID** HIGH. The arrow shows when the transfer occurs.

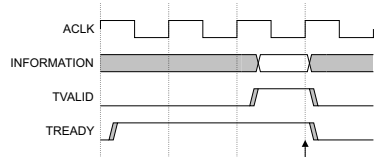


Figure 2-2 TREADY before TVALID handshake

TVALID with TREADY handshake

In Figure 2-3 the master asserts **TVALID** HIGH and the slave asserts **TREADY** HIGH in the same cycle of **ACLK**. In this case, transfer takes place in the same cycle as shown by the arrow.

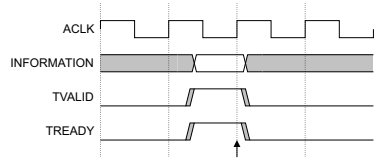
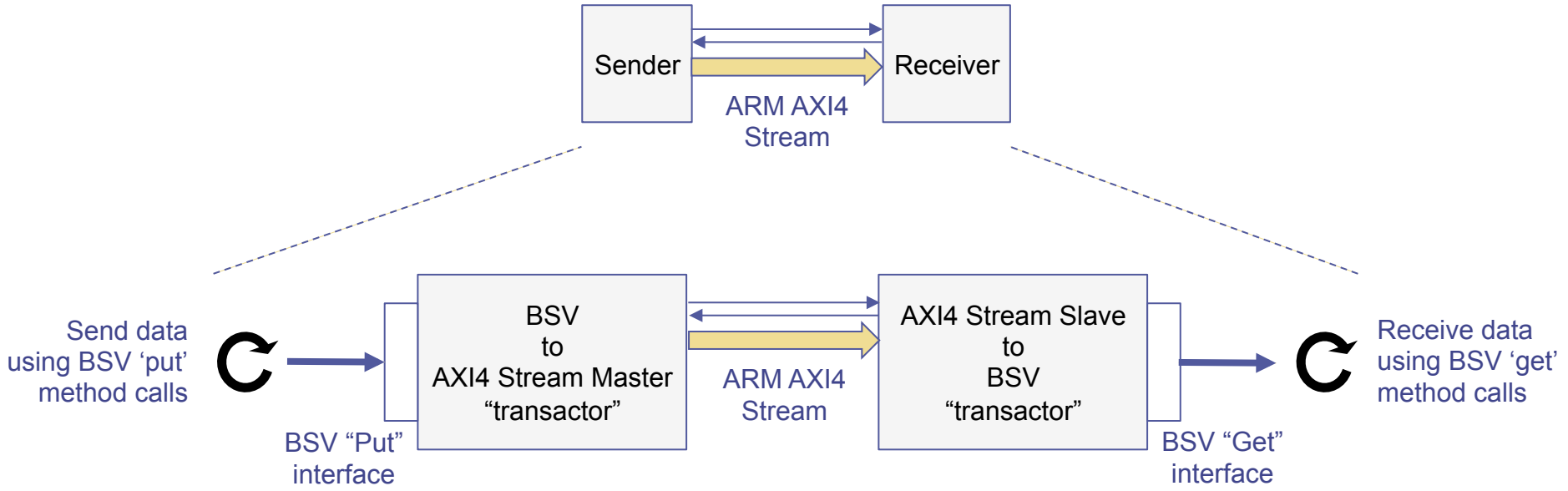


Figure 2-3 TVALID with TREADY handshake

Overview of this example



- We will define two transactors, as shown above. Each behaves like a FIFO.
- The testbench will "put" data into the master transactor using a standard BSV Put interface. Each datum will appear on the other interface, which is a standard AXI4 Stream interface.
- The slave transactor receives data on a standard AXI4 Stream interface. The testbench "gets" each datum using a standard BSV Get interface.
- In our testbench, we simply connect the AXI4 interfaces of the two transactors, like a simple "loopback" test. When used in an SoC, these interfaces would be connected to other IP blocks with AXI4 Stream interfaces.
- The testbench puts and gets data with varying pauses, to verify that the stream is properly flow-controlled (no data is lost or overwritten).

Directory: Eg09_AXI4_Stream/

Please examine the source file: src_BSV/AXI4_Stream.bsv

We first define a BSV struct describing the values carried on the datapath on an AXI4 stream connection:

```
typedef struct {
   Vector #(n, Bit #(8))  tDATA;
   Bit #(n)               tSTRB;
   Bit #(n)               tKEEP;
   Bool                   tLAST;
   Bit #(i)               tID;
   Bit #(d)               tDEST;
   Bit #(u)               tUSER;
   } AXI4_Stream_Payload #(numeric type n,    // width of TDATA (# of bytes)
                           numeric type i,    // width of TID (# of bits)
                           numeric type d,    // width of TDEST (# of bits)
                           numeric type u)    // width of TUSER (# of bits)
   deriving (Bits, FShow);
```

These definitions directly follow the AXI4 Stream specification document (see previous slides).

Directory: Eg09_AXI4_Stream/

Please examine the source file: src_BSV/AXI4_Stream.bsv

Next, we define the AXI4 Stream Master and Slave interfaces. E.g., the Master interface:

```
interface AXI4_Stream_Master_IFC #(numeric type n,  numeric type i,
                                   numeric type d,  numeric type u);
   (* prefix="" *)
   (* always_ready, always_enabled  *) method Action put ((* port="TREADY" *) Bool tREADY);

   (* always_ready, result="TVALID" *) method Bool                    tVALID;
   (* always_ready, result="TDATA"  *) method Vector #(n, Bit #(8)) tDATA;
   (* always_ready, result="TSTRB"  *) method Bit #(n)                tSTRB;
   (* always_ready, result="TKEEP"  *) method Bit #(n)                tKEEP;
   (* always_ready, result="TLAST"  *) method Bool                    tLAST;
   (* always_ready, result="TID"    *) method Bit #(i)                tID;
   (* always_ready, result="TDEST"  *) method Bit #(d)                tDEST;
   (* always_ready, result="TUSER"  *) method Bit #(u)                tUSER;
endinterface
```

We use BSV "attributes" (in "(*...*)" brackets) to control the corresponding port signal names when bsc synthesizes Verilog for a module with this interfaces. For details, see: BSV Reference Guide, Section 13

- "prefix=": overrides normal prefixing of Verilog signal port names with the BSV interface name
- "port=": overrides normal use of the BSV argument name for the corresponding Verilog input port
- "result=": overrides normal use of the BSV method name for the Verilog output port for the result
- "always_ready": overrides normal creation of a RDY output signal port for every BSV method, representing the method condition (*bsc* formally verifies this assertion in BSV code)
- "always_enabled": overrides normal creation of an ENA input signal port for Action and ActionValue methods, by which the environment signals when it is performing that action (*bsc* formally verifies this assertion in BSV code)

Bottom line: we precisely specify which signals appear in the Verilog, and their names.

Directory: Eg09_AXI4_Stream/

Please examine the source file: src_BSV/AXI4_Stream.bsv

For convenience, we also define 'mkConnection' for AXI4 Master and Stream interfaces:

```
instance Connectable #(AXI4_Stream_Master_IFC #(n,i,d,u),
                       AXI4_Stream_Slave_IFC #(n,i,d,u));
   module mkConnection #(AXI4_Stream_Master_IFC #(n,i,d,u) axim,
                         AXI4_Stream_Slave_IFC #(n,i,d,u) axis)
                        (Empty);
      (* fire_when_enabled, no_implicit_conditions *)
      rule rl_every_clock;
         axim.put (axis.tREADY);
         axis.put (axim.tVALID, axim.tDATA, axim.tSTRB, axim.tKEEP,
                   axim.tLAST, axim.tID, axim.tDEST, axim.tUSER);
      endrule
   endmodule
endinstance
```

We use more BSV attributes (see Reference Guide Section 13) to express the idea that this rule fires on every clock

- "fire_when_enabled": asserts that this rule does not conflict with any other rule, hence WILL_FIRE = CAN_FIRE.
- "no_implicit_conditions": asserts that none of the methods used in this rule have method conditions.
- Together, these imply that the rule fires in every clock.

Note: *bsc* formally verifies these assertions in BSV code

Bottom line: in the Verilog, we will merely have wires connecting corresponding AXI master and slave ports. No extra wires, no extra logic.

bluespec

Directory: Eg09_AXI4_Stream/

Please examine the source file: src_BSV/AXI4_Stream.bsv

Finally, we create the desired transactors that convert from BSV Put to AXI4 Stream master and from AXI4 Stream slave to BSV Get.

Each transactor interface just has an AXI4 Stream interface and a Get/Put interface as sub-interfaces:

```
interface AXI4_Stream_Master_Xactor_IFC #(numeric type n, numeric type I,
                                          numeric type d, numeric type u);
   (* prefix="" *)
   interface AXI4_Stream_Master_IFC #(n,i,d,u)       axi_side;
   interface Put #(AXI4_Stream_Payload #(n,i,d,u))   bsv_side;
endinterface
```

In the module implementing a transactor, there is only one interesting feature, the use of mkGFIFOF:

```
module mkAXI4_Stream_Master_Xactor (AXI4_Stream_Master_Xactor_IFC #(n,i,d,u));
   …
   FIFOF #(AXI4_Stream_Payload #(n,i,d,u)) fifo <- mkGFIFOF (False, True);
   …
endmodule
```

- A standard BSV FIFO is “guarded”: you cannot invoke the ‘enq’ method when it is full, and you cannot invoke the ‘first’ or ‘deq’ methods when it is empty. Thus, it is impossible to drop or overwrite data.
- The mkGFIFO is a “dangerous” FIFO that allows you to selectively disable these guards. Like RTL FIFOs, the onus is back on you to perform the full/empty checks before invoking corresponding enq/first/deq methods.
- In the above example, the “False” parameter requests that the “enq” guard is not disabled (remains guarded). This is because it is fed by a standard BSV Put interface.
- The “True” parameter requests that the “first/deq” guard is disabled (is unguarded). This is because here we are relying on the AXI4 Stream protocol, not BSV rule/method conditions, to manage flow control.
- The removal of these guards allows the module to satisfy the “always_enabled” and “always_ready” assertions shown earlier on the interface.

Directory: Eg09_AXI4_Stream/

Please examine the source file: src_BSV/Testbench.bsv

At the bottom of the file we use our standard trick to create separately synthesizable specializations of the polymorphic transactor modules. E.g.,:

```
(* synthesize *)
module mkAXI4_Stream_Master_Xactor_2_4_4_1 (AXI4_Stream_Master_Xactor_IFC #(2,4,4,1));
   let ifc <- mkAXI4_Stream_Master_Xactor;
   return ifc;
endmodule
```

The module mkTestbench itself is very straightforward.

The conditions on the rules rl_gen and rl_drain are to make them pause at different times, in order to exercise positive pressure (sender ready, receiver not ready) and negative pressure (vice versa).

Build and run the example

- In the Build directory, build and run using the ‘make’ commands, with Bluesim and/or with Verilog sim, as described earlier. Since the focus of this example is on interfacing with an RTL protocol, you should build and simulate in Verilog (even though it will also work in Bluesim.)
- Observe the inputs and outputs and verify that they are reasonable (all data is transmitted across the transactors without dropping, overwriting, replication, etc.)
- ‘make v_simulate’ should create a ‘dump.vcd’ waveform file. Display this waveform to observe the AXI4 Stream signals TREADY, TVALID and TDATA, and verify that it is following the AXI4 Stream protocol according to the specification. The file “waves_screenshot.tiff” is a picture of the waveform.
- Examine the generated Verilog files in the verilog/ directory and observe that AXI4 Stream interface signals are exactly per the specification.

Suggested exercises

- Connect the example transactors to a real IP block of your choice that has an AXI4 Stream interface.

- The BSV software distribution contains source code for libraries for interfacing to other ARM AMBA buses (more complex than AXI4 Stream), in the directories:
  - $BLUESPECDIR/BSVSource/Axi/
  - $BLUESPECDIR/BSVSource/AHB/

  Study these codes for more examples of interfacing BSV to existing RTL buses and protocols

- Create similar transactors for some other bus of your choice (e.g., OCP)

Summary

- Verilog ports are just a special case of BSV methods: using “always_ready” and “always_enabled” attributes, we can eliminate BSV’s default handshaking signals, leaving us with Verilog-like ports. We can then write explicit logic for any specified protocol or handshaking (using unguarded FIFOs, if necessary and appropriate).
- Using other attributes, we can control the exact names of interface ports in the generated Verilog
- Together, these techniques allow us to write BSV code that will interface precisely into any specified RTL interface.

bluespec

End