DP Streaming Module Description
(The ready signal of the streaming interface)

	Organisatie / Organization	Datum / Date
Auteur(s) / Author(s): Eric Kooistra	ASTRON	
Controle / Checked: Andre Gunst	ASTRON	
Goedkeuring / Approval: Andre Gunst	ASTRON	
Autorisatie / Authorisation: **Handtekening / Signature**	ASTRON	

Distribution list:

Group:	Others:
Andre Gunst Daniel van der Schuur Rajan Raj Thilak Arpad Szomoru (JIVE) Jonathan Hargreaves (JIVE) Salvatore Pirrucci (JIVE)	Gijs Schoonderbeek Sjouke Zwier

Document history:

Revision	Date	Author	Modification / Change
0.1	2010-04-6	Eric Kooistra	Draft.
0.2	2010-05-4	Eric Kooistra	Initial proper version.
0.3	2010-09-7	Eric Kooistra	Replaced dp_latency_hold by dp_pipeline. Added dp_hold_input. Added dp_shiftreg. Added dp_fifo_sc and dp_fifo_fill. Added dp_mux and dp_demux. Added description of the stream test benches.
0.4	2010-11-04	Eric Kooistra	Added in_channel support to dp_mux and dp_demux in addition to the channel bits that are used for port selection.
0.5	2011-01-10	Eric Kooistra	Major rework. Reorganised sections. Use record types for the streaming interfaces. Added DP streaming Module Description to the title. Added dp_split and dp_concat. Improved test bench scheme for streaming components.
1.0	2011-02-04	Eric Kooistra	Minor updates after review on Jan 20, 2011 with EK, DS, JH. Added reference to ST components in SOPC Builder tool.

Table of contents:

Terminology:

ADC	Analogue to Digital Convertor
DP	Data Path
DSP	Digital Signal Processing
DUT	Device Under Test
FIFO	First In First Out
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IO	Input Output
IP	Intellectual Property
Nof	Number of
PHY	Physical layer
RL	Ready Latency
RTL	Register Transfer Level
SISO	Source In Sink Out
SOPC	System On a Programmable Chip (Altera)
SOSI	Source Out Sink In
ST	Streaming
TB	Test Bench
UNB	https://svn.astron.nl/UniBoard_FP7/UniBoard/trunk/, the UniBoard SVN repository

References:

1. “Avalon Interface Specifications”, mnl_avalon_spec.pdf, www.altera.com
2. “Quartus II Handbook”, quartusii_handbook.pdf, www.altera.com
3. “SOPC Builder User Guide”, Dec 2010, ug_sopc_builder.pdf, www.altera.com
4. “Specification for module interfaces using VHDL records”, ASTRON-RP-380, Eric Kooistra
5. “Data Path Interface Description”, ASTRON-RP-394, Eric Kooistra

1 Introduction

1.1 Purpose

The Data Path (DP) module contains a group of general purpose streaming components. This document describes these DP components. The DP module is located in the UniBoard RadioNet FP7 SVN repository at $UNB/Firmware/modules/dp.

1.2 Streaming components

The streaming components in the DP module follow the Altera Avalon streaming (ST) interface definition [1, 2]. In the ST interface all data and control signals go from source to sink, except for one signal. This signal from sink to source is the ready signal that allows the sink to apply backpressure flow control on the source when it is not ready to receive new data. If all or some components in a stream do not need backpressure, then the ready signal is not used (fixed always active). The usage of the ready signal is not trivial. Therefore this document also explains the details of when the ready signal is needed and how it can be used.

The Altera SOPC Builder tool [2, 3] also provides similar streaming ST components. Some are similar to the DP components described in this document. E.g. the ST timing adapter in [3] is equivalent to the dp_latency_adapter in this document. However these ST components are only available through the SOPC Builder tool, so not via the MegaWizard and also not directly as plain VHDL.

2 The streaming interface

2.1 DP streaming interface definition

All streaming components in the DP module use the same streaming interface. The signals from source to sink are referred to as SOSI (Source Out Sink In) and the ready signal that goes from sink to source is referred to as SISO (Source In Sink Out). Table 1 shows the streaming interface signals. In this document the valid, sop and eop signals are also referred to as SOSI control signals. Similar the data, empty, channel and error signals are refered to as SOSI data signals.

Signal	Type	Description
ready	SISO	Backpressure flow control signal. The ready latency is RL >= 0.
data[71:0]	SOSI	Data word, byte [7:0] is the LSByte and bit [0] is the LSBit.
empty[7:0]	SOSI	Indicates the number of invalid bytes in the last data word marked by eop.
channel[7:0]	SOSI	Indicates the channel number for this valid cycle.
err[7:0]	SOSI	Error number, value 0 = OK.
valid	SOSI	Data valid strobe.
sop	SOSI	Start of packet strobe.
eop	SOSI	End of packet strobe.

Table 1: DP streaming interface signals

It is convenient to define a ‘t_dp_siso’ record for the SISO ready signal and a ‘t_dp_sosi’ record for the SOSI signals. Some DP components also use an unconstrained array of these records called ‘t_dp_siso_arr’ and ‘t_dp_sosi_arr’ at their interface to support multiple streaming ports. These VHDL record types and array of record types are defined in the dp_stream_pkg in the DP module and can be used for all ST interfaces because they support all fields and have sufficient width for the vector fields. See reference [4] for more explanation on this ST interface definition in the DP module.

2.2 Avalon interface definition of ready

The SISO ready signal is optional:

- If the sink does not output ready, then the sink is always ready to input new data.
- If the source does not input ready, then the source always outputs new data when it asserts valid.

If the ready signal is used then together with the ready signal a ready latency (RL) parameter needs to be defined. The RL can be 0, 1, 2, or more clock cycles. Regarding the ready latency there are two different cases [1]:

- RL = 0, then the source can assert valid on any cycle, and it must keep valid asserted until the sink has asserted ready.
- RL > 0, then the source is allowed to assert valid only RL cycles later for each asserted sink ready.

For RL > 0 the ready signal can also be interpreted as a request signal, when the sink asserts ready it requests the source to output new valid data if available. For RL = 0 the source pushes valid output data when available and the ready signal can then also be interpreted as an acknowledge signal that the sink has accepted the current valid data.

2.3 Flow control in digital signal processing

Flow control or backpressure can only be applied if a data stream allows data invalid gaps. The amount of backpressure that can be applied varies per stage in a digital processing stream. Near an ADC no backpressure is possible because every clock cycle carries valid data. After the ADC the digital samples can be processed further at a somewhat higher clock rate to create data invalid gaps in the stream. Another way in which data invalid gaps can be created is by first leading the ADC samples through a filter bank. The output of the filter bank still has data valid on every cycle, but subsequent processing can skip a few frequency subbands to create data invalid gaps. Together with sufficient buffering these gaps then allow a sink to apply backpressure when it is not ready yet and to catch up when it is. Applying backpressure can be needed for example for packetizing data to create some idle cycles to insert a packet header and tail or to align multiple input streams. In real time DSP applications the processing stages and backpressure cause latency in the data stream. This latency is acceptable as long as it does not accumulate out of bound. For offline processing of data from a storage facility the duration of backpressure can be unlimited if the entire data amount is stored and the processing time is still acceptable.

2.3.1 Alternatives to using the ready signal

Using dynamic backpressure via the ready signal is the preferred way to connect data path components, because it is a generally applicable way. An alternative to using ready can be to have the source use a throttled output data valid rate with a fixed duty cycle of e.g. 1 in 4 clock cycles that fits the sink. An alternative to using ready to cope with random variations in data valid rate is to use a FIFO buffer. Given a known average throughput and a known maximum data valid burst size the FIFO size can be selected such that overflow will not occur.

2.4 No ready signal

A sink without ready output effectively has its ready output asserted always. Therefore it can connect to sources with or without ready and with any RL. For a source without ready input it is different. A source without ready input, so no flow control, can only connect to a sink without ready output, so a sink that does not need backpressure.

2.5 Ready latency

For streaming interfaces that do use backpressure via the ready signal the RL of the sink and the source must be equal, because otherwise valid data may get lost or duplicated. Table 2 summarizes the ready and RL connection possibilities.

	Sink		
Source	**No ready**	**RL = 0**	**RL > 0**
No ready	Yes	-	-
RL = 0	Yes	Yes	-
RL > 0	Yes	-	Yes when equal

Table 2: Ready and RL connection combinations

A RL > 1 is not convenient if the sink needs to apply backpressure or if the source does not always have valid data available. The complication is then that the sink then may have to hold RL ≥ 2 number of valid data words after it has de-asserted ready, which is difficult to manage. Therefore if the sink needs to use backpressure then only RL = 0 and RL = 1 are suitable. Sometimes RL = 0 is useful, like with a show-ahead FIFO, because that allows a sink to observe the valid data without extracting it from the source already. However in most cases using RL = 1 is the preferred RL, because then the sink requests new data on this clock cycle and on the next clock cycle it will get it if the source has data available.

2.6 Backpressure through multiple components

2.6.1 Serie

If all components in a stream implement the ready signal on their sink and source ports, then the end of the stream can control the flow at the input. For e.g. input from an external link this flow control may even be passed on to the remote source via flow control frames. However for data path streaming such remote flow control is typically not needed, because there is no real time available to pause too long and then catch up again anyway.

Typically backpressure through multiple components is only needed for sections of a data path stream up to some buffer located several components up stream. Such a buffer can be a FIFO, e.g. to store and forward a packet. This buffer causes the down stream sink and up stream source to be less tightly coupled regarding the backpressure control via ready. The FIFO write full and read valid signals may be used to control the up stream backpressure and the down stream data valid.

2.6.2 Tree

A streaming tree consists of multiplexers and de-multiplexers.

A multiplexer passes several input streams via one output stream. The multiplexer typically works at frame level. For example the inputs can be different streams of packets and the output stream passes them on to an external link. While the multiplexer accepts the packet from one up stream source, it keeps sink out ready low for the other up stream sources. In this scheme each up stream source needs to take care of appropriate data buffering itself or the multiplexer can provide input FIFOs.

A de-multiplexer distributes a single stream over multiple output streams. For example the input can be a stream of packets with different identifiers and the output gets distributed based on the identifier. The streaming interface offers a channel field to identify different logical streams in one physical stream [1]. Each down stream sink only gets the valid data that matches to its assigned channel number. The de-multiplexer can work at data word level and does not need to be aware of frames.

3 Interface and design of the components

3.1 dp_validate – Validate the SOSI control signals

The dp_validate component asserts the SOSI control signals when they are valid. As extra feature the dp_validate can map the input error vector into bit 0 of the output error vector. Table 3 and Table 4 define the interface of dp_validate.

Generic	Type	Description
g_ready_latency	natural	RL of the stream
g_boolean_error	boolean	When true then vector-OR the snk_in.err into bit 0 of src_out.err, where '0' = OK and '1' = error. When false pass on snk_in.err.

Table 3: Parameters of dp_validate

Signal	Type	IO	Description
snk_out	t_dp_siso	OUT	SISO
snk_in	t_dp_sosi	IN	SOSI
src_in	t_dp_siso	IN	SISO
src_out	t_dp_sosi	OUT	SOSI

Table 4: Ports of dp_validate

Figure 1 shows the RTL schematic of the dp_validate component. All assignments are combinatorial, so the component introduces no clock latency.

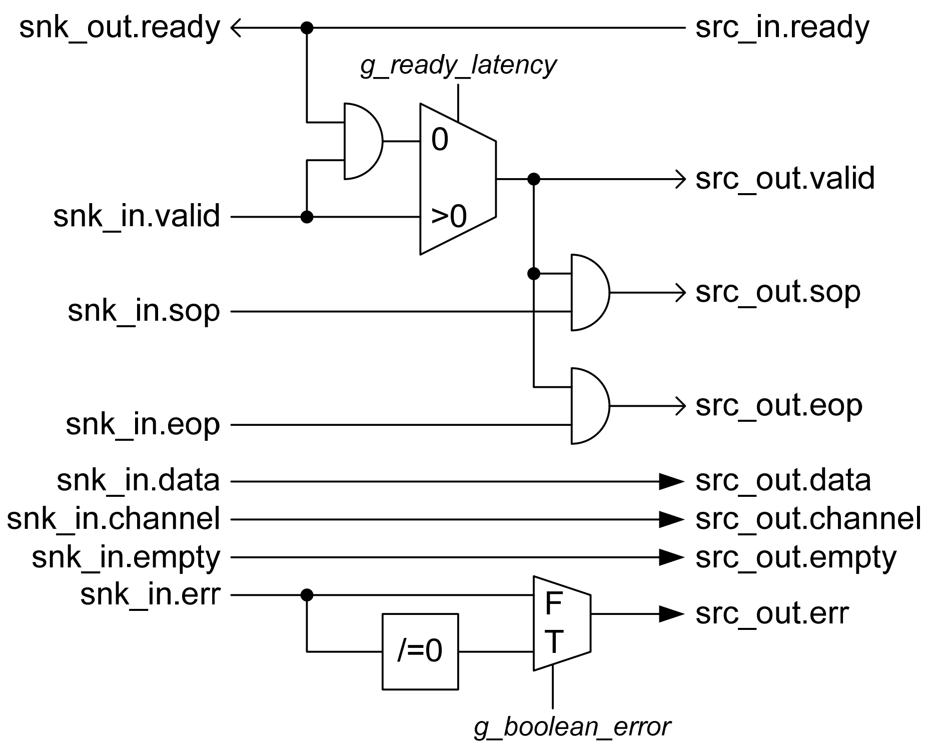

Figure 1: RTL schematic of dp_validate

If RL = 0 then the source data valid needs to be AND with the sink ready to strobe new valid data. If RL > 0 then the source data valid always strobes new valid data. According to the Avalon interface specification [1] the sop and eop are only valid when the valid is asserted. However it is not clear whether this statement is a requirement or a condition. Therefore this dp_validate component can be used to ensure that the sop and eop are only asserted when valid is active too.

3.2 dp_latency_increase – Ready latency increase

The dp_latency_increase component increases the ready latency (RL). This component is typically used to increase the ready latency from 0 to 1, because higher ready latencies are not useful. Table 5 and Table 6 define the interface of dp_latency_increase.

Generic	Type	Description
g_in_latency	natural	The input RL
g_incr_latency	natural	Increase the output RL by g_incr_latency compared to the input RL. Hence the output RL becomes g_in_latency + g_incr_latency

Table 5: Parameters of dp_latency_increase

Signal	IO	Type	Description
rst	IN	std_logic	Reset
clk	IN	std_logic	Clock
snk_out	OUT	t_dp_siso	SISO
snk_in	IN	t_dp_sosi	SOSI
src_in	IN	t_dp_siso	SISO
src_out	OUT	t_dp_sosi	SOSI

Table 6: Ports of dp_latency_increase

Figure 2 shows the RTL schematic of the dp_latency_increase component.

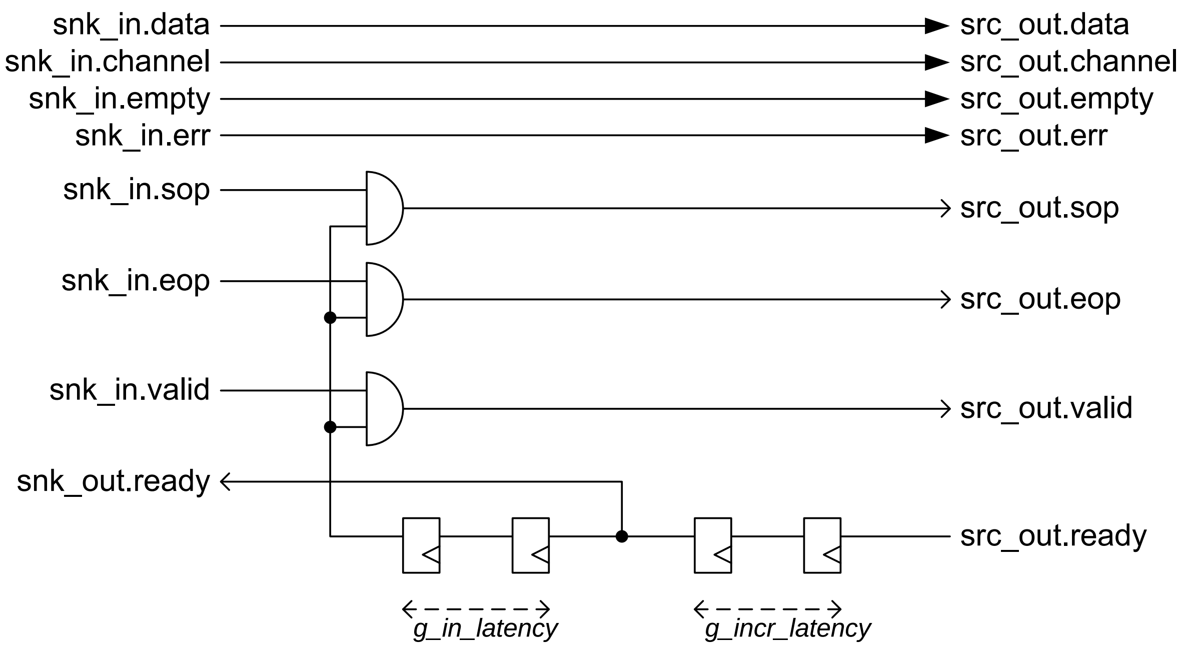

Figure 2: RTL schematic of dp_latency_increase

Incrementing the RL to match a smaller input RL to a larger output RL is easy. It merely requires some pipelining stages for the ready signal. The number of pipeline stages equals the ready latency difference. The output RL becomes g_in_latency + g_incr_latency. For increment RL = 0 the component implementation collapses to wires only. The SOSI control signals are internally AND with the delayed src_in.ready signal. This is only truely necessary if the input ready latency is 0, but it does not harm to do it also when the input ready latency > 0. The SOSI data signals are passed on as wires. Note that increasing the RL does register the ready signal, which can be useful to achieve timing closure.

3.3 dp_latency_adapter – Ready latency adapter

The dp_latency_adapter component adapts the input RL to the output RL. This component is typically used to decrease the ready latency to 0 or 1. Table 7 and Table 8 define the interface of dp_latency_adapter.

Generic	Type	Description
g_in_latency	natural	The input RL
g_out_latency	natural	The output RL

Table 7: Parameters of dp_latency_adapter

Signal	IO	Type	Description
rst	IN	std_logic	Reset
clk	IN	std_logic	Clock
snk_out	OUT	t_dp_siso	SISO
snk_in	IN	t_dp_sosi	SOSI
src_in	IN	t_dp_siso	SISO
src_out	OUT	t_dp_sosi	SOSI

Table 8: Ports of dp_latency_adapter

Decrementing the RL to match a larger input RL to a smaller output RL is difficult. It requires a FIFO buffer. Figure 3 shows the RTL schematic of ready latency decrease FIFO for adapting the input RL from 3 to 0.

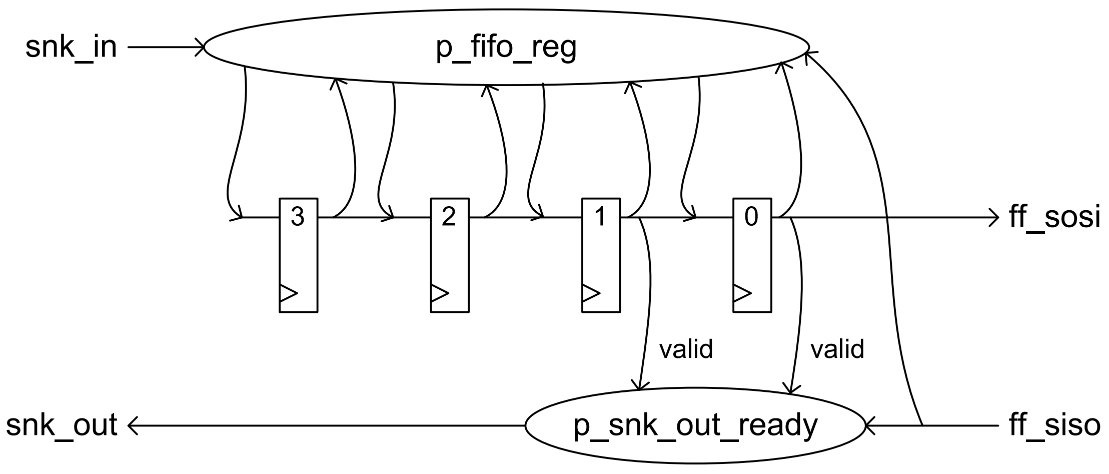

Figure 3: Ready latency decrease FIFO used in dp_latency_adapter (snk RL = 3, ff RL = 0)

If input RL > output RL then the input ready latency is first decreased to zero ready latency by means of a shift register FIFO. After that the dp_latency_increase component yields the required output ready latency. For input RL < output RL the component implementation collapses to using the dp_latency_increase component only. For input RL = output RL the component implementation collapses to wires only. If the output RL = 0 then the src_in.ready input connects combinatorially to the snk_out.ready output. If the output

RL > 0, then the dp_latency_adapter adds pipelining stages from the src_in.ready signal to the ff_siso.ready signal.

Note that the Avalon streaming interface [1] also provides a latency adapter component. It is called ST timing adapter, but it is only available from within SOPC Builder.

The RL decrease FIFO in Figure 3 consists of the 4 stage shift register that is of the type t_dp_sosi_arr. Register fifo_reg [0] contains the FIFO output with zero ready latency. Therefore the size of the FIFO needs to be input RL + 1 number of words. The process p_snk_out_ready for controlling the snk_out.ready depends on fifo_reg[0].valid and on fifo_reg[1].valid and on ff_siso.ready. Default snk_out.ready is '0'. The snk_out.ready can become '1' in two cases:

1. There must be free space for input RL + 1 number of data words in the FIFO, because when snk_out.ready is pulled low because ff_siso.ready went low then there can still arrive input RL number of new data with snk_in.valid asserted. The 1 extra word is needed to store the new data in case the current snk_in.valid is asserted. Checking for fifo_reg[0].valid is enough, there is no need to check that all fifo_reg[].valid bits are 0, because then they are 0 too.
2. If the ff_siso.ready is active, then 1 free space less is needed, because one ff_sosi will go out, so then snk_out.ready can be asserted also if there is free space for input RL number of data words in the FIFO. If this feature of checking ff_siso.ready is omitted then the throughput reduces by a factor 2. Checking for fifo_reg[1].valid is enough, there is no need to check that all fifo_reg[].valid bits are 0, because then they are 0 too.

The process p_fifo_reg for filling the FIFO depends on all fifo_reg[].valid and ff_siso.ready. Default the FIFO fifo_reg is shifted when ff_siso.ready is active. There is no need to explicitly check fifo_reg[].valid, because ff_siso.ready has zero ready latency, so it acts as a request for next data. The input data is put at the first available FIFO location dependent on fifo_reg[].valid and ff_siso.ready. There is no need to explicitly check snk_in.valid, because writing snk_in.valid = '0' to a free location is correct too.

3.3.1 Show-ahead FIFO

An interesting application example of a latency adapter is to make a show-ahead FIFO out of a default FIFO. A show-ahead FIFO is needed by applications that need to apply backpressure based on the contents of the data that are read. For a default FIFO the output data valid is typically asserted 1 clock cycle after the read request, so a source RL of 1. For a show-ahead FIFO the output data valid data is asserted as soon as there is data available so a source RL of 0.

Note that the Altera MegaWizard supports generating a show-ahead FIFO. The Xilinx CORE generator also supports generating a show-ahead FIFO, but there it is called ‘first word fall through’.

3.3.2 Ready pipeline

To achieve timing closure the SISO ready signal can be pipelined at the expense of some logic by first decreasing the RL with dp_latency_adapter and then increasing the RL with dp_latency_increase. Using the dp_latency_adapter to decrease the RL pipelines the SOSI data signals and using dp_latency_increase to get back to the original RL pipelines the SISO ready signal. Another way to break the combinatorial connection between the sink output ready signal with respect to the source input ready signal is to use a FIFO (e.g. dp_fifo_sc).

3.3.3 Adding backpressure support to components that do not support it

Figure 4 shows how a dp_latency_adapter can be used to add flow control to a series of components that do not support the backpressure SISO ready signal. The precondition is that the components do support the SOSI valid signal and that the upstream component again does support the SISO ready signal. The dp_latency_adapter compensates for the total latency of these components. If the latency is large then it becomes more efficient to implement the compensation by means of a FIFO using the dp_fifo_sc component.

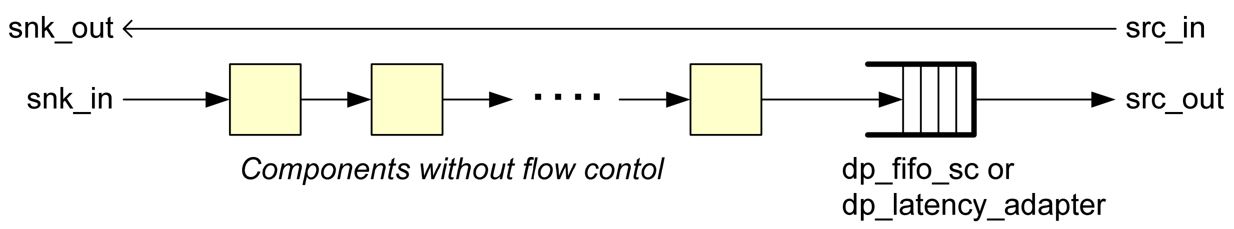

Figure 4: Providing flow control to SOSI only components

3.4 dp_hold_input – Input hold for stream processing

The purpose of the dp_hold_input component is to ease the implementation of a register stage for the SOSI signals in a streaming component. The dp_hold_input holds the valid input SOSI data when it can not be passed on to the output yet. Table 9 defines the interface of dp_hold_input. There are no parameters.

Signal	IO	Type	Description
rst	IN	std_logic	Reset
clk	IN	std_logic	Clock
snk_out	OUT	t_dp_siso	SISO
snk_in	IN	t_dp_sosi	SOSI
src_in	IN	t_dp_siso	SISO
next_src_out	OUT	t_dp_sosi	SOSI. Next src_out that is the same as the pending src_out, but only active if the src_in is ready.
pend_src_out	OUT	t_dp_sosi	SOSI. Pending src_out that may be due to a hold input or due to a new snk_in.
src_out_reg	IN	t_dp_sosi	SOSI. Feedback from external src_out register.

Table 9: Ports of dp_hold_input

The dp_hold_input function can also be interpreted as providing show-ahead capability for RL=1, somewhat similar to the show-ahead feature of RL=0. This ‘show-ahead’ information is available via pend_src_out.

Figure 5 shows the RTL schematic of the dp_hold_input component with RL = 1. In theory dp_hold_input could be made generic to also support RL > 1, however in practise RL > 1 is rarely used or adapted to RL = 1 first. The SOSI data signals are treated the same as shown for snk_in.data and the SOSI control signals are treated the same as shown for snk_in.valid.

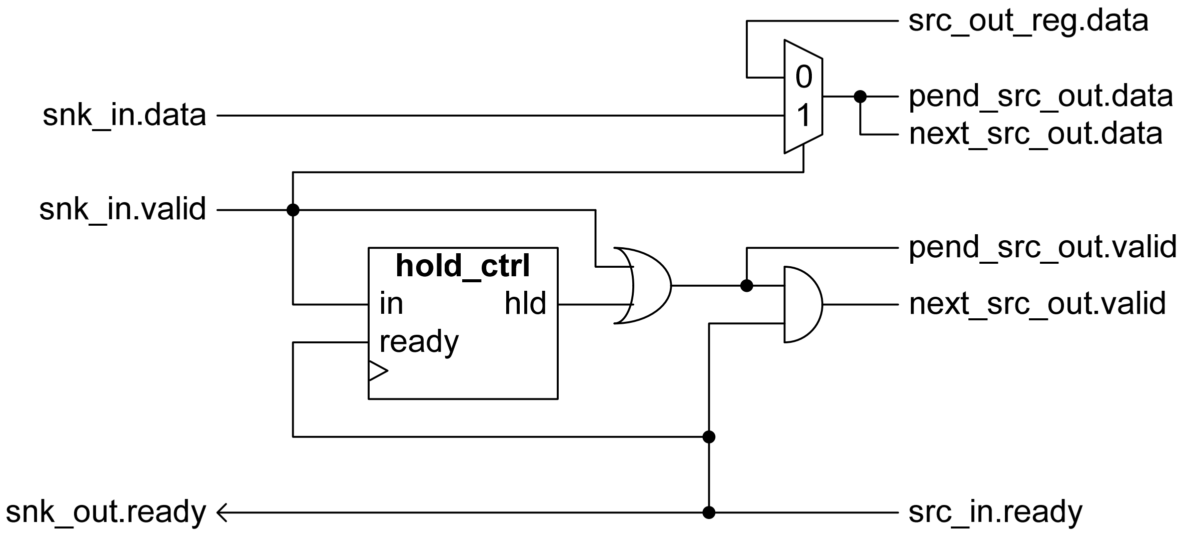

Figure 5: RTL schematic of dp_hold_input

Figure 6 shows the RTL schematic for the dp_hold_ctrl component that is used in dp_hold_input. The dp_hold_ctrl is used to hold an active SOSI control signal when src_in.ready goes low. The switch component that it uses is common_switch from the $UNB common module.

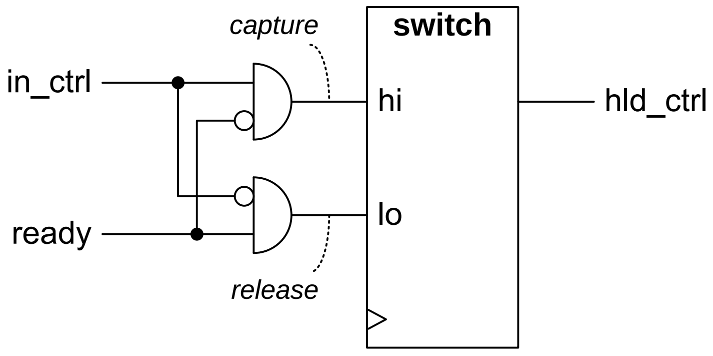

Figure 6: RTL schematic of dp_hold_ctrl

3.5 dp_pipeline – Data pipeline

The dp_pipeline component provides a single clock cycle delay of the SOSI signals. Pipelining the SOSI data signals provides a register stage which can be useful to ease timing closure. The SISO ready signal does not get registered by dp_pipeline. Table 10 defines the interface of dp_pipeline. There are no parameters.

Signal	IO	Type	Description
rst	IN	std_logic	Reset
clk	IN	std_logic	Clock
snk_out	OUT	t_dp_siso	SISO
snk_in	IN	t_dp_sosi	SOSI
src_in	IN	t_dp_siso	SISO
src_out	OUT	t_dp_sosi	SOSI

Table 10: Ports of dp_pipeline

Figure 7 shows the RTL schematic of dp_pipeline with RL = 1. The dp_pipeline is the simplest example of using dp_hold_input.

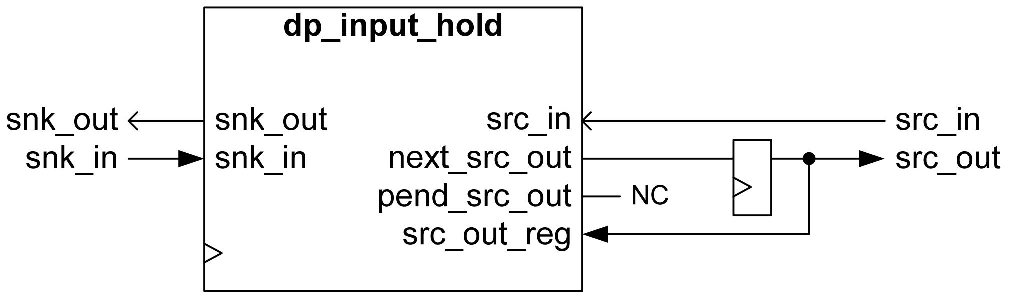

Figure 7: RTL schematic of dp_pipeline

Note that if the hld_ctrl output from dp_hold_ctrl would not be used for src_out.valid in dp_hold_input, then the dp_pipeline would still work. However the valid data that came when src_in.ready went low, would then only get pushed out by the next active snk_in.valid. Whereas using hld_ctrl ensures that it will get pushed out as soon as src_in.ready goes high again. This is typically necessary in case of packetized data where the eop of one packet should not have to wait for the active valid (sop) of a next packet to get pushed out.

3.6 dp_shiftreg – Shift register

The dp_shiftreg puts the valid input data through a SOSI shift register. Its purpose is to have access to the shift register contents, before output it. Table 11 and Table 12 define the interface of dp_shiftreg.

Generic	Type	Description
g_output_reg	boolean	When true use dp_pipeline to register the output SOSI.
g_flush_eop	boolean	When true then the shift register gets flushed if it contains an eop, provided that the output is ready. When false then no flush.
g_modify_support	boolean	When true then the input for all the words in de shift register can be modified via new_shiftreg_inputs. When false then the new_shiftreg_inputs are internally wired to the cur_shiftreg_inputs.
g_nof_words	natural	Number of SOSI words in the shift register (≥ 1).

Table 11: Parameters of dp_shiftreg

Signal	IO	Type	Description
rst	IN	std_logic	Reset
clk	IN	std_logic	Clock
snk_out	OUT	t_dp_siso	SISO
snk_in	IN	t_dp_sosi	SOSI
new_shiftreg_inputs	IN	t_dp_sosi_arr	SOSI array, range [0:g_nof_words-1]
cur_shiftreg_inputs	OUT	t_dp_sosi_arr	SOSI array, range [0:g_nof_words-1]
src_in	IN	t_dp_siso	SISO
src_out	OUT	t_dp_sosi	SOSI

Table 12: Ports of dp_shiftreg

Figure 8 shows the RTL schematic of the dp_shiftreg. When the shift register is full then every new valid input also causes a valid output. When g_flush_eop is true then the shift register gets flushed if it contains an eop, provided that the output is ready. Via cur_shiftreg_inputs the current shift register inputs are available externally for monitoring. When g_modify_support is true then the new shift register inputs can be modified via new_shiftreg_inputs.

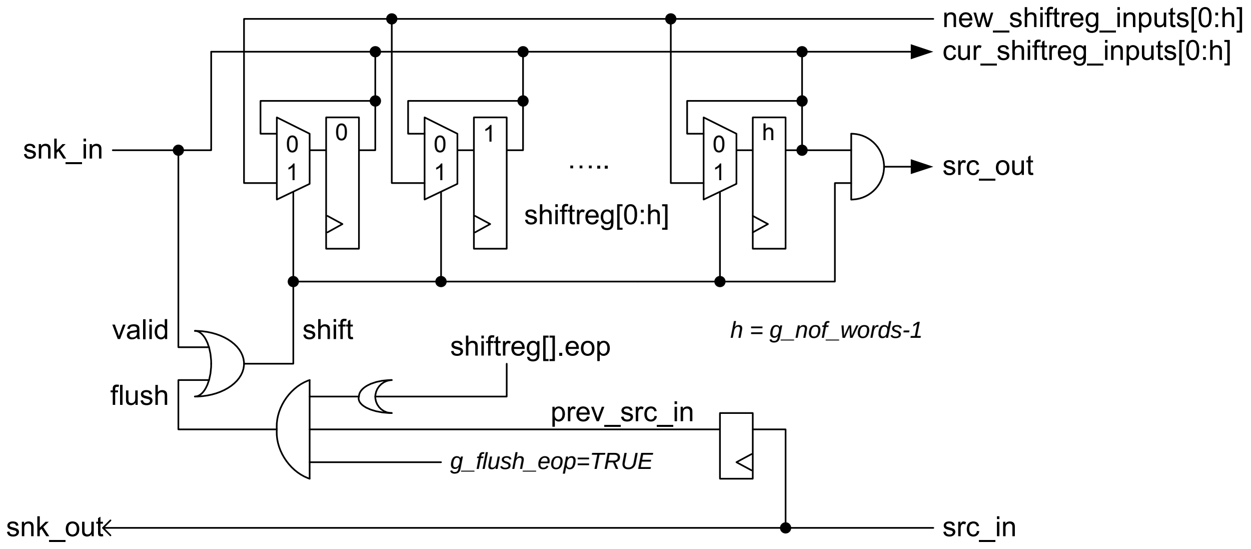

Figure 8: RTL schematic of dp_shiftreg

Note that dp_shiftreg with e.g. g_nof_words = 3 stages differs from using 3 dp_pipeline stages, because dp_shiftreg only outputs valid data after it has been filled or when it gets flushed, whereas dp_pipeline always outputs valid data when it can.

3.7 dp_fifo_sc – Streaming FIFO

The dp_fifo_sc component provides input ready control and output ready control to a common FIFO so that it can be used in a stream. The dp_fifo_sc operates in a single clock domain. Table 13 and Table 14 define the interface of dp_fifo_sc.

Generic	Type	Description
g_data_w	natural	Actual SOSI data width
g_empty_w	natural	Actual SOSI empty width
g_channel_w	natural	Actual SOSI channel width
g_error_w	natural	Actual SOSI error width
g_use_empty	boolean	When true pass empty via the FIFO. When false empty is not used.
g_use_channel	boolean	When true pass channel via the FIFO. When false channel is not used.
g_use_error	boolean	When true pass error via the FIFO. When false error is not used.
g_use_ctrl	boolean	Data without framing can use false to avoid passing the sop and eop bit fields via the FIFO.
g_fifo_size	natural	Number of SOSI words in the FIFO.
g_fifo_rl	natural	Output RL of the FIFO.

Table 13: Parameters of dp_fifo_sc

Signal	IO	Type	Description
rst	IN	std_logic	Reset
clk	IN	std_logic	Clock
snk_out	OUT	t_dp_siso	SISO
snk_in	IN	t_dp_sosi	SOSI
usedw	OUT	std_logic_vector	Monitor FIFO filling, range [ceil_log2(g_fifo_size)-1:0]
src_in	IN	t_dp_siso	SISO
src_out	OUT	t_dp_sosi	SOSI

Table 14: Ports of dp_fifo_sc

Figure 9 shows the RTL schematic of dp_fifo_sc. It uses the common_fifo_sc from the $UNB common module, so it uses the same synchronous clock for both write and read.

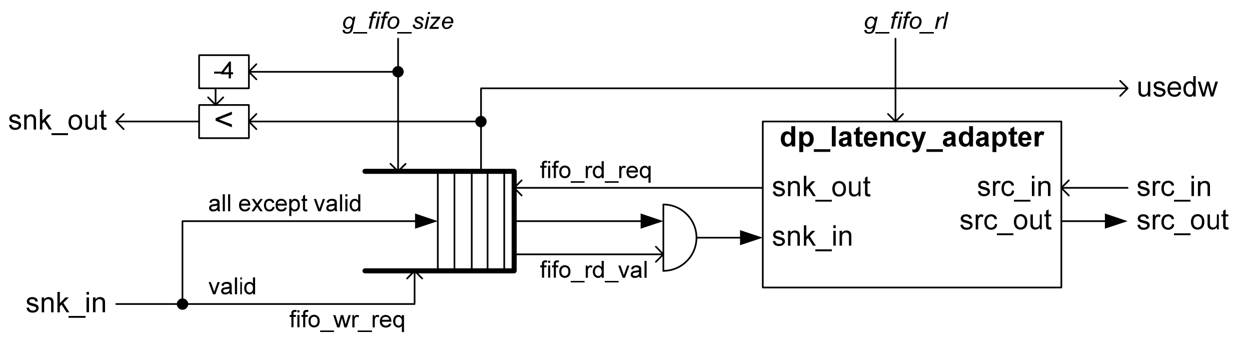

Figure 9: RTL schematic of dp_fifo_sc

The dp_fifo_sc passes on the SOSI fields by concatenating them into one vector. The SOSI valid is used as FIFO write request. The data field is always passed on via the FIFO. The other SOSI fields are only passed on when that field is enabled via the corresponding parameters in Table 13. To save memory it may be better to put the SOSI data fields empty, channel, error through a separate FIFO, because they only need one entry per frame. This FIFO can have a size equal to the maximum number of frames that is expected to be present in the data FIFO. The choice depends on the whether the aggregate data word still fits in a standard memory data word width and whether it is worth the design effort to save RAM.

Note that dp_fifo_sc makes that the src_in.ready and snk_out.ready are not combinatorially connected, so this can ease the timing closure for the ready signal.

3.8 dp_fifo_fill – Streaming FIFO with fill level

The dp_fifo_fill streaming FIFO starts outputting framed data when the output is ready and it has been filled sufficiently. The FIFO is filled per input frame, as defined by the snk_in.sop and then output until the snk_in.eop. Therefore the dp_fifo_fill is not suitable unframed data without sop and eop. Table 15 and Table 16 define the interface of dp_fifo_fill.

Generic	Type	Description
g_data_w	natural	Actual SOSI data width
g_empty_w	natural	Actual SOSI empty width
g_channel_w	natural	Actual SOSI channel width
g_error_w	natural	Actual SOSI error width
g_use_empty	boolean	When true pass empty via the FIFO. When false empty is not used.
g_use_channel	boolean	When true pass channel via the FIFO. When false channel is not used.
g_use_error	boolean	When true pass error via the FIFO. When false error is not used.
g_fifo_fill	natural	If the number of words in the FIFO is ≥ then this FIFO fill level, then the FIFO starts outputting data when the output is ready.
g_fifo_size	natural	Number of SOSI words in the FIFO.
g_fifo_rl	natural	Output RL of the FIFO.

Table 15: Parameters of dp_fifo_fill

Signal	IO	Type	Description
rst	IN	std_logic	Reset
clk	IN	std_logic	Clock
snk_out	OUT	t_dp_siso	SISO
snk_in	IN	t_dp_sosi	SOSI
src_in	IN	t_dp_siso	SISO
src_out	OUT	t_dp_sosi	SOSI

Table 16: Ports of dp_fifo_fill

Figure 10 shows RTL schematic of dp_fifo_fill.

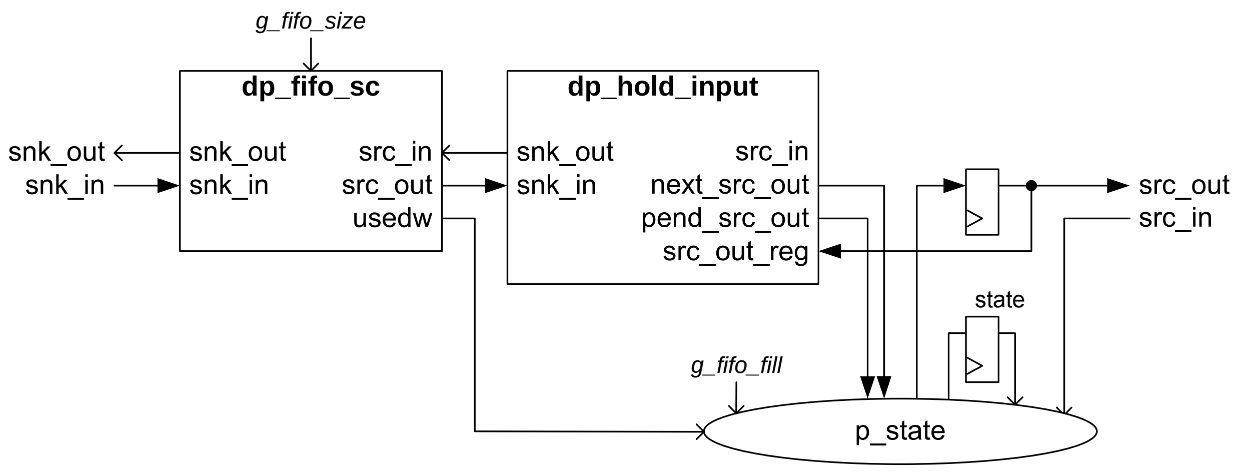

Figure 10: Stream FIFO with fill level

Given a fixed frame length, this dp_fifo_fill is useful when the snk_in.valid is throttled while the src_out.valid should not be inactive during a frame, i.e. from src_out.sop to src_out.eop. This is necessary e.g. for frame transport over a PHY link without separate data valid signal.

The architecture of dp_fifo_fill offers two implementations via g_fifo_rl. Dependent on g_fifo_rl it uses a show-ahead FIFO (RL = 0) or a normal FIFO (RL = 1). The show-ahead FIFO uses the dp_latency_adapter to get to RL = 0 internally. The normal FIFO architecture, shown in Figure 10, is preferred, because it uses less logic. It keeps the RL internally also at 1 and shows how to use dp_input_hold.

3.9 dp_mux – Multiplex multiple streams to one stream

The dp_mux multiplexes frames from one or more input streams to one output stream. Table 17 and Table 18 define the interface of dp_mux.

Generic	Type	Description
g_data_w	natural	Actual SOSI data width
g_empty_w	natural	Actual SOSI empty width
g_in_channel_w	natural	Actual SOSI channel width for the inputs
g_error_w	natural	Actual SOSI error width
g_use_empty	boolean	When true pass empty via the FIFO. When false empty is not used.
g_use_in_channel	boolean	When true pass channel via the FIFO. When false channel is not used.
g_use_error	boolean	When true pass error via the FIFO. When false error is not used.
g_nof_input	boolean	Number of multiplexer inputs, ≥ 1.
g_use_fifo	boolean	When true then the frames are buffered at the input, else there are no input FIFOs.
g_fifo_size	t_natural_arr	Number of SOSI words in the FIFO. Range [0:g_nof_input-1]
g_fifo_fill	t_natural_arr	If the number of words in the FIFO is ≥ then this FIFO fill level, then the FIFO starts outputting data when the output is ready. Range [0: g_nof_input-1]

Table 17: Parameters of dp_mux

Signal	IO	Type	Description
rst	IN	std_logic	Reset
clk	IN	std_logic	Clock
snk_out	OUT	t_dp_siso_arr	SISO array, range [0:g_nof_input-1]
snk_in	IN	t_dp_sosi_arr	SOSI array, range [0:g_nof_input-1]
src_in	IN	t_dp_siso	SISO
src_out	OUT	t_dp_sosi	SOSI

Table 18: Ports of dp_mux

Figure 11 shows the RTL schematic of the dp_mux with the optional input FIFOs. The dp_mux multiplexes frames from one or more (given by g_nof_input) input streams to one output stream. The frames are marked by snk_in[].sop and snk_in[].eop. The input frame that arrives first is passed on. Then if the next input has a frame pending then that is passed on. The output channel number reflects the input port number of the input that is being passed on. Using g_nof_input = 1 is allowed, but does cause an extra data invalid cycle after every frame due to the input select state machine. The low part of the src_out.channel has c_sel_w = log2(g_nof_input) number of bits and equals the input port number. The snk_in[].channel bits are shifted into the higher part of the src_out.channel. Hence the total effective output channel width becomes g_in_channel_w+c_sel_w when g_use_in_channel is true else c_sel_w. If g_use_fifo is true then the frames are buffered at the input, else the connecting inputs need to take care of that and then the FIFO parameters are not used.

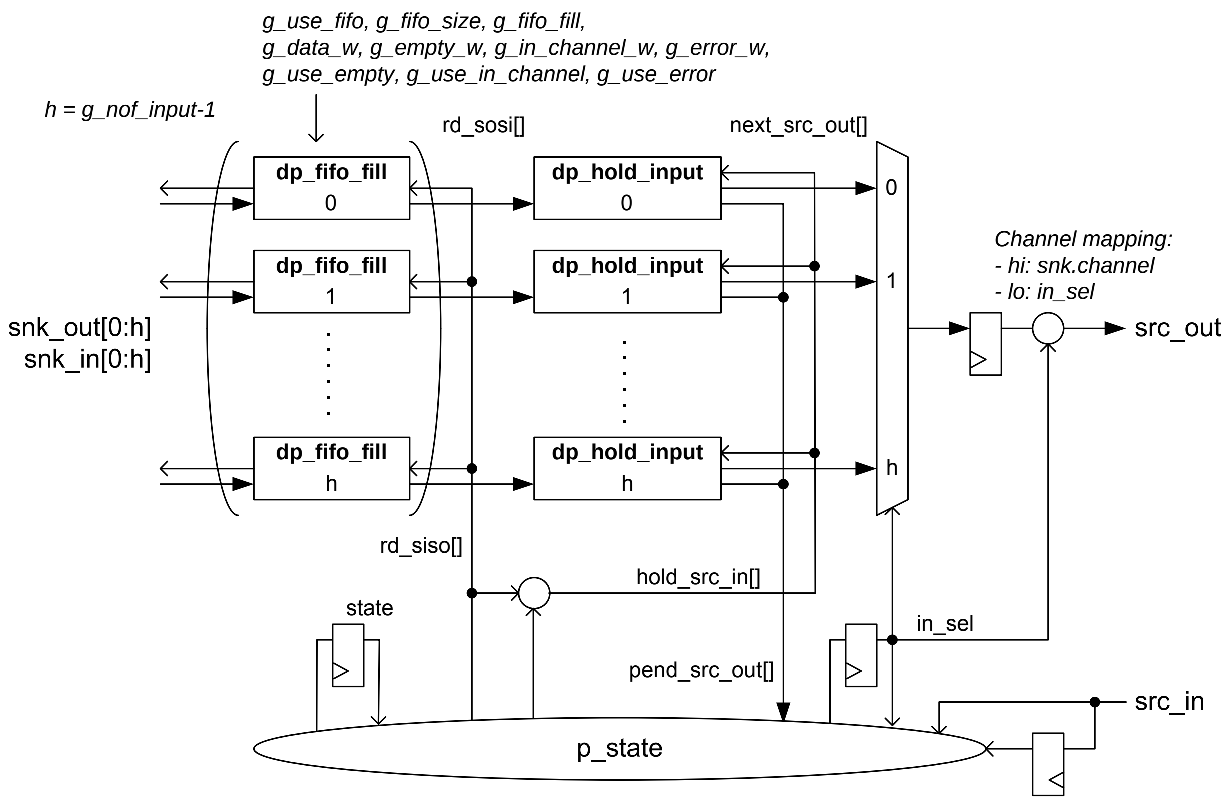

Figure 11: RTL schematic of dp_mux

3.10 dp_demux – De-multiplex one stream to multiple streams

The dp_demux de-multiplexes frames from one input stream to one or more output streams. The selection is based on the low part of the input channel field. The dp_mux and this dp_demux are complementary. The dp_mux adds the log2(g_nof_input) low bits of the channel field to represent this input port and this dp_demux removes the log2(g_nof_output) low bits from the channel field and uses these to select the output port. Table 19 and Table 20 define the interface of dp_demux.

Generic	Type	Description
g_nof_output	natural	Number of de-multiplexer outputs, ≥ 1.
g_combined	boolean	When true then all outputs must be ready for the dp_demux to be ready else only the output for the current channel needs to be ready.

Table 19: Parameters of dp_demux

Signal	IO	Type	Description
rst	IN	std_logic	Reset
clk	IN	std_logic	Clock
snk_out	OUT	t_dp_siso	SISO
snk_in	IN	t_dp_sosi	SOSI
src_in	IN	t_dp_siso_arr	SISO array, range [0:g_nof_output-1]
src_out	OUT	t_dp_sosi_arr	SOSI array, range [0:g_nof_output-1]

Table 20: Ports of dp_demux

The dp_demux component has two variants shown in Figure 12 and Figure 13. If g_combined is true as shown in Figure 12, then dp_mux is purely combinatorial and can only be ready when all output streams are ready. This restriction is due to the fact that the RL = 1, which implies that the input channel field and the output ready are not known in the same clock cycle. If g_combined = false as shown in Figure 13, then internally the dp_demux works at RL = 0, so then it can provide output dependent on each individual output ready signal.

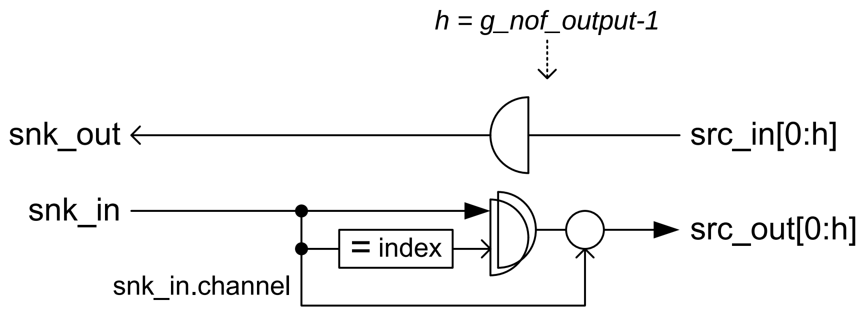

Figure 12: RTL schematic of dp_demux for g_combined = true

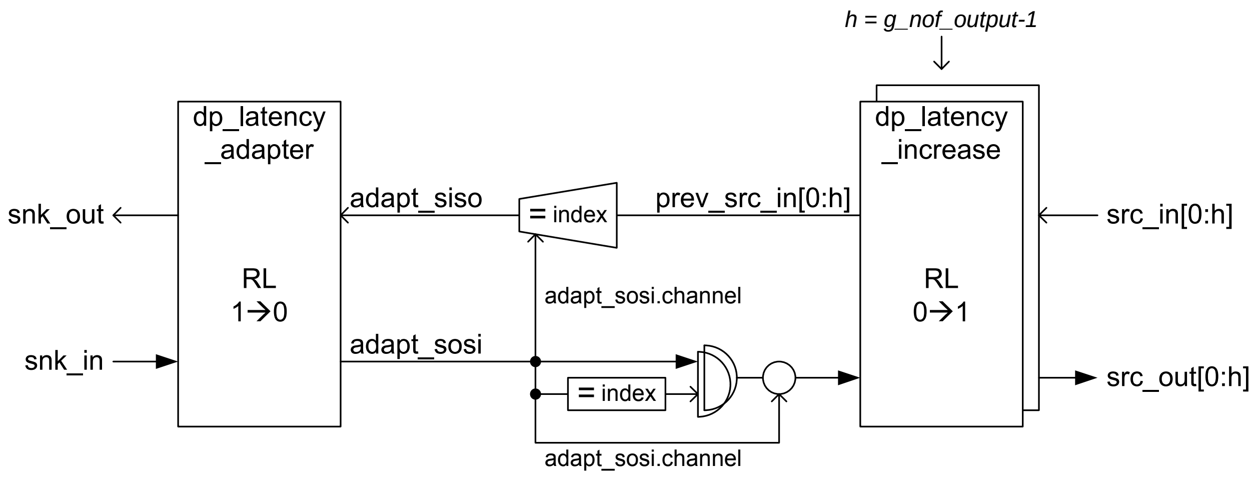

Figure 13: RTL schematic of dp_demux for g_combined = false

Note that the combinatorial variant of dp_demux requires the least logic. The registered variant of dp_demux may ease achieving timing closure, because both the SOSI data and de SISO ready signals get registered. Note that the dp_demux does not know about frames, it simple passes on the sop and eop. This differs from the dp_mux, because the dp_mux selects an input during the entire frame and then selects the next input.

3.11 dp_concat – Concatenate two frames into one frame

The dp_concat concatenates two input frames into one output frame. Table 21 and Table 22 define the interface of dp_concat.

Generic	Type	Description
g_data_w	natural	Actual SOSI data width, ≥ 1
g_symbol_w	natural	Symbol width, g_data_w/g_symbol_w must be an integer ≥ 1

Table 21: Parameters of dp_concat

Signal	IO	Type	Description
rst	IN	std_logic	Reset
clk	IN	std_logic	Clock
snk_out	OUT	t_dp_siso_arr	SISO array, range [0:1]
snk_in	IN	t_dp_sosi_arr	SOSI array, range [0:1]
src_in	IN	t_dp_siso	SISO
src_out	OUT	t_dp_sosi	SOSI

Table 22: Ports of dp_concat

Both frames must have the same data width and symbol width. The concatenation is done at symbol level and the output frame will have the aggregated empty. The concatenated frame gets the channel and error field of the tail frame from sink port 1. The RTL schematic of dp_concat consists of a dp_hold_input instance for each input and a state machine.

3.12 dp_split – Split one frame into two frames

The dp_split splits one input frame into two output frames, a head frame and a tail frame. The dp_concat and this dp_split are complementary. The number of symbols for the head frame is provided via a parameter or port. The remaining symbols are output to the tail frame. Table 23 and Table 24 define the interface of dp_concat.

Generic	Type	Description
g_data_w	natural	Actual SOSI data width, ≥ 1
g_symbol_w	natural	Symbol width, g_data_w/g_symbol_w must be an integer ≥ 1
g_nof_symbols	natrual	Number of symbols for the head frame, rest will go to the tail frame.

Table 23: Parameters of dp_split

Signal	IO	Type	Description
rst	IN	std_logic	Reset
clk	IN	std_logic	Clock
nof_symbols	IN	std_logic_vector	Range [ceil_log2(g_nof_symbols+1)-1:0]. Default value is g_nof_symbols,
snk_out	OUT	t_dp_siso	SISO
snk_in	IN	t_dp_sosi	SOSI
src_in	IN	t_dp_siso_arr	SISO array, range [0:1]. Port 0 for head and port 1 for tail.
src_out	OUT	t_dp_sosi_arr	SOSI array, range [0:1]. Port 0 for head and port 1 for tail.

Table 24: Ports of dp_split

The split is done at symbol level and the output frame will have the aggregated empty. The split head frame gets the current channel and error field value. The error field may be undefined because the true error information is not known yet as it is only available at the snk_in.eop. The split tail frame gets channel and error field from the input frame. If the nof_symbols input port is used, then g_nof_symbols must be set to fit the largest head frame and input nof_symbols. Otherwise if nof_symbols is fixed, then g_nof_symbols can be set to that value and the input port can be left not connected. If nof_symbols = 0, then the input frame is passed on entirely to the tail frame output. If nof_symbols ≥ number of input symbols then the input frame is passed on entirely to the head frame output. It is allowed to change nof_symbols during a frame split, because the new value only gets accepted when the split has finished. The RTL schematic of dp_concat consists of dp_hold_input and a state machine.

4 Verification of the components

4.1 Test bench package

Due to the feedback behaviour of the ready signal it is difficult to comprehend and prove the working of a stream component. Therefore it is essential to have a VHDL test bench with proper stimuli to verify that the DP components work OK. The DP module provides a VHDL package called tb_dp_pkg.vhd with procedures for stimuli and verification. Table 25 and Table 26 list all procedures and functions that are available in tb_dp_pkg.

Function	**Description**
proc_dp_gen_block_dat	Used to generate data frames in the tb_dp_packetizing test bench that is used to verify the packetizing components in the DP module [5]. The generated data frames contain counter data. The procedure does not support the ready signal, instead it supports output throttling. The data valid is throttled at an arbitrary rational rate during a certain block time. This defines a frame. The frame is then followed by a gap time. The procedure does not output sof and eof but it does output a sync pulse for every given number of frames.
proc_dp_cnt_dat	Output incrementing counter data for every asserted input strobe. No symbol support.
proc_dp_tx_data	Increments the RL of the input data to the specified ready latency ≥ 0.
proc_dp_tx_ctrl	Add data control strobe to the data when the data value equals c_offset + k * c_period, where k ≥ 0. Used to create a sop or an eop. By using the same c_period for sop and eop it is possible to mark frames.
proc_dp_sync_interval	Create a sync pulse with a given interval.
proc_dp_count_en	Stimuli for cnt_en that differ for every sync interval defined in Table 27. After the last stimuli interval it signals a done pulse.
proc_dp_stream_ready_latency	Handle stream ready signal. Output active when src is ready and in_en='1'
func_dp_data_init	Initialize the incrementing data per symbol. Uses big endian, so if c_data_w=32, c_symbol_w=8, init=3 then return 0x03040506
func_dp_data_incr	Increment the data per symbol. Use big endian, so if c_data_w=32, c_symbol_w=8 then 0x00010203 returns 0x04050607. The actual data width must be ≥ c_data_w, unused bits become 0 and c_data_w / c_symbol_w must be an integer.
proc_dp_gen_frame	Generate a frame with incrementing symbols. Uses: - proc_dp_stream_ready_latency - func_dp_data_init - func_dp_data_incr

Table 25: Stream source test bench functions in tb_dp_pkg

Function	Description
proc_dp_out_ready	Stimuli for out_ready that differ for every sync interval defined in Table 27.
proc_dp_verify_en	Uses an initial delay after which the verification is enabled. Alternatively it enables the verification after the first valid data (continuous mode) or it enables the after each sop and disables after each eop (frame mode).
proc_dp_verify_value	Verify the expected value. Used to check that a test has ran at all.
proc_dp_verify_data	Verify incrementing data by comparing with the previous data.
proc_dp_verify_symbols	Verify incrementing symbols in data by comparing with the previous data.
proc_dp_verify_data_empty	It does the same as proc_dp_verify_data, but for the last data word in a frame marked by the eop it only verifies the part that indicated by the stream empty signal. Furthermore it accepts an alternative last word value to check against. This feature is useful if the DUT replaces the last word in a frame by e.g. some known other word instead of the incremented counter value. Used to verify the eth_crc_ctrl component in the 1 Gb Ethernet module in $UNB.
proc_dp_verify_other_sosi	Suited to verify the empty, error, channel fields assuming that these are treated in the same way in parallel to the data field.
proc_dp_verify_valid	Verifies that the src_out.valid fits the required RL.
proc_dp_verify_ctrl	Verifies the data control strobing that was added with proc_dp_tx_ctrl. Can detect missing and unexpected output strobes.
proc_dp_stream_valid	Wait for asserted stream valid
proc_dp_stream_valid_sop	Wait for asserted stream valid AND sop
proc_dp_stream_valid_eop	Wait for asserted stream valid AND eop

Table 26: Stream sink test bench functions in tb_dp_pkg

The test bench stimulates the snk_in.valid and the src_in.ready of the device under test (DUT). The tb_dp_pkg provides two approaches for the stimuli:

- Using a list of stimuli intervals
- Using random stimuli

Figure 14 shows the test bench that uses proc_dp_count_en and proc_dp_out_ready to provide a predefined list of stimuli intervals.

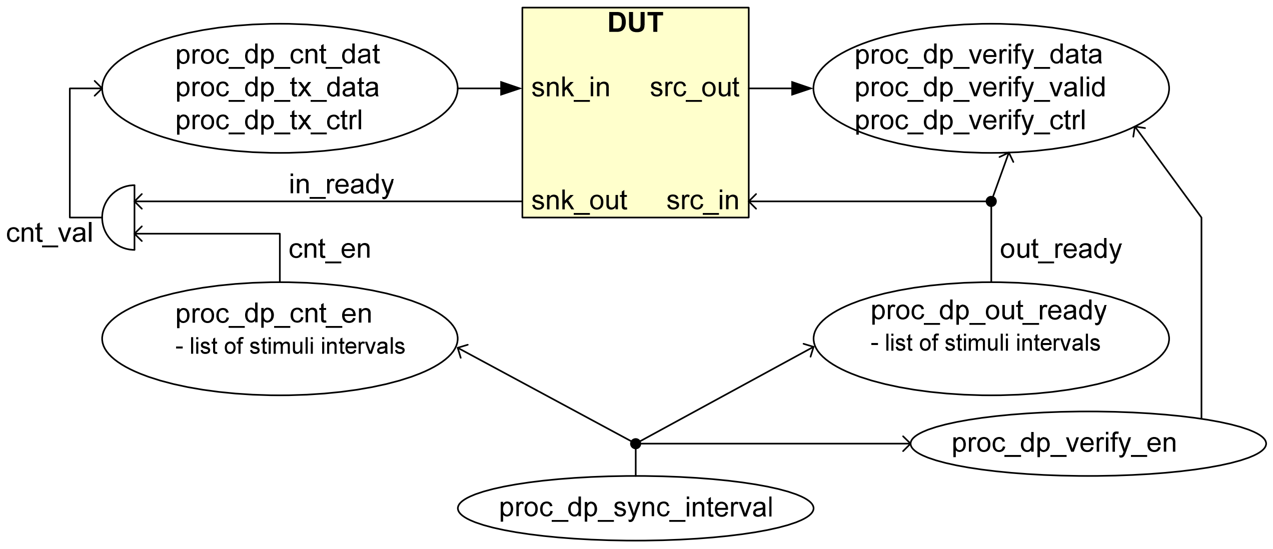

Figure 14: DP streaming test bench using a list of stimuli intervals

The subtests of each sync interval are named by an internal state signal. Table 27 lists the applied subtest schemes.

Interval	State	Description
-	s_idle	
1	s_both_active	Keep input enabled and output ready for every clock cycle.
2	s_pull_down_out_ready	Keep input enabled and make output ready low for 1, 2 and more clock cycles.
3	s_pull_down_cnt_en	Keep output ready active and disable input for 1, 2 and more clock cycles.
4	s_toggle_out_ready	Keep input enabled and toggle output ready in various patterns e.g. 1-1, 1-2, 2-1, 1-3, 3-1, 2-3, 3-2.
5	s_toggle_cnt_en	Keep output ready active and toggle input enable in various patterns e.g. 1-1, 1-2, 2-1, 1-3, 3-1, 2-3, 3-2.
6	s_toggle_both	Toggle both input enable and output ready in a sliding scheme of various patterns, whereby the duty cycle for input enable goes from active 100% to 0 and for output ready goes from 0 to 100%.
7	s_pulse_cnt_en	Keep output ready active and periodically enable the input for 1 in n clock cycles. The period n is gradually incremented from 2 to 15 and each period is applied 15 times.
8	s_chirp_out_ready	Keep input enabled and toggle output ready with increasing period from 1-1, 2-2, to 40-40.
9	s_random	Random toggle both input enable and output ready.
-	s_done	End

Table 27: DP stimuli subtest intervals

Figure 15 shows the test bench that use random stimuli and is the preferred approach for verifying a DUT. The test bench uses proc_dp_gen_frame to create the input frames and the verification procedures to verify the output frames. It can verify one or multiple symbols per data word. If g_random_control is false then snk_in.valid and src_in.ready are always active to ease initial debugging, else they are randomly asserted by means of func_diag_random from the $UNB DIAG module.

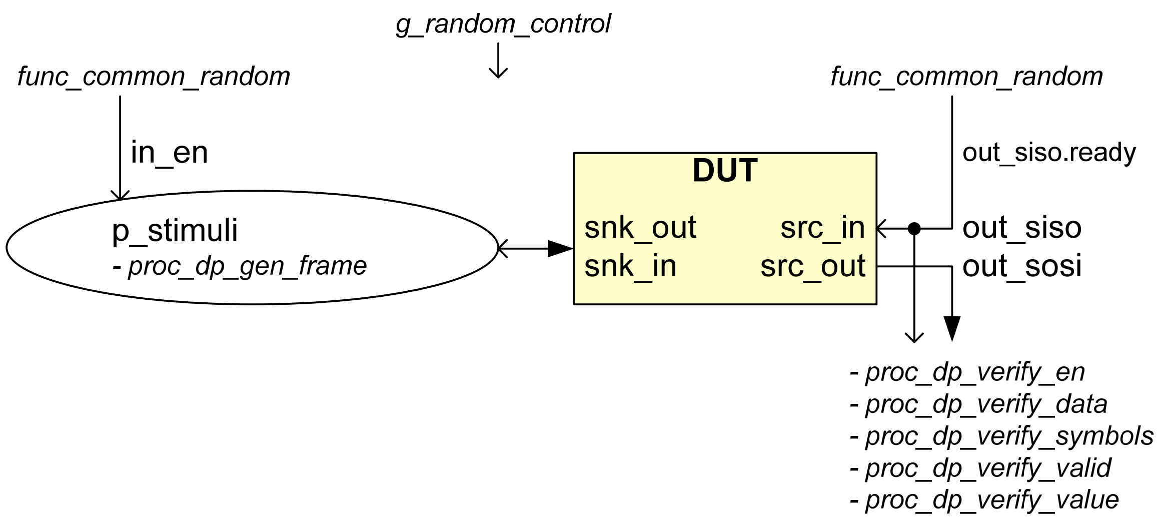

Figure 15: DP streaming test bench using random stimuli

4.2 Component test benches

4.2.1 tb_dp_latency_adapter

The tb_dp_latency_adapter test bench verifies the dp_latency_adapter component for several ready latency conversions using the list of stimuli intervals. The DUT consists of a series of dp_latency_adapter components. Each interface has arbitrary ready latency as shown in Figure 16.

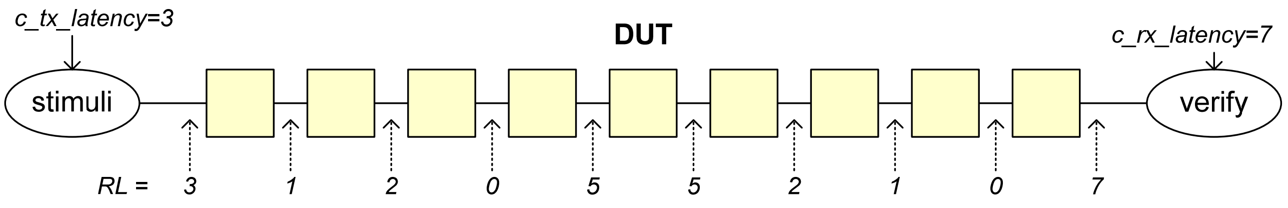

Figure 16: Block diagram of tb_dp_latency_adapter

The RL between the series of dp_latency_adapter components is chosen such that all relevant use cases occur:

- RL increase
- RL decrease to 0
- RL decrease, but still > 0
- RL stays the same

Note that the DUT also models the ready behaviour of a series of data path of streaming components, as described in section 2.6.1. However practical data path streaming components will typically use the same input and output ready latency for most interfaces.

4.2.2 tb_dp_pipeline

The tb_dp_pipeline test bench verifies the dp_pipeline component as DUT using the list of stimuli intervals.

4.2.3 tb_dp_shiftreg

The tb_dp_shiftreg test bench verifies the dp_shiftreg component as DUT using the list of stimuli intervals.

4.2.4 tb_dp_fifo_sc

The tb_dp_fifo_sc test bench verifies the dp_fifo_sc component as DUT using the list of stimuli intervals.

4.2.5 tb_dp_fifo_fill

The tb_dp_fifo_fill test bench verifies the dp_fifo_fill component as DUT using the list of stimuli intervals.

4.2.6 tb_dp_mux

The tb_dp_mux test bench verifies the dp_fifo_mux component as DUT using the list of stimuli intervals. The tb_dp_mux generates separate stimuli for each input of the dp_mux as shown in Figure 17. The output of the dp_mux consists of one stream whereby the channel number indicates the original input. The demux function in the test bench passes the output data on each of the verify instances (one per input). The data valid, sop and eop are forced to ‘0’ if the channel number does not match the verify instance number.

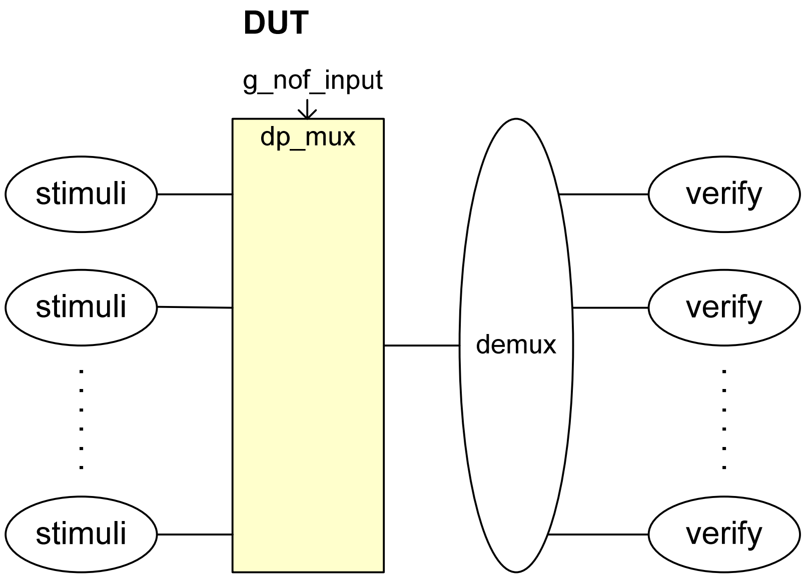

Figure 17: Block diagram of tb_dp_mux

4.2.7 tb_dp_demux

The tb_dp_demux test bench verifies the dp_fifo_demux component as DUT using the list of stimuli intervals. It works similar as tb_dp_mux and uses dp_mux as part of the test bench to create a multiplexed input stream with channel field as shown in Figure 18.

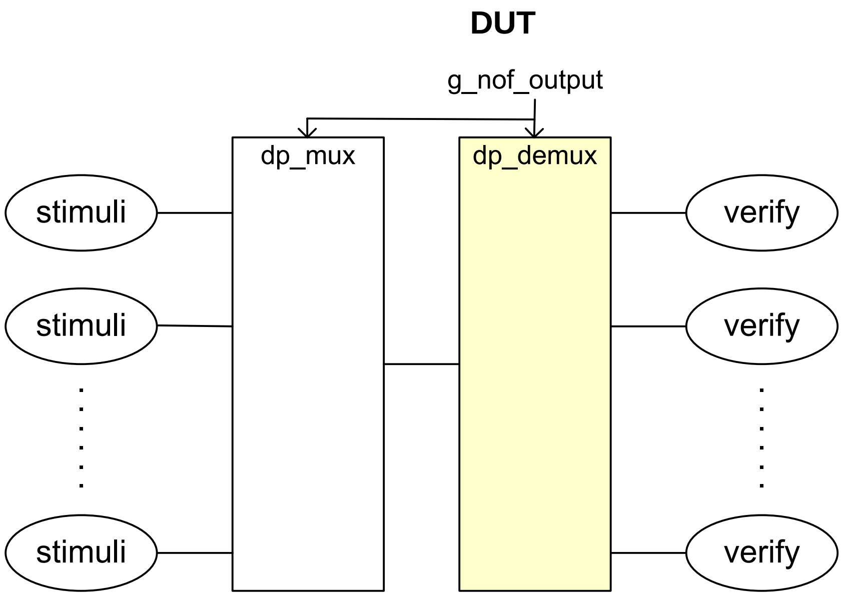

Figure 18: Block diagram of tb_dp_demux

Note that the test bench also models the ready behaviour of a tree of data path streaming components, as described in section 2.6.2.

4.2.8 tb_dp_concat

The tb_dp_concat test bench verifies the dp_concat component as DUT using random stimuli.

4.2.9 tb_dp_split

The tb_dp_split test bench verifies the dp_split component as DUT using random stimuli.

4.3 Multi test bench test benches

Some of the component test benches in section 4.2 have generics that allow the test bench to be run for different settings, e.g. the number of inputs for dp_mux or the number of symbols per dat word for dp_concat. By instantiating these test benches into another multi “tb_tb_” test bench these different parameter settings can be verified in parallel in a single simulation run. The following multi test benches are available:

- tb_tb_dp_fifo_sc.vhd
- tb_tb_dp_fifo_fill.vhd
- tb_tb_dp_mux.vhd
- tb_tb_dp_demux.vhd
- tb_tb_dp_concat.vhd
- tb_tb_dp_split.vhd

In addition to run all test benches for all DP components that use backpressure there is a multi “tb_tb_tb_” test bench:

- tb_tb_tb_dp_backpressure.vhd

This test bench runs instances of the multi test benches listed above and of:

- tb_dp_latency_adapter.vhd
- tb_dp_pipeline.vhd
- tb_dp_shiftreg.vhd

Hence the tb_tb_tb_dp_backpressure test bench is used for regression test of these streaming components in the DP module.