

UniBoard Beamformer module for APERTIF

	Organisatie / Organization	Datum / Date
Auteur(s) / Author(s): Harm Jan Pepping	ASTRON	24 February 2012
Controle / Checked: Eric Kooistra	ASTRON	
Goedkeuring / Approval: Andre Gunst	ASTRON	
Autorisatie / Authorisation: Handtekening / Signature Andre Gunst	ASTRON	

© ASTRON 2011
All rights are reserved. Reproduction in whole or in part is prohibited without written consent of the copyright owner.

UniBoard

Doc.nr.: ASTRON-RP-1291
Rev.: 0.4
Date: 04-04-2012
Class.: Public

Distribution list:

Group:	Others:
Andre Gunst Eric Kooistra Daniel van der Schuur	Gijs Schoonderbeek Sjouke Zwier Harro Verkouter (JIVE) Jonathan Hargreaves (JIVE) Salvatore Pirruccio (JIVE)

Document history:

Revision	Date	Author	Modification / Change
0.1	2012-01-30	Harm Jan Pepping	Creation
0.2	2012-03-07	Harm Jan Pepping	Processed comments of Eric. Moved chapter about the diag_block_gen to separate document (ASTRON-RP-1313).
0.3	2012-03-13	Harm Jan Pepping	Finished the implementation of the bf_switch.
0.4	2012-04-04	Harm Jan Pepping	Added the verification and validation chapters.

Table of contents:

1	Introduction.....	6
1.1	Purpose	6
1.2	Module overview.....	6
1.3	Mode of operation.....	6
2	Firmware interface.....	8
2.1	Clock domains	8
2.2	Parameters	8
2.3	Interface signals	9
2.3.1	IN_SOSI_ARR interface	10
2.3.2	OUT_RAW_SOSI_ARR and OUT_QUA_SOSI_ARR interfaces.....	11
2.3.3	MM_WEIGHT_MOSI interface	11
2.3.4	MM_BST_MOSI interface.....	11
2.3.5	MM_OFFSETS_MOSI interface.....	12
2.3.6	Clocks and resets	12
3	Software interface	13
3.1	Offsets span.....	13
3.2	Weight factors span.....	14
3.3	Beamlet statistics span.....	15
4	Module Design	17
4.1	Algorithm.....	17
4.2	Architecture.....	17
4.2.1	bf module	17
4.2.2	bf_switch.....	18
4.2.3	bf_unit	18
5	Implementation.....	20
5.1	bf_switch.....	20
5.1.1	Input format.....	20
5.1.2	Output format.....	20
5.1.3	Architecture: bf_switch_a_direct.....	21
5.1.4	Architecture: bf_switch_a_sort_distribute.....	22
5.2	bf_unit.....	27
5.2.1	bf_unit first stage	27
5.2.2	bf_unit second stage	28
5.2.3	bf_unit control process	30
5.2.4	Multiplexing mm interfaces	32
5.3	Bitwidths, bitgrowth and quantization	33
6	Verification.....	34
6.1	tb_bf_unit.....	34
6.1.1	fifo overflow and underflow check	34
6.1.2	p_init_offsets_register	34
6.1.3	p_weight_memory_write.....	34
6.1.4	Stimuli (input_data).....	34
6.1.5	p_verification.....	35
6.1.6	p_tester	35
6.2	tb_bf.....	35
6.2.1	p_init_offsets_register	35
6.2.2	p_init_weight_memory.....	35
6.2.3	gen_input_streams	36

6.2.4	p_read_bst_memory.....	36
6.2.5	gen_verification.....	36
6.2.6	gen_testers.....	36
7	FN_BF Reference design.....	37
7.1	Design.....	37
7.2	Verification.....	37
7.2.1	tb_node_fn_bf.....	37
7.2.2	tb_fn_bf.....	38
7.3	Software.....	38
8	Synthesis and Place & Route.....	39
8.1	Resources and F_{max}	39
8.2	Timing optimizations.....	39
8.2.1	Pipeline stages.....	39
8.2.2	Optimizing switch architecture.....	39
8.2.3	Synthesis constraints.....	39
8.2.4	LogicLock Regions.....	40
9	Validation.....	41
9.1	Python.....	41
9.1.1	pi_bf_bf.py.....	41
9.1.2	tc_pi_fn_bf.py.....	41
9.1.3	tc_pi_fn_bf.py.....	41
10	Appendix – list of files.....	42
10.1	Firmware VHDL.....	42
10.2	Testbench.....	42
10.3	Software.....	42

Terminology:

Beamlet	Beamformed subband
BF	Beamformer
DIAG	Diagnostics (VHDL module)
DP	Data Path (VHDL module)
DSP	Digital Signal Processing
DUT	Device Under Test
EOP	Start Of Packet
FIFO	First In First Out
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IO	Input Output
MISO	Master In Slave Out
MM	Memory-Mapped
MOSI	Master Out Slave In
Nof	Number of
RAM	Random Access Memory
Signal Path	Time series signal
SISO	Source In Sink Out
SOP	Start Of Packet
SOPC	System On a Programmable Chip (Altera)
SOSI	Source Out Sink In
SRC	Source
ST	Streaming
Subband	Frequency signal

References:

1. 'DP Streaming Module Description', ASTRON-RP-382, Eric Kooistra
2. 'Detailed Design of the Digital Beamformer System for Apertif', ASTRON-RP-413, G. Schoonderbeek, A. Gunst, E. Kooistra
3. Digital Beamformer Module for APERTIF, ASTRON-RP-517, H.J. Pepping
4. \$UNB/Firmware/modules/common/
5. \$UNB/Firmware/modules/Lofar/st/
6. 'DIAG Module Description', ASTRON-RP-131, H.J. Pepping, E.Kooistra
7. \$UNB/Software/python/peripherals/
8. \$UNB/Firmware/dsp/bf/tb/vhdl/
9. \$UNB/Firmware/designs/fn_bf/
10. \$UNB/Firmware/software/apps/fn_bf/
11. 'Timing Closure Methodology for Advanced FPGA Designs', AN 584, www.altera.com

1 Introduction

1.1 Purpose

The bf module performs a beamformer algorithm on a defined set of input streams where each stream can contain multiple signals. The output consists of one or more streams that contain series of beamlets. The beamformer weight factors can be set via a memory mapped interface. For each output stream a beamlet statistic unit is included and can be read out via the memory mapped interface as well.

1.2 Module overview

A high level overview of the bf module is shown in Figure 1. The bf module has a variable number of input streams where each stream contains subband samples from one or multiple antenna inputs. The incoming data streams are first re-ordered and shuffled by the bf_switch in order to make the data suitable for the bf_units. The bf module displayed in Figure 1 contains four bf_units where each bf_unit processes a limited number of subbands. In case of the Apertif beamformer as discussed in [3] there are four bf_units required to process all 24 subbands. The bf_unit performs the multiplication of the input data with the weight factors, the addition of all multiplier results and the quantization of the result. Each bf_unit also contains a beamlet statistic unit that calculates the power in each beamlet. The output of each bf_unit contains a stream of beamlets.

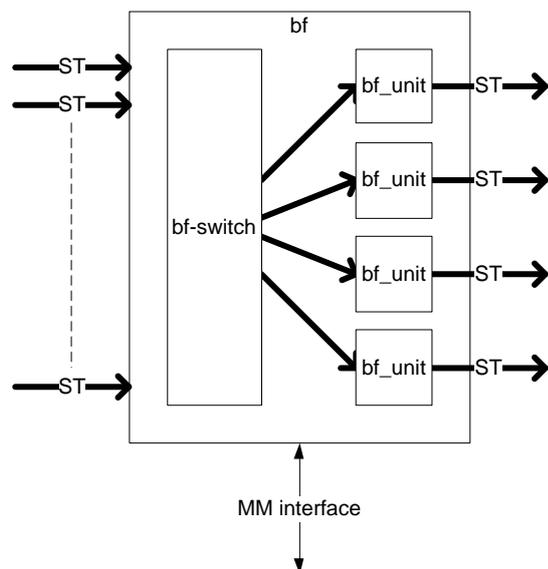


Figure 1: Beamformer module overview

The MM interface is used to upload (and read back) the weight factors, to specify the number of beamlets per subband and to read out the beamlet statistics.

1.3 Mode of operation

The bf module is a data-driven process that performs a series of DSP operations on the incoming data streams. The process is started as soon as data is offered on the inputs and it stops when the stream of input data stops. With the instantiation of the bf module for the Apertif project it is possible to calculate 1024 beamlets, based on 64 signal paths, where each signal path carries the data of 24 subbands. The 1024 beamlets can be equally divided over the 24 subbands, resulting in an average of 42 beamlets per subband. It is also possible to divide the 1024 beamlets in an unequal way over the 24 subbands. A single bf_unit as

shown in Figure 1 is responsible for processing $24/4 = 6$ subbands. The weight factors that are required to calculate the beamlets can then be divided over the subbands as shown in Figure 2.

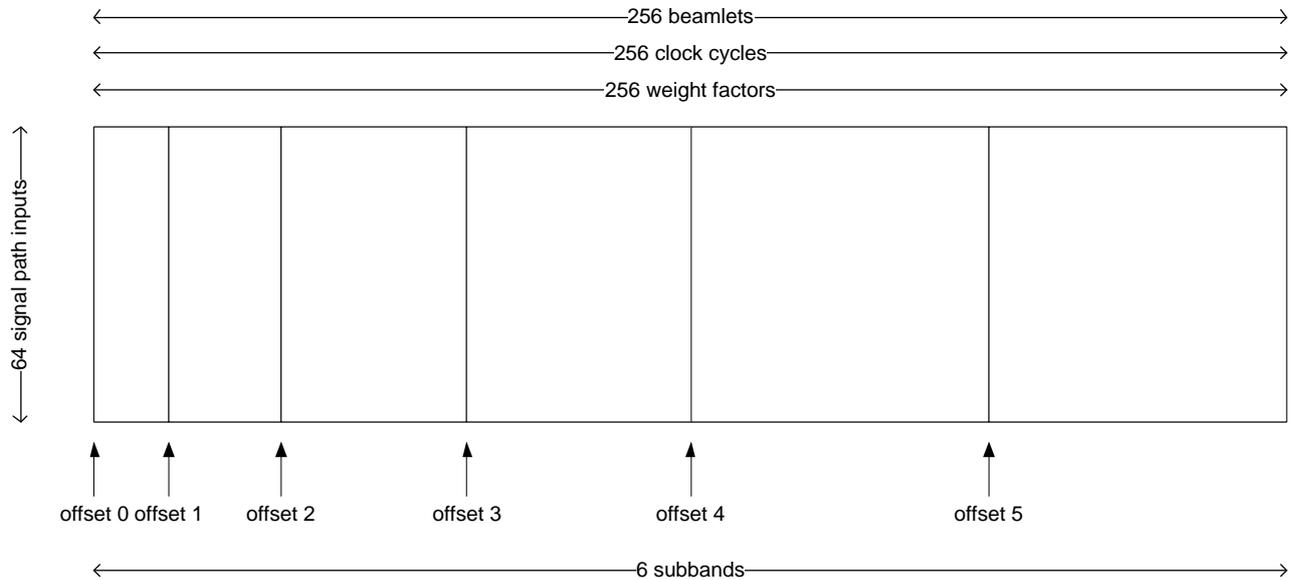


Figure 2 Beamlets divided over subbands for one bf_unit

When data is offered to the bf_units the buffer with 256 weight factors will be read in a cyclic way and the data will be fed to the first input of a single complex multiplier. On the second input of the multiplier the subband samples will be provided. The offsets in Figure 2 mark the moments in time whereas the next subband sample must be read from the input fifo and offered to the multiplier. A simple representation of such a single multiplier process is shown in Figure 3.

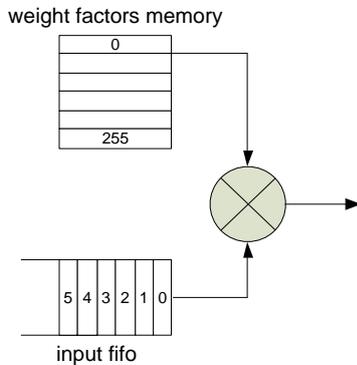


Figure 3 One bf multiplier per signal path input

Note 1: the only restriction in this process is that from every subband at least one beamlet must be made. If it is necessary to calculate beamlets of a single subband than this specific subband should be selected multiple times in the Subband Select module. The Subband Select module is located before the bf module in the back node. [2]

Note 2: A bf module with 4 bf_units contains a total of $4 \times 64 \times 256$ weight factors and $4 \times 64 = 256$ complex multipliers.

The subband data of all 64 inputs are processed (multiplied with the weight factors) in parallel. After multiplication the products of all 64 inputs are summed resulting in an output stream that carries the beamlets. Every bf_unit is equipped with a beamlet statistics module that allows reading out the power of every beamlet that is integrated over a sync period (= typically 1 second).

2 Firmware interface

This chapter covers all firmware interface related topics of the bf module. It describes the functionality of the in- and output ports.

2.1 Clock domains

There are two clock domains used in the bf module: the mm_clk and the dp_clk domain. Figure 4 shows an overview of the clock domains in the bf module. The connected units are a set of registers that hold the offset values for the input fifo, a dual ported ram that holds the weight factors and a dual ported ram that holds the results of the beamlet statistics. Table 1 lists both clocks and their characteristics.

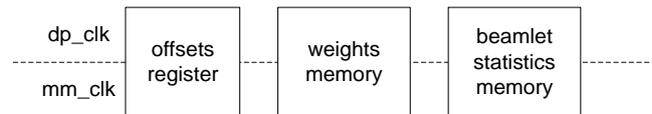


Figure 4: bf module clock domains

Name	Frequency (MHz)	Description
DP_CLK	200 MHz	Clock for the datapath
MM_CLK	125 MHz	Clock for the memory mapped interface.

Table 1: bf module clock signals

2.2 Parameters

The parameters that define an instantiation of the bf module are grouped in a VHDL record that is defined in the file bf_pkg.vhd. The items in this record are listed in Table 2. The column named “value” specifies the value that is used in the bf module for the Apertif system.

Generic	Type	Value	Description
nof_signal_paths	POSITIVE	64	Specifies the number of input signals that are used in the beamformer.
nof_input_streams	POSITIVE	16	The number of data streams that are provided on the input of the bf module. Each data stream contains the data of one or more signal paths.
nof_subbands	POSITIVE	24	Specifies the number of subbands per incoming signal path.
nof_weights	POSITIVE	256	The number of weight factors that are present in each bf_unit. This value merely defines the number of beamlets that are produced in a single bf_unit.
nof_offsets	POSITIVE	6	This number represents the number of subbands that are to be processed within one bf_unit.
nof_bf_units	POSITIVE	4	The number of bf_units that are instantiated in the bf module. Each bf_unit will process nof_subbands/nof_bf_units subbands which correspond to nof_offsets.
in_dat_w	POSITIVE	16	Width in bits of the incoming data. This value specifies the width of the real and the imaginary part.
in_weight_w	POSITIVE	16	The width in bits of the real and imaginary part of the weight factors.
bst_dat_w	POSITIVE	16	The bitwidth of the real and imaginary part of the beamlets that are fed to the beamlet statistics unit.
out_dat_w	POSITIVE	4	The bitwidth of the real and imaginary part of the beamlets that are sent tot the output stream.
stat_data_w	POSITIVE	56	Width of the output of the beamlet statistics unit. This value

Doc.nr.: ASTRON-RP-1291

Rev.: 0.4

Date: 04-04-2012

Class.: Public

UniBoard

			must be high enough to accommodate the highest possible bitgrowth in the beamlet statistic unit. Highest integration period is set to one second.
stat_data_sz	POSITIVE	2	This number specifies how many 32-bit registers are required to read out one accumulated value.

Table 2: bf parameters

2.3 Interface signals

The interface signals of the bf module are shown in Figure 5 and Table 3 lists the general specifications of these interfaces. More detailed information about the interfaces can be found in the following paragraphs.



Figure 5: interface signals

Interface	Type	Size or Span	Description
in_sosi_arr	t_dp_sosi_arr	nof_input_streams	Array of input streams where each stream holds the data of a number of input signals.
out_raw_sosi_arr	t_dp_sosi_arr	nof_bf_units	Array of output streams containing un-quantized beamlet data. Output only used for verification in simulation.
out_qua_sosi_arr	t_dp_sosi_arr	nof_bf_units	Array of output streams containing the quantized beamlet data.
mm_weight_mosi	t_mem_mosi	nof_bf_units * nof_signal_paths * nof_weights	Array of mosi interfaces for storing the weight factors.
mm_weight_miso	t_mem_miso	nof_bf_units * nof_signal_paths * nof_weights	Array of miso interfaces for reading back the written weight factors.
mm_bst_mosi	t_mem_mosi	nof_bf_units * nof_weights * stat_data_sz	A mosi interface for each bf_unit to read out the beamlet statistics data.
mm_bst_miso	t_mem_miso	nof_bf_units * nof_weights * stat_data_sz	A miso interface for each bf_unit to read out the beamlet statistics data.

Doc.nr.: ASTRON-RP-1291

Rev.: 0.4

Date: 04-04-2012

Class.: Public

UniBoard

mm_offsets_mosi	t_mem_mosi	nof_bf_units * nof_offsets	A mosi interface to write and read the offsets registers. The offset registers specify the offsets for the input data.
mm_offsets_miso	t_mem_miso	nof_bf_units * nof_offsets	A miso interface for reading back the offset registers.
dp_clk	std_logic	na	Datapath clock
dp_rst	std_logic	na	Datapath reset
mm_clk	std_logic	na	Memory mapped interface clock
mm_rst	std_logic	na	Memory mapped interface reset

Table 3: interface signals

2.3.1 IN_SOSI_ARR interface

The `in_sosi_arr` is an array of streams where each stream holds the data of one or more input signal paths. The data is expected to arrive in perfectly aligned packets meaning that every stream starts and ends a packet at the same clock cycle. The basic format of such a packet must comply with the DP streaming interface as defined in [1]. For the `bf` module however a more detailed definition of the data format is required. The data format for one input stream which is depicted in Figure 6. It shows the format for a `bf` module with the following parameters: `nof_signal_paths = 64`, `nof_input_streams = 16`, `nof_subbands = 24`. These parameters result in a packet format containing 4 (signal paths) x 24 (subbands) = 96 complex samples.

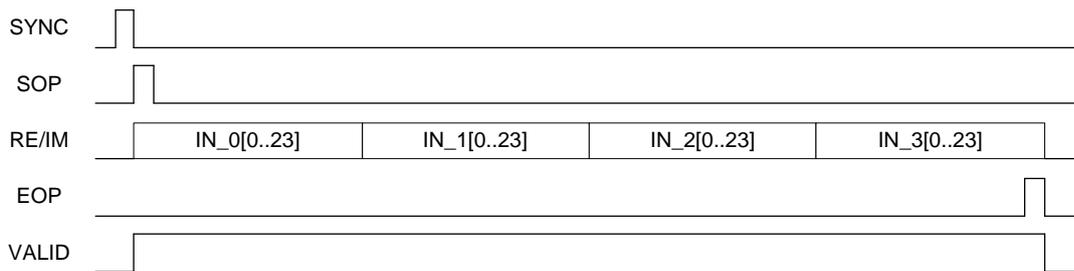


Figure 6 Input packet format beamformer

Two other requirements regarding the packet format have to be met as well:

- The whole packet must be sent in one burst. This means that the valid signal may not be interrupted during the transmission of a packet.
- A packet may only be transmitted once every `nof_weights` clock cycles as shown in Figure 7 in order to avoid fifo overflow in the `bf_units`.

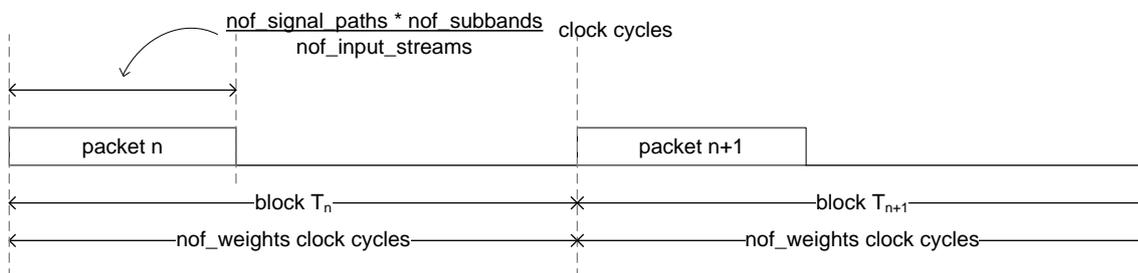


Figure 7 Minimum frequency of packet transmission

Note that the `bf` module does not provide a corresponding `in_siso_arr`, because there is no need for back pressure. The `bf_module` is designed in such a way that it can always receive data in case the above specifications are met.

2.3.2 OUT_RAW_SOSI_ARR and OUT_QUA_SOSI_ARR interfaces

The bf module has two output arrays for its data that both comply with the DP streaming interface as defined in [1]. One output (out_raw_sosi_arr) contains the raw data from the bf_units and the other one (out_qua_sosi_arr) contains the data after quantization. Both interfaces provide a stream of packets where each packet contains the data of nof_weight beamlets. The format of the outgoing packets is displayed in Figure 8 where the nof_weights is set to 256.

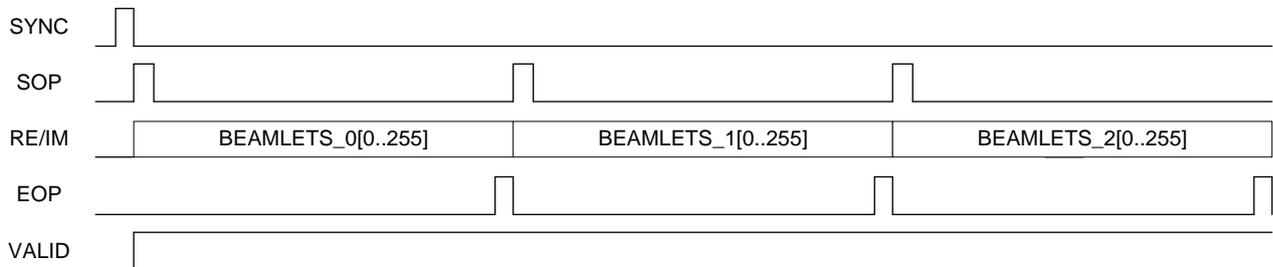


Figure 8 Output packet format beamformer

Note that the packet stream is an uninterrupted stream once it has started and the output arrays are not accompanied with a corresponding siso array, since the bf module is not required to cope with back pressure.

2.3.3 MM_WEIGHT_MOSI interface

The bf module has an mm interface to accommodate uploading of the weight factors. The total span of the mm interface is determined by $\text{nof_weights} \times \text{nof_signal_paths} \times \text{nof_bf_units}$. For the Apertif instantiation this means: $\text{nof_weights} = 256$, $\text{nof_signal_paths} = 64$ and $\text{nof_bf_units} = 4$. This leads to an interface span of $256 \times 64 \times 4 = 65536$ registers. For both the mosi and miso interface the signals are listed in Table 4.

Signal	Type	Description
mm_weight_mosi.address[15:0]	MOSI	Word addresses range for the mm_weight_mosi interface.
mm_weight_mosi.wrddata[31:0]	MOSI	Write data word, must be valid when wr is asserted.
mm_weight_mosi.wr	MOSI	Write strobe.
mm_weight_mosi.rd	MOSI	Read strobe.
mm_weight_miso.rddata[31:0]	MISO	Read data word which is valid one clock cycle after assertion of rd.

Table 4 mm_weight_mosi interface

2.3.4 MM_BST_MOSI interface

The beamlet statistics can be read via the mm_bst_mosi interface. The address span of this interface is determined by $\text{nof_bf_units} \times \text{stat_data_sz} \times \text{nof_weights}$. Table 5 shows the interface signals where $\text{nof_bf_units} = 4$, $\text{nof_weights} = 256$ and $\text{stat_data_sz} = 2$.

Signal	Type	Description
mm_bst_mosi.address[10:0]	MOSI	Word address range for mm_bst_mosi interface, supporting $4 \times 2 \times 256 = 2048$ registers.
mm_bst_mosi.wrddata[31:0]	MOSI	Write data word, must be valid when wr is asserted.
mm_bst_mosi.wr	MOSI	Write strobe.
mm_bst_mosi.rd	MOSI	Read strobe.
mm_bst_miso.rddata[31:0]	MISO	Read data word which is valid one clock cycle after assertion of rd.

Table 5 mm_bst_mosi interface

2.3.5 MM_OFFSETS_MOSI interface

The offsets for the incoming data can be set via the mm_offsets_mosi interface. The address span is determined by the nof_offsets and the nof_bf_units. For nof_bf_units = 4 and nof_offsets = 6 this results in the interface characteristics as shown in Table 6.

Signal	Type	Description
mm_offsets_mosi.address[4:0]	MOSI	Word address range for mm_offsets_mosi interface
mm_offsets_mosi.wrdata[31:0]	MOSI	Write data word, must be valid when wr is asserted.
mm_offsets_mosi.wr	MOSI	Write strobe.
mm_offsets_mosi.rd	MOSI	Read strobe.
mm_offsets_miso.rddata[31:0]	MISO	Read data word which is valid one clock cycle after assertion of rd.

Table 6 mm_offsets_mosi interface

2.3.6 Clocks and resets

Table 7 shows an overview of the clocks and reset signals that are available on the bf module.

Signal	Type	Description
dp_clk	Clock	Clock input for the datapath interface of the bf module.
dp_rst	Reset	Reset input for the datapath clock domain registers.
mm_clk	Clock	Clock input for the memory mapped interface parts of the bf module.
mm_rst	Reset	Reset input for the memory mapped interface parts of the bf module.

Table 7 Clocks and resets

3 Software interface

This chapter describes the software interface for the bf module. The bf module contains three different register spans. The first span holds the offsets for the weight factors. The second span contains the registers that represent the values of the weight factors. The third span contains the registers that contain the beamlet statistics data. The size of each span depends on the value of the parameters that are used to configure the bf module.

The next paragraphs give a detailed overview of all the registers and their addresses. The shown registers are based on a parameterized bf module. More information on the parameters can be found in paragraph 2.2.

3.1 Offsets span

Table 8 shows the registers that are available in the offsets span whereas `nof_offsets = 6` and `nof_bf_units = 4`. Word size is 32 bit. Note that the first offset value is read only and always zero. These are the criteria for valid values in all other offset registers:

- Values must be higher than zero.
- No double values are accepted per `bf_unit`.
- Values must be lower than `nof_weights`.

Name	Address (words)	Size (words)	Read/Write	Description
<code>bf_unit_0_offset_0</code>	0x0	1	ro	Register specifies at which weight factor input sample 0 has to be read from the input fifo of <code>bf_unit 0</code> . Value is fixed to 0x0.
<code>bf_unit_0_offset_1</code>	0x1	1	r/w	Register specifies at which weight factor input sample 1 has to be read from the input fifo of <code>bf_unit 0</code> .
<code>bf_unit_0_offset_2</code>	0x2	1	r/w	Register specifies at which weight factor input sample 2 has to be read from the input fifo of <code>bf_unit 0</code> .
<code>bf_unit_0_offset_3</code>	0x3	1	r/w	Register specifies at which weight factor input sample 3 has to be read from the input fifo of <code>bf_unit 0</code> .
<code>bf_unit_0_offset_4</code>	0x4	1	r/w	Register specifies at which weight factor input sample 4 has to be read from the input fifo of <code>bf_unit 0</code> .
<code>bf_unit_0_offset_5</code>	0x5	1	r/w	Register specifies at which weight factor input sample 5 has to be read from the input fifo of <code>bf_unit 0</code> .
<code>bf_unit_1_offset_0</code>	0x8	1	ro	Register specifies at which weight factor input sample 0 has to be read from the input fifo of <code>bf_unit 1</code> . Value is fixed to 0x0.
<code>bf_unit_1_offset_1</code>	0x9	1	r/w	Register specifies at which weight factor input sample 1 has to be read from the input fifo of <code>bf_unit 1</code> .
<code>bf_unit_1_offset_2</code>	0xA	1	r/w	Register specifies at which weight factor input sample 2 has to be read from the input fifo of <code>bf_unit 1</code> .
<code>bf_unit_1_offset_3</code>	0xB	1	r/w	Register specifies at which weight factor input sample 3 has to be read from the input fifo of <code>bf_unit 1</code> .
<code>bf_unit_1_offset_4</code>	0xC	1	r/w	Register specifies at which weight factor input sample 4 has to be read from the input fifo of <code>bf_unit 1</code> .
<code>bf_unit_1_offset_5</code>	0xD	1	r/w	Register specifies at which weight factor input sample 5 has to be read from the input fifo of <code>bf_unit 1</code> .
<code>bf_unit_2_offset_0</code>	0x10	1	ro	Register specifies at which weight factor input sample 0 has to be read from the input fifo of <code>bf_unit 2</code> . Value is fixed to 0x0.
<code>bf_unit_2_offset_1</code>	0x11	1	r/w	Register specifies at which weight factor input sample 1 has to be read from the input fifo of <code>bf_unit 2</code> .
<code>bf_unit_2_offset_2</code>	0x12	1	r/w	Register specifies at which weight factor input sample 2 has to be read from the input fifo of <code>bf_unit 2</code> .
<code>bf_unit_2_offset_3</code>	0x13	1	r/w	Register specifies at which weight factor input sample

Doc.nr.: ASTRON-RP-1291

Rev.: 0.4

Date: 04-04-2012

Class.: Public

UniBoard

				3 has to be read from the input fifo of bf_unit 2.
bf_unit_2_offset_4	0x14	1	r/w	Register specifies at which weight factor input sample 4 has to be read from the input fifo of bf_unit 2.
bf_unit_2_offset_5	0x15	1	r/w	Register specifies at which weight factor input sample 5 has to be read from the input fifo of bf_unit 2.
bf_unit_3_offset_0	0x18	1	ro	Register specifies at which weight factor input sample 0 has to be read from the input fifo of bf_unit 3. Value is fixed to 0x0.
bf_unit_3_offset_1	0x19	1	r/w	Register specifies at which weight factor input sample 1 has to be read from the input fifo of bf_unit 3.
bf_unit_3_offset_2	0x1A	1	r/w	Register specifies at which weight factor input sample 2 has to be read from the input fifo of bf_unit 3.
bf_unit_3_offset_3	0x1B	1	r/w	Register specifies at which weight factor input sample 3 has to be read from the input fifo of bf_unit 3.
bf_unit_3_offset_4	0x1C	1	r/w	Register specifies at which weight factor input sample 4 has to be read from the input fifo of bf_unit 3.
bf_unit_3_offset_5	0x1D	1	r/w	Register specifies at which weight factor input sample 5 has to be read from the input fifo of bf_unit 3.

Table 8: offsets span

3.2 Weight factors span

Each complex weight factor has a 32-bit memory location. The real part is located in the 16 least significant bits and the imaginary part is located in the 16 most significant bits of the register as shown in Figure 9.

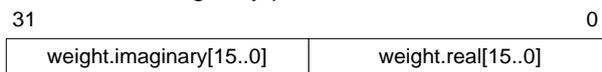


Figure 9 Weight factor register

For each signal path there are `nof_weight` weight factors which results in `nof_signal_paths*nof_weights` registers per `bf_unit`. For the Apertif system `nof_signal_paths = 64`, `nof_weights = 256` and `nof_bf_units = 4`. This leads to the address map of Table 9. Note that not all registers are listed in the table in order to save space.

Name	Address (words)	Size (words)	Read/Write	Description
bu_0_sp_0_wf_0	0x0	1	r/w	Weight factor 0 on bf_unit 0 for signal path 0
bu_0_sp_0_wf_1	0x1	1	r/w	Weight factor 1 on bf_unit 0 for signal path 0
bu_0_sp_0_wf_2	0x2	1	r/w	Weight factor 2 on bf_unit 0 for signal path 0
-----	-----	---	-----	-----
bu_0_sp_0_wf_254	0xFE	1	r/w	Weight factor 254 on bf_unit 0 for signal path 0
bu_0_sp_0_wf_255	0xFF	1	r/w	Weight factor 255 on bf_unit 0 for signal path 0
bu_0_sp_1_wf_0	0x100	1	r/w	Weight factor 0 on bf_unit 0 for signal path 1
bu_0_sp_1_wf_1	0x101	1	r/w	Weight factor 1 on bf_unit 0 for signal path 1
-----	-----	---	-----	-----
bu_0_sp_1_wf_254	0x1FE	1	r/w	Weight factor 254 on bf_unit 0 for signal path 1
bu_0_sp_1_wf_255	0x1FF	1	r/w	Weight factor 255 on bf_unit 0 for signal path 1
bu_0_sp_2_wf_0	0x200	1	r/w	Weight factor 0 on bf_unit 0 for signal path 2
bu_0_sp_2_wf_1	0x201	1	r/w	Weight factor 1 on bf_unit 0 for signal path 2
-----	-----	---	-----	-----
-----	-----	---	-----	-----
bu_0_sp_63_wf_254	0x3FFE	1	r/w	Weight factor 254 on bf_unit 0 for signal path 63
bu_0_sp_63_wf_255	0x3FFF	1	r/w	Weight factor 255 on bf_unit 0 for signal path 63
bu_1_sp_0_wf_0	0x4000	1	r/w	Weight factor 0 on bf_unit 1 for signal path 0
bu_1_sp_0_wf_1	0x4001	1	r/w	Weight factor 1 on bf_unit 1 for signal path 0

Doc.nr.: ASTRON-RP-1291
 Rev.: 0.4
 Date: 04-04-2012
 Class.: Public

-----	-----	---	-----	-----
-----	-----	---	-----	-----
bu_1_sp_63_wf_254	0x7FFE	1	r/w	Weight factor 254 on bf_unit 1 for signal path 63
bu_1_sp_63_wf_255	0x7FFF	1	r/w	Weight factor 255 on bf_unit 1 for signal path 63
bu_2_sp_0_wf_0	0x8000	1	r/w	Weight factor 0 on bf_unit 2 for signal path 0
bu_2_sp_0_wf_1	0x8001	1	r/w	Weight factor 1 on bf_unit 2 for signal path 0
-----	-----	---	-----	-----
-----	-----	---	-----	-----
bu_2_sp_63_wf_254	0xBFFE	1	r/w	Weight factor 254 on bf_unit 2 for signal path 63
bu_2_sp_63_wf_255	0xBFFF	1	r/w	Weight factor 255 on bf_unit 2 for signal path 63
bu_3_sp_0_wf_0	0xC000	1	r/w	Weight factor 0 on bf_unit 3 for signal path 0
bu_3_sp_0_wf_1	0xC001	1	r/w	Weight factor 1 on bf_unit 3 for signal path 0
-----	-----	---	-----	-----
-----	-----	---	-----	-----
bu_3_sp_63_wf_254	0xFFFFE	1	r/w	Weight factor 254 on bf_unit 3 for signal path 63
bu_3_sp_63_wf_255	0xFFFF	1	r/w	Weight factor 255 on bf_unit 3 for signal path 63

Table 9 weight factors span

3.3 Beamlet statistics span

Each bf_unit contains a beamlet statistics unit. The beamlet statistics unit estimates the power of each beamlet value and integrates these values during a sync period. When a sync period has expired (in other words: a sync pulse is applied) the registers will be updated with the new integrated power values. The number of registers is determined by $stat_data_sz * nof_weights * nof_bf_units$. In the Apertif system $stat_data_sz = 2$, $nof_weights = 256$ and $nof_bf_units = 4$. This leads to 2048 registers which are partly listed in Table 10. Note that each beamlet statistic value is spread over two 32-bit registers. The actual bit width of a beamlet statistic is set via parameter $stat_data_w$, which is set to 56 in the Apertif system. This means that the upper 8 bits of the “up” register can be discarded.

Name	Address (words)	Size (words)	Read/Write	Description
bu_0_bst_0_low	0x0	1	r/w	32 lsb's of power in beamlet 0 on bf_unit 0
bu_0_bst_0_up	0x1	1	r/w	32 msb's of power in beamlet 0 on bf_unit 0
bu_0_bst_1_low	0x2	1	r/w	32 lsb's of power in beamlet 1 on bf_unit 0
bu_0_bst_1_up	0x3	1	r/w	32 msb's of power in beamlet 1 on bf_unit 0
bu_0_bst_2_low	0x4	1	r/w	32 lsb's of power in beamlet 2 on bf_unit 0
bu_0_bst_2_up	0x5	1	r/w	32 msb's of power in beamlet 2 on bf_unit 0
-----	-----	---	-----	-----
bu_0_bst_255_low	0x1FE	1	r/w	32 lsb's of power in beamlet 255 on bf_unit 0
bu_0_bst_255_up	0x1FF	1	r/w	32 msb's of power in beamlet 255 on bf_unit 0
bu_1_bst_0_low	0x200	1	r/w	32 lsb's of power in beamlet 0 on bf_unit 1
bu_1_bst_0_up	0x201	1	r/w	32 msb's of power in beamlet 0 on bf_unit 1
bu_1_bst_1_low	0x202	1	r/w	32 lsb's of power in beamlet 1 on bf_unit 1
bu_1_bst_1_up	0x203	1	r/w	32 msb's of power in beamlet 1 on bf_unit 1
-----	-----	---	-----	-----
bu_1_bst_255_low	0x3FE	1	r/w	32 lsb's of power in beamlet 255 on bf_unit 1
bu_1_bst_255_up	0x3FF	1	r/w	32 msb's of power in beamlet 255 on bf_unit 1
bu_2_bst_0_low	0x400	1	r/w	32 lsb's of power in beamlet 0 on bf_unit 2
bu_2_bst_0_up	0x401	1	r/w	32 msb's of power in beamlet 0 on bf_unit 2
bu_2_bst_1_low	0x402	1	r/w	32 lsb's of power in beamlet 1 on bf_unit 2
bu_2_bst_1_up	0x403	1	r/w	32 msb's of power in beamlet 1 on bf_unit 2
-----	-----	---	-----	-----
bu_2_bst_255_low	0x5FE	1	r/w	32 lsb's of power in beamlet 255 on bf_unit 2
bu_2_bst_255_up	0x5FF	1	r/w	32 msb's of power in beamlet 255 on bf_unit 2
bu_3_bst_0_low	0x600	1	r/w	32 lsb's of power in beamlet 0 on bf_unit 3

Doc.nr.: ASTRON-RP-1291

Rev.: 0.4

Date: 04-04-2012

Class.: Public

bu_3_bst_0_up	0x601	1	r/w	32 msb's of power in beamlet 0 on bf_unit 3
bu_3_bst_1_low	0x602	1	r/w	32 lsb's of power in beamlet 1 on bf_unit 3
bu_3_bst_1_up	0x603	1	r/w	32 msb's of power in beamlet 1 on bf_unit 3
-----	-----	---	-----	-----
bu_3_bst_255_low	0x7FE	1	r/w	32 lsb's of power in beamlet 255 on bf_unit 3
bu_3_bst_255_up	0x7FF	1	r/w	32 msb's of power in beamlet 255 on bf_unit 3

Table 10 beamlet statistics span

4 Module Design

4.1 Algorithm

For the Apertif system the bf module should implement the following equation:

$$B_{j,k}[nN] = \sum_{i=0}^S s_{i,j}[nN] \cdot w_{i,j,k}[nM]$$

There are several ways to implement this equation into an FPGA. The following criteria as stated in [2] have been used to estimate the most suitable structure for the bf module:

- Does it scale nicely with larger or smaller values for S , K or N_{FN} ?
- Does it efficiently use the FPGA DSP-blocks, RAM and logic resources?
- Can it nicely handle different numbers of beamlets per subband?
- Does it easily fit to the way the subbands arrive at the input?

During estimation the structures as described/proposed in [2] have been examined on hardware in order to find the best solution. Efficient use of the FPGA's DSP blocks was the prime criteria in this survey. In [3] a detailed description of this survey can be found. The most important conclusion of the survey is that it is impossible to use any of the embedded adder-structures in the DSP blocks for accumulating all `nof_signal_paths` into a beamlet. The DSP blocks can only be used for complex multiplication and not for the accumulation and therefore the accumulation part has to be done in logic. This restriction has led to the architecture described in the following paragraphs.

4.2 Architecture

4.2.1 bf module

Figure 10 shows an overview of the bf module. It is build out of two building blocks: a unique `bf_switch` and a number of `bf_units`. The input consists of a set of input streams providing input data and the output is a set of data streams that holds beamlets. A memory mapped interface is used for control and monitoring. The selected structure divides the `nof_subbands` subbands equally over the `bf_units`. Each `bf_unit` then calculates all the beamlets for that subset of the subbands. For the Apertif system this means that the 24 subbands on the input are divided over 4 `bf_units` so each `bf_unit` will calculate the beamlets for 6 subbands. Taken into account that the bandwidth of one subband is 0.78 MHz and the target clock frequency (the multiplier frequency) is set to 200 MHz it can be derived that each `bf_unit` can calculate 256 beamlets out of the 6 subbands. This makes the average number of beamlets per subband $256/6 = 42$.

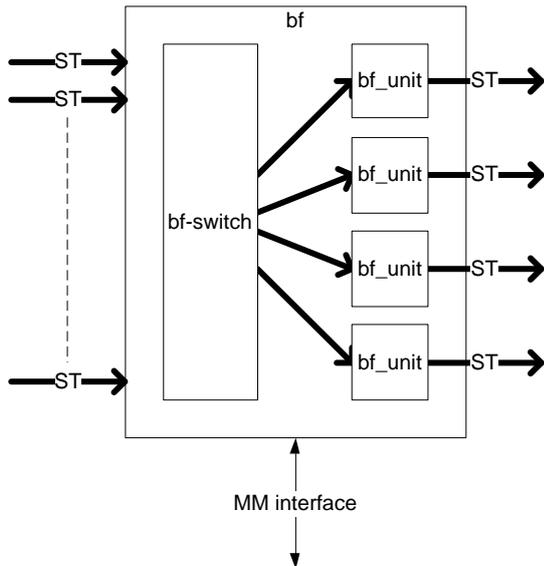


Figure 10 Architecture of the bf module

4.2.2 bf_switch

The `bf_switch` is predominately a unit that applies re-ordering and shuffling to the incoming data in order to prepare the data correctly for the `bf_units`. The number of inputs is defined by the `nof_input_streams` and since the switch has to provide all inputs of the `bf_units` with data, the number of outputs is defined by the `nof_signal_paths*nof_bf_units`.

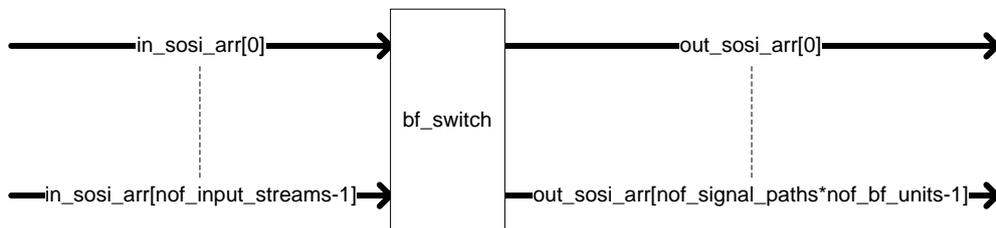


Figure 11 Architecture of the bf_switch

The internal architecture of the `bf_switch` depends on the format of the incoming data packets, since the format of the outgoing packets is defined by the `bf_unit`.

4.2.3 bf_unit

The internal architecture of a single `bf_unit` is shown in Figure 12. For each signal path (`nof_signal_paths`) a weight memory is used that can hold `nof_weights` weight factors, an input fifo is used for the incoming data and a complex multiplier is there for the multiplication. The results of all multipliers are connected to a multiple-input adder-tree that summarizes all its inputs. Then a packetizer unit is used to generate a data stream that consists of packets of `nof_weight` beamlets. That stream is fed to two quantizers. The first quantizer prepares the data stream for the beamlet statistics module whereas the second quantizer prepares the data for the data output stream.

The `bf_unit` is controlled by a `bf_unit` controller that takes care of addressing the weight memories, reading out the input fifos and the synchronization of the packetizer.

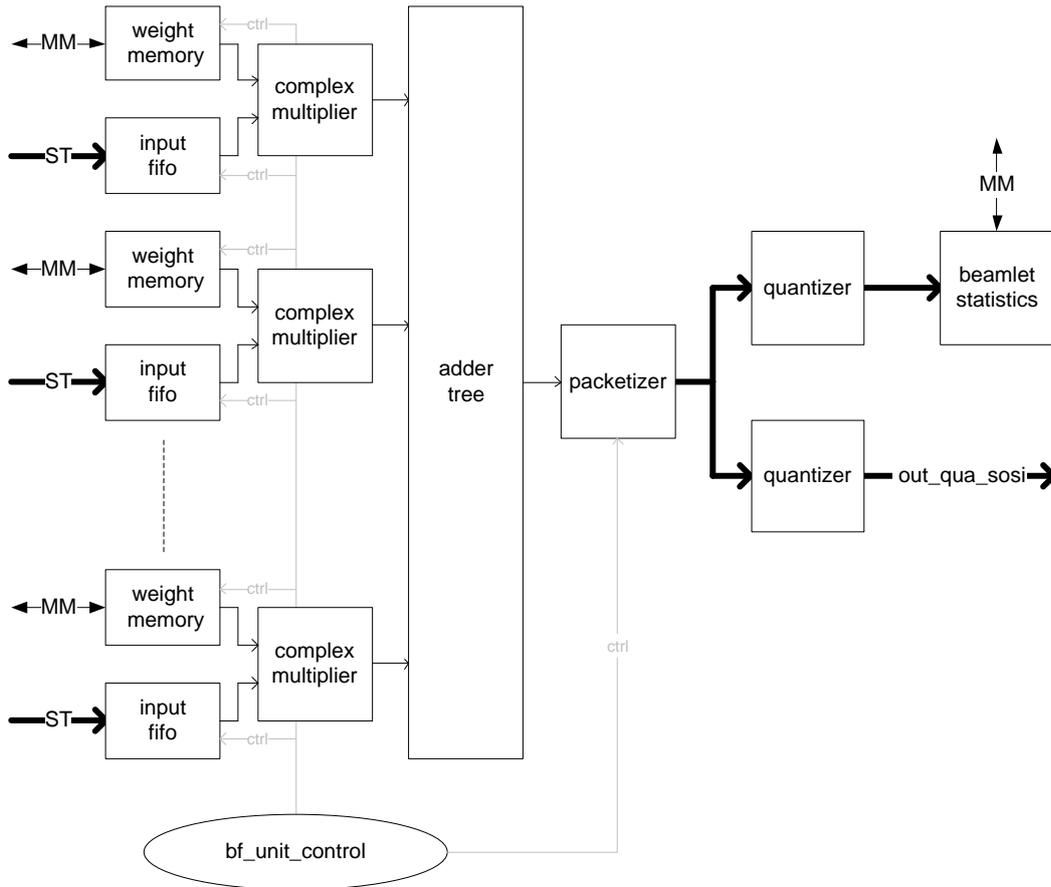


Figure 12: Basic architecture of a single bf_unit

5 Implementation

This chapter describes the implementation of the bf module. The bf module consists of bf_units and a bf_switch that are interconnected. Both units are described in detail in the following paragraphs.

5.1 bf_switch

The task of the bf_switch is the reordering of the incoming data streams into data streams that are suitable for the bf_units. In this description the actual values of the Apertif project are used to explain the working of the bf_switch. In order to fulfil the requirement there are several possibilities for implementation. A first approach led to a direct and effective architecture called bf_switch_a_direct. Unfortunately this direct version was causing timing issues during synthesis and therefore a second architecture was developed called bf_switch_a_sort_distribute. Both architectures will be described

5.1.1 Input format

The bf_switch expects nof_input_streams (16) data streams. Each stream contains the nof_subbands (24) of nof_signal_inputs (64) / nof_input_streams (16) = 4 signal paths. The format of the data streams is described in detail in paragraph 2.3.1.

5.1.2 Output format

The output of the bf_switch will feed all the inputs of all the connected bf_units. That are nof_signal_paths (64) * nof_bf_units (4) = 256 output streams. As earlier explained each bf_unit will process only nof_subbands (24) / nof_bf_units (4) = 6 subbands. This should lead to a streaming packet format that is displayed in Figure 13 which shows a packet that contains the first 6 subband samples (SB[0:5]) of signal path 0 (SP0). This packet will be offered to the first input of the first bf_unit.

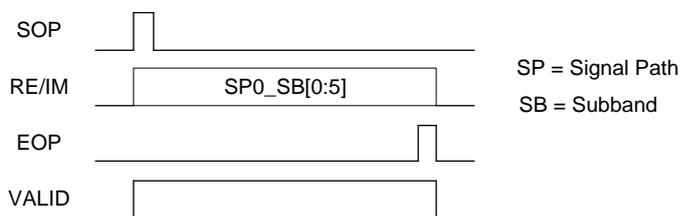


Figure 13 Output format of bf_switch

An overview of the packets for all inputs for all bf_units is shown in Figure 14.

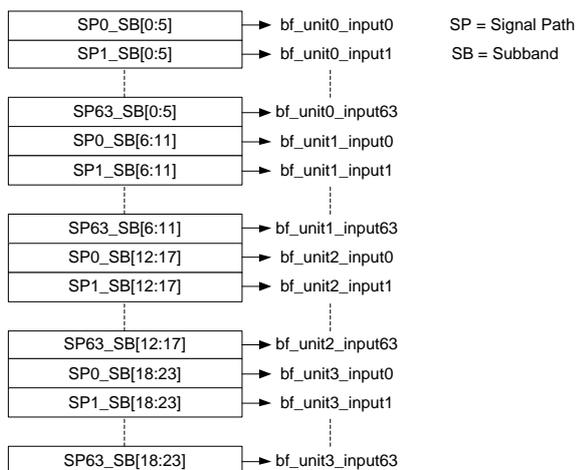


Figure 14 Packet distribution of bf_switch

5.1.3 Architecture: bf_switch_a_direct

The direct approach is based on sending every incoming sample straight to the output where it should go, in other words: the sorting and distribution is done in one step. The bf_switch expects the incoming data to be in the format as defined in paragraph 2.3.1 and based on that format it performs the serial to parallel function. Figure 15 shows the bf_switch result of one input stream. The incoming stream is registered and then passed on to the correct registered output.

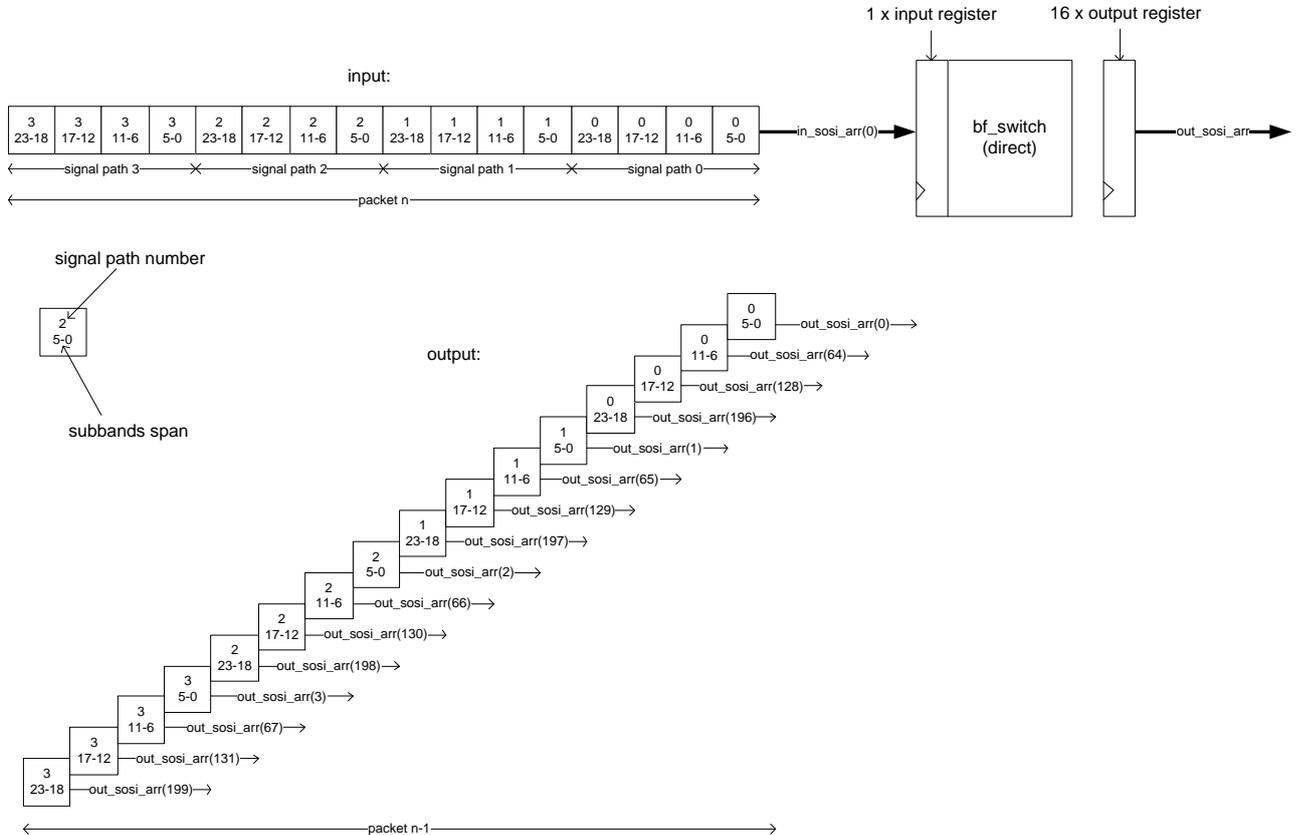


Figure 15 In- and output for one input stream of the bf_switch_a_direct architecture

Each outgoing packet is sent to one of the elements of the sino output array. The inputs of the bf_units should be connected as follows:

- out_sosi_arr(63:0) → bf_unit_0(63:0)
- out_sosi_arr(127:64) → bf_unit_1(63:0)
- out_sosi_arr(195:128) → bf_unit_2(63:0)
- out_sosi_arr(255:196) → bf_unit_3(63:0)

Note that the outgoing packets are not aligned in time. It is the responsibility of the bf_units to only start when the last packet has been written.

The architecture of bf_switch_a_direct is based on a 2-state state-machine. It waits in the idle state until data is received and it then continues in the run state with the reordering of the incoming samples. In the run state an address counter is incremented every nof_subbands/nof_bf_units=6 clock cycles. This address counter is used to address the correct element of the output array.

Although this implementation looks simple and efficient it is obvious that the routing effort for the place and route tool is enormous. Every input sample has to be transported to the according bf_unit in only two clock cycles.

5.1.4 Architecture: bf_switch_a_sort_distribute

In order to achieve better timing results the architecture for the bf_switch has been improved. The improvements have been made by separating the sorting function and the distribution function by creating a two-step approach. The first step performs the sorting where the sorted data for the output is written into a fifo. The second step consists of reading out the data from the fifo and distribute it over the inputs of the bf_units. A schematic overview of this architecture is given in Figure 16.

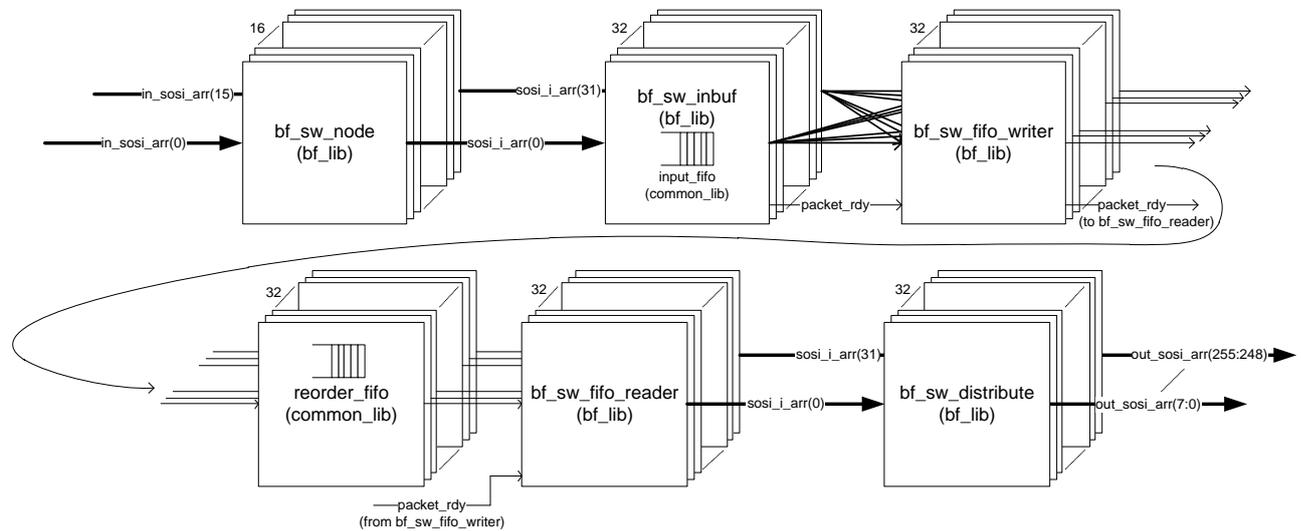


Figure 16 Schematic overview of bf_switch_a_sort_distribute

5.1.4.1 bf_sw_node

The first unit in the architecture is the bf_sw_node. Every incoming data stream has one bf_sw_node connected to it (total number of bf_sw_nodes = 16). The bf_sw_node splits the incoming stream in two new streams. This parallelization step is necessary in order to be able to do the sorting of the current packet before the next packet arrives. The parameters and interface signals of the bf_sw_node are listed in Table 11 and Table 12.

Generic	Type	Value	Description
g_nof_outputs	POSITIVE	2	Specifies the number of output streams.
g_switch_interval	POSITIVE	48	Specifies the alternating frequency, expressed in number of valid samples.

Table 11 bf_sw_node parameters

Interface	Type	Size or Span	Description
in_sosi	t_dp_sosi	na	The streaming interface that carries the incoming packets.
out_sosi_arr	t_dp_sosi_arr	g_nof_outputs	An array of streaming interfaces where each element carries a segment of the input stream.
clk	std_logic	na	Datapath clock input.
rst	std_logic	na	Datapath reset input.

Table 12 bf_sw_node interface signals

As soon as there is valid data present on the input the bf_sw_node will send the first g_switch_interval valid data samples to the first output. The second g_switch_interval valid data samples will be send to the second output. The next g_switch_interval samples will be send to the first output again etc. For g_nof_outputs=2 and g_switch_interval=48 this result in incoming and outgoing streams as depicted in Figure 17.

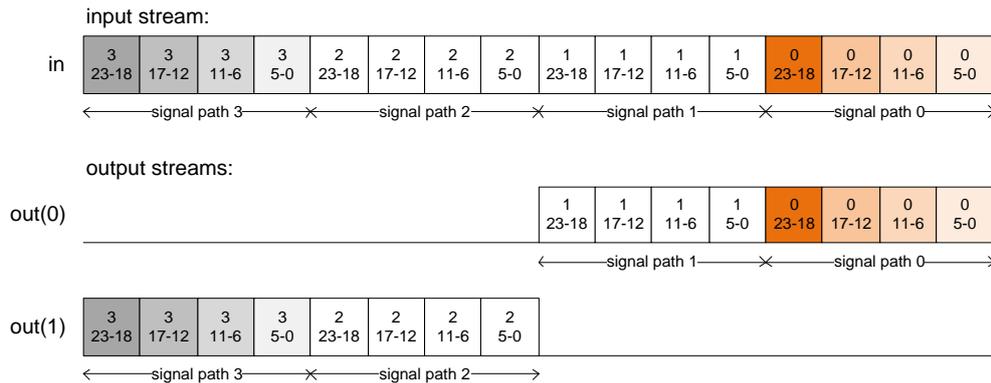


Figure 17 In- and output streams of bf_sw_node

The packets on all output streams are provided with the valid, sop and eop signals. Each output stream is connected to a bf_sw_inbuf.

5.1.4.2 bf_sw_inbuf

The bf_sw_inbuf is basically a fifo (from the common_lib) with some extra logic. The incoming data stream is written to the fifo where the valid signal is used to drive the fifos wr_req signal. The eop field of the incoming stream is used to notify the bf_sw_fifo_writer that there is a complete packet in the fifo. This is done via the packet_rdy signal. There are 32 bf_sw_inbuf units.

5.1.4.3 bf_sw_fifo_writer

The 32 bf_sw_fifo_writer units sort the data by reading data from the input fifos and then write the data in the reorder fifo. The parameters and interface signals of the bf_sw_fifo_writer are shown in the following tables:

Generic	Type	Value	Description
g_nof_input_streams	POSITIVE	4	The number of input streams that must be used to generate the reordered data stream. Used to estimate the data input span.
g_nof_signals_per_stream	POSITIVE	2	The number of signal paths represented in each input stream.
g_nof_subbands	POSITIVE	24	The number of subbands of each signal path.
g_nof_bf_units	POSITIVE	4	The total number of bf_units.
g_bf_unit_nr	NATURAL	0,1,2 or 3	This value is used for the timing of the fifo_rd_req and the fifo_wr_req signals.
g_in_dat_w	POSITIVE	16	The data width of the input and output data.

Table 13 Parameters of the bf_sw_fifo_writer

Interface	Type	Size or Span	Description
data_in	std_logic_vector	g_nof_input_streams* 2* g_in_dat_w	The data input span in which g_nof_input_streams(=4) datastreams are placed.
packet_rdy_in	std_logic	na	Input signal that is asserted when there is a complete packet available in the input fifos.
fifo_rd_req	std_logic	na	Read request signal to the input fifo.
fifo_wr_data	std_logic_vector	2* g_in_dat_w	Output data word to the reorder fifo.
fifo_wr_req	std_logic	na	Write request signal to the reorder fifo.
clk	std_logic	na	Datapath clock input.
rst	std_logic	na	Datapath reset input.

Table 14 Interface signals of bf_sw_fifo_writer

Doc.nr.: ASTRON-RP-1291
 Rev.: 0.4
 Date: 04-04-2012
 Class.: Public

Now every `bf_sw_fifo_writer` is responsible for generating one stream of data. This stream is composed out of slices of data that come from four consecutive input streams. This is best illustrated by viewing the content of four input fifos and the output of the first four `bf_sw_fifo_writer` units in Figure 18. The first four `bf_sw_fifo_writer` units all use different slices of data from the first four `bf_sw_inbuf` fifos.

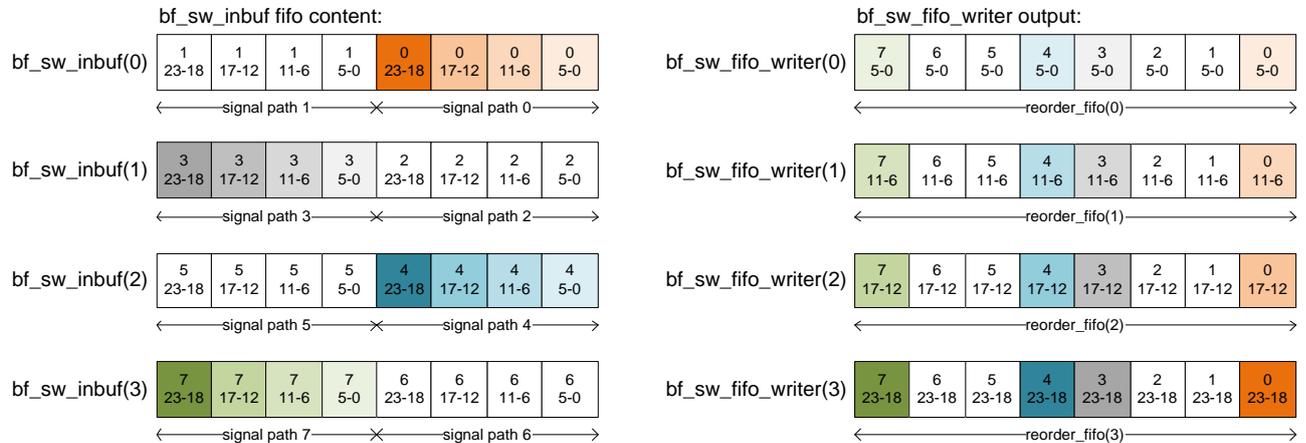


Figure 18 Content of input fifo and output of `bf_sw_fifo_writer`

In order to generate the stream for the reorder fifo the input fifos are read one by one in serial. This is done by asserting the `fifo_rd_req` signals one by one. First the data of `bf_sw_inbuf(0)` is read and written to the applicable `reorder_fifos`, then the `bf_sw_inbuf(1)` is read and distributed to the `reorder_fifos`, etc. An overview of the assertions of the `fifo_rd_req` and the `fifo_wr_req` signals is shown in Figure 19. This figure also shows the slices of data that are sent to the reorder fifo.

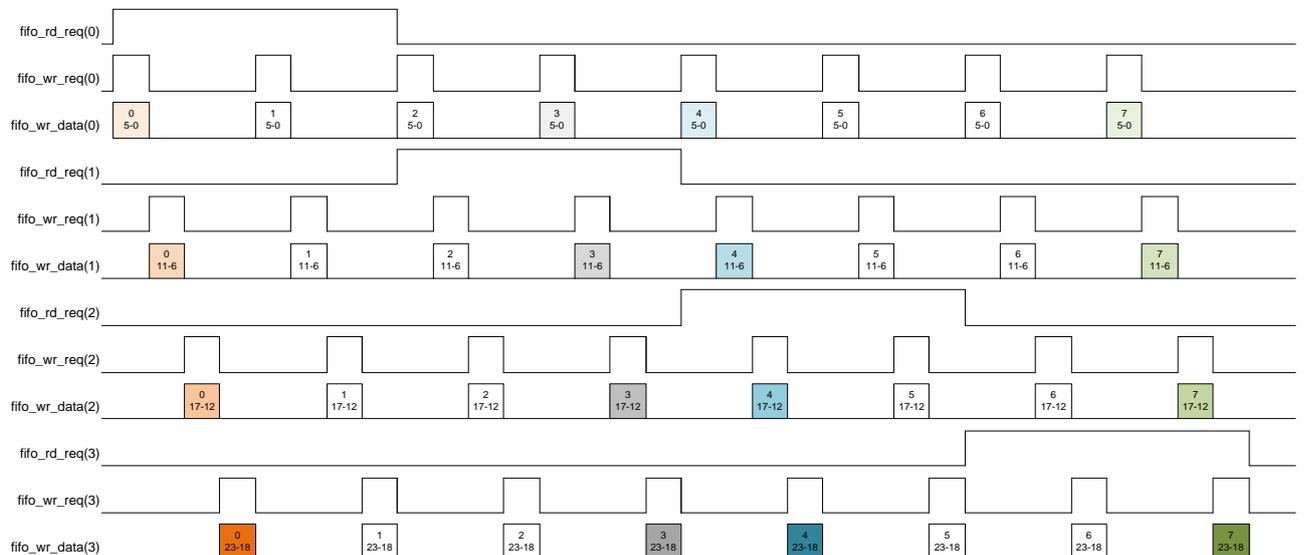


Figure 19 Assertion of `fifo_rd_req` and `fifo_wr_req`

As soon as the last data of the current packet is written to the `reorder_fifo` the `bf_sw_fifo_writer` unit notifies the `bf_sw_fifo_reader` by asserting the `packet_rdy`.

5.1.4.4 reorder_fifo

The reorder fifo is a single clock fifo (common_fifo_sc) that finds its origin in the common_lib, see [4]. The fifo is parameterized as follows:

- data width = 2*in_dat_w (=32)
- fifo depth = 6 subbands*8 signal paths (=48 words)

5.1.4.5 bf_sw_fifo_reader

The bf_sw_reader reads the data from the reorder fifo and generates a sosi stream that feeds the bf_sw_distribute unit. The unit also fills in the channel field of the sosi stream. The parameters that configure the bf_sw_fifo_reader are listed in Table 15 and the interface signals are listed in

Generic	Type	Value	Description
g_nof_input_streams	POSITIVE	4	The number of input streams that were used to generate the input stream for a single bf_sw_fifo_reader. Used to estimate the number of signal paths that are in the input stream.
g_nof_signals_per_stream	POSITIVE	2	The number of signals in the streams that were used to generate the input stream for a single bf_sw_fifo_reader. Value is used in combination with g_nof_input_streams to estimate the number of signal paths that are in the input stream.
g_nof_subbands	POSITIVE	24	The number of subbands of each signal path.
g_nof_bf_units	POSITIVE	4	The total number of bf_units.
g_in_dat_w	POSITIVE	16	The data width of the input and output data.

Table 15 Parameters of bf_sw_fifo_reader

Interface	Type	Size or Span	Description
packet_rdy_in	std_logic	na	Input signal that is asserted when there is a complete packet available in the reorder fifos.
fifo_rd_req	std_logic	na	Read request signal to the reorder fifo.
fifo_rd_data	std_logic_vector	2* g_in_dat_w	Read data from the reorder fifo.
out_sosi	t_dp_sosi	na	Streaming output that contains data and channel numbers.
clk	std_logic	na	Datapath clock input.
rst	std_logic	na	Datapath reset input.

Table 16 Interface signals of the bf_sw_fifo_reader

When the bf_sw_fifo_reader receives the packet_rdy signal from the bf_sw_fifo_writer unit it will start reading out the data from the reorder_fifo and send it to the connected bf_sw_distribute units. When doing that it also adds a number to the channel field of the sosi record. The channel number will be used by the bf_sw_distribute unit to determine to which output the data has to be transported. The data will be labelled with a channel number according to the example in Figure 20.

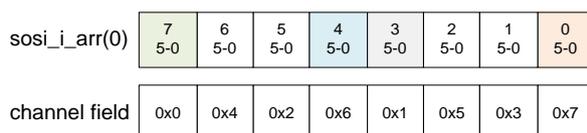


Figure 20 Channel labels

The reason for the non-incremental sequence of the channel numbers is that the bf_sw_distribute unit is based on single bit encoding which is explained in detail in the next paragraph.

5.1.4.6 *bf_sw_distribute*

The *bf_sw_distribute* unit splits an incoming datastream into multiple output streams based on the value that is represented in the channel field. The recursive architecture of the *bf_sw_distribute* unit creates a tree-like structure based on a fundamental 1-input 2-output unit (distribution node). In this way it is possible to generate as much outputs as specified. The parameters that configure the *bf_sw_distribute* unit are listed in Table 17. The interface signals are listed in Table 18.

Generic	Type	Value	Description
<i>g_nof_outputs</i>	POSITIVE	8	This value defines the number of outputs of the distributor. Value must be a power of 2.

Table 17 Parameters of the *bf_sw_distribute* unit

Interface	Type	Size or Span	Description
<i>in_sosi</i>	<i>t_dp_sosi</i>	na	Streaming input interface. Contains data with channel field numbers.
<i>out_sosi_arr</i>	<i>t_dp_sosi_arr</i>	<i>g_nof_outputs</i>	Array of streaming outputs. The channel field numbers are not present anymore.
<i>clk</i>	<i>std_logic</i>	na	Datapath clock input.
<i>rst</i>	<i>std_logic</i>	na	Datapath reset input.

Table 18 Interface signals of *bf_sw_distribute* unit

Figure 21 shows a schematic overview of an 8-output distribution tree with two routing examples. Every block can be considered as a distribution node. The values of the routing examples are binary represented. When a data sample enters a node (valid = high) the node reads the least significant bit of the channel field to determine which output should be selected. Before actually sending the data, the node bitshifts the channel field one value to the right. A logic '0' is shifted in on the left side. Now the next node will route the data further based on this "new" least significant bit in the channel field.

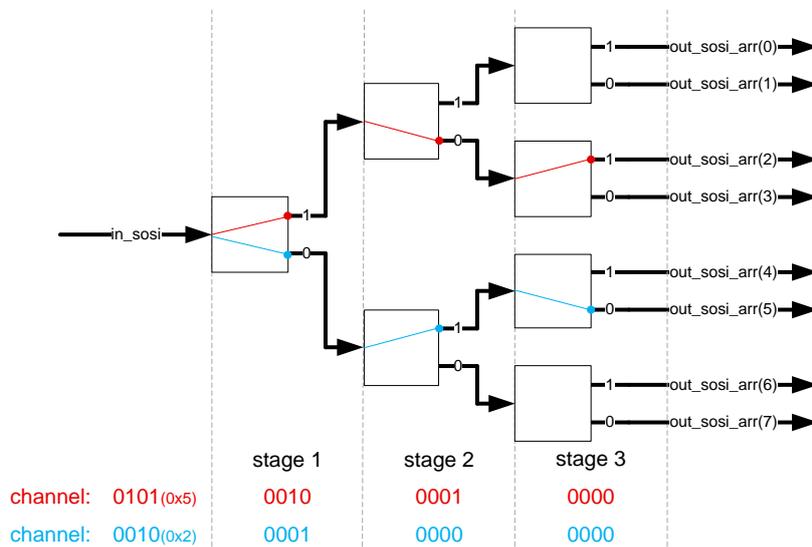


Figure 21 8-output *bf_sw_distribute* tree

5.2 bf_unit

A single bf_unit consists roughly out of two stages and a control process as shown in Figure 12. The first stage contains weight memories, input fifo's and the complex multipliers. The second stage contains the adder tree, packetizer, quantizer and beamlet statistics unit.

5.2.1 bf_unit first stage

A detailed schematic overview of the first stage is shown in Figure 22. A generate statement is used to instantiate this sub-design for every input path (nof_signal_paths).

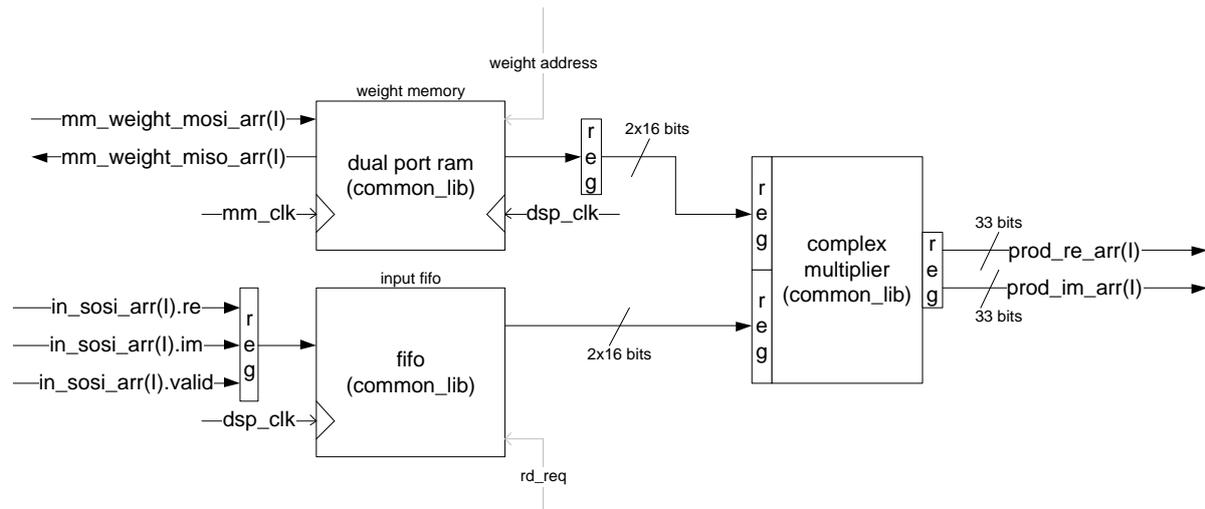


Figure 22 bf_unit first stage schematic

5.2.1.1 Weight memory

The weight memory is a dual ported ram block (common_ram_crw_crw) that is instantiated from the common_lib, see [4]. The size of the ram is defined as follows:

- address width = $\log_2(\text{nof_weights})$ (=8)
- data width = $2 * \text{in_weight_w}$ (=32) (since it are complex weights)

The left side is connected by an mm interface that can be connected to the host (microprocessor) whereas the right side of the ram is read by the bf_unit control process in order to feed the multiplier with the weight factors. The data that is read from the right side of the memory is registered in order to achieve better timing results during synthesis.

5.2.1.2 Input fifo

The input fifo is a single clock fifo (common_fifo_sc) that finds its origin in the common_lib, see [4]. The fifo is parameterized as follows:

- data width = $2 * \text{in_dat_w}$ (=32)
- fifo depth = $2 * \text{nof_offsets}$ (=16) (it can hold at least two input packets)

Incoming data from the in_sosi_arr is written to the fifo after being registered. The bf_unit control process reads out the data by asserting the rd_req signal.

5.2.1.3 Complex multiplier

The complex multiplier (`common_complex_mult`) is also part of the `common_lib`. By selecting the “stratix4” architecture the complex multiplier is mapped on an embedded Stratix IV DSP block. The in- and output registers are also part of the embedded DSP blocks. The complex multiplier is configured as follows:

- `architecture = stratix4`
- `g_in_a_w = in_weight_w (=16)`
- `g_in_b_w = in_dat_w (=16)`
- `g_out_p_w = in_weight_w + in_dat_w + 1 (=33)`
- `g_conjugate = FALSE`
- `g_pipeline_input = 1`
- `g_pipeline_product = 0`
- `g_pipeline_adder = 1`
- `g_pipeline_output = 1`

Note: the +1 that defines the `g_out_p_w` is due to the addition that is part of the complex multiplier. The result of the complex multiplier is a complex product of 2x33 bit and that is fed to one of the inputs of the adder tree in the second stage of the `bf_unit`.

5.2.2 bf_unit second stage

The second stage of the `bf_unit` combines all the outputs of the first stage multipliers using a set of adder trees. Then the `bf_unit` control process creates a stream of packets of the output of the adders. Further down the stream there are two quantizers that quantize the beamlets for the quantized output and the beamlet statistics unit.

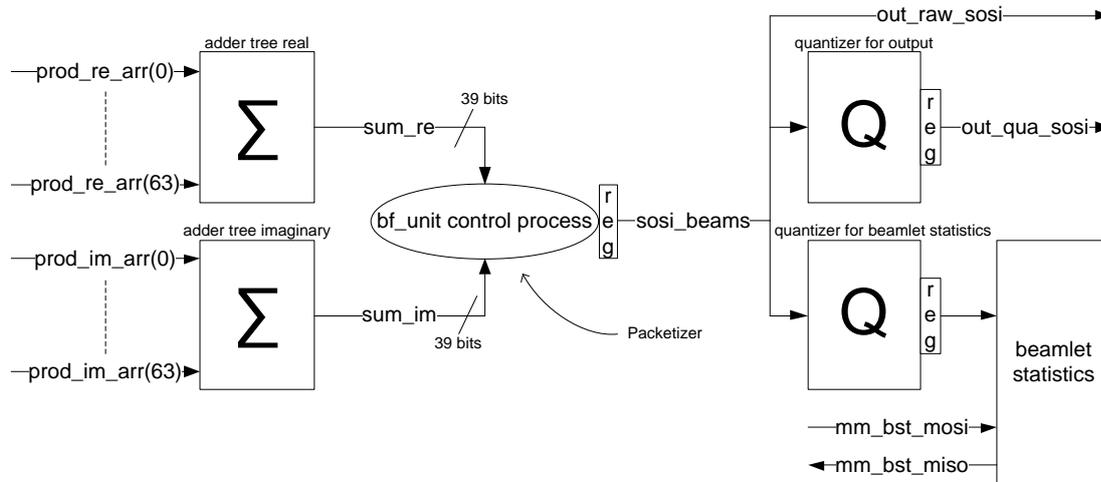


Figure 23 `bf_unit` second stage schematic

5.2.2.1 Adder tree

The adder tree is an adder structure that is based on an adder function that recalls itself, until the number of inputs is reduced to two. An adder tree for eight inputs results in a 3-stage tree as shown in Figure 24. Note that the output of each adder is registered. For the Apertif system the number of inputs is 64 (`nof_signal_paths`) which results in a 6-stage tree. The bitgrowth due to these adders is defined as $\log_2(64)=6$. Therefore the result of the adder tree is $33+6=39$ bits. The adder tree is located in the `common_lib` (see [4]) and is called `common_adder_tree`. A total of two adder trees are used to summarize both the real and imaginary part.

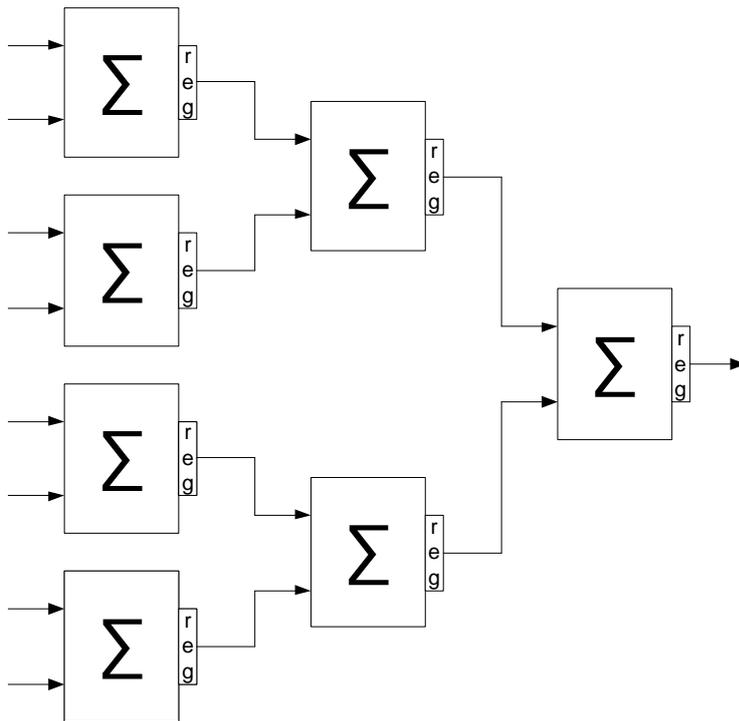


Figure 24 8-input adder tree

5.2.2.2 Packetizer

The `bf_unit` control process takes care of packetizing the adder results into a `sosi` stream. This packetizing consists of adding the `sync`, `sop`, `eop`, `valid`, `bsn` and error signals to the data. The format of the stream of packets that is created is displayed in Figure 25.

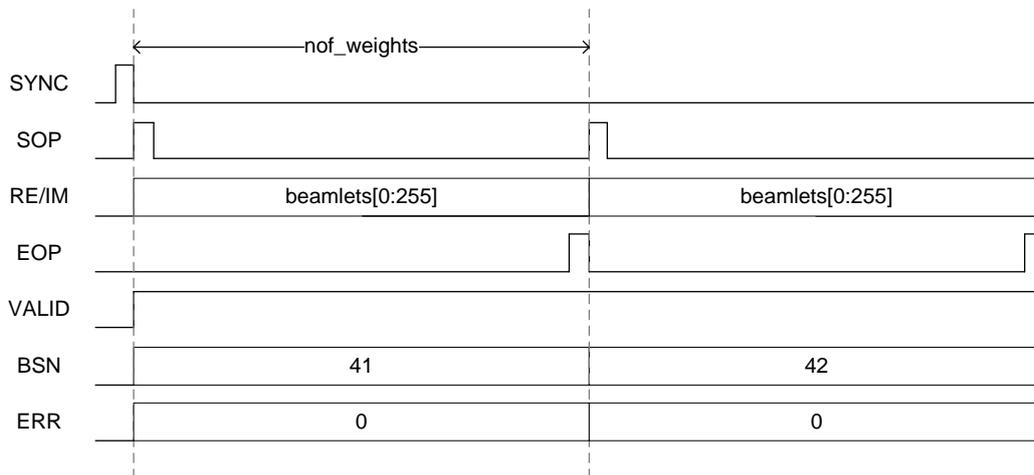


Figure 25 Beamlets packet format

The `sync`, `bsn` and `err` values are derived from the incoming data packets that contain the subband samples.

5.2.2.3 Quantizer

The quantizer is not yet fully implemented.

5.2.2.4 Beamlet statistics

The beamlet statistics unit calculates the power of each beamlet and integrates this over a sync period. After each sync period the accumulated powers are written to a set of registers that can be read via the mm_bst_mosi interface. The beamlet statistics module is reused from the Lofar project and is located in the st_lib, see [5] for more info.

5.2.3 bf_unit control process

This paragraph describes the control process of the bf_unit. The bf_unit control process is responsible for addressing the weight factors memory, reading out the input fifos and packetizing the beamlets.

5.2.3.1 bf_unit control ports

For a good understanding of the bf_unit control process it is necessary to have a good overview of the in- and output ports. These ports and their description are listed in Table 19.

Interface	In/Out	Type	Size or Span	Description
ctrl_sosi	In	t_dp_sosi	na	A streaming interface that is used to capture the bsn, err and sync field from. This input is connected to one of the input streams that also goes into the bf_switch.
out_sosi	Out	t_dp_sosi	na	A streaming sosi interface that contains the packets with beamlets.
in_sum_re	In	std_logic_vector	out_w	The real part of the beamformer result.
im_sum_im	In	std_logic_vector	out_w	The imaginary part of the beamformer.
weight_addr	Out	std_logic_vector	log2(nof_weights)	Signal used to address the memories with the weight factors.
rd_req	Out	std_logic	na	Read request signal that is connected to the input fifos.
usedw_fifo	In	std_logic_vector	log2(fifo_depth)	Input that represents the number of used words in the input fifo.
fifo_full	In	std_logic	na	The fifo full signal of one of the input fifos.
mm_offsets_mosi	In	t_mem_mosi		A mosi interface to write and read the offsets registers. The offset registers specify the offsets for the input data.
mm_offsets_miso	Out	t_mem_miso		A miso interface for reading back the offset registers.
dp_clk	In	std_logic	na	Datapath clock
dp_rst	In	std_logic	na	Datapath reset
mm_clk	In	std_logic	na	Memory mapped interface clock
mm_rst	In	std_logic	na	Memory mapped interface reset

Table 19 bf_unit control ports

5.2.3.2 State machine

All the control tasks are accomplished by a simple state-machine that is shown in Figure 26.

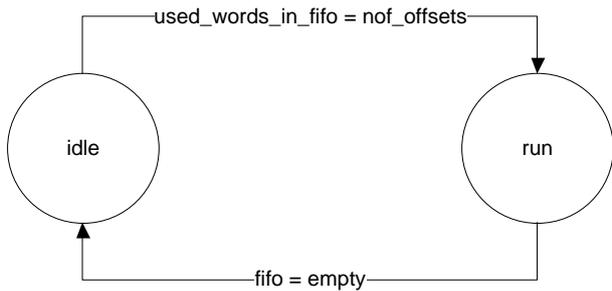


Figure 26 bf_unit control state-machine

In the idle-state the process waits until the input fifos are filled with `nof_offsets` samples. The run-state will be entered once the samples are there. The run-state performs a loop of `nof_weight` clock cycles that is called a beamlet loop. Within this loop all necessary signals are created according to the timing diagram as shown in Figure 27. An address counter runs from 0 to `nof_weights(255)` for addressing all the weight factors. When the address counter reaches one of the specified offset values the `rd_req` signal is asserted in order to read a new sample from the input fifo. If there is no data in the input fifo anymore at the end of a beamlet loop, the state-machine will return to the idle state. Otherwise it will immediately continue with a new beamlet loop.

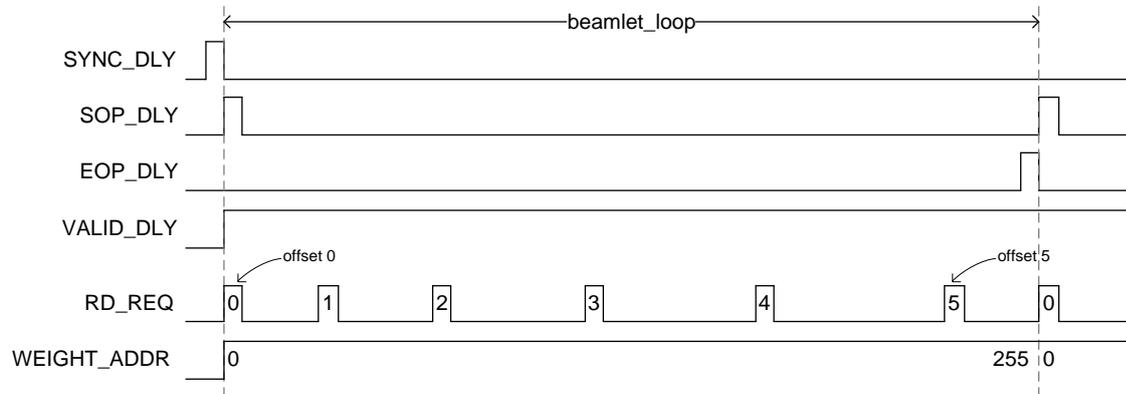


Figure 27 Timing diagram bf_unit control

5.2.3.3 Sop, eop and valid

The `sop_dly`, `eop_dly` and `valid_dly` signals are feeding shift registers that compensate for the latency that is introduced by the different stages in the datapath. This is shown in Figure 28. All three signals are delayed in such a way that they are proper aligned with the results of the adder trees.

5.2.3.4 Sync, bsn and err fields

For the sync signal a special sync-detection mechanism is implemented that is active in both the idle- and the run-state. The sync signal of the `ctrl_sosi` input is monitored and captured if asserted. When a new beamlet loop starts and a sync was detected then the `sync_dly` signal will be asserted as well. The `bsn` and `err` fields from the `ctrl_sosi` input are both written to a fifo where the `ctrl_sosi.sop` signal is used as `wr_req`. The `bsn` and `err` values are read back from the fifo by the for last register of the `sop` shift register. This is also displayed in Figure 28.

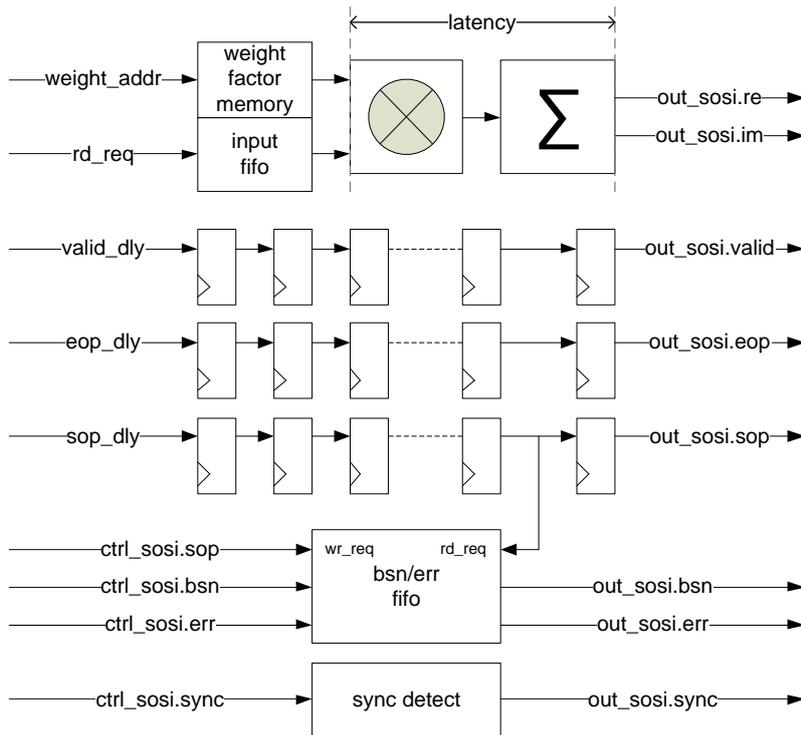


Figure 28 Shift register for latency compensation

5.2.4 Multiplexing mm interfaces

In order to avoid the usage of large arrays of mm interfaces in the port definition of design units a multiplexer is created that maps an array of mm interfaces to a single mm interfaces. The unit is called `common_mem_mux` and is located in the `common_lib` see [4]. Its basic functionality is depicted in Figure 29.



Figure 29 common_mem_mux

The `common_mem_mux` is parameterized by two generics:

- `g_nof_mosi`: the number of elements in the array
- `g_mult_addr_w`: the address width of a single element in the array

The `common_mem_mux` unit is used at several places in the `bf` module in order to distribute the mm interfaces in an efficient way through the design. Figure 30 shows how this is done in the `bf` module and in the `bf_unit` for the `mm_weight_mosi` interface.

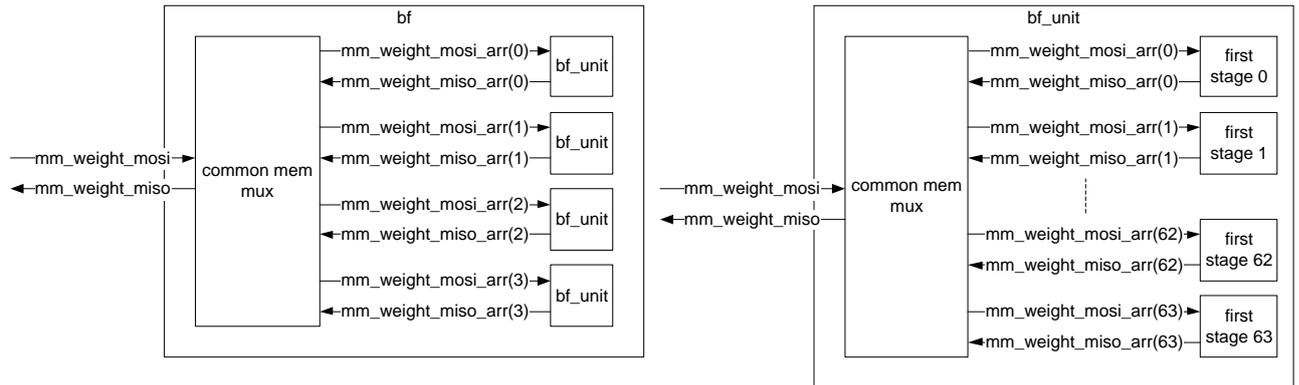


Figure 30 Usage of common_mem_mux in bf module

5.3 Bitwidths, bitgrowth and quantization

The bit-width of the inputs, outputs and several intermediate signals in the data path of a bf_unit is shown in Figure 31. Quantization is only performed at the end of the chain.

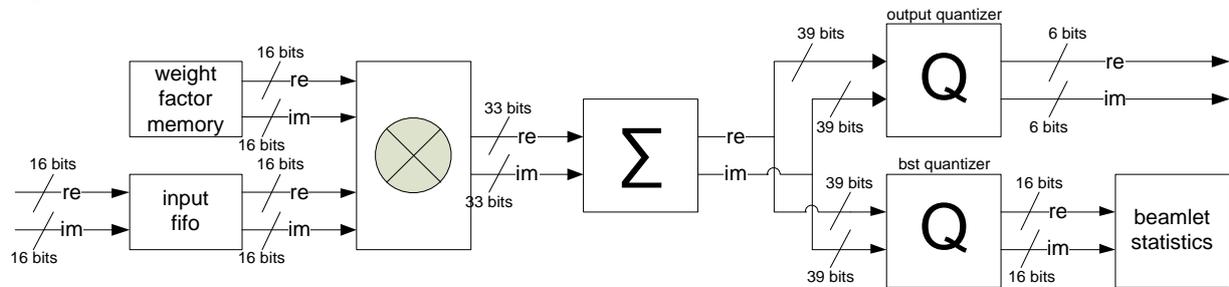


Figure 31 Overview of bitwidths

The output quantizer selects a configurable slice of 6 bits of the incoming 39 bits to send to the output. For the beamlet statistic unit the data must be resized to maximum 18 bits. The bst quantizer selects a configurable slice of 16 bits out of the 39 bits input.

6 Verification

All testbench related files can be found in [8]. A dedicated package is created that holds procedures and definitions that are aimed to be reused in the testbenches. The package is called `tb_bf_pkg.vhd`.

6.1 `tb_bf_unit`

The `bf_unit` module is verified in the `tb_bf_unit` testbench that can be found in [8]. An overview of the testbench is given in Figure 32. The testbench uses a set of processes that provide the DUT with stimuli. The stimuli are also offered to the verification process that calculates the reference data in parallel to the DUT. The tester process checks if the DUT output is the same as the calculated reference data.

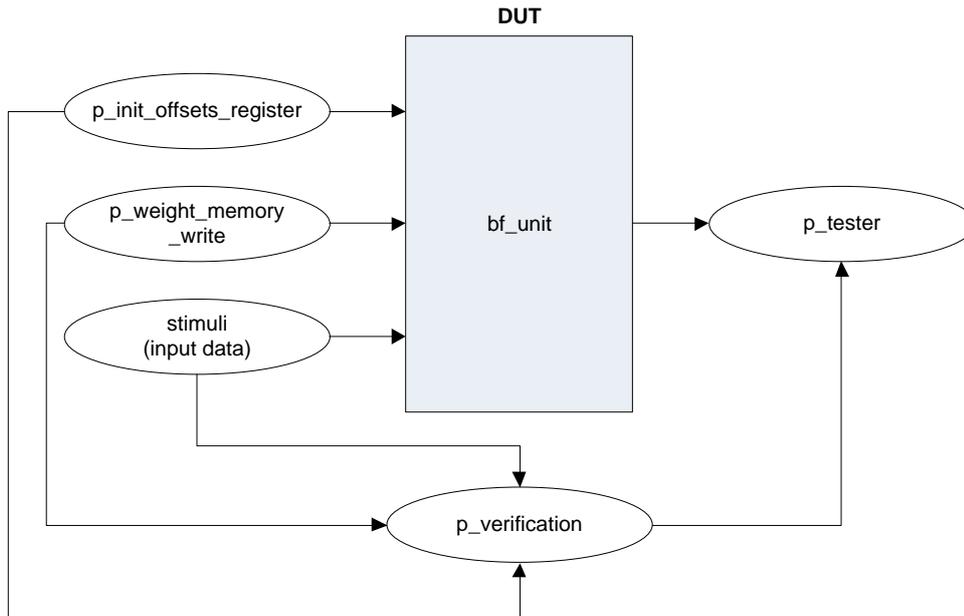


Figure 32 `bf_unit` testbench

6.1.1 `fifo overflow and underflow check`

Assert statements are included in the source code of the `bf_unit` to check if the input fifo suffers from overflow or underflow. The assert statements check at any time if the fifo is read when it is empty or if it is written while it is full. In case one of these rules is broken a message will be displayed.

6.1.2 `p_init_offsets_register`

This process writes the offsets to the DUT using the `mm_offsets_mosi` interface and stores the offsets also in an array that is used by the verification process. The complete functionality is captured in a procedure (`proc_bf_unit_init_offsets_register`) that can be found in the `tb_bf_pkg.vhd` file.

6.1.3 `p_weight_memory_write`

The next process reads a list of weight factors from a file and writes these weight factors to the weight memory. The file-access procedures that are used are imported from the `tb_common_pkg` from the `common_lib`. Every signalpath in the beamformer receives the same set of (`nof_weights`) weight factors.

6.1.4 `Stimuli (input_data)`

The generation of input data is based on the `proc_common_gen_data` procedure from the `tb_common_pkg`. To every input the same data stream is offered. The data stream is started when the weights have been written. The data consists of incremental values.

6.1.5 p_verification

The p_verification process performs the same algorithm that is placed in the DUT, based on the same weights, offsets and data input. Most of the calculations are performed using variables in order to optimize simulation time. To synchronize the output stream of the verification process with the DUT stream a number of pipeline stages is inserted. The output of the verification process is a stream of beamlets that should be equal to the output of the DUT.

6.1.6 p_tester

The p_tester process compares the beamlets stream of the DUT with the stream of the verification process. In case they are unequal a message will be displayed.

6.2 tb_bf

The bf module is verified with the tb_bf testbench. It shows a lot of similarities with the tb_bf_unit and some of the procedures are reused from the tb_bf_unit. The tb_bf testbench writes offsets to the offset registers and the weight factors to the weight factor memory. Then it will start sending data that represents the input streams that feed the bf_switch that is part of the bf module. A verification process runs in parallel that calculates the expected beamlets data and the expected beamlet statistics data. Another process reads out the beamlet statistics. As well the beamlet streams from the bf module and the beamlet statistics are sent to the testers to be checked. Most of the processes are simplified to procedures that can be found in the tb_bf_pkg.vhd file.

Once the simulation is loaded in ModelSim it can be started by typing "run -all". The testbench will run and stop automatically when finished.

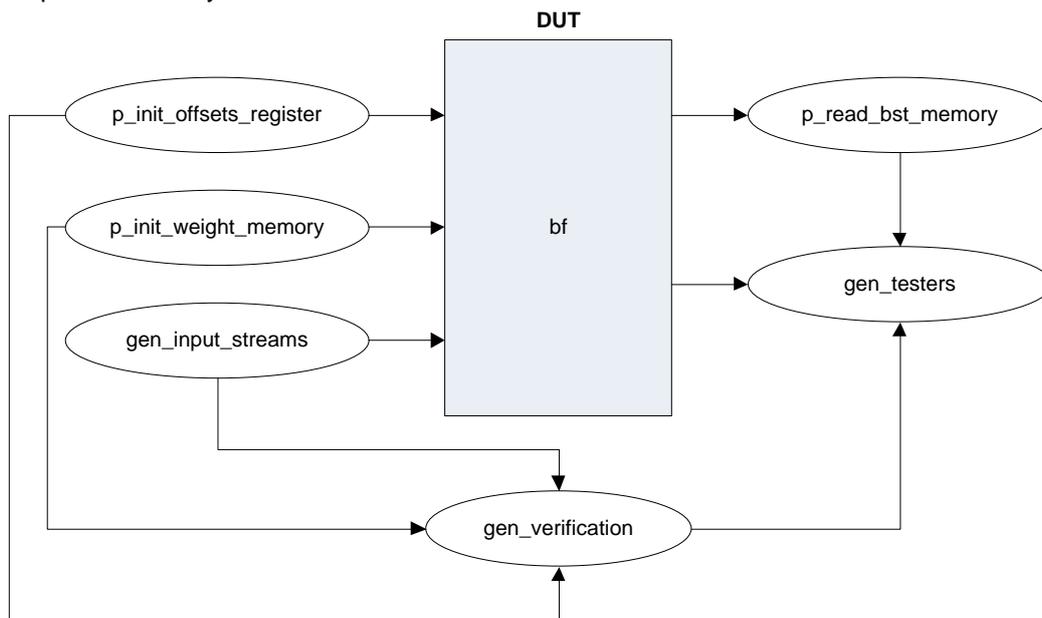


Figure 33 bf testbench

6.2.1 p_init_offsets_register

This process is responsible for sending the offsets to all the bf_units. Since there are multiple instances of the bf_unit in the bf module a 2-d array is created holding the offsets for all instances. This array is parsed to the verification process.

6.2.2 p_init_weight_memory

The process reads weight factors from a file and writes them to the weight factors memory. Since the mm_weight_mosi interface combines all the weight factor memories the writing of all weights is a very time consuming exercise in simulation. Therefore the process is enabled or disabled using the g_bf_write_weights

generic. Only when this generic is set to true the writing of weight factors will be simulated. When it is set to false the initial content of the weight factors memory will be used in the simulation.

6.2.3 gen_input_streams

For every input stream a unique data packet is composed in order to be able to trace the data more easily in the simulation results. The data is read from a file (in_data.dat) that contains 128 columns and 24 rows where every column duo represents a complex data value of a signal path and each row represents a subband. The number of composed packets that are sent to the DUT can be configured using the following constants:

- c_nof_sync = defines the number of sync intervals that should be send
- c_nof_accum_per_sync = defines the number of packets that must be send in one sync period

The format of the packets matches (of course) the required format as specified in paragraph 2.3.1. When all the specified packets are sent the simulation will stop automatically.

6.2.4 p_read_bst_memory

Every time a sync signal is detected in the sosi output stream this process reads the beamlet statistics results from the DUT and stores it in a 2d-array. This array is used by the tester process for verification. Reading out all the statistics is a time consuming task and therefor it is important that the simulation is long enough. The length of the simulation is determined by the values of c_nof_sync and c_nof_accum_per_sync.

6.2.5 gen_verification

The verification process calls the verification procedure (proc_bf_unit_expected_output) for each bf_unit and determines the expected output of all bf_units. It calculates the expected beamlet data as well as the expected beamlet statistics data. The process reads the data and weights from the files that are also used for the weight and data stimuli. Both expected data streams are offered to the tester process.

6.2.6 gen_testers

Two procedures are used for testing the output of the DUT with the calculated expected data. the proc_bf_unit_beamlet_data_tester procedure checks the beamlet data and the proc_bf_unit_beamlet_statistics_tester checks the beamlet statistics. Both procedures show a warning when

7 FN_BF Reference design

A reference design called `fn_bf` is made in order to test and validate the `bf` module. This chapter describes this design and the peripherals that are used to perform the validation. Also the software and python-scripts that are used for validation are explained here.

7.1 Design

The `fn_bf` design consists of a SOPC system holding a NIOS II processor that connects to the `ctrl_unb_common` unit and the `node_fn_bf` unit via `mm` interfaces. The `ctrl_unb_common` unit contains basic peripherals like the ethernet interface, system info, I2C sensor access and a PLL for clock generation. The `node_fn_bf` unit is build out of a `bf` module connected to a `mms_diag_block_gen` unit (see [6] for more information). The `mms_diag_block_gen` is a waveform generator that provides the `bf` module with input data on the `in_sosi_arr` inputs. The `bf` module is instantiated with four `bf_units` (`nof_bf_units = 4`) and 16 input streams (`nof_input_streams = 16`). All source and testbench files can be found in [9] .

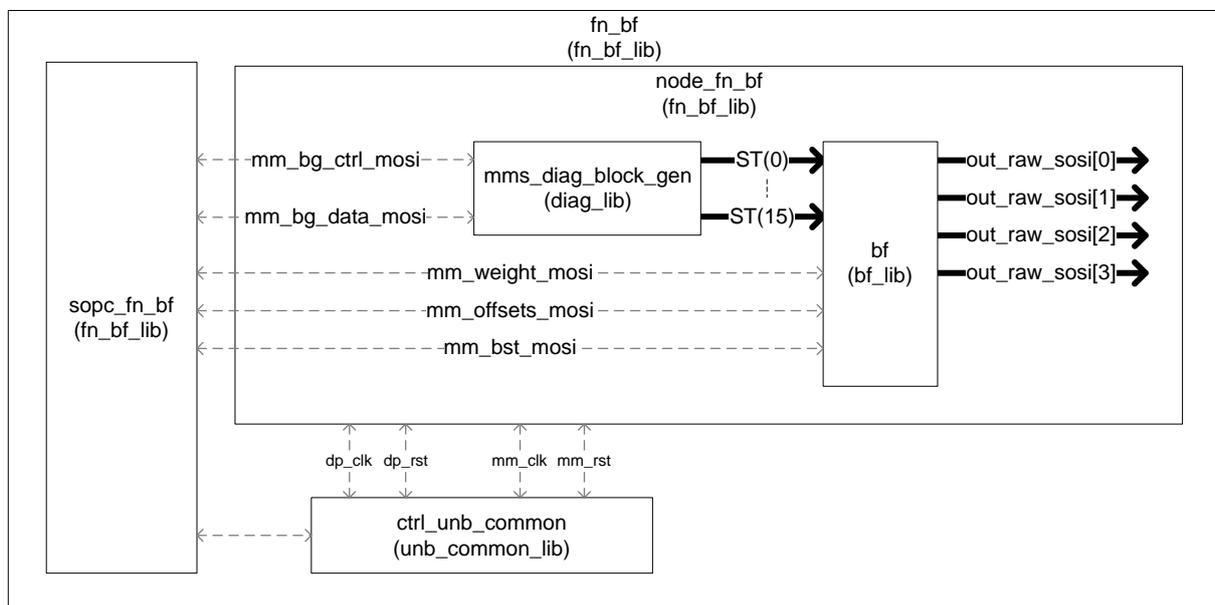


Figure 34: Design: `fn_bf`

7.2 Verification

Two testbenches are created to verify the correctness of the reference design. The first testbench simulates only the node design (`tb_node_fn_bf.vhd`) and the second testbench verifies the complete design, including the SOPC system (`tb_fn_bf.vhd`).

7.2.1 `tb_node_fn_bf`

The `tb_node_fn_bf` testbench is very similar to the `tb_bf` as described in paragraph 6.2. The only difference is that the input data is now generated with the `block_gen` unit from the `diag` library. Figure 35 shows an overview of the `tb_node_fn_bf`. All processes are reused from the `tb_bf`, except the processes `p_init_waveforms_memory` and `p_control_input_streams`.

7.2.1.1 `p_init_waveforms_memory`

This process reads data from an input file and sends it via the `mm_bg_data_mosi` interface to the waveform memories of the block generator. The data is sent in such a way that the block generators simulate the packets as described in 2.3.1.

7.2.1.2 p_control_input_streams

The p_control_input_streams process configures the block generator by sending the configuration data via the mm_bg_ctrl_mosi interface. After the block generator is enabled it will wait for c_dp_rof_cycles clock cycles before the block generator is disabled again which also marks the end of the simulation.

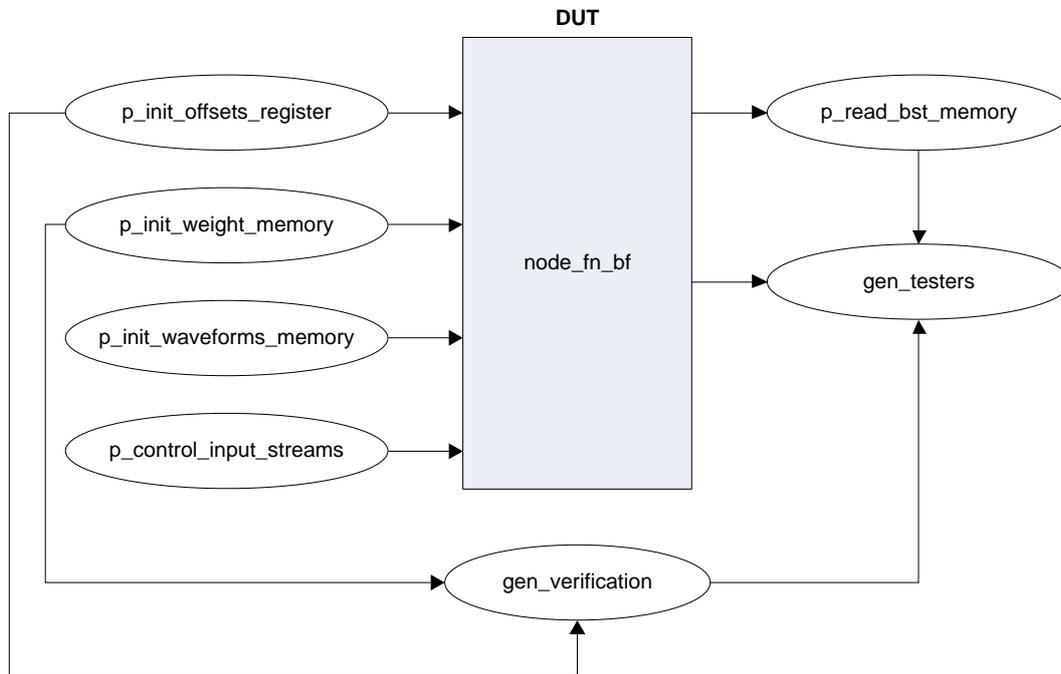


Figure 35 node_fn_bf testbench

7.2.2 tb_fn_bf

A testbench that simulates the complete design is called tb_fn_bf. This testbench is very simple as shown in Figure 36. The stimuli consists a two clock signals and the static assertion of the few input signals.

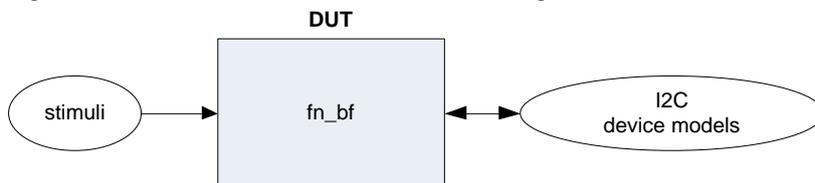


Figure 36 fn_bf testbench

A few I2C models are included in the simulation to facilitate the reading out of the temperatures and board voltages. The software that runs on the NIOS II processor can be simulated as well.

7.3 Software

For test and verification purposes a small application software has been written that runs on the NIOS II processor in the fn_bf design. The software is used in the tb_fn_bf simulation to check if reading and writing to the registers are executed as expected. The software was also used in the early stages of debugging when the python scripts were not yet available.

The application performs the following tasks:

- Writing the offset registers.
- Configuring (and enabling) the block generator.
- Reading out the beamlet statistics and print to screen in a loop.

The main.c file can be found in [9].

8 Synthesis and Place & Route

For synthesis and place & route the Altera EP4SGX230KF40C2 device is selected. This is the FPGA type that is currently placed on the UniBoards.

8.1 Resources and F_{max}

Table 20 gives an overview of the resource usage of some beamformer elements and the reference design (fn_bf). The percentages relate to the total resources available in the target device (EP4SGX230KF40C2). Also the achieved maximum frequency of the datapath clock of each (sub)-design is listed. The required frequency for the datapath clock is 200 MHz.

	bf_unit	bf_switch (a_sort_distribute)	bf	fn_bf
Registers	9354	31440	68539(38%)	64946(36%)
ALMs	5648(6%)	19184(21%)	46359(51%)	48221(53%)
M9K blocks	200(16%)	64(5%)	864(70%)	918
M144K blocks	0	0	0	8
Block Mem Bits	1,843Mbit	589Kbit	7,962Mbit	9,639Mbit
DSP blocks	258(20%)	0	1032(80%)	1032
PLLs	0	0	0	3
(Virtual) Pins	32436	129474	11968	32
F_{max} Slow 85C	384 MHz	467 MHz	274 MHz	246 MHz
F_{max} Slow 0C	401 MHz	486 MHz	286 MHz	261 MHz

Table 20 Overview resource usage

8.2 Timing optimizations

In order to achieve the best timing results several optimization have been performed. This chapter will describe the optimizations in more detail.

8.2.1 Pipeline stages

From early timing analysis it was concluded that at some places in the design it would be beneficial for the timing to insert pipeline stages. Most pipelines stages are inserted in the bf_unit where some of the inputs and outputs of the sub block are registered as can be seen in Figure 22, Figure 23 and Figure 24. It is also highly recommended as stated in [11] to register input and output of each sub design. However insertion of multiple pipeline stages at the same location has not led to better timing results.

8.2.2 Optimizing switch architecture

The first approach of implementing the bf_switch led to an efficient functional design (the bf_switch_a_direct as described in 5.1.3) but the timing results were very poor. The bad timing results were due to the fact that the direct approach performs the order and distribute functionality in one step introducing a latency of only 3 clock cycles. Therefor a second approach was undertaken. Taken they architecture of the FPGA into account another implementation of the bf_switch was designed that separated the order and distribution functionality (see 5.1.4). This approach introduced more latency and pipelined stages and therefore also better timing results since the sorting and distribution is divided over multiple stages.

In case multiple streams of data are to be sorted it is recommended to find a sorting algorithm that allows the insertion of pipeline stages rather than trying to sort everything in one cycle.

8.2.3 Synthesis constraints

Another way to improve the timing of the bf design is adding constraints that influence the behaviour of the synthesis tool. During an early timing analysis it was detected that the read_req signal that drives the input fifo's of the bf_unit was the critical timing signal. The fanout of this signal was 64 since the bf_unit_control

module drives 64 input fifos. By forcing the fanout to a maximum of 4 the register that holds the read_req signal is automatically duplicated resulting in better timing results. The synthesis constraints are sourced as a tcl script in the Quartus project file.

8.2.4 LogicLock Regions

LogicLock Regions are used to force the place & route tool to place a sub part of the design to a restricted area of the FPGA. Using this technique allows you to tell the tool which parts of the design should be placed close to each other. In case of the fn_bf design the following approach was taken in order to determine the best LogicLock parameters:

- At first all four bf_units in the design are assigned to a new (unique) LogicLock Region.
- The LogicLock Region properties are set to : Size=Auto, State=Floating
- With these settings a place & route is performed.
- Quartus will parameterize the four LogicLock Regions during place & route.
- After completion the LogicLock Region parameters are copied to a tcl script file.
- The tcl file is then sourced in the Quartus project file.

So it is recommended to let Quartus determine the size and origin of the LogicLock Regions in a first run. When a significant change has been applied to the design it is worthy to let Quartus determine the size and origins of the LogicLocks Regions again.

9 Validation

Validation of the design is done using the reference design in combination with a set of Python scripts that run on a host PC. All validation is executed on a UniBoard.

9.1 Python

In order to validate the correct working of the beamformer a set of python scripts is written that enables communication of a host PC and the bf design.

9.1.1 pi_bf_bf.py

The Python file pi_bf_bf.py is located in [7] and represents the bf_bf peripheral. It contains a class (PiBfBf) with methods that facilitates the uploading of weight factors and the offset values.

9.1.2 tc_pi_fn_bf.py

The tc_pi_bf_bf.py file is located in [7] and contains a test case that describes a scenario, using several methods of the pi_bf_bf.py file. The scenario covers the following steps:

- Write all offsets (write_offsets)
- Read back the offsets (read_offsets)
- Write weights (write_weights)
- Read Weights (read_weights)

9.1.3 tc_pi_fn_bf.py

This script is created to validate the total operation of the fn_bf design. The script supports the `-rep` and `-n` arguments to facilitate repetitive testing. The steps that are performed in the loop are:

- Write settings to the block generator
- Create and write data based on the loop number to the waveform memory of the block generator
- Create and write weight factors based on the loop number to the weights memory
- Write offsets to all bf_units.
- Enable the block generator
- Calculate the reference values
- Read and verify the beamlet statistics (for `-n` times)
- Disable the block generator

10 Appendix – list of files

10.1 Firmware VHDL

All VHDL source files that are used for the bf design can be found in the following directory:

\$UNB/Firmware/dsp/bf/src/vhdl

The next table gives an overview of the VHDL source files:

VHDL File	Description
bf.vhd	Toplevel entity that instantiates the bf_switch and a number of bf_units.
bf_offsets_reg.vhd	Contains the register interface for the offsets.
bf_pkg.vhd	A package that defines some constants and types. It also contains the definition of the record used for the generics.
bf_quant.vhd	Source file for the quantizer unit.
bf_sw_distribute.vhd	Part of the bf_switch responsible for the distribution of the intermediate streams to the input fifos of the bf_units.
bf_sw_fifo_reader.vhd	Source file containing functionality that reads out the bf_switch fifo.
bf_sw_fifo_writer.vhd	Source file containing functionality that writes to the bf_switch fifo.
bf_sw_inbuf.vhd	Description of the input fifo of the bf_switch.
bf_sw_node.vhd	Contains the entity that performs the actual switching.
bf_sw_reorder.vhd	This is the reorder part of the switch functionality.
bf_switch.vhd	Entity definition of the switch.
bf_switch_a_direct.vhd	Architecture for the bf_switch.
bf_switch_a_nodes_bf_unit_signals.vhd	Architecture for the bf_switch.
bf_switch_a_nodes_signals_bf_unit.vhd	Architecture for the bf_switch.
bf_switch_a_sort_distribute.vhd	Architecture for the bf_switch.
bf_unit.vhd	Contains the design for a single bf_unit.
bf_unit_control.vhd	Source file containing the control process for the bf_unit.

10.2 Testbench

The testbench files for simulation are in the following directory:

\$UNB/Firmware/dsp/bf/tb/vhdl

10.3 Software

In addition, a C main program is located at:

\$UNB/Firmware/software/apps/fn_bf