Contents

1 Source Files Description **2**

2 Implementation Description **3**
2.1 Branch Distance Calculation (branch_distance.vhd) 4
2.2 Add, Compare and Select (acs.vhd) 5
2.3 RAM Control (ram_ctrl.vhd) 6
2.4 Traceback (traceback.vhd) 9
2.5 Reorder (reorder.vhd) 9
2.6 Recursion (recursion.vhd) 9
2.7 Top entity (dec_viterbi.vhd) 9

3 Testbench (tb_dec_viterbi.vhd) **10**
3.1 Testbench description 10
3.2 Running the Simulation 11

4 Version Information **12**
4.1 Product Version 12
4.2 Document Versions 12

1 Source Files Description

The directory structure of the Viterbi decoder core is shown in Table 1.1. Each decoder component (package or entity) is implemented in a single file. Package or entity name correspond to the filename without file ending (".vhd"). Furthermore, parameter definitions, type definitions, and helper functions use own files to guarantee a simple navigation through the source code.

Directory	Containing
/scripts	simulation scripts
/doc	datasheets
/packages	used packages, see Table 1.2
/src	component implementations, see Table 1.3
/testbench	testbench and test vectors

Table 1.1: Directory and file structure

Package	Brief description
pkg_param	Global parameters, parametrizing the core
pkg_param_derived	Global parameters derived from pkg_param
pkg_types	Global type definitions
pkg_helper	Global helper functions
pkg_trellis	Trellis helper functions
pkg_components	Component declarations of used entities

Table 1.2: List of packages used within decoder.

Entity	Type	Brief description
branch_distance	m	Branch distance calculation unit, see Section 2.1
acs	m	Add-Compare-Select unit, see Section 2.2
ram_ctrl	m	RAM Control for traceback, see Section 2.3
traceback	m	Traceback unit, see Section 2.4
reorder	m	Reordering unit for correct output sequence, see Section 2.5
dec_viterbi	m	Top entity of the Viterbi decoder, see Section 2.7
generic_sp_ram	m	Generic single port RAM description
recursion	o	Recursion unit for recursive convolutional codes, see Section 2.6

Table 1.3: List of entities within decoder, source files are either mandatory (m) or optional (o).

2 Implementation Description

The decoder architecture is designed for flexibility at design-time and at run-time. It is easily possible to improve some decoder units or add more units. Therefore it is possible to adapt the design for different needs, or add more functionality. In the following the decoders units are explained in more detail.

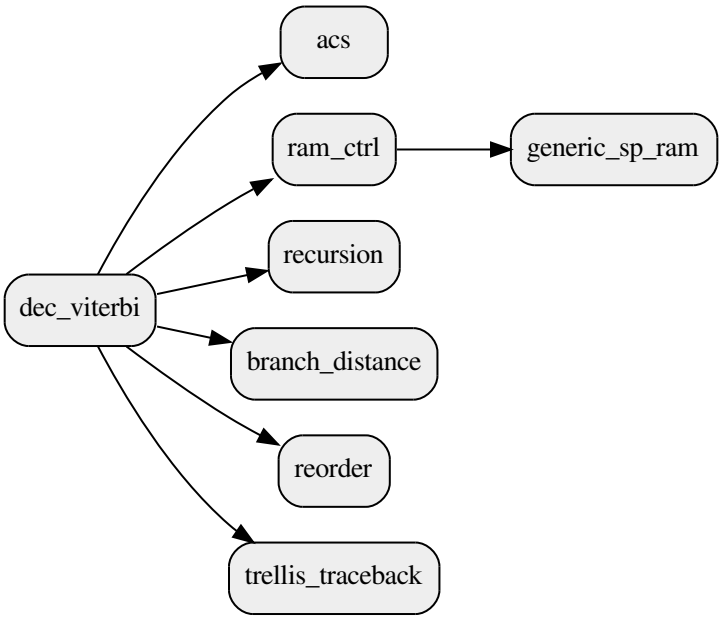

Figure 2.1: Entity instantiation hierarchy of the Viterbi decoder.

All units are connected via AXI4-Stream interfaces. Therefore modifying, exchanging or adding a unit is very natural.

2.1 Branch Distance Calculation (branch_distance.vhd)

This part computes all possible branch distances and does not depend on any other values or entity, but the input values. The branch distance is calculated by a euclidean distance. Thereby the incoming LLR value is added or subtracted to the overall distance. The parity LLR input values for one information bit are read and added or subtracted in parallel.

Depending on the code rate, 4 (code rate 1/2), 8 (code rate 1/3), or 16 (code rate 1/4) branch distances exist.

The branch distance calculation entity *branch_distance* is designed in a generic way such that it is adaptable to different code rates at design-time. It evaluates the generic *EDGE_WEIGHT* at compile time. This generic contains the number of branch distances to be calculated. If different sets of branch distances are required in order to support different polynomials and rates at run-time, it is possibility to instantiate more units and connect them in the top entity correctly to the *acs* entity.

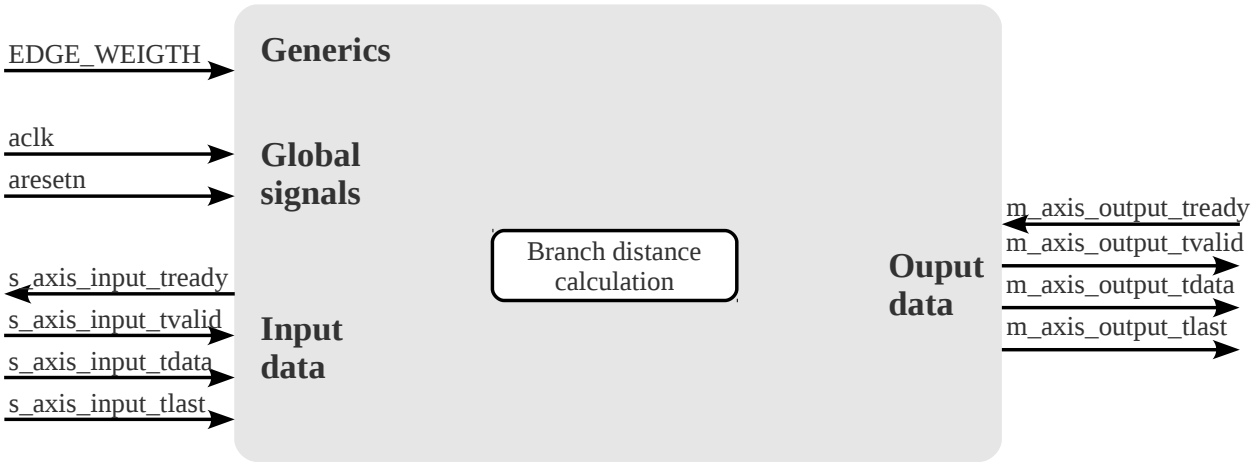

Figure 2.2: Interface of the branch distance calculation unit

2.2 Add, Compare and Select (acs.vhd)

The add, compare, and select (ACS) entity *acs* calculates the local most likely path, as described in the Viterbi algorithm. The ACS unit uses a recursion and therefore contains in most cases critical unit of the Viterbi decoder. Based on the Viterbi decoder each node of the trellis tree is represented as one ACS entity. Generating and connecting of the entities is done in the top entity *dec_viterbi*, as explained in Section 2.7.

Functionality

In order to implement one generic trellis node there are several operations which have to be performed. At first two incoming branch distances from a high and a low branch need to be added to the probability of the previous high and low node. The input data again is transmitted with an AXI4-Stream interface. Afterwards the high and the low probabilities are compared, which one is the more likely path. In this implementation the most likely path is the one with a higher numeric value (maximum search). At last, the probability of the more likely path is stored in an register within the ACS unit. Furthermore, the decision whether the high or the low branch is used is indicated on the output.

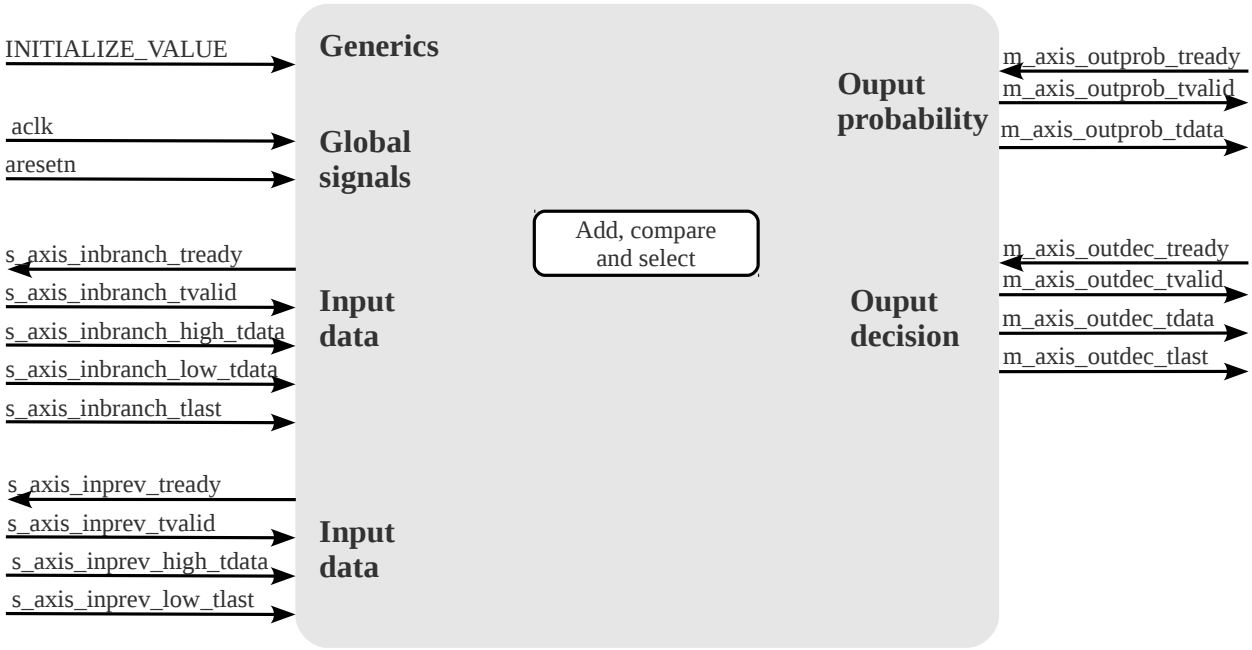

Figure 2.3: Interface of the ACS unit

Initialization

When a new block of parities arrives it is necessary to initialize the Viterbi trellis. Thereby all, but node zero are initialized with one quarter of the maximum value that can be represented with the current bit width. Node zero is initialized with zero. As a result only node zero is considered in the beginning and the trellis is built up. Because modulo normalization is used, it is possible to initialize the trellis with other values but it is not recommended.

The initialization itself is performed one cycle after the last signal from the branch distance calculation of Section 2.1 arrives.

2.3 RAM Control (ram_ctrl.vhd)

The data produced by the ACS units, see Section 2.2, needs to be stored for further processing. The *ram_ctrl* entity is used as a glue logic between ACS unit and traceback unit, which is used to perform the second part of the Viterbi algorithm. In Figure 2.4 the data accesses for one block are shown. Because of windowing, the results are stored into memory and read either one or two times from this memory for further processing. (once for acquitisition, once for traceback itself). Because of the parallel processing it is possible, that three memory accesses need to be performed during one clock cycle. An approach with four single port RAMs is used to handle these memory accesses, see Table 2.1.

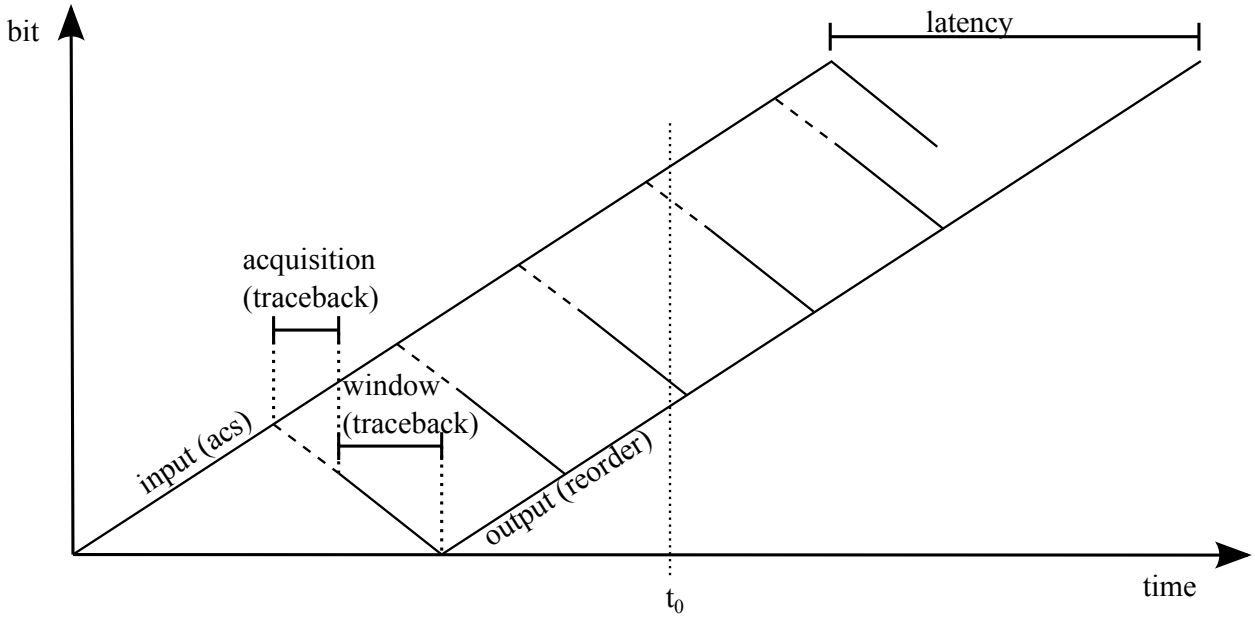

Figure 2.4: Data process diagram

General Controller

On the one hand using four RAMs results in a faster and more efficient design, but on the other hand memory management becomes more difficult. It is necessary to switch the RAM accesses at specific points in time. Every time the ACS unit has filled up a RAM with one window, data accesses from all units have to be switched. A RAM controller is used for this task.

The controller entity *ram_ctrl* is implemented as a finite state machine. It handles:

- Storage of configuration
- Waiting for a new block
- Beginning procedure, i.e., writing ACS results only
- Writing ACS results and reading for traceback
- Ending procedure, i.e., reading for traceback only

Therefore the state machine is implemented with five states. Each operation mode is handled in one state.

Cycle	RAM 1		RAM 2		RAM 3		RAM 4	
	Unit	Addr	Unit	Addr	Unit	Addr	Unit	Addr
1	write FWD	3						
2	write FWD	4						
3			write FWD	0				
4			write FWD	1				
5			write FWD	2				
6			write FWD	3				
7			write FWD	4				
8			read ACQ 1	4	write FWD	0		
9			read ACQ 1	3	write FWD	1		
10			read WIN 1	2	write FWD	2		
11			read WIN 1	1	write FWD	3		
12			read WIN 1	0	write FWD	4		
13	read WIN 1	4			read ACQ 2	4	write FWD	0
14	read WIN 1	3			read ACQ 2	3	write FWD	1
15					read WIN 2	2	write FWD	2
16					read WIN 2	1	write FWD	3
17					read WIN 2	0	write FWD	4
18	write FWD	0	read WIN 2	4			read ACQ 1	4
19	write FWD	1	read WIN 2	3			read ACQ 1	3
20	write FWD	2					read WIN 1	2
21	write FWD	3					read WIN 1	1
22	write FWD	4					read WIN 1	0
23	read ACQ 2	4	write FWD	0	read WIN 1	4		
24	read ACQ 2	3	write FWD	1	read WIN 1	3		
25	read WIN 2	2	write FWD	2				
26	read WIN 2	1	write FWD	3				
27	read WIN 2	0	write FWD	4				
28			read ACQ 1	4	write FWD	0	read WIN 2	4
29			read ACQ 1	3	write FWD	1	read WIN 2	3
30			read WIN 1	2	write FWD	2		
31			read WIN 1	1	write FWD	3		
32			read WIN 1	0	write FWD	4		
33	read WIN 1	4			read ACQ 2	4	write FWD	0
34	read WIN 1	3			read ACQ 2	3	write FWD	1

FWD = fowarding
WIN 1 = window 1; ACQ 1 = acquisition 1
WIN 2 = window 2; ACQ 2 = acquisition 2

Table 2.1: RAM usage for window length = 5 and acquisition length = 2

State: CONFIGURE

The *CONFIGURE* state is active when the decoder is ready for a new configuration. The configuration is received with AXI4-Stream within one cycle. Decoding can only start if a configuration is available.

State: START

When the configuration is received, the state machine waits for new parity data to arrive. It does take two cycles until a received parity block reaches the RAM control entity.

State: BEGINNING

Because of the four RAMs, the first decisions arriving from the ACS unit have to be processed differently. In Table 2.1 the RAM usage is shown. In first state the RAM is not fully written. Only one acquisition length is written to the first RAM to speed up and simplify further processing.

State: RUNNING

In main operation there are three accesses to the RAMs at one point of time, as was shown in Figure 2.4. In the implementation there are several pointers used. There are pointers for addressing a specific RAM, for requesting a specific RAM address, and because of delays for reading the outputs of a specific RAM. The reference point of time in this state is the time, when the ACS unit filled a RAM with one window length.

Further the entity signals to the traceback units if current data is in acquisition or traceback phase. For signaling the traceback unit if the data belongs to acquisition or traceback the tuser signals are used. Figure 2.5 shows the corresponding timing. The tuser signal *_window_tuser* is set to high when the acquisition

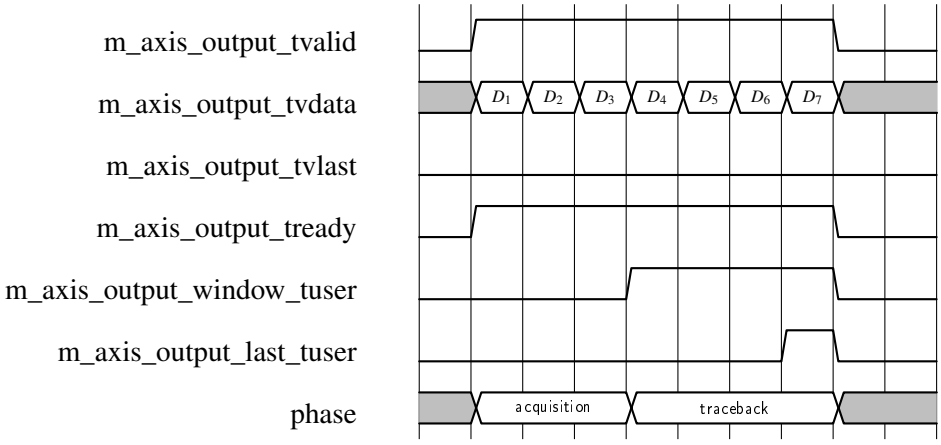

Figure 2.5: Timing from RAM control to one traceback unit

phase is done and the actual traceback starts. The tuser signal *_last_tuser* is set to high when the last data of a traceback arrives.

State: ENDING

When the ACS unit processed all incoming data of a block, the last state of the finite state machine is entered. The ending is handled separately since the processing no longer is synchronous to the ACS input. Further for the last traceback there is no acquisition used. This decoder only considers tailing, where the final traceback starting state zero is known. Tailing is applied in most common standards. When the last bit is sent to the traceback unit the state machine again will switch to *CONFIGURE*.

2.4 Traceback (traceback.vhd)

The traceback unit is required for the second part of the Viterbi algorithm. From a given starting state the decisions calculated by the ACS unit is processed backwards. As a result of this the output is in a reversed order. It is mentioned in Section 2.3 that currently only tailing is supported. Therefore the traceback always starts at the zero state of the trellis. Besides the starting state, there is no further information about the decoder configuration applied to the traceback unit.

Interface

The decoders configuration is implicitly given to the traceback with the tuser signals at the input. The timing is shown in Figure 2.5 and explained in Section 2.3. Since there is no explicit information about the decoder configuration available, the traceback uses this *_window_tuser* input signals to decide if the output is valid. The *_last_tuser* signal is used as an internal state reset and is forwarded to the output.

Implementation

The traceback is implemented with a shift register, which stores the current trellis state. The decision bit from the ACS unit is inserted into the shift register and the oldest bit is sent to the output. Resetting the starting state of the traceback is handled by the tuser signal, which indicates the last decision of a traceback length. Since the output of the traceback unit is in another order than the input was sent, reordering becomes necessary, see Section 2.5.

2.5 Reorder (reorder.vhd)

In order to resort the output of the traceback unit, the decoding results from the traceback units have to be stored. Shift registers are used to save the twisted decoding result. The traceback output is saved in a shift register. The content of the shift register is output in reversed order when the traceback unit finished one window. The reversed output begins right after the *_last_tuser* signal is asserted. Due to the resorting the latency of the decoder increases, which is shown in Figure 2.4. The distance between the input and output represents the overall latency.

2.6 Recursion (recursion.vhd)

The recursion entity only is added to the design if there is a feedback polynomial defined in *pkg_param.vhd*. The entity basically only consists of a shift register, with the length of ENCODER_MEMORY_DEPTH. The output of the reordering units is moved into this shift register. Based on the defined feedback polynomial, this shift register is convolved to the correct output. Using recursive convolutional codes is quite rare. If enabled, it will add another latency to the output of the decoder (equal to the encoder's memory depth).

2.7 Top entity (dec_viterbi.vhd)

In the top entity *dec_viterbi* the decoder units are connected. Furthermore the external AXI4-Stream control signals are handled here and the correct order of output bits is generated. Since the design applies parallel processing it is necessary to manage the data output behavior. It might be necessary to insert stalls in order to let other units finish their operation first. In order to handle this, the entity includes a process that is working with a semaphore. The semaphore guarantees that only one parallel working unit is sending data to the output.

3 Testbench (tb_dec_viterbi.vhd)

With the source code comes a self-testing testbench with the Viterbi decoder that allows for comparison of golden reference data with the VHDL model.

3.1 Testbench description

The self testing testbench consists of processes for sending and receiving data to/from the Viterbi decoder. LLR input data is sent to the decoder and decoded bits are received. Self testing is done with different testing vectors. The test vector file names of the input data are of the form:

`llr_BL_<block lengh>_WL_<window length>_AL_<acquisition length>.txt`

The test vector file names of the decoding result (output data) are of the form:

`decoded_BL_<block lengh>_WL_<window length>_AL_<acquisition length>.txt.`

The test vector files contain one value per line.
The top entity of the Viterbi decoder testbench is parameterizable and looks as follows:

```
entity tb_dec_viterbi is
generic(
CLK_PERIOD          : time    := 10.000 ns;

BLOCK_LENGTH_START   : natural := 200;
BLOCK_LENGTH_END     : natural := 500;
BLOCK_LENGTH_INCR    : natural := 20;

SIM_ALL_BLOCKS       : boolean := true;
SIM_BLOCK_START      : natural := 0;
SIM_BLOCK_END        : natural := 4;

WINDOW_LENGTH      : natural := 55;
ACQUISITION_LENGTH : natural := 50;

DATA_DIRECTORY     : string  := "../testbench/"
);
end entity tb_dec_viterbi;
```

The parameters have the following meaning:

CLK_PERIOD
Period of one clock cycle.

BLOCK_LENGTH_START, BLOCK_LEGNTH_END, BLOCK_LENGTH_INCR

The decoder supports a variable block length at runtime. Therefore it is possible to iterate over several block length. `BLOCK_LENGTH_START` gives the first block length to simulate, `BLOCK_LENGTH_INCR` gives the increment for the next block length, and `BLOCK_LENGTH_END` gives the very last block length to simulate.

SIM_ALL_BLOCKS

The test environments come with test vectors stored in files. Each configuration has two test vector files (one for input and one for output), and each test vector file contains multiple code blocks in consecutive order. By setting this parameter to true, all blocks stored within one file will be simulated. If this parameter is set to false, it is mandatory to select the blocks to simulate with `SIM_BLOCK_START` and `SIM_BLOCK_END`. Else these parameters are ignored.

SIM_BLOCK_START

In case `SIM_ALL_BLOCKS` is set to false, this parameter gives the first block of the test vector files to use for simulation. In case `SIM_ALL_BLOCKS` is set to true, this parameter is ignored.

SIM_BLOCK_END

In case `SIM_ALL_BLOCKS` is set to false, this parameter gives the last block of the test vector files to use for simulation. In case `SIM_ALL_BLOCKS` is set to true, this parameter is ignored.

WINDOW_LENGTH, ACQUISITION_LENGTH

The window and acquisition length are set here. According to these values the core is configured and the test vectors are selected.

DATA_DIRECTORY

Path to the test vector files, relative to the simulation directory.

3.2 Running the Simulation

In order to run the simulations, a VHDL simulator such as Aldec's Riviera-PRO or Mentor's ModelSim is required. The test environment comes with a script file, located in the script directory. To run the simulation the script perform_simulation.sh has to be executed. The testbench will give the comparison result between test vectors and VHDL model for each single code block.

After simulation of all blocks, the simulator will stop and signal END. In case there are no errors there will be no further message.

4 Version Information

4.1 Product Version

Version	Date	Comment
1.0.0	12/01/16	Initial version.

4.2 Document Versions

Version	Date	Comment
1.0.0	12/01/16	Initial version.