P&O EAGLE
Wireless Video Downlink

In depth task explanation

Bertold Van den Bergh
Yuri Murillo

September 2017

1 Introduction

This document serves as an introduction to familiarize with the VHDL framework for the software implementation of the DVB-S2 transmitter and the automated standard compliance tests. The complete architecture can be seen in the image below.

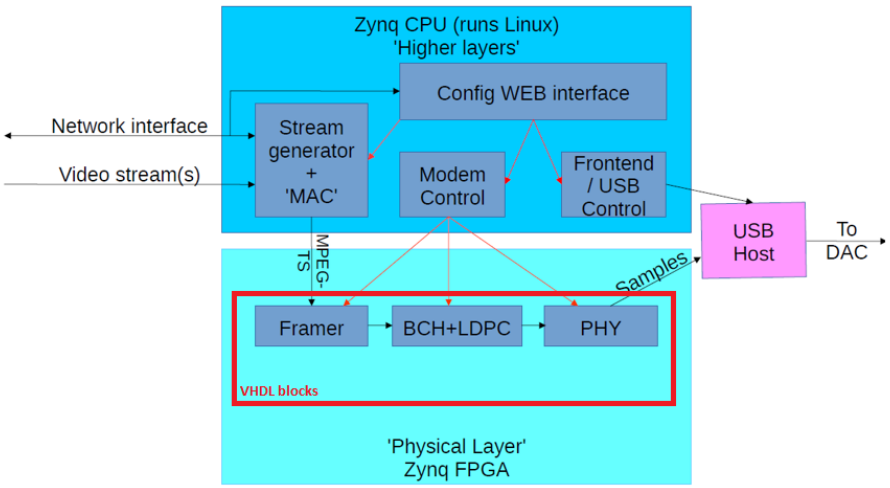


Figure 1: DVB-S2 architecture for the Zybo.

The provided VHDL blocks start with the baseband framer and continue with the LDPC encoding of the FEC submodule and remaining transmitter chain (see master Wireless Video Downlink document for documentation). Their behaviour can be understood by inspection of the code, as they are quite clearly structured and commented.

In the following, we provide information on how to develop custom VHDL code that ensures correct behaviour with the rest of the DVB-S2 blocks.

2 VHDL blocks

The supplied `students_vhdl` repository includes several folders:

- FECBlocks
- InputProcessing
- MPEGmux
- Modem-S2
- Link to the testvector repository

The first two directories should be used for developing the custom VHDL code (adding the code to the files located inside the `src` directory). The remaining two are used for multiplexing of several streams and full system implementation and testing. The testvector repository will be covered in the next section.

The custom VHDL blocks developed for the BCH coding need to include certain ports and feed certain signals to the rest of the DVB-S2 architecture to ensure compatibility.

As the signal travels through the different blocks of the system in a sequential manner, proper synchronization between them is a must. In order to ensure it, the ports definition generally follow the same structure:

- `clk`: clock signal.
- Data ports: `in_data` and `out_data` are used for the video stream signal. The architecture expects the custom blocks to deliver one output bit per clock signal.
- Result valid ports: `in_inputValid` and `out_outputValid` signals are used. If high, they indicate that the incoming / outgoing data is a valid bit in the stream and therefore should be processed by the current / next block of the chain, respectively.
- Block ready ports: `readyForPreviousBlock` and `nextBlockReady` signals are used for signalling the availability of the blocks to receive and process the data. `readyForPreviousBlock` is an output signal that communicates the previous block that the current one is ready to receive its data.

`nextBlockReady` is an input signal coming from the following block to indicate that the current one may send data to it. Note that in most applications the current block has to wait until the following one is ready to indicate its availability to the previous one, therefore ensuring that no data is lost in the chain.

2.1 BCH block

The FEC module of the transmitter is responsible for outer coding (BCH), inner coding (LDPC) and bit interleaving. The incoming baseband frames are then processed into FEC frames. The ports definition for the BCH block is given below:

- `clk`: clock signal.
- `in_frame_start`: the signal should be set to high before the beginning of a FEC frame to reset the internal state of the block.
- `in_frame_end`: should be high at the end of the user data.
- `bch_polynomial`: control signal to select which polynomial to use for encoding. Note that different LDPC coderates will need to use different types of polynomials (Table 5 of the DVB-S2 standard). The specified values are defined for short FEC frames (16200 bits), but when using normal frame length (64800 bits) this should be changed to `ctrl_LDPCCode_64` = `ctrl_LDPCCode_16` + 10.

`ctrl_LDPCCode_16`	LDPC code
0	1/4
1	1/2
3	1/3
4	3/5
5	2/3
6	3/4
7	4/5
8	5/6
9	8/9

- `out_bchFrameEnded`: control signal set to high when transmitting the last bit of a FEC frame.
- `in_data` and `out_data`.
- `inputValid-outputValid` and `readyForPreviousBlock-nextBlockReady`.

The coding of the incoming data is a straightforward process, as described in the standard. Refer to it for further information for development of this block.

You may code this module with any approach that you find suitable, but the recommended way is to do it following a finite state machine approach (chapter 7 of the free-range VHDL book) using a linear feedback-shift register (separate document on polynomial division and LFSR).

3 Automated standard compliance tests

Debugging the implemented VHDL code may become a very challenging task, as the initial approach is to test the software implementation of the transmitter by using a standard of-the-shelf receiver. In this case the system performs properly if everything is well designed, and completely fails to do so if there is any error, giving no feedback on where the bug in the implementation may be.

In order to speed up such process, automated standard compliance tests are provided. These testbenches include raw bitsreams of the output of every block of a standard compliant transmitter when using a predefined input video file. When applying this same input file to the software implemented transmitter, the outputs should match bit-by-bit if the developed VHDL code works as it should. If not, they will automatically point out the submodule causing the failure and in which stage of the whole stream.

The raw bitstreams are stored in the `testvector` repository, which should be cloned in the `tmp` folder of the PC due to quota limitations. The testvectors may be used for full standard compliance tests for the BCH block and the full transmitter (modem-s2). They are generated both using custom video files and standard ones obtained from the BBC (where a document is provided for further info). Different coderates are supported, and if a block needs to fulfil several tasks the testvectors are provided at the end of each in order to be able to do a finer debugging. In general the naming of the testvectors is self explanatory.

In order to run the tests, there is a Makefile included inside the VHDL repositories. `Make` will compile the VHDL block, while `Make test` will run the full compliance test for the block using all possible testvectors and random delays between the handshaking of adjacent blocks. Different sub-tests may also be done by selecting only a portion of the testvectors or a sub-block. Again, open the Makefile for more information, as it is self explanatory.

The tests will print on the screen the number of frames tested and how many bit errors were found. You may then simulate the VHDL code using *ghdl* and view all the signals using *gtkwave* in order to detect the exact moment where the bug happens and elaborate on why is such behaviour happening.

3.1 How do I implement a Finite State Machine (FSM) in VHDL?

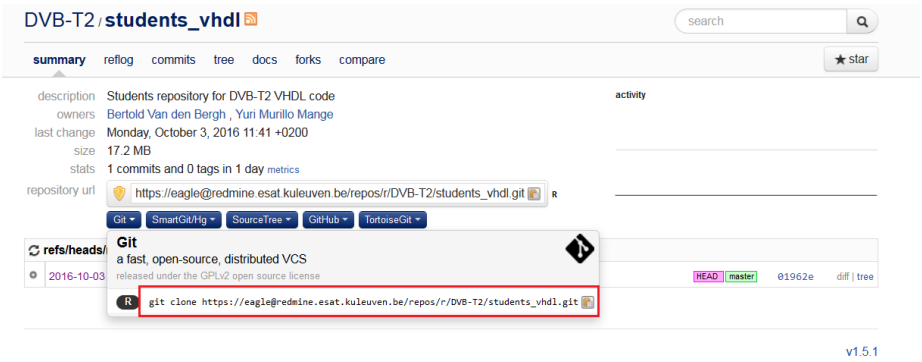
4 Frequently Asked Questions

4.1 How do I download the files?

Login to https://redmine.esat.kuleuven.be/repos/ with the following credentials:

- username: `eagle`
- password: `eagle`

Under the repositories tab you may download the `students_vhdl` and `testvector` repositories. You may download them as zip files or run the `git clone repository_url` command from a linux terminal.



4.2 I want to work with my windows laptop on the code. Can I do it?

Developing the code in windows is not recommended as the whole architecture was developed in linux and is possible that all paths and files are completely messed up. Moreover, all the necessary software is already installed on the ESAT computers, so try to use them. Again, if absolutely necessary, you may install a virtual machine on your laptop with some linux distribution running all the necessary software. Use your preferred ones, but some easy solutions might be VirtualBox with Ubuntu. For software you will need ghdl and GTKWave.

4.3 I've downloaded the files but I don't know how to start developing the code. Where is everything located?

The files you need to code are located inside `VHDL/FECBlocks/src/bch.vhd` for the BCH coding. You may also take a look at the rest of the files to familiarize

with the whole architecture. Also, take into account that these files are given to you as a template: just follow them if this makes the task easier for you. For example, you may split `bch.vhd` into different blocks, each doing one of the different sub-tasks that the whole module does. Finally, you may also use different port and signal definitions.

4.4 Which software do I need to write the code?

You may use your preferred text editor to write in them. Gedit is the default text editor for the GNOME environment and is very easy to use. You may also indicate VHDL syntax for better readability. Vim is a much more powerful tool, although it requires some hands-on experience.

4.5 But how do I compile and run the code?

The automated standard compliance tests are located in the path `VHDL/FECBlocks/Makefile`. Take a look at them if you want to see what they do as they are self-explanatory. There is also a whole transmitter test makefile located at `VHDL/Modem-S2/Makefile`. Running the command `make` will compile the whole `src` directory of each repository. `make test` will run the tests. Alternatively, you may compile and run every block manually using ghdl into a file and then open it to see the signals using GTKWave. Consult the documentation of these tools to learn how to do it.

4.6 There are a lot of things I don't understand in the code/standards. What are they? What are they used for?

Ask it. Many things are left there in order for you to ask and start getting an idea of how communication systems work. Things such as the scrambler, square root raised cosine roll-off factor, Forward Error Correction, and so on. It is not absolutely necessary to understand these terms in order to develop the VHDL code for the transmitter, but if you are now introduced to them then you will have a better understanding during next years!

4.7 How do I implement a Finite State Machine (FSM) in VHDL?

You have a book called *Free Range VHDL* in Toledo that will cover most of your immediate VHDL questions. When in doubt, make sure to first check it before asking. Also, there is a second reference book that can complements what is not fully covered in the one you are given. Chapter 7 of *Free Range VHDL* deals with FSM design and implementation, which will help you write your code and define all the different states and functionalities that you need to do.

4.8 Which LDPC coderate do I need to choose?

You will need to support all of them in order to have a transmitter that is able to offer the best performance in all situations. Forward error correcting codes work in the same fashion: they introduce a certain amount of redundant information, which is used to detect if there have been errors in the received data due to channel conditions. The more redundant information is sent, the higher amount of errors can be detected and corrected. However, the transmitter is sending just overhead and not actual data, so the game is to find an optimal rate (based on the channel quality, which may be variable) in which a certain amount of errors can be corrected while not wasting a lot of bandwidth. Therefore, the best way of implementing all coderate dependent signals in your code is to set the length of useful registers, RAM, etc according to certain generic that will be given a value depending on the actual coderate used.

4.9 Which type do I use for binary signals?

You can use the binary type if you want, but the recommended is `std_logic`. For vectors, such as a byte, you may use `std_logic_vector(7 downto 0)` and in the same fashion for any other vector length. Also, it is common practice to declare your signals from highest to lowest (most significant bit to the left).

4.10 How do I convert an integer into a binary number?

```
binary_number <= std_logic_vector(to_unsigned(integer_number, binary_number'length));
```

4.11 I'm working on the BCH polynomial division but I don't understand how this mathematical operation can be done with a Linear-Feedback Shift Register!

In case you haven't already covered this in any other course I'm attaching you a classic document on polynomial and LFSR arithmetic. This covers the actual math behind it, polynomial multiplication / division and other applications, such as random pattern generators (used, for example, for computing the random frequency hopping scheme of a Bluetooth communication or for scrambling the signal in the input processing part). The document will be uploaded to Toledo.

4.12 I get an error stating that my files are obsolete when I try to run the tests after having changed my code and now I cannot do anything. Help!

Don't worry, everything is going to be fine. You just compiled your code, run it to see that it had some errors, changed it and saved it, and then tried to run again without recompiling. You just messed with the references of the files, but a new compilation (`make`) will solve the error. In case everything fails, type

`make clean` to remove all output files from the compiler. After doing so, just compile again (`make`) and relaunch the tests. That should solve any issue. In fact, when in desperate need / having strange errors just clean the whole project and recompile everything and that will potentially fix it.

4.13 My disk quota is full and I can't download the testvectors. Also, where should I place them so the tests can find them?

First things first, quota limitations can be avoided by working on the `/tmp/` directory, since you can write any amount of data you want to it - well, until you fill the hard drive :) The downside of this is that any data you store there will be deleted by the time you come to the next session. So, I would suggest you to download the testvector repositories to the `/tmp/` directory. In fact, I would suggest you to also download the VHDL repositories there. A good workflow would be to work in the `/tmp/` directory every session, copying your code from your home folder and working on it through the whole session. When the session is finished simply copy the code that you have been working on back from `/tmp/` to your home directory. This will solve any quota issues, plus you may store different versions of your code week by week, for the times when "something was working last week but I don't really know what have I changed and now it is not working" :) Second, you should place the testvectors inside their correspondent directories. For the BCH part, they should be in the input processing folder (so you will have the bin, src, obj AND testvector folders).

4.14 What are the different tests? What is handshaking?

Open the makefile. Inside it you can see the different tests that you can do for every block. This helps to partially test your code to check for different sources of errors. In general the names of the tests are self explainatory. Regarding handshaking, we define it as delays in the ready and valid control signals of the previous and next blocks of the one you are testing. So, in the tests we generate random delays for these signals to test whether your block waits until the next one is ready, the previous one provides a valid data, etc. In general, try first to test your code without handshaking to ensure that the behavior is the one expected and after you have it working then test handshaking and timing issues.

4.15 How do I see the signals for debugging when the test fails?

The tests we provide do not write a file to disk by default. This is done to prevent quota limitations on ESAT PCs, since the generated files may be huge. Two options here:

1. Modify the makefile to include the `--vcd=/tmp/file.vcd` option while running.

2. Manually run the make test command and stop it. The first line that you get in the terminal is the whole command list that you are running. You can simply copy paste it and then add the `--vcd=/tmp/file.vcd` command at the end. For example, let's say I run the `inputprocessing` test. In my terminal, this will result in the following commands being executed (extra exercise, check if you understand all of them!): *./bin/inputprocessing_fulltest -ghandshake_transaction_probability_source="1" -ghandshake_transaction_probability_sink="1" -ginput_data_file="../testvector/inputprocessing/BeforeInputProcessing/telemic_video.txt" -gcheck_data_file="../testvector/inputprocessing/AfterInputProcessing/output34_short_binary.txt" -gcodeRate_in=6 -gbypassScrambler=true* ...and then the test starts. Stop the test (`ctrl+c`), copy the commands and then add the `--vcd=/tmp/file.vcd` command at the end and execute them!

You will be able to open the vcd file in GTKWave and see the different signals to debug your code. I would recommend to keep GTKWave open at all times, and after rerunning the test simply go to File→Reload Waveform. This will keep the position of your cursor and the list of signals displayed so you don't need to manually add them every time (as I used to do for quite some time before I discovered this feature!).

4.16 When I run the inputProcessing / BCH / Modem-S2 testbench alone the testvectors are not found!

Yes indeed, that may happen if you try to run any testbench alone, since the path that is written there does not correspond to the location of the testvectors that you have. Two things you can do:

1. [**Recommended**] Use the makefile for testing: it will input the correct paths for all testvectors and you do not need to worry about anything - just debugging errors :)

2. Edit the `tb.vhd` file if you want to manually execute the testbench. You will need to compile and run it (`-a, -e, -r` ghdl commands (search them in the ghdl documentation if you are not sure) and `--vcd` as in the question above) and then open the file in GTKWave.

4.17 Wait I just read something about Modem-S2 in the last question. What is that?

That is the complete transmitter implementation with the remaining blocks that take part in the standard. Open it and take a look at the code if you are curious. Other than that, the main purpose of it is to run the complete transmitter tests and flash it on the Zybo after everything is working.

4.18 I try to compile but seems like the makefile is not able to find GHDL. How do I solve this?

Since we provide GHDL within the `students_vhdl` repository, we define the path in the makefile pointing to it. This usually works but I've detected that sometimes it doesn't and some groups are simply not able to compile anything. Almost every time this is solved by specifying the absolute path to the GHDL file in the makefile. In order to do so, you can simply add it in the first line of the makefile.

4.19 I try to run the tests but they never start!

Make sure you turn the `output_valid` port of your block to '1' when valid data is sent. Otherwise the test will always wait until you do so!