Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Integration of FAPEC as Data Compressor Stage in a SpaceFibre Link

by

Alberto González Villafranca

Advisor: Enrique García-Berro Montilla

Co-advisor: Jordi Portell i de Mora

October 2016

If I have seen further it is by standing on the shoulders of giants

Isaac Newton

Acknowledgements

Este proyecto lo he realizado con muchas prisas por lo ajustado de los plazos. Sin embargo lo he disfrutado mucho, y me apena no haberle podido dedicar más tiempo. Voy a acabarlo justito justito y lo dejo con la sensación de que, con un poco más de trabajo, podría haber cerrado muchos de los flecos pendientes.

En primer lugar, me gustaría agradecer el tiempo que me han dedicado mis dos tutores, tanto Enrique como Jordi. Y esto no va solo por este proyecto, este agradecimiento se prolonga hacia atrás en el tiempo unos once años, si no me falla la memoria. Gracias a ellos estoy donde estoy y tengo la inmensa fortuna de haber cumplido el sueño de trabajar en el sector espacial.

Dicho esto, me gustaría continuar por mi familia y, en especial, mis padres. Si Enrique y Jordi me han dado la posibilidad de volar muy alto, mis padres me han dado la posibilidad de hacer cualquier cosa. Siempre me han empujado a estudiar, en especial mi madre, y han trabajado muchísimo para darnos a mí y a mi hermana todo lo que hemos necesitado y mucho más. También a mi hermana Sara porque sé todo lo que me quiere, aunque le cueste exteriorizarlo.

Deseo agradecer a STAR-Dundee Ltd. las facilidades para utilizar el código de SpaceFibre y la unidad STAR Fire.

Y, para acabar, a mis amig@s, sin quienes mi vida sería insoportablemente aburrida.

¡Gracias a todos!
Gràcies a tothom!
Thank you everybody!

Table of Contents

Index of figures

1. Introduction

The instruments used in modern space missions require increasing amounts of telemetry resources to download the acquired data to the ground. Transmission link speed has become the bottleneck of the data chain for many applications. One way to solve this issue is to apply on-board data processing techniques to reduce the amount of data to be sent down to ground, and its use is becoming increasingly necessary to deal with the large amounts of data generated by modern spacecrafts. Remarkably, data compression is a data processing technique that encodes information in fewer bits than the original representation. It is therefore currently seen as a mandatory stage for many missions in order to mitigate the saturation of the telemetry link. However, the available on-board processing power has been traditionally modest. Compression systems have thus been kept as simple as possible.

The Prediction Error Coder (PEC) is a lossless data compression algorithm belonging to the family of the entropy coders [1,2]. PEC was developed considering the tight constraints of a space mission and its main features are low complexity and resilience against statistical outliers in the data. PEC needs to be calibrated for different types of data, and its performance depends on the quality of this calibration. The Fully-Adaptive PEC (FAPEC) is an adaptive version of PEC that was developed to address this calibration problem. FAPEC typically delivers better ratios than the CCSDS 121.0 recommendation (General Purpose Lossless Data Compression [3]) on realistic data sets [4].

Embedded hardware implementations of data processing algorithms are becoming increasingly popular in space. Hardware data processing performs faster and uses less power than the conventional approach of using general purpose CPUs. The most cost-effective approach for custom data processing solutions as those used in space is usually Field Programmable Gate Array (FPGA) devices, but space-qualified FPGAs have been traditionally not very powerful. However, the capabilities of FPGAs are steadily improving, thus enabling the implementation of more complex algorithms. The new generation of radiation tolerant FPGAs such as Xilinx Virtex-5QV [5], Microsemi RTG4 [6] and NanoXplore BRAVE [7] (a new European space-qualified FPGA project) offer faster speeds and much more resources than the old Microsemi RTAX family traditionally used in space. Thanks to this dedicated logic, algorithms can run much faster and with a fraction of the power requirements necessary when they run in general purpose CPUs.

SpaceFibre (SpFi) is a new, multi-Gbits/s on-board network technology which runs over both electrical and fibre optic cables. SpFi currently operates at 3.125 Gbits/s in flight-qualified technology, and is capable of fulfilling a wide range of spacecraft on-board communications applications because of its inbuilt quality of service (QoS) and fault detection, isolation and recovery (FDIR) capabilities. SpaceFibre is now being standardised by the European Cooperation for Space Standardization (ECSS) and is expected to be published as a formal standard this year.

The aim of this project is to integrate the FAPEC data compressor into the SpFi codec. Both SpFi and FAPEC have been designed to withstand the harsh space environment and, hence, it seems a logical step to combine the data compressor into the data link technology to increase the net throughput achievable.

1.1 PEC

The Prediction Error Coder (PEC) is central in the operation of FAPEC. PEC was developed within the frame of the Gaia mission [8]. The effort was focused on the development of a very fast and robust compression algorithm, and PEC was the outcome – an entropy coder based on a segmentation strategy. PEC is composed of three different coding strategies, or variants, known as *Low Entropy* (LE), *Double-Smoothed* (DS) and *Large Coding* (LC). LE and DS are both ranged Variable Length Codes, and LC is a unary prefix code [9]. The three coding options share the same principle: the entire range of the data to be coded is split into four smaller sub-ranges or segments. The appropriate segment is selected depending on each individual value. In PEC the first segments are smaller than the original symbol size, while the last segments can be slightly larger. PEC follows the assumption made for most entropy coders that most values to be coded are close to zero [9]. Thus, the coding efficiency depends on the segment sizes chosen and on their relation with the probability density function of the data.

Compressing data with PEC requires only very few and simple calculations. The values inside these ranges are coded in a plain binary form, implicitly assuming equiprobable values inside each range. The resulting coding hierarchy is actually similar to a coding tree, but with very short branches because these only represent the prefixes, not the values themselves. In Fig. 1.1 a schematic view of PEC is shown and the coding strategy of each of the ranges is unveiled. The coding scheme is completely different to that of other entropy coders such as of the Rice coder, the compression core in the CCSDS 121.0 Lossless Data Compression Recommendation [3].

PEC has a very low computational cost and an excellent resiliency to outliers and noise in the data, also offering excellent efficiencies for a wide variety of data statistics. Using this coder the usual strategy in space data compression – based on a two-stage scheme, namely, an adequate pre-processing stage followed by an entropy coder – can be improved in most cases, provided that the pre-processing stage is properly tailored. This pre-processing can be seen as a prediction, and is defined in both the compressor and the decompressor. Every sample entering PEC is compared (subtracted) against its predicted value, thus leading to signed values (that is, prediction errors). Subsequently this difference between the real value and the prediction is coded using PEC.

Coding signed values adds some redundancy because of the existence of codes for both *+0* and
-0 values. The CCSDS recommendation uses a mapping algorithm to eliminate this redundancy at the expense of a slightly more complex algorithm [3]. However, a different alternative is possible, and this is a key feature of PEC. In PEC, both the LE and DS options use the *-0* code, as well as the last value of each segment (i.e. all bits set to one), as escape sequences. These implicitly indicate which of the coding segments are used. On the other hand, the LC option simply uses the unary coding to indicate the appropriate segment and avoids the output of the sign bit when coding a zero, thus eliminating the *-0* redundancy.

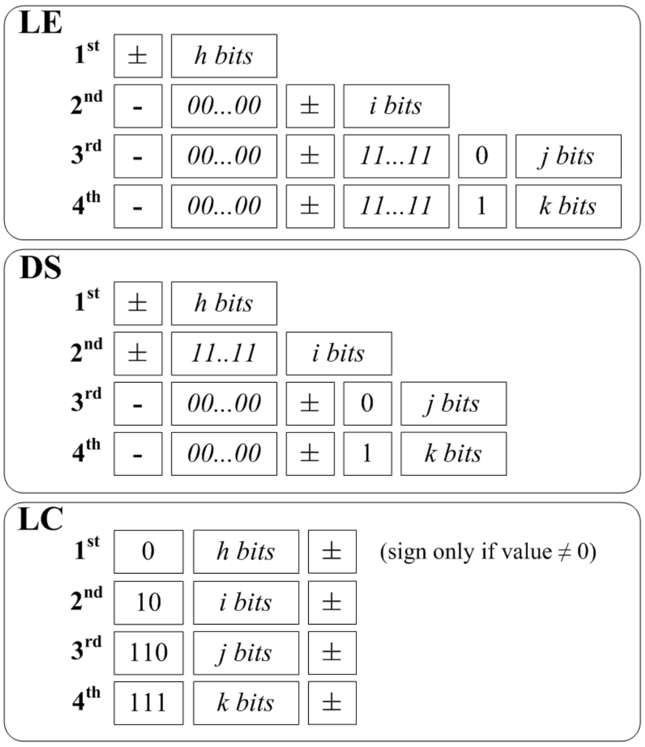


Fig. 1.1: PEC coding strategy

1.2 FAPEC

PEC is a low-complexity high-performance compressor which typically outperforms the CCSDS recommendation. However, it needs to be calibrated for every set of data. The Fully-Adaptive PEC (FAPEC) is an adaptive compression algorithm that calibrates PEC once every some hundred samples, thus allowing it to rapidly adapt to changes in the statistics of data. The operation of the FAPEC coder basically can be described as an algorithm which selects the best PEC coding configuration for each data block, followed by a PEC coding step that applies the optimal tables obtained on the first step.

FAPEC is a lossless data compression algorithm that typically offers better ratios than the CCSDS 121.0 on realistic data sets. Its compression efficiency is higher than 90% of the Shannon limit in most cases, even in the presence of large amounts of noise and outliers [4]. FAPEC was designed for space communications, where requirements are very tight in terms of energy consumption and efficiency. FAPEC low computing resources consumption and high compression speed cover a wide range of possibilities that current compressors cannot offer for high throughputs due to their high compression time. FAPEC can be integrated into almost any data transfer flow, enhancing the data rate of the system with very small energy and data processing time increment.

The data link in space missions, as any digital communications channel, is subject to noise and transmission errors. Despite the powerful techniques available for error correction an error-free transmission cannot be guaranteed. Also, re-transmissions of data blocks received with unrecoverable errors are not always possible. Therefore, the use of small independent data blocks in the data compression stage is highly advisable. Thus, adaptive algorithms requiring large amounts of data for their optimal operation, such as Huffman or LZW, are not applicable. Furthermore, these algorithms are quite demanding when compared with those studied here, and will usually yield little improvement in terms of compression ratio. In short, data compression systems used in space missions must use small and independent data blocks in order to guarantee the minimum possible losses in case of transmission errors.

FAPEC accumulates the values to be compressed in blocks of a user-configured size – typically ~200 samples. During this, an internal histogram of the moduli of the pre-processed values is calculated on-the-fly. Once the block of values has been completed, the algorithm analyses the histogram to obtain the best coding parameters, calculating the accumulated probability for each value. The choice of the coding option (LE, DS or LC) and the specific coding table are defined through a set of accumulated probability thresholds. That is, FAPEC defines the coding segments (and hence the coding table) according to their accumulated probability and code length. This nominal tuning offers excellent compression ratios for almost any case. Furthermore, FAPEC threshold levels can be modified to better suit other statistics if required. This is another significant advantage with respect to the Rice coder used by the CCSDS 121.0 recommendation, which is only optimal for noiseless Laplacian distributions.

Analysing a histogram of 16-bit values (which is the case studied in this project) can be very time consuming, and can lead to prohibitive processing times if naively (or exhaustively) implemented. For this reason FAPEC uses a logarithmic-like histogram, with increasing bin sizes for larger values. That is, large values are grouped and mapped to a single histogram bin, while full resolution is kept for the lowest values. This analysis is precise enough for the case of ranged entropy coding, such as PEC, which does not require a precise knowledge of the largest values. Once the coding parameters (coding table) have been determined, they are explicitly output as a small header at the beginning of the compressed data block. The decoder only has to invert the PEC process using the parameters indicated by the header, without requiring any knowledge on the adaptive algorithm used to calibrate the coder. In this way, the fine-tuning thresholds of FAPEC or even the auto-calibration algorithm can be safely changed without requiring any modification in the decoding stage. This is an advantage of FAPEC against other compression algorithms.

There is an early FAPEC implementation in an FPGA developed as a feasibility demonstrator. The benchmarked implementation on a Microsemi PROASIC3L successfully proved its operation at 32 Mbit/s (2 Msample/s) with a relatively simple design [10].

1.3 SpaceFibre

SpaceFibre (SpFi) is a spacecraft on-board data-link and network technology developed by STAR-Dundee Ltd. and the University of Dundee for the European Space Agency (ESA). It is the next generation of the widely used SpaceWire (SpW) technology, offering higher throughput, lower mass and new capabilities including quality of service (QoS) and fault detection, isolation and recovery (FDIR). Furthermore, it runs over both electrical and fibre optic cables. SpFi will be released as an ECSS standard later this year [11].

Initially targeted at very high data rate payloads such as Synthetic Aperture Radar (SAR) and high-resolution, multi-spectral imaging instruments, SpFi is capable of fulfilling a wider set of spacecraft on-board communications applications because of its inbuilt QoS and FDIR capabilities and its backwards compatibility at packet level with the ubiquitous SpW technology. This allows simple interconnection of existing SpW devices into a SpFi network and enables legacy equipment to take full advantage of the inbuilt QoS and FDIR in SpFi.

SpFi provides high data rate capabilities in radiation-hardened technology: 3.125 Gbits/s in Microsemi RTG4 and Xilinx Virtex-5QV FPGAs and 2.5 Gbits/s in Microsemi RTAX FPGAs, with ASICs that operate at 6.25 Gbits/s currently under development [12]. This high data rate currently provides more than 15 times the maximum throughput of a SpW link (200 Mbit/s). This allows data from multiple SpW devices to be concentrated over a single SpFi link, thus substantially reducing cable harness mass and simplifying redundancy strategies. Multi-laning provides lane redundancy and can also be used to achieve much higher data rates, e.g. 40 Gbits/s, sufficient for most spacecraft on-board data-handling operations.

The innovative inbuilt QoS mechanism uses Virtual Channels (VCs) to provide multiple independent communication channels over a single physical link. Each channel provides priority, bandwidth reservation and scheduled QoS. These QoS mechanisms operate together, resulting in a very versatile QoS which also provides “babbling node” protection and scheduled, deterministic communication without wasting any network bandwidth. This simplifies spacecraft system engineering, which reduces system engineering costs and streamlines integration and test.

Novel integrated FDIR detects, isolates and recovers from faults at the link level, which prevents faults from propagating and causing further errors. The FDIR capability of SpFi provides galvanic isolation, transparent recovery from transient errors, error containment in virtual channels and frames, enhancing on-board network robustness. This simplifies system level error-handling software, reducing development and system validation time and cost.

SpFi includes low latency event signalling and time distribution with broadcast messages. This enables a single network to be used for several functions including: transporting very high data rate payload data, carrying SpW traffic, deterministic delivery of command/control information, time distribution and event signalling.

With these capabilities SpFi brings many benefits to spacecraft on-board data handling systems:

- Very high data rates that meet the needs of very demanding instruments, mass-memory internal networks, and telecommunications systems.

- Reduction of harness mass by 33% and 50% when comparing the mass of a single SpW cable to SpFi electrical and fibre optic cables respectively, and by more than 90% when comparing per bit transferred.
- Simplification of redundancy though integration of several on-board communication functions into a single network, and through the carrying of the traffic of multiple SpW links over a single SpFi link.
- Increase in reliability by requiring one network rather than two or three to carry out the necessary on-board communication functions.
- Straightforward error recovery since transient errors are recovered on the link and do not need to be considered at the system level.
- Deterministic data delivery enabling AOCS/GNC and other control applications to be supported.
- Long distance communication enabling launcher applications to be addressed, where a single network can provide control, monitoring and video capture functions.
- Galvanic isolation improving system robustness by preventing fault propagation.

SpFi enables using a single, integrated network that carries instrument data, configuration and control information, deterministic traffic, high-resolution time information, and event signals. This improves reliability, saves mass, and reduces cost. Fig. 1.2 shows the SpFi protocol stack and outlines the functions of the different layers.

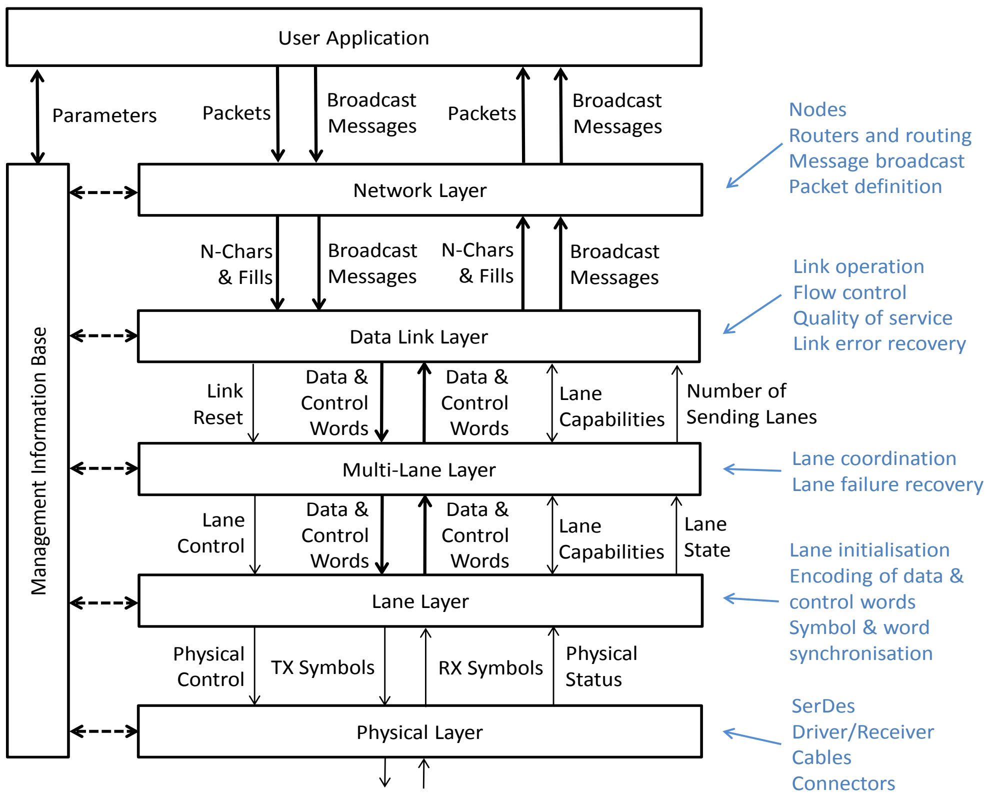


Fig. 1.2: SpFi protocol stack

1.4 Objectives of this work

The goal of this project is to integrate the FAPEC compressor with the SpaceFibre codec. Together they will provide an efficient way to achieve higher data rates without the penalties associated to resorting to higher line rates or using the SpFi multi-lane extension, such as increased complexity and energy consumption.

Firstly, the initial FAPEC implementation in VHDL needs to be analysed. The design delivered a throughput of 32 Mbit/s with a ProASIC3L FPGA. The output of FAPEC was serial. SpFi currently can send up to 2.5 Gbit/s (3.125 Gbit/s line rate) over a single Virtual Channel, and its input is a 32-bit parallel interface. Therefore, to integrate both FAPEC and SpFi, FAPEC needs to be much faster and feature a parallel output instead. Going from a throughput of 32 Mbit/s to 2.5 Gbit/s requires an 80-fold speed increase of the FAPEC implementation. Thus, massive changes in the implementation architecture are required to reach such performance gains.

Secondly, once FAPEC has been adapted to the SpFi constraints it needs to be integrated into a hardware design. Fortunately, the STAR Fire unit from STAR-Dundee provides a SpFi platform suitable for testing new designs. This unit features two SpFi interfaces and its design can be modified to add FAPEC on top of the SpFi protocol stack. The target FPGA family will be the Spartan-6 which is more representative than the ProAsic3L family used for the initial FAPEC prototyping.

This memory is organised as follows. Chapter 2 describes the changes applied to FAPEC to make it suitable for integration with SpaceFibre. The STAR Fire design and the performance of FAPEC in hardware are described in Chapter 3. Finally, Chapter 4 summarizes the work, elaborates our conclusions and proposes some forthcoming work. The annexes show the VHDL code developed for the new PEC coder module and the Word Packer modules.

2. Implementing FAPEC inside an FPGA

2.1 Introduction

The simplicity and robustness of FAPEC places it as an interesting alternative to the current standard for universal lossless data compression for space. However, the performance of a hardware implementation needed to be assessed before seriously considering FAPEC for such role. Considering the peculiarities of the internal operation and architecture of FPGAs, it was clear that the hardware implementation of FAPEC was not straightforward from its software counterpart or from its algorithmic definition. In order to achieve an optimal hardware implementation, several features of the original algorithm needed to be modified to be more hardware oriented. Specifically, floating-point operations, multiplications and divisions had to be avoided. Also, a logarithmic-like histogram used for lower complexity had to be modified to allow an easier (binary-like) rule of construction and analysis. Finally, it was decided to limit block size to 255 samples. After implementing these changes the new FAPEC was validated and it was proved that the modifications had little effect to the algorithm performance [10].

2.2 FAPEC Reference Design

In this section the initial hardware implementation of FAPEC is described. This implementation has been used as a reference design for this project.

2.2.1 Target Performance and Platform

The initial goal of compression for FAPEC was derived from the Gaia mission constraints, as FAPEC was developed from concepts proposed for this mission [4]. Specifically, the Gaia payload uses 16-bit A/D converters (ADC) at a very high conversion rate. It was established as an initial goal the compression of a raw CCD output stream of Gaia, which is about 2 Msample/s or, in other words, 32 Mbit/s. Although modest, this allowed to estimate the potential of the algorithm and to evaluate the possibility of further modifications to adapt it to a faster scheme. The input interface adopted was 16-bit words at 2 MHz, although a serial output interface able to operate up to 46 MHz (worst case) was selected owing to the intrinsic variability of the output data rate. A parallel interface allowing lower clock frequencies was discarded because it presented higher complexity and power consumption in the hardware interface.

Regarding the platform target, FPGA technology naturally appeared as the best option: reprogramming is usually allowed and it is a low-cost alternative. The preferred target for a space application of FAPEC was the radiation-hardened ACTEL RTAX antifuse technology, commonly used in space missions. However a flash-based FPGA from the same manufacturer was selected for prototyping. It provided re-programmability and portability of synthesis, thus reducing costs while assuring a high degree of similarity. An ACTEL PROASIC3L development kit was finally chosen for the sake of development simplicity. It included an M1A3P1000L FPGA, offering 24576 VersaTile logic elements and a 48 MHz reference clock. Also, the board contained a 1 MByte SRAM and a 16 MByte Flash memory, used in the design to input and output files.

2.2.2 Architecture

In Fig. 2.1 we show the structure of the algorithm implemented. In the text below we describe the different modules composing the whole FAPEC compressor. Note that in this memory the actual names of the different VHDL modules are indicated in upper case and *Consolas* font (e.g. `EXAMPLE`).

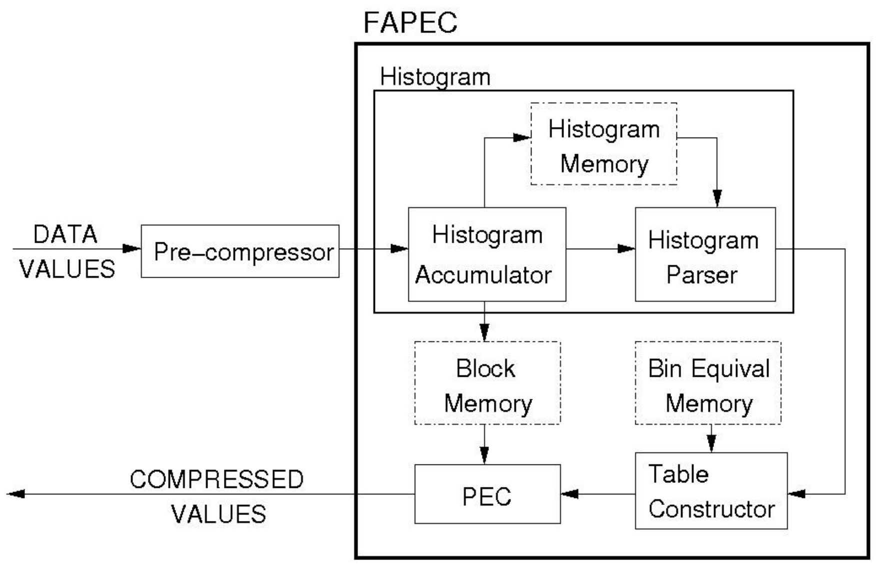


Fig. 2.1: FAPEC hardware implementation architecture

The pre-compressor (`PRECOMPRESSOR`) stage simply consists of a data predictor and a differentiator. That is, it predicts an input value to be equal to its predecessor. This is the simplest pre-compressor but it is still very effective when the sample values vary slowly, and it also allows removing offset values. The histogram accumulator (`HIST_CONSTRUCTOR`) analyses each of the pre-compressed samples and increments the appropriate bin value of the histogram memory. Because of timing constraints, it is necessary to have two different streams which alternatively process the incoming values to identify their corresponding histogram bin. The values are then stored in the block memory, where they wait until their coding table is ready. After processing the 255 samples of a block, the histogram boundary extractor (`HIST_BOUNDARY_EXTRACT`) operation begins. It parses the histogram, accumulating the occurrences stored in the bins and determining the ceilings for each of the four PEC segments. Additionally, it selects the PEC variant and the size of the first segment.

These initial modules plus their associated memory are in charge of performing the analysis of the statistical distribution of the data. Essentially, they build and analyse the histogram. Once the histogram procedure is complete, all the coding parameters are implicitly set. The next stage is the table constructor (`TABLE_CONSTRUCTOR`), which derives the size of the second, third and fourth PEC segments from the ceilings given by the histogram boundary extractor. These segment sizes constitute the coding table. The table constructor also provides the maximum value that can be coded with each segment. We must note that the use of a small bin-equivalence memory as a Look-Up Table (LUT) is required to avoid the continuous calculation of the mapping between each of the histogram bins and the input values. FAPEC uses a logarithmic-like histogram, mapping the 2^{16} possible values (16-bit samples) to just 37 bins. Storing the maximum value associated to each bin in the bin-equivalence memory avoids unnecessary operations. These modules constitute the adaptive stage of PEC, that is, the FAPEC algorithm. The last module shown in Fig. 2.1 is the PEC coder (PEC_CODER). It receives the segment sizes and maximum values from the table constructor for each block, outputting these coding parameters as a packet header. Finally, the values stored in the block memory are coded following the PEC algorithm.

The implementation of the FAPEC compressor was fully developed in VHDL. Apart from the memory blocks, neither IP cores nor non-standard functions were used, thus simplifying the porting of the algorithm to the RTAX model. A modular approach was adopted for validating the prototype. The modules were validated incrementally, that is, the validation of a module also included its predecessors in the compression chain.

2.2.3 Performance

ProASIC3L logic technology basically consists of a sea of VersaTiles [13]. Each VersaTile can be configured as a three-input logic function, a D-flip-flop (with or without enable) or a latch, by programming the appropriate flash switch interconnections (Fig. 2.2). This means that, contrary to other FPGA technologies, a combinational or a sequential element uses the same element in the ProASIC3L.

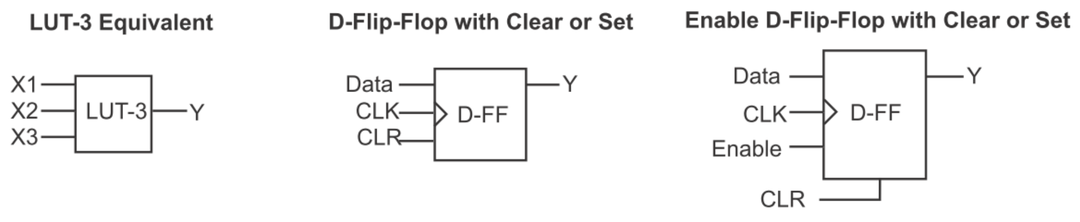

LUT-3 Equivalent
X1
X2
X3
LUT-3
Y
D-Flip-Flop with Clear or Set
Data
CLK
CLR
D-FF
Y
Enable D-Flip-Flop with Clear or Set
Data
CLK
Enable
D-FF
Y
CLR

Fig. 2.2: Different configuration options for a ProASIC3L VersaTile

The table shown in Fig. 2.3 describes the number of VersaTiles used by the main modules forming the FAPEC compressor. The central column shows the percentage of the whole FPGA VersaTiles used by each module, and the right column represents the percentage of the usage with respect to the total VersaTile elements used by the FAPEC module.

The most complex module is the PEC_CODER, using around a third of the total resources used by FAPEC. The TABLE_CONSTRUCTOR and the HISTOGRAM_CONSTRUCTOR use around a quarter of the total VersaTiles each, and the remaining is split between the PRECOMPRESSOR and the HIST_BOUNDARY_EXTRACTOR (10% each). In total, the whole FAPEC module takes a 14% of the logical resources of the ProASIC3L 1000 FPGA.

After place and routing, the results shown in Fig. 2.4 were obtained. As expected, they are in line with the synthesis results, with a small difference which is due to optimisations that are performed at a later stage by the placer tool. Interestingly enough, in the post-place and routing report the number of tiles used for combinational and sequential purposes is indicated. Note that there are as many as four times more VersaTiles used as combinational cells than sequential cells. This is mainly due to the large number of multiplexing operations required to generate the compressed codes.

	VersaTiles	**%**	**% of FAPEC**
PRECOMPRESSOR	334	1.4	9.6
HIST_CONSTRUCTOR	834	3.4	23.8
HIST_BOUNDARY_EXTRACT	349	1.4	10.0
TABLE_CONSTRUCTOR	855	3.5	24.4
PEC_CODER	1091	4.4	31.2
TOTAL	**3499**	**14.2**	**100.0**
Block RAMS	3	9.4	

Fig. 2.3: Resource usage for initial FAPEC code inside ProASIC3L FPGA

	M1A3P1000L	
	VersaTiles	**%**
Combinational (LUTs)	2724	-
Sequencial (DFFs)	706	-
TOTAL	**3430**	**14.0**
Block RAM	3	9.4

Fig. 2.4: Resource usage for initial FAPEC code inside ProASIC3L FPGA after Place & Routing

Regarding the timing analysis, the critical path for the compressor implementation was 18.32 ns, thus defining a theoretical maximum clock speed of ~55 MHz for ProASIC3L technology. This clock is used by the serial output, meaning that the maximum throughput of the compressed data would be 55 Mbit/s. The initial processing throughput requirement for this design was 2 Msample/s (i.e. 32 Mbit/s) and it was successfully achieved.

The goal for this design in VHDL was to implement the code in an RTAX device. Considering the information provided by the manufacturer (Microsemi), it would be possible to comfortably implement in parallel 2 FAPEC cores (aggregate input of 64 Mbit/s) with the low-end RTAX250S. The high-end RTAX4000S would theoretically allow more than 30 cores (aggregate input of more than 1 Gbit/s) using parallel data streams. It is very difficult to calculate the exact power consumption for the RTAX case because the underlying technology is different (antifuse in RTAX versus Flash in ProASIC). However, both technologies share the benefits of low start-up and static power consumption. In addition, their dynamic consumption is similar as well. Therefore the estimated consumption figure of a RTAX FPGA should be close to the 35 mW of the PROASIC3L prototype developed

2.3 The New Design of FAPEC

This initial design of FAPEC is going to be used as a reference for the development of the new FAPEC register-transfer level (RTL) design. This reference design presents a few problems that need to be addressed before FAPEC is suitable for integration with SpFi. The VHDL code developed in the reference design is not valid for integration in platforms other than ProASIC FPGAs and we intend to implement FAPEC with newer FPGA technologies. But the main issue is the fact that the compressed data output port is serial. SpFi inherently works with 32-bit data words and building a serial to parallel module would limit the throughput of FAPEC to that of the serial port (~55 Mbit/s for ProASIC3L). In the following sections we present the different changes introduced in the initial code to overcome all these problems.

2.3.1 Memories

The initial design used memory modules specifically generated for the Microsemi ProASIC3L FPGA family. One of the goals of this project is to decouple the VHDL code from a specific FPGA technology. Making the FAPEC compressor technology-agnostic provides a big advantage: it allows implementing FAPEC in available technologies with little effort. Equally important, this should guarantee support for future FPGAs, and even support ASIC implementation if required. This is because most FPGA synthesisers can automatically infer the memory modules if they are declared in specific ways. The original FAPEC design instantiated three different memory modules. However, after examining the code it appeared that only two different modules were strictly required.

The `ROM_TABLE` module was substituted by `ROM_TABLE_GNRC`. In the reference design this ROM module was created and initialised with the Libero Core Generation tool. This means that this module could only be used with a ProASIC3L. Instead, a generic VHDL module declaring an array of constant values was defined. An input address determines which array value is selected and output. This is what the code for this new module looks like:

```
type array_Nx8_t is array (natural range <>) of std_logic_vector(7 downto 0);
-- Replicate in this signal the LUT tables stored in the original
-- ROM_TABLE.mem file used by the ROM_TABLE component
constant lut_values : array_Nx8_t(0 to 127) := ("00000000",
                                                "00000001",
                                                ...
                                                "11111111");
sync_proc : process (Clk) is
begin
   if (rising_edge(Clk)) then
      -- Output selected value depending on the input address
      Dout <= lut_values(to_integer(unsigned(Addr)));

   end if;
end process sync_proc;
```

After synthesising this code with *Synplify* (synthesis tool provided by the *Libero* suite) the result is that 43 VersaTiles are used. This is roughly 0.17% of the ProASIC total resources or around 0.7% of the FAPEC implementation. Hence, we can conclude that the functionality is successfully inferred by the tool.

On the other hand, both RAM_HIST and RAM_DATA_BLCK modules were initially substituted by a generic DUAL_PORT_MEM module. This module corresponds to a dual-port memory featuring a single clock. Hence, two independent ports are available, each with read and write capabilities, although the same clock is used by both ports. During the verification stage it was observed that one of the modules was not operating as expected. The solution was to create two slightly different memory modules. DUAL_PORT_MEM_2 was created for the RAM_DATA_BLCK and uses a standard approach in which the output is updated at the next clock edge following an address port change. This is standard practice and the synthesis results revealed that memory was inferred automatically as expected.

RAM_HIST module simulation mismatch required the introduction of a slight variation in the memory behaviour. The module used (DUAL_PORT_MEM) is very similar to DUAL_PORT_MEM_2 with the exception that Port B output is pipelined. This means that port B output changes two clocks after the address port changes, not in the next clock. Port A output is not pipelined though. The following VHDL code shows in bold the difference between the two ports.

```
   buf_memory : process (Clk) is
      variable v_mem : buf_array_t;
   begin
      -- clocked memory
      if (rising_edge(Clk)) then

         -- Port A
         if (A_EN_N = '0') then
            if (A_RW = '1') then
               -- Read operation
               A_DOut <= v_mem(to_integer(unsigned(A_Addr)));
            else
               -- Write operation
               v_mem(to_integer(unsigned(A_Addr))) := A_DIn;
            end if;
         end if;

         -- Port B
         if (B_EN_N = '0') then
            if (B_RW = '1') then
               -- Read operation
               b_dout_r <= v_mem(to_integer(unsigned(B_Addr)));
               B_DOut   <= b_dout_r;
            else
               -- Write operation
               v_mem(to_integer(unsigned(B_Addr))) := B_DIn;
            end if;
         end if;

      end if;
   end process buf_memory;
```

By using these two different memory flavours, the operation of FAPEC was correctly simulated. These new memory declarations should allow to automatically infer memory blocks for most FPGAs. However, the asymmetric port behaviour of DUAL_PORT_MEM caused an issue when synthesising the code. This issue has been analysed in Section 2.4.1.

In general, it is recommended not to use pipelined output for memories. This produces a more natural behaviour as the output word can be read out of the memory in a clock cycle. If pipelined output is to be used, the same behaviour must be replicated in both ports of the memory to reduce complexity and simplify development efforts. Asymmetric behaviour can cause synthesis issues, as demonstrated with the `DUAL_PORT_MEM` module (Section 2.4.1).

2.3.2 Pre-compressor

A small change was introduced in the `PRECOMPRESSOR` module. The input sample value was not initially registered as the input model used kept this value constant for a few clock cycles. With the new design the sample value is updated after a read operation and thus it is required to internally register this value for the pre-compressor to operate as expected.

2.3.3 Histogram Constructor

The `HIST_CONSTRUCTOR` module has been optimised to reduce the number of clock cycles it takes to process a sample. In the reference design the constructor parsed the whole 37 bin values of the histogram each at a clock cycle. This meant that the minimum time between each input sample was ~40 clock cycles. As there are two of these modules operating in parallel this delay was effectively divided by two, but still constraining the input sample rate to one every ~20 clock cycles. A special function has been designed to calculate the corresponding bin number for the current value in a single clock cycle instead. This allows to process the histogram input values much faster. Additionally, thanks to the new memory modules, another optimisation has been done to save 2 clock cycles when incrementing the corresponding bin value.

The aggregate effect of these changes is that the new histogram module is able to process a new data value every 6 clock cycles. There is still room for improvement, but the changes have effectively increased speed by a factor of 4. Furthermore, when analysing the synthesis results, only 3% more VersaTile cells have been used with respect to the original module (see Fig. 2.8). Timing is now more constrained, but the critical path for the whole design is not related to this module.

2.3.4 Parallel-Output PEC Codec

The most important change applied to FAPEC, as explained before, has been to switch from serial to parallel output. A complete PEC coder module (`PEC_CODER`) has been designed. The inputs of this module are the same as the initial PEC coder, but instead of a single serial output, it now features four different parallel output ports, one for the coding table and three for each variant (i.e. LE, DS and LC). Each of these four parallel output ports is composed of a vector with the compressed value plus an additional signal carrying the number of bits valid in the compressed value output.

One of the firsts tasks to undertake when switching from serial to parallel is to dimension the size of the outputs. The following table (Fig. 2.5) calculates the maximum output values possible for any 16-bit input sample depending on the coding variant selected. Note that these are absolute maximum values regardless of the table coding values. The information on the number of valid bits is required because the vectors containing the compressed value have a fixed size which is determined by the worst case. However, normally fewer bits will be actually used for a given compressed value. For example, the maximum length for LE variant is 23 bits. Thus, the compressed value output port for LE will be 23-bit wide. But if a given value only requires 5 bits, the remaining 18 bits must not be used. So in this case the signal indicating the number of valid bits will indicate 5. In this way, the `WORD_PACKER` module knows how many bits to use from this 23-bit wide input every time a new value arrives. Additionally, a *Valid* signal validates the output whenever there is a new compressed value (or a coding table) to output.

	LE	**DS**	**LC**
h	2	4	< 16
i	2	< 16	< 16
j	< 16	< 16	< 16
k	16	16	16
Maximum Size	3+h+i+k 23 bits	3+h+k 23 bits	4+k 20 bits

Fig. 2.5: Maximum compressed value sizes depending on the Coding Variant

These values are important, as they will determine the maximum delay that can be expected when trying to parallelise this output into chunks of a specific size. For example, if 16-bit output words were to be used, this would mean that a single coded value could be spread into 3 different words. This constraint will actually be used in the parallelising module (`WORD_PACKER`) presented in the next section.

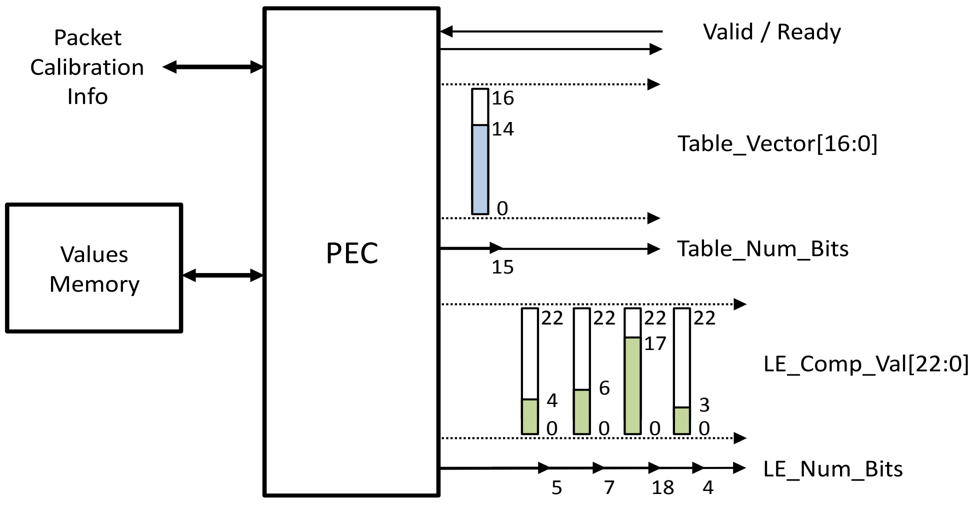


Fig. 2.6: Example of PEC operation

Fig. 2.6 shows an example of the PEC module outputting a coding table of 15 bits length, followed by four compressed values using the Low-Entropy variant (LE) with lengths of 5, 7, 18 and 4 bits respectively. These values are then received by the `WORD_PACKER` module which concatenates them in order to form the final compressed bit stream. This bit stream is output in chunks of 32 bits at a time. Obviously, not every clock cycle 32 bits will be ready for output. A valid signal asserted for a clock cycle indicates when this data can actually be read.

The module port declaration has been copied here. The whole file can be found at Annex 5.1.

```
entity pec_coder is

   port (
      Clk          : in std_logic;
      Reset        : in std_logic;
      Table_Valid : in std_logic;

      -- From encoder side
      Coding_Variant     : in std_logic_vector(1 downto 0);
      Segment_1_Num_Bits : in unsigned(LOG2_SSYZE-1 downto 0);
      Segment_2_Num_Bits : in unsigned(LOG2_SSYZE-1 downto 0);
      Segment_3_Num_Bits : in unsigned(LOG2_SSYZE-1 downto 0);
      Segment_4_Num_Bits : in unsigned(LOG2_SSYZE-1 downto 0);
      Ceiling_1_Val      : in unsigned(SYMBOL_SIZE-1 downto 0);
      Ceiling_2_Val      : in unsigned(SYMBOL_SIZE-1 downto 0);
      Ceiling_3_Val      : in unsigned(SYMBOL_SIZE-1 downto 0);

      -- To Word_Packer module
      Ready           : in  std_logic;
      Table_Valid_Out : out std_logic;
      Table_Num_Bits  : out unsigned(LOG2_SSYZE downto 0);
      Table_Vector    : out std_logic_vector(TAB_LONG_REF-1 downto 0);

      Comp_Sample_Valid  : out std_logic;
      Coding_Variant_Out : out std_logic_vector(1 downto 0);
      LE_Num_Bits        : out unsigned(4 downto 0);
      DS_Num_Bits        : out unsigned(4 downto 0);
      LC_Num_Bits        : out unsigned(4 downto 0);
      LE_Comp_Val        : out std_logic_vector(22 downto 0);
      DS_Comp_Val        : out std_logic_vector(22 downto 0);
      LC_Comp_Val        : out std_logic_vector(19 downto 0);

      -- Memories Management
      RD    : in  std_logic_vector(SYMBOL_SIZE downto 0);  -- Pixel/value + sign to
compress as read from the block RAM
      RADDR : out std_logic_vector(LOG2_BSIZE downto 0);  -- Address to read of the
block RAM (2 x block size)
      REN   : out std_logic  -- read enable for the RAM / LOW ACTIVE
      );

end entity pec_coder;
```

The fundamental operation of PEC has obviously not been altered as the compressed output must be the same. However, the operation of this new module is radically different. Table coding values are calculated in a single clock cycle. PEC compressed values, on the other hand, require two clock cycles to reduce the timing stress on the operation. On the first clock cycle the binary value to be coded within a given segment is calculated. This value depends on the segment number to be used, which depends on the ceiling values passed to PEC. In the second clock the entire bit stream is calculated (prefix/unary sequence, escape sequences, etc.) together with the length of the output vector. Finally, an additional clock cycle is added to wait for the ready signal coming from the `WORD_PACKER` module. When it is ready, the histogram memory address is incremented and in the next clock cycle a new value to code arrives to the PEC module.

This PEC implementation can output compressed data much faster than the initial version. Currently, the new PEC is limited to compressing one value every 3 clock cycles. This is enough for the current operation as the HIST_CONSTRUCTOR can only process one sample every 6 clock cycles. It is however important to remark that there is no fundamental limitation on speed in the way PEC is currently constructed. By adding a pipeline stage and optimising the way in which samples are obtained from the memory, the module can be modified to compress a value every clock cycle. If these modifications are implemented adequately, timing should not be noticeably affected.

2.3.5 Generation of 32-bit FAPEC Output

The new PEC coder outputs parallel data in a very particular way, as explained in the previous section. There is a separate port output for the coding table and every coding variant, together with a port stating the number of valid bits. However, the expected output of a paralleliser is a single fixed-size port, e.g. 16 or 32 bits. Transmission modules such as SerDes or data buses always have a fixed number of bits as input port width. For example, the SpFi protocol uses a native bus width of 32 bits as user interface. This seems a good trade-off value for the port width and has hence been the adopted width for FAPEC.

A new module (WORD_PACKER) has been created for this purpose. This module gets the output ports of PEC and generates a fixed-width output of 32 bits. The module port declaration has been copied here. The whole file can be found at Annex 5.2.

```vhdl
entity word_packer is

   port (
      Clk   : in std_logic;
      Reset : in std_logic;

      -- From PEC compressor
      Ready            : out std_logic;
      Table_Valid_Out  : in  std_logic;
      Table_Num_Bits   : in  unsigned(LOG2_SSYZE downto 0);
      Table_Vector     : in  std_logic_vector(TAB_LONG_REF-1 downto 0);

      Comp_Sample_Valid   : in std_logic;
      Coding_Variant_Out  : in std_logic_vector(1 downto 0);
      LE_Num_Bits         : in unsigned(4 downto 0);
      DS_Num_Bits         : in unsigned(4 downto 0);
      LC_Num_Bits         : in unsigned(4 downto 0);
      LE_Comp_Val         : in std_logic_vector(22 downto 0);
      DS_Comp_Val         : in std_logic_vector(22 downto 0);
      LC_Comp_Val         : in std_logic_vector(19 downto 0);

      -- To VC buffer
      VCB_Half_Full : in  std_logic;
      Out_Valid     : out std_logic;
      Out_Data      : out std_logic_vector(31 downto 0)
      );

end entity word_packer;
```

A two-stage strategy has been adopted inside this module to reduce the timing stress. This module needs to multiplex between four different parallel data inputs – the Table Coding and the three coding variants, LE, DS and LC – each with a variable number of bits. The first stage consists of merging the four different incoming data streams into a single registered signal. There is a lot of multiplexing involved here, and this signal is registered to improve the timing. In the next clock cycle, this signal is then used to pad the 16 bits of an intermediate buffer. Note that depending on the size of the incoming data value, up to 3 clock cycles might be required to write all the input bits in this 16-bit buffer, as depicted in the example of Fig. 2.7. In this example only 2 bits can be inserted in the buffer during the first clock cycle. This is because the buffer already had 14 bits occupied by the previous compressed value. Of the remaining 20 bits, 16 bits can then be written down in the second clock cycle, while the last 4 bits have to wait until the third clock cycle.

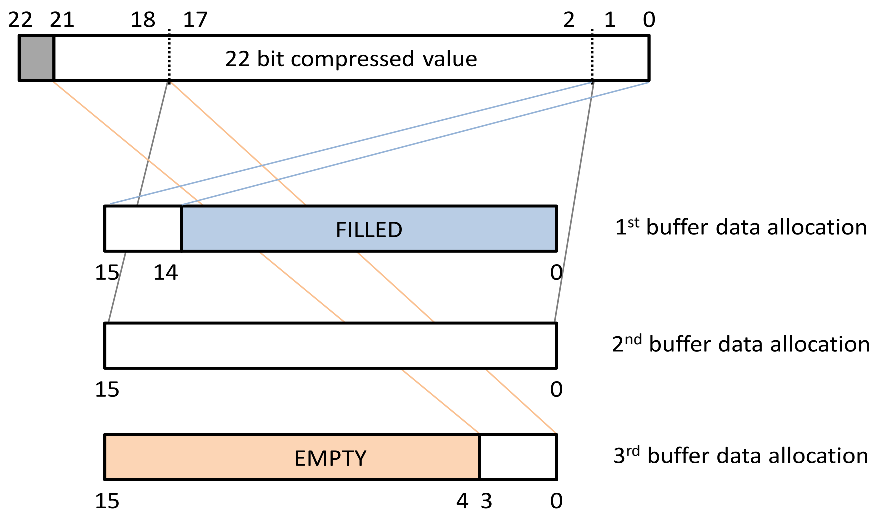

22
21
18
17
2
1
0
22 bit compressed value
FILLED
15
14
0
1st buffer data allocation
2nd buffer data allocation
15
0
EMPTY
15
4
3
0
3rd buffer data allocation

Fig. 2.7: Operation of writing a large value in the 16-bit intermediate buffer

Finally, the last stage consists of writing the 16-bit buffer into the corresponding half of the
32-bit final output. Once the output vector has 32 valid bits the output is validated. This output corresponds to the parallel output of the FAPEC module.

Note that only 16 bits are used as size for the intermediate buffer width instead of what would be the natural 32 bit vector. This size has been adopted to reduce the risk of using large vectors. The large number of multiplexors required by this operation would set timing restrictions that could render impossible the effort to adapt FAPEC to high operating frequencies.

2.3.6 Optimising FAPEC Speed

The initial VHDL version of FAPEC used a serial output and this output required a dedicated clock. Current FPGA technologies (especially the space-qualified ones) do not support operating frequencies beyond the 200-300 MHz range. Moreover, values in the high part of the frequency range are really hard to achieve for applications other than basic combinatorial operations. Consequently, there was an intrinsic strong limitation to the speed of a FAPEC module inside an FPGA due to this serial output. The logical step to overcome these limitations has been to switch to a parallel output. On the other hand, dedicated SerDes blocks are commonly available in modern FPGAs. They are analogue modules integrated in the chip die, and they operate taking input parallel data streams and outputting them in serial format. This is the technology that SpFi needs to use in its physical layer (bottom layer of Fig. 1.2). FPGA that do not feature inbuilt SerDes (e.g. radiation-tolerant RTAX family from Microsemi) can still make use of external SerDes modules. For space, a variant of the TLK2711 WizardLink is available from Texas Instruments. The TLK2711-SP is a space-rated transceiver reaching up to 2.5 Gbps [14].

The PROASIC3L reference design has its serial link operating at a 40 MHz. This limits the output of the FAPEC codec at 40 Mbit/s. Thus, the input frequency of values is not required to be very high, typically in the range of ~5 MHz. The new parallel output eliminates this limitation. The new PEC module is able to compress a sample every three clock cycles. This three-clock limitation arises from two facts: the steps the current algorithm requires to code a sample, but also from the fact that the worst case maximum length for a compressed value is 23 bits. The intermediate parallel size is 16 bits, which produces a three clock cycle worst-case passing from 23 to 16 bits (see Fig. 2.7). Nevertheless, if normal compression ratios are assumed, it is theoretically possible to compress and output samples every clock cycle on average.

The current version of the histogram generator is now able to process one input sample every 6 clock cycles. The initial performance was much worse, working at 1 sample every ~20 clock cycles. This is currently the most limiting module regarding speed. So the histogram generation is the current bottleneck for the overall speed of FAPEC. There is no theoretical reason as to why a sample could not be processed every clock cycle. The histogram already works with two memories, so while one data block is being processed and its histogram generated, the previous data block can be compressed. This allows not stopping the input data flow while compressing data blocks. However, processing one sample per clock requires major changes in the histogram generation logic.

2.4 Verification procedure

When developing new code, verification is always one of the most critical stages. Thanks to the reference design presented above, there is already a FAPEC module that can be used as a reference for verification. Thus, simulations of the new codec have benefited from the fact that there was an initial version to compare with. This helped to reduce debugging times. The strategy used to verify this new FAPEC was hence slightly different to the one used for the reference design.

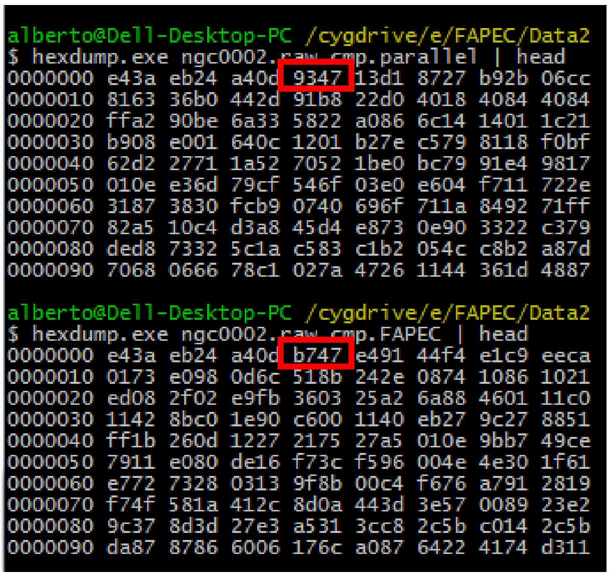
In the reference design an incremental verification approach was used. This allowed verifying the different modules by taking advantage of the previously verified modules. Also, a set of test files was created for the reference design. This set of files included different scenarios, so that all the coding variants were tested. It also included corner-cases to test the compressor under the most stressing scenarios (e.g. values after the pre-compressing stage that were always 0 or always 65535). The compressed files were compared at binary level against the output of the equivalent software version of FAPEC. In this way it was guaranteed that the compressor was working correctly. As the main functional changes in the new FAPEC have been applied at the last stage of the compressor

(PEC), the verification has been performed by directly compressing the set of files. This is in contrast with the incremental validation approach used for the reference design.

The *Modelsim* simulator tool has been used to verify the operation of FAPEC. The errors have been found and debugged directly with the simulation of these files. Whenever there was a mismatch between files, the difference was located in the binary file and then tracked down to the simulation. For example, first a *cmp* command is run between the reference and the compressed files:

```
alberto@Dell-Desktop-PC /cygdrive/e/FAPEC/Data2
$ cmp ngc0002.raw.cmp.parallel ngc0002.raw.cmp.FAPEC
ngc0002.raw.cmp.parallel ngc0002.raw.cmp.FAPEC differ: byte 7, line 1
```

If there is a difference, *hexdump* command is then used to find the exact difference with the information provided by *cmp* (Fig. 2.8).



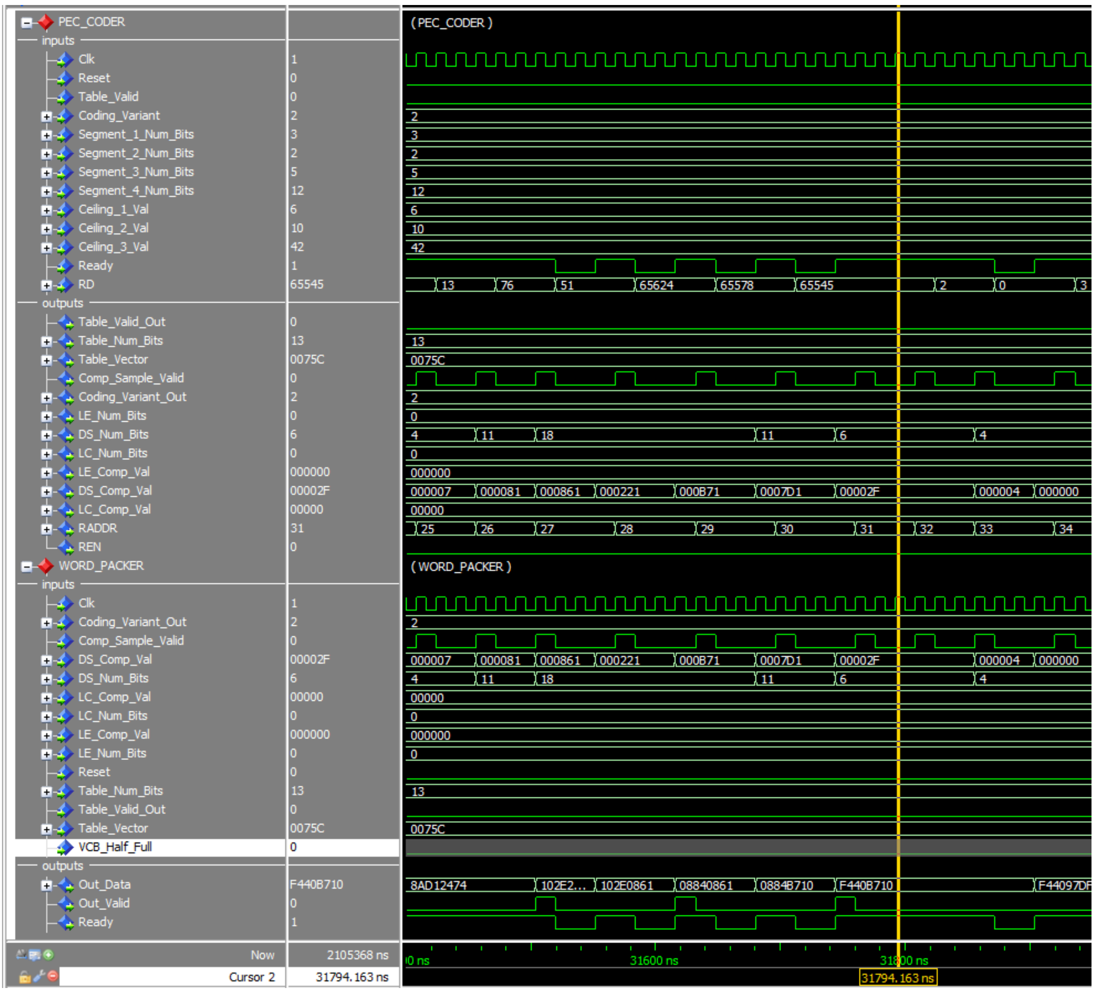
```
alberto@Dell-Desktop-PC /cygdrive/e/FAPEC/Data2
$ hexdump.exe ngc0002.raw.cmp.parallel | head
0000000 e43a eb24 a40d 9347 13d1 8727 b92b 06cc
0000010 8163 36b0 442d 91b8 22d0 4018 4084 4084
0000020 ffa2 90be 6a33 5822 a086 6c14 1401 1c21
0000030 b908 e001 640c 1201 b27e c579 8118 f0bf
0000040 62d2 2771 1a52 7052 1be0 bc79 91e4 9817
0000050 010e e36d 79cf 546f 03e0 e604 f711 722e
0000060 3187 3830 fcb9 0740 696f 711a 8492 71ff
0000070 82a5 10c4 d3a8 45d4 e873 0e90 3322 c379
0000080 ded8 7332 5c1a c583 c1b2 054c c8b2 a87d
0000090 7068 0666 78c1 027a 4726 1144 361d 4887

alberto@Dell-Desktop-PC /cygdrive/e/FAPEC/Data2
$ hexdump.exe ngc0002.raw.cmp.FAPEC | head
0000000 e43a eb24 a40d b747 e491 44f4 e1c9 eeca
0000010 0173 e098 0d6c 518b 242e 0874 1086 1021
0000020 ed08 2f02 e9fb 3603 25a2 6a88 4601 11c0
0000030 1142 8bc0 1e90 c600 1140 eb27 9c27 8851
0000040 ff1b 260d 1227 2175 27a5 010e 9bb7 49ce
0000050 7911 e080 de16 f73c f596 004e 4e30 1f61
0000060 e772 7328 0313 9f8b 00c4 f676 a791 2819
0000070 f74f 581a 412c 8d0a 443d 3e57 0089 23e2
0000080 9c37 8d3d 27e3 a531 3cc8 2c5b c014 2c5b
0000090 da87 8786 6006 176c a087 6422 4174 d311
```

Fig. 2.8: *Hexdump* output example for the original file (bottom) and the new algorithm (top), marking the first difference between files

Once the first failing bit is located, it is possible to go to the Modelsim simulation to track down the problem. Fig 2.9 shows the input and output ports of the PEC_CODER and WORD_PACKER modules in a simulation. The 32-bit words that FAPEC outputs are at the bottom of the figure, in the *Out_Data* port. When the word to be examined is located, then it is possible to further expand the WORD_PACKER module for a more in depth inspection. By looking at the internal signals it is usually possible to figure out whether it is performing correctly or not. If it is working fine, then the problem might be in the module coming before (PEC_CODER). The operation is then repeated, find the outputs towards WORD_PACKER that are causing the wrong output and then figure out what is the problem. With this method all the bugs in the code of the new FAPEC design where identified and solved.

Once all the problems have been fixed, the binary comparison between the two files has to report End Of File (EOF) found, meaning that the end of the file was reached without finding any difference. All the files in the set have been successfully compressed and compared with the new FAPEC design.

```
alberto@Dell-Desktop-PC /cygdrive/e/FAPEC/Data2
$ cmp ngc0002.raw.cmp.parallel ngc0002.raw.cmp.FAPEC
cmp: EOF on ngc0002.raw.cmp.parallel
```



```
PEC_CODER
 inputs
   Clk                   1
   Reset                 0
   Table_Valid           0
   Coding_Variant        2
   Segment_1_Num_Bits    3
   Segment_2_Num_Bits    2
   Segment_3_Num_Bits    5
   Segment_4_Num_Bits    12
   Ceiling_1_Val         6
   Ceiling_2_Val         10
   Ceiling_3_Val         42
   Ready                 1
   RD                    65545
 outputs
   Table_Valid_Out       0
   Table_Num_Bits        13
   Table_Vector          0075C
   Comp_Sample_Valid     0
   Coding_Variant_Out    2
   LE_Num_Bits           0
   DS_Num_Bits           6
   LC_Num_Bits           0
   LE_Comp_Val           000000
   DS_Comp_Val           00002F
   LC_Comp_Val           00000
   RADDR                 31
   REN                   0
WORD_PACKER
 inputs
   Clk                   1
   Coding_Variant_Out    2
   Comp_Sample_Valid     0
   DS_Comp_Val           00002F
   DS_Num_Bits           6
   LC_Comp_Val           00000
   LC_Num_Bits           0
   LE_Comp_Val           000000
   LE_Num_Bits           0
   Reset                 0
   Table_Num_Bits        13
   Table_Valid_Out       0
   Table_Vector          0075C
   VCB_Half_Full         0
 outputs
   Out_Data              F440B710
   Out_Valid             0
   Ready                 1
Now        2105368 ns
Cursor 2   31794.163 ns
```

Fig. 2.9: Modelsim simulation of the IO ports of PEC_CODER and WORD_PACKER modules

2.5 Performance Analysis

In this section we analyse the differences in resource usage, timing and performance of the new FAPEC design with respect to the reference design.

2.5.1 ProASIC3L Resource Usage Analysis

In Fig. 2.10 the table with the new FAPEC resource usage for ProASIC3L FPGA is indicated. These values can be compared with Fig. 2.3 which contains the usage for the reference design. An additional column has been added at the right side of the table in Fig. 2.10. This represents the increase or decrease (in %) of the number of used VersaTiles from the reference design to this new design. A positive value means more resources used by the new FAPEC compressor, and negative means fewer resources used, i.e. negative is good.

	M1A3P1000L			
	VersaTiles	**%**	**% of FAPEC**	**% wrt Ref Design**
PRECOMPRESSOR	358	1.5	5.9	+ 7.2
HIST_CONSTRUCTOR	858	3.5	14.2	+ 2.9
HIST_BOUNDARY_EXTRACT	346	1.4	5.7	- 0.9
TABLE_CONSTRUCTOR	1045	4.3	17.3	+ 22.2
PEC_CODER	2612	10.6	43.2	+ 139.4
WORD_PACKER	781	3.2	12.9	N/A
TOTAL	**6043**	**24.6**	**100.0**	**+ 72.7**
Block RAMS	2	6.3		- 33.3

Fig. 2.10: Resource usage in the ProASIC FPGA for the new FAPEC and its comparison against the reference design

Changes in the first three modules are irrelevant (PRECOMPRESSOR, HIST_CONSTRUCTOR and HIST_BOUNDARY_EXTRACT). The 22% increase of the TABLE_CONSTRUCTOR is a curious case. Its usage has significantly increased despite the fact that no changes at all have been applied to this module. The reason for this variation is the way the synthesiser optimises resources. The external ports of the TABLE_CONSTRUCTOR module are connected to the new PEC Coder. The way in which PEC internally connects its inputs has been completely changed, as it features now a parallel output. Therefore, the way in which the TABLE_CONSTRUCTOR output ports are connected inside PEC has changed. This has prevented *Synplify* from doing more optimisations, as it did with the reference design, hence the different results.

A very important difference between the two memory modules has not been reflected in the previous table but it is worth mentioning here. The DUAL_PORT_MEM_2 module has been reported as not taking any logical resources at all (i.e. 0 VersaTiles). However, the DUAL_PORT_MEM module uses roughly 3000 VersaTiles. This is a huge number of tiles, considering the whole new FAPEC module uses ~6000 tiles (the 3000 tiles of the DUAL_PORT_MEM are not included). The reason for this asymmetry between two almost identical memory modules is the asymmetric operation of the output ports for DUAL_PORT_MEM. This is causing the Synthesis tool to infer a large amount of wrapping logic because a memory block alone cannot reproduce the behaviour indicated in the VHDL code. Therefore, port B memory pipeline stage of the DUAL_PORT_MEM should be removed. This port is connected to the HIST_BOUNDARY_EXTRACT and it seems that this behaviour can be changed by placing the pipelining stage inside the HIST_BOUNDARY_EXTRACT module and removing it from the memory declaration.

The parallel PEC_CODER has increased his usage in 140% – a huge increase – due to the big difference in architecture. This means that the initial PEC module is more simple and compact than this new module, but also much slower. The higher speed does not come for free.

If we pay attention to the usage percentage of the different modules with respect to the full FAPEC coder, the most obvious consequence is that, due to the increase in resources used by PEC, the rest of the modules now take a lower percentage than in the reference design. This effect is further increased by the new `WORD_PACKER` module, which is not present in the old design and that takes a 13% of the total number of VersaTiles used by the new FAPEC.

Note also that the number of block RAM has gone from 3 to 2. This is due to the fact that the ROM memory (`ROM_TABLE_GNRC`) is hardcoded in VHDL and not implemented with a memory. As explained in Section 2.3.1, only 43 tiles (0.17% of the FPGA area) were used by this ROM implemented with logical resources.

Finally, if we consider the total number of tiles used by FAPEC, we see that it has increased a 73% in the new version. The reference design uses 3500 tiles for a total usage of 14% of the FPGA, whereas the new version uses 6043 tiles for an almost 25% of FPGA used. The change is significant, and this is the toll that FAPEC has paid for a faster operation and a parallel output.

	RTG4			
	Regs	**%**	**LUTs**	**%**
`PRECOMPRESSOR`	71	0.1	69	0.1
`HIST_CONSTRUCTOR`	205	0.1	304	0.2
`HIST_BOUNDARY_EXTRACT`	64	0.0	167	0.1
`TABLE_CONSTRUCTOR`	153	0.1	337	0.2
`PEC_CODER`	191	0.1	1548	1.0
`WORD_PACKER`	111	0.1	609	0.4
TOTAL	**829**	**0.6**	**3078**	**2.0**
Block RAMS	2	0.5		

Fig. 2.11: Resource usage in the RTG4 FPGA for the new FAPEC design

2.5.2 RTG4 Resource Usage Analysis

RTG4 is a new radiation-tolerant FPGA developed by Microsemi [6]. It is the evolution of the successful RTAX family although, in this case, the technology used is completely different. RTG4 uses Flash technology, offering reprogrammability, and also has 24 embedded SerDes cores, among other advanced characteristics. The original FAPEC reference design was aiming at RTAX as target for space applications. However, given that the new RTG4 platform has already been used to implement SpFi, and that it has created a lot of interest in the space community, it makes sense to examine its performance with FAPEC.

Fig. 2.11 shows the table with the usage values obtained for the RTG4. As the numbers in the table indicate, the RTG4 is a much bigger device. Furthermore, it also is much faster than the RTAX. In this case, RTG4 features separated combinational (Look-Up Tables, or LUTs) and sequential elements (registers). A register is a flip-flop, and it stores a bit of information that is updated every clock cycle. LUTs, on the other hand, are used to recreate the operation of logical functions and multiplexers. The `PEC_CODER` and `WORD_PACKER` modules make an intensive use of multiplexers to be able to place the compressed binary values into any given bit of the 32-bit output port. The effect of this large number of multiplexers required can be seen in the table values. This explains why the number of LUTs in the RTG4 is much higher than the number of registers. Note that the ratio between sequential (Registers) and combinational (LUTs) elements, 3.7, is similar to the one obtained with the reference design in the ProASIC device, 3.9 (see Fig. 2.10).

In terms of total logical resources used by FAPEC, it can be claimed that the logic average usage of the device is only a 1.3%. This means that FAPEC can currently be implemented inside a design using the RTG4 with almost no impact.

2.5.3 ProASIC3L Timing Analysis

The target speed of the reference design for its fast clock was 40 MHz. The timing analysis determined that the theoretical maximum frequency for the output serial clock (`CLK`) was 42.6 MHz (see Fig. 2.12). This limits the maximum output data rate to 42.6 Mbit/s. On the other hand, the new FAPEC presented a maximum estimated frequency of 31.6 MHz for the same clock. This shows how the changes used to optimise the FAPEC operation have increased its complexity. Higher complexity means a greater number of logical levels, which translate in more net delays due to the greater number of components crossed by the net paths and also the increased net lengths. These two effects cause higher delays in signals travelling from one register to another, thus limiting the maximum frequency speed. Nevertheless, maximum frequency can still be improved. The options to achieve better timing are analysed in next Section.

	Ref Design (MHz)	**New FAPEC (MHz)**	**Δ (%)**
`CLK`	42.6	31.6	- 25.8
`SCLK`	31.3	52.1	+ 66.5

Fig. 2.12: Maximum frequency of the clock domains in ProASIC for the initial and new FAPEC designs

Note that the new compressor output is now parallel. This would imply a theoretical maximum throughput of 31.6 Msamples/s (with 16 bit/sample ≈ 500 Mbit/s) – although to achieve this performance further changes in the compressor are required – while the theoretical maximum speed for the reference design equals that of the clock domain (42.6 Mbit/s) due to its serial output.

The performance of the other clock (SCLK) is not that important. As a matter of fact, in the new implementation both clocks share the same clock source. Therefore, the timing limitation comes from the clock presenting the slowest path, i.e. CLK.

2.5.4 RTG4 Timing Analysis

It is more significant to analyse the potential of the new FAPEC with modern FPGAs. As mentioned before, the new RTG4 has created a lot of expectation among the space community. It is a big FPGA and SpFi only takes around 2 – 3 % of the resources [15]. It is thus realistic to think about the potential integration of FAPEC and SpFi modules inside a design running in an RTG4.

	New FAPEC (MHz)
CLK	55.0
SCLK	127.2

Fig. 2.13: Maximum frequency of the clock domains in RTG4 for the new FAPEC design

When analysing the timing performance in the RTG4, the initial value observed does not seem very high. It is roughly a 30% higher than the ProASIC value, but still it does not imply a great performance increase (Fig. 2.13). However, after a close examination of the reported paths, it becomes obvious that this maximum frequency is far from being the limit of what can be expected from the new FAPEC. Specifically, the initial 60 critical paths reported by the tool all look similar to this one:

```
Path information for path number 60:
      Requested Period:                4.557
    - Setup time:                      0.229
    + Clock delay at ending point:     0.000 (ideal)
    = Required time:                   4.329

    - Propagation time:                5.980
    - Clock delay at starting point:   0.000 (ideal)
    = Slack (non-critical) :           -1.652

    Number of logic level(s):          0
    Starting point:                    INSTANTIATE_HIST_BLCK_MEM\.block_mem_generic\.BLCK_MEM.v_mem_v_mem_0_0 / A_DOUT[16]
    Ending point:                      INSTANTIATE_PEC\.PEC_COD.SIGN / D
    The start point is clocked by      FAPEC|CLK [rising] on pin A_CLK
    The end   point is clocked by      FAPEC|CLK [falling] on pin CLK

Instance / Net                                                                Pin            Pin              Arrival
Name                                                             Type         Name           Dir     Delay    Time
-----------------------------------------------------------------------------------------------------------------------
INSTANTIATE_HIST_BLCK_MEM\.block_mem_generic\.BLCK_MEM.v_mem_v_mem_0_0   RAM1K18_RT   A_DOUT[16]     Out     4.997    4.997
RD[16]                                                           Net          -              -       0.983    -
INSTANTIATE_PEC\.PEC_COD.SIGN                                    SLE          D              In      -        5.980
=======================================================================================================================
Total path delay (propagation time + setup) of 6.209 is 5.225(84.2%) logic and 0.983(15.8%) route.
Path delay compensated for clock skew. Clock skew is added to clock-to-out value, and is subtracted from setup time value
```

Basically, they indicate that the paths are related to the use of combination of rising and falling clock edges (in bold). This is not a recommended practice for RTL code and should be avoided whenever possible. The use of the different clock edges in the new FAPEC is due to legacy, as this was already used in the reference design. There is no real justification for the new version to use this clocking scheme. Furthermore, fixing this does not seem very difficult. The consequence of mixing rising and falling edges is that the requested period is half of the *real* clock period, because the tool is analysing the timing between the falling and the rising edge of a clock for these paths. It is also worth noting that memories are slow and it typically takes almost 5ns for the value to be valid out of the memory in RD port (in bold).

In the timing report 1000 paths were requested from the tool. After the initial 60 paths analysed above, the remaining 940 paths were all related to the multiplexing operations in the PEC module:

```
Path information for path number 61:
      Requested Period:                      9.115
    - Setup time:                            0.264
    + Clock delay at ending point:           0.000 (ideal)
    = Required time:                         8.850

    - Propagation time:                      10.459
    - Clock delay at starting point:         0.000 (ideal)
    = Slack (non-critical) :                 -1.609

    Number of logic level(s):                30
    Starting point:                          INSTANTIATE_HIST_BLCK_MEM\.block_mem_generic\.BLCK_MEM.v_mem_v_mem_0_0 / A_DOUT[0]
    Ending point:                            pec_opt_1.segment_value_r[15] / D
    The start point is clocked by            FAPEC|CLK [rising] on pin A_CLK
    The end   point is clocked by            FAPEC|CLK [rising] on pin CLK

. . .
. . .
. . .

Path information for path number 1000:
      Requested Period:                      9.115
    - Setup time:                            0.264
    + Clock delay at ending point:           0.000 (ideal)
    = Required time:                         8.850

    - Propagation time:                      10.253
    - Clock delay at starting point:         0.000 (ideal)
    = Slack (non-critical) :                 -1.403

    Number of logic level(s):                20
    Starting point:                          INSTANTIATE_HIST_BLCK_MEM\.block_mem_generic\.BLCK_MEM.v_mem_v_mem_0_0 / A_DOUT[8]
    Ending point:                            pec_opt_1.segment_value_r[15] / D
    The start point is clocked by            FAPEC|CLK [rising] on pin A_CLK
    The end   point is clocked by            FAPEC|CLK [rising] on pin CLK
```

The *starting point* of the path is always the output memory data port and the *ending point* is a register in the PEC coder. There are two problems here. Firstly, as shown above, the memory output is very slow and it takes 5 ns for the data to get out of the memory. This means that outputting data from the memory – without doing any operation with this output value and not accounting for any net delay – can only work at a maximum operation frequency of 200 MHz (1/5 ns). And this relates to the second problem, which is the large number of logic levels due to the multiplexing currently required by PEC. Specifically, path 61 has 30 logic levels, which is a huge number. When combined, these two issues considerably limit the maximum frequency.

The first problem described above presents a relatively easy solution. The operation of the PEC coder can be modified to work with a registered value instead of directly working with the value coming from the memory. This way the memory delay will not apply to the worst path, and also has the advantage of reducing net delays because in general registers can be placed much closer to the combinational logic (i.e. LUTs) than memories. A quick feasibility test has been run to examine the potential performance that FAPEC can achieve if the memory value were to be registered inside the PEC module. The *RD* input (the read value coming from the memory) in PEC has been registered with the `CLK` signal. The internals of PEC have not been further modified, meaning that the output of the PEC module is not correct, although this modified module presents a valid performance in terms of timing analysis. Now the timing is different:

```
Path information for path number 61:
      Requested Period:                      7.861
    - Setup time:                            0.264
    + Clock delay at ending point:           0.000 (ideal)
    = Required time:                         7.596

    - Propagation time:                      8.983
    - Clock delay at starting point:         0.000 (ideal)
    = Slack (non-critical) :                 -1.387

    Number of logic level(s):                10
    Starting point:                          word_packer_1.ptr_r_fast[1] / Q
    Ending point:                            word_packer_1.half_word_r[4] / D
    The start point is clocked by            FAPEC|CLK [rising] on pin CLK
    The end   point is clocked by            FAPEC|CLK [rising] on pin CLK
```

The results obtained are really promising. Requested period of 7.861 ns corresponds to 127.2 MHz. Slack is -1.387 ns, which if added to the requested period, means that the module can potentially run up to ~108 MHz only by fixing falling edge clocks and registering the memory input of PEC. In fact, these results point to a much faster potential with a few other improvements. All the new paths down to number 1000 have been examined again. With no exception, all paths are either located inside the `PEC_CODER` or the `WORD_PACKER` modules. This is due to the large number of multiplexing operations performed inside these two modules. However, slack at path 1000 is -0.318 ns, which means that with additional effort to increase the pipelining in both modules we can expect FAPEC to run at least at 125 MHz. This is a magical figure for the SpFi integration. Typically, SpFi links run at 2.5 Gbit/s, meaning that continuous data input at 62.5 MHz is required for the link to saturate. This frequency is determined by the 32-bit user data path. As currently FAPEC operates with 16-bit samples, then running FAPEC at 125 MHz would be equivalent to 32-bit at 62.5 MHz.

2.5.5 Post Place and Routing Analysis

The place and routing operation has been carried out for the FAPEC compressor with a Xilinx Spartan-6 FPGA. The technology used by Spartan-6 is very similar to that of Virtex-5QV (a space-qualified device also from Xilinx) and it can thus be used as a benchmark for the Virtex-5QV resource utilisation.

Results are displayed in Fig. 2.14. As both Spartan and Virtex use LUT-6 tables, the number of LUT and DFF elements should be very similar. As noted for both ProASIC and RTG4 devices, the number of combinatorial logic used is much larger than the number of sequential elements. In the case of Spartan-6 the ratio is ~5, considerably higher than values (~4) obtained for the other two FPGAs. It is worth nothing that the values indicated in this table have been obtained from the actual implementation in the final design (integrated with SpFi). Hence, it could be that the Place and Route tool (Xilinx XST) has done certain changes to improve performance taking into consideration the whole design.

	SPARTAN-6 75T		**VIRTEX-5QV**	
	Cells	**%**	**Cells**	**%**
Combinational (LUTs)	3736	8.0	3736	4.3
Sequential (DFFs)	763	0.8	763	0.9
Logic Average Use		**4.4**		**2.6**
Block RAM	1	0.6	1	0.3

Fig. 2.14: Spartan-6 and Virtex-5QV utilisation after Place and Routing for the new FAPEC design

As happened in the RTG4 resource analysis, the new FAPEC logic average usage is less than 3% for the Virtex-5QV. This means that FAPEC can too be implemented in a Virtex-5QV device with a small impact in the global design.

3. The STAR Fire Design

In this section we explain the process of integrating the new version of the FAPEC compressor as an intermediate stage at the transmit Virtual Channel (VC) of a SpaceFibre codec.

3.1 Introduction

The STAR Fire is ground support equipment specifically designed to support the evaluation and early adoption of SpaceFibre technology (Fig. 3.1). It is a SpFi Diagnostic Interface and Analyser that provides a complete SpFi test and development solution. The STAR Fire unit has two SpFi interfaces with an embedded link analyser, two SpW ports, multiple very high data rate inbuilt data pattern generators and checkers, and an embedded SpW router. STAR Fire can operate as a bridge between SpW and SpFi, as a SpFi link analyser, as a rapid SpFi packet generation and checking unit, and as a decoder of SpFi signals for operation with a Logic Analyser.

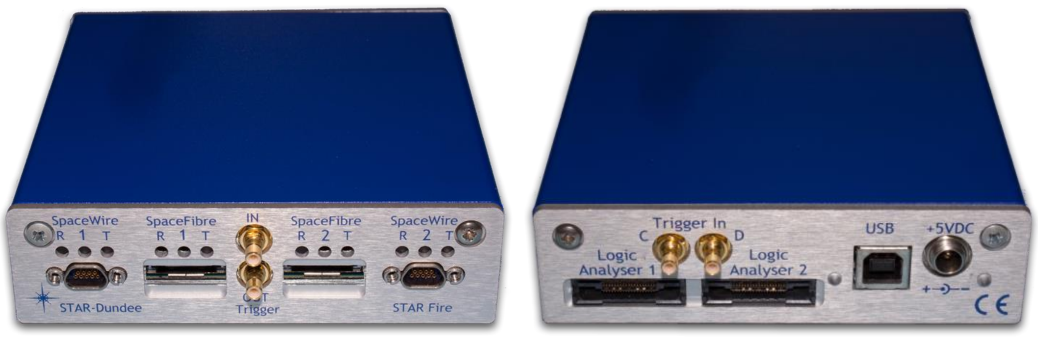
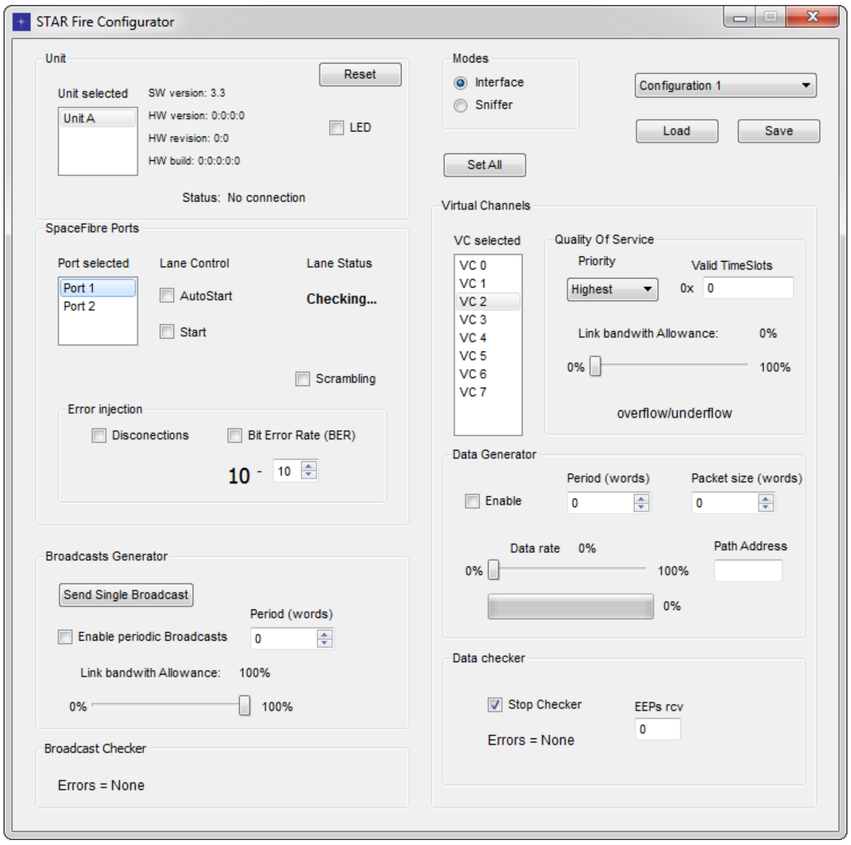


Fig. 3.1: The STAR Fire Mk2 Unit front panel (left) and bottom panel (right)

STAR Fire features a USB port which provides communications with a host PC, allowing to interface SpFi with a computer. Unfortunately only USB 2.0 connection is allowed which does not allow sending enough data to saturate the link. Nevertheless, the inbuilt basic internal data generators and checkers can be used to force SpFi to send data at maximum speed. The USB interface also provides status and control communications from the PC, thanks to specially designed software.

The STAR Fire software is based on a Graphical User Interface (GUI) that allows the configuration of the SpFi interfaces and the use of the embedded link analyser. It also controls the parameters of the data generators and monitors the status of the data checkers for virtual channels and broadcast data (Fig. 3.2). Furthermore, there is a trigger module (Fig. 3.3) that decodes the SpFi data stream which can be analysed using the Word or the Frame based view (described in the next section).


STAR Fire Configurator

Unit

Unit selected: Unit A

SW version: 3.3
HW version: 0:0:0:0
HW revision: 0:0
HW build: 0:0:0:0

Status: No connection

SpaceFibre Ports

Port selected: Port 1, Port 2

Lane Status: Checking...

Error injection: Disconections, Bit Error Rate (BER) 10 - 10

Broadcasts Generator

Period (words): 0

Link bandwith Allowance: 100%

Broadcast Checker

Errors = None

Modes: Interface (selected), Sniffer

Configuration 1

Virtual Channels

VC selected: VC 2

Quality Of Service

Priority: Highest

Valid TimeSlots: 0x 0

Link bandwith Allowance: 0%

overflow/underflow

Data Generator

Period (words): 0

Packet size (words): 0

Data rate 0%

Path Address

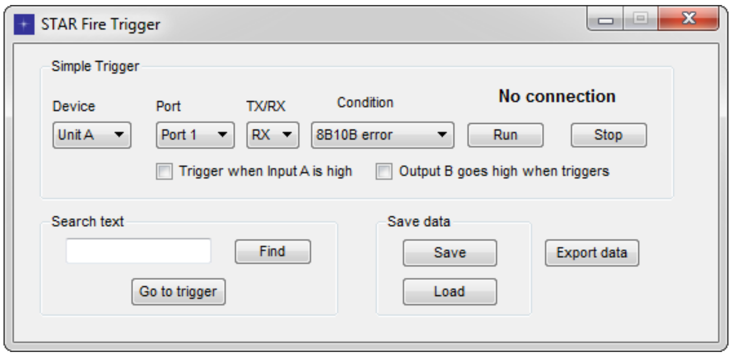
Data checker

Stop Checker (checked)

EEPs rcv: 0

Errors = None

Fig. 3.2: The STAR Fire software *Configuration* window


STAR Fire Trigger

Simple Trigger

Device: Unit A

Port: Port 1

TX/RX: RX

Condition: 8B10B error

No connection

Trigger when Input A is high

Output B goes high when triggers

Search text

Save data

Fig. 3.3: The STAR Fire software *Trigger* window

A SpFi link typically runs at 2.5 Gbit/s in RTAX FPGAs (using an external SerDes device like TLK2711-SP) and 3.125 Gbit/s in RTG4 or Virtex-5QV. SpFi also supports lane aggregation thanks to the multi-laning capabilities. This means that the link speed can be multiplied by using several lanes – physical connections – to form a link. However, this requires higher system complexity, and more mass (additional cables) and power consumption (to send the signals over the cables). An option to reduce this complexity, mass and power consumption is to compress the data prior to the transmission over the link. Thus, the new RTL version of FAPEC can be used to mitigate these problems.

3.2 Implementation of FAPEC inside STAR Fire Design

The original design of the STAR Fire unit has been modified to integrate FAPEC. The unit features 2 SpFi ports of 8 VCs each. The lowest VCs (VC 0 and VC 1) are connected to the SpW Router over a SpW to SpFi data format converter. The remaining VCs (VC 2 to VC 7) are connected to independent data pattern generators and checkers. The reason for this embedded generator/checker is that the unit is connected to a host computer over a USB 2.0 connection. The theoretical maximum data rate that can be achieved over USB 2.0 is 480 Mbit/s (the actual rate is well below this figure). This means that it is impossible to stress a single SpFi link at its maximum capacity of 2 Gbit/s (2.5 Gbit/s line rate) over a PC connection. The internal data generator and checkers allow to generate and check data at the maximum rate supported by SpFi. They can work independently on different VCs and on any of the SpFi ports. This allows testing the QoS offered by the SpFi codec. STAR-Dundee is working on a new version of the STAR Fire – called the STAR Fire Mk3 – that will feature a USB 3.0 connection, thus allowing much higher data rates to be directly sent from the PC.

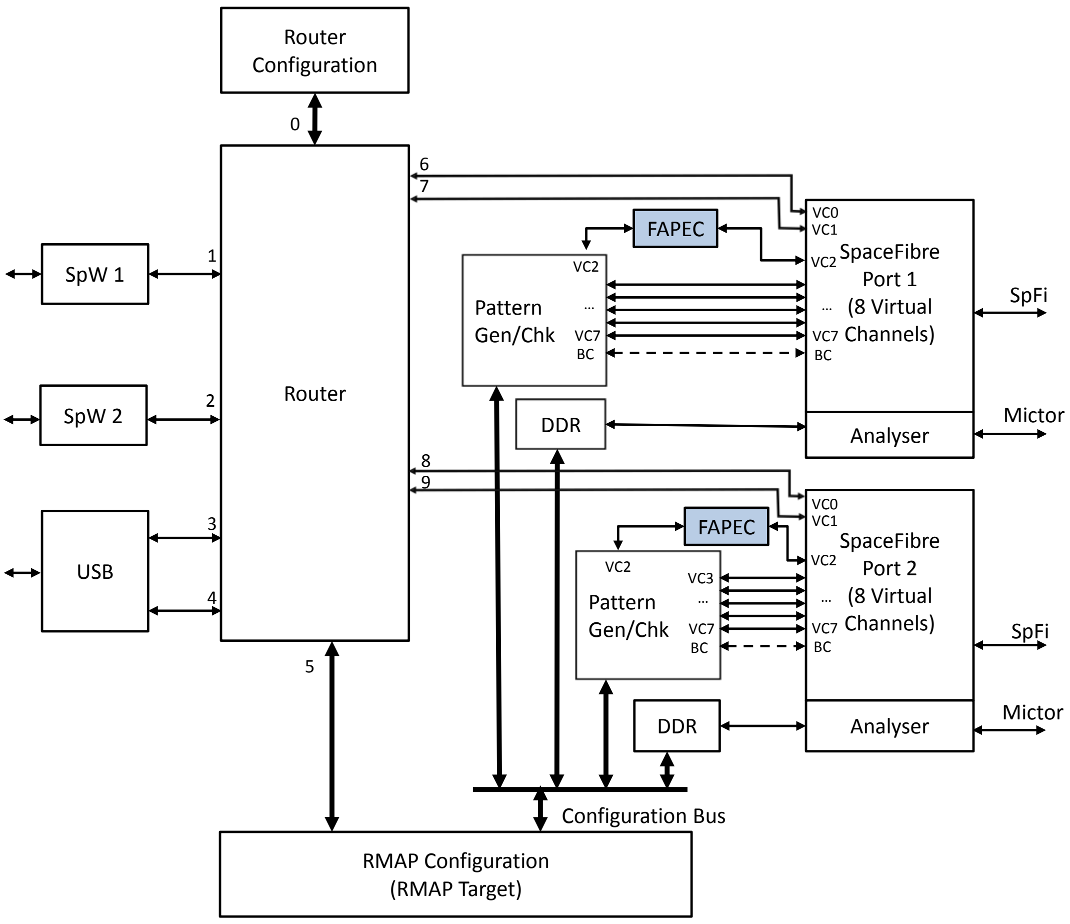
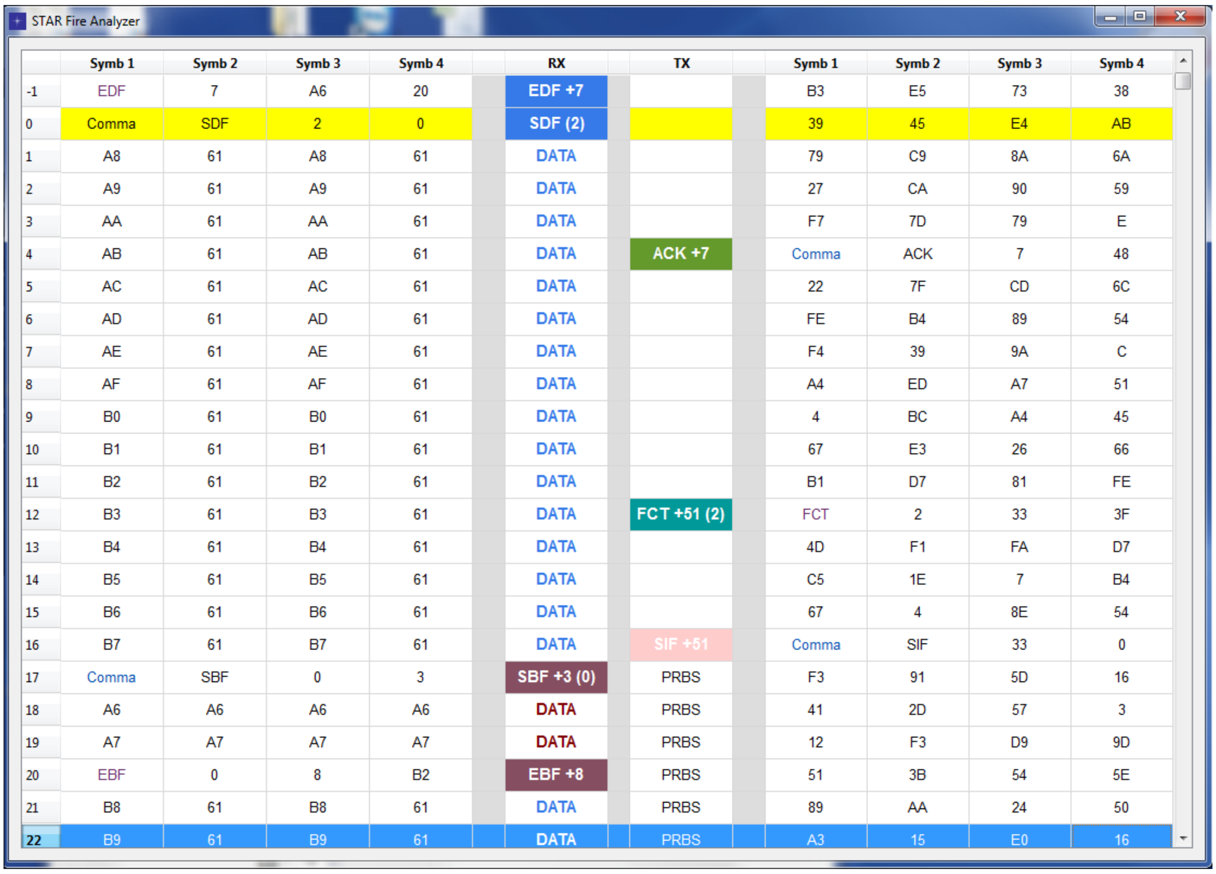


Fig. 3.4: The STAR Fire design architecture. In blue the FAPEC modules added to the design.

To optimise the development time and minimise risks, it was decided to insert the FAPEC codec between the data generator output and the transmit VC input, as shown in Fig. 3.4. To minimise resource usage only VC 2 has access to the FAPEC compressor. The original

STAR Fire design is the same as depicted in Fig. 3.4, except for the FAPEC modules (in blue). Originally VC 2 was directly connected to the data pattern and checker generators. This new design allows testing the FAPEC codec with the STAR Fire software *Configurator* window as shown in Fig. 3.2. The software is very powerful and apart from monitoring and controlling the status of the SpFi links, it also allows to trigger on different control words both in the transmission and reception sides in the *Trigger* window (Fig. 3.3). Upon triggering on a control word, a *Frame View* and *Analyser* windows appear. The Analyser window shows (Fig. 3.5) in the central part the SpFi words received (left half) and transmitted (right half) for the port selected in the trigger setup. Each word consists of four 8B10B symbols or characters that are shown at each side. The Frame View window shows data frames with a separate column for each VC (Fig. 3.6).

Regarding the code, there have been a couple of changes required by this design related to the interface between the data pattern generator and FAPEC. As explained in Section 2.3.2, the `PRECOMPRESSOR` module was updated to guarantee that the right value was read from the input data stream. The `data_generator` module updates the data value immediately after being read by FAPEC and the `PRECOMPRESSOR` needed to be updated accordingly.


STAR Fire Analyzer

	Symb 1	Symb 2	Symb 3	Symb 4	RX	TX	Symb 1	Symb 2	Symb 3	Symb 4
-1	EDF	7	A6	20	EDF +7		B3	E5	73	38
0	Comma	SDF	2	0	SDF (2)		39	45	E4	AB
1	A8	61	A8	61	DATA		79	C9	8A	6A
2	A9	61	A9	61	DATA		27	CA	90	59
3	AA	61	AA	61	DATA		F7	7D	79	E
4	AB	61	AB	61	DATA	ACK +7	Comma	ACK	7	48
5	AC	61	AC	61	DATA		22	7F	CD	6C
6	AD	61	AD	61	DATA		FE	B4	89	54
7	AE	61	AE	61	DATA		F4	39	9A	C
8	AF	61	AF	61	DATA		A4	ED	A7	51
9	B0	61	B0	61	DATA		4	BC	A4	45
10	B1	61	B1	61	DATA		67	E3	26	66
11	B2	61	B2	61	DATA		B1	D7	81	FE
12	B3	61	B3	61	DATA	FCT +51 (2)	FCT	2	33	3F
13	B4	61	B4	61	DATA		4D	F1	FA	D7
14	B5	61	B5	61	DATA		C5	1E	7	B4
15	B6	61	B6	61	DATA		67	4	8E	54
16	B7	61	B7	61	DATA	SIF +51	Comma	SIF	33	0
17	Comma	SBF	0	3	SBF +3 (0)	PRBS	F3	91	5D	16
18	A6	A6	A6	A6	DATA	PRBS	41	2D	57	3
19	A7	A7	A7	A7	DATA	PRBS	12	F3	D9	9D
20	EBF	0	8	B2	EBF +8	PRBS	51	3B	54	5E
21	B8	61	B8	61	DATA	PRBS	89	AA	24	50
22	B9	61	B9	61	DATA	PRBS	A3	15	E0	16

Fig. 3.5: The STAR Fire *Analyser* window

The `data_generator` module used by the STAR Fire design was also updated to meet the restrictions of timing between consecutive values of the `HIST_CONSTRUCTOR` module. Specifically, it had to be modified to guarantee that regardless of the configuration set up by the control software, the maximum data generation rate never exceeded 1 data value every 6 clock cycles. Section 2.3.3 explains how the new FAPEC implementation still presents some limitations that prevent the module to work on a sample-per-clock basis.

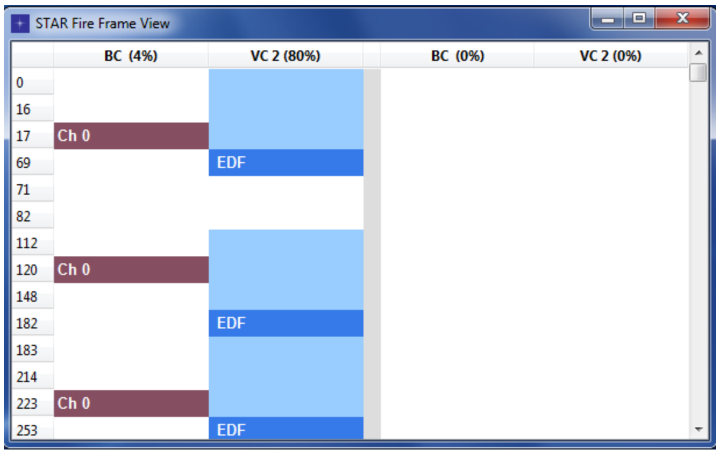
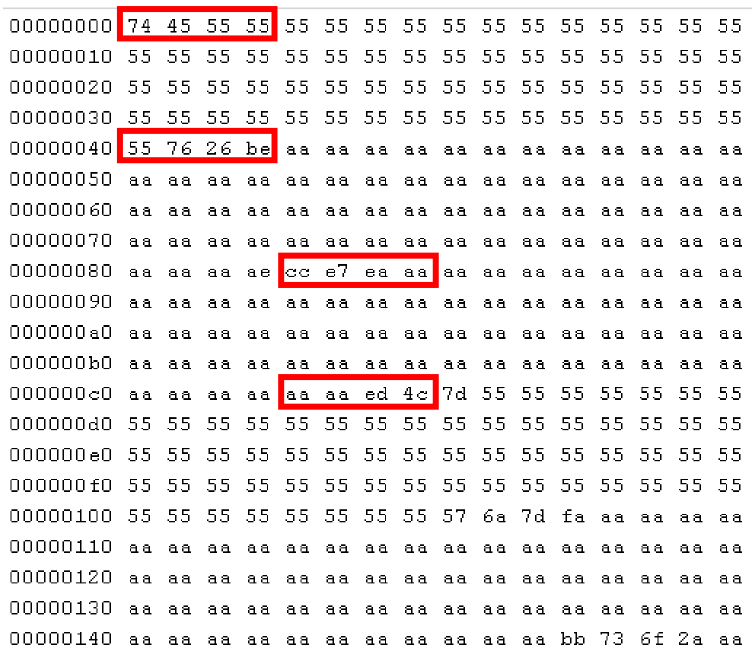


Fig. 3.6: The STAR Fire *Frame View* window

3.2.1 Verifying the Design

The data generator embedded in the STAR Fire is a very simple one. It consists of a simple increasing pattern, incrementing its output value by one every clock cycle. This means that differential values will always be ‘1’ with the exception of the leading value of each data block. This is not very representative of typical data, but bear in mind that this same algorithm has been successfully tested in simulation against different types of data sets. The purpose of the hardware set-up was to validate the operation of FAPEC with SpFi.



```
00000000 74 45 55 55 55 55 55 55 55 55 55 55 55 55 55 55
00000010 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
00000020 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
00000030 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
00000040 55 76 26 be aa aa aa aa aa aa aa aa aa aa aa aa
00000050 aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
00000060 aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
00000070 aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
00000080 aa aa aa ae cc e7 ea aa aa aa aa aa aa aa aa aa
00000090 aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
000000a0 aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
000000b0 aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
000000c0 aa aa aa aa aa aa ed 4c 7d 55 55 55 55 55 55 55
000000d0 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
000000e0 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
000000f0 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
00000100 55 55 55 55 55 55 55 55 57 6a 7d fa aa aa aa aa
00000110 aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
00000120 aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
00000130 aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
00000140 aa aa aa aa aa aa aa aa aa aa aa bb 73 6f 2a aa
```

Fig. 3.7: View of the compressed output by the data pattern generator

The pattern generated by the `data_generator` was printed into a file. Then this file was passed to the reference FAPEC design to obtain the FAPEC compressed master file. This is the file used to validate the operation of the STAR Fire design. Fig. 3.7 shows the compressed output of the master file. The red rectangles indicate the places where there is a boundary between packets. In the boundary the coding table is coded and these bit sequences can be easily identified.

The verification of the design has again been carried out with *Modelsim*. The whole STAR Fire design has been simulated and the output has been inspected to verify that the compressed file is being correctly generated by FAPEC. Once this has been verified, the rest of the design verification is simple because the initial STAR Fire design was already working correctly.

Fig. 3.8 shows the Modelsim simulation for the STAR Fire design. In the top panel the first two words are output (`0x55554574` and `0x55555555`). Note that they are only valid if `Out_Valid` signal is asserted. In the bottom panel the beginning of the second packet is displayed (`0xBE267655` and `0xAAAAAAAA`). The simulator output was binary compared against the compressed master file (Fig. 3.7) to verify that the output matched. Note the inverse endianness between the simulation and the file.

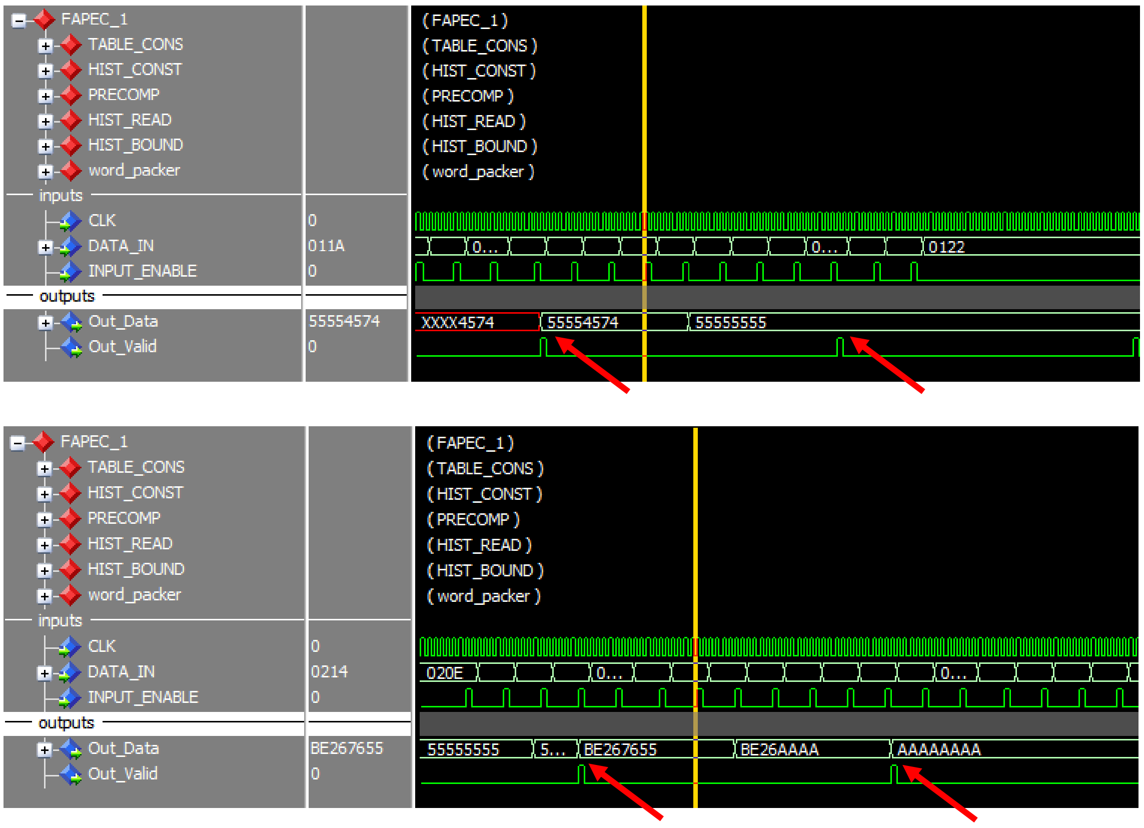


Fig. 3.8: Modelsim simulations of the STAR Fire design

3.2.2 Building the design

The original STAR Fire design took most of the FPGA resources. One problem encountered with the new design integrating the FAPEC compressor is that this new design was setting the FPGA tools at the limit of what is possible with the target device (Spartan-6 75T). The new parallel PEC module uses few registers but the number of multiplexers that uses in comparison is high. So the tool (Xilinx XST) presented several problems when trying to place the whole design inside the FPGA. In some runs (in 3 out of 4 tries) it got stuck for several hours before crashing, indicating that it was impossible to place the design. In the fourth try the design was successfully placed, but then in the routing phase there were plenty of timing errors. The device utilisation and logic distribution summary for that run was:

```
Slice Logic Utilization:
  Number of Slice Registers:                24,884 out of  93,296   26%
  Number of Slice LUTs:                     37,855 out of  46,648   81%

Slice Logic Distribution:
  Number of occupied Slices:                11,218 out of  11,662   96%
  Number of MUXCYs used:                     6,772 out of  23,324   29%
  Number of LUT Flip Flop pairs used:       39,715
    Number with an unused Flip Flop:        15,323 out of  39,715   38%
    Number with an unused LUT:               1,860 out of  39,715    4%
    Number of fully used LUT-FF pairs:      22,532 out of  39,715   56%
    Number of slice register sites lost
      to control set restrictions:               0 out of  93,296    0%
```

Whereas the original STAR Fire design without FAPEC required:

```
Slice Logic Utilization:
  Number of Slice Registers:                23,394 out of  93,296   25%
  Number of Slice LUTs:                     30,404 out of  46,648   65%

Slice Logic Distribution:
  Number of occupied Slices:                10,657 out of  11,662   91%
  Number of MUXCYs used:                     6,460 out of  23,324   27%
  Number of LUT Flip Flop pairs used:       34,014
    Number with an unused Flip Flop:        11,118 out of  34,014   32%
    Number with an unused LUT:               3,610 out of  34,014   10%
    Number of fully used LUT-FF pairs:      19,286 out of  34,014   56%
    Number of slice register sites lost
      to control set restrictions:               0 out of  93,296    0%
```

The design with FAPEC is congested, with 96% of occupied slices. If we compare the difference obtained while building these two designs, we can see that it is:

$$\Delta \text{ Registers} = 1490$$
$$\Delta \text{ LUTs} = 7451$$

In Section 2.5.5 the resources required for the FAPEC codec have been presented. A FAPEC module requires roughly 750 registers and 3750 LUTs. As there are two modules in this new design, ~1500 additional registers and ~7500 additional LUTs are needed. This matches almost perfectly the observed difference.

With a 96% of occupied slices, the FPGA is so congested by the addition of FAPEC that it is not possible to find suitable routes that meet timing for all the nets. Note that during the synthesis phase timing issues were not reported and constraints were easily met. This is a clear indication that the problem is not intrinsic of the design itself but related to the routing phase inside the FPGA. The solution to correctly place and route the design is to reduce complexity. An easy way to avoid changing large parts of the design is to remove the data generators and checkers connected to VCs 4 to 7. This change is easy to implement and the results obtained are satisfactory. The aggregate design requirements for LUTs descend from 81% to 74% and Registers usage goes from 26% down to 23%:

```
Slice Logic Utilization:
  Number of Slice Registers:              21,705 out of  93,296   23%
  Number of Slice LUTs:                   34,739 out of  46,648   74%

Slice Logic Distribution:
  Number of occupied Slices:              10,943 out of  11,662   93%
  Number of MUXCYs used:                   5,844 out of  23,324   25%
  Number of LUT Flip Flop pairs used:     36,909
    Number with an unused Flip Flop:      15,548 out of  36,909   42%
    Number with an unused LUT:             2,170 out of  36,909    5%
    Number of fully used LUT-FF pairs:    19,191 out of  36,909   51%
    Number of slice register sites lost
      to control set restrictions:             0 out of  93,296    0%
```

The most important parameter here, the number of occupied slices has not been altered that much: it simply has gone from 96% to 93%. But this 3% decrease makes all the difference, because congestion effects are not linear. Routing is a problem with exponential complexity, so small reductions in complexity can provide large gains in routing time. Routing issues become patent when approaching device saturation and then quickly disappear when resources are freed, especially when timing constraints are not very tight.

After this change, only one timing error is reported. As a matter of fact, this is not truly an error because it corresponds to the maximum skew allowed for the SpW clock recovery networks. Up to 1 ns is tolerated, but the constraint is set to 0.5 ns to force the tool to minimise this skew. The last clock in this timing report corresponds to the system clock used by FAPEC and most of the SpFi related logic.

```
--------------------------------------------------------------------------------------------
  Constraint                                |    Check     | Worst Case |  Best Case | Timing |   Timing
                                            |              |    Slack   | Achievable | Errors |    Score
--------------------------------------------------------------------------------------------
* NET "i_din(0)" MAXSKEW = 0.5 ns           | NETSKEW      |   -0.026ns|     0.526ns|       1|          26
--------------------------------------------------------------------------------------------
  NET "i_din(1)" MAXSKEW = 0.5 ns           | NETSKEW      |    0.023ns|     0.477ns|       0|           0
--------------------------------------------------------------------------------------------
  TS_txusrclk_sys = PERIOD TIMEGRP "TNM_txu | SETUP        |    0.158ns|    15.842ns|       0|           0
  srclk_sys" 16 ns HIGH 50%                 | HOLD         |    0.124ns|            |       0|           0
--------------------------------------------------------------------------------------------
```

The slack value indicates that timing is met for a worst case scenario – that is operating at the maximum of the temperature range, 85º Celsius, and at the minimum voltage of the range, 1.14 V. The frequency required for 2.5 Gbit/s link operation is 62.5 MHz for this clock (i.e. 16 ns of period). Worst path has a delay of 15.842 ns, which gives 0.158 ns of margin. This indicates that timing closure has been achieved and that no set-up violations will occur even in the worst case scenario.

3.3 Validation of the design

The hardware build procedure generates an *.mcs* file that can then be programmed in the target FPGA. The STAR Fire unit was reprogrammed with this new design featuring SpFi and FAPEC. VC 3 (not using FAPEC) was tested first to check that data frames were passing fine over the SpFi link. The trigger was configured to trigger on SDF (Start of Data Frame) control word. In this way the next data frame being sent over the link is captured. After setting the trigger, VC 2 data generation (using FAPEC) was enabled. Fig. 3.9 shows the STAR Fire unit sending data from SpFi Port 1 to Port 2.

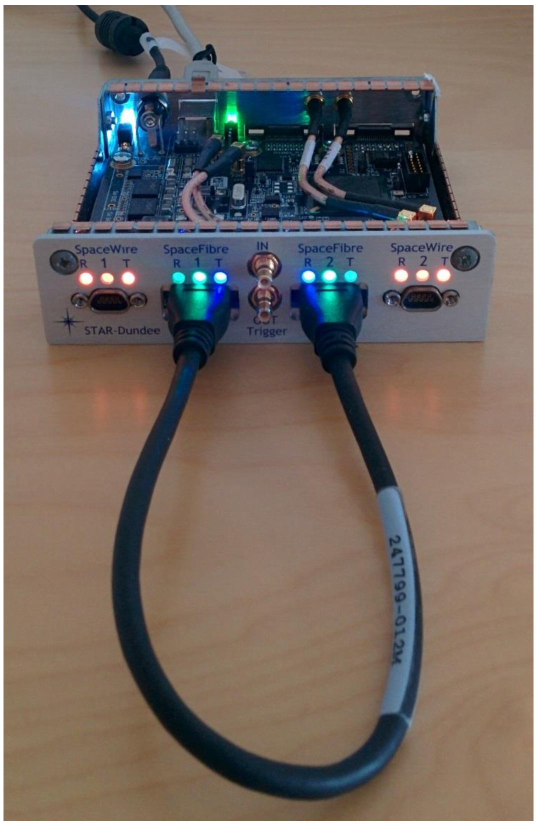
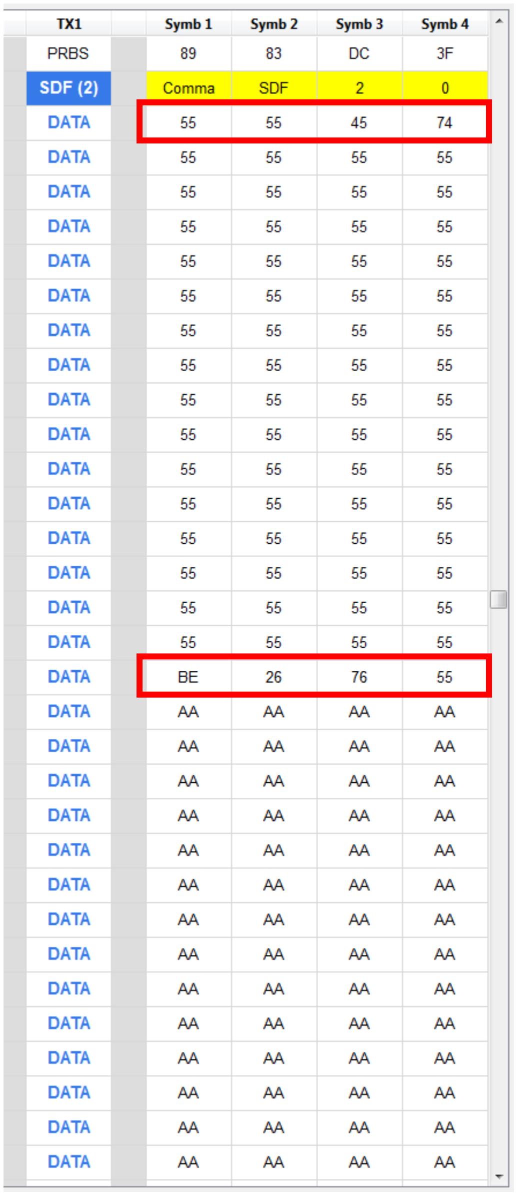


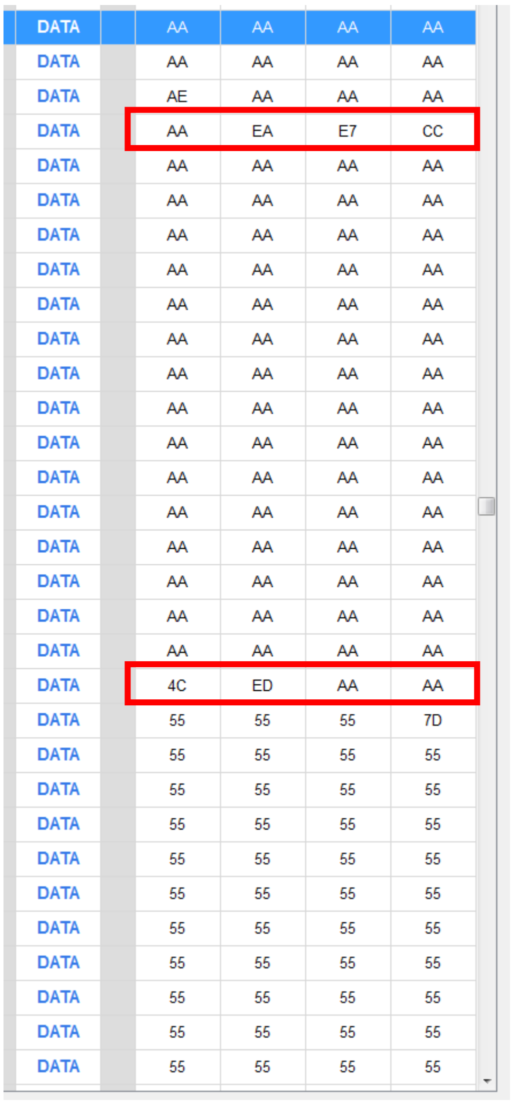
Fig. 3.9: STAR Fire unit in operation

Upon starting the data generation in VC 2, the STAR Fire software immediately triggers. The contents of the data frame can be inspected in the Word Analyser window. Fig. 3.10 shows the initial frame being sent over SpFi as captured by the trigger. On the left panel the initial data words travelling over the SpFi link are displayed. The word highlighted in yellow is the SDF, the word that caused the trigger event. The word above (PRBS) is simply a Pseudo-Random Binary Sequence that is sent over the link when there is nothing to be sent. After the SDF it can be seen as data exactly matches the compressed master file. Note that endianness between the master file and the data frames is reversed. This has to do with the way in which SpFi sends data produced by FAPEC. The right panel of the Fig. 3.10 shows the start of the second compressed block.

In summary, once the congestion issue was solved (Section 3.2.2) no problems were encountered during the validation phase. The verification performed with simulations showed that the design was working fine. When testing the hardware device the results matched the simulations as expected.



TX1	Symb 1	Symb 2	Symb 3	Symb 4
PRBS	89	83	DC	3F
SDF (2)	Comma	SDF	2	0
DATA	55	55	45	74
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	BE	26	76	55
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA



DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AE	AA	AA	AA
DATA	AA	EA	E7	CC
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	AA	AA	AA	AA
DATA	4C	ED	AA	AA
DATA	55	55	55	7D
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55
DATA	55	55	55	55

Fig. 3.10: Data frame carrying the compressed data packet captured by the Analyser

4. Conclusions

In this project a new register-transfer level (RTL) implementation of the FAPEC compressor has been developed. This implementation offers a greater throughput than the previous version, while maintaining a relatively small footprint. The main weaknesses of the initial prototype have been addressed. The initial implementation of FAPEC had been developed in the past to demonstrate the feasibility of an FPGA implementation, and offered a throughput of 32 Mbit/s. The new VHDL generated can target any FPGA technology, and its serial output has been substituted by a 32-bit parallel interface. This allows a much higher throughput, as the parallel interface can be easily used to interface a SerDes device or an AXI-like bus to send data at high speeds to other applications. Specifically, the speed of the algorithm has been improved by a factor 6 while the resource usage remains low, around 2% of a Virtex-5QV or an RTG4.

SpaceFibre (SpFi) is a new technology for use onboard spacecraft that provides point-to-point and networked interconnections at Gigabit rates. SpFi is an ESA initiative and will substitute the ubiquitous SpaceWire for high speed applications in space. In this work we have demonstrated that FAPEC can be easily integrated on top of SpFi to reduce the amount of information that the spacecraft network has to deal with. The integration of FAPEC with SpFi has successfully been validated in a representative FPGA platform. In this design FAPEC operated at ~12 Msamples/s (~200 Mbit/s) using a Xilinx Spartan-6 but it is expected to reach Gbit/s speeds with some additional work. This can increase the effective bandwidth of a single lane SpFi link well over the original 2.5/3.125 Gbit/s currently achieved with space-qualified technology, typically enabling effective throughputs of > 5 Gbit/s for common high-speed applications (e.g. instrument data). The combination of these two technologies can help to reduce the large amounts of data generated by some instruments in a transparent way, without the need of user intervention, and to provide a solution to the increasing data volumes in spacecrafts. Consequently the combination of FAPEC with SpFi can help to save mass, power consumption and reduce system complexity.

4.1 Forthcoming work

In the near future FAPEC is expected to be able to achieve more than 65 Msample/s (~ 1 Gbit/s) capability with some additional effort. Resource usage inside the FPGA is also expected to be slightly reduced with the adoption of optimised strategies to deal with the data compression. Specifically, the following tasks need to be done to improve FAPEC performance:

- Remove the pipeline stage inside the memory module:
  Depending on the technology it is better to have a register stage outside the memory, if required. In double-port or dual-port memories the operation for both ports must be symmetric.

- Remove falling edge logic:
  This is not normally used unless justified in some parts of the code. Sometimes used when changing data between clock domains to guarantee that data is sampled around the central bit time. This is not the case for FAPEC.

- Optimise the histogram generation logic:
  One sample value per clock should be processed. This would allow FAPEC to process one sample per clock cycle.

- Improve the calculation of the coding tables:
  The coding tables should mirror as much as possible the software version. The compression ratios of the software version are better due to a more complex algorithm calculating the coding tables.

- Use a more complex data generator:
  It is required a more complex data source to further validate FAPEC in real hardware. One possibility is to connect FAPEC to VC 1 too, so it can be accessed through the SpW Router. In this way it would be possible to send data directly from the computer over USB, although this will be relatively low-speed (USB 2.0).

- Increase the pipelining in `PEC_CODER` and `WORD_PACKER` modules:
  These two modules perform large multiplexing operations. It seems possible to reduce the number of operations (i.e. to use less FPGA resources) and to improve timing by increasing the number of pipelining stages inside them.

5. Annex

The calibration algorithm for PEC used by FAPEC is patented. This algorithm is kept secret to protect the know-how that allows to exploit the benefits of PEC with a low-complexity fast adaptive calibration method. However the PEC codec itself is of public domain. In the following sections the VHDL code for the parallel-output PEC_CODER and the WORD_PACKER modules is shown. The PEC_CODER represents the most important modification done to the initial FAPEC VHDL code. The WORD_PACKER, on the other hand, is a newly developed module presenting the compressed data in 32-bit chunks. This module was not present in the initial FAPEC.

5.1 Parallel-Output PEC VHDL Code

```vhdl
--============================================================================--
--
-- Design Units   :
--
-- Entity         : pec_opt(rtl)
--
-- File           : pec_opt.vhd
--
--    Function:
--
--      - This module takes the compression table and samples to compress and
--        generates the compressed bit stream. It has a separate parallel
--        output for the different fields instead of the original serial output
--        used by 1st version of FAPEC@FPGA
--
--
-- Limitations    :
--
-- Dependencies   :
--
-- Author       = Alberto Gonzalez
--
-- Last update: 2016-09-08
--------------------------------------------------------------------------------


-- IEEE library includes
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.constants_definition_package.all;  -- the constants are defined here

--------------------------------------------------------------------------------
-- entity declaration.
--
entity pec_opt is

   port (
      Clk         : in std_logic;
      Reset       : in std_logic;
      Table_Valid : in std_logic;

      -- From encoder side
      Coding_Variant     : in std_logic_vector(1 downto 0);
      Segment_1_Num_Bits : in unsigned(LOG2_SSYZE-1 downto 0);
      Segment_2_Num_Bits : in unsigned(LOG2_SSYZE-1 downto 0);
      Segment_3_Num_Bits : in unsigned(LOG2_SSYZE-1 downto 0);
      Segment_4_Num_Bits : in unsigned(LOG2_SSYZE-1 downto 0);
      Ceiling_1_Val      : in unsigned(SYMBOL_SIZE-1 downto 0);
      Ceiling_2_Val      : in unsigned(SYMBOL_SIZE-1 downto 0);
      Ceiling_3_Val      : in unsigned(SYMBOL_SIZE-1 downto 0);

      -- To Word_Packer module
      Ready           : in  std_logic;
      Table_Valid_Out : out std_logic;
      Table_Num_Bits  : out unsigned(LOG2_SSYZE downto 0);
      Table_Vector    : out std_logic_vector(TAB_LONG_REF-1 downto 0);

      Comp_Sample_Valid  : out std_logic;
      Coding_Variant_Out : out std_logic_vector(1 downto 0);
      LE_Num_Bits        : out unsigned(4 downto 0);
      DS_Num_Bits        : out unsigned(4 downto 0);
      LC_Num_Bits        : out unsigned(4 downto 0);
      LE_Comp_Val        : out std_logic_vector(22 downto 0);
      DS_Comp_Val        : out std_logic_vector(22 downto 0);
      LC_Comp_Val        : out std_logic_vector(19 downto 0);

      -- Memories Management
      RD    : in  std_logic_vector(SYMBOL_SIZE downto 0);  -- Pixel/value + sign to compress as read
      from the block RAM
      RADDR : out std_logic_vector(LOG2_BSIZE downto 0);  -- Address to read of the block RAM (2 x
      block size)
      REN   : out std_logic  -- read enable for the RAM / LOW ACTIVE ???
      );

end entity pec_opt;


-- architecture
architecture rtl of pec_opt is

   constant TBSZ : positive := TAB_LONG_REF;

   ---[ signals ]------------------------------------------------------------
   type fsm_state is (S_IDLE, S_TABLE_CODING, S_WAIT_1, S_WAIT_2, S_OUTPUT_COMP_VAL);
   signal state_n, state_r : fsm_state;

   signal sign_n : std_logic;
   signal sign_r : std_logic;

   signal seg_1_n : unsigned(LOG2_SSYZE-1 downto 0);
   signal seg_1_r : unsigned(LOG2_SSYZE-1 downto 0);
   signal seg_2_n : unsigned(LOG2_SSYZE-1 downto 0);
   signal seg_2_r : unsigned(LOG2_SSYZE-1 downto 0);
   signal seg_3_n : unsigned(LOG2_SSYZE-1 downto 0);
   signal seg_3_r : unsigned(LOG2_SSYZE-1 downto 0);
   signal seg_4_n : unsigned(LOG2_SSYZE downto 0);
   signal seg_4_r : unsigned(LOG2_SSYZE downto 0);

   signal table_num_bits_n : unsigned(LOG2_SSYZE downto 0);
   signal table_num_bits_r : unsigned(LOG2_SSYZE downto 0);
   signal table_vector_n   : unsigned(TBSZ-1 downto 0);
   signal table_vector_r   : unsigned(TBSZ-1 downto 0);

   signal le_num_bits_n : unsigned(4 downto 0);
   signal le_num_bits_r : unsigned(4 downto 0);
   signal ds_num_bits_n : unsigned(4 downto 0);
   signal ds_num_bits_r : unsigned(4 downto 0);
   signal lc_num_bits_n : unsigned(4 downto 0);
   signal lc_num_bits_r : unsigned(4 downto 0);

   signal le_compressed_val_n : unsigned(22 downto 0);
   signal le_compressed_val_r : unsigned(22 downto 0);
   signal ds_compressed_val_n : unsigned(22 downto 0);
   signal ds_compressed_val_r : unsigned(22 downto 0);
   signal lc_compressed_val_n : unsigned(19 downto 0);
   signal lc_compressed_val_r : unsigned(19 downto 0);

   signal segment_n       : unsigned(1 downto 0);
   signal segment_r       : unsigned(1 downto 0);
   signal segment_value_n : unsigned(SYMBOL_SIZE-1 downto 0);
   signal segment_value_r : unsigned(SYMBOL_SIZE-1 downto 0);

   signal table_valid_out_n   : std_logic;
   signal comp_sample_valid_n : std_logic;

   signal coding_variant_r : std_logic_vector(1 downto 0);

   signal word_count_n : unsigned(LOG2_BSIZE downto 0);
   signal word_count_r : unsigned(LOG2_BSIZE downto 0);

begin
```

```
-- Alias
sign_n <= RD(SYMBOL_SIZE);           -- Highest bit codes the sign

-- Segment 4 can be up to 16 bits. Coded with 4 bits, this is represented by
-- "0000"
seg_4_n <= "10000" when (Segment_4_Num_Bits = 0) else ('0' & Segment_4_Num_Bits);
seg_3_n <= Segment_3_Num_Bits;
seg_2_n <= Segment_2_Num_Bits;
seg_1_n <= Segment_1_Num_Bits;


-----------------------------------------------------------------------------
-- FSM controlling the encoding of a data block
--
encod_fsm : process (all) is
begin
   -- Default is to hold state
   state_n              <= state_r;
   table_valid_out_n    <= '0';
   comp_sample_valid_n <= '0';
   word_count_n         <= word_count_r;

   -- next state is dependent on current state
   case (state_r) is

      when S_IDLE =>
         if (Table_Valid = '1') then
            state_n <= S_TABLE_CODING;
         end if;

      when S_TABLE_CODING =>
         state_n           <= S_WAIT_1;
         table_valid_out_n <= '1';

      when S_WAIT_1 =>
         state_n <= S_WAIT_2;

      when S_WAIT_2 =>
         state_n <= S_OUTPUT_COMP_VAL;

      when S_OUTPUT_COMP_VAL =>
         if (Ready = '1') then
            -- Only if module getting this data is Ready to accept it
            comp_sample_valid_n <= '1';

            if (word_count_r = to_unsigned(509, LOG2_BSIZE+1)) then
            -- End of the memory. Reset the counter counter init
            word_count_n <= (others => '0');
         else
            word_count_n <= word_count_r + 1;
         end if;

            if (word_count_r = to_unsigned(509, LOG2_BSIZE+1)) or
               (word_count_r = to_unsigned(254, LOG2_BSIZE+1)) then
               state_n <= S_IDLE;
            else
               state_n <= S_WAIT_1;
            end if;
         end if;

   end case;
end process encod_fsm;


-----------------------------------------------------------------------------
-- Initialisation table calculation
--
table_const : process (all) is
begin
   -- Default
   table_vector_n <= (others => '0');

   if (Coding_Variant(1) = '0') then
      -- LE variant
      table_num_bits_n                                <= to_unsigned(LE_TAB_LONG,
   LOG2_SSYZE+1);  -- 10 bits
```

```
      table_vector_n(TBSZ-1 downto TBSZ-2)                <= "01";  -- flag
      table_vector_n(TBSZ-3)                              <= Segment_1_Num_Bits(0);
      table_vector_n(TBSZ-4)                              <= Segment_2_Num_Bits(0);
      table_vector_n(TBSZ-5 downto TBSZ-6)                <= Segment_3_Num_Bits(1 downto 0);
      table_vector_n(TBSZ-7 downto TBSZ -LOG2_SSYZE -6) <= Segment_4_Num_Bits;

   elsif (Coding_Variant(0) = '0') then
      -- DS variant
      table_num_bits_n                                     <= to_unsigned(DS_TAB_LONG,
   LOG2_SSYZE+1);  -- 13 bits
      table_vector_n(TBSZ-1 downto TBSZ-2)                 <= "00";  -- flag
      table_vector_n(TBSZ-3 downto TBSZ-4)                 <= Segment_1_Num_Bits(1 downto 0);
      table_vector_n(TBSZ-5 downto TBSZ-6)                 <= Segment_2_Num_Bits(1 downto 0);
      table_vector_n(TBSZ-7 downto TBSZ-9)                 <= Segment_3_Num_Bits(2 downto 0);
      table_vector_n(TBSZ-10 downto TBSZ -LOG2_SSYZE -9) <= Segment_4_Num_Bits;

   else
      -- LC variant
      table_num_bits_n                                                      <=
   to_unsigned(LC_TAB_LONG, LOG2_SSYZE+1);  -- 13 bits
      table_vector_n(TBSZ-1)                                                <= '1';  -- flag
      table_vector_n(TBSZ-2 downto TBSZ -LOG2_SSYZE -1)                    <= Segment_1_Num_Bits;
      table_vector_n(TBSZ -LOG2_SSYZE -2 downto TBSZ -2*LOG2_SSYZE -1)    <= Segment_2_Num_Bits;
      table_vector_n(TBSZ -2*LOG2_SSYZE -2 downto TBSZ -3*LOG2_SSYZE -1) <= Segment_3_Num_Bits;
      table_vector_n(TBSZ -3*LOG2_SSYZE -2 downto TBSZ -4*LOG2_SSYZE -1) <= Segment_4_Num_Bits;
   end if;

end process table_const;



----------------------------------------------------------------------------
-- Segment calculation
--
segment_calc : process (all) is
   variable abs_value : unsigned(SYMBOL_SIZE-1 downto 0);
begin
   -- Default (segment 0)
   abs_value       := unsigned(RD(SYMBOL_SIZE-1 downto 0));
   segment_n       <= "00";
   segment_value_n <= abs_value;

   if (abs_value > Ceiling_3_Val) then
      segment_value_n <= abs_value - Ceiling_3_Val - 1;
      segment_n       <= "11";

   elsif (abs_value > Ceiling_2_Val) then
      segment_value_n <= abs_value - Ceiling_2_Val - 1;
      segment_n       <= "10";

   elsif (abs_value > Ceiling_1_Val) then
      segment_value_n <= abs_value - Ceiling_1_Val - 1;
      segment_n       <= "01";

   end if;
end process segment_calc;


----------------------------------------------------------------------------
-- Assign the output bit sequence depending on the selected variant and the
-- coding table
--
bit_assign : process (all) is
   variable s_1        : integer;
   variable s_2        : integer;
   variable s_3        : integer;
   variable s_4        : integer;
   variable v_num_bits : integer;
   variable abs_value  : unsigned(SYMBOL_SIZE-1 downto 0);
begin
   -- Default
   le_compressed_val_n <= (others => '0');
   ds_compressed_val_n <= (others => '0');
   lc_compressed_val_n <= (others => '0');
   le_num_bits_n       <= (others => '0');
   ds_num_bits_n       <= (others => '0');
   lc_num_bits_n       <= (others => '0');
```

```
v_num_bits          := 0;
--
abs_value           := segment_value_r;
s_1                 := to_integer(seg_1_r);
s_2                 := to_integer(seg_2_r);
s_3                 := to_integer(seg_3_r);
s_4                 := to_integer(seg_4_r);

case (coding_variant_r) is

   ----------------------------------------------------------------------
   when LE_VAR =>
      if (segment_r = 0) then
         le_compressed_val_n(0)           <= sign_r;
         le_compressed_val_n(s_1 downto 1) <= abs_value(s_1 -1 downto 0);
         v_num_bits                        := s_1 + 1;
      end if;

      if (segment_r = 1) then
         le_compressed_val_n(0)                       <= '1';
         le_compressed_val_n(s_1 downto 1)            <= (others => '0');
         le_compressed_val_n(s_1 + 1)                 <= sign_r;
         le_compressed_val_n(s_1 +1 +s_2 downto s_1 +2) <= abs_value(s_2 -1 downto 0);
         v_num_bits                                   := 1 + s_1 + 1 + s_2;
      end if;

      if (segment_r > 1) then
         -- For both 3rd and 4th segments
         le_compressed_val_n(0)                               <= '1';
         le_compressed_val_n(s_1 downto 1)                    <= (others => '0');
         le_compressed_val_n(s_1 + 1)                         <= sign_r;
         le_compressed_val_n(s_1 +1 +s_2 downto s_1 +2)       <= (others => '1');
         -- '0' for 3rd segment
         -- '1' for 4th segment
         le_compressed_val_n(s_1 +2 +s_2)                     <= segment_r(0);

         if (segment_r = 2) then
            -- This is required because s_3 can be greater than s_4
            -- E.g. s_3 = 2 and s_4 = 1
            -- This adds a little bit more complexity
            le_compressed_val_n(s_1 +2 +s_2 +s_3 downto s_1 +3 +s_2) <= abs_value(s_3 -1
downto 0);
         else
            le_compressed_val_n(s_1 +2 +s_2 +s_4 downto s_1 +3 +s_2) <= abs_value(s_4 -1
downto 0);
         end if;
      end if;

      if (segment_r = 2) then
         v_num_bits := 1 + s_1 + 1 + s_2 + 1 + s_3;
      elsif (segment_r = 3) then
         v_num_bits := 1 + s_1 + 1 + s_2 + 1 + s_4;
      end if;

      le_num_bits_n <= to_unsigned(v_num_bits, 5);

   ----------------------------------------------------------------------
   when DS_VAR =>
      if (segment_r = 0) then
         ds_compressed_val_n(0)           <= sign_r;
         ds_compressed_val_n(s_1 downto 1) <= abs_value(s_1 -1 downto 0);
         v_num_bits                        := 1 + s_1;
      end if;

      if (segment_r = 1) then
         ds_compressed_val_n(0)                      <= sign_r;
         ds_compressed_val_n(s_1 downto 1)           <= (others => '1');
         ds_compressed_val_n(s_1 + s_2 downto s_1 +1) <= abs_value(s_2 -1 downto 0);
         v_num_bits                                  := 1 + s_1 + s_2;
      end if;

      if (segment_r > 1) then
         -- For both 3rd and 4th segments
         ds_compressed_val_n(0)                      <= '1';
         ds_compressed_val_n(s_1 downto 1)           <= (others => '0');
         ds_compressed_val_n(s_1 + 1)                <= sign_r;
         -- '0' for 3rd segment
```

```
            -- '1' for 4th segment
            ds_compressed_val_n(s_1 + 2)                      <= segment_r(0);

            if (segment_r = 2) then
               -- This is required because s_3 can be greater than s_4
               -- E.g. s_3 = 2 and s_4 = 1
               -- This adds a little bit more complexity
               ds_compressed_val_n(s_1 + 2 + s_3 downto s_1 + 3) <= abs_value(s_3 -1 downto 0);
            else
               ds_compressed_val_n(s_1 + 2 + s_4 downto s_1 + 3) <= abs_value(s_4 -1 downto 0);
            end if;

         end if;

         if (segment_r = 2) then
            v_num_bits := 1 + s_1 + 2 + s_3;
         elsif (segment_r = 3) then
            v_num_bits := 1 + s_1 + 2 + s_4;
         end if;

         ds_num_bits_n <= to_unsigned(v_num_bits, 5);

      -------------------------------------------------------------------
      when LC_VAR =>
         if (segment_r = 0) then
            lc_compressed_val_n(0)           <= '0';
            lc_compressed_val_n(s_1 downto 1) <= abs_value(s_1 -1 downto 0);
            lc_compressed_val_n(s_1 + 1)       <= sign_r;
            if (abs_value = 0) then
               v_num_bits := 1 + s_1;
            else
               v_num_bits := 2 + s_1;
            end if;
         end if;

         if (segment_r = 1) then
            lc_compressed_val_n(1 downto 0)        <= "01";
            lc_compressed_val_n(s_2 + 1 downto 2) <= abs_value(s_2 -1 downto 0);
            lc_compressed_val_n(s_2 + 2)           <= sign_r;
            v_num_bits                             := 3 + s_2;
         end if;

         if (segment_r = 2) then
            lc_compressed_val_n(2 downto 0)        <= "011";
            lc_compressed_val_n(s_3 + 2 downto 3) <= abs_value(s_3 -1 downto 0);
            lc_compressed_val_n(s_3 + 3)           <= sign_r;
            v_num_bits                             := 4 + s_3;
         end if;

         if (segment_r = 3) then
            lc_compressed_val_n(2 downto 0)        <= "111";
            lc_compressed_val_n(s_4 + 2 downto 3) <= abs_value(s_4 -1 downto 0);
            lc_compressed_val_n(s_4 + 3)           <= sign_r;
            v_num_bits                             := 4 + s_4;
         end if;

         lc_num_bits_n <= to_unsigned(v_num_bits, 5);


      when others =>
         null;

   end case;
end process bit_assign;


-----------------------------------------------------------------------------
--
control_path : process (all) is
begin
   if (rising_edge(Clk)) then
      if (Reset = '1') then

         state_r            <= S_IDLE;
         Table_Valid_Out    <= '0';
         Comp_Sample_Valid <= '0';
         word_count_r       <= (others => '0');
         else
            state_r            <= state_n;
            Table_Valid_Out    <= table_valid_out_n;
            Comp_Sample_Valid  <= comp_sample_valid_n;
            word_count_r       <= word_count_n;

         end if;
      end if;
   end process control_path;


   ------------------------------------------------------------------------
   --
   data_proc : process (Clk) is
   begin
      if (rising_edge(Clk)) then

            sign_r              <= sign_n;
            coding_variant_r    <= Coding_Variant;
            Coding_Variant_Out  <= coding_variant_r;

            seg_1_r <= seg_1_n;
            seg_2_r <= seg_2_n;
            seg_3_r <= seg_3_n;
            seg_4_r <= seg_4_n;

            segment_r       <= segment_n;
            segment_value_r <= segment_value_n;

            table_num_bits_r <= table_num_bits_n;
            table_vector_r   <= table_vector_n;

            le_num_bits_r <= le_num_bits_n;
            ds_num_bits_r <= ds_num_bits_n;
            lc_num_bits_r <= lc_num_bits_n;

            le_compressed_val_r <= le_compressed_val_n;
            ds_compressed_val_r <= ds_compressed_val_n;
            lc_compressed_val_r <= lc_compressed_val_n;

      end if;
   end process data_proc;


   -- Map Outputs

   -- Signal assignments for the memories
   REN   <= LOW;  -- low level active. We are always reading the memory, but only take the value
      when we are interested...
   RADDR <= std_logic_vector(word_count_r);  -- the addressed is always the value of the register

   Table_Num_Bits  <= table_num_bits_r;
   Table_Vector    <= reverse_bits(std_logic_vector(table_vector_r));

   LE_Num_Bits <= le_num_bits_r;
   DS_Num_Bits <= ds_num_bits_r;
   LC_Num_Bits <= lc_num_bits_r;

   LE_Comp_Val <= std_logic_vector(le_compressed_val_r);
   DS_Comp_Val <= std_logic_vector(ds_compressed_val_r);
   LC_Comp_Val <= std_logic_vector(lc_compressed_val_r);

end architecture rtl;
```

5.2 Word Packer VHDL Code

```vhdl
--=============================================================================--
--
-- Design Units   :
--
-- Entity         : word_packer(rtl)
--
-- File           : word_packer.vhd
--
--    Function:
--
--      - Takes the parallel-like compressed stream generated by the new PEC
--        coder and sets it into chunks of 32 bits. They can then be directly
--        interfaced to a VC buffer
--
-- Limitations    :
--
-- Dependencies   :
--
-- Author       = Alberto Gonzalez
--
-- Last update: 2016-09-08
-------------------------------------------------------------------------------


-- IEEE library includes
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.constants_definition_package.all;  -- the constants are defined here

-------------------------------------------------------------------------------
-- entity declaration.
--
entity word_packer is

   port (
      Clk   : in std_logic;
      Reset : in std_logic;

      -- From PEC compressor
      Ready           : out std_logic;
      Table_Valid_Out : in  std_logic;
      Table_Num_Bits  : in  unsigned(LOG2_SSYZE downto 0);
      Table_Vector    : in  std_logic_vector(TAB_LONG_REF-1 downto 0);

      Comp_Sample_Valid  : in std_logic;
      Coding_Variant_Out : in std_logic_vector(1 downto 0);
      LE_Num_Bits        : in unsigned(4 downto 0);
      DS_Num_Bits        : in unsigned(4 downto 0);
      LC_Num_Bits        : in unsigned(4 downto 0);
      LE_Comp_Val        : in std_logic_vector(22 downto 0);
      DS_Comp_Val        : in std_logic_vector(22 downto 0);
      LC_Comp_Val        : in std_logic_vector(19 downto 0);

      -- To VC buffer
      VCB_Half_Full : in  std_logic;
      Out_Valid     : out std_logic;
      Out_Data      : out std_logic_vector(31 downto 0)
      );

end entity word_packer;


-- architecture
architecture rtl of word_packer is

   ---[ signals ]-------------------------------------------------------------
   type fsm_state is (S_1ST_STAGE, S_2ND_STAGE, S_3RD_STAGE);
   signal state_n, state_r : fsm_state;

   signal in_value_n       : std_logic_vector(22 downto 0);
   signal in_value_r       : std_logic_vector(22 downto 0);
   signal num_bits_n       : unsigned(LOG2_SSYZE+1 downto 0);
```

```
    signal num_bits_r       : unsigned(LOG2_SSYZE+1 downto 0);
    signal remaining_bits_n : unsigned(LOG2_SSYZE downto 0);
    signal remaining_bits_r : unsigned(LOG2_SSYZE downto 0);

    signal write_half_word_n : std_logic;
    signal write_half_word_r : std_logic;
    signal half_word_valid_n : std_logic;
    signal half_word_valid_r : std_logic;
    signal half_word_n       : std_logic_vector(15 downto 0);
    signal half_word_r       : std_logic_vector(15 downto 0);
    -- Point to the location of the next bit to be filled up with incoming data
    -- in 'half_word' signal
    signal ptr_n             : unsigned(4 downto 0);
    signal ptr_r             : unsigned(4 downto 0);

    signal full_word_n          : std_logic_vector(31 downto 0);
    signal full_word_r          : std_logic_vector(31 downto 0);
    signal full_word_valid_n    : std_logic;
    signal full_word_valid_r    : std_logic;
    signal full_word_1st_half_n : std_logic;
    signal full_word_1st_half_r : std_logic;

begin

    write_half_word_n <= Table_Valid_Out or Comp_Sample_Valid;

    ----------------------------------------------------------------------------
    -- Places the next chunk of bits to be added to the bit stream in a
    -- register. Also, the length of the sequence is registered too
    --
    input_mux : process (all) is
    begin
       -- Default
       in_value_n <= in_value_r;
       num_bits_n <= num_bits_r;

       if (Table_Valid_Out = '1') then
          -- Conding Table to go out
          in_value_n(TAB_LONG_REF-1 downto 0) <= Table_Vector;
          num_bits_n                          <= '0' & Table_Num_Bits;

       elsif (Comp_Sample_Valid = '1') then

          if (Coding_Variant_Out = LE_VAR) then
             in_value_n <= LE_Comp_Val;
             num_bits_n <= '0' & LE_Num_Bits;

          elsif (Coding_Variant_Out = DS_VAR) then
             in_value_n <= DS_Comp_Val;
             num_bits_n <= '0' & DS_Num_Bits;

          else
             -- LC_VAR
             in_value_n(19 downto 0) <= LC_Comp_Val;
             num_bits_n              <= '0' & LC_Num_Bits;

          end if;
       end if;
    end process input_mux;


    ----------------------------------------------------------------------------
    --
    halfword_writing : process (all) is
    begin
       -- Default
       half_word_valid_n <= '0';
       state_n           <= state_r;
       ptr_n             <= ptr_r;

       remaining_bits_n <= remaining_bits_r;
       half_word_n      <= half_word_r;

       case (state_r) is

          when S_1ST_STAGE =>
             if (write_half_word_r = '1' and VCB_Half_Full = '0') then
```

```
            if (ptr_r + num_bits_r >= 16) then
               half_word_valid_n <= '1';
            end if;

            if (ptr_r + num_bits_r > 16) then
               state_n                                  <= S_2ND_STAGE;
               ptr_n                                    <= 16 - ptr_r;  -- Use as pointer
   for last bit of current value written
               remaining_bits_n                         <= num_bits_r(4 downto 0) - (16 -
   unsigned(ptr_r));
               half_word_n(15 downto to_integer(ptr_r)) <= in_value_r(to_integer(15 -
   ptr_r) downto 0);
            else
               state_n                                                                 <=
   S_1ST_STAGE;
               -- Use this unsigned(integer(), 4) cast formula to wrap-up on 15
               ptr_n                                                                   <=
   '0' & to_unsigned(to_integer(ptr_r + num_bits_r), 4);  -- Use as pointer for last bit of
   next 'half-word'
               remaining_bits_n                                                        <=
   (others => '0');
               half_word_n(to_integer(ptr_r + num_bits_r -1) downto to_integer(ptr_r)) <=
   in_value_r(to_integer(num_bits_r -1) downto 0);
            end if;
         end if;


      when S_2ND_STAGE =>
         if (remaining_bits_r >= 16) then
            half_word_valid_n <= '1';
         end if;

         if (remaining_bits_r > 16) then
            state_n          <= S_3RD_STAGE;
            ptr_n            <= ptr_r + 16;
            remaining_bits_n <= remaining_bits_r - 16;
            half_word_n      <= in_value_r(to_integer(15 + unsigned('0' & ptr_r)) downto
   to_integer(ptr_r));
         else
            state_n                                                <= S_1ST_STAGE;
            ptr_n                                                  <= '0' &
   remaining_bits_r(3 downto 0);
            remaining_bits_n                                       <= (others => '0');
            half_word_n(to_integer(remaining_bits_r - 1) downto 0) <=
   in_value_r(to_integer(remaining_bits_r + ptr_r - 1) downto to_integer(ptr_r));
         end if;


      when S_3RD_STAGE =>
         state_n                                                <= S_1ST_STAGE;
         ptr_n                                                  <= '0' & remaining_bits_r(3
   downto 0);
         remaining_bits_n                                       <= (others => '0');
         half_word_n(to_integer(remaining_bits_r - 1) downto 0) <=
   in_value_r(to_integer(remaining_bits_r + ptr_r - 1) downto to_integer(ptr_r));

   end case;
end process halfword_writing;


----------------------------------------------------------------------------
--
fullword_writing : process (all) is
begin
   -- Default
   full_word_1st_half_n <= full_word_1st_half_r;
   full_word_n          <= full_word_r;
   full_word_valid_n    <= '0';

   if (half_word_valid_r = '1') then
      if (full_word_1st_half_r = '0') then
         -- 1st half of the world
         full_word_1st_half_n     <= '1';
         full_word_n(15 downto 0) <= half_word_r;

      else
```

```
            -- 2nd half of the world
            -- Output the word
            full_word_valid_n          <= '1';
            full_word_1st_half_n       <= '0';
            full_word_n(31 downto 16)  <= half_word_r;

         end if;
      end if;

   end process fullword_writing;



   ----------------------------------------------------------------------------
   --
   control_path : process (all) is
   begin
      if (rising_edge(Clk)) then
         if (Reset = '1') then

            state_r              <= S_1ST_STAGE;
            ptr_r                <= (others => '0');
            num_bits_r           <= (others => '0');
            half_word_valid_r    <= '0';
            write_half_word_r    <= '0';
            full_word_valid_r    <= '0';
            full_word_1st_half_r <= '0';

         else
            state_r              <= state_n;
            ptr_r                <= ptr_n;
            num_bits_r           <= num_bits_n;
            half_word_valid_r    <= half_word_valid_n;
            write_half_word_r    <= write_half_word_n;
            full_word_valid_r    <= full_word_valid_n;
            full_word_1st_half_r <= full_word_1st_half_n;

         end if;
      end if;
   end process control_path;



   -------------------------------------------------------------------------
   --
   data_proc : process (Clk) is
   begin
      if (rising_edge(Clk)) then

         in_value_r       <= in_value_n;
         remaining_bits_r <= remaining_bits_n;
         half_word_r      <= half_word_n;
         full_word_r      <= full_word_n;

      end if;
   end process data_proc;

   -- Map Outputs
   Out_Valid <= full_word_valid_r;
   Out_Data  <= full_word_r;

   Ready <= '1' when (state_r = S_1ST_STAGE and state_n = S_1ST_STAGE) else '0';

end architecture rtl;
```

6. Bibliography

[1] J. Portell, A. G. Villafranca and E. García-Berro, "Designing optimum solutions for lossless data compression in space", in *Proceedings of the On-Board Payload Data Compression Workshop* 2008, pages 35-44. ESA, (2008).

[2] J. Portell, A.G. Villafranca and E. García-Berro, "Quick outlier-resilient entropy coder for space missions", Journal of Applied Remote Sensing 4 (2010).

[3] Consultative Committee for Space Data Systems, "Lossless Data Compression, Blue Book", CCSDS Tech. Rep., 121.0-B-1, CCSDS (1993).
http://ccsds.cosmos.ru/publications/archive/121x0b1c2.pdf
[Last accessed Sept 2016]

[4] J. Portell, A.G. Villafranca and E. García-Berro, "Outlier-resilient entropy coding", chapter in "Recent advances in satellite data compression", pages 87-113. Springer (2011)

[5] Space-grade Virtex-5QV FPGA
https://www.xilinx.com/products/silicon-devices/fpga/virtex-5qv.html
[Last accessed Sept 2016]

[6] Microsemi RTG4 FPGAs
http://www.microsemi.com/products/fpga-soc/radtolerant-fpgas/rtg4
[Last accessed Sept 2016]

[7] ESA Ongoing ASIC Developments
http://www.esa.int/Our_Activities/Space_Engineering_Technology/Microelectronics/Ongoing_ASIC_developments
[Last accessed Sept 2016]

[8] J. Portell, E. García-Berro, X. Luri, and A. G. Villafranca, "Tailored data compression using stream partitioning and prediction: application to Gaia", Experimental Astronomy 21, 125–149 (2006).

[9] D. Salomon, "Data Compression. The complete reference", Springer-Verlag (2004).

[10] A.G. Villafranca, S. Mignot, J. Portell and E. García-Berro, "Hardware implementation of FAPEC lossless data compressor for space", NASA/ESA Conference on Adaptive Hardware and Systems (2010).

[11] S. Parkes, A. Ferrer-Florit, A. González and C. McClements, "SpaceFibre Standard", Draft H4
https://indico.esa.int/indico/event/126/session/0/contribution/1
[Last accessed Sept 2016]

[12] A. Ginosar and P. Aviely, "RC64. A rad-hard many-core high performance DSP for space applications", DASIA (2014).

[13] Military ProASIC3/EL Low-Power Flash FPGAs
http://www.microsemi.com/index.php?option=com_docman&task=doc_download&gid=130697
[Last accessed Sept 2016]

[14] TLK2711-SP Space-rated 1.6 to 2.5 GBPS Transceiver
http://www.ti.com/product/TLK2711-SP
[Last accessed Sept 2016]

[15] SpaceFibre on Microsemi RTG4
https://www.star-dundee.com/knowledge-base/spacefibre-microsemi-rtg4
[Last accessed Sept 2016]