NTNU

Norwegian University of
Science and Technology

FPGA implementation of an efficient high-speed DVB-S2X block-interleaver

Youri Vladimirovitch Vassiliev

Master of Science in Electronics
Submission date: July 2018
Supervisor: Kimmo Kansanen, IES

Norwegian University of Science and Technology
Department of Electronic Systems

Preface

This Master's Thesis was written during the spring of 2018 at the Norwegian University of Science and Technology (NTNU), located in Trondheim, in association with WideNorth, as part of an Erasmus+ exchange program form Ghent University. This project has been conducted under the supervision of Professor Dr. Kimmo Kansanen, Department of Electronic Systems and Bjarne Rislw from WideNorth

This thesis consists of 5 chapters. The first part is the introduction that talks about the overarching project that this thesis is a part of. The second chapter gives background information about the subject. The third chapter presents a theoretical design while the forth chapter puts the theory into practice. The final chapter concludes the Master's Thesis.

First and foremost I would like to thank my mother and father who have supported me during my exchange to Norway. Without their help, I would not have succeeded.

I would also like to thank WideNorth for giving me this Master's Thesis that has allowed me to realise my potential.

And finally, I would like to thank Kimmo Kansanen, he might not realise it, but his simple suggestions and tips have inspired many solution to all the problems that I faced in my design.

Abstract

An overview of the DVB-S2 modem is given with the focus on the forward error correction block. The block-interleaver in the DVB-S2 and DVB-S2X standard is examined. Different interleaver configurations are evaluated. A suitable way to preform the interleaving process with one block of memory is found. The concept is proven with MATLAB models and implemented in VHDL. The entity has been verified and exceeds the desired performance.

Table of Contents

Preface **1**

Abstract **3**

Table of Contents **6**

List of Tables **7**

List of Figures **10**

Abbreviations **11**

1 Introduction **1**

2 Literature Review **3**
2.1 DVB-S2 modem 3
2.2 forward error correction 4
2.2.1 coding theory 4
2.2.2 coding and decoding of frames 5
2.2.3 LDPC in more detail 5
2.3 the block-interleaver 7
2.3.1 purpose 7
2.3.2 the DVB-S2 and DVB-S2X standard 8
2.4 existing block-interleaver designs 12
2.5 Stratix 10 SX 14
2.5.1 high level overview 14
2.5.2 Adaptive Logic Module 14
2.5.3 20-kilo-bit memory block 16

3 Interleaver design **19**
3.1 project description 19
3.1.1 objective 19

3.1.2 performance goal . . . 19
3.1.3 design priorities . . . 20
3.2 testing methodology . . . 20
3.2.1 modeling the interleaver as a permutation operation . . . 20
3.2.2 from model to hardware in steps . . . 21
3.2.3 examples . . . 21
3.3 block-interleaver models . . . 21
3.3.1 the block-interleaver as a matrix transposition . . . 21
3.3.2 memory block address generator as described in the literature . . . 23
3.3.3 memory block address generator based on a linear congruential generator . . . 25
3.3.4 expanding from examples to real use cases . . . 28
3.4 block-interleaver with only one memory block . . . 29
3.4.1 One-memory-block design explained . . . 29
3.4.2 permutation groups . . . 31
3.4.3 challenge in frame size scalability . . . 31
3.4.4 possible compromise for large frames . . . 33
3.4.5 solving the frame size scalability issue . . . 33
3.4.6 uninterrupted interleaving for one configuration . . . 37
3.4.7 uninterrupted interleaving across all configurations . . . 38
3.4.8 scalability of the one-memory-block design . . . 40
3.5 de-interleaving . . . 41
3.6 processing multiple sequential bits at a time . . . 41
3.6.1 subdividing the transposition operation . . . 41
3.6.2 subdividing the interleaver . . . 42
3.6.3 row read order permutation . . . 44
3.6.4 challenge with indivisibility of words into columns . . . 45
3.6.5 allowing variable word sizes . . . 47
3.7 the final design . . . 49

4 implementation and verification **51**
4.1 big-interleaver implementation . . . 51
4.2 ModelSim simulation and verification . . . 53
4.3 Quartus timing and resource usage analysis . . . 61

5 Conclusion **65**

Bibliography **67**

Appendix **69**

List of Tables

2.1 Block interleaver dimensions 9
2.2 row read out order for small frames 10
2.3 row read out order for normal frames 11

3.1 permutation group element count for each interleaver configuration 32
3.2 row-read-count pattern for all column sizes 44
3.3 column sizes when 16-bit words are used 46
3.4 column sizes when 20-bit words are used 46
3.5 column sizes when 18-bit words are used 47
3.6 column sizes when 30-bit words are used 47

4.1 big-interleaver maximum theoretical frequency 61
4.2 block resource usage 61
4.3 routing resource usage 62

List of Figures

2.1 functional block diagram of the DVB-S2 System 4
2.2 tanner graph representation of the parity check matrix 6
2.3 bit interleaved code modulation structure 8
2.4 8PSK bit interleaving of a normal frame 9
2.5 8PSK bit interleaving of a normal frame with coding rate of 3/5 10
2.6 interleaver memory block structure 12
2.7 address generators for WiMAX interleaver 13
2.8 modified Finite State Machine (FSM) 13
2.9 block diagram of the proposed interleaver/de-interleaver 14
2.10 configurable FIFO structure 14
2.11 Intel Stratix 10 FPGA architecture Block Diagram 15
2.12 Intel Stratix 10 FPGA ALM Block Diagram 15
2.13 Intel Stratix 10 LAB structure and interconnects overview 16
2.14 read-during-write data flow 17
2.15 mixed-port read-during-write: old data mode 17

3.1 alternative configuration for the address generator, 8 Phase-Shift Keying (PSK) 24
3.2 Alternative configuration for the address generator, 16 Amplitude and Phase-Shift Keying (APSK) 24
3.3 block diagram of the simplified structure 26
3.4 representation of the input range divided in two regions 26
3.5 example of a sequences created by an LCG based generator 27
3.6 representation of the input range divided in two regions after modification 27
3.7 interleaver contents of example 30
3.8 interleaver time efficiency 33
3.9 initial memory contents 34
3.10 memory contents after the first frame is read and the second frame is written 34
3.11 unwrapping the memory contents with the second frame inside of it . . . 35
3.12 memory contents after the second frame is read and the third frame is written 36

3.13 unwrapping the memory contents with the third frame inside of it 36
3.14 memory contents after the first frame is read and the second frame is written under a different configuration 39
3.15 unwrapping the memory contents with the second frame inside of it under a different configuration 39
3.16 block interleaver in a 6 column configuration for normal frames 43
3.17 first row of the big-interlear inside the small-interleaver 43
3.18 small-interleaver 44
3.19 contents of the small-interleaver after the columns have been written in the sequence of the inverse row permutation 45
3.20 interleaver in a 4 columns configuration for a normal frame 46
3.21 interleaver in a 3 column configuration for a word count of 3360 48
3.22 interleaver in a 4 column configuration for a word count of 3360 48
3.23 interleaver in a 5 column configuration for a word count of 3360 48
3.24 interleaver in a 6 column configuration for a word count of 3360 48
3.25 interleaver in a 7 column configuration for a word count of 3360 48
3.26 interleaver in a 8 column configuration for a word count of 3360 48

4.1 overview of the full interleaving process of 12 normal frames 58
4.2 overview of the full interleaving process of 12 small frames 58
4.3 start up behaviour for the small frame configuration 59
4.4 output small frame 0 and input small frame 1 59
4.5 output small frame 7 and input small frame 8 60
4.6 output small frame 8 and input small frame 9 60
4.7 big-interleaver implementation on the Field Programmable Gate Array (FPGA) fabric 62
4.8 the whole 1SX085HN3F43I3XG FPGA fabric 63

Abbreviations

ALM Adaptive Logic Module. 14, 16

APSK Amplitude and Phase-Shift Keying. 19, 21, 22, 24, 38, 40, 42, 44, 47, 49, 51, 65

AWGN Added White Noise Gaussian. 4

BCH Bose Chaudhuri Hocquenghem. 1, 3, 5

BICM Bit-Interleaved Coded Modulation. 1, 7

DAC Digital to Analogue Converters. 4

DSP Digital Signal Processor. 20

ESA European Space Agency. 1

FEC Forward Error Correcting. 1, 3

FIFO First In First Out. 12

FPGA Field Programmable Gate Array. 14, 20, 61–63

FSM Finite State Machine. 12, 13, 24, 27

Gbps gigabit per second. 19

Gs/s Giga-symbol-per-second. 1, 19

LAB Logic Array Block. 14, 16, 61

LCG Linear congruential Generator. 25, 27, 37

LDPC Low-Density Parity-Check. 1, 3, 5, 7–9

LUT Look-Up-Table. 14, 16, 29, 31, 32, 53, 61

M20K Memory 20-Kilo-bit. 14, 16, 29, 46, 61

MHz megahertz. 20, 38

MLAB Memory Logic Array Block. 14, 16

PSK Phase-Shift Keying. 19, 21, 22, 24, 27, 30, 34, 38, 42

VHTS Very High Throughput Satellite. 1

VSAT Very-small-aperture terminals. 1

Chapter 1

Introduction

The objective of this Master's Thesis is to design a DVB-S2X compliant block-interleaver, which will be used in a modem designed by WideNorth, for a research project commissioned by the European Space Agency (ESA). The purpose of this research project is to demonstrate a re-programmable wideband modem for Very High Throughput Satellite (VHTS) networks through ultra-wideband Very-small-aperture terminals (VSAT). The goal is to design a transmitter and receiver capable of handling a throughput of 1.4 Giga-symbol-per-second (Gs/s), which is around 3 to 4 times what current modems are capable of. The modem will be connected to various networks in a base station through an Ethernet interface. Wireless point-to-point and point-to-multipoint communication will be established through the existing VSAT satellite links that have a bent-pipe architecture.

The block-interleaver is only a part of the full Forward Error Correcting (FEC) block in the modem. Its purpose is to break the correlation caused by symbol mapping as it degrades the performance of the Low-Density Parity-Check (LDPC) decoder. The FEC block, which consists of Bose Chaudhuri Hocquenghem (BCH) encoding, LDPC encoding, and a block-interleaver, together with the constellation mapper form what is called a Bit-Interleaved Coded Modulation (BICM) structure.

Chapter 2

Literature Review

In this chapter a brief description will be given of the DVB-S2 modem. In this modem an overview is given of the FEC block. The function and importance of the block-interleaver in the FEC chain will be explained. Existing designs found in the literature will be presented. Because the block-interleaver is to be implemented on an Intel Stratix 10 FPGA its architecture will be studied.

2.1 DVB-S2 modem

In order for a message to be transmitted it needs to pass through the modem consisting of the following modules[1]:

- Mode adaptation: This is the input stream interface. Here the data is synchronised, null-packet are deleted and the resulting stream is encoded with CRC-8 coding. Multiple streams are merged from multiple inputs and then sliced into data fields. A Base-Band Header is appended in front of the data field to notify the receiver of the input stream format and Mode Adaptation type.
- Stream adaptation: Zero padding is introduced when the available user data for transmission is insufficient to completely fill up a Base-Band Frame.
- Forward Error Correction Encoding: Each frame is first encoded with BCH code and then followed up by LDPC code. If the modulation scheme is either 8 PSK, 16 APSK or 32 APSK then the resulting message is interleaved in a block-interleaver.
- Mapping: The data-stream is then mapped into an I-Q data stream with the use of a constellation mapper.
- Physical layer framing: Dummy frames are transmitted when no useful data is ready to be sent. A slot is devoted to physical layer signalling such as start-of-Frame delimitation and transmission mode definition. A regular raster of pilot symbols is introduced to facilitate Carrier frequency recovery in the receivers.

- Quadrature Modulation Base-Band Filtering: The I-Q data is modulated into a waveform. This digital wave is filtered with a squared-root raised cosine filter that has a roll-off factor of either 0.35, 0.25 or 0.20.
- Digital to analogue conversion: WideNorth has made the design choice of doing all the filtering in the digital domain. By using state of the art high-speed Digital to Analogue Converters (DAC) the signal is put directly onto the L-band for transmission [2].

A detailed overview of the entire DVB-S2 modem for transmission is seen in Figure 2.1.

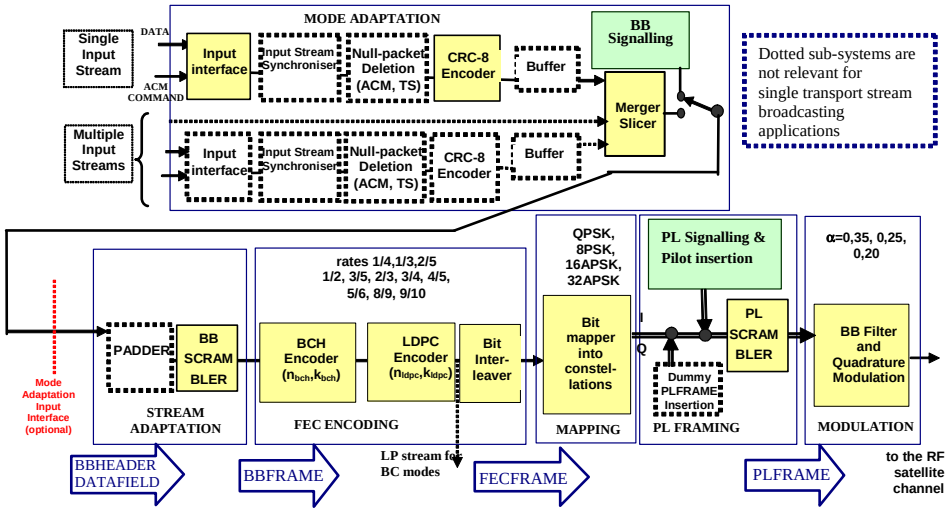

Figure 2.1: functional block diagram of the DVB-S2 System [1]

2.2 forward error correction

2.2.1 coding theory

In 1948 Claude Shannon established a theory that is now known as the noisy-channel coding theorem, but also referred to as the Shannon's theorem. It states that for any Added White Noise Gaussian (AWGN) channel, there exists a maximum rate for which data can be transmitted error-free with the use of error correcting codes [3]. The theorem, however, does not state what these error correcting codes are, only that such a code can exist. This opened up a new field in computer science. Since then, scientists and engineers have tried to come up with codes that reach this limit yet remain practical and feasible to implement. Practical means that the algorithm that is to be implemented, grows in polynomial complexity with respect to the message length.

2.2.2 coding and decoding of frames

In the DVB-S2 and DVB-S2X standard the message is encoded with BCH, afterwards it is encoded again with LDPC [1, 4]. LDPC is currently the best coding scheme available as it has the advantage of design flexibility, decoding simplicity, and a universally excellent error correction performance over various channel types [5]. In a soft-decision approach LDPC, encoded messages are decoded using a sum-product algorithm, also known as the belief propagation algorithm. This algorithm needs multiple iterations in order for the received data to converge to the most probable code-word. In certain cases, the maximum amount of iterations is exceeded, and the algorithm is halted prematurely. The resulting sub-optimal estimation is then passed through the BCH decoder. Here, the remaining errors are corrected in a reliable time-frame. It is shown that this combination of BCH and LDPC code provide excellent error correction performance and achieve an extremely low bit-error rate [6]. Since the error-correcting performance is improved compared to previous standards, it becomes possible to employ higher order modulation schemes to increase the spectral efficiency and maximal data rate [6].

2.2.3 LDPC in more detail

LDPC code preforms bad under certain circumstances. In order to understand what these circumstances are, a brief explanation is needed regarding the decoding process. The LDPC code is in essence a linear block code[7]. This means that data is encoded using a generator matrix and decoded using a parity check matrix. A property of this parity check matrix is that it has more zeroes then ones. This sparseness guarantees a linear increase in time complexity of decoding with increased block size [7, 8]. An example of such a parity check matrix is provided below [7].

$$H = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix} \tag{2.1}$$

This LDPC decoder takes 8 bits of input and is constrained by 4 linear equations. These linear equations are:

$$\begin{cases} f_1 = C_2 + C_4 + C_5 + C_8 (\mathrm{mod} 2) = 0 \\ f_2 = C_1 + C_2 + C_3 + C_6 (\mathrm{mod} 2) = 0 \\ f_3 = C_3 + C_6 + C_7 + C_8 (\mathrm{mod} 2) = 0 \\ f_4 = C_1 + C_4 + C_5 + C_7 (\mathrm{mod} 2) = 0 \end{cases} \tag{2.2}$$

Another way to represent this parity check matrix is by visualising it in a Tanner graph as seen in Figure 2.2. The C-nodes are the variable nodes that contain the received data. The f-nodes are the check nodes where each linear equation result is calculated. The edges represent an addition of the C-node in the f-node.

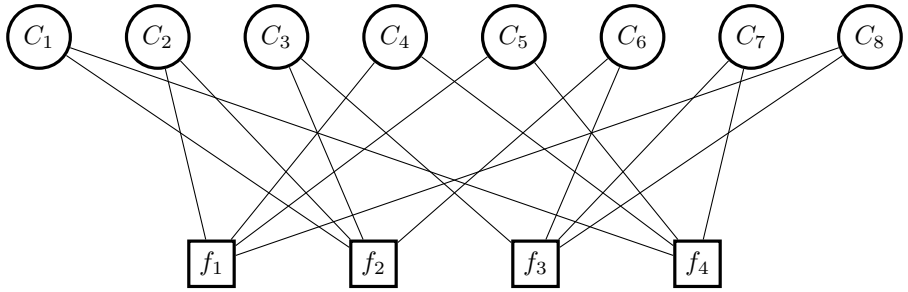

Figure 2.2: tanner graph representation of the parity check matrix [7]

$$C \cdot H^T = 0 \tag{2.3}$$

Consider C to be the received data. If equation 2.3 is satisfied, then the data contains no errors. Another way to phrase this is that for each linear equation in 2.2, the amount of bits that are 1 in each equations must be even. In other words, f1 through f4 must be 0. When a message contains an error, the condition in equation 2.3 is not satisfied, iterative decoding will takes place. For instance, lets assume that the received code-word from C1 through C8 is 10110100. The linear equations 2.2 become:

$$\begin{cases} f_1 = 0+1+0+0 (\mathrm{mod} 2) \\ f_2 = 1+0+1+1 (\mathrm{mod} 2) \\ f_3 = 1+1+0+0 (\mathrm{mod} 2) \\ f_4 = 1+1+0+0 (\mathrm{mod} 2) \end{cases} \iff \begin{cases} f_1 = 1 \\ f_2 = 1 \\ f_3 = 0 \\ f_4 = 0 \end{cases} \tag{2.4}$$

In this example the equations for f1 and f2 are not satisfied, therefore, the C-nodes in the Tenner graph, that are connected to both the check-nodes f1 and f2, must change. In this case C_2 becomes 1. The final correct code-word then becomes 10110100. This is the essence of the bit-flipping algorithm. If multiple bits are wrong, then the bit-flipping process is repeated until equation 2.3 is satisfied. In bigger coding schemes, a C-node is flipped if the majority of the f-nodes it is connected to are 1. The Bit-flipping algorithm is essentially a message passing algorithm. The C-nodes broadcast their data to their f-nodes. In turn the f-nodes broadcast their parity-check information back to their C-nodes. The C-nodes change their value based on majority consensus [8]. These steps are repeated until equation 2.3 is satisfied.

When a message is received, the symbols are de-mapped into a word consisting of bits that are all binary. This is called a hard-decision approach. A better approach would be to do a log-likelihood estimation and infer both the chance of bit x being a 1 and a 0. $P(x = 1)$ is $1 - P(x = 0)$ so both probabilities can be described by a single value. This value is the logarithm of the proportion of $P(x = 1)$ over $P(x = 0)$ as per the following formula:

$$L(x) = \log\left(\frac{P(x=1)}{P(x=0)}\right) \tag{2.5}$$

This values is used as a metric to determine whether or not bit x is closer to being a 0 or a 1 [8].

The decoder will thus receive real-valued probabilities instead of bits. This approach is much more computationally heavy, but yields much better results. The best performing algorithm for this problem is the sum-product algorithm [3]. Just like the bit-flipping algorithm, the sum-product algorithm is also a message passing algorithm.

In the first part of the iteration the C-nodes broadcast their normalised probabilities to the f-nodes. The f-nodes sum up all the values they receive. In the second step these f-node broadcast their results back to the C-nodes. Each individual C-node subtracts its own contribution to the sums before multiplying everything they receive. The current values are replaced by the products. In order to verify if the message is error-free, the f-nodes threshold the incoming C-node values between step one and two. Anything smaller then 0.5 is considered to be a 0 and anything above is considered to be a 1. These values are then put through equations 2.3 and checked for parity. If the parity conditions are met then the message is done with the decoding process, if not, the algorithm proceeds to step two and loops back to step one. The probabilities themselves will eventually converge after multiple iterations. This process continues until the parity is satisfied or until the maximum allowed iterations is reached.

Both the soft-decision and hard-decision approach have something in common. If more then one received bit is wrong then the algorithm suddenly needs more iterations in order to successfully reconstruct the message. LDPC works therefor best in environments where errors are uncorrelated and the channel memory-less[6]. More information regarding the topic of decoding can be found here [3, 7, 8].

2.3 the block-interleaver

2.3.1 purpose

When information is transmitted, it is chopped into smaller messages where multiple bits are mapped into to one symbols. It stands to reason that when one symbol is incorrectly received, that multiple bits are affected by the distortion. In case of a soft-decision architecture, the individual bits that made up that symbol, receive a smaller certainty when the symbols are de-mapped. Grouping together of bits into symbols is a source of correlation. Where if one bit is affected then the surrounding bits are more likely to be affected as well.

The purpose of an interleaver is to rearrange the bit ordering of a message, so that the bits that become correlated through symbol mapping, are spread out throughout the whole message. Whenever an error does occur, the surrounding bits will be uncorrelated and unaffected after the deinterleaving process. This makes the bit stream memory-less and uncorrelated [6]. An interleaver enhances the ability of the decoder to correct the errors in a message. A soft-decision LDPC decoder requires less iterations to correct multiple regions with small uncertainties, then one region with a big uncertainty. This is because the error is now dilute across multiple linear equations [6]. Bit-interleaving of LDPC code guarantees good performance [6]. A configuration that uses error correcting code together with an interleaver that covers its shortcomings is a BICM scheme. Such a configuration is shown in Figure 2.3.

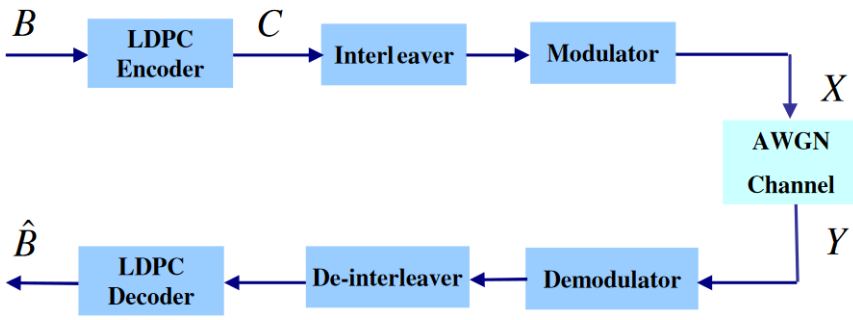

Figure 2.3: bit interleaved code modulation structure [5]

Interleaving can also happen between different messages. Bits from multiple messages can be swapped between each other. These messages are then send across different channels. This makes transmissions more robust in scenarios where channel fading is an issue. In this process the correlation of data caused by transmission across an individual channel is broken up by sending the data across multiple channels and assembling the data back together at the destination. In case of DVB-S2 and DVB-S2X , this type of interleaving is not necessary as the sender and receiver will be stationary.

2.3.2 the DVB-S2 and DVB-S2X standard

The interleaver that follows the LDPC encoder in the DVB-S2 and DVB-S2X standard is a Block-interleaver [1, 4]. There are two types of frames that need interleaving, a normal frame that is 64800 bits wide, and a short frame that is 16200 bits wide. There is also a medium frame that is 32400 bits wide, but that one doesn't need to be interleaved. Whether or not a normal or small frame needs to be interleaved, depends on the modulation scheme. In the DVB-S2 specification, normal and small frames that have to be transmitted over 8PSK, 16APSK and 32APSK need to be interleaved. In the DVB-S2X extension, additional modulation schemes for the normal frame have been added, for which interleaving also needs to take place. These modulation schemes are 64APSK, 128APSK, and 256APSK. The block interleaver is not a fixed implementation whereby every frame is subject to an identical operation. Different modulation schemes require different block dimensions. 8PSK, 16APSK 32APSK, 64APSK, 128APSK, and 256APSK respectively encode 3,4,5,6,7, and 8 bits of date. The amount of columns that are present in the block-interleaver for each modulation scheme, is equal to the amount of bits each modulation scheme encodes. For example the block interleaver for 8PSK would have 3 deep columns while the block interleaver for 256APSK would have 8 shallower columns. The incoming data stream is written vertically, column-wise from top to bottom, starting with the left-most column, and progressing from left to right. When the block is full, the data is read horizontally, row-wise from left to right, starting from the topmost row, and progressing from top to bottom. This process is visualised in Figure 2.4 in the case of 8PSK modulation. Because the column count is chosen to be the same size as the symbol bit size, whole rows can be taken to the constellation-mapper as they are.

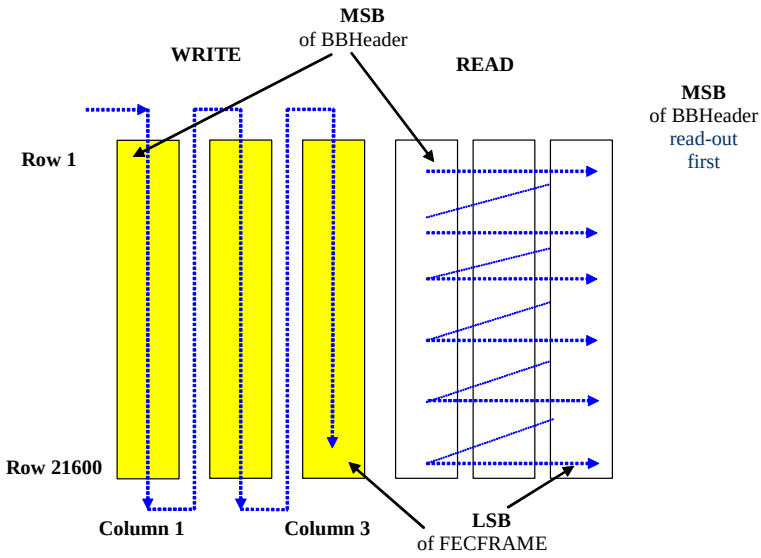

Figure 2.4: 8PSK bit interleaving of a normal frame [1]

A summary of all the block-dimensions can be found in Table 2.1 for both the normal and the small frames.

Table 2.1: Block interleaver dimensions [1, 4]

Modulation	8PSK	16APSK	32APSK	64APSK	128APSK	256APSK
column size	3	4	5	6	7	8
normal	21600	16200	12960	10800	(9258)	8100
small	5400	4050	3240	N/A	N/A	N/A

When one tries to divide 64800 bits equally into 7 columns, he or she will find that the bit-size of each column becomes 9257,143. In the specification it is stated that for the case of 128APSK, an additional 6 bit padding is introduced after the LDPC encoder, in order to achieve an integer amount of symbols for the interleaver and constellation mapper [4].

Frames can be coded in a variety of coding rates. The ordering in which individual bits inside each row need to be read out, is tied to the coding rate. In the DVB-S2 standard, the read order of a row, for the case of 8PSK with a coding rate of 3/5, is reversed. This is depicted in Figure 2.5.

For the DVB-S2X standard the story is a bit more complicated as every coding rate comes with a custom read-order. All the read orders for the small frame and normal frame can be found in Table 2.2 and 2.3 respectively. As an example, the bit interleaver pattern 102 means that for each row, the middle entry (1) is read out first, followed by the leftmost entry (0), and finally the rightmost entry (2) [4].

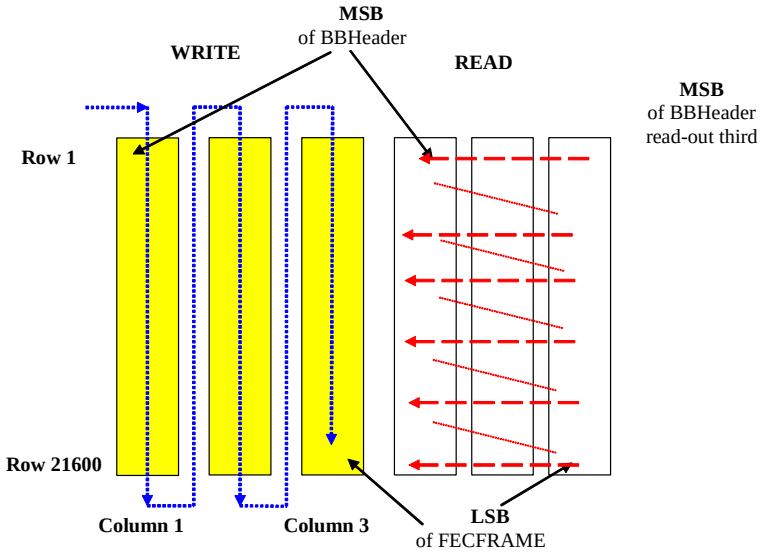

Figure 2.5: 8PSK bit interleaving of a normal frame with coding rate of 3/5 [1]

Table 2.2: row read out order for small frames [4]

Modulation	coding rate	read sequence
8PSK	7/15	102
8PSK	8/15	102
8PSK	26/45	102
8PSK	32/45	012
4+12APSK	7/15	2103
4+12APSK	8/15	2103
4+12APSK	26/45	2130
4+12APSK	3/5	3201
4+12APSK	32/45	0123
4+12+16rbAPSK APSK	2/3	41230
4+12+16rbAPSK APSK	32/45	10423

Table 2.3: row read out order for normal frames [4]

Modulation	coding rate	read sequence
8PSK	23/36	012
8PSK	25/36	102
8PSK	13/18	102
4+12APSK	26/45	3201
4+12APSK	3/5	3210
8+8APSK	18/30	0123
4+12APSK	28/45	3012
4+12APSK	23/36	3021
8+8APSK	20/30	0123
4+12APSK	25/36	2310
4+12APSK	13/18	3021
4+12+16rbAPSK	2/3	21430
8+16+20+20APSK	7/9	201543
8+16+20+20APSK	4/5	124053
8+16+20+20APSK	5/6	421053
2+4+2APSK	100/180	012
2+4+2APSK	104/180	012
8+8APSK	90/180	3210
8+8APSK	96/180	2310
8+8APSK	100/180	2301
4+12APSK	140/180	3210
4+12APSK	154/180	0321
4+8+4+16APSK	128/180	40312
4+8+4+16APSK	132/180	40312
4+8+4+16APSK	140/180	40213
16+16+16+16APSK	128/180	305214
4+12+20+28APSK	132/180	520143
128APSK	135/180	4250316
128APSK	140/180	4130256
256APSK	116/180	40372156
256APSK	20/30	01234567
256APSK	124/180	46320571
256APSK	128/180	75642301
256APSK	22/30	01234567
256APSK	135/180	50743612

2.4 existing block-interleaver designs

Before setting of on a journey to create an interleaver that will take over the world, some existing designs will be studied first.

In paper [9] a block-interleaver is created that can be used in both the WiMax and WLAN standard. In both cases a block of data is subject to two permutations as seen in equation 2.6 and 2.7.

$$M_k = \left(\frac{N}{d}\right) \times (k\%d) + \left\lfloor \frac{k}{d} \right\rfloor \tag{2.6}$$

$$J_k = s \times \left\lfloor \frac{M_k}{s} \right\rfloor + \left(\left(M_k + N - \left\lfloor d \times \frac{M_k}{N} \right\rfloor \right) \%s \right) \tag{2.7}$$

Here d represents the number of columns, M_k is the output of the first permutation while J_k is the output of the second permutation, both are indexed by k from 0 to $N-1$ in which N represents the message bit-size, and finally s is the maximum of 1 and $N/2$. Two memory blocs are used in this design. While data is written to the first block, the second block is being read from using the generated permutations as addresses, afterwards the roles reverse. A similar design is proposed in paper [10]. The memory block setup is depicted in Figure 2.6 taken from the paper.

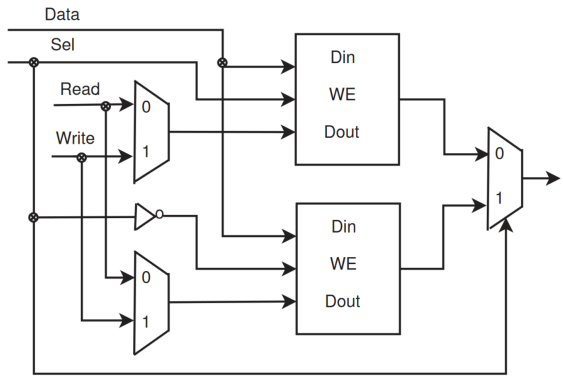

Figure 2.6: interleaver memory block structure [10]

In this architecture the address is generated locally by a rather complicated circuit network, accompanied by a rather complicated FSM, as seen in Figure 2.7 and 2.8 respectively, taken from the paper.

In paper [11] a similar block interleaver is designed, but instead of using memory blocks as a storage units, the author uses a First In First Out (FIFO) structure to store the data as depicted in Figure 2.9 and 2.10. This design is faster and more flexible but consumes more power [11].

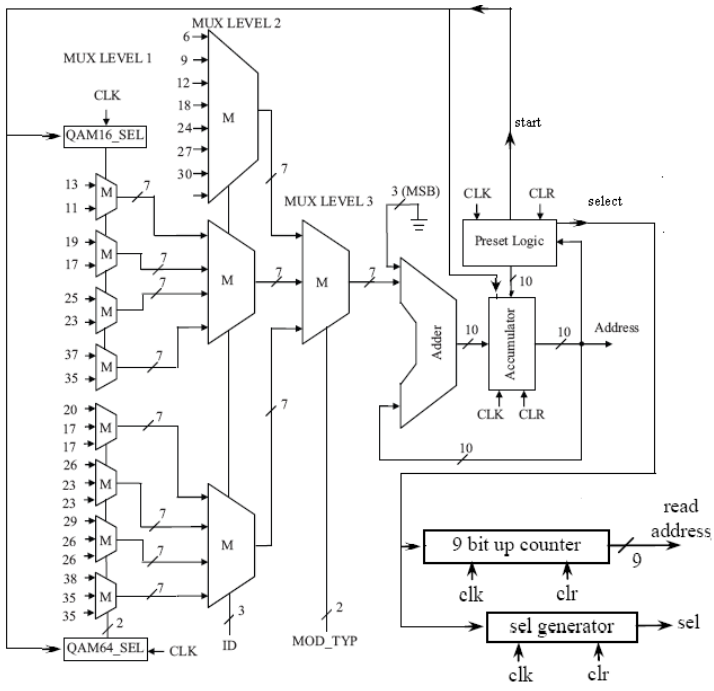

Figure 2.7: address generators for WiMAX interleaver [9]

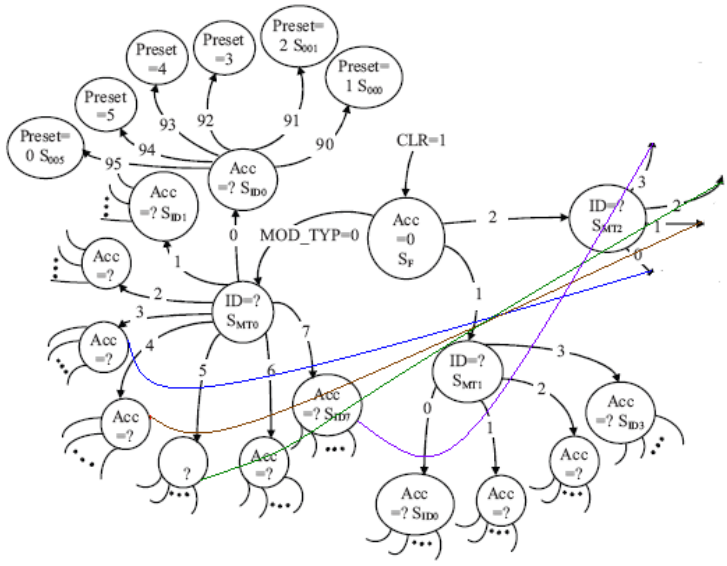

Figure 2.8: modified FSM [9]

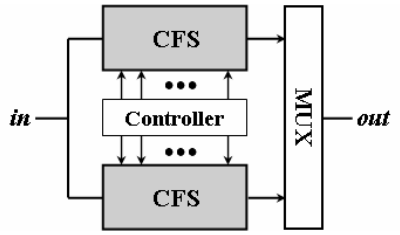

Figure 2.9: block diagram of the proposed interleaver/de-interleaver [11]

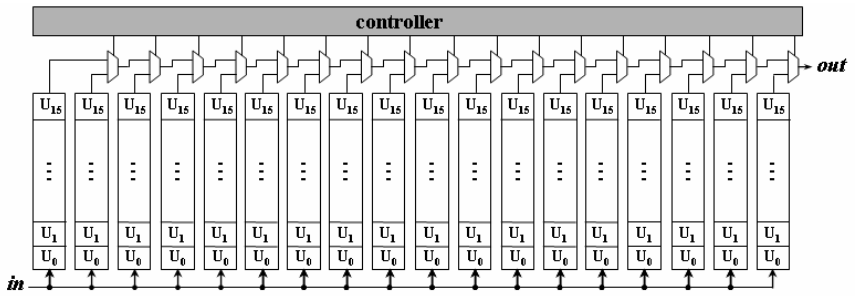

Figure 2.10: configurable FIFO structure [11]

2.5 Stratix 10 SX

WideNorth hasn't fully decided yet on what FPGA will be used in the final product, but it will most likely be the Stratix 10 SX FPGA series from Intel. The interleaver that will be designed in this Master's thesis, will therefore be geared towards, and optimised for, this particular FPGA series. The resources available on the Stratix 10 SX series will now be discussed.

2.5.1 high level overview

A high level overview of the chip can be seen in Figure 2.11. Here the HPS is a Quad ARM Cortex-A53 Hard Processor System, the SDM is a Secure Device Manager, and the EMIB is an Embedded Multi-Die Interconnect Bridge [12].

The most important blocks for the interleaver design are the following: Adaptive Logic Module (ALM), Logic Array Block (LAB), Memory Logic Array Block (MLAB) and Memory 20-Kilo-bit (M20K) blocks. These will be discussed in more detail.

2.5.2 Adaptive Logic Module

The ALM is a block where combinatorial logic is realised. It contains two adaptive Look-Up-Table (LUT), two two-bit full adders, four multiplexers, and four latched outputs that can be bypassed. The structure of the ALM can be seen in Figure 2.12.

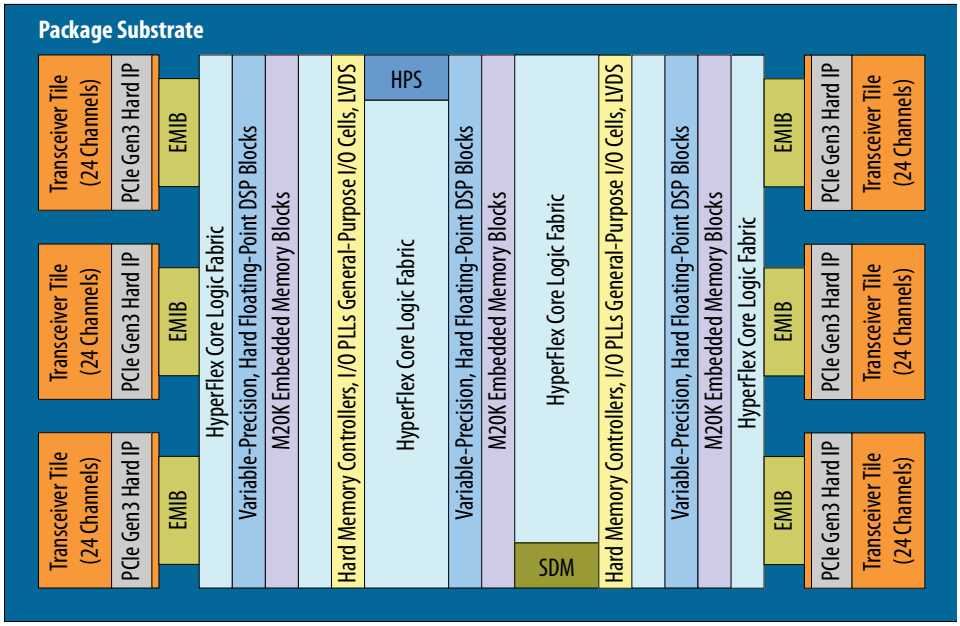

Figure 2.11: Intel Stratix 10 FPGA architecture Block Diagram [12]

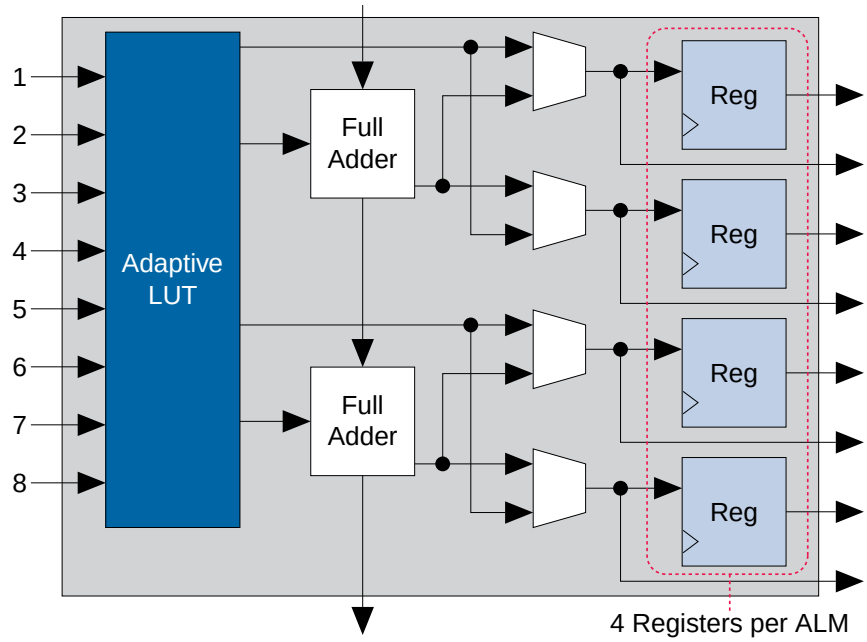

Figure 2.12: Intel Stratix 10 FPGA ALM Block Diagram [12]

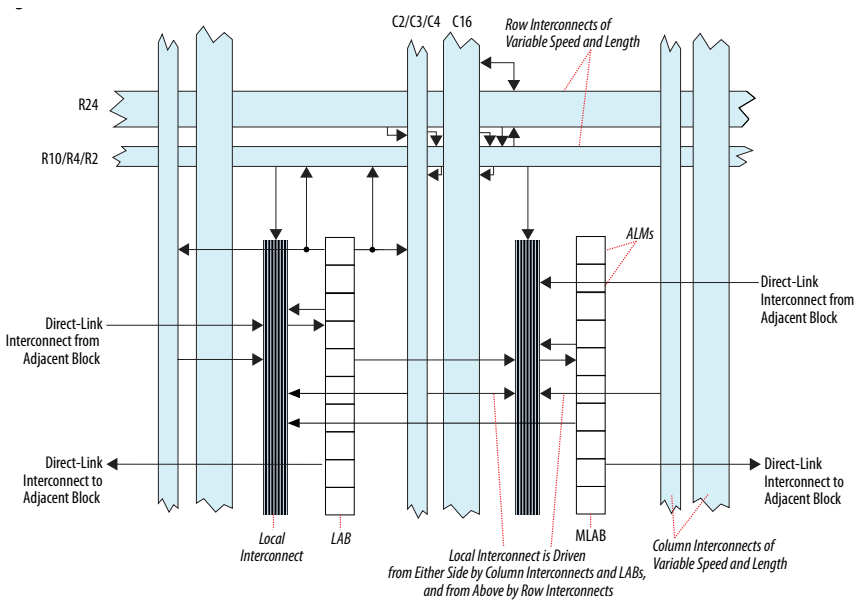

Figure 2.13: Intel Stratix 10 LAB structure and interconnects overview [13]

The ALMs are grouped together in groups of 10 into what is called a LAB [13]. The full structure can seen in Figure 2.13. This makes it easy to create fast and complex combinatorial structures, by feeding the output of one one ALM into the input of an adjacent ALM, through a dedicated local interconnect. Adjacent ALMs have their full-adder carry bit daisy-chained between each other to create ripple-adders of arbitrary length. Over a quarter of all LABs can be used as bit-addressable memory block that are of 640 bits in size. This structure is referred to as an MLAB. This is achieved by using modified LUTs as addressable storage space [13].

2.5.3 20-kilo-bit memory block

The M20K is a dedicated memory unit that can store 20,480 bits of data [14]. These blocks are flexible in how the input-size versus address space can be configured. The following options are possible: 512 x 40-bit, 1024 x 20-bit and 2048 x 10-bit. 8-bit, 16-bit, and 32-bit is not natively supported, this means that 20% of the total storage space will be wasted. Bigger address space or input size can be achieved by combining multiple M20K blocks together with multiplexers. M20K blocks can be configured as up to two true port and 4 simple port configuration. A true dual port allows one to read and write independently of each other, to two different addresses, on the same clock-cycle. On a simple dual port, one can read and the other one can write, but are excluded from both reading and both writing at the same time. On a four simple port configuration, only two reads and two writes are allowed each clock cycle.

It is possible to both read and write the same address during one clock-cycle. The behaviour however depends on whether or not a single port or a mixed dual port configuration is chosen [14]. If a single port solution is chosen, then the output of the read operation during a write, can be configured to output either the new data or don't care values on the next clock cycle. In a dual port, each port has its own output. In a mixed port configuration it is possible to cross the outputs. In essence, the read result of port B can be routed through the output of port A and vice versa. A mixed port configuration is visualised in Figure 2.14.

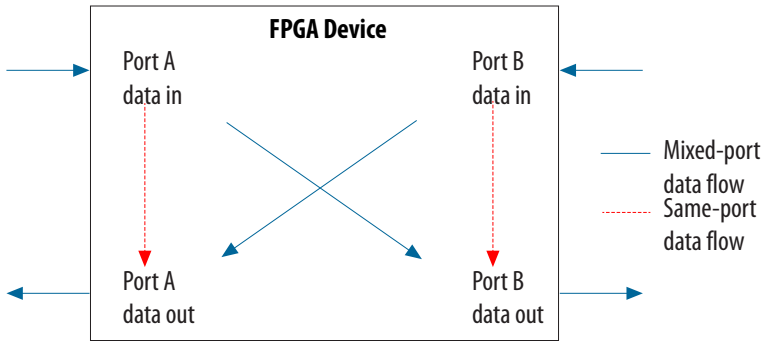

Figure 2.14: read-during-write data flow [14]

In this mixed port configuration, the output of a read during write to the same address, can now either be the newly written data, or the old data that is currently being overwritten [14]. In the latter configuration it is possible to take out stored data from memory, use it in the next block, and have it overwritten with new data, all in one clock cycle. This is comparable to a bit shift operation, but on a larger scale. This behaviour can be seen in the wave form in Figure 2.15.

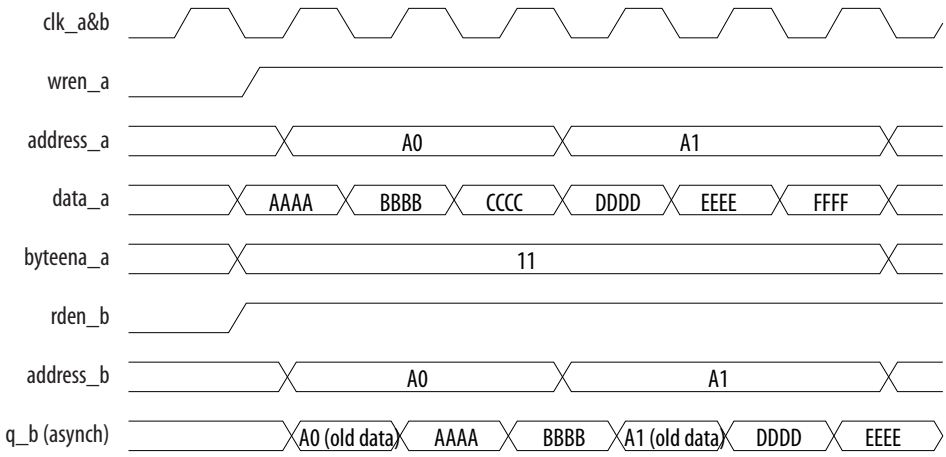

Figure 2.15: mixed-port read-during-write: old data mode [14]

Chapter 3

Interleaver design

In this chapter the design goal is stated along with its constraints. The metrics that will be used to benchmark the performance of the interleaver will be specified. The design process begins with a mathematical model using MATLAB, that proves the correctness of a concept. In the final part of this chapter, the mathematical model is broken up into a sequence of smaller independent operations that can easily be implemented in hardware.

3.1 project description

3.1.1 objective

The objective consists of creating a DVB-S2 compliant block-interleaver that can handle normal and small frames that are destined for 8 PSK, 16 APSK, and 32 APSK modulation. An added bonus objective is to make it expandable to the DVB-S2X standard, where the interleaver would additionally have to support normal frames that are destined for 64 APSK, 128 APSK and 256 APSK modulation. Subsequent frames have no gaps in between them and each subsequent frame can be destined for a different modulation scheme, that in turn requires all frames to be interleaved under different interleaver configurations. This means that the interleaver must be capable of accepting frames as a never ending streaming data and be able to reconfigure itself on the fly. There is an exception when it comes to transitioning between frame types, from normal to small and vice versa. It is tolerable to delay the incoming frame in order to read out the current frame from the block of memory and reconfigure the interleaver for a different frame size.

3.1.2 performance goal

The goal of the research project is to create a DVB-S2 transmitter and receiver capable of handling 1.4 Gs/s. Each 32 APSK symbol encodes 4 bits. Therefore the target throughput should be $1.4 \times 4 = 5.6$ gigabit per second (Gbps). The design choice has been made to process 16 bits in parallel every clock cycle at a clock-speed of 350 megahertz (MHz) to

reach this throughput. The addition of the DVB-S2X standard, that allows higher order modulation, would push the maximum throughput over that limit.

3.1.3 design priorities

The metrics that are important in this design are simplicity, engineering time and time to market. This means that code simplicity, design simplicity, human resource, and ease of integrating the module into the system is valued over FPGA resource use and energy usage to a certain degree so long as the performance goal is met.

3.2 testing methodology

An analogy can be made in the way that the interleaver model is verified with the method by which a Digital Signal Processor (DSP)-system is verified. A DSP-system is tested by introducing a Dirac-impulse at the input and measuring the impulse response on the output. The reason for this is that the Dirac-impulse is a neutral element to the system. What flows out of the output is the function that identifies the operation done by the system. The tests that will be preformed on the mathematical models of the interleaver are similar in nature.

3.2.1 modeling the interleaver as a permutation operation

The operation done by the block-interleaver as described in section 2.3.2 can be described by a permutation. A vector will be used to represent a frame. A permutation that represents the interleaving process, is applied to this vector. The resulting vector represents the interleaved frame after the interleaving process. The neutral element of a permutation is the identity permutation.

The permutation that is preformed by the interleaver will from henceforth be denoted as $\mathcal{P}$, the identity permutation as $\mathbb{1}$, and the bit-size of the input as N. The normal frame has an N of 64800 and the short frame has an N of 16200. A permutation can be written in two ways. One of those is the Cauchy's two-line notation. Permutation $\mathcal{P}$ and $\mathbb{1}$ can be seen in this notation in Equation 3.1 and 3.2 respectively.

$$\mathcal{P} = \begin{pmatrix} 0 & 1 & 2 & \dots & m-1 \\ \mathcal{P}(0) & \mathcal{P}(1) & \mathcal{P}(2) & \dots & \mathcal{P}(N-1) \end{pmatrix} \tag{3.1}$$

$$\mathbb{1} = \begin{pmatrix} 0 & 1 & 2 & \dots & N-1 \\ 0 & 1 & 2 & \dots & N-1 \end{pmatrix} \tag{3.2}$$

For this Master's Thesis a custom permutation notation will be used. This is for the sake of simplicity and because it corresponds to the output given by the MATLAB code. The custom notation is the Cauchy's two-line notation but without explicitly writing down the first row. Instead, the first row is always implied to go from 0 to N-1 and omitted from the notation. This notation can be seen in equation 3.3 and 3.4 for $\mathcal{F}$ and $\mathbb{1}$ respectively. Because the notation is essentially becomes a vector the permutations will henceforth be referred to as permutation vectors.

$$\mathcal{P} = [\mathcal{P}(0), \mathcal{P}(1), \mathcal{P}(2), \dots, \mathcal{P}(m-1)] \tag{3.3}$$

$$\mathbb{1} = [0, 1, 2, \ldots, m-1] \tag{3.4}$$

3.2.2 from model to hardware in steps

The first step of the design process is to create a high-level model that will serve as a definition of the interleaver operation. This model will be used to define the permutation vector $\mathcal{P}$. The next step is to break up the high-level model into a model that consists of a sequence of smaller operations that can easily be translated into hardware operations. The validity of this lower-level model can be verified by comparing it to the original high-level model that serves as the definition. Consider the following low level-model that realises a permutation $\mathcal{F}$.

$$\mathbb{1} \times \mathcal{P}_{sys} = \mathcal{P}_{out} \tag{3.5}$$

$$\mathbb{1} \times \mathcal{F}_{sys} = \mathcal{F}_{out} \tag{3.6}$$

These two models consisting of operation $\mathcal{P}_{sys}$ and $\mathcal{F}_{sys}$ are equal if and only if the outputs $\mathcal{P}_{out}$ and $\mathcal{F}_{out}$ from a neutral element are equal. This way of testing guarantees that the low-level model, destined for translation into hardware, is correctly implemented. This will serve as mathematical proof of correctness of the system.

The mathematical model can accept integers, but the eventual hardware will interleave binary data. In order to verify the correct implementation of the entity in hardware, the MATLAB model will generate an input frame using a random number generator. This frame is interleaved using the high level model that serves as the definition. In a test bench, the generated frame is fed to the input of the device. The output will then be compared to the expected output that was created by the high level model.

3.2.3 examples

For the sake of simplicity, small examples will be given to demonstrate the different models and ideas. Later on, the concepts are extrapolated to larger structures that will serve as a basis for the hardware design. In these limited example the frame size N is 12 and the supported modulations are 8 PSK and 16 APSK. This means that the example interleaver has two configurations. For 8 PSK it is configured as a block consisting of 3 columns of size 4 and for 16 APSK it is configured as a block consisting of 4 columns of size 3.

3.3 block-interleaver models

3.3.1 the block-interleaver as a matrix transposition

The block-interleaver as described in section 2.3.2 can be seen as a matrix transposition. Data is written to the block from top to bottom and from left to right. The data is then read from left to right and from top to bottom. The relationship between the reading sequence and the writing sequence is that they are a transposition of each other. If the data inside the matrix is transposed then the reading sequence needs to be transposed as well to preserve the function. Transposing the reading sequence will result in it becoming equal to the writing sequence. Because the reading and writing sequence become equal they do not

contribute anymore to the permutation. The only operation left is the transposition of the data. This implies that the interleaving process can be translated to a transposition of the data. This is significant as the problem of implementing an interleaver has been translated in to the implementation of a transposition operation. The permutation of an interleaver is therefore equal to the permutation realised by a matrix transposition.

In reality no real transposition operation will happen inside the memory modules. What will happen instead is that the elements of the permutation vector will be used as an address. The memory will thus be accessed in a sequence defined by the permutation vector.

Consider the following neutral element in equation 3.7. This frame will be interleaved in both configurations of the interleaver. First the frame is written to the block as can be seen in equation 3.9 and 3.8 for the case of 8 PSK and APSK respectively. Afterwards the data is transposed as can be seen in equation 3.11 and 3.10. Afterwards the data is read and the results can be seen in equation 3.12 and 3.13.

$$\mathbb{1} = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11] \tag{3.7}$$

$$Interleaver_{(8PSK)}(\mathbb{1}) = \begin{bmatrix} 0 & 4 & 8 \\ 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \end{bmatrix} \tag{3.8}$$

$$Interleaver_{(16APSK)}(\mathbb{1}) = \begin{bmatrix} 0 & 3 & 6 & 9 \\ 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \end{bmatrix} \tag{3.9}$$

$$Interleaver^{T}_{(8PSK)}(\mathbb{1}) = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix} \tag{3.10}$$

$$Interleaver^{T}_{(16APSK)}(\mathbb{1}) = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 11 \end{bmatrix} \tag{3.11}$$

$$\mathcal{P}_{(8PSK)} = [0, 4, 8, 1, 5, 9, 2, 6, 10, 3, 7, 11] \tag{3.12}$$

$$\mathcal{P}_{(16APSK)} = [0, 3, 6, 9, 1, 4, 7, 10, 2, 5, 8, 11] \tag{3.13}$$

The MATLAB code that was used to demonstrate these operations can be found in Listing 3.1.

Listing 3.1: MATLAB code that realises the interleaver as a transposition operation

```
1 frame_size = 12;
2 column_count = 3;
3 column_size = frame_size / column_count;
4 input = 0:(frame_size-1); %neutral element
```

```
5
6  Interleaver_A = reshape(input,[],column_count);
7  Interleaver_B = Interleaver_A';
8  ideal_permutation_vector = reshape(Interleaver_B,1,[]);
9
10 disp(input);
11 disp(Interleaver_A);
12 disp(Interleaver_B);
13 disp(ideal_permutation_vector);
```

Storing a full permutation vector for small frames is simple but expensive.The most straight forward solution for this problem is to recognise that there is a patter that can be exploited. This way it is possible to generate the addresses as they are needed.

3.3.2 memory block address generator as described in the literature

One way of generating this permutation vector is by using equation 2.6 [9, 10, 11]. This equation has been modified a bit to suit the example.

$$\mathcal{P}_i = V \times (i\%H) + \left\lfloor \frac{i}{H} \right\rfloor \tag{3.14}$$

H stands for horizontal and represents the column count while V stands for vertical and represents the column depth. This permutation vector is generated by the use of a counter, this counter value is represented by i and goes from 0 to N. Equation 3.14 can be split into two parts along the sum operation. The first part iterates over the rows. This is done by moving in steps that are the size of a column width, which is achieved by the multiplication of the counter value i with V. The counter value i wraps around and continues back at the beginning of the row duo to the modulo operation. The second part determines what row is being accessed by providing an offset value. This is done by dividing the counter value by the column count and rounding it down. When the first part is finished traversing a row, the second part will increase the offset by one so that the first part will will start traversing the next row. The MATLAB code that puts this theory into action can be seen in Listing 3.2

Listing 3.2: MATLAB code that generate address with a modulo of an index

```
1  frame_size = 12;
2  column_count = 3;
3  column_size = frame_size / column_count;
4  input = 0:(frame_size-1); %neutral element
5
6  permutation = zeros(1,frame_size);
7  for i=0:(frame_size-1)
8      index = column_size * mod(i,column_count) + floor(i/
           column_count);
9     permutation(i+1) =  input(index+1);
10 end
11
12 disp(input)
```

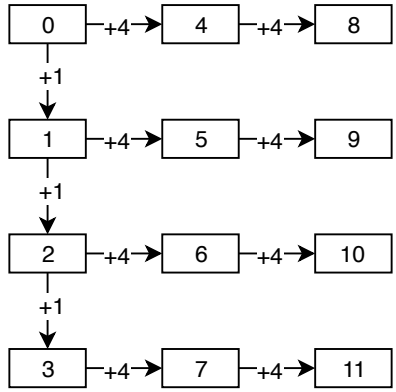

Figure 3.1: alternative configuration for the address generator, 8 PSK

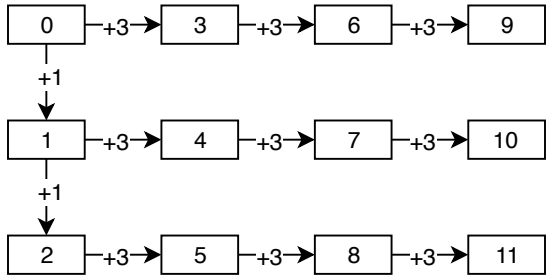

Figure 3.2: Alternative configuration for the address generator, 16 APSK

13 **disp**(permutation)

This generator does its job well, but the multiplication and division operation are expensive to implement in hardware. The modulo operation on the other hand is simply not synthesizable unless it is a power of 2 or a custom implementation.

The same function, however, can be implemented using a counter, a full adder and a FSM for some needed intelligence. The counter will be used to offset the row. The adder copies the counter value and repeatedly adds the column size value to it in order to traverse the row. At the end of the row the counter is increased by one and serves as the next address. Afterwards the adder takes over the address generation. This continues until the counter reaches a threshold where it resets back to zero for the next frame. This operation is visualised in Figure 3.1 for the case of 8 PSK and in Figure 3.2 for the case of 16 APSK. To switch from one configuration to the next the, the only parameters that would have to change are the counter threshold and the column size value that is used in the addition.

3.3.3 memory block address generator based on a linear congruential generator

Another way to generate the addresses is by using a Linear congruential Generator (LCG) that recursively generates all the elements of the permutation. An LCG is usually used as a simple random number generator. In this case it will be appropriated to generate addresses in an orderly fashion by constructing what would generally be called a bad and predictable random number generator. The general formula has three parameters and looks as follows:

$$x_{n+1} = a \times X_n + b \mod m \tag{3.15}$$

Parameter a is set to one because no multiplication is needed to achieve the goal. Parameter b is set to the $column\_size$ value in order to traverse the row. Parameter m is set to $N-1$. When the last element of the first row is accesses, the next address after the addition becomes N, the remainder after the modulo operation becomes 1. This serves as the offset for the next row and accumulates with other remainders as the generator progresses. After adjusting the formula in equation 3.15, the new formula now looks as follows:

$$\mathcal{P}_{i+1} = \mathcal{P}_i + column\_size \mod (N-1) \tag{3.16}$$

The permutation vector that this formula generates, however, is incomplete. This is because $N-1$, which is supposed to be the final value in the permutation, becomes a 0 duo to the modulo operation and ends the cycle one element short. It is therefore necessary to pause the sequence and hard-code the final address at the end of every sequence. Luckily this address is always the same regardless of the configuration of the interleaver. The MATLAB code that demonstrates this formula can be seen in Listing 3.3.

Listing 3.3: MATLAB code that generate addresses recursively

```
1  frame_size = 12;
2  column_count = 3;
3  column_size = frame_size / column_count;
4  input = 0:(frame_size-1); %neutral element
5
6  out = zeros(1,frame_size);
7  index = 0;
8  for i = 0:frame_size-1
9      out(i+1) = input(index+1);
10     index = mod((index + column_size) , frame_size-1) ;
11 end
12 out(frame_size) = input(frame_size);
13
14 disp(input)
15 disp(out)
```

This recursive calculation consists of two operations, a sum operation and a modulo operation. The modulo operation can be simplified when taking into account the fact that the input range is always constrained to a certain interval. Consider the following case where

$\mathcal{P}_i \in [0, m-1]$ and $column\_size \in [0, m-1]$, in which m stands for the modulo value. The sum of both lies in the interval of $[0, 2(m-1)]$. A modulo operation can be realised as a conditional subtraction with m if the input is within the interval of $[0, 2m-1]$. Because the interval of the sum is a subset of the simplified modulo interval, it is possible to apply this simplification. The result of the subtraction of m from the sum would result in $\mathcal{P}_{i+1}$ ending up in the interval of $[0, m-2]$, which is a subset of the original input $\mathcal{P}_i$, meaning that the maximum value that the simplified modulo can accept, is never exceeded in all subsequent iterations of this formula. Keep in mind that the value of m is defined as N-1.

The next simplification that can be achieved is by combining the addition and conditional subtraction into a single operation. Instead of checking whether or not the sum is greater then or equal to modulo m, the input is compared with a value μ which is defined as:

$$\mu = m - column\_size \tag{3.17}$$
$$= N - 1 - column\_size \tag{3.18}$$

If the value of $\mathcal{P}_i$ is smaller then μ, then the result of the sum will be smaller then m. If $\mathcal{P}_i$, on the other hand, is greater than or equal to μ, then an addition with $column\_size$ followed by a subtraction of m is concatenated into a single subtraction of the value μ. In other words, depending on whether or not the input value is smaller then μ, the input will receive either an addition with the $column\_size$ value or a subtraction with the μ value. A block diagram of this construction can be seen in Figure 3.3 where the ADD register contains the value $column\_size$ and the SUB register contains the same value as μ that is defined in equation 3.17. Figure 3.4 gives an overview of the two different intervals and the arithmetic operation being done the each interval.

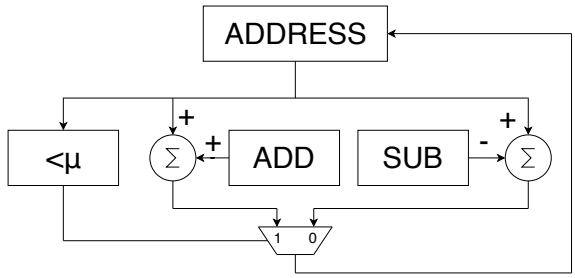

Figure 3.3: block diagram of the simplified structure

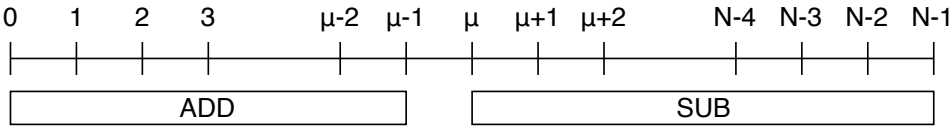

Figure 3.4: representation of the input range divided in two regions

As stated earlier, this generator can not generate the final address in the permutation vector that is needed to access the final element. Consider the simple example in the case

of 8 PSK modulation, for which the permutation vector is defined as equation 3.12. As it stands now, the way that the current LCG based generator creates the addresses is depicted in Figure 3.5 where the full lines represent the generated sequence. In this case the second

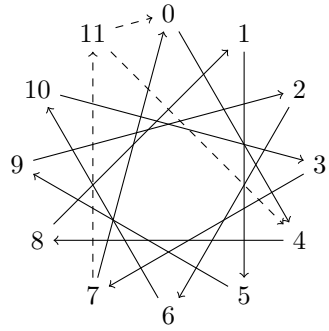

Figure 3.5: example of a sequences created by an LCG based generator

to last element, that is generated before going back to zero, is 7. In the ideal case it would have gone from node 7 to node 11 before going back to node 0 as depicted by the dotted lines in Figure 3.5. In order to achieve this, an intervention at the end of the cycle needs to happen. This can be done by applying the following trick. The subtraction is still done by μ, but the input will not be compared with μ anymore but with $\mu + 1$. The address μ is the final address that is generated in the sequence and goes to 0 in the next iteration duo to it being subtracted with μ itself. By offsetting the compare value by one, the address that is currently μ will be summed with the $column\_size$ value and become $N - 1$, which is the modulo value and the correct final address. Figure 3.6 shows how the intervals of the summation and subtraction have shifted.

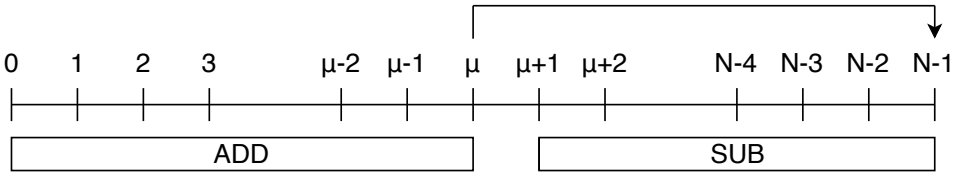

Figure 3.6: representation of the input range divided in two regions after modification

The trouble now is that address 0 will be skipped. This is because the address that is now equal to $N - 1$, will be subjected to a subtraction of μ, which will result in the second element of the permutation. By comparing the current address value with $N - 1$ it is possible to reset the address on that condition and force it to 0 to restart the sequence.

This design is much more simpler then the design suggested in section 3.2.3 because it contains no FSM. The address is generated deterministically based on the previous value. Because of this, the choice has been made to go forward with this design. Simplicity of course is not the only consideration, later on it will become clear why this design is so powerful.

3.3.4 expanding from examples to real use cases

All the above designs have been explained using small examples. What follows now is MATLAB code that proves these concepts on the same scale of the application with real input sizes and real interleaver configurations. Each frame will be accompanied by two data fields. The first one is $frame\_select$, it will tell if the interleaver is dealing with a normal frame or a small frame. The second one is the $APSK\_select$, it tells the interleaver what modulation scheme will be used for the transmission of this frame. The column configuration of the interleaver depends on what modulation scheme sill be used as explained in section 2.3.2. A neutral element is is interleaved using the first high level model and compared to the two lower level models for all valid configurations. The exception is the 128APSK that needs to be verified separately in a separate MATLAB script because of the added zero padding that increases the input size to 64806 bits. The full MATLAB test script can be seen in Listing 3.4

Listing 3.4: MATLAB script that compares all the modles

```
close all; clear all; clc
max_disp_size = 9;
%% parameters
frame_select = 0;
    % 0 -> 64800 bits
    % 1 -> 16200 bits
APSK_select = 0;
    % 0 -> 8 PSK
    % 1 -> 16 APSK
    % 2 -> 32 APSK
    % 3 -> 64 APSK
    % 4 -> 128 APSK
    % 5 -> 256 APSK
%% constants
frame_size = [64800, 16200]; % 20 bit wide input
column_count = [3,4,5,6,7,8];
column_size = [ [21600, 16200, 12960, 10800, 0, 8100],
                [5400, 4050, 3240, 0, 0, 0 ] ];
frame_size = frame_size(frame_select+1);
column_count = column_count(APSK_select+1);
column_size = column_size(frame_select+1,APSK_select+1);
input = 0:(frame_size-1); %neutral element
%% High level model of the block interleaver using matrix
    transposition.
Interleaver_A = reshape(input,[],column_count);
Interleaver_B = Interleaver_A';
ideal_permutation_vector = reshape(Interleaver_B,1,[]);
disp("ideal_permutation_vector");
disp(ideal_permutation_vector(1:max_disp_size))
disp(ideal_permutation_vector(frame_size-max_disp_size+1:
    frame_size))
%% for loop generation
loop_permutation_vector = zeros(1,frame_size);
```

```
32 for i=0:(frame_size-1)
33     index = column_size * mod(i,column_count) + floor(i/
           column_count);
34    loop_permutation_vector(i+1) =  input(index+1);
35 end
36 disp("loop_permutation_vector. Equal to ideal: "+isequal(
       ideal_permutation_vector,loop_permutation_vector));
37 disp(loop_permutation_vector(1:max_disp_size))
38 disp(loop_permutation_vector(frame_size-max_disp_size+1:frame_size
       ))
39 %% Linear Congruent Generator (LCG) permuation model
40 LCG_permutation_vector = zeros(1,frame_size);
41 index = 0;
42 for i = 0:frame_size-1
43     LCG_permutation_vector(i+1) = input(index+1);
44     index = mod((index + column_size) , frame_size-1) ;
45 end
46 LCG_permutation_vector(frame_size) = input(frame_size);
47 disp("LCG_permutation_vector. Equal to ideal: "+isequal(
       ideal_permutation_vector,LCG_permutation_vector));
48 disp(LCG_permutation_vector(1:min(max_disp_size,frame_size)))
49 disp(LCG_permutation_vector(frame_size-max_disp_size+1:frame_size)
       )
```

3.4 block-interleaver with only one memory block

All the design so far have assumed that a double memory buffer is used. Such a configuration can be seen in Figure 2.6. With this configuration it is possible to interleave subsequent frames under different configurations. A frame can be read from one block with one configuration, afterwards the address generator switch to another set of register values using a LUT, and then reads the other frame in the other block under a different configuration. High performance interleaving has always being done by alternating two memory blocks, until now.

In this Master's thesis an attempt is made at creating an interleaver that uses one memory block for the interleaving process without sacrificing all the capabilities of a double memory block interleaver.

3.4.1 One-memory-block design explained

The inspiration of using only one memory block came from reading the Stratix 10 memory documentation [14]. As explained in section 2.5.3, the M20K modulo is capable of doing a concurrent read and write to the same address. The idea is to read out a piece of the current frame, while at the same time writing a piece of the new frame to the same address. In a larger sense, current frames are constantly swapped out with new frames, as if it was a giant shift register. As a demonstration, a 12 bit frame will be interleaved using a 4 by 3 block configuration for 8 PSK modulation. The contents of the interleaver in each step can

be seen in Figure 3.7. At first, the interleaver memory block is empty, this is visualised

step 0	step 1	step 2	step 3	step 4	step 5
$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 4 & 8 \\ 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 11 \end{bmatrix}$	$\begin{bmatrix} 0 & 3 & 6 \\ 9 & 1 & 4 \\ 7 & 10 & 2 \\ 5 & 8 & 11 \end{bmatrix}$	$\begin{bmatrix} 0 & 9 & 7 \\ 5 & 3 & 1 \\ 10 & 8 & 6 \\ 4 & 2 & 11 \end{bmatrix}$	$\begin{bmatrix} 0 & 5 & 10 \\ 4 & 9 & 3 \\ 8 & 2 & 7 \\ 1 & 6 & 11 \end{bmatrix}$
empty	frame 1	frame 2	frame 3	frame 4	frame 5

Figure 3.7: interleaver contents of example

at step 0 with the memory block initialised to zero. The first frame, which is represented by a neutral element, is written sequentially into the memory. (Please note that the linear addressing of the matrix, in a sense what element is selected by a given address, coincides with the way that the neutral element is written to in step 1. In other words, accessing address 5 would return 5 at step 1.) The correct permutation vector to read and interleave frame 1 is given by equation 3.12. At the same time that frame 1 is read from memory, frame 2 will be stored to memory. Elements 0, 4 and 8 in the first row at step 1 are read, while at the same time, elements 0, 1 and 2 of frame 2 are stored in their place. The same goes for all the remaining elements of frames 1 and 2. Frame 1 is interleaved and frame 2 is now stored in memory, as can be seen in Figure 3.7 at step 2. Frame 2 is interleaved by reading addresses number 0, 5,and 10, which will access all the elements of the first row, while at the same time, elements 0, 1 and 2 of frame 3 are stored in their place. The same goes for all the remaining elements of frames 2 and 3. This cycle would ideally repeat indefinitely. In this case it does because reading out frame 5 in step 5, in the sequence of the unity permutation $\mathbb{1}$, while storing the next frame, would put the interleaver contents in the same state as step 1. There are some challenges that need to be overcome in order for this idea to work. The big question is: What permutation vector is needed to read out frame 2. The bigger question is: What permutation vector is needed to read out all the subsequent frames. The biggest question is: how would one generate these permutation vectors?

It has been confirmed that when writing a frame to memory in the sequence of $\mathbb{1}$, which is the unity permutation or in this case linear addressing, the frame can be interleaved by reading it in the sequence of $\mathcal{P}$. When a frame that has been written in the sequence of $\mathcal{P}$, like frame 2 in step 2, through experimentation with MATLAB it has been found that, it can be read and interleaved by reading it in the sequence of $\mathcal{P} \times \mathcal{P} = \mathcal{P}^2$. Frames that are written in a sequence of $\mathcal{P}^2$ can be correctly interleaved by reading them in the sequence of $\mathcal{P}^3$. In general frames can be interleaved by writing them in the sequence of $\mathcal{P}^n$ and reading them in the sequence of $\mathcal{P}^{n+1}$

3.4.2 permutation groups

The permutation vectors needed to read all the subsequent frames are listed below.

$$\mathbb{1} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \end{bmatrix} \tag{3.19}$$

$$\mathcal{P} = \begin{bmatrix} 0 & 4 & 8 & 1 & 5 & 9 & 2 & 6 & 10 & 3 & 7 & 11 \end{bmatrix} \tag{3.20}$$

$$\mathcal{P}^2 = \begin{bmatrix} 0 & 5 & 10 & 4 & 9 & 3 & 8 & 2 & 7 & 1 & 6 & 11 \end{bmatrix} \tag{3.21}$$

$$\mathcal{P}^3 = \begin{bmatrix} 0 & 9 & 7 & 5 & 3 & 1 & 10 & 8 & 6 & 4 & 2 & 11 \end{bmatrix} \tag{3.22}$$

$$\mathcal{P}^4 = \begin{bmatrix} 0 & 3 & 6 & 9 & 1 & 4 & 7 & 10 & 2 & 5 & 8 & 11 \end{bmatrix} \tag{3.23}$$

$$\mathcal{P}^5 = \mathbb{1} \tag{3.24}$$

$$\mathcal{P}^n = \mathcal{P}^{n-5} \qquad n \in [5, \infty[\tag{3.25}$$

What has been stumbled upon is actually permutation group with a neutral element $\mathbb{1}$ and a seed element $\mathcal{P}$. As empirical evidence, one could manually read out the contents of the interleavers in Figure 3.7 by using the above mentioned permutation vector in order to interleave the individual frames and check the validity for themselves. Through experimentation with MATLAB it has been found that all the permutations in and of themselves can be generated using the same structure as described in section 3.3.3. The generator would now look as follows:

$$\mathcal{P}^n_{i+1} = \mathcal{P}^n_i + b(n) \mod (N-1) \tag{3.26}$$

$$b(n) = [1, 4, 5, 9, 3] \tag{3.27}$$

Vector b in equation 3.27 is a look up table that stores all the second elements of all the permutations in a LUT. The address generator that can be used is the same as the one depicted in Figure 3.3. If the current permutation vector is $\mathcal{P}^n$ then in order to generate vector $\mathcal{P}^{n+1}$, $b(n+1)$ is put into the ADD register, $N-1-b(n+1)$ is put into the SUB register, $N-1-b(n+1)+1$ is put into the μ register, and the address is reset to zero.

3.4.3 challenge in frame size scalability

The previous example has demonstrated the possibility of using the concurrent read and write operations, to interleaver subsequent frames with the use of only one memory block. The theory has been proven to work with small frames, where all the parameters needed to generate the full permutation group, can be stored in a small LUT. For larger frames, however, the permutation groups become larger, which in turn means that the LUT would have to store a significantly larger amount of parameters.

Because every permutation can be generated from the second element, it can therefore be used as a unique key, that identifies the full permutation vector. It is therefore possible to count all the permutation vectors in a group by comparing only the second elements. Counting all the elements is done by preforming permutations until the neutral element is encountered, which has a second element of 1. All the permutation group sizes for all the valid interleaver configurations can be seen in Table 3.1. They have been counted using the MATLAB code found in Listing: 3.5

Table 3.1: permutation group element count for each interleaver configuration

Modulation	8PSK	16APSK	32APSK	64APSK	128APSK	256APSK
column size	3	4	5	6	7	8
normal	27767	6941	27767	9255	(995)	4627
small	3983	1991	7967	N/A	N/A	N/A

Listing 3.5: MATLAB script that counts all the elements of all the used permutation groups

```
1  close all; clear all; clc
2  max_disp_size = 9;
3  %% parameters
4  frame_select = 0;
5      % 0 -> 64800 bits,  1 -> 16200 bits
6  APSK_select = 1;
7      % 0-> bypass      1 -> 8 PSK
8      % 2 -> 16 APSK    3 -> 32 APSK
9      % 4 -> 64 APSK    5 -> 128 APSK
10     % 6 -> 256 APSK
11 %% constants
12 frame_size = [64800, 16200]; % 64806 in case of 128APSK
13 column_count = [3,4,5,6,7,8];
14 column_size = [ [21600, 16200, 12960, 10800, 0, 8100],
15                 [5400,  4050,  3240,  0,     0, 0   ] ];
16 frame_size = frame_size(frame_select+1);
17 column_count = column_count(APSK_select+1);
18 column_size = column_size(frame_select+1,APSK_select+1);
19 input = 0:(frame_size-1); %neutral element
20 %% High level model of the block interleaver using matrix
       transposition.
21 Interleaver_A = reshape(input,[],column_count);
22 Interleaver_B = Interleaver_A';
23 ideal_permutation_vector = reshape(Interleaver_B,1,[]);
24 disp("ideal_permutation_vector");
25 disp(ideal_permutation_vector(1:max_disp_size))
26 %% counting the permutation group elements
27 count = 0;
28 element = ideal_permutation_vector;
29 while(element(2) ~= 1)
30     element=element(ideal_permutation_vector+1);
31     count = count + 1;
32 end
33 disp(element(1:max_disp_size))
34 disp(count);
```

To do uninterrupted interleaving for all configurations, a total of 91293 parameters of 16 bit need to be saved in a LUT, which comes down to 1460688 bits of data, which is around the size of 22.54 normal frame. Storing this much data would defeat the initial purpose of cutting down on memory usage.

3.4.4 possible compromise for large frames

One way that this design can be salvaged is by breaking off after the first few iterations and reset the interleaver. This will slightly lower the maximum throughput but would otherwise make for an excellent low cost interleaver that can handle data in bursts.

It takes a full time-frame for a frame to be both written to and read from memory. When only one frame is interleaved, the time efficiency drops to 50% as only one frame is processed every two time-frames. Traditional block-interleavers would therefor use two memory block in order to a achieve a full throughput. Interleaving consecutive frames using the above mentioned design will net a higher time efficiency because it takes $n + 1$ time-frames to process n frames. The formula for efficiency can be seen in equation 3.28 along with the graph of the first 16 values of n in Figure 3.8.

$$\eta = \frac{n}{n+1} \tag{3.28}$$

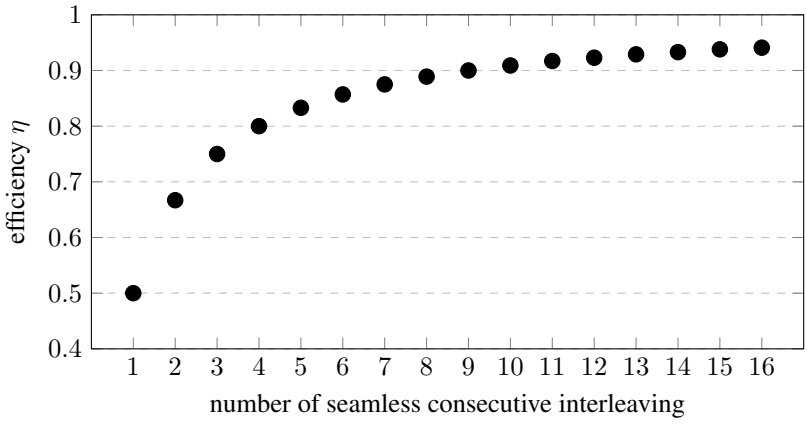

Figure 3.8: interleaver time efficiency

The efficiency has a diminishing return, but it is possible to achieve a maximum throughput of 94.1% with only 16 values for each configuration. In order for this interleaver to work continuously, however, a buffer at the input and output is needed to smooth out the inconsistencies duo to the need for a periodic reset. This again defeats the purpose of cutting down on memory usage, if it means that multiple buffers must be introduced.

3.4.5 solving the frame size scalability issue

At first glance there seems to be no structure in all the second elements of all the permutation that can be exploited. This is because the structure is obfuscated by the modulo operator that wraps around all the elements, making it look like everything is random. In order to see the structure one would have to unwrap all the elements of the permutation vector by undoing the modulo operation. Consider the following case where a 12 bit frame

is interleaved for 8 PSK modulation. Figure 3.9 gives a representation of the memory contents. The horizontal axis represents the frame elements and the vertical axis represents the memory location these elements are stored at. The current situation of the interleaver is comparable to step 1 in Figure 3.7.

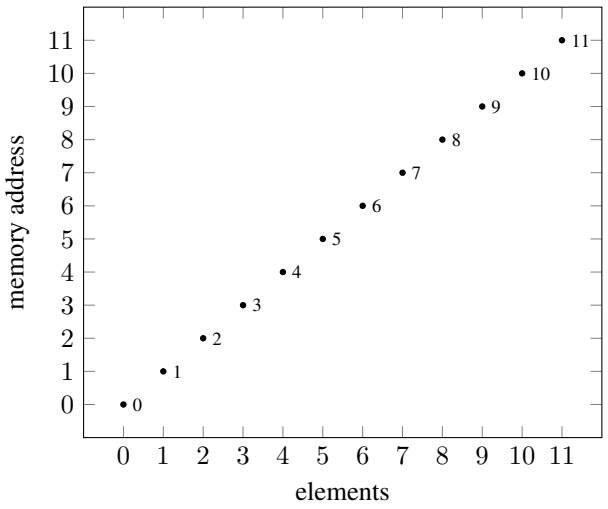

Figure 3.9: initial memory contents

Frame 1 is read from and frame 2 is written to memory using the permutation vector $\mathcal{P}$ in Equation 3.20. The result of this action can be seen in Figure 3.10 and is comparable to step 2 of Figure 3.7.

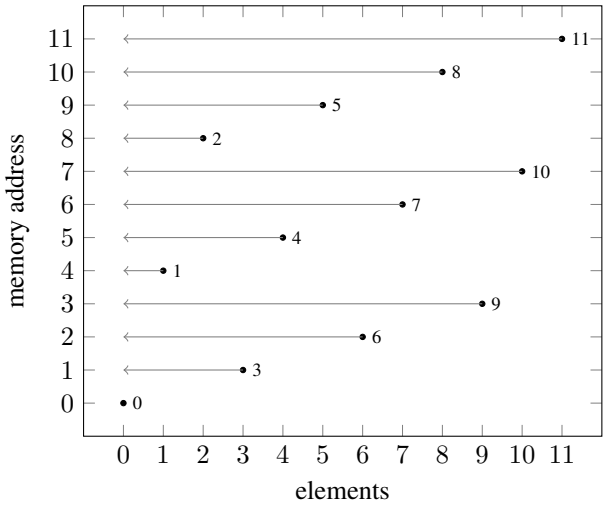

Figure 3.10: memory contents after the first frame is read and the second frame is written

The address space of the interleaver will now be unwrapped and projected onto a virtual address space. The relation between the two is that taking the modulo of the virtual adders space will give back the real address space. This virtual address space can be seen in Figure 3.11 on the left vertical axis while the real address space can be seen on the right vertical axis. Notice how all the elements of the next frame are ordered in the sequence of permutation $\mathcal{P}$.

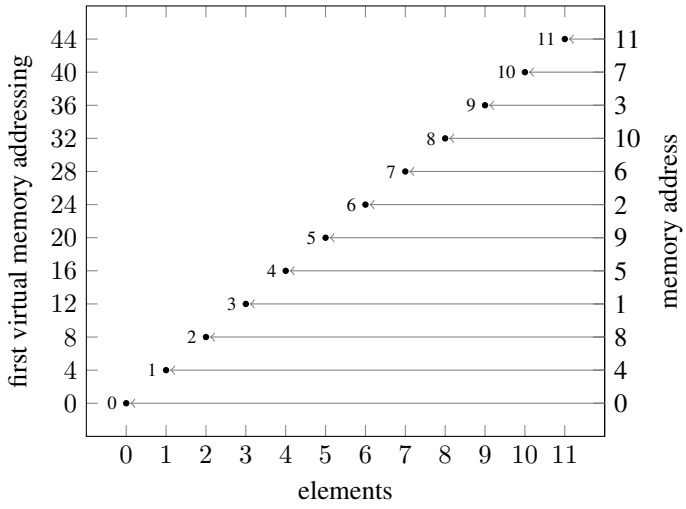

Figure 3.11: unwrapping the memory contents with the second frame inside of it

It is possible to conclude that all the consecutive elements of a frame are now spaced out in steps of 4, which is the $column\_size$ value. This comes from the fact that the generator had to traverse the memory in steps of 4 to iterate over each row in the matrix representation. Because the permutation takes elements in steps of 4 and all the elements are spaced out in steps of 4 in the virtual address space, the generator now has to traverse the virtual address space in steps of 4 by 4, which comes down to 16, in order to achieve the interleaving of the second frame. Through the modulo operation it is possible to achieve the same result by traversing the real address space in steps of 5 as it is the modulo 11 of 16.

The contents of the interleaver after reading out frame 2 and writing frame 3 to memory, are depicted in Figure 3.12 and is comparable to step 3 of Figure 3.7. Again, the first virtual address space is unwrapped and projected onto a second virtual adders space, which can be seen in Figure 3.13 on the left vertical axis. The first virtual address space along with the real address space can be seen on the two right vertical axes. Notice how all the elements of the next frame are ordered in the sequence of permutation $\mathcal{P}^2$.

In general, the permutation simply expands the distance between the elements by a factor of $column\_size$. The modulo operations confines the address space and serves as a projection from one address space to the next. Using this knowledge it is now possible to predict how the next permutation vector looks like based on the current permutation vector. The second element of each permutation vector, which is used for the generation

of the whole vector, can be constructed by using the formula in Equation 3.29.

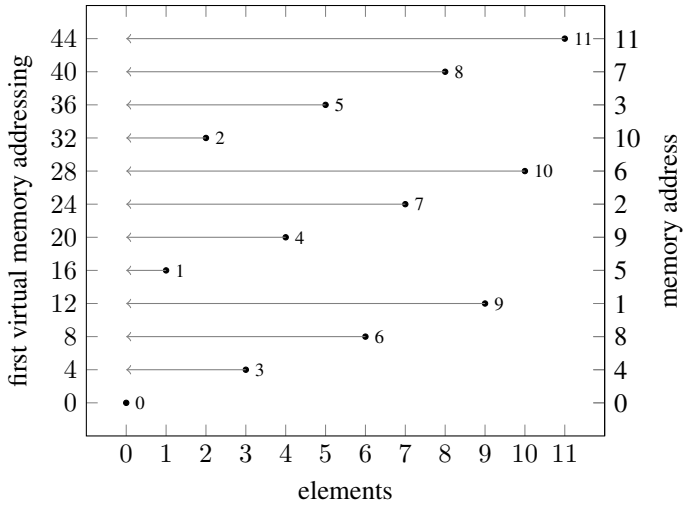

Figure 3.12: memory contents after the second frame is read and the third frame is written

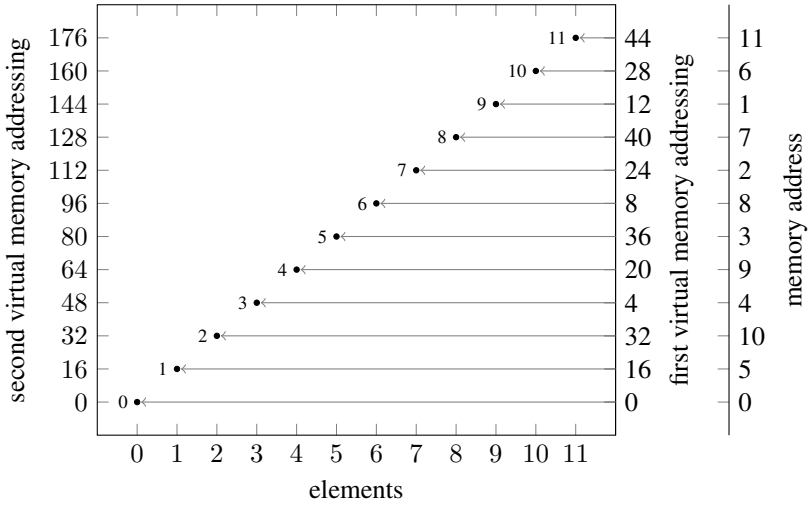

Figure 3.13: unwrapping the memory contents with the third frame inside of it

$$\mathcal{P}_2^n = column\_size^n \mod (N-1) \tag{3.29}$$
$$= b(n) \tag{3.30}$$

3.4.6 uninterrupted interleaving for one configuration

Calculating the next ADD value, which is the second element of the next permutation, is not easy. Expensive hardware is needed to preform a 32-bit (16×16) multiplication followed by a modulo operation. The good news is that this value needs to be calculated only ounce for every frame. This allows for area efficient solution that can use multiple pipeline stages or multicycle paths.

A major simplification can be made, however, if the multiplication is broken up into a series of additions. Doing this for Equation 3.29 gives the following result:

$$\mathcal{P}_2^{n+1} = column\_size^{n+1} \mod (N-1) \tag{3.31}$$

$$= \sum_{i=0}^{column\_size} column\_size^{n} \mod (N-1) \tag{3.32}$$

$$= \sum_{i=0}^{column\_size} \mathcal{P}_2^{n} \mod (N-1) \tag{3.33}$$

The operation done is Equation 3.33 can be done by using the same structure as discussed in subsection 3.3.3. This means that interleaving at full throughput is possible by using only one memory block and two LCG-based generators.

As simple as this design already is, it can be simplified down even further. This is because the generator that generates all the second elements for all the permutations, and the generator that generates all the elements of each permutations from these second elements, can be unified into one generator. Consider the following formula, which is a rewrite of Equation 3.16.

$$\mathcal{P}_{i+1}^{n} = \mathcal{P}_i^{n} + \mathcal{P}_2^{n} \mod (N-1) \tag{3.34}$$

The k^{th} element in the permutation vector $\mathcal{P}^n$ can be redefined as:

$$\mathcal{P}_k^{n} = k \times \mathcal{P}_2^{n} \mod (N-1) \tag{3.35}$$

The multiplication in equation 3.35 can be expanded as a sumation of $\mathcal{P}_2^n$

$$\mathcal{P}_k^{n} = \sum_{i=0}^{k} \mathcal{P}_2^{n} \mod (N-1) \tag{3.36}$$

By extension the $column\_size^{th}$ element will be defined as follows:

$$\mathcal{P}_{column\_size}^{n} = \sum_{i=0}^{column\_size} \mathcal{P}_2^{n} \mod (N-1) \tag{3.37}$$

Substitution of Equation 3.37 in Equation 3.33 gives the following result:

$$\mathcal{P}_2^{n+1} = \mathcal{P}_{column\_size}^{n} \tag{3.38}$$

There is no need to have a second generator as the same calculation is already preformed by the address generator. The only addition would be a counter that counts from 1 to the current $column\_size$ value and captures the $column\_size^{th}$ address. Subsequently, the next ADD, SUB and μ register values will be updated using the captured values according to their corresponding formula as described in section 3.3.3. With this design it is now possible and practical to seamlessly interleave frames at full throughput by using only one memory block instead of two.

Another way to explain it is that the second element of permutation $\mathcal{P}$ from Equation 3.20 is 4. When preforming $\mathcal{P} \times \mathcal{P}$, the 5^{th} element of $\mathcal{P}$ is taken and it becomes the second element in the result. When preforming $\mathcal{P}^2 \times \mathcal{P}$, then again, the 5^{th} element of $\mathcal{P}^2$ is taken and it becomes the second element in the result. This behaviour can be verified in Equations 3.20 through 3.23.

This generator iterates through all the permutations in the permutation group. According to Table 3.1, the configurations that employ the biggest permutation groups are that of normal frames for 8 PSK and 32 APSK. Both contain 27767 different permutation and each permutation contains 64800 elements. The size of the longest loop that is perpetually generated is $\mathcal{L} = 27767 \times 64800 = 1799301600$. At 350 MHz it would take roughly 5.14 second to iterate through all the addresses of all permutations.

Even though this is a good design, it is not up to the challenge as it can not switch interleaver configurations on the fly. When for example, an 8 PSK frame is followed by a 16 APSK frame, the interleaver has to read out the current frame and reconfigure itself for the next frame.

3.4.7 uninterrupted interleaving across all configurations

In order to solve this problem, consider going back to Figure 3.11. The first frame was interleaved using a 3 column configuration, this means that the current frame is also stored according to the same configuration. The current frame, however, needs to be interleaved using a 4 column configuration. In a 4 column configuration the step size between elements is 3, this is in order to traverse the rows in the matrix representation. Because the distance between the elements in the virtual address space is 4 and the generator has to take the elements in steps of 3, the generator would have to traverse the virtual adders space in steps of 4 by 3, which is 12. Doing this will interleaver the current frame and write the next frame in its place, this can be seen in Figure 3.14. The first virtual address space is unwrapped and projected onto a third virtual adders space, which can be seen in Figure 3.15 on the left vertical axis. The first virtual address space along with the real address space can be seen on the two right vertical axes. Notice how all the elements of the next frame are ordered in the sequence of permutation $\mathcal{P}_{8PSK} \times \mathcal{P}_{16APSK}$. The fact that it becomes a unity permutation will be attributed to coincidence as it has not been studied any further.

In general, when the current frame was written to memory under a previous configuration in the sequence of permutation ${}^{old}\mathcal{P}^n$, it is possible to interleave the current frame by reading it in the sequence of permutation ${}^{old}\mathcal{P}^n \times^{new_1} \mathcal{P}$. When the next frame is written in sequence of permutation ${}^{old}\mathcal{P}^n \times^{new_1} \mathcal{P}$, it is possible it interleaver it under another configuration by reading it in the sequence of ${}^{old}\mathcal{P}^n \times^{new_1} \mathcal{P} \times^{new_2} \mathcal{P}$

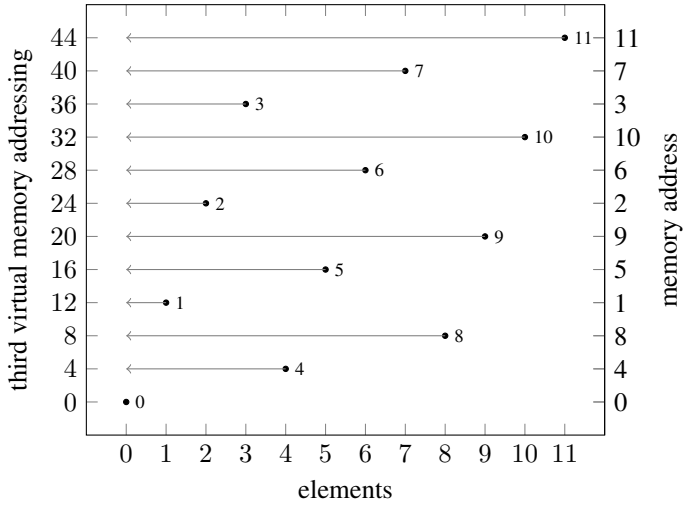

Figure 3.14: memory contents after the first frame is read and the second frame is written under a different configuration

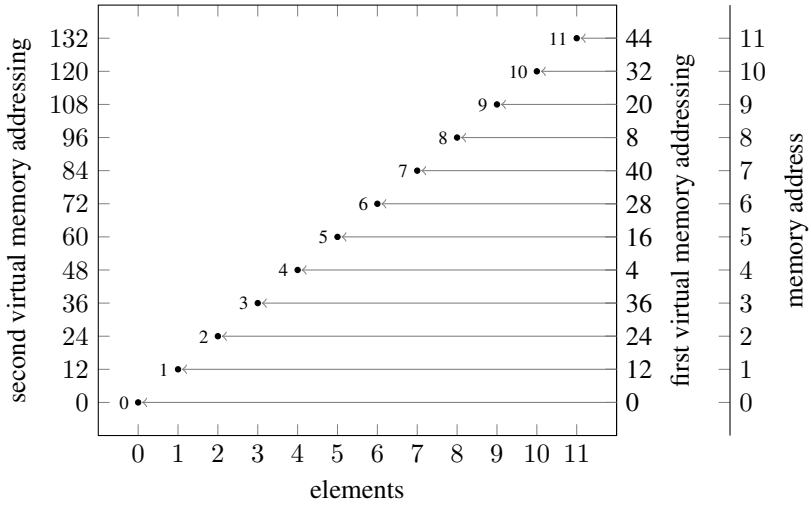

Figure 3.15: unwrapping the memory contents with the second frame inside of it under a different configuration

A more general form of Equations 3.31 through 3.33 can be found below:

$$({}^{old}\mathcal{P}^n \times {}^{new}\mathcal{P})_2 = {}^{old}\mathcal{P}_2^n \times {}^{new}\mathcal{P}_2 \mod (N-1) \tag{3.39}$$

$$= column\_size_{old}^n \times column\_size_{new} \mod (N-1) \tag{3.40}$$

$$= \sum_{i=0}^{column\_size_{new}} column\_size_{old}^n \mod (N-1) \tag{3.41}$$

$$= \sum_{i=0}^{column\_size_{new}} {}^{old}\mathcal{P}_2^n \mod (N-1) \tag{3.42}$$

A more general form of equation 3.37 is the following:

$${}^{old}\mathcal{P}_k^n = \sum_{i=0}^{k} {}^{old}\mathcal{P}_2^n \mod (N-1) \tag{3.43}$$

By extension the $column\_size_{new}{}^{th}$ element will be defined as follows:

$${}^{old}\mathcal{P}^n_{column\_size^{new}} = \sum_{i=0}^{column\_size_{new}} {}^{old}\mathcal{P}_2^n \mod (N-1) \tag{3.44}$$

Substitution of Equation 3.44 in Equation 3.42 gives the following result:

$$({}^{old}\mathcal{P}^n \times {}^{new}\mathcal{P})_2 = {}^{old}\mathcal{P}^n_{column\_size^{new}} \tag{3.45}$$

When the interleaver configuration data is clocked in along with the first bit of the frame, the counter would counts up to the threshold value set by the configuration. When the frame is fully written, the interleaver will configure itself according to the captured value and read out the frame according to the new configuration. Meanwhile, the next frame will set its own threshold value in order to be interleaved in the next iteration.

This new system does not iterate through a single permutation group, but through a direct product of multiple permutation groups. Each next permutation is deterministically calculated based on the needed configuration and the current permutation vector.

With this design it is now possible to seamlessly interleave a stream of frames under a mix of different configuration. The only limitation of this design is that it can not handle a stream of frames with varying frame sizes. This is not much of a problem as it is tolerated to have some delay in switching between frame sizes in this project. The zero padding introduced by the 128 APSK will, however, introduce some problems for this structure, this problem will be solved later on.

3.4.8 scalability of the one-memory-block design

The block interleaver consists of two main components, the memory block and the address generator. The area usage of full-adders, comparators, and registers scale $\log_2(O)$ in proportion to the input size. Because the address generator consists of only full-adders, comparators, and registers, it will scale similarly with respect to the frame size. The memory block on the other hand, scales linearly with respect to the frame size. This makes

the cost of the address generator negligible compared to the cost of the memory block for larger frame sizes.

Because of the simplicity of the address generator, it can generate addresses at higher frequencies. The combinatorial path from the address generator and the data to the memory block is shorter. This is because it does away with a full memory block and the (de-)multiplexers that come along with it. This implies that less memory-access-pipeline-stages are needed to achieve higher frequencies.

3.5 de-interleaving

De-interleaving is done by writing the frame to memory in the sequence of the permutation vector $\mathcal{P}^{-n}$ and reading it out of memory in the sequence of the permutation vector $\mathcal{P}^{-n-1}$. The permutation group that is used for interleaving is a cyclic group, this means that every permutation has an inverse permutation that belongs to the same group.

Everything that has been established for the interleaver so far is the result of the properties of the permutation group. The de-interleaver uses the same permutation group. This means that everything that is established so far for the interleaver, is also applicable to the de-interleaver, and thus the same hardware structure can be used for the de-interleaving operation.

The interleaver increments the current address by the $column\_size$ value. The de-interleaver, on the other hand, increments the current address by the $column\_count$ value. The counter treshold is also set at the $column\_count$ value. For example, if the de-interleaver is configured for 5 columns, then the ADD value and counter threshold are both set to 5.

An interesting hypothesis that arises from this is that it would be possible to process a mixed stream of frames that are destined for both interleaving and de-interleaving under different configurations and still use only one memory block. This would have been an interesting study if more time was given to do the Master's Thesis.

3.6 processing multiple sequential bits at a time

In order to archive high throughput, multiple sequential bits of a frame can be clocked-in for interleaving. It is necessary to exploit some properties of the transposition operation in order to expand the interleaver design in this direction.

3.6.1 subdividing the transposition operation

Transposing a matrix can be done explicitly by swapping out bits from the upper triangle part of the matrix with the lower triangle part of the matrix. Another option is to do it recursively. First, the matrix is divided up into smaller submatrices, as can be seen in Equation 3.46. Then, the submatrices from the upper triangle part of the matrix are swapped with the submatrices of the lower triangle part of the matrix. Finally, all the submatrices themselves are to be transposed. The result can be seen in Equation 3.47.

$$A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \dots & A_{mn} \end{bmatrix} \tag{3.46}$$

$$A^T = \begin{bmatrix} A_{11}^T & A_{21}^T & \dots & A_{m1}^T \\ A_{12}^T & A_{22}^T & \dots & A_{m2}^T \\ \vdots & \vdots & \ddots & \vdots \\ A_{1n}^T & A_{2n}^T & \dots & A_{nm}^T \end{bmatrix} \tag{3.47}$$

3.6.2 subdividing the interleaver

In order for the interleaver to accept multiple consecutive bits, it needs to be broken up into two parts. The first part is the big interleaver. It takes in words that consist of consecutive bits and interleaves the words, without touching the contents, the same way it would interleave individual bits. The second part is the small interleaver. It takes in a small number of interleaved words and interleaves the individual bits inside these words. The small interleaver would have to interleave multiple groups of words before a full frame is eventually interleaved.

For example, Figure 3.16 shows a block interleaver of a normal frame that is destined for 64 APSK modulation. A normal frame is 64800 bit wide and thus consist of 4050 16-bit words. Consequently the address space N for the big interleaver becomes 4050 and the column size value becomes 675. The permutation addresses are generated the same way as before, but now with smaller numbers. In this example the words have been sequentially written to memory. In order to achieve an intermediate form of interleaving, the words need to be read out in the sequence of the generated permutation.

As can be observed in Figure 3.16, the words of consecutive bit are actually pieces of a column. Before multiple bits were packed into words, reading out a full row from the interleaver would give a vector that has the same size as the $column\_count$, that can be taken directly to the constellation mapper. Now with consecutive bits packed into words, which represent pieces of a column, reading out a row from the big-interleaver would give back a matrix that has the dimension of $word\_size$ by $column\_count$. This matrix contains a $word\_size$ amount of symbols, but, because the matrix is read from the big interleaver in columns, the small interleaver would have to write these columns into its own memory, before it is able to extract the rows, which can then be directly fed to the constellation mapper. In Figure 3.17, the contents of the small interleaver can be seen after after the first row has been written from the big interleaver to the small interleaver.

In order to keep up with the throughput, multiple rows would have to be read out at the same time. This amount depends on the $column\_count$ value. As an example, a 3 column block-interleaver for 8 PSK modulation is taken. In order to sustain a 16 bit throughput, the small interleaver has to output $16/3 = 5.333$ rows each clock cycle. This can be achieved by reading out rows in a 5-5-6 pattern. This means that 5 rows will be read on the first two clock cycles while 6 rows will be read on the third clock cycle. This will

come down to an average throughput of $(5 + 5 + 6)/3 = 5.333$ symbols. All the different row-read-count patters for all the valid configurations can be seen in Table 3.2.

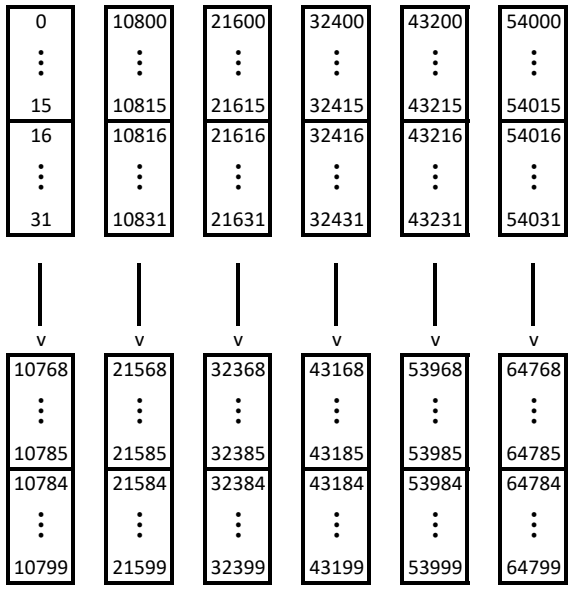

Figure 3.16: block interleaver in a 6 column configuration for normal frames

0	10800	21600	32400	43200	54000
1	10801	21601	32401	43201	54001
2	10802	21602	32402	43202	54002
3	10803	21603	32403	43203	54003
4	10804	21604	32404	43204	54004
5	10805	21605	32405	43205	54005
6	10806	21606	32406	43206	54006
7	10807	21607	32407	43207	54007
8	10808	21608	32408	43208	54008
9	10809	21609	32409	43209	54009
10	10810	21610	32410	43210	54010
11	10811	21611	32411	43211	54011
12	10812	21612	32412	43212	54012
13	10813	21613	32413	43213	54013
14	10814	21614	32414	43214	54014
15	10815	21615	32415	43215	54015

Figure 3.17: first row of the big-interlear inside the small-interleaver

Table 3.2: row-read-count pattern for all column sizes

column size	3	4	5	6	7	8
symbol rate	5.333	4	3.2	2.666	2.286..	2
pattern	5-5-6	4	3-3-3-3-4	2-3-3	2-2-2-2-2-3-3	2

The trick of doing a concurrent read and write will not work for the small interleaver, it will therefore be a dual memory block configuration, where the memory blocks alternate between writing full columns and reading multiple rows. In order to achieve the necessary reading and writing flexibility, the memory blocks would have to be emulated with registers. This is not much of a concern because the memory blocks themselves are small. The memory is of dimension $word\_size$ by maximum $column\_count$, which in this case comes down to 16 by 8. A memory block would thus consist of 8 registers of size 16. This small interleaver can be seen in Figure 3.18. By giving each register an address, it is possible to write the columns that come from the big interleaver to the appropriate place.

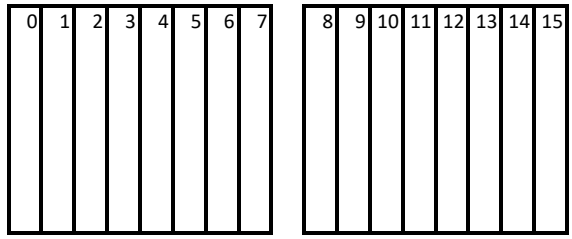

Figure 3.18: small-interleaver

3.6.3 row read order permutation

As described in section 2.3.2, the sequence in which the bits of a row must be read is different depending on the chosen modulation scheme and coding rate. The read sequences can be found in Table 2.2 for the small frames and 2.3 for the normal frames. The correct read sequence can be achieved by writing the columns to the small interleaver according to the inverted read sequence. In this example a normal frame with a modulation scheme of 16+16+16+16 APSK and a coding rate of 128/180 is taken. According to Table 2.3, the read sequence $\mathcal{F}$ is:

$$\mathcal{F} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 0 & 5 & 2 & 1 & 4 \end{pmatrix} \tag{3.48}$$

Inverting this permutation gives the following result:

$$\mathcal{F}^{-1} = \begin{pmatrix} 3 & 0 & 5 & 2 & 1 & 4 \\ 0 & 1 & 2 & 3 & 4 & 5 \end{pmatrix} \tag{3.49}$$

$$= \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 4 & 3 & 0 & 5 & 2 \end{pmatrix} \tag{3.50}$$

Writing the columns in the sequence of [1,4,3,0,5,2], as they come from the big-interleaver, will achieve a read sequences of [3,0,5,2,1,4] when the row is read from left to right. This can be seen in Figure 3.19 for the first row of the big interleaver.

32400	0	54000	21600	10800	43200
32401	1	54001	21601	10801	43201
32402	2	54002	21602	10802	43202
32403	3	54003	21603	10803	43203
32404	4	54004	21604	10804	43204
32405	5	54005	21605	10805	43205
32406	6	54006	21606	10806	43206
32407	7	54007	21607	10807	43207
32408	8	54008	21608	10808	43208
32409	9	54009	21609	10809	43209
32410	10	54010	21610	10810	43210
32411	11	54011	21611	10811	43211
32412	12	54012	21612	10812	43212
32413	13	54013	21613	10813	43213
32414	14	54014	21614	10814	43214
32415	15	54015	21615	10815	43215

Figure 3.19: contents of the small-interleaver after the columns have been written in the sequence of the inverse row permutation

The read sequences has now been detached from the actual reading of the row. Because of this, the outputs of the registers in the small interleaver can be hardwired to the constellation mapper through a much smaller multiplexer network. The column selection operation is something that has to happen regardless and consists of simple register transfers, the total complexity is therefore smaller compared to an explicit bit permutation at the output. Writing to the second block of memory can be achieved by adding an offset of 8.

3.6.4 challenge with indivisibility of words into columns

Packing consecutive bits into words has some problems associated with it. This is because not every configuration has an integer amount of words that can fit in a column. For instance, when dividing 4050 16-bit words across 4 columns, the $word\_size$ becomes 1012.5, this means that a each column has to consist of 1012 full words and one half of a word. How this looks like can be seen in Figure 3.20. Here, the first column of the interleaver has 1012 words written to it. The 1013^{th} word would have to be split into two pieces, where the first part is stored separately. All the subsequent memory contents of the second column contain the tail of the previous word, concatenated with the head of the next word. Eventually, the second column is also full. The remaining tail from the last word is also stored separately. The same process repeats for the remaining two columns.

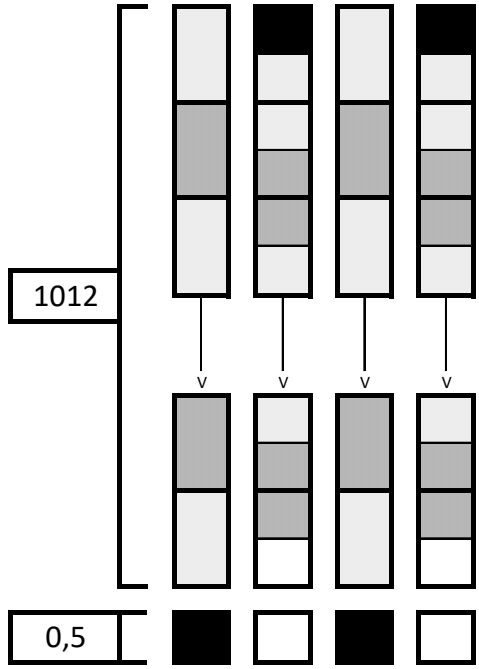

Figure 3.20: interleaver in a 4 columns configuration for a normal frame

The column sizes of all the different configurations for 16-bit words can be seen in Table 3.3. Notice how a lot of these configurations have to deal with the same problem.

Table 3.3: column sizes when 16-bit words are used

word count	column count	3	4	5	6	7	8
4050	normal	1350	1012.5	810	675	578.625	506,25
1012,5	short	337,5	253,125	202,5			

One way to reduce the amount of problem cases is to chosen am alternative word size and repackage the consecutive bits into bigger vectors. In Table 3.4 the column sizes of different configurations can be seen after 16-bit words were repackaged into 20-bit words. This reduces the bad cases to only 2 different configuration. On top of that the memory efficiency is greater as the M20K memory module are be configured to 20-bit input either way.

Table 3.4: column sizes when 20-bit words are used

word count	column count	3	4	5	6	7	8
3240	normal	1080	810	648	540	462.9	405
810	short	270	202,5	162			

It is possible to do even better by compromising on memory efficiency by repackaging the 16-bit words into 18-bit words. The resulting column sizes from the different configurations can be seen in Table Table 3.5. Now only the 128 APSK case remains to be solved.

Table 3.5: column sizes when 18-bit words are used

word count	column count	3	4	5	6	7	8
3600	normal	1200	900	720	600	514,333	450
900	short	300	225	180			

Another similar sweet spot can be found by repacking 16-bit words into 30-bit words. The resulting column sizes from the different configurations can be seen in Table 3.6. This can be used if a decision is made to increase the amount of sequential bits being processed at each clock cycle to increase bandwidth. The downside to this arrangement however is that the small interleaver would have to be bigger in order to support 30-bit columns.

Table 3.6: column sizes when 30-bit words are used

word count	column count	3	4	5	6	7	8
2160	normal	720	540	432	360	308,333	270
540	short	180	135	108			

3.6.5 allowing variable word sizes

When using fixed word sizes that are equal to or bigger then 16 ,it will never be possible to seamlessly interleave a 128 APSK normal frame with other frames. One reason is that this particular frame is bigger duo to the introduction of zero padding. Another reason is that the prime factors of 64806 are 2, 3, 7 and 1543. This makes it impossible to divide 64806 into words bigger then 16 and still have an integer division of 7. In order to seamlessly interleave all the normal frames, variable word-sizes are needed to. Chopping off parts of the frame and storing them separately will not work because the frames then become too small. One solution is to design a word count in such a way that it is divisible by 3 all the way through 8. This can be done by doing a prime factorisation of all the numbers it needs to be divided by and multiplying all the unique prime factors with the highest exponent. The result is 840, this word count can be divided by numbers 3 trough 8. Now a multiple of this number is taken so it would be closer to but below 4050, which is the 3360 in this case. Dividing 64800 bits of a frame (64806 for 128 APSK) into 3360 words gives a word size of 19,28571429 (19,2875 for 128 APSK). This can be achieved by have on part of the words be 16 bits in length and another part of the words be 20 bits in length. In the case of 128 APSK there will be one additional word that is 18 bits in size in each column. All the column sizes and word size distributions can be seen in Figures 3.21 trough 3.26.

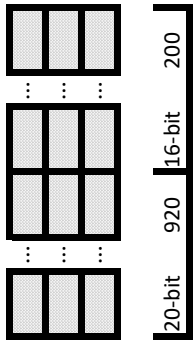

Figure 3.21: interleaver in a 3 column configuration for a word count of 3360

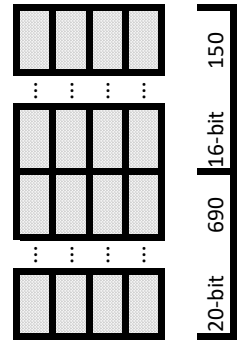

Figure 3.22: interleaver in a 4 column configuration for a word count of 3360

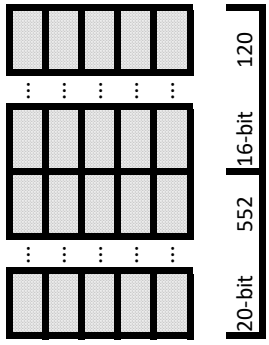

Figure 3.23: interleaver in a 5 column configuration for a word count of 3360

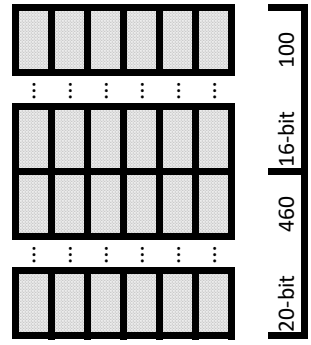

Figure 3.24: interleaver in a 6 column configuration for a word count of 3360

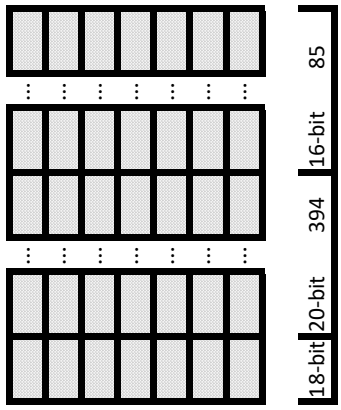

Figure 3.25: interleaver in a 7 column configuration for a word count of 3360

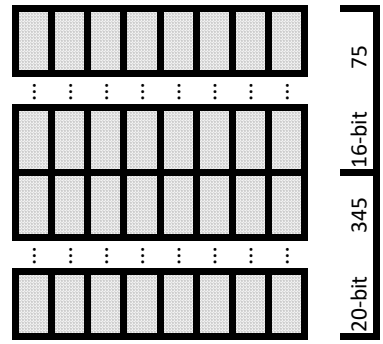

Figure 3.26: interleaver in a 8 column configuration for a word count of 3360

In order for this to work, a flexible repackaging network is needed in front of the big-interleaver and a flexible small-interleaver, that can deal with varying column lengths, after the big-interleaver. Introducing these entities increases the total complexity of the interleaver, but it is necessary if the 128 APSK configuration is a requirement.

3.7 the final design

The final design will consist of a repackaging network followed by a big-interleaver and finally the small-interleaver. It has been agreed upon, for efficiency purposes, that the small interleaver would be incorporated into the constellation mapper itself. A repackaging network has been designed, but duo to a lot of recent developments it is too outdated to be relevant for this design, it has therefore been dropped from the Master's thesis entirely.

Chapter 4

implementation and verification

The big-interleaver has evolved a great deal over the course of the master thesis. The most recent implementation of the big interleaver will be presented and its performance results evaluated.

4.1 big-interleaver implementation

The VHDL implementation of the big-interleaver can be seen in Listing 4.1. This implementation works with a fixed 18-bit word size and is design to be compliant to only the DVB-S2 standard, it can be expanded to the DVB-S2X standard, but only if the 128 APSK modulation is excluded. The full interleaver can be considered complete if the big-interleaver is combined with a 16-bit to 18-bit repackager network and a small interleaver.

Listing 4.1: VHDL code for the big-interleaver

```
1  library IEEE;
2  use IEEE.std_logic_1164.ALL;
3  use IEEE.numeric_std.ALL;
4  entity DVB_interleaver_18bit is
5  port
6  ( clk          : in   std_logic;
7    enable       : in   std_logic;
8    reset        : in   std_logic;
9    frame_select : in   std_logic_vector(0 downto 0);
10   APSK_select  : in   std_logic_vector(2 downto 0);
11   data_i       : in   std_logic_vector(17 downto 0);
12   data_o       : out  std_logic_vector(17 downto 0)
13 );
14 end DVB_interleaver_18bit;
15 architecture concurrent_read_write of DVB_interleaver_18bit is
16   type M20K_block is array (0 to 64800-1) of std_logic_vector(17
       downto 0);
```

```
17    type modulo_LUT is array (0 to 1) of integer;
18    type ADD_LUT is array (0 to 1, 0 to 3) of integer;
19    constant LUT_modulo : modulo_LUT := (0 => 3599, 1=> 899); -- N-1
20    constant LUT_add    : ADD_LUT := (0 => (0=> 1,      1 => 1200,
          2 => 900,      3 => 720),    -- normal
21     1 => (0=> 1,      1 => 300,    2 => 225,      3 => 180)   ); --
          small
22        -- 0=>bypass,   1 => 8PSK,   2 => 16APSK,   3 => 32APSK,   4=>64
              APSK
23    signal interleaver          : M20K_block;
24    signal modulo               : integer range LUT_modulo(1) to
          LUT_modulo(0);
25    signal address              : integer range 0 to LUT_modulo(0);
26    signal next_add             : integer range 1 to LUT_modulo(0);
27    signal add                  : integer range 1 to LUT_modulo(0);
28    signal sub                  : integer range 1 to LUT_modulo(0);
29    signal compare              : integer range 1 to LUT_modulo(0);
30    signal counter              : integer range 1 to LUT_add(0,1)+1;
31    signal threshold            : integer range LUT_add(0,0) to
          LUT_add(0,1);
32    signal enable_counting      : std_logic;
33    signal frame_latch          : integer range 0 to 1;
34  begin
35    main:process(clk)
36      variable var_APSK    : integer range 0 to 3;
37      variable var_frame   : integer range 0 to 1;
38    begin
39      if(rising_edge(clk)) then
40        if(enable = '1') then
41          data_o <= interleaver(address);
42          interleaver(address) <= data_i;
43          if   (address = modulo)  then
44            address <= 0;
45            add     <= next_add;
46            sub     <= modulo - next_add;
47            compare <= modulo - next_add + 1; -- +1 trick
48          elsif(address = 0) then
49            var_APSK := to_integer(unsigned(APSK_select));
50            threshold <= LUT_add(frame_latch,var_APSK);
51            enable_counting <= '1';
52            address <= address + add;
53          elsif(address < compare) then
54            address <= address + add;
55          else
56            address <= address - sub;
57          end if;
58          if ( enable_counting = '1' ) then
59            counter <= counter + 1;
60            if( counter = threshold ) then
61              enable_counting <= '0';
```

```
62              counter <= 1;
63              next_add <= address;
64            else end if;
65          else end if;
66        else -- enable = 0
67          if(reset = '1') then -- numbers valid for every
                configuration.
68            var_frame := to_integer(unsigned(frame_select));
69            frame_latch <= var_frame;
70            modulo <= LUT_modulo(var_frame);
71            add         <= 1;     -- linear fill up
72            sub         <= LUT_modulo(var_frame)-1; -- add-mod, but
                  add = 1 and sign reversed
73            compare     <= LUT_modulo(var_frame);    -- (add-mod) +1
                  but add=1 and sign reversed
74            address     <= 0;
75          else end if;
76        end if;
77      else end if;
78    end process;
79  end concurrent_read_write;
```

Upgrading this interleaver to accept variable word lengths is easy, transforming this interleaver to a de-interleaver is also easy. The only difference is in the input and output vector size, and what values are stored in the LUT. The problem here, however, is that there wasn't much time left for the full verification process.

4.2 ModelSim simulation and verification

The theory has been mathematically proven to work, now it is time to test if the implementation matches the theoretical design. Multiple random frames are generated as input, these frames are then interleaved using the high-level model from subsection 3.2.1, this will then serve as the expected output. All the configurations, inputs and expected outputs are written to a text file. This file is then read by the test bench, the input is given to the big-interleaver and its output is compared to the expected output. The MATLAB code that generates the testing data can be found in Listing 4.2. The test bench that does the verification of the big-interleaver is found in Listing 4.3. Figures 4.1 trough 4.6 show the simulation results form the test bench.

Listing 4.2: MATLAB code that generates the input and expected output

```
1  %% parameters
2  frame_select = 0;
3  % 0 -> normal frame     % 1 -> short frame
4  APSK_select = [0,1,2,3,2,1,0,3,1,2,0,1]+1;
5  % 0 -> bypass      % 2 -> 16APSK
6  % 1 -> 8PSK        % 3 -> 32APSK
7
8  %% constants
```

```
bits_in_word = 18;
address_space = [3600, 900]; % 18 bit wide input
address_space = address_space(frame_select+1);
column_size =  [ [3600, 1200, 900, 720] ; [900, 300, 225, 180] ];
column_size = column_size(frame_select+1,:);
column_count = [ 1,     3,     4,     5    ];
map          = ["000", "001", "010", "011"];
frame_count = size(APSK_select,2);

%% interleaver
input = randi([0,1],[frame_count, address_space, bits_in_word]);

fileID = fopen('Test_input_interleaver.txt','w');
fprintf(fileID,'%d %d\n',frame_count, frame_select);
for i = 1:frame_count
    fprintf(fileID,'%s ',map(APSK_select(i)));
    fprintf(fileID,'%d',permute(input(i,:,:), [3,2,1])) ;
    fprintf(fileID,' ');
    interleaver = reshape(input(i,:,:),[column_size(APSK_select(i)
        ), column_count(APSK_select(i)), bits_in_word]);
    interleaver = permute(interleaver, [2 1 3]);
    fprintf(fileID,'%d',permute(reshape(interleaver,[1,
        address_space, bits_in_word]), [3,2,1]));
    fprintf(fileID,'\n');
end
fclose(fileID);
disp("done");
```

Listing 4.3: VHDL testbench code for the big-interleaver

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.ALL;
use STD.textio.all;
use IEEE.std_logic_textio.all;
entity tb is end entity;
architecture test of tb is
  signal EndOfSim    : boolean := false;
  constant half_clk  : time := 10 ps;
  component DVB_interleaver_18bit
    port
    ( clk           : in   std_logic;
    enable          : in   std_logic;
    reset           : in   std_logic;
    frame_select    : in   std_logic_vector(0 downto 0);
    APSK_select     : in   std_logic_vector(2 downto 0);
    data_i          : in   std_logic_vector(17 downto 0);
    data_o          : out  std_logic_vector(17 downto 0)
    );
  end component;
```

```
21    type frame_size_array     is array (0 to 1) of integer;
22    constant frame_size     : frame_size_array := (0 => 64800, 1 =>
          16200);
23    constant address_space : frame_size_array := (0 => 3600, 1 =>
          900);
24    constant input_size     : integer := 18;
25    signal   clk            :  std_logic := '0';
26    signal   enable         :  std_logic := '0';
27    signal   reset          :  std_logic := '0';
28    signal   frame_select   :  std_logic_vector(0 downto 0);
29    signal   APSK_select    :  std_logic_vector(2 downto 0);
30    signal   data_i         :  std_logic_vector(input_size-1 downto
          0);
31    signal   data_o         :  std_logic_vector(input_size-1 downto
          0);
32    signal   expected       :  std_logic_vector(input_size-1 downto
          0);
33    signal   start_r        :  std_logic;
34    signal   correct        :  std_logic;
35    file     testing_data   :  text;
36  begin
37    dut: DVB_interleaver_18bit
38    port map
39    ( clk             =>   clk,
40      enable          =>   enable,
41      reset           =>   reset,
42      data_i          =>   data_i,
43      data_o          =>   data_o,
44      frame_select    =>   frame_select,
45      APSK_select     =>   APSK_select
46    );
47    clock: process -- clock generator, toggel clk every half periode
          .
48    begin
49      if EndOfSim then wait; end if;
50      clk <= not clk;
51      wait for half_clk;
52    end process;
53    main: process
54      variable read_line         : line;
55      variable frame_count       : integer;
56      variable M_frame_select    : std_logic_vector(0 downto 0);
57      variable int_frame_select : integer;
58      variable M_APSK_SELECT     : std_logic_vector(2 downto 0);
59      variable M_INPUT           : std_logic_vector(0 to frame_size
            (0) - 1);
60      variable M_OUTPUT          : std_logic_vector(0 to frame_size
            (0) - 1);
61      variable frame_counter     : integer;
62      variable input_counter     : integer;
```

```
63      variable input              : std_logic_vector(input_size-1
            downto 0);
64      variable output             : std_logic_vector(input_size-1
            downto 0);
65      variable lower_bound        : integer;
66      variable upper_bound        : integer;
67    begin
68      file_open(testing_data, "Test_input_interleaver.txt",
            read_mode);
69      readline(testing_data, read_line);
70      read(read_line,frame_count);
71      read(read_line,M_frame_select);
72      int_frame_select := to_integer(unsigned(M_frame_select));
73      report "frame_count = " & integer'image(frame_count);
74      wait until falling_edge(clk);
75      enable <= '0';
76      reset <= '1';
77      frame_select <= M_frame_select;
78      wait until rising_edge(clk);
79      wait until falling_edge(clk);
80      enable <= '1';
81      reset <= '0';
82      frame_counter := 0;
83      --first loop that writes frames
84      while frame_counter < frame_count+1 loop
85        if(frame_counter < frame_count ) then
86        readline(testing_data, read_line);
87        read(read_line,M_APSK_SELECT);
88        read(read_line,M_INPUT(0 to frame_size(int_frame_select)-1))
              ;
89        report "Frame: " & integer'image(frame_counter)  & "
              M_APSK_SELECT = " & integer'image(to_integer(unsigned(
              M_APSK_SELECT)));
90        else end if;
91        --second loop that writes words
92        input_counter := 0;
93        while input_counter < (address_space(int_frame_select)) loop
94          lower_bound := input_size * input_counter;
95          upper_bound := input_size * (input_counter+1)-1;
96          -- write input ( don't write at the end )
97          if(frame_counter < frame_count) then
98            APSK_select <= M_APSK_SELECT;
99            data_i <= M_INPUT(lower_bound to upper_bound);
100         else
101           data_i <= (others => 'U');
102         end if;
103         wait until rising_edge(clk);
104                 -- write expected output (don't write at the
                        beginning)
105         if(frame_counter >0) then
```

```
106             expected <= M_OUTPUT(lower_bound to upper_bound);
107           else end if;
108           wait until falling_edge(clk);
109           input_counter := input_counter + 1;
110         end loop;
111         if(frame_counter < frame_count ) then
112         read(read_line,M_OUTPUT(0 to frame_size(int_frame_select)-1)
              );
113         else end if;
114         start_r <= '1';
115         frame_counter := frame_counter + 1;
116       end loop;
117       file_close(testing_data);
118       start_r <= '0';
119       enable <= '0';
120       wait for 10 * half_clk;
121       EndOfSim <= true;
122       report "simulation finished successfully" severity FAILURE;
123       wait;
124     end process;
125     testing: process
126     begin
127       wait until falling_edge(clk);
128       if(start_r = '1') then
129         if( expected = data_o) then
130           correct <= '1';
131         else
132           report "ERROR: incorrect";
133           wait for 4*half_clk;
134           report "ERROR: exiting" severity FAILURE ;
135           correct <= '0';
136         end if;
137       end if;
138     end process;
139     end architecture;
```

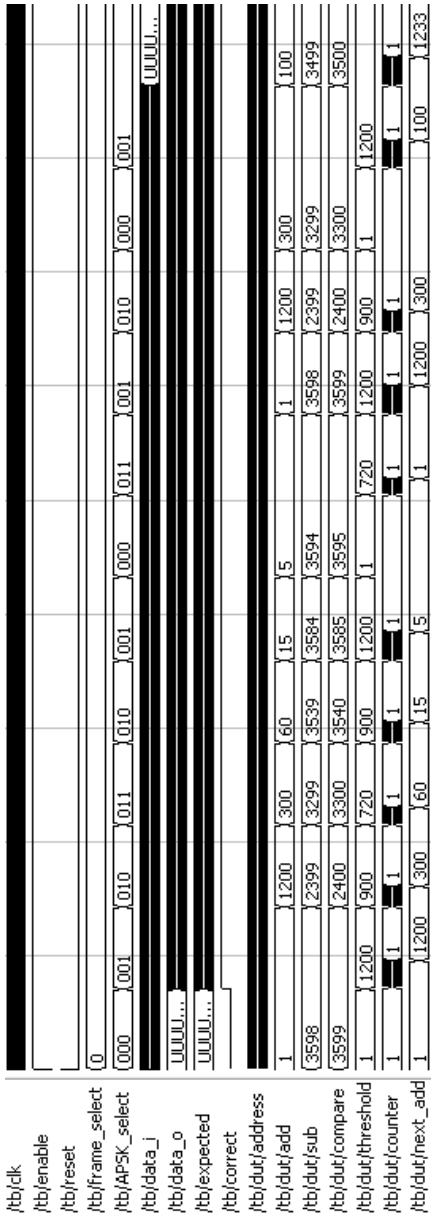

Figure 4.1: overview of the full interleaving process of 12 normal frames

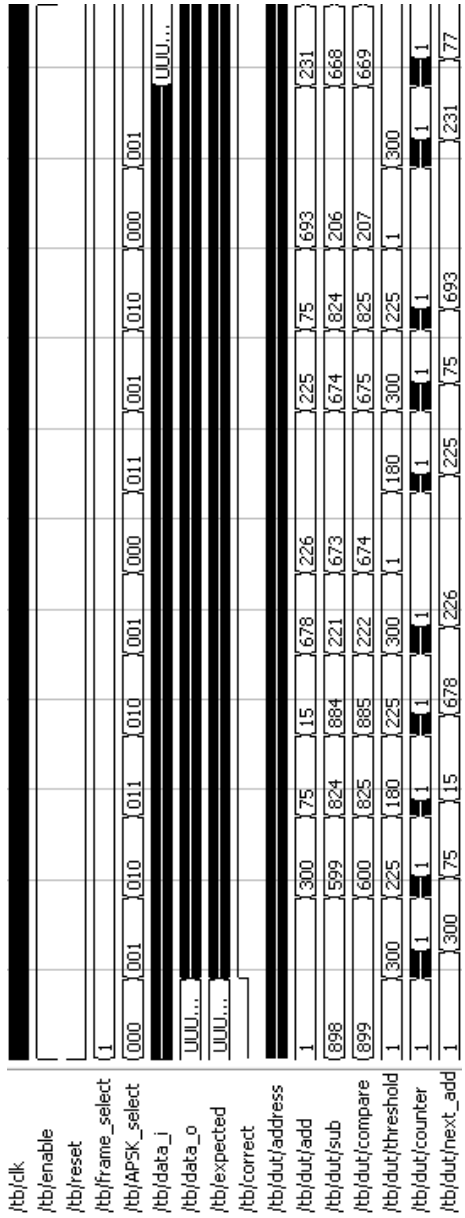

Figure 4.2: overview of the full interleaving process of 12 small frames

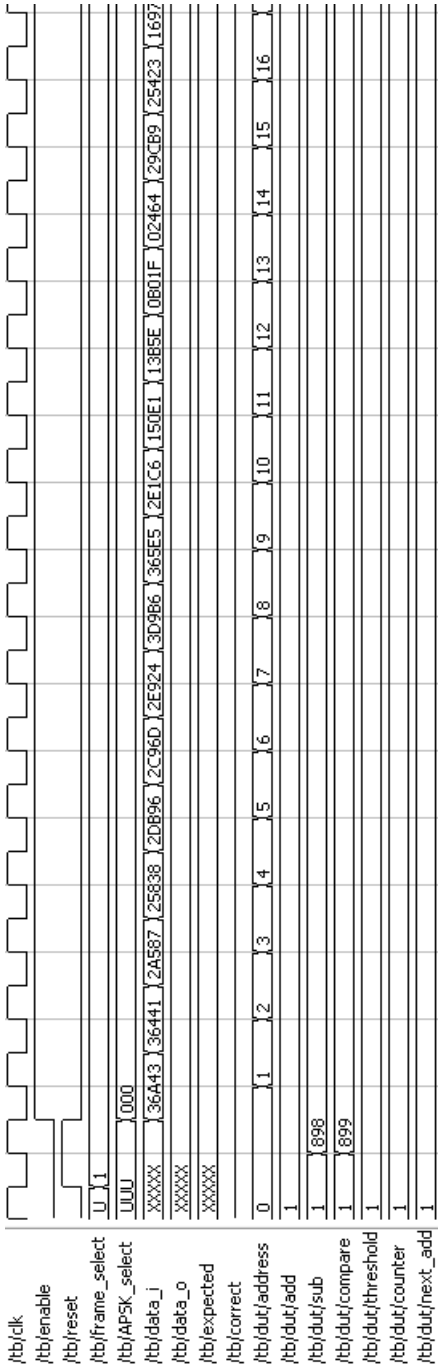

Figure 4.3: start up behaviour for the small frame configuration

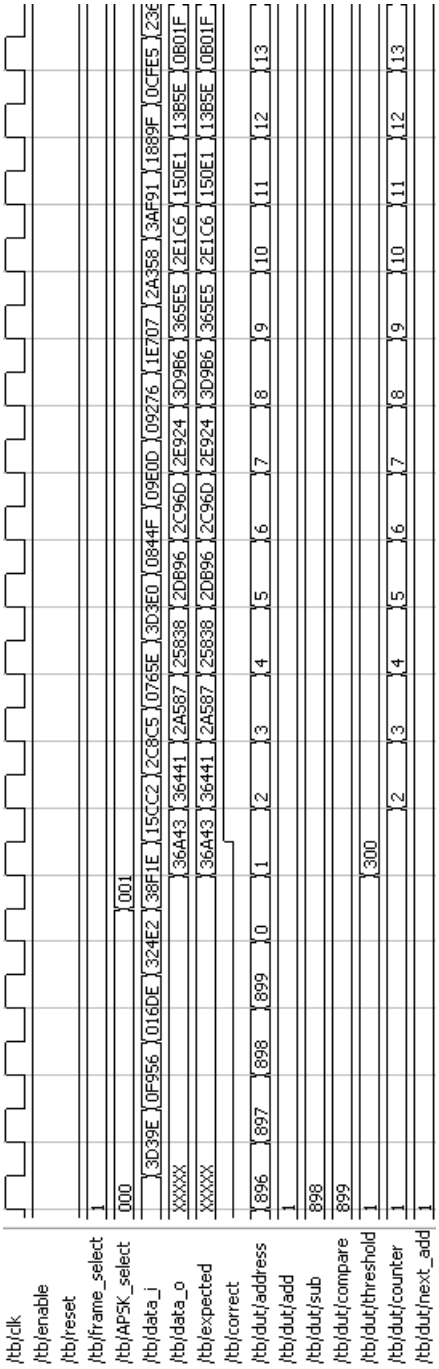

Figure 4.4: output small frame 0 and input small frame 1

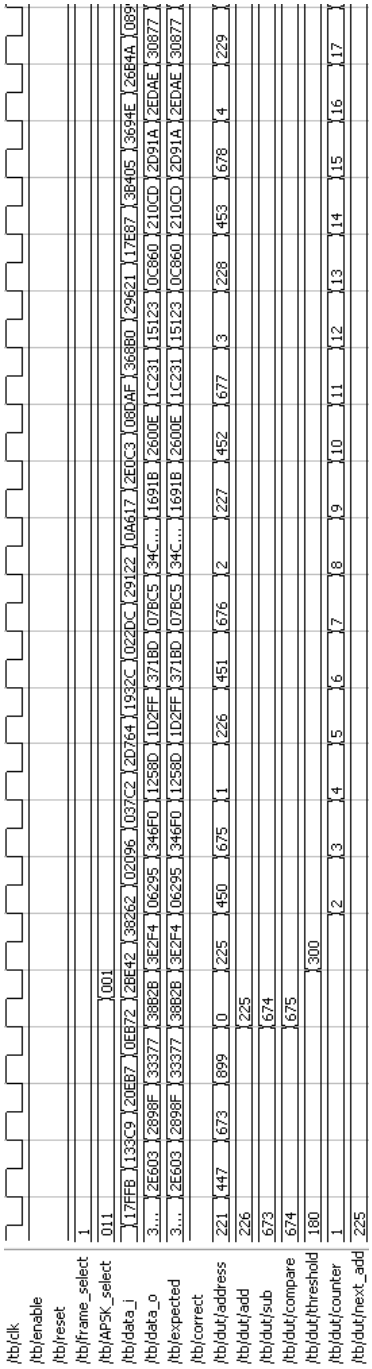

Figure 4.5: output small frame 7 and input small frame 8

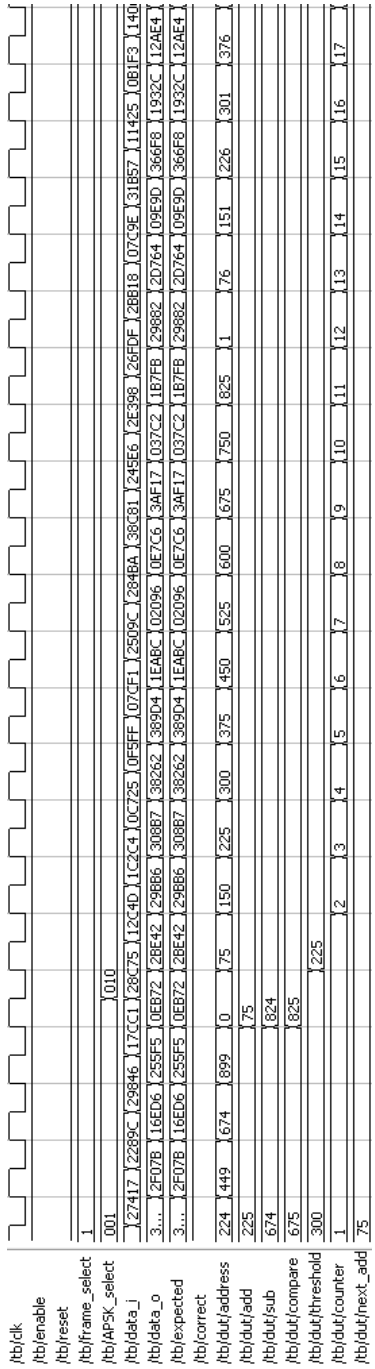

Figure 4.6: output small frame 8 and input small frame 9

4.3 Quartus timing and resource usage analysis

WideNorth has not yet decided on what FPGA will be used, but it will most probably be the Intel Stratix 10 FPGA, The particular device choice ranges from the SX 850 to the SX 2800. Out of this range, 1SX085HN3F43I3XG has been chosen as the target device for compilation and verification. It is the slowest, low power SX 850 device, this is needed in order to have a good worst-case scenario.

The design, as can be found in Listing 4.1, has been compiled in Quartus Prime Pro Edition v18.0 with a trail licence. The timing results can be found in Table 4.1. As can be seen from the results, under the worst performing scenario, the maximum frequency this design can operate at is 476.87 MHz. This design exceeds the 350 MHz target by an additional 126,87 MHz. It can be said that this design has successfully reached its goal by a large margin. A good portion of this performance, however, can be given credit to the fact that the Stratix 10 is the fastest and most state-of-the-art FPGA series currently on the market. A direct comparison to other designs is not possible duo them being implemented on other devices and silicon processes.

Based on the top failing paths analysis it is possible to increase the speed even further. One way is to duplicate the modulo LUT for the reset part of the interleaver. Another way is to calculate the next ADD and compare values as soon as the appropriate address is captured instead of at the end of each frame.

A summary of the block resource usage and routing resource usage can be seen in Table 4.2 and 4.3 respectively. On the chip planner it is possible to see where all the used resources are located. Figure 4.7 shows the occupied M20K and LAB blocks on the FPGA fabric, Figure 4.8 shows the whole FPGA fabric for reference. With these results it is possible to conclude that this design is very resource efficient.

Table 4.1: big-interleaver maximum theoretical frequency

corner model	temperature	max freq (MHz)
slow	100C	501.5
slow	-40C	476.87
fast	100C	696.38
fast	-40C	768.05

Table 4.2: block resource usage

block resource usage	usage	available	relative
logic utilization (in ALMs)	93	284,96	<1%
total dedicated logic registers	85		
total block memory bits	73,728	71,208,960	<1%
total RAM Blocks	4	3,477	<1%

Table 4.3: routing resource usage

routing resource type	usage	available	relative
block input muxes	18	361,828	<1 %
block interconnects	485	4,381,776	<1 %
C16 interconnects	29	108,864	<1 %
C2 interconnects	57	653,184	<1 %
C3 interconnects	65	1,325,376	<1 %
C4 interconnects	141	851,904	<1 %
CLOCK_INVERTs	2	4,032	<1 %
DCM_muxes	1	1,088	<1 %
direct links	140		<1 %
GAP Interconnects	59	173,376	<1 %
GAPs	0	29,304	0%
HIO Buffers	23	145,152	<1 %
horizontal Buffers	12	115,776	<1 %
horizontal_clock_segment_muxes	6	4,032	<1 %
programmable inverts	22	195,696	<1 %
R10 interconnects	86	1,216,008	<1 %
R2 interconnects	75	1,088,640	<1 %
R24 interconnects	94	152,064	<1 %
R24/C16 interconnect drivers	36	217,728	<1 %
R4 interconnects	86	1,574,496	<1 %
row clock tap-offs	14	350,352	<1 %
switchbox_clock_muxes	22	23,04	<1 %
vertical_seam_tap_muxes	11	12,096	<1 %

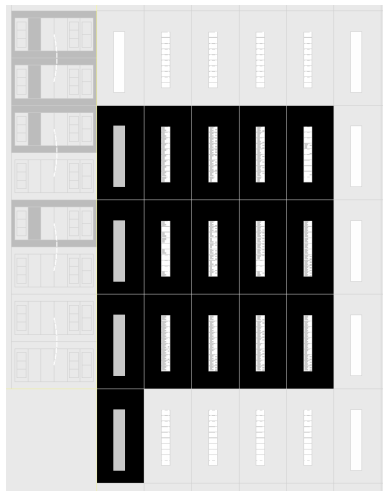

Figure 4.7: big-interleaver implementation on the FPGA fabric

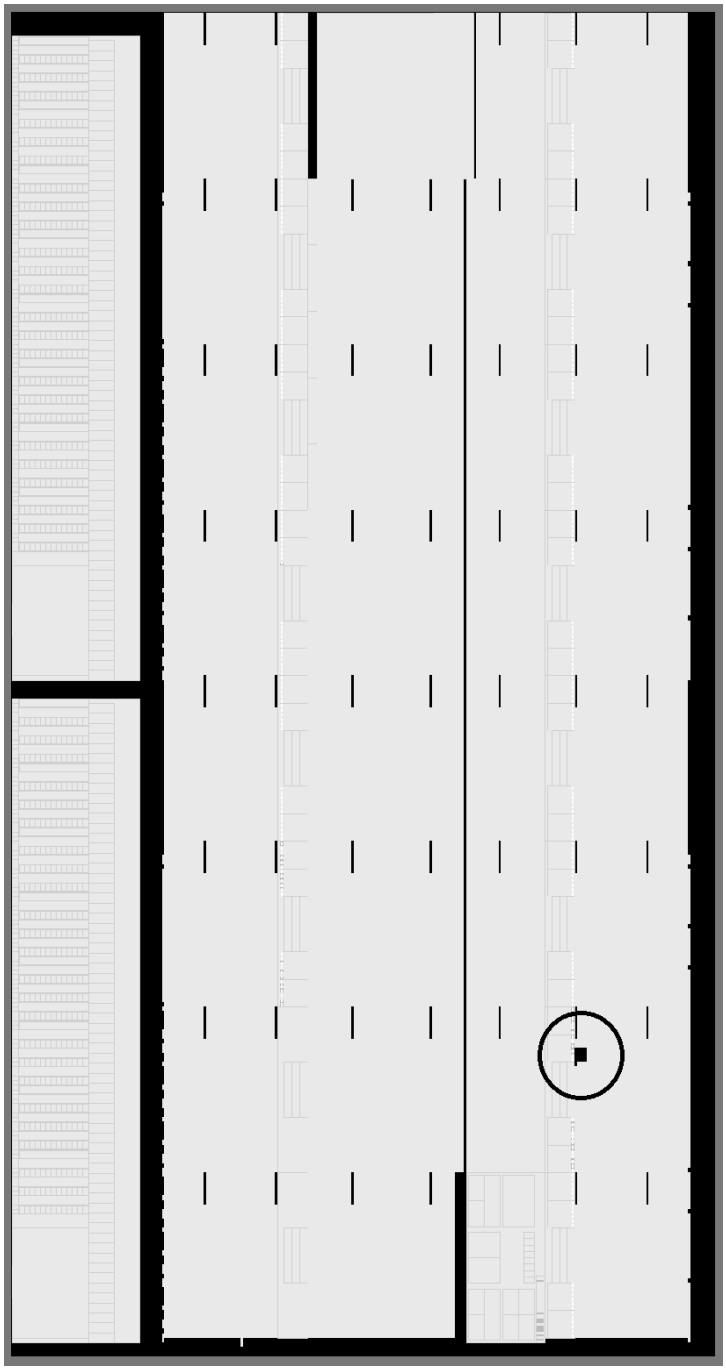

Figure 4.8: the whole 1SX085HN3F43I3XG FPGA fabric

Chapter 5

Conclusion

The goal of this Master's Thesis is to implement a high speed DVB-S2 compliant block-interleaver with the possibility to extend it to the DVB-S2X standard. The interleaving process was equated to a permutation operation that realises a matrix transposition. Two MATLAB models have been evaluated. The first one was based on what was already present in the literature. The second one is an entirely new proposition. Both designs have been evaluated and the second design has been chosen as being the better one suited for this application. With this new design it is possible to seamlessly interleave consecutive frames with only one memory block. arguments, proofs, and demonstrations have been given in order to back up that claims. Multiple challenges have been solved such as, switching between configurations without having to reset the interleaver, handling multiple consecutive bits in high throughput applications, and handling slightly different frame sizes, which was needed to allow 128 APSK modulation. the end result is that the interleaver has been split up into three pieces. The first piece is the words repackaging network that has to repackage consecutive bits in varying word sizes, it has not been implemented successfully duo to a lack of time. The second part is the Big interleaver that achieves intermediate interleaving by interleaving the words themselves instead of their contents. The final part is the small-interleaver that reads in multiple words, which can be seen as columns, into a matrix in order to then read out all the rows to finish the interleaving process. This part will be incorporated into the constellation mapper. A slightly older, but still relevant version of the big interleaver has been implemented and verified. A ModelSim simulation and a Quartus timing and resource usage analysis was preformed. The predefined performance goal of 350 MHz was exceeded by an additional 126,87 MHz. This new way of doing block-interleaving is practical to implement and scales very well duo to its simplicity.

Bibliography

[1] *Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications; Part 1: DVB-S2*, European Telecommunications Standards Institute, 11 2014, eTSI EN 302 307-1 V1.4.1.

[2] J. Bruant, "Generic high-speed broadband sdr platform implemented in fpga using direct sampling at l-band," 2018, trondheim FPGA Forum 2018.

[3] D. MacKay, *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[4] *Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications; Part 2: DVB-S2 Extensions (DVB-S2X)*, European Telecommunications Standards Institute, 10 2014, eTSI EN 302 307-2 V1.1.1.

[5] J. Lei and W. Gao, "Matching Graph Connectivity of LDPC Codes to High-Order Modulation by Bit Interleaving," in *2008 46TH ANNUAL ALLERTON CONFERENCE ON COMMUNICATION, CONTROL, AND COMPUTING, VOLS 1-3*, ser. Annual Allerton Conference on Communication Control and Computing. 345 E 47TH ST, NEW YORK, NY 10017 USA: IEEE, 2008, Proceedings Paper, pp. 1059+, 46th Annual Allerton Conference on Communication, Control and Computing, Monticello, IL, SEP, 2008.

[6] M. Jang, H. Lee, S.-H. Kim, S. Myung, H. Jeong, and J. Kim, "Design of LDPC Coded BICM in DVB Broadcasting Systems With Block Permutations," *IEEE TRANSACTIONS ON BROADCASTING*, vol. 61, no. 2, pp. 327–333, JUN 2015.

[7] R. Jose and A. Pe, "Analysis of Hard Decision and Soft Decision Decoding Algorithms of LDPC Codes in AWGN," in *2015 IEEE INTERNATIONAL ADVANCE COMPUTING CONFERENCE (IACC)*, ser. IEEE International Advance Computing

Conference. 345 E 47TH ST, NEW YORK, NY 10017 USA: IEEE, 2015, Proceedings Paper, pp. 430–435, IEEE International Advance Computing Conference (IACC 2015), Bangalore, INDIA, JUN 12-13, 2015.

[8] N. P. Bhavsar and B. Vala, "Article: Design of hard and soft decision decoding algorithms of ldpc," *International Journal of Computer Applications*, vol. 90, no. 16, pp. 10–15, March 2014, full text available.

[9] D. Bhardwaj, S. P. Singh, and V. K Pandey, "Vhdl implementation of efficient multimode block interleaver for wimax," *International Journal of Engineering Trends and Technology (IJETT)*, vol. 1, 03 2012.

[10] S. Mohanty and N. M. Sk, "A Novel Interleaver Design for Multimode Communication in WLAN," in *2014 INTERNATIONAL CONFERENCE ON SIGNAL PROCESSING AND INTEGRATED NETWORKS (SPIN)*, IEEE UP Sect; IEEE. 345 E 47TH ST, NEW YORK, NY 10017 USA: IEEE, 2014, Proceedings Paper, pp. 286–290, 1st International Conference on Signal Processing and Integrated Networks (SPIN), Amity Univ Campus, Amity Sch Engn & Technol, Noida, INDIA, FEB 20-21, 2014.

[11] C. Yu, M.-H. Yen, P.-A. Hsiung, and S.-J. Chen, "Design of a High-Speed Block Interleaving/Deinterleaving Architecture for Wireless Communication Applications," in *2009 IEEE INTERNATIONAL CONFERENCE ON CONSUMER ELECTRONICS*, IEEE. 345 E 47TH ST, NEW YORK, NY 10017 USA: IEEE, 2009, Proceedings Paper, pp. 237+, 27th IEEE International Conference on Consumer Electronics, Las Vegas, NV, JAN 10-14, 2009.

[12] *Stratix 10 GX/SX Device Overview*, Intel, 10 2017, s10-overview.

[13] *Intel Stratix 10 Logic Array Blocks and Adaptive Logic Modules User Guide*, Intel, 11 2017, ug-s10lab.

[14] *Intel Stratix 10 Embedded Memory User Guide*, Intel, 12 2017, ug-s10memory.

Appendix

The copyright notice of ETSI can be found below.

Sophia Antipolis, 3 July 2018

Youri Vladimirovitch Vassiliev
Ghent University
Belgium

Our ref.: L2018-001-015

Subject: COPYRIGHT LICENSE from ETSI to reproduce parts of ETSI Deliverables or other ETSI material

Dear Youri Vladimirovitch Vassiliev,

In response to your copyright request, please be informed that ETSI material is protected by copyright, the ownership of which vests in ETSI, unless otherwise provided.

ETSI is pleased to hereby grant you the non-exclusive right to reproduce free of charge, without modification whatsoever, parts of the ETSI Deliverables as listed in Annex, to be used as described in your request, provided that you insert the relevant copyright notice provided in Annex, as due acknowledgement of the source, wherever appropriate.

This copyright license allows for reproduction only, on a world-wide basis and for an unlimited period of time.

Yours sincerely,

Christian Loyau
Legal and Governance Affairs Director

European Telecommunications Standards Institute - Institut Européen des Normes de Télécommunications - Europäisches Institut für Telekommunikationsnormen
ETSI - 650, route des Lucioles F-06921 Sophia Antipolis Cedex - FRANCE
Tél. : +33 (0)4 92 94 42 00
e-mail : info@etsi.org - http://www.etsi.org
Fax : +33 (0)4 93 65 47 16
Siret N° 348 623 562 00017 – APE 7112B - Association à but non lucratif enregistrée à la Sous-Préfecture de Grasse (06) N° W061004871 - N° TVA : FR 14 348 623 562

ANNEX

ETSI EN 302 307-1	V1.4.1 (2014)	Clause 5.3, Figures 6, 7, 8 and Table 8 Clause 4.2, Figure 1

Paragraph to be inserted wherever appropriate:

ETSI EN 302 307-2	V1.1.1 (2015)	Clause 5.3.3 Tables 9a, 9b

Paragraph to be inserted wherever appropriate: