NTNU

Norwegian University of
Science and Technology

Efficient Streaming and Compression of Hyperspectral Images

Johan Austlid Fjeldtvedt

Master of Science in Electronics
Submission date: July 2018
Supervisor: Kjetil Svarstad, IES
Co-supervisor: Milica Orlandic, IES

Norwegian University of Science and Technology
Department of Electronic Systems

PROJECT ASSIGNMENT

Candidate name: Johan Fjeldtvedt

Assignment title: Testing of Communication between Various Peripherals on ZED-BOARD

Assignment text:

This topic is part of the large project Hyperspectral Imaging in Small Satellites. Hyperspectral imaging relies on sophisticated acquisition and on data processing of hundreds or thousands of image bands. Remote sensing data interpretation can be performed on board resulting in significant reduction of communication bandwidth. The trend in remote sensing missions has been increasing towards using hardware devices of small size, low cost, high computational power and high flexibility. This project is intended to explore various communication channels and IP creation on ZedBoard for establishing efficient System on Chip supporting fast hyperspectral imagining algorithm modules.

ZedBoard features ZYNQ device and a number of peripheral interfaces such as GPIOs, Audio Codecs, Video outputs, Ethernet, USB ports, SD slots, flash and DDR3 memory, etc. Zynq comprises a processing system (PS) around a dual core ARM Cortex A9 processor and programmable logic (PL) equivalent to FPGA. It also features integrated memory and a variety of high-speed communication interfaces.

- Establish complete dataflow for OTFP pre-processing

- Implement CCSDS123 – BSQ and BI processing order, Test communication with the rest of the system

Requirements:

C/C++, VHDL, tool Vivado

Assignment proposer / Co-supervisor: Milica Orlandic, IES

Supervisor: Kjetil Svarstad, IES

Abstract

As a part of the SmallSat project at NTNU, a satellite payload capable of capturing and processing of hyperspectral images is being developed. Several processing steps are performed on-board in the satellite, as well as compression of the resulting data to reduce storage needs and to improve utilization of the limited throughput of the radio link to the ground station. As the spatial, spectral and temporal dimensions of hyperspectral images have increased, the need for performing these processing steps in hardware has emerged. FPGA based solutions are attractive due to their reprogrammability and reduced cost compared to dedicated ASICs. In this thesis, two problems related to hyperspectral image processing in FPGA are explored.

The first problem is related to achieving high throughput when data is streamed into and out of hardware processing cores that require streaming in different orders. To achieve high performance, memory accesses must be performed as efficiently as possible for all memory access patterns. Investigating this problem in the feasibility study prior to this thesis work has led to the conclusion that the available direct memory access (DMA) solutions are not suitable. During that work, a new special-purpose DMA core for streaming of hyperspectral images, the *Cube DMA*, was developed. Further developments of this core are presented in this thesis, including improvements in how transfers are performed, implementation of stream to memory channel and addition of stream control signals. The results show an increased throughput of 128% for block-wise transfers compared to existing DMA solutions, while FPGA resource utilization is lower.

The other problem that has been explored is compression of image data in the satellite. CCSDS123 is a compression algorithm designed for hyperspectral images. An efficient and parallelized hardware implementation of this algorithm has been designed, implemented and thoroughly verified. The results show that its performance is better than any state of the art implementations in terms of achievable data throughput, while showing modest power usage and resource utilization.

Sammendrag

Som en del av SmallSat-prosjektet ved NTNU utvikles det en nyttelast for satelitter som er i stand til takning og prosessering av hyperspektrale bilder. Flere prosesseringssteg utføres om bord i satelitten, samt komprimering av resulterende data for å redusere behovene for lagringsplass og for å utnytte den begrensede radioforbindelsen til bakkestasjonen bedre. Etter hvert som de romlige, spektrale og temporale dimensjonene i hyperspektrale bilder har økt, har det blitt nødvendig å utføre disse prosesseringsstegene i maskinvare. FPGA-baserte løsninger er attraktive på grunn av reprogrammerbarheten og de reduserte kostnadene sammenlignet med dedikerte ASIC-er.

I denne oppgaven har to problemer knyttet til hyperspektral bildeprosessering i FPGA blitt utforsket. Det første er relatert til å oppnå høy hastighet ved strømming av data inn og ut av prosesseringskjerner i maskinvare som trenger at data strømmes i forskjellige rekkefølger. For å oppnå høy ytelse må minneaksesser skje så effektivt som mulig for alle minneaksessmønstere. Undersøkelser av dette problemet har ledet til den konklusjonen at eksisterende løsninger for direkte minneaksess (DMA) er uegnede. I det forgående prosjektarbeidet ble en ny DMA-kjerne, *Cube DMA* utviklet spesifikt for strømming av hyperspektrale bilder. Videreutvikling av denne kjernen er presentert i denne oppgaven, som inkluderer forbedringer av hvordan overføringer utføres, implementasjon av en kanal for overføring fra strøm til minne, og strømkontrollsignaler. Resultatene viser betydelige forbedringer i ytelse sammenlignet med eksisterende DMA-løsninger, og ressursbruken er mindre.

Det andre problemet som har blitt utforsket er komprimering av bildedata i satelitten. CCSDS123 er en kompresjonsalgoritme som er designet for hyperspektrale bilder. En effektiv og parallelisert FPGA-implementasjon av denne algoritmen har blitt designet, implementert og grundig verifisert. Resultatene viser at ytelsen er bedre enn de nyeste av implementasjoner når det kommer til oppnåelig gjennomstrømning (throughput), og viser samtidig beskjeden effekt- og arealforbruk.

Preface

This master's thesis is the final part of my Master of Science degree in Electronics, and concludes eight great years at the Norwegian University of Science and Technology (NTNU).

I would first of all like to thank my co-supervisor Milica Orlandić for her support, encouragement and discussions, and for her tireless work with writing and editing the journal papers that we have published based on this work. It's been a great experience and I have learned a lot from it. Many thanks to Espen Moen for testing the Cube DMA core as a part of his own master's thesis work. Lastly, thanks to my family and friends for their support through the years.

Johan Fjeldtvedt,

Oslo, July 2018

Contents

1 Introduction **1**
- 1.1 The NTNU SmallSat project . . . 1
- 1.2 Hyperspectral imaging . . . 3
  - 1.2.1 Component orderings . . . 4
- 1.3 SmallSat HSI payload . . . 5
  - 1.3.1 HSI camera . . . 6
  - 1.3.2 Zynq 7000-series All Programmable System on Chip . . . 8
  - 1.3.3 Integration of hardware and software . . . 10
- 1.4 Main contributions . . . 11
- 1.5 Structure of this thesis . . . 12

2 Cube DMA: A DMA core for hyperspectral images **15**
- 2.1 Introduction . . . 15
  - 2.1.1 Summary of feasibility study findings . . . 16
- 2.2 AXI bus standards . . . 20
  - 2.2.1 The AXI bus standards . . . 20
- 2.3 Overview of the Cube DMA core . . . 23
  - 2.3.1 MM2S channel . . . 24
  - 2.3.2 S2MM . . . 26
- 2.4 Ports, generics and register layout . . . 26
  - 2.4.1 Interface ports . . . 26
  - 2.4.2 Generic parameters . . . 27
  - 2.4.3 Register configuration . . . 28
- 2.5 Overview of MM2S channel implementation . . . 31
  - 2.5.1 DataMover . . . 31
  - 2.5.2 Unpacker . . . 32
  - 2.5.3 Controller . . . 34
- 2.6 The TinyMover core . . . 37
- 2.7 Integrating DMAs with software . . . 38
  - 2.7.1 Interrupts on Zynq-7000 . . . 38
  - 2.7.2 Caches . . . 40

3 Cube DMA implementation **43**

3.1 New address generation logic . . . 43
3.2 Control stream implementation . . . 45
3.3 S2MM channel implementation . . . 47
3.3.1 Controller . . . 47
3.3.2 Component packer . . . 47
3.4 Test setup for comparing Xilinx AXI DMA and Cube DMA . . . 50

4 Cube DMA results 53
4.1 Performance comparison . . . 53
4.2 Resource utilization . . . 55
4.3 Timing . . . 60

5 CCSDS123 theory and background 61
5.1 Overview . . . 61
5.2 Definitions . . . 62
5.3 Prediction . . . 63
5.3.1 Local sum and local difference vector . . . 63
5.3.2 Weights . . . 66
5.3.3 Prediction calculation . . . 67
5.3.4 Residual mapping . . . 67
5.3.5 Weight update . . . 68
5.4 Encoding . . . 69
5.4.1 Golomb-Power-Of-2 coding . . . 69
5.4.2 Adaptation to image statistics . . . 69
5.4.3 Encoding of a sample . . . 70
5.5 Summary of parameters . . . 71

6 Hardware implementation of CCSDS123 compressor 73
6.1 Memory and performance trade-offs . . . 73
6.1.1 Streaming efficiency . . . 74
6.1.2 Local space requirements . . . 74
6.1.3 Pipelining and parallelization . . . 76
6.2 Previous work . . . 76
6.3 Existing software implementations . . . 78
6.4 Serial implementation . . . 78
6.4.1 Control signal generation . . . 80
6.4.2 Sample delay . . . 80
6.4.3 Local sum and difference calculations . . . 81
6.4.4 Central difference store . . . 82
6.4.5 Weight store . . . 82
6.4.6 Dot product . . . 83
6.4.7 Predictor . . . 84
6.4.8 Weight update . . . 86
6.4.9 Residual mapping . . . 86
6.4.10 Encoding . . . 88

6.4.11 Bit packing . . . 88
6.4.12 AXI Stream interfacing . . . 90
6.5 Parallel implementation . . . 91
6.5.1 Streaming of samples in parallel . . . 91
6.5.2 Overview of architecture . . . 93
6.5.3 Definitions and terms . . . 93
6.5.4 Sample delay . . . 95
6.5.5 Local differences . . . 96
6.5.6 Weight and accumulator storage . . . 97
6.5.7 Packing of variable length words . . . 100
6.5.8 Improved packer . . . 104

7 CCSDS123 implementation results 109
7.1 Compression analysis . . . 109
7.2 Implementation results . . . 110
7.2.1 Resource utilization . . . 111
7.2.2 Timing . . . 123
7.2.3 Power estimation . . . 124
7.3 Comparison with existing work . . . 126

8 Verification and testing 129
8.1 Simulation . . . 129
8.1.1 Cube DMA . . . 129
8.1.2 CCSDS123 . . . 130
8.2 Testing on hardware . . . 133
8.2.1 Interfacing with Zynq 7000 SoC . . . 134
8.2.2 Xilinx ChipScope debugging . . . 135
8.2.3 Stream monitors . . . 135
8.2.4 Typical hardware testing flow . . . 137

9 Conclusion 139
9.1 Cube DMA . . . 139
9.2 CCSDS123 . . . 140
9.3 Future work . . . 141
9.3.1 Cube DMA . . . 141
9.3.2 CCSDS123 . . . 142

A Using the automatic verification scripts 145
A.1 Installing Emporda . . . 145
A.2 Using the automatic verification scripts . . . 146
A.2.1 Creating a simulation snapshot . . . 146
A.2.2 Generating a cube . . . 146
A.2.3 Creating configuration file and running . . . 147
A.2.4 Running manually or in Vivado GUI . . . 148
A.3 Performing randomized testing . . . 148

A.3.1 Handling of failed tests 149
A.4 Investigating errors 150

B Using simulation results to improve power estimates 153

List of Figures

1.1 The main components of the proposed multi-agent marine observation system [1] 2
1.2 Phytoplankton bloom at the coast of Norway, observed from space [1] . 3
1.3 Hyperspectral image cube 4
1.4 Different sample orderings in hyperspectral images, illustrated with an example image of size $4 \times 4 \times 4$ 5
1.5 The HSI payload processing and control architecture [1] 7
1.6 Hyperspectral image acquisition process using push broom scanning . . 7
1.7 Overview of the Zynq-7000 architecture [2] 9
1.8 Typical setup for on the fly processing 10
1.9 Typical architecture of a hardware/software system in the Zynq-7000 . . 11

2.1 Block-wise streaming of a HSI cube in BIP order. The arrows indicate the order in which the components are streamed from a block, and the numbers indicate the order in which each block is streamed. 17
2.2 Block wise streaming of a HSI cube in BSQ order. The arrows indicate the order in which the components are streamed from a block, and the block numbers indicate the order in which each block is streamed. The numbers on each plane indicate the order in which each band is streamed. 17
2.3 Scatter-gather transfer using block descriptors 18
2.4 Horizontal size, stride and vertical size when using the Video DMA or the AXI DMA in 2D mode 19
2.5 AXI channel architecture for reads [3] 21
2.6 AXI channel architecture for writes [3] 22
2.7 Overview of Cube DMA core 24
2.8 Order of processing in the Cube DMA for a block-wise transfer with blocks of size 4×4 and $N_c = 4$ planes, starting at an initial offset. . . . 25
2.9 Example of a HSI cube whith spatial dimensions 10x10 and block size 4x4 26
2.10 Relation between configuration register fields and HSI cube 31

2.11 Example of packed stream coming from memory and the resulting unpacked stream when $W_c = 12$ and $N_c = 4$. Left: the stream of packed 64-bit words from memory. Right: the unpacked stream with N_c components in parallel. 32
2.12 Overview of the MM2S unpacker module 32
2.13 The stream from the shifter, given the input shown to the left in Figure 2.11 33
2.14 The stream from the restructurer, given the input stream from the shifter shown in Figure 2.13 34
2.15 Overview of the controller module 35
2.16 State transition diagram for the state machine 37
2.17 Overview of the TinyMover core 38
2.18 The Zynq-7000 Generic Interrupt Controller (GIC) [2] 40
2.19 Memory hierarchy in Zynq-7000 42

3.1 Control bits 46
3.2 Example of packing when $W_c = 12$ and $N_c = 4$. Left: stream from accelerator with N_c components in parallel. Right: the packed stream of 64-bit words. 48
3.3 The component packer 48
3.4 Component joiner behavior for one set of cycles when $W_c = 10$ and $W = 64$ 50
3.5 Descriptor setup for doing block transfers with AXI DMA 52

4.1 MM2S channel LUT usage for different bits per component W_c, as a function of the number of components in parallel N_c. 57
4.2 MM2S channel register usage for different bits per component W_c, as a function of the number of components in parallel N_c 57
4.3 S2MM channel packer LUT usage for different bits per component W_c as a function of the number of components in parallel N_c 59
4.4 S2MM channel packer register usage for different bits per component as a function of the number of components in parallel 59

5.1 CCSDS123 compressor overview [4] 62
5.2 Neighboring samples in same band 64
5.3 Neighbors used in local sum calculations in neighbor-oriented and column-oriented modes [4] 65
5.4 Prediction neighborhood in spatial and spectral dimensions [4] 65

6.1 Overview of the CCSDS123 BIP implementation. Bold arrows show the main data path. Each box represents a VHDL module 79
6.2 Scheduling of pipeline operations. The dashed bars indicate clock cycles. Packing into 64 bit output words takes a variable number of clock cycles depending on the size of encoded words. 79
6.3 Sample delay FIFOs. Each box represents a FIFO with a depth of exactly the number shown on each box. 80

6.4 Local sum, local difference and central difference calculations 81
6.5 Local difference store 82
6.6 The state of the weight store when current input sample is $s_{N_z-3}(t)$ . . . 83
6.7 Timing diagram of pipeline operations from reading a weight vector until writing back the updated weight vector 84
6.8 Dot product when $C_z = 4$ 85
6.9 Dot product when $C_z = 3$. The dashed multiplication and sum are removed by the synthesis tool during elaboration. 85
6.10 Predictor implementation 85
6.11 Weight update implementation 87
6.12 Residual mapping implementation 87
6.13 Overview of sample adaptive encoder. The dashed lines indicate the divide between different pipeline stages 89
6.14 Illustration of the packing of variable length code words into fixed-size packets 90
6.15 The top level diagram for the CCSDS123 IP module 91
6.16 Sample placement in lanes when $N_z = 8$ and $N_p = 4$ 92
6.17 Sample placement in lanes when $N_z = 9$ and $N_p = 4$ 92
6.18 Overview of the parallel CCSDS123 implementation when number of pipelines is 4 94
6.19 Overview of pipeline architecture 94
6.20 Sample delay in parallel CCSDS123 implementation. The actual routing might be different than shown, depending on the value of N_z and the number of pixels Δt. 95
6.21 Example of sample delay to obtain W neighbor samples when $N_p = 4$ and $N_z = 9$ 96
6.22 Example of sample delay to obtain W neighbor samples when $N_p = 4$ and $N_z = 11$ 96
6.23 Routing of central differences between pipelines when $N_p = 4$ and $P = 5$ 98
6.24 Implementation of the shared store module 99
6.25 State of the shared store when used as weight store, when $N_z = 61$ and $N_p = 4$. The figure to the left shows the initial state after reset, while the right figure shows the state when the last samples of pixel 0 and the first samples of pixel 1 are arriving 99
6.26 Implementation of the shared store module 100
6.27 Implementation of variable length word packer 103
6.28 Implementation of combiner chain used in packing 104
6.29 Operation of the variable length word packer 105
6.30 Implementation of improved variable length word packer 107
6.31 Memory utilization when using separate block set FIFOs 108
6.32 Memory utilization when using one block set FIFO 108

7.1 Compression performance for CCSDS123 and JPEG 2000 110
7.2 LUT usage in total and in dot product, predictor and weight update, for different values of P 114

7.3 Register usage in total and in dot product, predictor and weight update, for different values of P . . . 114
7.4 LUT usage for different values of sample width D . . . 115
7.5 LUT usage in dot product, predictor and weight update, for different values of sample width D . . . 116
7.6 Block RAM usage for different values of sample width D . . . 116
7.7 Resource utilization on Zynq Z-7035 . . . 117
7.8 Resource utilization on Zynq Z-7020 . . . 119
7.9 Ratio of number of LUTs and registers used by pipeline logic by the total number of LUTs . . . 120
7.10 Total LUT usage for weight store, accumulator store and sample delay for parallel implementation . . . 121
7.11 Total register usage in weight store, accumulator store and sample delay for parallel implementation . . . 121
7.12 LUT usage for different block sizes when processing different number of words per combiner chain, when $N_p = 4$ . . . 122
7.13 LUT usage for different block sizes and different maximum variable word lengths $U_{\max + \mathrm{D}}$ when $N_p = 4$ . . . 122
7.14 Worst negative slack (WNS) for different number of pipelines . . . 124
7.15 Power estimates for different N_p. Stores refer to the sum of the power used in the weight, accumulator, sample and local difference stores. . . . 125
7.16 Dynamic power as percentage of total power usage . . . 126

8.1 Test bench used for Cube DMA testing . . . 130
8.2 Overview of automatic verification of design . . . 131
8.3 ZedBoard development board used for hardware testing [5] . . . 134
8.4 Overview of Zynq 7000 system for testing Cube DMA and CCSDS123 implementation . . . 135
8.5 The stream monitor module . . . 136

A.1 Searching for waveform value in Vivado simulator . . . 151

List of Tables

1.1 Mapping from cube to one-dimensional coordinates for different orderings 5

2.1 Comparison of capabilities of considered DMA solutions 20
2.2 Input and output ports in Cube DMA 27
2.3 Generic parameters for Cube DMA 27
2.4 Helper values that must be computed in software and given to the core through the register interface 28
2.5 Register layout for the MM2S channel of the Cube DMA 29
2.6 Register layout for the S2MM channel of the Cube DMA 30

3.1 Overview of component joiner cycles 49
3.2 Overview of component joiner cycles 50

4.1 Parameters used for performance comparison 54
4.2 Comparison of performance for AXI DMA and Cube DMA for different transfer types on a HSI cube of size $500 \times 2000 \times 100$, block size 8×8. 54
4.3 Area usage of modules and IPs used in Cube DMA whose area is independent of generic parameters 55
4.4 MM2S channel controller and unpacker area utilization at varying component widths and number of components in parallel 56
4.5 S2MM channel packer area utilization at varying component widths W_c and number of components in parallel N_c 58

5.1 Selectable parameters for the CCSDS123 compressor 71

6.1 Number of samples from the current sample to the previous samples. . . 74
6.2 Number of samples from the current sample to sample from the same pixel in the previous band 75
6.3 Memory usage comparison between sample orderings 75
6.4 Previous CCSDS123 implementations 76

7.1 Default CCSDS123 parameters used when analyzing utilization, power and performance 111
7.2 LUT and register usage for different number of previous bands P 113

7.3 LUT and DSP usage for different sample widths D 115
7.4 Register usage for different sample widths D 115
7.5 LUT and register usage for different choices of weight resolution 115
7.6 LUT and register utilization in pipelines, sample delay, accumulator store, weight store and packer for different N_p 118
7.7 Register utilization in pipelines, sample delay, accumulator store, weight store and packer for different N_p 118
7.8 BRAM and DSP usage for different N_p 119
7.9 Resource utilization needed to compress images from different sensors, with $N_p = 4$ 123
7.10 Power usage for different N_p 125
7.11 Summary of previous CCSDS123 implementations and the proposed implementation 127
7.12 Performance comparison of CCSDS123 implementations 127

8.1 Register layout for the stream monitor module 136

Abbreviations

ACP	Accelerator Coherency Port
AVIRIS	Airborne Visible / Infrared Imaging Spectrometer
AXI	Advanced Extensible Interface
BIL	Band Interleaved by Line
BIP	Band Interleaved by Pixel
BRAM	Block Random Access Memory
BSQ	Band Sequential
CCD	Charge-coupled Device
CCSDS	Consultative Committee for Space Data Systems
CPU	Central Processor Unit
DDR	Double Data Rate
DMA	Direct Memory Access
DSP	Digital Signal Processor
FIFO	First In First Out
FIQ	Fast Interrupt Request
FPGA	Field Programmble Gate Array
GIC	Generic Interrupt Controller
GPU	Graphics Processor Unit
GUI	Graphical User Interface
HICO	Hyperspectral Imager for the Coastal Ocean
HSI	Hyperspectral Image / Imaging
ILA	Integrated Logic Analyzer
IP	Intellectual Property
IRQ	Interrupt Request
JSON	JavaScript Object Notation
JTAG	Joint Test Action Group
LUT	Lookup Table

MM2S	Memory Map to Stream
MODIS	Moderate Resolution Imaging Spectroradiometer
OTFP	On-the-fly Processing
PL	Programmable Logic
PS	Processing System
RAM	Random Access Memory
S2MM	Stream to Memory Map
SCU	Snoop Control Unit
SDK	Software Development Kit
SoC	System on Chip
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WNS	Worst Negative Slack
XSDB	Xilinx System Debugger

1

Introduction

1.1 The NTNU SmallSat project

The oceans, covering 70% of the Earth surface area, are important parts of the global environment, with their function as sinks for green-house gases and the environment for marine life and resources. Facing the challenge of climate change, study of the oceans from a fine scale (micro-biology) to a large scale (hurricanes, ice melt, harmful algal blooms, fronts) is important. Traditionally, ship-based measurements have been the norm, but these have several draw-backs, including the need for extensive engineering and science infrastructure, subjecting people to harsh seafaring conditions and providing only point measurements for phenomena that are spread across large areas [1].

An alternative approach is envisioned in the Autonomous Ocean Sampling Network (AOSN), where a network of autonomous underwater vehicles (AUVs), autonomous surface vehicles (ASVs) and unmanned aerial vehicles (UAVs) is capable of coordinated missions that are executed together with conventional vehicles, buoys and fixed sensor networks [6]. The benefits are significant reductions in cost and increased safety, more information as well as more continuous information.

The NTNU SmallSat project's focus is the development of a small satellite (SmallSat) which is to be part of a proposed AOSN called a *multi-agent marine observation system*, which is illustrated in Figure 1.1. The system is a *cyber-physical* system where the different components are tightly knit together by communication technology and intelligent information processing [1]. The role of the SmallSat that is developed in the project is to provide *hyperspectral imaging* (HSI) capabilities to this system.

One example where satellite hyperspectral imaging is of utility is in detecting algae blooms. This is of interest beacuse some blooms generate neurotoxins that have signif-

icant impacts on coastal marine and human populations [1]. Due to the spatial vastness of such algae blooms, satellite imaging is particularly suited for tracking such activity in the ocean. A satellite image of an algae bloom is shown in Figure 1.2.

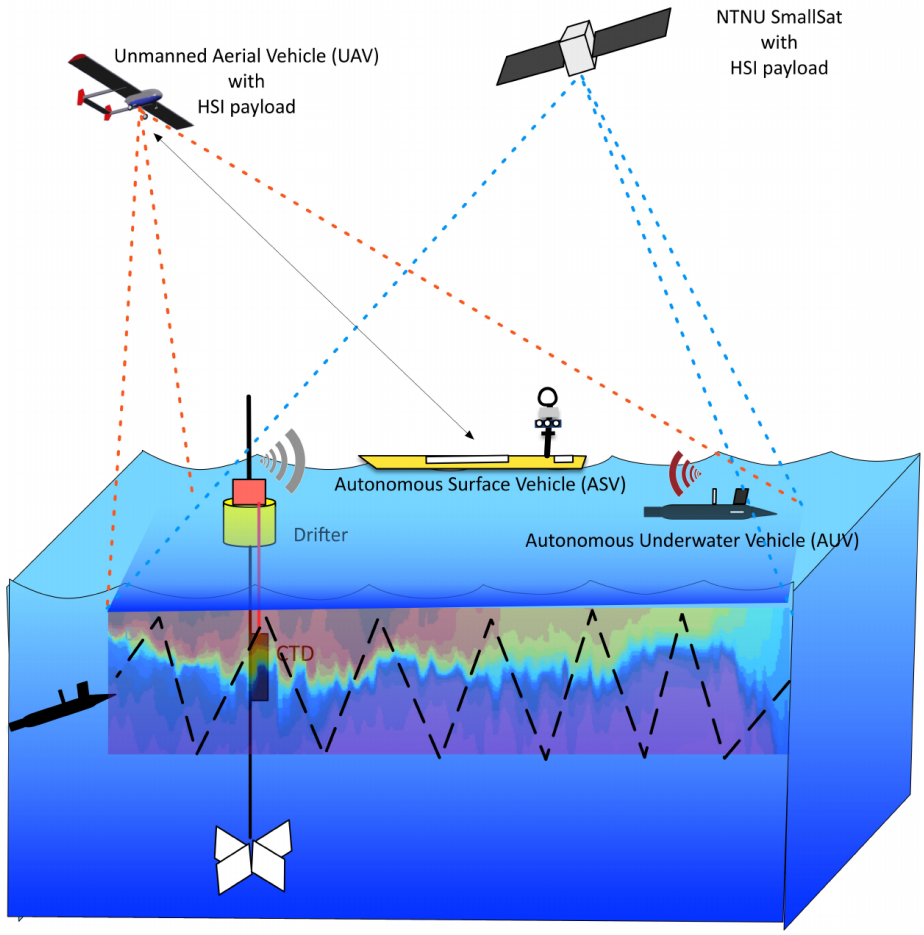

Figure 1.1: The main components of the proposed multi-agent marine observation system [1]

Figure 1.2: Phytoplankton bloom at the coast of Norway, observed from space [1]

1.2 Hyperspectral imaging

Hyperspectral imaging refers to digital imaging where the spectrum of the incoming light at each pixel is sampled at many different wavelengths across a wide range of the spectrum, typically more than hundred samples taken across wavelengths from the near-infrared to beyond visible light. Each pixel thus has an approximation of the spectrum of light reflected from the corresponding location in the scene that is imaged. This extensive spectral information makes it possible to detect objects and materials with much greater precision than with conventional color imaging.

A hyperspectral image is often called a hyperspectral image *cube* (HSI cube) due to its three-dimensional structure with two spatial dimensions and one spectral dimension. The cube consists of samples, also called components, with a spatial coordinate (x, y) and a spectral coordinate z. The set of components at a fixed spatial coordinate is a pixel in the image. This is illustrated in Figure 1.3 for a cube with spatial dimensions 8×8 and a spectral dimension of 4.

The HSI cube can also be viewed as a series of two-dimensional images, one for each of the sampled spectral wavelengths. These are called *planes*. Figure 1.3 highlights one plane in the HSI cube.

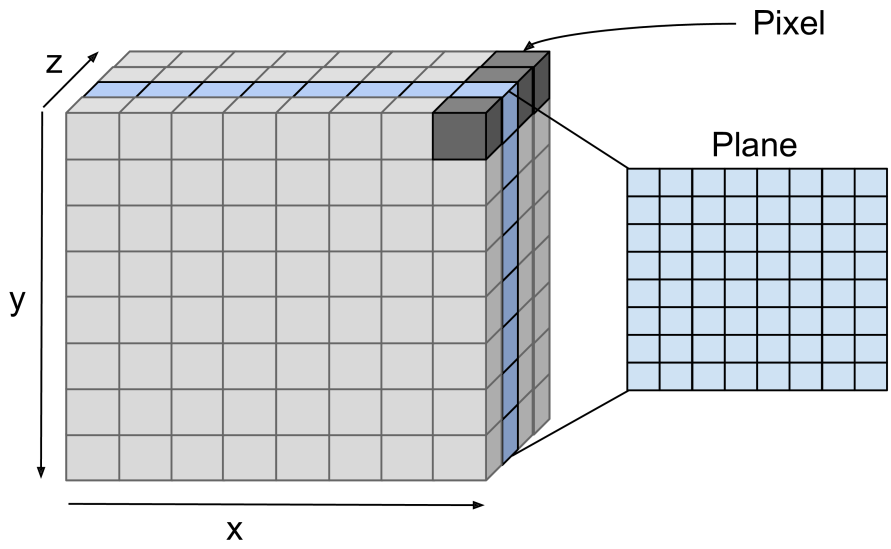

Figure 1.3: Hyperspectral image cube

1.2.1 Component orderings

A HSI cube is a three dimensional structure, so streaming it or storing it in memory requires the components to be ordered serially by defining a mapping from the three dimensional coordinates to a unique one dimensional index. The most common ways of ordering the components are Band Interleaved by Pixel (BIP), Band Interleaved by Line (BIL) and Band Sequential (BSQ). Figure 1.4 illustrates these orderings for an image of size $4 \times 4 \times 4$.

In BSQ ordering, the components are ordered such that all the components in the first band, from the upper-left pixel to the lower-right pixel are followed by all the components in the second band from the upper-left pixel to the lower-right pixel, and so on.

In BIL ordering, the components are also ordered separately for each band, but only for each line. The first component of each pixel in the first line are followed by the second component of each pixel in the first line, and so on. This pattern is repeated for the rest of the lines in the cube.

In BIP ordering, all the components of a pixel are contiguous, meaning that all the components of the uppler-left pixel are followed by all the components of the pixel to the right, and so on, all the way to the lower-right pixel.

Table 1.1 shows how cube coordinates are mapped to one dimensional indices for these three orderings.

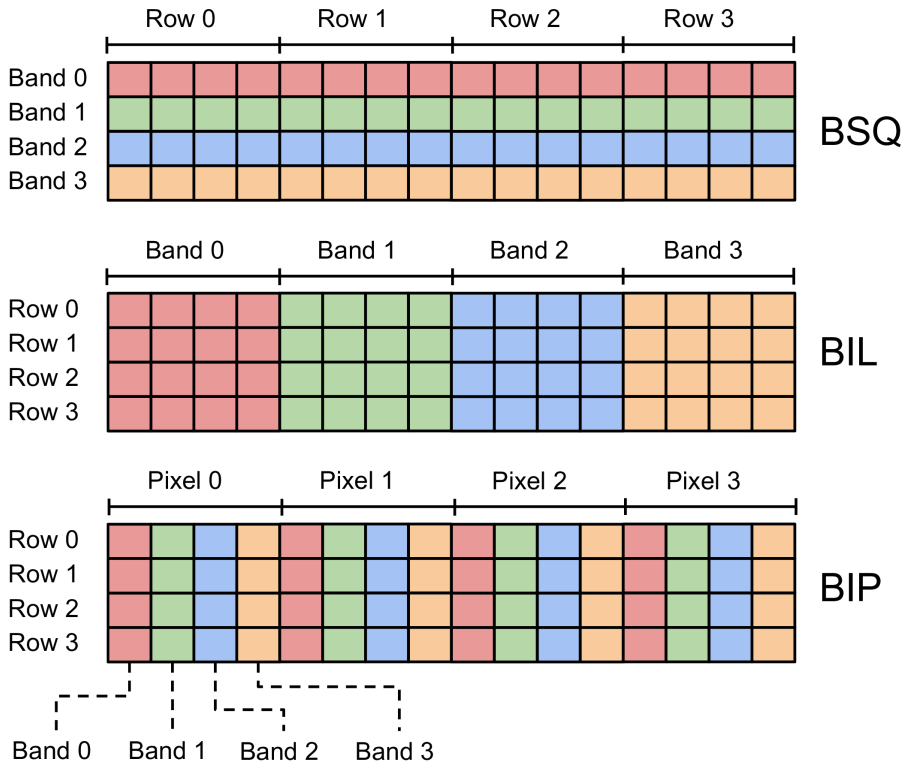

Figure 1.4: Different sample orderings in hyperspectral images, illustrated with an example image of size $4 \times 4 \times 4$

Name	Mapping
Band Interleaved by Pixel (BIP)	$i = yN_xN_z + xN_z + z$
Band Interleaved by Line (BIL)	$i = yN_xN_z + zN_x + x$
Band Sequential (BSQ)	$i = zN_xN_y + yN_x + x$

Table 1.1: Mapping from cube to one-dimensional coordinates for different orderings

1.3 SmallSat HSI payload

One of the current areas of focus in the NTNU SmallSat project is the development of a hyperspectral imaging payload that is capable of capturing and processing of hyperspectral images. The camera payload has so far been specified, and prototypes are being developed. The idea is that hyperspectral images are not only acquired in the satellite, but also processed in several steps before data is sent to the ground station.

Figure 1.5 shows the proposed processing architecture, and how it is controlled from the ground station. After raw image acquisition, the geometric and radiometric image processing steps transform each pixel into geographic locations on the ocean surface, and each of the samples into absolute reflectance values that are calibrated using measurements of the atmosphere, solar radiation and so on. The spectral and spatio-temporal steps remove undesirable optical features such as water reflections or shadows due to clouds, and enhances spatial resolution by applying deconvolution techniques to consecutive frames [1].

At some point, data needs to be sent to the ground. Due to bandwidth limitations in radio links as well as the desire for shortest possible download time, it is important that image data is compressed, while also retaining the most important information.

The HSI payload consists of an HSI push broom camera connected to a Xilinx Zynq-7000 APSoC which performes the processing. The HSI payload itself is in turn connected through a *Cubesat bus* to the rest of the satellite.

1.3.1 HSI camera

The camera used in the HSI payload uses a technique called *push broom scanning* to record the image. An overview of push broom scanning is shown in Figure 1.6. The satellite is capturing the image line by line as it moves across the area of interest, as seen to the left of the figure. Using optics, the incoming strip of light is separated spectrally across a 2-dimensional CCD image sensor array. The image captured by the sensor array has one spatial dimension, indicating the different locations (pixels) in the imaged scene, and one spectral dimension which indicates light intensities sampled at the different wavelengths. Each such captured line of pixels is called a *frame*, and can be viewed as a 2-dimensional image with a spatial dimension and a spectral dimension.

When the capturing of one line has completed and the satellite has moved further on, a new line of the scene can be captured. Relating this to Figure 1.3, the resulting HSI cube is built up from top to bottom. Captured frames are read out from the sensor using Low Voltage Differential Signalling (LVDS).

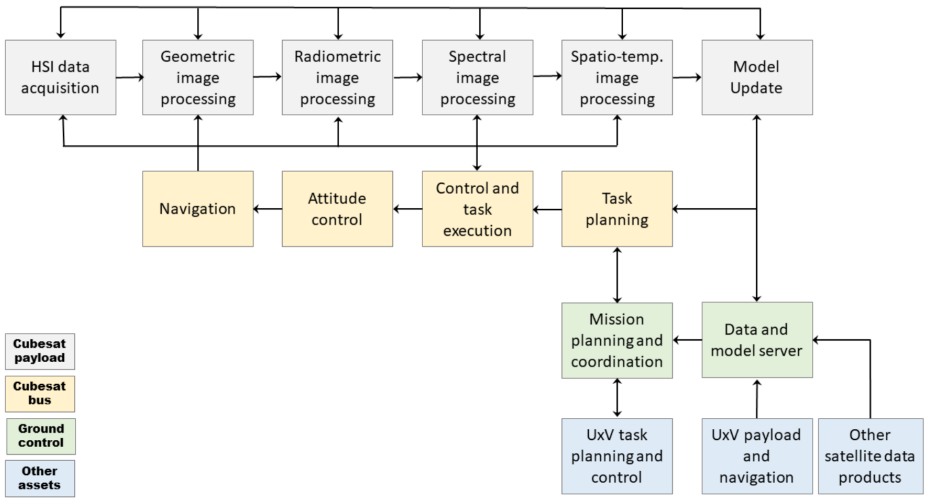

Figure 1.5: The HSI payload processing and control architecture [1]

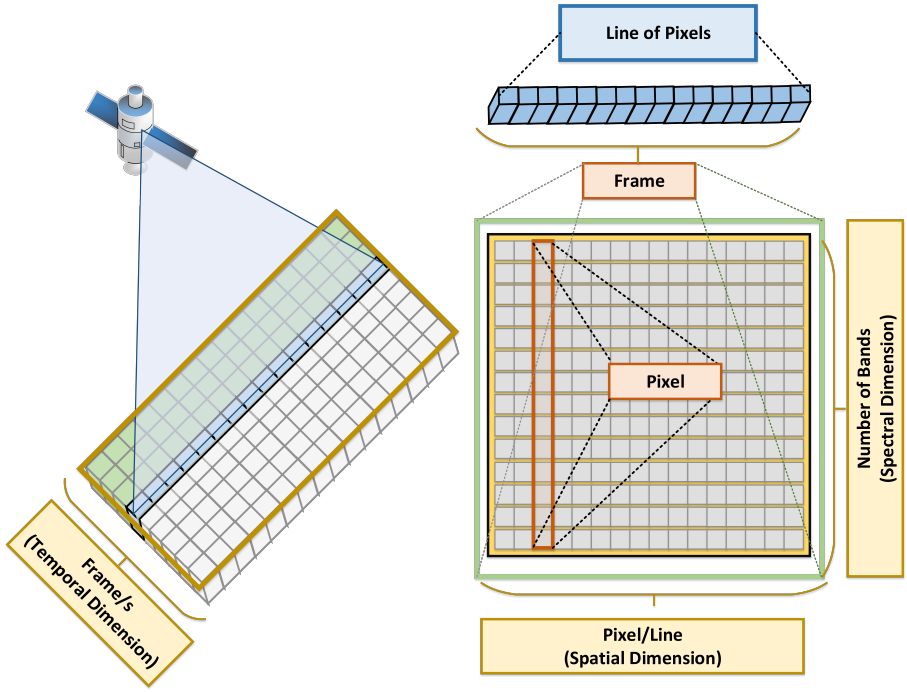

Figure 1.6: Hyperspectral image acquisition process using push broom scanning

The captured raw image is commonly referred to as a level 0 (L0) image. Various corrections are applied to this image to account for systematic errors in the sensor system, to provide a level 1 (L1) image. From this L1 image, further corrections for atmospheric conditions can be performed to produce a level 2 (L2) image [7]. The distinctions between these levels are not important for this thesis, but will be used when discussing compression performance in Chapter 7.

1.3.2 Zynq 7000-series All Programmable System on Chip

The Xilinx Zynq 7000 is a family of APSoC (All Programmable System on Chip) from Xilinx that combine ARM processor cores with FPGA technology. This makes them attractive for use in systems where tight integration between hardware and software is important. In the HSI payload, the image sensor is connected through LVDS directly to the FPGA, so that the captured frames can be processed and stored in memory.

An overview of the Zynq 7000 architecture is shown in Figure 1.7. The two main parts of the system are the Processing System (PS), consisting of the CPU cores, on-chip memory and peripherals, and the Programmable Logic (PL) which is the FPGA. The architecture is quite intricate, but for the scope of this thesis, the following parts are the important parts:

- Programmable Logic (PL): The FPGA
- Application Processor Unit (APU): Consists of the ARM Cortex A9 CPU cores, cache and on-chip memory as well as timers, interrupt control, etc.
- Memory Interface: Connects to external DDR memory
- Central Interconnect: Provides flexible connections between the APU, PL, I/O peripherals and memory
- PL to Memory Interconnect: Provides the PL with high-speed connections to the on-chip and external memories

Various kinds of buses connect the different components in the system via the Central Interconnect and the PL to Memory Interconnect. In this thesis, the focus will be on communication between the processors in the APU, the external DDR memory and the PL. These are all connected using the AXI bus standard, which will be detailed in the next chapter.

The CPUs connect to the rest of the system through AXI buses connected to the Central Interconnect, and also directly to the DDR memory interface. Via the Central Interconnect, the CPUs can access the PL using the general purpose (GP) ports, as shown in Figure 1.7. The GP ports are 32 bit wide, and are typically used for accessing control registers of soft cores that are instantiated in the PL.

From the view of the PL, the rest of the system is available through the Central Interconnect via two GP ports, and the on-chip and external DDR memory are available through four high-performance (HP) ports. The high performance ports are 64-bit wide and are optimized for high data bandwidths.

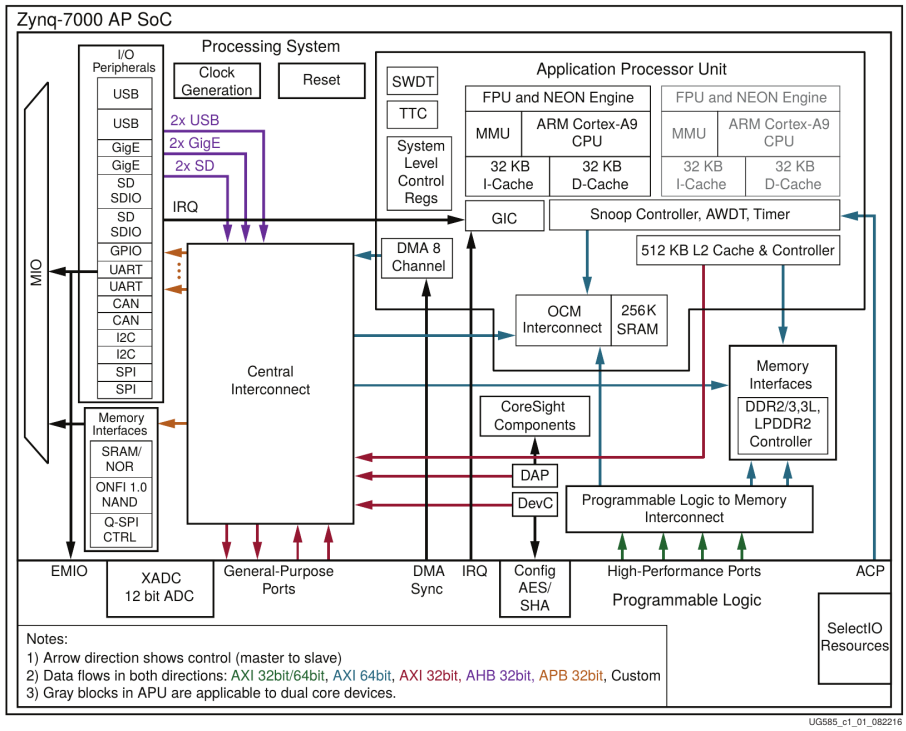

Figure 1.7: Overview of the Zynq-7000 architecture [2]

1.3.3 Integration of hardware and software

One strength of FPGAs is the ability to interface directly with external devices and process data *on the fly* as it arrives. This avoids the need for storing data in memory before starting to process it, which is often the case with other processing platforms (for instance GPUs). In the SmallSat HSI payload, data from the image sensor is streamed directly into the FPGA, which can then perform several processing steps before data is stored in DDR memory.

Figure 1.8 shows the typical setup for this kind of on the fly processing. Data arrives from the source, in this case the HSI camera sensor, and is processed directly in the FPGA before being sent to the DDR memory using a *direct memory access* (DMA) core. The DMA core takes care of issuing write transactions to the memory. The high performance (HP) ports are used to provide maximal data bandwidth performance. The CPU has access to configuration registers in the DMA core and in the hardware processing core through connections to the Central Interconnect via one of the general purpose (GP) ports in the PL.

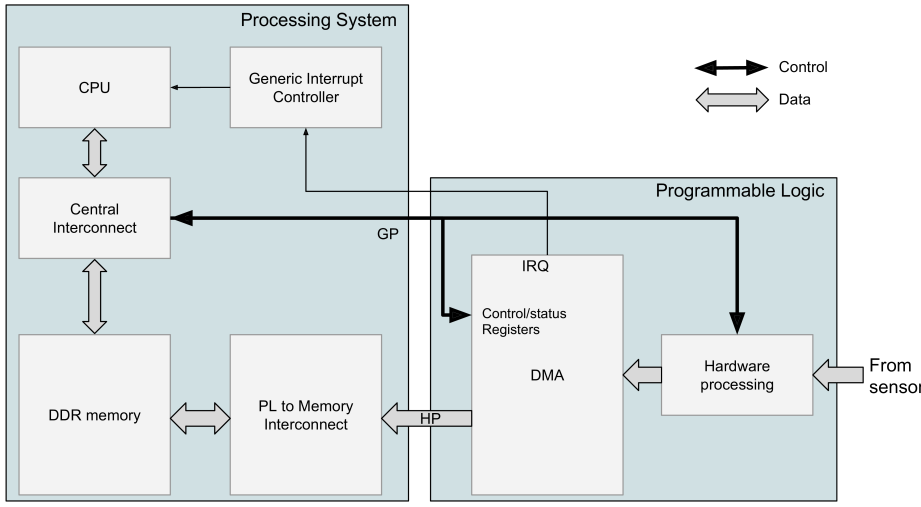

Figure 1.8: Typical setup for on the fly processing

Another common use of FPGAs is to offload parts (or all) of software algorithms to custom processing logic in the FPGA that can take advantage of massive data parallelism or perform certain operations quicker than the CPU while consuming less power. This is typically done by storing the data to be processed in DDR memory and then using a DMA core to transfer data from the memory and into the hardware processing core, and collecting the results and storing them back in memory, similarly to in the previous case. This is illustrated in Figure 1.9.

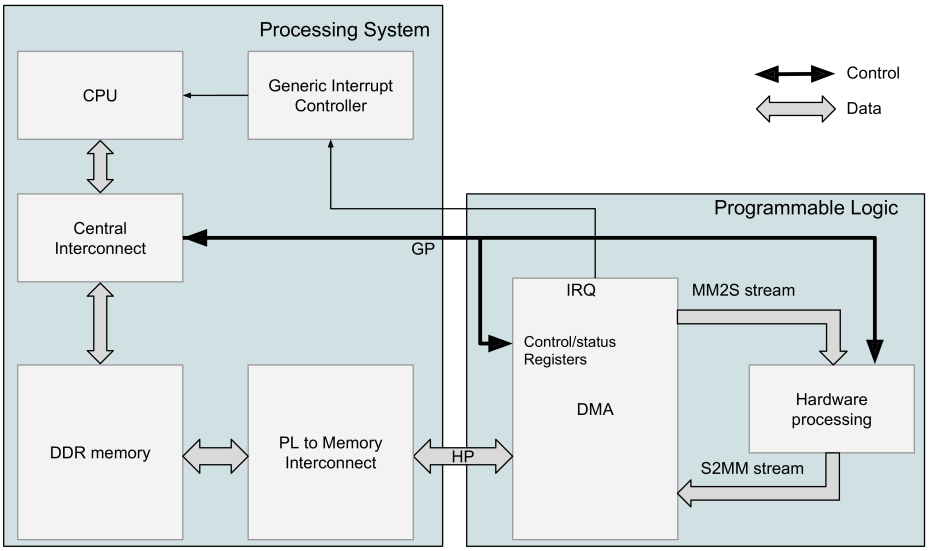

Figure 1.9: Typical architecture of a hardware/software system in the Zynq-7000

An important and limiting factor in the effectiveness of such hardware accelerations is the obtainable data throughput to and from the memory. This depends on primarily on the bus widths and clock frequencies used, but also on the effectiveness of the DMA cores that are used.

1.4 Main contributions

The project assignment printed at the beginning of this thesis document lists two points:

- Establish complete dataflow for OTFP pre-processing
- Implement CCSDS123 – BSQ and BI processing order, test communication with the rest of the system

The following paragraphs will explain how these two points are addressed.

A Direct Memory Access (DMA) core, the *Cube DMA*, has been created specifically for streaming hyperspectral images into and out of hardware processing cores. The Cube DMA is meant to take the place of the DMA block that is seen in Figures 1.9 and 1.8. Its main role is to provide hardware processing cores with HSI cube data streamed in different orderings depending on the needs of the particular core. Some processing cores might require BSQ ordering of data that is stored in BIP order, while others might need the cube data ordered in a blocks-wise fashion. The Cube DMA can be configured to accommodate different combinations of BIP or BSQ ordering and sequential or block-wise ordering. The OFTP pre-processing mentioned in the first point of the project

assignment is one of several processing algorithms that are of interest to perform in FPGA hardware in the satellite. The Cube DMA is flexible enough to support this core as well as other processing cores. As such, the first point in the project assignment has been expanded upon in this thesis.

CCSDS123 is a lossless compression algorithm for hyperspectral images that takes advantage of the 3D structure of HSI cubes. An efficient FPGA implementation of this algorithm has been developed for compressing image data in BIP order. The core has been thoroughly verified. The CCSDS123 implementation can also be viewed as an application for the Cube DMA core, as it is one of many possible hardware processing cores that can be used in the way depicted in Figure 1.9 (compressing image data already in memory) and Figure 1.8 (compressing image data straight from image sensor). The second point of the project assignment also mentions BSQ processing order. As it will be detailed in Chapter 6, the choice of sample ordering has no effect on compression performance, and in addition, the hyperspectral camera that is being developed for the SmallSat project is using BIP ordering. The scope of this point of the assignment has therefore been narrowed to only focusing on implementing CCSDS123 for BIP ordering.

The title of the thesis has been changed from the assignment title that is shown at the first page of this thesis document. During the work on this thesis, efficient streaming and compression has stood out as the two main themes throughout, and while the streaming part of the work deals with communication between peripherals, it has little to do with the ZedBoard development board in particular. The title ”Efficient Streaming of Hyperspectral Images” has therefore been chosen instead.

Two journal papers have been submitted based on this work:

- J. Fjeldtvedt, M. Orlandić, ”CubeDMA - Optimizing Three-Dimensional DMA transfers for Hyperspectral Imaging Applications”, Microprocessors and Microsystems Journal, Second round of review, 2018
- J. Fjeldtvedt, M. Orlandić, T. A. Johansen, ”An Efficient Real-Time FPGA Implementation of the CCSDS-123 Compression Standard for Hyperspectral Images”, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, Second round of review, 2018

A third paper detailing the parallelization of the CCSDS123 implementation is planned for submission during 2018.

1.5 Structure of this thesis

The work done in this thesis has been focused on two separate problems, and hence the thesis will be split in two parts. Three chapters are dedicated to the Cube DMA core:

- **Chapter 2** will present the findings of the feasibility study that was conducted prior to this thesis, and present the Cube DMA implementation that was performed

then. It will also introduce some background details about interrupts and caches in the Zynq-7000 SoC, as well as the AXI bus standards.
- **Chapter 3** details the continued work on the Cube DMA done for this thesis.
- **Chapter 4** presents results for the Cube DMA and compares performance to another DMA core from Xilinx.

The next three chapters cover the CCSDS123 compression standard:

- **Chapter 5** presents a detailed go-through of the CCSDS123 compression standard and introduces other background information that is necessary to understand the implementation.
- **Chapter 6** discusses various trade-offs to consider when implementing the algorithm, presents and discusses previous implementations, and covers in detail the hardware implementation of the CCSDS123 algorithm.
- **Chapter 7** presents results for the CCSDS123 implementation and compares it to previous implementations.

Verification performed of the hardware implementations of the Cube DMA and the CCSDS123 algorithm is detailed in **Chapter 8**. Some conclusions and notes on future work will be presented in **Chapter 9**.

2

Cube DMA: A DMA core for hyperspectral images

This chapter will introduce the Cube DMA, which was partly implemented in the semester project leading up to this master's thesis. First the feasibility study that was conducted will be summarized, motivating for the need for the Cube DMA. This is followed by some background information regarding the AXI bus standards that are used, and then an overview of the Cube DMA core, showing its architecture, port interfaces and generics, and register interface for configuration. Following this, a more in-depth overview of the implementation of the MM2S channel will be shown. Lastly, some necessary information about interrupts and cache coherency in the Zynq-7000 system will be presented.

2.1 Introduction

Hyperspectral images are processed in different ways by different processing algorithms. Some might process one plane or band in the cube at a time (BSQ ordering), while others might divide the image into blocks, or do a combination of the two. This means that cube data must be streamed in different orders depending on the application. A common way to achieve this is to arrange the image data in memory in such a way that when streamed sequentially, the data is ordered in the desired way. This is problematic in a typical satellite system, because the ordering of the components is already fixed when captured by the camera sensor.

In addition, several different hardware processing algorithms operating with different component orderings might be doing processing on the same data in memory. Changing

the data layout at run-time is a costly operation both in terms of time and the space needed for temporary storage during such a conversion.

Another approach is to let the component ordering in memory stay the same, and instead change the *access pattern* when reading from memory. For instance, if the image is stored in BIP format in memory, streaming in BSQ order can be achieved by reading only the first component in the first pixel, skipping the rest of the components and reading the first component in the second pixel, and so on. This was explored in my feasibility study prior to this master's thesis work [8].

2.1.1 Summary of feasibility study findings

At the outset of the feasibility study, the following specifications for the DMA core were set:

1. Capability of streaming a HSI cube (stored in BIP format) in BIP and BSQ order
2. Capability of streaming a HSI cube block-wise
3. Support for components of sizes that are not byte multiples, e.g. 10 or 12 bits

The two first requirements have to do with the issues described in the introduction, namely that different HSI algorithms might require components to be streamed in BIP or BSQ order, and also sequentially or in blocks. A DMA core for use in hyperspectral imaging should therefore support all of these streaming orders. The capability of doing block-wise transfers is important for several compression algorithms, such as JPEG and JPEG2000, where the image is divided into blocks and processed block-wise. Figure 2.1 shows in more detail what is meant by a block-wise transfer: The components are streamed starting at the top left pixel of the block and ending at the lower right pixel in the block, as the arrow in the figure illustrates. The blocks themselves are processed in the same order, starting at the upper left block and ending at the lower right block. Figure 2.2 shows block-wise transfer in BSQ order, where each plane is streamed separately.

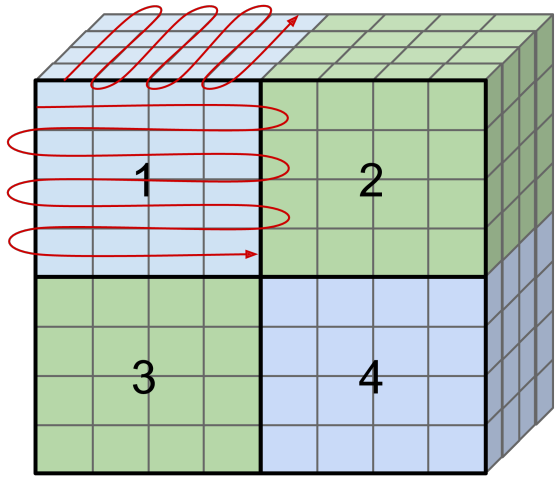

Figure 2.1: Block-wise streaming of a HSI cube in BIP order. The arrows indicate the order in which the components are streamed from a block, and the numbers indicate the order in which each block is streamed.

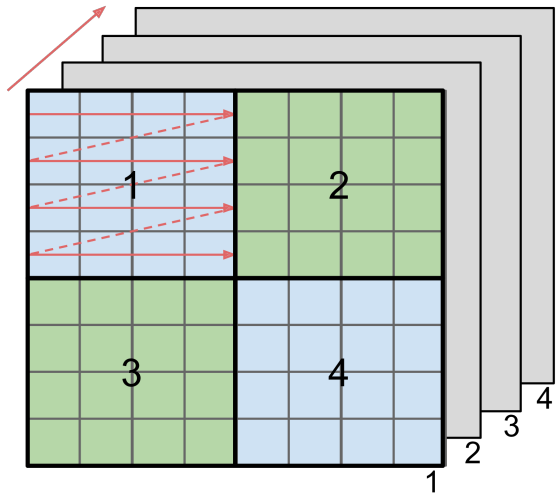

Figure 2.2: Block wise streaming of a HSI cube in BSQ order. The arrows indicate the order in which the components are streamed from a block, and the block numbers indicate the order in which each block is streamed. The numbers on each plane indicate the order in which each band is streamed.

The third requirement has to do with typical component sizes used in image sensors. While the native sizes in a typical memory system are byte multiples, the image sensor

selected for the SmallSat project at this time is capable of sampling the image with 10 or 12 bits per component. To optimize memory utilization and streaming bandwidths, these should be stored in memory with the same number of bits and not padded to use e.g. 16 bits.

In the study, three existing DMA IP cores from Xilinx were considered: The Xilinx AXI DMA, the Xilinx Video DMA and the Xilinx DataMover. For all three cores, there is no built-in support for handling data elements that have widths that are not byte multiples. As such, the third requirement requires some extra logic to be developed no matter which core is used. The following sections will go through the different cores with focus on the two first requirements.

2.1.1.1 AXI DMA

The AXI DMA [9] is the go-to general purpose DMA solution from Xilinx. It is capable of scatter-gather transfers where stream data is collected from non-consecutive segments in memory, and it uses so-called *block descriptors* to describe such transfers. A block descriptor contains the start address and length of a transfer, and several descriptors can be chained together to describe longer transfers, as Figure 2.3 illustrates. The main drawback of the AXI DMA is that each time there is a gap in the memory access pattern, such as when starting on the next row in a block (and skipping the rest of the data in the row), a new block descriptor must be used. This means that for block-wise transfers with many blocks, there can be an unacceptable number of descriptors needed. That problem can be solved by using fewer block descriptors and instead re-use the same block descriptors. This does however require the CPU to intervene several times during the transfer to set up the block descriptors with new data. BSQ ordered access patterns are not possible for the same reason, since that would require one block descriptor for every component in the image.

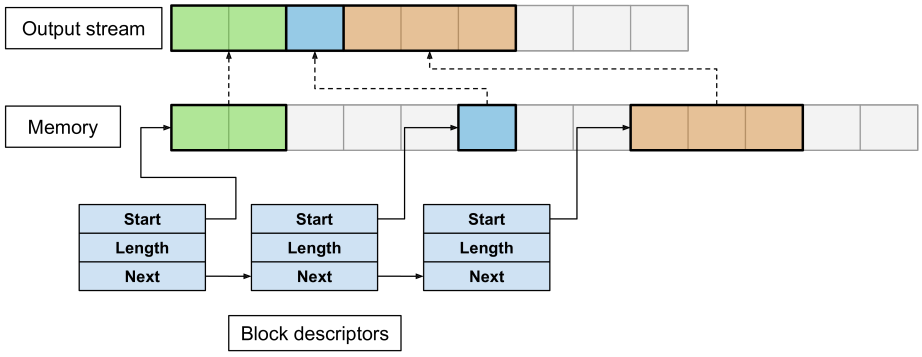

Figure 2.3: Scatter-gather transfer using block descriptors

The AXI DMA can be instantiated in a so-called 2D mode where block-wise transfers

can be performed without needing to specify a new descriptor for each row in the block. Instead of just giving the transfer length, a *stride* is also given, which indicates how many bytes to skip before starting a new transfer of the same length. A *vertical size* parameter is used to indicate how many such transfers to repeat. This is illustrated in Figure 2.4. As an example, doing a transfer of an $8 \times 8 \times 128$ block from a $512 \times 2000 \times 128$ cube would be done by setting the length to $8 \cdot 128$ (the number of components in one row of the block), the stride to $512 \cdot 128$ (the number of components in one row of the whole cube) and the vertical size to 8 (the number of rows in the block).

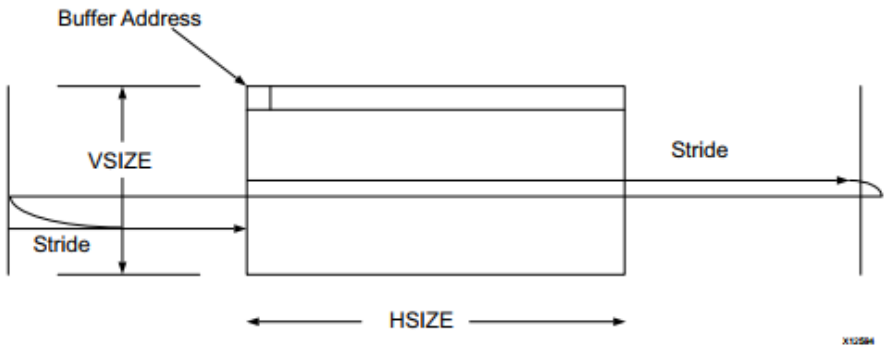

Figure 2.4: Horizontal size, stride and vertical size when using the Video DMA or the AXI DMA in 2D mode

This would be nearly ideal for HSI processing if it was not for two issues: 1) The descriptor fields used for the stride is only 16 bits, limiting the cube width and depth that can be used (the product must be less than 65536) and 2) When instantiated in 2D mode, the start address of any transfer is required to be divisible by 8 bytes, which rules out any bit non-byte aligned component widths and any cubes that have dimensions that are not divisible by 8 bytes.

2.1.1.2 Video DMA

The Video DMA [10] is a more specialized core that is meant for streaming video. Instead of using block descriptors, the Video DMA has a set of 32 fixed frame buffer pointers. The DMA cycles through the frame buffers and streams them either sequentially or block-wise from memory. Block-wise streaming is done in the exact same way as for the AXI DMA in 2D mode, and it also suffer from the same register width limitations, meaning that it can only be used for sufficiently small HSI cubes. The fixed buffer nature of the Video DMA also makes it hard to do an automated block-wise transfer, as it would require the CPU to manually start new transfers after each block has been completed.

The capabilities of the different DMA solutions are summarized in Table 2.1.

2.1.1.3 DataMover

The DataMover [11] is a simpler DMA core than the AXI DMA or Video DMA. It accepts command words that contain a start address and the number of bytes to transfer, and it will execute the command and return back a status word when the transfer has been completed. It is lower level than the AXI DMA and Video DMA in the sense that it can only execute the given commands, and not automatically start new transfers like the AXI DMA and Video DMA do. Because configuring the DataMover is done through command words, some additional logic is required to make it controllable from software, for example through a memory mapped register interface. The strength of the DataMover, however, is that it allows very fine grained control of transfers and as such it can be used to do any of the specified transfers if it is used together with logic that is issuing the right commands.

2.1.1.4 Conclusions from the study

The conclusion of the feasibility study was that neither the AXI DMA or the Video DMA cores from Xilinx can be used for HSI processing in all the ways that were specified. It was therefore looked into how to create a special purpose DMA core for hyperspectral images using the DataMover to perform the actual transfers. This led to the implementation of a special purpose DMA IP, the Cube DMA, that meets all the specifications.

Core	BIP	BSQ	Block-wise (BIP or BSQ)
AXI DMA	Yes	No	Yes (BIP)
AXI DMA 2D	Depends	No	Depends (BIP)
Video DMA	Depends	No	Depends
DataMover	With ext. logic	With ext. logic	With ext. logic

Table 2.1: Comparison of capabilities of considered DMA solutions

2.2 AXI bus standards

This section will go through the AXI bus standards, which are used in the Zynq-7000 SoC platform and in all the Xilinx DMA cores, and which is necessary for understanding some of the implementation details of the Cube DMA and TinyMover cores that have been developed in this work.

2.2.1 The AXI bus standards

Advanced eXstensible Interface (AXI) is a set of open bus standards developed by ARM as a part of their Advanced Microcontroller Bus Architecture (AMBA) standard. Two

revisions, AXI3 and AXI4 have up until now been specified. The AXI specifications define three kinds of bus interfaces: AXI (sometimes referred to as full AXI), AXI-Lite and AXI-Stream. These are tailored to different use cases, but have some general principles in common.

An AXI bus connects two components together, where one is a master and the other is a slave. The master initiates all communication, while the slave responds to requests made by the master. There is always exactly one master and one slave on an AXI bus. To build larger systems where several masters share access to several slaves, *interconnects* provide the logic to route requests from a master to any of the connected slaves. The AXI protocols are used in many of the internal buses in the Zynq-7000, as indicated in Figure 1.7 where the green, blue and red colored buses use AXI. The arrows point in the direction from master to slave.

AXI and AXI-Lite consist of five *channels* (the prefixes used in signal names shown in parentheses):

- Write Address channel (**AW**)
- Write Data channel (**W**)
- Write Response channel (**B**)
- Read Address channel (**AR**)
- Read Data channel (**R**)

The channels are separate and independent from one another. Figures 2.5 and 2.6 illustrate the direction of data flow in these channels.

2.2.1.1 Handshake mechanism

All of the channels use a common handshake mechanism to transfer data from the source to the destination. This is controlled by two signals, **xvalid** and **xready** (where **x** is the prefix for the channel). The **xvalid** signal is asserted by the interface which is sending data to indicate that valid data is present on the data lines. The **xready** signal is asserted by the recipient when it is ready to get the data. Either signal can be asserted first, but

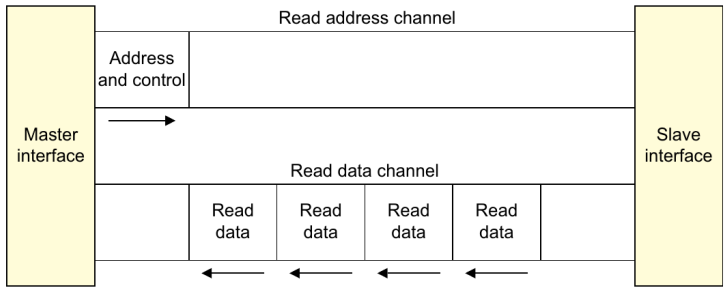

Figure 2.5: AXI channel architecture for reads [3]

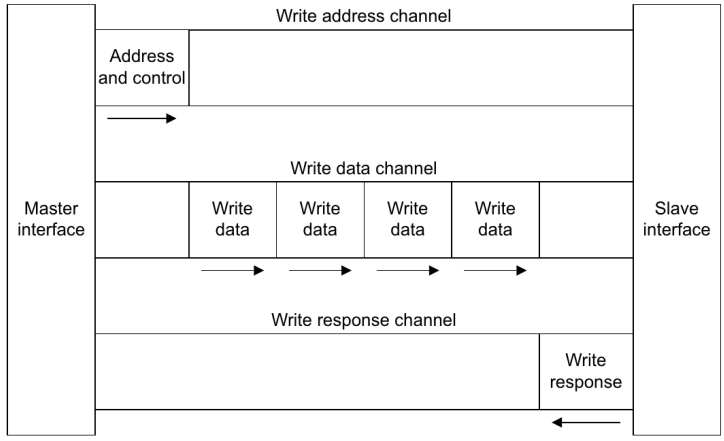

Figure 2.6: AXI channel architecture for writes [3]

in the clock cycle when both are asserted simultaneously, the transfer is said to have occured. The cycle when this happens is called a *beat*. The beat is the fundamental unit of data transfers.

2.2.1.2 Requests and transactions

The master initiates any communication by sending *requests* over the Write Address and Read Address channels, using the described handshaking mechanism. The request contains information about how many bytes to read or write, and the address to read from or write to. A request initiates a read or write *transaction* between the master and the slave. In case of reads, the slave will send the requested data on the Read Data channel, and in the case of writes, the master will send the data on the Write Data channel. The transaction ends when the associated data transfer is finished, which is indicated by the data channel's **xlast** signal being asserted. A slave might accept several requests on the address channel before servicing the first request. This allows highly efficient operation, since the master doesn't have to wait for one transaction to complete before issuing the next.

In addition to any data transfers, the slave also sends a *response* to all requests from the master using a **xresp** signal in the Read Data and Write Response channels. This response tells the master of any errors that might have occured; for instance that an invalid address was contained in the request.

2.2.1.3 Burst transfers

An important performance feature in the full AXI protocol (not AXI-Lite) is the support for *burst transfers*. Instead of initating a new transaction for every unit of data to be transferred, the master may give the slave a *burst size* (**axsize**) and *burst length* (**axlen**) in addition to the address. The burst size indicates how many bytes will be transferred in one beat, and the burst length tells the slave how many beats to expect in the case of writes, or how many beats to transfer to the master in the case of reads.

2.2.1.4 AXI-Stream

AXI-Stream is a slimmed-down protocol for transfers where data is just moved from one point to another, without any concept of addresses. It is modelled after the read and write channels in the AXI protocol. Like for AXI buses, handshaking signals (**tvalid** and **tready**) are used when transferring data, and **tlast** is used to indicate the end of a transfer.

2.3 Overview of the Cube DMA core

An overview of the Cube DMA is shown in Figure 2.7. The Cube DMA consists of two independent channels: The Memory Map to Stream channel (abbreviated to MM2S) which reads data from memory and streams it into an accelerator, and the Stream to Memory Map channel (S2MM) which receives a data stream from the accelerator and stores the incoming data in memory. A common register interface is used to configure both channels.

Both channels are capable of handling component sizes that are not byte multiples. In the MM2S channel, packed data from the memory is unpacked into separate components that are streamed to the accelerator, and in the S2MM channel, components from the accelerator are packed into 64-bit words before being stored in memory.

A specialized core for doing BSQ transfers more efficiently than the Xilinx DataMover was also developed, called the TinyMover. This will be detailed in section 2.6.

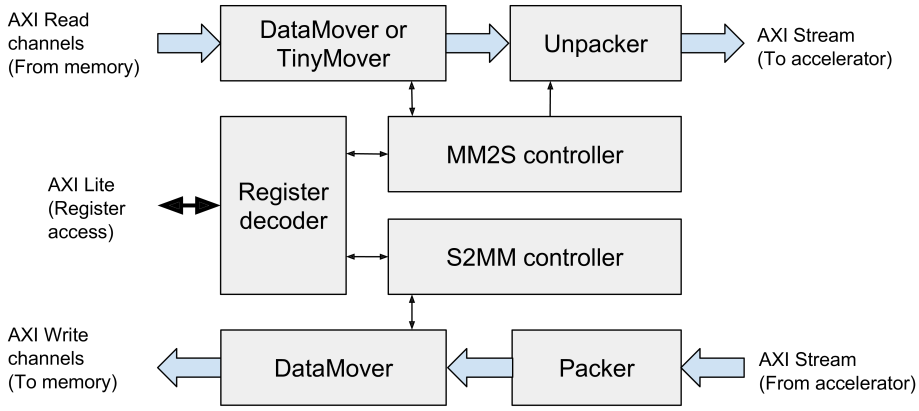

Figure 2.7: Overview of Cube DMA core

2.3.1 MM2S channel

The MM2S channel in the Cube DMA can perform BIP ordered or BSQ ordered transfers, which is selectable at run-time through register configuration. In BSQ ordered mode, components from several planes can be streamed in parallel, meaning that in every beat of the transfer, components from the same pixel but in different planes are output in parallel. In BIP mode, several components from the same pixel can also be transferred in parallel. The number of planes (in case of BSQ) or pixel components (in case of BIP) to transfer in parallel is selected using generic parameters when instantiating the DMA core.

In both BSQ mode and BIP mode the Cube DMA can order the pixels *sequentially* or *block-wise*. A sequential transfer will start at the first pixel (upper left) and proceed through the cube row by row until the last pixel (lower right). In a block-wise transfer, the cube is divided into blocks with given sizes in the x and y directions. The pixels in each block are transferred in an upper left to lower right fashion, and the blocks themselves are also read in this order, starting with the upper left block and finishing with the lower right block. Figure 2.8 illustrates this, with the arrows showing the order of the components in the stream, and the numbers indicating the order that each block is streamed in.

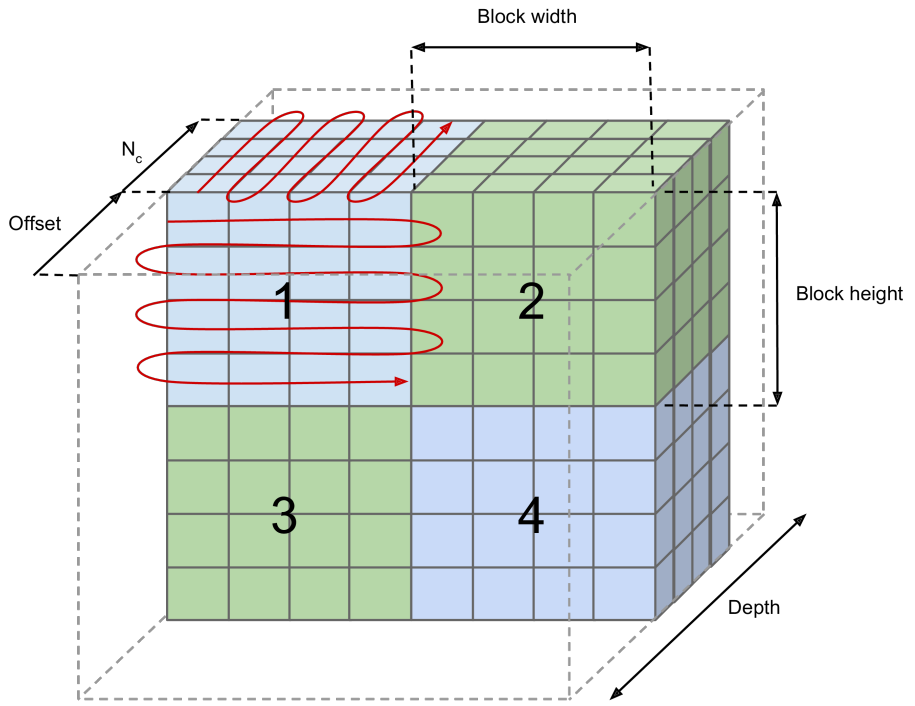

Figure 2.8: Order of processing in the Cube DMA for a block-wise transfer with blocks of size 4×4 and $N_c = 4$ planes, starting at an initial offset.

The block dimensions are required to be powers of 2. This simplifies the hardware, and it satisfies most block- or tile-based processing algorithms. There are no such restrictions for the HSI cube size. The cube width or height might therefore not necessarily be divisible by the block width or block height. As Figure 2.9 shows, this means that the last block in each block row might have a width less than the other block widths, or the last row of blocks might contain blocks that have a height that is less than the other blocks'.

A control stream is output in parallel with the data stream from the DMA. This stream contains a set of control bits for each component in the stream, which indicate whether the component is in the last pixel of a block, in a block that is in the last row of blocks, and so on. These control bits can be used by the accelerator to detect when it is handling components that belong to blocks that are smaller than the given block size.

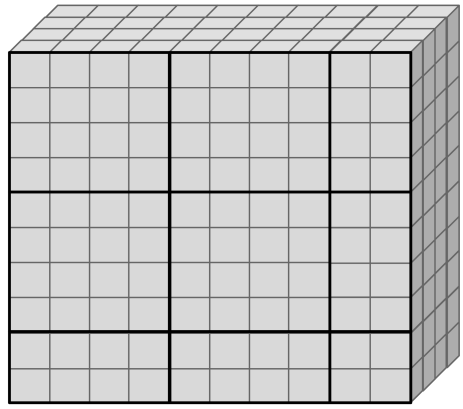

Figure 2.9: Example of a HSI cube whith spatial dimensions 10x10 and block size 4x4

2.3.2 S2MM

The S2MM channel writes incoming stream data sequentially to memory. It supports data words of different sizes, and will collect up data into 64 bit packets that are stored in memory. The S2MM channel will continuously store data until the incoming stream's **TLAST** signal is asserted. The amount of bytes that were received can subsequently be read out from the register interface. This approach has been chosen so that the DMA can easily be used in cases where the accelerator's result size is unknown, such as for compression cores where the compressed size is unknown in advance.

2.4 Ports, generics and register layout

2.4.1 Interface ports

Table 2.2 shows the different I/O ports of the Cube DMA module. The control stream, `mm2s_ctrl` and the IRQ outputs, `mm2s_irq` and `s2mm_irq` are optional.

Name	Direction	Interface type	Description
s_axi_ctrl_status_*	In/out	AXI-Lite	Register interface for configuration and status readout
m_axi_mem	In/out	AXI	DMA memory read/write interface
m_axis_mm2s_*	Out	AXI-Stream	MM2S stream (to accelerator)
mm2s_ctrl	Out	Bit vector	Control bits for each component in the stream
s_axis_mm2s_*	In	AXI-Stream	S2MM stream (from accelerator)
mm2s_irq	Out	Bit	MM2S IRQ flag
s2mm_irq	Out	Bit	S2MM IRQ flag

Table 2.2: Input and output ports in Cube DMA

2.4.2 Generic parameters

Table 2.3 shows the generic parameters that can be chosen when the Cube DMA is instantiated. In the following explanations, the parameters for component width and number of components in parallel will be referred to as W_c and N_c, respectively.

Name	Symbol	Description	Values
MM2S_COMP_WIDTH	W_c	The width of each component in bits	Even number larger than 8
MM2S_NUM_COMP	N_c	The number of components to output in each beat (BIP) or number of planes to transfer in parallel (BSQ)	Greater or equal to 1
MM2S_AXIS_WIDTH	-	The width of the AXI Stream	Large enough to hold select number of components with selected width
TINYMOVER	-	Whether to include TinyMover core or not	True/false
S2MM_COMP_WIDTH	W_c	Similar to MM2S	
S2MM_NUM_COMP	N_c	Similar to MM2S	
S2MM_AXIS_WIDTH	-	Similar to MM2S	

Table 2.3: Generic parameters for Cube DMA

2.4.3 Register configuration

Programming the Cube DMA is done through a register interface. There are two sets of registers, one for the MM2S channel, shown in Table 2.5, and one for the S2MM channel, shown in Table 2.6.

For the MM2S channel, the dimensions (width, height and depth) of the HSI cube, as well as the block dimensions are given. Block dimensions are restricted to be powers of 2. In addition, a few extra *helper values* must be given to the Cube DMA through the register interface. These are explained in Table 2.4. The extra values are used during address calculations in the Cube DMA, and are computed in software before starting a transfer. This reduces the amount of logic needed in hardware to do these computations.

Field	Description	Expression
Number of plane transfers	How many iterations needed to process a complete cube	$\left\lceil \frac{d}{N_c} \right\rceil$
Row size	The size of one row of the cube in number of components	$w \cdot d$
Last block row size	The size of one row in the last block in a row of blocks	$(w \mod w_b) \cdot d$

Table 2.4: Helper values that must be computed in software and given to the core through the register interface

Field	Description	Unit	Bits
Control register (0x00)			
Start	Core starts transfer when this bit transitions from 0 to 1		0
Block-wise mode	Cube is read in blocks of specified size		2
Plane-wise mode	Cube is read planewise, with a given number of planes in parallel		3
Error IRQ enable	Trigger IRQ when error condition arises		4
Completion IRQ enable	Trigger IRQ when transfer is complete		5
Number of plane transfers	How many plane transfers to perform		15 - 8
Start offset	Plane offset to start transferring from	c	23 - 16
Status register (0x04)			
Transfer done	Indicates whether the transfer is completed		0
Error mask	Indicates which errors occured		3 - 1
Error IRQ flag	Set when IRQ was triggered due to error. Cleared when 1 is written to this bit.		4
Completion IRQ flag	Set when IRQ was triggered due to completion. Cleared when 1 is written to this bit.		5
Base address register (0x08)			
Base address	The address of the first component in the first pixel of the HSI cube	b	31 - 0
Cube dimension register (0x0C)			
Width	The width of the HSI cube	p	11 - 0
Height	The height of the HSI cube	p	23 - 12
Depth	The depth of the HSI cube	c	31 - 24
Block dimension register (0x10)			
Block width	$\log_2$ of the width of each block	p	3 - 0
Block height	$\log_2$ of the height of each block	p	7 - 4
Last block row size	Number of components in each row of the last block in a row	c	31 - 12
Row size register (0x14)			
Row size	Number of components in one row of the cube	c	19 - 0

Table 2.5: Register layout for the MM2S channel of the Cube DMA

For the S2MM channel, only the address of where to put incoming data needs to be given.

Field	Description	Unit	Bits
Control register (0x20)			
Start	Core starts transfer when this bit transitions from 0 to 1		0
Error IRQ enable	Trigger IRQ when error condition arises		4
Completion IRQ enable	Trigger IRQ when transfer is complete		5
Status register (0x24)			
Transfer done	Indicates whether the transfer is completed		0
Error mask	Indicates which errors occured		3 - 1
Error IRQ flag	Set when IRQ was triggered due to error. Cleared when 1 is written to this bit.		4
Completion IRQ flag	Set when IRQ was triggered due to completion. Cleared when 1 is written to this bit.		5
Base address register (0x28)			
Base address	The address of where to store the incoming stream data	b	31 - 0
Received length register (0x2C)			
Received length	The number of bytes received from start of transfer until TLAST was asserted	b	31 - 0

Table 2.6: Register layout for the S2MM channel of the Cube DMA

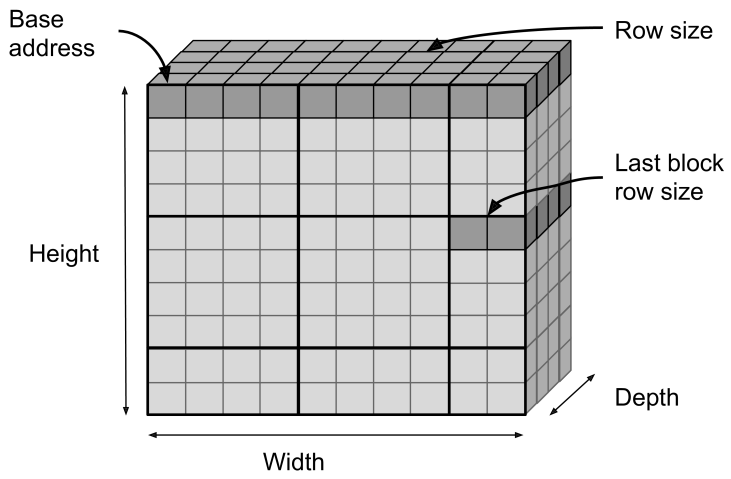

Figure 2.10: Relation between configuration register fields and HSI cube

2.5 Overview of MM2S channel implementation

This section will go briefly through the implementation of the MM2S channel which was done in the project work leading up to this thesis [8].

A complete cube transfer consists of a series of *sub-transfers*. In the case of BIP non-blocked transfers, the transfer consists of h sub-transfers, each transferring one row in the cube. In the case of block-wise transfers, each sub-transfer covers one row in the block. For BSQ transfers, there is one sub-transfer performed for each set of N_c components.

As shown in Figure 2.7, the MM2S channel consists of the Xilinx DataMover or the TinyMover which perform the sub-transfers, the unpacker which takes care of unpacking the packed components from memory into a selected number N_c of components to feed in parallel to the accelerator, and the controller which sends sub-transfer commands to the DataMover or TinyMover and configuration words to the unpacker.

2.5.1 DataMover

The DataMover IP from Xilinx can perform transfers with a length of up to 2^{23} bytes, and will issue the appropriate AXI read requests to perform the transfer. The DataMover is controlled through a command word interface, where command words contain the start address, the number of bytes to read and flags that control some aspects of the transfer, such as whether **tlast** should be asserted on the stream coming out from the DataMover when the transfer has been completed.

2.5.2 Unpacker

The unpacker converts the incoming packed stream from memory to a stream where the user selected number of components, N_c, are output in parallel, as illustrated in Figure 2.11. An overview of the unpacker architecture is shown in Figure 2.12. The unpacking is performed in three steps: shifting, restructuring and buffering.

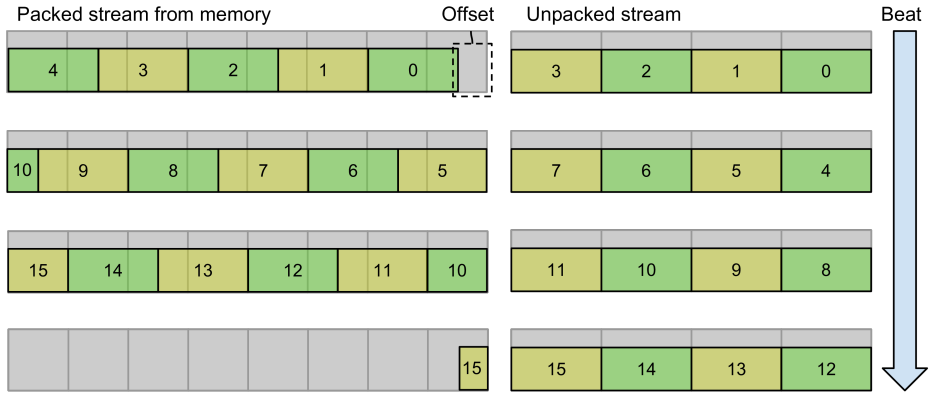

Figure 2.11: Example of packed stream coming from memory and the resulting unpacked stream when $W_c = 12$ and $N_c = 4$. Left: the stream of packed 64-bit words from memory. Right: the unpacked stream with N_c components in parallel.

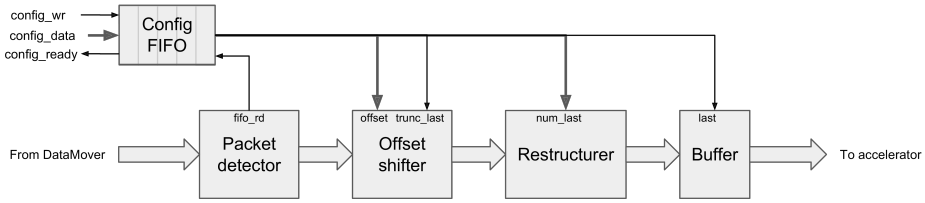

Figure 2.12: Overview of the MM2S unpacker module

2.5.2.1 Shifter

Shifting is necessary to perform when $W_c \bmod W \neq 0$, because in that case the LSB of a component might not be aligned with the LSB of a byte as is shown in Figure 2.13. For instance, if $W_c = 12$, a component might have its LSB at a byte boundary, but it might also have an offset of 4. Similarly, for $W_c = 10$, a component might have an offset of 0, 2, 4 or 6 from the nearest byte boundary.

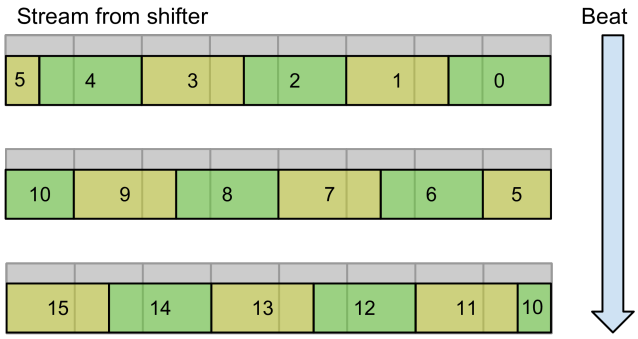

Figure 2.13: The stream from the shifter, given the input shown to the left in Figure 2.11

2.5.2.2 Restructurer

After shifting has been performed, the data in the incoming 64-bit stream contains a number of whole components, but might also contain *partial* components. The least significant bits might be the remaining bits from a partial component in the previous 64-bit word, while the most significant bits might be the least significant bits of a component that is completed by bits from the next 64-bit word. This can be observed in Figure 2.13: Component 5 has its least significant bits in the most significant bits of the first word, and its most significant bits in the least significant bits of the second word. Similarly, component 10 has its least significant bits in the most significant bits of the second word, and the remaining bits in the least significant bits of the third word.

The restructurer performs the necessary operations to collect up whole (unsplit) components before forwarding them to the next stage. For the example case, this means that when receiving the first word, the whole components 0 through 4 can be forwarded, while the least significant bits of component 5 are saved until the next word arrives and they can be joined with the remaining bits of component 5 and be forwarded together with components 6 through 10. The resulting stream is shown in Figure 2.14.

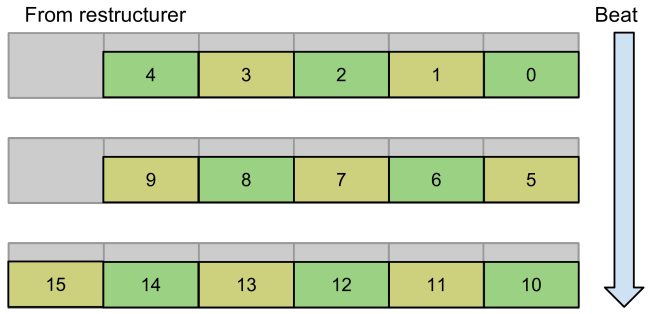

Figure 2.14: The stream from the restructurer, given the input stream from the shifter shown in Figure 2.13

2.5.2.3 Buffer

The buffer collects up the whole components arriving from the restructurer until there are N_c components ready to be output in parallel to the accelerator. The output stream from the buffer is what is shown to the right in Figure 2.11.

2.5.2.4 Configuration FIFO

Each time data from a new sub-transfer starts arriving from the memory, the bit-offset of the first component might be different, and the number of components in the packet might change. The necessary information that is needed by the unpacker to handle a sub-transfer correctly is called a *configuration*. The configuration word consists of the number of bits to shift the incoming word from memory in the shifter module, the number of components in the last words from memory, which is needed by the restructuring module, and some flags that are used during the unpacking process.

Each time a new packet arrives (first beat where **tlast** is 0 after being 1) from the DataMover or TinyMover, the next configuration word is read from the configuration FIFO.

2.5.3 Controller

The controller module controls operations during DMA transfers. It keeps track of transfer progress, calculates start addresses and transfer lengths for each sub-transfer, and feeds the necessary commands to the DataMover (or TinyMover) and configuration words to the unpacker.

An overview of the controller is shown in Figure 2.15. A slightly different controller is used when the TinyMover core is used instead of the Xilinx DataMover.

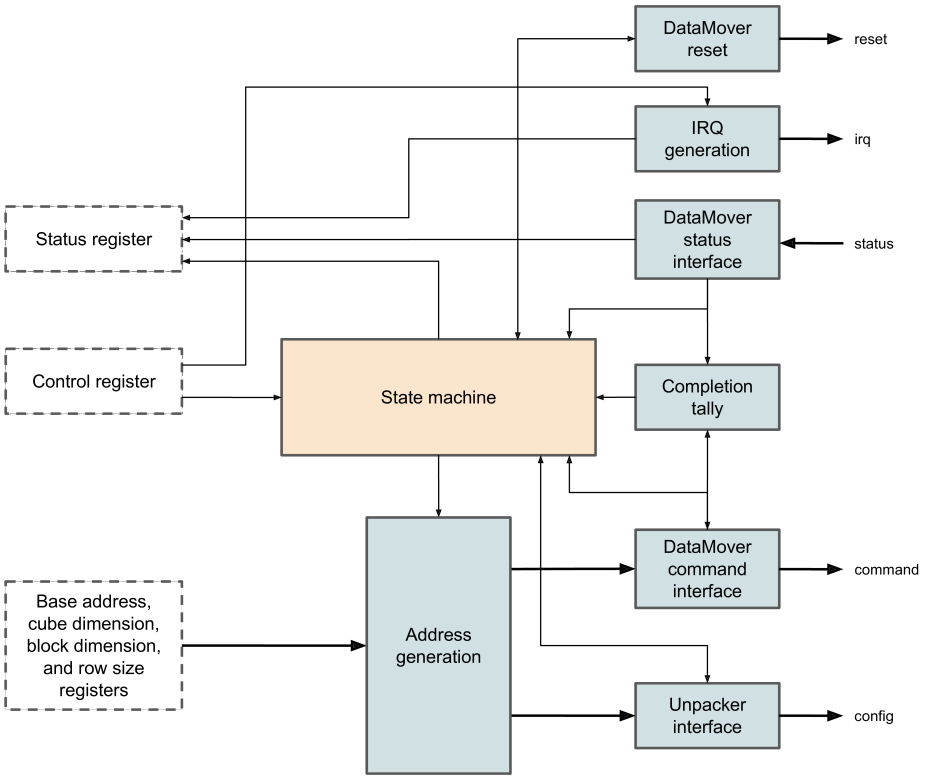

Figure 2.15: Overview of the controller module

2.5.3.1 Address generation

Based on the configuration set in the configuration registers, the address generation module takes care of calculating the necessary information needed to issue a command to the DataMover and configure the unpacker for that particular transfer.

The address generator keeps track of the current position in the cube by using a set of counters representing the x and y coordinates of the current block as well as the x and y coordinates of the current pixel within that block in case of BSQ transfers. This part of the address generation logic was rewritten for the master's thesis, and is therefore shown in more detail in the next chapter.

Based on the current position in the cube, the address generator calculates the following:

- The start address for the next sub-transfer
- The number of bytes in the next sub-transfer
- The bit-offset from the LSB in the first byte to the LSB of the first component

Each time a command is handshaked with the DataMover or TinyMover, the address generator increments its counters and moves on to the next position in the cube, and calculates the necessary information for the next sub-transfer.

2.5.3.2 IRQ generation

The IRQ generator generates an interrupt request signal (IRQ) to the CPUs when either an error occurs, or when a transfer completes. The software can decide which events should trigger an IRQ by setting a 2-bit *IRQ mask*, where bit 0 is the error event bit and bit 1 is the completion bit. When an IRQ event has occured, and the corresponding bit is set in the IRQ mask, the IRQ output signal will be asserted. The software can find out which event triggered the IRQ by reading the status register. The software must then acknowledge the IRQ by writing to the same bit positions in the status register.

2.5.3.3 State machine

Sequencing of operations in the controller is controlled by a state machine. The state transition diagram is shown in Figure 2.16.

The state machine goes from the idle state to the running state when the start bit in the control register is set from software. When commands for all the sub-transfers needed to transfer the whole cube have been accepted by the DataMover or TinyMover, the state machine moves into a state where it is waiting for all status words from the DataMover or TinyMove to come back. A *completion tally* counter keeps track of how many commands have been issued versus how many have been completed. When all issued commands have been completed, the state machine can move back to the idle state. The other states shown in Figure 2.16 deal with error conditions that might arise during the transfer, for instance that a read is attempted from an invalid memory address.

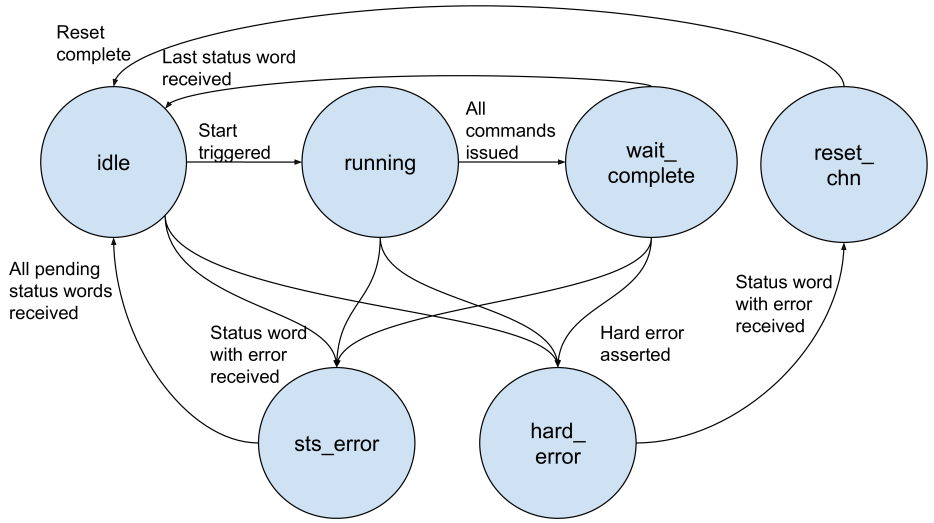

Figure 2.16: State transition diagram for the state machine

2.6 The TinyMover core

During testing of the Cube DMA, it was found that BSQ transfers were being slower than expected. This turned out to be due to a limitation in the DataMover where a new read request cannot be issued more frequently than every 7th clock cycle. For BSQ transfers where one read request is issued for each set of N_c components in the cube, this puts a severe limit on the achievable throughput.

A replacement for the DataMover, the TinyMover, was therefore implemented, specialized for tiny transfers where the number of bytes to fetch fits within one burst transfer on the AXI bus. An overview of the TinyMover is shown in Figure 2.17.

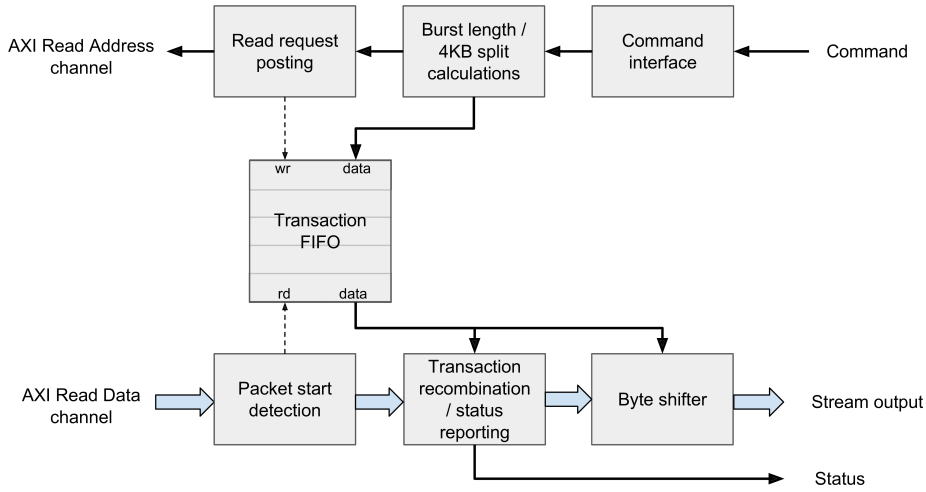

Figure 2.17: Overview of the TinyMover core

The TinyMover consists of a *read request path*, which is shown going right to left at the top of Figure 2.17, and a *read data path* which is shown going left to right at the bottom of the figure. The read request path translates the incoming command word containing the start address and number of bytes to read, into an AXI read request. Information needed when the data associated with the transaction that this request initiates comes back from memory, is pushed to a FIFO. The read data path performs the necessary steps to produce a contiguous AXI Stream containing the read data. It also takes care of byte level shifting that is needed when read requests are made to memory addresses that are not aligned with 32-bit boundaries.

2.7 Integrating DMAs with software

This section will lay out some necessary concepts and information about the Zynq-7000 SoC with regards to integrating DMAs with software running on the ARM processor cores.

2.7.1 Interrupts on Zynq-7000

The software has two ways to find out when a DMA transfer has completed: It can either repeatedly *poll* the DMA, typically by reading a status register and checking bits that indicate completion, or set itself up to be interrupted when a DMA transfer has completed.

Using interrupts has several advantages over polling. Whereas polling requires the CPU to periodically check the state of the DMA, interrupts allow the CPU to save power by going into a sleep mode or to do other useful work while the transfer is on-going.

2.7.1.1 The Generic Interrupt Controller

Each CPU in the Zynq-7000 only has two interrupt inputs, the FIQ (fast interrupt request) and IRQ (interrupt request). FIQs have higher priorities than IRQs, meaning that if both occur at the same time, the CPU will first handle the FIQ.

The Zynq-7000 has several different interrupt sources, such as different software generated interrupts, I/O peripherals and timers, and 16 interrupt signals that can be connected to custom logic in the PL. Selecting which interrupt sources to forward to the FIQ or IRQ inputs on the CPUs is done by the Generic Interrupt Controller (GIC), which is illustrated in Figure 2.18.

The GIC allows the software to control which interrupt sources to enable (not ignore), and how different interrupt sources will be prioritized when they signal interrupts at the same time. Software can also control how interrupts are distributed to each of the two CPU cores in the system. When an interrupt causes one of the CPUs to be interrupted with a FIQ or IRQ, the software can check the GIC status registers to figure out which of the many interrupt sources caused the FIQ or IRQ in question.

2.7.1.2 Interrupt handling

In ARM processor terminology, FIQs and IRQs are part of the broader *exception* concept. Exceptions are events that cause the CPU to do a *context switch*: It stops running its current instruction stream, saves the state of all registers, and looks up in a *exception vector table* to find the address of an *exception handler* routine that it will start executing. The routine will handle whatever event caused the exception, and then the CPU will restore the previous state and continue executing as it was before the exception occured.

FIQ and IRQ are two of the possible exception events. When these occur, an exception handler must be run that will handle them. At that point, there is no information about which one of the interrupt sources caused the IRQ. The exception handler must check status registers in the GIC to find out.

Xilinx provides a driver for using the GIC in software. This driver also includes an exception handler that can be registered in the exception vector table. The driver allows the user to add custom interrupt handlers for different interrupt sources. When the exception occurs, the exception handler will then check with the GIC and call the correct interrupt handler.

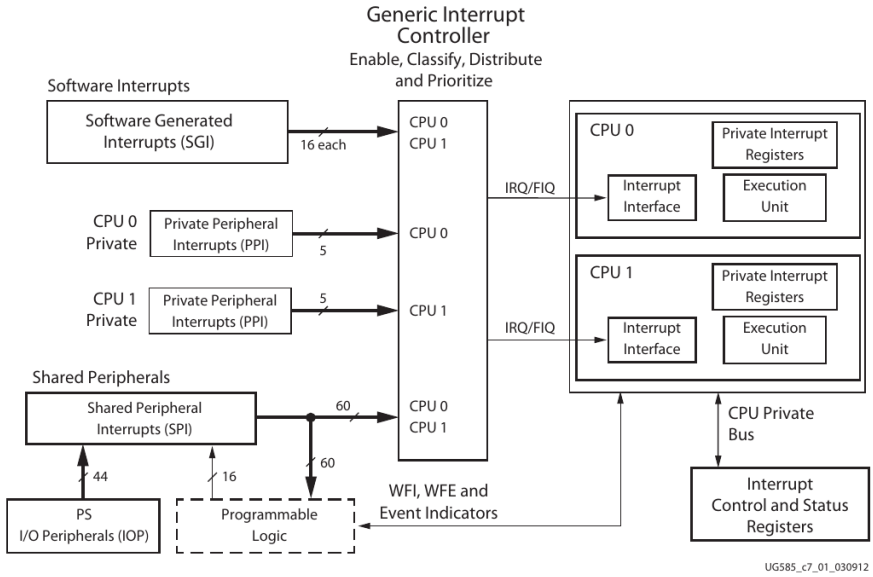

Figure 2.18: The Zynq-7000 Generic Interrupt Controller (GIC) [2]

2.7.2 Caches

As in most modern processing systems, the Zynq-7000, uses a *memory hierarchy* to hide the read- and write latencies of large DDR memories. The hierarchy is shown in Figure 2.19. Two levels of cache memory, Level 1 (L1) and Level 2 (L2) is used between the CPUs and the DDR memory, but to simplify the following text, these will be grouped together and referred to as ”the cache”.

2.7.2.1 Caching and cache lines

The two main principles behind caches are that memory locations close to the currrent location are likely to be used next (spatial locality), and if a memory location is accessed now, it is likely to be accessed again later in the program (temporal locality). It is therefore beneficial to copy data from the memory locations close to the location that the CPU is currently accessing into the cache, so that when the CPU needs to access one of these locations, it can find the data in the cache instead of doing slower read/write operations on the main memory.

The Zynq-7000 caches are divided into *lines* of 32 bytes. These lines correspond to 32 byte continuous blocks in the main memory. Whenever the CPU accesses one particular memory location that is not cached, the whole 32 byte memory block that the location belongs to is copied into an available cache line. It remains in the cache line until it gets

replaced by a newer cache line, or until it is explicitly removed by the software.

2.7.2.2 Coherency

When memory is shared between several units, such as between the CPUs and the DMA in the Zynq-7000, the problem of *cache coherency* is introduced. The data in the DDR memory isn't necessarily the same as in the caches. For example, if the CPU reads from a particular address, the surrounding data is put in a cache line. In the mean time, the DMA might write new data into the DDR memory. The next time the CPU reads from the same address, it will get the cached data instead of the new data written by the DMA. The cached data has become *stale*.

Another scenario that can happen is that the CPU writes to an address that is cached. The change will only occur in the cache, and not be visible in the DDR memory until the cache line is evicted from the cache and written back to the memory. If the DMA reads from the same region of memory during this period, it will get the old data and not the new data that was written by the CPU. The data in the memory region has become stale.

The Zynq-7000 has a Snoop Control Unit (SCU) designed to keep the caches and memory coherent with an external hardware accelerator. To use this feature, it requires that the external hardware, in our case the DMA, is connected to a special Accelerator Coherency Port (ACP) such that all memory requests are routed through the SCU. This has several tradeoffs; for instance, accesses where data is present in the cache will be quicker, but on the other hand the accelerator will compete with the CPUs to access data. In the Zynq-7000 manual, Xilinx conclude that using the SCU and ACP is optimal for medium-grain accelerators that operate on fairly small data sets. [2].

When the DMA is connected directly to the DDR memory through one of the High Performance ports, cache coherency must be managed manually. To sum up the previous discussion, the two issues that must be taken care of are:

- *Memory becomes stale:* The CPU writes data to a location in memory which is to be streamed by the DMA, but the new data is only written to the cache and not to the actual DDR memory.
- *Cache becomes stale:* The DMA has written an incoming stream to memory, but when reading from this location the CPU instead reads an out-dated cache line.

2.7.2.3 Flusing and invalidating

Cache coherency is achieved manually by doing two operations from software: *flushing* and *invalidating*. These are available as library functions in the Xilinx SDK, `Xil_CacheFlushRange` and `Xil_CacheInvalidateRange`. Behind the scenes, these functions talk to the cache controllers inside the Zynq-7000. Both functions take two arguments: A pointer to the start of the memory region in question and its length.

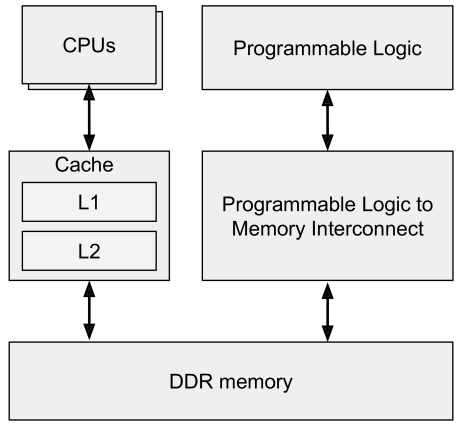

Figure 2.19: Memory hierarchy in Zynq-7000

In the case of flushing, any cache line that contains data from within the given region is written out to memory. This means that new values written by the software is actually visible in the DDR memory, and the DMA will read the correct data from memory.

In the case of invalidating, any cache line that contains data from within the given region is marked as invalid. This means that the next time the CPU reads from any address in the region, it is forced to read data from the DDR memory and not outdated values from the cache.

3

Cube DMA implementation

This chapter will present the continued work on the Cube DMA that was performed for this thesis. It will also explain briefly how benchmarking was performed for the Xilinx AXI DMA and the Cube DMA.

The following list shows the changes and new features that have been implemented for the Cube DMA:

- Address generation and register layout has been simplified under the assumption that block dimensions are powers of 2
- Added capability of doing BSQ transfer of complete cube without any user intervention
- Added output of control stream in parallel with output data
- Completed implementation of S2MM channel

During the next sections these points will be detailed.

3.1 New address generation logic

A serious problem in the address generation logic implemented in the project was that it did not handle the case where the cube width and height dimensions are not whole multiples of the block dimensions. This would lead to the transfers of the last blocks in each row of blocks being too large, and the number of transfers in the last row of blocks to be too large.

To fix this, it was decided to restrict the block dimensions to be powers of 2. This is a reasonable restriction, as any block- or tile based algorithms that have been looked

at for hyperspectral image processing use block dimensions that are powers of 2. Constraining the block dimensions like this allows several computations to be made much easier: Multiplying and dividing by the block dimensions becomes a matter of shifting, and finding residuals (modulo operations) becomes a matter of picking out the least significant bits.

This also turned out to improve the ease of use of the core. With the old address generation logic, quite a few helper values had to be computed in software and supplied by the user through configuration registers. These are now for the most part computed by the core itself, leaving only the values that were listed in 2.4 to be supplied externally. These values could have been computed in hardware as well, but as they are only calculated once before the start of a cube transfer, it is better to save the additional logic and perform the computations in software.

Code listing 3.1 shows in pseudo-code how the new address generation logic works.

```
if !mode_block:
  num_blocks_y = 1
  num_blocks_x = 1
  current_block_height = height
  current_block_width = width
  length = row_size
else:
  num_blocks_y = height / 2**block_height
  num_blocks_x = width / 2**block_width

if !mode_plane:
  num_plane_transfers = 1
  current_block_width = 1
  length = depth

for num_plane_transfers-1 to 0:
  block_address     = offset
  row_address       = offset
  block_row_address = offset
  component_address = offset

  for block_y in num_blocks_y-1 to 0:
    for block_x in num_blocks_x-1 to 0:
      for y in current_block_height-1 to 0:
        for x in current_block_width-1 to 0:
          if mode_block:
            if block_x = 0:
              current_block_width = width mod 2**block_width
              length = last_block_row_size
            else:
              current_block_width = 2**block_width
              length = 2**block_width * depth

            if block_y = 0:
              current_block_height = height mod 2**block_height
            else:
              current_block_height = 2**block_height

          issue command(component_address, length)
```

```
        if x = 0 and y = 0:
          if block_x != 0:
            block_address = block_address + (width mod 2**block_width)
          else:
            block_row_address = block_row_address
                              + 2**block_height * row_size
            block_address = block_row_address
        if x != 0:
          component_address = component_address + depth
        else:
          if y != 0:
            row_address = row_address + width
          else:
            row_address = block_address
          component_address = row_address

        wait for tick from state machine
offset = offset + comp_per_cycle
```

Listing 3.1: Pseudo-code of address generation

3.2 Control stream implementation

The buffer module in the unpacker will collect up N_c components before these are forwarded to the accelerator, even if the buffer already contains components from another sub-transfer. This means that the components that are output on the MM2S stream might originate from different pixels or different blocks in the HSI cube. For the accelerator that is processing the streamed data, it might be necessary to know whether the components are from different blocks, and also whether or not the incoming components come from a truncated block in the last column of blocks or from a truncated block in the last row of blocks, such that appropriate padding etc. can be performed.

To help identify components correctly, a set of control signals per component are put out in parallel with the data stream. These indicate:

1. Whether the component is part of the last pixel in a block
2. Whether the component is part of a block in the last column of blocks
3. Whether the component is part of a block that is in the last row of blocks

Figure 3.1 illustrates how these control signals relate to the HSI cube with an example where $N_c = 4$. The three first components in the stream are from the last pixel in a block, and also from a block that is in the last column of blocks, and hence the corresponding control bits are 1. The fourth component is from the first pixel in the next block, and is not in the last column of blocks, so the corresponding bits are 0. However, it is in a block that is in the last row of blocks, and the corresponding control bit for this is 1.

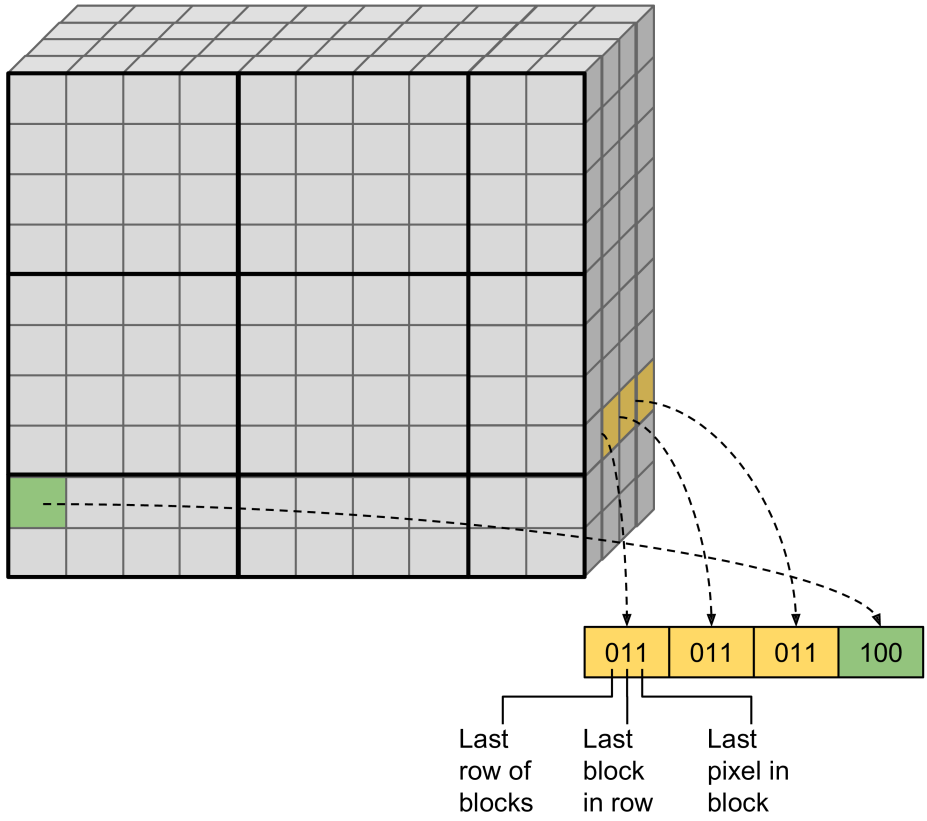

Figure 3.1: Control bits

To implement this feature, the calculation of these signals was added to the address generation logic. Since there are already counters keeping track of the current pixel and block coordinates, this is mostly a matter of comparing these counter values to the total number of blocks and total number of pixels, respectively.

The unpacker configuration FIFO was extended with room for the three new bits. The bits are put in the unpacker configuration FIFO together with the other information that describes the particular sub-transfer. This is done so that when the corresponding sub-transfer data arrives from the DataMover or TinyMover, the associated control bits are read from the FIFO.

The control bits are appended to each component before it is stored in the buffer. At the buffer output, the control signals are split from the component again and put out on a separate control bus. This makes sure that the right control signals are output simultaneously with the components they belong to.

3.3 S2MM channel implementation

3.3.1 Controller

The S2MM controller functionality shares much of the same logic with the MM2S controller. It is therefore implemented in the same VHDL module, using a generic parameter to indicate whether the controller is to be used for S2MM or MM2S, and enabling or disabling features using VHDL `if generate` statements. The difference between the two is mainly how commands are issued to the DataMover.

The S2MM channel continually writes stream data to the memory until the incoming stream asserts **tlast**. To achieve this behavior, a feature of the DataMover called *indeterminate byte transfer* is enabled. When instantiated in this mode, the DataMover will accept packets that are larger than the given size in the command. It will report back a status word with an End Of Packet bit (EOP) that indicates whether **tlast** was asserted during the sub-transfer described by the command or not, and the actual number of bytes that were read.

This allows transfers of any length by issuing commands for new sub-transfers every time the DataMover reports a status that does not have the EOP bit set, because that means that the DataMover has performed the largest possible sub-transfer, and there is still more stream data to come.

Listing 3.2 shows how commands are issued to the DataMover in the S2MM Controller.

```
MAX_LENGTH = 0x3FFFFF
bytes_total = 0
start_address = base_address
loop:
  issue command (start_address, MAX_LENGTH)
  wait for status
  if status.eop:
    bytes_total = bytes_total + status.bytes_received
    exit
  else:
    start_address = start_address + MAX_LENGTH
    bytes_total = bytes_total + MAX_LENGTH
```

Listing 3.2: Pseudocode for S2MM command issuing

3.3.2 Component packer

Similarly as with the MM2S output stream, the incoming S2MM stream can contain any number of components N_c in parallel, and the component width W_c does not have to be a multiple of 8 bits. Since the AXI bus that connects the DMA to memory is 64 bits wide, this means that the incoming components must be packed into 64 bit words before

being written to memory. The component packer performs the opposite operation of the component unpacker that was detailed in Section 2.5.2. Figure 3.2 shows an incoming stream from an accelerator, and the resulting packed 64-bit words that are written to memory.

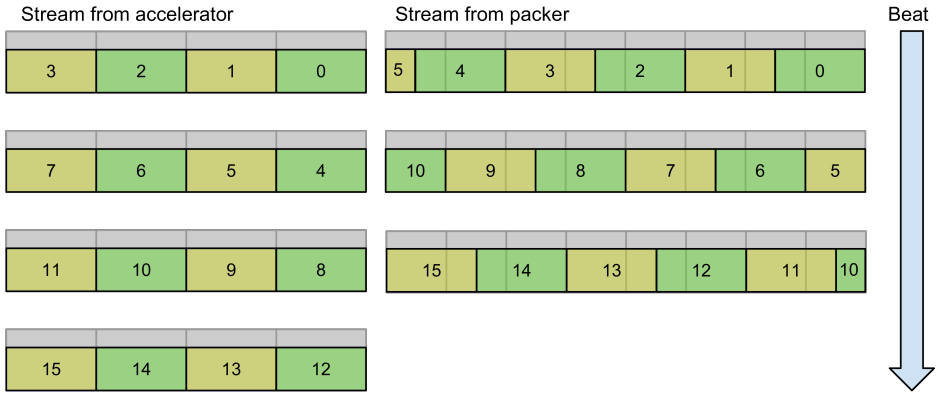

Figure 3.2: Example of packing when $W_c = 12$ and $N_c = 4$. Left: stream from accelerator with N_c components in parallel. Right: the packed stream of 64-bit words.

The component packer architecture is shown in Figure 3.3

3.3.2.1 Buffer

The S2MM stream data is first put in a component buffer similar to the one used at the last stage in the MM2S unpacker. The buffer collects components until there are enough components to output to the next stage. The component joiner in the next stage notifies the buffer about how many components it expects. When the number of components in the buffer is larger or equal to the requested number, the buffer will forward the selected number of components to the component joiner.

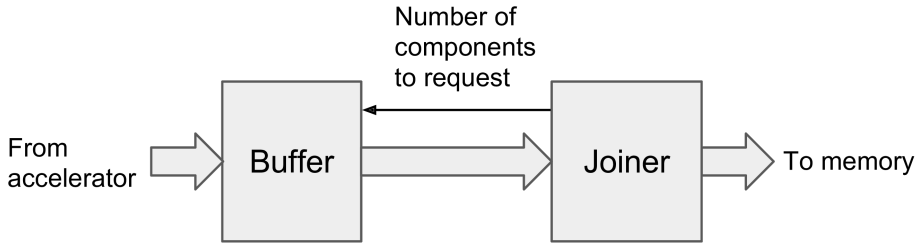

Figure 3.3: The component packer

3.3.2.2 Component joiner

The component joiner does the opposite operation of the component restructurer shown in Section 2.5.2.2. It requests a number of components from the buffer that is great enough to cover 64 bits of data, and any bits that are left over are saved and joined with the next set of components from the buffer, as shown in Figure 3.4.

How many bits are needed to request from the buffer to cover 64 bits of data and how many are left over changes each cycle, but there is a consistent pattern that allows this to be efficiently done in hardware. The pattern repeats every N cycles, where N can be determined by how many input words of width $N_c \cdot W_c$ are needed to cover a multiple of the memory width which is 64 bits. This can be found by finding the least common multiple between 64 and $N_c \cdot W_c$ (the least number of bits that divides both stream widths) and dividing it by the memory stream width, which can also be expressed in terms of the greatest common divisor:

$$N = \frac{\text{lcm}(64, N_c \cdot W_c)}{64} = \frac{64}{\text{gcd}(64, N_c \cdot W_c)}.$$

For each cycle, the number of components to request from the buffer must be large enough such that the number of bits from the buffer plus the number of leftover bits from the previous cycle is larger than or equal to 64. Table 3.1 shows the expressions that determine how many components to request from the buffer, how many bits to join from the leftovers of the previous cycle, and how many bits are left over after the current cycle. These are used in the VHDL implementation to select the arrangement of the output bits and the next value of the leftover register for the different cycles.

Table 3.2 shows an example of these computations when the component width is 10 bits, and Figure 3.4 shows the behavior of the component joiner during one full cycle.

Parameter	Expression
Number of components to request	$\left\lfloor \frac{(i+1)\cdot 64}{W_c} \right\rfloor - \left\lfloor \frac{i \cdot 64}{W_c} \right\rfloor$
Leftover bits from previous cycle	$((i+1) \mod N) \cdot 64 \mod W_c$
Leftover bits after current cycle	$i \cdot 64 \mod W_c$

Table 3.1: Overview of component joiner cycles

Cycle	Number to request	Leftover from prev.	Leftover bits
0	$\lfloor 5 \cdot 64/10 \rfloor - \lfloor 4 \cdot 64/10 \rfloor = 7$	$0 \cdot 64 \mod 10 = 0$	$4 \cdot 64 \mod 10 = 6$
1	$\lfloor 4 \cdot 64/10 \rfloor - \lfloor 3 \cdot 64/10 \rfloor = 6$	$4 \cdot 64 \mod 10 = 6$	$3 \cdot 64 \mod 10 = 2$
2	$\lfloor 3 \cdot 64/10 \rfloor - \lfloor 2 \cdot 64/10 \rfloor = 7$	$3 \cdot 64 \mod 10 = 2$	$2 \cdot 64 \mod 10 = 8$
3	$\lfloor 2 \cdot 64/10 \rfloor - \lfloor 1 \cdot 64/10 \rfloor = 6$	$2 \cdot 64 \mod 10 = 8$	$1 \cdot 64 \mod 10 = 4$
4	$\lfloor 1 \cdot 64/10 \rfloor - \lfloor 0 \cdot 64/10 \rfloor = 6$	$1 \cdot 64 \mod 10 = 4$	$0 \cdot 64 \mod 10 = 0$

Table 3.2: Overview of component joiner cycles

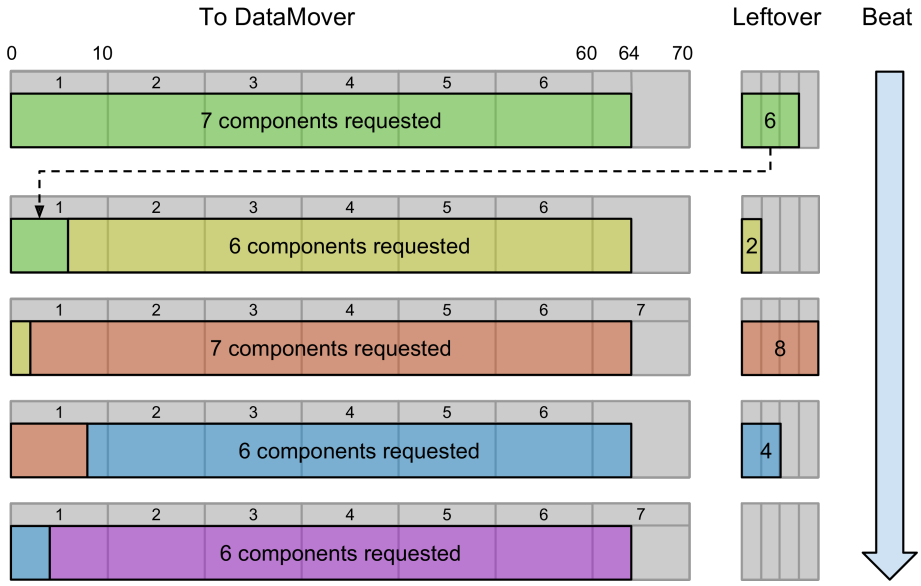

Figure 3.4: Component joiner behavior for one set of cycles when $W_c = 10$ and $W = 64$

3.4 Test setup for comparing Xilinx AXI DMA and Cube DMA

A typical DMA setup in the Zynq-7000 was shown in Figure 1.9. This setup has been used when comparing the Cube DMA and the AXI DMA, with a simple FIFO used in place of an actual hardware processing core. The FIFO simply forwards the incoming MM2S stream from the DMA back to the S2MM input of the DMA.

Testing of the Cube DMA can easily be performed without involving any software, as detailed in 8, but testing the Xilinx AXI DMA involves a lot more work because block descriptor chains need to be set up to describe the transfer. Testing of both the AXI DMA

and the Cube DMA has therefore been performed using a software program running on one of the ARM cores on the Zynq-7000.

Similarly to the Cube DMA, the AXI DMA is also configured through registers. The registers are accessible through an AXI-Lite interface, which is connected to the Central Interconnect in the Processing System. This makes the registers appear as memory locations from the CPUs point of view, and transactions with these registers can be initiated through read and write operations by the CPU.

The most important registers are the control and status registers, and two registers that are pointing to the first and last block descriptor in the current chain, respectively. However, Xilinx provides ready-made libraries (drivers) with C code for using many of their peripherals and IP cores, including the AXI DMA. This driver code takes care of many of the register-level details of configuring the core.

A software program must generally perform the following steps to perform a DMA transfer:

1. Set up the Generic Interrupt Controller (GIC)
2. Register an interrupt handler function for DMA interrupts
3. Register a GIC exception handler
4. Enable exceptions
5. Place data to be used in memory
6. Flush all data in the cache belonging somewhere inside the input data region in memory
7. (For AXI DMA: Set up initial block descriptor chain)
8. Set up DMA registers and start transfer
9. Wait for interrupt(s) to occur
10. (For AXI DMA: Set up new chain of block descriptors and repeat step 9 until transfer is done)
11. Invalidate cache data belonging to regions in memory where the DMA stores the received data

Configuring the GIC, the ARM exception system and dealing with caches, is done through driver code provided by Xilinx. The GIC driver provides an exception handler that will itself call the user-provided interrupt handlers for specific GIC interrupts such as the DMA interrupt. When only testing performance and not caring about data validity, the cache related steps can be excluded from the program. However, in general they are important to include to avoid the issues that were described in 2.7.2.

For the Xilinx AXI DMA, the test program is a bit more involved. The AXI DMA performs transfers as described by a chain of block descriptors. Each time a transfer needs to be started at a new location, a new block descriptor must be used. For sequential

BIP transfers, only one block descriptor is needed, provided that the cube is small enough for its total data size to be representable in the number of bits that the length field in the descriptor has. For larger cubes, the transfer is simply described using a chain of two (or more) descriptors, where the first have the maximum length, and the last has the remaining length.

For block-wise transfers, each row in each block needs to be described by a separate block descriptor, as shown in Figure 3.5. It is clear that this might require enormous block descriptor chains; for instance, if a cube with spatial dimensions $512 \times 2000\times$ is used with blocks of size 8×8, then a total of $(512/8) \cdot (2000/8) \cdot 8 = 128000$ descriptors would be needed. This is larger than what the AXI DMA can handle, and it is therefore necessary to set up only a certain amount of block descriptors, start the transfer, set up the next block descriptors in the chain, continue the transfer, and so on.

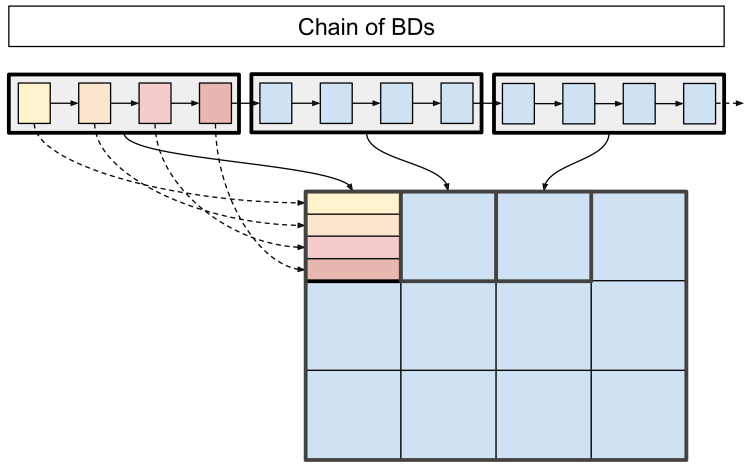

Figure 3.5: Descriptor setup for doing block transfers with AXI DMA

The test code used for the AXI DMA sets up an initial chain of block descriptors and starts of a transfer. When that transfer has completed and an interrupt is triggered, a new chain of block descriptors is set up for the next portion of the transfer and a new transfer is started. This is repeated until the whole cube has been traversed. The test code for the Cube DMA is similar, but it will perform the whole cube transfer without any intervening from software. When the interrupt occurs, the transfer has been finished.

In both cases, an internal timer in the Application Processor Unit of the Zynq-7000 is used to record the cycle counts right before a transfer is started and right after the transfer has completed. The timer operates at half the frequency of the CPU, meaning that the elapsed time can be found by multiplying the cycle difference between start end end by half the CPU frequency.

4

Cube DMA results

This chapter presents results for the Cube DMA implementation. It will briefly show the results gathered in the project work regarding timing and resource utilization, and provide more in-depth discussions of the results from the continued work in this thesis.

4.1 Performance comparison

The AXI DMA and Cube DMA are compared as shown in Table 4.2. The parameters used for this comparison are shown in Table 4.1.

The results show that the achievable throughput for BIP block transfers in the Cube DMA increases by 128% compared to the Xilinx AXI DMA. This is likely due to the much greater overhead present when doing block transfers using the AXI DMA. For each row in the block, the AXI DMA must fetch a new descriptor from memory. The Cube DMA has no such overhead, but there are still possible delays related to the Zynq's memory system having to jump to new addresses when starting on a transfer of a new row in the block.

For BSQ (plane-wise) transfers, the Cube DMA can achieve 14.1% of the theoretical throughput. With the TinyMover used in place of the Xilinx DataMover, this rises to 73%. Further improvements are unlikely, as the TinyMover issues new read requests as quickly as possible, meaning that the stalls that are occurring are due to the memory system of the Zynq-7000. It is expected that close to theoretical performance is unachievable, as the memory system is optimized for sequential burst transfers and not small and strided memory accesses.

Cube parameters	
Width	500
Height	2000
Bands	100
Component width	8
Stored order	BIP
Block size	8×8
AXI DMA	
Scatter-Gather	Yes
Burst size	16
Stream width	64
Dynamic Realignment Engine	Yes
Length reg. size	Maximum (23 bit)
Cube DMA	
Stream width	64
Components in parallel	8
Component width	8
Burst size	16
TinyMover	Tested with and without

Table 4.1: Parameters used for performance comparison

	BIP, seq.	BIP, block	BSQ, seq.
Theoretical	800 MB/s	800 MB/s	100 MB/s
AXI DMA	800 MB/s	340 MB/s	
Cube DMA	800 MB/s	775 MB/s	14.1 MB/s
Cube DMA (TinyMover)			73 MB/s

Table 4.2: Comparison of performance for AXI DMA and Cube DMA for different transfer types on a HSI cube of size $500 \times 2000 \times 100$, block size 8×8.

4.2 Resource utilization

Table 4.3 shows the resource utilization of the Cube DMA components that are fixed in size regardless of choice of generic parameters. Table 4.4 shows the LUT and register usage of the MM2S channel's controller and unpacker as the number of bits per component W_c and the number of components in parallel N_c are changed. These results are also plotted in Figures 4.1 and 4.2. For both, the general trend is an expected approximately linear increase as N_c grows. These results were gathered and analyzed more in detail in the feasibility study report [8].

Table 4.5 shows the resource usage in terms of LUTs and registers for the S2MM packer, with plots in Figures 4.3 and 4.4. For the values of W_c that are not byte multiples, a clear linear growth is observed as N_c increases, similarly to in the MM2S channel. This is expected, as the width of muxes, registers and so on scale linearly with N_c. It is also observed that the LUT usage for $W_c = 18$ is highest, followed by $W_c = 10$ and $W_c = 12$ and then $W_c = 8$ and $W_c = 16$. This ordering is due to the complexity of the component joiner. For $W_c = 18$, $W_c = 10$, $W_c = 12$, the number of cycles in the component joiner are 9, 5 and 3, respectively. Since for each cycle there is a different choice of leftover bits, arrangement of output word, and so on, the amount of logic grows with the number of cycles.

In the cases where $W_c = 8$ and $W_c = 16$, the LUT usage is smallest. This is because the component joiner is not needed for these component widths, as 64 (the bus width) is a multiple of W_c in these cases. Thus the incoming components can go straight from the buffer to the DataMover.

The LUT and register usage for $W_c = 8$ and $W_c = 16$ fluctuates considerably. When $W_c = 16$ and $N_c = 4$, or when $W_c = 8$ and $N_c = 8$, no LUTs or registers are used, because in this case there is no packing to perform. For the other values of N_c, variations are due to how efficiently the buffer can be implemented for the different combinations. For instance, if $W_c = 8$, then if $N_c = 4$, the buffer will always first fill up the lowest four components, and then in the next cycle the upper four components, and the resulting 64-bit word is forwarded to the DataMover. It appears that the synthesis tool can deduce this from its control flow analysis, and optimize the logic accordingly.

Module	LUTs	Registers
DataMover IP for MM2S	918	784
DataMover IP for S2MM	149	1527
Register Interface	629	497
TinyMover	241	181
S2MM Controller	82	82
Total	2019	3071

Table 4.3: Area usage of modules and IPs used in Cube DMA whose area is independent of generic parameters

W_c	N_c	LUTs			Regs		
		Controller	Unpacker	Total	Controller	Unpacker	Total
8	1	336	85	421	248	199	447
8	2	357	129	486	248	209	457
8	3	331	173	504	248	219	467
8	4	282	201	481	248	229	477
8	5	354	235	589	248	239	487
8	6	288	281	569	248	249	497
8	7	354	313	667	248	259	507
8	8	351	340	691	248	269	517
10	1	368	303	671	248	290	538
10	2	363	396	759	248	303	551
10	3	428	402	830	248	315	563
10	4	435	431	866	248	327	575
10	5	431	463	894	248	339	587
10	6	362	506	868	248	351	599
10	7	423	535	958	248	363	611
12	1	361	229	590	248	288	536
12	2	356	275	631	248	302	550
12	3	362	331	693	248	317	565
12	4	365	371	736	248	331	579
12	5	362	402	764	248	345	593
12	6	339	439	778	248	359	607
16	1	277	73	350	243	190	433
16	2	276	103	379	243	208	451
16	3	277	147	424	243	226	469
16	4	349	170	519	243	244	487
18	1	410,	517	927	233	299	532
18	2	340,	611	951	233	316	549
18	3	410	571	981	233	335	568
18	4	331	581	912	233	355	588

Table 4.4: MM2S channel controller and unpacker area utilization at varying component widths and number of components in parallel

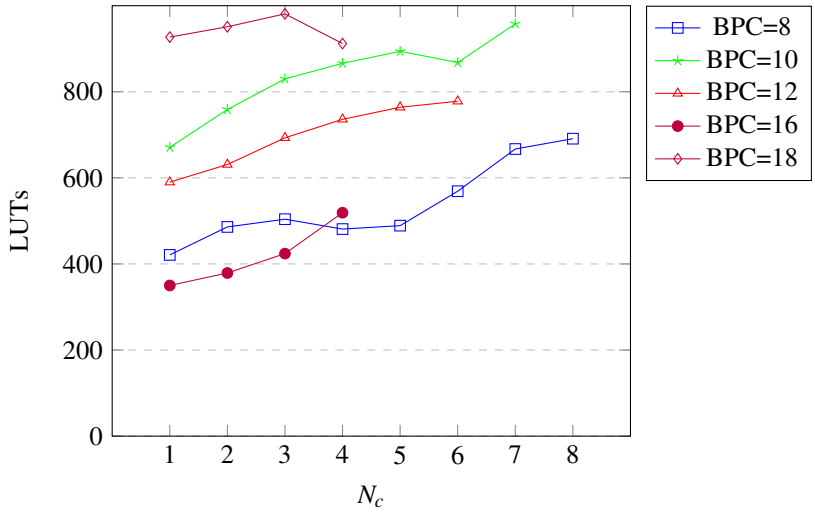

Figure 4.1: MM2S channel LUT usage for different bits per component W_c, as a function of the number of components in parallel N_c.

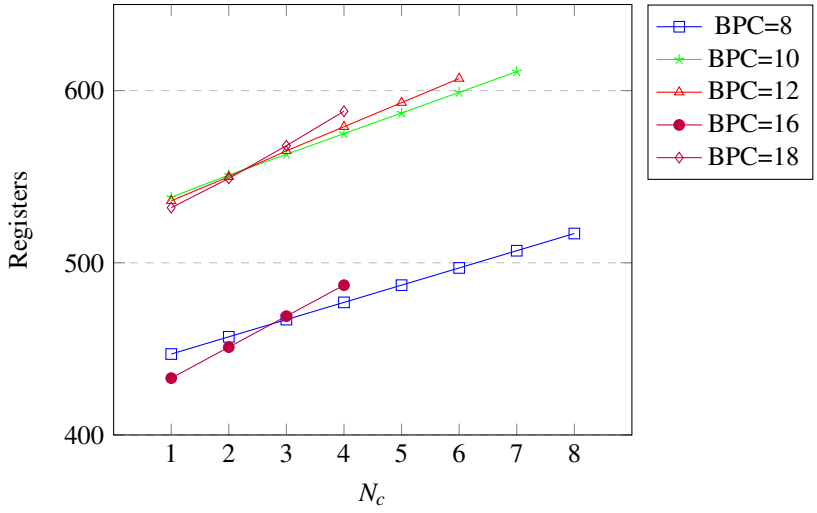

Figure 4.2: MM2S channel register usage for different bits per component W_c, as a function of the number of components in parallel N_c

W_c	N_c	LUTs	Registers
8	1	73	69
8	2	54	68
8	3	152	85
8	4	58	67
8	5	232	101
8	6	156	100
8	7	316	117
8	8	0	0
10	1	211	85
10	2	272	95
10	3	356	106
10	4	428	116
10	5	460	126
10	6	517	136
10	7	564	146
12	1	132	86
12	2	192	98
12	3	278	111
12	4	330	123
12	5	385	135
12	6	459	147
16	1	39	68
16	2	58	67
16	3	138	100
16	4	0	0
18	1	318	96
18	2	387	114
18	3	454	132
18	4	527	150

Table 4.5: S2MM channel packer area utilization at varying component widths W_c and number of components in parallel N_c

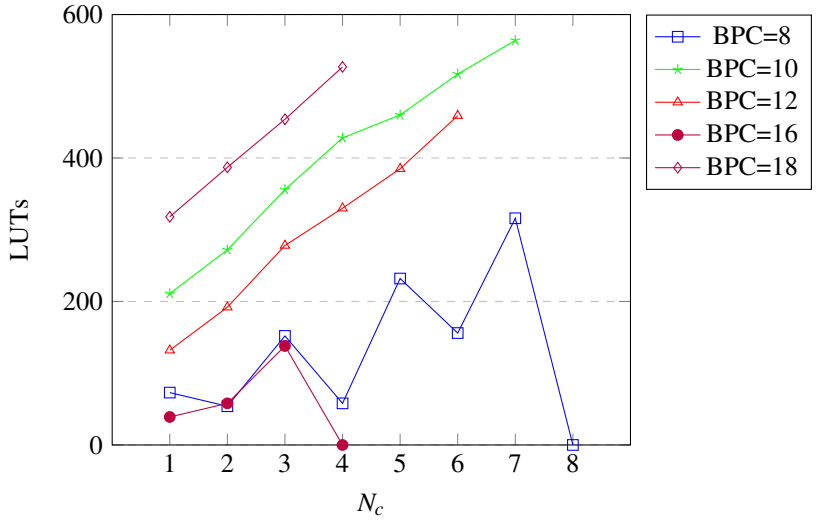

Figure 4.3: S2MM channel packer LUT usage for different bits per component W_c as a function of the number of components in parallel N_c

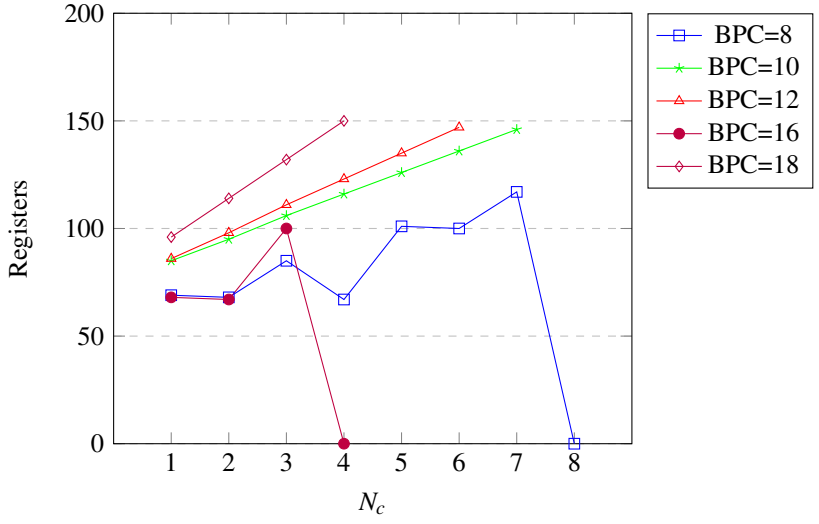

Figure 4.4: S2MM channel packer register usage for different bits per component as a function of the number of components in parallel

If a pure 64 bit stream is used for both the MM2S and S2MM channels (no unpacking or packing is performed), the total number of LUTs and registers used are 2355 and 3319, respectively. The Xilinx AXI DMA at comparable settings uses an estimated 2890 LUTs and 4046 registers [9].

4.3 Timing

In the project work leading up to this thesis, the MM2S channel was found to have a worst negative slack (WNS) of 1.988 ns when a constraint of 100 MHz is used, meaning that a maximum frequency of 124 MHz could be achieved. The critical path was within the address generation logic. With the changes made to the address generation logic and the new feautres implemented for this thesis, the critical path of the complete design is no longer in the address generation logic in the MM2S channel, but the received length calculation in the S2MM channel. The WNS is now an improved value of 2.396 ns, corresponding to a maximum frequency of 131 MHz. Further improvements are still possible but were not prioritized in this thesis. The largest gain in clock speed can be achieved by pipelining the additions that are performed in the address generation logic in the MM2S channel, and the received length calculation in the S2MM channel.

5

CCSDS123 theory and background

This chapter will present the CCSDS123 algorithm and explain all its computational steps, as well as introduce relevant terminology and mathematical notations. The bulk of the text is based on the CCSDS123 Recommended Standard document [4]. The other main source used is an Informational Report [12] from CCSDS, which provides a more practical insight into how the algorithm works, which trade-offs are important, and so on.

5.1 Overview

CCSDS123 is a compression algorithm standardized by the Consultative Committee for Space Data Systems (CCSDS), specifically designed for lossless compression of hyperspectral images. It is a formalized version of the *Fast Lossless* (FL) algorithm devised by the NASA Jet Propulsion Laboratory [13]. It is intended to be suitable for use on-board in spacecraft, with run-time complexity and memory usage low enough to make it feasible to implement in high-speed hardware [12].

The algorithm is lossless, meaning that the exact original image can be restored from the compressed image. The two main steps in the compression process are shown in Figure 5.1: prediction and encoding. The prediction stage computes estimates of each component based on previous components that are spatially and spectrally close in the cube. The difference between the estimates and the actual component values, so-called residuals, are encoded into variable length code words.

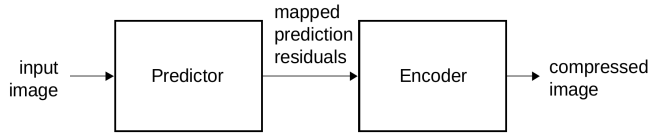

Figure 5.1: CCSDS123 compressor overview [4]

5.2 Definitions

This section will introduce some terminology and definitions that are used further when explaining the CCSDS123 compression scheme.

5.2.0.1 Samples

Individual components in a HSI cube are referred to as *samples* in the CCSDS123 standard documentation, and are denoted by s. The same terminology will be used when describing CCSDS123 and the implementations done in this work.

Samples have a bit resolution given by the parameter D in the range $2 \leq D \leq 16$. The lower sample value limit, the upper sample value limit and the mid-range sample value are denoted by $s_{\min}$, $s_{\max}$ and s_{mid}, respectively. Samples can be unsigned or signed integers. If unsigned samples are used,

$$s_{\min} = 0, \;\; s_{\max} = 2^D - 1, \;\; s_{\text{mid}} = 2^{D-1}.$$

If signed samples are used,

$$s_{\min} = -2^{D-1}, \;\; s_{\max} = 2^{D-1} - 1, \;\; s_{\text{mid}} = 0.$$

5.2.0.2 Cube size and coordinates

The HSI cube size is denoted by the symbols N_X, N_Y and N_Z.

Samples and other values that are computed during the compression process can be addressed by their x, y and z coordinates on the form $s_{z,y,x}$, meaning the sample in band z at spatial location (x,y). The indices start at 0, meaning that $0 \leq x \leq N_X - 1$ and similarly for y and z.

Samples can also be addressed on the form $s_z(t)$ where $t = y \cdot N_X + x$. The two representations are used interchangeably, depending on which fits best.

5.2.0.3 Mathematical notations

For any integer x and positive integer R, the function $\mod^*_R[x]$ is defined as

$$\mod^*_R[x] = ((x+2^{R-1}) \mod 2^R) - 2^{R-1}.$$

This is the mathematical description of what happens when the value x, represented in two's complement form, is stored in a register with R bits.

The notation $\text{clip}(x, \{x_{\min}, x_{\max}\})$ denotes clipping the value of x to be in the range $[x_{\min}, x_{\max}]$:

$$\text{clip}(x, \{x_{\min}, x_{\max}\}) = \begin{cases} x_{\min}, & x < x_{\min}, \\ x, & x_{\min} \leq x \leq x_{\max} \\ x_{\max}, & x > x_{\max} \end{cases}$$

Finally, the function $\text{sgn}^+(x)$ is defined by

$$\text{sgn}^+(x) = \begin{cases} 1, & x \geq 0 \\ -1, & x < 0 \end{cases}$$

5.3 Prediction

Prediction is based on values calculated from the neighboring previous samples to the current sample $s_{z,y,x}$. Figure 5.4 shows an overview of a typical prediction neighborhood.

5.3.1 Local sum and local difference vector

To simplify matters, neighboring previous samples in the same band as the current sample are named as shown in Figure 5.2, with NE denothing "North East", N denoting "North", NW denoting "North West" and W denoting "West".

5.3.1.1 Local sum

The neighboring samples in the band are used to compute a *local sum* $\sigma_{z,y,x}$. The local sum can be *neighbor oriented* or *column oriented*, which is illustrated in Figure 5.3. The neighbor oriented local sum is the sum of each of the W, NW, N and NE samples. For edge cases where the current sample is in the first or last row and/or column, special rules apply. These are shown in (5.1).

$$\sigma_{z,y,x} = \begin{cases} s_{z,y,x-1} + s_{z,y-1,x-1} + s_{z,y-1,x} + s_{z,y-1,x+1}, & y > 0, 0 < x < N_X - 1 \\ 4s_{z,y,x-1}, & y = 0, x > 0 \\ 2(s_{z,y-1,x} + s_{z,y-1,x+1}), & y > 0, x = 0 \\ s_{z,y,x-1} + s_{z,y-1,x-1} + 2s_{z,y-1,x}, & y > 0, x = N_X - 1 \end{cases} \tag{5.1}$$

The column-oriented local sum is just the previous sample in the same column (N), but weighted with 4 instead of 1 since it is the only sample in the sum. When $y = 0$ there is no previous sample in the same column, so the W sample is used instead. The local sum computation is summarised in (5.2).

$$\sigma_{z,y,x} = \begin{cases} 4s_{z,y-1,x}, & y > 0 \\ 4s_{z,y,x-1}, & y = 0, x > 0 \end{cases} \tag{5.2}$$

5.3.1.2 Local differences

Using the local difference $\sigma_{z,y,x}$, a *central local difference* $d_{z,y,x}$ is computed:

$$d_{z,y,x} = 4s_{z,y,x} - \sigma_{z,y,x}.$$

Similarly, three additional *directional local differences* are computed. These are labelled $d^N_{z,y,x}$, $d^{NW}_{z,y,x}$ and $d^W_{z,y,x}$, respectively. The directional differences are computed in the same way as the central local difference, but use the corresponding neighbor sample instead of $s_{z,y,x}$ as the first term in the subtraction. Again special rules apply in edge cases where the current sample is on one of the borders and the neighbor doesn't exist. The computations are shown in (5.3), (5.4) and (5.5).

$$d^N_{z,y,x} = \begin{cases} 4s^N_{z,y,x} - \sigma_{z,y,x}, & y > 0 \\ 0, & y = 0 \end{cases} \tag{5.3}$$

NW $s_{z,y-1,x-1}$	N $s_{z,y-1,x}$	NE $s_{z,y-1,x+1}$
W $s_{z,y,x-1}$	current $s_{z,y,x}$	

Figure 5.2: Neighboring samples in same band

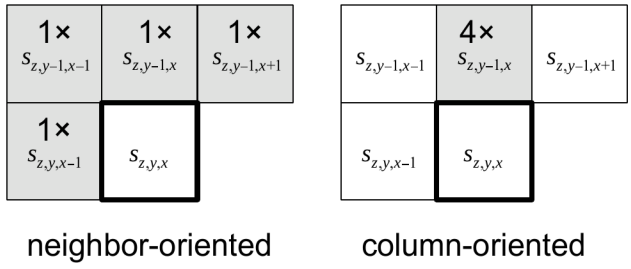

Figure 5.3: Neighbors used in local sum calculations in neighbor-oriented and column-oriented modes [4]

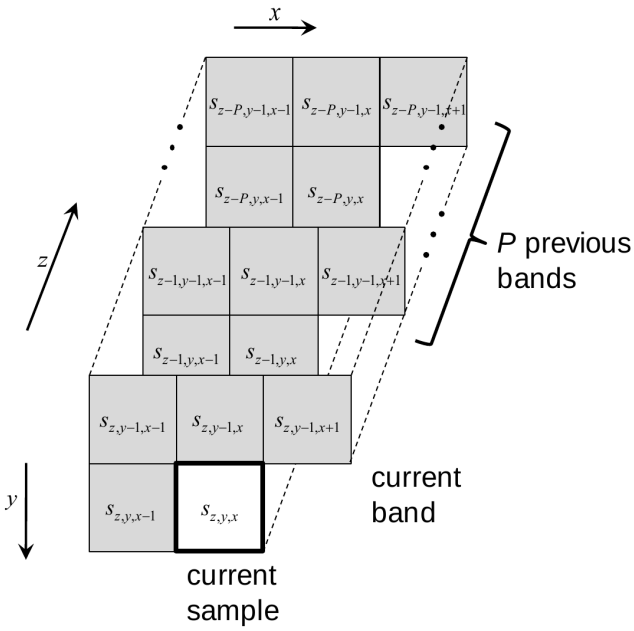

Figure 5.4: Prediction neighborhood in spatial and spectral dimensions [4]

$$d^N_{z,y,x} = \begin{cases} 4s^{NW}_{z,y,x} - \sigma_{z,y,x}, & x > 0, y > 0 \\ 4s^{N}_{z,y,x} - \sigma_{z,y,x}, & x = 0, y > 0 \\ 0, & y = 0 \end{cases} \tag{5.4}$$

$$d^N_{z,y,x} = \begin{cases} 4s^{W}_{z,y,x} - \sigma_{z,y,x}, & x > 0, y > 0 \\ 4s^{N}_{z,y,x} - \sigma_{z,y,x}, & x = 0, y > 0 \\ 0, & y = 0 \end{cases} \tag{5.5}$$

5.3.1.3 Local difference vector

The local differences from the current band and the P previous bands are collected into a *local difference vector*, $\mathbf{U}_z(t)$ which is used in the prediction calculation.

Which differences are included in the local difference vector depends on the prediction mode. In full prediction mode, $\mathbf{U}_z(T)$ is defined as

$$\mathbf{U}_z(t) = [d^N_z(t), d^W_z(t), d^{NW}_z(t), d_{z-1}(t), d_{z-2}, ..., d_{z-P^*_z}(t)]^T.$$

The notation P^*_z is shorthand for $\min\{z, P\}$. Put simply, the number of local central differences included from previous bands is P, unless $z < P$, in which case we must naturally stop at $z = 0$.

In reduced prediction mode, directional local differences are not included. $\mathbf{U}_z(t)$ is then defined as

$$\mathbf{U}_z(t) = [d_{z-1}(t), d_{z-2}, ..., d_{z-P^*_z}(t)]^T.$$

The number of elements in each local difference vector in band z is denoted C_z and is given by

$$C_z = \begin{cases} P^*_z, & \text{reduced prediction mode} \\ P^*_z + 3, & \text{full prediction mode} \end{cases}$$

5.3.2 Weights

Associated with each component in the local difference vector $\mathbf{U}_z(t)$ there is a corresponding weight that the component is multiplied with during prediction calculation. The *weight vector* $\mathbf{W}_z(t)$ contains the C_z weight values. In full mode,

$$\mathbf{W}_z(t) = [\omega^N_z(t), \omega^W_z(t), \omega^{NW}_z(t), \omega^{(1)}(t), \omega_{(2)}(t), ..., \omega_{(P^*_z)}(t)]^T,$$

and in reduced mode,

$$\mathbf{W}_z(t) = [\omega_{(1)}(t), \omega_{(2)}(t), ..., \omega_{(P^*_z)}(t)]^T.$$

The weight value bit resolution is determined by a parameter Ω in the range $4 \leq \Omega \leq 19$. The weight values are represented as signed integers, using $\Omega + 3$ bits.

For all pixels except $t = 1$, the weight vector $\mathbf{W}_z(t)$ is computed from $\mathbf{W}_z(t-1)$ using the weight update procedure shown in section 5.3.5. The weight vector $\mathbf{W}_z(1)$ however must be initialized to a starting value. This can be done in two ways: default weight initialization and custom weight initialization. Only the default initialization will be focused on in this thesis.

Default initialization is performed as shown in (5.6).

$$\omega_z^{(1)}(1) = \frac{7}{8}2^{\Omega}, \;\; \omega_z^{(i)}(1) = \left\lfloor \frac{1}{8}\omega_z^{(i-1)}(1) \right\rfloor \text{ for } i \in [2,3,...,P_z^*]. \tag{5.6}$$

5.3.3 Prediction calculation

The prediction calculation for sample $s_z(t)$ uses the local sum $\sigma_z(t)$, local difference vector $\mathbf{U}_z(t)$ and weight vector $\mathbf{W}_z(t)$ to produce a *scaled predicted sample*, $\tilde{s}_z(t)$ as shown in (5.7). The value $\hat{d}_z(t)$ is given by

$$\hat{d}_z(t) = \begin{cases} 0, & \text{reduced mode and } z = 0 \\ \mathbf{W}_z(t)^T \mathbf{U}_z(t), & \text{otherwise} \end{cases}$$

The scaled predicted sample is used in the weight update calculation. The predicted sample value $\hat{s}_z(t)$ is defined in (5.8. This value is passed on to the residual mapping calculation.

$$\tilde{s}_z(t) = \begin{cases} \text{clip}\left(\left\lfloor \dfrac{\text{mod}_R^* \left[\hat{d}_z(t) + 2^{\Omega}(\sigma_z(t) - 4s_{\text{mid}}) \right]}{2^{\Omega+1}} \right\rfloor \right), & t > 0 \\ 2s_{z-1}(t), & t = 0, P > 0, z > 0 \\ 2s_{\text{mid}}, & t = 0 \text{ and } (P = 0 \text{ or } z = 0) \end{cases} \tag{5.7}$$

$$\hat{s}_z(t) = \left\lfloor \frac{\tilde{s}_z(t)}{2} \right\rfloor \tag{5.8}$$

5.3.4 Residual mapping

The last stage of the prediction process is to calculate the *prediction residual* $\Delta_z(t)$ and perform a mapping from this residual to a *mapped residual* $\delta_z(t)$ which is the final output value from the predictor.

The prediction residual is the difference between the actual sample value, $s_z(t)$ and the predicted sample, $\hat{s}_z(t)$ as shown in (5.9). The residual mapping converts the signed prediction residual to an unsigned value that can be represented using D bits. The mapping is designed such that it maps small prediction residuals to small mapped values. This is done because the encoder generally will code small values with fewer bits [12]. The residual mapping calculation is shown in (5.10).

$$\Delta_z(t) = s_z(t) - \hat{s}_z(t) \tag{5.9}$$

$$\delta_z(t) = \begin{cases} |\Delta_z(t)| + \Theta_z(t), & |\Delta_z(t)| > \Theta_z(t) \\ 2|\Delta_z(t)|, & 0 \leq (-1)^{\tilde{s}_z(t)}\Delta_z(t) \leq \Theta_z(t) \\ 2|\Delta_z(t)| - 1, & \text{otherwise} \end{cases} \tag{5.10}$$

$$\Theta_z(t) = \min\{\hat{s}_z(t) - s_{\min}, s_{\max} - \hat{s}_z(t)\} \tag{5.11}$$

5.3.5 Weight update

After each prediction of sample $s_z(t)$, the weight vector $\mathbf{W}_z(t+1)$ for the sample in the next pixel in the same band can be calculated. This calculation is called a weight update. Put very simply, the weight update adjusts the weights based on the result of the previous prediction in such a way that if the predicted sample was larger than the actual sample, weight values are decreased and vice versa. This causes the weights to gradually adapt to the statistics of the image data.

How much the weights should change is determined by a *weight uppdate scaling exponent*, $\rho(t)$. Small values of $\rho(t)$ yield larger weight increments, which causes the weights to converge faster at the cost of reduced steady-state performance [12]. $\rho(t)$ is defined as

$$\rho(t) = \text{clip}\left(\nu_{\min}, + \left\lfloor \frac{t - N_x}{t_{\text{inc}}} \right\rfloor, \{\nu_{\min}, \nu_{\max}\}\right) + D - \Omega \tag{5.12}$$

The value of $\rho(t)$ changes during traversal of the image, and the three parameters $\nu_{\min}$, $\nu_{\max}$ and t_{inc} are used to control the value of $\rho(t)$ and how quickly it should change. $\rho(t)$ will start at the value $\nu_{\min} + D - \Omega$ and gradually increase at a rate given by t_{inc} until it reaches the maximum value $\nu_{\max} + D - \Omega$. This has the effect that the rate at which the weights adapt to image statistics becomes slower over time, yielding a trade-off between initial quick adaptation to image statistics and then gradually better steady state performance.

The weight update calculation is done as follows:

$$\mathbf{W}_z(t+1) = \text{clip}\left(\mathbf{W}_z(t) + \left\lfloor \frac{1}{2}\left(\text{sgn}^+[e_z(t)] \cdot 2^{-\rho(t)} \cdot \mathbf{U}_z(t) + \mathbf{1}\right)\right\rfloor, \{\omega_{\min}, \omega_{\max}\}\right) \tag{5.13}$$

$e_z(t)$ is the *scaled prediction error*, defined as

$$e_z(t) = 2s_z(t) - \tilde{s}_z(t) \tag{5.14}$$

5.4 Encoding

There are two different encoders defined by the CCSDS123 standard, the *sample adaptive encoder* and the *block adaptive encoder*. The block adaptive encoder was originally defined for a previous CCSDS 120 algorithm, but is included as an option in CCSDS123 so that implementers can re-use existing space-qualified hardware implementations [12]. In this thesis, the sample adaptive encoder will be used.

5.4.1 Golomb-Power-Of-2 coding

The sample adaptive encoder assigns variable-length code words of maximum size $D + U_{\max}$ to each incoming mapped residual. The coding technique used is called length-limited Golomb-Power-Of-2 (GPO2), which is also used in JPEG-LS and other compression schemes [12]. $U_{\max}$ is called the *unary length limit*, and is a variable parameter in the range $8 \leq U_{\max} \leq 32$.

For a fixed $U_{\max}$ there are several code word mappings available depending on the value of an integer $k \leq D-2$. When k is fixed, the mapping from input residual δ to code word is found by writing δ as

$$\delta = u \cdot 2^k + r,$$

where

$$u = \left\lfloor \frac{\delta}{2^k} \right\rfloor, \quad r = \delta \mod 2^k.$$

The code word generally has two components:

- If $u < U_{\max}$: A *unary encoding* of u (u number of 0s followed by a 1) followed by the k lowest bits of r.
- If $u \geq U_{\max}$: $U_{\max}$ number of 0s followed by the D-bit representation of δ.

.

5.4.2 Adaptation to image statistics

For a given $U_{\max}$, which GPO2-mapping (value of k) gives the smallest codes depends on characteristics of the incoming data. The sample adaptive encoder adaptively changes the value of k based on image statistics, so that a mapping that fits the incoming data better is chosen. This is performed by keeping track of the average value of the input residuals in each band. The average calculation is performed by accumulating the sample

values separately for each band and dividing by the number of samples that have been processed.

More concretely, there is an accumulator $\Sigma_z(t)$ for each band that accumulates the incoming residual values in that band. A counter $\Gamma(t)$ common to every band is incremented once per pixel. The ratio $\Sigma_z(t)/\Gamma(t)$ is the average value of the incoming residuals in band z. The counter has an initial value given by

$$\Gamma(1) = 2^{\gamma_0},$$

where the *initial count exponent*, γ_0, is a parameter in the range $1 \leq \gamma_0 \leq 8$. Each band's accumulator also has an initial value given by

$$\Sigma_z(1) = \left\lfloor \frac{1}{2^7}\left(3 \cdot 2^{K+6} - 49\right)\Gamma(1) \right\rfloor,$$

where the *accumulator initialization constant*, K, is a parameter in the range $0 \leq K \leq D-2$.

The counter's maximum value is given by the *re-scaling counter size* parameter γ^*. The counter's maximum value is $2^{\gamma^*}-1$. When the counter reaches this maximum value, the accumulators in each band and the counter are *re-scaled* by dividing their values by 2. This is done to make more recent sample values have more impact on the mean [12].

The accumulator and counter logic can be summed up as follows:

$$\Sigma_z(t) = \begin{cases} \Sigma_z(t-1) + \delta_z(t-1), & \Gamma(t-1) < 2^{\gamma^*}-1 \\ \left\lfloor \dfrac{\Sigma_z(t-1) + \delta_z(t-1) + 1}{2} \right\rfloor, & \Gamma(t-1) = 2^{\gamma^*}-1 \end{cases} \tag{5.15}$$

$$\Gamma(t) = \begin{cases} \Gamma(t-1) + 1, & \Gamma(t-1) < 2^{\gamma^*}-1 \\ \left\lfloor \dfrac{\Gamma(t-1)+1}{2} \right\rfloor, & \Gamma(t-1) = 2^{\gamma^*}-1 \end{cases} \tag{5.16}$$

The value of k for a given sample, $k_z(t)$, is selected based on the accumulator and counter as follows:

$$k_z(t) = \begin{cases} 0, & 2\Gamma(t) > \Sigma_z(t) + \left\lfloor \dfrac{49}{2^7}\Gamma(t) \right\rfloor \\ \max\left\{ 1 \leq i \leq D-2 \mid \Gamma(t)2^i \leq \Sigma_z(t) + \left\lfloor \dfrac{49}{2^7}\Gamma(t) \right\rfloor \right\}, & \text{otherwise} \end{cases} \tag{5.17}$$

5.4.3 Encoding of a sample

For the encoding to be reversible, the decoder needs to know at every step which mapping (k value) has been used to encode a sample. To do this it needs to repeat the same

calculations for k as the encoder has done, which means that it needs to use the same accumulator values as the encoder. For this to work the first mapped residuals $\delta_z(0)$ in each band of the compressed image must be stored uncoded in their original D-bit representation, so that the decoder has the same initial starting point as the encoder. The remaining residuals ($t > 0$) are coded using length-limited GPO2 as described previously. To summarize:

- If $t = 0$, the code word is the D-bit unsigned integer binary representation of $\delta_z(t)$
- If $t > 0$ and $u_z(t) < U_{\max}$, the code word consists of $u_z(t)$ '0's followed by '1', followed by the $k_z(t)$ least significant bits of $\delta_z(t)$
- If $t > 0$ and $u_z(t) \geq U_{\max}$, the code word consists of $U_{\max}$ '0's, followed by the D-bit unsigned integer binary representation of $\delta_z(t)$

5.5 Summary of parameters

Table 5.1 shows a summary of the selectable parameters for the CCSDS123 compression algorithm that have been introduced in the previous sections.

Parameter	Description	Range
Predictor parameters		
D	Sample bit resolution	$[2, 16]$
P	Number of previous bands to use in prediction	$[0, 15]$
Ω	Weight resolution (weight bit resolution is $\Omega + 3$)	$[4, 19]$
$\nu_{\min}$	Weight update scaling exponent initial parameter	$[-6, \nu_{\max}]$
$\nu_{\max}$	Weight update scaling exponent final parameter	$[\nu_{\min}, 9]$
t_{inc}	Weight update scaling exponent change interval	Power of two, $[2^4, 2^{11}]$
R	Register size	$[\max\{32, D+\Omega+2\}, 64]$
Encoder parameters		
$U_{\max}$	Unary length limit	$[8, 32]$
γ_0	Initial count exponent	$[1, 8]$
γ^*	Re-scaling counter size	$[\max\{4, \gamma_0\}, 9]$
K	Accumulator initialization constant	$[0, D-2]$

Table 5.1: Selectable parameters for the CCSDS123 compressor

6

Hardware implementation of CCSDS123 compressor

This chapter will detail the implementation of a CCSDS123 compressor for FPGA in VHDL. The chapter will start with an analysis of the various tradeoffs that must be considered when designing a CCSDS123 compressor, followed by a presentation of previous hardware implementations. The bulk of the chapter will be detailing two implementations of CCSDS123: First, a *serial* implementation consisting of one processing pipeline that can compress one image sample per clock cycle is presented, with detailed descriptions of each stage in the pipeline. Following this, a parallelized version will be presented where several pipelines are compressing samples in parallel.

6.1 Memory and performance trade-offs

CCSDS123 supports compression in all three of the common sample orderings, BIP, BIL and BSQ. When using the sample adaptive encoder, the choice of ordering has no effect on the compressed image size, because the compression process is completely separate for each band, and for each of the sample orderings pixels are processed in an upper-left to lower-right fashion in each band. The only difference between sample orderings when using the sample adaptive encoder is the ordering of the encoded samples in the output bitstream. Sample ordering can therefore be chosen such that it fits best with the larger system (image sensor, from memory streaming) and such that it optimizes resource usage or parallelization. In the subsequent sections several tradeoffs between the different sample orderings will be considered:

- Input image sample ordering with regards to streaming

- Space requirements for weights, previous local differences and accumulators
- Pipelining and parallelization limitations

6.1.1 Streaming efficiency

In a typical HSI system, samples are either streamed from a memory or directly from a camera sensor pipeline (on the fly). In the case of streaming from a camera sensor, the native sample ordering from the sensor essentially dictates the sample ordering chosen for the CCSDS123 implementation. Converting on-the-fly between BIP and BIL orderings can be performed with relatively small memory needs (storage is needed for approximately one line of image data to buffer up full pixels), but converting between BI orderings and BSQ is not feasible to do on-the-fly, as it would require a whole cube to be buffered.

In the case of streaming from memory, there is more leeway since memory access patterns can be changed to achieve the wanted sample ordering regardless of how samples are ordered in memory. However, as Chapter 2 showed, this comes with performance penalties related to the memory subsystem's ability to do strided accesses.

6.1.2 Local space requirements

6.1.2.1 Neighboring samples

When encoding a sample $s_{z,y,x}$, the neighboring previous samples must also be available in order to perform the local sum and difference calculations. This means that some memory must be used to store the samples from the cycle where they are first streamed until the cycle where they are used as neighbors.

Table 6.1 shows how many cycles this takes for different sample orderings.

Order	W	NE	N	NW
BIP	N_z	$(N_x-1)N_z$	N_xN_z	$(N_x+1)N_z$
BIL	1	N_xN_z-1	N_xN_z	N_xN_z+1
BSQ	1	N_x-1	N_x	N_x+1

Table 6.1: Number of samples from the current sample to the previous samples.

6.1.2.2 Weight vectors and accumulators

Each band has its own weight vector used in prediction and accumulator used in the encoder. These must also be stored in memory while samples from other bands are being processed. In the case of BIP ordering, $s_z(t+1)$ will be processed N_z samples after $s_z(t)$, which means that the weight vector and accumulator for each band must be

stored in memory. This is also the case for BIL ordering. Under BSQ ordering however, $s_z(t+1)$ is processed immediately after $s_z(t)$, and there is no need to store more than one weight vector or accumulator.

6.1.2.3 Previous local differences

Another source of memory usage is the need to store the central local differences computed in the P previous bands of the same pixel. Under BIP ordering, these were computed during the P previous cycles. Under BIL ordering, they were computed in the P previous strips of samples, the least recent PN_x cycles previously. Under BSQ ordering, the whole plane is processed before starting on the next, meaning that the most recent local difference was produced N_xN_y cycles previously, and the least recent PN_xN_y cycles previously. These are summarised in Table 6.2.

Order	$z-1$	$z-2$	...	$z-P$
BIP	1	2		P
BIL	N_x	$2N_x$		PN_x
BSQ	N_xN_y	$2N_xN_y$		PN_xN_y

Table 6.2: Number of samples from the current sample to sample from the same pixel in the previous band

6.1.2.4 Summed up

Table 6.3 shows a summary of memory requirements for different sample orderings. Which sample ordering requires most memory, depends somewhat on the image sizes used. If the spatial dimensions N_x and N_y are relatively small but the number of bands is large, BSQ ordering might be the best choice, but for most images where spatial dimensions are larger and the number of bands N_z is smaller, it is clear that BSQ ordering requires huge amounts of memory, even for small P. For example, the HICO images have the size $512 \times 2000 \times 128$ with 16 bits per sample, and with a typical value of $P = 3$, 6.95 MB of storage is required just to store the previous local differences. This alone exceeds the total available block RAM of 4.9MB in a mid-range Zynq-7020 SoC. For larger P it quickly grows past the available block RAM capacity also in high end devices.

Order	Previous samples	Local differences	Weights vectors	Accumulator
BIP	$(N_x+1)N_zD$	$P(D+3)$	$N_zC_z(\Omega+3)$	$N_z(D+\gamma^*)$
BIL	$(N_xN_z+1)D$	$PN_x(D+3)$	$N_zC_z(\Omega+3)$	$N_z(D+\gamma^*)$
BSQ	$(N_x+1)D$	$PN_xN_y(D+3)$	$C_z(\Omega+3)$	$D+\gamma^*$

Table 6.3: Memory usage comparison between sample orderings

6.1.3 Pipelining and parallelization

The limiting factor for pipelined and parallel operation of the CCSDS123 algorithm is the fact that calculating the predicted sample for $s_z(t+1)$ requires the weight vector $\mathbf{W}_z(t+1)$ which is calculated from the prediction of $s_z(t)$. This means that the prediction calculation for $s_z(t+1)$ cannot be started simultaneously with the calculation for $s_z(t)$. This rules out parallelization for BIL and BSQ sample orderings, and it also limits how often new samples can be accepted by the core. A completely pipelined design with one sample accepted per clock cycle would be hard to implement, as it would require the whole prediction and weight update process to be done in one clock cycle.

For BIP ordering, this is not an issue since the sample following $s_z(t)$ is $s_{z+1}(t)$, which has no relation with $s_z(t)$ and can be processed simultaneously with $s_z(t)$. This also means that a pipelined serial implementation which accepts one new sample per clock cycle is easy to achieve.

6.2 Previous work

Several hardware implementations of CCSDS123 have been done previously, with different optimization goals in focus. A summary of previous implementations is shown in Table 6.4.

Implementation	Order	Memory	On-the-fly
Keymeulen et al [13]	BIP	Internal	Unknown
Santos et al [14]	BSQ	Internal, multiple access	No
Bascones et al [15]	BIP	Internal	Yes
Theodorou et al [16]	BIP	External and internal	No

Table 6.4: Previous CCSDS123 implementations

The oldest implementation looked at is by Keymeulen et al [13], which is the first hardware implementation of the CCSDS123 standard. It assumes BIP ordering of incoming samples and is capable of compressing one sample per clock cycle. Although the paper shows incoming samples coming from a DDR memory, the authors also make it clear that samples are read once in a row by row fashion, and hence this implementation could be used for on-the-fly processing. The core stores weigths, accumulators and other temporaries within the core.

Santos et al [14] present an implementation using BSQ sample ordering, with the main focus being on low complexity and low memory footprint. As is seen in Table 6.3, BSQ ordering has big advantages in the space needed to store previous samples as well as just needing to store one weight vector and one accumulator, but local differences for each sample in the P previous bands must be stored. The paper presents an approach where local differences don't have to be stored but are re-calculated when needed, and

as such the implementation achieves very low memory usage. However, the price is paid in input bandwidth efficiency, because the approach requires each sample to be read $2(P+1)$ times [14]. In addition, arranging the input stream in such a way that samples are repeated requires either the memory access pattern to be non-sequential, which can potentially reduce streaming efficiency, or that the data is arranged in memory in the desired way, which would require $2(P+1)$ as much storage. In addition, reading each sample several times during the compression process prohibits on the fly compression of image data coming straight from the image sensor.

The most high performant serial CCSDS123 implementation that has been found is proposed by Theodorou et al[17]. It uses BIP ordering and is capable by compressing one sample per clock cycle. It relies on external DDR memory to buffer samples coming from the image sensor such that the current sample as well as the North and North East neighboring samples can be streamed in parallel into the core. This greatly reduces the amount of on-chip memory needed, since previous sample storage is the largest contributor to total memory usage for BIP ordering, as seen in Table 6.3. The downside to this is however that the implementation does not support direct on the fly compression, at least not without some extra logic dealing with streaming to and from the DDR memory.

Another implementation using BIP ordering is proposed by Báscones et al [15]. The ability to perform compression without relying on external memory is highlighted in this paper. This is achieved by queuing incoming samples in FIFOs internally in the FPGA to obtain the neighboring samples. Local memory requirements for this implementation is as shown in the BIP row of Table 6.3; memory usage scales linearly with $N_x \cdot N_z$. The advantage of this solution is the ability to perform on the fly compression, since it compresses samples sequentially and requires each sample to be read only once.

Báscones et al also propose a parallel CCSDS123 implementation [18]. It naturally uses BIP ordering, since that is the only ordering in which data dependencies between subsequent samples can be avoided. The architecture proposed in this paper consists of C instances of the same CCSDS core, each processing their own sample. Local differences are shared between the cores, since e.g. the core handling s_{z+1} will need the local central difference d_z produced by the core that is handling s_z. Other than sharing local differences, the cores are operating independently of one another, each with its own weight storage, sample storage and so on. This means that each CCSDS core must handle samples from a fixed subset of bands to avoid sharing of weights and accumulators between cores. For instance, core 0 must handle s_0, s_C, s_{2C} and so on, while core 1 must handle s_1, s_{C+1}, s_{2C+1} and so on. This has the implication that if N_z is not divisible by C, some of the cores will be unused when the last samples of each pixel are being processed, which is pointed out in the paper with the recommendation that C is selected such that it divides N_z.

Another limitation of this parallel design is that even though C CCSDS cores are operating in parallel, the proposed design as a whole is not fully parallel because packing the resulting code words from each core into blocks is done serially. The paper does point out that the serial packing circuit can be clocked faster, but nonetheless this represents a throughput bottleneck for large C.

6.3 Existing software implementations

When developing a hardware implementation of an algorithm, having a software implementation to compare results with is a great advantage. There are at least two software implementations of CCSDS123 that have been considered when work on this thesis commenced.

The European Space Agency (ESA) have a reference software implementation available for download on their website [19]. This implementation is written in C and is open source. Being a reference implementation, it supports all input sample orderings (BIP, BIL, BSQ), output orderings, and all the selectable parameters listed in Table 5.1.

Another implementation is *Emporda*, an implementation created by the Group on Interactive Coding of Images (GICI) at the Autonomous University of Barcelona [20]. The Emporda implementation is also very complete, covering all possible parameters, compression orders, sample formats, and so on. It is written in Java and is also open source.

From a performance stand point, the reference implementation is somewhat faster as it is written in C, but the code in the Emporda implementation seems more organized and easier to understand. Emporda was therefore chosen as the implementation to use during development of the hardware core. Throughout the development process, it turned out to be easy to add debug printing statements in Emporda to show intermediate values during various computations and comparing those to signal values in simulation. Emporda was also used to create an automatic test system that will be detailed in Chapter 8.

6.4 Serial implementation

In the SmallSat project, the image sensor is scanning the image in a push broom fashion, meaning that all spectral components of one line of pixels are captured at once. If image data is to be compressed using CCSDS123 directly from the image sensor, this means that either BIL or BIP ordering must be used. BSQ ordering would require the whole cube to be captured and stored before compression can start. For the initial serial implementation of the CCSDS123 algorithm, BIP was chosen because it makes it much easier to later develop a parallelized version of the algorithm.

An overview of the CCSDS123 implementation is shown in Figure 6.1. Each box in this figure represents a module, each of which will be detailed in the next sections.

Figure 6.2 show how pipeline operations are scheduled.

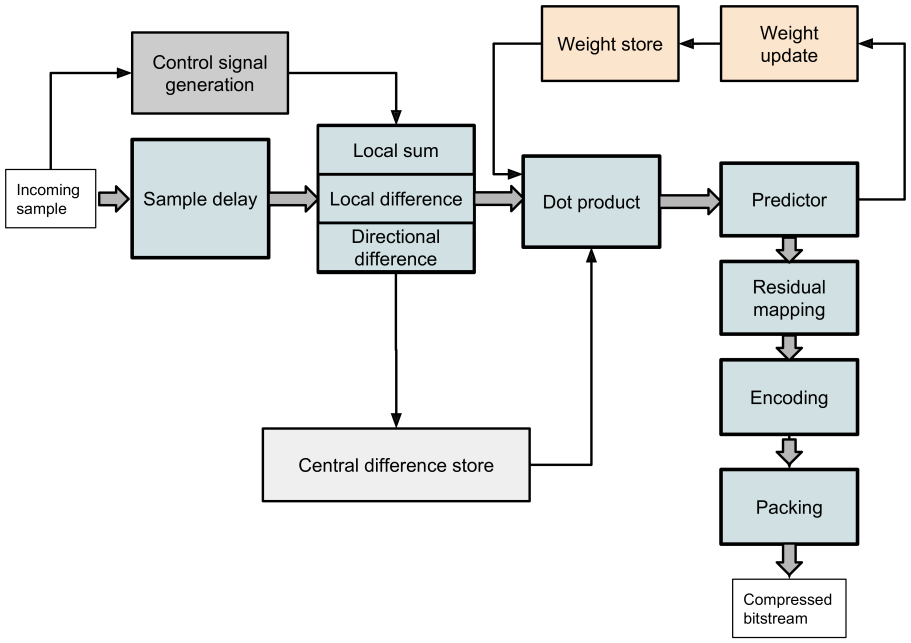

Figure 6.1: Overview of the CCSDS123 BIP implementation. Bold arrows show the main data path. Each box represents a VHDL module

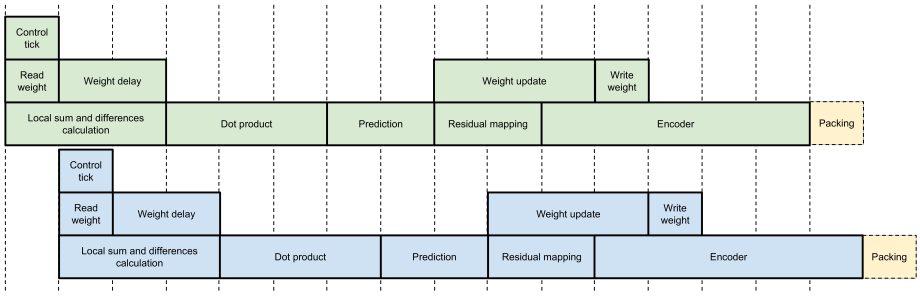

Figure 6.2: Scheduling of pipeline operations. The dashed bars indicate clock cycles. Packing into 64 bit output words takes a variable number of clock cycles depending on the size of encoded words.

6.4.1 Control signal generation

In addition to the actual data passed between the different modules, some control information is passed along as well.

A set of counters is used to keep track of the x, y and z coordinates of the current sample that is coming in. The counter values are used to generate the following control signals:

- z value, the band of the current sample
- Flags indicating if the pixel is in the first line, the first pixel in a line, last in a line or the last of all
- Weight update scaling exponent $\rho(t)$, used in the weight update calculation

6.4.2 Sample delay

The sample delay module takes care of delaying incoming samples in such a way that at each clock cycle, the current, as well as the neighboring previous samples are available. This is achieved by chaining together FIFOs of particular lengths, as shown in Figure 6.3.

As an example, the number of samples between a given sample and the one in the same band but one pixel to the left, is exactly N_Z. This means that if $s_{z,y,x-1}$ is pushed into a FIFO of depth N_Z, it will be present at the FIFO's output N_Z cycles later, in the same cycle that $s_{z,y,x}$ is read.

The sample delay is the most memory consuming part of the CCSDS123 implementation. The amount of memory needed in bits is

$$\text{RAM bits} = D(3 \cdot N_Z + (N_X - 2) \cdot N_Z) = D(N_X + 1)N_Z.$$

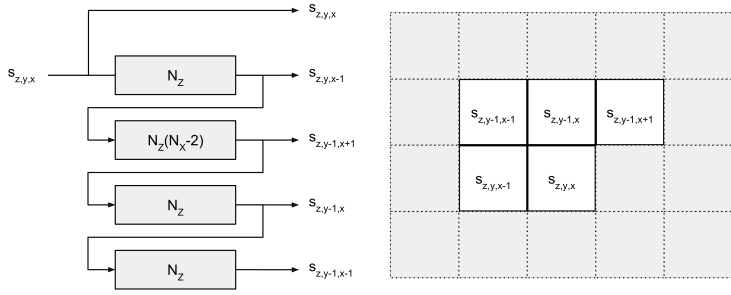

Figure 6.3: Sample delay FIFOs. Each box represents a FIFO with a depth of exactly the number shown on each box.

6.4.3 Local sum and difference calculations

The local sum and difference calculations are performed in a three stage pipeline, shown in Figure 6.4. The first two stages compute the local sum value, while the last stage computes the local central difference, $d_{z,y,x}$ and the directional differences, $d^{NW}_{z,y,x}$, $d^{W}_{z,y,x}$, $d^{N}_{z,y,x}$.

The local sum calculation shown in (5.1) is split across two pipeline stages to reduce delay. In the first stage, two values, term_1 and term_2 are calculated as shown in the following equations.

$$\text{term}_1 = \begin{cases} s_{z,y,x-1} + s_{z,y-1,x-1}, & y > 0, 0 < x < N_X - 1 \\ 4s_{z,y,x-1}, & y = 0, 0 < x < N_X - 1 \\ 2s_{z,y-1,x}, & y > 0, x = 0 \\ s_{z,y,x-1} + s_{z,y-1,x-1}, & y > 0, x = N_X - 1 \end{cases}$$

$$\text{term}_2 = \begin{cases} s_{z,y-1,x} + s_{z,y-1,x+1}, & y > 0, 0 < x < N_X - 1 \\ 0, & y = 0, 0 < x < N_X - 1 \\ 2s_{z,y-1,x+1}, & y > 0, x = 0 \\ 2s_{z,y-1,x}, & y > 0, x = N_X - 1 \end{cases}$$

In the next pipeline stage, the local sum is produced by summing term_1 and term_2.

In the last pipeline stage, the local central and directional differences are calculated.

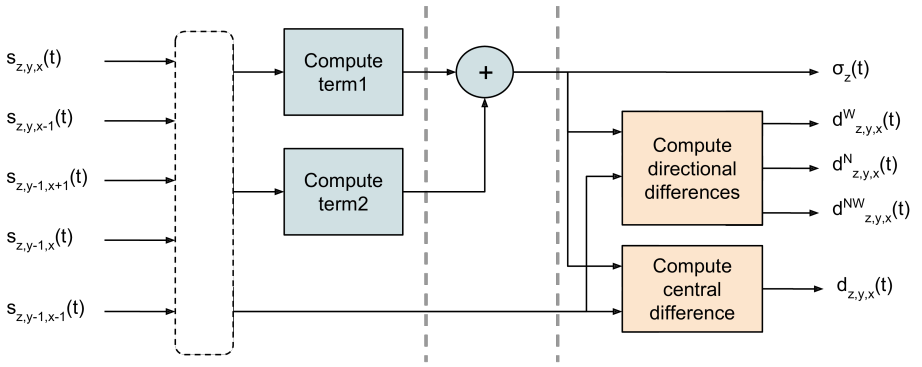

Figure 6.4: Local sum, local difference and central difference calculations

6.4.4 Central difference store

The central difference store keeps the local central differences computed in the previous P bands, since these are needed together with the directional differences to form the local difference vector $\mathbf{U}_{z,y,x}$.

The local differences are stored in a shift register, as shown in Figure 6.5. Each cycle, the local central difference just computed is stored in the first register, while the previous contents is shifted one step. When $z = N_z - 1$, the contents of the shift register is zeroed, such that prediction of a new pixel doesn't use local differences from the previous pixel.

6.4.5 Weight store

The weight store keeps the weight vectors in between updating the weights for band z and reading the same band again for the next pixel. Weight vectors are stored in a block RAM, in the order corresponding to the band they belong to, meaning that the first index in the RAM is the weight vector for $z = 0$, the next is for $z = 1$ and so on. A dual port block RAM is used to be able to read a weight vector from one location while simultaneously updating the weight value at another.

The z value of the incoming sample is used as an address to read the corresponding weight from the weight store. Similarly, when updating weights, the z value of the new weight vector is used as a write address. Figure 6.6 illustrates the situation when the weight vector $W_{z_{in}}(t)$ in band z_{in} is read and the weight vector $W_{z_{update}}(t+1)$ is being updated. The band z_{update} is related to the band z_{in} of the currently processed sample as:

$$z_{update} = (z_{in} - N) \bmod N_z, \tag{6.1}$$

where $N = 1 + 2 + S + 2 + 3$ is the delay which corresponds to the number of pipeline stages from the weight reading operation to the end of the weight update operation. The value of N depends on the number of tree adder stages S in the dot product. The time diagram of the described pipeline is given in Figure 6.7.

Reading a weight vector takes one clock cycle. The read is initiated when a new input sample is handshaked, and is done in parallel with the local sum and difference compu-

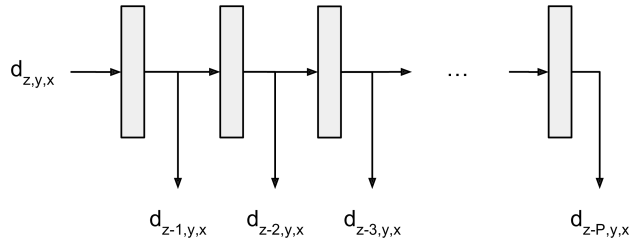

Figure 6.5: Local difference store

tations. Because these computations take three clock cycles, it is necessary to delay the read weight vector from the RAM by two cycles, such that the local difference vector and weight vector for the same sample arrive at the dot product module simultaneously, as shown in the timing diagram in Figure 6.7. This is taken into account by providing the weight store with a variable length shift register which will delay the data by a given number of cycles set by a generic constant in the module.

Since it takes N cycles from a weight has been read until the updated weight is stored, the number of bands in the image, N_z, can generally not be less than N unless the input stream is stalled such that the updated weights for a given sample are stored before attempting to read them. This is of course only possible for from-memory streaming; for on-the-fly processing it is not an option to stall the stream, so in this case we must have $N_z \geq N$. For most hyperspectral imagers, this is not an issue as N_z is much greater than N.

6.4.6 Dot product

The dot product is performed in a pipeline where the first stage multiplies each element in $\mathbf{U}_z(t)$ with the corresponding element in $\mathbf{W}_z(t)$, and the following stages make up a tree of adders that computes the sum of the multiplication results. Figure 6.8 shows the structure of the dot product module.

The number of stages needed in the adder tree is given by $S = \lceil \log_2(C_z) \rceil$. When the number of elements is a power of 2, the adder tree is a perfect binary tree, and it can be described as follows:

$$s(2^S + i) = s(2i) + s(2i+1) \text{ for } 0 \leq i \leq 2^S - 2,$$

where the initial indices are the multiplications:

$$s(i) = u_i \cdot \omega_i \text{ for } 0 \leq i \leq 2^S - 1.$$

The result taken from the last index:

$$\hat{d}_z = s(2^{S+1} - 2).$$

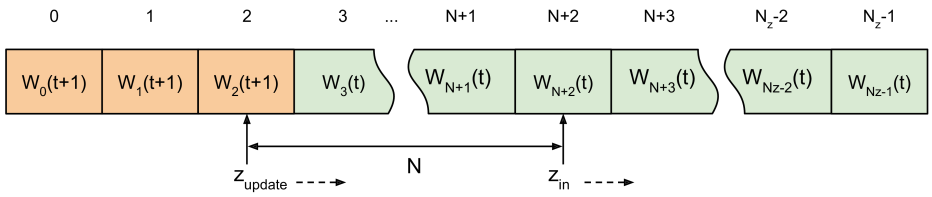

Figure 6.6: The state of the weight store when current input sample is $s_{N_z-3}(t)$

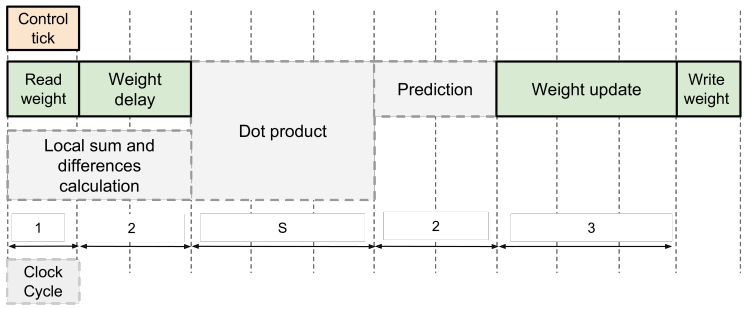

Figure 6.7: Timing diagram of pipeline operations from reading a weight vector until writing back the updated weight vector

For the case when C_z is not a power of 2, the tree is no longer a perfect binary tree, but it can still be described as one where the elements $s(i)$ for $i \geq C_z$ are set to 0. This is illustrated in Figure 6.9 for the case where $C_z = 3$. The benefit of this approach is that the adder tree is described in the same way no matter the value of C_z. Even though unnecessary registers and adders are included, the synthesis tool is able to infer that the 0 inputs will have no impact and removes them.

6.4.7 Predictor

The predictor computes the scaled predicted sample, as defined in (5.7). This calculation is split across two pipeline stages, where the first stage computes the numerator in the fraction that is part of the argument to the clip function:

$$\text{temp}_1 = \text{mod}_R^* \left[\hat{d}_z(t) + 2^{\Omega}(\sigma_z(t)) \right]$$

Multiplying by a power of 2 is implemented by shifting. The term $4s_{\text{mid}}$ in (5.7) is removed since this is an implementation using signed numbers, hence $s_{\text{mid}} = 0$.

In the next stage, the scaled predicted value is computed. For the case where $t > 0$, the computation performed is

$$\tilde{s}_z(t) = \text{clip}\left(\left\lfloor \frac{\text{temp}_1}{2^{\Omega+1}} \right\rfloor + 1, \{2s_{\min}, 2s_{\max} + 1\} \right),$$

where the floor value of the division by $2^{\Omega+1}$ is performed by right shifting.

For the case where $t = 0$ and $P > 0, z > 0$, the sample value from the previous band is taken from a register that stores the sample from the previous calculation.

A multiplexer finally chooses between one of the three cases in (5.7).

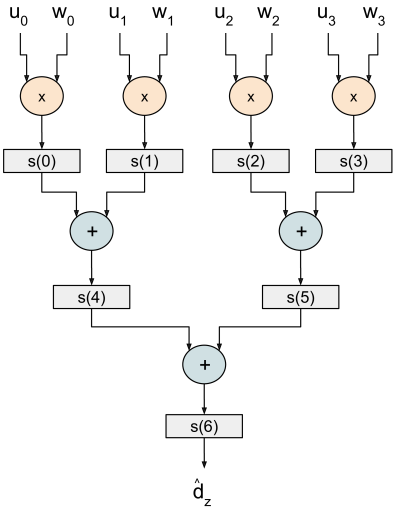

Figure 6.8: Dot product when $C_z = 4$

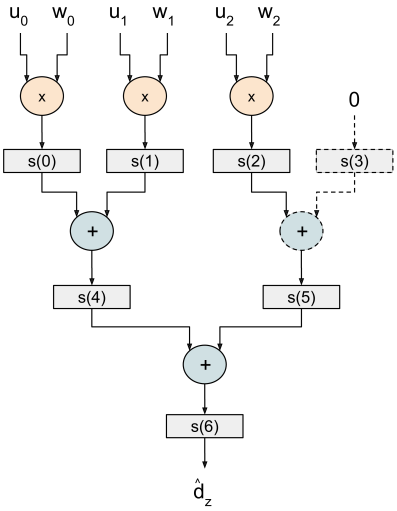

Figure 6.9: Dot product when $C_z = 3$. The dashed multiplication and sum are removed by the synthesis tool during elaboration.

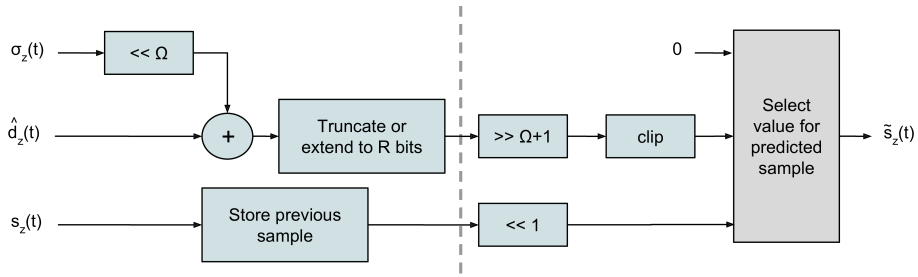

Figure 6.10: Predictor implementation

6.4.8 Weight update

The weight update computation shown in (5.13) is performed in three pipeline stages. The weight update scaling exponent, $\rho(t)$, is already computed in the Control signal generation and sent along with the pixel.

The first stage of the computation is the component-wise product

$$\mathbf{temp}_1 = \mathrm{sgn}^+[e_z(t)] \cdot \mathbf{U}_z(t),$$

This product is equivalent to changing the sign of each component depending on whether $e_z(t)$ is positive or negative. Since $e_z(t)$ is defined as $2s_z(t) - \hat{s}_z(t)$, this is equivalent to changing the sign when $2s_z(t) < \hat{s}_z(t)$, and otherwise performing no change to $\mathbf{U}_z(t)$. This is implemented by computing both $\mathbf{U}_z(t)$ and $-\mathbf{U}_z(t)$ and choosing based on the result of the comparison.

The second stage computes

$$\mathbf{temp}_2 = \left\lfloor \frac{1}{2}\left(2^{-\rho(t)}\mathbf{temp}_1 + \mathbf{1}\right) \right\rfloor.$$

Since $\rho(t)$ has a fairly small range of possible values (at most -6 to 9), the values of $2^{-\rho(t)}\mathbf{temp}_1$ resulting from each possible value of $\rho(t)$ are calculated in parallel, and then a multiplexer chooses which one to use based on the actual value of $\rho(t)$. The calculations are mere shifts, either to the left or right, depending on the sign of $\rho(t)$. The selected vector out from the multiplexer is added with 1 and shifted one step to the right to perform the divison by 2.

The final stage of the new weight calculation computes

$$\mathbf{W}_z(t+1) = \mathrm{clip}\left(\mathbf{W}_z(t) + \mathbf{temp}_2, \{\omega_{\min}, \omega_{\max}\}\right).$$

6.4.9 Residual mapping

The residual mapping is computed in two pipeline stages. The first stage computes $\Delta_z(t)$ and $\theta_z(t)$ as defined in (5.9) and (5.11), respectively. The second stage computes the residual mapping $\delta_z(t)$ as defined in (5.10), using a multiplexer to select between the different cases.

One of the cases in (5.10) is where the inequality $0 \leq (-1)^{\tilde{s}_z(t)}\Delta_z(t) \leq \theta_z(t)$ holds. The expression $(-1)^{\tilde{s}_z(t)}$ is equivalent to 1 when $\tilde{s}_z(t)$ is even, and -1 when $\tilde{s}_z(t)$ is odd. Using this, the inequality can be re-stated as

$$\tilde{s}_z(t) \text{ is even and } \Delta_z \geq 0 \text{ or } \tilde{s}_z(t) \text{ is odd and } \Delta_z(t) \leq 0.$$

This is easily implemented in hardware, as checking for odd/evenness is determined by whether the LSB is 0 or 1.

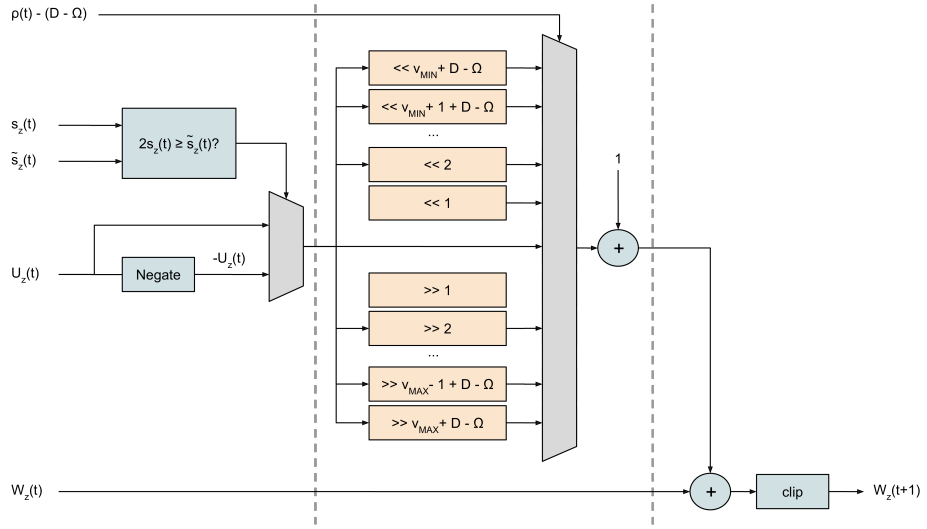

Figure 6.11: Weight update implementation

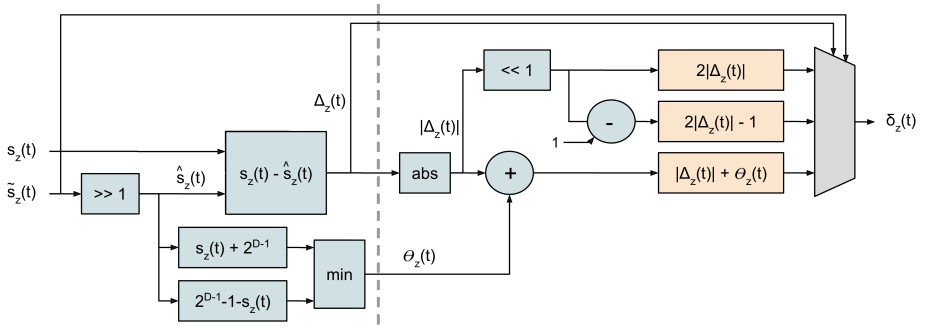

Figure 6.12: Residual mapping implementation

6.4.10 Encoding

An overview of the sample adaptive encoder implementation is shown in Figure 6.13. As shown, the encoder is implemented as a five stage pipeline.

The first and second stages compute the right hand side (rhs) used in the inequalities shown in (5.17) that determine the value of k_z. In the third stage, each of these inequalities are evaluated in parallel.

In the fourth stage k_z and u_z are chosen based on the results from each inequality evaluation. A priority encoder is used in combination with a multiplexer to do this selection, such that the highest integer i where the inequality with left hand side $\Gamma(t)2^i$ holds, is chosen as the value for k_z, and $\delta_z(t)/2^i$ is chosen as the value for u_z. In addition, a truncated version of $\delta_z(t)$ with only the k_z least significant bits is created. The bits are right-shifted so that the most significant bits are taken from the truncated $\delta_z(t)$ and the rest is filled with zeros.

The fifth stage of the computation puts together the code word based on the rules detailed in ¡some section¿, and computes the number of bits to use.

6.4.11 Bit packing

The bit packing module collects variable-length encoded words into packets of a given, configurable size N. The packer's operation centers around two registers of the output size N. The registers alternate between being the *current* and *next*. Incoming words from the encoder are stored in the current register. When the current register is full, any left over bits are put in the next register, and the current register's data is forwarded to the output. In the following cycle, the registers switch roles such that the next register becomes the current register, and the current register becomes the next register.

Due to the variable length of the incoming code words, the bit position in the current register where an incoming word should be stored can be any value ranging from the most significant bit $N-1$ to the least significant bit 0. It is therefore necessary to create N different candidates for the new value of the register where for each $i \in [0, N-1]$, the candidate for the new value consists of the i most significant bits of the current register followed by the M bit padded input word. To select the correct candidate, a *write pointer* is used to keep track of where the first non-occupied bit position is in the current register.

Figure 6.14 illustrates an example of how the packing is performed. In the example, the left-most register starts out as the current register, and the incoming words in the first 4 cycles are put into the current register. The fifth word is larger than the remaining space in the current register, so the leftover bits are stored as the most significant bits of the next register. In the next clock cycle, the current register, which is now filled up, is written to the output FIFO, and the next and current registers swap roles. The next input words are written to what is now the current register, until we get the same situation again in cycle 8.

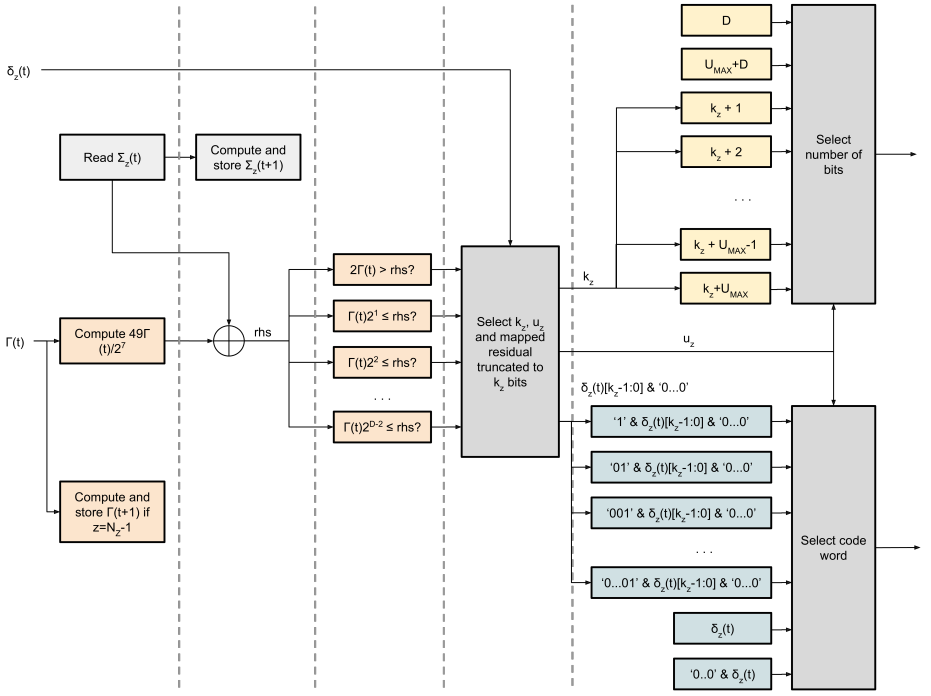

Figure 6.13: Overview of sample adaptive encoder. The dashed lines indicate the divide between different pipeline stages

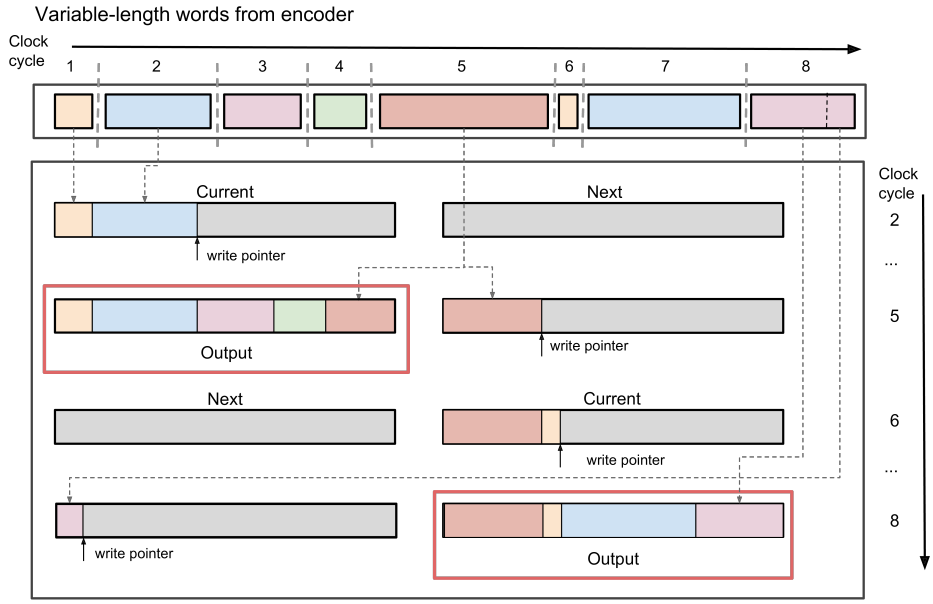

Figure 6.14: Illustration of the packing of variable length code words into fixed-size packets

6.4.12 AXI Stream interfacing

When part of a complete system, the output stream might need to be stalled, for example when the DMA core is waiting for write access to the memory. Stopping the output stream is not supported by the CCSDS123 core itself. Only data and valid signals are used to propagate data through the pipelines internally in the core. This choice was made to simplify the implementation by avoiding valid/ready handshaking at every pipeline stage within the core. Because the CCSDS123 core itself cannot handle output stalls, it is necessary to buffer the output from the CCSDS123 core in a FIFO, so that pending data is not lost when the AXI Stream slave that is receiving the compressed data is not ready to receive.

Figure 6.15 shows the top level AXI Stream interfacing with the core, with the output FIFO. If the output is stalled for a long period of time, the FIFO might become full and data will be lost. To avoid this situation, the number of data words written to the FIFO, which is available as an output signal from the FIFO, is monitored. Whenever this count grows larger than a certain limit, the **tready** signal in the input AXI Stream interface is de-asserted. No new data will enter into the CCSDS123 core, but all data present in all pipeline stages of the core at that point in time will still come out in the following cycles. This means that the FIFO capacity limit at which **tready** is de-asserted must be low enough such that the FIFO has room for the additional data words that will come out

of the CCSDS123 core. The limit is given by the worst-case number of packed words that can come out of the core, which is when each of the N_{stages} pipeline stages has valid data, and each data word out from the encoder has the maximum length of $U_{\max} + D$:

$$\text{limit} = \text{FIFO capcity} - \left\lceil \frac{N_{\text{stages}}(U_{\max} + D)}{\text{packed block size}} \right\rceil .$$

The size of the FIFO can be varied, as long as it is larger than this limit. Choosing the size is a trade-off between area usage and how often the input of the core is stalled when the output is stalled.

6.5 Parallel implementation

This section will detail a parallel implementation of CCSDS123. First some challenges related to distribution of samples will be introduced, followed by helpful terminology that will make descriptions of the implementation clearer.

6.5.1 Streaming of samples in parallel

If N_p samples are streamed in parallel, N_p instances of each computational module are needed to compress each sample simultaneously. The chain of computation modules, starting at the local sum and difference calculations and ending at the sample adaptive encoding, will be referred to as a *pipeline*.

When streamed in BIP order with N_p samples in parallel, the incoming data words can be thought of as consisting of N_p *lanes* which can be numbered from 0 to $N_p - 1$, where lane 0 are the D least significant bits, lane 1 are the D next least significant bits, and so on. Figures 6.16 and 6.17 show how samples are placed in lanes for the first 10 beats of a transfer, when N_p is 4 and N_z is 8 and 9, respectively. The first sample in each pixel is highlighted.

If it is assumed that N_z is divisible by N_p, i.e. $N_z \bmod N_p = 0$, which is the case in Figure 6.16, then each lane will always contain a fixed subset of bands in each pixel. For

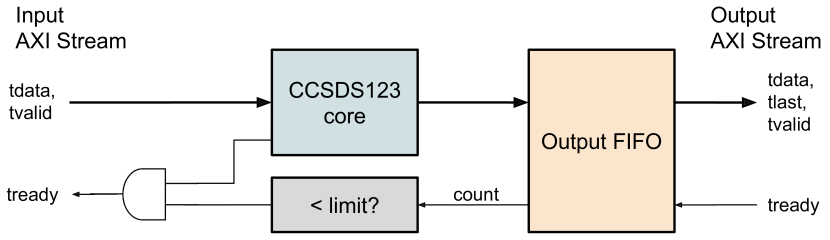

Figure 6.15: The top level diagram for the CCSDS123 IP module

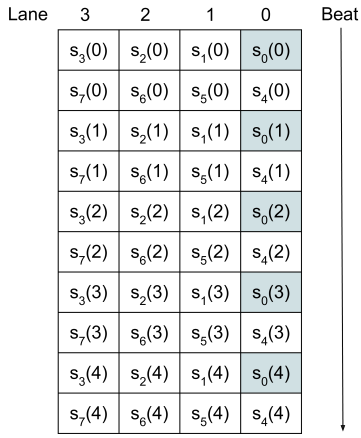

Figure 6.16: Sample placement in lanes when $N_z = 8$ and $N_p = 4$

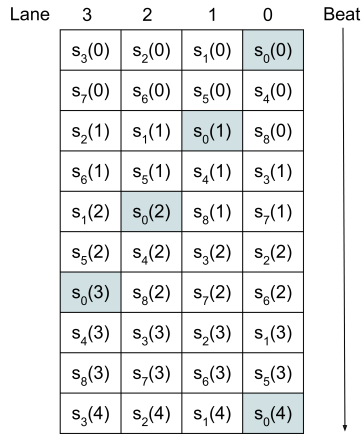

Figure 6.17: Sample placement in lanes when $N_z = 9$ and $N_p = 4$

instance, lane $n = 0$ contains samples $s_0(t)$ and $s_4(t)$ in bands $z = 0$ and $z = 4$, whereas lane $n = 1$ always contains $s_1(t)$ and $s_5(t)$. More generally, $s_z(t)$ is always streamed in lane $n = z \bmod N_z$. This means that the serial pipeline implementation can be used almost as-is, with one instance for each lane. All FIFO depths, RAM sizes and so that are dependent on z, are replaced by z/N_z. During development, this kind of parallel implementation was the first step.

Things become more complicated when N_z is not divisible by N_p. When this is the case, bands are no longer confined to a specific lane, but ”shift” between lanes as Figure 6.5.3 shows: $s_0(0)$ is in lane 0, but $s_0(1)$ is in lane 1, and so on. Two options were considered in this case.

One option is to stall the input stream when the last sample in a pixel has been received, and realign the stream such that in the next beat the first sample of the next pixel is in lane 0. This would allow each pipeline to process a fixed subset of samples similarly to when N_z is divisible by N_p, but it would reduce throughput (since the input stream is stalled once per pixel) and require extra logic to perform input stalling and realigning.

The other option is to not stall the input and instead let the set of bands handled by each pipeline overlap. This means that the pipelines must share information besides the local differences between them. Using Figure 6.17 as an example, pipeline 0 will produce $\mathbf{W}_0(1)$ when processing $s_0(0)$, which is needed by pipeline 1 when processing $s_0(1)$. The same applies to accumulators used in the encoder. Sample delay is also more complicated, since a sample that arrived in lane 0 might for example be used as the neighbor of a sample that arrived in lane 2. The upside is that throughput would be maximized under this approach.

6.5.2 Overview of architecture

An overview of the parallelized implementation is shown in Figure 6.18. The weight store, central local difference store and accumulator storage is shared between the pipelines. On overview of each pipeline is shown in Figure 6.19. The modules inside each pipeline remain largely unchanged from their descriptions in Section 6.4. The only changes that have been made are related to routing of data.

6.5.3 Definitions and terms

Before describing the building blocks of the parallel implementation, some helpful terms and notations will be defined. When N_z is not divisible by N_p, it was observed that samples from a particular band shift between different lanes. Therefore, the samples from the same band in previous or future pixels are not necessarily in the same lane as the current sample. It is therefore necessary to be able to find which lane a given sample will occur in.

The stream of N_p samples per beat can be thought of as being in a grid with N_p columns numbered from 0 to $N_p - 1$. If a certain sample is in lane i, that is in column i of this imagined grid, then the sample n samples later is in lane $(i+n) \bmod N_p$, where the modulo operation accounts for the wrap-around that occurs when $i = N_p - 1$. In particular, we are interested in the case where $n = N_z$, i.e. the distance between samples from different pixels in the same band. In this case, we can define

$$shift(i, \Delta t) = (i + N_z \Delta t) \bmod N_p$$

as the lane which the sample $s_z(t + \Delta t)$ is in, given that $s_z(t)$ is in lane i. Referring to Figure 6.17, we have for example that sample $s_3(0)$ is in lane 3, and $shift(3,2) = 0$.

Another observation we make from Figure 6.17 is the fact that the number of beats *between* samples in the same band is not constant. For instance, $s_0(1)$ arrives two beats after $s_0(0)$, but $s_0(4)$ arrives three beats after $s_0(3)$. For the sample delaying which produces the neighboring samples this becomes important, since the current and neighboring samples must arrive in sync. Again thinking of the samples as placed in a grid, finding the number of beats between two samples is the same as finding how many rows there are between them. This is influenced by which lanes the samples are in. For instance, if $s_z(t)$ is in lane 1, then $s_{z+1}(t)$ is in lane 2 of the same row, but if $s_z(t)$ is in lane 3 (and we assume $N_p = 4$), then $s_{z+1}(t)$ would be in lane 0 of the next row. Finding how many row boundaries are crossed can be found by adding the initial lane number to the number of samples to skip, and doing integer division of this sum by N_p. Letting the number of samples to skip be N_z, we can define

$$delay(i, \Delta t) = \left\lfloor \frac{i + N_z \Delta t}{N_p} \right\rfloor$$

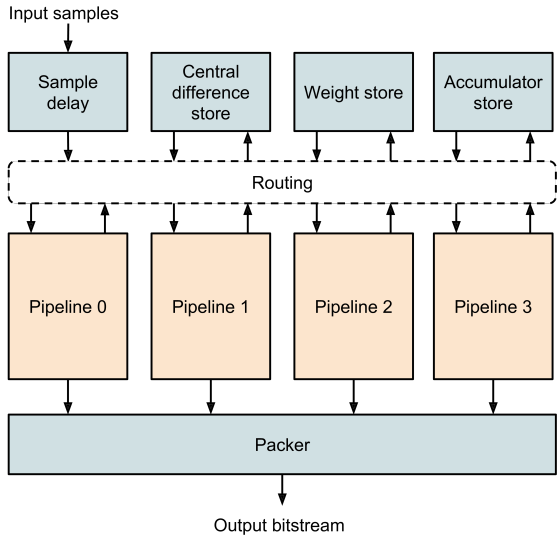

Figure 6.18: Overview of the parallel CCSDS123 implementation when number of pipelines is 4

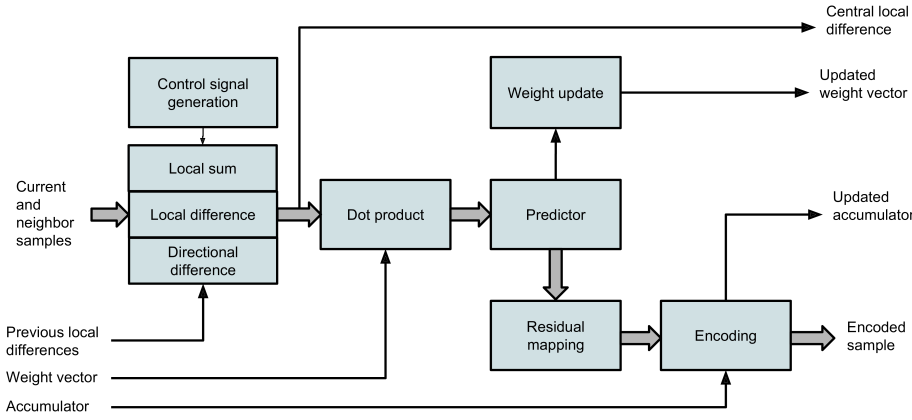

Figure 6.19: Overview of pipeline architecture

as the number of beats from $s_z(t)$ arrives until $s_z(t+\Delta t)$ arrives, given that $s_z(t)$ is in lane i. As an example using Figure 6.17, we have that $s_0(0)$ is in lane 0, and $delay(0,1) = 2$, and that $s_0(3)$ is in lane 3 and $delay(3,1) = 3$.

6.5.4 Sample delay

The sample delay implementation is shown in Figure 6.20. The basic structure is similar to the serial implementation, with four different FIFOs to produce the W, NE, N and NW delayed samples. For each lane i there is a set of such FIFOs, with the depth of the FIFOs given by the *delay* function to get the appropriate delay such that the neighboring sample is available at the same time as the corresponding input sample.

Due to the shifting described previously, the samples coming out of the FIFOs must be shifted according to the *shift* function, so that the delayed sample is used by the same pipeline that is processing the corresponding input sample. Figures 6.21 and 6.22 show examples of how this works when $N_p = 4$ and $N_z = 9$ and $N_z = 11$, respectively. The rows of samples highlighted show the incoming samples at a given beat, and the output from the W delay stage in the same beat. In each lane in the incoming stream we can observe $s_z(t)$ and in the same lane from the delay stage, the W neighbor $s_z(t-1)$.

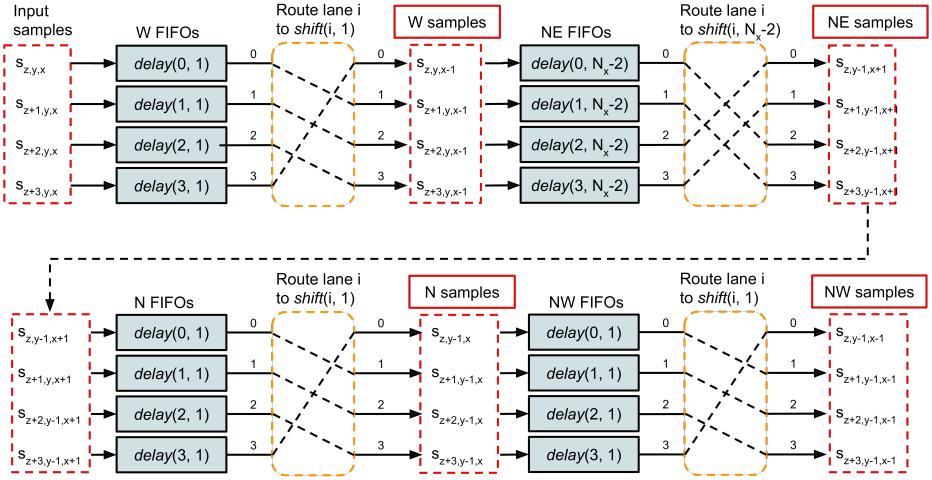

Figure 6.20: Sample delay in parallel CCSDS123 implementation. The actual routing might be different than shown, depending on the value of N_z and the number of pixels Δt.

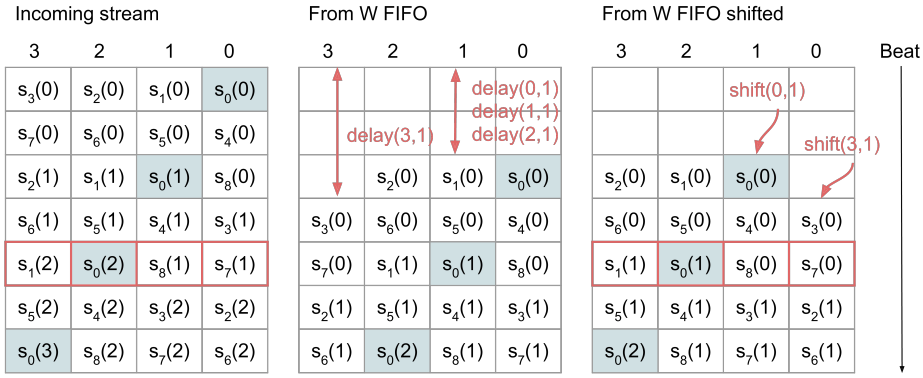

Figure 6.21: Example of sample delay to obtain W neighbor samples when $N_p = 4$ and $N_z = 9$

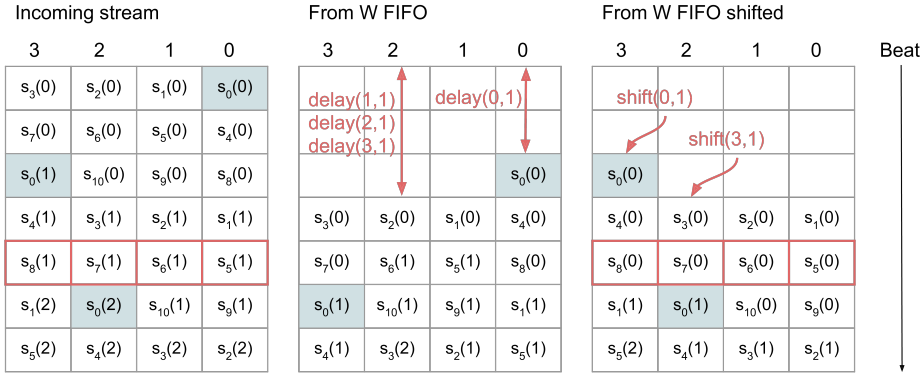

Figure 6.22: Example of sample delay to obtain W neighbor samples when $N_p = 4$ and $N_z = 11$

6.5.5 Local differences

In the serial implementation, central local differences are stored in the local difference store for use in prediction of subsequent samples. In the parallel implementation there is still a need for a local difference store (as long as $P > 0$), but it is also necessary for the pipelines to share local differences with pipelines that have a higher index. For instance, if pipeline 3 is processing $s_{z+3}(t)$, then the local difference $d_{z+2}(t)$ is calculated by pipeline 2 and can be taken directly from there. However, the local difference $d_{z-1}(t)$ was processed in the cycle before and therefore needs to be stored. Figure 6.23 shows how the local difference vectors for each pipeline are assembled from local differences from lower indexed pipelines and from the local difference store.

When $z < P$, only the z previous local differences should be used. In the serial implementation this was solved by zeroing the contents of the local difference store each time $z = N_z - 1$, such that no local remaining differences from the previous pixel was used. The same approach does not work for the parallel implementation, since previous local differences are coming directly from other pipelines. If, for instance, pipeline 2 is handling $s_{N_z-1}(t)$ and pipeline 3 is handling $s_0(t+1)$, then the local difference from pipeline 2 should *not* be used by pipeline 3. This was solved by having each pipeline *mask* the incoming previous local differences based on the z coordinate of the sample it is processing, such that if $z < P$ then the local differences with index $i \leq z$ are included in the local difference vector, while the positions with index $i > z$ are set to zero.

6.5.6 Weight and accumulator storage

Weights and accumulators are stored in the exact same way, so two instances of the same module, called *shared store* is used for both of them. The shared store is implemented in much the same way as the sample delay, but an important difference is that while the sample delay consists of rigid fixed-length FIFOs where the least recent element is pushed out simultaneously with the current samples coming in, the shared store must handle variable distances between read and write indices.

Figure 6.24 shows an overview of the shared store implementation. There are N_p block RAMs, labeled bank 0 to $N_p - 1$, one for each lane. Each bank has the same depth, $M = \lceil N_z/N_p \rceil$. To keep track of where to read and write from, a *read counter* and a *write counter* is used, which the read and write addresses in each bank are derived from. The counters are initialized such that the distance between their values is $delay(0,1)$, and are subsequently incremented each time read enable or write enable is high. The write counter is used directly as the write address in each bank, while the read address for each bank i is calculated as follows:

$$\text{read address}(i) = \begin{cases} \text{read counter}, & i + N_z \bmod N_p < N_p \\ (\text{read counter} - 1) \bmod M, & i + N_z \bmod N_p \geq N_p \end{cases}$$

This makes the initial distance between the initial read address and write address for each bank equal to $delay(i,1)$

Figure 6.25 shows the state of the weight store when $N_z = 61$ and $N_p = 4$, at two different points in time. The figure to the left shows the state right after reset, before any processing has started. The read counter is initialized to 0, while the write counter is initialized to $delay(0,1) = 15$. For lanes 0 through 2, the read addresses are equal to the counter value, which is 0, whereas for lane 3, the read address is one less, which wraps around to 15. Hence, for lane 3, both the read address and the write address are the same.

As samples start arriving at the input of the core, the read enable of the weight store will be high, and the read counter will start incrementing. The data that is read at this point in time from the weight store is not actually used, since for the first pixel there is no prediction performed. N cycles after the first set of samples arrived at the input of

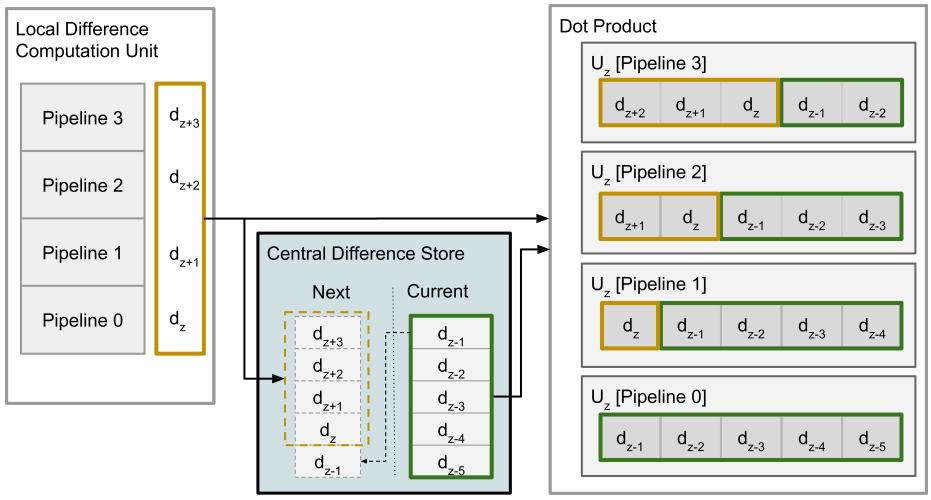

Figure 6.23: Routing of central differences between pipelines when $N_p = 4$ and $P = 5$

the core, the updated weights for pixel 1 will start being produced by the weight update module. From that point on, the write enable is high, and weights are written into each lane in the weight store at the address pointed to by the weight counter. The figure to the right shows the state of the shared store at this time, where the first weights for the next pixel are being written to memory when the read counter has moved N steps from the initial position.

Figure 6.26 shows the continuation of this example. To the left the state of the weight store is shown a few clock cycles later, when the read pointer has come to 15, and the first weights for the next pixel are being read simultaneously with samples $s_{60}(0)$, $s_0(1)$, $s_1(1)$ and $s_2(1)$ arriving at the input. To the right the state is shown 15 cycles later when the next pixel is arriving at the input.

An optional read delay is also available for cases where read data should be output a given number of clock cycles after the read enable signal is asserted. The read delay is set by a generic parameter and the read enable signal will be delayed by the given number of clock cycles before triggering a read in the bank memories.

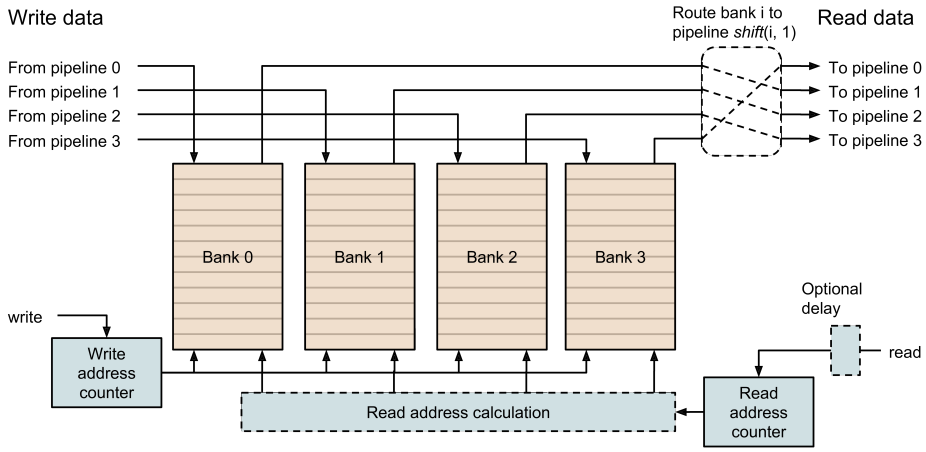

Figure 6.24: Implementation of the shared store module

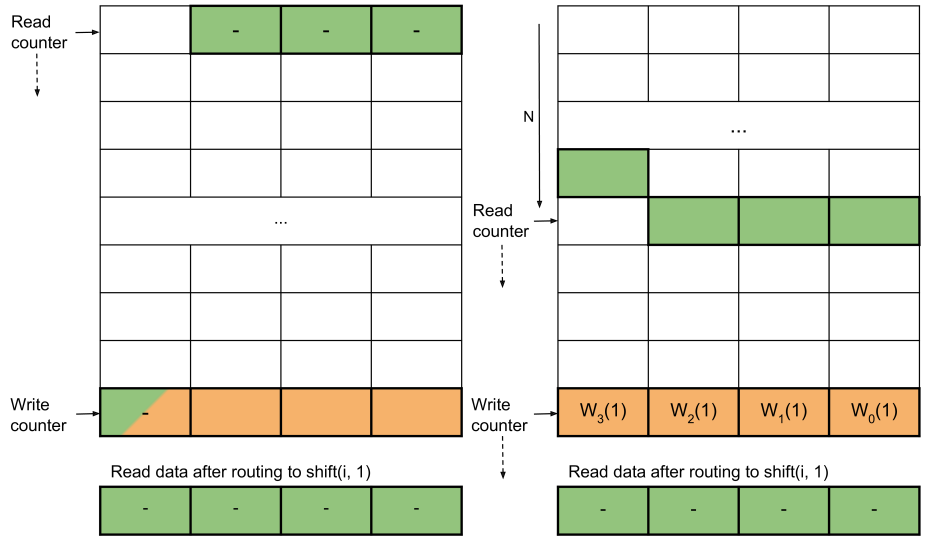

Figure 6.25: State of the shared store when used as weight store, when $N_z = 61$ and $N_p = 4$. The figure to the left shows the initial state after reset, while the right figure shows the state when the last samples of pixel 0 and the first samples of pixel 1 are arriving

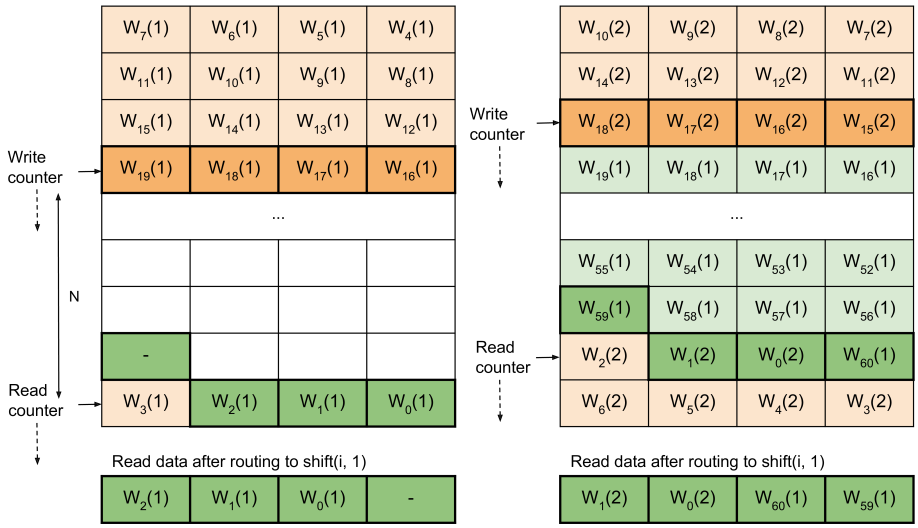

Figure 6.26: Implementation of the shared store module

6.5.7 Packing of variable length words

The last stage of the parallel implementation data flow is packing the variable-length encoded words from each pipeline into fixed-size blocks and outputting these blocks on the output stream interface.

Packing of the N_p encoded variable-length words from the pipelines is performed by right-shifting each input word by the length of the previous words and combining (or-ing) it with the previous results. If the words from the pipelines are labelled W_0, ..., W_{N_p-1} with lengths L_0, ... L_{N_p-1}, then W_0 must be shifted by the number of bits leftover from the previous cycle, L_{prev} and or-ed with the bits remaining from the last cycle, W_{prev}, to get $W_{\text{prev}}W_0$. W_1 must be shifted by $L_{\text{prev}}+L_0$ and or-ed with W_0 into $W_{\text{prev}}W_0W_1$. W_2 must be shifted by $L_{\text{prev}+}L_0+L_1$ and or-ed with $W_{\text{prev}}W_0W_1$ to get $W_{\text{prev}}W_0W_1W_2$ and so on. A problem with this approach is that as the number of possible shifts required for word W_i grows with N_p and with the maximum length $U_{\max}+D$ of each word.

It is possible to take advantage of the fact that the output is going to be in fixed-size blocks of size B. Instead of shifting each word as described, combining them and then splitting up into blocks, blocks can be extracted as soon as they are filled up. Each time the sum of previous words' lengths exceeds B, a block will then have been extracted, and the next word can be shifted relative only to the number of bits left over after that block has been extracted.

If we define s_i as how many bits input word W_i must be right shifted, we can state this as

$$s_i = \Sigma L_{i-1} \bmod B,$$

where

$$\Sigma L_i = L_{\text{prev}} + \sum_{j=0}^{i-1} L_j.$$

Each word needs to be shifted by at most $B-1$ bits regardless of N_p or maximum word length.

Figure 6.29 shows an example of how this packing works for $N_p = 4$.

The packer is implemented as a three stage pipeline which combines words and packs them into blocks and pushes them to a FIFO, followed by logic that fetches blocks from the FIFO and shifts them out when the output stream is ready. An overview is shown in Figure 6.27.

6.5.7.1 Packing implementation

The first stage in the pipeline computes s_i and an *extraction count* e_i for each input word W_i that indicates how many blocks can be extracted when that word W_i is added. The extraction count is computed as follows:

$$e_i = \left\lfloor \frac{\Sigma L_i}{B} \right\rfloor.$$

It should be noted that the extraction count can only be larger than 1 if $U_{\max} + D > B$. For cases where the block size is larger than the maximum output size, the extraction count can be thought of as a flag indicating whether a block has been filled up when W_i is added or not.

The second and third stages form what will be termed a *combiner chain*, and are shown in Figure 6.28. The combiner chain performs the combining of input words, using the calculated shift amounts s_i and extraction counts e_i. In addition it receives the remaining L_{prev} bits from the previous cycle.

Shifting of each word is performed in parallel, using s_i to select the desired shifted version of W_i from a multiplexer. The last pipeline stage performs the combining of shifted words and the extraction of full blocks. This is performed in a chain of or-operations and extraction operations associated with each shifted input word. At each step in this chain, the shifted input word is or-ed with the remaining bits from the previous step. If the extraction count e_i is non-zero, adding the word W_i has accumulated enough bits to fill one or more blocks. If $e = 0$, there is not yet enough bits to extract a full block, so the accumulated bits are just passed on to the next step.

New full blocks produced at any step are appended to the filled blocks from earlier steps in the chain and forwarded to the next step together with the new count of full blocks. The remaining bits after extraction are forwarded to the next step. The remaining bits from the last step in the chain are connected to a register such that they can be combined with the words arriving in the next cycle.

In the case where the last flag is set, the remaining bits are output as a separate block if the number of remaining bits is non-zero.

6.5.7.2 Output buffering and interfacing

The output from the last step in the combiner chain is the filled blocks and a count of how many there are. This information and the last flag is pushed into a FIFO. The FIFO is necessary to handle the limited output bus bandwidth when N_p and the length of the incoming words is larger than the bus width. Even though the average number of bits output per cycle is lower than the bus width, there might be times during the compression where this is not the case. Hence it is necessary to buffer the incoming full blocks so that they can be output sequentially one at a time.

The data word width of the FIFO is given by the maximum number of blocks that can be extracted in one clock cycle. This value is given by

$$\text{max number of blocks} = \left\lfloor \frac{B-1+N_p(U_{\max}+D)}{B} \right\rfloor + 1,$$

where $B-1$ in the numerator is the maximum number of leftover bits from the previous cycle, $N_p(U_{\max}+D)$ is the number of input word bits in the worst case where all have maximum length, and 1 is added to account for the extra block that can be included when the last flag is set.

The output logic fetches the set of full blocks, the number of full blocks and the last flag from the FIFO. In order to sequentially output the blocks from the set of blocks a counter is used, starting at 0 and counting up each time a block is handshaked on the output. The counter is used as an index into the set of blocks. At the last block the stream's **tlast** signal is asserted if the last flag from the FIFO is set. When the counter reaches the number of blocks, a new word is read from the FIFO provided that it is non-empty.

At times when the average bitrate of the encoded samples is higher than the output bus width, the FIFO might risk becoming full, similarly to the situation described in Section 6.4.12. It is therefore necessary to stall the input stream of the CCSDS123 core before the FIFO might overflow. The FIFO module used has the ability to set a threshold on the number of data words in the FIFO, and will output a signal when this threshold is reached. This signal is used to de-assert **tready** at the input stream when this is the case. Similarly to the case in Section 6.4.12, the threshold must be set such that there is room left in the FIFO for all the packed encoded samples arriving from within the core when the input stream has been stalled. This number is equal to the total number of pipeline stages from the input of the core and to the FIFO, which is given by

$$N_{\text{stages}} = 3+S+2+2+5+3 = S+15,$$

where S is the number of pipeline stages in the dot product and the other numbers are the number of pipeline stages in local difference calculation, prediction, residual mapping, encoding and packing.

The depth of the FIFO must be at least as large as N_{stages}, but should be larger to avoid stalling the input stream in brief moments where encoded bit rate is high or the output stream is stalling. The FIFO depth is ultimately a trade-off between performance and block RAM utilization.

In the case of on the fly processing it is not possible to stall the input stream. Hence, it is important for the FIFO to be large enough to avoid any overflow situation. The necessary FIFO depth to avoid overflow is dependant on the expected image statistics and how fast the predictor can adapt, and is therefore only possible to determine through testing on actual images.

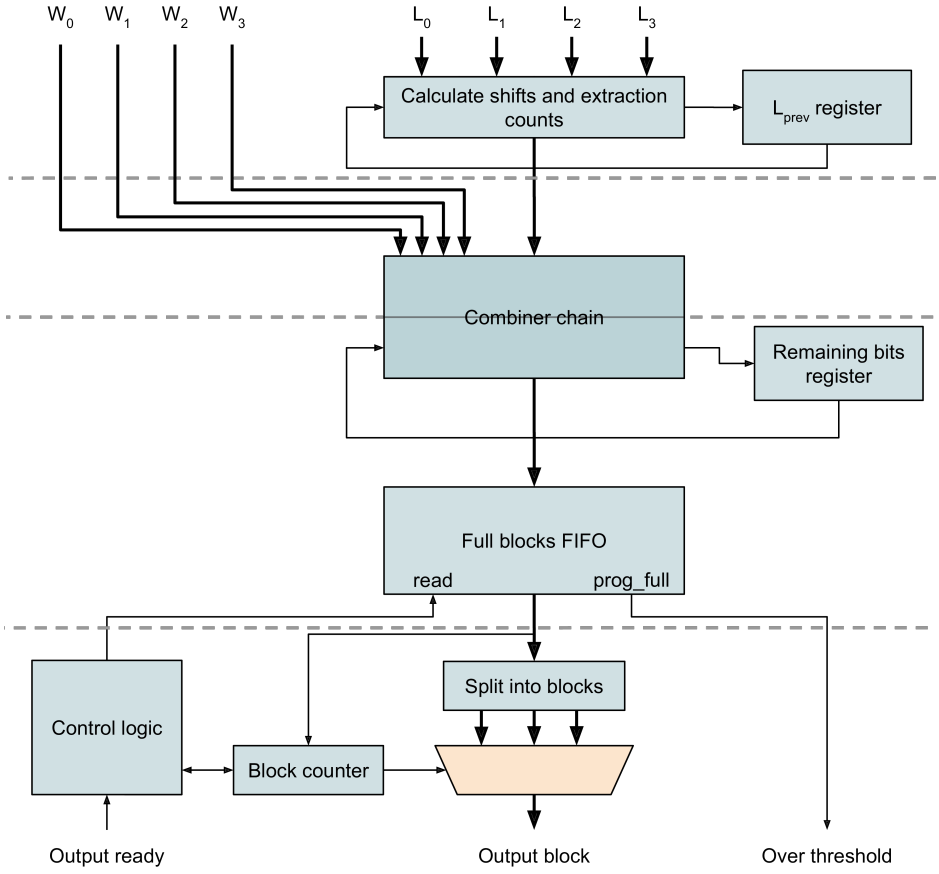

Figure 6.27: Implementation of variable length word packer

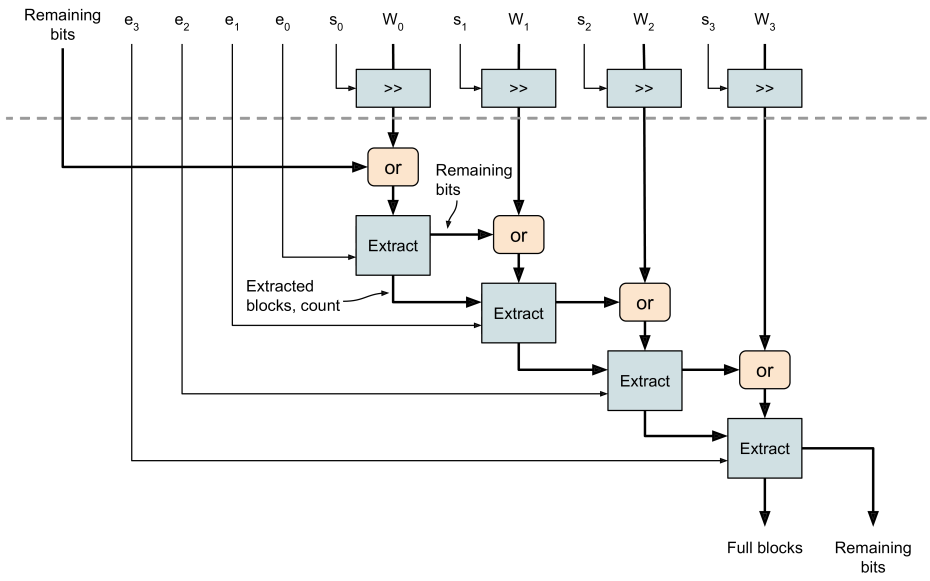

Figure 6.28: Implementation of combiner chain used in packing

6.5.8 Improved packer

6.5.8.1 Overview

The packing scheme described in the previous section combines each incoming word serially in combinatorial logic. This is doable for $N_p < 6$, but for larger N_p the critical path has been shown to become too large to meet a timing constraint of 10ns (100 MHz). The critical path is the summation of lengths L_i used to determine the shift of each word and the extraction count.

To make it possible to have larger N_p, an improved version was designed, shown in Figure 6.30. The new version does not attempt to combine all the N_p incoming words in one large combiner chain, but distributes them across several combiner chains operating in parallel. The number of combiner chains is determined by a generic parameter setting the number of words per combiner chain, $1 \leq N_{\text{per chain}} \leq N_p$. Selecting $N_{\text{per chain}}$ has several tradeoffs which will be discussed further on. Given N_p and $N_{\text{per chain}}$, the number of combiner chains is $N_c = \lceil N_p / N_{\text{per chain}} \rceil$, and the chains can be numbered from 0 to $N_c - 1$, where the chain with the lowest index handles the $N_{\text{per chain}}$ words with the lowest indices, and so on.

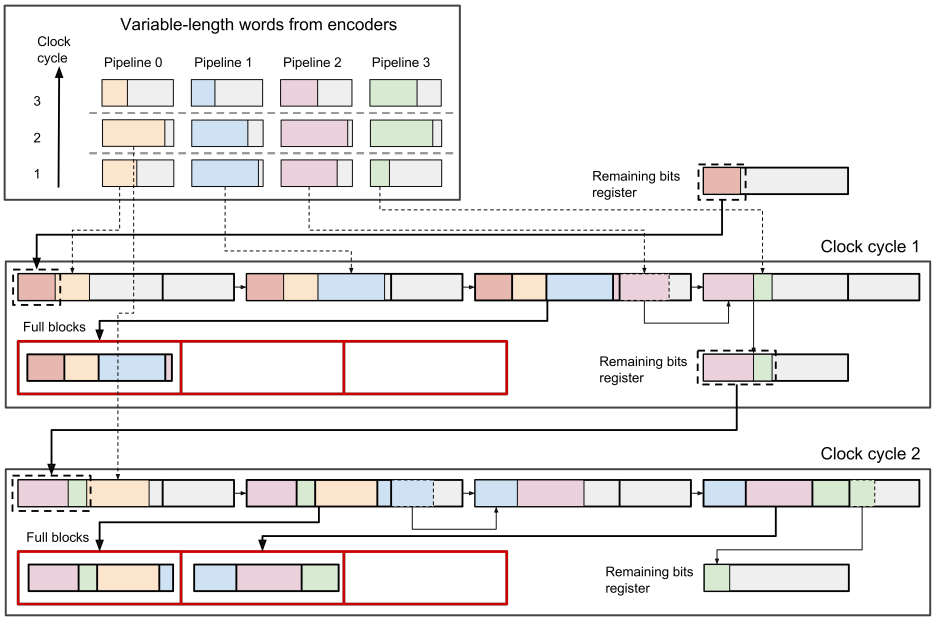

Figure 6.29: Operation of the variable length word packer

6.5.8.2 Shifts, extraction counts and combiner chains

To eliminate the critical path in the previous packer implementation, the shift amounts s_i and extraction counts e_i are computed sequentially for each combiner chain, using N_c clock cycles, as shown in Figure 6.30. Since this takes more than one clock cycle, L_{prev} from the previous set of words is not available when computation starts. It is therefore not possible to compute s_i and e_i directly. What is done instead, is to compute the sums

$$\Sigma\bar{L}_i = \sum_{j=0}^{i-1} L_j = \Sigma L_i - L_{\text{prev}},$$

first, and then in the last cycle computing $\Sigma L_i = \Sigma\bar{L}_i + L_{\text{prev}}$ and then computing s_i, e_i for use in the combiner chains, and the left over number of bits, L_{prev} for the next set of words.

When several combiner chains are working in parallel, the remaining bits out from chain i cannot be input into chain $i+1$, since this would create a large critical path from the start of combiner chain 0 all the way through combiner chain $N_c - 1$. One way this could be solved, is to apply the same kind of sequential delay as was done for the length calculations, displacing each combiner chain by one clock cycle from the previous. This would however require large delay registers for the left-most chains to keep the full blocks from each chain synchronized with the last chain $i = N_c - 1$, and it would require

larger FIFOs to prevent overflows due to the extra number of total pipeline stages in the circuit.

The solution that was developed, was to let each combiner chain shift its input words to make room for the remaining bits from the previous chain, but not combine them with the remaining bits from the preceding chain. Instead, the gaps are filled with remaining bits at the output of each combiner chain. This can be done very efficiently by logically or-ing the first block from each combiner chain with the remaining bits from the previous chain. When a chain has produced no blocks, the remaining bits from the chain are or-ed with the remaining bits from the previous chain and used as the remaining bits to join with the first block of the next chain, and so on.

Each combiner chain outputs a *block set* containing the blocks that were produced and a count of how many there are. Each combiner chain has a dedicated block set FIFO where this information is pushed when the chain has produced at least one block. A separate *control FIFO* is used to keep track of which block set FIFOs contain valid data. When one or more block sets is pushed to one of the block FIFOs, a new word is pushed to the control FIFO with a *block set mask* as well as the last flag that indicates end of compression. The block set mask is a bit string where each bit corresponds to one of the combiner chains. For instance, a block set mask '101' means that there are valid block sets from combiner chains 0 and 2. Figure 6.31 shows an example of how this works when $N_c = 3$.

An alternative option could be to have all the combiner chains share a common FIFO. This solution would reduce some of the overhead related to having separate FIFOs (block RAM instances and control logic), but it would also lead to less effective memory utilization since it would require the maximum number of bits to be stored even if only a small number of blocks contain valid data, as Figure 6.32 illustrates.

The block output logic detailed in Section 6.5.7.2 is also expanded in the new packer. Since there are now several block sets if $N_c > 1$, the output logic must go through each valid block set and output each valid block within each set. The output logic fetches the block set mask from the control FIFO and reads from the block set FIFOs corresponding to the block set mask. The block set corresponding to the left-most bit in the block mask is loaded into a shift register and during the subsequent cycles, each block in the set is output until the given number of valid blocks in the set is reached. The next block set (if any) is loaded into the shift register and the same procedure is repeated until all block sets have been output, in which case a new control word is read from the control FIFO.

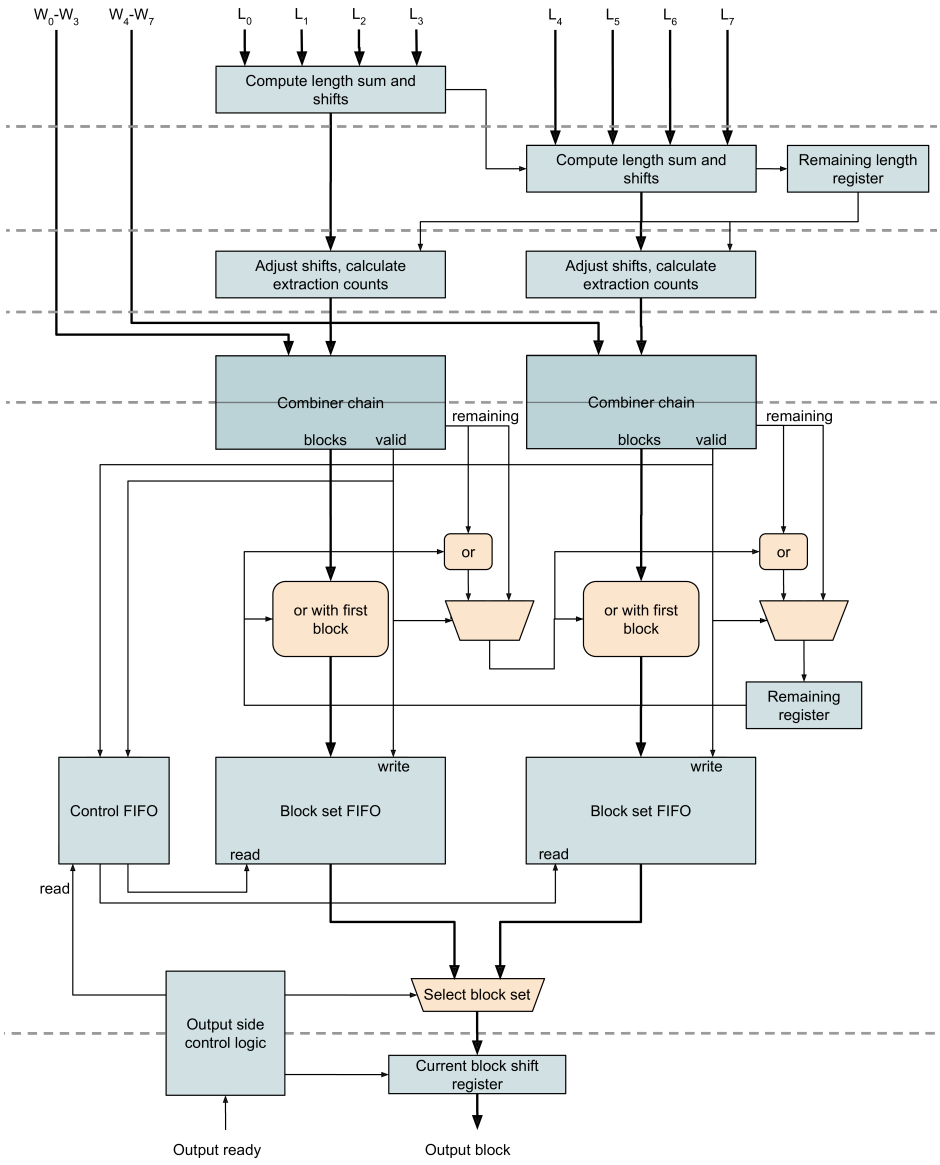

Figure 6.30: Implementation of improved variable length word packer

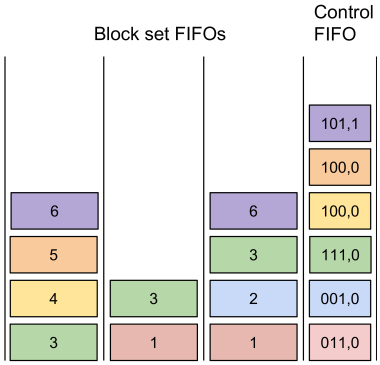

Figure 6.31: Memory utilization when using separate block set FIFOs

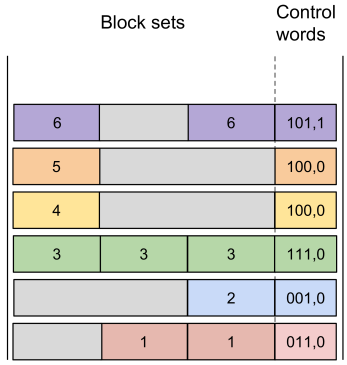

Figure 6.32: Memory utilization when using one block set FIFO

7

CCSDS123 implementation results

This chapter presents the results gathered for the CCSDS123 implementations. It will first present an issue that was discovered when gathering results, followed by resource utilization, timing and power consumption results with analysis and discussion. Finally, some comparisons with previous implementations will be presented.

7.1 Compression analysis

The Emporda software compressor has been used to run CCSDS123 on a selection of hyperspectral images to evaluate compression performance and compare it to results on the same images using JPEG 2000. Images from two different hyperpsectral imagers, HICO and AVIRIS were used, both calibrated images (L1) and atmospherically correct versions of the images (L2). Hyperspectral Imager for the Coastal Ocean (HICO) created at Oregon State University is a spaceborne hyperspectral imaging spectrometer designed to sample the coastal ocean [21]. Its images should therefore be quite similar to the images taken by the SmallSat HSI payload, which is also focused on coastal and ocean imaging. HICO images have the size $512 \times 2000 \times 128$. The Airborne Visible / Infrared Imaging Spectrometer (AVIRIS) is another imager, developed by the NASA Jet Propulsion Laboratory. It is used on board aircraft flying at high altitudes. Images taken by AVIRIS have the size $614 \times 512 \times 224$, in other words smaller spatial dimensions than HICO, but almost twice the number of wavelengths or bands.

The results are shown in Figure 7.1. As the results show, the two perform very similarly. It should be noted that JPEG 2000 is used here in lossless mode. With loss, JPEG2000 can presumably achieve much better compression ratios. The two show a very similar compression ratio, with CCSDS123 slightly better for most images. For most of the L1

images, the compression ratio is about 3 (meaning that the compressed image has a size reduced by a factor of 3). Level 2 images do not achieve as good compression ratios. Interestingly, the drop in compression ratio is smaller for the AVIRIS image. Why there is such a drop for L2 images is not clear, but this is something that could be investigated if the CCSDS123 implementation should be applied to compression of L2 images.

Based on this analysis, it is clear that CCSDS123 performs at least as well, and sometimes better than lossless JPEG 2000. A big advantage with CCSDS123 is that it is designed to be implementable in hardware [12].

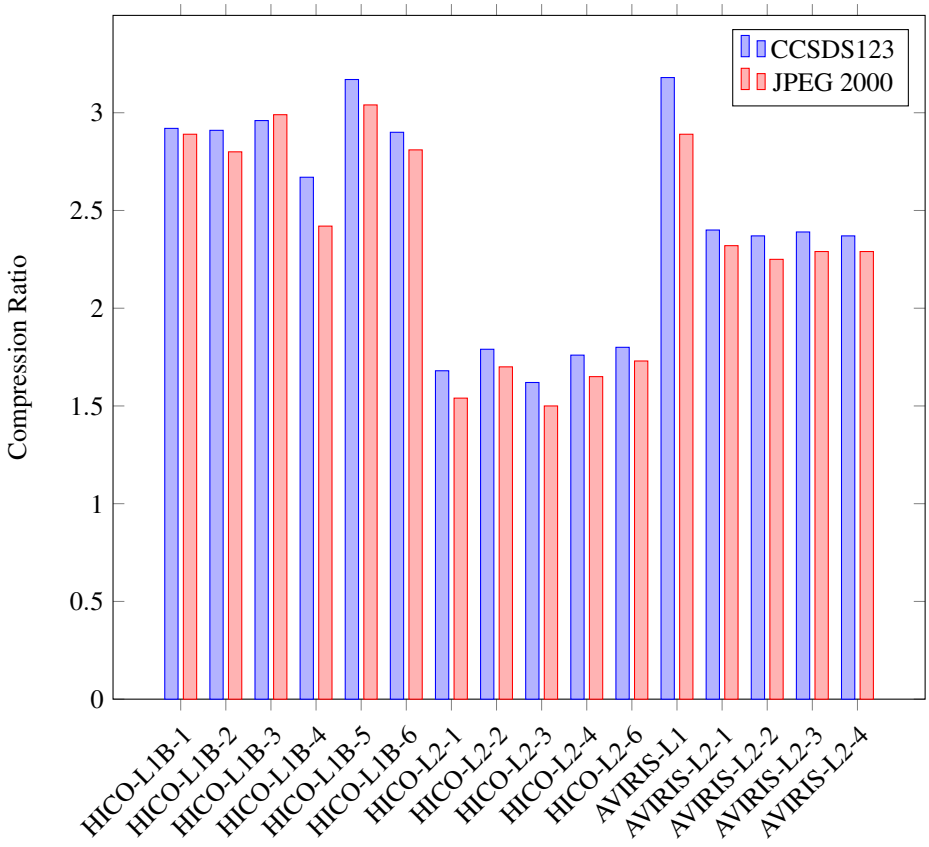

Figure 7.1: Compression performance for CCSDS123 and JPEG 2000

7.2 Implementation results

Table 7.1 shows the default values used for the CCSDS123 parameters when looking at utilization. The values are the same as used in the CCSDS Information Report on

CCSDS123 [12], as well as in the papers by Báscones et al, which makes it easier to compare results.

Parameter	Description	Value
Predictor parameters		
D	Sample bit resolution	16
P	Number of previous bands to use in prediction	3
Ω	Weight resolution (weight bit resolution is $\Omega+3$)	19
$v_{\min}$	Weight update scaling exponent initial parameter	-1
$v_{\max}$	Weight update scaling exponent final parameter	3
t_{inc}	Weight update scaling exponent change interval	2^6
R	Register size	64
	Prediction mode	Full
	Local sum mode	Neighbor
Encoder parameters		
$U_{\max}$	Unary length limit	18
γ_0	Initial count exponent	1
γ^*	Re-scaling counter size	6
K	Accumulator initialization constant	3
Image size		
N_x	Width	512
N_y	Height	2000
N_z	Bands	128
Core parameters		
$N_{\text{per chain}}$	Words per combiner chain	$\min(4, N_p)$
-	Packer FIFO depth	256

Table 7.1: Default CCSDS123 parameters used when analyzing utilization, power and performance

The default synthesis and implementation (place and route) settings in Xilinx Vivado have been used. There have been no special optimizations enabled. A clock constraint of 100 MHz has been used in all synthesis and implementation runs.

7.2.1 Resource utilization

7.2.1.1 Issue with sub-optimal block RAM utilization

When synthesizing the core for different number of pipelines it was discovered that the number of block RAMs used varied considerably even though it is theoretically supposed to remain constant. The two effects observed were (for the same set of parameters used in utilization analysis, shown in Table 7.1):

- Shared store block RAM utilization for weights increasing linearly with N_p

- Sample delay block RAM utilization varying from 32 to 48 depending on N_p

Both observations can be explained by looking closer at the block RAMs present in the Zynq 7-series FPGAs. The fundamental block RAM unit has a capacity of 36Kb and can be configured in a slew of different ways: $32K \times 1$ (meaning 32768 memory locations of 1 bit each), $16K \times 2$, $8K \times 4$, $4K \times 9$, $2K \times 18$, $1K \times 36$ and 512×72 [22]. The synthesis tool tries to map the RTL to a fitting set of such configurations.

In the case of the weight store (shared store), the issue is the very wide data words used. For $P = 3$ and $\Omega = 19$, the width is $C_z \cdot (\Omega + 3) = 132$ per pipeline. With e.g. $N_p = 4$ the width is already 528 bits. If $N_z < 512$, this means that block RAMs in the widest 512×72 configuration can be used, resulting in $132N_p/72$ block RAMs needed. If N_z is much less than 512, this means that these block RAMs will be under-utilized, since only a depth of N_z is needed. A better solution to using block RAMs for this purpose is to use LUT elements as RAM, which is a feature of the 7-series FPGAs [23]. One LUT element can be configured as a 32×1 bit dual port RAM, and several such elements can be connected together to form RAMs. For wide but shallow RAMs, such as the weight store, this is a good solution since it allows just the right number of LUTs to be used to create the desired RAM instead of instantiating many under-utilized block RAMs.

In the case of sample delay, the reason for the varying block RAM usage turned out to be sub-optimal inference on the part of the Vivado synthesis tool. When instantiating block RAMs to implement an array in RTL code, the synthesis tool will extend its depth to the closest power of 2 [24]. For large RAM depths, this can have huge effects. For instance, if the desired depth is 20000, the depth actually implemented will be rounded up to $2^{15} = 32768$, which is an increase of 64%. The suggested solution to this by Xilinx is to manually write the RTL in such a way that the intended memory is split into smaller chunks that are powers of 2. In the example case with a depth of 20000, the memory can be split into one section of depth $2^{14} = 16384$ and one section of the remaining depth of 3616, which will be extended to 4096 when inferring block RAM. The result is a much improved total of 20480 instead of 32768.

To perform this split of large block RAMs into smaller power of 2 sized block RAMs, a small wrapper module was written. The wrapper module decomposes the given depth into powers of 2 larger than a given lower limit. For instance, it will decompose 20000 into $2^{14} + 2^{12}$. This is all done at elaboration time, and the result is used to instantiate just the right block RAMs to cover the depth given. Address decoding logic is also instantiated, such that the correct block RAM is selected for different address ranges.

7.2.1.2 Serial implementation results

The area utilization in terms of LUTs and registers is shown in Table 7.2 and illustrated in Figures 7.2 and 7.3 for different values of P. In addition to the total utilization, the utilization of the dot product, predictor and weight update modules are included. Other modules are not included because their utilization is independent of P. For the typical value of $P = 3$, utilization is 2952 LUTs and 2469 registers, which correspond to 1.71%

and 0.72% utilization on a Zynq Z-7035 SoC, respectively. As the plots show, both the number of LUTs and the number of registers scale linearly with P, which is expected as P determines the size of weight vectors and local difference vectors, which in turn determine the number of terms to compute in the dot product and the number of weights to process in parallel in the weight update module.

Table 7.4 shows resource utilization for different D, and Figures 7.4 and 7.5 show the total number of LUTs and a breakdown of the LUT usage in the components that are affected by D, respectively. Interestingly, DSP usage is higher for smaller D than for larger D. It is not immediately clear why this is the case. The DSPs are used in the dot product to perform each of the C_z multiplications needed to compute each term in $\mathbf{W}_z(t) \cdot \mathbf{U}_z(t)$, and hence 6 is the expected number for the default test parameters where $P = 3$ and full prediction mode is used ($C_z = 6$). The use of 7 DSP blocks for $D < 16$ might be due to optimizations performed by the synthesis tool under its given set of optimization settings. In all synthesis and implementations done here, the default optimization settings are used. Figure 7.5 does indeed show that the LUT usage for the dot product module is very low for $D < 16$, indicating that some of the addition logic might be done in an additional DSP instead of with LUTs and carry chains. For $D = 16$ one less DSP is used, but also the number of LUTs increases sharply. By placing stricter constraints on DSP usage, the number of DSPs used could assumably be reduced.

Increasing the weight resolution, Ω has little impact on LUT usage, but some impact on register usage as Table 7.5 shows. Larger Ω increases the dot product LUT usage due to larger adders in the adder tree, but the predictor and weight update total is fairly constant. Register usage increases by 20% from the lowest to the highest value of Ω. This is due to wider pipeline registers needed to store the weight vector from the weight store and until it is used in weight update. These results show that the choice of Ω has only a modest impact on resource usage, and given the impact it has on compression ratio, it should be set to to the maximum value.

Block RAM usage is shown in Fig 7.6, for different values of D. For $D = 8$, 20 block RAMs are used, while for the maximum $D = 16$, 36 block RAMs are used, corresponding to 14% and 25% of available block RAM of a Zynq Z-7020 SoC.

P	LUTs				Registers			
	Dot.	Pred.	Weight.	Total	Dot.	Pred.	Weight.	Total
0	111	344	310	2258	133	121	412	1686
3	270	531	617	2952	272	238	807	2469
6	420	741	964	3716	501	288	1206	3280
9	567	950	1297	4426	583	347	1612	3950
12	712	1157	1621	5123	721	404	2011	4667
15	909	1364	1945	5872	862	461	2410	5387

Table 7.2: LUT and register usage for different number of previous bands P

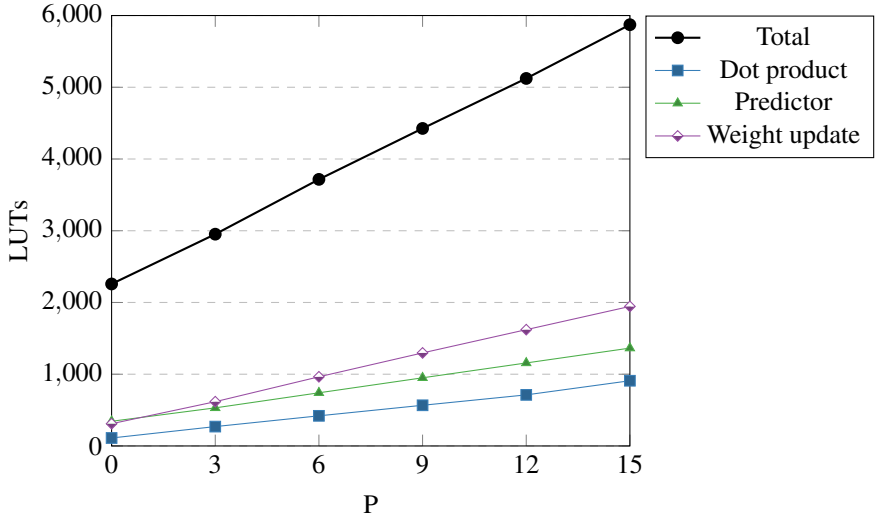

Figure 7.2: LUT usage in total and in dot product, predictor and weight update, for different values of P

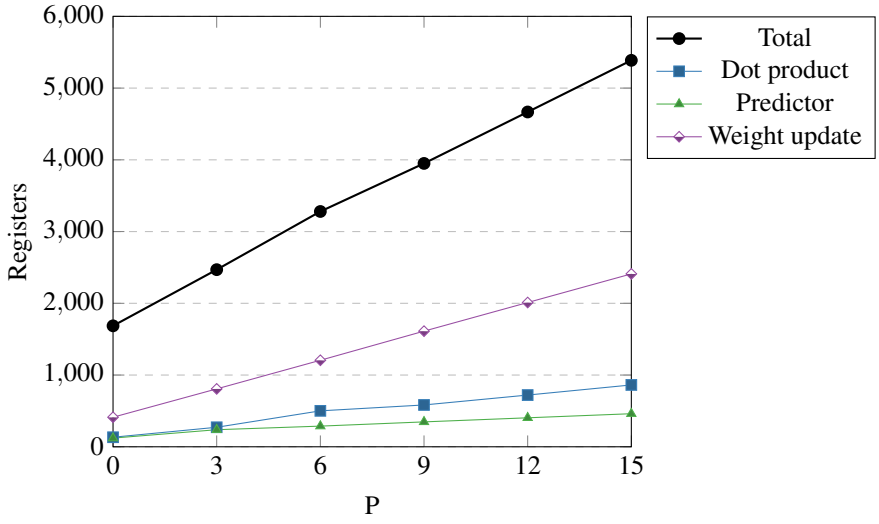

Figure 7.3: Register usage in total and in dot product, predictor and weight update, for different values of P

D	LUTs					DSP
	Dot.	Pred.	Weight.	Coder	Total	
8	77	337	476	281	1892	7
10	83	414	513	355	2184	7
12	89	457	549	423	2439	7
16	270	531	617	577	2952	6

Table 7.3: LUT and DSP usage for different sample widths D

D	Dot.	Pred.	Weight.	Coder	Total
8	74	139	682	168	1758
10	76	162	707	193	2143
12	82	184	755	213	2285
16	272	228	807	250	2469

Table 7.4: Register usage for different sample widths D

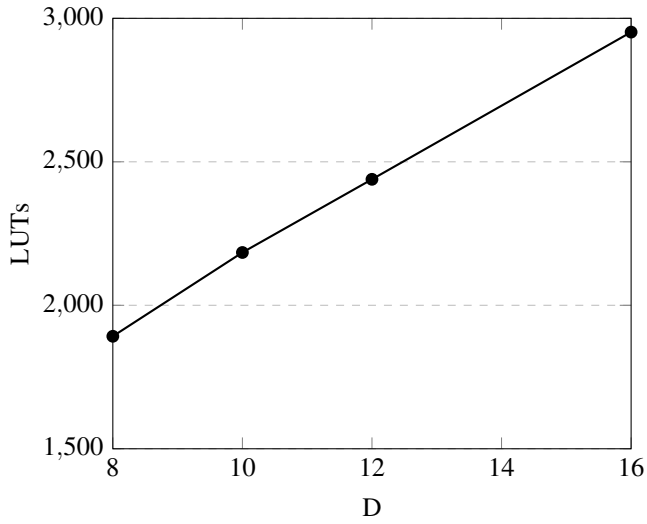

Figure 7.4: LUT usage for different values of sample width D

Ω	LUTs				Registers			
	Dot.	Pred.	Weight.	Total	Dot.	Pred.	Weight.	Total
4	86	414	793	2823	79	228	600	2067
8	90	450	751	2821	81	228	656	2199
12	94	486	668	2777	87	228	704	2323
16	255	512	584	2886	257	228	737	2366
19	270	531	617	2952	272	238	807	2469

Table 7.5: LUT and register usage for different choices of weight resolution

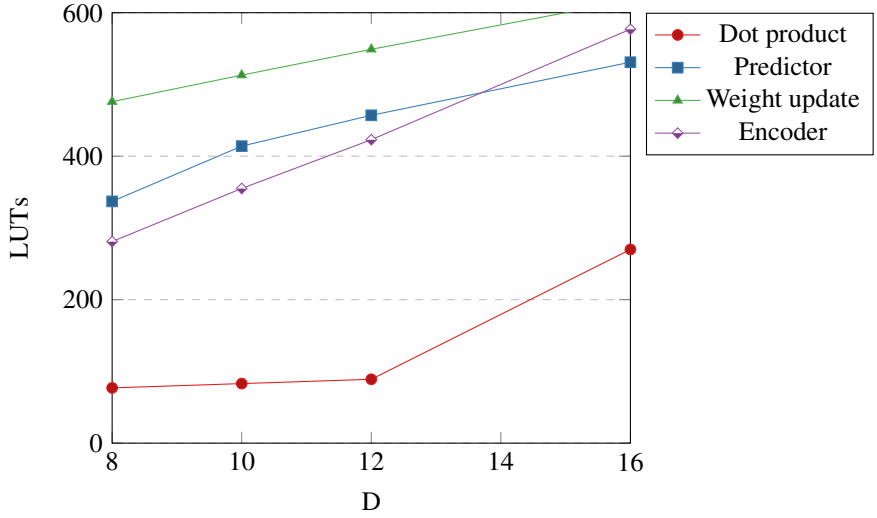

Figure 7.5: LUT usage in dot product, predictor and weight update, for different values of sample width D

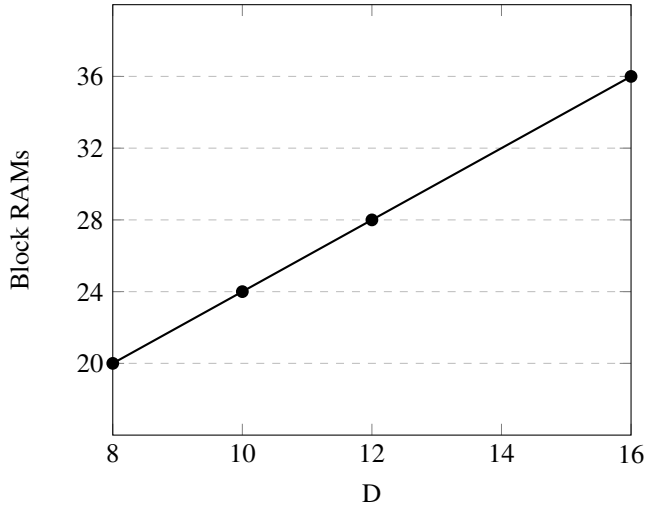

Figure 7.6: Block RAM usage for different values of sample width D

7.2.1.3 Parallel implementation results

Tables 7.6, 7.7 and 7.8 show resource utilization in terms of LUT, registers, block RAM and DSPs in total and broken down into the main components shown in Figure 6.18.

Figure 7.8 and 7.7 show the resource utilizations in terms of percentages in total of the available resources in a Zynq Z-7020 SoC and a Zynq Z-7035 SoC, respectively. From these figures it is clear that resource utilization grows linearly with N_p across the whole range, and that for both FPGAs, available LUT resources are the limiting factor for how large N_p can be made with the given set of compression parameters and image size.

Figure 7.9 shows how many of the utilized LUTs and registers are used by pipeline logic (local sum and difference calculations, prediction, weight update, residual mapping and encoding). For $N_p \geq 4$ this stabilizes at around 72%, meaning that resource utilization grows about equally for pipeline logic and the other components (weight store, accumulator store, sample delay and local difference store).

As Figures 7.10 and 7.11 show, the packer is by far the largest contributor to LUT and register usage among the components other than the pipelines. As N_p grows, the size of the combiner chains ($N_p \leq 4$) and eventually the number of combiner chains in parallel also increases ($N_p > 4$). As the number of combiner chains increases, the number of block sets to select in the output logic also increases and requires larger muxes to select block sets. The close to linear LUT utilization in the weight store, accumulator store and sample delay is due to the use of distributed memory, where LUT elements are used as memory elements. Similarly, register usage in these modules scales linearly with N_p as each lane has its own memory element with read data registers.

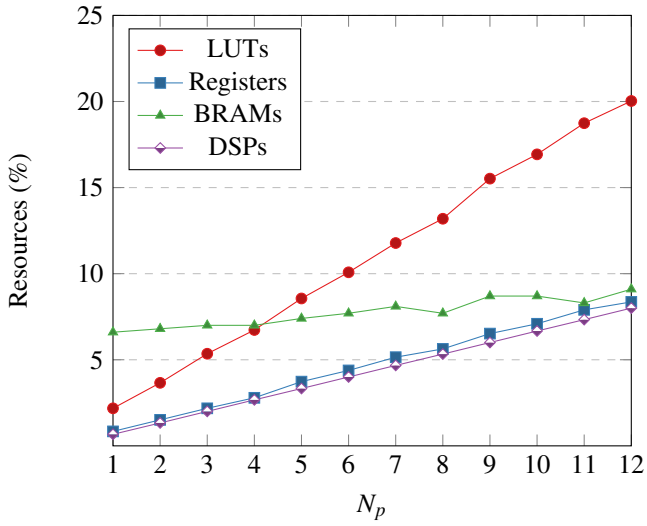

Figure 7.7: Resource utilization on Zynq Z-7035

N_p	Pipelines	Sample store	Acc. store	Weight store	Packer	Total
1	2137	468	112	504	526	3747
2	4247	672	128	366	884	6297
3	6435	866	196	566	1139	9202
4	8499	856	180	366	1665	11566
5	10723	1029	230	464	2263	14709
6	12765	1226	272	555	2513	17331
7	15005	1458	317	647	2826	20253
8	16550	1802	350	731	3238	22671
9	19297	2042	397	815	4131	26682
10	21191	1886	440	923	4668	29108
11	23584	2186	416	1014	5008	32208
12	25136	2268	454	1112	5455	34425

Table 7.6: LUT and register utilization in pipelines, sample delay, accumulator store, weight store and packer for different N_p

N_p	Pipelines	Sample store	Acc. store	Weight store	Packer	Total
1	1856	156	36	152	687	2887
2	3532	238	56	280	1069	5175
3	5394	351	78	410	1255	7488
4	6869	440	98	546	1636	9589
5	8921	540	120	670	2579	12830
6	10424	648	142	814	3033	15061
7	12460	756	164	951	3358	17689
8	13455	808	184	1085	3810	19342
9	15994	909	206	1209	4094	22412
10	17311	1000	228	1332	4507	24378
11	19546	1100	250	1476	4784	27156
12	20479	1200	272	1611	5189	28751

Table 7.7: Register utilization in pipelines, sample delay, accumulator store, weight store and packer for different N_p

N_p	Block RAM			DSPs
	Sample delay	Packer	Total	
1	32	1	33	6
2	32	2	34	12
3	33	2	35	18
4	32	3	35	24
5	32.5	4.5	37	30
6	33	5.5	38.5	36
7	35	5.5	40.5	42
8	32	6.5	38.5	48
9	36	7.5	43.5	54
10	35	8.5	43.5	60
11	33	8.5	41.5	66
12	36	9.5	45.5	72

Table 7.8: BRAM and DSP usage for different N_p

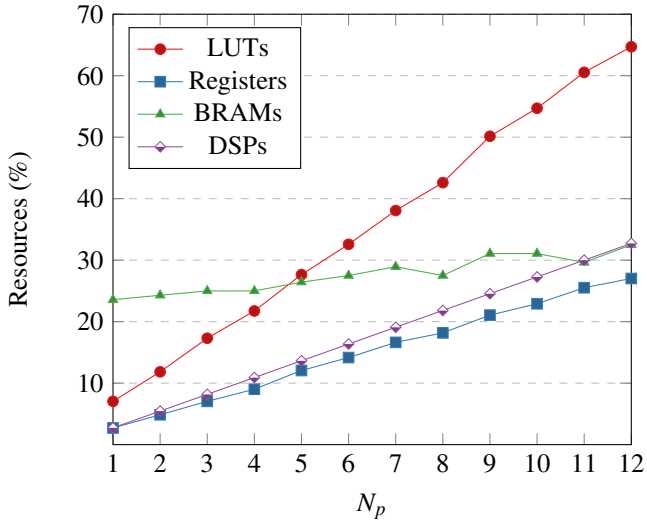

Figure 7.8: Resource utilization on Zynq Z-7020

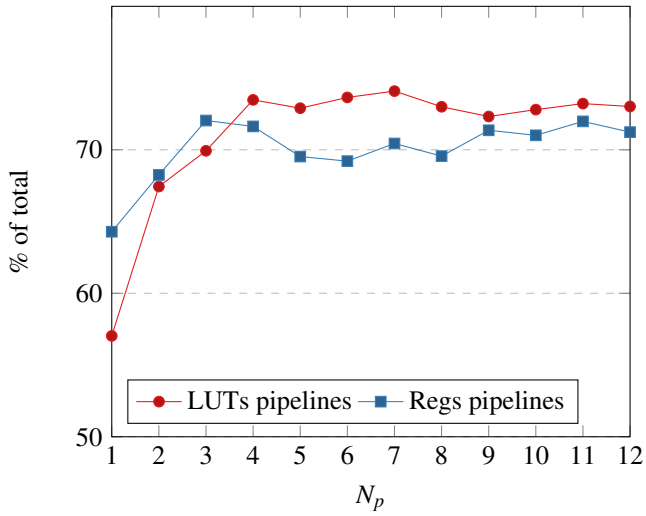

Figure 7.9: Ratio of number of LUTs and registers used by pipeline logic by the total number of LUTs

The LUT usage of the packer is examined for different number of words per chain and different block sizes in Figure 7.12 when $N_p = 4$. Regardless of the number of words per combiner chain, it is clear that choosing the smallest possible block size can reduce area usage greatly. There is also a clear benefit in using more words per combiner chain than one, although this leads to worse storage efficiency in the block set FIFOs which might require deeper FIFOs to be used. This is a tradeoff that must be investigated and tuned using the expected image statistics for the particular application the core is being used for.

Figure 7.13 shows how the LUT usage in the FIFO scales with the maximum encoded word length $U_{\max} + D$ for three choices of block size when $N_p = 4$. The growth in utilization is highest for block size 64, while quite modest for lower block sizes. When larger block sizes are used it is clear that finding a good compromise on $U_{\max}$ in terms of compression performance versus area usage can be important to reduce LUT usage.

7.2.1.4 Utilization for different image sensors

Table 7.9 shows resource utilization for a variety of different image sensors. The main factor that influences the differences between these results is $N_x \cdot N_z$, since it decides the amount of memory that is needed to store delayed samples, weights and accumulators. Since LUTs are used as RAM for storing weights, accumulators and in the one-pixel delay FIFOs, LUT usage increases with $N_x \cdot N_z$. The amount of block RAM used also increases as the depth of the NE FIFO in the sample delay module increases. Images from the IASI image sensor with its large spectral resolution of 8461 are too large to

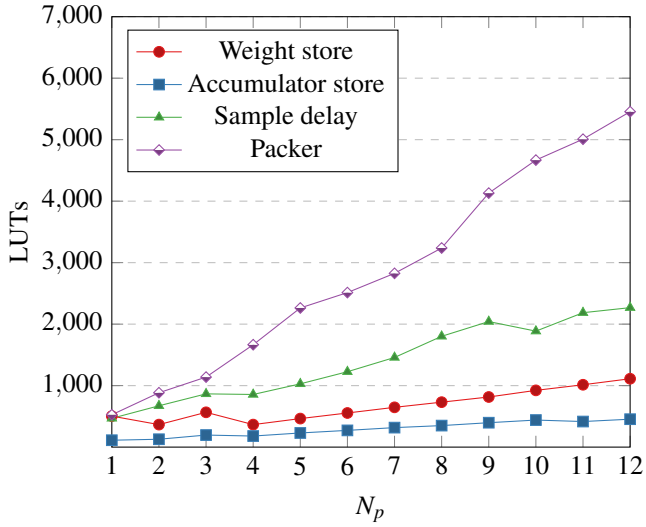

Figure 7.10: Total LUT usage for weight store, accumulator store and sample delay for parallel implementation

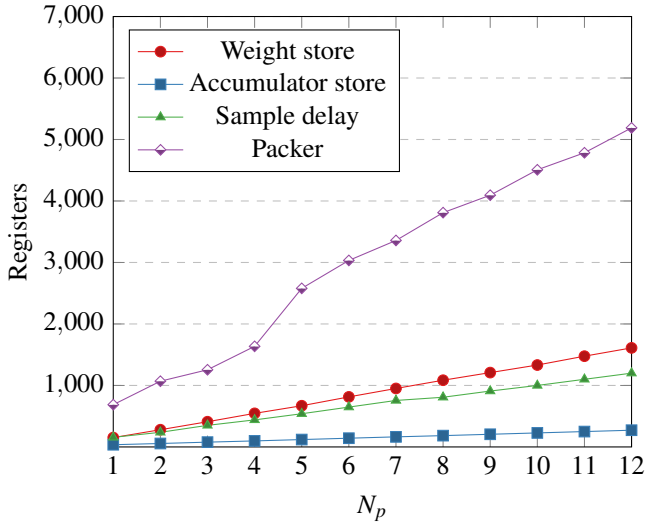

Figure 7.11: Total register usage in weight store, accumulator store and sample delay for parallel implementation

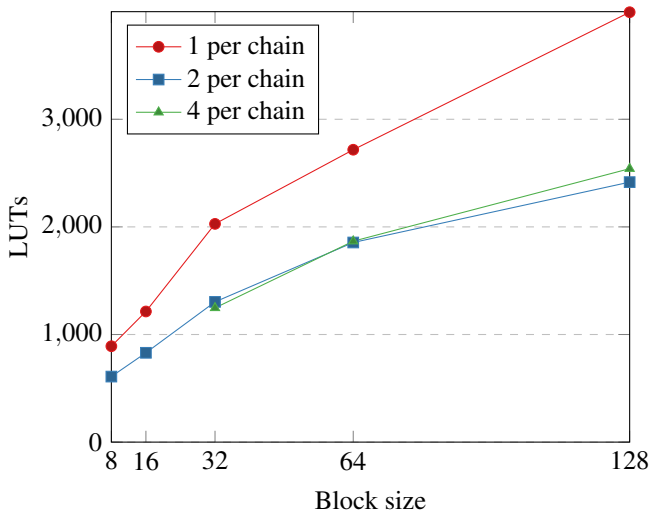

Figure 7.12: LUT usage for different block sizes when processing different number of words per combiner chain, when $N_p = 4$

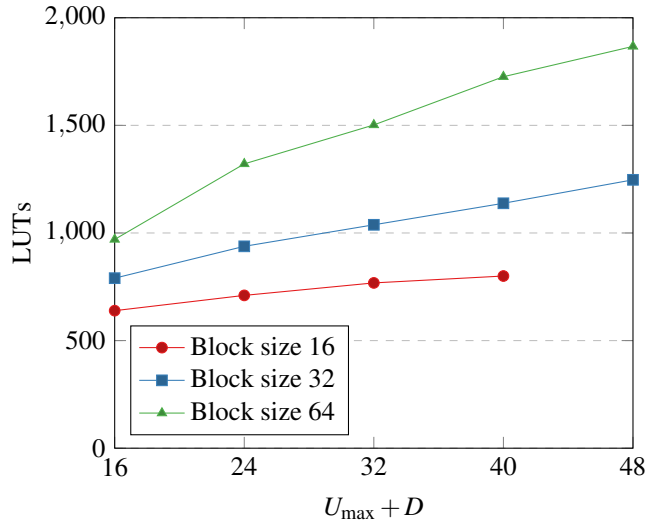

Figure 7.13: LUT usage for different block sizes and different maximum variable word lengths $U_{\max + \mathrm{D}}$ when $N_p = 4$

Model	D	N_x	N_y	N_z	LUTs	Regs	BRAM	DSPs
SFSI	12	496	140	240	9416	8730	46	28
MSG	10	3712	3712	11	7984	8133	16	28
MODIS	12	1354	2030	17	8859	8682	12	28
M3-Target	12	640	2843	260	10824	8827	64	28
M3-Global	12	320	28283	386	11351	9086	48	28
Landsat	8	1024	1024	8	6583	7410	7	28
Hyperion	12	256	3242	242	9640	8888	28	28
CRISM-FRT	12	640	510	545	12882	9313	130	28
CRISM-HRL	12	320	480	545	12646	9130	68	28
CRISM-MSP	12	64	2700	74	8803	8843	6	28
CASI	12	405	2852	72	8922	8960	16	28
AVIRIS	16	614	512	224	12033	10696	71	24
AIRS	14	90	135	1501	12191	8569	68	28
IASI	12	66	60	8461	-	-	-	-
HICO	16	512	2000	128	11589	10661	35	24

Table 7.9: Resource utilization needed to compress images from different sensors, with $N_p = 4$

be able to compress on a Zynq-7020 or Zynq-7035 and are therefore excluded from this comparison.

7.2.2 Timing

In the serial implementation, the top ten critical paths of the design are in the first pipeline stage of the local sum computation, where two samples of size D are summed. The worst negative slack (WNS) is 3.20ns when synthesized for meeting 10ns (100 MHz) constraints, meaning that the design could run at approximately 147 MHz with a throughput of 147 Msamples/s.

The WNS of the parallel implementation is shown in Figure 7.14. Even though the value is fluctuating, a general downward trend can be observed. The critical path is related to the logic that produces the *last* signal by or-ing the *last* signal from the control module in each of the pipelines. It is not immediately clear why this turns out to be the critical path, especially since the negative slack varies sharply from one value of N_p to the next (e.g. from 8 to 9). It should be noted that since these timing results are produced at default synthesis settings and with a constraint on 100 MHz, the fluctuations in negative slack might simply have to do with the synthesis tool weighing other factors, such as area, differently for different N_p.

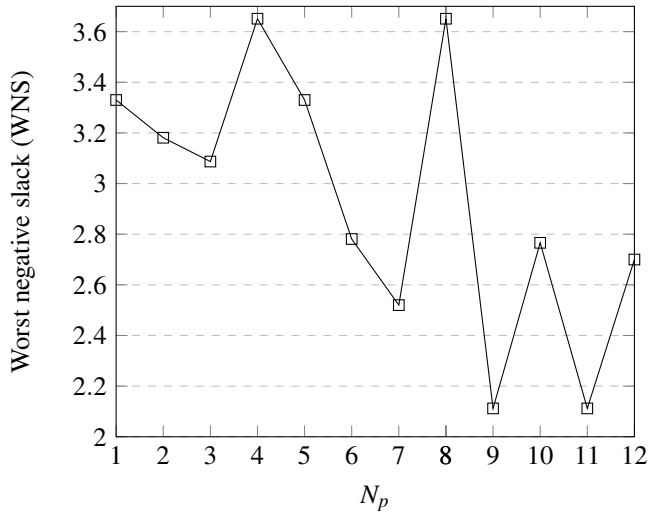

Figure 7.14: Worst negative slack (WNS) for different number of pipelines

7.2.3 Power estimation

Power usage of the implementation has been estimated using the power estimation facilities in Xilinx Vivado. To get the most accurate estimates, the post-implementation design has been used together with simulation data from post-implementation functional simulation such that realistic activity factors for the signals in the design are used.

Figure 7.15 shows estimates for $1 \leq N_p \leq 8$. Power usage increases roughly linearly with N_p. The linear increase is due to the added logic for each pipeline as well as the increasing complexity of the packer. It can also be observed that there are some fluctuations in the power contribution from the weight, accumulator, sample and local difference stores, with minimums when N_p is 1, 2, 4 and 8. This is mainly due to the different number of block RAMs that are instantiated in the NE FIFO of the sample delay module for different N_p. Block RAM inference is most effective when the depth is a power of two, as was detailed in Section 7.2.1.1. The depth of each FIFO is given by $N_x N_z / N_p$, which in this case with $N_x = 512$ and $N_z = 128$, will be a power of two if N_p also is a power of two. For other values of N_p, more block RAMs will be instantiated per lane, thus increasing both static and dynamic power consumption in these cases.

Figure 7.16 shows how much of the total power is dynamic power. Static power consumption in the design is mainly due to leakage in the memories used in the stores, which fluctuates around 0.125W in a similar way as the dynamic power of the stores shown in 7.15.

N_p	Power (W)			
	Packer	Stores	Pipelines	Total
1	0.009	0.088	0.154	0.251
2	0.018	0.095	0.195	0.308
3	0.038	0.144	0.247	0.429
4	0.047	0.112	0.282	0.441
5	0.055	0.142	0.318	0.515
6	0.062	0.17	0.353	0.585
7	0.087	0.202	0.425	0.714
8	0.098	0.157	0.443	0.698
9	0.111	0.185	0.517	0.813
10	0.129	0.203	0.562	0.894

Table 7.10: Power usage for different N_p

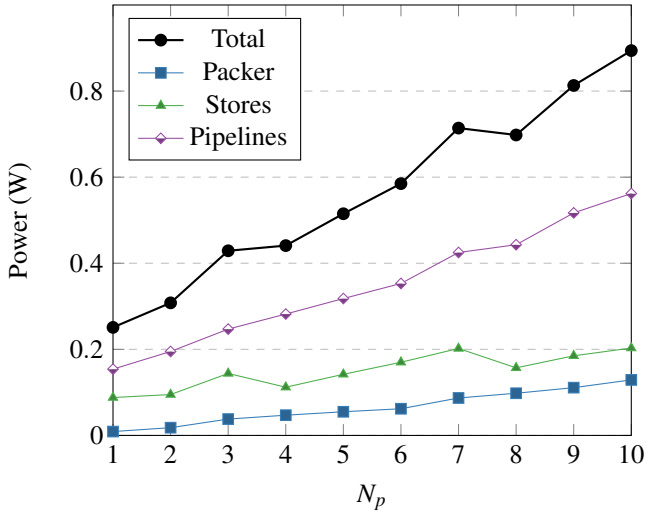

Figure 7.15: Power estimates for different N_p. Stores refer to the sum of the power used in the weight, accumulator, sample and local difference stores.

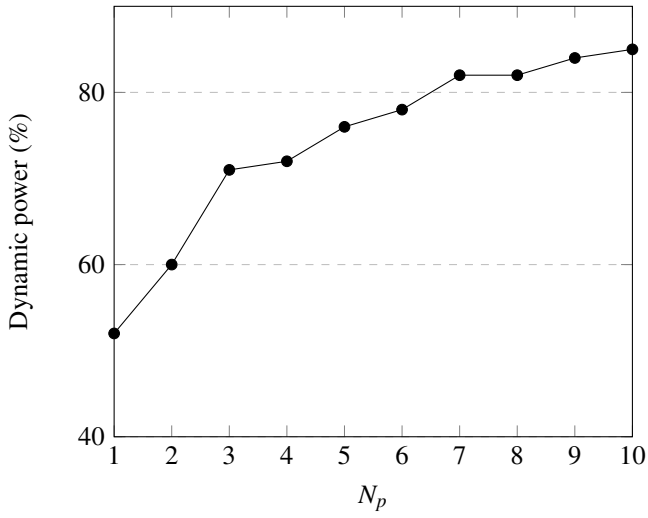

Figure 7.16: Dynamic power as percentage of total power usage

7.3 Comparison with existing work

Table 7.12 shows the performance of the implementations that were listed in Table 6.4, together with the design presented in this thesis. Table 7.11 shows the values of P and D that have been used, as well as the hardware platform.

Based on the information that is present in the papers, all the BIP implementations have a roughly similar architecture, but large differences in performance can be observed. The implementation by Theodorou et al[17] achieves a high maximum frequency for the Virtex-5 series of devices, which seems to suggest a highly pipelined architecture, similarly to the one presented in this thesis. The implementation by Keymeulen et al [13] does not achieve more than 40 MHz on the same device, suggesting less pipelined architecture. The implementation by Bascones et al [15] cannot achieve more than 50 MHz on a Virtex-7 device, and at that frequency the throughput is less than 50 Msamples/s, suggesting that the implementation does not compress one sample per clock cycle. The paper does not go into implementation details, so further comparison is not possible.

The parallel implementation done in this work is much faster than any previous designs that have been looked at. The key difference between this implementation and the parallel implementation proposed by Bascones et al[18] is that variable-length word packing is performed fully in parallel in this implementation. As long as the output stream bandwidth is large enough, this means that N_p samples can be compressed per clock cycle. Another advantage in this implementation is that the pipelines are fully utilized even if N_p does not divide N_z. In the implementation by Bascones et al, each clock cycle where the last samples in a pixel are transferred will contain some lanes that are empty, thus not utilizing the parallel cores fully.

Implementation	Order	P	D	Platform	On-the fly
UAB Emporda [20]	All	15	16	i7 7500U	No
Keymeulen et al [13]	BIP	3	13	Virtex-5	No
Santos et al [14]	BSQ	3	16	Virtex-4	No
Bascones et al [15]	BIP	15	16	Virtex-4	Yes
Bascones et al [15]	BIP	15	16	Virtex-7	Yes
Theodorou et al [16]	BIP	3	16	Virtex-5	No
Proposed - Serial	BIP	3	16	Zynq-7000	Yes
Proposed - Parallel	BIP	3	16	Zynq-7000	Yes

Table 7.11: Summary of previous CCSDS123 implementations and the proposed implementation

Implementation	Platform	f_{max} [MHz]	Throughput [Msamples/s]	Throughput [Mb/s]
UAB Emporda [20]	i7 7500U	-	4.928 [15]	78
Keymeulen et al [13]	Virtex-5	40	40	520
Santos et al [14]	Virtex-4	134	11.2	179
Bascones et al [15]	Virtex-4	50	23.3	379
Bascones et al [15]	Virtex-7	50	47.6	760
Theodorou et al [16]	Virtex-5	110	110	1760
Proposed- Serial	Zynq-7000	147	147	2352
Proposed- Parallel - 4	Zynq-7000	156	624	9984
Proposed- Parallel - 5	Zynq-7000	150	750	12000

Table 7.12: Performance comparison of CCSDS123 implementations

8

Verification and testing

The Cube DMA and CCSDS123 implementations have been tested both in simulation and on physical FPGA hardware. This chapter will present the verification and testing that has been performed both on the Cube DMA and the CCSDS123 implementation.

8.1 Simulation

8.1.1 Cube DMA

An overview of the test bench used when simulating the Cube DMA design is shown in Figure 8.1. A Xilinx-provided RAM module with an AXI interface is used to simulate the actual RAM system in the Zynq-7000. Test data is fed into the S2MM channel stream and the Cube DMA is configured through the register interface to perform a transfer. The test data is just values counting from 0 and up. After the S2MM transfer has completed, a transfer is configured on the MM2S channel, reading from the location in memory where the S2MM channel wrote the incoming data. The test bench does not automatically check this data, but it allows for easy visual inspection of waveform diagrams showing the stream coming out of the MM2S channel.

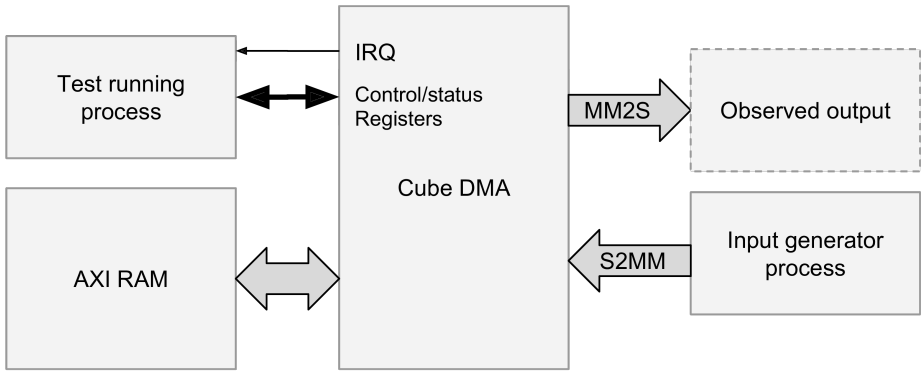

Figure 8.1: Test bench used for Cube DMA testing

8.1.2 CCSDS123

8.1.2.1 Automated verification system

An automated verification system has been created, which runs simulations with data from a given HSI cube and compares the compressed bitstream from the implemented design with the compressed bitstream from the Emporda software implementation. Spending some time on this was well worth it, as it allowed changes to be made to design with a much larger degree of confidence: If the bit streams match exactly after a code change, there is a very slim chance that the new code change introduced a bug in the code.

To be able to easily configure both Emporda and the CCSDS123 core with the exact same parameters, a common configuration file format was created, and a Python script to read such a configuration file and generate the necessary configuration file for Emporda as well as an include file for the Verilog test bench. The verification flow is shown in Figure 8.2.

The configuration is given as a JSON file, an example of which is shown in Listing 8.1. The *parameters* section lists the different CCSDS-123 parameters that are summarized in Table 5.1, as well as prediction mode and local sum mode. The *images* section contains details about the image that is going to be encoded, such as its dimensions, sample ordering and endianness.

Listing 8.1: Example of CCSDS-123 configuration

```
{"parameters":
 {"D": 16,
  "P": 5,
  "R": 32,
  "OMEGA": 14,
  "TINC_LOG": 5,
  "V_MIN": -1,
```

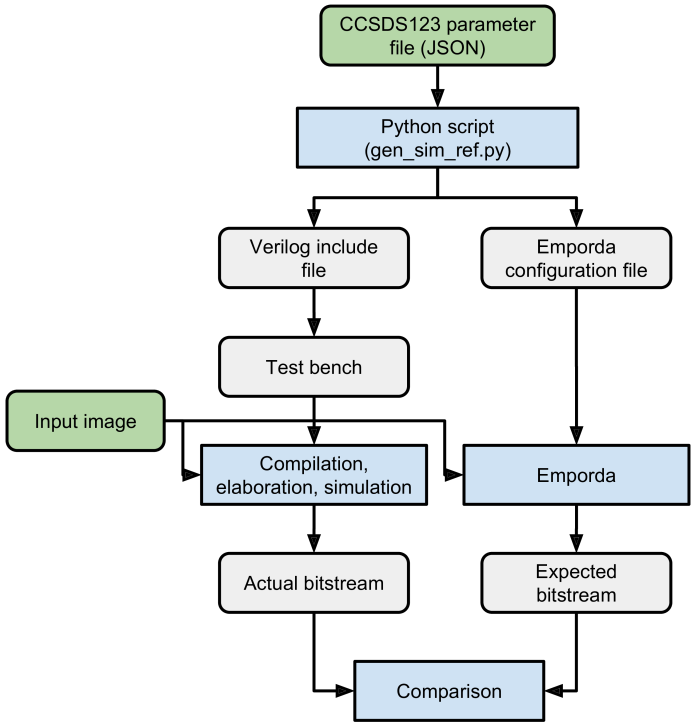

Figure 8.2: Overview of automatic verification of design

```
  "V_MAX": 1,
  "UMAX": 18,
  "COUNTER_SIZE": 6,
  "INITIAL_COUNT": 1,
  "K": 7,
  "out_word_size": 8,
  "encoder": "sample",
  "mode": "full",
  "locsum_mode": "neighbor"},

 "images": [
     {"filename": "image.bip",
      "order": "BIP",
      "endianness": "little",
      "NX": 50,
      "NY": 50,
      "NZ": 5}
 ]
}
```

A Python script `verify.py` reads the configuration file and does the following:

- Generates a Verilog file with all the parameter definitions, which is later included in the test bench.
- Generates a configuration file for Emporda and runs Emporda with the necessary command line arguments to compress the input file given in the JSON configuration.
- Strips the header from the compressed image to produce the final *golden reference* file that can be compared against

The script sets up what is needed to perform a simulation and compare the result with a golden reference. It can optionally run the simulation, or simulation can be run manually using a shell script `simulate.sh`. The script takes the file name of the image as argument, calls the Vivado-generated shell scripts `compile.sh` and `elaborate.sh`, and starts the simulation by calling the Vivado simulator `xsim` with the image file name as argument. The test bench will stream data from the image sequentially into the design, and store each 64-bit word coming out of the design into a file `out_0.bin` until the design asserts the `LAST` signal, indicating that it is done. The test bench then feeds the image into the design once more, and stores resulting 64-bit words to a file `out_1.bin`. This is done to test that the core correctly compresses a new image after a previous compression has been completed. After the simulation is done, the `verify.py` script will optionally check whether the `out_0.bin` and `out_1.bin` files are identical to the file compressed by Emporda.

The test bench also calculates some statistics about the core. It counts how many cycles the input is stalled and presents at the end of the simulation this count, as well as the total number of cycle spent, and the percentage ratio. This information is useful when tuning the packer FIFO size. The test bench also keeps track of how many of the cycles

contain valid output data and presents this in a similar fashion. This is useful to get a picture of the output bandwidth utilization.

8.1.2.2 Random parameter testing

Since the CCSDS123 algorithm accepts a wide range of parameters, it is unfeasible to test the implementation for all possible combinations of parameters. A script to perform randomized parameter testing was therefore created, which runs the above described verification flow with randomized configurations and input images with random sizes. Even if the total coverage of the input space is still limited by this approach, using random parameters improves the span of parameters that are covered and ensures that many different areas of the parameter space are covered.

The script performes a user-given number of *runs*. For each run, it picks random (but valid) values for each of the parameters listed in Table 5.1 and creates a randomly sized HSI cube. The verification steps outlined in the previous section are then performed. If verification fails, all the data that is particular for that run, such as the parameters used, input HSI cube and the bistreams, are saved to a folder such that the error can be investigated. Each failed run has its own such folder which is uniquely named using the time and date when the run was performed.

Several issues were discovered using this script:

- The number of bits used to represent the encoded word length was calculated wrongly based on the generic parameters for $U_{\max}$ and D, causing too few bits to be used in the specific case when $U_{\max} = 16$.
- Comparison of actual and expected bitstreams failed when the last bytes in the actual bitstream are 0
- Test bench did not wait for handshake when last samples were being input, causing problems when the core happens to stall the input before the last samples were accepted.

The core has been tested in simulation against the Emporda compressor for a selection of parameters and edge cases. To reduce simulation time, cube dimenions have been restricted to less than 100 for these tests. The core has also been tested on a set of special test pattern images produced by CCSDS for verifying that certain edge cases where overflows occur are handled correctly.

8.2 Testing on hardware

The Cube DMA and CCSDS123 core has been tested on a hardware Zynq 7000 SoC using a Digilent ZedBoard development board shown in Figure 8.3. The ZedBoard contains a Zynq Z-7020 SoC, 512MB DDR memory and various peripheral connections.

The board has a USB connector for connecting to the JTAG debugging system on the Zynq Z-7020 SoC.

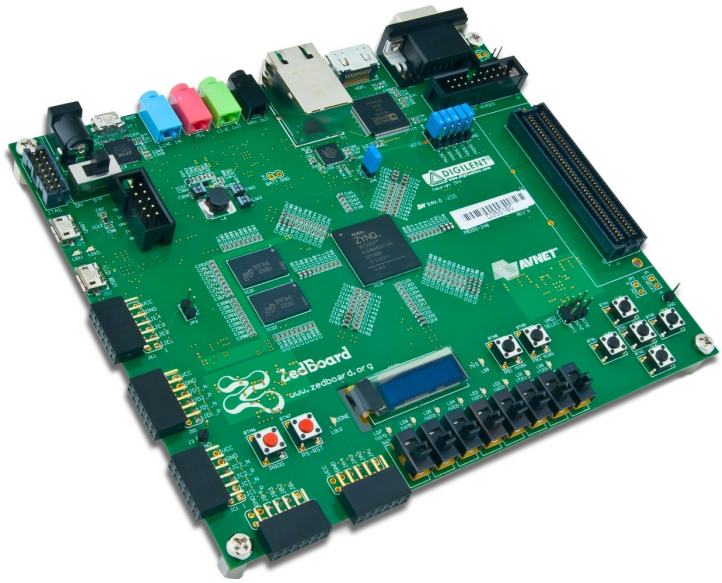

Figure 8.3: ZedBoard development board used for hardware testing [5]

An overview of the Zynq 7000 system when testing on hardware is shown in Figure 8.4. The following sections will go through various parts shown in this figure.

8.2.1 Interfacing with Zynq 7000 SoC

The Zynq 7000 system can be interacted with using a JTAG interface that connects to a PC. A Xilinx command line tool, XSDB, can be used to interact with the system by issuing commands to perform various operations:

- Access the memory map as seen from the CPUs, e.g. register interfaces instantiated in logic implemented in the FPGA
- Upload programs and run them on the CPUs
- Program the FPGA with a bitstream file
- Upload and download binary files to/from DDR memory

The XSDB tool is also scriptable using Tcl, which makes it easy to automate tests and making functions for programming registers with correct values when doing transfers.

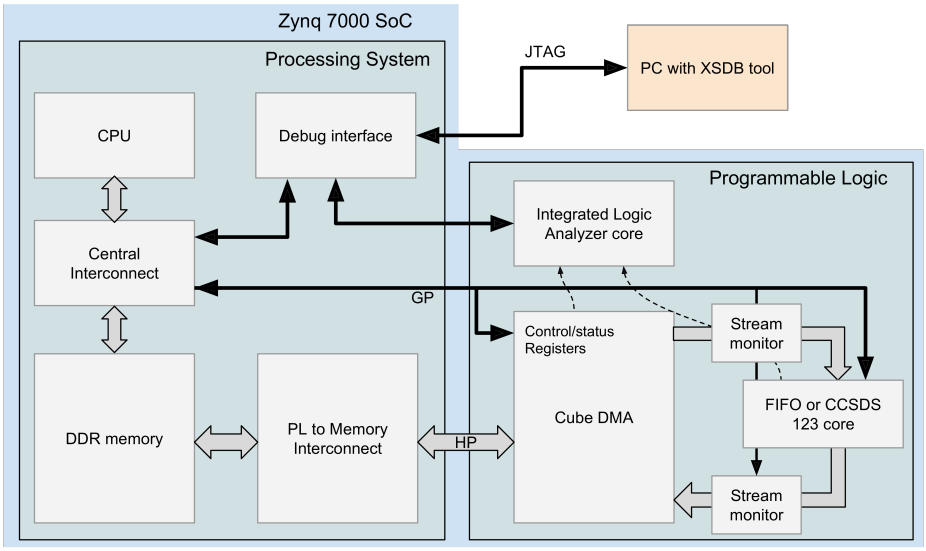

Figure 8.4: Overview of Zynq 7000 system for testing Cube DMA and CCSDS123 implementation

8.2.2 Xilinx ChipScope debugging

A feature of Xilinx FPGA products is ChipScope debugging, which is the ability to monitor signals in the FPGA design and sample them over a period of time. The sampled signals are shown in a waveform diagram, similar to when simulating. The start of the sampling period is triggered by any kind of user-defined trigger event, such as on a rising edge of a particular signal.

ChipScope debugging is set up in the Vivado GUI or by using XDC constraint files, by selecting a set of signals to monitor from the synthsized netlist. The capture of signals is performed by Integrated Logic Analyzer (ILA) cores that are instantiated in the FPGA. The ILA cores consume block RAMs to store signal transitions as they are captured. This is the main limiting factor for how many signals can be sampled, and for how long.

8.2.3 Stream monitors

For data transfers of whole HSI images, it is infeasible to use ChipScope debugging to sample the whole transfer. There are not enough hardware resources to do so, and it would be impractical to analyze enormous waveform captures. For this reason, a small VHDL module, the *stream monitor* was created, illustrated in Figure 8.5. The stream monitor sits between two AXI Stream end points and measures the following information:

- Total number of cycles from start of monitoring to **tlast** is asserted
- Number of handshakes
- Number of cycles where there is no handshake because **tready** is low
- Number of cycles where there is no handshake because **tvalid** is low

This is information that is useful for confirming that a transfer has been properly conducted (by checking that the number of handshakes matches what is expected), and to analyze the causes of slow-downs in data rate between the end points; whether it is due to the master not having valid data or whether it is the slave that is not ready to receive. In addition, it provides an accurate way of figuring out the total transfer time, which can be found by multiplying the number of clock cycles by the clock frequency.

The stream monitor has a register interface which can be connected to the PS to allow the CPU or JTAG debugging system to read out the measured counts. The register map is shown in Table 8.1. The stream monitor is by default in an idle state, where any handshake activity will trigger it to go into a measurement state where it starts counting the events that were previously listed. When **tlast** is asserted, the stream monitor goes back to the idle state again. The stream monitor also has a control register which allows for manually stopping measurements (in cases where **tlast** is not used for instance) and also manually starting measurements.

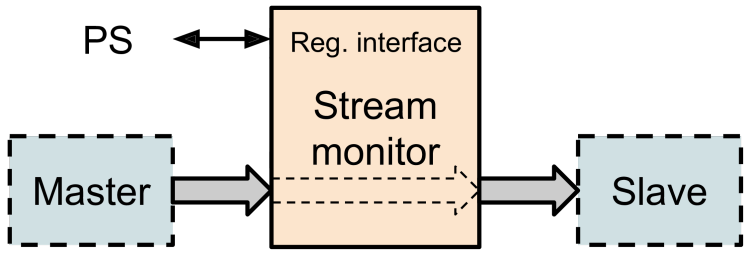

Figure 8.5: The stream monitor module

Offset	Register	Field	Bits
0	Control	Start	0
0	Control	Reset	1
4	Total count		31-0
8	Handshake count		31-0
12	Not ready count		31-0
16	Not valid count		31-0

Table 8.1: Register layout for the stream monitor module

8.2.4 Typical hardware testing flow

When testing the Cube DMA and/or the CCSDS123 core, the following steps were generally performed:

1. Set up ChipScope debugging (if needed)
2. Initialize Zynq-7000 system using XSDB scripts provided by Xilinx
3. Program FPGA with bitstream for system with Cube DMA and FIFO or CCSDS123 core
4. Connect to FPGA from Vivado GUI and set up trigger for ChipScope debugging (if needed)
5. Upload input data from PC to DDR memory on the ZedBoard via JTAG
6. Set up S2MM channel registers
7. Set up MM2S channel registers, this starts transfer
8. Check that transfer is complete by checking S2MM and MM2S status registers
9. Check stream monitor registers
10. Download resulting data from S2MM channel from memory to a binary file on the PC, via JTAG
11. Verify correctness of data

9

Conclusion

The work presented in this thesis has provided the NTNU SmallSat project with a flexible and efficient DMA core and a state of the art FPGA implementation of the CCSDS123 compression standard for hyperspectral images. This chapter will draw some conclusions from the implementation and gathered results, and ideas for future work will be presented.

9.1 Cube DMA

The Cube DMA core is adaptable to a wide range of hardware accelerators with different requirements for stream ordering, component bit widths and number of components to process in parallel. The Cube DMA provides an easier programming model from a software perspective, since a complete HSI cube transfer can be set up once through configuration registers instead of needing to set up chains of block descriptors. Packing and unpacking of data allows maximal memory utilization when hyperspectral images with component sizes other than byte multiples are used, e.g. 10 or 12 bits.

A performance comparison with the Xilinx AXI DMA has been performed. It performs similarly to the Xilinx AXI DMA for sequential transfers, and has an improvement of 128% in achievable throughput for block-wise transfers. BSQ (plane-wise) transfers are made possible by this core, achieving 73% of the theoretically achievable throughput when using the TinyMover core instead of the Xilinx DataMover.

Utilization results show that including the logic for packing and unpacking in these cases results in an increase in area utilization that is linear with the number of bits per component and the number of components per beat in the transfer. Logic utilization is

less than in the Xilinx AXI DMA, and maximal clock frequency that can be achieved is 131 MHz.

The Cube DMA core has been tested in simulation and on a physical Zynq Z-7020 FPGA using the ZedBoard development board, using all combinations of BIP and BSQ (plane-wise) orderings with and without block-wise ordering.

9.2 CCSDS123

The CCSDS123 implementation has been implemented with generic parameters covering the full range of all the parameters defined in the CCSDS123 standard. The core can compress any number of samples in parallel per clock cycle, provided that resource and I/O bandwidth constraints are obeyed.

A serial implementation which consists of one compression pipeline has been implemented first, and has been used to analyze how the number of previous bands used in prediction P, the sample width D and the weight resolution affects resource utilization. The main conclusions are that LUT, register and block RAM usage scales linearly with all three parameters. For , the increase is almost negligible, with register usage being the most affected with an increase of 20% from the lowest to highest value of .

Building on the serial implementation, the final parallel implementation has been performed, where any number N_p of computational pipelines can compress one sample per clock cycle in parallel. LUT, register and DSP utilization scales linearly with N_p. Block RAM usage remains largely independent of N_p, but does increase slowly due to additional overhead when weights, delayed samples, local differences and accumulator values must be shared between several pipelines. As N_p is increased, the LUT utilization becomes the limiting factor in terms of resources, reaching 70% of the total available LUTs on a lower-end Zynq Z-7020 device when $N_p = 12$. LUT usage can be reduced somewhat by replacing distributed RAMs used in weight and accumulator storage and sample delay with block RAM.

Analysis of LUT and register usage for different N_p reveals that for $N_p \geq 4$, the computational pipelines account for roughly 70% of the utilization, while the rest is due to storage of weight, samples, local differences and accumulators, and the variable length word packing. Of these, the packing is by far the largest contributor to both LUT and register usage. Analysis of how the packed block size, number of combiner chains and the maximum code word length $U_{\max} + D$ affects the packer has been performed, and shows that significant reductions in resource utilization can be achieved by combining at least two words per combiner chain and by reducing the block size. Reducing the maximum code word length shows only moderate improvements.

Timing analysis shows that the serial implementation can achieve a clock speed of 147 MHz, with the critical path being the local sum computation (summation of two $D+3$ bit numbers). For the parallel implementation, achievable clock speed depends on N_p,

and varies from 126 MHz to 157 MHz, with some sporadic jumps for some values of N_p but with a general downward trend as N_p increases.

Power estimation has been performed using simulation data to achieve realistic signal activity data. The power use scales approximately linearly with N_p, with a total power usage of 0.25W at $N_p = 1$ and 0.89W at $N_p = 12$. Static power remains at around 0.13W, while dynamic power is increasing as N_p is increased.

The implementation has been verified by comparing compressed images both from simulation and real hardware tests, with images compressed using an existing software implementation. To cover a broad range of parameters, randomized testing has been used, where simulation is run repeatedly using randomly chosen parameters.

Comparing with the state of the art, it is clear that the parallel implementation supersedes any previous implementations with regards to achievable compression performance. The greatest improvement in the presented implementation is the ability to pack any number of variable length words into fixed-size words each clock cycle. Other improvements include the ability to effectively handle images where the number of spectral bands N_z is not a multiple of the number of samples to compress in parallel N_p, which is achieved by performing shifting and delaying operations to reorder weights, accumulators and delayed samples in such a way that samples from different pixels can be compressed in the same clock cycle.

9.3 Future work

9.3.1 Cube DMA

The Cube DMA has been tested extensively using the sequential BIP transfer mode, since this is the mode used by the CCSDS123 core. For other transfer modes, however, testing has been performed by initiating transfers of small cubes consisting of components numbered from 0 and upwards, and checking waveform diagrams for the expected stream output. A fully automated test bench could be implemented also for the Cube DMA.

An algorithm for performing principal component analysis (PCA) is under development in the SmallSat project. This algorithm needs to sample random pixels from the HSI cube and process them. This could be performed using the Xilinx AXI DMA by creating a chain of block descriptors where each descriptor points to a random pixel in the cube. Another option could be to extend the Cube DMA implementation with support for this built-in. This would require a random number generator to be added, and the logic in the address generator to be changed for this new transfer mode.

9.3.2 CCSDS123

There is an option in the CCSDS123 standard for providing custom weights and custom accumulator initialization constants instead of using the default ones. The idea is that optimized weights and accumulators can be found for a set of training images that are similar to images that are expected to apply the algorithm on. Adding support for this would only change the code for the weight update module and in the sample adaptive encoder module, and would presumably be easy to implement.

Another area to look into is power usage. The implementation presented here has not been done with power optimization in mind. For small N_p it is doubtful that much of a difference can be made, as the Zynq-7000 processing system alone consumes quite a lot of power. For larger N_p however, there could be considerable improvements made by applying power saving techniques.

Finally, at the present moment, the core does not have any configuration interface for control by the CPUs. It keeps track of compression progress internally and will be in a state where it is ready to compress a new image when a complete image of the hard coded size has been compressed. Being able to externally reset the core's internal state could be of interest in cases of corrupt data. It could also be of interest to be able to set the image dimensions through a register interface instead of as generic parameters.

Bibliography

[1] M. E. Grøtte, J. Fortuna, R. Birkeland, J. Veisdal, M. Orlandic, H. Martens, J. T. Gravdahl, F. Sigernes, T. A. Johansen, J. O. Reberg, G. Johnsen, and K. Rajan, "Hyper-spectral imaging Small satellite in multi-agent marine observation system," *Internal Document*, vol. 1, no. Non-published, pp. 1–16, 2017.

[2] Xilinx, "Zynq-7000 All Programmable SoC Technical Reference Manual," Tech. Rep., 2016.

[3] ARM, "AMBA AXI and ACE Protocol Specification," Tech. Rep., 2011.

[4] Consultative Committee for Space Data Systems, "Lossless Multispectral and Hyperspectral Image Compression Recommended Standard, CCSDS 123.0-B-1," Tech. Rep., 2012.

[5] Xilinx, "Digilent ZedBoard." [Online]. Available: https://www.xilinx.com/support/university/boards-portfolio/xup-boards/DigilentZedBoard.html

[6] T. B. Curtin, J. G. Bellingham, J. Catipovic, and D. Webb, "Autonomous oceanographic sampling networks," *Oceanography*, vol. 6, no. Non-published, pp. 86–94, 1993.

[7] Oregon State University, "HICO Sensor and Data Characteristics," Tech. Rep., 2009. [Online]. Available: http://hico.coas.oregonstate.edu/datasets/datacharacteristics.shtml

[8] J. Fjedtvedt, *Direct Memory Access for Hyperspectral Imaging Applications*. NTNU, 2017.

[9] Xilinx, "LogiCORE IP Product Guide, AXI DMA v7.1," Tech. Rep., 2017.

[10] ——, "LogiCORE IP Product Guide, AXI Video Direct Memory Access v6.2," Tech. Rep., 2016.

[11] ——, "LogiCORE IP Product Guide, AXI DataMover v5.1," Tech. Rep., 2017.

[12] Consultative Committee for Space Data Systems, "Lossless Multispectral and Hyperspectral Image Compression Information Report, CCSDS 120.2-G-1," Tech. Rep., 2015.

[13] D. Keymeulen, N. Aranki, A. Bakhshi, H. Luong, C. Sarture, and D. Dolman, "Airborne demonstration of fpga implementation of fast lossless hyperspectral data compression system," in *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*. IEEE, 2014, pp. 278–284.

[14] L. Santos, L. Berrojo, J. Moreno, J. F. López, and R. Sarmiento, "Multispectral and hyperspectral lossless compressor for space applications (hyloc): A low-complexity fpga implementation of the ccsds 123 standard," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 9, no. 2, pp. 757–770, 2016.

[15] D. Báscones, C. González, and D. Mozos, "FPGA Implementation of the CCSDS 1.2.3 Standard for Real-Time Hyperspectral Lossless Compression," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 2017.

[16] G. Theodorou, N. Kranitis, A. Tsigkanos, and A. Paschalis, "High performance ccsds 123.0-b-1 multispectral & hyperspectral image compression implementation on a space-grade sram fpga," in *Proceedings of the 5th International Workshop on On-Board Payload Data Compression, Frascati, Italy*, 2016, pp. 28–29.

[17] ——, "High performance ccsds 123.0-b-1 multispectral & hyperspectral image compression implementation on a space-grade sram fpga," in *Proceedings of the 5th International Workshop on On-Board Payload Data Compression, Frascati, Italy*, 2016, pp. 28–29.

[18] D. Báscones, C. González, and D. Mozos, "Parallel Implementation of the CCSDS 1.2.3 Standard for Hyperspectral Lossless Compression," *MDPI Remote Sensing*, 2017.

[19] European Space Agency, "ESA Reference Implementation of CCSDS123," 2017. [Online]. Available: https://www.esa.int/Our_Activities/Space_Engineering_Technology/Onboard_Data_Processing/Data_Compression_Tools

[20] GICI group, Universitat Autonoma de Barcelona, "Emporda Software," 2011. [Online]. Available: http://www.gici.uab.es

[21] Oregon State University, "HICO Instrument Design and Heritage," Tech. Rep., 2009. [Online]. Available: http://hico.coas.oregonstate.edu/hico/design.shtml

[22] Xilinx, "7 Series FPGAs Memory Resources User Guide," Tech. Rep., 2016.

[23] ——, "7 Series FPGAs Configurable Logic Block User Guide," Tech. Rep., 2016.

[24] ——, "Answer Record #61995," Tech. Rep., 2018. [Online]. Available: https://www.xilinx.com/support/answers/61995.html

A

Using the automatic verification scripts

This appendix will go through in detail how to use the automatic verification scripts that were mentioned in Section 8.1.2. The verification scripts currently only work in a Linux environment, as several Linux tools such as `dd`, `od`, and `cmp` are used, in addition to Bash scripting.

A.1 Installing Emporda

Emporda can be downloaded from the following website: `http://gici.uab.cat/GiciWebPage/emporda.php`.

Emporda is a Java program which comes pre-built as `dist/emporda.jar` in the extracted directory tree. The source code is also included in the archive, and can be compiled using the `ant` tool (requires a Java Development Kit to be installed). The pre-built program is enough to just run Emporda.

Installing Emporda can most easily be performed by copying the `emporda.jar` to `/usr/bin` and creating the following bash script in `/usr/bin`:

/usr/bin/emporda

```
#!/bin/bash
java -jar /usr/bin/emporda.jar "$@"
```

The bash script should be given the file name `emporda` (note: no extensions), such that it can be invoked from the command line by just typing `emporda`.

An alternative is to keep the script somewhere else, e.g. in the same directory where Emporda was extracted, and instead add that directory to the PATH variable.

When these steps have been completed, Emporda should run when typing `emporda`.

A.2 Using the automatic verification scripts

The automatic verification system assumes that Emporda is started by running the command `emporda`, so it is important to set up Emporda as described in the previous chapter.

The verification scripts found in the `tools` directory of the CCSDS123 source tree are the following:

- **gen_cube.py**: Generate a cube of any given size
- **verify.py**: Generate golden reference and simulation include files for performing verification using a specific image and a specific set of parameters defined in a JSON configuration file. Optionally also run simulation and check results
- **fuzzer.py**: Perform randomized testing ("fuzzing") by creating random images and instantiating the core with random sets of parameters

All three scripts use a common set of functions defined in `ccsds_lib.py`.

A.2.1 Creating a simulation snapshot

To be able to run a simulation, the design must be elaborated for simulation use. The easiest way to do this is to open the CCSDS123 project file (`project/project.xpr`) and start a behavioral simulation. This will run the necessary elaboration commands to create a new simulation snapshopt at `project/project.sim/sim_1/behav`. This can also be accomplished using command line tools.

A.2.2 Generating a cube

Before running verification, a HSI cube must be generated (if none is available). The tool `gen_cube.py` will create a HSI cube where the sample values are just numbers counting upwards, starting at 0.

```
python tools/gen_cube.py image.bip 50 50 50
```

It is important to match the file name and the dimensions with what is given in the `conf.json` file.

A.2.3 Creating configuration file and running

To run verification of the CCSDS123 core with a specific image and a set of parameters, a configuration file in JSON format has to be written. The following is an example of such a file. This file will from now on be referred to as `conf.json`, but of course the file can have any name.

conf.json

```
{"parameters":
 {"D": 16,
  "P": 5,
  "R": 32,
  "OMEGA": 14,
  "TINC_LOG": 5,
  "V_MIN": -1,
  "V_MAX": 1,
  "UMAX": 18,
  "COUNTER_SIZE": 6,
  "INITIAL_COUNT": 1,
  "K": 7,
  "out_word_size": 8,
  "encoder": "sample",
  "mode": "full",
  "locsum_mode": "neighbor"},

 "images": [
     {"filename": "image.bip",
      "order": "BIP",
      "endianness": "little",
      "NX": 50,
      "NY": 50,
      "NZ": 50}
 ]
}
```

When this file has been filled with the desired parameters and image information, verification is run as follows:

```
python tools/verify.py run conf.json
```

This will run Emporda to generate the golden reference from the given image, write the simulation parameter file to be included in the test bench, run the simulation script that calls the Xilinx simulator `xsim`, and compares the resulting bitstreams afterwards and reports whether they are correct.

Optionally, the simulation parameter and golden reference file names can be given as arguments to `verify.py`. If not given, the defaults are `params.v` and `golden.bin`.

A.2.4 Running manually or in Vivado GUI

Verification can also be run manually if that is desired. In that case, `verify.py` can be run in `generate` mode:

```
python tools/verify.py generate conf.json
```

This will create the simulation parameters file `params.v` and the golden reference file `golden.bin`. Simulation will not be run when the `generate` parameter is given.

To run a manual simulation run, the `simulate.sh` script found in the root directory of the source tree is used:

```
./simulate.sh image.bip params.v
```

An optional argument, `BUBBLES` can be supplied at the end of the command to make the test bench insert "bubbles" in the input and output streams. At random times, the input stream will then be stalled (`tvalid` is low) or the output stream be stalled (`tready` is low). This tests that the core can handle streaming stalls correctly without data corruption or data loss.

To run the simulation in the Vivado GUI, an image and a parameter file must be created by using `verify.py` in `generate` mode. To be able to run the simulation correctly, the two generated files must be copied. The parameter file must be moved/copied to the `tb` source folder:

```
cp params.v tb/comp_params.v
```

The image must be copied to where the simulation snapshot is:

```
cp image.bip project/project.sim/sim_1/behav/test.bin
```

`test.bin` is the default filename used in the test bench, and it is expected to be found in the simulation snapshot folder. If another kind of simulation is to be run, e.g. a post-implementation functional simulation, the path is changed accordingly to where that simulation snapshot has been created.

A.3 Performing randomized testing

The `fuzzer.py` script performs randomized testing of the CCSDS123 core. Running this script requires a simulation snapshot to be set up, as described in Section A.2.1. When this has been done, random testing is performed by running

```
python tools/fuzzer.py N
```

where N is the number of random tests to perform. For each of the N iterations, the verification flow described in the previous sections is performed, but with a randomly sized cube and randomly selected parameters. For each iteraton, Emporda is used to

create a golden reference, and the results from running simulation are checked with it. After all runs have finished, a summary looking like this will be printed:

```
Done. 5 out of 5 tests passed.

Parameters that have been covered:
PIPELINES: 4, 7, 2, 8
D: 16
OMEGA: 5, 17, 12, 16
K: 5, 7, 10, 11, 9
TINC_LOG: 7, 8, 6, 5, 4
COUNTER_SIZE: 5, 7, 8, 6
encoder: sample
V_MAX: 3, 9, 2, -6
P: 9, 3, 6, 1, 11
R: 46, 38, 57, 54, 48
mode: full, reduced
UMAX: 14, 8, 11, 21
locsum_mode: column, neighbor
out_endianness: little
out_word_size: 8
V_MIN: 1, 7, 0, -1, -6
INITIAL_COUNT: 2, 6, 1, 4
```

The list shows each of the different parameter values that have been exercised. During runs, the specific set of parameters for each run is also printed.

In some cases it might be interesting to constrain one or more of the parameters to a specific value. For instance, it might of interest to test the core exclusively when $= 8$. To do this, a set of fixed parameters can be defined in a JSON file whose filename is supplied as the second argument to `fuzzer.py`:

```
fuzzer.py 5 fixed.json
```

A.3.1 Handling of failed tests

If a test fails, either because simulation itself could not run properly or if the bitstreams do not match the golden reference, then all the files related to that particular test run is copied into `failed_runs/MMDD-hhmmss/`, where `MMDD` is the month and date in two digits, and `hhmmss` is the hours, minute and second when the test failed. For example, a failed run might be found in `failed_runs/0626-083025`. Inside the directory for the failed run, the following files are found:

- `conf.json`: A configuration file with the parameters that were used
- `gen_comp_params.v`: Include file with the parameters used
- `golden.bin`: The golden reference that was used

- `input.bip`: The generated randomly sized input image
- `out_0.bin`, `out_1.bin`: The bitstreams that were captured from the CCSDS123 core during simulation

Saving this data for each failed test allows easy investigation of the particular failure. All information necessary to debug the problem is found in these files.

A.4 Investigating errors

When errors occur in the bitstream, it is helpful to run the simulation in the Vivado GUI to look for possible causes by investigating signals inside the core. To do this, we need to find the location in the waveform diagram where the first erroneous word is output. To find the byte position where the bitstreams start to differ, the `cmp` command in Linux can be used. Running `cmp golden.bin out_x.bin` results in something like the following output:

```
golden.bin out_0.bin differ: byte 8321, line 48
```

The output stream from the core has a width equal to the block size that is used, which is typically 8 bytes (but can be set to other sizes depending on the `BUS_WIDTH` generic parameter). Assuming a bus width of 8, we can find the first erroneous 64-bit word by using the `od` command. To do this, we must find the nearest 8-byte boundary. In the example, $8321 \bmod 8 = 1$, so $8321 - 1 = 8320$ is the 8-byte boundary where the 64-bit word starts in the file. The following call to `od` will extract the 64-bit word at this location:

```
> od -t x8 -j 8320 -N 1 out_0.bin
0020200 00000000000000db
0020201
```

This tells us that the 64-bit hexadecimal string we need to search for in the Vivado waveform window is `00000000000000db`. To perform the search, right click on the `s_axis_tdata` signal in the waveform viewer and select *Find Value...*. A search field will appear at the top of the window, where the value can be pasted, as shown in Figure A.1.

When investigating the cause of the error, it is important to keep in mind that there is a FIFO in the packer module. Depending on the occupancy of the FIFO (this can be monitored by looking at the read/write counts), the word might have been pushed into the FIFO many cycles before.

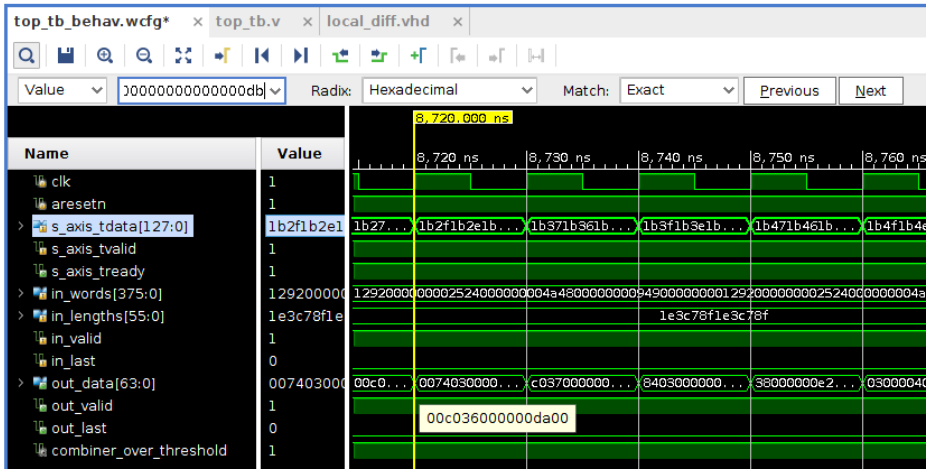

Figure A.1: Searching for waveform value in Vivado simulator

B

Using simulation results to improve power estimates

Power estimation can be performed on the implemented design by opening it in project mode in Vivado and selecting Report Power. Using the default settings, this will use probabilistic methods to arrive at a rough estimate of the power consumption of the design.

The accuracy of the power estimate can usually be improved by giving the power estimation algorithm typical activity factors for each signal in the design. This can be achieved by running a functional simulation on the implemented design and recording the transitions of each signal when applying typical (real-world) inputs. This can be done as follows:

1. Create a new top module which instantiates the actual top module with all generics tied to specific values
2. Create an amended version of the test bench that instantiates the new top module (without any generics)
3. Run implementation with the new top module as top level entity
4. Create a new simulation environment with the amended test bench as simulation top
5. Run Tcl commands to start logging of signal and run simulation
6. Run power estimation and use the simulation activity file (.SAIF) created during the previous step

The first two steps are necessary because the regular test bench cannot be used with the implemented design. During synthesis and subsequent steps, any generic parameters are replaced with actual values. Trying to instantiate e.g. the implemented design in the regular test bench will cause errors because the test bench will try to instantiate the module with generic parameters. The solution is to create a new top module without generics, and change the test bench to instantiate this module instead.

To actually retrieve the switching activity of the signals in the design, the following sequence of Tcl commands can be run in Vivado when a new simulation environment has been created:

```
open_saif <filename.saif>
log_saif [get_object -r /<test bench instance>/*]
run <time>
close_saif
```

The argument to `log_saif` is a list of signals to capture. The easiest choice here is to just fetch all signals within the design, which is what the `get_object` command does with the `-r` flag. When logging has been set up, the simulation can be run for any duration, but it should run long enough to exercise as much logic as possible so that switching activity for as many signals as possible is recorded.

In the CCSDS123 sources, `tb/synth_top.vhd` contains a top level entity that instantiates the actual CCSDS123 core top with generic parameters tied to specific values given in `tb/synth_params.vhd`. An amended test bench that uses this new top is found in `tb/top_impl_tb.v`.