
## NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

**SCHOOL OF SCIENCE**
**DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS**

**INTERDISCIPLINARY MASTERS PROGRAM IN MICROELECTRONICS**


**MSc THESIS**

# Design and Implementation in FPGA Technology of a High-Performance Block Scan and Map to Symbols Module for CCSDS-122 Image Data Compression


**Maria K. Taipliadou**

**Supervisors:** **Antonios Paschalis,** Professor
**Nektarios Kranitis,** Assistant Professor



**ATHENS**

**JANUARY 2022**


**ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ**

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ**
**ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ**

**ΔΙΑΤΜΗΜΑΤΙΚΟ ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ ΣΤΗ ΜΙΚΡΟΗΛΕΚΤΡΟΝΙΚΗ**


**ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ**

# Σχεδίαση και Υλοποίηση σε Τεχνολογία FPGA μίας Μονάδας BSMS Υψηλής Απόδοσης σύμφωνα με το Διαστημικό Πρότυπο CCSDS-122 Συμπίεσης Δεδομένων Εικόνας


**Μαρία Κ. Ταϊπλιάδου**

***Επιβλέποντες*:** **Αντώνης Πασχάλης,** Καθηγητής
**Νεκτάριος Κρανίτης,** Αναπληρωτής Καθηγητής



**ΑΘΗΝΑ**

**ΙΑΝΟΥΑΡΙΟΣ 2022**

**MSc THESIS**

# Design and Implementation in FPGA Technology of a High-Performance Block Scan and Map to Symbols Module for CCSDS-122 Image Data Compression

**Maria K. Taipliadou**

**S.N.:** MM305

**SUPERVISORS:** **Antonios Paschalis,** Professor
**Nektarios Kranitis,** Assistant Professor

January 2022

# ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Σχεδίαση και Υλοποίηση σε Τεχνολογία FPGA
μίας Μονάδας BSMS Υψηλής Απόδοσης
σύμφωνα με το Διαστημικό Πρότυπο CCSDS-122
Συμπίεσης Δεδομένων Εικόνας

**Μαρία Κ. Ταϊπλιάδου**

**Α.Μ.:** ΜΜ305

**ΕΠΙΒΛΕΠΟΝΤΕΣ:** **Αντώνης Πασχάλης,** Καθηγητής
**Νεκτάριος Κρανίτης,** Αναπληρωτής Καθηγητής

Ιανουάριος 2022

# ABSTRACT

Remote sensing is recognized as a cornerstone monitoring technology. The latest high resolution and high-speed space-borne imagers provide an explosive growth in data volume and instrument data rates in the range of several Gbps. This competes with the limited on-board storage resources and downlink bandwidth, making image data compression a mission-critical on-board processing task.

The Consultative Committee for Space Data Systems (CCSDS) issued in 2005 a recommended standard for Image Data Compression (IDC) (CCSDS-122.0-B-1) which defines a transform-based 2D image data compression algorithm designed specifically for use on-board in a space platform or a payload. An extension of this standard, CCSDS-122.0-B-2, was issued in 2017 to define all necessary modifications to support a recommended standard for Spectral Preprocessing Transform for Multispectral and Hyperspectral Image Compression. The new issue supports images of higher dynamic range and for larger word sizes. Another recommended standard, CCSDS-122.1-B-1, was issued concurrently in 2017 to define the dedicated spectral preprocessing transforms.

In this master thesis is introduced a new high-performance architecture and implementation in FPGA technology for a key-part of the CCSDS-IDC algorithm, the submodule of the Bit Plane Encoder which implements the Block Scan and Map to Symbols process, hereafter termed BSMS, is described. The proposed architecture implementation is based on the standard's existing parallelism, while at the same time introduces new attributes of speed, since it can process one data sample per one clock cycle and thus outperforms previous implementations that required more clock cycles.

**SUBJECT AREA**: Digital Design, FPGA hardware accelerators

**KEYWORDS**: Hardware Accelerator, FPGA, VHDL, Image Data Compression, CCSDS, Bit Plane Encoder

# ΠΕΡΙΛΗΨΗ

Η τηλεπισκόπιση αποτελεί ακρογωνιαίο λίθο των σύγχρονων τεχνολογιών παρατήρησης. Τα σύγχρονα διαστημικά οπτικά όργανα απεικόνισης υψηλής ανάλυσης και υψηλής ταχύτητας οδηγούν σε εκρηκτική αύξηση του όγκου δεδομένων και επιβάλλουν ρυθμούς δεδομένων της τάξης των αρκετών Gbps. Αυτό έρχεται σε αντίθεση με τους περιορισμένους πόρους αποθήκευσης δεδομένων εν πτήσει και το περιορισμένο εύρος ζώνης κατερχόμενης ζεύξης, καθιστώντας την συμπίεση δεδομένων εικόνας μια βασική υποστηρικτική τεχνολογία επεξεργασίας δεδομένων εν πτήσει.

Η Συμβουλευτική Επιτροπή για Συστήματα Διαστημικών Δεδομένων (CCSDS) εξέδωσε το 2005 ένα συνιστώμενο πρότυπο για τη συμπίεση δεδομένων εικόνας (Image Data Compression – IDC) (CCSDS-122.0-B-1), το οποίο ορίζει έναν αλγόριθμο συμπίεσης δεδομένων 2D εικόνας που βασίζεται σε μετασχηματισμό, σχεδιασμένο ειδικά για χρήση εν πτήσει σε διαστημική πλατφόρμα ή ωφέλιμο φορτίο. Μια επέκταση αυτού του προτύπου, CCSDS-122.0-B-2, εκδόθηκε το 2017 για να οριστούν όλες οι απαραίτητες τροποποιήσεις για την υποστήριξη ενός συνιστώμενου προτύπου για τον μετασχηματισμό φασματικής προεπεξεργασίας για πολυφασματική και υπερφασματική συμπίεση εικόνας. Η δεύτερη έκδοση υποστηρίζει εικόνες υψηλότερου δυναμικού εύρους και για μεγαλύτερα μεγέθη λέξεων. Ένα άλλο συνιστώμενο πρότυπο, το CCSDS-122.1-B-1, εκδόθηκε ταυτόχρονα το 2017 για τον καθορισμό των αποκλειστικών φασματικών μετασχηματισμών προεπεξεργασίας.

Στην παρούσα διπλωματική εργασία, εισάγεται μια νέα αρχιτεκτονική υψηλής απόδοσης και η αντίστοιχη υλοποίησή της σε τεχνολογία FPGA μίας υπομονάδας κλειδί του αλγορίθμου CCSDS-IDC, της υπομονάδας του Bit Plane Encoder που πραγματοποιεί τη διαδικασία Block Scan and Map to Symbols, που στην συνέχεια θα ονομάζουμε μονάδα BSMS. Η νέα υλοποίηση βασίζεται επίσης στην εκμετάλλευση της παραλληλίας του προτεινόμενου αλγορίθμου, ενώ ταυτόχρονα επιτυγχάνει την επεξεργασία ενός δείγματος δεδομένων ανά κύκλο.

**ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ**: Ψηφιακή Σχεδίαση, Επιταχυντές υλικού σε FPGA

**ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ**: Επιταχυντής υλικού, FPGA, VHDL, συμπίεση εικόνας, CCSDS, Bit Plane Encoder

*Αφιερωμένη στον πατέρα μου Κωνσταντίνο Ταϊπλιάδη*

*που χωρίς εκείνον δεν θα ήμουν εδώ σήμερα.*

# ΕΥΧΑΡΙΣΤΙΕΣ

Θα ήθελα να ευχαριστήσω τον κ. Αντώνη Πασχάλη, καθηγητή μου στο Μεταπτυχιακό και μετέπειτα επιβλέποντα καθηγητή μου στην Διπλωματική εργασία, που μου έδωσε την ευκαιρία να συμμετάσχω σε ένα ερευνητικό project και με εμπιστεύτηκε με ένα κρίσιμο κομμάτι αυτού. Ακόμα, θέλω να ευχαριστήσω τον σύντροφο μου Γιάννη που με την στάση του και τις πολύτιμες συμβουλές του μου έδωσε τη στήριξη που χρειαζόμουν ώστε να ολοκληρώσω την παρούσα δουλειά. Τέλος θέλω να ευχαριστήσω τους δύο πυλώνες της ζωής μου, τους γονείς μου Κωνσταντίνο και Περιστέρα, που μου επέτρεψαν να ανοίξω τα φτερά μου και να πετάξω, πάντα ελεύθερη να ορίζω την κατεύθυνση και προσφέροντας παράλληλα απλόχερα τη στήριξή τους σε όλο το δύσκολο αυτόν δρόμο.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# PREFACE

This thesis was conducted at the Digital Systems and Computer Architecture Laboratory (DSCAL) of the Department of Informatics and Telecommunications, of the National and Kapodistrian University of Athens (NKUA). It was carried out within the Space Technology Group in the context of the research project SISYFOS.

This research has been co-financed by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH – CREATE – INNOVATE (project code: T1EDK-04298).

# 1. INTRODUCTION

The huge amounts of data generated from latest and future high-resolution, high-speed imagers in Earth Observation (EO) satellite missions, make image data compression one of the most challenging on-board payload data processing tasks [1]. On-board data compression is the key to overcome the telemetry rates bottleneck and hardware implementation of on-board data compression is the key to address the data-rate challenges of today's and future remote sensing payloads. On-board processing of payload data is a challenging task since data-rates and data volumes produced by remote sensing payloads increase while the available downlink bandwidth is comparatively stable [2].
Source coding for data compression is a method utilized in data systems to reduce the volume of digital data to achieve benefits in areas including, but not limited to,

a) reduction of transmission channel bandwidth

b) reduction of the buffering and storage requirement

c) reduction of data-transmission time at a given rate.

In 2005, the Consultative Committee for Space Data Systems (CCSDS) issued the Image Data Compression (IDC) standard CCSDS 122.0-B.1 [3]. CCSDS-IDC defines a particular transform-based image data compression algorithm applicable to many types of spaceborne instrument payloads. The recommended standard provides both lossless and lossy (both rate and quality limited) compression, suitable for monoband two-dimensional (2D) images.
This Recommended Standard addresses image data compression, which is applicable to a wide range of space-borne digital data, where the requirement is for a scalable data reduction, including the option to use lossy compression, which allows some loss of fidelity in the process of data compression and decompression and provides a compression method that ensures that the distortion in the reconstructed image does not exceed user-specified limits.

This Recommended Standard applies to data compression applications of space missions anticipating packetized telemetry cross support. In addition, it serves as a guideline for the development of compatible CCSDS Agency standards in this field, based on good engineering practice.

The purpose of this Thesis is to go one step further than the existing implementation of the established Recommended Standard for the image data compression algorithm. The submodule that was implemented for this project can be applied for either two-dimensional digital spatial image or and digital three-dimensional image data from payload instruments, such as multispectral and hyperspectral imagers.

This thesis is organized as follows: In Chapter 2, a brief description of the CCSDSIDC algorithm is presented along with the focus of this thesis, the Bit Plane Encoder. Related work is also presented. In Chapter 3, the proposed architecture for the BSMS is introduced. The verification and validation strategy of the BSMS design is described in Chapter 4. Experimental implementation results and comparisons are provided in Chapter 5, while Chapter 6 concludes the thesis.

# 2. CCSDS STANDARD OVERVIEW AND RELATED WORK

## 2.1 Compressor overview

This work is an implementation of the CCSDS 122.0-B-2 standard for Image Data Compression [4]. This standard proposes the Bit Plane Encoder as the encoder that precedes the Entropy encoding process. The Bit Plane Encoder can be used for both simple (2D) and Multispectral/Hyperspectral (3D) image compression.

The standard for Multispectral/Hyperspectral (3D) image compression extends the (two-dimensional) CCSDS Image Data Compression standard by providing an effective method of encoding three-dimensional image data. The input to the compressor is a three-dimensional image that has signed or unsigned integer sample values [5]. The compressed image output from the compressor is an encoded bitstream from which an exact or approximate reconstruction of the input image can be recovered.
The compressor consists of two main functional parts, depicted in figure 2.1: a spectral transform, and a set of 2D encoders.

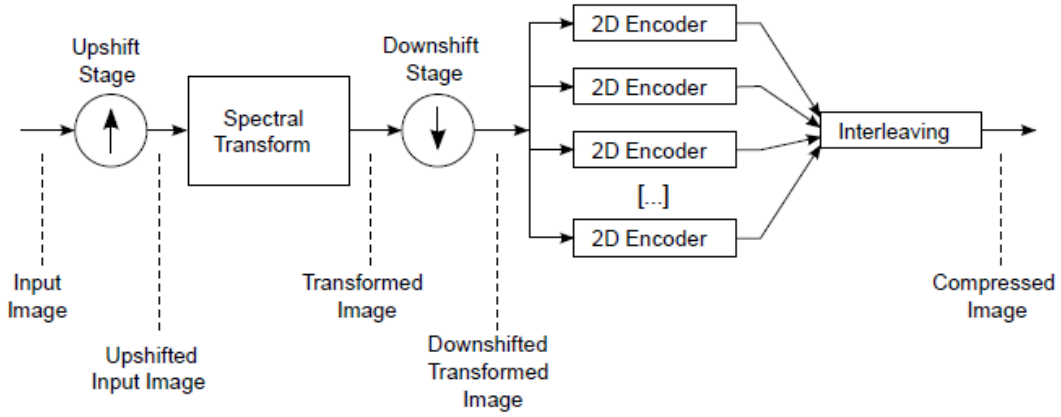


**Figure 2.1: Block Diagram of the Compressor for 3D Images**

The purpose of the spectral transform is to exploit the similarities between the spectral bands of an image, creating a transformed image that it is more efficiently compressed by the 2D encoders. Each transformed band is independently compressed by a 2D encoder. In practice, an implementation of the 2D encoder may be reused multiple times to accomplish this task. Two additional minor functional stages are also included, named *upshift* stage and *downshift* stage, with the purpose of adapting the bit depth before each of the two main functional parts.
The compressed image, consists of a header that encodes image and compression parameters followed by a body that is produced by an entropy coder, which losslessly encodes the mapped quantizer indices.

A standard specifically for multispectral and hyperspectral (three-dimensional) lossless image compression has not been developed. Rather than developing a new standard, an existing two-dimensional image compression standard is used. Candidates include the wavelet-based image compression standards JPEG2000 and CCSDS 122.0-B-2, and the predictive-based JPEG-LS standard, all of which are capable of providing lossless image compression. For this project the CCSDS 122.0-B-2 standard was used.

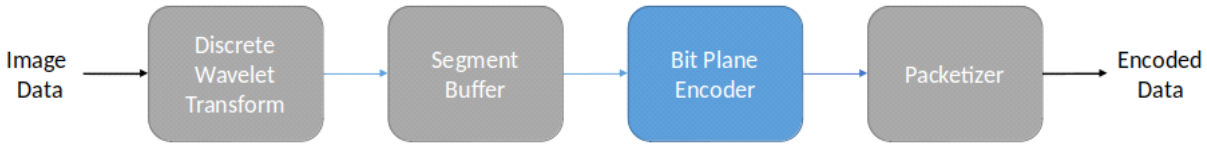


**Figure 2.2: Block Diagram of the Compressor for 2D Images**

The CCSDS 122.0-B-2 Image Data Compression (CCSDS-IDC) algorithm consists of two functional parts: a) a Discrete Wavelet Transform (DWT) that performs decorrelation and b) a Bit Plane Encoder (BPE) which encodes the decorrelated data [4].

### 2.1.1 Bit Plane Encoder overview

Following the DWT, the Bit Plane Encoder (BPE) processes wavelet coefficients in groups of 64 coefficients referred to as *blocks*. An example of a block is illustrated in Figure 2.3 as comprised of shaded pixels. A block loosely corresponds to a localized region in the original image.

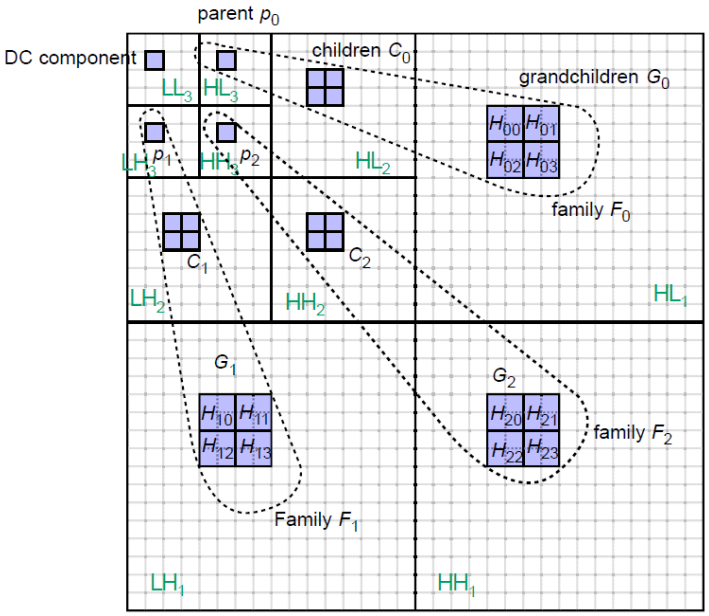


**Figure 2.3: Schematic of Wavelet-Transformed Image**

Information pertaining to a block of coefficients is jointly encoded by the BPE. A block consists of a single coefficient from the $LL_3$ subband, referred to as the *DC coefficient*, and 63 *AC coefficients*. The AC coefficients in a block are arranged into three *families*,$F_0$,$F_1$ and $F_2$. Figure 2.3 illustrates a single block of coefficients and the family structure.

Each family $F_i$ in the block has one *parent* coefficient, $p_i$, a set $C_i$ of four *children* coefficients, and a set $G_i$ of sixteen *grandchildren* coefficients. The grandchildren in family $F_i$ are further partitioned into groups numbered $j$=0,1,2,3, denoted $H_{ij}$, as illustrated in Figure 2.3. This structure is used for jointly encoding information pertaining to groups of coefficients in the block.

A wavelet coefficient is identified by its coordinates within its subband. Thus coordinates ($r$, $c$) indicate the wavelet coefficient in row $r$, column $c$ within the subband, with the upper left pixel in a subband having coordinates (0,0).
The DC coefficient for each block is a single coefficient from the LL3 subband. The coordinates for the other coefficients in the block can be determined from the coordinates of the DC coefficient. For a block with DC coefficient with coordinates ($r$, $c$) within the LL3 subband, Table 2-1 lists the coordinates for the AC coefficients, within their respective subbands of origin.

**Table 2.1: Within-Subband Coordinates for Coefficients in a Single Family**

| Coefficient Group in Family $i$ | Coordinates |
|---|---|
| Parent, $p_i$ | $(r,c)$ |
| Children group, $C_i$ | $(2r, 2c)$, $(2r, 2c+1)$, $(2r+1, 2c)$, $(2r+1, 2c+1)$ |
| Grandchildren group, $H_{i0}$ | $(4r, 4c)$, $(4r, 4c+1)$, $(4r+1, 4c)$, $(4r+1, 4c+1)$ |
| Grandchildren group, $H_{i1}$ | $(4r, 4c+2)$, $(4r, 4c+3)$, $(4r+1, 4c+2)$, $(4r+1, 4c+3)$ |
| Grandchildren group, $H_{i2}$ | $(4r+2, 4c)$, $(4r+2, 4c+1)$, $(4r+3, 4c)$, $(4r+3, 4c+1)$ |
| Grandchildren group, $H_{i3}$ | $(4r+2, 4c+2)$, $(4r+2, 4c+3)$, $(4r+3, 4c+2)$, $(4r+3, 4c+3)$ |

**Table 2.2: Subband of Origin for AC Coefficients**

| | Family 0 | Family 1 | Family 2 |
|---|---|---|---|
| Parent | $HL_3$ | $LH_3$ | $HH_3$ |
| Children | $HL_2$ | $LH_2$ | $HH_2$ |
| Grandchildren | $HL_1$ | $LH_1$ | $HH_1$ |

Blocks shall be processed by the Bit Plane Encoder consecutively in the raster scan in the order in which their corresponding DC coefficients occur in $LL_3$: row by row, each row being processed from left to right.

A *segment* is defined as a group of *S* consecutive blocks. Coding of DWT coefficients proceeds segment-by-segment and each segment is coded independently of the others.

A segment of blocks is further partitioned into *gaggles*. Each gaggle consists of 16 blocks, except for possibly the last gaggle in a segment, which contains *S* mod 16 blocks when *S*is not a multiple of 16.

An AC coefficient is represented using the binary representation of the magnitude of the coefficient, along with a bit indicating the sign when the coefficient is nonzero.

BitDepthAC_Block$_m$ denotes the maximum number of bits needed to specify the magnitude of any AC coefficient in the $m_{th}$ block. For each segment, the BPE computes BitDepthAC, which denotes the maximum value of BitDepthAC_Block$_m$ for the segment.

The BPE successively encodes bit planes of coefficient magnitudes in a segment, inserting AC coefficient sign values at appropriate points in the coded segment data stream. *Bit plane b* consists of the $b_{th}$ bit of the two's-complement integer representation of each DC coefficient, and the $b_{th}$ bit of the binary integer representation of the magnitude of each AC coefficient. Here, bit plane index $b$=0 corresponds to the least significant bit. The BPE proceeds from most-significant bit to least significant bit, thus *b* decreases from one bit plane to the next, beginning with $b$= BitDepthAC-1, and ending with $b$=0.

The structure of a *coded segment* is shown in Figure 2.4(a). Within a coded segment, header information is encoded. Then quantized DC coefficients from the blocks are encoded. Then AC bit depths are encoded. Then DWT coefficient blocks are encoded, one bit plane at a time, proceeding from the most significant to the least significant bit plane. The coding of a single bit plane is performed in several stages, and the resulting order of encoded data is illustrated in Figure 2.4(b). E.g., parent coefficients are coded in stage 1 for all blocks of the segment before encoding child coefficients in stage 2. The resulting encoded bit stream constitutes an embedded data format that provides progressive transmission.

**(a) With All Bit Planes**

| |
|---|
| Segment Header (see 4.2) |
| Initial coding of DC coefficients (see 4.3) |
| Coded AC coefficient bit depths (see 4.4) |
| Coded bit plane $b$ = BitDepthAC-1 (see 4.5) |
| Coded bit plane $b$ = BitDepthAC-2 (see 4.5) |
| ⋮ |
| Coded bit plane $b$ = 0 (see 4.5) |

**(b) Within One Bit Plane**

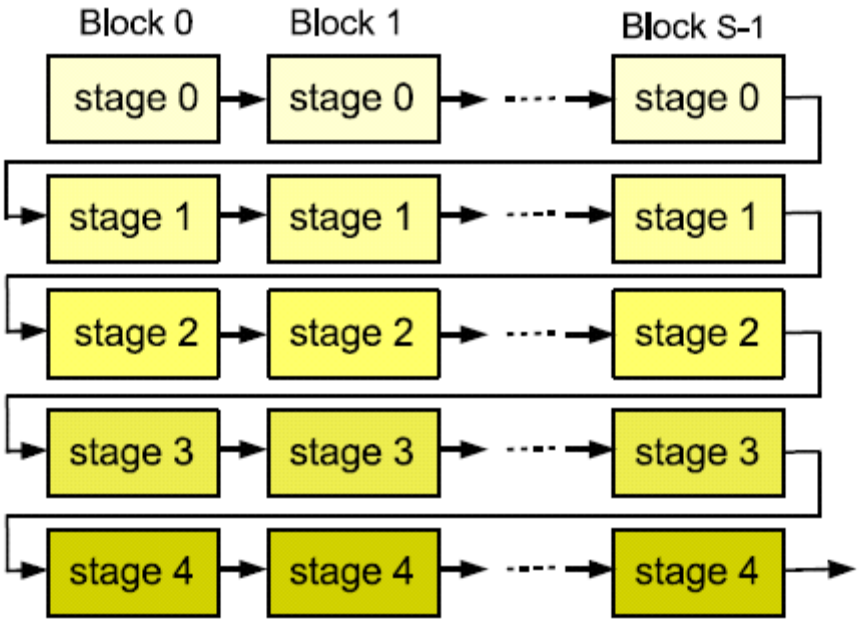


**Figure 2.4: Overview of the Structure of a Coded Segment**

The Bit Plane Encoder, for the AC coefficients coding, mainly consists of three processes. First the scan of a block takes place which results in a sequence of words. The words are generated from the bitplane bits, starting with bitplane *b*and proceeds per bitplane until bitplane 0 is reached. Afterwards, these words are mapped to variable length binary words, called symbols, according to length. Next, a subset of these words is further entropy coded using variable-length binary codes, again according to length. The first two processes form the BSMS (Block Scan and Map to Symbols).

#### 2.1.1.1 Block Scan and Map to Symbols

Coding of a bit plane is performed in *stages* numbered 0-4. The coded bits produced at the stages for each block are interleaved, as illustrated in Figure 2.4(b) and Figure 2.5. Thus, a coded bit plane first consists of all the stage 0 bits (if any) in the segment, then all of the coded stage 1 bits in the segment, and so on, finishing with all of the encoded stage 4 bits in the segment. This produces an embedded bit string with information from the highest bit plane of all *S* blocks in the first part of the output bit string followed by information from lower bit planes, and allows progressive decoding of the coded string. This improves image reconstruction quality when the coded bit sequence is truncated.

| |
|---|
| Stage 0 bits from each block in the segment (if any) |
| Coded stage 1 from block 0, 1, …, *S*-1 |
| Coded stage 2 from block 0, 1, …, *S*-1 |
| Coded stage 3 from block 0, 1, …, *S*-1 |
| Coded stage 4 from block 0, 1, …, *S*-1 |

**Figure 2.5: Coded Bit Plane Structure for a Coded Segment**

**Stage 0** for a block consists of at most a single bit, which is simply the $b_{th}$ most significant bit of the two's-complement representation of the DC coefficient.

The remaining stages (1-4) encode AC coefficient bits. The stage in which bits from AC coefficients in a bit plane are coded depends on the *type* of the AC coefficient at the bit plane, which we now define. At bit plane *b*, the type of an AC coefficient *x*, denoted $t_b(x)$, has one of the following values:

–$t_b(x) = 0$ if $|x| < 2^b$, (*x* is not due for selection at this bit plane);

–$t_b(x) = 1$ if $2^b \leq |x| < 2^{b+1}$, (*x* is due for selection at this bit plane);

–$t_b(x) = 2$ if $2^{b+1} \leq |x|$, (*x* has already been selected at a previous bit plane);

–$t_b(x) = -1$ if $b <$ BitShift(Γ), (*x* must be zero at this bit plane due to subband scaling).

Here, Γ denotes the subband containing *x*. Thus, during bit-plane encoding, each AC coefficient typically proceeds from type 0 to 1, to 2, to -1. For a set of coefficients Ψ, we define tmax(Ψ) as the maximum of the coefficient types in Ψ.

An AC coefficient *x* is said to be *selected* at bit plane *b* if $t_b(x) = 1$. I.e., the 'selection' of a coefficient marks the first bit plane where a non-zero magnitude bit is encoded for the coefficient. Note that $t_b(x) = 1$ if the $b_{th}$ most significant magnitude bit of *x* is equal to '1' and all more significant magnitude bits of *x* are equal to '0'.

The type of a coefficient determines the stage when coding of a coefficient bit takes place. When an AC coefficient *x* is of type 0 or 1 (implying $t_{b+1}(x)=0$), the $b_{th}$

most significant magnitude bit of $x$ is coded in stages 1-3. Otherwise, the bit is included, uncompressed, in stage 4 if $x$ is of type 2, or not encoded at all when $x$ is of type -1.

In **stages 1-3** of BPE encoding bit plane $b$, the $b_{th}$ magnitude bit of each AC coefficient $x$ such that $t_{b+1}(x)=0$ is encoded. The $b_{th}$ magnitude bits of the parent coefficients are coded in stage 1, the children in stage 2, and the grandchildren in stage 3. Each of these stages also includes coded bits indicating the sign of each coefficient $x$ for which $t_b(x)=1$. The coding in stages 1-3 makes use of the family structure to group together AC coefficients for entropy coding.

The coding performed in stages 1-3 for a block consists of two parts. First, a sequence of variable length binary words are defined which completely describe the bits to be encoded in these stages. Next, a subset of these words are further entropy coded.

**Stage 4** of coding consists of the $b_{th}$ magnitude bit of each AC coefficient $x$ with $t_b(x)=2$. These bits are included in the coded data stream uncompressed.

#### 2.1.1.1.1 AC coefficient words coding stages 1-3

The bits encoded in stages 1-3 for a block can be determined by a sequence of words, as described below.

In addition to the sets $C_i$, $G_i$, $H_{ij}$, $P$ is defined as the list of parents in the block:

$$P = \{p_0, p_1, p_2\}.$$

The list of descendants in family $i$, denoted $D_i$, is defined as

$$D_i = \{C_i, G_i\}.$$

The list of descendants in a block, denoted $B$, is defined as

$$B = \{D_0, D_1, D_2\}.$$

$\{A, B\}$ denotes the concatenation of the lists $A$ and $B$.

A shorthand notation for certain binary words that describe information about bit plane $b$ for a list of coefficients $\Psi$ is defined as follows:

–let $types_b[\Psi]$ denote the binary word consisting of the $b_{th}$ magnitude bit of each coefficient $x$ in $\Psi$ such that $t_b(x)$ equals 0 or 1.

–let $signs_b(\Psi)$ denote the binary word consisting of the sign bit of each coefficient $x$ in $\Psi$ such that $t_b(x) = 1$, with a sign bit of ‘1’ for negative coefficients and ‘0’ for nonnegative coefficients.

–given a list of type values $\Lambda=\{\lambda_0,\lambda_1,\lambda_2, \ldots, \lambda_l\}$, let $tword[\Lambda]$ denote the binary word consisting of the sequence of type values $\lambda_i$ in $\Lambda$ that are equal to 0 or 1.

Any of these words can be null (i.e., have length zero).

The list $P$ shall be ordered $P = \{p_0, p_1, p_2\}$, while the ordering on the lists $C_i$ and $H_{ij}$ shall be determined by the order in which their member coefficients' coordinates are listed in Table 2.1.

The $b_{th}$ magnitude bits for all AC coefficients that are type 0 at bit plane $b$+1 (i.e., not selected before the current bit plane) shall be communicated to the decoder by joining them to form binary words associated with each data type (parent, child, grandchild):

–$types_b[P]$

–$types_b[C_i]$ for $i$= 0, 1, 2; and

–$types_b[H_{ij}]$ for $i$= 0, 1, 2, $j$= 0, 1, 2, 3.

At early bit planes, many sets of coefficients in a block tend to all be of type 0, and thus many of these words are initially all zeros. To effectively encode in this situation, the BPE shall make use of the following *transition* words to indicate when groups of coefficients at a lower depth are all Type 0:

–$tran_B$ = null, if $tran_B$= 1 at any more significant bit plane, $tword[\{t_{max}(B)\}]$, otherwise.

–$tran_D$ = $tword[\{t_{max}(D_i)$ : $i$=0,1,2, such that $t_{max}(D_i) \neq 1$ in all more significant bit planes$\}]$.

–$tran_G$ = $tword[\{t_{max}(G_i)$ : $i$=0, 1, 2, such that $t_{max}(D_i)>0$ in current or any more significant bit planes $\}]$.

–$tran_{Hi}$ = $tword[\{t_{max}(H_{i0}), t_{max}(H_{i1}), t_{max}(H_{i2}), t_{max}(H_{i3})\}]$ for $i$= 0,1,2.

At bit plane $b$, the BPE shall use the following sequence of words, generated in three stages, to update all of the AC coefficients in the block that were Type 0 at the previous bit plane:

a)**Stage 1 (parents)**:

$types_b[P]$,$signs_b[P]$.

b)**Stage 2 (children)**:

1)$tran_B$

2)$tran_D$, if $tran_B \neq 0$ and $t_{max}(B) \neq -1$

3)$types_b[C_i]$ and $signs_b[C_i]$ for each $i$ such that $t_{max}(D_i)>0$ in current or any more significant bit planes.

c)**Stage 3 (grandchildren)**:

If $tran_B$ = 0 or $t_{max}(B)$ = -1, then stage 3 is unnecessary and shall be omitted.

Otherwise stage 3 consists of:

1)$tran_G$

2)$tran_{Hi}$, for each $i$ such that $t_{max}(G_i) \neq 0, -1$

3)$types_b[H_{ij}]$ and $signs_b[H_{ij}]$ for each $i$ such that $t_{max}(G_i)\neq 0$, -1 and each $j$ such that $t_{max}(H_{ij})\neq 0$, -1.

All of the words generated in the above stages are variable length (including the null word).

Words $types_b[P]$, $types_b[C_i]$, $types_b[H_{ij}]$, $tran_D$, $tran_G$, $tran_{Hi}$ shall be entropy coded, i.e., each shall be replaced with a corresponding variable-length codeword, whenever such a word has a length of at least 2 bits.

The sign bit words are not coded further, because AC coefficients are generally positive and negative with about equal probability. The $tran_B$ word is always, at most, one bit in length and is never entropy coded.

**Stage 4 Coding**

In stage 4 of coding, the $b_{th}$ magnitude bit of each AC coefficient $x$ with type $t_b(x)=2$ shall be included in the output bit stream.

For each block, the output bit string shall consist of the $b_{th}$ magnitude bit of type 2 coefficients, in the following order:

–$p_i$, for each $i$= 0,1,2

–members of $C_i$, for each $i$= 0,1,2

–members of $H_{ij}$, for each $i$= 0,1,2, and each $j$= 0,1,2,3.

Members of the sets $C_i$ and $H_{ij}$ shall be processed in the order listed in Table 2.1.

No bits shall be coded in stage 4 for AC coefficients $x$ not of type 2 ($t_b(x) \neq 2$).

The resulting strings for all blocks in the segment shall be concatenated to produce the entire stage 4 output string for the coded segment.

### 2.1.1.1.2 Mapping words to symbols

The entropy coding procedure used to encode the words $types_b[P]$, $types_b[Ci]$, $types_b[Hij]$, $tran_D$, $tran_G$, $tran_{Hi}$ shall be accomplished through the use of variable-length codes. Words having a length of one bit, and sign-bit words, shall be included in the compressed data stream without further coding. Words of length greater than one bit shall be coded in the sequence in which they occur within each stage with entropy coding.

Certain bit sequences cannot appear as values for certain words and this fact is taken into account in the entropy coding process. For example, $tran_D$ can never equal 000, because this condition would be inferred from the fact that $tran_B$=0. Table 2.3 summarizes the maximum word lengths and impossible values for each word that is entropy coded.

**Table 2.3: Summary of Maximum Word Lengths and Impossible Word Values**

| Word | Maximum Length (bits) | Impossible Value |
|---|---|---|
| $types_b[P]$ | 3 | – |
| $types_b[C_i]$ | 4 | – |
| $types_b[H_{ij}]$ | 4 | 0000 |
| $tran_D$ | 3 | 000 |
| $tran_G$ | 3 | – |
| $tran_{Hi}$ | 4 | 0000 |

The process of variable-length coding of these words shall follow a two-step process:

–first, the word values shall be mapped to integer values referred to as *symbols* and then

–each integer shall be encoded using a variable-length binary codeword.

Under the mapping, two-bit, three-bit, and four-bit words shall be mapped to symbols using table 2.4, 2.5, or 2.6, respectively.

The mapping process takes into account the fact that certain words can never be assigned certain bit sequences, as tabulated in table 2.3 and this is reflected in tables 2.5 and 2.6.

The entropy encoding, which is the second step, follows the outputs of the BSMS design, since the BSMS concludes with the mapping to symbols process.

**Table 2.4: Integer Mapping for Two-Bit Words**

| Word | Symbol |
|---|---|
| 00 | 0 |
| 01 | 2 |
| 10 | 1 |
| 11 | 3 |

**Table 2.5: Integer Mapping for Three-Bit Words**

| Word | Symbol ($types_b[P]$, $types_b[C_i]$, $types_b[H_{ij}]$, $tran_G$, $tran_{Hi}$) | Symbol ($tran_D$) |
|---|---|---|
| 000 | 1 | - |
| 001 | 4 | 3 |
| 010 | 0 | 0 |
| 011 | 5 | 4 |
| 100 | 2 | 1 |
| 101 | 6 | 5 |
| 110 | 3 | 2 |
| 111 | 7 | 6 |

**Table 2.6: Integer Mapping for Four-Bit Words**

| Word | Symbol ($types_b[C_i]$) | Symbol ($types_b[H_{ij}]$, $tran_{Hi}$) |
|---|---|---|
| 0000 | 10 | - |
| 0001 | 1 | 1 |
| 0010 | 3 | 3 |
| 0011 | 6 | 6 |
| 0100 | 2 | 2 |
| 0101 | 5 | 5 |
| 0110 | 9 | 9 |
| 0111 | 12 | 11 |
| 1000 | 0 | 0 |
| 1001 | 8 | 8 |
| 1010 | 7 | 7 |
| 1011 | 13 | 12 |
| 1100 | 4 | 4 |
| 1101 | 14 | 13 |
| 1110 | 11 | 10 |
| 1111 | 15 | 14 |

The mappings are intended to produce symbol values in order of decreasing frequency. (I.e., the most frequently occurring word is mapped to symbol value 0, the next most frequent to 1, etc.) This makes effective coding possible through the entropy encoding procedure that follows, because the codewords are arranged in order of increasing length.

## 2.2 Related work

A Reconfigurable FPGA Implementation of CCSDS 122.0-B-1 Image Data Compression took place at Digital Systems & Computer Architecture Laboratory (DSCAL) on 2014. When the existing implementation took place, the goal was to implement the CCSDS's proposed algorithm for Image Data Compression, with a Hardware Description Language, in this case VHDL. Functionality was the main requirement and as a result the design was slow in terms of clock frequency.

The previous implementation was based on an FSMD design. That is a finite-state machine, which controls the program flow along with the datapath inside each state, while performing bitplane processing operations. FSMDs are essentially sequential programs in which statements have been scheduled into states, thus resulting in more complex state diagrams and more complex design altogether.

In addition to the low clock frequency, the FSMD of this implementation was configured in such a way that for each input sample or for each output the design needed two clock cycles, thus resulting to a very slow data flow. So a need for a faster design occurred. For this thesis a new architecture was developed that achieves both clock frequency increase and also the need for just one clock cycle for each sample.

This work was published at the On-Board Payload Data Compression Workshop (OBPDC 2014) [1].

# 3. PROPOSED ARCHITECTURE

## 3.1 Architecture of the BSMS

The implementation of the CCSDS 122.0-B-2 proposed algorithm for lossless image data compression, that took place in this project is based on a three-process architecture. There's one synchronous process, where all the registers are defined and two asynchronous processes, one that consists of the word generation logic and one for the control signals.

The components of the BSMS design are a DRAM used as a history table, two encoding-mapping modules and one component for the IDs assignment for each symbol generated.

The BSMS module receives its inputs from the Segment Bitplane Buffer. The Segment Bitplane Buffer unit implements a transformation of the block-oriented data organization of DWT coefficients to a bit plane-oriented data organization and provides BPE with a high-performance access to all the bitplanes of a segment. The BSMS module fetches an already formatted bitplane for all the S blocks of the segment concurrently from the SBB unit in one BRAM access.

As it is proposed in the standard, there are two different indicators for the number of bitplanes that need to be processed. The BitDepthAC and the BitDepthAC_blockm. The first refers to the maximum number of bitplanes for the whole segment and the second to the maximum number of bitplanes for each block. The BSMS module processes one bitplane at a time starting with b=BitDepthAC and ending with b=1. It begins with the b=BitDepthAC bitplane for the first block of the segment, then moves to the b=BitDepthAC bitplane for the second block and continues until the b=BitDepthAC bitplane for the last (S) block in the segment. Afterwards it moves on to the next biplane (b=BitDepthAC-1), again for all the blocks in the segment, and the procedure ends at bitplane b=1 and block=S. For every bitplane processed, when the current bitplane b is larger than BitDepthAC_blockm, the word generation procedure is omitted, as there is nothing to be encoded for the current block.

Once the bitplane, signs and BitDepthAC_blockm have been obtained from the segment buffer, there's one last thing needed so that the generation of the binary words can begin. Due to the sequence described above, in which the bitplanes have to be processed, and because of the dependencies that occur from a bitplane to the next in the same block, a memory is introduced into the design. This memory is necessary so that every time the BSMS moves on to the next bitplane, it can have access to all the needed information from the previous bitplane for the encoding of the current bitplane for the same block.

The proposed description for the types and transition words generation is translated into gate logic with a mask-based implementation. Next, the words and signs along with their corresponding masks go through the encoding and mapping components. There is one component for three-bit words (tranD, tranG, P) and one for four-bit words (tranH0, tranH1, tranH2, C0, C1, C2, H00, H01, …, H23). First a decision is made, according to the corresponding masks, about

which bits are useful in the incoming words and then the useful bits are left shifted, as most significant bits. This procedure, along with the derivation of the word length, again according to mask, is the first step for the encoding process with second being the mapping into symbols. The outputs of these components, symbols, signs and their lengths, compose accordingly each output stage.

The length values of the transition and type words are then used as inputs for the IDs generation component. There is one component instantiation for two bit words, one for three and one for four. The outputs of these components go through some combinatorial logic, from which another output of the BSMS is derived (QTD_out) every time the end of a gaggle is reached. QTD_out exits the BSMS module along with the stages outputs.

The control part of the design is implemented with a Finite State Machine. The need for an FSM design occurred when additional cases where introduced in the design to ensure a valid execution. The encoding part of the algorithm is done in one state and additional states are used, before and after the encoding, to check if the encoding can begin, to catch up with the arrival of valid data at the beginning and to insert some needed flags between output data, to denote when a whole bitplane has been processed for all the blocks in a segment, in the output FIFOs and lastly a state for when the encoding is finished.

The submodule of BSMS was developed to be integrated into the existing 2D encoder as a Plug-and-Play module. So the original block's ports are used. The ports' name and their function are presented in Table 3.1 and the block of the BSMS is depicted in Figure 3.1.

**Table 3.1: Ports' name and function for the BSMS module**

| Port Name | Port Function |
|---|---|
| **Inputs** | |
| clk | Clock signal |
| reset | Asynchronous reset |
| srst | Synchronous reset |
| en_AC | Enable signal to start encoding |
| double_buf | Address indicator for Segment buffer |
| seg_id | Segment ID |
| S | Number of blocks in current segment |
| BitDepthAC | Max number of bitplanes in current segment |
| AC_Bit_depth_out_ram | Max number of bitplanes in current block |
| bp_seg_buff_dout | Bitplane data from Segment buffer |
| signs_seg_buff_dout | Signs data from Segment buffer |
| stage1_full, stage2_full, stage3_full, stage4_full, QTD_full | FIFO is full signals from the output FIFOs |
| **Outputs** | |
| bp_seg_buff_addr_bpe | Bitplane address for the Segment buffer |
| signs_seg_buff_addr_bpe | Signs address for the Segment buffer |
| addr_p_bpe | BitDepthAC_Blockm address for the Segment Buffer |
| stage1out, stage2out, stage3out, stage4out, QTD_out | BSMS outputs |
| stage1_we, stage2_we, stage3_we, stage4_we, QTD_we | Write in FIFO signals for the output FIFOs |

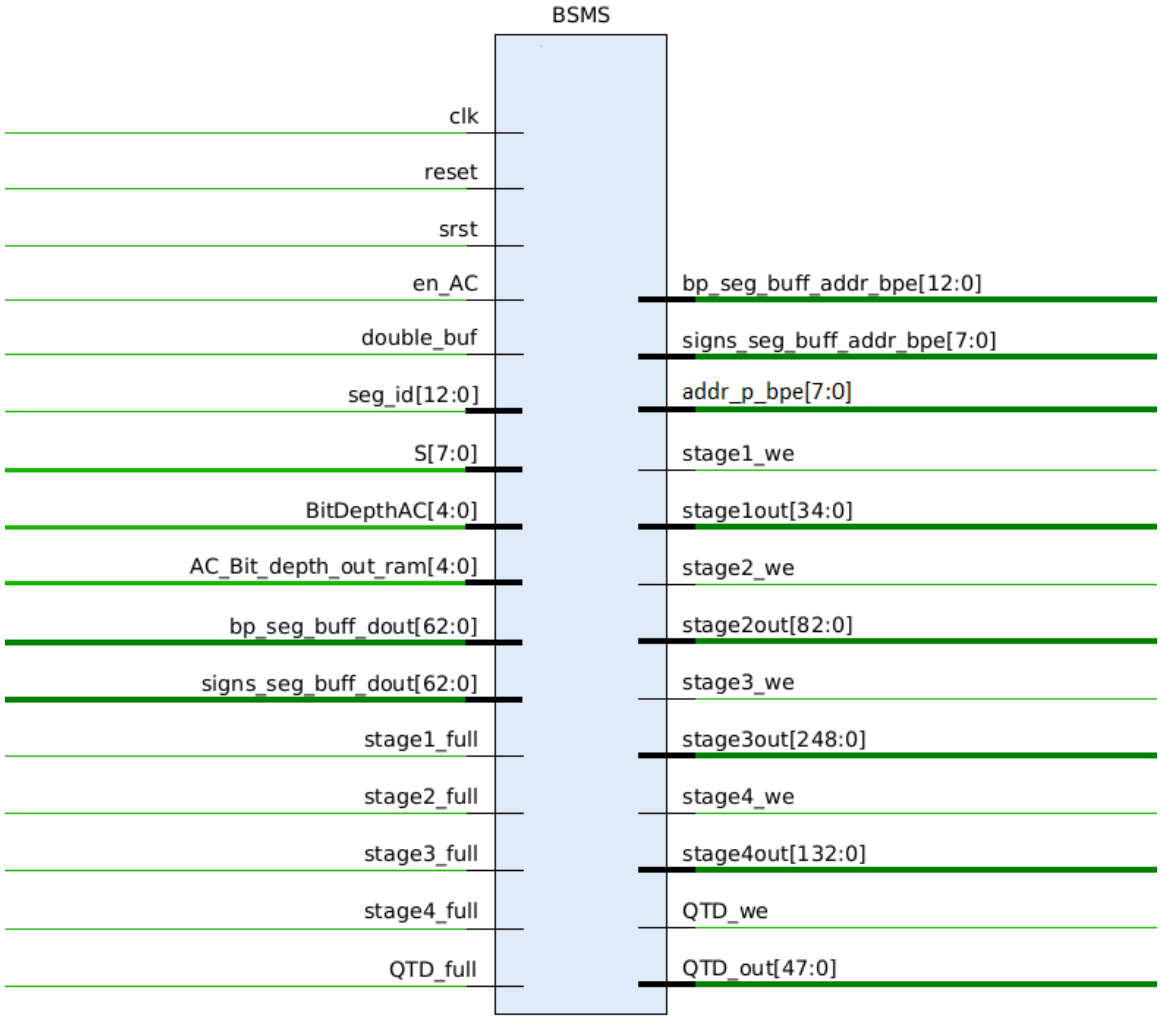


**Figure 3.1: Block and ports of the BSMS module**

## 3.2 Pipeline architecture

The first requirement for the new implementation was that the processing of each bitplane should be completed in one clock cycle, unlike the existing implementation in which the same procedure was done in two clock cycles. That was taken into consideration from the beginning of the design development.

When that was succeeded and the functionality was verified, the next requirement was for the clock frequency to increase. This was achieved by introducing some pipelining into the design.

In Figures 3.1 and 3.2 the architecture of the BSMS before the pipelining is presented and the architecture of the BSMS after the pipelining is presented in Figures 3.3 and 3.4.

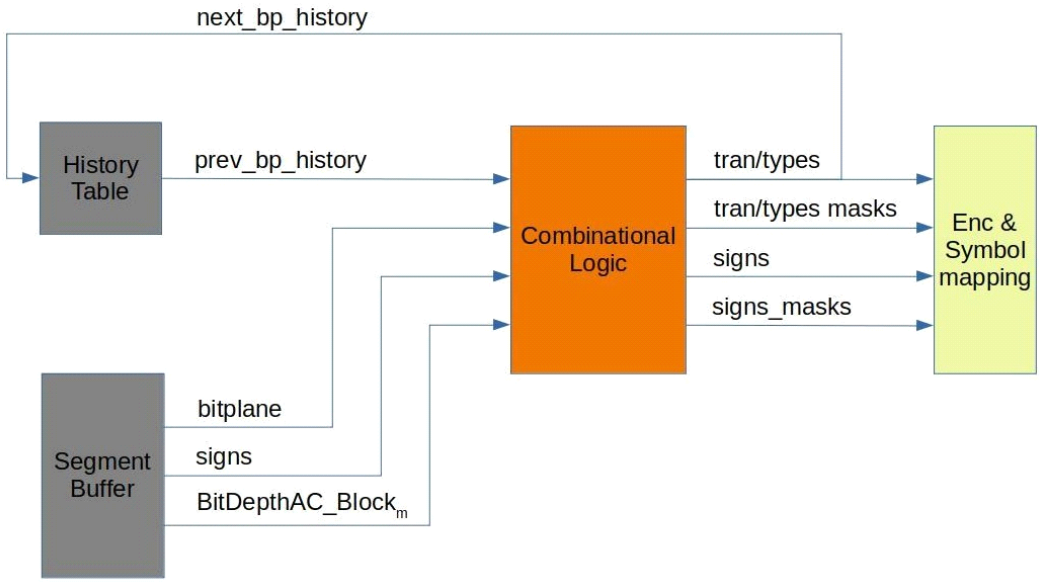


**Figure 3.2: Block Diagram of BSMS design without pipeline (1/2)**

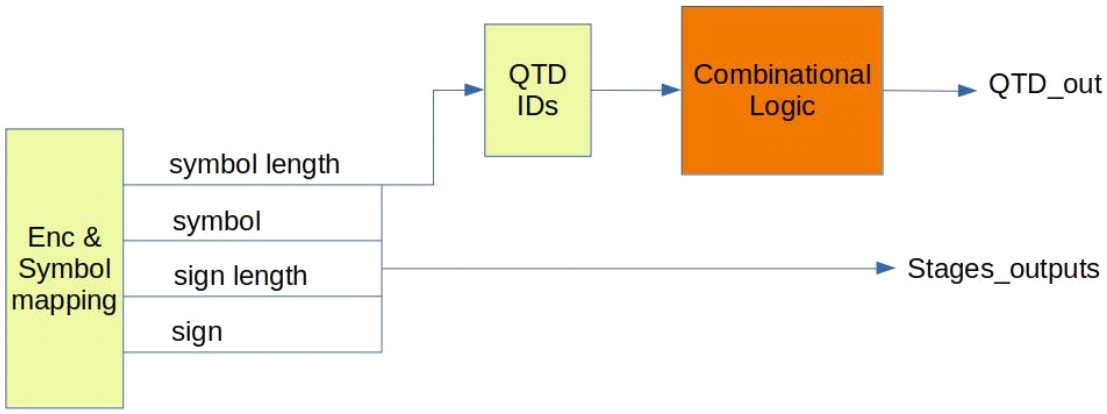


**Figure 3.3: Block Diagram of BSMS design without pipeline (2/2)**

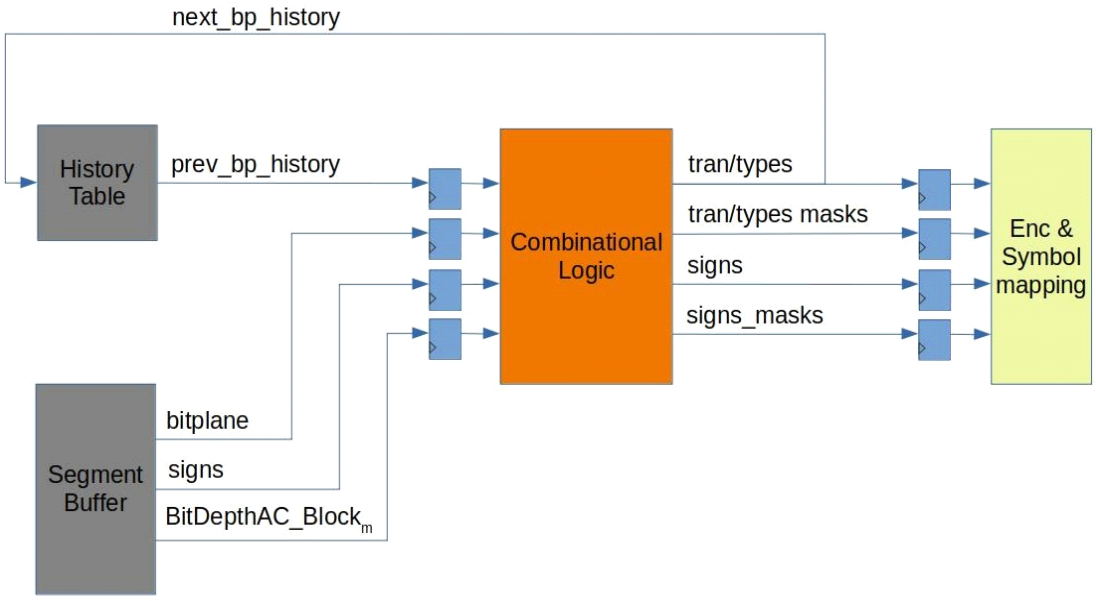


**Figure 3.4: Block Diagram of BSMS design with pipeline (1/2)**

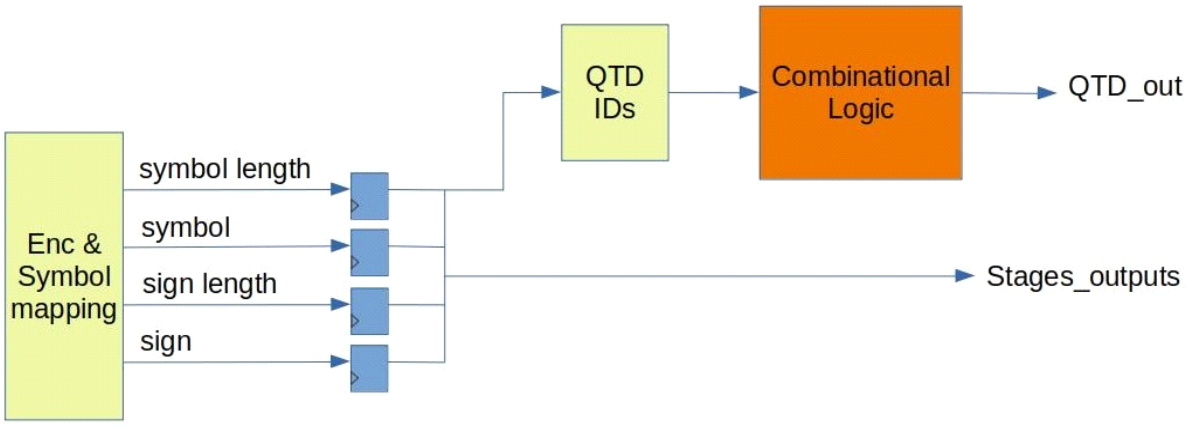


**Figure 3.5: Block Diagram of BSMS design with pipeline (2/2)**

## 3.3 Block scan example

The encoding that takes place in the BSMS module, concerns only the AC coefficients. So, for the Block Scan example below, we assume that the DC coefficients are omitted. Lets also assume that the values of the AC coefficients, that is parents, children and grandchildren, are the following:

**Table 3.2: Coefficient values for block example**

| | |
|---|---|
| Parents<br>-p0 = -6<br>-p1 = 10<br>-p2 = 5 | Children<br>-C0 = {2, 5, 2, 0}<br>-C1 = {3, -5, 0, 0}<br>-C2 = {-1, 3, 3, 0} |
| Grandchildren<br>-G0 = {1, 13, 0, 0, 9, 15, -5, 6, 10, 0, -4, -1, 0, 8, 0, 11}<br>-G1 = {0, -1, 6, 3, -2, 0, 0, 4, 1, -3, 4, 1, 0, 0, 0, -7}<br>-G2= {0, 1, -11, 13, 7, 5, 0, 1, -2, 9, 11, 1, -4, 0, 0, -8} | |
| BitDepthAC_Blockm = 4 | |

The above values, if put together one by one, with their binary equivalent vertically, they form four bitplanes as shown in tables 3.2, 3.3, 3.4, 3.5 below. The first row of each bitplane b is the formed bitplane. In the second (yellow) row are the previous (b+1) types, that initially are all set to zero, and in the third (red) row is the result of the logical OR from the previous two lines, that is the current (b) types.

The next calculations are the ones taking place in the combinational logic right before the symbol mapping.

**Table 3.3: Coefficient and bitplane values for Parents and Children**

| b | p0 | p1 | p2 | C0 | | | | C1 | | | | C2 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | -6 | 10 | 5 | 2 | 5 | 2 | 0 | 3 | -5 | 0 | 0 | -1 | 3 | 3 | 0 |
| | | | | | | | | | | | | | | | |
| 3 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 2 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 2 | 2 | 2 | 1 | 2 | 1 | 0 | 1 | 2 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| | 2 | 2 | 2 | 2 | 2 | 2 | 0 | 2 | 2 | 0 | 0 | 1 | 2 | 2 | 0 |

**Table 3.4: Coefficient and bitplane values for Grandchildren (1/3)**

| b | G0 | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 13 | 0 | 0 | 9 | 15 | -5 | 6 | 10 | 0 | -4 | -1 | 0 | 8 | 0 | 11 |
| | H00 | | | | H01 | | | | H02 | | | | H03 | | | |
| 3 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 2 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| | 0 | 2 | 0 | 0 | 2 | 2 | 1 | 1 | 2 | 0 | 1 | 0 | 0 | 2 | 0 | 2 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| | 0 | 2 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 0 | 2 | 0 | 0 | 2 | 0 | 2 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| | 1 | 2 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 0 | 2 | 1 | 0 | 2 | 0 | 2 |

**Table 3.5: Coefficient and bitplane values for Grandchildren (2/3)**

| b | G1 | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | -1 | 6 | 3 | -2 | 0 | 0 | 4 | 1 | -3 | 4 | 1 | 0 | 0 | 0 | -7 |
| | H10 | | | | H11 | | | | H12 | | | | H13 | | | |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| | 0 | 0 | 2 | 1 | 1 | 0 | 0 | 2 | 0 | 1 | 2 | 0 | 0 | 0 | 0 | 2 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| | 0 | 1 | 2 | 2 | 2 | 0 | 0 | 2 | 1 | 2 | 2 | 1 | 0 | 0 | 0 | 2 |

**Table 3.6: Coefficient and bitplane values for Grandchildren (3/3)**

| b | G2 | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | -11 | 13 | 7 | 5 | 0 | 1 | -2 | 9 | 11 | 1 | -4 | 0 | 0 | -8 |
| | H20 | | | | H21 | | | | H22 | | | | H23 | | | |
| 3 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| | 0 | 0 | 2 | 2 | 1 | 1 | 0 | 0 | 0 | 2 | 2 | 0 | 1 | 0 | 0 | 2 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| | 0 | 0 | 2 | 2 | 2 | 2 | 0 | 0 | 1 | 2 | 2 | 0 | 2 | 0 | 0 | 2 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| | 0 | 1 | 2 | 2 | 2 | 2 | 0 | 1 | 2 | 2 | 2 | 1 | 2 | 0 | 0 | 2 |

**Bitplane b=4**

typesb = typesb+1+ bitplaneb

**Stage 1**

**Mask based implementation of 3-bit words typesb[P] and signsb[P]**

-types3[P] = 010
-signs3[P] = _0_

Initially, typesb+1[P] = 000

typesb[P] = typesb+1[P] + bitplaneb[P] = 000 + 010 = 010

types_datab[P] = typesb[P] = 010

types_maskb[P] = typesb+1[P] = 000

signs_datab[P] = signsb[P] = 100

signs_maskb[P] = types_datab[P] • /types_maskb[P] = 010 • 111 = 010

typesb[P] = types_datab[P] = 010

**Stage 2**

**Mask based implementation of 1-bit tranBb& 3-bit tranDb**

-tranB3= 1

Initially, tranBb+1= 0

tranB_datab= (or typesb[B] ) = 1

tranB_maskb= tranBb+1= 0

-tranD3= 101

Initially, tranD0b+1= 0, tranD1b+1= 0, tranD2b+1= 0

tranD0_datab= (or typesb[D0]) = 1

tranD1_datab= (or typesb[D1]) = 0

tranD2_datab= (or typesb[D2]) = 1

tranD0_maskb= tranD0b+1+ /tranB_datab= 0 • 1 = 0

tranD1_maskb= tranD1b+1+ /tranB_datab= 0 • 1 = 0

tranD2_maskb= tranD2b+1+ /tranB_datab= 0 • 1 = 0

**Mask based implementation of 4-bit words typesb[Ci] and signsb[Ci]**

-types3[C0] = 0000

-signs3[C0] = ____

Initially, typesb+1[C0] = 0000

typesb[C0] = typesb+1[C0] + bitplaneb[C0] = 0000 + 0000 = 0000

types_datab[C0] = typesb[C0] = 0000

types_maskb[C0] = typesb+1[C0] + /tranD0b+ /tranBb= 0000 + 0000 + 0000 = 0000

signs_datab[C0] = signsb[C0] = 0000

signs_maskb[C0] = types_datab[C0] • /types_maskb[C0] = 0000 • 1111= 0000

-types3[C1] = ____
-signs3[C1] = ____

Initially, typesb+1[C1] = 0000
typesb[C1] = typesb+1[C1] + bitplaneb[C1] = 0000 + 0000 = 0000

types_datab[C1] = typesb[C1] = 0000
types_maskb[C1] = typesb+1[C1] + /tranD1b+ /tranBb= 0000 + 1111 + 0000 = 1111

signs_datab[C1] = signsb[C1] = 0100
signs_maskb[C1] = types_datab[C1] • /types_maskb[C1] = 0000 • 0000 = 0000

-types3[C2] = 0000
-signs3[C2] = ____

Initially, typesb+1[C2] = 0000
typesb[C2] = typesb+1[C2] +bitplaneb[C2] = 0000 + 0000 = 0000

types_datab[C2] = typesb[C2] = 0000
types_maskb[C2] = typesb+1[C2] + /tranD2b+ /tranBb= 0000 + 0000 + 0000 = 0000

signs_datab[C2] = signsb[C2] = 1000
signs_maskb[C2] = types_datab[C2] • /types_maskb[C2] = 0000 • 1111= 0000

**Stage 3**

**Mask based implementation of 3-bit tranbG**

-tran3G = 1_1

Initially, tranG0b+1= 0, tranG1b+1= 0, tranG2b+1= 0

tranG0_datab= (or typesb[G0]) = 1

tranG1_datab= (or typesb[G1]) = 0

tranG2_datab= (or typesb[G2]) = 1

tranG0_maskb= tranG0b+1+ /tranD0b+ /tranBb= 0 + 0 + 0 = 0

tranG1_maskb= tranG1b+1+ /tranD1b+ /tranBb= 0 + 1 + 0 = 1

tranG2_maskb= tranG2b+1+ /tranD2b+ /tranBb= 0 + 0 + 0 = 0

**Mask based implementation of 4-bit tranb[H0]**

-tran3H0= 1111

Initially, tranH00b+1= 0, tranH01b+1= 0, tranH02b+1= 0, tranH03b+1= 0

tranH00_datab= (or typesb[H00]) = 1

tranH01_datab= (or typesb[H01]) = 1

tranH02_datab= (or typesb[H02]) = 1

tranH03_datab= (or typesb[H03]) = 1

tranH00_maskb= tranH00b+1+ /tranG0b+ /tranBb= 0 + 0 + 0 = 0

tranH01_maskb= tranH01b+1+ /tranG0b+ /tranBb= 0 + 0 + 0 = 0

tranH02_maskb= tranH02b+1+ /tranG0b+ /tranBb= 0 + 0 + 0 = 0

tranH03_maskb= tranH03b+1+ /tranG0b+ /tranBb= 0 + 0 + 0 = 0

**Mask based implementation of 4-bit tranb[H1]**

-tran3H1= ____

Initially, tranH10b+1= 0, tranH11b+1= 0, tranH12b+1= 0, tranH13b+1= 0

tranH10_datab= (or typesb[H10]) = 0

tranH11_datab= (or typesb[H11]) = 0

tranH12_datab= (or typesb[H12]) = 0

tranH13_datab= (or typesb[H13]) = 0

tranH10_maskb= tranH10b+1+ /tranG1b+ /tranBb= 0 + 1 + 0 = 1

tranH11_maskb= tranH11b+1+ /tranG1b+ /tranBb= 0 + 1 + 0 = 1

tranH12_maskb= tranH12b+1+ /tranG1b+ /tranBb= 0 + 1 + 0 = 1

tranH13_maskb= tranH13b+1+ /tranG1b+ /tranBb= 0 + 1 + 0 = 1

**Mask based implementation of 4-bit tranb[H2]**

-tran3H2= 1011

Initially, tranH20b+1= 0, tranH21b+1= 0, tranH22b+1= 0, tranH23b+1= 0

tranH20_datab= (or typesb[H20]) = 1

tranH21_datab= (or typesb[H21]) = 0

tranH22_datab= (or typesb[H22]) = 1

tranH23_datab= (or typesb[H23]) = 1

tranH20_maskb= tranH20b+1+ /tranG2b+ /tranBb = 0 + 0 + 0 = 0

tranH21_maskb= tranH21b+1+ /tranG2b+ /tranBb = 0 + 0 + 0 = 0

tranH22_maskb= tranH22b+1+ /tranG2b+ /tranBb = 0 + 0 + 0 = 0

tranH23_maskb= tranH23b+1+ /tranG2b+ /tranBb = 0 + 0 + 0 = 0

**Mask based implementation of 4-bit words types[Hij] and signs[Hij]**

-types3[H00] = 0100

-signs3[H00] = _0__

Initially, typesb+1[H00] = 0000

typesb[H00] = typesb+1[H00] + bitplaneb[H00] = 0000 + 0100 = 0100

types_datab[H00] = typesb[H00] = 0100

types_maskb[H00] = typesb+1[H00] + /tranH00b+ /tranBb= 0000+ 0000 + 0000 = 0000

signs_datab[H00] = signsb[H00] = 0000

signs_maskb[H00] = types_datab[H00] • /types_maskb[H00] = 0100• 1111 = 0100

-types3[H01] = 1100

-signs3[H01] = 00__

Initially, typesb+1[H01] = 0000

typesb[H01] = typesb+1[H01] + bitplaneb[H01] = 0000 + 1100 = 1100

types_datab[H01] = typesb[H01] = 1100

types_maskb[H01] = typesb+1[H01] + /tranH01b+ /tranBb= 0000 + 0000 + 0000 = 0000

signs_datab[H01] = signsb[H01] = 0010

signs_maskb[H01] = types_datab[H01] • /types_maskb[H01] = 1100 • 1111 = 1100

-types3[H02] = 1000

-signs3[H02] = 0___

Initially, typesb+1[H02] = 0000

typesb[H02] = typesb+1[H02] + bitplaneb[H02] = 0000 + 1000 = 1000

types_datab[H02] = typesb[H02] = 1000

types_maskb[H02] = typesb+1[H02] + /tranH02b+ /tranBb= 0000 + 0000 + 0000 = 0000

signs_datab[H02] = signsb[H02] = 0011

signs_maskb[H02] = types_datab[H02] • /types_maskb[H02] = 1000 • 1111 = 1000

-types3[H03] = 0101

-signs3[H03] = _0_0

Initially, typesb+1[H03] = 0000

typesb[H03] = typesb+1[H03] + bitplaneb[H03] = 0000 + 0101 = 0101

types_datab[H03] = typesb[H03] = 0101

types_maskb[H03] = typesb +1[H03] + /tranH03b+ /tranBb= 0000 + 0000 + 0000 = 0000

signs_datab[H03] = signb[H03] = 0000

signs_maskb[H03] = types_datab[H03] • /types_maskb[H03] = 0101 • 1111 = 0101

-types3[H10] = ____

-signs3[H10] = ____

Initially, typesb+1[H10] = 0000

typesb[H10] = typesb+1[H10] + bitplaneb[H10] = 0000 + 0000 = 0000

types_datab[H10] = typesb[H10] = 0000

types_maskb[H10] = typesb+1[H10] + /tranH10b+ /tranBb= 0000 + 1111 + 0000 = 1111

signs_datab[H10] = signsb[H10] = 0100

signs_maskb[H10] = types_datab[H10] • /types_maskb[H10] = 0000 • 0000 = 0000

-types3[H11] = ____

-signs3[H11] = ____

Initially, typesb +1[H11] = 0000

typesb[H11] = typesb[H11] + bitplaneb[H11] = 0000 + 0000 = 0000

types_datab[H11] = typesb[H11] = 0000

types_maskb[H11] = typesb+1[H11] + /tranH11b+ /tranBb= 0000 + 1111 + 0000 = 1111

signs_datab[H11] = signsb[H11] = 1000

signs_maskb[H11] = types_datab[H11] • /types_maskb[H11] = 0000 • 0000 = 0000

-types3[H12] = ____

-signs3[H12] = ____

Initially, typesb +1[H12] = 0000

typesb[H12] = typesb +1[H12] + bitplaneb[H12] = 0000 + 0000 = 0000

types_datab[H12] = typesb[H12] = 0000

types_maskb[H12] = typesb+1[H12] + /tranH12b+ /tranBb= 0000 + 1111 + 0000 = 1111

signs_datab[H12] = signsb[H12] = 0100

signs_maskb[H12] =types_datab[H12] • /types_maskb[H12] = 0000 • 0000 = 0000

-types3[H13] = ____

-signs3[H13] = ____

Initially, typesb +1[H13] = 0000

typesb[H13] = typesb+1[H13] + bitplaneb[H13] = 0000 + 0000 = 0000

types_datab[H13] = typesb[H13] = 0000

types_maskb[H13] = typesb+1[H13] + /tranH13b+ /tranBb= 0000 + 1111 + 0000 = 1111

signs_datab[H13] = signsb[H13] = 0001

signs_maskb[H13] = types_datab[H13] • /types_maskb[H13] = 0000 • 0000 = 0000

-types3[H20] = 0011

-signs3[H20] = __10

Initially, typesb +1[H20] = 0000

typesb[H20] = typesb+1[H20] + bitplaneb[H20] = 0000 + 0011 = 0011

types_datab[H20] = typesb[H20] = 0011

types_maskb[H20] = typesb+1[H20] + /tranH20b+ /tranBb= 0000 + 0000 + 0000 = 0000

signs_datab[H20] = signsb[H20] = 0010

signs_maskb[H20] = types_datab[H20] • /types_maskb[H20] = 0011 • 1111 = 0011

-types3[H21] = ____

-signs3[H21] = ____

Initially, typesb +1[H21] = 0000

typesb[H21] = typesb+1[H21] + bitplaneb[H21] = 0000 + 0000 = 0000

types_datab[H21] = typesb[H21] = 0000

types_maskb[H21] = typesb+1[H21] + /tranH21b+ /tranBb= 0000 + 1111 + 0000 = 1111

signs_datab[H21] = signh_vb[H21] = 0000

signs_maskb[H21] = types_datab[H21] • /types_maskb[H21] = 0000 • 0000 = 0000

-types3[H22] = 0110

-signs3[H22] = _00_

Initially, typesb+1[H22] = 0000

typesb[H22] = typesb+1[H22] + bitplaneb[H22] = 0000 + 0110 = 0110

types_datab[H22] = typesb[H22] = 0110

types_maskb[H22] = typesb+1[H22] + /tranH22b+ /tranBb= 0000 + 0000 + 0000 = 0000

signs_datab[H22] = signh_vb[H22] = 1000

signs_maskb[H22] = types_datab[H22] • /types_maskb[H22] = 0110 • 1111 = 0110

-types3[H23] = 0001

-signs3[H23] = ___1

Initially, typesb+1[H23] = 0000

typesb[H23] = typesb+1[H23] + bitplaneb[H23] = 0000 + 0001 = 0001

types_datab[H23] = typesb[H23] = 0001

types_maskb[H23] = typesb+1[H23] + /tranH23b+ /tranBb= 0000 + 0000 + 0000 = 0000

signs_datab[H23] = signh_vb[H23] =1001

signs_maskb[H23] = types_datab[H23] • types_maskb[H23] = 0001 • 1111 = 0001

### Stage 4

Refinement bits:
-Parents = {}
-Children = {}
-Grandchildren = {}

ref_datab[P] = bitplaneb[P] = 010

ref_maskb[P] = typesb+1[P] = 000

ref_datab[C0] = bitplaneb[C0] = 0000

ref_maskb[C0] = typesb+1[C0] = 0000

ref_datab[C1] = bitplaneb[C1] = 0000

ref_maskb[C1] = typesb+1[C1] = 0000

ref_datab[C2] = bitplaneb[C2] = 0000

ref_maskb[C2] = typesb+1[C2] = 0000

ref_datab[H00] = bitplaneb[H00] = 0100

ref_maskb[H00] = typesb+1[H00] = 0000

ref_datab[H01] = bitplaneb[H01] = 1100

ref_maskb[H01] = typesb+1[H01] = 0000

ref_datab[H02] = bitplaneb[H02] = 1000

ref_maskb[H02] = typesb+1[H02] = 0000

ref_datab[H03] = bitplaneb[H03] = 0101

ref_maskb[H03] = typesb+1[H03] = 0000

ref_datab[H10] = bitplaneb[H10] = 0000

ref_maskb[H10] = typesb+1[H10] = 0000

ref_datab[H11] = bitplaneb[H11] = 0000

ref_maskb[H11] = typesb+1[H11] = 0000

ref_datab[H12] = bitplaneb[H12] = 0000

ref_maskb[H12] = typesb+1[H12] = 0000

ref_datab[H13] = bitplaneb[H13] = 0000

ref_maskb[H13] = typesb+1[H13] = 0000

ref_datab[H20] = bitplaneb[H20] = 0011

ref_maskb[H20] = typesb+1[H20] = 0000

ref_datab[H21] = bitplaneb[H21] = 0000

ref_maskb[H21] = typesb+1[H21] = 0000

ref_datab[H22] = bitplaneb[H22] = 0110

ref_maskb[H22] = typesb+1[H22] = 0000

ref_datab[H23] = bitplaneb[H23] = 0001

ref_maskb[H23] = typesb+1[H23] = 0000

# 4. BSMS VERIFICATION AND VALIDATION STRATEGY

The BSMS VHDL design has been extensively verified by RTL simulation using Vivado Simulation. The existing VHDL code that was developed in 2014 was used as a golden reference model of BSMS.

On a first level the verification was accomplished with tests on the BSMS code alone, using three memories, one with random bitplanes, one with random signs and one with random bit depths, that simulate the segment buffer. Moreover to verify the correct flow of data at the outputs of the BSMS module, a testbench with random stalls was developed. That testbench triggered the FIFOs' full signals to rise at random times throughout encoding. With the addition of some logic in the design it was ensured that when a FIFO is full, the design behaves correctly saving the information to be outputted when the FIFO is ready to receive it.

And on a second level the submodule was verified with the BSMS being integrated into the 2D encoder design and tested with real images, using the output bitstream of the 2D encoder with the exiting BSMS module as a golden reference. The tests performed on the Bit plane encoder include a significant amount of test images from the corpus of images [3] available in [6].

# 5. EXPERIMENTAL RESULTS

The proposed architecture was implemented targeting the ZC706 Evaluation Board for the Zynq-7000 XC7Z045 SoC FPGA.

The frequency of 280.9 MHz (period 3.56 ns) was achieved for the BSMS module and 181.8 MHz (period 5.5 ns) for the 2D Encoder with the new BSMS module integrated.

The proposed architecture processes 1 sample / cycle, in contrast with the previous implementation that processes 1 sample / 2 cycles.

Moreover, experimental results are also provided for comparisons with the previous implementation. The Xilinx Vivado Design Suite tool was used for the implementation, analysis and simulation.

The detailed implementation statistics including FPGA resources are shown in Table 5.1.

**Table 5.4.1: Implementation statistics targeting XC7Z045 FPGA.**

| | Used | Available | Util% |
|---|---|---|---|
| **LUTs** | 1890 | 218600 | 0.86 |
| **BRAMs** | 0 | 545 | 0.00 |
| **Registers** | 1320 | 437200 | 0.30 |

Comparison in power and timing analysis with the existing work, targeting the same XC7Z045 SoC FPGA is shown in Table 5.2.

**Table 5.4.2: Comparisons with the existing implementation targeting the same XC7Z045 FPGA**

| | this Thesis | Theodorou |
|---|---|---|
| **Power** | 0.329 W | 0.262 W |
| **Frequency** | 280.9 MHz | 207.2 MHz |
| **Clock Period** | 3.560 ns | 4.825 ns |
| **Clock cycles** | 2328 | 4641 |
| **Samples/cycle** | 1 | 0.5 |

The power consumption statistics were evaluated using the Xilinx Vivado power estimator on the post Synthesis design using default environmental settings.
To compare in terms of Clock cycles, the two designs were simulated for a segment of 128 blocks and 18 bitplanes, assuming in both that the FIFOs, where the stages' data are pushed in, are never full.

# 6. CONCLUSIONS

In this thesis, we have introduced a high performance architecture for the BSMS module of the Bit Plane Encoder of the CCSDS 122.0B2 Image Data Compression (IDC) algorithm. The proposed parallel architecture achieves 1 sample/cycle while the deep pipeline enables high clock frequencies.

# REFFERENCES

[1] N. Kranitis, G. Theodorou, A. Tsigkanos, A. Paschalis and R. Vitulli, "An Over 2 Gbps Reconfigurable FPGA Implementation of CCSDS 122.0-B-1 Image Data Compression," 2014.

[2] N. Kranitis, G. Theodorou, A. Tsiganos, A. Paschalis and R. Vitulli, "A Reconfigurable FPGA Implementation of CCSDS 122.0-B-1 Image Data Compression for ESA PROBA-3 Coronagraph System Payload," in *On-Board Payload Data Compression Workshop (OBPDC 2014)*, Venice, Italy.

[3] CCSDS, "Image Data Compression, 120.1-G-2, Green Book," 2005.

[4] CCSDS, "Image Data Compression, 122.0-B-2, Blue Book," 2017.

[5] CCSDS, "Spectral Preprocessing Transform for Multispectral and Hyperspectral Image Compression, 122.1-B-1, Blue book," 2017.

[6] CCSDS, "CCSDS 122 Test Image Set," [Online]. Available: http://cwe.ccsds.org/sls/docs/.

[7] CCSDS, “Image Data Compression, 122.0-B-1, Blue Book,” 2005.