Implementation of Context-Based Adaptive Binary Arithmetic Coding on KiloCore Processor Arrays

By

SHARMILA SRIRANGA KULKARNI

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

____

Bevan Baas, Chair

____

Venkatesh Akella

____

Soheil Ghiasi

Committee in charge

2021

© Copyright by Sharmila Sriranga Kulkarni 2021
All Rights Reserved

Abstract

H.264/AVC is a popular video coding standard used in the fields of communication, video streaming and broadcasting. The H.264/AVC standard as specified in ITU-T — ISO/IEC has two methods of entropy coding, namely Context-based Adaptive Variable Length Coding (CAVLC) and Context-based Adaptive Binary Arithmetic Coding (CABAC). CABAC utilizes probability estimation to achieve a bit-rate reduction of 19% compared to CAVLC.

In order to deal with the higher level of computational complexity of the CABAC entropy coding over the CAVLC coding, the CABAC algorithm is sometimes implemented in hardware to achieve real-time high resolution video coding. The CABAC algorithm can be broken into smaller tasks that can be performed independently. This makes the many-core processor array an appropriate option for the hardware implementation of the CABAC encoding algorithm. The independent tasks within the algorithm can be assigned to individual cores of the array. The KiloCore II, which this thesis uses for its hardware platform, contains hundreds of programmable processors and multiple 64kB shared memories per chip. Lanes of processors are constructed to perform the functions of the blocks within the CABAC.

The aim of this thesis is to compare the throughput results with existing hardware and software implementations of CABAC and to show that the throughput, power and energy in the case of the KiloCore II is competitive. The CABAC algorithm was mapped on the KiloCore II array using the Project manager and Simulator platform. The total area occupied by the algorithm was 3.52 mm^2 in 32 nm technology with 64 cores and 177 routing links. The implementation achieved a throughput of 37 million bins per second at 1.1 V operating voltage and an energy of 34.37 μJ per individual bin at 0.8 V operating voltage.

This implementation of the CABAC has an improvement of 57 times in throughput, when compared to the software implementation, that is, the JM software reference run on the Intel Xeon Processor E5-2680 v2. Despite being a fully-software implementation, the presented KiloCore design achieves a throughput within a factor of five when compared to hardware CABAC implementations scaled to the same 32 nm fabrication technology.

Acknowledgments

The body of work that is described in this thesis has been a major part of my masters and would not be possible without the support and help of so many people. Firstly, I would like to thank Professor Bevan Baas for inspiring me to challenge myself and take up this project. Professor Baas was always available with his guidance and his time, which is greatly appreciated.

I would like to thank Professor Venkatesh Akella and Professor Soheil Ghiasi for serving on my thesis committee and for reviewing my work.

I would like to extend my gratitude towards Renjie Chen for giving me a detailed introduction into the working of CABAC. I am grateful for the work done by Brent Bohenstiehl on the Project Manager and the Compiler, tools which I utilized to complete my research. I would like to thank Mark Hildebrand and Satyabrata Sarangi for their work on the Mapper software and the DeepScaleTool respectively, which were important tools that made my work easier. I am thankful for the help and assistance I received from my fellow lab members Yushan Wu and Filipe Borges. I am grateful for Shifu Wu, Timothy Andreas for their advice, support and informative tutorials.

I am eternally grateful for all the support I received from my family. Their encouragement and belief in me helped me through the course of my research.

Contents

Abstract ii

Acknowledgments iii

List of Figures vi

List of Tables viii

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Thesis Organization 2

2 Overview of H.264/Advanced VideoCoding (AVC) Standard 3
- 2.1 Overview of H.264/AVC 3
  - 2.1.1 Macroblocks and Slices 4
  - 2.1.2 Encoding process of H.264 6
- 2.2 Context-based Adaptive Binary Arithmetic Coding (CABAC) 8
  - 2.2.1 Overview of CABAC 9
  - 2.2.2 Syntax elements 12

3 The KiloCore Many-Core Processor Array Architecture 16
- 3.1 Processors 16
- 3.2 Memory 17
- 3.3 Communication between processors 17
- 3.4 Project Manager 18

4 Methodology 20
- 4.1 Overview 20
  - 4.1.1 Input data handling 22
- 4.2 Binarizer 25
  - 4.2.1 Tabular lookup coding 27
  - 4.2.2 Fixed Length coding 27
  - 4.2.3 Unary coding 29
  - 4.2.4 Truncated unary coding 30
  - 4.2.5 Unary Exponential Golomb k-th order coding 30
- 4.3 Context Modeler 35
  - 4.3.1 Context index increment computation cores 38
  - 4.3.2 Calculation of Context index 47

4.4 Binary Arithmetic Encoder . . . 47

5 JM software and functional verification **55**
5.1 JM software . . . 55
5.2 Binarization verification . . . 56
5.2.1 Debugging the binarizer . . . 57
5.3 Context Modeler verification . . . 58
5.4 Binary Arithmetic encoder verification . . . 58
5.4.1 Debugging the Context modeler and Arithmetic Encoder . . . 59

6 Experimental Results and Analysis **61**
6.1 Analysis of core usage . . . 61
6.1.1 Mapping to the KiloCore II . . . 64
6.2 Throughput and energy results . . . 66
6.2.1 Throughput through the stages . . . 66
6.2.2 Scaling with Voltage results . . . 68
6.2.3 Energy results . . . 70
6.3 Comparison with other implementations of CABAC . . . 70

7 Thesis summary and Future work **74**
7.1 Thesis summary . . . 74
7.2 Future work . . . 74
7.2.1 Improving Throughput and energy . . . 75

Bibliography **77**

List of Figures

2.1 Progressive and interlaced frames and fields [1] 4
2.2 4:2:0 Sampling of YCbCr color space [2] 5
2.3 Macroblock and submacroblock partitions [1] 5
2.4 H.264 coding structure for a macroblock [1] 6
2.5 H.264/AVC profiles [2] 7
2.6 CABAC encoding engine [3] 9
2.7 Updating the Range and Low variables in BAE 11
2.8 Updating the Range and Low variables in BAE 12

3.1 KiloCore pipeline [4] 17

4.1 CABAC data flow 20
4.2 Representation of a single core's ports. 21
4.3 Colors of various stages of cores 22
4.4 Complete input and output connections of the core *bin_distributor*(I_2). I_0: *input_reader*, I_1: *input_slice*, I_3: *slice_mb_storage*, B_0 to B_4: binarizer cores 23
4.5 Complete input and output connections of the core *input_reader* (I_0) and *slice_mb_storage* (I_3). I_2: *bin_distributor*, C_0: *ctxIdxInc_mbtype*, C_1: *ctxIdxInc_fixed*, C_2: *ctxIdxInc_blockcat*, C_3: *ctxIdxInc_tunary*, C_4: *uegk_parser* 24
4.6 Prefix and Suffix words in 16-bit register: Case 1 33
4.7 Prefix and Suffix words in 16-bit register: Case 2 33
4.8 Prefix and Suffix words in 16-bit register: Case 3 34
4.9 Flowchart of the Context modeler block 36
4.10 Connections for the Context modeler block. B_0 to B_6 are the binarizer block cores. C_0 to C_5 are the context index increment cores. C_6 to C_10 are the context index calculator cores. J_0 cores are the *bin_join* cores. 37
4.11 Neighbors in reference to current macroblock. [5] 38
4.12 Neighbor of CBPLuma block [6] 45
4.13 Flow chart of the Binary Arithmetic encoder 48
4.14 Connections for the Binary arithmetic encoder block. J_0e is the last *bin_join* core. A_0: *bae_stage0_0*, P_X: pStateIdx cores, D_X: Demux cores, R_X: rangeTableLPS cores, A_1: *bae_stage1_0*, A_2: *bae_stage1_1*, A_3: *renorm*, A_4: *Accumulate_bins* . . 49
4.15 Connection between the Context table cores and the pStateIdx cores 50

6.1 Chart showing distribution of tasks and the number of cores 62
6.2 Standard mapping of cores 64
6.3 Placed and Routed map of cores 65
6.4 Chart showing link length data 66

6.5 Throughput of the stages 68
6.6 Throughput variation with Voltage 69
6.7 Energy variation with Voltage 69
6.8 Power variation with Voltage 69

7.1 Neighbor information to be stored for ctxIdx calculation [6] 75

List of Tables

2.1 Syntax element categories for CABAC 13

4.1 Input varaibles from JM software 22
4.2 Values of ctxIdxOffset for each syntax element type 26
4.3 Example of fixed length coding when cmax = 7 [6] 29
4.4 cmax values for truncated unary coding 30
4.5 Parameters for UEGk coding 31
4.6 Macroblock partition categories 40
4.7 Sub-macroblock sub-partition categories 40
4.8 Indices representing partition type 41
4.9 Indices for when partitions are 8x8 42
4.10 Indices for when partitions are 4x4 42
4.11 Categories of transform block 46
4.12 ctxIdxBlockCatOffset based on the syntax element and ctxBlockCat 47
4.13 Range of ctxIdx handled by each pState core 51

5.1 JM generated file information 56
5.2 Ranges of syntax elements 57
5.3 File test_output variables 59

6.1 Area usage per task 63
6.2 Mapper details about links 65
6.3 Stage-wise throughput 67
6.4 Effects of voltage 70
6.5 Energy per bin measurements 70
6.6 CABAC throughput measurements 72
6.7 CABAC scaled measurements 73

Chapter 1

Introduction

1.1 Motivation

With the ever-increasing demand and consumption of streaming video content across the globe, be it Netflix, Youtube, Tiktok or Zoom meetings, the underlying algorithm for video transmission has to be well-tuned to deal with variable internet speeds and not allow video buffering. As a result, there is a pressing need to come up with good data compression techniques and improve upon existing ones. The H.264 Advanced Video Coding (AVC) standard provides very good data compression, making it a popular algorithm for video or data coding and transmission.

The two methods of entropy coding available in the H.264/AVC standard are Context-based Adaptive Variable Length Coding (CAVLC) and Context-based Adaptive Binary Arithmetic Coding (CABAC). CABAC particularly, is able to reduce the data redundancy based on its statistical property and is able to achieve a significantly higher compression ratio especially for video with high resolution. The CABAC algorithm can be broken into many small tasks that can be performed independently. With the CABAC hardware implementation, it is possible to reduce the storage and resource cost by running multiple arithmetic encoding engines at the low level (coding block) and enabling the pipeline within the arithmetic coding engine. This makes the many-core processor array an appropriate option for the hardware implementation of the CABAC encoding algorithm.

The Kilcore 2 processor array contains 1000 processors that can be independently programmed. The instruction memory in each core is 128x40b. This allows a small task from the CABAC to fit on each processor. Communication between cores (processors) on the chip is through

a highly reliable Circuit switched link. Each processor has two input buffers for the processing of this input data. Each core also has 8 output ports. This allows us to fan out the information being processed by the input handling cores, to perform the tasks in parallel. The goal of this research is provide the Implementation of CABAC on KiloCore II and a comparative analysis of the performance of the KiloCore II implementation of the CABAC with other existing hardware implementations. Due to serial processing nature of the Binary Arithmetic Encoder, parallelism at the slice level in the CABAC is challenging [7]. Task parallelism is employed for the probability state variables updating in this implementation.

1.2 Thesis Organization

The thesis chapters are organized as follows.

Chapter 2 gives a background on the H.264 standard. In particular, information on the CABAC algorithm is provided here in detail, including explanation of the CABAC encoding engine, the CABAC implementation along with all the syntax elements used in the CABAC coding process.

Chapter 3 explains the working of the Multi-core processor array, KiloCore Platform, with details on the Processors, Memory and Inter-Processor Communication.

Chapter 4 details the methodology of the CABAC implementation with the following subsections: one on Input Data handling, input output connections of the cores, a subsection on the Binarizer implementation with its constituent cores, syntax elements and the coding methods and algorithms used, a subsection with a detailed explanation on the working of the Context Modeler with its constituent cores and Content Index calculation, and finally a subsection on the Binary Arithmetic Encoder with its cores and algorithms.

Chapter 5 gives the Functional verification. Firstly it talks about the H.264 JM reference software which is used as the reference for functional verification. The CABAC related configuration of JM encoder and the testing methodology are explained. Then the Binarizer verification, Context Modeler verification is listed.

Chapter 6 has the Experimental results and analysis. The Results of throughput and energy are presented. The analysis is done by taking comparisons from previous works and implementations of CABAC on different hardware platforms.

Chapter 2

Overview of H.264/Advanced VideoCoding (AVC) Standard

2.1 Overview of H.264/AVC

The H.264/AVC standard as described by the Telecommunication Standardization Sector of International Telecommunication Union (ITU-T) details the encoding and decoding of video. As demand for video streaming has gone up, so has the resolution, thus greatly increasing file sizes to be stored and transmitted over the network. The raw video files are compressed for storage and transmission in accordance with the H.264/AVC standard. The Motion Pictures Experts Group (MPEG) and the Video Coding Experts Group (VCEG) developed the standard. The popularity for the standard is due to the good graphic quality despite having high compression rates. Compared with existing standards, H.264/AVC provides improvements in rate-distortion efficiency [1].

H.264/AVC standard has found significance in the areas of broadcast, satellite, storage (optical or magnetic storage devices), LAN, modem, wireless or mobile networks, streaming services, Multimedia messaging services and many more.

The standard elaborates on Video Coding Layer (VCL), which is used to represent the video content, and on Network Abstraction Layer (NAL), which is responsible for generating header information, either for transmitting the video coded data over the network or for storage. The Video Coding Layer is the portion of the standard that will be discussed here, given that the Entropy coding methods make up a part of the VCL.

The flexibility that is availed by the standard is due to improvements in motion compensation, motion vector accuracy, reference picture increase, better prediction methods, improvements in transform, data partitioning and introduction of entropy coding.

Some of the salient features of the H.264 algorithm are explained in this section. Macroblocks and Slices, explained in Section 2.1.1, make up the building blocks of the representation of images in the video being encoded. The encoding process is detailed in Section 2.1.2.

2.1.1 Macroblocks and Slices

In the VCL, coded pictures make up a coded video sequence. Each coded picture is represented in block-shaped units of its constituent luma and chroma samples. These block units are called Macroblocks.

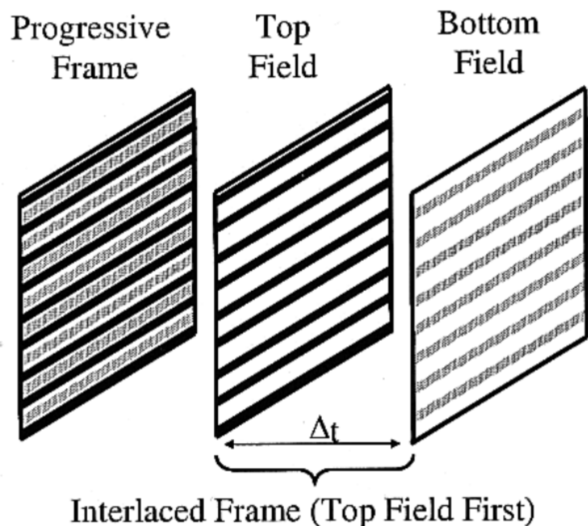

Figure 2.1: Progressive and interlaced frames and fields [1]

A coded picture can either be a frame or a field. A frame consists of a top and bottom field which are interleaved in the frame. The top field contains the even-numbered rows and the bottom field contains the odd-numbered rows. When the two fields are captured at different points of time, the frame is called an interlaced frame otherwise called a progressive frame [1]. Figure 2.1 shows the interlaced and progressive frames.

The brightness and color information is represented using the YCbCr color space. The chroma information (CbCr) has one fourth of the sampling resolution compared with the luma (Y) information. This is called 4:2:0 sampling, with each sample being 8 bits, is shown in Figure 2.2.

Each picture is divided into Macroblocks which each contain 16x16 samples of the luma component and 8x8 smaples of the two chroma components.

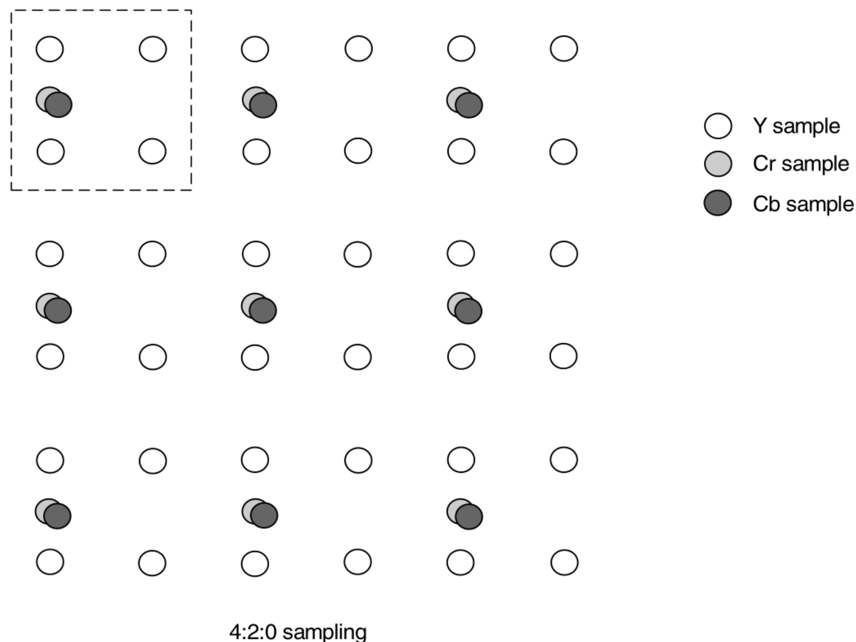

Figure 2.2: 4:2:0 Sampling of YCbCr color space [2]

A sequence of macroblocks in a raster scan order make up a slice. There are three types of slices used in the Main profile of H.264/AVC. They are the I slice, the P slice and the B slice.

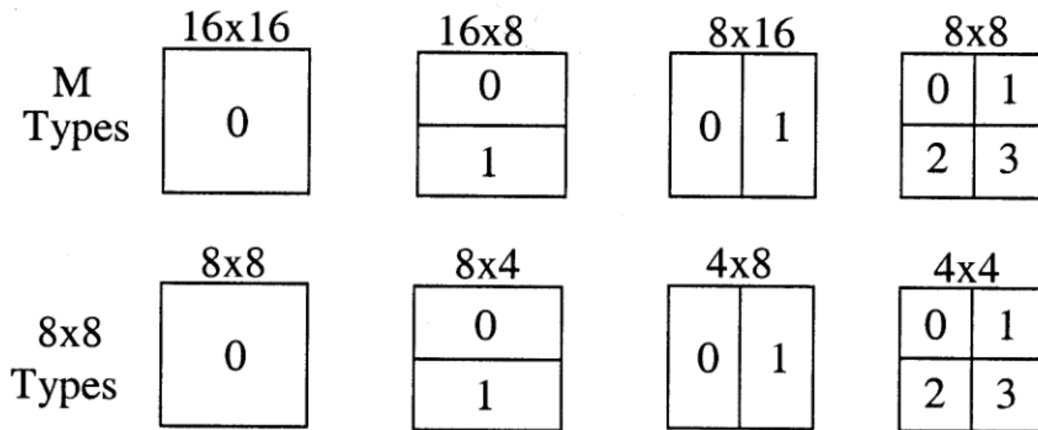

Figure 2.3: Macroblock and submacroblock partitions [1]

The I slice is the one within which all the macroblocks are coded using intra prediction. The P slice uses inter prediction with only one motion-compensated prediction signal for some

macroblocks, along with intra-prediction. Finally the B slice, which along with the P slice coding prediction methods, can also use two motion-compensated prediction signals per prediction block.

Macroblock partitioning Each macroblock contains 256 pixels (16x16 square). These macroblocks can be further partitioned into submacroblocks, which in turn can also be partitioned again. The 16x16 macroblock can be partitioned into 16x8, 8x16 or 8x8 partitions. The 8x8 submacroblock can be partitioned into 8x4, 4x8 and 4x4 sub-partitions. These partitions are shown in Figure 2.3.

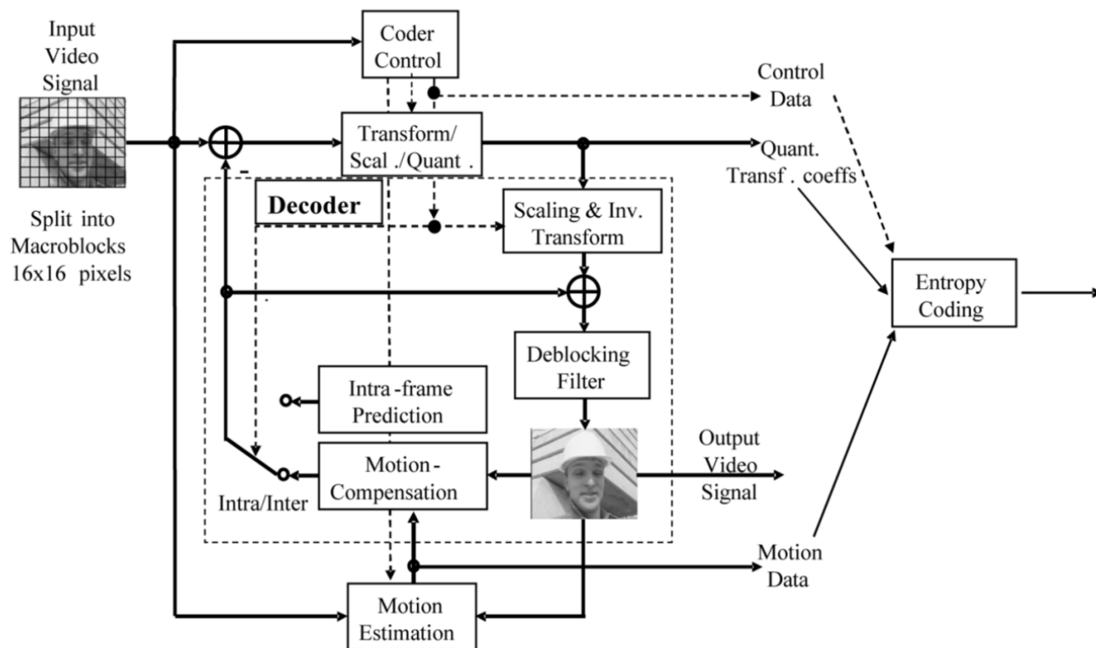

Figure 2.4: H.264 coding structure for a macroblock [1]

2.1.2 Encoding process of H.264

The encoding consists of four stages. The first stage is Prediction, of which there are two types. Inter and Intra prediction. In intra prediction, the signal is predicted only from previously coded samples of the same slice. In inter prediction, the signal is predicted using coded samples of both before and after the current sample in the sequence. The residual data, which is the difference between the prediction and the original picture, is sent to the next stage which is the Transform stage. Here coefficients are produced by using an interger transform such as Discrete Cosine Transform (DCT). The produced coefficients are quantized in the third stage. Finally the

quantized coeffiencients are converted into a bit stream using the entropy encoder. The coding procedure is shown in Figure 2.3.

There are three profiles for the H.264/AVC standard, based on requirements of the applications. They are Baseline, Main and Extended profile.

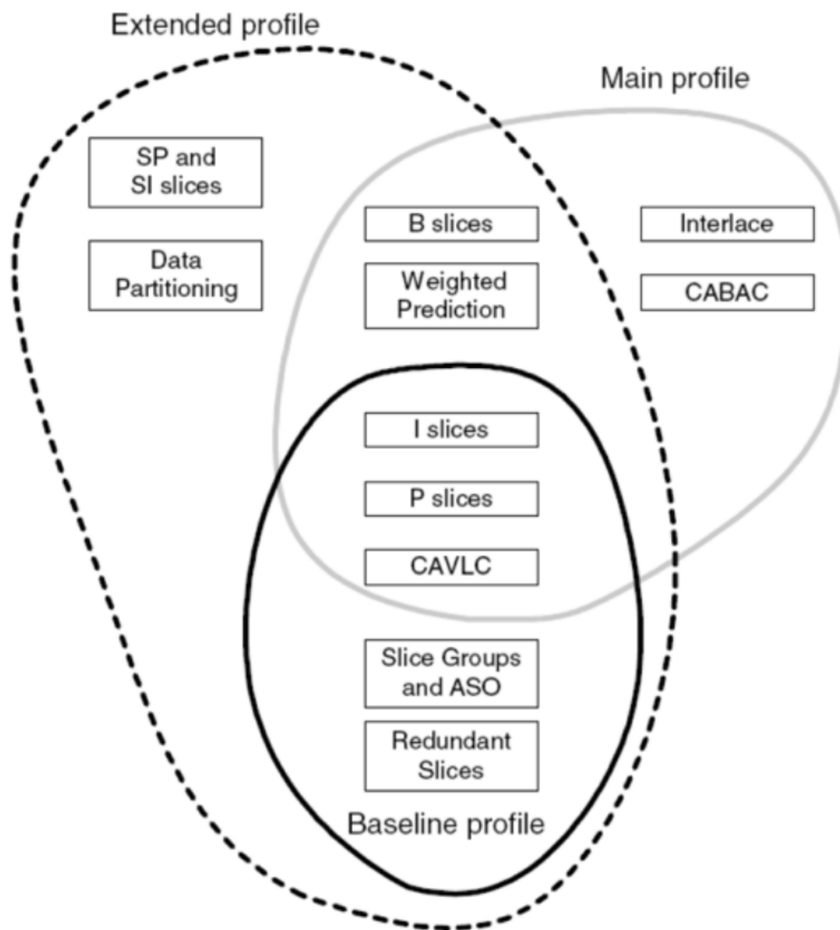

Figure 2.5: H.264/AVC profiles [2]

The profiles and their salient features are shown in Figure 2.4. In this body of work, the main profile is used as it includes the CABAC encoding engine.

There is a fully-parallel H.264/AVC baseline encoder on a 167-core asynchronous array of simple processors(AsAP) computation platform [8]. However, this implementation uses the other entropy encoding method, namely the Context-based Adaptive Variable Length Coding (CAVLC) [9]. Another paper [10] focuses on the parallelization of the H.264/AVC baseline residual encoder and CAVLC.

2.2 Context-based Adaptive Binary Arithmetic Coding (CABAC)

Entropy coding is an inherently lossless compression scheme which, based on its statistical property, reduces the redundancy of the input data. Some of the most commonly used entropy coding techniques are Run Length Coding (RLC), Huffman coding and arithmetic coding. The H.264 standard uses CAVLC or Context-Adaptive Variable-Length Coding for baseline profile, and CABAC or Context-based Adaptive Binary Arithmetic Coding is applied for main profile, extended profile and also high profile.

The CAVLC, in addition to being adopted for encoding the zig-zag order transformed residual coefficients, is also used in the prediction modes of intra prediction and the motion vectors of inter prediction. Huffman coding is applied in CAVLC for coding transform coefficients and Exponential Golomb coding is used for encoding the prediction modes and motion vectors.

CABAC utilizes a special scheme of binary arithmetic coding for encoding the semantics of the syntax elements (SE) from previous H.264 coding procedures. These include the type of macroblock, reference index and motion vector difference in inter prediction, prediction modes in intra prediction, parameters for quantization, residual data parameters and coefficients. Both binary and non-binary syntax elements can be encoded by CABAC, while the non-binary syntax element will get binarized before being coded in arithmetic coding. During binary arithmetic coding process, a probability model, or the context model as it is known, is selected adaptively based on the local coding context, which includes the previous coded information in current coding macroblock, as also the neighboring macroblocks information.

This adaptive selection of context model in CABAC allows for a more accurate probability modeling than the conventional arithmetic coding scheme, as a result of which CABAC achieves high compression performance. In addition, CABAC is a multiplication-free coding system, where, without the use of multiplication, the interval division and probability updating process in arithmetic coding is implemented by table a look-up algorithm which is based on quantized probability states and quantized interval range. Hence, CABAC acheives accelerated computation when compared to conventional multiplication-based arithmetic coding.

As can be seen by the coding performance evaluation in [11], CABAC achieves an average bit rate reduction of 15 percent to 19 percent over CAVLC when tested video sequences with different formats. With higher definition ideo sequence, the reduction in bit rate over CAVLC

is more significant. Hence, considering the current increasing demand for high resolution video, CABAC is a promising entropy coding option.

The next subsection explains in detain about the CABAC algorithm and the stages involved in it. The processing element, namely the Syntax element is also further explained in the following subsection.

2.2.1 Overview of CABAC

The CABAC coding has three elementary procedures: binarization, context modeling (CM) and binary arithmetic encoding (BAE). The coding engine for CABAC is illustrated in Figure 2.6.

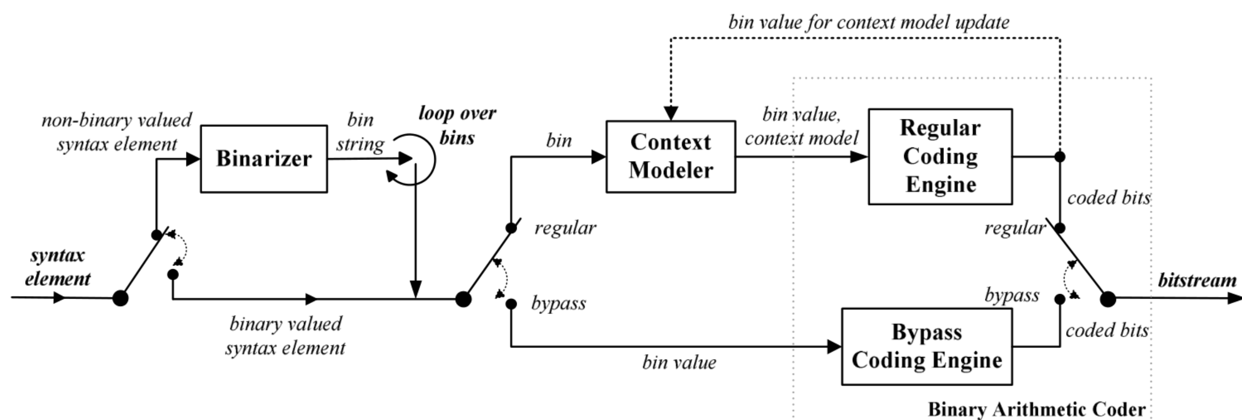

Figure 2.6: CABAC encoding engine [3]

Binarization is a data pre-processing procedure. In the Binarization procedure, non-binary syntax elements are coded into a string of binary symbols called *bins*. An individual binary symbol is referred to as simply a *bin*. There are totally five methods to binarize the input syntax elements. The methods are listed below:

- Table mapping
- Unary coding
- Truncated Unary coding
- Fixed length coding
- Unary Exponential Golomb k-th order (UEG-k) coding

The context modeler computes a context index (ctxIdx) for each bin. This context index is used to find a context model that is stored as the probability state tables. These tables are updated with each bin and reinitialised at the start of each slice.

For the Main profile of the H.264 standard, a total of 399 context models are stored. The context index (ctxIdx) is usually the sum of the context index offset (ctxIdxOffset) and the context index increment (ctxIdxInc). The only exceptions to the above statement is the calculation of the context index for residual syntax elements, where it is the sum of ctxIdxOffset, ctxIdxInc and context block category offset (ctxBlockCatOffset). The ctxBlockCatOffset depends on the context block category of the macroblock presently being encoded.

The ctxIdxOffset is solely determined by the syntax element type and slice type. The ctxIdxInc, however is more complicated and differs for each bin of the coded syntax element. This indicates it is dependent on the index of the bin or bin index (binIdx).

The calculation of the ctxIdxInc is dependent upon neighbor information in some of the cases. For residual syntax elements, the ctxIdxInc calculation also depends upon the scanning position of the current element being coded and upon the number of previously encoded coefficients. The calculations specific to each element are further elaborated in Chapter 4.

Finally each bin value and its corresponding ctxIdx is sent to the Binary Arithmetic encoder. The Binary Arithmetic encoder stores information such as most probable symbol (MPS), and the probability of that state. This consists of the context information that can be accessed with the ctxIdx. In the Binary Arithmetic encoder, there are two possible symbols, namely 0 (zero) and 1 (one). If one of the symbols is the most probable symbol, then the other symbol becomes the least probable symbol (LPS). Typically a memory is employed to store the context information consisting of probability state index and value of most probable state (ranging from 0-399), called context memory or context table.

In arithmetic coding, a coding interval is setup and updated based on the probability of MPS and LPS. The code word of arithmetic coding is generated from recursively dividing the interval. Two variables are used to keep track of the interval. The Low variable and the Range variable. In this implementation they are referred to as codILow and codIRange respectively. The Figure 2.7 shows the Range and Low values and when they are updated.

The initial value of the Range is 510 and it is a 9-bit register. The initial value of the

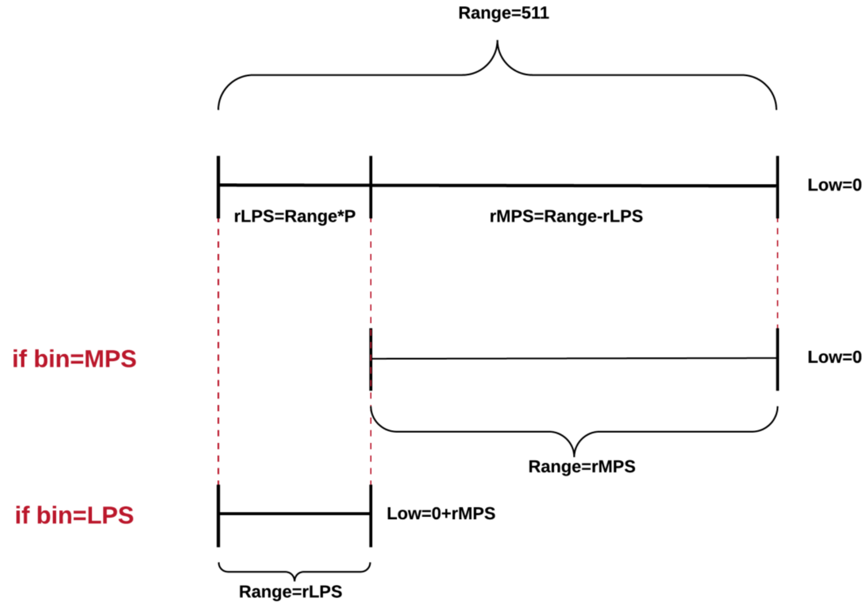

Figure 2.7: Updating the Range and Low variables in BAE

Low is 0 and it is a 10-bit register. rMPS and rLPS represents the two corresponding sub-intervals of MPS and LPS, respectively. If input bin is equal to MPS, rMPS is chosen as the new interval, otherwise rLPS is selected. When the updated Range is found to lie outside the interval 256 and 511 inclusive, a renormalization procedure is employed. The renormalization procedure is where most of the code word is constructed.

The probability state P_{LPS} is needed for computing the rLPS value. This value ranges from 0 to 0.5 and it is quantised to 64 discrete probability states. These states are indexed by a variable pStateIdx ranging from 0 to 63. The transition to next state based on the current bin is shown in Figure 2.8. Hence a table look up is used instead of multiplication to update the probability state.

Another multiplication operation in this stage, the computation of rLPS = Range * R_{LPS}, is also converted to a table lookup. The Range is quantized to four R_Q values. The product rLPS is also quantized to 256 values based on R_Q and pStateIdx. As a result, computing of rLPS can be simply done by looking up in a two-dimension table, in which R Q and pStateIdx are the two indices.

The other two coding methods are elaborated below. The first one is the Bypass coding engine. In this engine, the context modeler stage is bypassed. This means that the previous values

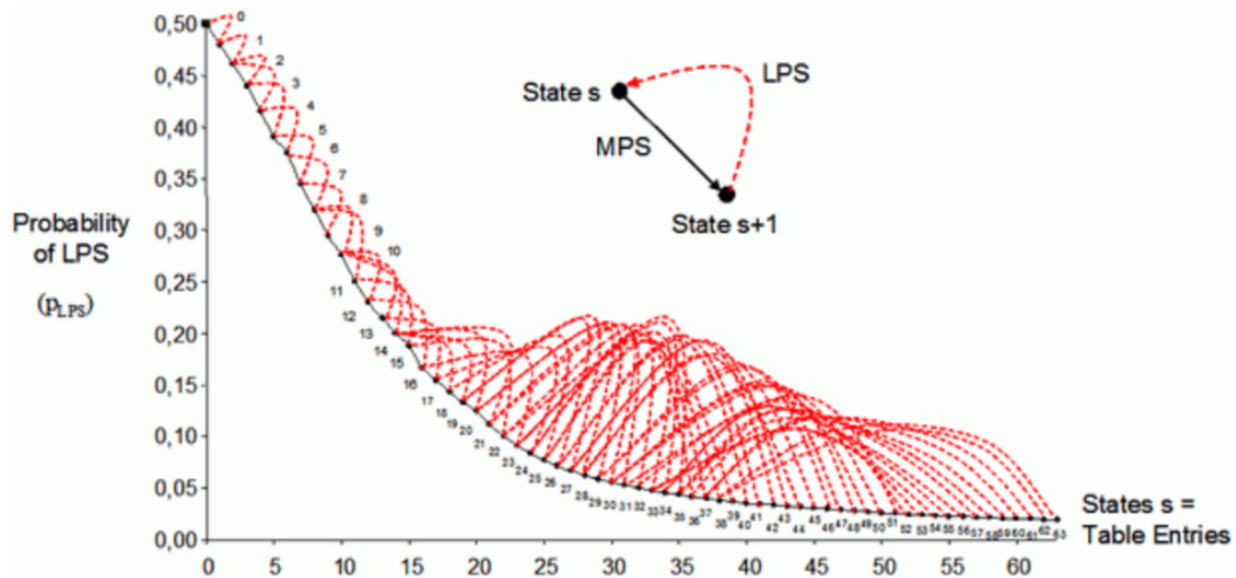

Figure 2.8: Updating the Range and Low variables in BAE

of Range and Low are used and the renormalization for the bypass method is invoked. In the bypass coding engine the probability of the two symbols is considered to be equal to 0.5.

The second alternative coding engine, namely the terminate coding engine, is invoked when the end_of _slice syntax element is encountered or when the mb_type is of the IPCM variety. Here also no context model is chosen. The LPS is fixed to 1 and the rLPS is fixed to 2. Otherwise, the renormalization is the same. When the end of a slice is encountered, then there is also a flushing algorithm that is called.

Every time a new slice begins, an initialize algorithm is invoked which resets all the 399 context models, based on the slice type and the cabac_init_idc value. The slice_QP variable is used to compute the exact context.

2.2.2 Syntax elements

The CABAC coding process uses a total 18 different types of syntax elements which, based on the carried semantics, are divided into 5 categories. This section explains the semantics specified by each syntax element.

mb_type specifies the type of macroblock. The macroblock type in I slice is specified based on the partition scheme of the macroblock in intra prediction. In B slice and P slice, the macroblock

Category	Syntax Element
Macroblock type	mb_type sub_mb_type
Inter Prediction	mvd_lX ref_idx_lX
Intra Prediction	intra_chroma_pred_mode prev_intra4x4_pred_mode_flag rem_intra4x4_pred_mode
Residual Data	coded_block_pattern coded_block_flag significant_coeff_flag last_significant_coeff_flag coeff_abs_level_minus1 coeff_sign_flag
Control flags and parameters	mb_qp_delta mb_field_coding_flag mb_skip_flag end_of_slice_flag

Table 2.1: Syntax element categories for CABAC

type is based on both partition scheme and inter prediction mode of this macroblock.

sub_mb_type specifies sub-macroblock type. The sub-macroblock is only used in the case of B slice and P slice. The sub-mb type is based on both partition scheme and inter prediction mode of this sub-macroblock.

mvd_lX_[mbPartIdx][subMbPartIdx][compIdx] motion vector difference (MVD) is the difference between the motion vector component and its prediction during the process of motion estimation. The value X in mvd lX can be either 0 or 1, designating the reference list used in prediction. list 0 is used for backward prediction and list 1 for forward prediction. mbPartIdx

specifies the index of macroblock partition and subMbPartIdx specifies the index of sub-macroblock partition. compIdx specifies the motion vector component index. For horizontal vector, it is assigned 0 and 1 for vertical vector. The horizontal and vertical MVD are considered to be two separate types of syntax elements.

ref_idx_lX[mbPartIdx] specifies the index of the reference picture in the reference list for motion estimation. The X in ref idx lX is the same as the X in MVD. mbPartIdx is the index of the macroblock partition.

intra_chroma_pred_mode (ICPM) specifies the intra prediction mode for chroma information inside a macroblock.

prev_intra4x4_pred_mode_flag and rem_intra4x4_pred_mode specify intra 4x4 prediction mode for each 4x4 luma block. prev_intra4x4_pred_mode_flag is set to 0 when there is no rem_intra4x4_pred_mode syntax element in the macroblock.

coded_block_pattern (CBP) specifies which of the four 8x8 luma blocks and two 8x8 chroma blocks contain non-zero transform coefficients. The four bits for luma blocks in CBP are called CBP-Luma and the two bits for chroma blocks in CBP are called CBP-Chroma.

coded_block_flag (CBF) is set to 0 when a transform block contains no non-zero transform coefficients. It is set to 1 when the block contains at least one non-zero transform coefficient.

significant_coeff_flag[scanningPos] (SCF) is set to 0 when the transform coefficient level at current scanning position is equal to 0. It is equal to 1 when this position has non-zero transform coefficient level value.

last_significant_coeff_flag[scanningPos] (LSCF) is set to 1 when the following scanning positions within this transform block have all zero values. It is set to 0 when there is at least one non-zero transform coefficient value in the following scanning positions.

coeff_abs_level_minus1[scanningPos] is the absolute value of the transform coefficient level value minus 1 when his position has non-zero value.

coeff_sign_flag[scanningPos] is the sign of the transform coefficient level value at this position.

mb_qp_delta (QPD) specifies the difference between the QP used in the current macroblock and the previous macroblock. The QPD of the first macroblock in each slice specifies the difference between the first macroblock QP and the slice QP.

mb_field_coding_flag is set to 0 when current macroblock pair is a frame coding macroblock pair, and 1 when it is field coding macroblock pair.

mb_skip_flag indicates if current macroblock is skipped or not.

end_of_slice_flag this is set to 0 when this macroblock is not the final macroblock in a slice, is set to 1 when it is the final macroblock of a slice. This is always the final syntax element within the macroblock.

Chapter 3

The KiloCore Many-Core Processor Array Architecture

The body of work that is this thesis, describes the implementation of the Context-based Adaptive Binary Arithmetic Coder on the KiloCore II platform. The KiloCore II is a large array of independent, programmable, single issue, RISC-type processors [4]. Each processor has its own memory module and there are multiple big memory modules on the chip that have dual access to the last row of the processors. A single processor can fan out to up to 8 other independent processors. Although this work was implemented on a fourth-generation KiloCore II architecture, the third-generation KiloCore architecture is functionally identical for the purposes of this work. Thus, this chapter will provide the salient features of the KiloCore architecture and chip.

3.1 Processors

Each processor contains a 128x40-bit instruction memory, 512 Bytes of data memory, three programmable data address generators, two 32x16-bit input buffers, and a 16-bit fixed-point datapath with a 32-bit multiplier output and a 40-bit accumulator. The 72 instruction types include signed and unsigned operations to enable efficient scaling to 32 bit or larger word widths, with no instructions being algorithm-specific. Processors support predication for any instruction using two conditional execution masks, static branch prediction, and automated hardware looping for accelerating inner loops.

Each processor issues one 40-bit instruction in-order per cycle into its seven-stage pipeline

as shown in Figure 3.1.

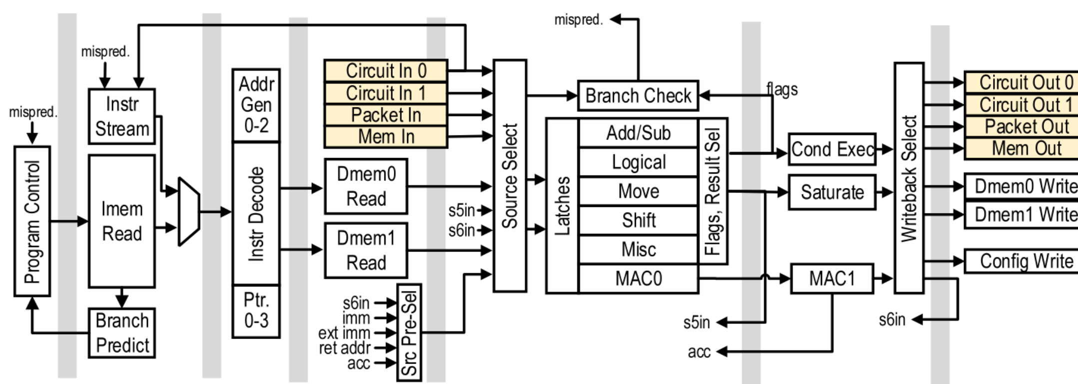

Figure 3.1: KiloCore pipeline [4]

3.2 Memory

Independent memory modules each contain a 64-kByte SRAM and are shared between two neighboring processors. Modules support random and a variety of programmable burst access patterns for data reading and writing, and are also capable of streaming instructions for large-program execution to an adjoining processor using an internal control module. When executing an instruction stream from an independent memory, a processor transfers program control and branch prediction control to dedicated circuits inside the memory block to more efficiently execute across branches. Each memory module contains two 32x18-bit input buffers, two 32x16-bit output buffers, and one 16x2-bit processor response buffer, and supports 28.4 Gb/s of I/O bandwidth.

3.3 Communication between processors

The processor array connects processors and independent memories via a 2-D mesh, a topology which maps well to planar integrated circuits and scales simply as the number of processors per die increases. Communication on-chip is accomplished by two comple- mentary means: a very high throughput and low-latency circuit-switched network and a very-small-area packet router. The circuit-switched links are source-synchronous, so the source clock travels with the data to the destination, where it is translated to the destination-processors clock domain. The network supports communication between ad- jacent and distant processors, as resources allow, with each

link supporting a maximum rate of 28.5 Gb/s with optionally inserted registers to maintain data integrity over long distances. Each of the four edges of each processor has two such links entering and two links exiting the processor. The high-throughput circuit-switched network is especially efficient transferring data to an adjacent processor, dissipates 59% less energy than writ- ing and later reading that data using local data memory, and transferring that data to a processor four tiles away requires only 1% more energy than using local data memory. The packet router inside each processor occupies only 9% of each processors area and is especially effective for high fan-in and high fan-out communication, as well as for admin- istrative messaging. Each router supports 45.5 Gb/s of throughput with a maximum of 9.1 Gb/s per port. Routers operate autonomously from their host processors and contain their own clock oscillators, so they can power down to zero active power when there are no packets to process. Each router contains five 4x18-bit input buffers, one for each cardinal direction and one for the local processor. Routers utilize wormhole routing to efficiently transfer long data bursts, in which a header packet will reserve a path and is followed by an arbitrary number of data packets, terminating in a tail packet which releases the path.

The clocking system in the KiloCore is governed by a Globally Asynchronous Locally Synchronous (GALS) clocking style, which separates processing blocks such that each part is clocked by an independent clock domain [12].

3.4 Project Manager

The Project Manager provides support for writing task parallel applications for the KiloCore I/II style of many-core array, preparing applications to run on a target architecture, and launching simulations to verify correctness and gather measurements. Apps are set up by a python script that will import and apply the various Project_Manager functions and classes.

An Application object is the basic repository for all application information, including user defined setup, annotations and simulation results. Tasks are defined which make up the compute part of the application. Common tasks include Processor, Memory, Input_Handler, Output_Handler. Links between cores connect indexed output ports to indexed input ports of the tasks. Links include Circuit_Link, Memory_Link, and Packet_Link. Input source files need to added to the project. Code files may be C++ (to be compiled to assembly) or simulator formatted assembly. Data files provide test input data or reference output data, with some optional formatting support to translate them

into the expected data width of the architecture.

The Asap3 and Asap4 arrays have been predefined architecture targets in the Project Manager. Simulator related options are also accessible. Finally transforms are run to get analysis information of the application like simulation time, throughput, energy, branch accuracy, utilization information and if applied, simulation reference accuracy.

Chapter 4

Methodology

4.1 Overview

The implementation of CABAC on the AsAP array involved breaking up the algorithm into smaller tasks that could be performed independently of the other tasks.

The data flow for the whole application is given in Figure 4.1.

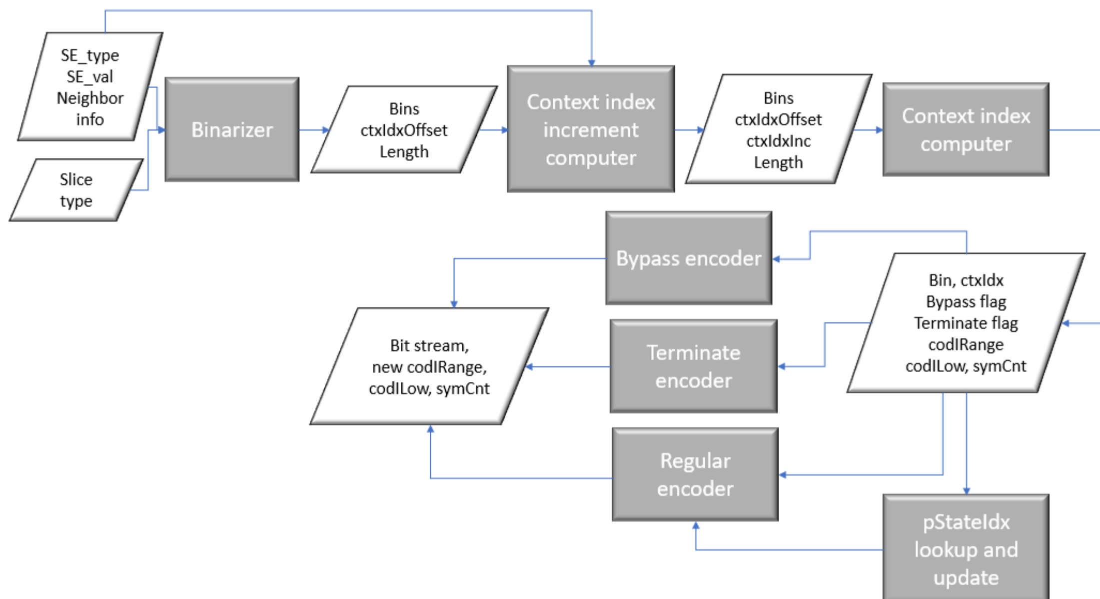

Figure 4.1: CABAC data flow

The grey boxes represent a block within the CABAC. The white parallelogram-shaped boxes represent the variables that are processed by the grey boxes. These variables are either inputs

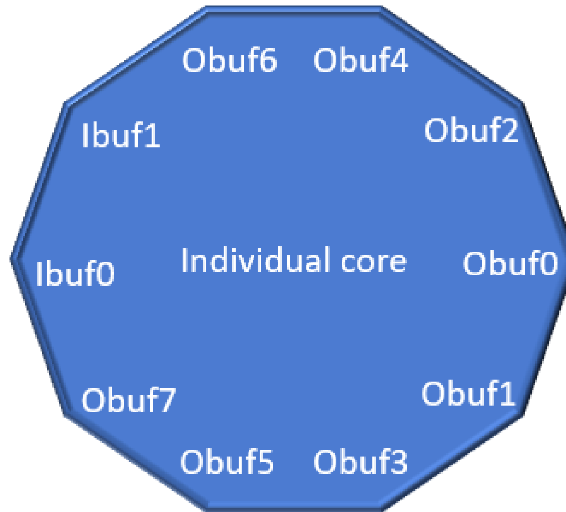

Figure 4.2: Representation of a single core's ports.

to the system or processed outputs.

The flow of bits starts at the input handling cores. These cores process the input that is received by the JM software. These cores are also responsible for fanning out the syntax element information to the binarizer cores. All the binarizer cores have their own context index increment cores as well as context index calculator cores.

Two context index calculator cores connect to a single bin accumulator core which fans in the data for further processing. There are three stages of the bin accumulator cores. The purpose of fanning in at this stage is due to the fact that all the cores have to access the same set of context tables. This brings us to the last section of the CABAC algorithm which is the Binary Arithmetic encoder.

The Binary Arithmetic encoder (BAE) takes in all the bins associated with a single syntax element, and processes each bin individually, updating the context tables after each bin. The last function of the BAE block is the renormalization and symbol count updating. The BAE block is also responsible for packing the bits into words of 16 bits. The output written to a text file consists is the bin stream of the words that were produced by the last stage of the BAE.

This chapter elaborates the working of the implementation starting with the input handling and discussing each block of the CABAC algorithm. The representation of a core will look like Figure 4.2. The color representation for each block is given in Figure 4.3.

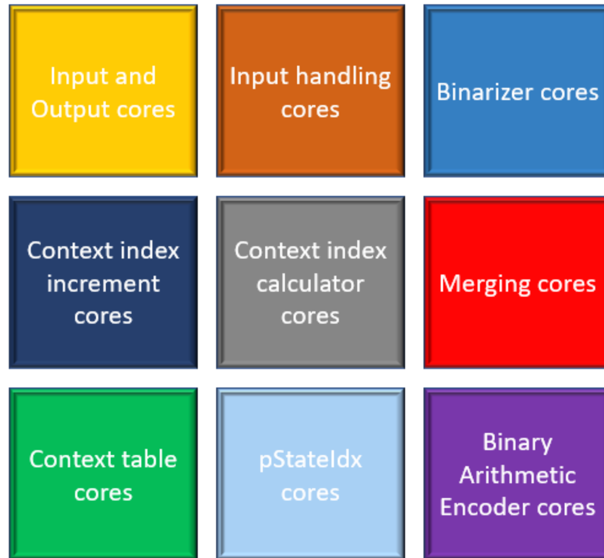

Figure 4.3: Colors of various stages of cores

4.1.1 Input data handling

The input handling cores receive input from the JM software. Each line of the input represents one syntax element. The Table 4.1 shows the data that is received for each syntax element.

1.	Initialise enable signal	9.	Intra Chroma Pred Mode for B and A
2.	Syntax element value	10.	Motion vectors for B and A
3.	Syntax element type	11.	ref_idx_lX for B and A
4.	Indices of x and y coordinates	12.	mb_type for B and A
5.	List variable for ref_idx_lX	13.	Prediction direction for ref_idx_lX
6.	Variable ctxBlockCat	14.	mbSkip flags for B and A
7.	Coded block flag	15.	mbField flags for B and A
8.	Coded block pattern for B and A	16.	mbAvailable flags for B and A

Table 4.1: Input varaibles from JM software

Not all the values received for each syntax element is relevant information. The input information is distributed to more specific cores in the *slice_mb_storage* cores. These extract the relevant data for each element and pass it on to specific context modeling cores.

This subsection elaborates on the working of the following cores:

- I_0 *input_reader*: uses one input port and five output ports.
- I_1 *input_slice*: uses one input port and one output port.
- I_2 *bin_distributor*: uses both input ports and six output ports.
- I_3 *slice_mb_storage*: uses both input ports and one output port.
- I_4 *slice_mb_storage_2*: uses one input port and four output ports.

The CABAC implementation on KiloCore II has two external input handlers. The first handler named *input_reader* is utilised for the Syntax Element information and the second one named *input_slice* is used for the Slice information.

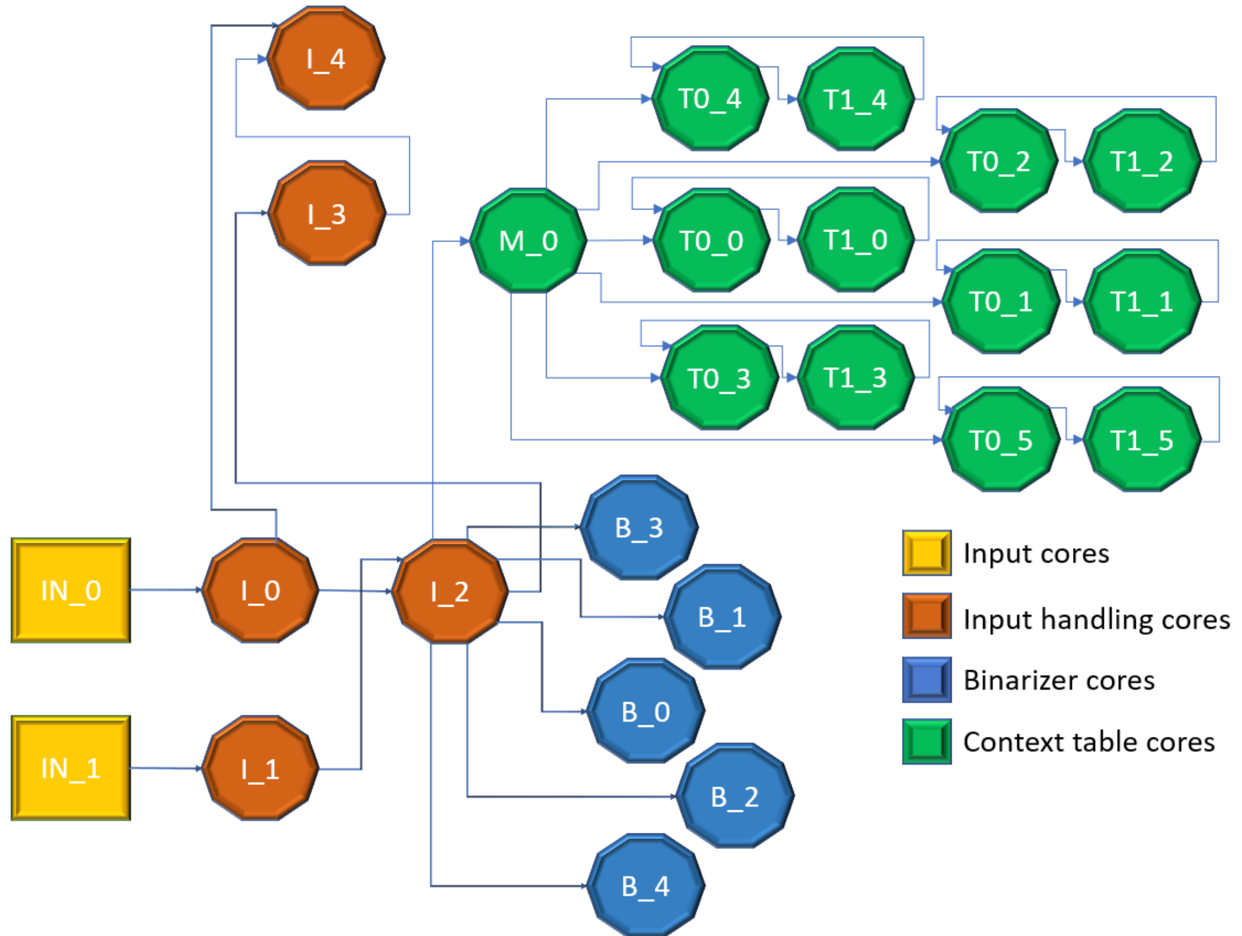

Figure 4.4: Complete input and output connections of the core *bin_distributor*(I_2). I_0: *input_reader*, I_1: *input_slice*, I_3: *slice_mb_storage*, B_0 to B_4: binarizer cores

The *input_reader* core also deals with processing the neighbor information that it receives along with the Syntax Element data. The *input_reader* core has three output circuit links connected to context index increment calculating cores discussed in 4.3. Another output link is connected to the core, *bin_distributor*, that is responsible for collecting the syntax element information and the slice information from the *input_slice* core and passing it forward to the binarizer cores, which are *binarizer_mbtypeIP*, *binarizer_mbtypeB*, *binarizer_submbtype_fixed*, *binarizer_tunary* and *binarizer_uegk*. The Figure 4.1 shows the connections to the *bin_distributor*.

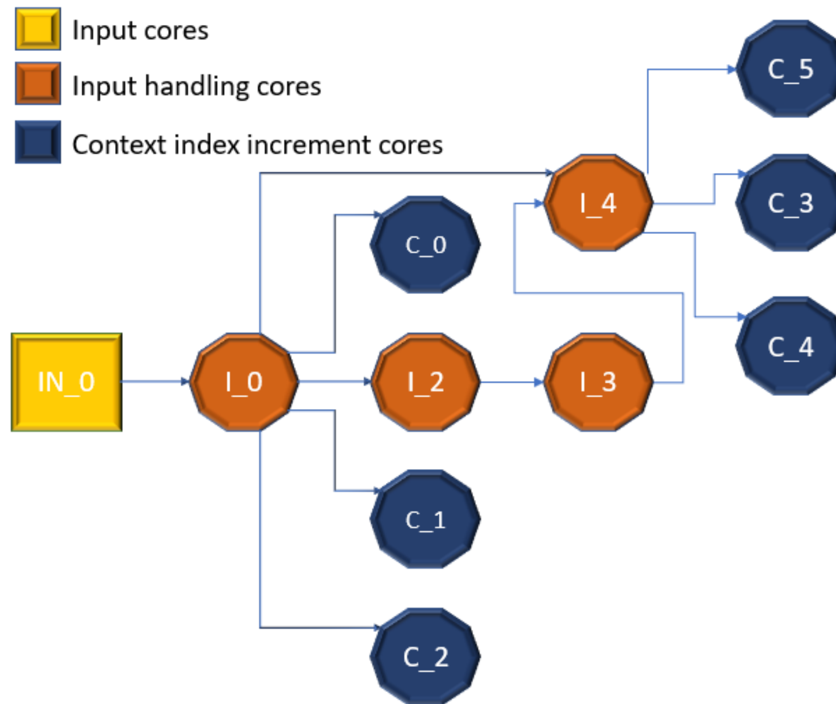

Figure 4.5: Complete input and output connections of the core *input_reader* (I_0) and *slice_mb_storage* (I_3). I_2: *bin_distributor*, C_0: *ctxIdxInc_mbtype*, C_1: *ctxIdxInc_fixed*, C_2: *ctxIdxInc_blockcat*, C_3: *ctxIdxInc_tunary*, C_4: *uegk_parser*

The last output circuit link goes to *slice_mb_storage* core, which stores the relevant information about the current slice and current macroblock. The *slice_mb_storage* core is also responsible for parsing neighbor information like the *input_reader* core. Hence, it also has output links to context index increment calculator cores. The Figure 4.2 shows the input and output connections for *input_reader* and *slice_mb_storage* cores.

The macroblock information that the *slice_mb_storage* core stores includes values of syntax elements that have a set value for the given macroblock. The syntax elements in question are the

mb_type, mb_skip_flag, sub_mb_type, mb_field_decoding_flag, mb_qp_delta. It is also responsible for maintaining the scanning position, numLevelGt1, numLevelEq1 counters. These counters are required in the encoding of Residual syntax elements.

The slice information it stores is the slice_type, frame_coded_flag, MBaffFrameFlag, slice_QP and cabac_init_idc.

4.2 Binarizer

Each binarizer core deals with one or more method of converting the syntax elements into bins, as specified by the standard [5], based on the syntax element type. The algorithms for doing that are elaborated in this section.

The Binarizer consists of the following cores:

- B_0 *binarizer_mbtypeIP*: uses one input port and one output port.
- B_1 *binarizer_mbtypeB*: uses one input port and one output port.
- B_2 *binarizer_submbtype_fixed*: uses one input port and one output port.
- B_3 *binarizer_tunary*: uses one input port and one output port.
- B_4 *binarizer_uegk*: uses one input port and one output port.
- B_5 *sub_mb_type_table*: uses one input port and two output ports.
- B_6 *uegk*: uses one input port and one output port.

The input to this section is the syntax element value, the syntax element type and the slice type.

The output consists of the bins created, ctxIdxOffset for each type of syntax element, the length of the bins and the length of the suffix word, if that is applicable. The ctxIdxOffset for each syntax element is elaborated in Table 4.2. The binarizer core that processes each syntax element is also mentioned in Table 4.2. [3]

For the mb_type syntax element part of the P and B slices, the ctxIdxOffset is assigned based on whether the bin index is part of the prefix or suffix. The ctxIdxOffsets for the significant_coeff_flag and last_significant_coeff_flag change depending on whether it is field coding or frame coding.

Syntax Element type	Slice type	ctxIdxOffset	Binarizer core
mb_type	I	3	*binarizer_mbtypeIP*
mb_type	P	14/17	*binarizer_mbtypeIP*
mb_type	B	27/32	*binarizer_mbtypeB*
mb_skip_flag	P	11	*binarizer_submbtype_fixed*
mb_skip_flag	B	24	*binarizer_submbtype_fixed*
sub_mb_type	P	21	*binarizer_submbtype_fixed*
sub_mb_type	B	36	*binarizer_submbtype_fixed*
mvd_l0	all	40	*binarizer_uegk*
mvd_l1	all	47	*binarizer_uegk*
ref_idx_lX	all	54	*binarizer_tunary*
mb_qp_delta	all	60	*binarizer_tunary*
intra_chroma_pred_mode	all	64	*binarizer_tunary*
prev_intra4x4_pred_mode_flag	all	68	*binarizer_submbtype_fixed*
rem_intra4x4_pred_mode	all	69	*binarizer_submbtype_fixed*
mb_field_coding_flag	all	70	*binarizer_submbtype_fixed*
coded_block_pattern	all	73	*binarizer_submbtype_fixed*
coded_block_flag	all	85	*binarizer_submbtype_fixed*
significant_coeff_flag	all	105/277	*binarizer_submbtype_fixed*
last_significant_coeff_flag	all	166/338	*binarizer_submbtype_fixed*
abs_coeff_level_minus1	all	227	*binarizer_uegk*
coeff_sign_flag	all	0	*binarizer_submbtype_fixed*
end_of_slice_flag	all	276	*binarizer_submbtype_fixed*

Table 4.2: Values of ctxIdxOffset for each syntax element type

As previously mentioned in the input handling section, the *bin_distributor* core sends the syntax element and slice information to all the binarizer cores. Of these cores, the *binarizer_submbtype_fixed* and *binarizer_uegk* ones are connected to the *sub_mb_type* and *uegk* cores respectively. The reason for this is to spilt these binarization function to fit into 128 instructions available in each core.

4.2.1 Tabular lookup coding

The syntax elements that use tabular lookup coding are the mb_type and sub_mb_type. The tables are divided based on the macroblock type, whether it is I type, P type or B type. When the macroblock is P or B type, there is also a suffix that is added to the coded word based on the syntax value. These tables have the length of the bins stored in the tables. The maximum length of the bins is 13, which is fewer bits than the size of a word in the KiloCore Architecture [4].

Syntax Element mb_type

The mb_type syntax element has 26 values for the I macroblock, 31 values for the P macroblock, 25 of which have a suffix and 49 values for the B macroblock, again 25 of which have a suffix.

The bins of the I and P macroblocks are stored in tables in the *binarizer_mbtypeIP* core. The bins of the B macroblock are stored in the *binarizer_mbtypeB* core.

Syntax Element sub_mb_type

The sub_mb_type syntax element only appears in P and B macroblocks. There are 4 table entries for the P macroblock and 13 entries for the B macroblock.

The tabular lookup coding for the sub_mb_type syntax element is split across two cores, namely the *binarizer_submbtype_fixed* and the *sub_mb_type_table*. The first core mainly deals with the syntax elements that have fixed length coding discussed in subsection 4.2.2. The second core has the tabular values for the sub_mb_type.

4.2.2 Fixed Length coding

The syntax elements that follow the fixed length coding are:

- mb_skip_flag,
- prev_intra4x4_pred_mode_flag,
- rem_intra4x4_pred_mode,
- mb_field_coding_flag,
- coded_block_pattern,
- coded_block_flag,
- significant_coeff_flag,
- last_significant_coeff_flag,
- end_of_slice_flag,
- coeff_sign_flag.

The syntax element coded_block_pattern follows both the fixed length coding for the lower four bits and truncated unary coding for the upper two bits, which is explained further in subsection 4.2.4. All these above mentioned syntax elements are dealt with in the *binarizer_submgtype_fixed* core.

The fixed length has a variable cmax which determines the length of the coded bins. The formula for finding the length from cmax is:

$$length = \lceil log_2(cmax + 1) \rceil \tag{4.1}$$

The rem_intra4x4_pred_mode syntax element has cmax value equal to 7. The lower four bits of the syntax element coded_block_pattern has a value of cmax equal to 15. The remaining syntax elements that follow fixed length coding all have the fixed length of 1. All the syntax elements of this type produce bins that can easily fit in a single word of the KiloCore. [13]

For the elements that have cmax equal to 1, it is a matter of simple assignment to get the coded bin. The elements with higher cmax follow a method of bit shifting shown in Equation 4.2.

$$bins = \sum_{i=0}^{length-1} ((SE_val >> (length - (i + 1)))\&1) << i \tag{4.2}$$

An example of fixed length coding is shown in Table 4.3

Syntax Element value	Fixed length coded bins
1	100
2	010
3	110
4	001
5	101
6	011
7	111

Table 4.3: Example of fixed length coding when cmax = 7 [6]

4.2.3 Unary coding

The syntax elements that follow the unary coding are ref_idx_lX, mb_qp_delta. These are handled in the *binarizer_tunary* core.

In the unary coding method, the bins produced include SE_val number of ones followed by a zero. Thus, the syntax element value determines the length of the bins, given by Equation 4.3.

$$length = SE_val + 1 \tag{4.3}$$

The range of values of the ref_idx_lX syntax element is 0 to 31 inclusive. The maximum length of the bins coded is thus 32.

The range of values that the syntax element mb_qp_delta can take is from -26 to 25. The values associated with this syntax element are first mapped to unsigned values [1] ranging from 0 to 52 inclusive before the unary coding method is applied. That makes the maximum length of the bins coded to be 53.

The size of a word in the KiloCore is 16 bits [4]. Therefore, we need a total of four words to fully represent the bins produced by this coding technique. This is the first binarization coding method where we encounter bins larger than a single word in the KiloCore.

The mapping of the signed values of the mb_qp_delta elements is governed by Equation 4.4

that is shown here.

$$map_value = \begin{cases} (SE_val << 1) + 1, SE_val > 0 \\ |SE_val| << 1, SE_val < 0 \\ 0, otherwise \end{cases} \quad (4.4)$$

4.2.4 Truncated unary coding

The syntax elements using the truncated unary method are intra_chroma_pred_mode, coded_block_pattern, and prefixs of mvd_l0, mvd_l1 and abs_coeff_level_minus1. These elements are handled in *binarizer_tunary*, *binarizer_submbtype_fixed* and *binarizer_uegk* cores.

The truncated unary method is very similar to the unary coding method. The difference is that for syntax elements having values greater than a variable cmax get truncated to cmax number of bits. This code word consists of cmax number of ones.

The cmax values for all the syntax elements encoded in this manner is listed in Table 4.5.

Syntax Element type	cmax for Truncated unary coding
intra_chroma_pred_mode	3
coded_block_pattern prefix	2
mvd_l0 prefix	9
mvd_l1 prefix	9
abs_coeff_level_minus1 prefix	14

Table 4.4: cmax values for truncated unary coding

The bins, length and ctxIdxOffset of the Syntax elements are forwarded to the next section of context modeler cores.

4.2.5 Unary Exponential Golomb k-th order coding

The syntax elements that Unary Exponential Golomb k-th order coding (UEGk) are mvd_l0, mvd_l1 and abs_coeff_level_minus1. These are dealt with in the *binarizer_uegk* and *uegk* cores.

The parameters associated with the UEGk coding are shown in Table 4.4. These values are initialised in the *binarizer_uegk* core. The computation of the bins is done in the *uegk* core.

Syntax Element type	k	uCoff	signedValFlag	Range
mvd_l0	3	9	1	[-2048, 2048)
mvd_l1	3	9	1	[-512, 512)
abs_coeff_level_minus1	0	14	0	$[0, 2^{15} - 1]$

Table 4.5: Parameters for UEGk coding

As already stated in 4.2.4, the prefix for these syntax elements are coded using the Truncated coding method. The process of coding the syntax element prefix is detailed in Algorithm 1.

The suffix of the Syntax elements coded using the UEGk method is more complicated than the prefix. Whether the incoming syntax element is positive or negative is to be considered for the coding. A variable called sufS is made use of in the calculation. The size of sufS is determined based on the range of values of the syntax element. The ranges of mvd_l0, mvd_l1, abs_coeff_level_minus1 are given in Table 4.5.

Algorithm 1 Prefix of UEGk coding

if *SE_val is less than uCoff* **then**
| Assign prefix equal to SE_val number of bits all equal to 1 followed by bit 0;
else
| Assign prefix equal to uCoff number of bits all equal to 1;
end
if *signedValFlag is equal to zero and SE_val is less than uCoff or signedValFlag is equal to 1 and prefix is equal to 0* **then**
| Output is equal to only prefix;
end

From the ranges table, it can be determined that the maximum size of prefix for mvd_l0 and mvd_l1 is 9 and maximum size for prefix of abs_coeff_level_minus1 is 14. The maximum size of suffix for mvd_l0 and mvd_l1 is 19 bits and the maximum size of suffix for abs_coeff_level_minus1 is 29 bits. This means that the variable sufS has be larger than 29 bits. In the KiloCore II architecture,

the int32_t register is used for the purpose.

The complete algorithm for coding the suffix is detailed in Algorithm 2.

Algorithm 2 Suffix of Unary Exponential Golomb kth-order coding

---

```
Reset signFlag to 0;
if SE_val is less than zero then
    Set signFlag to 1;
    Assign the absolute value of SE_val to SE_val;
end
Initialize suffix to be 0;
if SE_val is greater than or equal to uCoff then
    Assign the difference betwenn SE_val and uCoff to sufS;
    Initialize stoploop to be 0;
    while stoploop is not equal to 1 do
        if sufS is greater than or equal to 2^k then
            Add bit 1 to right end of Suffix;
            Subtract 2^k from sufS;
            Increment k;
        end
        else
            Add bit 0 to right end of suffix;
            while value of k decremented by 1 is not equal to zero do
                Shift sufS to the right k times;
                Addend the least significant bit of the shifted value to the right end of suffix;
            end
            Set stoploop to equal 1;
        end
    end
end
```

The maximum length of the bins produced is 43 bits. The bins produced is divided into three 16 bit words, because the coded word will not fit in even the 32-bit register available. The prefix bits must occupy the most significant bits of the first 16-bit word sent to the next core.

Three cases are formed for the division of the prefix and suffix into the three 16-bit registers.

The first case is when all the bits will fit into a single word. This condition is satisfied only when length of suffix is less than difference between 16 and the length of prefix. In this case we shift the prefix bits to the left by length of suffix and bit wise OR is performed between the shifted prefix and the suffix. Figure 4.5 shows the register in this case.

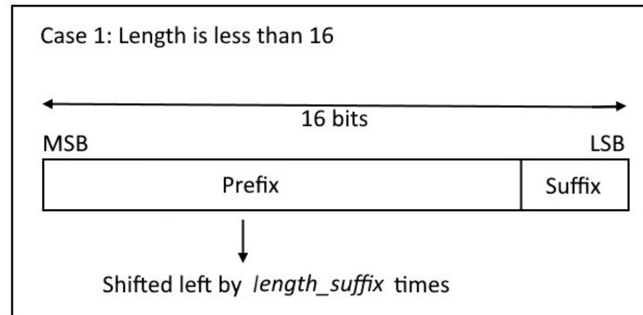

Figure 4.6: Prefix and Suffix words in 16-bit register: Case 1

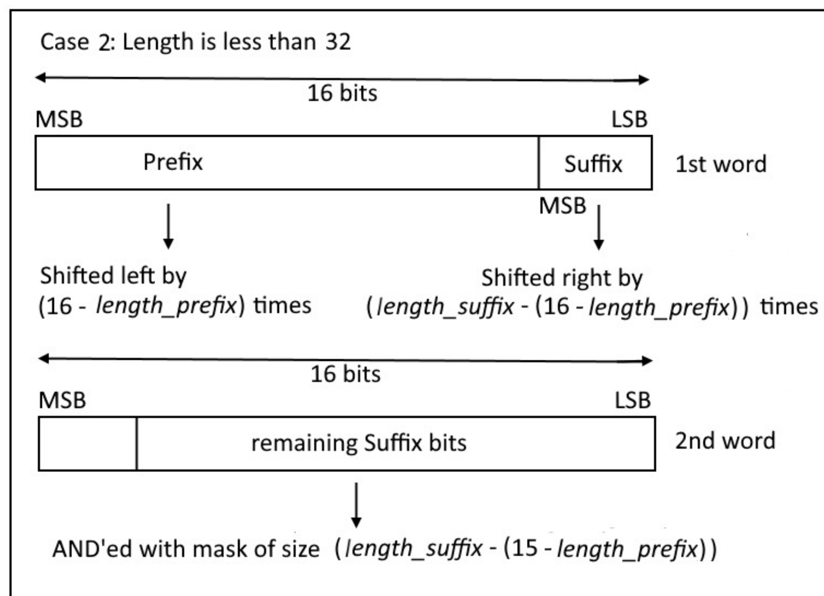

Figure 4.7: Prefix and Suffix words in 16-bit register: Case 2

The second case is when the total length is less than 32 bits, this means that two registers are sufficient to store the complete word. In this case, the first word contains the prefix shifted to the left by total length of register (16) - length of prefix bits. The remaining bits are occupied by the most significant bits of the suffix. So, the suffix is shifted right by length of suffix - (total

register length - length of prefix). The second 16 bit word has the remaining suffix bits. A mask is constructed of the remaining length of the suffix. A bit-wise AND operation is performed between this mask and the suffix.

The last case is the one where the length of the coded word is larger than two registers. The shifting for the first 16-bit register is the same as the second case. The prefix bits shifted to the leftmost and the remaining bits are made up of the most significant suffix bits.

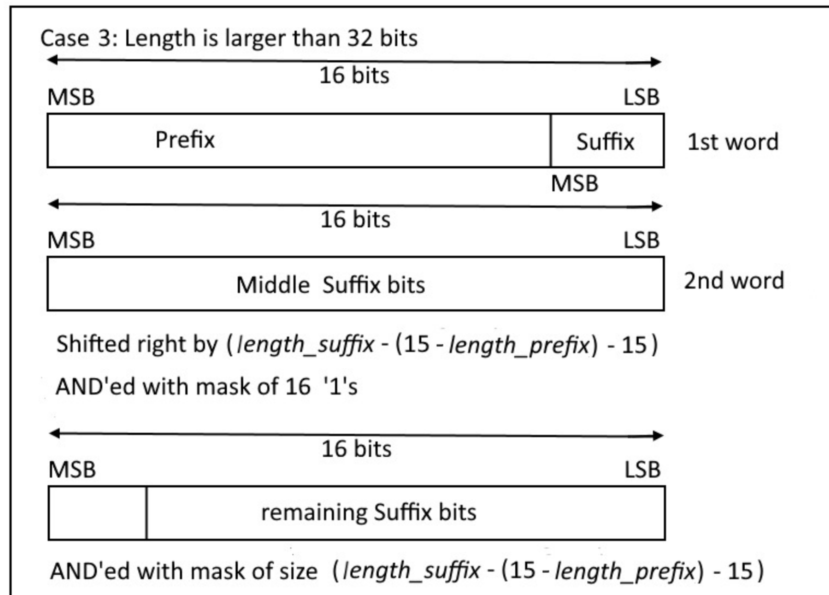

Figure 4.8: Prefix and Suffix words in 16-bit register: Case 3

The second word consists of the middle part of the suffix. We can be certain that the prefix will not extend to this word, since the maximum length of the prefix is 14. The middle of the suffix is got by shifting out the right most bits of the suffix that should appear in the third word. So the suffix is shifted right by length of the suffix subtracted by how much was in the first word and by how much should be in the second word, which is the full size of the word, that is 16.

$$shift_value = length_suffix - (16 - length_prefix) - 16 \tag{4.5}$$

The shift value is given by the Equation 4.5. The middle word should also masked in order to remove the left most bits that would have been part of the first word. So the shifted suffix and a mask of sixteen ones are inputs to a logical AND gate.

The third word is the remaining bits of the suffix. The suffix and a mask of length

shift_value given in 4.5 are applied to a bit-wise AND gate to get the remaining bits.

When there is a SE_type is a signed value, it is coded as shown below in Algorithm 3.

Algorithm 3 Sign bit of UEGk coding

if *signedValFlag is equal to 1 and SE_val is not equal to zero* **then**
  if *signFlag is equal to 0* **then**
    Add bit 0 to the right end of suffix;
  else
    Add bit 1 to the right end of suffix;
  end
end

Once the prefix, suffix and sign bit are correctly coded into bins, the bins, along with the length of the full word and the length of suffix are forwarded to the next section.

4.3 Context Modeler

The context modeler is explained in this subsection. Figure 4.9, shows the flow of the information in the block. As can be seen in Figure 4.9, for each bin of the bins produced, there is a ctxIdxInc and ctxIdx generated. The same ctxIdxOffset is assigned for each set of bins produced. The ctxIdxBlockOffset is based on the ctxIdxBlockCat. This is further elaborated in the subsection 4.3.2.

The context modeler requires the following cores:

- C_0 *ctxIdxInc_mbtypeIP*: uses both input ports and one output port.
- C_1 *ctxIdxInc_mbtypeB*: uses both input ports and one output port.
- C_2 *ctxIdxInc_fixed*: uses both input ports and one output port.
- C_3 *ctxIdxInc_blockcat*: uses both input ports and one output port.
- C_4 *ctxIdxInc_tunary*: uses both input ports and one output port.
- C_5 *uegk_parser*: uses one input port and one output port.
- C_6 *ctxIdxInc_uegk*: uses both input ports and one output port.

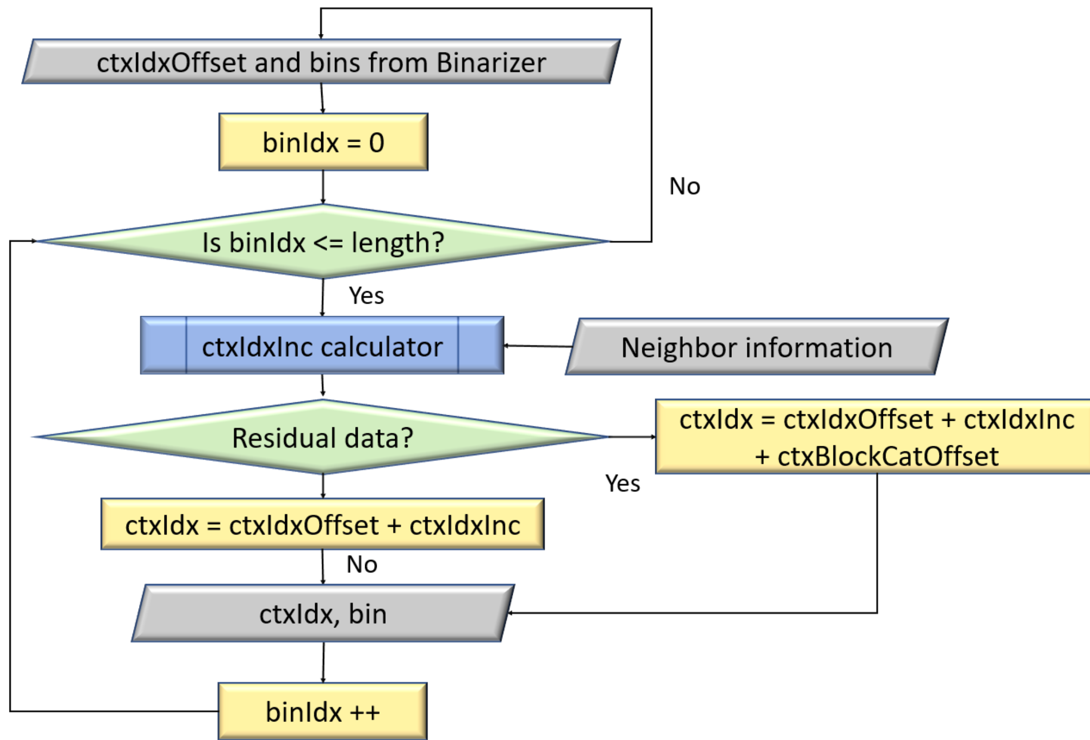

Figure 4.9: Flowchart of the Context modeler block

- C_7 *ctxIdx_mbtype*: uses one input port and one output port.
- C_8 *ctxIdx_fixed*: uses one input port and one output port.
- C_9 *ctxIdx_blockcat*: uses one input port and one output port.
- C_10 *ctxIdx_tunary*: uses one input port and one output port.
- C_11 *ctxIdx_uegk*: uses one input port and one output port.
- J_0 *bin_join*: uses both input ports and one output port.

The input to this block is the ctxIdxOffset, the bins generated by the binarizer cores, the length of the bins and the suffix. The relevant neighbor information is also passed on as input to this block. While the bins and associated information are passed to the context index increment cores through the binarizer cores, the neighbor information is sent from the input handling cores, *input_reader* and *slice_storage_2*. The output of this block is the context index (ctxIdx), the bins to be encoded and the length of the bins.

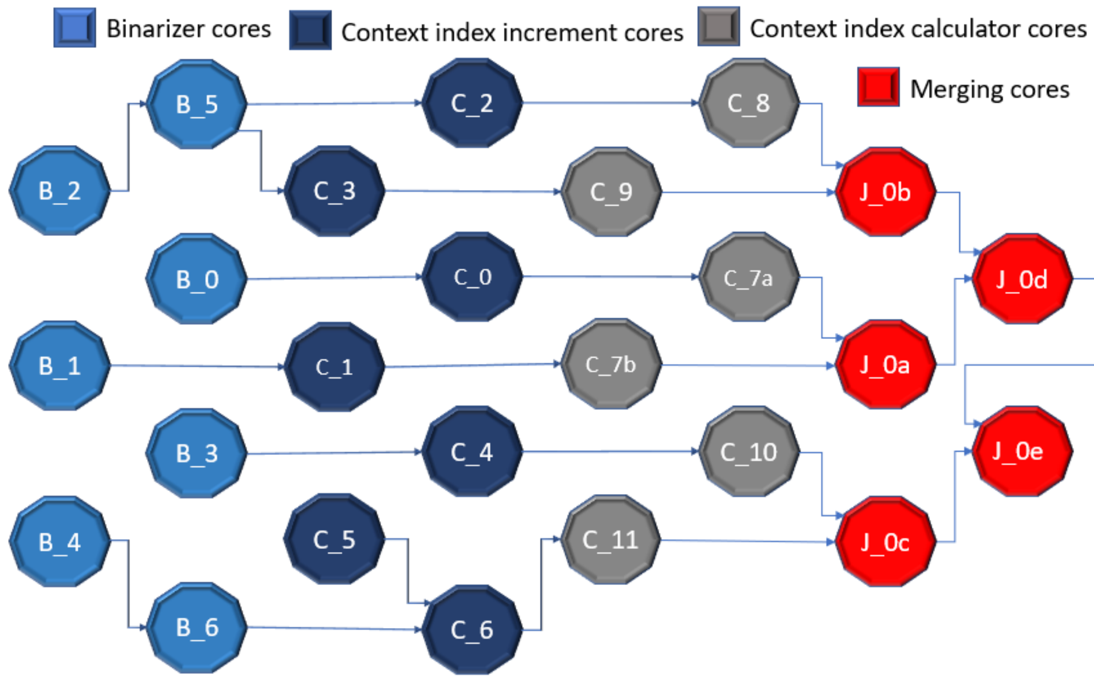

Figure 4.10: Connections for the Context modeler block. B_0 to B_6 are the binarizer block cores. C_0 to C_5 are the context index increment cores. C_6 to C_10 are the context index calculator cores. J_0 cores are the *bin_join* cores.

The first section of the Context modeler is the calculation of the context index increment (ctxIdxInc) based on the context index offset (ctxIdxOffset). The ctxIdxInc is computed based on neighbor partition information in the case of mvd_l0, mvd_l1, coded_block_pattern and ref_idx_lX syntax elements. The residual syntax elements, that is coded_block_flag, significant_coeff_flag, last_significant_coeff_flag and abs_coeff_level_ minus1 all make use of the scanning position of the present macroblock to compute the ctxIdxInc. The syntax elements mb_type, mb_skip_flag, sub_mb_type, mb_qp_delta, intra_chroma_pred_mode, mb_field_decoding_flag all rely on A and B neighbor information in order to compute the ctxIdxInc. The A and B neighbors are shown in diagram below.

Once the ctxIdxInc is computed the ctxIdx is calculated using this and ctxIdxOffset and in some cases the value of ctxIdxBlockCat.

Each method is detailed in the subsections below. Connections for the context modeler cores is shown in Figure 4.10

4.3.1 Context index increment computation cores

In CABAC, the current macroblock's context index increment is computed using neighbor information. The neighbors, as shown in Figure 4.11, that are relevant are the B and A neighbors.

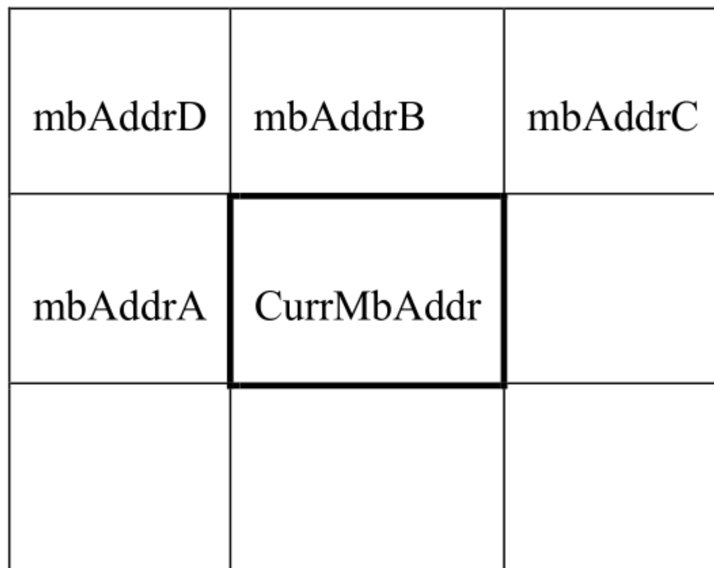

Figure 4.11: Neighbors in reference to current macroblock. [5]

The *input_reader* core parses and sends neighbor information to the *ctxIdxInc_mbtype* and *ctxIdxInc_fixed* cores. The *slice_storage_2* core sends neighbor information to the *ctxIdxInc_blockcat*, *ctxIdxInc_tunary*, *ctxIdxInc_uegk* cores. The list shows what kind of neighbor information each syntax element requires. It also elaborates the calculating of the ctxIdxInc in each case.

mb_type mbAvailB, mbAvailA, mbTypeB, mbTypeA.

There is only one instance of mb_type syntax element in a macroblock. Hence, the neighbor information is the actual neighbor as shown in Figure 4.9. For I slice, the value of ctxIdxOffset is equal to 3. In this case, there are four bins that have a special valued ctxIdxInc. When bin index (binIdx) is zero, ctxIdxInc is computed using Equation 4.6.

$$ctxIdxInc = condTermFlagA + condTermFlagB \tag{4.6}$$

The values condTermFlagA and condTermFlagB refer to the neighbors A and B. The values of condTermFlag are set to zero either if that neighbor is not available or if it is available and equal to I_4x4 (SE_val equal to 0). Otherwise the value is set to one.

When binIdx is equal to one, the value of ctxIdx is directly set to 276. When binIdx is equal to four and five depend on the value of bin at binIdx 3.

When the slice is of P type, the bins either belong to the prefix of suffix, getting different ctxIdxOffset or either 14 or 17. Here, again the value of ctxIdxInc depends on the value of bin at binIdx one and three.

Similarly, for B slice, there is a prefix word with ctxIdxOffset of 27 and suffix word with ctxIdxOffset of 32. When binIdx is equal to one for the prefix, the ctxIdxInc is computed according to Equation 4.6. The value of condTermFlag is set to zero if either the neighbor is unavailable or if it equal to B_Direct_16x16 (SE_val equal to 0).

mb_skip_flag mbAvailB, mbAvailA, mbSkipB, mbSkipA

Here, ctxIdxOffset is equal to 11 or 24 based on slice type. Equation 4.6 is used to find ctxIdxInc. If the neighbor macroblock is unavailable or it is skipped, the condTermFlag is set to zero. Otherwise one. In this case as well there is only one instance of mb_skip_flag syntax element in a macroblock.

sub_mb_type None

The ctxIdxOffset for slice type P is 21. The ctxIdxOffset for slice type B is 36. In this case the bins at binIdx one and three are used to compute the ctxIdxInc.

mvd_l0 and mvd_l1 idx_x, idx_y, list, mvdB, mvdA, mbFieldB, mbFieldA, mbAvailB, mbAvailA, curr_mbtype, curr_submbtype, slice_type, currMVD, MBaffFrameFlag, compIdx, part_idx, MBpartition, curr_mbfield

The ctxIdxOffset for mvd_l0 is 40 and mvd_l1 is 47. The ctxIdxInc calculation for motion vector syntax elements is dealt with in two cores, *uegk_parser*, *ctxIdxInc_uegk*. The first *uegk_parser* core, where the top and left neighbor indices is determined. Also whether these neighbors lie within the same macroblock or are neighbor macroblocks as specified in Figure 4.11.

In the case of mvd_lX syntax elements, the neighbors B and A may be within the same macroblock or may be one of the previous macroblocks as shown in the Figure 4.11. The partitions of the macroblock and the sub-partitions of the submacroblock play a key role in determining the neighbors B and A. There are a few cases based on the type of macroblock being coded and if there

are any submacroblocks within the macroblock.

Slice	mbtype	MB partition	Slice	mbtype	MB partition
I	[0, 25]	16x16	B	0	Direct_16x16
P	0	16x16	B	[1, 3]	16x16
P	1	16x8	B	Even values in [4, 21]	16x8
P	2	8x16	B	Odd values in [4, 21]	8x16
P	3	8x8	B	22	8x8
P	[5, 30]	P_skip	B	[23, 40]	B_skip

Table 4.6: Macroblock partition categories

Only when the slice type is P or B do we encounter this syntax element. The number of motion vectors depends on the current macroblock type, whether it is paritioned into sub-macroblocks and the sub-partitions of these sub-macroblocks. The macroblock paritions are given in Table 4.6. Each partition, if it is not further divided, has its own horizontal and vertical motion vectors.

Slice	submbtype	sub-MB partition	Slice	submbtype	sub-MB partition
P	0	8x8	B	0	Direct_8x8
P	1	8x4	B	[1, 3]	8x8
P	2	4x8	B	Even values in [4, 9]	8x4
P	3	4x4	B	Odd values in [4, 9]	4x8
			B	[10, 12]	4x4

Table 4.7: Sub-macroblock sub-partition categories

When the macroblock partition is 8x8, then it can be further divided into sub-macroblocks. The types of sub-partitions of the sub-macroblocks are given in Table 4.7. If the partition is further divided into sub-partitions, each sub-partition has its own horizontal and vertical motion vectors.

An important step in this process is the recording of the types of the mb_type of the block and also the sub_mb_type of each of the sub macroblocks. When the mb_type syntax element is encountered, a variable stores the type of partition of the macroblock. Similarly, when the

sub_mb_type syntax element is encountered, a variable called partIdxPartition is updated.

MB partition	Partition type index	sub-MB partition	sub-Partition type index
16x16	0	8x8	0
16x8	1	8x4	1
8x16	2	4x8	2
8x8	3	4x4	3

Table 4.8: Indices representing partition type

In the neighbor information processed for a sub_mb_type SE, the part index (partIdx) is also received. The partIdx ranges from 0 to 3 and indicates which 8x8 block is being processed. Two bits are needed to store the type of partition or sub-partition. The values for each partition and sub-partition is given in Table 4.8. For the sub_mb_type SE, there could be a maximum of four partition types to be stored. This means that we need a total of 4*2 bits to store the information in partIdxPartition.

When a sub_mb_type syntax element is encountered, the partIdx is recorded and the partition type index is determined. To update the value into partIdxPartition, the two bit partition index is shifted to the left by (partIdx * 2) times and this shifted value is logically OR'd with the original partIdxPartition. This effectively updates the value for further processing later.

The idx_x and idx_y values for each motion vector is got from the JM software [14] and helps to determine the neighbor information. The indices for an 8x8 partitioned macroblock and the indices for a 4x4 partitioned macroblock are shown below.

$$block8x8Idx = idx_x + idx_y * 2 \tag{4.7}$$

Equation 4.7 gives calculates the block8x8Idx value with the help of indices idx_x and idx_y. These values are obtained from the JM software input. Table 4.9 shows the numbering of 8x8 block.

$$block4x4Idx = idx_x + idx_y * 4 \tag{4.8}$$

This type of numbering will occur when the macroblock partition is of type 3 as shown in Table 4.8 and the sub-macroblock type is of type 0. When the sub-macroblock type is 3, we use Equation 4.8 to calculate the block4x4Idx variable. Table 4.10 shows the numbering of indices in such a case.

idx_y ↓ idx_x →	0	1
0	0	1
1	2	3

Table 4.9: Indices for when partitions are 8x8

If the sub-macroblock type is not 4x4 partitions, then the indexing changes from the table. These indices are crucial to calculating the top and left neighbor especially if the neighbors are within the current macroblock. The method to compute these neighbors is elaborated here.

idx_y ↓ idx_x →	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Table 4.10: Indices for when partitions are 4x4

Only blocks at left or top edges of a macroblock refer to neighboring macroblocks for ctxIdxInc calculation, and the blocks at left and top edge are indexed by idx_x=0 and idx_y=0, respectively. Algorithm 4 details the computation of certain flags to determine the neighbor.

The value ctxIdxInc depends on the sum of two values; absMvdCompA and absMvdCompB which is from the neighbors A and B, respectively. The Equations 4.7 and 4.8 show the way that ctxIdxInc is calculated.

$$sum_abs = absMvdCompA + absMvdCompB \tag{4.9}$$

The value of absMvdCompN is equal to a shifted value of Abs(mvd_lX [mbPartIdx] [subMbPartIdx] [compIdx]). The shift direction and value depend on variables compIdx and MBaffFrameFlag and whether there the current macroblock is coded in a different way than the neighbor at N, that is either frame coded or field coded.

$$ctxIdxInc = ((sum_abs > 32)?2 : (sum_abs < 3)?0 : 1) \tag{4.10}$$

Algorithm 4 Top and left neighbor indices computation

```
if idx_x != 0 then
    left_idx_x = idx_x - 1;
    left_is_currmb = 1;
else
    left_idx_x = idx_x;
    left_is_currmb = 0;
end
if idx_y != 0 then
    top_idx_y = idx_y - 1;
    top_is_currmb = 1;
else
    top_idx_y = idx_y;
    top_is_currmb = 0;
end
```

The value of compIdx is 0 for mvd_l0 and 1 for mvd_l1. The MBaffFrameFlag is a value that is got from the input slice information. If compIdx is equal to 1 and MBaffFrameFlag is equal to 1, and the current macroblock is a frame macroblock and the neighbor is a field macroblock, then the shift is 1 bit to the left. If compIdx is equal to 1 and MBaffFrameFlag is equal to 1, and the current macroblock is a field macroblock and the neighbor is a frame macroblock, then the shift is 1 bit to the right. If neither of the above conditions is satisfied, then the value is not shifted left or right.

Computation of mvd_lX All the motion vector information, which is 2 lists of horizontal (16*11 bits) and vertical (16*9 bits) motion vectors for each neighbor and the current macroblock horizontal and vertical motion vectors, would require 160 registers. To reduce the amount of information stored, we make an assumption that we require only to know whether the motion vector value is greater than, equal to or less than 64 (refer to Equation 4.8) for the upper limit of bits to store. This means that we can store values of motion vector up to 64 as they are and values greater than 64 are simply stored as 65, as this is all the information we need to decide that ctxIdxInc is 2.

This reduces the significant information we need to 7 bits. Since the KiloCore II has registers of size 16, we can accommodate two motion vectors in a single register. This considerably

reduces the number of registers to 48.

ref_idx_lX idx_x, idx_y, list, refidxB, refidxA, mbTypeB, mbTypeA, preddir, mbFieldB, mbFieldA, mbAvailB, mbAvailA, curr_mbtype, curr_submbtype, slice_type, part_idx, curr_ mbfield, curr_refidxlx, MBaffFrameFlag

The ctxIdxOffset for this is 54. ref_idx_lX corresponds to each 8*8 macroblock partition (sub-macroblock). Therefore, there are four ref_idx_lX in a macroblock. The procedure for finding the top and left neighbor are the same as detailed in Algorithm 4.

The two types of ref idx are stored: forward-prediction ref_idx and backward-prediction ref_idx. The list variable determines whether it is the ref_idx_l0 or backward prediction or the ref_idx_l1 or forward prediction.

The five bit ref_idx is reduced to only 2 bit storage since it is only needed to be known whether the value is equal to 0 or 1 or greater than 1. Apart from neighbor's ref_idx, whether or not the neighbor is coded in direct mode is another reference in this process.

mb_qp_delta prev_mbqpdelta, prev_mbskipflag

The ctxIdxOffset is 60. When binIdx is equal to zero, it needs the previous macroblock's mb_skip_flag and mb_qp_delta. If the previous macroblock was skipped or if the previous mb_qp_delta was zero, then the ctxIdxInc is also equal to zero. Otherwise, the ctxIdxInc is set to 1.

intra_chroma_pred_mode ICPMB, ICPMA, mbAvailB, mbAvailA

The ctxIdxOffset is 64. When the binIdx is equal to zero, Equation 4.6 is used to compute the ctxIdxInc. condTermFlag is 1 unless the neighbor's intra_chroma_pred_mode is equal to zero. If the neighbor is not available, then the condTermFlag is set to zero.

prev_intra4x4_pred_mode_flag None

ctxIdxOffset is 68 and ctxIdxInc is 0.

rem_intra4x4_pred_mode None

ctxIdxOffset is 69 and ctxIdxInc is 0.

mb_field_coding_flag mbAvailB, mbAvailA, mbFieldB, mbFieldA

The ctxIdxOffset here is 70. Equation 4.6 is used to find ctxIdxInc. If the neighbor macroblock is unavailable or if its mb_field_decoding_flag is 0, then the condTermFlag is set to 0 otherwise 1.

coded_block_pattern CBPb, CBPa, mbTypeB, mbTypeA, mbAvailB, mbAvailA, currMBCBP

The ctxIdxOffset is 73 for the prefix and is 77 for the suffix. In the coded_block_pattern (CBP) bins, there are usually four bits for the prefix and either one or two bits for the suffix. The prefix bits signify the Luma component or CBPLuma and the suffix bits signify the Chroma component or CBPChroma. In a single macroblock, there can be a maximum of four 8x8 luma blocks, one 16x16 Cb chroma block and one 16x16 Cr chroma block. The number of blocks is determined by the partitions within a macroblock.

The CBP syntax element only appears once in a macroblock. However, due to the four 8x8 luma blocks, the neighbor of a luma block being coded could be within the same macroblock, leading to more complicated method of computing the top and left neighbors. In the prefix, the position of the bin or the binIdx signifies which 8x8 luma block is being coded. The Figure 4.12 shows the neighbours of the luma blocks and their indices as represented in this work.

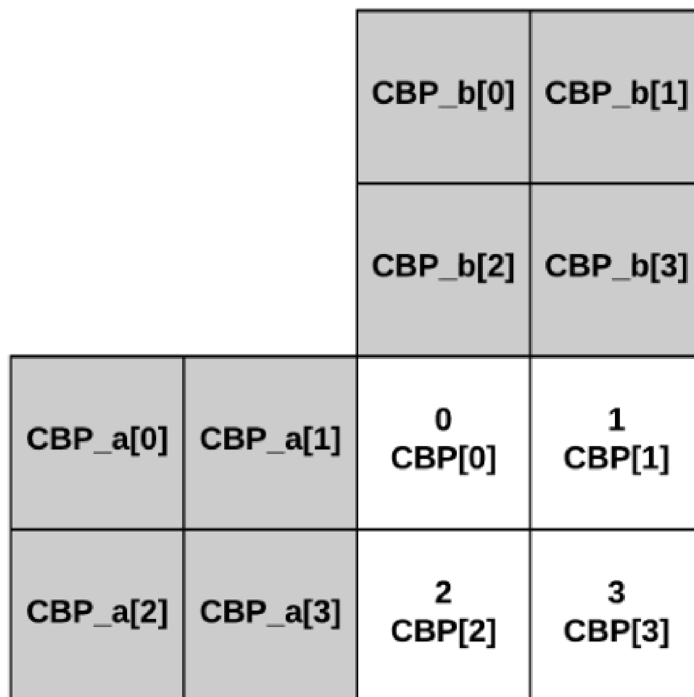

Figure 4.12: Neighbor of CBPLuma block [6]

Thus, the luma component top and left neighbors can be within the same macroblock or within the neighbor macroblock B or A. The chroma component has only one 16x16 block per macroblock, leading to its top and left neighbors being the actual macroblock neighbors B and A, respectively. These neighbors are used to calculate the ctxIdxInc for each bin.

coded_block_flag idx_x, idx_y, iCbCr, ctxBlockCat, CBFB, CBFA, mbTypeB, mbTypeA, mbAvailB, mbAvailA, currMBtype

The ctxIdxOffset is 85. The numbers of Coded Block Flags (CBF) within one macroblock is determined by the category of the transform coefficient block, which is specified by the parameter ctxBlockCat. The categories are shown in Table 4.11.

Block category	Maximum number of Coefficients	ctxBlockCat
Intra16x16_lumaDC	16	0
Intra16x16_lumaAC	15	1
Intra4x4_luma	16	2
chroma_DC	4	3
chroma_AC	15	4

Table 4.11: Categories of transform block

significant_coeff_flag currMBfield, scanningPos, ctxBlockCat, frame_coded_flag

The ctxIdxOffset is 277. For the residual syntax elements like the significant_coeff_flag and the last_significant_coeff_flag, the scanning position is used to compute the ctxIdxInc. This value is initialized to zero when the first coded_block_flag is encountered and incremented on every subsequent one.

last_significant_coeff_flag currMBfield, scanningPos, ctxBlockCat, frame_coded_flag

The ctxIdxOffset is 338. Refer to significant_coeff_flag section to see how ctxIdxInc for last_significant_coeff_flag is found.

abs_coeff_level_minus1 numLevelEq1, numLevelGt1, slice_type, ctxBlockCat

The ctxIdxOffset is 227. The ctxIdxInc is computed based on the number of coefficients encountered up to this point and whether they are greater than or equal to zero.

coeff_sign_flag None

ctxIdx is set to 0.

end_of_slice_flag None

ctxIdx is set to 276.

4.3.2 Calculation of Context index

The formula to calculate context index for all the syntax elements except the residual elements is given below 4.9:

$$ctxIdx = ctxIdxOffset + ctxIdxInc \quad (4.11)$$

The residual elements have their context index calculated using the following formula 4.10. The Table 4.12 shows the calculation of ctxBlockCatOffset for the residual syntax elements.

$$ctxIdx = ctxIdxOffset + ctxIdxInc + ctxBlockCatOffset \quad (4.12)$$

SE_type ↓ ctxBlockCat →	0	1	2	3	4
coded_block_flag	0	4	8	12	16
significant_coeff_flag	0	15	29	44	47
last_significant_coeff_flag	0	15	29	44	47
abs_coeff_level_minus1	0	10	20	30	39

Table 4.12: ctxIdxBlockCatOffset based on the syntax element and ctxBlockCat

4.4 Binary Arithmetic Encoder

The Binary Arithmetic encoder (BAE) needs the context model to be looked up first and then this model is passed to the renormalization stage. The context model is looked up by the

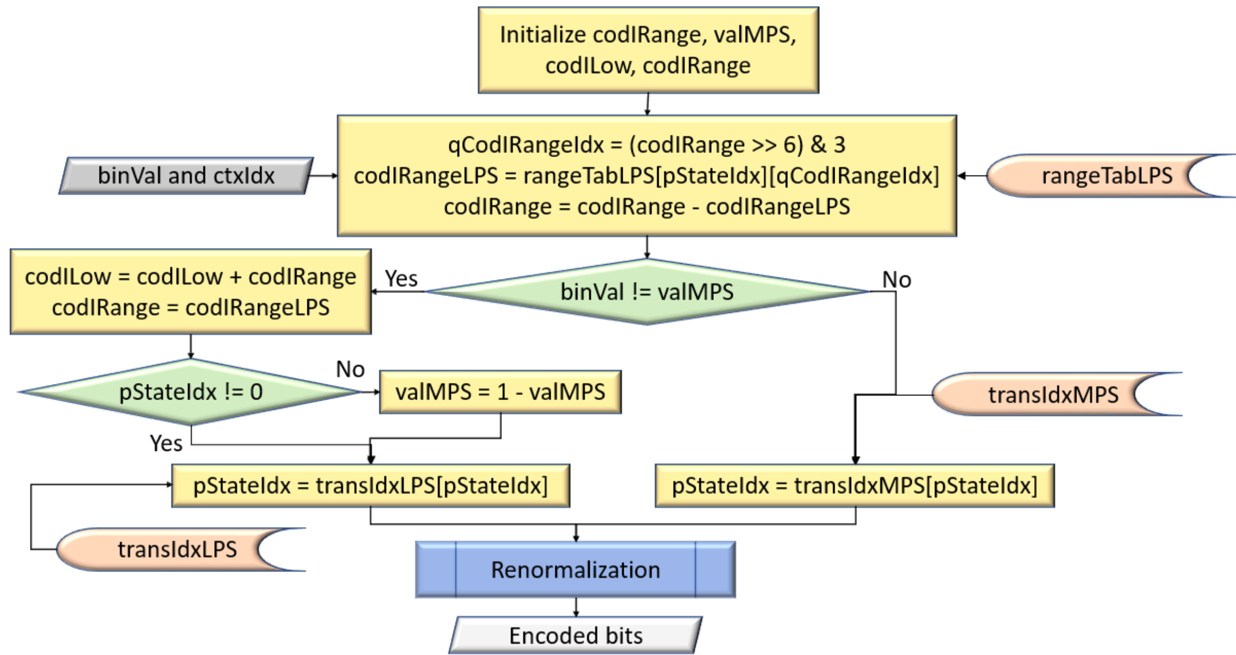

Figure 4.13: Flow chart of the Binary Arithmetic encoder

context index (ctxIdx) from the previous stage. The last merging core from the previous section also iterates over each bin from the collection of bins to send only a single bin and its associated ctxIdx. The Figure 4.13 shows the flow of the data of the Binary arithmetic encoder.

The Arithmetic encoder block consists of the following cores:

- R_X *rangeTabLPS_X*: uses one input port and one output port.
- A_0 *bae_stage0_0*: uses one input port and one output port.
- A_1 *bae_stage1_0*: uses both input ports and two output ports.
- A_2 *bae_stage1_1*: uses both input ports and one output port.
- A_3 *renormalization*: uses one input port and two output ports.
- A_4 *Accumulate_bins*: uses one input port and one output port.

The Figure 4.14 shows the overall connections of the cores that comprise the binary arithmetic encoder. The last core of the merging stage goes through the bins starting at the most significant bin (binIdx = 0) and sends it to the pState cores. The associated ctxIdx for each binIdx is also sent along with each bin. For any binIdx greater than 6, the ctxIdx is the same as when

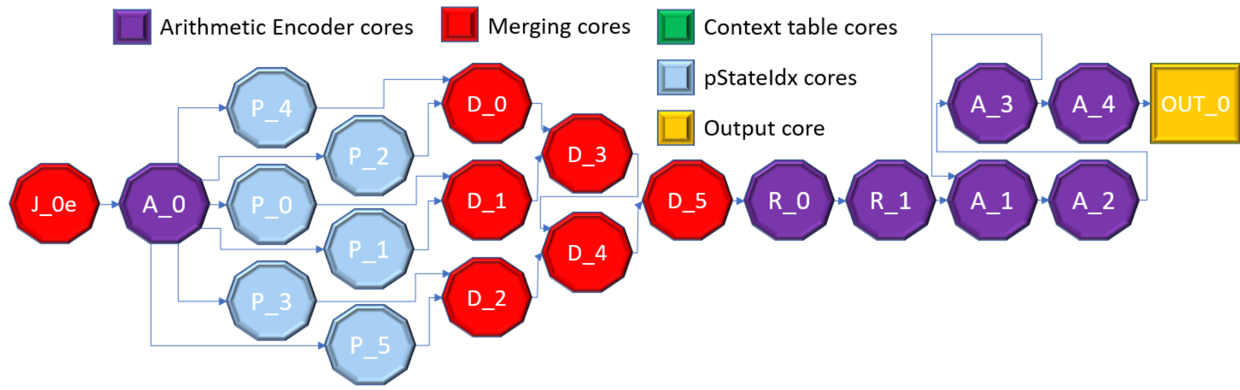

Figure 4.14: Connections for the Binary arithmetic encoder block. J_0e is the last *bin_join* core. A_0: *bae_stage0_0*, P_X: pStateIdx cores, D_X: Demux cores, R_X: rangeTableLPS cores, A_1: *bae_stage1_0*, A_2: *bae_stage1_1*, A_3: *renorm*, A_4: *Accumulate_bins*

binIdx is equal to six. This core requires the length field and the length_suffix field in order to cycle through all the bins.

There are also cores for the context tables and probability state tables. The connections between the context table cores and the probability state cores is shown in Figure 4.15. The cores performing this function is listed below:

- M_0 *ctx_mux*: uses one input port and two output ports.
- D_0 to D_4 *ctx_demux_X*: uses both input ports and one output port.
- T0_0 to T0_5 *context_table0_X*: uses two input ports and two output ports.
- T1_0 to T1_5 *context_table1_X*: uses one input port and one output port.
- P_0 to P_5 *pState_table_X*: uses one input port and one output port.
- D_5 *ctx_demux_5*: uses one input port and one output port.

In the Probability state cores, based on the value of cabac_init_idc and slice_type, one of a set of four tables is initialized into the cores' Dmem memories. These values are the pStateIdx, which can take a maximum value or 63 and a minimum of 0. Thus, we need 6 bits to store the data. The associated most probable symbol (valMPS), which is one bit long, needs to be stored as well. Since, the KiloCore has a word size of 16 bits, the pStateIdx and valMPS for a particular value of ctxIdx are logically OR'd together. This result is stored in the lower 7 bits of a single word.

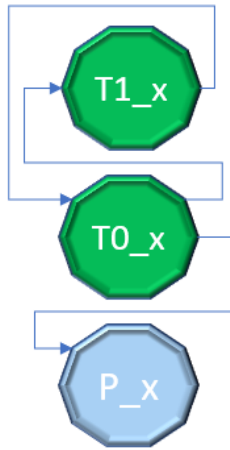

Figure 4.15: Connection between the Context table cores and the pStateIdx cores

The ctxIdx for field coding ranges from 3 to 398 excluding the range of 105 to 226. The values of 105 to 226 is used for frame coding. That leaves us with a need to store 274 probability states. Along with that, we also require to store the transIdxLPS table in each pState core, as this shows the transition to the next state for pState, when the bin value is not equal to the valMPS.

Algorithm 5 pState and valMPS look-up and update

Initialize tables based on slice_type and cabac_init_idc;

Receive bin and ctxIdx from previous block;

pStateIdx = table[ctxIdx] & 63;

valMPs = (table[ctxIdx] shifted right by 6) & 1;

if *bin != valMPS* **then**

  if *pStateIdx is equal to 0* **then**

    valMPS = 1 - valMPS

  end

  pStateIdx = transIdxLPS[pStateIdx]

else

  if *pStateIdx is less than 63* **then**

    Increment pStateIdx

  end

end

There is no need to store the transIdxMPS table (when bin is equal to valMPS) in each

pState core, since the computation of the transition to the next state is trivial in this case. It is only a matter of adding 1 to the current state unless the current state is 63, in which case, it remains the same.

The algorithm for accessing the context model and the updating of the probability state is shown in Algorithm 5. This whole algorithm is done in each of the pStateIdx cores.

The syntax elements are dealt with in different pstateIdx cores based on their ctxIdx. The distribution of the ctxIdx with the core that handles it is shown in the Table 4.13.

ctxIdx range	Core name	ctxIdx range	Core name
[3, 53]	*pState_table_0*	[277, 318]	*pState_table_3*
[54, 104]	*pState_table_1*	[319, 359]	*pState_table_4*
[227, 275]	*pState_table_2*	[360, 398]	*pState_table_5*

Table 4.13: Range of ctxIdx handled by each pState core

The size of the transIdxLPS table is 64 words. This gives us less than 64 words in each pState core to store the pStateIdx and valMPS hybrid word. Giving allowance for intermediate variables in each core, we will need 6 cores to store one set of the pState tables. The algorithm for look up and update of the pState and valMPS is given in Algorithm 5.

Algorithm 6 Regular encoder

Set codIRange equal to 510;

Set codILow equal to 0;

qCodIRangeIdx = (codIRange shifted right by 6) & 3;

codIRangeLPS = rangeTabLPS [pStateIdx] [qCodIRangeIdx];

codIRange -= codIRangeLPS;

if *binVal is not equal to valMPS* **then**
| codILow += codIRange;

end

Renormalization();

Increment symCnt;

The values of ctxIdx that are not included are explained here. The ctxIdx 276 and 0 are associated with the terminate encoder and the bypass encoder respectively. These do not change

the value of the context model. The values of ctxIdx from 105 to 226 are used for frame encoding.

When each new slice is encountered, or at the end of a slice, the tables are reinitialized from the context table cores as shown in Figure 4.16.

The encoding engines are all written in the *bae_stage1_0* and *renorm* cores. The three encoding engines are the regular encoding engine, bypass encoding engine and terminate encoding engine. They are explained in the algorithms below. The regular arithmetic encoding algorithm is given in Algorithm 6.

The calculation of codIRangeLPS is done in a two-fold step. In the rangeTabLPS cores, the 256 valued table is stored. Here all based on the pStateIdx, four values are sent to the *bae_stage1_0* core. These four values represent all the possible values that can be taken by the codIRangeLPS. Out of these four values, the one chosen depends on qCodIRangeLPS. This value depends on codIRange. Since codIRange is initialised only in the *bae_stage1_0* core, all four possible values are sent from the previous core. This also eliminates the need to increase the number of cores in the feedback loop.

The bypass encoder is shown in Algorithm 7. There is no update of pStateIdx or valMPS in this case.

Algorithm 7 Bypass encoder

codILow shifted to the left by one bit is assigned to codILow;

if *bin is not equal to zero* **then**
  Assign the sum of codILow and codIRange to codILow;
end

if *codILow is greater than or equal to 1024* **then**
  PutBit(1);
  codILow = codILow - 1024;
else
  if *codILow is less than 512* **then**
    PutBit(0);
  else
    codILow = codILow - 512;
    Increment bitsOutstanding by 1;
  end
end

Increment symCnt by 1;

The algorithm for encoding the bins just before termination is given in Algorithm 8. Here also there is no update on pStateIdx or valMPS variable.

Algorithm 8 Terminate encoder

```
Subtract 2 from codIRange;
if bin is not equal to 0 then
    codILow = codILow + codIRange;
    EncodeFlush();
else
    Renormalization();
end
Increment symCnt by 1;
```

Renormalization is employed by the Binary Arithmetic encoder to ensure that the codIRange and codILow values do not exceed their register sizes of 9 and 10 respectively. It also dictates the generation of the bitstream that is bit packed in the next stage. The algorithm for renormalization is given in Algorithm 9.

Algorithm 9 Renormalization

```
while codIRange less than 256 do
    if codILow less than 256 then
        PutBit(0);
    else
        if codILow greater than or equal to 510 then
            Subtract 510 from codILow;
            PutBit(1);
        else
            Subtract 256 from codILow;
            Increment bitsOutstanding;
        end
    end
    Left shift codIRange by 1;
    Left shift codILow by 1;
end
```

The terminate encoder calls a special function called the *EncodeFlush*. This function ensures that the system knows when the slice has ended and bit-packs the rest of the 8 bit output word with information signaling to the decoder to either prepare for a new slice or to fully terminate the decoding. The algorithm for EncodeFlush is given in Algorithm 10.

Algorithm 10 EncodeFlush

```
codIRange = 2;
Renormalization();
PutBit(Ninth most significant bit of codILow);
WriteBits(Eigth most significant bit of codILow, 1);
WriteBits(1, 1);
```

The method *PutBit* is used to send bits to the bit-packer. It needs the bitsOutstanding and firstBitFlag variables for its execution. The algorithm for PutBit is given in Algorithm 11.

Algorithm 11 PutBit

```
Input is B;
if firstBitFlag is not equal to 0 then
|  firstBitFlag = 0;
else
|  WriteBits(B, 1);
end
while bitsOutstanding is greater than 0 do
|  WriteBits(1-B,1);
|  Decrement bitsOutstanding by 1
end
```

Post the renormalization stage is the bit-packer stage which packs 8 bits into a word to write to the output file. The only special consideration is when the termination engine signals the core to flush the rest of the words. The process for that is explained in Algorithm 10.

Chapter 5

JM software and functional verification

5.1 JM software

The implementation of CABAC on KiloCore II platform, gets its input from the JM reference software [14]. JM software is the H.264/AVC reference software developed by joint team of ISO/IEC MPEG and ITU-T VCEG who formulate the standard for H.264 video coding. Compared to other software implementation of H.264 encoder like X264, The JM software has more clear and complete coding procedure implemented as described in the standard. Therefore, the intermediate data, used as input and test vector of CABAC encoder, is more easily extracted from JM software. In the functional verification of the proposed CABAC encoder implementation, the JM version 8.6 is used for verifying the functional features required in main profile, level 5.1 of H.264 standard.

The input to the JM software has to be .yuv files. Ultra Video Group 4k video test sequences [15] and Xiph video test media [16] are two sources of CIF, QCIF, HD and 4k video sequences.

The JM software processes these video files and intermediate values are written to a file to treat as input for the KiloCore's input ports.

The outputs of the JM software's CABAC block are also written to a file to compare and validate the results. Four output files are generated. The first one named *file_binarizer.txt* has the bins for each syntax elements, broken into 16 bit words, so that the Kilcore's input handler can

File name	Information extracted
file_input.txt	Syntax element information, neighbor information
file_sliceinfo.txt	Slice information
file_binarizer.txt	16-bit word outputs of binarizer block
file_context.txt	Context modeler output like pState, valMPS
file_rnl.txt	Renormalization block output like codIRange, codILow
file_byteoutput.txt	8-bit word outputs of Arithmetic encoder

Table 5.1: JM generated file information

process the value. Since the binarizer of the implementation also outputs values that are split into 16 bit words for further processing, it is a simple compare of the words to check if they are equal. So the first output file effectively verifies the binairzer block.

The second output file, *file_context.txt*, has the values of the probability state index and the most probable symbol values. It also has the coding method, whether regular, bypass or terminate. This is useful to verify the output of the context modeler block. Although the actual output of the context modeler block is the context index, the JM software does not make use of the same storage method for the context, so it has no number to specify the context index. The value that is looked up using the ctxIdx is the pStateIdx and valMPS, which is what we use to verify the block.

The third file called *file_rnl.txt*, checks the renormalization registers. The codIRange and codILow registers are checked with every symbol encoded. The values are output even if there are no bits output in that round.

The last output files *file_byteoutput* has the bit output of the CABAC encoder packed into 8 bits a word.

5.2 Binarization verification

There are 18 syntax elements, 8 of which are flags that can take only one of two values. The ranges of the rest of the elements are listed in the table below.

These are the ranges for which the binarizer needs to be checked. The logical correctness of the binarizer is checked against the JM software output *file_binarizer.txt*. An application was

Syntax element	Slice	Range	Syntax element	Slice	Range
mb_type	I	[0, 25]	ref_idx_lX	all	[0, 31]
mb_type	P	[0, 3], [5, 30]	mb_qp_delta	all	[-26, 25]
mb_type	B	[0, 48]	intra_chroma_pred_mode	all	[0, 4]
sub_mb_type	P	[0, 3]	rem_intra4x4_pred_mode	all	[0, 7]
sub_mb_type	B	[0, 12]	coded_block_pattern	all	[0, 47]
mvd_l0	all	[-2048, 2047]	abs_coeff_level_minus1	all	[-32768, 32768)
mvd_l1	all	[-512, 511]			

Table 5.2: Ranges of syntax elements

written in the Project manager for this purpose. The input source files for this application are the *file_binarizer.txt* and the output of the binarizer that is got from the KiloCore II implementation. The inputs are compared to check correctness and output in all the locations that the values were found to not match. The array implementation was found to have accurate results for all the syntax elements range of values, upon running this test application.

5.2.1 Debugging the binarizer

The binarizer debugging was simple as most of the values generated by the binarizer are fixed values or tabular values. The mechanism that was used to check the binarizer was comparing files; one generated by the JM software and one generated by this implementation. The files had the syntax element type number and the binarized value. When the application finds that two of the binarized values do not match, it prints out the previous two values. It also prints out a count variable that has been incrementing since the start of the program. This allows the exact location of the error to be found.

The only syntax elements that needed to be debugged were the coded block pattern, the motion vectors and the coefficient absolute value. With the coded block pattern, there was an error where the suffix and prefix positions were switched. This was easily fixed with shifting the bits to the correct position. With the motion vectors and the coefficient absolute values, they are syntax elements that are binarized using the UEGk method. The bug that was discerned in this case was that there was a mistake in the arrangement of the prefix and suffix. When combined they were

larger than the 16 bit word that is available to use in the KiloCore II. The shift value of the suffix needed to be adjusted to ensure the word that is first sent to output is filled first and then the second and then consequently the third.

5.3 Context Modeler verification

The functional correctness of the Context modeler can be verified by checking the output against the golden reference. The output of the context modeler block is the context index (ctxIdx). The JM software not store the value of ctxIdx directly, instead it uses a struct data structure to store the relevant information for a macroblock. The workaround is to verify the values that the ctxIdx is used to look up. These values are the probability state index (pStateIdx) and value of most probable symbol (valMPS).

The *file_context* file produces the output we need to verify the functionality of the context modeler block. The contents of the file are the pStateIdx, valMPS, binVal and mode for each bin that is encoded.

Details about the debugging process for the Context modeller block are elaborated in subsection 5.4.1.

5.4 Binary Arithmetic encoder verification

The verification of the Binary arithmetic encoder is two-fold. The first aspect that needs to be verified is that the renormalized limits of the coding are correct. The variables that are used to keep track of the limits are the codIRange and codILow variables. These are output from the JM software in the *file_rnl.txt* file. These variables are compared with the proposed implementation's values using a Project manager application.

The second aspect of the Arithmetic encoder to be verified is the correctness of the bit output of the full CABAC algorithm. The JM software is made to generate the *file_byteoutput.txt* that contains bit-packed byte-sized words of the CABAC output. These values are used to verify the correctness of the KiloCore II's implementation of CABAC.

5.4.1 Debugging the Context modeler and Arithmetic Encoder

It was found to be easier to debug the two blocks at once. The reason for this is that a test_output file is generated to keep track of certain critical variables for both the blocks. These critical variables gives us an indication of which block is generating the error, based on which values are not in compliance with the reference.

In the debug application that was written specifically to check the correctness of these blocks, two files are generated as output. The first one as already mentioned above is the test_output file. The second is the output of the renormalization core. This file has the variables codIRange and codILow output to it after the renormalization has occurred. In the application written, these values are compared with the reference file *file_rnl.txt*. A check file is generated from the application which tells us the line that does not match the reference as well as the previous two matching values. This allows us to find the very first point at which the error is occurring.

Once the line in *file_rnl.txt* has been found, the same line number in *file_context* gives us the probability state data from the reference for checking the context modeler. The test_output file has the following variables shown in table 5.3.

Context Variables	Renormalization variables
bypass_flag	codIRange before renormalization
terminate_flag	codILow before renormalization
ctxIdx	firstBitFlag
bin	bitsOutstanding
pStateIdx	codIRange after renormalization
valMPS	codILow after renormalization
	symCnt

Table 5.3: File test_output variables

Some of the bugs that were found in these sections were related to bit shifting by the wrong amount. There were bugs related to incorrect access of neighbor information that was stored, especially in the case of coded_block_pattern, mvd_lX, ref_idx_lX and abs_level_coefficients_minus1.

There was a lot of debugging effort that had to go into figuring out the working of the

bypass and terminate engine. The main reason is that the data required by these engines was lesser than the regular encoding, which meant a mismatch in the pipeline. Thus, the workaround for this bug was to have case specific inputs in certain cores as opposed to a more generic input handling.

There was also a issue encountered of buffer filling up in the context table lookup. These cores received input with every syntax element processed, but had to wait for the full encoding process to send data to the probability state cores. This lead to the buffer overflow and stopping of the process. The workaround was to fan out the tables, so they did not have to wait for the previous to fill up before beginning the receiving of data.

Chapter 6

Experimental Results and Analysis

The work described in this thesis is implemented on the KiloCore Project manager and Simulator. The KiloCore II or AsAP 4 architecture consists of a thousand independently programmable processors, connected by circuit linked network and each having its independent on memory of 128x16b words. Each core can accommodate a task. The tasks are written as C++ files that are compiled using the Compiler program. The area of each processor is 0.055 mm^2 [17]. The area usage for the implementation can be computed with this information.

This chapter discusses the results got from the simulations. It also talks about the mapping of the application onto the KiloCore II architecture. The throughput and energy data is discussed in relation to alterable parameters. The last section of the chapter compares the current implementation with other hardware and software implementations of CABAC.

6.1 Analysis of core usage

This CABAC implementation has blocks of cores for specific tasks of the algorithm. The application utilizes a total of 64 cores. Therefore the application occupies a total area of 3.52 mm^2 on the KiloCore II chip. Table 6.1 shows the area distribution of the cores and their tasks.

Figure 6.1 shows the distribution of the cores for the various tasks. In this section, the task distribution is studied. The task that occupies the most space on the KiloCore is the context table functionality. The functionality consists of the context initialize cores and the probability state table cores that are used for computing the probability state index and the most probable symbol for future bins with the same context index.

Overall these cores need to store data that is of size 989x9 bits. This data is spread across 12 cores' Dmem memories. The remaining 7 cores of this category are used to compute the new probability state and most probable symbol and to combine the results for the next stage of the CABAC which is the renormalization. Typically, in other implementations of CABAC, the context tables are stored in ROM and are only accessed during initialisation or when a new slice begins.

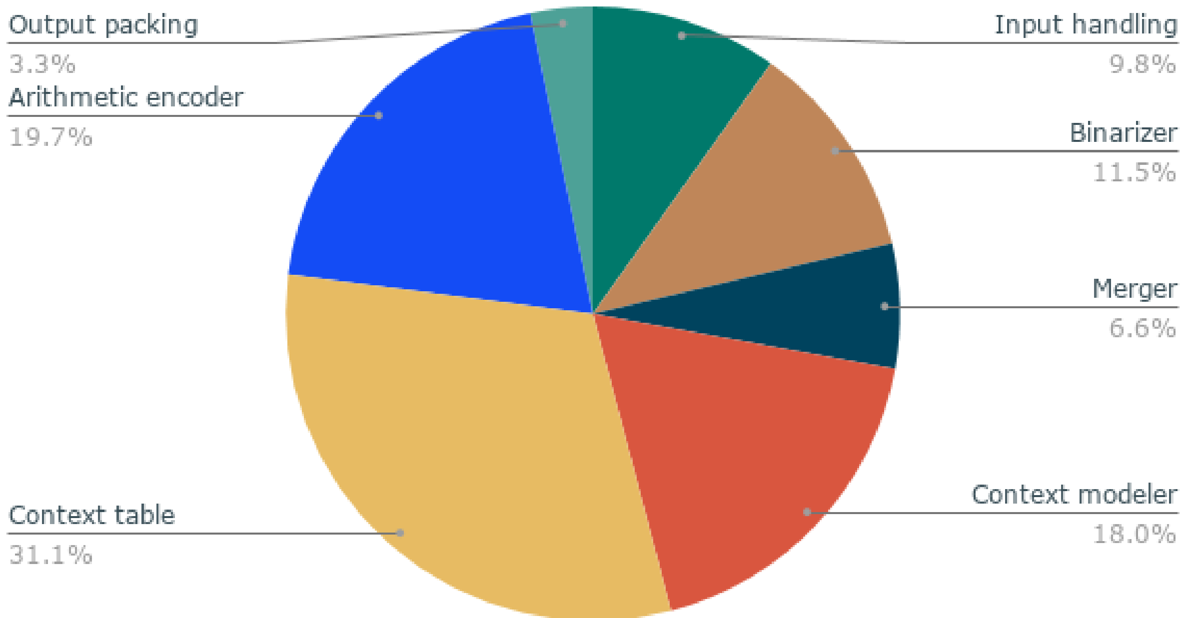

Figure 6.1: Chart showing distribution of tasks and the number of cores

The total area of the context table also comes into consideration when computing the area. In most implementations the ROM is treated as separate and not considered when calculating the area. This indicates that area in this implementation is still competitive with the smaller areas of the ASIC and hardware implementations.

The task with the least cores is bit packing or output packing cores. There are only two cores involved in this functionality. The core waits for eight bits to make a byte and sent to the output handling cores.

The next lowest number of cores are the input handling task cores. These cores send the syntax element information to the binarizer cores. These cores are also responsible for the parsing of the neighbor information that need to be used by the context modeler cores.

Task	Number of cores	Area usage in KC (mm^2)
Input handling	6	0.33
Binarizer	7	0.385
Context modeling	11	0.605
Context table	19	1.045
Merging	4	0.22
Arithmetic Encoder	12	0.66
Bit packing	2	0.11
Total	64	3.52

Table 6.1: Area usage per task

The context modeler cores take the bins from the binarizer cores and the neighbor information from the input handling cores where the neighbor information is parsed. These cores calculate the ctxIdx.

While the context tables cores are very similar in function, the arithmetic encoder cores all perform a specific task of the of the encoding engine. Each of the cores has a different task. The tasks include accessing the rangeTabLPS table, initialisation of the codIRange, codILow values, and renormalization, among others. The bypass and termination engines are also tasks that are performed by individual cores. Similarly, each binarizer core performs one of the binarization techniques.

The merging cores consist of the same type of core replicated a number of times to combine the output of the context modeler into the input of the context tables.

6.1.1 Mapping to the KiloCore II

The Mapper2 software [18] which uses Julia was used to map the application onto the KiloCore II architecture is discussed in this subsection. The Mapper2 tool is a general place-and-route framework. It allows for heterogeneous mapping to arbitrary-dimensional architectural models. The mapping is to the KiloCore II architecture for this application.

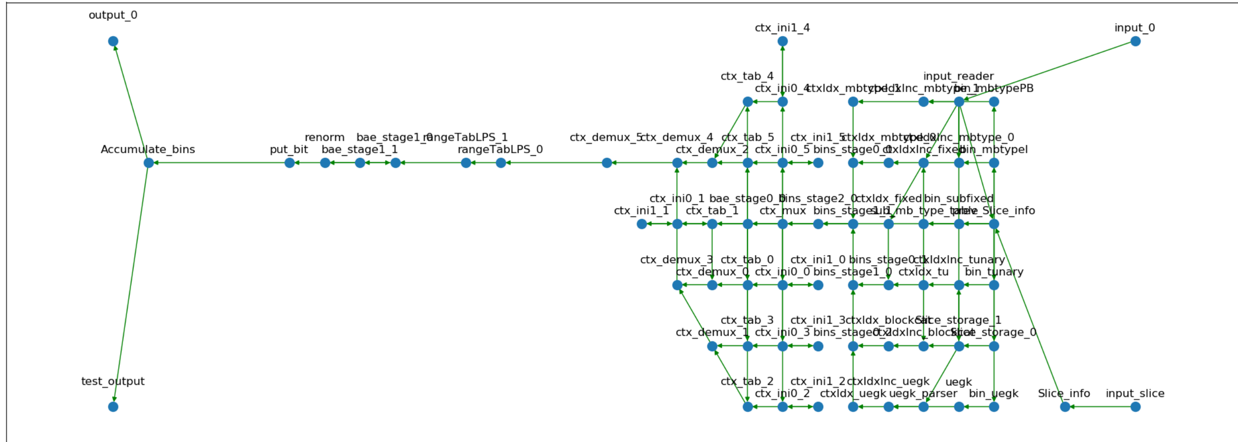

Figure 6.2: Standard mapping of cores

This application uses two mappings. The first mapping is the standard mapping on the Project Manager. This shows only the cores that are used and the connections between them. There are also labels for each of the cores. The Figure 6.2 shows the standard map.

The figure shows that there are two input handlers and two output handlers. The second output handler is mainly to test the application. The application allows for a more parallel execution in the first two stages of the CABAC application. However, when the actual encoding process begins, it is more serial. This is due to its dependencies on previous values. This can be seen on the right side of the image, where the cores are more fanned out. The merger cores fan in the data at the end of the context modeler stage in order to feed it to the BAE, which is the last stage. Finally, the encoding and bit packing are almost serial cores, with just one output and one input.

The second mapping is the application being placed and routed onto the floor plan of the KiloCore II architecture. The Figure 6.3 shows the mapping.

Even though this mapping does not have labels, it can be thought of as the standard map placed on the KiloCore II floor plan. There are clear parallels between the two images, with the input cores being on the right hand side and the output cores on the left hand side.

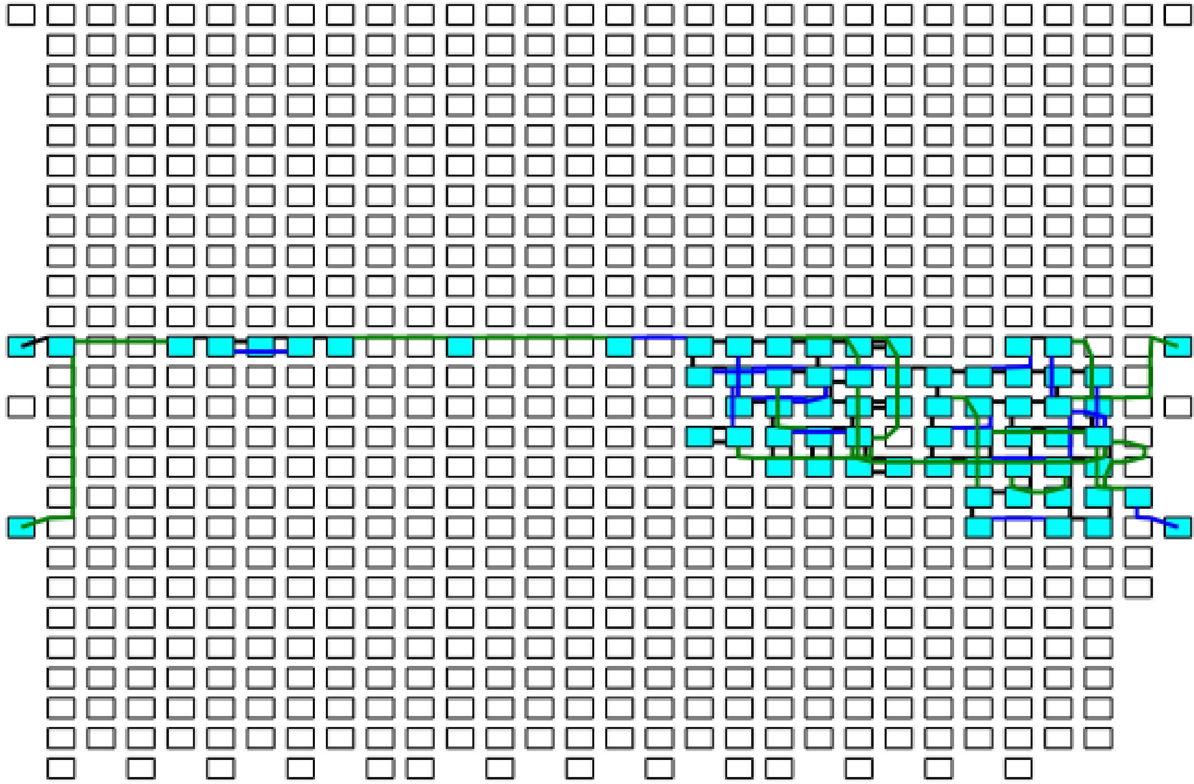

Figure 6.3: Placed and Routed map of cores

The blue connectors represent hops of one or two cores, and the green connectors represent hops of more than two cores.

The link length is calculated in terms of how many cores it is hopping over. The data and its distribution is shown below in the table and the chart.

Number of communication channels	97
Total global routing links used	177
Average Link Length	1.82
Maximum Link Distance	6

Table 6.2: Mapper details about links

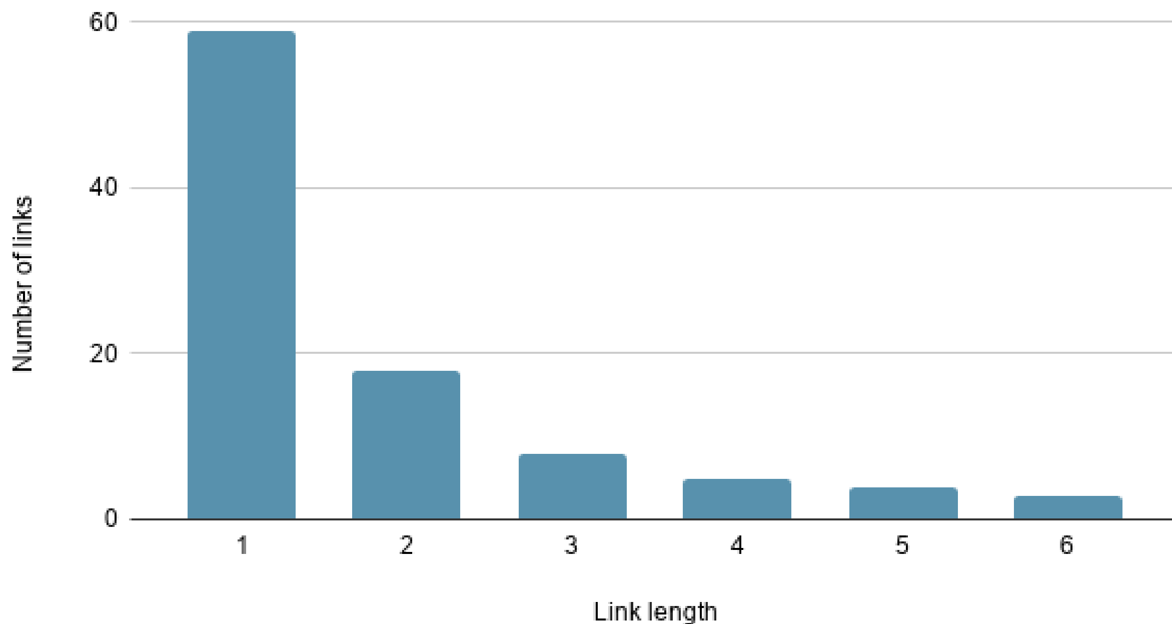

Figure 6.4: Chart showing link length data

6.2 Throughput and energy results

The metrics used to assess the performance of the CABAC implementation on the KiloCore array would be to measure the throughput and the energy. The total number of processors used is 64 making the total area come to 3.52 mm^2.

6.2.1 Throughput through the stages

The throughput is the first metric that is analyzed. The throughput is computed in MWords per second. A word can mean different things depending on which stage of the algorithm is being inspected.

At the binarizer stage, the bins of the syntax element constitute a word. The length of the bins varies from 1 to 53 bits depending on the syntax element. At the context modeler stage, the output consists of a single bin and its associated context index. The information about which encoding engined is also passed along. The context index itself ranges from 3 to 399. In the CABAC application, the final stage output word is composed of 8 bits. For comparing with

other implementations, we multiply the throughput measured by the Simulator, by 8. The reason for doing this is that most of the implementations compute throughput in kilo-bits-per-second or mega-bits-per-second.

Stage	Core name	Throughput (MWords per second)
Stage 1: Binarization	binarizer_mb_typeIP	12.63
	binarizer_mb_typeB	12.201
	binarizer_fixed	83.92
	binarizer_tunary	13.633
	binarizer_uegk	27.816
Stage 1: Binarization	*Average*	30.04
Stage 2: Context modeler	ctx_demux_5	82.989
Stage 3: Renormalization	renorm	25.463
Stage 4: Bit packing	Accumulate_bins	4.681

Table 6.3: Stage-wise throughput

Throughput is recorded at the end of each stage. The table 6.2 shows the throughput measurements of the various stages. At the binarizer stage, the outputs are still parallel. The average of the cores is found to compute the throughput at this stage. It can be observed that the different cores, process varying number of Syntax elements. The *binarizer_fixed* core processes the most number of syntax elements, since it handles the binarizing of 10 syntax elements.

The graph in Figure 6.5 shows the average throughput in each stage.

In the next stage, the context modeler, we have a much higher throughput. The throughput is more than double of the binarizer. In the binarizer stage, Syntax elements are dealt with. However, the context modeler deals with data related to individual bins of a syntax element. This means that the Context modeler cores encounter data either equal to or greater than the data encountered by the binarizer cores. The number of bins for a single syntax element varies from 1 to 53. In order to fully analyse the difference in throughput, we would also need to know the number of different types of syntax elements and their lengths of bins. This value will vary depending on each video.

The reduction of throughput for the next stage, namely the Arithmetic encoder stage could be because of two reasons. The *renorm* core at which the throughput is measured only outputs in

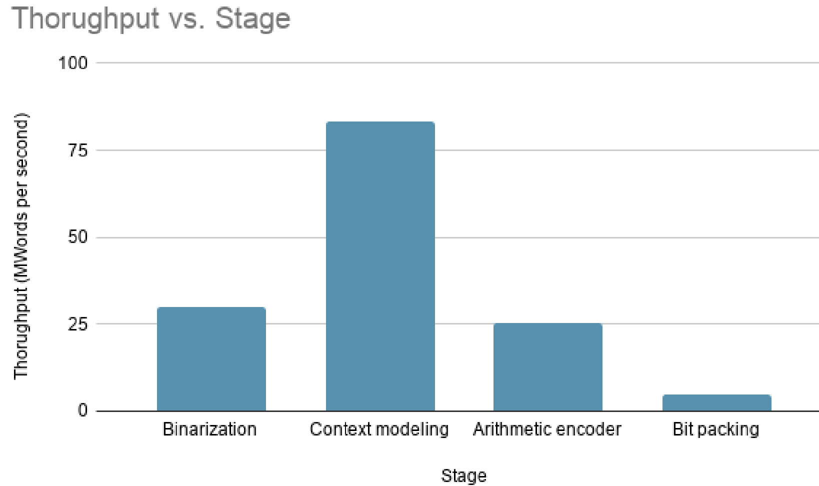

Figure 6.5: Throughput of the stages

certain situations. This is because, every bin encountered does not produce an output. The values of codIRange and codILow determine how many output bits are sent to the bit packing stage.

Another explanation for the reduction of the throughput could be that the core in question has a feed-back loop with a previous core. The new codILow and codIRange values are sent to the beginning of the Binary Arithmetic encoder for the next bin. This extra latency might reduce the throughput and calculation rate.

Finally, the bit packing stage has a further reduction in throughput as it waits for eight bits to arrive before producing a single output.

6.2.2 Scaling with Voltage results

An experiment was conducted by changing the base voltages of each core on the KiloCore II. The varying was altered from 0.8 V to 1.1 V [19], which is the maximum for a core. The effects of this varying was measured for final throughput, total energy and power.

Table 6.4 shows the overall effects of varying voltage.

The method to change the voltage of each core is to add a section of code in the main file that is executed by the Simulator. This code iterates over all the cores in the core vector and using the function core.Set_Voltage(x) to change the operating voltage. The value x that is passed as an argument is the new voltage in mVolt.

The plots for the three metrics, throughput, energy and power are also shown below. As

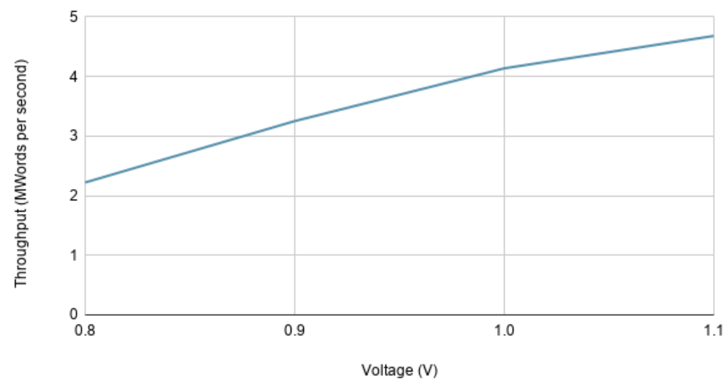

Figure 6.6: Throughput variation with Voltage

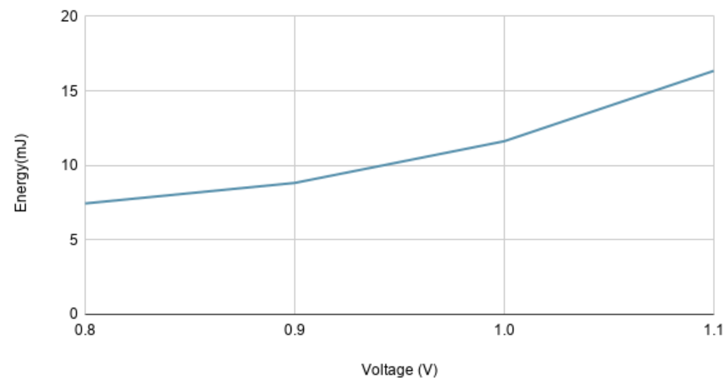

Figure 6.7: Energy variation with Voltage

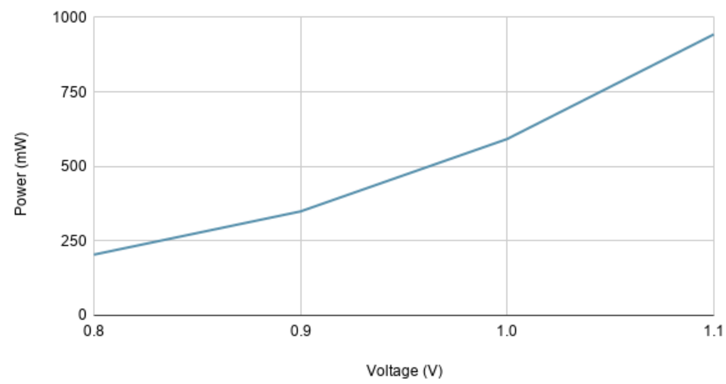

Figure 6.8: Power variation with Voltage

Voltage (V)	Throughput (MWords/s)	Energy (mJ)	Power (mW)
0.8	2.22	7.44	204
0.9	3.52	8.82	349
1.0	4.13	11.62	592
1.1	4.68	16.34	943

Table 6.4: Effects of voltage

can be seen the throughput improves with increase in voltage.

It is also evident that the power and energy also increase with voltage. This is not entirely desirable, but it is inevitable. Based on the requirements of the system, keeping in mind the trade-offs, a decision can be made about which operating voltage each core needs to be fixed at.

6.2.3 Energy results

The energy of the system is shown in Table 6.4. A metric to consider is the amount of energy expended to produce each bin. The number of 8-bit words that were generated was 27060. Thus, the number of bins is 27060*8 which is 216480 bins.

Using this value, we can compute the energy consumption for each bin shown in Table 6.5.

Voltage (V)	Energy per bin (μJ/bin)
0.8	34.37
0.9	40.74
1.0	53.67
1.1	75.48

Table 6.5: Energy per bin measurements

6.3 Comparison with other implementations of CABAC

CABAC encoder has been implemented on many different platforms. Software implementations include the JM software [14], X264. The JM software is implemented on the Intel Xeon

Processor E5-2680 v2. This processor has a 2.8 GHz clock and uses 22 nm technology. The processor consists of 10 cores and 20 threads.

The CABAC implementation on KiloCore II is also compared to various existing hardware implementations. The hardware implementations include ASIC designs and FPGA designs. The implementation of CABAC on the KiloCore II platform is a software implementation. Therefore, it is expected that the hardware implementations, which are meant to accelerate performance, will perform better than the KiloCore II's implementation. These are not a fair comparisons as the hardware implementations have optimizations that cannot be matched in a software platform. Despite these limitations, the throughput for the KiloCore II's implementation, is found to be within a factor of five, when compared to hardware implementations. This Chapter discusses the details of these hardware implementations.

For the purpose of comparison, the technology of the implementations was scaled to match the current technology which is 32 nm. The tool used to this is DeepScaleTool [20], [21], [22]. Table 6.7 shows the scaled results of other implementations. The tool is a spreadsheet which provides the scaling factors, with the inputs of current technology and target technology. It provides scaling factors for metrics such as Area, Delay, Energy, Energy delay product, Power and Throughput.

This section will give a brief description of the various types of implementations. In designs explained in [23], [24], [25], only the BAE is implemented in hardware. The binarizer and context modeler are implemented in software.

A fully hardwired implementation is found in [26]. Designs in [27] and [28] are also fully hardwired. [26], [27] and [28] all make use of neighbor memories for the context index computation. For 4K video encoding, neighbor memory might be as large as 34.6K bits.

Due to the high computational complexity and large size data storing in context modeling, the reported FPGA implementations mainly focus on the processes of BAE and binarization. The BAE implementations in [23], [34], [29] all have FPGA designs. [35] has an FPGA implementation for the Binarizer.

As can be observed from the table, most of the hardware implementations have better throughput. When compared with the software implementation, the JM software, we find that the throughput is much better for this implementation.

The latency introduced, in this implementation, for fanning out at the initial stages and

Design	Throughput	Technology	Max Clock Freq	Area	Power
Shojania HW [23]	87 Mbin/sec	0.18 μm	263 MHz	0.42 mm^2	48 mW
Li HW [24]	80 Mbin/sec	0.35 μm	150 MHz	-	-
Kuo HW [25]	200 Mbin/sec	0.18 μm	200 MHz	0.21 mm^2	43 mW
Osorio HW [29]	350-428 Mbin/sec	0.35 μm	186 MHz	-	-
Chen HW [28]	216 Mbin/sec	0.15 μm	333 MHz	-	-
Liu HW [26]	134 Mbin/sec	0.13 μm	200 MHz	-	-
Tian HW [27]	620 Mbin/sec	0.13 μm	620 MHz	-	-
Chen HW [30]	315 Mbin/sec	0.13 μm	222 MHz	-	-
Liu HW [31]	476 Mbin/sec	90 nm	238 MHz	-	23 mW
Tsai HW [32]	1191 Mbin/sec	0.13 μm	254 MHz	-	-
Zhou HW [33]	1409 Mbin/sec	65 nm	330 MHz	0.43 mm^2	-
Chen HW [6]	1886 Mbin/sec	28 nm	1.88 GHz	0.03 mm^2	11 mW
JM SW [14]	0.731 Mbin/sec	22 nm	2.8 GHz	-	-
This work	37 Mbin/sec	32 nm	1.78 GHz	3.52 mm^2	943 mW

Table 6.6: CABAC throughput measurements

the subsequent fanning in at later stages, might contribute to the low throughput. The availability of neighbor information at specific cores is another factor. The feedback loop in the renormalization is also a critical point where there is a slow-down of data.

Most of the papers report their area as gate count. They also possess RAM and memory modules which make up the chip area, that would not be considered in the gate count. Thus, it is not ideal to compare those implementations' area and its throughput per area measurements with this work. The papers which had the chip area mentioned is added to Table 6.6.

The ratio of the throughput of this implementation to the throughput of the scaled technology is also plotted in the Table 6.7 to determine the difference in performance.

The power comparisons are shown in Table 6.6. Most of the hardware implementations are designed to be low power. This work has the entire context tables stored on the chip, which further increases the power usage. Most other implementations either have a software context modeler or separate RAM to store the context tables. This significantly reduces their power usage.

Design	Scaled Throughput MBin/sec	Scaled Area mm^2	Throughput per area GBin/secmm^2	Scaled Power mW	Throughput ratio
Shojania HW [23]	380	13.32×10^{-3}	28.53	5.04	0.097
Li HW [24]	1025	-	-	-	0.036
Kuo HW [25]	873	6.65×10^{-3}	131.28	4.52	0.042
Osorio HW [29]	4487-5487	-	-	-	0.008
Chen HW [28]	732	-	-	-	0.051
Liu HW [26]	357	-	-	-	0.103
Tian HW [27]	1736	-	-	-	0.021
Chen HW [30]	882	-	-	-	0.041
Liu HW [31]	898	-	-	12.6	0.041
Tsai HW [32]	3336	-	-	-	0.011
Zhou HW [33]	201	103×10^{-3}	19.563	-	0.184
Chen HW [6]	1809	46×10^{-3}	45.224	12.4	0.020
JM SW [14]	0.649	-	-	-	57.01
This work	37	3.52	10.51×10^{-3}	943	1

Table 6.7: CABAC scaled measurements

Energy comparison While some papers report their power usage, very few have energy data to compare. Liu's [31] hardware implementation is one paper with energy information. The Liu paper has throughput reported as 476 Mbin/sec and power dissipated is 23.44 mW. Taking these values, we can find that the energy per million bins is 23.44 mW $\div$ 476 Mbin/sec = 49.2 μJ per million bins [1].

When this value is scaled to 32nm, the energy is 14.33 μJ per million bins encoded. In comparison, the software KiloCore implementation dissipates 34.37 J per million bins.

[1] Liu, email message to author, March 14, 2021

Chapter 7

Thesis summary and Future work

7.1 Thesis summary

A bit-accurate implementation of Context-based Adaptive Binary Arithmetic on a many-core processor array platform KiloCore II is presented in this thesis. The CABAC encoder in question is implemented for the Main profile of the H.264 standard as described in the ITU-T — ISO/IEC [5]. The context tables are fully implemented in Dmem memories of the KiloCore II for faster access. Algorithm is broken into small tasks that are performed by individual cores. The throughput and energy of the complete implementation is comparable to hardware implementations and outperforms the software implementations by a factor of 50.

Isolating the variables to be updated in the cores that they are used in, helped to prevent lot of data movement between cores. This allowed for a degree of parallelism that can be properly exploited by the KiloCore II's architectural design.

7.2 Future work

This implementation takes advantage of the JM software to get neighbor information for the calculation of the context index. An improvement can be made to this system by storing the values of the neighbor information in additional cores. The number of neighbor macroblock information to be stored is equal to the width of the video at all times, as we need left and top neighbors of each macroblock. Once a macroblock is completely encoded, we can discard the top neighbor as it is not needed anymore.

The total neighbor information to be stored for each macroblock is 240*144 bits as specified in [6]. There also two 144 bit registers required. or each macroblock, syntax elements including mb_field_coding_flag, mb_type, mb_skip, MVD, ref_idx, ICPM, CBP and CBF, need to be stored. If we designate 9 words in Dmem for each macroblock's neighbor information, this means we need 17 cores' Dmem to store all the neighbor information. The two 144 bit registers can be 18 words in Dmem.

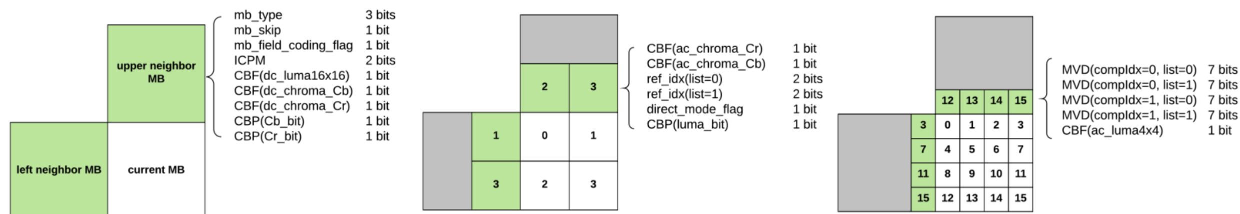

Figure 7.1: Neighbor information to be stored for ctxIdx calculation [6]

Figure 7.1 shows all the neighbor information to be stored for the calculation of ctxIdx.

7.2.1 Improving Throughput and energy

Throughput can be improved by eliminating the need for the feedback loop that the renormalization core has. The values codIRange and codILow that are updated need to be sent to the first stage of the BAE. This can possibly be avoided by attempting to include all the manipulations of codIRange and codILow in the same core. The instruction count restriction prevents this implementation from finding an optimal workaround for performing the entire BAE on a single core.

Another region of slowdown in the code is the region past the context modeller where the merger cores fan in the data. This section can be eliminated by having the input of the pState cores directly connected to each ctxIdx core's output. This means that the tables stored will be only for the syntax elements passing through that specific core. This implementation attempted a more generic context table lookup. The number of cores necessary might be significantly reduced. The context modeller core that has the largest context table to be stored would be the *ctxIdx_blockCat* core with 568 values to be stored. The possible solution is to further fan out the table lookup into three to accommodate all the table values. This would lead to only one fan-in stage unlike the current implementation where there is a necessity for two.

Improvements to energy can be made ensuring each core being used is performing to its maximum capacity of instructions. This ensures efficiency and reduces the number of cores where there are just a few instructions. This reduces the overall energy of the system. This sort of instruction packing decisions have to be made with careful deliberation keeping in mind that the functionality is not altered.

Bibliography

[1] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h.264/avc video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.

[2] Iain E. G Richardson. *H.264 and MPEG-4 video compression : video coding for next generation multimedia*. Wiley, Chichester ; Hoboken, NJ, 2003.

[3] Heiko Schwarz Detlev Marpe and Thomas Wiegand. Context-based adaptive binary arithmetic coding in the h.264/avc video compression standard. *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY*, X(Y), 2003.

[4] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. Kilocore: A 32-nm 1000-processor computational array. *IEEE Journal of Solid-State Circuits (JSSC)*, 52(4):891–902, April 2017.

[5] Claude E. Shannon and Warren Weaver. *Advanced video coding for generic audiovisual services*. ITU-T H-SERIES RECOMMENDATIONS, Urbana, Chicago, and London, 2004.

[6] Renjie Chen. Architecture and hardware for a 1 bin per cycle context-adaptive binary arithmetic coder (CABAC) encoder. Master's thesis, University of California, Davis, Davis, CA, USA, December 2019. http://vcl.ece.ucdavis.edu/pubs/theses/2019-3.renjie/.

[7] Ben Juurlink. Scalable parallel programming applied to h.264/avc decoding, 2012.

[8] Zhibin Xiao, Stephen Le, and Bevan Baas. A fine-grained parallel implementation of a H.264/AVC encoder on a 167-processor computational platform. In *IEEE Asilomar Conference on Signals, Systems and Computers (ACSSC)*, November 2011.

[9] Zhibin Xiao and Bevan M. Baas. A high-performance parallel cavlc encoder on a fine-grained many-core system. In *International Conference on Computer Design, (ICCD '08)*, pages 248–254, October 2008.

[10] Zhibin Xiao and Bevan M. Baas. A 1080p H.264/AVC baseline residual encoder for a fine-grained many-core system. *Circuits and Systems for Video Technology, IEEE Transactions on*, 21(7):890–902, july 2011.

[11] Donia Ammous, Fahmi Kammoun, and Nouri Masmoudi. A comparative evaluation between cabac and cavlc. *Journal of Testing and Evaluation*, 46:1111–1121, 05 2018.

[12] Zhiyi Yu and Bevan M. Baas. Implementing tile-based chip multiprocessors with gals clocking styles. In *IEEE International Conference of Computer Design (ICCD)*, October 2006.

[13] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. KiloCore: A fine-grained 1,000-processor array for task parallel applications. *IEEE Micro*, 37(2):63–69, March 2017.

[14] Karsten Suehring. Itu. h.264/avc reference software. http://iphome.hhi.de/suehring/tml/download/, May 2015.

[15] Tampere University. Ultra video group 4k video test sequences. http://ultravideo.cs.tut.fi/#testsequences.

[16] Xiph.org. Xiph.org video test media: derf's collection. https://media.xiph.org/video/derf/.

[17] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. Kilocore: A 32 nm 1000-processor array. In *IEEE HotChips Symposium on High-Performance Chips*, August 2016.

[18] Mark Hildebrand. Mapper2 project. https://github.com/hildebrandmw/Mapper2.jl.git, 2018.

[19] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. A 5.8 pJ/Op 115 billion Ops/sec, to 1.78 trillion Ops/sec 32 nm 1000-processor array. In *Symposium on VLSI Circuits*, June 2016.

[20] S. Sarangi and B. Baas. Deepscaletool : A tool for the accurate estimation of technology scaling in the deep-submicron era. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2021.

[21] Aaron Stillmaker, Zhibin Xiao, and Bevan Baas. Toward more accurate scaling estimates of cmos circuits from 180 nm to 22 nm. Technical Report ECE-VCL-2011-4, VLSI Computation Lab, ECE Department, University of California, Davis, December 2011. http://www.ece.ucdavis.edu/cerl/techreports/2011-4/.

[22] A. Stillmaker and B. Baas. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration, the VLSI Journal*, 58:74–81, 2017. http://vcl.ece.ucdavis.edu/pubs/2017.02.VLSIintegration.TechScale/.

[23] H Shojania and S Sudharsanan. A high performance cabac encoder. In *3rd International IEEE-NEWCAS Conference, 2005*, volume 2005, pages 315–318. IEEE, 2005.

[24] Lingfeng Li, Yang Song, T Ikenaga, and S Goto. A cabac encoding core with dynamic pipeline for h.264/avc main profile. In *APCCAS 2006 - 2006 IEEE Asia Pacific Conference on Circuits and Systems*, pages 760–763. IEEE, 2006.

[25] Chien-Chung Kuo and Sheau-Fang Lei. Design of a low power architecture for cabac encoder in h.264. In *APCCAS 2006 - 2006 IEEE Asia Pacific Conference on Circuits and Systems*, pages 243–246. IEEE, 2006.

[26] Po-Sheng Liu, Jian-Wen Chen, and Youn-Long Lin. A hardwired context-based adaptive binary arithmetic encoder for h. 264 advanced video coding. In *2007 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–4. IEEE, 2007.

[27] X.H Tian, T.M Le, B.L Ho, and Y Lian. A cabac encoder design of h.264/avc with rdo support. In *18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP '07)*, pages 167–173. IEEE, 2007.

[28] Jian-Long Chen, Yu-Kun Lin, and Tian-Sheuan Chang. A low cost context adaptive arithmetic coder for h.264/mpeg-4 avc video coding. In *2007 IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '07*, volume 2, pages II–105–II–108. IEEE, 2007.

[29] R.R Osorio and J.D Bruguera. High-throughput architecture for h.264/avc cabac compression system. *IEEE Transactions on Circuits and Systems for Video Technology*, 16(11):1376–1384, 2006.

[30] Jian-Wen Chen, Li-Cian Wu, Po-Sheng Liu, and Youn-Long Lin. A high-throughput fully hardwired cabac encoder for qfhd h.264/avc main profile video. *IEEE Transactions on Consumer Electronics*, 56(4):2529–2536, 2010.

[31] Zhenyu Liu and Dongsheng Wang. One-round renormalization based 2-bin/cycle h.264/avc cabac encoder. In *2011 18th IEEE International Conference on Image Processing*, pages 369–372. IEEE, 2011.

[32] Chen-Han Tsai, Chi-Sun Tang, and Liang-Gee Chen. A flexible fully hardwired cabac encoder for uhdtv h.264/avc high profile video. *IEEE transactions on consumer electronics*, 58(4):1329–1337, 2012.

[33] Jinjia Zhou, Dajiang Zhou, Wei Fei, and Satoshi Goto. A high-performance cabac encoder architecture for hevc and h.264/avc. In *2013 IEEE International Conference on Image Processing*, pages 1568–1572. IEEE, 2013.

[34] Subramania Sudharsanan and Adam Cohen. A hardware architecture for a context-adaptive binary arithmetic coder. volume 5683, pages 104–112. SPIE, 2005.

[35] Asma Ben Hmida, Salah Dhahri, and Abdelkrim Zitouni. A hardware architecture binarizer design for the h.264/ avc cabac entropy coding. In *2014 International Conference on Electrical Sciences and Technologies in Maghreb (CISTEM)*, pages 1–4. IEEE, 2014.