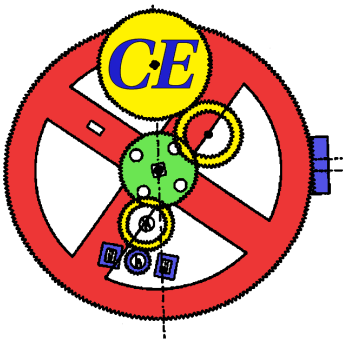


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2010

MSc THESIS

Analysis and Implementation of the H.264 CABAC entropy decoding engine

Martinus Johannes Pieter Berkhoff

Abstract

In this thesis we present an FPGA software/hardware co-design for the CABAC decoder. CABAC is the Context-based Adaptive Binary Arithmetic Coding used in the H.264/AVC video standard. This standard gives better compression efficiency, but with greater complexity and implementation cost. A large part of this cost comes from the CABAC entropy coding. The CABAC coding has a tight feedback loop between the binary arithmetic coding stage and the context modeler stage of the coding process. This means that the video stream has to be coded in a sequential way. We attempt acceleration of the CABAC decoding process in a fashionable way on dedicated programmable hardware. An FPGA implementation of the CABAC entropy decoding process is used in co-operation with the decoding software on a Xilinx Virtex 4 platform. Actual synthesis results show that our approach results in a fast and compact implementation, targeted at the state-of-the-art FPGA devices.

CE-MS-2009-04

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Analysis and Implementation of the H.264 CABAC entropy decoding engine

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Martinus Johannes Pieter Berkhoff
born in Delft, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Analysis and Implementation of the H.264 CABAC entropy decoding engine

by Martinus Johannes Pieter Berkhoff

Abstract

In this thesis we present an FPGA software/hardware co-design for the CABAC decoder. CABAC is the Context-based Adaptive Binary Arithmetic Coding used in the H.264/AVC video standard. This standard gives better compression efficiency, but with greater complexity and implementation cost. A large part of this cost comes from the CABAC entropy coding. The CABAC coding has a tight feedback loop between the binary arithmetic coding stage and the context modeler stage of the coding process. This means that the video stream has to be coded in a sequential way. We attempt acceleration of the CABAC decoding process in a fashionable way on dedicated programmable hardware. An FPGA implementation of the CABAC entropy decoding process is used in co-operation with the decoding software on a Xilinx Virtex 4 platform. Actual synthesis results show that our approach results in a fast and compact implementation, targeted at the state-of-the-art FPGA devices.

Laboratory : Computer Engineering
Codenumber : CE-MS-2009-04

Committee Members :

Advisor: Dr.ir. G.N. Gaydadjiev, CE, TU Delft

Chairperson: Dr.ir. K.L.M. Bertels, CE, TU Delft

Member: Prof.dr.ir. A.J. van der Veen, CAS, TU Delft

To my parents, for their unconditional believe in me.

Contents

List of Figures

List of Tables

List of Source Codes

Acknowledgments

For the past year I have worked with great pleasure on this Master of Science thesis project at the Computer Engineering Laboratory in Delft. I would like to give special thanks to Georgi Gaydadjiev, for guiding and supporting me throughout this project.

The support of Bert Meijs, Eric de Vries, Lidwina Tromp and Cor Meenderinck should not go unnoticed. I would also like to thank Marco van der Leije, Anthony Brandon and Chi Ching Chi for their good company during a great part of my study. During my period of study at the Delft Technical University I felt greatly inspired by Albert Einstein and Ajahn Brahmavamso Mahathera. Thank you.

I also would like to thank my parents and my sisters for their endless confidence in me. And last, but certainly not least, I want to thank my girlfriend, Suzanne Leeflang for her love and support during the period I studied Computer Engineering.

Martinus Johannes Pieter Berkhoff
Delft, The Netherlands
February 25, 2010

1 Introduction

1.1 General Introduction

H.264 represents the state of the art in current video coding standards. In the consumer electronics market it is more often adapted. We see movies in the cinemas, we rent movies on DVD or Blue Ray disks and we watch movies and our popular series every evening on the television. Nowadays we can also download our favorite movies and series from Internet and watch them on our computers, HD-television or mobile device. We can even record our own movies with a digital camcorder, edit them on our computer and show them to our family at birthday parties.

In the last couple of years there was a tremendous shift in the world of consumer video from VHS to DVD to Blue Ray and to even more exotic standards found on Internet. The demand for better quality pictures, smaller sizes, lower energy consumption, lower cost of appliances and watching movies every time, any time, anywhere has driven this shift to even better video compression standards even further.

To provide better compression of video images the Moving Picture Experts Group and the Video Coding Expert Group (MPEG and VCEG) have developed a successor to the earlier MPEG-4 and H.263 standards. The new standard is called Advanced Video Coding (AVC) and is published jointly as MPEG-4 Part 10 and H.264[14, 16]. It achieves very high compression efficiency compared to earlier standards[19]. It can handle a wide range of applications and is more friendly to networks such as the internet.

The downside of the increased compression efficiency is that the decoder complexity also grows. Context-based Adaptive Binary Arithmetic Coding (CABAC) is one of the two alternative entropy coding methods specified in H.264. The other alternative is called Context-based Adaptive Variable Length Coding (CAVLC). The H.264 standard improves the compression efficiency up to 50% with CABAC entailing a frame rate increase of 25% to 30% with bit rate reduction up to 16% [15].

1.2 Research scope

The encoding and decoding of video images can be done on many very different platforms. Encoding of video images is usually done on high performance computers, since it is required only once to encode the master video sequence to a format suitable for distribution. The decoding of the video sequence is done on many different systems, from general purpose computers to set-top boxes, mobile phones and hand held media players.

In this thesis we focus on the design of embedded hardware which improves decoding performance. We focus only on the entropy coding on the decoding side. That implies that the application will not be executed by a high performance computer, but rather on an embedded system which has to deal with limited area, speed and power. Our main goal is to make the application on the given embedded platform as fast as possible, but within the available boundaries. Power or energy consumption is in our case not as important, because we are not depended on battery power, but we assume appliances connected to the energy grid. In other cases such as embedded systems for mobile devices, energy consumption will be more important.

1.3 Problem statement

The encoder and decoder complexity is big as there is a very tight feedback loop between the context modeler and the arithmetic coder [2]. At the decoder side the feedback loop is even tighter than at the encoder side, because a model is needed to decode a symbol and the decoded symbol is needed to calculate the next model. This can be seen in figure 1.1. The main bottleneck is the arithmetic decoder since it has to process all encoded data in a sequential way. And there is no way around the arithmetic decoder, all the data in the bitstream has to pass the entropy decoder in the H.264 scheme before the other blocks such as inverse DCT and motion compensation can start decoding.

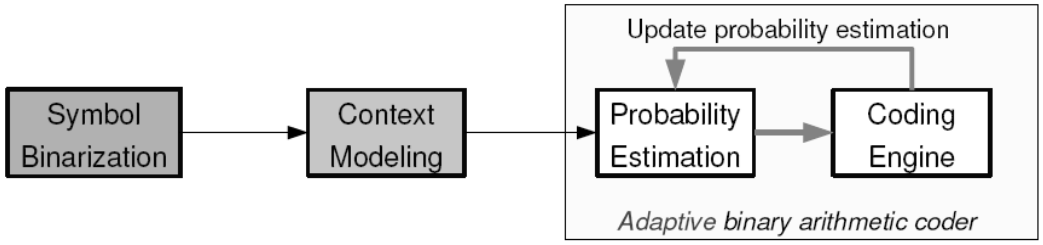


Figure 1.1: Elementary stages CABAC coding

The problem becomes bigger for higher bit rates and resolutions [2]. A video sequence with a resolution of 352x288 pixels at 30 frames per second has to produce about 1.5 million decoded symbols per second. For an HDTV resolution of 1280 x 720 at 30 frames per second this will increase to up to 50 million decoded symbols per second.

In this work we present an exploration of the hardware acceleration of the CABAC decoder. As is most common in the university environment, we did this with an FPGA implementation. Algorithms can be speed up by hardware acceleration by exploiting parallelism. But in the CABAC part of the H.264 coded there is little parallelism. We researched different techniques such as (advanced pipelining, speculative execution and data fetching). We engineered an CABAC decoder of our own on a FPGA based on a hardware and software co-design.

As all other authors with related work done on CABAC entropy decoders have

chosen to only synthesize their solutions, and therefore have only theoretical values. We decided to actual implement the CABAC entropy decoder into real FGPA prototyping hardware. The timing values measured are therefore real-life values, measured with an internal timer.

The project is done using an on beforehand chosen methodology. We have chosen to use a bottom-up approach on building the hardware / software co-design CABAC decoder implementation. The first iteration only contains a little part of the CABAC algorithm in hardware. First the interfacing with the software was tested and the performance was measured. In every following iteration the hardware CABAC decoding core was enlarged with more specific tasks. Only the time available for the MSc thesis project was the limiting factor. Further researchers can take off were we stopped.

1.4 Thesis overview

The remainder of this thesis is organized as follows. Chapter 2 provides the background of the CABAC encoding and decoding process. It introduces the different parts in the CABAC algorithm and gives us a theoretical platform. It also provides the related work.

Chapter 3 presents the overview of the CABAC decoding scheme. In this chapter we emphasize more on the practical approach of our CABAC decoding scheme. We also provide an analysis of the de-binarization stage of the CABAC decoder and conclude this chapter with our motivation for the proposed implementation.

Chapter 4 gives the detailed description of the CABAC decoder we implemented in hardware. In this chapter we describe the hardware and software co-design and how the implementation was tested.

Chapter 5 provides the verification and simulation results of the tested implementation of the CABAC decoder.

Chapter 6 gives a summary of our conclusions, an overview of the main contributions and presents directions for future research.

CABAC encoding and decoding process

2

2.1 H.264/MPEG-4 Part 10

2.1.1 Terminology

To provide a better understanding of the H.264 standard it is important to explain the terminology used in the H.264 standard [14]. A coded picture exists of an encoded field (of interlaced video) or a frame (of progressive or interlaced video). Each coded frame has its own frame number and each field has its picture order count, which defines the decoding order. Reference pictures can be used to inter predict further coded pictures.

A coded picture is made of a number of macroblocks. These macroblocks each contain 16 x 16 luma samples and the associated 8 x 8 Cb and 8 x 8 Cr chroma samples. The macroblocks are arranged in slices. A slice is a set of macroblocks in raster scan order. An I slice may only contain I type macroblocks, a P slice may contain P and I type macroblocks and a B slice may contain I-type and B-type macroblocks. Intra prediction is used to predict I-type macroblocks from decoded samples in the current slice. P-type macroblocks are predicted from reference pictures using inter prediction. The prediction of each macroblock is done from one picture. The B-type macroblocks are also predicted using inter prediction from reference pictures, but two pictures may be used to predict.

2.1.2 The H.264 Codec

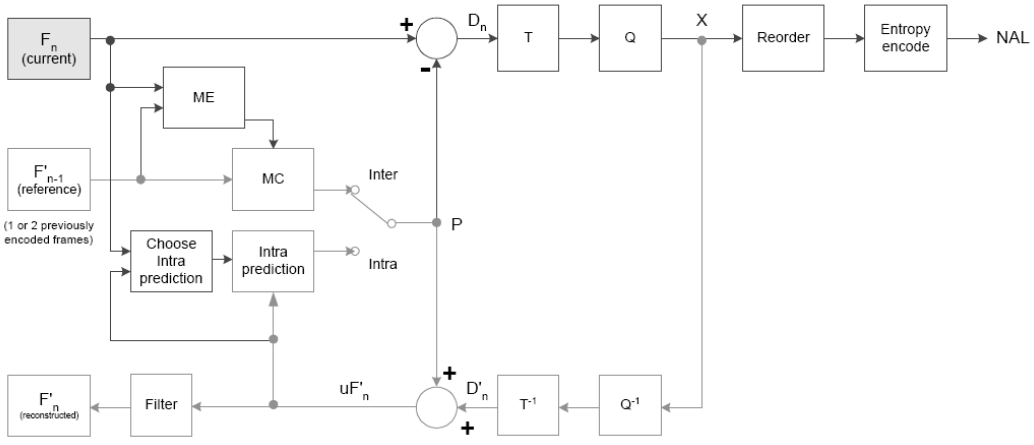


Figure 2.1: H.264 Encoder

In the coding standards h.264 does not define an encoder decoder pair but rather defines the syntax of an encoded video stream to be decoded properly. This means that everyone is free to design his or her own hardware as long as the encoded video stream can properly be decoded by any decoder. Most of the encoders and decoders will have similar basic functional elements as shown on figure 2.1 and figure 2.2.

The encoder has a forward dataflow path and a reconstruction dataflow path. The decoder has only a reconstruction dataflow path. We will look deeper into de decoding dataflow path.

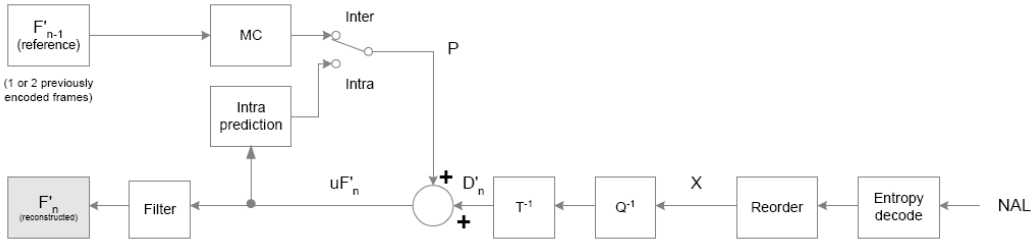


Figure 2.2: H.264 Decoder

The decoder receives a compressed bitstream from the network abstraction layer (NAL). Before the NAL the video data is stored on a harddisk or is being transmitted over a transmission line. First the compressed data has to be entropy decoded to produce a set of quantized coefficients X. These are scaled and inverse transformed to produce a residual difference block D'_n, identical to the D'_n shown in the encoder. From the decoded header information, the decoder creates a prediction block PRED. This prediction block is also identical to the prediction block PRED in the encoder. To produce uF'_n, PRED is added to D'_n. And finally to create each decoded block F'_n, uF'_n is filtered.

2.1.3 H.264 structure

The H.264 standard defines three profiles. The profiles support a particular set of coding functions as can be seen in figure 2.3. The baseline profile supports intra and inter-prediction coding and entropy coding with context-adaptive variable-length codes (CAVLC). The main profile includes support for Context-based adaptive binary arithmetic coding (CABAC), interlaced video, inter coding using weighted prediction and inter-coding using B-slices. The extended profile adds modes to enable efficient switching between coded bitstreams and improved error resilience, but does not support interlaced video and CABAC.

2.2 Entropy encoding

As could be seen in figure 2.1 on page 5 the last step of the encoding process is the entropy encode. The input frame (F_n) is processed in a series of steps towards residual (difference) blocks (D_n) and transformed and quantized to X. After a reorder step the macroblock

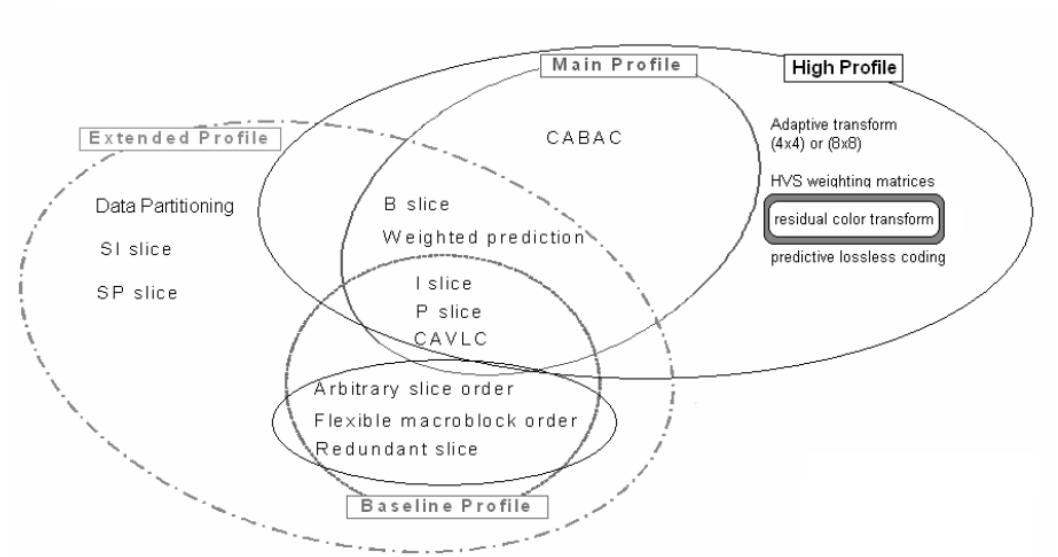


Figure 2.3: H.264 Baseline, Main and Extended profiles

units are to be entropy encoded to be sent to the Network Abstraction Layer (NAL). The compressed bitstream from the entropy encoder is made of the entropy-encoded coefficients and side information to decode each block within a macroblock. The side information includes prediction modes, quantizer parameters, motion vector information etc. The compressed bitstream is sent to the Network Abstraction Layer for transmission or storage.

In the main profile CABAC can be selected for the entropy encoding process. The alternative is CAVLC. When CABAC is selected for the entropy encoding process, the syntax elements are routed to a CABAC encoding algorithm to achieve good compression performance. This is done by:

- selecting probability models for each syntax element according to the elements context;
- adapting probability estimates based on local statistics and
- using arithmetic coding rather than variable-length coding.

As can be seen from figure 2.4 the CABAC encoding process involves the following stages: binarization, context model selection, arithmetic encoding and probability update.

2.2.1 Binarization

In the first stage of the CABAC entropy encoder the binarization stage maps non-binary symbols to a binary sequence. Most of the syntax elements that are to be encoded are represented by symbols, some syntax elements are represented by a binary code. Because

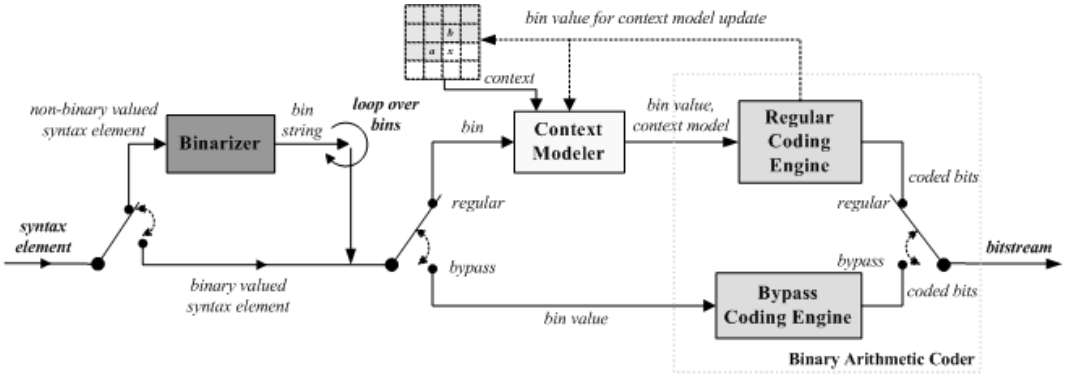


Figure 2.4: CABAC encoder block diagram

the CABAC encoder is a binary encoder only binary decisions are encoded. The process to binarize a symbol into a binary code is very similar to the process of converting a data symbol into a variable length code.

Symbol	Binarization
0	1
1	0 1
2	0 0 1
3	0 0 0 1
4	0 0 0 0 1
5	0 0 0 0 0 1
6	0 0 0 0 0 0 1
.	...
Bin_num	1 2 3 4 5 6 7 ...

Mapping to a binary sequence, e.g., using the unary code tree:

- Applies to all non-binary syntax elements except for macroblock type
- Ease of implementation
- Discriminate between binary decisions (bins) by their position in the binary sequence

⇒ Usage of different models for different bin_num in the table-based arithmetic coder

Figure 2.5: Binarization in CABAC

An example of a binarization is given in figure 2.5. All of the binarization schemes are defined by the standard.

The binary code the binarizer gives for one symbol encoded syntax element is called a bin. A binary valued syntax element is bypassed from the binarizer, because it is already a bin. The binary code, the bins, is then further encoded prior to transmission.

2.2.2 Context Model Selection

For each bin that comes from the binarization stage the context model selection stage gives that bin a context model [5, 3]. The context model is chosen from a selection of available models and depends on the statistics of recently coded bins. The context

model stores the probability of each bit in the bin being 0 or 1.

All of the context models for each syntax element are predefined in the standard. There are almost 400 separate context models for the various syntax elements. At the beginning of each slice the context models are initialized to an initial value. These initial values depend on the initial value of the Quantization Parameters (QP). These parameters have a significant effect on the probability of occurrence of the various syntax elements. The encoder may also choose one of the three sets of initialization parameters for the context model selection at the beginning of each slice to allow better adaptation to different types of video content. After each slice, the values of the context model selection stage are re-initialized [16].

2.2.3 MPS/LPS

Some of the bins encode a value that is equally probable. For this sort of encoded symbols the probability modeling used by the context model selection would be useless and even wasted overhead. As can be seen in figure 2.4 there is a bypass mechanism that allows bins to bypass the context model selection stage and go directly to the arithmetic encoding.

2.2.4 Arithmetic Encoding

Whether the bins are bypassed or are accompanied by a context model, all the bins have to be arithmetically encoded [9, 7]. The coder has been designed to facilitate low-complexity implementations of the arithmetic encoding and decoding. But although the simpler complexity of the coder, it provides improved coding efficiency compared with CAVLC.

The arithmetic coder is described in the H.264 standard and has three distinct properties:

- Probability estimation in CABAC is based on a table-driven estimator using a finite-state machine (FSM) approach with transition rules. Each probability model in CABAC can take one out of 64 different states with associated probability values. These values are found in the rLPS table.
- The current state of the arithmetic coder is quantized to a small range of pre-set values. At each step in the encoding process a new range is calculated. This makes it possible to have a lookup table for the calculation of the new range every step in the arithmetic coding process.
- There are two coding engines available. A regular coding engine and a bypass coding engine. The bypass coding engine is used for syntax elements with a near-uniform probability distribution. The chance of each syntax element in that group appearing is equal. The bypass coding engine is a simplified version of the regular coding engine.

2.2.5 Variable Length Coder

A variable Length enCoder (VLC) maps input symbols, syntax elements, to a series of codewords. These codewords are of variable length. All of the codewords must have an integral number of bits. But probabilities are almost never an integral number of bits. This makes VLC a sub-optimal solution.

The most occurring syntax elements are mapped to the shortest codewords, while syntax elements that are less common are mapped to a longer codeword. An example of Variable Length Coding is Huffman Coding. Huffman and Huffman-based codes are used in H.264 but have some serious disadvantages.

One of them is that the probability table which is used to map the syntax elements to the codewords has to be known by both sides. This creates extra transmission overhead and reduces compression efficiency. To overcome this problem pre-calculated Huffman-tables can be used which are defined in the standard. It reduces transmission overhead, but the probability-table is not as optimal as the one calculated at the end of the video sequence.

Another disadvantage is that Huffman-based codes are very sensitive to transmission errors. An error in the bitstream can cause the coder to lose synchronization and fail to decode subsequent codes correctly.

2.2.6 Arithmetic Decoding

Because the use of integral number of bits with Variable Length Coding, this coding is sub-optimal. The compression efficiency is extremely poor when symbols have a probability higher than 0.5. This can best be coded with a single bit, but the error will be high.

Arithmetic coding provides an important and practical alternative to Variable Length Coding. Arithmetic coding can more closely approach the theoretical maximum compression ratios [20]. An arithmetic coder converts syntax elements, or bins, into a single fractional number and can therefore approach the optimal fractional number of bits required to represent each symbol.

After the bins have been arithmetically encoded the coded bits form a bitstream. This bitstream is passed to the Network Abstraction Layer for transmission or storage. If the bins have been arithmetically encoded using the regular coding engine and not the bypass coding engine, the selected context model has to be updated for the following bins to be encoded.

2.2.7 Probability Update

Successful entropy coding depends on accurate models for symbol probability. If the bins have been arithmetically encoded using the regular coding engine the outcome must be

fed back to update the context modeler. E.q. if the resulting bin value was 1, the frequency count for that particular context model is increased.

2.3 Related work

In this section we will discuss previous research on the topics discussed in this thesis.

In [19] the author provides an overview of the technical features of the H.264/AVC standard. It describes profiles and applications for the standard and outlines the history of the standardization process. H.264/AVC is the newest video coding standard of the ITU-T Video Coding Experts Group. Figure 2.6 shows the basic coding structure for H.264/AVC for a macroblock. The input video signal is split into macroblocks. Macroblocks are associated to slice groups and slices. After that each macroblock is processed. Parallel processing of macroblocks in different slices is possible. The coder consists of spatial and temporal prediction, transform coding, quantization and finally entropy coding. Inside the encoder exists a decoder to make the prediction and motion compensation more efficient. The entropy coding is where the CABAC coding takes place. The decoding of a coding stream is the inverse, with first the CABAC decoding stage.

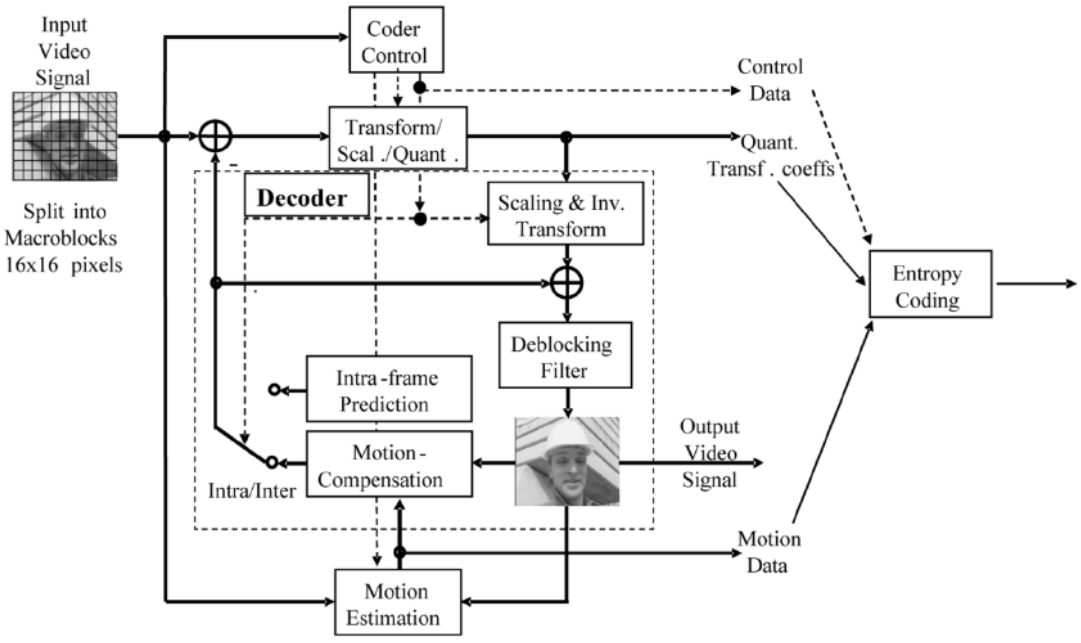


Figure 2.6: Basic block structure for H.264 macroblock encoding

In [6] the author describes the Context-Based Adaptive Binary Arithmetic Coding (CABAC) of the new H.264/AVC coding standard. In figure 2.7 the CABAC encoder

block diagram is shown. The different stages in the encoding process are shown. First the binarization stage of converting a syntax element to a binary string, a bin. The next stage is the context modeling stage where the bins are assigned a model probability according to the model probability distribution. The outcome of each context model prediction is dependent on the result one bin before. The sequential feedback loop makes the coding process hard to parallelize. The final stage is the binary arithmetic coding stage. Here the bin values are coded into the bitstream. The binary arithmetic coding stage is based on the principle of recursive interval subdivision. After the final stage the bitstream is ready to be stored or sent over a network. The decoding of the bitstream is the inverse of the encoding.

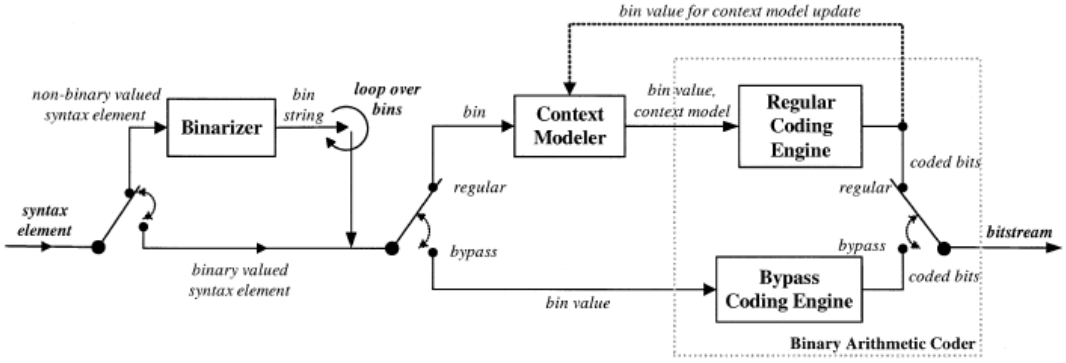


Figure 2.7: CABAC encoder block diagram

In [33] the author proposes a high performance hardware architecture of the CABAC decoder. The paper takes advantage of one of the characteristics of CABAC decoding. The occurring frequency of certain syntax elements make it feasible to accelerate the most occurring bins in a macroblock. The new decoding architecture can decode two regular bins together with one bypass bin in one cycle. In figure 2.8 the organization of the elementary decoding engines for one bin is shown. It shows a two-hierarchy decoding tree for decoding two regular bins RB1 and RB2; and a two-hierarchy decoding tree for decoding two bypass bins BP1 and BP2. In figure 2.9 the author shows his basic decoding circuit kernel. It shows the four decoding engines together with the control signals. Ctx1 and ctx2 are the two context models for RB1 and RB2. The main points of the architecture are:

1: Four different values of rLPS are prefetched in the previous cycle. This means that in the current cycle only two bits of range has to be used to select the right rLPS.

2: The decoding procedure for RB2 is carried out in parallel with the decoding procedure of RB1.

3: The most significant bit (MSB) of the result is used to determine if MPS of LPS happens. This saves a nine bits comparator in the decoding engine.

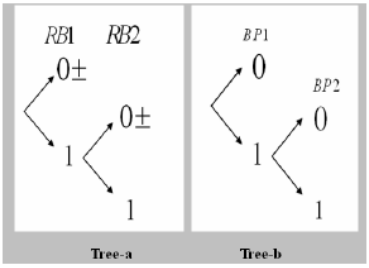


Figure 2.8: Two hierarchy decoding tree, with regular bin (RB) and bypass bin (BP) decoding

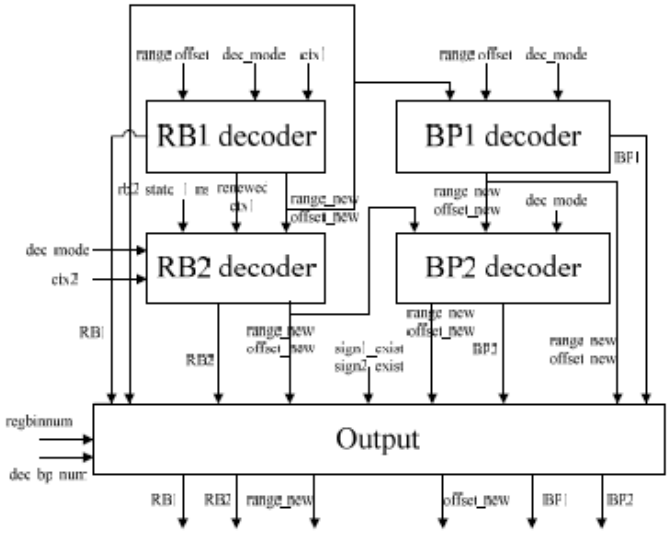


Figure 2.9: Basic CABAC decoding circuit units

In [32] the author proposes a new approach to the CABAC decoding procedure. Since CABAC decoding is highly sequential and has strong data dependencies, it is difficult to exploit parallelism and pipelining. By modifying the operation chain the author was able to enable both parallel operations and pipelining. In figure 2.10 the elementary operations of the CABAC decoding process are shown. Figure 2.11 shows the proposed pipeline arrangement, where a data hazard exists due to the context model. It is resolved by forwarding the changed context model. The data hazard caused by the context selection is avoided by inserting a stall. This stall cannot be avoided. The structural hazard caused by the context model loading and the context model update are resolved by the context model reservoir. Several context models are simultaneously loaded from memory, while context selection is performed in parallel. In figure 2.12 the data arrangement is shown of the context memory. By modifying the data arrangement in memory and utilization of a context model reservoir (CMR) two stalls from the proposed pipeline arrangement are removed. Figure 2.13 shows the essential parts for the proposed CABAC decoding architecture.

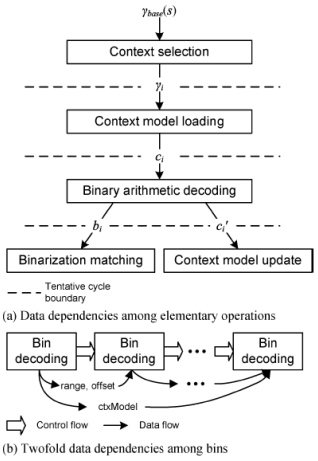


Figure 2.10: Elementary operations of CABAC decoding and their data dependencies

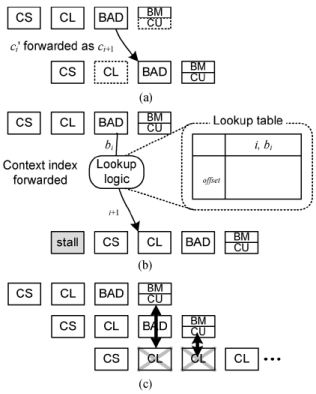


Figure 2.11: Pipeline hazards. (a) Data hazard due to context model. (b) Data hazard due to context selection. (c) Structural hazard caused by CL and CU

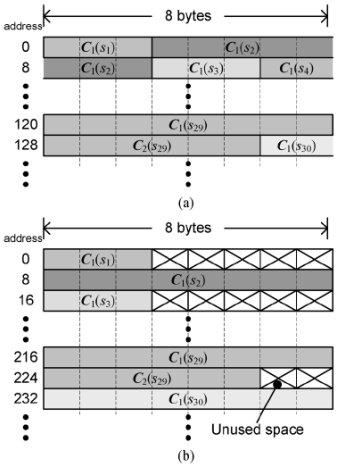


Figure 2.12: Modification of the data arrangement of the context memory

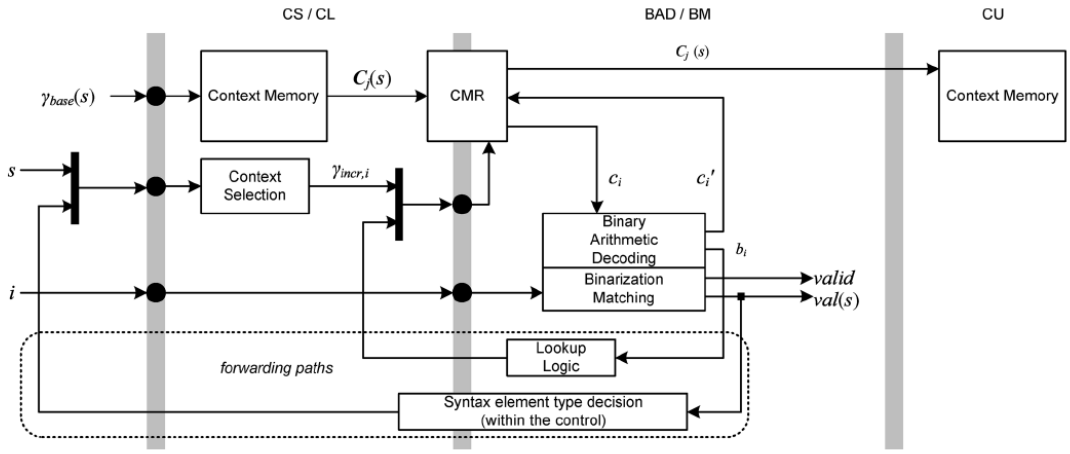


Figure 2.13: Essential parts of the CABAC decoding process, with the different pipeline stages

In [31] the author present a high throughput architecture for CABAC decoding. In figure 2.14 and figure 2.15 the new proposed architecture can be seen. To speed up the inherent sequential operation, the processing bottleneck is broken down by a look-ahead codeword parsing technique. This technique is used on the segmenting context tables with cache registers. The look-ahead parsing detection (LAPD) is used to detect two conditions. If these conditions are met, a second bin is generated in the same cycle. This also means a more efficient way to access memory is needed. This is done by partitioning one context table into multiple segmented context memories.

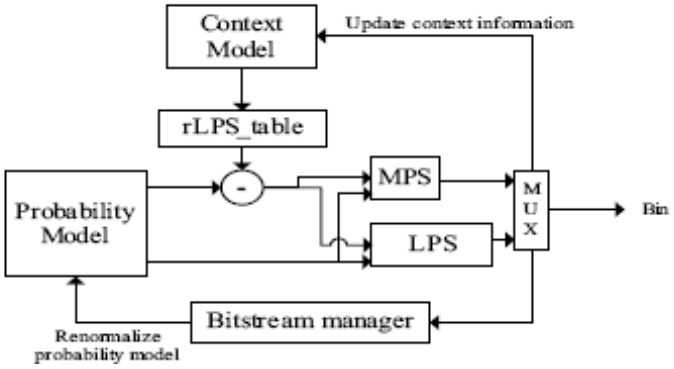


Figure 2.14: Original decoding decision flow

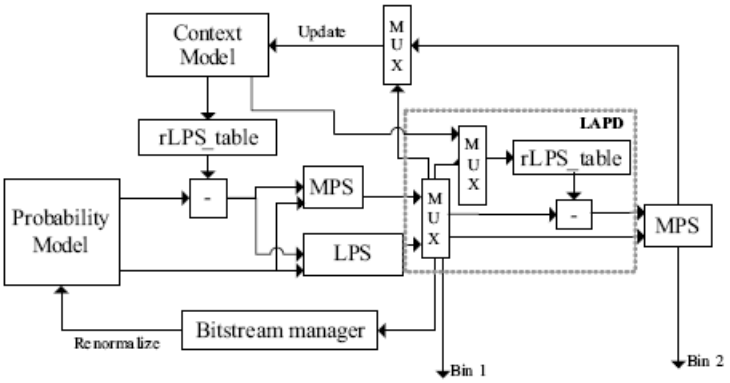


Figure 2.15: Proposed decision flow with look-ahead codeword parsing

In [4] a CABAC decoder using most probable symbol prediction is proposed. Figure 2.16 shows the proposed CABAC decoder. Analysis of variable changes shows MPS decoding a bin is usually followed by no renormalization, while a LPS decoding is always followed by a renormalization. On this conclusion a decoding engine is proposed which decodes two bins at a time. The first binary symbol is decoded as the conventional scheme and the second one is decoded with predicting that the first symbol is the MPS. The decoder includes two BADs and reads two sequential context models at a time.

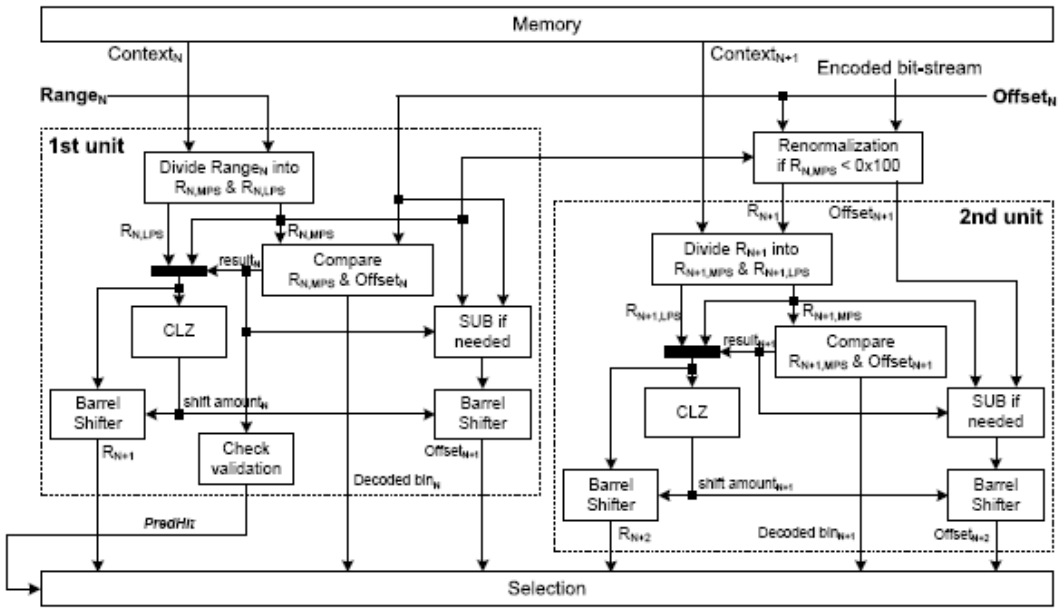


Figure 2.16: The proposed CABAC decoder architecture with most probable symbol prediction

In [5] a compact hardware architecture for CABAC decoding is presented. The architecture as shown in figure 2.17 uses the similarities between the encoding and decoding algorithms to achieve remarkable hardware reuse. Also a dynamic pipeline scheme is implemented which increases the processing throughput. Dual-port SRAM is utilized to store the 399 context models. The relative context is updated each time a context write appears. The three layers of registers are correspondent to six tasks and belong to three pipeline stages. In the best case a bin is encoded or decoded in one cycle, in the worst case in two cycles.

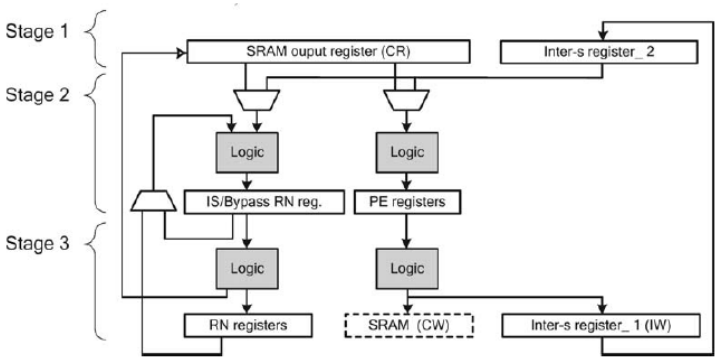


Figure 2.17: Architecture of the binary arithmetic coding engine

In [34] the author proposes an efficient CABAC decoding architecture using parallelism. The parallelism includes line-bit-rate decoding, multiple bin arithmetic decoding and an efficient probability propagation scheme. Figure 2.18 shows the top level architecture of the CABAC decoder. The decoding is done at line-rate in stead of fix bin rate decoding. This saves large bin buffers.

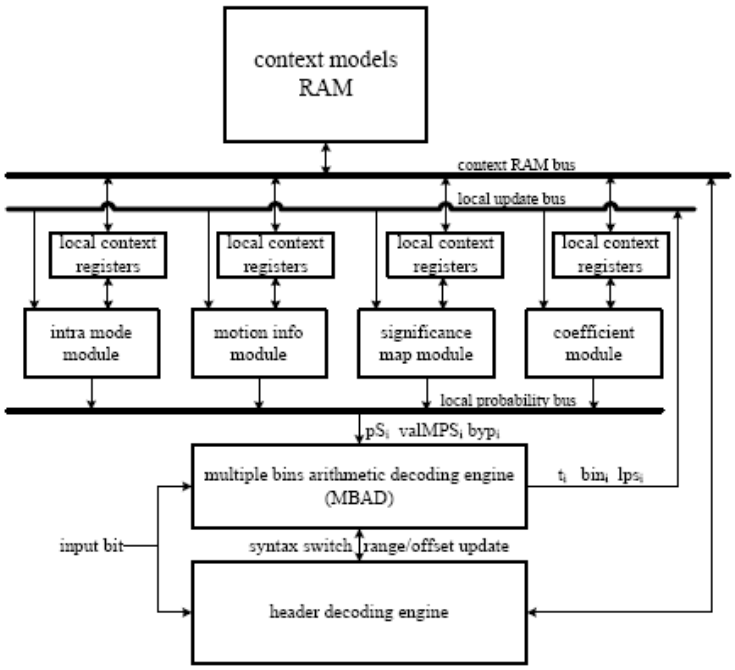


Figure 2.18: Top level architecture of the proposed CABAC decoding engine

In [35] the author presents a novel hardware design for the CABAC decoding engine. The data hazards are analyzed in current CABAC decoding and are resolved using pipeline-based architecture. Standard look-ahead technique is used in parallel with a context maintainer. Figure 2.19 shows the sequential and the pipelined bin decoding. The processing is separated in two stages. Stage 1 is responsible to provide probability information. Stage 2 fulfills the arithmetic decoding and context updating. Stage 1 is performed twice, predicting a LPS and a MPS. The architecture can perform one bin decoding per cycle, but the cycle time can be twice as small as the sequential bin decoding.

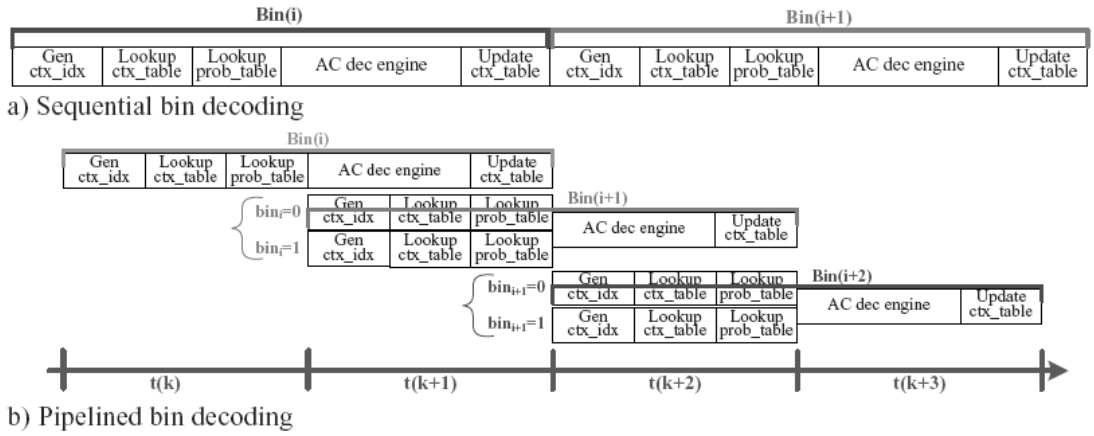


Figure 2.19: Sequential and pipelined bin decoding flow

In [2] the author presents an innovative hardware implementation of the CABAC decoder. Through the use of speculative prefetching and aggressive pipelining a decoder capable of decoding one syntax element per clock cycle was achieved. Figure 2.20 shows the original decoding flow diagram. The memory operations are labeled with sequential numbers. Figure 2.21 shows the decoder algorithm after speculative fetching all next possible states. This is done to make sure all potentially needed information is available with a single read operation. The decoding calculations can now start after one read operation.

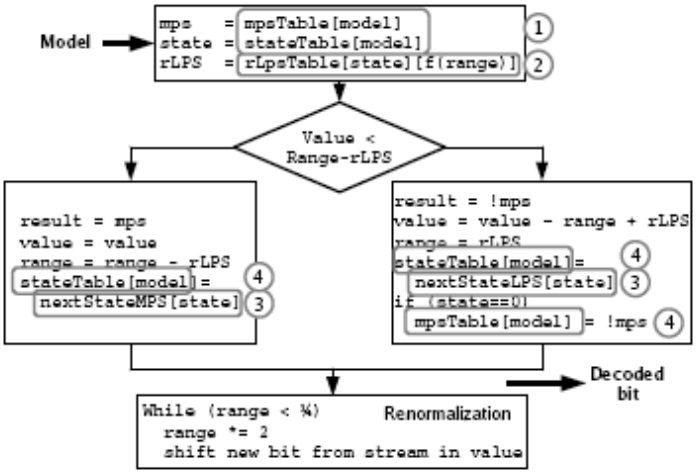


Figure 2.20: Flow diagram of the CABAC decoding process. The memory operations are labeled with sequential numbers

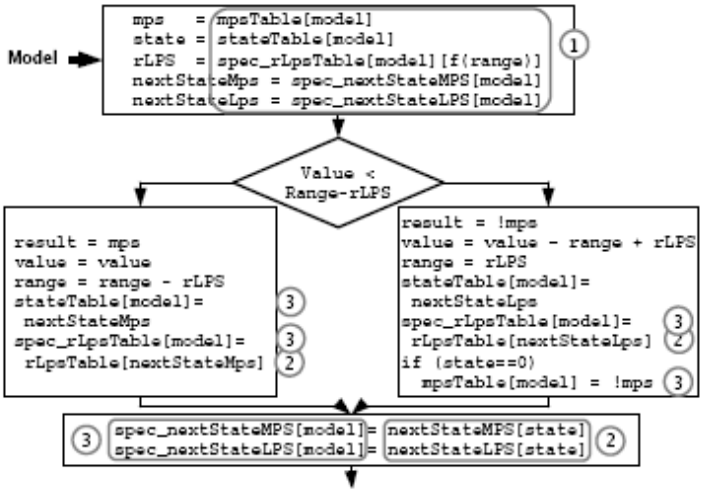


Figure 2.21: Decoder algorithm with speculative fetching and renormalization phase

In [1] a hardware accelerator is proposed for the CABAC decoding. A new efficient memory system is proposed for easy integration with other video components. In figure 2.22 such a macroblock memory is shown. The memory is a dual-port SRAM and stores the syntax elements of 24 macroblocks. These are also used by motion compensation and intra prediction. The memory is read in a 2d-wave fashion. When the decoder starts to decode macroblock A25, it will first read in macroblock A4 into memory and write macroblock A24 into memory. This is used to adequately use the motion compensation and intra prediction subsystems.

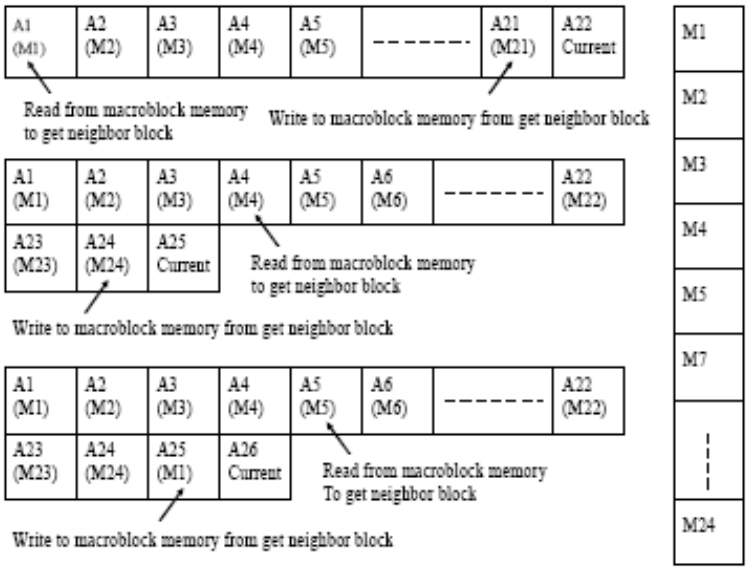


Figure 2.22: Macroblock memory configuration

In [18] the author proposes a system-on-chip software / hardware co-design of the CABAC decoder. In figure 2.23 the software / hardware architecture of the CABAC decoder can be seen. A network abstraction layer (NAL) is used to communicate between the software and the hardware. Three tables, the state transition table, the rangeLPS table and the initialization table are implemented in combinational circuits. A dual-port SRAM is used for reading and writing the context models, which can take place at the same time. Residual data for a macroblock is stored in a single-port SRAM.

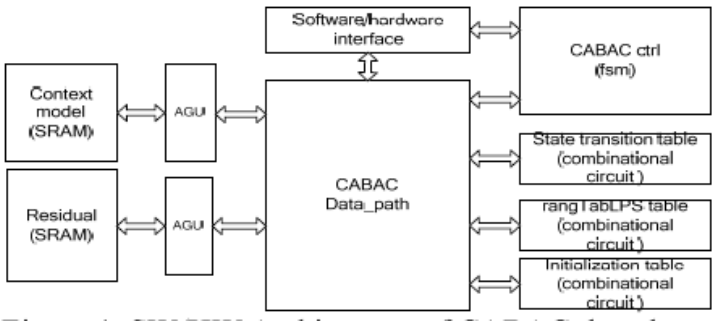


Figure 2.23: Software / hardware architecture of the CABAC decoder

In [13] presents an architecture to decode CABAC and CAVLC. Combining two decoding modes the architecture saves space as the logic and storage elements are shared. In figure 2.24 the block diagram of the double-mode binarization unit can be seen. It consists of four stages: 1. selection stage, where input data are submitted through dedicated ports, 2. mapping stage, where syntax elements are maps onto their binary representations, 3. assembling stage, where all code strings are forwarded in the next stage on one of two paths. The first path supports CABAC mode, the second path supports the CAVLC mode. 4. NAL stage, where the code streams produced by the binarization and CABAC paths are combined and encapsulated into network abstraction layer (NAL) units.

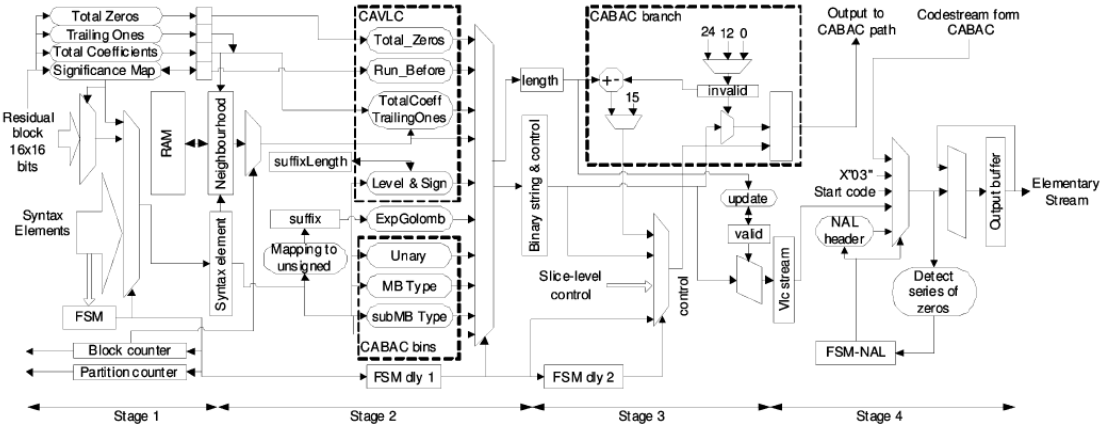


Figure 2.24: Block diagram of the double-mode binarization unit

In [12] an FPGA architecture for CABAC decoding is proposed. It consists of a multi core system. An FPGA accelerator takes care of the arithmetic decoding while a large number of microprocessor cores implement the parallel tasks. The parallel tasks are at the macroblock level and the frame level of the H.264 algorithm. The macroblocks can be decoded in a diagonal way, which enables parallel decoding of the macroblocks. In figure 2.25 the proposed architecture can be seen. The architecture has two pipeline stages, with very short cycle length. The critical path is defined by memory access and range updating. The program memory is controlled by the fine-grain control and context managing, the range updating by the state memory.

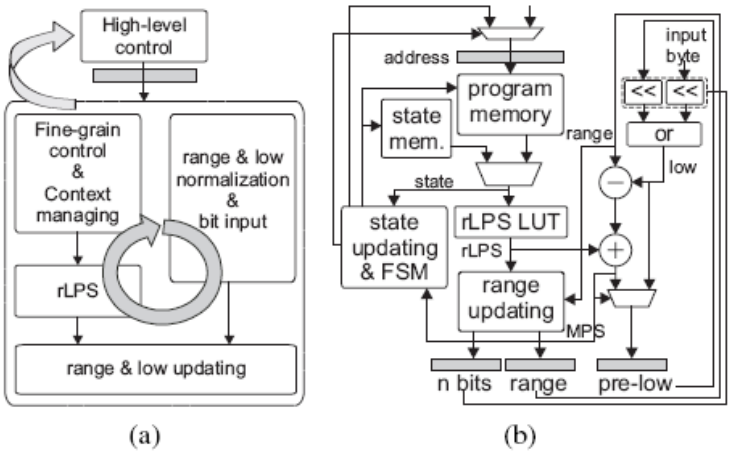


Figure 2.25: Proposed CABAC decoding architecture

Overview of the CABAC decoding scheme

3

3.1 CABAC Encoding Steps

As could be seen in figure 2.4 in the previous chapter the CABAC encoding process consists of the following three steps:

- binarization;
- probability modeling;
- binary arithmetic coding.

In the binarization process of the CABAC encoding a given non-binary valued syntax element is uniquely mapped to a binary sequence [32, 31]. This binary sequence is called a bin string. If the syntax element is already a binary sequence this binarization process can be bypassed. In the probability modeling process the bin string enters and a probability model is selected. The choice of the probability model may depend on previously encoded syntax elements or bins.

After the selection of the probability model the bin enters the arithmetic coding process where the bins are entropy coded into the bitstream. In the binary arithmetic coding process also the model update takes place for the subsequent bins in the probability modeling process. The two last steps can also be bypassed if there is no need for a probability modeling. This can be the case if there is equal probability of the value of the syntax elements. The encoding of the bin values takes place in the bypass coding engine.

3.2 CABAC Decoding

The CABAC decoding process is the inverse of the CABAC encoding process [13, 12]. First is the corresponding context model selected to decode the bin. The bin is then decoded using the arithmetic decoding engine. The arithmetic decoding engine is quite similar to the binary arithmetic encoding engine.

3.2.1 FFmpeg

The CABAC decoder is based on the H.264 standard and on the FFMPEG implementation of the standard. FFmpeg is a complete, cross-platform solution to record, convert and stream audio and video. It includes libavcodec, a leading audio/video codec library. FFmpeg is free software and is licensed under the LGPL or GPL. The libavcodec includes a highly optimized version of the CABAC-decoder for the Intel

processor platform, but we used the less optimal general implementation of the H.264 decoder with the CABAC-decoder in it.

As we have seen in the second chapter, each H.264 video sequence consists of frames. Each frame is build up out of one or more slices and each slice can have one or more macroblocks. Macroblocks are the units that carry the 16x16 luma samples and associated 8x8 Cr an Cb chroma samples. When the video sequence reaches the CABAC-decoder, it is just received by the Network Abstraction Layer either from transmission or from storage. The video sequence consists of a bitstream of encoded and compressed syntax elements. These syntax elements are only readable after the first step in the decoding process. Then these syntax elements can be used to reconstruct the original frame. The first step is the entropy decoder, in our case CABAC (Context-based Adaptive Binary Arithmetic Coding).

3.2.2 Context Model Selection

The first step in the decoding process is to initialize the CABAC-decoder [11, 10]. This is done every time a new slice starts. Together with the encoded syntax elements or the bins, there is extra information sent with the bitstream. For example the Quantization Parameters are sent with the bitstream. The initial values of the Context Model Selection table are depended on this Quantization Parameters. The initial value of the Context Model Selection table is also depended on some other parameters, which increases adaptation to different types of video content.

There are a total of 366 Different Context Models which are all initialized into the table at the beginning of each slice. With the different parameters there are a large number of different tables that could be selected to be the initial table for the Context Model Selection table.

3.2.3 Coding engine

The coding engine consists of two registers, named Range and Low (or Value)[20]. At the beginning of a decoding sequence, i.e. at the beginning of a new slice, the coding engine is initialized. The range is set to 0x1FE. In the low register the first 9 bits of the bitstream are loaded. The CABAC engine is now initialized and can be used to decode the bitstream to bins. Bins are a string of bits that represent a syntax element. Some syntax elements are just the bits found in the bin, but other syntax elements are represented as symbols and should be de-binarized.

The decoding engine is being called either in regular mode or in bypass mode. In the bypass mode there is no use of the context model selection table. In the regular mode the decoding engine has to know which context model or state to use. The state is the value found in the context model selection table at a specified index. Every bit is decoded with the same or a different state as the previous decoded bit in the bins.

We now have values for Range, Low and State and the arithmetic decoder can do a first iteration.

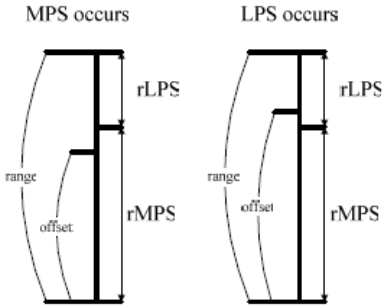


Figure 3.1: Arithmetic decoding engine for one bin

Every iteration of the arithmetic decoder will have one bit as a result. What the result is, depends on the value of Low compared to Range. While the range register keeps track of the width of the current interval, the Low register keeps track of the input bitstream. The range is split in two intervals: rLPS and rMPS. The rLPS is the estimated probability interval of the Least Probable Symbol. rMPS is the estimated probability interval of the Most Probable Symbol.

The rLPS value is read from a fixed table and indexed by the first two bits of the range value and six bits of the state value. The value of the input bitstream, named Low, falls into one of the two intervals, rLPS or rMPS. This decides whether the bit is decoded as a LPS or a MPS symbol. The results depend further on the LSB of the value state. If the result is MPS than the LSB of the value state is the output bit. If the result is LPS than the output bit will be the value of the LSB of the state inverted.

Figure 3.1 shows the case that MPS occurs and the case that LPS occurs. MPS occurs if the Low is less than rMPS and LPS occurs if the Low is greater or equal to rMPS. After this iteration the values of range and low have to be renewed by the equation (3.1).

$$\begin{aligned} if MPS \begin{cases} range\_new &= rMPS \\ low\_new &= offset \end{cases} \\ else \begin{cases} range\_new &= rLPS \\ low\_new &= offset - rMPS \end{cases} \end{aligned} \tag{3.1}$$

After this renewal step the next iteration can take place. To keep the precision of the decoding process, the MSB of range has to be always 1. To ensure this, the value of range has to be renormalized when detected a zero as MSB. The renormalization process shifts the value of range to the left, so that the MSB of range is again 1. The last bits are stuffed in as zeros so that the value remains 9 bits. The value of Low also shifts the same amount as the Range to the left. The Low register however receives

the new bits at the LSB position from the input bitstream. This way the Low register receives bits from the input bitstream and keeps track of the position of the input bitstream in the current interval.

In the bypass mode no context model is needed because of the equal probability of the syntax elements. The probability of the LPS is in this case 0.5. But we can compare the value of Low with the value of Range divided by two.

3.2.4 De - Binarization

In the last phase of the CABAC decoding the resulting bits from the decoding engine are taken and de-binarized. A sequence of bits can form a bin which can be translated to a symbol. This symbol represents the syntax element that was encoded. Not all bins and thus syntax elements are represented by a symbol, some are just the string of bits they were in the bin.

To de-binarize the bins the bitstream has to go through a decoding tree. We dont know on before hand where every bin starts and where they end. We dont know which bits from the decoding engine together form a syntax element which is represented as a bin. This makes it hard to parallelize and very time consuming. The whole tree has to be walked in order to get the right syntax element or symbol.

3.3 Motivation

The total CABAC decoder consists of three main stage: context model selection, arithmetic decoding and de-binarization. In this thesis we are going to research the arithmetic decoding engine. We are going to implement the arithmetic decoding engine into hardware and let it run in a software / hardware co-design.

The main reason why we choose the arithmetic decoding engine to be implemented in hardware is the fact that it has very strong uniform, iterative data dependencies between all stages in the algorithm. Every decoded bit is depended on all the previous decoded bits in the same slice. This is because for every decoded bit in a slice, the context model selection table is updated. And the next bit to decoded can be depended on that updated value in the context model selection table.

We like to see how fast we can make a software hardware co design implementation of the arithmetic decoding engine. We would also like to see what the speedup is and how we can arrange the architecture of the hardware implementation in such a way that we get the best increase in speed.

As we focus on the arithmetic decoding engine, we only look at one slice to decode, so we initialize the context model selection table only once. We also dont bother ourselves with the de-binarization phase of the CABAC decoder. This would however be

a very good topic for further research. We could add the de-binarizer to our arithmetic decoding engine and measure if we could get an addition speedup from an intelligent de-binarization architecture.

Since every slice is independent of each other in terms of CABAC decoding, major improvement can be achieved with parallelism on the level of slices. Every frame is made up of one or more slices and every slice is made up of one or more macroblocks. The focus in this thesis is to accelerate the decoding of independent slices. Several slice accelerators could be used in parallel to achieve higher frame decoding rates. The main bottleneck in these slice accelerators is the CABAC decoding stage. To accelerate the whole slice decoding, acceleration of the CABAC decoding is necessary. This is done by the making of specialized hardware for the CABAC decoding, in stead of decoding CABAC on a general purpose processor.

3.4 Conclusion

In this chapter, we presented an overview of the CABAC decoding scheme. The CABAC decoding scheme is based on the FFmpeg implementation of the standard. The arithmetic background of the coding engine has been shown as well as the software implementation of the coding engine. A motivation has been given as why to implement the coding engine into hardware.

Implementation of the CABAC decoder

4

4.1 Overall system description

4.1.1 Introduction

The CABAC decoder is build around the Xilinx ML410 evaluation board [17, 25, 22, 27, 8]. This board includes the Xilinx Virtex 4 FPGA. This FPGA has two PowerPC 440 processors of which we will use only one. A part of the CABAC decoder we will make in hardware. The part that we want to accelerate is being build in hardware on the FPGA and the rest of the CABAC decoder is run in software on the PowerPC processor (also residing on the Virtex 4 FPGA). We only implement a part of the total H.264 video decoder. The part we want to test is the CABAC decoder. So this system is only capable of producing test-results and can not actually decode a video stream. It therefore misses the Network Abstraction Layer (NAL) and the H.264 decoder parts after the CABAC decoder.

The CABAC decoder software which is run on the PowerPC will delegate some computational intensive parts of the algorithm to the custom build hardware [28, 29, 21, 24]. This hardware on the FPGA is specially build to accelerate that part and can only be used to accelerated that part of the software. The hardware resides on the FPGA and communication between the PowerPC processor and the hardware accelerator is done over the Auxiliary Processing Unit (APU) bus of the processor. This bus is specially designed to incorporate custom hardware accelerators onto the processors local system. The hardware accelerator can be handled by the processor via the APU through the use of a special processor instruction.

4.1.2 Validation

To validate and verificate the different parts of the system, the system was tested with predefined test vectors. The hardware made in VHDL was simulated in ModelSim. A testbench was written to validate the correct operation. Different input vectors were made and put into the system. The input vector were first run trough software, so the software could be compared to the hardware output.

On the level of hardware/software co-design, the system was run on the Xilinx ML410 development board. Since large parts of the H.264 video decoding algorithm were not implemented in the software, they were out of the scope of our research, we couldn't test the system with actual video streams. Testvectors of the videostream were made by pointing in the original software the input and output of our total system. This way we made testvectors from real videostream, but only the parts that needed to be tested. Again the outputs were compared for validating the correct operation.

4.2 Different parts in the system

4.2.1 Hardware Accelerator (cabac decoder)

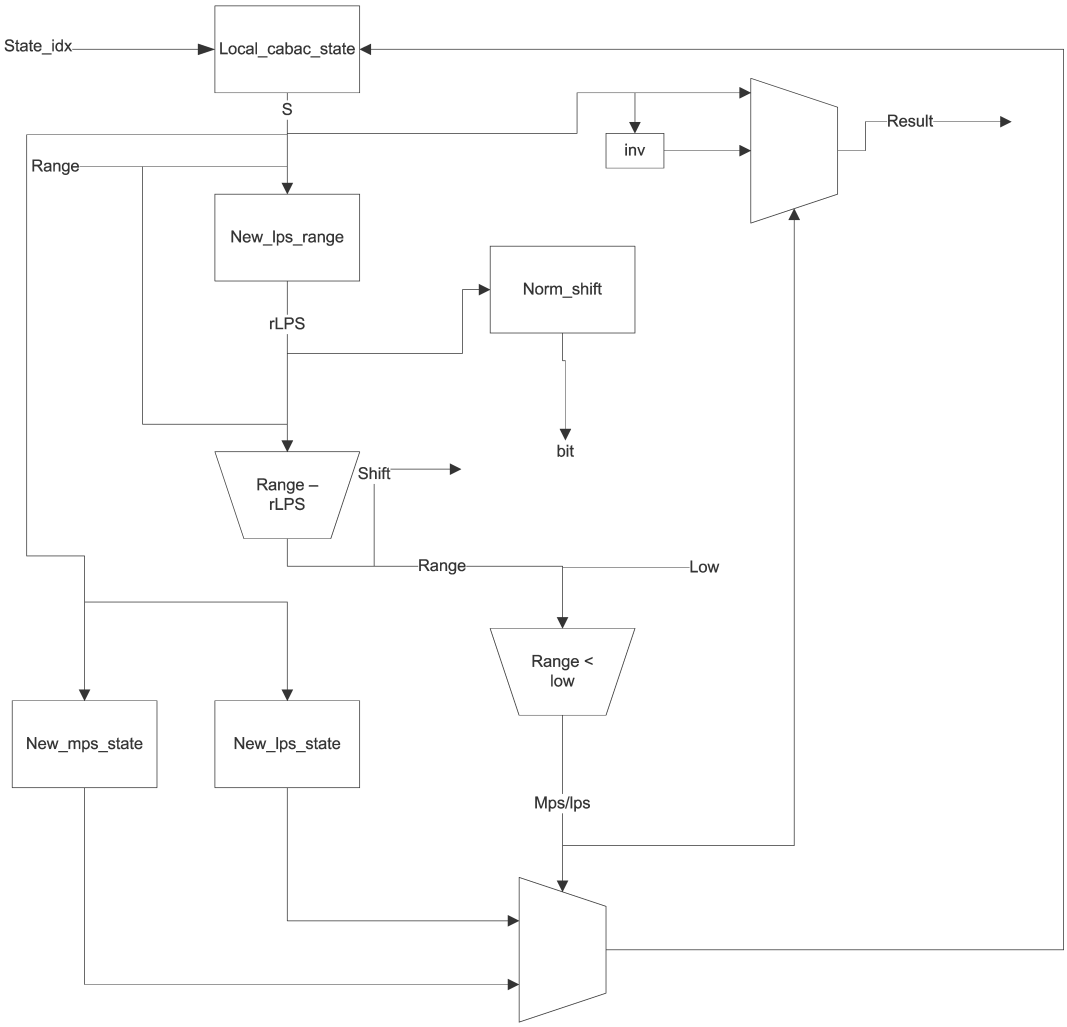


Figure 4.1: Hardware accelerator

The hardware accelerator as built in the FPGA can be seen in figure 4.1. To be able to run this architecture on the FPGA it had to be arranged in a sequential way. In figure 4.2 the sequential architecture can be seen. The decoding of one symbol cost one cycle. And every cycle the registers are updated. Figure 4.3 shows the finite state machine (FSM) for the Cabac hardware accelerator.

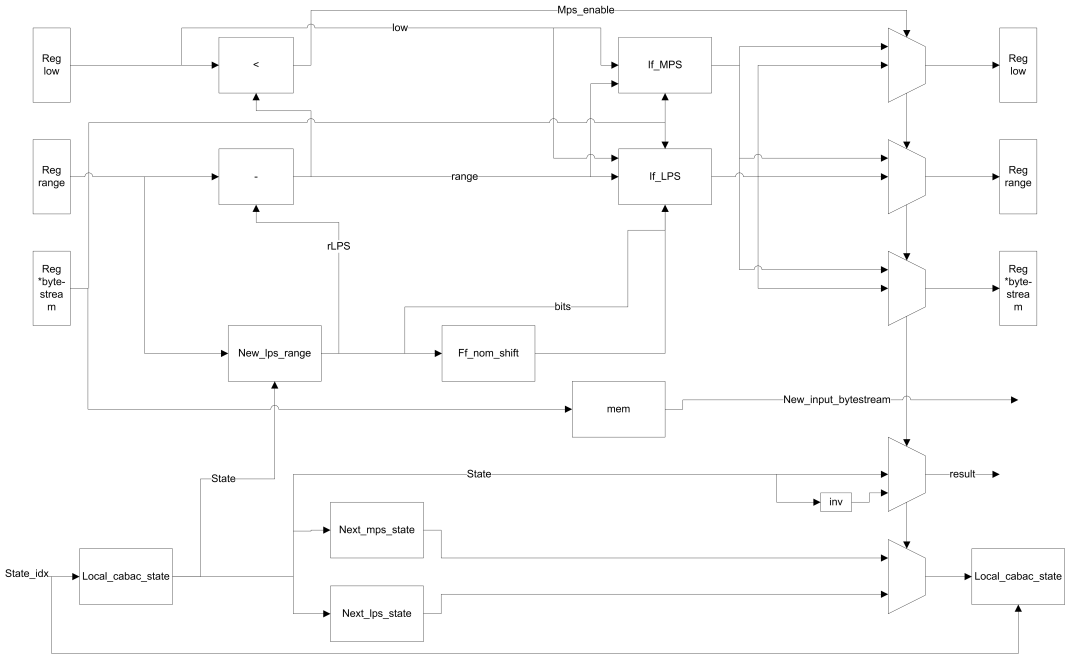


Figure 4.2: Sequential hardware accelerator

Figure 4.3: CABAC CCU Finite State Machine

4.2.2 CPU HW/SW (ppc, bootloader)

The processor the CABAC decoder software is run on is the PowerPC 405 processor. It is a hardcore processor embedded in the Virtex-4 FPGA. It is run at a clock speed of 300 MHz.

4.2.3 APU Controller

The Auxiliary Processor Unit (APU) controller allows us to extend the native PowerPC405 instruction set with custom instructions [23, 26, 30]. These instruction are executed by an FPGA Fabric Co-processor Module (FCM). In our case the FCM is the CABAC decoder Hardware Accelerator. It enables a very tight integration of the hardware accelerator with the processors pipeline. The PowerPC core, the APU controller and the FCM can be seen in figure 4.4.

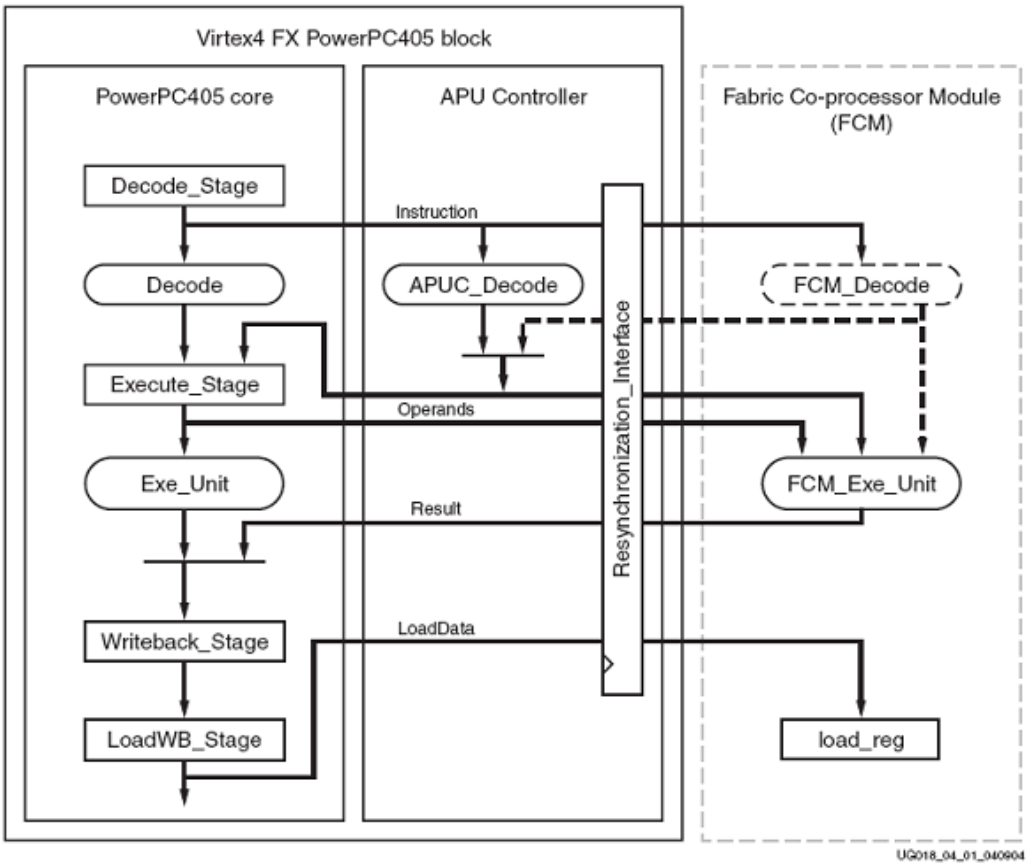


Figure 4.4: PowerPC core, the APU controller and the FCM

The APU controller serves to perform clock domain synchronization. The PowerPC405 Core runs on a much higher clock frequency than the slower FCM or our CABAC hardware accelerator. The PowerPC runs at a clockrate of 300 MHz and the CABAC hardware accelerator can in this stage not run any faster than 25 MHz. The

APU has a clock ratio setting of 1:12. If the hardware accelerator is done useful work, the processors pipeline is being stalled until the hardware accelerator is done working. The APU also decodes the specific FCM instructions and notifies the CPU or the CPU resources needed by the instruction.

4.2.3.1 Instructions

For the CABAC decoder hardware accelerator we used two custom instruction to communicate between the processor and the FCM or hardware accelerator. The instructions all have the general instruction format as can be seen in figure 4.5.

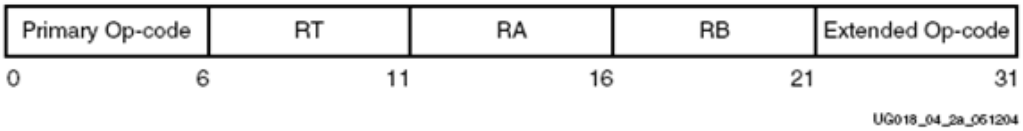


Figure 4.5: Instruction format

To communicate from the processor to the hardware accelerator we used a load instruction, lwfcmx(rn, base, adr). With this instruction we can load an integer to a specific register inside the hardware accelerator. To communicate from the hardware accelerator to the processor we used a store instruction, stwfcmx(rn, base, adr). This instruction reads the value of a defined register and stores it to a local register.

An example can be seen in listing 4.1.

```
1 Int I = 0;
  lwfcmx(0, &i, 0);
  stwfcmx(0, dst, i*4);
```

Listing 4.1: Load instruction between processor and hardware accelerator

This short code listing will send the value of I to the hardware accelerator and wait for an answer. It will store this answer into the register dst. This way the processor can communicate with the CABAC decoder hardware accelerator on a very simple and effective way.

On a hardware level it looks like figure 4.6.

In the first cycle the instruction is sent from the APU to the FCM (APUFCMINSTRUCTION). In the next cycle the APU will send the data, the value of the integer, to the FCM. In the meantime the processors pipeline is stalled. When the FCM has received all the information correctly it will sent back a FCMAPUDONE and the processor can execute the next instruction.

The next instruction in our example is a store instruction, which can be seen in figure 4.7. The instruction is sent to the FCM (APUFCMINSTRUCION). The instruction is decoded and the result is sent back to the APU (FCMAPURESULT) and the FCM reports it is done (FCMAPUDONE).

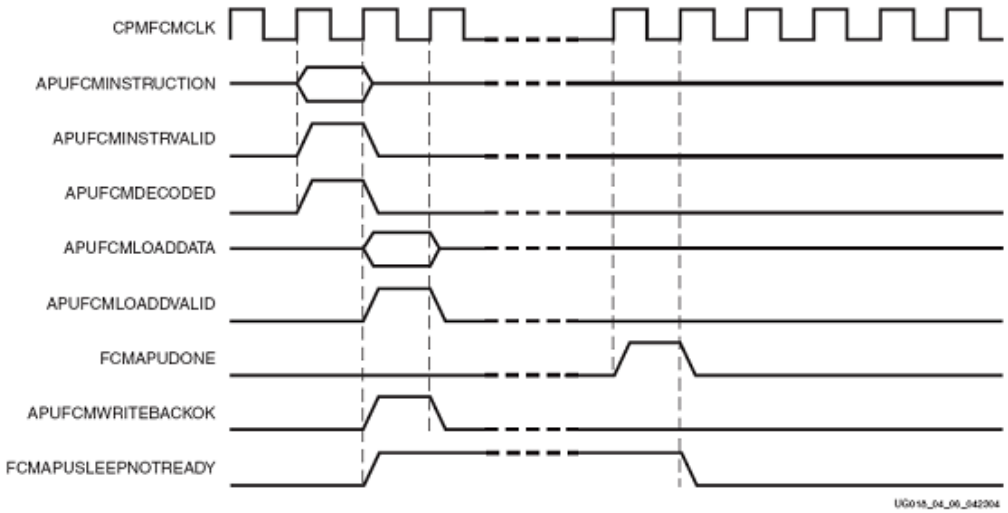


Figure 4.6: Decoded load instruction

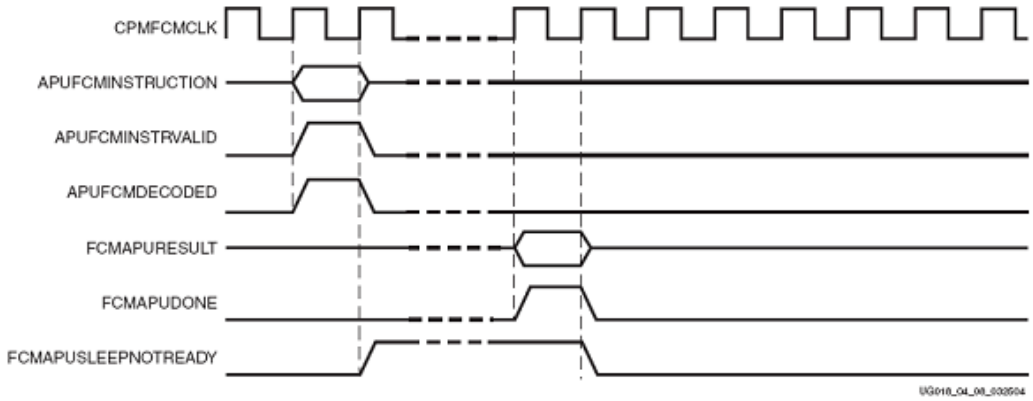


Figure 4.7: Decoded store instruction

4.2.3.2 Software

The CABAC decoder algorithm was adapted to run on the PowerPC processor. Several changed had to be made to just run the algorithm solely on the PowerPC processor instead of on an Intel processor. In the next phase the software was adapted to use the hardware accelerator instead of a part of the algorithm. To allow this, the custom instructions had to be added to the software. The APU had also to be initialized in the software and also some timers had to be set and started. Listing 4.2 shows shortened version of the software run on the PowerPC processor. The full software can be found in appendix B.

```
2 #include "xbasic_types.h"
  #include "xcache_l.h"
```

```
#include "xparameters.h"
#include "xpseudo_asm.h"
#include "xutil.h"
#include "stdio.h"
#include "xuartns550_l.h"
#include "xtmrctr.h"

#define lwfcmx(rn, base, adr)     __asm__ __volatile__(\
                            "lwfcmx " #rn ",%0,%1\n"\
                            : : "b" (base), "r" (adr)\
                          )

#define stwfcmx(rn, base, adr)     __asm__ __volatile__(\
                             "stwfcmx " #rn ",%0,%1\n"\
                             : : "b" (base), "r" (adr)\
                           )

volatile Xint32 __attribute__ ((aligned (32))) src[4] = {214,49,-3,20};
volatile Xint32 __attribute__ ((aligned (32))) dst[1000];

int main(void)
{
        XUartNs550_SetBaud(XPAR_RS232_UART_1_BASEADDR, XPAR_XUARTNS550_CLOCK_HZ, 9600);
        XUartNs550_mSetLineControlReg(XPAR_RS232_UART_1_BASEADDR, XUN_LCR_8_DATA_BITS);

        XTmrCtr InstancePtr;
        u16 DeviceId = 0;
        u32 time;
        if (XTmrCtr_Initialize(&InstancePtr, XPAR_XPS_TIMER_0_DEVICE_ID)==XST_SUCCESS)
        {
                printf("Timer initialized\n");
        }
        mtmsr(XREG_MSR_APU_AVAILABLE);

        int i=1;
        for (i=0; i<11; i++)
        {
                XTmrCtr_Start(&InstancePtr, 0);
                lwfcmx(0, &i, 0);
                        //apu_to_cabac accelerator running
                stwfcmx(0, dst, i*4);
                XTmrCtr_Stop(&InstancePtr, 0);
        }

        time = (XTmrCtr_GetValue(&InstancePtr, 0)+2) * 10;
        printf("time: %d (ns)\n",time);
        XTmrCtr_Reset(&InstancePtr, 0);
        print("Ending APU_TO_CABAC\r\n");

        return 0;
}
```

Listing 4.2: apu_to_cabac.c; Short software to run hardware accelerated CABAC decoding

The ELF file run on the PowerPC processor was generated by the Xilinx Platform Studio SDK. The ELF file could also be disassembled by the Xilinx Platform Studio SDK. The disassembly of the generated ELF file [28] is done with the command: powerpc-eabi-objdump -S apu_to_cabac.elf >> apu_to_cabac.dis. Listing 4.3 shows the part of the assembly that lets the processor communicate with the CABAC hardware accelerator.

```
lwfcmx(0, &i, 0); // load i into cabac-decoder
ffff0288:       39 3f 00 24     addi    r9,r31,36
ffff028c:       38 00 00 00     li      r0,0
ffff0290:       7c 09 00 8e     lwfcmx  0,r9,r0
        /*lwfcmx(1, src, 4);
        lwfcmx(2, src, 8);

        stwfcmx(2, dst, 8);
        stwfcmx(1, dst, 4);*/
        stwfcmx(0, dst, i*4); //store anwser from cabac-decoder to dst[i]
ffff0294:       80 1f 00 24     lwz     r0,36(r31)
ffff0298:       54 00 10 3a     rlwinm  r0,r0,2,0,29
ffff029c:       3d 20 00 00     lis     r9,0
```

```
15 ffff02a0:        39 29 bc e0      addi     r9,r9,-17184
   ffff02a4:        7c 09 01 8e      stwfcmx  0,r9,r0
```

Listing 4.3: Part of the disassembled ELF file. Calling the CABAC hardware accelerator and waiting for its result.

4.2.4 Stages in engineering process

4.2.4.1 Xilinx ML410 with Xilinx Virtex-4 FPGA

For our implementation of the CABAC decoder in hardware we used the ML410 evaluation board from Xilinx. This evaluation board includes the Xilinx Virtex 4 FPGA. The Virtex-4 FXT includes two PowerPC 440 processor blocks.

4.2.4.2 ModelSim

A part of the software for the CABAC-decoder we wanted to accelerate. This part includes the arithmetic coder and was isolated in the software. To accelerate this part of the software we used VHDL to describe the functionality into hardware. With the use of ModelSim we simulated the resulting hardware and could test the correctness of the functionality.

4.2.4.3 Xilinx ISE

With the hardware we made in VHDL we could synthesize it with Xilinx ISE and test the functionality of the standalone accelerator. We also tested the design synthesized for our specific FPGA and found timing, area and power reports. The values for these timing, energy, area results can be found in chapter 5. The synthesized architecture for the CABAC decoder can be found in figure 4.8.

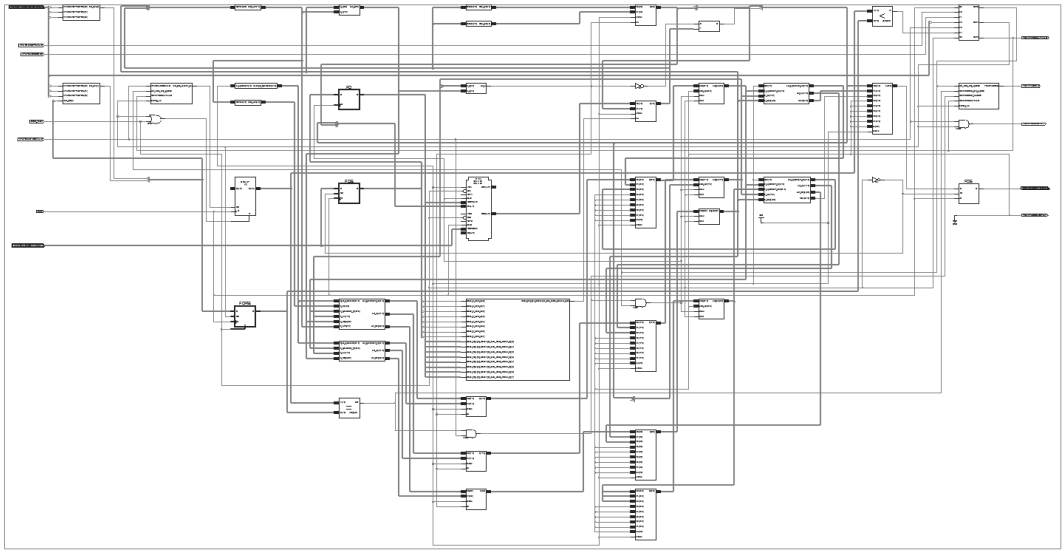


Figure 4.8: The synthesized architecture for the CABAC decoder

4.2.4.4 Xilinx Platform Studio

In the Xilinx Platform Studio we specified our desired hardware and software platform. This included the CABAC- accelerator unit we made in hardware. It also included one PowerPC 440 processor block, a serial interface and some memory organization for inputting and outputting the data. The total configuration of the CABAC accelerator with the PowerPC platform as implemented on the FPGA can be seen in figure 4.9.

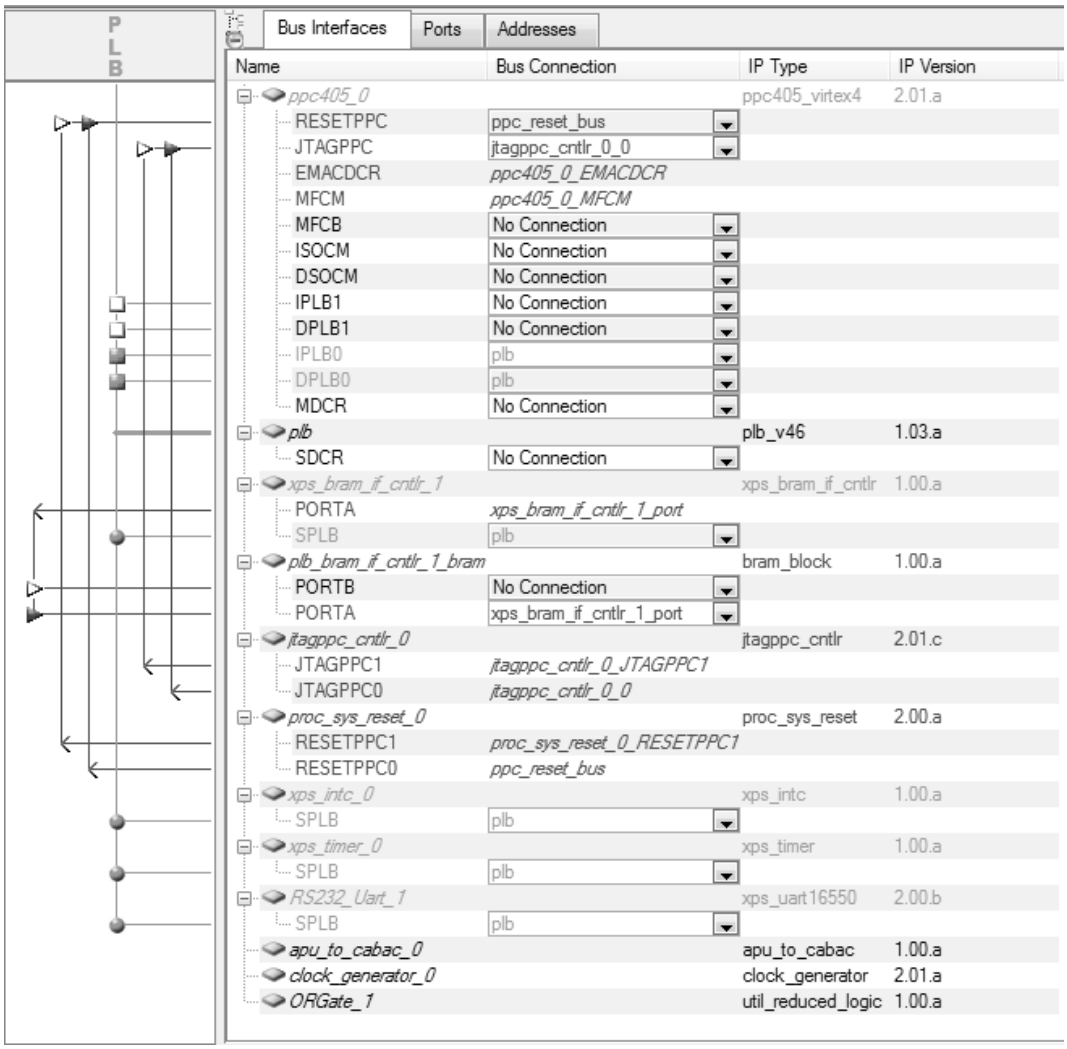


Figure 4.9: CABAC accelerator and PowerPC platform configuration

The PowerPC 440 processor block was synthesized with an addition APU interface. The hardware CABAC-accelerator was build as an Auxiliary Processing Unit. The PowerPC processor could speak directly with the CABAC-accelerator via its APU-interface. With the Xilinx Platform Studio SDK we wrote our software to run on the embedded PowerPC 440 processor block. The software included the function for addressing the CABAC-accelerator and reading the value it gave back. The software was compiled

using the Platform Studio SDK and with the Platform Studio it was loaded onto the ML410 evaluation board and run for several times. The total architecture of the CABAC accelerator with the PowerPC platform as implemented on the FPGA can be seen in figure 4.10.

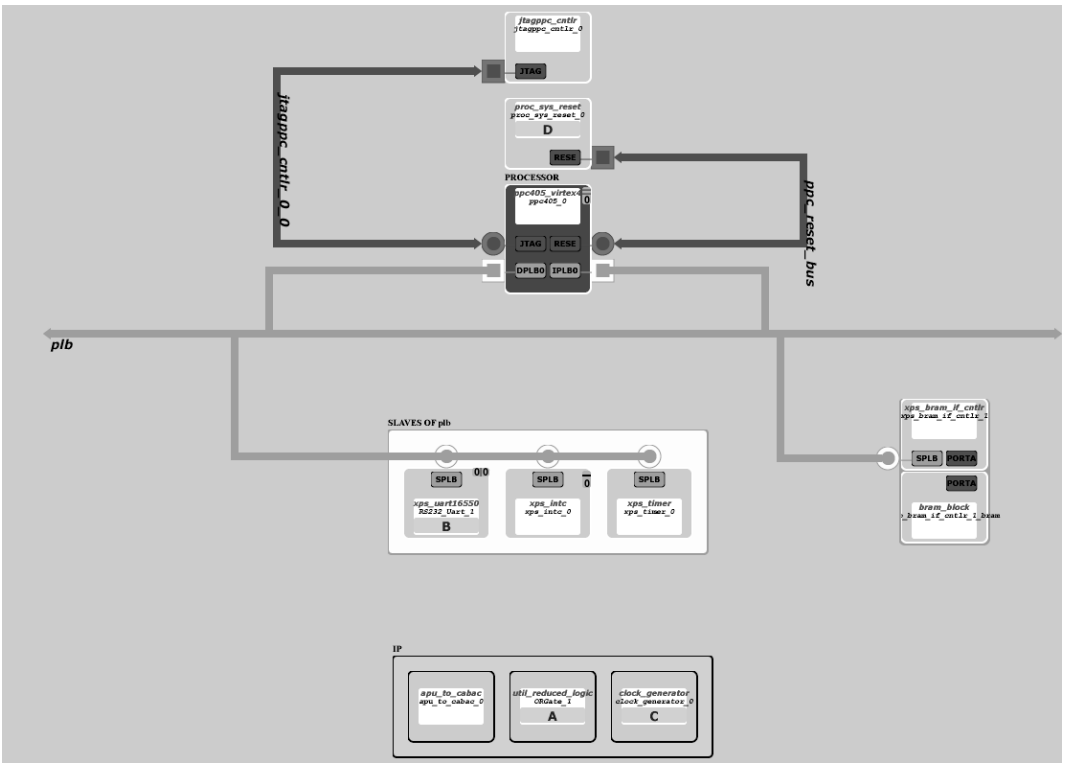


Figure 4.10: CABAC accelerator and PowerPC platform implemented on the FPGA

4.2.5 Final design and testing

The final design was made with the Xilinx Platform Studio. The main parts of the final design are the PowerPC processor, the APU and the CABAC decoder hardware accelerator. To be able to run the main parts on the platform, different subparts were added. The interface with the user was made with a RS232 UART. The data was read from a separate terminal which communicated with the platform using the RS232 UART. To clock the speed of the algorithm and to calculate the speed-up, a xps-timer block was used. This block accurately counted the frequency pulse and could be managed from the software. Also a clock generator was present to produce clock signals for the processor (300MHz) and for the hardware accelerator (25MHz). BRAM was utilized to store the software algorithm and the bootloader. The processor would load the bootloader and the bootloader would start the CABAC decoder software.

4.2.5.1 Software

The CABAC decoder software that was run on the platform was a loop of different CABAC decoding instructions. First run only on the processor and secondly run with the hardware accelerator. Both of the runs were timed and the difference in the timings would give us a speed-up.

4.3 Conclusion

In this chapter, we presented the work we have done to implement the CABAC decoder. After first giving an overall system description, we looked at the different parts in the system. We showed how these different parts work together. Also the different engineering stages and tests have been highlighted.

Simulation and implementation results of the CABAC decoder

5

5.1 Modelsim simulation and verification

In Modelsim the first version of the hardware accelerated CABAC decoder was simulated and verified. The CABAC decoder was existing merely of the functional unit with the interface of the Auxiliary Processing Unit (APU). The CABAC decoder model was tested with a testbench and previously defined input. Output from the model was verified with output from the software version of the CABAC decoder.

5.2 Results of related CABAC decoders

This section summarizes the results of some of the CABAC decoder simulations, synthesized hardwares and/or FPGA implementations of the authors found in the related work section of chapter 2.

[32] Measured in RTL simulations, CABAC accelerator at slice level; Conventional scheme Average 7.43 cycles/bin, proposed scheme average 3.93, speedup 1.81; Synthesis results: 0.18 um standard CMOS technology; Max frequency 225 MHz; Critical path 4.42 ns; Equivalent gate count: 81,162 gates; Context memory 662 Bytes; Data memory 11.52 Kbytes

[12] FPGA simulation of CABAC decoder at slice level; Xilinx Virtex-4 LX-200 FPGA; Clock speed 100 MHz; Area 346 slices (0.5%); Memory 2 Block-RAM; 0.99 bins/cycle (frame type I, QP=20)

[13] RTL simulation, CABAC accelerator at slice level; FPGA and ASIC synthesis/simulation for TOWER 0.18 um technology and Stratix II technology; TOWER 0.18 um; Clock rate 147 MHz; Throughput 147 Msamples; Logic: 23049 gates; Stratix II; Clock rate 159 MHz; Throughput 159 Msamples; Logic: 3730 ALUT

[18] RTL simulation and gate-level synthesis/simulation of CABAC decoder at Macro-block level; 0.18 um technology; Clock cycle 160 MHz; Logic: 26100 gates

[1] RTL simulation and synthesized for TSMC 0.13 um standard cell library; CABAC decoder at macroblock level; Logic: 138,226 gates (including context table); Clock cycle: 200 MHz; Average clock cycles: 1661 (frame type I macroblock); Throughput: 1 bit per 2-3 cycles

[2] FPGA simulation for Altera Stratix S25 (C5) and Altera Stratix S60 (C3); CABAC decoder at slice level; Clock speed: 70 MHz / 100 MHz; Logic arithmetic decoder: 1287 LEs / 590 ALMs

[35] RTL simulation and synthesized for 0.18 um CMOS cells library; CABAC decoder at slice level; Frequency 160 MHz; Critical path 6.2 ns; Logic: 30200 + 16200 gates (logic + register banks); Throughput 1 bin/cycle

[34] RTL simulation and synthesized for 0.18 um technology; CABAC decoder at Marcoblock level; Critical path 22ns; Maximum frequency 45 MHz; Logic: 42000 gates (excluding context RAM)

[5] RTL simulation and synthesized for 0.18 um CMOS technology; CABAC decoder at slice level; Equivalent gate count 35870 gates; Critical path delay 4.02 ns; Frequency 230 MHz; Estimated peak bit-rate 115 Mb/s

[4] RTL simulation and synthesized for 0.18 um CMOS technology; CABAC decoder at macroblock level; Critical path 4.5 ns; 0.41 bins/cycle

[31] RTL simulation and synthesized for TSMC 0.18 um CMOS technology; Frequency: 120 MHz; Logic: 83157 gates (including context RAM); Macroblock CABAC decoder; 463 cycles (I type macroblock with qp=36)

As can be seen the above results are hard or even impossible to compare with each other. Different parts of the CABAC decoding algorithm are used with different solutions how to speed up the decoding process. Other technologies are used to implement the CABAC decoder in. But the measurements taken are from the simulation of the synthesized results, they are theoretical values. As we can see different RAM sizes, different frequencies and different results, comparing is very hard.

5.3 Xilinx ISE synthesis and simulation

The hardware accelerated CABAC decoder model that was made, simulated and verified using Modelsim was next synthesized to the desired hardware platform. In our case the Virtex 4 FPGA from Xilinx. The Virtex 4 model was: xc4vfx60-11ff1152. The synthesis was done use Xilinx ISE 10.1.03.

A short summary of the results from our FPGA implementation of the CABAC decoder at slice level on a hardware / software co-design basis:

- ML-410 Embedded Development Platform
- Total logic cells: 56880, total slices: 25280, distributed RAM 395 kb, blockRAM: 4176 kb

- 1.2V core voltage, 90nm Copper CMOS technology
- Dual POWERPC 405 processor run at 300 MHz (single core used)
- CABAC decoding 1 bin/cycle

Table 5.1: Timing Summary

Timing Summary (Speed Grade: -11):	
Minimum period:	39.122 ns
Maximum Frequency:	25.561 MHz
Minimum input arrival time before clock:	3.357 ns
Maximum output required time after clock:	5.819 ns
Maximum combinational path delay:	5.419 ns

Table 5.2: Device Utilization Summary

Device Utilization Summary:		
Number of BUFGs	2 out of 32	6%
Number of External IOBs	88 out of 576	15%
Number of LOCed IOBs	0 out of 88	0%
Number of RAMB16s	1 out of 232	1%
Number of Slices	336 out of 25280	1%
Number of SLICEMs	16 out of 12640	1%

Table 5.3: XPower Analysis Report

XPower Analysis Report:		
Power summary	I(mA)	P(mW)
Total estimated power consumption		893
Total Vccint 1.20V	279	335
Total Vccaux 2.50V	219	547
Total Vcco25 2.50V	4	11
Clocks	26	31
Inputs	0	0
Logic	9	10
Outputs Vcco25	3	8
Signals	11	13
Quiescent Vccint 1.20V	233	280
Quiescent Vccaux 2.50V	219	547
Quiescent Vcco25 2.50V	2	4

Results for the timing of the synthesis of the hardware accelerated CABAC decoder are found in table 5.1. This is a measure of how fast the implemented architecture can be run. In table 5.2 the summary of the device utilization is given. As can be seen the number of external input/output blocks (IOBs) is relative high, but extra space is still available for other applications. Table 5.3 summarizes the current and power needed to run the hardware accelerated CABAC decoder. This would be more interesting if we had implemented our design specifically for battery powered embedded systems. But we still find this analysis useful, it gives a good estimate of how many heat is generated and must be dissipated using only a small heatsink.

5.4 Xilinx Platform Studio

The whole CABAC decoder implementation was tested and simulated with Xilinx Platform Studio. In table 5.4 the speedup results can be seen. The speedup is calculated out of the time when running the software solely on the processor and the time when running the software with the special build hardware accelerator.

Table 5.4: Speedup results

Speedup Summary (five test runs):	
Test run 1	Speedup: 2.85629
Test run 2	Speedup: 2.83489
Test run 3	Speedup: 2.83489
Test run 4	Speedup: 2.83482
Test run 5	Speedup: 2.83484

As can be seen in section 5.2 the work done by the related work authors is very hard to compare. All described solutions adopt other technologies, other clock rates and other ways to speed-up the selected part of the CABAC entropy decoder. All of the solutions described in section 5.2 where only synthesized and not implemented into real FPGA or ASIC fabric. All of the measurement results were therefore only theoretical values. This means that these speeds and speedups are theoretical possible if implemented on the chosen hardware.
Our solution of the CABAC entropy decoding accelerator was actually implemented into real FPGA hardware. The measurements were taken with an internal timer, inside the FPGA. Therefor the results are real world values and not theoretical values.

For building the CABAC entropy decoder accelerator a methodology was chosen on beforehand and used throughout the project. The prime objective was to have a working hardware / software co-design CABAC entropy decoder. We have chosen to work the a bottom-up approach. First only a small part of the CABAC decoding algorithm was implemented into hardware. This little part was tested stand-alone and when the workings were correct, it was combined with the software in the FPGA. This hardware / software co-design was tested for correct workings, correct interfacing between the hardware and the software and the performance was

measured. With the good results, the little hardware CABAC decoding core was expanded step-by-step. Every step checking the consistency of the whole hardware / software co-design and validating the results. The process of the methodology was limited by the time available for the MSc thesis project. Boundaries we kept a close eye on during the project were the measured speed-up and the ratio of the hardware and software clockrate. A negative speedup would not be a desired result and a great difference in clock rate ratio would mean we had to choose another approach.

5.5 Conclusion

In this chapter, we presented our results we have gathered during the simulation, verification and synthesis of the hardware accelerated CABAC decoder. Modelsim, Xilinx ISE and Xilinx Platform Studio supplied us with the different results for timing, area, power and speedup. The different software tools packages were used during different parts of the engineering process. The final implementation of the hardware accelerated CABAC decoder was tested using the Xilinx Platform Studio. Although the hardware ran 12 times slower than the processor, the speedup we got for our final implementation was 2.83.

6 Conclusion

6.1 Summary

In chapter 1 we gave a general introduction on the CABAC decoding algorithm. We discussed the research scope and presented the problem statement discussed in this thesis. Also an overview of the thesis was given in chapter 1.

In chapter 2 we introduced the background of the CABAC encoding and decoding process. Firstly, we gave the terminology and structure used in the H.264 coding standard. Secondly, we presented the entropy encoding process. This process consists of the following stages: binarization, context model selection, arithmetic encoding and probability update. Of every stage we presented the working algorithms and how they are connected to each other. Furthermore we also referred to related research work done by other groups.

In chapter 3 we gave an overview of the CABAC decoding scheme as we were going to implement it. The different stages in the encoding process also play an important role in the decoding process. The stages are presented with more emphasize on the details of the decoding algorithm. Firstly the context model selection is explained in more detail. Also the coding engine is explained in every detail. First the inner workings of the coding engine in software are explained and how they can be made in hardware. The last stage of the de-binarization is explained and presented in a more practical view. Secondly, we presented an analysis of the de-binarization stage of the CABAC decoding engine. As we are concerned with speed in our hardware and software co-design implementation of the encoding engine, exploring the de-binarization stage could be profitable. At last we motivate our choices made to implement the arithmetic decoding engine. In future research we also want to implement the de-binarization stage.

In chapter 4 we present a detailed description of the implementation of the CABAC decoder on the chosen platform. The platform is the Xilinx ML410 with the Xilinx Virtex-4 FPGA. This FPGA includes a hardcore PowerPC 440 processor block which is used to run the CABAC decoding software on. Parts of the CABAC decoding software are accelerated using custom hardware on the FPGA fabric. The PowerPC block and the hardware accelerator on the FPGA communicate with each other through the Auxiliary Processing Unit (APU) interface. The accelerator was written, simulated and verified using VHDL in ModelSim. The accelerator was synthesized, simulated and tested for our specific FPGA using Xilinx ISE. The final step was to integrate the hardware accelerator, the APU interface, the software run on the PowerPC into one system using Xilinx Platform Studio. The whole implementation was also simulated

and tested using the Xilinx Platform Studio tools.

In chapter 5 we discussed simulation and verification results of the implementation we presented in chapter 4. The results were in terms of speedup of the hardware version versus the software only version. And in terms of timing, device utilization and power of the implemented hardware version.

6.2 Main contributions

In this section, we list the most important contributions of our research.

- We have presented a stand-alone hardware accelerator for the CABAC arithmetic decoding engine in VHDL.
- We have presented a CABAC arithmetic decoding engine as a hardware and software co-design implemented on the Virtex-4 using the PowerPCs Auxiliary Processing Unit (APU) interface. Measurements were not theoretical, but actual, real-life values.
- We have introduced a basis for the research on the H.264 CABAC decoding engine as a hardware / software co-design. Further research can be done on this platform to better understand the algorithms and to get even better speedups.

6.3 Future work

In this section, we present directions for future research and development. The directions are originated from the idea to implement more functionality of the CABAC decoding engine algorithm from software into hardware and to gain more speedup, less area and less energy consumption.

- **Implement the CABAC decoding engine hardware accelerator on the MOLEN prototype.**
  The MOLEN prototype is extensively used as a platform to dynamically accelerate computation intensive algorithms using custom hardware accelerators. On the MOLEN prototype different accelerator are dynamically used only when they are required. Interesting would be how the algorithm behaves when first the bitstream is CABAC decoded, and then the rest of the H.264 decoding takes place with a different hardware accelerator on the MOLEN prototype.
- **Investigate the implementation of the CABAC decoding engine on the SarcSim platform.**
  The SarcSim is a simulation platform based on the IBM Cell processor. The Cell processor is a multicore processor and can be used to accelerate multimedia applications. The SarcSim group is currently trying to optimize and accelerate the H.264 decoding standard for the Cell processor. The CABAC decoding engine

is a very though one to optimize because of its sequential nature. More research has to be done to figure what the best implementation would be for the SarcSim platform.

- **Analysis the de-binarization stage of the CABAC decoder**
  The de-binarization stage of the CABAC entropy decoder holds a large potential to further speed-ups. At present every branch of the de-binarization trees are searched step by step, decoding one bin at a time. The believe is this can be more intelligently be processed in a parallel fashion.

- **Multicore parallelism on the slice level**
  The CABAC entropy decoder has a very sequential algorithm to decode the incoming bitstream. But because every slice is independently decoded, this can be done in a massive parallel way. Multicore parallelism can be used in our advantage to speedup the decoding of H.264 video streams.

Bibliography

[1] Jian-Wen Chen, Cheng-Ru Chang, and Youn-Long Lin, *A hardware accelerator for context-based adaptive binary arithmetic decoding in H.264/AVC*, ISCAS (5), IEEE, 2005, pp. 4525–4528.

[2] Hendrik Eeckhaut, Mark Christiaens, Dirk Stroobandt, and Vincent Nollet, *Optimizing the critical loop in the h.264/avc cabac decoder*, Proceedings of International Conference on Field Programmable Technology (Bangkok), IEEE, 12 2006, pp. 113–118.

[3] M. Jeanne, C. Guillemot, T. Guionnet, and F. Pauchet, *Error-resilient decoding of context-based adaptive binary arithmetic codes*, Signal Image and Video Processing **1** (2007), no. 1, 77–87.

[4] Chung-Hyo Kim and In-Cheol Park, *High speed decoding of context-based adaptive binary arithmetic codes using most probable symbol prediction*, ISCAS, IEEE, 2006.

[5] Lingfeng Li, Yang Song, Shen Li, Takeshi Ikenaga, and Satoshi Goto, *A hardware architecture of CABAC encoding and decoding with dynamic pipeline for H.264/AVC*, - (2008), – (En).

[6] D. Marpe, H. Schwarz, and T. Wiegand, *Context-based adaptive binary arithmetic coding in the h.264/avc video compression standard*, Circuits and Systems for Video Technology, IEEE Transactions on **13** (2003), no. 7, 620–636.

[7] M.E.Castro, R.R.Osorio, and J.D.Bruguera, *Optimizing cabac for vliw architectures*, - (Barcelona (Spain)), 2006.

[8] Harn Hua Ng, *Xilinx: Accelerated system performance with the apu controller and xtremedsp slices*, v1.1.1 ed., 2009.

[9] Jari Nikara, Stamatis Vassiliadis, Jarmo Takala, and Petri Liuha, *Multiple-symbol parallel decoding for variable length codes*, IEEE Trans. VLSI Syst **12** (2004), no. 7, 676–685.

[10] R. R. Osorio and J. D. Bruguera, *High-throughput architecture for H.264/AVC CABAC compression system*, IEEE Trans. Circuits and Systems for Video Technology **16** (2006), no. 11, 1376–1384.

[11] Roberto R. Osorio and Javier D. Bruguera, *Arithmetic coding architecture for H.264/AVC CABAC compression system*, DSD, IEEE Computer Society, 2004, pp. 62–69.

[12] ———, *An FPGA architecture for CABAC decoding in manycore systems*, ASAP, IEEE Computer Society, 2008, pp. 293–298.

[13] G. Pastuszak, *A high-performance architecture of the double-mode binary coder for H.264.AVC*, IEEE Trans. Circuits and Systems for Video Technology **18** (2008), no. 7, 949–960.

[14] Iain E. Richardson, *H.264 and mpeg-4 video compression: Video coding for next generation multimedia*, 1 ed., Wiley, August 2003.

[15] Sergio Saponara, Carolina Blanch, Kristof Denolf, and Jan Bormans, *The jvt advanced video coding standard: Complexity and performance analysis on a tool-by-tool basis*, unknown journal name (2003), –.

[16] H. Schwarz, D. Marpe, and T. Wiegand, *Cabac and slices*, JVT document JVT-D020 (2002), –.

[17] Glenn Steiner, *Xilinx: Code acceleration with an apu coprocessor: a case study of an lpm algorithm*, Xilinx, 2008.

[18] Liang-Hao Wang, Zheng Zhu, Kai Luo, Bingbo Li, and Ming Zhang, *System-on-chip design for a statistical decoder*, ASIC, 2007. ASICON '07. 7th International Conference on (2007), 966–969.

[19] Thomas Wiegand, Gary J. Sullivan, Gisle Bjntegaard, and Ajay Luthra, *Overview of the H.264/AVC video coding standard*, IEEE Trans. Circuits Syst. Video Techn **13** (2003), no. 7, 560–576.

[20] I. Witten, R. Neal, and J. Clearly, *Arithmetic coding for data compression*, Communication of the ACM (1987), –.

[21] Xilinx, *Xilinx: Fcb to fsl bridge (v1.00a)*, v 1.00a ed., 2005.

[22] ———, *Xilinx: Powerpc 405 apu controller*, v2.1 ed., 2005.

[23] ———, *Xilinx: Powerpc instruction set extention guide, isa support for the powerpc apu controller in virtex-4*, Xilinx, rev 2.0 ed., 2005.

[24] ———, *Xilinx: Ml410 embedded development platform, user guide*, v 1.7 ed., 2007.

[25] ———, *Xilinx: Ppc405 virtex-4 (wrapper) (v2.01a)*, 2007.

[26] ———, *Xilinx: Powerpc 405 processor block reference guide, embedded development kit*, v 2.3 ed., 2008.

[27] ———, *Xilinx: Xps timer/counter (v1.00a)*, v1.00a ed., 2008.

[28] ———, *Xilinx: Edk concepts, tools, and techniques; a hands-on guide to effective embedded system design*, v 10.1 ed., 2009.

[29] ———, *Xilinx: Embedded system tools reference manual, embedded development kit*, v 10.1 sp 3 ed., 2009.

[30] ———, *Xilinx: Synthesis and simulation design guide*, v 8.2i ed., 2009.

[31] Yao-Chang Yang, Chien-Chang Lin, Hsui-Cheng Chang, Ching-Lung Su, and Jiun-In Guo, *A high throughput VLSI architecture design for H.264 context-based adaptive binary arithmetic decoding with look ahead parsing*, ICME, IEEE, 2006, pp. 357–360.

[32] Y. S. Yi and I. C. Park, *High-speed H.264/AVC CABAC decoding*, IEEE Trans. Circuits and Systems for Video Technology **17** (2007), no. 4, 490–494.

[33] Wei Yu and Yun He, *A high performance cabac decoding architecture*, Consumer Electronics, IEEE Transactions on **51** (2005), no. 4, 1352–1359.

[34] Peng Zhang, Wen Gao, Don Xie, and Di Wu, *High-performance cabac engine for h.264/avc high definition real-time decoding*, Consumer Electronics, 2007. ICCE 2007. Digest of Technical Papers. International Conference on (2007), 1–2.

[35] Junhao Zheng, David Wu, Don Xie, and Wen Gao, *A novel pipeline design for H.264 CABAC decoding*, Advances in Multimedia Information Processing - PCM 2007, 8th Pacific Rim Conference on Multimedia, Hong Kong, China, December 11-14, 2007, Proceedings (Horace Ho-Shing Ip, Oscar C. Au, Howard Leung, Ming-Ting Sun, Wei-Ying Ma, and Shi-Min Hu, eds.), Lecture Notes in Computer Science, vol. 4810, Springer, 2007, pp. 559–568.

VHDL

A

```
library ieee;
--library unisim;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
--use unisim.vcomponents.RAMB16;

entity apu_to_cabac is
    port(-- inputs: misc
         clock                  : in std_logic;
         reset                  : in std_logic;
         -- inputs: From APU to FCM
         APUFCMINSTRUCTION      : in std_logic_vector (0 to 31);
         APUFCMINSTRVALID       : in std_logic;
         APUFCMRADATA           : in std_logic_vector (0 to 31);
         APUFCMRBDATA           : in std_logic_vector (0 to 31);
         APUFCMOPERANDVALID     : in std_logic;
         APUFCMFLUSH            : in std_logic;
         APUFCMWRITEBACKOK      : in std_logic;
         APUFCMDECUDI           : in std_logic_vector (0 to 2);
         APUFCMDECUDIVALID      : in std_logic;
         APUFCMDECODED          : in std_logic;
         -- not used
         APUFCMLOADDATA         : in std_logic_vector (0 to 31);
         APUFCMLOADDVALID       : in std_logic;
         APUFCMLOADBYTEEN : in std_logic_vector(0 to 3); --
         APUFCMLOADBYTEADDR     : in std_logic_vector (0 to 3);
         APUFCMENDIAN           : in std_logic;
         APUFCMXERCA : in std_logic; --
         APUFCMDECFPUOP         : in std_logic;
         APUFCMDECLOAD          : in std_logic;
         APUFCMDECSTORE         : in std_logic;
         APUFCMDECLDSTXFERSIZE  : in std_logic_vector (0 to 2);
         APUFCMDECNONAUTON      : in std_logic;
         APUFCMNEXTINSTRREADY   : in std_logic;
         APUFCMMSRFE0           : in std_logic;
         APUFCMMSRFE1           : in std_logic;

         -- for timing specifications of APU/FCM signals, see APU documentation
         -- outputs: From FCM to APU
         --FCMAPURESULT          : out std_logic_vector(0 to 31);
         --FCMAPURESULTVALID     : out std_logic; -- nrar
         --FCMAPUDONE            : out std_logic; -- nrar
         --FCMAPUSLEEPNOTREADY   : out std_logic; -- nrar
         -- not useful
         --FCMAPUCR              : out std_logic_vector(0 to 3);
         --FCMAPUEXCEPTION       : out std_logic;
         --FCMAPUSTOREDATA       : out std_logic_vector(0 to 31);
         --FCMAPUCONFIRMINSTR    : out std_logic;
         --FCMAPUFPSCRFEX        : out std_logic



        FCMAPUINSTRACK : out std_logic;
      FCMAPURESULT : out std_logic_vector(0 to 31);
      FCMAPUDONE :   out std_logic;
                FCMAPUSLEEPNOTREADY : out std_logic;
                FCMAPUDECODEBUSY : out std_logic;
                FCMAPUDCDGPRWRITE : out std_logic;
                FCMAPUDCDRAEN : out std_logic;
                FCMAPUDCDRBEN : out std_logic;
                FCMAPUDCDPRIVOP : out std_logic;
                FCMAPUDCDFORCEALIGN : out std_logic;
                FCMAPUDCDXEROVEN : out std_logic;
                FCMAPUDCDXERCAEN : out std_logic;
                FCMAPUDCDCREN : out std_logic;
                FCMAPUEXECRFIELD : out std_logic_vector(0 to 2);
                FCMAPUDCDLOAD : out std_logic;
                FCMAPUDCDSTORE : out std_logic;
```

```
                FCMAPUDCDUPDATE : out std_logic;
                FCMAPUDCDLDSTBYTE : out std_logic;
                FCMAPUDCDLDSTHW : out std_logic;
                FCMAPUDCDLDSTWD : out std_logic;
                FCMAPUDCDLDSTDW : out std_logic;
                FCMAPUDCDLDSTQW : out std_logic;
                FCMAPUDCDTRAPLE : out std_logic;
                FCMAPUDCDTRAPBE : out std_logic;
                FCMAPUDCDFORCEBESTEERING : out std_logic;
                FCMAPUDCDFPUOP : out std_logic;
                FCMAPUEXEBLOCKINGMCO : out std_logic;
                FCMAPUEXENONBLOCKINGMCO : out std_logic;
                FCMAPULOADWAIT : out std_logic;
                FCMAPURESULTVALID : out std_logic;
                FCMAPUXEROV : out std_logic;
                FCMAPUXERCA : out std_logic;
                FCMAPUCR : out std_logic_vector(0 to 3);
                FCMAPUEXCEPTION : out std_logic);
end entity apu_to_cabac;

architecture apu_to_cabac_arch of apu_to_cabac is

    -- type declarations
    type states is (STATE_IDLE, STATE_CABAC, STATE_CABAC_2);

    -- signal declarations
    signal state      : states; --state machine state
    signal next_state : states; --nrar

    signal data_a   : std_logic_vector(0 to 31);
    signal data_b   : std_logic_vector(0 to 31);


    begin

        -- logic
        --FCMAPURESULT <= (others=>'0');


        --state machine next state/combinational logic
        process(APUFCMINSTRVALID, APUFCMDECUDI, APUFCMOPERANDVALID,
            state, APUFCMRADATA, APUFCMRBDATA, APUFCMWRITEBACKOK)

            begin
                --some defaults
                next_state <= state;
                FCMAPURESULT <= (others=>'0');
                FCMAPUSLEEPNOTREADY <= '0';
                FCMAPUDONE <= '0';

                --
                            case(state) is
                            when STATE_IDLE =>


          if(APUFCMINSTRVALID = '1') then
                                --if (APUFCMDECUDI(0 to 2)="000") then
                                    next_state <= STATE_CABAC;
                                    --else
                                    --next_state <= STATE_IDLE;
                                --end if;

                                    else
                                    next_state <= STATE_IDLE;
                               end if;


    when STATE_CABAC =>
                               FCMAPUSLEEPNOTREADY <= '1';
                               if((APUFCMWRITEBACKOK = '1') and (APUFCMOPERANDVALID = '1')) then
                                       next_state <= STATE_CABAC_2;
                                       data_a <= APUFCMRADATA;
                                       data_b <= APUFCMRBDATA;
                               end     if;

        when STATE_CABAC_2 =>
            FCMAPUSLEEPNOTREADY <= '1';
             next_state <= STATE_IDLE;
                                       FCMAPUDONE <= '1';
                                       FCMAPURESULTVALID <= '1';
                                       FCMAPURESULT <= data_b;

    when others =>
                               next_state <= STATE_IDLE;
                end case;
        end process;
```

```
158         --state machine sequential elements
            process(clock) is
                begin
                    if (clock'event and clock = '1') then
                        if (reset = '1')then
163                         state <= STATE_IDLE;
                            else
                            state <= next_state;
                        end if;
                    end if;
168             end process;




173


            end architecture apu_to_cabac_arch;
```

Listing A.1: apu_to_cabac.vhdl

```
    library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
  3 use IEEE.STD_LOGIC_ARITH.ALL;
    use IEEE.STD_LOGIC_UNSIGNED.ALL;
    use ieee.numeric_std.all;

    entity bshift is
  8     generic (width: natural:=32);
            port (x: in std_logic_vector (width-1 downto 0);
                  s: in std_logic_vector (3 downto 0);
              z: out std_logic_vector (width-1 downto 0));
    end entity bshift;
 13
    architecture structural of bshift is

    component mux_2 is
            port (in0, in1, sel: in std_logic;
 18         z: out std_logic);
    end component mux_2;

    type matrix is array (0 to 4) of std_logic_vector(width-1 downto 0);

 23 signal p: matrix;

    begin
       p(0) <= x;

 28    mux_rows: for i in 0 to 3 generate --4
         mux_cols: for j in 0 to width-1 generate
         mux_col1: if(j - 2**i < 0) generate
             mux: mux_2 port map (in0 => p(i)(j), in1=> '0', sel => s(i), z => p(i+1)(j));
         end generate;
 33      mux_col2: if(j - 2**i >= 0) generate
            mux: mux_2 port map (in0 => p(i)(j), in1=> p(i)(j - 2**i), sel => s(i), z => p(i+1)(j));
         end generate;
       end generate;
     end generate;
 38
     z <= p(4);--p(5);

    end architecture structural;
```

Listing A.2: bshift.vhdl

```
    -- register 8 bits bytestream_ptr

    library IEEE;
  4 use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.STD_LOGIC_ARITH.ALL;
    use IEEE.STD_LOGIC_UNSIGNED.ALL;
    use ieee.numeric_std.all;

  9 entity bytestream_ptr_register is
      Port (
            clock : in std_logic;
            reset : in std_logic;
            data_set: in std_logic_vector(31 downto 0);
 14         data: in std_logic_vector(31 downto 0);
            output: out std_logic_vector(31 downto 0));
    end entity bytestream_ptr_register;
```

```
architecture behavioural of bytestream_ptr_register is

begin
    process(clock, reset)
        begin

      if (reset = '1') then
        output <= data_set;
      elsif (clock'event and clock = '1') then
        output <= data;
      end if;

end process;

end behavioural;
```

Listing A.3: bytestream_ptr_register.vhdl

```
--cabac
-- includes get_cabac, cabac_bypass, get_cabac_terminate

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity cabac is
    Port ( in_mode : in std_logic_vector(31 downto 0);
           in_state_idx : in std_logic_vector(31 downto 0);
           clock : in std_logic;
           reset : in std_logic;

           out_result : out std_logic_vector(31 downto 0)
           );
end entity cabac;


architecture behavioural of cabac is

component get_cabac is
    port(
        state_idx: in std_logic_vector(8 downto 0); --9 bits state_idx?
        in_low: in std_logic_vector(17 downto 0);
        in_range: in std_logic_vector(8 downto 0);
        in_bytestream_ptr: in std_logic_vector(31 downto 0);
        in_data: in std_logic_vector(7 downto 0);
        clock: in std_logic;
        reset: in std_logic;

        out_low: out std_logic_vector(17 downto 0);
        out_range : out std_logic_vector(8 downto 0);
        out_bytestream_ptr : out std_logic_vector(31 downto 0);
        out_result : out std_logic_vector(31 downto 0)

        );
end component get_cabac;

component cabac_bypass is
    Port ( in_low : in std_logic_vector(17 downto 0);
           in_bytestream_ptr: in std_logic_vector(31 downto 0);
           in_range : in std_logic_vector(8 downto 0);

           data : in std_logic_vector(7 downto 0);

           out_low: out std_logic_vector(17 downto 0);
           out_bytestream_ptr: out std_logic_vector(31 downto 0);
           result : out std_logic_vector(31 downto 0)
           );
end component cabac_bypass;

component get_cabac_terminate is
    Port ( in_low : in std_logic_vector(17 downto 0);
           in_bytestream_ptr: in std_logic_vector(31 downto 0);
           in_range : in std_logic_vector(8 downto 0);

           in_bytestream_start : in std_logic_vector(31 downto 0);

           data : in std_logic_vector(7 downto 0);

           out_low: out std_logic_vector(17 downto 0);
           out_range: out std_logic_vector(8 downto 0);
           out_bytestream_ptr: out std_logic_vector(31 downto 0);
           result : out std_logic_vector(31 downto 0)
```

```
            );
end component get_cabac_terminate;

component bytestream_ptr_register is
    Port (
          clock : in std_logic;
          reset : in std_logic;

          data_set: in std_logic_vector(31 downto 0);
          data: in std_logic_vector(31 downto 0);
          output: out std_logic_vector(31 downto 0));
end component bytestream_ptr_register;

component low_register is
    Port (
          clock : in std_logic;
          reset : in std_logic;

          data_set: in std_logic_vector(17 downto 0);
          data: in std_logic_vector(17 downto 0);
          output: out std_logic_vector(17 downto 0));
end component low_register;

component range_register is
    Port (
          clock : in std_logic;
          reset : in std_logic;

          data: in std_logic_vector(8 downto 0);
          output: out std_logic_vector(8 downto 0));
end component range_register;

component mux8_x is
    generic (width: natural:=32);
    port (
    in0, in1, in2, in3, in4, in5, in6, in7 : in std_logic_vector(width-1 downto 0);
         enable: in std_logic;
         sel: in std_logic_vector(2 downto 0);
         z: out std_logic_vector(width-1 downto 0));
end component mux8_x;

signal in_low, out_low, out_low_mux1, out_low_mux2, out_low_mux3 : std_logic_vector(17 downto 0);

signal in_range, out_range, out_range_mux1, out_range_mux3
    : std_logic_vector(8 downto 0);
signal in_bytestream_ptr, out_bytestream_ptr, in_bytestream_start,
    set_bytestream_start, out_bytestream_ptr_mux1, out_bytestream_ptr_mux2, out_bytestream_ptr_mux3
     , out_bytestream_start: std_logic_vector(31 downto 0);

signal in_data : std_logic_vector(7 downto 0) := x"FF";

signal out_result_mux1, out_result_mux2, out_result_mux3 : std_logic_vector(31 downto 0);

begin

--registers
low_reg: low_register port map(
          clock => clock,
          reset => reset,
          data_set => "00000000000000010",--first 12 bits of the bytestream +2
          data => out_low,
          output => in_low
);

range_reg: range_register port map(
          clock => clock,
          reset => reset,
          data => out_range,
          output => in_range
);

bytestream_ptr_reg: bytestream_ptr_register port map(
          clock => clock,
          reset => reset,
          data_set => x"00000000",
          data => out_bytestream_ptr,
          output => in_bytestream_ptr
);

bytestream_start_reg: bytestream_ptr_register port map(
          clock => clock,
          reset => reset,
          data_set => set_bytestream_start,
          data => out_bytestream_start,
          output => in_bytestream_start
);
```

```
    --combinational logic
    get_cabac1: get_cabac port map(

158         state_idx => in_state_idx(8 downto 0),
            in_low => in_low,
            in_range => in_range,
            in_bytestream_ptr => in_bytestream_ptr,
            in_data => in_data,
163         clock => clock,
            reset => reset,
            out_low => out_low_mux1,
            out_range => out_range_mux1,
            out_bytestream_ptr => out_bytestream_ptr_mux1,
168         out_result => out_result_mux1

            );

    cabac_bypass1: cabac_bypass port map(
173         in_low => in_low,
            in_bytestream_ptr => in_bytestream_ptr,
            in_range => in_range,
            data => in_data,
            out_low => out_low_mux2,
178         out_bytestream_ptr => out_bytestream_ptr_mux2,
            result => out_result_mux2
            );

    get_cabac_terminate1: get_cabac_terminate port map(
183         in_low => in_low,
            in_bytestream_ptr => in_bytestream_ptr,
            in_range => in_range,
            in_bytestream_start => in_bytestream_start,
            data => in_data,
188         out_low => out_low_mux3,
            out_range => out_range_mux3,
            out_bytestream_ptr => out_bytestream_ptr_mux3,
            result => out_result_mux3
            );
193

    --muxes

    low_mux1: mux8_x generic map (18) port map(
198         in0 => out_low_mux1,
            in1 => out_low_mux2,
            in2 => out_low_mux3,
            in3 => "000000000000000000",
            in4 => "000000000000000000",
203         in5 => "000000000000000000",
            in6 => "000000000000000000",
            in7 => "000000000000000000",
                enable => '1',
                sel => in_mode(2 downto 0),
208             z => out_low
            );

    range_mux1: mux8_x generic map (9) port map(
            in0 => out_range_mux1,
213         in1 => in_range,
            in2 => out_range_mux3,
            in3 => "000000000",
            in4 => "000000000",
            in5 => "000000000",
218         in6 => "000000000",
            in7 => "000000000",
                enable => '1',
                sel => in_mode(2 downto 0),
                z => out_range
223         );

    bytestream_ptr_mux1: mux8_x generic map (32) port map(
            in0 => out_bytestream_ptr_mux1,
            in1 => out_bytestream_ptr_mux2,
228         in2 => out_bytestream_ptr_mux3,
            in3 => x"00000000",
            in4 => x"00000000",
            in5 => x"00000000",
            in6 => x"00000000",
233         in7 => x"00000000",
                enable => '1',
                sel => in_mode(2 downto 0),
                z => out_bytestream_ptr
            );
238
    result_mux1: mux8_x generic map (32) port map(
            in0 => out_result_mux1,
```

```
        in1 => out_result_mux2,
        in2 => out_result_mux3,
        in3 => x"00000000",
        in4 => x"00000000",
        in5 => x"00000000",
        in6 => x"00000000",
        in7 => x"00000000",
             enable => '1',
             sel => in_mode(2 downto 0),
             z => out_result
        );

bytestream_start_mux1: mux8_x generic map (32) port map(
        in0 => in_bytestream_start,
        in1 => in_bytestream_start,
        in2 => in_bytestream_start,
        in3 => set_bytestream_start,
        in4 => x"00000000",
        in5 => x"00000000",
        in6 => x"00000000",
        in7 => x"00000000",
             enable => '1',
             sel => in_mode(2 downto 0),
             z => out_bytestream_start
        );

end behavioural;
```

Listing A.4: cabac.vhdl

```
--cabac_bypass

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity cabac_bypass is
    Port ( in_low : in std_logic_vector(17 downto 0);
           in_bytestream_ptr: in std_logic_vector(31 downto 0);
           in_range : in std_logic_vector(8 downto 0);

           data : in std_logic_vector(7 downto 0);

           out_low: out std_logic_vector(17 downto 0);
           out_bytestream_ptr: out std_logic_vector(31 downto 0);
           result : out std_logic_vector(31 downto 0)
           );
end entity cabac_bypass;

architecture behavioural of cabac_bypass is

    component mux_x is
    generic (width: natural:=4);
    port (
          in0, in1: in std_logic_vector(width-1 downto 0);
               enable, sel: in std_logic;
               z: out std_logic_vector(width-1 downto 0));
      end component mux_x;

      component less_than is
          generic (width: natural);
    Port ( in_a : in std_logic_vector(width-1 downto 0);
           in_b : in std_logic_vector(width-1 downto 0);
           out_c: out std_logic);
end component less_than;

    signal low_1, low_2, low_3, low_4, range_2, range_3 : std_logic_vector(17 downto 0);
    signal data_1 : std_logic_vector(17 downto 0);
    signal out_bytestream_ptr_renorm : std_logic_vector(31 downto 0);
    signal renorm, mux_lt : std_logic;

    begin

--shift
low_1(0) <= '0';
low_1(17 downto 1) <= in_low(16 downto 0);

--and / not
renorm <= not (low_1(7) or low_1(6) or low_1(5) or low_1(4) or low_1(3) or
            low_1(2) or low_1(1) or low_1(0));
```

```
    data_1(0) <= '0';
    data_1(8 downto 1) <= data(7 downto 0);
58  data_1(17 downto 9) <= "000000000";

    low_2 <= low_1 + data_1;
    low_3 <= low_2 - "000000000011111111"; --0xFF

63  --mux

    renorm_or_not_low: mux_x generic map (18) port map(
                    in0 => low_1,
                    in1 => low_3,
68                  sel => renorm,
                    enable => '1',
                    z => low_4);

    out_bytestream_ptr_renorm <= in_bytestream_ptr + 1;
73
    renorm_or_not_bytestream: mux_x generic map (32) port map(
                    in0 => in_bytestream_ptr,
                    in1 => out_bytestream_ptr_renorm,
                    sel => renorm,
78                  enable => '1',
                    z => out_bytestream_ptr);


    --second part
83
    range_2(8 downto 0) <= "000000000";
    range_2(17 downto 9) <= in_range(8 downto 0);

    --compare
88
    less_than_1: less_than generic map (18) port map(
                    in_a => low_4,
                    in_b => range_2,
                    out_c => mux_lt);
93


    range_3 <= low_4 - range_2;

98  low_out_mux: mux_x generic map (18) port map(
                    in0 => range_3,
                    in1 => low_4,
                    sel => mux_lt,
                    enable => '1',
103                 z => out_low);


    result(31 downto 1) <= "00000000000000000000000000000000";
    result(0) <= not(mux_lt);
108
    end behavioural;
```

Listing A.5: cabac_bypass.vhdl

```
 1  library IEEE;
    --use IEEE.numeric_bit.all;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.STD_LOGIC_ARITH.ALL;
    use IEEE.STD_LOGIC_UNSIGNED.ALL;
 6  use ieee.numeric_std.all;
    use ieee.std_logic_textio.all;
    use std.textio.all;


11  entity cabac_tb is
    end cabac_tb;

    architecture behavioural of cabac_tb is

16  component cabac is
        Port ( in_mode : in std_logic_vector(31 downto 0);
               in_state_idx : in std_logic_vector(31 downto 0);
               clock : in std_logic;
               reset : in std_logic;
21             out_result : out std_logic_vector(31 downto 0)
               );
    end component cabac;
    --component cabac is
    --  port (
26  --    reset : in STD_LOGIC := 'X';
    --    clock : in STD_LOGIC := 'X';
    --    out_result : out STD_LOGIC_VECTOR ( 31 downto 0 );
```

```
--     in_state_idx : in STD_LOGIC_VECTOR ( 31 downto 0 );
--     in_mode : in STD_LOGIC_VECTOR ( 31 downto 0 )
--   );
--end component cabac;

signal in_mode : std_logic_vector(31 downto 0);
signal in_state_idx : std_logic_vector(31 downto 0);
signal clock : std_logic := '1';
signal reset : std_logic := '1';
signal out_result : std_logic_vector(31 downto 0);

constant Tpw_clk : time := 10 ns;

begin
    tb_1: cabac port map(
    in_mode => in_mode,
    in_state_idx => in_state_idx,
    clock => clock,
    reset => reset,
    out_result => out_result
    );


    clock_gen: process is
        begin
            clock <= '0' after Tpw_clk, '1' after 2*Tpw_clk;
            wait for 2*Tpw_clk;
        end process clock_gen;

    tb_2: process
          variable a : integer;
    begin


        reset <= '1';
        in_mode <= x"00000000";
        in_state_idx <=x"00000000";

        wait for 20 ns;


        reset <= '0';
        in_mode <= x"00000000";
        wait for 20 ns;

        for a in 1 to 460 loop
          in_state_idx <= std_logic_vector(to_unsigned(a,32));

        wait for 20 ns;
       end loop;

    end process;



end behavioural;
```

Listing A.6: cabac_tb.vhdl

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;


entity ff_h264_norm_shift is
    Port ( address : in std_logic_vector(7 downto 0);
           data_out : out std_logic_vector(3 downto 0));
end ff_h264_norm_shift;

architecture behavioural of ff_h264_norm_shift is

    type ROM_Array is array (0 to 255) of integer;

    constant Content: ROM_Array :=(
    9,8,7,7,6,6,6,6,5,5,5,5,5,5,5,5,
 4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,
 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,
 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
```

```
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
);


begin

       data_out <= std_logic_vector(to_unsigned(Content(conv_integer(address)),4));

end behavioural;
```

Listing A.7: ff_h264_norm_shift.vhdl

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;


entity ff_h264_norm_shift_lps is
    Port ( address : in std_logic_vector(8 downto 0);
           data_out : out std_logic_vector(3 downto 0));
end ff_h264_norm_shift_lps;

architecture behavioural of ff_h264_norm_shift_lps is

    type ROM_Array is array (0 to 511) of integer;

    constant Content: ROM_Array :=(
    -2,-1,0,0,1,1,1,1,2,2,2,2,2,2,2,2,
 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,
 4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
 4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
 5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
 5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
 5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
 5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
 6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
 6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
 6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
 6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7);


begin

       data_out <= std_logic_vector(to_signed(Content(conv_integer(address)),4));

end behavioural;
```

Listing A.8: ff_h264_norm_shift_lps.vhdl

```
library IEEE;
use IEEE.numeric_bit.all;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;
use STD.textio.all;

entity get_cabac is
    port(
        state_idx: in std_logic_vector(8 downto 0); --9 bits state_idx?
        in_low: in std_logic_vector(17 downto 0);
        in_range: in std_logic_vector(8 downto 0);
        in_bytestream_ptr: in std_logic_vector(31 downto 0);
        in_data: in std_logic_vector(7 downto 0);
        clock: in std_logic;
        reset: in std_logic;

        out_low: out std_logic_vector(17 downto 0);
        out_range : out std_logic_vector(8 downto 0);
        out_bytestream_ptr : out std_logic_vector(31 downto 0);
```

```
          out_result : out std_logic_vector(31 downto 0)

          --result: out std_logic;
          --clk: in std_logic;
          --res: in std_logic;
          --bytestream_addr_out: out std_logic_vector(8 downto 0);
          --bytestream_data_in: in std_logic_vector(7 downto 0);
          --out_low_register_out: out std_logic_vector(17 downto 0);
          --out_range_register_out: out std_logic_vector(8 downto 0);
          --mps_enable_out : out std_logic


          );
end entity get_cabac;

architecture structural of get_cabac is

--
-- Components
--

component local_cabac_state is
     Port ( address : in std_logic_vector(8 downto 0);
            data_out : out std_logic_vector(7 downto 0);
            data_in : in std_logic_vector(6 downto 0);
            clk : in std_logic;
            res: in std_logic);
end component local_cabac_state;

component new_lps_range is
     Port ( address : in std_logic_vector(8 downto 0);
            data_out : out std_logic_vector(7 downto 0));
end component new_lps_range;

component new_mps_state is
     Port ( address : in std_logic_vector(7 downto 0);
            data_out : out std_logic_vector(6 downto 0));
end component new_mps_state;

component new_lps_state is
     Port ( address : in std_logic_vector(7 downto 0);
            data_out : out std_logic_vector(6 downto 0));
end component new_lps_state;

component ff_h264_norm_shift is
     Port ( address : in std_logic_vector(7 downto 0);
            data_out : out std_logic_vector(3 downto 0));
end component ff_h264_norm_shift;

component ff_h264_norm_shift_lps is
     Port ( address : in std_logic_vector(8 downto 0);
            data_out : out std_logic_vector(3 downto 0));
end component ff_h264_norm_shift_lps;

component substract is
     Port ( in_a : in std_logic_vector(8 downto 0);
            in_b : in std_logic_vector(7 downto 0);
            out_c: out std_logic_vector(8 downto 0));
end component substract;

component less_than is
     Port ( in_a : in std_logic_vector(8 downto 0);
            in_b : in std_logic_vector(8 downto 0);
            out_c: out std_logic);
end component less_than;

component mux_x is
     generic (width: natural:=4);
     port (
     in0, in1: in std_logic_vector(width-1 downto 0);
          enable, sel: in std_logic;
          z: out std_logic_vector(width-1 downto 0));
end component mux_x;

component if_MPS is
     Port ( in_low : in std_logic_vector(17 downto 0);
            in_range : in std_logic_vector(8 downto 0);
            input_bytestream: in std_logic_vector(31 downto 0);-- enough bits?
            in_bytestream_ptr: in std_logic_vector(31 downto 0);-- enough bits?
            out_bytestream_ptr: out std_logic_vector(31 downto 0);-- enough bits?
            out_low: out std_logic_vector(17 downto 0);
            out_range: out std_logic_vector(8 downto 0));
end component if_MPS;

component if_LPS is
     Port ( in_low : in std_logic_vector(17 downto 0);
            in_range : in std_logic_vector(8 downto 0);
```

```
            in_rLPS : in std_logic_vector(7 downto 0);
            in_bits: in std_logic_vector(3 downto 0);
            in_bytestream_ptr: in std_logic_vector(31 downto 0);-- enough bits?
            input_bytestream: in std_logic_vector(31 downto 0);-- enough bits?
            out_bytestream_ptr: out std_logic_vector(31 downto 0);-- enough bits?
            out_low: out std_logic_vector(17 downto 0);
            out_range: out std_logic_vector(8 downto 0));
end component if_LPS;

--component bytestream_ptr_register is
--    Port (
--           clock : in std_logic;
--           reset : in std_logic;

--           data_set: in std_logic_vector(8 downto 0);
--           data: in std_logic_vector(8 downto 0);
--           output: out std_logic_vector(8 downto 0));
--end component bytestream_ptr_register;

--component low_register is
 --    Port (
--           clock : in std_logic;
--           reset : in std_logic;

--           data_set: in std_logic_vector(17 downto 0);
--           data: in std_logic_vector(17 downto 0);
--           output: out std_logic_vector(17 downto 0));
--end component low_register;

--component range_register is
--    Port (
--           clock : in std_logic;
--           reset : in std_logic;

--           data: in std_logic_vector(8 downto 0);
--           output: out std_logic_vector(8 downto 0));
--end component range_register;

component new_input_bytestream is
    port(   in_bytestream : in std_logic_vector(7 downto 0);
            newinput_bytestream : out std_logic_vector(31 downto 0)
            );
end component new_input_bytestream;


--
-- Signals
--
signal state: std_logic_vector(7 downto 0);
signal range_2: std_logic_vector(8 downto 0); --9 bits?

--signal low_1: std_logic_vector(17 downto 0);

signal state_and_range: std_logic_vector(8 downto 0);

signal rLPS: std_logic_vector(7 downto 0);
signal bits: std_logic_vector(3 downto 0);

signal next_mps_state: std_logic_vector(6 downto 0);
signal next_lps_state: std_logic_vector(6 downto 0);
signal next_cabac_state: std_logic_vector(6 downto 0);

signal mps_enable: std_logic;

--signal shift_amount: std_logic;


signal out_low_lps, out_low_mps: std_logic_vector(17 downto 0);
signal out_range_lps, out_range_mps: std_logic_vector(8 downto 0);
signal out_bytestream_ptr_lps, out_bytestream_ptr_mps: std_logic_vector(31 downto 0);

signal newinput_bytestream: std_logic_vector(31 downto 0);
--signal bytestream : std_logic_vector(7 downto 0);

begin



--
-- Signal maps
--
state_and_range(8 downto 7) <= in_range(7 downto 6);
state_and_range(6 downto 0) <= state(6 downto 0);

--
```

```
196 -- Port maps
    --

    get_input_bytestream: new_input_bytestream port map(
                in_bytestream => in_data,
201             newinput_bytestream => newinput_bytestream);


    get_local_cabac_state: local_cabac_state port map(
                    address => state_idx,
206                 data_out => state,
                    data_in => next_cabac_state,
                    clk => clock,
                    res => reset);

211
    get_new_lps_range: new_lps_range port map(
                    address => state_and_range,
                    data_out => rLPS);

216 get_bits: ff_h264_norm_shift port map(
                    address => rLPS,
                    data_out => bits);

    next_mps_state_1: new_mps_state port map(
221                 address => state,
                    data_out => next_mps_state);

    next_lps_state_1: new_lps_state port map(
                    address => state,
226                 data_out => next_lps_state);

    substract_1: substract port map(
                    in_a => in_range,
                    in_b => rLPS,
231                 out_c => range_2);

    less_than_1: less_than port map(
                    in_a => in_low(17 downto 9),--(8 downto 0),
                    in_b => range_2,
236                 out_c => mps_enable);

    select_mps_lps: mux_x generic map (7) port map(
                    in0 => next_lps_state,
                    in1 => next_mps_state,
241                 sel => mps_enable,
                    enable => '1',
                    z => next_cabac_state);

    if_mps_1: if_mps port map(
246                 in_low => in_low,
                    in_range => range_2,
                    input_bytestream => newinput_bytestream,
                    in_bytestream_ptr => in_bytestream_ptr,
                    out_bytestream_ptr => out_bytestream_ptr_mps,
251                 out_low => out_low_mps,
                    out_range => out_range_mps
    );

    if_lps_1: if_lps port map(
256                 in_low => in_low,
                    in_range => range_2,
                    in_rLPS => rLPS,
                    in_bits => bits,
                    in_bytestream_ptr => in_bytestream_ptr,
261                 input_bytestream => newinput_bytestream,
                    out_bytestream_ptr => out_bytestream_ptr_lps,
                    out_low => out_low_lps,
                    out_range => out_range_lps
    );
266
    -- registers for bytestream, low and range
    --bytestr_1: bytestream_ptr_register port map(
    --          clock => clk,
    --          reset => res,
271 --          data_set => "000000000",
    --          data => out_bytestream_ptr_register,
    --          output => out_bytestream_ptr_in
    --);

276 --low_reg1: low_register port map(
    --          clock => clk,
    --          reset => res,
    --          data_set => "0000000000000010",--first 12 bits of the bytestream +2
    --          data => out_low_register,
281 --          output => low_1
    --);
```

```
    --range_reg1: range_register port map(
    --            clock => clk,
286 --            reset => res,
    --            data => out_range_register,
    --            output => range_1
    --);

291 -- mux low, range and *bytestream.. if mps or lps

    low_mps_lps: mux_x generic map (18) port map(
                  in0 => out_low_lps,
                  in1 => out_low_mps,
296               sel => mps_enable,
                  enable => '1',
                  z => out_low);

    range_mps_lps: mux_x generic map (9) port map(
301               in0 => out_range_lps,
                  in1 => out_range_mps,
                  sel => mps_enable,
                  enable => '1',
                  z => out_range);
306
    bytestream_mps_lps: mux_x generic map (32) port map(
                  in0 => out_bytestream_ptr_lps,
                  in1 => out_bytestream_ptr_mps,
                  sel => mps_enable,
311               enable => '1',
                  z => out_bytestream_ptr);



316




    --
321 -- Small logic
    --
    out_result(0) <= state(0) xor (not mps_enable);
    out_result(31 downto 1) <= "00000000000000000000000000000000";

326
    --out_low_register_out <= out_low_register;
    --out_range_register_out <= out_range_register;
    --mps_enable_out <= mps_enable;

331 --
    -- Mux
    --

    --next_cabac_state <= (next_lps_state and (not mps_enable)) or (next_mps_state and mps_enable);
336
    end architecture structural;
```

Listing A.9: get_cabac.vhdl

```
 1  library IEEE;
    --use IEEE.numeric_bit.all;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.STD_LOGIC_ARITH.ALL;
    use IEEE.STD_LOGIC_UNSIGNED.ALL;
 6  use ieee.numeric_std.all;
    use ieee.std_logic_textio.all;
    use std.textio.all;


11  entity get_cabac_tb is
    end get_cabac_tb;

    architecture behaviour of get_cabac_tb is

16      component get_cabac is
        port(
            state_idx: in std_logic_vector(8 downto 0); --9 bits state_idx?
            result: out std_logic;
            clk: in std_logic;
21          res: in std_logic;
            bytestream_addr_out: out std_logic_vector(8 downto 0);
            bytestream_data_in: in std_logic_vector(7 downto 0);
            out_low_register_out: out std_logic_vector(17 downto 0);
            out_range_register_out: out std_logic_vector(8 downto 0);
26          mps_enable_out : out std_logic
            );
```

```
end component get_cabac;

signal state_idx : std_logic_vector(8 downto 0);
signal clk : std_logic;
signal res : std_logic;
signal bytestream_data_in : std_logic_vector(7 downto 0);

signal result : std_logic;
signal bytestream_addr_out : std_logic_vector(8 downto 0);
signal low : std_logic_vector(19 downto 0);
signal range_out : std_logic_vector(11 downto 0);
signal mps : std_logic;

constant Tpw_clk : time := 10 ns;

begin
    tb_1: get_cabac port map(
    state_idx => state_idx,
    result => result,
    clk => clk,
    res => res,
    bytestream_addr_out => bytestream_addr_out,
    bytestream_data_in => bytestream_data_in,
    out_low_register_out => low(17 downto 0),
    out_range_register_out => range_out(8 downto 0),
    mps_enable_out => mps
    );

    low(19 downto 18) <= "00";
    range_out(11 downto 9) <= "000";

    clock_gen: process is
        begin
            clk <= '0' after Tpw_clk, '1' after 2*Tpw_clk;
            wait for 2*Tpw_clk;
        end process clock_gen;


    tb_2: process
       variable a : integer;

       file outfile : text open write_mode is "cabac_testbench.txt";
       variable buf : line;
       variable start_msg : string(1 to 30) := "state_idx low range mps result";

    begin

        write( buf,start_msg);
        writeline(outfile, buf);

        res <= '1';


        wait for 20 ns;

        res <= '0';
        state_idx <="000000000";
        bytestream_data_in <=x"FF";




        wait for 19 ns;


        for a in 1 to 460 loop
           state_idx <= std_logic_vector(to_unsigned(a,9));


           write(buf,conv_integer(state_idx));
           write(buf,' ');
           hwrite(buf, low);
           write(buf,' ');
           hwrite(buf, range_out);
           write(buf,' ');
           write(buf, conv_integer(mps));
           write(buf,' ');
           write(buf, conv_integer(result));

        writeline(outfile, buf);
        wait for 20 ns;
       end loop;


        file_close(outfile);
```

```
116         wait;
        end process;

    end;
```

Listing A.10: get_cabac_tb.vhdl

```
 1 --get_cabac_terminate
   library IEEE;
   use IEEE.STD_LOGIC_1164.ALL;
   use IEEE.STD_LOGIC_ARITH.ALL;
   use IEEE.STD_LOGIC_UNSIGNED.ALL;
 6 use ieee.numeric_std.all;

   entity get_cabac_terminate is
       Port ( in_low : in std_logic_vector(17 downto 0);
              in_bytestream_ptr: in std_logic_vector(31 downto 0);
11            in_range : in std_logic_vector(8 downto 0);

              in_bytestream_start : in std_logic_vector(31 downto 0);

              data : in std_logic_vector(7 downto 0);
16
              out_low: out std_logic_vector(17 downto 0);
              out_range: out std_logic_vector(8 downto 0);
              out_bytestream_ptr: out std_logic_vector(31 downto 0);-- enough bits?
              result : out std_logic_vector(31 downto 0)
21            );
   end entity get_cabac_terminate;

   architecture behavioural of get_cabac_terminate is

26     component mux_x is
       generic (width: natural:=4);
       port (
             in0, in1: in std_logic_vector(width-1 downto 0);
                  enable, sel: in std_logic;
31                z: out std_logic_vector(width-1 downto 0));
         end component mux_x;

         component less_than is
             generic (width: natural);
36     Port ( in_a : in std_logic_vector(width-1 downto 0);
              in_b : in std_logic_vector(width-1 downto 0);
              out_c: out std_logic);
   end component less_than;

41 signal range_2, range_3, range_2_shifted : std_logic_vector(8 downto 0);
   signal range_2_lt, low_shifted, low_3, low_4, low_5, low_6, data_1 : std_logic_vector(17 downto 0);
   signal out_mux_lt, shift, renorm : std_logic;
   signal range_2_wide, range_3_wide : std_logic_vector(31 downto 0);
   signal out_bytestream_ptr_renorm, bytestream_ptr_2, bytestream_result : std_logic_vector(31 downto 0);
46
   begin

       range_2 <= in_range - 2;

51     range_2_lt(8 downto 0) <= "000000000";
       range_2_lt(17 downto 9) <= range_2(8 downto 0);

   --compare

56 less_than_1: less_than generic map (18) port map(
                  in_a => in_low,
                  in_b => range_2_lt,
                  out_c => out_mux_lt);

61 --
   range_2_wide(31 downto 9) <= "00000000000000000000000";
   range_2_wide(8 downto 0) <= range_2(8 downto 0);

   range_3_wide <= range_2_wide - x"100";
66
   shift <= range_3_wide(31);

   --
   range_2_shifted(0) <= '0';
71 range_2_shifted(8 downto 1) <= range_2(7 downto 0);

   low_shifted(0) <= '0';
   low_shifted(17 downto 1) <= in_low(16 downto 0);

76 shift_range: mux_x generic map (9) port map(
                  in0 => range_2,
```

```
                in1 => range_2_shifted,
                sel => shift,
                enable => '1',
                z => range_3);

shift_low: mux_x generic map (18) port map(
                in0 => in_low,
                in1 => low_shifted,
                sel => shift,
                enable => '1',
                z => low_3);

--
--and / not
renorm <= not (low_3(7) or low_3(6) or low_3(5) or low_3(4) or low_3(3) or
           low_3(2) or low_3(1) or low_3(0));

data_1(0) <= '0';
data_1(8 downto 1) <= data(7 downto 0);
data_1(17 downto 9) <= "000000000";

low_4 <= low_3 + data_1;
low_5 <= low_4 - "000000000011111111"; --0xFF

renorm_or_not_low: mux_x generic map (18) port map(
                in0 => low_3,
                in1 => low_5,
                sel => renorm,
                enable => '1',
                z => low_6);

out_bytestream_ptr_renorm <= in_bytestream_ptr + 1;

renorm_or_not_bytestream: mux_x generic map (32) port map(
                in0 => in_bytestream_ptr,
                in1 => out_bytestream_ptr_renorm,
                sel => renorm,
                enable => '1',
                z => bytestream_ptr_2);

 --end muxes

 bytestream_result <= in_bytestream_ptr - in_bytestream_start;

 end_result: mux_x generic map (32) port map(
                in0 => bytestream_result,
                in1 => x"00000000",
                sel => out_mux_lt,
                enable => '1',
                z => result);

  end_low: mux_x generic map (18) port map(
                in0 => in_low,
                in1 => low_6,
                sel => out_mux_lt,
                enable => '1',
                z => out_low);

  end_range: mux_x generic map (9) port map(
                in0 => range_2,
                in1 => range_3,
                sel => out_mux_lt,
                enable => '1',
                z => out_range);

end_bytestream: mux_x generic map (32) port map(
                in0 => in_bytestream_ptr,
                in1 => bytestream_ptr_2,
                sel => out_mux_lt,
                enable => '1',
                z => out_bytestream_ptr);


end behavioural;
```

Listing A.11: get_cabac_terminate.vhdl

```
--if_LPS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
```

```
 6 use IEEE.STD_LOGIC_UNSIGNED.ALL;
   use ieee.numeric_std.all;

   entity if_LPS is
       Port ( in_low : in std_logic_vector(17 downto 0);
11            in_range : in std_logic_vector(8 downto 0);
              in_rLPS : in std_logic_vector(7 downto 0);
              in_bits: in std_logic_vector(3 downto 0);
              in_bytestream_ptr: in std_logic_vector(31 downto 0);-- enough bits?
              input_bytestream: in std_logic_vector(31 downto 0);-- enough bits?
16            out_bytestream_ptr: out std_logic_vector(31 downto 0);-- enough bits?
              out_low: out std_logic_vector(17 downto 0);
              out_range: out std_logic_vector(8 downto 0));
   end entity if_LPS;

21 architecture behavioural of if_LPS is

   --
   -- Components
   --
26
   component ff_h264_norm_shift_lps is
       Port ( address : in std_logic_vector(8 downto 0);
              data_out : out std_logic_vector(3 downto 0));
   end component ff_h264_norm_shift_lps;
31
   component bshift is
       generic (width:natural);
           port (x: in std_logic_vector (width-1 downto 0);
                 s: in std_logic_vector (3 downto 0);
36           z: out std_logic_vector (width-1 downto 0));
   end component bshift;

   component mux_x is
       generic (width: natural:=4);
41     port (
       in0, in1: in std_logic_vector(width-1 downto 0);
            enable, sel: in std_logic;
            z: out std_logic_vector(width-1 downto 0));
   end component mux_x;
46


   signal range_2: std_logic_vector(17 downto 0);
   signal x: std_logic_vector(17 downto 0);
   signal low_1, low_2: std_logic_vector(17 downto 0);
51 --signal low_bs_in, low_bs_out: std_logic_vector(31 downto 0);

   signal input_bytestream_1, renorm_low: std_logic_vector(17 downto 0);

   signal renorm: std_logic;
56 signal i: std_logic_vector(3 downto 0);
   signal bits_bs, i_bs: std_logic_vector(3 downto 0);
   signal rlps_bs: std_logic_vector(8 downto 0);

   signal input_bytestream_bs_in, input_bytestream_bs_out: std_logic_vector(31 downto 0);
61 signal out_bytestream_ptr_renorm : std_logic_vector(31 downto 0);

   begin

   range_2(17 downto 9) <= in_range;
66 range_2(8 downto 0) <= "000000000";
   low_1 <= in_low - range_2;

   -- bshift low

71 bshift_1: bshift generic map (18) port map(
      x => low_1,
      s => in_bits,
      z => low_2);

76
   -- bshift range = shift rLPS by bits
   rlps_bs(8) <='0';
   rlps_bs(7 downto 0) <= in_rLPS;

81 bshift_2: bshift generic map (9) port map(
      x => rlps_bs,
      s => in_bits,
      z => out_range);

86
   x <= (low_2 xor (low_2-1));


   ff_h264_norm_shift_lps_1: ff_h264_norm_shift_lps port map(
91                 address => x(15 downto 7),
                   data_out => i);
```

```
-- shift input_bytestream by i
96
bshift_3: bshift generic map (18) port map(
   x => input_bytestream(17 downto 0),
   s => i,
   z => input_bytestream_1);
101

renorm <= not (low_2(7) or low_2(6) or low_2(5) or low_2(4) or low_2(3) or
            low_2(2) or low_2(1) or low_2(0));

106


--make muxes

111 renorm_low <= low_2 + input_bytestream_1;

 renorm_or_not_low: mux_x generic map (18) port map(
               in0 => low_2,
               in1 => renorm_low,
116            sel => renorm,
               enable => '1',
               z => out_low);

 out_bytestream_ptr_renorm <= in_bytestream_ptr + 1;
121
 renorm_or_not_bytestream: mux_x generic map (32) port map(
               in0 => in_bytestream_ptr,
               in1 => out_bytestream_ptr_renorm,
               sel => renorm,
126            enable => '1',
               z => out_bytestream_ptr);

end behavioural;
```

Listing A.12: if_LPS.vhdl

```
--if_MPS

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
 5 use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity if_MPS is
10     Port ( in_low : in std_logic_vector(17 downto 0);
            in_range : in std_logic_vector(8 downto 0);
            input_bytestream: in std_logic_vector(31 downto 0);-- enough bits?
            in_bytestream_ptr: in std_logic_vector(31 downto 0);
            out_bytestream_ptr: out std_logic_vector(31 downto 0);
15          out_low: out std_logic_vector(17 downto 0);
            out_range: out std_logic_vector(8 downto 0));
end entity if_MPS;

architecture behavioural of if_MPS is
20
component mux_x is
     generic (width: natural:=4);
     port (
     in0, in1: in std_logic_vector(width-1 downto 0);
25         enable, sel: in std_logic;
           z: out std_logic_vector(width-1 downto 0));
end component mux_x;


30 signal shift_amount: std_logic;

signal shifted_range: std_logic_vector(8 downto 0);
signal shifted_low: std_logic_vector(17 downto 0);

35 signal renorm: std_logic;
signal low_mux, renorm_low: std_logic_vector(17 downto 0);

signal out_bytestream_ptr_renorm: std_logic_vector(31 downto 0);

40 begin

   shift_amount <= in_range(8) xor '1';
   shifted_range(8 downto 1) <= in_range(7 downto 0);
   shifted_range(0) <= '0';
45 shifted_low(17 downto 1) <= in_low(16 downto 0);
```

```
    shifted_low(0) <= '0';


shift_or_not_range: mux_x generic map (9) port map(
                in0 => in_range,
                in1 => shifted_range,
                sel => shift_amount,
                enable => '1',
                z => out_range);

shift_or_not_low: mux_x generic map (18) port map(
                in0 => in_low,
                in1 => shifted_low,
                sel => shift_amount,
                enable => '1',
                z => low_mux);

                renorm <= not (low_mux(7) or low_mux(6) or low_mux(5) or low_mux(4) or low_mux(3) or
            low_mux(2) or low_mux(1) or low_mux(0));


 renorm_low <= low_mux + input_bytestream(17 downto 0);

 renorm_or_not_low: mux_x generic map (18) port map(
                in0 => low_mux,
                in1 => renorm_low,
                sel => renorm,
                enable => '1',
                z => out_low);

 out_bytestream_ptr_renorm <= in_bytestream_ptr + 1;

 renorm_or_not_bytestream: mux_x generic map (32) port map(
                in0 => in_bytestream_ptr,
                in1 => out_bytestream_ptr_renorm,
                sel => renorm,
                enable => '1',
                z => out_bytestream_ptr);


end behavioural;
```

Listing A.13: if_MPS.vhdl

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--use ieee.numeric_std.all;


entity less_than is
    generic (width: natural:=9);
    Port ( in_a : in std_logic_vector(width-1 downto 0);
           in_b : in std_logic_vector(width-1 downto 0);
           out_c: out std_logic);
end entity less_than;

architecture behavioural of less_than is

begin
process(in_a, in_b)
begin

   if (in_a < in_b) then
       out_c <= '1';
       else
       out_c <= '0';
  end if;
end process;


end behavioural;
```

Listing A.14: less_than.vhdl

```
--under construction..
--local_cabac_state

--must make bram..

```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;


entity local_cabac_state is
    Port ( address : in std_logic_vector(8 downto 0);
           data_out : out std_logic_vector(7 downto 0);
           data_in : in std_logic_vector(6 downto 0);
           clk : in std_logic;
           res : in std_logic);
end local_cabac_state;

architecture behavioural of local_cabac_state is

component mux_x is
    generic (width: natural:=4);
    port (
    in0, in1: in std_logic_vector(width-1 downto 0);
         enable, sel: in std_logic;
         z: out std_logic_vector(width-1 downto 0));
end component mux_x;

   signal a_reg : std_logic_vector(8 downto 0);
   signal in0, in1 : std_logic_vector(7 downto 0);
   signal sel_data : std_logic;

    type RAM_Array is array (0 to 459) of integer;

    signal ram: RAM_Array := (
    124, 18, 21, 124, 18, 21, 123, 77, 22, 20, 24, 124, 124, 124, 124, 124, 124, 124,
        124, 124, 124, 124, 124, 124, 124, 124, 124, 124, 124, 124, 124, 124, 124, 124,
        124, 124, 124, 124, 124, 124, 124, 124, 124, 124, 124, 124, 124, 124, 124, 124,
        124, 124, 124, 124, 124, 124, 124, 124, 124, 124, 44, 0, 0, 0, 37, 45, 67, 15,
        44, 2, 104, 16, 11, 123, 75, 37, 19, 83, 123, 123, 123, 59, 123, 97, 123, 115,
        101, 115, 101, 2, 7, 39, 79, 11, 57, 51, 123, 19, 65, 53, 123, 16, 35, 23, 119,
        57, 45, 25, 13, 2, 7, 39, 4, 1, 4, 14, 3, 1, 4, 27, 26, 22, 56, 38, 50, 36, 34,
        38, 90, 24, 26, 86, 58, 2, 87, 71, 73, 53, 59, 47, 39, 43, 37, 45, 57, 13, 17,
        19, 6, 75, 71, 63, 21, 17, 21, 13, 34, 9, 2, 3, 50, 15, 12, 16, 2, 17, 124, 108,
        76, 90, 108, 88, 52, 90, 68, 58, 66, 36, 10, 2, 4, 50, 36, 48, 42, 38, 34, 44,
        28, 56, 40, 16, 22, 32, 51, 124, 124, 124, 124, 124, 124, 124, 124, 124, 124,
        124, 124, 124, 120, 88, 124, 118, 80, 124, 124, 124, 124, 116, 112, 122, 90,
        78, 30, 50, 4, 9, 67, 13, 44, 28, 20, 4, 10, 0, 15, 19, 53, 5, 74, 50, 40, 34,
        0, 7, 27, 25, 43, 55, 18, 8, 4, 1, 17, 23, 31, 47, 89, 65, 37, 29, 17, 19, 43,
        63, 65, 103, 27, 62, 32, 22, 4, 13, 29, 43, 51, 65, 124, 57, 39, 29, 5, 13, 2,
        8, 10, 21, 4, 12, 3, 29, 15, 9, 38, 4, 54, 46, 70, 68, 38, 52, 60, 42, 30, 2,
        32, 1, 79, 65, 63, 47, 41, 41, 41, 47, 5, 25, 23, 23, 10, 23, 37, 69, 61, 63,
        25, 19, 13, 21, 9, 3, 17, 2, 2, 7, 21, 16, 1, 13, 114, 88, 94, 98, 98, 104,
        96, 94, 80, 80, 86, 74, 38, 44, 30, 90, 82, 80, 70, 66, 54, 26, 12, 0, 25,
        35, 59, 11, 89, 124, 124, 124, 124, 124, 124, 124, 124, 124, 124, 122, 118,
        96, 54, 10, 124, 124, 64, 124, 124, 124, 116, 124, 112, 100, 112, 98, 100,
        66, 70, 46, 7, 82, 62, 24, 109, 93, 97, 41, 55, 49, 11, 33, 31, 9, 11, 18,
        5, 30, 19, 124, 124, 124, 124, 116, 110, 54, 14, 39, 21, 82, 58, 40, 18, 18,
        4, 1, 9, 55, 83, 65, 51, 51, 45, 17, 29, 43, 17, 11, 9, 3, 0, 12, 8, 124,
        124, 124, 124, 118, 106, 82, 52, 7);

  begin

       --data_out <= std_logic_vector(to_signed(Content(conv_integer(address)),8));


       process(clk)
           begin

               if (clk'event and clk = '1') then

                   ram(conv_integer(a_reg)) <= conv_integer(data_in);
              a_reg <= address;

    end if;
               end process;

       --process(address,a_reg)--is this allowed? no clk..
       --begin

               --if (address = a_reg) then
                --    data_out(6 downto 0) <= data_in;
                --    else

               --data_out <= std_logic_vector(to_unsigned(ram(conv_integer(address)),8));
               --end if;

               --end process;
```

```
            in0 <= std_logic_vector(to_unsigned(ram(conv_integer(address)),8));
            in1(7) <= '0';
            in1(6 downto 0) <= data_in;

            sel_data <= ((a_reg(0) and address(0)) or (not a_reg(0) and not address(0))) and
            ((a_reg(1) and address(1)) or (not a_reg(1) and not address(1))) and
            ((a_reg(2) and address(2)) or (not a_reg(2) and not address(2))) and
            ((a_reg(3) and address(3)) or (not a_reg(3) and not address(3))) and
            ((a_reg(4) and address(4)) or (not a_reg(4) and not address(4))) and
            ((a_reg(5) and address(5)) or (not a_reg(5) and not address(5))) and
            ((a_reg(6) and address(6)) or (not a_reg(6) and not address(6))) and
            ((a_reg(7) and address(7)) or (not a_reg(7) and not address(7)));


            data_bypass: mux_x generic map (8) port map(
                  in0 => in0,
                  in1 => in1,
                  sel => sel_data,
                  enable => '1',
                  z => data_out);


          --process(clk, res)
          --    begin
          --        if (res = '0') then
          --        if (clk'event and clk = '0') then

       --            ram(conv_integer(a_reg)) <= conv_integer(data_in);

       --   end if;
       --end if;
          --        end process;

    end behavioural;
```

Listing A.15: local_cabac_state.vhdl

```
    -- register 18 bits low

    library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.STD_LOGIC_ARITH.ALL;
    use IEEE.STD_LOGIC_UNSIGNED.ALL;
    use ieee.numeric_std.all;

    entity low_register is
        Port (
              clock : in std_logic;
              reset : in std_logic;
              data_set: in std_logic_vector(17 downto 0);
              data: in std_logic_vector(17 downto 0);
              output: out std_logic_vector(17 downto 0));
    end entity low_register;

    architecture behavioural of low_register is

    begin
        process(clock, reset)
            begin

          if (reset = '1') then
             output <= data_set;
          elsif (clock'event and clock = '1') then
             output <= data;
          end if;

    end process;

    end behavioural;
```

Listing A.16: low_register.vhdl

```
    --8 input mux
    --x wide

    library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.STD_LOGIC_ARITH.ALL;
    use IEEE.STD_LOGIC_UNSIGNED.ALL;

    entity mux8_x is
        generic (width: natural:=32);
```

```
    port (
    in0, in1, in2, in3, in4, in5, in6, in7 : in std_logic_vector(width-1 downto 0);
13          enable: in std_logic;
            sel: in std_logic_vector(2 downto 0);
            z: out std_logic_vector(width-1 downto 0));
end entity mux8_x;

18 architecture behavioural of mux8_x is


function zeros(z: std_logic_vector) return std_logic_vector is
variable outvalue: std_logic_vector(z'length-1 downto 0);
23 begin

    for i in z'range loop
    outvalue(i) := '0';
end loop;
28 return outvalue;
end function zeros;



33
begin

    process(in0, in1, in2, in3, in4, in5, in6, in7, enable, sel) is

38  begin

                if (enable='1' and sel="000") then
                    z <= in0;
                elsif (enable='1' and sel="001") then
43                  z <= in1;
                elsif (enable='1' and sel="010") then
                    z <= in2;
                elsif (enable='1' and sel="011") then
                    z <= in3;
48              elsif (enable='1' and sel="100") then
                    z <= in4;
                elsif (enable='1' and sel="101") then
                    z <= in5;
                elsif (enable='1' and sel="110") then
53                  z <= in6;
                elsif (enable='1' and sel="111") then
                    z <= in7;
                else
                    z <= zeros(in0);-- '0';
58              end if;



63
   end process;

end architecture behavioural;
```

Listing A.17: mux8_x.vhdl

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity mux_2 is
        port (in0, in1, sel: in std_logic;
9       z: out std_logic);
end entity mux_2;

architecture structural of mux_2 is

14 begin

      z <= (in0 and (not sel)) or (in1 and sel);

end architecture structural;
```

Listing A.18: mux_2.vhdl

```
library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
```

```
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mux_x is
    generic (width: natural:=4);
    port (
    in0, in1: in std_logic_vector(width-1 downto 0);
         enable, sel: in std_logic;
         z: out std_logic_vector(width-1 downto 0));
end entity mux_x;

architecture behavioural of mux_x is


function zeros(z: std_logic_vector) return std_logic_vector is
variable outvalue: std_logic_vector(z'length-1 downto 0);
begin

    for i in z'range loop
    outvalue(i) := '0';
end loop;
return outvalue;
end function zeros;




begin

    process(in0, in1, enable, sel) is

    begin

                if (enable='1' and sel='0') then
                    z <= in0;
                elsif (enable='1' and sel='1') then
                    z <= in1;
                else
                    z <= zeros(in0);-- '0';
                end if;




   end process;

end architecture behavioural;
```

Listing A.19: mux_x.vhdl

```
--new_input_bytestream

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity new_input_bytestream is
    port(   in_bytestream : in std_logic_vector(7 downto 0);
            newinput_bytestream : out std_logic_vector(31 downto 0)
            );

end entity new_input_bytestream;

architecture behavioural of new_input_bytestream is

    signal in_bytestream_large : std_logic_vector(31 downto 0):= x"00000000";

    begin

        in_bytestream_large(8 downto 1) <= in_bytestream; --1 shift
        newinput_bytestream <= in_bytestream_large + x"FFFFFF01";


end architecture behavioural;
```

Listing A.20: new_input_bytestream.vhdl

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
use ieee.numeric_std.all;


entity new_lps_range is
    Port ( address : in std_logic_vector(8 downto 0);
           data_out : out std_logic_vector(7 downto 0));
end new_lps_range;

architecture behavioural of new_lps_range is

    type ROM_Array is array (0 to 511) of integer;

    constant Content: ROM_Array :=(
128, 128, 128, 128, 128, 128, 123, 123, 116, 116, 111, 111, 105, 105, 100, 100, 95, 95, 90, 90, 85, 85, 81, 81
 56, 56, 53, 53, 51, 51, 48, 48, 46, 46, 43, 43, 41, 41, 39, 39, 37, 37, 35, 35, 33, 33, 32, 32, 30, 30, 29, 29
 20, 20, 19, 19, 18, 18, 17, 17, 16, 16, 15, 15, 14, 14, 14, 14, 13, 13, 12, 12, 12, 12, 11, 11, 11, 11, 10, 10
 7, 6, 6, 6, 6, 6, 6, 6, 2, 2, 176, 176, 167, 167, 158, 158, 150, 150, 142, 142, 135, 135, 128, 128, 122, 122, 116
 85, 85, 80, 80, 76, 76, 72, 72, 69, 69, 65, 65, 62, 62, 59, 59, 56, 56, 53, 53, 50, 50, 48, 48, 45, 45, 43, 43
 30, 30, 28, 28, 27, 27, 26, 26, 24, 24, 23, 23, 22, 22, 21, 21, 20, 20, 19, 19, 18, 18, 17, 17, 16, 16, 15, 15
 11, 11, 10, 10, 9, 9, 9, 9, 9, 9, 8, 8, 8, 8, 7, 7, 7, 7, 2, 2, 208, 208, 197, 197, 187, 187, 178, 178, 169, 169
 130, 123, 123, 117, 117, 111, 111, 105, 105, 100, 100, 95, 95, 90, 90, 86, 86, 81, 81, 77, 77, 73, 73, 69, 69
 48, 48, 46, 46, 43, 43, 41, 41, 39, 39, 37, 37, 35, 35, 33, 33, 32, 32, 30, 30, 29, 29, 27, 27, 26, 26, 25, 25
 17, 17, 16, 16, 15, 15, 15, 15, 14, 14, 13, 13, 12, 12, 12, 12, 11, 11, 11, 11, 10, 10, 10, 10, 9, 9, 9, 9, 8
 195, 195, 185, 185, 175, 175, 166, 166, 158, 158, 150, 150, 142, 142, 135, 135, 128, 128, 122, 122, 116, 116,
 80, 80, 76, 76, 72, 72, 69, 69, 65, 65, 62, 62, 59, 59, 56, 56, 53, 53, 50, 50, 48, 48, 45, 45, 43, 43, 41, 41
 28, 28, 27, 27, 25, 25, 24, 24, 23, 23, 22, 22, 21, 21, 20, 20, 19, 19, 18, 18, 17, 17, 16, 16, 15, 15, 14, 14
 10, 10, 9, 9, 2, 2);


begin

        data_out <= std_logic_vector(to_unsigned(Content(conv_integer(address)),8));

end behavioural;
```

Listing A.21: new_lps_range.vhdl

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;


entity new_lps_state is
    Port ( address : in std_logic_vector(7 downto 0);
           data_out : out std_logic_vector(6 downto 0));
end new_lps_state;

architecture behavioural of new_lps_state is

    type ROM_Array is array (0 to 128) of integer;

    constant Content: ROM_Array :=(
    1, 0, 0, 1, 2, 3, 4, 5, 4, 5, 8, 9, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 18, 19, 22, 23, 22, 23,
26, 27, 26, 27, 30, 31, 30, 31, 32, 33, 32, 33, 36, 37, 36, 37, 38, 39, 38, 39, 42, 43, 42, 43, 44, 45, 44, 45
46, 47, 48, 49, 48, 49, 50, 51, 52, 53, 52, 53, 54, 55, 54, 55, 56, 57, 58, 59, 58, 59, 60, 61, 60, 61, 60, 61
62, 63, 64, 65, 64, 65, 66, 67, 66, 67, 66, 67, 68, 69, 68, 69, 70, 71, 70, 71, 70, 71, 72, 73, 72, 73, 72, 73
74, 75, 74, 75, 74, 75, 76, 77, 76, 77, 126, 127, 2);


begin

        data_out <= std_logic_vector(to_unsigned(Content(conv_integer(address)),7));

end behavioural;
```

Listing A.22: new_lps_state.vhdl

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;


entity new_mps_state is
    Port ( address : in std_logic_vector(7 downto 0);
           data_out : out std_logic_vector(6 downto 0));
end new_mps_state;

architecture behavioural of new_mps_state is
```

```
    type ROM_Array is array (0 to 128) of integer;

    constant Content: ROM_Array :=(
    2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 3
   44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73,
   84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 1
  119, 120, 121, 122, 123, 124, 125, 124, 125, 126, 127, 128);


begin

        data_out <= std_logic_vector(to_unsigned(Content(conv_integer(address)),7));

end behavioural;
```

Listing A.23: new_mps_state.vhdl

```
-- register 9 bits range

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity range_register is
    Port (
          clock : in std_logic;
          reset : in std_logic;
          data: in std_logic_vector(8 downto 0);
          output: out std_logic_vector(8 downto 0));
end entity range_register;

architecture behavioural of range_register is

begin
    process(clock, reset)
        begin

      if (reset = '1') then
         output <= "111111110"; --0x1FE
      elsif (clock'event and clock = '1') then
         output <= data;
      end if;

end process;

end behavioural;
```

Listing A.24: range_register.vhdl

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;


entity substract is
    Port ( in_a : in std_logic_vector(8 downto 0);
           in_b : in std_logic_vector(7 downto 0);
           out_c: out std_logic_vector(8 downto 0));
end entity substract;

architecture behavioural of substract is

begin
   out_c <= in_a - in_b;


end behavioural;
```

Listing A.25: substract.vhdl

Benchmark program

B

```
%this is the software run om the xilinx machine


/* apu_to_cabac
 * made by: Martijn Berkhoff
 * CE, TU DELFT
 */

#include "xbasic_types.h"
#include "xcache_l.h"
#include "xparameters.h"
#include "xpseudo_asm.h"
#include "xutil.h"
#include "stdio.h"
#include "xuartns550_l.h"

#include "xtmrctr.h"

// Assembly mnemonics
#define lwfcmx(rn, base, adr)     __asm__ __volatile__(\
                                "lwfcmx " #rn ",%0,%1\n"\
                                : : "b" (base), "r" (adr)\
                              )

#define stwfcmx(rn, base, adr)     __asm__ __volatile__(\
                                    "stwfcmx " #rn ",%0,%1\n"\
                                    : : "b" (base), "r" (adr)\
                                  )

// Data structures
volatile Xint32 __attribute__ ((aligned (32))) src[4] = {214,49,-3,20};
volatile Xint32 __attribute__ ((aligned (32))) dst[460];




/**//**/
#include "xparameters.h"
#include "stdio.h"
#include "xutil.h"
#include "xuartns550_l.h"

/*int main()
{
        XUartNs550_SetBaud(XPAR_RS232_UART_1_BASEADDR, XPAR_XUARTNS550_CLOCK_HZ, 9600);
    XUartNs550_mSetLineControlReg(XPAR_RS232_UART_1_BASEADDR, XUN_LCR_8_DATA_BITS);

    print("\r\n");
    print("Hello World\r\n");
    print("By Martijn Berkhoff\r\n");



        return 0;
}*/

/*test cabac-core*/

/*
 * H.26L/H.264/AVC/JVT/14496-10/... encoder/decoder
 * Copyright (c) 2003 Michael Niedermayer <michaelni@gmx.at>
 *
 * This file is part of FFmpeg.
 *
```

```
 * FFmpeg is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * FFmpeg is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with FFmpeg; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
 */

/* MARTIJN EDIT */
/* ONLY CABAC DECODE */




/**
 * @file cabac.c
 * Context Adaptive Binary Arithmetic Coder.
 */

#include <string.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

//#include "libavutil/common.h"
//#include "bitstream.h"
//#include "cabac.h"

static const uint8_t new_lps_range[512]= {
128, 128, 128, 128, 128, 128, 123, 123, 116, 116, 111, 111, 105, 105, 100, 100, 95, 95, 90, 90, 85, 85, 81, 81, 77, 77, 7
 14, 14, 14, 13, 13, 12, 12, 12, 12, 11, 11, 11, 11, 10, 10, 9, 9, 2, 2
};

static const uint8_t new_mps_state[129]= {
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 3
};

static const uint8_t new_lps_state[129]= {
1, 0, 0, 1, 2, 3, 4, 5, 4, 5, 8, 9, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 18, 19, 22, 23, 22, 23, 24, 25, 26, 27,
};

static const uint8_t ff_h264_norm_shift[256]= {
 9,8,7,7,6,6,6,6,5,5,5,5,5,5,5,5,
 4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,
 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,
 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
};

static const uint8_t ff_h264_norm_shift_lps[512]= {
 /*-2,-1,*/0,0,0,0,1,1,1,1,2,2,2,2,2,2,2,2,
 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,
 4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
 4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
 5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
 5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
 5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
 5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
 6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
 6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
 6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
 6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
};
```

```
//static uint8_t local_cabac_state[460];
/*for test purposes*/
static uint8_t local_cabac_state[460] = {124, 18, 21, 124, 18, 21, 123, 77, 22, 20, 24, 124, 124, 124, 124, 12
160
int low;
int range;
const uint8_t *bytestream_start;
//const uint8_t *bytestream;
165  uint8_t *bytestream;


/* To transform get_cabac to a MOLEN CCU, input and output parameters have to be either values or pointers. mem
So ff-h264_tables need to be included or pointed at.. if included, the initialization needs also be included!
170 */

/*
modes:
0: get_cabac
175 1:
2: get_cabac_bypass
3: get_cabac_terminate
4: memcpy
5: ff_init_cabac_decoder
180 */

int get_cabac(int mode, int state_idx)
{

185 //
char mpslps;
switch (mode)
{

190 //get_cabac
case 0:

{
        uint8_t s;// = local_cabac_state[state_idx];
195     uint8_t rLPS;// = new_lps_range[((range&0xC0)<<1) | s]; //joining two lines [8..9]+[0..7] // bit 8 ove
        uint8_t bit, result, shift;
        int newinput_bytestream; // = (bytestream[0]<<1) + 0xFFFFFF01;

//state_idx = state_idx&0x1FF;//9 bits
200 //low = low&0x3FFFF;//18 bits
//range = range&0x1FF;//9 bits


        s = local_cabac_state[state_idx];
205 //s=s&0x7F; //s can be 7 bits?
        rLPS = new_lps_range[((range&0xC0)<<1) | s];
//rLPS=rLPS&0xFF; //8 bits

//printf("%x\t%8x\n", (((range&0xC0)<<1) | s), rLPS);
210     newinput_bytestream = (bytestream[0]<<1) + 0xFFFFFF01; //why?



        bit = ff_h264_norm_shift[rLPS];
215     range = range - rLPS;
//printf("%x\t%x\n", range, (low>>9));
//range = range&0x1FF;//9 bits

        shift = ((range&0x100)>>8)^0x1;//select 9th bit, shift down 8 and invert
220

        //if(low < (range<<9)) //if MPS
        if((low>>9) < (range)) //if MPS
        {
225     //      mpslps= 'M';
                result = s&0x1;
                local_cabac_state[state_idx] = new_mps_state[s]; //update state

                range = range<<shift;
230             low = low<<shift;

                if(!(low & 0xFF)) //renormalize
                {
                        low = low + newinput_bytestream;
235                     bytestream++;

                }
        }

240     else //LPS
        {
                int i;
                uint16_t x;
```

```
                //mpslps= 'L';

                result = (s&0x1)^0x1;
                local_cabac_state[state_idx] = new_lps_state[s]; //update state

                low = (low - (range<<9))<<bit; //value = value - range + rLPS
//printf("low=%x\n", low);
                        low = low &0x3FFFF;//may shift of the rest

                range = rLPS<<bit; //range = rLPS

                x = (low ^ (low-1))>>7; //this can be made more efficient in hw
//                      x= x&0x1FF;//9 bits
                i = ff_h264_norm_shift_lps[x]; //0,1,3,7,15,31,63 .. ' how many ones'..

                if(!(low & 0xFF))       //renormalize
                {
                        low = low + (newinput_bytestream<<i);
                        bytestream++;

                }
        //printf("x=%d, i=%d\n", x, i);
        }


        //printf("%d\t\t0x%8x\t0x%x\t%c\t%d\t%x\n", state_idx, low, range, mpslps, result, bytestream);


        return result;
}
break;


case 1:
{
}
break;

//get_cabac_bypass
case 2:
{
        int b_range;
        int result;
        //low = low + low;
        low = low<<1;

        if(!(low & 0xFF))
        {
                low+= bytestream[0]<<1;

                low -= 0xFF;
                bytestream+= 1;
        }

        b_range = range<<9;

        if(low < b_range)
        {
                result = 0;
        }
        else
        {
                low = low - b_range;
                result = 1;
        }

        printf("0x%8x\t0x%x\t%d\t%x\n", low, range, result, bytestream);
        return result;
}
break;

//get_cabac_terminate
case 3:
{
        range -= 2;
        if(low < range<<(9))
        {
                //renorm_cabac_decoder_once(c);
                {
                        int shift= (uint32_t)(range - 0x100)>>31;
                        range<<= shift;
                        low  <<= shift;

                        if(!(low & 0xFF))
                        //refill(c);
```

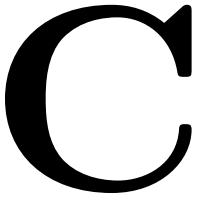
```
                        {
                        low+= bytestream[0]<<1;

                        low -= 0xFF;
                        bytestream+= 1;
                        }
                }
                return 0;
        }
        else
        {
                return bytestream - bytestream_start;
        }
}
break;

//memcpy
case 4:
{
        //uint8_t * cabac_state = state_idx;
        //memcpy ( local_cabac_state, cabac_state, 460);
}
break;

//ff_init_cabac_decoder
case 5:
{
        bytestream_start = bytestream = state_idx;

        low =  (*bytestream++)<<10;
        low += ((*bytestream++)<<2) + 2;
        range = 0x1FE;
}
break;

default:
break;

}
}
/**//**//**/


/*main*/
int main_old(int argc, char **argv)
{

        int i, k;

        /*initialize bytestream*/
        bytestream = malloc(sizeof(uint8_t)*1024);

        bytestream[0] = 0x00;
        bytestream[1] = 0x00;

        for (i=2;i<1024;i++)
        {
                bytestream[i] = 0xFF;
                //bytestream[i] = 0x00;
        }

        /*init cabac*/
        get_cabac(5, (int)&bytestream[0]);


        /*test cabac*/

        printf("state_idx\tlow\t\trange\tMPS/LPS\tresult\tbytestream\n");
        printf("\t\t0x%8x\t0x%x\t\t\n", low, range );
        /*for (k=0;k<460;k++)
        {
                //get_cabac(0, k);
                get_cabac(0, 0);

        }*/

        /*for (k=0; k<460;k++)
        {
                get_cabac(0, k);
        }*/
        for (k=0; k<460;k++)
        {
                get_cabac(0, k);
        }
        //printf("%d\n",get_cabac(3, 0));
        return 0;
```

```
}

int main(void)
{
#define PLB_CLOCK 100000000
#define PLB_CLOCK_PERIOD 10

//how many bins should get_cabac decode in one iteration?
#define units 100000


int i, input;
double r,x;

        //initialize uart
        XUartNs550_SetBaud(XPAR_RS232_UART_1_BASEADDR, XPAR_XUARTNS550_CLOCK_HZ, 9600);
        XUartNs550_mSetLineControlReg(XPAR_RS232_UART_1_BASEADDR, XUN_LCR_8_DATA_BITS);

        //printf disclaimer
        printf("\r\n");
        printf("APU_TO_CABAC\r\n");
        printf("By_Martijn_Berkhoff_-_CE,_TU_DELFT\r\n");


        //initialize TIMER
        XTmrCtr InstancePtr;
        u16 DeviceId = 0;
        u32 time_sw, time_hw, start_value, total_time_sw, total_time_hw;


        double speedup;
        if (XTmrCtr_Initialize(&InstancePtr, XPAR_XPS_TIMER_0_DEVICE_ID)==XST_SUCCESS)
        {
                        printf("Timer_initialized\n");

        }
        start_value = XTmrCtr_GetValue(&InstancePtr, 0);

        //initialize CABAC_SW_ONLY
        /*initialize bytestream*/
        bytestream = malloc(sizeof(uint8_t)*1024);

        bytestream[0] = 0x00;
        bytestream[1] = 0x00;

        for (i=2;i<1024;i++)
        {
                bytestream[i] = 0xFF;
                //bytestream[i] = 0x00;
        }

        /*init cabac*/
        get_cabac(5, (int)&bytestream[0]);

        //initialize APU
        mtmsr(XREG_MSR_APU_AVAILABLE);


int k;
for(k=0; k<5;k++) {
/**/
/*      printf("\n\n***\r\n");
        printf("Running SW design\r\n");
        printf("CPU: 300 MHz\r\n");
*/
        XTmrCtr_SetResetValue(&InstancePtr, 0, start_value);
        for (i=0; i<units;i++)
        {
                r = (    (double)rand() / ((double)(RAND_MAX)+(double)(1)) );
                x = (r * 460);
                input = (int) x;


                XTmrCtr_Start(&InstancePtr, 0);
                dst[input] = get_cabac(0, input);
                XTmrCtr_Stop(&InstancePtr, 0);
                XTmrCtr_SetResetValue(&InstancePtr, 0, XTmrCtr_GetValue(&InstancePtr, 0));
        }

        //TIMING_INTERVAL = (TLRx + 2) x PLB_CLOCK_PERIOD
        //CLK = 100 (MHz)
        //CLOCK_PERIOD = 0,00000001 (s) = 10 (ns)

        time_sw = (XTmrCtr_GetValue(&InstancePtr, 0)+2) * PLB_CLOCK_PERIOD;
//      printf("time: %d (ns)\n",time_sw);
        XTmrCtr_Reset(&InstancePtr, 0);
```

```
505 /**/
    /*      printf("\n\n***\r\n");
            printf("Running SW - HW codesign\r\n");
            printf("CPU: 300 MHz - APU: 25 MHz\r\n");
    */
510         XTmrCtr_SetResetValue(&InstancePtr, 0, start_value);

            for (i=0; i<units; i++)
            {
                    r = (    (double)rand() / ((double)(RAND_MAX)+(double)(1)) );
515                 x = (r * 460);
                    input = (int) x;

                    XTmrCtr_Start(&InstancePtr, 0);
                    lwfcmx(0, &input, 0);
520                 stwfcmx(0, dst, input*4);
                    XTmrCtr_Stop(&InstancePtr, 0);
                    XTmrCtr_SetResetValue(&InstancePtr, 0, XTmrCtr_GetValue(&InstancePtr, 0));
            }

525         time_hw = (XTmrCtr_GetValue(&InstancePtr, 0)+2) * PLB_CLOCK_PERIOD;
    //      printf("time: %d (ns)\n",time_hw);
            XTmrCtr_Reset(&InstancePtr, 0);

    /**/
530         //Calculating speedup
            speedup = ((double)time_sw/(double)time_hw);
    //      printf("\n\n***\r\n");
            printf("Speedup: %.5lf\r\n", speedup);
    //      printf("\nEnding APU_TO_CABAC\r\n");
535 }
            return 0;
    }
```

Listing B.1: Benchmark program C-code

Programming Files



This page contains a CD-ROM with the programming files made for the CABAC decoder accelerator.

The CD-ROM contains the following directories:

- **001_CABAC_ModelSimSE_6.3c**: This directory contains the programming files for the ModelSim simulations of the CABAC decoder accelerator.
- **002_CABAC_XilinxISE_10.1**: This directory contains the programming files of the hardware CABAC decoder accelerator for Xilinx ISE 10.1.
- **003_CABAC_XilinxPlatformStudioEDK_10.1**: This directory contains the programming files of the software and hardware CABAC decoder accelerator for the Xilinx ML410 platform.
- **004_CABAC_papers**: This directory contains scientific papers and journals on the topic of CABAC decoding.

Curriculum Vitae

Martinus Johannes Pieter Berkhoff was born in Delft, The Netherlands, on March 28, 1981. He obtained his VWO diploma in 2000 at the St. Stanislascollege in Delft. In the same year he started the Bachelor of Science study Electrical Engineering at the Technical University of Delft. During his Bachelor of Science study he did a minor in Physics Education at the Technical University Teachers Program and gave physics lectures to college students for several months. The research topic for his Bachelor of Science degree was titled: *"3D LED-Display software: Concepts and implementation of user software to control a three dimensional LED-display".*

After getting his Bachelor of Science degree in Electrical Engineering he choose to start the Master of Science study of Computer Engineering. The specialization within the Computer Engineering Laboratory was Embedded Systems. He will graduate in March 2009 by completing his Masters of Science thesis titled:*"Analysis and Implementation of the H.264 CABAC entropy decoding engine".*

After graduating for his Master of Science degree he will work for Imtech Marine & Offshore in Rotterdam. At this international operating company he will start as a consultant in drive technology for diesel-electric propulsion on ships.

MSc Computer Engineering Grades

Code	Name	ECTS	Grade
Common core			
ET8019:	Computer Arithmetic	9	7
ET4246:	Introduction to Comp. System Engineering	2	pass
ET4054:	Methods and Algoritms for System Design	5	8
ET4074:	Modern Computer Architectures	5	9
IN4020:	Compiler Construction	6	7
IN4026:	Parallel Algorithms and Parallel Computer Systems	6	7
Specialisation courses			
ET4361:	Networks on Chip	5	7
ET4362:	High Speed Digital Design for Embedded Systems	5	8
ET4368:	Embedded Application Development	5	9
IN4073TU:	Real Time Embedded Systems	6	7
IN4024:	Real-Time Systems	6	9
Free electives			
ET4263:	System Programming in C	2	pass
ET4272:	System System Design with HDL	2	pass
ET4036:	Transmission Systems Engineering	4	7
ET4284:	Ad-hoc Networks	4	6
ET4146:	Advances in Networking	4	7
Thesis project			
ET4300:	Master Thesis	45	