Optimizing the critical loop in the H.264/AVC CABAC decoder

Hendrik Eeckhaut [#1], Mark Christiaens [#2], Dirk Stroobandt [#3], Vincent Nollet [*4],

[#] *Parallel Information Systems Group, Department of Electronics and Information Systems*
Ghent University
Sint-Pietersnieuwstraat 41, Ghent, Belgium
[1] Hendrik.Eeckhaut@elis.UGent.be
[2] Mark.Christiaens@elis.UGent.be
[3] Dirk.Stroobandt@elis.UGent.be

[*] *DESICS, IMEC*
Kapeldreef 75, Leuven, Belgium
[4] nollet@imec.be

***Abstract*— In this paper we present an innovative hardware implementation of the H.264/AVC CABAC binary arithmetic decoder and context modeler capable of decoding one symbol per clock cycle at high clock frequencies while maintaining a slim hardware footprint. This was achieved by substantially decreasing the latency of the central feedback loop through extensive use of speculative prefetching and aggressive pipelining. Actual synthesis results targeted at the state-of-the-art FPGA families show that our approach results in a fast and compact IP core, ideal for a SoC H.264/AVC implementation.**

I. Introduction

H.264 or AVC (**A**dvanced **V**ideo **C**oding), is a new digital video codec standard which achieves a very high compression efficiency [1] compared to earlier standards. It was developed as an answer to the growing needs for better compression in a wide range of applications and for improved network friendliness. The compression efficiency is up to 50% over a wide range of bit rates and video resolutions compared to previous standards (e.g. MPEG2 or H.263). The downside is that the decoder complexity also increased; it is about four times higher [2].

Figure 1 shows H.264/AVC's basic coding structure for encoding one macro block, a sub block of a frame of the video stream. The decoder is used inside the encoder to obtain best perceptual quality at the decoder side. To reduce block-artifacts an adaptive deblocking filter is used in the motion compensation loop. This combined with multiple reference frames and sub-pixel inter and intra mode motion compensation gives very strong compression results. The Discrete Cosine Transform (DCT, 'Transform' in Figure 1) used in former standards is replaced with a faster integer transform.

The final step of the encoder is entropy coding, a sequential process that encodes the quantized DCT coefficients and the motion vector data as compact as possible. This is a lossless compression step that tries to exploit as much statistical redundancy as possible. The arithmetic codec used in H.264/AVC [3] was inspired by the Q-coder [4], [5], [6] and MQ-coder [7] but is faster and achieves better compression than state-of-the-art MQ-coders [8].

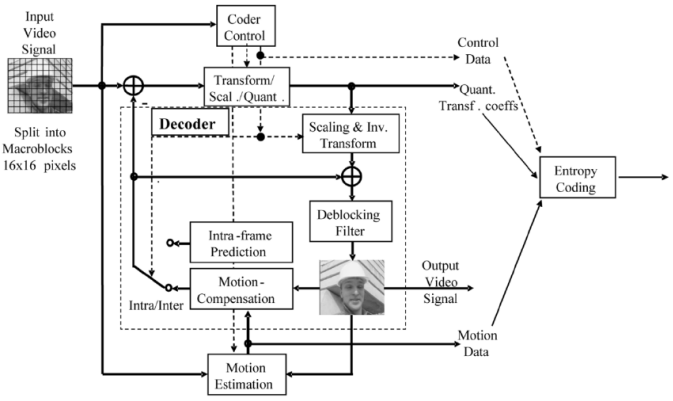

Fig. 1. H.264/AVC macroblock encoder with functional blocks and data flows. The decoder is a central part of the encoder. Picture reproduced from [1].

At the decoder side, the entropy decoder is the first step of the algorithm. So all encoded data has to through the entropy decoder before it can be processed by the other blocks (inside the dashed box in Figure 1). This becomes more and more problematic as the bit rate increases.

A. H.264/AVC CABAC

In H.264/AVC, two methods of entropy coding are supported: Context-Adaptive Variable-Length Coding (CAVLC) and Context-bAsed Binary Arithmetic Coding (CABAC) [3]. CABAC offers superior coding efficiency at the expense of greater computational complexity. By combining an adaptive binary arithmetic coding technique with context modeling it achieves a high degree of adaptation and redundancy reduction.

The CABAC encoder consists of three elementary steps: binarization, context modeling and binary arithmetic coding. The first step uniquely maps all incoming data to a binary sequence. These incoming data are the coefficients from the transformations in Figure 1 together with some context

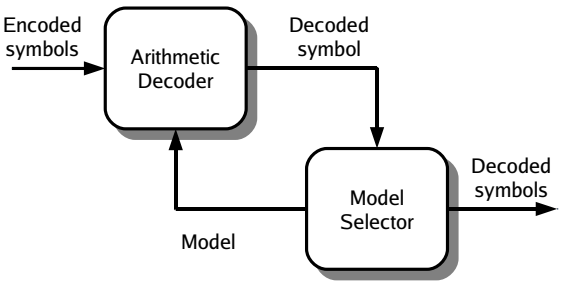

Fig. 2. The critical feedback loop in the entropy decoder: The model selector has to provide the arithmetic decoder with a new model that depends on the output of the arithmetic decoder.

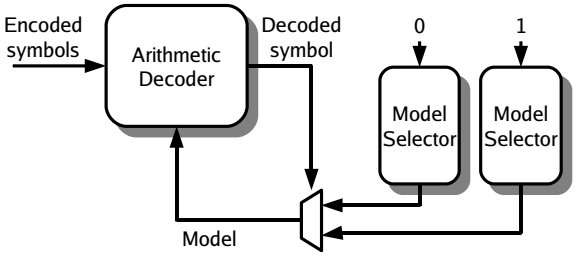

Fig. 3. Shorten critical feedback loop in the entropy decoder: Calculate the two possible models in parallel and choose when the decoded bit is known. Of course the two parallel model selectors use the decoded bit to update their states.

information. In the second step a fitting probability model, based on the context, is selected for each binary symbol. This model drives the arithmetic coder (step three) by providing an estimate of the probability density function (PDF) of the symbol that will be encoded. The better this estimate, the better the compression. CABAC uses in total 399 models to model the PDFs of each syntax element such as macro block type, motion vector data, texture data ... The models are kept 'up to date' during encoding through the use of an *adaptive* coder which estimates the PDF based on previously coded syntax elements.

In both the encoder and the decoder the context modeler and the arithmetic (de)coder are coupled in a tight feedback loop. At the encoder side a model is needed to encode the symbols and statistical data is read back from the arithmetic coder for switching several estimated probability models. At the decoder side the loop is even tighter. A model is needed to decode a symbol and the decoded symbol in turn is needed to calculate the next model as illustrated in Figure 2.

Since the arithmetic decoder has to process all encoded data in a sequential way, this feedback loop is a central bottleneck. It is a bottleneck for the entire decoder since all encoded data has to be processed by the arithmetic decoder before the other blocks (inverse DCT, motion compensation ...) can start decoding. This problem becomes more acute for increasing bit rates and resolutions. For example a CIF (352x288) resolution sequence at 30 frames per second and a YPSNR of around 40 dB has to produce about 1.5 million decoded symbols per second. A HDTV resolution (1280x720@30Hz) video sequence with high image quality (50 dB YPSNR) needs more than 50 million decoded symbols per second.

Because of its computational complexity we explored the option of accelerating the CABAC decoder with specialized hardware. For feasibility in an university environment, we examined an FPGA implementation, but all findings could be easily generalized to other platforms.

Hardware acceleration can radically speed up algorithms by exploiting available parallelism. But because there is so little parallelism in the CABAC part of the H.264/AVC codec, what is there to gain with a hardware implementation? Through advanced pipelining and speculative execution and data fetching, we managed to develop a decoder that processes one decoded symbol per clock cycle without consuming exaggerated resources.

The structure of this paper is as follows. In Section II we tackle the problem of the model selector. In Section III we cut down the time needed to decode a symbol. Section IV summarizes the results of a hardware implementation and Section V summarizes the main contributions of this work.

II. Context Modeling

To speed up CABAC, the critical loop of Figure 2 must be as fast as possible. Therefore we have to make the critical path through both the arithmetic decoder and the model selector as short as possible. At the model selector side it is possible to avoid waiting for the currently decoded bit, needed to calculate the next model, by speculatively calculating every possible next model in parallel. Since the currently decoded bit will be either 0 or 1, the selection process can result in only two distinct models. This is in fact only a Shannon decomposition. The final decision, at the time the decoded bit is known, can be performed by a simple multiplexer as illustrated in Figure 3. Of course the two parallel model selectors use the finally chosen model (or the result bit) to update their internal states in order to calculate the next two possible models in the next iteration.

III. Arithmetic Decoder

Binary arithmetic coding is based on the principle of recursive interval subdivision and its state is characterized by two quantities: the current interval `range` and the current `value` in the current code interval. The `value` is read from the encoded bitstream and, as illustrated in Figure 4, the interval `range` is subdivided in two regions: `rLPS` and `rMPS`, where LPS stands for Least Probable Symbol and MPS for Most Probable Symbol. These ranges correspond to the probabilities of LPS ($\texttt{rLPS} = \texttt{range} \times p_{LPS}$) and MPS ($\texttt{rangeMPS} = \texttt{range} \times p_{MPS} = \texttt{range} - \texttt{rLPS}$) provided by the model that was selected to encode the next symbol.

In other variants of arithmetic coding it was found that the multiplication by p_{LPS} is a major bottleneck. The CABAC-codec avoids this multiplication by approximation with a table-based approach.

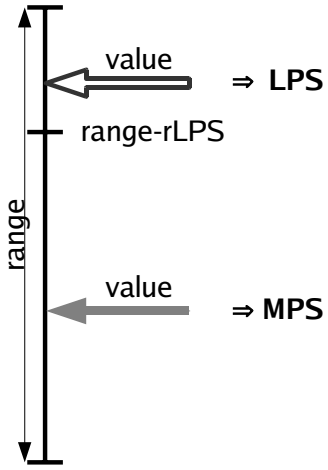

Fig. 4. Illustration of the meaning of range and value. If value is in the LPS range, the decoded bit is LPS. Otherwise the result is MPS.

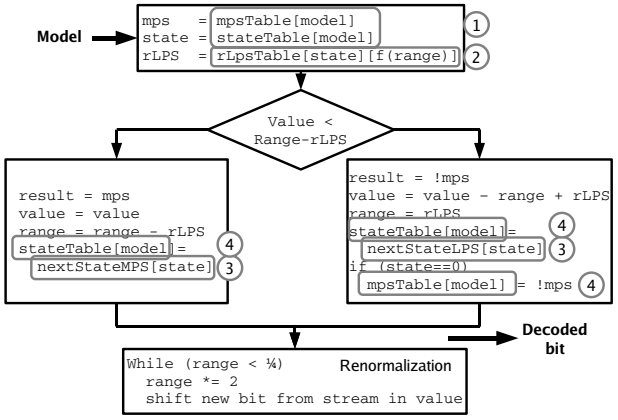

Fig. 5. The flow diagram of the binary arithmetic decoding process including the binary decision, the updating process of the PDF estimation and the renormalization for a single result bit. $f(range)$ is a simple function with only four possible result values: $\in \{0, 1, 2, 3\}$. The memory operations are labeled with sequential numbers.

The flow diagram for the binary arithmetic decoding process for a single binary value is illustrated in Figure 5. The decoding algorithm consists of three major parts.

In the first step the state of the current model, which corresponds to the current PDF of the current model, is fetched from the stateTable. Once the state is known, the corresponding LPS range (rLPS) can be fetched. This rLPS is an approximation, a quantized value (of range $\times\ p_{LPS}$), which is stored in a table that contains all quantized pre-computed product values in 8-bit precision.

In a second step the decoder verifies whether the current value falls in the MPS range or in the LPS range of the current interval. Depending on whether we are decoding a MPS or a LPS the value, range and state are updated. In the MPS case the new range is range$-$rLPS. Otherwise the new range becomes rLPS and because the zero point of range changes, value changes to value $-$ (range $-$ rLPS) (see also Figure 4). The new state is the result of a complicated function that performs the PDF estimation. Because of its complexity the state transition function is stored in two tables nextStateLPS and nextStateMPS.

In the final step the range is renormalized. If range becomes too small (i.e. smaller than one fourth of its maximum), it is doubled until it is large enough again to be split into two regions (rLPS and rMPS) with enough accuracy for decoding the next symbol. During renormalization new bits from the encoded bitstream are read into value to keep up with the rescaled range.

There are two major impediments for a fast execution of the decoding process in Figure 5: the algorithm needs multiple sequential memory operations and the third step, the renormalization loop, has a variable number of iterations.

A. Sequential Memory Accesses

As illustrated in Figure 5 four sequential memory accesses are needed per decoded bit. In the first read operation the MPS and state of the incoming model are fetched. The second memory operation is needed to fetch rLPS. A third read operation is needed to get the next state and this value is written back in the fourth operation to the stateTable. Only the fourth access, the write operation, can be pipelined by using dual port memories which enable reading and writing to the same memory at the same time. These memories are readily available on all modern FPGA-architectures. Still, three cycles are unsatisfactory, since then we would need very high clock frequencies and it would be hard to equally balance the work over the different cycles.

In a first code transformation (Figure 6) we replaced the two sequential read operations with only one by speculatively prefetching and caching rLPS for each model in a new table "spec_rLpsTable". This new table contains the row of the rLpsTable that corresponds with current state of the current model. The entire row has to be fetched because the second index ($f(range)$) is not known beforehand. It is not known when the considered model will occur again, so range could have any value. But since the result of $f(range)$ can only be four distinct values, the extra amount of data to be stored is limited.

Actually we are speculating on two levels: first, we fetch the rLPS-data for the next occurrence of the considered model, while it is not known when it will reoccur. This could be the next cycle, in ten cycles or it could never reoccur at all. On the second level op speculation, we fetch rLPS for all possible values of $f(range)$, while we know only one will be used.

Although, after this first transformation, we can start decoding earlier this does not shorten the loop since we still need multiple cycles to keep the newly introduced table up to date. This could possibly be pipelined but nevertheless cycles would be lost in case of subsequent iterations with the same model.

If we apply the same code transformation a second time by also speculatively prefetching and caching the next state

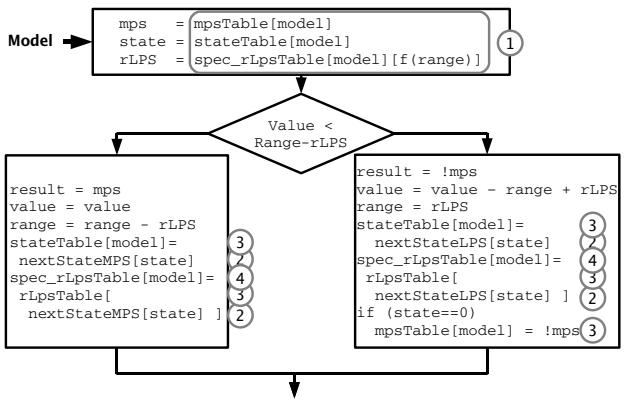

Fig. 6. The decoder algorithm after speculatively fetching rLPS data into a table indexed by the model number. The decoding calculations can now start after one read operation. The renormalization phase is here omitted.

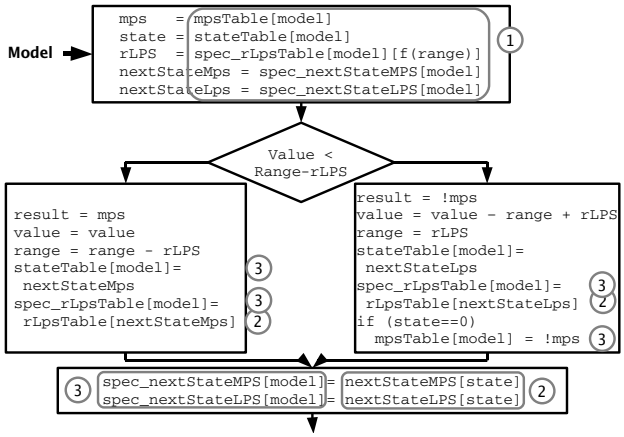

Fig. 7. By also speculatively fetching all next possible states, we make sure all potentially needed information is available with a single read operation.

for both the LPS and MPS decision, we obtain the code of Figure 7. The algorithm is now transformed to:

1) Fetch all the data needed to decode a bit for the current model and decode that bit. (Indexed with 'model'),
2) Speculatively prefetch all data needed at the next occurrence of the current model. (Indexed with 'state'),
3) Update the model tables with this data. (Indexed with `model`)

Of course we still have to renormalize after the calculations.

The advantage of these transformations is that we now can pipeline these three memory operations as illustrated in Figure 8. It is now possible to produce one decoded symbol per cycle. If a certain model reoccurs before its updated data is written to memory, this data is forwarded through the pipeline.

`rLpsTable`, `nextStateLPS` and `nextStateMPS` can be implemented in simple ROMs. For the tables, indexed with 'model', we can use simple dual port memories, which are abundantly available on modern FGPAs.

In Figure 7 we actually do not need to store the current

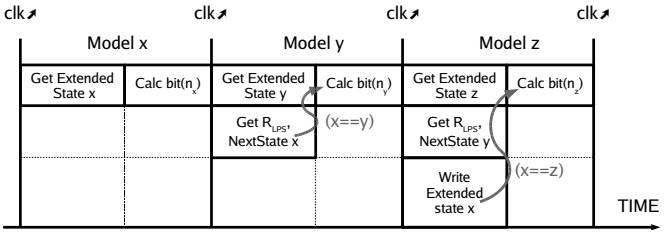

Fig. 8. The pipelining of the memory operations. If the same model used before the updated data for this model is written to the model tables, this information is forwarded through the pipeline.

state of the model any more. The current state is only tested to be zero or not. So we could, instead of storing the current state, only store whether it is zero or not.

Compared with the original approach of Figure 5, the critical loop is now much faster. The extra cost is limited: Some extra logic is needed to control the pipelining and forwarding and extra memory is needed to store the speculatively prefetched data. Per model this corresponds to four rLPS values and one bit plus two state values instead of just the current state. I.e. 45 bit instead of just 6. The associated increase in *bandwidth* is not a problem because all these memories are read in parallel over short connections.

B. Renormalization

In the original algorithm of Figure 5 the renormalization of `range` and `value` is specified as a sequential loop process that shifts only one bit per cycle until the `range` and `value` are renormalized. To speed up this process, a technique based on an idea of [9] was used. The loop condition, $range < 1/4\ (of\ the\ interval)$, corresponds actually with the number of zeros in front of the newly calculated `range`. So instead of shifting `range` and `value` over multiple iterations, we can simply count the number of leading zeroes of the new `range` and shift the new `range` this number of bits. In case of LPS, `range` is equal to rLPS. In case of MPS we can anticipate the number of leading zeroes of $Range - rLPS$. Leading Zero Anticipation (LZA) [10] can perform this count in parallel with the actual subtraction, which further reduces the time needed for normalization.

The core of the architecture is illustrated in Figure 9. We now shift a varying number (between 0 and the number or `range` bits minus one) of new (encoded) symbols into `value` every cycle. This implies a dedicated input buffer ('`SymbolBuffer`') that is able to deliver these symbols in time. This is nothing more than a buffer that reads the encoded data bits from the input stream and presents the decoder the symbols in front of the unprocessed stream. If the decoder consumes n symbols, the `SymbolBuffer` replaces these n symbols with n new bits from the stream. A block diagram of different parts of the implementation is given in Figure 10.

IV. Implementation Results

The design of Figure 9 and 10 is implemented in VHDL and synthesized for an Altera Stratix S25 (C5) and an Altera Stratix

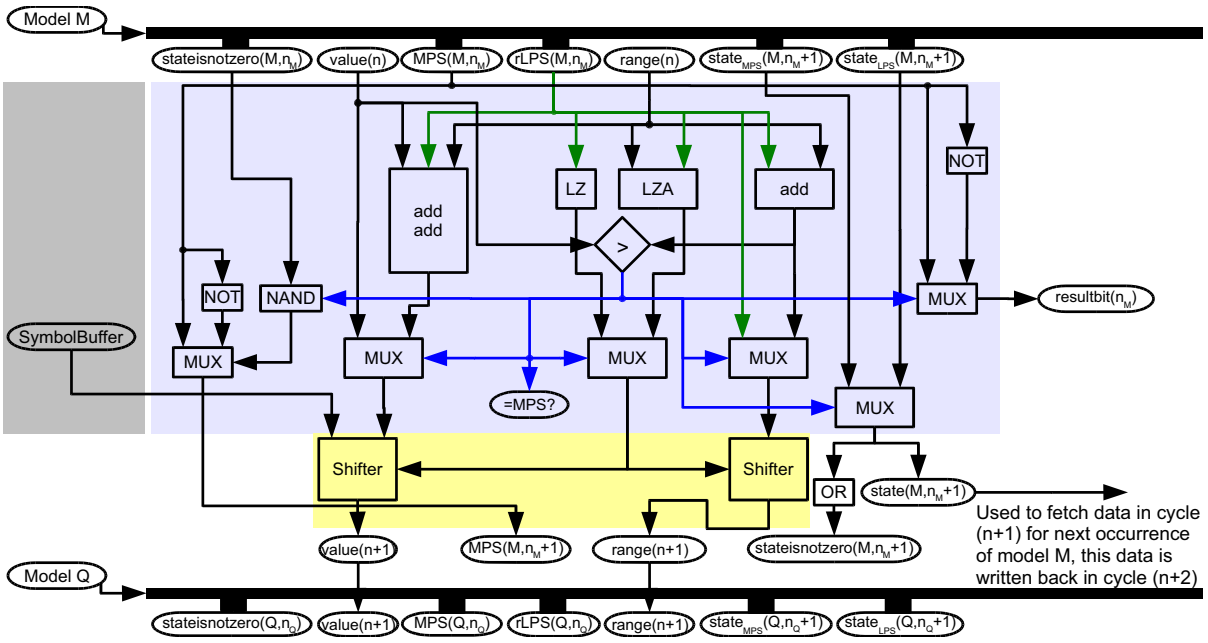

Fig. 9. Schematic of the calculations of the arithmetic decoder for decoding a result bit (n_M) with model M. The time goes from top to bottom; the bold, horizontal line indicates a rising clock edge and the black rectangles underneath it the memory read operations. For clarity, the schematic is simplified and the pipelining and forwarding infrastructure is omitted. LZ stands for Leading Zero (detection), LZA for Leading Zero Anticipation and MUX for Multiplexer.

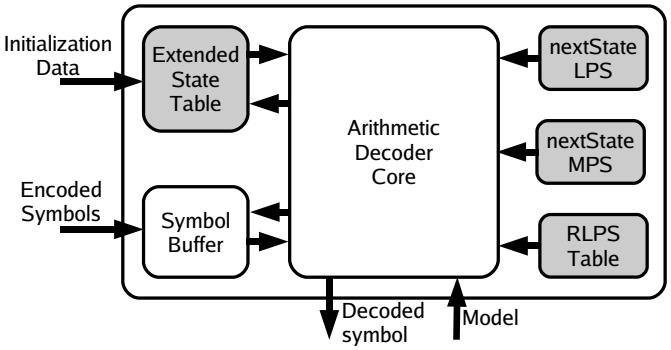

Fig. 10. Block diagram of the arithmetic decoder. The core (see Figure 9) is in the center and takes its (encoded) input symbols from the `SymbolBuffer`. The memories (single and dual port) are marked in gray.

II S60 (C3) FPGA. The synthesis results are summarized in Table I. The Stratix can decode at more than 70 MHz. I.e. more than 70 million symbols per second. The newer and faster Stratix II can decode more than 100 million symbols per second. As a consequence, one decoder is comfortably fast enough for high quality HDTV while consuming only very limited resources.

The design is written in a clean and generic way, so it could easily be used in other applications than H.264/AVC. It could be used as an IP core in any situation where a fast, compact and powerful arithmetic decoder is needed.

TABLE I
SYNTHESIS RESULTS

	Stratix S25	Stratix II S60
ArithmeticDecoder	1287 LEs	590 ALMs
→ArithmeticDecoderCore	684 LEs	389 ALMs
→SymbolBuffer	534 LEs	203 ALMs
Maximum frequency	71.97 MHz	105.37 MHz

(LE: Logic Element [11], ALM: Adaptive Logic Module [12])

V. CONCLUSION

In this work we have presented a novel FPGA-design for CABAC. CABAC is a key element in the new H.264/AVC video standard as it offers a high bit-rate reduction compared to the variable length coder CAVLC. We analyzed the bottlenecks of this computationally intensive algorithm and presented a solution that decodes one symbol per clock cycle using limited resources and achieving a high clock speed. In this new approach the data throughput is fixed which eases high level scheduling for real time behavior.

ACKNOWLEDGMENT

This research is supported by I.W.T. grant 020174, F.W.O. grant G.0021.03 and by GOA project 12.51B.02 of Ghent University. Altera provided development boards and tools through the Altera university program.

REFERENCES

[1] T. Wiegand, G. J. Sullivan, and G. B. A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 560–576, July 2003.

[2] J. Ostermann, J. Bormans, P. List, D. Marpe, M. Narroschke, F. Pereira, T. Stockhammer, and T. Wedi, "Video coding with H.264/AVC: tools, performance, and complexity," *IEEE Circuits and Systems Magazine*, vol. 4, no. 1, pp. 7–28, 2004.

[3] D. Marpe, H. Schwarz, and T. Wiegand, "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 620–636, July 2003.

[4] W. B. Pennebaker, J. L. Mitchell, G. G. J. Langdon, and R. B. Arps, "An overview of the basic principles of the Q-Coder adaptive binary arithmetic coder," *IBM Journal of Research and Development*, vol. 32, no. 6, pp. 717–726, November 1988.

[5] W. B. Pennebaker and J. L. Mitchell, "Probability estimation for the Q-Coder," *IBM Journal of Research and Development*, vol. 32, no. 6, pp. 737–752, November 1988.

[6] J. L. Mitchell and W. B. Pennebaker, "Optimal hardware and software arithmetic coding procedures for the Q-Coder," *IBM Journal of Research and Development*, vol. 32, no. 6, pp. 727–736, November 1988.

[7] D. S. Taubman and M. W. Marcellin, *JPEG2000, Image Compression Fundamentals, Standards and Practice*. Kluwer Academic Publishers Group Distribution Centre Post Office Box 322 3300 AH Dordrecht The Netherlands: Kluwer Academic Publishers, 2002.

[8] D. Marpe, H. Schwarz, G. Blättermann, G. Heising, and T. Wiegand, "Context-based adaptive binary arithmetic coding in JVT/H.26L," *Proc. IEEE International Conference on Image Processing (ICIP'02)*, vol. 2, pp. 513–516, September 2002.

[9] R. R. Osorio and J. D. Bruguera, "Arithmetic coding architecture for H.264/AVC CABAC compression system," in *Euromicro Symposium on Digital System Design*. Rennes, France: IEEE, August 31 - September 03 2004, pp. 62–69.

[10] M. S. Schmookler and K. J. Nowka, "Leading zero anticipation and detection-a comparison of methods," in *15th IEEE Symposium on Computer Arithmetic (Arith-15 2001)*. Vail, Colorado: IEEE Computer Society, June 2001.

[11] *Stratix Device Handbook*, Altera, Jan 2006. [Online]. Available: http://www.altera.com/literature/lit-stx.jsp

[12] *Stratix II Device Handbook*, Altera, April 2006. [Online]. Available: http://www.altera.com/literature/lit-stx2.jsp